Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное

Образовательное учреждение высшего образования

«Уфимский университет науки и технологий»

Факультет информатики и робототехники

Кафедра Информатики

Отчет по лабораторной работе № 2

на тему: «Преобразования трёхмерных объектов»

Выполнил:

Студент группы ПРО-233Б Ермолов И О

Проверил:

канд. техн. наук В.А. Котельников

Уфа - 2023

**Цель работы**

Изучение преобразования для трехмерных объектов.

**Задание**

Выполнить 6-8, 12, 13 уроки по OpenGL <https://triplepointfive.github.io/ogltutor/>

**Ход работы**

1. Преобразования трехмерных объектов. В ходе лабораторной работы я изучил способ изменения положения, вращения и изменения размера объекта, заключающийся в задании матриц.

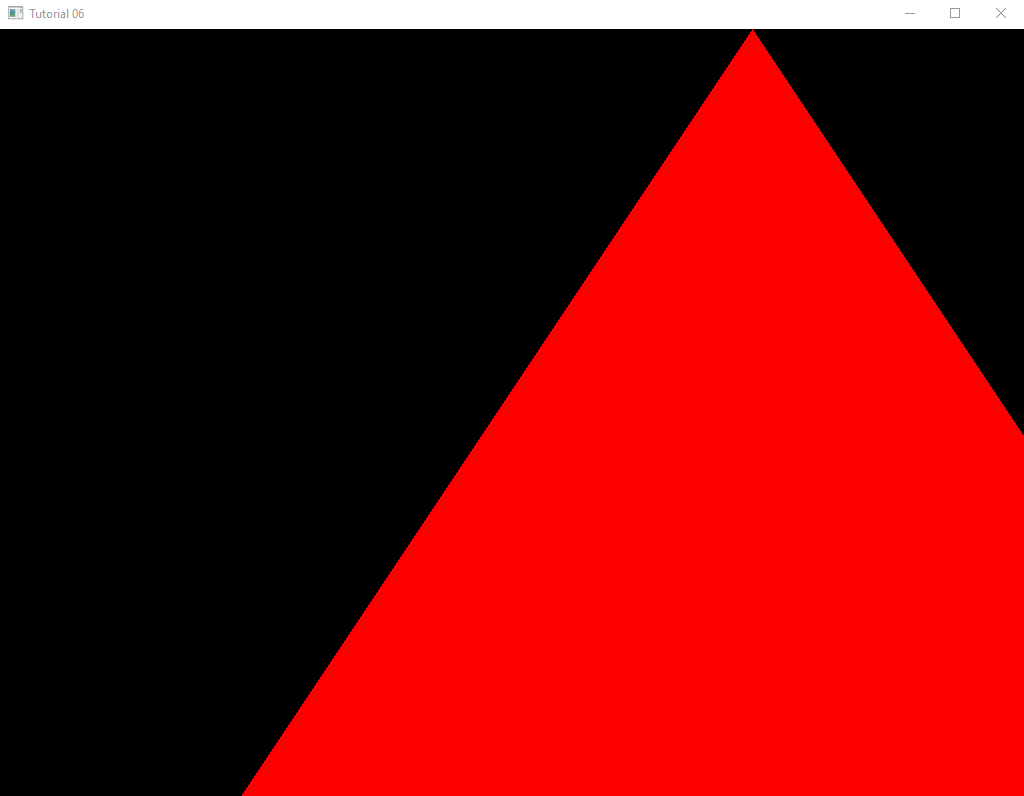


Рисунок 1. Перемещение объекта по оси Х.

