# Computer Graphics - Lectures ( version 2023)

Kamil Niedziałomski

Department of Mathematics and Computer Science
University of Lodz

# Part I - OpenGL

- history of OpenGL
- initial settings; graphics pipeline
- drawing primitives; colors
- transformations
- light, materials
- projections
- few words on modern OpenGL

# 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 (first release)
- OpenGL 1.5 – July 2003
- OpenGL 2.0 – September 2004
- OpenGL 2.1 – July 2006
- OpenGL 3.0 – August 2008 (starting from this version we often say modern OpenGL)
- OpenGL 3.3 – March 2010
- OpenGL 4.0 – March 2010
- OpenGL 4.5 – August 2014
- OpenGL 4.6 – July 2017 (last version)
- Vulcan (known as glNext – the next generation of OpenGL) – February 16, 2016 (first release)
- last stable release of Vulcan – October 13, 2023 (version 1.3.268)

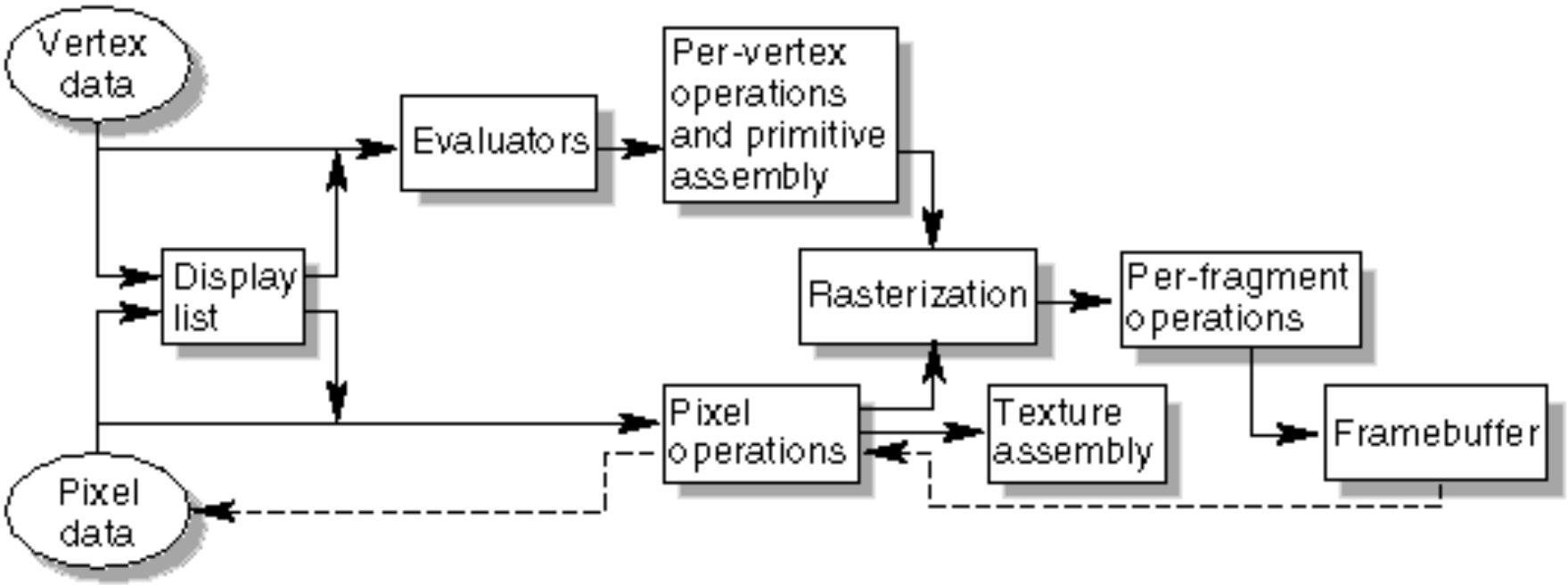
# Additional libraries for OpenGL

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)
- SFML (Simple and Fast Mutlimedia Library)
- GLFW (For modern OpenGL)

# Graphics pipeline

A series of processing operations from vertex data to raster image is often called **graphics pipeline**. In OpenGL 2.1 it containes the following stages:



