Computer Graphics Lab Report

Qingyang Liu

January 7, 2021

Contents

Part I

Lab 0 Basics

Chapter 1

Requirement

1.1 Basic Requirement

The basic requirement for the experiment is shown below:

1. Read a triangle mesh file.
2. Draw the model with OpenGL, with the funciton to rotate and translate it through user interface.
3. User can set the object starting& ending position& state, then plays an animation between them with quat to achieve smooth rotating.

1.2 Extra

There are also some extra functions the code implements:

1. Read multiple objects from a file, they can be agregated together to be rendered as one or rendered one by one.
2. Read the texture file in DDS, whether to use it depending on setting in the begining of code.
3. Camera postion and direction can be set with user interaction.
4. Implemention of *Phong reflection model* in *OenGL4* *Shader*.

Chapter 2

Theory and Application

2.1 Matrix

The final position of a point of object is calculated by transformations by model matrix, camera matrix and projection matrix sequently. In OpenGL programing, this part is done with *Vertex Shader*, which will be explained in 2.4.

2.2 Viewing

2.2.1 Perspective Projection

In this program, it only implements perspective projection.

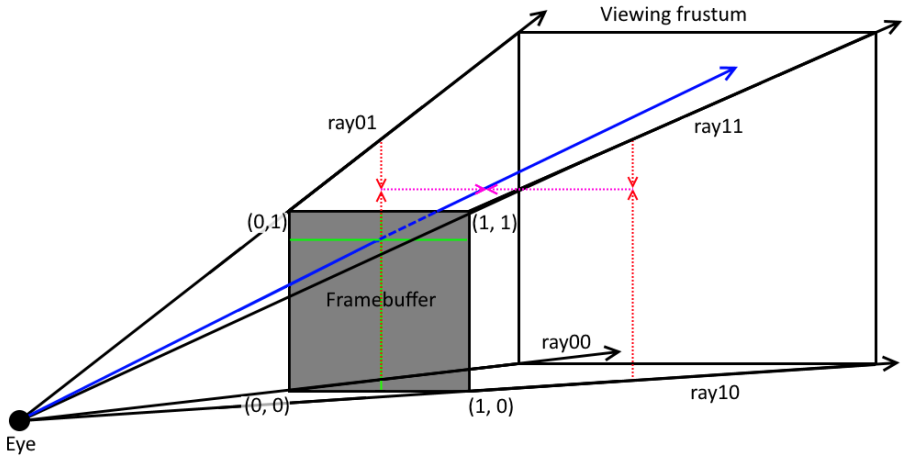

Figure 2.1: Perspective Projection, from [1]

As the matrix shows in [2], it projects the *z=-near* plane to *z=-1*, *z=-far* plane to *z=1*, *(xmax,ymax,-near)* plane to *(1,1,-1))*. With *near, far, fov*, the perspective projection can be calculated as:

$$\begin{Bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{Bmatrix}$$

2.2.2 Camera

In OpenGL programing, camera is spcified with *lookat* function. The program allows user to change vertical and horizonal angle to calculate the *direction* parameter and x,y,z position to calculate the *eye* parameter.

On user keyboard input, callback function is used to set variables.

2.3 Quaternion Rotation

In OpenGL programing, with user keyboard input, the start rotation angles along the camera direction, camera normal, camera right are known. Then start roatation euler angle can be calculated by adding angles in x,y,z direction. The ending euler angle is calculated similarly. However, the interpolation process cannot be calculated with euler angle because of *gimbal lock*. So they should be converted to quaternions and apply other algorithm for quaternions.

With *slerp* algorithm for spherical linear interpolation in the context of quaternion interpolation, [3], the interporating quaternion representing the rotation status can be calculated.

Since the rotation is centered at the origin, the position transformation (generated simply with 1-d linear interporation) should be applied to the model after it.

2.4 Shader

This program uses *OenGL4* *Shader* to render and specify vertex and lighting:

A *Shader* is a user-defined program designed to run on some stage of a graphics processor, [4]. *Shader* works as certain stages of the rendering pipeline, [5]. In *Vertex Shader*, it computes the rendered position of each point on object. In *Fragment Shader*, it compute the color of each point. The code describing the rendering methods are write in *GLSL(OpenGL Shading Language)* and compiled on execution.

In the end, the compiled code will run on graphics card to achieve better result, for example, the experiemnt below implementing ray tracing with user

interface for specifying camera status achieves a rather low delay instead simply outputs a static image for a long time.

In this experiment, only *Fragment Shader* and *Vertex Shader* are used for vectex position and implementing *Phong Shading*. Corresponding files are stored separated in *.fs* and *.vs* file extension.

2.5 Phong Reflection Model

For each surface point, the illumination equation is:[6]

$$I_{\mathrm{p}}=k_{\mathrm{a}} i_{\mathrm{a}}+\sum_{m \in \text{ lights}}\left(k_{\mathrm{d}}\left(\hat{L}_m \cdot \hat{N}\right) i_{m,\ \mathrm{d}}+k_{\mathrm{s}}\left(\hat{R}_m \cdot \hat{V}\right)^{\alpha} i_{m,\ \mathrm{s}}\right)$$

In short, the ambient reflection, specular reflection and diffuse reflection are calculated for each point. For simplcity, there is only one light source in the program. Also, shading is not implemented.

Chapter 3

Program

3.1 Compiling Guide

First restore packets with *NuGet*, then compile the project with Visual Studio 2019.

3.1.1 Environment

OpenGL4, Windows 10, Visual Studio 2019 *(v142)*.

3.1.2 Packages Used

1. *assimp*
2. *glfw*
3. *glm*
4. *glew*

3.2 User Guide

The program can only be interacted with keyboard.

1. Input filename(without *.obj*) and whether to load texture. Empty input will lead to the default model.
2. Press *1* to toggle edit mode. (In edit mode, model can be moved and rotated. In view mode, camera postion and direction can be changed.)
3. Long press *asdwqe* for changing position.

4. Long press *fghrty* for changing direction.
5. Press *esc* to exit.

3.3 Results

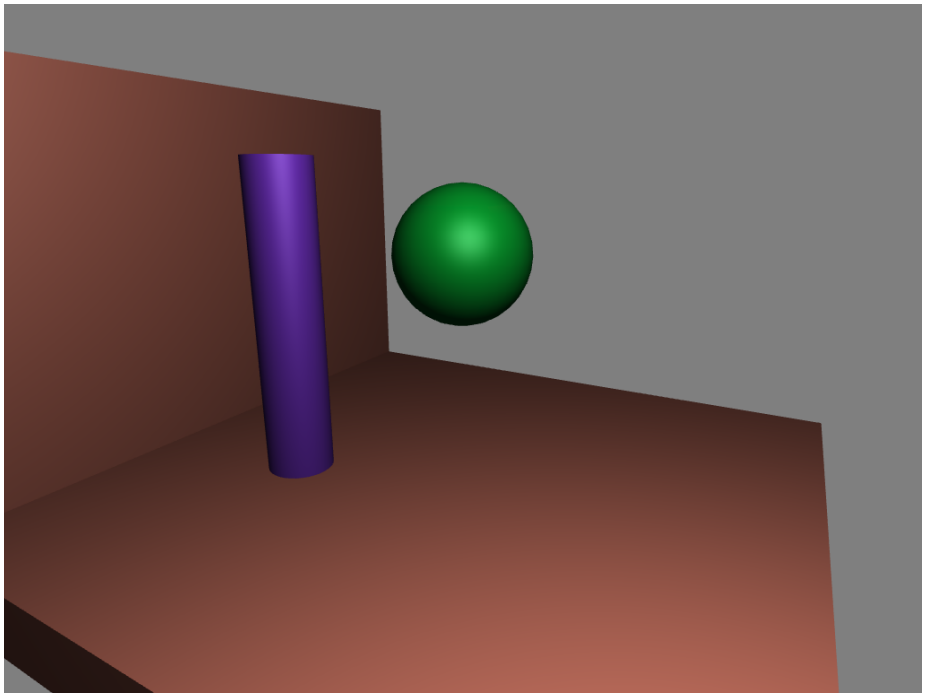

Figure 3.1: Loading the same model and texture as ray tracing program.

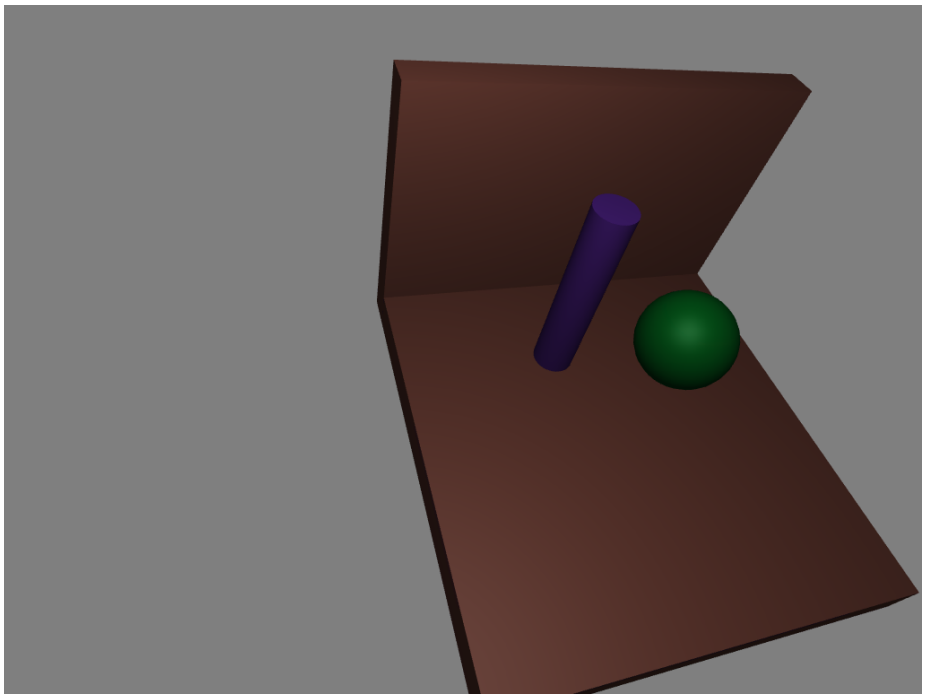

Figure 3.2: After translation and rotation.

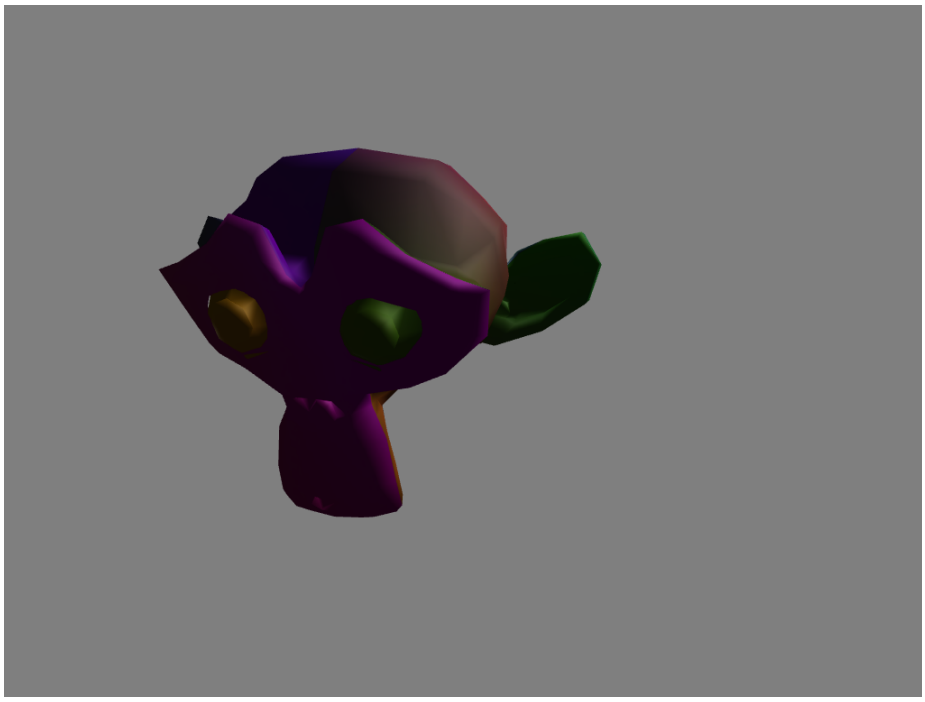

Figure 3.3: *suzanne.obj* , backlighting

Chapter 4

Source Code

4.1 Structure

1. *main.cpp* : main code
2. *object.fs.glsl* : vertex shader file
3. *object.vs.glsl* : fragment shader file
4. *utils*
   (a) *loadTexture.h* : functions for loading *.dds* file
   (b) *meshio.h* : functions for loading *.obj* file and shader file
5. *res/* :store model files

4.2 Code files

Code 4.1: main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>

#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
```

```
#include <glm/gtx/quaternion.hpp>

#include <GLFW/glfw3.h>

#include "utils/loadTexture.h"
#include "utils/meshio.h"

#pragma comment( lib, "OpenGL32.lib" )
using std::string;
using glm::quat;
using std::cout;
using std::cin;
using std::endl;

int wwidth = 1024, wheight = 768;

const int animateTime = 70;
const float viewMoveSpeed = 0.04f;
const float editMoveSpeed = 0.04f;
const float viewRotateSpeed = 0.01f;
const float editRotateSpeed = 0.025f;

glm::vec3 mTranslate = glm::vec3(0, 0, 0);
float vHorizontalAngle = -0.73;
float vVerticalAngle = -0.4;
glm::vec3 vPosition = glm::vec3(8.5, 5, -6);
glm::vec3 vDirection;
glm::vec3 vRight;
glm::vec3 vUp;

// loading obj
bool isloadDDS = true;
string filename = "room_thickwalls";
Mesh mesh;
GLuint Texture;
GLuint vertexbuffer;
GLuint texturebuffer;
GLuint normalbuffer;
GLuint elementbuffer;

// view matrix
glm::mat4 MVP;
glm::mat4 ViewMatrix;
glm::mat4 ModelMatrix = glm::mat4(1.0);
glm::mat4 ModelTransMatrix = glm::mat4(1.0);
quat ModelRoQuat = quat(1, 0, 0, 0);
```

```
// model rotation and moving
glm::mat4 oldModelTransMatrix = glm::mat4(1.0);
quat oldModelRotateQuat = quat(1, 0, 0, 0);
glm::mat4 atransMatrix;
quat interporateModelRotateQuat = quat(1, 0, 0, 0);


GLFWwindow* window;
// view
GLuint program0;
GLuint MatrixID;
GLuint ViewMatrixID;
GLuint ModelMatrixID;

//animation
int animateCounter = 0;
// user control temp var
bool modified = false;

bool escPress = false;
bool viewMode = true;
bool animateMode = false;

bool wPress = false;
bool aPress = false;
bool sPress = false;
bool dPress = false;
bool qPress = false;
bool ePress = false;
bool tPress = false;
bool fPress = false;
bool gPress = false;
bool hPress = false;
bool rPress = false;
bool yPress = false;

void initShaders()
{
    MatrixID = glGetUniformLocation(program0, "MVP");
    ViewMatrixID = glGetUniformLocation(program0, "V");
    ModelMatrixID = glGetUniformLocation(program0, "M");

    glUseProgram(program0);
    GLuint loadtexture = glGetUniformLocation(program0, "↩
    useTexture");
```

```
    glUniform1i(loadtexture, isloadDDS);
    glUseProgram(0);

}

void loadOBJtoPrg()
{
    string file = "res/" + filename + ".DDS";
    if (isloadDDS)
    {
        Texture = loadDDS(file.c_str());
        GLuint TextureID = glGetUniformLocation(program0, "↩
    textureSampler");
        // Bind our texture in Texture Unit 0
        glActiveTexture(GL_TEXTURE0 + 1);
        glBindTexture(GL_TEXTURE_2D, Texture);
        glActiveTexture(GL_TEXTURE0);
        glUseProgram(program0);
        glUniform1i(TextureID, 1);
        glUseProgram(0);
    }

    file = "res/" + filename + ".obj";
    if (!loadObj(isloadDDS, -1, -1, file.c_str(), mesh))
    {
        fprintf(stderr, "failed␣to␣load␣obj␣file");
        getchar();
        exit(-1);
    }

    // Load it into a VBO
    glGenBuffers(1, &vertexbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glBufferData(GL_ARRAY_BUFFER,
        mesh.vs.size() * sizeof(glm::vec3), &mesh.vs[0], ↩
    GL_STATIC_DRAW);

    glGenBuffers(1, &texturebuffer);
    glBindBuffer(GL_ARRAY_BUFFER, texturebuffer);
    glBufferData(GL_ARRAY_BUFFER,
        mesh.vts.size() * sizeof(glm::vec2), &mesh.vts[0], ↩
    GL_STATIC_DRAW);

    glGenBuffers(1, &normalbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
    glBufferData(GL_ARRAY_BUFFER,
```

```
    mesh.vns.size() * sizeof(glm::vec3), &mesh.vns[0], ↩
GL_STATIC_DRAW);

glGenBuffers(1, &elementbuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
    mesh.fs.size() * sizeof(unsigned short), &mesh.fs[0], ↩
GL_STATIC_DRAW);

// VAO
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glVertexAttribPointer(
    0,                  // attribute
    3,                  // size
    GL_FLOAT,           // type
    GL_FALSE,           // normalized?
    0,                  // stride
    (void*)0            // array buffer offset
);

glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, texturebuffer);
glVertexAttribPointer(
    1,                                  // attribute
    2,                                  // size
    GL_FLOAT,                           // type
    GL_FALSE,                           // normalized?
    0,                                  // stride
    (void*)0                            // array buffer ↩
offset
);
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glVertexAttribPointer(
    2,                                  // attribute
    3,                                  // size
    GL_FLOAT,                           // type
    GL_FALSE,                           // normalized?
    0,                                  // stride
    (void*)0                            // array buffer ↩
offset
```

```
    );
}

void init()
{
    if (!glfwInit())
    {
        fprintf(stderr, "Failed␣to␣initialize␣GLFW\n");
        getchar();
        exit(-1);
    }


    glfwDefaultWindowHints();
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, ↩
    GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, true);
    glfwWindowHint(GLFW_VISIBLE, false);
    glfwWindowHint(GLFW_RESIZABLE, false);

    window = glfwCreateWindow(wwidth, wheight, "exp00", NULL, ↩
    NULL);
    if (window == NULL) {
        fprintf(stderr, "Failed␣to␣open␣GLFW␣window.\n");
        getchar();
        glfwTerminate();
        exit(-1);
    }

    printf("press␣'1'␣to␣toggle␣edit␣mode\n"
        "press␣again␣to␣finish␣editing␣and␣the␣play␣the␣↩
    animation\n"
        "press␣'esc'␣to␣ternimate␣program\n"
        "press␣'wasdqe'␣to␣move␣object␣or␣camera␣depending␣on␣↩
    corrent␣mode\n"
        "press␣'fghtry'␣to␣rotate␣object␣or␣camera␣depending␣on↩
    ␣corrent␣mode.\n");
    glfwSetKeyCallback(window, [](GLFWwindow* window, int key, ↩
    int scancode, int action, int mode) {
        if (action == GLFW_PRESS)
        {
            if (key == GLFW_KEY_ESCAPE)
            {
```

```
            escPress = true;
        }
        else if (key == GLFW_KEY_1)
        {
            if (viewMode)
            {
                interporateModelRotateQuat = ModelRoQuat;
                oldModelTransMatrix = ModelTransMatrix;
                oldModelRotateQuat = ModelRoQuat;
                printf("entering␣edit␣mode\n");
            }
            else if (modified)
            {
                printf("exiting␣edit␣mode\n");
                printf("starting␣animation\n");
                atransMatrix = glm::translate(
                    glm::mat4(1.0f),
                    mTranslate * (float)(1 / (float)↩
animateTime));

                ModelTransMatrix = oldModelTransMatrix;
                ModelRoQuat = oldModelRotateQuat;
                animateMode = true;

                mTranslate = glm::vec3(0, 0, 0);
            }
            else
            {
                printf("exiting␣edit␣mode\n");
            }
            viewMode = !viewMode;
        }
        else if (!animateMode || animateMode && viewMode)
        {
            if (!viewMode)
            {
                modified = true;
            }

            if (key == GLFW_KEY_A)
            {
                aPress = true;
            }
            else if (key == GLFW_KEY_S)
            {
                sPress = true;
```

```
            }
            else if (key == GLFW_KEY_D)
            {
                dPress = true;
            }
            else if (key == GLFW_KEY_W)
            {
                wPress = true;
            }
            else if (key == GLFW_KEY_Q)
            {
                qPress = true;
            }
            else if (key == GLFW_KEY_E)
            {
                ePress = true;
            }
            else if (key == GLFW_KEY_F)
            {
                fPress = true;
            }
            else if (key == GLFW_KEY_G)
            {
                gPress = true;
            }
            else if (key == GLFW_KEY_H)
            {
                hPress = true;
            }
            else if (key == GLFW_KEY_T)
            {
                tPress = true;
            }
            else if (key == GLFW_KEY_R)
            {
                rPress = true;
            }
            else if (key == GLFW_KEY_Y)
            {
                yPress = true;
            }
        }
    }
    else if (action == GLFW_RELEASE)
    {
        if (key == GLFW_KEY_A)
```

```
            {
                aPress = false;
            }
            else if (key == GLFW_KEY_S)
            {
                sPress = false;
            }
            else if (key == GLFW_KEY_D)
            {
                dPress = false;
            }
            else if (key == GLFW_KEY_W)
            {
                wPress = false;
            }
            else if (key == GLFW_KEY_Q)
            {
                qPress = false;
            }
            else if (key == GLFW_KEY_E)
            {
                ePress = false;
            }
            else if (key == GLFW_KEY_F)
            {
                fPress = false;
            }
            else if (key == GLFW_KEY_G)
            {
                gPress = false;
            }
            else if (key == GLFW_KEY_H)
            {
                hPress = false;
            }
            else if (key == GLFW_KEY_T)
            {
                tPress = false;
            }
            else if (key == GLFW_KEY_R)
            {
                rPress = false;
            }
            else if (key == GLFW_KEY_Y)
            {
                yPress = false;
```

```
            }
        }
        });

    const GLFWvidmode* vidmode = glfwGetVideoMode(↩
    glfwGetPrimaryMonitor());
    glfwSetWindowPos(
        window,
        (vidmode->width - wwidth) / 2,
        (vidmode->height - wheight) / 2
    );
    glfwMakeContextCurrent(window);
    glfwSwapInterval(1);

    // Initialize GLEW
    glewExperimental = true; // Needed for core profile
    if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Failed to initialize GLEW\n");
        getchar();
        glfwTerminate();
        exit(-1);
    };

    program0 = LoadShaders("object.vs.glsl", "object.fs.glsl");
    initShaders();
    loadOBJtoPrg();

    glfwShowWindow(window);

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glEnable(GL_CULL_FACE);
}


quat slerp(quat v0, quat v1, float t) {
    v0 = glm::normalize(v0);
    v1 = glm::normalize(v1);


    float dot = glm::dot(v0, v1);

    if (dot < 0.0f) {
```

```
        v1 = -v1;
        dot = -dot;
    }

    const float DOT_THRESHOLD = 0.9995;
    if (dot > DOT_THRESHOLD) {
        quat result = glm::mix(v0, v1, t);
        result = glm::normalize(result);
        return result;
    }

    float theta_0 = acos(dot);
    float theta = theta_0 * t;
    float sin_theta = sin(theta);
    float sin_theta_0 = sin(theta_0);

    float s0 = cos(theta) - dot * sin_theta / sin_theta_0;
    float s1 = sin_theta / sin_theta_0;

    quat tmp = glm::normalize(glm::mix(v0, v1, s1 / (s0 + s1)))↩
    ;

    return tmp;
}


void viewModel()
{
    glm::mat4 ProjectionMatrix = glm::perspective(
        glm::radians(60.0f), (float)wwidth / wheight, 1.0f, ↩
    100.0f);


    if (animateMode)
    {
        if (animateCounter < animateTime)
        {
            ModelTransMatrix = ModelTransMatrix * atransMatrix;
            ModelMatrix = ModelTransMatrix * glm::toMat4(slerp(↩
    ModelRoQuat, interporateModelRotateQuat, (float)↩
    animateCounter / animateTime));
            animateCounter++;
        }
        else
        {
            printf("finishing␣animation\n");
```

```
            animateMode = false;
            animateCounter = 0;
            ModelRoQuat = interporateModelRotateQuat;
        }
    }

    vDirection = glm::vec3(
        cos(vVerticalAngle) * sin(vHorizontalAngle),
        sin(vVerticalAngle),
        cos(vVerticalAngle) * cos(vHorizontalAngle)
    );

    vRight = glm::vec3(
        sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
        0,
        cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
    );

    vUp = glm::cross(vRight, vDirection);

    if (!animateMode && !viewMode)
    {
        float modelDirectionAngle = 0.0f;
        float modelRightAngle = 0.0f;
        float modelUpAngle = 0.0f;

        if (aPress == true)
        {
            mTranslate -= vRight * editMoveSpeed;
        }
        else if (sPress == true)
        {
            mTranslate -= vDirection * editMoveSpeed;
        }
        else if (dPress == true)
        {
            mTranslate += vRight * editMoveSpeed;
        }
        else if (wPress == true)
        {
            mTranslate += vDirection * editMoveSpeed;
        }
        else if (qPress == true)
        {
            mTranslate += vUp * editMoveSpeed;
        }
```

```
        else if (ePress == true)
        {
            mTranslate -= vUp * editMoveSpeed;
        }
        else if (fPress == true)
        {
            modelUpAngle -= viewRotateSpeed;
        }
        else if (gPress == true)
        {
            modelRightAngle += viewRotateSpeed;
        }
        else if (hPress == true)
        {
            modelUpAngle += viewRotateSpeed;
        }
        else if (tPress == true)
        {
            modelRightAngle -= viewRotateSpeed;
        }
        else if (rPress == true)
        {
            modelDirectionAngle -= viewRotateSpeed;

        }
        else if (yPress == true)
        {
            modelDirectionAngle += viewRotateSpeed;

        }

        ModelTransMatrix = glm::translate(glm::mat4(1.0f), ↩
    mTranslate);

        glm::quat ModelDirectionQuaternion = glm::quat(
            cos(modelDirectionAngle / 2),
            vDirection.x * sin(modelDirectionAngle / 2),
            vDirection.y * sin(modelDirectionAngle / 2),
            vDirection.z * sin(modelDirectionAngle / 2)
        );

        glm::quat ModelUpQuaternion = glm::quat(
            cos(modelUpAngle / 2),
            vUp.x * sin(modelUpAngle / 2),
            vUp.y * sin(modelUpAngle / 2),
            vUp.z * sin(modelUpAngle / 2)
```

```
        );

        glm::quat ModelRightQuaternion = glm::quat(
            cos(modelRightAngle / 2),
            vRight.x * sin(modelRightAngle / 2),
            vRight.y * sin(modelRightAngle / 2),
            vRight.z * sin(modelRightAngle / 2)
        );

        interporateModelRotateQuat =
            ModelDirectionQuaternion *
            ModelUpQuaternion *
            ModelRightQuaternion *
            interporateModelRotateQuat;
        modelRightAngle = modelUpAngle = modelDirectionAngle = ↩
    0;
        ModelMatrix = oldModelTransMatrix * ModelTransMatrix * ↩
    glm::toMat4(interporateModelRotateQuat);
    }
    else if (viewMode)
    {
        if (aPress == true)
        {
            vPosition -= vRight * viewMoveSpeed;
        }
        else if (sPress == true)
        {
            vPosition -= vDirection * viewMoveSpeed;
        }
        else if (dPress == true)
        {
            vPosition += vRight * viewMoveSpeed;
        }
        else if (wPress == true)
        {
            vPosition += vDirection * viewMoveSpeed;
        }
        else if (qPress == true)
        {
            vPosition += vUp * viewMoveSpeed;
        }
        else if (ePress == true)
        {
            vPosition -= vUp * viewMoveSpeed;
        }
        else if (fPress == true)
```