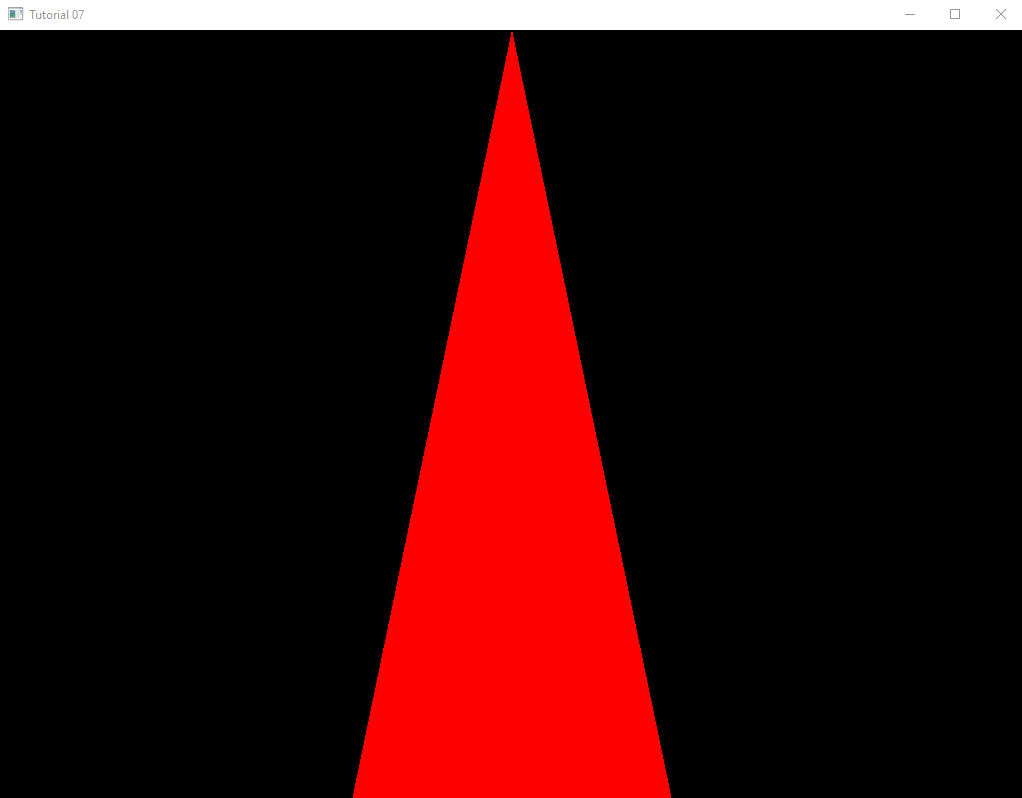


Рисунок 2. Вращение объекта по оси Y.

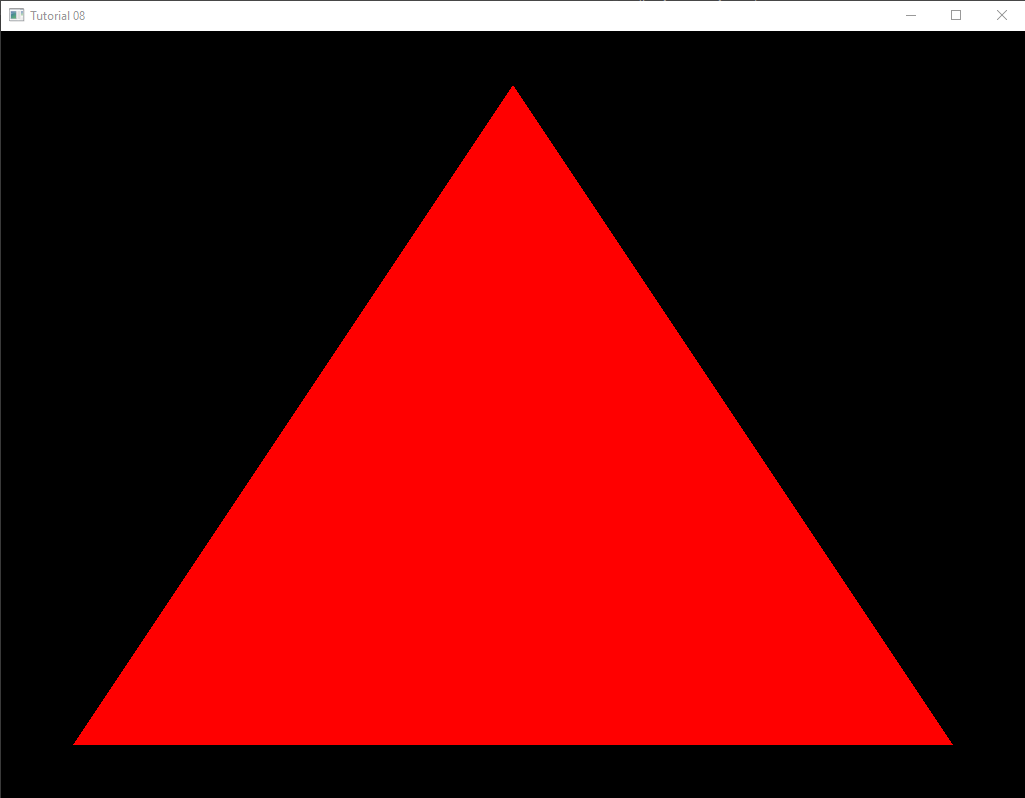


Рисунок 3. Изменение размера объекта.

1. Помимо этого, в ходе работы мною были изучены проекции перспективы и пространство камеры. Я написал программу, эмулирующую вращение и перемещение объекта посредством преобразования свойств камеры.

