计算机图形学课程报告

计算机科学与技术学院

班　　级：	**CS1601**
学　　号：	**U201614531**
姓　　名：	**刘本嵩**
指导教师：	
完成日期：	**2019.12.17**

1 回答问题

（1）你选修计算机图形学课程，想得到的是什么知识？现在课程结束，对于所得的知识是否满意？如果不满意，你准备如何寻找自己需要的知识。

想大致了解计算机图形学相关知识，例如了解游戏界面如何渲染之类的常识。满意。

（2）你对计算机图形学课程中的哪一个部分的内容最感兴趣，请简述之。

图形渲染相关内容。这部分涉及到线性代数等知识，高级光线处理等部分还涉及物理学知识。

（3）你对计算机图形学课程的内容，教学方法有什么看法和建议。

无

2 实验内容

请注意，下面的源码有库依赖，请在随报告附赠的源码包获取，或访问 Gitlab: https://git.recolic.org/recolic-hust/opengl

2.1 OpenGL 动画

2.1.1 实验内容及要求

利用 OpenGL，设计一个动画，让一个矩形（或者是球）沿着一个椭圆的轨道运行，椭圆的长轴和短轴的比值为 4:3。

（1）可以参考中点法画椭圆的方式

（2）注意椭圆与坐标轴的四个交点以及切线斜率为正负 1 的位置。

（3）可以利用 GLUT 框架实现

（4）注意使用双缓存，其他颜色等属性不限。

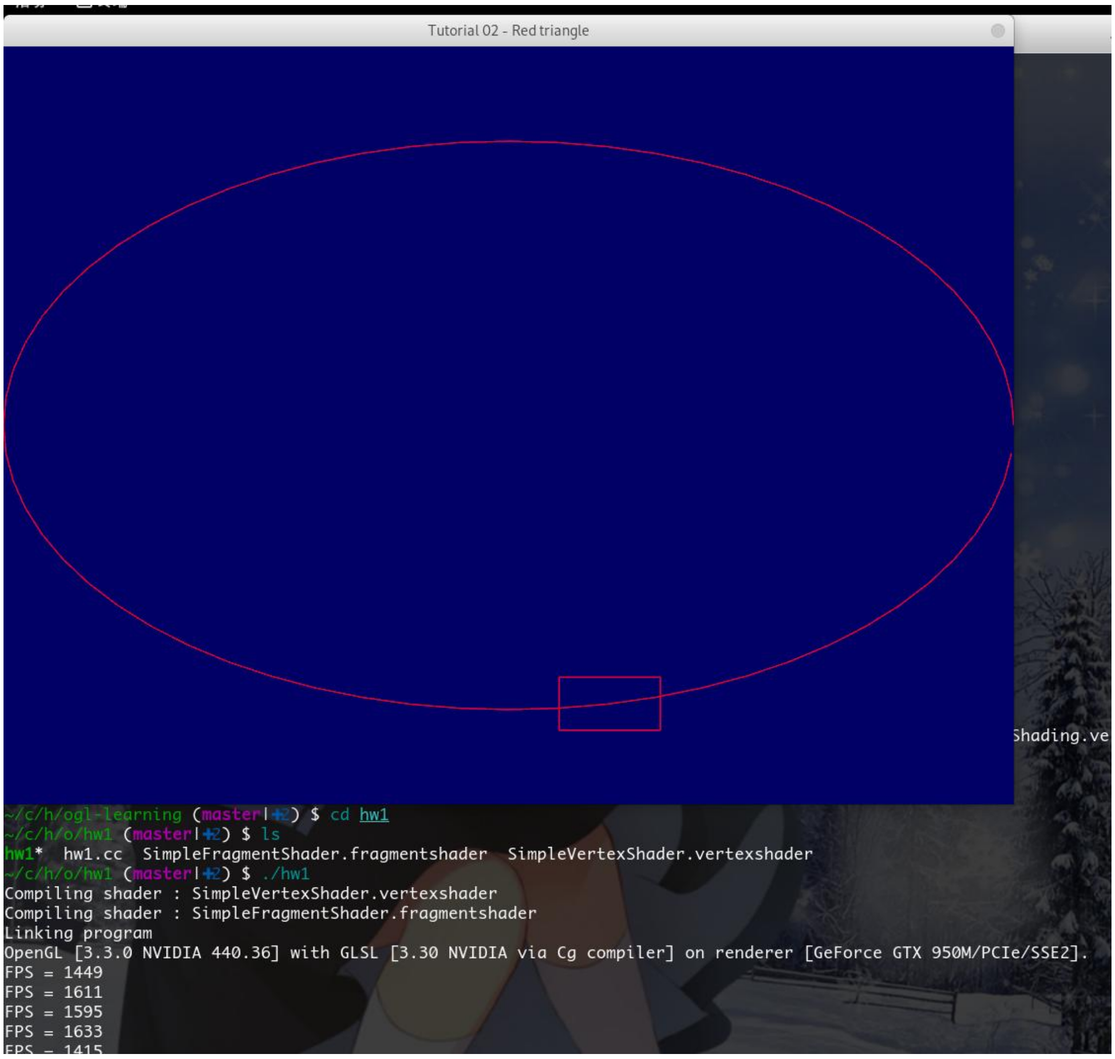
2.1.2 实现的方法、过程

实现函数 calc_square_vertex_buf_data，用来生成一个矩形所需的 vertex buffer。

实现函数 calc_ellipse_point，输入 0 到 1 的一个数字，用来确定椭圆上对应的一个点。

实现函数 calc_ellipse_vertex_buf_data，用来预先把椭圆所需的 vertex buffer 画好。

其余代码都是固定套路，例如初始化 VAO，VBO，texture，shader，计算 fps 信息，逐帧渲染等等。



```
~/c/h/ogl-learning (master|+2) $ cd hw1
~/c/h/o/hw1 (master|+2) $ ls
hw1*  hw1.cc  SimpleFragmentShader.fragmentshader  SimpleVertexShader.vertexshader
~/c/h/o/hw1 (master|+2) $ ./hw1
Compiling shader : SimpleVertexShader.vertexshader
Compiling shader : SimpleFragmentShader.fragmentshader
Linking program
OpenGL [3.3.0 NVIDIA 440.36] with GLSL [3.30 NVIDIA via Cg compiler] on renderer [GeForce GTX 950M/PCIe/SSE2].
FPS = 1449
FPS = 1611
FPS = 1595
FPS = 1633
FPS = 1415
```