```
        {
            vHorizontalAngle += viewRotateSpeed;
        }
        else if (gPress == true)
        {
            vVerticalAngle -= viewRotateSpeed;
        }
        else if (hPress == true)
        {
            vHorizontalAngle -= viewRotateSpeed;
        }
        else if (tPress == true)
        {
            vVerticalAngle += viewRotateSpeed;
        }

        vDirection = glm::vec3(
            cos(vVerticalAngle) * sin(vHorizontalAngle),
            sin(vVerticalAngle),
            cos(vVerticalAngle) * cos(vHorizontalAngle)
        );

        vRight = glm::vec3(
            sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
            0,
            cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
        );

        vUp = glm::cross(vRight, vDirection);
    }


    ViewMatrix = glm::lookAt(
        vPosition,
        vPosition + vDirection,
        vUp
    );

    MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;
}

void loop()
{

    while (escPress != true && glfwWindowShouldClose(window) ==↩
     0)
```

```
    {
        glfwPollEvents();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // start edit objects in program
        glUseProgram(program0);

        viewModel();

        // Send our transformation to the currently bound ↩
    shader,
        // in the "MVP" uniform
        glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
        glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &↩
    ModelMatrix[0][0]);
        glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &↩
    ViewMatrix[0][0]);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);

        glDrawElements(
            GL_TRIANGLES,      // mode
            mesh.fs.size(),    // count
            GL_UNSIGNED_SHORT,   // type
            (void*)0           // element array buffer offset
        );

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
        glfwSwapBuffers(window);
        glUseProgram(0);
    }

    glfwTerminate();
}

int main(int argc, char* argv[]) {
    string yn, filen;
    cout << "please␣input␣filename:" << endl;
    getline(std::cin, filen);
    if (!filen.empty())
    {
        filename = filen;
        cout << "Does␣it␣has␣texture?␣please␣input␣y/n" << endl↩
    ;
        getline(std::cin, yn);
        if (!yn.empty())
        {
```

```
            if (yn.c_str()[0] == 'y')
                isloadDDS = true;
            else if (yn.c_str()[0] == 'n')
                isloadDDS = false;
            else
            {
                cout << "please␣input␣y/n" << endl;
                getchar();
                exit(-1);
            }
        }
    }
    cout << "filename:" <<filename<<"␣isloadtexture:"<< ↩
    isloadDDS << endl;
    init();
    loop();
}
```

Code 4.2: object.fs.glsl

```
#version 430 core

in vec2 UV;
in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;

uniform bool useTexture;
uniform sampler2D textureSampler;

out vec4 color;

void main(){

    // Light emission properties
    vec3 LightColor = vec3(1,1,1);
    float LightPower = 50.0f;

    // Material properties
    vec3 MaterialDiffuseColor = vec3(0.5,0.5,1.0);
    if(useTexture)
    {
        MaterialDiffuseColor=texture( textureSampler, UV ).rgb;
    };
```

```
    vec3 MaterialAmbientColor = vec3(0.1,0.1,0.1) * ↩
    MaterialDiffuseColor;
    vec3 MaterialSpecularColor = vec3(0.3,0.3,0.3);

    // Distance to the light
    float distance = length( vec3(6,6,-6) - Position_worldspace↩
     );

    vec3 n = normalize( Normal_cameraspace );
    vec3 l = normalize( LightDirection_cameraspace );
    float cosTheta = clamp( dot( n,l ), 0,1 );

    vec3 E = normalize(EyeDirection_cameraspace);
    vec3 R = reflect(-l,n);
    float cosAlpha = clamp( dot( E,R ), 0,1 );

    color.rgb=// Ambient : simulates indirect lighting
        MaterialAmbientColor +
        // Diffuse
        MaterialDiffuseColor * LightColor * LightPower * ↩
    cosTheta / (distance*distance) +
        // Specular
        MaterialSpecularColor * LightColor * LightPower * pow(↩
    cosAlpha,5) / (distance*distance);
    //color.rgb=MaterialDiffuseColor;
    color.a=1;
}
```

Code 4.3: object.vs.glsl

```
#version 430 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

out vec2 UV;
out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;

uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;
```

```
void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace, 1);

    Position_worldspace = (M * vec4(vertexPosition_modelspace↩
    ,1)).xyz;

    vec3 vertexPosition_cameraspace = ( V * M * vec4(↩
    vertexPosition_modelspace,1)).xyz;
    EyeDirection_cameraspace = vec3(0,0,0) - ↩
    vertexPosition_cameraspace;

    vec3 LightPosition_cameraspace = ( V * vec4(6,6,-6,1)).xyz;
    LightDirection_cameraspace = LightPosition_cameraspace + ↩
    EyeDirection_cameraspace;

    Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace↩
    ,0)).xyz;
    UV = vertexUV;
}
```

Code 4.4: utils/loadTexture.h

```
    #pragma once
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    #include <GL/glew.h>

    #include <GLFW/glfw3.h>

    GLuint loadBMP_custom(const char* imagepath);

    GLuint loadDDS(const char* imagepath);
```

Code 4.5: utils/loadTexture.cpp

```
#include "loadTexture.h"

/*
* modified from: ogl-master
* common/texture.cpp
```

```
* */
#define FOURCC_DXT1 0x31545844 // Equivalent to "DXT1" in ASCII
#define FOURCC_DXT3 0x33545844 // Equivalent to "DXT3" in ASCII
#define FOURCC_DXT5 0x35545844 // Equivalent to "DXT5" in ASCII

GLuint loadDDS(const char* imagepath) {

    unsigned char header[124];

    FILE* fp;

    /* try to open the file */
    fp = fopen(imagepath, "rb");
    if (fp == NULL) {
        printf("%s could not be opened.\n", imagepath); getchar↩
    ();
        return 0;
    }

    /* verify the type of file */
    char filecode[4];
    fread(filecode, 1, 4, fp);
    if (strncmp(filecode, "DDS ", 4) != 0) {
        fclose(fp);
        return 0;
    }

    /* get the surface desc */
    fread(&header, 124, 1, fp);

    unsigned int height = *(unsigned int*)&(header[8]);
    unsigned int width = *(unsigned int*)&(header[12]);
    unsigned int linearSize = *(unsigned int*)&(header[16]);
    unsigned int mipMapCount = *(unsigned int*)&(header[24]);
    unsigned int fourCC = *(unsigned int*)&(header[80]);


    unsigned char* buffer;
    unsigned int bufsize;
    /* how big is it going to be including all mipmaps? */
    bufsize = mipMapCount > 1 ? linearSize * 2 : linearSize;
    buffer = (unsigned char*)malloc(bufsize * sizeof(unsigned ↩
    char));
    fread(buffer, 1, bufsize, fp);
    /* close the file pointer */
    fclose(fp);
```

```
unsigned int components = (fourCC == FOURCC_DXT1) ? 3 : 4;
unsigned int format;
switch (fourCC)
{
case FOURCC_DXT1:
    format = GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
    break;
case FOURCC_DXT3:
    format = GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
    break;
case FOURCC_DXT5:
    format = GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
    break;
default:
    free(buffer);
    return 0;
}

// Create one OpenGL texture
GLuint textureID;
glGenTextures(1, &textureID);

// "Bind" the newly created texture : all future texture ↩
functions will modify this texture
glBindTexture(GL_TEXTURE_2D, textureID);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

unsigned int blockSize = (format == ↩
GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) ? 8 : 16;
unsigned int offset = 0;

/* load the mipmaps */
for (unsigned int level = 0; level < mipMapCount && (width ↩
|| height); ++level)
{
    unsigned int size = ((width + 3) / 4) * ((height + 3) /↩
 4) * blockSize;
    glCompressedTexImage2D(GL_TEXTURE_2D, level, format, ↩
width, height,
        0, size, buffer + offset);

    offset += size;
    width /= 2;
    height /= 2;
```

```
        // Deal with Non-Power-Of-Two textures. This code is ↩
    not included in the webpage to reduce clutter.
        if (width < 1) width = 1;
        if (height < 1) height = 1;

    }

    free(buffer);

    return textureID;


}
```

Code 4.6: utils/meshio.h

```
#pragma once
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <sstream>
#include <cstring>
#include <stdlib.h>

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <GLFW/glfw3.h>

// Include AssImp
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // Output data structure
#include <assimp/postprocess.h>     // Post processing flags
using std::vector;
using glm::vec2;
using glm::vec3;
using glm::vec4;

struct Mesh
{
    int vn;
    int fn;
    vector<vec3> vs;
```

```
    vector<vec2> vts;
    vector<vec3> vns;
    vector<short> fs;
    vector<vec3> maxvs;
    vector<vec3>  minvs;
    vector<vec3>  ns;
};

bool loadObj(
    bool hasDDS,
    int start0,
    int end0,
    const char* path,
    Mesh& mesh
);

bool loadAssImp(
    const char* path,
    std::vector<unsigned short>& indices,
    std::vector<glm::vec3>& vertices,
    std::vector<glm::vec2>& uvs,
    std::vector<glm::vec3>& normals
);

void LoadShader(GLuint ShaderID, const char* file_path);

GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path);
```

Code 4.7: utils/meshio.cpp

```
#include "meshio.h"

// anyway, we only allow one mesh and one texture here,
// and other input may cause rendering error as a result
bool loadObj(
    bool hasDDS,
    int start0,
    int end0,
    const char* path,
    Mesh& mesh
) {
    Assimp::Importer importer;

    const aiScene* scene = importer.ReadFile(
```

```
    path,
    0       /*aiProcess_JoinIdenticalVertices | ↩
aiProcess_SortByPType*/
);
if (!scene) {
    fprintf(stderr, importer.GetErrorString());
    getchar();
    return false;
}

int start = start0 == -1 ? 0 : start0;
int end = end0 == -1 ? scene->mNumMeshes : end0 + 1;
// for now the models donnot overlap
for (int j = start; j < end; j++)
{
    int vertexnum = mesh.vs.size();
    const aiMesh* inmesh = scene->mMeshes[j];

    mesh.vn += inmesh->mNumVertices;
    mesh.fn += inmesh->mNumFaces;

    // Fill vertices positions
    mesh.vs.reserve(vertexnum + inmesh->mNumVertices);
    for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
 {
        aiVector3D pos = inmesh->mVertices[i];
        mesh.vs.push_back(vec3(pos.x, pos.y, pos.z));
    }

    mesh.vts.reserve(vertexnum + inmesh->mNumVertices);
    // no texture
    if (hasDDS)
    {
        // Fill vertices texture coordinates
        for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
i++) {
            aiVector3D UVW = inmesh->mTextureCoords[j][i];
            // Assume only 1 set of UV coords; AssImp ↩
supports 8 UV sets.
            mesh.vts.push_back(vec2(UVW.x, -UVW.y));
        }
    }
    else
    {
        for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
i++) {
```

```
            mesh.vts.push_back(vec2(0, 0));
        }
    }

    // Fill vertices normals
    mesh.vns.reserve(vertexnum + inmesh->mNumVertices);
    for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
 {
        aiVector3D n = inmesh->mNormals[i];
        mesh.vns.push_back(glm::vec3(n.x, n.y, n.z));
    }


    // Fill face indices
    mesh.fs.reserve(mesh.fs.size() + 3 * inmesh->mNumFaces)↩
;
    mesh.maxvs.reserve(mesh.maxvs.size() + inmesh->↩
mNumFaces);
    mesh.minvs.reserve(mesh.minvs.size() + inmesh->↩
mNumFaces);
    mesh.ns.reserve(mesh.ns.size() + inmesh->mNumFaces);
    for (unsigned int i = 0; i < inmesh->mNumFaces; i++) {
        // Assume the model has only triangles.
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[0] + ↩
vertexnum);
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[1] + ↩
vertexnum);
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[2] + ↩
vertexnum);
        vec3 v1 = mesh.vs[inmesh->mFaces[i].mIndices[0] + ↩
vertexnum];
        vec3 v2 = mesh.vs[inmesh->mFaces[i].mIndices[1] + ↩
vertexnum];
        vec3 v3 = mesh.vs[inmesh->mFaces[i].mIndices[2] + ↩
vertexnum];
        mesh.maxvs.push_back(glm::max(glm::max(v1, v2), v3)↩
);
        mesh.minvs.push_back(glm::min(glm::min(v1, v2), v3)↩
);
        // no interpolation for edges
        vec3 n1 = mesh.vns[inmesh->mFaces[i].mIndices[0] + ↩
vertexnum];
        vec3 n2 = mesh.vns[inmesh->mFaces[i].mIndices[1] + ↩
vertexnum];
        vec3 n3 = mesh.vns[inmesh->mFaces[i].mIndices[2] + ↩
vertexnum];
```

```
            mesh.ns.push_back(glm::normalize(n1 + n2 + n3));
        }

    }

    return true;
}

/*
* modified from: ogl-master
* common/objloader.cpp
* author: opengl-tutorial.org
*/
bool loadAssImp(
    const char* path,
    std::vector<unsigned short>& indices,
    std::vector<glm::vec3>& vertices,
    std::vector<glm::vec2>& uvs,
    std::vector<glm::vec3>& normals
) {

    Assimp::Importer importer;

    const aiScene* scene = importer.ReadFile(
        path,
        0       /*aiProcess_JoinIdenticalVertices | ↩
    aiProcess_SortByPType*/
    );
    if (!scene) {
        fprintf(stderr, importer.GetErrorString());
        getchar();
        return false;
    }

    // for now the models donnot overlap
    for (int i = 0; i < scene->mNumMeshes; i++)
    {
        int vertexnum = vertices.size();
        const aiMesh* mesh = scene->mMeshes[i];

        // Fill vertices positions
        vertices.reserve(vertices.size() + mesh->mNumVertices);
        for (unsigned int i = 0; i < mesh->mNumVertices; i++) {
            aiVector3D pos = mesh->mVertices[i];
            vertices.push_back(glm::vec3(pos.x, pos.y, pos.z));
        }
```

```
        uvs.reserve(uvs.size() + mesh->mNumVertices);
        // no texture
        if (scene->mNumTextures > 0)
        {
            // Fill vertices texture coordinates
            for (unsigned int i = 0; i < mesh->mNumVertices; i↩
    ++) {
                aiVector3D UVW = mesh->mTextureCoords[0][i];
                // Assume only 1 set of UV coords; AssImp ↩
    supports 8 UV sets.
                uvs.push_back(glm::vec2(UVW.x, UVW.y));
            }
        }

        // Fill vertices normals
        normals.reserve(normals.size() + mesh->mNumVertices);
        for (unsigned int i = 0; i < mesh->mNumVertices; i++) {
            aiVector3D n = mesh->mNormals[i];
            normals.push_back(glm::vec3(n.x, n.y, n.z));
        }


        // Fill face indices
        indices.reserve(indices.size() + 3 * mesh->mNumFaces);
        for (unsigned int i = 0; i < mesh->mNumFaces; i++) {
            // Assume the model has only triangles.
            indices.push_back(mesh->mFaces[i].mIndices[0] + ↩
    vertexnum);
            indices.push_back(mesh->mFaces[i].mIndices[1] + ↩
    vertexnum);
            indices.push_back(mesh->mFaces[i].mIndices[2] + ↩
    vertexnum);
        }

    }

    return true;
}


/*
* modified from: ogl-master
* common/shader.cpp
* author: opengl-tutorial.org
*/
```

```
void LoadShader(GLuint ShaderID, const char* file_path)
{
    // Read the Vertex Shader code from the file
    std::string ShaderCode;
    std::ifstream ShaderStream(file_path, std::ios::in);
    if (ShaderStream.is_open()) {
        std::stringstream sstr;
        sstr << ShaderStream.rdbuf();
        ShaderCode = sstr.str();
        ShaderStream.close();
    }
    else {
        printf("Impossible␣to␣open␣%s.␣\n", file_path);
        getchar();
        exit(-1);
    }

    GLint Result = GL_FALSE;
    int InfoLogLength;

    printf("Compiling␣shader␣:␣%s\n", file_path);
    char const* xSourcePointer = ShaderCode.c_str();
    glShaderSource(ShaderID, 1, &xSourcePointer, NULL);
    glCompileShader(ShaderID);

    glGetShaderiv(ShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(ShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength)↩
    ;
    if (InfoLogLength > 0) {
        std::vector<char> ShaderErrorMessage(InfoLogLength + 1)↩
    ;
        glGetShaderInfoLog(ShaderID, InfoLogLength, NULL, &↩
    ShaderErrorMessage[0]);
        printf("%s\n", &ShaderErrorMessage[0]);
    }
}



GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path) {

    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER↩
    );
```

```
    LoadShader(VertexShaderID, vertex_file_path);
    LoadShader(FragmentShaderID, fragment_file_path);

    GLint Result = GL_FALSE;
    int InfoLogLength;

    // Link the program
    printf("Linking program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);

    // Check the program
    glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &↩
    InfoLogLength);
    if (InfoLogLength > 0) {
        std::vector<char> ProgramErrorMessage(InfoLogLength + ↩
    1);
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &↩
    ProgramErrorMessage[0]);
        printf("%s\n", &ProgramErrorMessage[0]);
    }


    glDetachShader(ProgramID, VertexShaderID);
    glDetachShader(ProgramID, FragmentShaderID);

    glDeleteShader(VertexShaderID);
    glDeleteShader(FragmentShaderID);

