Computer Graphics - Lectures

Kamil Niedziałomski

Department of Mathematics and Computer Science
University of Lodz

Short description of lectures

- History of OpenGL
- Initial settings
- Drawing primitives
- Colors
- Order of transformations
- Matrix representation of transformations
- Light and materials
- Projection matrices
- Representation of geometric objects

OpenGL library

OpenGL is a multi–platform library for rendering 2D and 3D graphics. First version was released in January 1991.

Short history of OpenGL:

- OpenGL 1.0 – January 1991
- OpenGL 1.5 – July 2003
- OpenGL 2.0 – September 2004
- OpenGL 2.1 – July 2006 (**the one we use during laboratories**)
- OpenGL 3.0 – August 2008 (starting from this version we often call OpenGL a modern OpenGL)
- OpenGL 3.3 – March 2010
- OpenGL 4.0 – March 2010
- OpenGL 4.5 – August 2014
- Vulcan (known as glNext – the next generation of OpenGL) – will be released probably at the end of this year.

Additional libraries for OpenGL

It is not easy to create OpenGL window. Some of libraries support OpenGL context (window) creation:

- GLUT (GL Utility Kit), which is no longer mantained, or freeglut, which is more up to date (and originally written and developed by Polish programmer Paweł Olszta)
- Allegro 5
- SDL (Simple DirectMedia Layer) – **the one we use during laboratories**
- SFML (Simple and Fast Mutlimedia Library)

Initial setttings

We give a source code of the program, which allows to set OpenGL window.

```
#include<SDL/SDL.h>
#include<GL/gl.h>
#include<GL/glu.h>

void init()
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45,640.0/480.0,1.0,500.0);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_DEPTH_TEST);
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

Initial settings cont.

```
int main(int argc, char* args[])
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_SetVideoMode(640,480,32,SDL_SWSURFACE|SDL_OPENGL);
    bool loop=true;
    SDL_Event myevent;
    init();
    while (loop==true)
    {
        while (SDL_PollEvent(&myevent))
        {
            switch(myevent.type)
            {
                case SDL_QUIT: loop=false; break;
            }
        }
        display();
        SDL_GL_SwapBuffers();
    }
    SDL_Quit();
    return 0;
}
```

Initial settings cont.

- `init()` function initializes basic OpenGL properties,
- `display()` function displays the window,
- `SDL_SetVideoMode()` sets the parameters of the window.

Inside the `main()` function:

- we call `init()` function,
- then we have the game loop, in which we check events (for example for keyboard, mouse event) and call `display()` function
- the loop ends while `myevent.type` equals `SDL_QUIT`, that is when we close window,
- at the end we close all SDL events.

Notice that `display()` function is called (executed) each time inside the loop. In other words, the window is refreshed all the time.

Now we move to description of OpenGL 2.1 properties.

Drawing primitives

There are two ways to draw a certain object:

1. by using `glBegin()`, `glEnd()` block, or
2. by using vertex arrays.

We concentrate on the first method. We encourage readers who would like to learn modern OpenGL also to focus on the second option.
In order to draw a primitive we should put the name of the type of the primitive as the argument of the `glBegin()` function. There are the following types of primitives in OpenGL:

1. points `GL_POINTS`
2. lines `GL_LINES`
3. triangles `GL_TRIANGLES`
4. quadrilaterals `GL_QUADS`
5. polygons `GL_POLYGON`

Drawing primitives cont.

Inside the block you specify the coordinates of the vertices of the primitive by using the command `glVertex3f()`. `3f` stands for the 3 coordinates of each point, each coordinate of a float type. For example, to draw a triangle with the vertices $(0.4, -1.1, 0.3)$, $(1.0, 0.2, 0.0)$, $(1.0, -5.0, 3.5)$ we should write

```
glBegin(GL_TRIANGLES);
glVertex3f(0.4,-1.1,0.3);
glVertex3f(1.0,0.2,0.0);
glVertex3f(1.0,-5.0,3.5);
glEnd();
```

While drawing primitives we should remember about the following issues:

Drawing primitives cont.

1. The primitive must be planar and convex.
2. The vertices should be in the right order, i.e, the lines between consecutive vertices cannot cross.
3. If we want to draw few primitives of the same type we may use one block and specify the appropriate number of vertices being the multiplicity of number of vertices for one primitive or use few blocks of the same time. There are more possibilities to draw primitives with a common vertex (vertices) or edge:
   - `GL_TRIANGLE_FAN` – enables to draw fan of triangles, which share common edges and common vertex - the first vertex in the list,
   - `GL_LINE_STRIP` – draws a series of line segments connecting consecutive vertices,
   - `GL_TRIANGLE_STRIP` – enables to draw triangles, first made of first, second and trird vertex from the list, second triangle made of second, third and fourth vertex from the list, and so on,
   - `GL_QUAD_STRIP` – draws a sequence of quadrilaterals. The ith quadrilateral is made of vertices $v_{2i-1}, v_{2i}, v_{2i+1}, v_{2i+2}$ from the list $v_1, v_2, \ldots, v_{2n+2}$, where n is the number of quadrilaterals.

Colors

OpenGL uses RGB and RGBA color models. Each coordinate can take real values in the interval $[0, 1]$ or integers from 0 to 255 depending on the command: `glColor3f()` or `glColor3ub` for *RGB* model and `glColor4f()` or `glColor4ub()` for *RGBA* model.
The specified color by one of above commands is set for all primitives which are followed by this command. In OpenGL you may also specify the color of each vertex. Then the color of the primitive will be interpolated linearly (see also the section about light and shading).

Colors cont.

The following two examples illustrate above considerations.

```
glColor3f(1.0,0.0,0.0);
glBegin(GL_TRIAGNLES);
glVertex3f(0.0,0.0,-5.0);
glVertex3f(0.0,1.0,-5.0);
glVertex3f(1.0,0.0,-5.0);
glEnd();
```

```
glBegin(GL_TRIAGNLES);
glColor3f(1.0,0.0,0.0);
glVertex3f(0.0,0.0,-5.0);
glColor3f(0.0,1.0,0.0);
glVertex3f(0.0,1.0,-5.0);
glColor3f(0.0,0.0,1.0);
glVertex3f(1.0,0.0,-5.0);
glEnd();
```

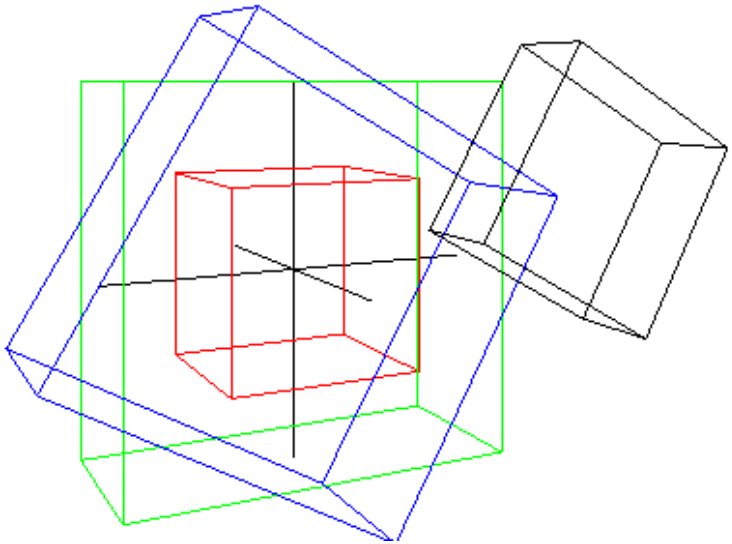
Order of transformations in OpenGL

Assume we want to transform certain point $p(x_0, y_0, z_0)$ in the three dimensional space in the following way: first we execute scaling with scales $(2, 2, 1)$, then rotate in xy–plane by an angle $\frac{\pi}{3}$ and translate by vector $(4, 0, -3)$,

(1) $$\text{Transformation:} \quad T_{(4,0,-3)} \longleftarrow R_{(0,0,1),\frac{\pi}{3}} \longleftarrow S_{(2,2,1)}.$$

In other words, point p transforms to a point q, where

$$q = T_{(4,0,-3)}\, R_{(0,0,1),\frac{\pi}{3}}\, S_{(2,2,1)} p.$$



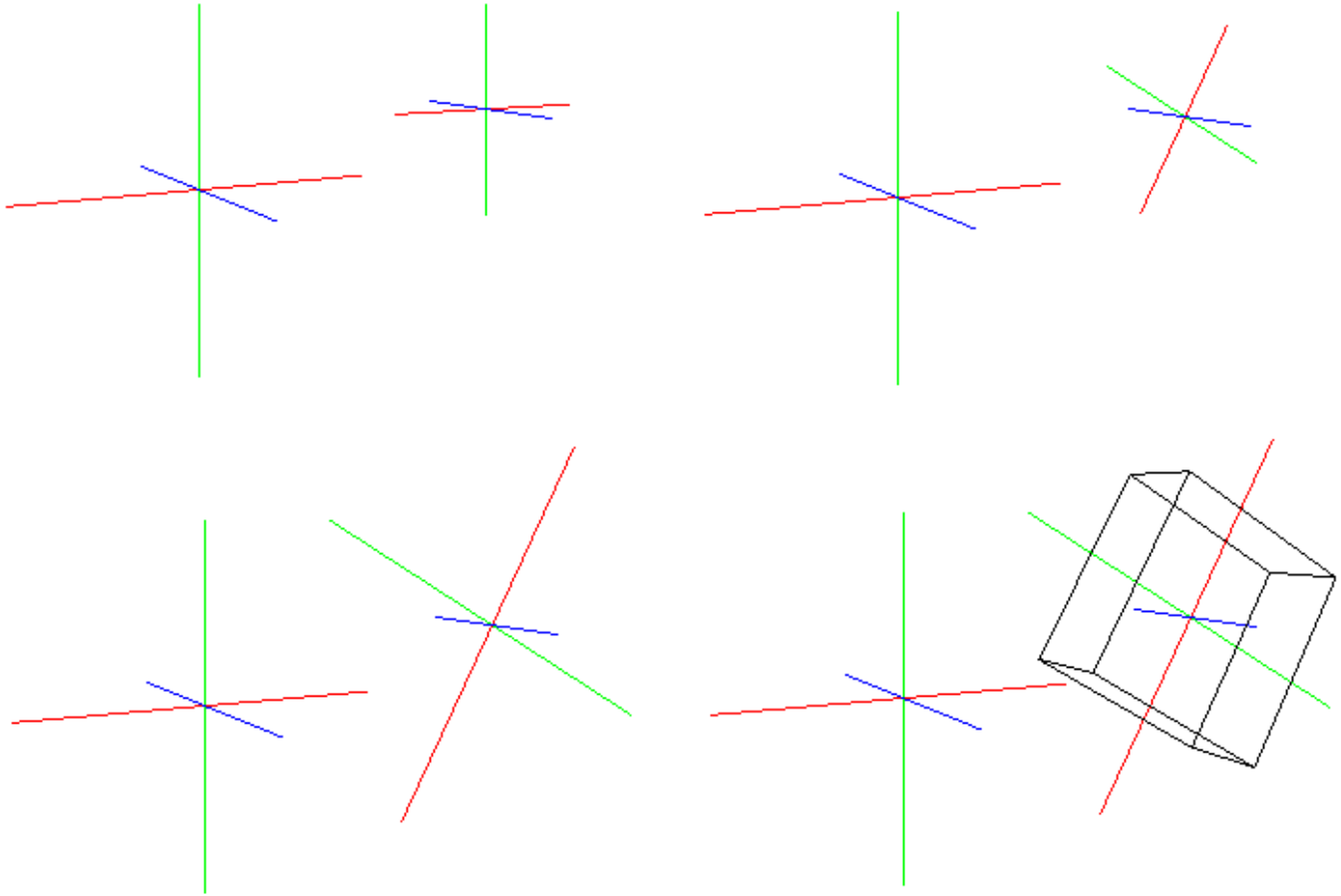
Order of ransformations in OpenGL cont.

Since the first transformation is in deed the last one in order of appearance in the source code