2.1.3 源程序

程序 1：OpenGL 动画

```
//*# -------------------- ./SimpleVertexShader.vertexshader ---------------------*/
#version 330 core
```

```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

void main(){

	gl_Position.xyz = vertexPosition_modelspace;
	gl_Position.w = 1.0;

}

//*# -------------------- ./hw1.cc ---------------------*/
// Include standard headers
#include <stdio.h>
#include <stdlib.h>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;

// Include GLM
#include <glm/glm.hpp>
using namespace glm;

#include <common/shader.hpp>
#include <vector>
#include <chrono>
using namespace std::chrono;

#include <rlib/stdio.hpp>
#include <common/fps.hpp>
```

```
auto calc_square_vertex_buf_data(std::pair<float, float> center) {
	auto dx = 0.1, dy = 0.07;
	auto x1 = center.first - dx, x2 = center.first + dx,
	     y1 = center.second - dy, y2 = center.second + dy;

#ifdef DEBUG
	rlib::println("debug: square, x1,y1,x2,y2 = ", x1, y1, x2, y2);
#endif

	return std::vector<float>{
		x1, y1,
		x1, y2,
		x1, y1,
		x2, y1,

		x2, y2,
		x1, y2,
		x2, y2,
		x2, y1
	};
}

auto calc_ellipse_point(float location_0_to_1) {
	auto y_sign = location_0_to_1 > 0.5 ? -1.0 : 1.0;
	auto x = cos(location_0_to_1 * 2 * M_PI);
	auto y = sqrt(1.0-x*x) * 0.75 * y_sign;
#ifdef DEBUG
	rlib::println("debug: center x,y = ", x, y);
#endif
	return std::make_pair((float)x, (float)y);
}

auto calc_ellipse_vertex_buf_data()
{
	const float cx = 0, cy = 0, rx = 1, ry = 0.75, num_segments = 64;
```

```
    float theta = M_PI * 2 / float(num_segments);
    float c = cosf(theta); //precalculate the sine and cosine
    float s = sinf(theta);
    float t;

    float x = 1;//we start at angle = 0
    float y = 0;

    std::vector<float> buf;
    for(int ii = 0; ii < num_segments; ++ii)
    {
        //apply radius and offset
        if(ii != 0) { buf.emplace_back(x * rx + cx); buf.emplace_back(y * ry +
cy); }
        if(ii != num_segments-1) { buf.emplace_back(x * rx + cx);
buf.emplace_back(y * ry + cy); }

        //apply the rotation matrix
        t = x;
        x = c * x - s * y;
        y = s * t + c * y;
    }

    return buf;
}

int main( void )
{
    // Make println faster
    // rlib::enable_endl_flush(false);
    rlib::sync_with_stdio(false);

    // Initialise GLFW
    if( !glfwInit() )
```

```
	{
		fprintf( stderr, "Failed to initialize GLFW\n" );
		getchar();
		return -1;
	}

	glfwWindowHint(GLFW_SAMPLES, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make
MacOS happy; should not be needed
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	// Open a window and create its OpenGL context
	window = glfwCreateWindow( 1024, 768, "Tutorial 02 - Red triangle", NULL,
NULL);
	if( window == NULL ){
		fprintf( stderr, "Failed to open GLFW window. If you have an Intel GPU,
they are not 3.3 compatible. Try the 2.1 version of the tutorials.\n" );
		getchar();
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);

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

	// Ensure we can capture the escape key being pressed below
	glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

	// Dark blue background
	glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

	GLuint VertexArrayID;
	glGenVertexArrays(1, &VertexArrayID);
	glBindVertexArray(VertexArrayID);

	// Create and compile our GLSL program from the shaders
	GLuint   programID   =   LoadShaders(   "SimpleVertexShader.vertexshader",
"SimpleFragmentShader.fragmentshader" );

	GLuint vertexbuffer;
	glGenBuffers(1, &vertexbuffer);
	glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);

	const size_t msPerRound = 10000;
	auto ellipse_vertex_buffer = calc_ellipse_vertex_buf_data();

	init_fps();

	do {
		// Prepare vertex buffer
		auto g_vertex_buffer = ellipse_vertex_buffer;

		auto location = (float)( duration_cast< milliseconds >(
			system_clock::now().time_since_epoch()
		).count() % msPerRound ) / msPerRound;
		auto                                square_vertex_buffer                              =
std::move(calc_square_vertex_buf_data(calc_ellipse_point(location)));

		g_vertex_buffer.insert(g_vertex_buffer.end(),
square_vertex_buffer.begin(), square_vertex_buffer.end());
		 glBufferData(GL_ARRAY_BUFFER, g_vertex_buffer.size() * sizeof(float),
g_vertex_buffer.data(), GL_STATIC_DRAW);

            // Clear the screen
            glClear( GL_COLOR_BUFFER_BIT );

            // Use our shader
            glUseProgram(programID);

            // 1rst attribute buffer : vertices
            glEnableVertexAttribArray(0);
            glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
            glVertexAttribPointer(
                  0,                              // attribute 0. No particular reason for 0, but
must match the layout in the shader.
                  2,                          // size
                  GL_FLOAT,                   // type
                  GL_FALSE,                   // normalized?
                  0,                          // stride
                  (void*)0                    // array buffer offset
            );

            // Draw the lines!
            glDrawArrays(GL_LINES, 0, g_vertex_buffer.size() / 2);
            ++ fps_counter;

            glDisableVertexAttribArray(0);

            // Swap buffers
            glfwSwapBuffers(window);
            glfwPollEvents();

      } // Check if the ESC key was pressed or the window was closed
      while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
                glfwWindowShouldClose(window) == 0 );
```

```
    // Cleanup VBO
    glDeleteBuffers(1, &vertexbuffer);
    glDeleteVertexArrays(1, &VertexArrayID);
    glDeleteProgram(programID);

    // Close OpenGL window and terminate GLFW
    glfwTerminate();

    return 0;
}

//*# -------------------- ./SimpleFragmentShader.fragmentshader ---------------------*/
#version 330 core

// Ouput data
out vec3 color;

void main()
{

    // Output color = red
    color = vec3(1,0,0);

}
```