    return ProgramID;
}
```

Part II

Lab 1 Hierarchival Modeling

Chapter 5

Requirement

Animate character skeleton as a tree of transformations.

5.1 Basic Requirement

The basic requirement for the experiment is to animate character skeleton as a tree of transformations.

5.2 Extra

There are also some extra functions the program implements:

1. Read meshes from obj file. Flexibly render by each mesh.
2. Camera postion and direction can be set through user interaction.
3. Implemention of *Phong reflection model* in *OenGL4* *Shader*.

Chapter 6

Theory and Application

6.1 Hierarchical Modeling

In hierarchival modeling, the transformation of parent node must be applied to child nodes. As a result, stack structure in code implemention is often used. In *OenGL4* , there is no built in stack for models, so we implement in a similar way.

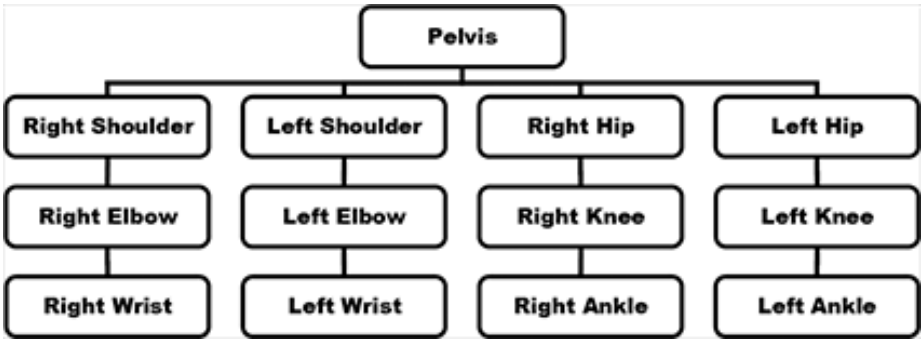

Figure 6.1: Hierarchical structure of the character skeleton.

6.2 Model Building

We use *Blender* to build the model and export *obj* file for the program to load from.

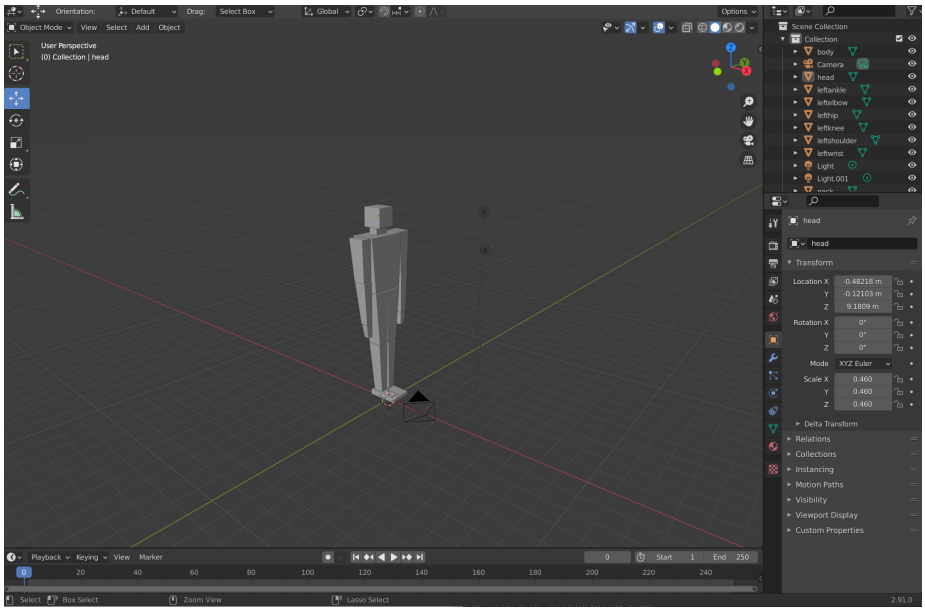

Figure 6.2: Editing model in *Blender*

The *.blend* source file is in */01/res/*.

6.3 Motion

We referred to [7] when buidling character skeleton's motions.

Chapter 7

Program

7.1 Compiling Guide

First restore packets with *NuGet*, then compile the project with Visual Studio 2019.

7.1.1 Environment

OpenGL4, Windows 10, Visual Studio 2019 *(v142)*.

7.1.2 Packages Used

1. *assimp*
2. *glfw*
3. *glm*
4. *glew*

7.2 User Guide

The program can only be interacted with keyboard.

1. Long press *asdwqe* for changing position.
2. Long press *fghrty* for changing direction.
3. Press *esc* to exit.

7.3 Results

Figure 7.1: Result 1

Figure 7.2: Result 2

Chapter 8

Source Code

8.1 Structure

1. *main.cpp* : main code
2. *object.fs.glsl* : vertex shader file
3. *object.vs.glsl* : fragment shader file
4. *utils*
   (a) *loadTexture.h* : functions for loading *.dds* file
   (b) *meshio.h* : functions for loading *.obj* file and shader file
5. *res/* :store model files

8.2 Code files

Code 8.1: main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>

#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
```

```
#include <glm/gtx/quaternion.hpp>

#include <GLFW/glfw3.h>

#include "utils/loadTexture.h"
#include "utils/meshio.h"

#pragma comment( lib, "OpenGL32.lib" )
using std::string;
using std::vector;
using glm::mat4;

int wwidth = 1024, wheight = 768;
GLFWwindow* window;


const int runTime = 70;
const float viewMoveSpeed = 0.04f;
const float viewRotateSpeed = 0.01f;

// running
int currentRun = 0;

bool escPress = false;
bool wPress = false;
bool aPress = false;
bool sPress = false;
bool dPress = false;
bool qPress = false;
bool ePress = false;
bool tPress = false;
bool fPress = false;
bool gPress = false;
bool hPress = false;
bool rPress = false;
bool yPress = false;


glm::mat4 MVP;
glm::mat4 ViewMatrix;
glm::mat4 ModelMatrix = glm::mat4(1.0);

float vHorizontalAngle = -glm::pi<float>();
float vVerticalAngle = 0;
glm::vec3 vPosition = glm::vec3(7, 5, 15);
glm::vec3 vDirection;
```

```
glm::vec3 vRight;
glm::vec3 vUp;

float modelDirectionAngle = 0.0f;
float modelRightAngle = 0.0f;
float modelUpAngle = 0.0f;

GLuint program0;
// MVP
GLuint MatrixID;
//V
GLuint ViewMatrixID;
//M
GLuint ModelMatrixID;

// loading obj
bool isloadDDS = false;
string filename = "01";
Mesh mesh;
vector<Mesh> meshes;
GLuint Texture;
GLuint vertexbuffer;
GLuint texturebuffer;
GLuint normalbuffer;
GLuint elementbuffer;
GLuint vao;

void initShaders()
{
    MatrixID = glGetUniformLocation(program0, "MVP");
    ViewMatrixID = glGetUniformLocation(program0, "V");
    ModelMatrixID = glGetUniformLocation(program0, "M");

    glUseProgram(program0);
    GLuint loadtexture = glGetUniformLocation(program0, "↩
    useTexture");
    glUniform1i(loadtexture, isloadDDS);
    glUseProgram(0);

}

void loadOBJtoPrg()
{
    string file = "res/" + filename + ".obj";
    if (!loadObj(isloadDDS, file.c_str(), meshes, mesh))
    {
```

```
        fprintf(stderr, "failed␣to␣load␣obj␣file");
        getchar();
        exit(-1);
    }

    // Load it into a VBO
    glGenBuffers(1, &vertexbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glBufferData(GL_ARRAY_BUFFER,
        mesh.vs.size() * sizeof(glm::vec3), &mesh.vs[0], ↩
    GL_DYNAMIC_DRAW);

    glGenBuffers(1, &texturebuffer);
    glBindBuffer(GL_ARRAY_BUFFER, texturebuffer);
    glBufferData(GL_ARRAY_BUFFER,
        mesh.vts.size() * sizeof(glm::vec2), &mesh.vts[0], ↩
    GL_DYNAMIC_DRAW);

    glGenBuffers(1, &normalbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
    glBufferData(GL_ARRAY_BUFFER,
        mesh.vns.size() * sizeof(glm::vec3), &mesh.vns[0], ↩
    GL_DYNAMIC_DRAW);

    glGenBuffers(1, &elementbuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
        mesh.fs.size() * sizeof(unsigned short), &mesh.fs[0], ↩
    GL_DYNAMIC_DRAW);

    // VAO
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);
    if (isloadDDS)
    {
        file = "res/" + filename + ".DDS";
        Texture = loadDDS(file.c_str());
        GLuint TextureID = glGetUniformLocation(program0, "↩
    textureSampler");
        // Bind our texture in Texture Unit 0
        glActiveTexture(GL_TEXTURE0 + 1);
        glBindTexture(GL_TEXTURE_2D, Texture);
        glActiveTexture(GL_TEXTURE0);
        glUseProgram(program0);
        glUniform1i(TextureID, 1);
        glUseProgram(0);
```

```
    }

    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(
        0,                  // attribute
        3,                  // size
        GL_FLOAT,           // type
        GL_FALSE,           // normalized?
        0,                  // stride
        (void*)0            // array buffer offset
    );

    glEnableVertexAttribArray(1);
    glBindBuffer(GL_ARRAY_BUFFER, texturebuffer);
    glVertexAttribPointer(
        1,                                  // attribute
        2,                                  // size
        GL_FLOAT,                           // type
        GL_FALSE,                           // normalized?
        0,                                  // stride
        (void*)0                            // array buffer ↩
    offset
    );
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
    glVertexAttribPointer(
        2,                                  // attribute
        3,                                  // size
        GL_FLOAT,                           // type
        GL_FALSE,                           // normalized?
        0,                                  // stride
        (void*)0                            // array buffer ↩
    offset
    );

}

void GLAPIENTRY
MessageCallback(GLenum source,
    GLenum type,
    GLuint id,
    GLenum severity,
    GLsizei length,
    const GLchar* message,
    const void* userParam)
```

```
{
    fprintf(stderr, "GL␣CALLBACK:␣%s␣type␣=␣0x%x,␣severity␣=␣0x↩
    %x,␣message␣=␣%s\n",
        (type == GL_DEBUG_TYPE_ERROR ? "**␣GL␣ERROR␣**" : ""),
        type, severity, message);
}


void init()
{
    if (!glfwInit())
    {
        fprintf(stderr, "Failed␣to␣initialize␣GLFW\n");
        getchar();
        exit(-1);
    }


    glfwDefaultWindowHints();
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, ↩
    GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, true);
    glfwWindowHint(GLFW_VISIBLE, false);
    glfwWindowHint(GLFW_RESIZABLE, false);

    window = glfwCreateWindow(wwidth, wheight, "exp01", NULL, ↩
    NULL);
    if (window == NULL) {
        fprintf(stderr, "Failed␣to␣open␣GLFW␣window.\n");
        getchar();
        glfwTerminate();
        exit(-1);
    }

    printf("press␣'esc'␣to␣ternimate␣program\n"
        "press␣'wasdqe'␣to␣move␣camera\n"
        "press␣'fghtry'␣to␣rotate␣camera\n");
    glfwSetKeyCallback(window, [](GLFWwindow* window, int key, ↩
    int scancode, int action, int mode) {
        if (action == GLFW_PRESS)
        {
            if (key == GLFW_KEY_ESCAPE)
            {
```

```
            escPress = true;
        }
        else if (key == GLFW_KEY_A)
        {
            aPress = true;
        }
        else if (key == GLFW_KEY_S)
        {
            sPress = true;
        }
        else if (key == GLFW_KEY_D)
        {
            dPress = true;
        }
        else if (key == GLFW_KEY_W)
        {
            wPress = true;
        }
        else if (key == GLFW_KEY_Q)
        {
            qPress = true;
        }
        else if (key == GLFW_KEY_E)
        {
            ePress = true;
        }
        else if (key == GLFW_KEY_F)
        {
            fPress = true;
        }
        else if (key == GLFW_KEY_G)
        {
            gPress = true;
        }
        else if (key == GLFW_KEY_H)
        {
            hPress = true;
        }
        else if (key == GLFW_KEY_T)
        {
            tPress = true;
        }
        else if (key == GLFW_KEY_R)
        {
            rPress = true;
        }
```

```
            else if (key == GLFW_KEY_Y)
            {
                yPress = true;
            }

        }
        else if (action == GLFW_RELEASE)
        {
            if (key == GLFW_KEY_A)
            {
                aPress = false;
            }
            else if (key == GLFW_KEY_S)
            {
                sPress = false;
            }
            else if (key == GLFW_KEY_D)
            {
                dPress = false;
            }
            else if (key == GLFW_KEY_W)
            {
                wPress = false;
            }
            else if (key == GLFW_KEY_Q)
            {
                qPress = false;
            }
            else if (key == GLFW_KEY_E)
            {
                ePress = false;
            }
            else if (key == GLFW_KEY_F)
            {
                fPress = false;
            }
            else if (key == GLFW_KEY_G)
            {
                gPress = false;
            }
            else if (key == GLFW_KEY_H)
            {
                hPress = false;
            }
            else if (key == GLFW_KEY_T)
            {
```

```
            tPress = false;
        }
        else if (key == GLFW_KEY_R)
        {
            rPress = false;
        }
        else if (key == GLFW_KEY_Y)
        {
            yPress = false;
        }
    }
    });

const GLFWvidmode* vidmode = glfwGetVideoMode(↩
glfwGetPrimaryMonitor());
glfwSetWindowPos(
    window,
    (vidmode->width - wwidth) / 2,
    (vidmode->height - wheight) / 2
);
glfwMakeContextCurrent(window);
glfwSwapInterval(1);

// Initialize GLEW
glewExperimental = true; // Needed for core profile
if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Failed␣to␣initialize␣GLEW\n");
    getchar();
    glfwTerminate();
    exit(-1);
};

// enable debug output
glEnable(GL_DEBUG_OUTPUT);
glDebugMessageCallback(MessageCallback, NULL);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

program0 = LoadShaders("object.vs.glsl", "object.fs.glsl");
initShaders();
loadOBJtoPrg();

glfwShowWindow(window);

glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
```

```
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glEnable(GL_CULL_FACE);
}



void viewModel()
{
    glm::mat4 ProjectionMatrix = glm::perspective(
        glm::radians(60.0f), (float)wwidth / wheight, 1.0f, ↩
    100.0f);

    vDirection = glm::vec3(
        cos(vVerticalAngle) * sin(vHorizontalAngle),
        sin(vVerticalAngle),
        cos(vVerticalAngle) * cos(vHorizontalAngle)
    );

    vRight = glm::vec3(
        sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
        0,
        cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
    );

    vUp = glm::cross(vRight, vDirection);

    if (aPress == true)
    {
        vPosition -= vRight * viewMoveSpeed;
    }
    else if (sPress == true)
    {
        vPosition -= vDirection * viewMoveSpeed;
    }
    else if (dPress == true)
    {
        vPosition += vRight * viewMoveSpeed;
    }
    else if (wPress == true)
    {
        vPosition += vDirection * viewMoveSpeed;
    }
    else if (qPress == true)
```

```
{
    vPosition += vUp * viewMoveSpeed;
}
else if (ePress == true)
{
    vPosition -= vUp * viewMoveSpeed;
}
else if (fPress == true)
{
    vHorizontalAngle += viewRotateSpeed;
}
else if (gPress == true)
{
    vVerticalAngle -= viewRotateSpeed;
}
else if (hPress == true)
{
    vHorizontalAngle -= viewRotateSpeed;
}
else if (tPress == true)
{
    vVerticalAngle += viewRotateSpeed;
}

vDirection = glm::vec3(
    cos(vVerticalAngle) * sin(vHorizontalAngle),
    sin(vVerticalAngle),
    cos(vVerticalAngle) * cos(vHorizontalAngle)
);

vRight = glm::vec3(
    sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
    0,
    cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
);

vUp = glm::cross(vRight, vDirection);


ViewMatrix = glm::lookAt(
    vPosition,
    vPosition + vDirection,
    vUp
);

MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;
```

```
}

Mesh moveMeshMat(Mesh mesh, mat4 m)
{

    for (int i = 0; i < mesh.vn; i++)
    {
        mesh.vs[i] = m * vec4(mesh.vs[i], 1);
        mesh.vns[i] = m * vec4(mesh.vns[i], 0);
    }
    return mesh;
}


void moveModel()
{
    currentRun += 1;
    float t = (float)(currentRun % runTime) / runTime;
    // 0 lefthip leftknee leftankle
    // -0.4,5.7,-0.33   0,3.11,-0.2 0.2,0.63,-0.16
    // 3 neck head body
    // 6 righthip rightknee rightankle
    // -0.4,5.7,0.5 0,3.11,0.3  0.2,0.63,0.35
    // 9 leftshoulder leftelbow leftwrist
    // 0,8.37,-1.22 0,6.37,-1.2 0,4.36,-1.11
    // 12 rightelbow rightshoulder rightwrist
    //  0,6.36,1.40   0,8.37,1.44   0,4.36,1.33

    //  left arm
    float armAngle0 = glm::pi<float>() * 1 / 4.0;
    float armAngle1 = glm::pi<float>() * 1 / 2.0;
    float armAngle2 = glm::pi<float>() * 2 / 3.0;
    mat4 r = glm::toMat4(glm::quat(
        cos(armAngle0 * (abs(0.5 - t)-0.25) / 2),
        0,
        0,
        1 * sin(armAngle0 * (abs(0.5 - t) - 0.25) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0, 4.36, -1.11))
        * r
        * glm::translate(mat4(1.0), -vec3(0, 4.36, -1.11));
    Mesh n0 = moveMeshMat(meshes[11], r);

    n0 = merge(meshes[10], n0);
    r = glm::toMat4(glm::quat(
        cos(armAngle1 * abs(0.5 - t) / 2),
```

```
        0,
        0,
        1 * sin(armAngle1 * abs(0.5 - t) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0, 6.37, -1.2))
        * r
        * glm::translate(mat4(1.0), -vec3(0, 6.37, -1.2));
    n0 = moveMeshMat(n0, r);
    n0 = merge(meshes[9], n0);
    r = glm::toMat4(glm::quat(
        cos(armAngle2 * (abs(0.5 - t) - 0.25) / 2),
        0,
        0,
        1 * sin(armAngle2 * (abs(0.5 - t) - 0.25) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0, 8.37, -1.22))
        * r
        * glm::translate(mat4(1.0), -vec3(0, 8.37, -1.22));
    n0 = moveMeshMat(n0, r);

    //right leg
    float legAngle0 = glm::pi<float>() *1.0/2;
    float legAngle1 = glm::pi<float>() * 1 / 3.0;
    float legAngle2 = glm::pi<float>() * 4 / 4.0;
    r = glm::toMat4(glm::quat(
        cos(legAngle0 * (0.25 - abs(0.5 - t)) / 2),
        0,
        0,
        1 * sin(legAngle0 * (0.25 - abs(0.5 - t)) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0.2, 0.63, 0.35))
        * r
        * glm::translate(mat4(1.0), -vec3(0.2, 0.63, 0.35));
    Mesh n1 = moveMeshMat(meshes[8], r);

    n1 = merge(meshes[7], n1);
    r = glm::toMat4(glm::quat(
        cos(legAngle1 * (abs(0.5 - t) - 0.5) / 2),
        0,
        0,
        1 * sin(legAngle1 * (abs(0.5 - t) - 0.5) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0, 3.11, 0.3))
        * r
        * glm::translate(mat4(1.0), -vec3(0, 3.11, 0.3));
    n1 = moveMeshMat(n1, r);
```

```
n1 = merge(meshes[6], n1);
r = glm::toMat4(glm::quat(
    cos(legAngle2 * (abs(0.5 - t) - 0.25) / 2),
    0,
    0,
    1 * sin(legAngle2 * (abs(0.5 - t) - 0.25) / 2)
));
r = glm::translate(mat4(1.0), vec3(-0.4, 5.7, 0.5))
    * r
    * glm::translate(mat4(1.0), -vec3(-0.4, 5.7, 0.5));
n1 = moveMeshMat(n1, r);
n0 = merge(n0, n1);
//right arm
r = glm::toMat4(glm::quat(
    cos(armAngle0 * abs(0.25 - t)/ 2),
    0,
    0,
    1 * sin(armAngle0 * abs(0.25 - t) / 2)
));
r = glm::translate(mat4(1.0), vec3(0, 4.36, 1.33))
    * r
    * glm::translate(mat4(1.0), -vec3(0, 4.36, 1.33));
Mesh n2 = moveMeshMat(meshes[14], r);

n2 = merge(meshes[12], n2);
r = glm::toMat4(glm::quat(
    cos(armAngle1 * (0.5-abs(0.5 - t)) / 2),
    0,
    0,
    1 * sin(armAngle1 * (0.5 - abs(0.5 - t)) / 2)
));
r = glm::translate(mat4(1.0), vec3(0, 6.36, 1.40))
    * r
    * glm::translate(mat4(1.0), -vec3(0, 6.36, 1.40));
n2 = moveMeshMat(n2, r);

n2 = merge(meshes[13], n2);
r = glm::toMat4(glm::quat(
    cos(armAngle2 * (0.25 - abs(0.5 - t)) / 2),
    0,
    0,
    1 * sin(armAngle2 * (0.25 - abs(0.5 - t)) / 2)
));
r = glm::translate(mat4(1.0), vec3(0, 8.37, 1.44))
    * r
```

```
        * glm::translate(mat4(1.0), -vec3(0, 8.37, 1.44));
    n2 = moveMeshMat(n2, r);
    n0 = merge(n0, n2);

    // left leg
    r = glm::toMat4(glm::quat(
        cos(legAngle0 * (-0.25 + abs(0.5 - t)) / 2),
        0,
        0,
        1 * sin(legAngle0 * (-0.25 + abs(0.5 - t)) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0.2, 0.63, -0.16))
        * r
        * glm::translate(mat4(1.0), -vec3(0.2, 0.63, -0.16));
    Mesh n3 = moveMeshMat(meshes[2], r);

    n3 = merge(meshes[1], n3);
    r = glm::toMat4(glm::quat(
        cos(legAngle1 * -abs(0.5 - t) / 2),
        0,
        0,
        1 * sin(legAngle1 * -abs(0.5 - t) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(0, 3.11, -0.2))
        * r
        * glm::translate(mat4(1.0), -vec3(0, 3.11, -0.2));
    n3 = moveMeshMat(n3, r);

    n3 = merge(meshes[0], n3);
    r = glm::toMat4(glm::quat(
        cos(legAngle2 * (-abs(0.5 - t) + 0.25) / 2),
        0,
        0,
        1 * sin(legAngle2 * (-abs(0.5 - t) + 0.25) / 2)
    ));
    r = glm::translate(mat4(1.0), vec3(-0.4, 5.7, -0.33))
        * r
        * glm::translate(mat4(1.0), -vec3(-0.4, 5.7, -0.33));
    n3 = moveMeshMat(n3, r);
    n0 = merge(n0, n3);
    //body
    n0 = merge(n0, meshes[3]);
    n0 = merge(n0, meshes[4]);
    n0 = merge(n0, meshes[5]);
    r = glm::translate(mat4(1.0), vec3(4.0 * fmod(( (float)↩
    currentRun / runTime ),5), 0, 0));
```

```
    n0=moveMeshMat(n0, r);

    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glBufferData(GL_ARRAY_BUFFER,
        n0.vs.size() * sizeof(glm::vec3), &n0.vs[0], ↩
    GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
    glBufferData(GL_ARRAY_BUFFER,
        n0.vns.size() * sizeof(glm::vec3), &n0.vns[0], ↩
    GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, 0);


}

void loop()
{

    while (escPress != true && glfwWindowShouldClose(window) ==↩
     0)
    {
        glfwPollEvents();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glUseProgram(program0);

        viewModel();
        moveModel();

        glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
        glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &↩
    ModelMatrix[0][0]);
        glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &↩
    ViewMatrix[0][0]);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);

        glDrawElements(
            GL_TRIANGLES,      // mode
            mesh.fs.size(),    // count
            GL_UNSIGNED_SHORT,   // type
            (void*)0           // element array buffer offset
        );
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
        glfwSwapBuffers(window);
```

```
        glUseProgram(0);
    }

    glfwTerminate();
}

int main(int argc, char* argv[]) {
    init();
    loop();
}
```

Code 8.2: object.fs.glsl

```
#version 430 core

in vec2 UV;
in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;

uniform bool useTexture;
uniform sampler2D textureSampler;

out vec4 color;

void main(){

    // Light emission properties
    vec3 LightColor = vec3(1,1,1);
    float LightPower = 50.0f;

    // Material properties
    vec3 MaterialDiffuseColor = vec3(0.5,0.5,1.0);
    if(useTexture)
    {
        MaterialDiffuseColor=texture( textureSampler, UV ).rgb;
    };
    vec3 MaterialAmbientColor = vec3(0.1,0.1,0.1) * ↩
    MaterialDiffuseColor;
    vec3 MaterialSpecularColor = vec3(0.3,0.3,0.3);

    float distance = length( vec3(7,7,10) - Position_worldspace↩
     );
```

```
    vec3 n = normalize( Normal_cameraspace );
    vec3 l = normalize( LightDirection_cameraspace );
    float cosTheta = clamp( dot( n,l ), 0,1 );

    vec3 E = normalize(EyeDirection_cameraspace);
    vec3 R = reflect(-l,n);
    float cosAlpha = clamp( dot( E,R ), 0,1 );

    color.rgb=// Ambient : simulates indirect lighting
        MaterialAmbientColor +
        // Diffuse
        MaterialDiffuseColor * LightColor * LightPower * ↩
    cosTheta / (distance*distance) +
        // Specular
        MaterialSpecularColor * LightColor * LightPower * pow(↩
    cosAlpha,5) / (distance*distance);
    color.a=1;
}
```

Code 8.3: object.vs.glsl

```
#version 430 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

out vec2 UV;
out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;

uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;

void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace, 1);
    Position_worldspace = (M * vec4(vertexPosition_modelspace↩
    ,1)).xyz;

    vec3 vertexPosition_cameraspace = ( V * M * vec4(↩
    vertexPosition_modelspace,1)).xyz;
    EyeDirection_cameraspace = vec3(0,0,0) - ↩
```

```
    vertexPosition_cameraspace;

    vec3 LightPosition_cameraspace = ( V * vec4(7,7,10,1)).xyz;
    LightDirection_cameraspace = LightPosition_cameraspace + ↩
    EyeDirection_cameraspace;

    Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace↩
    ,0)).xyz;
    UV = vertexUV;
}
```

Code 8.4: utils/loadTexture.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

GLuint loadDDS(const char* imagepath);
```

Code 8.5: utils/loadTexture.cpp

```
#include "loadTexture.h"

/*
* modified from: ogl-master
* common/texture.cpp
* */
#define FOURCC_DXT1 0x31545844 // Equivalent to "DXT1" in ASCII
#define FOURCC_DXT3 0x33545844 // Equivalent to "DXT3" in ASCII
#define FOURCC_DXT5 0x35545844 // Equivalent to "DXT5" in ASCII

GLuint loadDDS(const char* imagepath) {

    unsigned char header[124];

    FILE* fp;

    /* try to open the file */
    fp = fopen(imagepath, "rb");
```

```
    if (fp == NULL) {
        printf("%s␣could␣not␣be␣opened.\n", imagepath); getchar↩
    ();
        return 0;
    }

    /* verify the type of file */
    char filecode[4];
    fread(filecode, 1, 4, fp);
    if (strncmp(filecode, "DDS␣", 4) != 0) {
        fclose(fp);
        return 0;
    }

    /* get the surface desc */
    fread(&header, 124, 1, fp);

    unsigned int height = *(unsigned int*)&(header[8]);
    unsigned int width = *(unsigned int*)&(header[12]);
    unsigned int linearSize = *(unsigned int*)&(header[16]);
    unsigned int mipMapCount = *(unsigned int*)&(header[24]);
    unsigned int fourCC = *(unsigned int*)&(header[80]);


    unsigned char* buffer;
    unsigned int bufsize;
    /* how big is it going to be including all mipmaps? */
    bufsize = mipMapCount > 1 ? linearSize * 2 : linearSize;
    buffer = (unsigned char*)malloc(bufsize * sizeof(unsigned ↩
    char));
    fread(buffer, 1, bufsize, fp);
    /* close the file pointer */
    fclose(fp);

    unsigned int components = (fourCC == FOURCC_DXT1) ? 3 : 4;
    unsigned int format;
    switch (fourCC)
    {
    case FOURCC_DXT1:
        format = GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
        break;
    case FOURCC_DXT3:
        format = GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
        break;
    case FOURCC_DXT5:
        format = GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
```

```
        break;
    default:
        free(buffer);
        return 0;
    }

    // Create one OpenGL texture
    GLuint textureID;
    glGenTextures(1, &textureID);

    // "Bind" the newly created texture : all future texture ↩
    functions will modify this texture
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    unsigned int blockSize = (format == ↩
    GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) ? 8 : 16;
    unsigned int offset = 0;

    /* load the mipmaps */
    for (unsigned int level = 0; level < mipMapCount && (width ↩
    || height); ++level)
    {
        unsigned int size = ((width + 3) / 4) * ((height + 3) /↩
     4) * blockSize;
        glCompressedTexImage2D(GL_TEXTURE_2D, level, format, ↩
    width, height,
            0, size, buffer + offset);

        offset += size;
        width /= 2;
        height /= 2;

        // Deal with Non-Power-Of-Two textures. This code is ↩
    not included in the webpage to reduce clutter.
        if (width < 1) width = 1;
        if (height < 1) height = 1;

    }

    free(buffer);

    return textureID;


}
```

Code 8.6: utils/meshio.h

```
#pragma once
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <sstream>
#include <cstring>
#include <stdlib.h>

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <GLFW/glfw3.h>

// Include AssImp
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // Output data structure
#include <assimp/postprocess.h>     // Post processing flags
using std::vector;
using glm::vec2;
using glm::vec3;
using glm::vec4;

struct Mesh
{
    int vn=0;
    int fn=0;
    vector<vec3> vs;
    vector<vec2> vts;
    vector<vec3> vns;
    vector<short> fs;
};

Mesh merge(Mesh a, Mesh b);

bool loadObj(
    bool hasDDS,
    const char* path,
    vector<Mesh>& meshes,
    Mesh& mesh
```

```
);

void LoadShader(GLuint ShaderID, const char* file_path);

GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path);
```

Code 8.7: utils/meshio.cpp

```
#include "meshio.h"

Mesh merge(Mesh a, Mesh b)
{
    Mesh c = a;
    c.fn += b.fn;
    c.vn += b.vn;
    c.vs.reserve(c.vn);
    c.vs.insert(c.vs.end(), b.vs.begin(), b.vs.end());
    c.vts.reserve(c.vn);
    c.vts.insert(c.vts.end(), b.vts.begin(), b.vts.end());
    c.vns.reserve(c.vn);
    c.vns.insert(c.vns.end(), b.vns.begin(), b.vns.end());
    c.fs.reserve(c.fn * 3);
    for (int i = 0; i < b.fn; i++)
    {
        c.fs.push_back(b.fs[i * 3] + a.vn);
        c.fs.push_back(b.fs[i * 3 + 1] + a.vn);
        c.fs.push_back(b.fs[i * 3 + 2] + a.vn);
    }
    return c;
}

/*
* modified from: ogl-master
* common/objloader.cpp
* author: opengl-tutorial.org
*/
// load multiple meshes
bool loadObj(
    bool hasDDS,
    const char* path,
    vector<Mesh>& meshes,
    Mesh& mesh0
) {
    Assimp::Importer importer;
```

```
const aiScene* scene = importer.ReadFile(
    path,
    0       /*aiProcess_JoinIdenticalVertices | ↩
aiProcess_SortByPType*/
);
if (!scene) {
    fprintf(stderr, importer.GetErrorString());
    getchar();
    return false;
}

// for now the models donnot overlap
for (int j = 0; j < scene->mNumMeshes; j++)
{
    Mesh mesh;
    int vertexnum = mesh0.vs.size();
    const aiMesh* inmesh = scene->mMeshes[j];

    mesh.vn += inmesh->mNumVertices;
    mesh.fn += inmesh->mNumFaces;
    mesh0.vn += inmesh->mNumVertices;
    mesh0.fn += inmesh->mNumFaces;

    // Fill vertices positions
    mesh0.vs.reserve(vertexnum + inmesh->mNumVertices);
    mesh.vs.reserve(inmesh->mNumVertices);
    for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
 {
        aiVector3D pos = inmesh->mVertices[i];
        mesh.vs.push_back(vec3(pos.x, pos.y, pos.z));
        mesh0.vs.push_back(vec3(pos.x, pos.y, pos.z));
    }

    mesh.vts.reserve(inmesh->mNumVertices);
    mesh0.vts.reserve(vertexnum + inmesh->mNumVertices);
    // no texture
    if (hasDDS)
    {
        // Fill vertices texture coordinates
        for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
i++) {
            aiVector3D UVW = inmesh->mTextureCoords[j][i];
            // Assume only 1 set of UV coords; AssImp ↩
supports 8 UV sets.
            mesh.vts.push_back(vec2(UVW.x, -UVW.y));
```

```
                mesh0.vts.push_back(vec2(UVW.x, -UVW.y));
            }
        }
        else
        {
            for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
    i++) {
                mesh.vts.push_back(vec2(0, 0));
                mesh0.vts.push_back(vec2(0, 0));
            }
        }

        // Fill vertices normals
        mesh.vns.reserve(inmesh->mNumVertices);
        mesh0.vns.reserve(vertexnum + inmesh->mNumVertices);
        for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
     {
            aiVector3D n = inmesh->mNormals[i];
            mesh.vns.push_back(glm::vec3(n.x, n.y, n.z));
            mesh0.vns.push_back(glm::vec3(n.x, n.y, n.z));
        }

        // Fill face indices
        mesh.fs.reserve( 3 * inmesh->mNumFaces);
        mesh0.fs.reserve(mesh0.fs.size() + 3 * inmesh->↩
    mNumFaces);
        for (unsigned int i = 0; i < inmesh->mNumFaces; i++) {
            // Assume the model has only triangles.
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[0]);
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[1]);
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[2]);
            mesh0.fs.push_back(inmesh->mFaces[i].mIndices[0] + ↩
    vertexnum);
            mesh0.fs.push_back(inmesh->mFaces[i].mIndices[1] + ↩
    vertexnum);
            mesh0.fs.push_back(inmesh->mFaces[i].mIndices[2] + ↩
    vertexnum);
        }
        meshes.push_back(mesh);
    }

    return true;
};


void LoadShader(GLuint ShaderID, const char* file_path)
```

```
{
    // Read the Vertex Shader code from the file
    std::string ShaderCode;
    std::ifstream ShaderStream(file_path, std::ios::in);
    if (ShaderStream.is_open()) {
        std::stringstream sstr;
        sstr << ShaderStream.rdbuf();
        ShaderCode = sstr.str();
        ShaderStream.close();
    }
    else {
        printf("Impossible␣to␣open␣%s.␣\n", file_path);
        getchar();
        exit(-1);
    }

    GLint Result = GL_FALSE;
    int InfoLogLength;

    printf("Compiling␣shader␣:␣%s\n", file_path);
    char const* xSourcePointer = ShaderCode.c_str();
    glShaderSource(ShaderID, 1, &xSourcePointer, NULL);
    glCompileShader(ShaderID);

    glGetShaderiv(ShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(ShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength)↩
    ;
    if (InfoLogLength > 0) {
        std::vector<char> ShaderErrorMessage(InfoLogLength + 1)↩
    ;
        glGetShaderInfoLog(ShaderID, InfoLogLength, NULL, &↩
    ShaderErrorMessage[0]);
        printf("%s\n", &ShaderErrorMessage[0]);
    }
}



GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path) {

    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER↩
    );
```

```
    LoadShader(VertexShaderID, vertex_file_path);
    LoadShader(FragmentShaderID, fragment_file_path);

    GLint Result = GL_FALSE;
    int InfoLogLength;

    // Link the program
    printf("Linking program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);

