Study Guide

Computer Graphics

Rimvydas Krasauskas

Vilnius University
2012

Computer Graphics
Study Guide

Cycle: First
Study program: Information Technologies
Course unit code:ITKOG
Awarding institution: Department of Computer Science II, Faculty of Mathematics and Informatics

Preparation of the study guide was supported by
the project „Increasing Internationality in Study Programs of the Department of Computer Science II", project number VP1–2.2–ŠMM-07-K-02-070, funded by The European Social Fund Agency and the Government of Lithuania.

Studijų vadovo medžiagos rengimą rėmė
projektas „Kompiuterijos katedros studijų programų tarptautiškumo didinimas", projekto kodas VP1–2.2–ŠMM-07-K-02-070, finansuojamas iš Europos socialinio fondo ir Lietuvos valstybės biudžeto lėšų.

Contents

1 Reading Guide **6**
- 1.1 Reading list . . . 6
- 1.2 Reading references by lectures . . . 6

2 Homework assignments **8**
- 2.1 Line drawing algorithm . . . 8
- 2.2 2D affine fractals . . . 10
  - 2.2.1 2D affine transformations . . . 10
  - 2.2.2 Fractals . . . 12
- 2.3 3D rotations and quaternions . . . 17
  - 2.3.1 Algebra of quaternions . . . 17
  - 2.3.2 Reflections in space . . . 18
  - 2.3.3 Rotations in space . . . 18
- 2.4 Bézier Curves and surfaces . . . 20

3 Test **23**
- 3.1 Examples of questions . . . 23
- 3.2 Answers and solutions . . . 24

4 Exam **26**
- 4.1 Examples of assignments: . . . 26
- 4.2 Assessment rules . . . 27

References **29**

Abstract

This is a study guide to the introductory course on computer graphics. This module teaches some graphics hardware devices, reviews the relevant mathematics, and discusses the fundamental areas of computer graphics. After completing the course, students are expected to know theoretical principles of computer graphics and to be able to apply these methods in practice, i.e. to be able to design and implement simple 3D interactive computer graphics related programs.

1 Reading Guide

1.1 Reading list

Required literature:

[2] S.R. Buss, *3-D Computer graphics. A Mathematical introduction with OpenGL* Cambridge University Press, 2003.

[4] *OpenGL Programming Guide, The official guide for learning OpenGL*, Addison Wesley, Release 1, 1994. http://fly.cc.fer.hr/~unreal/theredbook/

Optional literature:

[1] E. Angel, *Interactive Computer Graphics: a Top-Down Approach with Shader Based OpenGL*, Addison Wesley, 2011.

[3] R.J. Rost et al., *OpenGL Shading Language*, Addison Wesley, 2009.

1.2 Reading references by lectures

Lecture 1 Introduction to computer graphics:

- problem domain and applications,
- overview of history of computer graphics.

Lecture 2 Drawing with OpenGL [2, Chapter I, p. 1–16]:

- the simplest OpenGL program [2, Exercises I.1, I.2, p. 8],
- 2D primitives and their attributes [2, Chapter I.2, p. 4–15],
- interaction via keyboard and mouse `http://www.lighthouse3d.com/tutorials/glut-tutorial/`

Lecture 3 Geometrical transformations I [2, Chapter II.1.1-II.1.3, p. 17–26]:

- 2D transformations: translation, scaling, rotation, reflection,

Lecture 4 Geometrical transformations I (continued):

- affine fractals on the plane [1, Chapter 11.7]

Lectures 5-6 Geometrical transformations II:

- matrices and homogeneous coordinates [2, Chapter II.1.1-II.1.3, p. 17–26] ,
- 3D transformations [2, Chapter II.2, p. 34–46],
- viewing transformations and perspective [2, Chapter II.3, p. 46–57].

Lecture 7 Geometric modeling:

- 3D data sources and acquisition,
- representation schemes, hierarchy.

Lectures 8-9 Geometric models:

- primitive 3D shapes,
- indexed face sets,
- 3D modeling packages: wings3d, SketchUp.

Lectures 10-11 Lighting and shading [2, Chapter III.1]:

- Phong lighting model, illumination and materials,
- Phong and Gouroud shading methods.

Lectures 12-13 Texturing [2, Chapter V]:

- parametrization of primitive surfaces (cylinder, cone, sphere, torus),
- texture space and coordinates, automatic texture generation,
- mipmapping, filtering, bumpmapping.

Lectures 14-15 Advanced geometric models [2, Chapter VII]:

- Bezier curves and surfaces, De Casteljau algorithm,
- splines, NURBS,
- subdivision surfaces.

2 Homework assignments

2.1 Line drawing algorithm

It is impossible to draw the exact line on the raster grid. Hence, we need to approximate the true line. The first and most famous algorithm for line drawing was developed by Jack E. Bresenham in 1962 at IBM. Here we consider this Bresenham line drawing algorithm in details.

The goal is to draw a line between two points (x_0, y_0) and (x_1, y_1) on a raster grid, as it is shown in Fig. 2.1.

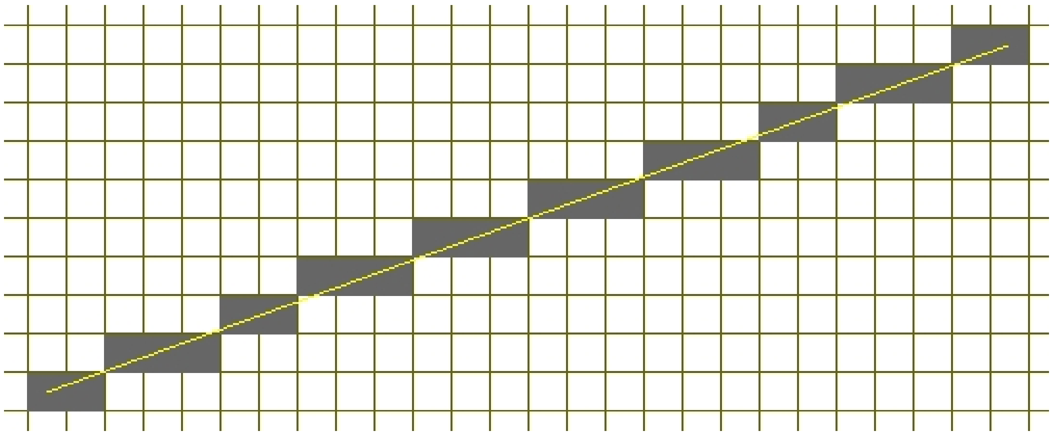

Figure 2.1: A line segment on the raster grid.

It is easy to derive the equation of the line

$$y = k(x - x_0) + y_0, \quad k = \frac{y_1 - y_0}{x_1 - x_0}.$$

For simplicity, let us first restrict the allowable slopes of the line $0 \le k \le 1$.

Suppose, the line-drawing routine so always increments x as it plots. When the routine plots a point at (x, y), there are only two possibilities for the next point on the line: either the point $(x + 1, y)$ or the point $(x + 1, y + 1)$. The idea is to keep track of error, i.e. difference between the true line and the corresponding point on the raster grid.

