Affine transformations

In **Cartesian** coordinates:

$$f(\boldsymbol{p}) = L\boldsymbol{p} + \boldsymbol{t},$$

L denotes a 3×3 matrix of the **linear part**, $\boldsymbol{t}$ is the **translation vector**.

In **homogeneous** coordinates the transformation is done by multiplication by the matrix

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

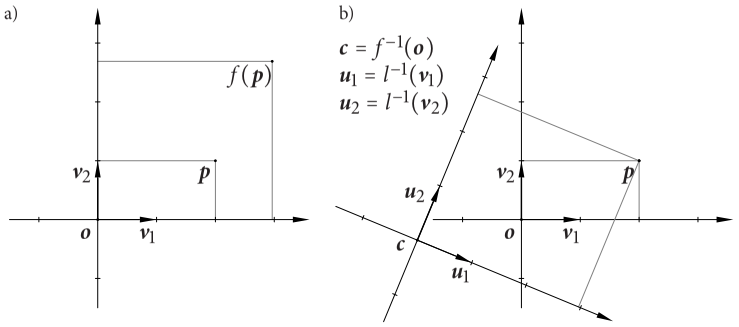
hence, the vector $A\boldsymbol{P}$ represents the point $f(\boldsymbol{p})$.

Both reprezentations are related to a specific system of Cartesian coordinates.

The straightforward and dual interpretation

The formula for the mapping f may be interpreted in two ways:

- The vector $A\boldsymbol{p} + \boldsymbol{t}$ is made of the Cartesian coordinates of a **new point**.
- The vector $A\boldsymbol{p} + \boldsymbol{t}$ is made of the Cartesian coordinates of the same point in a **new coordinate system**.



An affine transformation and its dual interpretation

The matrix of the affine transformation f may be rewritten in the form

$$A = \left[\begin{array}{cccc} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \boldsymbol{w}_3 & \boldsymbol{t} \\ 0 & 0 & 0 & 1 \end{array}\right].$$

Its columns are vectors of homogeneous coordinates of three free vectors and one point.

If the vectors $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, $\boldsymbol{w}_3$ are linearly independent, then, together with the *point* $\boldsymbol{t}$, they make a frame for a new Cartesian coordinate system.

The point $\boldsymbol{t}$ is the image of the origin $\boldsymbol{o}$ of the current system under the mapping f, and the vector $\boldsymbol{w}_i$ is the image of the unit vector $\boldsymbol{e}_i$ of the i-th axis under the linear mapping l described by the matrix $L = [\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3]$.

In dual interpretation: let $\boldsymbol{c} = f^{-1}(\boldsymbol{o})$, $\boldsymbol{u}_1 = l^{-1}(\boldsymbol{v}_1)$, $\boldsymbol{u}_2 = l^{-1}(\boldsymbol{v}_2)$, $\boldsymbol{u}_3 = l^{-1}(\boldsymbol{v}_3)$. The point $\boldsymbol{c}$ and the vectors $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, $\boldsymbol{u}_3$ make a frame for the coordinate system, to which we pass, i.e., $L\boldsymbol{p} + \boldsymbol{t}$ is the vector of Cartesian coordinates of our point in the new system.

Composing of affine transformations

Let A_1 and A_2 be matrices of the transformations f_1 and f_2, done in a sequence.

If both transformations are defined in **the same** coordinate system, then the composition $f = f_2 \circ f_1$, i.e., the maping given by $f(\boldsymbol{p}) = f_2(f_1(\boldsymbol{p}))$ is represented by the matrix A_2A_1.

If the transformations are given in the system **attached to the transformed object**, which moves or rotates together with the object, then in the original coordinate system the transformation f_1 is described by the matrix A_1, and the transformation f_2 in this system is described by the matrix $A_1A_2A_1^{-1}$.

Namely: A_1^{-1} represents the transformation to the system changed by f_1, A_2 describes the transformation f_2 in this system and A_1 describes the return to the original system. The composition of these transformations is described by the matrix

$$A_1A_2A_1^{-1}A_1 = A_1A_2.$$

The first method of composing affine transformations is useful when we turn the object to see it from various directions.

> *Za czym spod ich rąk wypełzła szara myszka cynowa,*
> *która puszczając pyszczkiem bańki mydlane, zarazem*
> *biegała po stole, a spod ogonka sypał się jej biały kredowy*
> *pył tak kunsztownie, iż powstał z tego kaligrafowany napis:*
> *A WIĘC NAPRAWDĘ NAS NIE KOCHACIE?*
>
> Stanisław Lem: *Cyberiada*

The second method is used in the so called **turtle graphics**. The turtle is walking on the plane, according to the commands like "turn by x degrees to the left" or "step forward by the distance y".

Transformations of the normal vector

The normal vector of a surface plays the crucial role in lighting models. Any object may be described in a system most convenient for the designer, but then all objects must be individually transformed to compose the entire scene to draw. Usually, each object must be transformed to a certain **world coordinate system**.

Points of objects are transformed directly, using the formula $f(\boldsymbol{p}) = L\boldsymbol{p} + \boldsymbol{t}$.

Free vectors are differences of points: if $\boldsymbol{v} = \boldsymbol{p}_1 - \boldsymbol{p}_2$, then the image of $\boldsymbol{v}$ is

$$f(\boldsymbol{p}_1) - f(\boldsymbol{p}_2) = L\boldsymbol{p}_1 + \boldsymbol{t} - (L\boldsymbol{p}_2 + \boldsymbol{t}) = L(\boldsymbol{p}_1 - \boldsymbol{p}_2) = L\boldsymbol{v}.$$

The normal vector of a surface is **not** a free vector. The proper way of transforming it may be obtained from the plane equation: $\pi = \{\, \boldsymbol{p} : \boldsymbol{n}^T(\boldsymbol{p} - \boldsymbol{p}_0) = 0 \,\}$.

The equation of the image of the plane π under the mapping f is

$$0 = \boldsymbol{m}^T(f(\boldsymbol{p}) - f(\boldsymbol{p}_0)) = \boldsymbol{m}^T L(\boldsymbol{p} - \boldsymbol{p}_0).$$

The equation will be satisfied if we take $\boldsymbol{m} = L^{-T}\boldsymbol{n}$, as in this case

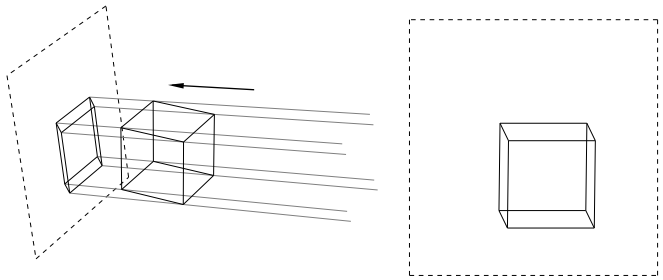
$$\boldsymbol{n}^T L^{-1} L(\boldsymbol{p} - \boldsymbol{p}_0) = \boldsymbol{n}^T(\boldsymbol{p} - \boldsymbol{p}_0) = 0.$$

Note that $L^{-T} = L$ if and only if the matrix L is orthogonal, i.e., if the mapping f is an **isometry**.

In general the matrix L may be used to transform normal vectors if f is a **geometric similarity**. Their lengths in lighting models are irrelevant, as most often the normal vector is **normalised** to obtain a unit vector of the same direction and orientation.

Projections

Parallel projections



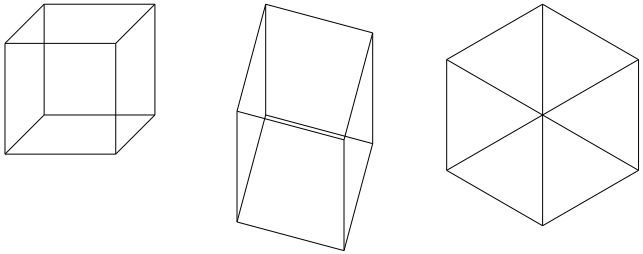
Particular cases of parallel projections are **orthogonal projections** and **axonometry**, used most often in technical drawing.

Axonometry is usually a skew projection, whose direction needs not be orthogonal to the projection plane.

By **Pohlke theorem**, for any four points $(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)$, not coplanar, and any coplanar points $(\boldsymbol{q}_0, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$ such that none three are collinear, there exist a projection plane and a projection direction such that the image of the four-tuple $(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)$ under the projection is similar to the four-tuple $(\boldsymbol{q}_0, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$.

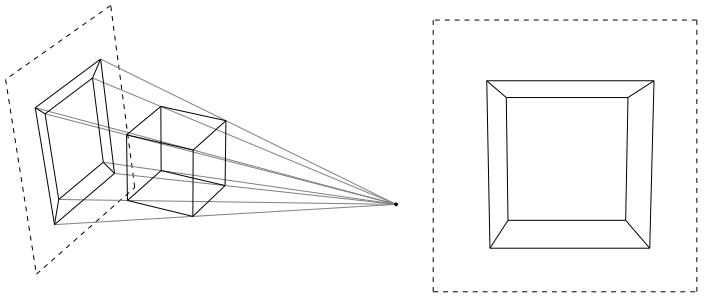
By this theorem, one can choose the image of the frame of the coordinate system on the plane in an (almost) arbitrary way.

The most often types of axonometry in technical drawings:

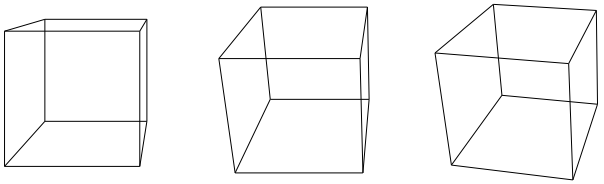


cavalier **military** **isometric**

Perspective projections



Perspective



one-point **two-point** **three-point**

Projections in OpenGL

The transformations applied to points (**vertices**) of objects in order to obtain a picture are consecutive changes of the coordinate systems.

- **The model coordinate system** is used to define the object; model coordinates of vertices are given on input of the rendering pipeline.

  For each object being drawn a different model system may be used.

- **The world coordinate system** is one common system to which all objects are transformed from their individual model systems.

  In this system usually calculations of lighting, shadows and object collisions are performed.

- **The viewer coordinate system** defines the position of the "camera" in the scene.

  Usually the transformation from the world system to the viewer system is an isometry. It positions the projection centre and defines the direction of optical axis of the camera.

- **The standard cube coordinates system**, known also as *normalized device coordinates* (*NDC*), is not Cartesian in the space if the projection to use is not parallel (i.e. it is perspective or nonlinear).

  The standard cube $[-1, 1]^3$ is the area, whose contents may be visible in the picture. Anything outside this area will be clipped off.

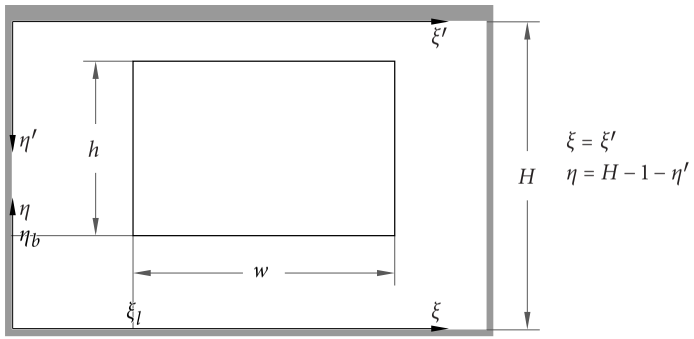
  The transformation from the viewer system to the standard cube system is a projective transformation, represented by a 4×4 matrix.

- **The OpenGL viewport system** is defined in a rectangle on the screen. The axis units of this system are equal to the width and height of one pixel.

- **The coordinate systems of OpenGL viewport** i **and the windowing system** are bound to the window used to display the graphics.

  Usually the two systems have vertical axes oriented in the opposite directions and the origins in different corners of the viewport.

The viewport in the window



To define the viewport of a given size and position in the window, one has to execute the command

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

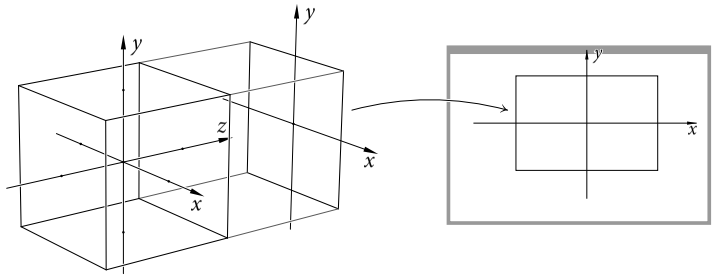
or

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

If the viewport has to fill the entire window, it reduces to the following:

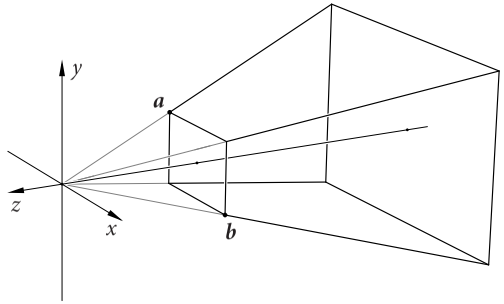
```
glViewport ( 0, 0, W, H );
```

The mapping of the standard cube to the viewport



This transformation is done between the clipping and rasterisation stages of the rendering pipeline. The lines having directions of the z-axis are mapped to points. Of two points located on such a line, the one "closer to the viewer" is the one having the z coordinate smaller.

The implementation of perspective projections



The **viewing frustum** is a truncated pyramid, given by six parameters—l, r, b, t, n, f. The points $\boldsymbol{a} = (l, t, -n)$ and $\boldsymbol{b} = (r, b, -n)$ are its vertices, its back face is in the plane $z = -f$ (in the viewer coordinate system).

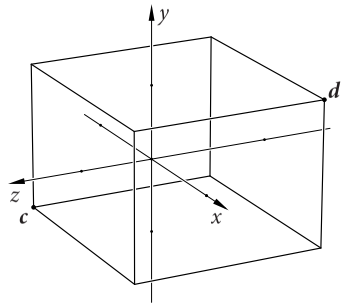
The letters to denote these parameters are taken from the words `left`, `right`, `bottom`, `top`, `near`, `far`.

The matrix P to represent the perspective transformation (a perspective collineation) transforming the viewing frustum to the standard cube and its inverse are given by the formulae

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

We multiply the vector of homogeneous coordinates of a point to transform by the matrix P and we obtain homogeneous coordinates of this point in the standard cube system. Usually the fourth (weight) coordinate is different from 1.

For a **parallel projection** the view volume (in the viewer system) is a rectangular parallelpiped.



It is defined by the points $\boldsymbol{c} = (l, b, -n)$ i $\boldsymbol{d} = (r, t, -f)$.

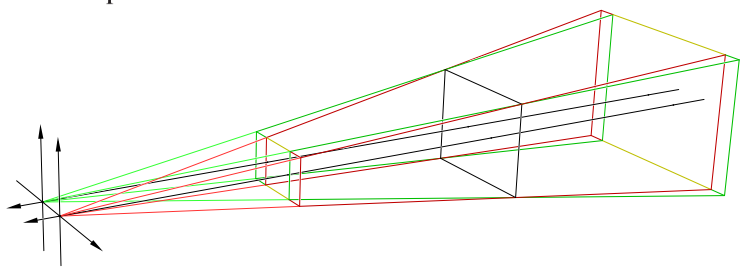
The matrix P used to transform the view volume to the standard cube and its inverse are given by

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

Every raster device has a specific so-called **aspect**—the ratio a of the width and height of a pixel (in physical units). The viewport of $w \times h$ pixels has physical dimensions having the ratio $aw : h$. To avoid the distortion of objects on the picture, one has to ensure that the parameters of the view volume satisfy the proportion $(r - l) : (t - b) = aw : h$.

Most modern computer screens have the aspect $a = 1$ or a close enough to 1 to be replaced by 1 without a noticeable error.

Stereoscopic vision



Two perspective projections for stereoscopy require taking into account four physical dimensions: the **distance between the viewer eyes**, **the distance of the viewer from the screen** and the **width** and **height of the viewport**.

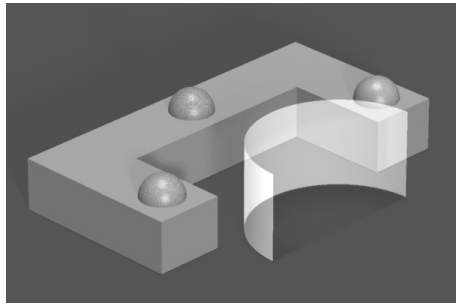
Unlike with the "usual" perspective projection, the unit length of the viewer coordinate system must be related with the physical dimensions, in millimeters, or, the other way, the above-mentioned four physical dimensions must be specified in the viewer-system units to obtain the proper parameters l, r, b, t, n, f for each viewing frustum. Both of them will have the same parameters b, t, n and f.

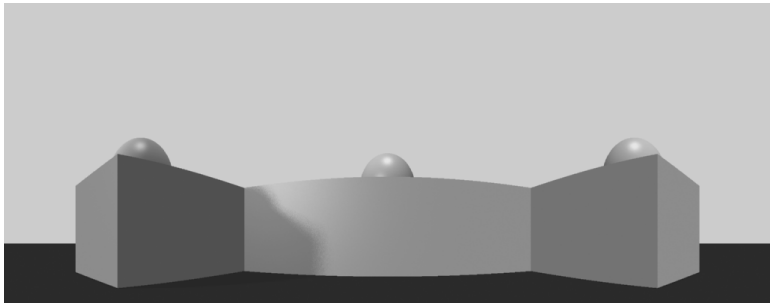
If the transformation from the world system to the viewer system is an isometry, the unit lengths of both systems are the same.

Nonlinear projections

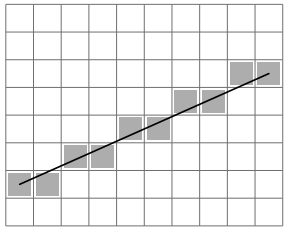
Transformations of the three-dimensional space to a plane other than parallel and perspective projections do not preserve collinearity nor coplanarity of points. Images of straight line segments may be curved, which complicates the methods of drawing. A line segment must be divided to pieces short enough, similarly a triangle must be divided to a number of triangles short enough to approximate the curvature of the image.

The most common nonlinear projection is the **panorama**, i.e., a central projection on a cylinder, developed to the image plane. The centre of projection is located on the cylinder axis.





Rasterisation algorithms



For a **line segment** the purpose is to find the pixels making its image—being *closest* to it. The classical **Bresenham algorithm** is developed as follows.

We assume that the end points have integer coordinates (i.e., they are located at pixel centres), (x_0, y_0) and (x_1, y_1). For a while we assume the inequalities $x_1 > x_0$, $y_1 \geq y_0$.

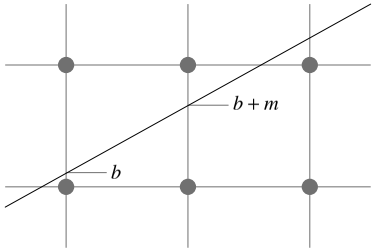
In each column of pixels from x_0 to x_1 we choose one pixel, whose colour is to be set by some procedure `SetPixel`.

First approach:

```
Δx = x1 − x0;   Δy = y1 − y0;   m = Δy/Δx;
for ( x = x0, y = y0;   x <= x1;   x++, y += m )
  SetPixel ( x, round(y) );
```

The variables `m` and `y` take fractional values, hence, floating-point artihmetic must be used, rather costly and not free of rounding errors. The value of `y` must be rounded, which takes additional time.

In every new column we stay in the same row or we move up by one pixel, depending on which of two pixels is closer to the line. The idea is to move up, whenever it gives us a **smaller error**.



The dots here mark pixel centres. The letter b is the signed distance of the intersection of our line with the vertical raster line. In the next column it will be $b + m$, and if its greater than 1/2, then we move up.

Second approach:

```
Δx = x1 − x0;   Δy = y1 − y0;   m = Δy/Δx;
for ( x = x0, y = y0, b = 0;   x <= x1;   x++ ) {
  SetPixel ( x, y );
  b += m;
  if ( b > 0.5 ) { y++; b -= 1.0; }
}
```

Now `y` takes only integer values, but the variables `m` and `b` may have fractional values.

All fractions assigned to these variables are ratios of integers with the denominator $2\Delta x$. The calculations may thus be done with numerators only.

Bresenham algorithm:

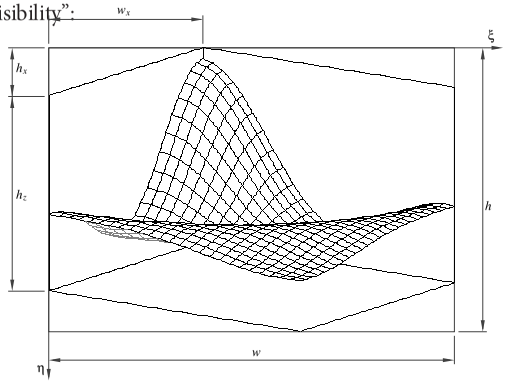
```
Δx = x1 − x0;   Δy = y1 − y0;
for ( x = x0, y = y0, c = −Δx;   x <= x1;   x++ ) {
  SetPixel ( x, y );
  c += 2Δy;
  if ( c > 0 ) { y++; c -= 2Δx; }
}
```

All operations are done on integers. If the assumption $x_1 > x_0$, $y_1 \geq y_0$ is not satisfied, then the roles of the coordinates x, y may be swapped, and their increments have to be taken with appropriate signs.

```
dx = x1 - x0;  g = dx > 0 ? +1 : -1;  dx = abs(dx);
dy = y1 - y0;  h = dy > 0 ? +1 : -1;  dy = abs(dy);
if ( dx > dy ) {
  for ( x = x0, y = y0, c = -dx;  x != x1;  x += g ) {
    SetPixel ( x, y );
    c += 2*dy;
    if ( c > 0 ) { y += h; c -= 2*dx; }
  }
else {
  for ( x = x0, y = y0, c = -dy;  y != y1;  y += h ) {
    SetPixel ( x, y );
    c += 2*dx;
    if ( c > 0 ) { x += g; c -= 2*dy; }
  }
}
```


Floating horizon algorithm

Here is a graph of a function of two variables, drawn with an axonometric projection and "visibility":



We are going to draw a net of lines on the graph surface, $z = f(x, y)$, for a continuous function $f\colon [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \to [z_{\min}, z_{\max}] \subset \mathbb{R}$.

To construct the projection, one should specify the dimensions w, h of the surrounding rectangle and the dimensions w_x, h_x i h_z. The images of points $(x_{\min}, y_{\min}, z_{\min})$, $(x_{\min}, y_{\min}, z_{\max})$, $(x_{\max}, y_{\min}, z_{\min})$, $(x_{\max}, y_{\max}, z_{\min})$ are (w_x, h_z), $(w_x, 0)$, $(0, h_x + h_z)$ i $(w - w_x, h,)$. Using the above one can easily find the coefficients $a_\xi, \ldots, d_\xi$ and $a_\eta, \ldots, d_\eta$ for the formulae

$$\xi = a_\xi x + b_\xi y + c_\xi z + d_\xi,$$
$$\eta = a_\eta x + b_\eta y + c_\eta z + d_\eta.$$

The line segments are drawn "from the front to the rear". During the drawing process the lines drawn earlier determine a "forbidden area", which is represented by two arrays of integers `hb` and `ht`, called **horizons**. Each element of the array corresponds to one pixel column.

According to the orientation of the vertical axis η, the initial value of the lower horizon (`hb`) is -1, and the upper horizon (`ht`) is initialised to $h + 1$; the values of the lower horizon will increase and the top horizon will decrease. In a column `x` the forbidden area is between the rows `ht[x]` and `hb[x]`; if `ht[x] > hb[x]`), then the forbidden area in the column `x` is empty.

If the pixel `(x,y)` of the line being drawn satisfies the condition `ht[x] <= y <= hb[x]`, then it is located in the forbidden area and it must not be drawn. Otherwise it is either above the upper horizon (`y < ht[x]`) or below the lower horizon (`y > hb[x]`). It has to be assigned a colour, which may depend on the inequality satisfied. Then the y coordinate of the pixel must be assigned to the appropriate horizon (one of the two or both, if it is the first pixel drawn in the column `x`).

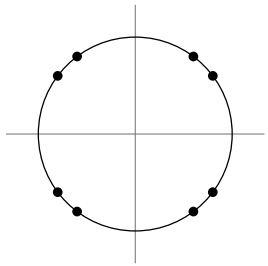
One problem is to avoid the situation, where the pixel obscures other pixels of the same line segment, in the same column, but not drawn yet.

This may be dealt with by drawing the line segments twice. The first time, the pixels are drawn (i.e. their colour is assigned), but the horizons are left unchanged. In the second pass the horizons are updated without drawing.

A fully correct effect is obtained by drawing pairs of line segments having a common end point—first we draw them, then we update the horizons to the pair.

Drawing a circle

The calculation necessary to draw a circle, whose radius r and the coordinates of the centre are integers, may also be done using integer numbers only. If the centre is at the point $(0, 0)$, then one pixel, (x, y), determines 7 other pixels, by swapping the coordinates and giving them all possible signs.



Consider two equalities:

$$\sum_{i=0}^{x}(2i+1) = (x+1)^2,$$
$$\sum_{i=y}^{r}(2i-1) = r^2 - (y-1)^2.$$

The value of the function

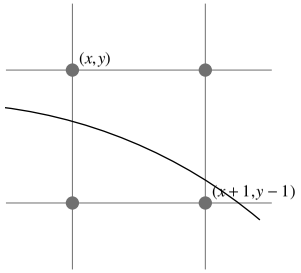
$$f(x,y) = \sum_{i=0}^{x}(2i+1) - \sum_{i=y}^{r}(2i-1) \;=: (x+1)^2 + (y-1)^2 - r^2$$

is 0, if the point $(x+1, y-1)$ is located on the circle, is positive if this point is outside and negative if it is inside the circle.

We shall find the pixels (x, y) such that $0 \le x \le y \le r$, and we shall obtain the other pixels from the symmetry. Beginning with the pixel $(0, r)$, in every step we increase x, and, when necessary, we decrease y.

The value of f will measure the error; after setting a pixel value, we decrease y if

$$|f(x, y)| < |f(x, y+1)|.$$



Let $c = f(x, y)$. Then,
$f(x, y+1) = c + 2y - 1$, $f(x+1, y) = c + 2x + 3$, $f(x+1, y-1) = f(x+1, y) - 2y + 3$.

We always choose between a pixel inside and outside the circle, i.e., we have $f(x, y) \le 0 \le f(x, y+1)$. Thus we avoid calculating absolute values; the condition equivalent to $|f(x, y)| < |f(x, y+1)|$ is $-c < c + 2y - 1$, i.e. $2c > 1 - 2y$. Hence, the algorithm:

```
for ( x = 0, y = r, c = 2*(1-r);  x <= y;  x++, c += 2*x+1 )
{
  Set8Pixels ( x, y );
  if ( 2*c > 1-2*y ) {
    y --;
    c += 1-2*y;
  }
}
```


Obviously, we shall see a circle on the screen if the screen's aspect is equal or close enough to 1. Otherwise it is necessary to draw an ellipse, whose main axes are parallel to the screen edges and their lengths are appropriately chosen.

One can draw an ellipse whose half-axes—horizontal and vertical—have lengths a and b respectively. A possible algorythm draws a circle of radius $r = ab$ on a virtual raster. Each step the x-coordinate is increased by b and the y-coordinate is decreased by a. This means that the variable `c` is modified by adding or subtracting a number of terms, not just one. On the physical raster the coordinate increments are +1 and –1.

So, we take $r = ab$ and we define the function

$$f(x, y) = (x+b)^2 + (y-a)^2 - r^2 = \sum_{i=0}^{x+b-1} (2i+1) - \sum_{i=y-a+1}^{r} (2i-1).$$

Hence,

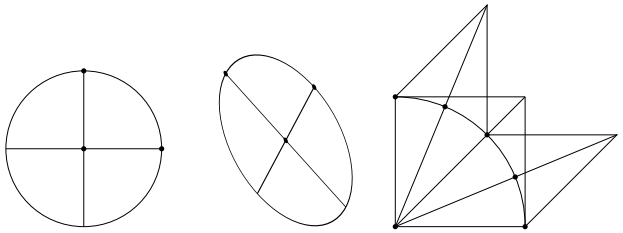
$$f(x, y+a) = f(x, y) + (2y - a)a,$$
$$f(x+b, y) = f(x, y) + (2x + 3b)b,$$
$$f(x+b, y-a) = f(x+b, y) + (3a - 2y)a.$$

Drawing the pixel (x, y), we already know $c = f(x, y)$. With the step b to the left, we decide, whether to go also down by a pixels. To decide, we compare $|f(x, y)|$ with $|f(x, y+b)|$, or, equivalently, we chech whether $-c < c + (2y - a)a$, i.e., $2c + (2y - a)a > 0$—if this inequality is satisfied, then we go down, upating the value of `c`.

The raster image of the ellipse has only two axes of symmetry, and we can draw only one eighth of it. Another one eighth must be drawn separately, in a similar way.

Another method of drawing curves (including ellipses) is replacing them by polylines made of line segments short enough, and drawing the semgents. The density of points of the curve (vertices of the polyline) may be chosen adaptively, to achieve a good effect with a possibly small cost.

A general ellipse may be represented by its centre and two vectors, which correspond to **conjugate halfaxes**. An ellipse is the image of a unit circle under an affine mapping, and the conjugate axes are images of two mutually orthogonal diameters.



Let $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ denote vectors, which determine end points of an arc of the unit circle taking the angle α. The midpoint of the arc is determined by the vector $\boldsymbol{v} = (\boldsymbol{v}_0 + \boldsymbol{v}_1)/c$, where $c = 2\cos\alpha/2$. We can take $c_i = \cos \pi/2^{i+1}$ and compute (in preprocessing) a number of initial elements of the sequence $1/c_0, \ldots, 1/c_{n-1}$. Then it is possible to divide recursively the circle arc to halves, quarters etc.

Instead of the mutually orthogonal unit vectors, at the beginning we can take any two vectors—corresponding to the conjugate halfaxes of the ellipse. Then we shall get points of the ellipse.

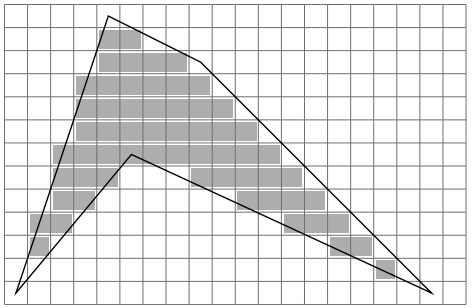
The recursion may be stopped afret reaching the depth limit or after getting a line segment short enough.

The recursive procedure might look like

```
void r_Ellipse ( k, s, v0, v1 )
{
  if ( k == n || close_enough ( v0, v1 ) )
    draw_line ( s+v0, s+v1 );
  else {
    v2 = a[k]*(v0+v1);
    r_Ellipse ( k+1, s, v0, v2 );
    r_Ellipse ( k+1, s, v2, v1 );
  }
} /*r_Ellipse*/
```

The first parameter of the first call (from the application) must be `0`.

Filling polygons



A well designed procedure of rasterisation ought to ensure a unique classification of pixels to polygons having a common edge. Of course, a pixel whose centre point is inside the polygon belongs to the polygon image. What about the pixels on the boundary?

For example:

If the points to the right of the pixel centre are *inside* the polygon, then the pixel belongs to the polygon image.

If the pixel centre is the left end point of a horizonlat edge, then it belongs to the polygon image if points below this edge are inside the polygon or there is a left-side vertical edge below the pixel centre.

Due to this (or similar) convention every pixel of a plane covered by polygons having common edges (and disjoint insides) each pixel belongs to the image of one (and only one) polygon.

The OpenGL standard does not enforce a convention for implementations.

Parity rule: a point not located on the boundary of a (bounded) polygon is inside it, if any halfline beginning at this point crosses the boundary at an odd number of places.

A sequential algorithm of filling a polygon with horizontal line segments uses a table of so-called **active edges**. A polygon edge is active if it is not horizontal and it is crossed by the horizontal raster line at the current level y. For each non-vertical edge we reject its lower end point.

Due to the parity rule the number of active edges is always even.

The algorithm: (assuming that the vertices have integer coordinates)

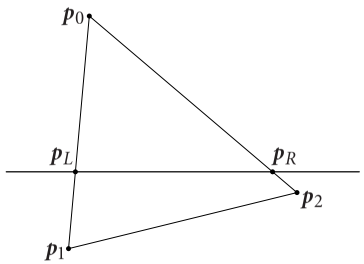
- Based on the array of vertices find the polygon's edges (pairs of consecutive vertices, including the "closing edge", $\overline{(x_{n-1}, y_{n-1})(x_0, y_0)}$) and store them in an array. Skip all horizontal edges.
- For any edge $\overline{(x_i, y_i)(x_{i+1}, y_{i+1})}$ such that $y_{i+1} < y_i$ swap the end points.
- Sort the edges so as to obtain a sequence with increasing y coordinates of the first end point.
- Create an initially empty active edge array.

- In a loop, for the variable y changing from the minimal to maximal y-coordinate of raster lines crossing the polygon, do the following.
  - Remove from the active edge array all edges, whose second end point is at the level y.
  - Add to the active edge array edges whose first end point is at the level y.
  - Compute the x coordinate of intersection of each edge with the current horizontal line.
  - Sort the active edge array increasingly with respect to the x coordinates found above.
  - Fill with a colour horizontal lines of the current line between consecutive pairs of active edges.