```
glTranslatef(4,0,-3);
glRotatef(60,0,0,1);
glScalef(2,2,1);
DrawCube();
```

it is hard to follow each step.
The solution to this problem is the following: instead of considering transformations as functions on the points or objects, we can think about transformations as transformations on the coordinate system.

Order of transformations in OpenGL cont.



Order of transformations in OpenGL cont.

Then the order is, as the transformations appear (from left to right), the following

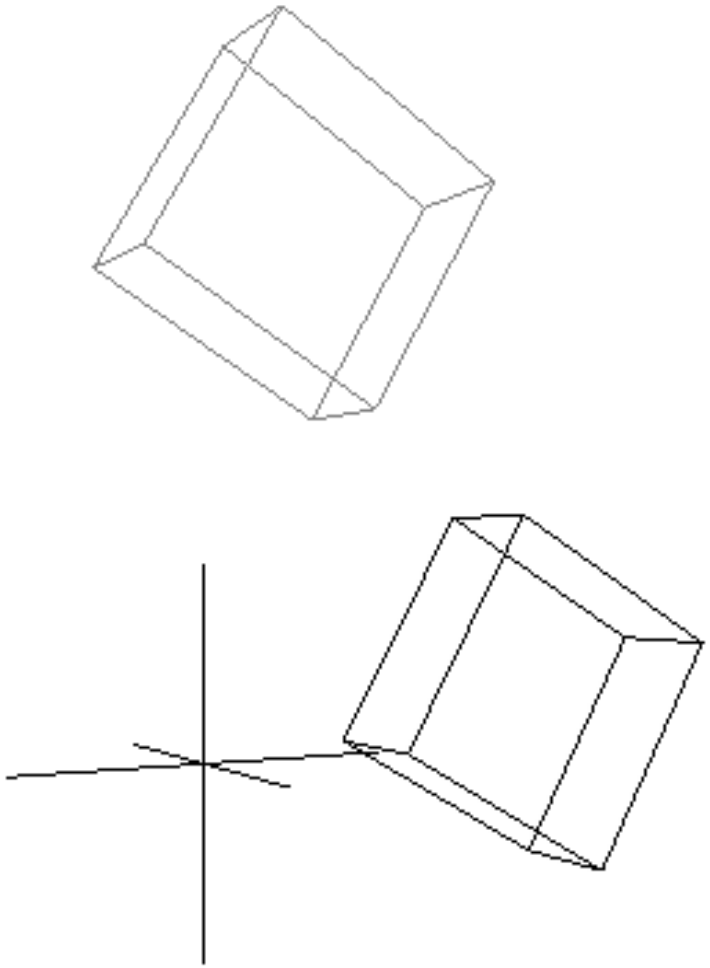
$$T_{(4,0,-3)} \longrightarrow R_{(0,0,1),\frac{\pi}{3}} \longrightarrow S_{(2,2,1)}. \tag{2}$$

In other words, first we translate the coordinate system, then rotate and scale. In the end we draw an object (point) with respect to this "new" coordinate system. Notice that if we interchange, for example, rotation with translation, that is

$$\text{Transformation:} \quad R_{(0,0,1),\frac{\pi}{3}} \longleftarrow T_{(4,0,-3)} \longleftarrow S_{(2,2,1)}. \tag{3}$$

we get a different point.
This follows from the fact that the order of transformations is important.

Order of transformations in OpenGL cont.



Linear transformations

Let us first recall some knowledge of linear algebra. Let V be a linear space (for our purposes it is enough to take $V = \mathbb{R}^2$ or $V = \mathbb{R}^3$). Denote by $\mathbf{0}$ the zero vector. By a linear transformation we mean a mapping $A : V \to V$ such that

1. $A(v + w) = A(v) + A(w)$ for $v, w \in V$ (linearity),
2. $A(\alpha v) = \alpha A(v)$ for $v \in V$, $\alpha \in \mathbb{R}$ (homogenity).

Notice that taking $\alpha = 0$ we get $A(\mathbf{0}) = \mathbf{0}$. More generally, homogenity implies that any line passing through $\mathbf{0}$ is mapped on the line passing through $\mathbf{0}$. Let us consider few examples.

Example

Let $V = \mathbb{R}^2$ and let $A : \mathbb{R}^2 \to \mathbb{R}^2$ be of the form

$$A(x, y) = (2x + 3y, -x + y), \quad (x, y) \in \mathbb{R}^2.$$

One can check that A is linear. Notice, for example, that A maps line $t \mapsto t(1, 1)$ to the line $t \mapsto tA(1, 1)$, which equals $t \mapsto t(5, 0)$.

Matrix representation of linear transformation

Let $A : V \to V$ be a linear transformation, $\dim V = n$. Fix a basis $e_1, e_2, \ldots, e_n$ in V. Then we can express vectors $A(e_i)$, $i = 1, 2, \ldots, n$ with respect to that basis. Thus there are coefficients a_{ij}, $i, j = 1, 2, \ldots, n$, such that

$$A(e_i) = \sum_{j=1}^{n} a_{ji} e_j.$$

The matrix $(a_{ij})_{i,j=1,2,\ldots,n}$ is called the matrix of A an denoted by the same letter A.

Example

Let $e_1 = (1, 0)$ and $e_2 = (0, 1)$. Then for the transformation A in the previous example we have

$$A(e_1) = A(1, 0) = (2, -1) = 2e_1 - e_2, \quad A(e_2) = A(0, 1) = (3, 1) = 3e_1 + e_2.$$

Thus the matrix of A equals

$$A = \begin{pmatrix} 2 & 3 \\ -1 & 1 \end{pmatrix}.$$

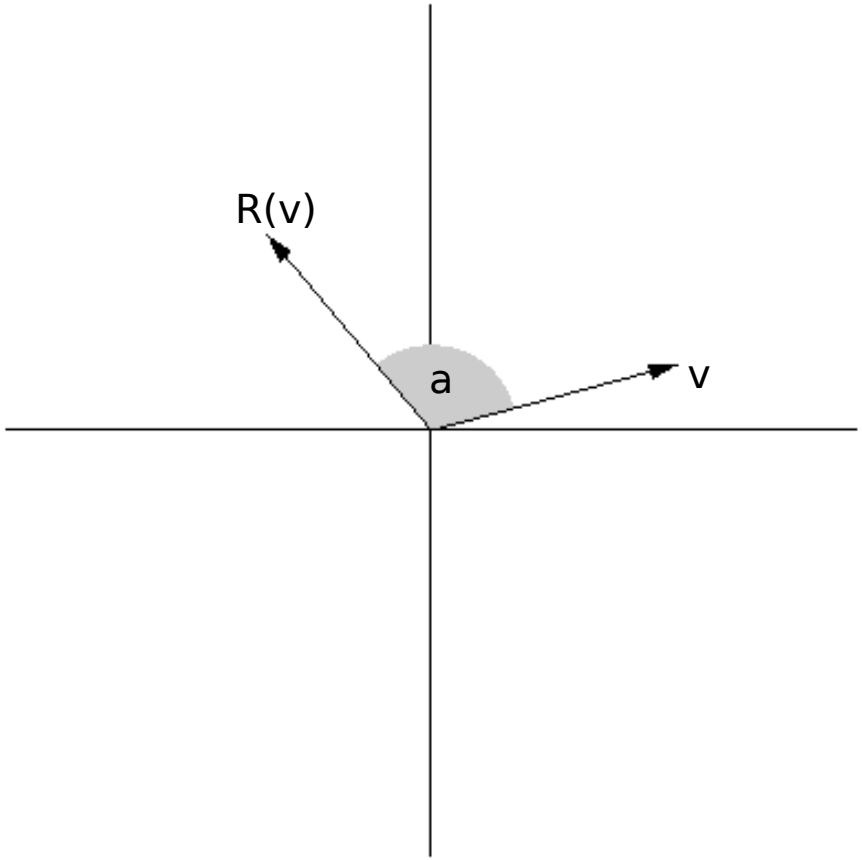
Geometric transformation – Rotation

Fix an angle a. Consider a rotation R_a around $(0,0)$ through an angle a. It is nod hard to show that

$$R_a(x,y) = (x\cos a - y\sin a, x\sin a + y\cos a).$$

and R_a is a linear transformation. In matrix notation

$$R_a \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$



Geometric transformation – Scaling

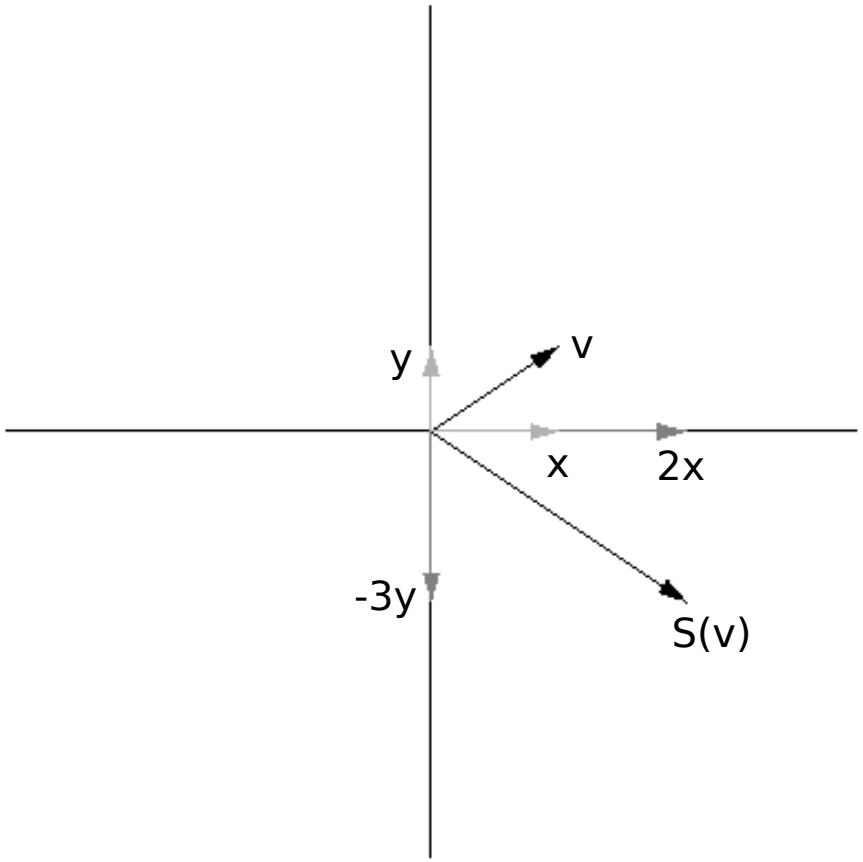
We can scale a figure in each direction by a different factor. Let s_x and s_y represent scaling factors along x and y–axis, respectively.
Scaling S can be described as follows

$$S(x, y) = (s_x x, s_y y), \quad (x, y) \in \mathbb{R}^2.$$

S is a linear transformation and in matrix notation

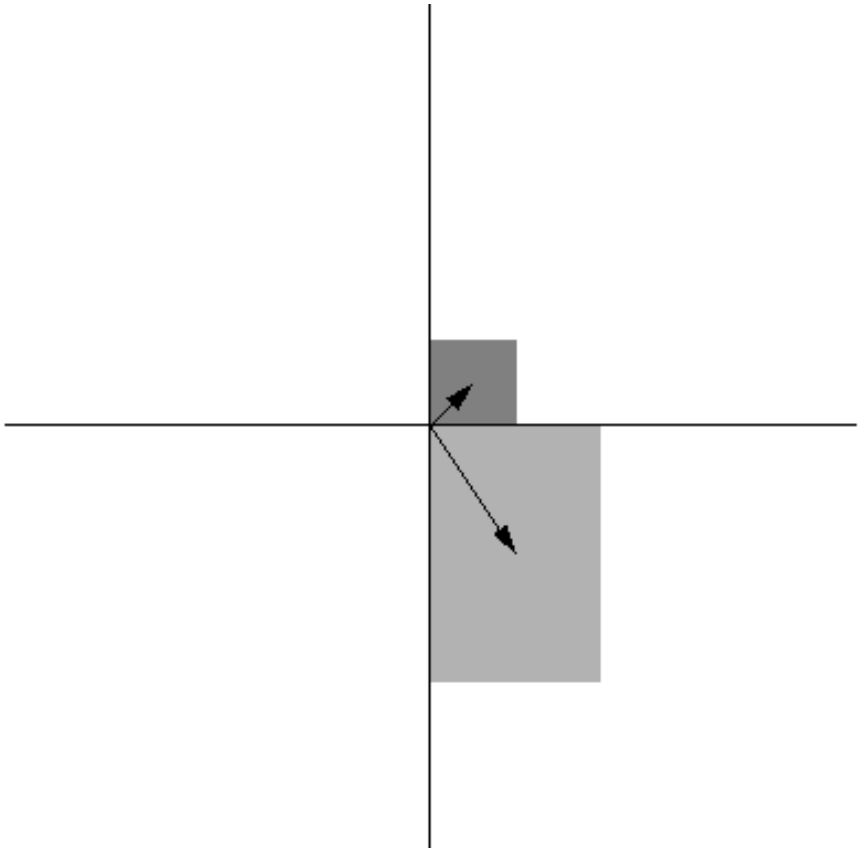
$$S \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

In the figure, $s_x = 2$ and $s_y = -3$.



Be careful with scaling

Notice that scaling changes the sizes of objects but if the "center" of the object is different from $\mathbf{0} = (0, 0)$, then the object is also translated.



Scaled square with the scales $(s_x, s_y) = (2, -3)$