2.2 日地月模型

2.2.1 实验内容及要求

利用 OpenGL，设计一个日地月运动模型动画。

（1）月亮绕地球可以使用圆形轨道，地球绕太阳使用实验 1 的椭圆轨道。

（2）运动关系正确，相对速度合理，且圆形轨道和椭圆轨道不能在一个平面内。

（3）增加光照处理，光源设在太阳上面。

（4）为了提高太阳的显示效果，可以在侧后增加一个专门照射太阳的灯。

（5）加分项：增加纹理处理。

2.2.2 实现的方法、过程

除了框架代码外,寻找地球月球太阳和星空的高清图片,写一个函数从图片建立球纹理。背景星空就是一个巨大半径的球，和其他星球用相同方式处理。

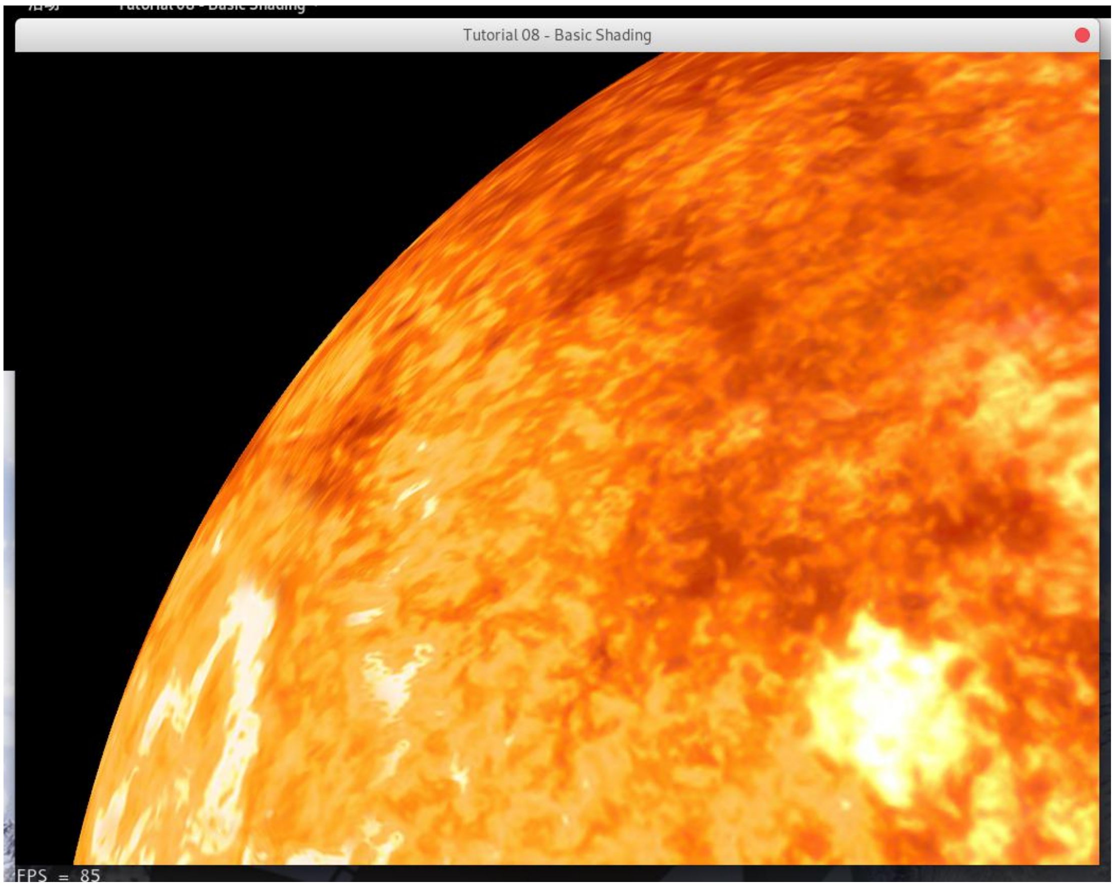
阴影效果，光照效果，光线反射效果均有标准 shader 的实现，稍作修改即可采用。

地球/月球/太阳的自转由 rotatePlanet 实现，公转由 orbitPlanet 实现。

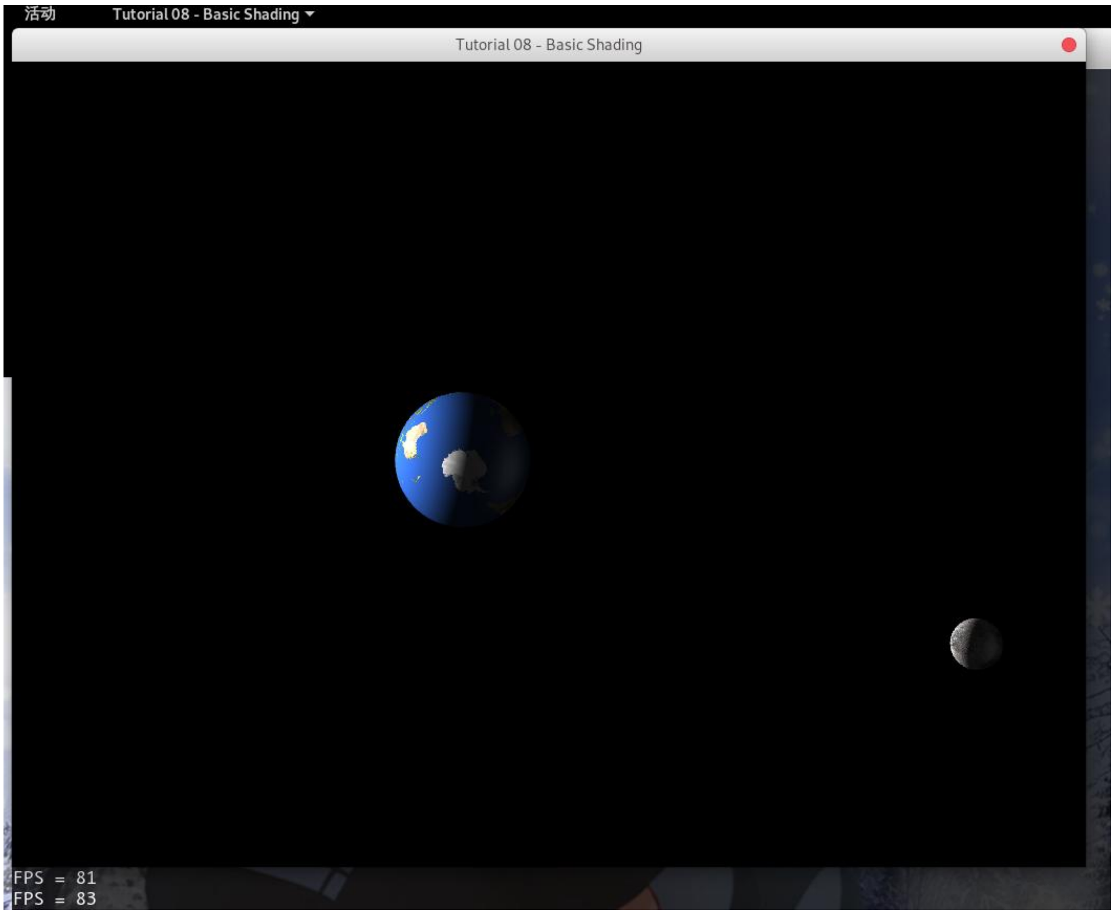
玩家操作时，使用 WASD 控制前后左右移动，用空格键向上飞行，Z 键向下飞行。移动鼠标可以自由移动视角，主键盘+(=)键可以加快宇宙运行，-键减慢宇宙运行，PAUSE 键暂停宇宙运行，再按一次即可恢复运行。