    // Check the program
    glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &↩
    InfoLogLength);
    if (InfoLogLength > 0) {
        std::vector<char> ProgramErrorMessage(InfoLogLength + ↩
    1);
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &↩
    ProgramErrorMessage[0]);
        printf("%s\n", &ProgramErrorMessage[0]);
    }


    glDetachShader(ProgramID, VertexShaderID);
    glDetachShader(ProgramID, FragmentShaderID);

    glDeleteShader(VertexShaderID);
    glDeleteShader(FragmentShaderID);

    return ProgramID;
}
```

Part III

Lab 2 Ray Tracing

Chapter 9

Requirement

9.1 Basic Requirement

The basic requirement for the experiment is shown below:

1. Implement ray tracing with shadow and reflection.
2. Draw the model with experiment 00, compare the effect.

9.2 Extra

There are also some extra functions the code implements:

1. Read model in triangles from a file.
2. Read the texture file in DDS, whether to use it depending on setting in the begining of code.
3. Camera postion and direction can be set with user interaction.
4. Implemention of *Phong shading model* in *OenGL4* *Shader*.
5. Implemention of position interoration for texture mapping.

Chapter 10

Theory and Application

As show in [8], ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects. The target is to achieve global illumination. We use the technique of path tracing here.

We trace ray that sent to the eye backwards, by pixel. Then the color is decided upon the point on object that ray intersects with.

The calculation part is down in *Compute Shader* and write to texture. Another *Shader* that draws a rectangle mapped to the scren will fill each pixel with the texture.

10.1 Ray Source Per Pixel

As 2.2.1 shows, the near side is projected onto the screen. By calculating the coordinates of corners in model space and interporation with pixels, we can generate ray source and direction for each pixel.

10.2 Intersection

Intersection is down for each mesh. To avoid calculation, we implement a bounding box firstly. Then we calculate the point that ray intersects with the plane at if ray casts within bounding box. After testing whether the point lies within the mesh, it stores the point and use that for rejecting following bounding boxes.

After testing all the bounding boxes, the intersection is calculated.

10.3 Coloring

We get texture position at vertices, then use interporation for sampling position of current point.

If no texture is used, default color is used.

10.4 Phong Shading

Phong shading adds interporation of triangle faces compared to Gourard shading. Normal of the intersection point is calculated the same way as for the texture sampler position.

10.5 Lighting

We use *Phong Lighting* to generate local illumination. After that, global light from other objects should be added to color as seperate light sources.

Deciding where other lights are from corresponds to deciding the reflection direction of the ray sent from the eye. Since when a ray intersects with the mesh, it reflects around the normal. Samely, invert the ray target and source, and it also follows *Law of Reflection*, so we can get the source of light simply reflecting the reverse of ray reaching the eye around the normal.

To achieve more degree of realism, the reflection direction is calculated with *Bidirectional reflectance distribution function*, [1]. We use the reflection adding a random hemisphere to form a cone to sample from.

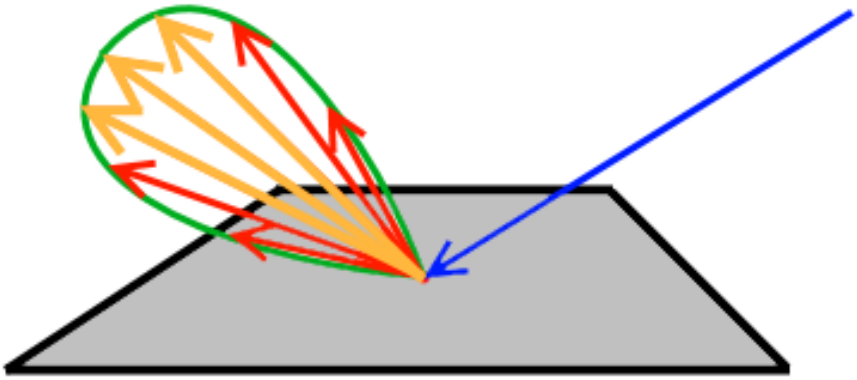

Figure 10.1: *BRDF* ,from [1]

The tracing of reflected ray is calculated the same way. The difference lies in that before adding to the pixel color, it should multiply material coefficients of the former meshes it reflects from.

10.6 Shadow Ray

To test whether the mesh is lighted, a ray is sent from the intersection point to the ray source. If it intersects before reaching the source, the mesh is blocked from the light, thus not implementing the diffuse and reflection part in *Phong Lighting*.

The lighted and blocked mesh are decided as below. It also deplicts indirect lighting.

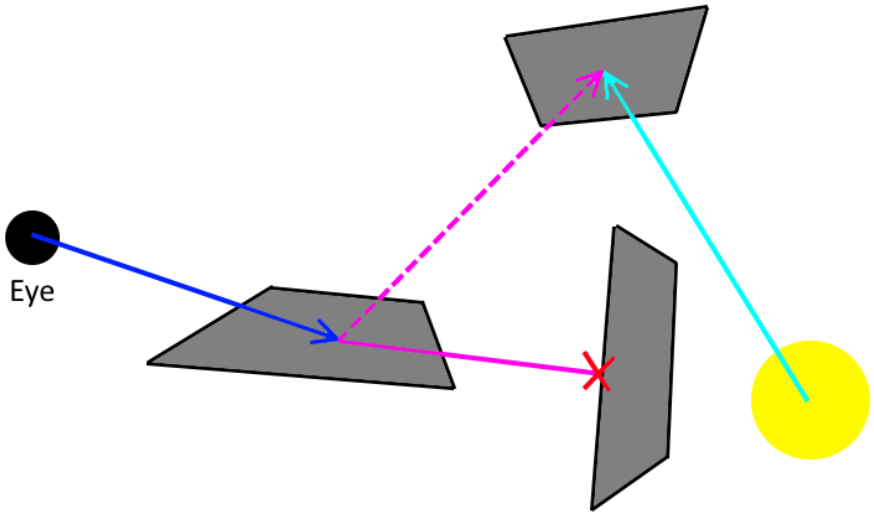

Figure 10.2: Shadow Ray, from [1]

Chapter 11

Program

11.1 Compiling Guide

First restore packets with *NuGet*, then compile the project with Visual Studio 2019.

11.1.1 Environment

OpenGL4, Windows 10, Visual Studio 2019 *(v142)*.

11.1.2 Packages Used

1. *assimp*
2. *glfw*
3. *glm*
4. *glew*

11.2 User Guide

The program can only be interacted with keyboard.

1. Input filename(without *.obj*) and whether to load texture. Empty input will lead to the default model.
2. Input bound times. Empty input will lead to *4*.
3. Long press *asdwqe* for changing position.

4. Long press *fghrty* for changing direction.

5. Press *esc* to exit.

11.3 Results

This is the result, with loads the same model as experiment 00 **??**. The bound times is set to 4;

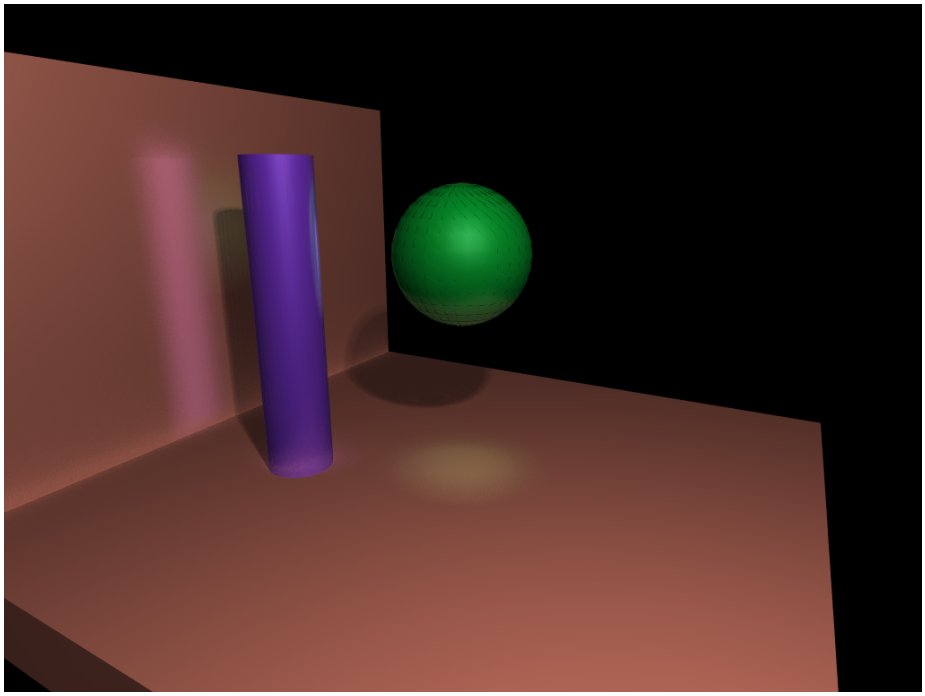

Figure 11.1: model *room_thickwalls*, bound 4

Bound times set to 1;

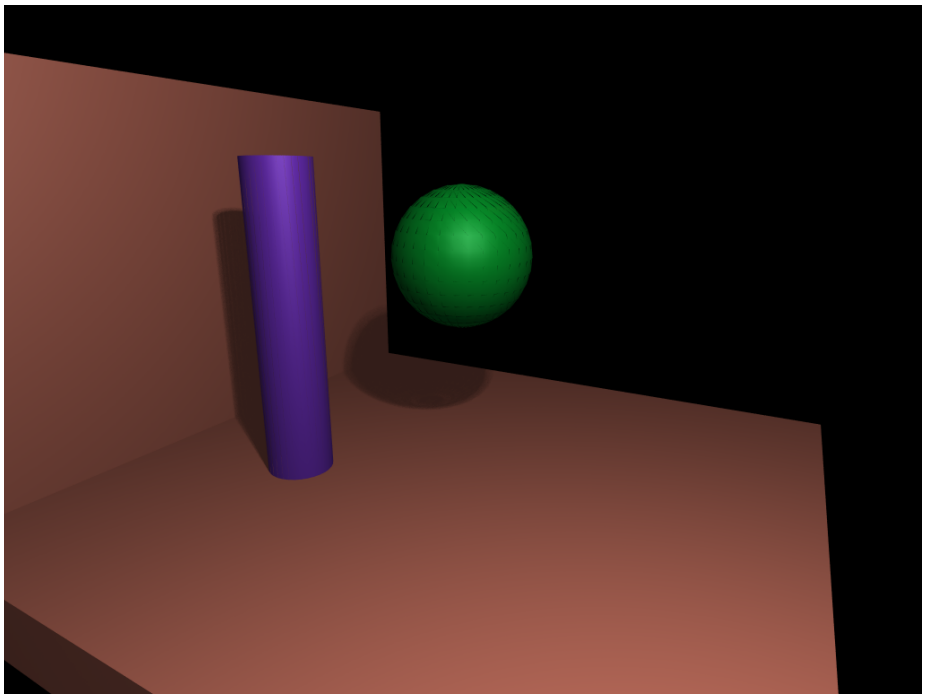

Figure 11.2: model *room_thickwalls*, bound 1

Figure 11.3: model *scene*, bound 4

Figure 11.4: model *scene*, bound 1

Figure 11.5: model *lwjgl3*, bound 4

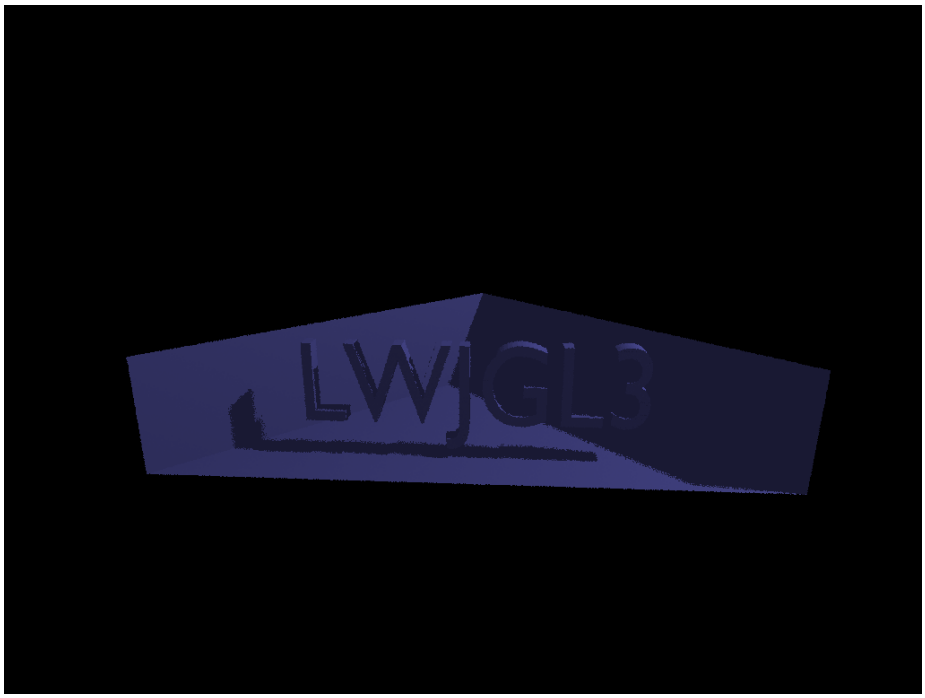

Figure 11.6: model *lwjgl3* , bound 1

11.3.1 Texture

The effect of texture postion interporation.

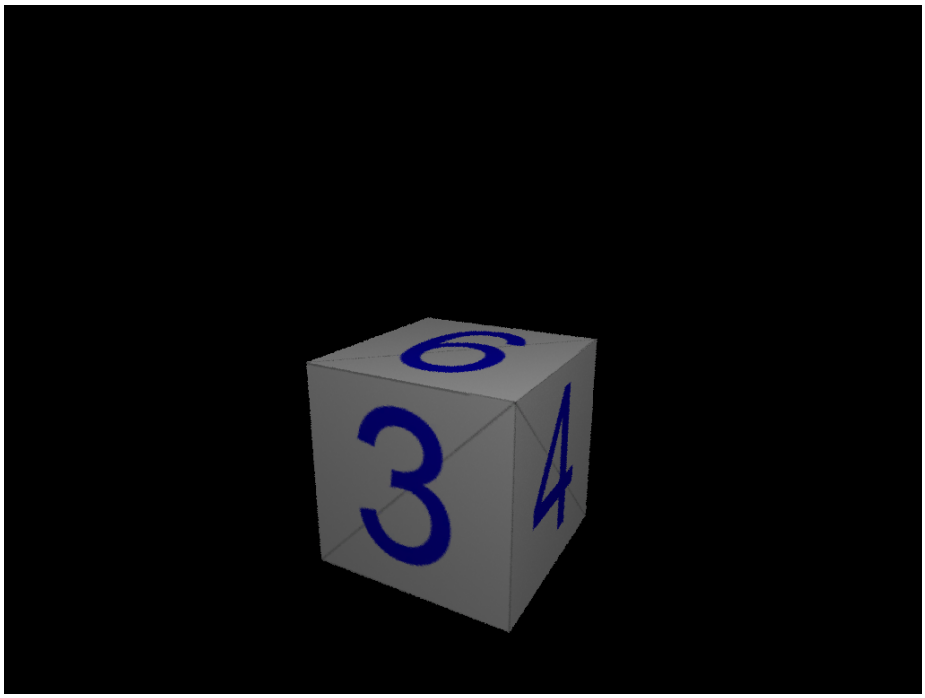

Figure 11.7: model *cube*, bound 4

11.3.2 Shading

The effect of Flat shading lies below, where each mesh has the same color. We can see *Phong Shading* has better effect.

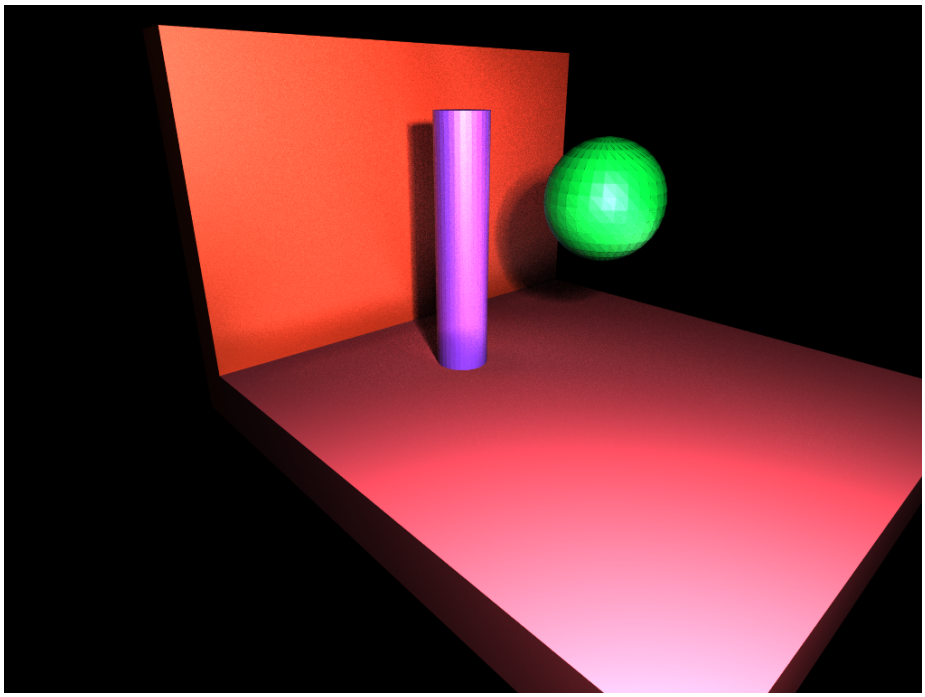

Figure 11.8: Flat shading

11.3.3 Problems

As shown above, there are empty space between meshes. We assume the problem is caused by inaccurate models and ignore them. For imporvement in software, interporation in image plane may work for the problem.

Chapter 12

Source Code

12.1 Structure

1. *main.cpp* : main code
2. *quad.fs.glsl* : image plane vertex shader file, from *LWJGL demo*
3. *quad.vs.glsl* : image plane fragment shader file, from *LWJGL demo*
4. *random.cs.glsl* : random library compute shader file, from *LWJGL demo*
5. *randomCommon.cs.glsl* : random library compute shader file, from *LWJGL demo*
6. *tracer.glsl* : ray tracing compute shader file
7. *utils*
   (a) *loadTexture.h* : functions for loading *.dds* file
   (b) *meshio.h* : functions for loading *.obj* file and shader file
8. *res/* :store model files and textures

12.2 Code files

Code 12.1: main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
```

```cpp
#include <math.h>
#include <chrono>

#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtx/quaternion.hpp>

#include <GLFW/glfw3.h>

#include "utils/loadTexture.h"
#include "utils/meshio.h"

#pragma comment( lib, "OpenGL32.lib" )
using std::string;
using glm::mat4;
using glm::vec4;
using glm::vec3;
using std::cout;
using std::cin;
using std::endl;

int wwidth = 1024, wheight = 768;
GLFWwindow* window;

const float viewMoveSpeedRatio = 0.1f;
const float viewRotateSpeedRatio = 0.1f;

// controls
bool escPress = false;
bool wPress = false;
bool aPress = false;
bool sPress = false;
bool dPress = false;
bool qPress = false;
bool ePress = false;
bool tPress = false;
bool fPress = false;
bool gPress = false;
bool hPress = false;
bool rPress = false;
bool yPress = false;
auto pressTime = std::chrono::high_resolution_clock::now();
```

```
// redraw
bool modified=false;
int frameNumber = 0;
//time
auto firstTime= std::chrono::high_resolution_clock::now();

glm::mat4 MVP;
glm::mat4 ModelMatrix = glm::mat4(1.0);
glm::mat4 ViewMatrix;
float vHorizontalAngle = -1.067;
float vVerticalAngle = -0.28;
glm::vec3 vPosition = glm::vec3(7.434, 4.477, -4.11);
glm::vec3 vDirection;
glm::vec3 vRight;
glm::vec3 vUp;
glm::mat4 ProjectionMatrix;
float zfar = 100;
float znear = 1;

// loading obj
bool isloadDDS = true;
string filename = "room_thickwalls" ;
Mesh mesh;
GLuint Texture;
GLuint vertexbuffer;
GLuint texturebuffer;
GLuint normalbuffer;
GLuint elementbuffer;

// quad
GLuint vao;
GLuint tex;
GLuint sampler;
GLuint quadProgram;
// compute shader
GLuint computeProgramID;
GLuint workGroupSizeX;
GLuint workGroupSizeY;
GLuint eyeUniform;
GLuint ray00Uniform;
GLuint ray10Uniform;
GLuint ray01Uniform;
GLuint ray11Uniform;
GLuint timeUniform;
GLuint blendFactorUniform;
GLuint bounceCountUniform;
```

```
GLuint framebufferImageBinding;

int bounceCount = 4;

void initShaders()
{
    glGenTextures(1,&tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA32F, wwidth, ↩
    wheight);
    glBindTexture(GL_TEXTURE_2D, 0);

    glGenSamplers(1, &sampler);
    glSamplerParameteri(sampler, GL_TEXTURE_MIN_FILTER, ↩
    GL_NEAREST);
    glSamplerParameteri(sampler, GL_TEXTURE_MAG_FILTER, ↩
    GL_NEAREST);

    quadProgram = LoadShaders("quad.vs.glsl", "quad.fs.glsl");
    glUseProgram(quadProgram);
    int texUniform = glGetUniformLocation(quadProgram, "tex");
    glUniform1i(texUniform, 0);
    glUseProgram(0);

    computeProgramID = glCreateProgram();
    GLuint cshader = glCreateShader(GL_COMPUTE_SHADER);
    LoadShader(cshader, "raytracer.cs.glsl");
    GLuint random = glCreateShader(GL_COMPUTE_SHADER);
    LoadShader(random, "random.cs.glsl");
    GLuint randomCommon = glCreateShader(GL_COMPUTE_SHADER);
    LoadShader(randomCommon, "randomCommon.cs.glsl");
    glAttachShader(computeProgramID, cshader);
    glAttachShader(computeProgramID, random);
    glAttachShader(computeProgramID, randomCommon);
    glLinkProgram(computeProgramID);
    GLint Result = GL_FALSE;
    int InfoLogLength;
    // Check the program
    glGetProgramiv(computeProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(computeProgramID, GL_INFO_LOG_LENGTH, &↩
    InfoLogLength);
    if (InfoLogLength > 0) {
        std::vector<char> ProgramErrorMessage(InfoLogLength + ↩
    1);
        glGetProgramInfoLog(computeProgramID, InfoLogLength, ↩
    NULL, &ProgramErrorMessage[0]);
```

```
        printf("%s\n", &ProgramErrorMessage[0]);
    }
    glDetachShader(computeProgramID, cshader);
    glDetachShader(computeProgramID, random);
    glDetachShader(computeProgramID, randomCommon);
    glDeleteShader(cshader);
    glDeleteShader(random);
    glDeleteShader(randomCommon);

    glUseProgram(computeProgramID);
    GLint workGroupSize[3];
    glGetProgramiv(computeProgramID, GL_COMPUTE_WORK_GROUP_SIZE↩
    , workGroupSize);
    workGroupSizeX = workGroupSize[0];
    workGroupSizeY = workGroupSize[1];
    eyeUniform = glGetUniformLocation(computeProgramID, "eye");
    ray00Uniform = glGetUniformLocation(computeProgramID, "↩
    ray00");
    ray10Uniform = glGetUniformLocation(computeProgramID, "↩
    ray10");
    ray01Uniform = glGetUniformLocation(computeProgramID, "↩
    ray01");
    ray11Uniform = glGetUniformLocation(computeProgramID, "↩
    ray11");
    timeUniform = glGetUniformLocation(computeProgramID, "time"↩
    );
    blendFactorUniform = glGetUniformLocation(computeProgramID,↩
     "blendFactor");
    bounceCountUniform = glGetUniformLocation(computeProgramID,↩
     "bounceCount");

    /* Query the "image binding point" of the image uniform */
    int params;
    int loc = glGetUniformLocation(computeProgramID, "↩
    framebufferImage");
    glGetUniformiv(computeProgramID, loc, &params);
    framebufferImageBinding = params;

    GLuint loadtexture = glGetUniformLocation(computeProgramID,↩
     "useTexture");
    glUniform1i(loadtexture, isloadDDS);
    glUseProgram(0);
}

void loadOBJtoPrg()
{
```

```
string file = "res/" + filename + ".obj";
if (!loadObj(isloadDDS,-1,-1, file.c_str() ,mesh))
{
    fprintf(stderr, "failed to load obj file");
    getchar();
    exit(-1);
}
glUseProgram(computeProgramID);
GLuint numberid = glGetUniformLocation(computeProgramID, "vn");
glUniform1i(numberid, mesh.vn);
numberid = glGetUniformLocation(computeProgramID, "fn");
glUniform1i(numberid, mesh.fn);

GLuint ssbo;
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER,
    mesh.vs.size() * sizeof(glm::vec4),
    &mesh.vs[0],
    GL_STATIC_READ);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER,
    mesh.vts.size() * sizeof(glm::vec2),
    &mesh.vts[0],
    GL_STATIC_READ);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER,
    mesh.vns.size() * sizeof(glm::vec4),
    &mesh.vns[0],
    GL_STATIC_READ);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER,
    mesh.fs.size() * sizeof(int),
```

```
        &mesh.fs[0],
        GL_STATIC_READ);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

    glGenBuffers(1, &ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
    glBufferData(GL_SHADER_STORAGE_BUFFER,
        mesh.maxvs.size() * sizeof(glm::vec4),
        &mesh.maxvs[0],
        GL_STATIC_READ);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

    glGenBuffers(1, &ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
    glBufferData(GL_SHADER_STORAGE_BUFFER,
        mesh.minvs.size() * sizeof(glm::vec4),
        &mesh.minvs[0],
        GL_STATIC_READ);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind

    glGenBuffers(1, &ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
    glBufferData(GL_SHADER_STORAGE_BUFFER,
        mesh.ns.size() * sizeof(glm::vec4),
        &mesh.ns[0],
        GL_STATIC_READ);
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 7, ssbo);
    glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // unbind


    if (isloadDDS)
    {
        file = "res/"+filename+".DDS";
        Texture = loadDDS(file.c_str());
        GLuint TextureID = glGetUniformLocation(↩
    computeProgramID, "textureSampler");
        // Bind our texture in Texture Unit 0
        glActiveTexture(GL_TEXTURE0+1);
        glBindTexture(GL_TEXTURE_2D, Texture);
        glActiveTexture(GL_TEXTURE0);

        glUniform1i(TextureID, 1);
    }
```

```
    glUseProgram(0);
}

void init()
{
    if (!glfwInit())
    {
        fprintf(stderr, "Failed␣to␣initialize␣GLFW\n");
        getchar();
        exit(-1);
    }


    glfwDefaultWindowHints();
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, ↩
    GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, true);
    glfwWindowHint(GLFW_VISIBLE, false);
    glfwWindowHint(GLFW_RESIZABLE, true);

    window = glfwCreateWindow(wwidth, wheight, "exp02", NULL, ↩
    NULL);
    if (window == NULL) {
        fprintf(stderr, "Failed␣to␣open␣GLFW␣window.\n");
        getchar();
        glfwTerminate();
        exit(-1);
    }

    printf("press␣'esc'␣to␣ternimate␣program\n"
        "press␣'wasdqe'␣to␣move␣object\n"
        "press␣'fghtry'␣to␣rotate␣object\n");
    glfwSetKeyCallback(window, [](GLFWwindow* window, int key, ↩
    int scancode, int action, int mode) {
        if (action == GLFW_PRESS)
        {
            modified = true;
            pressTime = std::chrono::high_resolution_clock::now↩
    ();
            if (key == GLFW_KEY_ESCAPE)
            {
                escPress = true;
            }
```

```
            else if (key == GLFW_KEY_A)
            {
                aPress = true;
            }
            else if (key == GLFW_KEY_S)
            {
                sPress = true;
            }
            else if (key == GLFW_KEY_D)
            {
                dPress = true;
            }
            else if (key == GLFW_KEY_W)
            {
                wPress = true;
            }
            else if (key == GLFW_KEY_Q)
            {
                qPress = true;
            }
            else if (key == GLFW_KEY_E)
            {
                ePress = true;
            }
            else if (key == GLFW_KEY_F)
            {
                fPress = true;
            }
            else if (key == GLFW_KEY_G)
            {
                gPress = true;
            }
            else if (key == GLFW_KEY_H)
            {
                hPress = true;
            }
            else if (key == GLFW_KEY_T)
            {
                tPress = true;
            }
            else if (key == GLFW_KEY_R)
            {
                rPress = true;
            }
            else if (key == GLFW_KEY_Y)
            {
```

```
            yPress = true;
        }
    }
    else if (action == GLFW_RELEASE)
    {
        modified = false;
        if (key == GLFW_KEY_A)
        {
            aPress = false;
        }
        else if (key == GLFW_KEY_S)
        {
            sPress = false;
        }
        else if (key == GLFW_KEY_D)
        {
            dPress = false;
        }
        else if (key == GLFW_KEY_W)
        {
            wPress = false;
        }
        else if (key == GLFW_KEY_Q)
        {
            qPress = false;
        }
        else if (key == GLFW_KEY_E)
        {
            ePress = false;
        }
        else if (key == GLFW_KEY_F)
        {
            fPress = false;
        }
        else if (key == GLFW_KEY_G)
        {
            gPress = false;
        }
        else if (key == GLFW_KEY_H)
        {
            hPress = false;
        }
        else if (key == GLFW_KEY_T)
        {
            tPress = false;
        }
```

```
            else if (key == GLFW_KEY_R)
            {
                rPress = false;
            }
            else if (key == GLFW_KEY_Y)
            {
                yPress = false;
            }
        }
        });

    const GLFWvidmode* vidmode = glfwGetVideoMode(↩
    glfwGetPrimaryMonitor());
    glfwSetWindowPos(
        window,
        (vidmode->width - wwidth) / 2,
        (vidmode->height - wheight) / 2
    );
    glfwMakeContextCurrent(window);
    glfwSwapInterval(1);

    // Initialize GLEW
    glewExperimental = true; // Needed for core profile
    if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Failed to initialize GLEW\n");
        getchar();
        glfwTerminate();
        exit(-1);
    };

    initShaders();
    loadOBJtoPrg();

    firstTime = std::chrono::high_resolution_clock::now();

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    glfwShowWindow(window);

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glViewport(0, 0, wwidth, wheight);
    glfwSwapBuffers(window);
}
```

```
void viewModel()
{
    ProjectionMatrix = glm::perspective(
        glm::radians(60.0f), (float)wwidth / wheight, znear, ↩
    zfar);

    vDirection = glm::vec3(
        cos(vVerticalAngle) * sin(vHorizontalAngle),
        sin(vVerticalAngle),
        cos(vVerticalAngle) * cos(vHorizontalAngle)
    );

    vRight = glm::vec3(
        sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
        0,
        cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
    );

    vUp = glm::cross(vRight, vDirection);

    auto thisTime = std::chrono::high_resolution_clock::now();
    float elapsedSeconds = std::chrono::duration_cast<std::↩
    chrono::nanoseconds>(thisTime-pressTime).count() / 1E9f;
    float viewMoveSpeed = elapsedSeconds * viewMoveSpeedRatio;
    float viewRotateSpeed = elapsedSeconds * ↩
    viewRotateSpeedRatio;
    if (aPress == true)
    {
        vPosition -= vRight * viewMoveSpeed;
    }
    else if (sPress == true)
    {
        vPosition -= vDirection * viewMoveSpeed;
    }
    else if (dPress == true)
    {
        vPosition += vRight * viewMoveSpeed;
    }
    else if (wPress == true)
    {
        vPosition += vDirection * viewMoveSpeed;
    }
    else if (qPress == true)
    {
```

```
        vPosition += vUp * viewMoveSpeed;
    }
    else if (ePress == true)
    {
        vPosition -= vUp * viewMoveSpeed;
    }
    else if (fPress == true)
    {
        vHorizontalAngle += viewRotateSpeed;
    }
    else if (gPress == true)
    {
        vVerticalAngle -= viewRotateSpeed;
    }
    else if (hPress == true)
    {
        vHorizontalAngle -= viewRotateSpeed;
    }
    else if (tPress == true)
    {
        vVerticalAngle += viewRotateSpeed;
    }

    vDirection = glm::vec3(
        cos(vVerticalAngle) * sin(vHorizontalAngle),
        sin(vVerticalAngle),
        cos(vVerticalAngle) * cos(vHorizontalAngle)
    );

    vRight = glm::vec3(
        sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
        0,
        cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
    );

    vUp = glm::cross(vRight, vDirection);


    ViewMatrix = glm::lookAt(
        vPosition,
        vPosition + vDirection,
        vUp
    );

    MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;
}
```

```
void loop()
{
    // reference: LWJGL demo/raytracing
    while (escPress != true && glfwWindowShouldClose(window) ==↩
     0)
    {
        glfwPollEvents();
        glViewport(0, 0, wwidth, wheight);
        // computeProgramID
        glUseProgram(computeProgramID);
        viewModel();

        if (modified) {
            frameNumber = 0;
        }
        mat4 invViewProjMatrix = glm::inverse(MVP);

        //time
        auto thisTime = std::chrono::high_resolution_clock::now↩
    ();
        float elapsedSeconds = std::chrono::duration_cast<std::↩
    chrono::nanoseconds>(thisTime-firstTime).count() / 1E9f;
        glUniform1f(timeUniform, elapsedSeconds);

        /*
            * We are going to average multiple successive ↩
    frames, so here we
            * compute the blend factor between old frame and ↩
    new frame. 0.0 - use
            * only the new frame > 0.0 - blend between old ↩
    frame and new frame
            */
        float blendFactor =  frameNumber / (frameNumber + 1.0f)↩
    ;
        glUniform1f(blendFactorUniform, blendFactor);
        glUniform1i(bounceCountUniform, bounceCount);

        /* Set viewing frustum corner rays in shader */
        float wsize = 1;
        vec3 tmpVector;
        glUniform3f(eyeUniform, vPosition.x, vPosition.y, ↩
    vPosition.z);
        tmpVector=vec3(-1, -1, -1);
        tmpVector = glm::vec3(invViewProjMatrix * vec4(↩
    tmpVector, 1.0)) - vPosition;
```

```
    glUniform3f(ray00Uniform, tmpVector.x, tmpVector.y, ↩
tmpVector.z);
    tmpVector = vec3(-1, 1 ,-1);
    tmpVector = glm::vec3(invViewProjMatrix * vec4(↩
tmpVector, 1.0)) - vPosition;
    glUniform3f(ray01Uniform, tmpVector.x, tmpVector.y, ↩
tmpVector.z);
    tmpVector = vec3(1, -1, -1);
    tmpVector = glm::vec3(invViewProjMatrix * vec4(↩
tmpVector, 1.0)) - vPosition;
    glUniform3f(ray10Uniform, tmpVector.x, tmpVector.y, ↩
tmpVector.z);
    tmpVector = vec3(1, 1, -1);
    tmpVector = glm::vec3(invViewProjMatrix * vec4(↩
tmpVector, 1.0)) - vPosition;
    glUniform3f(ray11Uniform, tmpVector.x, tmpVector.y, ↩
tmpVector.z);

    /* Bind level 0 of framebuffer texture as writable ↩
image in the shader. */
    glBindImageTexture(framebufferImageBinding, tex, 0, ↩
false, 0, GL_READ_WRITE, GL_RGBA32F);
    /*
        * Compute appropriate global work size dimensions.
        */
    int numGroupsX = (int)ceil((double)wwidth / ↩
workGroupSizeX);
    int numGroupsY = (int)ceil((double)wheight / ↩
workGroupSizeY);

    /* Invoke the compute shader. */
    glDispatchCompute(numGroupsX, numGroupsY, 1);
    /*
        * Synchronize all writes to the framebuffer image ↩
before we let OpenGL
        * source texels from it afterwards when rendering ↩
the final image with
        * the full-screen quad.
        */
    glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);

    /* Reset bindings. */
    glBindImageTexture(framebufferImageBinding, 0, 0, false↩
, 0, GL_READ_WRITE, GL_RGBA32F);
    glUseProgram(0);
```

```
        frameNumber++;

        //quadProgram
        glUseProgram(quadProgram);
        glBindVertexArray(vao);
        glBindTexture(GL_TEXTURE_2D, tex);
        glBindSampler(0, sampler);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        // unbind
        glBindSampler(0, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
        glBindVertexArray(0);
        glUseProgram(0);

        glfwSwapBuffers(window);
    }

    glfwTerminate();
}

int main(int argc, char* argv[]) {
    string yn, filen;
    cout << "please␣input␣filename:" << endl;
    getline(std::cin, filen);
    if (!filen.empty())
    {
        filename = filen;
        cout << "Does␣it␣has␣texture?␣please␣input␣y/n" << endl↩
    ;
        getline(std::cin, yn);
        if (!yn.empty())
        {
            if (yn.c_str()[0] == 'y')
                isloadDDS = true;
            else if (yn.c_str()[0] == 'n')
                isloadDDS = false;
            else
            {
                cout << "please␣input␣y/n" << endl;
                getchar();
                exit(-1);
            }
        }
    }
    cout << "filename:" << filename << "␣isloadtexture:" << ↩
    isloadDDS << endl;
```

```
    string bnum;
    cout << "please␣input␣bounce␣count:" << endl;
    getline(std::cin, bnum);
    if (!bnum.empty())
    {
        bounceCount = std::stoi(bnum);
        if (bounceCount <= 0)
            bounceCount = 1;
    }
    cout << "bounce␣count:" << bounceCount << endl;
    init();
    loop();
}
```