# 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 $i$th 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

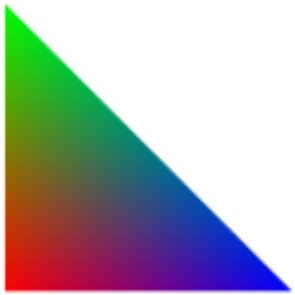
OpenGL uses RGB and RGBA color models. Each coordinate can take real values in the interval $[0, 1]$ or integers from 0 to 255 depending on the command: `glColor3f()` or `glColor3ub` for *RGB* model and `glColor4f()` or `glColor4ub()` for *RGBA* model.
The specified color by one of above commands is 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();
```



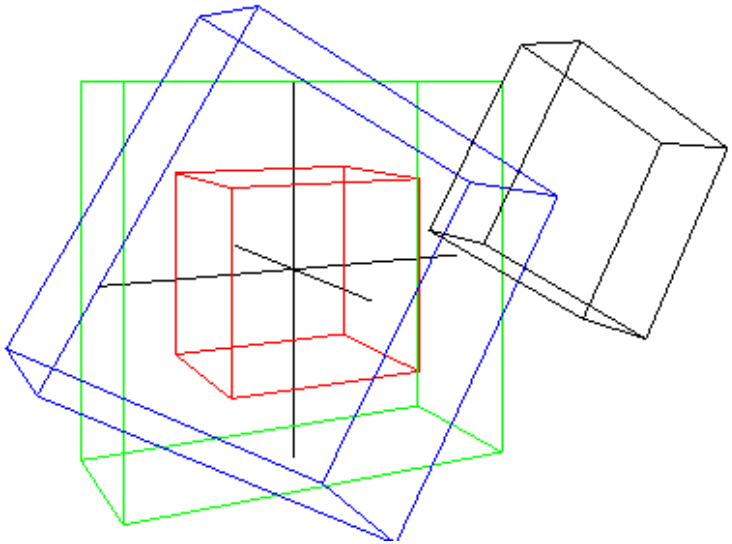
# Order of transformations in OpenGL

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

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



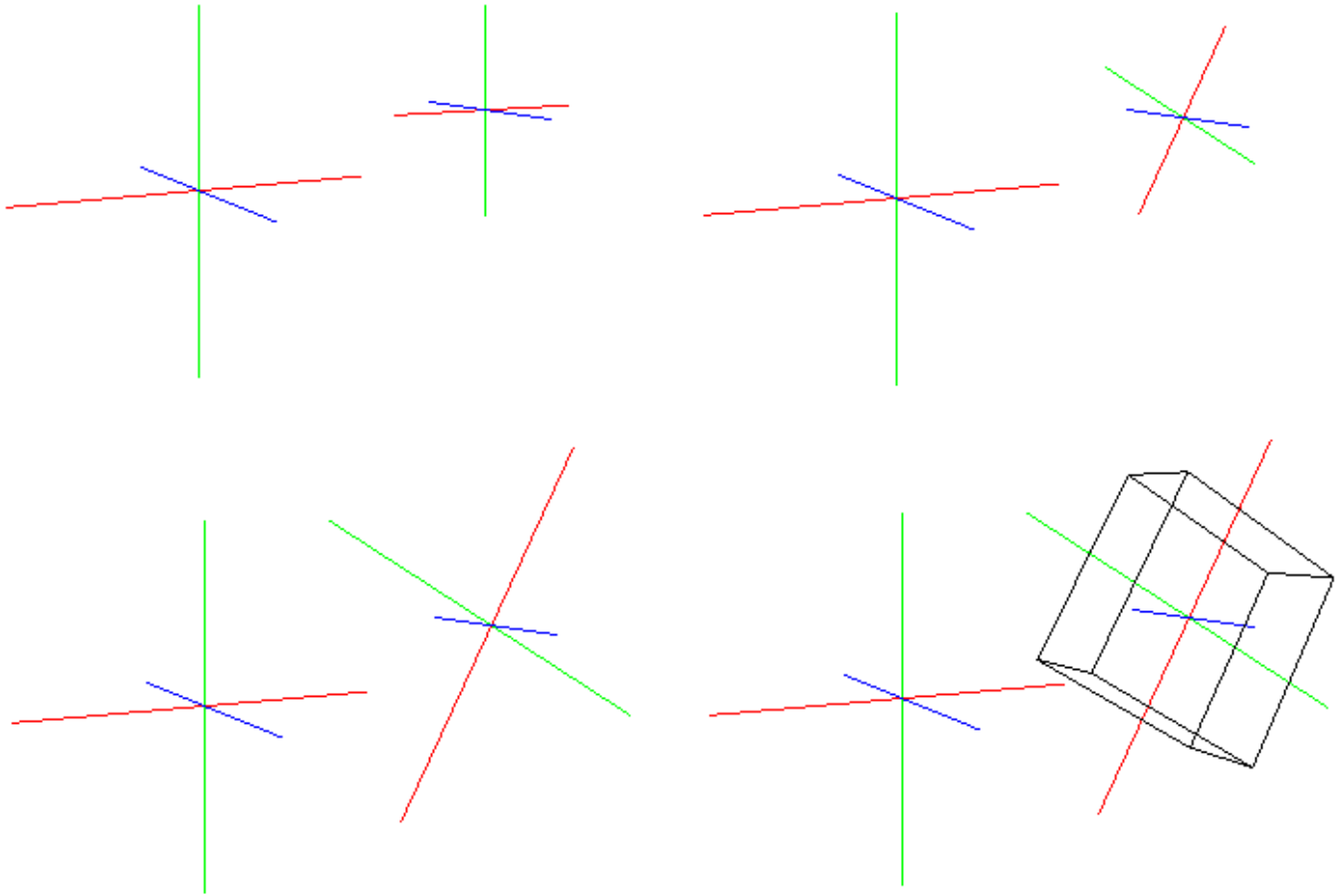
# Order of ransformations in OpenGL cont.

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

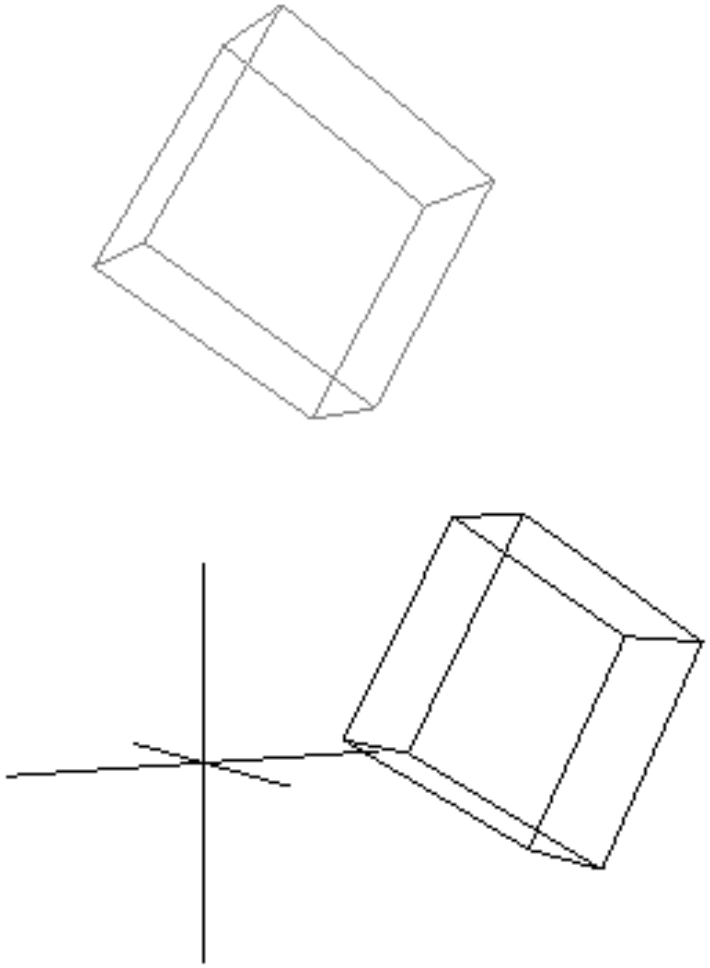
$$T_{(4,0,-3)} \longrightarrow R_{(0,0,1),\frac{\pi}{3}} \longrightarrow S_{(2,2,1)}. \tag{2}$$

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

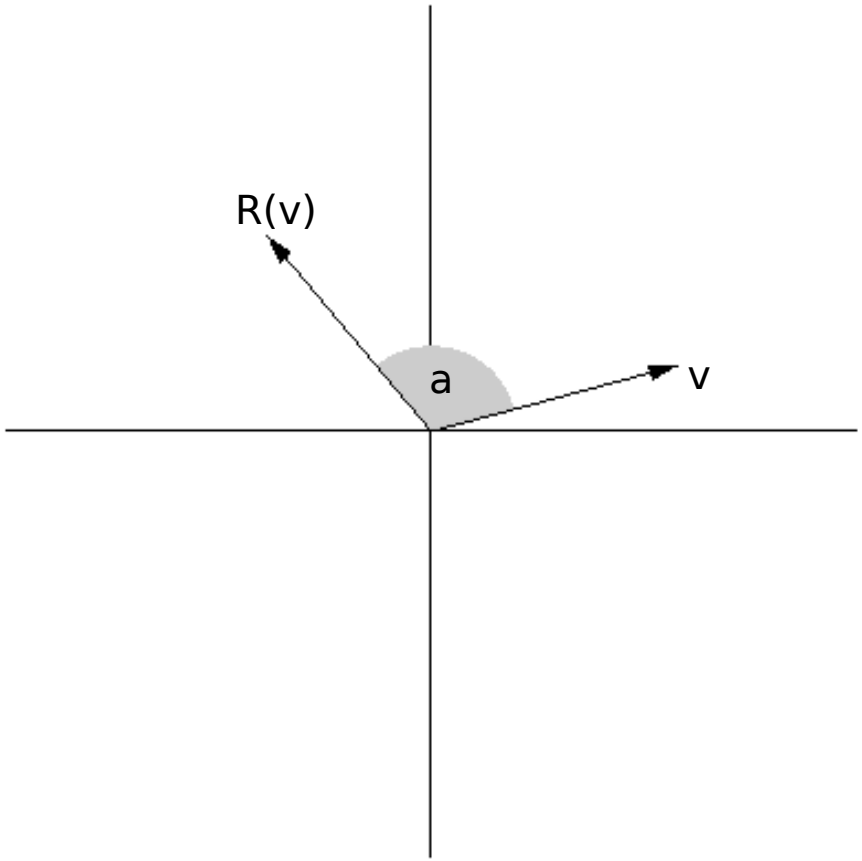
$$A = \begin{pmatrix} 2 & 3 \\ -1 & 1 \end{pmatrix}.$$

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

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



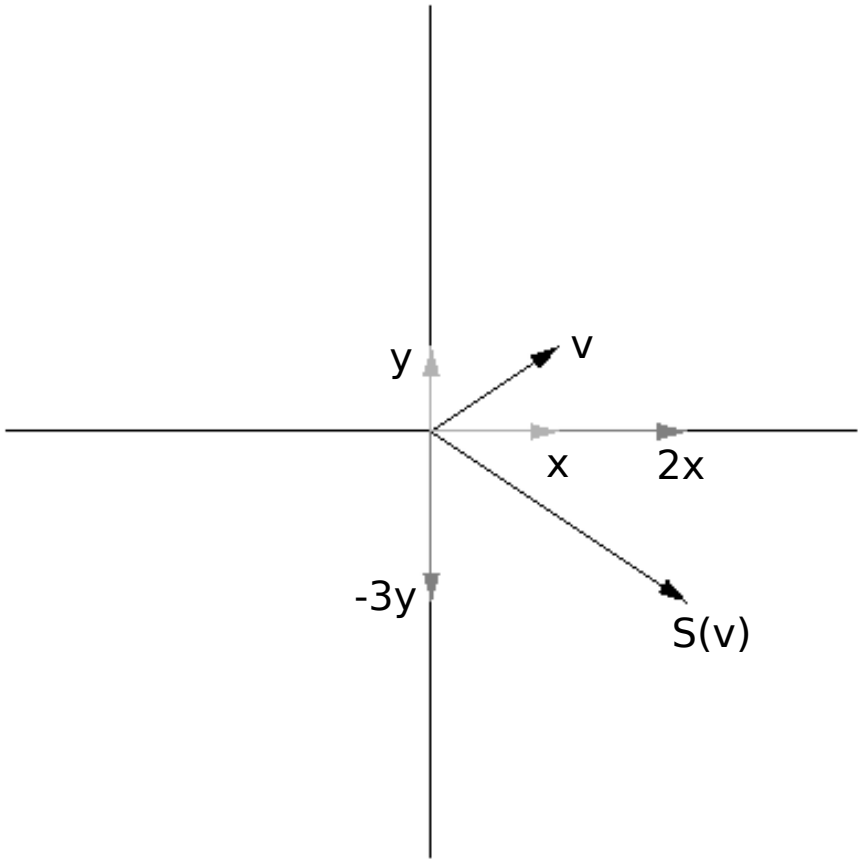
We can scale a figure in each direction by a different factor. Let $s_x$ and $s_y$ represent scaling factors along $x$ and $y$–axis, respectively.
Scaling $S$ can be described as follows

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

$S$ is a linear transformation and in matrix notation

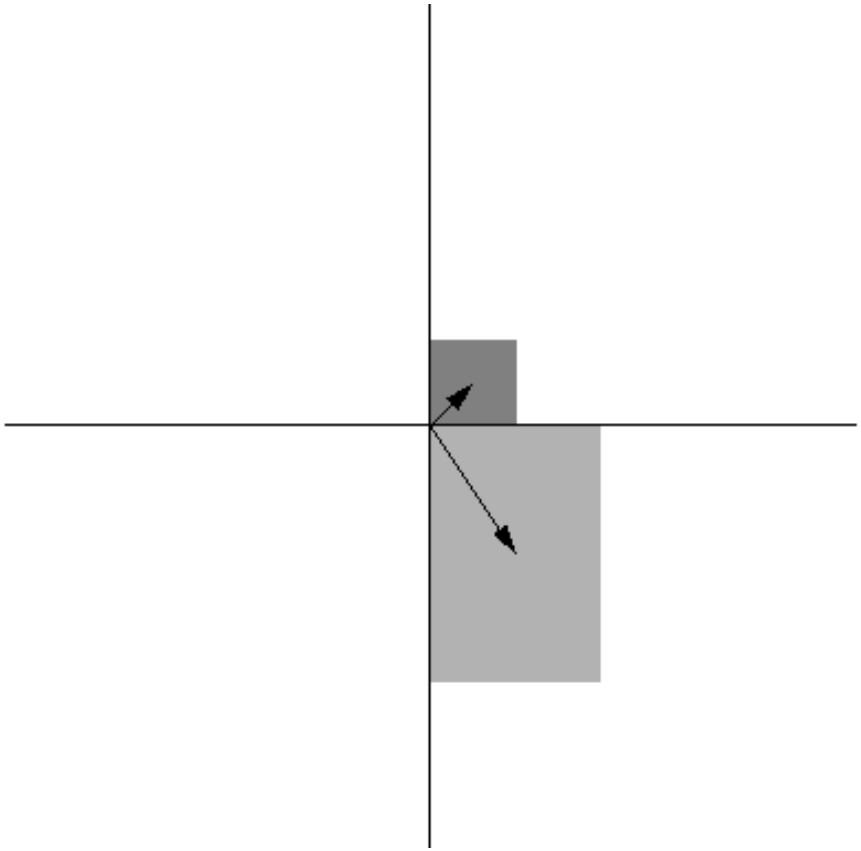
$$S \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

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

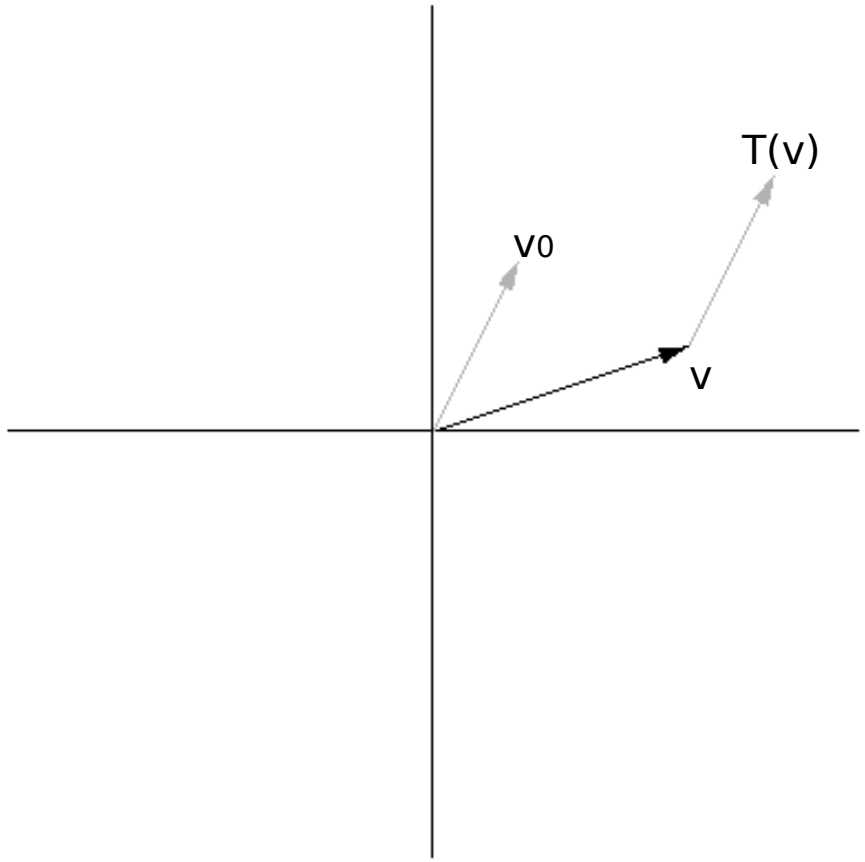
$$A \cdot B \quad (= A \circ B).$$

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:

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

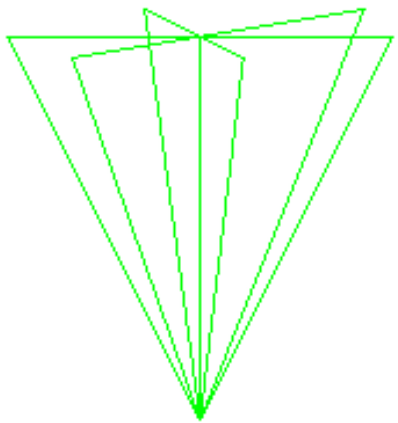
$$I \cdot T \cdot R = \begin{pmatrix} \cos a & 0 & -\sin a & 0 \\ 0 & 1 & 0 & 0 \\ \sin a & 0 & \cos a & 0 \\ 2 & 1 & -7 & 1 \end{pmatrix} \rightarrow$$

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

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

# OpenGl transformation – outcome of the example

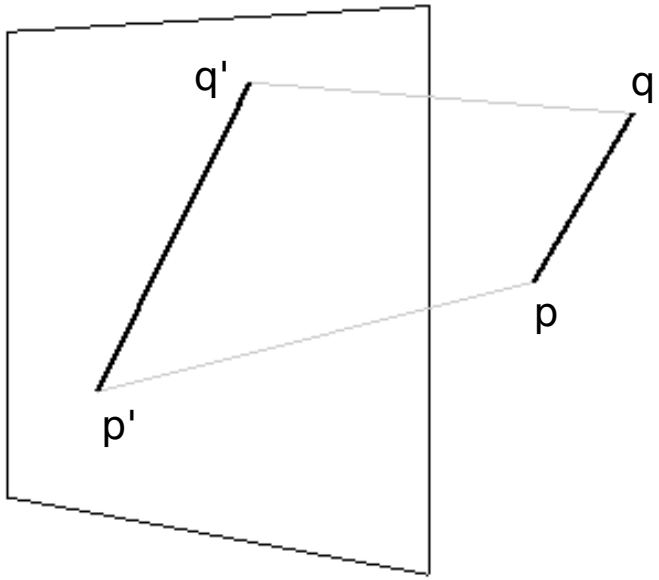
We show the outcome for `alpha` taking values 30.0, 90.0, 150.0, 210.0, 270.0 and 330.0 degrees. Notice that if we put at the end of `display()` function, for example, `alpha+=1.0`; we get animation of rotating triangle with the values of the angle increasing by 1.0 starting from alpha equal 30.0.



# Projections

Since the image on a screen is 2–dimensional, to view 3–dimensional objects we need to transfer (project) them to 2–dimensional ones. We distinguish two types of projections: **orthographic** and **perspective**.

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



# Orthographic projection

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.

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

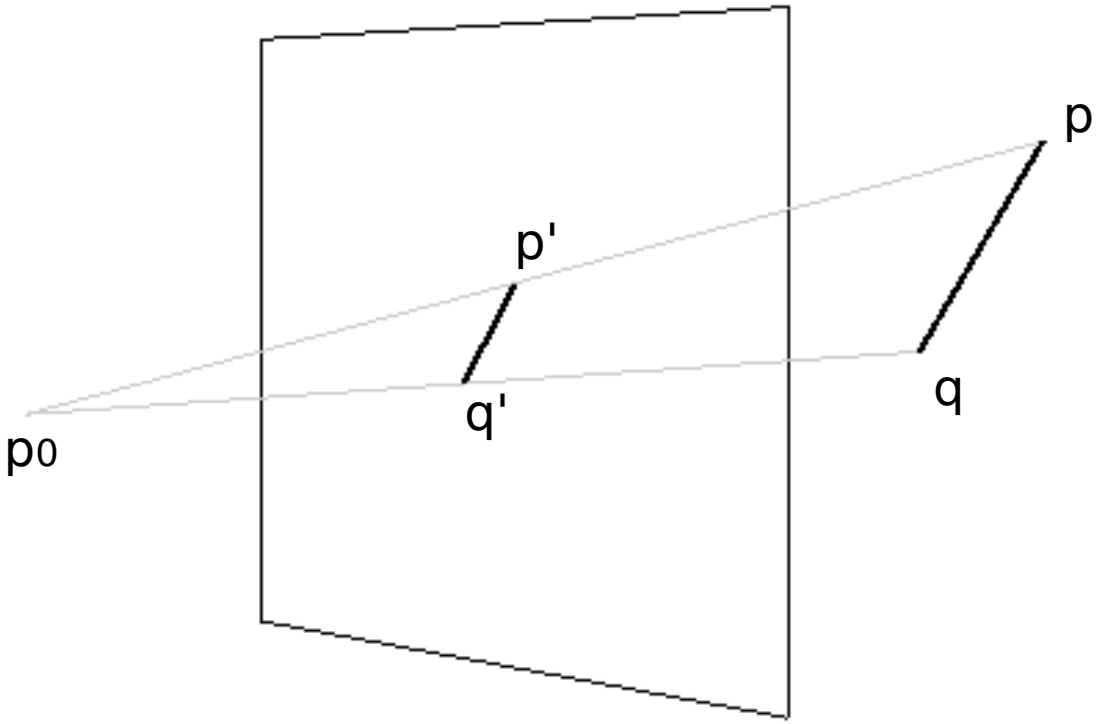
$$(x, y, z) \mapsto (x, y, 0).$$

Hence in the homogeneous coordinates this projection has the form

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

# Perspective projection

**Perspective projection** is defined as follows: Let $p_0$ be the center of projection (point where the camera is situated) and $\Pi$ 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

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

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

# Projections in OpenGL

The projection matrices in OpenGL differ a little, but first let us introduce the commands, which cause each type of projection to be executed:

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.

# How to set projections in OpenGL

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);
```

