OpenGL i GLSL (krótki kurs)

2017/2018

Przemysław Kiciak

Źródła

[1] *The OpenGL Graphics System: A Specification (Version 4.5)*, Khronos Group,
https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

[2] *The OpenGL Graphics System: A Specification (Version 4.6)*, Khronos Group,
https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[3] *The OpenGL Shading Language 4.5*, Khronos Group,
https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[4] *The OpenGL Shading Language 4.6*, Khronos Group,
https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[5] docs.gl.

[6] https://www.khronos.org/registry/OpenGL/extensions/.

[7] https://github.com/KhronosGroup/glslang.

[8] J. Kessenich, G. Sellers, D. Shreiner, *OpenGL Programming Guide*, nineth ed., Addison–Wesley, 2017.

[9] G. Sellers, R.S. Wright Jr, N. Haemel, *OpenGL Księga eksperta*, Helion, 2016.

[10] J. Ganczarski, *OpenGL Podstawy programowania grafiki 3D*, Helion, 2015.

[11] E. Luten, www.openglbook.com.

[12] https://en.wikipedia.org/wiki/OpenGL.

[13] https://en.wikipedia.org/wiki/OpenGL_Shading_Language.

[14] https://www.x.org/releases/X11R7.6/doc/.

[15] OpenGL Graphics with the X Window System (Version 1.4),
https://www.khronos.org/registry/OpenGL/specs/gl/glx1.4.pdf.

[16] M.J. Kilgard *The OpenGL Utility Toolkit (GLUT) Programming Interface. API Version 3*, Silicon Graphics, Inc., 1996.
https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf.

[17] freeglut.sourceforge.net/docs/api.php.

[18] `http://www.glfw.org/docs/latest/`.

[19] `https://github.com/skaslev/gl3w`.

[20] P. Kiciak, *Podstawy modelowania krzywych i powierzchni. Zastosowania w grafice komuterowej*, WNT, Warszawa, 2005.

[21] A.S. Glassner (ed.): *An introduction to ray tracing*. Academic Press, London 1989.

1. Wprowadzenie

Aplikacja OpenGL-a składa się z dwóch części: programu wykonywanego przez główny procesor komputera (CPU, *central processing unit*) i zbioru programów wykonywanych przez procesor graficzny (GPU, *graphics processing unit*). Pierwsza z tych części odpowiada za przygotowanie danych opisujących obiekty do narysowania oraz za współpracę ze środowiskiem (z systemem operacyjnym i podsystemem okien), zaś druga z nich realizuje poszczególne etapy tworzenia obrazu przez procesor graficzny. Części aplikacji OpenGL-a działające na CPU zazwyczaj pisze się w języku C lub jednym z jego potomków (np. C++). Części aplikacji działające na GPU można (choć nie jest to jedyna możliwość) napisać w języku GLSL.

Odrobina historii i ideologii

Historia technologicznego rozwoju grafiki komputerowej ze szczególnym uwzględnieniem dziejów standardu OpenGL jest pięknie opisana w (niestety niedokończonym) samouczku [11], którego lekturę polecam. Jednak odrobina historii jest potrzebna do poznania zasady działania aplikacji OpenGL-a, oraz do wyjaśnienia pewnych cech najnowszych wersji standardu. Zatem, bezpośrednim poprzednikiem OpenGL-a była biblioteka IRIS GL (*Integrated Raster Imaging System Graphical Library*) rozwijana przez firmę Silicon Graphics w latach 1982–1992. Biblioteka ta miała silne związki ze sprzętem i oprogramowaniem systemowym firmy Silicon Graphics.

Standard OpenGL 1.0 powstał w roku 1992 przez usunięcie tych związków; powołany został wtedy komitet nazwany *OpenGL Architecture Review Board*, w skrócie ARB, odpowiedzialny za opracowanie kolejnych wersji standardu. Wersje te, do OpenGL 2.1 włącznie (rok 2006), dawały coraz więcej możliwości, zachowując pełną zgodność wstecz; aplikacja napisana zgodnie z dowolną z tych specyfikacji mogła współpracować z *bibliotekami* OpenGL zgodnymi z wszystkimi późniejszymi wersjami.

Podstawowym sposobem działania aplikacji OpenGL-a w wersji 2.1 i wcześniejszych jest tryb natychmiastowy rysowania (*immediate mode*). Polega on na tym, że dane opisujące rysowane obiekty geometryczne (dokładniej: punkty lub wierzchołki łamanych lub wielokątów, z dodatkowymi atrybutami takimi jak kolor) są na bieżąco dostarczane przez aplikację działającą na CPU; procesor graficzny po otrzymaniu ostatniego wierzchołka łamanej natychmiast przystępuje do rysowania obiektu i nie zapamiętuje tych wierzchołków. W związku z tym,

chcąc wykonać następny rysunek (na przykład sceny, w której niewiele obiektów uległo zmianie), trzeba cały opis *wszystkich* obiektów ponownie przesłać do GPU. Oczywiście, to nie ma znaczenia, jeśli ma być wykonany tylko jeden obraz, i ma niewielkie znaczenie w świecie CAD, gdzie szybkość rysowania jest mało istotna. Ale ma to ogromne znaczenie w grach komputerowych, wyświetlających tworzone w czasie rzeczywistym animacje skomplikowanych scen w tempie kilkudziesięciu obrazów na sekundę.

Zmiany standardu OpenGL po wersji 2.1 zostały wymuszone przez wzrost mocy obliczeniowej oraz wprowadzenie możliwości programowania GPU; okazało się, że magistrala danych łącząca CPU z GPU stała się wąskim gardłem w przepływie danych. Rolą CPU, zamiast pracowitego dostarczania na bieżąco wszystkich danych do narysowania, stało się umieszczenie w pamięci GPU reprezentacji obiektów, z których składa się scena, „dyrygowanie" GPU, która przeprowadza znaczną część obliczeń i dostarczanie na bieżąco tylko tych danych, które w kolejnej klatce animacji muszą być zmienione. Jak sami to zobaczymy, GPU może generować obszerne dane na podstawie modelu opisanego przez niewiele parametrów przesyłanych magistralą przez CPU.

Obiekty geometryczne należy w celu narysowania odpowiednio oświetlić i dokonać rzutowania. Na opis obiektu oprócz kształtu składa się informacja o odbijaniu światła przez ten obiekt — w najprostszym przypadku jest to kolor całej powierzchni, ale może też być tekstura, czyli funkcja opisująca odbijanie światła przez poszczególne punkty tej powierzchni. Tekstury, a także opis przekształceń prowadzących do obliczenia rzutu obiektu na płaszczyznę obrazu, opisy źródeł światła, mgły itp., także przy pracy w trybie natychmiastowym, należało umieścić w pamięci GPU *przed* przystąpieniem do rysowania. Te dane (zwłaszcza opisy tekstur) są zazwyczaj dość duże i niekoniecznie zmieniają się w kolejnych klatkach animacji. Także wierzchołki figur geometrycznych można umieścić w tablicach w pamięci GPU, dzięki czemu zamiast przesyłać je dla każdej kolejnej klatki animacji, CPU może wydawać tylko polecenia rysowania obiektów opisanych przez zawartość tych tablic (np. wywołując procedurę `glDrawArrays`).
W nowym OpenGL-u (w wersji 3.0 i późniejszych) to jest w zasadzie jedyny sposób rysowania; wprawdzie tryb natychmiastowy jest dostępny na żądanie (aplikacja, tworząc tzw. kontekst OpenGL-a określa, czy będzie używać tego trybu), ale nie jest zalecany. Żegnaj, trybie natychmiastowy, było miło.

W standardzie OpenGL 2.0[1] pojawiła się *możliwość* (jeszcze nie konieczność) pisania programów wykonywanych przez GPU — w stworzonym w tym celu

[1] a wcześniej w rozszerzeniach standardu OpenGL 1.4

języku GLSL. Programy te, nazwane staropolskim słowem szadery (*shaders*), są kompilowane i przesyłane do GPU, gdzie zostają włączone do potoku przetwarzania grafiki. Początkowo istniały dwa rodzaje szaderów: szadery wierzchołków i szadery fragmentów, wywoływane odpowiednio dla każdego punktu przesłanego przez aplikację i dla każdego piksela zrasteryzowanego odcinka lub wielokąta. W szczególności szader wierzchołków mógł realizować rzutowanie punktu na płaszczyznę obrazu dowolną metodą, niekoniecznie jednym ze standardowych w OpenGL-u sposobów rzutowania. Szader fragmentów mógł realizować dowolny efekt końcowy wyglądu obiektu (np. związany z teksturą przezroczystości), także nieosiągalny w ramach standardowego zestawu operacji OpenGL-a.

Ten standardowy zestaw i tak jest dość duży; OpenGL udostępnia mnóstwo przełączników umożliwiających włączanie i wyłączanie poszczególnych efektów, a także wiele zmiennych, których wartości są parametrami we wzorach stosowanych do obliczania oświetlenia itp. Takie bogactwo ma tę wadę, że GPU w trakcie obliczeń musi wykonać dla każdego piksela wiele sprawdzeń, które możliwości są w danej chwili włączone, aby wykonać właściwe obliczenia. Dalsze rozszerzanie standardu stało się zbyt dużym ciężarem; przy tym chyba nie istnieje aplikacja korzystająca ze *wszystkich* możliwości OpenGL-a 2.1, a ich obecność zajmuje pamięć i spowalnia proces tworzenia obrazu.

Z tego powodu w nowym OpenGL-u nastąpiła radykalna zmiana: wszystkie procedury związane z natychmiastowym trybem rysowania zostały ze standardu usunięte, a dokładniej, zdeprecjonowane (*deprecated*), tj. oficjalnie uznane za pozbawione wartości i przestarzałe. Zdeprecjonowane procedury zostały usunięte z tzw. profilu podstawowego OpenGL-a (*core profile*). Aplikacja może z nich korzystać po zadeklarowaniu, że będzie[2]. Zdeprecjonowane zostały też wszystkie standardowe procedury wprowadzające macierze przekształceń wykonywanych w celu rzutowania obiektów i wszystkie procedury wprowadzające parametry oświetlenia; zadania przekształcania i oświetlania obiektów w nowym OpenGL-u mają wykonywać szadery, których dostarczenie przez aplikację stało się odtąd obowiązkowe.

W roku 2006 opiekę nad standardem OpenGL i jego rozwojem objęło konsorcjum Khronos Group, zrzeszające głównych producentów sprzętu i oprogramowania. Opublikowane wersje od 3.0 do 3.3 miały na celu umożliwienie wykorzystania

[2]Procedury starego OpenGL-a zostały przeniesione do tzw. profilu zgodności (*compatibility profile*). Aplikacja, tworząc kontekst OpenGL-a, może podać parametry deklarujące używanie profilu zgodności.

starszego sprzętu, natomiast wersje 4.0 i dalsze[3] wymagają sprzętu nowszego, który w chwili opublikowania specyfikacji 4.0 był pieśnią przyszłości, zaś obecnie jest powszechnie dostępny, choć nie w każdym komputerze — nie ma go np. w większości produkowanych obecnie laptopów nie przeznaczonych do grania[4]. Nowe wersje zapewniają zgodność wstecz, do wersji 3.0 włącznie. Nowością w OpenGL-u 3.2 były szadery geometrii, zaś w wersji 4.0 pojawiły się szadery rozdrabniania; jedne i drugie znacznie poszerzają możliwości programowania GPU, przy czym nie ma obowiązku ich używania. W wersji 4.3 doszły szadery obliczeniowe, które służą do wykonywania dowolnych obliczeń przez GPU w sposób masywnie równoległy[5]. Obliczenia te mogą, ale nie muszą, być związane z tworzeniem obrazu i szadery obliczeniowe nie są „wmontowywane" w potok przetwarzania grafiki.

Nazwa standardu, OpenGL, od samego początku oznaczała możliwość jego rozszerzania. Z tej możliwości korzystają producenci. Poszczególne rozszerzenia (*extensions*) mają nazwy, będące (dosyć długimi) napisami. Aplikacja może zbadać, czy biblioteka OpenGL, z którą aplikacja współpracuje w danej chwili, obsługuje dane rozszerzenie i jeśli tak, to może uzyskać dostęp do odpowiedniej procedury, a jeśli nie, to musi albo zrezygnować z efektów wytwarzanych przez to rozszerzenie, albo korzystać ze swoich podprogramów wytwarzających te efekty w jakiś inny sposób, zabierający na przykład więcej czasu obliczeń. Lista rozszerzeń dostępnych na danym komputerze może być też wyświetlona przez odpowiedni program stanowiący część instalacji OpenGL-a[6]. Poszczególne rozszerzenia mogą być włączane do kolejnych wersji standardu OpenGL i wtedy stają się jego integralną częścią.

Podstawowym ograniczeniem wszystkich wersji OpenGL-a jest jednowątkowość; kontekst OpenGL-a może być związany tylko z jednym wątkiem obliczeniowym programu działającego na CPU, co stanowi pewne ograniczenie dla programistów. Dlatego w roku 2015 pojawił się znoszący to ograniczenie następca standardu OpenGL, nazwany Vulkan. Programowanie aplikacji w tym standardzie jest bardziej skomplikowane niż pisanie aplikacji OpenGL-a, choć oba standardy mają

[3] Najnowsza specyfikacja w chwili pisania tego tekstu ma numer 4.6 [2], [4], przy czym dla mnie podstawą była specyfikacja 4.5 [1], [3], realizowana przez dostępny mi sprzęt.

[4] Z tego powodu wersje 3.3 i 4.0 zostały opublikowane jednocześnie. Bywa też tak, że procesor grafiki zgodny w chwili zakupu z pewną specyfikacją, np. 4.3, może obsługiwać nowszą wersję po zainstalowaniu nowych sterowników.

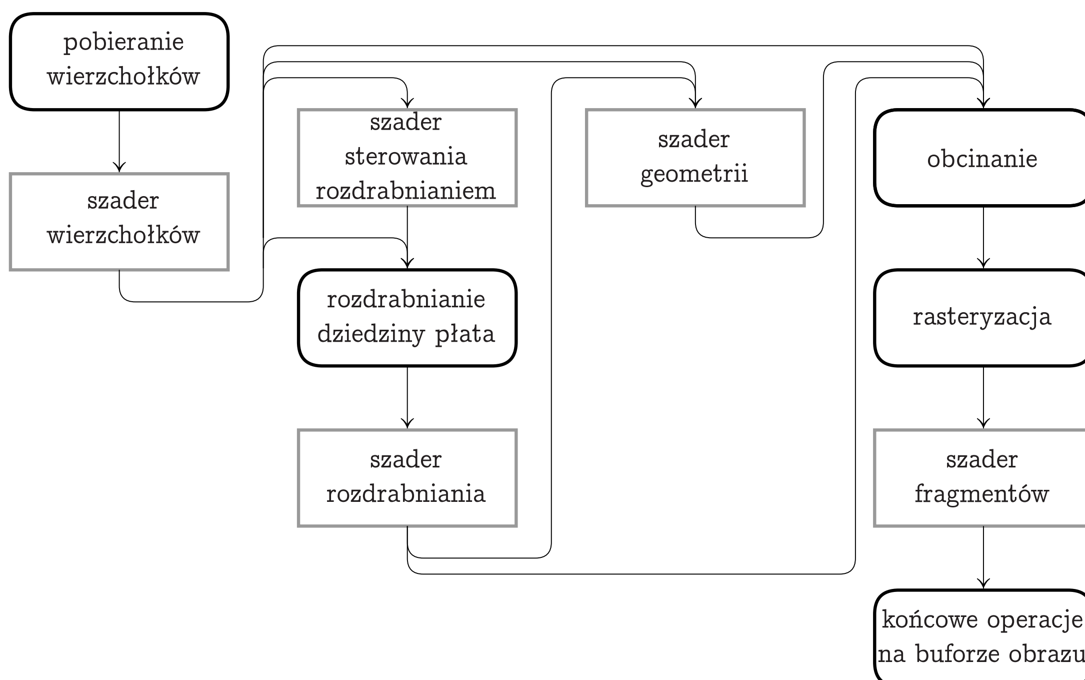
[5] Obecnie nawet tanie GPU mają co najmniej kilkadziesiąt procesorów, a te najbardziej wypasione mają ich kilka tysięcy. Komu procesor za mniej niż złotówkę? Tylko trzeba ich kupić dużo naraz ...

[6] Na przykład odpowiedni program instalowany razem ze sterownikami firmy NVIDIA nazywa się `nvidia-settings`.

analogiczne elementy, takie jak potok przetwarzania grafiki, obiekty w pamięci GPU, szadery itd. Być może standard Vulkan przejmie kiedyś wiodącą rolę w grafice, choć intencją Khronos Group jest współistnienie i dalszy rozwój obu standardów. Ale Vulkanem tu się nie zajmiemy.

Potok przetwarzania grafiki

Każdy rysowany obiekt elementarny, tzw. prymityw, jest reprezentowany przez skończony ciąg wierzchołków. Prymityw może być ciągiem osobnych punktów, ciągiem osobnych odcinków, łamaną (otwartą lub zamkniętą), ciągiem osobnych trójkątów, taśmą trójkątową, wachlarzem trójkątów lub płatem. Wierzchołki są umieszczone w pamięci GPU, w tablicy nazywanej buforem wierzchołków (*vertex buffer*), skąd na polecenie aplikacji są wprowadzane do potoku przetwarzania grafiki (*rendering pipeline*), co uruchamia proces rysowania.



Rysunek 1.1: Uproszczony schemat potoku przetwarzania grafiki OpenGL

Rysunek 1.1 przedstawia schemat przetwarzania danych przez GPU podczas rysowania; strzałki pokazują możliwe drogi przepływu danych między kolejnymi etapami obliczeń. Szare prostokątne ramki oznaczają *programowalne* etapy tego przetwarzania, czyli miejsca, w których obliczenia są wykonywane przez poszczególne szadery. Pozostałe etapy, uwidocznione w czarnych ramkach, są w znacznym stopniu *konfigurowalne* — ich przebiegiem sterują dane dostarczone przez aplikację i ustawione przez nią parametry i przełączniki. Podany niżej opis

poszczególnych etapów jest przeglądowy i bardzo uproszczony; miejsce na szczegóły jest w dalszych rozdziałach.

Etap pobierania wierzchołków (*vertex fetching*) polega na skompletowaniu dla każdego wierzchołka jego atrybutów w celu przekazania ich na wejście szadera wierzchołków; wśród atrybutów prawie zawsze jest wektor współrzędnych położenia wierzchołka i pewne dane identyfikacyjne dla wierzchołka. Ponadto aplikacja może określić atrybuty dodatkowe, takie jak wektor reprezentujący kolor, wektor normalny i wektor współrzędnych tekstury. Wierzchołek może mieć wiele atrybutów dodatkowych, może też nie mieć ani jednego. Interpretacja atrybutów wierzchołka należy do szaderów, które mogą same produkować wartości potrzebnych w dalszych etapach atrybutów, na przykład koloru, położenia, wielkości kropki rysowanej jako obraz wierzchołka, danych wykorzystywanych do obcinania itd.

Szader wierzchołków (*vertex shader*) ma za zadanie wykonać pewne obliczenie dla pojedynczego wierzchołka. Jego wynikiem jest przekształcony wierzchołek, który będzie przetwarzany dalej, i jego atrybuty. W najprostszym przypadku wierzchołek przekształcony jest kopią wierzchołka dostarczonego na wejście; szader wierzchołków może też dokonać przekształcenia, które obszar, który ma być widoczny na obrazie (bryłę widzenia, *view volume*), przekształci na opisaną dalej kostkę standardową. Przekształcenie to wyznaczy rzutowanie obiektów (np. równoległe lub perspektywiczne), efektywnie przeprowadzane na etapie rasteryzacji.

Jeśli w potoku przetwarzania grafiki są obecne szadery rozdrabniania (*tesselation shaders*), to wierzchołki przetworzone przez szader wierzchołków są zbierane w płaty poddawane rozdrabnianiu, czyli podziałowi na kawałki. Dziedzina płata może być trójkątem lub kwadratem. Zadaniem szadera sterowania rozdrabnianiem (*tesselation control shader*) jest określenie, jak drobny ma to być podział. Umożliwia to dostosowanie podziału do wielkości obiektu na obrazie. Jeśli obraz obiektu jest bardzo mały (np. wielkości kilku pikseli), to podział może być zgrubny; jeśli zaś obraz obiektu jest duży, to podział na wiele drobnych kawałków może być konieczny do uzyskania wystarczająco dobrej dokładności obrazu. W etapie rozdrabniania dziedziny płata, na podstawie parametrów podanych przez szader sterowania rozdrabnianiem, są generowane odcinki lub trójkąty będące kawałkami dziedziny płata. Szader rozdrabniania (*tesselation evaluation shader*) dla każdego wierzchołka takiego odcinka lub trójkąta ma skonstruować punkt w przestrzeni (tj. wektor współrzędnych jednorodnych) będący wierzchołkiem odpowiedniego fragmentu płata. Szadery rozdrabniania mają dostęp do tablicy

wierzchołków prymitywu (płata), przy czym liczba tych wierzchołków jest ograniczona (zależnie od implementacji, np. do 32). Jeśli wierzchołki płata mają dodatkowe atrybuty (np. kolor lub wektor współrzędnych tekstury), to szader rozdrabniania ma za zadanie dokonanie odpowiedniej interpolacji (lub obliczenia w inny sposób) tych atrybutów dla wygenerowanych punktów w przestrzeni.

Kolejny opcjonalny etap obliczeń wykonuje szader geometrii (*geometry shader*), który wykonuje obliczenia dla pojedynczych punktów, odcinków lub trójkątów wprowadzonych bezpośrednio do potoku przetwarzania grafiki lub otrzymanych z podzielenia łamanej lub taśmy trójkątowej, a także w wyniku rozdrabniania płata. Jeśli na przykład przetwarzane dane reprezentują trójkąt (o wierzchołkach dostarczonych przez szader wierzchołków lub rozdrabniania), to szader geometrii (inaczej niż szader wierzchołków) ma dostęp do wszystkich wierzchołków tego trójkąta. Dzięki temu szader geometrii może wygenerować wektor normalny płaszczyzny trójkąta, który będzie następnie użyty do obliczenia koloru na podstawie przyjętego modelu oświetlenia (alternatywą jest dostarczanie wektorów normalnych jako atrybutów wierzchołków). Szader geometrii może dodatkowo rozdrobnić dany na wejściu odcinek lub trójkąt, produkując z niego łamane lub taśmy trójkątowe. Jeśli żaden szader opisany wcześniej nie dokonał rzutowania (a ściślej przekształcenia wspomnianego w opisie szadera wierzchołków), to powinien to zrobić szader geometrii.

Etapy (stałe i programowalne) opisane wyżej składają się na część przednią potoku przetwarzania grafiki. Dane wytworzone w ostatnim z tych etapów opisują punkty, odcinki lub trójkąty, które są następnie obcinane. Obszar przestrzeni $\mathbb{R}^3$, którego zawartość znajdzie się na obrazie (po przekształceniu wykonanym przez szadery części przedniej) jest standardową kostką trójwymiarową, $[-1, 1]^3$. Obcinanie polega na znalezieniu części wspólnej prymitywu ze standardową kostką, przy czym oprócz sześciu płaszczyzn ścian tej kostki aplikacja może wprowadzić dodatkowe płaszczyzny obcinające; odrzuca się wszystkie części prymitywu położone po „niewłaściwej” stronie każdej z tych płaszczyzn.

Każdy prymityw i to, co zostaje z niego po obcięciu, jest figurą wypukłą: punktem, odcinkiem albo wielokątem wypukłym. Współrzędne x, y wierzchołków obciętego prymitywu są przekształcane tak, aby odwzorować kwadrat $[-1, 1]^2$ (ścianę standardowej kostki) na prostokąt o wymiarach klatki (*viewport*) podanych w pikselach, po czym następuje rasteryzacja — wyznaczenie pikseli, które odpowiadają rzutom punktów obiektów. Dla każdego piksela wyznaczana jest głębokość (tj. współrzędna z punktu obiektu) oraz atrybuty punktu powstałe przez interpolację dostarczonych atrybutów wierzchołków prymitywu.

Dane te są podawane na wejście szadera fragmentów (*fragment shader*), który jest włączony do części tylnej potoku przetwarzania grafiki. Na ich podstawie szader ma obliczyć kolor, który będzie przypisany (albo nie) odpowiedniemu pikselowi (elementowi bufora obrazu, *image buffer*). Szader fragmentów może obliczać kolor na podstawie otrzymanych na wejściu atrybutów a także tekstur i informacji o źródłach światła umieszczonych zawczasu w pamięci GPU.

Ostateczne przypisanie koloru piksela zależy od testu nożyczek (*scissor test*), testu szablonu (wykorzystującego bufor szablonu, *stencil buffer*, w którym można określić obszar o dowolnym kształcie zabroniony dla rysowania), testu widoczności (wykonywanego przy użyciu bufora głębokości, *depth buffer*) oraz wybranej funkcji mieszającej (*blending function*), która określa ostateczny kolor na podstawie koloru podanego przez szader fragmentów i poprzedniej wartości piksela.

Programy szaderów

Szadery dla poszczególnych etapów przetwarzania grafiki po skompilowaniu łączy się w programy szaderów. Kompletny program przeznaczony do wbudowania w potok przetwarzania grafiki *musi* zawierać szader wierzchołków i *powinien* zawierać szader fragmentów — jeśli ten ostatni jest nieobecny, to program może być poprawny, ale wykonany przy jego użyciu obraz jest nieokreślony. Wyniki obliczeń takiego programu mogą jednak zostać zapamiętane i użyte jako dane dla innego programu szaderów, lub przesłane do pamięci CPU i przetwarzane dalej przez aplikację. Szadery rozdrabniania i geometrii mogą być w programie nieobecne, zaś obecność szaderów obliczeniowych w programie zawierającym szadery innych rodzajów jest zabroniona.[7]

Podstawowe dane wejściowe i wyniki obliczeń szaderów w potoku przetwarzania grafiki są przekazywane za pomocą tzw. zmiennych interfejsu, w tym zmiennych wbudowanych opisanych w specyfikacji języka GLSL i dodatkowych zmiennych (globalnych) o nazwach nadanych przez autora szaderów (te ostatnie są używane m.in. do przekazywania wartości dodatkowych atrybutów wierzchołków). Szader ma też dostęp do opisanych dalej zmiennych jednolitych i do tekstur zadeklarowanych w jego treści.

Aplikacja zazwyczaj tworzy wiele programów szaderów, przeznaczonych do rysowania różnych obiektów (np. jeden z nich ma wyświetlać krawędzie wielościanu, a inny ma rysować jego ściany jako poteksturowane i oświetlone

[7] Program może też składać się tylko z szaderów obliczeniowych i wtedy działa poza potokiem przetwarzania grafiki.

wielokąty). Przed przystąpieniem do rysowania należy wybrać odpowiedni program, wywołując procedurę `glUseProgram` z parametrem będącym identyfikatorem tego programu. Spowoduje to wmontowanie jego szaderów w potok przetwarzania grafiki (i wymontowanie szaderów nieobecnych w tym programie z odpowiednich miejsc potoku).

Źródła danych w potoku przetwarzania grafiki

Jak już wiemy, wierzchołki, których wprowadzenie do potoku przetwarzania grafiki uruchamia proces rysowania, muszą być zawczasu umieszczone w buforze wierzchołków (*vertex buffer*), utworzonym w pamięci GPU. Wierzchołek jest opisany przez współrzędne (kartezjańskie lub jednorodne) swojego położenia w przestrzeni oraz atrybuty dodatkowe, których liczba, w różnych zastosowaniach, może być od zera do kilkunastu. OpenGL umożliwia umieszczenie wszystkich atrybutów w jednym buforze (zawierającym tablicę struktur z polami przechowującymi poszczególne atrybuty) albo w osobnych buforach. Aby umożliwić skompletowanie wszystkich atrybutów w etapie pobierania wierzchołka, bufory te są rejestrowane w obiekcie tablicy wierzchołków (*vertex array object, VAO*), który zawiera dane umożliwiające odczytanie wszystkich zadeklarowanych przez aplikację atrybutów z właściwych miejsc. Każdy obiekt do narysowania powinien mieć swój VAO, zarejestrowany w kontekście OpenGL-a. Aby obiekt narysować, należy jego VAO przywiązać (*bind*), tj. uczynić bieżącym obiektem tablicy wierzchołków w kontekście, a następnie wywołać jedną z procedur rysowania (np. `glDrawArrays`), która sprawi, że potok przetwarzania grafiki przystąpi do pracy.

Dane opisujące przekształcenia geometryczne wierzchołków, inne ich atrybuty (np. domyślny kolor wszystkich wierzchołków pewnego obiektu), oświetlenie, tekstury, fonty itp., aplikacja musi również umieścić w odpowiednich buforach w pamięci GPU. Szadery mogą z nich czytać dane, mogą też w nich zapisywać wyniki swoich obliczeń w celu przekazania dalej — do innych szaderów albo do aplikacji. Każdy bufor po utworzeniu i wypełnieniu po raz pierwszy danymi ma określony sposób używania (na przykład tylko do odczytu przez szadery, do odczytu przez aplikację itd.), ma też ustaloną wielkość, której nie można zmienić.

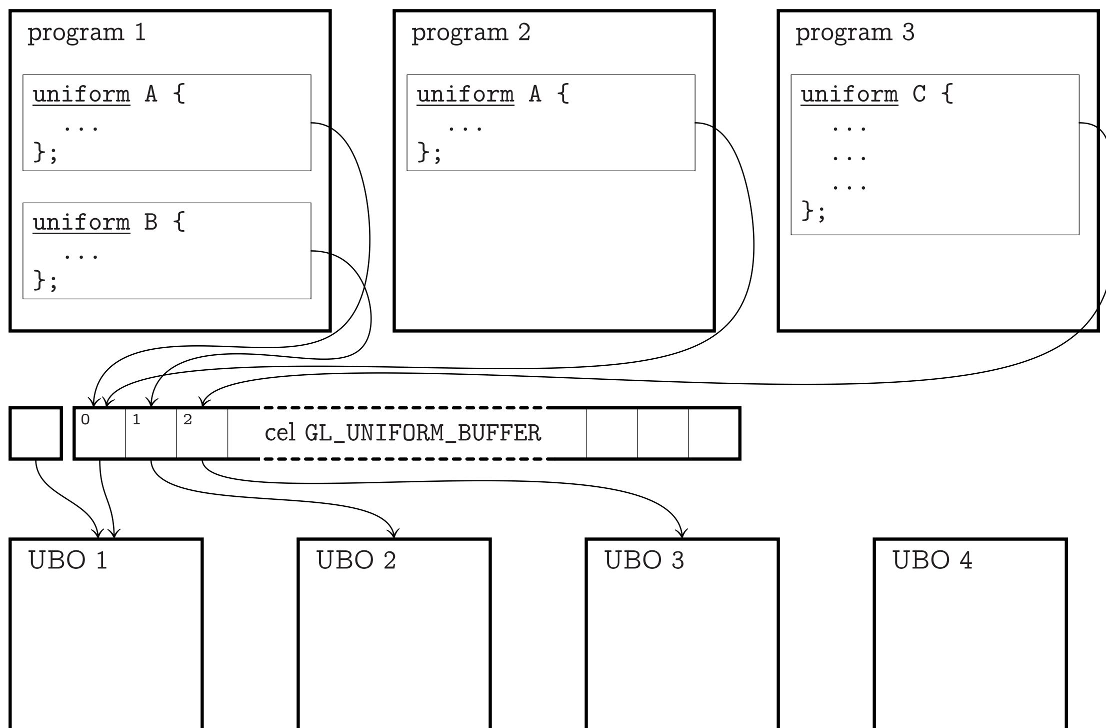
Zmienne jednolite (*uniform variables*[8]) są to zmienne globalne zadeklarowane w szaderach, którym wartości nadaje aplikacja. Szadery mogą tylko odczytywać

[8] Autorzy polskich książek lub przekładów jak dotąd nie radzili sobie z tym terminem na różne sposoby. Można na przykład przeczytać o „zmiennych uniform” (co jest kapitulacją) lub o „zmiennych jednorodnych” (co jest matematycznym nonsensem). Chyba tylko jeszcze nikt nie napisał o „zmiennych mundurowych”. Dobre i to.

ich wartości, *jednakowe dla wszystkich* szaderów (stąd nazwa); stwierdzenie to dotyczy zarówno składających się na program szaderów wmontowanych w różne etapy potoku przetwarzania grafiki, jak i wątki jednego szadera (np. wierzchołków lub fragmentów) działające równolegle na wielu procesorach GPU (oczywiście, aplikacja może nadać zmiennym jednolitym nowe wartości przed ponownym uruchomieniem potoku). Zmienne jednolite są stosowane do przechowywania macierzy przekształceń wierzchołków, opisów źródeł światła itd.

Obiekt bufora zmiennych jednolitych (*uniform buffer object, UBO*) jest buforem, który przechowuje zmienne jednolite opisane w bloku zmiennych jednolitych (*uniform block*) zadeklarowanym w programie szaderów. Aplikacja może utworzyć kilka takich obiektów, jednakowo zbudowanych, ale o różnych wartościach poszczególnych pól. W kontekście OpenGL-a występują tzw. cele (*targets*), do których przywiązywane są bufory, aby programy działające na GPU miały do nich dostęp. Obiekty buforów zmiennych jednolitych przywiązuje się do celu nazwanego `GL_UNIFORM_BUFFER`. Ten cel jest *indeksowany*; mianowicie zawiera on tablicę punktów dowiązania. Wywołanie odpowiedniej procedury wiąże wskazany UBO z punktem dowiązania o podanym numerze (zrywając poprzednie wiązanie). Bloki zmiennych jednolitych zadeklarowane w programach szaderów są skojarzone z konkretnymi punktami dowiązania w celu `GL_UNIFORM_BUFFER`. Dzięki temu różne programy szaderów mogą mieć dostęp do *tych samych* zmiennych jednolitych w UBO; na przykład wspólne dla różnych programów mogą być macierze przekształceń obiektów — wpisanie nowych współczynników macierzy do UBO (albo dowiązanie UBO z innymi współczynnikami do odpowiedniego punktu) spowoduje, że zmiana będzie widoczna dla wszystkich zainteresowanych programów. Inny przykład zastosowania to UBO zawierający wzorce czcionek (fonty) dla różnych krojów pisma; zmiana kroju pisma dla wyświetlanego tekstu polega wtedy na dowiązaniu UBO z innym fontem do odpowiedniego punktu. Zasadę połączeń bloków zmiennych jednolitych w programach szaderów z UBO za pośrednictwem tablicy punktów dowiązania ilustruje rysunek 1.2. Kwadrat z lewej strony narysowanej tablicy punktów dowiązania symbolizuje główny cel `GL_UNIFORM_BUFFER`; program wykonywany przez CPU ma za pośrednictwem tego celu dostęp do przywiązanego w tym momencie bufora (w tym przypadku do UBO 1), którego zawartość (przez punkt dowiązania `0`) jest też widoczna dla programów 1 i 2 jako blok zmiennych jednolitych `A`.

Zmienne jednolite mogą też być zadeklarowane „statycznie” w szaderach. Taka zmienna jednolita jest widoczna dla wszystkich szaderów wchodzących w skład jednego programu i niewidoczna dla innych programów — znajduje się ona w tzw. domyślnym bloku zmiennych jednolitych programu.



Rysunek 1.2: Wiązanie UBO z blokami zmiennych jednolitych

Szadery mogą korzystać z buforów magazynowych (*storage buffers*, jest też skrót SSBO, czyli *shader storage buffer object*, obiekt bufora magazynowego); od zmiennych jednolitych różnią się one tym, że szadery *mogą* do nich wpisywać wyniki swoich obliczeń. Jeden z możliwych scenariuszy działania aplikacji jest taki, że pewien program szaderów (np. składający się z szadera obliczeniowego) wykonuje obliczenia pomocnicze przed rysowaniem, po czym inny program szaderów, wbudowany w potok przetwarzania grafiki, wykonuje obraz, korzystając przy tym z danych wpisanych przez pierwszy program do bufora magazynowego. Mechanizm wiązania bloków magazynowych zadeklarowanych w programach szaderów z buforami magazynowymi jest podobny do opisanego wcześniej mechanizmu dla bloków zmiennych jednolitych, ale w tym celu wykorzystuje się tablicę punktów dowiązania w celu nazwanym `GL_SHADER_STORAGE_BUFFER`.

Bufory magazynowe podlegają mniejszym ograniczeniom niż zmienne jednolite; mogą one być znacznie większe i dopuszczają bardziej upakowane rozmieszczenie danych w pamięci. Z drugiej strony, dostęp do danych w buforze magazynowym może zabierać nieco więcej czasu.

Ważnym elementem w tworzeniu obrazu są tekstury; są one zazwyczaj

reprezentowane przez tablice wartości w pewnych punktach[9]. Element takiej tablicy, czyli teksel, może przechowywać kolor lub inny parametr realizowanego przez szader fragmentów modelu oddziaływania światła z powierzchnią. Przed rysowaniem obiektu z nałożoną teksturą aplikacja wiąże odpowiednie obiekty tekstury, czyniąc je bieżącymi w kontekście i dostępnymi dla szaderów. Dostęp do tekstury odbywa się za pomocą ewaluatora tekstury (*sampler*), czyli obiektu przechowującego dodatkowe informacje dla procedur obliczających wartości tekstury. Poszczególne procedury podają „surowe" wartości tekseli lub dokonują interpolacji i filtrowania w celu poprawienia jakości obrazu.

Szadery mogą mieć też dostęp do bufora obrazu i do związanych z nim buforów dodatkowych (*ancillary buffers*), czyli wspomnianych wcześniej buforów głębokości i szablonu.

Następne pięć rozdziałów zawiera dość dużo informacji „technicznych". Czytelnik Niecierpliwy, który dopiero zaczyna swoje doświadczenia z OpenGL-em, może przejrzeć je pobieżnie, aby w trakcie studiowania aplikacji w rozdziale 7 i dalszych zaglądać do tych rozdziałów w celu uzupełnienia wiedzy. Jeśli jednak trzeba dopiero zainstalować OpenGL-a, to może warto na początek dokładniej przeczytać rozdział 2.

[9] Ale można też określić teksturę proceduralną — jest to funkcja opisana dowolnym wzorem, której wartości obliczy szader fragmentów lub wywołany przezeń podprogram.

2. Biblioteki i pliki nagłówkowe OpenGL-a

Aplikacja napisana w *standardzie* OpenGL musi być połączona z odpowiednią *biblioteką* OpenGL (w Linuksie libGL.so); biblioteki są w pakietach dystrybucji systemu operacyjnego (jeśli nie są zainstalowane domyślnie, to trzeba doinstalować odpowiedni pakiet) lub też są instalowane razem ze sterownikiem procesora graficznego dostarczanym przez jego producenta.

Oprócz biblioteki powinniśmy też mieć odpowiednie pliki nagłówkowe; podstawowy plik ma nazwę GL/gl.h i są w nim prototypy procedur podstawowych oraz procedur *starego* OpenGL-a, przeznaczonych do wykonywania obrazów w trybie natychmiastowym. Procedury nowego OpenGL-a są opisane w pliku GL/glext.h. Włączenie obu plików daje dostęp do procedur z *obu* rodzajów OpenGL-a — starego i nowego. Pisząc program, należy uważać, aby konsekwentnie używać tylko starego albo tylko nowego zestawu procedur, bo od mieszania głowa naprawdę boli. Dlatego, pisząc program *w starym stylu*, należy włączyć do aplikacji plik GL/gl.h[1].

Pisząc program *w nowym stylu*, zamiast pary plików GL/gl.h i GL/glext.h, lepiej jest włączyć plik GL/glcorearb.h. Zawiera on tylko makrodefinicje, typy danych i prototypy procedur nowego OpenGL-a. Niezależnie od tego, czy używamy pary plików GL/gl.h i GL/glext.h, czy też pliku GL/glcorearb.h, napisanie samych dyrektyw #include *nie zadziała*: kompilator nie będzie „widział” prototypów procedur nowego OpenGL-a.

Aby uwidocznić prototypy procedur nowego OpenGL-a, możemy na początku programu (po innych potrzebnych dyrektywach, np. #include <stdlib.h>, #include <stdio.h>, #include <math.h> itd.) napisć

```
#define GL_GLEXT_PROTOTYPES
#include <GL/gl.h>
#include <GL/glext.h>
```

albo

```
#define GL_GLEXT_PROTOTYPES
#include <GL/glcorearb.h>
```

[1] a po nim ewentualnie GL/glu.h; jest to plik nagłówkowy dodatkowej biblioteki libGLU.so, zawierającej wiele procedur ułatwiających pracę w trybie natychmiastowym (np. rysowanie kwadryk i powierzchni B-sklejanych). Niektóre procedury z tej biblioteki przydają się też w programach napisanych w nowym stylu.

Jeśli zainstalowana biblioteka OpenGL zawiera procedury wywoływane przez aplikację, to to wystarczy, aby można ją było skompilować i uruchomić. Ale skompilowana aplikacja może być przeniesiona do innego systemu, wyposażonego w bibliotekę innej wersji standardu i w szczególności zawierającej inny zestaw procedur realizujących rozszerzenia standardu. Dlatego przygotowanie współpracy aplikacji z biblioteką OpenGL powinno się odbyć bardziej skomplikowanym sposobem.

Oto ten sposób: w katalogu z bibliotekami powinna być też biblioteka, która służy do zapewnienia współpracy aplikacji z systemem okien. Dla systemu X Window zazwyczaj dodatkowa biblioteka nie jest potrzebna, bo odpowiednie procedury są obecne w `libGL.so`[2]. Plik nagłówkowy zawierający prototypy procedur obsługujących współpracę z systemem okien ma nazwę `GL/glx.h`. Są w nim opisane procedury, które tworzą kontekst OpenGL-a[3] i przywiązują go do okna aplikacji. Oprócz tego jest tam procedura, która dla podanej *nazwy* dowolnej procedury OpenGL-a (łańcucha ASCIIZ) podaje *adres* tej procedury. Dla systemu X Window procedura podająca adresy ma nazwę `glXGetProcAddress`[4]. Podany adres może być pusty (tj. `NULL`), jeśli implementacja OpenGL-a nie zawiera procedury o takiej nazwie, bo na przykład taka procedura została określona w wersji OpenGL-a późniejszej niż wersja dostępna w danym komputerze, albo jest to procedura niestandardowa, należąca do rozszerzenia standardu dokonanego przez producenta innego sprzętu niż ten, który mamy[5].

Zatem, aplikacja napisana w sposób przenośny powinna zadeklarować odpowiednie zmienne, będące wskaźnikami do procedur OpenGL-a i na początku działania „powyciągać" i przypisać tym zmiennym adresy potrzebnych procedur. Nazwy tych zmiennych wskaźnikowych powinny być identyczne z nazwami odpowiednich procedur OpenGL-a. W ten sposób na przykład instrukcja

```
glCompileShader ( shader_id );
```

[2]Istnieje osobna biblioteka o nazwie `libGLX.so` z tymi procedurami, która może być potrzebna, jeśli podstawowa biblioteka OpenGL-a tych procedur nie zawiera, bo na przykład jest to nietypowa implementacja standardu.

[3]Informacje o kontekstach są w następnym rozdziale.

[4]Rozszerzeniem specyfikacji GLX 1.3 jest procedura `glXGetProcAddressARB`, w obecnej wersji 1.4 doszła `glXGetProcAddress` (w mojej instalacji *ta sama* procedura ma dwie nazwy). To jest ewidentny ślad po włączeniu rozszerzenia do następnej wersji standardu.

[5]Jeden z podstawowych mechanizmów rozwoju standardu OpenGL polega na tym, że producenci wprowadzają swoje rozszerzenia, które następnie, po podjęciu odpowiedniej decyzji przez Khronos Group, zostają włączone (albo nie) do kolejnej wersji standardu. Zresztą, rozszerzenia wprowadzone przez poszczególnych producentów, nawet jeśli nie są w standardzie, często są odtwarzane i udostępniane przez innych producentów, którzy nie chcą być gorsi.

ma identyczne skutki w obu przypadkach: zarówno wtedy, gdy identyfikator glCompileShader jest nazwą procedury dołączonej bezpośrednio do programu, jak i wtedy, gdy jest to nazwa zmiennej wskaźnikowej, której została nadana wartość otrzymana w odpowiedzi na pytanie o adres czegoś co się nazywa "glCompileShader".

Jeśli *nie poprzedzimy* dyrektyw #include makrodefinicją #define GL_GLEXT_PROTOTYPES, to *zamiast* prototypów procedur nowego OpenGL-a kompilator do swoich tablic wprowadzi deklaracje typów wskaźnikowych do procedur o odpowiednich nagłówkach. Przykładowo, dla glCompileShader jest to (w uproszczeniu)

```
typedef void (*PFNGLCOMPILESHADERPROC) (GLuint shader);
```

W programie powinniśmy napisać deklarację zmiennej

```
PFNGLCOMPILESHADERPROC glCompileShader;
```

i tej zmiennej należy przypisać odpowiedni adres, co w aplikacji systemu X Window wykonuje instrukcja

```
  glCompileShader = glXGetProcAddress ( "glCompileShader" );
```

Bibioteki pomocnicze libGLEW i gl3w

Procedur OpenGL-a jest kilkaset, co sprawia, że pisanie deklaracji potrzebnych zmiennych i kodu, który pracowicie „wyciąga” adresy tych procedur, zabiera czas, który lepiej byłoby spędzić w przyjemniejszy sposób. Można użyć jednej z pomocniczych bibliotek, libGLEW.so lub gl3w, które wykonują tę pracę. Dodatkową zaletą tego sposobu jest ukrycie przed aplikacją zależności systemowych: bibliotek tych używamy tak samo w systemie Unix/Linux+X Window, jak i w Windowsach.

Aby użyć biblioteki libGLEW, *zamiast* plików GL/gl.h i GL/glext.h albo GL/glcorearb.h, do programu włączamy plik GL/glew.h[6]. Biblioteka libGLEW zawiera (globalne) zmienne wskaźnikowe do procedur OpenGL-a, zgodne z opisem

[6] Uwaga: Jeśli korzystamy z libGLEW, to dyrektywę #include <GL/glew.h> dajemy we *wszystkich* plikach źródłowych (w C) naszej aplikacji — w ten sposób wszystkie wywołania procedur OpenGL-a będą wykonywane za pośrednictwem zmiennych wskaźnikowych w bibliotece libGLEW, które dzięki temu będą widoczne podczas kompilacji. To samo dotyczy pliku nagłówkowego biblioteki gl3w.

podanym wyżej. Na początku działania (*po* utworzeniu okna i kontekstu OpenGL-a, co jest opisane w następnym rozdziale) aplikacja wykonuje instrukcję

```
if ( (ec = glewInit ()) != GLEW_OK ) {
  fprintf ( stderr, "Error: %s\n", glewGetErrorString ( ec ) );
  exit ( 1 );
}
```

(zmienna `ec` ma być typu int). Procedura `glewInit` przypisuje odpowiednie adresy zmiennym i w razie niepowodzenia informuje o tym aplikację, zwracając kod określający, co poszło źle. Procedura `glewGetErrorString` tłumaczy ten kod na angielski.

Uwaga: Biblioteka `libGLEW` zainstalowana na moim sprzęcie ma jakiś błąd, który objawia się wywaleniem programu podczas inicjalizacji. Aby tego uniknąć, trzeba podczas tworzenia kontekstu OpenGL-a dla aplikacji (o czym traktuje następny rozdział) podać parametry żądające *starej* wersji, tj. 2.1, z opcją zgodności w przód[7]. Nie ma tego problemu z opisaną niżej biblioteką `gl3w`, która wydaje mi się ogólnie lepiej dopracowana niż `libGLEW`.

Sposób użycia biblioteki `gl3w` jest podobny, z tym, że w odróżnieniu od biblioteki `libGLEW`, dostępnej w pakiecie[8], musiałem ją ręcznie skompilować i zainstalować, co okazało się dość proste; poza plikiem nagłówkowym kod źródłowy składa się z jednego pliku napisanego w C. Ze strony projektu [19] trzeba ściągnąć plik `gl3w-master.zip` i po jego rozpakowaniu wydać następujące polecenia:

```
cd gl3w-master
python gl3w_gen.py
cd src
```

Dalej mamy dwie możliwości: wygenerowany przez powyższe polecenia plik źródłowy `gl3w.c` możemy skompilować w zwykły sposób, otrzymując pojedynczy plik obiektowy, `gl3w.o`, który dodajemy do aplikacji na etapie łączenia. Do listy bibliotek dołączanych do aplikacji należy dodać bibliotekę `libdl`, która zawiera procedury `dlopen` i `dlclose` wywoływane przez procedury w pliku `gl3w.o` (kompilując plik z procedurą `main` aplikacji, trzeba podać kompilatorowi opcję `-ldl`). Od nas zależy, gdzie na dysku umieścimy ten plik oraz plik nagłówkowy

[7] W bibliotekach zapewniających współpracę aplikacji i OpenGL-a z systemem okien makrodefinicja dla tej opcji ma różne nazwy, np. `GLUT_FORWARD_COMPATIBLE`, `GLFW_OPENGL_FORWARD_COMPAT` lub `GLX_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB`.

[8] w używanej przeze mnie dystrybucji Linuxa

gl3w.h, który musi być włączony przez źródła aplikacji dyrektywą #include. Druga możliwość to utworzenie biblioteki łączonej dynamicznie (*shared object*). Wykonują to polecenia

```
gcc -c -Wall -ansi -pedantic -O2 -fPIC -I../include gl3w.c -o gl3w.o
gcc -shared -Wl,-soname,libgl3w.so -o libgl3w.so.1 gl3w.o
```

Potem trzeba umieścić pliki nagłówkowe i bibliotekę w katalogach, w których będą dostępne dla kompilatora (czyli w /usr/include/GL oraz /usr/lib64, albo gdzieś we własnym katalogu domowym, jeśli nie mamy uprawnień administratora). Korzystanie z biblioteki współdzielonej ma tę zaletę, że plik binarny aplikacji jest mniejszy niż plik z procedurami dołączonymi statycznie. Ale uruchomienie tak przygotowanej aplikacji na innym komputerze wymaga, aby biblioteka współdzielona była tam zainstalowana, na co nie zawsze można liczyć.

W plikach źródłowych aplikacji należy *zamiast* #include <GL/glew.h> napisać dyrektywę #include "gl3w.h" albo #include <GL/gl3w.h> (zasady umieszczania tych dyrektyw w kodzie aplikacji są takie same jak dla biblioteki libGLEW). Po utworzeniu okna i kontekstu OpenGL-a aplikacja powinna wykonać instrukcje

```
if ( gl3wInit () ) {
  fprintf ( stderr, "Error: gl3w initialisation failure\n" );
  exit ( 1 );
}
if ( !gl3wIsSupported ( 4, 2 ) ) {
  fprintf ( stderr, "Error: OpenGL version 4.2 not supported\n" );
  exit ( 1 );
}
```

Mamy tu sprawdzenie, czy implementacja OpenGL-a udostępnia wystarczającą wersję standardu OpenGL[9]. W liście bibliotek dołączanych do aplikacji zamiast biblioteki libGLEW trzeba podać libgl3w (na co właściwe miejsce jest w pliku Makefile) i tyle, a jeśli to nie wystarczy, to resztę proszę doczytać w dokumentacji.

Uwaga: Jednocześnie z plikami źródłowymi gl3w.c i gl3w.h program gl3w_gen.py generuje również plik glcorearb.h. Warto ten plik skopiować do katalogu /usr/include/GL (nadpisując plik glcorearb.h zainstalowany razem ze sterownikiem GPU, ale do tego potrzebne są uprawnienia administratora),

[9] To autor aplikacji decyduje, która wersja jest wystarczająca.

ponieważ zawiera on makrodefinicje i nagłówki procedur związanych z większą liczbą rozszerzeń, w tym takich, które weszły do następnej wersji standardu OpenGL. Bywa tak, że sterownik obsługuje rozszerzenia nieopisane w instalowanym razem z nim pliku `glcorearb.h`.

Dla wygody warto plik nagłówkowy biblioteki `libGLEW` albo `gl3w` włączyć do źródeł aplikacji za pośrednictwem dodatkowego pliku takiego jak na listingu 2.1, który łatwo można dostosować do lokalnej instalacji OpenGL-a (także wtedy, gdy nie mamy prawa pisania w katalogu `/usr/include`). Zmiana biblioteki używanej do pobierania adresów procedur OpenGL-a wymaga tylko zakomentowania lub odkomentowania makrodefinicji w pierwszej linii oraz odpowiedniej modyfikacji plików `Makefile`.

Listing 2.1: Plik `openglheader.h`

C

```c
1: #define USE_GL3W
2:
3: #ifdef USE_GL3W
4: #include <GL/gl3w.h>
5: #define ENABLE_SPIRV  /* opcja dla zaawansowanych, można nie używać */
6: #else
7: #include <GL/glew.h>
8: #define USE_GLEW
9: #endif
```

Listing 2.2 przedstawia procedurę (umieszczoną w pliku `utilities.c`, zawierającym różne procedury pomocnicze opisane w rozdziałach 4–6), która

Listing 2.2: Procedura pobierania adresów

C

```c
1: void GetGLProcAddresses ( void )
2: {
3: #ifdef USE_GL3W
4:   if ( gl3wInit () )
5:     ExitOnError ( "GetGLProcAddresses (gl3w)\n" );
6: #else
7:   int ec;
8:
9:   if ( (ec = glewInit ()) != GLEW_OK )
10:     ExitOnError ( glewGetErrorString ( ec ) );
11: #endif
12: } /*GetGLProcAddresses*/
```

używa wybranej biblioteki do pobrania tych adresów. Procedura `ExitOnError`, wywoływana w razie niepowodzenia, wypisuje do terminala napis podany jako parametr i zatrzymuje program.

Nazwy i typy danych w OpenGL-u

Typy zmiennych liczbowych przetwarzanych przez OpenGL-a (w tym parametrów procedur OpenGL-a) mają nazwy zdefiniowane w opisanych wcześniej plikach nagłówkowych; nazwy te zaczynają się od przedrostka `GL` i są w zasadzie synonimami nazw typów standardowych w C. Filozofia przyświecająca ich wprowadzeniu jest taka, że nazwa ma podkreślać zastosowanie, a ponadto ma być zagwarantowana jednoznaczność reprezentacji: `GLint` ma zawsze oznaczać typ liczb całkowitych 32-bitowych ze znakiem w kodzie uzupełnieniowym do 2 niezależnie od tego, czy właśnie taki typ ukrywa się pod nazwą `int`. Nazwy najważniejszych typów liczbowych OpenGL-a są zebrane w tabeli 2.1.

typ OpenGL	typ C	zastosowanie
GLvoid	void	typ pusty,
GLboolean	unsigned char	zmienne boolowskie o wartościach GL_FALSE i GL_TRUE,
GLbyte	signed char	8-bitowe liczby całkowite ze znakiem,
GLubyte	unsigned char	8-bitowe liczby całkowite bez znaku,
GLchar	char	kody ASCII,
GLshort	short	16-bitowe liczby całkowite ze znakiem,
GLushort	unsigned short	16-bitowe liczby całkowite bez znaku,
GLint	int	32-bitowe liczby całkowite ze znakiem,
GLuint	unsigned int	32-bitowe liczby całkowite bez znaku,
		identyfikatory obiektów (buforów, szaderów itp.),
GLenum	unsigned int	stałe o nadanych nazwach,
GLsizei	int	wielkości buforów i tablic,
GLbitfield	unsigned int	32-bitowe pole bitowe,
GLfloat	float	32-bitowe liczby zmiennopozycyjne,
GLdouble	double	64-bitowe liczby zmiennopozycyjne.

Tabela 2.1: Typy OpenGL-a

Wszystkie stałe symboliczne określone za pomocą dyrektyw `#define` w plikach nagłówkowych OpenGL-a mają nazwy zaczynające się od przedrostka `GL_`.

Przedrostki i przyrostki nazw procedur

Wszystkie procedury podstawowego OpenGL-a mają nazwy zaczynające się od przedrostka `gl`, po którym następuje wielka litera — początek właściwej nazwy procedury. Zatem np. `glGetError` jest nazwą procedury w podstawowej bibliotece

OpenGL, podczas gdy gluErrorString jest nazwą procedury z pomocniczej biblioteki libGLU.so (której plik nagłówkowy ma nazwę GL/glu.h). Ale procedur o nazwach zaczynających się od glX *może* nie być w libGL.so i wtedy należy ich szukać w bibliotece libGLX; tak czy inaczej, ich prototypy są w pliku GL/glx.h.

Przyrostek nazwy procedury OpenGL-a, *jeśli występuje*, określa typ jej parametrów i sposób ich podania — służy to do rozróżniania procedur spełniających tę samą rolę. Przyrostki składają się z jednej, dwóch lub trzech liter i ewentualnie poprzedzającej je jednej cyfry. Na przykład procedura glUniform1f nadaje (skalarnej zmiennopozycyjnej) zmiennej jednolitej wartość podaną przez jeden parametr typu GLfloat. Procedura glUniform3f nadaje zmiennej mającej trzy składowe zmiennopozycyjne wartość podaną w trzech parametrach typu GLfloat, a procedura glUniform3fv robi to samo, ale w tym przypadku trzy liczby typu GLfloat mają być podane w przekazanej jako parametr tablicy.

Zatem: cyfra 1, 2, 3 lub 4 w przyrostku określa liczbę podanych (jako osobne parametry lub w tablicy) liczb. Litery b, ub, s, us, i, ui, f albo d określają typ parametru lub parametrów, przy czym obecność litery u oznacza typ całkowity bez znaku, litery b, s i i oznaczają odpowiednio liczby całkowite 8-, 16- i 32-bitowe, a litery f i d — liczby zmiennopozycyjne, 32- lub 64-bitowe. Obecność litery v oznacza, że parametry są przekazywane w tablicy.

Zestawienie bibliotek

libGL — podstawowa biblioteka OpenGL-a, zawiera procedury rysowania i procedury niezbędne do tego, aby przygotować proces rysowania. Jej pliki nagłówkowe mają nazwy GL/gl.h, GL/glext.h, GL/glcorearb.h, sposób ich używania w aplikacji jest opisany wcześniej w tym rozdziale.

libGLU — biblioteka pomocnicza, większość procedur w niej zawartych (ale nie wszystkie) jest związana z pracą w trybie natychmiastowym starego OpenGL-a. Jej plik nagłówkowy to GL/glu.h.

libGLX — biblioteka z procedurami organizującymi współpracę OpenGL-a z systemem X Window; jej głównym zadaniem jest tworzenie kontekstów OpenGL-a, wiązanie ich z kanwami (*drawables*), tj. oknami (*windows*) i wewnętrznymi buforami obrazu (*pixmaps*) systemu X Window oraz podawanie adresów procedur OpenGL-a. Jej plik nagłówkowy nazywa się GL/glx.h.

Procedury z tej biblioteki mogą być obecne w bibliotece libGL (tak jest w używanej przeze mnie instalacji w Linuksie) i wtedy osobna biblioteka

`libGLX` jest niepotrzebna (nawet jeśli jest obecna na dysku).

`libGLEW` — biblioteka z zadeklarowanymi zmiennymi wskaźnikowymi do procedur OpenGL-a, ma za zadanie nadać tym zmiennym odpowiednie wartości. Jej plik nagłówkowy ma nazwę `GL/glew.h`, do aplikacji włączamy go *zamiast* plików nagłówkowych biblioteki `libGL`.

`gl3w.o`, `libgl3w` — alternatywa dla `libGLEW`, plik obiektowy do bezpośredniego dołączenia do aplikacji lub biblioteka dołączana dynamicznie, z plikiem nagłówkowym `gl3w.h` (lub `GL/gl3w.h`, jeśli go wpiszemy do katalogu `/usr/include/GL`).

`libglut` — biblioteka FreeGLUT opisana w następym rozdziale; jej zadaniem jest umożliwienie pisania aplikacji niezależnych od systemu operacyjnego i systemu okien. Dyrektywę włączającą jej plik nagłówkowy, `GL/freeglut.h` należy w aplikacjach nowego OpenGL-a poprzedzić włączeniem plików nagłówkowych biblioteki `libGL` *albo* biblioteki `libGLEW` *albo* `gl3w`.

`libGLFW` — inna (też opisana w następnym rozdziale) biblioteka ukrywająca kod zależny od systemu operacyjnego i systemu okien, nowsza i nowocześniejsza niż FreeGLUT. Jej plik nagłówkowy ma nazwę `GLFW/glfw3.h`.

W tej książce *nie będzie* dokładnego opisu procedur OpenGL-a z wyszczególnieniem *wszystkich* informacji na temat parametrów, ich możliwych wartości itp., nie będzie też wszystkich szczegółów i niuansów języka GLSL. Przepisywanie pełnej dokumentacji, która liczy wiele setek stron, nie ma sensu. Szczegółowych informacji można (i w pewnym momencie warto) poszukać w tej dokumentacji, dostępnej w Internecie na stronach Khronos Group, np. [1], [3] lub [5].

W książce będą natomiast podane przykłady zastosowania najważniejszych procedur OpenGL-a w aplikacjach — z opisem, co dokładnie te procedury tam robią, dlaczego, i w szczególności jak poszczególne procedury współpracują ze sobą. Czytelnika, który zapoznał się z zastosowaniem procedury w którejś z opisanych dalej aplikacji zachęcam, *po zrozumieniu* jaką rolę ta procedura spełnia w aplikacji, do zajrzenia na stronę [5], wyszukania tej procedury i przeczytania opisu, w tym opisu innych niż użyte w aplikacji wartości parametrów i możliwych skutków ich użycia. Zachęcam też do samodzielnych eksperymentów, w tym do modyfikowania aplikacji i patrzenia, co się ~~zepsu~~ zmieniło.

3. Otoczenie OpenGL-a

Aplikacja graficzna zazwyczaj nie tylko wyświetla obraz, ale także prowadzi dialog z użytkownikiem, przyjmując jego polecenia wydawane za pomocą urządzeń wejściowych (np. klawiatury, myszy, dżojstika). Ponadto aplikacja wyświetla obrazy na ekranie, na który jednocześnie inne programy wyprowadzają wyniki swoich działań; od tego jest system operacyjny i system okien, aby aplikacje nie przeszkadzały sobie nawzajem i aby otrzymywały komunikaty o działaniach użytkownika.

Niezbędnym elementem działającej aplikacji jest kontekst OpenGL-a. Jest to struktura danych zawierająca dane opisujące stan OpenGL-a (w tym wspominane wcześniej przełączniki) oraz dane aplikacji — bufory wierzchołków, zmienne jednolite, tekstury, programy szaderów itd. Kontekst składa się z dwóch części; jedna z nich znajduje się w pamięci CPU, a druga GPU. Sposób tworzenia kontekstu jest zależny od środowiska, w którym aplikacja ma działać. Inne instrukcje muszą być w tym celu wykonane przez aplikację Unixa (lub Linuxa), a inne przez aplikację Macintosha i jeszcze inne w Windowsach. Najprostszym sposobem poradzenia sobie z tym fragmentem aplikacji jest użycie biblioteki GLUT (albo FreeGLUT) albo GLFW; biblioteki te, skompilowane dla różnych środowisk, ukrywają kod specyficzny dla środowiska.

FreeGLUT

Biblioteka GLUT (*The OpenGL Utility Toolkit*) była rozwijana w latach 1994–1998; jej autorem jest Mark Kilgard, pracujący wtedy w Silicon Graphics. Biblioteka ta ma na celu uniezależnienie interakcyjnej aplikacji od środowiska. Projekt ten nie jest już rozwijany, a ponadto nie jest to wolne oprogramowanie. Ale istnieją jego wolno dostępne i nadal rozwijane zamienniki. Jednym z nich jest biblioteka FreeGLUT (autor: Paweł W. Olszta, początek projektu w 1999 r.), która dokładnie odtwarza funkcjonalność oryginalnego GLUTa i ma rozszerzenia dostosowujące projekt do nowego OpenGL-a. Opis procedur oryginalnego GLUTa jest w dokumencie [16], zaś uzupełnienia FreeGLUTa można znaleźć na stronie [17].

Makra w pliku nagłówkowym FreeGLUTa mają nazwy zaczynające się od przedrostka `GLUT`, zaś wszystkie procedury mają na początku nazwy przedrostek `glut`. Na listingu 3.1 jest przedstawiony szkielet aplikacji FreeGLUTa, która tworzy jedno okno aby w nim wyświetlać grafikę i która reaguje na zdarzenia takie jak polecenia użytkownika wydawane za pomocą myszy, klawiatury i (jeśli trzeba) dżojstika oraz na upływ czasu.

Listing 3.1: Szkielet aplikacji FreeGLUTa

aplikacjaFreeGLUTa.c

```
1: #include <stdlib.h>
2: #include <stdio.h>
3: #include "openglheader.h> /* najpierw ten */
4: #include <GL/freeglut.h>  /* potem ten */
5:
6: #include "utilities.h"
7: #include "myheader.h"
8:
9: int WindowHandle;
10:  ... /* inne zmienne globalne potrzebne w programie */
11:
12: void Cleanup ( void ) { ... }
13:
14: void ReshapeFunc ( int width, int height ) { ... }
15: void DisplayFunc ( void ) { ... }
16: void KeyboardFunc ( unsigned char key, int x, int y ) { ... }
17: void SpecialKeyFunc ( int key, int x, int y ) { ... }
18: void MouseFunc ( int button, int state, int x, int y ) { ... }
19: void MotionFunc ( int x, int y ) { ... }
20: void JoystickFunc ( unsigned int buttonmask, int x, int y, int z ) { ... }
21: void TimerFunc ( int value ) { ... }
22: void IdleFunc ( void ) { ... }
23:
24: void LoadMyShaders ( void ) { ... }
25: void InitMyObject ( void ) { ... }
26:
27: void Initialise ( int argc, char *argv[] )
28: {
29:   glutInit ( argc, argv );
30:   glutInitContextVersion ( 2, 1 );
31:   glutInitContextFlags ( GLUT_FORWARD_COMPATIBLE );
32:   glutInitContextProfile ( GLUT_CORE_PROFILE );
33:   glutSetOption ( GLUT_ACTION_ON_WINDOW_CLOSE,
34:                   GLUT_ACTION_GLUTMAINLOOP_RETURNS );
35:   glutInitWindowSize ( 480, 360 );
36:   glutInitDisplayMode ( GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA );
37:   WindowHandle = glutCreateWindow ( "Aplikacja FreeGLUTa" );
38:   if ( WindowHandle < 1 ) {
39:     fprintf ( stderr, "Error: Could not create a window\n" );
40:     exit ( 1 );
41:   }
42:   glutReshapeFunc ( ReshapeFunc );
```

```
43:   glutDisplayFunc ( DisplayFunc );
44:   glutKeyboardFunc ( KeyboardFunc );
45:   glutSpecialFunc ( SpecialKeyFunc );
46:   glutMouseFunc ( MouseFunc );
47:   glutMotionFunc ( MotionFunc );
48:   /*glutJoystickFunc ( JoystickFunc, 16 );*/
49:   /*glutTimerFunc ( 0, TimerFunc, 0 );*/
50:   /*glutIdleFunc ( IdleFunc );*/
51:   GetGLProcAddresses ();
52:   LoadMyShaders ();
53:   InitMyObject ();
54: } /*Initialise*/
55:
56: int main ( int argc, char *argv[] )
57: {
58:   Initialise ( argc, argv );
59:   glutMainLoop ();
60:   exit ( 0 );
61: } /*main*/
```

Plik nagłówkowy `openglheader.h` i procedura `GetGLProcAddresses` są opisane w rozdziale 2. Jeśli nie chcemy używać biblioteki `gl3w` ani `libGLEW` i nie mamy możliwości dołączenia procedur nowego OpenGL-a bezpośrednio, to mamy do dyspozycji procedurę `glutGetProcAddress` (która wywołuje po kryjomu odpowiednią procedurę podającą adres potrzebnego podprogramu, taką jak `glXGetProcAddress`).

Jeśli dyrektywa `#include <GL/freeglut.h>` *nie jest* poprzedzona włączeniem pliku `gl3w.h`, `GL/glew.h` albo oryginalnych plików nagłówkowych OpenGL-a, to kompilacja pliku `GL/freeglut.h` spowoduje włączenie plików `GL/gl.h` i `GL/glu.h` z prototypami procedur *starego* OpenGL-a.

Plik nagłówkowy `utilities.h` zawiera prototypy przydatnych w każdej aplikacji OpenGL-a (nie tylko FreeGLUTa) podprogramów pomocniczych, które kompilują i instalują programy szaderów i wykonują najczęściej potrzebne działania na macierzach i wektorach; zajmiemy się nimi w następnych trzech rozdziałach. Związane z aplikacją pliki nagłówkowe, takie jak `myheader.h` (który może mieć inną nazwę) są potrzebne, jeśli kod źródłowy aplikacji ma być podzielony na kilka osobnych plików. W plikach nagłówkowych trzeba umieścić odpowiednie definicje typów i prototypy procedur aplikacji, aby umożliwić niezależną kompilację tych plików.

Procedura main wykonuje dwie instrukcje: wywołuje procedurę Initialise, której zadaniem jest przygotowanie programu do pracy, w tym utworzenie okna i zarejestrowanie odpowiednich procedur obsługi zdarzeń dla tego okna, a następnie glutMainLoop, która działa aż do momentu zatrzymania programu; procedura ta wywołuje zarejestrowane procedury w odpowiedzi na zdarzenia wejściowe i zdarzenia spowodowane przez aplikację.

Instrukcje w liniach 31–36 przygotowują struktury danych FreeGLUTa do pracy. Procedura glutInit inicjalizuje (z uwzględnieniem parametrów wywołania programu) wewnętrzne struktury danych FreeGLUTa.

W zasadzie procedura glutInitContextVersion powinna być wywołana z parametrami na przykład 4, 2, aby przekazać FreeGLUTowi informację, że ta aplikacja jest napisana zgodnie ze specyfikacją OpenGL 4.2. Ale z niejasnych dla mnie powodów jeśli używamy biblioteki GLEW, to coś się tam psuje i dlatego trzeba dać parametry 2 i 1, aby program dał się uruchomić. Ale to nie szkodzi, procedura glutInitContextFlags przyjmuje w imieniu FreeGLUTa deklarację, że chcemy, aby ta aplikacja działała także z implementacjami OpenGL-a zgodnymi z późniejszymi wersjami, w tym 4.2.

Procedura glutInitContextProfile przyjmuje od aplikacji zapewnienie, że nie będą używane procedury starego OpenGL-a (gdybyśmy chcieli, to trzeba podać parametr GLUT_COMPATIBILITY_PROFILE, ale nie róbmy tego).

Procedura glutSetOption wprowadza informacje o pożądanym zachowaniu procedur FreeGLUTa — w rozpatrywanym tu przykładzie jest żądanie, aby aplikacja mogła spowodować powrót z procedury glutMainLoop[1].

W liniach 37–43 tworzymy okno. Procedura glutInitWindowSize określa jego wymiary (w pikselach, z pominięciem dołączanej przez menedżera okien ramki).

Procedura glutInitDisplayMode określa potrzebne aplikacji zasoby OpenGL-a. W tym przykładzie aplikacja domaga się trybu RGBA, tzn. takiego, w którym składowe koloru piksela są przechowywane bezpośrednio w pikselu — w nowym OpenGL-u to jest jedyna dopuszczalna możliwość (w starym OpenGL-u była też możliwość używania trybu z paletą, wartość piksela była indeksem do palety, czyli tablicy przechowującej kolory do wyświetlenia na ekranie). Ponadto maska

[1]Należy w tym celu wywołać procedurę glutLeaveMainLoop. Takiej możliwości nie było w oryginalnej bibliotece GLUT; aby zakończyć działanie programu, jedna z zarejestrowanych procedur aplikacji musiała wykonać exit.

`GLUT_DEPTH` deklaruje zapotrzebowanie na bufor głębokości (do wyznaczania widoczności obiektów na obrazie), zaś maska `GLUT_DOUBLE` wybiera podwójne buforowanie — dla okna będą utworzone dwa bufory obrazu. W dowolnej chwili zawartość jednego z nich jest widoczna w oknie, a rysowanie odbywa się w drugim buforze; po zakończeniu rysowania bufory są zamieniane rolami, dzięki czemu obraz widoczny w oknie zawsze jest ukończony.

Po opisanych wyżej przygotowaniach można wywołać procedurę `glutCreateWindow`, z parametrem — napisem, który ma być wyświetlony na ramce okna. Wartość zwrócona przez tę procedurę jest identyfikatorem okna, tj. pewną liczbą dodatnią. Jeśli nie udało się utworzyć okna (bo np. zażądaliśmy niedostępnych zasobów), to procedura `glutCreateWindow` zwróci wartość `0`, co powinno spowodować zatrzymanie programu i próbę wyjaśnienia przez jego autora, co poszło nie tak. Bardziej skomplikowane aplikacje mogą tworzyć wiele okien i/lub podokien (za pomocą procedury `glutCreateWindow` albo `glutCreateSubwindow`), z których każde otrzyma swój unikalny identyfikator.

Okno lub podokno zaraz po utworzeniu, a także w chwili wywołania przez FreeGLUTa zarejestrowanej procedury obsługi komunikatu o zdarzeniu dla tego okna lub po wywołaniu (w trakcie obsługi takiego komunikatu dla innego okna) procedury `glutSetWindow` z parametrem — identyfikatorem okna — jest oknem aktywnym. W szczególności procedury rejestrujące przywiązują procedury obsługi zdarzeń do okna aktywnego. W naszym przykładzie procedury wywołane w liniach 44–49 przywiążą procedury obsługi komunikatów o zdarzeniach dla okna utworzonego przez instrukcję w linii 39, które jest jedynym oknem w tej aplikacji i pozostanie aktywne przez cały czas jej działania.

Procedura `ReshapeFunc` rejestrowana w linii 44 przez `glutReshapeFunc` zostanie wywołana (przez `glutMainLoop`) po utworzeniu okna i po każdej zmianie jego szerokości lub wysokości. Jej parametry opisują nowe wymiary okna, a jej zadaniem jest obliczenie macierzy rzutowania dla okna w nowym kształcie.

Procedura `DisplayFunc` rejestrowana w linii 45 przez `glutDisplayFunc` zostanie wywołana po każdej zmianie wymiarów okna (ale wcześniej będzie wywołana procedura zarejestrowana przez `glutReshapeFunc`) oraz po jego odsłonięciu na ekranie (gdy użytkownik odsunął lub zamknął okno zasłaniające) i po zawiadomieniu FreeGLUTa przez aplikację, że zawartość okna zmieniła się i należy wyświetlić nowy obraz. Do takiego zawiadamiania służą procedury `glutPostRedisplay` (wywołuje procedury rysowania dla wszystkich okien) lub `glutPostWindowRedisplay` (wywołuje procedurę rysowania jednego okna lub

podokna, o identyfikatorze podanym jako parametr).

Uwaga: Jedyną procedurą, która wykonuje rysowanie w oknie, ma prawo być procedura zarejestrowana dla tego okna przez glutDisplayFunc i wywoływana tylko przez FreeGLUTa. Jeśli komunikat o zdarzeniu (np. naciśnięciu klawisza) spowodował konieczność wykonania nowego obrazu, to procedura obsługi tego zdarzenia ma przygotować odpowiednio zmienioną reprezentację obiektów na obrazie i wywołać procedurę glutPostRedisplay albo glutPostWindowRedisplay.

Procedura KeyboardFunc, rejestrowana w linii 46 przez glutKeyboardFunc jest wywoływana po naciśnięciu klawisza, gdy kursor znajduje się w oknie. Jej parametry to kod klawisza (kod ASCII skojarzonego z tym klawiszem znaku) oraz współrzędne położenia kursora w oknie. Uwaga: układ współrzędnych używany przez FreeGLUTa ma początek w *górnym lewym* narożniku okna; większą współrzędną y mają punkty położone niżej. To jest orientacja odwrotna do orientacji układu stosowanego przez OpenGL-a (którego początek jest w dolnym lewym narożniku okna). Jednostki długości osi x i y odpowiadają szerokości i wysokości jednego piksela.

W linii 47 jest rejestrowana procedura SpecialKeyFunc, która będzie wywołana po naciśnięciu klawisza specjalnego, takiego jak strzałka lub F1, ..., F12. Naciśnięty klawisz jest identyfikowany przez parametr key, którego wartość liczbowa jest w programie zastępowana przez czytelniejszą nazwę odpowiedniej makrodefinicji zdefiniowanej w pliku nagłówkowym GL/freeglut_std.h.

Procedury glutMouseFunc i glutMotionFunc wywoływane w liniach 48 i 49 rejestrują odpowiednio procedury, które będą wywołane po naciśnięciu lub zwolnieniu przycisku myszy oraz po przesunięciu myszy (czyli zmianie położenia kursora). Parametry x i y tych procedur opisują położenie kursora w oknie. Parametry button i state identyfikują przycisk, którego dotyczy zdarzenie i jego stan po zmianie; ich możliwe wartości to odpowiednio GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON lub GLUT_RIGHT_BUTTON oraz GLUT_DOWN lub GLUT_UP.

Jeśli aplikacja ma przyjmować komunikaty wejściowe od dżojstika, to możemy (w linii 50) zarejestrować procedurę JoystickFunc. Drugi parametr procedury glutJoystickFunc określa częstotliwość, z jaką FreeGLUT „odpytuje” dżojstik i wywołuje zarejestrowaną procedurę, a dokładniej, odstęp (w milisekundach) między kolejnymi wywołaniami tej procedury. Jej pierwszy parametr jest maską bitową; poszczególne bity określają stan przycisków dżojstika (1, jeśli przycisk jest przyciśnięty, 0 w przeciwnym razie). Parametry x, y, z opisują położenie drążka

dżojstika, tj. kąty jego obrotów wokół osi x, y, z. Miarą każdego kąta jest liczba całkowita z przedziału $[-1000, 1000]$, który odpowiada maksymalnemu zakresowi każdego z tych obrotów. FreeGLUT ma też procedury umożliwiające badanie, ile dżojstik ma osi i przycisków i konfigurowanie go do szczególnych potrzeb aplikacji[2].

Procedury `glutTimerFunc` używamy, jeśli chcemy, aby rejestrowana przez nią procedura została wywołana po upływie określonego czasu. Pierwszy parametr określa ilość czasu podaną w milisekundach, drugi to adres rejestrowanej procedury, a trzeci parametr może być użyty do przekazania tej procedurze (przez parametr) informacji o miejscu w programie, w którym została zarejestrowana. Jeśli chcemy, aby aplikacja wykonywała jakąś czynność co sekundę, to powinniśmy w procedurze `TimerFunc` wywołać procedurę `glutTimerFunc` z pierwszym parametrem równym `1000`, a drugim — równym adresowi procedury `TimerFunc`, a potem wykonać instrukcje realizujące tę czynność.

Wreszcie, procedura `glutIdleFunc` rejestruje podprogram, który ma być natychmiast wywołany za każdym razem, gdy FreeGLUT nie ma żadnych innych zadań do wykonania. Taki podprogram przydaje się w animacji; na podstawie zegara systemowego powinien określić stan obiektów do narysowania w danej chwili i spowodować narysowanie tych obiektów (przez wywołanie `glutPostRedisplay` lub `glutPostWindowRedisplay`). Procedury `glutTimerFunc` i `glutIdleFunc` rejestrują odpowiednie podprogramy dla całej aplikacji, a nie dla jednego okna.

„Wyrejestrowanie” podprogramu obsługi komunikatów następuje przez wywołanie odpowiedniej procedury rejestrującej z parametrem `NULL`, można też w każdej chwili zarejestrować inny podprogram, który przejmie obsługę komunikatów w aplikacji.

Ostatnie dwie instrukcje w procedurze inicjalizacji (linie 52, 53) wywołują procedury, z których pierwsza ma za zadanie umieszczenie w kontekście OpenGL-a potrzebnych programów szaderów, a druga ma utworzyć początkową reprezentację obiektów do narysowania, co obejmuje inicjalizację struktur danych aplikacji w pamięci CPU oraz utworzenie potrzebnych buforów (np. VAO, UBO, zobacz rozdział 1) i tekstur w pamięci GPU.

[2] Zwróćmy uwagę, że o ile procedury obsługi zdarzeń związanych z myszą i klawiaturą są wywoływane po wystąpieniu odpowiedniego zdarzenia (np. po naciśnięciu klawisza lub przesunięciu myszy), procedura związana z dżojstikiem jest wywoływana co chwila, niezależnie od tego, czy stan dżojstika się zmienił.

Aby zakończyć działanie programu, jedna z procedur obsługi komunikatów powinna wywołać procedurę glutLeaveMainLoop, co spowoduje powrót z procedury glutMainLoop. Przedtem jednak wypada wywołać procedurę Cleanup, której zadaniem jest zwolnienie zasobów, które aplikacja zarezerwowała na swoje potrzeby; oprócz bloków pamięci zarezerwowanych bezpośrednio przez aplikację za pomocą procedury malloc są to zasoby OpenGL-a i systemu okien, w tym programy szaderów, bufory, tekstury, kontekst OpenGL-a i okna. Do zamknięcia okna służy procedura glutDestroyWindow, która oprócz okna likwiduje także związany z nim kontekst graficzny.

FreeGLUT — uwagi dodatkowe

Domyślnie dla każdego okna i podokna FreeGLUT tworzy nowy kontekst OpenGL-a. Programy, bufory i inne obiekty obecne w kontekście są niewidoczne dla innych kontekstów, co sprawia, że jeśli mamy użyć takich samych obiektów (np. programów) do rysowania w różnych oknach, to musimy je utworzyć dla każdego okna. Można chcieć, aby okna miały wspólny kontekst. W tym celu należy przed tworzeniem okien wykonać instrukcję

```
glutSetOption ( GLUT_RENDERING_CONTEXT, GLUT_USE_CURRENT_CONTEXT );
```

W czasie, gdy okno jest aktywne, jest również aktywny związany z nim kontekst. Jeśli zatem aplikacja najpierw utworzy kilka okien z osobnymi kontekstami, a potem będzie instalować programy szaderów, tworzyć bufory itd., to *przed* utworzeniem tych obiektów dla każdego okna należy je uaktywnić (co uaktywnia związany z tym oknem kontekst) — wywołując procedurę glutSetWindow z parametrem identyfikującym to okno.

GLFW

Biblioteka GLFW występuje w dwóch istotnie różnych wersjach: 2.? (np. 2.7) i 3.? (np. 3.1, 3.2); zajmiemy się tą drugą wersją. Choć można przy jej użyciu pisać aplikacje w stylu bardziej „niskopoziomowym” niż aplikacje FreeGLUTa, na listingu 3.2 zamieściłem szkielet możliwie podobny do tego z listingu 3.1. Uwagi na temat „niskopoziomowego” stylu pisania aplikacji są podane na końcu tego podrozdziału, a pełną dokumentację można znaleźć na stronie [18].

Przed włączeniem pliku nagłówkowego GLFW/glfw3.h biblioteki GLFW (linia 4) należy włączyć plik nagłówkowy odpowiedniej biblioteki „wprowadzającej”. Zamiast identyfikatorów okien, będących liczbami całkowitymi, biblioteka GLFW

używa wskaźników do struktur typu GLFWwindow; razem z każdym oknem otwieranym przez aplikację biblioteka tworzy taką strukturę, której budowa jest dla aplikacji niewidoczna.

W bibliotece FreeGLUT działania takie jak rysowanie, zmiany wymiarów itp. są wykonywane na oknie aktywnym; chcąc je zmienić (np. po to, aby w trakcie reakcji na komunikat skierowany do okna móc zmienić obrazek w innym oknie), należy wywołać procedurę glutSetWindow z identyfikatorem okna podanym jako parametr. Dla odmiany, procedury w bibliotece GLFW i w jej aplikacji mają wskaźnik do struktury GLFWwindow, pełniący obowiązki identyfikatora okna, podawany jako parametr.

W procedurze main (linie 75–81) mamy, jak w poprzednim szkielecie, wywołania kolejno procedury inicjalizacji Initialise, pętli komunikatów MessageLoop i sprzątania Cleanup (która na końcu powinna wywołać procedurę glfwTerminate, sprzątającą po procedurach biblioteki GLFW). Przyjrzyjmy się inicjalizacji.

W linii 34 procedura glfwSetErrorCallback rejestruje procedurę obsługi błędów; zostanie ona wywołana, jeśli któraś z procedur biblioteki GLFW wykryje sytuację błędną, przekazując liczbowy kod błędu i tekst go opisujący. Procedura zapisana w liniach 13–17 wypisze ten tekst i przerwie działanie aplikacji.

Procedura wywołana w linii 35 inicjalizuje wewnętrzne struktury danych biblioteki GLFW. Jeśli to się nie powiedzie, to trudno, koniec, kropka i exit.

Podobnie, aplikacja zostanie zatrzymana, jeśli nie powiedzie się tworzenie nowego okna w liniach 39–40. Parametry procedury glfwCreateWindow określają odpowiednio wymiary (szerokość i wysokość) okna w pikselach, napis do wyświetlenia na ramce okna, wskaźnik monitora i wskaźnik okna współdzielącego kontekst. Wskaźnik monitora, jeśli jest pusty (NULL), zgłasza potrzebę utworzenia zwykłego okna przez system okien. Jeśli nie jest pusty, to rysowanie ma się odbywać w trybie pełnoekranowym na wskazanym monitorze (po szczegóły na ten temat odsyłam do dokumentacji). Wskaźnik okna współdzielącego, jeśli nie jest pusty (NULL), oznacza, że okno ma mieć wspólny kontekst OpenGL-a z innym, wcześniej utworzonym oknem.

Przed utworzeniem okna można podać rozmaite parametry wpływające na jego własności, wywołując procedurę glfwWindowHint. W podanym szkielecie używane są domyślne wartości tych parametrów, bez potrzeby nie warto ich zmieniać.

Listing 3.2: Szkielet aplikacji biblioteki GLFW

aplikacjaGLFW.c

```
1: #include <stdlib.h>
2: #include <stdio.h>
3: #include "openglheader.h"
4: #include <GLFW/glfw3.h>
5:
6: #include "utilities.h"
7: #include "myheader.h"
8:
9: GLFWwindow *mywindow;
10: char redraw;
11:  ... /* inne zmienne globalne potrzebne w programie */
12:
13: void myGLFWErrorHandler ( int error, const char *description )
14: {
15:   fprintf ( stderr, "GLFW error: %s\n", description );
16:   exit ( 1 );
17: } /*myGLFWErrorHandler*/
18:
19: void Redraw ( GLFWwindow *win ) { ... redraw = false; }
20: void ReshapeFunc ( GLFWwindow *win, int width, int height ) { ... }
21: void DisplayFunc ( GLFWwindow *win ) { redraw = true; }
22: void CharFunc ( GLFWwindow *win, unsigned int charcode ) { ... }
23: void KeyFunc ( GLFWwindow *win, int key, int scancode,
24:                int action, int mods ) { ... }
25: void MouseFunc ( GLFWwindow *win, int button, int action, int mods ) { ... }
26: void MotionFunc ( GLFWwindow *win, double x, double y ) { ... }
27:
28: void LoadMyShaders ( void ) { ... }
29: void InitMyObject ( void ) { ... }
30: void Cleanup ( void ) { ... glfwTerminate (); }
31:
32: void Initialise ( int argc, char **argv )
33: {
34:   glfwSetErrorCallback ( myGLFWErrorHandler );
35:   if ( !glfwInit () ) {
36:     fprintf ( stderr, "Error: glfwInit failed\n" );
37:     exit ( 1 );
38:   }
39:   if ( !(mywindow = glfwCreateWindow ( 480, 360,
40:                                        "Aplikacja GLFW", NULL, NULL )) ) {
41:     glfwTerminate ();
42:     fprintf ( stderr, "Error: glfwCreateWindow failed\n" );
```

```
43:     exit ( 1 );
44:   }
45:   glfwMakeContextCurrent ( mywindow );
46:   GetGLProcAddresses ();
47:   if ( !gl3wIsSupported ( 4, 2 ) ) {
48:     fprintf ( stderr, "Error: OpenGL version 4.2 not supported\n" );
49:     exit ( 1 );
50:   }
51:   glfwSetWindowSizeCallback ( mywindow, ReshapeFunc );
52:   glfwSetWindowRefreshCallback ( mywindow, DisplayFunc );
53:   glfwSetCharCallback ( mywindow, CharFunc );
54:   glfwSetKeyCallback ( mywindow, KeyFunc );
55:   glfwSetMouseButtonCallback ( mywindow, MouseFunc );
56:   glfwSetCursorPosCallback ( mywindow, MotionFunc );
57:   LoadMyShaders ();
58:   InitMyObject ();
59:   ReshapeFunc ( mywindow, 480, 360 );
60:   redraw = true;
61: } /*Initialise*/
62:
63: void MessageLoop ( void )
64: {
65:   do {
66:     glfwWaitEvents ();
67:     if ( redraw )
68:       Redraw ( mywindow );
69:   } while ( !glfwWindowShouldClose ( mywindow ) );
70: } /*MessageLoop*/
71:
72: int main ( int argc, char **argv )
73: {
74:   Initialise ( argc, argv );
75:   MessageLoop ();
76:   Cleanup ();
77:   exit ( 0 );
78: } /*main*/
```

Biblioteka GLFW wymaga, aby przed rysowaniem lub przesyłaniem dowolnych danych (w tym opisu obiektów do rysowania oraz szaderów) aplikacja jawnie wybierała aktywny kontekst OpenGL-a (w bibliotece FreeGLUT robi to procedura `glutSetWindow`). Dla aplikacji z jednym oknem, na przykład takiej o szkielecie z listingu 3.2, odbywa się to „raz na zawsze” w linii 45.

Po określeniu bieżącego kontekstu należy uzyskać dostęp do procedur OpenGL-a

— następuje wywołanie procedury inicjalizacji biblioteki wprowadzającej, tu gl3w.

W liniach 51–56 rejestrowane są procedury obsługi komunikatów o zdarzeniach dla okna. Pozwoliłem sobie nadać im takie same lub podobne nazwy, jak w szkielecie aplikacji FreeGLUTa na listingu 3.1, ale trzeba zwrócić uwagę na inne parametry tych procedur. Pierwszy parametr każdej z nich (linie 20–26) to wskaźnik struktury `GLFWwindow`. Poza tym procedura `ReshapeFunc` jest taka, jak w aplikacji FreeGLUTa. Natomiast zadaniem procedury `DisplayFunc` jest tylko zapamiętanie informacji, że zawartość okna należy narysować — przez przypisanie niezerowej wartości zmiennej `redraw`[3]. Procedura rysująca `Redraw` jest wywoływana w pętli komunikatów aplikacji; po wykonaniu rysunku procedura zmienia wartość zmiennej `redraw` na `false` (czyli `0`). Procedury obsługi komunikatów (np. o przesunięciu myszy lub naciśnięciu klawisza) powinny przypisać niezerową wartość zmiennej `redraw`, jeśli obraz w oknie przestał być aktualny i należy narysować nowy obraz.

Parametry wywoływanej po przesunięciu myszy procedury `MotionFunc`, opisujące położenie kursora, są typu <u>`double`</u> a nie <u>`int`</u>, ale układ współrzędnych w oknie jest taki sam jak we FreeGLUCie, tj. z początkiem w górnym lewym narożniku okna i z osią y skierowaną do dołu.

Procedura `CharFunc`, rejestrowana dla okna przez wywołanie `glfwSetCharCallback` jest wywoływana po napisaniu znaku na klawiaturze.

Procedura `KeyFunc`, rejestrowana przez wywołanie `glfwSetKeyCallback` jest wywoływana po naciśnięciu lub zwolnieniu dowolnego klawisza, w tym klawisza specjalnego, np. strzałki lub `F1`, ..., `F12`. Po naciśnięciu klawisza „zwykłego” znaku (np. litery) procedura ta jest również wywoływana, ale jej parametry nie podają napisanego znaku, tylko identyfikują kod klawisza.

<u>Uwaga:</u> Choć pewne klawisze specjalne, np. `Esc`, są związane ze znakami, które mają kody ASCII, po ich naciśnięciu jest wywoływana tylko procedura zarejestrowana przez `glfwSetKeyCallback`. Nazwy makrodefinicji identyfikujących klawisze specjalne można znaleźć w pliku nagłówkowym `GLFW/glfw3.h`.

W liniach 57 i 58 są wywoływane procedury przygotowujące programy szaderów

[3] Do reprezentowania wartości boolowskich w aplikacji w C będziemy używać zmiennych typu <u>char</u>, którym będą przypisywane wartości `0` lub `1`, nazwane dla wygody `false` i `true`. W plikach nagłówkowych OpenGL-a są też nazwy `GL_FALSE` i `GL_TRUE`, których będziemy używać w wywołaniach procedur OpenGL-a z parametrami boolowskimi.

i obiekty do wyświetlania. W linii 59 jest wywołana procedura ReshapeFunc; wywołanie takiej procedury po utworzeniu okna FreeGLUT bierze na siebie, ale biblioteka GLFW nie, więc aplikacja musi to zrobić osobiście.

Pętla komunikatów musi być oprogramowana w aplikacji (nie ma odpowiednika procedury glutMainLoop). Najprostsza procedura pętli komunikatów jest podana w liniach 63–70. Aplikacja będzie czekać na odpowiednie zdarzenie i po jego wystąpieniu zostanie wywołana odpowiednia zarejestrowana procedura. Jeśli przypisze ona wartość niezerową zmiennej redraw, to procedura Redraw narysuje co trzeba. Warunek zakończenia pętli to niezerowa wartość powrotna procedury glfwWindowShouldClose; można spowodować podanie takiej wartości wykonując instrukcję

```
glfwSetWindowShouldClose ( mywindow, true );
```

na przykład w odpowiedzi na naciśnięcie klawisza Esc.

Są dwie najważniejsze procedury, które można wywoływać w pętli komnikatów aplikacji biblioteki GLFW: glfwWaitEvents i glfwPollEvents. Pierwsza z nich czeka na zdarzenie (np. naciśnięcie przycisku, przesunięcie myszy itd.) i wraca po wystąpieniu tego zdarzenia. Druga z tych procedur wraca natychmiast, także wtedy, gdy nic się nie wydarzyło. Dzięki tej procedurze można stworzyć mechanizm analogiczny do tego, który udostępnia procedura glutIdleFunc. Sposób, jak to zrobić, jest pokazany na listingu 3.3.

Jeśli zmienna idlefunc ma wartość NULL, to aplikacja będzie czekać na zdarzenia bez angażowania procesora (w tym czasie może on wykonywać instrukcje innych uruchomionych równolegle programów albo oszczędzać energię elektryczną). Przypisanie tej zmiennej adresu dowolnej procedury powoduje wywoływanie tej procedury, gdy tylko aplikacja nie ma nic innego do roboty — do odwołania, tj. do ponownego nadania tej zmiennej wartości NULL[4].

Alternatywą dla użycia procedur obsługi komunikatów rejestrowanych dla okna, tak jak w szkielecie aplikacji na listingu 3.2, jest używanie procedur, które „odpytują” system okien (ukryty przed aplikacją) na temat zdarzeń, które nastąpiły. Jest cała seria procedur biblioteki GLFW, które podają bieżące wymiary okna, położenie kursora, ostatnio napisane na klawiaturze znaki itd.

[4] Ten kod nie nadaje się do użycia w programie wielowątkowym, w którym przypisanie wartości zmiennej idlefunc może wykonać inny wątek niż ten, który wykonuje procedurę MessageLoop.

Listing 3.3: Alternatywna procedura pętli komunikatów

C

```c
static void (*idlefunc)(void) = NULL;

void SetIdleFunc ( void(*IdleFunc)(void) )
{
  idlefunc = IdleFunc;
} /*SetIdleFunc*/

void MessageLoop ( void )
{
  do {
    if ( idlefunc ) {
      idlefunc ();
      glfwPollEvents ();
    }
    else
      glfwWaitEvents ();
    if ( redraw )
      Redraw ( mywindow );
  } while ( !glfwWindowShouldClose ( mywindow ) );
} /*MessageLoop*/
```

Zastosowanie tych procedur to właśnie jest ten „niskopoziomowy” styl programowania aplikacji biblioteki GLFW.

GLFW — obsługa dżojstika

Postaci informacji od dżojstika w API bibliotek GLFW i FreeGLUT są różne, inny jest też sposób ich przekazywania. Zamiast rejestrować procedurę, która będzie co chwila wywoływana z parametrami niosącymi te informację, aplikacja GLFW musi sama odpowiednio często „pytać” o stan i samopoczucie dżojstika. Do komputera może być podłączony więcej niż jeden; poszczególne urządzenia mają identyfikatory `GLFW_JOYSTICK_1`, . . . , `GLFW_JOYSTICK_16`. Procedura `glfwJoystickPresent` bada, czy w chwili jej wywołania wskazany dżojstik jest podłączony (co może się zmieniać podczas działania aplikacji). Procedury `glfwGetJoystickAxes` i `glfwGetJoystickButtons` podają wskaźniki tablic, w których zapisany jest stan dżojstika. Drugi parametr każdej z tych procedur jest adresem zmiennej, w której procedura zapisuje liczbę osi lub przycisków dżojstika (czyli długość tablicy). Aplikacja nie ma prawa przypisywać wartości do tych tablic ani sama ich dealokować. Jeśli dżojstik w międzyczasie został odłączony, procedury `glfwGetJoystickAxes` i `glfwGetJoystickButtons` zwracają

wskaźnik pusty. Przykład procedury, która odczytuje (i wypisuje w terminalu) stan dżojstika jest pokazany na listingu 3.4. Parametr `joy` ma być jednym z identyfikatorów, np. `GLFW_JOYSTICK_1` itd. Kąty wychylenia drążka są reprezentowane w tablicy `axes` przez liczby zmiennopozycyjne z przedziału $[-1,1]$. Jeśli `i`-ty przycisk jest przyciśnięty, to `i`-ty element tablicy `buttons` ma wartość `1`, a w przeciwnym razie `0`.

Listing 3.4: Czytanie stanu dżojstika w aplikacji GLFW

```c
void PollJoystick ( int joy )
{
  const float          *axes;
  const unsigned char *buttons;
  int                  count, i;

  if ( (axes = glfwGetJoystickAxes ( joy, &count )) )
    for ( i = 0; i < count; i++ )
      printf ( "%6.3f ", axes[i] );
  if ( (buttons = glfwGetJoystickButtons ( joy, &count )) )
    for ( i = 0; i < count; i++ )
      printf ( "%1d ", buttons[i] );
  printf ( "\n" );
} /*PollJoystick*/
```

GLFW — uwagi dodatkowe

Nie ma w bibliotece GLFW możliwości tworzenia podokien. Aplikacja, która potrzebuje podzielić okno na podobszary (np. widok sceny i menu z wihajstrami albo widok z okna samolotu i deska z przyrządami pokładowymi) musi dokonać odpowiedniego podziału sama. Efekt równoważny rysowaniu w podoknach może być osiągnięty przez odpowiednie ustawianie klatki (*viewport* — o tym w rozdziale 6) albo użycie bufora szablonu lub testu nożyczek.

Procedura podająca adresy procedur OpenGL-a o nazwach wskazanych przez parametr ma nazwę `glfwGetProcAddress`; zależnie od systemu okien, dla którego została skompilowana, wywołuje odpowiednią procedurę w tym systemie, np. `glXGetProcAddress`.

Biblioteka GLFW może być wykorzystana w programach wielowątkowych, ale w zasadzie tylko jeden wątek obliczeniowy może wywoływać procedury OpenGL-a (nazwijmy go wątkiem graficznym). Inne wątki aplikacji mogą z nim współdziałać. Na przykład wywołanie procedury `glfwPostEmptyEvent` przez inny

wątek spowoduje powrót z procedury `glfwWaitEvents` wywołanej przez wątek graficzny i czekającej na jakiekolwiek wydarzenie.

Biblioteka GLFW może też być użyta w aplikacjach standardu Vulkan. Może jest więc bardziej przyszłościowa niż FreeGLUT.

X Window i GLX

Aplikacja systemu X Window ma takie same elementy jak aplikacje FreeGLUTa i biblioteki GLFW, tj. procedurę inicjalizacji, która wykonuje niezbędne przygotowania (tworzy co najmniej jedno okno, tworzy kontekst OpenGL-a, kompiluje szadery i przygotowuje struktury danych reprezentujące m.in. obiekty do narysowania), pętlę komunikatów (która obsługuje zdarzenia spowodowane działaniami użytkownika, w szczególności wykonuje obrazy w oknie, gdy jest to potrzebne) i procedurę sprzątającą na końcu. Opisany tu szkielet aplikacji korzysta tylko ze struktur i procedur opisanych w plikach nagłówkowych `X11/Xlib.h` i `X11/Xutil.h` biblioteki `libX11` oraz z biblioteki `libGLX`[5] z plikiem nagłówkowym `GL/glx.h`. Ten interfejs jest znacznie bardziej niskopoziomowy i bardziej pracochłonny dla autora aplikacji w porównaniu z interfejsami udostępnionymi przez wcześniej opisane biblioteki, ale jest bardziej elastyczny i daje znacznie większą (bo pełną) kontrolę nad środowiskiem, w którym aplikacja działa. W szczególności umożliwia tworzenie okien o innym niż prostokątny kształcie (korzystając z rozszerzeń systemu X Window) i mieszanie grafiki tworzonej przez OpenGL-a z grafiką dwuwymiarową otrzymywaną za pomocą procedur rysujących systemu X Window — ale akurat od takiego mieszania głowa może bardzo rozboleć; jeśli trzeba w aplikacji korzystać zarówno z OpenGL-a, jak i procedur grafiki systemu X Window, które umożliwiają znacznie łatwiejsze stworzenie graficznego interfejsu użytkownika (*GUI — graphical user interface*) to najlepiej jest utworzyć w aplikacji okno z podoknami i w każdym podoknie tworzyć obrazy tylko za pomocą OpenGL-a albo tylko X Window. Szkielet głównej części aplikacji jest pokazany na listingu 3.5. W tym szkielecie jedynie dyrektywy `#include` w liniach 6 i 7 (oraz pominięte treści procedur) sprawiają, że jest to aplikacja OpenGL-a, a nie tylko systemu X Window.

Plik nagłówkowy `utilities.h` zawiera prototypy procedur opisanych w następnych trzech rozdziałach. W pliku `initglxctx.h` są zawarte deklaracje zmiennych używanych do komunikacji z systemem X Window i prototyp procedury, która tworzy kontekst OpenGL-a — jej opis jest zamieszczony dalej.

[5] być może stanowiącej część biblioteki `libGL`

Listing 3.5: Szkielet aplikacji OpenGL-a w systemie XWindow

C

```c
1: #include <stdlib.h>
2: #include <stdio.h>
3:
4: #include <X11/Xlib.h>
5: #include <X11/Xutil.h>
6: #include "openglheader.h"
7: #include <GL/glx.h>
8:
9: #include "utilities.h"
10: #include "initglxctx.h"
11: #include "myheader.h"
12:
13: Window xmywin;
14: char   terminate;
15:
16: void LoadMyShaders ( void ) { ... }
17: void DestroyMyShaders ( void ) { ... }
18: void InitMyObject ( int argc, char **argv ) { ... }
19: void DestroyMyObject ( void ) { ... }
20:
21: void InitMyGLXWindow ( int argc, char **argv,
22:                        int major, int minor, int width, int height ) { ... }
23: void DestroyMyGLXWindow ( void ) { ... }
24:
25: void Initialise ( int argc, char **argv )
26: {
27:   InitMyGLXWindow ( argc, argv, 4, 2, 480, 360 );
28:   LoadMyShaders ();
29:   InitMyObject ( argc, argv );
30: } /*Initialise*/
31:
32: void MyWinExpose ( void ) { ... }
33: void MyWinConfigNotify ( int width, int height ) { ... }
34: void MyWinButtonPress ( int button, int x, int y ) { ... }
35: void MyWinButtonRelease ( int button, int x, int y ) { ... }
36: void MyWinMotionNotify ( int x, int y ) { ... }
37: void MyWinKeyPress ( unsigned int state, unsigned int key ) { ... }
38: void MyWinClientMessage ( void ) { ... }
39:
40: void MyWinMessageProc ( XEvent *event )
41: {
42:   switch ( event->xany.type ) {
```

```
43: case Expose:
44:     if ( event->xexpose.count == 0 )
45:       MyWinExpose ();
46:     break;
47: case ConfigureNotify:
48:     MyWinConfigNotify ( event->xconfigure.width, event->xconfigure.height );
49:     break;
50: case ButtonPress:
51:     MyWinButtonPress ( event->xbutton.button,
52:                        event->xbutton.x, event->xbutton.y );
53:     break;
54: case ButtonRelease:
55:     MyWinButtonRelease ( event->xbutton.button,
56:                          event->xbutton.x, event->xbutton.y );
57:     break;
58: case MotionNotify:
59:     MyWinMotionNotify ( event->xmotion.x, event->xmotion.y );
60:     break;
61: case KeyPress:
62:     MyWinKeyPress ( event->xkey.state, event->xkey.keycode );
63:     break;
64: case ClientMessage:
65:     MyWinClientMessage ();
66:     break;
67: default:
68:     break;
69:   }
70: } /*MyWinMessageProc*/
71:
72: void MessageLoop ( void )
73: {
74:   XEvent event;
75:
76:   terminate = false;
77:   do {
78:     XNextEvent ( xdisplay, &event );
79:     if ( event.xany.window == xmywin )
80:       MyWinMessageProc ( &event );
81:   } while ( !terminate );
82: } /*MessageLoop*/
83:
84: void Cleanup ( void )
85: {
86:   DestroyMyObject ();
87:   DestroyMyShaders ();
```

```
88:   DestroyMyGLXWindow ();
89: } /*Cleanup*/
90:
91: int main ( int argc, char **argv )
92: {
93:   Initialise ( argc, argv );
94:   MessageLoop ();
95:   Cleanup ();
96:   exit ( 0 );
97: } /*main*/
```

Plik myheader.h zawiera opisy zmiennych i procedur tej konkretnej aplikacji, przydatne jeśli jej kod źródłowy będzie podzielony na osobne pliki.

Procedura Initialise, wywołana na początku działania aplikacji, przygotowuje wszystko, czego trzeba do pracy. Opisana dalej procedura InitMyGLXWindow rozpoczyna komunikację z serwerem X Window, tworzy kontekst OpenGL-a, tworzy okno aplikacji i przygotowuje je do pracy; identyfikator tego okna zostaje przypisany zmiennej xmywin. Procedury LoadMyShaders i InitMyObject przygotowują programy szaderów i tworzą obiekty składające się na scenę do narysowania.

Po powrocie z procedury Initialise następuje wywołanie procedury MessageLoop, która realizuje pętlę komunikatów. Procedura XNextEvent czeka na komunikat, a po jego nadejściu wpisuje informacje na temat zdarzenia, które spowodowało komunikat do zmiennej xevent i kończy swoje działanie. Zmienna ta jest unią 31 struktur opisujących zdarzenia; jest 35 rodzajów zdarzeń powodujących wysłanie komunikatu. Pola każdej struktury zawierają informacje odpowiednie dla danego zdarzenia. Dokładny opis tych struktur można znaleźć w dokumentacji [14] i na stronach podręcznika systemowego wyświetlanych przez polecenie man.

Po otrzymaniu komunikatu w linii 79 następuje sprawdzenie, czy komunikat jest związany z danym oknem; w aplikacji z jednym oknem jest ono niepotrzebne (ale nieszkodliwe), ale pisząc ten kod, chciałem pokazać sposób zidentyfikowania adresata w aplikacji, która otworzyła więcej niż jedno okno. Procedura MyWinMessageProc wywołana w linii 80 reaguje na komunikaty wysłane do jedynego okna naszej aplikacji, wywołując procedury obsługi poszczególnych komunikatów.

Treść procedury MyWinMessageProc jest pokazana w liniach 40–70; instrukcja

przełącznika wybiera działanie odpowiednie do rodzaju zdarzenia. W tym przykładzie obsługiwane są zdarzenia odsłonięcia okna (`Expose`), którego zawartość trzeba narysować, zmiany wymiarów (`ConfigureNotify`), po którym trzeba dostosować reprezentację obrazu do nowych wymiarów okna, naciśnięcie i zwolnienie przycisku myszy (`ButtonPress`, `ButtonRelease`), przesunięcie kursora (`MotionNotify`), naciśnięcie klawisza (`KeyPress`) i wiadomość z zewnątrz (`ClientMessage`), którą aplikacja może wysłać sama do siebie[6], na przykład w celu wykonywania obliczeń wtedy, gdy działania użytkownika nie powodują wysyłania innych komunikatów (to może pełnić rolę podobną do mechanizmu udostępnionego przez procedurę `glutIdleFunc` biblioteki FreeGLUT). Dla każdego z tych komunikatów jest wywoływana procedura, której autor (mam nadzieję) wiedział, co ta procedura ma robić.

Odsłonięcie części okna może spowodować wysłanie serii komunikatów `Expose`, jeśli odsłonięty obszar nie jest jednym prostokątem (obszar taki jest dzielony na prostokąty, każdy komunikat zawiera informację o wymiarach i położeniu jednego z nich). Warunek w linii 44 powoduje rysowanie tylko dla ostatniego komunikatu w takiej serii — system X Window wyświetli całą widoczną część obrazu w oknie.

Po zakończeniu rysowania trzeba wywołać procedurę `glXSwapBuffers`, która realizuje podwójne buforowanie — rysowanie odbywało się w niewidocznym buforze, podczas gdy w oknie było widać poprzedni obraz; po zakończeniu rysowania gotowy obraz staje się widoczny.

Uwaga: System X Window zezwala na rysowanie w oknie w trakcie obsługi dowolnego komunikatu, nie tylko komunikatu `Expose`; nie ma więc konieczności używania procedur takich jak `glutPostWindowRedisplay`[7]. Ale można to robić, posługując się na przykład procedurą pokazaną na listingu 3.8.

Procedura `MyWinMessageLoop` kończy działanie, jeśli dowolna procedura obsługi komunikatu przypisze niezerową wartość zmiennej `terminate`. Procedura `main` wywołuje wtedy procedurę `Cleanup`, której zadaniem jest posprzątanie.

Procedura `InitMyGLXWindow` przygotowująca komunikację z serwerem X Window, okno i kontekst OpenGL-a, jest pokazana na listingu 3.6. Pierwsza instrukcja wykonywana przez procedurę `InitMyGLXWindow` (linia 6) nawiązuje komunikację

[6]Wiadomość do aplikacji X Window może też wysłać inny działający w tym samym czasie program, musi tylko znać identyfikator okna, do którego ta wiadomość ma trafić.

[7]Ściślej biorąc, polecenia rysowania wydawane przez aplikację są pakowane do kolejki. System X Window wyjmuje z kolejki i wykonuje te polecenia w dogodnych dla siebie momentach.

Listing 3.6: Procedura tworzenia okna X Window do pracy z OpenGL-em

C

```
1: void InitMyGLXWindow ( int argc, char **argv,
2:                        int major, int minor, int width, int height )
3: {
4:   XSetWindowAttributes swa;
5:
6:   if ( !(xdisplay = XOpenDisplay ( "" )) )
7:     exit ( 1 );
8:   xscreen = DefaultScreen ( xdisplay );
9:   xrootwin = RootWindow ( xdisplay, xscreen );
10:  InitGLContext ( major, minor );
11:  if ( !(xcolormap = XCreateColormap ( xdisplay, xrootwin,
12:                                       xvisual, AllocNone )) )
13:    exit ( 1 );
14:  swa.colormap = xcolormap;
15:  swa.event_mask = ExposureMask | StructureNotifyMask| ButtonPressMask |
16:                   ButtonReleaseMask | PointerMotionMask | KeyPressMask;
17:  xmywin = XCreateWindow ( xdisplay, xrootwin, 0, 0, width, height,
18:                           0, 24, InputOutput, xvisual,
19:                           CWColormap | CWEventMask, &swa );
20:  XMapWindow ( xdisplay, xmywin );
21:  if ( !glXMakeCurrent ( xdisplay, xmywin, glxcontext ) )
22:    exit ( 1 );
23:  GetGLProcAddresses ();
24: } /*InitMyGLXWindow*/
25:
26: void DestroyMyGLXWindow ( void )
27: {
28:  glXMakeCurrent ( xdisplay, xwindow, NULL );
29:  glXDestroyContext ( xdisplay, glxcontext );
30:  XDestroyWindow ( xdisplay, xwindow );
31:  XCloseDisplay ( xdisplay );
32: } /*DestroyMyGLXWindow*/
```

z serwerem X Window. Jego identyfikator (wskaźnik będący pierwszym parametrem znakomitej większości procedur, których prototypy są w pliku `Xlib.h`) jest zapamiętywany w zmiennej `xdisplay` (typu `Display`; jej deklaracja jest na listingu 3.7). Jeśli komunikacja nie zaiskrzyła, to aplikacja nie ma nic więcej do powiedzenia. W przeciwnym razie w zmiennych `xscreen` i `xrootwin` zostają zapamiętane identyfikatory domyślnego ekranu i okna będącego tłem dla wszystkich innych okien wyświetlanych na tym ekranie.

W linii 10 jest wywoływana opisana dalej procedura InitGLContext, która tworzy kontekst OpenGL-a i zapamiętuje jego identyfikator w zmiennej glxcontext typu GLXContext. Następnie procedura InitMyGLXWindow tworzy okno.

W zmiennej xcolormap jest zapamiętywany wskaźnik struktury systemu X Window opisującej paletę (*colormap*) — wprawdzie nie jest ona używana jeśli wizual okna jest klasy TrueColor, ale tworząc okno, system wymaga, aby była określona. W liniach 14–16 w (pewnych) polach zmiennej swa są zapamiętywane informacje wymagane przez procedurę tworzącą okno (informacje, które można podać w pozostałych polach nie są koniecznie potrzebne). Pierwsze z tych pól zawiera identyfikator palety, a w drugim należy podać maskę bitową określającą komunikaty, których odbieraniem przez okno aplikacja jest zainteresowana. Na listingu są wybrane (prawie wszystkie) komunikaty, których procedury obsługi są pokazane na listingu 3.5; nie ma tylko (nieistniejącej) maski dla komunikatów ClientMessage, które i tak będą przesyłane gdy się pojawią.

Wywołana w liniach 17–19 procedura XCreateWindow tworzy okno o wymiarach i innych własnościach opisanych przez parametry; w szczególności klasa okna InputOutput oznacza, że to okno może przyjmować komunikaty wygenerowane przez urządzenie wejściowe i może być widoczne na ekranie. Przedostatni parametr procedury jest maską bitową określającą, które pola struktury — ostatniego parametru — mają nadane wartości. Identyfikator utworzonego okna jest przypisywany zmiennej xmywin.

Nowo utworzone okno *nie jest* widoczne na ekranie; aby się na nim pojawiło, w linii 20 jest wywołana procedura XMapWindow. W linii 21 kontekst OpenGL-a skonstruowany wcześniej przez procedurę InitGLContext jest przywiązywany do okna[8], a po „wyciągnięciu" adresów procedur OpenGL-a (najlepiej jest zrobić to za pomocą biblioteki gl3w[9]) okno jest gotowe do pracy. W tej procedurze tylko instrukcje w liniach 10 (tworzenie kontekstu), 21 (wiązanie go z oknem) i 23 (wyciąganie adresów procedur) są związane z OpenGL-em.

Na końcu działania aplikacja powinna wywołać procedurę DestroyMyGLXWindow, która likwiduje kontekst OpenGL-a i okno, po czym żegna się z serwerem X Window i wtedy aplikacja może udać się na zasłużony spoczynek.

[8] Jednocześnie kontekst jest przywiązywany do wywołującego procedurę glXMakeCurrent wątku aplikacji na CPU.

[9] I w tym przypadku biblioteka GLEW sprawia opisany wcześniej kłopot; aby jej użyć, należy utworzyć kontekst *starego* OpenGL-a (w wersji 2.1) z opcją GLX_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB. Jeśli tworząc kontekst, zażądamy jawnie nowszej specyfikacji, to pojawiają się tajemnicze błędy wykonania i aplikacja pada. Nie miałem czasu, aby znaleźć tego przyczynę. Może kiedyś to zgłębię.

Procedura tworzenia kontekstu OpenGL-a dla aplikacji systemu X Window jest pokazana na listingu 3.7. W linii 45 procedura zwraca się do biblioteki GLX o wybranie z zasobów systemu X Window tzw. wizualu o własnościach określonych przez podaną listę atrybutów. Wizual (*visual*) jest strukturą danych opisującą odwzorowanie wartości pikseli w oknie na wyświetlane na ekranie kolory; zależnie od tzw. klasy wizualu odwzorowanie to może wykorzystywać paletę lub prowadzić do otrzymania obrazów czarno-białych[10]. Atrybuty w tablicy `visattr` określają zapotrzebowanie na wizual, w którym wszystkie trzy składowe koloru mają co najmniej 8 bitów, a ponadto jest do dyspozycji podwójny bufor obrazu i bufor głębokości o co najmniej 24 bitach na każdy piksel.

W liniach 48 i 49 wybierana jest konfiguracja domyślnego tzw. bufora ramki (*framebuffer*), który będzie związany z oknem.

Listing 3.7: Procedura tworzenia kontekstu OpenGL-a

initglxctx.c

```
1: #include <stdlib.h>
2: #include <string.h>
3: #include <stdio.h>
4:
5: #include <X11/Xlib.h>
6: #include <X11/Xutil.h>
7: #include <GL/gl3w.h>
8: #include <GL/glx.h>
9:
10: #include "initglxctx.h"
11:
12: Display    *xdisplay;
13: int        xscreen;
14: Window     xrootwin;
15: Visual     *xvisual;
16: Colormap   xcolormap;
17: GLXContext glxcontext;
18:
19: typedef GLXContext (*PFNGLXCREATECONTEXTATTRIBSARBPROC)
20:                    ( Display *dpy, GLXFBConfig config,
21:                      GLXContext share_context, Bool direct,
22:                      const int *attrib_list );
23:
24: static void InitGLContext ( int major, int minor )
```

[10]W aplikacjach nowego OpenGL-a wymagany jest wizual klasy `TrueColor`, w którym piksele mają składowe RGB wyświetlane bezpośrednio na ekranie.

```
25: {
26:   PFNGLXCREATECONTEXTATTRIBSARBPROC glXCreateContextAttribsARB;
27:   int nelements;
28:   int visattr[] =
29:     { GLX_RGBA,
30:       GLX_DOUBLEBUFFER,
31:       GLX_RED_SIZE,    8,
32:       GLX_GREEN_SIZE,  8,
33:       GLX_BLUE_SIZE,   8,
34:       GLX_DEPTH_SIZE, 24,
35:       None };
36:   int ctxattr[] =
37:     { GLX_CONTEXT_MAJOR_VERSION_ARB, 0,
38:       GLX_CONTEXT_MINOR_VERSION_ARB, 0,
39:       GLX_CONTEXT_PROFILE_MASK_ARB, GLX_CONTEXT_CORE_PROFILE_BIT_ARB,
40:       GLX_CONTEXT_FLAGS_ARB, GLX_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB,
41:       None };
42:   XVisualInfo *vii;
43:   GLXFBConfig *fbc;
44:
45:   if ( !(vii = glXChooseVisual( xdisplay, 0, visattr )) )
46:     exit ( 1 );
47:   xvisual = vii->visual;
48:   ctxattr[1] = major;  ctxattr[3] = minor;
49:   fbc = glXChooseFBConfig ( xdisplay, xscreen, 0, &nelements );
50:   if ( (glXCreateContextAttribsARB =
51:          (PFNGLXCREATECONTEXTATTRIBSARBPROC)glXGetProcAddress
52:              ( (const GLubyte*)"glXCreateContextAttribsARB" )) ) {
53:     if ( !(glxcontext = glXCreateContextAttribsARB ( xdisplay,
54:                                           *fbc, 0, 1, ctxattr )) )
55:       exit ( 1 );
56:   }
57:   else
58:     exit ( 1 );
59: } /*InitGLContext*/
```

W liniach 50–52 korzystamy z procedury `glXGetProcAddress` w celu otrzymania adresu procedury `glXCreateContextAttribsARB`, która konstruuje kontekst (nowego) OpenGL-a. Ta procedura realizuje rozszerzenie specyfikacji GLX 1.4[11], więc teoretycznie może jej nie być i wtedy aplikacja kapituluje.

[11]Specyfikacja GLX 1.4, ostatnia dostępna, prawie w całości jest związana ze starym OpenGL-em. Kontekst dla nowego OpenGL-a trzeba utworzyć przy użyciu rozszerzenia. Wypada wyrazić ubolewanie, że nie jest ono w tej specyfikacji udokumentowane i że nie ma nowszej specyfikacji, w pełni dostosowanej do nowego OpenGL-a.

Procedura glXCreateContextAttribsARB konstruuje kontekst o postulowanych przez aplikację własnościach, w szczególności realizujący wersję standardu o podanym numerze, który w linii 48 został umieszczony w liście wymaganych przez aplikację atrybutów kontekstu[12] jest pokazana na listingu 3.6. Identyfikator kontekstu jest zapamiętywany w zmiennej glxcontext.

Listing 3.8 przedstawia wcześniej wspomnianą procedurę PostExposeEvent, która może być używana w roli analogicznej do glutPostWindowRedisplay, oraz procedurę PostClientMessageEvent, która umieszcza w kolejce systemu X Window komunikat ClientMessage.

Listing 3.8: Procedury PostExposeEvent i PostClientMessageEvent

C

```c
1: void PostExposeEvent ( Window win, int width, int height )
2: {
3:   XExposeEvent ev;
4:
5:   memset ( &ev, 0, sizeof(ev) );
6:   ev.type = Expose;  ev.send_event = True;
7:   ev.display = xdisplay;  ev.window = win;
8:   ev.width = width;  ev.height = height;
9:   XSendEvent ( xdisplay, win, True, ExposureMask, (XEvent*)&ev );
10: } /*PostExposeEvent*/
11:
12: void PostClientMessageEvent ( Window win, Atom message_type,
13:                               int format, void *data )
14: {
15:   XClientMessageEvent ev;
16:
17:   memset ( &ev, 0, sizeof(ev) );
18:   ev.type = ClientMessage;  ev.send_event = True;
19:   ev.display = xdisplay;  ev.window = win;
20:   ev.message_type = message_type;  ev.format = format;
21:   if ( data )
22:     memcpy ( ev.data.b, data, 20*sizeof(char) );
23:   XSendEvent ( xdisplay, win, True, 0, (XEvent*)&ev );
24: } /*PostClientMessageEvent*/
```

Pierwszy parametr procedury PostClientMessageEvent jest identyfikatorem okna — adresata komunikatu. Drugi parametr jest typu Atom; atom jest to liczba

[12] Jest to bardzo prosta procedura; nie gwarantuję, że na każdym komputerze zadziała poprawnie, ale zapewiniam, że na co najmniej pięciu zadziałała.

całkowita używana jako identyfikator struktury danych. Pewien zbiór „gotowych" atomów (zobacz plik X11/Xatom.h) służy do identyfikowania struktur zdefiniowanych przez system X Window, ale system ten, przesyłając komunikat ClientMessage, nie interpretuje atomu będącego wartością pola message_type. Aplikacja X Window może zatem w komunikatach ClientMessage używać do swoich celów dowolnych identyfikatorów. Parametr type powinien mieć wartość 8, 16 albo 32, a parametr data powinien wskazywać tablicę z 20 bajtami, 10 liczbami 16-bitowymi lub z pięcioma liczbami 32-bitowymi, które zostaną przesłane w komunikacie. Wysyłanie komunikatów ClientMessage przez aplikację do siebie jest jednym ze sposobów zapewnienia, że odpowiednia procedura aplikacji zostanie wywołana, gdy nie ma innych zadań do wykonania[13].

Jak wspomniałem wcześniej, komunikaty ClientMessage mogą do siebie nawzajem przesyłać różne aplikacje działające jednocześnie. W tym celu muszą się najpierw skomunikować i przesłać sobie (np. przez łącze komunikacyjne) identyfikatory okien, które mają odbierać te komunikaty. Tego tematu rozwijać tu nie będziemy.

Uzupełnienia — wymiary ekranu

Biblioteki FreeGLUT, GLFW i GLX umożliwiają otrzymanie przez aplikację informacji na temat wymiarów ekranu — szerokości i wysokości w pikselach i w milimetrach — co umożliwia obliczenie współczynnika aspektu (zobacz rozdz. 6). Aplikacja biblioteki FreeGLUT może w tym celu wywołać procedurę glutGet kolejno z parametrami GLUT_SCREEN_WIDTH, GLUT_SCREEN_HEIGHT, GLUT_SCREEN_WIDTH_MM i GLUT_SCREEN_HEIGHT_MM. Wartość powrotna (typu int) to odpowiednio szerokość i wysokość ekranu w pikselach oraz szerokość i wysokość ekranu w milimetrach.

Aplikacja biblioteki GLFW uzyska wymiary ekranu w pikselach za pomocą procedury glfwGetVideoMode; jej parametrem jest zmienna typu (zamkniętego) GLFWmonitor, a wartość powrotna jest wskaźnikiem struktury typu GLFWvidmode, której pola zawierają m.in. szerokość i wysokość ekranu w pikselach. Fizyczne wymiary ekranu (w milimetrach) podaje procedura glfwGetMonitorPhysicalSize.

W systemie X Window mamy do dyspozycji procedury (a raczej makra) DisplayWidth, DisplayHeight, DisplayWidthMM i DisplayHeightMM, które podają wymiary ekranu w pikselach i w milimetrach. Trzeba jednak pamiętać, że informacja o wymiarach fizycznych nie zawsze jest dostępna i nie zawsze jest

[13] A ściślej, gdy komunikat ClientMessage znajdzie się na początku kolejki.

rzetelna. Na przykład z informacji podanych przez system okien wynika, że (czternastocalowy) ekran mojego laptopa ma dwadzieścia cali szerokości. Jeśli używam rzutnika, to obraz jest znacznie większy, ale komputer nie ma żadnej możliwości zmierzenia go. Dlatego najbezpieczniej jest wymiary ekranu zapisać w pliku konfiguracyjnym, który aplikacja przeczyta na początku swojego działania. Fizyczne wymiary ekranu użytkownik może podać podczas instalowania aplikacji.

*Uzupełnienia — technologia Optimus

Poważnym problemem ludzkości jest nadmierne zużycie energii i dlatego są wynajdowane rozmaite sposoby jej oszczędzania. Innym poważnym problemem jest konkurencja między producentami sprzętu, którzy nie bardzo ułatwiają sobie nawzajem działalność. Jedną z konsekwencji tego stanu rzeczy jest stworzona przez firmę NVIDIA technologia Optimus, która w nowoczesnych[14] laptopach realizuje współpracę GPU tej firmy z podukładem graficznym wbudowanym w CPU firmy Intel. W tej technologii GPU jest czynna (i zużywa energię) tylko w czasie pracy aplikacji używających GPU. Obrazy tworzone przez GPU są przesyłane do okien systemu X Window, którego działanie organizuje i obsługuje CPU, posługując się swoim podukładem graficznym. Większość wersji tego podukładu zazwyczaj nie realizuje standardu OpenGL w wersji 4.0 i dalszych, bo nie po to powstały.

W komunikacji CPU i GPU następują opóźnienia, przy czym nie chodzi tu o czas, tylko o pełną sekwencję komunikatów, które muszą być przesłane w obie strony, aby wszystkie dane dotarły na właściwe miejsca. Obraz widoczny w oknie podczas wyświetlania animacji (np. przez grę) jest przez to opóźniony, zazwyczaj o jedną klatkę[15]. Również informacja o tym, że użytkownik zmienił wymiary okna, dociera do GPU z opóźnieniem. W rezultacie obraz wyświetlony (przez procedurę `DisplayFunc` aplikacji FreeGLUTa, procedurę `Redraw` aplikacji GLFW lub procedurę obsługi komunikatu `Expose` aplikacji X Window) bezpośrednio po zmianie tych wymiarów składa się z fragmentów obrazu o poprzednich wymiarach okna i z obszarów o zawartości nieokreślonej.

Jeśli aplikacja wyświetla obrazy wiele razy na sekundę, to nieudany obraz w oknie jest od razu zastępowany obrazem takim, jak trzeba. Jeśli jednak zawartość okna jest zmieniana tylko w odpowiedzi na działania użytkownika, to efekt zmiany wymiarów okna wygląda nieciekawie. Aby temu przeciwdziałać, trzeba po zmianie wielkości okna wymusić *kilkakrotne* narysowanie obrazu. Potrzeba kilkakrotnego

[14] „Mnie wiele można zarzucić, no nie że ja nowoczesny.” — Kaźmirz Pawlak, *Nie ma mocnych.*
[15] Czyli typowo o 1/60s.

odświeżenia obrazu w oknie nie oznacza konieczności poniesienia pełnego kosztu kilku rysowań; można wykonać obraz tylko raz, w tzw. pozaekranowym buforze ramki (*off-screen framebuffer*), a potem tylko go przesłać tyle razy ile trzeba do okna, co zabiera znikomą ilość czasu. Sposób tworzenia obrazu poza ekranem jest opisany w rozdziale 18 i dalszych.

Wielokrotne rysowanie można zrealizować tak: deklarujemy globalną zmienną typu całkowitego, nazwijmy ją `opti`. W aplikacji FreeGLUTa do procedury `ReshapeFunc` dodajemy instrukcję przypisania `opti = 4;`, a do procedury `DisplayFunc` dopisujemy instrukcję

```
if ( opti > 0 ) {
  opti --;
  glutPostWindowRedisplay ( WindowHandle );
}
```

Taką samą instrukcję przypisania trzeba dodać do procedury `ReshapeFunc` w aplikacji biblioteki GLFW, a druga potrzebna zmiana jest w procedurze `Redraw`: przypisanie wartości zmiennej `redraw` powinno być obłożone warunkiem

```
if ( opti > 0 ) opti --;
if ( opti == 0 ) redraw = false;
```

Wreszcie, aplikacja systemu X Window po otrzymaniu komunikatu `ConfigureNotify` powinna zapamiętać (np. w zadeklarowanych w tym celu zmiennych `window_width` i `window_height`) nowe wymiary okna i nadać zmiennej `opti` wartość 3, a procedura obsługi komunikatu `Expose`, jeśli wartość tej zmiennej jest dodatnia, powinna ją zmniejszyć i wysłać do okna kolejny komunikat `Expose`, na przykład tak:

```
if ( opti > 0 ) {
  opti --;
  PostExposeEvent ( xmywin, window_width, window_height );
}
```

Nie jestem pewien, dlaczego w aplikacjach bibliotek FreeGLUT i GLFW rozwiązanie problemu wymaga aż czterokrotnego przerysowania obrazu, a aplikacji X Window wystarczą tylko trzy razy.

Biorąc pod uwagę kłopoty z tą technologią (w tym także z bardzo nietrywialną na

laptopie z Optimusem instalacją sterownika GPU i związaną z tym konfiguracją systemu X Window), zamieszczony na stronie firmy NVIDIA tekst, cytuję

> *NVIDIA's Optimus technology works right out of the box. There's absolutely no setup or configuration required—it's that simple. Optimus technology will automatically optimize your notebook PC, providing you the outstanding graphics performance you need, when you need it ...*

uważam za zapis myślenia życzeniowego i propagandę (ale to nie znaczy, że uważam technologię Optimus za bezwartościową, co to, to nie). Posiadaczom laptopów z technologią Optimus szczerze życzę dużo cierpliwości i powodzenia.

4. Utensylia

Aplikacje OpenGL-a potrzebują procedur pomocniczych, z których wiele jest na początek opisanych w tym rozdziale. Najlepiej umieścić je w osobnym pliku źródłowym (np. `utilities.c`) i napisać odpowiedni plik nagłówkowy (`utilities.h`). Będą one użyte we wszystkich opisanych dalej aplikacjach.

Wypisanie informacji o wersji

Procedura `PrintGLVersion` wypisuje napis zawierający informację o wersji standardu OpenGL i wersji języka GLSL obsługiwanych przez środowisko, w którym została wywołana aplikacja (tj. przez zainstalowane w tym środowisku biblioteki) oraz o producencie implementacji standardu. Warto ją wywołać na początku działania aplikacji (ale po uzyskaniu dostępu do procedury `glGetString`, np. przez wywołanie procedury `glewInit` albo `gl3wInit`), aby dowiedzieć się, czy aplikacja nie wymaga za dużo.

Listing 4.1: Wypisanie informacji na temat wersji OpenGL-a

C

```c
1: void PrintGLVersion ( void )
2: {
3:   printf ( "OpenGL %s, GLSL %s\n",
4:            glGetString ( GL_VERSION ),
5:            glGetString ( GL_SHADING_LANGUAGE_VERSION ) );
6: } /*PrintGLVersion*/
```

Reakcje na błąd

Procedury `ExitOnError` i `ExitIfGLError` pokazane na listingu 4.2 są niesłychanie przydatne podczas uruchamiania programu. Wywołanie pierwszej z nich powoduje wypisanie tekstu przekazanego jako parametr i zatrzymanie programu, co umożliwia zwięźlejszy zapis w programie reakcji na błąd uniemożliwiający dalsze działanie.