Code 12.2: quad.fs.glsl

```
/*
* Copyright LWJGL. All rights reserved.
* License terms: https://www.lwjgl.org/license
*/
#if __VERSION__ >= 130
    #define varying in
    #define texture2D texture

    out vec4 color;
    #define OUT color
#else
    #define OUT gl_FragColor
#endif

varying vec2 texcoord;

uniform sampler2D tex;

void main(void) {
    OUT = texture2D(tex, texcoord);
}
```

Code 12.3: quad.vs.glsl

```
/*
* Copyright LWJGL. All rights reserved.
* License terms: https://www.lwjgl.org/license
```

```
*/
#if __VERSION__ >= 130
    #define varying out
#endif

/* Write interpolated texture coordinate to fragment shader */
varying vec2 texcoord;

void main(void) {
    vec2 vertex = vec2(-1.0) + vec2(
    float((gl_VertexID & 1) << 2),
    float((gl_VertexID & 2) << 1));
    gl_Position = vec4(vertex, 0.0, 1.0);

    /*
    * Compute texture coordinate by simply
    * interval-mapping from [-1..+1] to [0..1]
    */
    texcoord = vertex * 0.5 + vec2(0.5, 0.5);
}
```

Code 12.4: raytracer.cs.glsl

```
#version 430 core
layout(binding = 0, rgba32f) uniform image2D framebufferImage;
layout(std430, binding = 1) buffer ln1
{
    vec4 vs[];
};
layout(std430, binding = 2) buffer ln2
{
    vec2 vts[];
};
layout(std430, binding = 3) buffer ln3
{
    vec4 vns[];
};
layout(std430, binding = 4) buffer ln4
{
    int fs[];
};
layout(std430, binding = 5) buffer ln5
{
    vec4 maxv[];
};
```

```
layout(std430, binding = 6) buffer ln6
{
    vec4 minv[];
};
layout(std430, binding = 7) buffer ln7
{
    vec4 ns[];
};

uniform bool useTexture;
uniform sampler2D textureSampler;

uniform vec3 eye;
uniform vec3 ray00;
uniform vec3 ray01;
uniform vec3 ray10;
uniform vec3 ray11;

uniform float blendFactor;
uniform float time;
uniform int bounceCount;

uniform int vn;
uniform int fn;

#define MAX_SCENE_BOUNDS 100.0

#define EPSILON 0.00001
#define LIGHT_RADIUS 0.2
#define MATERIAL_AMBIENT_CO 0.2
#define LIGHT_BASE_INTENSITY 40.0

const vec3 lightCenterPosition = vec3(6, 8, -5);
const vec4 lightColor = vec4(1.0);

int p=5;
vec4 MaterialSpecularColor = vec4(0.3,0.3,0.3,1.0);

vec3 random3(vec3 f);
vec3 randomDiskPoint(vec3 rand, vec3 n);
vec3 randomHemispherePoint(vec3 rand, vec3 n);
vec3 randomCosineWeightedHemispherePoint(vec3 rand, vec3 n);
vec3 randomSpherePoint(vec3 rand);

struct Hitinfo {
    vec3 point;
```

```
    float near;
    int faceid;
};

vec4 getColor(vec2 uv)
{
    if(useTexture)
        // implement interporation outside
        return texture( textureSampler, uv );
    else
        return vec4(0.5,0.5,1.0,1.0);
}

vec2 checkBoundingBox(vec3 origin, vec3 dir, int faceid) {
    vec3 tMin = (minv[faceid].xyz - origin) / dir;
    vec3 tMax = (maxv[faceid].xyz - origin) / dir;
    vec3 t1 = min(tMin, tMax);
    vec3 t2 = max(tMin, tMax);
    float tNear = max(max(t1.x, t1.y), t1.z);
    float tFar = min(min(t2.x, t2.y), t2.z);
    return vec2(tNear, tFar);
}

bool isSameDir(vec3 dir, int faceid, bool indir)
{
    if(dot(dir,ns[faceid].xyz)<0)
    {
        if (indir)
            return true;
        else
            return false;
    }
    if (indir)
        return false;
    else
        return true;
}

bool PointIn( vec3 P1, vec3 P2, vec3 A, vec3 B )
{
    vec3 CP1 = cross( B - A, P1 - A );
    vec3 CP2 = cross( B - A, P2 - A );
    return dot( CP1, CP2 ) >= 0;
}
bool PointInTriangle(vec3  P,vec3  A,vec3  B, vec3 C )
{
```

```
    return PointIn( P, A, B, C ) &&
            PointIn( P, B, C, A ) &&
            PointIn( P, C, A, B );
}

bool intersect(vec3 origin, vec3 dir, int faceid, bool indir, ↩
    out Hitinfo info)
{
    vec3 p0=vs[fs[faceid*3]].xyz;
    // we use interporated normal here
    vec3 normal=ns[faceid].xyz;
    if (!indir)
        normal=-normal;
    float t=abs(dot(p0-origin,normal)/dot(dir, ns[faceid].xyz))↩
    ;
    vec3 intersectPoint=origin+dir*t;
    if(PointInTriangle(intersectPoint,vs[fs[faceid*3]].xyz,vs[↩
    fs[faceid*3+1]].xyz,vs[fs[faceid*3+2]].xyz))
    {
        info.point=intersectPoint;
        info.near=t;
        info.faceid=faceid;
        return true;
    }
    return false;
}

bool intersectAll(vec3 origin, vec3 dir,  bool indir, out ↩
    Hitinfo info)
{
    float smallest = MAX_SCENE_BOUNDS;
    bool found=false;
    for (int i = 0; i<fn; i++)
    {
        if(isSameDir(dir, i, indir))
        {
            vec2 lambda = checkBoundingBox(origin, dir, i);
            if (lambda.y >= 0.0 && lambda.x <= lambda.y && ↩
    lambda.x < smallest)
            {
                if (intersect( origin,  dir,  i,  indir, info))
                {
                smallest = info.near;
                found = true;
                }
            }
```

```
        }
    }
    return found;
}

vec3 calNormal(int faceID, vec3 poz)
{
    int id1=faceID*3,id2=faceID*3+1,id3=faceID*3+2;
    if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
    {
        int tmp=id1;
        id1=id3;
        id3=tmp;
    }

    if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
    {
        if(abs(vs[fs[id2]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id3;
            id3=tmp;
        }
        else if(abs(vs[fs[id3]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id2;
            id2=tmp;
        }
        vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs[id3↩
    ]].xyz;
        float alpha1=(poz.z-ap.z)/(bp.z-ap.z),alpha2=(poz.z-ap.↩
    z)/(cp.z-ap.z);
        vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-alpha2↩
    )*ap;
        float beta=(poz.y-p1.y)/(p2.y-p1.y);

        vec3 an=vns[fs[id1]].xyz,bn=vns[fs[id2]].xyz,cn=vns[fs[↩
    id3]].xyz;
        return beta*(alpha2*cn+(1-alpha2)*an)+(1-beta)*(alpha1*↩
    bn+(1-alpha1)*an);
    }
    else
    {
        if(abs(vs[fs[id2]].y-vs[fs[id1]].y)<0.01)
        {
```

```
            int tmp=id1;
            id1=id3;
            id3=tmp;
        }
        if(abs(vs[fs[id2]].y-vs[fs[id1]].y)<0.01)
        {
            if(abs(vs[fs[id2]].z-vs[fs[id1]].z)<0.01)
            {
                int tmp=id1;
                id1=id3;
                id3=tmp;
            }
            else if(abs(vs[fs[id3]].z-vs[fs[id1]].z)<0.01)
            {
                int tmp=id1;
                id1=id2;
                id2=tmp;
            }
            vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs↩
[id3]].xyz;
            float alpha1=(poz.z-ap.z)/(bp.z-ap.z),alpha2=(poz.z↩
-ap.z)/(cp.z-ap.z);
            vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-↩
alpha2)*ap;
            float beta=(poz.x-p1.x)/(p2.x-p1.x);

            vec3 an=vns[fs[id1]].xyz,bn=vns[fs[id2]].xyz,cn=vns↩
[fs[id3]].xyz;
            return beta*(alpha2*cn+(1-alpha2)*an)+(1-beta)*(↩
alpha1*bn+(1-alpha1)*an);
        }
        if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
        {
            int tmp=id1;
            id1=id3;
            id3=tmp;
        }
        else if(abs(vs[fs[id3]].x-vs[fs[id1]].x)<0.01)
        {
            int tmp=id1;
            id1=id2;
            id2=tmp;
        }

        vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs[id3↩
]].xyz;
```

```
        float alpha1=(poz.x-ap.x)/(bp.x-ap.x),alpha2=(poz.x-ap.↩
    x)/(cp.x-ap.x);
        vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-alpha2↩
    )*ap;
        float beta=(poz.y-p1.y)/(p2.y-p1.y);

        vec3 an=vns[fs[id1]].xyz,bn=vns[fs[id2]].xyz,cn=vns[fs[↩
    id3]].xyz;
        return beta*(alpha2*cn+(1-alpha2)*an)+(1-beta)*(alpha1*↩
    bn+(1-alpha1)*an);
    }
}

vec2 calFace(int faceID, vec3 poz)
{
    int id1=faceID*3,id2=faceID*3+1,id3=faceID*3+2;
    if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
    {
        int tmp=id1;
        id1=id3;
        id3=tmp;
    }
    if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
    {
        // yz
        if(abs(vs[fs[id2]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id3;
            id3=tmp;
        }
        else if(abs(vs[fs[id3]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id2;
            id2=tmp;
        }
        vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs[id3↩
    ]].xyz;
        float alpha1=(poz.z-ap.z)/(bp.z-ap.z),alpha2=(poz.z-ap.↩
    z)/(cp.z-ap.z);
        vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-alpha2↩
    )*ap;
        float beta=(poz.y-p1.y)/(p2.y-p1.y);

        vec2 at=vts[fs[id1]],bt=vts[fs[id2]],ct=vts[fs[id3]];
```

```
        return beta*(alpha2*ct+(1-alpha2)*at)+(1-beta)*(alpha1*↩
bt+(1-alpha1)*at);
}
else
{
        if(abs(vs[fs[id2]].y-vs[fs[id1]].y)<0.01)
    {
        int tmp=id1;
        id1=id3;
        id3=tmp;
    }
    if(abs(vs[fs[id2]].y-vs[fs[id1]].y)<0.01)
    {
        if(abs(vs[fs[id2]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id3;
            id3=tmp;
        }
        else if(abs(vs[fs[id3]].z-vs[fs[id1]].z)<0.01)
        {
            int tmp=id1;
            id1=id2;
            id2=tmp;
        }
        // xz
        vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs↩
[id3]].xyz;
        float alpha1=(poz.z-ap.z)/(bp.z-ap.z),alpha2=(poz.z↩
-ap.z)/(cp.z-ap.z);
        vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-↩
alpha2)*ap;
        float beta=(poz.x-p1.x)/(p2.x-p1.x);

        vec2 at=vts[fs[id1]],bt=vts[fs[id2]],ct=vts[fs[id3↩
]];
        return beta*(alpha2*ct+(1-alpha2)*at)+(1-beta)*(↩
alpha1*bt+(1-alpha1)*at);
    }

    if(abs(vs[fs[id2]].x-vs[fs[id1]].x)<0.01)
    {
        int tmp=id1;
        id1=id3;
        id3=tmp;
    }
```

```
        else if(abs(vs[fs[id3]].x-vs[fs[id1]].x)<0.01)
        {
            int tmp=id1;
            id1=id2;
            id2=tmp;
        }
        // xy
        vec3 ap=vs[fs[id1]].xyz,bp=vs[fs[id2]].xyz,cp=vs[fs[id3]].xyz;
        float alpha1=(poz.x-ap.x)/(bp.x-ap.x),alpha2=(poz.x-ap.x)/(cp.x-ap.x);
        vec3 p1=alpha1*bp+(1-alpha1)*ap, p2=alpha2*cp+(1-alpha2)*ap;
        float beta=(poz.y-p1.y)/(p2.y-p1.y);

        vec2 at=vts[fs[id1]],bt=vts[fs[id2]],ct=vts[fs[id3]];
        return beta*(alpha2*ct+(1-alpha2)*at)+(1-beta)*(alpha1*bt+(1-alpha1)*at);
    }
}

/*
    * We need random values every now and then.
    * So, they will be precomputed for each ray we trace and
    * can be used by any function.
    */
vec3 rand;
vec3 cameraUp;

vec4 trace(vec3 origin, vec3 dir, int maxb)
{
    Hitinfo i;
    float distS=1;
    int count=0;
    vec4 accumulated = vec4(0.0);
    vec4 attenuation = vec4(1.0);
    for (int bounce = 0; bounce < maxb; bounce++) {
    if (intersectAll(origin, dir, true, i)) {
        int fi=i.faceid;
        vec3 hitPoint = origin + i.near * dir;
        vec3 normal = calNormal(fi, hitPoint);//ns[fi].xyz;
        vec3 lightNormal = normalize(hitPoint - lightCenterPosition);
        vec3 lightPosition = lightCenterPosition + randomDiskPoint(rand, lightNormal) * LIGHT_RADIUS;
        vec4 cColor=getColor(calFace(fi, hitPoint));
```

```
        // ambient
        if(bounce != 0)
        {
        distS*=clamp(dot(hitPoint-origin,hitPoint-origin)↩
    ,1,1.0/EPSILON);
        }
        accumulated += attenuation *cColor* MATERIAL_AMBIENT_CO↩
    /pow(distS,0.5);
        vec3 shadowRayDir = lightPosition - hitPoint;
        vec3 shadowRayStart = hitPoint + normal * EPSILON;
        Hitinfo shadowRayInfo;
        bool lightObstructed = intersectAll(shadowRayStart, ↩
    shadowRayDir, true, shadowRayInfo);
        // not obstructed by other objects
        if (shadowRayInfo.near >= 1.0) {
        count+=1;
        float cosineFallOff = clamp(dot(normal, normalize(↩
    shadowRayDir)),0,1);
        float oneOverR2 = 1.0 / pow(length(shadowRayDir),2)/↩
    distS;
        vec3 R = reflect(normalize(shadowRayDir),normal);
        float cosAlpha = clamp( dot( normalize(dir),R ), 0,1 );
        // diffuse
        accumulated += attenuation * cColor * lightColor * ↩
    LIGHT_BASE_INTENSITY * cosineFallOff * oneOverR2;
        // reflection
        accumulated += attenuation * MaterialSpecularColor * ↩
    lightColor * LIGHT_BASE_INTENSITY * pow(cosAlpha,p) * ↩
    oneOverR2;
        }
        origin = shadowRayStart;
        dir = randomCosineWeightedHemispherePoint(rand, normal)↩
    ;
        // cone centered at reflect direction
        dir = normalize(reflect(normalize(-dir),normal)+dir↩
    *1.2);
        attenuation *= dot(normal, dir);
    } else {
        break;
    }
    }

    return accumulated;
}
layout (local_size_x = 16, local_size_y = 8) in;
```

```
void main(void) {
    ivec2 pix = ivec2(gl_GlobalInvocationID.xy);
    ivec2 size = imageSize(framebufferImage);
    if (pix.x >= size.x || pix.y >= size.y) {
    return;
    }
    vec2 pos = (vec2(pix) + vec2(0.5, 0.5)) / vec2(size.x, size.y);
    vec4 color = vec4(0.0, 0.0, 0.0, 1.0);
    cameraUp = normalize(ray01 - ray00);
    for (int s = 0; s < 1; s++) {
    rand = random3(vec3(pix, time + float(s)));
    vec2 jitter = rand.xy / vec2(size);
    vec2 p = pos + jitter;
    vec3 dir = mix(mix(ray00, ray01, p.y), mix(ray10, ray11, p.y), p.x);
    color += trace(eye, dir, bounceCount);
    }
    color /= 1.0;
    vec4 oldColor = vec4(0.0);
    if (blendFactor > 0.0) {
    oldColor = imageLoad(framebufferImage, pix);
    }
    vec4 finalColor = mix(color, oldColor, blendFactor);
    imageStore(framebufferImage, pix, finalColor);
}
```

Code 12.5: random.cs.glsl

```
/*
* Copyright LWJGL. All rights reserved.
* License terms: https://www.lwjgl.org/license
*/
#version 330 core

/**
* http://www.jcgt.org/published/0009/03/02/
*/
uvec3 pcg3d(uvec3 v) {
    v = v * 1664525u + 1013904223u;
    v.x += v.y * v.z;
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    v ^= v >> 16u;
    v.x += v.y * v.z;
```

```
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    return v;
}
vec3 random3(vec3 f) {
    return uintBitsToFloat((pcg3d(floatBitsToUint(f)) & 0↩
    x007FFFFFu) | 0x3F800000u) - 1.0;
}
```

Code 12.6: randomCommon.cs.glsl

```
/*
* Copyright LWJGL. All rights reserved.
* License terms: https://www.lwjgl.org/license
*/
#version 330 core
#define PI 3.14159265359

/**
* Generate a uniformly distributed random point on the unit ↩
    disk oriented around 'n'.
*
* After:
* http://mathworld.wolfram.com/DiskPointPicking.html
*/
vec3 randomDiskPoint(vec3 rand, vec3 n) {
    float r = rand.x * 0.5 + 0.5; // [-1..1) -> [0..1)
    float angle = (rand.y + 1.0) * PI; // [-1..1] -> [0..2*PI)
    float sr = sqrt(r);
    vec2 p = vec2(sr * cos(angle), sr * sin(angle));
    /*
    * Compute some arbitrary tangent space for orienting
    * our disk towards the normal. We use the camera's up ↩
    vector
    * to have some fix reference vector over the whole screen.
    */
    vec3 tangent = normalize(rand);
    vec3 bitangent = cross(tangent, n);
    tangent = cross(bitangent, n);

    /* Make our disk orient towards the normal. */
    return tangent * p.x + bitangent * p.y;
}

/**
```

```
* Generate a uniformly distributed random point on the unit-sphere.
*
* After:
* http://mathworld.wolfram.com/SpherePointPicking.html
*/
vec3 randomSpherePoint(vec3 rand) {
    float ang1 = (rand.x + 1.0) * PI; // [-1..1) -> [0..2*PI)
    float u = rand.y; // [-1..1), cos and acos(2v-1) cancel each other out, so we arrive at [-1..1)
    float u2 = u * u;
    float sqrt1MinusU2 = sqrt(1.0 - u2);
    float x = sqrt1MinusU2 * cos(ang1);
    float y = sqrt1MinusU2 * sin(ang1);
    float z = u;
    return vec3(x, y, z);
}

/**
* Generate a uniformly distributed random point on the unit-hemisphere
* around the given normal vector.
*
* This function can be used to generate reflected rays for diffuse surfaces.
* Actually, this function can be used to sample reflected rays for ANY surface
* with an arbitrary BRDF correctly.
* This is because we always need to solve the integral over the hemisphere of
* a surface point by using numerical approximation using a sum of many
* sample directions.
* It is only with non-lambertian BRDF's that, in theory, we could sample them more
* efficiently, if we knew in which direction the BRDF reflects the most energy.
* This would be importance sampling, but care must be taken as to not over-estimate
* those surfaces, because then our sum for the integral would be greater than the
* integral itself. This is the inherent problem with importance sampling.
*
* The points are uniform over the sphere and NOT over the projected disk
```

```
* of the sphere, so this function cannot be used when sampling a spherical
* light, where we need to sample the projected surface of the light (i.e. disk)!
*/
vec3 randomHemispherePoint(vec3 rand, vec3 n) {
    /**
    * Generate random sphere point and swap vector along the normal, if it
    * points to the wrong of the two hemispheres.
    * This method provides a uniform distribution over the hemisphere,
    * provided that the sphere distribution is also uniform.
    */
    vec3 v = randomSpherePoint(rand);
    return v * sign(dot(v, n));
}

vec3 ortho(vec3 v) {
    //  See : http://lolengine.net/blog/2013/09/21/picking-orthogonal-vector-combing-coconuts
    return abs(v.x) > abs(v.z) ? vec3(-v.y, v.x, 0.0)  : vec3(0.0, -v.z, v.y);
}

/**
* Generate a cosine-weighted random point on the unit hemisphere oriented around 'n'.
*
* @param rand a vector containing pseudo-random values
* @param n    the normal to orient the hemisphere around
* @returns    the cosine-weighted point on the oriented hemisphere
*/
vec3 randomCosineWeightedHemispherePoint(vec3 rand, vec3 n) {
    float r = rand.x * 0.5 + 0.5; // [-1..1) -> [0..1)
    float angle = (rand.y + 1.0) * PI; // [-1..1] -> [0..2*PI)
    float sr = sqrt(r);
    vec2 p = vec2(sr * cos(angle), sr * sin(angle));
    /*
    * Unproject disk point up onto hemisphere:
    * 1.0 == sqrt(x*x + y*y + z*z) -> z = sqrt(1.0 - x*x - y*y)
    */
    vec3 ph = vec3(p.xy, sqrt(1.0 - p*p));
    /*
    * Compute some arbitrary tangent space for orienting
```

```
    * our hemisphere 'ph' around the normal. We use the camera's up vector
    * to have some fix reference vector over the whole screen.
    */
    vec3 tangent = normalize(rand);
    vec3 bitangent = cross(tangent, n);
    tangent = cross(bitangent, n);

    /* Make our hemisphere orient around the normal. */
    return tangent * ph.x + bitangent * ph.y + n * ph.z;
}
```