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

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

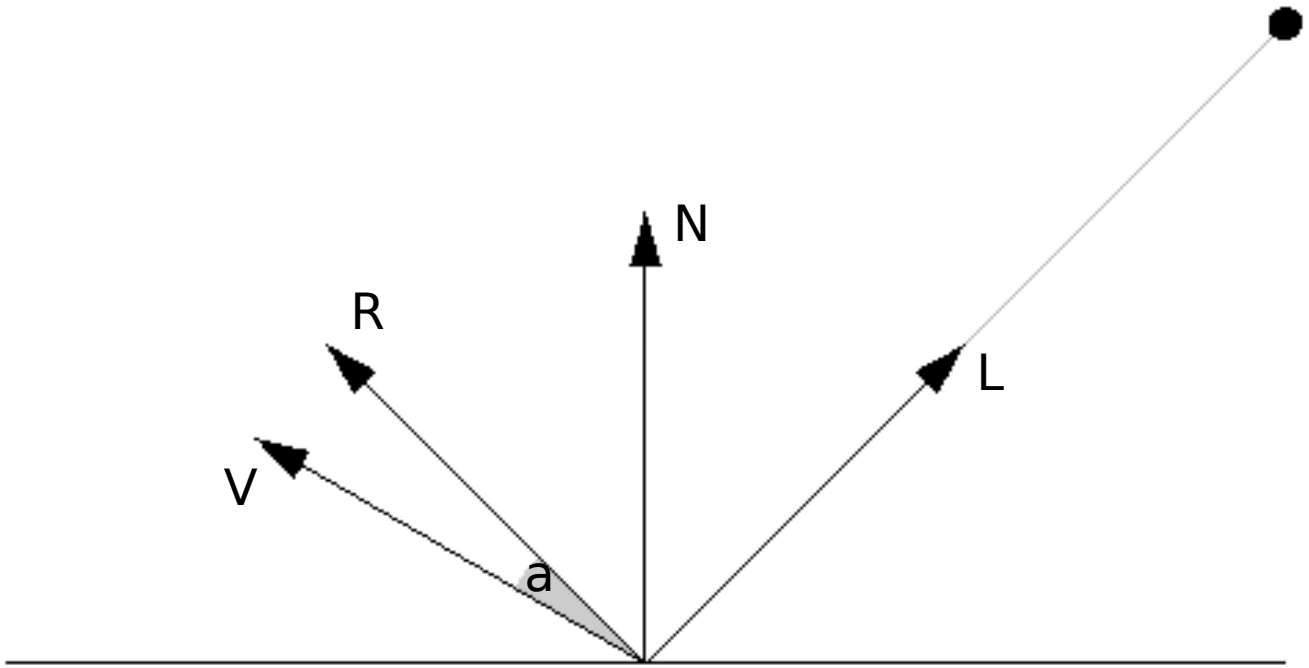
where $I_d$ is the brightness of SL and $t$ is the angle between the direction of the light and the vector normal to the surface of an object. If we denote by $L$ the unit vector of direction of the light and by $N$ the unit normal (outer) vector to the surface, above formula can be rewritten as

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

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

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

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

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

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

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

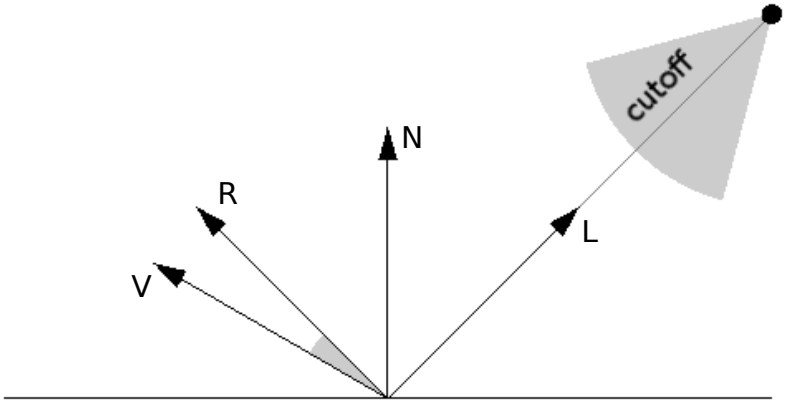
# The practical use of above formula

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 $e_{\text{spot}}$.