Wywołanie drugiej z tych procedur powinno być umieszczone po instrukcjach wywołujących procedury OpenGL-a; *jeśli* nastąpił błąd, to procedura ta zatrzyma program, ale przedtem wyświetli komunikat diagnostyczny, który może pomóc w zrozumieniu natury błędu i w jego naprawieniu. Warto ją wywoływać dosyć często, bo to ułatwia odkrycie miejsca i przyczyny błędu. Jeśli OpenGL nie odnotował błędu, to następuje powrót z procedury i aplikacja działa dalej jak gdyby nigdy nic.

Listing 4.2: Procedury ExitOnError i ExitIfGLError

C

```c
void ExitOnError ( const char *msg )
{
  fprintf ( stderr, "Error: %s\n", msg );
  exit ( 1 );
} /*ExitOnError*/

void ExitIfGLError ( const char *msg )
{
  GLenum err;

  if ( (err = glGetError ()) != GL_NO_ERROR ) {
    fprintf ( stderr, "Error %d: %s, %s\n",
              err, gluErrorString ( err ), msg );
    exit ( 1 );
  }
} /*ExitIfGLError*/
```

Procedura `ExitIfGLError` wywołuje procedurę `glGetError`, której wartość wskazuje, czy nastąpił błąd i jeśli tak, to jaki (jeśli był więcej niż jeden błąd, to procedura podaje kod pierwszego z nich), zaś procedura `gluErrorString` tłumaczy liczbowy kod błędu na język angielski. Parametr procedury `ExitIfGLError` służy do wskazywania miejsca wykrycia błędu w aplikacji.

Reprezentacje kodów źródłowych szaderów

Nie zwracając na razie uwagi na składnię języka GLSL, zobaczmy kod źródłowy przykładowego szadera na listingu 4.3. Szader ten można umieścić w pliku na dysku; wtedy aplikacja musi ten plik przeczytać, skompilować i połączyć z innymi szaderami w program szaderów[1].

Są dwa drobne kłopoty. Po pierwsze, szadery mogą mieć spore wspólne części kodu źródłowego (takie jak widoczny w przykładzie opis bloku zmiennych jednolitych `MyGC`). Chciałoby się nie powielać takich fragmentów kodu.

[1] Nazwy i rozszerzenia nazw plików źródłowych szaderów są w zasadzie obojętne, bo to aplikacja jest odpowiedzialna za przeczytanie tych plików i przekazanie ich treści do kompilacji, choć nabiera to znaczenia, jeśli chcemy kompilować szadery do formatu SPIR-V, o czym będzie dalej. Kompilator rozpoznaje typ szadera po rozszerzeniu, które ma mieć postać `.vert`, `.tesc`, `.tese`, `.geom`, `.frag` albo `.comp`. Ja daję rozszerzenia podwójne, np. `.glsl.vert`. Można tak skonfigurować edytor, aby dla plików o powyższych rozszerzeniach np. wyróżniał (odpowiednim kolorem) literały, komentarze i słowa kluczowe języka GLSL, co ułatwia programowanie.

Listing 4.3: Przykładowy szader wierzchołków

GLSL

```
1: #version 430
2:
3: uniform MyGC {
4:   mat4 pm;      /* projection matrix */
5:   vec4 fg, bk;  /* foreground and background */
6: } gc;
7:
8: layout(location=0) in vec4 in_Position;
9:
10: void main ( void )
11: {
12:   gl_Position = gc.pm * in_Position;
13: } /*main*/
```

W języku C do tego celu służy dyrektywa #include; niestety, nie jest ona dostępna w GLSL[2]. Wielokrotne pisanie czegokolwiek jest procesem podatnym na błędy. Dlatego możemy podzielić kod w GLSL-u na krótsze fragmenty i składać źródła szaderów z takich fragmentów. Ale wtedy będziemy mieli na dysku wiele króciutkich plików, których obecność obok pliku skompilowanej aplikacji w C musimy zapewnić — i to jest drugi kłopot, istotny zwłaszcza wtedy, gdy chcemy program rozpowszechniać.

Kompilator języka GLSL będący częścią implementacji OpenGL-a umożliwia podanie źródeł szadera w postaci ciągu napisów ASCIIZ. Należy przekazać tablicę wskaźników do tych napisów; można je przeczytać z plików, co jest wygodne podczas tworzenia, uruchamiania i testowania szaderów. Natomiast po ich uruchomieniu lepiej jest je w program wbudować, to znaczy umieścić odpowiednie napisy w kodzie źródłowym aplikacji w C. Ten sam szader, co na listingu 4.3, wbudowany w źródła aplikacji, może wyglądać tak, jak na listingu 4.4.

Zauważmy, że kod źródłowy szadera pozostał perfekcyjnie powcinany i czytelny, a przy tym spacje wcięć i komentarze nie przejdą do kodu skompilowanej aplikacji[3], a zatem nie powiększą objętości jej pliku binarnego. Poszczególne napisy można umieścić w tablicach „linii” kodu innych szaderów, co efektywnie zastępuje nieodżałowaną dyrektywę #include. Należy tylko pamiętać, że kompilator C może ograniczyć długość napisu (w ANSI C napis może mieć co najwyżej 508 znaków), a więc dłuższe szadery trzeba podzielić na więcej kawałków.

[2]Istnieje rozszerzenie standardu udostępniające dyrektywę #include w GLSL (nieidentyczną z dyrektywą #include preprocesora C), ale przecież chcemy dbać o przenośność.

[3]¡No pasarán!

Listing 4.4: Ten sam przykładowy szader wierzchołków

GLSL w C

```
1: static const char sh_version430[] =
2:   "#version 430\n"
3:   ;
4: static const char sh_gc[] =
5:   "uniform MyGC {"
6:     "mat4 pm;"      /* projection matrix */
7:     "vec4 fg, bk;"  /* foreground and background */
8:   "} gc;"
9:   ;
10: static const char sh_vert[] =
11:   "layout(location=0) in vec4 in_Position;"
12:
13:   "void main ( void )"
14:   "{"
15:     "gl_Position = gc.pm * in_Position;"
16:   "}" /*main*/
17:   ;
18: static const char *shader_v_src[] =
19:   { sh_version430, sh_gc, sh_vert };
```

Kompilowanie szaderów i łączenie programów

Procedura `CompileShaderStrings` pokazana na listingu 4.5 otrzymuje następujące parametry: `shader_type` musi być jedną z sześciu stałych określających typ szadera: `GL_VERTEX_SHADER`, `GL_TESS_CONTROL_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_GEOMETRY_SHADER`, `GL_FRAGMENT_SHADER` lub `GL_COMPUTE_SHADER`, `nsl` jest liczbą napisów — fragmentów kodu źródłowego szadera, parametr `srclines` jest wskaźnikiem tablicy wskaźników początków tych napisów, które (w tym podprogramie) muszą być zakończone bajtem zerowym (czyli są ASCIIZ).

Zadaniem procedury `CompileShaderStrings` jest utworzenie szadera, skompilowanie go i wypisanie komunikatów diagnostycznych, jeśli wystąpił błąd kompilacji. Pierwsze zadanie wykonuje procedura `glCreateShader`, która zwraca identyfikator szadera. Jego kod źródłowy rejestruje procedura `glShaderSource`.

Kompilacja jest wykonywana przez procedurę `glCompileShader`. Jeśli kompilator wykrył błędy, to jego komunikat diagnostyczny jest wyciągany przez procedurę `glGetShaderInfoLog` (wcześniej ustala się długość tego komunikatu) i wypisywany do pliku `stderr`. Jeśli kompilacja zakończyła się sukcesem, to identyfikator szadera (liczba dodatnia) jest zwracany jako wartość procedury. Aby

Listing 4.5: Procedura CompileShaderStrings

C
```
1: GLuint CompileShaderStrings ( GLenum shader_type, int nsl,
2:                               const GLchar **srclines )
3: {
4:   GLuint shader_id;
5:   GLint  logsize;
6:   GLchar *log;
7:
8:   if ( (shader_id = glCreateShader ( shader_type )) != 0 ) {
9:     glShaderSource ( shader_id, nsl, srclines, NULL );
10:    glCompileShader ( shader_id );
11:    glGetShaderiv ( shader_id, GL_INFO_LOG_LENGTH, &logsize );
12:    if ( logsize > 1 ) {
13:      if ( (log = malloc ( logsize+1 )) != 0 ) {
14:        glGetShaderInfoLog ( shader_id, logsize, &logsize, log );
15:        fprintf ( stderr, "%s\n", log );
16:        free ( log );
17:      }
18:    }
19:  }
20:  ExitIfGLError ( "CompileShaderStrings" );
21:  return shader_id;
22: } /*CompileShaderStrings*/
```

użyć tej procedury do skompilowania szadera z listingu 4.4, należy ją wywołać w taki sposób:

```
s_id[i] = CompileShaderStrings ( GL_VERTEX_SHADER, 3, shader_v_src );
```

W tablicy `s_id`, której elementy są typu `GLuint`, zapamiętujemy identyfikatory wszystkich szaderów, które chcemy połączyć w program; tablicę tę przekażemy następnie jako parametr procedury `LinkShaderProgram`.

Listing 4.6 przedstawia procedurę, która czyta pliki o podanych nazwach, umieszcza ich zawartość w odpowiedniej tablicy i wywołuje opisaną wyżej procedurę `CompileShaderStrings`. Pierwszy parametr określa typ szadera, drugi liczbę plików, a trzeci jest tablicą napisów ASCIIZ, które są nazwami tych plików.

Instrukcje w liniach 10–20 znajdują długości tych plików (w bajtach). Następnie jest alokowany odpowiedni bufor i w pętli w liniach 23–32 pliki są czytane do bufora i tworzona jest tablica wskaźników do początków przeczytanych napisów. Procedura zwraca identyfikator skompilowanego szadera.

Listing 4.6: Procedura CompileShaderFiles

C

```c
GLuint CompileShaderFiles ( GLenum shader_type, int nfiles,
                            const char **filenames )
{
  GLuint shader_id = 0;
  FILE   *f;
  int    i;
  GLint  *fsize = NULL, totalsize;
  GLchar *src = NULL, **srclines = NULL;

  if ( !(fsize = malloc ( nfiles*sizeof(GLint) )) ||
       !(srclines = malloc ( nfiles*sizeof(GLchar*) )) )
    goto way_out;
  for ( i = 0, totalsize = nfiles;  i < nfiles;  i++ ) {
    if ( !(f = fopen ( filenames[i], "rb" )) )
      goto way_out;
    fseek ( f, 0, SEEK_END );
    fsize[i] = ftell ( f );
    totalsize += fsize[i];
    fclose ( f );
  }
  if ( !(src = malloc ( totalsize )) )
    goto way_out;
  for ( i = 0, totalsize = 0;  i < nfiles;  i++ ) {
    if ( !(f = fopen ( filenames[i], "rb" )) )
      goto way_out;
    srclines[i] = &src[totalsize];
    if ( fread ( srclines[i], sizeof(char), fsize[i], f ) != fsize[i] )
      goto way_out;
    srclines[i][fsize[i]] = 0;
    totalsize += fsize[i]+1;
    fclose ( f );
  }
  shader_id = CompileShaderStrings ( shader_type, nfiles,
                                     (const GLchar**)srclines );
way_out:
  if ( fsize )    free ( fsize );
  if ( srclines ) free ( srclines );
  if ( src )      free ( src );
  return shader_id;
} /*CompileShaderFiles*/
```

Procedura LinkShaderProgram na listingu 4.7 tworzy nowy program szaderów,

dopisuje do niego do niego skompilowane szadery i łączy program szaderów. Parametr nsh jest liczbą szaderów, których identyfikatory (zwrócone jako wartości procedury glCreateShader, a następnie przekazane przez CompileShaderStrings) są zapamiętane w tablicy shaders. Procedura glCreateProgram tworzy nowy, początkowo pusty obiekt programu i podaje jego identyfikator. Szadery są doczepiane do programu przez glAttachShader. Łączenie wykonuje procedura glLinkProgram; jeśli wystąpiły błędy, to procedura wyciąga tekst diagnostyczny i wypisuje go do pliku stderr. Wartością zwracaną przez procedurę LinkShaderProgram jest identyfikator programu. Program może być wmontowany w potok przetwarzania grafiki przez wywołanie procedury glUseProgram z tym identyfikatorem podanym jako parametr.

Listing 4.7: Procedura LinkShaderProgram

C

```
1: GLuint LinkShaderProgram ( int nsh, const GLuint *shaders )
2: {
3:   GLuint program_id;
4:   int    i;
5:   GLint  logsize;
6:   GLchar *log;
7:
8:   if ( (program_id = glCreateProgram ()) ) {
9:     for ( i = 0; i < nsh; i++ )
10:      glAttachShader ( program_id, shaders[i] );
11:    glLinkProgram ( program_id );
12:    glGetProgramiv ( program_id, GL_INFO_LOG_LENGTH, &logsize );
13:    if ( logsize > 1 ) {
14:      if ( (log = malloc ( logsize+1 )) ) {
15:        glGetProgramInfoLog ( program_id, logsize, &logsize, log );
16:        fprintf ( stderr, "%s\n", log );
17:        free ( log );
18:      }
19:    }
20:  }
21:  ExitIfGLError ( "LinkShaderProgram" );
22:  return program_id;
23: } /*LinkShaderProgram*/
```

Po zbudowaniu programu szaderów poszczególne obiekty szaderów w zasadzie przestają być potrzebne i można je zlikwidować; służy do tego procedura glDeleteShader (podobnie, do likwidowania niepotrzebnych programów służy procedura glDeleteProgram). Ale jeden szader może być częścią wielu programów

— wystarczy go skompilować tylko raz. Ponadto programy można przebudowywać w trakcie działania aplikacji, usuwając szadery z programu za pomocą procedury `glDetachShader` i dodając nowe (skompilowane) szadery za pomocą `glAttachShader`. Po takiej zmianie program trzeba na nowo połączyć za pomocą procedury `glLinkProgram`[4].

*Uzupełnienia — SPIR-V

SPIR-V jest to binarny format (a właściwie język) dla szaderów i innych programów działających na GPU. Kod źródłowy szaderów napisanych w GLSL-u (lub w innym języku) można skompilować przy użyciu zewnętrznego kompilatora, otrzymując ich kod w SPIR-V i rozpowszechniać je w takiej postaci[5]; jest ona niezależna od architektury GPU, czyli tak przetworzone szadery powinny działać na GPU różnych producentów[6]. Należy mieć na uwadze, że

- obecnie język SPIR-V nie zawiera dokładnych odpowiedników wszystkich konstrukcji GLSL-a, a sam zewnętrzny kompilator nie jest jeszcze doskonały[7], w związku z czym nie realizuje pełnej specyfikacji GLSL (o czym lojalnie ostrzega), wskutek czego
- nie każda konstrukcja języka GLSL jest dopuszczalna w szaderach, które mają być skompilowane do SPIR-V (niedopuszczalne są m.in. wskaźniki do procedur i pewne postaci kwalifikatorów układu zmiennych),
- kod w SPIR-V jest zwykle dłuższy niż kod źródłowy szadera w GLSL,
- korzyścią z tej postaci jest skrócenie czasu uruchamiania aplikacji (o czas kompilowania szaderów, raczej niezbyt długi) i możliwość pisania aplikacji dla okrojonych środowisk, w których nie ma kompilatora GLSL-a, działających na przykład w urządzeniach przenośnych (telefonach),
- SPIR-V jest podstawową (a właściwie wymaganą) reprezentacją szaderów w aplikacjach standardu Vulkan.

W opisanych dalej aplikacjach będziemy korzystać z procedur opisanych wcześniej w tym rozdziale. Umożliwiają one aplikacji czytanie plików źródłowych w GLSL

[4]Indeksy zmiennych jednolitych i ich bloków zmieniają się przy tym, zatem po każdym połączeniu programu trzeba je na nowo z niego odczytać.

[5]nieczytelnej dla ludzi, choć możliwej do zdeasemblowania

[6]co koniecznie trzeba przetestować *przed* rozpowszechnianiem programu

[7]Ufam, że to się zmieni.

i ich kompilowanie i łączenie; taka postać jest najłatwiejsza do uruchamiania szaderów i aplikacji. Po uruchomieniu i przetestowaniu można jednak posłużyć się opisanym niżej sposobem, aby skompilować szadery razem z aplikacją, a nawet wbudować do niej szadery skompilowane (można je przetworzyć na tekst w C, zawierający deklaracje tablic liczb typu unsigned int podanych na przykład w układzie szesnastkowym), dzięki czemu po skompilowaniu aplikacji z szaderami powstanie jeden plik binarny.

Kompilator i sposób jego instalowania można znaleźć na stronie [7]; plik wykonywalny nazywa się glslangValidator. Kod źródłowy w GLSL należy zapisać w pliku, którego nazwa kończy się jednym z sześciu rozszerzeń: .vert, .tesc, .tese, .geom, .frag albo .comp, po którym kompilator rozpoznaje rodzaj szadera. Przypuśćmy, że kod szadera w GLSL-u jest w pliku app1b0.glsl.vert i nie ma w nim błędów. Polecenie kompilacji tego szadera może mieć postać

```
glslangValidator -G app1b0.glsl.vert -o app1b0.vert.spv
```

Opcja -G wybiera postać kodu wynikowego odpowiednią do współpracy z aplikacją OpenGL-a (alternatywna opcja -V obowiązuje dla aplikacji biblioteki Vulkan). Po skompilowaniu powstaje plik app1b0.vert.spv.

W aplikacji, która ma korzystać z szaderów skompilowanych w ten sposób należy podczas inicjalizacji sprawdzić, czy obecna w systemie implementacja OpenGL-a jest zgodna ze specyfikacją co najmniej 4.6, a jeśli nie, to czy obsługuje rozszerzenie standardu o nazwie "GL_ARB_gl_spirv". Możemy w tym celu użyć

Listing 4.8: Procedura IsGLExtensionPresent

```C
 1: char IsGLExtensionPresent ( const char *name )
 2: {
 3:   GLint         extn, i;
 4:   const GLubyte *ext;
 5:
 6:   glGetIntegerv ( GL_NUM_EXTENSIONS, &extn );
 7:   for ( i = 0; i < extn; i++ ) {
 8:     ext = glGetStringi ( GL_EXTENSIONS, i );
 9:     if ( !strcmp ( name, (char*)ext ) )
10:       return true;
11:   }
12:   return false;
13: } /*IsGLExtensionPresent*/
```

procedury przedstawionej na listingu 4.8, wywołując ją z parametrem — napisem będącym nazwą rozszerzenia. W linii 6 procedura dowiaduje się, ile rozszerzeń jest obecnych w implementacji, a następnie w pętli otrzymuje adresy kolejnych nazw rozszerzeń i porównuje je z nazwą podaną jako parametr. Procedura kończy działanie po znalezieniu podanej nazwy lub po jej nieznalezieniu.

Jeśli rozszerzenie "GL_ARB_gl_spirv" jest obecne, to aplikacja może działać dalej. W tym celu będzie potrzebować procedury glSpecializeShaderARB, której adres trzeba „wyciągnąć" indywidualnie (bo tego nie robi procedura glewInit ani gl3wInit, ich zadanie polega na uzyskaniu dostęu do procedur niebędących częściami rozszerzeń[8]). W aplikacji trzeba zadeklarować typ i zmienną

```
typedef void (*PFNGLSPECIALIZESHADERARB) (GLuint shader,
            const GLchar* pEntryPoint, GLuint numSpecializationConstants,
            const GLuint* pConstantIndex, const GLuint* pConstantValue);
PFNGLSPECIALIZESHADERARB glSpecializeShaderARB = NULL;
```

Zmiennej glSpecializeShaderARB należy przypisać wartość podaną (zależnie od używanej biblioteki współpracującej z systemem okien) przez procedurę glXGetProcAddress, glutGetProcAddress albo glfwGetProcAddress.

Na listingu 4.9 są pokazane dwie procedury; pierwsza z nich tworzy szader z danych przekazanych jako parametr, a druga czyta dane z pliku (który powinien był zostać utworzony przez program glslangValidator) i wywołuje pierwszą procedurę. Każda z tych procedur zwraca identyfikator szadera, który należy zapamiętać w tablicy, a następnie wywołać procedurę LinkShaderProgram z listingu 4.7. Dalsze postępowanie jest takie samo jak z szaderami otrzymanymi w wyniku kompilacji źródeł w GLSL podanych przez aplikację.

Procedura glShaderBinary (wywoływana zamiast glShaderSource) może otrzymać tablicę z wieloma identyfikatorami szaderów, choć w przykładzie jest tylko jeden. Celem procedury glSpecializeShaderARB jest określenie nazwy

[8]Dodatkowa przewaga biblioteki gl3w nad GLEW polega na włączeniu (poprzez plik gl3w.h) pełnego pliku nagłówkowego glcorearb.h, który zawiera deklaracje i makrodefinicje związane ze wszystkimi wersjami standardu OpenGL. W moim przypadku wersja obsługiwana przez sprzęt i sterownik to 4.5 z pewnymi rozszerzeniami, które stały się częścią standardu 4.6. Próbując zbudować aplikację pierwszą C (rozdział 11) z szaderami SPIR-V i z biblioteką GLEW, otrzymałem błąd kompilacji spowodowany brakiem definicji stałej GL_SHADER_BINARY_FORMAT_SPIR_V_ARB, potrzebnej jako parametr procedury glShaderBinary, a to dlatego, że plik nagłówkowy glew.h zawierał definicje tylko dla wersji OpenGL 1.0–4.5. Aby pokonać tę przeszkodę, wystarczy dopisać odpowiednią makrodefinicję do kodu aplikacji. Skąd wziąć stałą? Z dokumentacji rozszerzenia ze strony [6] lub z pliku glcorearb.h. Chyba jednak lepiej jest używać biblioteki gl3w.

Listing 4.9: Procedury CreateSPIRVShader i LoadSPIRVFile

C

```
1: GLuint CreateSPIRVShader ( GLenum shader_type,
2:                            int size, const GLuint *spirv )
3: {
4:   GLuint shader_id;
5:   GLint  status;
6:
7:   if ( (shader_id = glCreateShader ( shader_type )) != 0 ) {
8:     glShaderBinary ( 1, &shader_id, GL_SHADER_BINARY_FORMAT_SPIR_V_ARB,
9:                      spirv, size );
10:    glSpecializeShaderARB ( shader_id, "main", 0, NULL, NULL );
11:    glGetShaderiv ( shader_id, GL_COMPILE_STATUS, &status );
12:    if ( !status ) {
13:      glDeleteShader ( shader_id );
14:      shader_id = 0;
15:    }
16:  }
17:  ExitIfGLError ( "CompileSPIRVString" );
18:  return shader_id;
19: } /*CreateSPIRVShader*/
20:
21: GLuint LoadSPIRVFile ( GLenum shader_type, const char *filename )
22: {
23:   GLuint shader_id = 0;
24:   FILE   *f;
25:   int    size;
26:   GLuint *spirv;
27:
28:   if ( !(f = fopen ( filename, "rb" )) )
29:     return 0;
30:   fseek ( f, 0, SEEK_END );
31:   size = ftell ( f );
32:   rewind ( f );
33:   if ( !(spirv = malloc ( size )) )
34:     goto way_out;
35:   if ( fread ( spirv, sizeof(char), size, f ) != size )
36:     goto way_out;
37:   shader_id = CreateSPIRVShader ( shader_type, size, spirv );
38: way_out:
39:   fclose ( f );
40:   if ( spirv ) free ( spirv );
41:   return shader_id;
42: } /*LoadSPIRVFile*/
```

punktu wejściowego szadera (w tym przykładzie jest to procedura `main`) oraz nadanie wartości tzw. stałym specjalizacji szadera (temu służą ostatnie trzy parametry, których wartości w przykładzie na listingu 4.9 oznaczają zaniechanie tej czynności). Stałe specjalizacji mogą np. wyznaczać wielkości pewnych tablic, lub zapotrzebowanie innych zasobów (np. liczby używanych tekstur), które można ustalić w chwili instalowania szadera w postaci SPIR-V w kontekście OpenGL-a na początku działania aplikacji (co umożliwia dostosowanie szadera do sprzętu, na którym ta aplikacja zostaje uruchomiona). Dalsze wiadomości na ten temat daleko wykraczają poza zakres tego kursu; zainteresowanych Czytelników odsyłam do strony [6] i innych źródeł dostępnych w Internecie.

5. Działania na wektorach i macierzach

Działania na wektorach w $\mathbb{R}^4$ i macierzach 4×4 są nadzwyczaj intensywnie wykorzystywane w grafice trójwymiarowej. Wektory reprezentują punkty przestrzeni i wektory swobodne, a także wektory normalne płaszczyzn (które nie są wektorami swobodnymi), zaś macierze reprezentują przekształcenia afiniczne i rzutowe. Nie próbuję streścić w tym rozdziale wykładu z algebry liniowej z geometrią, ale staram się przypomnieć własności reprezentacji punktów i wektorów swobodnych w postaci współrzędnych kartezjańskich, jednorodnych, barycentrycznych i jednorodnych barycentrycznych oraz jednorodną reprezentację przekształceń afinicznych, albowiem mają one zasadnicze znaczenie podczas pisania programów tworzących grafikę. Wprawdzie większość działań na wektorach i macierzach będzie wykonywać GPU, ale podstawowy zestaw procedur w C, wykonujących te działania na CPU jest potrzebny w aplikacji. Dzięki niemu CPU może konstruować macierze potrzebnych przekształceń, które potem GPU zastosuje do tysięcy punktów.

Punkty i wektory swobodne

Obiekty do narysowania konstruujemy w trójwymiarowej przestrzeni afinicznej, która składa się z punktów. Mamy w tej przestrzeni działanie odejmowania punktów; jego wynikiem jest wektor swobodny. Zbiór wszystkich wektorów swobodnych jest trójwymiarową przestrzenią liniową. Trzeba pamiętać, że przestrzeń afiniczna i przestrzeń liniowa są *różnymi* strukturami algebraicznymi, ponieważ w każdej z nich mamy określone inne działania — nawet jeśli reprezentacje punktów i wektorów w programie *mogą być* identyczne.

Elementy przestrzeni liniowej możemy mnożyć przez skalary (liczby rzeczywiste) i dodawać; w ogólności możemy tworzyć dowolne kombinacje liniowe, czyli wyrażenia postaci $\sum_{i=1}^{k} \boldsymbol{v}_i a_i$, gdzie $\boldsymbol{v}_i$ to wektory, a a_i to dowolne liczby. Wartość takiego wyrażenia jest wektorem. Mamy też wyróżniony wektor zerowy, $\boldsymbol{0}$: dla każdego wektora $\boldsymbol{v}$ jest $\boldsymbol{v} + \boldsymbol{0} = \boldsymbol{v}$. Natomiast definicja przestrzeni afinicznej nie wyróżnia żadnego jej punktu. Działanie odejmowania punktów umożliwia określenie dodawania wektora do punktu; jego wynikiem jest punkt. Na tej podstawie można określić dalsze dwa działania na punktach. Jeśli $\mathbf{p}_0, \ldots, \mathbf{p}_k$ to punkty przestrzeni afinicznej, to wyrażenie postaci $\sum_{i=0}^{k} \mathbf{p}_i a_i$ ma sens wtedy, gdy $a_0 + \cdots + a_k = 1$ albo $a_0 + \cdots + a_k = 0$. W pierwszym przypadku mamy punkt zwany kombinacją afiniczną punktów $\mathbf{p}_0, \ldots, \mathbf{p}_k$ (o współczynnikach $a_0, \ldots, a_k$), a w drugim przypadku otrzymujemy kombinację wektorową (albo różnicę uogólnioną) punktów; jest ona wektorem swobodnym.

Dobre określenie działań kombinacji afinicznej i kombinacji wektorowej polega na tym, że wyniki tych działań są jednoznacznie określone i nie zależą od reprezentacji punktów i wektorów, w tym od rodzaju używanych współrzędnych ani od konkretnego układu współrzędnych. Na przykład punkt $\mathbf{c} = \frac{1}{3}\mathbf{p}_0 + \frac{1}{3}\mathbf{p}_1 + \frac{1}{3}\mathbf{p}_2$ jest zawsze tym samym środkiem ciężkości trójkąta o wierzchołkach $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ i podobnie wektor $\boldsymbol{v} = -1\mathbf{p}_0 + \frac{1}{2}\mathbf{p}_1 + \frac{1}{2}\mathbf{p}_2$ określa przesunięcie, które przeprowadza wierzchołek $\mathbf{p}_0$ na środek przeciwległego boku tego trójkąta. Opisane niżej współrzędne umożliwiają szczególnie łatwe implementowanie tych działań.

Współrzędne kartezjańskie, jednorodne i barycentryczne

Układ współrzędnych kartezjańskich w trójwymiarowej przestrzeni afinicznej otrzymamy, wybierając układ odniesienia, na który składa się dowolny punkt $\mathbf{o}$ (początek układu) i trzy niezależne liniowo wektory swobodne $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$, zwane wersorami osi układu. Dla każdego punktu $\mathbf{p}$ przestrzeni istnieje jednoznacznie określona trójka liczb x, y, z, taka że $\mathbf{p} = \mathbf{o} + \boldsymbol{v}_1 x + \boldsymbol{v}_2 y + \boldsymbol{v}_3 z$. Układy odniesienia można wybierać na nieskończenie wiele sposobów, dlatego utożsamienie punktu $\mathbf{p}$ z wektorem $(x, y, z) \in \mathbb{R}^3$ jego współrzędnych kartezjańskich ma sens wtedy, gdy wiadomo, w jakim układzie te współrzędne są podane.

Opisane wyżej działania na punktach i wektorach możemy sprowadzić do rachunków na wektorach w przestrzeni $\mathbb{R}^3$ (czyli trójkach liczb), składających się ze współrzędnych kartezjańskich w ustalonym układzie współrzędnych. Dowolny wektor swobodny $\boldsymbol{v} = \boldsymbol{v}_1 x + \boldsymbol{v}_2 y + \boldsymbol{v}_3 z$ jest wtedy utożsamiony z trójką liczb (x, y, z). Jednak ta reprezentacja nie zawiera podstawowej informacji: co dana trójka liczb reprezentuje — punkt, czy wektor swobodny. W każdym układzie współrzędnych kartezjańskich początek tego układu jest reprezentowany przez wektor zerowy, a jego wersory osi odpowiednio przez wektory $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3 \in \mathbb{R}^3$; i-ta współrzędna wektora $\boldsymbol{e}_i$ jest jedynką, a pozostałe dwie są zerem.

Punkt w przestrzeni trójwymiarowej można reprezentować również przez czwórkę liczb, (X, Y, Z, W), zwanych współrzędnymi jednorodnymi[1]; możemy z niej otrzymać współrzędne kartezjańskie jako wyniki dzielenia:

$$x = \frac{X}{W}, \quad y = \frac{Y}{W}, \quad z = \frac{Z}{W}. \tag{5.1}$$

Ta reprezentacja jest oczywiście nadmiarowa: każdy punkt jest reprezentowany

[1] Należałoby mówić o współrzędnych kartezjańskich jednorodnych, ponieważ dalej „ujednorodnimy” też współrzędne barycentryczne, otrzymując współrzędne barycentryczne jednorodne. Ale dla skrótu słowo „kartezjańskie” przed słowem „jednorodne” będziemy pomijać.

przez jeden wektor współrzędnych kartezjańskich i przez nieskończenie wiele wektorów współrzędnych jednorodnych; jeśli $W \neq 0$ i $a \neq 0$, to wektory (X, Y, Z, W) i (aX, aY, aZ, aW) reprezentują ten sam punkt. W szczególności, mając współrzędne kartezjańskie x, y, z punktu, otrzymamy jego współrzędne jednorodne, „doczepiając" jedynkę: $(x, y, z) \to (x, y, z, 1)$. Ostatnia współrzędna jednorodna (W) ma nazwę współrzędnej wagowej albo wagi.

Jeśli wektory współrzędnych jednorodnych reprezentujące punkty ograniczymy do takich, których współrzędna wagowa jest równa 1 (co oznacza, że pierwsze trzy współrzędne jednorodne są również współrzędnymi kartezjańskimi punktu), to odejmowanie punktów jest tożsame z odejmowaniem ich wektorów współrzędnych jednorodnych, przy czym wynik — reprezentacja wektora swobodnego — ma współrzędną wagową równą 0. Możemy też obliczać kombinacje liniowe wektorów swobodnych, dodawać wektory swobodne do punktów oraz obliczać kombinacje afiniczne i wektorowe punktów, wykonując odpowiednie działania liczbowe na wektorach współrzędnych jednorodnych. Wyniki tych działań mają współrzędną wagową 0 albo 1, dzięki czemu zawsze wiadomo, jakie obiekty (wektory swobodne czy punkty) reprezentują. Możliwość jednolitego reprezentowania punktów i wektorów swobodnych (pod warunkiem przestrzegania opisanego tu ograniczenia) jest jedną z wielu korzyści używania współrzędnych jednorodnych.

Współrzędne barycentryczne otrzymamy, wybierając układ odniesienia składający się z czterech punktów, $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, nieleżących w jednej płaszczyźnie. Dowolny punkt $\mathbf{p}$ możemy wtedy zapisać w postaci

$$\mathbf{p} = \mathbf{p}_0 t_0 + \mathbf{p}_1 t_1 + \mathbf{p}_2 t_2 + \mathbf{p}_3 t_3, \tag{5.2}$$

przy użyciu czwórki liczb (t_0, t_1, t_2, t_3), których suma jest równa 1. Zauważmy, że mamy również pełną swobodę wybierania układów współrzędnych barycentrycznych w przestrzeni. Jeśli w miejsce punktów $\mathbf{p}_0, \ldots, \mathbf{p}_3$ podstawimy wektory ich współrzędnych kartezjańskich (w dowolnym ustalonym układzie), to podany wyżej wzór umożliwia obliczenie współrzędnych kartezjańskich punktu $\mathbf{p}$ (w tymże układzie) na podstawie jego współrzędnych barycentrycznych.

Współrzędne barycentryczne można również „ujednorodnić"; w tym celu warunek $t_0 + t_1 + t_2 + t_3 = 1$ zastępujemy przez $T_0 + T_1 + T_2 + T_3 \neq 0$. „Zwykłe" współrzędne barycentryczne $t_0, \ldots, t_3$ otrzymamy z barycentrycznych współrzędnych jednorodnych, dzieląc każdą z nich przez sumę $T_0 + T_1 + T_2 + T_3$, która odpowiada wadze wektora współrzędnych. Mając jednorodne współrzędne barycentryczne T_0, T_1, T_2, T_3 punktu $\mathbf{p}$, możemy obliczyć jego wektor współrzędnych

jednorodnych $\mathbf{P}$ na podstawie wzoru

$$\mathbf{P} = \mathbf{P}_0 T_0 + \mathbf{P}_1 T_1 + \mathbf{P}_2 T_2 + \mathbf{P}_3 T_3, \tag{5.3}$$

w którym występują wektory $\mathbf{P}_0, \ldots, \mathbf{P}_3$ współrzędnych jednorodnych punktów $\mathbf{p}_0, \ldots, \mathbf{p}_3$ pod warunkiem, że współrzędne wagowe we wszystkich tych wektorach mają tę samą wartość[2].

Ograniczenie wektorów współrzędnych barycentrycznych punktów do takich, których suma współrzędnych jest równa 1 umożliwia dokładnie taką samą odpowiedniość między działaniami na punktach i wektorach swobodnych i rachunkami na wektorach współrzędnych barycentrycznych. Odejmując dwa takie wektory otrzymamy wektor, którego suma współrzędnych jest zerem — a więc wektory w $\mathbb{R}^4$ o zerowej sumie współrzędnych (barycentrycznych) reprezentują wektory swobodne w przestrzeni trójwymiarowej.

Przekształcenia afiniczne

Przekształcenie afiniczne przestrzeni jest dane wzorem

$$f(\mathbf{p}) = L\mathbf{p} + \mathbf{t}, \tag{5.4}$$

w którym występuje macierz L o wymiarach 3×3 opisująca część liniową przekształcenia f i wektor przesunięcia $\mathbf{t} \in \mathbb{R}^3$; w podanym wzorze symbole $\mathbf{p}$ i $f(\mathbf{p})$ oznaczają wektory współrzędnych kartezjańskich punktu $\mathbf{p}$ i jego obrazu. Aby działania we wzorze były dobrze określone, wektory te są zapisywane jako macierze kolumnowe.

Długo by pisać o podstawowych własnościach przekształceń afinicznych, ale wychodzę z założenia, że Czytelnik je zna. Zatem, opiszę tylko ich jednorodną reprezentację. Jest nią macierz A o wymiarach 4×4, która zawiera bloki L i $\mathbf{t}$:

$$A = \left[\begin{array}{ccc|c} & & & \\ & L & & \mathbf{t} \\ & & & \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{5.5}$$

Aby poddać punkt $\mathbf{p}$ przekształceniu, wystarczy obliczyć iloczyn $A\mathbf{P}$ macierzy A i wektora współrzędnych jednorodnych punktu $\mathbf{p}$. Jeśli współrzędna wagowa wektora $\mathbf{P}$ jest równa 1, to iloczyn — wektor współrzędnych jednorodnych

[2] Zauważmy, że jeśli współrzędna wagowa wektorów $\mathbf{P}_0, \ldots, \mathbf{P}_3$ jest równa 1, to otrzymamy wektor współrzędnych jednorodnych punktu $\mathbf{p}$ o współrzędnej wagowej $T_0 + T_1 + T_2 + T_3$.

punktu $f(\mathbf{p})$ — ma również współrzędną wagową równą 1, a więc pierwsze 3 współrzędne jednorodne są współrzędnymi kartezjańskimi tego punktu.

Przekształcenie wektora swobodnego odpowiadające przekształceniu afinicznemu f polega na zastosowaniu do niego przekształcenia liniowego l reprezentowanego przez macierz L. Iloczyn macierzy A i wektora $\mathbf{V} \in \mathbb{R}^4$ otrzymanego przez dopisanie współrzędnej wagowej 0 do wektora $\mathbf{v}$ składa się z iloczynu $L\mathbf{v}$ i wagi 0. Zatem reprezentacja jednorodna w postaci macierzy A umożliwia jednolite traktowanie przekształcanych punktów i wektorów swobodnych, o ile tylko przestrzegamy ograniczenia, że wagi punktów są równe 1, a ostatni wiersz macierzy A składa się z trzech zer na początku i jedynki.

Jednorodność powyższej reprezentacji oznacza, że macierz aA, dla dowolnej stałej $a \neq 0$, reprezentuje to samo przekształcenie co macierz A.[3] Jedną z zalet tej reprezentacji jest prostota wzoru opisującego złożenie przekształceń; jeśli przekształcenia afiniczne f_1 i f_2 są reprezentowane przez macierze A_1 i A_2, to złożenie $f_2 \circ f_1$ reprezentuje iloczyn A_2A_1. Iloczyn ten reprezentuje również złożenie związanych z f_2 i f_1 przekształceń liniowych l_2 i l_1, którym podlegają wektory swobodne.

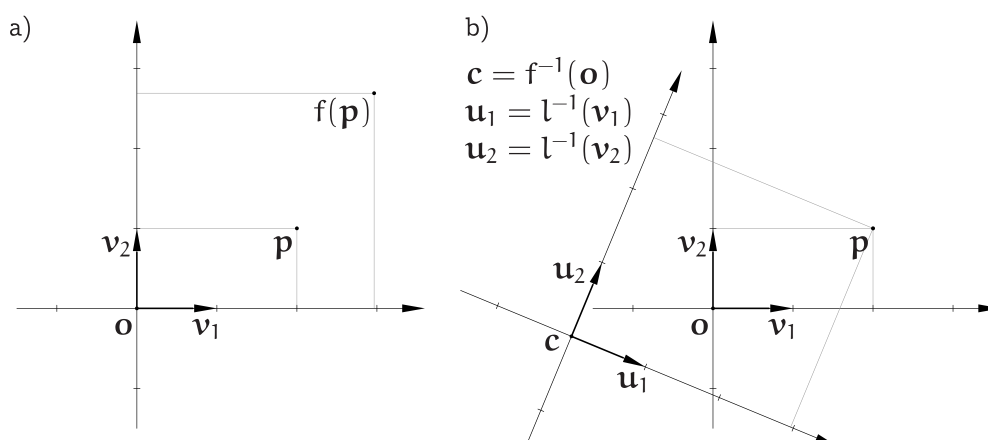
Macierz przekształcenia afinicznego możemy przedstawić w taki sposób:

$$A = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \mathbf{w}_3 & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Wtedy widzimy, że jej kolumny są jednorodnymi reprezentacjami trzech wektorów swobodnych i punktu. Punkt ten jest obrazem początku układu współrzędnych w przekształceniu f, zaś wektory $\mathbf{w}_1$, $\mathbf{w}_2$, $\mathbf{w}_3$ reprezentują obrazy wersorów osi w przekształceniu l będącym częścią liniową przekształcenia f.

Przekształcenia afiniczne mają dualną interpretację, z której również będziemy intensywnie korzystać w aplikacjach OpenGL-a. W interpretacji dualnej wektor $f(\mathbf{p}) \in \mathbb{R}^3$ obliczony ze wzoru (5.4) reprezentuje *ten sam* punkt co wektor $\mathbf{p}$, ale w *nowym* układzie współrzędnych. Rozważmy dwa układy współrzędnych: pierwszy, o wersorach osi $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ i początku $\mathbf{o}$ i drugi układ, o wersorach osi $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3$ i początku $\mathbf{c}$, taki że wersory osi i początek *pierwszego* układu mają *w drugim* układzie wektory współrzędnych kartezjańskich $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, \mathbf{t} \in \mathbb{R}^3$.

[3]Oczywiście, dopuszczając macierze z ostatnim wierszem $[0, 0, 0, a]$ dla $a \neq 1$, rezygnujemy z otrzymywania wektorów współrzędnych jednorodnych punktów o wadze 1, wskutek czego tracimy opisane wcześniej utożsamienie odejmowania punktów z odejmowaniem ich wektorów współrzędnych jednorodnych.



Rysunek 5.1: Przekształcenie afiniczne i jego dualna interpretacja

Macierz A, której kolumny powstały przez dołączenie trzech zer i jedynki do tych wektorów, opisuje przejście od pierwszego układu współrzędnych do drugiego; macierz przejścia od układu drugiego do pierwszego jest jej odwrotnością[4]. Zatem przekształcenie afiniczne f reprezentowane przez macierz A jest w interpretacji dualnej przejściem do układu współrzędnych, którego układ odniesienia otrzymamy, stosując przekształcenie f^{-1} i jego część liniową l^{-1} do elementów wyjściowego układu odniesienia: $\mathbf{c} = f^{-1}(\mathbf{o})$, $\mathbf{u}_1 = l^{-1}(\mathbf{v}_1)$, $\mathbf{u}_2 = l^{-1}(\mathbf{v}_2)$ i $\mathbf{u}_3 = l^{-1}(\mathbf{v}_3)$ (zobacz rysunek 5.1b; nie ma na nim wersorów $\mathbf{v}_3$ i $\mathbf{u}_3$, bo nie leżą w płaszczyźnie kartki). Temat ten wielokrotnie będzie wracał w opisie aplikacji.

Choć to rzadziej będzie potrzebne, zobaczmy, jak można znaleźć macierz przekształcenia afinicznego f, przez którą chcielibyśmy mnożyć wektor współrzędnych barycentrycznych punktu $\mathbf{p}$, aby otrzymać wektor współrzędnych barycentrycznych punktu $f(\mathbf{p})$. Ze wzoru (5.3) wynika, że macierz $B = [\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3]$, której kolumny są wektorami współrzędnych jednorodnych punktów układu odniesienia ze współrzędną wagową równą 1, opisuje przejście od współrzędnych barycentrycznych dowolnego punktu do kartezjańskich (iloczyn $\mathbf{P} = B\mathbf{T}$ macierzy B i wektora $\mathbf{T}$ współrzędnych barycentrycznych punktu $\mathbf{p}$ jest wektorem współrzędnych jednorodnych punktu $\mathbf{p}$ z wagą 1). Zatem poszukiwana macierz jest iloczynem $B^{-1}AB$ — reprezentuje on złożenie trzech przekształceń: przejścia do współrzędnych kartezjańskich, przekształcenia f zapisanego we współrzędnych kartezjańskich i powrotu do współrzędnych barycentrycznych[5].

[4]Przekształcenie f musi być różnowartościowe, co ma miejsce, gdy macierz $L = [\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3]$ jest nieosobliwa.

[5]Macierz B jest nieosobliwa gdy punkty $\mathbf{p}_0, \ldots, \mathbf{p}_3$ nie leżą w jednej płaszczyźnie.

Prostopadłość

Pojęcie prostopadłości (np. prostych) w przestrzeni afinicznej jest związane z iloczynem skalarnym określonym w przestrzeni wektorów swobodnych; jest to funkcja, która parze wektorów przyporządkowuje liczbę i która *w pewnym układzie współrzędnych kartezjańskich* może być zapisana wzorem

$$\langle \boldsymbol{v}, \boldsymbol{w} \rangle = x_v x_w + y_v y_w + z_v z_w. \tag{5.6}$$

W notacji macierzowej $\langle \boldsymbol{v}, \boldsymbol{w} \rangle = \boldsymbol{v}^T \boldsymbol{w} = \boldsymbol{w}^T \boldsymbol{v}$. Przyjmiemy, że ten wzór obowiązuje w układzie współrzędnych świata, którego definicja jest podana w następnym rozdziale. Dwa wektory są wzajemnie prostopadłe, jeśli ich iloczyn skalarny jest zerem. Przestrzeń wyposażona w iloczyn skalarny to przestrzeń euklidesowa.

Iloczyn skalarny umożliwia określenie długości wektora,

$$\|\boldsymbol{v}\| = \sqrt{\langle \boldsymbol{v}, \boldsymbol{v} \rangle},$$

a dalej odległości punktów,

$$\rho(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|,$$

a także pojęcia kąta (nieskierowanego) między wektorami niezerowymi $\boldsymbol{v}$ i $\boldsymbol{w}$; kąt ten jest dany wzorem

$$\cos \phi = \frac{\langle \boldsymbol{v}, \boldsymbol{w} \rangle}{\|\boldsymbol{v}\| \|\boldsymbol{w}\|}.$$

Umiejętność mierzenia odległości umożliwia zdefiniowanie izometrii — jest to takie przekształcenie f, że dla dowolnych punktów $\mathbf{p}$, $\mathbf{q}$ ma miejsce równość $\rho\big(f(\mathbf{p}), f(\mathbf{q})\big) = \rho(\mathbf{p}, \mathbf{q})$. Izometria w afinicznej przestrzeni euklidesowej jest przekształceniem afinicznym, którego część liniowa jest opisana przez macierz ortogonalną. Kolumny macierzy ortogonalnej są wektorami jednostkowymi (tj. o długości 1), przy czym każda z nich jest prostopadła do pozostałych. Stąd wynika, że macierz L jest ortogonalna wtedy i tylko wtedy, gdy jej transpozycja jest jej odwrotnością: $L^{-1} = L^T$. Jeśli przejście od układu współrzędnych świata do dowolnego innego jest izometrią, to w tym innym układzie iloczyn skalarny też może być obliczony ze wzoru (5.6); takie układy nazwiemy izometrycznymi.

W trójwymiarowej przestrzeni euklidesowej możemy określić iloczyn wektorowy dwóch wektorów, dobrze znanym wzorem

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} \wedge \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} y_v z_w - y_w z_v \\ z_v x_w - z_w x_v \\ x_v y_w - x_w y_v \end{bmatrix}, \tag{5.7}$$

w którym występują współrzędne kartezjańskie wektorów w układzie izometrycznym. Iloczyn wektorowy jest wektorem prostopadłym do swoich czynników. Ponadto działanie to *nie jest przemienne* (bo zawsze $\boldsymbol{v} \wedge \boldsymbol{w} = -\boldsymbol{w} \wedge \boldsymbol{v}$, a nie zawsze $\boldsymbol{v} \wedge \boldsymbol{w} = 0$) i *nie jest łączne* (bo dla niezależnych liniowo wektorów $\boldsymbol{v}$ i $\boldsymbol{w}$ jest $(\boldsymbol{v} \wedge \boldsymbol{w}) \wedge \boldsymbol{w} \neq \boldsymbol{0} = \boldsymbol{v} \wedge (\boldsymbol{w} \wedge \boldsymbol{w})$).

Procedury

Dla macierzy o wymiarach 4×4 i mniejszych język GLSL ma zdefiniowane typy podstawowe (o nazwach takich jak `mat4` lub `mat4x4`). Współczynniki takich macierzy są upakowane jeden za drugim, przy czym ich porządek w pamięci jest *kolumnowy*: współczynniki pierwszej kolumny poprzedzają współczynniki drugiej kolumny itd. To jest inny porządek niż najczęściej stosowany w C wierszowy porządek współczynników macierzy. Dlatego macierze 4×4 przetwarzane przez procedury opisane niżej są reprezentowane jako tablice jednowymiarowe; do ułatwienia dostępu do współczynników służy makrodefinicja `IND` (zobacz listing 5.2), której pierwszy parametr jest numerem wiersza, a drugi — kolumny (numerowanych od 0 do 3).

Listingi 5.1, 5.2 i 5.3 przedstawiają dalsze części pliku `utilities.c`. Na pierwszym z nich są procedury konstrukcji macierzy elementarnych przekształceń afinicznych. Procedura `M4x4Identf` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy jednostkowej, reprezentującej przekształcenie tożsamościowe.

Procedura `M4x4Translatef` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy T przesunięcia o wektor o podanych współrzędnych; macierz ta jest określona wzorem

$$T = \left[\begin{array}{ccc|c} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right].$$

Procedura `M4x4Scalef` wpisuje do tablicy przekazanej jako parametr współczynniki macierzy skalowania osi o podane czynniki. Górny lewy blok 3×3 tej macierzy (odpowiadający macierzy L we wzorach (5.4) i (5.5)) jest równy

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix},$$

a ostatnia kolumna i wiersz są takie, jak w macierzy jednostkowej.

Listing 5.1: Procedury konstrukcji macierzy przekształceń

C

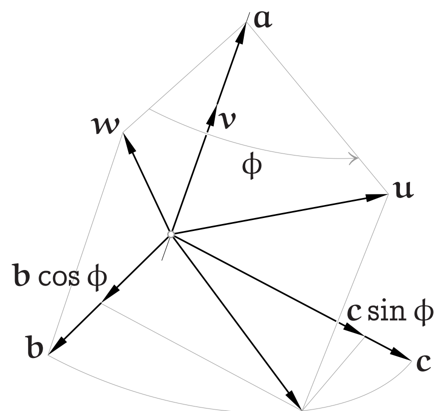
```
1: void M4x4Identf ( GLfloat a[16] )
2: {
3:   memset ( a, 0, 16*sizeof(GLfloat) );
4:   a[0] = a[5] = a[10] = a[15] = 1.0;
5: } /*M4x4Identf*/
6:
7: void M4x4Translatef ( GLfloat a[16], float x, float y, float z )
8: {
9:   M4x4Identf ( a );
10:  a[12] = x;  a[13] = y;  a[14] = z;
11: } /*M4x4Translatef*/
12:
13: void M4x4Scalef ( GLfloat a[16], float sx, float sy, float sz )
14: {
15:   M4x4Identf ( a );
16:   a[0] = sx;  a[5] = sy;  a[10] = sz;
17: } /*M4x4Scalef*/
18:
19: void M4x4RotateXf ( GLfloat a[16], float phi )
20: {
21:   M4x4Identf ( a );
22:   a[5] = a[10] = cos ( phi);  a[9] = -(a[6] = sin ( phi ));
23: } /*M4x4RotateXf*/
24:
25: void M4x4RotateYf ( GLfloat a[16], float phi )
26: {
27:   M4x4Identf ( a );
28:   a[0] = a[10] = cos ( phi);  a[2] = -(a[8] = sin ( phi ));
29: } /*M4x4RotateYf*/
30:
31: void M4x4RotateZf ( GLfloat a[16], float phi )
32: {
33:   M4x4Identf ( a );
34:   a[0] = a[5] = cos ( phi );  a[4] = -(a[1] = sin ( phi ));
35: } /*M4x4RotateZf*/
36:
37: void M4x4RotateVf ( GLfloat a[16], float x, float y, float z, float phi )
38: {
39:   float l, s, c, c1;
40:
41:   M4x4Identf ( a );
42:   l = 1.0/sqrt ( x*x + y*y + z*z );  x *= l;  y *= l;  z *= l;
```

```
43:   s = sin ( phi );  c = cos ( phi );  c1 = 1.0-c;
44:   a[0] = x*x*c1 + c;  a[1] = a[4] = x*y*c1;  a[5] = y*y*c1 + c;
45:   a[2] = a[8] = x*z*c1;  a[6] = a[9] = y*z*c1;  a[10] = z*z*c1 + c;
46:   a[6] += s*x;  a[2] -= s*y;  a[1] += s*z;
47:   a[9] -= s*x;  a[8] += s*y;  a[4] -= s*z;
48: } /*M4x4RotateVf*/
```

Procedury `M4x4RotateXf`, `M4x4RotateYf`, `M4x4RotateZf` wpisują do tablicy współczynniki macierzy obrotu o kąt ϕ (podany w radianach), odpowiednio wokół osi x, y i z. Macierze tych obrotów mają ostatni wiersz i kolumnę takie jak macierz jednostkowa i bloki 3×3 w górnym lewym rogu określone wzorami

$$R_{x,\phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, \qquad R_{y,\phi} = \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix}, \qquad R_{z,\phi} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

gdzie $s = \sin\phi$, $c = \cos\phi$.



Rysunek 5.2: Konstrukcja obrotu wokół osi określonej przez wektor v

Obraz $\boldsymbol{u}$ wektora $\boldsymbol{w}$ w obrocie o kąt ϕ wokół osi o kierunku *jednostkowego* wektora $\boldsymbol{v}$ można znaleźć w sposób pokazany na rysunku 5.2. Wektory $\boldsymbol{a} = \boldsymbol{v}\langle\boldsymbol{v},\boldsymbol{w}\rangle$ i $\boldsymbol{b} = \boldsymbol{w} - \boldsymbol{a}$ są obrazami wektora $\boldsymbol{w}$ w rzutach prostopadłych na oś obrotu i na płaszczyznę do niej prostopadłą. Wektor $\boldsymbol{c} = \boldsymbol{v} \wedge \boldsymbol{b} = \boldsymbol{v} \wedge \boldsymbol{w}$ jest prostopadły do wektorów $\boldsymbol{w}$, $\boldsymbol{a}$, $\boldsymbol{b}$ i ma długość taką jak wektor $\boldsymbol{b}$. Wektor $\boldsymbol{u}$ jest sumą $\boldsymbol{a} + \boldsymbol{b}\cos\phi + \boldsymbol{c}\sin\phi$, a zatem

$$\boldsymbol{u} = \boldsymbol{v}\langle\boldsymbol{v},\boldsymbol{w}\rangle + \cos\phi(\boldsymbol{w} - \boldsymbol{v}\langle\boldsymbol{v},\boldsymbol{w}\rangle) + \sin\phi\boldsymbol{v} \wedge \boldsymbol{w}. \tag{5.8}$$

Procedura `M4x4RotateVf` wpisuje do tablicy współczynniki macierzy obrotu o kąt ϕ wokół przechodzącej przez początek układu współrzędnych osi o kierunku wektora mającego współrzędne `x`, `y`, `z`. Wektor ten musi być niezerowy; po

podzieleniu go przez jego długość powstaje wektor jednostkowy $\boldsymbol{v}$. Na podstawie wzoru (5.8), macierz części liniowej obrotu jest równa:

$$R_{\boldsymbol{v},\phi} = \boldsymbol{v}\boldsymbol{v}^T + c(I - \boldsymbol{v}\boldsymbol{v}^T) + s\boldsymbol{v} \wedge I. \tag{5.9}$$

We wzorze tym występuje wektor jednostkowy $\boldsymbol{v}$ otrzymany przez podzielenie wektora $[x, y, z]^T$ przez jego długość, macierz jednostkowa I o wymiarach 3×3 oraz liczby $s = \sin\phi$, $c = \cos\phi$. Wyrażenie $\boldsymbol{v} \wedge I$ oznacza macierz, której kolumny są iloczynami wektorowymi wektora $\boldsymbol{v}$ i odpowiednich kolumn macierzy jednostkowej; jeśli $\boldsymbol{v} = [x_v, y_v, z_v]^T$, to

$$\boldsymbol{v} \wedge I = \begin{bmatrix} 0 & -z_v & y_v \\ z_v & 0 & -x_v \\ -y_v & x_v & 0 \end{bmatrix}.$$

Iloczynem tej macierzy i dowolnego wektora $\boldsymbol{w}$ jest iloczyn wektorowy $\boldsymbol{v} \wedge \boldsymbol{w}$.

Procedury na listingu 5.2 wykonują podstawowe działania na macierzach. Procedura `M4x4Multf` oblicza iloczyn dwóch macierzy, A i B, i umieszcza go w tablicy `ab`. Macierz ta reprezentuje złożenie przekształceń realizowanych przez macierze B i A.

Procedura `M4x4LUDecompf` wyznacza rozkład danej macierzy A na czynniki trójkątne, dolny L i górny U: jest $LU = PA$ dla pewnej macierzy permutacji P. Współczynniki macierzy L i U są wpisywane do tablicy `lu`, zaś w pomocniczej tablicy `p` jest zapisywana reprezentacja permutacji P. Czynniki tego rozkładu mogą być użyte do rozwiązywania układów równań liniowych $A\boldsymbol{x} = \boldsymbol{b}$ i do znalezienia macierzy odwrotnej do A.[6] Procedura zwraca wartość `true` (czyli liczbę 1), jeśli czynniki rozkładu udało się znaleźć, co jest możliwe, jeśli macierz A jest nieosobliwa, albo `false` (liczbę 0) w przeciwnym razie.

Procedura `M4x4LUsolvef` oblicza rozwiązanie układu równań liniowych $A\boldsymbol{x} = \boldsymbol{v}$, czyli wektor $\boldsymbol{x} = A^{-1}\boldsymbol{v} = U^{-1}L^{-1}P\boldsymbol{v}$, korzystając z czynników L, U, P rozkładu macierzy nieosobliwej A znalezionych przez procedurę `M4x4LUDecompf`.

Procedura `M4x4Invertf` znajduje odwrotność macierzy nieosobliwej A i zwraca wartość `true`; jeśli macierz przekazana jako parametr jest osobliwa, to procedura zwraca wartość `false`.

Uwaga: Jeśli macierz A jest ortogonalna (na przykład jest to macierz obrotu lub odbicia symetrycznego), to dla zasady nie należy rozwiązywać tego układu ani

[6] Do numerycznego rozwiązywania takich zadań *nie należy* stosować wyznaczników.

Listing 5.2: Procedury działań na macierzach

C

```c
#define IND(i,j) ((i)+4*(j))

void M4x4Multf ( GLfloat ab[16], const GLfloat a[16], const GLfloat b[16] )
{
  int i, j, k;

  for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 4; j++ ) {
      ab[IND(i,j)] = a[IND(i,0)]*b[IND(0,j)];
      for ( k = 1; k < 4; k++ )
        ab[IND(i,j)] += a[IND(i,k)]*b[IND(k,j)];
    }
} /*M4x4Multf*/

char M4x4LUDecompf ( GLfloat lu[16], int p[3], const GLfloat a[16] )
{
  int     i, j, k;
  GLfloat d;

  memcpy ( lu, a, 16*sizeof(GLfloat) );
  memset ( p, 0, 3*sizeof(int) );
  for ( j = 0; j < 3; j++ ) {
    d = fabs ( lu[IND(j,j)] );  p[j] = j;
    for ( i = j+1; i < 4; i++ )
      if ( fabs ( lu[IND(i,j)] ) > d )
        { d = fabs ( lu[IND(i,j)] );  p[j] = i; }
    if ( d == 0.0 )
      return false;
    if ( p[j] != j ) {
      i = p[j];
      for ( k = 0; k < 4; k++ )
        { d = lu[IND(i,k)]; lu[IND(i,k)] = lu[IND(j,k)]; lu[IND(j,k)] = d; }
    }
    for ( i = j+1; i < 4; i++ ) {
      d = lu[IND(i,j)] /= lu[IND(j,j)];
      for ( k = j+1; k < 4; k++ )
        lu[IND(i,k)] -= d*lu[IND(j,k)];
    }
  }
  return lu[15] != 0.0;
} /*M4x4LUDecompf*/

```

```
43: void M4x4LUsolvef ( GLfloat aiv[4], const GLfloat lu[16],
44:                     const int p[3], const GLfloat v[4] )
45: {
46:   int     i, j;
47:   GLfloat d;
48:
49:   memcpy ( aiv, v, 4*sizeof(GLfloat) );
50:   for ( i = 0; i < 3; i++ )
51:     if ( p[i] != i ) { d = aiv[i];  aiv[i] = aiv[p[i]];  aiv[p[i]] = d; }
52:   for ( i = 1; i < 4; i++ )
53:     for ( j = 0; j < i; j++ ) aiv[i] -= lu[IND(i,j)]*aiv[j];
54:   for ( i = 3; i >= 0; i-- ) {
55:     for ( j = i+1; j < 4; j++ ) aiv[i] -= lu[IND(i,j)]*aiv[j];
56:     aiv[i] /= lu[IND(i,i)];
57:   }
58: } /*M4x4LUsolvef*/
59:
60: char M4x4Invertf ( GLfloat ai[16], const GLfloat a[16] )
61: {
62:   int i, p[3];
63:   GLfloat lu[16], e[4];
64:
65:   if ( !M4x4LUDecompf ( lu, p, a ) )
66:     return false;
67:   for ( i = 0; i < 4; i++ ) {
68:     memset ( e, 0, 4*sizeof(GLfloat) );  e[i] = 1.0;
69:     M4x4LUsolvef ( &ai[4*i], lu, p, e );
70:   }
71:   return true;
72: } /*M4x4Invertf*/
73:
74: void M4x4UTLTSolvef ( GLfloat ativ[4], const GLfloat lu[16],
75:                       const int p[3], const GLfloat v[4] )
76: {
77:   int     i, j;
78:   GLfloat d;
79:
80:   memcpy ( ativ, v, 4*sizeof(GLfloat) );
81:   for ( i = 0; i <= 3; i++ ) {
82:     for ( j = 0; j < i; j++ ) ativ[i] -= lu[IND(j,i)]*ativ[j];
83:     ativ[i] /= lu[IND(i,i)];
84:   }
85:   for ( i = 2; i >= 0; i-- )
86:     for ( j = i+1; j < 4; j++ ) ativ[i] -= lu[IND(j,i)]*ativ[j];
87:   for ( i = 2; i >= 0; i-- )
```

```
88:     if ( p[i] != i ) { d = ativ[i]; ativ[i] = ativ[p[i]]; ativ[p[i]] = d; }
89:  }
90: } /*M4x4UTLTSolvef*/
91:
92: char M4x4TInvertf ( GLfloat ati[16], const GLfloat a[16] )
93: {
94:   int     i, p[3];
95:   GLfloat lu[16], e[4];
96:
97:   if ( !M4x4LUDecompf ( lu, p, a ) )
98:     return false;
99:   for ( i = 0; i < 4; i++ ) {
100:    memset ( e, 0, 4*sizeof(GLfloat) );  e[i] = 1.0;
101:    M4x4UTLTSolvef ( &ati[4*i], lu, p, e );
102:  }
103:  return true;
104: } /*M4x4TInvertf*/
105:
106: void M4x4Transposef ( GLfloat at[16], const GLfloat a[16] )
107: {
108:  int i, j;
109:
110:  for ( i = 0; i < 4; i++ )
111:    for ( j = 0; j < 4; j++ ) at[IND(i,j)] = a[IND(j,i)];
112: } /*M4x4Transposef*/
```

znajdować odwrotności przy użyciu powyższych procedur. Znacznie lepszy (szybszy i dokładniejszy) sposób opiera się na fakcie, że odwrotność macierzy ortogonalnej jest jej transpozycją — można zatem użyć opisanej dalej procedury `M4x4MultMTVf` albo `M4x4Transposef`.

Procedury `M4x4UTLTSolvef` i `M4x4TInvertf` służą do rozwiązywania układu równań $A^T\boldsymbol{x} = \boldsymbol{v}$ i do znajdowania odwrotności transpozycji macierzy nieosobliwej A; parametry `lu` i `p` pierwszej z nich opisują czynniki rozkładu LU, znalezione przez procedurę `M4x4LUDecompf`. Odwrotność transpozycji macierzy A oznaczamy symbolem A^{-T}; macierz taka często jest potrzebna w obliczeniach oświetlenia.

Procedura `M4x4Transposef` znajduje macierz transponowaną do danej macierzy A.

Procedura `M4x4MultMVf` na listingu 5.3 oblicza iloczyn macierzy A i wektora $\boldsymbol{v} \in \mathbb{R}^4$. Procedura `M4x4MultMTVf` oblicza iloczyn $A^T\boldsymbol{v}$. Aby można było prościej

Listing 5.3: Procedury działań na macierzach i wektorach

C

```c
1: void M4x4MultMVf ( GLfloat av[4], const GLfloat a[16], const GLfloat v[4] )
2: {
3:   int i, j;
4:
5:   for ( i = 0; i < 4; i++ ) {
6:     av[i] = a[IND(i,0)]*v[0];
7:     for ( j = 1; j < 4; j++ ) av[i] += a[IND(i,j)]*v[j];
8:   }
9: } /*M4x4MultMVf*/
10:
11: void M4x4MultMTVf ( GLfloat av[4], const GLfloat a[16], const GLfloat v[4] )
12: {
13:   int i, j;
14:
15:   for ( i = 0; i < 4; i++ ) {
16:     av[i] = a[IND(0,i)]*v[0];
17:     for ( j = 1; j < 4; j++ ) av[i] += a[IND(j,i)]*v[j];
18:   }
19: } /*M4x4MultMTVf*/
20:
21: void M4x4MultMV3f ( GLfloat av[3], const GLfloat a[16], const GLfloat v[3] )
22: {
23:   int i, j;
24:
25:   for ( i = 0; i < 3; i++ ) {
26:     av[i] = a[IND(i,0)]*v[0];
27:     for ( j = 1; j < 3; j++ ) av[i] += a[IND(i,j)]*v[j];
28:   }
29: } /*M4x4MultMV3f*/
30:
31: void M4x4MultMP3f ( GLfloat ap[3], const GLfloat a[16], const GLfloat p[3] )
32: {
33:   int i, j;
34:
35:   for ( i = 0; i < 3; i++ ) {
36:     ap[i] = a[IND(i,3)];
37:     for ( j = 0; j < 3; j++ ) ap[i] += a[IND(i,j)]*p[j];
38:   }
39: } /*M4x4MultMP3f*/
40:
41: GLfloat V3DotProductf ( const GLfloat v1[3], const GLfloat v2[3] )
42: {
```

```
43:   return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
44: } /*V3DotProductf*/
45:
46: void V3CrossProductf ( GLfloat v1xv2[3],
47:                        const GLfloat v1[3], const GLfloat v2[3] )
48: {
49:   v1xv2[0] = v1[1]*v2[2] - v1[2]*v2[1];
50:   v1xv2[1] = v1[2]*v2[0] - v1[0]*v2[2];
51:   v1xv2[2] = v1[0]*v2[1] - v1[1]*v2[0];
52: } /*V3CrossProductf*/
```

programować pewne konstrukcje, wprowadziłem procedury `M4x4MultMV3f` i `M4x4MultMPf`. Ostatni parametr obu tych procedur jest wektorem $\boldsymbol{v} \in \mathbb{R}^3$. Wynik obliczeń pierwszej procedury jest iloczynem tego wektora i górnego lewego bloku 3×3 macierzy A podanej jako drugi parametr; jeśli macierz ta ma ostatni wiersz $[0, 0, 0, 1]$, to wynik ten jest obrazem wektora $\boldsymbol{v}$ w przekształceniu opisującym część liniową przekształcenia afinicznego reprezentowanego przez macierz A.

Procedura `M4x4MultMP3f` oblicza pierwsze trzy współrzędne iloczynu $A\mathbf{P}$ macierzy A o wymiarach 4×4 i wektora $\mathbf{P} \in \mathbb{R}^4$, którego pierwsze trzy współrzędne są dane w tablicy podanej jako trzeci parametr, a ostatnia współrzędna, przez domniemanie, jest równa 1. Otrzymujemy w ten sposób współrzędne kartezjańskie obrazu punktu w przekształceniu afinicznym reprezentowanym przez macierz A (pod warunkiem, że jej ostatni wiersz ... itd.).

Procedury `V3DotProductf` i `V3CrossProductf` obliczają odpowiednio iloczyn skalarny $\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle$ i iloczyn wektorowy $\boldsymbol{v}_1 \wedge \boldsymbol{v}_2$ danych wektorów $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{R}^3$.

Arytmetyka zmiennopozycyjna pojedynczej precyzji używana przez opisane tu procedury jest odpowiednia w wielu typowych zastosowaniach w grafice, ale może być niewystarczająca np. w zastosowaniach CAD. W razie potrzeby należy skorzystać z procedur działających w podwójnej precyzji; można takie procedury otrzymać, zmieniając typ `GLfloat` na `GLdouble` w procedurach opisanych wyżej i modyfikując ich nazwy: przez analogię do konwencji przyjętych w nazewnictwie procedur OpenGL-a końcówkę `f` wypada zmienić na `d`.[7]

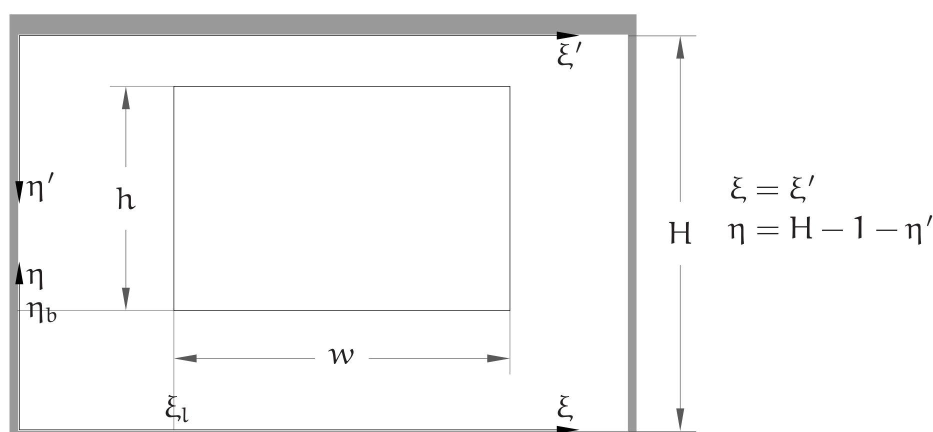
[7] Warto też rozważyć użycie profesjonalnej biblioteki procedur numerycznych, takiej jak LAPACK.

6. Rzutowanie

Dwa rodzaje rzutowania przestrzeni trójwymiarowej na płaszczyznę mają największe znaczenie w grafice komputerowej: rzutowanie równoległe i perspektywiczne. W obu przypadkach obrazem prostej w przestrzeni jest prosta lub punkt na płaszczyźnie i oba rodzaje rzutowania mogą być dokonane za pomocą liniowych przekształceń wektorów współrzędnych jednorodnych. W tym rozdziale są opisane konstrukcje macierzy takich przekształceń i procedury realizujące te konstrukcje.

Klatka, aspekt ekranu i kostka standardowa

Klatka (*viewport*) jest to prostokątny obszar w oknie, w którym ma być tworzony obraz. Klatka jest określona przez podanie czterech liczb całkowitych (typu `GLint`): ξ_l, η_b, w, h. Liczby ξ_l, η_b są współrzędnymi dolnego lewego narożnika klatki[1], zaś w i h to jej szerokość i wysokość. Jednostki osi to odpowiednio szerokość i wysokość jednego piksela.



Rysunek 6.1: Klatka w oknie

Uwaga: OpenGL posługuje się układem współrzędnych, którego początek znajduje się w *dolnym lewym* narożniku okna, oś ξ jest zorientowana w prawo, a oś η do góry. Systemy okien zazwyczaj używają układu ξ', η' o początku w górnym narożniku okna i z osią η' zorientowaną przeciwnie do osi η (zobacz rys. 6.1).

Często (choć nie zawsze) chcemy, aby klatka była całym oknem (albo całym podoknem np. FreeGLUTa). Aby ustawić wielkość i położenie w oknie klatki,

[1] Używam tu greckich liter ξ, η, aby odróżnić współrzędne na ekranie od współrzędnych x, y, z w układach w przestrzeni trójwymiarowej, np. w układzie kostki standardowej.

w której chcemy rysować, wykonujemy instrukcję

```
glViewport ( xi_l, eta_b, w, h );
```

albo

```
glViewport ( xiprime_l, H-h-etaprime_t, w, h );
```

w zależności od tego, czy chcemy podać współrzędne ξ_l, η_b dolnego lewego narożnika klatki w układzie współrzędnych OpenGL-a, czy współrzędne ξ_l', η_t' górnego lewego narożnika w układzie okna. Jeśli klatka ma zajmować całe okno o szerokości W i wysokości H, to podajemy parametry `( 0, 0, W, H )`.

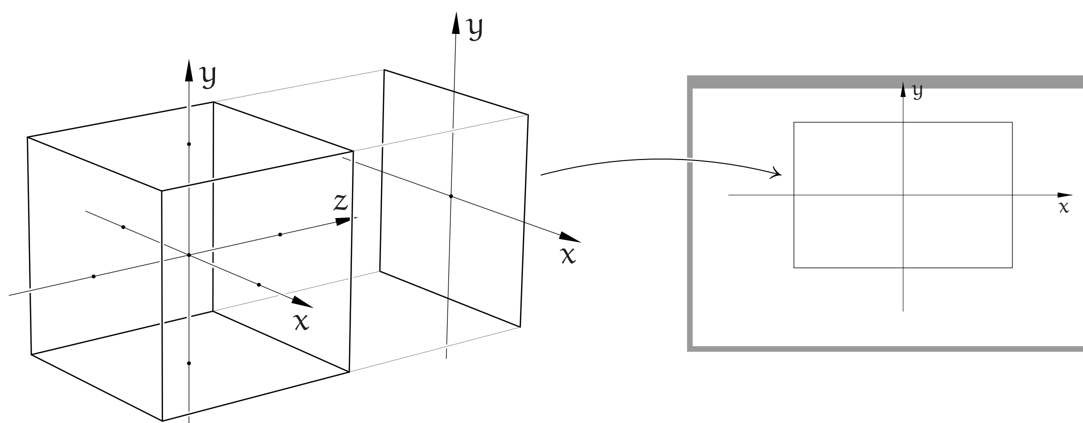
Ważnym parametrem każdego urządzenia rastrowego jest aspekt: określa on kształt piksela, którego wysokość nie musi być równa szerokości. Większość ekranów obecnie produkowanych monitorów ma współczynnik aspektu a, będący ilorazem szerokości i wysokości piksela, równy 1 (co oznacza, że piksele są kwadratowe), lub na tyle bliski 1, że można się nie przejmować błędem i przyjąć $a = 1$.[2] Jeśli klatka ma wysokość h pikseli i szerokość w pikseli, to jej fizyczne wymiary na ekranie pozostają w proporcji $aw : h$.

Kostka standardowa jest to sześcian o boku o długości 2: składa się z punktów, których wszystkie trzy współrzędne kartezjańskie mają wartości między -1 i 1. Zgodnie z opisem w poprzednim rozdziale, współrzędne kartezjańskie punktu w układzie, w którym jest określona kostka otrzymuje się ze współrzędnych jednorodnych X, Y, Z, W przez podzielenie pierwszych trzech przez ostatnią (wzór 5.1), przy czym obliczenia podczas obcinania prymitywów i rasteryzacja wykonywane są przy użyciu współrzędnych jednorodnych.

Właściwe rzutowanie punktów polega na odrzuceniu współrzędnej z (ale jest ona wykorzystywana w testach widoczności) oraz przeskalowaniu i przesunięciu współrzędnych x i y tak, aby ich przedziały zmienności ($[-1, 1]$ dla obu współrzędnych) przeszły odpowiednio na przedziały $[\xi_l, \xi_l + w]$ i $[\eta_b, \eta_b + h]$ (zobacz rys. 6.2).

Bryła widzenia, czyli obszar w przestrzeni, w którym zawarte są obiekty do

[2] Według informacji podanych przez system okien (zobacz s. 3.26), jeden z używanych przeze mnie monitorów ma rozdzielczość 118 pikseli na cal w poziomie i 117 pikseli na cal w pionie, a inny ma 93 piksele na cal w poziomie i 95 w pionie. Stąd aspekt pierwszego monitora $a = \frac{117}{118} = 0.99152\ldots$, a drugiego $a = \frac{95}{93} = 1.0215\ldots$. W obu przypadkach błąd spowodowany przez przyjęcie $a = 1$ jest naprawdę mały.



Rysunek 6.2: Kostka standardowa i jej odwzorowanie na klatkę w oknie

narysowania na obrazie, musi być przekształcony (przez jeden z szaderów części przedniej potoku przetwarzania grafiki: szadera wierzchołków, rozdrabniania lub geometrii) na kostkę standardową. Zatem aby uniknąć zniekształceń w postaci niejednakowego skalowania wymiarów poziomych i pionowych na obrazie, musimy podczas określania kształtu bryły widzenia uwzględnić proporcję fizycznych wymiarów klatki, określoną przez jej wymiary w pikselach i aspekt ekranu.

Rzutowanie perspektywiczne

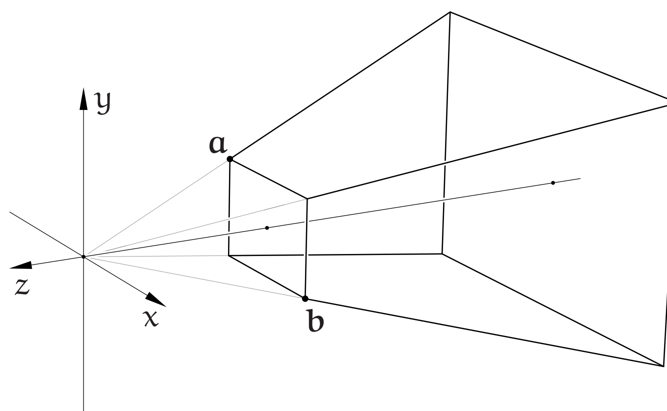
Bryła widzenia dla rzutowania perspektywicznego jest ściętym ostrosłupem o podstawie prostokątnej. Na rysunku 6.3 jest pokazana bryła zdefiniowana przy użyciu sześciu parametrów, które są używane jako standardowy opis w starym OpenGL-u i są na tyle wygodne, że warto tego opisu używać także w aplikacjach w nowym OpenGL-u[3]. Trzeba tylko go oprogramować.

Bryła (ostrosłup) widzenia jest opisany w układzie obserwatora, którego początek jest środkiem rzutowania. Rzutnia jest płaszczyzną równoległą do płaszczyzny xy.

Parametry opisujące bryłę widzenia mają nazwy `left`, `right`, `bottom`, `top`, `near` i `far`; we wzorach podanych niżej są oznaczane symbolami l, r, b, t, n i f. Podstawy ostrosłupa, przednia i tylna, są położone w płaszczyznach $z = -n$ i $z = -f$; liczby n i f są dodatnie, a zatem wszystkie punkty bryły widzenia mają współrzędną z *ujemną*. Zauważmy, że dzięki temu układ xyz z osiami x i y zorientowanymi w prawo i do góry jest *prawoskrętny*.

Zaznaczone na rysunku wierzchołki przedniej ściany $\mathbf{a} = (l, t, -n)$ i $\mathbf{b} = (r, b, -n)$ wyjaśniają znaczenie pozostałych parametrów; ponieważ boki tej (oraz tylnej)

[3] *Matematyka* używana w starym OpenGL-u nie zestarzała się nic a nic.



Rysunek 6.3: Bryła widzenia dla rzutowania perspektywicznego

ściany bryły widzenia są równoległe do osi x i y, a ponadto ściana tylna jest obrazem ściany przedniej w jednokładności o środku w początku układu i o skali f/n, bryła widzenia jest określona jednoznacznie.

Macierz P, która pomnożona przez wektor współrzędnych jednorodnych reprezentujący dowolny punkt z opisanego wyżej ostrosłupa widzenia da w wyniku wektor współrzędnych jednorodnych punktu z kostki standardowej[4], oraz odwrotność tej macierzy, są opisane wzorami

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \qquad P^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{n-f}{2fn} & \frac{n+f}{2fn} \end{bmatrix}.$$

Można to wykazać, biorąc wektory współrzędnych jednorodnych wierzchołków odpowiedniej bryły, np. $\boldsymbol{A} = [l, t, -n, 1]^T$ dla punktu a, i obliczając iloczyny macierzy P i tych wektorów: w szczególności, $P\boldsymbol{A} = [-n, n, -n, n]^T$. Po podzieleniu pierwszych trzech przez czwartą współrzędną każdego wektora otrzymanego w wyniku tego mnożenia dostajemy współrzędne kartezjańskie odpowiedniego wierzchołka kostki standardowej (dla punktu a to jest wierzchołek $[-1, 1, -1]^T$). Odwrotność macierzy P może się przydać podczas odtwarzania położenia punktu w przestrzeni na podstawie współrzędnych odpowiedniego punktu na obrazie.

Procedura `M4x4Frustumf` przedstawiona na listingu 6.1 wpisuje współczyniki tych macierzy do tablic wskazywanych przez parametry. Jeśli któraś macierz jest

[4]Macierz przekształcenia, które przeprowadza bryłę widzenia na kostkę standardową jest często nazywana macierzą rzutowania (*projection matrix*); jest to niewątpliwie skrót myślowy.

Listing 6.1: Procedura M4x4Frustumf

C

```c
 1: void M4x4Frustumf ( GLfloat a[16], GLfloat ai[16],
 2:                     float left, float right, float bottom,
 3:                     float top, float near, float far )
 4: {
 5:   float rl, tb, nf, nn;
 6:
 7:   rl = right-left;  tb = top-bottom;
 8:   nf = near-far;    nn = near+near;
 9:   if ( a ) {
10:     memset ( a, 0, 16*sizeof(GLfloat) );
11:     a[0] = nn/rl;           a[8] = (right+left)/rl;
12:     a[5] = nn/tb;           a[9] = (top+bottom)/tb;
13:     a[10] = (far+near)/nf;  a[14] = far*nn/nf;
14:     a[11] = -1.0;
15:   }
16:   if ( ai ) {
17:     memset ( ai, 0, 16*sizeof(GLfloat) );
18:     ai[0] = rl/nn;  ai[12] = (right+left)/nn;
19:     ai[5] = tb/nn;  ai[13] = (top+bottom)/nn;
20:     ai[14] = -1.0;
21:     ai[11] = nf/(far*nn);
22:     ai[15] = (far+near)/(far*nn);
23:   }
24: } /*M4x4Frustumf*/
```

niepotrzebna, to można zamiast tablicy podać parametr NULL.

Zwróćmy uwagę, że przekształcenie współrzędnej *kartezjańskiej* z w odwzorowaniu ostrosłupa widzenia na kostkę standardową jest nieliniowe; w istocie jest to funkcja homograficzna. W przedziale $[-f, -n]$ funkcja ta jest monotonicznie malejąca, dzięki czemu testy widoczności wykonywane przez porównywanie współrzędnych z punktów w kostce standardowej (bliżej obserwatora jest ten punkt, którego współrzędna z jest mniejsza) są poprawne.

Ustalając wartości parametrów n i f, należy zachować umiar. Oczywiście, obiekty, które chcemy przedstawić na obrazie, muszą się mieścić w bryle widzenia, a więc zakres odległości określony przez te parametry musi być dostatecznie duży, ale im jest on większy, tym *mniejszą* dokładność będą mieć testy widoczności. Iloraz f/n powinien zatem być możliwie mały. Najlepiej, aby nie przekraczał rzędu kilkanaście, w ostateczności kilkadziesiąt.

Aby zapewnić jednakowe skalowanie wymiarów poziomych i pionowych na obrazie w klatce o szerokości w i wysokości h pikseli utworzonym na ekranie, który ma współczynnik aspektu a, należy przyjąć parametry spełniające warunek $r - l : t - b = aw : h$. Liczba $d = \sqrt{(r-l)^2 + (t-b)^2}$ jest długością przekątnej przedniej ściany ostrosłupa widzenia. Iloraz n/d określa rodzaj „obiektywu" realizującego rzutowanie: jeśli jest mały, to mamy obiektyw szerokokątny, zaś ze wzrostem tego ilorazu otrzymujemy coraz większe zbliżenia obiektów na obrazie. W klasycznej fotografii małoobrazkowej klatka na błonie światłoczułej ma wymiary 36mm × 24mm, zatem jej przekątna ma długość ok. 43mm. Obiektywy o ogniskowej 50mm nazywane standardowymi były chyba najczęściej używane w fotografii amatorskiej; dla takich obiektywów $n/d \approx 1.15$.[5] W grafice komputerowej lepiej jest przyjmować parametry, dla których iloraz n/d jest większy (orientacyjnie od 2 do ok. 5) i oddalić się od przedmiotów[6], aby osłabić efekt przerysowania.

Przyjęcie parametrów $l = -r$ i $b = -t$ daje symetryczny ostrosłup widzenia; punkty na jego osi symetrii są rzutowane na środek klatki. Zdarzają się sytuacje, gdy potrzebne są niesymetryczne bryły widzenia: najczęściej wtedy, gdy chcemy wykonać parę obrazów do stereoskopii.

Rzutowanie równoległe

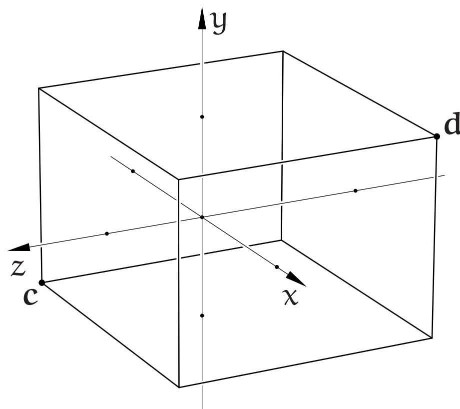
Rzutowanie równoległe jest podstawą rysunku technicznego. Bryła widzenia dla takiego rzutu jest prostopadłościanem. Sześć parametrów opisujących tę bryłę w układzie obserwatora ma te same nazwy, co parametry dla rzutowania perspektywicznego. Przekształcenie bryły widzenia na kostkę standardową jest afiniczne, a dokładniej, każda ze współrzędnych x, y, z jest poddawana niezależnie przekształceniu afinicznemu (tj. skalowaniu z przesunięciem).

Na rysunku 6.4 mamy narysowaną bryłę widzenia; są na nim zaznaczone punkty $\mathbf{c} = (l, b, -n)$ i $\mathbf{d} = (r, t, -f)$. Macierz określająca przekształcenie tej bryły na kostkę standardową i odwrotność tej macierzy są opisane wzorami

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad P^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{l+r}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{b+t}{2} \\ 0 & 0 & \frac{n-f}{2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

[5] Tradycyjnie w fotografii symbol f jest używany do oznaczenia długości ogniskowej obiektywu, zaś n oznacza względny otwór przysłony. Usiłujemy się tym nie przejmować.

[6] O ile jest to możliwe; obserwator w grze komputerowej zwykle musi widzieć scenę „fotografowaną" przez kamerę znajdującą się w pomieszczeniu z przedmiotami do narysowania.



Rysunek 6.4: Bryła widzenia dla rzutowania równoległego

Listing 6.2 przedstawia procedurę, która konstruuje macierz przekształcenia takiej bryły na kostkę standardową, a także macierz przekształcenia odwrotnego.

Listing 6.2: Procedura M4x4Orthof

C

```
1: void M4x4Orthof ( GLfloat a[16], GLfloat ai[16],
2:                   float left, float right, float bottom,
3:                   float top, float near, float far )
4: {
5:   float rl, tb, fn;
6:
7:   rl = right-left;  tb = top-bottom;  fn = far-near;
8:   if ( a ) {
9:     memset ( a, 0, 16*sizeof(GLfloat) );
10:    a[0] = 2.0/rl;    a[12] = -(right+left)/rl;
11:    a[5] = 2.0/tb;    a[13] = -(top+bottom)/tb;
12:    a[10] = -2.0/fn;  a[14] = (far+near)/fn;
13:    a[15] = 1.0;
14:  }
15:  if ( ai ) {
16:    memset ( ai, 0, 16*sizeof(GLfloat) );
17:    ai[0] = 0.5*rl;    ai[12] = 0.5*(right+left);
18:    ai[5] = 0.5*tb;    ai[13] = 0.5*(top+bottom);
19:    ai[10] = -0.5*fn;  ai[14] = 0.5*(far+near);
20:    ai[15] = 1.0;
21:  }
22: } /*M4x4Orthof*/
```

Dla $n < f$ współczynnik $\frac{2}{n-f}$ w trzecim wierszu i kolumnie jest ujemny, a ostatni współczynnik na diagonali macierzy P jest dodatni. W konsekwencji

przekształcenie współrzędnej z jest funkcją malejącą. Jeśli dwa punkty po zrzutowaniu mają ten sam obraz, to punkt, którego współrzędna z *w układzie kostki standardowej* jest mniejsza, zasłania drugi punkt[7]. Ponieważ odwzorowanie przedziału zmienności współrzędnej z jest afiniczne, „rozdzielczość” głębokości (tj. informacji o głębokości punktów) w buforze głębokości OpenGL-a jest stała w całej bryle widzenia.

Przekształcenia rzutowanych wierzchołków

Położenie wierzchołka wprowadzanego do potoku przetwarzania grafiki przez etap pobierania wierzchołków jest zwykle reprezentowane przez wektor o czterech współrzędnych, X, Y, Z, W. To są opisane w rozdziale 5 współrzędne jednorodne. W procesie tworzenia obrazu używamy wielu układów współrzędnych (kartezjańskich i związanych z nimi współrzędnych jednorodnych) w przestrzeni, w której znajduje się rysowana scena. Wyróżnimy cztery układy:

Układ współrzędnych modelu — w tym układzie podajemy współrzędne wierzchołków obiektu; zazwyczaj to one są przechowywane w pamięci GPU w odpowiednim buforze i to te współrzędne etap pobierania wierzchołków podaje na wejście szadera wierzchołków.

Układ współrzędnych świata — wyróżniony układ, w którym poszczególne obiekty są odpowiednio rozmieszczone względem siebie. W tym układzie trzeba określić położenie obserwatora i w tym układzie jest obliczane oświetlenie obiektów.

Układ współrzędnych obserwatora — początek tego układu (w przypadku rzutowania perspektywicznego) jest położeniem obserwatora, który patrzy w kierunku ujemnej półosi z. W tym układzie opisujemy bryłę widzenia.

Układ kostki standardowej — współrzędne wierzchołków w tym układzie są przekazywane z części przedniej potoku przetwarzania grafiki do etapu obcinania, po którym dokonywana jest rasteryzacja, przeprowadzana także w tym układzie współrzędnych[8].

[7] Zauważmy, że tu nie można mówić o odległości tych punktów od obserwatora.

[8] Ściśle biorąc, wyróżniamy cztery *klasy* układów współrzędnych. Każdy obiekt (model) może być zdefiniowany w swoim układzie, innym niż pozostałe obiekty. Możemy mieć też więcej niż jednego obserwatora (np. w stereoskopii, podczas tworzenia obrazów obiektów odbitych w lustrze lub obserwatorów związanych ze źródłami światła w algorytmie cieni) i każdy obserwator będzie miał inną bryłę widzenia, a więc też inaczej określony w świecie układ współrzędnych kostki standardowej. Tylko układ świata jest tylko jeden.

W części przedniej potoku przetwarzania grafiki szadery muszą dokonać przejścia od współrzędnych podanych w układzie modelu do układu kostki standardowej. Obliczenie to może być wykonane przez *dowolny* szader części przedniej — wierzchołków, sterowania rozdrabnianiem, rozdrabniania lub geometrii. Potrzebne przekształcenie jest złożeniem trzech przekształceń opisanych przez macierze 4×4: macierz M opisuje przejście od układu modelu do układu świata, macierz V opisuje przejście od układu świata do obserwatora, macierz P opisuje przejście od układu obserwatora do układu kostki standardowej. Jeśli symbolem $\boldsymbol{A}$ oznaczymy wektor współrzędnych jednorodnych wierzchołka w układzie modelu, to któryś szader części przedniej ma obliczyć wektor

$$\boldsymbol{A}' = PVM\boldsymbol{A}$$

i przekazać go dalej. Macierze M, V i P są podawane zazwyczaj w odpowiednich zmiennych jednolitych (pierwszy przykład zobaczymy już wkrótce).

Uwaga: Jeśli obliczenie oświetlenia ma wykonać szader fragmentów, to trzeba też przekazać dalej wektor $M\boldsymbol{A}$ zawierający współrzędne wierzchołka w układzie świata.

Uwaga: Można utożsamić układy modelu i świata lub świata i obserwatora. W tym celu wystarczy przyjąć macierz M lub V jednostkową, można też napisać szader, który nie uwzględnia odpowiedniego przejścia między układami, tj. nie wykonuje mnożenia wektora przez macierz tego przejścia, co jest równoważne pomnożeniu tego wektora przez macierz jednostkową.

Dodatkowo szadery mogą przeprowadzać obliczenia w układach współrzędnych (kartezjańskich lub barycentrycznych) określonych w dziedzinie rozdrabnianego płata lub w dziedzinie tekstury; tym będziemy się zajmować w rozdziałach 12–13 i 17–20.

Rzutowanie dla grafiki dwuwymiarowej

W pewnych sytuacjach występuje potrzeba określenia takiego rzutowania, aby współrzędne x, y w układzie obserwatora (albo modelu — patrz wyżej) były tożsame ze współrzędnymi ξ', η' w oknie (początek tego układu, przypomnijmy, jest w lewym górnym narożniku okna, współrzędna η' rośnie do dołu, a jednostki osi mają długości równe szerokości i wysokości jednego piksela). Najczęściej jest to potrzebne wtedy, gdy chcemy wykorzystać OpenGL-a do wykonania grafiki dwuwymiarowej, na przykład narysować menu z wihajstrami w oknie. Aby określić takie przekształcenie dla klatki, której górny lewy narożnik ma

współrzędne ξ', η' i która ma szerokość w i wysokość h, położonej w oknie o wysokości H, możemy wykonać następujące instrukcje:

```
glViewport ( xiprime, H-h-etaprime, w, h );
M4x4Orthof ( a, NULL, (float)xiprime-0.5, (float)(xiprime+w)-0.5,
             (float)(H-h-etaprime)-0.5, (float)(H-1-etaprime)-0.5,
             -1.0, 1.0 );
```

a następnie załadować współczynniki macierzy P przekształcenia z tablicy a do odpowiedniej zmiennej jednolitej. Jeśli chcemy, aby układ miał początek w górnym lewym narożniku *klatki* (która nie musi być całym oknem), to odpowiednią macierz przekształcenia otrzymamy, wykonując instrukcję

```
M4x4Orthof ( a, NULL, -0.5, (float)w-0.5,
             (float)h-0.5, -0.5, -1.0, 1.0 );
```

Zauważmy korekty przesunięcia w pionie i poziomie o pół piksela; one są wprowadzone dlatego, że linie $\eta' = 0$ i $\xi' = 0$ przechodzą przez *środki* pikseli odpowiednio w pierwszym wierszu i w pierwszej kolumnie pikseli w oknie lub klatce. Chcemy, aby liczby *całkowite* x, y, będące współrzędnymi punktu w bryle widzenia, po przekształceniu odpowiadały prostym przechodzącym przez środki pikseli w odpowiedniej kolumnie i wierszu rastra w oknie.

7. Pierwsza aplikacja

Aplikacja przedstawiona w tym rozdziale wyświetla obraz dwudziestościanu foremnego; powstała przez wypełnienie szkieletu aplikacji FreeGLUTa z rozdziału 3 odpowiednimi procedurami. Najpierw przedstawimy szadery, nieformalnie omawiając użyte w nich konstrukcje języka GLSL, a następnie procedury w C, które robią wszystko co trzeba aby powstał obraz.

Szadery

Program, który zostanie użyty w tej aplikacji składa się z tylko dwóch szaderów, wierzchołków i fragmentów, przedstawionych na listingach 7.1 i 7.2.

Listing 7.1: Szader wierzchołków pierwszej aplikacji

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec4 in_Colour;
5:
6: out Vertex {
7:   vec4 Colour;
8: } Out;
9:
10: uniform TransBlock {
11:   mat4 mm;
12:   mat4 vm;
13:   mat4 pm;
14: } trb;
15:
16: void main ( void )
17: {
18:   gl_Position = trb.pm * (trb.vm * (trb.mm * in_Position));
19:   Out.Colour = in_Colour;
20: } /*main*/
```

Pierwsza linia każdego szadera[1] musi zawierać informację o użytej wersji języka GLSL, w postaci liczby całkowitej podanej po symbolu preprocesora `#version`. W naszym przykładzie jest to wersja 4.2.

[1]która nie jest pusta lub nie zawiera tylko spacji i komentarzy

W liniach 3 i 4 są opisane atrybuty wierzchołka danego na wejściu; dane wejściowe są zadeklarowane z użyciem słowa kluczowego `in`. Słowo kluczowe `layout` z zawartością podanego za nim nawiasu jest kwalifikatorem (*qualifier*) zmiennej; informacje podane w kwalifikatorach w liniach 3 i 4 definiują numery miejsc atrybutów wierzchołka; atrybut `in_Position`, który jest wektorem współrzędnych jednorodnych położenia wierzchołka, ma numer miejsca `0`, zaś atrybut `in_Colour`, który jest wektorem współrzędnych RGBA koloru wierzchołka, ma numer miejsca 1. Aplikacja, tworząc VAO[2], podaje te numery razem z informacją, gdzie w tablicy wierzchołków należy odnaleźć odpowiednie atrybuty.

Szader wierzchołków zawsze powinien nadać wartość zmiennej `gl_Position`, która jest częścią struktury wyjściowej (zadeklarowanej „odgórnie"). Dodatkowy atrybut wierzchołka przetworzonego, w tym przykładzie kolor, jest wpisywany do odpowiedniego pola struktury wyjściowej (`out`, linie 6–8), która ma *dwie* nazwy: globalną `Vertex` i prywatną `Out`.

Dwie nazwy struktur są użyteczne, ponieważ szadery komunikują się ze sobą: *wynik* obliczeń szadera wierzchołków trafi (po wszystkich pośrednich etapach przetwarzania w potoku) na *wejście* szadera fragmentów. Nazwa globalna w obu szaderach musi (co jest oczywiste) być taka sama, natomiast lokalnie jest używana nazwa krótsza, która dodatkowo podkreśla, co jest daną wejściową, a co wynikiem obliczeń konkretnego szadera.

W liniach 10–14 jest zadeklarowany blok zmiennych jednolitych, który zawiera trzy macierze; pierwsza (`trb.mm`) to macierz M, która opisuje przejście od układu współrzędnych, w którym jest zdefiniowany obiekt (tj. w którym są podane współrzędne jego wierzchołków) do układu świata, druga macierz, V, (`trb.vm`) opisuje przejście od układu świata do układu obserwatora, a trzecia macierz, P, (`trb.pm`) reprezentuje przekształcenie bryły widzenia na kostkę standardową, skonstruowane zgodnie z opisem w poprzednim rozdziale. Blok zmiennych jednolitych też ma dwie nazwy. Nazwa globalna `TransBlock` jest używana przez aplikację w celu uzyskania dostępu do tych zmiennych jednolitych.

Dygresja. W starym OpenGL-u przekształcenie wierzchołków jest wykonywane przy użyciu dwóch macierzy, o nazwach *ModelView* i *Projection*. Pierwsza z nich jest iloczynem macierzy przejścia od układu obiektu do układu świata i od układu świata do układu obserwatora; druga z nich opisuje przekształcenie bryły widzenia na kostkę standardową. Z powodów praktycznych (z którymi spotkamy

[2] *vertex array object*, obiekt tablicy wierzchołków

się dalej) chyba lepiej jest rozdzielić przekształcenia, których złożenie opisuje pierwsza z tych macierzy, tak jak to tu zrobiłem.

Obliczenie realizuje procedura main (linie 16–20), która musi mieć pustą listę parametrów i pustą wartość. W linii 18 obliczany jest wektor współrzędnych położenia wierzchołka w kostce standardowej; zwracam uwagę na nawiasy, które z pięciu możliwych sposobów obliczania iloczynu trzech macierzy i wektora wybierają sposób najtańszy. Atrybut koloru w linii 18 jest po prostu kopiowany do zmiennej w bloku wyjściowym.

Listing 7.2: Szader fragmentów pierwszej aplikacji

GLSL

```
1: #version 420
2:
3: in Vertex {
4:   vec4 Colour;
5: } In;
6:
7: out vec4 out_Colour;
8:
9: void main ( void )
10: {
11:   out_Colour = In.Colour;
12: } /*main*/
```

Szader fragmentów na listingu 7.2 spełnia bardzo proste zadanie: kopiuje kolor otrzymany w strukturze wejściowej na wynik, który zgodnie z deklaracją w linii 7 ma być przypisany do zmiennej out_Colour.

Należy zwrócić uwagę, że struktura Vertex w obu szaderach ma *identyczną* budowę, w szczególności typy i nazwy wszystkich pól; różne są tylko prywatne nazwy tej struktury. Szader fragmentów wykonuje obliczenie dla pojedynczego piksela obrazu odpowiedniego prymitywu (punktu, odcinka lub trójkąta) i ma za zadanie tylko podać kolor (lub ewentualnie podjąć decyzję o zaniechaniu rysowania tego piksela). Ale jaki jest kolor podany w strukturze Vertex, jeśli wierzchołki trójkąta mają różne kolory, a piksel odpowiada punktowi innemu niż wierzchołek? Otóż, wszelkie atrybuty są interpolowane między wartościami podanymi dla wierzchołków. W ten sposób realizowane jest cieniowanie (*shading*, stąd nazwa szadery) i na przykład piksel odpowiadający środkowi ciężkości trójkąta będzie miał kolor otrzymany jako średnia z kolorów wierzchołków trójkąta. Kolor jest atrybutem wektorowym; interpolowane są wszystkie cztery jego współrzędne.

Przygotowanie programu szaderów

Listing 7.3 przedstawia procedurę LoadMyShaders którą trzeba wpisać w miejsce pustej procedury w linii 20 na listingu 3.1, oraz zmienne globalne aplikacji, w których będą zapamiętane identyfikatory szaderów i inne informacje konieczne do współpracy części aplikacji działających na CPU i GPU. Zakładam, że opisane wcześniej szadery wierzchołków i fragmentów są zapisane (w całości) w plikach o nazwach app1.glsl.vert i app1.glsl.frag.

Listing 7.3: Procedura LoadMyShaders

```c
GLuint shader_id[2];
GLuint program_id;
GLuint trbi, trbuf;
GLint  trbsize, trbofs[3];

void LoadMyShaders ( void )
{
  static const char *filename[] =
    { "app1.glsl.vert", "app1.glsl.frag" };
  static const GLchar *UTBNames[] =
    { "TransBlock", "TransBlock.mm", "TransBlock.vm", "TransBlock.pm" };
  GLuint ind[3];

  shader_id[0] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[0] );
  shader_id[1] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1, &filename[1]);
  program_id = LinkShaderProgram ( 2, shader_id );
  trbi = glGetUniformBlockIndex ( program_id, UTBNames[0] );
  glGetActiveUniformBlockiv ( program_id, trbi,
                              GL_UNIFORM_BLOCK_DATA_SIZE, &trbsize );
  glGetUniformIndices ( program_id, 3, &UTBNames[1], ind );
  glGetActiveUniformsiv ( program_id, 3, ind, GL_UNIFORM_OFFSET, trbofs );
  glUniformBlockBinding ( program_id, trbi, 0 );
  glGenBuffers ( 1, &trbuf );
  glBindBufferBase ( GL_UNIFORM_BUFFER, 0, trbuf );
  glBufferData ( GL_UNIFORM_BUFFER, trbsize, NULL, GL_DYNAMIC_DRAW );
  ExitIfGLError ( "LoadMyShaders" );
} /*LoadMyShaders*/
```

W liniach 14–16 procedura LoadMyShaders wywołuje opisane w rozdz. 4 procedury, które czytają z pliku i kompilują szadery (nazwy plików są podane w linii 9), a następnie łączą je w program. Identyfikatory szaderów są zapisywane w tablicy shader_id, a identyfikator programu w zmiennej program_id.

Program szaderów korzysta z bloku zmiennych jednolitych, który zawiera trzy macierze 4×4 o współczynnikach rzeczywistych (zmiennopozycyjnych pojedynczej precyzji). Instrukcje w liniach 17–22 wyciągają z programu szaderów niezbędne informacje o bloku zmiennych jednolitych. W liniach 23–25 jest tworzony i przygotowywany do pracy obiekt bufora zmiennej jednolitej (*uniform buffer object*, *UBO*), w którym będą przechowywane macierze przekształceń.

Procedura glGetUniformBlockIndex w linii 17 zwraca indeks bloku zmiennych jednolitych — jest to identyfikator bloku w utworzonym programie szaderów. Parametry procedury to identyfikator programu i napis ASCIIZ, który jest *globalną* nazwą struktury zadeklarowanej w programie szaderów (zobacz linię 11 na listingu 7.3 i linię 10 na listingu 7.1).

Procedura glGetActiveUniformBlockiv w linii 18–19 przypisuje zmiennej trbsize wielkość (w bajtach) bloku; bufor, który utworzymy dla tego bloku musi mieć co najmniej taką wielkość.

Następnie procedura glGetUniformIndices wyciąga z programu informacje o poszczególnych polach struktury będącej zmienną jednolitą. Mamy tu trzy pola (stąd drugi parametr ma wartość 3), których nazwy (napisy ASCIIZ) są podane w tablicy napisów przekazanej jako parametr trzeci. Parametr czwarty jest tablicą, do której zostają wpisane indeksy (identyfikatory) tych pól.

W linii 21 procedura glGetActiveUniformsiv wyciąga informacje o przesunięciu (w bajtach) poszczególnych pól w strukturze zmiennej jednolitej względem jej początku. Zwróćmy uwagę na sposób używania indeksów pól struktury otrzymanych w poprzedniej instrukcji.

W linii 22 jest wywołana procedura glUniformBlockBinding. Jej zadaniem jest określenie, że blok zmiennych jednolitych TransBlock ma być skojarzony z punktem dowiązania o numerze 0 w celu GL_UNIFORM_BUFFER w kontekście OpenGL-a[3]. Warto tu odwołać się do zamieszczonego w pierwszym rozdziale opisu kojarzenia UBO z blokami zmiennych jednolitych (zobacz rysunek 1.2). Utworzony przez dalsze instrukcje bufor, do którego aplikacja będzie wpisywać wartości zmiennych jednolitych w bloku TransBlock szadera, będzie przywiązany do punktu 0.

[3]Numer punktu dowiązania bloku zmiennych jednolitych można jawnie zapisać w treści szadera, poprzedzając blok, tj. słowo kluczowe uniform, kwalifikatorem np. layout(binding=1). Procedura glUniformBlockBinding jest władna zmienić ten numer.

Procedura `glGenBuffers` w linii 23 tworzy (w tym przypadku jeden) obiekt bufora — w tym momencie jeszcze nie wiadomo, jaki długi to będzie bufor, jakie w nim będą dane ani jak będzie używany. Jego identyfikator ląduje w zmiennej `trbuf`, z której zrobiliśmy *ad hoc* tablicę o długości 1, ponieważ ostatni parametr procedury `glGenBuffers` ma być tablicą.

Procedura `glBindBufferBase` w linii 24 przywiązuje nowy bufor do celu `GL_UNIFORM_BUFFER`; dalsze działania na buforze odbywają się za pośrednictwem tego celu.

Podstawowa procedura, która przywiązuje bufor do wskazanego celu, ma nazwę `glBindBuffer`; będziemy jej wielokrotnie używać. W kontekście OpenGL-a jest kilkanaście celów, z których każdy ma swoje zastosowania. Pewne cele, w tym właśnie `GL_UNIFORM_BUFFER`, są indeksowane, tj. zawierają tablice punktów dowiązania. Procedura `glBindBufferBase` może być wywołana dla takiego celu z tablicą; oprócz przywiązania bufora do samego celu, przywiązuje go również do wskazanego przez drugi parametr punktu dowiązania w tym celu (nie zmieniając, co ważne, pozostałych punktów dowiązania). Jeśli program szaderów zawiera wiele bloków zmiennych jednolitych, to każdy UBO będzie przywiązany do innego, (mam nadzieję) właściwego punktu dowiązania.

Procedura `glBufferData` w zasadzie służy do przesłania danych z pamięci CPU do bufora, ale w linii 25 jako adres tablicy z danymi jest podany parametr `NULL`. Rzecz w tym, że jest to sposób na ustalenie długości bufora (bo to jest dokonywane w momencie pierwszego wywołania procedury `glBufferData` dla bufora) oraz postulowanego sposobu wykorzystywania bufora — stała `GL_DYNAMIC_DRAW` wskazuje, że będziemy chcieli dane do bufora wpisywać wielokrotnie. Zmienna `trbsize`, przechowująca potrzebną wielkość bufora, otrzymała wartość w linii 19.

Przygotowanie obiektów w programie

Procedura `InitMyObject`, pusta na listingu 3.1 (linia 21) ma za zadanie określenie przekształceń, tzn. obliczenie współczynników macierzy przekształcenia modelu i przejścia do układu obserwatora i umieszczenie ich w bloku zmiennych jednolitych oraz przygotowanie reprezentacji obiektu. Podprogramy wywoływane przez tę procedurę są opisane niżej.

Listing 7.4: Procedura InitMyObject

C

```
1: void InitMyObject ( void )
2: {
3:   InitModelMatrix ();
4:   InitViewMatrix ();
5:   ConstructIcosahedronVAO ();
6: } /*InitMyObject*/
```

Przekształcenia współrzędnych

Macierz przekształcenia modelu w pierwszej aplikacji będzie jednostkowa; to oznacza, że układ, w którym są podane współrzędne wierzchołków jest układem świata. Oczywiście, w ramach eksperymentów można to będzie zmienić. Procedura InitModelMatrix na listingu 7.5 wpisuje współczynniki macierzy jednostkowej do UBO, w miejscu odpowiadającym zmiennej jednolitej TransBlock.mm. Miejsce to jest określone przez podanie przesunięcia pola mm struktury TransBlock względem początku tej struktury — wielkość (w bajtach) tego przesunięcia (i przesunięć pozostałych dwóch pól) uzyskaliśmy, wykonując instrukcje w liniach 20–21 na listingu 7.3.

Listing 7.5: Procedury InitModelMatrix i InitViewMatrix

C

```
1: void InitModelMatrix ( void )
2: {
3:   GLfloat m[16];
4:
5:   M4x4Identf ( m );
6:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
7:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0], 16*sizeof(GLfloat), m );
8:   ExitIfGLError ( "InitModelMatrix" );
9: } /*InitModelMatrix*/
10:
11: void InitViewMatrix ( void )
12: {
13:   GLfloat m[16];
14:
15:   M4x4Translatef ( m, 0.0, 0.0, -10.0 );
16:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
17:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), m );
18:   ExitIfGLError ( "InitViewMatrix" );
19: } /*InitViewMatrix*/
```

Procedura glBindBuffer w liniach 6 i 16 przywiązuje UBO (czyli bufor o identyfikatorze trbuf) do celu GL_UNIFORM_BUFFER. Szczerze mówiąc, w tej aplikacji jest to zbędne (bo nie ma w niej innych buforów przywiązywanych do tego celu, więc wystarczyło zrobić to tylko raz, po utworzeniu bufora), ale jak zaczniemy aplikację przerabiać, to oszczędzimy sobie niespodzianek. Procedura glBufferSubData wpisuje do bufora podaną w trzecim parametrze liczbę bajtów, zaczynając od miejsca (przesunięcia) określonego przez drugi parametr. Procedury glBufferSubData można użyć *po ustaleniu wielkości* bufora — to nastąpiło podczas pierwszego wywołania procedury glBufferData dla tego bufora.

Uwaga: W OpenGL 4.5 można przesyłanie danych do buforów wykonać prościej, używając procedur glNamedBufferData i glNamedBufferSubData, których pierwszy parametr jest identyfikatorem *bufora*, a nie *celu*; pozostałe parametry są takie same jak parametry procedur glBufferData i glBufferSubData. Wywołań procedur glNamedBufferData i glNamedBufferSubData nie trzeba poprzedzać wywołaniem procedury glBindBuffer. Nadal jednak bufor musi być przywiązany do określonego celu gdy wymagają tego pewne procedury, na przykład opisane dalej procedury glVertexAttribPointer i glDrawElements.

Dwudziestościan, który będziemy rysować, jest wpisany w kulę o promieniu 1 i środku $[0,0,0]^T$ (tj. w początku układu świata). Na listingu 7.5 jest pokazana procedura inicjalizacji macierzy przejścia od układu świata do układu obserwatora, którego początek znajduje się w punkcie $[0,0,10]^T$ — przekształcenie jest przesunięciem o wektor $[0,0,-10]^T$.

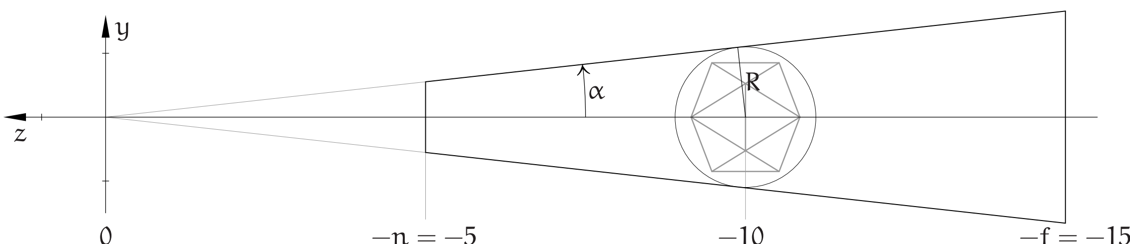
Macierz przekształcenia ostrosłupa widzenia na kostkę standardową musi być wyznaczona po nadaniu oknu wymiarów początkowych i po każdej zmianie jego wymiarów. Dlatego to obliczenie jest przeprowadzane w procedurze ReshapeFunc (linia 12 na listingu 3.1) zarejestrowanej przez glutReshapeFunc, zamiast w procedurze inicjalizacji.

Umieściliśmy obserwatora w odległości 10 od środka obiektu. Chcemy dobrać rzutowanie tak, aby kula o tym samym środku i promieniu $R = 1.1$ miała obraz o wysokości klatki zajmującej całe okno. Aby tak było, kąt dwuścienny[4] między płaszczyznami dolnej i górnej ściany ostrosłupa widzenia musi być równy

$$2\alpha = 2\arcsin\frac{1.1}{10} \approx 0.22045.$$

Przyjmuję (arbitralnie) parametry ostrosłupa widzenia $n = 5$, $f = 15$, co daje aż

[4] Miary kątów obliczam konsekwentnie w radianach.



Rysunek 7.1: Bryła widzenia w układzie obserwatora

nadto wystarczający zakres odległości dla naszego obiektu. Wtedy musi być[5]

$$t = -b = n \operatorname{tg} \alpha \approx 0.5533.$$

Parametry l i r należy obliczyć na podstawie wymiarów okna. Listing 7.6 przedstawia procedurę ReshapeFunc, wykonującą obliczenia na podstawie powyższych rachunków; przyjęty współczynnik aspektu ekranu jest równy 1.[6]

Listing 7.6: Procedura ReshapeFunc

C
```
1: void ReshapeFunc ( int width, int height )
2: {
3:   GLfloat m[16];
4:   float lr;
5:
6:   glViewport ( 0, 0, width, height );      /* klatka jest całym oknem */
7:   lr = 0.5533*(float)width/(float)height;  /* przyjmujemy aspekt równy 1 */
8:   M4x4Frustumf ( m, NULL, -lr, lr, -0.5533, 0.5533, 5.0, 15.0 );
9:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
10:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), m );
11:  ExitIfGLError ( "ReshapeFunc" );
12: } /*ReshapeFunc*/
```

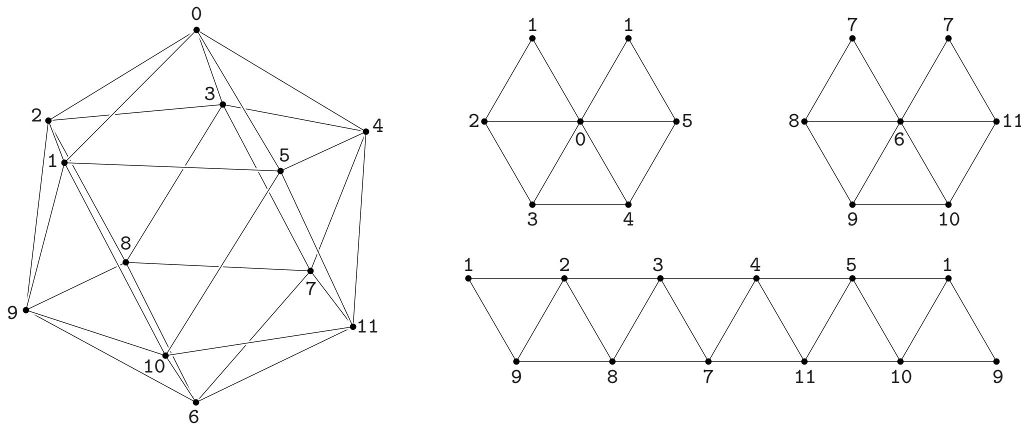
Ustawianie klatki i macierzy rzutowania w procedurze ReshapeFunc jest poprawnym rozwiązaniem jeśli zawsze klatka jest całym oknem, w którym obraz jest wykonywany przy użyciu jednego tylko rzutu. Gdyby tak nie było (w bardziej skomplikowanej aplikacji), to odpowiednie dane trzeba umieszczać w kontekście OpenGL-a w procedurze RedrawFunc bezpośrednio przed rysowaniem obiektów w poszczególnych klatkach w oknie.

[5] W tym obliczeniu dokładność do czterech cyfr dziesiętnych jest aż nadto wystarczająca.

[6] Dla ciekawości: gdyby okno, czyli klatka, miało proporcje szerokości i wysokości jak $3:2$, to przekątna przedniej ściany ostrosłupa widzenia miałaby długość $d \approx 1.995$. Wtedy $n/d \approx 2.506$, co odpowiada obiektywowi o długości ogniskowej 125mm. W fotografii to jest obiektyw długoogniskowy, ale jeszcze nie teleobiektyw.

Tworzenie obiektu tablicy wierzchołków

Rysunek 7.2 przedstawia wierzchołki i krawędzie dwudziestościanu; liczby na rysunku oznaczają numery wierzchołków. W aplikacji będziemy chcieli rysować dwudziestościan na trzy sposoby: jako zbiór wierzchołków, szkielet zbudowany z krawędzi albo jako wielościan (czyli zbiór nieprzezroczystych ścian).



Rysunek 7.2: Model dwudziestościanu foremnego: wachlarze i taśma trójkątowa

Każdy z dwunastu wierzchołków dwudziestościanu jest końcem pięciu krawędzi i wierzchołkiem pięciu ścian. Procedura `glDrawArrays` może rysować odcinki i trójkąty, biorąc pary lub trójki kolejnych punktów z tablicy wierzchołków. Ale wtedy każdy wierzchołek musiałby wystąpić w tablicy pięciokrotnie, czego wolimy uniknąć. W tym celu do rysowania użyjemy procedury `glDrawElements`. Procedura ta posługuje się dodatkową tablicą liczb całkowitych — indeksów do tablicy wierzchołków; indeksy zajmują znacznie mniej miejsca, a ponadto w ten sposób eliminowane są błędy w kopiowaniu współrzędnych wierzchołka[7]. Tablica ta będzie przechowywana w osobnym buforze, który bywa określany skrótem IBO (od *index buffer object*).

Aby skrócić tablicę indeksów, zamiast oddzielnych odcinków lub trójkątów będziemy rysować łamane lub taśmy albo wachlarze trójkątów. Dla każdego odcinka lub trójkąta, z wyjątkiem pierwszego, trzeba podać tylko jeden indeks. Taśma i wachlarze trójkątów są narysowane schematycznie na rysunku 7.2; procedura konstruująca VAO z modelem (tj. atrybutami wierzchołków) dwudziestościanu i tablicę indeksów wierzchołków potrzebną do rysowania jego ścian i krawędzi jest podana na listingu 7.7.

[7]Okazji do popełniania błędów i tak nie zabraknie.

Listing 7.7: Procedura ConstructIcosahedronVAO

```c
GLuint icos_vao, icos_vbo[3];

void ConstructIcosahedronVAO ( void )
{
#define A 0.52573115
#define B 0.85065085
  static const GLfloat vertpos[12][3] =
    {{ -A,0.0, -B},{  A,0.0, -B},{0.0, -B, -A},{ -B, -A,0.0},
     { -B,  A,0.0},{0.0,  B, -A},{  A,0.0,  B},{ -A,0.0,  B},
     {0.0, -B,  A},{  B, -A,0.0},{  B,  A,0.0},{0.0,  B,  A}};
  static const GLubyte vertcol[12][3] =
    {{255,0,0},{255,127,0},{255,255,0},{127,255,0},{0,255,0},{0,255,127},
     {0,255,255},{0,127,255},{0,0,255},{127,0,255},{255,0,255},{255,0,127}};
  static const GLubyte vertind[62] =
      0, 1, 2, 0, 3, 4, 0, 5, 1, 9, 2, 8, 3,  /* łamana, od 0 */
       7, 4, 11, 5, 10, 9, 6, 8, 7, 6, 11, 7,
       1, 10, 6,                              /* łamana, od 25 */
       2, 3, 4, 5, 8, 9, 10, 11,              /* 4 odcinki, od 28 */
       0, 1, 2, 3, 4, 5, 1,                   /* wachlarz, od 36 */
       6, 7, 8, 9, 10, 11, 7,                 /* wachlarz, od 43  */
       1, 9, 2, 8, 3, 7, 4, 11, 5, 10, 1, 9;  /* taśma, od 50 */

  glGenVertexArrays ( 1, &icos_vao );
  glBindVertexArray ( icos_vao );
  glGenBuffers ( 3, icos_vbo );
  glBindBuffer ( GL_ARRAY_BUFFER, icos_vbo[0] );
  glBufferData ( GL_ARRAY_BUFFER,
                 12*3*sizeof(GLfloat), vertpos, GL_STATIC_DRAW );
  glEnableVertexAttribArray ( 0 );
  glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
                          3*sizeof(GLfloat), (GLvoid*)0 );
  glBindBuffer ( GL_ARRAY_BUFFER, icos_vbo[1] );
  glBufferData ( GL_ARRAY_BUFFER,
                 12*3*sizeof(GLubyte), vertcol, GL_STATIC_DRAW );
  glEnableVertexAttribArray ( 1 );
  glVertexAttribPointer ( 1, 3, GL_UNSIGNED_BYTE, GL_TRUE,
                          3*sizeof(GLubyte), (GLvoid*)0 );
  glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
  glBufferData ( GL_ELEMENT_ARRAY_BUFFER,
                 62*sizeof(GLubyte), vertind, GL_STATIC_DRAW );
  ExitIfGLError ( "ConstructIcosahedronVAO" );
} /*ConstructIcosahedronVAO*/
```

Liczby

$$A = \sqrt{\frac{5-\sqrt{5}}{10}} \quad \text{i} \quad B = \sqrt{\frac{5+\sqrt{5}}{10}}$$

podane w liniach 5 i 6 spełniają następujące równania: $A^2 + B^2 = 1$ i $\frac{B}{A} = \frac{A}{B-A}$; pierwsze z nich oznacza, że wierzchołki podane w tablicy `vertpos` leżą na sferze jednostkowej. Liczby B i A spełniające drugie równanie pozostają w złotej proporcji — punkty w tablicy `vertpos`, które mają takie współrzędne, są wierzchołkami dwudziestościanu foremnego.

Procedura `glGenVertexArrays` w linii 23 tworzy nowy, początkowo pusty VAO. Jak zwykle w procedurach tworzących obiekty OpenGL-a (szadery, programy, bufory, tekstury itp.), procedura może utworzyć wiele VAO na raz (tyle, ile określa pierwszy parametr). Ich identyfikatory są wpisywane do tablicy, która jest drugim parametrem (tu zmienną `icos_vao` potraktowaliśmy jak tablicę jednoelementową).

Nowo utworzony VAO został przywiązany, tj. uczyniony bieżącym w kontekście OpenGL-a, przez wywołanie procedury `glBindVertexArray` w linii 24. W linii 25 tworzymy trzy bufory na dane opisujące dwudziestościan; ich identyfikatory są wpisywane do tablicy `icos_vbo`; bufory te wypełniamy kolejno.

W liniach 26 i 32 pierwszy i drugi bufor są kolejno przywiązywane do celu `GL_ARRAY_BUFFER`. W liniach 27–28 oraz 33–34 do bufora przywiązanego do tego celu przesyłamy dane z zadeklarowanych w liniach 7–10 oraz 11–13 tablic. Ostatni parametr procedury `glBufferData` w obu tych przypadkach deklaruje, że zawartość tych buforów nie zmieni się przez cały czas działania aplikacji[8].

Procedura `glEnableVertexAttribArray` zapisuje w bieżącym VAO informację o tym, że atrybut o numerze miejsca podanym jako parametr znajduje się w tablicy, skąd ma być pobierany dla kolejnych wierzchołków. Numery miejsc atrybutów podane w liniach 29 i 35 to są te same numery miejsc, które były zapisane w kwalifikatorze `layout` w liniach 3 i 4 na listingu 7.1. Alternatywne rozwiązanie to umieszczenie wartości atrybutu (jednakowej dla wszystkich wierzchołków) w tzw. zmiennej statycznej. Ma to sens, jeśli wszystkie wierzchołki mają na przykład ten sam kolor i chcemy go podać tylko raz. Wartość atrybutu przypisuje się zmiennej statycznej, wywołując jedną z procedur z rodziny

[8]Parametr ten daje *wskazówkę* OpenGL-owi, który może wybrać sposób utworzenia bufora optymalny ze względu na czas dostępu procedur, które będą najczęściej wykonywać działania na tym buforze; konkretna implementacja może też tę wskazówkę zignorować.

glVertexAttrib*[9]. Pobieranie dowolnego atrybutu z tablicy „włączone” przez glEnableVertexAttribArray można „wyłączyć”, wywołując procedurę glDisableVertexAttribArray z odpowiednim parametrem.

Wreszcie, procedura glVertexAttribPointer przekazuje do VAO informację na temat miejsc w tablicy umieszczonej w buforze (przywiązanym w danym momencie do celu GL_ARRAY_BUFFER), w których znajdują się wartości atrybutu dla kolejnych wierzchołków. Atrybuty wprowadzone za pomocą tej procedury na wejściu szadera wierzchołków mają współrzędne zmiennopozycyjne reprezentowane w pojedynczej precyzji[10].

Pierwszy parametr procedury glVertexAttribPointer jest numerem atrybutu. Drugi parametr określa liczbę podanych współrzędnych, a trzeci i czwarty opisują sposób ich reprezentowania. Przypomnijmy, że szader wierzchołków (na listingu 7.1) otrzymuje oba atrybuty typu vec4; tymczasem nasza procedura, zarówno dla położenia wierzchołka, jak i koloru, przesyła do bufora tylko 3 początkowe współrzędne. Współrzędne niepodane otrzymają (w etapie pobierania wierzchołków w potoku przetwarzania grafiki) wartości domyślne — druga i trzecia współrzędna, w razie niepodania, byłyby równe 0, a czwarta — 1. W ten sposób ze współrzędnych kartezjańskich punktu na płaszczyźnie lub w przestrzeni trójwymiarowej etap pobierania wierzchołków skonstruuje wektor współrzędnych jednorodnych, wymagany przez dalsze etapy potoku.

Trzeci parametr procedury glVertexAttribPointer w naszej aplikacji ma wartości GL_FLOAT i GL_UNSIGNED_BYTE, oznaczające odpowiednio liczby typu GLfloat i GLubyte. Czwarty parametr jest boolowski; jego wartość GL_FALSE oznacza, że podane wartości współrzędnych atrybutu należy przyjąć bezpośrednio. Podanie GL_TRUE (dozwolone tylko dla typów całkowitych) spowoduje normalizację, czyli podzielenie przez ustaloną liczbę; dla typu GLubyte dzielnik jest równy 255. Normalizacja wytwarza liczby rzeczywiste z przedziału $[0, 1]$ (dla typów liczbowych bez znaku) albo $[-1, 1]$ (dla typów ze znakiem).

Do bufora koloru, w linii 33–34, wpisujemy tylko 3 współrzędne koloru, opisujące składowe RGB. Czwarta współrzędna, tzw. kanał alfa (A), otrzyma wartość 1,

[9]Czyli np. glVertexAttrib3fv albo glVertexAttrib4ub — zobacz opis przyrostków nazw procedur OpenGL-a w rozdziale 2; pierwszy parametr każdej procedury z tej rodziny jest numerem atrybutu, a pozostałe parametry, których typ jest określony przez przyrostek, podają współrzędne wektora atrybutu.

[10]Do wprowadzania atrybutów opisanych przez liczby całkowite służy procedura glVertexAttribIPointer, a atrybuty w podwójnej precyzji (która ma być utrzymana w potoku przetwarzania grafiki) wprowadza się, wywołując procedurę glVertexAttribLPointer.

której domyślna interpretacja jest taka, że „farba” o podanym kolorze jest całkowicie nieprzezroczysta.

Piąty parametr procedury `glVertexAttribPointer` określa <u>krok</u> (*stride*) w tablicy; jest to odległość w bajtach początków danych dla kolejnych wierzchołków. Można podać wartość `0`, jeśli dane są w buforze upakowane bez przerw (tak jak w naszej aplikacji), albo faktyczną odległość (tak jak w naszej aplikacji).

Ostatni parametr, będący z powodów historycznych wskaźnikiem[11], jest w istocie liczbą całkowitą — odległością w bajtach początku danych dla pierwszego wierzchołka od początku bufora. W programie w C możemy zdefiniować dowolny typ strukturalny z polami struktury opisującymi kilka (lub nawet wszystkie) atrybuty wierzchołka. Wtedy do VAO wpiszemy informacje o różnych atrybutach znajdujących się w jednym buforze; dla poszczególnych atrybutów, w kolejnych wywołaniach procedury `glVertexAttribPointer`, podamy odległości pól z tymi atrybutami od początku struktury.

W liniach 38–40 przywiązujemy trzeci z utworzonych przez `glGenBuffers` w linii 25 buforów do celu `GL_ELEMENT_ARRAY_BUFFER` i przesyłamy do tego bufora tablicę indeksów. Indeksy muszą być liczbami całkowitymi bez znaku: 8-bitowymi (`GLubyte`), 16-bitowymi (`GLushort`) albo 32-bitowymi (`GLuint`). Ponieważ dwudziestościan ma 12 wierzchołków, ich indeksy są małymi liczbami, stąd używamy dla nich typu `GLubyte`. Kolejne fragmenty ciągu liczb, który wpisujemy do bufora, opisują dwie łamane i cztery oddzielne odcinki, dające w sumie wszystkie 30 krawędzi dwudziestościanu, oraz pokazane na rysunku 7.1 dwa wachlarze i jedną taśmę trójkątową, które zawierają wszystkie 20 ścian (zobacz też komentarze w liniach 15–21 na listingu 7.7).

Rysowanie

Rysowanie wykonuje procedura `DisplayFunc` (pusta w linii 13 na listingu 3.1), która wywołuje `DrawIcosahedron`. Procedury te są przedstawione na listingach 7.8 i 7.9.

Procedura `glUseProgram` uaktywnia program szaderów skompilowany i złączony podczas inicjalizacji, tj. wmontowuje jego szadery w potok przetwarzania grafiki. Procedura `glBindVertexArray` w linii 4 przywiązuje VAO z modelem dwudziestościanu, również przygotowany na początku działania aplikacji.

[11] W starym OpenGL-u parametr ten był adresem w pamięci CPU.

Dalsze działania zależą od wartości parametru opt, który wybiera rodzaj rysunku. Jeśli ma on wartość 0, to mają być narysowane wierzchołki, a dokładniej kropki w miejscach obrazów wierzchołków. Procedura glPointSize określa wielkość kwadratowej kropki (jej szerokość i wysokość w pikselach)[12]. Następnie jest wywołana procedura glDrawArrays; jej pierwszy parametr określa, że chcemy narysować kropki. Drugi parametr jest indeksem (w buforach zarejestrowanych w VAO) pierwszego wierzchołka do narysowania, a parametr trzeci określa liczbę tych wierzchołków. W tym przykładzie rysujemy wszystkie 12 wierzchołków.

Listing 7.8: Procedura DrawIcosahedron

C