Code 12.7: utils/loadTexture.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

GLuint loadDDS(const char* imagepath);
```

Code 12.8: utils/loadTexture.cpp

```
#include "loadTexture.h"

/*
* modified from: ogl-master
* common/texture.cpp
* */
#define FOURCC_DXT1 0x31545844 // Equivalent to "DXT1" in ASCII
#define FOURCC_DXT3 0x33545844 // Equivalent to "DXT3" in ASCII
#define FOURCC_DXT5 0x35545844 // Equivalent to "DXT5" in ASCII

GLuint loadDDS(const char* imagepath) {

    unsigned char header[124];

    FILE* fp;

    /* try to open the file */
```

```
    fp = fopen(imagepath, "rb");
    if (fp == NULL) {
        printf("%s could not be opened.\n", imagepath); getchar↩
    ();
        return 0;
    }

    /* verify the type of file */
    char filecode[4];
    fread(filecode, 1, 4, fp);
    if (strncmp(filecode, "DDS ", 4) != 0) {
        fclose(fp);
        return 0;
    }

    /* get the surface desc */
    fread(&header, 124, 1, fp);

    unsigned int height = *(unsigned int*)&(header[8]);
    unsigned int width = *(unsigned int*)&(header[12]);
    unsigned int linearSize = *(unsigned int*)&(header[16]);
    unsigned int mipMapCount = *(unsigned int*)&(header[24]);
    unsigned int fourCC = *(unsigned int*)&(header[80]);


    unsigned char* buffer;
    unsigned int bufsize;
    /* how big is it going to be including all mipmaps? */
    bufsize = mipMapCount > 1 ? linearSize * 2 : linearSize;
    buffer = (unsigned char*)malloc(bufsize * sizeof(unsigned ↩
    char));
    fread(buffer, 1, bufsize, fp);
    /* close the file pointer */
    fclose(fp);

    unsigned int components = (fourCC == FOURCC_DXT1) ? 3 : 4;
    unsigned int format;
    switch (fourCC)
    {
    case FOURCC_DXT1:
        format = GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
        break;
    case FOURCC_DXT3:
        format = GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
        break;
    case FOURCC_DXT5:
```

```
        format = GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
        break;
    default:
        free(buffer);
        return 0;
    }

    // Create one OpenGL texture
    GLuint textureID;
    glGenTextures(1, &textureID);

    // "Bind" the newly created texture : all future texture ↩
    functions will modify this texture
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    unsigned int blockSize = (format == ↩
    GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) ? 8 : 16;
    unsigned int offset = 0;

    /* load the mipmaps */
    for (unsigned int level = 0; level < mipMapCount && (width ↩
    || height); ++level)
    {
        unsigned int size = ((width + 3) / 4) * ((height + 3) /↩
     4) * blockSize;
        glCompressedTexImage2D(GL_TEXTURE_2D, level, format, ↩
    width, height,
            0, size, buffer + offset);

        offset += size;
        width /= 2;
        height /= 2;

        // Deal with Non-Power-Of-Two textures. This code is ↩
    not included in the webpage to reduce clutter.
        if (width < 1) width = 1;
        if (height < 1) height = 1;

    }

    free(buffer);

    return textureID;
```

```
}
```

Code 12.9: utils/meshio.h

```
#pragma once
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <sstream>
#include <cstring>
#include <stdlib.h>


#include <glm/glm.hpp>
#include <GL/glew.h>

// Include AssImp
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // Output data structure
#include <assimp/postprocess.h>     // Post processing flags

using std::vector;
using glm::vec2;
using glm::vec3;
using glm::vec4;

struct Mesh
{
    int vn;
    int fn;
    vector<vec4> vs;
    vector<vec2> vts;
    vector<vec4> vns;
    vector<int> fs;
    vector<vec4> maxvs;
    vector<vec4>  minvs;
    vector<vec4>  ns;
};

bool loadObj(
    bool hasDDS,
    int start0,
```

```
    int end0,
    const char* path,
    Mesh& mesh
);

void LoadShader(GLuint ShaderID, const char* file_path);

GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path);
```

Code 12.10: utils/meshio.cpp

```
#include "meshio.h"

/*
* modified from: ogl-master
* common/objloader.cpp
* author: opengl-tutorial.org
*/

// anyway, we only allow one mesh and one texture here,
// and other input may cause rendering error as a result
bool loadObj(
    bool hasDDS,
    int start0,
    int end0,
    const char* path,
    Mesh& mesh
) {
    Assimp::Importer importer;

    const aiScene* scene = importer.ReadFile(
        path,
        0       /*aiProcess_JoinIdenticalVertices | ↩
    aiProcess_SortByPType*/
    );
    if (!scene) {
        fprintf(stderr, importer.GetErrorString());
        getchar();
        return false;
    }

    int start = start0 == -1 ? 0 : start0;
    int end = end0 == -1 ? scene->mNumMeshes : end0 + 1;
    // for now the models donnot overlap
```

```
    for (int j = start; j < end; j++)
    {
        int vertexnum = mesh.vs.size();
        const aiMesh* inmesh = scene->mMeshes[j];

        mesh.vn += inmesh->mNumVertices;
        mesh.fn += inmesh->mNumFaces;

        // Fill vertices positions
        mesh.vs.reserve(vertexnum + inmesh->mNumVertices);
        for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
     {
            aiVector3D pos = inmesh->mVertices[i];
            mesh.vs.push_back(vec4(pos.x, pos.y, pos.z,0));
        }

        mesh.vts.reserve(vertexnum + inmesh->mNumVertices);
        // no texture
        if (hasDDS)
        {
            // Fill vertices texture coordinates
            for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
    i++) {
                aiVector3D UVW = inmesh->mTextureCoords[j][i];
                // Assume only 1 set of UV coords; AssImp ↩
    supports 8 UV sets.
                mesh.vts.push_back(vec2(UVW.x, -UVW.y));
            }
        }
        else
        {
            for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
    i++) {
                mesh.vts.push_back(vec2(0, 0));
            }
        }

        // Fill vertices normals
        mesh.vns.reserve(vertexnum + inmesh->mNumVertices);
        for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
     {
            aiVector3D n = inmesh->mNormals[i];
            mesh.vns.push_back(vec4(n.x, n.y, n.z,0));
        }
```

```
        // Fill face indices
        mesh.fs.reserve(mesh.fs.size() + 3 * inmesh->mNumFaces)↩
    ;
        mesh.maxvs.reserve(mesh.maxvs.size() + inmesh->↩
    mNumFaces);
        mesh.minvs.reserve(mesh.minvs.size() + inmesh->↩
    mNumFaces);
        mesh.ns.reserve(mesh.ns.size() + inmesh->mNumFaces);
        for (unsigned int i = 0; i < inmesh->mNumFaces; i++) {
            // Assume the model has only triangles.
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[0] + ↩
    vertexnum);
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[1] + ↩
    vertexnum);
            mesh.fs.push_back(inmesh->mFaces[i].mIndices[2] + ↩
    vertexnum);
            vec3 v1 = mesh.vs[inmesh->mFaces[i].mIndices[0] + ↩
    vertexnum];
            vec3 v2 = mesh.vs[inmesh->mFaces[i].mIndices[1] + ↩
    vertexnum];
            vec3 v3 = mesh.vs[inmesh->mFaces[i].mIndices[2] + ↩
    vertexnum];
            mesh.maxvs.push_back(vec4(glm::max(glm::max(v1, v2)↩
    , v3),0));
            mesh.minvs.push_back(vec4(glm::min(glm::min(v1, v2)↩
    , v3),0));
            // no interpolation for edges
            vec3 n1 = mesh.vns[inmesh->mFaces[i].mIndices[0] + ↩
    vertexnum];
            vec3 n2 = mesh.vns[inmesh->mFaces[i].mIndices[1] + ↩
    vertexnum];
            vec3 n3 = mesh.vns[inmesh->mFaces[i].mIndices[2] + ↩
    vertexnum];
            mesh.ns.push_back(vec4(glm::normalize(n1 + n2 + n3)↩
    ,0));
        }

    }

    return true;
}


/*
* modified from: ogl-master
* common/shader.cpp
```

```
* author: opengl-tutorial.org
*/
void LoadShader(GLuint ShaderID, const char* file_path)
{
    // Read the Vertex Shader code from the file
    std::string ShaderCode;
    std::ifstream ShaderStream(file_path, std::ios::in);
    if (ShaderStream.is_open()) {
        std::stringstream sstr;
        sstr << ShaderStream.rdbuf();
        ShaderCode = sstr.str();
        ShaderStream.close();
    }
    else {
        printf("Impossible␣to␣open␣%s.␣\n", file_path);
        getchar();
        exit(-1);
    }

    GLint Result = GL_FALSE;
    int InfoLogLength;

    printf("Compiling␣shader␣:␣%s\n", file_path);
    char const* xSourcePointer = ShaderCode.c_str();
    glShaderSource(ShaderID, 1, &xSourcePointer, NULL);
    glCompileShader(ShaderID);

    glGetShaderiv(ShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(ShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength)↩
    ;
    if (InfoLogLength > 0) {
        std::vector<char> ShaderErrorMessage(InfoLogLength + 1)↩
    ;
        glGetShaderInfoLog(ShaderID, InfoLogLength, NULL, &↩
    ShaderErrorMessage[0]);
        printf("%s\n", &ShaderErrorMessage[0]);
    }
}



GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path) {

    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
```

```
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER↩
    );

    LoadShader(VertexShaderID, vertex_file_path);
    LoadShader(FragmentShaderID, fragment_file_path);

    GLint Result = GL_FALSE;
    int InfoLogLength;

    // Link the program
    printf("Linking␣program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);

    // Check the program
    glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &↩
    InfoLogLength);
    if (InfoLogLength > 0) {
        std::vector<char> ProgramErrorMessage(InfoLogLength + ↩
    1);
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &↩
    ProgramErrorMessage[0]);
        printf("%s\n", &ProgramErrorMessage[0]);
    }


    glDetachShader(ProgramID, VertexShaderID);
    glDetachShader(ProgramID, FragmentShaderID);

    glDeleteShader(VertexShaderID);
    glDeleteShader(FragmentShaderID);

