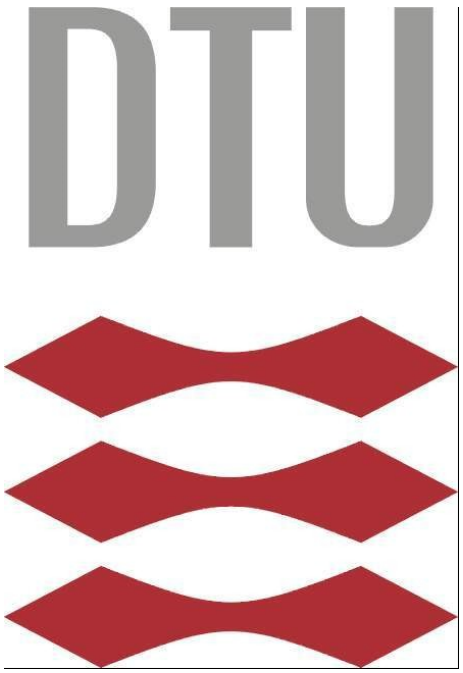
DTU 02561

Computer Graphics (fall 2013)

Metin Balaban s131105 metin.balaban@metu.edu.tr



Contents

List of Figures

Exercise 1

1

The purpose of this exercise is to give a short introduction to create, edit, link and run C/C++ programs that are based on the graphics library OpenGL, and the utility libraries Angel and GLUT.

1.1 Part 1

I have added the duties and meanings of functions as comments in source file. The extra white triangle can be seen in Figure 1.1.1.

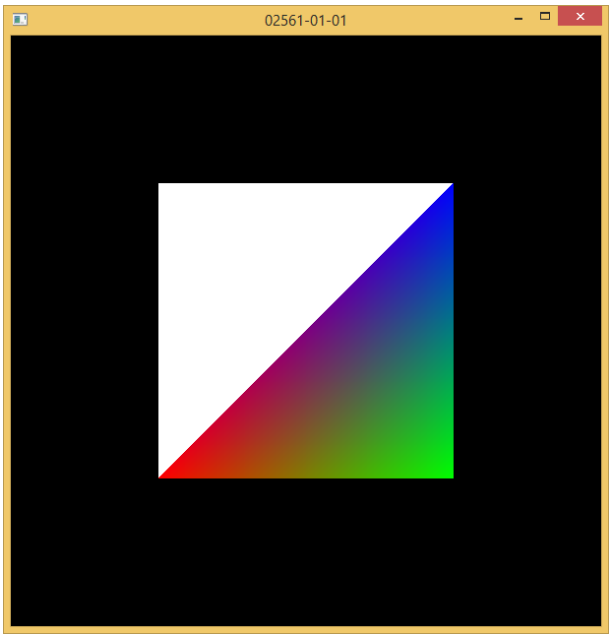


Figure 1.1.1: White Triangle

1.2 Part 2

I understand the overall structure of the program. I extend the the program with a red triangle according to given description. Then I translate the triangle with the vector (6,7,0) by modifying modelView-matrix. I define the colors of the vertices accordingly. Finally I rotate the rectangle 45 counter-clockwise around its middle point. The result is captured by a screenshot, given in Figure 1.2.1.

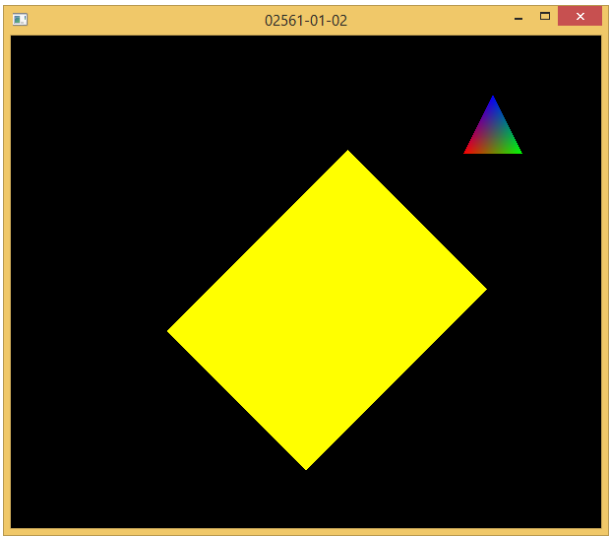


Figure 1.2.1: Resulting scene

1.3 Part 3

- glDrawArrays uses vertices in given order. Unlike glDrawArrays, glDrawArrays can use the vertices array elements in any order.
- GL_TRIANGLES: Draws triangles on screen. Every three vertices specified compose a triangle.
  GL_TRIANGLE_STRIP: Draws connected triangles on screen. Every vertex specified after first three vertices creates a triangle.
  GL_TRIANGLE_FAN: Draws connected triangles like GL_TRIANGLE_STRIP, except draws triangles in fan shape.

You can see resulting screenshot of extension in Figure 1.3.1.

1.4 Part 4

You can see resulting screenshot of extension in Figure 1.4.1

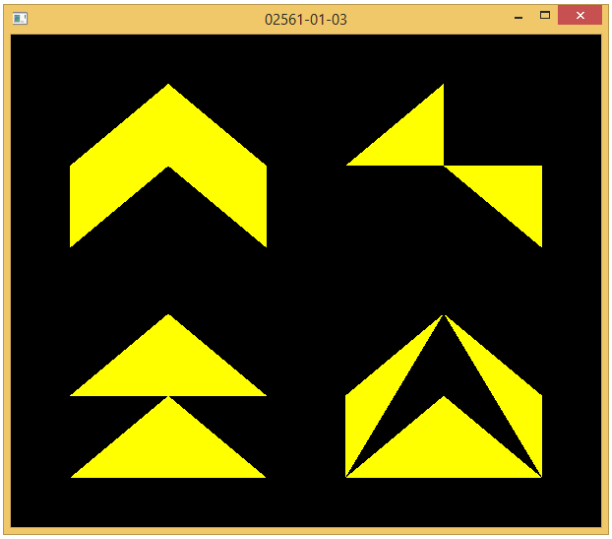


Figure 1.3.1: Yellow Triangles

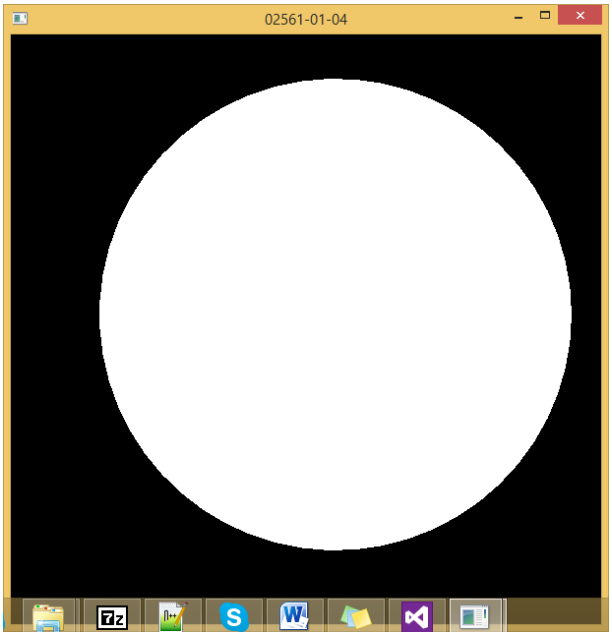


Figure 1.4.1: A (not a perfect) Circle

1.5 Part 5

Instead of changing vertex positions and colors, by making necessary model transformations (like scaling, rotating, translating...) I manage to get the result. You can see the result in Figure 1.5.1.

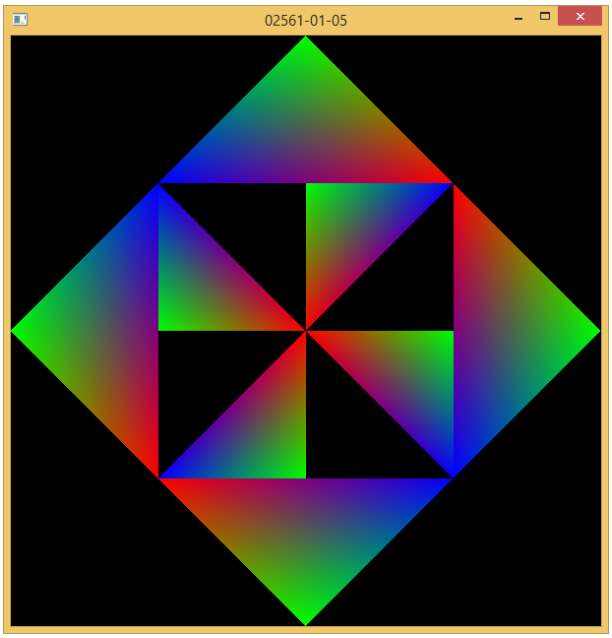


Figure 1.5.1: After eight times drawing with different model matrices, resulting scene.

1.6 Part 6

- $$M = \begin{bmatrix} \frac{xv2 - xv1}{xw2 - xw1} & 0 & -xw1 \cdot \frac{xv2 - xv1}{xw2 - xw1} + xv1 \\ 0 & \frac{yv2 - yv1}{yw2 - yw1} & -yw1 \cdot \frac{yv2 - yv1}{yw2 - yw1} + yv1 \\ 0 & 0 & 1 \end{bmatrix}$$

- $$M = \begin{bmatrix} 1 & 0 & xv1 \\ 0 & 1 & yv1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{xv2 - xv1}{xw2 - xw1} & 0 & 0 \\ 0 & \frac{yv2 - yv1}{yw2 - yw1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -xw1 \\ 0 & 1 & -yw1 \\ 0 & 0 & 1 \end{bmatrix}$$

  Multiplication of these matrices gives the same result above.

1.7 Part 7

The pictures below Figure 1.7.1, Figure 1.7.2, and Figure 1.7.3 show the result for 1, 2 and 3 iterations respectively.