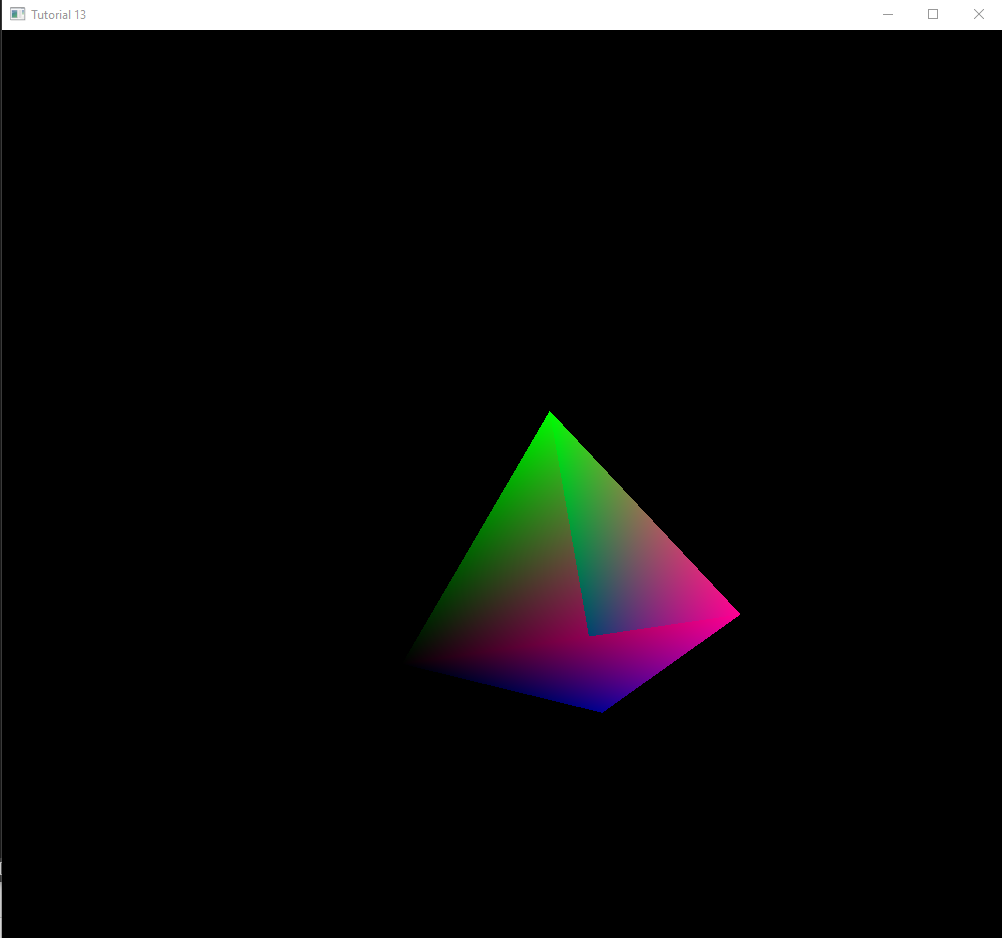


Рисунок 4. Перемещение объекта по оси Х, вращение по всем осям, FOV - 120

**Выводы к работе**

При выполнении данной лабораторной работы я познакомился с преобразованием трёхмерных объектов и изменением свойств камеры. Полный код программ по ссылке: [https://github.com/Ivan122727/EKG/tree/master/LW2](https://github.com/VerVyle/LW_ECG/tree/master/LW2)

**Приложение 1**

**math\_3d.cpp**

#include "math\_3d.h"

Vector3f Vector3f::Cross(const Vector3f& v) const

{

const float \_x = y \* v.z - z \* v.y;

const float \_y = z \* v.x - x \* v.z;

const float \_z = x \* v.y - y \* v.x;

return Vector3f(\_x, \_y, \_z);

}

Vector3f& Vector3f::Normalize()

{

const float Length = sqrtf(x \* x + y \* y + z \* z);

x /= Length;

y /= Length;

z /= Length;

return \*this;

}

void Matrix4f::InitScaleTransform(float ScaleX, float ScaleY, float ScaleZ)

{

m[0][0] = ScaleX; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0f;

m[1][0] = 0.0f; m[1][1] = ScaleY; m[1][2] = 0.0f; m[1][3] = 0.0f;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = ScaleZ; m[2][3] = 0.0f;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitRotateTransform(float RotateX, float RotateY, float RotateZ)