Key fact about matrix representation of transformation

The use of the matrix instead of the linear transformations itself is useful, what illustrates the following simple proposition.

Theorem

Let $A, B : V \to V$ be two linear transformations. Then the composition $A \circ B : V \to V$ is a linear transformation and the matrix of this transformation equals

$$A \cdot B \quad (= A \circ B).$$

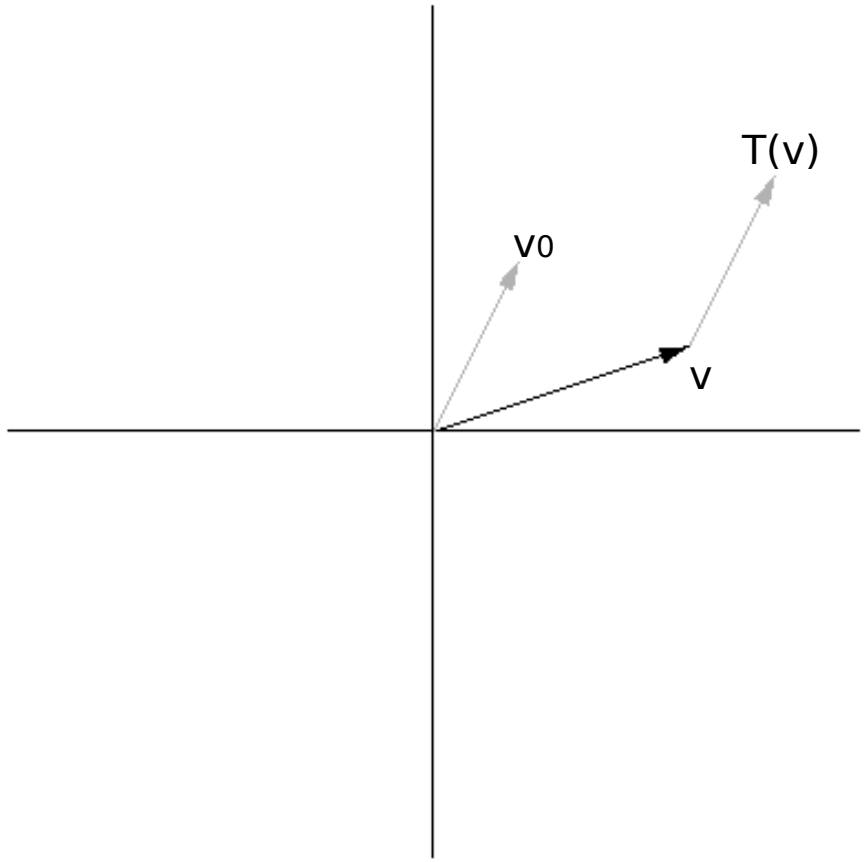
Geometric transformation – Translation

Fix a vector $v_0 = (x_0, y_0)$ and consider transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ of the form

$$T(x, y) = (x + x_0, y + y_0).$$

In other words, not using coordinates, $T(v) = v + v_0$.

The transformation T if $v_0 \neq \mathbf{0}$ is not linear. It is clear since $T(\mathbf{0}) = v_0 \neq \mathbf{0}$. This transformation is very often used in computer graphics. It would be convenient to represent it also by some matrix. The solution follows by introducing so called homogeneous coordinates.



Homogeneous coordinates

Notice that above transformation T maps lines to lines but not necessary lines passing through $\mathbf{0}$ to lines passing through $\mathbf{0}$. Indeed, any line $t \mapsto tw$, $w \in V$ is mapped to the line $t \mapsto v_0 + tw$, i.e., the line passing through v_0 in the direction of w. Such maps are called affine.

The attempt to obtain a linear map from the affine map is to introduce a new variable, which corresponds to the condition of homogenity, i.e, the movement along lines passing through $\mathbf{0}$. The value $\lambda = 1$ corresponds to identity. More precisely, a point (x, y) is identified with all the points lying on the line passing through $\mathbf{0} = (0, 0, 0)$ and induced by the vector $(x, y, 1)$ in $\mathbb{R}^3$. Therefore (x, y) corresponds to all points $(\alpha x, \alpha y, \alpha)$. Since T maps (x, y) to $(x + x_0, y + y_0)$, then $(x, y, 1)$ corresponds to $(x + x_0, y + y_0, 1)$. Therefore

$$(\alpha x, \alpha y, \alpha z) \sim (x, y, 1) \mapsto (x + x_0, y + y_0, 1) \sim (\alpha x + \alpha x_0, \alpha y + \alpha y_0, \alpha).$$

Defining new variables

$$x' = \alpha x, \quad y' = \alpha y. \quad z' = \alpha,$$

transformation T takes the form

$$(x', y', z') \mapsto (x' + x_0 z', y' + y_0 z', z'),$$

which is clearly a linear transformation.

The matrix of this mapping is

$$T = \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix}.$$

To sum up all above considerations, each point $(x, y) \in \mathbb{R}^2$ is identified with $(x, y, 1)$ and with any point $(\alpha x, \alpha y, \alpha)$ and we consider any affine transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ as a linear map $T : \mathbb{R}^3 \to \mathbb{R}^3$ constructed as above. In particular, rotation R_a and scaling S are represented by matrices

$$R_a = \begin{pmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Transformations in OpenGL

Three transformations described in the previous sections – translation, rotation and scaling – are set in OpenGl, respectively, as follows:

1. `glTranslatef(x,y,z)`, where (x, y, z) defines the translation vector,
2. `glRotatef(angle,x,y,z)`, where `angle` is the angle of rotation in degrees, (x, y, z) is a vector defining the axis of rotation. Notice that the length of this vector is of no importance.
3. `glScalef(sx,sy,sz)`, where `sx`, `sy`, `sz` define the scale factors on each coordinate.

In OpenGL each transformation is represented by the matrix, so called, model view matrix (`GL_MODELVIEW`). The identity transformation, that is, a transformation, which does nothing, is represented by the identity matrix. This matrix in OpenGL is called by the command: `glLoadIdentity()`.
This is why at the beginning of `display()` function we always call this function.

OpenGL transformations – example

Let us consider the following example, in which we define the `renderScene()` to initiate the scene and `DisplayScene()` functions. We transform a green triangle.