Listing 1: drawLine (float-point version).

```
void drawLine(int x0, int y0, int x1, int y1)
{
        int dx = x1 - x0;
        int dy = y1 - y0;
        int y = y0;
        float e = 0.0;
        for (int x = x0; x < x1; x++) // -0.5 <= e < 0.5
        {
                drawPoint(x,y);
                e = e + dy/dx;
                if (e >= 0.5) { y = y+1; e = e - 1.0; }
        }
}
```

Here we use floating point operations, that should be avoided in order to increase the speed of drawing. So we multiply the error e by $2dx$ and shift the resulting range of e from $[-dx, dx)$ to $[0, 2dx)$ (only lines from 6 to 12 are changed):

Listing 2: drawLine (integer version).

```
 6          int e = dx;
 7          for (int x = x0; x < x1; x++) // 0 <= e < 2*dx
 8          {
 9                  drawPoint(x,y);
10                  e = e + 2*dy;
11                  if (e >= 2*dx) { y = y+1; e = e - 2*dx; }
12          }
```

The Bresenham algorithm results from then moving the update of the error to after the comparison, shifting the initial value of e, and using a zero test in the 'if' statement (only lines from 6 to 12 are changed):

Listing 3: drawLine (Bresenham algorithm).

```
 6          int e = 2*dy - dx;
 7          for (int x = x0; x < x1; x++) // 2*dy - dx <= e < 2*dy + dx
 8          {
 9                  drawPoint(x,y);
10                  if (e >= 0) { y = y+1; e = e - 2*dx; }
11                  e = e + 2*dy;
12          }
```

Homework assignment 1: - Bresenham's line algorithm

Writte a code of OpenGL programm visualizing line drawing:

- draw a line segment between two points onto a square-shaped window consisting of 32×32 equal small squares;
- user interface (mouse, keyboard, menu) should allow to choose endpoints of the line; convenience of the user interface is *important*!
- the true line segment (shown, e.g. in red) should be approximated by nearest squares;
- switching endpoints of the segment should produce the same result! (see Fig. 2.2, where were are two different possibilities to draw the same line - two pixels are in equal distance from the true line).

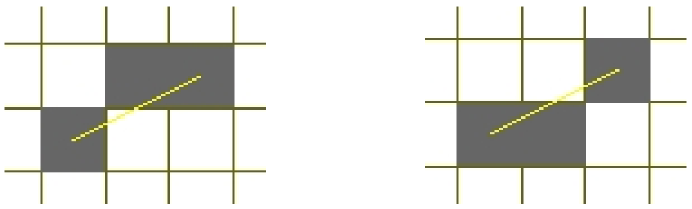

Figure 2.2: Two possibilities to draw the same line.

- compare float point and integer versions of the algorithm by their computational speed (by drawing accidental lines many times); think about how to estimate pure time of computation not including drawing time;
- (additional points): check if your code draws the same lines as the standard OpenGL functions.

Remark

For graphical windows and user interface use the GLUT library, see e.g.
http://www.lighthouse3d.com/tutorials/glut-tutorial/

2.2 2D affine fractals

2.2.1 2D affine transformations

How to represent points and vectors on a plane? We will begin by considering vectors, linear transformations (i.e. elements of linear algebra) and then generalize them to affine case. We know from linear algebra that if we have a basis (i.e. 2 linearly independent vectors) $(\vec{e}_1, \vec{e}_2)$ then we can represent any other vector on the plane uniquely as a linear combination, which can be expressed in the formal matrix notations

$$\vec{v} = x_1\vec{e}_1 + x_2\vec{e}_2 = \begin{pmatrix} \vec{e}_1 & \vec{e}_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

for some scalars x_1 and x_2.

For any other basis $(\vec{e}\,'_1, \vec{e}\,'_2)$ we can express its vectors in the former basis: $\vec{e}\,'_1 = a_{11}\vec{e}_1 + a_{21}\vec{e}_2$ and $\vec{e}\,'_2 = a_{12}\vec{e}_1 + a_{22}\vec{e}_2$, or in the matrix form:

$$\begin{pmatrix} \vec{e}\,'_1 & \vec{e}\,'_2 \end{pmatrix} = \begin{pmatrix} \vec{e}_1 & \vec{e}_2 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{1}$$

Then denoting by x'_1 and x'_2 coordinates of the same vector $\vec{v}$ in the new basis we have

$$\vec{v} = x'_1\vec{e}\,'_1 + x'_2\vec{e}\,'_2 = \begin{pmatrix} \vec{e}\,'_1 & \vec{e}\,'_2 \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} \vec{e}_1 & \vec{e}_2 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}.$$

Therefore, transformation of coordinates are defined by the inverse of matrix that transformation of the coordinate systems:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix}, \quad \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^{-1} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

The simplest examples are:

- scaling with factors λ_1, λ_2

  $$S_{(\lambda_1,\lambda_2)} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix},$$

  in particular, $S_{(-1,1)}$ is the reflection with respect to x_2-axis, and $S_{(-1,-1)}$ is the central symmetry (coinciding with the rotation by angle $\alpha = \pi$, see below);

- rotation by the angle α (counter clock-wise)

  $$R_\alpha = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Affine plane contains points. Vectors appear as differences of points (see e.g. ...). We will treat points and vectors as objects of different type. To define coordinates for an affine plane we would like to find some way to represent any object (point or vector) as a sequence of scalars. Thus it seems natural to generalize the notion of a basis in linear algebra to define a basis in affine plane. Note that two vectors alone are not enough to define a point since we cannot define a point by any combination of vector operations. Indeed, to specify position we will designate an arbitrary a point denoted $\mathcal{O}$ to serve as the origin of our coordinate frame. We see that for any point P, $P - \mathcal{O}$ is just some vector $\vec{v}$. Such a vector can be expressed uniquely as a linear combination of basis vectors. Therefore, given the origin point $\mathcal{O}$ and a couple of basis vectors $(\vec{e}_1, \vec{e}_2)$ any point P can be expressed uniquely as a sum of $\mathcal{O}$ and some linear combination of the basis vectors

$$P = x_1\vec{e}_1 + x_2\vec{e}_2 + \mathcal{O} = \begin{pmatrix} \vec{e}_1 & \vec{e}_2 & \mathcal{O} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

Now we have a possibility to consider transformations between coordinate systems that can have different origins. Similarly to transformation of vector basis (1), we can describe transformations of affine frames $(\vec{e}_1, \vec{e}_2, \mathcal{O})$ and $(\vec{e}'_1, \vec{e}'_2, \mathcal{O}')$:

$$\begin{pmatrix} \vec{e}'_1 & \vec{e}'_2 & \mathcal{O}' \end{pmatrix} = \begin{pmatrix} \vec{e}_1 & \vec{e}_2 & \mathcal{O} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}. \tag{2}$$

The matrix A on the right in (2) corresponds to most general affine transformation. The simplest examples include earlier described linear transformations and translations:

- scaling with factors λ_1, λ_2

$$S_{(\lambda_1,\lambda_2)} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

- rotation by the angle α (counter clock-wise)

$$R_\alpha = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

- translation by a vector $\vec{u} = u_1\vec{e}_1 + u_2\vec{e}_2$

$$T_{\vec{u}} = \begin{pmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{pmatrix};$$

The general affine matrix A can be decomposed into the product of two simple ones $A = T_{\vec{u}}L$, i.e. translation by vector $\vec{u} = a_{13}\vec{e}_1 + a_{23}\vec{e}_2$ and the general linear transformation L:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a_{13} \\ 0 & 1 & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3}$$

2.2.2 Fractals

Fractals can be shortly defined as geometric objects with two properties:

- self-similarity, i.e. the whole object is similar to its smaller parts;
- having fractal dimension.