# 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 $f_{\mathrm{att}}$ – 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

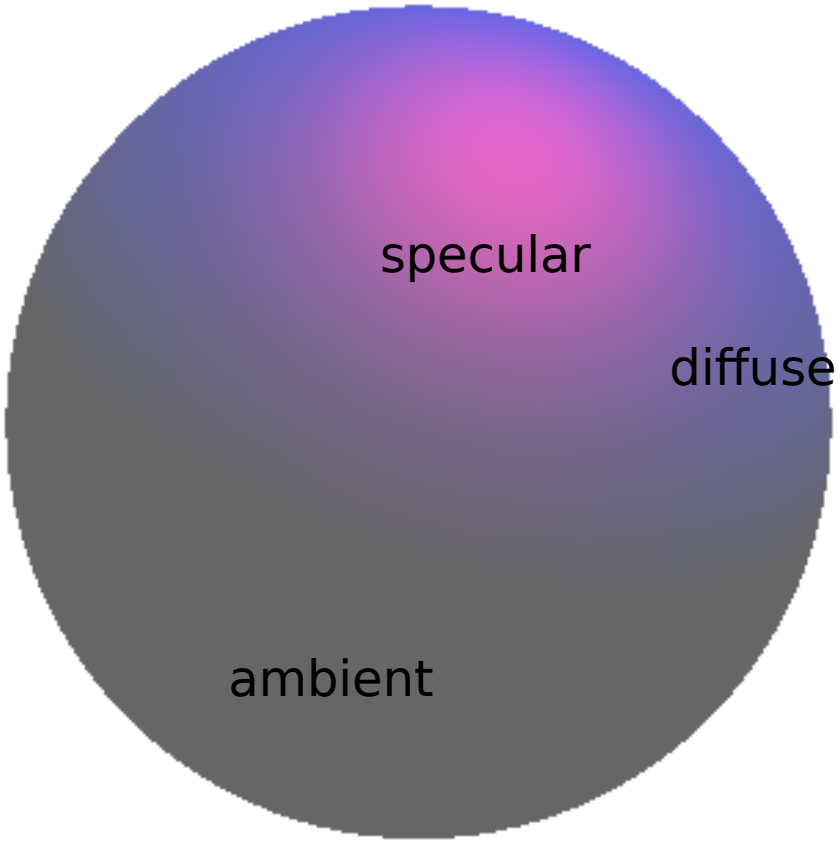
- `GL_SPOT_CUTOFF` – defines the cutoff angle, i.e., half of the light dispersal angle (if the value is in the range $(0, 90)$). The other possible value is 180, which corresponds to the light going in all directions from the source.
- `GL_SPOT_EXPONENT` – allows to compute $e_{\text{spot}}$ in (4) as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

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



# Part II - Basics of Computer Graphics

- vector and raster graphics
- raster graphics algorithms - drawing a line and a circle, filling areas
- color and color models - RBG, CMY and HSV,
- hidden surface removal

## Raster graphics

An image on a monitor or on a sheet of paper in copywriter consists of finite rectangular elements called **pixels**. Each pixel can be of a different color. We often call a displayed image in raster graphics a **bitmap**. Raster monitors, which are now widely used, 'remember' displayed **primitives**, that is simple geometric objects as lines, circles etc. made of pixels. A bitmap is thus a set of horizontal and vertical lines of pixels, which retains in the memory as a table of pixels of a screen. The advantage of raster graphics is that each image takes the same amount of memory (use of memory depends only on resolution of a monitor, that is the number of pixels) and can be easily displayed.

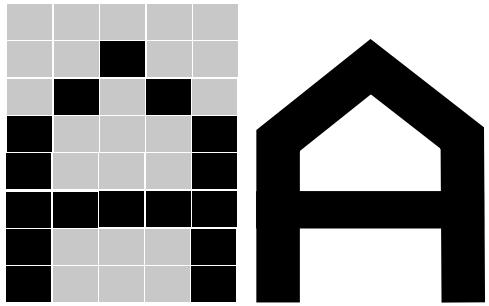


Figure: Letter A as a raster and vector image.

## Vector graphics

In vector graphics objects are made of arbitrary small lines (points), which may be placed everywhere. Therefore, while scaling and rotating objects we don't loose the quality of the displayed objects. In vector monitors the stream of electrons goes only to the place where the object is displayed, whereas in raster monitors, as we said before, the stream of electrones runs through the whole screen. One of the drawbacks of vector graphics is that it is hard to transfer vector object to raster ones.

We can distinguish four graphics file formats:

1. **Image file formats** related to raster graphics. The most common image formats include GIF, JPEG, PNG, TIFF and BMP.
2. **Vector graphics file format** Examples of vector formats are DXF used by Autodesk Company in its applications (AutoCad) and SVG made for WWW.
3. **Metafile formats** Can include both raster and vector information. Examples are WMF and EMF used in MS Windows.
4. **Page description language** refers to formats used to describe the layout of a printed page containing text, objects and images. Examples are PostScript and PDF created by Adobe.

# Drawing a line

In raster graphics even drawing a line or a circle is not easy, because it is impossible to draw a straight line or round circle with finitely many points (pixels). Thus we need algorithms to draw these geometric objects (primitives).

We want to draw a line as straight and as close to given one as it is possible. Let $P_0 = (x_0, y_0)$ and $P_p = (x_p, y_p)$ denote the begining and the end of a line $l$, and assume that the coordinates of the point $P_0$ are integer valued. As a first pixel we take $P_0$. The next one can be chosen from the 8 pixels that are nearby.

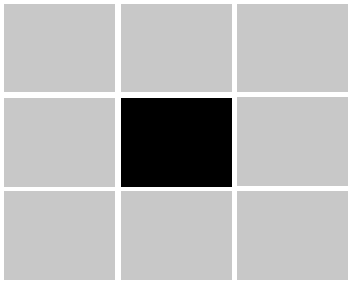


Figure: 8 nearby pixels.

# Drawing a line

We choose the one closest to the line and continue the process. The easiest strategy to do it is as follows. It is called **DDA** algorithm. Firstly, we may assume for simplicity, that $x_0 < x_p$ and that the slope $m$ of the line satisfies $|m| \leq 1$. Put

$$\Delta x = x_p - x_0, \quad \Delta y = y_p - y_0.$$

Then $m = \frac{\Delta y}{\Delta x}$ and the line $l$ is described by an equation

$$y = \frac{\Delta y}{\Delta x}(x - x_0) + y_0.$$

As we said before we start from the pixel $P_0 = (x_0, y_0)$. We increase the value of $x_0$ by 1, $x_1 = x_0 + 1$. Then the point $Q_1 = (x_1, y_0 + m)$ lies on $l$. We choose $P_1$ as the pixel closest to $Q_1$. Hence $P_1 = (x_1, \mathrm{Round}(y_0 + m))$, where $\mathrm{Round}$ is the function which rounds the value. In general, if $P_i = (x_i, y_i)$ is given, then $P_{i+1} = (x_i + 1, \mathrm{Round}(y_i + m))$.

# Drawing a line

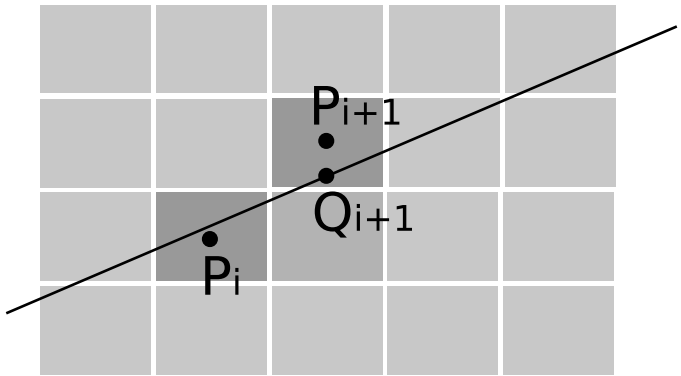



Figure: DDA algorithm.

The above algorithm is not perfect. Parameters $y_i$ and $m$ are real numbers and the function $\mathrm{Round}$ requires time consuming computations. In 1963, Bresenham proposed an algorithm based on the same observation as above, which uses only integer values.

# Drawing a line

**Bresenham algorithm** is the following. The begininig is the same, however for simplicity we assume $0 < m \leq 1$. We start from the pixel $P_0 = (x_0, y_0)$. Assume we have picked the point $P_i = (x_i, y_i)$ and we want to find the next one $P_{i+1} = (x_{i+1}, y_{i+1}$. Since the slope of the line satisfies $0 < m \leq 1$, there are two possibilities of choosing the point $P_{i+1}$: $S_{i+1} = (x_i + 1, y_i)$ or $T_{i+1} = (x_i + 1, y_i + 1)$.

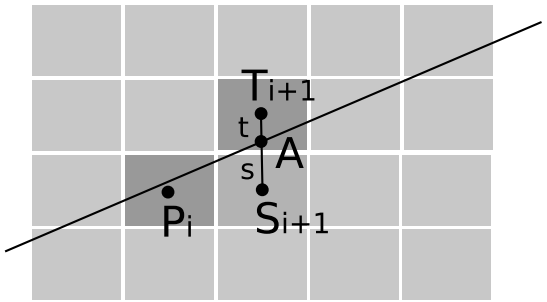



Figure: Finding $P_{i+1}$. Here $P_{i+1} = T_{i+1}$.