```c
1: void DrawIcosahedron ( int opt )
2: {
3:   glUseProgram ( program_id );
4:   glBindVertexArray ( icos_vao );
5:   switch ( opt ) {
6: case 0:    /* wierzchołki */
7:     glPointSize ( 5.0 );
8:     glDrawArrays ( GL_POINTS, 0, 12 );
9:     break;
10: case 1:    /* krawędzie */
11:     glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
12:     glDrawElements ( GL_LINE_STRIP, 25,
13:                      GL_UNSIGNED_BYTE, (GLvoid*)0 );
14:     glDrawElements ( GL_LINE_STRIP, 3,
15:                      GL_UNSIGNED_BYTE, (GLvoid*)(25*sizeof(GLubyte)) );
16:     glDrawElements ( GL_LINES, 8,
17:                      GL_UNSIGNED_BYTE, (GLvoid*)(28*sizeof(GLubyte)) );
18:     break;
19: default:   /* ściany */
20:     glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
21:     glDrawElements ( GL_TRIANGLE_FAN, 7,
22:                      GL_UNSIGNED_BYTE, (GLvoid*)(36*sizeof(GLubyte)) );
23:     glDrawElements ( GL_TRIANGLE_FAN, 7,
24:                      GL_UNSIGNED_BYTE, (GLvoid*)(43*sizeof(GLubyte)) );
25:     glDrawElements ( GL_TRIANGLE_STRIP, 12,
26:                      GL_UNSIGNED_BYTE, (GLvoid*)(50*sizeof(GLubyte)) );
27:     break;
28:   }
29:   ExitIfGLError ( "DrawIcosahedron" );
30: } /*DrawIcosahedron*/
```

[12] Szader fragmentów może nadać kropce dowolny kształt.

Jeśli (opt == 1), to procedura ma narysować krawędzie. Indeksy końców tych krawędzi znajdują się w tablicy umieszczonej w buforze, którego identyfikator jest zapamiętany w zmiennej icos_vbo[2]; aby użyć tej informacji, należy ten bufor (IBO) przywiązać do celu GL_ELEMENT_ARRAY_BUFFER, co jest robione w linii 11. Rysowanie odbywa się przez wywołania procedury glDrawElements; pierwsze dwa wywołania, z parametrem GL_LINE_STRIP, powodują rysowanie łamanych (odpowiednio z 25 i 3 wierzchołkami, czyli złożonych z 24 i 2 odcinków). Trzeci parametr określa typ zmiennych użyty do reprezentowania indeksów w tablicy.

Parametr ten musi być równy GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT albo GL_UNSIGNED_INT. Parametr czwarty, zadeklarowany w nagłówku procedury jako wskaźnik z tych samych historycznych powodów, które wcześniej były podane dla procedury glVertexAttribPointer, jest przesunięciem (w bajtach) od początku tablicy do miejsca, z którego etap pobierania wierzchołków ma wziąć pierwszy indeks.

Procedura glDrawElements wywołana w liniach 16–17 z parametrem GL_LINES rysuje 4 oddzielne odcinki. Liczba tych odcinków jest wartością drugiego parametru podzieloną przez liczbę końców jednego odcinka.

Jeśli parametr opt ma wartość inną niż 0 lub 1, to mają być narysowane obrazy ścian trójkątnych, wypełnione odpowiednimi kolorami. W tym celu, po przywiązaniu bufora z indeksami wierzchołków w linii 20, w kolejnych wywołaniach procedury glDrawElements są rysowane dwa wachlarze trójkątów (za to odpowiada parametr GL_TRIANGLE_FAN) oraz taśma trójkątowa (GL_TRIANGLE_STRIP). W każdym przypadku drugi parametr określa liczbę wierzchołków rysowanego prymitywu; w przypadku wachlarza lub taśmy liczba trójkątów jest o 2 mniejsza od wartości tego parametru.

Procedura DisplayFunc przedstawiona na listingu 7.9 jest wywoływana za każdym razem, gdy należy wykonać obraz w oknie aplikacji. Jej zadaniem jest przygotowanie tła, ustawienie opcji rysowania (w szczególności włączenie algorytmu widoczności), wywołanie procedur rysujących obiekty (w naszym przypadku jednej procedury) i pociągnięcie gotowego obrazu werniksem.

Procedura glClearColor ustawia kolor tła, na którym będzie wykonywany rysunek. Ponieważ na rysunku wydrukowanym na papierze jasne kropki na czarnym tle są słabo widoczne[13], parametry podane w linii 5 ustawiają białe tło.

[13] a poza tym szkoda mi tonera

Listing 7.9: Procedura DisplayFunc

C

```c
1: int opcja = 2;
2:
3: void DisplayFunc ( void )
4: {
5:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
6:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
7:   glEnable ( GL_DEPTH_TEST );
8:   DrawIcosahedron ( opcja );
9:   glFlush ();
10:  glutSwapBuffers ();
11: } /*DisplayFunc*/
```

Wszystkie piksele w buforze obrazu otrzymują kolor tła[14] przez wywołanie procedury glClear w linii 6. Jej parametr określa, że jednocześnie z buforem obrazu ma być skasowany bufor głębokości używany przez algorytm widoczności. Procedura glEnable w linii 7 włącza algorytm widoczności, dzięki czemu obraz sceny składającej się z nieprzezroczystych obiektów da się oglądać.

Po narysowaniu wszystkich obiektów (czyli jednego), należy wywołać procedurę glFlush. Informuje ona OpenGL-a, że na tym rysunku wszelkie obiekty już zostały przekazane do narysowania i nie należy czekać na dalsze. Powrót z procedury glFlush może nastąpić przed zakończeniem rysowania (pamiętajmy, że CPU i GPU działają równolegle), ale GPU dokończy rysowanie najszybciej, jak potrafi.

Istnieją sytuacje, w których należy poczekać na dokończenie rysowania, na przykład wtedy, gdy chcemy odczytać obraz z bufora obrazu i zapisać go w pliku. Wywołanie procedury glFlush wtedy nie wystarczy; zamiast niej trzeba wywołać procedurę glFinish, z której powrót nastąpi po zakończeniu rysowania.

Ostatnią instrukcją procedury DisplayFunc jest wywołanie procedury glutSwapBuffers. Powoduje ona, że po zakończeniu rysowania (nie wcześniej!) bufory obrazu zostaną zamienione. Na początku działania aplikacji zadeklarowane zostało podwójne buforowanie obrazu (listing 3.1, linia 35, parametr z maską bitową GLUT_DOUBLE): rysowanie odbywa się w buforze niewidocznym, a w tym czasie w oknie jest widoczna zawartość drugiego bufora. Po zakończeniu rysowania bufory są zamieniane, dzięki czemu w oknie nie bywa widoczny obraz

[14] To jest chyba jedyna operacja rysowania, jaką w nowym OpenGL-u da się wykonać bez programu szaderów.

niedokończony. W animacji bez podwójnego buforowania niedokończone obrazy objawiają się jako silne i nieprzyjemne dla użytkownika migotanie obrazu.

Interakcja

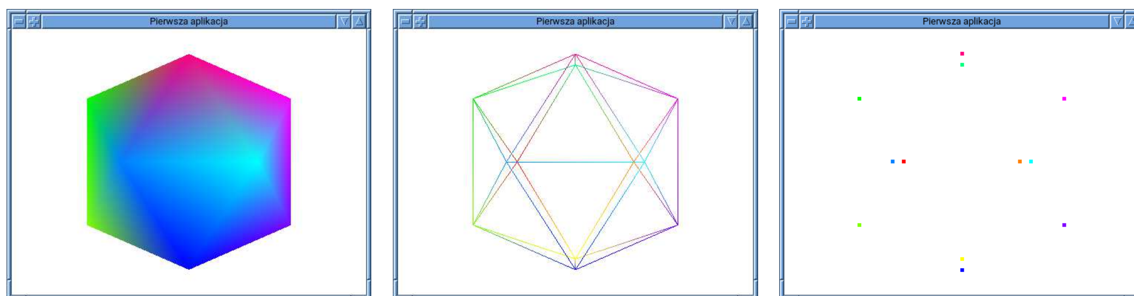
W pierwszej aplikacji pozwolimy użytkownikowi tylko na zatrzymanie programu i na przełączanie trzech rodzajów obrazów; wszystkie te polecenia można wydawać za pomocą klawiatury. Listing 7.10 przedstawia procedurę `KeyboardFunc` (na listingu 3.1 ona jest pusta), którą FreeGLUT będzie wywoływać w reakcji na naciskanie klawiszy.

Listing 7.10: Procedura `KeyboardFunc`

C

```c
1: void KeyboardFunc ( unsigned char key, int x, int y )
2: {
3:   switch ( key ) {
4: case 0x1B:            /* klawisz Esc - zatrzymanie programu */
5:     Cleanup ();
6:     glutLeaveMainLoop ();
7:     break;
8: case 'W':  case 'w':  /* przełączamy na wierzchołki */
9:     opcja = 0;
10:    glutPostWindowRedisplay ( WindowHandle );
11:    break;
12: case 'K':  case 'k':  /* przełączamy na krawędzie */
13:    opcja = 1;
14:    glutPostWindowRedisplay ( WindowHandle );
15:    break;
16: case 'S':  case 's':  /* przełączamy na ściany */
17:    opcja = 2;
18:    glutPostWindowRedisplay ( WindowHandle );
19:    break;
20: default:              /* ignorujemy wszystkie inne klawisze */
21:    break;
22:   }
23: } /*KeyboardFunc*/
```

W odpowiedzi na naciśnięcie klawisza `Esc` procedura wywołuje procedurę sprzątania, a po niej procedurę `glutLeaveMainLoop`, co spowoduje zakończenie działania procedury `glutMainLoop` i powrót sterowania do procedury `main`. Po napisaniu litery `W`, `w`, `K`, `k`, `S` lub `s` zmiennej `opcja` jest przypisywana wartość określająca rodzaj obrazu. Wywołanie procedury `glutPostWindowRedisplay` spowoduje wywołanie procedury `DisplayFunc` i w oknie ukaże się nowy obraz.



Rysunek 7.3: Okno pierwszej aplikacji z obrazami dwudziestościanu

Sprzątanie

Aplikacja, którą napisaliśmy wykorzystując szkielet z listingu 3.1, przed wyjściem z pętli komunikatów FreeGLUTa wywoła jeszcze procedurę `Cleanup`. W zasadzie można by nie zawracać sobie głowy jej pisaniem; system operacyjny w komputerze zawsze dokładnie sprząta i szoruje podłogę po procesie, który zakończył działanie. Ale dobre wychowanie wymaga, aby samemu sprzątać po sobie, co *nie jest* tylko pustym gestem. Bardziej skomplikowane aplikacje, takie na miarę naszych ambicji, będą tworzyć wiele obiektów zajmujących pamięć CPU i GPU. Wiele z nich będzie likwidowanych w trakcie działania aplikacji po to, aby zrobić miejsce dla następnych obiektów. Dlatego warto już od małego[15] przyzwyczajać się do takiego programowania, aby *wszystkie* zasoby zajmowane przez program były zwalniane natychmiast, gdy przestają być potrzebne; jeśli nie wcześniej, to w chwili zatrzymania programu.

Listing 7.11: Procedura `Cleanup`

C

```c
1: void Cleanup ( void )
2: {
3:   int i;
4:
5:   glUseProgram ( 0 );
6:   for ( i = 0; i < 2; i++ )
7:     glDeleteShader ( shader_id[i] );
8:   glDeleteProgram ( program_id );
9:   glDeleteBuffers ( 1, &trbuf );
10:  glDeleteVertexArrays ( 1, &icos_vao );
11:  glDeleteBuffers ( 3, icos_vbo );
12:  ExitIfGLError ( "Cleanup" );
13:  glutDestroyWindow ( WindowHandle );
14: } /*Cleanup*/
```

[15]projektu

Procedura sprzątająca pierwszej aplikacji jest pokazana na listingu 7.11. Wywołanie procedury `glUseProgram` z parametrem `0` usuwa szadery z potoku przetwarzania grafiki (liczba `0` pełni obowiązki identyfikatora programu pustego). W pętli w liniach 6–7 szadery są kasowane, a w linii 8 kasowany jest program szaderów. Następnie zwalniany jest UBO, tj. bufor, w którym przechowywane były zmienne jednolite (macierze przekształceń) oraz VAO i bufory z tablicami wierzchołków i indeksów do wierzchołków. Dzieło sprzątania wieńczy likwidacja okna (i kontekstu OpenGL-a) w wykonaniu procedury `glutDestroyWindow`.

Sprzątanie kontekstu OpenGL-a, tj. likwidacja szaderów i buforów ma sens *przed* usunięciem okna, ponieważ procedura `glutDestroyWindow` unicestwia kontekst stworzony dla tego okna (razem z ewentualnym śmietnikiem w nim zawartym[16]). Dlatego ostatnie sprawdzenie, czy wystąpił błąd OpenGL-a następuje, gdy kontekst jeszcze istnieje. Podobnie, procedura `glutLeaveMainLoop` powoduje powrót z procedury `glutMainLoop` poprzedzony likwidacją wszystkich okien (razem z ich kontekstami), zatem po zakończeniu jej działania nie ma już czego sprzątać. Aplikacja *może* wznowić działanie, ale musi w tym celu zacząć wszystko od nowa, tj. od ponownego wywołania procedury `glutInit`.

Uwagi dodatkowe

Pierwszy parametr procedur `glDrawArrays` i `glDrawElements` określa tryb pracy potoku przetwarzania grafiki. W pierwszej aplikacji wykorzystane są tryby rysowania punktów (`GL_POINTS`), osobnych odcinków (`GL_LINES`), łamanej (`GL_LINE_STRIP`), taśmy trójkątowej (`GL_TRIANGLE_STRIP`) i wachlarza trójkątów (`GL_TRIANGLE_FAN`). Pozostałe dopuszczalne wartości tego parametru wybierają następujące tryby:

`GL_LINE_LOOP` — rysowanie łamanej zamkniętej; w dodatku do odcinków łączących kolejne wierzchołki jest rysowany odcinek między wierzchołkiem ostatnim i pierwszym.

`GL_TRIANGLES` — rysowanie osobnych trójkątów; ciąg wierzchołków jest dzielony na trójki, każda z nich określa jeden trójkąt.

`GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`, `GL_PATCHES` — tryby stosowane z użyciem szaderów rozdrabniania i geometrii, będzie o nich mowa dalej.

[16] Proszę jednak nie traktować tego jako wymówki od sprzątania.

Dla obiektów z zamkniętą objętością, czyli w OpenGL-u zbudowanych z trójkątów powierzchni brył, istotne znaczenie ma orientacja brzegu. Jedna strona trójkąta może być widziana przez obserwatora *znajdującego się na zewnątrz* bryły, a druga strona jest „wewnętrzna". W szczególności trójkąty „odwrócone tyłem" do obserwatora można podczas rysowania pominąć, ponieważ są one zasłonięte przez inne trójkąty, „odwrócone przodem" do obserwatora. Dla skomplikowanych scen daje to znaczną oszczędność czasu rysowania.

Trójkąt, którego obraz nie jest odcinkiem, może być zorientowany względem obserwatora na dwa sposoby: przechodząc przez obrazy jego wierzchołków zgodnie z podaną kolejnością obchodzimy obraz jego środka ciężkości w kierunku zgodnym z ruchem wskazówek zegara (*clockwise*) albo przeciwnym (*counterclockwise*).

Warto zadbać o to, aby wszystkie trójkąty powierzchni bryły miały zgodną orientację. Niech $\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k$ oznacza wierzchołki trójkąta (w $\mathbb{R}^3$), podane w tej kolejności, i niech $\boldsymbol{v}_{ij} = \mathbf{p}_j - \mathbf{p}_i$, $\boldsymbol{v}_{ik} = \mathbf{p}_k - \mathbf{p}_i$ oraz $\boldsymbol{n}_{ijk} = \boldsymbol{v}_{ij} \wedge \boldsymbol{v}_{ik}$. Wektor $\boldsymbol{n}_{ijk}$ jest prostopadły do płaszczyzny trójkąta. Orientacja trójkątów jest zgodna, jeśli dla wszystkich trójkątów tak obliczone wektory są zwrócone na zewnątrz bryły, albo dla wszystkich trójkątów są zwrócone do wewnątrz.

Aby pominąć ściany odwrócone tyłem, należy poprzedzić rysowanie ścian bryły wywołaniem procedur

```
glEnable ( GL_CULL_FACE );
glCullFace ( GL_FRONT );
glFrontFace ( GL_CW );
```

przy czym procedura `glCullFace` może mieć też parametr `GL_BACK`, jeśli chcemy rysować tylko ściany „odwrócone tyłem", bo na przykład obserwator znalazł się wewnątrz bryły. Parametr `GL_CW` procedury `glFrontFace` określa, że przodem są odwrócone ściany, których obrazy są zorientowane zgodnie z ruchem wskazówek zegara, a `GL_CCW` deklaruje orientację przeciwną. Pamiętajmy, aby rysując następny obiekt, jeśli jego brzeg nie jest odpowiednio zorientowany, wyłączyć ten mechanizm za pomocą instrukcji `glDisable ( GL_CULL_FACE );`.

OpenGL zapewnia, że wszystkie trójkąty w wachlarzu lub taśmie są zgodnie zorientowane, wystarczy zatem zadbać tylko o poprawną orientację ich pierwszego trójkąta — reprezentacja dwudziestościanu w pierwszej aplikacji ten warunek spełnia.

Ćwiczenia

1. Dopisz do aplikacji (w procedurze `DrawIcosahedron`) instrukcje pomijania ścian o jednej, a potem o drugiej orientacji, skompiluj i obejrzyj skutki.

2. Napisz procedury, które tworzą reprezentacje czworościanu foremnego, sześcianu i ośmiościanu foremnego i procedury rysujące obrazy tych brył i dołącz je do aplikacji. Kolory wierzchołków wybierz dowolnie. Rozbuduj procedurę `KeyboardFunc`, aby móc przełączać wyświetlane bryły.

   Krawędzie czworościanu można reprezentować jako łamaną zamkniętą i dwa osobne odcinki, a z jego ścian można zrobić jedną taśmę trójkątową.

   Z krawędzi sześcianu można zrobić jedną łamaną zamkniętą i cztery osobne odcinki, a ze ścian, po podzieleniu ich na trójkąty, można zrobić dwie taśmy trójkątowe albo dwa wachlarze.

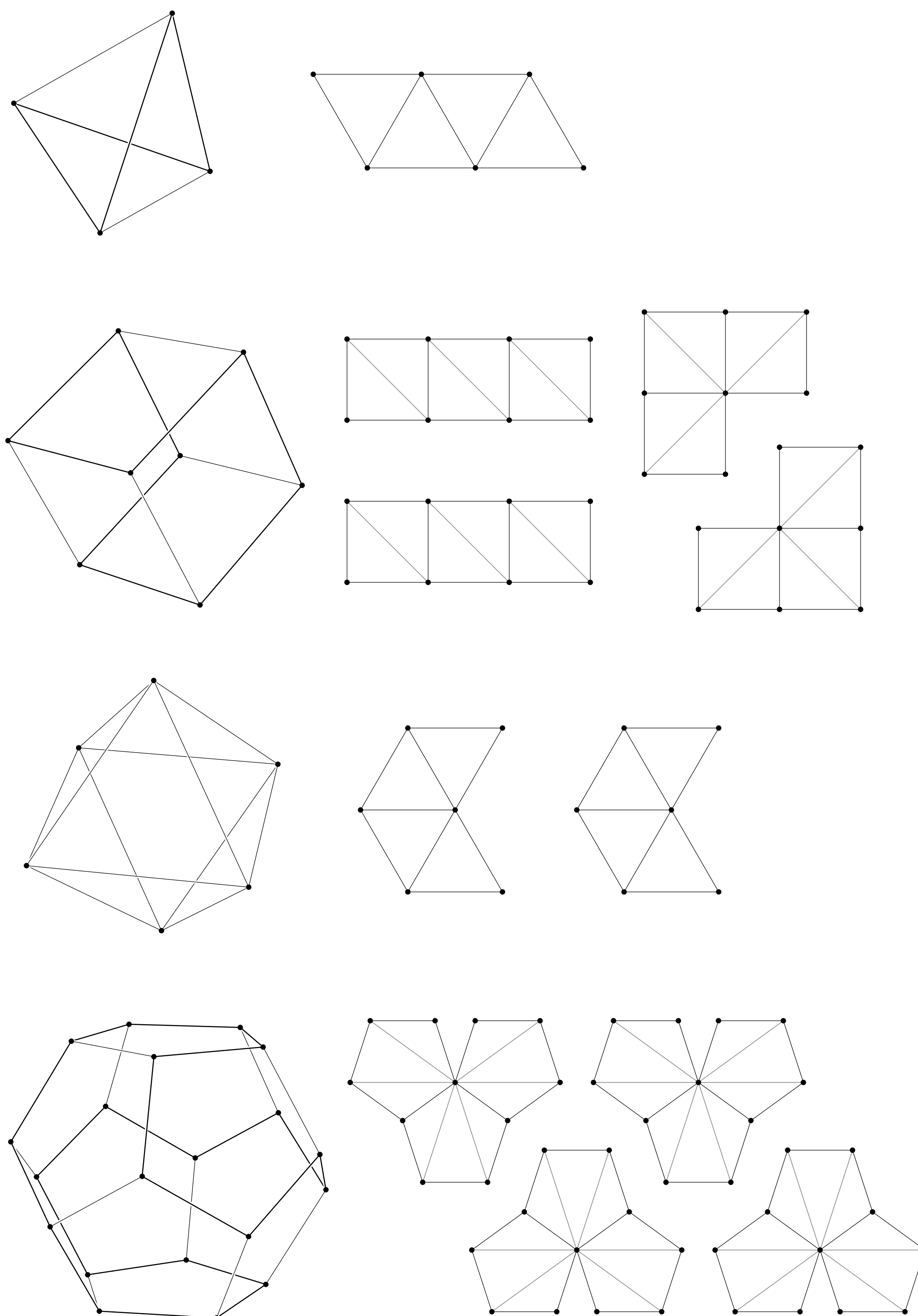
   Krawędzie ośmiościanu można ustawić w takiej kolejności, aby powstała jedna łamana zamknięta; ściany można narysować jako dwa wachlarze trójkątów.

   Wskazówka. Weź najpierw kolorowe kredki i ponumeruj (dowolnie) wierzchołki brył na rysunku 7.4, a potem wierzchołki ścian w taśmach i wachlarzach narysowanych obok.

3. Napisz i uruchom procedury tworzenia reprezentacji i rysowania dwunastościanu foremnego. Z jego krawędzi można utworzyć łamaną zamkniętą przechodzącą przez wszystkie wierzchołki (cykl Hamiltona) i wtedy pozostaje 10 krawędzi do narysowania osobno. Ściany pięciokątne trzeba podzielić na trójkąty (każdą na 3). Można zgrupować po trzy ściany i każdą taką trójkę reprezentować w postaci wachlarza z 9 trójkątów — wtedy narysowanie ścian dwunastościanu wymaga wyświetlenia czterech takich wachlarzy.

   Wskazówka. Dwunastościan jest bryłą dualną do dwudziestościanu. Jego wierzchołki można znaleźć w środkach ciężkości trójkątnych ścian dwudziestościanu — do obliczenia ich współrzędnych można wykorzystać liczby z listingu 7.7.

4. Narysuj szkielet krawędziowy dwudziestościanu (lub innej bryły) i obrazy wierzchołków w postaci kropek. Dla wszystkich krawędzi użyj *jednego* koloru, natomiast wierzchołki narysuj w różnych kolorach. Kolor krawędzi podaj w zmiennej statycznej i przed ich rysowaniem wyłącz tablicę kolorów w VAO za pomocą `glDisableVertexAttribArray` (a przed rysowaniem wierzchołków ją włącz).



Rysunek 7.4: Bryły platońskie: wierzchołki, krawędzie i ściany

5. Utwórz bufor z tablicą z liczbami `0, 1, 9, 8, 3, 7, 11, 5, 10, 9, 2, 8, 7, 4, 11, 10, 1, 6, 7, 4, 5, 10, 1, 2, 8, 3, 4, 11, 5, 1, 9, 2, 3, 7, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11`. Używając go jako tablicy indeksów wierzchołków razem z VAO zawierającym wierzchołki dwudziestościanu, narysuj dwa wachlarze trójkątów z 17 wierzchołkami (od miejsc 0 i 17) i dwa wachlarze trójkątów z 5 wierzchołkami (od miejsc 34 i 39). Otrzymasz obraz bryły zwanej dwunastościanem wielkim.

6. Korzystając z tablicy zawierającej wierzchołki sześcianu, narysuj dwa czworościany wpisane w ten sześcian — ich suma jest bryłą wielościenną zwaną *stella octangula*.

7. W dwunastościan foremny można wpisać sześcian tak, aby jego wierzchołki były niektórymi wierzchołkami dwunastościanu. Narysuj (na jednym obrazie) wszystkie takie sześciany oraz (na innym obrazie) wszystkie czworościany wpisane w te sześciany.

8. Jeśli użytkownik, zmieniając wymiary okna, sprawi, że Jego Wysokość będzie znacznie większa niż szerokość, to nie cały obiekt zmieści się na obrazie. Napraw to, modyfikując procedurę `ReshapeFunc` (listing 7.6): jeśli wysokość jest większa niż szerokość, to kąt dwuścienny między płaszczyznami lewej i prawej ściany bryły widzenia ma być równy $2\alpha = 2\arcsin\frac{1.1}{10}$, a kąt dwuścienny między płaszczyznami górnej i dolnej ściany odpowiednio większy.

8. Aplikacja pierwsza A

Do pierwszej aplikacji dodamy większe możliwości interakcji. Zatem, nie będzie tu nowych konstrukcji OpenGL-a ani GLSL-a, ale wykorzystamy więcej możliwości FreeGLUTa.

Składanie obrotów

Będziemy chcieli obracać obserwatora wokół wyświetlanej bryły. Dowolny obrót w $\mathbb{R}^3$ może być reprezentowany za pomocą wektora jednostkowego $\boldsymbol{v}$ wyznaczającego kierunek osi obrotu i liczby ϕ, która jest miarą kąta obrotu. Trzeba będzie tak reprezentowane obroty składać — mając dane reprezentacje dwóch kolejno wykonanych obrotów w postaci par $(\boldsymbol{v}_1, \phi_1)$ i $(\boldsymbol{v}_2, \phi_2)$, potrzebujemy obliczyć parę $(\boldsymbol{v}, \phi)$, która reprezentuje złożenie tych obrotów[1]. Do tego obliczenia potrzebna będzie procedura `V3CompRotationsf` pokazana na listingu 8.1, którą najwygodniej jest dodać do pliku `utilities.c` (a jej prototyp do `utilities.h`). Macierz obrotu na podstawie wektora i kąta skonstruuje procedura `M4x4RotateVf` podana na listingu 5.1.

Należy znaleźć parę $(\boldsymbol{v}, \phi)$ złożoną z wektora jednostkowego $\boldsymbol{v}$ i liczby ϕ, reprezentującą obrót przestrzeni $\mathbb{R}^3$ będący złożeniem dwóch obrotów reprezentowanych w ten sam sposób; obrotu wokół osi o kierunku $\boldsymbol{v}_1$ o kąt ϕ_1, *po którym* następuje obrót o kąt ϕ_2 wokół osi o kierunku $\boldsymbol{v}_2$. Wektory $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathbb{R}^3$, których współrzędne są podane w tablicach `v1` i `v2`, muszą być jednostkowe. Wzory umożliwiające obliczenie reprezentacji złożenia obrotów są następujące:

$$\begin{aligned}
\cos\frac{\phi}{2} &= \cos\frac{\phi_1}{2}\cos\frac{\phi_2}{2} - \langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle \sin\frac{\phi_1}{2}\sin\frac{\phi_2}{2},\\
\sin\frac{\phi}{2} &= \|\boldsymbol{w}\|_2,\\
\boldsymbol{v} &= \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2},
\end{aligned}$$

przy czym

$$\boldsymbol{w} = \boldsymbol{v}_2 \sin\frac{\phi_2}{2}\cos\frac{\phi_1}{2} + \boldsymbol{v}_1 \sin\frac{\phi_1}{2}\cos\frac{\phi_2}{2} + \boldsymbol{v}_2 \wedge \boldsymbol{v}_1 \sin\frac{\phi_1}{2}\sin\frac{\phi_2}{2}.$$

[1] Obroty można reprezentować za pomocą macierzy ortogonalnych i wtedy złożenie obrotów jest reprezentowane przez iloczyn tych macierzy. Ale iloczyn *wielu* takich macierzy obliczony przy użyciu arytmetyki zmiennopozycyjnej może być (wskutek kumulacji błędów zaokrągleń) złym przybliżeniem macierzy ortogonalnej, a to spowodowałoby zniekształcenia obiektów. Reprezentacja obrotu za pomocą wektora $\boldsymbol{v}$ i kąta ϕ jest pod tym względem lepsza, ponieważ kumulacja błędów powoduje niedokładne wyznaczenie wektora i kąta, które *zawsze* określają *jakiś* obrót — macierz tego obrotu będzie obliczona na ich podstawie całkiem dokładnie.

Listing 8.1: Procedura V3CompRotationsf

C

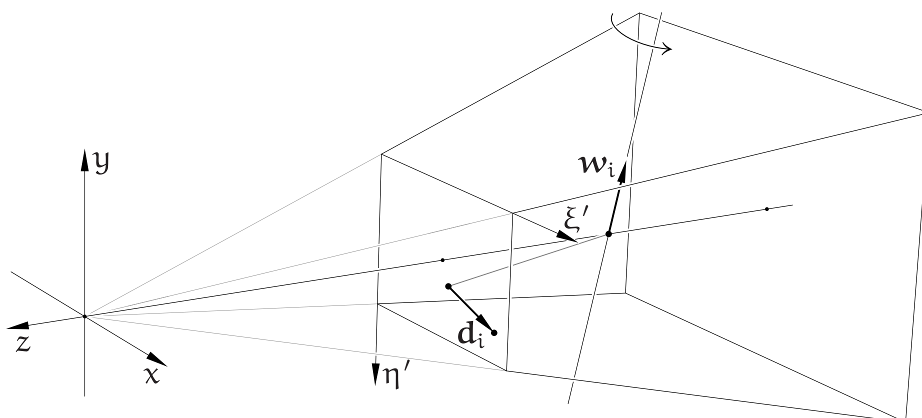
```
1: void V3CompRotationsf ( GLfloat v[3], GLfloat *phi,
2:                         const GLfloat v2[3], GLfloat phi2,
3:                         const GLfloat v1[3], GLfloat phi1 )
4: {
5:   float s2, c2, s1, c1, s, c, s2c1, s1c2, s2s1, v2v1, v2xv1[3];
6:
7:   s2 = sin ( 0.5*phi2 );  c2 = cos ( 0.5*phi2 );
8:   s1 = sin ( 0.5*phi1 );  c1 = cos ( 0.5*phi1 );
9:   s2c1 = s2*c1;  s1c2 = s1*c2;  s2s1 = s2*s1;
10:  v2v1 = V3DotProductf ( v2, v1 );
11:  V3CrossProductf ( v2xv1, v2, v1 );
12:  c = c2*c1 - v2v1*s2s1;
13:  v[0] = v2[0]*s2c1 + v1[0]*s1c2 + v2xv1[0]*s2s1;
14:  v[1] = v2[1]*s2c1 + v1[1]*s1c2 + v2xv1[1]*s2s1;
15:  v[2] = v2[2]*s2c1 + v1[2]*s1c2 + v2xv1[2]*s2s1;
16:  s = sqrt ( V3DotProductf ( v, v ) );
17:  if ( s > 0.0 ) {
18:    v[0] /= s, v[1] /= s, v[2] /= s;
19:    *phi = 2.0*atan2 ( s, c );
20:  }
21:  else
22:    v[0] = 1.0, v[1] = v[2] = *phi = 0.0;
23: } /*V3CompRotationsf*/
```

Obracanie obserwatora wokół obiektu

Chcemy zrealizować następujący scenariusz interakcji: użytkownik naciska przycisk (np. lewy) myszy, a następnie przesuwa mysz, powodując przemieszczanie kursora w oknie. Spowoduje to wysłanie do aplikacji serii komunikatów o przesunięciu kursora — w aplikacji FreeGLUTa każdy taki komunikat jest przekazywany przez wywołanie procedury zarejestrowanej przez `glutMotionFunc` (procedury `MotionFunc` na listingu 3.1). Przemieszczenie kursora od poprzedniej pozycji jest opisane przez pewien wektor równoległy do płaszczyzny obrazu, czyli rzutni. Chcemy, aby aplikacja obróciła obserwatora o pewien kąt wokół osi równoległej do rzutni i prostopadłej do tego wektora, po czym wykonała obraz dla nowego położenia obserwatora. Po zwolnieniu przycisku obracanie ma się zakończyć.

Procedura `MouseFunc` wywołana po naciśnięciu przycisku musi zapamiętać współrzędne ξ'_0, η'_0 położenia kursora w oknie[2] i wprowadzić aplikację w tryb

[2] FreeGLUT używa układu z osią η' skierowaną do dołu, zobacz rozdział 6.



Rysunek 8.1: Przemieszczenie kursora i obrót w układzie obserwatora

obracania obserwatora; procedura `MotionFunc` wywołana z parametrami $x = \xi_i'$ i $y = \eta_i'$, korzystając z zapamiętanych współrzędnych ξ_{i-1}', η_{i-1}' poprzedniego położenia kursora musi wyznaczyć odpowiedni obrót, a potem zapamiętać liczby ξ_i', η_i' na użytek obsługi następnego komunikatu.

Wektor przemieszczenia kursora w oknie (w układzie OpenGL-a) ma współrzędne $(\xi_i' - \xi_{i-1}')$ i $(\eta_{i-1}' - \eta_i')$. W układzie współrzędnych obserwatora (tym, w którym jest opisana bryła widzenia) wektor ten ma kierunek i zwrot wektora

$$\mathbf{d}_i = c \begin{bmatrix} (\xi_i' - \xi_{i-1}')(r-l)/w \\ (\eta_{i-1}' - \eta_i')(t-b)/h \\ 0 \end{bmatrix},$$

gdzie l, r, b, t to parametry określające bryłę widzenia (zobacz rozdział 6), liczby w i h to szerokość i wysokość klatki (okna), zaś c jest pewną liczbą dodatnią; ponieważ tu interesuje nas tylko kierunek i zwrot wektora $\mathbf{d}_i$ (który jest potrzebny do wyznaczenia kierunku osi obrotu), wartość bezwzględna c jest nieistotna[3].

Oś obrotu ma być równoległa do rzutni i prostopadła do wektora $\mathbf{d}_i$ (rysunek 8.1); stąd możemy przyjąć, że ma ona kierunek wektora

$$\boldsymbol{w}_i = \begin{bmatrix} (\eta_i' - \eta_{i-1}')(t-b)/h \\ (\xi_i' - \xi_{i-1}')(r-l)/w \\ 0 \end{bmatrix},$$

który natychmiast podzielimy przez jego długość, aby otrzymać wektor jednostkowy $\boldsymbol{v}_i$.

[3]Nie jest zresztą oczywiste, jak należałoby ją zdefiniować dla rzutowania perspektywicznego.

Obrót układu obserwatora od położenia wyjściowego (w którym układ ten jest tylko przesunięty, ale nie obrócony względem układu świata) do bieżącego będziemy reprezentować za pomocą pary (z_{i-1}, ϕ_{i-1}). Obrót do nowego położenia jest złożeniem tego obrotu z obrotem reprezentowanym przez parę (v_i, ϕ), przy czym wbrew pozorom nie jest złym pomysłem, aby miara kąta tego obrotu była stała w każdym kroku; z mojego doświadczenia wynika, że dobrym wyborem jest przyjęcie $\phi = 0.052359878 \approx 3^\circ$ (jak ktoś zechce, to sobie to zmieni). Należy zatem obliczyć i zapamiętać odpowiednią parę (z_i, ϕ_i), co jest opisane dalej.

Listing 8.2 przedstawia deklaracje zmiennych i procedury, które trzeba dodać do aplikacji opisanej w poprzednim rozdziale (albo zmodyfikować), aby można było obracać obserwatora względem obiektu. W zmiennych `win_width` i `win_height` będą przechowywane wymiary okna. W zmiennych `left`, `right`, `bottom`, `top`, `near` i `far` zostaną zapisane parametry bryły widzenia po każdym ich obliczeniu przez `ReshapeFunc` — danych tych potrzebujemy do wyznaczania wektora osi obrotu. W zmiennych `last_xi`, `last_eta` procedury `MouseFunc` i `MotionFunc` będą zapisywać współrzędne położenia kursora w oknie.

Wartość zmiennej `app_state` określa stan (albo tryb pracy) aplikacji. W tej aplikacji są zdefiniowane dwie wartości tej zmiennej, liczby nazwane (za pomocą makrodefinicji) `STATE_NOTHING` i `STATE_TURNING`[4]. Wartość początkowa to `STATE_NOTHING`. Przemieszczenie kursora w trybie obracania (tj. gdy zmienna `app_state` ma wartość `STATE_TURNING`) powoduje obracanie obserwatora.

W zmiennych `viewer_rvec` i `viewer_rangle` jest przechowywana reprezentacja obrotu obserwatora wokół obiektu w postaci wektora z_i i kąta ϕ_i. Wartości początkowe reprezentują obrót o kąt 0, czyli przekształcenie tożsamościowe[5].

Wreszcie, w zmiennej `viewer_pos0` mamy położenie obserwatora w obróconym układzie współrzędnych, niezmienne w tej aplikacji. Przejście od układu świata do układu obserwatora jest złożeniem obrotu reprezentowanego przez parę (z_i, ϕ_i) i przesunięcia. Jednak lepiej, aby współrzędne położenia obserwatora były przechowywane w zadeklarowanej zmiennej, a nie twardo zakodowane w treści procedury obliczającej macierz tego przejścia.

Zmiana procedury `ReshapeFunc` polega na dodaniu (w liniach 20 i 21) instrukcji zapamiętujących parametry bryły widzenia w dodanych zmiennych globalnych.

[4] Nieważne, jakie to są liczby, ważne, że są różne.

[5] Oś takiego obrotu ma nieokreślony kierunek, zatem liczby obecne początkowo w tablicy `viewer_rvec` mogą być dowolne, byleby suma ich kwadratów była równa 1.

Listing 8.2: Dodatkowe zmienne i procedury w aplikacji pierwszej A

C

```
1: int   win_width, win_height;
2: int   last_xi, last_eta;
3: float left, right, bottom, top, near, far;
4:
5: int   app_state = STATE_NOTHING;
6: float viewer_rvec[3] = {1.0,0.0,0.0};
7: float viewer_rangle = 0.0;
8: const float viewer_pos0[4] = {0.0,0.0,10.0,1.0};
9:
10: void ReshapeFunc ( int width, int height )
11: {
12:   GLfloat m[16];
13:   float   lr;
14:
15:   glViewport ( 0, 0, width, height );      /* klatka jest całym oknem */
16:   lr = 0.5533*(float)width/(float)height;  /* przyjmujemy aspekt równy 1 */
17:   M4x4Frustumf ( m, NULL, -lr, lr, -0.5533, 0.5533, 5.0, 15.0 );
18:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
19:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), m );
20:   win_width = width,  win_height = height;
21:   left = -(right = lr);  bottom = -(top = 0.5533);  near = 5.0;  far = 15.0;
22:   ExitIfGLError ( "ReshapeFunc" );
23: } /*ReshapeFunc*/
24:
25: void InitViewMatrix ( void )
26: {
27:   GLfloat m[16];
28:
29:   M4x4Translatef ( m, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
30:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
31:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), m );
32:   ExitIfGLError ( "InitViewMatrix" );
33: } /*InitViewMatrix*/
34:
35: void RotateViewer ( int delta_xi, int delta_eta )
36: {
37:   float   vi[3], lgt, angi, vk[3], angk;
38:   GLfloat tm[16], rm[16], vm[16];
39:
40:   if ( delta_xi == 0 && delta_eta == 0 )
41:     return;  /* natychmiast uciekamy - nie chcemy dzielić przez zero */
42:   vi[0] = (float)delta_eta*(right-left)/(float)win_height;
```

```
43:   vi[1] = (float)delta_xi*(top-bottom)/(float)win_width;
44:   vi[2] = 0.0;
45:   lgt = sqrt ( V3DotProductf ( vi, vi ) );
46:   vi[0] /= lgt;  vi[1] /= lgt;
47:   angi = -0.052359878;  /* -3 stopnie */
48:   V3CompRotationsf ( vk, &angk, viewer_rvec, viewer_rangle, vi, angi );
49:   memcpy ( viewer_rvec, vk, 3*sizeof(float) );
50:   viewer_rangle = angk;
51:   M4x4Translatef ( tm, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
52:   M4x4RotateVf ( rm, viewer_rvec[0], viewer_rvec[1], viewer_rvec[2],
53:                  -viewer_rangle );
54:   M4x4Multf ( vm, tm, rm );
55:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
56:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
57:   ExitIfGLError ( "RotateViewer" );
58: } /*RotateViewer*/
59:
60: void MouseFunc ( int button, int state, int x, int y )
61: {
62:   switch ( app_state ) {
63: case STATE_NOTHING:
64:     if ( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN ) {
65:       last_xi = x,  last_eta = y;
66:       app_state = STATE_TURNING;
67:     }
68:     break;
69: case STATE_TURNING:
70:     if ( button == GLUT_LEFT_BUTTON && state != GLUT_DOWN )
71:       app_state = STATE_NOTHING;
72:     break;
73: default:
74:     break;
75:   }
76: } /*MouseFunc*/
77:
78: void MotionFunc ( int x, int y )
79: {
80:   switch ( app_state ) {
81: case STATE_TURNING:
82:     if ( x != last_xi || y != last_eta ) {
83:       RotateViewer ( x-last_xi, y-last_eta );
84:       last_xi = x,  last_eta = y;
85:       glutPostWindowRedisplay ( WindowHandle );
86:     }
87:     break;
```

```
88: default:
89:     break;
90:   }
91: } /*MotionFunc*/
```

Procedura `InitViewMatrix` została zmieniona w ten sposób, że początkowe przekształcenie jest konstruowane na podstawie wartości zmiennej `viewer_pos0` (porównaj z listingiem 7.4); ta zmiana jest kosmetyczna.

Nowa jest procedura `RotateViewer`, której parametry `delta_xi` $= \xi_i' - \xi_{i-1}'$, `delta_eta` $= \eta_i' - \eta_{i-1}'$ podają współrzędne wektora przemieszczenia kursora w oknie.

Działanie tej procedury szczegółowo wyjaśnimy, bo jeszcze ważniejsze od tego aby program działał, jest to, aby jego autor[6] rozumiał, co się tam dzieje. Przede wszystkim każda macierz nieosobliwa reprezentuje co najmniej *dwa* przekształcenia, albo też wynik mnożenia wektora przez macierz można interpretować na co najmniej dwa sposoby[7]. Pierwsza interpretacja jest taka, że mnożąc macierz przez wektor współrzędnych (jednorodnych) punktu, otrzymujemy współrzędne *nowego punktu*. Jeśli na przykład przekształcenie jest przesunięciem, to nowy punkt jest przesunięty o ustalony wektor $\mathbf{t}$ względem punktu danego.

W drugiej interpretacji wynik mnożenia macierzy przez wektor jest wektorem współrzędnych tego samego punktu *w nowym układzie współrzędnych*. Dla przesunięcia punktu o wektor $\mathbf{t}$ nowy układ jest przesunięty względem wyjściowego o wektor $-\mathbf{t}$. Podobnie obrót punktu wokół ustalonej osi o kąt ϕ można zinterpretować jako przejście do układu współrzędnych obróconego wokół tej samej osi o kąt $-\phi$. Zawsze układ odniesienia (składający się z początku układu i wersorów osi) w dualnej interpretacji jest poddawany przekształceniu odwrotnemu do przekształcenia punktu w interpretacji pierwotnej[8].

Naciskając przycisk i przesuwając mysz, intuicyjnie zamierzamy „chwycić” pewien

[6] czyli niestety ja

[7] Była o tym mowa w rozdziale 6. Trzecia interpretacja, tu nieistotna, to przejścia między współrzędnymi kartezjańskimi i barycentrycznymi.

[8] To samo dotyczy nieobecnych w tej aplikacji skalowań. Pomnożenie współrzędnych kartezjańskich przez 2.54 można zinterpretować jako jednokładność *powiększającą obiekt* w tej skali, lub obliczenie współrzędnych punktu w centymetrach na podstawie jego współrzędnych podanych w calach — czyli przejście do układu, którego jednostki są 2.54 razy *krótsze*. Aby zmienić centymetry na cale (czyli *powiększyć jednostkę długości*), trzeba współrzędne przez 2.54 *podzielić*.

wyimaginowany punkt $\mathbf{p}$ znajdujący się *przed osią obrotu* (zobacz rysunek 8.1) i spowodować jego przemieszczenie w kierunku ruchu myszy. Postrzegamy to jako polecenie obrócenia oglądanego obiektu; w rzeczywistości (dualnej) jako obserwator „odpychamy się” od „chwyconego” punktu i obracamy *się* (tj. obserwatora z jego układem współrzędnych) w przeciwną stronę względem nieruchomego obiektu. Jeśli chcemy, aby obiekt obrócił się o 3° wokół osi o kierunku wektora $\boldsymbol{v}_i$, to jest to równoznaczne z obróceniem obserwatora o kąt $-3°$ wokół tej osi.

Ponieważ obrót w i-tym kroku jest określony w układzie *obserwatora*, złożenie obrotów: dotychczas wykonanego, reprezentowanego przez wartości zmienych `viewer_rvec` i `viewer_rangle` oraz bieżącego, reprezentowanego w zmiennych `vi` i `angi` (lokalnych w procedurze `RotateViewer`), złożenie obrotów trzeba wykonać tak, aby współrzędne punktu były pomnożone *najpierw* przez macierz bieżącego obrotu, a potem wynik tego mnożenia przez macierz reprezentującą złożenie wszystkich obrotów wykonanych wcześniej. Tę kolejność realizują parametry procedury `V3CompRotationsf` wywołanej w linii 48.

Zatem, po wykonaniu procedury `V3CompRotationsf` mamy reprezentację *nowego obrotu układu* obserwatora względem układu świata, przypisaną (w liniach 49 i 50) do zmiennych `viewer_rvec` i `viewer_rangle`. Ale obrót układu obserwatora o kąt ϕ_i oznacza, że *przejście do układu* współrzędnych obserwatora jest obrotem wokół tej samej osi o kąt $-\phi_i$. Stąd ostatni parametr procedury `M4x4RotateVf` w linii 53 został opatrzony znakiem „−”.

Osie wszystkich rozpatrywanych tu obrotów przechodzą przez początek układu świata. Po obróceniu obserwator musi się jeszcze przesunąć do punktu, który w układzie obróconym ma współrzędne $(0, 0, 10)$; a więc obliczenie współrzędnych punktów w układzie przesuniętym jest przesunięciem o wektor $(0, 0, -10)$. Przesunięcie to jest dane w układzie obróconym. Znów, właściwa kolejność czynników iloczynu macierzy, który obliczamy w linii 54 jest taka, jakby najpierw był wykonany obrót, a potem przesunięcie.

W liniach 55 i 56 przywiązujemy UBO, tj. bufor z macierzami przekształceń do celu `GL_UNIFORM_BUFFER` i przesyłamy do tego bufora współczynniki nowej macierzy przejścia do układu obserwatora.

Interakcję z użytkownikiem zapewniają procedury `MouseFunc` i `MotionFunc`, które trzeba wstawić w miejsce pustych procedur na listingu 3.1. Procedura `MouseFunc` jest wywoływana za każdym razem, gdy któryś przycisk myszy został naciśnięty

lub zwolniony. Jeśli jest to lewy przycisk, to jego naciśnięcie spowoduje przejście aplikacji do trybu obracania obserwatora (zmienna `app_state` otrzymuje wartość `STATE_TURNING`) i zapamiętanie położenia kursora w oknie. Jeśli lewy przycisk został zwolniony, to aplikacja wraca do stanu początkowego.

Wywołanie procedury `MotionFunc` w trybie obracania obserwatora powoduje obliczenie przemieszczenia kursora; jeśli przemieszczenie jest niezerowe (a może być zerowe), to wywoływana jest procedura `RotateViewer`, która oblicza nową macierz przejścia do układu obserwatora i przypisuje współczynniki tej macierzy odpowiedniej zmiennej jednolitej. Nowe położenie kursora jest zapamiętywane i na koniec FreeGLUT zostaje poinformowany o tym, że poprzedni obraz w oknie jest już nieaktualny i trzeba narysować nowy.

Animacja

Niech po naciśnięciu klawisza spacji obiekt sam się obraca, albo niech się przestanie obracać. Będziemy animować macierz przekształcenia modelu, tj. powodować obracanie modelu w układzie świata, niezależnie od ruchu obserwatora względem układu świata. Oś obrotu obiektu będzie tu ustalona; jest to oś y układu świata, będzie też stała prędkość obrotowa, $\pi/4$ na sekundę (czyli będzie jeden pełny obrót na 8 sekund). Napiszemy procedury, które to realizują.

Na listingu 8.3 są przedstawione pomocnicze procedury, których zadaniem jest odczytywanie zegara systemowego i podawanie informacji o czasie. Uwaga: te procedury są zależne od systemu operacyjnego i *nie są przenośne*; chcąc uruchomić tę i dalsze aplikacje do innego środowiska niż system Linux, trzeba odpowiednio zmodyfikować treść tych procedur, zostawiając niezmienione nagłówki i deklaracje zmiennych globalnych `app_time`, `tic_time` i `toc_time`, czyli utworzony przez te procedury interfejs programowania aplikacji, ukrywający zależności systemowe. Dlatego procedury te są umieszczone w osobnym pliku źródłowym — jedynym, który trzeba zmodyfikować podczas przenoszenia aplikacji do innego środowiska.

Po uruchomieniu aplikacja powinna wywołać procedurę `TimerInit`, której zadaniem jest uzyskanie informacji o liczbie tyknięć zegara na sekundę (informacja ta zostaje zapamiętana w zmiennej `ticks_per_sec`) oraz zapamiętanie (w zmiennej `tick0`) bieżącego licznika zegara. Zmienne `app_time`, `tic_time` i `toc_time`, dostępne dla aplikacji, otrzymują wartość 0. Procedura `TimerTic` zapamiętuje w zmiennych `app_time` i `tic_time` i zwraca czas (liczbę sekund), które upłynęły od chwili wywołania procedury `TimerInit`. Procedura `TimerToc`

Listing 8.3: Procedury odczytywania zegara

timer.c — wersja dla systemu Linux

```
1: #include <unistd.h>
2: #include <sys/times.h>
3: #include "myglheader.h"
4:
5: #include "utilities.h"
6:
7: static clock_t tick0, tick1, tick2;
8: static double  ticks_per_sec = 100.0;
9: double         app_time, tic_time, toc_time;
10:
11: void TimerInit ( void )
12: {
13:   struct tms clk;
14:
15:   ticks_per_sec = (double)sysconf ( _SC_CLK_TCK );
16:   tick0 = tick1 = tick2 = times ( &clk );
17:   app_time = toc_time = 0.0;
18: } /*TimerInit*/
19:
20: double TimerTic ( void )
21: {
22:   struct tms clk;
23:
24:   tick1 = tick2 = times ( &clk );
25:   toc_time = 0.0;
26:   return app_time = tic_time = (double)(tick2-tick0)/ticks_per_sec;
27: } /*TimerTic*/
28:
29: double TimerToc ( void )
30: {
31:   struct tms clk;
32:
33:   tick2 = times ( &clk );
34:   app_time = (double)(tick2-tick0)/ticks_per_sec;
35:   return toc_time = (double)(tick2-tick1)/ticks_per_sec;
36: } /*TimerToc*/
```

zapamiętuje w zmiennej `app_time` czas, który upłynął od wywołania procedury `TimerInit`, a potem oblicza, zapamiętuje w zmiennej `toc_time` i zwraca liczbę sekund od wywołania procedury `TimerInit` lub od ostatniego wywołania procedury `TimerTic`.

Zwróćmy uwagę na lokalną odmianę problemu milenijnego. Czas jest wielkością analogową, zatem wyniki jego pomiarów przyjmują wartości ułamkowe i do obliczania fazy ruchu animowanych obiektów trzeba je podawać w reprezentacji zmiennopozycyjnej. Mantysa liczby zmiennopozycyjnej pojedynczej precyzji (typu float) ma 23 bity, co oznacza, że można liczyć na dokładność względną tej reprezentacji nie lepszą niż $2^{-24} \approx 5.9 \cdot 10^{-8}$. W Linuksie domyślna częstotliwość tyknięć, tj. przerwań powodujących zwiększanie systemowego licznika zegarowego[9] to 100Hz (czyli czas może być odczytywany z licznika z dokładnością do 0.01s). To znaczy, że po upłynięciu $0.01 \cdot 2^{24}$s $\approx$ 46h błędy zaokrągleń wytwarzane podczas przypisywania zmiennej typu float wyników dzielenia przyrostów licznika zegara od chwili uruchomienia aplikacji przez częstotliwość tyknięć będą większe niż 0.01s, a w konsekwencji mimo zwiększenia licznika aplikacja może nie wykryć, że nastąpił dający się zmierzyć upływ czasu. Skutkiem tego może być zmniejszenie płynności animacji lub w ogóle złe działanie programu. Dlatego do reprezentowania wyników pomiarów czasu lepiej jest używać zmiennych podwójnej precyzji (typu double). Umożliwi to poprawne działanie aplikacji — gry komputerowej — nawet przez $0.01 \cdot 2^{53}$s; większości graczy tyle czasu powinno wystarczyć.

Na listingu 8.4 mamy następujące zmienne globalne: `model_rot_axis` z ustalonymi współrzędnymi wektora osi obrotu, `model_rot_angle`, której wartość jest miarą kąta obrotu z danej chwili, przełącznik `animate`, któremu przypisuje się wartość `true` w chwili uruchomienia animacji i `false` w chwili jej zatrzymania.

Nazwa procedury `InitModelMatrix` została zmieniona na `SetupModelMatrix`, co jest motywowane tym, że procedura ta nie służy już tylko do inicjalizacji macierzy przekształcenia modelu, ale do konstruowania tej macierzy dla wszelkich kątów obrotu podawanych w trakcie animacji. Parametry procedury (wektor i kąt) reprezentują obrót modelu. Inicjalizacja macierzy przekształcenia modelu jest wykonywana przez procedurę `SetupModelMatrix` z parametrami, których wartości (nadane w deklaracjach w liniach 1 i 2) wyznaczają przekształcenie tożsamościowe.

Do procedury `KeyboardFunc` została dodana reakcja na naciśnięcie klawisza spacji, które uruchamia lub zatrzymuje animację. Niezmienione instrukcje w tej procedurze pozwoliłem sobie wykropkować. Procedura `ToggleAnimation`, wywoływana przez `KeyboardFunc`, zależnie od wartości zmiennej `animate` uruchamia albo zatrzymuje animację. Uruchomienie polega na przypisaniu

[9] Można skonfigurować jądro systemu operacyjnego tak, aby zmienić tę częstotliwość, np. podwyższyć ją do 250Hz, w celu zwiększenia dokładności pomiarów czasu przez programy działające w tym systemie.

Listing 8.4: Procedury animacji

C

```
1: float model_rot_axis[3] = {0.0,1.0,0.0},
2:       model_rot_angle0 = 0.0, model_rot_angle;
3: char  animate = false;
4:
5: void SetupModelMatrix ( float axis[3], float angle )
6: {
7:   GLfloat m[16];
8:
9:   M4x4RotateVf ( m, axis[0], axis[1], axis[2], angle );
10:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
11:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0], 16*sizeof(GLfloat), m );
12:   ExitIfGLError ( "SetupModelMatrix" );
13: } /*SetupModelMatrix*/
14:
15: void IdleFunc ( void )
16: {
17:   model_rot_angle = model_rot_angle0 + 0.78539816 * TimerToc ();
18:   SetupModelMatrix ( model_rot_axis, model_rot_angle );
19:   glutPostWindowRedisplay ( WindowHandle );
20: } /*IdleFunc*/
21:
22: void ToggleAnimation ( void )
23: {
24:   if ( (animate = !animate) ) {  /* zaczynamy obracanie */
25:     TimerTic ();
26:     glutIdleFunc ( IdleFunc );
27:   }
28:   else {                         /* kończymy obracanie */
29:     model_rot_angle0 = model_rot_angle;
30:     glutIdleFunc ( NULL );
31:   }
32: } /*ToggleAnimation*/
33:
34: void KeyboardFunc ( unsigned char key, int x, int y )
35: {
36:   switch ( key ) {
37: .........  /* tu linie 4 - 18 z listingu 7.10 */
38: case ' ':              /* włączamy albo wyłączamy animacje */
39:     ToggleAnimation ();
40:     break;
41: default:               /* ignorujemy wszystkie inne klawisze */
42:     break;
```

```
43:   }
44: } /*KeyboardFunc*/
45:
46: void InitMyObject ( void )
47: {
48:   TimerInit ();
49:   SetupModelMatrix ( model_rot_axis, model_rot_angle0 );
50:   InitViewMatrix ();
51:   ConstructIcosahedronVAO ();
52: } /*InitMyObject*/
```

zmiennej `animate` wartości `true`, wywołaniu procedury `TimerTic` i zarejestrowaniu (w linii 26) procedury `IdleFunc`, która będzie wywoływana co chwila aż do odwołania. Zatrzymanie animacji polega na wyrejestrowaniu procedury `IdleFunc` (przez wywołanie `glutIdleFunc` z parametrem `NULL`) i zapamiętaniu w zmiennej `model_rot_angle` bieżącego kąta obrotu — aby móc gładko wznowić animację. Oczywiście, linia 47 na listingu 3.1 powinna pozostać komentarzem, a jeszcze lepiej ją skasować.

Procedura `IdleFunc` (pusta w linii 18 na listingu 3.1) oblicza w linii 17 bieżący kąt obrotu, dodając do kąta pamiętanego w zmiennej `model_rot_angle0`, odpowiadającego chwili uruchomienia animacji, iloczyn czasu od uruchomienia animacji (podanego przez procedurę `TimerToc`) i prędkości obrotowej (wyrażonej w radianach na sekundę), równej 0.78539816, czyli w przybliżeniu $\pi/4$. Bieżący kąt obrotu jest podawany jako parametr procedury `SetupModelMatrix`, która skonstruuje i prześle do GPU odpowiednią macierz. Następnie FreeGLUT zostaje zawiadomiony o potrzebie wykonania nowego obrazu.

Ćwiczenia

1. Czy potrafisz tak zmienić aplikację, aby po naciśnięciu spacji obiekt zaczynał i przestawał się obracać płynnie, tj. stopniowo zwiększając i zmniejszając swoją prędkość obrotową między zerem i prędkością maksymalną? W okresach „rozpędzania” i „hamowania” można przyjąć stałe przyspieszenia kątowe.

2. Rozszerzając poprzednie ćwiczenie, spróbuj zmienić aplikację tak, aby w reakcji na naciśnięcie dwóch klawiszy, np. `'+'` i `'-'`, prędkość obrotowa płynnie się zwiększała albo zmniejszała, co umożliwiałoby obserwowanie bryły obracającej się w różnym tempie.

3. Rozbuduj aplikację o możliwość powiększania i zmniejszania obrazu w oknie.

W tym celu należy wprowadzić dodatkową zmienną, „długość ogniskową obiektywu”, na podstawie której ma być obliczany kąt α między płaszczyną xy układu obserwatora oraz górną i dolną ścianą bryły widzenia (zobacz rachunki na s. 7.8). Naciśnięcie prawego przycisku myszy ma wprawić aplikację w tryb „najazdów” i „odjazdów” obserwatora; przesunięcie myszy w tym trybie ma spowodować odpowiednią zmianę (w ustalonych granicach) kąta α i wykonanie nowego obrazu w oknie. Zwolnienie przycisku ma spowodować powrót aplikacji do stanu podstawowego.

9. Podstawy GLSL-a

Poszczególne rodzaje szaderów mają istotnie różne role do spełnienia, w związku z czym zestaw dopuszczalnych konstrukcji jest w każdym przypadku inny. Dlatego specyfikacja [3] oficjalnie stwierdza, że GLSL to w istocie *sześć różnych* języków, każdy dla szaderów określonego typu. Ale wiele elementów języki te mają wspólne.

Język GLSL jest dosyć podobny do języka C; wychodząc z założenia, że ten ostatni jest Czytelnikowi znany, przedstawię go przez wskazanie różnic między GLSL i C.

Symbole leksykalne

Tekst źródłowy programu w GLSL (szadera) składa się z jednego lub wielu napisów (łańcuchów ASCII lub ASCIIZ), których tablicę rejestruje się w obiekcie szadera (zobacz rozdział 4) przed kompilacją. W napisach tych wyróżniane są separatory (spacje, znaki końca linii i komentarze), identyfikatory (ciągi liter i cyfr zaczynające się od litery), przy czym pewne identyfikatory są słowami kluczowymi (w tej książce będą podkreślane), literały (czyli liczby, nie ma znanych w C znaków ani napisów) i operatory zbudowane ze znaków specjalnych.

Tak jak w C, w GLSL rozróżnia się wielkie i małe litery w identyfikatorach, zatem identyfikator `Out` jest czymś innym niż słowo kluczowe out. Znak podkreślenia '_' jest też literą, ale nie wolno w identyfikatorze napisać dwóch takich znaków obok siebie. Ponadto identyfikatory zaczynające się od przedrostka `gl_` są zarezerwowane.

Komentarze mają dwie dopuszczalne formy, tak jak w języku C++. Pierwsza forma, znana z C, to `/* dowolny tekst */`, wewnątrz którego nie może wystąpić para znaków `*/`. Druga forma to

```
// dowolny tekst do końca linii.
```

Kompilator GLSL komentarze pomija, ale trzeba je pisać; programy piszemy dla swojej przyjemności, nie kompilatora.

Preprocesor

Kompilator GLSL jest wyposażony w preprocesor podobny do tego znanego z C; dyrektywa preprocesora składa się ze słowa kluczowego preprocesora z doklejonym

z przodu znakiem #, a po nim następuje treść dyrektywy. W linii przed słowem kluczowym dyrektywy i po jej treści mogą być tylko spacje i tabulatory.

`#version` — dyrektywa deklarująca wersję języka, w której szader iest napisany; *musi* wystąpić na początku jego tekstu[1]. Użycie konstrukcji językowej z wersji nowszej niż podana w tej dyrektywie powoduje błąd kompilacji. Numer wersji podaje się jako liczbę trzycyfrową, 100 razy większą niż numer wersji traktowany jak ułamek. Po tym numerze *można* podać słowo `core`, `compatibility` albo `es`, które deklaruje używany profil: odpowiednio profil podstawowy, profil zgodności (ze starym OpenGL-em) lub profil dla systemów wbudowanych (*embedded systems*), czyli po ludzku dla telefonów komórkowych[2].

Zaczynając od wersji 3.3 OpenGL-a numeracja wersji GLSL i OpenGL jest zsynchronizowana. Zatem, pisząc program w OpenGL-u 4.5 powinno się deklarować wersję GLSL 4.5, choć można używać też starszych wersji języka. Przykładowa deklaracja wersji języka może wyglądać tak:

```
#version 450 core
```

Przypominam, że konieczne jest zakończenie tej deklaracji znakiem końca linii (zobacz linię 2 na listingu 4.4).

`#define`, `#undef` — utworzenie i likwidacja makrodefinicji. Te dyrektywy działają tak samo jak w C, w szczególności makra mogą być sparametryzowane. Jeśli makrodefinicja jest długa, to można ją zapisać w większej liczbie linii, stawiając na końcu każdej linii makra oprócz ostatniej znak `'\'`.

`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif` — dyrektywy kompilacji warunkowej, działają jak w C, ale wyrażenia opisujące warunki kompilacji mają ograniczenia wynikające z dostępnych konstrukcji w języku GLSL (zobacz [3]), nieco innych niż w C.

`#pragma` — opcja dla kompilatora, na przykład `optimize(off)` wyłączająca optymalizację, albo `debug(on)` ułatwiająca uruchamianie szadera. Te pragmy nie mogą wystąpić wewnątrz podprogramu.

`#extension` — dyrektywa określająca sposób postępowania w sytuacji, gdy pewne rozszerzenie języka jest albo nie jest dostępne.

---

[1] Jeśli dyrektywa `#version` jest nieobecna, to kompilator uznaje, że jest to wersja 1.10, przeznaczona do współpracy ze starym OpenGL-em.

[2] Program na CPU w telefonie powinien być napisany zgodnie ze standardem OpenGL ES, którym tu się nie zajmujemy.

`#error` — powoduje błąd kompilacji. Zazwyczaj umieszcza się ją w miejscu obłożonym jakąś dyrektywą kompilacji warunkowej, na przykład uzależniającej powodzenie kompilacji od obecności potrzebnego rozszerzenia.

`#line` — dla porządku odnotujmy tę dyrektywę, służącą do zapisywania numerów linii tekstu w jakimś innym języku, na podstawie którego tekst w GLSL-u został wygenerowany automatycznie, i nie przejmujmy się tym zbytnio. Trochę bardziej przejmujemy się *brakiem* dyrektywy `#include`, ale bez przesady.

Podstawowe typy zmiennych

Typy proste to `void` (dla procedur, które nie zwracają wyniku), `bool` (typ boolowski, tj. logiczny), `int` (liczby całkowite bez znaku), `uint` (liczby ze znakiem), `float` (liczby zmiennopozycyjne pojedynczej precyzji) i `double` (liczby zmiennopozycyjne podwójnej precyzji). Zamiast `uint` można napisać `unsigned int`.

Liczby całkowite typu `int` oraz `uint` są 32-bitowe. Literały tych typów mogą być podane w postaci ciągu cyfr dziesiętnych, ósemkowych lub szesnastkowych, zapisywanych tak samo jak w C. Może wystąpić przyrostek `u`, który oznacza, że podany literał jest typu `uint`.

Liczby zmiennopozycyjne są reprezentowane zgodnie ze standardem IEEE-754, ale działania na nich *nie muszą* w pełni realizować tego standardu — chodzi o możliwość wyboru sposobu zaokrąglania (której nie ma), postępowanie w przypadku niedomiaru i o dostępność reprezentacji nie-liczb i nieskończoności.

Typy wektorowe składają się z dwóch, trzech lub czterech składowych prostego typu liczbowego lub boolowskiego. Cztero- lub pięcioznakowe nazwy tych typów składają się z opcjonalnego jednoliterowego przedrostka `b`, `i`, `u` lub `d`, rdzenia `vec` i przyrostka — cyfry 2, 3 lub 4 określającej liczbę składowych. Przedrostek oznacza, że składowe są odpowiednio typu `bool`, `int`, `uint` lub `double`, a brak przedrostka oznacza, że są typu `float`. Nazwy wszystkich tych typów (jest ich 15) są słowami kluczowymi. Proszę zatem samemu rozszyfrować, co oznaczają typy `ivec2`, `bvec3` i `vec4`.

W podobny sposób są utworzone 24 nazwy 18 typów macierzowych, przy czym składowe tych macierzy mogą być tylko typu `float` albo `double`, co jest określone odpowiednio brakiem przedrostka lub przedrostkiem `d` przed rdzeniem `mat`.

Przyrostek jest jedno- lub trzyznakowy i albo jest to cyfra, albo dwie cyfry przedzielone literą x. Typy macierzy kwadratowych mają po *dwie nazwy*, np. mat2 to jest to samo, co mat2x2, natomiast pozostałe macierze prostokątne mają tylko przyrostek trzyznakowy, np. dmat3x4.

Zmienne typów zamkniętych (*opaque types*) są w zasadzie reprezentowane jak liczby całkowite, ale służą tylko jako identyfikatory obiektów — tekstur, ewaluatorów tekstury i liczników niepodzielnych (*atomic counters*) — i nie mogą być argumentami działań arytmetycznych, ale mogą być parametrami podprogramów i mogą być składowymi zmiennych jednolitych i innych struktur. W sumie naliczyłem 41 typów zamkniętych (i 75 słów kluczowych będących ich nazwami), ale tu ich na razie nie wypiszę.

Nie ma typów wskaźnikowych. Jedyne zmienne wskaźnikowe, jakie mogą wystąpić w programie w GLSL-u to opisane dalej zmienne jednolite wskazujące podprogramy.

Struktury

Definicja typu strukturalnego wykorzystuje słowo kluczowe struct[3] i może być jednocześnie deklaracją zmiennych tego typu. Na przykład

```
struct moje {
  float a, b, c;
  vec4  qq[3];
} z;
```

definiuje nowy typ strukturalny o nazwie moje i jednocześnie deklaruje zmienną z tego typu. Struktura zadeklarowana wyżej ma cztery pola, z których trzy są typu float, a czwarte jest tablicą trzech wektorów, z których każdy ma cztery współrzędne. Po zdefiniowaniu typu strukturalnego można deklarować zmienne tego typu, pisząc np.

```
moje e, f, g;
```

Dostęp do pól struktury odbywa się za pomocą operatora kropki, na przykład e.c.

[3] Nie ma znanego z C słowa kluczowego typedef, ani powodu do używania go.

Tablice

Podanie po nazwie zmiennej nawiasów kwadratowych `[ ]` obejmujących stałe wyrażenie liczbowe, czyni z tej zmienej tablicę (o długości takiej, jak wartość tego wyrażenia). Zgodnie ze specyfikacją [3], dopuszczalne jest puste określenie długości *ostatniej* tablicy zadeklarowanej w bloku magazynowym (czyli tam i tylko tam można napisać np. `moje a[];`). W szczególności wszystkie tablice w blokach zmiennych jednolitych oraz tablicowe parametry podprogramów muszą mieć jawnie podaną długość. Tak jak w C, indeksy tablicy o długości n mają zakres od 0 do $n-1$. Tablica *może* być typem wyniku podprogramu — funkcji.

Deklaracja zmiennej tablicowej może zawierać wartości początkowe, podane w konstruktorze tablicy, np.

```
float a[4] = float[]{0.0, 1.0, 2.0, 3.0};
```

(można podać długość także w konstruktorze, obie długości muszą być wtedy jednakowe). Tablice są zasadniczo tylko jednowymiarowe, ale (tak jak w C) można deklarować tablice tablic, np.

```
float b[2][3];
```

Elementy takiej tablicy zajmują kolejne pozycje w pamięci, przy czym najszybciej (ze wzrostem adresów kolejnych elementów tablicy w pamięci) zmienia się *ostatni* indeks (ale między elementami tablicy mogą być odstępy, jeśli to wynika z reguł określonych przez układ (*layout*) przyjęty dla zmiennej tablicowej).

Tablica jest *obiektem* wyposażonym w *metodę* o nazwie `length`. Metoda ta podaje długość tablicy. Na przykład dla zmiennej `b` zadeklarowanej jak wyżej jest `b.length == 2`, `b[0].length == 3`.

Deklaracje zmiennych

Deklaracja zmiennej może być na zewnątrz podprogramów (globalna), wewnątrz podprogramu (lokalna), albo nawet wewnątrz instrukcji złożonej. W ostatnich dwóch przypadkach zmienna jest widoczna tylko między najciaśniej obejmującymi ją nawiasami klamrowymi, definiującymi zakres widoczności. Natomiast w pierwszym przypadku zakres widoczności zmiennej jest wszędzie tam, gdzie jej nazwa nie jest zasłonięta przez nazwę jakiegoś obiektu lokalnego. Można też zadeklarować zmienną sterującą pętli w jej nagłówku i wtedy zmienna jest

widoczna wewnątrz pętli, np.

```
for ( int i = 0; i < n; i++ ) { if ( i > 0 ) ...  }
```

Uwaga: Wszystkie nazwy zmiennych, typów strukturalnych i podprogramów znajdują się w *tej samej* przestrzeni nazw w danym zakresie widoczności. Zatem, choć nazwy podprogramów są przeciążane (np. funkcje obliczające iloczyn skalarny wektorów mają tę samą nazwę `dot`, ale każda ma argumenty innego typu), nie można mieć jednocześnie zmiennej i podprogramu o tej samej nazwie.

Globalna deklaracja zmiennej może (*nie musi*) być poprzedzona kwalifikatorem zmiennej (*storage qualifier*). Używane w profilu podstawowym kwalifikatory zmiennych są takie:

`buffer` — blok magazynowy, którego zawartość mogą przypisywać i odczytywać zarówno szadery, jak i aplikacja działająca na CPU, wywołując odpowiednie procedury OpenGL-a.

`const` — zmienna, której wartości nie wolno zmieniać; jej wartość jest przypisywana w deklaracji.

`in` — zmienna, której wartość została nadana przez wcześniejszy etap potoku przetwarzania grafiki.

`out` — zmienna, której szader ma nadać wartość, przekazywaną następnie do kolejnego etapu przetwarzania grafiki.

`shared` — zmienna, do której dostęp ma (działający jednocześnie na wielu procesorach GPU) szader obliczeniowy.

`uniform` — zmienna jednolita lub blok zmiennych jednolitych.

Zmienne globalne (tj. zadeklarowane poza podprogramem) opatrzone kwalifikatorami `buffer`, `in`, `out`, `shared` i `uniform` są tzw. zmiennymi interfejsu; zgodnie z opisem wyżej, służą one do przekazywania danych między aplikacją działającą na CPU i szaderami lub między poszczególnymi etapami potoku przetwarzania grafiki. Pozostałe zmienne globalne są widoczne tylko w obrębie jednego szadera.

Wyrażenia

Wyrażenia w GLSL-u, tak jak w C, buduje się z argumentów (stałych, zmiennych, funkcji) i operatorów zebranych w tabeli 9.1. Większość operatorów (ale nie wszystkie) ma takie samo działanie jak w języku C. W szczególności podobne są ich priorytety i łączność. Operatory o wyższym priorytecie są w tabeli podane wyżej (i od operatorów o niższym priorytecie są oddzielone kreską).

<table>
<tr><th>operator</th><th>opis</th><th>łączność</th></tr>
<tr><td>( )</td><td>ogranicznik podwyrażenia</td><td>nieokreślona</td></tr>
<tr><td>[ ]
( )
.
++, --</td><td>indeks tablicy
wywołanie podprogramu, konstruktor
dostęp do pól struktury, przestawianie
zwiększanie i zmniejszanie po</td><td>lewostronna</td></tr>
<tr><td>++, --
+, -, ~, !</td><td>zwiększanie i zmniejszanie przed
operacje jednoargumentowe</td><td>prawostronna</td></tr>
<tr><td>*, /, %</td><td>mnożenie, dzielenie, reszta z dzielenia</td><td>lewostronna</td></tr>
<tr><td>+, -</td><td>dodawanie, odejmowanie</td><td>lewostronna</td></tr>
<tr><td><<, >></td><td>przesunięcia bitowe</td><td>lewostronna</td></tr>
<tr><td><, >, <=, >=</td><td>operatory relacyjne</td><td>lewostronna</td></tr>
<tr><td>==, !=</td><td>operatory relacyjne</td><td>lewostronna</td></tr>
<tr><td>&</td><td>koniunkcja bitowa</td><td>lewostronna</td></tr>
<tr><td>^</td><td>bitowa alternatywa wyłączająca (xor)</td><td>lewostronna</td></tr>
<tr><td>|</td><td>alternatywa bitowa</td><td>lewostronna</td></tr>
<tr><td>&&</td><td>koniunkcja logiczna</td><td>lewostronna</td></tr>
<tr><td>^^</td><td>logiczna alternatywa wyłączająca</td><td>lewostronna</td></tr>
<tr><td>||</td><td>alternatywa logiczna</td><td>lewostronna</td></tr>
<tr><td>? :</td><td>wybór (operator trójargumentowy)</td><td>prawostronna</td></tr>
<tr><td>=
+=, -=, *=, /=, %=,
<<=, >>=, &=, ^=, |=</td><td>przypisanie
} działania z przypisaniem</td><td>prawostronna</td></tr>
<tr><td>,</td><td>separator wyrażeń</td><td>lewostronna</td></tr>
</table>

Tabela 9.1: Operatory GLSL-a, priorytety i łączność

Łączność określa kolejność wykonywania działań określonych przez sąsiadujące operatory o tym samym priorytecie. Na przykład w wyrażeniu `a-b+c` odejmowanie `a-b` zostanie wykonane najpierw, ponieważ priorytet obu operatorów w tym wyrażeniu jest taki sam, a ich łączność jest lewostronna[4].

[4]Przed nauczeniem się tabeli na pamięć, w razie wątpliwości, można używać nawiasów okrągłych do zapewnienia pożądanej kolejności wykonywania działań.

W języku GLSL nie ma wskaźników (z wyjątkiem opisanych dalej zmiennych wskazujących podprogramy), zatem nie ma znanych z C operatorów brania adresu (&) i sięgania pod adres (*). Nie ma też operatora sizeof; jest on zbędny, ponieważ szadery nie dokonują dynamicznej alokacji pamięci (wszelkie bufory alokuje i zwalnia kod działający na CPU).

Nie ma znanych z C operatorów rzutowania typu, dokonujących konwersji typu wartości wyrażenia. Zamiast tego słowa kluczowe int, uint, bool, float i double mogą być użyte w celu dokonania odpowiedniej konwersji, przy czym składnia jest taka, jak wywołanie funkcji — można napisać np. double(3). Konwersja liczby zmiennopozycyjnej (float lub double) do całkowitej (int lub uint) wiąże się z odrzuceniem części ułamkowej. Wartości logiczne false i true są konwertowane na liczby 0 (lub 0.0) i 1 (lub 1.0).

Argumentami operatorów logicznych mogą być tylko zmienne lub wyrażenia logiczne (typu bool). Argumentami operatorów arytmetycznych mogą być tylko zmienne liczbowe. Dzielenie przez 0 nie jest sygnalizowane (tzn. nie przerywa normalnego działania programu wywołaniem jakiejś procedury obsługi sygnału), ale jego wynik jest nieokreślony.

Operatory jednoargumentowe ~ i ! dokonują odpowiednio negacji bitowej (argument jest liczbą całkowitą, zmieniane są wszystkie bity) i logicznej (argument jest typu bool).

Typy wektorowe i macierzowe

Zmienna typu vec4 składa się z czterech pól typu float. Pola te mają aż trzy zestawy nazw: xyzw, rgba i stpq, które można wybierać dowolnie. Intencją jest poprawianie czytelności tekstu źródłowego, przez używanie nazw odpowiednich do zastosowania danej zmiennej: pierwszy zestaw jest odpowiedni dla współrzędnych punktów i wektorów opisujących obiekty geometryczne. Drugi zestaw sugeruje, że pola są współrzędnymi koloru (łącznie z kanałem alfa), a trzeci zestaw ma zastosowanie do opisu współrzędnych tekstury.

Typy vec2 i vec3 mają odpowiednio tylko dwie albo trzy początkowe nazwy pól w każdym zestawie. Dostęp do poszczególnych pól odbywa się za pomocą operatora kropki, np. dla zmiennej a typu vec4 możemy pisać a.x, a.t lub a.b, co wybiera odpowiednio pierwsze, drugie i trzecie pole. Nazwy tych pól (ale tylko wzięte z jednego, dowolnego zestawu) można sklejać, aby uzyskać dostęp do kilku pól naraz, w celu ich „wypreparowania”, poprzestawiania lub nawet powielenia. Na

przykład wyrażenia `a.xy`, `a.ywz` i `a.xxxx` mają odpowiednio typy vec2, vec3 i vec4; pierwszy wektor ma pierwsze dwie współrzędne wektora a, drugi ma jego ostatnie trzy współrzędne, przy czym ostatnie dwie są przestawione, a wszystkie cztery współrzędne trzeciego wektora są równe pierwszej współrzędnej wektora a. Nie wolno w ten sposób „wyprodukować" wektora o większej niż 4 liczbie współrzędnych.

Wektory i wybrane ich pola można dodawać i odejmować (byleby składniki miały tyle samo współrzędnych), a także mnożyć i dzielić przez liczby. W szczególności wyrażenie `a.xyz/a.w` ma typ vec3; produkuje ono wektor współrzędnych kartezjańskich punktu, którego współrzędne jednorodne są wartościami pól wektora a.

Konstruktory wektorów mają nazwy takie, jak typy tych wektorów. Jeśli zmienne `b`, `c` są typu vec2, to zmiennej a typu vec4 możemy nadać wartość wyrażenia takiego jak `vec4(b,c)`, `vec4(b.xxy,1.0)` lub `vec4(1.0)`. W ostatnim przypadku wszystkie cztery pola otrzymują tę samą wartość.

Macierz typu mat4 (mającego też nazwę mat4x4) ma cztery wiersze i tyleż kolumn; dostęp do poszczególnych współczynników uzyskuje się tak, jakby to była tablica, np. `a[2][1]` jest to trzeci współczynnik w drugim wierszu[5]. Można odwoływać się do całych kolumn, pisząc np. `a[2]` — dla macierzy a typu mat4 jest to wektor typu vec4.

Działania na wektorach i macierzach są tak intensywnie wykorzystywane, że zapisuje się je jak działania elementarne, bez potrzeby obliczania osobno wszystkich współrzędnych wyniku[6] (np. mnożenie macierzy a przez wektor v zapisujemy jako wyrażenie `a*v`). Istotne jest dopasowanie argumentów działań — dodawane i odejmowane mogą być wektory i macierze o tych samych wymiarach. W mnożeniu dwóch macierzy liczba kolumn pierwszej z nich musi być równa liczbie wierszy drugiej.

Powyższe uwagi stosują się też do wektorów i macierzy, których pola mają typ inny niż `float`.

[5] Odwrotnie niż zazwyczaj w C — to ma związek z kolumnowym przechowywaniem współczynników macierzy typów mat2x2, ..., mat4x4

[6] Procesory graficzne mają specjalne rozkazy wykonujące działania na całych wektorach i macierzach (szczegóły znają tylko producenci sprzętu).

Instrukcje

Instrukcje w GLSL są podobne do tych w C, choć gramatyka opisująca ich składnię różni się od gramatyki C. Ale mamy w GLSL do dyspozycji instrukcje proste — deklaracje, instrukcje wyrażeniowe, instrukcje wyboru, pętle i skoki, z których można budować instrukcje złożone, ujmując ciąg instrukcji w nawiasy klamrowe `{ }`. Deklaracje, instrukcje wyrażeniowe i skoki, jak w C, muszą być zakończone średnikiem.

Instrukcja wyrażeniowa jest wyrażeniem; w szczególności może ono zawierać wywołania podprogramów i operatory przypisania. Wyrażenie może być puste, instrukcja wyrażeniowa z pustym wyrażeniem nic nie robi.

Instrukcje wyboru to znane z C instrukcje

```
if ( <warunek> ) <instrukcja>
if ( <warunek> ) <instrukcja> else <instrukcja>
switch ( <wyrażenie> ) { <instrukcje do wyboru> }
```

przy czym *<warunek>* musi być wyrażeniem logicznym[7]. Wyrażenie w instrukcji przełącznika (`switch`) musi być całkowite. Instrukcje do wyboru są opatrzone etykietami postaci `case` *<wyrażenie stałe>*`:` albo `default:`, a całość działa tak, jak instrukcja przełącznika w C.

Pętle mają trzy znane z C postaci

```
for ( <wyrażenie>; <warunek>; <wyrażenie> ) <instrukcja>
while ( <warunek> ) <instrukcja>
do <instrukcja> while ( <warunek> );
```

Warunek w każdym przypadku musi być wyrażeniem logicznym (tj. typu `bool`). Poza tym ograniczeniem powyższe instrukcje działają tak, jak ich odpowiedniki w C.

Zagnieżdżanie instrukcji warunkowych i pętli może wymagać użycia nawiasów klamrowych, tak samo jak w C. Na przykład poniższe instrukcje, choćby nawet miały identyczne warunki i instrukcje składowe, robią co innego:

---

[7]Inaczej niż w C, gdzie dopuszczalne są także wyrażenia liczbowe i wskaźnikowe.

```
if ( <warunek> )
  while ( <warunek> )
    if ( <warunek> )
      <instrukcja>
else <instrukcja>
```

```
if ( <warunek> ) {
  while ( <warunek> )
    if ( <warunek> )
      <instrukcja>
} else <instrukcja>
```

Instrukcje skoku to znane z C instrukcje continue;, break; i return; oraz nowa instrukcja discard;. Nie ma instrukcji skoku goto.

Instrukcja continue; może wystąpić w pętli. Jej wykonanie pomija instrukcje do końca najbardziej wewnętrznej pętli, w której ta instrukcja się znajduje. Zależnie od warunku sterującego wykonaniem tej pętli następuje potem kolejna iteracja albo pętla kończy działanie.

Instrukcja break; kończy działanie najbardziej wewnętrznej pętli lub instrukcji przełącznika.

Instrukcja return; (albo return *<wyrażenie>*;) kończy działanie podprogramu, ewentualnie przekazując wartość wyrażenia jako wartość funkcji. Wystąpienie instrukcji return; w procedurze main szadera kończy jego działanie.

Instrukcja discard;, dopuszczalna tylko w szaderach fragmentów, kończy działanie szadera i powoduje odstąpienie od dalszego przetwarzania fragmentu — po jej wykonaniu odpowiedni piksel w buforze obrazu pozostanie niezmieniony.

Podprogramy

Tak jak w C, podprogramy w GLSL-u są oficjalnie nazywane *funkcjami*, co może gmatwać opis, w którym słowo „funkcja” jest używane także (albo przede wszystkim) w znaczeniu przyjętym w matematyce. Dlatego będę raczej używał słów „podprogram” lub „procedura”. Podprogram składa się z nagłówka i bloku (instrukcji złożonej, tj. ciągu instrukcji zamkniętego w nawiasach klamrowych). Nagłówek, po którym następuje średnik *zamiast* instrukcji, jest prototypem podprogramu. Prototyp zawiera informacje konieczne i wystarczające do wygenerowania przez kompilator ciągu rozkazów przekazujących parametry i wywołujących podprogram, który może być częścią innego szadera (tego samego etapu); ten mechanizm jest podobny do mechanizmu znanego z C. Zatem, w tekście źródłowym szadera przed pierwszą instrukcją wywołującą podprogram musi wystąpić albo ten podprogram, albo jego prototyp (oczywiście, nie dotyczy to podprogramów wbudowanych GLSL-a).

Ogólna postać podprogramu to

```
<typ wyniku> nazwa ( <param_1>, ..., <param_n> )
{
  ... /* jakieś instrukcje */
  return <wyrażenie>;
}
```

Typ wyniku musi być jawnie podany (nie ma domyślnego typu wyniku, jakim w C jest int) i może to być typ liczbowy, strukturalny lub tablicowy. Jeśli podprogram nie przekazuje wyniku obliczeń przez swoją nazwę w wywołaniu, to powinien mieć typ void i wtedy wyrażenie w instrukcji return musi być puste. W przeciwnym razie typ wyrażenia musi być identyczny z typem wyniku podanym w nagłówku lub musi być możliwa niejawna konwersja (np. typu int do float).

Lista parametrów może być pusta (między nawiasami okrągłymi można nic nie pisać lub napisać słowo kluczowe void), albo może zawierać parametry. Każdy parametr jest zmienną, która ma nazwę poprzedzoną typem parametru, przed którym *można* podać kwalifikator parametru. Jest on słowem kluczowym in, out lub inout. Przed kwalifikatorem in można podać dodatkowy kwalifikator const, który nie dopuszcza zmieniania wartości parametru przez instrukcje w podprogramie, mogą też być użyte kwalifikatory precyzji, których opis pominę.

Parametry podprogramu są jego zmiennymi lokalnymi; parametry wejściowe (opatrzone kwalifikatorem in lub bez kwalifikatora) otrzymują wartości podczas wywołania podprogramu. Wartości parametrów wyjściowych (z kwalifikatorem out) są w trakcie powrotu z podprogramu kopiowane do zmiennych przekazanych jako parametry, zaś parametry wejściowo-wyjściowe (z kwalifikatorem inout) podlegają odpowiednim operacjom i na początku i na końcu działania podprogramu.

Działanie podprogramu kończy instrukcja return (lub discard), ale jeśli takiej instrukcji autor podprogramu nie napisał, to powrót z podprogramu następuje po wykonaniu ostatniej instrukcji w jego bloku. Jeśli jednak podprogram jest funkcją (tj. jego typ wyniku jest inny niż void), to trzeba napisać instrukcję return z wyrażeniem określającym wartość tej funkcji.

Nazwy podprogramów mogą być *przeciążane*, tj. może istnieć wiele podprogramów o tej samej nazwie. Muszą one różnić się listami parametrów, tj. mieć inne liczby parametrów lub przynajmniej inne *typy* parametrów tak, aby

kompilator mógł na podstawie listy parametrów wywołania podprogramu wybrać odpowiedni podprogram.

Punktem wejściowym szadera jest podprogram o nazwie `main`. Jego nagłówek powinien mieć postać

```
void main ( void )
```

ponieważ wszelkie dane i wyniki szadery przekazują przez zmienne interfejsu.

Rekurencyjne wywoływanie podprogramów jest niedozwolone.

Zmienne wskazujące podprogramy

Zamiast instrukcji wyboru (dokonywanego np. na podstawie wartości jakiejś liczbowej lub logicznej zmiennej jednolitej) można do modyfikowania działania szadera używać jednolitych zmiennych wskazujących podprogramy. W tym celu trzeba zadeklarować typ podprogramu, co wygląda tak:

```
subroutine <typ wyniku> nazwa_typu ( <param_1>, ..., <param_n> );
```

Deklaracja wygląda zatem jak prototyp podprogramu poprzedzony słowem kluczowym `subroutine`. Podprogram, który ma być wskazywany, musi być poprzedzony słowem kluczowym `subroutine` i podaną w nawiasach nazwą typu (lub listą nazw typów, oddzielonych przecinkami) zadeklarowanym jak wyżej, przy czym typ wyniku i lista parametrów podprogramu muszą być takie same jak typ wyniku i lista parametrów w deklaracji typu podprogramu. Czyli na przykład

```
subroutine (nazwa_typu)
<typ wyniku> nazwa ( <param_1>, ..., <param_n> )
{
  ...  /* jakieś instrukcje */
  return <wyrażenie>;
}
```

Wreszcie zmienne wskazujące podprogram muszą być globalne i jednolite, zadeklarowane w taki sposób:

```
subroutine uniform nazwa_typu nazwa_zmiennej;
```

Przypisanie wartości takiej zmiennej może wykonać *tylko* aplikacja działająca na CPU, za pomocą procedur `glGetSubroutineUniformLocation`, która podaje identyfikator położenia zmiennej wskazującej, `glGetSubroutineIndex`, która podaje tzw. indeks podprogramu[8] i `glUniformSubroutinesuiv`, która dokonuje przypisania odpowiednich wartości zmiennym wskazującym. Przykłady użycia podprogramów wskazywanych będą pokazane w aplikacji drugiej C i dalszych. Składnia wywołania podprogramu wskazywanego jest taka jak składnia wywołania „zwykłego" podprogramu, ale zamiast nazwy podprogramu podaje się nazwę zmiennej wskazującej.

Można deklarować tablice zmiennych wskazujących podprogramy i w wywołaniu podawać indeks do tablicy, ale z uwagi na jednolitość obliczeń (zobacz niżej) wszystkie działające równolegle instancje szadera muszą w chwili wywołania podać ten sam indeks takiej tablicy.

Uwaga: Deklaracja typu wskazującego, zmiennych wskaźnikowych tego typu i wszystkich podprogramów, które mogą być wskazywane przez te zmienne, muszą być elementami *jednego* szadera. Program szaderów może zawierać *kilka* szaderów tego samego etapu (wierzchołków, rozdrabniania, geometrii lub fragmentów), z których tylko jeden ma procedurę `main`, wywołującą podprogramy z innych szaderów tego samego etapu (tekst źródłowy szadera wywołującego podprogramy z innego szadera musi zawierać prototypy tych podprogramów). Ale umieszczenie deklaracji typu wskazującego w więcej niż jednym szaderze powoduje błąd sygnalizowany na etapie łączenia programu szaderów.

Równoległość i jednolitość obliczeń

GPU składa się z wielu procesorów, które w odróżnieniu od poszczególnych rdzeni CPU nie pracują niezależnie. Przetwarzanie danych (np. dla wierzchołków lub fragmentów) odbywa się równolegle, przy czym poszczególne procesory GPU przetwarzają *różne dane* wykonując jednocześnie *te same rozkazy*. Jeśli jednak poszczególne instancje szadera, po obliczeniu warunku w instrukcji wyboru (`if`, `if`–`else`, `switch`) albo warunku sterującego wykonaniem lub zakończeniem pętli (`while`, `do`–`while`, `for`), muszą wykonać różne instrukcje, to część procesorów (mających do wykonania te same instrukcje) działa dalej, podczas gdy pozostałe procesory czekają. Ma to oczywisty (i niekorzystny) wpływ na szybkość obliczeń.

Autorzy szaderów mogą podejmować wysiłki zmierzające do zminimalizowania

[8] Te dwie procedury trzeba wywołać po skompilowaniu i połączeniu programu szaderów i zapamiętać podane przez nie informacje.

kodu wykonywanego niejednolicie (np. usuwając z instrukcji wyboru wszelkie obliczenia wspólne dla wszystkich możliwych wyborów). Istotniejsze są ograniczenia dla konstrukcji językowych. To z tego powodu jeśli szader ma wykonać podprogram wskazywany przez zmienną będącą elementem tablicy, to wszystkie instancje szadera muszą obliczyć ten sam indeks do tej tablicy.

Bloki zmiennych interfejsu

Zmienne interfejsu służące do przekazywania informacji między szaderami i aplikacją na CPU można grupować w bloki interfejsu. Deklaracja bloku interfejsu składa się z opcjonalnego kwalifikatora układu (`layout`), obowiązkowego kwalifikatora interfejsu, nazwy bloku, listy pól w nawiasach klamrowych i opcjonalnej nazwy instancji (tj. prywatnej nazwy bloku dla szadera).

Kwalifikator interfejsu jest słowem kluczowym `buffer`, `in`, `out` lub `uniform`, albo parą słów `patch in` lub `patch out`.

Dosyć liczne przykłady deklaracji bloku interfejsu spotkaliśmy już we wcześniejszych rozdziałach, zobacz np. listing 7.1, na którym są deklaracje bloków o nazwach (globalnych) `Vertex` i `TransBlock`. Pierwszy z tych bloków służy do przekazania danych między etapami w potoku przetwarzania grafiki (z szadera wierzchołków do etapu obcinania — w aplikacji w rozdz. 7 nie ma szaderów rozdrabniania ani geometrii), a drugi jest blokiem zmiennych jednolitych. Bloki te mają prywatne nazwy (nazwy instancji) `Out` i `trb`. Szader do pól w tych blokach odwołuje się za pomocą operatora kropki następującego po nazwie prywatnej, np. `trb.pm`. Jeśli nie ma nazwy prywatnej, to w odwołaniach do pól szader nie podaje także kropki, czyli gdyby deklaracja bloku miała postać

```
uniform TransBlock {
  mat4 mm, vm, pm;
};
```

to do jego pól `mm`, `vm`, `pm` szader odwoływałby się bez żadnych dodatkowych ceregieli[9]. Ale nazwa instancji jest potrzebna, jeśli blok interfejsu jest tablicą. Po nazwie instancji podaje się wtedy nawiasy kwadratowe `[ ]`, ewentualnie obejmujące wyrażenie stałe (liczbę naturalną — długość tablicy). W odwołaniach

[9] Coś tu jednak nie gra: jak nie podałem nazwy instancji, to kompilacja szaderów i łączenie programu przebiegły bez błędów, ale procedura `glGetUniformIndices` przekazała indeksy -1, czyli nie odnalazła pól bloku `TransBlock`. Kiedyś się dowiem, o co tu chodzi, a tymczasem piszę nazwy instancji bloków zmiennych jednolitych i już.

do pól takiego bloku muszą być podane odpowiednie indeksy (mogłoby to wyglądać tak: `trb[i].mm`).

Komunikacja między szaderami

Komunikacja między szaderami odbywa się za pomocą zmiennych wbudowanych i bloków interfejsu. Podstawowe dane są przekazywane w zmiennych wbudowanych, „gotowych do użycia" bez potrzeby deklarowania ich, zaś bloki interfejsu są wprowadzane przez autora szaderów aby przekazywać wszelkie dane specyficzne dla konkretnego programu szaderów.

Na wszystkich etapach części przedniej potoku przetwarzania grafiki występują zmienne strukturalne o nazwie zewnętrznej `gl_PerVertex`. To *nie jest* nazwa typu strukturalnego, tylko zewnętrzna nazwa zmiennej strukturalnej, która może nie mieć nazwy instancji, ale jeśli występuje jednocześnie jako zmienna wejściowa i wyjściowa (np. szadera rozdrabniania lub geometrii), to jedna lub druga zmienna `gl_PerVertex` ma nazwę instancji — co najmniej jedna z nich jest tablicą i wtedy ma nazwę instancji `gl_in` lub `gl_out`.

```
<kwalifikator> gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
  };
```

Pole `gl_Position` jest wektorem współrzędnych jednorodnych wierzchołka; jest to najważniejsza dana, którą szadery odczytują ze zmiennej wejściowej i zapisują w zmiennej wyjściowej. Pozostałe pola mają wartości domyślne, które szader może nadpisać, ale może też zignorować. Pole `gl_PointSize` jest średnicą (w pikselach) kropki[10], która zostanie narysowana, jeśli wierzchołek jest wyświetlany w trybie `GL_POINTS` (parametr określający tryb podaje się w wywołaniu procedur z rodziny `glDraw`). Pola `gl_ClipDistance` i `gl_CullDistance` są tablicami, których długości są liczbami płaszczyzn obcinających i odrzucających. Płaszczyznami obcinającymi są płaszczyzny ścian kostki standardowej (których jest 6) i dodatkowe płaszczyzny obcinające wprowadzone przez aplikację. Płaszczyzny odrzucające służą do odrzucania całych prymitywów. Szader, który korzysta z którejś z tych tablic musi ponowić jej deklarację, jawnie podając długość tablicy. Szczegółami obcinania zajmiemy się później.

[10] a ściślej jest to wysokość i szerokość kwadratu

W programach szaderów odwołujących się do tablic gl_ClipDistance i gl_CullDistance należy przedefiniować strukturę gl_PerVertex, podając jawnie długości tych tablic (w postaci wyrażenia stałego). Inne dopuszczalne modyfikacje polegają na pominięciu pól nieużywanych, natomiast nie wolno niczego dodać. Zmodyfikowana struktura musi być taka sama we wszystkich szaderach programu.

Zmienne wbudowane szadera wierzchołków:

in int gl_VertexID; — numer wierzchołka rysowanego prymitywu, może mieć wartość nieokreśloną.

in int gl_InstanceID; — numer instancji prymitywu rysowanego w wielu egzemplarzach (np. przez procedurę glDrawArraysInstanced).

out gl_PerVertex { ... }; — dane opisujące wierzchołek obliczone przez szader.

Zmienne wbudowane szadera sterowania rozdrabnianiem:

in gl_PerVertex { ... } gl_in[gl_MaxPatchVertices]; — tablica wierzchołków płata dostarczonych przez szader wierzchołków.

in int gl_PatchVerticesIn; — liczba wierzchołków płata (tj. długość tablicy gl_in).

in int gl_PrimitiveID; — wartość tej zmiennej jest liczbą prymitywów elementarnych (np. trójkątów) przetworzonych od początku bieżącego zbioru prymitywów (np. taśmy trójkątowej).

in int gl_InvocationID; — numer wierzchołka w obrębie płata, od zera do liczby o 1 mniejszej niż liczba wierzchołków płata.

out gl_PerVertex { ... } gl_out[]; — tablica, do której (do jednego elementu) szader ma przypisać dane wyjściowe opisujące wierzchołek.

patch out float gl_TessLevelOuter[4]; — poziomy rozdrabniania brzegu płata.

patch out float gl_TessLevelInner[2]; — poziomy rozdrabniania wewnątrz płata.

Szader sterowania rozdrabnianiem ma za zadanie przekazać na wyjście dane opisujące *jeden* wierzchołek (o numerze `gl_InvocationID`) i wypełnić odpowiednimi danymi tablice `gl_TessLevelOuter` i `gl_TessLevelInner`, ale to ma zrobić *tylko jedna* instancja szadera — najprościej jest umieścić instrukcje przypisania do tych tablic w instrukcji warunkowej

```
if ( gl_InvocationID == 0 ) { ... }
```

Zmienne wbudowane szadera rozdrabniania:

`in gl_PerVertex { ... } gl_in[gl_MaxPatchVertices];` — tablica wierzchołków płata dostarczonych przez szader sterowania rozdrabnianiem (jeśli jest obecny w programie) lub przez szader wierzchołków.

`in int gl_PatchVerticesIn;` — liczba wierzchołków płata (tj. długość tablicy `gl_in`).

`in int gl_PrimitiveID;` — ma tę samą wartość, co dla szadera sterowania rozdrabnianiem.

`in vec3 gl_TessCoord;` — wektor współrzędnych kartezjańskich lub barycentrycznych punktu *w dziedzinie płata* otrzymanego w etapie rozdrabniania.

`patch in float gl_TessLevelOuter[4];` — poziomy rozdrabniania brzegu płata.

`patch in float gl_TessLevelInner[2];` — poziomy rozdrabniania wewnątrz płata.

`out gl_PerVertex { ... } gl_out[];` — opis wytworzonego przez szader wierzchołka płata.

Zmienne wbudowane szadera geometrii:

`in gl_PerVertex { ... } gl_in[];` — tablica wierzchołków primitywu elementarnego (pojedynczego wierzchołka, końców odcinka lub wierzchołków trójkąta).

`in int gl_PrimitiveIDIn;` — ma tę samą wartość, co zmienne `gl_PrimitiveID` szaderów sterowania rozdrabnianiem i rozdrabniania.

in int `gl_InvocationID;` — numer wierzchołka prymitywu, np. dla trójkąta ma wartość 0, 1 lub 2, wtedy tablica `gl_in` ma długość 3.

out `gl_PerVertex { ... };` — opis wytworzonego przez szader wierzchołka prymitywu.

out int `gl_PrimitiveID;` — identyfikator prymitywu, przekazywany na wejście szadera fragmentów w jego zmiennej wejściowej `gl_PrimitiveID`.

out int `gl_Layer;` — numer warstwy, może służyć m.in. do wyboru ściany sześcianu utworzonego z tekstur.

out int `gl_ViewportIndex;` — numer klatki, tj. obszaru na obrazie, na którym prymityw ma być narysowany (domyślnie otrzymuje wartość 0).

Szader geometrii otrzymuje tablicę wierzchołków prymitywu (z dwoma końcami odcinka lub z trzema wierzchołkami trójkąta) i ma wygenerować odpowiedni ciąg wierzchołków. Zamiast wpisywać je do tablicy (tak jak szader sterowania rozdrabnianiem), szader geometrii ma kolejno dla każdego wierzchołka wpisać dane do pól struktury wyjściowej `gl_PerVertex` (i w razie potrzeby do swojego wyjściowego bloku interfejsu z danymi dodatkowymi), a następnie wywołać procedurę `EmitVertex`. Na zakończenie powinien wywołać procedurę `EndPrimitive`.

Zmienne wbudowane szadera fragmentów:

in vec4 `gl_FragCoord;` — wektor współrzędnych fragmentu w oknie, $(x, y, z, 1/W)$. Liczby x i y określają punkt na obrazie, z jest głębokością punktu, natomiast W jest współrzędną wagową wektora współrzędnych jednorodnych (X, Y, Z, W) punktu otrzymanego w etapie rasteryzacji, któremu odpowiada dany fragment.

in bool `gl_FrontFacing;` — ma wartość true jeśli prymityw jest odwrócony przodem do obserwatora i false w przeciwnym razie.

in float `gl_ClipDistance[];` — tablica odległości punktu od płaszczyzn obciania.

in float `gl_CullDistance[];` — tablica odległości punktów od płaszczyzn odrzucania.

in vec2 gl_PointCoord; — dotyczy wierzchołków rysowanych jako punkty (w trybie GL_POINTS). Jego współrzędne przyjmują wartości od 0 do 1 dla punktów w obszarze (kwadratowej) kropki. Można na ich podstawie, korzystając z instrukcji discard, nadać kropce kształt inny niż kwadratowy.

in int gl_PrimitiveID; — identyfikator prymitywu.

in int gl_Layer; — numer warstwy nadany przez szader geometrii (wartość nieokreślona, jeśli to nie nastąpiło)

in int gl_ViewportIndex; — numer klatki nadany przez szader geometrii (0, jeśli to nie nastąpiło).

in bool gl_HelperInvocation; — ma wartość true podczas wywołania pomocniczego i false kiedy indziej. Wywołania pomocnicze są wykonywane w celu obliczenia gradientu koloru (albo tekstury).

in int gl_SampleID; — numer próbki w technice wielokrotnego próbkowania (*multisampling*) używanej do antyaliasingu.

in vec2 gl_SamplePosition; — położenie próbki w pikselu.

in int gl_SampleMaskIn[]; — pole bitowe, które opisuje zbiór próbek pokrytych przez prymityw geometryczny, którego fragment jest przetwarzany.

out int gl_SampleMask[]; — pole bitowe, które opisuje zbiór próbek pokrytych przez prymityw geometryczny, którego fragment jest przetwarzany. Szader fragmentów może zmodyfikować pole dane na wejściu w tablicy gl_SampleMaskIn.

out float gl_FragDepth; — głębokość punktu, używana dalej w testach widoczności. Szader fragmentów może zmodyfikować jej wartość, wpływając w ten sposób na wynik testu.

Wiele zmiennych interfejsu szadera fragmentów jest związanych z antyaliasingiem, tj. poprawianiem jakości obrazu przez „wygładzanie" ząbkowanych krawędzi na obrazie rastrowym. Najważniejszym zadaniem szadera fragmentów jest obliczenie koloru fragmentu, przy czym, co ciekawe, trzeba w tym celu jawnie zadeklarować zmienną wyjściową typu vec4 (nie ma zmiennej wbudowanej służącej do tego celu). Współrzędne r,g,b,a tej zmiennej muszą mieć wartości z przedziału $[0, 1]$ i opisują odpowiednio składowe czerwoną, zieloną i niebieską oraz kanał alfa. Ten ostatni jest parametrem dla ostatniego etapu w potoku przetwarzania grafiki,

w którym (po przejściu przez fragment testu widoczności) następuje końcowe obliczenie koloru przypisywanego pikselowi, w którym zależnie od przyjętej funkcji mieszającej może być uwzględniony dotychczasowy kolor piksela (i wartości współrzędnej alfa koloru dotychczasowego i koloru przekazanego przez szader).

Zmienne wbudowane szaderów obliczeniowych będą opisane dalej.

Kwalifikatory układu

Kwalifikatory układu (*layout qualifiers*) spełniają dwie role w opisie zmiennych w GLSL-u: wpływają na sposób rozmieszczania składowych (np. pól struktury) w pamięci (czyli na przydzielanie tym składowym adresów przez kompilator GLSL-a) a także (dla zmiennych interfejsu) zawierają informacje umożliwiające właściwe zorganizowanie przepływu danych między etapami potoku przetwarzania grafiki. Kwalifikator układu może występować osobno, może poprzedzać zmienną, blok (tj. strukturę) lub indywidualne składowe takiego bloku. Ma on ogólną postać

```
layout(<lista identyfikatorów kwalifikatora>)
```

przy czym `<lista identyfikatorów kwalifikatora>` to jeden lub kilka (przedzielonych przecinkami) napisów postaci *<nazwa>* lub *<nazwa>*=*<wartość>*. Nie da się pokrótce opisać wszystkich nazw, ich znaczenia ani możliwych wartości (w konkretnych miejscach) i nie warto wkuwać wszystkiego na ten temat, zwłaszcza „na sucho”, tj. w oderwaniu od zastosowań. Dlatego poniżej opisuję tylko kilka najważniejszych przykładów.

Dla zmiennych jednolitych możemy podać kwalifikator `shared` (domyślny), `std140` lub `packed`. Osobny kwalifikator ma np. postać

```
layout(std140) uniform;
```

i występujące za nim zmienne jednolite będą kompilowane w takim układzie, z wyjątkiem zmiennych indywidualnie opatrzonych innym kwalifikatorem układu, na przykład

```
layout(shared) uniform MojaZmienna { ... } jednolita;
```

Kwalifikator `packed` jest najbardziej oszczędny pod względem przydziału pamięci, ale nie może być używany dla bloków zmiennych jednolitych, do których dostęp

ma więcej niż jeden szader (nawet w tym samym programie szaderów)[11]. Domyślny kwalifikator shared jest zależny od implementacji, ale zapewnia zgodność układu opisanych tak samo zmiennych w różnych (oddzielnie kompilowanych) szaderach i jest dosyć oszczędny, m.in. nie zostawia pustych miejsc między elementami tablic liczb całkowitych. Kwalifikator układu `std140`[12] wyrównuje adres każdego elementu tablicy do wartości podzielnej przez 4 razy rozmiar skalarnej składowej elementu; na przykład tablica liczb typu int albo bool w układzie `std140` dla każdego elementu rezerwuje 16 bajtów, z których tylko 4 będą przechowywać odpowiednią liczbę. Dokładne reguły przydziału adresów w tym układzie można znaleźć w Internecie. Dają one możliwość obliczenia przesunięć poszczególnych elementów przez aplikację w C (albo przez jej autora) i uniknięcia korzystania z procedury `glGetActiveUniformsiv` (zobacz listing 7.3 i jego opis). W szczególności można napisać takie definicje typów strukturalnych w C (z nieużywanymi polami powodującymi odpowiednie modyfikacje położeń pól używanych), aby przesunięcia potrzebnych pól względem początku struktury w C były identyczne z przesunięciami odpowiadających im pól względem początku bufora w pamięci GPU, co umożliwia przesyłanie zawartości wielu pól za pomocą *jednego* wywołania procedury `glBufferData`[13].

Kwalifikatory układu zmiennych intefejsu oprócz rozmieszczenia pól struktur w pamięci podają informacje potrzebne dla dopasowania wejścia i wyjścia szaderów do innych etapów potoku przetwarzania grafiki. Poniżej opisuję tylko niektóre kwalifikatory zmiennych interfejsu.

Zmienne przekazywane przez etap pobierania wierzchołków do szadera wierzchołków muszą mieć kwalifikator miejsca

```
layout(location=n)
```

(przykład był na listingu 7.1), w którym wyrażenie stałe n oznacza numer miejsca

[11] Przyczyna jest taka, że w ramach optymalizacji kompilator może usunąć pola, do których szader się nie odwołuje. Jeśli więc dwa różne szadery odwołują się do różnych pól, to programu nie da się poprawnie połączyć.

[12] Nazwa jest związana z wersją języka GLSL, w której ten układ został wprowadzony; to samo dotyczy nazwy bardziej upakowanego układu `std430`, który może być stosowany dla buforów magazynowych, ale nie dla zmiennych jednolitych.

[13] Nie jestem przekonany, czy ta alternatywa jest wygodniejsza. Na pewno jest bardziej podatna na błędy, a marnowanie pamięci w układzie `std140` (motywowane osiąganiem jak najszybszego dostępu do pamięci) może być duże — zwłaszcza dla tablicy elementów typu bool. Ale można, przynajmniej teoretycznie, napisać translator, który po przeczytaniu opisu struktury w GLSL-u obliczy przesunięcia poszczególnych pól i wygeneruje tekst źródłowy z deklaracją odpowiedniej struktury w C.

zajmowanego przez atrybut; każdy atrybut skalarny lub wektorowy zajmuje jedno miejsce (możemy więc numerować miejsca 0, 1 itd.). Wyjątkiem są atrybuty typu dvec3 i dvec4, które zajmują dwa kolejne miejsca. Zmienne z atrybutami wierzchołków mogą być tablicami i wtedy zajmują odpowiednio więcej miejsc (tyle, ile tablica ma elementów), zaczynając od miejsca podanego w kwalifikatorze.

Szader sterowania rozdrabnianiem może mieć *tylko* kwalifikator układu wyjścia, o postaci

```
layout(vertices=n) out;
```

z wyrażeniem stałym n określającym liczbę wierzchołków płata, będącą też długością tablicy gl_out (zobacz opis wbudowanych zmiennych interfejsu) i liczbą wywołań szadera — po jednym razie dla każdego wierzchołka.

Wejście do szadera rozdrabniania powinno mieć jeden z następujących kwalifikatorów układu: triangles, quads, isolines. Dodatkowo można podać kwalifikatory equal_spacing, fractional_even_spacing albo fractional_odd_spacing oraz cw, ccw albo point_mode. Pierwsze trzy kwalifikatory określają kształt dziedziny rozdrabnianego płata (trójkąt, kwadrat albo rodzina linii równoległych w kwadracie). Kolejne trzy kwalifikatory sterują rozmieszczeniem w tej dziedzinie wierzchołków generowanych przez etap rozdrabniania dziedziny. Ostatnie trzy wybierają orientację trójkątów wytwarzanych przez ten etap (zgodnie z ruchem wskazówek zegara albo przeciwnie) lub tryb, w którym przekazywane są tylko osobne wierzchołki. Możemy napisać na przykład

```
layout(triangles,cw,equal_spacing) in;
```

Przykłady wyjaśniające jak to działa są w opisach aplikacji pierwszej D i drugiej.

Wejście szadera geometrii ma mieć jeden z następujących kwalifikatorów: points, lines, lines_adjacency, triangles, triangles_adjacency. Kwalifikatory te wybierają rodzaj prymitywu przetwarzanego przez szader geometrii, punkty, odcinki lub trójkąty, przy czym kwalifikatory „adjacency” dostarczają także wierzchołki sąsiednich odcinków lub trójkątów. Dodatkowo może być kwalifikator invocations=n, gdzie n jest liczbą wywołań szadera geometrii dla każdego prymitywu (domyślnie jest jedno wywołanie).

Wyjście szadera geometrii może mieć kwalifikator points, line_strip albo triangle_strip oraz max_vertices=n, określający liczbę wierzchołków

wyjściowej łamanej lub taśmy trójkątowej — przykład będzie w opisie aplikacji pierwszej B.

Zmienna wejściowa gl_FragCoord szadera fragmentów może być przedeklarowana kwalifikatorem origin_upper_left i/lub pixel_center_integer, na przykład

```
layout(pixel_center_integer) in vec4 gl_FragCoord;
```

Deklaracja ta zmienia domyślny układ współrzędnych w oknie (z początkiem w dolnym lewym narożniku i osią y skierowaną do góry) na układ odwrócony (z początkiem w górnym lewym narożniku i osią y skierowaną do dołu). Kwalifikator pixel_center_integer przesuwa początek układu w poziomie i pionie o połowę szerokości i wysokości piksela, wskutek czego środki pikseli mają współrzędne całkowite zamiast liczb o częściach ułamkowych równych 0.5.

Zmienna wyjściowa gl_FragDepth szadera fragmentów może być przedeklarowana z kwalifikatorem depth_any (domyślny), depth_greater, depth_less albo depth_unchanged, np.

```
layout(depth_unchanged) out float gl_FragDepth;
```

Używanie tych kwalifikatorów ma związek z optymalizacją obliczeń w powiązaniu z testami widoczności; jeśli dany fragment nie jest widoczny (bo jest zasłonięty przez pewien fragment narysowany wcześniej), to zazwyczaj nie warto tracić czasu na skomplikowane obliczenia teksturowania i oświetlenia tego fragmentu.

Opis pozostałych kwalifikatorów można znaleźć w dokumentacji [3], w miarę nabywania doświadczenia i rosnących potrzeb.

Funkcje wbudowane

Język GLSL dysponuje bogatą biblioteką funkcji, które szadery mogą wywoływać. Nazwy tych funkcji są przeciążone, tj. jednej nazwie zazwyczaj odpowiada wiele funkcji, które spełniają podobną rolę, ale mają różne parametry. Na przykład nazwę sin mają funkcje obliczające sinus, których parametry są typu float, vec2, vec3 lub vec4. Każda z tych funkcji oblicza sinusy, odpowiednio jednej, dwóch, trzech lub czterech liczb — współrzędnych wektora podanego jako parametr, przy czym wartość tej funkcji jest tego samego typu co parametr. Podobnie funkcje dot obliczają iloczyny skalarne wektorów, odpowiednio typu vec2, vec3 lub vec4. Poniżej wymieniam tylko niektóre funkcje wbudowane. Ich pełną listę ze

szczegółowymi opisami można znaleźć w dokumencie [3].

Funkcje elementarne:

`abs`, `sign` — wartość bezwzględna, znak.

`floor`, `trunc`, `round`, `ceil`, `fract` — zaokrąglenia liczb rzeczywistych do liczb całkowitych: w dół, w stronę zera, do najbliższej i w górę, oraz część ułamkowa.

`roundEven` — zaokrąglanie do najbliższej liczby parzystej.

`mod` — reszta z dzielenia, $x - y * \lfloor x/y \rfloor$.

`modf` — część ułamkowa (pierwszego) parametru wejściowego, (drugi) parametr wyjściowy otrzymuje wartość równą części całkowitej.

`min`, `max` — wybór mniejszego i większego argumentu.

`clamp` — element przedziału określonego przez drugi i trzeci parametr, najbliższy pierwszego parametru.

`mix` — interpolacja afiniczna między pierwszym i drugim parametrem.

Funkcje związane z potęgowaniem:

`pow` — oblicza x^y.

`exp`, `exp2` — obliczają e^x, 2^x.

`log`, `log2` — obliczają $\ln x$, $\log_2 x$

`sqrt`, `inversesqrt` — obliczają $\sqrt{x}$, $1/\sqrt{x}$.

Funkcje geometryczne:

`length` — długość wektora.

`distance` — odległość punktów.

`dot` — iloczyn skalarny.

`cross` — iloczyn wektorowy (tylko w $\mathbb{R}^3$).

`normalize` — normalizacja (tj. dzielenie wektora przez jego długość, powstaje wektor jednostkowy).

`faceforward` — pierwszy parametr ze znakiem dobranym na podstawie iloczynu skalarnego wektorów podanych jako drugi i trzeci parametr.

`reflect` — odbicie.

`refract` — załamanie.

Funkcje związane z kątami (trygonometryczne, cyklometryczne itp.) Ich argumenty i wartości są reprezentowane w postaci zmiennopozycyjnej pojedynczej precyzji:

`radians`, `degrees` — mnożą argumenty odpowiednio przez $\frac{\pi}{180}$ oraz $\frac{180}{\pi}$, co jest równoznaczne z przejściami od stopni do radianów i w drugą stronę.

`sin`, `cos`, `tan` — obliczają sinusy, kosinusy i tangensy współrzędnych wektorów podanych jako parametry. Współrzędne te są miarami kątów podanymi w radianach.

`asin`, `acos`, `atan` — obliczają (wyrażone w radianach) miary kątów, których sinusy, kosinusy lub tangensy są współrzędnymi wektorów podanych jako parametry.

`atan` dwuargumentowy — oblicza miary kątów, których tangensy są ilorazami kolejnych współrzędnych pary wektorów podanych jako parametry (odpowiada to dostępnej w C funkcji `atan2`).

`sinh`, `cosh`, `tanh` — obliczają funkcje hiperboliczne, odpowiednio sinus, kosinus i tangens.

`asinh`, `acosh`, `atanh` — obliczają funkcje odwrotne do funkcji hiperbolicznych.

Funkcje macierzowe:

`matrixCompMult` — oblicza macierz iloczynów współczynników dwóch macierzy na odpowiadających sobie miejscach.

`outerProduct` – dla danych wektorów x, y oblicza macierz xy^{T}.

`transpose` — wyznacza macierz transponowaną do danej.

`determinant` — oblicza wyznacznik macierzy kwadratowej.

`inverse` — oblicza odwrotność macierzy kwadratowej nieosobliwej.

Funkcje relacji wektorowych:

`lessThan`, `lessThanEqual`, `greaterThan`, `greaterThanEqual`, `equal`, `notEqual` — porównują odpowiadające sobie współrzędne wektorów danych jako parametry i dostarczają wyniki porównań jako wartość typu wektorowego boolowskiego (`bvec2`, `bvec3` albo `bvec4`).

`any`, `all` — zwracają `true` jeśli odpowiednio co najmniej jeden lub wszystkie współrzędne wektora boolowskiego podanego jako parametr mają wartość `true`, a w przeciwnym razie `false`.

`not` — neguje współrzędne wektora boolowskiego.

Funkcje dla liczb całkowitych:

`uaddCarry`, `usubBorrow` — dodawanie z przeniesieniem i odejmowanie z pożyczką liczb 32-bitowych bez znaku, umożliwiające zaimplementowanie arytmetyki dowolnie wielkich liczb (przechowywanych w wielu zmiennych typu `uint`).

`umulExtended`, `imulExtended` — mnożenie liczb 32-bitowych (bez i ze znakiem), dające wyniki 64-bitowe.

`bitfieldExtract` — wycinają wskazane bity z całkowitych współrzędnych wektora.

`bitfieldInsert` — wstawiają wskazane bity do całkowitych współrzędnych wektora.

`bitfieldReverse` — odwracają kolejność bitów.

`bitCount`, `findLSB`, `findMSB` — znajdują liczbę jedynek oraz pozycję najmniej i najbardziej znaczącej pozycji niezerowego bitu.

Funkcje dla tekstur:

`textureSize` — obliczają wymiary tekstur.

`textureQueryLod`, `textureQueryLevels` — funkcje podające informacje związane z mipmapingiem (będzie opisany przy okazji omawiania aplikacji 2D).

`textureSamples` — liczba próbek tekstury wielopróbkowej.

`texture` — podają wartość tekstury (otrzymaną w wyniku interpolacji i filtrowania tekseli).

`textureProj` — działa jak `texture`, ale parametr jest wektorem współrzędnych jednorodnych punktu w dziedzinie tekstury.

`textureLod` — działa jak `texture`, ale dodatkowy parametr jawnie wskazuje poziom mipmapy.

`textureOffset` — działa jak `texture`, ale argument (punkt w dziedzinie tekstury) jest poddawany dodatkowemu przesunięciu.

`textureProjOffset`, `textureProjLod`, `textureProjLodOffset` — przed obliczeniem wartości tekstury punkt w jej dziedzinie, reprezentowany przez podany wektor współrzędnych jednorodnych, może być poddany dodatkowemu przesunięciu, można też wskazać konkretny poziom mipmapy.

`textureGrad`, `textureGradOffset`, `textureProjGrad`, `textureProjGradOffset` — obliczenie gradientu tekstury, za pomocą ilorazów różnicowych.

`textureGather`, `textureGatherOffset`, `textureGatherOffsets` — obliczają wartości wskazanej składowej tekstury w czterech punktach.

`texelFetch`, `texelFetchOffset` — podają wartość najbliższego teksela, bez dokonywania interpolacji.

Jest też wiele procedur obsługujących liczniki niepodzielne (*atomic counters*) i obrazy (*images*). Na razie ich tu nie opisuję.

Szadery obliczeniowe

GPU składa się z dużej (od kilkudziesięciu do kilku tysięcy) liczby procesorów pracujących równolegle i mających dostęp do dosyć dużej (obecnie zazwyczaj od 300MB do 12GB) pamięci. Choć każdy z tych procesorów jest wolniejszy niż CPU i nie mogą one działać całkowicie niezależnie, masywna równoległość obliczeń prowadzonych przy ich użyciu przekłada się na bardzo dużą moc obliczeniową GPU. Szadery obliczeniowe służą do masywnie równoległego przetwarzania danych, przy czym mogą to być obliczenia wstępne (preprocesing) dla potrzeb syntezy obrazu (jeśli wyniki obliczeń są zapamiętywane w buforach, do których dostęp będą miały szadery wmontowane w potok przetwarzania grafiki), albo mogą to być zupełnie dowolne obliczenia (niekoniecznie związane z grafiką), których wyniki z buforów w pamięci GPU odczyta aplikacja.

Szader obliczeniowy jest wykonywany równolegle jako wiele wątków; wątki te są wykonywane w lokalnych grupach roboczych (*local workgroups*), zorganizowanych w jedno-, dwu- lub trójwymiarową tablicę[14], zwaną globalną grupą roboczą

[14]Tablice jedno- i dwuwymiarowe mają jednoelementowe zakresy niepotrzebnych indeksów.

(*global workgroup*). Lokalna grupa robocza jest jedno-, dwu- lub trójwymiarową tablicą wątków. Każdy wątek otrzymuje swoje wektory indeksów do tych tablic, przypisane przedstawionym niżej zmiennym wbudowanym. Na ich podstawie wątek może pobrać odpowiednie dane, na przykład pewien element tablicy, której wszystkie elementy mają być poddane obróbce, i wykonać obliczenie na tym elemencie, a wynik wpisać do odpowiedniego miejsca tablicy na wyniki.

Zmienne wbudowane szadera obliczeniowego:

`in uvec3 gl_NumWorkGroups;` — wektor $(x_{\mathrm{GWS}}, y_{\mathrm{GWS}}, z_{\mathrm{GWS}})$ z wymiarami globalnej grupy roboczej, określonymi przez parametry wywołania procedury

```
glDispatchCompute ( xgws, ygws, zgws );
```

która uruchamia wątki. Elementami tablicy o takich wymiarach są lokalne grupy robocze.

`const uvec3 gl_WorkGroupSize;` — wektor $(x_{\mathrm{LWS}}, y_{\mathrm{LWS}}, z_{\mathrm{LWS}})$ z wymiarami wielkość lokalnej grupy roboczej, zadeklarowanymi w kodzie szadera, w kwalifikatorze wejścia wyglądającym tak:

```
layout (local_size_x=xlws,local_size_y=ylws,local_size_z=zlws) in;
```

przy czym muszą być podane konkretne liczby (wielkość lokalnej grupy roboczej jest ustalana podczas kompilacji szadera), a ponadto jeśli wymiary y_{LWS} lub z_{LWS} są równe 1, to można napisać krócej

```
layout (local_size_x=xlws,local_size_y=ylws) in;
```

albo

```
layout (local_size_x=xlws) in;
```

`in uvec3 gl_WorkGroupID;` — wektor $(x_{\mathrm{WID}}, y_{\mathrm{WID}}, z_{\mathrm{WID}})$, trójka indeksów globalnej grupy roboczej.

`in uvec3 gl_LocalInvocationID;` — wektor $(x_{\mathrm{LID}}, y_{\mathrm{LID}}, z_{\mathrm{LID}})$, trójka indeksów wątku w lokalnej grupie roboczej.

`in uvec3 gl_GlobalInvocationID;` — wektor $(x_{\mathrm{GID}}, y_{\mathrm{GID}}, z_{\mathrm{GID}})$, trójka indeksów wątku w globalnej grupie roboczej, określona wzorem

$$\begin{bmatrix} x_{\mathrm{GID}} \\ y_{\mathrm{GID}} \\ z_{\mathrm{GID}} \end{bmatrix} = \begin{bmatrix} x_{\mathrm{LWS}}x_{\mathrm{WID}} + x_{\mathrm{LID}} \\ y_{\mathrm{LWS}}y_{\mathrm{WID}} + y_{\mathrm{LID}} \\ z_{\mathrm{LWS}}z_{\mathrm{WID}} + z_{\mathrm{LID}} \end{bmatrix}$$

`in uint gl_LocalInvocationIndex;` — jedna liczba, numer wątku w lokalnej grupie roboczej uszeregowanej w jednowymiarowy ciąg, określona wzorem

$$i = (z_{\mathrm{LID}} y_{\mathrm{LWS}} + y_{\mathrm{LID}}) x_{\mathrm{LWS}} + x_{\mathrm{LID}}.$$

Istnieją limity wielkości lokalnych grup roboczych; możemy je poznać, wykonując instrukcję

```
for ( i = 0; i < 3; i++ )
  gl_GetIntegeri_v ( GL_MAX_COMPUTE_WORK_GROUP_SIZE, i, &d[i] );
```

która wpisze limity dla poszczególnych wymiarów grupy do tablicy `d`. Na komputerze, którego używam pisząc ten fragment książki, podane limity to 1536, 1024 i 64.

Więcej informacji na ten temat podam w dalszych rozdziałach (21 i dalszych), przedstawiając je na przykładach aplikacji wykorzystujących szadery obliczeniowe.

10. Aplikacja pierwsza B

Każdy model matematyczny oświetlenia musi uwzględniać trzy elementy: źródła światła, oświetlaną powierzchnię i obserwatora. Model jest wzorem, do którego trzeba podstawić odpowiednie parametry tych trzech elementów. W tym rozdziale zrealizujemy model najprostszy — oświetlenia tzw. lambertowskiego, w którym przyjmuje się, że oświetlone powierzchnie są matowe[1].

Podstawową rolę we wszystkich modelach oświetlenia powierzchni odgrywa jej wektor normalny, który jest częścią opisu obiektu. Problemem do rozwiązania jest uzyskanie wektora normalnego dla każdego wierzchołka. Można go oczywiście podać jako atrybut wierzchołka. Ale dla bryły wielościennej, jaką jest dwudziestościan, każdy wierzchołek należy do kilku (pięciu) ścian i dlatego musiałby wystąpić wielokrotnie w tablicy wierzchołków — każdy egzemplarz z innym wektorem normalnym, właściwym dla odpowiedniej ściany. Niestety, nie ma możliwości podawania atrybutów *ścian*. Podanie wektora normalnego dla wierzchołka należącego do wielu ścian leżących w różnych płaszczyznach ma sens, jeśli ściany te stanowią przybliżenie fragmentu gładkiej powierzchni zakrzywionej (i w kolejnych aplikacjach będziemy z tego korzystać). Jeśli wektor normalny nie jest atrybutem wierzchołka, to szader wierzchołków nie dysponuje informacją umożliwiającą jego obliczenie dla ściany. Ale taką informacją może dysponować szader geometrii, którego właśnie użyjemy.

Szadery — pierwszy program

Aplikacja będzie rysować wierzchołki i krawędzie figury tak jak poprzednio, natomiast do wyświetlania ścian użyje innego programu szaderów, tak więc będą w użyciu dwa programy; w pierwszym zastosujemy szadery wypróbowane wcześniej (z pewną drobną modyfikacją), a w drugim programie zrealizujemy model oświetlenia. Szader wierzchołków pierwszego programu, pokazany na listingu 10.1, został przystosowany do nowo zdefiniowanego bloku zmiennych jednolitych `TransBlock` opisujących przekształcenia. Blok ten jest wspólny dla obu programów, zatem musi (powinien) być wszędzie taki sam. Do trzech macierzy przekształceń obecnych wcześniej doszła macierz będąca ich iloczynem; dzięki temu przekształcenie wierzchołka wymaga tylko jednego mnożenia wektora przez macierz (linia 18), a nie trzech mnożeń. Za obliczenie iloczynu tych trzech macierzy i umieszczenie go w zmiennej jednolitej `TransBlock.mvpm` odpowiada oczywiście aplikacja.

[1] Model ten wynalazł Johann Heinrich Lambert w r. 1760.

Listing 10.1: Szader wierzchołków pierwszego programu

GLSL

```
#version 420

layout(location=0) in vec4 in_Position;
layout(location=1) in vec4 in_Colour;

out Vertex {
  vec4 Colour;
} Out;

uniform TransBlock {
  mat4 mm, vm, pm, mvpm;
} trb;

void main ( void )
{
  gl_Position = trb.mvpm * in_Position;
  Out.Colour = in_Colour;
} /*main*/
```

Szader fragmentów pierwszego programu jest podany na listingu 7.2; nie ma potrzeby wprowadzać w nim żadnych zmian.

Model oświetlenia i drugi program szaderów

Lambertowski model oświetlenia jest opisany wzorem

$$I = a\sum_{i=0}^{n-1}\left(I_i^{amb} + I_i^{dir}\nu_i|\langle \mathbf{l}_i, \mathbf{n}\rangle|\right), \tag{9.1}$$

w którym poszczególne symbole mają następujące znaczenie: I oznacza intensywność[2] światła odbitego od powierzchni w stronę obserwatora. W świecie fizycznym jest to funkcja długości fali świetlnej $\lambda \in [380\text{nm}, 680\text{nm}]$.
W komputerze jest to wektor w $\mathbb{R}^3$, którego współrzędne RGB określają wartości składowych czerwonej, zielonej i niebieskiej przypisywanych pikselowi (do tego doczepiamy współrzędną A, czyli kanał alfa); wszystkie współrzędne koloru przypisywanego pikselowi powinny mieć wartości z przedziału $[0, 1]$.

Czynnik a (też funkcja zmiennej λ reprezentowana przez wektor w $\mathbb{R}^3$) opisuje zdolność powierzchni do odbijania światła; jest on kolorem podanym jako atrybut

[2] W tym miejscu zaniedbujemy dokładne definicje wielkości występujących tu wielkości, pochodzące z fotometrii.

wierzchołka. Indeks i służy do ponumerowania źródeł światła, o których zakładamy, że są punktowe. Światło do powierzchni może dochodzić z określonego punktu (np. z lampy umieszczonej w rysowanej scenie lub w jej pobliżu) lub z określonego kierunku (czyli z lampy umieszczonej „nieskończenie daleko”, takiej jak Słońce). Wektor jednostkowy $\mathbf{l}_i$ wyznacza kierunek od punktu na powierzchni do źródła światła, natomiast $\mathbf{n}$ oznacza jednostkowy wektor normalny powierzchni. Iloczyn skalarny tych dwóch wektorów, $\langle \mathbf{l}_i, \mathbf{n} \rangle$, jest kosinusem kąta między nimi. Czynnik ν_i odpowiada za widoczność: ma on wartość 1, jeśli obserwator znajduje się po tej samej stronie powierzchni co źródło światła[3] i 0, jeśli po przeciwnej — zakładamy, że powierzchnia jest całkowicie nieprzezroczysta. W bardziej zaawansowanej grafice czynnik ν_i ma wartość 0 także wtedy, gdy między źródłem światła i danym punktem na powierzchni znajduje się jakiś obiekt rzucający cień. Ale na razie nie bierzemy się jeszcze za algorytmy wyznaczania cieni.

Intensywność światła dochodzącego do danego punktu na powierzchni bezpośrednio od źródła światła jest oznaczona symbolem I_i^{dir}. Przyjęte jest założenie, że część światła wysyłanego przez to źródło rozprasza się (w atmosferze i w wielokrotnych odbiciach od różnych obiektów), w związku z czym we wzorze mamy składnik I_i^{amb}, opisujący światło pochodzące z tego samego źródła, ale dochodzące do danego punktu ze *wszystkich* kierunków[4].

Jeśli źródło światła znajduje się bardzo daleko od oświetlanych obiektów, to kierunek $\mathbf{l}_i$ dla wszystkich punktów powierzchni tych obiektów jest taki sam. Ponadto intensywność I_i^{dir} oświetlenia kierunkowego jest stała. Jeśli natomiast odległość między źródłem światła a obiektami jest skończona, to po pierwsze, kierunki $\mathbf{l}_i$ od poszczególnych punktów powierzchni do źródła światła są różne, a ponadto intensywność światła maleje ze wzrostem odległości od źródła. Z zasady zachowania energii wynika, że dla idealnego źródła punktowego intensywność powinna być odwrotnie proporcjonalna do kwadratu odległości, ale w praktyce lepiej sprawdza się wzór

$$I_i^{\mathrm{dir}} = \frac{I_i^{\mathrm{em}}}{at^2 + bt + c}, \tag{9.2}$$

w którym t oznacza odległość, I_i^{em} to moc źródła światła, zaś a, b, c to współczynniki, które trzeba dobrać doświadczalnie (musi być $a > 0$, $b, c \geqslant 0$).

Wiemy już, co trzeba dostarczyć szaderowi fragmentów, aby można było obliczyć

[3] a właściwie płaszczyzny stycznej do powierzchni

[4] W tym najprostszym modelu nie badamy rozkładu kierunkowego oświetlenia, co umożliwiłoby otrzymanie półcieni na obrazie.

kolor piksela, możemy zatem zaprojektować szadery. Szader wierzchołków na listingu 10.2 dokonuje w linii 20 przekształcenia wierzchołka do kostki standardowej, a w linii 23 przepisuje do pola `Colour` struktury wyjściowej atrybut koloru wierzchołka, który będzie dalej wykorzystany jako współczynnik α w modelu oświetlenia. Do dalszych obliczeń jest też potrzebny wektor współrzędnych kartezjańskich wierzchołka w układzie świata. Wynikiem mnożenia przez macierz przekształcenia modelu w linii 21 jest wektor współrzędnych jednorodnych, a współrzędne kartezjańskie obliczamy w linii 22, dzieląc pierwsze trzy współrzędne jednorodne przez współrzędną wagową.

Listing 10.2: Szader wierzchołków drugiego programu

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec4 in_Colour;
5:
6: out Vertex {
7:   vec4 Colour;
8:   vec3 Position;
9: } Out;
10:
11: uniform TransBlock {
12:   mat4 mm, vm, pm, mvpm;
13:   vec4 eyepos;
14: } trb;
15:
16: void main ( void )
17: {
18:   vec4 Pos;
19:
20:   gl_Position = trb.mvpm * in_Position;
21:   Pos = trb.mm * in_Position;
22:   Out.Position = Pos.xyz / Pos.w;
23:   Out.Colour = in_Colour;
24: } /*main*/
```

Zadaniem szadera geometrii, pokazanego na listingu 10.3, jest obliczenie wektora normalnego trójkątnej ściany. Kwalifikator `layout` w linii 3 opisuje dane wejściowe dla szadera; podczas rysowania trójkątów (osobnych lub wchodzących w skład taśmy lub wachlarza — po etapie pobierania wierzchołków to już nie ma znaczenia) szader geometrii zostanie wywołany dla każdego trójkąta i otrzyma dane opisujące wszystkie trzy wierzchołki.