The rasterisation algorithm of OpenGL-a admits triangles only, but it is possible to draw triangles with non-integer coordinates of the vertices (single-precision floating point representation is used). The algorythm works in homogeneous coordinates, computing also depths (i.e. z-coordinates) of the pixels, necessary for visibility tests. If vertices have additional attributes, they are appropriately interpolated.

The computation done by GPU is highly parallel.

Triangle's vertices are represented by vectors of homogeneous coordinates, $\boldsymbol{P}_i = (X_i, Y_i, Z_i, W_i)$. All weight coordinates must be positive. First the vertices with the greatest and smallest values of the coordinate y are found.



For a raster line at the level y the algorithm finds the points $\boldsymbol{p}_L$ and $\boldsymbol{p}_R$ of intersection of the edges with the horizontal line. In what follows I assume that it looks as on the picture.

A parametric form of the line segment $\overline{\boldsymbol{p}_0\boldsymbol{p}_1}$ in homogeneous representation is

$$\{\, \boldsymbol{P} : \boldsymbol{P} = (1-t)P_0 + t\boldsymbol{P}_1, t \in [0,1] \,\}.$$

We are looking for a point $\boldsymbol{P}_L = (X, Y, Z, W)$ such that $Y/W = y$. From

$$\frac{(1-t)Y_0 + tY_1}{(1-t)W_0 + W_1} = y$$

it follows

$$t = \frac{-Y_0 - W_0 y}{(Y_1 - Y_0) - (W_1 - W_0)y} \quad \text{and} \quad x_L = \frac{X_L}{W_L} = \frac{(1-t)X_0 + tX_1}{(1-t)W_0 + tW_1}.$$

Similarly the point $\boldsymbol{P}_R$ is found together with the coordinate x_R. It is done in parallel for all horizontal raster lines crossing the triangle.

Given y, with the points $\boldsymbol{P}_L$ and $\boldsymbol{P}_R$, for each pixel (x, y) of the line segment $\overline{\boldsymbol{P}_L\boldsymbol{P}_R}$ in parallel, the point $\boldsymbol{P}$ of the triangle mapped onto this pixel is found. It is

$$\boldsymbol{P} = (1-s)\boldsymbol{P}_L + s\boldsymbol{P}_R, \qquad s = \frac{-X_L - W_L x}{(X_R - X_L) - (W_R - W_L)x}$$

In particular the z-coordinate, i.e., the depth of the point is computed.

All extra attributes are interpolated just as the homogeneous coordinates. If the attribute is declared with the qualifier `noperspective`, the parameters t and s are computed based on the Cartesian coordinates of the triangle vertices, which results in a different final effect.

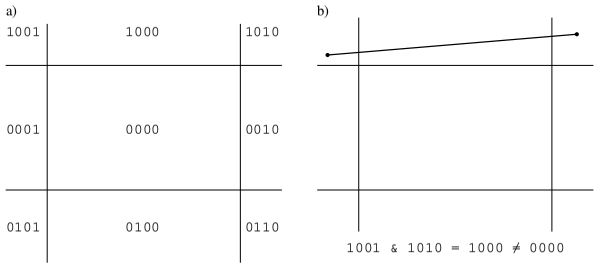
Line clipping

We begin with finding the intersection of a line segment, given by its end points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ in a plane, with a rectangle (e.g. a viewport on the screen).

Historically the earliest is the **Sutherland–Cohen algorithm**, dividing the plane with the straight lines of the rectangle's edges into 9 regions.

Each straight line is associated with one bit; each point of the plane has a code made of these bits; hence, there are four-bit codes. The bit is zero, if the point is located on the "right" side of the edge line, and one, if it is on the "wrong" side.

The first step is to find the codes of the points $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$. If the bitwise conjunction of the two codes is nonzero, both end points of the line segment are located on the wrong side of at least one rectangle's edge, and thus the entire line segment is outside the rectangle.

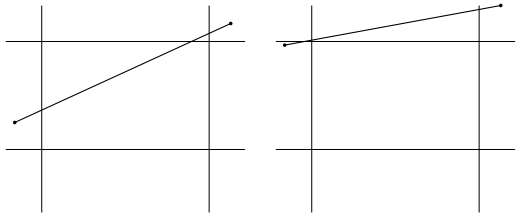


If both codes are zero, the entire line segment is in the rectangle.

If one of the codes is nonzero and their bitwise conjunction ("and") is zero, a part of the line may be inside the rectangle. If the code of $\boldsymbol{p}_0$ is zero—we swap the points and their codes.

Then, based on the code, we choose the edge line crossed by our line segment and we find the intersection point. We replace $\boldsymbol{p}_0$ by this point and we find its code. The bit corresponding to the edge line must be zero (we may enforce this to compensate for rounding errors).

Then we repeat the computation—until we find the intersection of the line segment with the rectangle or we reject all its pieces.



The algorithm is costly, as full codes are being found, and some bits may later be ignored. Also, there is an error cumulation—the intersection points other than the first are computed using intermediate results (not the original data).

Liang–Barsky algorithm

Here the original points $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ are kept until the final result is computed. At each stage only the necessary computations are done. A part of a line is represented by two numbers, t_0 and t_1, which determine the interval of a part of the line in parametric form: $\{\, \boldsymbol{p} = (1-t)\boldsymbol{p}_0 + t\boldsymbol{p}_1 : t \in [t_0, t_1] \,\}$.

Initially $t_0 = 0$, $t_1 = 1$, later t_0 may only be increased and t_1 decreased. Note that we can take arbitrary initial values for t_0 and t_1 if another segment of the line passing through $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ is to be clipped. We can even take $t_0 = -\infty$ or $t_1 = \infty$, which may be used to represent the entire line.


```
char LBTest ( float p, float q, float *t0, float *t1 )
{
  float r;

  if ( p < 0.0 ) {
    if ( (r = q/p) > *t1 ) return false;
    else if ( r > *t0 ) *t0 = r;
  }
  else if ( p > 0.0 ) {
    if ( (r = q/p) < *t0 ) return false;
    else if ( r < *t1 ) *t1 = r;
  }
  else return q < 0.0;
  return true;
} /*LBTest*/
```



```
char LBClip ( point *p0, point *p1 )
{
  float t0, t1, dx, dy;

  t0 = 0.0;  t1 = 1.0;  dx = p1->x - p0->x;
  if ( LBTest ( -dx, p0->x - xmin ) )
    if ( LBTest ( dx, xmax - p0->x ) ) {
      dy = p1->y - p0->y;
      if ( LBTest ( -dy, p0->y-ymin ) )
        if ( LBTest (dy, ymax-p0->y ) ) {
          if ( t1 != 1.0 )
            { p1->x = p0->x + t1*dx; p1->y = p0->y +t1*dy; }
          if ( t0 != 0.0 )
            { p0->x += t0*dx;  p0->y += t0*dy; }
          return true;
        }
    }
  return false;
} /*LBClip*/
```

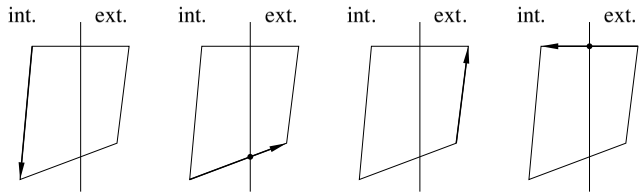

A generalisation of the algorithms discussed above may be used for clipping lines in spaces of arbitrary dimensions, to any convex polyhedra. Such a polyhedron is the intersection of a finite number of halfspaces. Each end point of the intersection of the line segment with the polyhedron is either an original end point or the common point of the line segment with the hyperplane bounding one of the halfspaces.

The algorithms may easily be modified to work using homogeneous coordinates—but in that case weight coordinates of the points must be positive, as this is necessary to determine, whether the point is on the “right” side of the hyperplane.

Clipping polygons

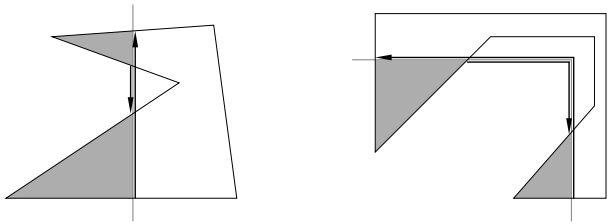
The **Sutherland–Hodgman algorithm** may be used to find the intersection of a polygon with a halfspace. To clip a polygon to a convex polyhedron (or polygon), one can use the algorithm as many times as the number of the clipping area faces (unless the polygon is entirely rejected at an early stage).

We distinguish two sides of a hyperplane, naming them "internal" ("right") and "external" ("wrong").




```
void SHClip ( int n, point w[] )
{
  s = w[n-1];  is = Inside ( s );
  for ( i = 0; i < n; i++ ) {
    p = w[i];  ip = Inside ( p );
    if ( is ) {
      if ( ip ) Output ( p );
      else Output ( q = Intersect ( s, p ) );
    }
    else if ( ip ) {
      Output ( q = Intersect ( s, p ) );
      Output ( p );
    }
    s = p;  is = ip;
  }
} /*SHClip*/
```


The auxiliary procedures `Inside`, `Intersect` and `Output`, respectively, determine the side of the clipping hyperplane for a point, find the intersection of a line segment (edge of polygon) with the hyperplane and output a vertex of the result, the clipped polygon.



If the polygon is not convex, then its intersection with a halfspace may be not connected. In that case we obtain "false edges" of the polygon. Obviously, this problem does not exist with triangles.

The **Weiler–Atherton algorithm** may be used to finding intersections, unions, differences and symmetric differences of arbitrary polygons, which need not be connected and may have holes. As opposed to the previous algorithm, the problem here is planar, because in this case orientation of the polygon boundary is essential. We can define orientation in a plane, which may be immersed in a higher-dimensional space.

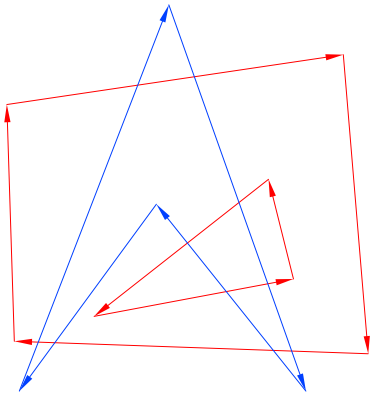
We assume that the boundary of each polygon consists of one or more closed polylines. Each edge is oriented, i.e., it has the first and second end point. Going along the edges in accordance with their orientation, we shall have the polygon's interior on the right-hand side.

Each vertex is the first and the second point of exactly one edge. If necessary, we can introduce more vertices at the same location.

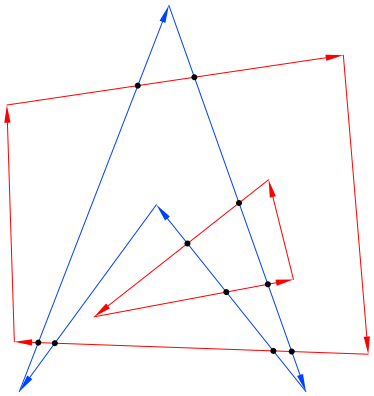
Such a representation makes it easy to find the complement of the polygon—one can just reverse the orientation of all edges. Thus, having a procedure of intersecting polygons, we can easily find the union (complement of intersection of complements), difference (intersection of the first polygon with the complement of the second one) or symmetric difference of the polygons.

The algorithm constructs a directed graph, whose vertices are vertices of the polygons and whose edges are polygons edges or their pieces. This graph is processed to output polylines bounding the intersection.

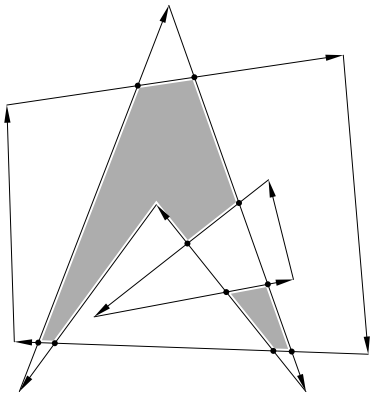
Initially the graph is the union of graphs constructed separately for the polygons:



New vertices are added at the intersection points of the edges; the edges are divided:



The graph is searched using depth-first search method:



We mark all vertices as not-visited yet. Beginning with any not-visited vertex, we test, whether it is a vertex of the intersection. Each vertex at the intersection of polygon's edges is a vertex of the intersection, for other vertices we have to check, whether they are located in the interior of the other polygon (using the parity rule).

Beginning with such a vertex we use the DFS method to search the graph; we output the edges as we pass them, until we go back to the beginning point. If this point is a vertex at the intersection of polygon edges, then it is the beginning of two graph edges; we choose the one entering the interior of the other polygon.

Crossing an intersection vertex we *always* choose the edge of the other polygon. The vertices are marked as visited.

The algorithm terminates when all vertices of the intersection have been visited.

The computational cost for polygons having m and n edges respectively may be of order mn, which is optimal if each edge of one polygon intersects all edges of the other polygon. Often the intersection has less edges. Using techniques of Computational Geometry (like sweeping or plane division trees) it is possible to reduce the cost.

The second most costly part of the algorithm is testing, whether a vertex is located inside the other polygon. The cost of graph searching is linear with respect to the number of its vertices.

Most problems are caused by rounding errors, when a vertex is located on an edge of the other polygon or it is close to the edge. Also, cases when edges of the polygons have common parts, are troublesome, but it is necessary to write code dealing with them (it may even be greater than the basic part of the algorithm implementation).

In OpenGL the clipping is done as the last stage of the front part of the rendering pipeline. All primitives (points, line segments and triangles) passed to this stage, are clipped to the standard cube. Apart from the six planes of the cube facets, the application may specify a number of additional clipping planes. It is done as follows:

The equation of a plane containing the point $\boldsymbol{p}_0 = (x_0, y_0, z_0)$, whose normal vector is $\boldsymbol{n} = (a, b, c)$ is the folowing:

$$ax + by + cz + d = 0,$$

where $d = -ax_0 - by_0 - cz_0 = \langle \boldsymbol{n}, \boldsymbol{o} - \boldsymbol{p}_0 \rangle$. In homogeneous coordinates it is

$$aX + bY + cZ + dW = 0. \qquad (*)$$

Let $W > 0$. Depending on the sign of the left-hand side expression, the point $\boldsymbol{p} = (X/W, Y/W, Z/W)$ is on the clipping plane (if 0), on the "right" side (if the expression value is positive) or on the "wrong" side (if negative).

For each vertex (of a line segment or a triangle) one has to output the value of the expression (*) or any other—it is interpreted as the signed distance from the clipping plane. If these distances for consecutive vertices have different signs, the clippng stage of OpenGL will find the intersection point and then it will output the appropriate piece of the line segment or triangle.

Even an octagon may result from clipping a triangle to the cube and any extra clipping plane may increase the number of final vertices by 1. A polygon obtained by clipping a triangle is divied to triangles which are output to the rasterisation stage.

Clipping involves interpolation of all attributes of the vertices—also in this case signed distances are used in the computations.

The last shader of the front part of the rendering pipeline must have the declaration

```
out float gl_ClipDistance[1];
```

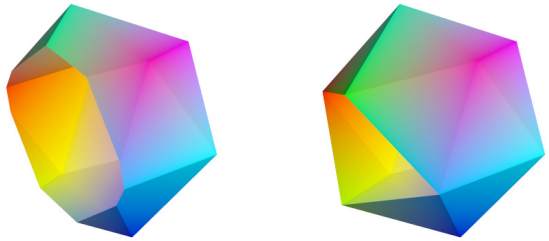
(the length of the array may of course be different); when a vertex is output, its distances from the extra clipping planes must be stored in this array.

The application, before drawing must enable the extra clipping planes by calling the procedure

```
glEnable ( GL_CLIP_DISTANCE0 + i );
```

(it may then be disabled by the procedure `glDisable`).

There is also a mechanism for **culling** primitives — a line segment or a triangle is rejected if *all* its vertices are on the wrong side of the clipping plane. The array to be declared is called **gl_CullDistance** and the results of clipping and culling triangular facets of the icosahedron may look like this:



It is possible to use the clipping mechanism of OpenGL to clip objects to areas bounded by curved surfaces. The numbers given in the array `gl_ClipDistance` may just be distances of a vertex from the surface—but the boundary of the rejected part of a triangle will always be a straight line segment. Therefore big triangles must be divided into small enough parts, by a geometry shader or a tesselation shader.

Example: The icosahedron inscribed in the unit ball has been clipped to the ball of radius 0.85; for each vertex output to the clipping stage the following instruction has been executed:

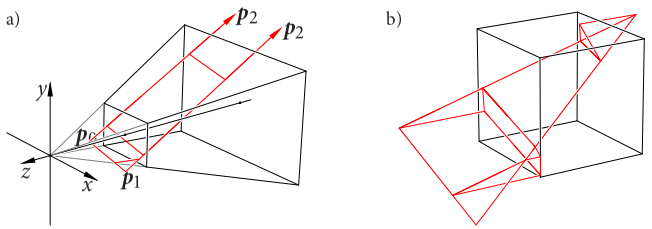
```
gl_ClipDistance[0] = 0.85 - length ( pos[k] );
```

where `pos` is the array with positions of vertices of 36 subtriangles obtained from division of a triangular facet of the icosahedron.

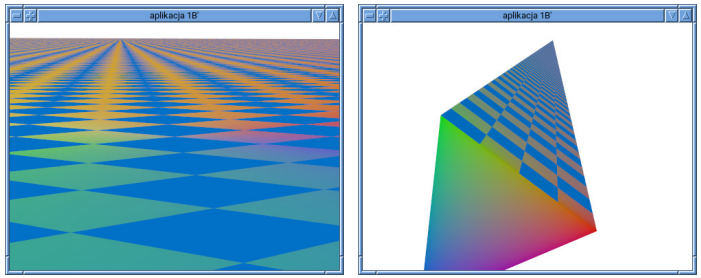
By changing the sign of numbers output in the `gl_ClipDistance` array we can obtain the result of clipping the icosahedron facets to the complement of the ball of radius 0.85.

Some applications require implementing the clipping algorithm in a shader, as it must be done before the perspective transformation (i.e., in the viewer coordinate system, not in the system of the standard cube). We may want to draw images of unbounded figures, like entire planes visible "to the horizon" at infinity. To do that, we need to clip our primitives to the side facets of the view frustum, then, what passed this clipping, must be divided into parts protruding in front of the near clipping plane, staying inside the view volume and sticking behind the far clipping plane.

The parts of primitives in front of the near plane and behind the far plane, after the perspective transformation, must be projected on the front and back facets of the standard cube to be drawn; note that the visibility algorithm for these parts does not work, and it is necessary to display them in a proper order to get the correct image.

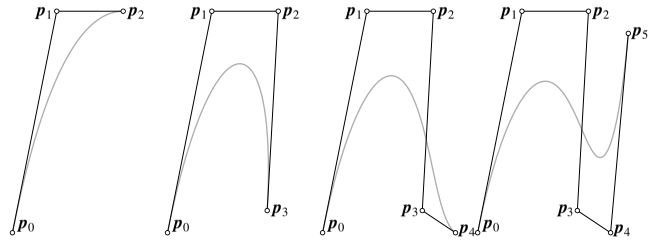


Results may look as follows:



Elements of geometric design

A **Bézier curve** of degree n is a representation of a parametric polynomial curve taking the form of a sequence of $n + 1$ **control points**. The so-called **control polygon**, whose vertices are these points, approximates the curve.



Parametrisation of the Bézier curve is given by the formula

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i B_i^n(t),$$

with **Bernstein basis polynomials**

$$B_i^n(t) \stackrel{\text{def}}{=} \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n.$$

As we can see, it is a parametric parameterisation of a polynomial curve of degree at most n.

Its domain may be an arbitrary interval, but usually it is $t \in [0, 1]$.

To study properties of this representation and develop algorithms, we prove the recursive formula

$$B_0^0(t) = 1,$$
$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \quad \text{for } n > 0,$$

where by convention $B_i^n(t) \equiv 0$ for $i < 0$ and $i > n$.

We check directly that $B_0^0(t) = \binom{0}{0} t^0 (1-t)^0$.
For $n > 0$ and $i \in \{1, \ldots, n-1\}$ we write

$$(1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) =$$
$$(1-t)\binom{n-1}{i} t^i (1-t)^{n-i-1} + t\binom{n-1}{i-1} t^{i-1}(1-t)^{n-i} =$$
$$\left(\binom{n-1}{i} + \binom{n-1}{i-1}\right) t^i (1-t)^{n-i} = \binom{n}{i} t^i (1-t)^{n-i} = B_i^n(t).$$

If $i = 0$ or $i = n$, then this calculation may be done without the term with the zero factor $B_{-1}^{n-1}(t)$ or $B_n^{n-1}(t)$. There is $\binom{n-1}{0} = \binom{n-1}{n-1} = \binom{n}{0} = \binom{n}{n} = 1$. □

With the formula just proved, for $n > 0$ we obtain

$$\begin{aligned}
\boldsymbol{p}(t) &= \sum_{i=0}^{n} \boldsymbol{p}_i B_i^n(t) = \boldsymbol{p}_0 B_0^n(t) + \sum_{i=1}^{n-1} \boldsymbol{p}_i B_i^n(t) + \boldsymbol{p}_n B_n^n(t) \\
&= (1-t)\boldsymbol{p}_0 B_0^{n-1}(t) + \sum_{i=1}^{n-1} \boldsymbol{p}_i \big((1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)\big) + t\boldsymbol{p}_n B_{n-1}^{n-1}(t) \\
&= (1-t)\sum_{i=0}^{n-1} \boldsymbol{p}_i B_i^{n-1}(t) + t\sum_{i=0}^{n-1} \boldsymbol{p}_{i+1} B_i^{n-1}(t).
\end{aligned}$$

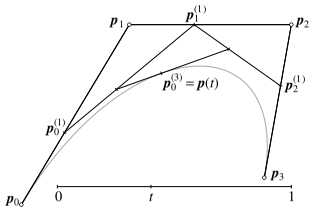
The point $\boldsymbol{p}(t)$ may therefore be obtained by interpolation between points of two Bézier curves of degree $n-1$; the first is represented by the points $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_{n-1}$ and the second by $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$.

Going further, we can get to the points on the curves of degree 0: for all i there is $\sum_{i=0}^{0} \boldsymbol{p}_i B_0^0(t) = \boldsymbol{p}_i$.

The above is the basis for the **de Casteljau algorithm**: we label the points $\boldsymbol{p}_i^{(0)} = \boldsymbol{p}_i$ and for the given number t we compute

```
for ( j = 1; j <= n; j++ )
  for ( i = 0; i <= n-j; i++ )
    p_i^(j) = (1-t)p_i^(j-1) + tp_(i+1)^(j-1);
```

The last point obtained in this way is $\boldsymbol{p}_0^{(n)} = \boldsymbol{p}(t)$.



Properties of Bézier curves:

- For all $n \in \mathbb{N}$ and $t \in \mathbb{R}$ there is $\sum_{i=0}^{n} B_i^n(t) = 1$; hence, the curve and its control points are located in the same space. Moreover, if f is any affine transformation, the image of the curve $\boldsymbol{p}$, represented by the points $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n$, is represented by the points $f(\boldsymbol{p}_0), \ldots, f(\boldsymbol{p}_n)$. The Bézier representation of the curve is therefore an **affine invariant**.

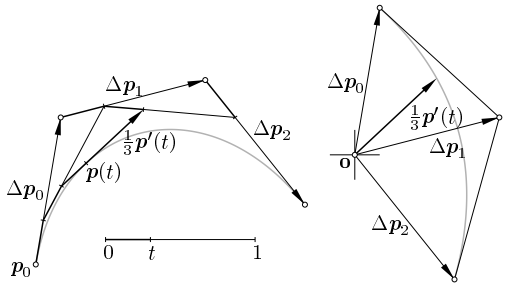
- If $t \in [0,1]$, then for $i \in \{0, \ldots, n\}$ there is $B_i^n(t) \geq 0$. Together with the previous property it means that all points $\boldsymbol{p}(t)$ of the curve, for $t \in [0,1]$, are located in the **convex hull** of the set of control points.

- There is $B_0^n(0) = 1$, $B_1^n(0) = \cdots = B_n^n(0) = 0$ and $B_0^n(1) = \cdots = B_{n-1}^n(1) = 0$, $B_n^n(1) = 1$. Hence, it follows the **interpolation of the first and last control point**: $\boldsymbol{p}(0) = \boldsymbol{p}_0$, $\boldsymbol{p}(1) = \boldsymbol{p}_n$.

- One can easily prove that $\frac{\mathrm{d}}{\mathrm{d}t} B_i^n(t) = n\big(B_{i-1}^{n-1}(t) - B_i^{n-1}(t)\big)$ (this is left as an exercise, note that: $B_{-1}^{n-1}(t) = B_n^{n-1}(t) = 0$). Based on that, we can obtain

$$\begin{aligned}\boldsymbol{p}'(t) &= \sum_{i=0}^{n} \boldsymbol{p}_i \big(B_i^n(t)\big)' \\ &= n\Big(-\boldsymbol{p}_0 B_0^{n-1}(t) + \sum_{i=1}^{n-1} \boldsymbol{p}_i \big(B_{i-1}^{n-1}(t) - B_i^{n-1}(t)\big) + \boldsymbol{p}_n B_{n-1}^{n-1}(t)\Big) \\ &= \sum_{i=0}^{n-1} n(\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) B_i^{n-1}(t).\end{aligned}$$

The derivative of a Bézier curve of deree n is a curve of degree $n-1$ whose control points are the vectors $n(\boldsymbol{p}_1 - \boldsymbol{p}_0), \ldots, n(\boldsymbol{p}_n - \boldsymbol{p}_{n-1})$.



- We can also see that

$$\boldsymbol{p}'(t) = n\Big(\sum_{i=0}^{n} \boldsymbol{p}_{i+1} B_i^{n-1}(t) - \sum_{i=0}^{n-1} \boldsymbol{p}_i B_i^{n-1}(t)\Big) = n\big(\boldsymbol{p}_1^{(n-1)} - \boldsymbol{p}_0^{(n-1)}\big).$$

The (vector) derivative for a given t may be obtained by multiplying by n the difference of the points obtained in the one-but-last step of the de Casteljau algorithm.

- The de Casteljau algorithm makes it possible to **divide** Bézier curves.
If $0 < t < 1$, then the points obtained by this algorithm, $\boldsymbol{p}_0^{(0)}, \ldots, \boldsymbol{p}_0^{(n)}$ and $\boldsymbol{p}_0^{(n)}, \ldots, \boldsymbol{p}_n^{(0)}$, represent curve arcs corresponding to the intervals $[0,t]$ and $[t,1]$. We can introduce local parameters, e.g., s and u, changing from 0 to 1 in these intervals respectively.

The division of the curve my be done using the procedure

```
void Divide ( int n, point p[], point q[], float t )
{
  q[0] = p[0];
  for ( j = 1; j <= n; j++ ) {
    for ( i = 0; i <= n-j; i++ )
      p[i] = (1-t)*p[i] + t*p[i+1];
    q[j] = p[0];
  }
} /*Divide*/
```

The array `q` is filled with representation of the first arc of the curve and the original data in the array `p` are replaced by the control points of the second arc.

- The procedure above may be used for **recursive subdivision** of the curve, e.g., in order to draw it as an approximating polyline made of line segments short enough—just like ellipses.

  The subdivision stop criterion may be as follows: if the distance of the control points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1}$ from the line segment $\overline{\boldsymbol{p}_0\boldsymbol{p}_n}$ is less than some ε, then from the convex hull property all points of the arc are not more distant from this segment than ε. In practice we can assume ε close to the diameter of a single pixel.

- No straight line on a plane, nor a plane (in a 3D space) has more intersections with the curve than with the control polygon. This is called the **variation-diminishing property**.

- Bernstein basis polynomials satisfy also the formula

$$B_i^n(t) = \frac{n-i}{n+1-i} B_i^{n+1}(t) + \frac{i-1}{n+1} B_{i+1}^{n+1}(t).$$

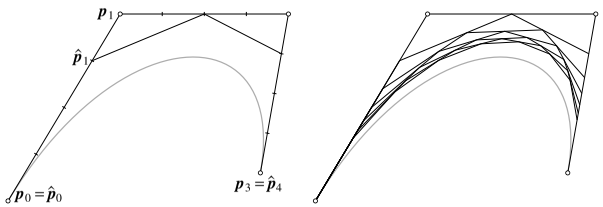
  By substituting the right-hand side to the curve definition, after some calculation, we obtain

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i B_i^n(t) = \sum_{i=0}^{n+1} \hat{\boldsymbol{p}}_i B_i^{n+1}(t).$$

  where

$$\hat{\boldsymbol{p}}_i = \frac{n+1-i}{n+1} \boldsymbol{p}_i + \frac{i}{n+1} \boldsymbol{p}_{i-1}.$$

  In this way we can do a **degree elevation**, i.e., construction of a representation of the curve of a higher degree.



- The formula for degree elevation implies that if the control points $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n$ are collinear, ordered along the line and equidistant, then the Bézier curve is a straight line segment parameterised with a constant velocity.

- The computational cost of the de Casteljau algorithm is of order n^2, but apart from the point $\boldsymbol{p}(t)$ the algorithm produces many other useful results (e.g., derivatives, arc division). A linear algorithm, whose result is just the point $\boldsymbol{p}(t)$, is based on the Horner scheme. Let $s = 1 - t$. Then

$$\boldsymbol{p}(t) = \boldsymbol{p}_0 \binom{n}{0} s^n + \boldsymbol{p}_1 \binom{n}{1} t s^{n-1} + \cdots + \boldsymbol{p}_{n-1} \binom{n}{n-1} t^{n-1} s + \boldsymbol{p}_n \binom{n}{n} t^n =$$
$$\left(\cdots\left(\boldsymbol{p}_0 \binom{n}{0} s + \boldsymbol{p}_1 \binom{n}{1} t\right) s + \cdots + \boldsymbol{p}_{n-1} \binom{n}{n-1} t^{n-1}\right) s + \boldsymbol{p}_n \binom{n}{n} t^n.$$

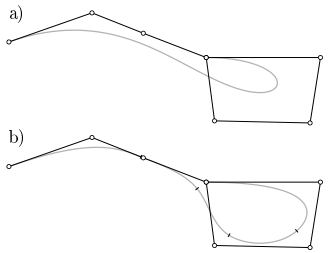
  We can use the formulae $\binom{n}{0} = 1$, $\binom{n}{1} = n$ and $\binom{n}{i+1} = \frac{n-i}{i+1}\binom{n}{i}$ to obtain the algorithm

```
s = 1-t;  p = p_0;  d = t;  b = n;
for ( i = 1; i <= n; i++ ) {
  p = s*p + b*d*p_i;
  d *= t;  b = (b*(n-1))/(i+1);
}
```

B-spline curves

Bézier curves may be joined, which enables designing complicated shapes: from interpolation of the first and last control point it follows a simple method of joining curves in a continuous way, and the fact that the curve is tangent to the first and last line segment of the control polyline implies a method of joining the curves smoothly. It is possible to extend the construction so as to obtain continuity of higher order derivatives, but it is not that convenient.

A more systematic method of obtaining smooth piecewise polynomial curves uses the **B-spline representation**. The curve consists of a number of polynomial arcs of degree at most n. The number n needs not be big, but the number of arcs may be as big as necessary.



To obtain a complicated shape, one needs many control points. For a Bézier curve it enforces a high degree and then the shape of the curve may have little to do with the shape of the control polygon. This is different for B-spline curves.

To define a B-spline curve one has to specify the **degree** n, a non-decreasing sequence of **knots** $u_0, \ldots, u_N$ and a sequence of **control points** $\boldsymbol{d}_0, \ldots, \boldsymbol{d}_{N-n-1}$. The parametrisation is given by the formula

$$\boldsymbol{s}(t) = \sum_{i=0}^{N-n-1} \boldsymbol{d}_i N_i^n(t), \qquad t \in [u_n, u_{N-n}),$$

with **normalised B-spline functions of degree** n, determined by the number n and the knot sequence, using the **Mansfield–de Boor–Cox formula**:

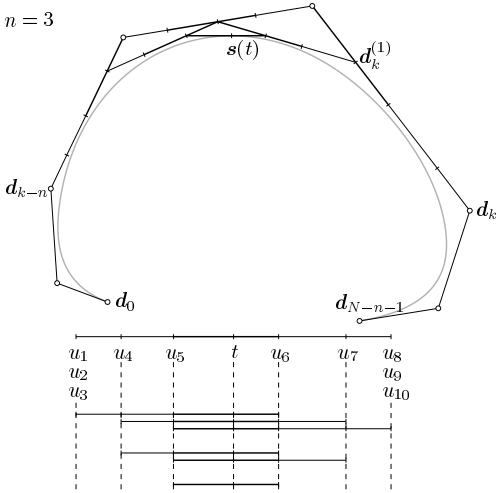
$$N_i^0(t) = \begin{cases} 1 & \text{for } t \in [u_i, u_{i+1}), \\ 0 & \text{else,} \end{cases}$$

$$N_i^n(t) = \frac{t - u_i}{u_{i+n} - u_i} N_i^{n-1}(t) + \frac{u_{i+n+1} - t}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(t) \qquad \text{for } n > 0.$$

It is a generalisation of the recursive formula for the Bernstein basis polynomials, proved earlier.