Let $A$ be the intersection of $l$ and the segment $\overline{T_{i+1}S_{i+1}}$. Then the $x$–coordinate of $A$ is $x_A = x_i + 1$. Hence the $y$–coordinate is $y_A = \frac{\Delta y}{\Delta x}(x_i + 1 - x_0) + y_0$.

# Drawing a line

Therefore, the length $t$ of the segment $\overline{T_{i+1}A}$ and the length $s$ of the segment $\overline{AS_{i+1}}$ are respectively

$$t = y_i + 1 - y_A = y_i + 1 - y_0 - \frac{\Delta y}{\Delta x}(x_i + 1 - x_0),$$

$$s = y_A - y_i = \frac{\Delta y}{\Delta x}(x_i + 1 - x_0) - (y_i - y_0).$$

Hence

$$d_i = \Delta x(s - t) = -2\Delta x(y_i - y_0) + 2\Delta y(x_i - x_0) + 2\Delta y - \Delta x. \tag{5}$$

If $d_i \geq 0$ $(t \leq s)$, then $P_{i+1} = T_{i+1}$. If $d_i < 0$ $(t > s)$, then $P_{i+1} = S_{i+1}$. To improve the recurrence, let us make few observations. Writing (5) for $d_{i+1}$ we have

$$d_{i+1} = 2\Delta x(y_{i+1} - y_0) + 2\Delta y(x_{i+1} - x_0) + 2\Delta y - \Delta x. \tag{6}$$

Thus, since $x_{i+1} - x_i = 1$

$$d_{i+1} - d_i = -2\Delta x(y_{i+1} - y_i) + 2\Delta y(x_{i+1} - x_i) = -2\Delta x(y_{i+1} - y_i) + 2\Delta y.$$

Finally

$$d_{i+1} = d_i - 2\Delta x(y_{i+1} - y_i) + 2\Delta y.$$

# Drawing a line

Going back to choosing between $T_{i+1}$ and $S_{i+1}$, we have

1. $\underline{d_i \geq 0}$. Then $P_{i+1} = T_{i+1}$, so $y_{i+1} = y_i$ and $d_{i+i} = d_i - 2\Delta x + 2\Delta y$.
2. $\underline{d_i < 0}$. Then $P_{i+1} = S_{i+1}$, so $y_{i+1} = y_i$ and $d_{i+1} = d_i + 2\Delta y$.

We can now write the pseudo-code for this algorithm, we write $dx$ and $dy$ instead of $\Delta x$ and $\Delta y$:

```
constants:      P[0]=(x[0],y[0]), dx=x[p]-x[0], dy=y[p]-y[0]
                d0=2*dy-dx
algorithm:      for (i=0) to (x[p]-x[0]-1) do {
                x[i+1]=x[i]
                if (d[i] >= 0) then {
                        d[i+1]=d[i]-2*dx+2*dy
                        y[i+1]=y[i]+1
                        }
                else {
                 d[i+1]=d[i]+2*dy
                 y[i+1]=y[i]
                 }
                P[i+1]=(x[i+1],y[i+1])
                }
```

# Drawing a circle

When we want to draw a circle, there is another one aspect which doesn't play the role in line drawing. Namely, we need to take into account the shape of a pixels. If the pixels are rectangles but not squares, we can obtain an ellipse instead of a circle. Therefore, we define the **aspect** of a graphics device as a ratio of the length and the height of a pixel. Assume the aspect is a rational number $a = p/q$. We should distinguish between Cartesian coordinate system (X Y) and coordinate system of pixels (x y). If we move up $y$ and right $x$ in Pixel coordinate system, then we move up $Y = y$ and right $X = ax$ in Cartesian coordinate system.

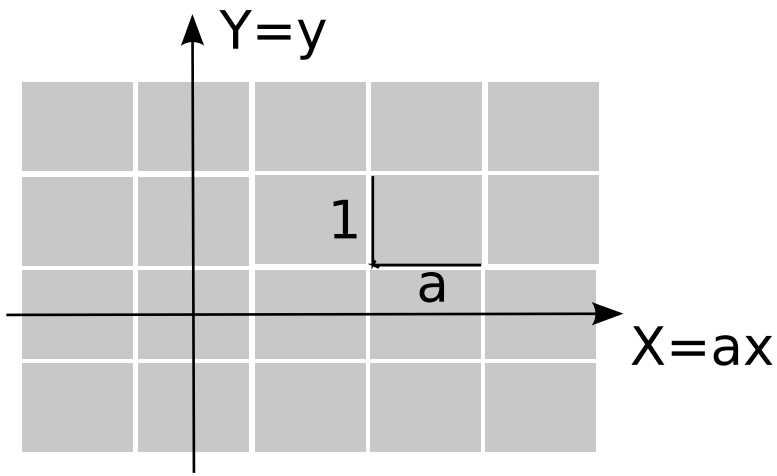



Figure: Transfer from Cartesian to Pixel coordinate system.

# Drawing a circle

Assume we want to draw a circle $C$ of radius $R$ centered at $(0,0)$. Then $C$ is given by $X^2 + Y^2 - R^2 = 0$ or equivalently $(ax)^2 + y^2 - R^2 = 0$. Since $a = p/q$ the circle $C$ is given by an implicit equation

$$f(x,y) = p^2x^2 + q^2y^2 - q^2R^2 = 0. \tag{7}$$

Then the area inside the circle is given by $f(x,y) < 0$ and the area outside the circle by $f(x,y) > 0$.
Due to the symmetry we consider only the part of a circle $f(x,y) = 0$, where $x, y > 0$. Let the starting point be $P_0 = (0,R)$. Assume we have found the pixel $P_i = (x_i, y_i)$ and we seek for $P_{i+1} = (x_{i+1}, y_{i+1})$. It is easy to see that the choice restricts to three points $S_{i+1} = (x_i + 1, y_i - 1)$, $T_{i+1} = (x_i + 1, y_i)$, $U_{i+1} = (x_i, y_i - 1)$.

# Drawing a circle

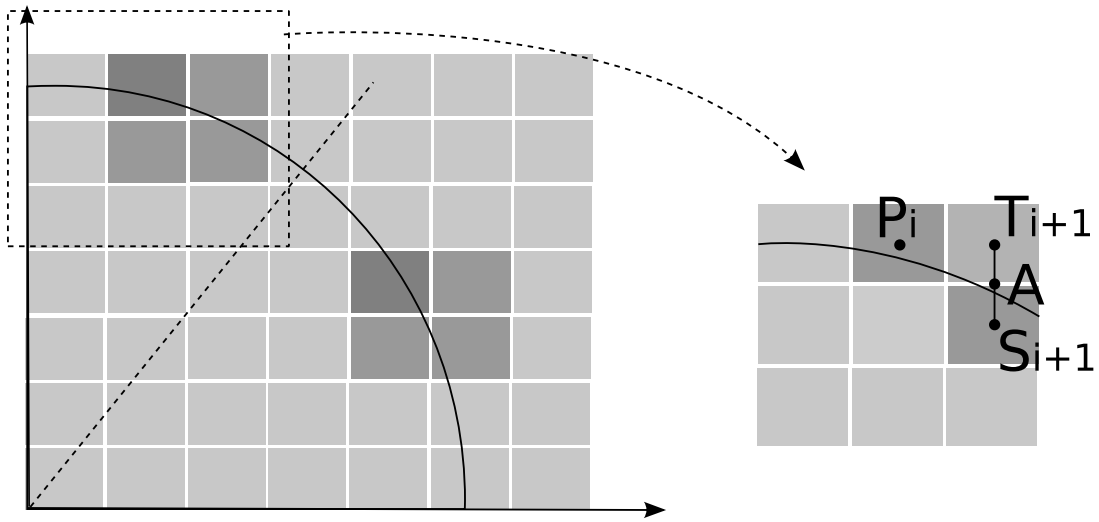



Figure: Possibilities of choosing $P_{i+1}$.

The tangent vector $v(x, y)$ to a circle at a point $(x, y)$ is by implicit function theorem

$$v(x, y) = (1, \frac{f'_x}{f'_y}(x, y)) = (1, \frac{-p^2 x}{q^2 y}).$$

Thus if the slope of $v(x, y)$ is $< -1$, which means $p^2 x > q^2 y$, we choose between $T_{i+1}$ and $S_{i+1}$, whereas if the slope of $v(x, y)$ is $> -1$, which means $p^2 x < q^2 y$, we choose between $U_{i+1}$ and $S_{i+1}$.

# Drawing a circle

We now concentrate on the part of a circle $p^2x > q^2y$ leaving the second case to the reader. We proceed in the similar way as in the case of a line. Let $A$ be the middle of a segment $\overline{T_{i+1}S_{i+1}}$. If $A$ is outside the circle, we choose $S_{i+1}$. If $A$ is inside the circle, we choose $T_{i+1}$. If $A$ is on a circle, we can take any of these two points (for our purposes we choose $T_{i+1}$). We have $A = (x_i + 1, y_i - \frac{1}{2})$. Then

$$fa_i = f(A) = p^2(x_i + 1)^2 + q^2(y_i - \frac{1}{2})^2 - q^2R^2.$$

To improve the recurrence let us compute

$$fa_{i+1} - fa_i = p^2(x_{i+1} - x_i)(x_{i+1} + x_i + 2) + q^2(y_{i+1} - y_i)(y_{i+1} + y_i - 1).$$

For the starting point $P_0 = (0, R)$ we have

$$fa_0 = p^2 - q^2R + \frac{R^2}{4}.$$

To avoid division, we multiply $fa_i$ by a factor 4.

# Drawing a circle

Concluding we have

1. If $fa_i \geq 0$, then $P_{i+1} = S_{i+1}$, so $x_{i+1} = x_i + 1$, $y_{i+1} = y_i - 1$. Hence $fa_{i+1} = fa_i + 4p^2(2x_{i+1} + 1) - 8q^2 y_{i+1}$.
2. If $fa_i < 0$, then $P_{i+1} = T_{i+1}$, so $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$. Hence $fa_{i+1} = fa_i + 4p^2(2x_{i+1} + 1)$.

We can now write a pseudo-code for this algorithm:

```
constants:      x=0, y=R, fa=4*p*p-4*q*q*R+R*R
algorithm:      while (p*p*x<q*q*y) do
                {
                P=(x,y)
                x=x+1
        if (fa>=0) then
                {
                y=y-1
                fa=fa+4*p*p*(2*x+1)
                }
        else
                {
                fa=fa+4*p*p*(2*x+1)-8*q*q*y
                }
```

# Filling areas

Consider the following problem:

Given the boundary of some area, fill this area in.