Listing 10.3: Szader geometrii drugiego programu

GLSL

```
1: #version 420
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in Vertex {
7:   vec4 Colour;
8:   vec3 Position;
9: } In[];
10:
11: out NVertex {
12:   vec4 Colour;
13:   vec3 Position;
14:   vec3 Normal;
15: } Out;
16:
17: void main ( void )
18: {
19:   int i;
20:   vec3 v1, v2, nv;
21:
22:   v1 = In[1].Position - In[0].Position;
23:   v2 = In[2].Position - In[0].Position;
24:   nv = normalize ( cross ( v1, v2 ) );
25:   for ( i = 0; i < 3; i++ ) {
26:     gl_Position = gl_in[i].gl_Position;
27:     Out.Position = In[i].Position;
28:     Out.Normal = nv;
29:     Out.Colour = In[i].Colour;
30:     EmitVertex ();
31:   }
32:   EndPrimitive ();
33: } /*main*/
```

Zgodnie z opisem w tym kwalifikatorze dane wejściowe są wierzchołkami trójkąta podanymi w tablicach o długości 3. Zmienna wbudowana o nazwie lokalnej `gl_in` składa się ze struktur `gl_PerVertex`, opisanych w poprzednim rozdziale. Każda z tych struktur zawiera współrzędne jednorodne wierzchołka w polu `gl_Position` typu vec4, przypisane przez szader wierzchołków (w linii 20 na listingu 10.2). Pozostałe pola tych struktur nie są używane.

Druga tablica danych wejściowych składa się ze struktur o nazwie globalnej

`Vertex` i o lokalnej nazwie `In`. Długość tej tablicy nie jest podana jawnie, bo jest ona określona przez wspomniany wyżej kwalifikator. Struktura `Vertex` ma identyczną budowę jak na listingu 10.2, bo musi mieć.

Kwalifikator `layout` w linii 4 opisuje postać wyjścia szadera. Szader geometrii może *zmienić rodzaj* obiektu geometrycznego; rodzaj ten jest określony właśnie przez ten kwalifikator. W naszym przypadku trójkąt ma pozostać trójkątem. Zapis w kwalifikatorze formalnie wygląda tak, że szader geometrii produkuje taśmę trójkątową, w której są najwyżej 3 wierzchołki — czyli jest w niej tylko jeden trójkąt.

Struktura wyjściowa `Out` (o nazwie globalnej `NVertex`) (linie 11–15) *nie jest* tablicą. Szader ma do tej struktury kolejno dla każdego wierzchołka wpisać dane wyjściowe i wywołać procedurę `EmitVertex`, która wprowadza dane z tej struktury do dalszych etapów w potoku przetwarzania grafiki.

Procedura `main` szadera w liniach 22–23 oblicza dwa wektory o kierunkach i długościach boków trójkąta, a następnie w linii 24 oblicza ich iloczyn wektorowy i poddaje go normalizacji. W ten sposób obliczony jest jednostkowy wektor normalny płaszczyzny trójkąta, który zostanie wyprowadzony jako atrybut `Normal` wszystkich trzech wierzchołków. Współrzędne wierzchołków używane w tym obliczeniu są podane w układzie świata i kartezjańskie; obliczył je szader wierzchołków w liniach 21 i 22 na listingu 10.2.

W pętli w liniach 25–31 szader geometrii wpisuje do struktury wyjściowej kolejno położenie, wektor normalny i kolor; położenie (w układzie świata) i kolor są kopiowane z danych wejściowych. Po wpisaniu danych dla każdego wierzchołka jest wywoływana procedura `EmitVertex`, a na zakończenie jest wywoływana procedura `EndPrimitive`, która sygnalizuje zakończenie wyprodukowanej przez szader „taśmy” złożonej z jednego trójkąta.

Uwaga: Choć atrybut `Normal` jest taki sam dla wszystkich wierzchołków, przypisanie `Out.Normal = nv;` trzeba wykonać dla każdego wierzchołka. Procedura `EmitVertex` sprawia, że wartości wszystkich pól struktury `Out` stają się nieokreślone.

Obliczenie koloru oświetlonego punktu powierzchni wykonuje szader fragmentów przedstawiony na listingu 10.4. Dane wejściowe opisujące fragment są obecne w zmiennych wbudowanych opisanych w poprzednim rozdziale (ten szader w ogóle z nich nie korzysta) oraz w strukturze `NVertex` o budowie identycznej ze strukturą `NVertex` opisującą dodatkowe wyjście szadera geometrii.

Listing 10.4: Szader fragmentów drugiego programu

GLSL

```
1: #version 420
2:
3: #define MAX_NLIGHTS 8
4:
5: in NVertex {
6:   vec4 Colour;
7:   vec3 Position;
8:   vec3 Normal;
9: } In;
10:
11: out vec4 out_Colour;
12:
13: uniform TransBlock {
14:   mat4 mm, vm, pm, mvpm;
15:   vec4 eyepos;
16: } trb;
17:
18: struct LSPar {
19:     vec4 ambient;
20:     vec4 direct;
21:     vec4 position;
22:     vec3 attenuation;
23:   };
24:
25: uniform LSBlock {
26:   uint  nls;              /* liczba źródeł światła */
27:   uint  mask;             /* maska włączonych źródeł */
28:   LSPar ls[MAX_NLIGHTS];  /* poszczególne źródła światła */
29: } light;
30:
31: vec3 posDifference ( vec4 p, vec3 pos, out float dist )
32: {
33:   vec3 v;
34:
35:   if ( p.w != 0.0 ) {
36:     v = p.xyz/p.w-pos.xyz;
37:     dist = sqrt ( dot ( v, v ) );
38:   }
39:   else
40:     v = p.xyz;
41:   return normalize ( v );
42: } /*posDifference*/
```

```
43:
44: float attFactor ( vec3 att, float dist )
45: {
46:   return 1.0/(((att.z*dist)+att.y)*dist+att.x);
47: } /*attFactor*/
48:
49: void main ( void )
50: {
51:   vec3  normal, lv, vv;
52:   float d, e, dist;
53:   uint  i, mask;
54:
55:   normal = normalize ( In.Normal );
56:   vv = posDifference ( trb.eyepos, In.Position, dist );
57:   e = dot ( vv, normal );
58:   out_Colour = vec4(0.0);
59:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
60:     if ( (light.mask & mask) != 0 ) {
61:       out_Colour += light.ls[i].ambient * In.Colour;
62:       lv = posDifference ( light.ls[i].position, In.Position, dist );
63:       d = dot ( lv, normal );
64:       if ( e > 0.0 ) {
65:         if ( d > 0.0 ) {
66:           if ( light.ls[i].position.w != 0.0 )
67:             d *= attFactor ( light.ls[i].attenuation, dist );
68:           out_Colour += (d * light.ls[i].direct) * In.Colour;
69:         }
70:       }
71:       else {
72:         if ( d < 0.0 ) {
73:           if ( light.ls[i].position.w != 0.0 )
74:             d *= attFactor ( light.ls[i].attenuation, dist );
75:           out_Colour -= (d * light.ls[i].direct) * In.Colour;
76:         }
77:       }
78:     }
79:     out_Colour.a = 1.0;
80: } /*main*/
```

Obliczony kolor piksela ma być przypisany zmiennej out_Colour. Pozostałe dane potrzebne do tego obliczenia są obecne w dwóch blokach zmiennych jednolitych.

W bloku TransBlock, do którego wcześniej odwoływał się szader wierzchołków, są podane macierze przekształceń (których ten szader nie używa) oraz (w polu

eyepos) współrzędne położenia obserwatora w układzie świata.

W bloku LightBlock znajduje się opis źródeł światła. Pola nls i mask zawierają informację o liczbie zdefiniowanych źródeł światła i o tym, które z tych źródeł są w danej chwili włączone. Pole ls jest tablicą struktur typu LSPar, zdefiniowanego w liniach 18–23. Struktura taka zawiera parametry jednego źródła światła. Długość tablicy jest określona przez makrodefinicję w linii 3. W razie potrzeby długość ta może być zmieniona, ale nie może ona być większa niż 32, ponieważ zmienna light.mask ma dokładnie 32 bity i każde źródło światła jest związane z jednym z nich.

Pole ambient struktury LSPar przechowuje współrzędne wektora I_i^{amb} (wzór (9.1)) opisującego intensywność światła rozproszonego w otoczeniu, pochodzącego od i-tego źródła. Pole direct reprezentuje wektor I_i^{dir}, jeśli źródło światła jest położone bardzo daleko od obiektu, albo wektor I_i^{em} ze wzoru (9.2), jeśli źródło jest w skończonej odległości. Wektory te nie muszą mieć tego samego kierunku, tj. mogą opisywać różne odcienie koloru światła. Rozpraszając się, światło może zmienić odcień, jeśli na przykład obiekt i żarówka oświetlająca go znajdują się w pomieszczeniu, którego ściany zostały pomalowane na fioletowo.

Pole position przechowuje współrzędne jednorodne położenia źródła światła. Jeśli współrzędna wagowa jest równa 0, to źródło to jest położone daleko od obiektu i pierwsze trzy współrzędne określają kierunek, z którego światło dochodzi do wszystkich punktów sceny. Jeśli współrzędna wagowa nie jest zerem, to współrzędne kartezjańskie jego położenia są obliczane przez podzielenie współrzędnych jednorodnych przez współrzędną wagową.

Pole attenuation zawiera współczynniki a, b, c wielomianu w mianowniku we wzorze (9.2). Są one używane w obliczeniach, jeśli źródło światła jest w skończonej odległości od obiektu.

Procedura PosDifference (linie 30–42) oblicza wektor jednostkowy $\boldsymbol{v}$ skierowany od punktu pos do punktu p. Pierwszy z tych punktów znajduje się na oświetlanej powierzchni; podaje się jego współrzędne kartezjańskie. Dla drugiego z tych punktów podaje się współrzędne jednorodne, ponieważ *może* on być „punktem w nieskończoności" i wtedy pierwsze trzy współrzędne jednorodne wyznaczają kierunek i zwrot wektora $\boldsymbol{v}$. Jeśli współrzędna wagowa nie jest zerem, to procedura w linii 36 oblicza współrzędne kartezjańskie punktu p i odejmuje punkty, a potem normalizuje wektor różnicy. Dodatkowo w linii 37 jest obliczana odległość punktów. Odległość ta jest przypisywana do parametru wyjściowego dist.

Procedura `attFactor` (linie 44–47) oblicza (przy użyciu schematu Hornera) wartość wyrażenia $1/(at^2 + bt + c)$ dla liczb a, b, c będących współrzędnymi wektora `att` i liczby t będącej wartością parametru `dist`.

Procedura `main` w linii 55 dokonuje normalizacji wektora `In.Normal`, aby otrzymać jednostkowy wektor $\boldsymbol{n}$ potrzebny we wzorze (9.1). Wprawdzie dla ściany bryły wielościennej, której wszystkie wierzchołki mają podany ten sam wektor jednostkowy, to jest zbędne, ale chcemy, aby szader nadawał się także do tworzenia obrazów powierzchni zakrzywionych, dla których będziemy podawać różne wektory dla różnych wierzchołków ściany. W takim przypadku interpolacja wektorów jednostkowych wytwarza wektory o długościach innych niż 1 i trzeba je normalizować.

W linii 56 obliczany jest wektor jednostkowy $\boldsymbol{v}$ opisujący kierunek od punktu na powierzchni do obserwatora. Punkt na powierzchni jest dany w zmiennej `In.Position` i został otrzymany przez interpolację położeń wierzchołków trójkąta (w układzie świata). Położenie obserwatora jest dane w zmiennej jednolitej `trb.eyepos`. Jeśli obraz jest wykonywany przy użyciu rzutu perspektywicznego, to obserwator jest w skończonej odległości od obiektu i współrzędna wagowa jego położenia jest niezerowa. Jeśli stosowany jest rzut równoległy, to obserwator widzi scenę z ustalonego kierunku. Współrzędna wagowa jego położenia jest równa 0. Aplikacja musi zadbać o to, aby za każdym razem, gdy zmieniana jest macierz przejścia od układu świata do układu obserwatora (przechowywana w zmiennej `trb.vm`) zmienna `trb.eyepos` była odpowiednio uaktualniana. W linii 57 obliczany jest iloczyn skalarny wektora normalnego i wektora $\boldsymbol{v}$, w celu ustalenia, po której stronie płaszczyzny ściany (lub ogólniej, płaszczyzny stycznej do powierzchni obiektu) znajduje się obserwator. Istotny jest *znak* tego iloczynu.

W linii 58 szader inicjalizuje zmienną `out_Colour`, a następnie w pętli oblicza i dodaje kolejne składniki sumy we wzorze (9.1). Liczba źródeł światła jest podana w zmiennej `light.nls`. Dodatkowo maska bitowa `light.mask` określa źródła światła „włączone” w chwili rysowania; źródła wyłączone są pomijane w obliczeniach. W linii 61 do światła odbijanego przez punkt powierzchni dodawany jest składnik aI_i^{amb}. W linii 62 jest obliczany wektor $\boldsymbol{l}_i$, a w linii 63 jego iloczyn skalarny z wektorem normalnym $\boldsymbol{n}$.

Instrukcje warunkowe w liniach 64, 65 i 72 mają na celu ustalenie, czy źródło światła i obserwator znajdują się po tej samej stronie płaszczyzny ściany. Polega to na zbadaniu, czy iloczyny skalarne $\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle$ oraz (obliczony wcześniej) $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$, gdzie wektor $\boldsymbol{v}$ wyznacza kierunek do obserwatora, mają ten sam znak.

Zauważmy, że zwrot wektora $\boldsymbol{n}$ nie ma znaczenia. Jeśli obserwator widzi oświetloną stronę powierzchni, zależnie od potrzeb (sprawdzane to jest w liniach 66 i 73) jest obliczany czynnik osłabienia światła ze wzrostem odległości od jego źródła. Następnie składnik $aI_i^{\mathrm{dir}}|\langle \boldsymbol{l}_i, \boldsymbol{n}\rangle|$ jest dodawany do światła odbijanego przez punkt powierzchni; zwracam uwagę na operatory „+=" i „-=" w liniach 68 i 75, które efektywnie produkują wartość bezwzględną.

Cztery procedury pomocnicze

Napisanie programu korzystającego z wielu bloków zmiennych jednolitych i złożonego z wielu niezależnie pisanych części wiąże się z pewnym problemem porządkowym: chcemy zapewnić, aby bloki wykorzystywane w różnych celach były przywiązywane do różnych punktów dowiązania. Twarde zakodowanie numerów bloków w programie w C stwarza ten kłopot, że trudno jest zadbać o zachowanie porządku, gdy część programu przenosi się do innego programu, albo gdy tworzy się bibliotekę przeznaczoną do wykorzystania w wielu aplikacjach. Od wpisywania numerów w instrukcjach lepsze jest zdefiniowanie nazw używanych punktów (przy użyciu `#define`), ale to też jest półśrodek. Ale jeszcze lepszym rozwiązaniem jest zapamiętanie numerów punktów dowiązania w zmiennych zadeklarowanych w aplikacji. Numery te może generować procedura `NewUniformBindingPoint` przedstawiona na listingu 10.5 (najlepiej dopisać ją i pozostałe trzy procedury do pliku `utilities.c`). Za każdym kolejnym wywołaniem procedura ta podaje kolejny numer.

Procedura `glGetIntegerv` wywołana w linii 7 podaje maksymalną liczbę punktów dowiązania w celu `GL_UNIFORM_BUFFER`[5]; w razie, gdyby procedura `NewUniformBindingPoint` została wywołana więcej razy, aplikacja zostanie zatrzymana.

Uwaga: Jeśli aplikacja korzysta z więcej niż jednego kontekstu OpenGL-a, to należy utworzyć osobny licznik wykorzystanych numerów dla każdego kontekstu[6].

Ponieważ uzyskiwanie podstawowych informacji na temat pól w każdym bloku zmiennych jednolitych wykonuje się tak samo, przydaje się do tego procedura `GetAccessToUniformBlock`. Jej parametry wejściowe to: `prog` — identyfikator programu szaderów, `n` — liczba pól, oraz `names` — tablica $n+1$ napisów, z których pierwszy jest nazwą globalną bloku, a pozostałe są nazwami pól

[5]Specyfikacja OpenGL 4 gwarantuje dostępność co najmniej 36 punktów dowiązania, w komputerze, na którym uruchamiałem aplikacje do tej książki, można korzystać z 84 punktów.

[6]Przypomnijmy, że FreeGLUT domyślnie tworzy nowy kontekst OpenGL-a dla każdego okna i podokna.

Listing 10.5: Cztery procedury pomocnicze

C

```
1: static GLint  max_uniform_bp = 0;
2: static GLuint next_uniform_bp = 0;
3:
4: GLuint NewUniformBindingPoint ( void )
5: {
6:   if ( !max_uniform_bp )
7:     glGetIntegerv ( GL_MAX_UNIFORM_BUFFER_BINDINGS, &max_uniform_bp );
8:   if ( next_uniform_bp < max_uniform_bp )
9:     return next_uniform_bp ++;
10:  else
11:    ExitOnError ( "NewUniformBindingPoint" );
12: } /*NewUniformBindingPoint*/
13:
14: void GetAccessToUniformBlock ( GLuint prog, int n, const GLchar **names,
15:                               GLuint *ind, GLint *size, GLint *ofs,
16:                               GLuint *bpoint )
17: {
18:   GLuint ufi[32];
19:
20:   *ind = glGetUniformBlockIndex ( prog, names[0] );
21:   glGetActiveUniformBlockiv ( prog, *ind,
22:                              GL_UNIFORM_BLOCK_DATA_SIZE, size );
23:   if ( n > 0 ) {
24:     glGetUniformIndices ( prog, n, &names[1], ufi );
25:     glGetActiveUniformsiv ( prog, n, ufi, GL_UNIFORM_OFFSET, ofs );
26:   }
27:   *bpoint = NewUniformBindingPoint ();
28:   glUniformBlockBinding ( prog, *ind, *bpoint );
29:   ExitIfGLError ( "GetAccessToUniformBlock" );
30: } /*GetAccessToUniformBlock*/
31:
32: GLuint NewUniformBlockObject ( GLint size, GLuint bp )
33: {
34:   GLuint buf;
35:
36:   glGenBuffers ( 1, &buf );
37:   glBindBufferBase ( GL_UNIFORM_BUFFER, bp, buf );
38:   glBufferData ( GL_UNIFORM_BUFFER, size, NULL, GL_DYNAMIC_DRAW );
39:   ExitIfGLError ( "NewUniformBlockObject");
40:   return buf;
41: } /*NewUniformBlockObject*/
42:
```

```
43: void AttachUniformBlockToBP ( GLuint prog, const GLchar *name, GLuint bp )
44: {
45:   GLuint ind;
46:
47:   ind = glGetUniformBlockIndex ( prog, name );
48:   glUniformBlockBinding ( prog, ind, bp );
49:   ExitIfGLError ( "AttachUniformBlockToBP" );
50: } /*AttachUniformBlockToBP*/
```

w bloku. Zmiennej *ind procedura przypisuje indeks bloku, zmienna *size otrzymuje wartość równą wielkości bloku w bajtach, do tablicy ofs wpisywane są przesunięcia w bajtach poszczególnych pól od początku bloku. Wreszcie, wartość przypisana zmiennej *bpoint to numer punktu dowiązania wygenerowany przez procedurę NewUniformBindingPoint.

Procedura NewUniformBlockObject alokuje bufor, nadaje mu podaną wielkość i przywiązuje jako UBO do podanego punktu dowiązania.

Zadaniem procedury AttachUniformBlockToBP jest przywiązanie zdefiniowanego w programie prog bloku zmiennych jednolitych o nazwie name do punktu dowiązania bp. Założenie jest takie, że blok ten jest zdefiniowany także w innym programie i potrzebne informacje o nim (przesunięcia pól) zostały wcześniej uzyskane przez wywołaną dla tego innego programu procedurę GetAccessToUniformBlock, która dokonała także alokacji numeru punktu dowiązania.

Aplikacja pierwsza B

Listing 10.6 przedstawia definicje typów i dodatkowe zmienne używane do oświetlenia obiektu. Maksymalna liczba źródeł światła określona w linii 1 może zostać zmieniona, ale nie może być większa niż 32 z wcześniej podanych powodów. Zmiana tej makrodefinicji *musi* być wykonana jednocześnie ze zmianą linii 3 na listingu 10.4, bo jak nie, to będzie źle.

Struktura LSPar zawiera parametry źródła światła, opisane wcześniej. Struktura LightBl zawiera liczbę źródeł światła, maskę włączonych źródeł światła i tablicę struktur LSPar, opisujących poszczególne źródła światła.

Uwaga: Choć struktura LightBl jest zbudowana tak samo, jak blok zmiennych jednolitych LSBlock szadera na listingu 10.4, *nie można* liczyć na to, że poszczególne pola obu struktur mają takie same przesunięcia od początku

struktury[7] i dlatego nie wolno przesyłać danych do UBO „hurtem”; trzeba przesyłać dane pracowicie po jednym polu.

Listing 10.6: Struktury danych i zmienne związane z oświetleniem

C

```
1: #define MAX_NLIGHTS 8
2:
3: typedef struct {
4:           GLfloat ambient[4];
5:           GLfloat direct[4];
6:           GLfloat position[4];
7:           GLfloat attenuation[3];
8:         } LSPar;
9:
10: typedef struct LightBl {
11:           GLuint nls, mask;
12:           LSPar  ls[MAX_NLIGHTS];
13:         } LightBl;
14:
15: typedef struct TransBl {
16:           GLfloat mm[16], vm[16], pm[16];
17:           GLfloat eyepos[4];
18:         } TransBl;
19:
20: GLuint  lsbi, lsbuf, lsbbp;
21: GLint   lsbsize, lsbofs[7];
22: LightBl light;
23: TransBl trans;
```

Zmienne `lsbi`, `lsbuf` i `lsbbp` służą do przechowywania indeksu bloku zmiennych jednolitych `LSBlock`, identyfikatora UBO dla tego bloku i numeru jego punktu dowiązania. W zmiennej `lsbsize` będzie umieszczona wielkość (w bajtach) tego bloku. W tablicy `lsbofs` zostaną umieszczone przesunięcia pól bloku `LSBlock`. Zmienna `light` zawiera informacje o światłach. Będzie w niej przechowywana kopia informacji obecnej w bloku zmiennych jednolitych `LSBlock`.

W zmiennej `trans` będą przechowywane trzy macierze przekształceń, które służą do przejścia od układu modelu do kostki standardowej. We wcześniejszych

[7] Kompilator GLSL ma swoje zasady przydzielania adresów pól struktury. Istnieją kwalifikatory układu zmiennych strukturalnych (np. często stosowany dla UBO kwalifikator `layout(std140)`), które wręcz wykluczają takie same przesunięcia pól struktury w językach GLSL i C. Na podstawie reguł układu określonego przez ten kwalifikator można *obliczyć* przesunięcia pól względem początku struktury, zamiast wywoływać procedurę `glGetActiveUniformsiv`.

wersjach aplikacji wystarczyło je trzymać tylko w pamięci GPU. Ponieważ obecna wersja oblicza iloczyn tych macierzy, są one potrzebne w pamięci CPU, gdy dowolna z nich ulega zmianie. Ponadto w polu eyepos przechowywane jest położenie obserwatora (tj. współrzędne jednorodne początku układu współrzędnych obserwatora w układzie świata). Będzie ono potrzebne w aplikacji pierwszej C.

Listing 10.7: Procedura LoadMyShaders

C

```
1: void LoadMyShaders ( void )
2: {
3:   static const char *filename[] =
4:     { "app1b0.glsl.vert", "app1b0.glsl.frag",
5:       "app1b1.glsl.vert", "app1b1.glsl.geom", "app1b1.glsl.frag" };
6:   static const GLchar *UTBNames[] =
7:     { "TransBlock", "TransBlock.mm", "TransBlock.vm", "TransBlock.pm",
8:       "TransBlock.mvpm", "TransBlock.eyepos" };
9:   static const GLchar *ULSNames[] =
10:    { "LSBlock", "LSBlock.nls", "LSBlock.mask", LSBlock.ls[0].ambient",
11:      "LSBlock.ls[0].direct", "LSBlock.ls[0].position",
12:      "LSBlock.ls[0].attenuation", "LSBlock.ls[1].ambient" };
13:
14:  shader_id[0] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[0] );
15:  shader_id[1] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1,  &filename[1]);
16:  program_id[0] = LinkShaderProgram ( 2, &shader_id[0] );
17:  shader_id[2] = CompileShaderFiles ( GL_VERTEX_SHADER, 1, &filename[2] );
18:  shader_id[3] = CompileShaderFiles ( GL_GEOMETRY_SHADER, 1, &filename[3] );
19:  shader_id[4] = CompileShaderFiles ( GL_FRAGMENT_SHADER, 1, &filename[4] );
20:  program_id[1] = LinkShaderProgram ( 3, &shader_id[2] );
21:  GetAccessToUniformBlock ( program_id[1], 5, &UTBNames[0],
22:                            &trbi, &trbsize, trbofs, &trbbp );
23:  GetAccessToUniformBlock ( program_id[1], 7, &ULSNames[0],
24:                            &lsbi, &lsbsize, lsbofs, &lsbbp );
25:  AttachUniformBlockToBP ( program_id[0], UTBnames[0], trbbp );
26:  trbuf = NewUniformBlockObject ( trbsize, trbbp );
27:  lsbuf = NewUniformBlockObject ( lsbsize, lsbbp );
28:  ExitIfGLError ( ŁoadMyShaders" );
29: } /*LoadMyShaders*/
```

Listing 10.7 przedstawia procedurę ładowania szaderów. Zmiany w porównaniu z procedurą z listingu 7.3 polegają na modyfikacji starych i dodaniu nowych napisów — nazw plików z szaderami i nazw nowych zmiennych jednolitych. Sposób kompilowania i łączenia pierwszego programu szaderów nie uległ zmianie.

Drugi program jest kompilowany w podobny sposób, jest w nim tylko o jeden szader więcej. Zamiast instrukcji w liniach 17–22 na listingu 7.3, w obecnej wersji jest wywołanie procedury `GetAccessToUniformBlock` z listingu 10.5. Ta sama procedura jest wywołana w linii 23 w celu uzyskania dostępu do pól bloku `LSBlock` z parametrami źródeł światła. Instrukcje w liniach 26 i 27 tworzą UBO dla bloków `TransBlock` i `LSBlock` i przywiązują je (na stałe w tej aplikacji) do odpowiednich punktów dowiązania.

Listing 10.8: Inicjalizacja i sprzątanie

```c
1: void InitMyObject ( void )
2: {
3:   TimerInit ();
4:   memset ( &trans, 0, sizeof(TransBl) );
5:   memset ( &light, 0, sizeof(LightBl) );
6:   SetupModelMatrix ( model_rot_axis, model_rot_angle );
7:   InitViewMatrix ();
8:   ConstructIcosahedronVAO ();
9:   InitLights ();
10: } /*InitMyObject*/
11:
12: void InitLights ( void )
13: {
14:   GLfloat amb0[4] = { 0.2, 0.2, 0.3, 1.0 };
15:   GLfloat dif0[4] = { 0.8, 0.8, 0.8, 1.0 };
16:   GLfloat pos0[4] = { 0.0, 1.0, 1.0, 0.0 };
17:   GLfloat atn0[3] = { 1.0, 0.0, 0.0 };
18:
19:   SetLightAmbient ( 0, amb0 );
20:   SetLightDiffuse ( 0, dif0 );
21:   SetLightPosition ( 0, pos0 );
22:   SetLightAttenuation ( 0, atn0 );
23:   SetLightOnOff ( 0, true );
24: } /*InitLights*/
25:
26: void Cleanup ( void )
27: {
28:   int i;
29:
30:   glUseProgram ( 0 );
31:   for ( i = 0; i < 5; i++ )
32:     glDeleteShader ( shader_id[i] );
33:   glDeleteProgram ( program_id[0] );
```

```
34:   glDeleteProgram ( program_id[1] );
35:   glDeleteBuffers ( 1, &trbuf );
36:   glDeleteBuffers ( 1, &lsbuf );
37:   glDeleteVertexArrays ( 1, &icos_vao );
38:   glDeleteBuffers ( 3, icos_vbo );
39:   ExitIfGLError ( "Cleanup" );
40:   glutDestroyWindow ( WindowHandle );
41: } /*Cleanup*/
```

W procedurach inicjalizacji pokazanych na listingu 10.8 są dopisane instrukcje inicjalizacji świateł. Dodatkowo przed wpisaniem pierwszych nietrywialnych danych zmienne `trans` i `light` są wypełniane zerami. Tak trzeba. Do sprzątania doszło kasowanie nowych szaderów, drugiego programu i UBO źródeł światła.

Procedura `InitLights` przesyła dane opisujące *jedno* źródło światła, o numerze 0, położone daleko od obiektu, i włącza to światło. Procedury przesyłające do UBO parametry źródeł światła są podane na listingu 10.11 i opisane dalej.

Na listingu 10.9 są pokazane modyfikacje mające na celu obsługę zmian przekształceń wierzchołków. Każda procedura, która zmienia dowolną z trzech macierzy przekształceń musi wpisać współczynniki tej macierzy do bufora w pamięci GPU *oraz* do odpowiedniego pola zmiennej `trans`, a następnie wywołać procedurę `SetupMVPMatrix`, która oblicza iloczyn tych trzech macierzy i przesyła go do zmiennej jednolitej `TransBlock.mvpm`.

Listing 10.9: Procedury przetwarzania przekształceń

C

```
1: void SetupMVPMatrix ( void )
2: {
3:   GLfloat m[16], mvp[16];
4:
5:   M4x4Multf ( m, trans.vm, trans.mm );
6:   M4x4Multf ( mvp, trans.pm, m );
7:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
8:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*sizeof(GLfloat), mvp );
9:   ExitIfGLError ( "SetupNVPMatrix" );
10: } /*SetupMVPMatrix*/
11:
12: void SetupModelMatrix ( float axis[3], float angle )
13: {
14:   M4x4RotateVf (  trans.mm, axis[0], axis[1], axis[2], angle );
15:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
```

```
16:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0],
17:                     16*sizeof(GLfloat), trans.mm );
18:   ExitIfGLError ( "SetupModelMatrix" );
19:   SetupMVPMatrix ();
20: } /*SetupModelMatrix*/
21:
22: void InitViewMatrix ( void )
23: {
24:   M4x4Translatef (  trans.vm, -viewer_pos0[0], -viewer_pos0[1],
25:                    -viewer_pos0[2] );
26:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
27:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1],
28:                     16*sizeof(GLfloat), trans.vm );
29:   memcpy ( trans.eyepos, viewer_pos0, 4*sizeof(GLfloat) );
30:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4],
31:                     4*sizeof(GLfloat), trans.eyepos );
32:   ExitIfGLError ( "InitViewMatrix" );
33:   SetupMVPMatrix ();
34: } /*InitViewMatrix*/
35:
36: void RotateViewer ( int delta_xi, int delta_eta )
37: {
38: ........ /* instrukcje bez zmian */
39:   M4x4RotateVf ( rm, viewer_rvec[0], viewer_rvec[1], viewer_rvec[2],
40:                  -viewer_rangle );
41:   M4x4Multf (  trans.vm, tm, rm );
42:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
43:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1],
44:                     16*sizeof(GLfloat), trans.vm );
45:   M4x4Transposef ( tm, rm );
46:   M4x4MultMVf ( trans.eyepos, tm, viewer_pos0 );
47:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4],
48:                     4*sizeof(GLfloat), trans.eyepos );
49:   ExitIfGLError ( "RotateViewer" );
50:   SetupMVPMatrix ();
51: } /*RotateViewer*/
52:
53: void ReshapeFunc ( int width, int height )
54: {
55: ........ /* instrukcje bez zmian */
56:   ExitIfGLError ( "ReshapeFunc" );
57:   SetupMVPMatrix ();
58: } /*ReshapeFunc*/
```

Procedury InitViewMatrix i RotateViewer, z których pierwsza nadaje początkowe położenie obserwatora, a druga je zmienia, muszą obliczyć i przesłać położenie obserwatora do pola TransBlock.eyepos. Współrzędne położenia początkowego są współrzędnymi wektora viewer_pos0, które wystarczy tylko skopiować w odpowiednie miejsca (linie 29–31). Przemieszczenie obserwatora realizowane przez procedurę RotateViewer jest obrotem wokół osi przechodzącej przez początek układu współrzędnych świata. Macierz obrotu jest ortogonalna, zatem jej odwrotność jest transpozycją macierzy obrotu układu świata względem układu obserwatora (którego złożenie z odpowiednim przesunięciem jest przejściem od układu świata do ukądu obserwatora). W linii 45 obliczana jest ta transpozycja, a w linii 46 jest obliczany iloczyn, który jest wektorem współrzędnych jednorodnych położenia obserwatora w układzie świata.
W liniach 47–48 jest on przesyłany do UBO.

Trzeba jeszcze zmienić procedurę rysowania dwudziestościanu i procedurę obsługi klawiatury, w sposób pokazany na listingu 10.10. Dodatkowy parametr procedury DrawIcosahedron wybiera sposób rysowania ścian dwudziestościanu — przy użyciu pierwszego programu (czyli tak, jak wcześniej), czy przy użyciu drugiego programu, z oświetleniem. Krawędzie i wierzchołki rysujemy przy użyciu pierwszego programu, bo drugi jest dostosowany tylko do trójkątów.

Procedura KeyboardFunc po napisaniu znaku 'L' albo 'l' wywołuje procedurę ToggleLight, która zmienia wartość zmiennej globalnej enlight i wnosi (za pośrednictwem FreeGLUTa) postulat wykonania nowego obrazu w oknie.

Listing 10.10: Procedury rysowania

C

```
1: void DrawIcosahedron ( int opt, char enlight )
2: {
3:   glBindVertexArray ( icos_vao );
4:   switch ( opt ) {
5: case 0:    /* wierzchołki */
6:     glUseProgram ( program_id[0] );
7: ........ /* instrukcje bez zmian */
8:     break;
9: case 1:    /* krawędzie */
10:     glUseProgram ( program_id[0] );
11: ........ /* instrukcje bez zmian */
12:     break;
13: default:   /* sciany */
14:     glUseProgram ( program_id[enlight ? 1 : 0] );
15: ........ /* instrukcje bez zmian */
16:     break;
```

```
17:   }
18: } /*DrawIcosahedron*/
19:
20: char enlight = true;
21:
22: void ToggleLight ( void )
23: {
24:   enlight = !enlight;
25:   glutPostWindowRedisplay ( WindowHandle );
26: } /*ToggleLight*/
27:
28: void KeyboardFunc ( unsigned char key, int x, int y )
29: {
30:   switch ( key ) {
31: ........ /* instrukcje bez zmian */
32: case 'L':  case 'l':
33:     ToggleLight ();
34:     break;
35: default:              /* ignorujemy wszystkie inne klawisze */
36:     break;
37:   }
38: } /*KeyboardFunc*/
```

Procedury obsługi świateł

Na listingu 10.11 są pokazane procedury, które przesyłają do UBO bloku LSBlock parametry źródeł światła. Kopie tych parametrów są też przechowywane w zmiennej light. Pierwszy parametr każdej z tych procedur jest numerem źródła światła, od zera do liczby o 1 mniejszej od treści makrodefinicji MAX_NLIGHTS.

Drugi parametr pierwszych czterech procedur jest wektorowym atrybutem źródła światła, opisującym odpowiednio intensywność światła rozproszonego I_i^{amb}, intensywność światła dochodzącego bezpośrednio I_i^{dir} (albo intensywność światła emitowanego I_i^{em}), wektor współrzędnych jednorodnych położenia oraz wektor współczynników a, b, c występujących we wzorze (9.2).

Zwróćmy uwagę na sposób obliczania przesunięcia miejsca, do którego należy przesłać dane, względem początku bufora. Procedura GetAccessToUniformBlock wywołana w linii 23–24 na listingu 10.5 wpisała do kolejnych elementów tablicy lsbofs przesunięcia pól nls, mask, ls[0].ambient, ls[0].direct, ls[0].position, ls[0].attenuation i ls[1].ambient. Różnica przesunięć pól ls[1].ambient i ls[0].ambient jest rozmiarem (w bajtach) miejsca zajmowanego przez każdy element tablicy ls, tj. strukturę LSPar. Zatem aby obliczyć

przesunięcie dowolnego pola l-tego elementu tej tablicy, należy do przesunięcia odpowiedniego pola w elemencie zerowym dodać iloczyn indeksu l elementu i różnicy `lsbofs[6]-lsbofs[2]`. To obliczenie wykonują instrukcje w liniach 8, 21, 34 i 47.

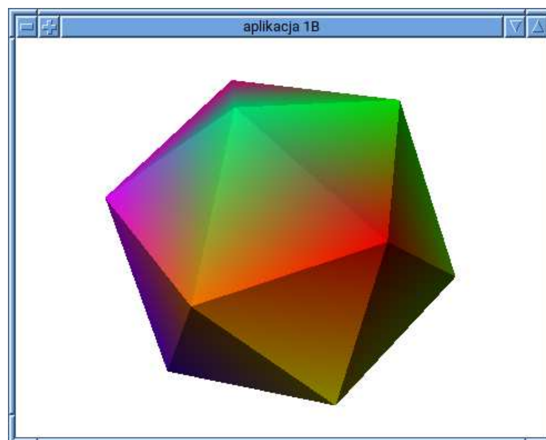
Listing 10.11: Procedury obsługi świateł

C

```
1: void SetLightAmbient ( int l, GLfloat amb[4] )
2: {
3:   GLint ofs;
4:
5:   if ( l < 0 || l >= MAX_NLIGHTS )
6:     return;
7:   memcpy ( light.ls[l].ambient, amb, 4*sizeof(GLfloat) );
8:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[2];
9:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
10:  glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 4*sizeof(GLfloat), amb );
11:  ExitIfGLError ( "SetLightAmbient" );
12: } /*SetLightAmbient*/
13:
14: void SetLightDirect ( int l, GLfloat dir[4] )
15: {
16:   GLint ofs;
17:
18:   if ( l < 0 || l >= MAX_NLIGHTS )
19:     return;
20:   memcpy ( light.ls[l].direct, dir, 4*sizeof(GLfloat) );
21:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[3];
22:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
23:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 4*sizeof(GLfloat), dir );
24:   ExitIfGLError ( "SetLightDirect" );
25: } /*SetLightDirect*/
26:
27: void SetLightPosition ( int l, GLfloat pos[4] )
28: {
29:   GLint ofs;
30:
31:   if ( l < 0 || l >= MAX_NLIGHTS )
32:     return;
33:   memcpy ( light.ls[l].position, pos, 4*sizeof(GLfloat) );
34:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[4];
35:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
36:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 4*sizeof(GLfloat), pos );
37:   ExitIfGLError ( "SetLightPosition" );
```

```
38: } /*SetLightPosition*/
39:
40: void SetLightAttenuation ( int l, GLfloat atn[3] )
41: {
42:   GLint ofs;
43:
44:   if ( l < 0 || l >= MAX_NLIGHTS )
45:     return;
46:   memcpy ( light.ls[l].attenuation, atn, 3*sizeof(GLfloat) );
47:   ofs = l*(lsbofs[6]-lsbofs[2]) + lsbofs[5];
48:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
49:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 3*sizeof(GLfloat), atn );
50:   ExitIfGLError ( "SetLightAttenuation" );
51: } /*SetLightAttenuation*/
52:
53: void SetLightOnOff ( int l, char on )
54: {
55:   GLuint mask;
56:
57:   if ( l < 0 || l >= MAX_NLIGHTS )
58:     return;
59:   mask = 0x01 << l;
60:   if ( on ) {
61:     light.mask |= mask;
62:     if ( l >= light.nls )
63:       light.nls = l+1;
64:   }
65:   else {
66:     light.mask &= ~mask;
67:     for ( mask = 0x01 << (light.nls-1); mask; mask >>= 1 ) {
68:       if ( light.mask & mask )
69:         break;
70:       else
71:         light.nls --;
72:     }
73:   }
74:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
75:   glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[0], sizeof(GLuint),
76:                     &light.nls );
77:   glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[1], sizeof(GLuint),
78:                     &light.mask );
79:   ExitIfGLError ( "SetLightOnOff" );
80: } /*SetLightOnOff*/
```

Procedura SetLightOnOff ustawia w polu mask bit odpowiadający wskazanemu źródłu światła — wartość false (0) oznacza, że światło jest wyłączone, a true (1), że włączone. Dodatkowo procedura oblicza numer ostatniego włączonego światła i nadaje polu nls taką wartość, aby ostatni przebieg pętli szadera fragmentów odpowiadał ostatniemu włączonemu światłu.



Rysunek 10.1: Okno aplikacji pierwszej B

Ćwiczenia

1. Włącz dodatkowe źródła światła, położone w innych miejscach, i poeksperymentuj z ich położeniami i intensywnościami.
2. Oprogramuj interakcyjne zmienianie położeń źródeł światła.

Uzupełnienia

W opisanej tu aplikacji kolory pikseli są obliczane przez szader fragmentów, który zawiera pełną implementację modelu oświetlenia. Możliwe jest też obliczenie koloru wierzchołków przez jeden z szaderów części przedniej potoku przetwarzania grafiki — dla brył wielościennych może to robić szader geometrii, który przetwarza wierzchołki trójkąta i dysponuje wektorem normalnym płaszczyzny ściany. Szader geometrii mógłby obliczyć kolory wszystkich wierzchołków trójkąta; w procesie rasteryzacji kolory te zostałyby poddane interpolacji i zadaniem szadera fragmentów byłoby tylko przepisanie koloru fragmentu z wejścia na wyjście[8]. Ponieważ liczba fragmentów jest od dwóch do pięciu rzędów wielkości większa od liczby wierzchołków, takie rozwiązanie, zwane cieniowaniem Gourauda, działałoby znacznie szybciej.

[8] Można by tu użyć szadera fragmentów z listingu 7.2.

Zastosowana w aplikacji metoda, w której wektor normalny jest interpolowany między wierzchołkami trójkąta, normalizowany i podstawiany do modelu oświetlenia dla każdego fragmentu (tj. piksela) ma nazwę cieniowania Phonga. Jakość obrazów otrzymanych przy użyciu obu metod cieniowania, dla lambertowskiego modelu oświetlenia, niewiele się różni. Ale wykonywanie pełnych obliczeń oświetlenia dla każdego fragmentu jest koniecznością, jeśli używamy bardziej skomplikowanego modelu oświetlenia (w którym powierzchnie nie muszą być matowe) lub gdy na powierzchnię nakładamy teksturę. Tym zajmiemy się w aplikacji drugiej i dalszych.

11. Aplikacja pierwsza C

Do pierwszej aplikacji dodamy pisanie tekstu; niech obracanie obserwatora wokół obiektu powoduje wyświetlenie na tle obrazu współrzędnych położenia obserwatora (czyli początku układu obserwatora) w układzie świata. Wyświetlanie tekstu jest o tyle trudne, że w większości krojów pisma poszczególne znaki mają różne szerokości, a ponadto zajmowane przez nie pola mogą (wskutek kerningu) nachodzić na siebie. Tu ograniczymy się do wyświetlania napisów złożonych ze znaków o ustalonej szerokości, zostawiając składanie bardziej skomplikowanego tekstu jako temat dla przyszłych badań.

Reprezentacja fontów

W kroju pisma (foncie) o stałej szerokości każdy znak zajmuje prostokąt o tych samych wymiarach, powiedzmy $w \times h$ pikseli; liczby w i h są naturalne. Założymy, że $1 \leqslant w \leqslant 32$. Każdy wiersz rastra w polu znaku chcemy zakodować w jednym, dwóch lub czterech bajtach, to znaczy w jednej liczbie typu `GLubyte`, `GLushort` albo `GLuint`, przy czym wybieramy typ zajmujący najmniejszą wystarczającą ilość miejsca. Dla każdego znaku podamy h liczb tego typu, zaczynając od wiersza na samej górze.

```
0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 1 1 1 1 1 1 0  0x07E
0 0 1 1 0 0 0 1 1 0  0x0C6
0 1 1 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 1 1 0  0x186
0 1 1 0 0 0 0 1 1 0  0x186
0 0 1 1 0 0 0 1 1 0  0x0C6
0 0 0 1 1 1 1 1 1 0  0x07E
0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 1 1 0  0x006
0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0  0x000
0 0 0 0 0 0 0 0 0 0  0x000
```

Rysunek 11.1: Reprezentacja jednej litery

Przykład jest pokazany na rysunku 11.1: piksele w każdym wierszu zamieniamy na zera i jedynki, których kolejność w wierszu jest odwrócona[1]. Otrzymane zera i jedynki możemy zgrupować po 4 i każdą taką czwórkę zamienić na jedną cyfrę szesnastkową. Dla każdego znaku zajmującego pole o wymiarach 10×18 podajemy 18 liczb typu `GLushort`.

[1]Powód odwrócenia jest taki, że wygodniejsza jest reprezentacja fontu, w której najmniej znaczący bit odpowiada pikselowi z lewej strony, a zapis pozycyjny umieszcza najmniej znaczące cyfry z prawej.

Chcemy, aby piksele wzorców znaków odpowiadały pikselom w oknie. W tym celu skonstruujemy macierz rzutowania zgodnie z przepisem na końcu rozdziału 6.

Przed wyświetlaniem tekstu należy umieścić w pamięci GPU (w odpowiednich zmiennych jednolitych) font (czyli zestaw wzorców znaków z potrzebnymi informacjami dodatkowymi), kolory znaków i tła oraz sam napis, tj. ciąg kodów ASCII kolejnych znaków.

Algorytm wyświetlania tekstu w zarysie wygląda tak: długość (tj. liczbę znaków) napisu pomnożymy przez szerokość znaku. Należy narysować prostokąt, zwany dalej obszarem tekstu, o otrzymanej w ten sposób szerokości i o wysokości znaku. *Dolny lewy* wierzchołek obszaru tekstu ma współrzędne ξ', η' podane w układzie okna[2]. Zadaniem szadera fragmentów jest określenie, czy przetwarzany piksel obszaru tekstu ma otrzymać kolor znaku albo kolor tła, czy też ma zachować niezmienioną wartość.

Możemy chcieć wyświetlić znaki w ustalonym kolorze na tle w innym ustalonym kolorze, czyli przypisać jeden albo drugi kolor każdemu pikselowi obszaru tekstu. Inna możliwość to przypisanie koloru tylko tym pikselom, które należą do obszaru znaku (a więc tło miałoby zostać niezmienione), albo tylko tym pikselom, które należą do tła. Aby wybrać jedną z tych możliwości, użyjemy kanału alfa (tj. składowej A wektora RGBA) koloru znaków lub tła. Jeśli wartość tej składowej jest zerem, to odpowiedni piksel zostawimy niezmieniony, a w przeciwnym razie przypiszemy mu odpowiedni kolor.

Szadery

Program szaderów zbudujemy z szadera wierzchołków podanego na listingu 4.3 i z szadera fragmentów na listingu 11.1. Oba szadery odwołują się do tego samego bloku zmiennych jednolitych o nazwie globalnej `MyGC` i lokalnej `gc`, który zawiera macierz przekształcenia bryły widzenia na kostkę standardową oraz kolory znaków i tła.

Zadaniem szadera wierzchołków jest tylko obliczenie współrzędnych wierzchołków obszaru tekstu w kostce standardowej; polega to na obliczeniu iloczynu macierzy rzutowania podanej w polu `gc.pm` przez wektor współrzędnych jednorodnych wierzchołka, czyli atrybut `in_Position`.

[2]Czyli położenie napisu podajemy w dosyć tradycyjny sposób. Rzutowanie zapewnia, że współrzędne wierzchołków obszaru tekstu są współrzędnymi w układzie okna.

Listing 11.1: Szader fragmentów dla pisania tekstu

GLSL

```
1: #version 420
2:
3: layout(origin_upper_left) in vec4 gl_FragCoord;
4:
5: uniform MyGC {
6:   mat4 pm;      /* macierz rzutowania */
7:   vec4 fg, bk;  /* kolory znaków i tła */
8: } gc;
9:
10: uniform MyFont {
11:   int  chw, chh;   /* szerokość i wysokość znaku */
12:   int  chf, chl;   /* kody pierwszego i ostatniego znaku */
13:   uint glyphs[288];
14: } font;
15:
16: uniform MyText {
17:   int  x, y;      /* pozycja pierwszego znaku */
18:   int  l;         /* długość napisu */
19:   uint text[64];  /* upakowane znaki napisu */
20: } text;
21:
22: out vec4 out_Color;
23:
24: #define SETFRAG(C) \
25:  { if ( gc.C.a > 0.0 ) {\
26:      out_Color = gc.C;\
27:      return;\
28:    }\
29:    else discard; }
30:
31: #define EXTRACTBYTE(X,B)\
32:   switch ( B ) {\
33: default: break;  /* 0 */\
34: case  1: X >>=  8;  break;\
35: case  2: X >>= 16;  break;\
36: case  3: X >>= 24;  break;\
37:   }
38:
39: void main ( void )
40: {
41:   int  x0, y0;
42:   uint c, r, mask, chrow;
```

```
43:
44:   x0 = int(gl_FragCoord.x) - text.x;
45:   if ( x0 < 0 || x0 >= font.chw*text.l )
46:     SETFRAG ( bk )
47:   y0 = int(gl_FragCoord.y) - text.y - 1;
48:   if ( y0 < 0 || y0 >= font.chh )
49:     SETFRAG ( bk )
50:   c = text.text[x0/(font.chw*4)];
51:   EXTRACTBYTE ( c, (x0/font.chw) % 4 )
52:   c &= 0xFF;
53:   if ( c < font.chf || c > font.chl )
54:     SETFRAG ( bk )
55:   c -= font.chf;
56:   r = c*font.chh + y0;
57:   if ( font.chw <= 8 ) {
58:     chrow = font.glyphs[r/4];
59:     EXTRACTBYTE ( chrow, r % 4 )
60:   }
61:   else if ( font.chw <= 16 ) {
62:     chrow = font.glyphs[r/2];
63:     if ( r % 2 != 0 )
64:       chrow >>= 16;
65:   }
66:   else
67:     chrow = font.glyphs[r];
68:   x0 %= font.chw;
69:   mask = 0x01 << x0;
70:   if ( (chrow & mask) != 0 )
71:     SETFRAG ( fg )
72:   else
73:     SETFRAG ( bk )
74: } /*main*/
```

W linii 3 mamy przedeklarowaną zmienną wejściową `gl_FragCoord` z kwalifikatorem układu `origin_upper_left`, który powoduje zastąpienie domyślnego układu współrzędnych w oknie przez układ, którego początek jest w górnym lewym narożniku okna, a oś y jest skierowana do dołu (zamiast rozważanego w rozdziale 6 układu (ξ, η) używamy układu (ξ', η')).

Font, którego należy użyć, jest umieszczony w bloku zmiennych jednolitych `MyFont` (lokalnie nazywanym `font`); są w nim podane: szerokość znaku `chw`, wysokość znaku `chh` oraz kody pierwszego i ostatniego znaku reprezentowanego w tablicy wzorców, odpowiednio `chf` i `chl`. W fontach opisanych dalej w tym

rozdziale będą to kody od 32 (spacja) do 127 (znak DEL), ale jedną z rzeczy możliwych (i wartych zrobienia) jest dodanie wzorców dziewięciu wielkich i dziewięciu małych liter z polskimi znakami diakrytycznymi. W tablicy glyphs są wzorce bitowe znaków, upakowane w liczbach typu uint. Jeśli szerokość znaku nie jest większa niż 8, to w każdym elemencie tablicy są po 4 wiersze pikseli wzorca. Jeśli szerokość znaku jest między 9 a 16, to w elemencie tablicy są 2 wiersze, a dla bardzo już szerokich znaków (od 17 do 32 pikseli) jeden wiersz wzorca zajmuje cały element tablicy.

Zadeklarowana długość tablicy glyphs w bloku MyFont odpowiada łącznej długości wzorców 96 znaków w foncie 12×6. Wzorzec każdego znaku zajmuje 12 bajtów, czyli tablica musi pomieścić 288 liczb czterobajtowych. Kompilator GLSL bardzo dba o zgodność typów w wyrażeniach, długości tablic itd., ale ta tablica jest na końcu bloku i w działającym programie będzie obecna w buforze, którego długość zostanie określona stosownie do ilości miejsca zajmowanego przez font. Jeśli reprezentacja fontu jest dłuższa, to szader nadal będzie poprawnie działał, byleby przekazane mu dane nie spowodowały próby czytania danych za końcem bufora.

Blok zmiennych jednolitych nazwany MyText (a lokalnie text) zawiera informacje związane z konkretnym napisem do wyświetlenia: są tam współrzędne x, y (czyli ξ', η') położenia początku tekstu, długość napisu l i tablica liczb typu uint; w elementach tej tablicy kody ASCII kolejnych znaków napisu są upakowane po 4.

Dobrze napisane makrodefinicje mogą znacznie skrócić i uczytelnić kod źródłowy. Szader na listingu 11.1 wykorzystuje dwie. Kod generowany przez makro SETFRAG, zależnie od składowej alfa koloru podanego jako parametr (koloru znaków albo tła), przypisuje ten kolor zmiennej wyjściowej out_Color i kończy (za pomocą instrukcji return) działanie szadera albo kończy działanie szadera za pomocą instrukcji discard, co powoduje odrzucenie fragmentu (czyli pozostawienie niezmienionego koloru piksela).

Makro EXTRACTBYTE służy do „wycinania” bajtu ze zmiennej typu uint podanej jako pierwszy parametr tego makra. Wskazany bajt, jeden z czterech, zostaje w tej zmiennej przesunięty na najmniej znaczącą pozycję.

Szader fragmentów dysponuje informacją podaną (przez etap rasteryzacji w potoku przetwarzania grafiki) w zmiennej gl_FragCoord. Jej pola x i y są współrzędnymi ξ', η' piksela w oknie (w układzie o początku w górnym lewym narożniku okna). W liniach 44–46 szader oblicza (i przypisuje zmiennej x0) przesunięcie piksela względem lewego brzegu obszaru tekstu. Jeśli piksel jest poza

tym obszarem[3], to zależnie od kanału alfa (badamy to w linii 25) nadajemy pikselowi (przez przypisanie do zmiennej `out_Color`) kolor tła albo odrzucamy fragment.

W liniach 47–49 obliczamy numer wiersza we wzorcu znaku, odpowiadający pikselowi przetwarzanemu przez szader. Jeśli ten numer jest mniejszy niż 0, to piksel leży powyżej obszaru tekstu, a jeśli jest większy lub równy wysokości znaku, to poniżej. W obu tych przypadkach, zależnie od kanału alfa koloru tła, przypisujemy zmiennej `out_Color` kolor tła albo odrzucamy fragment.

W liniach 50–54 z tablicy, w której jest podany napis, jest wyciągany kod ASCII znaku, w obszarze którego znajduje się przetwarzany piksel[4]; jest do tego użyte makro `EXTRACTBYTE`, które w zmiennej `c`, zawierającej początkowo kod potrzebnego znaku i kody trzech znaków sąsiednich, przesuwa potrzebny bajt na najmniej znaczącą pozycję. W linii 51 pozostałe bajty są kasowane. Jeśli otrzymana w ten sposób liczba jest poza zakresem kodów obecnych w foncie znaków, to w linii 54 nadajemy pikselowi kolor tła.

Zmniejszenie wartości zmiennej `c` o najmniejszy kod reprezentowanego znaku (`font.chf`) w linii 55 jest początkiem obliczenia indeksu potrzebnego wiersza wzorca znaku. Jeśli wiersze wzorców wszystkich znaków ustawimy kolejno jeden pod drugim, to wartość nadana zmiennej `r` w linii 56 jest numerem wiersza w takim wzorcu całego fontu. Instrukcje warunkowe w liniach 57 i 61 wybierają właściwe postępowanie dla przypadków, gdy znaki są wąskie (co najwyżej 8 pikseli), średniej szerokości (od 9 do 16) lub szerokie (do 32 pikseli). W linii 58 z tablicy `font.glyphs` wybierany jest element zawierający odpowiedni wiersz, a następnie makro `EXTRACTBYTE` wybiera odpowiedni bajt z tego elementu. Analogicznie działa wybieranie potrzebnej połówki liczby 32-bitowej dla znaków o średniej szerokości zaprogramowane w liniach 62–64.

W linii 68 zmienna `x0` otrzymuje wartość będącą numerem kolumny we wzorcu znaku. Wartość nadana zmiennej `mask` w linii 69 jest maską bitową, w której jest jedynka na pozycji tej kolumny i zera wszędzie indziej. Warunek w linii 70 jest spełniony, gdy w wierszu wzorca znaku też jest jedynka na tej pozycji i wtedy

[3] To może się zdarzyć, jeśli chcielibyśmy narysować figurę większą niż obszar tekstu.

[4] Zauważmy, że dla kroju pisma o stałej szerokości ustalenie, o który znak napisu chodzi, jest możliwe przy użyciu jednego dzielenia. Pismo tzw. proporcjonalne, ze znakami o różnych szerokościach, wymagałoby wcześniejszego utworzenia dodatkowej tablicy, w której dla każdego znaku napisu należałoby podać odległość jego początku od początku napisu, tj. od lewej krawędzi obszaru napisu. Szader fragmentów musiałby wyszukać w tej dodatkowej tablicy numer znaku w napisie, na przykład metodą bisekcji. A jeszcze należałoby obsłużyć kerning. Brrr . . .

należy nadać pikselowi kolor znaku (albo odrzucić fragment, gdy kanał alfa tego koloru jest zerem). Jak warunek jest niespełniony, to makro SETFRAG w linii 73 nadaje pikselowi kolor tła, albo fragment jest odrzucany.

Zwróćmy uwagę na postać warunku w linii 70. W GLSL wyrażenie w warunku musi być boolowskie. Tu jego wartość jest ustalana przez zbadanie relacji między liczbami. W języku C można by napisać po prostu `if ( chrow & mask ) ...`, ponieważ wyrażenie liczbowe jest uznawane za opis warunku, który jest spełniony wtedy, gdy wartość wyrażenia nie jest zerem, ale GLSL na to nie pozwala.

Fonty i procedury wyświetlania tekstu

Procedury związane z wyświetlaniem tekstu umieściłem w pliku mygltext.c, a dwa przygotowane fonty są w plikach font12x6.c i font18x10.c. Na listingu 11.2 jest pokazany fragment pliku nagłówkowego mygltext.h zawierający definicje struktur danych reprezentujących font i tekst do wyświetlenia. Sposób ich używania prześledzimy dalej.

Listing 11.2: Definicje struktur myFont i myTextObject

mygltext.h

```
 1: typedef struct {
 2:     GLint  chw, chh;  /* szerokość i wysokość znaku */
 3:     GLuint ubo;       /* bufor bloku MyFont */
 4:   } myFont;
 5:
 6: typedef struct {
 7:     GLuint vao, buf[2];  /* obszar tekstu i bufor bloku MyText */
 8:     int    maxlength;    /* maksymalna długość napisu */
 9:     myFont *font;        /* font dla tego napisu */
10:   } myTextObject;
```

Listing 11.3 pokazuje fragmenty pliku z definicją fontu, którego znaki mają 18 pikseli wysokości i 10 pikseli szerokości. Tak samo jest „zrobiony” font o wymiarach 12×6 i podobnie *można* zrobić dalsze fonty.

W tablicy glyphs18x10 jest podany ciąg liczb 16-bitowych z wzorcami znaków; znaków tych jest 96 (od spacji o kodzie 32 do znaku DEL o kodzie 127), zatem jest tam $96 \cdot 18 = 1728$ liczb, czyli 3456 bajtów. Wykropkowałem większość liczb, których i tak nikt by nie przeczytał, zostawiając tylko wzorce trzech znaków, w tym litery z rysunku 11.1. Procedura NewFontObject, która alokuje

Listing 11.3: Reprezentacja przykładowego fontu

font18x10.c

```
1: #include <GL/glew.h>
2: #include "mygltext.h"
3:
4: /* ten font został otrzymany przez konwersję fontu z pliku */
5: /* 10x20-ISO8859-1-pcf.gz z systemu X Window */
6: static GLushort glyphs18x10[] =
7:  {0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,
8:   0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,0x000,  /* ' ' */
9: .........
10:   0x000,0x07e,0x0c6,0x186,0x186,0x186,0x186,0x0c6,0x07e,
11:   0x006,0x006,0x006,0x006,0x006,0x000,0x000,0x000,0x000,  /* 'P' */
12: .........
13:   0x000,0x1ce,0x102,0x102,0x000,0x000,0x102,0x102,0x102,
14:   0x000,0x000,0x102,0x102,0x1ce,0x000,0x000,0x000,0x000}; /* DEL */
15:
16: myFont *NewFont18x10 ( void )
17: {
18:   return NewFontObject ( 10, 18, 0x20, 0x7F,
19:                          sizeof(glyphs18x10), glyphs18x10 );
20: } /*NewFont18x10*/
```

w pamięci CPU strukturę typu myFont, tworzy UBO, umieszcza w nim dane i zwraca wskaźnik do tej struktury, jest pokazana na listingu 11.5.

Listing 11.4 przedstawia procedurę LoadTextShaders, która czyta i kompiluje szadery do wyświetlania tekstu, a następnie łączy je w program szaderów i przygotowuje ten program do pracy. Po skompilowaniu szaderów i połączeniu ich w program, procedura po trzykroć wywołuje GetAccessToUniformBlock (listing 10.5), aby uzyskać dostęp do pól w jego trzech blokach zmiennych jednolitych.

W liniach 31–35 procedura LoadTextShaders tworzy UBO dla bloku MyGC i przywiązuje go (na stałe) do odpowiedniego punktu dowiązania. Pozostałe dwa bloki mogą istnieć w wielu egzemplarzach, ponieważ aplikacja może używać wielu różnych fontów i może utworzyć wiele obiektów reprezentujących napisy do wyświetlenia — dlatego odpowiednie bufory zostaną utworzone później według potrzeby i UBO będą wiązane z odpowiednimi punktami bezpośrednio przed wyświetlaniem tekstu.

Zadaniem procedury NewFontObject (listing 11.5) jest utworzenie UBO

zawierającego strukturę MyFont zdefiniowaną w liniach 9–13 na listingu 11.1 i nadanie odpowiednich wartości polom zmiennej typu myFont zaalokowanej w linii 6. W liniach 8–9 jest generowany i przywiązywany do celu GL_UNIFORM_BUFFER bufor, który będzie wykorzystywany jako UBO. Wcześniej wywołana procedura LoadTextShaders umieściła w globalnej tablicy fontofs przesunięcia poszczególnych pól struktury MyFont używanej przez szader: kolejno

Listing 11.4: Procedura LoadTextShaders

mygltext.c

```
1: static GLuint text_program_id = 0, text_shader_id[2];
2: static GLuint gcbi, gcbp, fontbi, fontbp, textbi, textbp;
3: static GLint  gcofs[3], fontofs[5], textofs[4];
4: static GLuint gcbuf;
5:
6: void LoadTextShaders ( void )
7: {
8:   static const char *filename[] =
9:     { "font.glsl.vert", "font.glsl.frag" };
10:   static const GLchar *UGCNames[] =
11:     { "MyGC", "MyGC.pm", "MyGC.fg", "MyGC.bk" };
12:   static const GLchar *UFontNames[] =
13:     { "MyFont", "MyFont.chw", "MyFont.chh",
14:       "MyFont.chf", "MyFont.chl", "MyFont.glyphs" };
15:   static const GLchar *UTextNames[] =
16:     { "MyText", "MyText.x", "MyText.y", "MyText.l", "MyText.text" };
17:   GLint  gcbsize, fontsize, textsize;
18:
19:   text_shader_id[0] = CompileShaderFiles ( GL_VERTEX_SHADER,
20:                                            1, &filename[0] );
21:   text_shader_id[1] = CompileShaderFiles ( GL_FRAGMENT_SHADER,
22:                                            1, &filename[1] );
23:   text_program_id = LinkShaderProgram ( 2, text_shader_id );
24:   GetAccessToUniformBlock ( text_program_id, 3, UGCNames,
25:                             &gcbi, &gcbsize, gcofs, &gcbp );
26:   GetAccessToUniformBlock ( text_program_id, 5, UFontNames,
27:                             &fontbi, &fontsize, fontofs, &fontbp );
28:   GetAccessToUniformBlock ( text_program_id, 4, UTextNames,
29:                             &textbi, &textsize, textofs, &textbp );
30:   glGenBuffers ( 1, &gcbuf );
31:   glBindBufferBase ( GL_UNIFORM_BUFFER, gcbp, gcbuf );
32:   glBufferData ( GL_UNIFORM_BUFFER, gcbsize, NULL, GL_DYNAMIC_DRAW );
33:   ExitIfGLError ( "LoadTextShaders" );
34: } /*LoadTextShaders*/
```

są to przesunięcia pól chw, chh, chf, chl i glyphs. Pola te są umieszczone w bloku kolejno, zatem ostatnim polem struktury jest tablica glyphs. Wielkość bufora, określona przez wywołanie procedury glBufferData w liniach 10–11, jest sumą przesunięcia pola glyphs i podanej jako parametr długości tej tablicy w bajtach, zaokrąglonej w górę (przez makro ROUNDUP4) do wartości podzielnej przez 4 — tablica zawiera liczby czterobajtowe, ostatnia liczba też musi w buforze mieścić się cała.

Listing 11.5: Procedury NewFontObject i DeleteFontObject

mygltext.c

```
1: #define ROUNDUP4(X) (4*(((X)+3)/4))
2:
3: myFont *NewFontObject ( GLint chw, GLint chh, GLint chf, GLint chl,
4:                         int size, GLvoid *glyphs )
5: {
6:   myFont *font;
7:
8:   if ( (font = malloc ( sizeof(myFont) )) ) {
9:     font->chw = chw;  font->chh = chh;
10:    glGenBuffers ( 1, &font->ubo );
11:    glBindBuffer ( GL_UNIFORM_BUFFER, font->ubo );
12:    glBufferData ( GL_UNIFORM_BUFFER, fontofs[4] + ROUNDUP4(size), NULL,
13:                   GL_STATIC_DRAW );
14:    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[0], sizeof(GLint), &chw );
15:    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[1], sizeof(GLint), &chh );
16:    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[2], sizeof(GLint), &chf );
17:    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[3], sizeof(GLint), &chl );
18:    glBufferSubData ( GL_UNIFORM_BUFFER, fontofs[4], size, glyphs );
19:    ExitIfGLError ( "NewFontObject" );
20:  }
21:  return font;
22: } /*NewFontObject*/
23:
24: void DeleteFontObject ( myFont *font )
25: {
26:  glDeleteBuffers ( 1, &font->ubo );
27:  free ( font );
28:  ExitIfGLError ( "DeleteFontObject" );
29: } /*DeleteFontObject*/
```

Kolejne wywołania procedury glBufferSubData w liniach 14–18 przypisują polom w buforze potrzebne dane; ponadto wymiary znaku są zapamiętywane w zaalokowanej zmiennej typu myFont, której adres jest zwracany jako wartość procedury.

Procedura `DeleteFontObject`, którą aplikacja powinna wywołać podczas sprzątania po sobie, po prostu zwalnia bufor utworzony przez `NewFontObject` i zwalnia pamięć CPU zajmowaną przez font.

Listing 11.6 przedstawia cztery procedury związane z obiektami reprezentującymi teksty do wyświetlenia. Procedura `NewTextObject` alokuje w pamięci CPU zmienną typu `myTextObject`, a w pamięci GPU bufory na reprezentację obszaru tekstu i samego tekstu. Przypomnijmy, że obszar tekstu jest prostokątem na płaszczyźnie, którego wierzchołki mają współrzędne całkowite. Dlatego bufor, w którym mają być umieszczone te wierzchołki, w liniach 12–13 otrzymuje wielkość wystarczającą na 8 liczb całkowitych. Instrukcje w liniach 14–15 ustalają, że atrybut numer `0` wierzchołka ma współrzędne reprezentowane jako liczby całkowite, podane w tym buforze. W liniach 17–19 ustalana jest wielkość bufora tekstu; jest ona sumą przesunięcia tablicy `text` względem początku bloku `MyText` (zobacz listing 11.1) i maksymalnej, zadeklarowanej za pomocą parametru `maxlength` długości napisu, zaokrąglonej w górę do wartości podzielnej przez 4.

Procedura `SetTextObjectContents` umieszcza w obiekcie tekstowym (zmiennej wskazywanej przez parametr `to`) konkretny napis, kojarzy z nim wybrany font i oblicza współrzędne wierzchołków obszaru tekstu. Ciąg kodów ASCII podany w tablicy `text` powinien reprezentować *jedną linię* tekstu; znaki specjalne, takie jak znak końca linii, są przez opisany tu program ignorowane, a dokładniej, zamieniane na spacje. Dlatego teksty złożone z kilku linii muszą być reprezentowane przez kilka obiektów tekstowych — każda linia przez osobny obiekt. Parametry `x` i `y` określają współrzędne dolnego lewego piksela obszaru tekstu w oknie[5]. Parametr `font` jest adresem struktury `myFont` utworzonej przez procedurę `NewFontObject`.

Jeśli napis jest dłuższy niż pojemność tablicy ustalona podczas tworzenia obiektu tekstu, to napis jest skracany (linie 31–33). W liniach 35–40 potrzebne dane są przesyłane do UBO. W liniach 42–45 procedura oblicza współrzędne wierzchołków obszaru tekstu i umieszcza je w buforze wierzchołków.

Procedura `DisplayTextObject` wyświetla tekst. Przed wywołaniem tej procedury trzeba zadbać o właściwą zawartość bloku zmiennych jednolitych `MyGC`, co procedura `SetupTextFrame`, opisana dalej, powinna była zrobić wcześniej.

[5] Wyświetlając kilka linii tekstu, należy je odpowiednio rozmieścić względem siebie. W szczególności trzeba zadbać o odpowiednią interlinię. Kolejna linia tekstu wyświetlonego przy użyciu fontu 12×6 powinna być 13 pikseli niżej niż poprzednia; podobnie dla fontu 18×10 linie tekstu powinny być rozmieszczone w pionie co 20 pikseli.

Listing 11.6: Procedury przetwarzania obiektów tekstowych

mygltext.c

```
1: myTextObject *NewTextObject ( int maxlength )
2: {
3:   myTextObject *to;
4:
5:   if ( (to = malloc ( sizeof(myTextObject) )) ) {
6:     memset ( to, 0, sizeof(myTextObject) );
7:     to->maxlength = ROUNDUP4(maxlength);
8:     glGenVertexArrays ( 1, &to->vao );
9:     glBindVertexArray ( to->vao );
10:    glGenBuffers ( 2, to->buf );
11:    glBindBuffer ( GL_ARRAY_BUFFER, to->buf[0] );
12:    glBufferData ( GL_ARRAY_BUFFER, 8*sizeof(GLint),
13:                   NULL, GL_DYNAMIC_DRAW );
14:    glEnableVertexAttribArray ( 0 );
15:    glVertexAttribPointer ( 0, 2, GL_INT, GL_FALSE,
16:                            2*sizeof(GLint), (GLvoid*)0 );
17:    glBindBuffer ( GL_UNIFORM_BUFFER, to->buf[1] );
18:    glBufferData ( GL_UNIFORM_BUFFER, textofs[3] + to->maxlength,
19:                   NULL, GL_DYNAMIC_DRAW );
20:    ExitIfGLError ( "NewTextObject" );
21:  }
22:  return to;
23: } /*NewTextObject*/
24:
25: void SetTextObjectContents ( myTextObject *to,
26:                              GLchar *text, GLint x, GLint y, myFont *font )
27: {
28:   GLint ta[8];
29:   int   lgt;
30:
31:   lgt = strlen ( text );
32:   if ( lgt > to->maxlength )
33:     lgt = to->maxlength;
34:   y -= font->chh;
35:   glBindBuffer ( GL_UNIFORM_BUFFER, to->buf[1] );
36:   glBufferSubData ( GL_UNIFORM_BUFFER, textofs[0], sizeof(GLint), &x );
37:   glBufferSubData ( GL_UNIFORM_BUFFER, textofs[1], sizeof(GLint), &y );
38:   glBufferSubData ( GL_UNIFORM_BUFFER, textofs[2], sizeof(GLint), &lgt );
39:   glBufferSubData ( GL_UNIFORM_BUFFER, textofs[3],
40:                     lgt*sizeof(GLchar), text );
41:   to->font = font;
42:   ta[0] = ta[6] = x;               ta[1] = ta[3] = y;
```

```
43:   ta[2] = ta[4] = x+lgt*font->chw;  ta[5] = ta[7] = y+font->chh;
44:   glBindBuffer ( GL_ARRAY_BUFFER, to->buf[0] );
45:   glBufferSubData ( GL_ARRAY_BUFFER, 0, 8*sizeof(GLint), ta );
46:   ExitIfGLError ( "SetTextObjectContents" );
47: } /*SetTextObjectContents*/
48:
49: void DisplayTextObject ( myTextObject *to )
50: {
51:   glDisable ( GL_CULL_FACE );
52:   glDisable ( GL_DEPTH_TEST );
53:   glUseProgram ( text_program_id );
54:   glBindBufferBase ( GL_UNIFORM_BUFFER, fontbp, to->font->ubo );
55:   glBindBufferBase ( GL_UNIFORM_BUFFER, textbp, to->buf[1] );
56:   glBindVertexArray ( to->vao );
57:   glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
58:   ExitIfGLError ( "DisplayTextObject" );
59: } /*DisplayTextObject*/
60:
61: void DeleteTextObject ( myTextObject *to )
62: {
63:   glDeleteVertexArrays ( 1, &to->vao );
64:   glDeleteBuffers ( 2, to->buf );
65:   free ( to );
66:   ExitIfGLError ( "DeleteTextObject" );
67: } /*DeleteTextObject*/
```

W liniach 51–52 procedura wyłącza odrzucanie ścian odwróconych tyłem i test widoczności, które nijak nie pasują do wyświetlania tekstu. Następnie w potok przetwarzania grafiki jest wmontowywany program utworzony przez `LoadTextShaders`. W liniach 54–55 jako bloki zmiennych jednolitych `MyFont` i `MyText` są podłączane UBO zawierające odpowiednio font, którym tekst ma być wyświetlony oraz tekst. W linii 56 VAO reprezentujący obszar tekstu jest czyniony bieżącym, po czym procedura `glDrawArrays` w linii 57 rysuje prostokąt — obszar tekstu. Resztę załatwiają szadery.

Procedura `DeleteTextObject` likwiduje bufory tekstu i wierzchołków oraz VAO obszaru tekstu i zwalnia pamięć CPU zajmowaną przez strukturę `myTextObject`.

Procedura `SetupTextFrame` na listingu 11.7 powinna być wywołana za każdym razem, gdy okno otrzymało nowe wymiary — czyli jej wywołanie trzeba dodać do procedury `ReshapeFunc` w aplikacji FreeGLUTa. Procedura ta konstruuje macierz przekształcenia bryły widzenia na kostkę standardową zgodnie z przepisem na

Listing 11.7: Procedury ustawiania przekształcenia i koloru oraz sprzątania

mygltext.c

```
1: void SetupTextFrame ( GLint width, GLint height )
2: {
3:   GLfloat pm[16];
4:
5:   M4x4Orthof ( pm, NULL, -0.5, (float)width-0.5,
6:                (float)height-0.5, -0.5, -1.0, 1.0 );
7:   glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
8:   glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[0], 16*sizeof(GLfloat), pm );
9:   ExitIfGLError ( "SetupTextFrame" );
10: } /*SetupTextFrame*/
11:
12: void SetTextForeground ( GLfloat fg[4] )
13: {
14:   glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
15:   glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[1], 4*sizeof(GLfloat), fg );
16:   ExitIfGLError ( "SetForeground" );
17: } /*SetTextForeground*/
18:
19: void SetTextBackground ( GLfloat bk[4] )
20: {
21:   glBindBuffer ( GL_UNIFORM_BUFFER, gcbuf );
22:   glBufferSubData ( GL_UNIFORM_BUFFER, gcofs[2], 4*sizeof(GLfloat), bk );
23:   ExitIfGLError ( "SetBackground" );
24: } /*SetTextBackground*/
25:
26: void TextCleanup ( void )
27: {
28:   int i;
29:
30:   for ( i = 0; i < 2; i++ ) {
31:     glDetachShader ( text_program_id, text_shader_id[i] );
32:     glDeleteShader ( text_shader_id[i] );
33:   }
34:   glDeleteProgram ( text_program_id );
35:   glDeleteBuffers ( 1, &gcbuf );
36:   ExitIfGLError ( "TextCleanup" );
37: } /*TextCleanup*/
```

końcu rozdziału 6: współrzędne x, y wierzchołka podane w VAO mają być równe współrzędnym ξ', η' obrazu wierzchołka w oknie. Współczynniki tej macierzy są w linii 8 przesyłane do UBO przywiązanego do bloku zmiennych jednolitych `MyGC` w programie szaderów.

Procedury `SetTextForeground` i `SetTextBackground` przesyłają kolory znaków i tła do odpowiednich pól w bloku `MyGC`. Procedura sprzątająca `TextCleanup` likwiduje program szaderów i zwalnia UBO przywiązany do bloku `MyGC`.

Aplikacja pierwsza C

Świeżo nabytej umiejętności pisania użyjemy do wyświetlania położenia obserwatora, tj. współrzędnych początku układu obserwatora w układzie świata — w czasie, gdy aplikacja jest w trybie obracania obserwatora wokół obiektu. Na listingu 11.8 są pokazane zmiany w kodzie aplikacji; trzeba dodać deklaracje kilku zmienych globalnych i dopisać bardzo już niewiele instrukcji, aby wszystko pięknie działało.

Listing 11.8: Dodatkowe zmienne i zmienione procedury aplikacji pierwszej C

C

```
1: myTextObject *vptext;
2: myFont  *font;
3:
4: void Initialise ( int argc, char *argv[] )
5: {
6: ........ /* tu instrukcje bez zmian */
7:   LoadMyShaders ();
8:   LoadTextShaders ();
9:   InitMyObject ();
10: } /*Initialise*/
11:
12: void InitMyObject ( void )
13: {
14:   TimerInit ();
15:   font = NewFont18x10 ();
16: /*  font = NewFont12x6 ();*/ /* można wypróbować jeden lub drugi */
17:   vptext = NewTextObject ( 60 );
18:   memset ( &trans, 0, sizeof(TransBl) );
19: ........ /* tu instrukcje bez zmian */
20:   InitViewMatrix ();
21:   NotifyViewerPos ();
22:   ConstructIcosahedronVAO ();
23:   InitLights ();
24: } /*InitMyObject*/
25:
26: void NotifyViewerPos ( void )
27: {
28:   GLchar s[60];
```

```
29:
30:   sprintf ( s, "x = %5.2f, y = %5.2f, z = %5.2f",
31:             trans.eyepos[0], trans.eyepos[1], trans.eyepos[2] );
32:   SetTextObjectContents ( vptext, s, 0, font->chh-1, font );
33: } /*NotifyViewerPos*/
34:
35: void ReshapeFunc ( int width, int height )
36: {
37: ........ /* tu instrukcje bez zmian */
38:   glViewport ( 0, 0, width, height );      /* klatka jest całym oknem */
39:   SetupTextFrame ( width, height );
40:   lr = 0.5533*(float)width/(float)height;  /* przyjmujemy aspekt równy 1 */
41: ........ /* tu instrukcje bez zmian */
42: } /*ReshapeFunc*/
43:
44: void DisplayFunc ( void )
45: {
46:   static GLfloat fg[4] = { 0.0, 0.0, 1.0, 1.0 };
47:   static GLfloat bk[4] = { 0.0, 0.0, 0.0, 0.0 };
48:
49:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
50:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
51:   if ( app_state == STATE_TURNING ) {
52:     SetTextForeground ( fg );
53:     SetTextBackground ( bk );
54:     DisplayTextObject ( vptext );
55:   }
56:   glEnable ( GL_DEPTH_TEST );
57: ........ /* tu instrukcje bez zmian */
58: } /*DisplayFunc*/
59:
60: void InitViewMatrix ( void )
61: {
62:   GLfloat m[16];
63:
64:   memcpy ( trans.eyepos, viewer_pos0, 4*sizeof(float) );
65:   M4x4Translatef ( m, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
66: ........ /* tu instrukcje bez zmian */
67: } /*InitViewMatrix*/
68:
69: void RotateViewer ( int delta_xi, int delta_eta )
70: {
71: ........ /* tu instrukcje bez zmian */
72:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1], 16*sizeof(GLfloat), vm );
73:   M4x4MultMVf ( trans.eyepos, rm, viewer_pos0 );
```

```
74:   ExitIfGLError ( "RotateViewer" );
75: } /*RotateViewer*/
76:
77: void MouseFunc ( int button, int state, int x, int y )
78: {
79: ........ /* tu instrukcje bez zmian */
80:       app_state = STATE_TURNING;
81:       glutPostWindowRedisplay ( WindowHandle );
82: ........ /* tu instrukcje bez zmian */
83:     if ( button == GLUT_LEFT_BUTTON && state != GLUT_DOWN ) {
84:       app_state = STATE_NOTHING;
85:       glutPostWindowRedisplay ( WindowHandle );
86:     }
87: ........ /* tu instrukcje bez zmian */
88: } /*MouseFunc*/
89:
90: void MotionFunc ( int x, int y )
91: {
92: ........ /* tu instrukcje bez zmian */
93:       RotateViewer ( x-last_xi, y-last_eta );
94:       NotifyViewerPos ();
95:       last_xi = x,  last_eta = y;
96: ........ /* tu instrukcje bez zmian */
97: } /*MotionFunc*/
98:
99: void Cleanup ( void )
100: {
101: ........ /* tu instrukcje bez zmian */
102:   glDeleteBuffers ( 3, icos_vbo );
103:   DeleteTextObject ( vptext );
104:   DeleteFontObject ( font );
105:   TextCleanup ();
106:   ExitIfGLError ( "Cleanup" );
107:   glutDestroyWindow ( WindowHandle );
108: } /*Cleanup*/
```

W zmiennej `trans.eyepos` są zapisywane współrzędne położenia obserwatora — to te liczby chcemy wyświetlać w oknie. Czwarty element tablicy jest potrzebny, bo zawiera ona wektor współrzędnych jednorodnych, z których czwarta zawsze będzie jedynką, a więc pierwsze trzy liczby są też współrzędnymi kartezjańskimi. Zmienne `font` i `vptext` są wskaźnikami struktur reprezentujących font i napis do wyświetlenia.

Procedura `Initialise` musi przygotować program szaderów do wyświetlania

tekstu, a więc doszła do niej instrukcja wywołująca procedurę `LoadTextShaders`.

Do procedury `InitMyObject` zostały dopisane trzy instrukcje, które odpowiednio tworzą font i obiekt tekstu oraz wpisują do obiektu tekstu współrzędne początkowego położenia obserwatora. Odpowiedni napis tworzy nowa procedura `NotifyViewerPos`.

W procedurze `ReshapeFunc` zostało dodane wywołanie procedury `SetupTextFrame`. Ponieważ przekształcenia brył widzenia na kostkę standardową są reprezentowane przez macierze przechowywane w różnych buforach, podłączonych do różnych punktów dowiązania, można te macierze skonstruować i przesłać do buforów w tym miejscu.

Instrukcje dodane do procedury `DisplayFunc` wyświetlają tekst, jeśli w chwili wywołania tej procedury aplikacja jest w trybie obracania obserwatora. Ponieważ czwarta składowa koloru tła (czyli wartość zmiennej `bk[3]`) jest zerem, piksele tła zachowają wcześniej nadany kolor.

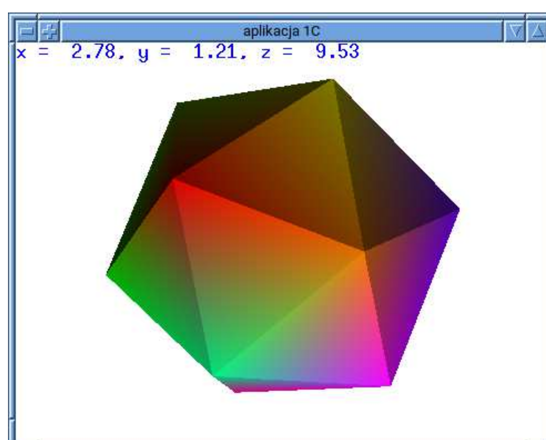
Procedura `MouseFunc` powiększyła się o dwie instrukcje, unieważniające obraz w oknie po zmianie trybu działania aplikacji: po wejściu w tryb obracania obserwatora napis w oknie ma się pojawić, a po wyjściu z tego trybu ma zniknąć, czyli w obu przypadkach obraz ma się zmienić.

Procedura `MotionFunc` po wywołaniu `RotateViewer` musi zadbać o uaktualnienie napisu do wyświetlenia w oknie i w tym celu wywołuje `NotifyViewerPos`.

Wreszcie, do procedury `Cleanup` zostały dopisane instrukcje likwidujące obiekty tekstu i fontu oraz likwidujące program szaderów wyświetlający tekst i bufor używany przez ten program.

Ćwiczenia

1. Przerób aplikację (program w C i szadery) tak, aby kolory znaków i tła były związane z obiektem tekstowym (w bloku `MyText`, a nie `MyGC`).
2. Dokonaj odpowiednich przeróbek i napisz dodatkowe procedury umożliwiające osobne podawanie (i zmienianie) napisu, jego położenia i fontu. Wykorzystaj to do wyświetlania informacji o położeniu obserwatora na dole okna; zmiany wymiarów okna przez użytkownika mają powodować umieszczanie napisu tuż nad dolną krawędzią okna.



Rysunek 11.2: Okno aplikacji pierwszej C

3. Przerób aplikację tak, aby kolory znaków i tła były podawane jako atrybuty wierzchołków obszaru tekstu. Wykorzystaj to do wyświetlenia tekstu, w którym kolor znaków i kolor tła zmieniają się płynnie.

4. Podaj najprostszy sposób, aby dwukrotnie powiększyć tekst, czyli spowodować, że każdy piksel wzorca znaku w używanym foncie będzie odwzorowany na kwadrat 2×2 piksele.

5. Napisz szader fragmentów i komplet procedur na CPU umożliwiające wyświetlanie tekstu złożonego z wielu linii (rozdzielonych znakami `'\n'`). Do reprezentacji tekstu trzeba wprowadzić dodatkową tablicę indeksów początków poszczególnych linii w tekście, przy czym dla uproszczenia można ustalić maksymalną liczbę (np. rzędu kilkadziesiąt, i tak więcej na ekranie się nie zmieści) linii reprezentowanych przez jeden obiekt tekstowy.

*Uzupełnienia

Na przykładzie aplikacji pierwszej C można prześledzić pewne problemy związane z używaniem szaderów skompilowanych do postaci SPIR-V. Bloki zmiennych jednolitych `MyFont` i `MyText` szadera fragmentów z listingu 11.1 używają domyślnego układu `shared`, w którym 32-bitowe liczby całkowite w tablicach są upakowane bez przerw. W takich liczbach pakujemy wzorce czcionek i kody ASCII kolejnych znaków napisu. Zewnętrzny kompilator GLSL-a obecnie nie obsługuje tego układu; w przypadku niepodania kwalifikatora układu samowolnie i chyłkiem (bez ostrzeżenia) zmienia układ na `std140`, zaś próba skompilowania szadera z deklaracją `layout(shared) uniform ...` kończy się otrzymaniem komunikatu o błędzie. W rozdziale 9 wspomniałem, że w układzie `std140` każdy element

tablicy liczb zajmuje tyle miejsca, co wektor o czterech współrzędnych złożony z takich liczb[6]. W związku z tym aplikacja, przesyłając dane do bufora, musiałaby odpowiednio „porozsuwać" elementy tablicy, co trzeba uznać za wyjątkowo niedołężny i pracochłonny sposób marnotrawienia miejsca w pamięci GPU.

Problem właściwego wykorzystania pamięci możemy rozwiązać w ten sposób, że zadeklarujemy tablice `MyFont.glyphs` i `MyText.text` z elementami typu uvec4; każdy taki element składa się z czterech 32-bitowych liczb bez znaku, a więc może w nim być upakowanych 16 bajtów, zaś między tymi elementami nie ma przerw. Ceną (akceptowalną) do zapłacenia jest komplikacja sposobu „wydobywania" danych z takiej tablicy. Jest jeszcze jeden problem. Kompilator (nie wypisując ostrzeżenia) ignoruje kwalifikator `layout(origin_upper_left)` zmiennej `gl_FragCoord`, w związku z czym trzeba wykonać obliczenia w domyślnym układzie współrzędnych, którego początek jest w dolnym lewym narożniku okna. Zmiany szadera fragmentów umożliwiające używanie jego wersji skompilowanej w SPIR-V są pokazane na listingu 11.9.

Kwalifikator `origin_upper_left` (z linii 3 na listingu 11.1) został usunięty. Zamiast niego w bloku `MyGC` pojawiło się pole `h`, którego wartość będzie wysokością klatki w pikselach; w linii 43 jest ona użyta w nowym sposobie obliczania numeru wiersza we wzorcu znaku. Tablice `MyFont.glyphs` i `MyText.text` składają się teraz z elementów typu uvec4. Spowodowało to konieczność dostosowania makra `EXTRACTBYTE`; pierwszym jego parametrem jest zmienna typu uint, której najmniej znaczącym ośmiu bitom ma być przypisany jeden z 16 bajtów zmiennej typu uvec4 podanej jako drugi parametr. Trzeci parametr jest liczbą od 0 do 15, która wybiera odpowiedni bajt.

Instrukcje w liniach 58–68 i 71–77 wybierają z elementu tablicy `MyFont.glyphs` potrzebne 16 albo 32 bity. Pozostałe instrukcje tego szadera nie wymagają zmian. Inne konieczne zmiany to dodanie pola `h` do bloku `MyGC` szadera wierzchołków tak, aby bloki te w obu szaderach były identyczne. W procedurze `LoadTextShaders` (listing 11.4) należy zadbać o uzyskanie dostępu do tego pola i zastąpić wywołania procedury `CompileShaderFiles` przez wywołania procedury `LoadSPIRVFile` (przedstawionej na listingu 4.9), której parametrem jest nazwa pliku szadera skompilowanego. Do procedury `ReshapeFunc` z listingu 11.8 trzeba dodać instrukcję przypisującą zmiennej jednolitej `MyGC.h` wysokość okna w pikselach. Wreszcie, rezerwując bufory, podane w bajtach rozmiary tablic na wzorce znaków i napis należy zaokrąglać w górę do wielokrotności 16 zamiast 4.

---

[6] Również wektory o 2 lub 3 współrzędnych są wyrównywane do długości wektorów o 4 współrzędnych przez wstawienie między nie odstępów w tablicy.

Listing 11.9: Zmodyfikowany szader fragmentów dla pisania tekstu

GLSL

```
1: #version 420
2:
3: uniform MyGC {
4:   mat4 pm;      /* macierz rzutowania */
5:   vec4 fg, bk;  /* kolory znakow i tła */
6:   int  h;       /* wysokość klatki */
7: } gc;
8:
9: uniform MyFont {
10:   int   chw, chh;   /* szerokość i wysokość znaku */
11:   int   chf, chl;   /* kody pierwszego i ostatniego znaku */
12:   uvec4 glyphs[1];
13: } font;
14:
15: uniform MyText {
16:   int   x, y;      /* pozycja pierwszego znaku */
17:   int   l;         /* długość napisu */
18:   uvec4 text[1];   /* upakowane znaki napisu */
19: } text;
20:
21: out vec4 out_Color;
22:
23: #define SETFRAG(C) .... /* makro SETFRAG bez zmian*/
24:
25: #define EXTRACTBYTE(a,b,s)\
26:   switch ( s ) {\
27: default: a = b.x;        break; /* 0 */\
28: case  1: a = b.x >>  8;  break;\
29: ........ /* podobne instrukcje dla wszystkich kolejnych bajtów */\
30: case 14: a = b.w >> 16;  break;\
31: case 15: a = b.w >> 24;  break;\
32:   }
33:
34: void main ( void )
35: {
36:   int   x0, y0;
37:   uvec4 b;
38:   uint  c, r, mask, chrow;
39:
40:   x0 = int(gl_FragCoord.x) - text.x;
41:   if ( x0 < 0 || x0 >= font.chw*text.l )
42:     SETFRAG ( bk )
```

```
43:   y0 = gc.h - 2 - int(gl_FragCoord.y) - text.y;
44:   if ( y0 < 0 || y0 >= font.chh )
45:     SETFRAG ( bk )
46:   b = text.text[x0/(font.chw*16)];
47:   EXTRACTBYTE ( c, b, (x0/font.chw) % 16 )
48:   c &= 0xFF;
49:   if ( c < font.chf || c > font.chl )
50:     SETFRAG ( bk )
51:   c -= font.chf;
52:   r = c*font.chh + y0;
53:   if ( font.chw <= 8 ) {
54:     b = font.glyphs[r/16];
55:     EXTRACTBYTE ( chrow, b, r % 16 )
56:   }
57:   else if ( font.chw <= 16 ) {
58:     b = font.glyphs[r/8];
59:     switch ( r % 8 ) {
60:   default: chrow = b.x;         break;
61:   case  1: chrow = b.x >> 16;  break;
62:   case  2: chrow = b.y;         break;
63:   case  3: chrow = b.y >> 16;  break;
64:   case  4: chrow = b.z;         break;
65:   case  5: chrow = b.z >> 16;  break;
66:   case  6: chrow = b.w;         break;
67:   case  7: chrow = b.w >> 16;  break;
68:     }
69:   }
70:   else {
71:     b = font.glyphs[r/4];
72:     switch ( r % 4 ) {
73:   default: chrow = b.x;  break;
74:   case  1: chrow = b.y;  break;
75:   case  2: chrow = b.z;  break;
76:   case  3: chrow = b.w;  break;
77:     }
78:   }
79:   x0 %= font.chw;
80:   mask = 0x01 << x0;
81:   if ( (chrow & mask) != 0 )
82:     SETFRAG ( fg )
83:   else
84:     SETFRAG ( bk )
85: } /*main*/
```

12. Aplikacja pierwsza D

Zamienimy krawędzie i ściany dwudziestościanu w figury zakrzywione — łuki okręgów i trójkąty sferyczne położone na sferze jednostkowej, w którą jest wpisany dwudziestościan. Każda krawędź zostanie zamieniona na złożoną z krótkich odcinków łamaną o wierzchołkach na sferze. Każda trójkątna ściana zostanie zastąpiona przez wiele małych trójkątów o wierzchołkach na sferze, dzięki czemu otrzymamy obraz trudno odróżnialny od obrazu sfery. Całe to obliczenie zostanie wykonane przez szadery rozdrabniania, które wprowadzimy do aplikacji.

Szadery i programy szaderów

Utworzymy trzy nowe programy szaderów, z szaderami rozdrabniania. Wszystkie te programy będą używać tego samego szadera wierzchołków, pokazanego na listingu 12.1. Szader ten nie wykonuje żadnych obliczeń, tylko kopiuje atrybuty wierzchołków (położenie i kolor) na wyjście.

Listing 12.1: Szader wierzchołków dla programów rozdrabniających

GLSL

```
1: #version 420
2:
3: layout(location=0) in vec4 in_Position;
4: layout(location=1) in vec4 in_Colour;
5:
6: out Vertex {
7:   vec4 Colour;
8: } Out;
9:
10: void main ( void )
11: {
12:   gl_Position = in_Position;
13:   Out.Colour = in_Colour;
14: } /*main*/
```

Pierwszy z nowych programów ma na celu narysowanie łuków okręgu położonych na sferze jednostkowej. Łuk będzie przybliżony łamaną złożoną z 10 odcinków. Użyty w tym programie szader sterowania rozdrabnianiem (listing 12.2) określa taki poziom podziału, a poza tym przesyła atrybuty wierzchołka (dostarczone przez szader wierzchołków) na swoje wyjście.

Prymitywy geometryczne, które mają być rozdrabniane, są nazywane płatami

(*patches*); aby je narysować, odpowiednią procedurę (`glDrawArrays`, `glDrawElements` albo którąś z dotąd nie opisanych procedur rysujących) należy wywołać ze stałą `GL_PATCHES` podaną jako pierwszy parametr. Wcześniej trzeba poinformować OpenGL-a o liczbie wierzchołków każdego płata, wywołując procedurę `glPatchParameteri`, co będzie opisane dalej. Liczba wierzchołków płata jest też podana w kwalifikatorze `layout` w linii 5 — oczywiście każdy łuk ma dwa końce.

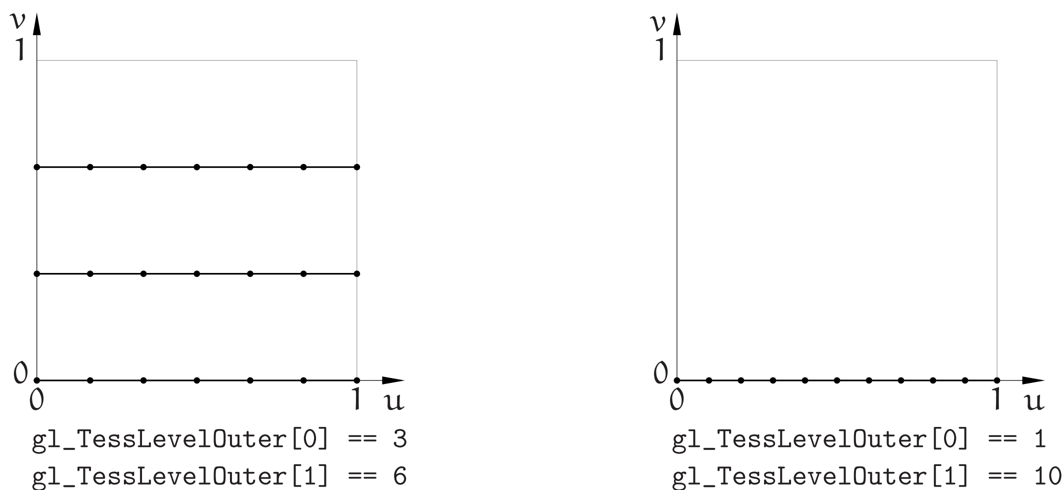
Listing 12.2: Pierwszy szader sterowania rozdrabnianiem

GLSL

```
1: #version 420
2:
3: #define MY_LEVEL 10
4:
5: layout(vertices=2) out;
6:
7: in Vertex {
8:     vec4 Colour;
9:   } In[];
10:
11: out TCVertex {
12:     vec4 Colour;
13:   } Out[];
14:
15: void main ( void )
16: {
17:   if ( gl_InvocationID == 0 ) {
18:     gl_TessLevelOuter[0] = 1;          /* jedna łamana */
19:     gl_TessLevelOuter[1] = MY_LEVEL;   /* z tylu odcinków */
20:   }
21:   gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
22:   Out[gl_InvocationID].Colour = In[gl_InvocationID].Colour;
23: } /*main*/
```

Szader sterowania rozdrabnianiem ma wejście i wyjście zadeklarowane jako tablice (dotyczy to nie tylko zmiennych wbudowanych `gl_in` i `gl_out`, ale też atrybutów w strukturach dodatkowych, zobacz linie 9 i 13 na listingu 12.2), przy czym szader ma za zadanie przetworzyć *tylko jeden* wierzchołek. Numer tego wierzchołka jest podany w zmiennej wbudowanej `gl_InvocationID`, która służy do indeksowania tych tablic; w tym przypadku przyjmuje ona tylko wartości 0 lub 1.

Dla jednego (i tylko jednego) wierzchołka płata szader sterowania rozdrabnianiem



Rysunek 12.1: Podziały dziedziny płata w trybie izolinii

ma dostarczyć informację o tym jak bardzo płat ma być rozdrobniony. Dlatego przypisania wartości elementom tablicy `gl_TessLevelOuter` (i `gl_TessLevelInner`, występujące w szaderach sterowania rozdrabnianiem opisanych dalej) są wykonywane w instrukcji warunkowej — tylko wtedy, gdy zmienna `gl_InvocationID` ma wartość 0. Potok przetwarzania grafiki z wmontowanym szaderem rozdrabniania przedstawionym na listingu 12.3 pracuje w <u>trybie izolinii</u> (*isoline mode*). Oznacza to, że dziedzina płata przetwarzana przez etap rozdrabniania w potoku (znajdująca się między szaderem sterowania rozdrabnianiem i szaderem rozdrabniania) jest kwadratem jednostkowym, $[0, 1] \times [0, 1]$, w którym ma być wygenerowanych n linii poziomych (o współrzędnych $v = i/n$ dla $i = 0, \ldots, n - 1$). Każda z tych linii ma być podzielona na m odcinków (rys. 12.1). Liczby n i m szader sterowania rozdrabnianiem ma wpisać do tablicy `gl_TessLevelOuter` i w liniach 18 i 19 to właśnie robi.

Szader rozdrabniania również ma dostęp do całej tablicy wierzchołków płata na wejściu i ma obliczyć i wyprowadzić dane dla *jednego* wierzchołka wyjściowego — odpowiadającego punktowi w dziedzinie płata wygenerowanemu przez etap rozdrabniania[1]. Zadaniem szadera na listingu 12.3 jest obliczenie wierzchołka łamanej przybliżającej łuk okręgu wielkiego sfery jednostkowej, którego końce są wierzchołkami płata podanymi na wejściu. Szader ma też dokonać przekształcenia wierzchołka łamanej do kostki standardowej i obliczyć jego kolor.

Kwalifikator <u>`layout`</u> w linii 3 określa, że ma być użyty tryb izolinii i każda (tj.

[1]Oczywiście, etap rozdrabniania produkuje wiele punktów w dziedzinie płata; generowanie przez szader rozdrabniania wierzchołków wyjściowych dla tych punktów odbywa się równolegle.

w tym przypadku jedna) linia ma być podzielona na części o jednakowej długości. W tablicy gl_in są podane podstawowe atrybuty obu wierzchołków płata, w szczególności ich położenia. W tablicy In jest dodatkowy atrybut, tj. kolor każdego z tych wierzchołków. Wyjście szadera to zmienna wbudowana gl_Position, do której ma być przypisany wektor współrzędnych jednorodnych nowego wierzchołka w kostce standardowej i struktura Out, do której jedynego pola ma być przypisany kolor nowego wierzchołka. Zwróćmy uwagę na identyczną nazwę zewnętrzną TCVertex wyjścia szadera z listingu 12.2 i wejścia szadera z listingu 12.3.

Listing 12.3: Pierwszy szader rozdrabniania

GLSL
```
1: #version 420
2:
3: layout(isolines,equal_spacing) in;
4:
5: in TCVertex {
6:     vec4 Colour;
7:   } In[];
8:
9: out TEVertex {
10:     vec4 Colour;
11:   } Out;
12:
13: uniform TransBlock {
14:   mat4 mm, mmti, vm, pm, mvpm;
15:   vec4 eyepos;
16: } trb;
17:
18: void main ( void )
19: {
20:   float t, t1;
21:   vec4  vert;
22:
23:   t = gl_TessCoord.x;  t1 = 1.0-t;
24:   vert = t1*gl_in[0].gl_Position + t*gl_in[1].gl_Position;
25:   vert.xyz = normalize ( vert.xyz );  vert.w = 1.0;
26:   gl_Position = trb.mvpm * vert;
27:   Out.Colour = t1*In[0].Colour + t*In[1].Colour;
28: } /*main*/
```

Szader rozdrabniania ma też dostęp do bloku zmiennych jednolitych TransBlock, przechowującego macierze przekształceń których złożenie jest przejściem od

układu współrzędnych obiektu do kostki standardowej; to właśnie ten szader w naszym programie ma dokonać tego przekształcenia. Natomiast informacja wygenerowana przez etap rozdrabniania dziedziny płata jest podana w zmiennej wejściowej `gl_TessCoord` — współrzędne u, v punktu w dziedzinie są wartościami pól `x`, `y` tej zmiennej. Jest to zmienna wektorowa; w tym miejscu potrzebujemy tylko pierwszej jej współrzędnej, którą dla wygody przypisujemy (w linii 23) zmienej `t`. W linii 24 dokonujemy interpolacji między wierzchołkami końcowymi płata. W linii 25 punkt na odcinku jest rzutowany na sferę jednostkową. W linii 26 punkt na sferze jest przekształcany do kostki standardowej, a w linii 27 obliczany jest jego kolor, przez interpolację kolorów końców odcinka.

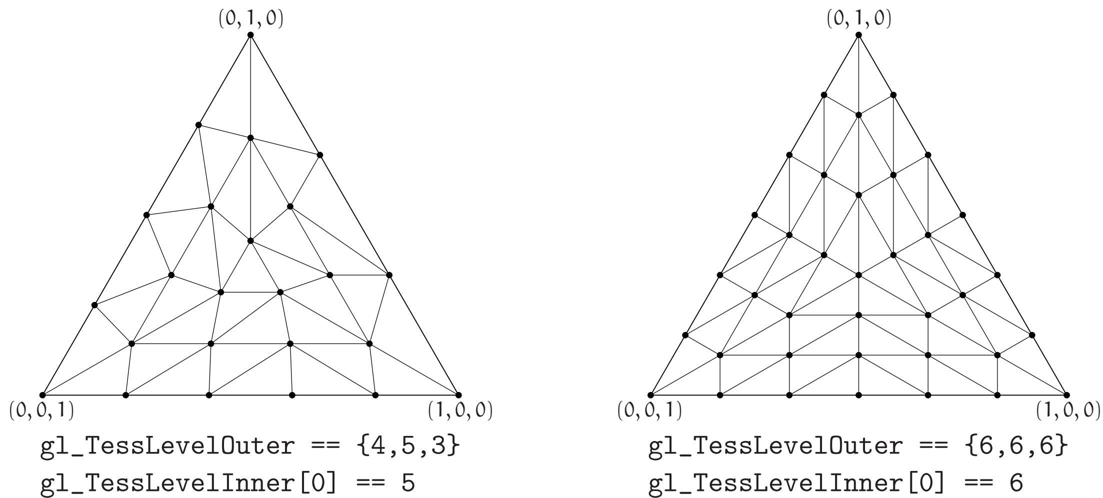
Listing 12.4: Szader fragmentów pierwszego i drugiego programu z rozdrabnianiem

GLSL

```
1: #version 420
2:
3: in TEVertex {
4:   vec4 Colour;
5: } In;
6:
7: out vec4 out_Colour;
8:
9: void main ( void )
10: {
11:   out_Colour = In.Colour;
12: } /*main*/
```

Na listingu 12.4 jest szader fragmentów programów, które odcinki i trójkąty otrzymane w wyniku rozdrabniania po prostu wyświetlają w kolorach otrzymanych przez interpolację kolorów będących atrybutami wierzchołków (bez uwzględnienia oświetlenia). Od szadera fragmentów na listingu 7.2 szader ten różni się tylko inną nazwą zewnętrzną struktury wejściowej.

Drugi program rozdrabniający ma przekształcić płaskie trójkąty, których wierzchołki leżą na sferze jednostkowej na trójkąty sferyczne położone na tej sferze (oczywiście trójkąty te będą przybliżane przez wiele drobnych trójkątów płaskich). Szader sterowania rozdrabnianiem pokazany na listingu 12.5, podobnie, jak szader z listingu 12.2, przepisuje atrybuty jednego wierzchołka trójkąta (położenie i kolor) z tablic wejściowych do tablic wyjściowych. Dla jednego z tych wierzchołków szader wpisuje do tablic `gl_TessLevelOuter` i `gl_TessLevelInner` informację o pożądanym stopniu rozdrobnienia. Fragmenty niezmienione w porównaniu z szaderem z listingu 12.2 zostały przerobione na szaro.



Rysunek 12.2: Podziały trójkątnej dziedziny płata

Na rysunku 12.2 są pokazane trójkąty podzielone w zależności od parametrów podanych w tablicach gl_TessLevelOuter i gl_TessLevelInner. Do pierwszej z tych tablic trzeba wpisać trzy liczby, które określają stopnie rozdrobnienia trzech boków trójkąta. Liczba podana w pierwszym elemencie drugiej tablicy

Listing 12.5: Drugi szader sterowania rozdrabnianiem

GLSL

```
#version 420

#define MY_LEVEL0 10
#define MY_LEVEL1 10

layout(vertices=3) out;

in Vertex { ... } In[];      /* wejście i wyjście tego szadera */
out TCVertex { ... } Out[];  /* są takie same jak szadera z listingu 12.2 */

void main ( void )
{
  if ( gl_InvocationID == 0 ) {
    gl_TessLevelOuter[0] = MY_LEVEL0;
    gl_TessLevelOuter[1] = MY_LEVEL0;
    gl_TessLevelOuter[2] = MY_LEVEL0;
    gl_TessLevelInner[0] = MY_LEVEL1;
  }
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
  Out[gl_InvocationID].Colour = In[gl_InvocationID].Colour;
} /*main*/
```

określa liczbę trójkątnych „warstw cebuli” mających powstać wskutek podziału; nieco ściślej biorąc, liczba tych warstw jest połową podanej liczby.

Szader rozdrabniania na listingu 12.6 oblicza i przekształca do kostki standardowej jeden wierzchołek odpowiadający pewnemu punktowi z dziedziny płata wygenerowanemu przez etap rozdrabniania dziedziny. W przypadku płatów trójkątnych zmienna wektorowa `gl_TessCoord` zawiera współrzędne barycentryczne tego punktu w układzie odniesienia wierzchołków trójkątnej dziedziny. Jak widać w liniach 15–16 i 19, interpolacja przy użyciu współrzędnych barycentrycznych jest bardzo łatwa do przeprowadzenia. Punkt płaskiego trójkąta w przestrzeni jest w linii 17 rzutowany na sferę jednostkową, a następnie przekształcany do kostki standardowej, identycznie jak w szaderze z listingu 12.3.

Listing 12.6: Drugi szader rozdrabniania

GLSL

```
1: #version 420
2:
3: layout(triangles,equal_spacing) in;
4:
5: in TCVertex { ... } In[];        /* ten fragment jest taki sam */
6: out TEVertex { ... } Out;        /* jak w liniach 5-16 */
7: uniform TransBlock { ... } trb;  /* na listingu 12.3 */
8:
9: void main ( void )
10: {
11:   float s, t, u;
12:   vec4  vert;
13:
14:   s = gl_TessCoord.x;  t = gl_TessCoord.y;  u = gl_TessCoord.z;
15:   vert = s*gl_in[0].gl_Position + t*gl_in[1].gl_Position +
16:          u*gl_in[2].gl_Position;
17:   vert.xyz = normalize ( vert.xyz );  vert.w = 1.0;
18:   gl_Position = trb.mvpm * vert;
19:   Out.Colour = s*In[0].Colour + t*In[1].Colour + u*In[2].Colour;
20: } /*main*/
```

Szader rozdrabniania używany przez trzeci program jest trochę bardziej skomplikowany, ponieważ wyniki działania tego szadera mają być użyte do obliczania oświetlenia. Szader musi zatem obliczyć wektor normalny powierzchni. Można to zadanie powierzyć szaderowi geometrii (podobnemu do tego z listingu 10.3), ale dla sfery jednostkowej znamy *dokładne* rozwiązanie zadania[2]:

[2]Uniwersalny szader geometrii może obliczyć wektor normalny płaszczyzny trójkąta, którego

w dowolnym punkcie $\mathbf{p}$ sfery jednostkowej o środku w początku układu współrzędnych jej jednostkowy wektor normalny jest wektorem współrzędnych kartezjańskich punktu $\mathbf{p}$.

Problem z wektorem normalnym polega na tym, że podczas przekształcania figur geometrycznych podlega on innym regułom niż punkty tych figur. Zbadajmy to. Każde przekształcenie afiniczne przestrzeni trójwymiarowej jest opisane wzorem (5.4), który tu przypomnę:

$$f(\mathbf{p}) = L\mathbf{p} + \mathbf{t}$$

Występuje w nim macierz L opisująca część liniową przekształcenia i wektor przesunięcia $\mathbf{t}$. Różnica punktów, $\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_1$, jest wektorem swobodnym, którego obraz, $\mathbf{w} = f(\mathbf{p}_2) - f(\mathbf{p}_1)$, otrzymamy, mnożąc ten wektor przez macierz L:

$$\mathbf{w} = (L\mathbf{p}_2 + \mathbf{t}) - (L\mathbf{p}_1 + \mathbf{t}) = L(\mathbf{p}_2 - \mathbf{p}_1) = L\mathbf{v}.$$

Natomiast obliczenie wektora normalnego obrazu płaszczyzny w przekształceniu afinicznym f opiera się na następującym rachunku: równanie płaszczyzny o wektorze normalnym $\mathbf{n}$ przechodzącej przez punkt $\mathbf{p}_0$ ma postać

$$\mathbf{n}^T(\mathbf{p} - \mathbf{p}_0) = 0.$$

Obrazy wszystkich punktów $\mathbf{p}$ płaszczyzny (w tym punktu $\mathbf{p}_0$) w przekształceniu f spełniają równanie

$$0 = \mathbf{n}^T L^{-1} L(\mathbf{p} - \mathbf{p}_0) = (L^{-T}\mathbf{n})^T\big(f(\mathbf{p}) - f(\mathbf{p}_0)\big).$$

Wynika stąd, że wektor normalny obrazu płaszczyzny w przekształceniu f jest (z dokładnością do czynnika stałego) wartością wyrażenia $L^{-T}\mathbf{n}$.[3] Jeśli przekształcenie reprezentujemy w postaci jednorodnej, to macierz L jest górnym lewym blokiem 3×3 odpowiedniej macierzy 4×4, zaś L^{-T} jest górnym lewym blokiem transpozycji odwrotności tej macierzy.

Jeśli przekształcenie f jest izometrią, to macierz L opisująca jego część liniową jest ortogonalna. W takim (i w tylko takim) przypadku $L^{-T} = L$. Mając obiekt określony w swoim lokalnym układzie, przekształcamy go najpierw do układu świata, następnie do układu obserwatora i wreszcie do układu kostki standardowej. Oświetlenie obliczamy w układzie świata, zatem musimy podać współrzędne wektora normalnego w tym układzie. W tym celu do bloku

wierzchołki leżą na sferze. To jest tylko dosyć niezłe przybliżenie wektora normalnego sfery.

[3] Zatem wektor normalny *nie jest* wektorem swobodnym. Musimy tu założyć, że macierz L jest nieosobliwa, co ma miejsce, gdy przekształcenie f jest różnowartościowe.

TransBlock zostało dodane jeszcze jedno pole, o nazwie mmti. Zawiera ono współczynniki transpozycji odwrotności macierzy przekształcenia modelu, podanej w polu mm. Za zapewnienie właściwej zawartości tych pól odpowiada aplikacja.

Uwaga: Ponieważ blok zmiennych jednolitych TransBlock jest używany przez wiele szaderów i przez wszystkie programy szaderów (z wyjątkiem opisanego w rozdz. 11 programu wyświetlającego tekst), trzeba zadbać o to, aby we wszystkich szaderach blok ten miał identyczny opis.

Listing 12.7: Trzeci szader rozdrabniania

GLSL

```
1: #version 420
2:
3: layout(triangles,equal_spacing) in;
4:
5: in TCVertex { ... } In[];  /* tak samo jak na listingu 12.3 */
6:
7: out NVertex {              /* tak samo jak na listingu 10.3 */
8:     vec4 Colour;
9:     vec3 Position;
10:    vec3 Normal;
11:  } Out;
12:
13: uniform TransBlock { ... } trb;  /* blok TransBlock taki sam jak wszędzie */
14:
15: void main ( void )
16: {
17:   float s, t, u;
18:   vec4  vert, vv;
19:
20: ........ /* tu linie 14-19 z listingu 12.6 */
21:   vv = trb.mm * vert;
22:   Out.Position = vv.xyz/vv.w;
23:   vert.w = 0.0;
24:   vv = trb.mmti * vert;
25:   Out.Normal = normalize ( vv.xyz );
26: } /*main*/
```

Dane wyjściowe szadera są dostosowane do wejścia szadera z listingu 10.4, który implementuje model oświetlenia lambertowskiego. W strukturze NVertex ma się znaleźć kolor wierzchołka (który będzie traktowany jak kolor farby odbijającej światło), współrzędne jego położenia w układzie świata i współrzędne wektora normalnego w układzie świata. W dodatku do obliczeń wykonywanych przez

szader z listingu 12.6, ten szader w linii 21 oblicza wektor współrzędnych jednorodnych wierzchołka w układzie świata. W linii 22 obliczane są jego współrzędne kartezjańskie. W linii 23 jest tworzona jednorodna reprezentacja wektora normalnego w układzie obiektu. W linii 24 następuje przejście do układu świata (w tym celu jest użyta macierz `trb.mmti` — transpozycja odwrotności macierzy `trb.mm`). W linii 25 wektor normalny w układzie świata jest poddawany normalizacji, której wynikiem jest wektor jednostkowy.

Podsumowując: wszystkie trzy nowe programy składają się z czterech szaderów — kolejno wierzchołków, sterowania rozdrabnianiem, rozdrabniania i fragmentów.

Pierwszy program składa się z szaderów przedstawionych na listingach 12.1, 12.2, 12.3 i 12.4. Jego przeznaczeniem jest wyświetlanie krawędzi dwudziestościanu, przerobionych na łuki okręgów.

Drugi program jest zbudowany z szaderów pokazanych na listingach 12.1, 12.5, 12.6 i 12.4. On z kolei zamienia płaskie trójkątne ściany na trójkąty sferyczne. Wyświetlenie wszystkich ścian dwudziestościanu powoduje powstanie obrazu sfery, przy czym kolory pikseli na tym obrazie są obliczane przez interpolację kolorów wierzchołków ścian.

Trzeci program składa się z szaderów na listingach 12.1, 12.5, 12.7 i 10.4. Szader fragmentów z listingu 10.4 podstawia kolor fragmentu jako kolor farby pokrywającej powierzchnię do modelu oświetlenia lambertowskiego. Otrzymany przy jego użyciu obraz przestawia sferę oświetloną.

Uwaga: Można w programie użyć tylko szadera rozdrabniania, bez szadera sterującego rozdrabnianiem. Wyjście szadera wierzchołków jest wtedy kierowane na wejście szadera rozdrabniania. Tryb (izolinii, trójkątów, czworokątów lub punktów) jest podany w kwalifikatorze `layout` wejścia szadera rozdrabniania. Parametry, które do tablic `gl_TessLevelOuter` i `gl_TessLevelInner` wpisywałby szader sterowania rozdrabnianiem podaje aplikacja, wykonując instrukcje

```
glPatchParameterfv ( GL_PATCH_DEFAULT_OUTER_LEVEL, otab );
glPatchParameterfv ( GL_PATCH_DEFAULT_INNER_LEVEL, itab );
```

W tablicach `otab` i `itab` należy podać odpowiednio cztery i dwie liczby typu `GLfloat`. Użycie szadera sterowania rozdrabnianiem umożliwia dostosowanie poziomu rozdrobnienia poszczególnych płatów do wielkości ich obrazów.

Aplikacja pierwsza D

Zmiany kodu aplikacji umożliwiające użycie szaderów opisanych wyżej są pokazane na listingach 12.8–12.12. Dotyczą one inicjalizacji programów szaderów, przygotowania obiektu do rysowania, odpowiedniej procedury go rysującej, interakcji z użytkownikiem, który może chcieć oglądać sferę lub dwudziestościan, i sprzątania.

Procedura `LoadMyShaders` przedstawiona na listingu 12.8 czyta, kompiluje i łączy pięć programów szaderów — dwa używane przez aplikację pierwszą B i trzy nowe. Ponieważ niektóre szadery wchodzą w skład więcej niż jednego programu, wszystkie są kompilowane w pętli w liniach 25–26. Nazwy plików z szaderami są podane w tablicy `filename`; kolejno są to szadery z listingów 10.1, 10.2, 12.1, 12.2, 12.5, 12.3, 12.6, 12.7, 10.3, 7.2, 10.4, 12.4, przy czym we wszystkich szaderach, w których występuje blok zmiennych jednolitych `TransBlock`, jego definicja jest taka, jak w tym rozdziale. W pomocniczej tablicy `shtype` podane są kolejno rodzaje tych szaderów, potrzebne kompilatorowi GLSL-a.

W liniach 27–39 następuje łączenie programów szaderów; każde z pięciu wywołań procedury `LinkShaderProgram` jest poprzedzone wpisaniem do tablicy `sh` identyfikatorów szaderów składających się na dany program.

W liniach 40–41 aplikacja tworzy UBO dla bloku `TransBlock` i uzyskuje informację o przesunięciach poszczególnych pól. Uwaga: Ponieważ do bloku zostało dodane nowe pole, `mmti`, elementy tablicy `trbofs` zmieniły znaczenie: `trbofs[0]` to nadal przesunięcie pola `mm`, ale teraz `trbofs[1]` oznacza przesunięcie pola `mmti`, `trbofs[2]` — pola `vm`, `trbofs[3]` — pola `pm`, `trbofs[4]` — pola `mvpm` i `trbofs[5]` — przesunięcie pola `eyepos`. Zgodnie z tym trzeba zmodyfikować parametry wywołań procedury `glBufferSubData` przesyłających dane do tego bufora w całej aplikacji, czego tu na listingach *nie zamieściłem*.

W liniach 42–43 blok `TransBlock` w pozostałych czterech programach szaderów jest przywiązywany do tego punktu dowiązania, do którego procedura `GetAccessToUniformBlock` przywiązała blok `TransBlock` w pierwszym programie.

Uzyskiwanie dostępu do bloku `LSBlock` z danymi opisującymi źródła światła odbywa się tak samo, jak w aplikacji pierwszej B. Ale do odpowiedniego punktu dowiązania trzeba też przyczepić blok `LSBlock` z ostatniego programu (linia 46). Instrukcje tworzące UBO dla obu bloków zmiennych jednolitych używanych przez programy pozostały niezmienione.

Listing 12.8: Procedura LoadMyShaders

C

```
1: GLuint program_id[5], shader_id[12], trbofs[6];
2:
3: void LoadMyShaders ( void )
4: {
5:   static const char *filename[12] =
6:     { "app1d0.glsl.vert", "app1d1.glsl.vert", "app1d2.glsl.vert",
7:       "app1d2.glsl.tesc", "app1d3.glsl.tesc",
8:       "app1d2.glsl.tese", "app1d3.glsl.tese", "app1d4.glsl.tese",
9:       "app1d1.glsl.geom",
10:      "app1d0.glsl.frag", "app1d1.glsl.frag", "app1d2.glsl.frag" };
11:  static const GLuint shtype[12] =
12:    { GL_VERTEX_SHADER, GL_VERTEX_SHADER, GL_VERTEX_SHADER,
13:      GL_TESS_CONTROL_SHADER, GL_TESS_CONTROL_SHADER,
14:      GL_TESS_EVALUATION_SHADER, GL_TESS_EVALUATION_SHADER,
15:      GL_TESS_EVALUATION_SHADER,
16:      GL_GEOMETRY_SHADER,
17:      GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER, GL_FRAGMENT_SHADER };
18:  static const GLchar *UTBNames[] =
19:    { "TransBlock", "TransBlock.mm", "TransBlock.mmti", "TransBlock.vm",
20:      "TransBlock.pm", "TransBlock.mvpm", "TransBlock.eyepos" };
21:  static const GLchar *ULSNames[] = { ... };  /* tak jak na listingu 10.7 */
22:  GLuint sh[4];
23:  int    i;
24:
25:  for ( i = 0; i < 12; i++ )
26:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
27:  sh[0] = shader_id[0];  sh[1] = shader_id[9];
28:  program_id[0] = LinkShaderProgram ( 2, sh );
29:  sh[0] = shader_id[1];  sh[1] = shader_id[8];  sh[2] = shader_id[10];
30:  program_id[1] = LinkShaderProgram ( 3, sh );
31:  sh[0] = shader_id[2];  sh[1] = shader_id[3];  sh[2] = shader_id[5];
32:  sh[3] = shader_id[11];
33:  program_id[2] = LinkShaderProgram ( 4, sh );
34:  sh[0] = shader_id[2];  sh[1] = shader_id[4];  sh[2] = shader_id[6];
35:  sh[3] = shader_id[11];
36:  program_id[3] = LinkShaderProgram ( 4, sh );
37:  sh[0] = shader_id[2];  sh[1] = shader_id[4];  sh[2] = shader_id[7];
38:  sh[3] = shader_id[10];
39:  program_id[4] = LinkShaderProgram ( 4, sh );
40:  GetAccessToUniformBlock ( program_id[0], 6, &UTBNames[0],
41:                            &trbi, &trbsize, trbofs, &trbbp );
42:  for ( i = 1; i <= 4; i++)
```

```
43:     AttachUniformBlockToBP ( program_id[i], UTBNames[0], trbbp );
44:   GetAccessToUniformBlock ( program_id[1], 7, &ULSNames[0],
45:                             &lsbi, &lsbsize, lsbofs, &lsbbp );
46:   AttachUniformBlockToBP ( program_id[4], ULSNames[0], lsbbp );
47: ........ /* tu niezmienione linie 26-28 z listingu 10.7 */
48: } /*LoadMyShaders*/
```

Na listingu 12.9 jest pokazana procedura obliczająca i przesyłająca do pamięci GPU (do pola `TransBlock.mm`) macierz przekształcenia modelu. Dodana do niej instrukcja przesyła transpozycję odwrotności tej macierzy do pola `TransBlock.mmti`. Ponieważ macierz przekształcenia modelu jest w tej aplikacji macierzą obrotu, czyli jest macierzą ortogonalną, jest ona równa swojej transpozycji odwrotności. Dlatego do obu pól, `mm` i `mmti`, są wpisywane te same dane. Ale zmiana postaci tego przekształcenia (w szczególności wprowadzenie nierównomiernego skalowania) spowodowałaby konieczność numerycznego wyznaczenia transpozycji odwrotności, do czego może posłużyć procedura `M4x4TInvertf` z listingu 5.2.[4]

Listing 12.9: Procedura `SetupModelMatrix`

```c
1: void SetupModelMatrix ( float axis[3], float angle )
2: {
3:   M4x4RotateVf ( trans.mm, axis[0], axis[1], axis[2], angle );
4:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
5:   glBufferSubData ( GL_UNIFORM_BUFFER,
6:                     trbofs[0], 16*sizeof(GLfloat), trans.mm );
7:   glBufferSubData ( GL_UNIFORM_BUFFER,
8:                     trbofs[1], 16*sizeof(GLfloat), trans.mm );
9:   ExitIfGLError ( "SetupModelMatrix" );
10:  SetupMVPMatrix ();
11: } /*SetupModelMatrix*/
```

Zmiana procedury `ConstructIcosaheronVAO` pokazana na listingu 12.10 polega na *zastąpieniu* ostatnich 36 liczb w tablicy indeksów wierzchołków przez 60 nowych — dajemy po trzy indeksy na każdą trójkątną ścianę[5]. W liniach 17–18 tworzony jest odpowiednio wydłużony bufor, do którego jest przesyłany nowy ciąg liczb.

[4] We wcześniejszych wersjach aplikacji nie było potrzeby wprowadzenia transpozycji odwrotności macierzy przekształcenia modelu, bo szader geometrii z listingu 10.3 oblicza wektory normalne ścian w układzie świata. Tu podajemy wektory normalne w innych układach współrzędnych i odpowiedni szader musi je do układu świata przekształcić.

[5] Rezygnujemy z wachlarzy i taśm trójkątowych, bo i tak potrzebujemy każdą ścianę wyspecyfikować osobno, jako płat do rozdrobnienia.