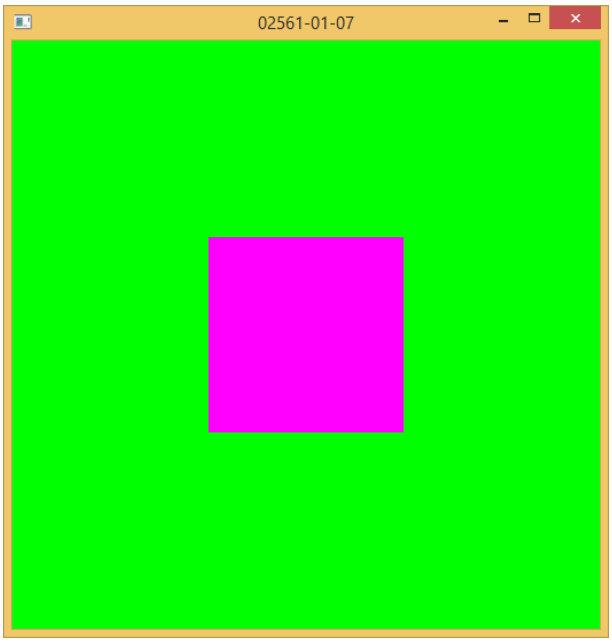


Figure 1.7.1: One iteration Sierpinski carpet

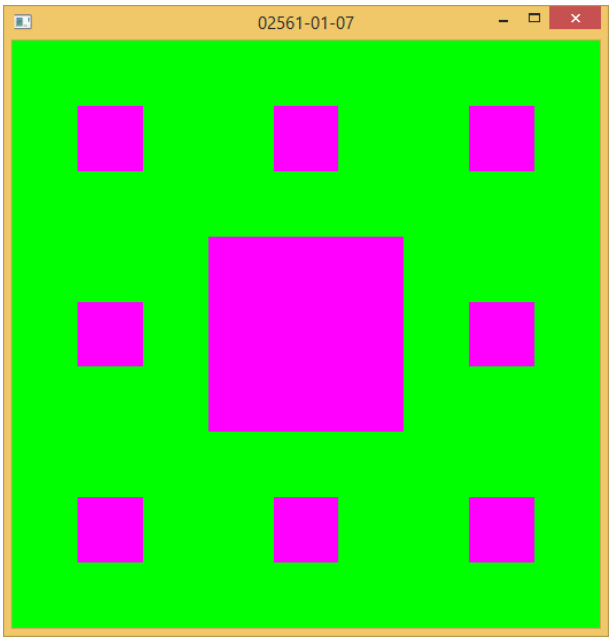


Figure 1.7.2: Two iteration Sierpinski carpet

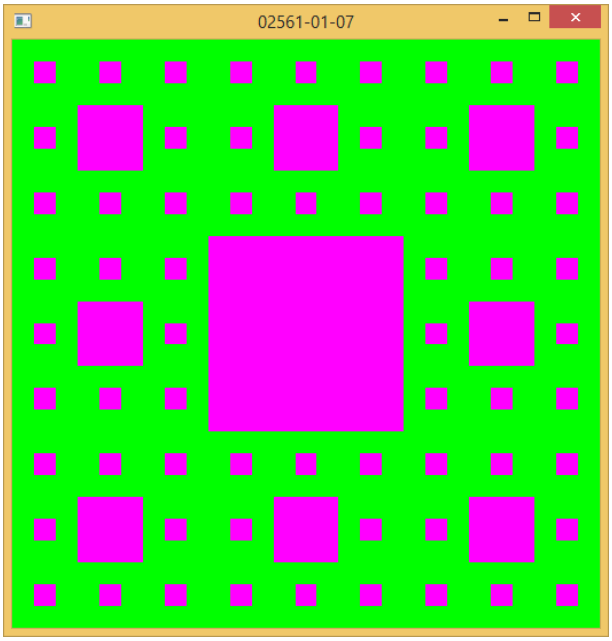


Figure 1.7.3: Three iteration Sierpinski carpet

Exercise 2

2

The purpose of this exercise is to understand the various method of setting up the virtual camera and to be able to adjust the parameters of the camera. We will get more acquainted with defining the matrices in the viewing pipeline and to concatenate them into the viewing matrix. Secondly, we will make different pictures of the scene using various projection method based on central projection (Front, X and 3-point perspective) and Orthographic parallel projection (Isometric, Dimetric, and Trimetric axonometric). Many of the principles will be demonstrated in OpenGL

2.1 Part 1

Vertex elements are considered in 2D space so their positions are represented cartesian coordinates. On the other hand, in this exercise, vertex elements are in 3D space and their positions are represented in homogenous coordinates, which are vec4.

In display function, we describe model-view properties of our vision (camera). In this exercise, it is defined with triplet of eye, at, up. LookAt() function, which takes an eye position, a position to look at, and an up vector, all in object space coordinates computes the inverse camera transform according to its parameters and multiplies it onto the current matrix stack.

2.2 Part 2

- You can see resulting Figure 2.2.1.
- Translation, rotation around Y axis, and Scaling
- Model matrix is similar with this:
  Translate(...)*RotateY(...)*Scale(..)

Matrix multiplication is NOT a commutative operation so the order is *important*.

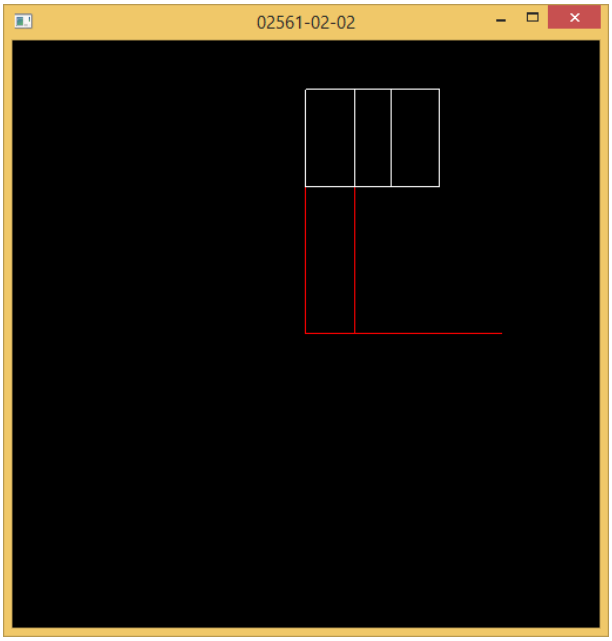


Figure 2.2.1: After three transformations

2.3 Part 3

After following descriptions in exercise text, I reach the expected result. You can see front perspective screenshot in Figure 2.3.1.

2.4 Part 4

I get the isometric view (see Figure 2.4.1) by using this LookAt call as a View matrix :

LookAt(vec3(5,5,5), vec3(0,0,0), vec3(0,0,1))

The idea is selecting the eye point on diagonal line of the cube.

2.5 Part 5

I make the necessary key assignments.

- Series of transformations for key 2 is
  *RotateY(-atan(0.25)/DegreesToRadians)*Translate(vec3(+.5, -1, -6))*

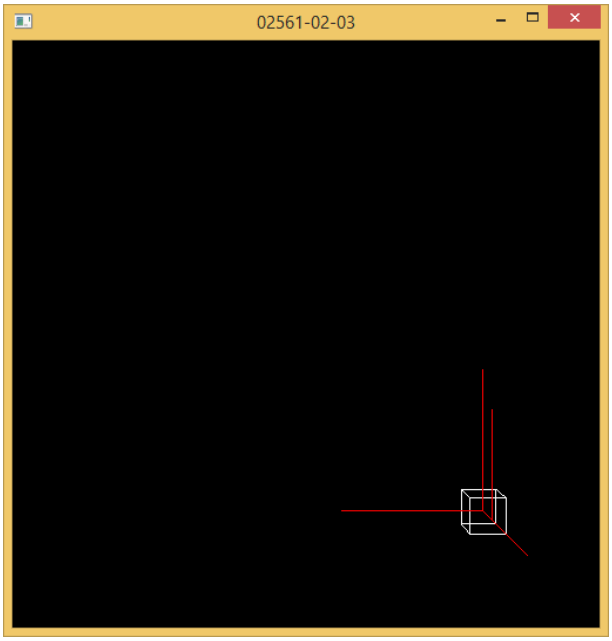


Figure 2.3.1: Front Perspective

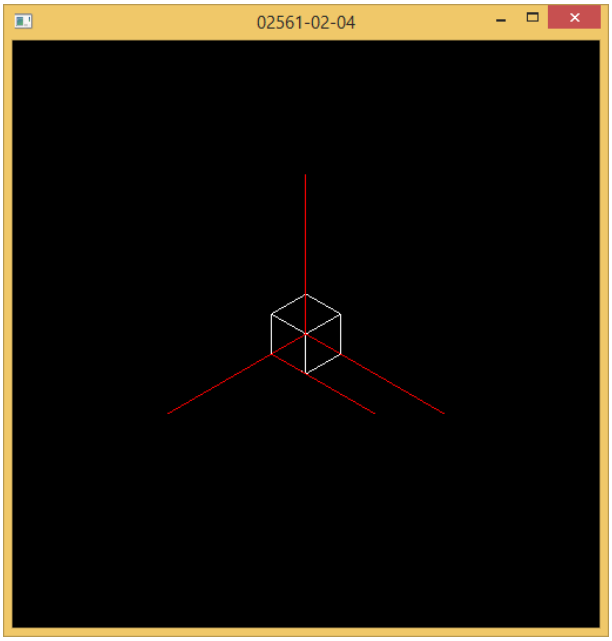


Figure 2.4.1: Isometric View

- Corresponding lookAt function for key 4 is
  LookAt(vec3(4, 1, 1), vec3(4, 1, 1) + vec3(-sqrt(3), 0, 1), vec3(0,1,0))
- The hardcoded matrix used in key 6 is
  mat4 (1,0,0,0, 0,1,0,0, 0,0,1,0, 1,-1,-9,1)

  To check the similarities of key couples 1-2, 3-4, and 5-6 please execute the program.

2.6 Part 6

- *modelView=RotateX(atan(1.0/2)/DegreesToRadians)* Translate(0,-3,-6);*
- *modelView=Translate(0,3,0)*RotateY(30)*Scale(2,2,2);*

Exercise 3

3

In this assignment, I complete every task given in exercise, including the bonus part. The resulting program has the abilities that are expected to perform in Part 1, Part 2 and Part 4 (bonus). The report includes some screenshots of implemented functionalities. For further details, please execute the program itself.

3.1 Part 1

You can see the screenshots of different blending levels in Figure 3.1.1.

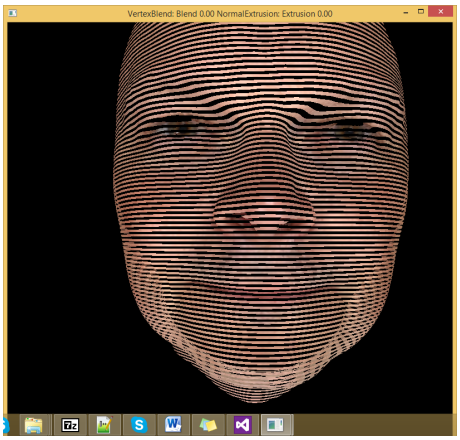
3.2 Part 2

You can see the screenshots of different normal extrusion levels in Figure 3.2.1.

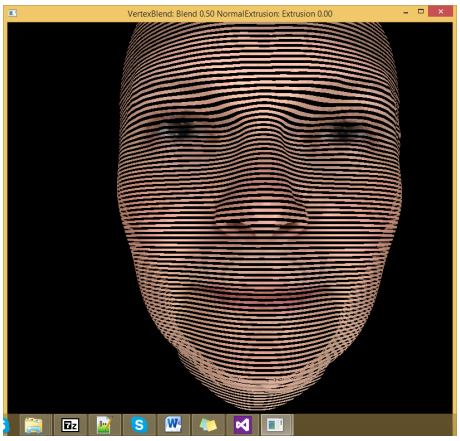
3.3 Part 3

I merge answers of the second, third, and fourth questions and give a mega answer.