{

Matrix4f rx, ry, rz;

const float x = ToRadian(RotateX);

const float y = ToRadian(RotateY);

const float z = ToRadian(RotateZ);

rx.m[0][0] = 1.0f; rx.m[0][1] = 0.0f; rx.m[0][2] = 0.0f; rx.m[0][3] = 0.0f;

rx.m[1][0] = 0.0f; rx.m[1][1] = cosf(x); rx.m[1][2] = -sinf(x); rx.m[1][3] = 0.0f;

rx.m[2][0] = 0.0f; rx.m[2][1] = sinf(x); rx.m[2][2] = cosf(x); rx.m[2][3] = 0.0f;

rx.m[3][0] = 0.0f; rx.m[3][1] = 0.0f; rx.m[3][2] = 0.0f; rx.m[3][3] = 1.0f;

ry.m[0][0] = cosf(y); ry.m[0][1] = 0.0f; ry.m[0][2] = -sinf(y); ry.m[0][3] = 0.0f;

ry.m[1][0] = 0.0f; ry.m[1][1] = 1.0f; ry.m[1][2] = 0.0f; ry.m[1][3] = 0.0f;

ry.m[2][0] = sinf(y); ry.m[2][1] = 0.0f; ry.m[2][2] = cosf(y); ry.m[2][3] = 0.0f;

ry.m[3][0] = 0.0f; ry.m[3][1] = 0.0f; ry.m[3][2] = 0.0f; ry.m[3][3] = 1.0f;

rz.m[0][0] = cosf(z); rz.m[0][1] = -sinf(z); rz.m[0][2] = 0.0f; rz.m[0][3] = 0.0f;

rz.m[1][0] = sinf(z); rz.m[1][1] = cosf(z); rz.m[1][2] = 0.0f; rz.m[1][3] = 0.0f;

rz.m[2][0] = 0.0f; rz.m[2][1] = 0.0f; rz.m[2][2] = 1.0f; rz.m[2][3] = 0.0f;

rz.m[3][0] = 0.0f; rz.m[3][1] = 0.0f; rz.m[3][2] = 0.0f; rz.m[3][3] = 1.0f;

\*this = rz \* ry \* rx;

}

void Matrix4f::InitTranslationTransform(float x, float y, float z)

{

m[0][0] = 1.0f; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = x;

m[1][0] = 0.0f; m[1][1] = 1.0f; m[1][2] = 0.0f; m[1][3] = y;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = 1.0f; m[2][3] = z;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitCameraTransform(const Vector3f& Target, const Vector3f& Up)

{

Vector3f N = Target;

N.Normalize();

Vector3f U = Up;

U.Normalize();

U = U.Cross(N);

Vector3f V = N.Cross(U);

m[0][0] = U.x; m[0][1] = U.y; m[0][2] = U.z; m[0][3] = 0.0f;

m[1][0] = V.x; m[1][1] = V.y; m[1][2] = V.z; m[1][3] = 0.0f;

m[2][0] = N.x; m[2][1] = N.y; m[2][2] = N.z; m[2][3] = 0.0f;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 0.0f; m[3][3] = 1.0f;

}

void Matrix4f::InitPersProjTransform(float FOV, float Width, float Height, float zNear, float zFar)

{

const float ar = Width / Height;

const float zRange = zNear - zFar;

const float tanHalfFOV = tanf(ToRadian(FOV / 2.0f));

m[0][0] = 1.0f / (tanHalfFOV \* ar); m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0;

m[1][0] = 0.0f; m[1][1] = 1.0f / tanHalfFOV; m[1][2] = 0.0f; m[1][3] = 0.0;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = (-zNear - zFar) / zRange; m[2][3] = 2.0f \* zFar \* zNear / zRange;

m[3][0] = 0.0f; m[3][1] = 0.0f; m[3][2] = 1.0f; m[3][3] = 0.0;

}

**task6.cpp**

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "math\_3d.h"

#include "tasks.h"

GLuint VBO6;

GLuint gWorldLocation6;

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

\n\

uniform mat4 gWorld; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWorld \* vec4(Position, 1.0); \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

out vec4 FragColor; \n\

\n\

void main() \n\

{ \n\

FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n\

}";

static void RenderSceneCB() {

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.001f;

Matrix4f World;

World.m[0][0] = 1.0f; World.m[0][1] = 0.0f; World.m[0][2] = 0.0f; World.m[0][3] = sinf(Scale);

World.m[1][0] = 0.0f; World.m[1][1] = 1.0f; World.m[1][2] = 0.0f; World.m[1][3] = 0.0f;

World.m[2][0] = 0.0f; World.m[2][1] = 0.0f; World.m[2][2] = 1.0f; World.m[2][3] = 0.0f;

World.m[3][0] = 0.0f; World.m[3][1] = 0.0f; World.m[3][2] = 0.0f; World.m[3][3] = 1.0f;

// переменная матрица, количество матриц, row-major rule, первый элемент

glUniformMatrix4fv(gWorldLocation6, 1, 1, &World.m[0][0]);

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO6);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glDrawArrays(GL\_TRIANGLES, 0, 3);

glDisableVertexAttribArray(0);

glutSwapBuffers();

}

static void InitializeGlutCallbacks() {

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(RenderSceneCB);

}

static void CreateVertexBuffer() {

Vector3f Vertices[3];

Vertices[0] = Vector3f(-1.0f, -1.0f, 0.0f);