To start dealing with this task we need some theoretical background. Let $A$ be the set of pixels. We say that $A$ is **connected** if any two pixels from $A$ can be joined by neighbouring pixels from $A$. If the neighbouring pixels are the only these which lie one above another or one on the left (right) of another we speak about 4-**connectedness**, whereas if the neighbouring pixels are these which can also lie diagonal to each other we speak about 8–**connectedness**.

# Flood fill algorithm

Assume the boundary of area is 8–connected and that the interior is 4–connected. Let the boundary consist of black pixels and we fill the interior with gray pixels. Assume we have chosen an interior pixel. Then we check if any of four neighbouring pixels is black. If not we color it gray and continue the process for this pixel.

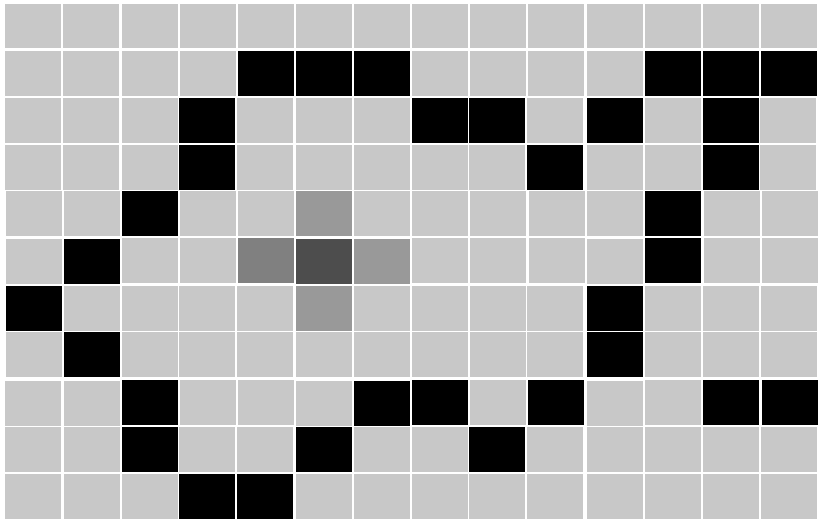


Figure: Flood Fill algorithm.

# Flood fill algorithm

The function **FloodFill** is thus of the form

```
FloodFill(x,y)
{
        if (color(x,y)<>black and color(x,y)<>gray) then
        {
                setcolor(x,y,gray)
                FloodFill(x,y-1)
                FloodFill(x,y+1)
                FloodFill(x-1,y)
                FloodFill(x+1,y)
        }
}
```

Flood Fill algorithm is short and simple, however its realization causes problems, mainly because of the recurrence, which consumes a lot of memory. Moreover, the color of a pixel is often checked few times.

# Filling polygons

We now concentrate on filling polygons. We do not require the polygon to be convex. The algorithm of filling the polygon can be described as follows:
For each horizontal line $l$

1. find all points $x_1, \ldots, x_p$ of intersection of $l$ with the polygon. $p$ is in general an even number $p = 2k$.
2. sort these points $x_1 < \ldots < x_p$.
3. using Bresenham algorithm draw the segments $\overline{x_i x_{i+1}}$ for $i = 1, \ldots, k$.

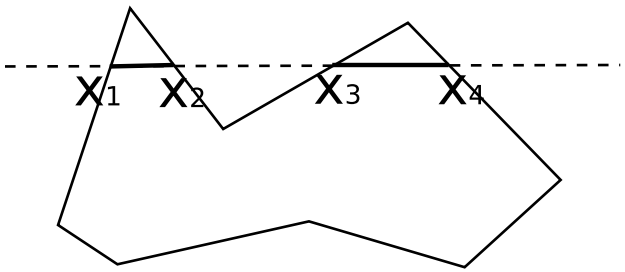



Figure: Scanning with horizontal lines.

# Filling polygons

The third step requires considering four problems:

(a) How to decide for the intersection point $x_i$ of non–integer $x$–value which pixel is an interior one?

(b) What to do in a special case of integer $x$–valued intersection?

(c) What to do in the case (b) for vertices?

(d) What to do in the case (b) when vertices define horizontal edge?

Before we ilustrate and give solutions to above problems we must explain why to consider such problems. Figure below shows that extremal pixels, that is end points of segments $\overline{x_i x_{i+1}}$ can be outside the polygon. This is because the Bresenham algorithm do not see if the point is an interior or exterior point, the algorythm chooses the point lying closest to the line. We do not want to draw exterior points, since if the edge of a polygon is at the same time the edge of neighbouring polygon, it could lead to interference of areas of neighbouring polygons.

# Filling polygons

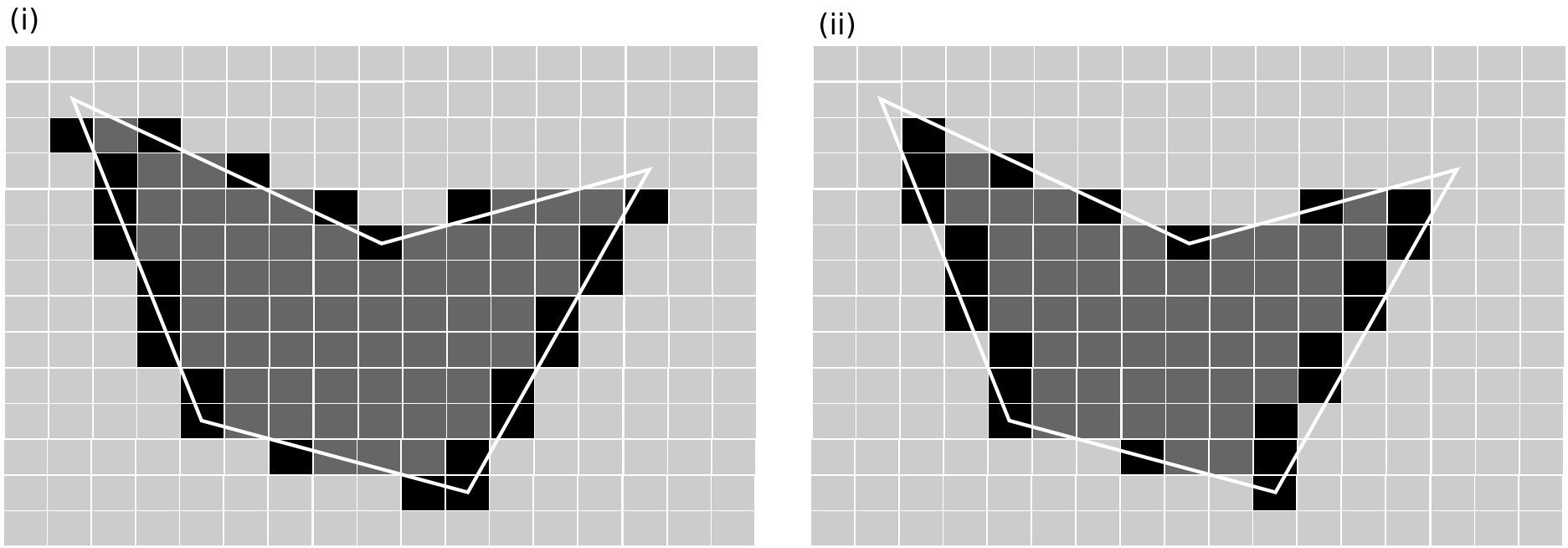



Figure: (i) Extremal points determined by Bresenham algorithm, (ii) extremal points lying inside the polygon.

# Filling polygons

As for the question (a), if we approach an intersection point going from the left and we are inside the polygon we round the $x$–coordinate down and if we are outside the polygon we round it up. In the case (b), if the point $x_i$ of the segment $\overline{x_i x_{i+1}}$ has integer $x$–coordinate, we treat it as an inside pixel, if the pixel $x_{i+1}$ of this segment has integer $x$–coordinate, we treat it is as outside pixel.

Solution of the problem (b) is a general convention for choosing interior and boundary points to avoid doubling pixels for neighbouring primitives. The convention says that a boundary pixel in not treated as a part of a primitive if the half–plane containing the primitive defined by the edge of a primitive and containing that pixel lies below or on the left from the edge.

# Filling polygons

We now move to the question (c). For the edge segment let $y_{\max}$ and $y_{\min}$ be the maximal and minimal value of $y$–coordinate of vertices of the edge. Then we include the vertex $y_{\min}$ and do not include the vertex $y_{\max}$. Vertex $y_{\max}$ is drawn if it is a $y_{\min}$ vertex for another edge.

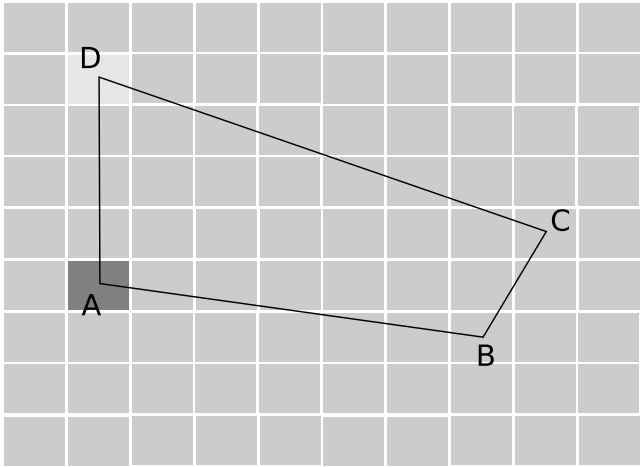



Figure: Vertex $A$ of the edge $\overline{AD}$ is included, vertex $D$ not.

In the case (d) of horizontal edges, bottom edges are drawn, top edges are omitted (this follows by solution to the problem (c)).

# Color in Computer Graphics

**Light** is an electromagnetic radiation, which is visible for a human eye. It is a radiation of wavelength around 400–700 nm (nanometres). The wavelength 400 nm responds to violet color, whereas wavelength 700 nm to red color. Decent man can distinguish about 150 colors. There are three attributes that affect human color impression: **hue**, **lightness** (or brightness) and **saturation** (or colorfulness). Hue is described by the wavelength. Lightness is the difference between a color against gray and saturation is the difference of a color against its own brightness. The most saturated colors are green, red, blue and yellow. Considering lightness and saturation we can distinguish between 400000 colors.

By above short characteristic, we see that color modeling is not easy and the same image on a screen can be interpreted differently depending on the light in the room where the monitor is situated.