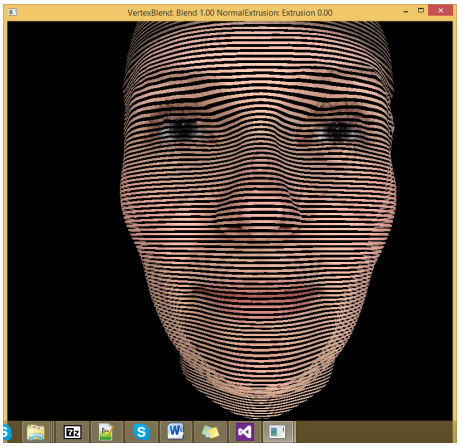
- Vertex attributes are specific to that vertex: each attribute might differentiate from others. Unlike vertex attributes, uniform vertex variables are the same for each vertex in model. They can be changed in any callback function.
- A vertex shader will transform the representation of a vertex location from whatever coordinate system in which it is speci?ed to a representation in clip coordinates for the rasterizer .Each invocation of the vertex shader outputs a vertex that then goes through primitive assembly and clipping before reaching the rasterizer. The rasterizer outputs fragments



(a) Blend Ratio: 0.0

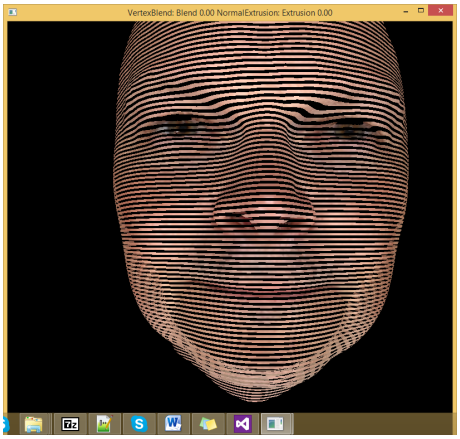


(b) Blend Ratio: 0.5

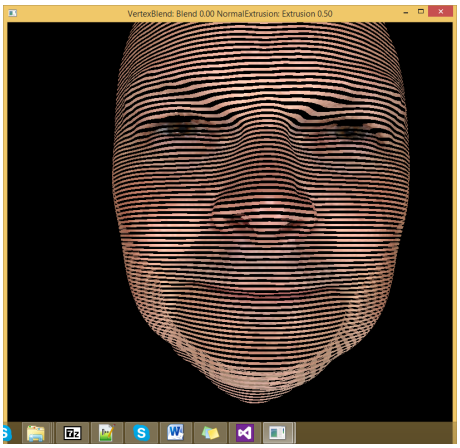


(c) Blend Ratio: 1.0

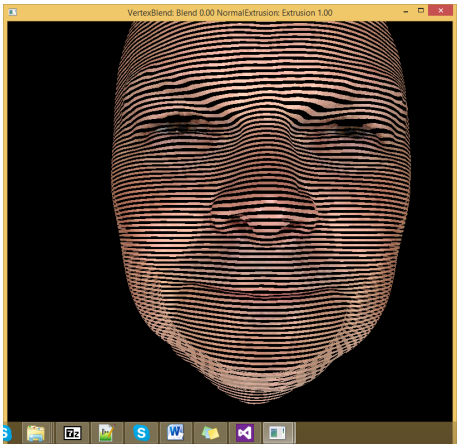
Figure 3.1.1: Different Blend Ratios



(a) Normal Extrusion: 0.0



(b) Normal Extrusion: 0.5



(c) Normal Extrusion: 1.0

Figure 3.2.1: Different Normal Extrusions

for each primitive inside the clipping volume. Each fragment invokes an execution of the fragment shader. Fragment shader assigns colors to fragments.

- Yes it is possible. We need to output two color vectors from vertex shader: color1 and color2. By using uniform *blendingValue* we can blend two colors in fragment shader. Texturing is applied in fragment shader generally. This is because blending in fragment shader sounds more natural.

3.4 Part 4

As you see in the Figures 3.1.1 and 3.2.1, I make the discard operation successfully. The two lines of code that write in fragment shader is

```
if(mod(round(glPosition[1]),2.0)==0)
        discard;
```

Exercise 4

4

The purpose of this exercise is to get acquainted with lighting and shading in OpenGL and GLSL.

4.1 Part 1

I succeed with the part by following the suggested order given. The key point of this part is to consider that the *lightPosition* in the shader is in eye-coordinates.
The result can be seen in Figure 4.1.1.

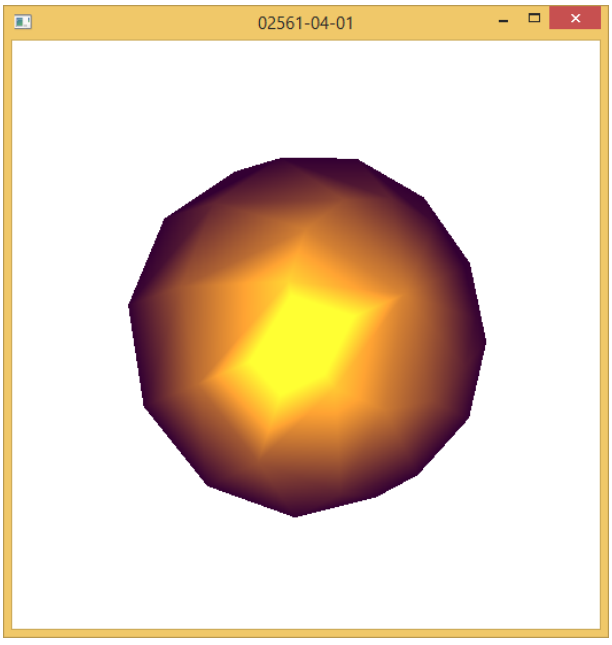


Figure 4.1.1: Gouraud Shading

4.2 Part 2

I succeed with the part by following the suggested order given. User can toggle between directional and point light model by pressing *L* key on keyboard. The result with directional light can be seen in Figure 4.2.1. On the other hand, the result with point light source can be seen in Figure 4.2.2.
Shininess is set to 20 in this part.

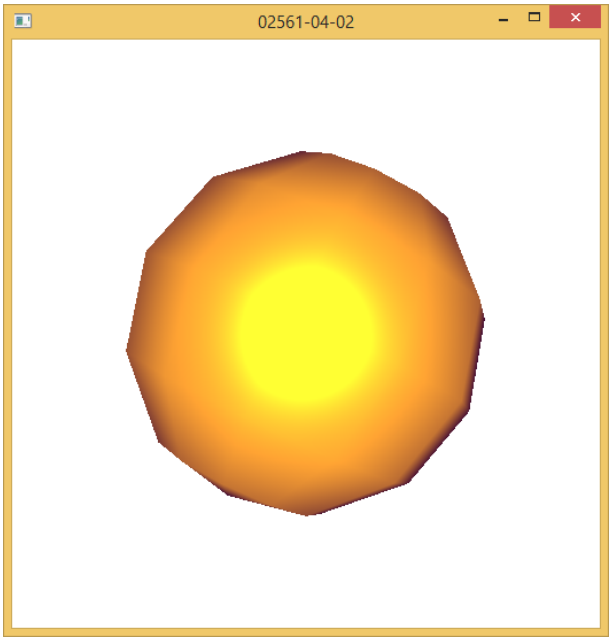


Figure 4.2.1: Directional Light Source

4.3 Part 3

- Phong shading is a appliation of phong lightning. If you consider each pixels' normal during phong lightning calculation, it is phong shading.
- If you assign each polygon defined by vertices a normal, it is Gouraud's method. Calculations are done before rasterizing. If you assign each pixel a normal, it is Phong's method. Calculations are done after rasterizing (fragment shading). Gouraud's method is deceptively simple and Easy to compute as against Phong shading.
- Unlike point light, rays are parallel to each other in directional light.
- Yes. to make an analogy, the more rays your eyes reached to, the brighter you see where you look at.

- Surfaces looks dull without it, somewhat like chalk. What we are missing are the highlights that we see re?ected from shiny objects.
- As shininess is increased, the re?ected light is concentrated in a narrower region centered on the angle of a perfect re?ector.
- Instead of computing reflection for every point on surface, we have replaced r*v component with n*h, where r is reflection and h is halfway vector. It is a succesful and efficient technique when used on flat or curved surfaces.
- In some cases, i.e. when modelView matrix makes non-uniform transformations, modelView matrix cannot be used to convert normal vectors in object space to eye space coordinates. In such cases, normal matrix is used. It is equal to $(M^{-1})^T$ where is upper left 3x3 matrix of modelView matrix.
- Eye space. Otherwise eye position dependent effects, such as specular lights would be harder to implement.

4.4 Part 4

I implement and complete part 5. Therefore I complete the objectives in Part 4. I add three light sources in positions $(0.0, 0.0, -1.0)$, $(-3.0, -3.0, -1.0)$, and $(3.0, 3.0, -1.0)$.

The reader should follow the Part 5 to learn more about multiple light sources.

4.5 Part 5

Material color (ambient, diffuse and specular) is changed into a fragment shader uniform. Light color (ambient, diffuse and specular) is changed into a fragment shader uniform. One diffuse and specular color is assigned to each light position. Ambient-product, diffuse-product and specular-product is computed in the fragment shader and removed as a shader-uniform. I set the light diffuse color to be red, green and blue.
I set the shininess value to 100.
You can see resulting screenshot in Figure 4.5.1.

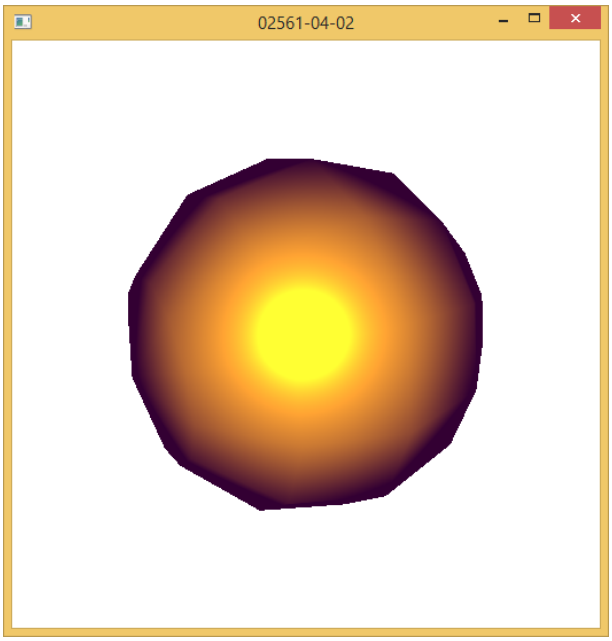


Figure 4.2.2: Point Light Source

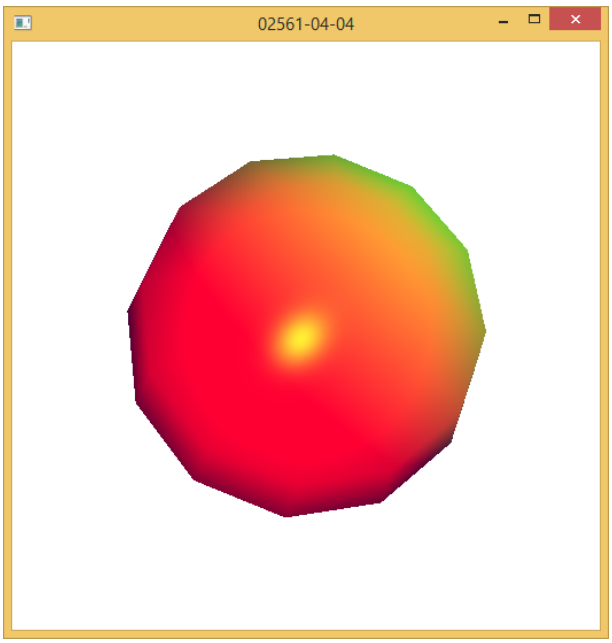


Figure 4.5.1: Three point light source with different colors

Exercise 5

5

The purpose of the exercise is to get acquainted with input and window management in Glut and OpenGL. In particular we will work with mouse input, keyboard input, menus, multiple windows, logic operations, selection and picking.

5.1 Part 1

According to mouse position information, pick function makes decision of the action that will be carried out. Those actions are draw-triangle, draw-line, draw-rectangle, draw-point and drawing-mode.

5.2 Part 2

- When we clicked on a selectable item, picking function binds FBO objects and renders the scene on the FBO instead of default framebuffer.
- According to the mouse position when clicked, selected item information is extracted from FBO.
- Now we know which object is selected. The program flow goes on. We can do anything we want with selected object.

5.3 Part 3

According to the recipe described in exercise sheet, I implement the Circuit diagram editor. I want to give my implementation details before giving a screenshot of a simple diagram that I create.
Move function can be used by user intuitively. User should left-click and hold on the object to be moved and should release the button when he or she has dragged the object to the desired place. Simply press on-hold left click,drag,release left-click.