Vertices[1] = Vector3f(1.0f, -1.0f, 0.0f);

Vertices[2] = Vector3f(0.0f, 1.0f, 0.0f);

glGenBuffers(1, &VBO6);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO6);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

static void AddShader(GLuint ShaderProgram, const char\* pShaderText, GLenum ShaderType) {

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

exit(0);

}

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success) {

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

exit(1);

}

glAttachShader(ShaderProgram, ShaderObj);

}

static void CompileShaders() {

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

fprintf(stderr, "Error creating shader program\n");

exit(1);

}

AddShader(ShaderProgram, pVS, GL\_VERTEX\_SHADER);

AddShader(ShaderProgram, pFS, GL\_FRAGMENT\_SHADER);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

glLinkProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_LINK\_STATUS, &Success);

if (Success == 0) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

exit(1);

}

glValidateProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_VALIDATE\_STATUS, &Success);

if (!Success) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

exit(1);

}

glUseProgram(ShaderProgram);

gWorldLocation6 = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation6 != 0xFFFFFFFF);

}

void invoke6(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(1024, 768);

glutInitWindowPosition(100, 100);

glutCreateWindow("Tutorial 06");

InitializeGlutCallbacks();

// Must be done after glut is initialized!

GLenum res = glewInit();

if (res != GLEW\_OK) {

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

CreateVertexBuffer();

CompileShaders();

glutMainLoop();

}

**task7.cpp**

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "math\_3d.h"

#include "tasks.h"

GLuint VBO7;

GLuint gWorldLocation7;

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

\n\

uniform mat4 gWorld; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWorld \* vec4(Position, 1.0); \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

out vec4 FragColor; \n\

\n\

void main() \n\

{ \n\

FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n\

}";

static void RenderSceneCB() {

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.01f;

Matrix4f World;

//z

/\*World.m[0][0] = cosf(Scale); World.m[0][1] = -sinf(Scale); World.m[0][2] = 0.0f; World.m[0][3] = 0.0f;

World.m[1][0] = sinf(Scale); World.m[1][1] = cosf(Scale); World.m[1][2] = 0.0f; World.m[1][3] = 0.0f;

World.m[2][0] = 0.0f; World.m[2][1] = 0.0f; World.m[2][2] = 1.0f; World.m[2][3] = 0.0f;

World.m[3][0] = 0.0f; World.m[3][1] = 0.0f; World.m[3][2] = 0.0f; World.m[3][3] = 1.0f;\*/

//y

World.m[0][0] = cosf(Scale); World.m[0][1] = 0.0f; World.m[0][2] = -sinf(Scale); World.m[0][3] = 0.0f;

World.m[1][0] = 0.0f; World.m[1][1] = 1.0f; World.m[1][2] = 0.0f; World.m[1][3] = 0.0f;

World.m[2][0] = sinf(Scale); World.m[2][1] = 0.0f; World.m[2][2] = cosf(Scale); World.m[2][3] = 0.0f;

World.m[3][0] = 0.0f; World.m[3][1] = 0.0f; World.m[3][2] = 0.0f; World.m[3][3] = 1.0f;

//x

/\*World.m[0][0] = 1.0f; World.m[0][1] = 0.0f; World.m[0][2] = 0.0f; World.m[0][3] = 0.0f;

World.m[1][0] = 0.0f; World.m[1][1] = cosf(Scale); World.m[1][2] = -sinf(Scale); World.m[1][3] = 0.0f;

World.m[2][0] = 0.0f; World.m[2][1] = sinf(Scale); World.m[2][2] = cosf(Scale); World.m[2][3] = 0.0f;

World.m[3][0] = 0.0f; World.m[3][1] = 0.0f; World.m[3][2] = 0.0f; World.m[3][3] = 1.0f;\*/

glUniformMatrix4fv(gWorldLocation7, 1, GL\_TRUE, &World.m[0][0]);

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO7);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glDrawArrays(GL\_TRIANGLES, 0, 3);

glDisableVertexAttribArray(0);

glutPostRedisplay(); // или glutIdleFunc(RenderSceneCB)

glutSwapBuffers();

}

static void InitializeGlutCallbacks() {

glutDisplayFunc(RenderSceneCB);

//glutIdleFunc(RenderSceneCB); // или постредисплей вызывать

}

static void CreateVertexBuffer() {

Vector3f Vertices[3];

Vertices[0] = Vector3f(-1.0f, -1.0f, 0.0f);

Vertices[1] = Vector3f(1.0f, -1.0f, 0.0f);

Vertices[2] = Vector3f(0.0f, 1.0f, 0.0f);

glGenBuffers(1, &VBO7);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO7);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

static void AddShader(GLuint ShaderProgram, const char\* pShaderText, GLenum ShaderType) {

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

exit(0);