```
float angle=0.0;
void init()
{
    glClearColor(1.0,1.0,1.0,1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0,640.0/480.0,1.0,100.0);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_DEPTH_TEST);
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(2.0,1.0,-7.0);
    glRotatef(angle,0.0,1.0,0.0);
    glScalef(1.0,-2.0,1.0);
    glColor3f(0.0,1.0,0.0);

    Draw a triangle;
}
```

OpenGL transformations – example continued

The modification of model view matrix is the following:

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \rightarrow I \cdot T = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -7 \\ 0 & 0 & 0 & 1 \end{pmatrix} \rightarrow$$

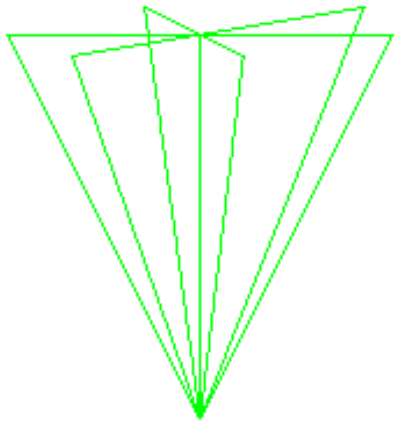
$$I \cdot T \cdot R = \begin{pmatrix} \cos a & 0 & -\sin a & 0 \\ 0 & 1 & 0 & 0 \\ \sin a & 0 & \cos a & 0 \\ 2 & 1 & -7 & 1 \end{pmatrix} \rightarrow$$

$$I \cdot T \cdot R \cdot S = \begin{pmatrix} \cos a & 0 & -\sin a & 0 \\ 0 & -2 & 0 & 0 \\ \sin a & 0 & \cos a & 0 \\ 2 & 1 & -7 & 1 \end{pmatrix}.$$

Then we multiply each point of the triangle by this matrix to obtain transformed triangle.

OpenGl transformation – outcome of the example

We show the outcome for `alpha` taking values 30.0, 90.0, 150.0, 210.0, 270.0 and 330.0 degrees. Notice that if we put at the end of `display()` function, for example, `alpha+=1.0`; we get animation of rotating triangle with the values of the angle increasing by 1.0 starting from alpha equal 30.0.



Light – introduction

To make the scene realistic, we need to consider different sources of light (the light of the Sun, light from the lamps, etc.) and the effect of the light. Let I denote the intensity (brightness, color) of the light. We describe how the intensity I changes after reflection from an object.

First consider the scene with sourceless light (ambient light, AL). Rays of this light go in all directions. Let $k_a \in \langle 0, 1 \rangle$ be the coefficient of reflection of the light from the object depending of the material the object is made of. Then the intensity of the light after reflection is given by

$$I = I_a k_a,$$

where I_a is the intensity of AL being constant for all objects.

Source of light

Now, we add to the ambient light one source of light (SL). The rays of the light go in all directions from the source. Let k_d denotes the coefficient of reflection of the light from the object with respect to this light. Then

$$I = I_a k_a + I_d k_d \cos t,$$

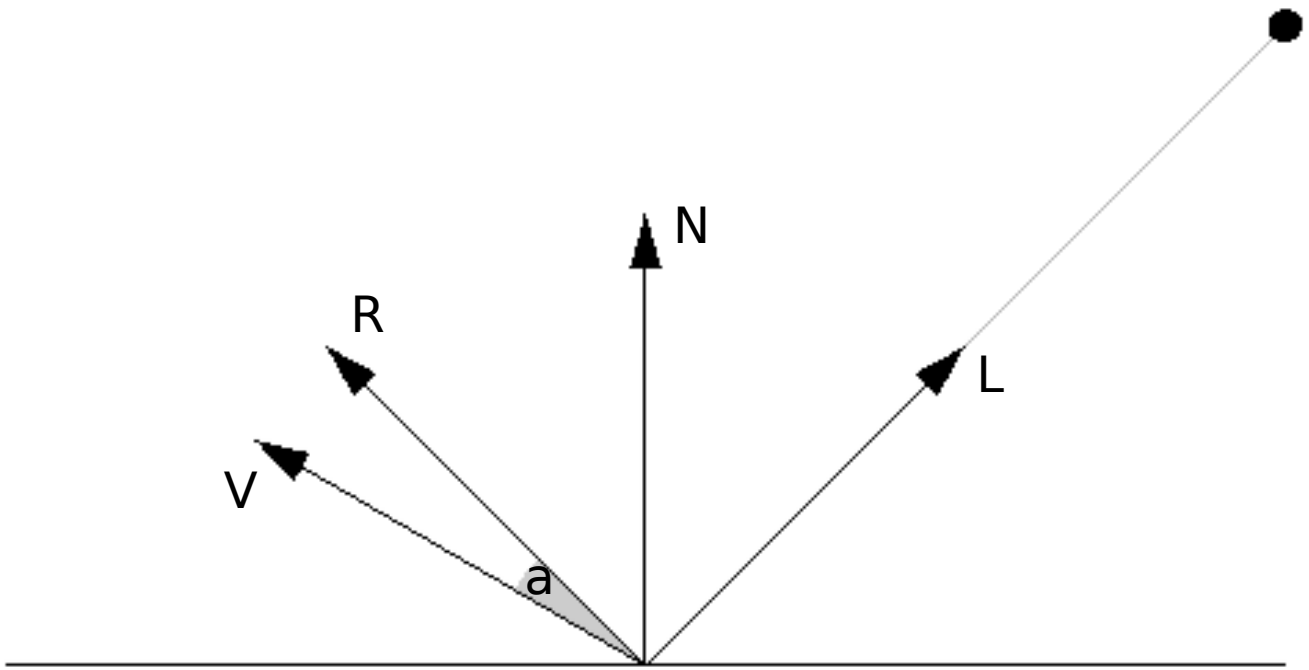
where I_d is the brightness of SL and t is the angle between the direction of the light and the vector normal to the surface of an object. If we denote by L the unit vector of direction of the light and by N the unit normal (outer) vector to the surface, above formula can be rewritten as

$$I = I_a k_a + I_d k_d \langle L, N \rangle,$$

where $\langle L, N \rangle$ is the scalar product on L and N,

$$\langle L, N \rangle = L_x N_x + L_y N_y + L_z N_z, \quad L = (L_x, L_y, L_z), \quad N = (N_x, N_y, N_z).$$

Source of light – picture



Source of light – continued

We also need to consider the distance between the object and the source of light. If the source of the light is further the intensity of the light is smaller, hence we consider the coefficient of attenuation

$$f_{\mathrm{att}} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right),$$

where d_L is the distance from the source to the object, c_1, c_2, c_3 constants depending on the light. Then I becomes

$$I = I_a k_a + f_{\mathrm{att}} I_d k_d \langle L, N \rangle.$$

Real objects are not ideal and reflect the light not only in one direction but in directions forming a cone. Phong proposed a coefficient $\cos^n a$, where n is a natural number from the interval $\langle 1, 200 \rangle$, and a is the angle between the vector of reflection R and the direction V of the viewer.

Source of the light – continued

For mat surfaces n is close to 1, for shiny surfaces n is large. Moreover, we should consider the coefficient $W(a)$ which measures the intensity of the light with respect to the angle a, since the intensity depends on the angle of reflection. If R and V are unit vectors, then $\cos a = \langle V, R \rangle$. Thus we obtain

$$I = I_a k_a + f_{\mathrm{att}}(I_d k_d \langle L, N \rangle + W(a) \langle V, R \rangle^n).$$

If we have many sources of light $SL_1, \ldots, SL_m$ then we add intensities to obtain

$$I = I_a k_a + \sum_{i=1}^{m} f_{\mathrm{att,i}}(I_{d_i} k_{d_i} \langle L_i, N \rangle + W(a_i) \langle V, R_i \rangle^n).$$

Often $W(a_i)$ is a constant I_{s_i}, called specular light intensity, hence we get

$$I = I_a k_a + \sum_{i=1}^{m} f_{\mathrm{att,i}}(I_{d_i} k_{d_i} \langle L_i, N \rangle + I_{s_i} \langle V, R_i \rangle^n).$$

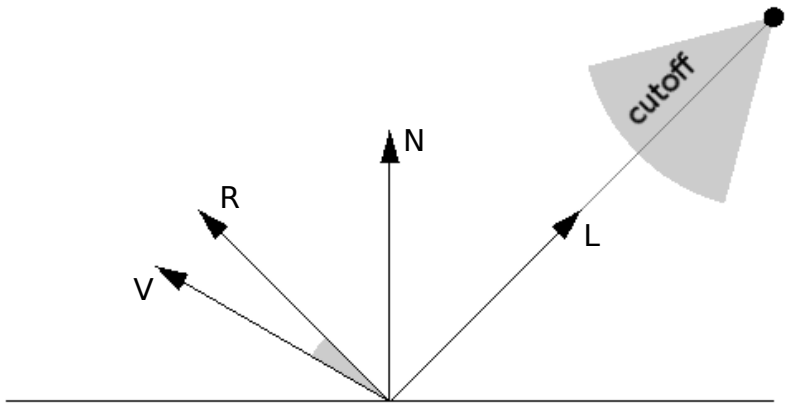
The practical use of above formula

Now we will use above formula in the considerations of the influence of color of the object on the color of the reflected light. In general:

1. I stands for the color of the light which comes to the viewer (after reflection from the object). Thus, in RGB model, I is a vector $I = (I_r, I_g, I_b)$, where each coordinate gives the amount of red, green and blue color, respectively. Analogously vectors I_a, I_d, I_s denote the ambient, diffuse and specular color of the source of light (AL or SL).
2. k_a, k_d denote vectors $k_a = (k_{ar}, k_{ag}, k_{ab})$, $k_d = (k_{dr}, k_{dg}, k_{db})$ defining colors, in RGB model, of the object which reflects and absorbs the light.

Notice that above formula should be considered separately for each coordinate of the color.

Light in OpenGL



In OpenGL the formula for the color of reflected light slightly differs and is given as follows

$$I = I_0 + k_e + e_{\text{spot}}\, f_{\text{att}}(I_a k_a + I_d k_d \langle L, N\rangle + I_s k_s \langle V, R\rangle^n), \tag{4}$$

where we have three more color vectors: k_s, which defines the specular color of the material of considered object, k_e, which defines the emission color of the material and I_0, which defines global ambient color independent of light sources; and some constant e_{spot}.

Light in OpenGL – continued

If we have more light sources $L0, L1, \ldots, Lm$, the formula takes the form

$$I = I_0 + k_e + \sum_{i=0}^{m} e_{\mathrm{spot,i}}\, f_{\mathrm{att,i}}(I_{a,i}k_a + I_{d,i}k_d\langle L_i, N\rangle + I_{s,i}k_s, \langle V, R_i\rangle^n),$$

where each light Li has its components $I_{a,i}, I_{d,i}, I_{s,i}, L_i, R_i, f_{\mathrm{att,i}}, e_{\mathrm{spot,i}}$.

To enable the use of the light we should write `glEnable(GL_LIGHTING)`. We can define up to eight light sources from `GL_LIGHT0` to `GL_LIGHT7`. We do it by, for example, `glEnable(GL_LIGHT0)`. To disable, for example `GL_LIGHT0`, we use `glDisable(GL_LIGHT0)`. The definition of the properties of certain light is provided by the following command