Listing 12.10: Nowe procedury konstrukcji i rysowania dwudziestościanu

C

```
1: void ConstructIcosahedronVAO ( void )
2: {
3: ........ /* tablice vertpos i vertcol takie jak na listingu 7.7 */
4:   static const GLubyte vertind[96] =
5:      { 0, 1, 2, 0, 3, 4, 0, 5, 1, 9, 2, 8, 3, /* łamana, od 0 */
6: ........ /* początek tablicy vertind taki jak na listingu 7.7 */
7:        2, 3, 4, 5, 8, 9, 10, 11,              /* 4 odcinki, od 28 */
8:        0, 1, 2, 0, 2, 3, 0, 3, 4, 0, 4, 5,    /* trójkątne płaty, od 36 */
9:        0, 5, 1, 6, 7, 8, 6, 8, 9, 6, 9, 10,
10:       6, 10, 11, 6, 11, 7, 1, 9, 2, 9, 8, 2,
11:       2, 8, 3, 8, 7, 3, 3, 7, 4, 7, 11, 4,
12:       4, 11, 5, 11, 10, 5, 10, 1, 5, 10, 9, 1};
13:
14: ........ /* tu wszystko tak samo, jak na listingu 7.7 */
15:   glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
16:   glBufferData ( GL_ELEMENT_ARRAY_BUFFER,
17:                  96*sizeof(GLubyte), vertind, GL_STATIC_DRAW );
18:   ExitIfGLError ( "ConstructIcosahedronVAO" );
19: } /*ConstructIcosahedronVAO*/
20:
21: void DrawIcosahedron ( int opt, char enlight )
22: {
23:   glBindVertexArray ( icos_vao );
24:   switch ( opt ) {
25: ........ /* rysowanie wierzchołków i krawędzi bez zmian */
26: default:   /* ściany */
27:     glUseProgram ( program_id[enlight ? 1 : 0] );
28:     glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
29:     glDrawElements ( GL_TRIANGLES, 60,
30:                      GL_UNSIGNED_BYTE, (GLvoid*)(36*sizeof(GLubyte)) );
31:     break;
32:   }
33: } /*DrawIcosahedron*/
34:
35: void DrawTessIcos ( int opt, char enlight )
36: {
37:   glBindVertexArray ( icos_vao );
38:   glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, icos_vbo[2] );
39:   switch ( opt ) {
40: case 0:
41:     break;
42: case 1:
```

```
43:     glUseProgram ( program_id[2] );
44:     glPatchParameteri ( GL_PATCH_VERTICES, 2 );
45:     glDrawElements ( GL_PATCHES, 24, GL_UNSIGNED_BYTE, (GLvoid*)0 );
46:     glDrawElements ( GL_PATCHES, 26, GL_UNSIGNED_BYTE,
47:                      (GLvoid*)(1*sizeof(GLubyte)) );
48:     glDrawElements ( GL_PATCHES, 10, GL_UNSIGNED_BYTE,
49:                      (GLvoid*)(26*sizeof(GLubyte)) );
50:     break;
51: default:
52:     glUseProgram ( program_id[enlight ? 4 : 3] );
53:     glPatchParameteri ( GL_PATCH_VERTICES, 3 );
54:     glDrawElements ( GL_PATCHES, 60, GL_UNSIGNED_BYTE,
55:                      (GLvoid*)(36*sizeof(GLubyte)) );
56:     break;
57:   }
58:   ExitIfGLError ( "DrawTessIcos" );
59: } /*DrawTessIcos*/
```

Podczas rysowania płatów trójkątnych każda trójka wierzchołków jest traktowana osobno — płatów nie łączy się w wachlarze ani taśmy. W związku ze zmianą tablicy indeksów procedura `DrawIcosahedron` została zmieniona tak, aby ściany dwudziestościanu były rysowane jako osobne trójkąty.

Nowa procedura `DrawTessIcos` wyświetla płaty — odcinki albo trójkąty, zależnie od wartości parametru `opt`. Jeśli jest równa 1, to rysowanych jest 30 płatów, z których każdy odpowiada jednej krawędzi dwudziestościanu. Do rysowania jest używany pierwszy z nowych programów (o identyfikatorze zapamiętanym w `program_id[2]`. Procedura `glPatchParameteri` w linii 44 zgłasza po dwa wierzchołki na płat. Następnie są trzy wywołania procedury `glDrawElements`. Został tu użyty pewien trik. Dane w tablicy indeksów wierzchołków opisują długą łamaną (o 25 wierzchołkach, czyli o 24 odcinkach), po niej krótką łamaną (o 3 wierzchołkach) i 4 osobne odcinki. Procedura wywołana w linii 31 rysuje co drugi odcinek długiej łamanej. Następne wywołanie rysuje pozostałe odcinki długiej łamanej i pierwszy odcinek łamanej krótkiej, a w liniach 48–49 jest rysowany drugi odcinek tej łamanej i cztery osobne odcinki.

Jeśli parametr `opt` ma wartość 2, to procedura `DrawTessIcos` rysuje płaty trójkątne. Indeksy ich wierzchołków to 60 liczb dodanych na końcu tablicy (w liniach 8–12 na listingu). Zależnie od wartości parametru `enlight` jest wybierany drugi lub trzeci z programów opisanych w tym rozdziale (ich identyfikatory są zapamiętane w zmiennych `program_id[3]` i `program_id[4]`). W linii 53 OpenGL jest informowany o tym, że kolejne płaty mają po 3 wierzchołki.

Procedura `DisplayFunc` na listingu 12.11 w zależności od wartości zmiennej `tesselate` wywołuje procedurę rysującą dwudziestościan albo sferę (otrzymaną przez rozdrobnienie płatów). Procedura `KeyboardFunc`, w dodatku do reakcji na dotychczas obsługiwane klawisze, po napisaniu litery `'T'` albo `'t'` zmienia wartość tej zmiennej i każe wykonać nowy obraz.

Listing 12.11: Procedury `DisplayFunc` i `KeyboardFunc`

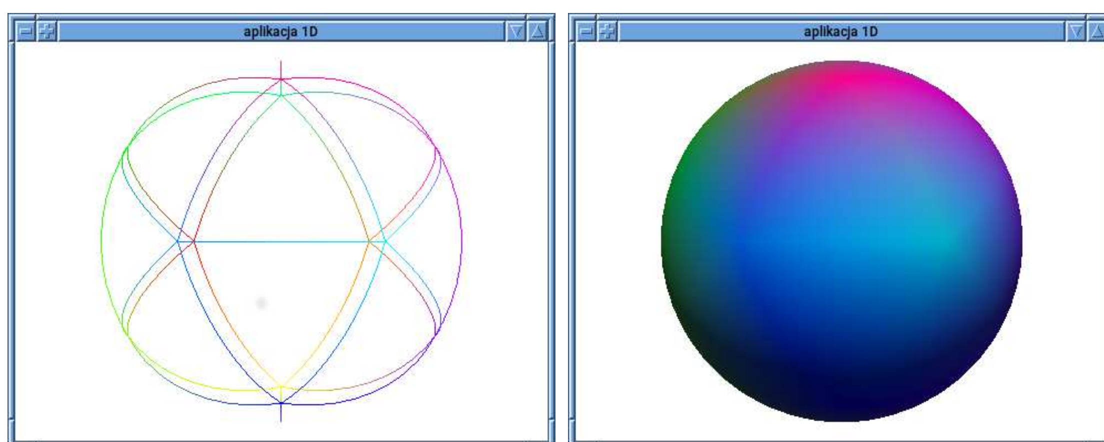
```c
1: char tesselate = 0;
2:
3: void DisplayFunc ( void )
4: {
5: ........ /* tu niezmienione linie 46-55 z listingu 11.8 */
6:   glEnable ( GL_DEPTH_TEST );
7:   if ( tesselate )
8:     DrawTessIcos ( opcja, enlight );
9:   else
10:    DrawIcosahedron ( opcja, enlight );
11:  glUseProgram ( 0 );
12:  glFlush ();
13:  glutSwapBuffers ();
14: } /*DisplayFunc*/
15:
16: void KeyboardFunc ( unsigned char key, int x, int y )
17: {
18:   switch ( key ) {
19: ........ /* tu instrukcje jak na listingach 7.10 i 10.10 */
20: case 'T':  case 't':
21:     tesselate = !tesselate;
22:     glutPostWindowRedisplay ( WindowHandle );
23:     break;
24: default:              /* ignorujemy wszystkie inne klawisze */
25:     break;
26:   }
27: } /*KeyboardFunc*/
```

Procedura sprzątania w dwóch pętlach likwiduje wszystkie programy i szadery, a następnie zwalnia pozostałe zasoby zarezerwowane przez aplikację. No i tyle.

Listing 12.12: Procedura sprzątania

```c
1: void Cleanup ( void )
2: {
3:   int i;
```

```
4:
5:   glUseProgram ( 0 );
6:   for ( i = 0; i < 5; i++ )
7:     glDeleteProgram ( program_id[i] );
8:   for ( i = 0; i < 12; i++ )
9:     glDeleteShader ( shader_id[i] );
10:  glDeleteBuffers ( 1, &trbuf );
11: ........ /* reszta sprzątania bez zmian */
12: } /*Cleanup*/
```



Rysunek 12.3: Okno aplikacji pierwszej D

Ćwiczenia

1. Wprowadź przekształcenie obiektu będące złożeniem skalowania nierównomiernego z obrotem. Zmodyfikuj odpowiednio instrukcję w liniach 7–8 na listingu 12.7 i wstaw przed nią instrukcję obliczającą transpozycję odwrotności macierzy przekształcenia modelu. Otrzymasz program wykonujący obrazy elipsoidy.

2. Wydłuż tablicę wierzchołków dwudziestościanu, dopisując do niej wierzchołki w połowie każdej krawędzi i w środkach ciężkości wszystkich ścian. Korzystając z nowych wierzchołków, podziel każdą ścianę na 6 trójkątów prostokątnych i utwórz nową tablicę trójek indeksów — ma w niej znaleźć się 180 liczb, będących numerami wierzchołków 60 trójkątów (trzech trójkątów otrzymanych z podziału każdej ściany dwudziestościanu). Wyświetl te trójkąty jako płaty, zamieniając je na trójkąty sferyczne.

3. Wprowadź przekształcenie nieliniowe obiektu, przez modyfikację szaderów z listingów 12.6 i 12.7. Po znalezieniu wierzchołka na sferze jednostkowej, a *przed* poddaniem go dalszym przekształceniom, pomnóż współrzędne x i y przez 0.75, a współrzędną z przez $e^{z/10}$ (możesz użyć funkcji exp).

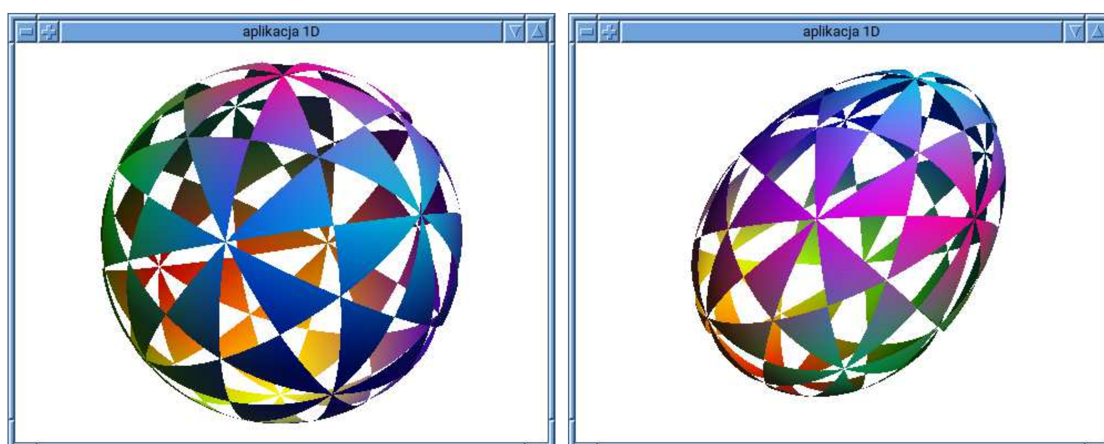
W obliczeniach wektora normalnego przekształcanej powierzchni przesunięcie jest nieistotne, skupmy się zatem na macierzach 3×3 opisujących część liniową i „część liniową” użytych przekształceń. W naszym przypadku przekształcenie obiektu jest opisane przez iloczyn macierzy MS, gdzie M jest macierzą obrotu modelu (to jej współczynniki są w polu `TransBlock.mm`), zaś $S = S(x, y, z)$ jest macierzą zmieniającą się ze współrzędnymi x, y, z przekształcanego punktu (co wprowadza nieliniowość). Dla ustalonych liczb x, y, z macierz S jest macierzą skalowania osi (o współczynnikach $\frac{3}{4}, \frac{3}{4}, e^{z/10}$). Równanie płaszczyzny stycznej do powierzchni będącej obrazem w przekształceniu f płaszczyzny o wektorze normalnym $\mathbf{n}$ opisanemu przez te macierze dla ustalonych liczb x, y, z otrzymamy tak:

$$0 = \mathbf{n}^T(\mathbf{p} - \mathbf{p}_0) = \mathbf{n}^T S^{-1} M^{-1} M S(\mathbf{p} - \mathbf{p}_0) = (M^{-T} S^{-T}\mathbf{n})^T\big(f(\mathbf{p}) - f(\mathbf{p}_0)\big).$$

Stąd wynika[6], że jeśli $\mathbf{n}$ oznacza wektor normalny sfery jednostkowej w rozpatrywanym (przetwarzanym przez szader) punkcie, to wektor normalny płaszczyzny stycznej do otrzymanej skorupki jest równy $M^{-T}S^{-T}\mathbf{n}$. Mamy

$$S = S^T = \begin{bmatrix} \frac{3}{4} & 0 & 0 \\ 0 & \frac{3}{4} & 0 \\ 0 & 0 & e^{z/10} \end{bmatrix}, \quad S^{-1} = S^{-T} = \begin{bmatrix} \frac{4}{3} & 0 & 0 \\ 0 & \frac{4}{3} & 0 \\ 0 & 0 & e^{-z/10} \end{bmatrix}.$$

Ćwiczenie polega m.in. na dodaniu do szadera z listingu 12.7 instrukcji obliczających wektor normalny zgodnie z tym rachunkiem.



Rysunek 12.4: Rozwiązania ćwiczeń 2 i 3

[6] Ten rachunek *nie jest* pełnym dowodem, ale to jest kurs OpenGL-a, nie Analizy II.

13. Druga aplikacja

Druga aplikacja rysuje powierzchnie zakrzywione zbudowane z płatów Béziera. Dla odmiany i poszerzenia horyzontów jest to aplikacja biblioteki GLFW. Wykorzystamy w niej wiele procedur z pierwszej aplikacji, bez żadnych zmian lub ze zmianami wymuszonymi przez inny interfejs aplikacji tej biblioteki. Ale to nie są wielkie zmiany.

Płaty powierzchni Béziera

Opis reprezentacji Béziera wielomianowych płatów parametrycznych można znaleźć w książkach na ten temat, na przykład w mojej [20]. Piękna matematyka będąca podstawą tej reprezentacji może wystraszyć wiele osób[1], więc nie powinienem jej tu przedstawiać zbyt szczegółowo. Z drugiej strony, bez tej matematyki nie ma mowy o eleganckich i efektywnych algorytmach umożliwiających wykonywanie obrazów takich płatów i w będących w moim posiadaniu książkach o OpenGL-u nie takie algorytmy są opisane. Dlatego upraszam Czytelników o zgodę na poniższy (zgniły, jakże by inaczej) kompromis.

Podstawą reprezentacji Béziera krzywych i płatów parametrycznych są wielomiany bazowe Bernsteina, określone wzorem

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \dots, n. \tag{13.10}$$

Krzywą Béziera stopnia n określa się wzorem

$$\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{p}_i B_i^n(t), \quad t \in [a, b]. \tag{13.11}$$

Odcinek $[a, b]$ osi liczbowej jest dziedziną parametryzacji $\mathbf{p}$. Każdej liczbie t z tego przedziału odpowiada pewien punkt $\mathbf{p}(t)$ krzywej. Krzywa ta znajduje się w przestrzeni, do której należą wektory $\mathbf{p}_0, \dots, \mathbf{p}_n$, zwane punktami kontrolnymi.

Odcinek (przedział) $[a, b]$ może być wybrany dowolnie, ale zazwyczaj przyjmuje się, że jest to odcinek $[0, 1]$. Wszystkie wielomiany Bernsteina przyjmują na tym odcinku wartości nieujemne, co w powiązaniu z faktem, że ich suma (dla każdego $t \in \mathbb{R}$) jest równa 1 sprawia, że jeśli $[a, b] = [0, 1]$, to krzywa jest położona w otoczce wypukłej zbioru swoich punktów kontrolnych.

[1] nad czym głęboko ubolewam

Łamana kontrolna jest złożona z n odcinków, a jej kolejnymi wierzchołkami są punkty $\mathbf{p}_0, \ldots, \mathbf{p}_n$. Jest ona przybliżeniem krzywej Béziera; ponieważ $\mathbf{p}(0) = \mathbf{p}_0$ oraz $\mathbf{p}(1) = \mathbf{p}_n$, punkty końcowe krzywej są końcami łamanej kontrolnej. Jeśli f oznacza dowolne przekształcenie afiniczne, to obraz $f(\mathbf{p})$ krzywej $\mathbf{p}$ jest reprezentowany przez punkty kontrolne $f(\mathbf{p}_0), \ldots, f(\mathbf{p}_n)$. Zatem, aby poddać krzywą Béziera dowolnemu przekształceniu afinicznemu (np. obrócić ją lub przesunąć), wystarczy zastosować to przekształcenie do jej punktów kontrolnych.

Tensorowy płat Béziera stopnia (n, m) jest określony wzorem

$$\mathbf{p}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{p}_{ij} B_i^n(u) B_j^m(v), \quad u \in [a, b],\ v \in [c, d]. \tag{13.12}$$

Dziedziną parametryzacji płata jest zatem prostokąt $[a, b] \times [c, d]$, przy czym zazwyczaj przyjmuje się, że $a = c = 0$, $b = d = 1$, a więc dziedzina ta jest kwadratem jednostkowym, $[0, 1]^2$. Krzywą można widzieć jako powyginany i porozciągany odcinek i podobnie płat tensorowy jest powyginanym i porozciąganym kwadratem.

W zbiorze punktów kontrolnych płata, $\mathbf{p}_{ij}$, wyróżniamy $n + 1$ kolumn (ciągów $\mathbf{p}_{i0}, \ldots, \mathbf{p}_{im}$) oraz $m + 1$ wierszy (ciągów $\mathbf{p}_{0j}, \ldots, \mathbf{p}_{nj}$), które wygodnie jest przedstawiać jako łamane. W ten sposób powstaje siatka kontrolna — odpowiednik łamanej kontrolnej krzywej. Jej kształt określa kształt płata. Podobnie jak dla krzywej, aby otrzymać obraz płata Béziera w dowolnym przekształceniu afinicznym, wystarczy przekształcić jego siatkę kontrolną.

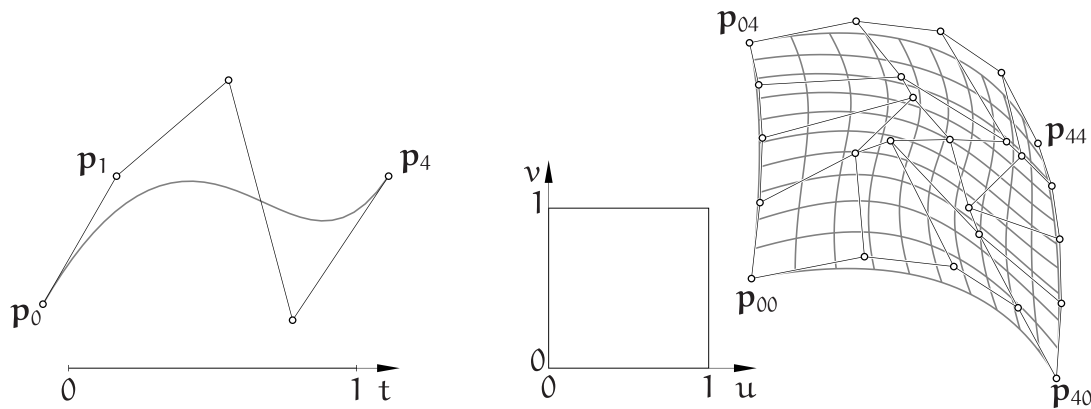
Wzór definiujący płat możemy przepisać w postaci

$$\mathbf{p}(u, v) = \sum_{i=0}^{n} \underbrace{\left(\sum_{j=0}^{m} \mathbf{p}_{ij} B_j^m(v) \right)}_{\mathbf{q}_i} B_i^n(u) = \sum_{i=0}^{n} \mathbf{q}_i B_i^n(u). \tag{13.13}$$

Wynika z niej, że mając dane liczby u, v, możemy obliczyć $n + 1$ punktów, $\mathbf{q}_0, \ldots, \mathbf{q}_n$, z których każdy jest punktem krzywej Béziera stopnia m reprezentowanej przez odpowiednią kolumnę siatki kontrolnej. Otrzymamy w ten sposób reprezentację Béziera krzywej stałego parametru v płata; punkt $\mathbf{p}(u, v)$ płata jest punktem tej krzywej, odpowiadającym danemu u.[2]

Zanim zajmiemy się algorytmami znajdowania punktów krzywych Béziera, zbadajmy wynikające z powyższych spostrzeżeń (niektóre) własności płatów

[2] Alternatywnie, zamiast kolumn możemy potraktować *wiersze* siatki kontrolnej jak łamane kontrolne krzywych Béziera stopnia n; otrzymamy w ten sposób reprezentację Béziera krzywej stałego parametru u płata i możemy obliczać punkty płata jako punkty tej krzywej.



Rysunek 13.1: Krzywa Béziera i tensorowy płat Béziera

i (niektóre) ich konsekwencje praktyczne. Zauważamy, że

$$\begin{aligned} \mathbf{p}(u,0) &= \sum_{i=0}^{n} \mathbf{p}_{i0} B_i^n(u), \quad \mathbf{p}(u,1) = \sum_{i=0}^{n} \mathbf{p}_{im} B_i^n(u), \\ \mathbf{p}(0,v) &= \sum_{j=0}^{m} \mathbf{p}_{0j} B_j^m(v), \quad \mathbf{p}(1,v) = \sum_{j=0}^{m} \mathbf{p}_{nj} B_j^m(v), \end{aligned}$$

a zatem skrajne wiersze i kolumny siatki kontrolnej wyznaczają cztery krzywe brzegowe płata o dziedzinie $[0,1]^2$, co więcej, narożniki siatki kontrolnej są narożnikami płata: $\mathbf{p}(0,0) = \mathbf{p}_{00}$, $\mathbf{p}(1,0) = \mathbf{p}_{n0}$, $\mathbf{p}(0,1) = \mathbf{p}_{0m}$, $\mathbf{p}(1,1) = \mathbf{p}_{nm}$.

Do wykonania obrazu potrzebny jest algorytm obliczania punktu $\mathbf{p}(u,v)$ dla danych liczb u, v oraz wektora $\mathbf{n}(u,v) = \mathbf{p}_u(u,v) \wedge \mathbf{p}_v(u,v)$, tj. iloczynu wektorowego pochodnych cząstkowych parametryzacji $\mathbf{p}$, który jest wektorem normalnym płata w punkcie $\mathbf{p}(u,v)$; wektor ten natychmiast po obliczeniu unormujemy, aby otrzymać wektor jednostkowy. Potrzebujemy zatem algorytmu obliczania pochodnych cząstkowych płata Béziera. Wyprowadzanie go zaczniemy od spojrzenia na krzywe.

Aby obliczać punkty krzywej Béziera, oznaczmy $s = 1 - t$ i rozpiszmy wzór (13.11):

$$\begin{aligned} \mathbf{p}(t) &= \mathbf{p}_0 \tbinom{n}{0} s^n + \mathbf{p}_1 \tbinom{n}{1} t s^{n-1} + \cdots + \mathbf{p}_{n-1} \tbinom{n}{n-1} t^{n-1} s + \mathbf{p} \tbinom{n}{n} t^n \\ &= \Big(\ldots \big(\mathbf{p}_0 \tbinom{n}{0} s + \mathbf{p}_1 \tbinom{n}{1} t \big) s + \cdots + \mathbf{p}_{n-1} \tbinom{n}{n-1} t^{n-1} \Big) s + \mathbf{p} \tbinom{n}{n} t^n. \end{aligned}$$

Na podstawie tego wzoru możemy obliczyć punkt $\mathbf{p}(t)$ jako wartość (wektorowego) wielomianu zmiennej s, korzystając ze schematu Hornera. W tym celu trzeba obliczyć (wektorowe) współczynniki $\mathbf{p}_i \binom{n}{i} t^i$ tego wielomianu w bazie potęgowej.

Punkty kontrolne $\mathbf{p}_i$ mamy dane, kolejne potęgi parametru t obliczymy w pętli, zaś do obliczenia współczynników dwumianowych Newtona możemy użyć wzorów

$$\binom{n}{0} = 1, \quad \binom{n}{1} = n, \quad \binom{n}{i+1} = \binom{n}{i}(n-i)/(i+1), \ i = 1, \ldots, n-1.$$

Współczyniki dwumianowe są liczbami całkowitymi; w szczególności iloczyn $\binom{n}{i}(n-i)$ jest zawsze podzielny przez $i+1$ i tu w działaniach stałopozycyjnych nie ma błędów zaokrągleń. Ale może być nadmiar — dla 32-bitowych liczb całkowitych ze znakiem wystąpi on gdy $n \geqslant 30$. W praktycznych zastosowaniach mamy do czynienia z krzywymi i płatami znacznie niższych stopni.

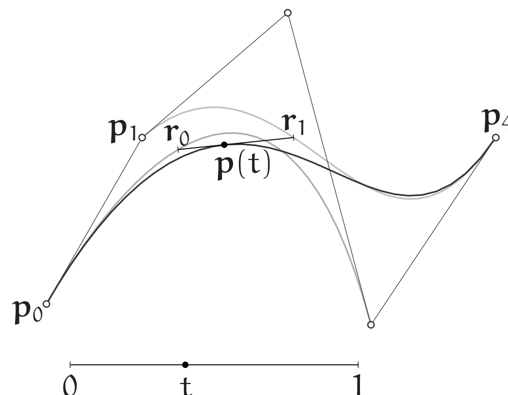
Aby znaleźć punkt krzywej i wektor pochodnej parametryzacji krzywej Béziera, skorzystamy z faktu, że parametryzacja określona wzorem (13.11) dla $n > 0$ ma równoważne przedstawienie w postaci

$$\mathbf{p}(t) = (1-t)\underbrace{\sum_{i=0}^{n-1} \mathbf{p}_i B_i^{n-1}(t)}_{\mathbf{r}_0} + t\underbrace{\sum_{i=0}^{n-1} \mathbf{p}_{i+1} B_i^{n-1}(t)}_{\mathbf{r}_1} = (1-t)\mathbf{r}_0 + t\mathbf{r}_1, \qquad (13.14)$$

a pochodna parametryzacji w punkcie t wyraża się wzorem

$$\mathbf{p}'(t) = \sum_{i=0}^{n-1} n(\mathbf{p}_{i+1} - \mathbf{p}_i) B_i^{n-1}(t) = n(\mathbf{r}_1 - \mathbf{r}_0). \qquad (13.15)$$

Obliczenie punktu $\mathbf{p}(t)$ i wektora $\mathbf{p}'(t)$ dla danego t można zatem zacząć od znalezienia punktów $\mathbf{r}_0$ i $\mathbf{r}_1$ położonych na dwóch krzywych Béziera stopnia $n-1$: pierwsza z nich jest reprezentowana przez punkty kontrolne $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$, a druga przez punkty $\mathbf{p}_1, \ldots, \mathbf{p}_n$ (rys. 13.2).



Rysunek 13.2: Obliczanie punktu i wektora pochodnej krzywej Béziera

Teraz zobaczmy, jak można obliczyć punkt tensorowego płata Béziera i pochodne cząstkowe jego parametryzacji. Pochodna cząstkowa względem u jest pochodną

krzywej stałego parametru v, będącej krzywą Béziera reprezentowaną przez punkty $\mathbf{q}_0, \ldots, \mathbf{q}_n$ (zobacz wzór (13.13)). Natomiast pochodną cząstkową względem v możemy otrzymać, jeśli oprócz punktów $\mathbf{q}_i$ (będących punktami krzywych Béziera stopnia m reprezentowanych przez kolumny siatki kontrolnej płata) dla ustalonego parametru v obliczymy wektory $\mathbf{q}'_0, \ldots, \mathbf{q}'_n$ — pochodne tych krzywych w punkcie v. Mamy zatem

$$\mathbf{p}_u(u,v) = n\sum_{i=0}^{n-1}\sum_{j=0}^{m}(\mathbf{p}_{i+1,j} - \mathbf{p}_{ij})B_i^{n-1}(u)B_j^m(v) = n(\mathbf{r}_1 - \mathbf{r}_0), \tag{13.16}$$

$$\mathbf{p}_v(u,v) = m\sum_{i=0}^{n}\sum_{j=0}^{m-1}(\mathbf{p}_{i,j+1} - \mathbf{p}_{ij})B_i^{n}(u)B_j^{m-1}(v) = \sum_{i=0}^{n}\mathbf{q}'_i B_i^n(u), \tag{13.17}$$

przy czym punkty $\mathbf{r}_0$ i $\mathbf{r}_1$ znajdziemy podczas obliczania punktu $\mathbf{p}(u,v)$ na krzywej stałego parametru v.

Obliczając pochodne cząstkowe płata Béziera, możemy pominąć czynniki n i m; pochodne są potrzebne do obliczenia jednostkowego wektora normalnego płata, a zatem ich długości są nieistotne — istotne są tylko kierunki i zwroty tych wektorów, bo natychmiast po obliczeniu iloczynu wektorowego odpowiednich wektorów różnic (przy użyciu dostępnej w GLSL funkcji `cross`) iloczyn ten unormujemy. Implementacje w GLSL algorytmów tu opisanych są pokazane na listingach 13.5 i 13.6.

Wymierne płaty Béziera

Wspomnijmy w tym miejscu o wymiernych płatach Béziera. Otrzymujemy je, konstruując płaty wielomianowe (takie jak opisane wyżej) w przestrzeni $\mathbb{R}^4$; ich punkty kontrolne są wektorami współrzędnych jednorodnych punktów w $\mathbb{R}^3$. Punkt $\mathbf{P}(u,v)$ płata wielomianowego w $\mathbb{R}^4$ (tzw. płata jednorodnego) traktujemy jak wektor współrzędnych jednorodnych punktu $\mathbf{p}(u,v)$ płata wymiernego w przestrzeni trójwymiarowej. Współrzędne kartezjańskie tego punktu jak zwykle otrzymamy przez podzielenie pierwszych trzech współrzędnych jednorodnych przez czwartą współrzędną (wagową). Jeśli wszystkie punkty kontrolne mają taką samą współrzędną wagową (która nie może być zerem), to mamy zwykły wielomianowy tensorowy płat Béziera. Ale jeśli dopuszczamy niejednakowe współrzędne wagowe punktów kontrolnych, to mamy istotnie szerszą klasę płatów powierzchni[3].

[3]Zawierającą m.in. wszystkie kwadryki, tj. powierzchnie drugiego stopnia, a także powierzchnie obrotowe, których tworzące mają parametryzacje wymierne.

Siatkę kontrolną płata wymiernego możemy narysować jako obiekt trójwymiarowy; w tym celu obliczamy współrzędne jej wierzchołków, dzieląc pierwsze trzy współrzędne przez współrzędną wagową. Modelując płaty wymierne, postępujemy odwrotnie: rozmieszczamy punkty kontrolne w przestrzeni trójwymiarowej i dobieramy ich wagi; na podstawie punktu w $\mathbb{R}^3$ i wagi możemy łatwo obliczyć punkty kontrolne płata jednorodnego w $\mathbb{R}^4$.

Dla płata wymiernego również potrzebujemy obliczać punkty i wektory normalne. Podam przepis (bez dowodu) jak to czynić. Jeśli wektory $\mathbf{P}(u,v)$, $\mathbf{P}_u(u,v)$ i $\mathbf{P}_v(u,v)$ oznaczają odpowiednio punkt płata jednorodnego i jego pochodne cząstkowe w punkcie (u,v), to iloczyn wektorowy *trzech* wektorów w $\mathbb{R}^4$

$$\mathbf{N}(u,v) = \mathbf{P}(u,v) \wedge \mathbf{P}_u(u,v) \wedge \mathbf{P}_v(u,v)$$

reprezentuje (jednoznacznie) płaszczyznę styczną do płata wymiernego w $\mathbb{R}^3$. W szczególności, biorąc pierwsze trzy współrzędne tego iloczynu, otrzymamy wektor normalny $\mathbf{n}(u,v)$ tej płaszczyzny[4].

Ponieważ GLSL nie udostępnia standardowej procedury obliczania iloczynu wektorowego w $\mathbb{R}^4$, trzeba taką procedurę napisać samemu. Dla wektorów $\mathbf{a} = [x_a, y_a, z_a, w_a]^T$, $\mathbf{b} = [x_b, y_b, z_b, w_b]^T$, $\mathbf{c} = [x_c, y_c, z_c, w_c]^T$ jest taki wzór:

$$\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c} = \begin{bmatrix} -d_{34}y_c + d_{24}z_c - d_{23}w_c \\ d_{34}x_c - d_{14}z_c + d_{13}w_c \\ -d_{24}x_c + d_{14}y_c - d_{12}w_c \\ d_{23}x_c - d_{13}y_c + d_{12}z_c \end{bmatrix},$$

a w nim występują liczby

$$\begin{aligned} d_{12} &= x_a y_b - y_a x_b, & d_{13} &= x_a z_b - z_a x_b, & d_{14} &= x_a w_b - w_a x_b, \\ d_{23} &= y_a z_b - z_a y_b, & d_{24} &= y_a w_b - w_a y_b, & d_{34} &= z_a w_b - w_a z_b. \end{aligned}$$

Iloczyn wektorowy z uwagi na często stosowany symbol „×” jest nazywany po angielsku *cross product*, dlatego wbudowana procedura obliczająca iloczyn wektorowy w $\mathbb{R}^3$ ma nazwę `cross`. Możemy użyć tej samej nazwy dla podprogramu obliczającego iloczyn wektorowy w $\mathbb{R}^4$, ponieważ podprogram ten ma więcej parametrów, a ich typ i typ wyniku różni się od typu parametrów i wyniku procedury wbudowanej, co umożliwia przeciążenie nazwy.

[4] Jeśli $\mathbf{N}(u,v) = \left[\begin{smallmatrix} \mathbf{n} \\ w \end{smallmatrix}\right]$, to punkt $\mathbf{q} = \frac{-w}{\mathbf{n}^T\mathbf{n}}\mathbf{n}$ jest punktem tej płaszczyzny położonym najbliżej początku układu współrzędnych w $\mathbb{R}^3$ — oczywiście nie musi on leżeć na płacie $\mathbf{p}$.

Listing 13.1: Procedura obliczania iloczynu wektorowego w $\mathbb{R}^4$

GLSL

```
1: vec4 cross ( vec4 a, vec4 b, vec4 c )
2: {
3:   float d12, d13, d14, d23, d24, d34;
4:
5:   d12 = a.x*b.y-a.y*b.x;  d13 = a.x*b.z-a.z*b.x;  d14 = a.x*b.w-a.w*b.x;
6:   d23 = a.y*b.z-a.z*b.y;  d24 = a.y*b.w-a.w*b.y;  d34 = a.z*b.w-a.w*b.z;
7:   return vec4 ( -d34*c.y+d24*c.z-d23*c.w, d34*c.x-d14*c.z+d13*c.w,
8:                 -d24*c.x+d14*c.y-d12*c.w, d23*c.x-d13*c.y+d12*c.z );
9: } /*cross*/
```

Szadery

Program szaderów do rysowania płatów Béziera korzysta z mechanizmu rozdrabniania; w pamięci GPU umieścimy tablicę z punktami kontrolnymi i będziemy rysować płat czworokątny. Jego dziedziną jest kwadrat jednostkowy $[0,1]^2$, dokładnie taki, jaki standardowo przyjmuje się za dziedzinę tensorowych płatów Béziera. Etap rozdrabniania podzieli tę dziedzinę na trójkąty, które przekaże do dalszych etapów potoku przetwarzania grafiki. Zadaniem szadera rozdrabniania jest obliczenie, dla podanego wierzchołka trójkątów w rozdrobnionej dziedzinie, odpowiedniego punktu płata Béziera i wektora normalnego płata w tym punkcie.

Opisane w znanych mi książkach o OpenGL-u procedury rysowania płatów Béziera zakładają, że punkty kontrolne płatów są dane w VAO, tj. w tablicy wierzchołków, z której poszczególne wierzchołki z odpowiednimi atrybutami są wybierane przez etap pobierania wierzchołków. Ten mechanizm jest wygodny i elastyczny, ale ma podstawowe ograniczenie: małą maksymalną liczbę wierzchołków płata. Limit liczby wierzchołków płata można poznać, wykonując instrukcję

```
glGetIntegerv ( GL_MAX_PATCH_VERTICES, &n );
```

W implementacji, której używam, płat może mieć co najwyżej 32 wierzchołki, co wystarczy do reprezentowania płatów Béziera stopnia $(5,4)$, ale już nie $(5,5)$.[5] Dlatego zastosujemy inne rozwiązanie: tablicę punktów kontrolnych umieścimy w bloku zmiennych jednolitych, a w VAO umieścimy *dowolne* punkty. Szader rozdrabniania będzie brał punkty kontrolne z tej zmiennej jednolitej, a nie ze zmiennej wbudowanej (tablicy) gl_in, której zawartość zignoruje. Tracimy przy

[5] Płat stopnia (n,m) ma $(n+1)(m+1)$ punktów kontrolnych.

tym nieco elastyczności, bo pobieranie wierzchołków musimy oprogramować samemu i raczej nie zrobimy tego dla *wszystkich* reprezentacji liczb (dopuścimy tylko liczby zmiennopozycyjne pojedynczej precyzji — chyba, że ktoś sobie doprogramuje inne możliwości). Z drugiej strony etap pobierania wierzchołków narzuca używanie współrzędnych jednorodnych (wektorów w $\mathbb{R}^4$), a tak możemy osobno oprogramować przetwarzanie płatów w $\mathbb{R}^2$, $\mathbb{R}^3$ i $\mathbb{R}^4$, otrzymując znacznie prostsze i trochę szybsze procedury dla pierwszych dwóch przypadków.

Napiszemy program składający się z pięciu szaderów; wypełnią one wszystkie programowalne etapy w potoku przetwarzania grafiki. Program ten ma umożliwiać rysowanie wielu płatów jednocześnie, przy czym mogą to być wielomianowe płaty płaskie oraz wielomianowe i wymierne płaty w przestrzeni trójwymiarowej. Punkty kontrolne umieścimy w tablicy w bloku zmiennych jednolitych; dodatkowo umożliwimy użycie dodatkowej tablicy indeksów do tablicy punktów kontrolnych. Dzięki niej można zmniejszyć długość tablicy punktów kontrolnych — przez wyeliminowanie kopii punktów wspólnych dla dwóch lub większej liczby płatów (np. wchodzących w skład wspólnego wiersza lub kolumny siatek kontrolnych płatów mających wspólną krzywą brzegową).

Listing 13.2: Szader wierzchołków

GLSL

```
1: #version 420 core
2:
3: layout(location=0) in vec4 in_Position;
4:
5: out VertInstance { int instance; } Out;
6:
7: void main ( void )
8: {
9:   Out.instance = gl_InstanceID;
10:  gl_Position = in_Position;
11: } /*main*/
```

Szader wierzchołków tego programu jest pokazany na listingu 13.2. Wypełnia on obowiązek przypisania czegokolwiek zmiennej wbudowanej `gl_Position`, przy czym dalej jej wartość zostanie zignorowana. Ważniejsze jest przekazanie (w linii 9) numeru instancji. Aby spowodować rysowanie, aplikacja wywoła procedurę `glDrawArraysInstanced`, której ostatni parametr określa *liczbę instancji* rysowanego prymitywu: jeśli jest ona równa n, to poszczególne instancje są ponumerowane od 0 do $n-1$. Numer instancji jest podawany szaderowi wierzchołków w zmiennej wbudowanej `gl_InstanceID` (typu int), a ten kopiuje ją do zmiennej wyjściowej `VertInstance.instance`.

Na listingu 13.3 jest pokazany szader sterowania rozdrabnianiem. Jego zadaniem jest przypisanie wartości wszystkim czterem elementom tablicy `gl_TessLevelOuter` i obu elementom tablicy `gl_TessLevelInner`. Wszystkie sześć zmiennych otrzyma tę samą wartość, przy czym zostanie ona wzięta ze zmiennej jednolitej `BezTessLevel`, do której przypisanie wykonała aplikacja[6]. Zmienna ta *nie jest* częścią żadnego nazwanego bloku, a zatem znajduje się w domyślnym bloku zmiennych jednolitych programu. Sposób nadawania wartości takim zmiennym będzie przedstawiony dalej, w opisie procedur w C, działających na CPU. Tymczasem odnotujmy, że takie zmienne są widoczne tylko dla *jednego* programu szaderów (choć mają do nich dostęp wszystkie szadery wchodzące w skład tego programu).

Listing 13.3: Szader sterowania rozdrabnianiem

GLSL

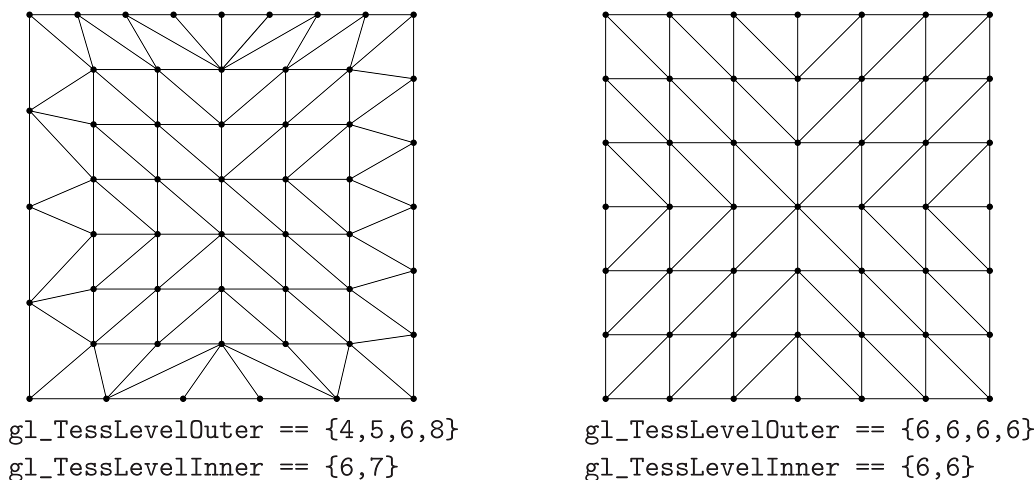
```glsl
1: #version 420 core
2:
3: layout (vertices=4) out;
4:
5: in VertInstance { int instance; } In[];
6:
7: out TCInstance { int instance; } Out[];
8:
9: uniform int BezTessLevel;
10:
11: void main ( void )
12: {
13:   if ( gl_InvocationID == 0 ) {
14:     gl_TessLevelOuter[0] = gl_TessLevelOuter[1] =
15:     gl_TessLevelOuter[2] = gl_TessLevelOuter[3] = BezTessLevel;
16:     gl_TessLevelInner[0] = gl_TessLevelInner[1] = BezTessLevel;
17:     Out[gl_InvocationID].instance = In[gl_InvocationID].instance;
18:   }
19:   gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
20: } /*main*/
```

Szader sterowania rozdrabnianiem przekazuje dalej numer instancji i pracowicie kopiuje niepotrzebny dalej wektor współrzędnych jednorodnych wierzchołka. Szader ten ma dostęp do całej tablicy wierzchołków płata (która tu ma długość 4

[6]Zamiast szadera sterowania rozdrabnianiem możemy użyć procedury `glPatchParameterfv`, zgodnie z opisem w poprzednim rozdziale. Usunięcie szadera sterowania rozdrabnianiem wymagałoby przerobienia szadera wierzchołków lub rozdrabniania w celu uzgodnienia zewnętrznej nazwy zmiennej użytej do przekazania numeru instancji.

— linia 3) i jest dla płata wywołany czterokrotnie. Zmienna wbudowana gl_InvocationID za każdym razem ma inną wartość, od 0 do 3. Zmienna wyjściowa Out musi być tablicą (o długości 4), ale numer instancji wystarczy przekazać tylko w jednym (pierwszym) jej elemencie.

Przykłady podziału dziedziny na trójkąty po podaniu różnych parametrów są pokazane na rysunku 13.3. W ogólności etap rozdrabniania dziedziny nie wytwarza triangulacji Delaunay (co byłoby świetne, ale chyba zbyt kosztowne), ale to, co wytwarza, może być.



Rysunek 13.3: Podziały kwadratowej dziedziny płata

W tej aplikacji nie będziemy zajmować się adaptacyjnym dostosowaniem parametrów rozdrabniania do wielkości obrazu płata, ale warto w tym miejscu uczynić pewną dygresję na ten temat. Zazwyczaj mamy do czynienia z obiektami składającymi się z wielu płatów, czyli z tzw. powierzchniami sklejanymi. Wchodzące w skład takiej powierzchni płaty mają wspólne brzegi, nieraz też są połączone gładko. Podczas rysowania płaty przybliżamy powierzchniami zbudowanymi z (dostatecznie małych) płaskich trójkątów i wtedy brzegi tych płatów są przybliżane łamanymi. Jeśli dwa płaty wchodzące w skład większej powierzchni mają wspólną krzywą brzegową, to łamane przybliżające tę krzywą, złożone z boków odpowiednich trójkątów, muszą być identyczne. W przeciwnym razie powierzchnia przedstawiona na obrazie będzie miała widoczną szczelinę. Pisząc szadery sterowania rozdrabnianiem, które dobierają adaptacyjnie parametry rozdrabniania poszczególnych płatów, trzeba to mieć na uwadze.

Szader rozdrabniania jest taki długi, że zmieścił się dopiero na pięciu listingach, 13.4–13.8. Pierwszy z nich pokazuje deklaracje wejścia/wyjścia i zmiennych

Listing 13.4: Szader rozdrabniania — używane zmienne

GLSL

```
#version 420 core

#define MAX_DEG 10

layout(quads,equal_spacing,cw) in;

in TCInstance { int instance; } In[];

out GVertex {
    vec4 Colour;
    vec3 Position;
    vec3 Normal;
  } Out;

uniform bool BezNormals;

uniform CPoints {
    float cp[1];
  } cp;

uniform CPIndices {
    int cpi[1];
  } cpi;

uniform BezPatch {
    int  dim, udeg, vdeg;
    int  stride_u, stride_v, stride_p, stride_q, nq;
    bool use_ind;
    vec4 Colour;
  } bezp;

uniform TransBlock {
    mat4 mm, mmti, vm, pm, mvpm;
    vec4 eyepos;
  } trb;
```

jednolitych używanych przez szader. W linii 5 jest podana informacja (dla etapu rozdrabniania dziedziny), że dziedzina, którą należy podzielić (na trójkąty) jest czworokątem, przy czym boki dziedziny mają być podzielone równomiernie, a orientacja trójkątów przekazywanych dalej ma być zgodna z ruchem wskazówek zegara. Tablica TCInstance (linia 7) zawiera w *pierwszym* elemencie numer instancji, który jest interpretowany jako numer płata do narysowania.

Blok wyjściowy GVertex, który będzie wejściem dla szadera geometrii, zawiera trzy znajome atrybuty wierzchołka: kolor (który tu pobierzemy z bloku zmiennych jednolitych BezPatch) oraz położenie wierzchołka w przestrzeni i wektor normalny płata, podane w układzie współrzędnych świata.

Zmienna jednolita BezNormals określa sposób obliczania wektorów normalnych przez program. Jeśli ma ona wartość true, to wektor normalny powierzchni ma być obliczany przez szader rozdrabniania i jest to „prawdziwy" wektor normalny powierzchni w obliczonym przez szader punkcie płata. Jeśli wartością tej zmiennej jest false, to zadanie to wykona szader geometrii; obliczy on wektor normalny płaszczyzny trójkąta, którego wierzchołki, położone na płacie, obliczył szader rozdrabniania.

W bloku CPoints (linie 17–19) jest tablica cp ze współrzędnymi punktów kontrolnych. Zadeklarowana długość tablicy jest nieistotna; za jej ustalenie i dopilnowanie, aby program nie wychodził poza zakres jej indeksów jest odpowiedzialna aplikacja.

Tablica cpi w bloku CPIndices (linie 21–23) zawiera indeksy do tablicy punktów kontrolnych. Jej faktyczna długość też jest ustalana przez aplikację.

Blok BezPatch zawiera opis zbioru płatów Béziera do narysowania. Wartość zmiennej dim, 2, 3 lub 4, jest liczbą współrzędnych punktów kontrolnych. Zmienne udeg i vdeg przechowują stopnie wszystkich płatów ze względu na parametry u i v. Zmienne stride_u, stride_v, stride_p, stride_q i nq służą do uzyskania dostępu do właściwych punktów kontrolnych, co będzie opisane dalej. Zmienna use_ind określa, czy po punkty kontrolne (do tablicy CPoints.cp) należy sięgać bezpośrednio, czy też posługując się indeksami z tablicy CPIndices.cpi. W zmiennej Colour jest podany kolor płatów.

Blok TransBlock w liniach 32–35, dobrze znany z pierwszej aplikacji, zawiera macierze przekształceń potrzebnych do rzutowania punktu i położenie obserwatora w układzie świata.

Na listingu 13.5 są podane dwie procedury realizujące algorytm obliczania punktu i wektora normalnego płata płaskiego, zawartego w płaszczyźnie xy. Procedura BCHorner2f oblicza punkt położony na krzywej stopnia n. Jej punkty kontrolne są podane w tablicy bcp. Parametr t podaje wartość zadanego argumentu parametryzacji t. Obliczony punkt krzywej ma być przypisany do parametru wyjściowego p.

Algorytm zrealizowany w procedurze BCHorner2f jest schematem Hornera przedstawionym wcześniej; wartości kolejno przypisywane zmiennej b to współczynniki dwumianowe, a zmiennej d są przypisywane kolejne potęgi parametru t.

Procedura BPHorner2f (linie 16–40) otrzymuje jako parametry u, v współrzędne u, v punktu w dziedzinie płata. Parametry wyjściowe pos i nv są zmiennymi, do których należy przypisać obliczony punkt i wektor normalny płata. Zauważmy, że w przypadku płata płaskiego wektor normalny jest stały i do jego obliczenia pochodne parametryzacji nie są potrzebne.

Listing 13.5: Obliczanie punktu i wektora normalnego płaskiego płata

GLSL

```
1: void BCHorner2f ( int n, vec2 bcp[MAX_DEG+1], float t, out vec2 p )
2: {
3:   int   i, b;
4:   float s, d;
5:   vec2  q;
6:
7:   s = 1.0-t;  d = t;  b = n;
8:   q = bcp[0];
9:   for ( i = 1; i <= n; i++ ) {
10:    q = s*q + (b*d)*bcp[i];
11:    d *= t;  b = (b*(n-i))/(i+1);
12:  }
13:  p = q;
14: } /*BCHorner2f*/
15:
16: void BPHorner2f ( float u, float v, out vec4 pos, out vec4 nv )
17: {
18:   vec2 p[MAX_DEG+1], q[MAX_DEG+1], r;
19:   vec4 Pos, Normal;
20:   int  i, j, k, l, i0;
21:
22:   if ( bezp.use_ind )
23:     i0 = In[0].instance*bezp.stride_p;
24:   else {
25:     i = In[0].instance / bezp.nq;
26:     j = In[0].instance % bezp.nq;
27:     i0 = i*bezp.stride_p + j*bezp.stride_q;
28:   }
29:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
30:     if ( bezp.use_ind ) {
31:       for ( j = 0;  j <= bezp.vdeg;  j++, k++ ) {
```

```
32:         l = 2*cpi.cpi[i0+k];
33:         p[j] = vec2 ( cp.cp[l], cp.cp[l+1] );
34:       }
35:     }
36:     else {
37:       for ( j = 0, l = i0+i*bezp.stride_u;
38:             j <= bezp.vdeg;
39:             j++, l += bezp.stride_v )
40:         p[j] = vec2 ( cp.cp[l], cp.cp[l+1] );
41:       k += bezp.stride_u;
42:     }
43:     BCHorner2f ( bezp.vdeg, p, v, q[i] );
44:   }
45:   BCHorner2f ( bezp.udeg, q, u, r );
46:   pos = vec4 ( r.xy, 0.0, 1.0 );
47:   nv = vec4 ( 0.0, 0.0, 1.0, 0.0 );
48: } /*BPHorner2f*/
```

W pętli w liniach 29–44 obliczane są punkty $\mathbf{q}_0, \ldots, \mathbf{q}_n$ krzywych Béziera reprezentowanych przez kolumny siatki kontrolnej, dla podanego argumentu parametryzacji v. Wywołanie procedury BCHorner2f w linii 45 oblicza punkt krzywej Béziera reprezentowanej przez te punkty, dla podanego parametru u. Do obliczonych w ten sposób współrzędnych x, y w linii 46 doczepiane są współrzędne $z = 0$ i $w = 1$, a w linii 47 jest (trywialne) wygenerowanie wektora normalnego płata.

Zwróćmy uwagę na pobieranie punktów kontrolnych z tablicy. Jest to robione jednym z dwóch sposobów. Pierwszy sposób (linie 31–34) korzysta z tablicy indeksów. Kolejne elementy tej tablicy, zaczynając od pozycji i_0, będącej iloczynem numeru instancji i wartości zmiennej jednolitej `bezp.stride_p`, numerują punkty w kolejnych $n + 1$ kolumnach siatki kontrolnej; każda kolumna składa się z $m + 1$ punktów. Indeksy są w linii 32 mnożone przez 2, bo każdy punkt kontrolny płata płaskiego składa się z dwóch współrzędnych.

Sposób drugi zakłada, że tablica punktów kontrolnych zawiera kolejne kolumny prostokątnej macierzy, przy czym wymiary tej macierzy (liczba kolumn i wierszy) mogą być dowolnie duże. Pierwsza współrzędna pierwszego punktu siatki kontrolnej płata Béziera jest na pozycji i_0, obliczanej w liniach 25–27. Przyjęte tu jest założenie, że każda kolumna tablicy zawiera kolumny należące do siatek n_q płatów Béziera (ta liczba jest wartością zmiennej `bezp.nq`). Początki kolumn w siatkach kolejnych *płatów* występują co `bezp.stride_q` miejsc w tablicy,

natomiast kolejne kolumny *tablicy* dzieli odległość `bezp.stride_p` miejsc[7].

Przejście do kolejnego punktu w kolumnie siatki kontrolnej wymaga dodania przyrostu `bezp.stride_v`, zaś przejście do kolejnego punktu w wierszu odbywa się z krokiem `bezp.stride_u`. Jeśli w tablicy mamy tylko upakowane punkty kontrolne jednego płata stopnia (n, m), to wartość zmiennej `bezp.stride_v` powinna być równa 2 (bo płat jest płaski), a zmienna `bezp.stride_u` powinna mieć wartość $2(m+1)$.

Procedury realizujące obliczanie punktu i wektora normalnego płata trójwymiarowego są przedstawione na listingu 13.6. W przypadku trój- i czterowymiarowym wektor normalny trzeba obliczyć przy użyciu pochodnych parametryzacji. Dlatego pierwsza procedura oblicza punkt i wektor pochodnej krzywej Béziera. Druga procedura posługuje się pierwszą w celu obliczenia punktu i pochodnych cząstkowych. Zgodnie z wcześniej podanym opisem teoretycznych własności krzywych, procedura `BCHorner3f` oblicza najpierw punkty $\mathbf{r}_0$ i $\mathbf{r}_1$ dwóch krzywych stopnia $n-1$.

Listing 13.6: Obliczanie punktu i wektora normalnego płata w $\mathbb{R}^3$

GLSL

```
1: void BCHorner3f ( int n, vec3 bcp[MAX_DEG+1], float t,
2:                   out vec3 p, out vec3 dp )
3: {
4:   int   i, b;
5:   float s, d, bd;
6:   vec3  r0, r1;
7:
8:   n --;    /* przetwarzamy dwie krzywe stopnia n-1 */
9:   s = 1.0-t;  d = t;  b = n;
10:  r0 = bcp[0];  r1 = bcp[1];
11:  for ( i = 1; i <= n; i++ ) {
12:    bd = b*d;
13:    r0 = s*r0 + bd*bcp[i];
14:    r1 = s*r1 + bd*bcp[i+1];
15:    d *= t;  b = (b*(n-i))/(i+1);
16:  }
17:  p = s*r0 + t*r1;
18:  dp = r1 - r0;  /* mnożenie różnicy przez stopień jest niepotrzebne */
19: } /*BCHorner3f*/
```

[7] Taką tablicę z siatkami kontrolnymi wielu płatów można otrzymać na podstawie siatki kontrolnej tensorowego płata B-sklejanego. W grafice, a zwłaszcza w zastosowaniach CAD takie płaty są używane często.

```
20:
21: void BPHorner3f ( float u, float v, out vec4 pos, out vec4 nv )
22: {
23:   vec3 p[MAX_DEG+1], q0[MAX_DEG+1], q1[MAX_DEG+1], r, ru, rv, ruv;
24:   int  i, j, k, l, i0;
25:
26:   .... /* obliczenie indeksu i0 tak jak w procedurze BPHorner2f */
27:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
28:       .... /* tu instrukcje są takie, jak w procedurze BPHorner2f */
29:       .... /* tylko punkty mają 3, a nie 2 współrzędne */
30:     BCHorner3f ( bezp.vdeg, p, v, q0[i], q1[i] );
31:   }
32:   BCHorner3f ( bezp.udeg, q0, u, r, ru );
33:   BCHorner3f ( bezp.udeg, q1, u, rv, ruv );
34:   pos = vec4 ( r, 1.0 );
35:   nv = vec4 ( cross ( ru, rv ), 0.0 );
36: } /*BPHorner3f*/
```

Aby otrzymać współrzędne jednorodne, po obliczeniu punktu płata w $\mathbb{R}^3$ doczepiamy współrzędną wagową 1, a do wektora normalnego doczepiamy 0.

Listing 13.7 przedstawia procedury obliczania punktu i wektora normalnego płata wymiernego. Zasadnicza zmiana w porównaniu z poprzednim przypadkiem polega na zwiększeniu liczby współrzędnych każdego punktu do 4. Ponadto zamiast wbudowanej funkcji cross, obliczającej iloczyn wektorowy w $\mathbb{R}^3$, używamy procedury cross obliczającej iloczyn trzech wektorów w $\mathbb{R}^4$.

Listing 13.7: Obliczanie punktu i wektora normalnego płata wymiernego

```
GLSL
1:   .... /* tu procedura cross z listingu 13.1 */
2:
3: void BCHorner4f ( int n, vec4 bcp[MAX_DEG+1], float t,
4:                   out vec4 p, out vec4 dp )
5: {
6:   .... /* ta procedura jest kopią BCHorner3f, */
7:   .... /* z typem vec3 zamienionym wszędzie na vec4 */
8: } /*BCHorner4f*/
9:
10: void BPHorner4f ( float u, float v, out vec4 pos, out vec4 nv )
11: {
12:   vec4 p[MAX_DEG+1], q0[MAX_DEG+1], q1[MAX_DEG+1], ru, rv;
13:   int  i, j, k, l, i0;
14:
```

```
15:   .... /* obliczenie indeksu i0 tak jak w procedurze BPHorner2f */
16:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
17:       .... /* tu instrukcje są takie, jak w procedurze BPHorner2f */
18:       .... /* tylko punkty mają 4, a nie 2 współrzędne */
19:     BCHorner4f ( bezp.vdeg, p, v, q0[i], q1[i] );
20:   }
21:   BCHorner4f ( bezp.udeg, q0, u, pos, ru );
22:   BCHorner4f ( bezp.udeg, q1, u, rv, nv );
23:   nv = cross ( pos, ru, rv );
24:   pos /= pos.w;
25: } /*BPHorner4f*/
```

Jeszcze jeden szczegół wymaga wyjaśnienia: w linii 24 dzielimy wszystkie współrzędne wektora pos przez współrzędną wagową. Zmiana długości wektora współrzędnych jednorodnych bez zmiany jego kierunku daje tylko inną reprezentację tego samego punktu. Ale chcemy już w tym miejscu otrzymać wektor, którego pierwsze trzy współrzędne są współrzędnymi kartezjańskimi punktu powierzchni. Iloczyn tego wektora i macierzy przekształcenia modelu również zawiera współrzędne kartezjańskie punktu w układzie świata, które zostaną zapamiętane w zmiennej `Out.Position` (listing 13.8, linia 12) i użyte w alternatywnym obliczeniu wektora normalnego przez szader geometrii i w obliczeniu oświetlenia przez szader fragmentów.

Listing 13.8: Procedura `main` szadera rozdrabniania

```glsl
1: void main ( void )
2: {
3:   vec4 pos, nv;
4:
5:   pos = nv = vec4 ( 0.0 );  /* to dla usunięcia ostrzeżenia */
6:   switch ( bezp.dim ) {
7: case 2: BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
8: case 3: BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
9: case 4: BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos, nv );  break;
10:   }
11:   gl_Position = trb.mvpm*pos;
12:   Out.Position = (trb.mm*pos).xyz;
13:   if ( !BezNormals || dot ( nv, nv ) < 1.0e-10 )
14:     Out.Normal = vec3 ( 0.0 );
15:   else {
16:     nv = trb.mmti*nv;
17:     Out.Normal = normalize ( nv.xyz );
18:   }
```

```
19:   Out.Colour = bezp.Colour;
20: } /*main*/
```

Na listingu 13.8 mamy główną procedurę szadera rozdrabniania. Zależnie od wymiaru płata wywołuje ona jedną z trzech opisanych wcześniej procedur obliczających punkt płata i wektor normalny w tym punkcie. Współrzędne u, v punktu w dziedzinie płata, wygenerowane przez etap rozdrabniania dziedziny, szader otrzymuje jako wartości pól `x` i `y` zmiennej wbudowanej `gl_TessCoord`.

Po obliczeniu (jednorodnej reprezentacji) punktu i wektora normalnego płata, w linii 11 następuje przekształcenie punktu do układu kostki standardowej. W linii 12 obliczane są współrzędne kartezjańskie punktu w układzie świata. W liniach 13–18 dokonujemy przekształcenia wektora normalnego. Mamy tu pewien problem. Jeśli pochodne cząstkowe parametryzacji są liniowo zależne, to obliczony wektor normalny jest wektorem zerowym. W takim przypadku procedura `normalize` zostaje zmuszona do dzielenia przez 0, a to jest karygodne[8]. Dlatego zamiast dzielić przez 0 wyprowadzamy wektor zerowy, co stanowi informację, że to szader geometrii ma obliczyć wektor normalny. Szader geometrii obliczy wektor normalny płaszczyzny trójkąta, co skutecznie „załata” niepowodzenie obliczenia „prawdziwego” wektora normalnego powierzchni.

Szader geometrii na listingu 13.9 powstał przez modyfikację szadera z listingu 10.3; niezmienione instrukcje są szare lub zastąpione komentarzami.

Listing 13.9: Szader geometrii

GLSL

```
1: ....  /* dyrektywa #version i kwalifikatory wejścia i wyjścia bez zmian */
2:
3: in GVertex {
4:   vec4 Colour;
5:   vec3 Position;
6:   vec3 Normal;  /* szader może przyjąć "gotowy" wektor normalny */
7: } In[];
8:
9: out NVertex {
10:   vec4 Colour;
11:   vec3 Position;
12:   vec3 Normal;
13: } Out;
14:
15: uniform bool BezNormals;
16:
```

[8] Karą są plamy na obrazie i na reputacji autora programu.

```
17: void main ( void )
18: {
19:   .... /* ten fragment bez zmian */
20:   nv = normalize ( cross ( v1, v2 ) );
21:   for ( i = 0; i < 3; i++ ) {
22:     gl_Position = gl_in[i].gl_Position;
23:     Out.Position = In[i].Position;
24:     if ( !BezNormals || dot ( nv, nv ) < 1.0e-10 )
25:       Out.Normal = nv;
26:     else
27:       Out.Normal = In[i].Normal;
28:     Out.Colour = In[i].Colour;
29:     EmitVertex ();
30:   }
31:   EndPrimitive ();
32: } /*main*/
```

Jeśli zmienna jednolita `BezNormals` ma wartość `true` i obliczony przez szader rozdrabniania wektor normalny jest niezerowy (a ściślej, dłuższy niż 10^{-5}), to szader wyprowadza ten wektor razem z wierzchołkiem trójkąta. W przeciwnym razie szader wyprowadza z wierzchołkiem wektor normalny płaszczyzny trójkąta, który dla trójkąta niezdegenerowanego (o wierzchołkach niewspółliniowych) jest niezerowy i *można* go unormować.

W tej aplikacji użyjemy szadera fragmentów z listingu 10.4, realizującego lambertowski model oświetlenia.

Procedury wprowadzania i rysowania płatów Béziera

Teraz pora na procedury w C; zaczniemy od opisu ogólnych procedur tworzących i wyświetlających płaty Béziera, a aplikacja korzystająca z tych procedur będzie opisana dalej.

Na listingu 13.10 jest pokazany fragment pliku nagłówkowego `bezpatches.h`, w którym jest ograniczenie stopnia płatów (takie samo jak na listingu 13.4)[9] i definicja struktury reprezentującej zespół płatów Béziera do narysowania.

[9]Choć trudno o praktyczny powód do tego, można je zwiększyć, ale co najwyżej do 29. Zauważmy, że szader rozdrabniania, wywołując procedury obliczające punkty krzywych Béziera, kopiuje tablice punktów kontrolnych będące parametrami — kopiowanie długich tablic zabiera czas, zatem warto ich długość, zależną od *maksymalnego* stopnia, ograniczyć. Przeprowadziwszy eksperyment, nie zauważyłem jednak istotnej różnicy szybkości rysowania (mierzonej liczbą klatek na sekundę) dla maksymalnych stopni ustalonych na 5 i 10.

Listing 13.10: Struktura danych reprezentująca płaty Béziera

```c
#define MAX_PATCH_DEGREE 10

typedef struct BezierPatchObjf {
    GLint  udeg, vdeg, dim,
           stride_u, stride_v, stride_p, stride_q, npatches;
    GLuint buf[3];
  } BezierPatchObjf;
```

Pola udeg, vdeg, dim, stride_u, stride_v, stride_p, stride_q i npatches przechowują odpowiednio stopnie n i m płatów, wymiar (tj. liczbę współrzędnych punktów kontrolnych), kroki dla procedur pobierania punktów kontrolnych przez szader rozdrabniania (opisane przy listingu 13.5) i liczbę płatów.

Tablica buf jest miejscem na przechowanie identyfikatorów UBO — buforów, w których są umieszczone bloki zmiennych jednolitych opisujących płaty. W pierwszym z tych buforów ma być blok BezPatch, w drugim blok CPoints z tablicą współrzędnych punktów kontrolnych, a w trzecim, jeśli ma być używany, blok CPIndices, czyli tablica indeksów do tablicy punktów kontrolnych.

Na listingu 13.11 są umieszczone deklaracje zmienych globalnych wykorzystywanych przez opisane tu procedury. Wartości tym zmiennym nadaje procedura LoadMyShaders, opisana dalej jako część aplikacji. W zmiennych bezpbi, cpbi i cpibi są zapamiętane indeksy bloków zmiennych jednolitych. W zmiennych bezpbbp, cpbbp i cpibbp są zapamiętane identyfikatory punktów dowiązania bloków zmiennych jednolitych. W zmiennej bezpbsize jest wielkość (w bajtach) bloku BezPatch, zaś w tablicach bezpbofs, cpbofs i cpibofs są przesunięcia poszczególnych pól względem początków bloków zmienych jednolitych. Kolejne elementy tych tablic przechowują przesunięcia pól o nazwach zapisanych w tablicach zadeklarowanych w treści procedury LoadMyShaders na listingu 13.18.

Listing 13.11: Zmienne globalne

```c
GLuint bezpbi, bezpbbp, cpbi, cpbbp, cpibi, cpibbp;
GLint  bezpbsize, bezpbofs[9], cpbofs[2], cpibofs[1];
GLuint ubeznloc, ubeztloc;
```

W zmiennych ubeznloc i ubeztloc są zapamiętane indeksy zmiennych jednolitych BezNormals i BezTessLevel. Zmienne te znajdują się w domyślnym bloku

zmiennych jednolitych programu złożonego z opisanych wyżej szaderów. Mamy tu nowy element OpenGL-a — sposób uzyskiwania dostępu do takich zmiennych jednolitych, które są widoczne tylko dla jednego programu szaderów (ale „widzą je” wszystkie szadery wchodzące w skład tego programu). Indeksy takich zmiennych jednolitych uzyskuje się za pomocą procedury glGetUniformLocation, wywołanej w naszej aplikacji przez procedurę z listingu 13.18.

Listing 13.12: Procedura SetBezierPatchOptions

C

```
1: void SetBezierPatchOptions ( GLuint program_id,
2:                              GLint normals, GLint tesslevel )
3: {
4:   glUseProgram ( program_id );
5:   glUniform1i ( ubeznloc, normals );
6:   glUniform1i ( ubeztloc, tesslevel );
7:   ExitIfGLError ( "SetBezPatchOptions" );
8: } /*SetBezierPatchOptions*/
```

Procedura SetBezierPatchOptions (listing 13.12) przypisuje wartości zmiennym jednolitym BezNormals i BezTessLevel. Przypisanie wykonuje odpowiednia procedura z serii glUniform*; pierwszy parametr wszystkich tych procedur jest indeksem zmiennej jednolitej. Dla zmiennej (w programie w GLSL) typu bool należy skorzystać z procedury glUniform1i i podać parametr typu GLint o wartości 0 lub 1. Inne procedury z tej serii są opisane w dokumentacji OpenGL-a; zachęcam do przejrzenia tych opisów w ramach ćwiczenia. Ich wywołania trzeba poprzedzić wywołaniem procedury glUseProgram, z parametrem identyfikującym program, do którego należą odpowiednie zmienne jednolite.

Listing 13.13 przedstawia procedurę ConstructBezierPatchDomain, która tworzy VAO reprezentujący dziedzinę płata oraz procedurę DeleteBezierPatchDomain, która go niszczy. Obiekt ten jest pusty, tj. nie zawiera *żadnych* tablic atrybutów wierzchołków. Wywołanie procedury glVertexAttrib1f w linii 7 ma ten skutek, że etap pobierania wierzchołków będzie wysyłał do przetwarzania wierzchołki o współrzędnych kartezjańskich równych 0. Wektor współrzędnych wszystkich tych wierzchołków jest zapisywany w odpowiedniej zmiennej statycznej (zobacz str. 7.12).

Procedura EnterBezierPatches na listingu 13.14 tworzy obiekt reprezentujący płaty Béziera, których wierzchołki są ustawione w prostokątną macierz; wiersze i kolumny siatki kontrolnej każdego z tych płatów powstają przez odrzucenie pewnych wierszy i kolumn tej macierzy. Miejsca w tablicy, z których wybierane są

Listing 13.13: Procedury tworzenia i dealokacji dziedziny płatów

C

```
1: GLuint pd_vao;
2:
3: void ConstructBezierPatchDomain ( void )
4: {
5:   glGenVertexArrays ( 1, &pd_vao );
6:   glBindVertexArray ( pd_vao );
7:   glVertexAttrib1f ( 0, 0.0 );
8:   ExitIfGLError ( "ConstructBezPatchDomain" );
9: } /*ConstructBezierPatchDomain*/
10:
11: void DeleteBezierPatchDomain ( void )
12: {
13:   glBindVertexArray ( 0 );
14:   glDeleteVertexArrays ( 1, &pd_vao );
15:   ExitIfGLError ( "DeleteBezierPatchDomain" );
16: } /*DeleteBezierPatchDomain*/
```

punkty kontrolne, są obliczane na podstawie parametrów `stride_p`, `stride_q`, `stride_u`, `stride_v`, `nq` i numeru instancji (czyli numeru płata w zbiorze reprezentowanym przez tworzony obiekt). Nie jest w tym przypadku używana tablica indeksów. Zero wpisane do bufora w liniach 38–39 reprezentuje wartość false pola `BezPatch.use_ind`, na podstawie którego szader wybiera sposób dostępu do tablicy punktów kontrolnych.

Sprawdzenie poprawności danych obejmuje zbadanie wymiaru, tj. liczby współrzędnych punktu kontrolnego, oraz stopnia płata ze względu na oba parametry; musi on być większy od zera. W liniach 17–20 procedura zapamiętuje potrzebne dane w strukturze typu `BezierPatchObjectf`, której adres poda jako wartość powrotną.

W liniach 21–41 procedura tworzy i wypełnia danymi UBO dla bloku zmiennych jednolitych `BezPatch`. W liniach 43–44 jest tworzony UBO dla bloku `CPoints` z tablicą punktów kontrolnych. Wielkość bufora jest dostosowana do liczby tych punktów i liczby współrzędnych każdego z nich.

Procedura `EnterBezierPatchesElem` na listingu 13.15 tworzy obiekt reprezentujący płaty Béziera, które mogą mieć wspólne lub powtarzające się punkty kontrolne. Dlatego tworzone są trzy UBO — trzeci bufor zawiera tablicę indeksów do tablicy punktów kontrolnych. W linii 29 zmienej `BezPatch.use_ind` przypisywana jest wartość true (czyli 1), aby szader używał tablicy indeksów

Listing 13.14: Procedura EnterBezierPatches

```c
BezierPatchObjf* EnterBezierPatches ( GLint udeg, GLint vdeg, GLint dim,
                      GLint np, GLint nq, int ncp, const GLfloat *cp,
                      GLint stride_p, GLint stride_q,
                      GLint stride_u, GLint stride_v,
                      const GLfloat *colour )
{
  BezierPatchObjf *bp;
  GLint           size, zero = 0;

  if ( dim < 2 || dim > 4 || npatches < 1 ||
       udeg < 1 || udeg > MAX_PATCH_DEGREE ||
       vdeg < 1 || vdeg > MAX_PATCH_DEGREE )
    return NULL;
  bp = malloc ( sizeof(BezierPatchObjf) );
  if ( bp ) {
    memset ( bp, 0, sizeof(BezierPatchObjf) );
    bp->udeg = udeg;  bp->vdeg = vdeg;  bp->dim = dim;
    bp->npatches = np*nq;
    bp->stride_p = stride_p;  bp->stride_q = stride_q;
    bp->stride_u = stride_u;  bp->stride_v = stride_v;
    bp->buf[0] = NewUniformBlockObject ( bezpbsize, bezpbbp );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[0],
                      sizeof(GLint), &bp->dim );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[1],
                      sizeof(GLint), &bp->udeg );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[2],
                      sizeof(GLint), &bp->vdeg );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[3],
                      sizeof(GLint), &bp->stride_u );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[4],
                      sizeof(GLint), &bp->stride_v );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[5],
                      sizeof(GLint), &bp->stride_p );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[6],
                      sizeof(GLint), &bp->stride_q );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[7],
                      sizeof(GLint), &nq );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[8],
                      sizeof(GLint), &zero );
    glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[9],
                      4*sizeof(GLfloat), colour );
    size = ncp*dim*sizeof(GLfloat);
```

```
43:     bp->buf[1] = NewUniformBlockObject ( cpbofs[0]+size, cpbbp );
44:     glBufferSubData ( GL_UNIFORM_BUFFER, cpbofs[0], size, cp );
45:     ExitIfGLError ( "EnterBezierPatch" );
46:   }
47:   return bp;
48: } /*EnterBezierPatches*/
```

w odwołaniach do tablicy punktów kontrolnych. Zmienna jednolita BezPatch.stride_p w linii 25 otrzymuje wartość $(m+1)(n+1)$, bo tyle punktów kontrolnych ma siatka każdego płata stopnia (n, m). Zmienne BezPatch.nq, BezPatch.stride_q, BezPatch.stride_u i BezPatch.stride_v nie są tu używane i dlatego procedura nie przypisuje im żadnej wartości.

Listing 13.15: Procedura EnterBezierPatchesElem

```c
1: BezierPatchObjf* EnterBezierPatchesElem ( GLint udeg, GLint vdeg, GLint dim,
2:                          int npatches, int ncp,
3:                          const GLfloat *cp, const GLint *ind,
4:                          const GLfloat *colour )
5: {
6:   BezierPatchObjf *bp;
7:   GLint           size, one = 1;
8: 
9:   if ( dim < 2 || dim > 4 || npatches < 1 ||
10:        udeg < 1 || udeg > MAX_PATCH_DEGREE ||
11:        vdeg < 1 || vdeg > MAX_PATCH_DEGREE )
12:     return NULL;
13:   bp = malloc ( sizeof(BezierPatchObjf) );
14:   if ( bp ) {
15:     memset ( bp, 0, sizeof(BezierPatchObjf) );
16:     bp->udeg = udeg;  bp->vdeg = vdeg;  bp->dim = dim;
17:     bp->npatches = npatches;
18:     bp->buf[0] = NewUniformBlockObject ( bezpbsize, bezpbbp );
19:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[0],
20:                       sizeof(GLint), &bp->dim );
21:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[1],
22:                       sizeof(GLint), &bp->udeg );
23:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[2],
24:                       sizeof(GLint), &bp->vdeg );
25:     bp->stride_p = (udeg+1)*(vdeg+1);
26:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[5],
27:                       sizeof(GLint), &bp->stride_p );
28:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[6], sizeof(GLint), &one );
```

```
29:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[8], sizeof(GLint), &one );
30:     glBufferSubData ( GL_UNIFORM_BUFFER, bezpbofs[9],
31:                       4*sizeof(GLfloat), colour );
32:     size = ncp*dim*sizeof(GLfloat);
33:     bp->buf[1] = NewUniformBlockObject ( cpbofs[0]+size, cpbbp );
34:     glBufferSubData ( GL_UNIFORM_BUFFER, cpbofs[0], size, cp );
35:     size = (udeg+1)*(vdeg+1)*npatches*sizeof(GLint);
36:     bp->buf[2] = NewUniformBlockObject ( size, cpibbp );
37:     glBufferSubData ( GL_UNIFORM_BUFFER, cpibofs[0], size, ind );
38:     ExitIfGLError ( "EnterBezierPatchesElem" );
39:   }
40:   return bp;
41: } /*EnterBezierPatchesElem*/
```

Aby narysować płaty, należy wywołać procedurę `DrawBezierPatches` z listingu 13.16. W liniach 4, 5 i 7 przywiązuje ona UBO opisujące płaty do odpowiednich punktów dowiązania. W linii 8 do kontekstu przywiązywany jest VAO reprezentujący dziedzinę płatów (w zasadzie to może być absolutnie dowolny VAO, który ma określony atrybut wierzchołków o numerze miejsca 0; jest on potrzebny, aby spowodować rysowanie, ale wszelkie dane opisujące płaty pochodzą ze zmiennych jednolitych). W linii 9 określamy, że dziedzina jest czworokątem. Procedura `glDrawArraysInstanced` w linii 10 daje impuls do rysowania wszystkich płatów jednocześnie; liczba tych płatów jest podana jako ostatni parametr.

Procedura `DeleteBezierPatches` zwalnia bufory w pamięci GPU i dealokuje strukturę typu `BezierPatchObjf`. W tablicy `bp->buf` są dwa albo trzy identyfikatory buforów; jeśli są tylko dwa, to jest `buf->bp[2] == 0`. Liczba 0 nigdy nie jest identyfikatorem bufora. Procedura `glDeleteBuffers` ignoruje taką liczbę, zamiast zwalniać nieistniejący bufor.

Listing 13.16: Procedury rysowania i dealokacji płatów Béziera

C

```
1: void DrawBezierPatches ( BezierPatchObjf *bp )
2: {
3:   if ( bp ) {
4:     glBindBufferBase ( GL_UNIFORM_BUFFER, bezpbbp, bp->buf[0] );
5:     glBindBufferBase ( GL_UNIFORM_BUFFER, cpbbp, bp->buf[1] );
6:     if ( bp->buf[2] )
7:       glBindBufferBase ( GL_UNIFORM_BUFFER, cpibbp, bp->buf[2] );
8:     glBindVertexArray ( pd_vao );
9:     glPatchParameteri ( GL_PATCH_VERTICES, 4 );
```

```
10:     glDrawArraysInstanced ( GL_PATCHES, 0, 4, bp->npatches );
11:     ExitIfGLError ( "DrawBezierPatches" );
12:   }
13: } /*DrawBezierPatches*/
14:
15: void DeleteBezierPatches ( BezierPatchObjf *bp )
16: {
17:   if ( bp ) {
18:     glDeleteBuffers ( 3, bp->buf );
19:     free ( bp );
20:   }
21: } /*DeleteBezierPatches*/
```

Czajnik z Utah

Nasza aplikacja rysuje słynny czajnik z Utah; stworzył go na podstawie porcelanowego pierwowzoru Martin Newell w r. 1975. Warto wiedzieć, że początkowo czajnik składał się z 28 płatów Béziera i nie miał dna, które zostało dorobione później. James Blinn przeskalował oryginalny model w kierunku osi z, „spłaszczając” go (tj. zmniejszając wysokość) o 1/4 i w tej wersji czajnik jest najbardziej znany z licznych portretów i przedstawień. Dane opisujące model wziąłem z książki [21].

Listing 13.17: Procedura wprowadzająca czajnik z Utah

C

```
1: BezierPatchObjf* ConstructTheTeapot ( const GLfloat *colour )
2: {
3:   static GLfloat teapotcp[306][3] =
4:    {{ 1.40000,  0.00000,  2.40000}, { 1.40000, -0.78400, 2.40000},
5:     { 0.78400, -1.40000,  2.40000}, { 0.00000, -1.40000, 2.40000},
6:     ........ /* tu jest jeszcze 300 punktów */
7:     { 0.79800, -1.42500,  0.00000}, { 1.42500, -0.79800, 0.00000}};
8:   static GLint teapotcn[32][16] =
9:    {{  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15},
10:     {  3, 16, 17, 18,  7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27},
11:     ........ /* tu jest jeszcze 29 wierszy, po 16 liczb w każdym */
12:     {269,269,269,269,299,304,305,278,296,302,303,274, 95, 94, 93, 92}};
13:
14:   return EnterBezierPatchesElem ( 3, 3, 3, 32, 306,
15:                               &teapotcp[0][0], &teapotcn[0][0], colour );
16: } /*ConstructTheTeapot*/
```

Na listingu 13.17 jest podany fragment procedury, która umieszcza w pamięci

GPU opisaną wyżej reprezentację 32 bikubicznych płatów Béziera składających się na model czajnika. Płaty bikubiczne (stopnia $(3, 3)$) mają po 16 punktów kontrolnych, ale wiele z tych punktów pokrywa się z innymi (zobacz np. linie 9, 10 i 12 na listingu). Jest zatem tylko 306 różnych punktów kontrolnych[10], ale tablica indeksów ma $32 \cdot 16 = 512$ elementów.

Druga aplikacja

Podstawą drugiej aplikacji jest szkielet z listingu 3.2, przy czym główna pętla komunikatów jest realizowana przez procedurę `MessageLoop` z listingu 3.3, co umożliwia animację („samoczynne" obracanie modelu). Na listingu 13.18 jest pokazana procedura czytająca i kompilująca szadery, łącząca je w program i uzyskująca dostęp do zmiennych jednolitych w tym programie. Nowym elementem OpenGL-a są wywołania procedury `glGetUniformLocation` w liniach 41 i 42, w celu uzyskania dostępu do zmiennych jednolitych `BezTessLevel` i `BezNormals`.

Listing 13.18: Procedura `LoadMyShaders`

```c
 1: void LoadMyShaders ( void )
 2: {
 3:   static const char *filename[] =
 4:     { "app2.glsl.vert", "app2.glsl.tesc", "app2.glsl.tese",
 5:       "app2.glsl.geom", "app2.glsl.frag" };
 6:   static const GLuint shtype[6] =
 7:     { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
 8:       GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER };
 9:   static const GLchar *UTBNames[] =
10:     { "TransBlock", "TransBlock.mm", "TransBlock.mmti", "TransBlock.vm",
11:       "TransBlock.pm", "TransBlock.mvpm", "TransBlock.eyepos" };
12:   static const GLchar *ULSNames[] =
13:     { "LSBlock", "LSBlock.nls", "LSBlock.mask",
14:       "LSBlock.ls[0].ambient", "LSBlock.ls[0].direct",
15:       "LSBlock.ls[0].position",
16:       "LSBlock.ls[0].attenuation", "LSBlock.ls[1].ambient" };
17:   static const GLchar *UCPNames[] =
18:     { "CPoints", "CPoints.cp" };
19:   static const GLchar *UCPINames[] = { "CPIndices", "CPIndices.cpi" };
20:   static const GLchar *UBezPatchNames[] =
```

[10] Punkty o numerach 204, 205, 215, 222, 270, 271, 272, 273, 282, 283, 284, 291, 292, 293, 300 i 301 są nieużywane — można je usunąć z tablicy `teapotcp`, co pociąga konieczność zmodyfikowania indeksów w tablicy `teapotcn`. Ale ja tu hołubię dane oryginalne.

```
21:    { "BezPatch", "BezPatch.dim", "BezPatch.udeg", "BezPatch.vdeg",
22:      "BezPatch.stride_u", "BezPatch.stride_v",
23:      "BezPatch.stride_p", "BezPatch.stride_q", "BezPatch.nq",
24:      "BezPatch.use_ind", "BezPatch.Colour" };
25:  static const GLchar *UVarNames[] = { "BezNormals", "BezTessLevel"};
26:  GLint i;
27:
28:  for ( i = 0; i < 5; i++ )
29:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
30:  program_id = LinkShaderProgram ( 5, shader_id );
31:  GetAccessToUniformBlock ( program_id, 6, &UTBNames[0],
32:                            &trbi, &trbsize, trbofs, &trbbp );
33:  GetAccessToUniformBlock ( program_id, 7, &ULSNames[0],
34:                            &lsbi, &lsbsize, lsbofs, &lsbbp );
35:  GetAccessToUniformBlock ( program_id, 1, &UCPNames[0],
36:                            &cpbi, &i, cpbofs, &cpbbp );
37:  GetAccessToUniformBlock ( program_id, 1, &UCPINames[0],
38:                            &cpibi, &i, cpibofs, &cpibbp );
39:  GetAccessToUniformBlock ( program_id, 10, &UBezPatchNames[0],
40:                            &bezpbi, &bezpbsize, bezpbofs, &bezpbbp );
41:  ubeznloc = glGetUniformLocation ( program_id, UVarNames[0] );
42:  ubeztloc = glGetUniformLocation ( program_id, UVarNames[1] );
43:  trbuf = NewUniformBlockObject ( trbsize, trbbp );
44:  lsbuf = NewUniformBlockObject ( lsbsize, lsbbp );
45:  ExitIfGLError ( "LoadMyShaders" );
46: } /*LoadMyShaders*/
```

Listing 13.19 przedstawia procedurę, która wywołana na początku działania aplikacji wprowadza do pamięci GPU potrzebne dane, w tym macierze przekształceń, opis źródeł światła i model czajnika. Procedura `InitLights` jest taka, jak na listingu 10.8. Kolor czajnika podany przez procedurę `ConstructMyTeapot` jest biały, bo z takiej porcelany został zrobiony oryginalny czajnik, przechowywany obecnie w Muzeum Historii Komputerów w Mountain View w Kalifornii.

Na listingu 13.20 są przedstawione procedury obliczające i przesyłające do pamięci GPU macierze przekształceń; dwie z nich (`SetupMVPMatrix` i `InitViewMatrix`) są identyczne jak w pierwszej aplikacji. Jedyna zmiana procedury `RotateViewer` dotyczy parametrów, które w aplikacji biblioteki GLFW są zmiennopozycyjne i to podwójnej precyzji (bo takie ma parametry opisana dalej procedura `MotionFunc` rejestrowana przez procedurę `glfwSetCursorPosCallback`). Natomiast procedura `SetupModelMatrix` tworzy macierz opisującą złożenie dwóch przekształceń.

Listing 13.19: Inicjalizacja i sprzątanie

C

```
1: BezierPatchObjf *myteapot;
2:
3: void InitMyObject ( void )
4: {
5:  .... /* początek taki sam jak na listingu 10.8 */
6:   InitViewMatrix ();
7:   ConstructBezierPatchDomain ();
8:   ConstructMyTeapot ();
9:   InitLights ();
10: } /*InitMyObject*/
11:
12: void ConstructMyTeapot ( void )
13: {
14:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
15:
16:   myteapot = ConstructTheTeapot ( MyColour );
17:   SetBezierPatchOptions ( BezNormals, TessLevel );
18: } /*ConstructMyTeapot*/
19:
20: void Cleanup ( void )
21: {
22:   int i;
23:
24:   glUseProgram ( 0 );
25:   for ( i = 0; i < 5; i++ )
26:     glDeleteShader ( shader_id[i] );
27:   glDeleteProgram ( program_id );
28:   glDeleteBuffers ( 1, &trbuf );
29:   glDeleteBuffers ( 1, &lsbuf );
30:   DeleteBezierPatchDomain ();
31:   DeleteBezierPatches ( myteapot );
32:   glfwDestroyWindow ( mywindow );
33:   glfwTerminate ();
34: } /*Cleanup*/
```

Pierwsze jest skalowaniem modelu, którego wielkość jest dostosowana do wielkości bryły widzenia: jest to zmniejszenie w skali 1/3. Współczynnik skali osi z jest większy (o czynnik 4/3), aby przywrócić czajnikowi oryginalne proporcje. Zauważmy, że transpozycja odwrotności macierzy przekształcenia modelu (przypisywana zmiennej jednolitej `TransBlock.mmti`) jest obliczana osobno, jako że macierz przekształcenia modelu *nie jest* ortogonalna.

Listing 13.20: Procedury tworzenia macierzy przekształceń

C

```
1: void SetupMVPMatrix ( void ) { ... /* bez zmian */ }
2: void InitViewMatrix ( void ) { ... /* bez zmian */ }
3: void RotateViewer ( double delta_xi, double delta_eta )
4: { ... /* bez zmian */ }
5:
6: void SetupModelMatrix ( float axis[3], float angle )
7: {
8: #define SCF (1.0/3.0)
9:   GLfloat ms[16], mr[16], mt[16], ma[16];
10:
11:   M4x4Scalef ( ms, SCF, SCF, SCF*4.0/3.0 );
12:   M4x4Translatef ( mt, 0.0, 0.0, -0.6 );
13:   M4x4Multf ( ma, mt, ms );
14:   M4x4RotateVf ( mr, axis[0], axis[1], axis[2], angle );
15:   M4x4Multf ( trans.mm, mr, ma );
16:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
17:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[0],
18:                     16*sizeof(GLfloat), trans.mm );
19:   M4x4Scalef ( ms, 1.0/SCF, 1.0/SCF, 1.0/SCF*3.0/4.0 );
20:   M4x4Multf ( trans.mmti, mr, ms );
21:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[1],
22:                     16*sizeof(GLfloat), trans.mmti );
23:   ExitIfGLError ( "SetupModelMatrix" );
24:   SetupMVPMatrix ();
25: } /*SetupModelMatrix*/
```

Instrukcje wykonywane przez procedurę DrawMyTeapot na listingu 13.21 nie wymagają dodatkowych komentarzy; procedura ta jest wywoływana przez procedurę Redraw po przygotowaniu tła i włączeniu testu widoczności. Przypomnijmy, że procedura Redraw jest wywoływana w pętli przez procedurę MessageLoop za każdym razem, gdy zmienna redraw otrzymała wartość różną od 0, a to ma miejsce wtedy, gdy skutkiem pewnego zdarzenia jest „unieważnienie” obrazu w oknie i konieczność wykonania nowego obrazu. Procedura glfwSwapBuffers zamienia bufory obrazu, tj. powoduje wyświetlenie w oknie zawartości bufora, która była i miała być niewidoczna, póki proces rysowania się nie zakończył.

Para procedur obsługujących komunikaty o zdarzeniach wygenerowanych przez mysz (naciśnięcie/zwolnienie przycisku i przesunięcie myszy), pokazanych na listingu 13.22 działa identycznie jak w pierwszej aplikacji; naciskając lewy przycisk

Listing 13.21: Rysowanie czajnika

```c
void DrawMyTeapot ( void )
{
  glUseProgram ( program_id );
  DrawBezierPatches ( myteapot );
} /*DrawMyTeapot*/

void Redraw ( GLFWwindow *win )
{
  glClearColor ( 1.0, 1.0, 1.0, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glEnable ( GL_DEPTH_TEST );
  DrawMyTeapot ();
  glUseProgram ( 0 );
  glFlush ();
  glfwSwapBuffers ( win );
  ExitIfGLError ( "Redraw" );
  redraw = false;
} /*Redraw*/
```

i przesuwając mysz powodujemy obracanie obserwatora wokół obiektu. Inne są tylko listy parametrów tych procedur oraz nazwy makrodefinicji identyfikujące przycisk i zdarzenie. Parametry procedury `MouseFunc` nie podają informacji o położeniu kursora w oknie. Dlatego informacja ta jest uzyskiwana przez wywołanie procedury `glfwGetCursorPos`.

Listing 13.22: Procedury obsługi komunikatów myszy

```c
void MouseFunc ( GLFWwindow *win, int button, int action, int mods )
{
  switch ( app_state ) {
case STATE_NOTHING:
    if ( button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS ) {
      glfwGetCursorPos ( win, &last_xi, &last_eta );
      app_state = STATE_TURNING;
    }
    break;
case STATE_TURNING:
    if ( button == GLFW_MOUSE_BUTTON_LEFT && action != GLFW_PRESS )
      app_state = STATE_NOTHING;
    break;
default:
```

```
15:     break;
16:   }
17: } /*MouseFunc*/
18:
19: void MotionFunc ( GLFWwindow *win, double x, double y )
20: {
21:   switch ( app_state ) {
22: case STATE_TURNING:
23:     if ( x != last_xi || y != last_eta ) {
24:       RotateViewer ( x-last_xi, y-last_eta );
25:       last_xi = x,  last_eta = y;
26:       redraw = true;
27:     }
28:     break;
29: default:
30:     break;
31:   }
32: } /*MotionFunc*/
```

Komunikat o naciśnięciu klawisza Esc jest odbierany przez procedurę KeyFunc, zarejestrowaną przez glfwSetKeyCallback zaś procedura CharFunc, zarejestrowana przez glfwSetCharCallback, otrzymuje komunikaty o napisanych na klawiaturze znakach. Aplikacja reaguje na naciśnięcie klawisza spacji (które powoduje wprawienie modelu w ruch obrotowy lub zatrzymanie obracania) oraz na klawisze '+', '-', 'N' i 'n', które zmieniają wygląd obiektu na obrazie. Napisanie litery 'N' powoduje przełączenie źródła wektorów normalnych przyjmowanych do obliczania oświetlenia — przez szader rozdrabniania albo szader geometrii. Klawisze '+' i '-' zmieniają stopień rozdrobnienia płatów, co umożliwia obejrzenie jego wpływu na obraz.

Na listingu 13.24 mamy pozostałe procedury obsługi komunikatów aplikacji. Procedura ReshapeFunc ma w porównaniu z tą z pierwszej aplikacji zmienioną listę parametrów i dodatkową instrukcję przypisującą redraw = true;. Przypisanie to[11] wymusi odrysowanie zawartości okna w pętli komunikatów. Ponadto zmienił się numer pola pm w bloku TransBlock. Procedura DisplayFunc jest wywoływana, gdy okno zostaje np. odsłonięte. Zamiast rysować, procedura też tylko przypisuje niezerową wartość zmiennej redraw. Procedury ToggleAnimation i IdleFunc realizują animację, tj. obracanie obiektu i są przystosowane do współpracy z procedurą MessageLoop z listingu 3.3.

[11] które odpowiada wywołaniu glutPostWindowRedisplay w aplikacji FreeGLUTa

Listing 13.23: Procedury obsługi komunikatów klawiatury

C

```c
void KeyFunc ( GLFWwindow *win, int key, int scancode,
               int action, int mode )
{
  if ( action == GLFW_PRESS || action == GLFW_REPEAT ) {
    switch ( key ) {
  case GLFW_KEY_ESCAPE:
      glfwSetWindowShouldClose ( mywindow, 1 );
      break;
  default:
      break;
    }
  }
} /*KeyFunc*/

void CharFunc ( GLFWwindow *win, unsigned int charcode )
{
  switch ( charcode ) {
case '+':
    if ( TessLevel < MAX_TESS_LEVEL ) {
      SetBezierPatchOptions ( BezNormals, ++TessLevel );
      redraw = true;
    }
    break;
case '-':
    if ( TessLevel > MIN_TESS_LEVEL ) {
      SetBezierPatchOptions ( BezNormals, --TessLevel );
      redraw = tru;
    }
    break;
case 'N':  case 'n':
    BezNormals = BezNormals == 0;
    SetBezierPatchOptions ( BezNormals, TessLevel );
    redraw = true;
    break;
case ' ':
    ToggleAnimation ();
    break;
default:    /* ignorujemy wszystkie inne klawisze */
    break;
  }
} /*CharFunc*/
```

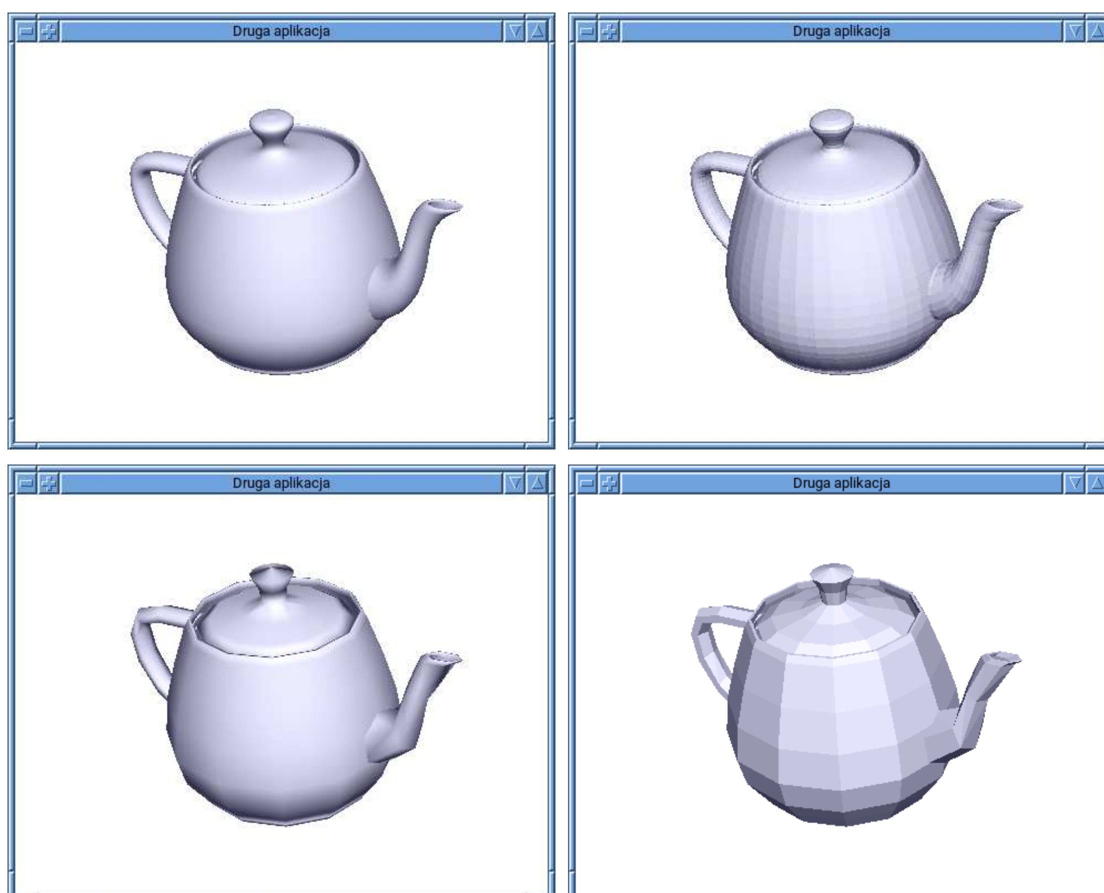
Listing 13.24: Pozostałe procedury obsługi komunikatów

C

```
1: void ReshapeFunc ( GLFWwindow *win, int width, int height )
2: {
3:  .... /* tu wszystko jak na listingu 8.2 */
4:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
5:   glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2],
6:                     16*sizeof(GLfloat), trans.pm );
7:   win_width = width,  win_height = height;
8:   left = -(right = lr);  bottom = -(top = 0.5533);  near = 5.0;  far = 15.0;
9:   ExitIfGLError ( "ReshapeFunc" );
10:   SetupMVPMatrix ();
11:   redraw = true;
12: } /*ReshapeFunc*/
13:
14: void DisplayFunc ( GLFWwindow *win )
15: {
16:   redraw = true;
17: } /*DisplayFunc*/
18:
19: void IdleFunc ( void )
20: {
21:   model_rot_angle = model_rot_angle0 + 0.78539816 * TimerToc ();
22:   SetupModelMatrix ( model_rot_axis, model_rot_angle );
23:   redraw = true;
24: } /*IdleFunc*/
25:
26: void ToggleAnimation ( void )
27: {
28:   if ( (animate = !animate) ) {
29:     TimerTic ();
30:     SetIdleFunc ( IdleFunc );
31:   }
32:   else {
33:     model_rot_angle0 = model_rot_angle;
34:     SetIdleFunc ( NULL );
35:   }
36: } /*ToggleAnimation*/
```

Rysunek 13.4 przedstawia okno aplikacji z czterema obrazami czajnika; w górnym wierszu parametry podziału płatów mają wartość 10, a w dolnym najmniejszą dopuszczalną, czyli 3. Obrazy z lewej strony zostały wykonane przy użyciu

wektorów normalnych płatów obliczonych przez szader rozdrabniania[12]. Podczas wykonywania obrazów z prawej strony wektory normalne obliczone przez szader rozdrabniania zostały zignorowane; szader geometrii obliczył wektory normalne trójkątów otrzymanych w wyniku rozdrabniania płatów, czego skutkiem jest obraz pokazujący płaskie fragmenty narysowanej powierzchni przybliżającej oryginalne, gładkie płaty zakrzywione.



Rysunek 13.4: Okno drugiej aplikacji — różne obrazy czajnika

[12] Z wyjątkiem sytuacji, gdy obliczony został wektor zerowy i zadanie obliczania wektora normalnego przejął szader geometrii.

Uzupełnienia

Istnieje ograniczenie wielkości bloku zmiennych jednolitych; specyfikacja OpenGL gwarantuje możliwość używania bloków o wielkości 16KB, przy czym implementacje mogą mieć większy limit, ale też niezbyt duży, np. 64KB. Limit w konkretnym systemie możemy poznać, wywołując procedurę

```
glGetIntegerv ( GL_MAX_UNIFORM_BLOCK_SIZE, &n );
```

której drugi parametr jest adresem zmiennej typu `GLint`. Dla siatek kontrolnych o wielu wierszach i kolumnach limit wielkości bloku może być za mały; wektory o trzech lub czterech współrzędnych pojedynczej precyzji zajmują 12 albo 16 bajtów, a więc zgodnie ze specyfikacją mamy gwarancję zmieszczenia w bloku zmiennych jednolitych `CPoints` co najwyżej 1365 punktów kontrolnych płata wielomianowego lub 1024 punktów kontrolnych płata wymiernego. Zamiast bloku zmiennych jednolitych możemy użyć bloku magazynowego; rozmiar takich bloków może być znacznie większy[13].

W programie w GLSL blok magazynowy deklarujemy podobnie jak blok zmiennych jednolitych, zastępując słowo kluczowe `uniform` słowem `buffer`. Deklaracja może wyglądać tak[14]:

```
layout (binding=0) buffer CPoints {
  float cp[]; /* ostatnia tablica w bloku magazynowym może nie mieć
                 podanej długości */
} cp;
```

Bufor dla bloku magazynowego (SSBO — *shader storage buffer object*) tworzymy tak samo, jak bufory dla bloków zmiennych jednolitych i przesyłamy dane do niego w taki sam sposób.

Aby dane w buforze były dostępne dla szadera, należy ten bufor przywiązać do celu `GL_SHADER_STORAGE_BUFFER`, który podobnie jak cel `GL_UNIFORM_BUFFER` jest indeksowany, tj. ma tablicę punktów dowiązania. Po skompilowaniu i połączeniu programu szaderów potrzebujemy znaleźć przesunięcia pól w tym bloku względem

[13] Nawet $2^{31} - 1$ bajtów, jeśli GPU ma dostatecznie dużo pamięci. Ceną za to może być nieco wolniejszy dostęp do danych.

[14] Podobny kwalifikator `layout` może też poprzedzać deklarację bloku zmiennych jednolitych; określa on numer punktu dowiązania (w celu `GL_UNIFORM_BUFFER`), który dotąd określaliśmy przy użyciu procedury `glUniformBlockBinding`. Jeśli użyjemy obu sposobów określania numeru punktu dowiązania, to ostatnie słowo ma procedura `glUniformBlockBinding`.

jego początku, aby móc im przypisywać wartości przy użyciu procedury glBufferSubData. Można użyć do tego procedury przedstawionej na listingu 13.25. Procedurę tę napisałem tak, aby maksymalnie upodobnić jej używanie przez aplikację do procedury GetAccessToUniformBlock z listingu 10.5.

Listing 13.25: Procedura GetAccessToStorageBlock

C

```
1: void GetAccessToStorageBlock ( GLuint prog, int n, const GLchar **names,
2:                                GLuint *ind, GLint *size, GLint *ofs,
3:                                GLuint *bpoint )
4: {
5:   int    i;
6:   GLuint prop;
7:
8:   ind[0] = glGetProgramResourceIndex ( prog, GL_SHADER_STORAGE_BLOCK,
9:                                        names[0] );
10:   prop = GL_BUFFER_DATA_SIZE;
11:   glGetProgramResourceiv ( prog, GL_SHADER_STORAGE_BLOCK, ind[0],
12:                            1, &prop, 1, NULL, size );
13:   prop = GL_BUFFER_BINDING;
14:   glGetProgramResourceiv ( prog, GL_SHADER_STORAGE_BLOCK, ind[0],
15:                            1, &prop, 1, NULL, (GLint*)bpoint );
16:   if ( n > 0 ) {
17:     prop = GL_OFFSET;
18:     for ( i = 1; i <= n; i++ ) {
19:       ind[i] = glGetProgramResourceIndex ( prog, GL_BUFFER_VARIABLE,
20:                                            names[i] );
21:       glGetProgramResourceiv ( prog, GL_BUFFER_VARIABLE, ind[i],
22:                                1, &prop, 1, NULL, &ofs[i-1] );
23:     }
24:   }
25:   ExitIfGLError ( "GetAccessToStorageBlock" );
26: } /*GetAccessToStorageBlock*/
```

Pierwszym parametrem jest identyfikator programu, drugim liczba n pól w bloku magazynowym, a trzeci parametr to tablica wskaźników do napisów, z których pierwszy jest nazwą globalną bloku, a kolejne n to nazwy pól w tym bloku (poprzedzone nazwą bloku i kropką, np. "CPoints.cp"). Kolejne parametry wskazują zmienne, którym procedura ma przypisać wyniki: kolejno tablicę z indeksami bloku i jego kolejnych pól, wielkość bloku w bajtach[15], tablicę

[15]Wielkość ta jest obliczona na podstawie deklaracji bloku; w szczególności, jeśli na jego końcu jest tablica, to brana jest jej długość podana w deklaracji, a nie faktyczna długość, jaką ta tablica będzie miała w utworzonym przez aplikację buforze.

przesunięć kolejnych pól w bloku i numer punktu dowiązania ustalony dla tego bloku magazynowego w celu GL_SHADER_STORAGE_BLOCK[16]. Poszczególne informacje są odczytywane z programu przez uniwersalne procedury glGetProgramResourceIndex i glGetProgramResourceiv[17].

Aby przywiązać bufor z danymi do bloku magazynowego w programie, przed wywołaniem tego programu (np. przed rysowaniem płatów) trzeba wykonać instrukcję

```
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, buf );
```

przy czym drugi parametr jest numerem punktu dowiązania w celu GL_SHADER_STORAGE_BUFFER — ma to być liczba podana w kwalifikatorze layout(binding=0) w deklaracji bloku magazynowego (to ta liczba zostaje przez procedurę GetAccessToStorageBlock przypisana zmiennej *bpoint). Trzeci parametr jest podanym przez procedurę glGenBuffers identyfikatorem bufora, do którego zostały wpisane odpowiednie dane.

[16]Mimo starań nie znalazłem odpowiednika procedury glUniformBlockBinding, która zmieniałaby numer punktu dowiązania bloku magazynowego zadeklarowany w tekście szadera, więc pewnie takiej procedury nie ma. Dlatego zamiast nadawać kolejny numer, procedura GetAccessToStorageBlock odczytuje informację na jego temat z programu szaderów. Jeśli kilka programów szaderów ma mieć dostęp do tego samego bloku magazynowego, to trzeba dopilnować, aby w każdym z tych programów blok magazynowy miał ten sam numer.

[17]Procedur tych można używać także do odczytywania informacji na temat bloków zmiennych jednolitych, natomiast nie ma odpowiednika „wysokopoziomowej” procedury glGetActiveUniformsiv dla bloków magazynowych.

14. Aplikacja druga A

Odrobinę rozszerzymy możliwości drugiej aplikacji. Na życzenie ma ona wyświetlać siatki kontrolne płatów Béziera oraz „kreskowy” obraz czajnika.

Wyświetlanie siatek kontrolnych — szadery

Na obraz siatki kontrolnej płata Béziera stopnia (n, m) składa się $(m+1)n + (n+1)m = 2mn + m + n$ odcinków. Aby narysować siatki p płatów, możemy wszystkie odcinki wyświetlić, wydając polecenie narysowania *jednego* odcinka w *odpowiedniej liczbie* instancji. Problem sprowadza się do wyznaczenia końców odcinka na podstawie numeru instancji. Szader wierzchołków, który to robi (listing 14.1), jest dostosowany do struktur danych (zmiennych jednolitych) opisanych w poprzednim rozdziale.

W liniach 16 i 17 obliczane są liczby punktów kontrolnych i liczby odcinków siatki każdego płata. W liniach 18 i 19 na podstawie numeru instancji obliczane są odpowiednio numer płata i numer odcinka w siatce, który należy narysować.

Warunek sprawdzany w linii 20 odróżnia odcinki należące do wierszy i kolumn siatki kontrolnej; przyjęte jest, że odcinki należące do wierszy mają w obrębie siatki numery od 0 do $(m+1)n - 1$.

W liniach 20–44 następuje obliczenie indeksu do tablicy punktów kontrolnych, pod którym znajduje się pierwsza współrzędna potrzebnego punktu kontrolnego, który jest końcem odcinka. Odcinek ma dwa końce; wybór właściwego końca jest dokonywany na podstawie zmiennej wbudowanej `gl_VertexID`, której wartość jest numerem wierzchołka rysowanego prymitywu: dla odcinka jest to liczba 0 albo 1.

Przyjęta reprezentacja zbioru płatów Béziera dopuszcza dwie możliwości: z użyciem albo bez użycia tablicy indeksów. W pierwszym przypadku (linie 22–23) odpowiedni indeks jest pobierany z tablicy `cpi.cpi`, z miejsca o numerze

$$(n+1)(m+1)n_p + n_l \quad \text{albo} \quad (n+1)(m+1)n_p + n_l + m + 1,$$

zależnie od tego, który koniec odcinka jest potrzebny; n_p oznacza numer płata a n_l oznacza numer odcinka w siatce (jeden koniec każdego odcinka w wierszu jest w jednym z pierwszych $(m+n)n$ punktów kontrolnych, a drugi koniec jest w następnej kolumnie siatki). Indeks odczytany z tablicy `cpi.cpi` jest mnożony przez liczbę współrzędnych punktu (podaną w zmiennej `bezp.dim`).

Listing 14.1: Szader wierzchołków programu wyświetlania siatek kontrolnych

GLSL

```
1: #version 420 core
2:
3: #define MAX_DEG 10
4:
5: layout(location=0) in vec4 in_Position;
6:
7:  .... /* tu deklaracje zmiennych jednolitych takie, */
8:  .... /* jak w liniach 17-36 na listingu 13.4 */
9:
10: void main ( void )
11: {
12:   int  n, m, nn, nlines, np, nl, a, b, i;
13:   vec4 p;
14:
15:   n = bezp.udeg;  m = bezp.vdeg;
16:   nn = (n+1)*(m+1);
17:   nlines = 2*m*n+m+n;           /* liczba odcinków siatki */
18:   np = gl_InstanceID / nlines;  /* numer siatki */
19:   nl = gl_InstanceID % nlines;  /* numer odcinka w siatce */
20:   if ( nl < (m+1)*n )   /* odcinek wiersza */
21:     if ( bezp.use_ind )
22:       i = cpi.cpi[nn*np +
23:                   ((gl_VertexID == 0) ? nl : nl+m+1)] * bezp.dim;
24:     else {
25:       a = nl / n;  /* numer wiersza */
26:       b = nl % n;  /* numer odcinka w wierszu */
27:       if ( gl_VertexID != 0 ) b ++;
28:       i = (np / bezp.nq)*bezp.stride_p + (np % bezp.nq)*bezp.stride_q +
29:           a*bezp.stride_v + b*bezp.stride_u;
30:     }
31:   }
32:   else {                  /* odcinek kolumny */
33:     nl -= (m+1)*n;
34:     a = nl / m;  /* numer kolumny */
35:     b = nl % m;  /* numer odcinka w kolumnie */
36:     if ( bezp.use_ind ) {
37:       i = cpi.cpi[nn*np + a*(m+1) +
38:                   ((gl_VertexID == 0) ? b : b+1)] * bezp.dim;
39:     }
40:     else {
41:       a = nl / m;  /* numer kolumny */
42:       b = nl % m;  /* numer odcinka w kolumnie */
```

```
43:      i = (np / bezp.nq)*bezp.stride_p + (np % bezp.nq)*bezp.stride_q +
44:          a*bezp.stride_u + b*bezp.stride_v;
45:    }
46:  }
47:  switch ( bezp.dim ) {
48: case 2: p = vec4 ( cp.cp[i], cp.cp[i+1], 0.0, 1.0 );  break;
49: case 3: p = vec4 ( cp.cp[i], cp.cp[i+1], cp.cp[i+2], 1.0 );  break;
50: case 4: p = vec4 ( cp.cp[i], cp.cp[i+1], cp.cp[i+2], cp.cp[i+3] );  break;
51: default: p = vec4 ( 0.0 );  break;
52:  }
53:  gl_Position = trb.mvpm * p;
54: /*main*/
```

Jeśli tablica indeksów nie jest używana, to odpowiedni indeks do tablicy punktów kontrolnych jest obliczany na podstawie kroków, z jakimi punkty kontrolne są rozmieszczone w tablicy `cp.cp` — kroki te są podane w zmiennych jednolitych `bezp.stride_p` (krok do następnej „kolumny” płatów w siatce), `bezp.stride_q` (krok do następnego „wiersza” płatów w siatce), `bezp.nq` (liczba płatów w kolumnie), `bezp.stride_u` (krok do następnej kolumny siatki) i `bezp.stride_v` (krok do następnego wiersza siatki).

Podobnie wygląda obliczenie indeksu wierzchołka, który jest końcem odcinka należącego do kolumny, w liniach 33–45. W liniach 47–52 jest zrealizowane pobieranie wierzchołka, tj. odczytywanie jego współrzędnych z tablicy `cp.cp`. Dopuszczalne jest podanie dwóch, trzech lub czterech współrzędnych — w pierwszych dwóch przypadkach współrzędne nieobecne otrzymują wartości domyślne. W linii 53 następuje obliczenie współrzędnych wierzchołka w układzie kostki standardowej. Można oczywiście rozwiązać rysowanie siatek inaczej — umieścić punkty kontrolne w VAO i użyć odpowiedniej tablicy indeksów. Ale użyty tu sposób nie wymaga żadnych zmian struktur danych reprezentujących płaty Béziera, co ma swoje zalety.

Listing 14.2: Szader fragmentów programu wyświetlania siatek kontrolnych

GLSL

```
1: #version 420 core
2:
3: out vec4 out_Colour;
4:
5: void main ( void )
6: {
7:   out_Colour = vec4 ( 0.0, 0.7, 0.0, 1.0 );
8: } /*main*/
```

Szader fragmentów na listingu 14.2 jest bardzo prościutki: tylko przypisuje zmiennej wyjściowej wartość reprezentującą kolor zielony. Oczywiście, można go zmienić na coś bardziej wyrafinowanego, jeśli ktoś poczuje, że tak trzeba.

Wyświetlanie siatek kontrolnych — procedury w C

Rysowanie siatki kontrolnej realizuje procedura pokazana na listingu 14.3. Przywiązuje ona do odpowiednich punktów dowiązania UBO ze zmiennymi jednolitymi reprezentującymi zbiór płatów (tak samo jak procedura `DrawBezierPatches` na listingu 13.16), a następnie (w linii 11) oblicza całkowitą liczbę odcinków siatki kontrolnej jednego płata i przekazuje iloczyn tej liczby i liczby płatów (czyli siatek) jako ostatni parametr w wywołaniu procedury `glDrawArraysInstanced`. Procedura ta ma narysować *jeden* odcinek (o dwóch końcach — liczbę końców odcinka określa trzeci parametr) w wielu instancjach. Wierzchołki, formalnie będące końcami tego odcinka, są ignorowane (szader wierzchołków ma własne zdanie na ten temat), dlatego mogą być pobierane z dowolnego VAO, czyli tu z VAO reprezentującego dziedzinę płatów.

Listing 14.3: Procedura `DrawBezierNets`

C

```c
1: void DrawBezierNets ( BezierPatchObjf *bp )
2: {
3:   int nlines;
4:
5:   if ( bp ) {
6:     glBindBufferBase ( GL_UNIFORM_BUFFER, bezpbbp, bp->buf[0] );
7:     glBindBufferBase ( GL_UNIFORM_BUFFER, cpbbp, bp->buf[1] );
8:     if ( bp->buf[2] )
9:       glBindBufferBase ( GL_UNIFORM_BUFFER, cpibbp, bp->buf[2] );
10:    glBindVertexArray ( pd_vao );
11:    nlines = 2*bp->udeg*bp->vdeg + bp->udeg + bp->vdeg;
12:    glDrawArraysInstanced ( GL_LINES, 0, 2, bp->npatches*nlines );
13:    ExitIfGLError ( "DrawBezierNets" );
14:  }
15: } /*DrawBezierNets*/
```

Nowe i zmienione procedury aplikacji

Zmieniona aplikacja używa dwóch programów szaderów; pierwszy z nich jest identyczny, jak w poprzednim rozdziale, a drugi składa się z dwóch szaderów opisanych w tym. Zatem, zmienna skalarna `program_id` została zmieniona na

tablicę o długości 2; wszystkie dotychczasowe odwołania do tej zmiennej muszą być zmienione na odwołania do program_id[0]. Długość tablicy shader_id trzeba zwiększyć do 7. Do procedury LoadMyShaders trzeba dodać instrukcje kompilacji dodatkowych szaderów i łączenia dodatkowego programu, a ponadto trzeba zadbać o przyłączenie wszystkich czterech bloków zmiennych jednolitych drugiego programu (o identyfikatorze zapamiętanym w program_id[1]) do odpowiednich punktów dowiązania (przez wywołanie procedury AttachUniformBlockToBP). Celowo nie zamieściłem listingu ze zmienioną procedurą LoadMyShaders, aby Czytelnik sam przećwiczył dokonanie tych modyfikacji.

Pozostałe procedury, które zostały dodane lub zmienione, są przedstawione na listingu 14.4.

Procedura DrawMyTeapot, zależnie od wartości zmiennej skeleton, wybiera sposób rysowania, przez wywołanie procedury glPolygonMode. Pierwszy jej parametr wybiera trójkąty, które mają być rysowane sposobem określonym przez drugi parametr: odwrócone przodem do obserwatora (GL_FRONT), tyłem (GL_BACK), albo wszystkie (GL_FRONT_AND_BACK), tak jak w tym przykładzie. Drugi parametr wybiera rysowanie pełnych trójkątów (GL_FILL), boków trójkątów (GL_LINE) lub tylko wierzchołków trójkątów (GL_POINT). Szerokość linii, którymi rysowany jest obraz czajnika, jest równa 2.[1]

Procedura DrawMyTeapotCNet wybiera właściwy program szaderów, ustawia szerokość linii (1 piksel) i wywołuje procedurę DrawBezierNets.

Naciskanie klawisza z literą C włącza i wyłącza rysowanie siatki kontrolnej. Naciskanie klawiszy z literami L i S powoduje wybieranie różnych rodzajów obrazu — dotychczas wykonywanego obrazu powierzchni oświetlonej albo rysunku kreskowego, przedstawiającego krawędzie trójkątów wygenerowanych przez szader rozdrabniania.

Jeszcze jedna potrzebna zmiana, nie pokazana na listingu, to dodanie do procedury sprzątającej (Cleanup) instrukcji likwidujących drugi program szaderów i szadery wchodzące w jego skład.

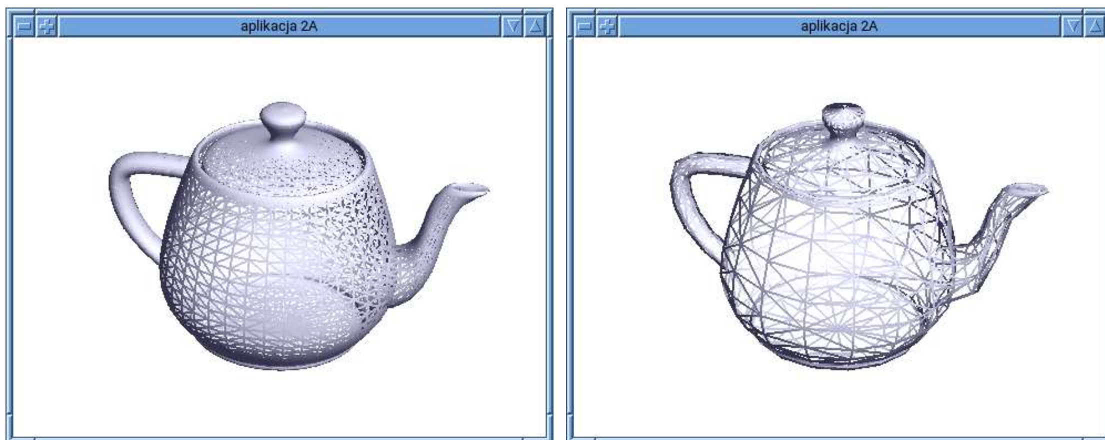
[1] Możemy też ustalać wielkość punktów za pomocą procedury glPointSize. Szadery wierzchołków, rozdrabniania i geometrii mogą indywidualnie określać wielkości punktów, przypisując odpowiednią wartość zmiennej gl_PointSize.

Listing 14.4: Zmiany procedur drugiej aplikacji

C

```
1: char cnet = 0, skeleton = 0;
2:
3: void DrawMyTeapot ( void )
4: {
5:   if ( skeleton ) {
6:     glLineWidth ( 2.0 );
7:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
8:   }
9:   else
10:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
11:   glUseProgram ( program_id );
12:   DrawBezierPatches ( myteapot );
13: } /*DrawMyTeapot*/
14:
15: void DrawMyTeapotCNet ( void )
16: {
17:   glUseProgram ( program_id[1] );
18:   glLineWidth ( 1.0 );
19:   DrawBezierNets ( myteapot );
20: } /*DrawMyTeapotCNet*/
21:
22: void Redraw ( GLFWwindow *win )
23: {
24:   .... /* początek procedury bez zmian */
25:   DrawMyTeapot ();
26:   if ( cnet )
27:     DrawMyTeapotCNet ();
28:   glUseProgram ( 0 );
29:   .... /* koniec też bez zmian */
30: } /*Redraw*/
31:
32: void CharFunc ( GLFWwindow *win, unsigned int charcode )
33: {
34:   switch ( charcode ) {
35: .... /* obsługa dotychczas używanych klawiszy bez zmian */
36: case 'C':  case 'c':
37:     cnet = !cnet;
38:     redraw = true;
39:     break;
40: case 'L':  case 'l':
41:     skeleton = 1;
42:     redraw = true;
```

```
43:       break;
44:   case 'S':  case 's':
45:       skeleton = 0;
46:       redraw = true;
47:       break;
48:   default:    /* ignorujemy wszystkie inne klawisze */
49:       break;
50:     }
51:   } /*CharFunc*/
```



Rysunek 14.1: „Kreskowe" obrazy czajnika dla różnych poziomów rozdrobnienia dziedziny płatów

Rysunek 14.1 przedstawia czajnik narysowany „po nowemu". Zwróćmy uwagę, że kolory pikseli składających się na obrazy odcinków są obliczone na podstawie realizowanego przez szadery modelu oświetlenia powierzchni. Wprawdzie wektor normalny odcinka w przestrzeni jest nieokreślony, ale w obliczeniu kolorów używany jest wektor normalny odpowiedniego płata Béziera przybliżanego przez trójkąt, którego bokiem jest rysowany odcinek.

Ćwiczenia

1. Dodaj możliwość interakcyjnego zmieniania grubości linii.
2. Dodaj możliwość rysowania tylko wierzchołków trójkątów, a także interakcyjnego wybierania (w pewnych granicach) średnicy kropki będącej obrazem punktu.
3. Zmień parametr `GL_FRONT_AND_BACK` w wywołaniu procedury `glPolygonMode` na `GL_FRONT` i zobacz, co z tego wyjdzie.

Uzupełnienia

Obie aplikacje nie mają jak dotąd wbudowanych ograniczeń na obroty obserwatora wokół sceny. W pewnych zastosowaniach (np. w grach i w tworzeniu filmów animowanych) może być pożądane „pionizowanie" obserwatora. Ma ono na celu sprawienie, aby obraz osi y układu obserwatora miał ten sam kierunek i zwrot, co obraz ustalonego wektora, który oznaczymy symbolem $\boldsymbol{z}$ (bo on wyznacza zenit — na ogół będzie to wersor osi z układu współrzędnych świata). Aby to zrobić, należy odpowiednio obrócić obserwatora wokół osi z *jego* układu, przy czym ma to sens pod warunkiem, że ta oś nie ma kierunku zenitu[2].

Kiedy funkcja pionizowania będzie w aplikacji włączona, procedura `RotateViewer` po złożeniu obrotu określonego przez ostatnie przemieszczenie kursora z obrotem dotychczasowym wykona dodatkowy obrót obserwatora o kąt θ znaleziony niżej. Wersor $\boldsymbol{z}'$ osi z układu obserwatora jest wektorem jednostkowym osi tego obrotu. Obraz $R_{\boldsymbol{z}',\theta}\boldsymbol{y}'$ wektora $\boldsymbol{y}'$, będącego wersorem osi y układu obserwatora, oraz wektory $\boldsymbol{z}'$ i $\boldsymbol{z}$ mają leżeć w jednej płaszczyźnie (czyli wszystkie trzy wektory mają być liniowo zależne). Warunek ten jest zapisany w równaniu

$$\det[R_{\boldsymbol{z}',\theta}\boldsymbol{y}', \boldsymbol{z}', \boldsymbol{z}] = \langle (R_{\boldsymbol{z}',\theta}\boldsymbol{y}') \wedge \boldsymbol{z}', \boldsymbol{z}\rangle = 0.$$

Na podstawie wzoru (5.8) możemy napisać[3]

$$\begin{aligned}(R_{\boldsymbol{z}',\theta}\boldsymbol{y}') \wedge \boldsymbol{z}' &= \big(\boldsymbol{z}'\boldsymbol{z}'^T\boldsymbol{y}' + c(\boldsymbol{y}' - \boldsymbol{z}'\boldsymbol{z}'^T\boldsymbol{y}') + s\boldsymbol{z}' \wedge \boldsymbol{y}'\big) \wedge \boldsymbol{z}' \\ &= c\boldsymbol{y}' \wedge \boldsymbol{z}' - s(\boldsymbol{y}' \wedge \boldsymbol{z}') \wedge \boldsymbol{z}' = c\boldsymbol{x}' + s\boldsymbol{y}'.\end{aligned}$$

Nowe symbole oznaczają $\cos\theta = c$, $\sin\theta = s$, wektor $\boldsymbol{y}' \wedge \boldsymbol{z}' = \boldsymbol{x}'$ jest wersorem osi x układu obserwatora, w związku z czym $\boldsymbol{x}' \wedge \boldsymbol{z}' = -\boldsymbol{y}'$. Ma zatem być $\langle c\boldsymbol{x}' + s\boldsymbol{y}', \boldsymbol{z}\rangle = 0$, skąd łatwo jest wywieść, że

$$\operatorname{tg}\theta = \frac{s}{c} = \frac{-\langle \boldsymbol{x}', \boldsymbol{z}\rangle}{\langle \boldsymbol{y}', \boldsymbol{z}\rangle}.$$

Do obliczenia kąta θ aplikacja może użyć funkcji `atan2` ze standardowej biblioteki funkcji matematycznych `libm`. Zwracam uwagę, że jeśli znaki obu jej argumentów (licznika i mianownika) zostaną zmienione, to tangens θ pozostanie ten sam, ale wartość funkcji `atan2` zmieni się o π lub $-\pi$; zły wybór znaku prowadzi do odwrócenia obserwatora do góry nogami. Znaki przyjęte we wzorze podanym wyżej dają poprawny wynik.

---

[2] Czyli nie wtedy, gdy obserwator patrzy pionowo w dół lub do góry.

[3] Pamiętamy, że iloczyn wektorowy nie jest działaniem łącznym, a zmiana kolejności wektorów zmienia zwrot ich iloczynu wektorowego na przeciwny. Ponadto układ współrzędnych obserwatora powstaje przez obrót i przesunięcie układu świata, tak więc jest to układ izometryczny (rozdz. 5). Jego wersory osi spełniają równania $\boldsymbol{x}' \wedge \boldsymbol{y}' = \boldsymbol{z}'$, $\boldsymbol{y}' \wedge \boldsymbol{z}' = \boldsymbol{x}'$, $\boldsymbol{z}' \wedge \boldsymbol{x}' = \boldsymbol{y}'$.

Jest jeszcze jeden problem: jeśli kąt między osią z układu obserwatora i prostą o kierunku wektora $\boldsymbol{z}$ jest mały, to kąt θ obrotu potrzebnego do spionizowania obserwatora może być bardzo duży. Kiedy tak jest, obroty obserwatora wokół sceny „szaleją”, co nie wydaje się pożądanym efektem. Można wypróbować różne sposoby przeciwdziałania temu zjawisku. Wypróbowane przeze mnie rozwiązanie polega na uzależnieniu kąta ϕ obrotu obserwatora od jego położenia. Jest on równy 3°, jeśli oś z układu obserwatora jest prostopadła do kierunku zenitu i maleje, gdy kąt między tą osią i zenitem maleje.

Listing 14.5 przedstawia zmienioną procedurę `RotateViewer`, wywoływaną, gdy użytkownik po naciśnięciu przycisku przesuwa mysz. Jej instrukcje konstruujące i przesyłające macierz przejścia do układu obserwatora i położenie obserwatora do bufora w pamięci GPU zostały przeniesione do nowej procedury `LoadViewMatrix`, którą `RotateViewer` wywołuje na końcu. Pozostałe zmiany w tej procedurze to obliczenie kąta obrotu spowodowanego przesunięciem myszy i wywołanie procedury `Verticalise`, która dokonuje obrotu pionizującego. Te instrukcje są wykonywane, jeśli pionizowanie jest włączone — przypisanie wartości `true` lub `false` (czyli 1 lub 0) jest sterowane odpowiednim klawiszem.