Just as the Bézier curves, B-spline curve parametrisations may be evaluated based on the Mansfield–de Boor–Cox formula, which makes it possible to derive the **de Boor algorithm**. For a given curve $\boldsymbol{s}$ and the number $t \in [u_n, u_{N-n})$ the algorithm computes the point $\boldsymbol{s}(t)$:

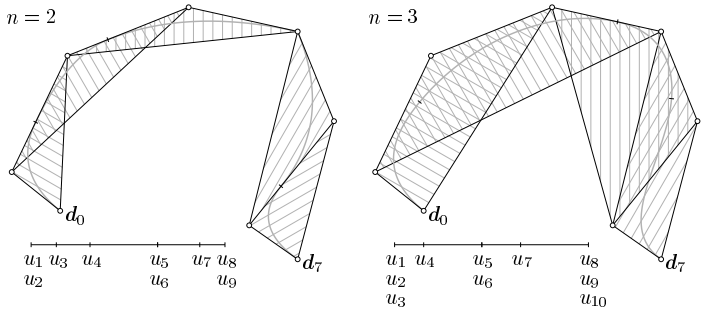
```
/* find k, such that t ∈ [u_k, u_{k+1}) */
/* denote d_i^(0) = d_i for i = k − n, ..., k */
for ( r = 0; r < N && t == u_{k−r}; r++ );
for ( j = 1; j < n-r; j++ )
  for ( i = k-n+j; i <= k-r; i++ ) {
    α_i^(j) = (t − u_i)/(u_{i+n+1−j} − u_i);
    d_i^(j) = (1 − α_i^(j)) d_{i−1}^(j−1) + α_i^(j) d_i^(j−1);
  }
/* s(t) = d_{k−r}^(n−r) */
```



Properties of B-spline curves:

- If all knots from u_n to u_{N-n} are unique (form an increasing sequence), then the curve consists of $N - 2n$ polynomial arcs of degree at most n; otherwise (if there are knots of multiplicity greater than 1) the number of arcs is smaller.

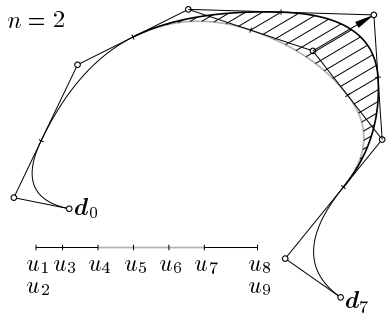
- The de Boor algorithm performs the interpolation of points obtained in consecutive steps, and there is always $\alpha_i^{(j)} \in [0,1]$; hence, all points obtained by this algorithm are located in the convex hull of the points $\boldsymbol{d}_{k-n}, \ldots, \boldsymbol{d}_k$. The entire arc corresponding to $t \in [u_k, u_{k+1})$ is contained in the convex hull of these control points—this is called the **strong convex hull property**.



- The sum of B-spline functions in the interval $[u_k, u_{k+1})$ is constant, namely 1. Hence, the **affine invariance** if the representation follows: any afine transformation applied to the control points produces a representation of the image of the curve under the same transformation.

- The representation enables a **local shape control**: as the point $\boldsymbol{s}(t)$ for $t \in [u_k, u_{k+1})$ depends only on the control points $\boldsymbol{d}_{k-n}, \ldots, \boldsymbol{d}_k$. Moving any other control point does not modify the polynomial arc for $t \in [u_k, u_{k+1})$.

  As a consequence, moving the control point $\boldsymbol{d}_i$ changes only the arc of the curve corresponding to $t \in [u_n, u_{N-n}) \cap [u_i, u_{i+n+1})$.

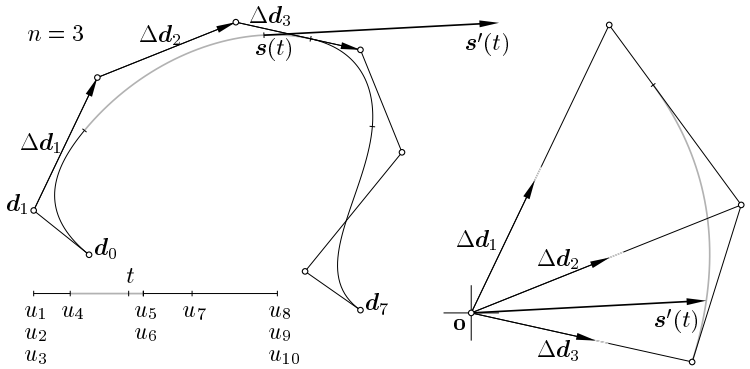


- As the curve consists of polynomial arcs of degree at most n, the derivative (**hodograph**) of the parametrisation is described by polynomial arcs of degree at most $n-1$.
  It may be described as follows:

$$\boldsymbol{s}'(t) = \sum_{i=0}^{N-n-2} \frac{n}{u_{i+n+1} - u_{i+1}} (\boldsymbol{d}_{i+1} - \boldsymbol{d}_i) N_{i+1}^{n-1}(t).$$

  The knot sequence used to define the functions N_i^{n-1} is the same as the one used in the definition of $N_i^n(t)$.

  Using the strong convex hull property to the derivative yields the **strong hodograph property** of B-spline curces: for $t \in (u_k, u_{k+1})$ the vector $\boldsymbol{s}'(t)$ is a linear combination of the differences $\boldsymbol{d}_{k-n+1} - \boldsymbol{d}_{k-n}, \ldots, \boldsymbol{d}_k - \boldsymbol{d}_{k-1}$, with positive coefficients.



- In a neighbourhood of a knot of multiplicity r the parametrisation has $n - r$ continuous derivatives. If $r = n$, then the parametrisation is continuous, but its derivative needs not be continuous. At a knot of multiplicity $r = n - 1$ the derivative is continuous. The multiplicity $r \leq n - 2$ guarantees also continuity of the second order derivative. In particular, a cubic ($n = 3$) curve with knots of multiplicity 1 has the parametrisation of class C^2. If its first-order derivative is nonzero, then the curvature of the curve is continuous.

- If two consecutive knots have multiplicities at least n, i.e., $u_{k-n+1} = \cdots = u_k < u_{k+1} = \cdots = u_{k+n}$, then the control polygon contains a Bézier representation of the polynomial arc corresponding to $t \in [u_k, u_{k+1})$. We have

$$\boldsymbol{s}(t) = \sum_{i=0}^{n} \boldsymbol{d}_{k-n+i} B_i^n(v), \quad \text{where} \quad v = \frac{t - u_k}{u_{k+1} - u_k}.$$

- If a knot has multiplicity n, then the corresponding control point is also a point of the curve. More specifically, if $u_{k-n+1} = \cdots = u_k < u_{k+1}$, then $\boldsymbol{s}(u_k) = \boldsymbol{d}_{k-n}$.

Knot insertion (W. Boehm, 1980) is a method of constructing a representation of *the same* curve (with the same parametrisation) using a new basis—inserting a knot causes extending the spline function space by one. We assume that the new knot is the number $t \in [u_n, u_{N-n})$.

1. We define so-called **Greville abscissae**:

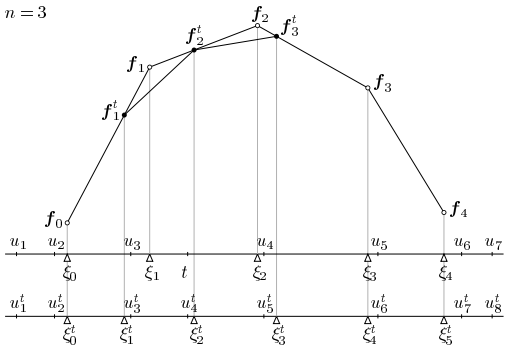
   $$\xi_i = \frac{1}{n}(u_{i+1} + \cdots + u_{i+n}).$$

   We are going to process a polyline whose vertices are $\boldsymbol{f}_i = (\xi_i, \boldsymbol{d}_i)$, $i = 0, \ldots, N-n-1$, considering it as a graph of piecewise linear (vector) function.

2. The number $t \in [u_n, u_{N-n})$ is inserted to the original knot sequence, so as to preserve its non-decreasing ordering.

3. We compute the Greville abscissae ξ_i^t for the new knot sequence.

4. On the polyline we find the points $\boldsymbol{f}_i^t = (\xi_i^t, \boldsymbol{d}_i^t)$. The points $\boldsymbol{d}_i^t$ are control points of the new representation.



Implementions need not compute the Greville abscissae.
Below the new representation replaces the old one in the arrays:

```
/* u[i] = u_i for i = 1,...,N-1, d[i] = d_i for i = 0,...,N-n-1, */
/* t ∈ [u_n, u_{N-n}] */
k = N-1;
while ( t < u[k] ) k --;
r = 0;  i = k;
while ( i ≥ 1 && t = u[i] ) { i --; r ++; }
for ( i = N-n-1; i ≥ k-r; i -- ) d[i+1] = d[i];
for ( i = k-r; i ≥ k-n+1; i -- )
  d[i] = ((u[i+n]-t)*d[i-1] + (t-u[i])*d[i])/(u[i+n]-u[i]);
for ( i = N-1; i ≥ k+1; i -- ) u[i+1] = u[i];
u[k+1] = t;
N ++;
/* the variable N and the arrays u and d contain the
result. */
```

Properties of knot insertion:

- The numbers of knots and control points are increased by 1.
- The result of inserting a number of knots does not depend on the ordering of the new knots.
- The parametrisation is unchanged (only the representation changes).
- The de Boor algorithm of computing the point $\boldsymbol{s}(t)$ is equivalent to inserting a new knot, i.e., the number t, as many times as necessary to obtain a representation with the knot t of multiplicity n.

- By inserting an infinite sequence of knots dense in the interval $[u_n, u_{N-n})$, we obtain a sequence of control polygons converging to the curve. The distance may be estimated by a constant times $h_{\max}^2$, where $h_{\max}$ is the maximal distance between consecutive knots.

Applications:

- A representation with a small number of knots is convenient to establish a general shape of the curve. Then we can refine the representation, i.e., insert knots and obtain the representation suitable for designing fine details.
- After inserting enough knots, we obtain a polyline being an approximation of the curve good enough for drawing.
- We can insert knots so as to raise their multiplicity to $n + 1$. Then the control polygon consists of control polygons of Bézier curves making the spline curve. Then we can use a procedure capable of drawing Bézier curves.

- In various constructions algebraic operations are done on spline functions (e.g. degree elevation, addition, multiplication). One can insert knots, then do the operations on polynomial arcs in Bézier representation and then remove unnecessary knots from the result (by solving some systems of linear equations).

The **Lane–Riesenfeld algorithm** is a method of inserting many knots at the same time, but the B-spline curve must be defined with a sequence of equidistant knots. The representation obtained with this algorithm has a sequence of knots twice denser—the new knots divide to halves the intervals between consecutive old knots. By repeating this algorithm we can obtain a sequence of polylines converging fast to the curve that we might like to draw.

With no loss of generality we can assume that the original knots are consecutive integers. The new knots will have the fractional part $\frac{1}{2}$. We can also write the formula

$$\boldsymbol{s}(t) = \sum_{i\in\mathbb{Z}} \boldsymbol{c}_i N_i^n(t) = \sum_{i\in\mathbb{Z}} \boldsymbol{d}_i M_i^n(t),$$

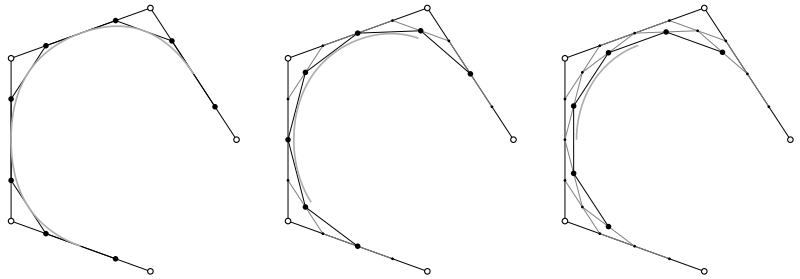
with the B-spline functions N_i^n based on the original knots and the functions M_i^n based on the new sequence. The infinitely many terms of both sums will be rejected later, leaving only the non-vanishing in the interval $[n, N-n)$.

The algorithm consists of the step of **doubling**, followed by n steps of **averaging**.

The doubling is done by taking for each point $\boldsymbol{c}_i$ two points at the same location, $\boldsymbol{d}_{2i}^{(0)} = \boldsymbol{d}_{2i+1}^{(0)} = \boldsymbol{c}_i$.

Averaging: in the j-th step, for all $i \in \mathbb{Z}$ we compute $\boldsymbol{d}_i^{(j)} = \frac{1}{2}(\boldsymbol{d}_{i-1}^{(j-1)} + \boldsymbol{d}_i^{(j-1)})$.

The result consists of the points $\boldsymbol{d}_i = \boldsymbol{d}_i^{(n)}$.



Tensor product Bézier and B-spline patches

A tensor product patch is a parametrisation using a so-called **tensor product** basis—having two bases of functions of one variable, $\{f_0, \ldots, f_n\}$ and $\{g_0, \ldots, g_m\}$, we obtain a tensor product basis made of all products $f_i \otimes g_j$, i.e., bivariate functions

$$(f_i \otimes g_j)(u, v) = f_i(u) g_j(v).$$

Their domain is the Cartesian product of domains of the functions f_i and g_j.

If the sum of elements of the tensor product basis is 1, then the vector coefficients of the tensor product patche are **control points** of the patch; they reside in the same space and they form a **control net** of the patch.

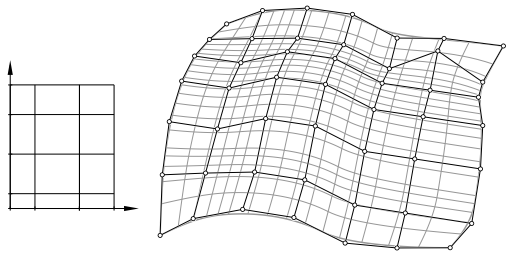
In particular we obtain **Bézier patches** of degree (n, m):

$$\boldsymbol{p}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} \boldsymbol{p}_{ij} B_i^n(u) B_j^m(v),\ \ (u,v) \in [0,1]^2$$

and **B-spline patches** of degree (n, m):

$$\boldsymbol{s}(u,v) = \sum_{i=0}^{N-n-1}\sum_{j=0}^{M-m-1} \boldsymbol{d}_{ij} N_i^n(u) N_j^m(v),\ \ (u,v) \in [u_n, n_{N-n}) \times [v_m, v_{M-m}).$$

The basis functions are defined by two sequences of knots, $u_0, \ldots, u_N$ and $v_0, \ldots, v_M$, different in general.



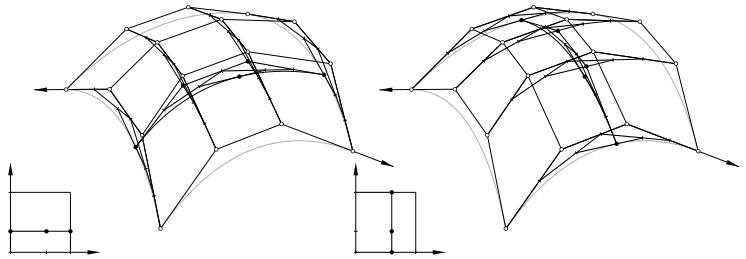
If the curve is a stretched and bent line segment, then the tensor product patch is a stretched and bent rectangle.

We can apply all algorithms of processing curves for tensor product patches. For example, to compute the point $\boldsymbol{p}(u, v)$ of a Bézier patch, given the numbers u, v, we can find points of a number of Bézier curves:

$$\boldsymbol{p}(u,v) = \sum_{i=0}^{n} \underbrace{\Big(\sum_{j=0}^{m} \boldsymbol{p}_{ij} B_j^m(v)\Big)}_{\boldsymbol{q}_i} B_i^n(u) = \sum_{i=0}^{n} \boldsymbol{q}_i B_i^n(u).$$

The points $\boldsymbol{q}_i$ may be found using the de Casteljau algorithm as well as the Horner scheme. The same point may also be found using the formula

$$\boldsymbol{p}(u,v) = \sum_{j=0}^{m} \underbrace{\Big(\sum_{i=0}^{n} \boldsymbol{p}_{ij} B_i^n(u)\Big)}_{\boldsymbol{r}_j} B_j^m(v) = \sum_{j=0}^{m} \boldsymbol{r}_j B_j^m(v).$$



The first-order partial derivatives of a Bézier patch are

$$\frac{\partial \boldsymbol{p}}{\partial u}(u,v) = \sum_{i=0}^{n-1}\sum_{j=0}^{m} n(\boldsymbol{p}_{i+1,j} - \boldsymbol{p}_{ij}) B_i^{n-1}(u) B_j^m(v),$$

$$\frac{\partial \boldsymbol{p}}{\partial v}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m-1} m(\boldsymbol{p}_{i,j+1} - \boldsymbol{p}_{ij}) B_i^{n}(u) B_j^{m-1}(v),$$

hence, we can obtain them if we can find the derivative of a Bézier curve $\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i B_i^n(t)$:

$$\boldsymbol{p}'(t) = n\left(\sum_{i=0}^{n-1} \boldsymbol{p}_{i+1} B_i^{n-1}(t) - \sum_{i=0}^{n-1} \boldsymbol{p}_i B_i^{n-1}(t)\right) = n\left(\boldsymbol{p}_1^{(n-1)} - \boldsymbol{p}_0^{(n-1)}\right).$$

The points $\boldsymbol{p}_0^{(n-1)}$ i $\boldsymbol{p}_1^{(n-1)}$ are obtained in the one-but last step of the de Casteljau algorithm.

We can use the de Casteljau algorithm to **divide** a tensor product patch along a curve of constant parameter u or v, we can also perform a **degree elevation**.

Similarly, all algorithms for processing B-spline curves may be applied to tensor product B-spline patches, e.g., **computing a point**, **computing derivatives** or **knot insertion**.

Also, properties of curves transfer to tensor product patches, e.g., **affine invariance of the representation**, (**strong**) **convex hull property** and (**strong**) **hodograph property**.

Rational Bézier and B-spline patches

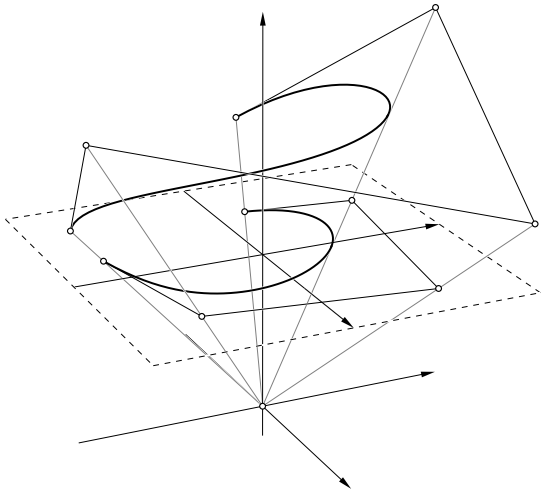
Rational curves and patches may be obtained by representing their control points using homogeneous coordinates. (Piecewise) polynomial parametric curves and patches in the space of homogeneous coordinates represent curves and patches whose coordinates are described with (piecewise) rational functions. Thus, a point

$$\boldsymbol{P}(t) = \sum_{i=0}^{n} \boldsymbol{P}_i B_i^n(t)$$

of a Bézier curve represented by the points $\boldsymbol{P}_i = (X_i, Y_i, Z_i, W_i)$ consists of homogeneous coordinates of the point

$$\boldsymbol{p}(t) = \frac{\sum_{i=0}^{n} w_i \boldsymbol{p}_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)},$$

where $w_i = W_i$ and $\boldsymbol{p}_i = (x_i, y_i, z_i) = (X_i/W_i, Y_i/W_i, Z_i/W_i)$ for $i = 0, \ldots, n$ (if all weights are nonzero).



To design a rational curve or a rational patch, we place its control points $\boldsymbol{p}_i$ (or $\boldsymbol{d}_{ij}$) in the space, i.e., we choose their Cartesian coordinates, and we manipulate with the weight coordinates.

All algorithms discussed so far may be applied to homogeneous curves or patches, and then we can transfer to the Cartesian coordinates in order to draw the object. Even this is not needed in OpenGL, as the front-end shaders are supposed to output vectors of homogeneous coordinates of vertices of primitives to the clipping stage, i.e., we need to output points of the homogeneous curves or patches.

If all weights are the same, e.g., 1, then the rational curves and patches are identical to the polynomial curves or patches, as in this case the denominator in the formula describing the parametrisation is a constant.

The only conic curves having polynomial parametrisations are parabolas.

The class of rational curves contains in particular parametrisations of all conic curves (circles, ellipses, hyperbolas). The class of rational patches contains in particular all quadrics (surfaces of degree 2) and surfaces of revolution, whose generatrices are polynomial or rational curves.

Another advantage is **projective invariance** of the representation. Note that a perspective projection of a parabola may be any conic curve. The image of a rational curve under a projective transformation (or a perspective projection) may be any rational curve (of the same degree).

For a rational Bézier curve we can write

$$\boldsymbol{p}(t) = \sum_{i=0}^{n} \boldsymbol{p}_i \frac{w_i B_i^n(t)}{\sum_{j=0}^{n} w_j B_j^n(t)}.$$

The sum of rational functions multiplied by the control points $\boldsymbol{p}_i$ is 1, hence, such a representation is an affine invariant. If all weights are positive, then the rational functions are nonnegative for $t \in [0,1]$ and we can prove the **convex hull property** of rational Bézier curves.

Rational spline curves and patches are denoted by an acronym NURBS, from *nonuniform rational B-splines*, which emphasises using sequences of non-equidistant knots.

A normal vector of a rational patch may be obtained as follows: let $\boldsymbol{P} = \boldsymbol{P}(u,v)$, $\boldsymbol{P}_u = \boldsymbol{P}_u(u,v)$ and $\boldsymbol{P}_v = \boldsymbol{P}_v(u,v)$ be a point of the homogeneous patch and its derivatives at (u,v). Then, we can compute the **vector product** of three vectors in $\mathbb{R}^4$:

$$\boldsymbol{N} = \boldsymbol{P} \wedge \boldsymbol{P}_v \wedge \boldsymbol{P}_u.$$

The vector $\boldsymbol{N}$ represents the tangent plane of the rational patch at the point $\boldsymbol{p}(u,v)$ and its first three coordinates describe the normal vector of this plane—after dividing it by its length we obtain a unit normal vector.

Here is the formula: if $\boldsymbol{a} = (x_{\boldsymbol{a}}, y_{\boldsymbol{a}}, z_{\boldsymbol{a}}, w_{\boldsymbol{a}})$, $\boldsymbol{b} = (x_{\boldsymbol{b}}, y_{\boldsymbol{b}}, z_{\boldsymbol{b}}, w_{\boldsymbol{b}})$, $\boldsymbol{c} = (x_{\boldsymbol{c}}, y_{\boldsymbol{c}}, z_{\boldsymbol{c}}, w_{\boldsymbol{c}})$, then

$$\boldsymbol{a} \wedge \boldsymbol{b} \wedge \boldsymbol{c} = \begin{bmatrix} -d_{34} y_{\boldsymbol{c}} + d_{24} z_{\boldsymbol{c}} - d_{23} w_{\boldsymbol{c}} \\ d_{34} x_{\boldsymbol{c}} - d_{14} z_{\boldsymbol{c}} + d_{13} w_{\boldsymbol{c}} \\ -d_{24} x_{\boldsymbol{c}} + d_{14} y_{\boldsymbol{c}} - d_{12} w_{\boldsymbol{c}} \\ d_{23} x_{\boldsymbol{c}} - d_{13} y_{\boldsymbol{c}} + d_{12} z_{\boldsymbol{c}} \end{bmatrix},$$

where

$$d_{12} = x_{\boldsymbol{a}} y_{\boldsymbol{b}} - y_{\boldsymbol{a}} x_{\boldsymbol{b}}, \quad d_{13} = x_{\boldsymbol{a}} z_{\boldsymbol{b}} - z_{\boldsymbol{a}} x_{\boldsymbol{b}}, \quad d_{14} = x_{\boldsymbol{a}} w_{\boldsymbol{b}} - w_{\boldsymbol{a}} x_{\boldsymbol{b}},$$
$$d_{23} = y_{\boldsymbol{a}} z_{\boldsymbol{b}} - z_{\boldsymbol{a}} y_{\boldsymbol{b}}, \quad d_{24} = y_{\boldsymbol{a}} w_{\boldsymbol{b}} - w_{\boldsymbol{a}} y_{\boldsymbol{b}}, \quad d_{34} = z_{\boldsymbol{a}} w_{\boldsymbol{b}} - w_{\boldsymbol{a}} z_{\boldsymbol{b}}.$$

Mesh methods

A control net of a B-spline patch approximates the patch itself and by repeating knot insertion we can obtain a sequence of control nets convergent to the patch. An element of this sequence, located sufficiently close to the surface, may be drawn instead of the surface, after dividing the facets to triangles.

The actual algorithm of knot insertion is irrelevant; moreover, there exist other methods of mesh refinement yielding sequences of meshes convergent to (various) surfaces. One can define a surface by choosing the refinement algorithm.

Suppose that we have a B-spline patch of degree (n, n), defined with two sequences of **equidistant** knots. Each of the two sequences may be refined, by inserting new knots in the middle of each interval between consecutive knots, using the Boehm algorithm or the Lane–Riesenfeld algorithm. Depending on the sequence to be refined, we process either rows or columns of the control net.

The Lane–Riesenfeld algorithm may be modified so as to process rows and columns at the same time, as both operations on rows, i.e. doubling and averaging, are commutative with the operations on columns. The algorithm for curves consists of a doubling step followed by n averaging steps, we can begin with doubling od rows and columns and then we can average n times rows and columns at the same time.

So, we have a B-spline patch

$$\boldsymbol{s}(u, v) = \sum_i \sum_k \boldsymbol{c}_{ik} N_i^n(u) N_k^n(v) = \sum_i \sum_k \boldsymbol{d}_{ik} M_i^n(u) M_k^n(v),$$

represented with given control points $\boldsymbol{c}_{ik}$, which represent the patch in the tensor product basis $\{ N_i^n \otimes N_k^n : i, k \in \mathbb{Z} \}$, defined with a sequence of equidistant knots. Our goal is to find the control points $\boldsymbol{d}_{ij}$, which represent the same patch in the tensor product basis made of the functions M_i^n defined with a knot sequence twice denser.

In the **doubling** step we take

$$\boldsymbol{d}_{2i,2k}^{(0)} = \boldsymbol{d}_{2i,2k+1}^{(0)} = \boldsymbol{d}_{2i+1,2k}^{(0)} = \boldsymbol{d}_{2i+1,2k+1}^{(0)} = \boldsymbol{c}_{ik}.$$

In the j-th step of **averaging**, for $j = 1, \ldots, n$, we compute

$$\boldsymbol{d}_{ik}^{(j)} = \frac{1}{4}\left(\boldsymbol{d}_{i-1,k-1}^{(j-1)} + \boldsymbol{d}_{i,k-1}^{(j-1)} + \boldsymbol{d}_{i-1,k}^{(j-1)} + \boldsymbol{d}_{i,k}^{(j-1)}\right).$$

At the end we obtain the points $\boldsymbol{d}_{ik} = \boldsymbol{d}_{ik}^{(n)}$.

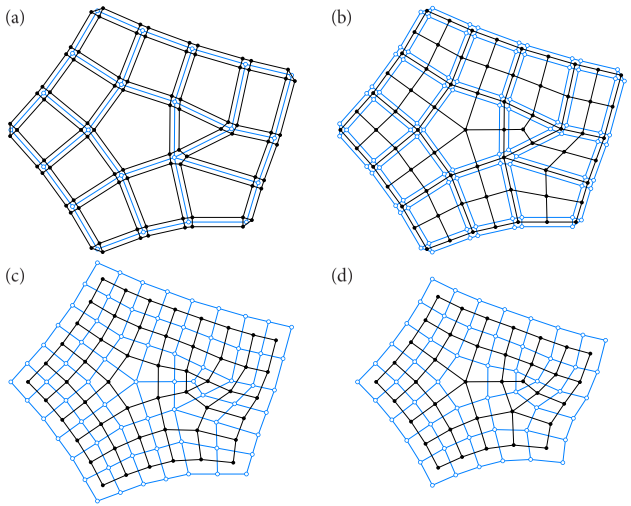
It is easy to implement this algorithm using a rectangular array. In the net we can distinguish **vertices**, **edges** (pairs of vertices $\left(\boldsymbol{d}^{(j)}_{i-1,k}, \boldsymbol{d}^{(j)}_{i,k}\right)$) and **facets** (quadruples $\left(\boldsymbol{d}^{(j)}_{i-1,k-1}, \boldsymbol{d}^{(j)}_{i,k-1}, \boldsymbol{d}^{(j)}_{i-1,k}, \boldsymbol{d}^{(j)}_{i,k}\right)$).

Doubling is replaced by replacing each column by its two copies and each row by its two copies. Then, for each edge and each vertex a new quadrangular facet appears (degenerated to a line or to a point).

The averaging is computing vertices at the centre of gravity of each facet. The edges of the resulting mesh corespond to common edges of facets in the given mesh. Facets of the new mesh correspond to the vertices of the given mesh incident with 4 edges. By interpreting the mesh as a graph, we notice that averaging is a construction of the dual graph (except for the vertices at the mesh boundary).

The refinement algorithm may be generalised to **irregular meshes**. Such a mesh has **vertices**, **edges** and **facets**, where an edge is a pair of vertices and a facet is a closed polyline made of edges. Then, each facet has to have distinct vertices nd each edge belongs either to one or to two facets. Each internal vertex (whose all incident edges belong to two facets) is incident with at least 3 different edges.

Such a mesh is an approximation of a surface, which may be obtained at the limit of the sequence of meshes obtained by repeated refinement.

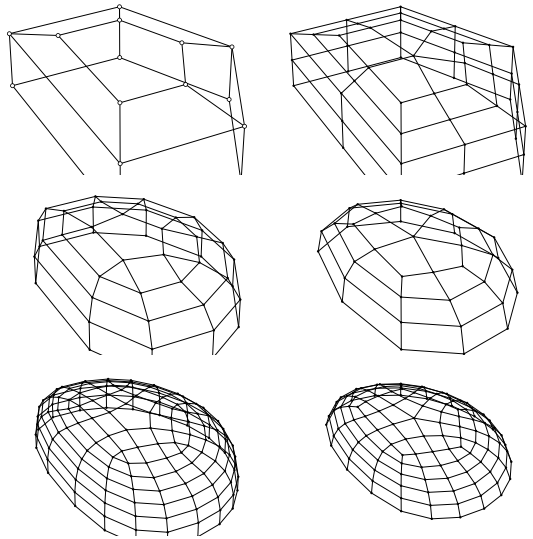


The limiting surface consists of a countable number of tensor product polynomial patches of degree (n, n), whose parametrisations are joined with continuity of derivatives of order $(n-1)$.

Extraordinary element of the mesh are non-quadrangular facets and internal vertices incident with a number of edges other than 4. After doubling all extraordinary elements bare facets. Averaging produces a mesh, whose extraordinary vertices correspond to extraordinary facets of the input mesh and all extraordinary facets correspond to extraordinary vertices. Thus, if n is odd, then all extraordinary elements of a refined mesh are vertices, and if n is even, all extraordinary elements are facets.

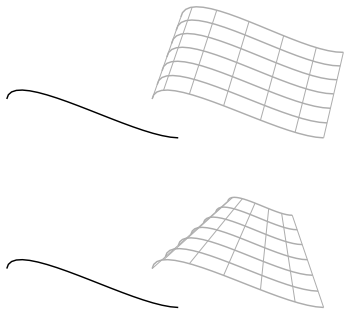
The number of extraordinary elements cannot increase in subsequent refinement operations.

Two variants most popular are called the Doo–Sabin algorithm (for $n = 2$) and Catmull–Clark (for $n = 3$).



Some constructions of surfaces

Sweeping is done by moving a curve or a surface (called the **cross-section**) along a line segment called **guide**. The motion may be done together with varying scaling.



Possible generalisations are:

- The guide may be a curve.
- Each point of the guide may be associated with a different transformation of the cross-section.
- During the motion the cross-section may change its shape.

After taking a curved guide we obtain the swept surface described by the following formula:

$$\boldsymbol{s}(u,v) = \boldsymbol{p}(u) + \boldsymbol{x}_1(u)x_q(v) + \boldsymbol{x}_2(u)y_q(v) + \boldsymbol{x}_3(u)z_q(v).$$

Five curves are involved here: the **guide** $\boldsymbol{p}(u)$, three **auxiliary guides**, $\boldsymbol{x}_1(u)$, $\boldsymbol{x}_2(u)$, $\boldsymbol{x}_3(u)$ and the **cross-section** $\boldsymbol{q}(v) = (x_q(v), y_q(v), z_q(v))$. The formula above may be rewritten as

$$\begin{bmatrix} \boldsymbol{s}(u,v) \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1(u) & \boldsymbol{x}_2(u) & \boldsymbol{x}_3(u) & \boldsymbol{p}(u) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{q}(v) \\ 1 \end{bmatrix},$$

hence, for all u the corresponding points of the guides define an affine transformation, applied to the cross-section.