}

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success) {

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

exit(1);

}

glAttachShader(ShaderProgram, ShaderObj);

}

static void CompileShaders() {

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

fprintf(stderr, "Error creating shader program\n");

exit(1);

}

AddShader(ShaderProgram, pVS, GL\_VERTEX\_SHADER);

AddShader(ShaderProgram, pFS, GL\_FRAGMENT\_SHADER);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

glLinkProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_LINK\_STATUS, &Success);

if (Success == 0) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

exit(1);

}

glValidateProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_VALIDATE\_STATUS, &Success);

if (!Success) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

exit(1);

}

glUseProgram(ShaderProgram);

gWorldLocation7 = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation7 != 0xFFFFFFFF);

}

void invoke7(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(1024, 768);

glutInitWindowPosition(100, 100);

glutCreateWindow("Tutorial 07");

InitializeGlutCallbacks();

// Must be done after glut is initialized!

GLenum res = glewInit();

if (res != GLEW\_OK) {

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

CreateVertexBuffer();

CompileShaders();

glutMainLoop();

}

**task8.cpp**

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "math\_3d.h"

#include "tasks.h"

GLuint VBO8;

GLuint gWorldLocation8;

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

\n\

uniform mat4 gWorld; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWorld \* vec4(Position, 1.0); \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

out vec4 FragColor; \n\

\n\

void main() \n\

{ \n\

FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n\

}";

static void RenderSceneCB() {

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.01f;

if (Scale >= 1.0f) {

Scale = 0.0f;

}

Matrix4f World;

Matrix4f Translation;

Translation.m[0][0] = 1.0f; Translation.m[0][1] = 0.0f; Translation.m[0][2] = 0.0f; Translation.m[0][3] = sinf(Scale);

Translation.m[1][0] = 0.0f; Translation.m[1][1] = 1.0f; Translation.m[1][2] = 0.0f; Translation.m[1][3] = 0.0f;

Translation.m[2][0] = 0.0f; Translation.m[2][1] = 0.0f; Translation.m[2][2] = 1.0f; Translation.m[2][3] = 0.0f;

Translation.m[3][0] = 0.0f; Translation.m[3][1] = 0.0f; Translation.m[3][2] = 0.0f; Translation.m[3][3] = 1.0f;

Matrix4f Scaling;

Scaling.m[0][0] = Scale; Scaling.m[0][1] = 0.0f; Scaling.m[0][2] = 0.0f; Scaling.m[0][3] = 0.0f;

Scaling.m[1][0] = 0.0f; Scaling.m[1][1] = Scale; Scaling.m[1][2] = 0.0f; Scaling.m[1][3] = 0.0f;

Scaling.m[2][0] = 0.0f; Scaling.m[2][1] = 0.0f; Scaling.m[2][2] = Scale; Scaling.m[2][3] = 0.0f;

Scaling.m[3][0] = 0.0f; Scaling.m[3][1] = 0.0f; Scaling.m[3][2] = 0.0f; Scaling.m[3][3] = 1.0f;

//World = Translation \* Scaling;

World = Scaling;

glUniformMatrix4fv(gWorldLocation8, 1, GL\_TRUE, &World.m[0][0]);

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO8);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glDrawArrays(GL\_TRIANGLES, 0, 3);

glDisableVertexAttribArray(0);

glutSwapBuffers();

}

static void InitializeGlutCallbacks() {

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(RenderSceneCB);

}

static void CreateVertexBuffer() {

Vector3f Vertices[3];

Vertices[0] = Vector3f(-1.0f, -1.0f, 0.0f);

Vertices[1] = Vector3f(0.0f, 1.0f, 0.0f);

Vertices[2] = Vector3f(1.0f, -1.0f, 0.0f);

glGenBuffers(1, &VBO8);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO8);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

static void AddShader(GLuint ShaderProgram, const char\* pShaderText, GLenum ShaderType) {

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

exit(0);

}

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success) {

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

exit(1);

}

glAttachShader(ShaderProgram, ShaderObj);

}

static void CompileShaders() {

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

fprintf(stderr, "Error creating shader program\n");

exit(1);

}

AddShader(ShaderProgram, pVS, GL\_VERTEX\_SHADER);

AddShader(ShaderProgram, pFS, GL\_FRAGMENT\_SHADER);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

glLinkProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_LINK\_STATUS, &Success);

if (Success == 0) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

exit(1);

}

glValidateProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_VALIDATE\_STATUS, &Success);

if (!Success) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

exit(1);

}

glUseProgram(ShaderProgram);

gWorldLocation8 = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation8 != 0xFFFFFFFF);

}

void invoke8(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(1024, 768);

glutInitWindowPosition(100, 100);

glutCreateWindow("Tutorial 08");

InitializeGlutCallbacks();

// Must be done after glut is initialized!