Listing 14.5: Pionizowanie obserwatora

C

```c
1: float zenith[3] = {0.0,0.0,1.0};
2: char  vertical = false;
3:
4: void LoadViewMatrix ( void )
5: {
6:   GLfloat tm[16], rm[16];
7:
8:   M4x4Translatef ( tm, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
9:   M4x4RotateVf ( rm, viewer_rvec[0], viewer_rvec[1], viewer_rvec[2],
10:                 -viewer_rangle );
11:  M4x4Multf ( trans.vm, tm, rm );
12:  glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
13:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat),
14:                    trans.vm );
15:  M4x4Transposef ( tm, rm );
16:  M4x4MultMVf ( trans.eyepos, tm, viewer_pos0 );
17:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[5], 4*sizeof(GLfloat),
18:                    trans.eyepos );
19:  ExitIfGLError ( "LoadViewMatrix" );
20:  SetupMVPMatrix ();
21: } /*LoadViewMatrix*/
```

```
22:
23: void Verticalise ( float vk[3], float *angk )
24: {
25:   float R[16], s, c, vr[3], theta, ang;
26:
27:   M4x4RotateVf ( R, vk[0], vk[1], vk[2], *angk );
28:   s = -V3DotProductf ( &R[0], zenith );
29:   c = V3DotProductf ( &R[4], zenith );
30:   theta = atan2 ( s, c );
31:   V3CompRotationsf ( vr, &ang, &R[8], theta, vk, *angk );
32:   memcpy ( vk, vr, 3*sizeof(float) );
33:   *angk = ang;
34: } /*Verticalise*/
35:
36: void RotateViewer ( double delta_xi, double delta_eta )
37: {
38:   float vi[3], lgt, angi, vk[3], angk;
39:
40:   if ( delta_xi == 0 && delta_eta == 0 )
41:     return;
42:   vi[0] = (float)delta_eta*(right-left)/(float)win_height;
43:   vi[1] = (float)delta_xi*(top-bottom)/(float)win_width;
44:   vi[2] = 0.0;
45:   lgt = sqrt ( V3DotProductf ( vi, vi ) );
46:   vi[0] /= lgt;  vi[1] /= lgt;
47:   angi = -0.052359878;
48:   if ( vertical )
49:     angi *= 0.01 + 0.99*fabs ( trans.vm[1]*zenith[0] +
50:             trans.vm[5]*zenith[1] + trans.vm[9]*zenith[2] );
51:   V3CompRotationsf ( vk, &angk, viewer_rvec, viewer_rangle, vi, angi );
52:   if ( vertical )
53:     Verticalise ( vk, &angk );
54:   memcpy ( viewer_rvec, vk, 3*sizeof(float) );
55:   viewer_rangle = angk;
56:   LoadViewMatrix ();
57: } /*RotateViewer*/
```

Procedura `Verticalise` w linii 27 tworzy macierz 4×4 reprezentującą obrót obserwatora w układzie świata przed spionizowaniem[4]. Dalej potrzebny jest tylko górny lewy blok 3×3 tej macierzy, oznaczymy go symbolem R_{z_i,ϕ_i}. Jego odwrotność, $R_{z_i,-\phi_i}$, opisuje część liniową przejścia od układu świata do układu obserwatora; blok ten jest macierzą ortogonalną (bo obrót jest izometrią), zatem

[4]Zwracam uwagę na znak kąta będącego ostatnim parametrem procedury `M4x4RotateVf`.

jego odwrotność jest jego transpozycją. Kolumny macierzy R_{z_i,ϕ_i} opisują (w układzie współrzędnych świata) wersory x', y', z' osi układu obserwatora. Zatem, parametry `&R[0]`, `&R[4]` i `&R[8]` w liniach 27, 28 i 30 przekazują wywoływanym procedurom te wersory. Natomiast w liniach 49–50 mamy obliczenie iloczynu skalarnego wersora y' układu obserwatora *przed obróceniem go* za pomocą myszy. Ponieważ w zmiennej `trans.vm` jest w tym momencie przechowywana macierz przejścia do poprzedniego układu obserwatora, część liniowa tego przekształcenia jest opisana przez macierz $R_{z_{i-1},-\phi_{i-1}} = R^T_{z_{i-1},\phi_{i-1}}$. Stąd współrzędne wersora y' są pierwszymi trzema liczbami w drugim wierszu tej macierzy. Jako że liczby te w tablicy `trans.vm` nie sąsiadują ze sobą, procedura, zamiast wywołać procedurę `V3DotProductf`, oblicza iloczyn skalarny „na piechotę".

Obliczony w liniach 49–50 iloczyn skalarny jest kosinusem kąta między osią z układu obserwatora i zenitem. Nominalny kąt obrotu obserwatora, 3°, jest mnożony przez czynnik $0.01 + 0.99|\langle y', z\rangle|$, który przyjmuje wartości z przedziału $[0.01, 1]$. Dzięki temu, jeśli pionizowanie jest włączone, obroty obserwatora są odpowiednio hamowane, co dosyć skutecznie stabilizuje zachowanie programu.

Listing 14.6: Włączanie i wyłączanie pionizowania

```c
void CharFunc ( GLFWwindow *win, unsigned int charcode )
{
  switch ( charcode ) {
........ /* pozostałe klawisze bez zmian */
case 'V':  case 'v':
    if ( (vertical = !vertical) ) {
      Verticalise ( viewer_rvec, &viewer_rangle );
      LoadViewMatrix ();
      redraw = true;
    }
    break;
default:
    break;
  }
} /*CharFunc*/
```

Listing 14.6 przedstawia dodatek do procedury `CharFunc` w aplikacji. Włączenie pionizowania powinno spowodować natychmiastową korektę układu obserwatora, dlatego wywoływane są procedury `Verticalise` i `LoadViewVertex` i zgłaszana jest potrzeba wykonania nowego obrazu. W tym momencie wyjaśniło się, po co powstała osobna procedura `LoadViewMatrix` z instrukcjami przeniesionymi z procedury `RotateViewer`.

15. Aplikacja druga B

Trochę wzbogacimy scenę: dodamy torus lewitujący i obracający się nad wylotem dziobka czajnika. Zastosujemy te same szadery co poprzednio, ale w większym stopniu wykorzystamy ich możliwości[1].

Iloczyn sferyczny i powierzchnie obrotowe

Mając dwie płaskie krzywe parametryczne, $\boldsymbol{e}(u) = [x_e(u), y_e(u)]^T$ i $\boldsymbol{m}(v) = [x_m(v), y_m(v)]^T$, możemy określić ich iloczyn sferyczny, czyli płat powierzchni, którego parametryzacja jest dana wzorem

$$\boldsymbol{p}(u, v) = \begin{bmatrix} x_e(u)x_m(v) \\ y_e(u)x_m(v) \\ y_m(v) \end{bmatrix}.$$

Dziedziną parametryzacji jest prostokąt — iloczyn kartezjański przedziałów będących dziedzinami krzywych $\boldsymbol{e}$ i $\boldsymbol{m}$. Nazwa opisanego wyżej działania wzięła się stąd, że jeśli krzywa $\boldsymbol{e}$ jest okręgiem jednostkowym o środku w punkcie $\boldsymbol{0}$, a krzywa $\boldsymbol{m}$ jest półokręgiem o promieniu R, którego środek i oba końce mają współrzędną x równą 0, to iloczyn sferyczny tych krzywych jest sferą o promieniu R.

Jeśli krzywa $\boldsymbol{e}$ jest opisanym wyżej okręgiem, to iloczyn sferyczny jest powierzchnią obrotową, której tworzącą jest krzywa $\boldsymbol{m}$. W szczególności, jeśli krzywa $\boldsymbol{m}$ jest okręgiem nie przecinającym osi y, to otrzymamy torus.

Łatwo możemy sprawdzić, że jeśli krzywe $\boldsymbol{e}$ i $\boldsymbol{m}$ są krzywymi Béziera,

$$\boldsymbol{e}(u) = \sum_{i=0}^{n} \begin{bmatrix} x_{ei} \\ y_{ei} \end{bmatrix} B_i^n(u), \qquad \boldsymbol{m}(v) = \sum_{j=0}^{m} \begin{bmatrix} x_{mj} \\ y_{mj} \end{bmatrix} B_j^m(v),$$

to ich iloczyn sferyczny jest płatem Béziera stopnia (n, m), o punktach kontrolnych

$$\boldsymbol{p}_{ij} = \begin{bmatrix} x_{ei}x_{mj} \\ y_{ei}x_{mj} \\ y_{mj} \end{bmatrix}.$$

Ponieważ nie istnieje wielomianowa parametryzacja okręgu[2], nie otrzymamy

[1] Przy tej okazji poprawiłem trochę błędów w pierwszej wersji.

[2] Łuki okręgu można z dużą dokładnością przybliżać łukami wielomianowymi, dzięki czemu np. korpus i pokrywka czajnika z Utah wyglądają jak powierzchnie obrotowe, choć nimi nie są.

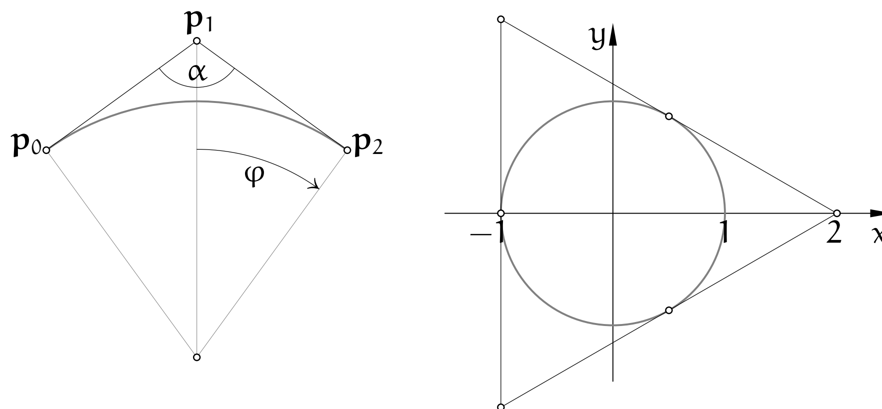
powierzchni obrotowej z wielomianowych krzywych Béziera. Ale możemy w tym celu użyć krzywych wymiernych. Płaskie krzywe wymierne reprezentujemy w postaci jednorodnej, tzn. jako krzywe wielomianowe w przestrzeni trójwymiarowej:

$$\mathbf{E}(u) = \sum_{i=0}^{n} \begin{bmatrix} X_{ei} \\ Y_{ei} \\ W_{ei} \end{bmatrix} B_i^n(u), \qquad \mathbf{M}(v) = \sum_{j=0}^{m} \begin{bmatrix} X_{mj} \\ Y_{mj} \\ W_{mj} \end{bmatrix} B_j^m(v),$$

Punkty kontrolne płata jednorodnego (w przestrzeni $\mathbb{R}^4$) reprezentującego iloczyn sferyczny naszych krzywych wymiernych są dane wzorem

$$\mathbf{P}_{ij} = \begin{bmatrix} X_{ei}X_{mj} \\ Y_{ei}X_{mj} \\ W_{ei}Y_{mj} \\ W_{ei}W_{mj} \end{bmatrix}.$$

Pozostaje skonstruować reprezentację Béziera okręgu. Cały okrąg można reprezentować jako krzywą stopnia 5, a półokrąg jako krzywą stopnia 3, co jest raczej niewygodne. Ale łuk będący dowolną inną częścią okręgu ma reprezentację drugiego stopnia, pokazaną na rysunku 15.1.



Rysunek 15.1: Reprezentacja Béziera łuku okręgu i okrąg złożony z trzech części

Łuk okręgu odpowiadający kątowi 2φ może być reprezentowany jako wymierna krzywa Béziera stopnia 2, której łamana kontrolna ma dwa odcinki o tej samej długości połączone pod kątem $\alpha = \pi - 2\varphi$. Środek okręgu jest punktem przecięcia prostych prostopadłych do tych odcinków, wystawionych w punktach końcowych łamanej, $\mathbf{p}_0$ i $\mathbf{p}_2$. Wagi tych dwóch punktów powinny być równe 1 i wtedy waga punktu $\mathbf{p}_1$ musi być równa $\cos\varphi$. Mając współrzędne kartezjańskie wszystkich trzech punktów, mnożymy je przez odpowiednie wagi i doczepiamy wagę każdego

punktu jako ostatnią (trzecią) współrzędną — dostajemy w ten sposób punkty kontrolne $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2 \in \mathbb{R}^3$ jednorodnej reprezentacji łuku.

Cały okrąg możemy otrzymać jako połączenie trzech łuków o tej samej długości; dla każdego z tych łuków jest $\varphi = \frac{\pi}{3} = 60^\circ$ oraz $\cos\varphi = \frac{1}{2}$. Reprezentacja okręgu jednostkowego o środku w początku układu współrzędnych może wyglądać tak, jak na rysunku 15.1 po prawej stronie.

Konstruowanie reprezentacji torusa

Na listingu 15.1 mamy reprezentację okręgu jednostkowego z rysunku 15.1 i dwie procedury, z których pierwsza konstruuje (i wprowadza do pamięci GPU) iloczyn sferyczny wymiernych krzywych Béziera, a druga przygotowuje tworzącą torusa i wywołuje pierwszą procedurę, aby otrzymać reprezentację torusa gotową do rysowania przez GPU.

W liniach 3–5 jest zadeklarowana tablica z wierzchołkami łamanej składającej się z połączonych łamanych kontrolnych trzech jednorodnych krzywych Béziera reprezentujących okrąg jednostkowy. Symbol SQRT3 reprezentuje liczbę $\sqrt{3}$; druga krzywa ma wspólne punkty kontrolne z krzywymi sąsiednimi.

Listing 15.1: Procedury konstrukcji iloczynu sferycznego i torusa

C

```c
1: #define SQRT3 1.7320508
2:
3: GLfloat UnitCircle[7][3] = {{0.5,0.5*SQRT3,1.0},
4:    {-0.5,0.5*SQRT3,0.5},{-1.0,0.0,1.0},{-0.5,-0.5*SQRT3,0.5},
5:    {0.5,-0.5*SQRT3,1.0},{1.0,0.0,0.5},{0.5,0.5*SQRT3,1.0}};
6:
7: BezierPatchObjf* EnterRSphericalProduct (
8:                      int eqdeg, int eqarcs, int eqstride, GLfloat eqcp[][3],
9:                      int mdeg, int marcs, int mstride, GLfloat mcp[][3],
10:                     GLfloat *colour )
11: {
12:   GLfloat         *cp;
13:   int             i, j, k, nw, nk;
14:   BezierPatchObjf *spr;
15:
16:   nw = (eqarcs-1)*eqstride + eqdeg + 1;
17:   nk = (marcs-1)*mstride + mdeg + 1;
18:   if ( !(cp = malloc ( nw*nk*4*sizeof(GLfloat)) ) )
19:     return NULL;
```

```
20:   for ( i = k = 0;  i < nk;  i++ )
21:     for ( j = 0;  j < nw;  j++, k += 4 ) {
22:       cp[k+0] = eqcp[i][0]*mcp[j][0];
23:       cp[k+1] = eqcp[i][1]*mcp[j][0];
24:       cp[k+2] = eqcp[i][2]*mcp[j][1];
25:       cp[k+3] = eqcp[i][2]*mcp[j][2];
26:     }
27:   spr = EnterBezierPatches ( eqdeg, mdeg, 4, eqarcs, marcs, nw*nk, cp,
28:                              nw*eqstride*4, mstride*4, 4*nw, 4, colour );
29:   free ( cp );
30:   return spr;
31: } /*EnterRSphericalProduct*/
32:
33: BezierPatchObjf* EnterTorus ( float R, float r, GLfloat *colour )
34: {
35:   GLfloat circ[7][3];
36:   int     i;
37:
38:   for ( i = 0; i < 7; i++ ) {
39:     circ[i][0] = r*UnitCircle[i][0] + R*UnitCircle[i][2];
40:     circ[i][1] = r*UnitCircle[i][1];
41:     circ[i][2] = UnitCircle[i][2];
42:   }
43:   return EnterRSphericalProduct ( 2, 3, 2, UnitCircle, 2, 3, 2, circ,
44:                                   colour );
45: } /*EnterTorus*/
```

Pierwsze 4 parametry procedury EnterRSphericalProduct opisują krzywą e złożoną z pewnej liczby (podanej jako wartość parametru eqarcs) płaskich wymiernych krzywych Béziera stopnia n (podanego w parametrze eqdeg). Zmienna eqstride określa odległość w tablicy eqcp między początkami reprezentacji kolejnych krzywych[3].

W taki sam sposób kolejne 4 parametry opisują krzywą m, złożoną z marcs krzywych Béziera stopnia mdeg. Parametr colour określa kolor powierzchni.

W liniach 16 i 17 obliczane są liczby punktów kontrolnych w tablicach eqcp i mcp, a następnie, w linii 18, jest alokowana tablica, w której będą zapamiętane punkty kontrolne powierzchni. W liniach 20–26, na podstawie wzorów podanych wcześniej, są obliczane współrzędne punktów kontrolnych iloczynu sferycznego;

[3] Jeśli krzywe stopnia n mają wspólne punkty kontrolne, jak w wykorzystywanej tu reprezentacji okręgu, to krok powinien być równy n. Ale krzywe mogą być niepołączone i skoro wtedy każda z nich jest reprezentowana przez kolejne $n+1$ punktów w tablicy, trzeba przyjąć krok $n+1$.

zewnętrzna pętla przebiega po kolumnach siatki kontrolnej (z których każda odpowiada jednemu punktowi krzywej e), a wewnętrzna pętla przebiega po kolejnych elementach kolumny.

Po obliczeniu wszystkich punktów kontrolnych, w liniach 27–28, jest wywoływana procedura `EnterBezierPatches` z listingu 13.14, która alokuje odpowiednie bufory (UBO) w pamięci GPU i przesyła do nich dane opisjące płaty. Rysowanie tych płatów będzie odbywać się bez używania tablicy indeksów. Kroki, z jakimi szadery będą „poruszać się” po tablicy punktów kontrolnych, są podane w jednostkach odpowiadających długości *jednej* liczby zmiennopozycyjnej[4]. Tablicę, w której procesor (CPU) umieścił obliczone punkty, można po przesłaniu danych do pamięci GPU oddać do recyklingu (linia 29).

Procedura `EnterTorus` konstruuje reprezentację torusa wyznaczonego przez promień szkieletu R i promień tworzącej r. Korzystając z reprezentacji okręgu jednostkowego w tablicy `UnitCircle`, procedura konstruuje tworzącą torusa, tj. okrąg o promieniu r, którego środek jest w punkcie $[R, 0]^T$. W tym celu wystarczy poddać łamane kontrolne jednokładności o skali r, a następnie przesunąć. Zwróćmy uwagę na sposób obliczania przesunięcia — mając wektor współrzędnych jednorodnych, należy współrzędne wektora przesunięcia pomnożyć przez wagę przesuwanego punktu, tak jak w linii 39.

Obie krzywe, których iloczynem sferycznym jest torus, są reprezentowane przez 7 punktów kontrolnych, przy czym krok w każdej z tablic (parametry trzeci i siódmy w linii 43) jest równy 2.

Zmiany w aplikacji

Ruchy czajnika i torusa są w pewnym sensie niezależne — czajnik ma się obracać, jeśli użytkownik wyda takie polecenie (naciskając klawisz spacji), zaś torus ma się obracać nad dziobkiem czajnika przez cały czas. Dlatego każdy z tych obiektów będzie miał swoją własną macierz przekształcenia modelu, uaktywnianą (tj. przypisywaną do odpowiedniej zmiennej jednolitej) przed jego rysowaniem.

Na listingu 15.2 w linii 1 jest dodana deklaracja wskaźnika struktury opisującej torus, a w liniach 4 i 5 są dodane zmienne przechowujące kąty obrotu czajnika i torusa.

[4] Jednostka miary kroków podawanych jako parametry procedury `EnterRSphericalProduct` jest długością reprezentacji jednego *punktu*, składającego się z trzech liczb. Czy lepiej jest ujednolicić jednostkę używaną w całym programie, czy stosować w każdym miejscu jednostkę najwygodniejszą lub najbardziej naturalną? Nie znam oczywistej odpowiedzi na ten dylemat.

Listing 15.2: Zmiany w aplikacji 2B — przygotowania

C

```
1: BezierPatchObjf *myteapot, *mytorus;
2:
3: float   model_rot_axis[3] = {0.0,0.0,1.0},
4:         teapot_rot_angle0 = 0.0, teapot_rot_angle = 0.0,
5:         torus_rot_angle0 = 0.0, torus_rot_angle = 0.0;
6: GLfloat teapot_mmatrix[16], teapot_mmti[16],
7:         torus_mmatrix[16], torus_mmti[16];
8:
9: void SetModelMatrix ( GLfloat mm[16], GLfloat mmti[16] )
10: {
11:   memcpy ( trans.mm, mm, 16*sizeof(GLfloat) );
12:   memcpy ( trans.mmti, mmti, 16*sizeof(GLfloat) );
13:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
14:   glBufferSubData ( GL_UNIFORM_BUFFER,
15:                     trbofs[0], 16*sizeof(GLfloat), trans.mm );
16:   glBufferSubData ( GL_UNIFORM_BUFFER,
17:                     trbofs[1], 16*sizeof(GLfloat), trans.mmti );
18:   ExitIfGLError ( "SetModelMatrix" );
19:   SetupMVPMatrix ();
20: } /*SetModelMatrix*/
21:
22: void SetupTeapotMatrix ( void )
23: {
24: #define SCF (1.0/3.0)
25:   GLfloat ms[16], mr[16], mt[16], ma[16];
26:
27:   M4x4Scalef ( ms, SCF, SCF, SCF*4.0/3.0 );
28:   M4x4Translatef ( mt, 0.0, 0.0, -0.6 );
29:   M4x4Multf ( ma, mt, ms );
30:   M4x4RotateVf ( mr, model_rot_axis[0], model_rot_axis[1],
31:                  model_rot_axis[2],
32:                  animate ? teapot_rot_angle : teapot_rot_angle0 );
33:   M4x4Multf ( teapot_mmatrix, mr, ma );
34:   M4x4TInvertf ( teapot_mmatrix, teapot_mmti );
35: #undef SCF
36: } /*SetupTeapotMatrix*/
37:
38: void SetupTorusMatrix ( void )
39: {
40:   GLfloat a[16], b[16], c[16], p[4] = {3.1,0.0,2.9,1.0}, q[4];
41:
42:   M4x4Scalef ( a, 0.1, 0.1, 0.1 );
```

```
43:   M4x4RotateXf ( b, 0.5*PI );
44:   M4x4Multf ( c, b, a );
45:   M4x4RotateZf ( a, torus_rot_angle );
46:   M4x4Multf ( b, a, c );
47:   M4x4MultMVf ( q, teapot_mmatrix, p );
48:   M4x4Translatef ( a, q[0], q[1], q[2] );
49:   M4x4Multf ( torus_mmatrix, a, b );
50:   M4x4TInvertf ( torus_mmatrix, torus_mmti );
51: } /*SetupTorusMatrix*/
52:
53: void ConstructMyTorus ( void )
54: {
55:   GLfloat MyColour[4] = { 0.2, 0.3, 1.0, 1.0 };
56:
57:   mytorus = EnterTorus ( 1.0, 0.5, MyColour );
58: } /*ConstructMyTorus*/
59:
60: void InitMyObject ( void )
61: {
62:   TimerInit ();
63:   memset ( &trans, 0, sizeof(TransBl) );
64:   memset ( &light, 0, sizeof(LightBl) );
65:   SetupTeapotMatrix ();
66:   SetupTorusMatrix ();
67:   InitViewMatrix ();
68:   ConstructBezierPatchDomain ();
69:   ConstructMyTeapot ();
70:   ConstructMyTorus ();
71:   SetBezPatchOptions ( BezNormals, TessLevel );
72:   InitLights ();
73: } /*InitMyObject*/
```

Zadeklarowane w liniach 6 i 7 macierze przekształcenia czajnika i torusa oraz transpozycje ich odwrotności są obliczane przez procedury `SetupTeapotMatrix` i `SetupTorusMatrix`, na podstawie kątów obrotu, obliczanych na podstawie zegara.

Procedura `SetupModelMatrix` została z aplikacji usunięta; zamiast niej mamy procedurę `SetModelMatrix`, która otrzymuje jako parametr „gotową” macierz przekształcenia modelu i jej transpozycję odwrotności. Macierze te są przypisywane zmiennym jednolitym `TransBlock.mm` i `TransBlock.mmti`. Obie te macierze są też zapamiętywane w zmiennych `trans.mm` i `trans.mmti`, ponieważ po każdej zmianie macierzy przejścia do układu obserwatora lub do kostki standardowej trzeba obliczyć iloczyn macierzy wszystkich trzech przekształceń.

Obliczenie to wykonuje procedura SetMVPMatrix, która przypisuje ten iloczyn zmiennej jednolitej TransBlock.mvpm.

Instrukcje obliczające macierz przekształcenia czajnika (które w poprzedniej wersji były w procedurze SetupModelMatrix) są obecne w nowej procedurze SetupTeapotMatrix. Wynik obliczenia nie jest tu przesyłany do pamięci GPU, tylko zapamiętywany w zmiennej teapot_mmatrix, ponieważ nadanie odpowiednich wartosci zmiennym jednolitym przechowującym macierze przekształceń trzeba wykonać bezpośrednio przed rysowaniem obiektu, który ma być poddany tym przekształceniom.

Procedura SetupTorusMatrix oblicza i zapamiętuje w zmiennej torus_mmatrix macierz przekształcenia torusa. Przekształcenie to jest złożeniem następujących przekształceń: torus (utworzony przez procedurę ConstructMyTorus) jest zmniejszany 10-krotnie (linia 42), następnie obracany o 90° wokół osi x. Kolejne przekształcenie (linia 43) jest animowane: jest to obrót wokół osi z o kąt zapamiętany w zmiennej torus_rot_angle. Ostatnie przekształcenie to przesunięcie torusa nad dziobek czajnika. Punkt $\mathbf{p}$ o podanych w linii 40 współrzędnych znajduje się nad dziobkiem nieprzekształconego czajnika. W linii 47 poddajemy go takiemu samemu przekształceniu, jak czajnik; współrzędne x, y, z otrzymanego punktu $\mathbf{q}$ wyznaczają przesunięcie odpowiednio obróconego torusa nad dziobek przekształconego czajnika[5]. Macierz tego przesunięcia jest wyznaczana w linii 48. W linii 49 wykonujemy ostatnie mnożenie macierzy i zapamiętujemy wynik w zmiennej torus_mmatrix, a w linii 50 obliczamy i zapamiętujemy transpozycję odwrotności przekształcenia torusa.

Procedury rysowania torusa i jego siatki kontrolnej (DrawMyTorus i TorusCNet) są tak podobne do analogicznych procedur rysowania czajnika, że nie jestem pewien, czy należało je tu (tj. na listingu 15.3) zamieszczać. Ale je zamieściłem.

Do procedury InitMyObject zostało dodane wywołanie procedury konstruującej torus, a wywołanie procedury SetupModelMatrix zostało zastąpione przez wywołaniaprocedur SetupTeapotMatrix i SetupTorusMatrix.

Ponieważ animacja ma działać przez cały czas, w procedurze main przed wywołaniem MessageLoop została dodana instrukcja

```
SetIdleFunc ( IdleFunc );
```

[5]Ponieważ przekształcenie czajnika jest afiniczne (i jego macierz ma w ostatnim wierszu i kolumnie współczynnik 1), a współrzędna wagowa punktu $\mathbf{p}$ jest równa 1, obliczone w ten sposób współrzędne jednorodne X, Y, Z punktu $\mathbf{q}$ są równe jego współrzędnym kartezjańskim x, y, z.

Listing 15.3: Zmiany w aplikacji 2B — rysowanie i interakcja

C

```
1: void DrawMyTorus ( void )
2: {
3:   if ( skeleton ) {
4:     glLineWidth ( 1.0 );
5:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
6:   }
7:   else
8:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
9:   glUseProgram ( program_id[0] );
10:  DrawBezierPatches ( mytorus );
11: } /*DrawMyTorus*/
12:
13: void DrawMyTorusCNet ( void )
14: {
15:   glUseProgram ( program_id[1] );
16:   glLineWidth ( 1.0 );
17:   DrawBezierNets ( mytorus );
18: } /*DrawMyTorusCNet*/
19:
20: void IdleFunc ( void )
21: {
22:   teapot_rot_angle = teapot_rot_angle0 + 0.78539816 * TimerToc ();
23:   torus_rot_angle = torus_rot_angle0 - 2.0*PI*app_time;
24:   SetupTeapotMatrix ();
25:   SetupTorusMatrix ();
26:   redraw = true;
27: } /*IdleFunc*/
28:
29: void Redraw ( GLFWwindow *win )
30: {
31:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
32:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
33:   glEnable ( GL_DEPTH_TEST );
34:   SetModelMatrix ( teapot_mmatrix, teapot_mmti );
35:   DrawMyTeapot ();
36:   if ( cnet )
37:     DrawMyTeapotCNet ();
38:   SetModelMatrix ( torus_mmatrix, torus_mmti );
39:   DrawMyTorus ();
40:   if ( cnet )
41:     DrawMyTorusCNet ();
42:   glUseProgram ( 0 );
```

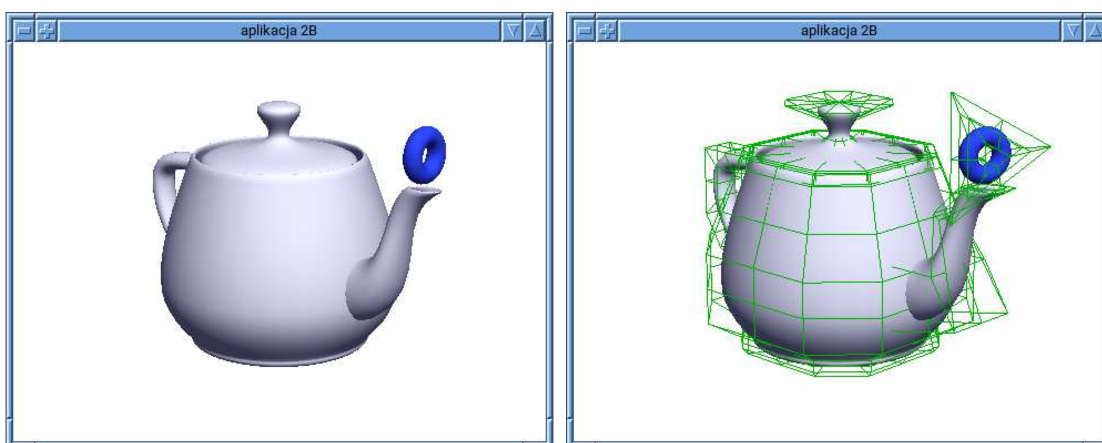
```
43:   glFlush ();
44:   glfwSwapBuffers ( win );
45:   ExitIfGLError ( "Redraw" );
46:   redraw = false;
47: } /*Redraw*/
48:
49: void ToggleAnimation ( void )
50: {
51:   if ( (animate = !animate) )
52:     TimerTic ();
53:   else
54:     teapot_rot_angle0 = teapot_rot_angle;
55: } /*ToggleAnimation*/
56:
57: int main ( int argc, char **argv )
58: {
59:   Initialise ( argc, argv );
60:   SetIdleFunc ( IdleFunc );
61:   MessageLoop ();
62:   Cleanup ();
63:   exit ( 0 );
64: } /*main*/
```

Zarejestrowana przez nią procedura IdleFunc będzie aktywna przez cały czas działania aplikacji. Jej zadaniem jest obliczenie *dwóch* kątów obrotu — czajnika i torusa — oraz wywołanie opisanych wcześniej procedur wyznaczających macierze przekształceń tych obiektów. Uwaga: w obliczeniu kąta obrotu torusa używana jest wartość zmiennej app_time, przypisana jako efekt uboczny wywołania procedury Timer_Toc. Procedura SetupTeapotMatrix powinna być zawsze wywołana *przed* procedurą SetupTorusMatrix, ponieważ ta druga procedura odwołuje się w swoich obliczeniach do macierzy przekształcenia czajnika. Torus obraca się 8 razy szybciej niż czajnik, ale w przeciwną stronę.

Instrukcje dodane do procedury Redraw wywołują procedury rysowania torusa i (na życzenie) jego siatki kontrolnej. Wywołania procedur rysujących są poprzedzone wywołaniami procedury SetModelMatrix z odpowiednim parametrem, aby każdy obiekt został poddany właściwemu przekształceniu.

Wywoływana po naciśnięciu klawisza spacji procedura ToggleAnimation w obecnej wersji nie wyrejestrowuje procedury IdleFunc. Jedynym jej zadaniem jest zapamiętanie czasu rozpoczęcia obracania czajnika i zapamiętanie kąta obrotu w chwili zatrzymania go.

Do procedury sprzątającej `Cleanup` trzeba dodać wywołanie procedury `DeleteBezierPatches` w celu zlikwidowania reprezentacji torusa.



Rysunek 15.2: Czajnik z torusem oraz ich siatki kontrolne

Ćwiczenia

1. Napisz procedurę, która oblicza punkty kontrolne płata Béziera będącego iloczynem sferycznym płaskich wielomianowych krzywych Béziera i dołącz do aplikacji rysowanie obiektu będącego takim płatem.

2. Utwórz reprezentację półokręgu (złożoną z dwóch wymiernych krzywych Béziera opisujących ćwiartki okręgu) i użyj jej do skonstruowania sfery, którą następnie dodaj do sceny.

3. Zmień aplikację tak, aby można było niezależnie włączać i wyłączać obroty czajnika i torusa.

4. Jeśli płat tensorowy $\mathbf{p}$ jest iloczynem sferycznym krzywych $\boldsymbol{e}$ i $\mathbf{m}$, to można przesłać do bufora w pamięci GPU tylko reprezentacje tych dwóch krzywych. Szader rozdrabniania, mając współrzędne u, v punktu w dziedzinie płata, może obliczyć punkty $\boldsymbol{e}(u)$ i $\mathbf{m}(v)$ (albo punkty krzywych jednorodnych $\mathbf{E}(u)$ i $\mathbf{M}(v)$), a następnie obliczyć punkt $\mathbf{p}(u, v)$, podstawiając odpowiednie współrzędne punktów krzywych do wzorów podanych na początku tego rozdziału. Taka reprezentacja płata zajmuje mniej miejsca, a obliczenie punktu zabiera mniej czasu niż pełna reprezentacja i ogólny algorytm przetwarzania płatów tensorowych. Zatem, napisz odpowiedni szader rozdrabniania i procedury w C przesyłające taką reprezentację iloczynu sferycznego krzywych Béziera do pamięci GPU, zmodyfikuj aplikację (dodając do sceny taki płat), *uruchom ją* i wykonaj eksperymenty.

16. Aplikacja druga C

Lambertowski model oświetlenia zastąpimy przez model Blinna–Phonga, który umożliwia osiągnięcie lepszego realizmu obrazu, dzięki wytworzeniu efektu zwierciadlanego odbicia światła. Umożliwia on tworzenie obrazów obiektów, których powierzchnie nie są matowe. Trzeba z mocą podkreślić, że model ten nie jest oparty o analizę fizycznego oddziaływania światła z odbijającą je powierzchnią. Model ten jest empirycznym wzorem, który po dobraniu występujących w nim parametrów dosyć dobrze przybliża skutki oglądania oświetlonej powierzchni. Występujące we wzorze parametry opisują własności materiału i powierzchni, w tym koloru i połysku.

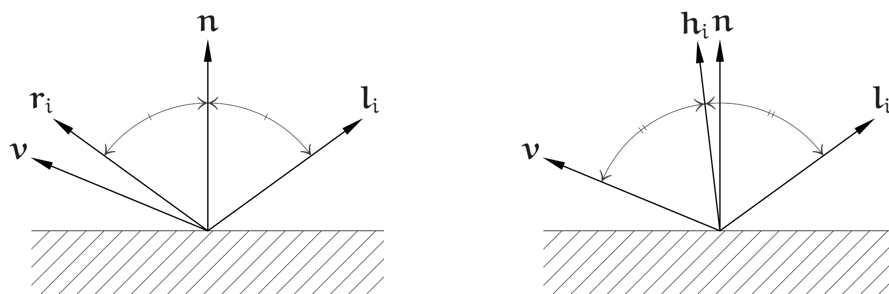
Modele oświetlenia Phonga i Blinna–Phonga

W modelu oświetlenia opracowanym w 1975 r. przez Bui Tuong Phonga do wyrażenia po prawej stronie wzoru (9.1) jest dodany składnik opisujący dodatkowe światło, które dotarło do obserwatora po odbiciu w powierzchni będącej niedoskonałym lustrem:

$$I = \mathbf{a}\sum_{i=0}^{n-1}\big(I_i^{\mathrm{amb}} + I_i^{\mathrm{dir}}\nu_i|\langle \mathbf{l}_i, \mathbf{n}\rangle|\big) + \mathbf{s}\sum_{i=0}^{n-1} I_i^{\mathrm{dir}}\nu_i W\big(|\langle \mathbf{l}_i, \mathbf{n}\rangle|\big)\langle \mathbf{r}_i, \boldsymbol{v}\rangle^m. \qquad (15.1)$$

We wzorze tym symbole $\mathbf{n}$, $\mathbf{l}_i$, $\boldsymbol{v}$ oraz $\mathbf{r}_i$ oznaczają wektory *jednostkowe*, odpowiednio wektor normalny powierzchni, wektor do źródła światła, wektor do obserwatora i wektor idealnego odbicia. Ten ostatni wektor wyznacza kierunek, w którym foton padający na dany punkt powierzchni z kierunku wektora $\mathbf{l}_i$ poleciałby po odbiciu, gdyby ta powierzchnia była idealnym lustrem. Można go obliczyć na podstawie wzoru

$$\mathbf{r}_i = 2\langle \mathbf{n}, \mathbf{l}_i\rangle\mathbf{n} - \mathbf{l}_i.$$



Rysunek 16.1: Wektory w modelach oświetlenia Phonga i Blinna–Phonga

Iloczyn skalarny wektorów jednostkowych $\mathbf{r}_i$ i $\boldsymbol{v}$ jest kosinusem kąta między tymi wektorami; im kąt ten jest mniejszy, tym bliżej prostej, wzdłuż której porusza się

światło idealnie odbite, znajduje się obserwator. Patrząc z kierunków bliskich kierunku wektora $\mathbf{r}_i$, obserwator widzi odblask. Im lepiej powierzchnia jest wypolerowana, tym silniejszy jest ten odblask i jednocześnie tym szybciej zanika on w miarę, jak obserwator oddala się od kierunku idealnego odbicia. Wykładnik m określa szybkość zanikania odblasku (co na całym obrazie powierzchni przekłada się na wielkość obszaru odblasku). Najczęściej w praktyce wykładnik ten jest rzędu kilkunastu lub kilkudziesięciu.

Czynnik ν_i, jak poprzednio, ma wartość 1, jeśli punkt na powierzchni jest bezpośrednio oświetlony przez i-te źródło światła i 0 w przeciwnym razie, tj. gdy obserwator znajduje się po przeciwnej stronie powierzchni niż źródło światła lub jeśli między źródłem światła i danym punktem znajduje się obiekt rzucający cień (ale cieniami na razie jeszcze się nie zajmujemy).

Wektor $\mathbf{s}$ opisuje zdolność powierzchni do lustrzanego odbijania światła o różnych długościach fali; tak jak kolor $\mathbf{a}$ matowej farby (określającej zdolność do odbijania światła w sposób rozproszony), jest on trójką liczb opisujących odbijanie światła czerwonego, zielonego i niebieskiego. W ogólności wektor ten jest inny niż wektor $\mathbf{a}$; jego współrzędne są najczęściej liczbami dodatnimi bliskimi siebie. Można zaobserwować, że odblaski na wielu przedmiotach mają kolor zbliżony do koloru światła *padającego* na powierzchnię.

We wzorze występuje jeszcze funkcja W, której argumentem jest kąt padania światła na powierzchnię (a dokładniej, kosinus tego kąta, obliczony jako iloczyn skalarny wektorów $\mathbf{l}_i$ i $\mathbf{n}$). Często przyjmuje się, że jest to funkcja stała, ale lepszy realizm można osiągnąć, biorąc funkcję rosnącą tego kąta (czyli malejącą ze wzrostem jego kosinusa). Dla kąta 0 (tj. światła padającego prostopadle do powierzchni) funkcja ta powinna być równa 1, a gdy kąt zbliża się do kąta prostego, wartość tej funkcji powinna dążyć do wartości granicznej rzędu kilku.

W 1977 r. James Blinn zmodyfikował model Phonga, otrzymując model trochę prostszy do implementacji. Wzór realizujący ten model ma postać

$$I = \mathbf{a}\sum_{i=0}^{n-1}\left(I_i^{\mathrm{amb}} + I_i^{\mathrm{dir}}\nu_i|\langle \mathbf{l}_i, \mathbf{n}\rangle|\right) + \mathbf{s}\sum_{i=0}^{n-1} I_i^{\mathrm{dir}}\nu_i W\big(|\langle \mathbf{l}_i, \mathbf{n}\rangle|\big)\langle \mathbf{h}_i, \mathbf{n}\rangle^{2m}. \qquad (15.2)$$

Zamiast kosinusa kąta między wektorami $\mathbf{r}_i$ i $\mathbf{n}$ oblicza się tu kosinus kąta między wektorami $\mathbf{h}_i$ i $\mathbf{n}$; wektor $\mathbf{h}_i$ ma kierunek dwusiecznej kąta między wektorami $\mathbf{l}_i$ i $\mathbf{v}$. Można go łatwo obliczyć, normalizując sumę $\mathbf{l}_i + \mathbf{v}$, tj. biorąc

$$\mathbf{h}_i = \frac{1}{\|\mathbf{l}_i + \mathbf{v}\|_2}(\mathbf{l}_i + \mathbf{v}).$$

Zauważmy, że gdyby wektory $\mathbf{l}_i$, $\mathbf{r}_i$ i $\mathbf{v}$ były współpłaszczyznowe, to wektor $\mathbf{h}_i$ też leżałby w ich płaszczyźnie, a kąt między wektorami $\mathbf{h}_i$ i $\mathbf{n}$ byłby dokładnie połową kąta między wektorami $\mathbf{r}_i$ i $\mathbf{n}$ (zobacz rys. 16.1). Aby przybliżyć czynnik $\langle \mathbf{r}_i, \mathbf{v} \rangle^m$ występujący w oryginalnym modelu Phonga, iloczyn skalarny $\langle \mathbf{h}_i, \mathbf{n} \rangle$ we wzorze (15.2) jest podnoszony do potęgi $2m$.

Jest możliwa dodatkowa modyfikacja polegająca na wprowadzeniu jeszcze jednego koloru powierzchni — w ten sposób można użyć różnych kolorów farby odbijającej padające na powierzchnią światło rozproszone w otoczeniu i światło dochodzące bezpośrednio od źródła światła. Wydaje się, że to nie ma wielkiego sensu fizycznego (tak samo, jak używanie różnych kolorów *światła*, które ulegnie odbiciu rozproszonemu i zwierciadlanemu, w starym OpenGL-u jest taka możliwość), ale można w ten sposób osiągać rozmaite efekty specjalne.

Szadery

Użyjemy nowego szadera fragmentów, w którym będą zrealizowane oba modele oświetlenia, do wyboru przez aplikację. Pozostałe szadery zostawimy niezmienione. Wybór modelu oświetlenia zrealizujemy za pomocą wskaźników do procedur GLSL-a — jedna procedura będzie realizować model Lamberta, a druga model Blinna–Phonga; procedura `main` szadera wywoła procedurę wskazywaną w danej chwili przez jednolitą zmienną wskaźnikową[1]. Ponadto dodamy możliwość wybierania źródła opisu własności materiału. Dotychczas kolor powierzchni był podawany na wejście szadera fragmentów w strukturze `NVertex` (zobacz listingi 10.4 i 13.9). Pozostawimy tę możliwość. Na podstawie wartości odpowiedniej zmiennej jednolitej szader albo użyje tego koloru i nada pozostałym parametrom opisującym materiał wartości domyślne, albo przyjmie kompletny opis materiału podany w osobnym bloku zmiennych jednolitych.

Szader fragmentów jest przedstawiony na listingach 16.1–16.4. Szader ten powstał przez modyfikację szadera z listingu 10.4. Na pierwszym listingu są przedstawione globalne deklaracje typów i zmiennych; struktura wejściowa `NVertex`, zmienna wyjściowa `out_Colour` oraz bloki zmiennych jednolitych `TransBlock` (opisującego przekształcenia) i `LSBlock` (opisującego śródła światła), skopiowane bez żadnych zmian, są podane w skrócie.

W liniach 10–15 jest zadeklarowany typ struktury `Material`, której pola opisują

[1] Jeśli szader miałby być skompilowany do postaci SPIR-V, to nie można w nim użyć wskaźników do procedur. Zamiast tego trzeba wprowadzić zmienną jednolitą, która posłuży do wybierania wywoływanej procedury w instrukcji `if` ... `else` ... albo `switch`.

kolory farby odbijającej światło rozproszone w otoczeniu (ambref), farby rozpraszającej światło dochodzące od źródła (dirref) i farby odbijającej światło w sposób zwierciadlany (specref). Pole shininess przechowuje wykładnik $2m$ we wzorze (15.2), a w polach wa i we są parametry a i e funkcji W, danej wzorem[2]

$$W(x) = 1 + (a - 1)(1 - x)^e.$$

Blok MatBlock zawiera strukturę typu Material. Aplikacja może tu przywiązywać różne UBO, opisujące różne materiały, z których są wykonane poszczególne obiekty w scenie. Zmienna jednolita ColourSource służy do wybierania źródła danych opisujących materiał. Zależnie od jej wartości, na początku szader

Listing 16.1: Szader fragmentów — deklaracje typów i zmienne globalne

GLSL

```
1:   ... pierwsze 29 linii jest identyczne, jak na listingu 10.4,
2:   ... dlatego tu jest tylko skrót dla przypomnienia
3:
4: in NVertex { ... } In;
5: out vec4 out_Colour;
6: uniform TransBlock { ... } trb;
7: struct LSPar { ... };
8: uniform LSBlock { ... } light;
9:
10: struct Material {
11:     vec4  ambref;     /* odbijanie światła rozproszonego */
12:     vec4  dirref;     /* odbijanie światła kierunkowego */
13:     vec4  specref;    /* odbijanie zwierciadlane światła */
14:     float shininess, wa, we;  /* parametry połysku */
15:   };
16:
17: uniform MatBlock {
18:     Material mat;
19:   } mat;
20:
21: uniform int ColourSource;
22:
23: subroutine void LightingProc ( void );
24:
25: subroutine uniform LightingProc Lighting;
26:
27: Material mm;
```

[2] Funkcja ta jest dobrana „na wyczucie”, ale spełnia warunki $W(0) = a$ i $W(1) = 1$. Jej argument x jest kosinusem kąta między wektorami $\mathbf{l}_i$ i $\mathbf{n}$.

przypisze polom zmiennej mm zadeklarowanej w linii 27 wartości skopiowane z bloku MatBlock albo wartości domyślne, w tym kolor farby wzięty ze struktury wejściowej NVertex.

W linii 23 jest zadeklarowany typ procedury wskazywanej, realizującej wybrany model oświetlenia. Obie te procedury nie mają parametrów i mają typ wyniku void, ponieważ wszelkie dane i wyniki są przekazywane w zmiennych globalnych. Zmienna wskaźnikowa dla tych procedur jest zadeklarowana w linii 25, a w linii 27 jest zmienna, z której procedura realizująca model oświetlenia bierze dane opisujące materiał.

Listing 16.2: Szader fragmentów — procedury wspólne

GLSL

```
1: void GetMaterial ( void )
2: {
3:   switch ( ColourSource ) {
4: default:
5:     mm.ambref = mm.dirref = In.Colour;
6:     mm.specref = vec4 ( 0.25 );
7:     mm.shininess = 20.0;  mm.wa = mm.we = 1.0;
8:     break;
9: case 1:
10:     mm = mat.mat;
11:     break;
12:   }
13: } /*GetMaterial*/
14:
15: vec3 posDifference ( vec4 p, vec3 pos, out float dist ) { ... }
16: float attFactor ( vec3 att, float dist ) { ... }
17:
18: void main ( void )
19: {
20:   GetMaterial ();
21:   Lighting ();
22: } /*main*/
```

Na listingu 16.2 jest procedura main szadera. Wywołuje ona kolejno procedurę GetMaterial a następnie procedurę wskazywaną przez zmienną Lighting. Procedura GetMaterial nadaje wartości polom zmiennej mm; jeśli zmienna jednolita ColourSource ma wartość 1, to następuje przypisanie danych z bloku MatBlock (o prywatnej nazwie mat). W przeciwnym razie procedura przypisuje kolor przekazany przez szader geometrii oraz domyślne wartości pozostałych

parametrów. Do wyboru jest użyta instrukcja przełącznika (switch), ponieważ w przyszłości szader może zostać rozbudowany o dodatkowe sposoby określania własności materiału, na przykład opisane przez teksturę.

Procedury pomocnicze posDifference i attFactor, używane przez procedury realizujące oba modele oświetlenia, są takie jak na listingu 10.4.

Listing 16.3: Szader fragmentów — model Lamberta

GLSL

```
1: subroutine (LightingProc) void LambertLighting ( void )
2: {
3:   ... /* wszystkie deklaracje i instrukcje procedury main, */
4:   ... /* w liniach 51-78 z listingu 10.4 zostały przeniesione do */
5:   ... /* tej procedury, po czym odwołania do zmiennej wejściowej */
6:   ... /* In.Colour zostały zastąpione przez odwołania do pól */
7:   ... /* mm.ambref i mm.dirref */
8:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
9:     if ( (light.mask & mask) != 0 ) {
10:      out_Colour += light.ls[i].ambient * mm.ambref;
11:      ...
12:          out_Colour += (d * light.ls[i].direct) * mm.dirref;
13:      ...
14:          out_Colour -= (d * light.ls[i].direct) * mm.dirref;
15:    }
16:  out_Colour = vec4 ( clamp ( out_Colour.rgb, 0.0, 1.0 ), 1.0 );
17: } /*LambertLighting*/
```

Na listingu 16.3 jest procedura realizująca model Lamberta; cała treść procedury main szadera z listingu 10.4 została do niej przeniesiona, a następnie zmodyfikowana przez zastąpienie odwołań do zmiennej In.Colour przez odwołania do pól ambref i dirref zmiennej mm. Jeśli taka jest wola aplikacji (a właściwie jej autora), to procedura GetMaterial przypisze wartość pola In.Colour obu tym polom i obiekty będą mieć wygląd identyczny jak w poprzedniej aplikacji. W przeciwnym razie zostaną użyte parametry materiału z bloku zmiennych jednolitych MatBlock.

W linii 16 jest wykonywane obliczenie końcowego koloru fragmentu: funkcja wbudowana clamp każdą współrzędną wektora podanego jako pierwszy parametr kopiuje albo zamienia na 0 lub 1, jeśli współrzędna jest mniejsza niż 0 albo większa niż 1. Współrzędne obliczonego koloru fragmentu zawsze powinny być w przedziale $[0, 1]$. Kanał alfa otrzymuje wartość 1.

Listing 16.4: Szader fragmentów — model Blinna–Phonga

GLSL

```
1: float wFactor ( float lvn, float wa, float we )
2: {
3:   return 1.0+(wa-1.0)*pow ( 1.0-lvn, we );
4: } /*wFactor*/
5:
6: subroutine (LightingProc) void BlinnPhongLighting ( void )
7: {
8:   vec3  normal, lv, vv, hv;
9:   float a, d, e, f, dist;
10:  uint  i, mask;
11:
12:  normal = normalize ( In.Normal );
13:  vv = posDifference ( trb.eyepos, In.Position, dist );
14:  e = dot ( vv, normal );
15:  out_Colour = vec4(0.0);
16:  for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
17:    if ( (light.mask & mask) != 0 ) {
18:      out_Colour += light.ls[i].ambient * mm.ambref;
19:      lv = posDifference ( light.ls[i].position, In.Position, dist );
20:      d = dot ( lv, normal );
21:      if ( e > 0.0 ) {
22:        if ( d > 0.0 ) {
23:          if ( light.ls[i].position.w != 0.0 )
24:            a = attFactor ( light.ls[i].attenuation, dist );
25:          else
26:            a = 1.0;
27:          out_Colour += (a*d*light.ls[i].direct) * mm.dirref;
28:          hv = normalize ( lv+vv );
29:          f = pow ( dot ( hv, normal ), mm.shininess );
30:          out_Colour += (a*f*wFactor(d,mm.wa,mm.we)) * mm.specref;
31:        }
32:      }
33:      else {
34:        if ( d < 0.0 ) {
35:          if ( light.ls[i].position.w != 0.0 )
36:            a = attFactor ( light.ls[i].attenuation, dist );
37:          else
38:            a = 1.0;
39:          out_Colour -= (a*d*light.ls[i].direct) * mm.dirref;
40:          hv = normalize ( lv+vv );
41:          f = pow ( -dot ( hv, normal ), mm.shininess );
42:          out_Colour += (a*f*wFactor(-d,mm.wa,mm.we)) * mm.specref;
```

```
43:        }
44:      }
45:    }
46:  out_Colour = vec4 ( clamp ( out_Colour.rgb, 0.0, 1.0 ), 1.0 );
47: } /*BlinnPhongLighting*/
```

Na listingu 16.4 jest podana procedura realizująca model oświetlenia Blinna–Phonga, razem z pomocniczym podprogramem `wFactor` obliczającym wartość funkcji W we wzorze (15.2). Procedura `BlinnPhongLighting` oblicza składniki oświetlenia modelu lambertowskiego i dodatkowo (w liniach 28–30 oraz 40–42) składniki odpowiedzialne za odbicie zwierciadlane. Pętla przebiegająca po wszystkich określonych i włączonych źródłach światła jest taka sama jak w procedurze `LambertLighting`. W liniach 28 i 40 jest obliczany wektor jednostkowy $\mathbf{h}_i$.

W liniach 29 i 41 jest obliczany czynnik $|\langle \mathbf{h}_i, \mathbf{n} \rangle|^{2m}$ i-tego składnika sumy intensywności światła odbitego w sposób zwierciadlany. Sam składnik jest obliczany i dodawany w liniach 30 i 42. Warunki sprawdzane w liniach 21, 22 i 34 wybierają odpowiednie instrukcje w zależności od tego, czy obserwator znajduje się po tej samej stronie płaszczyzny stycznej do powierzchni, czy po przeciwnej niż i-te źródło światła, tak jak w procedurze dla modelu Lamberta.

Końcowe obliczenie koloru fragmentu w linii 46 (obcinające współrzędne RGB „wystające" poza przedział $[0, 1]$) jest takie jak w procedurze `LambertLighting`.

Zmiany w aplikacji

Procedury na listingu 16.5 służą do utworzenia UBO zawierającego opis materiału, tzn. parametrów mających wpływ na odbijanie światła przez powierzchnię przedmiotu wykonanego z tego materiału, wybranie materiału, z którego jest wykonany obiekt do narysowania, oraz usunięcie opisu materiału. Procedura `SetupMaterial` alokuje (za pośrednictwem procedury `NewUniformBlockObject`) odpowiedni bufor, a następnie wpisuje do niego, w miejscach wskazanych przez (wywołaną wcześniej) procedurę `LoadMyShaders`, przekazane przez aplikację dane. Pierwszy parametr procedury jest numerem materiału — aplikacja określa ten numer. Identyfikatory buforów z opisami materiałów są zapamiętywane w tablicy `matbuf`.

Procedura `ChooseMaterial` przywiązuje UBO z parametrami odpowiedniego materiału do punktu dowiązania bloku zmiennych jednolitych `MatBlock`. Można by ten element rozwiązać inaczej — podobnie do opisów źródeł światła,

Listing 16.5: Procedury określania i wybierania materiałów

C

```
1: #define MAX_MATERIALS 8
2:
3: GLuint matbi, matbuf[MAX_MATERIALS], matbbp;
4: GLint  matbsize, matbofs[6];
5:
6: void SetupMaterial ( int m, const GLfloat ambr[4], const GLfloat diffr[4],
7:                      const GLfloat specr[4],
8:                      GLfloat shn, GLfloat wa, GLfloat we )
9: {
10:   if ( m >= 0 && m < MAX_MATERIALS ) {
11:     matbuf[m] = NewUniformBlockObject ( matbsize, matbbp );
12:     glBindBuffer ( GL_UNIFORM_BUFFER, matbuf[m] );
13:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[0],
14:                       4*sizeof(GLfloat), ambr );
15:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[1],
16:                       4*sizeof(GLfloat), diffr );
17:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[2],
18:                       4*sizeof(GLfloat), specr );
19:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[3],
20:                       sizeof(GLfloat), &shn );
21:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[4], sizeof(GLfloat), &wa );
22:     glBufferSubData ( GL_UNIFORM_BUFFER, matbofs[5], sizeof(GLfloat), &we );
23:     ExitIfGLError ( "SetupMaterial" );
24:   }
25: } /*SetupMaterial*/
26:
27: void ChooseMaterial ( int m )
28: {
29:   if ( m >= 0 && m < MAX_MATERIALS ) {
30:     glBindBufferBase ( GL_UNIFORM_BUFFER, matbbp, matbuf[m] );
31:     ExitIfGLError ( "ChooseMaterial" );
32:   }
33: } /*ChooseMaterial*/
34:
35: void DeleteMaterial ( int m )
36: {
37:   if ( m >= 0 && m < MAX_MATERIALS ) {
38:     glDeleteBuffers ( 1, &matbuf[m] );
39:     ExitIfGLError ( "DeleteMaterial" );
40:   }
41: } /*DeleteMaterial*/
```

które w pamięci GPU są przechowywane w *jednym buforze*, w tablicy w bloku zmiennych jednolitych LSBlock. Wtedy wybór materiału polegałby na przypisaniu dodatkowej zmiennej jednolitej indeksu do tablicy materiałów (i może tak byłoby lepiej). Niemniej, podczas rysowania szader musi mieć dostęp do *wszystkich* źródeł światła, ale wystarczy mu dostęp tylko do jednego materiału na raz, więc to można zaprogramować na oba sposoby. Kto uważnie przeczytał wcześniejsze rozdziały, dla tego rola i treść procedury DeleteMaterial jest oczywista.

Listing 16.6 przedstawia modyfikacje procedury LoadMyShaders z aplikacji 2B. W linii 8 jest nowa nazwa pliku źródłowego szadera fragmentów. Napisy będące nazwami zmiennych jednolitych w liniach 10–16 pozostały niezmienione, w linii 17 jest dopisana nazwa zmiennej ColourSource, a w liniach 18–21 są nazwy pól nowego bloku zmiennych jednolitych MatBlock. W liniach 22–24 mamy nazwę zmiennej wskaźnikowej procedury realizującej model oświetlenia i nazwy procedur wskazywanych przez tę zmienną. W liniach 37–38 uzyskujemy dostęp do pól bloku MatBlock; z przesunięć otrzymanych przez tę instrukcję korzysta opisana wcześniej procedura SetupMaterial. Dostęp do zmiennej jednolitej ColourSource otrzymujemy w linii 41.

Listing 16.6: Procedura LoadMyShaders

```c
1: GLint LightProcInd, LightProcLoc, LightProcInd,
2:       LambertProcInd, BlinnPhongProcInd, ColourSourceLoc;
3:
4: void LoadMyShaders ( void )
5: {
6:   static const char *filename[] =
7:     { "app2.glsl.vert", "app2.glsl.tesc", "app2.glsl.tese",
8:       "app2.glsl.geom", "app2c.glsl.frag",
9:       "app2a1.glsl.vert", "app2a1.glsl.frag" };
10:  static const GLuint shtype[9] = { ... }
11:  static const GLchar *UTBNames[] = { ... }
12:  static const GLchar *ULSNames[] = { ... }
13:  static const GLchar *UCPNames[] = { ... }
14:  static const GLchar *UCPINames[] = { ... }
15:  static const GLchar *UBezPatchNames[] = { ... }
16:  static const GLchar *UVarNames[] =
17:    { "BezNormals", "BezTessLevel", "ColourSource" };
18:  static const GLchar *UMatNames[] =
19:    { "MatBlock", "MatBlock.mat.ambref", "MatBlock.mat.dirref",
20:      "MatBlock.mat.specref", "MatBlock.mat.shininess",
21:      "MatBlock.mat.wa", "MatBlock.mat.we" };
```

```
22:   static const GLchar LightProcName[] = "Lighting";
23:   static const GLchar LambertProcName[] = "LambertLighting";
24:   static const GLchar BlinnPhongProcName[] = "BlinnPhongLighting";
25:
26:   GLint i;
27:
28:   for ( i = 0; i < 7; i++ )
29:     shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
30:   program_id[0] = LinkShaderProgram ( 5, shader_id );
31:   program_id[1] = LinkShaderProgram ( 2, &shader_id[5] );
32:   GetAccessToUniformBlock ( program_id[0], 6, &UTBNames[0],
33:                             &trbi, &trbsize, trbofs, &trbbp );
34:   ... /* tu instrukcje bez zmian */
35:   GetAccessToUniformBlock ( program_id[0], 10, &UBezPatchNames[0],
36:                             &bezpbi, &bezpbsize, bezpbofs, &bezpbbp );
37:   GetAccessToUniformBlock ( program_id[0], 6, &UMatNames[0],
38:                             &matbi, &matbsize, matbofs, &matbbp );
39:   ubeznloc = glGetUniformLocation ( program_id[0], UVarNames[0] );
40:   ubeztloc = glGetUniformLocation ( program_id[0], UVarNames[1] );
41:   ColourSourceLoc = glGetUniformLocation ( program_id[0], UVarNames[2] );
42:   LightProcLoc = glGetSubroutineUniformLocation ( program_id[0],
43:                         GL_FRAGMENT_SHADER, LightProcName );
44:   LambertProcInd = glGetSubroutineIndex ( program_id[0],
45:                         GL_FRAGMENT_SHADER, LambertProcName );
46:   BlinnPhongProcInd = glGetSubroutineIndex ( program_id[0],
47:                         GL_FRAGMENT_SHADER, BlinnPhongProcName );
48:   LightProcInd = LambertProcInd;
49:   trbuf = NewUniformBlockObject ( trbsize, trbbp );
50:   lsbuf = NewUniformBlockObject ( lsbsize, lsbbp );
51:   matbuf[1] = NewUniformBlockObject ( matbsize, matbbp );
52:   matbuf[0] = NewUniformBlockObject ( matbsize, matbbp );
53:   AttachUniformBlockToBP ( program_id[1], UTBNames[0], trbbp );
54:   ... /* dalej instrukcje bez zmian */
55: } /*LoadMyShaders*/
```

Procedura `glGetSubroutineUniformLocation` przekazuje położenie zmiennej wskaźnikowej (wskazującej procedury) o podanej nazwie, w liniach 42–43 jest to zmienna `Lighting`[3]. Dwa wywołania procedury `glGetSubroutineIndex` w liniach

[3] W tej aplikacji jest to instrukcja zbędna; przypisze ona zmiennej `LightProcLoc` wartość 0, ponieważ szader fragmentów zawiera tylko jedną zmienną wskazującą podprogramy. Gdyby takich zmiennych było więcej, np. k, to otrzymają one numery od 0 do $k-1$. Wywołując opisaną dalej procedurę `glUniformSubroutinesuiv`, należy podać jako parametr tablicę indeksów podprogramów, które mają być wskazywane przez poszczególne zmienne wskaźnikowe, uporządkowaną w kolejności numerów przyporządkowanych tym zmiennym.

44–47 uzyskują indeksy procedur LambertLighting i BlinnPhongLighting, które będą wskazywane przez tę zmienną. Ponieważ (jeśli) chcemy, aby na początku działania aplikacji był w użyciu model oświetlenia Lamberta, w zmiennej globalnej LightProcInd, zadeklarowanej w linii 1, zapamiętujemy (w linii 48) indeks procedury LambertLighting.

W liniach 51 i 52 tworzymy dwa bufory o odpowiedniej wielkości; będą w nich przechowywane parametry materiałów dla dwóch obiektów (czajnika i torusa) rysowanych przez aplikację.

Listing 16.7: Wprowadzanie opisów materiałów

```c
 1: void ConstructMyTeapot ( void )
 2: {
 3:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
 4:   const GLfloat ambr[4]  = { 0.75, 0.6, 0.2, 1.0 };
 5:   const GLfloat diffr[4] = { 0.75, 0.6, 0.2, 1.0 };
 6:   const GLfloat specr[4] = { 0.7, 0.7, 0.6, 1.0 };
 7:   const GLfloat shn = 60.0, wa = 5.0, we = 5.0;
 8:
 9:   myteapot = ConstructTheTeapot ( MyColour );
10:   SetupMaterial ( 0, ambr, diffr, specr, shn, wa, we );
11: } /*ConstructMyTeapot*/
12:
13: void ConstructMyTorus ( void )
14: {
15:   GLfloat MyColour[4] = { 0.2, 0.3, 1.0, 1.0 };
16:   const GLfloat ambr[4]  = { 0.0, 1.0, 1.0, 1.0 };
17:   const GLfloat diffr[4] = { 0.0, 0.4, 1.0, 1.0 };
18:   const GLfloat specr[4] = { 0.7, 0.7, 0.7, 1.0 };
19:   const GLfloat shn = 20.0, wa = 2.0, we = 5.0;
20:
21:   mytorus = EnterTorus ( 1.0, 0.5, MyColour );
22:   SetupMaterial ( 1, ambr, diffr, specr, shn, wa, we );
23: } /*ConstructMyTorus*/
```

Procedury ConstructMyTeapot i ConstructMyTorus na listingu 16.7 oprócz obiektów geometrycznych wprowadzają opisy materiałów, z których te obiekty mają być wykonane.

Na końcu procedury InitMyObject zmiennej jednolitej ColourSource przypisujemy początkową wartość 0; wykonuje to instrukcja

```
glUniform1i ( ColourSourceLoc, 0 );
```

W efekcie w chwili uruchomienia aplikacji nie będą używane opisy materiałów z bloku `MatBlock`, tylko opisy „poskładane” przez procedurę `GetMaterial` szadera fragmentów z koloru podanego w bloku wejściowym `In` i domyślnych wartości parametrów połysku.

Listing 16.8: Procedura `DrawMyTeapot`

```c
 1: void DrawMyTeapot ( void )
 2: {
 3:   if ( skeleton ) {
 4:     glLineWidth ( 2.0 );
 5:     glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
 6:   }
 7:   else
 8:     glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
 9:   glUseProgram ( program_id[0] );
10:   glUniformSubroutinesuiv ( GL_FRAGMENT_SHADER, 1, &LightProcInd );
11:   ChooseMaterial ( 0 );
12:   DrawBezierPatches ( myteapot );
13: } /*DrawMyTeapot*/
```

Listing 16.8 przedstawia zmienioną procedurę `DrawMyTeapot`, wywoływaną przez `Redraw` w celu narysowania czajnika. Mamy tu przykład przypisywania wartości zmiennym wskazującym procedury szadera, przez procedurę `glUniformSubroutinesuiv`. Przypisanie następuje jednocześnie dla *wszystkich* zmiennych wskazujących procedury obecnych w bieżąco aktywnym programie szaderów, w etapie określonym przez pierwszy parametr. Drugi parametr jest liczbą zmiennych, którym przypisywane są wartości, a trzeci jest tablicą indeksów kolejnych procedur (w tym przykładzie tablica jest jednoelementowa). Przypisanie wartości zmiennym wskazującym procedury trzeba zrobić *każdorazowo przed* rysowaniem obiektu, ale *po* wywołaniu procedury `glUseProgram`, której jednym ze skutków ubocznych jest nadanie tym zmiennym wskaźnikowym wartości domyślnych — każdej zmiennej będzie przypisana jedna z procedur, które zgodnie z treścią szadera ta zmienna może wskazywać. W procedurze, w linii 10, zmiennej `Lighting` przypisujemy wartość wskazującą procedurę, której indeks jest zapamiętany w zmiennej `LightProcInd` — jak pamiętamy, procedura `LoadMyShaders` przypisała tej zmiennej indeks procedury `LambertLighting` (ale później mogła to zmienić procedura `CharFunc`, opisana dalej).

W linii 11 znajduje się wywołanie procedury `ChooseMaterial`; *jeśli* zmienna jednolita `ColourSource` ma wartość 1, to podczas rysowania czajnika zostanie użyty utworzony dla niego materiał. Zmiana procedury `DrawMyTorus` jest analogiczna.

Listing 16.9: Procedura `CharFunc`

```c
1: void CharFunc ( GLFWwindow *win, unsigned int charcode )
2: {
3:   switch ( charcode ) {
4: ... /* obsługa klawiszy aplikacji 2B bez zmian */
5: case 'B':  case 'b':
6:     LightProcInd = (LightProcInd == LambertProcInd) ?
7:                    BlinnPhongProcInd : LambertProcInd;
8:     redraw = true;
9:     break;
10: case '0':
11:     glUseProgram ( program_id[0] );
12:     glUniform1i ( ColourSourceLoc, 0 );
13:     redraw = true;
14:     break;
15: case '1':
16:     glUseProgram ( program_id[0] );
17:     glUniform1i ( ColourSourceLoc, 1 );
18:     redraw = true;
19:     break;
20: default:    /* ignorujemy wszystkie inne klawisze */
21:     break;
22:   }
23: } /*CharFunc*/
```

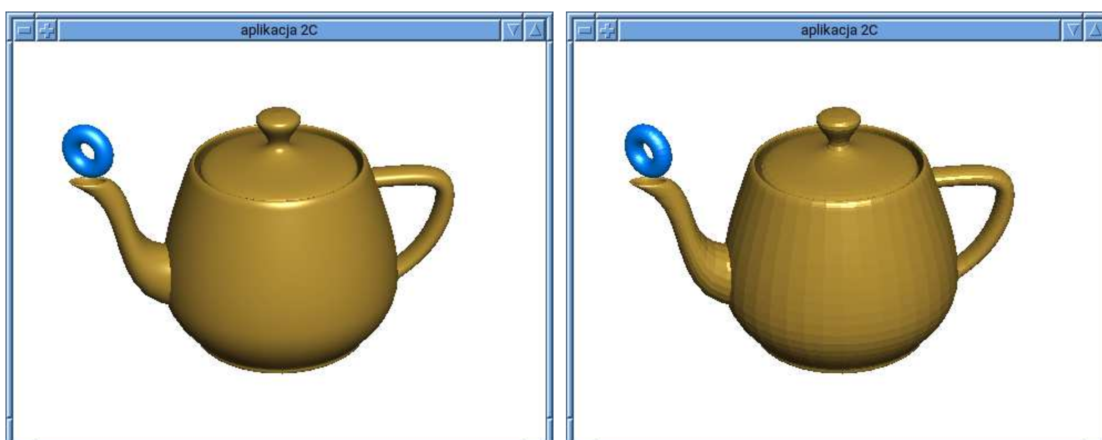
Na listingu 16.9 są pokazane nowe instrukcje procedury `CharFunc`. Po napisaniu znaku `'b'` lub `'B'` następuje przełączenie modelu oświetlenia. Zmiennej `LightProcInd` zostaje przypisany indeks procedury `LambertLighting` albo `BlinnPhongLighting`, po czym procedura odnotowuje (przypisując niezerową wartość zmiennej `redraw`), że należy wykonać nowy obraz.

Napisanie znaku `'0'` lub `'1'` powoduje przypisanie odpowiedniej wartości zmiennej jednolitej `ColourSource` i polecenie wykonania nowego obrazu. Zwróćmy uwagę, że wywołania procedur z rodziny `glUniform` (czyli tu `glUniform1i`, a wcześniej `glUniformSubroutinesuiv`) są poprzedzone

wywołaniem procedury `glUseProgram`, ponieważ „samodzielne” zmienne jednolite[4] znajdują się w domyślnym bloku zmiennych jednolitych programu szaderów; tylko ten jeden program ma do nich dostęp[5], a inne programy szaderów mogą zawierać swoje zmienne jednolite tak samo położone względem początków swoich domyślnych bloków zmiennych jednolitych.

Do procedury `Cleanup` trzeba dopisać wywołania procedury `DeleteMaterial`, o czym piszę na wszelki wypadek, choć chciałbym mieć nadzieję, że już nie muszę.

Okno aplikacji z obrazami wykonanymi przy użyciu modelu Blinna–Phonga jest pokazane na rysunku 16.2. Z lewej strony jest obraz, dla którego wektory normalne zostały obliczone przez szader rozdrabniania (a następnie poddane interpolacji), a z prawej strony wektory normalne obliczył szader geometrii (i też zostały one poddane interpolacji, która dla każdego punktu trójkąta wytworzyła ten sam wektor normalny).



Rysunek 16.2: Efekt użycia modelu oświetlenia Blinna–Phonga

Uzupełnienia

Test nożyczek

Można ograniczyć rysowanie w oknie do dowolnego prostokąta określonego niezależnie od klatki; sprawdzenie, czy dany fragment (piksel) jest w tym

[4]nie będące polami nazwanego bloku zmiennych jednolitych, które znajdują się w odpowiednim buforze przywiązanym do punktu dowiązania widocznego dla wielu programów

[5]„Zwykłe” zmienne jednolite zachowują nadane im wartości podczas kolejnych wywołań procedury `glUseProgram`. Zmienne wskaźnikowe do procedur swoich wartości nie zachowują i dlatego jeśli wartości domyślne, nadawane przez `glUseProgram` nie są tym, co trzeba, należy wywoływać `glUniformSubroutinesuiv` tak jak na listingu 16.8.

prostokącie nazywa się testem nożyczek (*scissor test*). Aby określić ten prostokąt, należy wywołać procedurę

```
glScissor ( x, y, w, h );
```

której parametry określają współrzędne (w układzie okna OpenGL-a) dolnego lewego wierzchołka oraz szerokość i wysokość prostokąta w pikselach. Uaktywnienie testu następuje przez wykonanie instrukcji

```
glEnable ( GL_SCISSOR_TEST );
```

a do jego wyłączenia służy procedura `glDisable` (z parametrem jak wyżej).

Wczesne testy fragmentu

Domyślnie szader fragmentów jest wywoływany *przed* wykonaniem testów nożyczek, szablonu[6] i widoczności, które mogą spowodować odrzucenie fragmentu. Jeśli obliczenie wykonywane przez szader fragmentów zabiera znaczącą ilość czasu — a to może mieć miejsce w przypadku stosowania porządnego modelu oświetlenia i będzie miało miejsce w dalszych wersjach aplikacji, nakładających tekstury na obiekty — a potem fragment okaże się niewidoczny (bo inny, przetworzony wcześniej fragment odwzorowany na ten sam piksel ma mniejszą głębokość), to czas zużyty przez szader na przetwarzanie bieżącego fragmentu jest zmarnowany. Dlatego można zażądać, by wspomniane testy były wykonane przed wywołaniem szadera fragmentów, dzięki czemu szader nie będzie wywoływany dla fragmentów, które nie przeszły któregoś z testów. Żądanie to zgłasza się, podając kwalifikator wejścia szadera fragmentów, o postaci

```
layout(early_fragment_tests) in;
```

Dlaczego w domyślnej kolejności szader fragmentów jest wywoływany przed wspomnianymi testami? Bo szader fragmentów może, w razie potrzeby (w celu wpłynięcia na wynik testu widoczności), zmienić głębokość fragmentu przez przypisanie wartości zmiennej wbudowanej `gl_FragDepth`[7]. Gdyby test widoczności został wykonany wcześniej, to nowa głębokość byłaby fragmentowi przypisana poniewczasie.

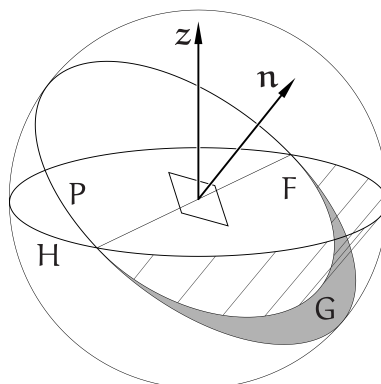
[6] Test sprawdzający, czy piksel nie jest w obszarze zabronionym do rysowania, reprezentowanym przez zawartość bufora szablonu (*stencil buffer*), o którym na razie nie napisałem.

[7] Głębokość fragmentu obliczona w etapie rasteryzacji jest wartością zmiennej wbudowanej `gl_FragCoord.z`; zmienne `gl_FragCoord.x` i `gl_FragCoord.y` zawierają współrzędne punktu w oknie, zaś `gl_FragCoord.w` ma wartość 1.

Oświetlenie hemisferyczne

Jeśli obiekt jest oświetlony przez jedno punktowe źródło światła, to część obiektu odwrócona tyłem do tego źródła jest oświetlona tylko przez światło rozproszone w otoczeniu. Efekt jest kompletnie nieplastyczny: zamiast detali kształtu obiektu na obrazie widać tylko plamę w jednolitym kolorze. Aby uzyskać lepszy obraz, można „włączyć” kilka źródeł, oświetlających przedmiot z różnych stron. Inna metoda polega na wprowadzeniu dochodzącego ze wszystkich stron światła rozproszonego, którego intensywność zmienia się z kierunkiem, z jakiego światło to dochodzi.

Najprostszy model takiego światła to oświetlenie hemisferyczne; sferę jednostkową, której punkty reprezentują kierunki, podzielimy na dwie półsfery. Światło padające z kierunków górnej półsfery ma intensywność i kolor „nieba” (czyli jest dosyć jasne i białe lub niebieskie), zaś światło dochodzące od dołu jest znacznie słabsze i ma kolor „gruntu”, na którym stoi obiekt (może to być też kolor trawy). Zakładamy, że intensywność i kolor światła w każdej półsferze są stałe. Rysunek 16.3 przedstawia sferę jednostkową, w środku której znajduje się fragment powierzchni obiektu. Zaznaczone są wektory jednostkowe $\boldsymbol{z}$ i $\boldsymbol{n}$; pierwszy z nich ma kierunek zenitu, a drugi jest wektorem normalnym powierzchni. Okręgi jednostkowe H i P leżą w płaszczynach prostopadłych do wektorów $\boldsymbol{z}$ i $\boldsymbol{n}$.[8]



Rysunek 16.3: Model oświetlenia hemisferycznego

Założymy, że rozpatrywane tu światło odbija się od powierzchni przedmiotu „po lambertowsku”, tj. wkład w końcowy kolor piksela na obrazie światła, które dochodzi do powierzchni z pewnego kierunku, jest proporcjonalny do kosinusa kąta padania. Przy tych założeniach możemy obliczyć współczynnik t opisujący

[8]Okrąg H odpowiada horyzontowi; zenit to kierunek wektorów prostopadłych do jego płaszczyzny i zorientowanych w stronę „nieba”.

proporcję, w jakiej światła pochodzące z górnej i dolnej półsfery są zmieszane po odbiciu od powierzchni. Do prawej strony wzoru (9.1), (15.1) lub (15.2) dodamy składnik

$$\boldsymbol{a}\big((1-t)I^{\text{sky}} + tI^{\text{ground}}\big), \tag{15.3}$$

w którym wektor $\boldsymbol{a}$ opisuje kolor powierzchni, a wektory I^{sky} oraz I^{ground} reprezentują intensywności świateł dochodzących z obu półsfer. W zasadzie można też użyć tylko tego składnika, tj. pominąć punktowe źródła światła obecne we wcześniej rozpatrywanych modelach.

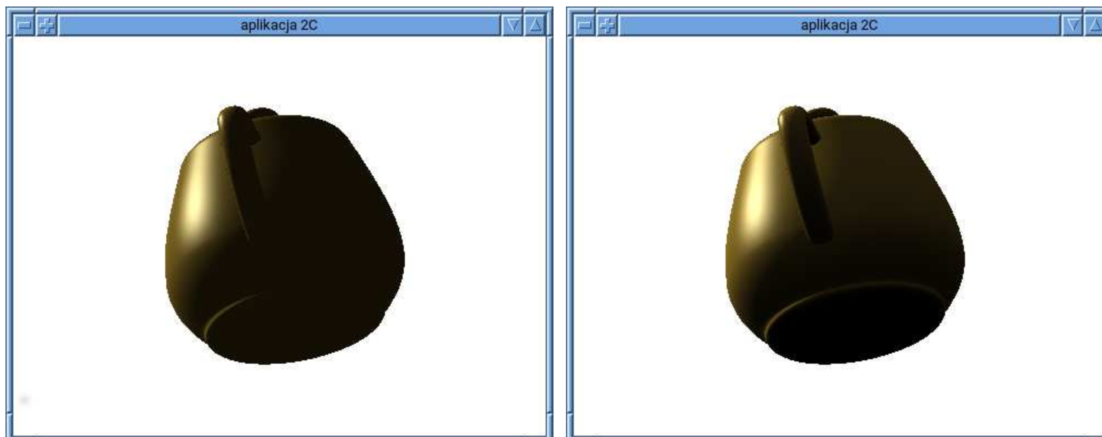
Całkowita intensywność światła odbitego od powierzchni jest iloczynem współczynnika $\boldsymbol{a}$ i całki po jednej z dwóch półsfer rozdzielonych okręgiem P położonym w płaszczyźnie stycznej do powierzchni — wybieramy oczywiście tę półsferę, której elementem jest wektor $\boldsymbol{v}$ mający kierunek do obserwatora, czyli tę, której punkty reprezentują kierunki, z których dochodzi światło padające na stronę powierzchni widzianą przez obserwatora. Funkcja podcałkowa to intensywność światła dochodzącego z kierunku $\boldsymbol{l}$ pomnożona przez kosinus kąta między wektorami $\boldsymbol{l}$ i $\boldsymbol{n}$ (co odpowiada modelowi Lamberta odbicia światła).

Rozważmy tak mały fragment odpowiedniej półsfery, aby jego punkty były wektorami o *prawie tym samym* kierunku wektora (jednostkowego) $\boldsymbol{l}$. Pole fragmentu pomnożone przez kosinus kąta między wektorami $\boldsymbol{l}$ i $\boldsymbol{n}$ jest równe polu rzutu prostopadłego tego fragmentu na płaszczyznę styczną do powierzchni. Zobaczmy rysunek. Rzuty wszystkich fragmentów półsfery, po której całkujemy, wypełniają koło o brzegu P. Jeśli $\langle \boldsymbol{v}, \boldsymbol{n}\rangle > 0$, tj. obserwator znajduje się po wskazywanej przez wektor $\boldsymbol{n}$ stronie płaszczyzny stycznej do powierzchni, to rzutem przecięcia górnej półsfery (o brzegu H) i półsfery kierunków oświetlających tę stronę powierzchni jest figura F, której brzeg składa się z połowy okręgu P i połowy elipsy będącej rzutem prostopadłym okręgu H na płaszczyznę styczną do powierzchni. Pozostałą część koła (figurę G, narysowaną w kolorze szarym) wypełniają rzuty odpowiednich fragmentów dolnej półsfery. W ten sposób obliczenie wspomnianej całki sprowadzone zostało do znalezienia pól figur F i G. Współczynnik t we wzorze (15.3) jest ilorazem pola figury G i pola całego koła[9]. Możemy zauważyć, że jest on równy $(1-\cos\theta)/2$, gdzie θ jest kątem między wektorami $\boldsymbol{z}$ i $\boldsymbol{n}$ (zatem $t = (1 - \langle \boldsymbol{z}, \boldsymbol{n}\rangle)/2$). Jeśli obserwator znajduje się po przeciwnej stronie powierzchni, to proporcja podziału koła na części składające się z rzutów odpowiednich fragmentów górnej i dolnej półsfery jest odwrotna. Do

[9]pole to jest oczywiście równe π

wzoru (15.3) należy zatem podstawić

$$t = \begin{cases} (1-\cos\theta)/2 = (1-\langle z, n\rangle)/2 & \text{jeśli } \langle v, n\rangle > 0, \\ (1+\cos\theta)/2 = (1+\langle z, n\rangle)/2 & \text{jeśli } \langle v, n\rangle \leqslant 0. \end{cases}$$



Rysunek 16.4: Obrazy otrzymane bez i z oświetleniem hemisferycznym

Ćwiczenia

1. Napisz procedurę realizującą model oświetlenia Phonga opisany wzorem (15.1).
2. Wykonaj eksperymenty polegające na zmienianiu parametrów materiału i oglądaniu skutków.
3. Rozbuduj szader fragmentów tak, aby można było „pomalować” różnymi „farbami” dwie strony powierzchni. Wybór „farby” powinien być dokonany na podstawie znaku iloczynu skalarnego wektora do obserwatora i wektora normalnego, $\langle v, n\rangle$. Iloczyn ten jest obliczany np. w linii 14 na listingu 16.4.
4. Zmodyfikuj aplikację tak, aby dodać oświetlenie hemisferyczne. Wymaga to m.in. dołączenia do szadera fragmentów odpowiednich zmiennych jednolitych (pola opisujące wektor z i wektory I^{sky} i I^{ground} najlepiej dodać do bloku `LSBlock`, przyda się tam też zmienna sterująca „włączaniem” i „wyłączaniem” tego oświetlenia). Kod aplikacji w C należy uzupełnić o uzyskiwanie dostępu do nowych pól w bloku oraz o procedury przypisujące tym polom odpowiednie wartości (pamiętaj, aby wektor z był jednostkowy).

17. Aplikacja druga D

Dodamy dwa ważne elementy syntezy obrazów: teksturowanie i antyaliasing. Tekstura jest określoną na rysowanej powierzchni funkcją, której (skalarna lub wektorowa) wartość w każdym punkcie ma wpływ na kolor tego punktu na obrazie; może to być po prostu kolor punktu albo pewien parametr występujący w modelu oświetlenia, na przykład kolor farby, połysk, faktura (tj. chropowatość, rysy lub inne zaburzenia kształtu), przezroczystość, lub wreszcie radiancja światła emitowanego przez powierzchnię. Wiele z tych czynników, określanych niezależnie, występuje jednocześnie, skąd wynika potrzeba jednoczesnego używania wielu tekstur dla jednej powierzchni, choć na początek wprowadzimy jedną. Bardzo często teksturę opisuje się za pomocą jedno- dwu- lub trójwymiarowej tablicy wartości. Elementy takiej tablicy są przez analogię do pikseli nazywane tekselami. Dwuwymiarowa tekstura często jest obrazem, który można po prostu wyświetlić na ekranie (lub wydrukować) i wtedy teksele są tożsame z pikselami. Inna możliwość opisu to tekstura proceduralna, której wartość w danym punkcie powierzchni oblicza (przy użyciu jakiegoś wzoru) odpowiedni podprogram. Tekstura trójwymiarowa może opisywać materiał, z którego jest wykonany przedmiot, na przykład kolor słojów drewna. Punkty powierzchni otrzymują kolory odpowiadające przecięciu tej powierzchni z obszarem trójwymiarowym, w którym tekstura jest określona.

Antyaliasing jest to przeciwdziałanie artefaktom będącym skutkami ograniczonej rozdzielczości rastra monitora. Jeśli dla każdego piksela zostanie obliczony kolor jednego punktu (na hipotetycznym obrazie o nieskończonej rozdzielczości) i kolor ten zostanie przypisany pikselowi, to na ekranie cały obszar piksela będzie miał kolor tego jednego punktu. W efekcie na wspólnym brzegu obszarów o różnych kolorach pojawią się ząbki o wysokości lub szerokości jednego piksela, tym lepiej widoczne, im bardziej kolory tych obszarów kontrastują ze sobą. Zjawisko to można zaobserwować na obrazach wykonanych przez wszystkie wersje obu aplikacji opisanych w poprzednich rozdziałach. To zjawisko, zwane intermodulacją (*alias*), może jeszcze bardziej popsuć jakość obrazu obiektów pokrytych teksturą. Dlatego podstawową technikę antyaliasingu (która z grubsza polega na obliczaniu i mieszaniu kolorów wielu punktów w obszarze każdego piksela) wprowadzimy jednocześnie z teksturami.

Mipmaping

Nałożymy teksturę na część korpusu czajnika składającą się z czterech największych płatów Béziera. Będzie to tekstura dwuwymiarowa, otrzymana

z czterech fotografii. W obszarze pokrytym przez teksele (obszarze tekstury) są określone współrzędne tekstury, które przyjmują w tym obszarze wartości z przedziału $[0,1]$. Fragmenty obrazu reprezentowanego przez teksturę chcemy odpowiednio zniekształcić (porozciągać lub skurczyć) i nałożyć na poszczególne płaty. Można to zrobić w ten sposób, że dziedzinę płata (która jest kwadratem jednostkowym) utożsamiamy z pewnym prostokątem w obszarze tekstury; ustanawia to odwzorowanie punktów z tego prostokąta na punkty płata Béziera w przestrzeni i dalej, po rzutowaniu, na punkty na obrazie.

Ponieważ każdy piksel jest prostokątem, opisane wyżej przekształcenia wiążą nieskończenie wiele punktów w obszarze tekstury z tylomaż punktami na obrazie należącymi do piksela; punkty te tworzą pewien podobszar obszaru tekstury, w którym tekstura na ogół przyjmuje różne wartości. Podobszar ten może być mały albo duży, może też mieć kształt zbliżony do kwadratu lub wydłużony. Rzecz w tym, że w obliczeniu koloru piksela może być konieczne uwzględnienie wielu tekseli — jeśli obszar tekstury odwzorowany na piksel jest duży (bo na przykład cały obiekt na obrazie jest bardzo mały).

Użyjemy stosunkowo prostej i dosyć skutecznej (choć nie zawsze wystarczającej) techniki, zaproponowanej w 1983 r. przez Lance'a Williamsa, który nazwał ją mipmapingiem (*MIP-mapping*, od łacińskiego *multum in parvo*, wiele w niewielu). Polega ona na wykorzystaniu dodatkowych tablic zawierających reprezentacje danej tekstury o mniejszych rozdzielczościach — dwukrotnie, czterokrotnie, ośmiokrotnie itd. Tablica o największej rozdzielczości ma poziom 0, a każda kolejna tablica ma poziom o jeden większy. W najprostszym przypadku można te tablice otrzymać przez zwykłe uśrednianie wartości czwórek tekseli na wyższym poziomie, choć lepsze efekty daje *filtrowanie sygnału*, jakim jest tekstura (tj. obliczanie splotu tekstury z odpowiednio dobraną funkcją zwaną filtrem, czym tu nie będziemy się zajmować). Dla tekstury dwuwymiarowej dodatkowe tablice zajmują (w pamięci GPU) jedną trzecią miejsca potrzebnego do przechowania tekstury o pełnej rozdzielczości. Aby obliczyć kolor piksela, szader fragmentów sięga do tekstury za pomocą funkcji `texture`. Funkcja ta dokonuje interpolacji wartości odpowiednich tekseli, przy czym zależnie od wielkości odpowiadającego pikselowi obszaru w teksturze wybierany jest odpowiedni poziom, na którym teksele mają wartości uśrednione w odpowiednich obszarach; poziom tym wyższy, im większy jest obszar odpowiadający pikselowi — szczegóły będą opisane dalej.

Na rysunku 17.1 jest pokazana tekstura o pełnej rozdzielczości 1024×1024 teksele oraz dodatkowe reprezentacje tej tekstury o mniejszych rozdzielczościach.

Rysunek 17.1: Tekstura i jej reprezentacje o mniejszej rozdzielczości

Szadery

Do szaderów używanych w poprzedniej wersji aplikacji trzeba wprowadzić przetwarzanie współrzędnych tekstury i zapewnić dostęp do samej tekstury. Wektor współrzędnych tekstury jest nowym atrybutem wierzchołka wytwarzanego przez szader rozdrabniania. Podczas rasteryzacji trójkątów będących wynikiem rozdrabniania płata, tak jak inne atrybuty wierzchołków, współrzędne tekstury są interpolowane. Otrzymany w wyniku interpolacji wektor jest przekazywany jako dana wejściowa do szadera fragmentów, który *może* (w zależności od wartości zmiennej jednolitej używanej do sterowania szaderem) użyć tekstury do otrzymania parametrów materiału występujących w modelu oświetlenia. Wektor współrzędnych tekstury dla wierzchołka jest wytwarzany przez szader rozdrabniania na podstawie współrzędnych punktu w dziedzinie płata[1].

Cała scena tworzona przez naszą aplikację składa się z płatów Béziera, rysowanych jednocześnie jako poszczególne instancje płata o czterech wierzchołkach. Ponieważ szader wierzchołków ma za zadanie tylko przekazać do dalszych obliczeń numer instancji, nie ma w nim żadnych zmian w porównaniu z wersją aplikacji 2C. To samo dotyczy szadera sterowania rozdrabnianiem.

[1]Można go też wygenerować w inny sposób, na przykład na podstawie współrzędnych wierzchołka (przez podstawienie ich do jakiegoś wzoru), lub podać jako atrybut wierzchołka w VAO (wtedy będzie on przekazany na wejście szadera wierzchołków przez etap pobierania wierzchołków), co jednak z naszym sposobem rysowania płatów Béziera jest niewykonalne.

Zmiany dokonane w szaderze rozdrabniania są pokazane na listingu 17.1. Do struktury wyjściowej GVertex (czyli Out) zostało dodane pole TexCoord typu vec2. Jest też nowy blok zmiennych jednolitych o nazwie BezPatchTexCoord (i nazwie lokalnej txc). Jeśli na płat ma być nałożona tekstura (co ma miejsce, gdy wartość zmiennej jednolitej ColourSource jest równa 2), to szader ma nadać

Listing 17.1: Modyfikacje szadera rozdrabniania

GLSL

```
1: #version 450 core
2:
3: layout(quads,equal_spacing,cw) in;
4: in TCInstance { ... } In[];
5: out GVertex {
6:     vec4 Colour;
7:     vec3 Position;
8:     vec3 Normal;
9:     vec2 TexCoord;
10:   } Out;
11:
12: ... /* niezmienione wszystkie dotychczasowe zmienne jednolite */
13: uniform bool BezNormals;
14: ...
15: uniform TransBlock { ... } trb;
16:
17: uniform BezPatchTexCoord {
18:     vec4 txc[1];
19:   } txc;
20:
21: int inst;
22:
23: ... /* niezmienione wszystkie procedury oprócz main */
24:
25: void main ( void )
26: {
27:   vec4 pos, nv;
28:
29:   inst = In[0].instance;
30:   ... /* pominięty fragment bez zmian */
31:   Out.Colour = bezp.Colour;
32:   if ( ColourSource == 2 )
33:     Out.TexCoord = mix ( txc.txc[inst].xy, txc.txc[inst].zw,
34:                          gl_TessCoord.yx );
35: } /*main*/
```

współrzędnym pola Out.TexCoord wartości liczbowe. Jeśli obie liczby należą do przedziału $[0, 1]$, to reprezentują punkt należący do obszaru tekstury, która zostanie użyta (przez szader fragmentów) do obliczenia koloru piksela.
W przeciwnym razie tekstura w obliczeniu koloru zostanie zignorowana.

Przyjrzyjmy się obliczaniu współrzędnych tekstury. Szader rozdrabniania ma dostarczyć punkt jednego z n płatów Béziera, wskazanego przez numer instancji podany w zmiennej In[0].instance i zapamiętany dla wygody w zmiennej inst (linia 29). Dla każdego z tych płatów jest podana informacja o prostokącie zawartym w obszarze tekstury (tj. w kwadracie jednostkowym), w odpowiednim elemencie tablicy txc w bloku zmiennych jednolitych BezPatchTexCoord. Element ten jest typu vec4; jego pierwsze dwie i ostatnie dwie współrzędne określają dwa punkty będące przeciwległymi wierzchołkami rozpatrywanego prostokąta; pierwszy z nich utożsamimy z punktem $(0, 0)$, a drugi z punktem $(1, 1)$ dziedziny *płata*. Etap rozdrabniania dziedziny dostarczył (w zmiennej gl_TessCoord) współrzędne punktu w dziedzinie płata. Procedury wywoływane przez pominięty fragment procedury main obliczają odpowiadający mu punkt płata i jego wektor normalny w tym punkcie. Aby otrzymać współrzędne tekstury, dokonujemy interpolacji wierzchołków podanych w zmiennej txc.txc[inst] przy użyciu współrzędnych punktu w dziedzinie płata, za pomocą funkcji mix; ponieważ wszystkie trzy parametry tej funkcji są wektorami, interpolacja odbywa się niezależnie dla każdej współrzędnej. Zwróćmy uwagę na przestawienie współrzędnych x, y punktu w dziedzinie płata (w wyrażeniu gl_TessCoord.yx). Jest ono potrzebne, aby obrazki na czajniku były właściwie zorientowane[2].

Modyfikacje szadera geometrii z listingu 13.9 są pokazane na listingu 17.2. Do bloku wejściowego GVertex i wyjściowego NVertex zostało dodane pole TexCoord; oprócz wykonywania dotychczasowych zadań szader kopiuje dane w tym polu z bloku wejściowego do wyjściowego.

Nowe elementy szadera fragmentów są pokazane na listingu 17.3. Do bloku wejściowego NVertex jest dodane nowe pole TexCoord, zawierające współrzędne tekstury. Ponadto w linii 21 widzimy deklarację zmiennej jednolitej tex typu sampler2D; dostęp do tekstury dostajemy przez tę zmienną, reprezentującą

[2]Orientacja tekstury na płacie Béziera jest ściśle związana z orientacją parametryzacji płata; można ją zmieniać, odwracając kolejność wierszy siatki kontrolnej, odwracając kolejność kolumn, lub zamieniając wiersze z kolumnami — możemy w ten sposób otrzymać 8 różnych orientacji. Dla dowolnego płata w modelu czajnika należałoby w tym celu znaleźć właściwy wiersz w tablicy teapotcn streszczonej na listingu 13.17 i ustawić podane w nim 16 liczb w macierz 4×4, a następnie odwracać kolejność wierszy lub kolumn, albo transponować, a potem przepisać te liczby z powrotem wiersz po wierszu do tablicy.

Listing 17.2: Modyfikacje szadera geometrii

GLSL

```
#version 450 core
... /* podstawowe wejście i wyjście bez zmian */

in GVertex {
  vec4 Colour;
  vec3 Position;
  vec3 Normal;
  vec2 TexCoord;
} In[];

out NVertex {
  vec4 Colour;
  vec3 Position;
  vec3 Normal;
  vec2 TexCoord;
} Out;

void main ( void )
{
  ... /* deklaracje zmiennych i początkowe instrukcje bez zmian */
  for ( i = 0; i < 3; i++ ) {
    ... /* instrukcje w pętli też bez zmian */
    Out.Colour = In[i].Colour;
    Out.TexCoord = In[i].TexCoord;
    EmitVertex ();
  }
  EndPrimitive ();
} /*main*/
```

ewaluator tekstury — obiekt pełniący rolę pomocniczą (ale niezbędną) podczas obliczania wartości tekstury. Zmieniona procedura `GetColours`, w przypadku, gdy zmienna jednolita `ColourSource` ma wartość 2, tworzy opis materiału wykorzystywany następnie do obliczania koloru oświetlonej powierzchni na podstawie tekstury, chyba że któraś ze współrzędnych tekstury przetwarzanego fragmentu jest poza przedziałem $[0, 1]$ — wtedy używany jest opis materiału podany w bloku zmiennych jednolitych `MatBlock`. Ten mechanizm umożliwia nałożenie tekstury tylko na wybrane płaty Béziera. Tekstura jest traktowana jak kolor farby w danym punkcie powierzchni i przypisywana do pól `ambref` i `dirref` zmiennej `mm`. Pola te opisują zdolność odbijania światła rozproszonego w otoczeniu i światła dochodzącego bezpośrednio od źródła światła i ulegającego odbiciu rozproszonemu. Parametry opisujące odbijanie światła w sposób

Listing 17.3: Modyfikacje szadera fragmentów

GLSL

```
1: #version 450 core
2:
3: in NVertex {
4:     vec4 Colour;
5:     vec3 Position;
6:     vec3 Normal;
7:     vec2 TexCoord;
8:   } In;
9:
10: uniform TransBlock { ... } trb;
11: struct LSPar { ... };
12: uniform LSBlock { ... } light;
13: struct Material { ... };
14: uniform MatBlock { ... } mat;
15:
16: uniform int ColourSource;
17: Material mm;
18: subroutine void LightingProc ( void );
19: subroutine uniform LightingProc Lighting;
20:
21: uniform sampler2D tex;
22:
23: void GetColours ( vec2 tc )
24: {
25:   switch ( ColourSource ) {
26: default: ... break;
27: case 1: ...  break;
28: case 2:
29:     mm = mat.mat;
30:     if ( tc.x >= 0.0 && tc.x <= 1.0 && tc.y >= 0.0 && tc.y <= 1.0 )
31:       mm.ambref = mm.dirref = texture ( tex, tc );
32:     break;
33:   }
34: } /*GetColours*/
35:
36: ... /* wszystkie pozostałe procedury bez zmian */
37:
38: void main ( void )
39: {
40:   GetColours ( In.TexCoord );
41:   Lighting ();
42: } /*main*/
```

zwierciadlany są brane z opisu materiału w bloku zmiennych jednolitych `mat`.

Drugi program szaderów, używany do wyświetlania siatek kontrolnych płatów Béziera, został zachowany bez żadnych zmian.

Czytanie i pisanie plików TIFF

Fotografie, które mają stać się teksturami są zapisane w plikach w formacie TIFF; na początku działania aplikacja czyta te pliki i tworzy na ich podstawie teksturę w pamięci GPU. Odkładając na chwilę na bok główny temat, tj. OpenGL, zobaczmy pomocniczą procedurę, której zadaniem jest przeczytanie takiego pliku i przekazanie aplikacji obrazu w nim zawartego. Procedura ta użyje procedur z biblioteki `libtiff`, którą trzeba zainstalować z odpowiedniego pakietu, jeśli nie jest to jeszcze zrobione. Oprócz standardowych plików nagłówkowych, takich jak

Listing 17.4: Czytanie pliku TIFF

C

```c
1: GLubyte *ReadTiffImage ( const char *fn, int *width, int *height )
2: {
3:   TIFF    *tif;
4:   int     w, h, npix;
5:   GLubyte *image;
6:
7:   image = NULL;
8:   if ( (tif = TIFFOpen ( fn, "r" )) ) {
9:     TIFFGetField ( tif, TIFFTAG_IMAGEWIDTH, &w );
10:    TIFFGetField ( tif, TIFFTAG_IMAGELENGTH, &h );
11:    *width = w;
12:    *height = h;
13:    npix = w*h;
14:    image = malloc ( 4*npix );
15:    if ( !image )
16:      goto way_out;
17:    memset ( image, 0, npix*sizeof(uint32) );
18:    if ( !TIFFReadRGBAImage ( tif, w, h, (uint32*)image, 0 ) )
19:      { free ( image );  image = NULL; }
20: way_out:
21:    TIFFClose ( tif );
22:  }
23:  return image;
24: } /*ReadTiffImage*/
```

`string.h` (z prototypami procedur `malloc`, `free`, `memset` i `memcpy`) oraz plików nagłówkowych OpenGL-a (z definicją typu `GLubyte`), trzeba włączyć do programu plik nagłówkowy tej biblioteki, `tiffio.h`.

Procedura `ReadTiffFile` na listingu 17.4 otrzymuje jako parametr wejściowy nazwę pliku i zwraca wskaźnik do tablicy, w której są umieszczone wartości pikseli przeczytanego obrazu. Jeśli obrazu nie uda się przeczytać (bo nie ma takiego pliku, jest on nieczytelny lub zabrakło pamięci), to procedura zwraca wskaźnik pusty. Jeśli czytanie pliku zakończyło się sukcesem, to zmiennym wskazywanym przez parametry `width` i `height` zostają przypisane wymiary obrazu — szerokość i wysokość w pikselach.

Wartość każdego piksela w tablicy jest zapisana w jednej liczbie 32-bitowej (składających się z czterech bajtów, kolejno RGBA); piksele w tablicy są uporządkowane wierszami. Po wykorzystaniu danych przeczytanych z pliku tablicę, której początku adres został przekazany przez procedurę `ReadTiffImage`, należy zwolnić przy użyciu procedury `free`.

Aby w książce był komplecik, na listingu 17.5 zamieściłem procedurę, która odczytuje teksturę (tj. tablicę tekseli) z pamięci GPU i zapisuje ją do pliku TIFF. Nie jest ona używana w opisanej tu aplikacji, ale może się przydać podczas uruchamiania programów. Parametr `fn` jest nazwą pliku TIFF, parametr `tex` jest identyfikatorem tekstury o szerokości `w` i wysokości `h` tekseli. Odczytania tekseli dokonuje procedura `glGetTextureImage`. Jest tu pewna niespójność procedur OpenGL-a. Otóż wiersze pikseli w plikach takich jak TIFF są przechowywane w kolejności „od góry do dołu”. Procedury `glTexImage2D` i `glTextureSubImage2D` otrzymują tablice o takim właśnie uporządkowaniu wierszy i przesyłają je przestawiając wiersze, aby otrzymać wewnętrzną reprezentację tekstury zgodną z konwencją przyjętą w standardzie OpenGL (w której wiersz `0` leży na prostej $y = 0$ w układzie współrzędnych tekstury, prosta $y = 1$ zawiera wiersz `h-1`, przy czym oś y jest zorientowana do góry). Procedura `glGetTextureImage` wpisuje wiersze tekstury do tablicy przekazanej jako parametr bez przestawiania. Zatem aby obraz nie był zapisany w pliku „do góry nogami”, trzeba dodać instrukcję odwracającą kolejność wierszy; jest nią pętla w liniach 14–20.

Ciąg wywołań procedury `TIFFSetField` w liniach 21–31 zapamiętuje w obiekcie utworzonym przez procedurę `TIFFOpen` niezbędne informacje (pewne parametry, takie jak rodzaj stosowanej kompresji, ustaliłem arbitralnie). Procedura `TIFFWriteEncodedStrip` dokonuje kompresji obrazu i zapisuje go w pliku.

Listing 17.5: Zapisywanie tekstury w pliku TIFF

C

```
1: void SaveTiffTexture ( const char *fn, GLuint tex, int w, int h )
2: {
3:   TIFF *tif;
4:   unsigned char *image, *a, *b, *c;
5:   int  n, i;
6:
7:   image = malloc ( 3*w*(h+1)*sizeof(char) );
8:   if ( !image )
9:     return;
10:  if ( (tif = TIFFOpen ( fn, "w" )) ) {
11:    n = w*h;
12:    glGetTextureImage ( tex, 0, GL_RGB, GL_UNSIGNED_BYTE,
13:                        3*n*sizeof(char), image );
14:    for ( i = 0, a = image, b = &image[3*w*(h-1)], c = &image[3*n];
15:          i+i < h;
16:          i++, a += 3*w, b -= 3*w ) {
17:      memcpy ( c, a, 3*w*sizeof(char) );
18:      memcpy ( a, b, 3*w*sizeof(char) );
19:      memcpy ( b, c, 3*w*sizeof(char) );
20:    }
21:    TIFFSetField ( tif, TIFFTAG_IMAGEWIDTH, w );
22:    TIFFSetField ( tif, TIFFTAG_IMAGELENGTH, h );
23:    TIFFSetField ( tif, TIFFTAG_BITSPERSAMPLE, 8 );
24:    TIFFSetField ( tif, TIFFTAG_SAMPLESPERPIXEL, 3 );
25:    TIFFSetField ( tif, TIFFTAG_ROWSPERSTRIP, h );
26:    TIFFSetField ( tif, TIFFTAG_COMPRESSION, COMPRESSION_LZW );
27:    TIFFSetField ( tif, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB );
28:    TIFFSetField ( tif, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG );
29:    TIFFSetField ( tif, TIFFTAG_XRESOLUTION, 300.0 );
30:    TIFFSetField ( tif, TIFFTAG_YRESOLUTION, 300.0 );
31:    TIFFSetField ( tif, TIFFTAG_RESOLUTIONUNIT, RESUNIT_INCH );
32:    TIFFWriteEncodedStrip ( tif, 0, image, 3*n );
33:    TIFFClose ( tif );
34:  }
35:  free ( image );
36: } /*SaveTiffTexture*/
```

Procedury przygotowania tekstur

Na listingu 17.6 są przedstawione trzy procedury pomocnicze, których zadaniem jest utworzenie i przygotowanie do pracy reprezentacji tekstury w pamięci GPU. Parametr procedury `CreateMyTexture` jest wysokością i szerokością tekstury

w tekselach; jest tu założenie, że oba te wymiary są jednakowe, choć nie jest to konieczne. W linii 6 procedura wywołuje procedurę glGenTextures; pierwszy jej parametr podaje liczbę obiektów tekstur do utworzenia, a drugi parametr jest tablicą (w tym przypadku jednoelementową), do której zostaną wpisane identyfikatory obiektów tekstur utworzonych przez glGenTextures. W tych obiektach trzeba jeszcze umieścić odpowiednie dane; wszystko po kolei.

W linii 7 nowy obiekt tekstury jest przywiązywany do celu GL_TEXTURE_2D, czego późniejszym skutkiem będzie określenie, że jest to tekstura dwuwymiarowa.

W pętli w liniach 8–9 obliczana jest liczba poziomów tekstury dla mipmapingu; jest to liczba l, taka że 2^l jest największym dzielnikiem wysokości i szerokości wh tekstury. Procedura glTextureStorage2D wywołana w linii 10 zapisuje w obiekcie informacje o liczbie poziomów i o wymiarach tablicy tekseli na każdym poziomie. Trzeci parametr, o wartości GL_RGBA8, określa, że każdy teksel ma być zapisany w czterech bajtach, które przechowują odpowiednio składowe RGBA. Dysponując tymi informacjami, procedura glTextureStorage2D dokonuje alokacji odpowiedniego obszaru w pamięci GPU.

Zadaniem procedury LoadMyTextureImage jest przeczytanie pliku w formacie TIFF o nazwie podanej w parametrze filename i przesłanie przeczytanych danych do tablicy w pamięci GPU zaalokowanej przez procedurę CreateMyTexture. Założenie jest takie, że wymiary (w pikselach) przeczytanego obrazu nie przekraczają wymiarów tekstury (w tekselach). Procedura sprawdza (w linii 23), czy przeczytany obraz, przesunięty o wektor (x, y), którego współrzędne są podane jako parametry, mieści się w obszarze tekstury i jeśli tak, to wywołuje procedurę glTextureSubImage2D, która przesyła dane do związanej z obiektem tekstury tablicy w pamięci GPU, na poziomie 0. Tablica z pikselami zaalokowana przez procedurę ReadTiffImage jest następnie zwalniana.

Procedura SetupMyTextureMipmaps powinna zostać wywołana po przeczytaniu i przesłaniu do tablicy na poziomie 0 wszystkich danych (tj. wszystkich obrazów, z których ma być utworzona tekstura). W linii 38 jest wywoływana procedura glGenerateTextureMipmap, która na podstawie zawartości tablicy tekseli na poziomie 0 oblicza i wpisuje reprezentacje o mniejszej rozdzielczości do tablic na wszystkich wyższych poziomach. Kolejne dwie instrukcje ustawiają w obiekcie tekstury (a dokładniej, w będącym jego częścią ewaluatorze, ang. *sampler*) parametry określające sposób odtwarzania wartości tekstury na podstawie tablic tekseli.

Listing 17.6: Procedury przygotowania tekstur

C

```
1: GLuint CreateMyTexture ( int wh )
2: {
3:   GLuint tex;
4:   int    w, l;
5:
6:   glGenTextures ( 1, &tex );
7:   glBindTexture ( GL_TEXTURE_2D, tex );
8:   for ( w = wh, l = 0;  !(w & 0x01);  l++ )
9:     w >>= 1;
10:  glTextureStorage2D ( tex, l, GL_RGBA8, wh, wh );
11:  ExitIfGLError ( "CreateMyTexture" );
12:  return tex;
13: } /*CreateMyTexture*/
14:
15: char LoadMyTextureImage ( GLuint tex, int txwidth, int txheight,
16:                           int x, int y, const char *filename )
17: {
18:   int     w, h;
19:   GLubyte *image;
20:
21:   if ( !(image = ReadTiffImage ( filename, &w, &h )) )
22:     return 0;
23:   if ( x+w <= txwidth && y+w <= txheight ) {
24:     glTextureSubImage2D ( tex, 0, x, y, w, h,
25:                           GL_RGBA, GL_UNSIGNED_BYTE, image );
26:     free ( image );
27:     ExitIfGLError ( "LoadMyTextureImage" );
28:     return 1;
29:   }
30:   else {
31:     free ( image );
32:     return 0;
33:   }
34: } /*LoadMyTextureImage*/
35:
36: void SetupMyTextureMipmaps ( GLuint tex )
37: {
38:   glGenerateTextureMipmap ( tex );
39:   glTextureParameteri ( tex, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
40:   glTextureParameteri ( tex, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );
41:   ExitIfGLError ( "SetupMyTextureMipmaps" );
42: } /*SetupMyTextureMipmaps*/
```

Pierwszy parametr procedury glTextureParameteri jest identyfikatorem obiektu tekstury. Drugi parametr procedury jest nazwą parametru w obiekcie, któremu nadajemy wartość, a trzeci określa tę wartość. Parametr GL_TEXTURE_MAG_FILTER określa sposób postępowania, gdy obraz teksela na ekranie jest większy niż piksel.

Nadanie temu parametrowi wartości GL_LINEAR jest równoznaczne z żądaniem, aby wartość tekstury podawana szaderowi fragmentów przez funkcję texture (zobacz listing 17.3) była otrzymana przez liniową interpolację wartości odpowiednich tekseli w tablicy na poziomie 0.

Parametr GL_TEXTURE_MIN_FILTER określa sposób postępowania, gdy obrazy tekseli są mniejsze od piksela na ekranie. Jeśli parametr ten otrzymał wartość GL_LINEAR_MIPMAP_LINEAR, to wyznaczane są dwa sąsiednie poziomy najlepiej pasujące do wielkości obrazu tekseli: teksele z poziomu niższego mają obrazy „trochę za małe”, a teksele z poziomu wyższego mają obrazy „trochę za duże” w porównaniu z rozmiarmi piksela. Dla każdego z tych poziomów jest wykonywana interpolacja liniowa wartości tekseli, a potem otrzymane wartości tekstury są interpolowane liniowo „między poziomami”. Taki sposób obliczania wartości tekstury dla fragmentów jest trochę czasochłonny, ale ma dobre skutki dla końcowej jakości obrazu.

Zmiany w aplikacji

Zmiana procedury LoadMyShaders polega na zmienieniu nazw plików z trzema szaderami opisanymi wcześniej oraz na dodaniu instrukcji dających dostęp do bloku zmiennych jednolitych BezPatchTexCoord; w odpowiedniej tablicy z nazwami są napisy "BezPatchTexCoord" i "BezPatchTexCoord.txc"; trzeba tam jeszcze dodać instrukcję

```
GetAccessToUniformBlock ( program_id[0], 1, &UTexCoordNames[0],
                          &txcbi, &i, txcofs, &txcbp );
```

podając jako parametry adresy nowych zmiennych globalnych txcbi, txcofs i txcbp, do których zostaną wpisane indeks bloku BezPatchTexCoord, przesunięcie pola txc w tym bloku[3] i numer punktu dowiązania dla tego bloku. Mniemam, że Czytelnik umie już wskazać właściwe miejsce w kodzie dla tych zmian.

Na końcu procedury InitMyObject trzeba dopisać wywołanie procedury LoadMyTextures pokazanej na listingu 17.7. Jej zadaniem jest przeczytanie

[3] Ono jest równe 0, bo to jest jedyne pole w tym bloku.

czterech plików TIFF z obrazami i utworzenie z nich tekstury.

Na listingu 17.8 są pokazane zmienione procedury tworzenia reprezentacji i rysowania czajnika. Chcemy, aby tekstura była nałożona tylko na 4 z 32 płatów Béziera, z których składa się czajnik, a ponadto chcemy na każdy z tych czterech płatów nałożyć inny fragment tekstury, tj. zdjęcie przeczytane z innego pliku TIFF. Służy do tego tablica `txc` w bloku zmiennych jednolitych `BezPatchTexCoord`; jej długość jest liczbą płatów, czyli dla czajnika 32.
W strukturze `BezierPatchObjf` (listing 13.10) wydłużymy tablicę `buf` o jeden element, potrzebny do przechowywania identyfikatora bufora z blokiem `BezPatchTexCoord` dla zbioru płatów Béziera reprezentowanego przez tę strukturę.

Listing 17.7: Tworzenie tekstury przez aplikację

C

```c
1: GLuint mytexture, mytxcbuf;
2:
3: void LoadMyTextures ( void )
4: {
5:   mytexture = CreateMyTexture ( 1024 );
6:   LoadMyTextureImage ( mytexture, 1024, 1024,   0,   0, "jaszczur.tif" );
7:   LoadMyTextureImage ( mytexture, 1024, 1024, 512,   0, "salamandra.tif" );
8:   LoadMyTextureImage ( mytexture, 1024, 1024,   0, 512, "lis.tif" );
9:   LoadMyTextureImage ( mytexture, 1024, 1024, 512, 512, "kwiatki.tif" );
10:  SetupMyTextureMipmaps ( mytexture );
11: } /*LoadMyTextures*/
```

W buforze, który przywiążemy do odpowiedniego punktu dowiązania, aby był on widziany przez szader jako blok `BezPatchTexCoord`, umieścimy dane z tablicy `txc` pokazanej w liniach 6–11. Mamy tam 32 czwórki liczb typu `GLfloat`, albo 32 wektory typu <u>`vec4`</u>; każdy z nich odpowiada jednemu płatowi Béziera. Cztery płaty, na które nakładamy teksturę, mają numery 4, 5, 6 i 7. Przykładowo, dla płata o numerze 4 mamy podane liczby `0.5,0.5,0.0,0.0`. Wierzchołek $(0,0)$ dziedziny tego płata zostanie zatem odwzorowany na punkt $(\frac{1}{2},\frac{1}{2})$ w dziedzinie tekstury, a wierzchołek $(1,1)$ na punkt $(0,0)$. To oznacza, że na płat o numerze 4 będzie nałożona odpowiednio obrócona lewa dolna ćwiartka tekstury pokazanej na rysunku 17.1, czyli obraz jaszczurki. Wszystkie płaty oprócz wymienionych czterech mają być niepokryte teksturą. Do osiągnięcia tego efektu służą ujemne liczby podane w pozostałych elementach tablicy `txc` — procedura `GetColours` na listingu 17.3 w razie otrzymania ujemnych współrzędnych tekstury przekazuje opis materiału z bloku `MatBlock`, bez uwzględnienia tekstury.

Listing 17.8: Tworzenie reprezentacji i rysowanie czajnika z teksturą

C

```
1: void ConstructMyTeapot ( void )
2: {
3:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
4:   ... /* parametry materiału czajnika takie jak poprzednio */
5:   const GLfloat txc[32][4] =
6:     {{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},
7:      {-1.0,0.-1,0.0,0.0},{0.5,0.5,0.0,0.0},{0.5,1.0,0.0,0.5},
8:      {1.0,0.5,0.5,0.0},{1.0,1.0,0.5,0.5},{-1.0,0.0,-1.0,0.0},
9:      {-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0},
10:      ... /* tu 6 kopii linii podanej wyżej */
11:      {-1.0,0.0,-1.0,0.0},{-1.0,0.0,-1.0,0.0}};
12:
13:   myteapot = ConstructTheTeapot ( MyColour );
14:   SetupMaterial ( 0, ambr, diffr, specr, shn, wa, we );
15:   myteapot->buf[3] = NewUniformBlockObject ( 32*4*sizeof(GLfloat), txcbp );
16:   glBufferSubData ( GL_UNIFORM_BUFFER, txcofs[0], 32*4*sizeof(GLfloat),
17:                     txc );
18:   ExitIfGLError ( "ConstructMyTeapot" );
19: } /*ConstructMyTeapot*/
20:
21: void DrawMyTeapot ( void )
22: {
23:   ... /* początkowe instrukcje bez zmian */
24:   ChooseMaterial ( 0 );
25:   glUniform1i ( ColourSourceLoc, colour_source );
26:   if ( colour_source == 2 ) {
27:     glBindBufferBase ( GL_UNIFORM_BUFFER, txcbp, myteapot->buf[3] );
28:     glBindTexture ( GL_TEXTURE_2D, mytexture );
29:     DrawBezierPatches ( myteapot );
30:     glBindTexture ( GL_TEXTURE_2D, 0 );
31:   }
32:   else
33:     DrawBezierPatches ( myteapot );
34: } /*DrawMyTeapot*/
```

Zatem: procedura `NewUniformBlockObject` wywołana w linii 15 tworzy UBO dla bloku zmiennych jednolitych `BezPatchTexCoord` (z danymi będącymi odtąd częścią modelu czajnika) i przywiązuje go do celu o numerze wygenerowanym przez `GetAccessToUniformBlock` i przechowywanym w zmiennej `txcbp`. Procedura `glBufferSubData` w liniach 16–17 przesyła do tego bufora dane z tablicy `txc` zadeklarowanej w procedurze `ConstructMyTeapot`.

Procedura DrawMyTeapot przygotowuje rysowanie czajnika tak, jak w aplikacji drugiej C, po czym przypisuje zmiennej jednolitej ColourSource wartość sterującą wyborem materiału[4]. Jeśli przypisana jest wartość 2, to w linii 27, za pomocą procedury BindBufferBase przywiązujemy UBO z tablicą txc czajnika, a w linii 28 wywołujemy procedurę glBindTexture, która przywiązuje naszą teksturę do celu GL_TEXTURE_2D — tekstura jest dwuwymiarowa, jej wymiar musi się zgadzać z typem zmiennej jednolitej tex, czyli sampler2D. Po narysowaniu poteksturowanego czajnika, w linii 30, odłączamy teksturę od zmiennej jednolitej tex.

Procedura konstrukcji torusa pozostała niezmieniona, natomiast do procedury rysowania torusa, DrawMyTorus, została dodana instrukcja

```
glUniform1i ( ColourSourceLoc, colour_source > 0 ? 1 : 0 );
```

Jeśli zmienna colour_source ma wartość 2, to zmiennej jednolitej ColourSource jest przypisywana wartość 1, aby torus został narysowany bez tekstury, z wykorzystaniem materiału, jaki został dla niego zdefiniowany.

Do procedury CharFunc została dodana reakcja na napisanie na klawiaturze cyfry '2'. Po jego naciśnięciu jest zmiennej colour_source jest przypisywana wartość 2, czego skutkiem jest używanie tekstury. Mamy więc trzy opcje rysowania: początkowo i po napisaniu '0' torus i czajnik są rysowane z wykorzystaniem materiału domyślnego, zdefiniowanego w treści szadera fragmentów, po napisaniu '1' czajnik i torus są rysowane przy użyciu materiałów zdefiniowanych dla każdego z nich, a po napisaniu '2' czajnik (a raczej jego część) jest teksturowany.

Niezręcznie jest mi o tym wspominać, ale do procedury Cleanup wypada dodać instrukcję

```
glDeleteTextures ( 1, &mytexture );
```

a w procedurze DeleteBezierPatches zwiększyć pierwszy parametr procedury glDeleteBuffers z 3 do 4.

[4]Przypisanie wartości zmiennej ColourSource jest wykonywane bezpośrednio przed rysowaniem czajnika, bo w scenie mamy jeszcze torus, a on nie jest teksturowany, więc podczas rysowania go zmienna ColourSource nie może mieć wartości 2 — dla torusa nie wprowadziliśmy UBO z tablicą txc, więc szader nie może wtedy odwoływać się do bloku BezPatchTexCoord.

Antyaliasing

Najprostsza technika antyaliasingu polega na wielokrotnym próbkowaniu (*multisampling*). Domyślnie dla każdego piksela obrazu oblicza się kolor jednego punktu, odpowiadającego środkowi piksela, i nadaje pikselowi ten kolor. Wielokrotne próbkowanie polega na obliczeniu kolorów pewnej liczby (kilku lub kilkunastu) punktów w obszarze piksela i zmieszanie tych kolorów w celu otrzymania wynikowego koloru piksela na obrazie. To oznacza w szczególności, że szader fragmentów będzie wywołany kilka lub kilkanaście razy więcej razy, a więc wykonanie obrazu zabierze odpowiednio więcej czasu.

Aby użyć tej techniki w aplikacji biblioteki `libglfw`, *tuż przed* utworzeniem okna (za pomocą procedury `glfwCreateWindow`) trzeba wykonać instrukcję

```
glfwWindowHint ( GLFW_SAMPLES, n );
```

przy czym n oznacza liczbę punktów (obliczanych próbek koloru) na piksel.

Po utworzeniu okna z wielopróbkowym buforem obrazu antyaliasing jest (domyślnie) włączony, zatem obrazy będą wykonywane z wielokrotnym próbkowaniem. Można to wyłączyć, wykonując instrukcję

```
glDisable ( GL_MULTISAMPLE );
```

co spowoduje (szybsze) rysowanie obrazów gorszej jakości (tj. bez antyaliasingu). Wielokrotne próbkowanie można ponownie włączyć, wywołując procedurę `glEnable` z parametrem jak wyżej.

Uwaga: Na końcu tego rozdziału jest opisany sposób tworzenia okna z buforem wielopróbkowym w aplikacji FreeGLUTa. Należy jednak liczyć się z tym, że opisany tu sposób osiągnięcia antyaliasingu nie zadziała. Przedstawiona w tym rozdziale aplikacja po uruchomieniu na moim laptopie, po utworzeniu okna z opcją (wskazówką) wielokrotnego próbkowania działa normalnie, ale wyświetla obrazy „ząbkowane", takie jak bez wielokrotnego próbkowania. Prawdopodobnie polecenie utworzenia bufora obrazu z wielokrotnym próbkowaniem dla tworzonego okna aplikacji jest ignorowane, jeśli komputer nie ma wystarczających zasobów (np. pamięci GPU). Pozostaje się z tym pogodzić lub użyć sposbu opisanego w rozdziale ??.

Rysunek 17.2: Okno aplikacji drugiej D

Ćwiczenia

1. Zapisz w formacie TIFF obrazki lub fotografie ze *swojej* kolekcji, skalując je do rozsądnej rozdzielczości, np. rzędu kilkuset pikseli wzdłuż i wszerz (jest wiele programów, których można do tego użyć, np. GIMP). W razie potrzeby powiększ rozdzielczość tekstury tworzonej przez aplikację i umieść w tej teksturze swoje obrazki. Następnie zmodyfikuj zawartość tablicy `txc` w procedurze `ConstructMyTeapot`, aby nałożyć teksturę na inne płaty Béziera w modelu czajnika.

2. Zmodyfikuj procedurę `ConstructMyTorus`, dodając tablicę `txc` oraz instrukcję, która tworzy UBO z zawartością tej tablicy dla torusa i wykorzystaj ją do nałożenia tekstury na 9 wymiernych płatów Béziera, z których składa się torus.

3. Dla eksperymentu wyłącz antyaliasing i mipmaping. W tym celu wykomentuj wywołanie procedury `glfwWindowHint` przed utworzeniem okna, w procedurze `CreateMyTexture` zmień na 1 drugi parametr wywołania procedury `glTextureStorage2D` (aby był tylko jeden poziom tekstury) i wykomentuj wywołanie procedury `SetupMyTextureMipmaps`. Możesz użyć jakiegoś klawisza do włączania i wyłączania wielokrotnego próbkowania za pomocą procedur `glEnable` i `glDisable`. Skompiluj aplikację, uruchom i obejrzyj skutki.

4. Wypróbuj różne liczby punktów na piksel w wielokrotnym próbkowaniu, podajac procedurze `glfwWindowHint` jako drugi parametr liczby od 1 do 16 i oceń (według własnego gustu) jakość obrazów.

Uzupełnienia

Do tworzenia tekstur dwuwymiarowych służy ciąg instrukcji podobny do podanego niżej:

```
glGenTextures ( GL_TEXTURE_2D, n, textures );
....
glBindTexture ( GL_TEXTURE_2D, textures[k] );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, data );
```

Procedura `glGenTextures` rezerwuje n identyfikatorów tekstur i wpisuje je do tablicy podanej jako ostatni parametr; związane z tymi identyfikatorami obiekty tekstur są początkowo puste. Procedura `glBindTexture` przywiązuje obiekt tekstury do celu `GL_TEXTURE_2D` i kolejne instrukcje związane z teksturą działają na przywiązanym do tego celu obiekcie.

Parametry `GL_TEXTURE_WRAP_S` i `GL_TEXTURE_WRAP_T` określają, co się ma dziać, jeśli współrzędne s i t reprezentują punkt poza dziedziną tekstury, tj. poza kwadratem $[0,1]^2$. Jeśli, jak w przykładzie wyżej, mają one wartość `GL_REPEAT`, to odpowiednia współrzędna zostanie zastąpiona przez jej część ułamkową, co skutkuje okresowym powieleniem tekstury na całą płaszczyznę. Inne możliwe wartości to `GL_CLAMP_TO_EDGE` i `GL_CLAMP_TO_BORDER`, `GL_MIRRORED_REPEAT`, `GL_MIRRORED_CLAMP_TO_EDGE` i `GL_MIRRORED_CLAMP_TO_BORDER`. Pierwsza z nich powoduje zastąpienie „wystającej” współrzędnej przez bliższą z liczb 0 lub 1, czyli użycie wartości teksela z pierwszej lub ostatniej kolumny albo wiersza. W drugim przypadku tekstura przyjmuje wartość określoną osobno, przez wykonanie instrukcji

```
glTextureParameterfv ( GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, kolor );
```

gdzie `kolor` jest tablicą zawierającą współrzędne RGBA teksela. Wartości „`MIRRORED`” powodują, że następuje odbicie wartości tekstury na odcinku $[0,1]$ dzięki czemu powstaje funkcja parzysta na odcinku $[-1,1]$, a następnie tekstura jest rozszerzana okresowo, albo współrzędna poza przedziałem $[-1,1]$ jest zastępowana przez któryś koniec przedziału.

Procedura `glTexImage2D` przesyła do tekstury obraz o szerokości `w` i wysokości `h`;

powoduje to nadanie tablicy tekseli takich wymiarów. Drugi parametr procedury jest poziomem, na którym ma być umieszczony obraz, trzeci określa wewnętrzną reprezentację teksela. OpenGL oferuje ogromną liczbę możliwych wewnętrznych reprezentacji tekseli, które mogą być opisane za pomocą liczb stało- i zmiennopozycyjnych[5] i mogą mieć od jednej do czterech współrzędnych. Szósty parametr określa obecność brzegu (jednej wyróżnionej „warstwy" tekseli naokoło tablicy tekseli), przy czym w nowym OpenGL-u parametr ten musi mieć wartość `0`.

Po nadaniu zawartości tekselom na poziomie `0`, możemy wywołać procedurę `glTexImage2D` dla wszystkich poziomów, podając `NULL` jako ostatni parametr, a potem wywołać procedurę `glGenerateTextureMipmaps`, która utworzy reprezentacje tekstury o mniejszej rozdzielczości. Możemy też samemu utworzyć te reprezentacje w pamięci CPU, przy użyciu dowolnych algorytmów przetwarzania obrazów, które mogą dać lepsze (albo gorsze) efekty niż algorytm w procedurze `glGenerateTextureMipmaps`.

Parametrom określającym sposób filtrowania tekstury, tj. `GL_TEXTURE_MAG_FILTER` i `GL_TEXTURE_MIN_FILTER` możemy (za pomocą procedury `glTexParameteri`) nadać wartości oznaczone przez nazwy, w których słowo „`LINEAR`" zastąpimy przez „`NEAREST`". Wtedy zamiast dokonywać interpolację, funkcja `texture` poda wartość teksela najbliższego punktu o podanych współrzędnych tekstury. Zabiera to mniej czasu, ale daje prawie taki sam efekt jak całkowite wyłączenie antyaliasingu tekstury.

W pierwszym akapicie tego rozdziału napisałem o tym, że często występuje potrzeba używania wielu tekstur naraz. Robi się to tak: OpenGL udostępnia pewną liczbę[6] punktów dowiązania tekstur (*texture units*), przy czym każdy z nich udostępnia dowolny z celów (*targets*) nazwanych `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` i kilku innych. Procedury działające na teksturach (np. wprowadzające parametry filtrowania lub przesyłające obraz do tekseli w pamięci GPU) zakładają, że tekstura jest przywiązana do odpowiedniego celu w bieżąco aktywnym punkcie dowiązania tekstury. Na początku działania aplikacji aktywny jest punkt o numerze `0`, ale w każdej chwili można go zmienić:

[5] Reprezentacja obrazu za pomocą stałopozycyjnych liczb ośmiobitowych może być bezpośrednio wyświetlana na ekranie, ale jeśli obraz jest teksturą stanowiącą dane dla dalszych obliczeń numerycznych, to taka reprezentacja może być niewystarczająco dokładna. W takich sytuacjach mogą być używane liczby zmiennopozycyjne typu `float`, tj. 32-bitowe, liczby zmiennopozycyjne „połówkowej precyzji", zajmujące 16 bitów, lub nawet liczby jedenasto- i dziesięciobitowe.

[6] Co najmniej 48, liczbę dostępną w konkretnym systemie aplikacja może poznać, wywołując procedurę `glGetIntegerv` z parametrem `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

w tym celu wystarczy wykonać instrukcję

```
glActiveTexture ( GL_TEXTURE0 + i );
```

i odtąd aktywny jest punkt i (można też podać parametr o nazwie GL_TEXTURE0 ... GL_TEXTURE31, to są makra rozwijające się do stałych symbolicznych o kolejnych wartościach całkowitych).

Aby szader fragmentów miał dostęp do tekstury przywiązanej do i-tego punktu, trzeba w nim zadeklarować zmienną jednolitą typu sampler2D w taki sposób:

```
layout (binding=i) uniform sampler2D texi;
```

przy czym wyrażenie i musi być stałe (tj. możliwe do obliczenia podczas kompilacji szadera). Można też zadeklarować tablicę ewaluatorów tekstury, np. w taki sposób:

```
layout (binding=i) uniform sampler2D tex[N];
```

Wtedy kolejne elementy tablicy są skojarzone z kolejnymi punktami dowiązania tekstur, od punktu i-tego do punktu $i + N - 1$.

Przed rysowaniem obiektu aplikacja dla każdej potrzebnej tekstury powinna wywołać procedurę glActiveTexture z numerem odpowiedniego punktu dowiązania (dodanym do stałej GL_TEXTURE0), a następnie procedurę glBindTexture, podając cel (np. GL_TEXTURE_2D dla tekstur dwuwymiarowych) i identyfikator tekstury, która w czasie rysowania tego obiektu ma być dowiązana do uaktywnionego punktu.

Ewaluator tekstury (*sampler*), reprezentowany przez zmienną typu sampler2D (lub podobnego) jest obiektem przechowującym parametry opisujące sposób obliczania wartości tekstury przez funkcję texture. Każda tekstura ma wbudowany ewaluator, ale można utworzyć dodatkowy obiekt (obiekty) tego rodzaju, ustawić w nich i spowodować użycie go zamiast ewaluatora wbudowanego w teksturę. Służą do tego procedury glCreateSamplers, glBindSampler i rodzina procedur o nazwach glSamplerParameter z różnymi przedrostkami.