Let us informally discuss dimensions by considering three simple examples: a unit line segment I, a unit square S, and a unit cube C. We expect that these objects are 1-, 2-, and 3-dimensional, respectively. Suppose we try to measure them with the smallest unit $\epsilon = 1/n$, for a certain integer n. By dividing each of the objects into similar parts of that size, we see that the line segment I will have $N = n$ parts, the unit square S will have $N = n^2$ parts, and the unit cube C will have $N = n^3$ parts. In each case we have generated new objects by scaling the original object by a factor of ϵ and replicating it k times. In all cases $k = n^d$, where d is dimension of the considered object. Hence, solving for d we get

$$\ln N = \ln(n^d) = d \ln n, \quad d = \frac{\ln N}{\ln n}.$$

Therefore, the formula $d = \ln N / \ln n$ can be treated as approximate dimension formula, when n goes to infinity. Hence, the fractal dimension of an object is determined by how many similar objects we create by subdivision.

Consider one more example – the Sierpinski triangle (also called gasket). We start from a triangle, subdivide it in 4 equal parts, and erase the middle part. For the three remaining triangles we apply the same procedure and repeat this infinite number of times. The figure that will remain in limit is called the Sierpinski triangle (see Fig. 2.3). Let us count dimension: after k iteration steps the side of triangle

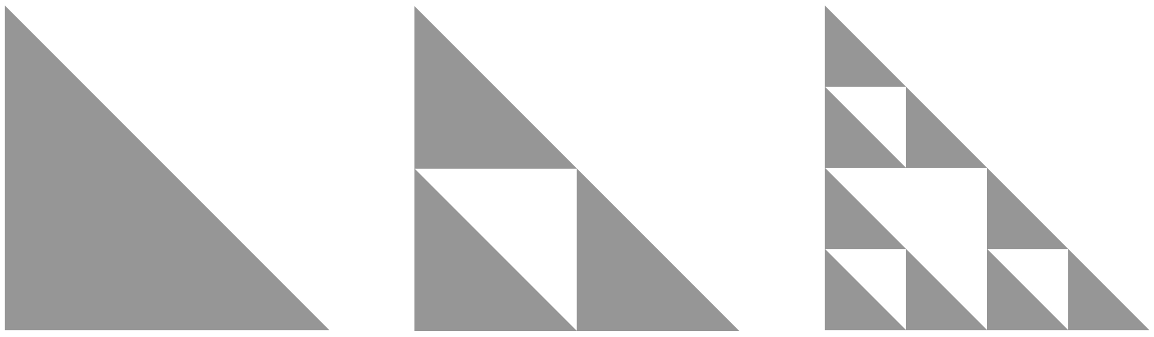
Figure 2.3: The Sierpinski triangle generation by subdivision.

is subdivided into $n = 2^k$ parts, but the Sierpinski triangle itself will have $N = 3^k$ similar parts. Hence

$$d = \lim_{n\to\infty} \frac{\ln N}{\ln n} = \lim_{k\to\infty} \frac{\ln 3^k}{\ln 2^k} = \lim_{k\to\infty} \frac{k \ln 3}{k \ln 2} = \frac{\ln 3}{\ln 2} \approx 1.58496$$

This is a particular case of 2D affine fractals which we are going to study below.

This kind of fractals can be defined using iterated function system (IFS) approach. For a collection of affine transformations $\{f_1, \ldots, f_r : \mathbb{R}^2 \to \mathbb{R}^2\}$ of a plane, define the *Hutchinson operator*

$$H(X) = \bigcup_{i=1}^{r} f_i(X), \quad X \subset \mathbb{R}^2,$$

acting on subsets in the plane $\mathbb{R}^2$. If all transformations f_i are similarities (i.e. they preserve angles and change all distances in the same ratio) with scaling coefficient less than 1 then for every compact set Δ (e.g. Δ can be finite number of points and/or polygons in $\mathbb{R}^2$)

$$F = \lim_{n\to\infty} H^{\circ n}(\Delta), \quad H^{\circ n} = \underbrace{H \circ H \circ \cdots \circ H}_{n \text{ times}},$$

exists and is called a fractal generated by IFS $\{f_1, \ldots, f_r\}$. Actually, the existence and uniqueness of F is a consequence of the contraction mapping principle, where F plays a role of a fixed point (i.e. set in this case) of the Hutchinson operator: $F = H(F)$. The interesting consequence: the limit figure F does not depend on the initial figure Δ! The resulting fractal F depends only on the list of IFS $\{f_1, \ldots, f_r\}$.

Let all IFS transformations $\{f_1, \ldots, f_r\}$ are similarities with scaling coefficients $\{\lambda_1, \ldots, \lambda_r\}$. Then there is a simple formula for the dimension d of the fractal:

$$\lambda_1^d + \cdots + \lambda_r^d = 1. \tag{4}$$

See derivation in: http://classes.yale.edu/fractals/fracanddim/Moran/Moran.html

Consider a simple case of affine fractal – the Sierpinski triangle (see Fig. 2.3). The generating IFS has three transformations:

$$f_1 = S_{0.5}, \quad f_2 = T_{0.5\vec{e}_1} S_{0.5}, \quad f_3 = T_{0.5\vec{e}_2} S_{0.5},$$

where $S_{0.5} = S_{(0.5,0.5)}$ is a uniform scaling by factor $1/2$.

We present an OpenGL code that shows several initial figures that eventually converges to this fractal. Listing 4 the function `drawFractal()` that actually draws a triangle on the step 0, and proceeds recursively applying three transformations f_1, f_2, f_3.

Listing 4: Function `drawFractal()`.

```
void drawFractal(int step)
{
 switch(step) {
   case 0:
        glBegin( GL_TRIANGLES );
                glVertex3f( 1.0, 0.0, 0.0 );
                glVertex3f( 0.0, 1.0, 0.0 );
                glVertex3f( 0.0, 0.0, 0.0 );
        glEnd();
        break;
   default:
        glPushMatrix();
        glPushMatrix();
        glScalef(0.5,0.5,1.0);
        drawFractal(step-1);
        glPopMatrix();
        glTranslatef(0.5,0.0,0.0);
        glScalef(0.5,0.5,1.0);
        drawFractal(step-1);
        glPopMatrix();
        glTranslatef(0.0,0.5,0.0);
        glScalef(0.5,0.5,1.0);
        drawFractal(step-1);
        break;
 }
}
```

Listing 5 contains the full code that uses the function `drawFractal()`. Interactivity is simple: by pressing space bar one goes to the next step, 'escape' button ends the program. The function `resizeWindow()` is relatively sophisticated because it takes into account non-square shaped windows (the fractal picture is not deformed - can be only uniformly scaled).