    return ProgramID;
}
```

Part IV

Lab 3 Curves and Surfaces

Chapter 13

Requirement

13.1 Basic Requirement

The basic requirement for the experiment is shown below:

1. Draw *Bezier curve*.
2. Draw surface of revolution.
3. Draw swept surfaces.

13.2 Extra

There are also some extra functions the code implements:

1. Camera postion and direction can be set with user interaction.
2. Rotate and translate object with keyboard.
3. Show coordinate system at cur point.
4. Draw surface of revolution and swept surfaces with control points, then interporate them with Bezier curves.
5. Output generated model as obj file.
6. Implemention of *Phong reflection model* in *OenGL4* *Shader*.

Chapter 14

Theory and Application

14.1 Bezier Curve

We use cubic *Bezier curve* for interportaion. The end control points will be the begin of another Bezier curve. If the remain points are less than 3, we use lower order *Bezier curve*.

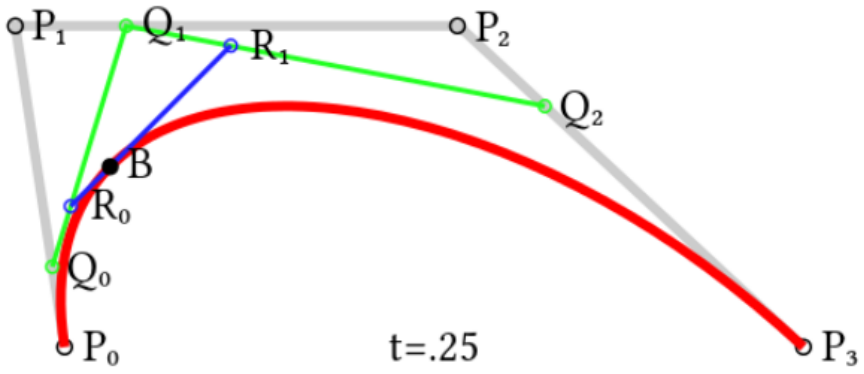

Figure 14.1: Bezier Curve

Firstly, for closed curves, we change the order to make them in a counterclockwise sequence. For open curves, the data must be counterclockwise sequences, we cannot decide the order.

After that, we choose the direction of curves as X axis in local coordinate for each point, manually pick a vector as start point's Y axis. Crossing above vectors forms X axis. Then for next point, last Y axis is used to calculate Z axis, then its Z axis is calculated by crossing new Z and X again.

14.2 Surface of Revolution

We assume the profile curve is always on the left of the y axis, the Xs of start point and end point are the same as the rotation axis and the sequence of points must be counterclockwise.

We rotate the curve each time by a small angle counterclockwise, then link sampled points together clockwise to form meshes.

14.3 Swept Surface

For points placed counterclockwise in sweep curve, we consturct local coordinates. Then for points placed counterclockwise in profile curve, we transform them from local coordinate to model coordinate.

Then we move profile curve by each sampled point on sweep curve, link sampled points on profile curve with former ones to form meshes.

Chapter 15

Program

15.1 Compiling Guide

First restore packets with *NuGet*, then compile the project with Visual Studio 2019.

15.1.1 Environment

OpenGL4, Windows 10, Visual Studio 2019 *(v142)*.

15.1.2 Packages Used

1. *assimp*
2. *glfw*
3. *glm*
4. *glew*

15.2 User Guide

The program can only be interacted with keyboard.

1. Press *1* to toggle edit mode. (In edit mode, model can be moved and rotated. In view mode, camera postion and direction can be changed.)
2. Long press *asdwqe* for changing position.
3. Long press *fghrty* for changing direction.

4. Press *2,4,5* to toggle displayed curve for surface of revolution profile, sweep curve, sweep profile respectly.

5. Press *z* to toggle control point mode.(In control point mode, control points ane linked with straight lines. In Bezier mode, interporated points with axes are shown.)

6. press *3* to show surface of revolution.

7. press *6* to show swept surface.

8. Press *esc* to exit.

9. After exiting, the command line interface will automatically generate *.obj* file for models.

15.3 Results

The control points data are stored in program memory. For rendered axes, green is X, red is Z and blue is Y.

15.3.1 Curves

The Bezier curve for surface of revolution profile:

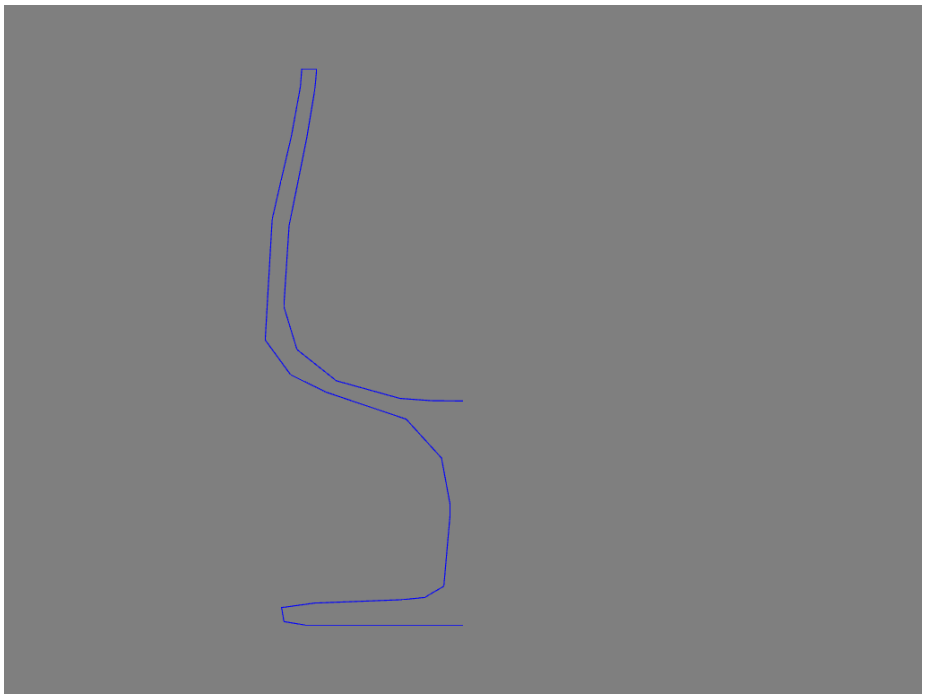

Figure 15.1: control points of surface of revolution profile

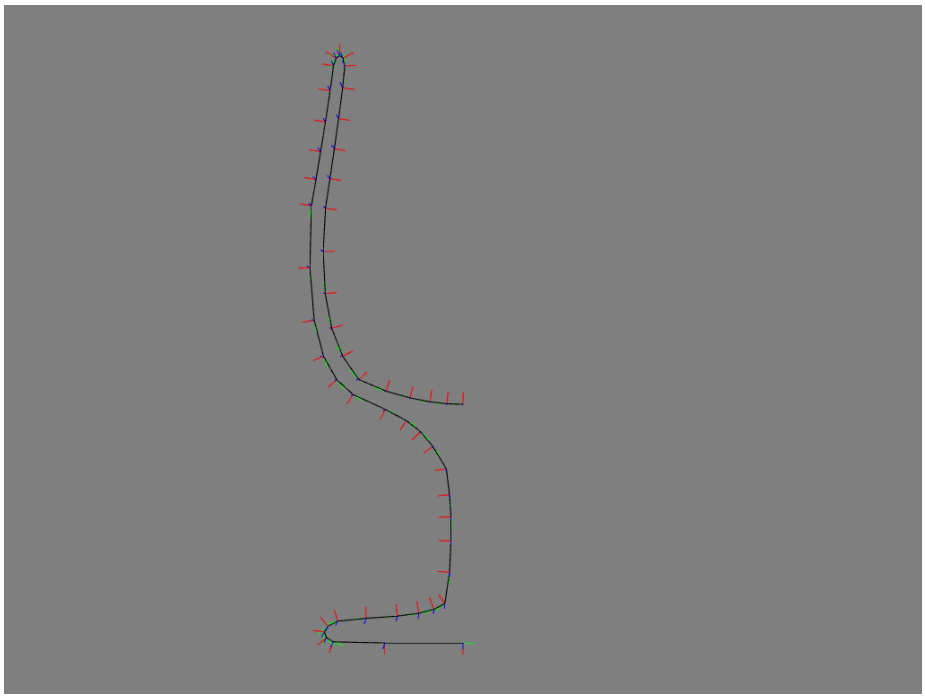

Figure 15.2: Bezier curve of surface of revolution profile

The Bezier curve for sweep curve:

Figure 15.3: control points of sweep curve

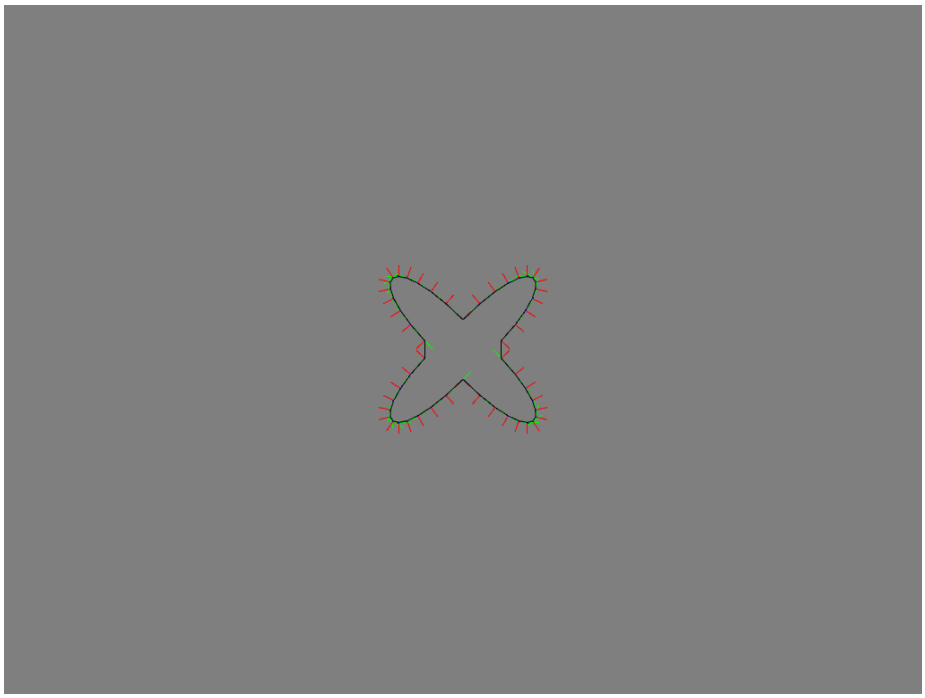

Figure 15.4: Bezier curve of sweeping

The Bezier curve for sweep profile:

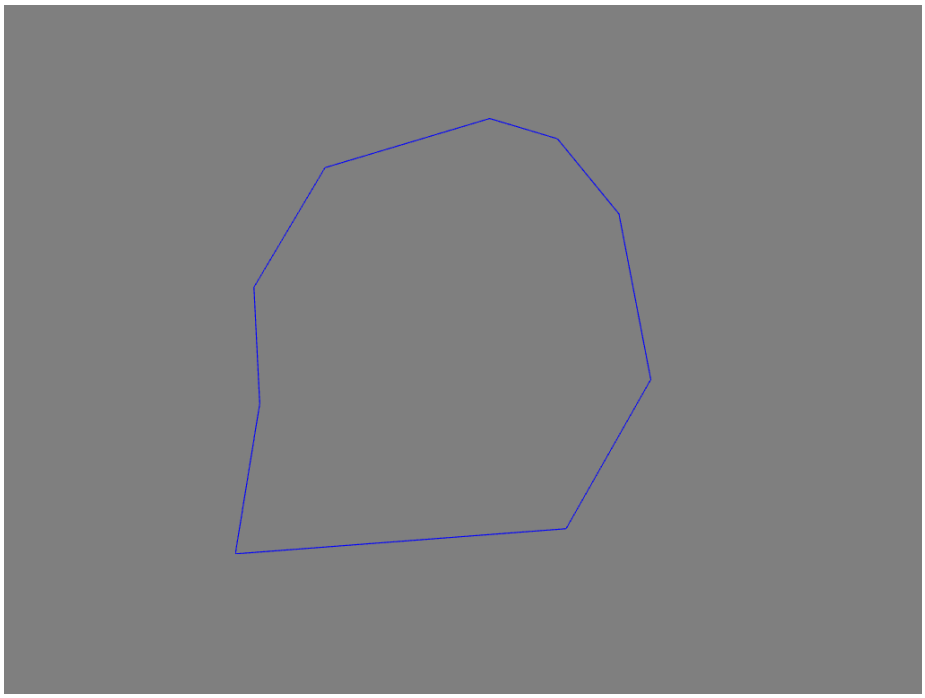

Figure 15.5: control points of sweep profile

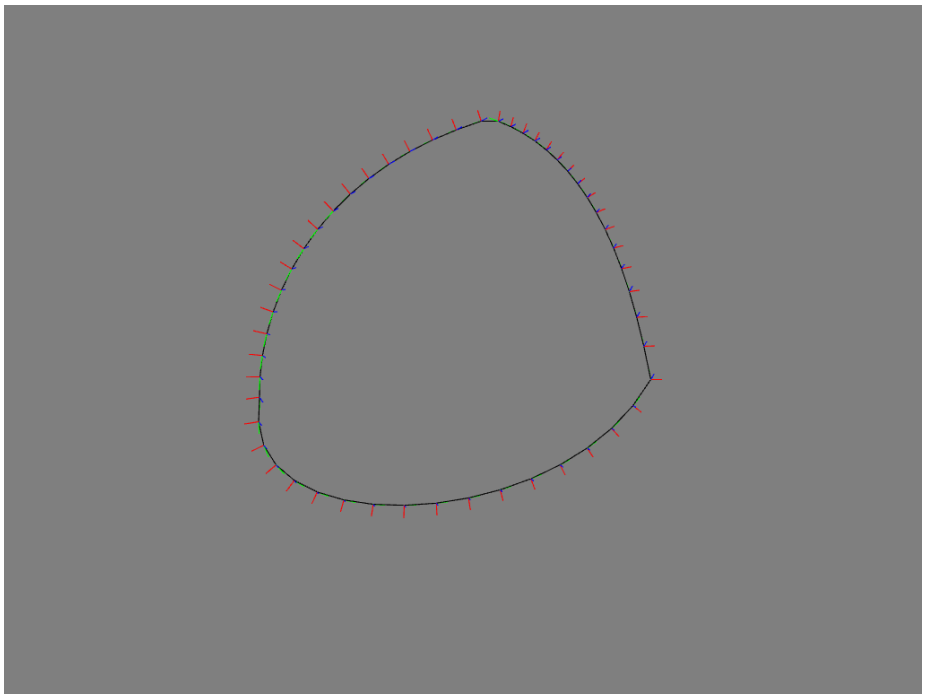

Figure 15.6: Bezier curve of sweep profile

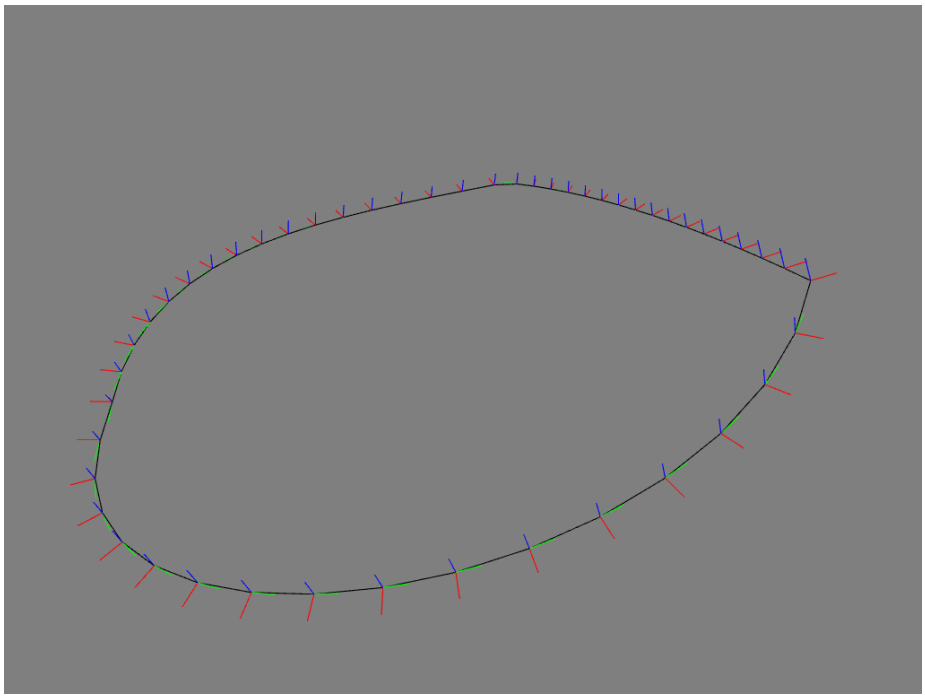

Figure 15.7: Bezier curve of sweep profile after camera transformations

15.3.2 Surfaces

Below is the surface of revolution for a cup.

Figure 15.8: surface of revolution

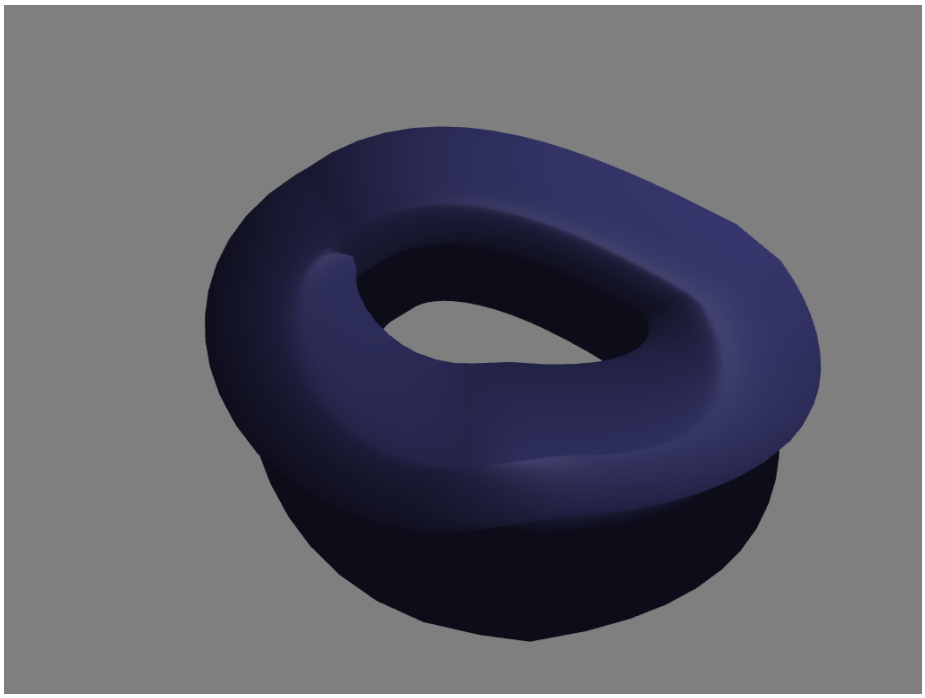

Figure 15.9: swept surface

Chapter 16

Source Code

16.1 Structure

1. *main.cpp* : main code
2. *object.fs.glsl* : object vertex shader file
3. *object.vs.glsl* : object fragment shader file
4. *curve.fs.glsl* : curve vertex shader file
5. *curve.vs.glsl* : curve fragment shader file
6. *utils*
   (a) *loadTexture.h* : functions for loading *.dds* file
   (b) *meshio.h* : functions for loading *.obj* file and shader file
   (c) *mathTool.h* : functions for calculating *Bezier curve* and other math operations
7. *res/* :store model files

16.2 Code files

Code 16.1: main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
```

```
#include <iterator>

#include <GL/glew.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtx/quaternion.hpp>

#include <GLFW/glfw3.h>

#include "utils/loadTexture.h"
#include "utils/meshio.h"
#include "utils/mathTool.h"

#pragma comment( lib, "OpenGL32.lib" )
using std::string;
using glm::quat;
using glm::vec3;
using glm::mat4;
using glm::mat3;

int wwidth = 1024, wheight = 768;
GLFWwindow* window;

// object view in program
GLuint program0;
GLuint MatrixID;
GLuint ViewMatrixID;
GLuint ModelMatrixID;
GLuint vertexbuffer;
GLuint normalbuffer;

// curve view in program
GLuint program1;
GLuint MVPID;
GLuint ColorID;
GLuint curvevertexbuffer;

GLuint vao;

const float viewMoveSpeed = 0.04f;
const float editMoveSpeed = 0.04f;
const float viewRotateSpeed = 0.01f;
const float editRotateSpeed = 0.025f;
```

```
float vHorizontalAngle = -glm::pi<float>();
float vVerticalAngle = 0;
glm::vec3 vPosition = glm::vec3(0, 0, 5);
glm::vec3 vDirection;
glm::vec3 vRight;
glm::vec3 vUp;

// view matrix
const mat4 ProjectionMatrix = glm::perspective(
    glm::radians(60.0f), (float)wwidth / wheight, 1.0f, 100.0f)↩
    ;
glm::mat4 MVP;
glm::mat4 ViewMatrix;
quat ModelSORRo = quat(1, 0, 0, 0);
glm::mat4 ModelMatrixSORTr = glm::mat4(1.0);
quat ModelSSRo = quat(1, 0, 0, 0);
glm::mat4 ModelMatrixSSTr = glm::mat4(1.0);
glm::mat4 ModelTransMatrix = glm::mat4(1.0);
quat ModelRoQuat = quat(1, 0, 0, 0);
mat4 ModelMatrix = glm::mat4(1.0);

// user control
bool viewMode = true;
bool PointCurveMode = true;
bool show2 = true;
bool show4 = false;
bool show5 = false;
// surface of revolution
bool SORMode = false;
// sweep surface
bool SSMode = false;

vector<float> SORProfile = {
    0.000000 ,-0.459543,
    -0.211882, -0.456747,
    -0.421882, -0.436747,
    -0.848656, -0.278898,

    -1.112097, 0,
    -1.2, 0.384005,
    -1.164785, 1.105511,

    -1.104785, 1.5,
    -1.044785, 1.9,
    -0.991667, 2.328629,
```

```
    -0.98, 2.503360,
    -1.08, 2.503360,
    -1.088800, 2.345600,

    -1.15, 1.9,
    -1.22, 1.5,
    -1.278000, 1.162800,

    -1.324800, 0.085200,
    -1.154800, -0.225200,
    -0.915600 ,-0.381200,

    -0.380400 ,-0.622000,
    -0.380400 ,-0.622000,
    -0.144000, -0.968400,

    -0.086800, -1.380000,
    -0.086800, -1.480000,
    -0.128400, -2.112400,

    -0.257200, -2.212800,
    -0.407200, -2.232800,
    -0.994400, -2.262800,

    -1.214800, -2.303200,
    -1.199200, -2.428400,
    -1.057600, -2.458800,

    0.000000, -2.458802
};

vector<float> SSProfile =
{
    0.000000, -0.250000,
    0.625,-0.875,
    0.875,-0.625,
    0.250000 ,0.000000,

    0.875,0.625,
    0.625,0.875,
    0.000000, 0.250000,

    -0.625,0.875,
    -0.875,0.625,
    -0.250000, 0.000000,
```

```
    -0.875,-0.625,
    -0.625,-0.875
};

vector<float> SSCurve = {
1.60, -0.255455, -0.080661,
1.582727, 1.374545, -1.050661,
0.912727, 2.034545, -0.760661,
0.242727, 2.094545, -0.410661,

-1.097273, 1.444545, 0.259339,
-1.427273, 0.424545, 0.929339,
-1.5, -0.4, 0.6,

-1.767273, -1.585455, 0.371157,
0.912727, -1.585455, -0.278843,
};

const bool close2 = false;
const bool close4 = false;
const bool close5 = false;

// for vexter order, thus in counterclockwise order from below ↩
    view point
// there exist many circumstances for determing the viewing ↩
    direction
const vec3 zview = vec3(0, 0, 1);
const vec3 yview = vec3(0, 1, 0);
// profile curve
const int ptInterval = 50;
// surface of revolution
const int angleInterval = 50;
// swept surfaces
const int stInterval = 50;

vector<vec3> curveVertex;
vector<vec3> curveTan;
vector<vec3> curveNormal;
vector<vec3> curveB;
vector<vec3> faceVertex;
vector<vec3> faceNormal;


vector<vector<vec3>> curveVertexs;
vector<vector<vec3>> curveTans;
```

```
vector<vector<vec3>> curveNormals;
vector<vector<vec3>> curveBs;
vector<vector<vec3>> faceVertexs;
vector<vector<vec3>> faceNormals;


bool escPress = false;
// transitions
bool wPress = false;
bool aPress = false;
bool sPress = false;
bool dPress = false;
bool qPress = false;
bool ePress = false;
bool tPress = false;
bool fPress = false;
bool gPress = false;
bool hPress = false;
bool rPress = false;
bool yPress = false;

void resetCamera()
{
    vHorizontalAngle = -glm::pi<float>();
    vVerticalAngle = 0;
    vPosition = glm::vec3(0, 0, 5);
}

void resetModel()
{
    if (SORMode)
    {
        ModelTransMatrix = ModelMatrixSORTr;
        ModelRoQuat = ModelSORRo;
    }
    else if (SSMode)
    {
        ModelTransMatrix = ModelMatrixSSTr;
        ModelRoQuat = ModelSSRo;
    }
    else
    {
        ModelTransMatrix = glm::mat4(1.0);
        ModelRoQuat = quat(1, 0, 0, 0);
    }
    ModelMatrix = ModelTransMatrix * glm::toMat4(ModelRoQuat);
```

```
}

void init()
{
    if (!glfwInit())
    {
        fprintf(stderr, "Failed␣to␣initialize␣GLFW\n");
        getchar();
        exit(-1);
    }

    glfwDefaultWindowHints();
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, ↩
    GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, true);
    glfwWindowHint(GLFW_VISIBLE, false);
    glfwWindowHint(GLFW_RESIZABLE, false);

    window = glfwCreateWindow(wwidth, wheight, "exp03", NULL, ↩
    NULL);
    if (window == NULL) {
        fprintf(stderr, "Failed␣to␣open␣GLFW␣window.\n");
        getchar();
        glfwTerminate();
        exit(-1);
    }

    printf("press␣'1'␣to␣toggle␣transformation␣mode␣for␣object\↩
    n"
        "press␣'wasdqe'␣to␣move␣object␣or␣camera␣depending␣on␣↩
    corrent␣mode\n"
        "press␣'fghtry'␣to␣rotate␣object␣or␣camera␣depending␣on↩
    ␣corrent␣mode\n"
        "press␣'2,4,5'␣to␣enter␣curve␣dislaying␣mode\n"
        "press␣'z'␣to␣toggle␣control␣point␣mode\n"
        "press␣'3'␣to␣show␣surface␣of␣revolution␣based␣on␣what␣↩
    displayed␣in␣'2'\n"
        "press␣'6'␣to␣show␣swept␣surface␣based␣on␣what␣↩
    displayed␣in␣'4'␣as␣profile␣curve␣and␣in␣'5'␣as␣sweep␣curve↩
    \n"
        "press␣'esc'␣to␣ternimate␣program.\n"
    );
```

```
glfwSetKeyCallback(window, [](GLFWwindow* window, int key, int scancode, int action, int mode) {
    if (action == GLFW_PRESS)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            escPress = true;
        }
        else if (key == GLFW_KEY_1)
        {
            if (viewMode && !show2 && !show4 && !show5)
            {
                resetModel();
                printf("entering transformation mode\n");
                viewMode = !viewMode;
            }
            // edit model mode
            else if (!viewMode && !show2 && !show4 && !show5)
            {
                if (SORMode)
                {
                    ModelMatrixSORTr = ModelTransMatrix;
                    ModelSORRo = ModelRoQuat;
                }
                else if (SSMode)
                {
                    ModelMatrixSSTr = ModelTransMatrix;
                    ModelSSRo = ModelRoQuat;
                };
                printf("exiting transformation mode\n");
                viewMode = !viewMode;
            }

        }
        else if (viewMode && key == GLFW_KEY_2)
        {
            PointCurveMode = true;
            SORMode = false;
            SSMode = false;
            show2 = true;
            show4 = false;
            show5 = false;
            resetCamera();
            resetModel();
```

```
            printf("displaying␣profile␣curve␣for␣surfaces␣↩
of␣revolution\n");
            printf("entering␣control␣point␣mode\n");
        }
        else if (!SORMode && viewMode && key == GLFW_KEY_3)
        {
            PointCurveMode = false;
            SORMode = true;
            SSMode = false;
            show2 = false;
            show4 = false;
            show5 = false;
            resetCamera();
            resetModel();
            printf("entering␣surfaces␣of␣revolution␣mode\n"↩
);

        }
        else if (viewMode && key == GLFW_KEY_4)
        {
            PointCurveMode = true;
            SORMode = false;
            SSMode = false;
            show2 = false;
            show4 = true;
            show5 = false;
            resetCamera();
            resetModel();
            printf("displaying␣profile␣curve␣for␣swept␣↩
surfaces\n");
            printf("entering␣control␣point␣mode\n");
        }
        else if (viewMode && key == GLFW_KEY_5)
        {
            PointCurveMode = true;
            SORMode = false;
            SSMode = false;
            show2 = false;
            show4 = false;
            show5 = true;
            resetCamera();
            resetModel();
            printf("displaying␣sweep␣curve␣for␣swept␣↩
surfaces\n");
            printf("entering␣control␣point␣mode\n");
        }
```

```
        else if (viewMode && key == GLFW_KEY_6)
        {

            PointCurveMode = false;
            SORMode = false;
            SSMode = true;
            show2 = false;
            show4 = false;
            show5 = false;
            resetCamera();
            resetModel();
            printf("entering␣swept␣surfaces␣mode\n");
        }
        else if ((show2 || show4 || show5) && key == ↩
GLFW_KEY_Z)
        {
            if (!PointCurveMode)
            {
                printf("entering␣control␣point␣mode\n");
            }
            else
            {
                printf("exiting␣control␣point␣mode\n");
            }
            PointCurveMode = !PointCurveMode;
        }
        if (key == GLFW_KEY_A)
        {
            aPress = true;
        }
        else if (key == GLFW_KEY_S)
        {
            sPress = true;
        }
        else if (key == GLFW_KEY_D)
        {
            dPress = true;
        }
        else if (key == GLFW_KEY_W)
        {
            wPress = true;
        }
        else if (key == GLFW_KEY_Q)
        {
            qPress = true;
        }
```

```
            else if (key == GLFW_KEY_E)
            {
                ePress = true;
            }
            else if (key == GLFW_KEY_F)
            {
                fPress = true;
            }
            else if (key == GLFW_KEY_G)
            {
                gPress = true;
            }
            else if (key == GLFW_KEY_H)
            {
                hPress = true;
            }
            else if (key == GLFW_KEY_T)
            {
                tPress = true;
            }
            else if (key == GLFW_KEY_R)
            {
                rPress = true;
            }
            else if (key == GLFW_KEY_Y)
            {
                yPress = true;
            }
        }
        else if (action == GLFW_RELEASE)
        {
            if (key == GLFW_KEY_A)
            {
                aPress = false;
            }
            else if (key == GLFW_KEY_S)
            {
                sPress = false;
            }
            else if (key == GLFW_KEY_D)
            {
                dPress = false;
            }
            else if (key == GLFW_KEY_W)
            {
                wPress = false;
```

```
        }
        else if (key == GLFW_KEY_Q)
        {
            qPress = false;
        }
        else if (key == GLFW_KEY_E)
        {
            ePress = false;
        }
        else if (key == GLFW_KEY_F)
        {
            fPress = false;
        }
        else if (key == GLFW_KEY_G)
        {
            gPress = false;
        }
        else if (key == GLFW_KEY_H)
        {
            hPress = false;
        }
        else if (key == GLFW_KEY_T)
        {
            tPress = false;
        }
        else if (key == GLFW_KEY_R)
        {
            rPress = false;
        }
        else if (key == GLFW_KEY_Y)
        {
            yPress = false;
        }
    }
    });

const GLFWvidmode* vidmode = glfwGetVideoMode(↩
glfwGetPrimaryMonitor());
glfwSetWindowPos(
    window,
    (vidmode->width - wwidth) / 2,
    (vidmode->height - wheight) / 2
);
glfwMakeContextCurrent(window);
glfwSwapInterval(1);
```

```
    // Initialize GLEW
    glewExperimental = true;
    if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Failed␣to␣initialize␣GLEW\n");
        getchar();
        glfwTerminate();
        exit(-1);
    };

    // init shaders
    program0 = LoadShaders("object.vs.glsl", "object.fs.glsl");
    MatrixID = glGetUniformLocation(program0, "MVP");
    ViewMatrixID = glGetUniformLocation(program0, "V");
    ModelMatrixID = glGetUniformLocation(program0, "M");

    program1 = LoadShaders("curve.vs.glsl", "curve.fs.glsl");
    MVPID = glGetUniformLocation(program1, "MVP");
    ColorID = glGetUniformLocation(program1, "lcolor");

    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vertexbuffer);
    glGenBuffers(1, &normalbuffer);
    glGenBuffers(1, &curvevertexbuffer);


    glfwShowWindow(window);
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glEnable(GL_CULL_FACE);
}

void viewModel()
{
    vDirection = glm::vec3(
        cos(vVerticalAngle) * sin(vHorizontalAngle),
        sin(vVerticalAngle),
        cos(vVerticalAngle) * cos(vHorizontalAngle)
    );

    vRight = vec3(
        sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
        0,
        cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
    );
```

```
vUp = glm::cross(vRight, vDirection);

if (!viewMode)
{
    vec3 mTranslate = glm::vec3(0, 0, 0);
    float modelDirectionAngle = 0.0f;
    float modelRightAngle = 0.0f;
    float modelUpAngle = 0.0f;

    if (aPress == true)
    {
        mTranslate -= vRight * editMoveSpeed;
    }
    else if (sPress == true)
    {
        mTranslate -= vDirection * editMoveSpeed;
    }
    else if (dPress == true)
    {
        mTranslate += vRight * editMoveSpeed;
    }
    else if (wPress == true)
    {
        mTranslate += vDirection * editMoveSpeed;
    }
    else if (qPress == true)
    {
        mTranslate += vUp * editMoveSpeed;
    }
    else if (ePress == true)
    {
        mTranslate -= vUp * editMoveSpeed;
    }
    else if (fPress == true)
    {
        modelUpAngle -= viewRotateSpeed;
    }
    else if (gPress == true)
    {
        modelRightAngle += viewRotateSpeed;
    }
    else if (hPress == true)
    {
        modelUpAngle += viewRotateSpeed;
    }
```

```
        else if (tPress == true)
        {
            modelRightAngle -= viewRotateSpeed;
        }
        else if (rPress == true)
        {
            modelDirectionAngle -= viewRotateSpeed;

        }
        else if (yPress == true)
        {
            modelDirectionAngle += viewRotateSpeed;
        }

        ModelTransMatrix = glm::translate(ModelTransMatrix, ↩
    mTranslate);

        glm::quat ModelDirectionQuaternion = glm::quat(
            cos(modelDirectionAngle / 2),
            vDirection.x * sin(modelDirectionAngle / 2),
            vDirection.y * sin(modelDirectionAngle / 2),
            vDirection.z * sin(modelDirectionAngle / 2)
        );

        glm::quat ModelUpQuaternion = glm::quat(
            cos(modelUpAngle / 2),
            vUp.x * sin(modelUpAngle / 2),
            vUp.y * sin(modelUpAngle / 2),
            vUp.z * sin(modelUpAngle / 2)
        );

        glm::quat ModelRightQuaternion = glm::quat(
            cos(modelRightAngle / 2),
            vRight.x * sin(modelRightAngle / 2),
            vRight.y * sin(modelRightAngle / 2),
            vRight.z * sin(modelRightAngle / 2)
        );

        ModelRoQuat =
            ModelDirectionQuaternion *
            ModelUpQuaternion *
            ModelRightQuaternion *
            ModelRoQuat;
        modelRightAngle = modelUpAngle = modelDirectionAngle = ↩
    0;
```

```
        ModelMatrix = ModelTransMatrix * glm::toMat4(↩
    ModelRoQuat);
    }
    else
    {
        if (aPress == true)
        {
            vPosition -= vRight * viewMoveSpeed;
        }
        else if (sPress == true)
        {
            vPosition -= vDirection * viewMoveSpeed;
        }
        else if (dPress == true)
        {
            vPosition += vRight * viewMoveSpeed;
        }
        else if (wPress == true)
        {
            vPosition += vDirection * viewMoveSpeed;
        }
        else if (qPress == true)
        {
            vPosition += vUp * viewMoveSpeed;
        }
        else if (ePress == true)
        {
            vPosition -= vUp * viewMoveSpeed;
        }
        else if (fPress == true)
        {
            vHorizontalAngle += viewRotateSpeed;
        }
        else if (gPress == true)
        {
            vVerticalAngle -= viewRotateSpeed;
        }
        else if (hPress == true)
        {
            vHorizontalAngle -= viewRotateSpeed;
        }
        else if (tPress == true)
        {
            vVerticalAngle += viewRotateSpeed;
        }
```

```
        vDirection = vec3(
            cos(vVerticalAngle) * sin(vHorizontalAngle),
            sin(vVerticalAngle),
            cos(vVerticalAngle) * cos(vHorizontalAngle)
        );

        vRight = vec3(
            sin(vHorizontalAngle - glm::pi<float>() / 2.0f),
            0,
            cos(vHorizontalAngle - glm::pi<float>() / 2.0f)
        );

        vUp = glm::cross(vRight, vDirection);
    }

    ViewMatrix = glm::lookAt(
        vPosition,
        vPosition + vDirection,
        vUp
    );


    MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;
}


void loop()
{
    while (escPress != true && glfwWindowShouldClose(window) ==↩
     0)
    {
        glViewport(0, 0, wwidth, wheight);
        glfwPollEvents();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        viewModel();

        if (SORMode || SSMode)
        {
            if (SORMode)
            {
                faceVertex = faceVertexs[0];
                faceNormal = faceNormals[0];
            }
            else
            {
```

```
            faceVertex = faceVertexs[1];
            faceNormal = faceNormals[1];
        }

        glUseProgram(program0);

        glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP↩
[0][0]);
        glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &↩
ModelMatrix[0][0]);
        glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &↩
ViewMatrix[0][0]);

        glBindVertexArray(vao);
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glBufferData(GL_ARRAY_BUFFER,
            faceVertex.size() * sizeof(glm::vec3), &↩
faceVertex[0], GL_STATIC_DRAW);
        glVertexAttribPointer(
            0,                  // attribute
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
        );

        glEnableVertexAttribArray(1);
        glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
        glBufferData(GL_ARRAY_BUFFER,
            faceNormal.size() * sizeof(glm::vec3), &↩
faceNormal[0], GL_STATIC_DRAW);
        glVertexAttribPointer(
            1,                                  // attribute
            3,                                  // size
            GL_FLOAT,                           // type
            GL_FALSE,                           // normalized↩
?
            0,                                  // stride
            (void*)0                            // array ↩
buffer offset
        );
        glDrawArrays(GL_TRIANGLES, 0, faceVertex.size());

        glDisableVertexAttribArray(0);
```

```
            glDisableVertexAttribArray(1);
            glUseProgram(0);
        }
        else if (PointCurveMode)
        {
            if (show2)
            {
                curveVertex = curveVertexs[0];
            }
            else if (show4)
            {
                curveVertex = curveVertexs[2];
            }
            else if (show5)
            {
                curveVertex = curveVertexs[4];
            }

            glUseProgram(program1);

            glUniformMatrix4fv(MVPID, 1, GL_FALSE, &MVP[0][0]);
            // blue
            glUniform3f(ColorID, 0.0f, 0.0f, 1.0f);
            glBindVertexArray(vao);
            glBindBuffer(GL_ARRAY_BUFFER, curvevertexbuffer);
            glBufferData(GL_ARRAY_BUFFER,
                curveVertex.size() * sizeof(glm::vec3), &↩
    curveVertex[0], GL_STATIC_DRAW);
            glEnableVertexAttribArray(2);
            glVertexAttribPointer(
                2,                  // attribute
                3,                  // size
                GL_FLOAT,           // type
                GL_FALSE,           // normalized?
                0,                  // stride
                (void*)0            // array buffer offset
            );
            glLineWidth(3.3f);
            glDrawArrays(GL_LINE_STRIP, 0, curveVertex.size());
            glDisableVertexAttribArray(2);
            glUseProgram(0);
        }
        else if (!PointCurveMode)
        {
            if (show2)
            {
```

```
            curveVertex = curveVertexs[1];
            curveNormal = curveNormals[1];
            curveTan = curveTans[1];
            curveB = curveBs[1];
        }
        else if (show4)
        {
            curveVertex = curveVertexs[3];
            curveNormal = curveNormals[3];
            curveTan = curveTans[3];
            curveB = curveBs[3];
        }
        else if (show5)
        {
            curveVertex = curveVertexs[5];
            curveNormal = curveNormals[5];
            curveTan = curveTans[5];
            curveB = curveBs[5];
        }
        glUseProgram(program1);

        glUniformMatrix4fv(MVPID, 1, GL_FALSE, &MVP[0][0]);
        // black
        glUniform3f(ColorID, 0.0f, 0.0f, 0.0f);
        glBindVertexArray(vao);
        glBindBuffer(GL_ARRAY_BUFFER, curvevertexbuffer);
        glBufferData(GL_ARRAY_BUFFER,
            curveVertex.size() * sizeof(glm::vec3), &↩
curveVertex[0], GL_STATIC_DRAW);
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(
            2,                  // attribute
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
        );
        glLineWidth(3.3f);
        glDrawArrays(GL_LINE_STRIP, 0, curveVertex.size());

        // red
        glUniform3f(ColorID, 1.0f, 0.0f, 0.0f);
        glBufferData(GL_ARRAY_BUFFER,
            curveNormal.size() * sizeof(glm::vec3), &↩
curveNormal[0], GL_STATIC_DRAW);
```

```
        glVertexAttribPointer(
            2,                  // attribute
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
        );
        glLineWidth(3.3f);
        glDrawArrays(GL_LINES, 0, curveNormal.size());

        // green
        glUniform3f(ColorID, 0.0f, 1.0f, 0.0f);
        glBufferData(GL_ARRAY_BUFFER,
            curveTan.size() * sizeof(glm::vec3), &curveTan↩
[0], GL_STATIC_DRAW);
        glVertexAttribPointer(
            2,                  // attribute
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
        );
        glLineWidth(3.3f);
        glDrawArrays(GL_LINES, 0, curveTan.size());

        // blue
        glUniform3f(ColorID, 0.0f, 0.0f, 1.0f);
        glBufferData(GL_ARRAY_BUFFER,
            curveB.size() * sizeof(glm::vec3), &curveB[0], ↩
GL_STATIC_DRAW);
        glVertexAttribPointer(
            2,                  // attribute
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
        );
        glLineWidth(3.3f);
        glDrawArrays(GL_LINES, 0, curveB.size());

        glDisableVertexAttribArray(2);
        glUseProgram(0);
    }
```

```
        glfwSwapBuffers(window);
    }

    glfwTerminate();
}

void calculateBezier(vector<float> points, vec3 up,int Interval↩
    , bool closed, bool sweep = false)
{
    viewModel();

    float mtx[16] = {
        1.0f / 3.0f,0,0,0,
        0,1.0f / 3.0f,0,0,
        0,0,1,0,
        0,0,0,1
    };
    mat4 invViewProjMatrix = glm::inverse(MVP) * glm::transpose↩
    (glm::make_mat4(mtx));
    vector<vec3> cp0, cp;
    for (int i = 0; i < points.size(); )
    {
        if (sweep)
        {
            cp.push_back(vec3(points[i], points[i + 1], points[↩
    i + 2]));
            cp0.push_back(vec3(points[i], points[i + 1], points↩
    [i + 2]));
            i += 3;
        }
        else
        {
            cp.push_back(invViewProjMatrix * vec4(points[i], ↩
    points[i + 1], -1.0 + 0.01, 1));
            cp0.push_back(vec3(points[i], points[i + 1], 0));
            i += 2;
        }
    }
    if (closed && !isCounterClockWise(cp, up))
    {
        std::reverse(cp.begin(), cp.end());
        std::reverse(cp0.begin(), cp0.end());
    }
```

```
    if (closed)
    {
        cp.push_back(cp[0]);
    }

    curveVertexs.push_back(cp);

    // Calculate Bezier
    vector<vec3> o, o1, o2, o3;
    o.reserve(Interval);
    o1.reserve(Interval);
    o2.reserve(Interval);
    o3.reserve(Interval);
    vec3 lastB = up;
    if (sweep)
    {
        for (int i = 0; i < Interval; i++)
        {
            o.push_back(cubicBeziers(cp0, (float)i / Interval, ↩
    closed));
            vec3 tmp = glm::normalize(getTan(cp0, (float)i / ↩
    Interval, closed));
            o1.push_back(tmp);
            vec3 N = glm::normalize(glm::cross(tmp, lastB));
            o2.push_back(N);
            lastB = glm::normalize(glm::cross(N, tmp));
            o3.push_back(lastB);
        }
        o.push_back(o[0]);
        o1.push_back(o1[0]);
        o2.push_back(o2[0]);
        o3.push_back(o3[0]);
    }
    else
    {
        for (int i = 0; i <= Interval; i++)
        {
            o.push_back(cubicBeziers(cp0, (float)i / Interval, ↩
    closed));
            vec3 tmp = glm::normalize(getTan(cp0, (float)i / ↩
    Interval, closed));
            o1.push_back(tmp);
            vec3 N = glm::normalize(glm::cross(tmp, lastB));
            o2.push_back(N);
            lastB = glm::normalize(glm::cross(N, tmp));
            o3.push_back(lastB);
```

```
        }
    }

    curveVertexs.push_back(o);
    curveTans.push_back(o1);
    curveNormals.push_back(o2);
    curveBs.push_back(o3);
}

void generateCurveOut(int idx, int interval)
{
    const float ratio = 0.1;
    vector<vec3> o1, o2, o3;
    o1.reserve(interval * 2);
    o2.reserve(interval * 2);
    o3.reserve(interval * 2);
    for (int i = 0; i <= interval; i++)
    {
        o1.push_back(curveVertexs[idx][i]);
        o1.push_back(curveVertexs[idx][i] + ratio * curveTans[↩
    idx - 1][i]);
        o2.push_back(curveVertexs[idx][i]);
        o2.push_back(curveVertexs[idx][i] + ratio * ↩
    curveNormals[idx - 1][i]);
        o3.push_back(curveVertexs[idx][i]);
        o3.push_back(curveVertexs[idx][i] + ratio * curveBs[idx↩
     - 1][i]);
    }
    curveTans.push_back(o1);
    curveNormals.push_back(o2);
    curveBs.push_back(o3);
}

void generateSOR(int idx)
{
    vector<vec3> o;
    vector<vec3> o1;
    vector<vec3> lastline = curveVertexs[idx];
    vector<vec3> lastNormalLine = curveNormals[idx - 1];
    float r = -glm::pi<float>() * 2.0f / angleInterval;
    mat4 roM = glm::toMat4(quat(
        cos(r / 2),
        0,
        sin(r / 2),
        0
    ));
```

```
    for (int i = 1; i <= angleInterval; i++)
    {
        vector<vec3> thisline;
        vector<vec3> thisNormalLine;
        vec3 lastPoint = roM * vec4(lastline[0], 1.0);
        vec3 lastNormal = roM * vec4(lastNormalLine[0], 0);
        thisline.push_back(lastPoint);
        thisNormalLine.push_back(lastNormal);
        for (int j = 1; j <= ptInterval; j++)
        {
            vec3 thisPoint = roM * vec4(lastline[j], 1.0);
            vec3 thisNormal = roM * vec4(lastNormalLine[j], 0);
            thisline.push_back(thisPoint);
            thisNormalLine.push_back(thisNormal);
            o.push_back(lastPoint);
            o.push_back(thisPoint);
            o.push_back(lastline[j]);

            o.push_back(lastPoint);
            o.push_back(lastline[j]);
            o.push_back(lastline[j - 1]);

            o1.push_back(lastNormal);
            o1.push_back(thisNormal);
            o1.push_back(lastNormalLine[j]);

            o1.push_back(lastNormal);
            o1.push_back(lastNormalLine[j]);
            o1.push_back(lastNormalLine[j - 1]);
            lastPoint = thisPoint;
            lastNormal = thisNormal;
        }
        lastline = thisline;
        lastNormalLine = thisNormalLine;
    }
    faceVertexs.push_back(o);
    faceNormals.push_back(o1);
}

void generateSS(int idx1, int idx2)
{
    vector<vec3> o;
    vector<vec3> o1;
    vector<vec3> sweepCurve = curveVertexs[idx2];
    float resize[16] =
    {
```

```
    1.0f,0,0,0,
    0,1.0f ,0,0,
    0,0,1.0f,0,
    0,0,0,1
};
mat4 resizeM = glm::transpose( glm::make_mat4(resize));
vector<vec3> profileCurve;
for (int j = 0; j <= ptInterval; j++)
{
    profileCurve.push_back(resizeM * vec4(curveVertexs[idx1↩
][j], 1.0));
}

vector<vec3> lastline;
vector<vec3> lastNormalLine;
vec3 V = sweepCurve[0];
vec3 coorX = curveNormals[idx2 - 1][0];
vec3 coorY = curveBs[idx2 - 1][0];
vec3 coorZ = curveTans[idx2 - 1][0];
float coorMp[16] =
{
    coorX.x,coorY.x,coorZ.x,V.x,
    coorX.y,coorY.y,coorZ.y,V.y,
    coorX.z,coorY.z,coorZ.z,V.z,
    0,0,0,1
};
float invcoorMp[9] =
{
    coorX.x,coorY.x,coorZ.x,
    coorX.y,coorY.y,coorZ.y,
    coorX.z,coorY.z,coorZ.z
};
mat4 tranM = glm::transpose( glm::make_mat4(coorMp));
mat3 inv = glm::inverse(glm::make_mat3(invcoorMp));
for (int j = 0; j <= ptInterval; j++)
{
    vec4 tmp1 = tranM * vec4(profileCurve[j], 1.0);
    lastline.push_back(tmp1/tmp1.w);
    lastNormalLine.push_back(inv * profileCurve[j]);
}

for (int i = 1; i <= stInterval; i++)
{
    vector<vec3> thisline;
    vector<vec3> thisNormalLine;
```

```
        vec3 V = sweepCurve[i];
        vec3 coorX = curveNormals[idx2 - 1][i];
        vec3 coorY = curveBs[idx2 - 1][i];
        vec3 coorZ = curveTans[idx2 - 1][i];
        float coorMp[16] =
        {
            coorX.x,coorY.x,coorZ.x,V.x,
            coorX.y,coorY.y,coorZ.y,V.y,
            coorX.z,coorY.z,coorZ.z,V.z,
            0,0,0,1
        };
        float invcoorMp[9] =
        {
            coorX.x,coorY.x,coorZ.x,
            coorX.y,coorY.y,coorZ.y,
            coorX.z,coorY.z,coorZ.z
        };
        mat4 tranM = glm::transpose(glm::make_mat4(coorMp));
        mat3 inv = glm::inverse(glm::make_mat3(invcoorMp));

        vec4 tmp1 = tranM * vec4(profileCurve[0], 1.0);
        vec3 lastPoint = tmp1 / tmp1.w;
        vec3 lastNormal = inv * profileCurve[0];
        thisline.push_back(lastPoint);
        thisNormalLine.push_back(lastNormal);
        for (int j = 1; j <= ptInterval; j++)
        {
            vec4 tmp1 = tranM * vec4(profileCurve[j], 1.0);
            vec3 thisPoint = tmp1 / tmp1.w;
            vec3 thisNormal = inv * profileCurve[j];
            thisline.push_back(thisPoint);
            thisNormalLine.push_back(thisNormal);
            o.push_back(lastPoint);
            o.push_back(thisPoint);
            o.push_back(lastline[j]);

            o.push_back(lastPoint);
            o.push_back(lastline[j]);
            o.push_back(lastline[j - 1]);

            o1.push_back(lastNormal);
            o1.push_back(thisNormal);
            o1.push_back(lastNormalLine[j]);

            o1.push_back(lastNormal);
            o1.push_back(lastNormalLine[j]);
```

```
            o1.push_back(lastNormalLine[j - 1]);
            lastPoint = thisPoint;
            lastNormal = thisNormal;
        }
        lastline = thisline;
        lastNormalLine = thisNormalLine;
    }
    faceVertexs.push_back(o);
    faceNormals.push_back(o1);
}

void data()
{
    calculateBezier(SORProfile, zview, ptInterval,false);
    generateCurveOut(1, ptInterval);
    calculateBezier(SSProfile, zview, ptInterval, true);
    generateCurveOut(3, ptInterval);
    calculateBezier(SSCurve, yview, stInterval, true, true);
    generateCurveOut(5, stInterval-1);

    generateSOR(1);
    generateSS(3, 5);
}

int main(int argc, char* argv[]) {
    data();
    init();
    loop();
    outputObjFile("SOR.obj", faceVertexs[0], faceNormals[0]);
    outputObjFile("SS.obj", faceVertexs[1], faceNormals[1]);
}
```