```
glLightfv(GL_LIGHTX, ATTRIBUTE, VALUE);
```

where `GL_LIGHTX` is one of the lights, $X = 0, 1, \ldots, 7$, `ATRRIBUTE` denotes one of the parameters of the light and the `VALUE` defines the value of the `ATTRIBUTE`.

Light attributes

The parameters can be the following:

- `GL_AMBIENT` – I_a in (4) – defines the RGBA color of the ambient light, i.e., the color of the light which seems to go in all directions and which is everywhere,
- `GL_DIFFUSE` – I_d in (4) – defines the RGBA color of the diffuse light, i.e., the light which comes from the light source and which is equally bright no matter where the viewer is situated,
- `GL_SPECULAR` – I_s in (4) – defines the RGBA color of the specular light, i.e., the light which has its source, direction and which approximates the reflection from the shiny surface,
- `GL_POSITION` – allows to compute d_L for f_{att} – defines the position of the specular light in homogeneous coordinates (x, y, z, w). If $w = 1$ then the position is (x, y, z), if $w = 0$, then the light has no source and goes in one direction defined by the vector (x, y, z).
- `GL_SPOT_DIRECTION` – L in (4) – defines the direction of the light in usual coordinates (x, y, z).

Light attributes – continued

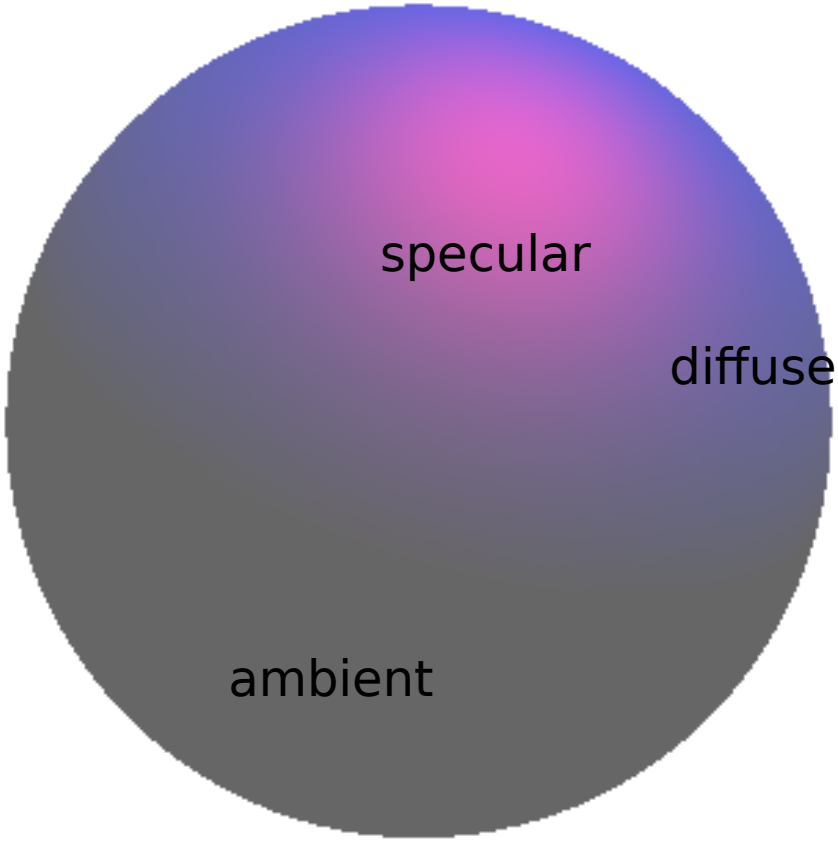
- `GL_SPOT_CUTOFF` – defines the cutoff angle, i.e., half of the light dispersal angle (if the value is in the range $(0, 90)$). The other possible value is 180, which corresponds to the light going in all directions from the source.
- `GL_SPOT_EXPONENT` – allows to compute e_{spot} in (4) as follows: e_{spot} equals 0 if the angle between N and L is greater than the cutoff angle; e_{spot} equals 1 if the light is directional with no source; e_{spot} equals $\langle N, L \rangle^e$, where e defines the value of the `GL_SPOT_EXPONENT`.
- `GL_CONSTANT_ATTENUATION` – c_1 in f_{att} – defines the constant for f_{att}.
- `GL_LINEAR_ATTENUATION` – c_2 in f_{att} – defines the "linear factor" for f_{att}
- `GL_QUADRATIC_ATTENUATION` – c_3 in f_{att} – defines the "quadratic factor" for f_{att}.

The global ambient light I_0 is set up by the command

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,I0)
```

where `I0` is the RGBA color of this global light.