# Color in Computer Graphics

Now we concentrate on the color in computer graphics. Our main source of light - the Sun - radiates electromagnetic radiation of all possible visible wavelength, which gives white color. We can obtain white color by mixing other colors in appropriate proportion (for example mixing red, green and blue colors in proportion 26:66:8). Mixing few colors, we can obtain a vast scale of different colors. These main colors, which together give white color, are called **primary**. There is no perfect choice of primary colors, since, as it will be soon remarked, any finite number of primary colors can't give all spectrum of colors. However, in 1931 International Commission on Illumination defined, so called, CIA XYZ color space, i.e. the standard for primary colors. According to this standard we distinguish three primary colors. Denoting by $A$, $B$, $C$ the amount of each of primary color in resulting color, we define **chromatic coordinates** of this color by

$$x = \frac{A}{A+B+C}, \quad y = \frac{B}{A+B+C}, z = \frac{C}{A+B+C}.$$

Since $x + y + z = 1$, any of two values of $x, y, z$ define remaining one ($x$, $y$ imply $z = 1 - x - y$). To a given color we define a **complementary** color, that is the color, which with given one gives primary colors.

Using chromaticity diagram we can measure saturation of a color, find the wavelength of the color, complementary colors etc.

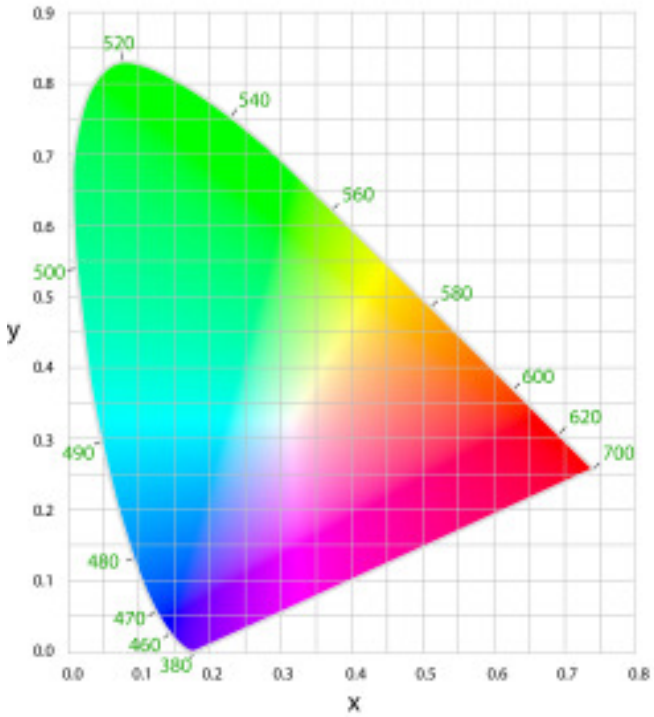



Figure: Chromaticity diagram.

Any color $C$ made of primary colors $C_1, C_2, C_3$ is described by equality $C = xC_1 + yC_2 + zC_3$ in chromaticity diagram, hence $C$ lies inside the triangle $\Delta C_1 C_2 C_3$. In general, any color $C$ made of primary colors $C_1, \ldots, C_n$ lies inside the convex hull of primary colors. Therefore, we see that we can't derive all colors using finite number of primary colors.

We will now describe three color models (spaces) which are widely used in computer graphics:

- RGB model,
- CMY model,
- HSV model.

# RGB model

In this model as a primary colors we choose red (R), green (G) and blue (B) and each color corresponds to a point in the unit cube.

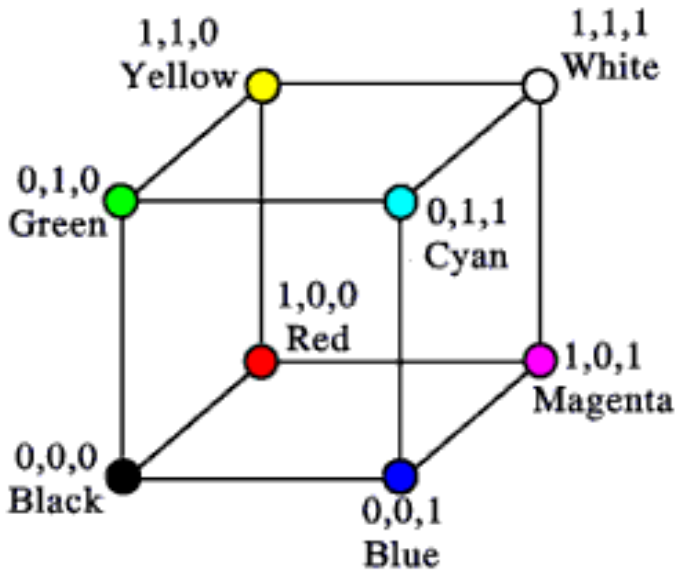



Figure: RGB model.

Origin $(0, 0, 0)$ corresponds to white color, $(1, 0, 0)$ to red, $(0, 1, 0)$ to green, $(0, 0, 1)$ to blue. The main diagonal consists of gray colors, from white to black.

# RGB model

Given two colors $(r_1, g_1, b_1)$ and $(r_1, g_2, b_2)$ in RGB model we can add these colors to obtain $(r, g, b)$ by the rule

$$r = \min(r_1 + r_2, 1), \quad g = \min(g_1 + g_2, 1), \quad b = \min(b_1 + b_2, 1).$$

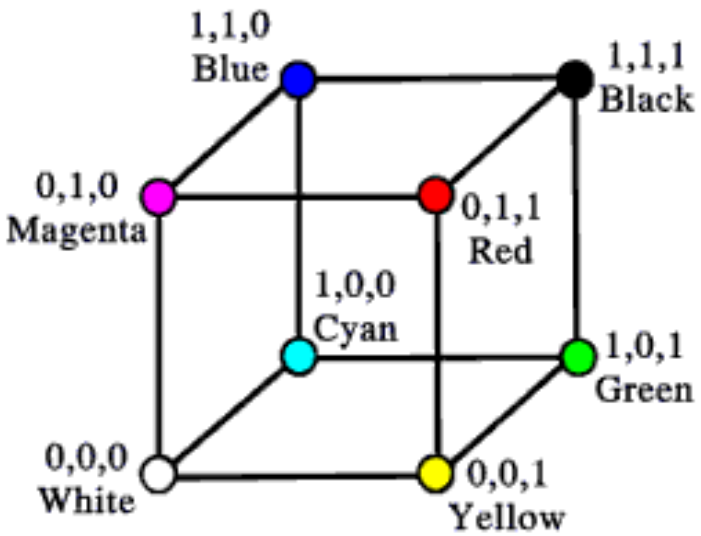
Since $(1, 1, 1)$ is a representation of a white color, to get the color complementary to $(r, g, b)$ we compute $(1 - r, 1 - g, 1 - b)$. RGB is an additive model, that is addition of color defined above, agrees with reality.

# CMY model

This model is similar to RGB model but uses dfferent primary colors: cyan (C), magenta (M) and yellow (Y), the complementary colors to red, green and blue. Thus to obtain a color in CMY model it is sufficient to take this color in RGB model $(r, g, b)$ and compute

$$(c, m, y) = (1, 1, 1) - (r, g, b).$$

Therefore, in contrary to RGB model, CMY is a substractive model.



# HSV model

This model was introduced by Smith in 1979. Name HSV comes from three attributes of color, described above, hue (H), saturation (S) and value (V). HSV model is based on regular pyramid based on hexagon.

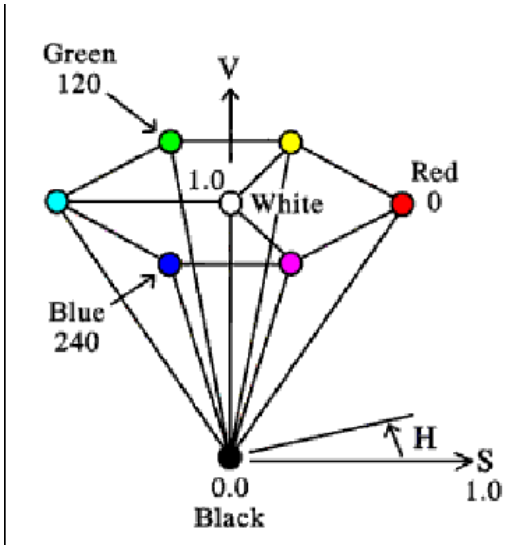



Figure: HSV model.

Color is measured by the angle $H$ around $V$ axis, saturation $S$ which measures the distance from $V$ axis and differs from 0 to 1 and the value $V$ which measures the lightness (distance from the white color, $V$ differs from 0 to 1). $H = 0^o$ corresponds to red color, $H = 120^o$ to green, $H = 240^o$ to blue.

# Hidden surface removal

Hidden surface removal concerns detecting which elements of objects are invisible for the observer (the camera). From the point of view of the camera, objects lying further are obscured by objects lying closer. However, this simple observation doesn't translate to simple algorithm. It is hard to measure precisely the distance from the camera to the object. We may imagine two objects obscuring each other.

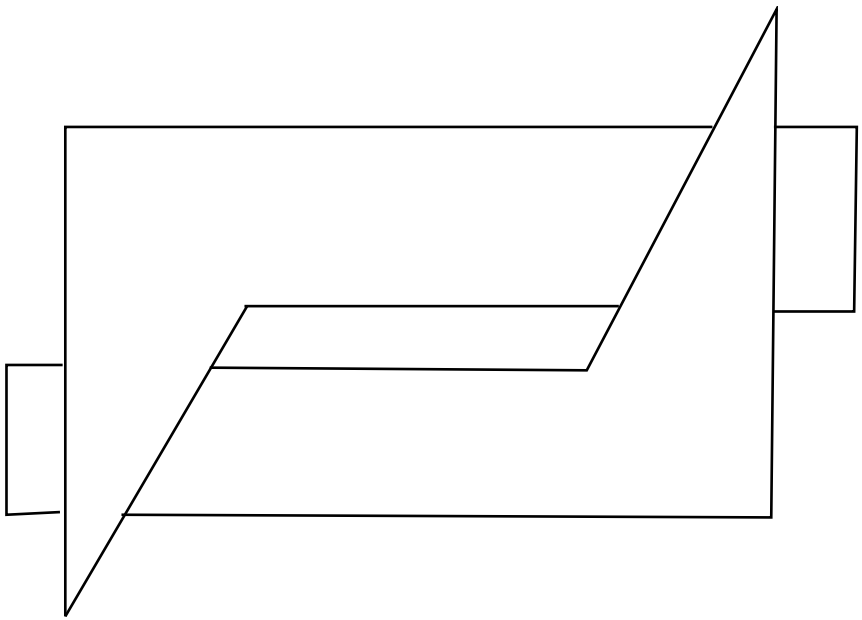


Figure: Two objects obscuring each other.