由于未知 bug，背景星空未能正常显示。

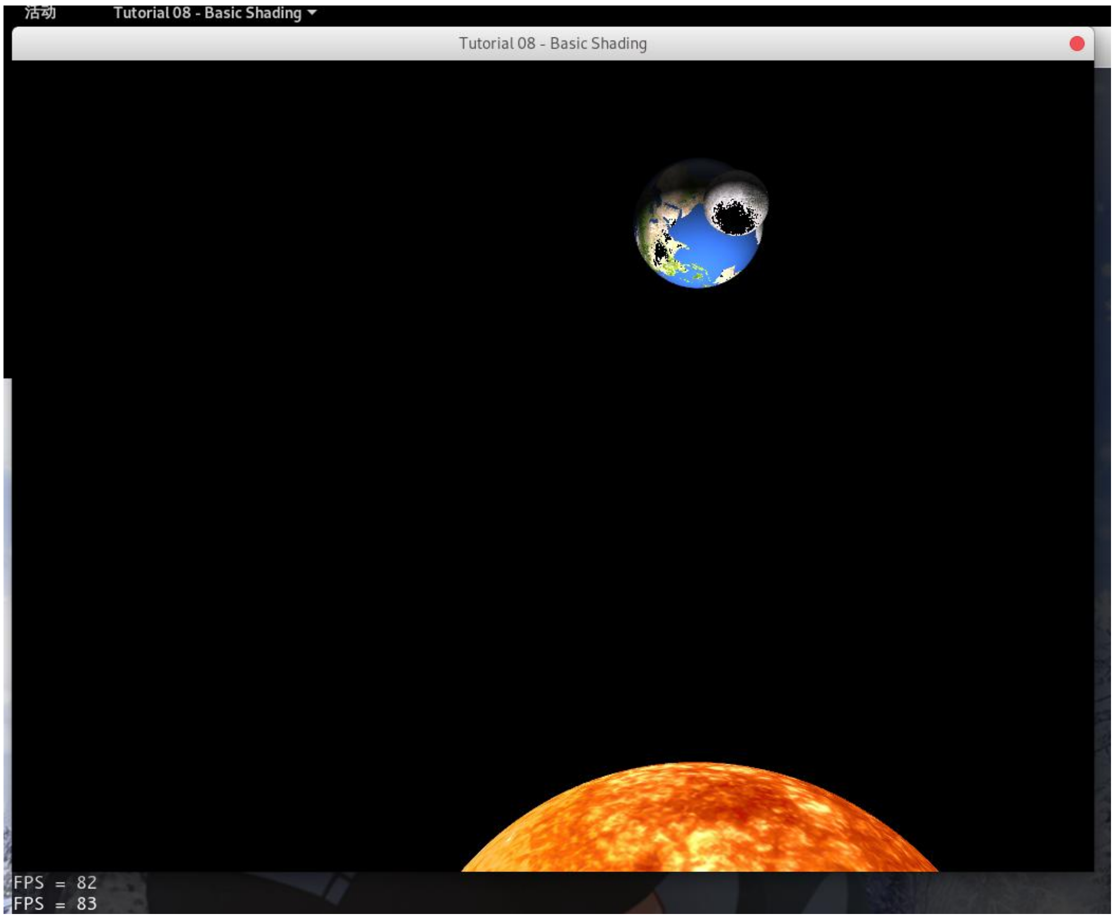
因为反射系数被设置的非常小，因此星球反射太阳光可能难以在截图中察觉。



```
FPS = 85
```



```
FPS = 81
FPS = 83
```



```
FPS = 82
FPS = 83
```