Ambient, diffuse and specular light



Materials

Materials define the color of objects (primitives), but each material has more components than just one color. The components, similar to the ones for the light, are the following

- `GL_AMBIENT` – k_a in (4) – defines the ambient color of the material.
- `GL_DIFFUSE` – k_d in (4) – defines the diffuse color of the material.
- `GL_SPECULAR` – k_s in (4) – defines the specular color of the material.
- `GL_SHININESS` – n in (4) – defines the shininess of the material, takes values in the interval $\langle 1.0, 128.0 \rangle$ or 0.0.
- `GL_EMMISION` – k_e in (4) – defines the emission color.

In order to set each of above components we use `glMaterialfv(FACE,COMP,VALUE)`, where `FACE` is one of `GL_FRONT`, `GL_BACK` or `GL_FRONT_AND_BACK` depending on which side of the faces of the object we want to put material, `COMP` denotes one of above mentioned components and `VALUE` is the value of the component (RGBA vector for k_a, k_d, k_s, k_e and the constant n).

Normal vectors

To guarantee correct use of the light we need to define normal vectors (denoted by N). The normal vector allows to compute the direction R of reflected light knowing the direction L of incoming light. In OpenGL we define a vector normal to the primitive (a face). We define normal vector while defining the primitive. Normal vector must be of unit length. If the vector normal to a certain face is (x, y, z) then we write `glNormal3f(x,y,z)`. For example,

```
glBegin(GL_TRIANGLES);
glNormal3f(0.0,1.0,0.0);
glVertex3f(-1.0,0.0,-1.0);
glVertex3f(1.0,0.0,-1.0);
glVertex3f(0.0,0.0,1.0);
glEnd();
```

defines the triangle in the xz–plane with the unit normal vector pointing towards positive y–axis.

Something more about initial settings

Recall the declaration of the `Init()` function:

```
void init()
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45,640.0/480.0,1.0,500.0);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_DEPTH_TEST);
}
```

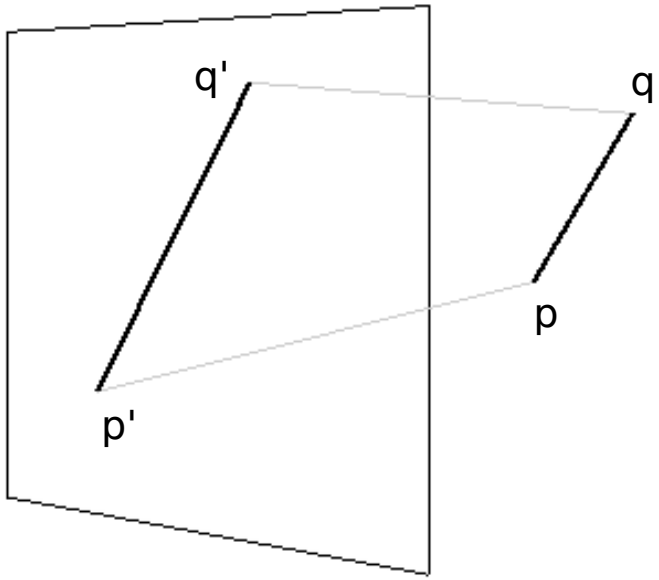
We will focus on the matrix mode and projection matrices. In order to see a 3D scene on a screen we have to project it to 2D image. This is realized by, so called, projections. We distinguish two types of projections:

- orthographic,
- perspective.

Let us describe each of them in more detail.

Orthographic projection

Orthographic projection is a projection on a plane along fixed vector.



This projection can be used for drawing 2D scenes, since it does not detect the depth of the scene. In other words, the projection of the object in independent of the distance (along vector of projection) of the object from the projection plane.

Orthographic projection cont.

For applications, it is convenient to choose a projection plane given by the equation $z = 0$, and a direction of projection as $v = (0, 0, 1)$. Then the projection takes the obvious form

$$(x, y, z) \mapsto (x, y, 0).$$

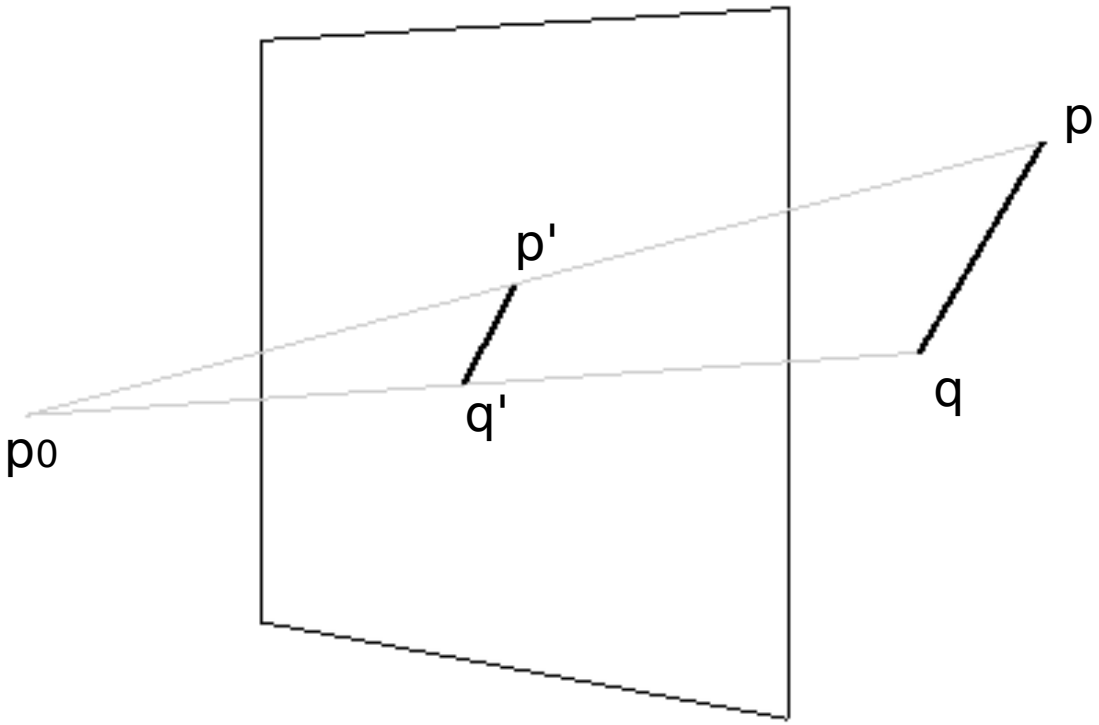
Hence in the homogeneous coordinates this projection has the form

$$\mathrm{Proj}_{\mathrm{ort}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This is a matrix of orthographic projection. Each point (vector) multiplied by this matrix gives the 2D point (in homogeneous coordinates) which is displayed on a screen.

Perspective projection

Perspective projection is defined as follows: Let p_0 be the center of projection (point where the camera is situated) and Π the plane of projection. Then any point p is projected to p' in such a way that $p' \in \Pi$ and p_0, p, p' lie on a line.



Perspective projection cont.

To obtain the formula for this projection, we take $\Pi : z = 0$ and $p_0 = (0, 0, -d)$, where $d > 0$. Let $p = (x, y, z)$, $p' = (x', y', 0)$. Since p' lies on the line $\overline{p_0 p}$ it is given by $p' = (1 - t)p_0 + tp$ for some $t \in \mathbb{R}$. Moreover the z–coordinate of p' is 0, hence $0 = (1 - t)(-d) + tz$. Thus $t = \frac{d}{d+z}$. Finally

$$p' = \left(\frac{dx}{d+z}, \frac{dy}{d+z}, 0 \right).$$

Since $\left(\frac{dx}{d+z}, \frac{dy}{d+z}, 0, 1 \right)$ and $(dx, dy, 0, d + z)$ define the same point in homogeneous coordinates, the matrix of perspective projection is

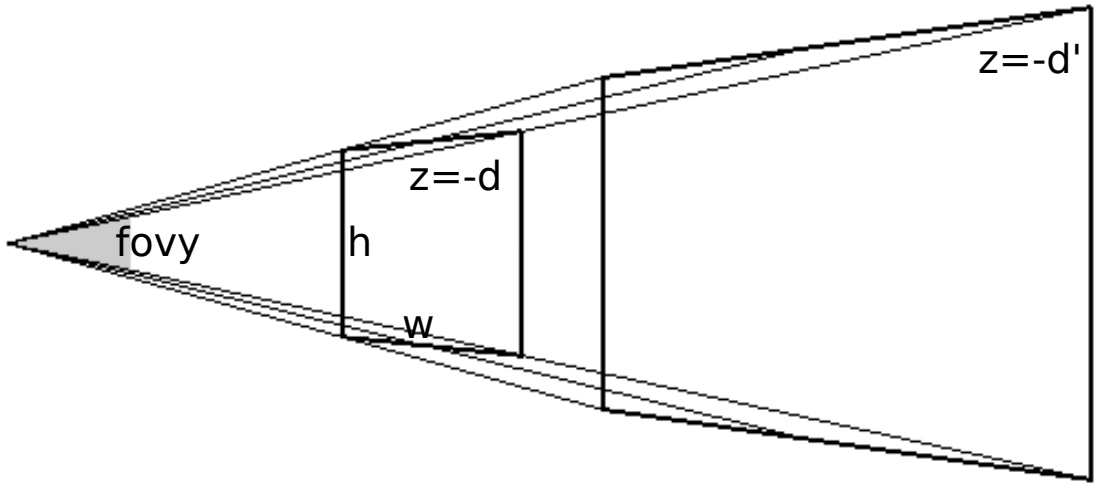
$$\mathrm{Proj}_{\mathrm{per}} = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & d \end{pmatrix}.$$

How to set up each type of projection?