Assume that all curves are B-splines and all guides are represented using the same basis (i.e., they have the same degree n and the same knot sequence):

$$\begin{aligned}\boldsymbol{p}(u) &= \sum_i \boldsymbol{p}_i N_i^n(u),\\ \boldsymbol{x}_k(u) &= \sum_i \boldsymbol{x}_{ki} N_i^n(u), \quad k = 1,2,3,\\ \boldsymbol{q}(v) &= \sum_j \boldsymbol{q}_j N_j^m(v).\end{aligned}$$

By substituting the expressions above to the formula for generalised swept surfaces we obtain the control points of the swept B-spline surface $\boldsymbol{s}(u,v)$:

$$\boldsymbol{s}(u,v) = \sum_i \sum_j \boldsymbol{d}_{ij} N_i^n(u) N_j^m(v), \quad \begin{bmatrix} \boldsymbol{d}_{ij} \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_{1i} & \boldsymbol{x}_{2i} & \boldsymbol{x}_{3i} & \boldsymbol{p}_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{q}_j \\ 1 \end{bmatrix}.$$

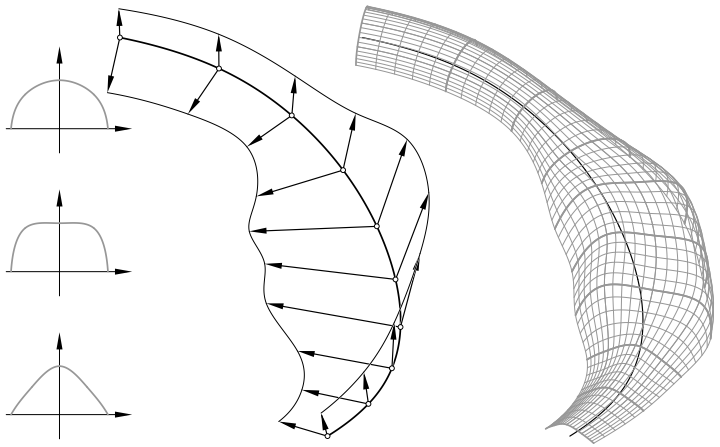
We can compute the control points and draw the surface using an appropriate procedure (if we have one), but to just compute points of this surface it is better to use the formula which defines the sweeping construction. It may be done also by a shader.

Further generalisation is obtained by modifying the cross-section as it is moved along the guides. We can take

$$\boldsymbol{s}(u,v) = \boldsymbol{p}(u) + \boldsymbol{x}_1(u)x_q(u,v) + \boldsymbol{x}_2(u)y_q(u,v) + \boldsymbol{x}_3(u)z_q(u,v).$$

The cross-section here is a one-parameter family of curves, $\boldsymbol{q}(u,v) = (x_q(u,v), y_q(u,v), z_q(u,v))$, i.e., formally it is a surface patch. If it is a B-spline patch, then finding its control net is a task more difficult (and costly), as the formula above cannot be directly rewritten as the tensor product. But to evaluate the parametrisation, i.e., compute points of the swept surface patch, we can use this formula directly.

Another possibility is to approximate the surface by a lofted surface.



Lofting (or **skinning**) is a construction of a surface of interpolation, having a set of prescribed curves of constant parameter. Given the knots (of interpolation) $u_n, \ldots, u_{N-n}$ and B-spline curves $\boldsymbol{q}_i(v)$ for $i = n, \ldots, N-n$ we shall obtain a parametrisation $\boldsymbol{s}$ of a surface patch such that $\boldsymbol{s}(u_i, v) = \boldsymbol{q}_i(v)$ for all i.

We assume that all curves $\boldsymbol{q}_i$ are B-splines which are represented using the same basis of degree m. After taking $n = 3$ we shall construct a lofted B-spline surface $\boldsymbol{s}$ of degree $(3, m)$.

A tensor product patch may be seen as a "curve whose points are curves". It allows us to reduce the problem to finding a B-spline curve of interpolation. It is supposed to pass through the *points* $\boldsymbol{q}_i$, being curves represented by their control polylines. The control points of this curve are control polygons of B-spline *curves*, represented by control polylines being columns of the control net of $\boldsymbol{s}$.

We shall take $u_1 = u_2 = u_3$ and $u_{N-3} = u_{N-2} = u_{N-1}$. Then we can write the following system of linear equations:

$$\begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & N_1^3(u_4) & N_2^3(u_4) & N_3^3(u_4) & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & N_{N-7}^3(u_{N-4}) & N_{N-6}^3(u_{N-4}) & N_{N-5}^3(u_{N-4}) & & & \\ & & & & & 1 & & & \\ & & & & & & 1 & & \end{bmatrix} \begin{bmatrix} \boldsymbol{d}_0 \\ \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \\ \vdots \\ \boldsymbol{d}_{N-6} \\ \boldsymbol{d}_{N-5} \\ \boldsymbol{d}_{N-4} \end{bmatrix} = \begin{bmatrix} \boldsymbol{q}_3 \\ \boldsymbol{d}_1 \\ \boldsymbol{q}_4 \\ \vdots \\ \boldsymbol{q}_{N-4} \\ \boldsymbol{d}_{N-5} \\ \boldsymbol{q}_{N-3} \end{bmatrix}.$$

The points $\boldsymbol{d}_1$ and $\boldsymbol{d}_{N-5}$ may be chosen arbitrarily (which is a small problem).
The nonzero coefficients of the mtrix of the system may be obtained as follows:

$$\begin{aligned} N_{k-3}^3(u_k) &= \frac{(u_{k+1} - u_k)^2}{(u_{k+1} - u_{k-2})(u_{k+1} - u_{k-1})}, \\ N_{k-2}^3(u_k) &= \frac{u_k - u_{k-2}}{u_{k+1} - u_{k-2}} \frac{u_{k+1} - u_k}{u_{k+1} - u_{k-1}} + \frac{u_{k+2} - u_k}{u_{k+2} - u_{k-1}} \frac{u_k - u_{k-1}}{u_{k+1} - u_{k-1}}, \\ N_{k-1}^3(u_k) &= \frac{(u_k - u_{k-1})^2}{(u_{k+2} - u_{k-1})(u_{k+1} - u_{k-1})}. \end{aligned}$$

Particular cases of swept surfaces are **spherical products**.
Given two planar parametric curves, called **equator** and **meridian**,

$$\boldsymbol{e}(u) = \begin{bmatrix} x_{\boldsymbol{e}}(u) \\ y_{\boldsymbol{e}}(u) \end{bmatrix} \quad \text{i} \quad \boldsymbol{m}(v) = \begin{bmatrix} x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{m}}(v) \end{bmatrix},$$

we define the surface patch having the parametrisation

$$\boldsymbol{p}(u, v) = \begin{bmatrix} x_{\boldsymbol{e}}(u) x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{e}}(u) x_{\boldsymbol{m}}(v) \\ y_{\boldsymbol{m}}(v) \end{bmatrix}.$$

If $\boldsymbol{e}$ is the unit circle with the centre at the origin of the coordinate system and $\boldsymbol{m}$ is a halfcircle whose end points are located on the y axis, then the spherical product is a sphere.

If $\boldsymbol{e}$ is the unit circle as above, then the spherical product is a surface of revolution, the meridian $\boldsymbol{m}$ is its generatrix.

If $\boldsymbol{e}$ and $\boldsymbol{m}$ are Bézier or B-spline curves,

$$\boldsymbol{e}(u) = \sum_i \begin{bmatrix} x_{\boldsymbol{e}i} \\ y_{\boldsymbol{e}i} \end{bmatrix} N_i^n(u), \qquad \boldsymbol{m}(v) = \sum_j \begin{bmatrix} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{m}j} \end{bmatrix} N_j^m(v),$$

then their spherical product is a patch with the control points

$$\boldsymbol{p}_{ij} = \begin{bmatrix} x_{\boldsymbol{e}i} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{e}i} x_{\boldsymbol{m}j} \\ y_{\boldsymbol{m}j} \end{bmatrix}.$$

If $\boldsymbol{e}$ and $\boldsymbol{m}$ are rational curves represented by the homogeneous curves

$$\boldsymbol{E}(u) = \sum_i \begin{bmatrix} X_{\boldsymbol{e}i} \\ Y_{\boldsymbol{e}i} \\ W_{\boldsymbol{e}i} \end{bmatrix} N_i^n(u), \qquad \boldsymbol{M}(v) = \sum_j \begin{bmatrix} X_{\boldsymbol{m}j} \\ Y_{\boldsymbol{m}j} \\ W_{\boldsymbol{m}j} \end{bmatrix} N_j^m(v),$$

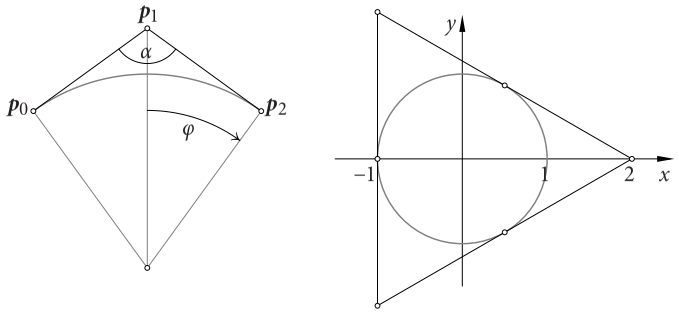
then the spherical product is a rational surface represented by the homogeneous patch whose control points ar

$$\boldsymbol{P}_{ij} = \begin{bmatrix} X_{\boldsymbol{e}i} X_{\boldsymbol{m}j} \\ Y_{\boldsymbol{e}i} X_{\boldsymbol{m}j} \\ W_{\boldsymbol{e}i} Y_{\boldsymbol{m}j} \\ W_{\boldsymbol{e}i} W_{\boldsymbol{m}j} \end{bmatrix}.$$

It may be useful if we need a surface of revolution, as the circle has no polynomial parametrisation—but it does have rational parametrisations.

The circle arc corresponding to the angle 2φ may be represented as a rational Bézier curve of degree 2, whose control polyline consists of two line segments of the same length, forming the angle $\alpha = \pi - 2\varphi$. The centre of the circle is at the intersection of lines perpendicular to these line segments, passing through the points $\boldsymbol{p}_0$ i $\boldsymbol{p}_2$. Their weights ought to be 1 and then the weight of $\boldsymbol{p}_1$ must be $\cos\varphi$.

The entire circle may consist of three arcs of the same length; then $\varphi = \frac{\pi}{3} = 60°$ and $\cos\varphi = \frac{1}{2}$.



Representations of 3D scenes

Three-dimensional objects are

- **closed-volume objects** (solids),
- **open-volume objects** (curves, surfaces),
- **volumetric objects**, partially transparent (clouds, smoke, flames, fur).

Images of closed-volume objects are obtained by drawing their surfaces, i.e., boundaries. A viewer may see only one side of such surfaces.

Another classification distinguishes **primitives** and **compound objects**; the former have direct description, the latter are obtained from simpler primitives.

The notion of a primitive needs an explanation, as whether an object is a primitive, depends on the context. For constructive solid geometry, discussed later, a primitive may be a polyhedron simple enough to be described directly, i.e., a cube or a prism. In an OpenGL application a primitive is a set of vertices (drawn as points), line segments, triangles or patches, also a line strip, a triangle strip, or a triangle fan. But a primitive for the clipping or rasterisation stage of the rendering pipeline a primitive is a single point, line segment or triangle.

A representation of a scene made of a number of objects is a data structure serving many purposes:

- drawing the scene (and acceleration of the process, e.g., by skipping the objects temporarily obscured),
- lighting computations,
- data amplification (e.g., producing details, instancing),
- placing the objects in space in a specific order,
- motion simulation,
- collision detection,
- changing states of the objects and their spatial configuration.

One data structure may be inappropriate for all these purposes, which may lead to another one: producing from a basic data structure auxiliary ones, specialised for particular tasks.

Polyhedral solids are closed-volume objects, whose boundary is made of planar polygonal facets. A polygon may be divided into triangles; hence, to draw the solid one needs a list of triangles. But other tasks may require a more detailed information, in particular about neighbourhood of the facets.

A boundary representation of a polyhedron may consist of arrays of **vertices**, **edges** and **facets**.

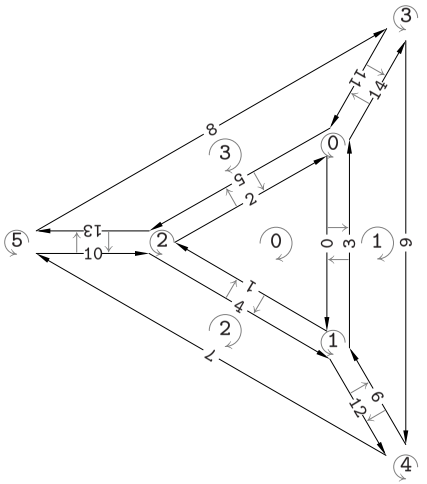
A **vertex** has a location in space and other attributes, like normal vector, colour, texture coordinates etc. It may also have a list of (identifiers) of incident edges.

An **edge** is described by identifiers of its vertices and facets. A convenient representation of an edge is a pair of **halfedges**. Each halfedge is oriented, i.e., it has the origin and end vertex and it belongs to only one facet. Each halfedge has also the identifier of its "other half", having the opposite orientation.

Orientation of halfedges is beneficial, because it simplifies searching some information. For instance it is easier to walk around a facet along its oriented halfedges.

A **facet** may contain a list of vertex identifiers or edges (or halfedges), it may also have attributes like normal vector.

One can admit facets being arbitrary polygons (even not simply-connected, i.e., with polygonal holes), but it makes the representation very complicated. By restricting the representation to polygons without holes makes it possible to store the list of edges (or halfedges) in a single one-dimensional array.



Vertices

```
0: 0, 14, 5
1: 1, 12, 3
2: 2, 13, 4
3: 11, 6
4: 9, 7
5: 10, 8
```

Facets

```
0: 0, 1, 2
1: 3, 14, 6, 9
2: 4, 12, 7, 10
3: 5, 13, 8, 11
```

Halfedges

```
 0: (0, 1, 0, 3)
 1: (1, 2, 0, 4)
 2: (2, 0, 0, 5)
 3: (1, 0, 1, 0)
 4: (2, 1, 2, 1)
 5: (0, 2, 3, 2)
 6: (3, 4, 1, -1)
 7: (4, 5, 2, -1)
 8: (5, 3, 3, -1)
 9: (4, 1, 1, 12)
10: (5, 2, 2, 13)
11: (3, 0, 3, 14)
12: (1, 4, 2, 9)
13: (2, 5, 3, 10)
14: (0, 3, 1, 11)
```

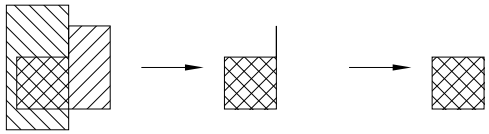
Constructive solid geometry

A rectangular parallelpiped, prism or pyramid, also a ball, cylinder or cone may be taken as primitives, as their description is rather simple. More complicated shapes may be too complicated to be described explicitly "by hand". Therefore their descriptions are obtained by computers: primitives and objects obtained from primitived are taken as arguments of set operations, which may produce quite complex objects.

Set operations are complement, union, difference, intersection and symmetric difference. Procedures of finding the complement and union or intersection make it possible to obtain results of all other set operations.

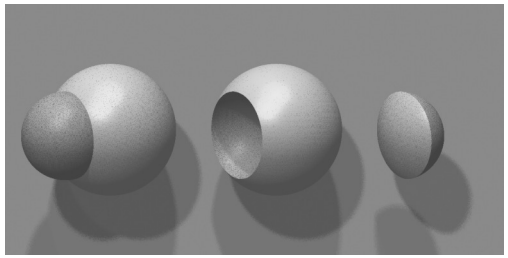
Constructive solid geometry, CSG, is obtained by subjecting the results of set operations on geometric figures to **regularisation**. It is defined mathematically as taking the closure of the inside of the figure. In this way **regular solids** are obtained: any neighbourhood of each point of the boundary of such a figure contains points of the inside and of the complement of the figure.

The idea of regularisation may be seen using an example of intersection of planar polygons:

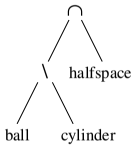


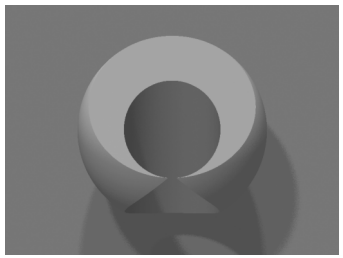
Regularisation of a 3D solid causes rejecting all points, curves and surfaces not bounding the inside, and including the boundary of what is left.

Elementary CSG operations (union, difference and intersection) are shown below:



More complicated shapes are described using set expressions with many operations. Such an expression may be represented as a binary tree. Such a tree may be implemented as a data structure with pointers.





The boundary representation of a polyhedron in the form of a mesh made of vertices, halfedges and facets makes it easy to find the **complement**. As the boundary is oriented (due to the orientation of halfedges), the normal vectors of convex facets, obtained as vector products of the halfedges, are oriented outside (or inside) of the solid. To find its complement, it suffices to invert the orientation of all halfedges.

Regularisation in this case does not involve any computation.

To find the **intersection** of two solids one has to use an algorithm being a three-dimensional generalisation of the Weiler–Atherton algorithm.

1. For each pair of facets, such that one belongs to the first and the other to the second solid, find their intersection. It may be the empty set, a point, a line segment, a polygon or (if the facets are not convex) the union of a number of points, line segments or polygons. If the intersection of facets consists of polygons, we need to find the line segments making the boundary of such polygons.

2. Using the line segments found in the previous step we divide the facets into connected polygons. The inside of each polygon is entirely inside, outside or on the boundary of the other solid. The polygons with holes should be divided to simply-connected ones.

3. Now we build a representation of a **facet neighbourhood graph**, whose vertices are the above-mentioned polygons and the edges are line segments found in the first step of the algorithm.

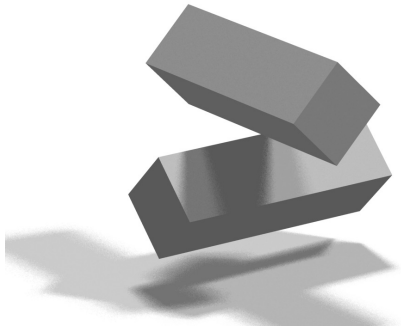
4. Regularisation is done by rejecting all vertices of the graph (i.e., polygons) not adjacent to the inside of the intersection of our solids.

5. After choosing a vertex of the graph not visited yet we check, whether it belongs to the boundary of solid intersection.

6. If it does, then using the DFS or BFS method the graph is searched and its vertices (i.e., polygons) which are parts of the boundary of intersection are output.

   Steps 5 and 6 are repeated until all vertices are visited.

The most troublesome part of the algorithm implementation may be making it robust in the face of particular cases like the one depicted below.



Trees and scene graphs

A scene representation should take into account the **object hierarchy**. Basically, to draw the scene a linear list of objects is sufficient, but graphical applications may have special demands.

Firstly, some objects are made of parts that may be manipulated, and they may be parts of bigger aggregates. A natural representation of a hierarchy is a tree, whose root represents the entire scene and whose leaves correspond to the simplest objects.

Tree vertices may have the following attributes:

A geometric transformation, which describes the transition between the systems of coordinates associated with the current tree vertex and the vertex above; the transformation of the root may represent the transition to the world coordinate system.

One can associate transformations with edges of the tree, which makes it easier to implement **kinematic linkages**, with animated transformations.

Bounding volume, most often a ball or a box (a rectangular parallelpiped). An **axis aligned bounding box** (AABB) may easily be found for polyhedral solids or B-spline surfaces (using the convex hull property).

Bounding volumes are useful for the rendering process; if a bounding volume is obscured, then all objects inside are invisible and they need not be drawn. Another application is **collision detection**; if two bounding volumes are disjoint, no object inside one of them collides with any object contained in the other bounding volume.

A simplified object representation, to be drawn if the image of the object is small; the choice of the **level of detail** (LOD) may be done based on the size of the image of the bounding volume.

Simplified representations may be present on many levels of the tree, e.g., a car model at the highest level may be a textured rectangular parallelpiped, at the middle level a wheel may be just a cylinder, and the most detailed representation, to be used for close-ups, may consist of the nave, spokes, rim, valve and tyre with tread.

CSG operation. A tree vertex may store an identifier of a CSG operation to be applied to solids represented by roots of the subtrees. It may be useful if the rendering method admits such a representation of CSG solids, typically it is the ray tracing algorithm.

If the scene is rendered using the depth buffer (as in OpenGL), then the tree representation of a CSG solid should be a separate data structure.

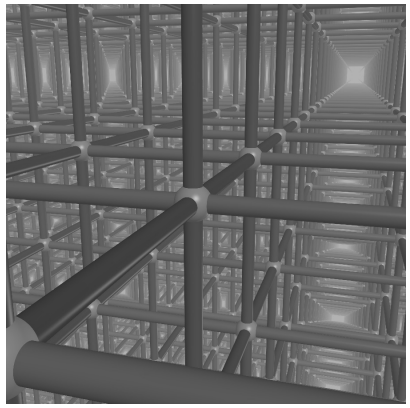
Often scenes contain a number of copies of an object. For example a car may have four identical wheels and there may be several cars of the same model, etc. Also, all leaves of a tree may have the same shape and colour, which should be specified once and drawn in many **instances**.

Directed acyclic graphs, DAG, are more general entities than trees; in a tree each vertex other than the root has only one parent. A vertex of DAG may be pointed by more than one vertex, the only restriction is the absence of cycles. It is a natural representation of scenes with repeated objects. A composition of transformations along each path from the root to a vertex should be different, resulting in a different position of each instance of the object represented by this vertex. Plenty of space in memory may be saved using a DAG.

It is assumed that a scene hierarchy DAG has a root, which is a starting point for the DFS procedure used to draw all objects of the scene. Such a procedure has to search all paths of this graph in order to display all instances of objects.

The scene description using a DAG may be parameterised. Any attribute of objects may be a parameter, e.g. the colour of the car body (then, cars having the same geometric model may have different colours).

The parameters may be also articulation parameters; in this way each car may have a different angle of turning the wheels, some cars may have doors open, etc.



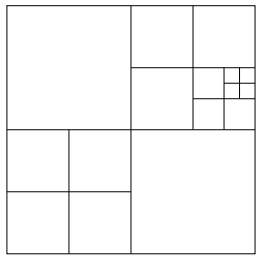
A picture shows a scene made of 2^{16} balls and $3 \cdot 2^{11}$ cylinders. The DAG representing it has 93 vertices. All vertices other than the root were pointed by two vertices at the higher level.

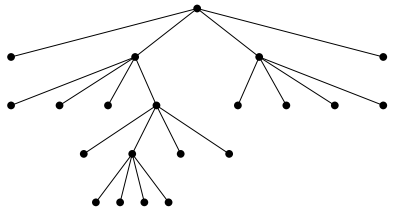
Binary trees, quadtrees and octrees

The complexity of many problems of computational geometry depends on the distribution in space of objects processed. Various algorithms of solving such problems are more efficient if they can "separate" the objects in order to deal at any moment with a small number of them.

One possibility is to use a scene hierarchy tree with bounding volumes stored in the tree vertices. Another possibility is to introduce such a spatial hierarchy by recursively dividing the space into cells. The space subdivision ought to be adaptive, the goal is to obtain cells with short lists of objects inside.

Quadtrees are often used for planar problems. If the objects are located in a bounded area, then a bounding rectangle is found for this area and then it is recursively divided into quarters. Each plane cell is thus a rectangle, which may be further divided until the number of objects incident with the cell is small enough or a subdivision depth limit is reached.





Applications:

- Searching areas (e.g., in spatial information databases).
- Constructive geometry of planar figures.
- Visibility algorithms.
- Detcting collisions of objects on the plane.

A three-dimensional analog of quadtrees are **octrees**. Here, a rectangular parallelpiped is recursively divided into octants, which are rectangular parallelpipeds twice smaller. Each vertex of the octree which is not a leaf has 8 pointers to the roots of subtrees.

Octrees are popular in ray tracing systems and in collision detection. The purposes are to reduce the number of pairs ray/object tested for intersection and to reduce the number of pairs of objects tested for collisions.

Typical subproblems solved using quadtrees and octrees are:

- Searching the entire tree and testing *all* pairs of objects present in the list of objects incident with the box represented by each vertex.
- Searching a tree vertex representing a box containing a given point.
- Finding the boxes intersecting a given figure (in particular a halfline, in ray tracing).

  To solve this problem one has to find neighbouring boxes, having a common edge or a facet with a given box.

One property of quadtrees and octrees is that boxes neighbouring a given box have either a common edge or facet or the intersection of the incident edges or facets is the entire edge or facet of one of the boxes. Hence, a number of possible strategies of searching are possible:

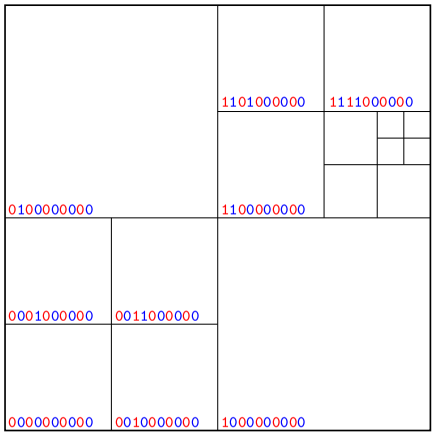
- Boxes containing given points may be found by searching always from the tree root.
- If a neighbouring box is to be found, then one can go up (from the current box) until the first box containing the neighbour is found, an then go down the tree.
- One can find the vertex representing the smallest box containing the given point using the Morton code and then go up (i.e., towards the root).

Bear in mind that none of the strategies above is always better than the others. The efficiency depends on the distribution of objects in space (which determined the tree shape).

The **Morton code** in a d-dimensional box is defined as follows: for a point $\boldsymbol{p}$, whose all coordinates are in the interval $[0,1)$, the most significant d bits are the most significant bits of the coordinates in their binary representations. They are followed by d bits taken as the second significant bits of each coordinate etc.

The length of the code is limited; it is the height of the tree multiplied by the dimension of space. 32-bit integers are capable of holding Morton codes for octrees whose height is 10. In each octree vertex we can store the Morton code of the box vertex closest to the point $(0,0,0)$. Then we can create an array of pointers to the octree leaves indexed by these Morton codes. Given a point, we can find its Morton code and use binary search of the array in order to find the tree vertex containing this point.

Example: a quadtree of height 5 has 10-bit Morton codes.



A better flexibility of recursive subdivision offer k-**d trees**, which are binary trees; the box at each level may be divided into two boxes being rectangular parallelpipeds, whose sizes may be different. At each tree level the direction of division is fixed, i.e., the division plane is either z = const or x = const or y = const for all boxes on this level. The constant is stored in the tree vertex; it is chosen so as to divide the set of objects in the box in the best way. If the objects are points, then it is best to divide them into the subsets with the same number of elements.

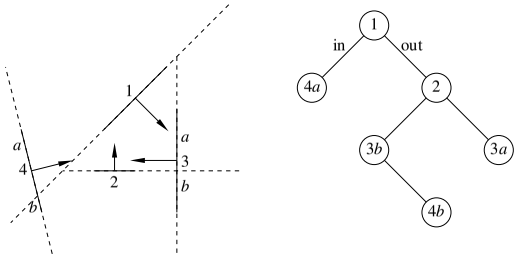
A drawback of the k-d trees is, that common boundaries of boxes do not fit to each other as in quadtrees and octrees, which narrows the choice of algorithms of searching neighbours. Also, we cannot use Morton codes here.

The most general method of dividing space with planes into cells is implemented by **binary space partition trees, BSP trees**, which (in theory) may also be used to subdivide spaces of arbitrary dimensions. The tree root representst the entire space. A representation of the hyperplane dividing it into halfspaces is stored in it. This representation contains the normal vector, with a given orientation. One of the subtrees represents the halfspace called "in", and the other halfspace (indicated by the normal vector) is "out".

Having facets, i.e., line segments in the plane or planar polygons in the 3D space, we choose the dividing hyperplanes containing the facets, in a given order. The first facet determines the space division into two cells being halfspaces. Each of the following facets is either contained entirely in one of two cells or it intersects the hyperplane separating the cells. In the latter case the facet is divided into pieces to be processed separately. The representation of the facet or its piece contained in the cell is stored in the tree vertex and then the cell is divided into two cells with the hyperplane of the facet.

If the facets in 3D space are convex, then they may be divided by the Sutherland–Hodgman algorithm into two pieces, which are also convex.

A planar example:



Facet 3 went to the cell 1in, but it intersects the line dividing this cell (the line containing facet 2). Therefore it has been divided into the pieces 3*a* and 3*b*.
A similar fate was experienced by facet 4.

- The computational cost of constructing the BSP-tree is not less than $o(n \log n)$ operations (when the tree is completely balanced and no facet is to be divided) and not greater than $O(n^3)$ operations (if all facets intersect hyperplanes of all other facets; then, for any ordering of the facets we shall obtain a tree with $\frac{1}{2}(n^2 + n)$ facet fragments).
- If the facets make the boundary of a convex polyhedron, then each of them divides the space into halfspaces such that all other facets are contained in one of the halfspaces, "in". Then, the height of the tree is equal to the number of facets. For that reason sometimes it is advantageous to introduce a number of halfplanes, artificially dividing the space into cells containing small enough subsets of the facets, to reduce the tree height.
- Choosing ordering of the facets it is best to choose for each step a facet which will cause subdivision of as few other facets as possible (none if possible) and will divide the set of the other facets into subsets of approximately the same number of elements, located on both sides of the dividing hyperplane.

A BSP tree may be used to implement the following **visibility algorithm**:

Using the DFS mthod, we search the tree, drawing the (pieces of) facets found in the tree vertices, in the infix order. For a given tree vertex, we first go into the tree on the side opposite to the viewer position (and we draw all facets located there), then the facet in the current vertex and then we go into the tree on the same side.

The above method ensures the correct final image: the dividing facet (the one in a given tree vertex) may obscure only the facets located on its side opposite to the viewer and it may be obscured only by the facets on the side of the viewer. Thus the last colour assigned to a pixel is the colour of the facet visible in that pixel. This is a variant of the so-called **painter's algorithm**.

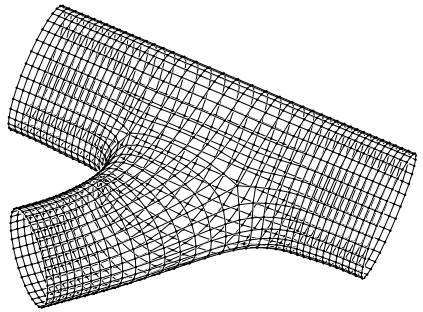
Visibility algorithms

A variety of visibility algorithms are known and even if one of them is the one most often used, others are worth knowing, as they offer possibilities hard to implement with that only one.

Among visibility algorithms one may see the following distinctions:

- **data space algorithms**, which find the visible parts of objects to be drawn, later they may be projected on a plane and drawn,
- **image space algorithms**, which determine the colour of every pixel based on the object visible at that pixel.

A further classification distinguishes **hidden line algorithms** and **hidden surface algorithms**. In the former case the algorithm finds visible parts of edges of opaque polygons and other lines which may be obscured by these polygons. Also it may be desirable to determine visibility in scenes consisting only of lines.



An important feature of any algorithm is the class of admissible data.

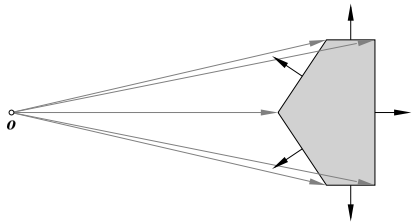
- Just flat surfaces (e.g., polygons) or also curved surfaces.
- Sets of surfaces which may intersect or may have at most common boundaries.
- Open volume objects or closed volume objects (i.e., surfaces of solids).
- Solids bounded by surfaces given explicitly or CSG trees.
- Only surfaces or also volumetric objects (clouds etc.).

The more general data may be dealt with an algorithm, the more complicated and tough to implement it has to be, in particular for parallel processing. For that reason hardware implementations use the "brute force" approach, by restricting the data to just few primitives (points, line segments and triagnles), which enforces approximating curved surfaces with large numbers of such primitives, even if the algorithm (using the depth buffer, or other) in theory makes it possible to deal with curved surfaces.

Another problem is the robustness of algorithms—some of them may produce incorrect results because of rounding errors. Wrong consequences of errors of the depth buffer algorithm are local, i.e., they do not propagate, which (together with simplicity) is the reason of popularity of this algorithm.

Data space algorithms

Some algorithms produce a part of solution, i.e., they quickly eliminate some invisible objects, thus reducing the size of the remaining data for the exact algorithm. Such an algorithm is the **back facet culling**.



If the scene consists of just one convex polyheron, then rejecting back facets gives us the full solution of the visibility problem, but in practice it is a rare situation. But statistically about a half of facets of polyhedra are rejected, thus reducing twice the size of remaining data.

In an OpenGL application one must take care that all triangles making the polyhedron surface have consistent orientation. The normal vector of the triangle having the vertices $\boldsymbol{p}_i$, $\boldsymbol{p}_j$ i $\boldsymbol{p}_k$ (specified in this order, it concerns the first triangle of all triangle strips or fans of the poluhedron), computed with the formula $\boldsymbol{n} = (\boldsymbol{p}_j - \boldsymbol{p}_i) \wedge (\boldsymbol{p}_k - \boldsymbol{p}_i)$ must be oriented outside (or inside) of the polyhedron.