GLenum res = glewInit();

if (res != GLEW\_OK) {

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

CreateVertexBuffer();

CompileShaders();

glutMainLoop();

}

**task12.cpp**

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "pipeline.h"

#include "tasks.h"

#define WINDOW\_WIDTH 1024

#define WINDOW\_HEIGHT 768

GLuint VBO12;

GLuint IBO12;

GLuint gWorldLocation12;

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

\n\

uniform mat4 gWorld; \n\

\n\

out vec4 Color; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWorld \* vec4(Position, 1.0); \n\

Color = vec4(clamp(Position, 0.0, 1.0), 1.0); \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

in vec4 Color; \n\

\n\

out vec4 FragColor; \n\

\n\

void main() \n\

{ \n\

FragColor = Color; \n\

}";

static void RenderSceneCB() {

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.1f;

Pipeline p;

p.Rotate(Scale,0.0f, 0.0f);

p.WorldPos(0.0f, 0.0f, 5.0f);

p.SetPerspectiveProj(40.0f, WINDOW\_WIDTH, WINDOW\_HEIGHT, 1.0f, 100.0f);

glUniformMatrix4fv(gWorldLocation12, 1, GL\_TRUE, (const GLfloat\*)p.GetTrans());

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO12);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO12);

glDrawElements(GL\_TRIANGLES, 12, GL\_UNSIGNED\_INT, 0);

glDisableVertexAttribArray(0);

glutSwapBuffers();

}

static void InitializeGlutCallbacks() {

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(RenderSceneCB);

}

static void CreateVertexBuffer() {

Vector3f Vertices[4];

Vertices[0] = Vector3f(-1.0f, -1.0f, 0.5773f);

Vertices[1] = Vector3f(0.0f, -1.0f, -1.15475);

Vertices[2] = Vector3f(1.0f, -1.0f, 0.5773f);

Vertices[3] = Vector3f(0.0f, 1.0f, 0.0f);

glGenBuffers(1, &VBO12);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO12);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

static void CreateIndexBuffer() {

unsigned int Indices[] = { 0, 3, 1,

1, 3, 2,

2, 3, 0,

0, 2, 1 };

glGenBuffers(1, &IBO12);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO12);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(Indices), Indices, GL\_STATIC\_DRAW);

}

static void AddShader(GLuint ShaderProgram, const char\* pShaderText, GLenum ShaderType) {

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

exit(0);

}

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success) {

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

exit(1);

}

glAttachShader(ShaderProgram, ShaderObj);

}

static void CompileShaders() {

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

fprintf(stderr, "Error creating shader program\n");

exit(1);

}

AddShader(ShaderProgram, pVS, GL\_VERTEX\_SHADER);

AddShader(ShaderProgram, pFS, GL\_FRAGMENT\_SHADER);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

glLinkProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_LINK\_STATUS, &Success);

if (Success == 0) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

exit(1);

}

glValidateProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_VALIDATE\_STATUS, &Success);

if (!Success) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

exit(1);

}

glUseProgram(ShaderProgram);

gWorldLocation12 = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation12 != 0xFFFFFFFF);

}

void invoke12(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

glutInitWindowPosition(100, 100);

glutCreateWindow("Tutorial 12");

InitializeGlutCallbacks();

// Must be done after glut is initialized!

GLenum res = glewInit();

if (res != GLEW\_OK) {

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

CreateVertexBuffer();

CreateIndexBuffer();

CompileShaders();

glutMainLoop();

}

**task13.cpp**

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "math\_3d.h"

#include "pipeline.h"

#include "tasks.h"

#define WINDOW\_WIDTH 1000

#define WINDOW\_HEIGHT 1000

GLuint VBO13;

GLuint IBO13;

GLuint gWVPLocation13;

static const char\* pVS = " \n\

#version 330 \n\

\n\

layout (location = 0) in vec3 Position; \n\

\n\

uniform mat4 gWVP; \n\

\n\

out vec4 Color; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWVP \* vec4(Position, 1.0); \n\

Color = vec4(clamp(Position, 0.0, 1.0), 1.0); \n\

}";

static const char\* pFS = " \n\

#version 330 \n\

\n\

in vec4 Color; \n\

\n\

out vec4 FragColor; \n\

\n\

void main() \n\

{ \n\

FragColor = Color; \n\

}";

static void RenderSceneCB() {

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.1f;

static float movement = 0.0f;

static bool flag = true;

if (movement >= 5.0f) {

flag = false;

}

else if (movement <= -5.0f) {

flag = true;

}

if (flag)

movement += 0.001f;

else

movement -= 0.001f;

Pipeline p;

p.Rotate(Scale, Scale, Scale);

p.WorldPos(movement, 0.0f, 3.0f);