Aby włączyć antyaliasing w aplikacji FreeGLUTa, podczas inicjalizacji (przed utworzeniem okien) trzeba wykonać instrukcje

```
glutSetOption ( GLUT_MULTISAMPLE, n );
glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH |
                      GLUT_MULTISAMPLE );
```

18. Aplikacja druga E

Dodamy do sceny lustro, w którym czajnik z torusem będzie się odbijał. Możemy taki efekt osiągnąć na dwa sposoby. Pierwszy sposób polega na odbiciu sceny (tj. obu obiektów i wszystkich źródeł światła) w płaszczyźnie lustra i narysowaniu odbitej sceny w obszarze zajmowanym przez obraz lustra. Drugi sposób, zrealizowany w opisanej tu aplikacji, polega na odbiciu w płaszczyźnie lustra położenia obserwatora i narysowania obrazu sceny „widzianej zza lustra" (przy czym lustro i wszelkie przedmioty położone za nim pominiemy). Płaszczyzna lustra jest wtedy rzutnią, a samo lustro (lub prostokąt otaczający lustro, które może nie być prostokątne) jest klatką, w której określimy raster o odpowiedniej rozdzielczości. Otrzymany obraz rastrowy nałożymy jako teksturę na lustro podczas rysowania sceny widzianej z oryginalnego położenia obserwatora[1].

Realizacja opisanego sposobu wymaga użycia odrobiny algebry z geometrią oraz nowego elementu OpenGL-a: tworzenia obrazu poza oknem na ekranie. Nie życzymy sobie oglądać obrazu zza lustra inaczej niż jako odbicie w lustrze odpowiednich obiektów w gotowym obrazie całej sceny. Ale po kolei.

Algebra z geometrią

Dowolną płaszczyznę, w tym płaszczyznę lustra, możemy reprezentować za pomocą jej dowolnego punktu, $\mathbf{p}_0$, i wektora normalnego, $\mathbf{m}$ (ewentualnie jednostkowego, $\mathbf{n} = \pm\mathbf{m}/\|\mathbf{m}\|_2$). Obraz $\mathbf{e}'$ położenia obserwatora $\mathbf{e}$ w odbiciu symetrycznym względem tej płaszczyzny otrzymamy ze wzoru

$$\mathbf{e}' = \mathbf{e} - \frac{2\langle\mathbf{m}, \mathbf{e}-\mathbf{p}_0\rangle}{\langle\mathbf{m},\mathbf{m}\rangle}\mathbf{m} = \mathbf{e} - 2\langle\mathbf{n}, \mathbf{e}-\mathbf{p}_0\rangle\mathbf{n}.$$

Dla położonego w przestrzeni trójwymiarowej prostokąta[2] (lustra) o wierzchołkach $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ (podanych w układzie współrzędnych świata) obliczymy wektory

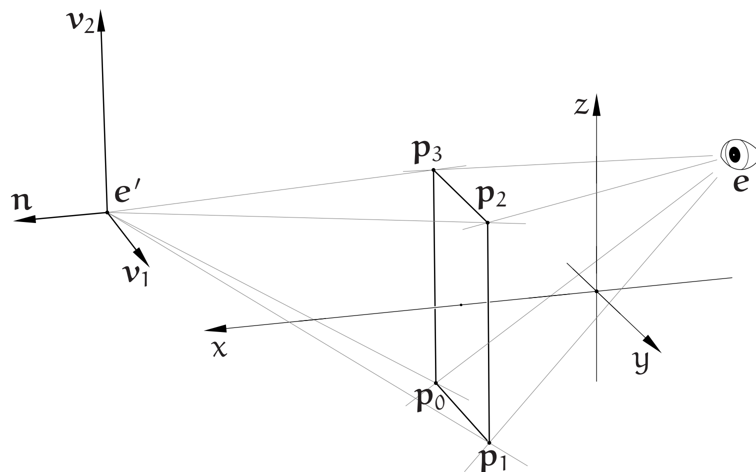
$$\mathbf{v}_1 = \mathbf{p}_1 - \mathbf{p}_0, \quad \mathbf{v}_2 = \mathbf{p}_3 - \mathbf{p}_0, \quad \mathbf{m} = \mathbf{v}_1 \wedge \mathbf{v}_2, \quad \mathbf{n} = \operatorname{sgn}\langle\mathbf{m}, \mathbf{p}_0 - \mathbf{e}\rangle\frac{\mathbf{m}}{\|\mathbf{m}\|_2}.$$

Punkt $\mathbf{e}'$ i wektory $\mathbf{v}_1, \mathbf{v}_2, \mathbf{n}$ stanowią układ odniesienia dla układu współrzędnych kartezjańskich w przestrzeni; będzie to <u>układ obserwatora odbitego w lustrze</u>; jego położenie, będące początkiem układu, to punkt $\mathbf{e}'$. Wektory $\mathbf{v}_1$ i $\mathbf{v}_2$ mają kierunki i długości boków lustra, a wektor jednostkowy $\mathbf{n}$ jest do nich (czyli do płaszczyzny lustra) prostopadły; w nowym układzie te trzy wektory są wersorami osi (rys. 18.1). Wybór zwrotu wektora $\mathbf{n}$ wynika z potrzeby otrzymania układu

---

[1] Zauważmy, że stosując każdy z tych sposobów, musimy scenę narysować dwukrotnie. Czyżby ktoś był zaskoczony?

[2] To może być równoległobok, gdyby ktoś tego potrzebował.



Rysunek 18.1: Położenie obserwatora, lustro i układ odbitego obserwatora

współrzędnych, w którym punkty płaszczyzny lustra mają współrzędną z ujemną. Macierz przejścia od nowego układu do układu świata jest taka[3]:

$$V^{-1} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \boldsymbol{n} & \boldsymbol{e}' \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{18.1}$$

Odwrotność tej macierzy, V, opisuje przejście od układu świata do układu odbitego obserwatora; wykorzystamy ją podczas rysowania obrazu na teksturze, która później będzie nałożona na lustro. Potrzebujemy jeszcze macierzy P opisującej przekształcenie bryły widoczności na kostkę standardową.

W układzie odbitego obserwatora punkty $\boldsymbol{p}_0$ i $\boldsymbol{p}_2$ mają odpowiednio współrzędne $(l, b, -n)$ i $(r, t, -n)$ (zobacz opis rzutowania perspektywicznego w rozdz. 6). Liczby l, r, b, t, n podamy jako parametry procedury `M4x4Frustumf`, która utworzy macierz P. Ponieważ wektor $\boldsymbol{n}$ jest jednostkowy, liczba n (`near`) jest odległością obserwatora (i jego obrazu odbitego w lustrze) od płaszczyzny lustra. Potrzebujemy jeszcze parametru f (`far`) określającego położenie tylnej ściany bryły widzenia. Sensownym wyborem wydaje się przyjęcie tego parametru o takiej samej wartości, jak podczas tworzenia macierzy dla końcowego obrazu. Jeszcze jedno: rysowane obiekty zostaną obcięte do kostki standardowej. To w szczególności oznacza, że gdyby obiekt wystawał poza płaszczyznę lustra, to jego wystająca część nie zostałaby narysowana. Właśnie tak powinno być.

Konsekwencją dokonanego wyboru wektorów $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ jest odwzorowanie prostokąta — lustra — na całą ścianę $[-1, 1] \times [-1, 1] \times \{-1\}$ kostki standardowej, dzięki czemu wykonany obraz będzie dokładnie odpowiadał obszarowi lustra.

[3]To jest macierz 4×4, mnożymy przez nią wektory współrzędnych jednorodnych.

Tworzenie obrazów poza oknem

Termin *framebuffer* jest na język polski powszechnie tłumaczony prawie dosłownie (i prawie sensownie) jako bufor ramki[4]; niech już będzie. Termin ten oznacza strukturę danych, w ramach której mogą być tworzone obrazy. Struktura taka jest tworzona przez system okien (z uwzględnieniem *jego* specyfiki) dla każdego okna, z którym OpenGL ma współdziałać. Aplikacja może utworzyć dodatkowe bufory ramki i wykorzystać je do tworzenia obrazów pomocniczych (na przykład obrazów sceny odbitej w lustrze) lub do tworzenia obrazów, które nie mają być natychmiast wyświetlane, tylko np. zapisywane w plikach jako klatki filmu animowanego (i mają na przykład znacznie większą rozdzielczość niż ekran monitora).

Sam bufor ramki po utworzeniu *nie jest* gotowy do pracy; trzeba w nim zainstalować załączniki (*attachments*), czyli konkretne bufory, w których powstaje obraz. Załącznik może być teksturą lub buforem roboczym (*renderbuffer*). Użycie tekstur jako załączników jest prostsze (w tej aplikacji skorzystamy z tego rozwiązania), zaś bufory robocze mają większe możliwości (na przykład umożliwiają antyaliasing przez wielokrotne próbkowanie). Aby utworzyć obraz, zawsze potrzebny jest załącznik będący buforem obrazu. Zazwyczaj potrzebny jest też załącznik używany jako bufor głębokości i rzadziej załącznik — bufor szablonu (który tu jest nam niepotrzebny).

Szadery

Wykorzystamy szadery z aplikacji drugiej D, z pewną zmianą wprowadzoną *dla wygody*[5]. Chodzi o przekształcanie wierzchołków do kostki standardowej. Przekształcenie to jest złożeniem przejścia od układu obiektu (modelu) do układu świata, przejścia od układu świata do układu obserwatora i przejścia od układu obserwatora do kostki standardowej. Wszystkie trzy przekształcenia są reprezentowane przez macierze 4×4. Opisane wcześniej aplikacje obliczają iloczyn macierzy i umieszczają go w zmiennej jednolitej `TransBlock.mvpm`. Zmiana dowolnego przekształcenia wymaga zatem ponownego obliczenia tego iloczynu i przesłania go oraz poszczególnych macierzy do bloku `TransBlock`. Ale teraz,

[4]Niech mi ktoś na przykład wytłumaczy, skąd i po co jest to zdrobnienie?

[5]Gdyby ktoś spytał, czy nie można było tak od razu, to bym odpowiedział, że w żadnej sprawie nie istnieje „ostateczne rozwiązanie" i nigdy nie należy uważać, że się je znalazło; zresztą w podróży więcej można zobaczyć wzdłuż drogi niż na jej końcu. Wraz z pojawianiem się kolejnych potrzeb jeszcze nieraz będziemy dokonywać przeróbek działającego kodu. Gdybym zaś w początkowych rozdziałach pokazał na listingach szadery z wszystkimi elementami, które kiedyś mogą się przydać, to byłoby to wrzucanie Czytelników na zbyt głęboką wodę, a poza tym to by nie były bardzo dobre szadery.

w związku z rysowaniem dwóch różnych rzutów sceny na każdym obrazie, będziemy znacznie częściej zmieniać macierze opisujące przejście od układu świata do kostki. Dlatego będziemy przechowywać osobno macierz przekształcenia modelu i iloczyn pozostałych dwóch macierzy — zmienimy nazwę pola `TransBlock.mvpm` na `TransBlock.vpm`. Przekształcając wierzchołek, szader wykona dwa mnożenia wektora przez macierz 4×4 zamiast jednego. Związany z tym dodatkowy koszt przetwarzania wierzchołków jest niezauważalny.

Szadery wierzchołków, sterowania rozdrabnianiem i geometrii pierwszego programu z aplikacji drugiej D są niezmienione. Na listingu 18.1 są pokazane zmienione dwie linie szadera rozdrabniania. W treści szadera fragmentów została zmieniona nazwa pola bloku `TransBlock` z `mvpm` na `vpm`, ale ponieważ szader do tego pola się nie odwołuje, cała reszta została taka, jak była.

Listing 18.1: Zmiany szadera rozdrabniania

GLSL

```
1: .... /* początek bez zmian */
2: uniform TransBlock {
3:     mat4 mm, mmti, vm, pm, vpm;
4:     vec4 eyepos;
5:   } trb;
6: .... /* dalszy ciąg bez zmian */
7: void main ( void )
8: {
9: ....
10:   gl_Position = trb.vpm * (trb.mm * pos);
11: ....
12: } /*main*/
```

Drugi program szaderów, używany do wyświetlania odcinków siatek kontrolnych płatów, składa się tylko z szadera wierzchołków i fragmentów. W pierwszym z nich zmienione są dwie linie — gdyby nie obecność linii 11 na listingu 18.1, to byłby on doskonałym zapisem tych zmian.

Aplikacja użyje jeszcze trzeciego programu szaderów do narysowania lustra z nałożoną teksturą. Również ten program składa się z dwóch szaderów, wierzchołków i fragmentów. Szadery te są pokazane na listingach 18.2 i 18.3.

Na początku szadera wierzchołków jest zadeklarowany blok zmiennych jednolitych `TransBlock`, identyczny, jak we wszystkich innych szaderach tej aplikacji, w których ten blok występuje. Oprócz struktury `gl_PerVertex` (z polem

Listing 18.2: Szader wierzchołków lustra

GLSL

```
1: #version 450 core
2:
3: layout(location=0) in vec4 in_Position;
4:
5: uniform TransBlock {
6:     mat4 mm, mmti, vm, pm, vpm;
7:     vec4 eyepos;
8:   } trb;
9:
10: out GVertex {
11:     vec2 TexCoord;
12:   } Out;
13:
14: void main ( void )
15: {
16:   switch ( gl_VertexID ) {
17: default: Out.TexCoord = vec2 ( 0.0, 0.0 );  break;
18:  case 1: Out.TexCoord = vec2 ( 1.0, 0.0 );  break;
19:  case 2: Out.TexCoord = vec2 ( 1.0, 1.0 );  break;
20:  case 3: Out.TexCoord = vec2 ( 0.0, 1.0 );  break;
21:   }
22:   gl_Position = trb.vpm * (trb.mm * in_Position);
23: } /*main*/
```

gl_Position, do którego szader przypisuje współrzędne wierzchołka w układzie kostki standardowej), wyjście szadera zawiera blok GVertex z polem TexCoord. Na podstawie numeru wierzchołka (te numery odpowiadają indeksom wierzchołków lustra, zobacz rys. 18.1) szader wierzchołków generuje współrzędne tekstury. Prostokąt lustra jest w układzie współrzędnych kwadratem jednostkowym, pokrywającym się z dziedziną tekstury, którą nałożymy na lustro.

Szader fragmentów na listingu 18.3 jest bardzo prosty, ponieważ jego jedynym zadaniem jest przekazanie na wyjście koloru pobranego z tekstury, bez żadnych zmian; wszelkie obliczenia oświetlenia zostały wykonane wcześniej, podczas tworzenia tej tekstury. Instrukcja warunkowa w liniach 13–16 dla lustra obróconego drugą stroną do obserwatora podaje na wyjście kolor ciemnoszary; „z tyłu” lustro jest zwykłym nieprzezroczystym prostokątem. Widzimy tu zastosowanie wbudowanej w GLSL zmiennej interfejsu gl_FrontFacing typu bool, która umożliwia szaderowi fragmentów rozróżnienie stron płaszczyzny rysowanego trójkąta.

Listing 18.3: Szader fragmentów lustra

GLSL
```
1: #version 450 core
2:
3: in GVertex {
4:     vec2 TexCoord;
5:   } In;
6:
7: out vec4 out_Colour;
8:
9: uniform sampler2D tex;
10:
11: void main ( void )
12: {
13:   if ( gl_FrontFacing )
14:     out_Colour = vec4(0.3);
15:   else
16:     out_Colour = texture ( tex, In.TexCoord );
17: } /*main*/
```

Utensylia

Na listingu 18.4 jest przedstawiona procedura pomocnicza `V3ReflectPointf`, która dokonuje odbicia punktu $\mathbf{q}$ względem płaszczyzny danej za pomocą punktu $\mathbf{p}_0$ i wektora normalnego $\mathbf{n}$; współrzędne otrzymanego punktu są wpisywane do tablicy `r`.

Listing 18.4: Procedury pomocnicze

C
```
1: void V3ReflectPointf ( GLfloat r[3], const GLfloat p[3],
2:                        const GLfloat nv[3], const GLfloat q[3] )
3: {
4:   GLfloat g;
5:
6:   r[0] = q[0] - p[0];     r[1] = q[1] - p[1];     r[2] = q[2] - p[2];
7:   g = 2.0*V3DotProductf ( nv, r )/V3DotProductf ( nv, nv );
8:   r[0] = q[0] - g*nv[0];  r[1] = q[1] - g*nv[1];  r[2] = q[2] - g*nv[2];
9: } /*V3ReflectPointf*/
```

Procedura `M4x4MultMP3f` oblicza pierwsze trzy współrzędne iloczynu $A\mathbf{p}$ macierzy A o wymiarach 4×4 i wektora $\mathbf{p} \in \mathbb{R}^4$, którego pierwsze trzy współrzędne są dane jako parametr, a ostatnia jest równa 1. Otrzymujemy w ten

sposób współrzędne kartezjańskie obrazu punktu w przekształceniu afinicznym reprezentowanym przez macierz A.

Procedury obsługi lustra

Przygotowanie lustra obejmuje utworzenie odpowiedniego obiektu geometrycznego (prostokąta w przestrzeni), tj. VAO z wierzchołkami lustra i bufora ramki z załącznikami — teksturami wykorzystywanymi do utworzenia obrazu widzianego w lustrze. Odpowiednie procedury, które trzeba wywołać na początku działania aplikacji, są pokazane na listingu 18.5. W liniach 1–3 są deklaracje zmiennych związanych z lustrem, tj. identyfikatorów obiektów w pamięci GPU i tablicy przechowującej macierz przekształcenia lustra. Macierz ta jest jednostkowa, co oznacza, że współrzędne wierzchołków $\mathbf{p}_0, \ldots, \mathbf{p}_3$ w tablicy mvertpos są podane w układzie współrzędnych świata.

Listing 18.5: Procedury tworzenia lustra

C

```
1: GLuint  mirror_fbo, mirror_txt[2];
2: GLuint  mirror_vao, mirror_vbo;
3: GLfloat mirror_matrix[16];
4:
5: static const GLfloat mvertpos[4][3] =
6:   {{1.5,-1.2,-1.0},{1.5,1.2,-1.0},{1.5,1.2,1.0},{1.5,-1.2,1.0}};
7:
8: void ConstructMirrorFBO ( void )
9: {
10:   GLenum status;
11:
12:   glGenTextures ( 2, mirror_txt );
13:   glBindTexture ( GL_TEXTURE_2D, mirror_txt[0] );
14:   glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
15:   glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
16:   glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, MIRRORTXT_W, MIRRORTXT_H,
17:                  0, GL_RGB, GL_UNSIGNED_BYTE, NULL );
18:   glBindTexture ( GL_TEXTURE_2D, mirror_txt[1] );
19:   glTexImage2D ( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, MIRRORTXT_W,
20:                  MIRRORTXT_H, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL );
21:   glGenFramebuffers ( 1, &mirror_fbo );
22:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, mirror_fbo );
23:   glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
24:                          mirror_txt[0], 0 );
25:   glFramebufferTexture ( GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
26:                          mirror_txt[1], 0 );
```

```
27:   if ( (status = glCheckFramebufferStatus ( GL_DRAW_FRAMEBUFFER )) !=
28:         GL_FRAMEBUFFER_COMPLETE )
29:       ExitOnError ( "ConstructMirrorFBO" );
30:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
31:   glBindTexture ( GL_TEXTURE_2D, 0 );
32:   ExitIfGLError ( "ConstructMirrorFBO" );
33: } /*ConstructMirrorFBO*/
34:
35: void ConstructMirrorVAO ( void )
36: {
37:   glGenVertexArrays ( 1, &mirror_vao );
38:   glBindVertexArray ( mirror_vao );
39:   glGenBuffers ( 1, &mirror_vbo );
40:   glBindBuffer ( GL_ARRAY_BUFFER, mirror_vbo );
41:   glBufferData ( GL_ARRAY_BUFFER,
42:                  4*3*sizeof(GLfloat), mvertpos, GL_STATIC_DRAW );
43:   glEnableVertexAttribArray ( 0 );
44:   glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE,
45:                           3*sizeof(GLfloat), (GLvoid*)0 );
46:   M4x4Identf ( mirror_matrix );
47:   ExitIfGLError ( "ConstructMirrorVAO" );
48: } /*ConstructMirrorVAO*/
```

Procedura `ConstructMirrorFBO` tworzy obiekt bufora ramki (*framebuffer object*, *FBO*) z załącznikami. Załącznikami są dwie tekstury, których identyfikatory rezerwuje procedura `glGenTextures` w linii 12. W liniach 13–17 nadawane są własności pierwszej z nich. Ma to być tekstura dwuwymiarowa, której sposób filtracji podczas zmniejszania i powiększania tekseli polega na interpolacji liniowej. W liniach 16–17 teksturze jest nadawana (przez procedurę `glTexImage2D`) wielkość, tj. szerokość i wysokość w tekselach oraz wewnętrzny format tekseli (RGB — nie potrzebujemy kanału alfa, więc nie zużywany na niego miejsca w pamięci GPU). Ostatni parametr jest wskaźnikiem tablicy z danymi — podajemy wskaźnik pusty, ponieważ nie mamy tu żadnej treści do wpisania do tablicy tekseli tekstury.

Druga tekstura, formowana w liniach 18–20, będzie używana jako bufor głębokości podczas tworzenia obrazu sceny odbitej w lustrze. Nie ma potrzeby określać dla niej sposobu filtracji. Parametry wywołania procedury `glTexImage2D` w liniach 19–20 powodują zarezerwowanie odpowiedniego miejsca dla bufora głębokości; jej teksele są liczbami zmiennopozycyjnymi, przy czym znów przekazujemy pusty adres tablicy z danymi, bo nie w tym momencie ma miejsce wpisywanie czegokolwiek do tekseli[6].

[6] I w ogóle aplikacja nie będzie przesyłać żadnych danych z pamięci CPU do tych tekstur

W linii 21 następuje wywołanie procedury glGenFramebuffers, która rezerwuje (jeden) identyfikator nowego bufora ramki, następnie przywiązywanego w linii 22 do celu zwanego GL_DRAW_FRAMEBUFFER[7].

W liniach 23–26 procedura glFramebufferTexture instaluje utworzone wcześniej tekstury jako załączniki bufora ramki. Pierwszy parametr identyfikuje cel, do którego został przywiązany bufor ramki, drugi opisuje rodzaj załącznika. Bufor ramki może mieć kilka załączników, w których znajduje się obraz (tj. piksele ze współrzędnymi koloru). W naszym przypadku używamy jednego takiego załącznika, który rejestrujemy jako załącznik GL_COLOR_ATTACHMENT0 (drugi załącznik, zarejestrowany jako GL_DEPTH_ATTACHMENT jest używany jako bufor głębokości). Trzeci parametr jest identyfikatorem tekstury, a czwarty jest numerem poziomu w tej teksturze. Obie tekstury zostały utworzone jako jednopoziomowe, zatem mają tylko poziom 0.

W linii 27 jest wywołana procedura glCheckFramebufferStatus, która sprawdza, czy bufor ramki ma wszystkie załączniki konieczne do poprawnej pracy. Wartość zwracana przez tę procedurę wskazuje, że wszystko jest o.k., albo podaje informację o wykrytym błędzie. W opisanej tu aplikacji w razie błędu następuje rezygnacja z dalszego działania.

Uwaga: Fakt, że na jednym komputerze został utworzony poprawny i kompletny bufor ramki *nie jest* gwarancją sukcesu na innym komputerze — może na nim zabraknąć pamięci GPU lub mogą wystąpić inne, lokalne przyczyny niepowodzenia.

W liniach 30 i 31 „odczepiamy” od punktów dowiązania bufor ramki i teksturę, przywracając stan domyślny, w którym rysowanie będzie miało miejsce w buforze ramki *okna*, a właściwą teksturę do nałożenia na obiekt, gdy będzie potrzebna, będzie się przywiązywać przed rysowaniem tego obiektu.

Procedura ConstructMirrorVAO wykonuje zadanie dobrze znane Czytelnikowi pierwszej aplikacji; tworzony jest obiekt tablicy wierzchołków, a następnie jest w tym obiekcie instalowany bufor ze współrzędnymi wierzchołków (nie ma innych atrybutów, np. koloru). Ponadto do tablicy mirror_matrix są wpisywane współczynniki macierzy jednostkowej, która opisuje przejście od układu modelu (lustra) do układu świata.

---

w pamięci GPU — tylko GPU będzie coś do tych tekstur pisać i je czytać.

[7] Istnieje też cel GL_READ_FRAMEBUFFER, który przydaje się w sytuacji, gdy trzeba przepisać dane z jednego bufora ramki do drugiego — każdy z buforów należy przywiązać do odpowiedniego celu, a następnie wywołać np. procedurę glBlitFramebuffer.

Na listingu 18.6 jest pokazana procedura SetupMirrorVPMatrices, której zadaniem jest obliczenie położenia odbitego w lustrze obserwatora oraz macierzy przejścia od układu współrzędnych świata do układu odbitego obserwatora i od tego ostatniego układu do kostki standardowej. Procedura ta wykonuje tylko obliczenie współczynników tych macierzy, bez przesyłania czegokolwiek do pamięci GPU. Parametrem wejściowym jest tablica eyepos ze współrzędnymi położenia obserwatora (w układzie świata). W tablicy reyepos procedura ma umieścić współrzędne położenia odbitego obserwatora, a w tablicach mvm i mpm współczynniki wspomnianych macierzy przekształceń.

Listing 18.6: Tworzenie macierzy przekształceń dla lustra

C

```
1: void SetupMirrorVPMatrices ( GLfloat eyepos[4], GLfloat reyepos[4],
2:                              GLfloat mvm[16], GLfloat mpm[16] )
3: {
4:   GLfloat mfm[16], *v1, *v2, *nv, *p;
5:   GLfloat s, a[3], b[3];
6:   int     i;
7:
8:   v1 = &mfm[0], v2 = &mfm[4], nv = &mfm[8], p = &mfm[12];
9:   for ( i = 0; i < 3; i++ ) {
10:    v1[i] = mvertpos[1][i]-mvertpos[0][i];
11:    v2[i] = mvertpos[3][i]-mvertpos[0][i];
12:    a[i] = mvertpos[0][i]-eyepos[i];
13:  }
14:  V3CrossProductf ( nv, v2, v1 );
15:  s = sqrt ( V3DotProductf ( nv, nv ) );
16:  if ( V3DotProductf ( nv, a ) < 0.0 )
17:    s = -s;
18:  nv[0] /= s;  nv[1] /= s;  nv[2] /= s;
19:  V3ReflectPointf ( reyepos, mvertpos[0], nv, eyepos );
20:  memcpy ( p, reyepos, 3*sizeof(GLfloat) );
21:  mfm[3] = mfm[7] = mfm[11] = 0.0;  mfm[15] = 1.0;
22:  M4x4Invertf ( mvm, mfm );
23:  M4x4MultMP3f ( a, mvm, mvertpos[0] );
24:  M4x4MultMP3f ( b, mvm, mvertpos[2] );
25:  M4x4Frustumf ( mpm, NULL, a[0], b[0], a[1], b[1], -a[2], 20.0 );
26: } /*SetupMirrorVPMatrices*/
```

Ponieważ macierze są przechowywane w porządku *kolumnowym*, można zrobić to, co zostało zrobione w linii 8: przypisać dodatkowym wskaźnikom adresy początków poszczególnych kolumn roboczej tablicy mfm, w której utworzymy macierz V^{-1} daną wzorem (18.1). W pętli w liniach 9–13 są obliczane współrzędne

wektorów $\boldsymbol{v}_1$ i $\boldsymbol{v}_2$ oraz współrzędne wektora $\mathbf{p}_0 - \boldsymbol{e}$ potrzebnego do ustalenia właściwego zwrotu wektora $\boldsymbol{n}$. Wektor ten jest obliczany w liniach 14–18. W linii 19 następuje obliczenie współrzędnych położenia obserwatora, które następnie jest kopiowane do tablicy mfm jako początek ostatniej kolumny macierzy V^{-1}. W linii 21 tworzony jest ostatni wiersz tej macierzy, po czym następuje wywołanie procedury M4x4Invertf, czego skutkiem jest obliczenie macierzy V opisującej przejście od układu świata do układu odbitego obserwatora. W liniach 23, 24 obliczamy współrzędne wierzchołków lustra $\mathbf{p}_0$ i $\mathbf{p}_2$ w układzie odbitego obserwatora, aby następnie podstawić je jako parametry wywołania procedury glFrustumf, która konstruuje macierz przekształcenia do układu kostki standardowej.

Procedura rysowania lustra pokazana na listingu 18.7 jest bardzo prosta, ponieważ wszelkie skomplikowane przygotowania do rysowania lustra są wykonywane przed jej wywołaniem. Należy wybrać właściwy program szaderów (złożony z szaderów na listingach 18.2 i 18.3, jego identyfikator jest przechowywany w zmiennej program_id[2]), przesłać do bufora w pamięci GPU macierz przekształcenia lustra (która jest jednostkowa), uczynić bieżącym VAO z wierzchołkami lustra, przywiązać do celu GL_TEXTURE_2D teksturę z widokiem sceny w lustrze i narysować prostokąt (jako wachlarz złożony z dwóch trójkątów).

Listing 18.7: Rysowanie lustra

C
```c
1: void DrawMirror ( void )
2: {
3:   glUseProgram ( program_id[2] );
4:   SetModelMatrix ( mirror_matrix, mirror_matrix );
5:   glBindVertexArray ( mirror_vao );
6:   glBindTexture ( GL_TEXTURE_2D, mirror_txt[0] );
7:   glDrawArrays ( GL_TRIANGLE_FAN, 0, 4 );
8: } /*DrawMirror*/
```

Na zakończenie działania aplikacji, jak zwykle, sprzątamy, czyli w dodatku do dotychczasowych czynności likwidujemy bufor ramki i jego załączniki, a także VAO i VBO z wierzchołkami lustra. Wywołania procedur pokazanych na listingu 18.8 trzeba dopisać do procedury Cleanup.

Listing 18.8: Sprzątanie kawałków lustra

C

```
1: void DestroyMirrorFBO ( void )
2: {
3:   glDeleteFramebuffers ( 1, &mirror_fbo );
4:   glDeleteTextures ( 2, mirror_txt );
5:   ExitIfGLError ( "DestroyMirrorFBO" );
6: } /*DestroyMirrorFBO*/
7:
8: void DestroyMirrorVAO ( void )
9: {
10:   glDeleteBuffers ( 1, &mirror_vbo );
11:   glDeleteVertexArrays ( 1, &mirror_vao );
12:   ExitIfGLError ( "DestroyMirrorVAO" );
13: } /*DestroyMirrorVAO*/
```

Zmiany w aplikacji

Zmiany, których trzeba dokonać, aby z aplikacji drugiej D powstała aplikacja druga E, nie są wielkie, ale są dosyć liczne. Do procedury inicjalizacji trzeba dodać instrukcje, które wywołują procedury z listingu 18.5.

Listing 18.9: Zmiany w procedurze inicjalizacji

C

```
1: void InitMyObject ( void )
2: {
3: .... /* początek taki jak był */
4:   SetBezPatchOptions ( BezNormals, TessLevel );
5:   ConstructMirrorFBO ();
6:   ConstructMirrorVAO ();
7:   InitLights ();
8: ... /* koniec też bez zmian */
9: } /*InitMyObject*/
```

Z procedur konstruujących macierze przekształceń z układu świata do układu obserwatora i z układu obserwatora do układu kostki standardowej trzeba usunąć wszelkie instrukcje przesyłające te macierze do bufora z blokiem zmiennych jednolitych `TransBlock`. Wykonanie każdego obrazu w oknie wymaga dwukrotnego narysowania sceny, za każdym razem w innym rzucie, dlatego macierze te będziemy wysyłać na miejsce bezpośrednio przed rysowaniem sceny.

Procedura `LoadMyShaders` tworzy, kompiluje i łączy w program szadery

z listingów 18.2 i 18.3 i zapamiętuje ich identyfikatory w odpowiednio wydłużonych tablicach `shader_id` i `program_id`. Ponadto, za pomocą procedury `AttachUniformBlockToBP`, wiąże blok `TransBlock` zmiennych jednolitych tego programu z odpowiednim punktem dowiązania.

Z procedury `ReshapeFunc` trzeba usunąć wywołania procedur `glBindBuffer` i `glBufferSubData` (a także `ExitIfGLError`, jako że teraz procedura `ReshapeFunc` nie wywołuje żadnej procedury, która mogłaby spowodować błąd OpenGL-a).

Strukturę `TransBl` (która jest typem zmiennej globalnej `trans`) zdefiniowaną na listingu 10.6 zmienimy, dodając pola wyróżnione na listingu 18.10. W polach `mm` i `mmti` będzie przechowywana macierz przekształcenia modelu i transpozycja jej odwrotności. W polach `wvm` i `wpm` będą zapamiętane macierze przejścia do układu obserwatora i kostki dla rzutu podczas rysowania sceny w oknie, a pola `mvm` i `mpm` są przeznaczone dla odpowiednich macierzy do rysowania na teksturze lustra.

Listing 18.10: Zmieniona struktura `TransBl`

C

```c
1: typedef struct TransBl {
2:           GLfloat mm[16], mmti[16],
3:                   wvm[16], wpm[16],
4:                   mvm[16], mpm[16];
5:           GLfloat eyepos[4], reyepos[4];
6:         } TransBl;
```

W polu `eyepos` jest przechowywane położenie obserwatora, a w polu `reyepos` położenie obserwatora odbitego w lustrze.

Procedurze `SetMVPMatrix` zmieniłem nazwę na `SetupVPMatrix`, a jej treść pokazałem na listingu 18.11. Procedura ta, zależnie od wartości parametru, przesyła do bloku `TransBl` macierze przejścia i położenie obserwatora odpowiednie dla rzutowania sceny na ekran i rzutowania na lustro. Macierze przekształcenia modelu są przesyłane osobno, przed rysowaniem każdego obiektu.

Listing 18.11: Procedura `SetupVPMatrix`

C

```c
1: void SetupVPMatrix ( char mirror )
2: {
3:   GLfloat vpm[16], *vm, *pm, *ep;
4:
5:   vm = mirror ? trans.mvm : trans.wvm;
```

```
6:   pm = mirror ? trans.mpm : trans.wpm;
7:   ep = mirror ? trans.reyepos : trans.eyepos;
8:   M4x4Multf ( vpm, pm, vm );
9:   glBindBuffer ( GL_UNIFORM_BUFFER, trbuf );
10:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[2], 16*sizeof(GLfloat), vm );
11:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[3], 16*sizeof(GLfloat), pm );
12:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[4], 16*sizeof(GLfloat), vpm );
13:  glBufferSubData ( GL_UNIFORM_BUFFER, trbofs[5], 4*sizeof(GLfloat), ep );
14:  ExitIfGLError ( "SetupVPMatrix" );
15: } /*SetupVPMatrix*/
```

Na listingu 18.12 są pokazane zmienione procedury inicjalizacji położenia obserwatora i macierzy przejścia do układu obserwatora i zmiany położenia obserwatora. Zmiany polegają na usunięciu wszystkich instrukcji przesyłających macierze do zmiennych jednolitych w pamięci GPU (ten obowiązek przejęła opisana wyżej procedura `SetupVPMatrix`), dostosowaniu instrukcji do nowych nazw pól zmiennej strukturalnej `trans` i na dodaniu instrukcji wywołania procedury `SetupMirrorVPMatrices`, która oblicza położenie odbitego obserwatora i macierze potrzebne do rzutowania sceny na obszar lustra.

Listing 18.12: Procedury tworzenia macierzy przejścia do układu obserwatora

```c
1: void InitViewMatrix ( void )
2: {
3:   memcpy ( trans.eyepos, viewer_pos0, 4*sizeof(GLfloat) );
4:   M4x4Translatef ( trans.wvm,
5:                    -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
6:   SetupMirrorVPMatrices ( trans.eyepos,
7:                           trans.reyepos, trans.mvm, trans.mpm );
8: } /*InitViewMatrix*/
9:
10: void RotateViewer ( double delta_xi, double delta_eta )
11: {
12:   .... /* początek procedury bez zmian */
13:   M4x4Translatef ( tm, -viewer_pos0[0], -viewer_pos0[1], -viewer_pos0[2] );
14:   M4x4RotateVf ( rm, viewer_rvec[0], viewer_rvec[1], viewer_rvec[2],
15:                  -viewer_rangle );
16:   M4x4Multf ( trans.wvm, tm, rm );
17:   M4x4Transposef ( tm, rm );
18:   M4x4MultMVf ( trans.eyepos, tm, viewer_pos0 );
19:   SetupMirrorVPMatrices ( trans.eyepos,
20:                           trans.reyepos, trans.mvm, trans.mpm );
21: } /*RotateViewer*/
```

Na listingu 18.13 mamy procedurę Redraw, wywoływaną przez procedurę MessageLoop. Dla uczytelnienia kodu ciąg instrukcji powodujących rysowanie poszczególnych obiektów w scenie został przeniesiony do nowej procedury DrawSceneToWindow.

Listing 18.13: Procedura Redraw

```c
void Redraw ( GLFWwindow *win )
{
  DrawSceneToWindow ();
  glUseProgram ( 0 );
  glFlush ();
  glfwSwapBuffers ( win );
  ExitIfGLError ( "Redraw" );
  redraw = false;
} /*Redraw*/
```

Na listingu 18.14 jest jedna procedura rysująca obiekty sceny (bez lustra) i dwie procedury pomocnicze, w których wywołanie procedury rysowania sceny jest poprzedzone czynnościami wstępnymi potrzebnymi odpowiednio dla rysowania odbicia sceny w lustrze i obrazu do wyświetlenia w oknie.

Listing 18.14: Procedury rysowania sceny

```c
void DrawScene ( void )
{
  SetModelMatrix ( teapot_mmatrix, teapot_mmti );
  DrawMyTeapot ();
  if ( cnet )
    DrawMyTeapotCNet ();
  SetModelMatrix ( torus_mmatrix, torus_mmti );
  DrawMyTorus ();
  if ( cnet )
    DrawMyTorusCNet ();
} /*DrawScene*/

void DrawSceneToMirror ( void )
{
  glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, mirror_fbo );
  glViewport ( 0, 0, MIRRORTXT_W, MIRRORTXT_H );
  SetupVPMatrix ( 1 );
  glClearColor ( 0.95, 0.95, 0.95, 1.0 );
  glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

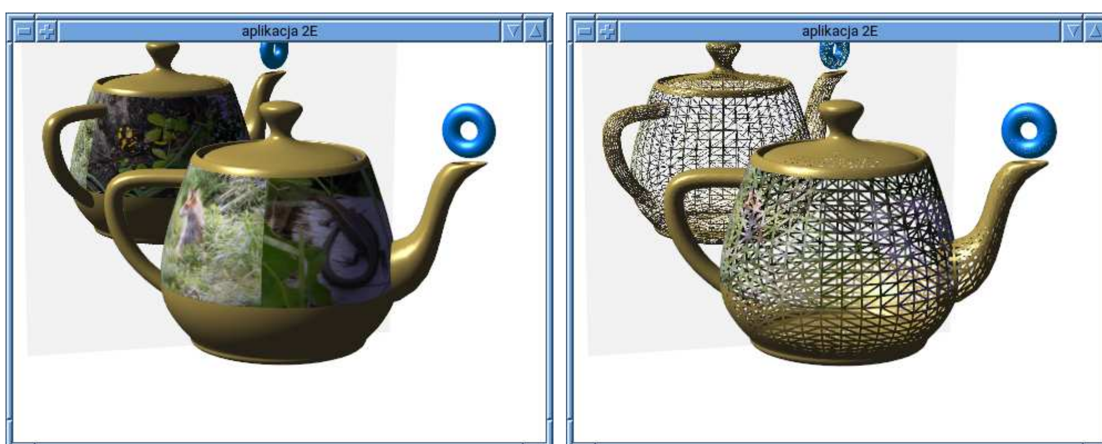
```
20:   DrawScene ();
21:   glFlush ();
22: } /*DrawSceneToMirror*/
23:
24: void DrawSceneToWindow ( void )
25: {
26:   glEnable ( GL_DEPTH_TEST );
27:   DrawSceneToMirror ();
28:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
29:   glViewport ( 0, 0, win_width, win_height );
30:   SetupVPMatrix ( 0 );
31:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
32:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
33:   glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
34:   DrawMirror ();
35:   DrawScene ();
36: } /*DrawSceneToWindow*/
```

Procedura `DrawScene` kolejno przesyła do bloku zmiennych jednolitych `TransBlock` macierz przekształcenia modelu czajnika, następnie wywołuje procedurę rysowania czajnika i jeśli `(cnet != 0)`, to również procedurę rysowania siatek kontrolnych modelu czajnika. Następnie analogiczne działania są wykonywane dla torusa.

Zadaniem procedury `DrawSceneToMirror` jest utworzenie obrazu czajnika i torusa widzianych przez odbitego obserwatora, na teksturze będącej załącznikiem utworzonego na początku działania aplikacji dodatkowego bufora ramki. Wywołanie procedury `glBindFramebuffer` w linii 15 powoduje, że odtąd do odwołania rysowanie będzie się odbywać w załącznikach dodatkowego bufora ramki, a nie domyślnego bufora związanego z oknem aplikacji. Parametry procedury `glViewport` w linii 16 określają, że klatka ma mieć wielkość całej tekstury (takie wymiary nadaliśmy teksturze, tworząc ją za pomocą procedury pokazanej na listingu 18.5). W linii 17 wywołujemy procedurę `SetupVPMatrix`, aby przypisać zmiennym jednolitym macierze przekształceń utworzone dla lustra. W liniach 18–19 kasujemy obie tekstury, tę z kolorami pikseli i tę pełniącą obowiązki bufora głębokości. Parametry wywołania procedury `glClearColor` opisują kolor o 5% ciemniejszy niż najjaśniejszy biały, aby na gotowym obrazie było widać, że mamy tam lustro. Po tych przygotowaniach możemy narysować scenę, co niezwłocznie czynimy.

Procedura `DrawSceneToWindow` wywołuje procedurę `DrawSceneToMirror`, aby otrzymać teksturę do nałożenia na lustro. Następnie (w linii 28) wywołuje

procedurę `glBindFramebuffer` z drugim parametrem `0`, aby przywrócić rysowanie w domyślnym buforze ramki związanym z oknem. Procedura `glViewport` ustawia wymiary klatki na całe okno, zaś procedura `SetupVPMatrix` przesyła do UBO z blokiem `TransBlock` macierze przekształceń dla finalnego rysunku. Następnie obraz i bufor głębokości są kasowane. Wywołanie procedury `glPolygonMode` w linii 33 ma na celu zapewnienie, że lustro na obrazie zostanie wypełnione (a nie będą narysowane tylko krawędzie lustra, jeśli czajnik i torus są rysowane jako siatki odcinków). Na końcu rysujemy lustro i pozostałe obiekty sceny.



Rysunek 18.2: Okno aplikacji drugiej E

Ćwiczenia

1. Wykonaj eksperymenty z lustrem „odwracającym”, tj. zamieniającym prawą stronę z lewą[8]. W tym celu zmodyfikuj szader wierzchołków na listingu 18.2, zmieniając współrzędne tekstury przyporządkowane poszczególnym wierzchołkom.

2. Rysowanie rzutu sceny na teksturze jest nieoszczędne, ponieważ ustawiamy klatkę o pełnej rozdzielczości tekstury, 1024×1024, nawet jeśli klatka z gotowym obrazem w oknie ma znacznie mniejsze wymiary. Zabiera to sporo czasu, a ponadto odcinki widoczne w lustrze na prawym rysunku 18.2 są znacznie cieńsze niż odcinki bezpośredniego obrazu czajnika. Zmodyfikuj aplikację (szadery i kod w C) tak, aby ustawiać wielkość klatki odpowiednio do wielkości obrazu w lustrze. Wymaga to m.in. zmienienia współrzędnych w układzie tekstury przyporządkowanych wierzchołkom lustra. Jeśli na przykład klatka miałaby wielkość połowy tekstury, to przedział zmienności współrzędnych tekstury powinien mieć długość $\frac{1}{2}$.

[8] To nieprawda, że zwykłe lustro zamienia stronę prawą z lewą. Lustro, jeśli można tak powiedzieć, „zamienia przód z tyłem”.

3. Spróbuj uruchomić antyaliasing podczas nakładania tekstury na lustro. W tym celu utwórz (dla obrazu, nie dla bufora głębokości) teksturę wielopoziomową, a następnie, po utworzeniu obrazu na poziomie `0` tekstury (na końcu procedury `DrawSceneToMirror`) wywołaj procedurę `glGenerateTextureMipmap`.

4. Zmień szadery programu do rysowania lustra tak, aby tylna strona lustra była „wykonana z określonego materiału” i oświetlona z użyciem odpowiedniego modelu oświetlenia.

5. Oprogramuj poruszanie lustrem (przez zmienianie macierzy modelu w zmiennej `mirror_matrix`); pamiętaj o uwzględnieniu przekształcenia lustra w obliczaniu macierzy przekształceń dla rysowania na teksturze lustra.

19. Aplikacja druga F

Nałożymy na obiekty proceduralną teksturę odkształceń; zaburzanie wektora normalnego powierzchni przekazywanego do obliczeń z ustalonym modelem oświetlenia umożliwia otrzymanie obrazu powierzchni chropowatej, porysowanej, pofalowanej albo np. pokrytej łuskami lub dachówkami. Osiągnięcie takich efektów przy użyciu dokładnego opisu kształtu (przez wygenerowanie tablicy z wierzchołkami wszystkich wypukłości i wnęk powierzchni) byłoby skrajnie niepraktyczne. Studiowanie lepszej metody rozwiązania problemu zaczniemy od zaaplikowania sobie homeopatycznej dawki geometrii różniczkowej.

Wektor normalny zaburzonej powierzchni

Rozważamy płat powierzchni parametrycznej, o parametryzacji $\mathbf{p}\colon A \to \mathbb{R}^3$ (w naszej aplikacji będzie to każdy z płatów Béziera, jego dziedzina A jest kwadratem jednostkowym $[0,1]^2$). Odwzorowanie Gaussa[1] jest to funkcja wektorowa, która każdemu punktowi $(u,v) \in A$ przyporządkowuje jednostkowy wektor normalny płata $\mathbf{p}$ w punkcie $\mathbf{p}(u,v)$. Jest ono dane wzorem

$$\mathbf{n}(u,v) = \frac{\mathbf{m}(u,v)}{\|\mathbf{m}(u,v)\|_2}, \quad \mathbf{m}(u,v) = \mathbf{p}_u(u,v) \wedge \mathbf{p}_v(u,v),$$

przy czym zakładamy, że wektory pochodnych cząstkowych $\mathbf{p}_u(u,v)$ i $\mathbf{p}_v(u,v)$ są liniowo niezależne. Wprowadzimy teraz przekształcenie $\mathbf{q}$ dziedziny A w pewien obszar w $\mathbb{R}^2$ i określoną w tym obszarze funkcję d, która przyjmuje wartości rzeczywiste o małych wartościach bezwzględnych. Zakładamy, że przekształcenie $\mathbf{q}$ i funkcja d są ciągłe i mają (kawałkami) ciągłe pochodne.

Określimy teraz parametryzację *nowej powierzchni*, wzorem

$$\hat{\mathbf{p}}(u,v) = \mathbf{p}(u,v) + d\big(\mathbf{q}(u,v)\big)\mathbf{n}(u,v),$$

który możemy zapisać krócej w postaci $\hat{\mathbf{p}} = \mathbf{p} + (d \circ \mathbf{q})\mathbf{n}$. Nowa powierzchnia powstaje z powierzchni oryginalnej przez przesunięcie każdego punktu w kierunku wektora normalnego, przy czym wielkość tego przesunięcia jest wartością funkcji d. Użycie tekstury odkształceń polega na narysowaniu oryginalnej powierzchni, ale do modelu oświetlenia podstawia się wektor normalny $\hat{\mathbf{n}}$ płata $\hat{\mathbf{p}}$. Jeśli przemieszczenia (czyli wartości funkcji d) są małe (w porównaniu z rozmiarami powierzchni), to otrzymany obraz bardzo trudno jest odróżnić od obrazu powierzchni $\hat{\mathbf{p}}$.

[1] Nie ma się czego bać: już od dawna korzystamy z tego odwzorowania, rysując oświetlony czajnik.

Znajdziemy wzory umożliwiające obliczenie wektora normalnego $\hat{\mathbf{n}}$. W tym celu potrzebujemy pochodnych cząstkowych parametryzacji $\hat{\mathbf{p}}$, które są kolumnami jej macierzy różniczki $D\hat{\mathbf{p}}$. Reguły obliczania pochodnych sumy, iloczynu i złożenia funkcji daje się wygodnie zapisać w postaci macierzowej, w której mamy

$$D\hat{\mathbf{p}} = D\mathbf{p} + D(d \circ \mathbf{q}) \cdot \mathbf{n} + (d \circ \mathbf{q})D\mathbf{n} \approx D\mathbf{p} + (Dd \cdot D\mathbf{q})\mathbf{n}.$$

W ostatnim kroku powyższego rachunku pominęliśmy składnik $(d \circ \mathbf{q})D\mathbf{n}$, ponieważ jest w nim czynnik d; zakładamy, że funkcja d ma małe wartości bezwzględne, dzięki czemu ten składnik istotnie jest pomijalny[2]. Do końcowego wzoru potrzebujemy zatem podstawić macierz $D\mathbf{p}$, której kolumny są pochodnymi cząstkowymi parametryzacji $\mathbf{p}$, macierz (wierszową) Dd, która jest gradientem funkcji d, macierz różniczki $D\mathbf{q}$ przekształcenia $\mathbf{q}$ i wektor normalny $\mathbf{n}$ płata $\mathbf{p}$. Po obliczeniu macierzy $D\hat{\mathbf{p}}$ wystarczy obliczyć iloczyn wektorowy jej kolumn i po unormowaniu tego iloczynu mamy wektor $\hat{\mathbf{n}}$.

Przyjmiemy, że przekształcenie $\mathbf{q}$ jest afiniczne; niech $\mathbf{s} = \mathbf{q}(\mathbf{u}) = L\mathbf{u} + \mathbf{t}$. Niech $d(\mathbf{s}) = \tilde{d}\big(\mathbf{f}(\mathbf{s})\big)$, gdzie $\tilde{d}$ jest funkcją określoną w prostokącie $B = [a, b) \times [c, d)$ zaś funkcja $\mathbf{f}$ sprowadza punkt $\mathbf{s} \in \mathbb{R}^2$ do prostokąta B, co polega na obliczeniu wektora $\mathbf{s}' = \mathbf{u} + [k(b-a), l(d-c)]^T$ z odpowiednio dobranymi liczbami *całkowitymi* k, l. Dzięki temu złożenie funkcji $d \circ \mathbf{q} = \tilde{d} \circ \mathbf{f} \circ \mathbf{q}$ jest funkcją podwójnie okresową w płaszczyźnie $\mathbb{R}^2$ zawierającej dziedzinę A rysowanego płata. Wypada zadbać o to, aby była to funkcja ciągła; w tym celu wystarczy wybrać taką funkcję $\tilde{d}$, której obcięcia do przeciwległych boków prostokąta B są identyczne.

Rola przekształcenia $\mathbf{q}$ polega na skalowaniu wzorca zaburzeń kształtu, jaki pojawi się na obrazie powierzchni. Średnica zaburzeń maleje ze wzrostem wartości własnych macierzy L, a jeśli one znacznie się różnią, to otrzymujemy efekt anizotropii. Dobierając współczynniki macierzy L, trzeba zapewnić ciągłość sklejenia funkcji $d \circ \mathbf{q}$ na wspólnym brzegu połączonych płatów[3]. Macierzą różniczki $D\mathbf{q}$ jest macierz L.

Pozostało wybrać konkretną funkcję $\tilde{d}$ i znaleźć sposób obliczania jej pochodnych cząstkowych (sama wartość funkcji jest nam niepotrzebna). Na początek eksperymentów proponuję dwie funkcje; potem Czytelnik może im wymyślać

[2]Poza tym różniczka odwzorowania Gaussa $D\mathbf{n}$ jest opisana przez pochodne drugiego rzędu parametryzacji $\mathbf{p}$. Nie chcemy się z nimi kłopotać.

[3]Jeśli to wyjaśnienie jest niezrozumiałe, to proszę poeksperymentować, zmieniając współczynniki tej macierzy i pooglądać wyniki eksperymentów. Jeśli jest zrozumiałe, to też proszę poeksperymentować.

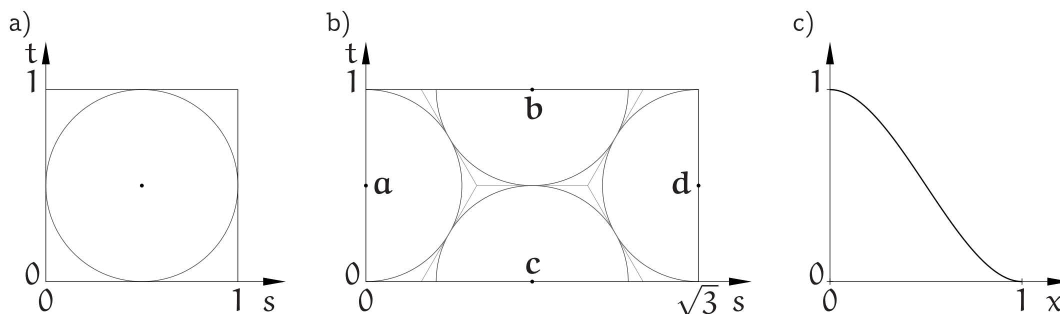
swoje zamienniki. Moje propozycje są dane za pomocą wzorów

$$\begin{aligned} r(s,t) &= \sqrt{(s-s_c)^2+(t-t_c)^2},\\ h(x) &= 1-2x^2+2x^3,\\ \tilde{d}(s,t) &= \begin{cases} ch\big(2r(s,t)\big) & \text{dla } r(s,t)<0.5,\\ 0 & \text{w przeciwnym razie.}\end{cases}\end{aligned}$$

Liczby s_c, t_c są współrzędnymi środka koła o promieniu 0.5, w którym funkcja $\tilde{d}$ przyjmuje wartości między zerem i ustaloną liczbą c. Gradient funkcji $\tilde{d}$ w punkcie (s,t) możemy obliczyć na podstawie wzorów

$$D\tilde{d} = \begin{cases} 2cDh(2r)Dr = \\ \quad 2c\big(-6(2r)+6(2r)^2\big)\left[\dfrac{s-s_c}{r}, \dfrac{t-t_c}{r}\right] & \text{dla } r<0.5,\\ [0,0] & \text{w przeciwnym razie.}\end{cases}$$

Pierwsza z proponowanych funkcji, $\tilde{d}_1$ jest określona w kwadracie $B=[0,1)^2$, którego punkt $(s_c,t_c)=(0.5,0.5)$ jest środkiem (rys. 19.1a).



Rysunek 19.1: Dziedziny funkcji $\tilde{d}_1$ i $\tilde{d}_2$ i wykres funkcji h

Dla drugiej funkcji, $\tilde{d}_2$, przyjąłem, że obszar B jest prostokątem $[0,\sqrt{3})\times[0,1)$. Można w niego wpisać cztery półkola o promieniu 0.5, których środkami są punkty $\mathbf{a}=(0,0.5)$, $\mathbf{b}=(0.5\sqrt{3},1)$, $\mathbf{c}=(0.5\sqrt{3},0)$ i $\mathbf{d}=(\sqrt{3},0.5)$. Mając punkt $(s,t)\in B$, należy wyznaczyć najbliższy mu punkt spośród tych czterech (oznaczmy go (s_c,t_c)) i obliczyć odległość r tych punktów. Aby znaleźć punkt najbliższy danego, wystarczy zbadać znaki następujących wyrażeń, które przyjmują wartość 0 na prostych, na których leżą zaznaczone na rysunku 19.1b odcinki oddzielające poszczególne półkola:

$$t-0.5,\quad \sqrt{3}s+t-1.5,\quad \sqrt{3}s-t-1.5,\quad \sqrt{3}s-t-0.5,\quad \sqrt{3}s+t-2.5.$$

Rozszerzenie okresowe funkcji $\tilde{d}_2$ na całą płaszczyznę daje parkietaż z sześciokątów foremnych z wpisanymi kołami o średnicy 1, w których ta funkcja

ma wartości niezerowe. Wybór funkcji h (rys 19.1c) zapewnia, że okresowe rozszerzenia obu funkcji, $\tilde{d}_1$ i $\tilde{d}_2$, mają ciągłe pochodne pierwszego rzędu.

Szadery

Szadery wierzchołków i sterowania rozdrabnianiem w programie używanym do rysowania oświetlonych obiektów nie wymagają żadnych zmian w porównaniu z poprzednim wariantem aplikacji. Szader rozdrabniania musi dodatkowo przekazać na swoje wyjście współrzędne punktu w dziedzinie płata i pochodne cząstkowe parametryzacji. Główna zmiana szadera geometrii wiąże się ze zmienionym interfejsem szaderów rozdrabniania i fragmentów — trzeba przekazać z wejścia na wyjście wektory pochodnych cząstkowych. Jeśli to szader geometrii ma wyprodukować wektor normalny, to musi też dokonać ortogonalizacji pochodnych cząstkowych do wektora normalnego. Do szadera fragmentów zostały dopisane procedury realizujące teksturę odkształceń.

Listing 19.1: Zmiany szadera rozdrabniania — wyjście

GLSL

```
1: layout(quads,equal_spacing,ccw) in;
2:
3: out GVertex {
4:     int  instance;
5:     vec4 Colour;
6:     vec3 Position;
7:     vec3 pu, pv, Normal;
8:     vec2 PatchCoord, TexCoord;
9:   } Out;
```

Listing 19.1 przedstawia zmiany bloku wyjściowego szadera rozdrabniania; dodane pole `instance` służy do przekazania dalej numeru instancji, tj. numeru płata Béziera, aby na jego podstawie można było wybrać różne tekstury odkształceń dla poszczególnych płatów rysowanych jednocześnie. To pole zostało dodane „na zaś”, aby ułatwić późniejsze przeróbki aplikacji. Ponadto na wyjście przekazywane są wektory pochodnych cząstkowych płata, w polach `pu` i `pv`, oraz punkt w dziedzinie płata (skopiowany ze zmiennej wejściowej `gl_TessCoord`) w polu `PatchCoord`.

Procedury `BPHorner2f`, `BPHorner3f` i `BPHorner4f`, których zadaniem jest obliczenie punktu i wektora normalnego płata Béziera odpowiednio wielomianowego płaskiego, wielomianowego trójwymiarowego i wymiernego trójwymiarowego reprezentowanego przez płat jednorodny w $\mathbb{R}^4$, zostały zmienione tak, aby dodatkowo obliczać wektory pochodnych cząstkowych;

Listing 19.2: Zmiany szadera rozdrabniania — wyznaczanie punktów płata

```
GLSL
1: void BPHorner3f ( float u, float v,
2:                   out vec4 pos, out vec4 pu, out vec4 pv, out vec4 nv )
3: {
4:   .... /*początek bez zmian */
5:   BCHorner3f ( bezp.udeg, q0, u, r, ru );
6:   BCHorner3f ( bezp.udeg, q1, u, rv, ruv );
7:   pos = vec4 ( r, 1.0 );
8:   pu = vec4 ( bezp.udeg * ru, 0.0 );
9:   pv = vec4 ( bezp.vdeg * rv, 0.0 );
10:  nv = vec4 ( cross ( ru, rv ), 0.0 );
11: } /*BPHorner3f*/
12: ....
13: void BPHorner4f ( float u, float v,
14:                   out vec4 pos, out vec4 pu, out vec4 pv, out vec4 nv )
15: {
16:   .... /*początek bez zmian */
17:   BCHorner4f ( bezp.udeg, q0, u, pos, ru );
18:   BCHorner4f ( bezp.udeg, q1, u, rv, nv );
19:   nv = cross ( pos, rv, ru );
20:   w = pos.w;
21:   pos /= w;
22:   ru *= bezp.udeg;
23:   pu = vec4 ( (ru.xyz-pos.xyz*ru.w)/w, 0.0 );
24:   rv *= bezp.vdeg;
25:   pv = vec4 ( (rv.xyz-pos.xyz*rv.w)/w, 0.0 );
26: } /*BPHorner4f*/
```

w związku z tym mają one dwa nowe parametry wyjściowe, pu i pv. Zmienione fragmenty procedur BPHorner3f i BPHorner4f są podane na listingu 19.2. Wektory te są reprezentowane w postaci jednorodnej, tj. z dołączoną współrzędną wagową 0. W liniach 7 i 8 obliczone wcześniej sumy są mnożone przez stopnie płata m i n (zobacz wzory (13.16) i (13.17)). W przypadku płata wymiernego wektory pochodnych parametryzacji wymiernej obliczamy na podstawie wzorów otrzymanych z następującego rachunku (parametry płata dla skrótu pomijamy):

$$\mathbf{p} = \frac{\mathbf{q}}{w} \quad \Rightarrow \quad \mathbf{q} = \mathbf{p}w,$$

$$\text{stąd} \qquad \mathbf{q}_u = \mathbf{p}_u w + \mathbf{p}w_u \quad \Rightarrow \quad \mathbf{p}_u = \frac{1}{w}(\mathbf{q}_u - \mathbf{p}w_u)$$

$$\text{oraz} \qquad \mathbf{q}_v = \mathbf{p}_v w + \mathbf{p}w_v \quad \Rightarrow \quad \mathbf{p}_v = \frac{1}{w}(\mathbf{q}_v - \mathbf{p}w_v),$$

przy czym $\mathbf{q}$ oznacza składający się z pierwszych trzech współrzędnych płata

jednorodnego $\mathbf{P}$ licznik ułamka definiującego płat wymierny $\mathbf{p}$, a funkcja w, czyli mianownik, opisuje czwartą współrzędną (wagową) płata $\mathbf{P}$. Obliczenie na podstawie powyższych wzorów jest realizowane przez instrukcje w liniach 20–25. Dodatkowo zmieniłem kolejność ostatnich dwóch parametrów wywołania procedury `cross` w linii 19. Powód tej zmiany jest taki: dalej w obliczeniach wektor normalny $\hat{\mathbf{n}}$ będzie obliczany jako iloczyn wektorowy pochodnych cząstkowych $\hat{\mathbf{p}}_u$ i $\hat{\mathbf{p}}_v$ płata zaburzonego i trzeba zadbać o zgodność zwrotów wektorów normalnych $\mathbf{n}$ i $\hat{\mathbf{n}}$; kąt między nimi ma być bliższy 0 niż π.

Nie zamieściłem w książce zmian procedury `BPHorner2f`, aby dać Czytelnikowi okazję do ćwiczenia polegającego na samodzielnym jej zmodyfikowaniu, co wymaga dokonania zmian także w procedurze `BCHorner2f`.

Zmiany w procedurze `main` szadera rozdrabniania są pokazane na listingu 19.3. W linii 6 jest przekazywany na wyjście numer instancji. W liniach 19 i 20 są przekazywane na wyjście wektory pochodnych cząstkowych płata przekształcone

Listing 19.3: Zmiany szadera rozdrabniania — procedura `main`

GLSL

```
1: void main ( void )
2: {
3:   vec4 pos, pu, pv, nv;
4:   int  i;
5:
6:   Out.instance = inst = In[0].instance;
7:   pos = nv = pu = pv = vec4 ( 0.0 );
8:   switch ( bezp.dim ) {
9: case 2:
10:     BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
11: case 3:
12:     BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
13: case 4:
14:     BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos, pu, pv, nv );  break;
15:   }
16:   Out.PatchCoord = gl_TessCoord.xy;
17:   gl_Position = trb.vpm * (trb.mm * pos);
18:   Out.Position = (trb.mm * pos).xyz;
19:   Out.pu = (trb.mm * pu).xyz;
20:   Out.pv = (trb.mm * pv).xyz;
21:   if ( !BezNormals || dot ( nv, nv ) < 1.0e-10 )
22:  .... /* dalej bez zmian */
23: } /*main*/
```

do układu współrzędnych świata (w którym szader podaje także punkt płata, w polu Position). Wyprowadzamy tylko pierwsze trzy współrzędne wektorów pochodnych, bo czwarta jest zerem; procedury BPHorner2f, BPHorner3f i BPHorner4f podają wektory o czterech współrzędnych, aby można je było wygodnie pomnożyć przez macierz przejścia do układu świata. Wektory pochodnych (w przeciwieństwie do wektora normalnego) są wektorami swobodnymi w przestrzeni, w której znajdują się nasze obiekty, dlatego mnożymy je przez macierz przejścia, a nie przez jej transpozycję odwrotności.

Na listingu 19.4 jest pokazany szader geometrii. W liniach 6–12 i 14–20 są pokazane zmienione bloki interfejsu, wejściowy (który ma identyczne pola jak blok wyjściowy na listingu 19.1) i wyjściowy, który mógłby być tak samo zbudowany, gdyby były w nim dozwolone pola typu int. Ponieważ nie są, numer instancji będzie dalej przekazywany do szadera fragmentów w zmiennej typu float[4].

Listing 19.4: Szader geometrii

GLSL

```
1: #version 450
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5:
6: in GVertex {
7:   int  instance;
8:   vec4 Colour;
9:   vec3 Position;
10:  vec3 pu, pv, Normal;
11:  vec2 PatchCoord, TexCoord;
12: } In[];
13:
14: out NVertex {
15:  float instance;
16:  vec4  Colour;
17:  vec3  Position;
18:  vec3  pu, pv, Normal;
19:  vec2  PatchCoord, TexCoord;
20: } Out;
21:
22: uniform bool BezNormals;
```

[4] W etapie rasteryzacji następuje interpolacja wartości zmiennych typu float, nie można jej zastosować do liczb całkowitych, nawet jeśli dla wszystkich wierzchołków trójkąta jest podana ta sama liczba.

```
23:
24: void main ( void )
25: {
26:   int i;
27:   vec3 v1, v2, nv;
28:
29:   v1 = In[1].Position - In[0].Position;
30:   v2 = In[2].Position - In[0].Position;
31:   nv = normalize ( cross ( v1, v2 ) );
32:   for ( i = 0; i < 3; i++ ) {
33:     gl_Position = gl_in[i].gl_Position;
34:     Out.Position = In[i].Position;
35:     if ( !BezNormals || dot ( nv, nv ) < 1.0e-10 ) {
36:       Out.pu = In[i].pu - dot ( In[i].pu, nv )*nv;
37:       Out.pv = In[i].pv - dot ( In[i].pv, nv )*nv;
38:       Out.Normal = nv;
39:     }
40:     else {
41:       Out.pu = In[i].pu;
42:       Out.pv = In[i].pv;
43:       Out.Normal = In[i].Normal;
44:     }
45:     Out.Colour = In[i].Colour;
46:     Out.PatchCoord = In[i].PatchCoord;
47:     Out.TexCoord = In[i].TexCoord;
48:     Out.instance = In[i].instance;
49:     EmitVertex ();
50:   }
51:   EndPrimitive ();
52: } /*main*/
```

Szader geometrii przekazuje na wyjście współrzędne punktu w dziedzinie płata oraz pochodne cząstkowe i wektor normalny, chyba że (z powodu osobliwości) wektor normalny jest wektorem zerowym lub zmiennej `BezNormals` została nadana wartość `false` — w tym przypadku szader geometrii ma za zadanie obliczyć i przekazać na wyjście wektor normalny płaszczyzny przetwarzanego trójkąta. Ale w takim przypadku szader musi dokonać rzutowania wektorów pochodnych cząstkowych na dwuwymiarową podprzestrzeń równoległą do tej płaszczyzny (czyli przeprowadzić ortogonalizację do wektora $\mathbf{n}$), co jest robione w liniach 36–37.

Listing 19.5 przedstawia zmieniony blok wejściowy i zmienne globalne oraz procedurę `main` szadera fragmentów. Blok wejściowy jest dopasowany do bloku

Listing 19.5: Zmiany szadera fragmentów — wejście i procedura `main`

GLSL

```
1: in NVertex {
2:     float instance;
3:     vec4  Colour;
4:     vec3  Position;
5:     vec3  pu, pv, Normal;
6:     vec2  PatchCoord, TexCoord;
7:   } In;
8:
9: uniform int ColourSource, NormalSource;
10:
11: Material mm;
12: vec3     normal, nv;
13:
14: void main ( void )
15: {
16:   nv = normalize ( In.Normal );
17:   normal = GetNormalVector ( In.PatchCoord );
18:   GetColours ( In.TexCoord );
19:   Lighting ();
20: } /*main*/
```

wyjściowego szadera geometrii. Mamy też nową zmienną jednolitą `NormalSource`, która służy do wybierania tekstury odkształceń, o czym będzie mowa dalej. Procedura `main` wywołuje kolejno procedurę odpowiedzialną za podanie (tj. przypisanie zmiennej `normal`) wektora normalnego, potem procedurę, której zadaniem jest przypisanie zmiennej `mm` własności materiału (które mogą być określone na podstawie tekstury), i wreszcie procedurę, która realizuje obliczenie koloru fragmentu przy użyciu wybranego modelu oświetlenia. Mamy tu jeszcze jedną zmienną globalną, `nv`, której procedura `main` przypisuje podany na wejściu szadera wektor normalny (po unormowaniu go, co jest potrzebne po etapie rasteryzacji). To jest wektor normalny $\mathbf{n}$ oryginalnej powierzchni. Procedura `GetNormalVector` przypisuje zmiennej `normal` wektor normalny $\hat{\mathbf{n}}$ powierzchni zaburzowej, przy czym jeśli tekstura odkształceń nie ma być stosowana, to jest to wektor normalny $\mathbf{n}$ oryginalnej powierzchni. Powód używania obu wektorów normalnych jest opisany dalej.

Na listingu 19.6 są pokazane procedury realizujące teksturę odkształceń, tj. obliczające wektor normalny $\hat{\mathbf{n}}$ w odpowiednim punkcie. Przekształcenie afiniczne $\mathbf{q}$ zostało tu zakodowane „na twardo” — w zmiennej `dtxm` jest taka

macierz L i wektor $\mathbf{t}$:

$$L = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}, \quad \mathbf{t} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Ponadto w linii 1 mamy „na twardo” zakodowaną stałą $c = -0.003$, która określa wysokość „nierówności” na powierzchni.

Procedura `GetNormalVector`, wywoływana przez `main`, zależnie od wartości zmiennej jednolitej `NormalSource` oddaje wektor normalny oryginalnego płata (świeżo unormowany przez `main`) lub wywołuje jeden z dwóch podprogramów realizujących teksturę odkształceń. Pierwszy z nich, `NormalTexture1`, w linii 24 realizuje przekształcenie $\mathbf{f} \circ \mathbf{q}$ punktu (u, v) z dziedziny płata; punkt ten jest poddawany przekształceniu afinicznemu $\mathbf{q}$ reprezentowanemu przez macierz L i wektor $\mathbf{t}$, a następnie funkcja `frac` zamienia otrzymane współrzędne na ich części ułamkowe — dostajemy więc punkt z kwadratu jednostkowego. W linii 25 jest wywoływana pomocnicza procedura `NormalTex`, której parametry są współrzędnymi wektora różnicy punktu $\mathbf{f}\bigl(\mathbf{q}(u, v)\bigr)$ i środka koła (x_c, y_c).

Procedura `NormalTexture2` w linii 33 poddaje punkt (u, v) przekształceniu afinicznemu $\mathbf{q}$. W linii 34 obraz tego punktu, ze współrzędną s przeskalowaną o czynnik $1/\sqrt{3}$, jest zastępowany punktem leżącym w kwadracie jednostkowym. Obliczenia w liniach 35–46 odbywają się w układzie z przeskalowaną o ten czynnik osią s. Wartość wyrażenia w przełączniku w linii 35 reprezentuje ćwiartkę obszaru B, w której znajduje się punkt (s, t). W każdej z tych ćwiartek leży jeden

Listing 19.6: Zmiany szadera fragmentów — tekstury proceduralne

GLSL

```
1: #define C     (-0.003)
2: #define SQRT3 1.7320508
3:
4: const vec2 dtxm[3] = { vec2(10.0,0.0), vec2(0.0,10.0), vec2(0.0,0.0) };
5:
6: vec3 NormalTex ( float dx, float dy )
7: {
8:   float r;
9:   vec2  Dd, Ddq;
10:  unlvec3  dpu, dpv;
11:
12:  r = sqrt ( dx*dx + dy*dy );
13:  if ( r >= 0.5 || r == 0.0 )
14:    return nv;
```

```
15:   Dd = C*(48.0*r-24.0)*vec2 ( dx, dy );
16:   Ddq = Dd.x*dtxm[0] + Dd.y*dtxm[1];
17:   dpu = In.pu + Ddq.x*nv;
18:   dpv = In.pv + Ddq.y*nv;
19:   return normalize ( cross ( dpu, dpv ) );
20: } /*NormalTex*/
21:
22: vec3 NormalTexture1 ( vec2 pc )
23: {
24:   pc = fract ( dtxm[0]*pc.x + dtxm[1]*pc.y + dtxm[2] );
25:   return NormalTex ( pc.x-0.5, pc.y-0.5 );
26: } /*NormalTexture1*/
27:
28: vec3 NormalTexture2 ( vec2 pc )
29: {
30:   int  c;
31:   float dx, dy;
32:
33:   pc = dtxm[0]*pc.x + dtxm[1]*pc.y + dtxm[2];
34:   pc = fract ( vec2 ( pc.x/SQRT3, pc.y ) );
35:   switch ( (pc.x > 0.5 ? 1 : 0) + (pc.y > 0.5 ? 2 : 0) ) {
36: case 0: c = 3.0*pc.x-pc.y > 0.5 ? 1 : 0;  break;
37: case 1: c = 3.0*pc.x+pc.y > 2.5 ? 3 : 1;  break;
38: case 2: c = 3.0*pc.x+pc.y > 1.5 ? 2 : 0;  break;
39: case 3: c = 3.0*pc.x-pc.y > 1.5 ? 3 : 2;  break;
40:   }
41:   switch ( c ) {
42: case 0: dx = pc.x;      dy = pc.y-0.5;  break;  /* a */
43: case 1: dx = pc.x-0.5;  dy = pc.y;      break;  /* c */
44: case 2: dx = pc.x-0.5;  dy = pc.y-1.0;  break;  /* b */
45: case 3: dx = pc.x-1.0;  dy = pc.y-0.5;  break;  /* d */
46:   }
47:   return NormalTex ( dx*SQRT3, dy );
48: } /*NormalTexture2*/
49:
50: vec3 GetNormalVector ( vec2 pc )
51: {
52:   switch ( NormalSource ) {
53: default: return nv;
54:  case 1: return NormalTexture1 ( pc );
55:  case 2: return NormalTexture2 ( pc );
56:   }
57: } /*GetNormalVector*/
```

z ukośnych odcinków dzielących obszar B na regiony Woronoja (tj. zbiory punktów leżących najbliżej każdego z czterech punktów (środków) $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$, zobacz rysunek 19.1). Instrukcje w liniach 36–39 badają, po której stronie odpowiedniego odcinka leży punkt (s,t) i przypisują zmiennej c *numer* najbliższego środka. Instrukcje w liniach 42–45 obliczają współrzędne różnicy punktu (s,t) i znalezionego najbliższego srodka półkola. W linii 47 następuje wywołanie procedury `NormalTex`, przy czym jest dokonywane skalowanie współrzędnej s odwrotne do skalowania dokonanego w linii 34.

Procedura `NormalTex` wykonuje końcowe zaburzenie wektora normalnego. W linii 12 jest obliczana odległość r punktu (s,t) od środka. Jeśli odległość jest zerowa lub nie mniejsza niż 0.5, to wektor normalny powierzchni zaburzonej jest taki sam jak wektor normalny oryginalnej powierzchni. W przeciwnym razie w linii 15 następuje obliczenie gradientu Dd funkcji d, a w linii 16 jest obliczany iloczyn $Dd \cdot D\mathbf{q}$ tego gradientu i macierzy różniczki przekształcenia $\mathbf{q}$ (czyli macierzy L). W liniach 17–18 są obliczane pochodne cząstkowe parametryzacji $\hat{\mathbf{p}}$, a całość wieńczy, w linii 19, obliczenie ich iloczynu wektorowego, normalizacja i przekazanie wyniku na zewnątrz w instrukcji return.

Listing 19.7: Zmiany szadera fragmentów — oświetlenie lambertowskie

GLSL
```
1: subroutine (LightingProc) void LambertLighting ( void )
2: {
3:   .... /* początek bez zmian */
4:   vv = posDifference ( trb.eyepos, In.Position, dist );
5:   e = dot ( vv, nv );
6:   out_Colour = vec4(0.0);
7:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
8:     if ( (light.mask & mask) != 0 ) {
9:       .... /* tu zmian też nie ma */
10:      d = dot ( lv, normal );
11:      if ( e > 0.0 ) {
12:        if ( dot ( lv, nv ) > 0.0 ) { .... /* jako i tu */ }
13:      }
14:      else {
15:        if ( dot ( lv, nv ) < 0.0 ) { .... /* i tu */ }
16:      }
17:    }
18:  out_Colour = vec4 ( clamp ( out_Colour.rgb, 0.0, 1.0 ), 1.0 );
19: } /*LambertLighting*/
```

Na listingu 19.7 są pokazane zmienione instrukcje procedury realizującej model oświetlenia Lamberta. W liniach 5, 12 i 17 do obliczeń zamiast wektora $\hat{\boldsymbol{n}}$ (przechowywanego w zmiennej globalnej `normal`) jest podstawiany wektor normalny $\boldsymbol{n}$ oryginalnej powierzchni (przechowywany w zmiennej `nv`). To są obliczenia mające na celu ustalenie, czy obserwator i źródło światła znajdują się po tej samej stronie płaszczyzny stycznej do powierzchni, czy też po przeciwnych stronach. Użycie w tym miejscu wektora $\hat{\boldsymbol{n}}$ skutkowałoby błędnie oświetlonymi (lub błędnie nieoświetlonymi) pikselami na obrazie. Do samego modelu oświetlenia podstawiamy oczywiście wektor $\hat{\boldsymbol{n}}$. Analogiczna zmiana powinna być wprowadzona w procedurze `BlinnPhongLighting`, ale uznałem, że można jej tu nie zamieszczać.

Zmiany w aplikacji

W kodzie aplikacji napisanym w C są potrzebne bardzo niewielkie zmiany. Procedura `LoadMyShaders` w dodatku do dotąd używanych zmiennych jednolitych musi uzyskać dostęp do zmiennej `NormalSource`, której będzie nadawać wartości `0`, `1` lub `2`. Aby umożliwić użytkownikowi przełączanie rodzajów tekstur odkształceń, do procedury `CharFunc` została dodana w instrukcji przełącznika reakcja na napisanie znaku `'D'` lub `'d'`:

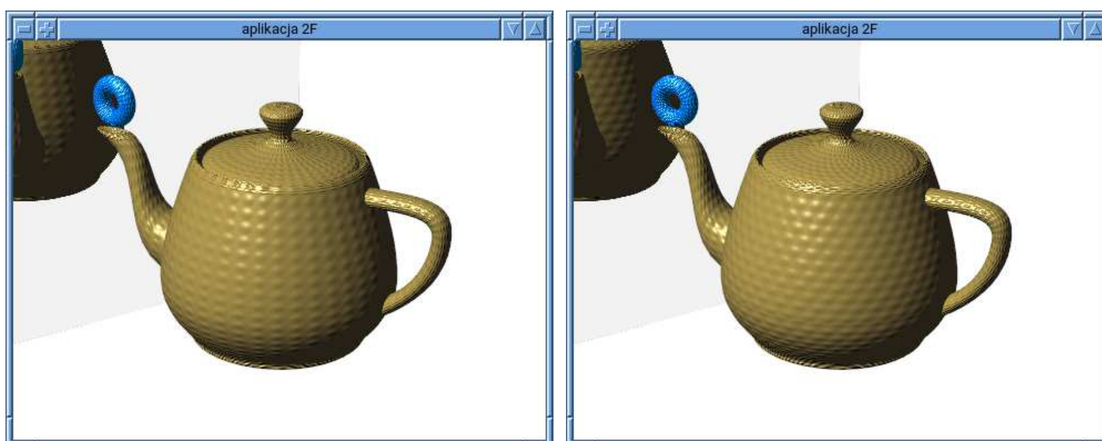
```
case 'D':  case 'd':
    normal_source = (normal_source+1) % 3;
    redraw = true;
    break;
```

przy czym `normal_source` jest zmienną globalną typu `GLint` o początkowej wartości `0`. Przed rysowaniem czajnika i torusa wartość tej zmiennej należy przypisać zmiennej jednolitej `NormalSource` za pomocą procedury `glUniform1i`.

Ćwiczenia

1. Wykonaj eksperymenty polegające na zmienianiu macierzy L w kodzie szadera fragmentów. Zmieniaj też stałą c i oglądaj skutki tych zmian na obrazach.

2. Przerób zmienną szadera przechowującą macierz L o nadanej stałej wartości na zmienną jednolitą, aby umożliwić aplikacji przypisywanie nowych wartości współczynnikom tej macierzy. Oprogramuj to przypisywanie i poeksperymentuj.

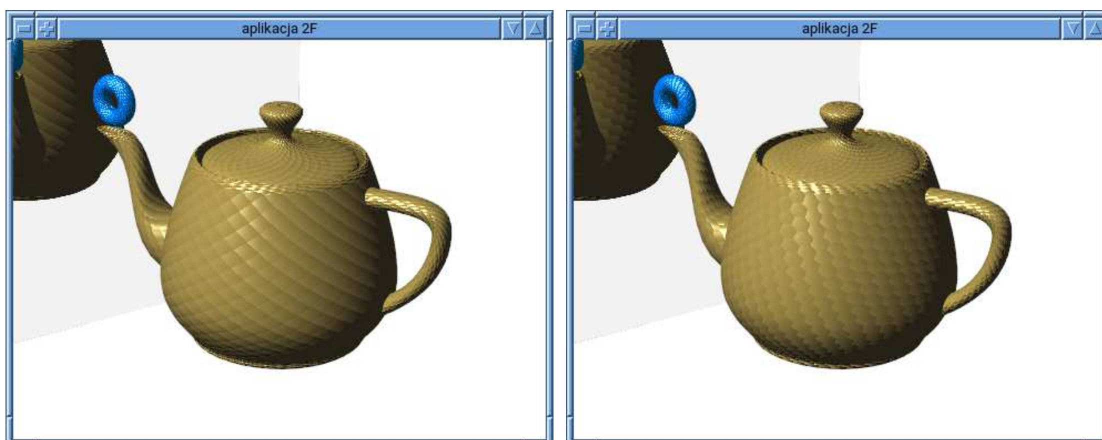


Rysunek 19.2: Okno aplikacji drugiej F

3. Zmień funkcję h użytą do zdefiniowania funkcji $\tilde{d}_1$ i $\tilde{d}_2$; zmodyfikuj procedurę `NormalTex` tak, aby realizowała obliczenie tekstury odkształceń z nową funkcją.

4. Dokonaj takich modyfikacji szaderów i aplikacji, aby dla każdego płata było możliwe nałożenie innej tekstury odkształceń (w tym indywidualne określanie przekształcenia $\mathbf{q}$ i stałej c).

5. Zastanów się, jak użyć tablicy tekseli do reprezentowania tekstury odkształceń i nie poprzestań na zastanawianiu się.



Rysunek 19.3: Rozwiązania niektórych ćwiczeń

20. Aplikacja druga G

Z dosyć już głębokiej wody przeskoczymy do jeszcze trochę głębszej: wprowadzimy na obrazach cienie. Nasze źródła światła są punktowe. Dowolny punkt $\mathbf{p}$ w przestrzeni jest bezpośrednio oświetlony przez takie źródło światła położone w punkcie $\mathbf{s}$, jeśli odcinek $\overline{\mathbf{sp}}$ nie ma żadnego punktu wspólnego z powierzchnią dowolnego obiektu w rysowanej scenie — z wyłączeniem punktu $\mathbf{p}$, który (jeśli jest punktem rysowanej powierzchni) zasłania przed światłem punkty położone dalej[1]. To samo dotyczy źródeł światła położonych w nieskończonej odległości od sceny (tylko że wtedy rozważamy półprostą zamiast odcinka).

Zbiór punktów, które nie są bezpośrednio oświetlone przez dane źródło światła, nazwiemy obszarem cienia tego źródła. Aby otrzymać poprawne cienie na obrazie, należy dla każdego fragmentu (piksela) zbadać, czy punkty powierzchni odpowiadające temu fragmentowi są bezpośrednio oświetlone, czy też należą do obszaru cienia. Algorytmy cieni odpowiednie do implementacji działającej na GPU tworzą pewną reprezentację obszaru cienia. Są dwa podstawowe sposoby reprezentowania go.

Pierwszy sposób[2] polega na zbudowaniu i narysowaniu bryły cienia, która jest (teoretycznie) sumą elementarnych brył cienia, tj. ściętych nieograniczonych ostrosłupów (dla źródeł światła w odległości skończonej) lub graniastosłupów. „Przednią" podstawą każdego z nich jest trójkątna ściana należącego do sceny obiektu, zaś boczne ściany są zbiorami punktów zasłoniętych od światła przez krawędzie tego trójkąta. Na potrzeby algorytmu elementarne bryły cienia obcina się także „z tyłu", za obszarem, w którym znajdują się obiekty sceny. Wszystkie ściany tych brył muszą być spójnie zorientowane, np. tak, aby wektor normalny obliczony dla podanej kolejności wierzchołków był zwrócony na zewnątrz bryły. Scenę rysuje się dwukrotnie, przy czym jedynym potrzebnym wynikiem pierwszego rysowania jest zawartość bufora głębokości — informacja o głębokości widocznego punktu dla każdego piksela. Następnie trzeba narysować wszystkie ściany elementarnych brył cienia. Dla każdego piksela wprowadzamy licznik o początkowej wartości 0. Rysując w tym kroku fragment, trzeba porównać głębokość punktu z liczbą zapisaną w buforze głębokości dla danego piksela i dalej przetwarzać tylko te fragmenty, których głębokość jest mniejsza (albo większa — mamy dwa warianty algorytmu, zwane *depth pass* i *depth fail*). Dla fragmentów ścian odwróconych przodem licznik zwiększamy, a dla odwróconych tyłem zmniejszamy o 1. Ostatni krok algorytmu to ponowne rysowanie sceny. Obliczenia

[1] Ale żaden punkt sam siebie od światła nie zasłania.

[2] Wynalazł go w r. 1977 Frank Crow, dalej rozwijało go wiele osób, przy czym najczęściej łączy się to podejście z nazwiskiem Johna Carmacka, który użył go w r. 2000 w grze *Doom 3*.

oświetlenia możemy wykonywać tylko dla fragmentów widocznych (o głębokości równej głębokości zapamiętanej po pierwszym rysowaniu sceny), co oszczędza czas. Jeśli licznik dla danego piksela ma wartość 0, to punkt jest oświetlony[3]. Efektywne implementacje, umożliwiające nawet animowanie cieni w czasie rzeczywistym (np. w grach), wykorzystują bufor szablonu (*stencil buffer* — wspomniane liczniki mogą być przechowywane w nim), czym tu się jednak nie zajmiemy.

Zastosujemy drugie podejście, które polega na narysowaniu obrazu sceny widzianej z punktu położenia źródła światła lub z kierunku padania światła dochodzącego z nieskończonej odległości. Ten obraz jest niepotrzebny, ale otrzymana podczas jego tworzenia zawartość bufora głębokości (nazywana niestety mapą cienia[4]) dla obrazu o dostatecznej rozdzielczości jest dostatecznie dokładną reprezentacją obszaru cienia. Podczas rysowania właściwego obrazu, dla każdego fragmentu szader sprawdzi, czy odpowiadający temu fragmentowi punkt w przestrzeni należy do bryły cienia i uwzględni ten fakt w obliczeniach oświetlenia.

Konstrukcja rzutowania sceny dla źródeł światła

Jeśli źródło światła jest położone w nieskończonej odległości od sceny, to potrzebny jest rzut równoległy; kierunek rzutowania jest kierunkiem do źródła światła, a rzutnia powinna (choć teoretycznie nie musi) być do tego kierunku prostopadła. Jeśli odległość do źródła światła jest skończona, to potrzebny jest rzut perspektywiczny. Rozwiążemy problem uproszczony, w którym źródło światła leży dostatecznie daleko od sceny, aby można było ją przedstawić w całości na obrazie wykonanym w jednym rzucie[5]. Założymy, że scena jest zawarta w kuli[6] o środku $\mathbf{c}$ i promieniu R. Przyjmiemy, że opisane tu rozwiązanie jest stosowalne, jeśli odległość d położenia źródła światła od punktu $\mathbf{c}$ jest nie mniejsza niż $R\sqrt{2}$, co daje gwarancję, że całą scenę można zrzutować (perspektywicznie) na jedną ścianę pewnego sześcianu, którego środek — położenie źródła światła — jest środkiem rzutowania. W obu przypadkach potrzebujemy macierzy V opisującej

[3] Przy założeniu, że położenie obserwatora jest poza bryłą cienia, jeśli w etapie drugim braliśmy pod uwagę fragmenty o głębokości mniejszej — pod tym względem lepszy jest drugi wariant, w którym licznik zwiększamy lub zmniejszamy, jeśli punkt na ścianie bryły cienia ma większą głębokość, bo w tym wariancie nie ma znaczenia, czy obserwator jest w cieniu, czy nie jest.

[4] Ja tak nie zamierzam.

[5] Jeśli tak nie jest (na przykład gdy źródło światła jest lampą umieszczoną między obiektami, np. w pomieszczeniu), to punkt położenia źródła światła trzeba „obudować” sześcienną kostką i narysować obrazy sceny na wszystkich ścianach tej kostki. System teksturowania w OpenGL-u umożliwia to, ale tymczasem zajmiemy się prostszym przypadkiem.

[6] Jest, mam nadzieję, jasne, że aby otrzymać dobrą dokładność, dla każdej konkretnej sceny trzeba starać się o znalezienie otaczającej scenę kuli o jak najmniejszym promieniu.

przejście od układu świata do układu obserwatora i macierzy P reprezentującej przejście od układu obserwatora do układu kostki standardowej. Założymy, że obraz jest kwadratowy, tj. odpowiedni kwadrat położony na rzutni będzie odwzorowany na klatkę, której wysokość i szerokość w pikselach są takie same.

Zaczniemy od skonstruowania macierzy przekształceń dla źródeł światła położonych w odległości nieskończonej. Początek układu obserwatora umieścimy w punkcie $\mathbf{c}$. Wektor $\mathbf{l}$ opisujący kierunek, z którego dochodzi światło, po unormowaniu ma być wersorem osi z w układzie obserwatora. Przyjmiemy, że wersory osi x i y są jednostkowe i wszystkie trzy wersory mają być do siebie nawzajem prostopadłe. Zauważmy, że te warunki zostawiają nam sporą swobodę — możemy te dwa wektory dowolnie obracać wokół osi z, możemy też zmienić ich zwroty lub kolejność (czyli zmienić orientację układu). Dlatego w konstrukcji użyjemy odbicia Householdera, którego skutkiem będzie przypadkowe (ale poprawne) położenie osi x i y, i które ma dobre własności numeryczne. Przeprowadzi ono wektor $\boldsymbol{e}_3 = (0, 0, 1)$ (wersor osi z układu świata) na wektor o kierunku $\mathbf{l}$. W tym celu obliczamy wektor normalny płaszczyzny odbicia

$$\boldsymbol{v} = \mathbf{l} \pm \|\mathbf{l}\|_2 \boldsymbol{e}_3,$$

przy czym znak „+” albo „−” wybieramy tak, aby wektor $\boldsymbol{v}$ był jak najdłuższy (a zatem wybieramy „+” jeśli trzecia współrzędna wektora $\mathbf{l}$ jest dodatnia i „−” w przeciwnym razie). Dalej obliczamy czynnik $\gamma = 2/\langle \boldsymbol{v}, \boldsymbol{v} \rangle$ i poddajemy odbiciu wersory osi x i y (wektory $\boldsymbol{e}_1 = (1, 0, 0)$ i $\boldsymbol{e}_2 = (0, 1, 0)$), tj. obliczamy

$$\boldsymbol{w}_i = \boldsymbol{e}_i - \boldsymbol{v}\gamma\langle \boldsymbol{v}, \boldsymbol{e}_i \rangle, \quad i = 1, 2.$$

Następnie konstruujemy macierz 4×4 opisującą przejście od układu obserwatora do układu świata:

$$V^{-1} = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \boldsymbol{w}_3 & \mathbf{c} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{gdzie} \quad \boldsymbol{w}_3 = \frac{1}{\|\mathbf{l}\|_2}\mathbf{l}.$$

Macierz V przejścia od układu świata do układu obserwatora otrzymamy, znajdując odwrotność[7] macierzy V^{-1}.

[7] Przejście opisane przez macierz V^{-1} jest afiniczną izometrią; macierz ta ma strukturę

$$\begin{bmatrix} Q & \mathbf{c} \\ 0^T & 1 \end{bmatrix}$$

z blokiem ortogonalnym $Q = [\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3]$. Odwrotność takiej macierzy ma postać

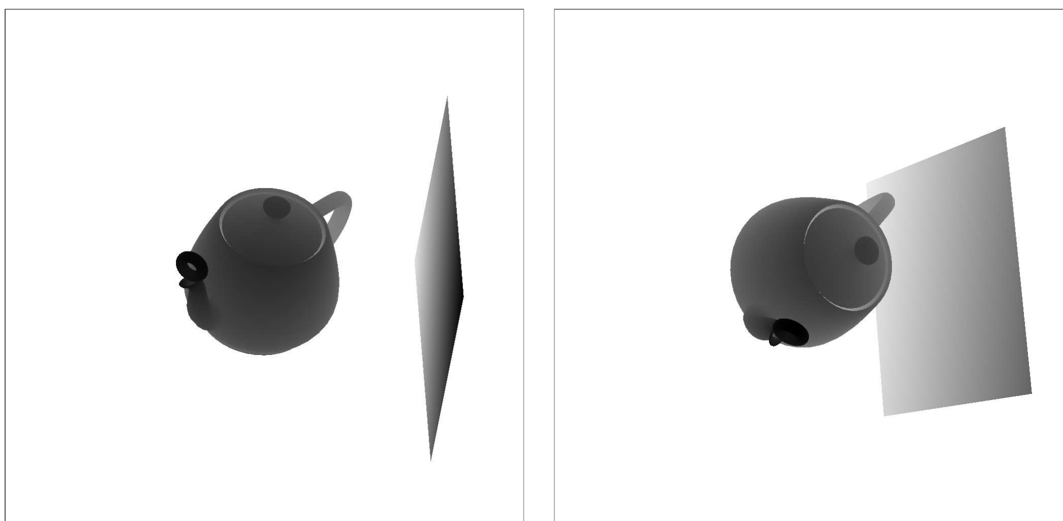
$$\begin{bmatrix} Q^T & -Q^T\mathbf{c} \\ 0^T & 1 \end{bmatrix}.$$

Użycie w tym przypadku ogólnej procedury znajdującej macierz odwrotną jest trochę mało eleganckie, choć dopuszczalne. Wzór podany wyżej jest tańszy i daje dokładniejszy wynik.

Zawierająca scenę kula o promieniu R i środku $\mathbf{c}$ jest w układzie obserwatora kulą o promieniu R i środku $\mathbf{0}$, możemy ją zatem wpisać w kostkę $[-R, R]^3$. Stąd współczynniki odpowiedniej macierzy P obliczy procedura `M4x4Orthof` wywołana z parametrami $l = b = n = -R$ i $r = t = f = R$ (zobacz rozdz. 6).

Dla źródła światła położonego w punkcie $\mathbf{s}$ (w skończonej odległości od sceny) układ obserwatora ma mieć początek w punkcie $\mathbf{s}$. Oś z tego układu będzie miała kierunek i zwrot wektora $\mathbf{s} - \mathbf{c}$ o długości d. Konstruując macierz V^{-1} przejścia od układu obserwatora do układu świata, podstawimy ten wektor w miejsce wektora $\mathbf{l}$ w opisanym wyżej rachunku dla światła dochodzącego z nieskończenie daleka, po czym pierwsze dwie kolumny macierzy V^{-1} obliczymy tak jak poprzednio, a trzecią, normując wektor $\mathbf{s} - \mathbf{c}$. Macierz P, opisującą przejście do układu kostki standardowej, skonstruuje procedura `M4x4Frustumf` z parametrami $n = d - R$, $f = d + R$, $l = b = -n \operatorname{tg} \alpha$, $r = t = n \operatorname{tg} \alpha$, gdzie $\sin \alpha = R/d$.

Na rysunku 20.1 jest pokazana zawartość bufora głębokości dla dwóch źródeł światła oświetlających scenę rysowaną przez naszą aplikację. Jasność punktów na obrazie jest powiązana z ich głębokością — im punkt jest dalej, tym jest jaśniejszy, w szczególności punkty położone na płaszczyźnie tylnej ściany kostki standardowej (tj. płaszczyźnie $z = 1$) i za nią są białe. Obraz z lewej strony powstał dla światła umieszczonego w odległości nieskończonej (i scena jest przedstawiona w rzucie równoległym), a obraz z prawej odpowiada źródłu w skończonej odległości (i rzut jest perspektywiczny). Punkt $\mathbf{c}$ jest tu początkiem układu świata, $R = 2.2$, drugie źródło światła znajduje się w punkcie $(-6, 2, 8)$ (zatem $d \approx 10.2$, $\alpha \approx 0.2174$).



Rysunek 20.1: Zawartość bufora głębokości dla rzutów sceny określonych przez dwa źródła światła

Na listingu 20.1 są pokazane procedury realizujące opisane wyżej obliczenia. Zmieniona struktura LSPar, opisująca źródło światła, ma nowe pola shadow_fbo i shadow_txt, które są identyfikatorami bufora ramki i tekstury używanej jako bufor głębokości, w którym powstaje reprezentacja obszaru cienia, oraz tablice shadow_view i shadow_proj, w których będą przechowywane macierze V i P potrzebne do wyznaczania cieni. Zmienna globalna light jest strukturą, w której mamy liczbę źródeł światła (w polu nls), maskę bitową opisującą źródła światła włączone w danej chwili i tablicę struktur LSPar opisujących źródła światła.

Listing 20.1: Procedury konstrukcji macierzy V i P dla źródeł światła

C

```
1: typedef struct LSPar {
2:           GLfloat ambient[4];
3:           GLfloat diffuse[4];
4:           GLfloat position[4];
5:           GLfloat attenuation[3];
6:           GLuint  shadow_fbo, shadow_txt;
7:           GLfloat shadow_view[16], shadow_proj[16];
8:         } LSPar;
9:
10: typedef struct LightBl {
11:           GLuint nls, mask, shmask;
12:           LSPar  ls[MAX_NLIGHTS];
13:         } LightBl;
14:
15: LightBl light;
16:
17: static void SetupShadowViewerTrans ( GLfloat org[3], GLfloat zv[3],
18:                                      GLfloat vmat[16] )
19: {
20:   GLfloat v[3], a[16], g, r, s;
21:   int     i, j;
22:
23:   memcpy ( v, zv, 3*sizeof(float) );
24:   r = sqrt ( V3DotProductf ( v, v ) );
25:   v[2] += v[2] > 0.0 ? r : -r;
26:   g = 2.0/V3DotProductf ( v, v );
27:   M4x4Identf ( a );
28:   for ( i = 0; i < 2; i++ ) {
29:     s = v[i]*g;
30:     for ( j = 0; j < 3; j++ )
31:       a[4*i+j] -= s*v[j];
32:   }
```

```
33:   a[8] = zv[0]/r;  a[9] = zv[1]/r;  a[10] = zv[2]/r;
34:   memcpy ( &a[12], org, 3*sizeof(GLfloat) );
35:   M4x4InvertAffineIsometryf ( vmat, a );
36: } /*SetupShadowViewerTrans*/
37:
38: void SetupShadowTransformations ( int l, float sc[3], float R )
39: {
40:   GLfloat *lvm, *lpm;
41:   GLfloat lpos[3], v[3], d, n, s, t;
42:   int     i;
43:
44:   if ( l < 0 || l >= MAX_NLIGHTS )
45:     return;
46:   if ( light.ls[l].shadow_txt ) {
47:     lvm = light.ls[l].shadow_view;
48:     lpm = light.ls[l].shadow_proj;
49:     if ( light.ls[l].position[3] != 0.0 ) {
50:             /* źródło światła w skończonej odległości */
51:       for ( i = 0; i < 3; i++ ) {
52:         lpos[i] = light.ls[l].position[i]/light.ls[l].position[3];
53:         v[i] = lpos[i] - sc[i];
54:       }
55:       SetupShadowViewerTrans ( lpos, v, lvm );
56:       d = V3DotProductf ( v, v );
57:       if ( d > 2.0*R*R ) {
58:         d = sqrt ( d );  n = d-R;
59:         s = R/d;                 /* sin(alpha) */
60:         t = s/sqrt ( 1.0-s*s );  /* tg(alpha) */
61:         M4x4Frustumf ( lpm, NULL, -n*t, n*t, -n*t, n*t, n, d+R );
62:       }
63:       else {  /* alpha == PI/4 */
64:         n = 0.4142*R;   /* sqrt(2)-1 ≅ 0.4142, sqrt(2)+1 ≅ 2.4143 */
65:         M4x4Frustumf ( lpm, NULL, -n, n, -n, n, n, 2.4143*R );
66:       }
67:     }
68:     else {  /* źródło światła w odległości nieskończonej */
69:       memcpy ( lpos, sc, 3*sizeof(GLfloat) );
70:       memcpy ( v, light.ls[l].position, 3*sizeof(GLfloat) );
71:       SetupShadowViewerTrans ( lpos, v, lvm );
72:       M4x4Orthof ( lpm, NULL, -R, R, -R, R, -R, R );
73:     }
74:   }
75: } /*SetupShadowTransformations*/
```

Dodatkowe pole shmask jest maską bitową, w której pamiętamy, dla których źródeł światła jest utworzona tekstura obszaru cienia — być może nie dla wszystkich źródeł światła zechcemy ją utworzyć, jako że to jest dosyć kosztowne (w tym sensie, że zajmuje dużo pamięci GPU) i możemy chcieć wyznaczać cienie tylko dla niektórych źródeł światła[8].

Pomocnicza procedura SetupShadowViewerTrans ma za zadanie skonstruowanie macierzy V, której współczynniki mają trafić do tablicy vmat. Parametr org jest tablicą ze współrzędnymi początku układu obserwatora (czyli środka kuli $\mathbf{c}$ albo punktu położenia źródła światła $\mathbf{s}$). Liczby w tablicy zv są współrzędnymi (w układzie świata) wektora $\mathbf{l}$ albo $\mathbf{s} - \mathbf{c}$, który ma kierunek i zwrot wersora osi z układu obserwatora (i który może mieć dowolną długość, byle nie 0). W liniach 23–25 jest konstruowany wektor normalny płaszczyzny odbicia $\boldsymbol{v}$. W linii 26 obliczamy czynnik γ, w linii 27 wpisujemy do tablicy a współczynniki macierzy jednostkowej, a w pętli w liniach 28–32 stosujemy odbicie do wersorów pierwszych dwóch osi układu świata — zauważmy, że iloczyn skalarny wektora $\boldsymbol{e}_i$ i wektora $\boldsymbol{v}$ jest i-tą współrzędną tego drugiego wektora. Zamiast odbijać wersor osi z (tj. wektor $\boldsymbol{e}_3$) a potem ewentualnie korygować zwrot otrzymanego wektora, w linii 33 dzielimy współrzędne wektora podanego w tablicy zv przez długość tego wektora i wpisujemy ilorazy do trzeciej kolumny konstruowanej macierzy V^{-1}. W linii 34 wpisujemy do czwartej kolumny współrzędne początku układu obserwatora. Wywołana w linii 35 procedura M4x4InvertAffineIsometryf (dodana chyłkiem do pliku utilities.c) znajduje odwrotność V macierzy V^{-1} sposobem opisanym w przypisie 7 na stronie 20.3.

Procedura SetupShadowTransformations otrzymuje jako parametry numer źródła światła, środek kuli otaczającej scenę i promień R tej kuli. Jeśli jest utworzona tekstura obszaru cienia związanego z l-tym źródłem światła, to procedura konstruuje odpowiednie macierze V_l i P_l (tj. V i P). Przypomnijmy, że jeśli wektor współrzędnych jednorodnych ma współrzędną wagową równą 0, to reprezentuje punkt niewłaściwy, czyli kierunek, w którym znajduje się źródło światła położone nieskończenie daleko. Zatem warunek sprawdzany w linii 49 jest spełniony przez źródła światła położone w odległości skończonej. W pętli w liniach 51–54 obliczamy współrzędne kartezjańskie punktu $\mathbf{s}$ (położenia źródła światła) i współrzędne wektora $\mathbf{s} - \mathbf{c}$. Wynikiem wywołania procedury SetupShadowViewerTrans w linii 55 jest macierz V, którą zapamiętujemy w polu shadow_view struktury LSPar. W liniach 57–67 jest konstruowana macierz P, przy czym jeśli odległość źródła światła od punktu $\mathbf{c}$ jest większa niż $R\sqrt{2}$ (to sprawdzamy w linii 57), to obliczamy tangens takiego kąta α, aby cała kula się

[8]Wbrew pozorom to miewa sens.

zmieściła na obrazie, a w przeciwnym razie przyjmujemy $\alpha = \frac{\pi}{4}$. Obliczenia w liniach 69–72, których wynikiem są macierze V i P dla światła dochodzącego z oddali, pozostawię bez komentarza.

Szadery

Do wykonywania obrazów z cieniami potrzebujemy *dwóch różnych* programów rysujących płaty Béziera, do wyznaczania reprezentacji obszaru cienia i do wykonywania końcowego obrazu[9]. W etapie wyznaczania reprezentacji obszaru cienia chcemy otrzymać tylko odpowiednią zawartość bufora głębokości, zatem możemy i powinniśmy użyć w tym etapie maksymalnie uproszczonych szaderów — nie mają dla nas znaczenia kolory pikseli i w gruncie rzeczy nawet nie musimy tworzyć obrazu. Rysowanie odbędzie się w trybie pozaekranowym, tj. w teksturze będącej załącznikiem do bufora ramki utworzonego specjalnie w tym celu. Tekstura ta, którą nazywamy teksturą obszaru cienia, zostanie użyta jako bufor głębokości, a potem będzie udostępniona szaderowi fragmentów programu wykonującego końcowy obraz. Szader fragmentów wstępnego etapu nie musi zatem wykonywać żadnych obliczeń; może przypisać na wyjście dowolny kolor[10], który i tak będzie zignorowany, więc nawet tego nie musi robić. Natomiast szadery części przedniej (wierzchołków, rozdrabniania i geometrii) muszą spowodować wygenerowanie, w etapie rasteryzacji, fragmentów tych samych trójkątów otrzymanych podczas rozdrabniania płata, które będą rysowane na końcowym obrazie. Ale nie muszą one dostarczać żadnych danych potrzebnych tylko do obliczania kolorów pikseli, w tym wektora normalnego, współrzędnych cienia ani współrzędnych tekstury.

W poprzednich wersjach aplikacji mieliśmy zmienną jednolitą `BezTessLevel`, której wartość określa stopień rozdrobnienia płatów Béziera; zmienna ta należała do domyślnego bloku zmiennych jednolitych programu rysującego płaty Béziera. Ponieważ teraz będziemy mieć dwa różne programy, które mają w identyczny sposób rozdrabniać płaty, zmienną sterującą poziomem rozdrobnienia przeniesiemy do bloku `BezPatch`, gdzie również umieścimy zmienną `BezNormals`

[9]W zasadzie można użyć jednego programu (i na początku uruchamiania aplikacji tak zrobiłem). Ale: szader fragmentów programu dla końcowego obrazu wykonuje wiele niepotrzebnych w pierwszym etapie obliczeń. Można by je pominąć, umieszczając odpowiednie instrukcje w instrukcji warunkowej sterowanej za pomocą zmiennej jednolitej. Rzecz w tym, że blok wejściowy dla szadera fragmentów jest teraz bardzo długi, bo zawiera współrzędne cienia dla *wszystkich zadeklarowanych* źródeł światła (także tych wyłączonych w danej chwili), co oznacza ogromną ilość danych, które musiałyby być przetwarzane (interpolowane) w etapie rasteryzacji — a potem ignorowane. Dlatego wolimy mieć dwa programy, z których każdy wykonuje tylko potrzebne obliczenia i przekazuje między etapami tylko potrzebne dane.

[10]Na przykład dowolny kolor Forda T.

określającą, czy chcemy mieć końcowy obraz wykonany z użyciem wektorów normalnych płatów Béziera, czy też trójkątów przybliżających kawałki tych płatów. Blok `BezPatch` (listing 20.2) będzie dostępny dla wszystkich potrzebujących go programów. To rozwiązanie ułatwi też niezależne określanie poziomu rozdrobnienia płatów różnych obiektów (tj. czajnika i torusa).

Listing 20.2: Zmieniony blok zmiennych jednolitych `BezPatch`

GLSL

```
1: uniform BezPatch {
2:     int  dim, udeg, vdeg;
3:     int  stride_u, stride_v, stride_p, stride_q, nq;
4:     bool use_ind;
5:     vec4 Colour;
6:     int  TessLevel;
7:     bool BezNormals;
8:   } bezp;
```

Szader wierzchołków obu programów rysowania płatów Béziera jest taki sam — jest to szader pokazany na listingu 13.2. Szader sterowania rozdrabnianiem też jest wspólny dla obu programów; pokazałem go na listingu 20.3. Zmiana w porównaniu z szaderem z listingu 13.3 polega na wzięciu poziomu rozdrobnienia z bloku `BezPatch` zamiast z osobnej zmiennej jednolitej `BezTessLevel`.

Listing 20.3: Szader sterowania rozdrabnianiem

GLSL

```
1: #version 420 core
2:
3: layout (vertices=4) out;
4: in VertInstance { ... } In[];
5: out TCInstance { ... } Out[];
6: uniform BezPatch { ... } bezp;
7:
8: void main ( void )
9: {
10:   if ( gl_InvocationID == 0 ) {
11:     gl_TessLevelOuter[0] = gl_TessLevelOuter[1] =
12:     gl_TessLevelOuter[2] = gl_TessLevelOuter[3] = bezp.TessLevel;
13:     gl_TessLevelInner[0] = gl_TessLevelInner[1] = bezp.TessLevel;
14:     Out[gl_InvocationID].instance = In[gl_InvocationID].instance;
15:   }
16:   gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
17: } /*main*/
```

Szader rozdrabniania programu używanego do znajdowania reprezentacji obszaru cienia powstał przez uproszczenie szadera wykorzystywanego wcześniej. Na listingu 20.4 są umieszczone najważniejsze jego części. Szader ten oblicza tylko współrzędne leżącego na płacie wierzchołka (w układzie kostki standardowej) i nie ma żadnych dodatkowych danych wyjściowych poza strukturą `gl_PerVertex` (z polem `gl_Position`, któremu nadaje wartość).

Listing 20.4: Uproszczony szader rozdrabniania

GLSL

```glsl
1: #version 420 core
2: #define MAX_DEG 10
3: layout(quads,equal_spacing,ccw) in;
4: in TCInstance { ... } In[];
5: uniform CPoints { ... } cp;
6: uniform CPIndices { ... } cpi;
7: uniform BezPatch { ... } bezp;
8: uniform TransBlock { ... } trb;
9: int inst;
10:
11: void BCHorner2f ( int n, vec2 bcp[MAX_DEG+1], float t, out vec2 p )
12: { ... } /*BCHorner2f*/
13: void BPHorner2f ( float u, float v, out vec4 pos )
14: { ... } /*BPHorner2f*/
15:
16: void BCHorner3f ( int n, vec3 bcp[MAX_DEG+1], float t, out vec3 p )
17: {
18:   int   i, b;
19:   float s, d;
20:   vec3  q;
21:
22:   s = 1.0-t;  d = t;  b = n;
23:   q = bcp[0];
24:   for ( i = 1; i <= n; i++ ) {
25:     q = s*q + (b*d)*bcp[i];
26:     d *= t;  b = (b*(n-i))/(i+1);
27:   }
28:   p = q;
29: } /*BCHorner3f*/
30:
31: void BPHorner3f ( float u, float v, out vec4 pos )
32: {
33:   vec3 p[MAX_DEG+1], q[MAX_DEG+1], r;
34:   int  i, j, k, l, i0;
35:   .... /* początek bez zmian */
```

```
36:   for ( i = k = 0;  i <= bezp.udeg;  i++ ) {
37:     .... /* wybieranie punktów kontrolnych z tablicy bez zmian */
38:     BCHorner3f ( bezp.vdeg, p, v, q[i] );
39:   }
40:   BCHorner3f ( bezp.udeg, q, u, r );
41:   pos = vec4 ( r, 1.0 );
42: } /*BPHorner3f*/
43:
44: void BCHorner4f ( int n, vec4 bcp[MAX_DEG+1], float t, out vec4 p )
45: { ... } /*BCHorner4f*/
46: void BPHorner4f ( float u, float v, out vec4 pos )
47: { ... } /*BPHorner4f*/
48:
49: void main ( void )
50: {
51:   vec4 pos;
52:
53:   inst = In[0].instance;
54:   pos = vec4 ( 0.0 );
55:   switch ( bezp.dim ) {
56: case 2: BPHorner2f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
57: case 3: BPHorner3f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
58: case 4: BPHorner4f ( gl_TessCoord.x, gl_TessCoord.y, pos );  break;
59:   }
60:   gl_Position = trb.vpm * (trb.mm * pos);
61: } /*main*/
```

Zmiany w procedurach BCHorner2f, BPHorner2f, BCHorner4f i BPHorner4f są podobne do tych wprowadzonych w pokazanych na listingu procedurach dla trójwymiarowych płatów wielomianowych. W szczególności zostały z nich usunięte wszystkie parametry i zmienne używane w obliczeniach pochodnych cząstkowych i wektorów normalnych oraz instrukcje wykonujące te obliczenia[11].

W programie używanym do znalezienia obszaru cienia można zrezygnować z szadera geometrii, ponieważ w razie jego braku w programie trójkąty wygenerowane przez szader rozdrabniania trafiają od razu do etapu obcinania (zobacz rys. 1.1). Szader fragmentów może mieć pustą treść, ale jest potrzebny,

[11] W związku z uproszczeniem algorytmu wyznaczania punktu na krzywej Béziera (który już nie oblicza wektora pochodnej parametryzacji), oba szadery rozdrabniania płatów dostarczą takie same wyniki z dokładnością do innych błędów zaokrągleń. To są małe błędy, a ponadto dalsze przekształcenia prowadzące do różnego rzutowania podczas wyznaczania obszaru cienia i podczas wykonywania końcowego obrazu wprowadzą dodatkowe, różne błędy zaokrągleń. To nie szkodzi, te błędy zostaną skompensowane w sposób opisany dalej.

aby odpowiednie informacje trafiły do bufora głębokości — jego obecność powoduje przystąpienie do pracy ostatniego etapu potoku przetwarzania grafiki, który wpisuje informacje do tego bufora.

Teraz opiszę zmiany szaderów rozdrabniania, geometrii i fragmentów używanych do wykonania końcowego obrazu płatów Béziera; one powstają przez *dodanie nowych* rzeczy do szaderów z poprzedniej wersji aplikacji. Na listingu 20.5 jest pokazany blok wyjściowy szadera fragmentów (który jest blokiem wejściowym szadera geometrii). Porównując go z blokiem na listingu 19.1, zauważamy nowe pole, tablicę `ShadowPos`; każdy element tej tablicy odpowiada jednemu źródłu światła. Szader rozdrabniania ma mu przypisać współrzędne jednorodne wierzchołka w opisanym niżej układzie kostki jednostkowej określonym dla tego źródła; nazwiemy je współrzędnymi cienia.

Listing 20.5: Szader rozdrabniania płatów Béziera

GLSL
```
1: #version 420 core
2:
3: #define MAX_NLIGHTS 8
4:
5: uniform TCInstance { ... } In[];
6:
7: out GVertex {
8:     int  instance;
9:     vec4 Colour;
10:    vec3 Position;
11:    vec3 pu, pv, Normal;
12:    vec2 PatchCoord, TexCoord;
13:    vec4 ShadowPos[MAX_NLIGHTS];
14:  } Out;
15:
16: uniform CPoints { ... } cp;
17: uniform CPIndices { ... } cpi;
18: uniform BezPatch { ... } bezp;
19: uniform TransBlock { ... } trb;
20:
21: struct LSPar {
22:     vec4 ambient;
23:     vec4 direct;
24:     vec4 position;
25:     vec3 attenuation;
26:     mat4 shadow_vpm;
27:  };
```

```
28:
29: uniform LSBlock {
30:     uint  nls;              /* liczba źródeł światła */
31:     uint  mask;             /* maska włączonych źródeł */
32:     uint  shmask;           /* maska tekstur cienia */
33:     LSPar ls[MAX_NLIGHTS];  /* poszczególne zródła światła */
34:   } light;
35:
36: .... /* procedury obliczające punkty i wektory normalne płatów Béziera */
37: .... /* jak w aplikacji 2F */
38:
39: void main ( void )
40: {
41:   vec4 pos, wpos, pu, pv, nv;
42:   uint l, mask;
43:
44:   .... /* początek bez zmian */
45:   Out.PatchCoord = gl_TessCoord.xy;
46:   wpos = trb.mm * pos;
47:   for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
48:     if ( (light.mask & mask) != 0 )
49:       Out.ShadowPos[l] = light.ls[l].shadow_vpm * wpos;
50:   gl_Position = trb.vpm * wpos;
51:   Out.Position = wpos.xyz;
52:   .... /* dalsze instrukcje bez zmian */
53:   if ( !bezp.BezNormals || dot ( nv, nv ) < 1.0e-10 ) ....
54:   ....
55: } /*main*/
```

Szader rozdrabniania musi mieć teraz dostęp do bloków zmiennych jednolitych opisujących źródła światła (tego, na którego podstawie szader fragmentów oblicza oświetlenie), ponieważ odwołuje się do nowego pola `shadow_vpm` struktury opisującej źródła światła. W polu tym jest podana macierz będąca iloczynem trzech macierzy, BP_lV_l. Macierze P_l i V_l opisują przejścia od układu świata do układu obserwatora związanego z l-tym źródłem światła i dalej do układu kostki standardowej. Natomiast macierz

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

opisuje przekształcenie, które kostkę *standardową* $[-1, 1]^3$ odwzorowuje na kostkę *jednostkową* $[0, 1]^3$. Przekształcenie to jest potrzebne dlatego, że ewaluatory

Listing 20.6: Szader geometrii dla obrazu płatów Béziera z cieniami

GLSL

```
1: #version 420
2:
3: layout(triangles) in;
4: layout(triangle_strip,max_vertices=3) out;
5: in GVertex { ... } In[]
6: out NVertex { ... } Out;
7: struct LSPar { ... };
8: uniform LSBlock { ... } light;
9: uniform BezPatch { ... } bezp;
10:
11: void main ( void )
12: {
13:   uint i, l, mask;
14:   vec3 v1, v2, nv;
15:
16:   v1 = In[1].Position - In[0].Position;
17:   v2 = In[2].Position - In[0].Position;
18:   nv = normalize ( cross ( v1, v2 ) );
19:   for ( i = 0; i < 3; i++ ) {
20:     gl_Position = gl_in[i].gl_Position;
21:     Out.Position = In[i].Position;
22:     if ( !bezp.BezNormals || dot ( nv, nv ) < 1.0e-10 ) {
23:       Out.pu = In[i].pu - dot ( In[i].pu, nv )*nv;
24:       Out.pv = In[i].pv - dot ( In[i].pv, nv )*nv;
25:       Out.Normal = nv;
26:     }
27:     else {
28:       Out.pu = In[i].pu;
29:       Out.pv = In[i].pv;
30:       Out.Normal = In[i].Normal;
31:     }
32:     Out.Colour = In[i].Colour;
33:     Out.PatchCoord = In[i].PatchCoord;
34:     Out.TexCoord = In[i].TexCoord;
35:     Out.instance = In[i].instance;
36:     for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
37:       if ( (light.mask & mask) != 0 )
38:         Out.ShadowPos[l] = In[i].ShadowPos[l];
39:     EmitVertex ();
40:   }
41:   EndPrimitive ();
42: } /*main*/
```

tekstur (zgodnie z ogólną konwencją przyjętą w OpenGL-u) spodziewają się współrzędnych tekstury z przedziału $[0,1]$. Instrukcja w linii 46 oblicza współrzędne wierzchołka w układzie świata (tj. dokonuje przejścia od układu modelu do układu świata). W liniach 47–49 obliczane są współrzędne cienia wierzchołka dla poszczególnych (włączonych) źródeł światła. Reszta instrukcji szadera rozdrabniania jest bez zmian, z wyjątkiem odwołania do pola `BezNormals` z bloku `BezPatch` zamiast wcześniej używanej zmiennej jednolitej, której już nie ma.

Nowy szader geometrii jest pokazany na listingu 20.6. Przypomnijmy, że jego zadaniem jest skopiowanie danych z wejścia na wyjście i ewentualne obliczenie (i wyprowadzenie) wektora normalnego płaszczyzny trójkąta, jeśli aplikacja nadała polu `BezNormals` bloku `BezPatch` wartość `false` lub jeśli płat ma osobliwość, której skutkiem było obliczenie przez szader rozdrabniania wektora zerowego.

Bloki wejściowy `GVertex` i wyjściowy `NVertex` mają identyczne pola jak blok wyjściowy `GVertex` na listingu 20.5 (prawie, bo pole `NVertex.instance` jest typu `float`). Szader geometrii odwołuje się też do bloków zmiennych jednolitych `BezPatch` i `LSBlock`, których deklaracje zostały dodane do jego treści. W dodatku do dotychczasowych zadań, w pętli w liniach 36–38 szader kopiuje współrzędne cienia wierzchołka dla włączonych źródeł światła z bloku wejściowego do wyjściowego.

Etap rasteryzacji dokonuje interpolacji współrzędnych wierzchołków (w układzie kostki standardowej), a także wszystkich danych dodatkowych obecnych w bloku wyjściowym szadera geometrii — współrzędnych koloru, współrzędnych położenia w układzie świata, pochodnych cząstkowych, wektora normalnego, parametrów płata, współrzędnych tekstury i współrzędnych cienia dla *wszyskich* (nie tylko włączonych) źródeł światła. Zobaczmy teraz, jak z tej informacji korzysta szader fragmentów dla końcowego obrazu (listing 20.7).

W liniach 3–9 są wymienione bloki interfejsu (tj. wejścia/wyjścia i zmiennych jednolitych) szadera, przy czym bloki o niezmienionej budowie (w porównaniu z szaderem aplikacji drugiej F) przerobiłem na szaro. W liniach 11 i 12 są zadeklarowane zmienne jednolite (ewaluatory tekstury), do których zostanie przywiązana tekstura koloru farby (opisująca obrazki na czajniku) i tekstury obszarów cienia dla poszczególnych źródeł światła. Zwróćmy uwagę na podane (w kwalifikatorach `layout`) numery punktów dowiązania tekstur — dla zmiennej `tex` jest to numer `0` (w OpenGL-u identyfikowany przez stałą symboliczną `GL_TEXTURE0`), a dla elementów tablicy `shtex` to są numery od 2 do 9, jako że

Listing 20.7: Fragmenty szadera fragmentów dla obrazów z cieniami

GLSL

```
1: #define MAX_NLIGHTS 8
2:
3: in NVertex { ... } In;
4: out vec4 out_Colour;
5: uniform TransBlock { ... } trb;
6: struct LSPar { ... };
7: uniform LSBlock { ... } light;
8: struct Material { ... };
9: uniform MatBlock { ... } mat;
10:
11: layout (binding = 0) uniform sampler2D tex;
12: layout (binding = 2) uniform sampler2DShadow shtex[MAX_NLIGHTS];
13:
14: float IsEnlighted ( uint l )
15: {
16:   return textureProj ( shtex[l], In.ShadowPos[l] );
17: } /*IsEnlighted*/
18:
19: subroutine (LightingProc) void LambertLighting ( void )
20: {
21:   vec3  lv, vv;
22:   float d, e, s, dist;
23:   uint  i, mask;
24:
25:   vv = posDifference ( trb.eyepos, In.Position, dist );
26:   e = dot ( vv, nv );
27:   out_Colour = vec4(0.0);
28:   for ( i = 0, mask = 0x00000001;  i < light.nls;  i++, mask <<= 1 )
29:     if ( (light.mask & mask) != 0 ) {
30:       out_Colour += light.ls[i].ambient * mm.ambref;
31:       s = ((light.shmask & mask) != 0) ? IsEnlighted ( i ) : 1.0;
32:       if ( s > 0.0 ) {
33:         lv = posDifference ( light.ls[i].position, In.Position, dist );
34:         d = dot ( lv, normal );
35:         if ( e > 0.0 ) {
36:           if ( dot ( lv, nv ) > 0.0 ) {
37:             if ( light.ls[i].position.w != 0.0 )
38:               d *= attFactor ( light.ls[i].attenuation, dist );
39:             out_Colour += (s*d * light.ls[i].direct) * mm.dirref;
40:           }
41:         }
42:         else {
```

```
43:         if ( dot ( lv, nv ) < 0.0 ) {
44:           if ( light.ls[i].position.w != 0.0 )
45:             d *= attFactor ( light.ls[i].attenuation, dist );
46:           out_Colour -= (s*d * light.ls[i].direct) * mm.dirref;
47:         }
48:       }
49:     }
50:   }
51:   out_Colour = vec4 ( clamp ( out_Colour.rgb, 0.0, 1.0 ), 1.0 );
52: } /*LambertLighting*/
```

ustalona w aplikacji maksymalna liczba źródeł światła to 8. Punktu dowiązania o numerze 1 użyjemy dla tekstury nakładanej na lustro — w tej aplikacji nie może to być numer 0 (tzn. ten sam numer, którego używamy dla tekstury nałożonej na czajnik), bo całość by nie zadziałała poprawnie. Typ ewaluatora `sampler2DShadow` jest przeznaczony specjalnie do badania (w teksturze przechowującej bufor głębokości), czy dany punkt leży w obszarze cienia.

Funkcja `IsEnlighted` w liniach 14–17 dokonuje badania, czy dany punkt należy do obszaru cienia. Parametr tej funkcji jest numerem źródła światła. Obliczenie wygląda tak: pierwszym parametrem funkcji wbudowanej `textureProj` jest odpowiednia tekstura, a drugi parametr to wektor (jednorodnych) współrzędnych cienia przetwarzanego punktu. Na jego podstawie są obliczane współrzędne kartezjańskie x, y, z (przez podzielenie pierwszych trzech współrzędnych jednorodnych przez czwartą), które mają być liczbami z przedziału $[0,1]$. Również w tekselach są przechowywane liczby z tego przedziału, przy czym 0 odpowiada punktowi na przedniej ścianie kostki, a 1 punktowi na tylnej ścianie. Funkcja `textureProj` (dla ewaluatora, którego parametrom aplikacja nadała odpowiednie wartości, opis będzie dalej) porównuje współrzędną z z wartością tekstury[12] w punkcie (x,y) i zwraca wartość 1 jeśli współrzędna z jest mniejsza, oraz 0 jeśli jest większa niż wartość tekstury w tym punkcie.

W liniach 19–52 jest pokazana zmodyfikowana procedura realizująca model oświetlenia Lamberta; analogiczne zmiany trzeba też wprowadzić w procedurze z modelem Blinna–Phonga. W linii 30 do światła odbijanego przez piksel jest dodawany składnik odpowiadający światłu pochodzącemu z danego źródła, rozproszonego w otoczeniu (czyli oświetlającego punkty także w obszarze cienia). W linii 31 szader bada, czy dla danego źródła światła tekstura obszaru cienia została utworzona; jeśli tak, to zmiennej `s` przypisywana jest wartość funkcji `IsEnlighted`, a w przeciwnym razie wartość 1 — wtedy z założenia punkt jest

[12] Pamiętajmy: tekstura jest funkcją, tablica tekseli jest tylko reprezentacją tej funkcji.

bezpośrednio oświetlony. W liniach 38 i 45 czynnik s jest uwzględniony w składniku opisującym światło dochodzące bezpośrednio od źródła. Dlaczego mnożymy wyrażenia w liniach 31 i 38 przez zmienną s, która ma wartość 1? Bo później zmienimy funkcję IsEnlighted (która będzie mogła przyjmować wartości ułamkowe), aby poprawić wygląd cieni.

Dwóch programów szaderów potrzebujemy także do rysowania lustra. Program używany do znajdowania obszaru cienia nie powinien nakładać na lustro tekstury reprezentującej obraz odbitej w lustrze sceny. Program rysujący lustro na końcowym obrazie w zasadzie nie powinien nakładać cienia na odbity obraz, ale można rozważyć cienie na odwrotnej stronie lustra, jeśli chcemy rysować bardziej skomplikowane sceny[13]. Natomiast darowałem sobie rysowanie cieni na odcinkach siatek kontrolnych płatów. I już.

Tworzenie buforów ramki i tekstur dla obszarów cienia

Na listingu 20.8 są pokazane procedury, które tworzą, przygotowują do pracy i sprzątają bufory ramki i będące ich załącznikami tekstury obszarów cienia dla poszczególnych źródeł światła. Procedura ConstructShadowTxtFBO powinna być wywołana dla każdego źródła światła, dla którego aplikacja ma wyznaczać cienie. Instrukcje w liniach 7 i 8 alokują identyfikatory tekstury i bufora ramki. Jeśli to się powiedzie, co jest sprawdzane w liniach 9 i 10 z jednoczesnym zapamiętaniem tych identyfikatorów w odpowiednich polach struktury opisującej l-te źródło światła, wykonywany jest ciąg instrukcji nadających potrzebne własności tym obiektom.

W linii 11 uaktywniamy punkt dowiązania tekstury o numerze $l + 2$, a w linii 12 przywiązujemy nową teksturę do celu GL_TEXTURE_2D w tym punkcie. W linii 13 następuje określenie rodzaju danych przechowywanych w tekselach oraz szerokości i wysokości tekstury (makro SHADOW_MAP_SIZE rozwija się do stałej 1024, można je zmieniać), a ponadto jest alokowany blok pamięci GPU na tablicę tekseli. Trzeci parametr (internalFormat) o wartości GL_DEPTH_COMPONENT32 określa 32 bity na teksel. Wartość GL_DEPTH_COMPONENT szóstego parametru określa, że tekstura ma być buforem głębokości. Siódmy parametr określa, że teksele (których wartość jest głębokością) mają być liczbami zmiennopozycyjnymi.

Sposób filtrowania tekstury określony przez parametry procedury glTexParameteri w liniach 16 i 17 polega na interpolacji liniowej wartości tekseli.

[13] Można rozważyć światło odbite w lustrze, oświetlające obiekty i narysować cienie dla takiego światła. Napisanie takiego programu to już spora rafinada.

Listing 20.8: Procedury obsługi tekstur dla obszarów cienia

C

```
1: void ConstructShadowTxtFBO ( int l )
2: {
3:   GLuint fbo, txt;
4:
5:   if ( l < 0 || l >= MAX_NLIGHTS )
6:     return;
7:   glGenTextures ( 1, &txt );
8:   glGenFramebuffers ( 1, &fbo );
9:   if ( (light.ls[l].shadow_txt = txt) &&
10:       (light.ls[l].shadow_fbo = fbo) ) {
11:     glActiveTexture ( GL_TEXTURE2+l );
12:     glBindTexture ( GL_TEXTURE_2D, txt );
13:     glTexImage2D ( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
14:                    SHADOW_MAP_SIZE, SHADOW_MAP_SIZE, 0,
15:                    GL_DEPTH_COMPONENT, GL_FLOAT, NULL );
16:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
17:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
18:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
19:                       GL_COMPARE_REF_TO_TEXTURE );
20:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL );
21:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
22:     glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
23:     glBindTexture ( GL_TEXTURE_2D, 0 );
24:     glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
25:     glFramebufferTexture ( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
26:                            light.ls[l].shadow_txt, 0 );
27:     glDrawBuffer ( GL_NONE );
28:     light.shmask |= 0x01 << l;
29:     glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
30:     glBufferSubData ( GL_UNIFORM_BUFFER, lsbofs[2], sizeof(GLuint),
31:                       &light.shmask );
32:     glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
33:     ExitIfGLError ( "ConstructShadowTxtFBO" );
34:   }
35: } /*ConstructShadowTxtFBO*/
36:
37: void UpdateShadowMatrix ( int l )
38: {
39:   int                   ofs;
40:   GLfloat               lvpm[16], a[16];
41:   const GLfloat b[16] = {0.5,0.0,0.0,0.0,0.0,0.5,0.0,0.0,
42:                          0.0,0.0,0.5,0.0,0.5,0.5,0.5,1.0};
```