We further assume that the viewer is located outside the polyhedron and the product of matrices VM (which describes the transition from the model to the viewer coordinate system) has a positive determinant.

Before drawing the object the following instructins must be executed:

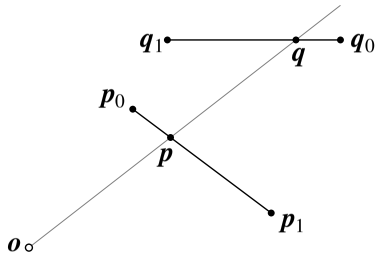
```
glEnable ( GL_CULL_FACE );
glCullFace ( GL_BACK );
glFrontFace ( GL_CCW );
```

The first enables the facet culling, the second specifies that the back facets are to be rejected and the third specifies that the front facets are the ones whose three vertices surround the triangular facet in the counterclockwise direction.

Before drawing the next object which is not the surface of a solid the back facet culling must be disabled, with the instruction

```
glDisable ( GL_CULL_FACE );
```


A basic step of hidden line and hidden surface data space algorithms is determining visibility between two line segments in space.



We need to find the points $\boldsymbol{p}$ and $\boldsymbol{q}$, located on a line passing through the viewer position $\boldsymbol{o}$. These points may be expressed as

$$\boldsymbol{p} = \boldsymbol{p}_0 + s(\boldsymbol{p}_1 - \boldsymbol{p}_0),$$
$$\boldsymbol{q} = \boldsymbol{q}_0 + t(\boldsymbol{q}_1 - \boldsymbol{q}_0).$$

The points $\boldsymbol{o}$, $\boldsymbol{p}$ and $\boldsymbol{q}$ are collinear if there exists a number u such that $\boldsymbol{p} - \boldsymbol{o} = u(\boldsymbol{q} - \boldsymbol{o})$. From that we obtain the system of equations

$$\boldsymbol{p}_0 + s(\boldsymbol{p}_1 - \boldsymbol{p}_0) - \boldsymbol{o} = u\big(\boldsymbol{q}_0 + t(\boldsymbol{q}_1 - \boldsymbol{q}_0) - \boldsymbol{o}\big).$$

After reordering we obtain an equivalent system of linear equations

$$[\boldsymbol{p}_1 - \boldsymbol{p}_0, \boldsymbol{o} - \boldsymbol{q}_0, \boldsymbol{q}_0 - \boldsymbol{q}_1]\boldsymbol{x} = \boldsymbol{o} - \boldsymbol{p}_1$$

with the unknown vector $\boldsymbol{x} = (s, u, ut)$.

Let's interpret the solutions:

- If the matrix of the system is of rank 0, then it is the zero matrix. It is not possible if both line segments have nonzero lengths.
- Rank 1 means that all columns have the same direction. If the system is consistent, then one line segment may obscure the other one, but the image of the two line segments is just one point. If the data have been filtered by back facet culling and in particular all facets such that the viewer is located in their planes have been rejected, then such a situation cannot happen.
- If the matrix is of rank 2 and the system is consistent, then (a part of) one line segment may obscure (a part of) the other line segment. To further investigate the case one needs to find four solutions with $s = 0$, $s = 1$, $t = 0$ and $t = 1$ and then test the signs of the other variables of the solution.

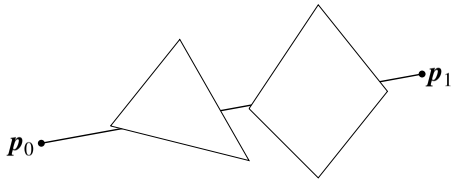
- If the matrix is full-rank, then the obscuring takes place, when $s, t \in [0,1]$ and $u > 0$. More specifically, if $u \in (0,1)$, then the point $\boldsymbol{p}$ obscures the point $\boldsymbol{q}$, and if $u > 1$, then the point $\boldsymbol{p}$ is obscured by $\boldsymbol{q}$.

Ricci algorithm (1980) is a hidden line algorithm for scenes made of convex polygons, which may have common edges, but cannot intersect otherwise. The data consist of the sets (lists) of polygons and their edges and other line segments, either located on the polygons or not (but the latter lines must not intersect the polygons). And of course the viewer location must be specified.

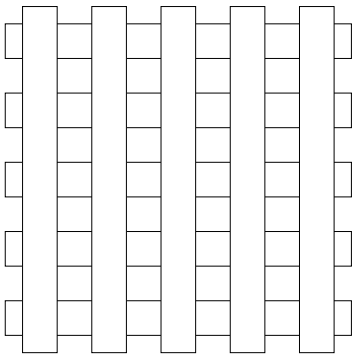
The result is the set of visible line segments. The algorithm tests all pairs of (polygon,line), but it is assumed that a polygon does not obscure its edges nor lines located on it.

For each line a list of unobscured fragments is created, initially it contains one element, the entire segment.

Each facet may obscure the line segment or its part. The convexity of the facets simplifies the algorithm, as the obscured part is connected — it is a line segment which has to be subtracted from the line segments in the list. Thus the list may become longer (if a line segment in it is to be split to three parts, with the middle part obscured) or it may shorten. It may also turn out that the entire line segment is obscured.



The computational complexity (square of the number of line segments in the scene) may be reduced by projecting the scene on a plane and using Computational Geometry techniques (sweeping) to eliminate some pairs (polygon,line) subject to the computations. The quadratic cost may however be optimal, due to the size of the result—in a scene having n line segments there may be $\Theta(n^2)$ visible parts of those line segments.



Weiler–Atherton algorithm is a hidden surface algorithm for scenes made of planar polygons. The strategy used is testing all pairs of facets, with possible acceleration based on Computational Geometry. The facet closer to the viewer is projected on the plane of the other one and then the Weiler–Atherton algorithm (described earlier) is used to find the difference of the facets is found, being the part of the more distant facet unobscured by the closer facet.

If the viewer position is the location of a light source, then the result is the enlighted part of the facets of the scene, which may be used to draw an image with shadows.

An extension of this algorithm is the **beam tracing algorithm** by Paul Heckbert and Pat Hanrahan (1984), which may produce images of scenes with some facets being mirrors. Nowadays this effect may be obtained in OpenGL, using textures, but geometric constructions (of symmetric reflections) are the same.

Appel algorithm has a reduced complexity if the number of silhouette edges is considerably less than the number of all edges. A **silhouette edge** belongs to just one facet or if it is a common edge of two facets, of which one is oriented front and the other back to the viewer.

The **degree of obscuring** of a point ***p*** in space is defined as the number of facets front-oriented to the viewer, crossed by the line segment, whose end points are the viewer position and the point ***p***.

The algorithm finds, for each edge, the points obscured by silhouette edges. For each such a point the increment of the degree of obscuring at that point along the edge is found.

The line segments of the scene are edges a graph. The points obscured by silhouette edges become additional vertices of this graph and they divide these edges so as to obtain edges representing line segments with constant degrees of obscuring. The visible line segments are those with the degree of obscuring equal to 0. A connected component of such a graph may be searched in order to find and output such line segments. Before the search it is necessary to find the degree of obscuring of just one vertex.

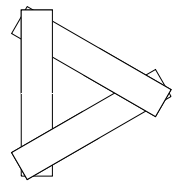
A drawback of this algorithm is that errors, caused e.g. by rounding errors, propagate. If the degree of obscuring of any edge is computed incorrectly, then all edges of the connected component will have the wrong degree of obscuring, i.e. they may be incorrectly output as visible or ignored as invisible. Making the algorithm robust makes it complicated, as it is necessary to deal with special cases, like line segments located on lines passing through the viewer position.

Image space algorithms

These algorithms produce a raster image; for each pixel either the identifier or the colour of the object visible at that pixel is determined and even if the viewer position has not changed, if we turn the viewer or change the raster resolution, the entire computation is to be repeated.

Painter's algorithm begins with sorting all facets (or edges or other objects) according to the distance from the viewer; then background is cleared and the objects are drawn, from the most distant to the closest to the viewer. In this way the last colour assigned to each pixel is the right one.

One problem is caused by the fact that distances of points of any object fill some intervals. The ordering of objects whose distance intervals are disjoint is obvious. However, if the intervals overlap, then the mutual ordering of the objects cannot be determined without additional teste. Moreover, it is possible that a correct ordering of the objects does not exists, an example of a "deadlock" is shown below.



A remedy is to split some objects into pieces. One method of sorting and avoiding deadlocks uses BSP trees.

The depth buffer algorithm (a.k.a. z-**buffer algorithm**) is the algorithm most significant in practice. Its advantages are simplicity, low computational complexity and robustness; it is particularly simple to implement in hardware.

For each pixel an additional variable is needed. Before drawing the depth buffer, i.e., the array of these variables, is cleared. All elements are set to the value representing the maximal distance in the view volume. The depth of a fragment of a rasterised primitive (point, line segment or triangle), corresponding to a pixel, is compared with the value of the variable in depth buffer. If it is less, then the colour of the fragment is assigned to the pixel and the variable obtains the new value. One can see the depth buffer algorithm as a special case of painter's algorithm, operating on a single pixel.

The depth buffer algorithm may be used to obtain the final image, with a specific lighting model, texturing etc., or it may be a part of an auxiliary computation, whose results are to be processed later.

- Drawing a scene seen from a light source position produces a **shadow area map**; objects in the the area of space obscured from the light source are in shadow. The approximate representation of this area is just the final contents of the depth buffer; one can use it to determine whether objects drawn for the final image are enlighted or not.
- One can draw (off screen) the scene viewed from the position of the viewer reflected in a mirror and then use this off-screen image as a texture for that mirror.

- Instead of the final colours of pixels one can store (in off-screen images) identifiers or other attributes of rasterised primitives. In this case the fragment shader needs not perform time-consuming computations based on sophisticated lighting models. The time of computing the colour of a fragment is wasted if some primitive processed later obscures that fragment. The final colours may be computed later, in an aditional rendering pass, and then no time is wasted on computations of colour of obscured fragments.

  The images whose pixels are used to store such data are known as **G-buffer** and the entire technique is called **deferred rendering** of **deferred shading**.
- Having images obtained as above one can use a variety of techniques of image processing to improve the final image, as it is then possible to take into account the information from the neighbourhood of each pixel. For example, a neighbourhood of the common edge of two walls in a room is a little darker. Another possible effect is obtaining glow around a flame of a candle.

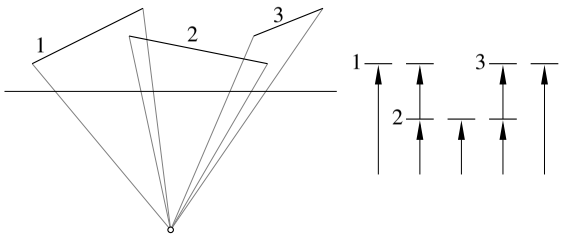
- Deferred rendering may be used to accelerate the ray tracing algorithm, as it is the fastest way to find intersection points of all primary rays (the ones originating from the viewer position) with objects of the scene.
- In the **energetic balance method** one has to compute so-called **form factors**, which are coefficients of a system of equations that decribe global lighting of a scene. Using this method we can compute effects of lighting objects by light reflected many times from objects.

Many variants of the **scanline algorithm** are now interesting mostly for historians. They were popular when RAM was an extremely limited resource: the depth buffer for an image of size 640 × 480 and 16 bits per pixel takes 600kB, while the entire RAM available in a PC was only 640kB.

The idea is to rasterise all polygons at the same time, one horizontal line of the image after another. Such a line together with the viewer position determines a plane. Visibility is resolved among line segments being intersections of polygons with the plane.

This idea makes it possible to reduce the depth buffer to a one-dimensional array, storing data for just one horizontal line of pixels of the image.

It is also possible to sort the horizontal lines with respect to increasing x-coordinates of the left end points of the line segments in the image and building a list of active line segments. The list muse be sorted according to the increasing distance from the viewer. The idea is shown below.



It is possible to obtain images with shadows using this method. For a line segment, whose image is horizontal, one needs to find the parts enlighted and obscured from light. Intersections of polyhedral solids with planes are polygons. One can resolve visibility of a line segment from the light source (the line segment may be obscured by such a polygon), which is less costly than resolving visibility of a polygon.

It is also possible to obtain images of CSG solids represented by CSG trees, wthout finding the explicit boundary representation. The problem reduces to two-dimensional problem of constructive geometry of planar polygons. As always, the reduction of the dimension of the problem simplifies it and enables using faster algorithms.

Shadow algorithms

We may want to find shadows cast by some objects to the others. If the light sources are points, then this problem is similar to resolving visibility.

Two data-space algorithms deserve a mention: the **Weiler–Atherton shadow algorithm** and the **BSP-tree shadow algorithm**. The former is just the Weiler–Atherton visibility algorithm, with the viewer position replaced by the light source position. The latter constructs a representation of the **shadow volume** in the form of a binary space partition tree. The shadow volume is the area whose points have degree of obscuring (from the light source) greater that 0.

An image-space algorithms was invented in 1977 by Frank Crow and later modified by John Carmack, used in 2000 in the famous game *Doom 3*. It assumes that all objects casting shadows are triangles.

The scene is drawn twice, and after the first pass only the contents of depth buffer is needed.

The second pass is drawing facets of **elementary shadow volumes**. Such a facet is a triangular facet of the scene or it is an unbounded quadrangle whose one edge is an edge of the triangular facet of the scene and two other edges are halflines of the lines passing through the light source position and originating at an end point of that edge. Unbounded facets are clipped at some distance and then a triangular facet closing the elementary shadow volume is attached.

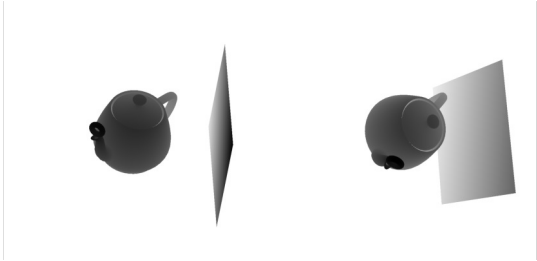
All facets of the elementary shadow volume have a consistent orientation: their normal vectors point outside the volume. When they are drawn, their visibility against the depth buffer is enabled, but writing to the depth buffer is blocked. For each pixel we need an additional counter, initially 0, and then increased or decreased by 1, depending on the orientation of the facet of the shadow volume. The most popular place, where these counters are stored is the so-called **stencil buffer**, a resource available in OpenGL. The counters are 8-bit, which is usually sufficient.

After drawing all facets of elementary shadow volumes the counter for each pixel contains an information whether the point whose depth has been stored in the depth buffer in the first pass is enlighted. It is, if the counter value is 0; this variant of the algorithm is called `depth pass` and it works correctly if the viewer is located outside of the shadow area.

The variant called *depth fail* or **Carmack's reverse** inverts the visibility tests when facets of shadow volumes are drawn. The counter is modified only after the fragment is invisible, i.e., if its depth is *greater* than the value stored in the depth buffer. The advantage of this variant is, that it gives correct results regardless of the viewer position inside or outside the shadow volume.

The last step of the algorithm is drawing the scene again, this time with the ready information about the lighting of a pixel from its counter. If G-buffer has been used, then this step may be implemented as deferred shading.

The most popular now is the **depth buffer shadow algorithm**. The scene has to be drawn as seen from the light source position. The image obtained then is irrelevant, but the final contents of the depth buffer is a representation of the shadow volume. The pictures drawn below are visualisations of the depth buffer.



An implementation of this algorithm is a part of Application 2G, which I wrote as a demo of various OpenGL possibilities.

Lighting models

Lighting is the most significant factor influencing the look of objects. The colour of a pixel of an image of enlighted object depends on properties of the object (its ability of emitting, reflecting and refracting light), properties of the light sources (location, power, spectrum and directional distribution) and the viewer (direction at which the object is seen).

Lighting models are **empirical** (the colour is computed based on a formula chosen so as to obtain a "pretty picture") and **physically based**, with the formula describing actual interaction between photons and the surface of the object.

Another classification distinguishes **local models**, taking into account only the interaction between light and surface in a small neighbourhood of a point, and **global models**, dealing with multiple reflections of light from all objects of the scene.

Local empirical models

Empirical models often use a somewhat imprecise notion of "light intensity", which may mean almost any radiometric quantity.

Lambertian lighting model, invented by J.H. Lambert in 1760, assumes that the surface of an object is opaque and perfectly mat. Let $\boldsymbol{n}$ denote the unit normal vector of the surface, i.e., of the plane tangent to the surface, which divides the space into two halfspaces. If the (unit) vectors $\boldsymbol{l}$ (towards the light source) and $\boldsymbol{v}$ (towards the viewer) are in the same halfspace, then the intensity of the light of a given wavelength is proportional to the cosine between the vectors $\boldsymbol{l}$ and $\boldsymbol{n}$.

Often it is assumed that the light source is a point, which defines a unique vector $\boldsymbol{l}$. If the light source is at infinite distance from the scene, then the vector $\boldsymbol{l}$ is the same for the entire scene and the light intensity is constant. If the distance of the light source from the scene is finite, then the vector $\boldsymbol{l}$ and the intensity I of the light vary. The incoming light intensity is often computed using the formula

$$I^{\mathrm{dir}} = \frac{I^{\mathrm{em}}}{ad^2 + bd + c},$$

where I^{em} is the intensity (power) of light emitted by the source located at the distance d from the enlighted point. I^{dir} denotes the intensity of light coming to that point from the direction of the vector $\boldsymbol{l}$. The coefficients $a > 0$, $b \geq 0$, $c > 0$ are chosen experimentally.

The intensities I^{em} and I^{dir} are vector quantities. In fact, they are functions of the wavelength, but most often they are represented by vectors having three coordinates, R, G, B, which suffices in many graphical applications.

A part of energy of light from a light source is dispersed among objects and may enlight a point from other directions. In the simplest model one may assume that the intensity of this light does not depend on direction. The total intensity of light coming directly and indirectly from all point light sources is expressed by the formula

$$L = \boldsymbol{a} \sum_{i=0}^{n-1} \left(I_i^{\mathrm{amb}} + I_i^{\mathrm{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n} \rangle| \right).$$

The factor v_i is 1 if the viewer is on the same side of the surface as the light source and 0 otherwise. Recall that the vectors $\boldsymbol{n}$ and $\boldsymbol{l}_i$ are unit vectors.

The vector $\boldsymbol{a}$ describes the ability of the surface to reflect light with a given wavelength. Again, most often it has only three coordinates, R, G, B.
The multiplication is done "coordinate by coordinate".

Phong model, invented by Bui-Tuong Phong in 1975, makes it possible to draw a surface, which is not perfectly mat. The formula is

$$L = \boldsymbol{a}\sum_{i=0}^{n-1}\left(I_i^{\mathrm{amb}} + I_i^{\mathrm{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n}\rangle|\right) + \boldsymbol{s}\sum_{i=0}^{n-1} I_i^{\mathrm{dir}} v_i W(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v}) \max\{0, \langle \boldsymbol{r}_i, \boldsymbol{v}\rangle\}^m.$$

The unit vector $\boldsymbol{r}_i$ in this formula is

$$\boldsymbol{r}_i = 2\langle \boldsymbol{n}, \boldsymbol{l}_i\rangle \boldsymbol{n} - \boldsymbol{l}_i.$$

Its direction is the direction of reflected light, provided that the surface is a perfect mirror with the normal vector $\boldsymbol{n}$.

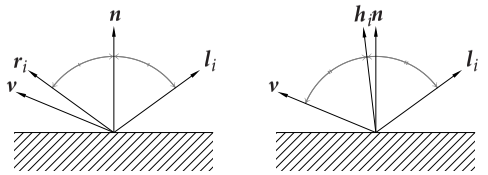
Often the function W is assumed constant, equal to 1, but sometimes it may vary with the angles between the vectors $\boldsymbol{l}_i$ and $\boldsymbol{v}$ and the vector $\boldsymbol{n}$. It is taken so as to approximate effects obtained using this model to the effects that might result from physically based models. More specifically, it might take into account the microscopic properties of the surface model (roughness) and the so-called Fresnel factor. Usually the vector $\boldsymbol{s}$ represents a constant function (with coordinates $R \approx G \approx B$), as the colour of the highlight on the surface has the colour of the incoming light.

The Blinn–Phong model (1977) is a modification of the Phong model. It uses the formula

$$L = \boldsymbol{a}\sum_{i=0}^{n-1}\left(I_i^{\mathrm{amb}} + I_i^{\mathrm{dir}} v_i |\langle \boldsymbol{l}_i, \boldsymbol{n}\rangle|\right) + \boldsymbol{s}\sum_{i=0}^{n-1} I_i^{\mathrm{dir}} v_i W(\boldsymbol{n}, \boldsymbol{l}_i, \boldsymbol{v}) \langle \boldsymbol{h}_i, \boldsymbol{n}\rangle^{2m}.$$

The vector $\boldsymbol{h}_i$ is

$$\boldsymbol{h}_i = \frac{1}{\|\boldsymbol{l}_i + \boldsymbol{v}\|}(\boldsymbol{l}_i + \boldsymbol{v}).$$

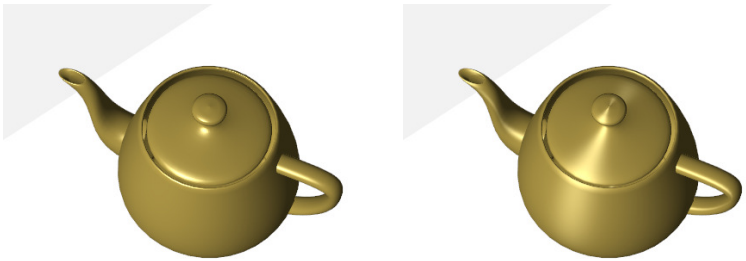


The exponent m in both models determines the "degree of polishing" the surface; the greater it is, the smaller is the highlight area.

Both models, Phong and Blinn–Phong, are **isotropic**. There is no distinguished direction on the surface. But we often have surfaces, whose roughness are directional, because of small scratches invisible by bare eye, but changing the shape of highlights. The simplest **anisotropic model** may be obtained by a modification of the Blinn–Phong model, with the vectors $\boldsymbol{h}_i$ replaced by $\tilde{\boldsymbol{h}}_i$ such that

$$\tilde{\boldsymbol{h}}_i = \frac{1}{\|\hat{\boldsymbol{h}}_i\|}\hat{\boldsymbol{h}}_i, \quad \hat{\boldsymbol{h}}_i = P_0\boldsymbol{w}_i + P_1\boldsymbol{w}_i + aP_2\boldsymbol{w}_i, \quad \boldsymbol{w}_i = \boldsymbol{l}_i + \boldsymbol{v},$$

where P_0, P_1, P_2 are orthogonal projections on the directions of the vector $\boldsymbol{n}$, a vector $\boldsymbol{r}$ tangent to the surface (having the direction of scratches) and the vector orthogonal to $\boldsymbol{n}$ and $\boldsymbol{r}$. The coefficient $a \in [0,1]$ determines the degree of anisotropy—for $a = 1$ we obtain $\tilde{\boldsymbol{h}}_i = \boldsymbol{h}_i$, i.e., the Blinn–Phong model, and by decreasing a we obtain the effect of scratches more pronounced (even if the scratches are never visible).

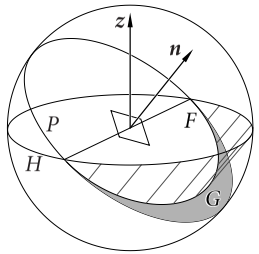


$a = 1$ $a = 0.2$

If the light dispersed in the environment is totally undirectional, then the result obtained using any of the models described above are completely shapeless parts of surface which are not enlighted directly. To obtain better results, the constant term describing the light dispersed in the environment is replaced by an integral.

The simplest model is the **hemispherical lighting**. We assume that the dispersed light is coming from all directions. Unit vectors representing these directions form the unit sphere, which consists of two hemispheres: "upper" and "lower". The light coming from the upper hemisphere is usually brighter and it has the colour of the sky (white or bluish). The light coming from the lower hemisphere has the colour of the ground (floor, rock, asphalt, grass etc.). The intensity of light from each hemisphere is assumed constant.

The sphere may also be divided into two hemispheres with the plane tangent to the surface. The viewer can see one or the other side of the surface, depending on which of the two hemispheres contains the vector $\boldsymbol{v}$.



Above we can see the two divisions of the sphere: by the circle H, the "horizon" contained in the horizontal plane (whose normal vector $\boldsymbol{z}$ indicates the "zenith") and by the circle P in the plane tangent to the surface, i.e., perpendicular to the vector $\boldsymbol{n}$. Given the two intensities, I^{sky} and I^{ground} coming from the upper and lower hemispheres, we are going to calculate the total intensity of the light reflected towards the viewer, by blending the two intensities in the right proportion.

We assume that light is reflected according to the Lambertian model, i.e., the surface is perfectly smooth and mat. Let $\boldsymbol{l}$ be a vector from the hemisphere bounded by the circle P, containing also the vector $\boldsymbol{v}$. Consider a small piece of the sphere containing the vector $\boldsymbol{l}$; it consists of unit vectors having almost the same directions that $\boldsymbol{l}$.

The intensity of light incoming from all (almost identical) directions, i.e., from this piece of the sphere is a constant, and the contribution in the light reflected from the surface is proportional to the cosine of the angle between the vectors $\boldsymbol{l}$ and $\boldsymbol{n}$, which is the scalar product of these vectors. The intensity has to be multiplied by the measure of this piece of the sphere and the cosine. But the product of the measure of the piece and the cosine is the area of the orthogonal projection of the piece on the surface's tangent plane plane.

The projections of all pieces of the intersecting of the upper hemisphere with the viewer's hemisphere fill the area F on the picture—it is the union of a halfdisc and the area whose boundary is a halfellipse being the projection of the half of the horizon (the circle H) on the tangent plane.

The projections of the pieces of intersection of the lower hemisphere with the viewer's hemisphere fill the area G. Thus the proportion of blending the lights from the upper and lower hemisphere is the proportion of the areas F and G. The intensity of the reflected light in the hemispherical lighting model is

$$\boldsymbol{a}\big((1-t)I^{\text{sky}} + tI^{\text{ground}}\big).$$

If we denote the angle between $\boldsymbol{z}$ and $\boldsymbol{n}$ with the symbol ϑ, then we (i.e., the shader) can compute

$$t = \begin{cases} (1-\cos\vartheta)/2 = (1-\langle \boldsymbol{z},\boldsymbol{n}\rangle)/2 & \text{if } \langle \boldsymbol{v},\boldsymbol{n}\rangle > 0, \\ (1+\cos\vartheta)/2 = (1+\langle \boldsymbol{z},\boldsymbol{n}\rangle)/2 & \text{if } \langle \boldsymbol{v},\boldsymbol{n}\rangle \le 0. \end{cases}$$

Adding the term which describes the Lambertian reflection of the hemispherical illumination may improve also images of glossy surfaces.

Functions more complicated than constants in two hemispheres may be used to obtain images of objects placed in compound environments, made of buildings, room walls etc. Such functions may be represented by cube textures, evaluated based on the normal vector $\boldsymbol{n}$.

Physically based models

The transport of light in space is described by **quantum electrodynamics**, which is a theory too complicated to be used directly in Computer Graphics.

Geometric optics is a simplification. Photons in a homogeneous medium travel along straight lines. Further, **linear optics** assumes that unless photons are absorbed by the medium, they preserve their energy.

The simplifications ignore phenomena like diffraction, interference and luminescence.

Two methods of quantitative light description are in use:

- **Radiometry,** focused on power of light (measured in watts) and the derived quantities, like power density on a surface,
- **Photometry,** where brightness is measured (in lumens), being the subjective impression of humans, whose eyes react in a different way to the lights of various wavelengths and the same power.

The mapping of power of light to the impression of brightness is done by human eyes. The computer screen emits light with a specific power, therefore radiometry is the approach most important in image synthesis.

Flux is the energy stream measured in watts [W]. If the medium (e.g., the air) is completely transparent, then we can deduce that the total power (i.e., flux) of light entering some area of space will be distributed on the part of the area boundary, through which it leaves the area.

Consider a point emitting monochromatic light. The flux passing through all spheres, whose centre is that point, is identical.

Radiant intensity is measured in watts per steradian [W/sr]. For a piece of the *unit* sphere it is the flux of light passing through this piece divided by the area of the piece.

If we consider a piece of the sphere of an arbitrary radius r, then the flux has to be divided by the area of the piece and multiplied by r^2. The area of the piece divided by square of the radius is a measure of the **solid angle** of the piece. Its unit is called **steradian** [sr]. The measure of the **full solid angle** is 4π sr.

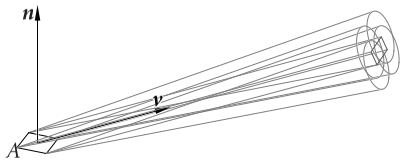
Radiant intensity of light emitted by a point in various directions may be different, i.e., it may be a non-constant function of the direction. But in a perfectly transparent medium the flux of light passing through any piece of the unit sphere is the same as the flux of light passing through the piece of another sphere obtained by a homotetia.

Now we consider a light source which is not a point. We assume that it is a piece of a planar surface, having the unit normal vector $\boldsymbol{n}$. The element as well as its area will be denoted by the letter A.

The vector $\boldsymbol{v}$ determines the direction from a point of the element A to the viewer located at a distance r, large in comparison to the size of the element A. For the viewer, the element A takes the solid angle of $A|\cos \measuredangle(\boldsymbol{n},\boldsymbol{v})|/r^2$; if the vectors $\boldsymbol{n}$ and $\boldsymbol{v}$ have the same direction, then this angle has the measure A/r^2, and if the two vectors are mutually orthogonal, then the solid angle is 0.

Shortened element measure is the product $A|\cos \measuredangle(\boldsymbol{n},\boldsymbol{v})|$ [m^2]. Hence, the solid angle of the element A seen by the viewer is the ratio of the shortened element measure and the square of the distance r between the element and the viewer.

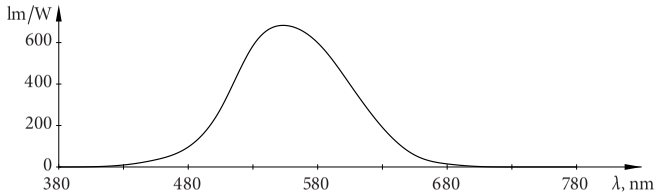
Radiance of the reflected or radiated light leaving the element is equal to the flux divided by the measure of the solid angle, within which the photons travel and the shortened area of the element. More precisely, it is the limit of the above ratio with the solid angle and the element size tending to zero. Radiance at a given point is thus a function of the unit vector $\boldsymbol{v}$, measured in watts per square meter and steradian [W/(m^2 sr)].



Irradiance is the density of power on the enlighted area; flux is divided by the *unshortened* area. Irradiance is measured in watts per square meter [W/m^2].

The above radiometric quantities have their corresponding photometric quantities:

- Flux [W]—**light stream** [lm].
- Radiant intensity [W/sr]—**lightness** [cd = lm/sr].
- Radiance [W/(m^2 sr)]—**luminance** [lm/(m^2 sr) = cd/m^2].
- Irradiance [W/m^2] — **illuminance** [lm/m^2].



The function called **luminous efficacy** describes a relation between the quantities above, which depends on the wavelength, λ. The maximal value of 683 lm/W is taken at $\lambda = 555$ nm.

Let's prove that the subjective impression of brightness of a surface depends on the radiance L of the light coming from that surface. Obviously the impression depends on the irradiance ot the area of retina in the human's eye on which the surface's image is formed.

Consider a surface element A seen from a distance r. For the viewer the element takes the solid angle $A|\cos \measuredangle(\boldsymbol{n}, \boldsymbol{v})|/r^2$. The area of the image of the element is proportional to this solid angle, with some constant factor C (if the element is directly in front of the viewer).

On the other hand, the pupil of the viewer's eye seen from the element A takes the solid angle $\pi R^2/r^2$, where R is the radius of the pupil. Thus the flux of light from the element A coming to the eye is the product of the radiance, the shortened area of the element and of the last solid angle. By dividing the flux by the area of image of the element, we obtain the irradiance of the retina:

$$I = \left(LA|\cos \measuredangle(\boldsymbol{n}, \boldsymbol{v})|\pi R^2/r^2\right)/\left(CA|\cos \measuredangle(\boldsymbol{n}, \boldsymbol{v})|/r^2\right) = \frac{\pi R^2}{C} L.$$

Square of the distance r cancels out. Hence, indeed, irradiance of the retina is proportional to the radiance L.

Why other stars are not as bright as the Sun? Because their (perfectly sharp) images on the retina would be much smaller than the diameter of a single light receptor.

Bidirectional scattering distribution function, BSDF is a property of a surface. At a given point it is defined as the ratio of the radiance reflected or refracted on the surface and the irradiance of the light coming to the surface. Its arguments are two unit vectors, $\boldsymbol{l}$ and $\boldsymbol{v}$, which determine the directions to the light source and to the viewer (or a point enlighted by the reflected light).

Now we have come to the core of physically based lighting models. A local model is in fact the bidirectional scattering distribution function. More detailed models take into account light polarisation and then the BSDF is a vector function.