Vector3f CameraPos(0.0f, 0.0f, -3.0f);

Vector3f CameraTarget(0.0f, 0.0f, 1.0f);

Vector3f CameraUp(0.0f, 1.0f, 0.0f);

p.SetCamera(CameraPos, CameraTarget, CameraUp);

p.SetPerspectiveProj(60.0f, WINDOW\_WIDTH, WINDOW\_HEIGHT, 1.0f, 100.0f);

glUniformMatrix4fv(gWVPLocation13, 1, GL\_TRUE, (const GLfloat\*)p.GetTrans());

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO13);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO13);

glDrawElements(GL\_TRIANGLES, 12, GL\_UNSIGNED\_INT, 0);

glDisableVertexAttribArray(0);

glutSwapBuffers();

}

static void InitializeGlutCallbacks() {

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(RenderSceneCB);

}

static void CreateVertexBuffer() {

Vector3f Vertices[4];

Vertices[0] = Vector3f(-1.0f, -1.0f, 0.5773f);

Vertices[1] = Vector3f(0.0f, -1.0f, -1.15475);

Vertices[2] = Vector3f(1.0f, -1.0f, 0.5773f);

Vertices[3] = Vector3f(0.0f, 1.0f, 0.0f);

glGenBuffers(1, &VBO13);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO13);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW);

}

static void CreateIndexBuffer() {

unsigned int Indices[] = { 0, 3, 1,

1, 3, 2,

2, 3, 0,

0, 2, 1 };

glGenBuffers(1, &IBO13);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO13);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(Indices), Indices, GL\_STATIC\_DRAW);

}

static void AddShader(GLuint ShaderProgram, const char\* pShaderText, GLenum ShaderType) {

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

fprintf(stderr, "Error creating shader type %d\n", ShaderType);

exit(0);

}

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

glCompileShader(ShaderObj);

GLint success;

glGetShaderiv(ShaderObj, GL\_COMPILE\_STATUS, &success);

if (!success) {

GLchar InfoLog[1024];

glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);

fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);

exit(1);

}

glAttachShader(ShaderProgram, ShaderObj);

}

static void CompileShaders() {

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

fprintf(stderr, "Error creating shader program\n");

exit(1);

}

AddShader(ShaderProgram, pVS, GL\_VERTEX\_SHADER);

AddShader(ShaderProgram, pFS, GL\_FRAGMENT\_SHADER);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

glLinkProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_LINK\_STATUS, &Success);

if (Success == 0) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Error linking shader program: '%s'\n", ErrorLog);

exit(1);

}

glValidateProgram(ShaderProgram);

glGetProgramiv(ShaderProgram, GL\_VALIDATE\_STATUS, &Success);

if (!Success) {

glGetProgramInfoLog(ShaderProgram, sizeof(ErrorLog), NULL, ErrorLog);

fprintf(stderr, "Invalid shader program: '%s'\n", ErrorLog);

exit(1);

}

glUseProgram(ShaderProgram);

gWVPLocation13 = glGetUniformLocation(ShaderProgram, "gWVP");

assert(gWVPLocation13 != 0xFFFFFFFF);

}

void invoke13(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(WINDOW\_WIDTH, WINDOW\_HEIGHT);

glutInitWindowPosition(100, 100);

glutCreateWindow("Tutorial 13");

InitializeGlutCallbacks();

// Must be done after glut is initialized!

GLenum res = glewInit();

if (res != GLEW\_OK) {

fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));

return;

}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

CreateVertexBuffer();

CreateIndexBuffer();

CompileShaders();

glutMainLoop();

}

**pipeline.cpp**

#include "pipeline.h"

const Matrix4f\* Pipeline::GetTrans() {

Matrix4f ScaleTrans, RotateTrans, TranslationTrans, CameraTranslationTrans, CameraRotateTrans, PersProjTrans;

ScaleTrans.InitScaleTransform(m\_scale.x, m\_scale.y, m\_scale.z);

RotateTrans.InitRotateTransform(m\_rotateInfo.x, m\_rotateInfo.y, m\_rotateInfo.z);

TranslationTrans.InitTranslationTransform(m\_worldPos.x, m\_worldPos.y, m\_worldPos.z);

CameraTranslationTrans.InitTranslationTransform(-m\_camera.Pos.x, -m\_camera.Pos.y, -m\_camera.Pos.z);

CameraRotateTrans.InitCameraTransform(m\_camera.Target, m\_camera.Up);

PersProjTrans.InitPersProjTransform(m\_persProj.FOV, m\_persProj.Width, m\_persProj.Height, m\_persProj.zNear, m\_persProj.zFar);

m\_transformation = PersProjTrans \* CameraRotateTrans \* CameraTranslationTrans \* TranslationTrans \* RotateTrans \* ScaleTrans;

return &m\_transformation;

}