2.2.3 源程序

程序 2：日地月模型

```
//*# -------------------- ./StandardShading.vertexshader ---------------------*/
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexNormal_modelspace;
layout(location = 2) in vec2 vertexUV;

// Output data ; will be interpolated for each fragment.
out vec2 UV;
out vec3 Position_worldspace;
out vec3 Normal_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 LightDirection_cameraspace;
```

```
// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;
uniform vec3 LightPosition_worldspace;

void main(){

    // Output position of the vertex, in clip space : MVP * position
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

    // Position of the vertex, in worldspace : M * position
    Position_worldspace = (M * vec4(vertexPosition_modelspace,1)).xyz;

    // Vector that goes from the vertex to the camera, in camera space.
    // In camera space, the camera is at the origin (0,0,0).
    vec3 vertexPosition_cameraspace = ( V * M *
vec4(vertexPosition_modelspace,1)).xyz;
    EyeDirection_cameraspace = vec3(0,0,0) - vertexPosition_cameraspace;

    // Vector that goes from the vertex to the light, in camera space. M is
ommited because it's identity.
    vec3 LightPosition_cameraspace = ( V *
vec4(LightPosition_worldspace,1)).xyz;
    LightDirection_cameraspace = LightPosition_cameraspace +
EyeDirection_cameraspace;

    // Normal of the the vertex, in camera space
    Normal_cameraspace = ( V * M * vec4(vertexNormal_modelspace,0)).xyz; //
Only correct if ModelMatrix does not scale the model ! Use its inverse transpose
if not.

    // UV of the vertex. No special space for this one.
    UV = vertexUV;
}

./sun.jpg
./space.jpg
./earth.jpg
./moonyy.jpg
//*# -------------------- ./hw2.cc ---------------------*/
// Include standard headers
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <cmath>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;

// Include GLM
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/string_cast.hpp>
using namespace glm;

#include <common/shader.hpp>
#include <common/texture.hpp>
#include <common/controls.hpp>
#include <common/objloader.hpp>
#include <common/vboindexer.hpp>
#include <common/fps.hpp>

#include <rlib/stdio.hpp>
#include "imported.hpp"
```

```
#include <thread>

int main()
{
	// Initialise GLFW
	if( !glfwInit() )
	{
		fprintf( stderr, "Failed to initialize GLFW\n" );
		getchar();
		return -1;
	}

	glfwWindowHint(GLFW_SAMPLES, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make
MacOS happy; should not be needed
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	// Open a window and create its OpenGL context
	window = glfwCreateWindow( 1024, 768, "Tutorial 08 - Basic Shading",
NULL, NULL);
	if( window == NULL ){
		fprintf( stderr, "Failed to open GLFW window. If you have an Intel GPU,
they are not 3.3 compatible. Try the 2.1 version of the tutorials.\n" );
		getchar();
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);
	R_GL_ERROR_CHECKPOINT;

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

	// Ensure we can capture the escape key being pressed below
	glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
	// Hide the mouse and enable unlimited mouvement
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

	// Set the mouse at the center of the screen
	glfwPollEvents();
	glfwSetCursorPos(window, 1024/2, 768/2);

	// Enable depth test
	glEnable(GL_DEPTH_TEST);
	// Accept fragment if it closer to the camera than the former one
	//glDepthFunc(GL_LESS);
	glDepthFunc(GL_LEQUAL);

	// Cull triangles which normal is not towards the camera
	glEnable(GL_CULL_FACE);

	GLuint VertexArrayID;
	glGenVertexArrays(1, &VertexArrayID);
	glBindVertexArray(VertexArrayID);
	R_GL_ERROR_CHECKPOINT;

	// Create and compile our GLSL program from the shaders
	programID = LoadShaders( "StandardShading.vertexshader", "StandardShading.fragmentshader" );
	R_GL_ERROR_CHECKPOINT;
```

```
    // Get a handle for our "MVP" uniform
    GLuint MatrixID = glGetUniformLocation(programID, "MVP");
    GLuint ViewMatrixID = glGetUniformLocation(programID, "V");
    GLuint ModelMatrixID = glGetUniformLocation(programID, "M");

    // import begin
    R_GL_ERROR_CHECKPOINT;
    generateIDs();
    R_GL_ERROR_CHECKPOINT;
    initVAO();
    R_GL_ERROR_CHECKPOINT;
    float distScale = 35.0 / 149597870.7; // AU in km
    float radScale = 1.0 / 6378.1; // E in km
    // world space
    vec3 sunCenter = vec3(0.0);
    vec3 earthCenter = vec3(0.0);
    vec3 moonCenter = vec3(0.0);

    // make sun
    std::vector<vec3> sunVertices;
    std::vector<vec3> sunNormals;
    std::vector<vec2> sunUvs;
    std::vector<unsigned int> sunIndices;
    sunCenter = vec3(0.0);
    float sunRadius = pow(radScale * 696000.0, 0.5);
    R_GL_ERROR_CHECKPOINT;
    generateSphere(sunVertices, sunNormals, sunUvs, sunIndices, sunCenter,
sunRadius, 96);
    R_GL_ERROR_CHECKPOINT;
    GLuint sun = createTexture("sun.jpg");
    R_GL_ERROR_CHECKPOINT;

    // make earth
    std::vector<vec3> earthVertices;
    std::vector<vec3> earthNormals;
    std::vector<vec2> earthUvs;
    std::vector<unsigned int> earthIndices;
    earthCenter = vec3(distScale * 149597890, 0.0, 0.0);
    float earthRadius = pow(radScale * 6378.1, 0.5);
    generateSphere(earthVertices, earthNormals, earthUvs, earthIndices,
earthCenter, earthRadius, 72);
    GLuint earth = createTexture("earth.jpg");

    rlib::println("Sun.R = ", sunRadius, "earth.R=", earthRadius);

    // make moon
    std::vector<vec3> moonVertices;
    std::vector<vec3> moonNormals;
    std::vector<vec2> moonUvs;
    std::vector<unsigned int> moonIndices;
    moonCenter = earthCenter - vec3((20 * distScale * 384399.0), 0.0, 0.0);
    float moonRadius = pow(radScale * 1737.1 / 2, 0.5);
    generateSphere(moonVertices, moonNormals, moonUvs, moonIndices,
moonCenter, moonRadius, 48);
    GLuint moon = createTexture("moonyy.jpg");

    // make space
    std::vector<vec3> spaceVertices;
    std::vector<vec3> spaceNormals;
    std::vector<vec2> spaceUvs;
    std::vector<unsigned int> spaceIndices;
    vec3 spaceCenter = vec3(0.0);
    generateSphere(spaceVertices, spaceNormals, spaceUvs, spaceIndices,
spaceCenter, 400.0, 128);
    GLuint space = createTexture("space1.png");
    R_GL_ERROR_CHECKPOINT;

    auto debugMsg = [&](){