# Hidden surface removal

Above considerations led to creation of many algorithms of removal of hidden surfaces, but there are two fundamental algorithms or we should say two types of algorithms: **algorithm with image precision** and **algorithm with object precision**.
In the first case, we check which of $n$ objects is visible in each pixel of the image. The pseudocode for this algorithm is the following

```
for (each pixel of the image)
{
     1. find the closest object to the camera, which lies on the
        line joining the camera and given pixel;
     2. draw this pixel with the right color, the color of
        the object;
}
```

For each pixel we need to check all $n$ objects and find the closest one. Thus for $p$ pixels the complexity of this algorithm is $pn$.

# Hidden surface removal

In the second case, we compare all objects with themselves (that is we do $n^2$ comparisons) and choose these or parts of these objects, which are visible. We can describe it in the following way

```
for (each object)
{
     1. find parts of the object, which are not obscured by other
        objects;
     2. draw these parts;
}
```

Alhtought it seems that the second solution is better for $n < p$, however its implementation is not easy.
Now we describe roughly some hidden surface removal algorithms.

# Backface removal

Backface removal can be considered as a preprocessing step to speed up hidden surface removal. Backface removal checks every face of the object by finding the normal outward vector $N$ to the surface. If the vector $N$ point away from the viewer it means that this surface is invisible and can be removed.

This algorithm can be described as follows: take any two edges $k$ and $l$ of the face, which are counterclockwise oriented. Then $N$ is a vector product of $k$ and $l$, $N = k \times l$. Let $v$ denotes the vector joining any point of the face with the viewer. If the angle between $N$ and $v$ is grater or equal to $90^o$, then the face is invisible. In other words, using inner product, if $\langle N, v \rangle \leq 0$.

# Backface removal

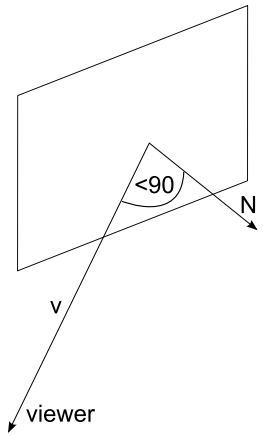



Figure: Backface removal.

There is one question: how to decide that the edges $k$ and $l$ are counterclockwise oriented? To avoid such problem it is convenient to make a representation of each object (see lecture 8), which contains such information. This information can be contained in the list of faces (3'). Each face is represented by a sequence of vertices. We put vertices in such order that first three vertices of the sequence $(X_1, X_2, X_3, \ldots)$ determine edges $k = \overline{X_1X_2}$ and $l = \overline{X_2X_3}$.

# Depthsort

This algorithm was proposed by Newell, Newell and Sancha in 1972 and can be described as follows: Paint each polygon in the scene in order, from the most distant to the nearest. We will try to explain more precisely this algorithm. For a object $T$ let $x$–**restriction** be the smallest interval $\langle x_{\min}, x_{\max} \rangle$ such that $x$–coordinates of all points of $T$ are in this interval. We define analogously $y$ and $z$–restriction.

The steps of depth sort are:

# Depthsort

1. Sort all faces of all objects in order from the furthest to the nearest with respect to $z_{\max}$ of each face.
2. Denote the furthest face by $S$. If $z$–restriction of $S$ and remaining faces are disjoint, then $S$ doesn't obscure remaining faces. Hence we draw $S$ and delete it from the list of faces. Otherwise we must check which of the remaining faces $T_1, \ldots, T_k$ that have nonempty $z$–restrictions with $S$ obscures $S$. For each $i = 1, \ldots, k$ we check if
   1. $x$–restriction of $T_i$ is disjoint with $x$–restriction of $S$
   2. $y$–restriction of $T_i$ is disjoint with $y$–restriction of $S$
   3. face $S$ lies on the side of the plane containing $T_i$ which is further from the viewer
   4. face $T_i$ lies on the side of the plane containing $S$ which is closer to the viewer
   5. projections of $S$ and $T_i$ on $xy$–plane are disjoint.
3. If any of the conditions (a)–(e) holds then $S$ doesn't obscure remaining faces and we draw $S$ and delete it from the list of faces. If some face $T_i$ doesn't satisfy any of these conditions, we switch $T_i$ and $S$ in the list of faces and repeat all steps for this new list.

Above strategy is not flawless. It is not effective for two or more faces which obscure themselves. In this situation we divide faces to smaller ones.

# Scan line

This is a generalization of algorithm of filling polygons. This is an algorithm with image precision.

First, we make a list of edges (LE) of all polygons but we do not consider horizontal edges. Therefore for each edge $l$ the end points $(x_0, y_0)$ and $(x_1, y_1)$ have different $y$–coordinetes. Assume $y_0 < y_1$. Then we sort edges with respect to $y_0$ from lowest to gratest value of $y_0$. In a group of edges with the same value of $y_0$ we sort edges with respect to $x_0$ from the greatest to lowest. Each edge in the list is represented by a sequence $(x_0, y_1, c, f_1, \ldots, f_k)$, where $c = \frac{x_1 - x_0}{y_1 - y_0}$ and $f_i$ are numbers representing faces containing this edge.
We also need the list of faces (LF), which beside the number repesenting the face, contains the following information:

1. coefficients of the plane $\pi$ containing the face, i.e. numbers A,B,C,D, where $\pi$ is of the form $Ax + By + Cz + D = 0$.
2. color of the face.
3. logical datatype in-out, which is primarly set to **false**.

# Scan line

We fill all pixels with background color. We scan projected image with horizontal lines from bottom to top and from left to right. Durign scaning we make the list of active edges (LAE). We will describe this algorithm on an example.

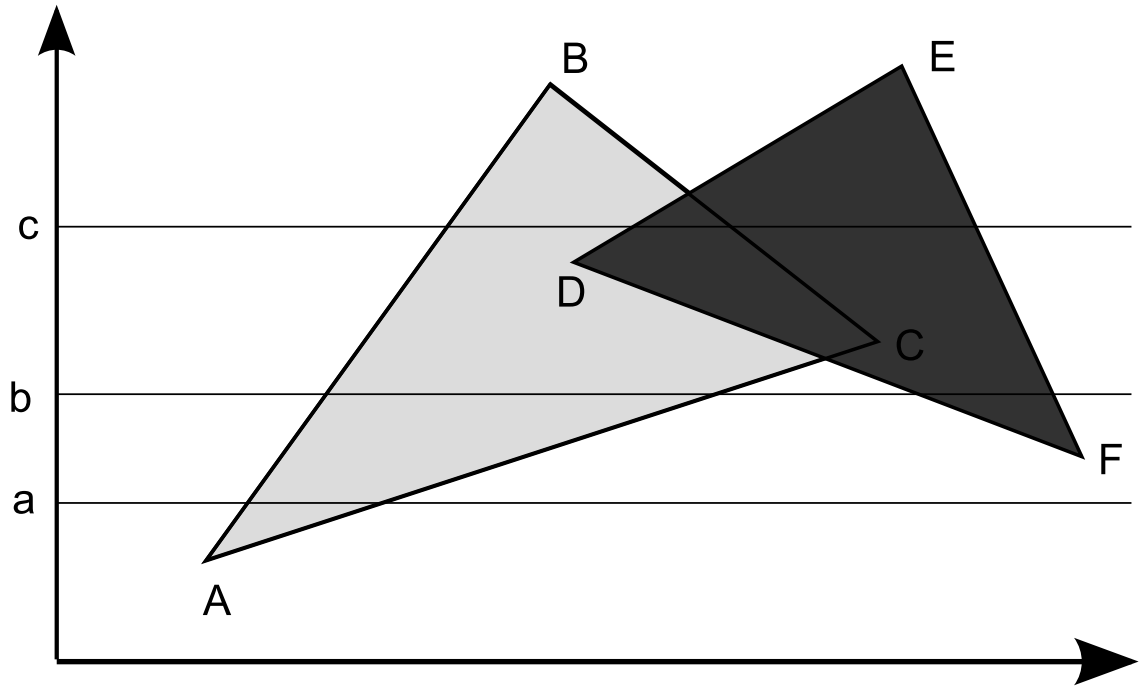



Figure: Scan line algorithm.

# Scan line

Consider three cases

$y = a$ Here $\mathrm{LAE} = \{AB, AC\}$. When we cross $AB$, the in-out data is **true** (we are inside $\Delta ABC$) and we fill the line $y = a$ from $AB$ to $AC$ with the color of the triangle $\Delta ABC$. Then in-out is set to **false** since we are outside triangles. The edge $AC$ is the last one in LAE, so we finish scaning this line.

$y = b$ Here $\mathrm{LAE} = \{AB, AC, FD, FE\}$. When we cross $AB$ the in-out is **true** and we fill the line from $AB$ to $AC$ with the color of the triangle $\Delta ABC$. Then in-out is set to **false** because we are outside triangles. The next edge intersecting the line is $FD$. Then the value of in-out is **true** and we fill the line from $FD$ to $FE$ with the color of the triangle $\Delta DEF$. Since $FE$ is the last edge in $LAE$ we finish scaning this line.

$y = c$ Here $\mathrm{LAE} = \{AB, DE, BC, EF\}$. When we cross $AB$, in-out takes the value **true** and we fill the line from $AB$ to $DE$ with the color of the triangle $\Delta ABC$. Then, since we are inside the next triangle $\Delta DEF$, in-out is still set to **true**. Now, since in-out hasn't changed, we need to check which of triangles $\Delta ABC$ and $\Delta DEF$ is closer. We compare $z$–coordinates of the planes containing these triangles, where $y = c$ and $x$ is the $x$–coordinate of the intersection of the line $y = c$ with $DE$. In our example, $\Delta DEF$ is closer. Hence we fill the line $y = c$ from $DE$ to $BC$ with the color of the triangle $\Delta DEF$. Then in-out of $\Delta ABC$ is **false** and in-out for $\Delta DEF$ is true, so we fill the line from $BC$ to $EF$ with the color of the triangle $\Delta DEF$. The edge $EF$ is the last one in LAE. Thus we end scaning the line $y = c$.