Scale function can also be used intuitively. Once you click on an circuit item, if you move cursor further from the center of the item, item enlarges. On the other hand, if you move cursor closer to the center of the item, item shrinks. Rotating may be a little bit tricky for the user at first glance. The mechanism works in this way: when user left-clicks o an item and holds on the button, if user moves cursor further from the center of the object, object rotates counter-clockwise around its center. If user moves the cursor further enough, the item makes a full 360 degrees turn. Therefore, user can give any angle to the item by rotating the item enough in counter-clockwise direction.
You can see a sample circuit diagram in Figure 5.3.1

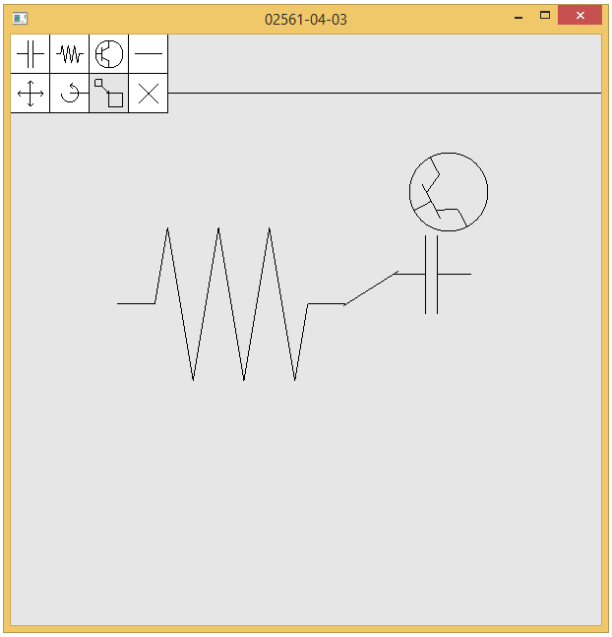


Figure 5.3.1: A sample circuit diagram

Exercise 6

6

The purpose of the exercise is to understand the principles of 2D texture mapping and how it can be used for polygon meshes. Furthermore, the purpose of the exercise is to understand the process of hidden surface removal and back-face culling

6.1 Part 1

Subpart a

You can see the scene when hidden surface removal is disabled and enabled in Figure 6.1.1 and Figure 6.1.2. When hidden surface removal is enabled, the cube and left half of small polygon becomes invisible (we cannot see the difference occuring in small polygon because both big and small polygon is green).

Subpart b

When GL_BACK mode is activated (Figure 6.1.3a) , small polygon disappears. When GL_FRONT mode is enabled (Figure 6.1.3b), big polygon disappears. When GL_FRONT_AND_BACK mode is activated (Figure 6.1.3c), small and big polygon and the cube disappears.

Subpart c

If the angle between normal vector of a primitive and camera view direction is greater then 90 degrees, then the primitive is front facing. Otherwise, it is back facing.

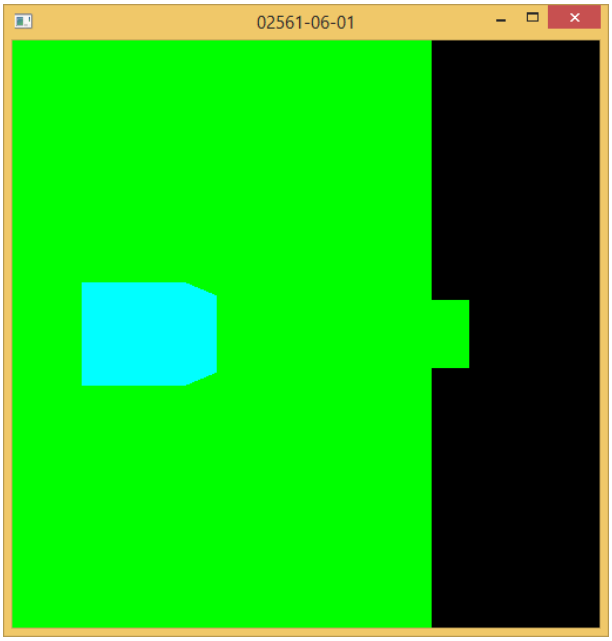


Figure 6.1.1: Hidden surface removal is disabled

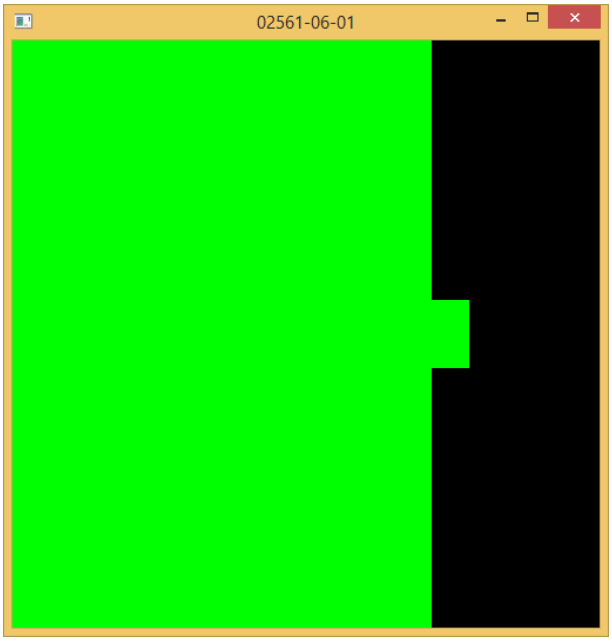
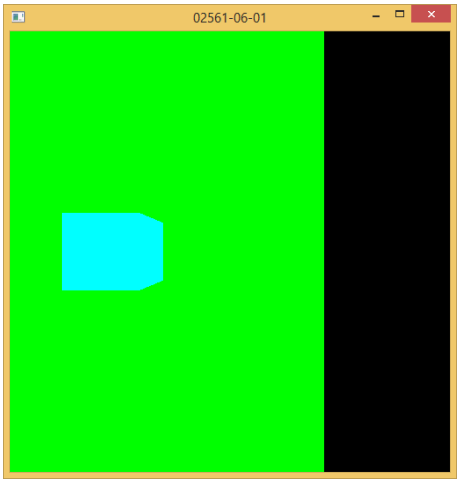
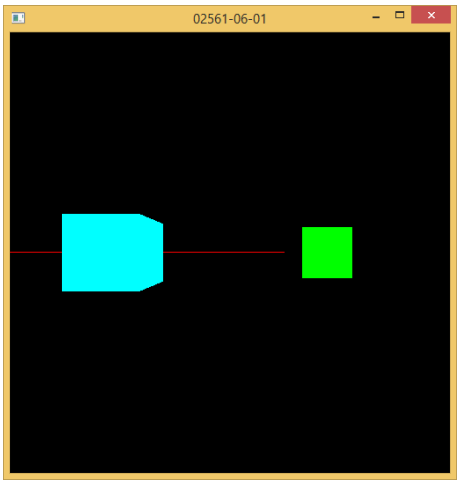


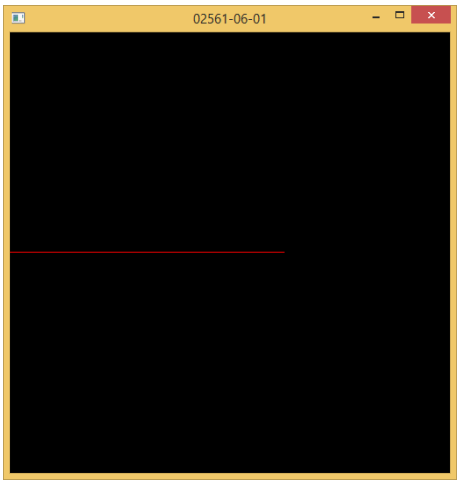
Figure 6.1.2: Hidden surface removal is enabled



(a) GL_BACK



(b) GL_FRONT



(c) GL_FRONT_AND_BACK

Figure 6.1.3: Different face culling modes

6.2 Part 2

I use the projection matrix given below.

```
Perspective(24.3,WINDOW_WIDTH/(float)WINDOW_HEIGHT, 0.01, 10)
```

You can see resulting view of the scene in Figure 6.2.1.

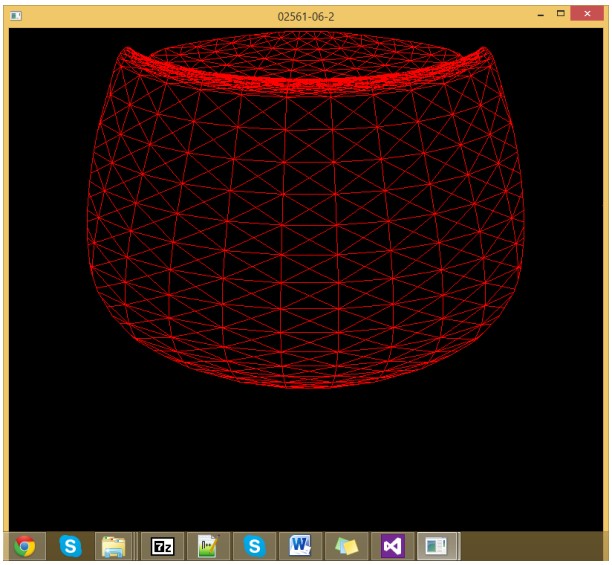


Figure 6.2.1: Parts are clipped

6.3 Part 3

I implement keyboard functions which alters magnification and minification filters. You can see keyboard key mappings below.
Key 1 : Magnification filter is set to linear.
Key 2 : Magnification filter is set to nearest.
Key 3 : Minification filter is set to nearest mipmap nearest.
Key 4 : Minification filter is set to nearest mipmap linear.
Key 5 : Minification filter is set to nearest.
Key 6 : Minification filter is set to linear.
Key 7 : Minification filter is set to linear mipmap nearest.
Key 8 : Minification filter is set to linear mipmap linear.
Minification filter can be adjusted to six different modes. Magnification filter can be adjusted to two different modes. Therefore, for a single 2D texture, there are *12* different combination of filters. Giving a sample and a screenshot for each combination in the report would spoil the flow of the report. This is

because I prefer to give one example screenshot from these combinations see (Figure 6.3.1). To see all possibilities, please navigate to program executable.

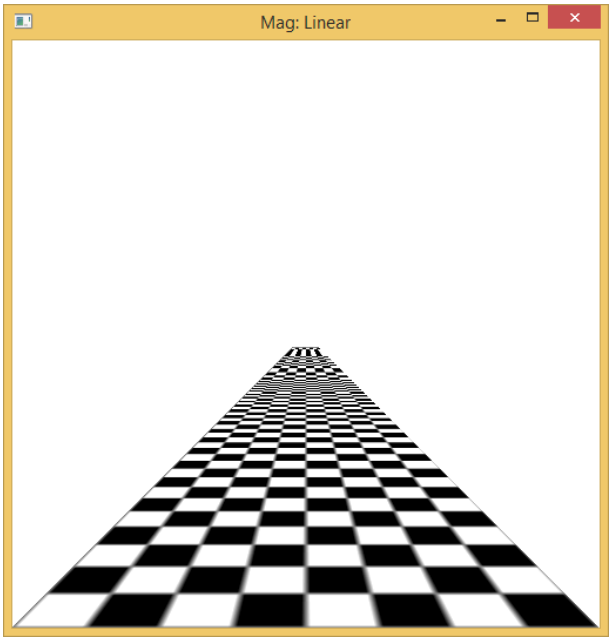


Figure 6.3.1: Magnification and minification filters are both linear

6.4 Part 4

You can see all possible texture wrapping options below.

6.5 Part 5

```
vTextureCoord =(textureTrans *vec4(textureCoord,0.0,1.0)).xy;
```

This code piece is fetched from vertex shader. *textureTrans* uniform matrix is multiplied with the extended vector of texture coordinates. Since *textureTrans* is uniform, we can modify texture mapping in vertex shader.