The radiance of light sent from the point $\boldsymbol{p}$ of a surface element A in the direction of a vector $\boldsymbol{v}$ is

$$L(\boldsymbol{p},\boldsymbol{v}) = L_\mathrm{e}(\boldsymbol{p},\boldsymbol{v}) + L_\mathrm{r}(\boldsymbol{p},\boldsymbol{v}) = L_\mathrm{e}(\boldsymbol{p},\boldsymbol{v}) + \int_{l\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) I(\boldsymbol{p},\boldsymbol{l})\,\mathrm{d}S.$$

L_e is the radiance of the light emitted, L_r is the radiance of the light reflected, ρ denotes the bidirectional scattering distribution function and I is the irradiance. S denotes the unit sphere, i.e., the set of all directions in space, from which light is coming to the point $\boldsymbol{p}$.

The above equation was given in 1986 by James T. Kajiya, who called it **The Rendering Equation**. If the medium (the air) is perfectly transparent, then the equation describes the radiance of the point $\boldsymbol{p}$ seen from the direction $\boldsymbol{v}$, which is the quantity to be assigned to pixels.

Bidirectional scattering distribution function is the sum of two functions known as **bidirectional reflectance distribution function** (**BRDF**) and **bidirectional transmission distribution function** (**BTDF**). For surfaces of opaque objects the latter function is zero.

The two functions below will be denoted by ρ_s and ρ_t.

Physical lighting models should satisfy two principles:

Energy preserving, according to which the flux of light reflected from a surface may only be less than the flux enlighting this surface. Some light energy is dispersed as heat.

Helmholtz principle, originating from the observation that if a photon may travel along some path in the space, another photon may travel along the same path in the opposite direction. A consequence of this principle is the symmetry of the bidirectional reflectance distribution function: $\rho_\mathrm{s}(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) = \rho_\mathrm{s}(\boldsymbol{p};\boldsymbol{v},\boldsymbol{l})$. Such a symmetry of BTDF would violate the energy preservation principle; however, it is preserved in the following formula:

$$\eta_1^2 \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) = \eta_2^2 \rho(\boldsymbol{p};\boldsymbol{v},\boldsymbol{l}),$$

where η_1 and η_2 are refraction indices of the media, in which the photon travelled respectively before and after interaction with the surface.

The flux of light enlighting an element A from the direction of a given vector $\boldsymbol{l}$ is equal to

$$E = AI(\boldsymbol{p}, \boldsymbol{l}).$$

The total flux of light that has been reflected or refracted in all directions is

$$\int_{\boldsymbol{v}\in S} L(\boldsymbol{p},\boldsymbol{v})A|\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S = AI(\boldsymbol{p},\boldsymbol{l})\int_{\boldsymbol{v}\in S}\rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v})|\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S.$$

It must be less than E (because some light energy is dispersed as heat). Hence,

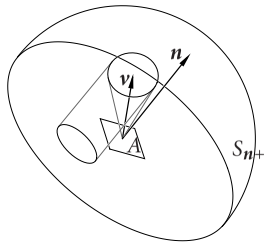
$$\int_{\boldsymbol{v}\in S}\rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v})|\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S < 1$$

for all unit vectors $\boldsymbol{l}$.

In the Lambertian model, BSDF at any point $\boldsymbol{p}$ takes only two values: a constant $\rho > 0$, if the scalar products $\langle \boldsymbol{l}, \boldsymbol{n}\rangle$ and $\langle \boldsymbol{v}, \boldsymbol{n}\rangle$ have the same sign and 0 otherwise. Note that the cosine of the angle between the vectors $\boldsymbol{l}$ and $\boldsymbol{n}$ is a factor of the irradiance, as the flux of incoming light is proportional to this cosine. One can easily check that the Lambertian model satisfies the Helmholtz principle.

The flux of light reflected from a Lambertian surface may be obtained as an integral over the hemisphere $S_{\boldsymbol{n}+}$, made of vectors $\boldsymbol{v}$ such that $\langle \boldsymbol{n}, \boldsymbol{l}\rangle\langle \boldsymbol{n}, \boldsymbol{v}\rangle > 0$. BSDF in this sphere takes a positive constant value, denoted by ρ.

No calculus is needed to integrate the flux.



$$\rho(\boldsymbol{p})\int_{\boldsymbol{v}\in S_{\boldsymbol{n}+}}|\cos\measuredangle(\boldsymbol{n},\boldsymbol{v})|\,\mathrm{d}S = \rho(\boldsymbol{p})\pi.$$

Hence, to preserve the energy conservation principle we have to take $\rho(\boldsymbol{p}) < \frac{1}{\pi}$.

The preservation of energy by the Phong and Blinn–Phong models depends on the choice of parameters for these models, but it is much harder to verify it than in the Lambertian case.

The Phong model does not satisfy the Helmholtz principle. Whether the Blinn–Phong model satisfies it, it depends on the choice of the function W chosen for this model. If it is

$$W(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v}) = |\langle \boldsymbol{n}, \boldsymbol{l}\rangle| Z(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v}),$$

with the factor $Z(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v})$ being a symmetric function, i.e., $Z(\boldsymbol{n},\boldsymbol{l},\boldsymbol{v}) = Z(\boldsymbol{n},\boldsymbol{v},\boldsymbol{l})$ for all vectors $\boldsymbol{v}$ and $\boldsymbol{l}$, then the Helmholtz principle is satisfied.

As stated before, BSDF is the sum of BRDF and BTDF:

$$\rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) = \rho_{\mathrm{r}}(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) + \rho_{\mathrm{t}}(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v});$$

If the surface is opaque then the second term is zero.

Physically based lighting models usually take the function ρ_{r} having the form

$$\rho_{\mathrm{r}} = v(k_{\mathrm{d}}\rho_{\mathrm{d}} + k_{\mathrm{s}}\rho_{\mathrm{s}}).$$

The factor v is equal to 1 or 0, depending on the fact that the light source are on the same side of the surface or not. The coefficients k_{d} and k_{s} are nonnegative and their sum is 1. The first term in the parentheses describes the diffuse reflection, while the second one is responsible for the specular reflection.

The function ρ_{s} in physical models is taken according to the formula

$$\rho_{\mathrm{s}} = \frac{DGF_\lambda}{4\langle \boldsymbol{l},\boldsymbol{n}\rangle\langle \boldsymbol{v},\boldsymbol{n}\rangle}.$$

The scalar products of the unit vectors in the denominator are cosines of angles between those vectors.

It is assumed that the surface is rough and it consists of **microfacets** visible only under a microscope. The microfacets are perfect mirrors. The surface is the graph of a function of two variables, i.e., any straight line having the direction of the vector $\boldsymbol{n}$ intersects this surface at only one point.

The factor $D = D(\boldsymbol{h})$ describes the distribution of normal vectors of the microfacets. More specifically, it determines, how many microfacets are oriented so that their unit normal vector is $\boldsymbol{h} = \frac{1}{\|\boldsymbol{l}+\boldsymbol{v}\|}(\boldsymbol{l}+\boldsymbol{v})$. Always it has to be

$$\int_{\boldsymbol{h}\in S_{\boldsymbol{n}+}} D(\boldsymbol{h}) \cos\measuredangle(\boldsymbol{h},\boldsymbol{n})\,\mathrm{d}S = 1.$$

The set $S_{\boldsymbol{n}+}$ consists of the unit vectors $\boldsymbol{h}$, such that $\langle \boldsymbol{h},\boldsymbol{n}\rangle > 0$.

The classical **Torrance and Sparrow model** from 1967 used the Gaussian distribution:

$$D(\vartheta) = ae^{-b^2\vartheta^2},$$

where ϑ denotes the angle between the normal vectors of the microfacet and the surface. Its range has to be restricted to the interval $[0,\pi/2)$. The coefficients a and b are constants. After fixing b, one has to take the matching a, which is rather troublesome.

The **Cook and Torrance model** (1981) uses the **Beckmann–Spizzichino distribution**,

$$D_{\mathrm{BS}}(\boldsymbol{h}) = \frac{e^{-(\tan\vartheta)^2/m^2}}{\pi m^2\cos^4\vartheta}.$$

The parameter m determines the roughness, its smaller value produce a better "polishing" of the surface. The computations may use the formula

$$\frac{\tan^2\vartheta}{m^2} = \frac{1-\cos^2\vartheta}{m^2\cos^2\vartheta},$$

obviously $\cos\vartheta = \langle \boldsymbol{n},\boldsymbol{h}\rangle$.

The anisotropic version, with two roughness parameters m_u, m_v is given by:

$$D_{\mathrm{BSa}}(\boldsymbol{h}) = \frac{\mathrm{e}^{-(c^2/m_u^2+s^2/m_v^2)\tan^2\vartheta}}{\pi m_u m_v\cos^4\vartheta}.$$

In the formula above c and s are the cosine and sine of the angle between the orthogonal projection of the vector $\boldsymbol{h}$ on the surface's tangent plane and a fixed vector $\boldsymbol{u}$ in this plane.

An alternative is the **Trowbridge–Reitz distribution**, popular in computer games:

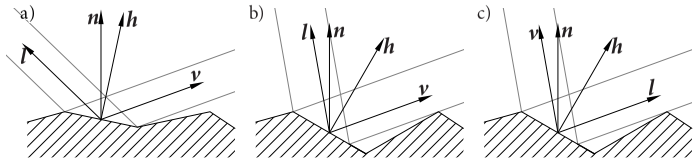
$$D_{\mathrm{TR}}(\boldsymbol{h}) = \frac{m^2}{\pi\big(1+(m^2-1)\big)^2\cos^4\vartheta}.$$

Its anisotropic version is as follows:

$$D_{\mathrm{TRa}}(\boldsymbol{h}) = \frac{1}{\pi m_u m_v\big(1+\sin^2\vartheta(c^2/m_u^2+s^2/m_v^2)\big)^2\cos^4\vartheta}.$$

There exist surfaces whose microfacets belong to several families, each having a diferent distribution of normal vectors. It may be the same distribution, e.g., Becmkann–Spizzichino, with various parameters like m. Such a surface may have been obtained by milling, honing, polishing and scratching.

To obtain such an effect on images, one can choose weights of the distributions, i.e., positive factors whose sum is 1, and take the convex combination of the distributions with these weights.



The factor G describes the degree of attenuation of parts of microfacets by other microfacets. Because of the obscuring only a part of a microfacet may be enlighted from the direction $\boldsymbol{l}$ or a part of the microfacet may be visible form the direction $\boldsymbol{v}$. Under a hypothesis that pairs of microfacets form symmetric groves one can derive the formula

$$G = \min\left\{1, \frac{2\langle\boldsymbol{h},\boldsymbol{n}\rangle\langle\boldsymbol{v},\boldsymbol{n}\rangle}{\langle\boldsymbol{v},\boldsymbol{h}\rangle}, \frac{2\langle\boldsymbol{h},\boldsymbol{n}\rangle\langle\boldsymbol{l},\boldsymbol{n}\rangle}{\langle\boldsymbol{l},\boldsymbol{h}\rangle}\right\}.$$

The hypothesis is usually false, but the formula is useful.

Finally, the Fresnel factor F_λ depends on the properties of the material, whose surface is enlighted, and more precisely on its refraction index, which depends on the wavelength.

For an unpolarised light we have the formula

$$F_\lambda = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{\big(c(g+c)-1\big)^2}{\big(c(g-c)+1\big)^2}\right),$$

where $c = \cos\vartheta_{\boldsymbol{hl}} = \langle\boldsymbol{h},\boldsymbol{l}\rangle$, $g = \sqrt{(\eta_{\lambda,2}/\eta_{\lambda,1})^2 + c^2 - 1}$, and the symbols $\eta_{\lambda,1}$ i $\eta_{\lambda,2}$ denote the refraction index of the medium, by which the light comes to the surface, and the refraction index of the enlighted object.

In practice Fresnel factors are replaced by expressions easier (and cheaper) to evaluate. Let $F_{\lambda,0}$ and $F_{\lambda,\pi/2}$ denote the Fresnel factors for the light coming to a microfacet from the direction of tits normal vector $\boldsymbol{h}$ and from a direction tangent to it. In the former case $\boldsymbol{l} = \boldsymbol{h}$, and it follows that $c = 1$ and $g = \eta_{\lambda,2}/\eta_{\lambda,1}$. In the latter case we have $c = 0$ i $g = \sqrt{(\eta_{\lambda,2}/\eta_{\lambda,1})^2 - 1}$. Hence,

$$F_{\lambda,0} = \left(\frac{\eta_{\lambda,2} - \eta_{\lambda,1}}{\eta_{\lambda,2} + \eta_{\lambda,1}}\right)^2, \quad F_{\lambda,\pi/2} = 1.$$

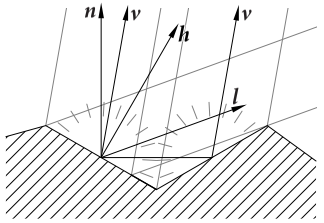
The **Schlick approximation** is given by the formula

$$F_{\lambda S} = F_{\lambda,0} + (F_{\lambda,\pi/2} - F_{\lambda,0})(1 - \cos\vartheta_{\boldsymbol{hl}})^5 = F_{\lambda,0} + (1 - F_{\lambda,0})(1 - \langle \boldsymbol{h}, \boldsymbol{l}\rangle)^5.$$

Here it suffices to specify only the function $F_{\lambda,0}$, whose argument is the wavelength λ. Often this function is represented by a vector having the coordinates R, G, B obtained from measurements of the refraction index for the red, green and blue light.

The function ρ_{d}, which describes the diffuse reflection, is most often taken according to the Lambertian model, assuming that the surface is perfectly smooth and perfectly mat. Roughness of the surface is taken into account by the model developed by **Oren and Nayar** in 1994. It assumes the presence of microfacets reflecting light in the Lambertian way, but the microfacets have different directions of the normal vector and they may obscure each other.

The model assumes that pairs of the microfacets form symmetric groves. The viewer receives the light reflected by one or two microfacets of each pair.



The distribution of the normal vectors assumed in the model is Gaussian, with the roughness parameter σ. It is necessary to integrate the light coming to the viewer, but the integrals are too hard to be computed analytically. Therefore the following approximate formulae have been proposed by the authors:

$$\rho_{\mathrm{d},1} = \frac{c_\lambda}{\pi}\left(C_1 + cC_2 \tan\beta + \big(1 - |c|\big)C_3 \tan\frac{\alpha + \beta}{2}\right).$$

The symbols used above denote the following expressions

$$\alpha = \max\{\vartheta_{\boldsymbol{l}}, \vartheta_{\boldsymbol{v}}\}, \qquad \beta = \min\{\vartheta_{\boldsymbol{l}}, \vartheta_{\boldsymbol{v}}\},$$

$$C_1 = 1 - \frac{0.5\sigma^2}{\sigma^2 + 0.33}, \quad C_2 = \frac{0.45\sigma^2}{\sigma^2 + 0.09}d, \quad C_3 = \frac{0.125\sigma^2}{\sigma^2 + 0.09}\left(\frac{4\alpha\beta}{\pi^2}\right)^2,$$

$$c = \cos\Delta\varphi, \qquad d = \begin{cases} \sin\alpha & \text{if } c \geq 0, \\ \sin\alpha - \left(\frac{2\beta}{\pi}\right)^3 & \text{else.} \end{cases}$$

The function c_λ with values in the interval $[0, 1]$ describes the fraction of photons of wavelength λ being reflected by microfacets. The symbols $\vartheta_{\boldsymbol{l}}$ and $\vartheta_{\boldsymbol{v}}$ denote the angles between the vectors $\boldsymbol{l}$ and $\boldsymbol{v}$ and the vector $\boldsymbol{n}$: there is $\cos\vartheta_{\boldsymbol{l}} = \langle \boldsymbol{l}, \boldsymbol{n}\rangle$ and $\cos\vartheta_{\boldsymbol{v}} = \langle \boldsymbol{v}, \boldsymbol{n}\rangle$.

The angle $\Delta\varphi$ is measured between the orthogonal projections of the vectors $\boldsymbol{l}$ and $\boldsymbol{v}$ on the tangent plane of the surface. If neither of the two vectors is orthogonal to the surface, then

$$c = \frac{\langle \boldsymbol{l}, \boldsymbol{v}\rangle - \langle \boldsymbol{n}, \boldsymbol{l}\rangle\langle \boldsymbol{n}, \boldsymbol{v}\rangle}{\|\boldsymbol{l} - \langle \boldsymbol{n}, \boldsymbol{l}\rangle\boldsymbol{n}\|\,\|\boldsymbol{v} - \langle \boldsymbol{n}, \boldsymbol{v}\rangle\boldsymbol{n}\|}.$$

Else $\beta = \tan\beta = C_3 = 0$, $d = \sin\alpha$ and the undefined number c is irrelevant.

The function $\rho_{\mathrm{d},1}$ is the term of ρ_{d}, which describes a single reflection of light from just one microfacet. To take into account the light sent to the direction of $\boldsymbol{v}$ after two reflections in both microfacets of the pair, one has to add the term

$$\rho_{\mathrm{d},2} = \frac{c_\lambda^2}{\pi}\frac{0.17\sigma^2}{\sigma^2+0.13}\left(1 - c\frac{4\beta^2}{\pi^2}\right).$$

In practice this term is often neglected. After substituting $\sigma = 0$, we obtain $C_1 = 1$, $C_2 = C_3 = \rho_{\mathrm{d},2} = 0$, which will reproduce the Lambertian light reflection.

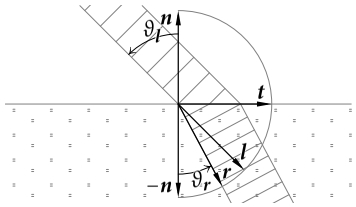
If the object is made of a transparent material (glass, ice etc.), and its surface is rough, then the **bidirectional transmission distribution function, BTDF** is nonzero. It is described by the formula

$$\rho_{\mathrm{t}} = -\frac{DG(1-F_\lambda)}{4\langle \boldsymbol{l},\boldsymbol{n}\rangle\langle \boldsymbol{v},\boldsymbol{n}\rangle}\frac{\eta_2^2}{\eta_1^2}, \tag{1}$$

with the same functions D, G and F_λ. As the Fresnel factor F_λ describes, how many of photons are reflected by a microfacet, the factor $1 - F_\lambda$ describes the other photons, which pass the boundary of the two transparent media. The factors D and G describe the distribution of the normal vectors of microfacets and their geometric attenuation, but now the calculation of the normal vector $\boldsymbol{h}$ of a microfacet, given the vectors $\boldsymbol{l}$ and $\boldsymbol{v}$, is a bit more complicated.

A photon moving in the direction $\boldsymbol{l}$ after crossing the surface will move in the direction of $\boldsymbol{r}$.

Caution: here I have changed the orientation of $\boldsymbol{l}$, assumed the convention taken in the function `refract` of the standard GLSL library. This function computes the vector $\boldsymbol{r}$.



The angle between $\boldsymbol{l}$ and $\boldsymbol{n}$ is denoted by $\vartheta_{\boldsymbol{l}}$, and the angle of refraction is $\vartheta_{\boldsymbol{r}}$.

The angles ϑ_{bml} and $\vartheta_{\boldsymbol{r}}$ are related with the **Snell's law**:

$$\eta_1 \sin\vartheta_{\boldsymbol{l}} = \eta_2 \sin\vartheta_{\boldsymbol{r}},$$

where η_1 i η_2 are light refraction indices.

Denote $\eta = \eta_1/\eta_2$. Then

$$\sin\vartheta_{\boldsymbol{r}} = \eta\sin\vartheta_{\boldsymbol{l}},$$
$$\cos\vartheta_{\boldsymbol{r}} = \sqrt{1-\sin^2\vartheta_{\boldsymbol{r}}} = \sqrt{1-\eta^2\sin^2\vartheta_{\boldsymbol{l}}} = \sqrt{1-\eta^2(1-\cos^2\vartheta_{\boldsymbol{l}})}.$$

We have $\cos\vartheta_i = -\langle \boldsymbol{n},\boldsymbol{l}\rangle$. Let $\boldsymbol{t} = \frac{1}{\sin\vartheta_{\boldsymbol{l}}}(\boldsymbol{l} - \langle \boldsymbol{n},\boldsymbol{l}\rangle\boldsymbol{n})$. The vector $\boldsymbol{t}$ has unit length and it is tangent to the surface. Using it, we can obtain the unit vector

$$\begin{aligned}\boldsymbol{r} &= -\boldsymbol{n}\cos\vartheta_{\boldsymbol{r}} + \boldsymbol{t}\sin\vartheta_{\boldsymbol{r}} = \boldsymbol{n}\cos\vartheta_{\boldsymbol{r}} + \boldsymbol{t}\eta\sin\vartheta_{\boldsymbol{l}}\\ &= \boldsymbol{n}\sqrt{1-\eta^2(1-\langle \boldsymbol{n},\boldsymbol{l}\rangle^2)} + \eta(\boldsymbol{l} - \langle \boldsymbol{n},\boldsymbol{l}\rangle\boldsymbol{n})\\ &= \eta\boldsymbol{l} - \left(\sqrt{1-\eta^2(1-\langle \boldsymbol{n},\boldsymbol{l}\rangle^2)} + \eta\langle \boldsymbol{n},\boldsymbol{l}\rangle\right)\boldsymbol{n}.\end{aligned}$$

After denoting $k = 1 - \eta^2(1 - \langle \boldsymbol{n}, \boldsymbol{l} \rangle^2)$, we get

$$\boldsymbol{r} = \eta \boldsymbol{l} - \left(\sqrt{k} + \eta \langle \boldsymbol{n}, \boldsymbol{l} \rangle\right) \boldsymbol{n}.$$

If $k \geq 0$, then $k = \cos^2 \vartheta_{\boldsymbol{r}}$, but it is possible that $k < 0$—in this case the **complete internal reflection** occurs. It is possible when the light travels to the surface through the optically denser medium.

Now we replace the vectors $\boldsymbol{l}$, $\boldsymbol{r}$ and $\boldsymbol{n}$ by $-\boldsymbol{l}$, $\boldsymbol{v}$ and $\boldsymbol{h}$—the normal vector of the microfacet. From the formula

$$\boldsymbol{v} = -\eta \boldsymbol{l} - \left(\sqrt{k} - \eta \langle \boldsymbol{h}, \boldsymbol{l} \rangle\right) \boldsymbol{h}$$

we obtain the unit vector

$$\boldsymbol{h} = \frac{\pm 1}{\|\boldsymbol{v} + \eta \boldsymbol{l}\|} (\boldsymbol{v} + \eta \boldsymbol{l}).$$

The orientation should be chosen so as to obtain $\langle \boldsymbol{n}, \boldsymbol{h} \rangle > 0$. Having the vector $\boldsymbol{h}$, we can compute ρ_{t}.

The Cook and Torrance model, used to draw the teapot made of iron and copper, with a single point light source.

Above we can compare the effects of using the models of Lambert and Oren and Nayar, the teapot made of clay.

Integrating radiance in the Lambertian model

Radiance of light reflected by a surface of an opaque object is described with the formula

$$L_{\mathrm{r}}(\boldsymbol{p}) = \rho(\boldsymbol{p}) \int_{\boldsymbol{l} \in S_{\boldsymbol{n}+}} L(\boldsymbol{p} + \boldsymbol{l}, -\boldsymbol{l}) \langle \boldsymbol{l}, \boldsymbol{n} \rangle \, \mathrm{d}S = \rho(\boldsymbol{p}) I(\boldsymbol{p}, \boldsymbol{n}).$$

The function $L(\boldsymbol{p} + \boldsymbol{l}, -\boldsymbol{l})$ describes the radiance of light incoming from all directions to the point $\boldsymbol{p}$.

If the object is small compared to the distance of surrounding objects, then one can compute the irradiance for all unit vectors $\boldsymbol{n}$ and a single point $\boldsymbol{p}$. This irradiance may replace the "intensity" of light dispersed in the environment, used by the empirical lighting model of Lambert or Blinn–Phong.

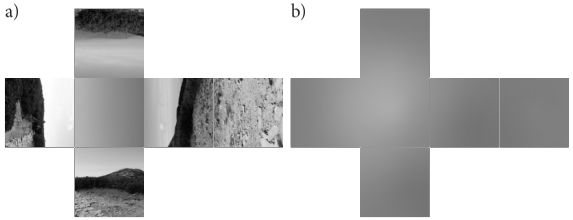
To represent functions whose domain is a sphere, OpenGL provides **cube textures**, which consist of six square arrays of texels corresponding to facets of the standard cube $[0,1]^3$. The argument of the GLSL `texture` function in this case is a nonzero vector $\boldsymbol{w} \in \mathbb{R}^3$. The function determines the facet of the cube intersected by the ray having direction of $\boldsymbol{w}$ (halfline $\{\, t\boldsymbol{w}\colon t > 0 \,\}$) and the intersection point of the ray with this facet and then it returns the result of interpolation of texels.

A cube texture representing the background of the object being drawn, which may be floor, walls and ceiling of a room or ground, buildings and sky, represents also the radiance of light from this background. One can use it to create a cube texture representing the irradiance—in each texel the value of the integral for the appropriate vector $\boldsymbol{n}$ will be stored.

This technique is known as the **image–based lighting**, or **IBL**.

A cube texture representing an image of surrounding objects may be obtained from photographs. For any vector $\boldsymbol{l} \neq \boldsymbol{0}$ its value represents the radiance of the light coming to a given point $\boldsymbol{p}$ from the direction of $\boldsymbol{l}$.

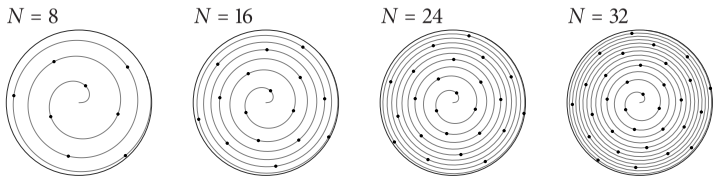
Based on this texture one has to construct a cube texture, whose value for any vector $\boldsymbol{n} \neq \boldsymbol{0}$ is the irradiance of the point $\boldsymbol{p}$ of a surface whose normal vector is $\boldsymbol{n}$. The irradiance may be computed for the vectors $\boldsymbol{n}$ corresponding to texels; the resolution of this texture needs not be high; $6 \times 64 \times 64$ is usually more than enough.



For a given vector $\boldsymbol{n}$ one has to integrate over the hemisphere $S_{\boldsymbol{n}+}$, which is the set of unit vectors $\boldsymbol{l}$ such that $\langle \boldsymbol{l}, \boldsymbol{n} \rangle \geq 0$. The integrand is the product $L(\boldsymbol{p} + \boldsymbol{l}, -\boldsymbol{l})\langle \boldsymbol{l}, \boldsymbol{n} \rangle$. One can use a quadrature obtained by evenly distributing N points in the unit circle.

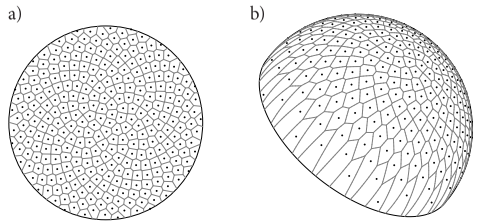
$$x_i = r_i \cos \varphi_i, \quad y_i = r_i \sin \varphi_i, \quad i = 0, \ldots, N-1, \tag{2}$$

where $r_i = \sqrt{(2i+1)/(2N)}$, $\varphi_i = (i + \frac{1}{2})2\pi\tau^2$, $\tau = (\sqrt{5}-1)/2$.



To obtain the vectors $\boldsymbol{l}_i$, one has to transform the vectors $\tilde{\boldsymbol{l}}_i = \left(x_i, y_i, \sqrt{1 - x_i^2 - y_i^2}\right)$ using an isometry, which maps the vector $\boldsymbol{e}_3 = (0,0,1)$ on $\boldsymbol{n}$.

An appropriate isometry is the Householder reflection, optionally followed by reversing the vector orientation.



The quadrature is defined by the formula

$$Q(L) = \frac{\pi}{N} \sum_{i=0}^{N-1} L(\boldsymbol{p} + \boldsymbol{l}_i, -\boldsymbol{l}_i).$$

The computation may be done by GPU—one has to draw facets of a cube on the facets of the cube texture. The shader program contains the fragment shader computing the integrals:

```
#version 450 core

#define PI   3.141592653
#define DPHI 2.399963229

in vec3 Normal;

layout(location=0) out vec4 out_Colour;

layout(binding=0) uniform samplerCube RadianceTxt;
uniform int N = 1500;

void main ( void )
{
  int   i;
  float phi, r, gamma;
  vec3  l, w, irrad;
```



```
  w = normalize ( Normal );
  w.z += Normal.z > 0.0 ?  1.0 :  -1.0;
  gamma = 2.0 / dot ( w, w );
  irrad = vec3(0.0);
  for ( i = 1, phi = 0.5*DPHI; i < 2*N; i += 2, phi += DPHI ) {
    r = sqrt ( float(i)/float(2*N) );
    l = vec3 ( r*cos ( phi ), r*sin ( phi ),
               sqrt ( float(2*N-i)/float(2*N) ) ); /* sqrt ( 1 - r*r ) */
    l -= w*(gamma*dot ( w, l ));
    if ( Normal.z > 0.0 ) l = -l;
    irrad += texture ( RadianceTxt, l ).rgb;
  }
  out_Colour = vec4 ( irrad*(PI/float(N)), 1.0 );
} /*main*/
```


Instead of the texture we can use a quadratic polynomial of the variables x, y, z, being the coordinates of $\boldsymbol{n}$, approximating the function I. This method, saving much space in GPU memory, was proposed by Ramamoorthi and Hanrahan.

The domain of I and its approximating polynomial p is the unit sphere S. The dimension of space of quadratic polynomials of three variables is 10, but the restriction of the domain to the sphere S reduces the dimension to 9, as the polynomial $x^2 + y^2 + z^2 - 1$ is equal to 0 on this sphere—the polynomials whose diference is divisible by $q(x, y, z) = x^2 + y^2 + z^2 - 1$ are identical.

In any set of polynomials having the same remainder of division by q, one can find a unique polynomial satisfying the **Laplace equation**, $\Delta p = 0$. Solutions of the Laplace equation are called **harmonic functions**.

A basis of the space V, whose elements are harmonic quadratic polunomials consists of the vectors

$$p_0 = 1, \quad p_1 = x, \quad p_2 = y, \quad p_3 = z, \quad p_4 = xy, \quad p_5 = yz, \quad p_6 = zx,$$
$$p_7 = x^2 - y^2, \quad p_8 = 2z^2 - x^2 - y^2.$$

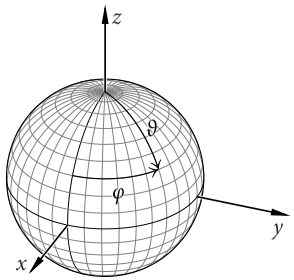
It is an orthogonal basis with respect to the scalar product

$$\langle f, g \rangle \stackrel{\text{def}}{=} \int_{\boldsymbol{w} \in S} f(\boldsymbol{w}) g(\boldsymbol{w}) \, \mathrm{d}S.$$

Using this basis one can find the polynomial p as the orthogonal projection of the function I on the space V:

$$p = \sum_{i=0}^{8} \frac{\langle p_i, I \rangle}{\|p_i\|^2} p_i.$$

It suffices to compute the coefficients $a_i = \langle p_i, I \rangle / \|p_i\|^2$.



The sphere S has the parametrisation

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}(\varphi, \vartheta) = \begin{bmatrix} \cos\varphi \sin\vartheta \\ \sin\varphi \sin\vartheta \\ \cos\vartheta \end{bmatrix}, \quad \varphi \in [0, 2\pi),\ \vartheta \in [0, \pi].$$

We need the integrals over S of the polynomials $x^i y^j z^k$ for $i + j + k \leq 4$. The measure $\mathrm{d}S$ of the piece of S corresponding to infinitesimal increments $\mathrm{d}\varphi$ and $\mathrm{d}\vartheta$ of the spherical coordinates is equal to $\sin\vartheta\,\mathrm{d}\vartheta\,\mathrm{d}\varphi$. Hence,

$$\int_S x^i y^j z^k \,\mathrm{d}S =$$
$$\int_{\varphi=0}^{2\pi}\int_{\vartheta=0}^{\pi} (\cos\varphi\sin\vartheta)^i (\sin\varphi\sin\vartheta)^j \cos^k\vartheta \sin\vartheta\,\mathrm{d}\vartheta\,\mathrm{d}\varphi = A_{ij}B_{k,i+j+1},$$

where $\quad A_{ij} = \int_0^{2\pi} \cos^i\varphi \sin^j\varphi\,\mathrm{d}\varphi, \quad B_{kl} = \int_0^{\pi} \cos^k\vartheta \sin^l\vartheta\,\mathrm{d}\vartheta.$

Some tedious calculations yield

$$A_{00} = 2\pi, \quad A_{20} = A_{02} = \pi, \quad A_{40} = A_{04} = \frac{3}{4}\pi, \quad A_{22} = \frac{\pi}{4},$$
$$B_{01} = 2, \quad B_{03} = \frac{4}{3}, \quad B_{05} = \frac{16}{15}, \quad B_{21} = \frac{2}{3}, \quad B_{23} = \frac{4}{15}, \quad B_{41} = \frac{2}{5}.$$

Using the above we can obtain

$$\|p_0\|^2 = 4\pi, \quad \|p_1\|^2 = \|p_2\|^2 = \|p_3\|^2 = \frac{4}{3}\pi, \quad \|p_4\|^2 = \|p_5\|^2 = \|p_6\|^2 = \frac{4}{15}\pi,$$
$$\|p_7\|^2 = \frac{16}{15}\pi, \quad \|p_8\|^2 = \frac{16}{5}\pi$$

and we chan check that the basis $\{p_0, \ldots, p_8\}$ is indeed orthogonal.