Let us introduce the commands, which cause each type of projection to be executed (in OpenGL):

1. `glOrtho()`, which we use in the following way: `glOrtho(left,right,bottom,top,near,far)`, where left, right, bottom, top are the values determining the view plane, near and far determine the z–position of clipping planes. We should notice that the ratio $\frac{\text{right}-\text{left}}{\text{top}-\text{bottom}}$ should be the same as the ratio $\frac{\text{width}}{\text{length}}$ of the sizes of the window in order to obtain the same units on each x and y–axis.
2. `gluPerspective()`, which requires glu library, sets the projective projection as follows: `gluPerspective(fovy,aspect,zNear,zFar)`, where all variables are of `double` type; `fovy` (`field of view y`) is the angle of the view, aspect is the ratio of the width to the height of the plane of view (window) at z–coordinate zNear, zNear and zFar determine the z–position of the clipping planes.

Matrix mode



To set one of above projections, we write, for example,

```
glMatrixMode(GL_PROJECTION);
glOrtho(-10.0,10.0,-10.0,10.0,1.0,50.0);
```

or

```
glMatrixMode(GL_PROJECTION);
gluPerspective(45.0,640.0/480.0,1.0,500.0);
```

Orthographic projection in OpenGL

The orthographic projection is defined as before, with the only differences, that

1. we need to control z–coordinate in order to enable OpenGL to use depth test (to know which objects are closer than others),
2. we project object between planes $z = -\texttt{near}$ and $z = -\texttt{far}$, $\texttt{near}, \texttt{far} > 0$,
3. we project, in deed, the window $[\mathrm{left}, \mathrm{right}] \times [\mathrm{bottom}, \mathrm{top}] \times [\mathrm{near}, \mathrm{far}]$ to the cube $[-1, 1] \times [-1, 1] \times [-1, 1]$.

It is not hard to see that the affine transformation which satisfies the third condition is

$$P(x, y, z) = \left(\frac{2x}{\mathrm{right} - \mathrm{left}} - \frac{\mathrm{right} + \mathrm{left}}{\mathrm{right} - \mathrm{left}}, \frac{2y}{\mathrm{top} - \mathrm{bottom}} - \frac{\mathrm{top} + \mathrm{bottom}}{\mathrm{top} - \mathrm{bottom}}, \frac{-2z}{\mathrm{far} - \mathrm{near}} - \frac{\mathrm{far} + \mathrm{near}}{\mathrm{far} - \mathrm{near}} \right).$$

In homogeneous coordinates, the matrix of this projection equals

$$\mathrm{Proj}_{\mathrm{glOrtho}} = \begin{pmatrix} \frac{2}{\mathrm{right}-\mathrm{left}} & 0 & 0 & -\frac{\mathrm{right}+\mathrm{left}}{\mathrm{right}-\mathrm{left}} \\ 0 & \frac{2}{\mathrm{top}-\mathrm{bottom}} & 0 & -\frac{\mathrm{top}+\mathrm{bottom}}{\mathrm{top}-\mathrm{bottom}} \\ 0 & 0 & \frac{-2}{\mathrm{far}-\mathrm{near}} & -\frac{\mathrm{far}+\mathrm{near}}{\mathrm{far}-\mathrm{near}} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Orthographic projection in OpenGL cont.

In practice, we take

$$\text{right} = -\text{left} = \frac{aw}{2}, \quad \text{top} = -\text{bottom} = \frac{ah}{2}, \quad \text{far} = -\text{near} = d,$$

where w and h denote the width and height of the window and a is some scale. This allows to get the same units on each coordinate x and y. Notice that z–coordinate is used only to control the order of displaying objects, from the furthest to the nearest.
Concluding, we can write, in `Init()` function and `main()` function:

```
glMatrixMode(GL_PROJECTION);
glOrtho(-13.3,13.3,-10.0,10.0,-200.0,200.0);
...
SDL_SetVideoMode(640,480,32,SDL_SWSURFACE|SDL_OPENGL);
```

Notice that the ratio $\frac{13.3}{10.0} \approx \frac{4}{3}$ is the same as the aspect $\frac{640.0}{480.0}$.

Perspective projection in OpenGL

The perspective projection is defined as a (regular) perspective projection but with scaled view plane to the square $[-1,1] \times [-1,1]$ and different center of projection and plane of projection. Namely, we project to $z = -d$ and with respect to the point $(0,0,0)$.
From the definition of `gluPerspective()` we have the following relations

$$\text{aspect} = \frac{w}{h}, \quad \text{tg}\left(\frac{\text{fovy}}{2}\right) = \frac{h}{2d},$$

where w and h are the width and the height of the "plane" of projection at $z = -d$. Put

$$\varphi = \text{ctg}\left(\frac{\text{fovy}}{2}\right).$$

It is easy to see, as in the section concerning transformations, that the formula for the projection to $z = -d$ plane, with respect to the point $(0,0,0)$ equals

$$\text{Proj}(x,y,z) = \left(\frac{dx}{-z}, \frac{dy}{-z}, -d\right).$$

Perspective projection in OpenGL cont.

Now, applying the scaling in xy–plane, which scales the rectangle $\left[-\frac{w}{2}, \frac{w}{2}\right] \times \left[-\frac{h}{2}, \frac{h}{2}\right]$ to the square $[-1, 1] \times [-1, 1]$ equals

$$S(x, y) = \left(\frac{2x}{w}, \frac{2y}{h}\right).$$

Thus projection takes the form

$$\mathrm{Proj}(S(x, y), z) = \left(\frac{2dx}{-wz}, \frac{2dy}{hz}, \frac{dz}{-z}\right) = \left(\frac{\varphi x}{-az}, \frac{\varphi y}{-z}, \frac{dz}{-z}\right).$$

In homogeneous coordinates, the matrix of this transformation equals

$$\begin{pmatrix} \frac{\varphi}{a} & 0 & 0 & 0 \\ 0 & \varphi & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Perspective projection in OpenGL cont.

In order to support depth test, i.e., to distinguish points on xy–plane, with respect to its previous z–coordinate, we want to store information about z–coordinate. Thus we seek for two constants A and B such that the third row of above matrix equals $0\,0\,A\,B$ and such that plane $z = -d$ is mapped to $z = -1$ and plane $z = -d'$ is mapped to $z = 1$. Thus we want

$$T(z) = \frac{Az + B}{-z}, \quad T(-d) = -1, \quad T(-d') = 1.$$

Solving this system we get

$$A = \frac{d + d'}{d - d'}, \quad B = \frac{2dd'}{d - d'}.$$

Finally, the matrix of `gluPerspective()` is the following

$$\mathrm{Proj}_{\mathrm{gluPersp}} = \begin{pmatrix} \frac{\varphi}{a} & 0 & 0 & 0 \\ 0 & \varphi & 0 & 0 \\ 0 & 0 & \frac{d+d'}{d-d'} & \frac{2dd'}{d-d'} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

The two–dimensional point projected to $z = -d$ plane is $\left(\frac{\varphi x}{-az}, \frac{\varphi y}{-z}\right)$.

Perspective projection in OpenGL cont.

In practice, we do not use only `gluPerspective()` but also `SDL_SetVideoMode`, which defines the size of the window (screen). **It is convenient to define the size of the window such that the ratio of width to height agrees with the aspect of the projection.** Thus we need to transfer square $[-1,1]\times[-1,1]$ back to the window, i.e., we execute the inverse transform S^{-1} with some scale α (which depends on the sizes of the screen). In this case, the final perspective projection, with the convention of remembering z–coordinate, takes the form

$$\mathrm{Proj}_{\mathrm{gluPersp+SetVideoMode}} = \begin{pmatrix} \alpha d & 0 & 0 & 0 \\ 0 & \alpha d & 0 & 0 \\ 0 & 0 & \frac{d+d'}{d-d'} & \frac{2dd'}{d-d'} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Concluding, we have to use for example,

```
glMatrixMode(GL_PROJECTION);
gluPerspective(45.0, 640.0/480.0,1.0,500.0);
...
SDL_SetVideoMode(640,480,32,SDL_SWSURFACE|SDL_OPENGL);
```

Perspective projection in OpenGL cont.

In the above example, the size of the window by `gluPerspective()` is

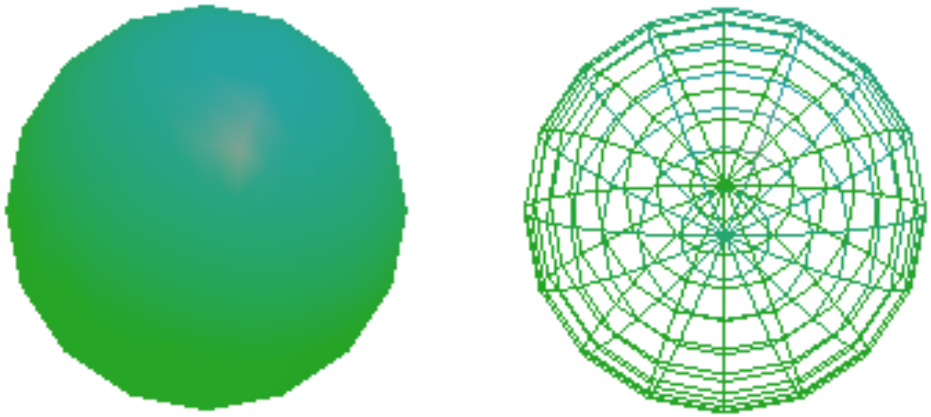
$$h = 2d\,\mathrm{tg}\left(\frac{\mathrm{fovy}}{2}\right) = 2\,\mathrm{tg}\left(\frac{\pi}{8}\right) \approx 0.83,$$

$$w = \mathrm{aspect} \cdot w = \frac{4}{3} \cdot w \approx 1.10.$$

Since the true window size is $h = 480$, $w = 640$ by `SDL_SetVideoMode()`, the scaling factor α equals approximately $\alpha \approx 582$. Notice that we transfer units to pixels. In this case, 582 pixels equals approximately one unit (at the level $z = -d$).