6.6 Part 6

I set *idle* function as *glutIdleFunc*. This is my implementation of *idle* function:

```
void idle()
{
        delta+=0.005;
        glutPostRedisplay();
```

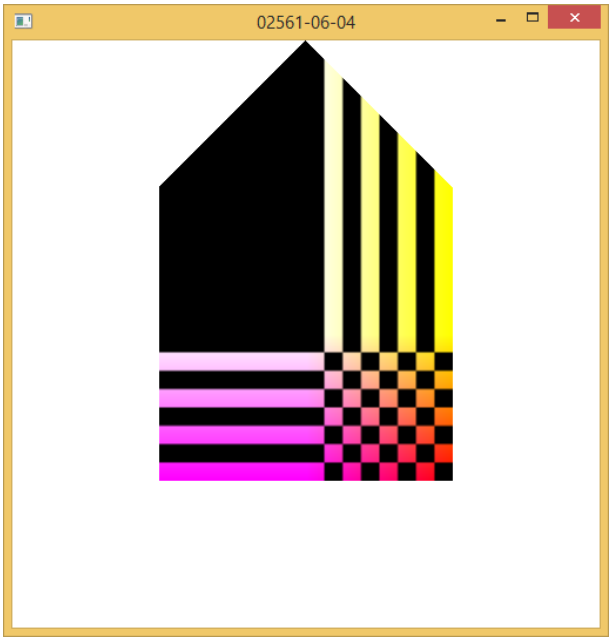


Figure 6.4.1: GL_TEXTURE_WRAP_T, GL_REPEAT

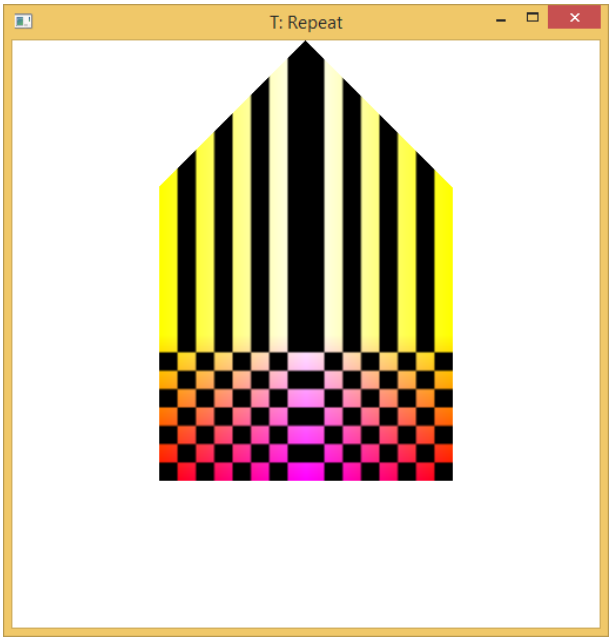


Figure 6.4.2: GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT

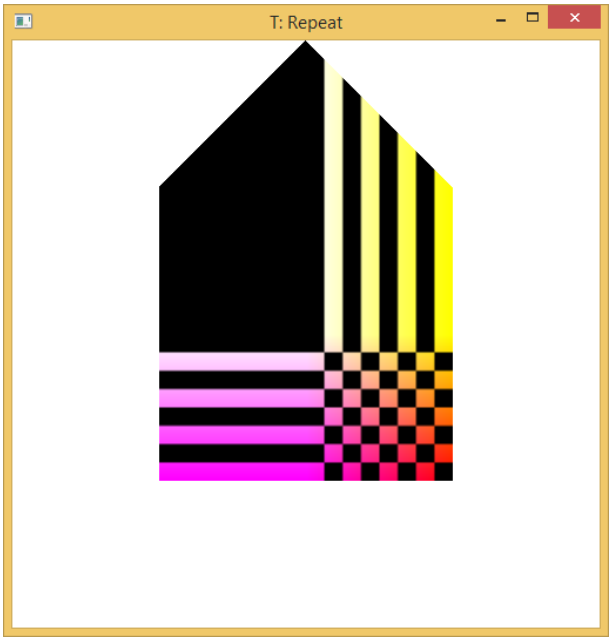


Figure 6.4.3: GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE

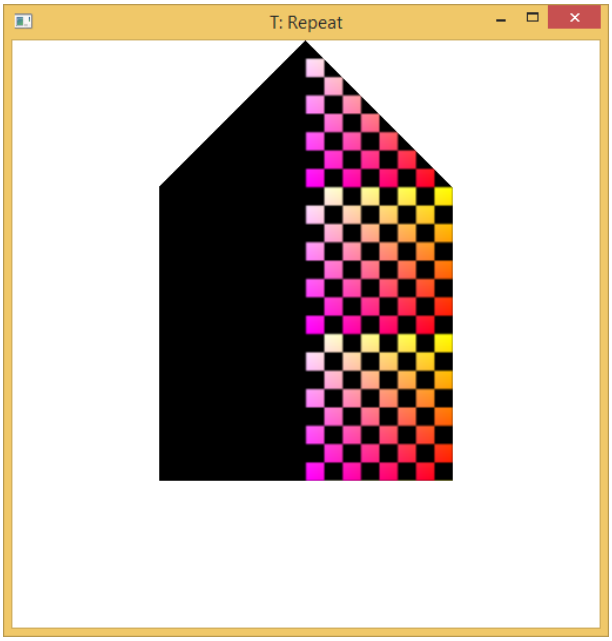


Figure 6.4.4: GL_TEXTURE_WRAP_S, GL_REPEAT

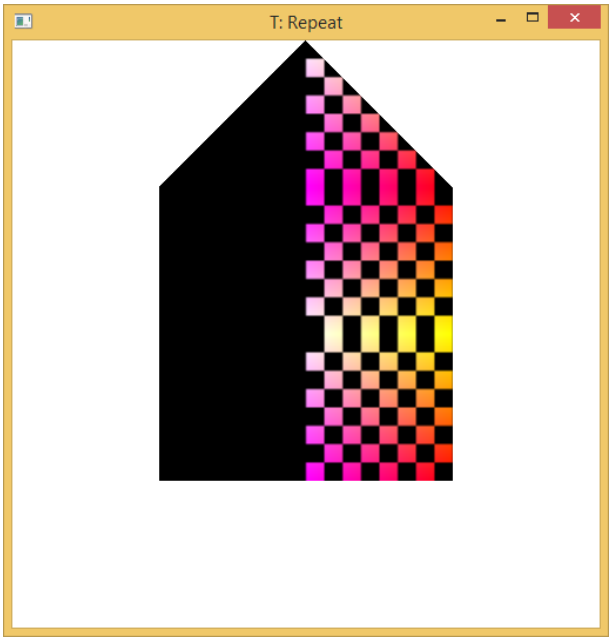


Figure 6.4.5: GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT

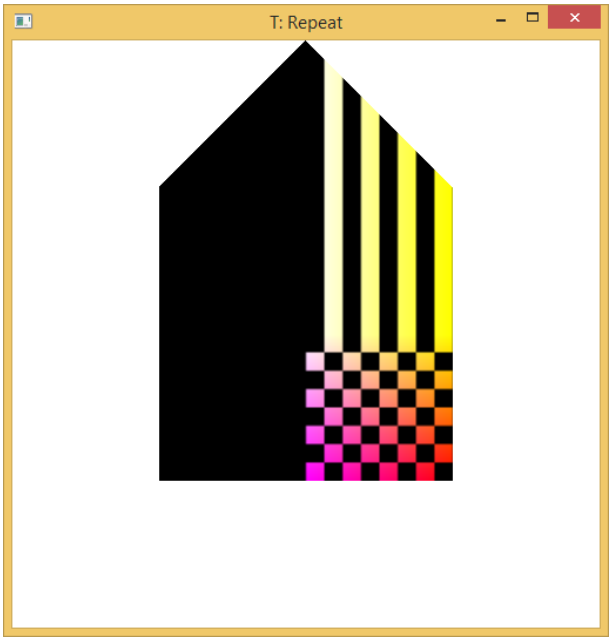


Figure 6.4.6: GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE

```
}
```

The *delta* float variable increases in each callback of idle function.
texturesTrans uniform matrix is defined as:

```
mat4 textureTrans=Translate(0.5, 0.5, 0.0) * RotateZ(delta)
* Translate(-0.5, -0.5, 0.0);
```

It means textureTrans matrix is updated and changed periodically. In this way, I have an animation. To have a look at the animation that I create, please navigate to the executable program.

Exercise 7

The purpose of the exercise is to use OpenGL to produce simple shadows using projection matrices and to get a better understanding of the output pipeline. We are only are concerned with generating the shadows - this means that using Phong lighting is an optional extension.

7.1 Part 1

Part a

You can see my *createShadowProjectionPointLight* implementation below.

```
mat4 createShadowProjectionPointLight(){
        mat4 m;

        m[3][0] = -1.0/(lightPos[0]-shadowDistance);
        m[3][3] = 0;
        mat4 shadow=Translate(lightPos[0],lightPos[1],
         lightPos[2]) * m * Translate(-lightPos[0],
         -lightPos[1],-lightPos[2]);
        return shadow;
}
```

The resulting scene can be seen in Figure 7.1.1.

Part b

You can see my *createShadowProjectionDirectionalLight* implementation below.

```
mat4 createShadowProjectionDirectionalLight() {
        mat4 m;
```

```
        vec4 newlightPos=10000*lightPos;
                //drag the light source far
                //far away without changing
                //jts direction.
        m[3][0] = -1.0/(newlightPos[0]-shadowDistance);
        m[3][3] = 0;
        mat4 shadow=Translate(newlightPos[0],newlightPos[1],
        newlightPos[2]) * m * Translate(-newlightPos[0],
        -newlightPos[1],-newlightPos[2]);
        return shadow;
}
```

When point source is so far away from the object, the system resembles to a directional light system because the rays coming from far point source become almost parallel. I use this fact to implement directional light source shadows.

The resulting scene can be seen in Figure 7.1.2.

Part c

I add the functionality with this simple code fragment:

```
        case 'd':
                if(shadowDistance==-4)
                        shadowDistance=-8;
                else
                        shadowDistance=-4;
                break;
```

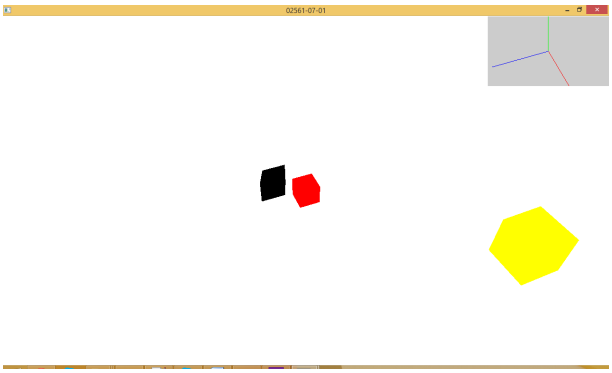


Figure 7.1.1: Point light source shadow

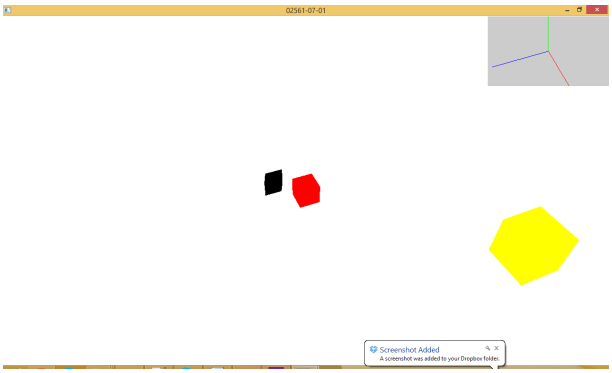


Figure 7.1.2: Directional light source shadow

Exercise 8

8

The purpose of this exercise is to understand and implement shadow mapping. This includes a deep understanding of the different coordinate spaces in the pipeline as well as mapping between these coordinate spaces.