        rlib::println("DEBUGLINE:                              SunNode_model=",
glm::to_string(sunVertices[0]), ", Earth=", glm::to_string(earthVertices[0]));
	};

	float scale;
	float sunRot;
	float earthOrb;
	float earthRot;
	float moonOrb;
	float moonRot;
	float spaceRot;
	GLuint diffUniformLocation;
	// import end

	// Get a handle for our "LightPosition" uniform
	glUseProgram(programID);
	GLuint		LightID		=		glGetUniformLocation(programID,
"LightPosition_worldspace");
	glfwSetKeyCallback(window, key_callback);

	init_fps();
	do{
		// Clear the screen
		glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

		// Use our shader
		glUseProgram(programID);

		// Compute the MVP matrix from keyboard and mouse input
		computeMatricesFromInputs();
		glm::mat4 ProjectionMatrix = getProjectionMatrix();
		glm::mat4 ViewMatrix = getViewMatrix();
		glm::mat4 ModelMatrix = glm::mat4(1.0);
		glm::mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;

		glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
	glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, &ModelMatrix[0][0]);
	glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, &ViewMatrix[0][0]);
	glm::vec3 lightPos = glm::vec3(0,0,0);
	glUniform3f(LightID, lightPos.x, lightPos.y, lightPos.z);



	scale = universeTimeSpeed * M_PI;
	sunRot = scale / 25.38;
	earthOrb = scale / 365;
	earthRot = -scale;
	moonOrb = scale / 27.32;
	moonRot = scale / 27.32;
	spaceRot = scale / 5000;

	// call function to draw our scene
	rotatePlanet(sunVertices, sunNormals, sunCenter, vec3(0.0, 0.0, 1.0),
sunRot);
	orbitPlanet(earthVertices, earthNormals, earthCenter, sunCenter,
vec3(0.0, 0.0, 1.0), earthOrb);
	rotatePlanet(earthVertices, earthNormals, earthCenter, vec3(0.0, 0.0,
1.0), earthRot);
	orbitPlanet(moonVertices, moonNormals, moonCenter, earthCenter,
vec3(0.0, 0.0, 1.0), moonOrb);
	rotatePlanet(moonVertices, moonNormals, moonCenter, vec3(0.0, 0.0,
1.0), moonRot);
	rotatePlanet(spaceVertices, spaceNormals, spaceCenter, vec3(0.0, 0.0,
1.0), spaceRot);
    R_GL_ERROR_CHECKPOINT;

	loadBuffer(sunVertices, sunNormals, sunUvs, sunIndices);
	loadTexture(sun, GL_TEXTURE0, programID, "myTextureSampler");
	diffUniformLocation = glGetUniformLocation(programID, "diffuse");
	glUniform1i(diffUniformLocation, false); // change this to sunDiffuse or
something in your free time because this is sloppy
```

```
        render(MVP, ViewMatrix, ModelMatrix, 0, sunIndices.size());
    R_GL_ERROR_CHECKPOINT;



        loadBuffer(earthVertices, earthNormals, earthUvs, earthIndices);
        loadTexture(earth, GL_TEXTURE0, programID, "myTextureSampler");
        diffUniformLocation = glGetUniformLocation(programID, "diffuse");
        glUniform1i(diffUniformLocation, true);
        render(MVP, ViewMatrix, ModelMatrix, 0, earthIndices.size());
    R_GL_ERROR_CHECKPOINT;



        loadBuffer(moonVertices, moonNormals, moonUvs, moonIndices);
        loadTexture(moon, GL_TEXTURE0, programID, "myTextureSampler");
        diffUniformLocation = glGetUniformLocation(programID, "diffuse");
        glUniform1i(diffUniformLocation, true);
        render(MVP, ViewMatrix, ModelMatrix, 0, moonIndices.size());



        loadBuffer(spaceVertices, spaceNormals, spaceUvs, spaceIndices);
        loadTexture(space, GL_TEXTURE0, programID, "myTextureSampler");
        diffUniformLocation = glGetUniformLocation(programID, "diffuse");
        glUniform1i(diffUniformLocation, false);
        render(MVP, ViewMatrix, ModelMatrix, 0, spaceIndices.size());

        ++fps_counter;

        // Swap buffers
        glfwSwapBuffers(window);
        glfwPollEvents();

    } // Check if the ESC key was pressed or the window was closed
    while(glfwGetKey(window,   GLFW_KEY_ESCAPE   )   !=   GLFW_PRESS   &&
glfwWindowShouldClose(window) == 0);
        glDisableVertexAttribArray(0);
```

```
            glDisableVertexAttribArray(1);
            glDisableVertexAttribArray(2);



/*
      // Cleanup VBO and shader
      glDeleteBuffers(1, &vertexbuffer);
      glDeleteBuffers(1, &uvbuffer);
      glDeleteBuffers(1, &normalbuffer);
      glDeleteProgram(programID);
      glDeleteTextures(1, &Texture);
      glDeleteVertexArrays(1, &VertexArrayID);
      */
      deleteIDs();

      // Close OpenGL window and terminate GLFW
      glfwTerminate();

      return 0;
}

//*# -------------------- ./StandardShading.fragmentshader ---------------------*/
#version 330 core

// Interpolated values from the vertex shaders
in vec2 UV;
in vec3 Position_worldspace;
in vec3 Normal_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 LightDirection_cameraspace;

// Ouput data
out vec3 color;

// Values that stay constant for the whole mesh.
```

```
uniform sampler2D myTextureSampler;
uniform mat4 MV;
uniform vec3 LightPosition_worldspace;

uniform bool diffuse;

void main(){
    if(!diffuse) {
        // No light processing
        color = texture( myTextureSampler, UV ).rgb;
        return;
    }

    // Light emission properties
    // You probably want to put them as uniforms
    vec3 LightColor = vec3(1,1,1);
    float LightPower = 3000.0f;
    float reflectFactor = 0.01f;

    // Material properties
    vec3 MaterialDiffuseColor = texture( myTextureSampler, 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( LightPosition_worldspace - Position_worldspace );

    // Normal of the computed fragment, in camera space
    vec3 n = normalize( Normal_cameraspace );
    // Direction of the light (from the fragment to the light)
    vec3 l = normalize( LightDirection_cameraspace );
    // Cosine of the angle between the normal and the light direction,
    // clamped above 0
    //   - light is at the vertical of the triangle -> 1
    //   - light is perpendicular to the triangle -> 0
```

```
	//   - light is behind the triangle -> 0
	float cosTheta = clamp( dot( n,l ), 0,1 );

	// Eye vector (towards the camera)
	vec3 E = normalize(EyeDirection_cameraspace);
	// Direction in which the triangle reflects the light
	vec3 R = reflect(-l,n);
	// Cosine of the angle between the Eye vector and the Reflect vector,
	// clamped to 0
	//   - Looking into the reflection -> 1
	//   - Looking elsewhere -> < 1
	float cosAlpha = clamp( dot( E,R ), 0,1 );

	color =
		// Ambient : simulates indirect lighting
		MaterialAmbientColor +
		// Diffuse : "color" of the object