Listing 5: Recursive Sierpinski Triangle.

```c
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

int CurrentStep = 0;
const int MaxStep = 10;
const double Xmin = -0.1, Xmax = 1.1;
const double Ymin = -0.1, Ymax = 1.1;

void myKeyboardFunc( unsigned char key, int x, int y )
{
        switch ( key ) {
        case ' ':
                CurrentStep = (CurrentStep+1)%MaxStep;
                glutPostRedisplay();
                break;
        case 27:
                exit(1);
        }
}

void drawFractal(int step) {...}

void drawScene(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f( 1.0, 1.0, 1.0 );
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  drawFractal(CurrentStep);
  glFlush();
}

void initRendering()
{
  glEnable ( GL_DEPTH_TEST );
}

void resizeWindow(int w, int h)
{
        double scale, center;
        double windowXmin, windowXmax, windowYmin, windowYmax;
        glViewport( 0, 0, w, h );
        w = (w==0) ? 1 : w;
        h = (h==0) ? 1 : h;
        if ( (Xmax-Xmin)/w < (Ymax-Ymin)/h ) {
                scale = ((Ymax-Ymin)/h)/((Xmax-Xmin)/w);
                center = (Xmax+Xmin)/2;
                windowXmin = center - (center-Xmin)*scale;
                windowXmax = center + (Xmax-center)*scale;
                windowYmin = Ymin;
                windowYmax = Ymax;
        }
        else {
                scale = ((Xmax-Xmin)/w)/((Ymax-Ymin)/h);
                center = (Ymax+Ymin)/2;
                windowYmin = center - (center-Ymin)*scale;
                windowYmax = center + (Ymax-center)*scale;
                windowXmin = Xmin;
                windowXmax = Xmax;
        }
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity();
        glOrtho( windowXmin, windowXmax, windowYmin, windowYmax, -1, 1 );
}

int main( int argc, char** argv )
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH );
        glutInitWindowPosition( 20, 60 );
        glutInitWindowSize( 500, 500 );
        glutCreateWindow( "Sierpinski triangle" );
        initRendering();
        glutKeyboardFunc( myKeyboardFunc );
        glutReshapeFunc( resizeWindow );
        glutDisplayFunc( drawScene );
        fprintf(stdout, "Press space bar -> the next step; esc -> to quit.\n");
        glutMainLoop( );
        return(0);
}
```

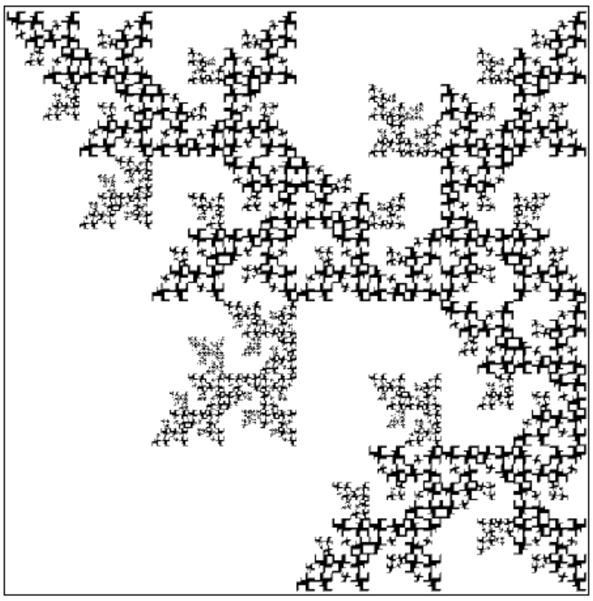

Figure 2.4: The example of 2D affine fractal.

Homework assignment 2: - Draw 2D fractal

For a given picture of 2D affine fractal in a square (e.g. see Fig. 2.4) write an OpenGL program that recreates this picture:

- find main 4 transformations that define IFS (iterated functional system) of the given fractal, and calculate its dimension;
- write an OpenGL program that recreates the picture, where different 4 parts (that correspond to 4 transformations) are colored differently; use Listings 5 and 4 as initial example;
- present an alternative code that uses probability approach.

2.3 3D rotations and quaternions

2.3.1 Algebra of quaternions

Complex numbers are an extension of the real number system and can be written in the form $a + b\mathbf{i}$, where a and b are both real numbers and $\mathbf{i}$ is called an *imaginary unit* with the property $\mathbf{i}^2 = -1$. The quaternions are just a further extension of these complex numbers. A quaternion is usually written as

$$q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k},$$

where a, b, and c are scalar values, and $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are the special quaternions satisfying conditions

$$\mathbf{i}^2 = -1, \quad \mathbf{j}^2 = -1, \quad \mathbf{k}^2 = -1$$

and

$$\mathbf{ij} = \mathbf{k},\ \mathbf{jk} = \mathbf{i},\ \mathbf{ki} = \mathbf{j},\ \mathbf{ji} = -\mathbf{k},\ \mathbf{kj} = -\mathbf{i},\ \mathbf{ik} = -\mathbf{j}.$$

Addition of quaternions is defined as addition of vectors. Hence for the two quaternions

$$q_1 = a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}, \quad q_2 = a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k},$$

their sum of the two quaternions is

$$q_1 + q_2 = (a_1 + a_2) + (b_1 + b_2)\mathbf{i} + (c_1 + c_2)\mathbf{j} + (d_1 + d_2)\mathbf{k}.$$

Multiplication is more complicated: first we have to multiply component-wise, and then use the product formulas for $\mathbf{i}$, $\mathbf{j}$, and k to simplify the resulting expression. Hence the product of q_1 and q_2 is

$$\begin{aligned} q_1q_2 &= (a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k})(a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}) \\ &= a_1a_2 + a_1b_2\mathbf{i} + a_1c_2\mathbf{j} + a_1d_2\mathbf{k} + b_1a_2\mathbf{i} + b_1b_2\mathbf{i}^2 + b_1c_2\mathbf{ij} + b_1d_2\mathbf{ik} \\ &\quad + c_1a_2\mathbf{j} + c_1b_2\mathbf{ji} + c_1c_2\mathbf{j}^2 + c_1d_2\mathbf{jk} + d_1a_2\mathbf{k} + d_1b_2\mathbf{ki} + d_1c_2\mathbf{kj} + d_1d_2\mathbf{k}^2 \\ &= (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + a_2b_1 + c_1d_2 - d_1c_2)\mathbf{i} \\ &\quad + (a_1c_2 + a_2c_1 + d_1b_2 - b_1d_2)\mathbf{j} + (a_1d_2 + a_2d_1 + b_1c_2 - c_1b_2)\mathbf{k}. \end{aligned}$$

It will be convenient to use the following alternative representation of quaternions. Any quaternion $q$can be represented as a sum of its scalar (real) part and its vector (imaginary) part:

$$q = r + \mathbf{v} = r\mathbf{1} + v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}, \quad r = \mathrm{Re}(q), \quad v = \mathrm{Im}(q).$$

The product of quaternions in these notations has the following compact formula:

$$qq' = (r + \mathbf{v})(r' + \mathbf{v}') = (rr' - \mathbf{v} \cdot \mathbf{v}') + (r\mathbf{v}' + r'\mathbf{v} + \mathbf{v} \times \mathbf{v}'),$$

where $\mathbf{v} \cdot \mathbf{v}'$ and $\mathbf{v} \times \mathbf{v}'$ are dot and vector products of vectors in $\mathbb{R}^3$. If $q = r + \mathbf{v}$ then $\bar{q} = r - \mathbf{v}$ is called a *conjugate* quaternion. The norm of a quaternion (or its length) is $|q| = \sqrt{q\bar{q}}$. If q is a non-zero quaternion the it has the inverse $q^{-1} = \bar{q}/|q|^2$. Indeed,

$$q^{-1}q = qq^{-1} = q\bar{q}/|q|^2 = 1.$$

2.3.2 Reflections in space

If $q \in \operatorname{Im}\mathbb{H}$ then it is a vector $q = \mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. We will identify vectors in 3D space $\mathbb{R}^3$ with imaginary quaternions $\operatorname{Im}\mathbb{H}$. Therefore, we can multiply vectors (as quaternions):

$$\mathbf{v}\mathbf{v}' = -\mathbf{v}\cdot\mathbf{v}' + \mathbf{v}\times\mathbf{v}', \tag{5}$$

Also conjugation and inverse operations are defined:

$$\bar{\mathbf{v}} = -\mathbf{v}, \quad \mathbf{v}^{-1} = -\mathbf{v}/|\mathbf{v}|^2. \tag{6}$$

Let $\mathbf{v}$ and $\mathbf{n}$ are vectors, $|\mathbf{n}| = 1$, then operation

$$A_{\mathbf{n}}\mathbf{v} = \mathbf{n}\mathbf{v}\mathbf{n}$$

defines a reflection about the plane perpendicular to $\mathbf{n}$ (and passing through the origin).