8.1 Part 1

Implementations of the methods *getLightProjection()* and *getLightView()* are below.

```
mat4 getLightProjection() {
        float d=length(teapotPosition-lightPosition);
        float theta = asin(teapotBoundingRadius/d)
                    *RADIAN_TO_DEGREE;
        mat4 perspective = Perspective(2*theta, 1, 0.1, 400.);
        return perspective;
}

        return LookAt(lightPosition,
                teapotPosition, vec3(0,1,0));
```

8.2 Part 2

After following the description given in exercise document, I render the teapot on shadow map. A screenshot of resulting scene can be seen in Figure 8.2.1

8.3 Part 3

I compute normalized device coordinated. I transform and map this coordinates to shadow map coordinates, ranging from $(0,0)$ to $(1,1)$. I make the transformation by multiplying normalized device coordinates with bias

matrix. The visualisation of shadowUV instead of texture lookup in shaders results in the scene presented in Figure 8.3.1.

8.4 Part 4

I use these code fragment in fragment shader to decide whether a fragment is in shadow or not:

```
vec4 blue= texture(texture1, vTextureCoordinate);
vec4 colors=texture(texture2, shadowUV);
if(colors.x<1)
        fragColor=colors;
else
        fragColor = blue;
```

The resulting scene can be seen in Figure 8.4.1.

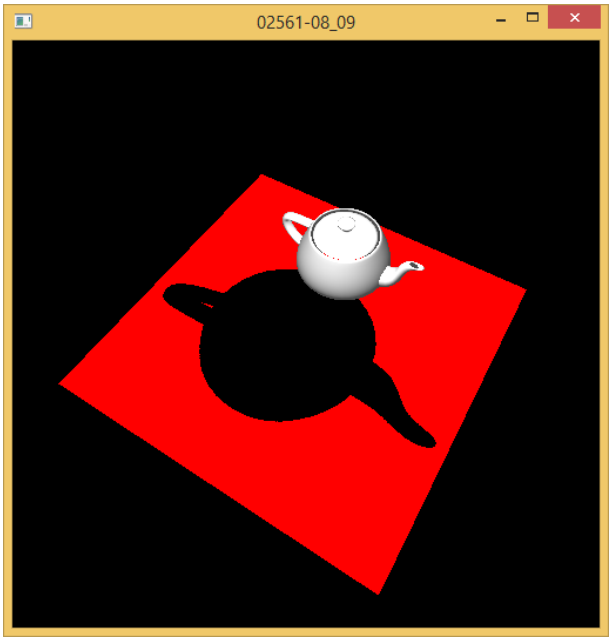


Figure 8.2.1: Rendering teapot on shadow map

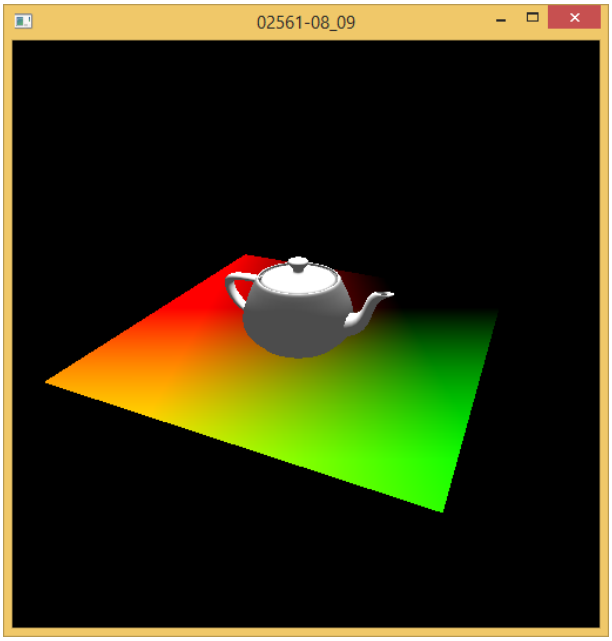


Figure 8.3.1: Visualisation of shadowUV on plane

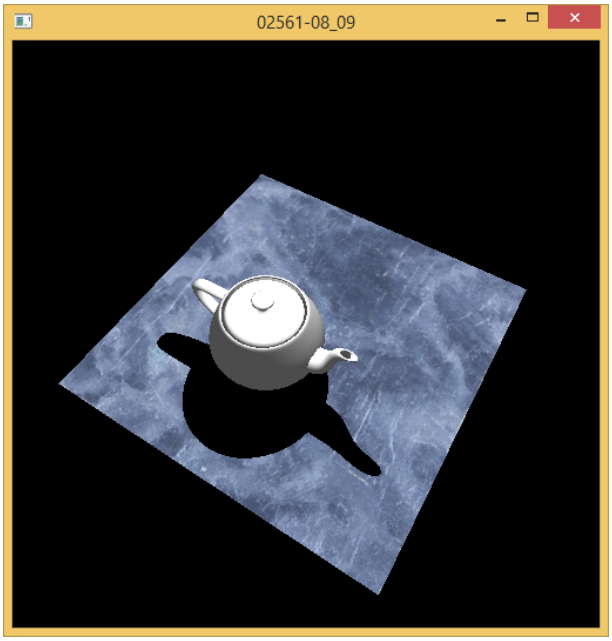


Figure 8.4.1: The final shadow map texture

Exercise 9

9

The purpose of this exercise is to understand and implement planar reflections using OpenGL.

9.1 Part 1

Completion of this part is considered as a future work.

9.2 Part 2

When I comment out the line that renders the plane in the *drawPlane()* function, the resulting scene is Figure 9.2.1.

9.3 Part 3

By following the steps described in exercise text document, I manage to display a reflected teapot in the scene (see Figure 9.3.1).

9.4 Part 4

I manage to reflect the light by multiplying y component of light position by minus one before rendering teapot and roll all changes back after rendering the reflection.

A screenshot of the scene after this implementation can be seen in Figure 9.4.1

9.5 Part 5

Firstly, I make the reflection teapot a little bit darker by multiplying its color with 0.5 and I roll the change back after rendering the reflection teapot. After

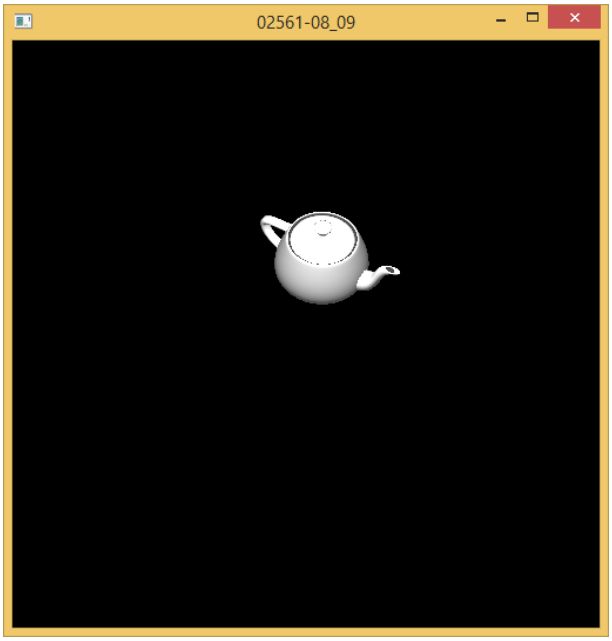


Figure 9.2.1: The scene without the plane being rendered

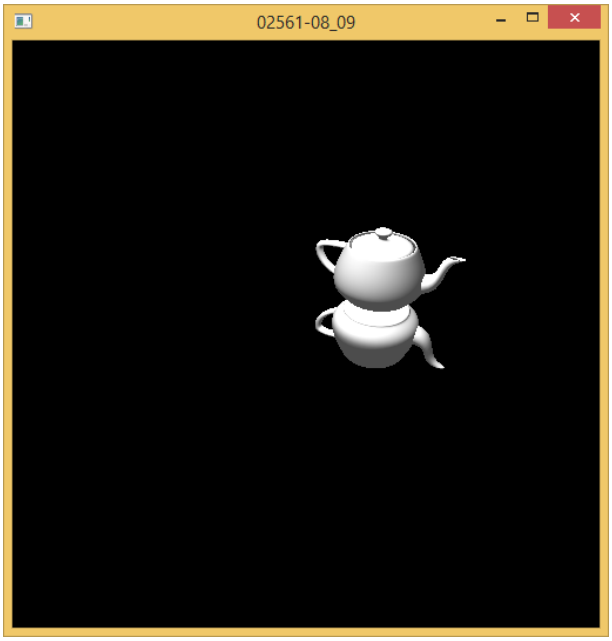


Figure 9.3.1: A teapot with a reflected teapot below

that, I enabled displaying the plane and blend the reflection with the plane. The blending process is carried out by this code fragment:

```
glEnable (GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
mat4 model;
drawMeshObject(projection, model, view,
    getLightViewProjection(), planeObject);
    //renders the plane
glDisable(GL_BLEND);
```

The resulting scene can be seen in Figure 9.5.1.

However, there is a problem while rendering the teapot reflection. Parts of the reflection teapot that is above the plane should be cut out the scene. You can see a screenshot of the problem in Figure 9.5.2.

The solution of the problem is introduced in Part 6.

9.6 Part 6

Completion of this part is considered as a future work.

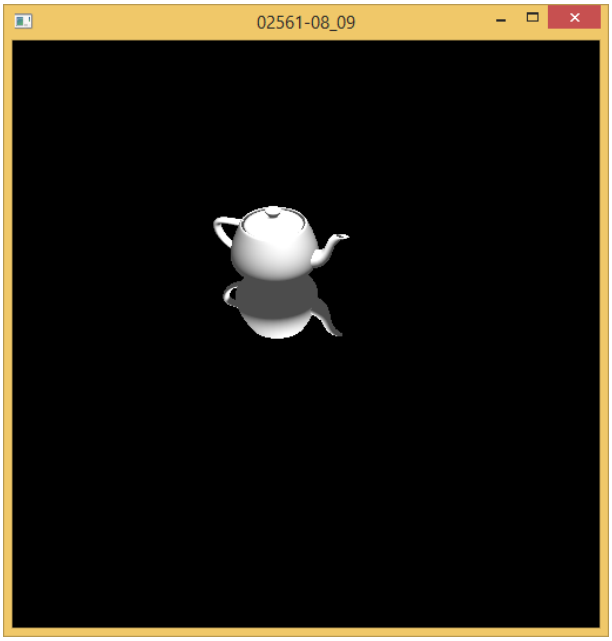


Figure 9.4.1: A teapot and its reflection with correct illumination

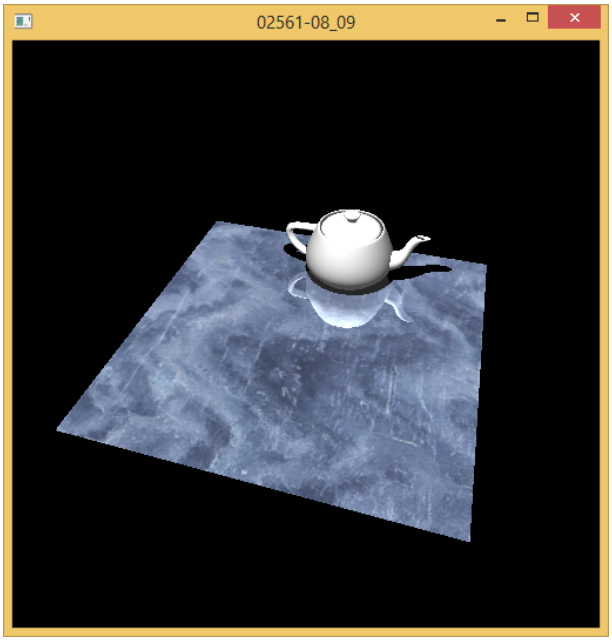


Figure 9.5.1: The teapot and its reflection on the plane

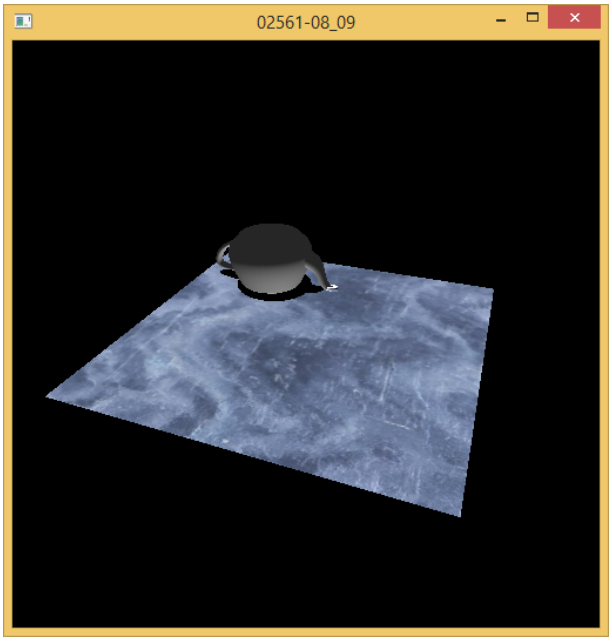


Figure 9.5.2: The planar reflection without introducing stencil buffer

Exercise 10

10

The purpose of this exercise is to become familiar with the concepts behind environment mapping and bump mapping. We will use environment mapping to simulate mirrors and glass and in the process learn to use cube maps, reflection/refraction functions and Schlick's approximation. Finally we will apply bump mapping to a sphere to add small scale details.