```
43:
44:   M4x4Multf ( a, light.ls[l].shadow_proj, light.ls[l].shadow_view );
45:   M4x4Multf ( lvpm, b, a );
46:   ofs = l*(lsbofs[8]-lsbofs[3]) + lsbofs[7];
47:   glBindBuffer ( GL_UNIFORM_BUFFER, lsbuf );
48:   glBufferSubData ( GL_UNIFORM_BUFFER, ofs, 16*sizeof(GLfloat), lvpm );
49:   ExitIfGLError ( "UpdateShadowMatrix" );
50: } /*UpdateShadowMatrix*/
51:
52: void BindShadowTxtFBO ( int l )
53: {
54:   if ( l < 0 || l >= MAX_NLIGHTS )
55:     return;
56:   if ( (light.ls[l].shadow_txt == 0) )
57:     return;
58:   SetVPMatrix ( light.ls[l].shadow_view, light.ls[l].shadow_proj,
59:                 light.ls[l].position );
60:   glBindFramebuffer ( GL_FRAMEBUFFER, light.ls[l].shadow_fbo );
61:   ExitIfGLError ( "BindShadowTxtFBO" );
62: } /*BindShadowTxtFBO*/
63:
64: void DestroyShadowFBO ( void )
65: {
66:   int l;
67:
68:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
69:   for ( l = 0; l < MAX_NLIGHTS; l++ ) {
70:     if ( light.ls[l].shadow_fbo != 0 )
71:       glDeleteFramebuffers ( 1, &light.ls[l].shadow_fbo );
72:     if ( light.ls[l].shadow_txt != 0 )
73:       glDeleteTextures ( 1, &light.ls[l].shadow_txt );
74:   }
75:   ExitIfGLError ( "DestroyShadowFBO" );
76: } /*DestroyShadowFBO*/
```

Dla badania, czy punkt jest w obszarze cienia, istotne są parametry podane w liniach 18–20; dla tekstury głębokości wartość `GL_COMPARE_REF_TO_TEXTURE` parametru `GL_TEXTURE_COMPARE_MODE` oznacza, że wbudowana w GLSL funkcja `texture` (lub `textureProj`) ma dokonać porównania współrzędnej z punktu w kostce jednostkowej z wartością tekstury i zwrócić wynik tego porównania. Funkcja porównująca `GL_LEQUAL` oznacza, że ma być zwrócona wartość 1, jeśli współrzędna z punktu jest mniejsza lub równa wartości tekstury, co oznacza, że punkt jest bezpośrednio oświetlony.

Wartości GL_CLAMP_TO_EDGE nadane parametrom sterującym obcinaniem w liniach 21 i 22 oznaczają, że współrzędne x, y punktu w układzie kostki jednostkowej, jeśli nie należą do przedziału $[0, 1]$, to mają być zastąpione przez 0 albo 1. Po określeniu wszystkich potrzebnych parametrów teksturę odczepiamy od celu, w linii 23.

W liniach 24–32 przygotowywany jest do pracy bufor ramki. W linii 24 jest on przyczepiany do celów GL_DRAW_FRAMEBUFFER i GL_READ_FRAMEBUFFER (jednocześnie do obu; to nie szkodzi). W liniach 25–26 dodajemy do niego załącznik — świeżo utworzoną teksturę, która będzie buforem głębokości. Nie zawracamy sobie (ani OpenGL-owi) głowy teksturą dla obrazu, który jest niepotrzebny, tylko wywołujemy procedurę glDrawBuffer z parametrem GL_NONE. W linii 26 ustawiamy bit maski, aby zaznaczyć, że dla l-tego źródła światła będziemy mieli teksturę obszaru cienia i nie zawahamy się jej użyć, po czym przesyłamy tę maskę do bloku zmiennych jednolitych zawierającego opisy źródeł światła. W linii 32 odczepiamy bufor ramki od obu celów, gdy przyjdzie właściwa pora, znów go przyczepimy.

Procedura UpdateShadowMatrix oblicza iloczyn macierzy BP_lV_l, który opisuje przejście od układu świata do układu kostki jednostkowej, a następnie przesyła ten iloczyn do pola shadow_vpm w strukturze LSPar opisującej l-te źródło światła w pamięci GPU. Współczynniki macierzy B są podane w tablicy b, zaś współczynniki macierzy V_l i P_l są brane z opisu źródła światła w pamięci CPU, z miejsca, w którym zostały zapamiętane przez wywołaną wcześniej procedurę SetupShadowTransformations z listingu 20.1. Przesunięcie obliczone w linii 46 wyznacza położenie pola shadow_vpm l-tego elementu tablicy ls w bloku zmiennych jednolitych LSBlock.

Procedura BindShadowTxtFBO, za pomocą procedury SetVPMatrix, przesyła macierze V_l i P_l oraz ich iloczyn i (niepotrzebne tu) położenie obserwatora (tj. źródła światła) do bloku zmiennych jednolitych TransBlock, a następnie przwiązuje odpowiedni bufor ramki do celu GL_DRAW_FRAMEBUFFER i już można rysować scenę (przy użyciu uproszczonych szaderów) w celu otrzymania reprezentacji obszaru cienia dla l-tego źródła światła.

Procedura DestroyShadowFBO przegląda tablicę opisów źródeł światła i likwiduje tekstury obszarów cienia i bufory ramki, których te tekstury były załącznikami. Wypada ją wywołać podczas końcowego sprzątania.

Zmiany w aplikacji

Na listingu 20.9 jest pokazana procedura LoadMyShaders, która kompiluje szadery, uzyskuje informacje dające dostęp do zmiennych jednolitych i tworzy niektóre bloki zmienych jednolitych potrzebne do działania aplikacji. Aplikacja korzysta z pięciu programów szaderów, które są zbudowane w sumie z dwunastu szaderów. Program, którego identyfikator zostaje (w linii 44) zapamiętany w zmiennej prog_id[0] służy do wykonywania końcowego obrazu płatów Béziera. Program prog_id[1] służy do rysowania siatek kontrolnych. Program prog_id[2] służy do rysowania lustra z nałożoną teksturą, która jest obrazem odbitej sceny. Program prog_id[3] jest uproszczonym programem rysowania płatów Béziera, potrzebnym do otrzymania tekstur cienia. Program prog_id[4] jest uproszczonym programem rysowania lustra. W liniach 52–54 identyfikatory skompilowanych i złączonych programów szaderów są zapamiętywane w dodatkowych tablicach; do tablicy

Listing 20.9: Procedura LoadMyShaders

C

```
1: GLuint shader_id[12], program_id[5], progid0[3], progid1[3];
2:
3: void LoadMyShaders ( void )
4: {
5:   static const char *filename[] =
6:     { "app2g0.glsl.vert", "app2g0.glsl.tesc", "app2g0.glsl.tese",
7:       "app2g0.glsl.geom", "app2g0.glsl.frag",
8:       "app2g1.glsl.vert", "app2g1.glsl.frag",
9:       "app2g2.glsl.vert", "app2g2.glsl.frag",
10:      "app2g3.glsl.tese","app2g3.glsl.frag", "app2g4.glsl.vert" };
11:  static const GLuint shtype[] =
12:    { GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER,
13:      GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER,
14:      GL_VERTEX_SHADER, GL_FRAGMENT_SHADER,
15:      GL_VERTEX_SHADER, GL_FRAGMENT_SHADER,
16:      GL_TESS_EVALUATION_SHADER, GL_FRAGMENT_SHADER, GL_VERTEX_SHADER };
17:  static const GLchar *UTBNames[] = { ... };
18:  static const GLchar *ULSNames[] =
19:    { "LSBlock", "LSBlock.nls", "LSBlock.mask", "LSBlock.shmask",
20:      "LSBlock.ls[0].ambient", "LSBlock.ls[0].direct",
21:      "LSBlock.ls[0].position", "LSBlock.ls[0].attenuation",
22:      "LSBlock.ls[0].shadow_vpm", "LSBlock.ls[1].ambient" };
23:  static const GLchar *UCPNames[] = { ... };
24:  static const GLchar *UCPINames[] = { ... };
25:  static const GLchar *UBezPatchNames[] =
26:    { "BezPatch", "BezPatch.dim", "BezPatch.udeg", "BezPatch.vdeg",
```

```
27:      "BezPatch.stride_u", "BezPatch.stride_v",
28:      "BezPatch.stride_p", "BezPatch.stride_q", "BezPatch.nq",
29:      "BezPatch.use_ind", "BezPatch.Colour",
30:      "BezPatch.TessLevel", "BezPatch.BezNormals" };
31:  static const GLchar *UMatNames[] = { ... }
32:  static const GLchar *UTexCoordNames[] = { ... };
33:  static const GLchar LightProcName[] = "Lighting";
34:  static const GLchar LambertProcName[] = "LambertLighting";
35:  static const GLchar BlinnPhongProcName[] = "BlinnPhongLighting";
36:  static const GLchar ColourSourceName[] = "ColourSource";
37:  static const GLchar NormalSourceName[] = "NormalSource";
38:
39:  GLint  i;
40:  GLuint shid[5];
41:
42:  for ( i = 0; i < 12; i++ )
43:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
44:  program_id[0] = LinkShaderProgram ( 5, shader_id );
45:  program_id[1] = LinkShaderProgram ( 2, &shader_id[5] );
46:  program_id[2] = LinkShaderProgram ( 2, &shader_id[7] );
47:  shid[0] = shader_id[0];  shid[1] = shader_id[1];
48:  shid[2] = shader_id[9];  shid[3] = shader_id[10];
49:  program_id[3] = LinkShaderProgram ( 4, shid );
50:  shid[0] = shader_id[11];  shid[1] = shader_id[10];
51:  program_id[4] = LinkShaderProgram ( 2, shid );
52:  progid0[0] = program_id[3];  progid0[1] = program_id[1];
53:  progid0[2] = program_id[4];  progid1[0] = program_id[0];
54:  progid1[1] = program_id[1];  progid1[2] = program_id[2];
55:  GetAccessToUniformBlock ( program_id[0], 6, &UTBNames[0],
56:                            &trbi, &trbsize, trbofs, &trbbp );
57:  GetAccessToUniformBlock ( program_id[0], 9, &ULSNames[0],
58:                            &lsbi, &lsbsize, lsbofs, &lsbbp );
59:  ....  /* tu instrukcje niezmienione */
60:  GetAccessToUniformBlock ( program_id[0], 12, &UBezPatchNames[0],
61:                            &bezpbi, &bezpbsize, bezpbofs, &bezpbbp );
62:  ....  /* tu instrukcje niezmienione */
63:  AttachUniformBlockToBP ( program_id[2], UTBNames[0], trbbp );
64:  AttachUniformBlockToBP ( program_id[3], UTBNames[0], trbbp );
65:  AttachUniformBlockToBP ( program_id[3], UCPNames[0], cpbbp );
66:  AttachUniformBlockToBP ( program_id[3], UCPINames[0], cpibbp );
67:  AttachUniformBlockToBP ( program_id[3], UBezPatchNames[0], bezpbbp );
68:  AttachUniformBlockToBP ( program_id[4], UTBNames[0], trbbp );
69:  ExitIfGLError ( "LoadMyShaders" );
70: } /*LoadMyShaders*/
```

progid0 trafiają identyfikatory programów uproszczonych, zaś w tablicy progid1 są umieszczane identyfikatory programów do rysowania końcowego obrazu.

Opisane dalej procedury rysowania sceny dostają jedną z tych tablic jako parametr i wywołując procedurę glUseProgram, będą podawać identyfikator programu z tablicy będącej parametrem. W liniach 61–65 nowe programy szaderów zostają połączone z punktami dowiązania dla bloków zmiennych jednolitych TransBlock, CPoints, CPIndices i BezPatch, używanych też przez pozostałe programy.

Procedura InitLights jest pokazana na listingu 20.10. Po przesłaniu parametrów opisujących źródło światła o numerze 0 i „włączeniu” go, procedura tworzy bufor ramki i teksturę cienia oraz konstruuje macierze przekształceń dla rysowania sceny widzianej z kierunku padania światła i przesyła iloczyn BP_0V_0 do pamięci GPU. Dodając kolejne źródła światła wokół sceny, należałoby zrobić dla nich to samo. Jeśli położenia źródeł światła miałyby być animowane, to po każdej zmianie położenia źródła światła trzeba wywołać procedury SetupShadowTxtTransformations i UpdateShadowMatrix.

Listing 20.10: Procedura InitLights

```c
1: void InitLights ( void )
2: {
3:   GLfloat amb0[4] = { 0.2, 0.2, 0.3, 1.0 };
4:   GLfloat dif0[4] = { 0.8, 0.8, 0.8, 1.0 };
5:   GLfloat pos0[4] = { -0.2, 1.0, 1.0, 0.0 };
6:   GLfloat atn0[3] = { 1.0, 0.0, 0.0 };
7:   GLfloat csc[3]  = { 0.0, 0.0, 0.0 };  /* środek kuli otaczającej scenę */
8:
9:   memset ( &light, 0, sizeof(LightBl) );
10:  SetLightAmbient ( 0, amb0 );
11:  SetLightDiffuse ( 0, dif0 );
12:  SetLightPosition ( 0, pos0 );
13:  SetLightAttenuation ( 0, atn0 );
14:  SetLightOnOff ( 0, 1 );
15:  ConstructShadowTxtFBO ( 0 );
16:  SetupShadowTxtTransformations ( 0, csc, 2.2 );  /* R == 2.2 */
17:  UpdateShadowMatrix ( 0 );
18: } /*InitLights*/
```

Do procedur ConstructMyTeapot i ConstructMyTorus trzeba dopisać wywołania procedury SetBezierPatchOptions, które nadają odpowiednie wartości polom TessLevel i BezNormals w blokach zmiennych jednolitych BezPatch, czego tu nie

uszczegółowiam. Na listingu 20.11 jest pokazana procedura rysowania czajnika; podobne (choć prostsze, bo torus nie ma nakładanej tekstury) zmiany trzeba wprowadzić w procedurze `DrawMyTorus`. Jeśli parametr `final` ma wartość 1, to ma być wykonany obraz końcowy; wtedy (w liniach 11–12) przypisujemy wartość zmiennej wskazującej procedurę z odpowiednim modelem oświetlenia[14] i (jeśli trzeba) przyczepiamy teksturę, która ma być nałożona na czajnik do celu `GL_TEXTURE_2D` w punkcie dowiązania tekstury `GL_TEXTURE0`. Jeśli parametr `final` ma wartość 0, to rysujemy czajnik bez tych czynności wstępnych (parametr `program` jest wtedy identyfikatorem uproszczonego programu szaderów).

Listing 20.11: Procedura `DrawMyTeapot`

```c
void DrawMyTeapot ( char final, GLuint program )
{
  if ( skeleton ) {
    glLineWidth ( 2.0 );
    glPolygonMode ( GL_FRONT_AND_BACK, GL_LINE );
  }
  else
    glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
  glUseProgram ( program );
  if ( final ) {
    glUniformSubroutinesuiv ( GL_FRAGMENT_SHADER, 1,
                              (GLuint*)&LightProcInd );
    glUniform1i ( ucs_loc, colour_source );
    glUniform1i ( uns_loc, normal_source );
    ChooseMaterial ( 0 );
    if ( colour_source == 2 ) {
      glBindBufferBase ( GL_UNIFORM_BUFFER, txcbp, myteapot->buf[3] );
      glActiveTexture ( GL_TEXTURE0 );
      glBindTexture ( GL_TEXTURE_2D, mytexture );
      DrawBezierPatches ( myteapot );
      glBindTexture ( GL_TEXTURE_2D, 0 );
      return;
    }
  }
  DrawBezierPatches ( myteapot );
} /*DrawMyTeapot*/
```

Zobaczmy wreszcie procedury rysowania całej sceny; mamy je na listingu 20.12.

[14]Przypominam, że to trzeba zrobić zawsze po wywołaniu procedury `glUseProgram` z programem szaderów, który zawiera zmienną wskaźnikową procedury.

Procedura DrawSceneToWindow, wywoływana w celu wykonania obrazu w oknie na ekranie, najpierw (w linii 52) wywołuje procedurę DrawSceneToShadows, której zadaniem jest wyznaczenie reprezentacji obszarów cienia dla wszystkich

Listing 20.12: Procedury rysowania sceny z cieniami

C

```c
1: void DrawScene ( char final, GLuint programs[3] )
2: {
3:   SetModelMatrix ( teapot_mmatrix, teapot_mmti );
4:   DrawMyTeapot ( final, programs[0] );
5:   if ( cnet )
6:     DrawMyTeapotCNet ( programs[1] );
7:   SetModelMatrix ( torus_mmatrix, torus_mmti );
8:   DrawMyTorus ( final, programs[0] );
9:   if ( cnet )
10:    DrawMyTorusCNet ( programs[1] );
11: } /*DrawScene*/
12:
13: void DrawSceneToMirror ( void )
14: {
15:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, mirror_fbo );
16:   glViewport ( 0, 0, MIRRORTXT_W, MIRRORTXT_H );
17:   SetVPMatrix ( trans.mvm, trans.mpm, trans.reyepos );
18:   glClearColor ( 0.95, 0.95, 0.95, 1.0 );
19:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
20:   DrawScene ( 1, progid1 );
21:   glFlush ();
22: } /*DrawSceneToMirror*/
23:
24: void DrawSceneToShadows ( void )
25: {
26:   int    l;
27:   GLuint mask;
28:
29:   glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );
30:   glEnable ( GL_POLYGON_OFFSET_FILL );
31:   glPolygonOffset ( 2.0f, 4.0f );
32:   for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
33:     if ( light.shmask & mask ) {
34:       BindShadowTxtFBO ( l );
35:       glClear ( GL_DEPTH_BUFFER_BIT );
36:       DrawScene ( 0, progid0 );
37:       glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
38:       DrawMirror ( 0, progid0[2] );
```

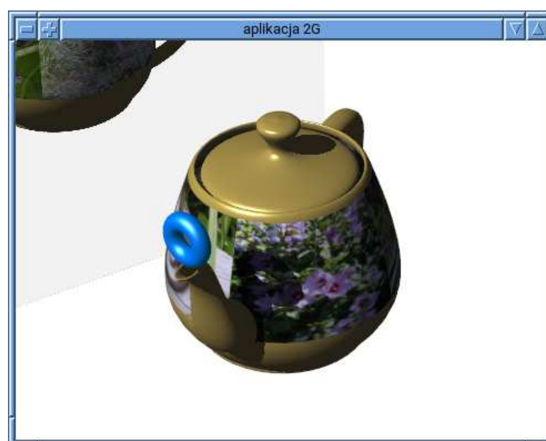
```
39:     }
40:   glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
41:   glDisable ( GL_POLYGON_OFFSET_FILL );
42:   for ( l = 0, mask = 0x00000001;  l < light.nls;  l++, mask <<= 1 )
43:     if ( light.shmask & mask ) {
44:       glActiveTexture ( GL_TEXTURE2+l );
45:       glBindTexture ( GL_TEXTURE_2D, light.ls[l].shadow_txt );
46:     }
47: } /*DrawSceneToShadows*/
48:
49: void DrawSceneToWindow ( void )
50: {
51:   glEnable ( GL_DEPTH_TEST );
52:   DrawSceneToShadows ();
53:   DrawSceneToMirror ();
54:   glBindFramebuffer ( GL_DRAW_FRAMEBUFFER, 0 );
55:   glViewport ( 0, 0, win_width, win_height );
56:   SetVPMatrix ( trans.wvm, trans.wpm, trans.eyepos );
57:   glClearColor ( 1.0, 1.0, 1.0, 1.0 );
58:   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
59:   glPolygonMode ( GL_FRONT_AND_BACK, GL_FILL );
60:   DrawMirror ( 1, progid1[2] );
61:   DrawScene ( 1, progid1 );
62: } /*DrawSceneToWindow*/
```

włączonych źródeł światła. Potem następuje rysowanie sceny (bez lustra) w teksturze, która zostanie na lustro nałożona; zauważmy, że cienie muszą być widoczne także na obrazie w lustrze. W linii 54 przywiązujemy do celu domyślny bufor ramki, a więc OpenGL będzie odtąd rysował w oknie. Po ustawieniu klatki i macierzy określających rzutowanie, rysujemy lustro i pozostałe obiekty sceny, podając jako parametry procedur identyfikatory odpowiednich programów szaderów.

Procedura `DrawSceneToShadows` w pętli rysuje scenę w rzutach określonych dla włączonych źródeł światła; działanie pętli i instrukcji w niej zawartych nie wymaga komentarza. Ale jest jeszcze jeden szczegół: przed rysowaniem wywołujemy procedurę `glEnable` z parametrem `GL_POLYGON_OFFSET_FILL` i procedurę `glPolygonOffset` z parametrami, które opisują korektę głębokości zrasteryzowanych fragmentów. Ma ona na celu „odsunięcie” powierzchni od obserwatora, co prowadzi do wstawienia do bufora głębokości nieco większych liczb. Bez tej korekty punkty na powierzchni, rysowane na końcowym obrazie, mogłyby „same siebie zasłaniać” od światła wskutek błędów zaokrągleń

i ograniczonej dokładności reprezentacji obszaru cienia. Korekta kompensuje też różnice błędów zaokrągleń w obliczeniach punktów powierzchni i rzutowaniu podczas wyznaczania obszarów cienia i podczas wykonywania końcowego obrazu. Po otrzymaniu wszystkich reprezentacji obszarów cienia w teksturach, korektę wyłączamy za pomocą procedury `glDisable`. W pętli w liniach 41–46 tekstury cieni przywiązujemy do celów `GL_TEXTURE_2D` w kolejnych punktach dowiązania, zaczynając od punktu o numerze 2 (identyfikowanego przez makro `GL_TEXTURE2`).

Zmiany w procedurach `DrawScene` i `DrawSceneToMirror` polegają na użyciu programów szaderów, których identyfikatory są podane w parametrach, zamiast brania ich z globalnej tablicy `program_id`.



Rysunek 20.2: Okno aplikacji drugiej G

Ćwiczenia

1. Wykonaj eksperymenty polegające na zmienianiu parametrów procedury `glPolygonOffset` (w procedurze `DrawSceneToShadows`) i obserwowaniu skutków tych zmian.

2. Rozszerz aplikację o animowanie źródła (lub źródeł) światła, polegającą na przykład na obracaniu położenia źródła światła wokół osi x z prędkością jednego obrotu na kilka sekund. Po zmianie położenia l-tego źródła światła trzeba uaktualnić macierze V_l i P_l.

21. Aplikacja druga H

Dokonamy bardziej skomplikowanej animacji. Utworzymy mianowicie łańcuch kinematyczny, który ułatwi systematyczny opis ruchu obiektów względem siebie i w szczególności umożliwi odkształcanie obiektów. Aby usprawnić nalewanie herbaty, będziemy odkształcać dziobek czajnika. Stanie się to okazją do zapoznania się z szaderami obliczeniowymi.

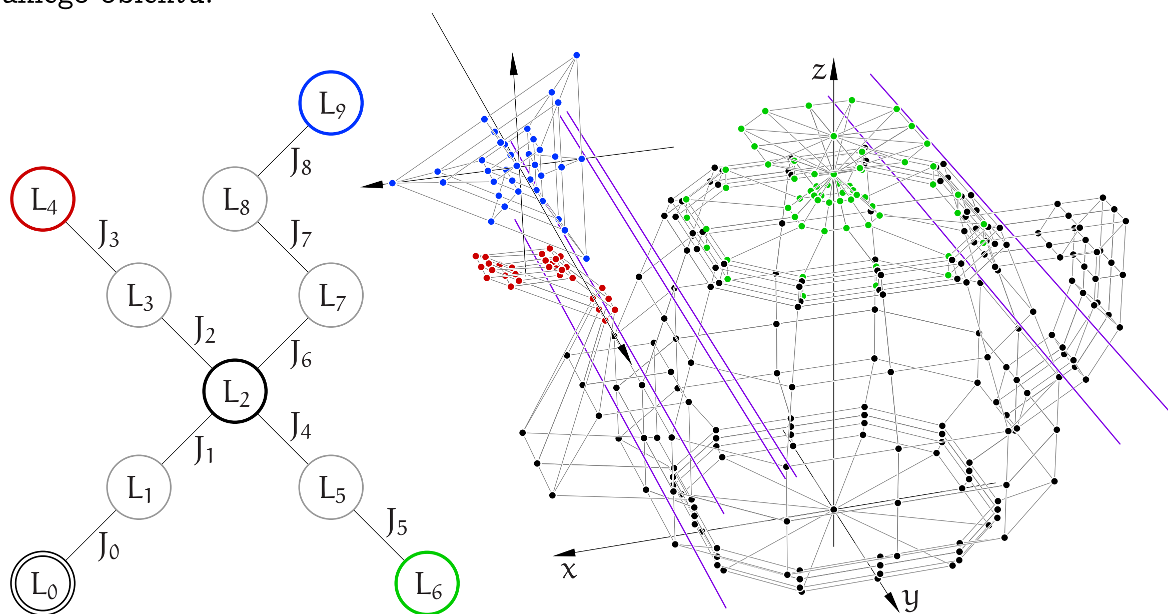
Łańcuchy kinematyczne

Bryła sztywna w przestrzeni trójwymiarowej ma 6 stopni swobody, tj. do opisu jej położenia trzeba użyć sześciu liczb (np. trzech współrzędnych kartezjańskich ustalonego punktu bryły i trzech liczb opisujących jej obrót). Jeśli dwie bryły są połączone w sposób ograniczający ich ruch względem siebie, to tworzą parę kinematyczną. Jeśli np. połączenie jest zawiasem, to układ złożony z takich brył ma 7 stopni swobody — o ile pierwszą bryłę można przesuwać i obracać dowolnie, położenie drugiej bryły po ustaleniu położenia pierwszej bryły i kąta obrotu drugiej wokół osi zawiasu jest określone jednoznacznie. Tak więc para utworzona przy użyciu zawiasu odebrała układowi 5 stopni swobody.

Łańcuch kinematyczny (*kinematic linkage*) *w mechanice* jest to układ brył sztywnych zwanych członami (*links*), których ruch względem siebie jest ograniczony przez pary kinematyczne (*joints*). Z każdą parą kinematyczną jest związany co najmniej jeden parametr artykulacji (*articulation parameter*), który opisuje np. kąt obrotu zawiasu lub przesunięcie (np. pręta w rurze). Pary z jednym parametrem artykulacji są zwane prostymi, a pary mające dwa lub więcej parametrów (takie jak przegub kulowy lub para wymuszająca pozostawanie obiektu na płaszczyźnie) są złożone. Łańcuch kinematyczny może być opisany za pomocą grafu, którego wierzchołkami są człony, a krawędziami — pary kinematyczne. Łańcuch jest otwarty, jeśli ten graf nie zawiera cyklu, a w przeciwnym razie mamy łańcuch zamknięty. Zauważmy, że parametry artykulacji par kinematycznych tworzących cykl nie są niezależne. Z jednej strony, poszczególne pary mogą odbierać te same stopnie swobody (na przykład jeśli to są zawiasy, których osie są równoległe). Z drugiej strony ustalenie parametrów dla pewnych par może jednoznacznie określić wartości parametrów wszystkich pozostałych par w cyklu, przy czym aby je znaleźć należy na ogół rozwiązać układ równań nieliniowych. Taki układ może być sprzeczny (tj. nie mieć rozwiązań), może też zdarzyć się kolizja członów. Tymi problemami nie będziemy się tu zajmować, ograniczając się do modelowania łańcuchów otwartych (których grafy są drzewami) i nie zwracając uwagi na kolizje.

Artykulacja łańcucha kinematycznego polega na znalezieniu położeń wszystkich członów łańcucha po ustaleniu wartości parametrów artykulacji i położenia pewnego członu łańcucha.

W programie człony łańcucha kinematycznego będą układami współrzędnych kartezjańskich w przestrzeni. W poszczególnych układach będą określone obiekty, a dokładniej wierzchołki lub punkty kontrolne figur rysowanej sceny. Mogą być człony, z którymi nie zwiążemy żadnych obiektów, mogą być też człony z wieloma przywiązanymi obiektami. Co więcej, różne części jednego obiektu mogą być związane z różnymi członami łańcucha. Artykulacja spowoduje odkształcanie takiego obiektu.



Rysunek 21.1: Łańcuch kinematyczny czajnika z torusem

Na rysunku 21.1 jest pokazany graf łańcucha tworzonego przez opisaną w tym rozdziale aplikację. Łańcuch ma 10 członów (oznaczonych symbolami $L_0, \ldots, L_9$) i 9 par kinematycznych ($J_0, \ldots, J_8$); jego graf jest drzewem. Punkty kontrolne korpusu, uchwytu i części dziobka czajnika zaznaczone czarnymi kropkami są ustalone w układzie współrzędnych członu L_2. Pozostałe punkty kontrolne dziobka (czerwone) mają ustalone położenia w układzie członu L_4. Punkty kontrolne pokrywki (zielone) są związane z członem L_6, zaś torus (którego punkty kontrolne są zaznaczone na niebiesko) jest związany z członem L_9. Pozostałe człony służą do określenia par kinematycznych realizujących dopuszczalne przemieszczenia obiektów (tj. ich punktów kontrolnych) względem siebie.

Opisane dalej procedury obsługi łańcucha umożliwiają arbitralne ustalenie położenia dowolnego członu łańcucha w układzie świata, a ponadto możliwe jest

zmienianie wyboru tego członu podczas działania programu[1]. W opisanej tu aplikacji członem o ustalonym położeniu (nazwiemy go korzeniem) jest człon L_0. Artykulacja łańcucha jest dokonywana za pomocą przeszukiwania grafu łańcucha w głąb (DFS), zaczynając od korzenia. Dla każdego członu należy znaleźć macierz przejścia od układu współrzędnych *będącego* tym członem do układu świata; taką macierz dla członu L_i oznaczymy symbolem A_i. Z każdą parą kinematyczną, tj. krawędzią grafu, są związane trzy przekształcenia reprezentowane przez macierze 4×4, które dla j-tej krawędzi oznaczymy F_j, R_j, B_j. Macierze F_j oraz B_j są ustalone, natomiast współczynniki macierzy R_j zależą od wartości co najmniej jednego parametru artykulacji. Aby znaleźć macierz A_i, podczas przeszukiwania grafu trzeba obliczyć iloczyny macierzy związanych z kolejnymi krawędziami przebywanymi podczas przeszukiwania. Pierwszy czynnik to macierz A_k przejścia dla korzenia L_k. Macierze kolejno przebywanych krawędzi (par kinematycznych) są domnażane z prawej strony. W przykładzie z rysunku 21.1 ścieżka od korzenia L_0 do wierzchołka L_2 składa się z krawędzi J_0 i J_1, zatem macierz przejścia od układu (członu) L_2 do układu świata jest dana wzorem

$$A_2 = A_0F_0R_0(\phi_0)B_0F_1R_1(\phi_1)B_1.$$

Podobnie, dla członów L_4, L_6 i L_9 otrzymamy odpowiednio macierze

$$\begin{aligned}
A_4 &= A_0F_0R_0(\phi_0)B_0F_1R_1(\phi_1)B_1F_2R_2(\phi_2)B_2F_3R_3(\phi_3)B_3,\\
A_6 &= A_0F_0R_0(\phi_0)B_0F_1R_1(\phi_1)B_1F_4R_4(\phi_4)B_4F_5R_5(\phi_5)B_5,\\
A_9 &= A_0F_0R_0(\phi_0)B_0F_1R_1(\phi_1)B_1F_6R_6(\phi_6)B_6F_7R_7(\phi_7)B_7F_8R_8(\phi_8)B_8.
\end{aligned}$$

Symbole $\phi_0, \ldots, \phi_8$ oznaczają parametry artykulacji, które są kątami obrotów wokół pewnych osi. Opiszę je dalej.

Tymczasem zauważmy, że na przykład

$$A_0 = A_2B_1^{-1}R_1^{-1}(\phi_1)F_1^{-1}B_0^{-1}R_0^{-1}(\phi_0)F_0^{-1}.$$

Gdybyśmy zatem ustalili położenie w przestrzeni członu L_2, aby od niego zacząć przeszukiwanie grafu (czyli wybrali ten człon na korzeń), to po drodze do członu L_0 należy jako czynniki obliczanych iloczynów przyjmować odwrotności opisanych wcześniej macierzy[2], a ponadto macierze F_j i B_j dla krawędzi przebywanych w drugą stronę „zamieniają się miejscami”. Tak więc krawędź reprezentująca dowolną parę kinematyczną musi być zorientowana. Rozwiązanie

[1]Może się to przydać, jeśli np. łańcuch reprezentuje idącego ludzika, którego stopy na przemian mają kontakt z podłożem i w związku z tym ich położenia w układzie świata na przemian dają początek artykulacji pozostałych części ludzika.

[2]Zakładamy, że wszystkie te macierze są nieosobliwe, a więc mają odwrotności.

umożliwiające przeszukiwanie grafu łańcucha od dowolnego wierzchołka polega na używaniu pary półkrawędzi reprezentującej każdą krawędź grafu. Półkrawędzie w parze są zorientowane przeciwnie (wierzchołek końcowy jednej z nich jest początkiem drugiej i nawzajem)[3] i jeśli z jedną z nich jest związana trójka macierzy (F_j, R_j, B_j), to z drugą półkrawędzią jest związana trójka $(B_j^{-1}, R_j^{-1}, F_j^{-1})$.

W łańcuchu z rysunku 21.1 wszystkie macierze F_j reprezentują przesunięcia, zaś każda macierz $R_j(\phi)$ reprezentuje obrót o kąt ϕ wokół osi y lub z układu współrzędnych. Jeśli F_j jest macierzą przesunięcia o *wektor* $\mathbf{f}_j$ oraz $B_j = F_j^{-1}$ (czyli macierz B_j opisuje przesunięcie o wektor $-\mathbf{f}_j$), to iloczyn $F_jR_j(\phi)B_j$ reprezentuje obrót o kąt ϕ wokół przechodzącej przez *punkt* $\mathbf{f}_j$ osi równoległej do osi obrotu $R_j(\phi)$ (która przechodzi przez początek układu). Zauważmy ponadto, że $R_j^{-1}(\phi) = R_j(-\phi)$ oraz $\left(F_jR_j(\phi)F_j^{-1}\right)^{-1} = F_jR_j(-\phi)F_j^{-1}$ — przekształcenie odwrotne do obrotu o kąt ϕ wokół dowolnej osi jest obrotem wokół tejże osi o kąt $-\phi$.

Rozważmy teraz złożenie przekształceń związanych z kolejnymi dwiema krawędziami w ścieżce prowadzącej od korzenia do dowolnego innego wierzchołka grafu; powiedzmy, że to są krawędzie J_j oraz J_k. Możemy napisać

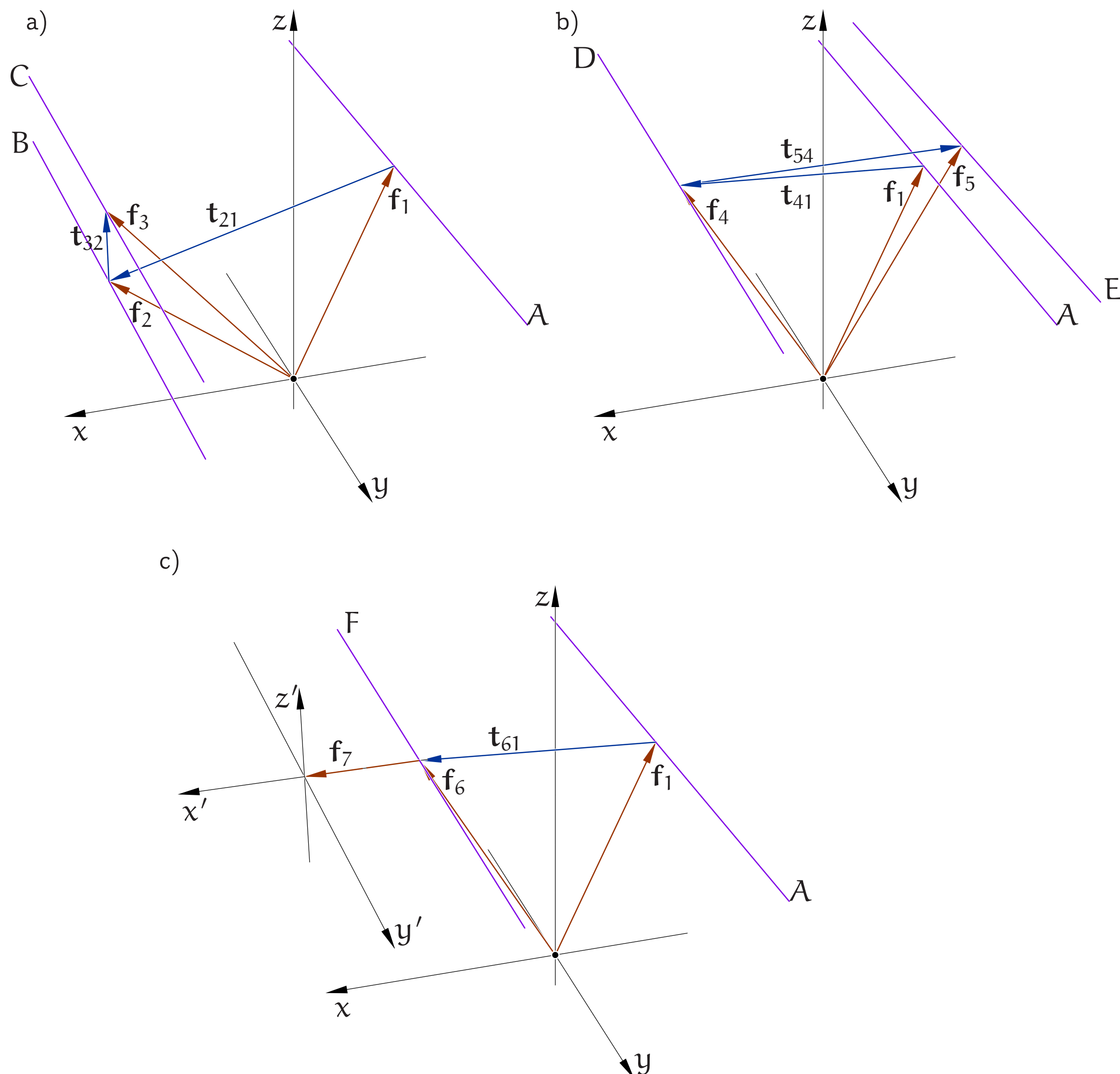
$$F_jR_j(\phi_j)B_jF_kR_k(\phi_k)B_k = F_jR_j(\phi_j)T_{kj}R_k(\phi_k)B_k.$$

Jeśli macierze $B_j^{-1} = F_j$ i F_k reprezentują przesunięcia odpowiednio o wektory $\mathbf{f}_j$ i $\mathbf{f}_k$, to iloczyn $T_{kj} = F_j^{-1}F_k$ reprezentuje złożenie tych przesunięć, czyli przesunięcie o wektor $\mathbf{t}_{kj} = \mathbf{f}_k - \mathbf{f}_j$. Jeśli macierze R_j i R_k reprezentują obroty wokół osi równoległych, a wektor $\mathbf{t}_{kj}$ jest do tych osi prostopadły, to określa przesunięcie nakładające oś pierwszego na oś drugiego obrotu. W ogólności (także gdy osie obrotów przecinają się lub są skośne) wektor $\mathbf{t}_{kj}$ jest różnicą *pewnych* punktów, z których pierwszy leży na osi drugiego obrotu, a drugi na osi pierwszego.

Na rysunku 21.2 są pokazane opisane wyżej wektory dla trzech ścieżek w grafie łańcucha kinematycznego z rysunku 21.1 i osie obrotów w położeniu wyjściowym, przyjmowanym, gdy wszystkie parametry artykulacji (kąty obrotów) są równe 0. Macierze F_j i B_j przyjmiemy tak, aby układy współrzędnych związane z członami $L_0, \ldots, L_6$ w położeniu wyjściowym były identyczne; na rysunkach 21.1 i 21.2abc są pokazane osie x, y, z wszystkich tych układów. Na rysunku 21.2c są też osie x', y', z' układu współrzędnych członu L_8 w położeniu wyjściowym.

Para kinematyczna J_0 umożliwia obracanie członów $L_1, \ldots, L_9$ wokół osi z układu współrzędnych członu L_0. Para J_1 odpowiada za obracanie członów $L_2, \ldots, L_9$

[3] Mamy tu w istocie dwa grafy — zwykły, którego krawędzie to pary kinematyczne i graf skierowany, którego krawędzie to nasze półkrawędzie.



Rysunek 21.2: Osie obrotów w położeniu wyjściowym

wokół prostej A, równoległej do osi y układu współrzędnych członu L_1 i przechodzącej przez przez punkt (o współrzędnych w tym układzie) $(-0.43, 0, 0.92)$, w związku z czym macierz F_1 jest macierzą przesunięcia o wektor $\mathbf{f}_1 = (-0.43, 0, 0.92)$. Macierz B_1 jest odwrotnością macierzy F_1. Zauważmy, że macierz $R_1(0)$ (obrotu o kąt 0 wokół osi y) jest jednostkowa, dlatego macierz $F_1R_1(0)B_1$ też jest jednostkowa. Właśnie dzięki temu układy członów L_1 i L_2 (połączonych parą kinematyczną J_1) w położeniu wyjściowym są identyczne.

Pary J_2 i J_3 realizują odkształcenia dziobka czajnika. Człon L_3 może się obracać względem członu L_2 (z którym jest związany korpus czajnika) wokół prostej B, równoległej do osi y i przechodzącej przez punkt $(0.78, 0, 0.59)$ w układzie członu L_2, natomiast człon L_4 może się obracać względem członu L_3 wokół

prostej C przechodzącej przez punkt $(0.78, 0, 0.91)$. Macierze F_2 i F_3 są macierzami przesunięć o wektory $\mathbf{f}_2 = (0.78, 0, 0.59)$ i $\mathbf{f}_3 = (0.78, 0, 0.91)$, zaś macierze B_2 i B_3 są ich odwrotnościami. Zauważmy, że w układzie współrzędnych członu L_3 prosta C ma położenie ustalone — powstaje ona przez przesunięcie prostej B o wektor $\mathbf{t}_{32} = \mathbf{f}_3 - \mathbf{f}_2$. Natomiast w układzie członu L_2 zmiany parametru ϕ_2 powodują obracanie prostej C wokół prostej B.

Podobnie, pary kinematyczne J_4 i J_5 realizują obroty wokół prostych D i E równoległych do osi y i przechodzących przez punkty $(0.6, 0, 1)$ i $(-0.6, 0, 1)$ (rysunek 21.1b). Ta gałąź łańcucha umożliwia przemieszczanie względem korpusu czajnika pokrywki, której punkty kontrolne mają ustalone położenia w układzie współrzędnych członu L_6. Tu również jest $B_4 = F_4^{-1}$ i $B_5 = F_5^{-1}$, co zapewnia pokrywanie się układów współrzędnych członów L_5 i L_6 z układem członu L_2 w położeniu wyjściowym. Co nam to daje? To, że wszystkie punkty kontrolne czajnika, dane w jednym układzie współrzędnych (w którym czajnik został zdefiniowany), możemy wygodnie przywiązać do różnych członów — wystarczy dla każdego punktu zadeklarować, w układzie którego członu jego położenie jest ustalone. Punkty zaznaczone różnymi kolorami na rysunku 21.1 ustalamy w układach członów L_2, L_4 i L_6; w położeniu wyjściowym da to nam czajnik nieodkształcony, ale zmieniając parametry artykulacji spowodujemy przemieszczanie się punktów związanych z różnymi członami względem siebie.

Nieco inaczej traktujemy krawędzie ścieżki złożonej z krawędzi J_6, J_7, J_8 w grafie łańcucha (rysunek 21.2c), ponieważ punkty kontrolne torusa są dane w układzie, którego początek jest środkiem torusa (a jego osią jest oś z). Chcemy, aby środek torusa pozostawał na osi y' układu członu L_8, obracającej się wokół równoległej do niej prostej F. Chcemy również, aby osie z układu członu L_2 i z' układu członu L_8 pozostawały równoległe. Dlatego w tym przypadku przyjmujemy macierze F_6 i F_7 reprezentujące przesunięcia o wektory $\mathbf{f}_6 = (0.52, 0, 0.97)$ i $\mathbf{f}_7 = (0.52, 0, 0)$, a na macierze B_6 i B_7 wybieramy macierz jednostkową. Jeśli zapewnimy, że $\phi_7 = -\phi_6$, to odpowiednie osie układów członów L_2 i L_8 będą zawsze równoległe, ale w położeniu wyjściowym początek układu członu L_8 jest przesunięty względem początku układu L_2 o wektor $\mathbf{f}_6 + \mathbf{f}_7$. Para J_8 realizuje obracanie torusa wokół osi z', bez przesunięć (zatem macierz $F_8 = B_8$ jest jednostkowa).

Macierz A_0 w opisanej tu aplikacji reprezentuje przesunięcie o wektor $(0, 0, -0.6)$.

Z każdym *obiektem* zwiążemy jeszcze jedną macierz, którą dla l-tego obiektu oznaczymy symbolem E_l. Macierz ta reprezentuje dodatkowe, wstępne przekształcenie obiektu. Dla czajnika macierz ta reprezentuje skalowanie osi

x, y, z o czynniki $\frac{1}{3}, \frac{1}{3}, \frac{4}{9}$ — w poprzednich wersjach drugiej aplikacji używaliśmy takiego skalowania, aby otrzymać czajnik o dogodnej wielkości, jednocześnie przywracając mu oryginalne proporcje (zobacz s. 13.26 i 13.29–29). Natomiast torus zdefiniowany przy użyciu promieni $R = 1$ i $r = \frac{1}{2}$ zmniejszamy w skali $\frac{1}{10}$ i obracamy wokół osi x o kąt $\frac{\pi}{2}$, podobnie jak w procedurze `SetupTorusMatrix` na listingu 15.2. Mając dany związany z członem L_i punkt $\mathbf{p}$ l-tego obiektu, potrzebujemy znaleźć wektor współrzędnych (jednorodnych) tego punktu w układzie świata. W tym celu obliczymy iloczyn $A_i E_l \mathbf{P}$ (gdzie $\mathbf{P}$ jest wektorem współrzędnych jednorodnych punktu $\mathbf{p}$ w układzie l-tego obiektu). To zadanie zlecimy szaderowi obliczeniowemu, ale cierpliwości.

Procedury obsługi łańcucha kinematycznego

W tym podrozdziale przedstawiam biblioteczkę procedur w C (działających na CPU), które służą do zbudowania łańcucha kinematycznego i umożliwiają, po ustaleniu wartości parametrów artykulacji, znalezienie macierzy A_i dla wszystkich jego członów. Można te procedury zastosować w programie, który użyje CPU do obliczenia współrzędnych punktów reprezentujących obiekty (tj. wierzchołków lub punktów kontrolnych) w układzie świata. Inna możliwość to przesłanie macierzy A_i (a raczej iloczynów $A_i E_l$) do pamięci GPU. Wtedy zadanie przekształcania punktów do układu świata może wykonać szader obliczeniowy, który, działając równolegle na wielu procesorach GPU, wykona to obliczenie dla wszystkich punktów naraz.

Grafy, takie jak w łańcuchu kinematycznym, można reprezentować przy użyciu wskaźnikowych struktur danych. Wybrałem jednak reprezentację składającą się z tablic; identyfikatory członów, półkrawędzi i innych elementów łańcucha są indeksami do tablic. Taka reprezentacja ma tę zaletę, że (w razie potrzeby) łatwiej jest ją zapisywać w i odczytywać z pliku.

W reprezentacji łańcucha mamy zatem tablicę członów (struktur typu `kl_link`), półkrawędzi par kinematycznych (struktur `kl_halfjoint`), obiektów (struktur `kl_object`) i referencji obiektów (`kl_obj_ref`); struktury te są zdefiniowane w pliku `linkage.h` i przedstawione na listingu 21.1. Obiekt zawiera tablicę swoich punktów i metody (procedury) ich przetwarzania. Referencja obiektu jest wykazem punktów obiektu związanych z konkretnym członem łańcucha; może zawierać informację, że *wszystkie* punkty obiektu są związane z danym członem albo tablicę indeksów punktów związanych z tym członem. Jednym z warunków poprawności łańcucha jest to, że każdy punkt każdego obiektu jest związany z pewnym członem, poprzez którąś z referencji.

Listing 21.1: Struktury łańcucha kinematycznego

C

```
 1: #define KL_ART_OTHERHALF -1
 2: #define KL_ART_NONE       0
 3: #define KL_ART_TRANS_X    1
 4: #define KL_ART_TRANS_Y    2
 5: #define KL_ART_TRANS_Z    3
 6: #define KL_ART_TRANS_XYZ  4
 7: #define KL_ART_SCALE_X    5
 8: #define KL_ART_SCALE_Y    6
 9: #define KL_ART_SCALE_Z    7
10: #define KL_ART_SCALE_XYZ  8
11: #define KL_ART_ROT_X      9
12: #define KL_ART_ROT_Y     10
13: #define KL_ART_ROT_Z     11
14: #define KL_ART_ROT_V     12
15:
16: typedef struct kl_object {
17:     int     nvc;
18:     int     nvert;
19:     GLfloat *vert, *tvert;
20:     GLfloat Etr[16];
21:     void    *usrdata;
22:     void    (*transform)(struct kl_object*,int,GLfloat*,int,int*);
23:     void    (*postprocess)(struct kl_object*);
24:     void    (*redraw)(struct kl_object*);
25:     void    (*destroy)(struct kl_object*);
26:   } kl_object;
27:
28: typedef struct kl_obj_ref {
29:     int     on;
30:     int     nextr;
31:     int     nv;
32:     int     *vn;
33:   } kl_obj_ref;
34:
35: typedef struct kl_link {
36:     int     fref;
37:     int     fhj;
38:     char    tag;
39:   } kl_link;
40:
41: typedef struct kl_halfjoint {
42:     int     l0, l1;
```

```
43:    int     otherhalf;
44:    int     nexthj;
45:    int     pnum;
46:    int     art;
47:    GLfloat Ftr[16];
48:    GLfloat Rtr[16];
49:    GLfloat Btr[16];
50:  } kl_halfjoint;
51:
52: typedef struct kl_linkage {
53:    int          maxobj,    nobj;
54:    int          maxorefs,  norefs;
55:    int          maxlinks,  nlinks;
56:    int          maxhj,     nhj;
57:    int          maxartpar, nartpar;
58:    kl_object    *obj;
59:    kl_obj_ref   *oref;
60:    kl_link      *link;
61:    kl_halfjoint *hj;
62:    GLfloat      *artp, *prevartp;
63:    int          current_root;
64:    GLfloat      current_root_tr[16];
65:    void         *usrdata;
66:  } kl_linkage;
```

W liniach 1–14 na listingu są definicje identyfikatorów możliwych rodzajów par kinematycznych; liczby ukryte pod tymi identyfikatorami są przypisywane polom `art` struktur `kl_halfjoint`. W każdej parze półkrawędzi reprezentujących parę kinematyczną jedna z półkrawędzi ma pole `art` o wartości `KL_ART_OTHERHALF`, która oznacza, że na podstawie odpowiednich parametrów artykulacji trzeba wyznaczyć macierz R_j dla drugiej półkrawędzi w parze i dla pierwszej półkrawędzi przyjąć odwrotność tamtej macierzy.

Kolejne identyfikatory określają sposób, w jaki macierz R_j może zależeć od parametrów artykulacji; `KL_ART_NONE` odpowiada macierzy jednostkowej (a więc nie ma tu pary kinematycznej, człony są połączone sztywno), zaś `KL_ART_TRANS_X`, ..., `KL_ART_TRANS_XYZ` wyznaczają przesunięcia, wzdłuż osi x, y, z i dowolne (w tym ostatnim przypadku trzeba podać trzy parametry artykulacji, będące współrzędnymi wektora przesunięcia). Identyfikatory `KL_ART_SCALE_X`, ..., `KL_ART_SCALE_XYZ` opisują skalowania osi x, y, z i wszystkich trzech osi naraz (skalowania nie są izometriami, zatem zmienianie parametrów artykulacji takich par kinematycznych sprawi, że obiekty nie będą

sztywne), wreszcie identyfikatory KL_ART_ROT_X, ..., KL_ART_ROT_V określają pary obrotowe, umożliwiające obroty wokół osi x, y, z i obroty wokół osi o dowolnym kierunku wyznaczonym przez wektor $\boldsymbol{v}$, którego współrzędne są parametrami artykulacji w dodatku do kąta obrotu (zatem są tu 4 parametry artykulacji).

Struktura kl_object opisuje pojedynczy obiekt, który może być np. bryłą wielościenną lub (w aplikacji drugiej H) zespołem płatów Béziera. Pole nvc ma wartość 3 albo 4; określa ono liczbę współrzędnych każdego punktu (wierzchołka lub punktu kontrolnego) obiektu; w pierwszym przypadku to są współrzędne kartezjańskie w $\mathbb{R}^3$, a w drugim współrzędne jednorodne. Wartość pola nvert jest liczbą punktów. Pola vert i tvert są wskaźnikami tablic tych punktów, przy czym pierwsza tablica zawiera punkty oryginalne, dane w układzie, w którym obiekt został zdefiniowany, a do drugiej będą wpisywane współrzędne punktów w układzie świata, otrzymane jako wynik artykulacji. Jeśli aplikacja (taka jak druga H) przechowuje te punkty tylko w pamięci GPU, to pola vert i tvert mogą mieć wartość NULL. Pole Etr przechowuje macierz E_l opisaną wcześniej. Pole usrdata jest wskaźnikiem dowolnych danych skojarzonych z obiektem przez aplikację.

Pola transform, postprocess, redraw i destroy wskazują metody obiektu, których działanie będzie opisane dalej. Tworząc obiekt, aplikacja może podać wskaźniki procedur, które mają być tymi metodami, albo wskaźniki puste (NULL), co spowoduje przypisane obiektowi metod domyślnych przedstawianej tu biblioteczki.

Struktura kl_obj_ref opisuje referencję obiektu. Struktury te są przechowywane w tablicy, w której są łączone w listy przypisane do członów łańcucha. Pole on jest numerem obiektu (tj. indeksem do tablicy obiektów struktury łańcucha). Pole nextr jest wskaźnikiem następnego elementu listy referencji (czyli indeksem do tablicy referencji), przy czym wskaźnik pusty w ostatnim elemencie listy jest reprezentowany przez liczbę -1. Pole nv opisuje, ile punktów obiektu jest przywiązanych do członu przez daną referencję, a pole vn jest wskaźnikiem tablicy indeksów tych punktów (indeksy te określają punkty w tablicach vert i nvert obiektu lub w odpowiednich tablicach w pamięci GPU). Jeśli pole vn ma wartość NULL, to z danym członem są związane wszystkie punkty obiektu.

Struktura kl_link opisuje człon łańcucha. Pole fref wskazuje pierwszy element listy referencji członu (jest to indeks do tablicy referencji). Pole fhj wskazuje pierwszy element listy par kinematycznych wiążących ten człon. Podobnie, jest to indeks do tablicy półkrawędzi reprezentujących krawędź grafu łańcucha, czyli parę

kinematyczą. Dany człon jest początkiem tej półkrawędzi. Pole tag służy do zaznaczania odwiedzonych wierzchołków podczas przeszukiwania grafu metodą DFS — przeszukiwany jest graf skierowany wzdłuż półkrawędzi, a każda ich para reprezentująca parę kinematyczną tworzy cykl, dlatego takie zaznaczanie jest konieczne dla uniknięcia wejścia procedury przeszukiwania grafu w pętlę.

Struktura kl_halfjoint reprezentuje półkrawędź grafu. Pola l0 i l1 zawierają numery (indeksy do tablicy) członów, z których pierwszy jest początkiem, a drugi końcem półkrawędzi. Pole otherhalf jest numerem drugiej półkrawędzi w parze; początkiem i końcem tej drugiej półkrawędzi są człony l1 i l0. Pole nexthj jest wskaźnikiem następnej półkrawędzi w liście półkrawędzi wychodzących z członu l0 (lub jest to wskaźnik pusty, czyli liczba -1 sygnalizująca koniec listy). Pole pnum jest indeksem do tablicy parametrów artykulacji; wskazuje ono pierwszy parametr pary, przy czym liczba tych parametrów jest określona przez pole art, mające wartość ukrytą za jednym z wcześniej opisanych identyfikatorów, takich jak KL_ART_ROT_X. Zawsze jedna z półkrawędzi ma w tym polu identyfikator KL_ART_OTHERHALF, a jej druga połowa jeden z pozostałych identyfikatorów. W polach Ftr, Rtr i Btr są przechowywane macierze F_j, R_j i B_j.

Struktura kl_linkage jest punktem wejściowym do reprezentacji całego łańcucha. Pola maxobj, maxorefs, maxlinks, maxhj i maxartpar określają długości tablic zaalokowanych dla obiektów, referencji, członów, półkrawędzi i parametrów artykulacji, czyli maksymalne (określone przez aplikację) liczby tych elementów łańcucha. Pola nobj, norefs, nlinks, nhj i nartpar oznaczają bieżące (w trakcie budowania łańcucha i po jego zakończeniu) liczby obiektów, referencji itd. Pola obj, oref, link i hj wskazują tablice obiektów, referencji, członów i półkrawędzi. W tablicy wskazywanej przez pole artp są przechowywane bieżące wartości parametrów artykulacji, natomiast w tablicy prevartp są pamiętane ich wartości poprzednie. Dzięki temu po zmianie niektórych parametrów artykulacji można obliczać tylko te macierze R_j, które zależą od zmienionych parametrów.

Pole current_root jest numerem członu, od którego zaczyna się przeszukiwanie grafu. W tablicy current_root_tr jest przechowywana macierz przejścia od układu współrzędnych korzenia do układu świata. Parametr usrdata może wskazywać dowolną strukturę danych określoną przez aplikację.

Po zastanowieniu postanowiłem nie zamieszczać tu pełnego kodu źródłowego wszystkich procedur, a tylko najważniejsze jego części — wystarczające do zrozumienia algorytmu artykulacji i do właściwego użycia procedur w aplikacji. Po szczegóły odsyłam Czytelnika do pliku źródłowego linkage.c.

Listing 21.2: Procedury kl_NewLinkage i kl_DestroyLinkage

```c
kl_linkage *kl_NewLinkage ( int maxo, int maxl, int maxr,
                            int maxj, int maxp, void *usrdata )
{
  kl_linkage *lkg;

  lkg = malloc ( sizeof(kl_linkage) );
  if ( lkg ) {
    ........ /* inicjalizacja pól i alokacja tablic */
  }
  return lkg;

failure:
    ........ /* likwidacja skutków niepowodzeia alokacji */
  free ( lkg );
  return NULL;
} /*kl_NewLinkage*/

void kl_DestroyLinkage ( kl_linkage *linkage )
{
    ........ /* dealokacja tablic, w tym niszczenie obiektów */
  free ( linkage );
} /*kl_DestroyLinkage*/
```

Budowę łańcucha należy zacząć od wywołania procedury kl_NewLinkage (listing 21.2), która rezerwuje pamięć na strukturę kl_linkage i wszystkie wskazywane przez nią tablice i nadaje polom tej struktury wartości początkowe, opisujące limity liczb elementów w tablicach i początkowe liczby tych elementów, równe 0. Kolejne parametry deklarują zapotrzebowanie na tablice wystarczające do pomieszczenia odpowiednich liczb obiektów, członów, referencji, par kinematycznych i parametrów artykulacji. Długość alokowanej tablicy półkrawędzi jest dwukrotnie większa niż deklarowana liczba par kinematycznych (tj. wartość parametru maxj). Ostatni parametr jest wskaźnikiem dowolnej struktury określonej w aplikacji, którą ta zamierza skojarzyć z całym łańcuchem.

Pole current_root otrzymuje domyślną wartość 0, a do tablicy current_root_tr są wpisywane współczynniki macierzy jednostkowej.

Procedura kl_DestroyLinkage likwiduje łańcuch, w tym zwalnia pamięć zajmowaną przez wszystkie tablice. W szczególności dla każdego obiektu wywoływana jest procedura destruktora podana przez aplikację podczas tworzenia

obiektu. Procedurę kl_DestroyLinkage wypada wywołać podczas sprzątania przy końcu działania aplikacji.

Listing 21.3 przedstawia (całą) procedurę kl_NewObject, którą trzeba wywołać tyle razy, ile obiektów ma być artykułowanych przy użyciu łańcucha. Parametr linkage jest wskaźnikiem struktury wcześniej utworzonej przez procedurę kl_NewLinkage, parametry nvc i nvert określają liczbę współrzędnych (3 albo 4) każdego punktu obiektu i liczbę tych punktów. Parametr etrans jest tablicą ze współczynnikami macierzy E_l dla obiektu; jeśli ma wartość NULL, to przyjęta zostanie macierz jednostkowa.

Parametr usrdata jest wskaźnikiem, który zostanie zapamiętany w polu usrdata, a ponadto przekazany metodzie konstrukcji obiektu (procedurze wskazywanej przez parametr init, jeśli ma wartość inną niż NULL). Ostatnie 5 parametrów to wskaźniki metod obiektu, dostarczonych przez aplikację. Parametr init jest adresem konstruktora obiektu. Może on zaalokować tablice wierzchołków i przypisać ich adresy polom vert i tvert. Konstruktor może też zaalokować bufory OpenGL-a w pamięci GPU i zapamiętać ich identyfikatory w strukturze danych wskazywanej przez parametr usrdata. Konstruktor powinien zawiadomić o sukcesie inicjalizacji, zwracając wartość niezerową, a o porażce, zwracając 0.

Parametry transform, postprocess, redraw i destroy mogą być równe NULL, co spowoduje przypisanie odpowiednim polom struktury kl_object adresów domyślnych metod przetwarzania obiektu. Domyślna metoda transform, czyli procedura kl_DefaultTransform pokazana na listingu 21.4, dokonuje przekształcenia punktów do układu świata, czytając punkty z tablicy vert obiektu i wpisując wyniki przekształcenia do tablicy tvert. Pozostałe trzy metody domyślne nic nie robią.

Choć aplikacja druga H nie korzysta z procedury kl_DefaultTransform, przyjrzyjmy się, jak ona działa. Procedura jest wywoływana przez opisaną dalej procedurę kl_Articulate z parametrami zawierającymi następujące informacje: obj jest wskaźnikiem (adresem) struktury obiektu, refn jest numerem referencji, tr jest tablicą współczynników macierzy przekształcenia (macierzy $A_i E_l$), a parametry nv i vn podają liczbę wierzchołków obiektu opisanych przez tę referencję i tablicę z numerami tych wierzchołków. Jeśli parametr vn ma wartość NULL, to należy przekształcić wszystkie punkty obiektu. W przeciwnym razie należy przekształcić nv punktów o indeksach podanych we wskazywanej przez ten parametr tablicy. Zależnie od liczby współrzędnych, kolejne punkty są opisane przez trójki albo czwórki liczb podanych w tablicy obj->vert.

Listing 21.3: Procedura kl_NewObject

C

```
1: typedef char (*kl_obj_init)(kl_object*,void*);
2: typedef void (*kl_obj_transform)(kl_object*,int,GLfloat*,int,int*);
3: typedef void (*kl_obj_postprocess)(kl_object*);
4: typedef void (*kl_obj_redraw)(kl_object*);
5: typedef void (*kl_obj_destroy)(kl_object*);
6:
7: int kl_NewObject ( kl_linkage *linkage, int type,
8:                    int nvc, int nvert, const GLfloat *etrans, void *usrdata,
9:                    kl_obj_init init,
10:                   kl_obj_transform transform,
11:                   kl_obj_postprocess postprocess,
12:                   kl_obj_redraw redraw,
13:                   kl_obj_destroy destroy )
14: {
15:   int        on;
16:   kl_object *obj;
17:
18:   if ( linkage->nobj < linkage->maxobj ) {
19:     on = linkage->nobj;
20:     obj = &linkage->obj[on];
21:     memset ( obj, 0, sizeof(kl_object) );
22:     obj->type = type;
23:     if ( etrans )
24:       memcpy ( obj->Etr, etrans, 16*sizeof(GLfloat) );
25:     else
26:       M4x4Identf ( obj->Etr );
27:     obj->transform = transform != NULL ? transform : kl_DefaultTransform;
28:     obj->postprocess = postprocess != NULL ? postprocess : kl_obj_stub;
29:     obj->redraw = redraw != NULL ? redraw : kl_obj_stub;
30:     obj->destroy = destroy != NULL ? destroy : kl_obj_stub;
31:     obj->nvc = nvc;
32:     obj->nvert = nvert;
33:     obj->vert = obj->tvert = NULL;
34:     obj->usrdata = usrdata;
35:     if ( !init || init ( obj, usrdata ) ) {
36:       linkage->nobj ++;
37:       return on;
38:     }
39:   }
40:   return -1;
41: } /*kl_NewObject*/
```

Listing 21.4: Procedura kl_DefaultTransform

```c
 1: void kl_DefaultTransform ( kl_object *obj, int refn, GLfloat *tr,
 2:                            int nv, int *vn )
 3: {
 4:   int     i, k;
 5:   GLfloat *vert, *tvert;
 6:
 7:   if ( !(vert = obj->vert) || !(tvert = obj->tvert) )
 8:     return;
 9:   if ( vn ) {
10:     switch ( obj->nvc ) {
11:   case 3:
12:       for ( i = 0; i < nv; i++ )
13:         { k = 3*vn[i];  M4x4MultMP3f ( &tvert[k], tr, &vert[k] ); }
14:       break;
15:   case 4:
16:       for ( i = 0; i < nv; i++ )
17:         { k = 4*vn[i];  M4x4MultMVf ( &tvert[k], tr, &vert[k] ); }
18:       break;
19:   default:
20:       return;
21:     }
22:   }
23:   else {
24:     ........ /* tu przekształcanie wszystkich punktów, od 0 do nv-1 */
25:   }
26: } /*kl_DefaultTransform*/
```

Zatem, po odpowiednie trójki albo czwórki liczb procedura sięga do miejsc otrzymanych przez pomnożenie numeru z tablicy vn przez 3 albo 4. Przypomnijmy, że procedura M4x4MultMP3f „dołącza” czwartą współrzędną jednorodną równą 1, mnoży otrzymany w ten sposób wektor przez podaną macierz 4×4 i odrzuca ostatnią współrzędną iloczynu, która w przypadku, gdy macierz ma w ostatnim wierszu liczby $0, 0, 0, 1$ (zakładamy, że zawsze tak jest), jest równa 1. Otrzymana trójka liczb (współrzędnych kartezjańskich punktu w układzie świata) jest zapamiętywana w odpowiednim miejscu tablicy obj->tvert. Jeśli punkty mają 4 współrzędne (które są współrzędnymi jednorodnymi), to wywoływana jest procedura M4x4MultMVf, która mnoży macierz 4×4 przez wektor w $\mathbb{R}^4$. Pominięty fragment procedury przetwarza w podobny sposób wszystkie punkty o numerach od 0 do nv-1.

Na listingu 21.5 są pokazane procedury budowania łańcucha. Każda z nich inicjalizuje kolejny element odpowiedniej tablicy zaalokowanej przez procedurę

kl_NewLinkage, przy czym procedura kl_NewJoint inicjalizuje dwa elementy — półkrawędzie reprezentujące jedną parę kinematyczną. Wartość powrotna każdej z tych procedur jest indeksem odpowiedniego elementu tablicy (w przypadku kl_NewJoint jest to indeks pierwszej półkrawędzi).

Procedura kl_NewLink alokuje strukturę kl_link i nadaje początkowe wartości domyślne wszystkim jej polom; w szczególności (w linii 10) wskaźniki, tj. indeksy, początków list półkrawędzi wychodzących z wierzchołka grafu reprezentującego człon i list referencji obiektów odpowiedniego członu otrzymują wartość -1, która jest interpretowana jako wskaźnik pusty.

Po utworzeniu obiektów i członów można wywoływać procedury wprowadzające referencje i pary kinematyczne. Parametry procedury kl_NewObjRef to kolejno wskaźnik struktury łańcucha, numer członu, w którego liście ma się znaleźć tworzona referencja (numer ten musi być wartością podaną przez kl_NewLink), numer obiektu, którego punkty są przywiązane do członu przez tę referencję, liczba tych punktów i tablica zawierająca indeksy tych punktów w tablicy punktów obiektu. Jeśli ten ostatni parametr ma wartość NULL, to referencja dotyczy wszystkich punktów obiektu; w przeciwnym razie zawartość przekazanej tablicy jest przepisywana do tablicy zaalokowanej przez procedurę kl_NewObjRef. Instrukcje w liniach 26–27 wstawiają referencję do listy referencji członu określonego przez parametr lkn.

Parametry procedury kl_NewJoint to wskaźnik struktury łańcucha, numery członów będących początkiem i końcem nowej półkrawędzi, identyfikator rodzaju pary kinematycznej (jedna ze stałych o nazwach symbolicznych KL_ART_NONE, ..., KL_ART_ROT_V) oraz indeks pierwszego parametru artykulacji wprowadzanej pary kinematycznej w tablicy parametrów łańcucha. Zgodnie z wcześniejszym stwierdzeniem, procedura inicjalizuje dwa elementy w tablicy półkrawędzi łańcucha, nadając im wartości reprezentujące przeciwnie zorientowane półkrawędzie. Każda z tych półkrawędzi jest wstawiana do listy półkrawędzi wychodzących z wierzchołka będącego jej początkiem. Ponadto do tablic Ftr, Rtr i Btr są wpisywane współczynniki macierzy jednostkowej.

W razie wywołania dowolnej z opisanych wyżej procedur zbyt wiele razy (po wywołaniu procedury kl_NewLink z parametrami o zbyt małych wartościach) procedury te zwracają wartość -1.

Procedury kl_SetJointFtr i kl_SetJointBtr (listing 21.6) służą do przypisania wskazanej parze kinematycznej macierzy F_j i B_j. Parametr jn podaje numer j

Listing 21.5: Procedury kl_NewLink, kl_NewObjRef i kl_NewJoint

C

```
1: int kl_NewLink ( kl_linkage *linkage )
2: {
3:   int     lkn;
4:   kl_link *lk;
5:
6:   if ( linkage->nlinks < linkage->maxlinks ) {
7:     lkn = linkage->nlinks ++;
8:     lk = &linkage->link[lkn];
9:     memset ( lk, 0, sizeof(kl_link) );
10:    lk->fref = lk->fhj = -1;
11:    lk->tag = 0;
12:    return lkn;
13:  }
14:  return -1;
15: } /*kl_NewLink*/
16:
17: int kl_NewObjRef ( kl_linkage *linkage, int lkn, int on, int nv, int *vn )
18: {
19:   int        orn;
20:   kl_obj_ref *oref;
21:
22:   if ( linkage->norefs < linkage->maxorefs ) {
23:     orn = linkage->norefs ++;
24:     oref = &linkage->oref[orn];
25:     ........ /* inicjalizacja pól struktury */
26:     oref->nextr = linkage->link[lkn].fref;
27:     linkage->link[lkn].fref = orn;
28:     return orn;
29:   }
30:   return -1;
31: } /*kl_NewObjRef*/
32:
33: int kl_NewJoint ( kl_linkage *linkage, int l0, int l1, int art, int pnum )
34: {
35:   int          jn0, jn1;
36:   kl_halfjoint *hj0, *hj1;
37:
38:   if ( linkage->nhj < linkage->maxhj-1 &&
39:        l0 < linkage->nlinks && l1 < linkage->nlinks ) {
40:     jn0 = linkage->nhj ++;
41:     jn1 = linkage->nhj ++;
42:     hj0 = &linkage->hj[jn0];
```

```
43:     hj1 = &linkage->hj[jn1];
44:     ........ /* inicjalizacja pól obu struktur półkrawędzi */
45:     return jn0;
46:   }
47:   return -1;
48: } /*kl_NewJoint*/
```

półkrawędzi, której przypisana ma być macierz o współczynnikach podanych w tablicy `tr`; numer ten był wartością powrotną procedury `kl_NewJoint`. Jednocześnie z przypisaniem j-tej półkrawędzi macierzy F_j, której współczynniki należy podać w tablicy `tr`, jej druga półkrawędź z pary, o numerze k, otrzymuje macierz $B_k = F_j^{-1}$. Jeśli parametr `back` nie jest zerem, to dodatkowo następują przypisania $B_j = F_j^{-1}$ i $F_k = F_j$. Służy to do osiągnięcia stanu, w którym układy członów połączonych w parę kinematyczną, mają tożsame układy współrzędnych gdy macierz $R_j = R_k^{-1}$ jest jednostkowa. Procedura `kl_SetJointBtr` działa podobnie, przypisując wskazanej przez parametr `jn` półkrawędzi macierz B_j (a także macierz $F_k = B_j^{-1}$ drugiej półkrawędzi z pary).

Listing 21.6: Procedury `kl_SetJointFtr` i `kl_SetJointBtr`

```C
1: void kl_SetJointFtr ( kl_linkage *linkage, int jn, GLfloat *tr, char back )
2: {
3:   kl_halfjoint *hj, *hj1;
4:
5:   if ( jn < 0 || jn >= linkage->nhj )
6:     return;
7:   hj = &linkage->hj[jn];
8:   hj1 = &linkage->hj[hj->otherhalf];
9:   memcpy ( &hj->Ftr, tr, 16*sizeof(GLfloat) );
10:  M4x4Invertf ( hj1->Btr, tr );
11:  if ( back ) {
12:    memcpy ( hj->Btr, hj1->Btr, 16*sizeof(GLfloat) );
13:    memcpy ( hj1->Ftr, hj->Ftr, 16*sizeof(GLfloat) );
14:  }
15: } /*kl_SetJointFtr*/
16:
17: void kl_SetJointBtr ( kl_linkage *linkage, int jn, GLfloat *tr, char front )
18: {
19:   ........ /* tu jest symetryczne odbicie treści procedury kl_SetJointFtr */
20: } /*kl_SetJointBtr*/
```

Listing 21.7 przedstawia w skrócie pomocniczą procedurę, która na podstawie parametrów artykulacji obecnych w tablicy `artp` łańcucha oblicza macierz R_j

Listing 21.7: Procedura _kl_UpdateArtTr

```c
static void _kl_UpdateArtTr ( kl_linkage *linkage, int jn )
{
  kl_halfjoint *hj, *hj1;
  int          pnum;
  GLfloat      *artp, *prevartp;

  artp = linkage->artp;
  prevartp = linkage->prevartp;
  hj = &linkage->hj[jn];
  if ( hj->art == KL_ART_OTHERHALF ) {
    return;
  hj1 = &linkage->hj[hj->otherhalf];
  pnum = hj->pnum;
  switch ( hj->art ) {
case KL_ART_TRANS_X:
    if ( _kl_changed ( artp, prevartp, pnum, 1 ) ) {
      M4x4Translatef ( hj->Rtr, artp[pnum], 0.0, 0.0 );
      M4x4Translatef ( hj1->Rtr, -artp[pnum], 0.0, 0.0 );
    }
    break;
    ........ /* tu inne rodzaje par kinematycznych */
case KL_ART_ROT_V:
    if ( _kl_changed ( artp, prevartp, pnum, 4 ) ) {
      M4x4RotateVf ( hj->Rtr, artp[pnum], artp[pnum+1], artp[pnum+2],
                     artp[pnum+3] );
      M4x4Transposef ( hj1->Rtr, hj->Rtr );
    }
    break;
default:
    M4x4Identf ( hj->Rtr );
    M4x4Identf ( hj1->Rtr );
    break;
  }
} /*_kl_UpdateArtTr*/
```

półkrawędzi i $R_k = R_j^{-1}$ drugiej połowy tej półkrawędzi. Przypomnę, że jedna z półkrawędzi w parze ma atrybut art, określający sposób artykulacji, równy KL_ART_OTHERHALF. Właściwy sposób artykulacji jest podany w drugiej półkrawędzi, dlatego jeśli półkrawędź określona przez parametr jn ma atrybut art równy KL_ART_OTHERHALF, następuje powrót — obliczenie macierzy nastąpi podczas przetwarzania drugiej półkrawędzi z pary.

W instrukcji przełącznika (linie 14–33) następuje wybór sposobu tworzenia macierzy R_j i R_k; spośród dwunastu sposobów odpowiadających identyfikatorom z listingu 21.1 są pokazane dwa: przesunięcie wzdłuż osi x i obrót wokół osi określonej przez wektor $\boldsymbol{v}$. W pierwszym przypadku przesunięcie zależy od jednego parametru, o numerze `pnum`; macierz przesunięcia i jej odwrotność są tworzone przez procedurę `M4x4Translatef` (opisaną w rozdziale 4). W drugim przypadku macierz jest określona przez cztery parametry artykulacji, zajmujące kolejne miejsca w tablicy zaczynając od numeru `pnum`; są to trzy współrzędne wektora $\boldsymbol{v}$ i kąt ϕ_j obrotu. Odwrotności macierzy ortogonalnych (reprezentujących obroty) są transpozycjami tych macierzy, zatem użycie procedury `M4x4Transposef` jest najprostszym (i najszybszym) sposobem znalezienia odwrotności macierzy obrotu. Pomocnicza procedura `_kl_changed` porównuje zawartości wskazanych fragmentów tablic `artp` i `prevartp` i zwraca 0 gdy te są identyczne; w takim przypadku obliczenie macierzy zostaje pominięte, bo zakłada się, że macierze te są niezmienione od poprzedniego obliczenia (po uaktualnieniu wszystkich macierzy R_j parametry artykulacji są kopiowane do tablicy `prevartp`).

Listing 21.8 przedstawia procedurę `kl_Articulate`, którą należy wywołać po nadaniu nowej wartości któremuś parametrowi artykulacji, przed narysowaniem sceny (tj. obiektów łańcucha) w nowym położeniu. Celem tej procedury jest wyznaczenie, dla każdego członu łańcucha, macierzy A_i przejścia do układu świata, a następnie, dla każdej referencji obiektu w liście tego członu, obliczenie iloczynu A_iE_l (gdzie macierz E_l jest określona dla obiektu, którego punkty są wymienione w tej referencji) i wywołanie metody `transform`, której zadaniem jest przekształcenie punktów do układu świata (domyślna metoda to pokazana na listingu 21.4 procedura `kl_DefaultTransform`; w aplikacji drugiej H użyjemy innej metody).

Pętla w liniach 36–37 wywołuje procedurę z listingu 21.7 w celu skonstruowania macierzy R_j dla wszystkich par kinematycznych na podstawie bieżących wartości parametrów artykulacji. Po zakończeniu tego obliczenia parametry te są kopiowane do tablicy `prevartp`, w opisanym wcześniej celu.

W linii 40 polom `tag` wszystkich członów przypisywana jest wartość 0, która oznacza człony — wierzchołki grafu — jako nieodwiedzone.

W linii 41 jest wywoływana procedura `_kl_rArticulate`, która przeszukuje graf łańcucha metodą DFS, zaczynając od wierzchołka będącego bieżącym korzeniem. Parametr `lkn` tej procedury jest numerem przetwarzanego wierzchołka, zaś parametr `tr` jest tablicą zawierającą współczynniki macierzy A_i, która jest

Listing 21.8: Procedura kl_Articulate

C

```
1: static void _kl_rArticulate ( kl_linkage *linkage, int lkn, GLfloat *tr )
2: {
3:   kl_link      *lk;
4:   GLfloat      t0[16], t1[16];
5:   int          r, on, j, l1;
6:   kl_object    *obj;
7:   kl_obj_ref   *oref;
8:   kl_halfjoint *hj;
9:
10:   lk = &linkage->link[lkn];
11:   lk->tag = 1;
12:   obj = linkage->obj;
13:   oref = linkage->oref;
14:   hj = linkage->hj;
15:   for ( r = lk->fref; r >= 0; r = oref[r].nextr ) {
16:     on = oref[r].on;
17:     M4x4Multf ( t1, tr, obj[on].Etr );
18:     obj[on].transform ( &obj[on], r, t1, oref[r].nv, oref[r].vn );
19:   }
20:   for ( j = lk->fhj; j >= 0; j = hj[j].nexthj ) {
21:     l1 = hj[j].l1;
22:     if ( !linkage->link[l1].tag ) {
23:       M4x4Multf ( t0, tr, hj[j].Ftr );
24:       M4x4Multf ( t1, t0, hj[j].Rtr );
25:       M4x4Multf ( t0, t1, hj[j].Btr );
26:       _kl_rArticulate ( linkage, l1, t0 );
27:     }
28:   }
29: } /*_kl_rArticulate*/
30:
31: void kl_Articulate ( kl_linkage *linkage )
32: {
33:   int        i;
34:   kl_object *obj;
35:
36:   for ( i = 0; i < linkage->nhj; i++ )
37:     _kl_UpdateArtTr ( linkage, i );
38:   memcpy ( linkage->prevartp, linkage->artp,
39:            linkage->maxartpar*sizeof(GLfloat) );
40:   for ( i = 0; i < linkage->nlinks; i++ ) linkage->link[i].tag = 0;
41:   _kl_rArticulate ( linkage, linkage->current_root,
42:                     linkage->current_root_tr );
```

```
43:   for ( i = 0; i < linkage->nobj; i++ )
44:     linkage->obj[i].postprocess ( &linkage->obj[i] );
45: } /*kl_Articulate*/
```

iloczynem macierzy $A_k F_{j_1} R_{j_1} B_{j_1} \cdot \ldots \cdot F_{j_m} R_{j_m} B_{j_m}$. Macierz A_k opisuje przejście od układu korzenia do układu świata, a kolejne czynniki to odpowiednie macierze krawędzi w ścieżce od korzenia do bieżącego (i-tego) wierzchołka.

W liniach 43–44 dla każdego obiektu jest wywoływana metoda `postprocess`, której zadaniem jest wykonanie dodatkowego obliczenia dla każdego obiektu, jeśli tylko takie obliczenie jest potrzebne[4].

Procedura `_kl_rArticulate` w linii 11 zaznacza, że przetwarzany przez nią wierzchołek grafu (człon łańcucha) jest już odwiedzony. W pętli w liniach 15–19 przeglądana jest lista referencji obiektów tego członu. Dla każdej referencji jest (w linii 17) obliczany iloczyn $A_i E_l$, po czym jest wywoływana metoda `transform` obiektu z parametrami opisującymi obiekt, numer referencji, macierz $A_i E_l$, liczbę wierzchołków i tablicę indeksów wierzchołków obiektu w tej referencji. W pętli w liniach 20–28 jest przeglądana lista półkrawędzi wychodzących z wierzchołka; dla każdej z tych półkrawędzi, które kończą się w nieodwiedzonym wierzchołku, macierz A_i jest mnożona z prawej strony przez macierze F_j, R_j i B_j tej półkrawędzi i następuje rekurencyjne wywołanie procedury dla wierzchołka końcowego półkrawędzi.

Listing 21.9: Procedura `kl_Redraw`

C

```
1: void kl_Redraw ( kl_linkage *linkage )
2: {
3:   int i;
4:
5:   for ( i = 0; i < linkage->nobj; i++ )
6:     linkage->obj[i].redraw ( &linkage->obj[i] );
7: } /*kl_Redraw*/
```

Procedura `kl_Redraw` pokazana na listingu 21.9 kolejno dla każdego obiektu wywołuje jego metodę `redraw`.

[4]Na przykład, po przekształceniu punktów kontrolnych powierzchni B-sklejanej może być potrzebne podzielenie tej powierzchni na płaty Béziera. W aplikacji drugiej H metoda `postprocess` wywoła szader obliczeniowy, który obliczy współrzędne punktów kontrolnych przekształconego czajnika i torusa w układzie świata na podstawie danych w pamięci GPU.

Szader obliczeniowy

Zadaniem szadera obliczeniowego aplikacji drugiej H jest obliczenie współrzędnych w układzie świata punktów kontrolnych czajnika i torusa na podstawie współrzędnych w układach, w których te obiekty zostały zdefiniowane. Kod szadera opisuje obliczenie dla *jednego* punktu. Numer tego punktu jest brany ze zmiennej wbudowanej `gl_GlobalInvocationId.x`, o czym będzie mowa dalej. Szader odczytuje współrzędne punktu z tablicy `cp` w buforze zawierającym blok zmiennych jednolitych `CPoints` (lokalnie nazwanym `cpin`) i wpisuje wynik do tablicy `cp` w buforze magazynowym dostępnym jako blok `CPointsOut` (o lokalnej nazwie `cpout`)[5]. Oba bloki, `CPoints` i `CPointsOut`, są identycznie zbudowane, zawierają tablicę punktów kontrolnych. Liczba współrzędnych każdego punktu, 3 lub 4, ma być podana w zmiennej jednolitej `dim`, o czym nie wolno zapomnieć.

W linii 21 są zadeklarowane trzy zmienne jednolite. Wartość zmiennej `ncp` jest liczbą punktów do przekształcenia. Szadery obliczeniowe wywołuje się w blokach zwanych grupami roboczymi (*workgroups*) o ustalonej wielkości. Gdyby liczba punktów była wielokrotnością liczby wątków (*invocations*) szadera w jednej grupie, to każdy wątek miałby do przetworzenia jeden punkt. W przeciwnym razie pewne wątki nie mają nic do roboty — warunek sprawdzany w linii 30 zapewnia, że „nadliczbowe" wątki powstrzymają się od działań, które byłyby destrukcyjne.

W bloku zmiennych jednolitych `Tr` (o lokalnej nazwie `cptr`) znajduje się tablica macierzy 4×4; to są macierze przejścia od układu modelu, poprzez układy odpowiednich członów łańcucha kinematycznego, do układu świata. Wartość zmiennej jednolitej `trnum`, jeśli jest nieujemna (co jest sprawdzane w linii 32), jest indeksem do tablicy `cptr.tr` i wybiera przekształcenie, któremu mają być poddane *wszystkie* punkty. To zdarza się w sytuacji, gdy wszystkie punkty danego obiektu mają położenia ustalone w układzie jednego członu (i nie korzysta się wtedy z tablicy `cptrind.ind`). Jeśli zmienna `trnum` ma wartość ujemną, to macierz przekształcenia należy wybrać z tablicy `cptrind.ind`, przy użyciu indeksu, który jest numerem przetwarzanego punktu. Tak więc szader czyta współrzędne jednego punktu, wybiera odpowiednią macierz, oblicza wynik i zapisuje go we właściwym miejscu bufora magazynowego. Działające równolegle (na wielu procesorach GPU) wątki szadera nie mają konfliktów w dostępie do tego bufora, bo każdy wątek wpisuje swój wynik do innego miejsca w nim.

Zależnie od liczby współrzędnych punktu, w liniach 33–35 albo 38–41 szader wybiera z tablicy odpowiednią trójkę albo czwórkę liczb, tworzy z nich wektor

[5] Przypomnijmy, że zmienne jednolite mogą być tylko czytane przez szadery.

Listing 21.10: Szader obliczeniowy artykulacji

GLSL

```
1: #version 450 core
2:
3: layout (local_size_x=128) in;
4:
5: uniform CPoints {
6:     float cp[1];
7:   } cpin;
8:
9: uniform Tr {
10:     mat4 tr[1];
11:   } cptr;
12:
13: uniform TrInd {
14:     uint ind[1];
15:   } cptrind;
16:
17: layout (binding=0) buffer CPointsOut {
18:     float cp[];
19:   } cpout;
20:
21: uniform int dim, ncp, trnum;
22:
23: void main ( void )
24: {
25:   uint i, k;
26:   vec4 v0, v1;
27:
28:   if ( (i = gl_GlobalInvocationID.x) < ncp ) {
29:     k = dim*i;
30:     i = trnum >= 0 ? trnum : cptrind.ind[i];
31:     switch ( dim ) {
32:   case 3:
33:       v0 = vec4 ( cpin.cp[k], cpin.cp[k+1], cpin.cp[k+2], 1.0 );
34:       v1 = cptr.tr[i]*v0;
35:       cpout.cp[k] = v1.x;  cpout.cp[k+1] = v1.y;  cpout.cp[k+2] = v1.z;
36:       break;
37:   case 4:
38:       v0 = vec4 ( cpin.cp[k], cpin.cp[k+1], cpin.cp[k+2], cpin.cp[k+3] );
39:       v1 = cptr.tr[i]*v0;
40:       cpout.cp[k] = v1.x;   cpout.cp[k+1] = v1.y;
41:       cpout.cp[k+2] = v1.z; cpout.cp[k+3] = v1.w;
42:       break;
```

```
43:   default:
44:       break;
45:     }
46:   }
47: } /*main*/
```

w $\mathbb{R}^4$, wykonuje mnożenie i zapisuje 3 albo wszystkie 4 współrzędne iloczynu w tablicy `cpout.cp`.

Mając przygotowany do pracy (tj. skompilowany i złączony oraz wybrany za pomocą procedury `glUseProgram`) program szaderów z opisanym tu szaderem obliczeniowym, po nadaniu stosownych wartości potrzebnym zmiennym jednolitym i przywiązaniu właściwych buforów do odpowiednich punktów dowiązania, uruchomimy obliczenia, wywołując procedurę `glDispatchCompute`. Do przekształcenia n punktów w aplikacji drugiej H używamy jednowymiarowej grupy roboczej ($y_{\mathrm{GWS}} = z_{\mathrm{GWS}} = y_{\mathrm{LWS}} = z_{\mathrm{LWS}} = 1$) (zobacz s. 9.29). Zakres x_{LWS} indeksów `x` grupy lokalnej jest w kwalifikatorze `layout` w linii 3 na listingu 21.10 ustalony na 128, czyli indeksy `x` wypełniają zakres od 0 do 127, natomiast zakresy indeksów `y` i `z` mają domyślne długości 1. Zakres x_{GWS} indeksów `x` globalnej grupy roboczej musi być równy zaokrąglonemu *w górę* ilorazowi n/x_{LWS}. Numer punktu przekształcanego przez dany wątek jest równy x_{GID}. Jako się rzekło, znajdziemy go w zmiennej wbudowanej `gl_GlobalInvocationID.x`.

Dzięki identycznej budowie bloków `CPoints` i `CPointsOut` możemy zrealizować następujący pomysł: tablice oryginalnych punktów kontrolnych czajnika i torusa umieścimy w odpowiednich buforach w pamięci GPU. Ponadto utworzymy dodatkowe bufory o tych samych wielkościach z przeznaczeniem na punkty przekształcone. Dokonując artykulacji łańcucha, zapamiętamy odpowiednie macierze w buforze, który stanie się blokiem `Tr`. Wykonując postprocesing czajnika lub torusa, przywiążemy bufor z jego punktami kontrolnymi jako blok zmiennych jednolitych `CPoints` i podstawimy bufor na przekształcone punkty jako bufor magazynowy `CPointsOut`. Rysując następnie czajnik lub torus, bufor z przekształconymi punktami przywiążemy jako blok zmiennych jednolitych `CPoints` dla szaderów rozdrabniania, które obliczają punkty płatów Béziera. W ten sposób nie musimy wprowadzać żadnych zmian w szaderach uruchomionych we wcześniejszych wersjach aplikacji[6].

[6] Aby ten pomysł działał, układy bloku zmiennych jednolitych i bloku magazynowego muszą być identyczne, np. domyślne (`shared`), tak jak w tej aplikacji. Nie wolno zmienić tylko jednego z nich, albo np. podać jednemu blokowi kwalifikator układu `std140`, a drugiemu `std430`. Oczywiście we wszystkich programach szaderów układ bloku `CPoints` też musi być taki sam.

Zmiany w aplikacji

Największej zmianie uległ sposób tworzenia sceny, który obejmuje teraz tworzenie łańcucha kinematycznego w dodatku do wyświetlanych obiektów, czyli czajnika, torusa i lustra. Listing 21.11 przedstawia procedurę budowania łańcucha, przy czym procedury wywoływane przez tę procedurę lub przypisywane jako metody obiektom łańcucha są opisane dalej.

Na początku są tworzone zespoły płatów Béziera opisujące czajnik i torus; zmiany w procedurach, które to robią, są opisane dalej. W linii 14 alokujemy strukturę łańcucha, deklarując, że ma on mieć 2 obiekty, 10 członów, 4 referencje, 9 krawędzi i 9 parametrów artykulacji. Następnie w liniach 15–21 tworzone są dwa obiekty łańcucha; pierwszy będzie czajnikiem, a drugi torusem. W liniach 15 i 18–19 tworzone są macierze E_l dla tych obiektów. Macierze te, razem z liczbami punktów kontrolnych i metodami obiektów, są przekazywane jako parametry procedury `kl_NewObject` wywoływanej w liniach 16 i 20. Opis metod będzie dalej.

Pętla w liniach 22–23 tworzy człony łańcucha. W liniach 24–27 tworzone są referencje obiektów; tablice `r0`, `r1` i `r2` (zadeklarowane statycznie na zewnątrz procedury `ConstructMyLinkage`) zawierają odpowiednio 210, 30 i 62 liczby całkowite — numery punktów kontrolnych czajnika związane odpowiednio z członami L_2, L_4 i L_6. Referencja utworzona w linii 27 wiąże wszystkie 49 punktów kontrolnych torusa z członem L_9, zatem ostatni parametr procedury `kl_NewObjRef` w tym przypadku jest wskaźnikiem pustym.

Wywołania procedury `kl_NewJoint` w liniach 28–36 tworzą pary kinematyczne łańcucha; dla każdej pary, która jest krawędzią grafu na rysunku 21.1 są tworzone dwie skierowane przeciwnie półkrawędzie. Ostatnie dwa parametry określają rodzaj pary kinematycznej i numer parametru artykulacji (wszystkie pary są proste, a więc mają po jednym parametrze).

W kolejnych instrukcjach są określane macierze ustalonych przekształceń: macierz A_0 przejścia od układu korzenia (którym w tej aplikacji jest zawsze człon 0) do układu świata oraz macierze F_j i B_j dla wszystkich półkrawędzi, dla których te macierze są inne niż macierz jednostkowa, przypisana podczas tworzenia par kinematycznych. Tworzone są macierze przesunięć o wektory $\mathbf{f}_1, \ldots, \mathbf{f}_7$ pokazane na rysunku 21.2. Przypominam, że procedura `kl_SetJointFtr` jednocześnie z przypisaniem macierzy F_j półkrawędzi identyfikowanej przez drugi parametr, przypisuje macierz $B_k = F_j^{-1}$ drugiej półkrawędzi danej pary. Dodatkowo, jeśli ostatni parametr nie jest zerem (schowanym pod nazwą `false`), to następują przypisania $B_j = B_k$ i $F_k = F_j$.

Listing 21.11: Budowanie łańcucha kinematycznego aplikacji

C

```
1: kl_linkage *mylinkage;
2: BezierPatchObjf *myteapots[2], *mytoruses[2];
3:
4: kl_linkage *ConstructMyLinkage ( void )
5: {
6: #define SCF (1.0/3.0)
7:   kl_linkage *lkg;
8:   int        obj[2], l[10], j[9];
9:   int        i;
10:  GLfloat    tra[16], trb[16], trc[16];
11:
12:  ConstructMyTeapot ( myteapots );
13:  ConstructMyTorus ( mytoruses );
14:  if ( (lkg = kl_NewLinkage ( 2, 10, 4, 9, 9, NULL )) ) {
15:    M4x4Scalef ( tra, SCF, SCF, SCF*4.0/3.0 );
16:    obj[0] = kl_NewObject ( lkg, 3, 306, tra, (void*)myteapots,
17:                NULL, LKTransformBP, LKPostprocessBP, NULL, NULL );
18:    M4x4Scalef ( tra, 0.1, 0.1, 0.1 );  M4x4RotateXf ( trb, 0.5*PI );
19:    M4x4Multf ( trc, trb, tra );
20:    obj[1] = kl_NewObject ( lkg, 4, 49, trc, (void*)mytoruses,
21:                NULL, LKTransformBP, LKPostprocessBP, NULL, NULL );
22:    for ( i = 0; i < 10; i++ )
23:      l[i] = kl_NewLink ( lkg );
24:    kl_NewObjRef ( lkg, l[2], obj[0], 210, r0 );
25:    kl_NewObjRef ( lkg, l[4], obj[0], 30, r1 );
26:    kl_NewObjRef ( lkg, l[6], obj[0], 62, r2 );
27:    kl_NewObjRef ( lkg, l[9], obj[1], 49, NULL );
28:    j[0] = kl_NewJoint ( lkg, l[0], l[1], KL_ART_ROT_Z, 0 );
29:    j[1] = kl_NewJoint ( lkg, l[1], l[2], KL_ART_ROT_Y, 1 );
30:    j[2] = kl_NewJoint ( lkg, l[2], l[3], KL_ART_ROT_Y, 2 );
31:    j[3] = kl_NewJoint ( lkg, l[3], l[4], KL_ART_ROT_Y, 3 );
32:    j[4] = kl_NewJoint ( lkg, l[2], l[5], KL_ART_ROT_Y, 4 );
33:    j[5] = kl_NewJoint ( lkg, l[5], l[6], KL_ART_ROT_Y, 5 );
34:    j[6] = kl_NewJoint ( lkg, l[2], l[7], KL_ART_ROT_Y, 6 );
35:    j[7] = kl_NewJoint ( lkg, l[7], l[8], KL_ART_ROT_Y, 7 );
36:    j[8] = kl_NewJoint ( lkg, l[8], l[9], KL_ART_ROT_Z, 8 );
37:    M4x4Translatef ( lkg->current_root_tr, 0.0, 0.0, -0.6 );
38:    M4x4Translatef ( tra, -0.43, 0.0, 0.92 );  /* f_1 */
39:    kl_SetJointFtr ( lkg, j[1], tra, true );
40:    M4x4Translatef ( tra, 0.78, 0.0, 0.59 );   /* f_2 */
41:    kl_SetJointFtr ( lkg, j[2], tra, true );
42:    M4x4Translatef ( tra, 0.78, 0.0, 0.91 );   /* f_3 */
```

```
43:     kl_SetJointFtr ( lkg, j[3], tra, true );
44:     M4x4Translatef ( tra, 0.6, 0.0, 1.0 );     /* f_4 */
45:     kl_SetJointFtr ( lkg, j[4], tra, true );
46:     M4x4Translatef ( tra, -0.6, 0.0, 1.0 );    /* f_5 */
47:     kl_SetJointFtr ( lkg, j[5], tra, true );
48:     M4x4Translatef ( tra, 0.52, 0.0, 0.97 );   /* f_6 */
49:     kl_SetJointFtr ( lkg, j[6], tra, false );
50:     M4x4Translatef ( tra, 0.52, 0.0, 0.0 );    /* f_7 */
51:     kl_SetJointFtr ( lkg, j[7], tra, false );
52:     PrepareKLBuffers ();
53:   }
54:   else
55:     ExitOnError ( "ConstructMyLinkage" );
56:   return lkg;
57: #undef SCF
58: } /*ConstructMyLinkage*/
```

W linii 52 jest wywoływana procedura, która dla obu obiektów (czajnika i torusa) alokuje bufory w pamięci GPU, w których będą przechowywane bloki zmiennych jednolitych `Tr` i (tylko dla czajnika) `TrInd` dla szadera z listingu 21.10. Procedura ta jest pokazana dalej, na listingu 21.13, natomiast na listingu 21.12 są pokazane zmiany procedury `ConstructMyTeapot` w porównaniu z procedurą z listingu 17.8 (niektóre z tych zmian zostały wprowadzone w aplikacji drugiej G). Tworzymy *dwie* struktury `BezierPatchObjf`; pierwszą przez wywołanie procedury `ConstructTheTeapot` — odpowiedni bufor w pamięci GPU zawiera oryginalne punkty kontrolne czajnika. Druga struktura jest alokowana za pomocą procedury `malloc`; zawiera ona kopie wszystkich pól pierwszej struktury (w tym identyfikatory buforów) z wyjątkiem pola `teapots[1]->buf[1]`, któremu (w linii 16) przypisywany jest identyfikator *nowego* bufora. Ten bufor będzie zawierał punkty kontrolne czajnika przekształcone w wyniku artykulacji. W linii 21 polu `dim` w bloku zmiennych jednolitych `CPoints`, którym stanie się ten bufor, przypisujemy liczbę współrzędnych każdego punktu w tablicy `cp` (czyli 3). Aby narysować czajnik poddany artykulacji, wywołamy procedurę `DrawBezierPatches` z parametrem `teapots[1]`. Analogiczne zmiany trzeba wprowadzić do procedury `ConstructMyTorus`.

Zadaniem procedury `PrepareKLBuffers` pokazanej na listingu 21.13 jest utworzenie buforów dla bloków zmiennych jednolitych wykorzystywanych przez szader artykulacji. Potrzebne są dwa bufory. W pierwszym z nich będzie blok `Tr` z tablicą macierzy opisujących przekształcenia, którym należy poddać punkty obiektów związane z poszczególnymi członami. W drugim buforze jest blok `TrInd`

Listing 21.12: Procedura ConstructMyTeapot

C

```
1: void ConstructMyTeapot ( BezierPatchObjf *teapots[2] )
2: {
3:   const GLfloat MyColour[4] = { 1.0, 1.0, 1.0, 1.0 };
4:         ........ /* deklaracje zmiennych bez zmian */
5:
6:   teapots[0] = ConstructTheTeapot ( MyColour );
7:   teapots[1] = malloc ( sizeof(BezierPatchObjf) );
8:   if ( teapots[0] && teapots[1] ) {
9:     SetupMaterial ( 0, ambr, diffr, specr, shn, wa, we );
10:    teapots[0]->buf[3] =
11:        NewUniformBlockObject ( 32*4*sizeof(GLfloat), txcbp );
12:    glBufferSubData ( GL_UNIFORM_BUFFER, txcofs[0],
13:                      32*4*sizeof(GLfloat), txc );
14:    SetBezierPatchOptions ( teapots[0], BezNormals, TessLevel );
15:    memcpy ( teapots[1], teapots[0], sizeof(BezierPatchObjf) );
16:    glGenBuffers ( 1, &teapots[1]->buf[1] );
17:    glBindBuffer ( GL_UNIFORM_BUFFER, teapots[1]->buf[1] );
18:    glBufferData ( GL_UNIFORM_BUFFER, cpbofs[1]+306*3*sizeof(GLfloat),
19:                   NULL, GL_DYNAMIC_DRAW );
20:    glBufferSubData ( GL_UNIFORM_BUFFER, cpbofs[0],
21:                      sizeof(GLint), &teapots[1]->dim );
22:    ExitIfGLError ( "ConstructMyTeapot" );
23:  }
24:  else
25:    ExitOnError ( "ConstructMyTeapot" );
26: } /*ConstructMyTeapot*/
```

z numerami przekształceń (tj. indeksami do tablicy `cptr.tr`), którym należy poddać poszczególne punkty kontrolne czajnika; nie potrzebujemy analogicznego bufora dla torusa, bo *wszystkie* jego punkty mają być poddawane *temu samemu* przekształceniu (i szader artykulacji jest do tego dostosowany). Czajnik jest obiektem numer 0 w łańcuchu. Tablica zaalokowana w linii 9 jest w linii 10 wypełniana zerami, bo nie wszystkie punkty podane w reprezentacji czajnika są potrzebne[7]. W pętli w liniach 11–15 wyszukiwane są referencje obiektu 0, czyli czajnika. Dla referencji o numerze `r`, w wewnętrznej pętli, numer `r` jest wpisywany do tablicy w miejscach, których numery to numery punktów podane w referencji. Właśnie numery referencji będą indeksami do tablicy macierzy przekształceń.

[7]Zobacz przypis na s. 13.27. Można oczywiście „wyczyścić” dane, ale zawsze warto tak napisać program, aby dobrze działał dla danych „niewyczyszczonych”. Dzięki wypełnieniu tablicy zerami, każdemu zbędnemu punktowi w tablicy `cptrind.trind` będzie odpowiadał indeks 0, zatem przekształcając je, szader nie będzie czytał macierzy przekształcenia spoza tablicy `cptr.tr`.

Listing 21.13: Procedura PrepareKLBuffers

C

```c
1: static GLuint lktrbuf[2];
2:
3: void PrepareKLBuffers ( kl_linkage *lkg )
4: {
5:   GLuint *cpi;
6:   int    r, i, nv;
7:
8:   nv = lkg->obj[0].nvert;
9:   if ( (cpi = malloc ( nv*sizeof(GLuint) )) ) {
10:     memset ( cpi, 0, nv*sizeof(GLuint) );
11:     for ( r = 1; r < lkg->norefs; r++ )
12:       if ( lkg->oref[r].on == 0 ) {
13:         for ( i = 0; i < lkg->oref[r].nv; i++ )
14:           cpi[lkg->oref[r].vn[i]] = r;
15:       }
16:     glGenBuffers ( 2, lktrbuf );
17:     glBindBuffer ( GL_UNIFORM_BUFFER, lktrbuf[0] );
18:     glBufferData ( GL_UNIFORM_BUFFER, lkg->norefs*16*sizeof(GLfloat),
19:                    NULL, GL_DYNAMIC_DRAW );
20:     glBindBuffer ( GL_UNIFORM_BUFFER, lktrbuf[1] );
21:     glBufferData ( GL_UNIFORM_BUFFER, nv*sizeof(GLuint), cpi,
22:                    GL_STATIC_DRAW );
23:     ExitIfGLError ( "PrepareKLBuffers" );
24:     free ( cpi );
25:   }
26:   else
27:     ExitOnError ( "PrepareKLBuffers" );
28: } /*PrepareKLBuffers*/
```

W linii 16 alokowane są identyfikatory buforów. Pierwszy z nich ma pomieścić tyle macierzy, ile łańcuch ma referencji; wielkość bufora jest ustalana w liniach 18–19, przy czym nie są przesyłane żadne dane do bufora (jeszcze ich nie ma), ale ponieważ macierze będą wielokrotnie zmieniane, ostatni parametr procedury `glBufferData` to `GL_DYNAMIC_DRAW`. Natomiast zawartość drugiego bufora, tj. numery przekształceń dla punktów kontrolnych czajnika, nie będzie zmieniana. Dlatego w pierwszym (i jedynym) wywołaniu procedury `glBufferData` dla tego bufora podajemy parametr `GL_STATIC_DRAW`.

Na listingu 21.14 są pokazane procedury, które podczas tworzenia obiektów łańcucha stają się ich metodami. Parametr `init` podany procedurze `kl_NewObject` podczas tworzenia obiektów czajnika i torusa był wskaźnikiem

pustym, bo nie było potrzeby alokowania (w pamięci CPU) tablic `vert` i `tvert` dla tych obiektów. Podczas domyślnej inicjalizacji wartość parametru `usrdata` procedury `kl_NewObject` jest przypisywana polu `usrdata` struktury obiektu (zobacz listing 21.3, linia 34). Procedura `LKTransformBP` jest metodą `transform` obu obiektów; jest ona wywoływana przez procedurę `kl_Articulate`, a jej parametry to kolejno wskaźnik struktury obiektu (`obj`), numer referencji (`r`), macierz $A_i E_l$ (`tr`), liczba punktów obiektu wymienionych w referencji (`nv`) i wskaźnik tablicy numerów tych punktów (`vn`, dla torusa jest to wskaźnik pusty). W naszym przypadku metoda nie przekształca tych punktów, tylko przesyła współczynniki macierzy do bufora, do miejsca określonego przez numer referencji.

Po zakończeniu przeszukiwania metodą DFS grafu łańcucha wszystkie macierze w buforze, który będzie blokiem zmiennych jednolitych `Tr`, mają aktualne wartości. Wtedy kolejno dla każdego obiektu procedura `kl_Articulate` wywołuje jego metodę `postprocess`, czyli dla torusa i czajnika procedurę `LKPostprocess`.

W linii 15 procedura `LKPostprocess` wybiera program składający się z szadera artykulacji. W linii 16 zmiennej `bp` przypisywany jest adres początku tablicy `myteapots` albo `mytoruses`; przypomnijmy, że pierwsze elementy tych tablic wskazują struktury reprezentujące *nieprzekształcony* zespół płatów Béziera reprezentujący czajnik albo torus. W linii 17 do punktu dowiązania (w celu `GL_UNIFORM_BUFFER`), którego numer jest wartością zmiennej `ctrbbp`, przywiązywany jest bufor z macierzami przekształceń. W linii 18 do punktu dowiązania o numerze `cpbbp` w tym samym celu jest przywiązywany bufor z oryginalnymi punktami kontrolnymi, a w linii 19 do punktu dowiązania 0 w celu `GL_SHADER_STORAGE_BUFFER` jest przywiązywany bufor, do którego szader artykulacji wpisze wyniki swojego działania.

Jeśli mają być przekształcane punkty czajnika, to w linii 21 do punktu dowiązania o numerze `ctribbp` przywiązywany jest bufor z numerami przekształceń punktów czajnika. Instrukcje w liniach 22 i 29 powodują przypisanie zmiennej jednolitej `trnum` wartości -1, aby szader poddawał punkty różnym przekształceniom. Wszystkie punkty kontrolne torusa mają być poddane temu samemu przekształceniu, numer 3, bo odpowiednia referencja dla torusa ma numer 3 i taką wartość otrzyma zmienna `trnum` przed przekształcaniem punktów torusa.
W linii 30 zmiennej jednolitej `ncp` jest przypisywana liczba punktów obiektu. Po tych wszystkich przygotowaniach, w linii 31 wykonujemy program szaderów, który przekształci te punkty. Wielkość globalnej grupy roboczej jest ustalana zgodnie z regułą podaną wcześniej.

Listing 21.14: Metody obiektów łańcucha

C

```
1: static void LKTransformBP ( kl_object *obj, int refn, GLfloat *tr,
2:                             int nv, int *vn )
3: {
4:   glBindBuffer ( GL_UNIFORM_BUFFER, lktrbuf[0] );
5:   glBufferSubData ( GL_UNIFORM_BUFFER, refn*16*sizeof(GLfloat),
6:                     16*sizeof(GLfloat), tr );
7:   ExitIfGLError ( "LKTransformBP" );
8: } /*LKTransformBP*/
9:
10: static void LKPostprocessBP ( kl_object *obj )
11: {
12:   BezierPatchObjf **bp;
13:   GLint           ntr;
14:
15:   glUseProgram ( program_id[5] );
16:   bp = (BezierPatchObjf**)obj->data1;
17:   glBindBufferBase ( GL_UNIFORM_BUFFER, ctrbbp, lktrbuf[0] );
18:   glBindBufferBase ( GL_UNIFORM_BUFFER, cpbbp, bp[0]->buf[1] );
19:   glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 0, bp[1]->buf[1] );
20:   if ( bp[0] == myteapots[0] ) {
21:     glBindBufferBase ( GL_UNIFORM_BUFFER, ctribbp, lktrbuf[1] );
22:     ntr = -1;
23:   }
24:   else if ( bp[0] == mytoruses[0] )
25:     ntr = 3;
26:   else
27:     ExitOnError ( "LKPostprocessBP" );
28:   glUniform1i ( dim_loc, bp[0]->dim );
29:   glUniform1i ( trnum_loc, ntr );
30:   glUniform1i ( ncp_loc, (GLint)obj->nvert );
31:   glDispatchCompute ( (obj->nvert+127)/128, 1, 1 );
32:   glMemoryBarrier ( GL_UNIFORM_BARRIER_BIT );
33:   ExitIfGLError ( "LKPostprocessBP" );
34: } /*LKPostprocessBP*/
```

Wywołanie procedury glMemoryBarrier w linii 32 ma na celu zapewnienie, że punkty przekształcone przez szader obliczeniowy zostaną zapisane do pamięci przed rozpoczęciem rysowania. Parametr tej procedry jest maską bitową, która w tym przypadku wskazuje konieczność zapisania w pamięci danych, które będą (przez program rysowania płatów Béziera) czytane z bloku zmiennych jednolitych.

Na listingu 21.15 są pokazane zmienione fragmenty procedury LoadMyShaders. Tablice shader_id i program_id, w których są zapamiętywane identyfikatory

Listing 21.15: Zmiany w procedurze LoadMyShaders

C

```
1: void LoadMyShaders ( void )
2: {
3:   static const char *filename[] =
4:     { "app2g0.glsl.vert", ........ "app2g4.glsl.vert", "app2h.glsl.comp" };
5:   static const GLuint shtype[] =
6:     { GL_VERTEX_SHADER, ........ GL_VERTEX_SHADER, GL_COMPUTE_SHADER };
7:    ........
8:   static const GLchar NormalSourceName[] = "NormalSource";
9:   static const GLchar *UCompTrNames[] = { "Tr", "Tr.tr" };
10:  static const GLchar *UCompTrIndNames[] = { "TrInd", "TrInd.ind" };
11:  static const GLchar DimName[] = "dim";
12:  static const GLchar NcpName[] = "ncp";
13:  static const GLchar TrNumName[] = "trnum";
14:
15:  GLint  i;
16:  GLuint shid[5];
17:
18:  for ( i = 0; i < 13; i++ )
19:    shader_id[i] = CompileShaderFiles ( shtype[i], 1, &filename[i] );
20:  program_id[0] = LinkShaderProgram ( 5, shader_id );
21:    ........
22:  program_id[5] = LinkShaderProgram ( 1, &shader_id[12] );
23:  GetAccessToUniformBlock ( program_id[0], 6, &UTBNames[0],
24:                            &trbi, &trbsize, trbofs, &trbbp );
25:    ........
26:  LightProcInd = LambertProcInd;
27:  GetAccessToUniformBlock ( program_id[5], 1, &UCompTrNames[0],
28:                            &ctrbi, &i, &i, &ctrbbp );
29:  GetAccessToUniformBlock ( program_id[5], 1, &UCompTrIndNames[0],
30:                            &ctribi, &i, &i, &ctribbp );
31:  dim_loc = glGetUniformLocation ( program_id[5], DimName );
32:  ncp_loc = glGetUniformLocation ( program_id[5], NcpName );
33:  trnum_loc = glGetUniformLocation ( program_id[5], TrNumName );
34:  trbuf = NewUniformBlockObject ( trbsize, trbbp );
35:    ........
36:  AttachUniformBlockToBP ( program_id[4], UTBNames[0], trbbp );
37:  AttachUniformBlockToBP ( program_id[5], UCPNames[0], cpbbp );
38:  ExitIfGLError ( "LoadMyShaders" );
39: } /*LoadMyShaders*/
```

szaderów i programów, należy odpowiednio wydłużyć. W linii 22 odbywa się kompilacja i łączenie programu z szaderem artykulacji. Dostęp do jego bloków zmiennych jednolitych i do zmiennych jednolitych w bloku domyślnym tego programu uzyskujemy w liniach 27–33. W linii 37 blok zmiennych jednolitych CPoints tego programu wiążemy z punktem dowiązania, którego numer jest wartością zmiennej cpbbp; wszystkie pozostałe programy szaderów też mają swoje bloki CPoints przywiązane do tego punktu. Natomiast w tej procedurze nie ma instrukcji mających uzyskać dostęp do pól bloku magazynowego CPointsOut. Przesunięcia pól w tym bloku są takie same jak przesunięcia pól w bloku zmiennych jednolitych CPoints, zaś punkt dowiązania odpowiedniego bufora w celu GL_SHADER_STORAGE_BLOCK ma z góry ustalony numer 0.

Listing 21.16: Procedura InitMyObject

C

```
1: void InitMyObject ( void )
2: {
3:   TimerInit ();
4:   memset ( &trans, 0, sizeof(TransBl) );
5:   memset ( &light, 0, sizeof(LightBl) );
6:   M4x4Identf ( ident_matrix );
7:   SetModelMatrix ( ident_matrix, ident_matrix );
8:   InitViewMatrix ();
9:   ConstructBezierPatchDomain ();
10:  mylinkage = ConstructMyLinkage ();
11:  ConstructMirrorFBO ();
12:  ConstructMirrorVAO ();
13:  InitLights ();
14:  LoadMyTextures ();
15:  ArticulateMyLinkage ();
16: } /*InitMyObject*/
```

Zmiany procedury inicjalizacji są pokazane na listingu 21.16. Za macierz przejścia od układu modelu do układu świata, wykorzystywaną przez szadery w programach rysujących scenę, jest w liniach 7–8 przyjmowana (raz na zawsze) macierz jednostkowa; za przekształcenia obiektów do układu świata będzie odpowiadał szader obliczeniowy artykulacji. Procedurę ConstructMyLinkage, opisaną wcześniej, trzeba wywołać *zamiast* wcześniej wywoływanych w tym miejscu procedur tworzących reprezentacje czajnika i torusa w pamięci GPU. Wreszcie, trzeba wywołać procedurę ArticulateMyLinkage (listing 21.17), która nada wartości początkowe parametrom artykulacji i określi odpowiednie położenia obiektów.

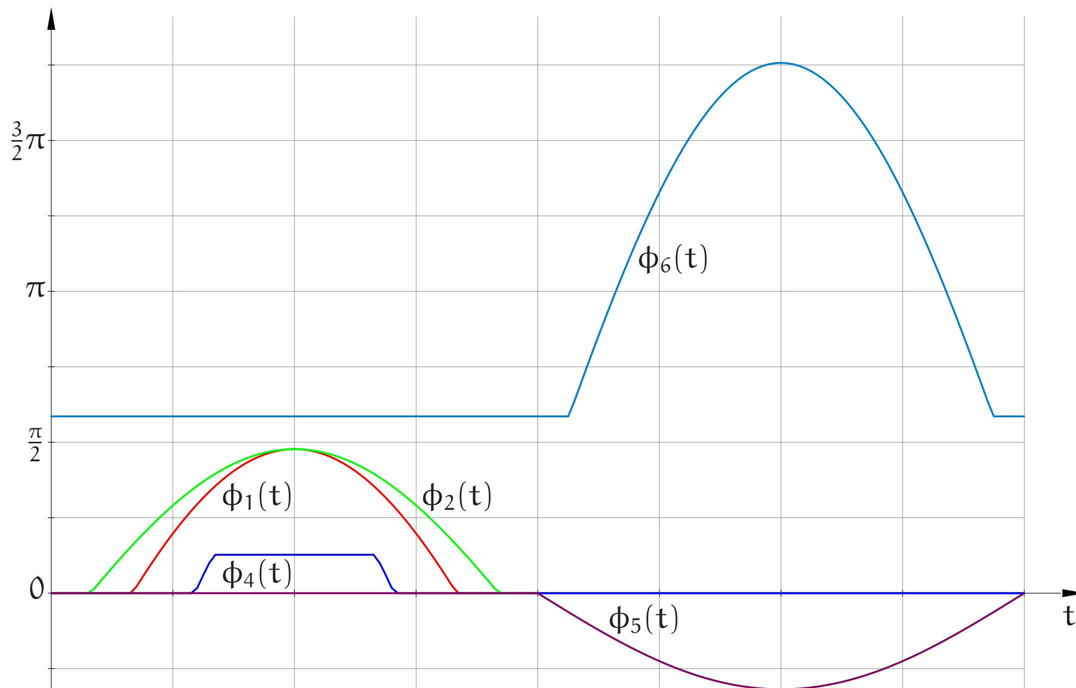
Procedura `ArticulateMyLinkage` oblicza wartości parametrów artykulacji, które są animowane, tj. zmieniające się w czasie. Przypomnę, że w zmiennej `teapot_time0` jest zapamiętana chwila ostatniego uruchomienia obracania czajnika przez naciśnięcie klawisza spacji, a w zmiennej `teapot_rot_angle0` jest pamiętany kąt obrotu czajnika w tym momencie (w tej aplikacji jest to kąt obrotu członu L_1 łańcucha kinematycznego wokół osi z układu członu L_0, która pokrywa się z osią z układu świata). W zmiennej `teapot_rot_angle` jest bieżący kąt obrotu czajnika; jest on przypisywany tej zmiennej, ponieważ kąt obrotu w chwili zatrzymania (przez ponowne naciśnięcie klawisza spacji) musi być zapamiętany (przez procedurę `ToggleAnimation`) w zmiennej `teapot_rot_angle0`, aby od tego kąta móc wznowić obracanie.

Listing 21.17: Procedury `ArticulateMyLinkage` i `IdleFunc`

C

```c
1: void ArticulateMyLinkage ( void)
2: {
3:   GLfloat par[9];
4:
5:   teapot_rot_angle = teapot_rot_angle0 + 0.78539816 * TimerToc ();
6:   par[0] = animate ? teapot_rot_angle : teapot_rot_angle0;
7:   par[1] = TeapotRotAngle2 ( app_time );
8:   par[2] = SpoutAngle ( app_time );
9:   par[3] = -0.5*par[2];
10:  par[4] = LidAngle1 ( app_time );
11:  par[5] = LidAngle2 ( app_time );
12:  par[7] = -(par[6] = TorusRotAngle1 ( app_time ));
13:  par[8] = -2.0*PI*app_time;
14:  kl_SetArtParam ( mylinkage, 0, 9, par );
15:  kl_Articulate ( mylinkage );
16: } /*ArticulateMyLinkage*/
17:
18: void IdleFunc ( void )
19: {
20:   ArticulateMyLinkage ( app_time );
21:   redraw = true;
22: } /*IdleFunc*/
```

Nie zamieszczam tu treści procedur obliczających wartości poszczególnych parametrów, jako że nie ma w nich niczego fascynującego; rysunek 21.3 przedstawia wykresy funkcji obliczanych przez te procedury. Funkcje te są okresowe, ich okres to 8s., przy czym podziałka na osi czasu (poziomej) ma długość jednej sekundy. Funkcje są (po pewnej liczbie eksperymentów) dobrane tak, aby torus „wskakując do” i „wyskakując z” czajnika, nie wchodził w kolizję



Rysunek 21.3: Wykresy parametrów artykulacji w funkcji czasu

z pokrywką ani korpusem czajnika. Zwracam też uwagę, że parametr ϕ_3 jest równy $-\frac{1}{2}\phi_2$, a ponadto $\phi_7 = -\phi_6$. Ostatnia równość zapewnia, że oś, wokół której obraca się torus, pozostaje równoległa do osi z członu L_2, z którym jest związany korpus czajnika. Wywołana w linii 14 na listingu 21.17 procedura `kl_SetArtParam`, która nie była opisana wcześniej, przesyła 9 parametrów artykulacji (czyli wszystkie) z tablicy par do tablicy `mylinkage->artp`, od miejsca 0 w tej tablicy.

Jedyna zmiana procedur rysowania torusa i czajnika polega na przekazaniu parametru `myteapots[1]` albo `mytoruses[1]` procedurze `DrawBezierPatches` i `DrawBezierNets`.

Na listingu 21.18 jest procedura sprzątania; ma ona do skasowania więcej szaderów i programów. Ponadto, o ile struktury wskazywane przez zmienne `myteapots[0]` i `mytoruses[0]` likwidujemy tak jak dotychczas przy użyciu procedury `DeleteBezierPatches`, która zwolni bufory w pamięci GPU o identyfikatorach pamiętanych w tych strukturach, sprzątnięcie struktur reprezentujących te obiekty po przekształceniach jest robione „ręcznie”: te reprezentacje mają tylko po jednym nowym buforze, o identyfikatorze przechowywanym w polu `buf[1]`. Pamiętamy też o sprzątnięciu łańcucha kinematycznego i buforów w pamięci GPU zaalokowanych przez procedurę `PrepareKLBuffers`.

Listing 21.18: Sprzątanie

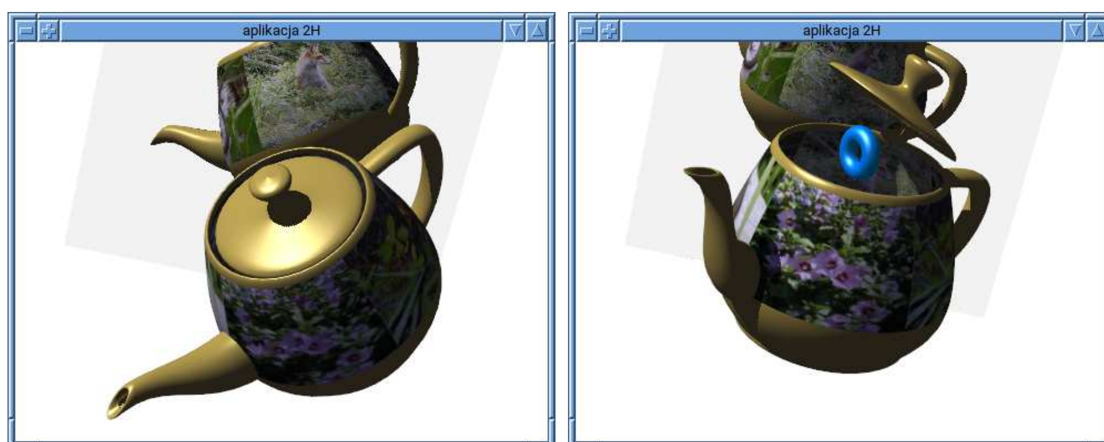
C

```
1: void Cleanup ( void )
2: {
3:   int i;
4:
5:   glUseProgram ( 0 );
6:   for ( i = 0; i < 13; i++ )
7:     glDeleteShader ( shader_id[i] );
8:   for ( i = 0; i < 6; i++ )
9:     glDeleteProgram ( program_id[i] );
10:  glDeleteBuffers ( 1, &trbuf );
11:  glDeleteBuffers ( 1, &lsbuf );
12:  DeleteBezierPatchDomain ();
13:  DeleteBezierPatches ( myteapots[0] );
14:  glDeleteBuffers ( 1, &myteapots[1]->buf[1] );
15:  free ( myteapots[1] );
16:  DeleteBezierPatches ( mytoruses[0] );
17:  glDeleteBuffers ( 1, &mytoruses[1]->buf[1] );
18:  free ( mytoruses[1] );
19:  glDeleteBuffers ( 2, lktrbuf );
20:  kl_DestroyLinkage ( mylinkage );
21:  for ( i = 0; i < 2; i++ )
22:    DeleteMaterial ( i );
23:  glDeleteTextures ( 1, &mytexture );
24:  DestroyMirrorVAO ();
25:  DestroyMirrorFBO ();
26:  DestroyShadowFBO ();
27:  glfwDestroyWindow ( mywindow );
28:  glfwTerminate ();
29: } /*Cleanup*/
```

Ćwiczenia

1. Zmodyfikuj aplikację tak, aby obraz czajnika w lustrze powtarzał ruchy czajnika z opóźnieniem o np. 1/4 sekundy.

2. Nawiązując do ćwiczenia 14.4, napisz szader obliczeniowy obliczający punkty kontrolne produktu sferycznego dwóch krzywych reprezentowanych przez łamane. Zastosuj w nim dwuwymiarową grupę roboczą. Użyj tego szadera zamiast procedury `EnterRSphericalProduct` z listingu 15.1 do utworzenia reprezentacji torusa w aplikacji.



Rysunek 21.4: Okno aplikacji drugiej H

A. Słowniki

Słownik TLS-ów i CzLS-ów

AMD — Advanced Micro Devices, jeden z dwóch wiodących producentów CPU i jeden z dwóch wiodących producentów GPU.

ANSI — American National Standards Institute, organizacja, która opracowała kanoniczny standard języka C. Staram się go trzymać.

API — *application programming interface*, po włosku słowo *api* oznacza pszczoły.

ARB — OpenGL Architecture Review Board, komitet, który w latach 1992–2006 odpowiadał za rozwój standardu OpenGL, później wszedł w skład Khronos Group.

CCD – *charge-coupled device*, współczesna namiastka szklanej płyty lub błony z octanu celulozy, pokrytej emulsją z bromkiem srebra.

CCW — *counterclockwise*, przeciwnie do ruchu wskazówek zegara.

CPU — *central processing unit*, „główny” procesor.

CUDA — *Compute Unified Device Architecture*, język programowania GPU opracowany przez firmę NVIDIA. Jest on podobniejszy do C niż GLSL i w zastosowaniach niezwiązanych z potokiem przetwarzania grafiki wydaje się mieć większą siłę wyrazu. Zamiast szaderów obliczeniowych napisanych w GLSL, można napisać odpowiedni program w języku CUDA, ale jego zastosowanie ograniczone jest do komputerów wyposażonych w GPU z procesorami firmy NVIDIA.

CzLS — czteroliterowy skrót, np. CzLS (nie mylić z TLS).

DFS — *depth-first search*, przeszukiwanie grafu w głąb.

DMA — *direct memory access*, układy wejścia/wyjścia umożliwiające przesyłanie danych m.in. między pamięcią operacyjną CPU i pamięcią GPU znacznie szybciej niż może to czynić CPU.

DPI — *dots per inch*, jednostka rozdzielczości obrazu, liczba pikseli na cal.

DRI — *direct rendering interface*, zaimplementowana w bibliotece GLX możliwość przekazywania danych i poleceń między CPU i GPU z pominięciem protokołu komunikacyjnego systemu X Window, dla przyspieszenia tworzenia grafiki.

FBO — *framebuffer object*, obiekt bufora ramki.

GCC — *GNU compiler collection*, pakiet kompilatorów różnych języków programowania, w tym języka C.

GLEW — *OpenGL Extension Wrangler*, jedna z bibliotek udostępniających aplikacji adresy procedur OpenGL-a.

GLSL — *OpenGL shading language*, bohater tej książki.

GLU — *OpenGL utilities*, pomocnicza biblioteka OpenGL-a.

GLUT — *OpenGL Utility Toolkit*, historycznie pierwsza biblioteka z API dla interakcyjnych aplikacji OpenGL-a, umożliwiająca uniezależnienie aplikacji od systemu operacyjnego i systemu okien.

GNU — *GNU's not Unix*, TLS, w którym rekurencja służy kokieterii.

GPU — *graphics processing unit*, procesor grafiki.

GUI — *graphical user interface*, zestaw wyświetlanych na ekranie wihajstrów, które służą do interakcji użytkownika z programem. Zobacz też WIMP.

IBO — *index buffer object*, bufor z indeksami do tablicy wierzchołków, umożliwia wygodne rysowanie łamanych, taśm trójkątowych lub wachlarzy określonych przez ciąg wierzchołków, które mogą się powtarzać.

IEEE — Institute of Electrical and Electronics Engineers, organizacja, która opracowała m.in. standard arytmetyki zmiennopozycyjnej IEEE 754, obejmujący reprezentacje liczb *oraz* najważniejsze własności działań na nich. CPU realizują ten standard w pełni, natomiast GPU zazwyczaj tylko w ograniczonym zakresie, o czym trzeba wiedzieć.

KISS — *keep it simple, stupid!*, najważniejsza maksyma, która powinna zawsze przyświecać każdemu programiście. W praktyce, niestety, nie każdemu, nie zawsze i nie przyświeca.

KHR — Khronos Group, konsorcjum sprawujące obecnie opiekę nad standardem OpenGL.

LOD — *level of detail*, poziom szczegółowości modelu dostosowany do jego wielkości na obrazie. Nie należy rysować wentyli w kołach samochodu, jeśli obraz całego tego samochodu w narysowanym krajobrazie ma średnicę kilkunastu pikseli (co innego, jeśli obraz wentyla ma kilkanaście pikseli).

MIP — *multum in parvo*, wiele w niewielu, określenie techniki teksturowania (mipmapingu, *MIP-mapping*) użytej w rozdziale 17.

NDC — *normalized device coordinates*, układ współrzędnych kostki standardowej; w tym układzie współrzędne kartezjańskie punktów bryły widzenia leżą w przedziale $[-1, 1]$.

ODW — ostatnia działająca wersja, czyli program, który działał zanim postanowiliśmy go ulepszyć. Trzeba było zrobić kopię zapasową.

PBO — *pixel buffer object*, bufor z tablicą pikseli, a właściwie dowolnych danych reprezentowanych przez pojedyncze liczby, pary lub czwórki liczb, przetwarzanych przez szadery jako obraz (*image*).

PCF — *percentage closer filtering* ....

RAM — *random access memory*, pamięć o dostępie bezpośrednim, po angielsku *ram* to także baran.

RGB — *red, green, blue*, współrzędne w przestrzeni koloru.

RGBA — *red, green, blue, alpha*, współrzędne w przestrzeni koloru i kanał alfa, pomocniczy w tworzeniu obrazu.

SPIR — *Standard Portable Intermediate Representation*, binarny format częściowo skompilowanych programów dla GPU, można go używać do rozpowszechniania szaderów bez udostępniania ich kodów źródłowych.

SSBO — *shader storage buffer object*, obiekt bufora magazynowego.

TBO — *texture buffer object*, obiekt bufora tekstury.

TIFF — *tagged image file format*, format plików do zapisu obrazów rastrowych, niesamowicie elastyczny.

TIGA — *Texas Instruments Graphics Architecture*, standard grafiki zbudowany w latach 1990-tych wokół procesorów TMS 34010 i TMS 34020, które były pierwszymi możliwymi do zainstalowania w komputerach osobistych całkowicie programowalnymi GPU (choć wtedy ten TLS jeszcze nie istniał). Standard ten okazał się ślepą uliczką w rozwoju technologii, ale był w swoim czasie inspirujący.

TLS — trzyliterowy skrót, np. TLS (nie mylić z CzLS).

UBO — *uniform buffer object*, bufor z blokiem zmiennych jednolitych.

VAO — *vertex array object*, obiekt tablicy wierzchołków.

VBO — *vertex buffer object*, bufor przechowujący atrybuty wierzchołków, rejestrowany w VAO.

WIMP — *windows, icons, menus, pointers*, dawno używany CzLS, w krótkim czasie wyparty przez TLS GUI, bo jak ktoś zauważył, każdy wolałby być *gui* niż *wimp*.

WWW — *world-wide web*, w praktyce wysypisko wszelkich wiadomości, w którym potrzebne (i rzetelne) informacje bywają trudne do znalezienia.