Representation of geometric objects

We will discuss different methods of description of geometric objects. By a **geometric object** we mean everything what can be drawn using computer. Hence, a curve, surface, solid, etc. are geometric objects. The amount of date needed to give a complete description of an object is called a **representation** of this object. The representation of a geometric object not only depends on the shape of the object, but also on the way of visualization. For example, consider a sphere of given radius. Depending on the resolution of the monitor, use of the light, number of primitives the sphere is build from, we get different representations.



We will deal only with geometric aspects of objects.

We may describe curves and surfaces in many ways. A plane curve may be given by implicit equation

$$f(x, y) = 0, \quad (x, y) \in \mathbb{R}^2.$$

Under some additional conditions, we can locally solve above equation and get description of a curve as a graph of a function

$$y = \varphi(x), \quad x \in \mathbb{R}.$$

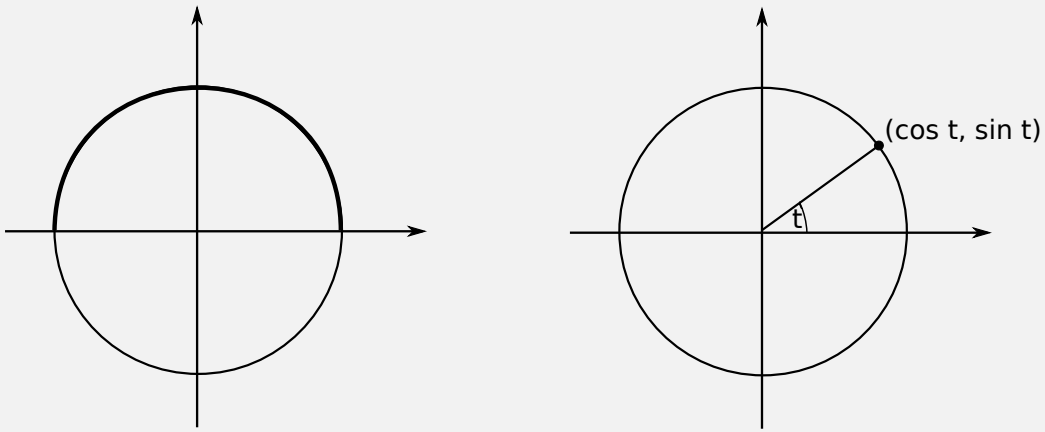
We may also consider parametric representation of a plane curve

$$x = x(t), \quad y = y(t), \quad t \in \mathbb{R}.$$

Example

A unit circle centered at $(0,0)$ can be described as $f(x,y) = x^2 + y^2 - 1 = 0$ which simplifies to

$$y = \sqrt{1-x^2} \quad \text{or} \quad y = -\sqrt{1-x^2}, \quad x \in \langle -1, 1 \rangle.$$



The parametric representation is obviously $\gamma(t) = (x(t), y(t))$, where

$$x(t) = \cos t, \quad y(t) = \sin t, \quad t \in \langle 0, 2\pi),$$

Notice that a circle cannot be represented as a graph of one function.

Representation of curves and surfaces cont.

When we consider curves in a 3–dimensional space analogously we have implicit description

$$f(x, y, z) = 0, \quad g(x, y, z) = 0$$

and parametric representation

$$x = x(t), \quad y = y(t), \quad z = z(t), \quad t \in \mathbb{R}.$$

Representation of a surface in a 3–dimensional space is simliar to representation of a curve. Namely, we can consider a surface given by implicit equation

$$f(x, y, z) = 0,$$

as a graph of a function

$$z = f(x, y), \quad (x, y) \in \mathbb{R}^2.$$

or given parametrically

$$x = x(u, v), \quad y = y(u, v), \quad z = z(u, v), \quad (u, v) \in \mathbb{R}^2.$$

Representation of solid objects

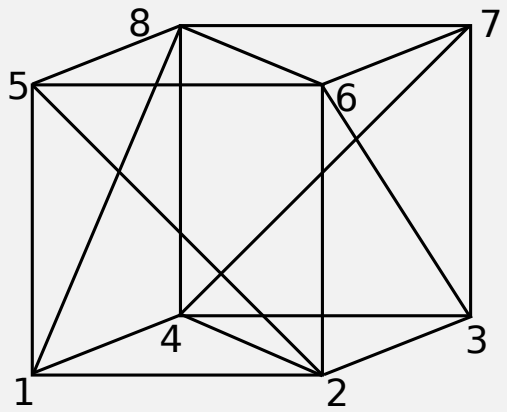
By a **solid object** we mean a 3–dimensional object made of finite number of faces, where each face is a polygon. We assume that the faces of the object intersect only along edges or vertices. The classical method of remembering the data needed to construct given object is to make few lists: list of vertices, list of edges and list of faces:

1. List of vertices consists of coordinates of vertices.
2. list of edges consists of pairs of numbers. Each pair represents two vertices of the edge.
3. list of faces consists of finite sequences of numbers. Each sequence represents the edges of the face.

Representation of a cube

Example

As an example consider a cube.



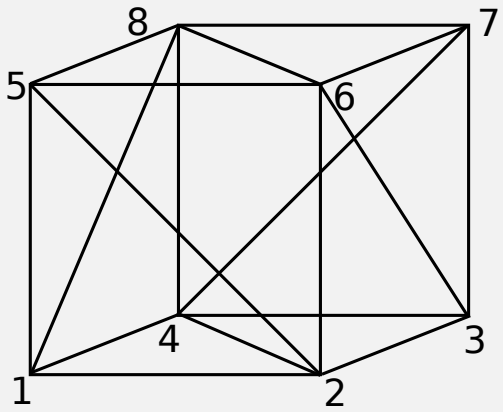
Then we have

- **list of vertices:** $(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_8, y_8, z_8)$.
- **list of edges:** (Number 1 denotes the first vertex in the list of vertices, etc.)

$$(1,2),(2,3),(3,4),(4,1),(5,6),(6,7),(7,8),(8,5),(1,5),$$
$$(2,6),(3,7),(4,8),(2,5),(3,6),(4,7),(1,8),(2,4),(6,8).$$

Representation of a cube cont.

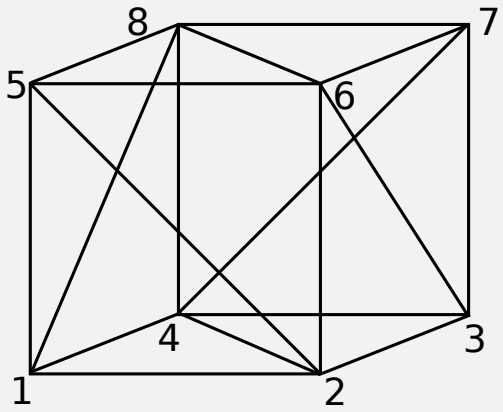
Example



- **list of faces** Each face is a triangle and we have 12 faces.

$$(1, 17, 4), (2, 3, 17), (1, 13, 9), (10, 5, 13), (2, 11, 14), (14, 2, 10),$$
$$(7, 12, 15), (11, 3, 15), (4, 12, 16), (16, 8, 9), (5, 18, 8), (6, 7, 18).$$

Example



We quite often consider the representation without the list of edges. Then the list of vertices is the same, whereas the list of faces consists of numbers of vertices (not edges) of each face. In our example

- **the new list of faces:**

$$(1,2,4),(2,3,4),(1,2,5),(2,6,5),(2,3,6),(3,7,6),$$
$$(3,4,7),(4,7,8),(1,4,8),(1,8,5),(5,6,8),(6,7,8).$$

Additional information in the representation

In order to get the representation of an object, which allows to apply properly illumination, we need to specify normal vectors. Normal vectors are unit length vectors orthogonal to faces or, with different types of shading, unit length vectors attached to vertices. Therefore, we need one more **list of normal vectors** and modified list of faces with the information about, so called, normals. Below we show the obj file obtained in 3D graphics program Blender representing a cube.

Obj file representing a cube in Blender

```
# Blender v2.69 (sub 0) OBJ File: ''
# www.blender.org
o Cube
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 -0.000000 0.000000
vn 0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
vn 1.000000 0.000000 0.000001
s off
f 1//1 2//1 4//1
f 5//2 8//2 6//2
f 1//3 5//3 2//3
f 2//4 6//4 3//4
f 3//5 7//5 4//5
f 5//6 1//6 8//6
f 2//1 3//1 4//1
f 8//2 7//2 6//2
f 5//7 6//7 2//7
f 6//4 7//4 3//4
f 7//5 8//5 4//5
f 1//6 4//6 8//6
```

Cube in Blender – explanation

Let us shortly explain the content of the file. First two lines are informative only. Letter `o` stands for the name of the object. Then we have the list of vertices (letter `v`) and the list of normal vectors (letters `vn`). There are six normal vectors as we have six quadratic faces of the cube. `s` stands for smoothing which is disabled. In the end there is list of faces (letter `f`). Since each face is a triangle there are 2×6 faces. For example,

```
f 5//6 1//6 8//6
```

means that this face is made of 5th, 1st and 8th vertex and that 6th normal vector is a normal vector at each vertex.