Indeed, using the identity for the cross product

$$(a\times b)\times c = (a\cdot c)b - (b\cdot c)a$$

one can derive

$$\mathbf{n}\mathbf{v}\mathbf{n} = (\mathbf{n}\times\mathbf{v})\times\mathbf{n} - (\mathbf{n}\cdot\mathbf{v})\mathbf{n} = \mathbf{v} - 2(\mathbf{n}\cdot\mathbf{v})\mathbf{n}.$$

The vector $\mathbf{v} - 2(\mathbf{n}\cdot\mathbf{v})\mathbf{n}$ is exactly the reflection of $\mathbf{v}$ with respect to $\mathbf{n}$.

2.3.3 Rotations in space

Recall that any rotation in space is a composition of two reflections, and the rotation angle is twice bigger than the angle between the planes of reflections. In terms of quaternions this can be expressed as

$$R\mathbf{v} = A_{\mathbf{n}_2}A_{\mathbf{n}_1}\mathbf{v} = \mathbf{n}_2\mathbf{n}_1\mathbf{v}\mathbf{n}_1\mathbf{n}_2 = q\mathbf{v}q^{-1},$$

where $q = \mathbf{n}_2\mathbf{n}_1$ (see (6)). Using (5) we get

$$q = \mathbf{n}_2\mathbf{n}_1 = -\mathbf{n}_2\cdot\mathbf{n}_1 + \mathbf{n}_2\times\mathbf{n}_1.$$

Suppose the rotational angle is θ and the rotational axis is defined by the unit vector $\mathbf{u}$ (in the direction of $\mathbf{n}_1\times\mathbf{n}_2$). Then

$$\mathbf{n}_2\cdot\mathbf{n}_1 = -\cos\frac{\theta}{2}, \quad \mathbf{n}_2\times\mathbf{n}_1 = -\sin\left(\frac{\theta}{2}\right)\mathbf{u}.$$

Finally the quaternion q is expressed via θ and $\mathbf{u}$:

$$q = \pm\cos\frac{\theta}{2} \pm \sin\left(\frac{\theta}{2}\right)\mathbf{u}. \tag{7}$$

Now we are ready to derive matrix representation of the rotation, which is given by a unit quaternion q. That is, we are looking for the matrix that affects a rotation about an axis $\mathbf{u}$ by angle θ in $\mathbb{R}^3$. Assume $q = q_0 + \mathbf{q}$, $\mathbf{q} = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$, is a unit quaternion, i.e. $|q|^2 = q_0^2 + |\mathbf{q}|^2 = 1$. Then, from $\mathbf{v}' = q\mathbf{v}\bar{q}$ we can determine the matrix that transforms the components of a given vector $\mathbf{v}$ to those of $\mathbf{v}'$. We calculate the first product

$$\mathbf{v}' = (q_0 + \mathbf{q})\mathbf{v}(q_0 - \mathbf{q}) = (-\mathbf{q}\cdot\mathbf{v} + q_0\mathbf{v} + \mathbf{q}\times\mathbf{v})(q_0 - \mathbf{q}).$$

Since $(\mathbf{q}\times\mathbf{v})\cdot\mathbf{q}=0$, the real part is zero:

$$\mathrm{Re}\mathbf{v}' = -q_0(\mathbf{q}\cdot\mathbf{v}) + (q_0\mathbf{v}+\mathbf{q}\times\mathbf{v})\cdot\mathbf{q} = 0.$$

Therefore, using $(\mathbf{q}\times\mathbf{v})\times\mathbf{q} = (\mathbf{q}\cdot\mathbf{q})\mathbf{v} - (\mathbf{v}\cdot\mathbf{q})\mathbf{q}$ we get

$$\begin{aligned}\mathbf{v}' &= \mathrm{Im}\mathbf{v}' = (\mathbf{q}\cdot\mathbf{v})\mathbf{q} + q_0^2\mathbf{v} + q_0(\mathbf{q}\times\mathbf{v}) - q_0(\mathbf{v}\times\mathbf{q}) - (\mathbf{q}\times\mathbf{v})\times\mathbf{q}\\ &= (q_0^2 - \mathbf{q}\cdot\mathbf{q})\mathbf{v} + 2\mathbf{q}(\mathbf{q}\cdot\mathbf{v}) + 2q_0\mathbf{q}\times\mathbf{v}.\end{aligned}$$

Hence, the matrix R, such that $\mathbf{v}'=R\mathbf{v}$, can be extracted

$$R = (q_0^2 - \mathbf{q}\cdot\mathbf{q})I + 2\mathbf{q}\mathbf{q}^T + 2q_0\begin{pmatrix}0 & -q_3 & q_2\\ q_3 & 0 & -q_1\\ -q_2 & q_1 & 0\end{pmatrix}$$

Finally the rotational matrix R corresponding to the quaternion $q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$ is

$$R = \begin{pmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & q_1q_2-q_0q_3 & q_1q_3+q_0q_2\\ q_1q_2+q_0q_3 & q_0^2-q_1^2+q_2^2-q_3^2 & q_2q_3-q_0q_1\\ q_1q_3-q_0q_2 & q_2q_3+q_0q_1 & q_0^2-q_1^2-q_2^2+q_3^2\end{pmatrix} \tag{8}$$

The same matrix R in terms of angle θ and rotational axis $\mathbf{u} = u_1\mathbf{i}+u_2\mathbf{j}+u_3\mathbf{k}$ is

$$R = (\cos\theta)I + (1-\cos\theta)\begin{pmatrix} u_1^2 & u_1u_2 & u_1u_3\\ u_2u_1 & u_2^2 & u_2u_3\\ u_3u_1 & u_3u_2 & u_3^2\end{pmatrix} + \sin\theta\begin{pmatrix}0 & -u_3 & u_2\\ u_3 & 0 & -u_1\\ -u_2 & u_1 & 0\end{pmatrix}.$$

Denoting $c=\cos\theta$ and $s=\sin\theta$ we get

$$R = \begin{pmatrix} (1-c)u_1^2+c & (1-c)u_1u_2 - su_3 & (1-c)u_1u_3+su_2\\ (1-c)u_2u_1+su_3 & (1-c)u_2^2+c & (1-c)u_2u_3-su_1\\ (1-c)u_3u_1-su_2 & (1-c)u_3u_2+su_1 & (1-c)u_3^2+c\end{pmatrix}. \tag{9}$$

Homework assignment 3: - Draw 3D fractal

For a given text file with a 'cloud' of points (i.e. their coordinates) representing a 3D affine fractal in a unit cube write an OpenGL program:

- that interactively demonstrates this cloud of points;
- find main 4 transformations that define IFS of the given fractal;
- the program should iteratively recreate the 3D fractal as polyhedral shape with the possibility to look at it from all directions with minimal shading;
- animate each of 4 transformations as a composition of translation, rotation, and scaling.