10.1 Part 1

After following descriptions described in exercise description document, I build the skymap. A screen capture of executing program can be seen in Figure 10.1.1.

10.2 Part 2

After following descriptions described in exercise description document, I manage to implement mirror shader. You can see a screenshot of this shader in Figure 10.1.1.

10.3 Part 3

After following descriptions described in exercise description document, I manage to implement glass shader. To get fresnel term, which is needed in shader calculations, I use schlick's approximation. You can see the GLSL code fragment that calculates fresnel term below.

```
float R0 = pow(abs((air-glass)/(air+glass)),2.0);
float R = R0 + (1-R0)*pow(1.-max(dot(n, -i),0.), 5);
```

where *air* is refracting index of air, *glass* is refracting factor of index, n is normal vector of the fragment (pixel), i is incident vector of the fragment.

You can see a screenshot of this shader in Figure 10.3.1.

Figure 10.1.1: Skybox and sphere with mirror shader

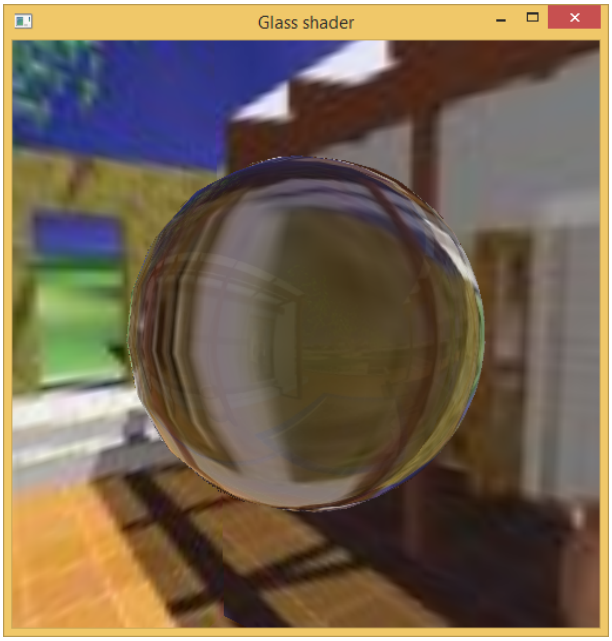


Figure 10.3.1: Glass shader

10.4 Part 4

After following descriptions described in exercise description document, I manage to implement a bump-map on our sphere.

You can see a screenshot of this shader in Figure 10.4.1.

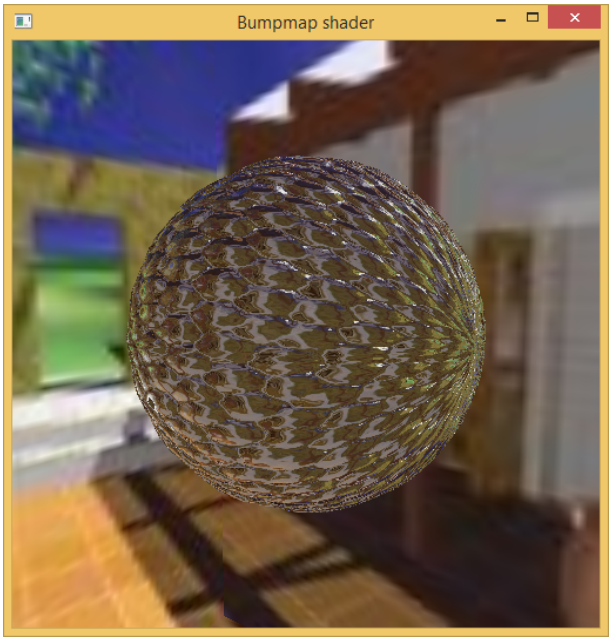


Figure 10.4.1: Bump-mapping on a sphere

10.5 Part 5

To simulate this, do three lookups in the cube map, one for red, green, and blue, each with different index of refraction, when computing the refraction color. The refraction indexes of the colors in glass are given as:

$n_{red} = 1.56$
$n_{green} = 1.615$
$n_{blue} = 1.63$
$n_{glass} = 1.615$ (used in calculation of fresnel term)

The resulting screenshot can be seen in Figure 10.5.1.

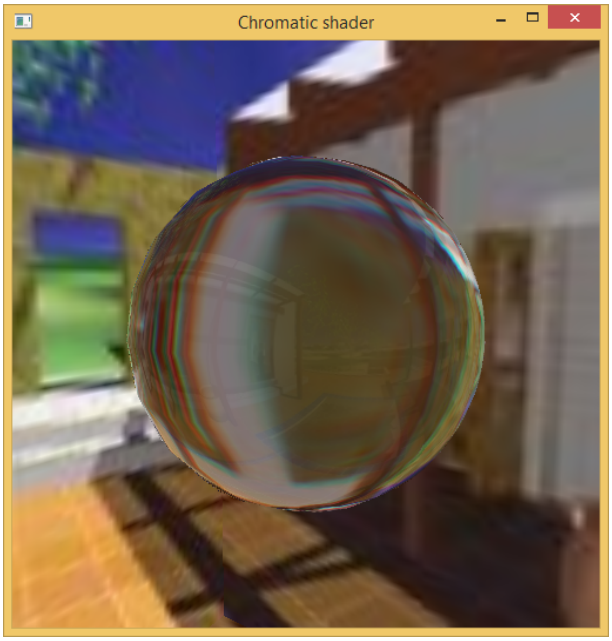


Figure 10.5.1: Chromatic glass shader

Exercise 11

11

The purpose of the exercise is to understand the theory behind parametric curves and surfaces and how to use this theory to create geometric models. The theory includes Bezier, uniform B-spline, non-uniform B-spline and NURBS (non-uniform rational B-spline) curves and surfaces. The advantage of using a parametric representation of curves and surfaces is that they can be represented using only a few control points instead of a detailed mesh.

11.1 Part 1

After following descriptions given in exercise description document, I manage to implement cubic Bezier curve.

You can see a screenshot of a cubic Bezier curve in Figure 11.1.1.

11.2 Part 2

I carry out the part by following the description given in the text and using the knot vector given. A screen capture of the circle can be seen in Figure 11.2.1.

11.3 Part 3

Item 1

Linear interpolation is already implemented. It is based on simple line segment drawing. A screenshot of these line segments can be seen in Figure 11.3.1.

Item 2

I draw a Bezier curve in order 4 by using first 4 control points. The knot vector that is used in this item is

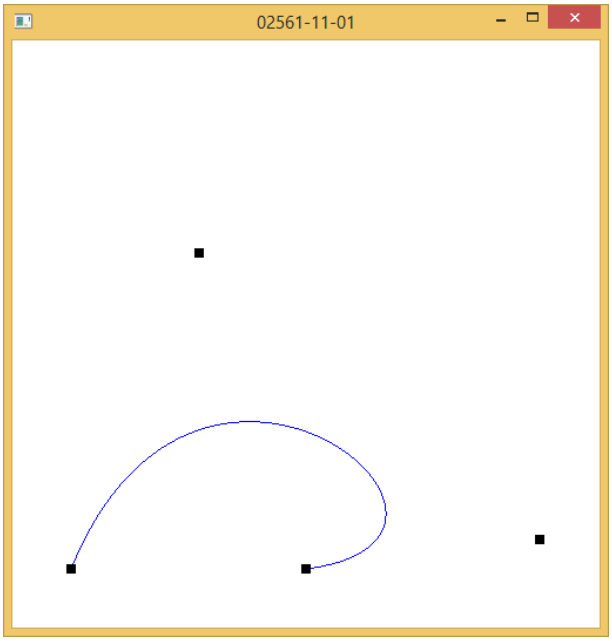


Figure 11.1.1: A cubic Bezier curve

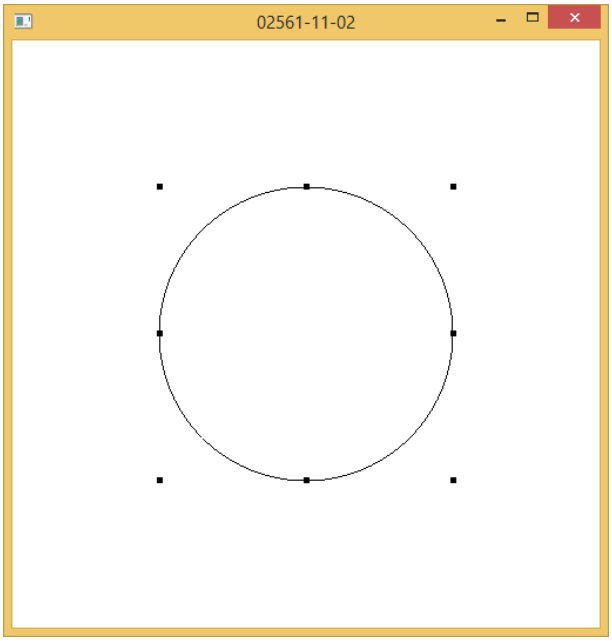


Figure 11.2.1: The exact circle created by Bezier curves and by using NURBS

$\{0, 0, 0, 0, 1, 1, 1, 1\}$

A screenshot of these line segments can be seen in Figure 11.3.2.

Item 3

I draw a uniform B-Spline in degree 3 by using all control points. The knot vector that is used in this item is

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

A screenshot of the curve can be seen in Figure 11.3.3.

Item 4

I draw a nonuniform B-Spline in degree 3 by using all control points. The knot vector that is used in this item is

$\{0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4\}$

A screenshot of the curve can be seen in Figure 11.3.4.

Item 5

I setup a NURBS curve with weights :

$\{4, 3, 2, 1, 2, 3, 4\}$

The knot vector that is used in this item is

$\{0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2\}$

A screenshot of the curve can be seen in Figure 11.3.5.