The quadrature used to find the polynomial p takes into account *all* texels of the irradiance texture. Each texel is a square on a facet of the cube $[-1, 1]^3$, whose edges have the length $a = \frac{1}{32}$. The central projection of such a square on the sphere S has the area $\approx a^2 \cos^3\alpha$, where α is the angle between the vector $\boldsymbol{w}$ corresponding to the central point of the texel and the normal vector of the cube facet.

We can use a compute shader, which makes the computation in three stages: computing the terms of the quadratures, summing (pairwise) these terms and dividing the sums by $\|p_i\|^2$ and storing the results in the destination place.

The first stage is done by the GLSL procedure shown below. The parameter `ti` describes the position of the shader thread in the (three-dimensional) workgroup. The parameter `size` specifies the resolution (in texels) of facets of the cube (irradiance) texture.

```
#version 450 core

layout(local_size_x=1) in;

layout(binding=0) uniform samplerCube IrradianceTxt;
layout(location=0) uniform int n;

layout(binding=0,std430) buffer Irrad { float a[27]; } irrad;
layout(binding=1,std430) buffer Aux { float t[]; } aux;

void StoreVec ( uint ind, vec3 v )
{ aux.t[ind] = v.r;  aux.t[ind+1] = v.g;  aux.t[ind+2] = v.b; }

void Integrate ( uvec3 ti, uint size )
{
 float a, b, s;
 vec3  w, ir;
 uint  ind;

 a = 2.0*((float(ti.x)+0.5)/float(size))-1.0;
 b = 2.0*((float(ti.y)+0.5)/float(size))-1.0;
 s = a*a + b*b + 1.0;
 s = 4.0/float(size*size) * inversesqrt ( s ) / s;
```



```
 switch ( ti.z ) {
case 0: w = vec3 (  1.0, a, b );  break;
case 1: w = vec3 ( -1.0, a, b );  break;
case 2: w = vec3 ( a,  1.0, b );  break;
case 3: w = vec3 ( a, -1.0, b );  break;
case 4: w = vec3 ( a, b,  1.0 );  break;
case 5: w = vec3 ( a, b, -1.0 );  break;
 }
 w = normalize ( w );
 ir = s*texture ( IrradianceTxt, w ).rgb;
 ind = 27*((ti.z*size + ti.y)*size + ti.x);
 StoreVec ( ind, ir );
 StoreVec ( ind+3, ir * w.x );
 StoreVec ( ind+6, ir * w.y );
 StoreVec ( ind+9, ir * w.z );
 StoreVec ( ind+12, ir * (w.x*w.y) );
 StoreVec ( ind+15, ir * (w.y*w.z) );
 StoreVec ( ind+18, ir * (w.z*w.x) );
 StoreVec ( ind+21, ir * ((w.x+w.y)*(w.x-w.y)) );
 StoreVec ( ind+24, ir * ((w.z+w.x)*(w.z-w.x) + (w.z+w.y)*(w.z-w.y)) );
} /*Integrate*/
```


The final result:

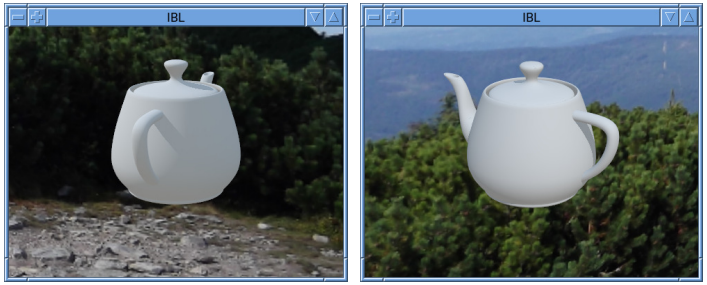


Image-based lighting for physical models

To improve realism of images of objects one has to take into account light coming from the environment. The complexity of integrals makes it impossible to evaluate them in a real-time animation. Therefore simplifications are necessary. To begin, we assume that the surface is opaque and isotropic, and it is not a light source.

After substituting the expression of BRDF to the Rendering Equation together with the irradiance of light coming from the direction of $\boldsymbol{l}$, we obtain the following expression

$$L_{\mathrm{rs}}(\boldsymbol{p},\boldsymbol{v}) = \int_{\boldsymbol{l}\in S_{n+}} \frac{DGF_{\lambda}}{4\langle \boldsymbol{l},\boldsymbol{n}\rangle\langle \boldsymbol{v},\boldsymbol{n}\rangle} L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})\langle \boldsymbol{l},\boldsymbol{n}\rangle \,\mathrm{d}S.$$

for the total radiance L_{rs} of light reflected in the specular way in the direction of $\boldsymbol{v}$.

To further simplify, we isolate two factors of the integrand, D and $L(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})$, and we replace them by the average value of their product. In this way we obtain the formula

$$L_{\mathrm{rs}}(\boldsymbol{p},\boldsymbol{v}) \approx \left(\frac{1}{2\pi}\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} DL(\boldsymbol{p}+\boldsymbol{l},-\boldsymbol{l})\,\mathrm{d}S\right)\left(\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{GF_\lambda}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S\right).$$

The factor $\frac{1}{2\pi}$ is the inverse of the measure of the integration set (i.e., the hemisphere $S_{\boldsymbol{n}+}$). The first integral, multiplied by this factor, will be denoted by L_{i}. It describes a filtered radiance of light coming from the environment.

We deal later with the second integral, M_λ.

The computation of the radiance L_{rs} will be done in two stages, preprocessing and final.

To reduce the volume of data obtained in preprocessing, we take two assumptions. Firstly, the factor D, depending on the vectors $\boldsymbol{n}$ and $\boldsymbol{l}$ (and on the surface roughness) will be uniquely defined by the angle $\vartheta_{\boldsymbol{h}}$ between $\boldsymbol{n}$ and $\boldsymbol{h}$, which implies isotropy of the surface. Secondly, we shall evaluate and store the integrals only for $\boldsymbol{v}=\boldsymbol{n}$, assuming that it is sufficient.

With this restriction, the angle $\vartheta_{\boldsymbol{h}}$ is a half of the angle $\vartheta_{\boldsymbol{l}}$ between $\boldsymbol{n}$ and $\boldsymbol{l}$, and the integral is a function of just the vector $\boldsymbol{n}$: $L_{\mathrm{i}}=L_{\mathrm{i}}(\boldsymbol{n})$.

The factor D may be taken according to Beckmann–Spizzichino or Trowbridge–Reitz; both dustributions have just one roughness parameter, m. Given a cube texture, which describes the radiance of the environment, we can evaluate the integral $L_{\mathrm{i}}(\boldsymbol{n})$ for all unit vectors $\boldsymbol{n}$ and for several chosen parameters m. The integrals may be stored in a (multilevel) cube texture. The higher value of m, the more "blurred" is the filtered radiance, and the smaller is the resolution of texture necessary to store it.

The filtered radiance L_{i} may be computed by a fragment shader being part of a program drawing six facets of the standard cube, on the cube texture. The shader uses the parametric representation of the hemisphere $S_{\boldsymbol{n}+}$ in a coordinate system such that $\boldsymbol{n}=\boldsymbol{e}_3=(0,0,1)$:

$$\begin{bmatrix} x\\ y\\ z\end{bmatrix}(\lambda,\vartheta)=\begin{bmatrix}\cos\lambda\sin\vartheta\\ \sin\lambda\sin\vartheta\\ \cos\vartheta\end{bmatrix},\quad \lambda\in[0,2\pi),\ \vartheta\in[0,\pi/2].$$

The integral of a function f over the hemisphere $S_{\boldsymbol{n}+}$ may be computed using the quadrature

$$\begin{aligned}\int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} f(\boldsymbol{l})\,\mathrm{d}S &= \int_0^{\pi/2}\left(\int_0^{2\pi} f\big(\boldsymbol{l}(\lambda,\vartheta)\big)\,\mathrm{d}\lambda\right)\sin\vartheta\,\mathrm{d}\vartheta\\ &\approx \frac{\pi^2}{NM}\sum_{j=1}^{M}\sum_{i=1}^{N} f\big(\boldsymbol{l}(\lambda_i,\vartheta_j)\big)\sin\vartheta_j,\\ &\text{where } \lambda_i=\pi\frac{2i-1}{N},\ \vartheta_j=\pi\frac{2j-1}{4M}.\end{aligned}$$

The rectangle $[0,2\pi)\times[0,\pi/2]$ has been divided to NM rectangles, whose central points became the quadrature knots. All coefficients of the quadrature are the same, $\pi^2/(NM)$. To take the factor $\frac{1}{2\pi}$ into account, the coefficients are replaced by $\pi/(2NM)$.

For a vector $\boldsymbol{n}$, processed by an instance of this shader, the Householder reflection is constructed and then the quadrature knots (λ_i,ϑ_j) are used to construct the vectors $\tilde{\boldsymbol{l}}=(\cos\lambda_i\sin\vartheta_j,\sin\lambda_i\sin\vartheta_j,\cos\vartheta_j)$, mapped to the vectors $\boldsymbol{l}$, and the radiance of incoming light is taken from the texture.

The texture with the image of the environment is accessed by the shader via the `RadianceTxt` sampler variable.

```
#version 450 core

#define PI 3.141592653
#define N 200
#define M 50

in vec3 Normal;

layout(location=0) out vec4 out_Colour;

layout(binding=0) uniform samplerCube RadianceTxt;

uniform float roughness2;

void main ( void )
{
  int i, j, k;
  float gamma, lambda, slambda, clambda, theta, stheta, ctheta,
        c2nh, s2nh, t2nh, D_BS;
  vec3 l, w, intrad, intradp;

  w = normalize ( Normal );
  w.z += Normal.z > 0.0 ?  1.0 :  -1.0;
```



```
  gamma = 2.0 / dot ( w, w );
  intrad = vec3 ( 0.0 );
  for ( j = 1; j < 2*M; j += 2 ) {
    intradp = vec3 ( 0.0 );
    theta = float(j)/float(4*M)*PI;
    stheta = sin ( theta ); ctheta = cos ( theta );
    c2nh = 0.5*(1.0+ctheta); s2nh = stheta*stheta/(4.0*c2nh);
    t2nh = s2nh/c2nh;
    D_BS = exp ( -t2nh/roughness2 ) / (PI*roughness2*c2nh*c2nh);
    for ( i = 1; i < 2*N; i += 2 ) {
      lambda = float(i)/float(N)*PI;
      slambda = sin ( lambda ); clambda = cos ( lambda );
      l = vec3 ( clambda*stheta, slambda*stheta, ctheta );
      l -= w*(gamma*dot ( w, l ));
      if ( Normal.z > 0.0 ) l = -l;
      intradp += texture ( RadianceTxt, l ).xyz * D_BS;
    }
    intrad += intradp*stheta;
  }
  out_Colour = vec4 ( intrad*(PI/(float(2*N*M))), 1.0 );
} /*main*/
```


The functions of the angle $\vartheta_{\boldsymbol{h}}$ (being a half of the angle between $\boldsymbol{l}$ and $\boldsymbol{n}$) are computed as follows:

$$\cos^2\vartheta_{\boldsymbol{h}} = \frac{1+\cos\vartheta_{\boldsymbol{l}}}{2}, \quad \sin^2\vartheta_{\boldsymbol{h}} = \frac{\sin^2\vartheta_{\boldsymbol{l}}}{4\cos^2\vartheta_{\boldsymbol{h}}}, \quad \tan^2\vartheta_{\boldsymbol{h}} = \frac{\sin^2\vartheta_{\boldsymbol{h}}}{\cos^2\vartheta_{\boldsymbol{h}}}.$$

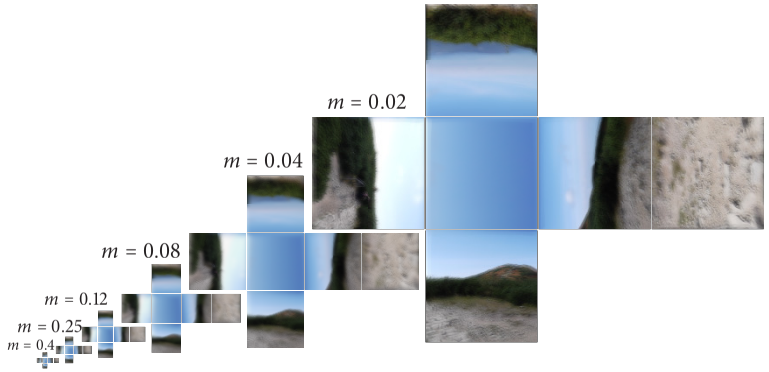
Then the factor D is computed according to the Beckmann–Spizzichino distribution. The value of the integral is passed to the output and stored in a texel of the result cube texture.

Before using the shader program shown above the result texture is created; it is a multilevel texture, whose levels are used to store the integrals for different roughness parameters m. Levels of this texture are attached to an off-screen framebuffer in a loop and then the shader program is executed.


```
#define IBLSPECTXTSIZE 128
#define IBLSPECTXTMIN 4

static const GLfloat roughness[6] = {0.02, 0.04, 0.08, 0.12, 0.25, 0.4};

GLuint ConstructIBLSpecTexture ( GLuint prog_id, GLuint r2_loc,
                                 GLuint radtxt )
{
  GLuint fbo, status, itxt;
  int i, wh;

  glActiveTexture ( GL_TEXTURE0 );
  itxt = CreateCubeTexture ( IBLSPECTXTSIZE, GL_RGB32F, 5 );
  glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri (GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glGenFramebuffers ( 1, &fbo );
  glBindFramebuffer ( GL_FRAMEBUFFER, fbo );
  glDrawBuffer ( GL_COLOR_ATTACHMENT0 );
  glBindTexture ( GL_TEXTURE_CUBE_MAP, radtxt );
  glBindVertexArray ( empty_vao );
  glUseProgram ( prog_id );
  for (i = 0, wh = IBLSPECTXTSIZE; wh >= IBLSPECTXTMIN; i++, wh /= 2) {
    glFramebufferTexture (GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, itxt, i);
```

```
    if ( (status = glCheckFramebufferStatus ( GL_FRAMEBUFFER)) !=
         GL_FRAMEBUFFER_COMPLETE )
      ExitOnError ( "ConstructIBLSpecTexture");
    glViewport ( 0, 0, wh, wh );
    glUniform1f ( r2_loc, roughness[i]*roughness[i] );
    glDrawArrays ( GL_POINTS, 0, 1 );
    glFlush ();
  }
  glUseProgram ( 0 );
  glBindVertexArray ( 0 );
  glBindFramebuffer ( GL_FRAMEBUFFER, 0 );
  glDeleteFramebuffers ( 1, &fbo );
  return itxt;
} /*ConstructIBLSpecTexture*/
```

The resolution of the texture at level 0 is 128 × 128 and at level 5 it is 4 × 4.



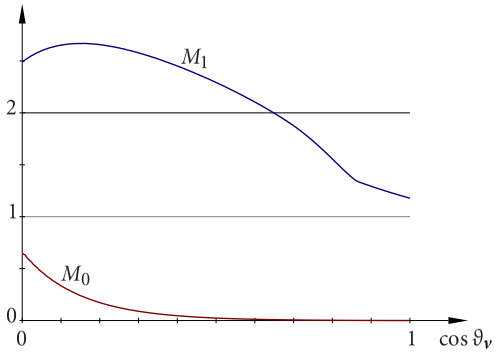
The second integral, M_λ, will also be simplified, by replacing the Fresnel factor F_λ with the Schlick approximation. Then,

$$M_\lambda \approx \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{G\big(F_{\lambda,0} + (1-F_{\lambda,0})(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5\big)}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S = M_0(\vartheta_{\boldsymbol{v}}) + F_{\lambda,0}M_1(\vartheta_{\boldsymbol{v}}),$$

where
$$M_0(\vartheta_{\boldsymbol{v}}) = \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{G(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S,$$

$$M_1(\vartheta_{\boldsymbol{v}}) = \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} \frac{G\big(1-(1-\langle \boldsymbol{l},\boldsymbol{h}\rangle)^5\big)}{4\langle \boldsymbol{v},\boldsymbol{n}\rangle}\,\mathrm{d}S.$$

It is then possible to compute the integrals M_0 and M_1, being functions of just one parameter, the angle $\vartheta_{\boldsymbol{v}}$ between $\boldsymbol{v}$ and $\boldsymbol{n}$, in preprocessing. GPU may be used to compute the integrals numerically and store them in a one-dimensional texture.



With the preprocessing done, we can draw final images. We have to enlight a surface fragment with the normal vector $\boldsymbol{n}$, seen from the direction $\boldsymbol{v}$. Had the surface been a perfect mirror, it would reflect towards the viewer light coming from the direction of $\boldsymbol{r} = 2\langle \boldsymbol{v}, \boldsymbol{n} \rangle \boldsymbol{n} - \boldsymbol{v}$. The image of environment reflected in our surface, a non-perfect mirror, is blurred—this is the contents of the first texture obtained in preprocessing. Though the function stored in this texture was evaluated as a function of $\boldsymbol{n}$, now its argument has to be $\boldsymbol{r}$.

Only a part of the fragment shader computing the image-based illumination for the simplified Cook and Torrance model is shown on the next two slides. For the diffuse reflection The Lambertian model may be used, with the irradiance texture obtained as described before.


```
layout(binding=4,std430) buffer Irrad { float a[27]; } irrad;

layout(binding=10) uniform samplerCube SpecRadianceTxt;
layout(binding=11) uniform sampler1D m0m1Txt;

vec3 SpecularEnvLighting ( vec3 normal, vec3 vv, float cthetav )
{
  vec3 Li, r, Fl;
  vec2 MOM1;
  int i, j, k;

  r = -reflect ( vv, normal );
  Li = texture ( SpecRadianceTxt, r ).rgb;
  MOM1 = texture ( m0m1Txt, cthetav ).xy;
  Fl = MOM1.xxx + MOM1.y*mm.Fl0;
  return Li*Fl;
} /*SpecularEnvLighting*/
```



```
vec3 PBRLighting ( void )
{
  vec3 lv, vv, Colour, rhoD, rhoS, lp;
  float dist, m2, cnv, cnl, s;
  uint i, mask;

  vv = posDifference ( trb.eyepos, In.Position, dist );
  if ( dot ( vv, tnormal ) < 0.0 ) normal = -normal;
  cnv = dot ( vv, normal );
  Colour = mm.kD > 0.0 ?
           mm.kD*mm.cl*LightPoly ( normal ) :  vec3 ( 0.0 );
  if ( mm.kS > 0.0 )
    Colour = mm.kS*SpecularEnvLighting ( normal, vv, cnv );
  m2 = mm.m*mm.m;
  for ( i = 0, mask = 0x00000001; i < light.nls; i++, mask <<= 1 )
    ....  /* adding light from point light sources */
  return clamp ( Colour, 0.0, 1.0 );
} /*PBRLighting*/
```


Before drawing one has to execute the following instructions, that bind the textures:

```
glActiveTexture ( GL_TEXTURE0 + 11 );
glBindTexture ( GL_TEXTURE_1D, fltxt );
glActiveTexture ( GL_TEXTURE0 + 10 );
glBindTexture ( GL_TEXTURE_CUBE_MAP, itxt );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_BASE_LEVEL, ... );
glBindBufferBase ( GL_SHADER_STORAGE_BUFFER, 4, irrpoly );
ChooseMaterial ( ... );
```

Unfortunately, OpenGL gives access to only one level of a cube texture. We choose the level according to the object to be drawn.

Here is the result: an iron teapot and a copper teapot on holidays.

Global lighting models

In 1986 James T. Kajiya published what he named *The Rendering Equation*.
It expresses the radiance of light leaving any point $\boldsymbol{p}$ of a surface, viewed from any direction $\boldsymbol{v}$, as the sum of radiance of light emitted by the surface (if there is any) and the light reflected by it:

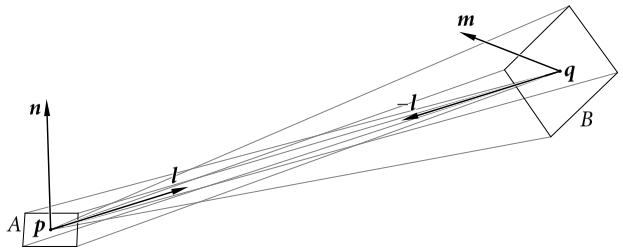
$$L(\boldsymbol{p};\boldsymbol{v}) = L_e(\boldsymbol{p};\boldsymbol{v}) + L_r(\boldsymbol{p};\boldsymbol{v}).$$

Radiance of light reflected in the direction of $\boldsymbol{v}$ is

$$L_r(\boldsymbol{p};\boldsymbol{v}) = \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) I(\boldsymbol{p};\boldsymbol{l})\,\mathrm{d}S.$$

The integral is taken over the unit sphere S.

Let's find the irradiance of the point $\boldsymbol{p}$ of a surface element A, by the light coming from an element B, located at the distance r from A.



Assume that the diameters of the elements are small, as compared to r. Then the solid angles of the element A seen from all points of B are (almost) the same and the solid angles of the element B seen from all points of A are the same. Also, each pair of points ($\boldsymbol{p} \in A, \boldsymbol{q} \in B$) is at (almost) the same distance r.

Radiance is a function piecewise continuous. Assume that it is (almost) the same at all points of B in all directions close enough to the vector $-\boldsymbol{l} = \boldsymbol{p} - \boldsymbol{q}$.

Then, if the radiance of light leaving the point $\boldsymbol{q} \in B$ in the direction of $-\boldsymbol{l}$ is $L(\boldsymbol{q};-\boldsymbol{l})$, then the flux of light coming from B to the element A from B is equal to $L(\boldsymbol{q};-\boldsymbol{l})B_s\psi_{AB}$, where B_s is the measure of shortened element B seen from the point $\boldsymbol{p} \in A$, and ψ_{AB} is the solid angle of the element A seen from the point $\boldsymbol{q} \in B$.

The iradiance of the element A may be obtained by dividing the flux by the area A. We can find the measure of the solid angle $\psi_{AB} = A|\cos\measuredangle(\boldsymbol{n},\boldsymbol{l})|/r^2$ and the measure of the shortened element B, $B_s = r^2\psi_{BA}$. Then we obtain

$$\begin{aligned} I(p;\boldsymbol{l}) &= L(\boldsymbol{q};-\boldsymbol{l})r^2\psi_{BA}A|\cos\measuredangle(\boldsymbol{n},\boldsymbol{l})|/r^2/A \\ &= L(\boldsymbol{q};-\boldsymbol{l})\psi_{BA}|\cos\measuredangle(\boldsymbol{n},\boldsymbol{l})|. \end{aligned}$$

The solid angle ψ_{BA} corresponds to the piece $\mathrm{d}S$ of the unit sphere, being the set of integration. The integral, i.e., the total radiance of light reflected by the element A is

$$L_r(\boldsymbol{p};\boldsymbol{v}) = \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) L\big(\boldsymbol{q}(\boldsymbol{p},\boldsymbol{l}),-\boldsymbol{l}\big)|\cos\measuredangle(\boldsymbol{n},\boldsymbol{l})|\,\mathrm{d}S.$$

In the formula above the value $\boldsymbol{q}(\boldsymbol{p},\boldsymbol{l})$ of the function $\boldsymbol{q}$ is the point of a surface seen from the point $\boldsymbol{p}$ in the direction of the vector $\boldsymbol{l}$. It may be found as the closest to $\boldsymbol{p}$ intersection of the halfline $\{\,\boldsymbol{p} + t\boldsymbol{l} : t > 0\,\}$ with an object of our scene.

The general equation of energetic balance, assuming the perfect homogeneity and transparency of the medium (e.g. the air) is as follows:

$$L(\boldsymbol{p};\boldsymbol{v}) = L_e(\boldsymbol{p};\boldsymbol{v}) + \int_{\boldsymbol{l}\in S} \rho(\boldsymbol{p};\boldsymbol{l},\boldsymbol{v}) L\big(\boldsymbol{q}(\boldsymbol{p},\boldsymbol{l}), -\boldsymbol{l}\big) |\cos \measuredangle(\boldsymbol{n},\boldsymbol{l})| \,\mathrm{d}S.$$

Such equations may be solved numerically, after a discretisation. The difficulty is caused by the dimension of the domain of the function L, which is 4—the domain is the Cartesian product of the union of all surfaces of the scene with the unit sphere S.

A radical simplification may be done by assuming that all surfaces are Lambertian, i.e., perfectly mat and opaque and that the function L_e takes just two values in two hemispheres separated by the tangent plane of the shining surface at any point. Assume in addition that all surfaces are surfaces of opaque solids, i.e., the light may be reflected only at one side of each surface. Then the functions L, L_e and ρ depend only on the point $\boldsymbol{p}$.

With these assumptions we have the following integral equation:

$$L(\boldsymbol{p}) = L_e(\boldsymbol{p}) + \rho(\boldsymbol{p}) \int_{\boldsymbol{l}\in S_{\boldsymbol{n}+}} L\big(\boldsymbol{q}(\boldsymbol{p},\boldsymbol{l})\big) \cos \measuredangle(\boldsymbol{n},\boldsymbol{l}) \,\mathrm{d}S.$$

The unknown function has a two-dimensional domain. The set of integration is the hemisphere $S_{\boldsymbol{n}+}$ made of the unit vectors $\boldsymbol{l}$ such that $\cos \measuredangle(\boldsymbol{n},\boldsymbol{l}) \geq 0$.

The integral over a hemisphere may be changed to an integral over the union of all surfaces of the scene, which is denoted below with the letter B. Let $\boldsymbol{q} = \boldsymbol{q}(\boldsymbol{p},\boldsymbol{l})$ be a point of a surface element $\mathrm{d}B$, whose unit normal vector is $\boldsymbol{m}$. This element seen from the point $\boldsymbol{p}$ takes the solid angle

$$\mathrm{d}S = \frac{|\cos \measuredangle(\boldsymbol{m}, \boldsymbol{p}-\boldsymbol{q})|}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} \mathrm{d}B,$$

but it is not necessarily visible from $\boldsymbol{p}$. Therefore another factor must be introduced, $v(\boldsymbol{p},\boldsymbol{q})$, equal to 1 if the points $\boldsymbol{p}$ and $\boldsymbol{q}$, located at the distance $r = \|\boldsymbol{q}-\boldsymbol{p}\|$, "see each other", or 0 otherwise. Thus we obtain the equation

$$L(\boldsymbol{p}) = L_e(\boldsymbol{p}) + \rho(\boldsymbol{p}) \int_{\boldsymbol{q}\in B} v(\boldsymbol{p},\boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n},\boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{m},\boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{p}-\boldsymbol{q}\|^2} L(\boldsymbol{q}) \,\mathrm{d}B.$$

Assuming that the orientation of normal vectors indicate the "external" side of each surface (i.e., the one enlighted and possible to be seen), the cosines will be nonnegative and the symbols of absolute value may be dropped.

Such an equation is a special case of a so-called **Fredholm integral equation of the second kind**. Before any attempt to solve it, one should check, whether a solution exists. The surfaces are piecewise smooth (though they may be curved), but they can make dihedral angles. We assume that the functions L_e and ρ are piecewise continuous, i.e., there may exist a finite number of smooth curves made of points of discontinuity of these functions.

The integral operator

$$\mathcal{K}(L)(\boldsymbol{p}) = \rho(\boldsymbol{p}) \int_{\boldsymbol{q}\in B} v(\boldsymbol{p},\boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n},\boldsymbol{q}-\boldsymbol{p}) \cos \measuredangle(\boldsymbol{m},\boldsymbol{p}-\boldsymbol{q})}{\|\boldsymbol{q}-\boldsymbol{p}\|^2} L(\boldsymbol{q})\,\mathrm{d}B$$

is a "mollifier" of the function L, i.e., the function being its value cannot be discontinuous apart from the (fixed) places of discontinuity. Therefore one can seek solutions in the space of piecewise continuous functions with the domain B. This space, equipped with the supremum norm, is a Banach space, i.e., it is a closed metric space.

Remark: The set of points of discontinuity must be fixed, as the set of all piecewise discontinuous functions with the supremum norm is not closed.

The energetic balance may be rewritten in the following form:

$$L = L_e + \mathcal{K}L, \quad \text{i.e.,} \quad (\mathcal{I}-\mathcal{K})L = L_e.$$

The symbol $\mathcal{I}$ denotes the identity operator.

Then we can write

$$\begin{aligned} L = (\mathcal{I}-\mathcal{K})^{-1}L_e &= (\mathcal{I}+\mathcal{K}+\mathcal{K}^2+\mathcal{K}^3+\cdots)L_e \\ &= L_e + \mathcal{K}L_e + \mathcal{K}^2L_e + \mathcal{K}^3L_e + \cdots, \end{aligned}$$

but the function L exists if the above **Neumann series** is convergent.

The Neumann series has the following interpretation: its first term, L_e, describes the emitted light. The term $\mathcal{K}L_e$ describes the light which has been reflected once, $\mathcal{K}^2L_e$ corresponds to the light after two reflections and in general $\mathcal{K}^nL_e$ decribes the light which after emission has been reflected n times. One can prove that the supremum norm of the operator $\mathcal{K}$ is not greater than $\pi\|\rho\|_\infty$. Thus, if the maximal value of ρ is less than $1/\pi$, then the Neumann series is convergent. The terms $\mathcal{K}^nL_e$ with a large n have a negligible contribution to the final effect and therefore they may be dropped.

The simplest **discretisation** of the energetic balance equation may be done by dividing the surfaces into small enough **finite elements**, triangular or quadrangular (whose diameter is at most h). Then we can assume that the given function L_e and the approximation of the solution L_h, which we need, are constant in these elements. Thus, having n elements, we need to find n values of the function L_h.

An example of numerical methods of discretisation is the so-called **collocation method**: we choose one point of each element (e.g., its centre of gravity) and we assume that the integral equation to be solved has to be satisfied at the chosen points. Another possibility is the **Galerkin method**, which assumes a zero average residuum in each eleemnt. As the operator $\mathcal{K}$ is linear, both methods lead to a system of n linear equations with n unknown variables.

The union B of all surfaces will be divided into the elements $A_1,\ldots,A_n$ having at most common edges: there is $B = \bigcup_{j=1}^n A_j$. We assume that each element A_j is planar. Its normal vector will be denoted by $\boldsymbol{n}_j$.

Let

$$L_h(\boldsymbol{p}) = \sum_{j=1}^n L_j\phi_j(\boldsymbol{p}), \qquad L_{eh}(\boldsymbol{p}) = \sum_{j=1}^n L_{ej}\phi_j(\boldsymbol{p}),$$

where $\phi_j(\boldsymbol{p})$ is equal to 1 if $\boldsymbol{p}\in A_j$ and it is 0 if $\boldsymbol{p}$ is a point of any other element (the values of the functions ϕ_j at common edges of elements are averaged). The function L_h is the radiance to be found. The function L_{eh} replaced the given radiance L_e of the emitted light.

At a point $\boldsymbol{p} \in A_i$ we have

$$L_h(\boldsymbol{p}) = L_e(\boldsymbol{p}) + \rho(\boldsymbol{p}) \sum_{j=1}^{n} \int_{\boldsymbol{q} \in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q} - \boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p} - \boldsymbol{q})}{\|\boldsymbol{q} - \boldsymbol{p}\|^2} L_j \, \mathrm{d}B.$$

By substituting $\boldsymbol{p} = \boldsymbol{p}_i$ (where $\boldsymbol{p}_i$ is the point chosen in the element A_i) for $i = 1, \ldots, n$, we implement the collocation method. Its result is the system of linear equations

$$L_i = L_{ei} + \rho(\boldsymbol{p}_i) \sum_{j=1}^{m} G_{ij} L_j.$$

To average the residuum in an element A_i (in the Galerkin method), we integrate the sides of the equation over the element A_i and we divide them by the area of this element:

$$L_i = \frac{1}{A_i} \int_{\boldsymbol{p} \in A_i} L_e(\boldsymbol{p}) + \rho(\boldsymbol{p}) \sum_{j=1}^{n} \left(\int_{\boldsymbol{q} \in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q} - \boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p} - \boldsymbol{q})}{\|\boldsymbol{p} - \boldsymbol{q}\|^2} \, \mathrm{d}B \right) L_j \, \mathrm{d}B.$$

After replacing the functions ρ and L_e by their average values in the element A_i:

$$\rho_i = \frac{1}{A_i} \int_{\boldsymbol{p} \in A_i} \rho(\boldsymbol{p}) \, \mathrm{d}B, \quad L_{ei} = \frac{1}{A_i} \int_{\boldsymbol{p} \in A_i} L_e(\boldsymbol{p}) \, \mathrm{d}B,$$

we obtain a (slightly different) equation

$$\begin{aligned} L_i &= L_{ei} + \\ &\quad \rho_i \sum_{j=1}^{n} \left(\frac{1}{A_i} \int_{\boldsymbol{p} \in A_i} \int_{\boldsymbol{q} \in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q} - \boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p} - \boldsymbol{q})}{\|\boldsymbol{p} - \boldsymbol{q}\|^2} \, \mathrm{d}B \, \mathrm{d}B \right) L_j \\ &= L_{ei} + \rho_i \sum_{j=1}^{n} F_{ij} L_j. \end{aligned}$$

The numbers

$$F_{ij} = \frac{1}{A_i} \int_{\boldsymbol{p} \in A_i} \int_{\boldsymbol{q} \in A_j} v(\boldsymbol{p}, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q} - \boldsymbol{p}) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p} - \boldsymbol{q})}{\|\boldsymbol{p} - \boldsymbol{q}\|^2} \, \mathrm{d}B \, \mathrm{d}B$$

are called the **form factors** or **optical influence factors**. They depend only on the shape, size and mutual position in space of the elements A_i and A_j and the possible presence or absence of obstacles between them.