2.4 Bézier Curves and surfaces

A Bézier curve of degree 1 is a line segment with endpoints $\mathbf{p}_0$ and $\mathbf{p}_1$ parametrized as usual

$$\mathbf{p}_0^1(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1.$$

The first non-trivial case is a Bézier curve of degree 2, also called a *quadratic Bézier curve*. This curve is built using repeated linear interpolations. Let $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ be three given points and let $0 \leq t \leq 1$. We calculate:

$$\begin{aligned} \mathbf{p}_0^1(t) &= (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \\ \mathbf{p}_1^1(t) &= (1-t)\mathbf{p}_1 + t\mathbf{p}_2 \\ \mathbf{p}_0^2(t) &= (1-t)\mathbf{p}_0^1(t) + t\mathbf{p}_1^1(t). \end{aligned}$$

Inserting the first two equations into the third one, we get

$$\mathbf{p}_0^2(t) \quad = \quad (1-t)^2\mathbf{p}_0 + 2(1-t)t\mathbf{p}_1 + t^2\mathbf{p}_2. \tag{10}$$

This formula is quadratic in t and defines the quadratic Bézier curve. The point $\mathbf{p}_0^2$ traces out the curve, if t varies in the interval $[0, 1]$. In Fig. 2.5 the geometric construction of the curve point $\mathbf{p}_0^2$ for $t = 1/3$ is shown.

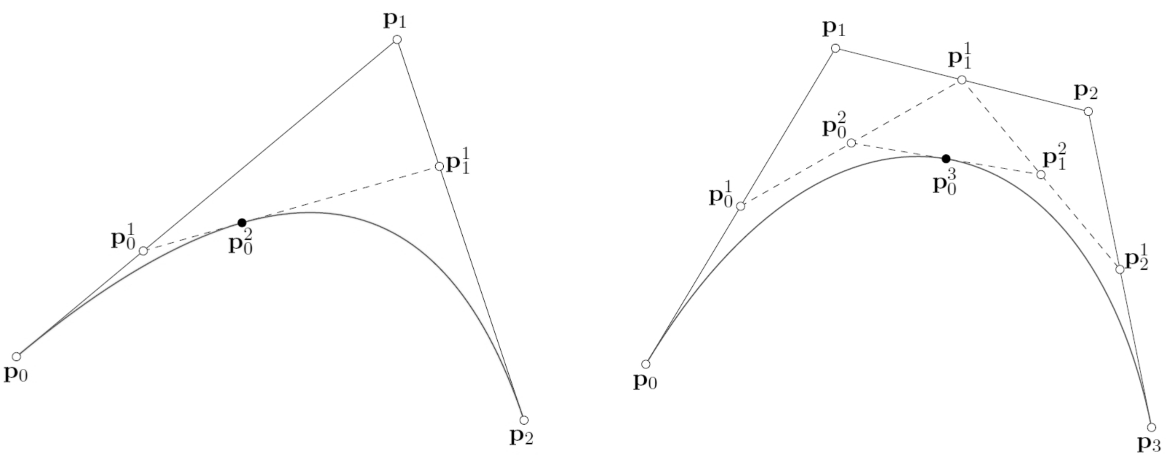

Figure 2.5: De Casteljau algorithm for quadratic and cubic Bézier curve.

For $0 \leq t \leq 1$ the curve lies in the triangle formed by $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$. This is called the *convex hull property*. As special points we have $\mathbf{p}^2(0) = \mathbf{p}_0$ and $\mathbf{p}^2(1) = \mathbf{p}_2$. The construction also shows the following property:

$$ratio(\mathbf{p}_0, \mathbf{p}_0^1, \mathbf{p}_1) = ratio(\mathbf{p}_1, \mathbf{p}_1^1, \mathbf{p}_2) = ratio(\mathbf{p}_0^1, \mathbf{p}_0^2, \mathbf{p}_1^1) = \frac{t}{1-t}. \tag{11}$$

All ratios are equal, this proves the affine invariance of the curve construction. If you look at Fig. 2.5, you see that for $t = 0$ the curve is tangent to the line $\mathbf{p}_0\mathbf{p}_1$ and for $t = 1$ it is tangent to the line $\mathbf{p}_1\mathbf{p}_2$. Similarly a *cubic Bézier curve* (i.e. of degree 3) is defined by four control points see Fig. 2.5(right):

$$\mathbf{p}_0^3(t) \quad = \quad (1-t)^3\mathbf{p}_0 + 3(1-t)^2t\mathbf{p}_1 + 3(1-t)t^2\mathbf{p}_2 + t^3\mathbf{p}_3. \tag{12}$$

In order to represent other curves of degree 2 (e.g. circles) we need to generalize the introduced formulas. Let us start from the description of a rational quadratic Bézier curve:

$$\mathbf{p}(t) \quad = \quad \frac{(1-t)^2 w_0 \mathbf{p}_0 + 2(1-t)tw_1\mathbf{p}_1 + t^2 w_2 \mathbf{p}_2}{(1-t)^2 w_0 + 2(1-t)tw_1 + t^2 w_2}. \tag{13}$$

As you can see, every point is associated with it's own weight. This is the general form of a rational quadratic Bézier curve. With some more involved math it can be translated into a simpler form, which is called the *standard form* and it looks like this:

$$\mathbf{p}(t) \quad = \quad \frac{(1-t)^2 \mathbf{p}_0 + 2(1-t)tw\mathbf{p}_1 + t^2 \mathbf{p}_2}{(1-t)^2 + 2(1-t)tw + t^2}. \tag{14}$$

The outer weights w_0 and w_2 are gone (their value is 1) and the central inner weight is transformed to w. The term rational simply reflects the fact, that every rational Bézier curve is build from a rational expression, which is much more complicated then the expression for integral curves.

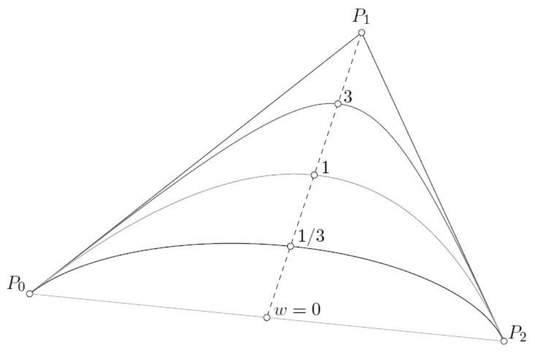

Figure 2.6: The middle weight variation of a rational quadratic Bézier curve.

Using rational Bézier surfaces one can represent patches of straight circular cylinders, spheres and torus patches (see Fig. 2.7). Details can be found in [2, Chapter VII.13-14].