		MaterialDiffuseColor   *   LightColor   *   LightPower   *   cosTheta   /
(distance*distance) +
		// Specular : reflective highlight, like a mirror
		reflectFactor  *  MaterialSpecularColor  *  LightColor  *  LightPower  *
pow(cosAlpha,5) / (distance*distance);
}//*# -------------------- ./imported.hpp ---------------------*/
#ifndef RLIB_GL_IMPORTED_HPP_
#define RLIB_GL_IMPORTED_HPP_ 1

// Include standard headers
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <cmath>
#define STB_IMAGE_IMPLEMENTATION
#include <common/stb/stb_image.h>
#include <rlib/macro.hpp>
```

```
// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>

// Include GLM
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
using namespace glm;

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
using std::vector;

//vec2 and vec3 are part of the glm math library.
//Include in your own project by putting the glm directory in your project,
//and including glm/glm.hpp as I have at the top of the file.
//"using namespace glm;" will allow you to avoid writing everyting as glm::vec2
std::vector<vec3> points;
std::vector<vec2> uvs;

namespace VAO {
	enum {GEOMETRY=0, COUNT};		//Enumeration assigns each name
a value going up
	//LINES=0, COUNT=1
}

namespace VBO {
	enum {POINTS=0, NORMALS, UVS, INDICES, COUNT};	//POINTS=0,
COLOR=1, COUNT=2
};
```

```
GLuint vbo [VBO::COUNT];        //Array which stores OpenGL's vertex buffer
object handles
GLuint vao [VAO::COUNT];        //Array which stores Vertex Array Object
handles
GLuint programID;

#ifdef DEBUG
#define R_GL_ERROR_CHECKPOINT CheckGLErrors("GL Error at " __FILE__ ":"
RLIB_MACRO_TO_CSTR(__LINE__))
#else
#define R_GL_ERROR_CHECKPOINT
#endif
bool CheckGLErrors(string location)
{
    bool error = false;
    for (GLenum flag = glGetError(); flag != GL_NO_ERROR; flag = glGetError())
    {
        cout << "OpenGL ERROR:   ";
        switch (flag) {
        case GL_INVALID_ENUM:
            cout << location << ": " << "GL_INVALID_ENUM" << endl; break;
        case GL_INVALID_VALUE:
            cout << location << ": " << "GL_INVALID_VALUE" << endl; break;
        case GL_INVALID_OPERATION:
            cout << location << ": " << "GL_INVALID_OPERATION" << endl;
break;
        case GL_INVALID_FRAMEBUFFER_OPERATION:
            cout << location << ": " <<
"GL_INVALID_FRAMEBUFFER_OPERATION" << endl; break;
        case GL_OUT_OF_MEMORY:
            cout << location << ": " << "GL_OUT_OF_MEMORY" << endl;
break;
        default:
            cout << "[unknown error code]" << endl;
        }
        error = true;
    }
    if(error)
        throw std::runtime_error("CheckGLErrors failed. See log.");
    return true;
}



//Describe the setup of the Vertex Array Object
void initVAO()
{
    glBindVertexArray(vao[VAO::GEOMETRY]);      //Set the active Vertex
Array
    R_GL_ERROR_CHECKPOINT;

    glEnableVertexAttribArray(0);       //Tell opengl you're using layout
attribute 0 (For shader input)
    R_GL_ERROR_CHECKPOINT;
    glBindBuffer( GL_ARRAY_BUFFER, vbo[VBO::POINTS] );      //Set the active
Vertex Buffer
    R_GL_ERROR_CHECKPOINT;
    glVertexAttribPointer(
        0,                  //Attribute
        3,                  //Size # Components
        GL_FLOAT,   //Type
        GL_FALSE,   //Normalized?
        sizeof(vec3),   //Stride
        (void*)0                //Offset
        );
    R_GL_ERROR_CHECKPOINT;

    glEnableVertexAttribArray(1);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[VBO::NORMALS]);
    glVertexAttribPointer(
        1,              //Attribute
        3,              //Size # Components
        GL_FLOAT,   //Type
        GL_FALSE,   //Normalized?
        sizeof(vec3),   //Stride
        (void*)0            //Offset
        );

    glEnableVertexAttribArray(2);       //Tell opengl you're using layout
attribute 1
    glBindBuffer(GL_ARRAY_BUFFER, vbo[VBO::UVS]);
    glVertexAttribPointer(
        2,
        2,
        GL_FLOAT,
        GL_FALSE,
        sizeof(vec2),
        (void*)0
        );

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[VBO::INDICES]);
    R_GL_ERROR_CHECKPOINT;
}
//Gets handles from OpenGL
void generateIDs()
{
    glGenVertexArrays(VAO::COUNT, vao);     //Tells OpenGL to create
VAO::COUNT many
                                            // Vertex Array Objects,
and store their
                                            // handles in vao array
    glGenBuffers(VBO::COUNT, vbo);      //Tells OpenGL to create
VBO::COUNT many
                                        //Vertex Buffer Objects and
store their
```

```
                                                    //handles in vbo array
}
//Clean up IDs when you're done using them
void deleteIDs()
{
    glDeleteProgram(programID);

    glDeleteVertexArrays(VAO::COUNT, vao);
    glDeleteBuffers(VBO::COUNT, vbo);
}



//Loads buffers with data
void loadBuffer(const vector<vec3>& points, const vector<vec3> normals,
                const   vector<vec2>&   uvs,   const   vector<unsigned   int>&
indices)
{
    R_GL_ERROR_CHECKPOINT;
    glBindBuffer(GL_ARRAY_BUFFER, vbo[VBO::POINTS]);
    R_GL_ERROR_CHECKPOINT;
    glBufferData(
        GL_ARRAY_BUFFER,                //Which buffer you're loading too
        sizeof(vec3)*points.size(),     //Size of data in array (in bytes)
        &points[0],                         //Start of array (&points[0] will give
you pointer to start of vector)
        GL_DYNAMIC_DRAW                         //GL_DYNAMIC_DRAW  if  you're
changing the data often
                                                //GL_STATIC_DRAW    if    you're
changing seldomly
        );

    R_GL_ERROR_CHECKPOINT;
    glBindBuffer(GL_ARRAY_BUFFER, vbo[VBO::NORMALS]);
    R_GL_ERROR_CHECKPOINT;
    glBufferData(
        GL_ARRAY_BUFFER,                //Which buffer you're loading too
        sizeof(vec3)*normals.size(),//Size of data in array (in bytes)
        &normals[0],                            //Start of array (&points[0] will
give you pointer to start of vector)
        GL_DYNAMIC_DRAW                         //GL_DYNAMIC_DRAW if you're
changing the data often
                                                //GL_STATIC_DRAW if you're
changing seldomly
        );

    glBindBuffer(GL_ARRAY_BUFFER, vbo[VBO::UVS]);
    glBufferData(
        GL_ARRAY_BUFFER,