We can notice that the form factors satisfy the equalities $A_i F_{ij} = A_j F_{ji}$, moreover, $F_{ii} = 0$ for all i. In addition $F_{ij} = 0$ if the elements A_i and A_j are coplanar or they "do not see each other".

Before solving the system of equations we must find the form factors F_{ij}. In practice they are often replaced by the easier to find numbers G_{ij} obtained from the collocation method:

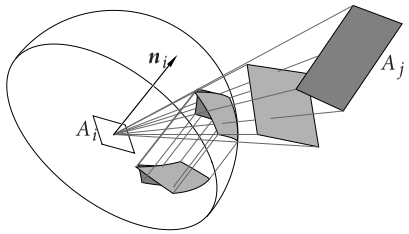
$$G_{ij} = \int_{\boldsymbol{q}\in A_j} v(\boldsymbol{p}_i, \boldsymbol{q}) \frac{\cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{q} - \boldsymbol{p}_i) \cos \measuredangle(\boldsymbol{n}_j, \boldsymbol{p}_i - \boldsymbol{q})}{\|\boldsymbol{q} - \boldsymbol{p}_i\|^2} \, \mathrm{d}B.$$

Consider two elements, A_i and A_j, such that from the point $\boldsymbol{p}_i$ of the element A_i we can see the entire element A_j. Then the coefficient G_{ij} is equal to

$$G_{ij} = \int_{\boldsymbol{l}\in S_j} \cos \measuredangle(\boldsymbol{n}_i, \boldsymbol{l}) \, \mathrm{d}S,$$

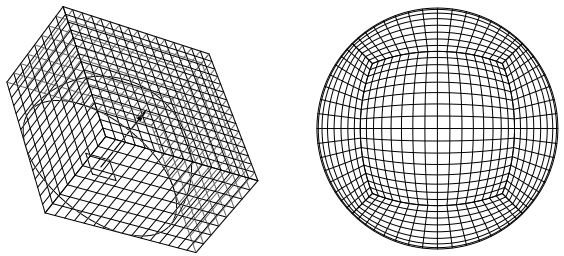
where S_j denotes the piece of the hemisphere over the element A_i, taken by the image of A_j.

The latter integral is the measure of the area obtained by double projection of A_j; first on the hemisphere $S_{\boldsymbol{n}+}$ and then on the plane of the element A_i. If only a part of the element A_j is visible, then only the projection of its visible part must be measured. It is the so-called **Nusselt analogy**, shown on the picture.



All coefficients G_{ij} are nonnegative and for all i they satisfy the inequality $\sum_{j=1}^{n} G_{ij} \leq \pi$, which is sharp if a part of the unit circle is not covered by the image of any element. In addition $G_{ii} = 0$ for all i, but there may occur $A_i G_{ij} \neq A_j G_{ji}$.

The coefficients G_{ij} may be computed as follows: for all i we draw our scene in five perspective projections, whose centre is the point $\boldsymbol{p}_i$. The projection planes are facets of a rectangular hemicube. The values stored in the pixels are numbers (identifiers) of the elements. By the Nusselt analogy, each pixel has its weight, equal to the area taken by its image under the double projection. Assuming that the entire pixel belongs to the image of one element, we can sum these weights to obtain (approximate) values of G_{ij}.



The accuracy of the approximation strongly depends on the resolution of the images. The resolutions taken in practice are not too high because of the computational cost. The entire scene is to be drawn $5n$ times; even with various acceleration techniques (eliminating all objects invisible from the points $\boldsymbol{p}_i$) this is the most expensive part of the computation. Many elements "do not see each other" and their coefficients G_{ij} are zero.

We have the system of equations

$$(I - DF)\underline{L}_h = \underline{L}_e,$$

with the identity matrix I, the diagonal matrix D whose diagonal coefficients are $d_{ii} = \rho(\boldsymbol{p}_i)$, the matrix F with the coefficients G_{ij}, the given vector $\underline{L}_e$, whose coordinates are the values of the function L_e at $\boldsymbol{p}_i$ or average values of this function in the elements A_i and the unknown vector $\underline{L}$, whose coordinates are the numbers $L_j = L_h(\boldsymbol{p}_j)$.

The number n of equations and unknown variables is usually between 10^3 and 10^5. but the matrix F is sparse, i.e., with many zero coefficients (depending on the scene usually several to umpteen percent of coefficients are nonzero).

The coordinates of L_e and diagonal coefficients of the matrix D are functions of the wavelength, which in practice are represented by vectors with three components, R, G and B. The matrix F, determined by geometry of the scene, is the same for all wavelengths. The sum of (nonnegative) coefficients in each row is less than or equal to π and all coefficients of D ought to be less than $1/\pi$. The maximum norm of the resulting matrix $K = DF$ is less than 1. Hence, the Neumann series for the discretised energetic balance equation,

$$\underline{L} = \underline{L}_e + K\underline{L}_e + K^2\underline{L}_e + K^3\underline{L}_e + \cdots$$

is convergent and its truncation after sufficiently many terms is a satisfactory approximation of the solution.

Consecutive partial sums will be obtained by taking $\underline{L}_0 = \underline{L}_e$ and then

$$\underline{L}_k = \underline{L}_e + K\underline{L}_{k-1}, \quad k = 1, 2, 3, \ldots,$$

and such a sequence converges to the solution $\underline{L}_h$ regardless of the choice of the vector $\underline{L}_0$. Hence, we can choose it as the best approximation of the solution that we have in hand.

Having the approximate solution of the system of equations, we can create a texture to be put on the objects of the scene on its final image. However, the discontinuities of radiance at the boundaries of elements (being consequences of assumption that the radiance is constant in each element) are clearly visible, which spoils the effect. The simplest method of fixing it is to compute, for each common vertex of coplanar elements the average of radiance in these elements and then to interpolate these averages among vertices of each element. If the function ρ in each element is constant, then the effect should be good enough.

If the function L_e is the sum of a function L_a, which describes the light emitted by the elements and a function L_b, being the product of ρ and the irradiance of light coming from point light sources, the problem is more complicated. Let $\tilde{L}$ denote the function obtained using the vector $K\underline{L}_h$ by averaging the values in the element surounding vertices. The colour of a point $\boldsymbol{p} \in A_i$ mapped to a pixel of the final image should then be taken as the value of the expression

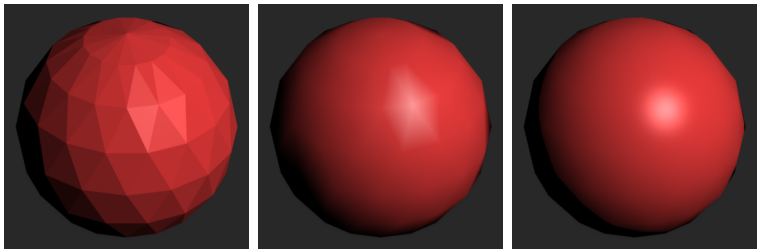
$$L_a(\boldsymbol{p}) + \big(L_b(\boldsymbol{p}) + \tilde{L}(\boldsymbol{p})\big)\rho(\boldsymbol{p})/\rho_i.$$

In this case we obtain a satisfactory effect when the function ρ is a texture and when the sharp boundaries of shadows produced by the point light sources do not match the boundaries of elements.

Remark: In literature on the energetic balance method, for scenes made of Lambertian surfaces, instead of the function ρ, taking values from the interval $[0, 1/\pi)$ functions π times greater are considered—their values describe the fraction of energy of light which is reflected (and not changed into heat). These functions are easier to design. In the formulae used to define the form factors F_{ij} and G_{ij} an additional factor $1/\pi$ is introduced. Also, the given function L_e and the unknown function L denote the **emitance** of the light, called also **radiosity**, which is π times greater than the radiance.

Shading methods

Colours of pixels of a triangle may be computed based on colours of the vertices, which is called **shading**. Various methods of shading give us different images.

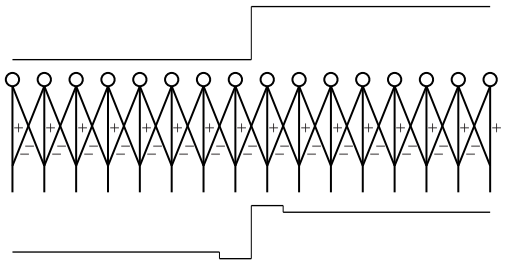


The simplest method fills the entire triangle with the colour of one point—this is **flat shading**. The effect of approximating a smooth surface with planar triangles is not very good.

A more sophisticated method called **Gouraud shading** uses linear (or rather affine) interpolation of colours among the vertices. It is cheap and it works well for mat surfaces, but not well enough for glossy surfaces, as highlight areas are polygonal. Also, some highlights may be missed.

The third method is the **Phong shading**; here, for each vertex we specify a normal vector (of the smooth surface approximated by the triangles), which is interpolated and substituted to the lighting model chosen for the object.

The reason, why common boundaries of areas with different colours are visible well is the structure of retina in human eyes, causing the so-called **lateral inhibition**.



The light receptor stimulated by light sends a signal along its nerve, which is weakened by signals from neighbouring receptors. The result is the increased contrast between the areas enlighted with different intensity. This is called the **Mach band effect**.

The image of a surface depends on tiny details of its shape. They may be rendered by modifying the normal vector and substituting it to the lighting model. It may be done as the **displacement texture** and the technique is caleld **bump mapping**.

Suppose that we have a parametric surface with a regular parametrisation $\boldsymbol{p}\colon A \to \mathbb{R}^3$ of class C^1. The unit normal vector at the point $\boldsymbol{p}(u,v)$ is the value of the **Gauss map**,

$$\boldsymbol{n}(u,v) = \frac{\boldsymbol{m}(u,v)}{\|\boldsymbol{m}(u,v)\|}, \quad \boldsymbol{m}(u,v) = \boldsymbol{p}_u(u,v) \wedge \boldsymbol{p}_v(u,v).$$

To increase flexibility of the construction, we can introduce another mapping $\boldsymbol{q}\colon A \to B \subset \mathbb{R}^2$. The function $d\colon B \to \mathbb{R}$ describes displacements of points of the surface. Using it we define a parametrisation of the new surface

$$\hat{\boldsymbol{p}}(u,v) = \boldsymbol{p}(u,v) + d\big(\boldsymbol{q}(u,v)\big)\boldsymbol{n}(u,v).$$

It is assumed that maximal absolute values of d are small. Using the formula $\hat{\boldsymbol{p}} = \boldsymbol{p} + (d \circ \boldsymbol{q})\boldsymbol{n}$ we can find the normal vector of the new surface. The partial derivatives of the parametrisation are columns of the matrix

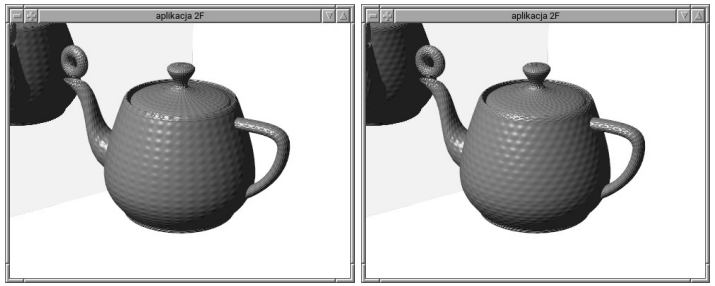
$$\mathrm{D}\hat{\boldsymbol{p}} = \mathrm{D}\boldsymbol{p} + \mathrm{D}(d \circ \boldsymbol{q}) \cdot \boldsymbol{n} + (d \circ \boldsymbol{q})\mathrm{D}\boldsymbol{n} \approx \mathrm{D}\boldsymbol{p} + (\mathrm{D}d \cdot \mathrm{D}\boldsymbol{q})\boldsymbol{n}.$$

Due to the small values of d we can neglect the term $(d \circ \boldsymbol{q})\mathrm{D}\boldsymbol{n}$, whose computation would involve second order derivatives of the parametrisation $\boldsymbol{p}$. The final formula needs just the 1×2 matrix $\mathrm{D}d$ (gradient of the function d) and the 2×2 matrix $\mathrm{D}\boldsymbol{q}$. If $\boldsymbol{q}$ is an affine mapping, the matrix $\mathrm{D}\boldsymbol{q}$ describes its linear part.

If the function d is given by a formula, we need to differentiate it. The function d may also be represented by an array of texels and then we can use divided differences to approximate the derivatives.

In an OpenGL application the fragment shader ought to obtain the parameters (u,v) of the point of the surface being processed and the partial derivatives of the parametrisation $\boldsymbol{p}$. The representation of the functions d and $\boldsymbol{q}$ may be given in a texture or in uniform variables. After computing the matrix $\mathrm{D}\hat{\boldsymbol{p}}$ we need to compute and normalise the cross product of its columns to obtain the unit normal vector $\hat{\boldsymbol{n}}$ of the deformed surface.

We substitute this vector to the lighting model, but there is an additional problem here: the checking, which side of the surface is visible, should be done using the vector $\boldsymbol{n}$, whose direction is different from that of $\hat{\boldsymbol{n}}$, ale jest tu. If the scalar products $\langle \boldsymbol{v}, \boldsymbol{n} \rangle$ and $\langle \boldsymbol{v}, \hat{\boldsymbol{n}} \rangle$ have different signs, then lighting computation should be done with the vector $\boldsymbol{n}$.



If the values of d cause the deformation of the surface has the magnitude of one pixel or more (on a close-up), then additional corrections of the surface parameters may be necessary. One is the modification of **depth** (for visibility tests) and the other takes the **parallax** into account.

Consider patches with rational parametrisations $\boldsymbol{p}(u,v)$ and $\hat{\boldsymbol{p}}(u,v)$. Let C denote the 4×4 matrix of transformation to the standard cube coordinate system. For perspective projections this is not an affine transformation.

The depth modification is done by assigning a new value to the `gl_FragDepth` built-in variable of the fragment shader. The correction is equal to dZ/W, where Z and W are the last two coordinates of the vector $C\boldsymbol{P}$, where $\boldsymbol{P}$ is the vector of homogeneous coordinates of the point $\boldsymbol{p}$ processed as a fragment. If this correction is the value of a variable `dz`, then we need to execute the instruction

```
gl_FragDepth = gl_FragCoord.z + dz;
```


Let $\boldsymbol{Q}$, $\overline{\boldsymbol{Q}}$ and W denote functions whose argument is a vector in $\mathbb{R}^4$. The first of them choses the first three coordinates, the second choses the first two coordinates and the last one chooses the weight coordinate. Then $\boldsymbol{r}(\boldsymbol{P}) = \boldsymbol{Q}(\boldsymbol{P})/W(\boldsymbol{P})$ describes the transition from homogeneous to Cartesian coordinates and the vector $\overline{\boldsymbol{r}}(\boldsymbol{P}) = \overline{\boldsymbol{Q}}(\boldsymbol{P})/W(\boldsymbol{P})$ consists of the first two coordinates of $\boldsymbol{r}(\boldsymbol{P})$.

Let

$$\boldsymbol{P}(u,v) = \begin{bmatrix} \boldsymbol{p}(u,v) \\ 1 \end{bmatrix}, \quad \boldsymbol{N}(u,v) = \begin{bmatrix} \boldsymbol{n}(u,v) \\ 0 \end{bmatrix}, \quad \hat{\boldsymbol{P}}(u,v) = \begin{bmatrix} \hat{\boldsymbol{p}}(u,v) \\ 1 \end{bmatrix}.$$

In the standard cube coordinate system the patches are described by

$$\boldsymbol{r}\big(C\boldsymbol{P}(u,v)\big) = \boldsymbol{Q}\big(C\boldsymbol{P}(u,v)\big)/W\big(C\boldsymbol{P}(u,v)\big),$$
$$\boldsymbol{r}\big(C\hat{\boldsymbol{P}}(u,v)\big) = \boldsymbol{Q}\big(C\hat{\boldsymbol{P}}(u,v)\big)/W\big(C\hat{\boldsymbol{P}}(u,v)\big).$$

The functions $\overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u,v)\big)$ and $\overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u,v)\big)$ are parametrisations of the patches on the facet of the standard cube to be mapped onto the viewport.

Let (u_0,v_0) be a point in the patch domain corresponding to the point of the surface processed by the fragment shader. The image of the point $\hat{\boldsymbol{p}}(u_0,v_0)$ on the cube facet has the coordinates $(x,y) = \overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u_0,v_0)\big)$. We need a point $(u^\star,v^\star)$ such that $(x,y) = \overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u^\star,v^\star)\big)$. To evaluate any texture imposed on the patch, instead of (u_0,v_0) we shall use $(u^\star,v^\star)$. It means assuming that the fragment being processed corresponds to the point $\hat{\boldsymbol{p}}(u^\star,v^\star)$.

The point $(u^\star,v^\star)$ (which may not exist) satisfies the system of nonlinear equations

$$\overline{\boldsymbol{r}}\big(C\hat{\boldsymbol{P}}(u_0,v_0)\big) = \overline{\boldsymbol{r}}\big(C\boldsymbol{P}(u^\star,v^\star)\big).$$

To simplify, assume that $W\big(C\hat{\boldsymbol{P}}(u_0,v_0)\big) \approx W\big(C\boldsymbol{P}(u^\star,v^\star)\big)$, which results in the system of equations $\overline{\boldsymbol{Q}}\big(C\hat{\boldsymbol{P}}(u_0,v_0)\big) = \overline{\boldsymbol{Q}}\big(C\boldsymbol{P}(u^\star,v^\star)\big)$. Thus our task is to find a zero of the function

$$\boldsymbol{f}(u,v) = \overline{\boldsymbol{Q}}\Big(C\big(\hat{\boldsymbol{P}}(u_0,v_0) - \boldsymbol{P}(u,v)\big)\Big).$$

We shall use the Newton method, but we make just one iteration. We have

$$\boldsymbol{f}(u,v) = \overline{\boldsymbol{Q}}\Big(C\big(\boldsymbol{P}(u_0,v_0) + d\big(\boldsymbol{q}(u_0,v_0)\big)\boldsymbol{N}(u_0,v_0) - \boldsymbol{P}(u,v)\big)\Big).$$

The Newton method computes the point

$$(u_1,v_1) = (u_0,v_0) - \big(\mathrm{D}\boldsymbol{f}(u_0,v_0)\big)^{-1}\boldsymbol{f}(u_0,v_0).$$

Columns of the matrix $\mathrm{D}\boldsymbol{f}(u,v)$ are partial derivatives with respect to u, v.
As the function $\overline{\boldsymbol{Q}}$ is linear,

$$\boldsymbol{f}_u(u,v) = \overline{\boldsymbol{Q}}\big(-C\boldsymbol{P}_u(u,v)\big), \qquad \boldsymbol{f}_v(u,v) = \overline{\boldsymbol{Q}}\big(-C\boldsymbol{P}_v(u,v)\big),$$

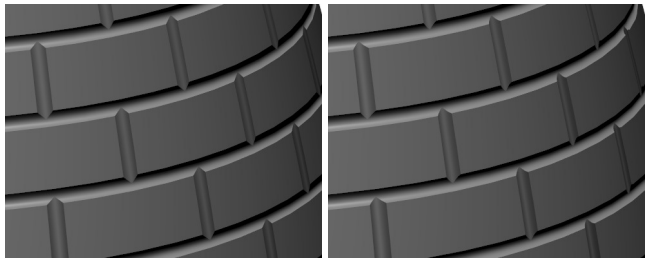
and we substitute

$$\boldsymbol{f}(u_0,v_0) = d\big(\boldsymbol{q}(u_0,v_0)\big)\overline{\boldsymbol{Q}}\big(C\boldsymbol{N}(u_0,v_0)\big),$$
$$\mathrm{D}\boldsymbol{f}(u_0,v_0) = \Big[-\overline{\boldsymbol{Q}}\big(C\boldsymbol{P}_u(u_0,v_0)\big), -\overline{\boldsymbol{Q}}\big(C\boldsymbol{P}_v(u_0,v_0)\big)\Big]$$

to the formula for the Newton method iteration.

Further steps of the Newton method would require computing the point $\boldsymbol{p}(u_1,v_1)$ and the partial derivatives of $\boldsymbol{p}$ at (u_1,v_1) and so on, and it would be the fragment shader's task. But the partial derivatives at (u_0,v_0) are produced by the tesselation shader.

Having the point (u_1,v_1), we can compute the gradient of $d\big(\boldsymbol{q}(u,v)\big)$ at this point and then we can substitute the derivatives and the normal vector of $\boldsymbol{p}$ at (u_0,v_0) to the formula $\mathrm{D}\hat{\boldsymbol{p}} = \mathrm{D}\boldsymbol{p} + (\mathrm{D}d \cdot \mathrm{D}\boldsymbol{q})\boldsymbol{n}$.



The picture to the left is obtained without the correction, whose effect is shown on the right picture.

Remarks: This method not always produces satisfactory effects. The displacement texture used above (the function d) has discontinuous derivatives, but $d = 0$ at the points of discontinuity, which is why it works well here.

The point $(u^\star,v^\star)$ being searched may not exist.

If the point $(u^\star,v^\star)$ exists, then the point (u_1,v_1) is only its approximation.

If the point (u_1,v_1) is outside the domain of the patch, then either the correction may be dropped or the closest point of the domain may be found, or the domain may be periodically extended onto the entire plane. A unique "always right" method *does not* exist!

This method cannot change the silhouette of the object.

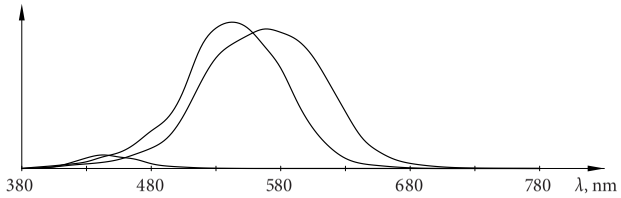
Colour space coordinates

Foundations of modern colorimetry were invented by Hermann Grassmann in 1853. He introduced the notions of light intensity (i.e., brightness), hue, saturation, dominant wavelength and complementary colours. He has also formulated laws of additive light blending.

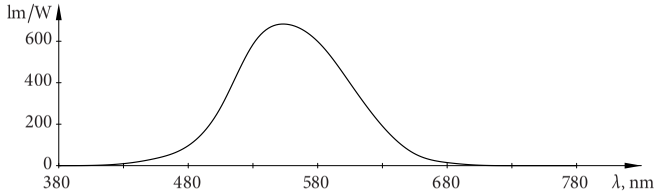
The Grassmann's laws in particular reflect the fact that many different stimuli (i.e., light having different spectra) may cause identical reactions of receptors in human eyes, which allows us to obtain a variety of colours by mixing just three primary colours, sufficient for most practical applications.

Light receptors in human eyes are of two kinds: **cones** and **rods**. Rods are more sensitive, but they cannot distinguish colours. Therefore the **scotopic vision** (in a nearly complete darkness, e.g., at night) shows a monochromatic world.

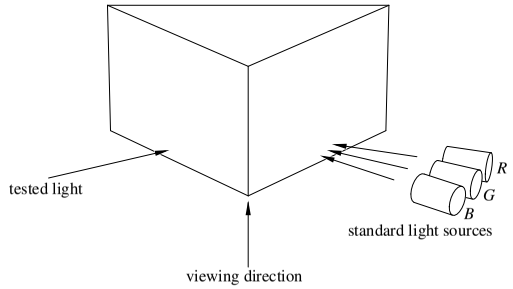
In the daylight a **photopic vision** takes place. There are three types of rods, most sensitive to light of different wavelengths. As a consequence, colour impressions are three-dimensional.



The function of light wavelength, which describes the total sensitivity of the receptors, is called the **luminous sensitivity**. Its value is the ratio of the light stream to the energy flux. Its maximal value, taken at $\lambda = 555\,\mathrm{nm}$ is $683\,\mathrm{lm/W}$. For the white light, depending on the light temperature, it ranges from about $50\,\mathrm{lm/W}$ to $250\,\mathrm{lm/W}$.

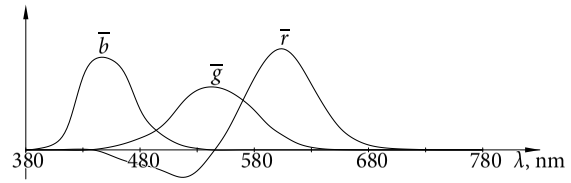


Research of the human vision is done with a **wedge colorimeter**. It is a chamber with black walls and a white wedge inside. One side of the wedge is enlighted by tested light and the other side by standard light. The standard light sources are a light bulb with a red filter (blocking light whose wavelength is less than 700 nm) and mercury lamps emitting light with wavelengths 546.1 nm and 435.8 nm.



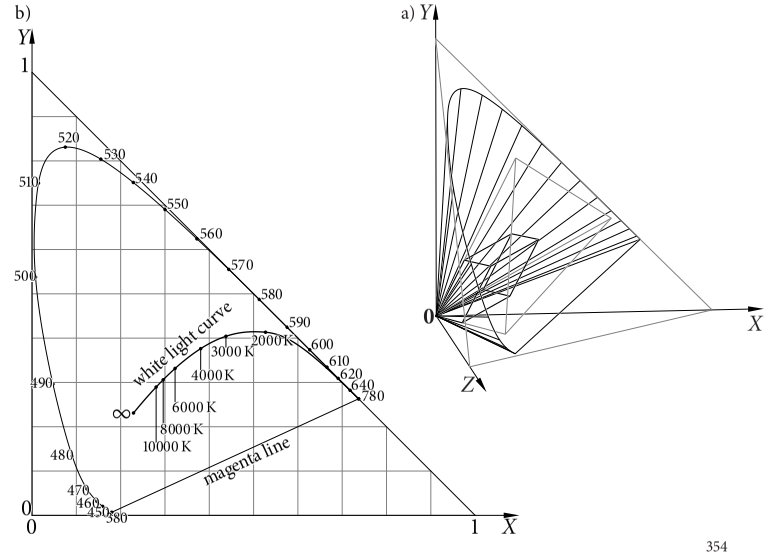
Both sides are seen through a peep-hole. The person being examined is supposed to set the diaphragms of the standard lights so as to obtain identical (according to that person) enlightment of both sides of the wedge. Colour coordinates are read from scales of the diaphragms. Adding some standard light to the tested light makes it possible to measure also negative coordinates.

A large number of experiments with colorimeters made it possible to find three functions called **chromatic coordinates**. The reaction of a cone to a light is the integral of the product of the chromatic coordinate function with the spectrum of this light.



The **chromaticity diagram** established in 1931 by the **International Commision on Illumination** (*CIE — Commision Internationale de l'Éclairage*) is a widely accepted basis for industry standards of colorimetry. It defines a system of coordinates called **CIE XYZ**. The "light" corresponding to the frame of this system *does not exist*. The area of visible colours is a convex cone in the positive octant of $\mathbb{R}^3$.

By fixing the light power, we obtain a planar cross-section of this area. It is convex and its boundary is made of two parts: the **rainbow curve**, whose points represent purely monochromatic light, and the **magenta line**.

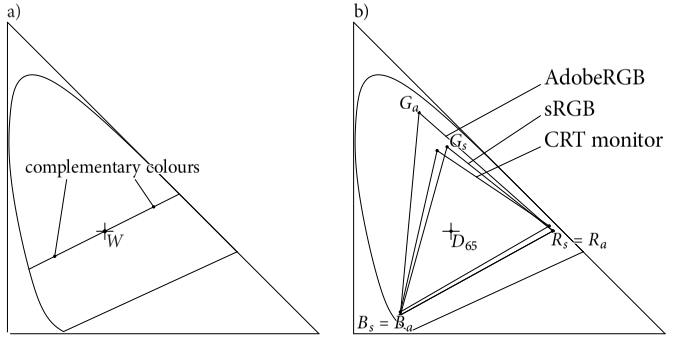


Points of the **white light curve** represent colours of light emitted by a **perfectly black object** heated to various temperatures. One can take any point of this curve as white. Most often it is the point labelled as D_{65}, which corresponds to 6500 K, which is a standard daylight. Computer screens make it possible to choose the white light temperature, usually ranging from 6000 K to 7500 K.

Light of any colour may be obtained by mixing the white light with a monochromatic light of one wavelength or two monochromatic lights, red and blue. In the former case we obtain light having a **dominant wavelength**.

Saturation of colour represented by any point ***c*** is the ratio of distance of this point from the white point and the distance of the point at the boundary of the visible light area from the white point such that the line segment joining the white point with the point at the boundary passes through ***c***.

Complementary colours are represented by points of a line segment passing through the white point, on the opposite sides and having the same saturation.



Colours possible to obtain on a computer screen are obtained by mixing light emitted by triplets of emitters of each pixels, which used to be phosphors in CRTs (cathode ray tubes) and now they are **light emitting diodes** (LED) or other devices. The picture b) shows the triangle of colours obtainable on a typical CRT screen and triangles defined by the standards **sRGB** (Microsoft & Hewlett–Packard, 1996) and **Adobe RGB** (Adobe Systems Inc, 1998).

If an image to be displayed on a monitor contains pixels with colours outside of the area (having some coordinates negative in a system whose frame is made of the vertices of the triangle), then it has to be converted so as to obtain and image with displayable colours, possibly with no prceptible distortions. Just clamping the coordinates to the range $[0,1]$ is usually a bad idea.

Human vision is least sensitive to changes of saturation. The correction to be done is called **desaturation** and it ought to be applied to *all* pixels, not just the ones with the colour sticking out of the triangle.

The transition between the coordinate systems CIE XYZ and sRGB is done in three steps. The first is a linar transformation, using the formula

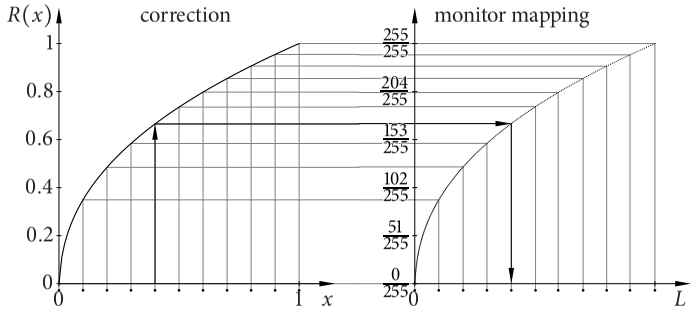
$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} 3.2406 & -1.5372 & -0.4986 \\ -0.9689 & 1.8758 & 0.0415 \\ 0.0557 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}.$$

The second step is clamping the coordinates to the interval $[0,1]$. The third step is the **gamma correction**. The power of light emitted by a phosphor in a CRT depends on the signal level (tension applied to the electrode) in a nonlinear way, approximately $L(x) = cx^{\gamma}$, where $\gamma \in [1.8, 2.8]$. To compensate, the coordinates r, g, b (proportional to the power) should be arguments of the inverse of this function. In the sRGB standard the following function is established:

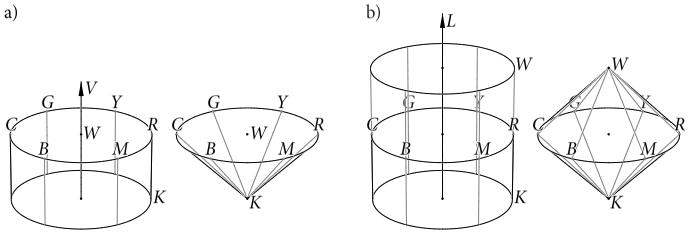
$$R(x) = \begin{cases} 12.92x & \text{dla } x < 0.0031308, \\ 1.055x^{1/2.4} - 0.055 & \text{w przeciwnym razie.} \end{cases}$$

The function values from the range $[0,1]$ are then represented as eight-bit numerators of fractions with the denominator 255.

This nonlinearity improves the accuracy of representation of hue of dark colours.



The RGB coordinate systems are convenient in computations, but they are not intuitive for users of grahhical applications, especially artists. What they need is the brightness, hue and saturation. The solid area of displayable colours is subject to a nonlinear mapping which produces a cylinder, though it is often shown as a cone of revolution or two such cones with a common basis. The coordinate systems used here are called **HSV** (*hue, saturation, value*) and **HSL** (*hue, saturation, lightness*) respectively.



(*Hue*) is usually specified in degrees. *Saturation* ranges between 0 on the cone axis, where the colour is grey or white (and hue is undefined) and 1 (on the cone surface). The V coordinate (*value*) ranges from 0 (black) and 1 (on the basis of the cone).

The HSL system was introduced in order to take into account the fact that the brightest light possible to dislay is white. Therefore the L coordinate takes values between 0 and 2. Pure colours of maximal lightness and saturation (at the boundary of the common basis of the cones) have lightness 1.