11.4 Part 3

I construct a set of control points. The set consists of 7x7, namely 49 control points. The edge control points with 7 units are non-linear. The edges has slight curvature. By setting 8 inner control points manually I create two hills. The resulting curve can be seen in Figure 11.4.1

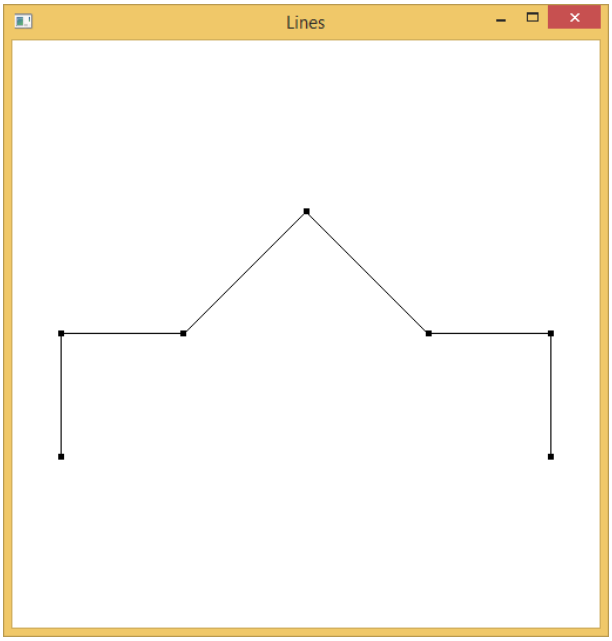


Figure 11.3.1: Linear

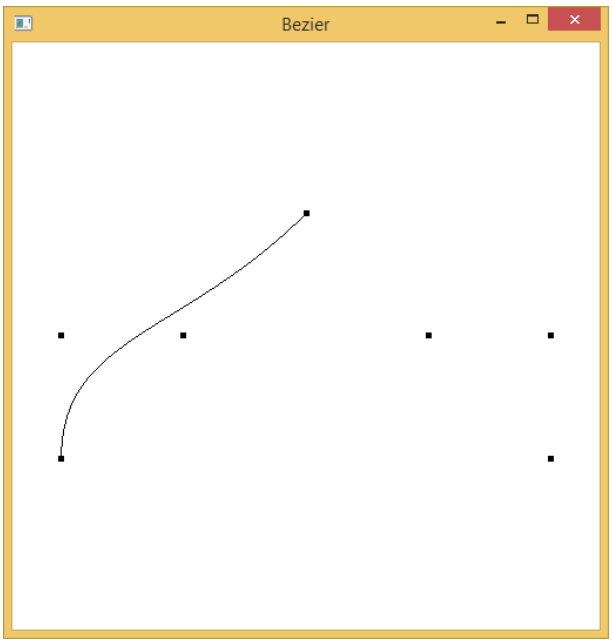


Figure 11.3.2: Bezier

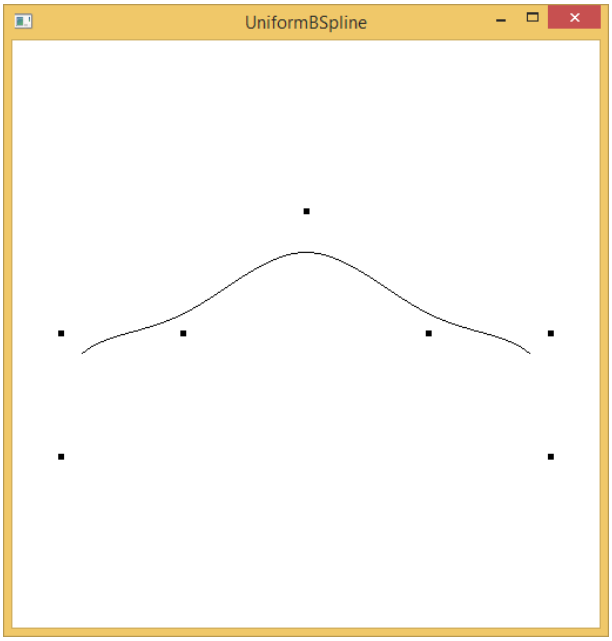


Figure 11.3.3: Uniform B-Spline

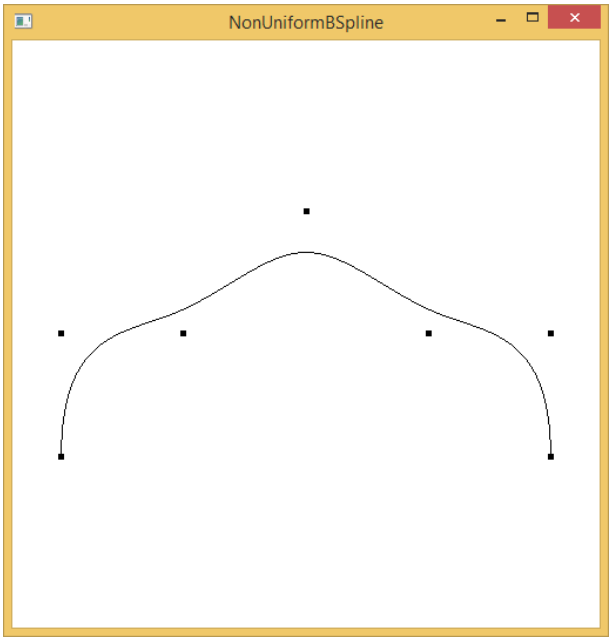


Figure 11.3.4: Non-uniform B-Spline

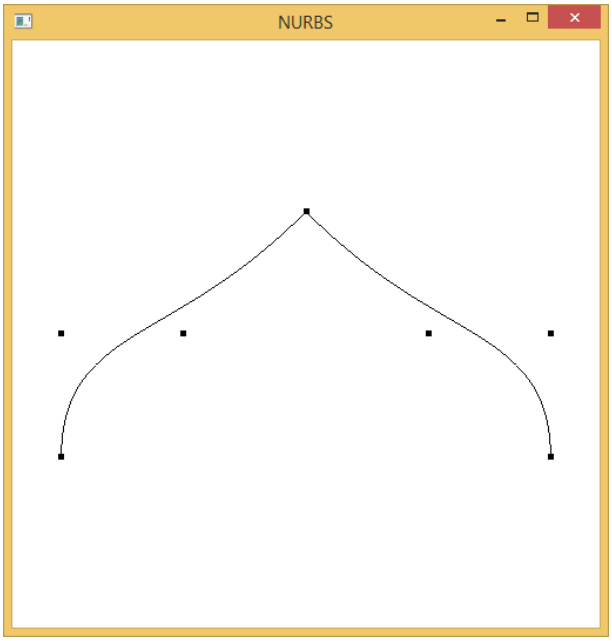


Figure 11.3.5: NURBS (Non-Uniform Rational B-Spline)

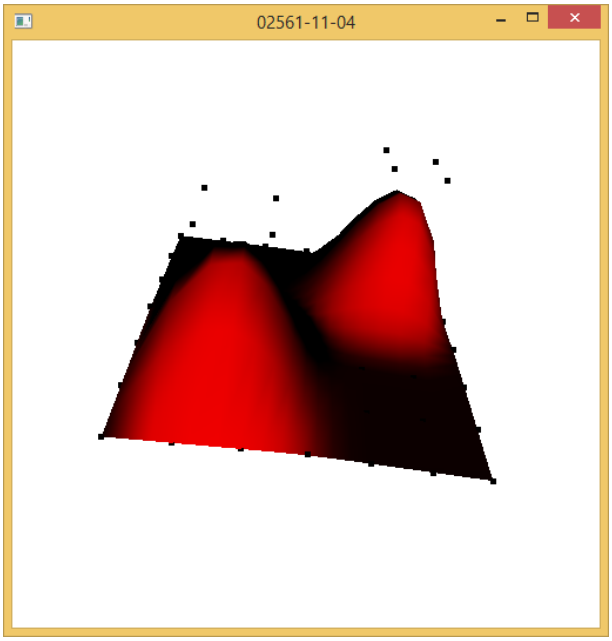


Figure 11.4.1: NURBS surface

Project: Soap Bubble Shader

12

12.1 Introduction

Development steps of soap bubble shader and visualisations of the results are presented in this report. Decision, design, implementation phases of the project are described.

For this project Andrew Glassner's Notebook [1] is the most dominant source of influence.

12.2 Motivation & Process

Among the exercises, exercise 10 was the one that makes me entertained, encouraged and inspired. I decided to extend the exercise by implementing a new, unusual shader and a chromatic glass shader as I was instructed. After I completed the chromatic glass shader, I made a research on internet to figure out which shader I should do. I came up with idea of introducing a bubble shader after a while.
I tried to understand the nature of a soap bubble, the physics behind it and I analyzed the implementation ideas, previous work that was done by others. Once I felt I gathered the background information needed for the project, I moved on implementation phase of bubble shader.

12.3 The Basis for Application Program

I copied entire project of exercise 10 and removed the shaders and the functionalities which is not needed for this task except one: glass shader.
Bubble is a just a surface, very thin surface in the shape of sphere, encapsulating some air. Since the width of this shell is not more than few micrometers,

the refraction of a soap bubble is zero. Therefore a soap bubble can be considered as a glass globe made by a special type of glass refraction index of which is equal to the one of air. I have chosen glass shader as a basis for my project.

12.4 The Bubble Shader

As stated in Section 12.3, there is no refraction in bubble. So I skip refraction lookup procedure. Instead, I simply assign incident vector to resulting refraction vector. The resulting scene can be seen in Figure 12.4.1.

Figure 12.4.1: Transparent sphere without refraction phenomena

The name of phenomena that gives a colorful appearance to bubble is called *iridescence*. Since a bubble is actually a thin film of soap-water mixture, the rays coming to the bubble reflect two times. The light rays reflects once from outer surface of the film and inner surface of the film. The width of the bubble shell is so small that these two reflections of photons interferes with each other, resulting *iridescence*.
Simulation of this natural phenomena in computer is possible by using the formula of reflection in bubble surface [1, p. 107]:

$$I_r = 4 I_i R_f \sin^2\left(\frac{2\pi}{\lambda} w\eta \cos\theta_t\right) \tag{12.1}$$

where I_r is the intensity of reflecting light, R_f is fresnel term that we are familiar with in glass shader, I_i is the intensity of incoming light, λ is wavelength

of incoming light ray in nanometers, w is width of soap bubble in nanometers, η is ratio between refraction index of air and that of bubble shell (refraction index of bubble shell is assumed 1.33, the same with that of water), θ_t is the angle of transmission in bubble shell.

I use Schlick's Approximation method [2] to calculate fresnel term.

After some experiments in application to determine ideal bubble width, I notice that a bubble with 2400 nm width appears realistic to human. Reader should consider that previous comments are quite subjective.

I make calculation of I_r for three main color; red, green and blue. I fuse single calculations for these three colors which resulted in the final term for iridescent reflection.

The resulting screenshot of scene can be seen in Figure 12.4.2.

Figure 12.4.2: Bubble shader. Per-fragment shading.

As a functionality, I add per-vertex and per-fragment bubble shader to the project. You can toggle between shaders by pressing s key. Per-vertex bubble shader can be seen in Figure 12.4.3. The difference between two shading strategies are noticeable.

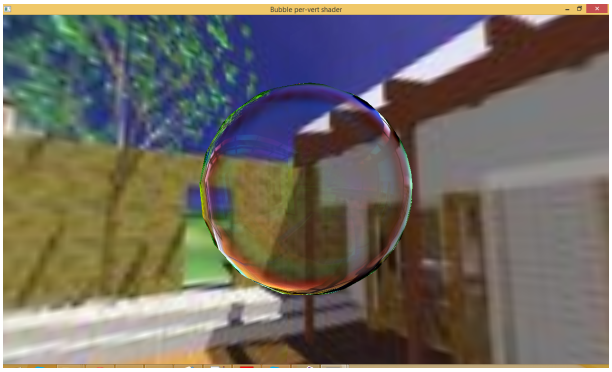


Figure 12.4.3: Bubble shader. Per-vertex shading.

Bibliography

[1] Andrew Glassner. *Andrew Glassner's Notebook: Recreational Computer Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[2] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13:233–246, 1994.