        sizeof(vec2)*uvs.size(),
        &uvs[0],
        GL_STATIC_DRAW
        );

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[VBO::INDICES]);
    glBufferData(
        GL_ELEMENT_ARRAY_BUFFER,
        sizeof(unsigned int)*indices.size(),
        &indices[0],
        GL_STATIC_DRAW
        );

    R_GL_ERROR_CHECKPOINT;
}

//For reference:
//    https://open.gl/textures
GLuint createTexture(const char* filename)
{
    int components;
    GLuint texID;
    int tWidth, tHeight;

    //stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load(filename, &tWidth, &tHeight,
&components, 0);

    if(data != NULL)
    {
        glGenTextures(1, &texID);
        glBindTexture(GL_TEXTURE_2D, texID);

        if(components==3)
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tWidth, tHeight, 0,
GL_RGB, GL_UNSIGNED_BYTE, data);
        else if(components==4)
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tWidth, tHeight, 0,
GL_RGBA, GL_UNSIGNED_BYTE, data);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);

        //Clean up
        glBindTexture(GL_TEXTURE_2D, 0);
        stbi_image_free(data);

        return texID;
    }

    return 0;    //Error
}
```

```
//Use program before loading texture
//    texUnit can be - GL_TEXTURE0, GL_TEXTURE1, etc...
void loadTexture(GLuint texID, GLuint texUnit, GLuint program, const char*
uniformName)
{
    glActiveTexture(texUnit);
    glBindTexture(GL_TEXTURE_2D, texID);

    GLuint uniformLocation = glGetUniformLocation(program, uniformName);
    glUniform1i(uniformLocation, 0);

    R_GL_ERROR_CHECKPOINT;
}

// fun fact: did you know planets are just elaborate spheres? Believe it.
void generateSphere(vector<vec3>& positions, vector<vec3>& normals,
                    vector<vec2>& uvs, vector<unsigned int>& indices,
                    vec3 center, float radius, int divisions)
{
    float step = 1.f / (float)(divisions - 1);
    float u = 0.f;

    // Traversing the planes of time and space
    for (int i = 0; i < divisions; i++) {
        float v = 0.f;

        //Traversing the planes of time and space (again)
        for (int j = 0; j < divisions; j++) {
            vec3 pos = vec3(    radius * cos(2.f * M_PI * u) * sin(M_PI * v),
                                radius * sin(2.f * M_PI * u) * sin(M_PI * v),
                                radius * cos(M_PI * v)) + center;

            vec3 normal = normalize(pos - center);
            positions.push_back(pos);
            normals.push_back(normal);
            uvs.push_back(vec2(u, v));

            v += step;
        }

        u += step;
    }

    for(int i = 0; i < divisions - 1; i++)
    {
        for(int j = 0; j < divisions - 1; j++)
        {
            unsigned int p00 = i * divisions + j;
            unsigned int p01 = i * divisions + j + 1;
            unsigned int p10 = (i + 1) * divisions + j;
            unsigned int p11 = (i + 1) * divisions + j + 1;

            indices.push_back(p00);
            indices.push_back(p10);
            indices.push_back(p01);

            indices.push_back(p01);
            indices.push_back(p10);
            indices.push_back(p11);
        }
    }
}

//Draws buffers to screen
void render(mat4 MVP, mat4 viewMatrix, mat4 modelMatrix, int startElement,
int numElements)
{
    //Don't need to call these on every draw, so long as they don't change
```

```
	glBindVertexArray(vao[VAO::GEOMETRY]);		//Use the LINES vertex array
	glUseProgram(programID);

	//mat4 MVP = projectionMatrix * viewMatrix * modelMatrix;

	glUniformMatrix4fv(glGetUniformLocation(programID, "MVP"),
						1,
						false,
						&MVP[0][0]);

	glUniformMatrix4fv(glGetUniformLocation(programID, "V"),
						1,
						false,
						&viewMatrix[0][0]);

	glUniformMatrix4fv(glGetUniformLocation(programID, "M"),
						1,
						false,
						&modelMatrix[0][0]);



	CheckGLErrors("loadUniforms");

	glDrawElements(
			GL_TRIANGLES,		//What shape we're drawing	- GL_TRIANGLES, GL_LINES, GL_POINTS, GL_QUADS, GL_TRIANGLE_STRIP
			numElements,		//How many indices
			GL_UNSIGNED_INT,	//Type
			(void*)0			//Offset
			);

	CheckGLErrors("render");
}
```

```
void rotatePlanet(vector<vec3>& points, vector<vec3>& normals, vec3 center, vec3 axis, float theta) {
	axis = normalize(axis);
	float x = axis.x;
	float y = axis.y;
	float z = axis.z;
	float x2 = x * x;
	float y2 = y * y;
	float z2 = z * z;

	mat3 rMat = mat3( cos(theta) + x2 * (1 - cos(theta)), x * y * (1 - cos(theta)) - z * sin(theta), x * z * (1 - cos(theta)) + y * sin(theta),
	                  y * x * (1 - cos(theta)) + z * sin(theta), cos(theta) + y2 * (1 - cos(theta)), y * z * (1 - cos(theta)) - x * sin(theta),
	                  z * x * (1 - cos(theta)) - y * sin(theta), z * y * (1 - cos(theta)) + x * sin(theta), cos(theta) + z2 * (1 - cos(theta)));

	for (int i = 0; i < points.size(); i++) {
		points[i] = (rMat * (points[i] - center)) + center;
		normals[i] = normalize(points[i] - center);
	}

}

void orbitPlanet(vector<vec3>& points, vector<vec3>& normals, vec3& childCenter, vec3 parentCenter, vec3 axis, float theta) {
	axis = normalize(axis);
	float x = axis.x;
	float y = axis.y;
	float z = axis.z;
	float x2 = x * x;
	float y2 = y * y;
	float z2 = z * z;

	rotatePlanet(points, normals, childCenter, axis, -theta);
	mat3 rMat = mat3( cos(theta) + x2 * (1 - cos(theta)), x * y * (1 - cos(theta)) - z * sin(theta), x * z * (1 - cos(theta)) + y * sin(theta),
						y * x * (1 - cos(theta)) + z * sin(theta), cos(theta) + y2 * (1 - cos(theta)), y * z * (1 - cos(theta)) - x * sin(theta),
						z * x * (1 - cos(theta)) - y * sin(theta), z * y * (1 - cos(theta)) + x * sin(theta), cos(theta) + z2 * (1 - cos(theta)));

	childCenter = (rMat * (childCenter - parentCenter)) + parentCenter;

	for (int i = 0; i < points.size(); i++) {
		points[i] = (rMat * (points[i] - parentCenter)) + parentCenter;
		normals[i] = normalize(points[i] - childCenter);
	}

}

#endif
```

2.3 心得与体会

在实验过程中，尽管过程中任由许多不会的地方，而且有待于今后的提高和改进，但我加深了对书本上知识的理解与掌握，同时也学到了很多书本上没有东西，并积累了一些宝贵的经验，这对我以后的学习与工作是不无裨益的。

我最初学图形相关的东西时，所有东西都只是当作黑盒，只知道这样算能做到什么事情。图形学有一个好处是，很容易看到结果，甚至是能实时互动地检查结果。后来我才慢慢了解到相关的数学，就好像掀开了黑盒看到里面，慢慢知其所以然。如果有足够的时间，不是赶着考试或面试，这种漫长的过程其实也没有太多问题。

另外，对一个事情的了解可能随着学习更多知识，以至于在更多应用当中，会逐步更深入了解个中的方方面面。例如学习了抽象代数，就会把矩阵和其乘法考虑成非交换群，以至

于可以考虑不用矩阵而是其他更局限的群来表示几何变换，也了解到旋转矩阵的冗余及非正交等问题。之后还可用几何代数等方法更直观地表示一些几何关系。