Code 16.2: object.fs.glsl

```
#version 430 core

in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;


out vec4 color;

void main(){
```

```
    // Light emission properties
    vec3 LightColor = vec3(1,1,1);
    float LightPower = 50.0f;

    // Material properties
    vec3 MaterialDiffuseColor = vec3(0.5,0.5,1.0);
    vec3 MaterialAmbientColor = vec3(0.1,0.1,0.1) * ↩
    MaterialDiffuseColor;
    vec3 MaterialSpecularColor = vec3(0.3,0.3,0.3);

    // Distance to the light
    float distance = length( vec3(7,10,7) - Position_worldspace↩
     );

    vec3 n = normalize( Normal_cameraspace );
    vec3 l = normalize( LightDirection_cameraspace );
    float cosTheta = clamp( dot( n,l ), 0,1 );

    vec3 E = normalize(EyeDirection_cameraspace);
    vec3 R = reflect(-l,n);
    float cosAlpha = clamp( dot( E,R ), 0,1 );

    color.rgb=// Ambient
        MaterialAmbientColor +
        // Diffuse
        MaterialDiffuseColor * LightColor * LightPower * ↩
    cosTheta / (distance*distance) +
        // Specular
        MaterialSpecularColor * LightColor * LightPower * pow(↩
    cosAlpha,5) / (distance*distance);
    color.a=1;
}
```

Code 16.3: object.vs.glsl

```
#version 430 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexNormal_modelspace;

out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;
```

```
uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;

void main(){

    gl_Position =  MVP * vec4(vertexPosition_modelspace, 1);
    Position_worldspace = (M * vec4(vertexPosition_modelspace↩
    ,1)).xyz;
    vec3 vertexPosition_cameraspace = ( V * M * vec4(↩
    vertexPosition_modelspace,1)).xyz;
    EyeDirection_cameraspace = vec3(0,0,0) - ↩
    vertexPosition_cameraspace;
    vec3 LightPosition_cameraspace = ( V * vec4(7,10,7,1)).xyz;
    LightDirection_cameraspace = LightPosition_cameraspace + ↩
    EyeDirection_cameraspace;
    Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace↩
    ,0)).xyz;
}
```

Code 16.4: curve.fs.glsl

```
#version 330 core

uniform vec3 lcolor;

out vec4 FragColor;

void main()
{
    FragColor = vec4(lcolor,1.0);
}
```

Code 16.5: curve.vs.glsl

```
#version 330 core
layout (location = 2) in vec3 aPos;

uniform mat4 MVP;

void main()
{
    gl_Position = MVP * vec4(aPos, 1.0);
```

```
}
```

Code 16.6: utils/loadTexture.h

```
#pragma once
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

GLuint loadDDS(const char* imagepath);
```

Code 16.7: utils/loadTexture.cpp

```
#include "loadTexture.h"
/*
* modified from: ogl-master
* common/texture.cpp
* */
#define FOURCC_DXT1 0x31545844 // Equivalent to "DXT1" in ASCII
#define FOURCC_DXT3 0x33545844 // Equivalent to "DXT3" in ASCII
#define FOURCC_DXT5 0x35545844 // Equivalent to "DXT5" in ASCII

GLuint loadDDS(const char* imagepath) {

    unsigned char header[124];

    FILE* fp;

    /* try to open the file */
    fp = fopen(imagepath, "rb");
    if (fp == NULL) {
        printf("%s␣could␣not␣be␣opened.\n", imagepath); getchar↩
    ();
        return 0;
    }

    /* verify the type of file */
    char filecode[4];
    fread(filecode, 1, 4, fp);
    if (strncmp(filecode, "DDS␣", 4) != 0) {
```

```
    fclose(fp);
    return 0;
}

/* get the surface desc */
fread(&header, 124, 1, fp);

unsigned int height = *(unsigned int*)&(header[8]);
unsigned int width = *(unsigned int*)&(header[12]);
unsigned int linearSize = *(unsigned int*)&(header[16]);
unsigned int mipMapCount = *(unsigned int*)&(header[24]);
unsigned int fourCC = *(unsigned int*)&(header[80]);


unsigned char* buffer;
unsigned int bufsize;
/* how big is it going to be including all mipmaps? */
bufsize = mipMapCount > 1 ? linearSize * 2 : linearSize;
buffer = (unsigned char*)malloc(bufsize * sizeof(unsigned ↩
char));
fread(buffer, 1, bufsize, fp);
/* close the file pointer */
fclose(fp);

unsigned int components = (fourCC == FOURCC_DXT1) ? 3 : 4;
unsigned int format;
switch (fourCC)
{
case FOURCC_DXT1:
    format = GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
    break;
case FOURCC_DXT3:
    format = GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
    break;
case FOURCC_DXT5:
    format = GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
    break;
default:
    free(buffer);
    return 0;
}

// Create one OpenGL texture
GLuint textureID;
glGenTextures(1, &textureID);
```

```
    // "Bind" the newly created texture : all future texture ↩
    functions will modify this texture
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    unsigned int blockSize = (format == ↩
    GL_COMPRESSED_RGBA_S3TC_DXT1_EXT) ? 8 : 16;
    unsigned int offset = 0;

    /* load the mipmaps */
    for (unsigned int level = 0; level < mipMapCount && (width ↩
    || height); ++level)
    {
        unsigned int size = ((width + 3) / 4) * ((height + 3) /↩
     4) * blockSize;
        glCompressedTexImage2D(GL_TEXTURE_2D, level, format, ↩
    width, height,
            0, size, buffer + offset);

        offset += size;
        width /= 2;
        height /= 2;

        // Deal with Non-Power-Of-Two textures. This code is ↩
    not included in the webpage to reduce clutter.
        if (width < 1) width = 1;
        if (height < 1) height = 1;

    }

    free(buffer);

    return textureID;


}
```

Code 16.8: utils/mathTool.h

```
#pragma once
# include <vector>
# include <cmath>
# include <glm/glm.hpp>
using glm::vec3;
using std::vector;
```

```
vec3 quadBezier(vec3 a, vec3 b, vec3 c, float t);
vec3 cubicBezier(vec3 a, vec3 b, vec3 c, vec3 d, float t);
vec3 cubicBeziers(vector<vec3> points, float t, bool closed = ↩
    false);
vec3 quadBezierTan(vec3 a, vec3 b, vec3 c, float t);
vec3 cubicBezierTan(vec3 a, vec3 b, vec3 c, vec3 d, float t);
vec3 getTan(vector<vec3> points, float t, bool closed = false);
bool isCounterClockWise(vector<vec3> points, vec3 view);
```

Code 16.9: utils/mathTool.cpp

```
# include "mathTool.h"

vec3 quadBezier(vec3 a, vec3 b, vec3 c, float t)
{
    if (t > 1 || t < 0)
        return vec3(0, 0, 0);
    return pow((1 - t), 2) * a + 2 * (1 - t) * t * b
        + pow(t, 2) * c;
}

vec3 cubicBezier(vec3 a, vec3 b, vec3 c, vec3 d, float t)
{
    if (t > 1 || t < 0)
        return vec3(0, 0, 0);
    return pow((1 - t), 3) * a + 3 * pow((1 - t), 2) * t * b
        + 3 * (1 - t) * pow(t, 2) * c + pow(t, 3) * d;
}

vec3 quadBezierTan(vec3 a, vec3 b, vec3 c, float t)
{
    if (t > 1 || t < 0)
        return vec3(0, 0, 0);
    return 2 * (1 - t) * (b - a) + 2 * t * (c - b);
}

vec3 cubicBezierTan(vec3 a, vec3 b, vec3 c, vec3 d, float t)
{
    if (t > 1 || t < 0)
        return vec3(0, 0, 0);
    return 3 * pow((1 - t), 2) * (b - a) + 6 * (1 - t) * t * (c↩
     - b)
        + 3 * pow(t, 2) * (d - c);
}
```

```
vec3 cubicBeziers(vector<vec3> points, float t, bool closed)
{
    if (points.size() == 0)
        return vec3(0, 0, 0);
    int start = floor(t * (float)(points.size() + closed-1) / ↩
    3) * 3;
    float portion = (t * (float)(points.size() + closed-1) - (↩
    float)start) / 3.0f;
    if (start + 3 >= points.size() + closed)
    {
        if (start + 2 >= points.size() + closed)
        {
            if (start + 1 >= points.size() + closed)
            {
                return points[start% points.size()];
            }
            else
            {
                // 2
                portion *= 3.0f;
                return (1 - portion) * points[start] + portion ↩
    * points[(start + 1)% points.size()];
            }
        }
        else
        {
            // 3
            portion *= 3.0f / 2;
            return quadBezier(points[start], points[start + 1],↩
     points[(start + 2) % points.size()], portion);
        }
    }
    else
    {
        // 4
        return cubicBezier(points[start], points[start + 1], ↩
    points[start + 2], points[(start + 3) % points.size()], ↩
    portion);
    }
}


vec3 getTan(vector<vec3> points, float t, bool closed)
{
    if (points.size() == 0 || points.size() + closed == 1)
```

```
        return vec3(0, 0, 0);
    int start = floor(t * (float)(points.size() + closed-1) / ↩
    3) * 3;
    float portion = (t * (float)(points.size() + closed-1) - (↩
    float)start) / 3.0f;
    if (start + 3 >= points.size() + closed)
    {
        if (start + 2 >= points.size() + closed)
        {
            if (start + 1 >= points.size() + closed)
            {
                // TODO may loop, debug here
                // last point
                return getTan(points, t - 0.0001, closed);
            }
            else
            {
                return -points[start] + points[(start + 1) % ↩
    points.size()];
            }
        }
        else
        {
            // 3
            portion *= 2.0f / 3;
            return quadBezierTan(points[start], points[start + ↩
    1], points[(start + 2) % points.size()], portion);
        }
    }
    else
    {
        // 4
        return cubicBezierTan(points[start], points[start + 1],↩
     points[start + 2], points[(start + 3) % points.size()], ↩
    portion);
    }
}


bool isCounterClockWise(vector<vec3> points, vec3 view)
{
    if (points.size() > 2)
    {
        vec3 c = vec3(1, 0, 0);
        for (int i = 0; i < points.size(); i++)
            c += points[i];
```

```
        c /= (float)points.size();
        vec3 y = glm::cross((points[0] - c), (points[1] - c));
        return glm::dot(y, view) > 0 ? true : false;
    }
    else
        return true;
}
```

Code 16.10: utils/meshio.h

```
#pragma once
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <sstream>
#include <cstring>
#include <stdlib.h>

#include <GL/glew.h>
#include <glm/glm.hpp>
#include <GLFW/glfw3.h>

// Include AssImp
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // Output data structure
#include <assimp/postprocess.h>     // Post processing flags
using std::vector;
using glm::vec2;
using glm::vec3;
using glm::vec4;

struct Mesh
{
    int vn;
    int fn;
    vector<vec3> vs;
    vector<vec2> vts;
    vector<vec3> vns;
    vector<short> fs;
    vector<vec3> maxvs;
    vector<vec3>  minvs;
    vector<vec3>  ns;
```

```
};

bool loadObj(
    bool hasDDS,
    int start0,
    int end0,
    const char* path,
    Mesh& mesh
);

void outputObjFile(std::string name, const vector<vec3>& vertex↩
    , const vector<vec3>& normal);

void LoadShader(GLuint ShaderID, const char* file_path);

GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path);
```

Code 16.11: utils/meshio.cpp

```
#include "meshio.h"

/*
* modified from: ogl-master
* common/objloader.cpp
* author: opengl-tutorial.org
*/

// anyway, we only allow one mesh and one texture here,
// and other input may cause rendering error as a result
bool loadObj(
    bool hasDDS,
    int start0,
    int end0,
    const char* path,
    Mesh& mesh
) {
    Assimp::Importer importer;

    const aiScene* scene = importer.ReadFile(
        path,
        0       /*aiProcess_JoinIdenticalVertices | ↩
    aiProcess_SortByPType*/
    );
    if (!scene) {
```

```
        fprintf(stderr, importer.GetErrorString());
        getchar();
        return false;
    }

    int start = start0 == -1 ? 0 : start0;
    int end = end0 == -1 ? scene->mNumMeshes : end0 + 1;
    // for now the models donnot overlap
    for (int j = start; j < end; j++)
    {
        int vertexnum = mesh.vs.size();
        const aiMesh* inmesh = scene->mMeshes[j];

        mesh.vn += inmesh->mNumVertices;
        mesh.fn += inmesh->mNumFaces;

        // Fill vertices positions
        mesh.vs.reserve(vertexnum + inmesh->mNumVertices);
        for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
     {
            aiVector3D pos = inmesh->mVertices[i];
            mesh.vs.push_back(vec3(pos.x, pos.y, pos.z));
        }

        mesh.vts.reserve(vertexnum + inmesh->mNumVertices);
        // no texture
        if (hasDDS)
        {
            // Fill vertices texture coordinates
            for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
    i++) {
                aiVector3D UVW = inmesh->mTextureCoords[j][i];
                // Assume only 1 set of UV coords; AssImp ↩
    supports 8 UV sets.
                mesh.vts.push_back(vec2(UVW.x, -UVW.y));
            }
        }
        else
        {
            for (unsigned int i = 0; i < inmesh->mNumVertices; ↩
    i++) {
                mesh.vts.push_back(vec2(0, 0));
            }
        }

        // Fill vertices normals
```

```
    mesh.vns.reserve(vertexnum + inmesh->mNumVertices);
    for (unsigned int i = 0; i < inmesh->mNumVertices; i++)↩
 {
        aiVector3D n = inmesh->mNormals[i];
        mesh.vns.push_back(glm::vec3(n.x, n.y, n.z));
    }


    // Fill face indices
    mesh.fs.reserve(mesh.fs.size() + 3 * inmesh->mNumFaces)↩
;
    mesh.maxvs.reserve(mesh.maxvs.size() + inmesh->↩
mNumFaces);
    mesh.minvs.reserve(mesh.minvs.size() + inmesh->↩
mNumFaces);
    mesh.ns.reserve(mesh.ns.size() + inmesh->mNumFaces);
    for (unsigned int i = 0; i < inmesh->mNumFaces; i++) {
        // Assume the model has only triangles.
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[0] + ↩
vertexnum);
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[1] + ↩
vertexnum);
        mesh.fs.push_back(inmesh->mFaces[i].mIndices[2] + ↩
vertexnum);
        vec3 v1 = mesh.vs[inmesh->mFaces[i].mIndices[0] + ↩
vertexnum];
        vec3 v2 = mesh.vs[inmesh->mFaces[i].mIndices[1] + ↩
vertexnum];
        vec3 v3 = mesh.vs[inmesh->mFaces[i].mIndices[2] + ↩
vertexnum];
        mesh.maxvs.push_back(glm::max(glm::max(v1, v2), v3)↩
);
        mesh.minvs.push_back(glm::min(glm::min(v1, v2), v3)↩
);
        // no interpolation for edges
        vec3 n1 = mesh.vns[inmesh->mFaces[i].mIndices[0] + ↩
vertexnum];
        vec3 n2 = mesh.vns[inmesh->mFaces[i].mIndices[1] + ↩
vertexnum];
        vec3 n3 = mesh.vns[inmesh->mFaces[i].mIndices[2] + ↩
vertexnum];
        mesh.ns.push_back(glm::normalize(n1 + n2 + n3));
    }

}
```

```
    return true;
}

// 6-837 MIT computer graphics basic code
void outputObjFile(std::string name, const vector<vec3>& vertex↩
    , const vector<vec3>& normal)
{
    std::ofstream out(name.c_str());

    if (!out)
    {
        std::cout<<name.c_str()<<"cannot␣be␣opened";
    }
    else
    {
        std::cout << "writing␣to␣"<<name.c_str() ;
        for (unsigned i = 0; i < vertex.size(); i++)
            out << "v␣␣"
            << vertex[i][0] << "␣"
            << vertex[i][1] << "␣"
            << vertex[i][2] << std::endl;

        for (unsigned i = 0; i < normal.size(); i++)
            out << "vn␣"
            << normal[i][0] << "␣"
            << normal[i][1] << "␣"
            << normal[i][2] << std::endl;

        out << "vt␣␣0␣0␣0" << std::endl;

        int t = 0;
        for (unsigned i = 0; i < vertex.size() / 3; i++)
        {
            out << "f␣␣";
            for (unsigned j = 0; j < 3; j++)
            {
                t += 1;
                out << t << "/" << 1 << "/" << t << "␣";
            }
            out << std::endl;
        }
        std::cout <<"wrote␣" << name << std::endl;
    }

}
```

```
/*
* modified from: ogl-master
* common/shader.cpp
* author: opengl-tutorial.org
*/
void LoadShader(GLuint ShaderID, const char* file_path)
{
    // Read the Vertex Shader code from the file
    std::string ShaderCode;
    std::ifstream ShaderStream(file_path, std::ios::in);
    if (ShaderStream.is_open()) {
        std::stringstream sstr;
        sstr << ShaderStream.rdbuf();
        ShaderCode = sstr.str();
        ShaderStream.close();
    }
    else {
        printf("Impossible␣to␣open␣%s.␣\n", file_path);
        getchar();
        exit(-1);
    }

    GLint Result = GL_FALSE;
    int InfoLogLength;

    printf("Compiling␣shader␣:␣%s\n", file_path);
    char const* xSourcePointer = ShaderCode.c_str();
    glShaderSource(ShaderID, 1, &xSourcePointer, NULL);
    glCompileShader(ShaderID);

    glGetShaderiv(ShaderID, GL_COMPILE_STATUS, &Result);
    glGetShaderiv(ShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength)↩
    ;
    if (InfoLogLength > 0) {
        std::vector<char> ShaderErrorMessage(InfoLogLength + 1)↩
    ;
        glGetShaderInfoLog(ShaderID, InfoLogLength, NULL, &↩
    ShaderErrorMessage[0]);
        printf("%s\n", &ShaderErrorMessage[0]);
    }
}



GLuint LoadShaders(const char* vertex_file_path, const char* ↩
    fragment_file_path) {
```

```
    // Create the shaders
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER↩
    );

    LoadShader(VertexShaderID, vertex_file_path);
    LoadShader(FragmentShaderID, fragment_file_path);

    GLint Result = GL_FALSE;
    int InfoLogLength;

    // Link the program
    printf("Linking␣program\n");
    GLuint ProgramID = glCreateProgram();
    glAttachShader(ProgramID, VertexShaderID);
    glAttachShader(ProgramID, FragmentShaderID);
    glLinkProgram(ProgramID);

    // Check the program
    glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
    glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &↩
    InfoLogLength);
    if (InfoLogLength > 0) {
        std::vector<char> ProgramErrorMessage(InfoLogLength + ↩
    1);
        glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &↩
    ProgramErrorMessage[0]);
        printf("%s\n", &ProgramErrorMessage[0]);
    }


    glDetachShader(ProgramID, VertexShaderID);
    glDetachShader(ProgramID, FragmentShaderID);

    glDeleteShader(VertexShaderID);
    glDeleteShader(FragmentShaderID);

    return ProgramID;
}
```

Bibliography

[1] 2.6.1. ray tracing with OpenGL compute shaders (part i) · LWJGL/lwjgl3-wiki wiki.

[2] OpenGL projection matrix.

[3] Slerp.

[4] Shader - OpenGL wiki.

[5] Rendering pipeline overview - OpenGL wiki.

[6] Phong reflection model.

[7] File:flip book animation created using ray tracing in 1976.gif - wikipedia.

[8] Ray tracing (graphics).