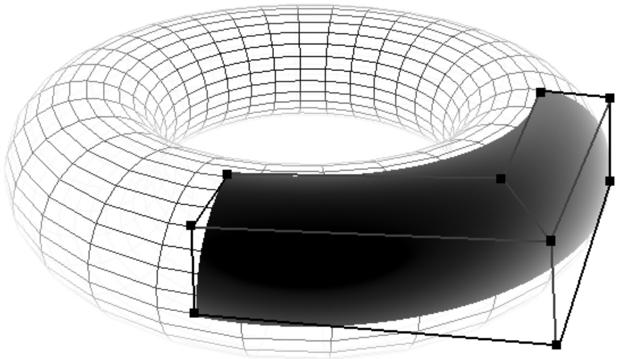

Figure 2.7: Representing a $1/16$ of a torus as a rational bi-quadratic Bézier surface.

Homework assignments 4-5: - Render 3D object using various CG techniques

Create a scene on the horizontal square with two vertical walls in one corner and one object in the center.

- The user interface – the menu invoked by RMB click.
- The object is a 3D solid made of cubes wings3d formate several distinguished edges that should be rounded according to the example on Fig. 2.8 (use Bézier surfaces for rounding!); the rounding radius is controlled by keys "R" (increase) and "r" (decrease).
- Navigation: at least two types – "walk" (use arrow-keys), "examine", or "game like".
- Collisions are computed in the "walk" mode with the walls and with the bounding box of the object (sliding – additional scores).
- Adjustable light sources: directional, point and spot light.
- The object shadow on the horizontal square and vertical walls.
- Camera control: at least 5 different viewpoints, orthogonal and perspective projections, adjustable fov (fiels of view).
- Textures on the object that is compatible with textures on Bézier surfaces.

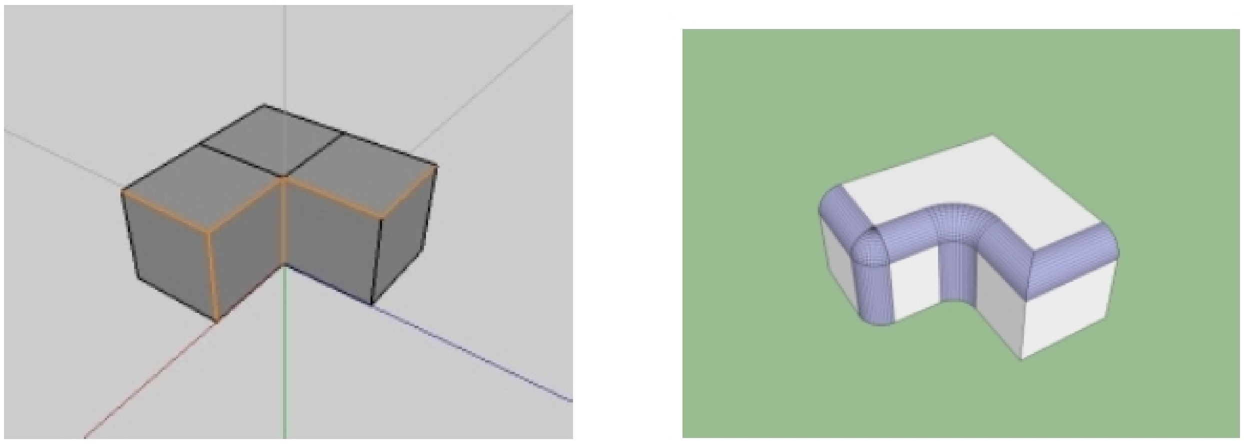
Figure 2.8: The 'cubical' object with distinguished edges and its rounding.

3 Test

3.1 Examples of questions

Here are examples of questions in the test.

1. Sketch a picture, that will be drawn by OpenGL code below, if a function drawTriangle() draws a triangle with vertices $(0,0,0)$, $(1,0,0)$, $(0,2,0)$:

   Listing 6: Fragment of OpenGL code.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotatef(45.0,0.0,0.0,1.0);
glTranslatef(2.0,0.0,0.0);
glPushMatrix();
glTranslatef(0.0,-2.0,0.0);
drawTriangle();
glPopMatrix();
glRotatef(180.0,0.0,0.0,1.0);
glTranslatef(0.0,-2.0,0.0);
drawTriangle();
```

2. Express a given affine transformation matrix of as a product of three matrices, that correspond to scale, rotation and translation:

$$\begin{pmatrix} -2 & -1 & 2 \\ -2 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}.$$

3. For a given picture of affine fractal find matrices of the main 4 transformations (its iterated functional system), and calculate its dimension:

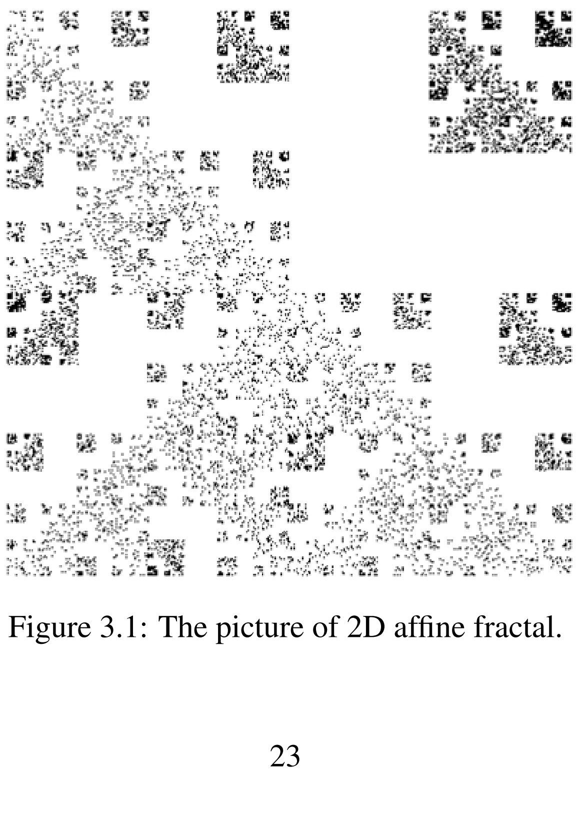

Figure 3.1: The picture of 2D affine fractal.

4. First we apply rotation by -90 degrees about y axis, then we rotate by 180 degrees about x axis. Find the angle and axis of the composite rotation:
   (a) using matrix approach;
   (b) using quaternions.

3.2 Answers and solutions

1. See Fig. 3.2.

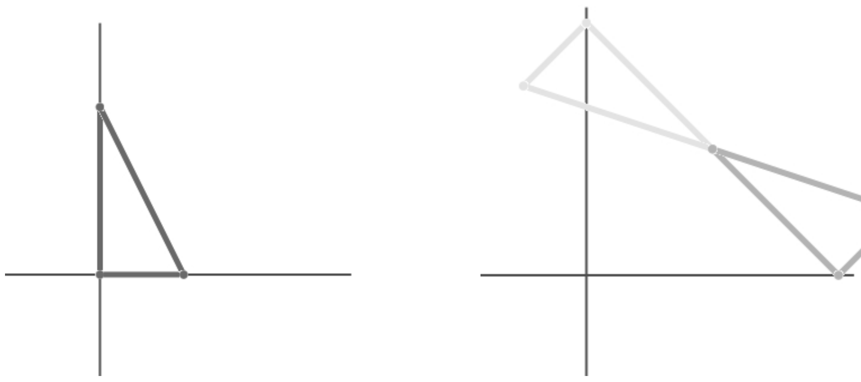

Figure 3.2: The triangle drawn by triangleDraw(), and the final picture.

2. Here is decomposition into product of scaling, rotation, and translation (from right to left):

$$\begin{pmatrix} -2 & -1 & 2 \\ -2 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -2\sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

3. The main 4 transformations of our iterated functional system have following matrices (visually corresponding to the four quadrants of the square that contains the picture)

$$\begin{pmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0.25 & 0 & 0.75 \\ 0 & 0.25 & 0.75 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & -0.5 & 0.5 \\ 0.5 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The fractional dimension can be computed using Moran formula (4). since all transformations are similarities (where d is dimension of the fractal):

$$\lambda_1^d + \lambda_2^d + \lambda_3^d + \lambda_4^d = 1, \quad \lambda_1 = 1/4, \quad \lambda_2 = \lambda_3 = \lambda_4 = 1/2$$

Denoting $x = 1/2^d$ we get the quadratic equation $x^2 + 3x - 1 = 0$ with the positive root $x = (\sqrt{13} - 3)/2$. Finally

$$d = \log_2 \frac{1}{x} = \log_2 \frac{2}{\sqrt{13} - 3} = \log_2 \frac{3 + \sqrt{13}}{2} \approx 1.723678968$$

4. *Answer*: 180 degrees, axis $(1/\sqrt{2}, 0, -1/\sqrt{2})$.

   (a) *Solution by matrix approach*
   we represent rotations by their matrices and then multiply them (note right-to-left order)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{pmatrix} = M.$$

   Since the matrix $M = (m_{ij})$ is a rotational matrix, from (9) follows

$$\cos\alpha = (m_{11} + m_{22} + m_{33} - 1)/2 = -1, \quad \sin\alpha = 0.$$

   Hence the rotational angle is $\theta = \pi$ (i.e. 180 degrees) and we have the following equation of matrices:

$$\begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2u_1^2 - 1 & 2u_1u_2 & 2u_1u_3 \\ 2u_2u_1 & 2u_2^2 - 1 & 2u_2u_3 \\ 2u_3u_1 & 2u_3u_2 & 2u_3^2 - 1 \end{pmatrix}.$$

   Thus $-1 = 2u_2^2 - 1$, $u_2 = 0$, and $1 = 2u_1^2 = 2u_3^2$, i.e. $u_1 = \pm 1/\sqrt{2}$, $u_3 = \pm 1/\sqrt{2}$. On the other hand, $2u_1u_3 = -1$. Therefore, the rotational axis is $\mathbf{u} = \pm(1/\sqrt{2}, 0, -1/\sqrt{2})$. Here the sign is not unique because of the exceptional case $\theta = \pi$.

   (b) *Solution using quaternions*
   According to the formula (7), the rotation by -90 degrees about y axis is represented by the quaternion

$$q_1 = \cos(-\pi/4) + \sin(-\pi/4)\mathbf{j} = (1 - \mathbf{j})/\sqrt{2},$$

   and the rotation by 180 degrees about x axis is represented by the quaternion

$$q_2 = \cos(\pi/2) + \sin(\pi/2)\mathbf{i} = \mathbf{i}.$$

   Hence the composition of these rotations is represented by the product of our quaternions

$$q_2q_1 = \mathbf{i}(1 - \mathbf{j})/\sqrt{2} = (\mathbf{i} - \mathbf{k})/\sqrt{2} = \cos(\pi/2) + \sin(\pi/2)\mathbf{u},$$

   where $\mathbf{u} = \pm(\mathbf{i} - \mathbf{k})/\sqrt{2}$. This is exactly the same answer.

4 Exam

Exam has 4 groups of assignments:

1. viewing in OpenGL;
2. lighting and shading;
3. texturing;
4. Bézier curves and surfaces.

4.1 Examples of assignments:

1. *Viewing in OpenGL:*
   We are looking from a position $(0, 2, -1)$ at a sphere with a center point $(1, 4, 1)$ and radius 1. Fill empty space in the following code in such a way that the image of the sphere will be $2/3$ height and $1/2$ width of the window, and exactly one half of the sphere will be visible.

   ```
   glMatrixMode(                    );
   glLoadIdentity();
   gluLookAt(                       );
   glMatrixMode(                    );
   glLoadIdentity();
   gluPerspective(                  );
   ```

2. *Lighting and shading:*

   - Mark features of ambient (respectively: diffuse, specular) reflection in the Phong lighting model:
     ○ Models rough surfaces – where light scatters equally in all directions
     ○ Approximates the effect of inter-reflections
     ○ Models highlights from smooth, shiny surfaces
     ○ Has a point or directional source
     ○ Sourceless – constant over entire surface
     ○ Depends on surface normal
     ○ Does not depend on surface normal
     ○ Depends on viewpoint
     ○ Does not vary based on viewpoint
   - Given three vertices $A_0 = (3, 1, 2)$, $A_1 = (3, -1, 3)$, $A_2 = (4, -1, 4)$ of a triangle compute a normal of the plane that the points lie in. Explain relation between the normal direction and orientation of the triangle.

3. *Texturing*

   - The brick texture is applied to the rectangular wall. We would like to reduce the width of every brick 4 times. Fill empty spaces in the following code:

```
glBegin(GL_POLYGON);
glTexCoord2f(   0.0,    0.0); glVertex3f(-1.0,0.0,0.0);
glTexCoord2f(      ,       ); glVertex3f(3.0,0.0, 0.0);
glTexCoord2f(      ,       ); glVertex3f(3.0,2.0,0.0);
glTexCoord2f(      ,       ); glVertex3f(-1.0,2.0,0.0);
glEnd();
```

- Consider the torus given by the parametric equations:

$$\begin{aligned} x &= (R + r\cos\theta)\sin\phi, \\ y &= r\sin\theta, \\ z &= (R + r\cos\theta)\cos\phi. \end{aligned}$$

  Find formulas for texture coordinates (s,t), $0 \le s,t \le 1$, such that the center of the texture map $(0.5, 0.5)$ appear at the front of the torus.

4. *Bézier curves and surfaces*

- Represent an arc of a unit circle $x^2 + y^2 \le 0$ defined by the angle $0 \le \alpha \le 2\pi/3$ as a rational Bézier curve.
- Represent a curvilinear triangle on the plane defined by $x^2 + y^2 \ge 0$, $x \le 1$, $y \le 1$, $z = 0$, as a rational Bézier surface of bi-degree $(1,2)$.
- Find control points and weights of $1/16$ of the torus (see the parametrization formulas above) defined by inequalities: $x \ge 0$, $y \le 0$, $z \ge 0$.

4.2 Assessment rules

1 score for each group of assignments: 0 – incorrect, 0.5 – partially correct, 1 – correct answer.

References

[1] E. Angel. *Interactive Computer Graphics: a Top-Down Approach with Shader Based OpenGL.* Addison Wesley, 2011.

[2] S.R. Buss. *3-D Computer graphics. A Mathematical introduction with OpenGL.* Cambridge University Press, 2003.

[3] R.J. Rost et al. *OpenGL Shading Language*. Addison Wesley, 2009.

[4] T. Davis J. Neider and M. Woo. *OpenGL Programming Guide, The official guide for learning OpenGL.* Addison Wesley, 1994.