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

Отчёт по лабораторной работе №2

По дисциплине “Инженерная и компьютерная графика”

Выполнил:

Студент группы ПРО-231 К.И. Арменшин

Проверил:

доцент кафедры ВМиК Котельников В.А.

Уфа – 2023

**Цель:**

Изучение преобразования для трехмерных объектов.

**Ход работы:**

**Уроки 6-8.**

Реализовано движение, изменение размеров и вращение объектов (Рисунок 1).

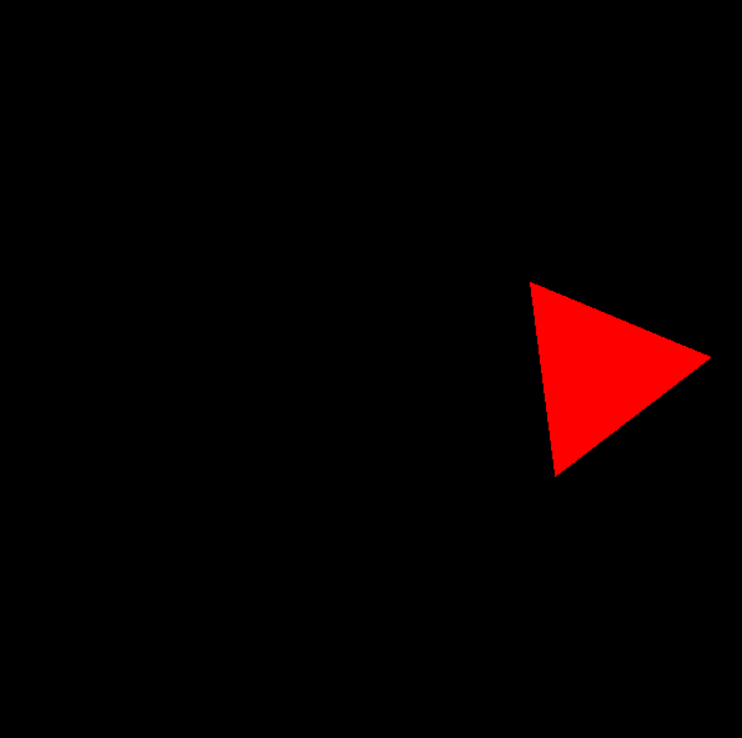
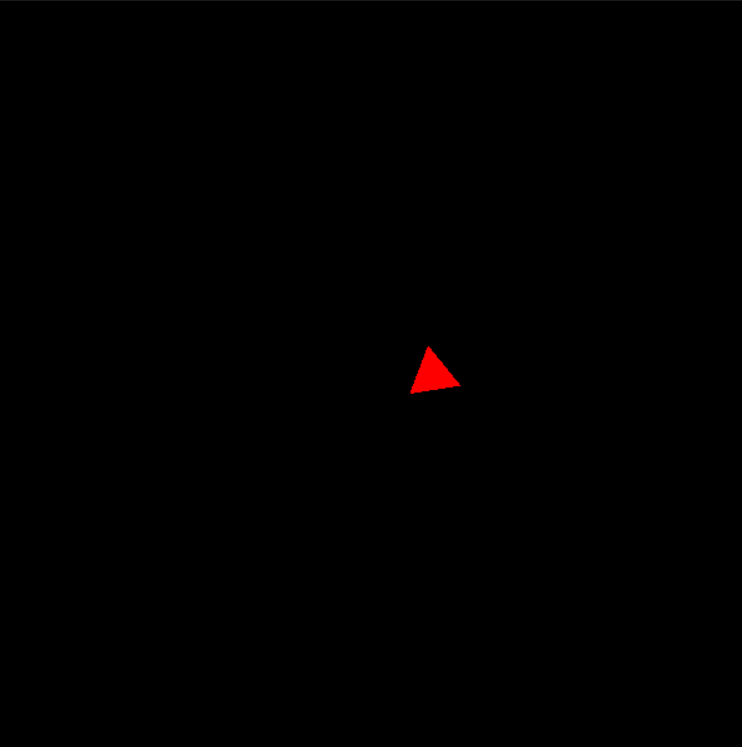
 

Рисунок . Трансформация объекта

**Уроки 12-13.**

Реализованы проекция перспективы и пространство камеры (Рисунок 3).

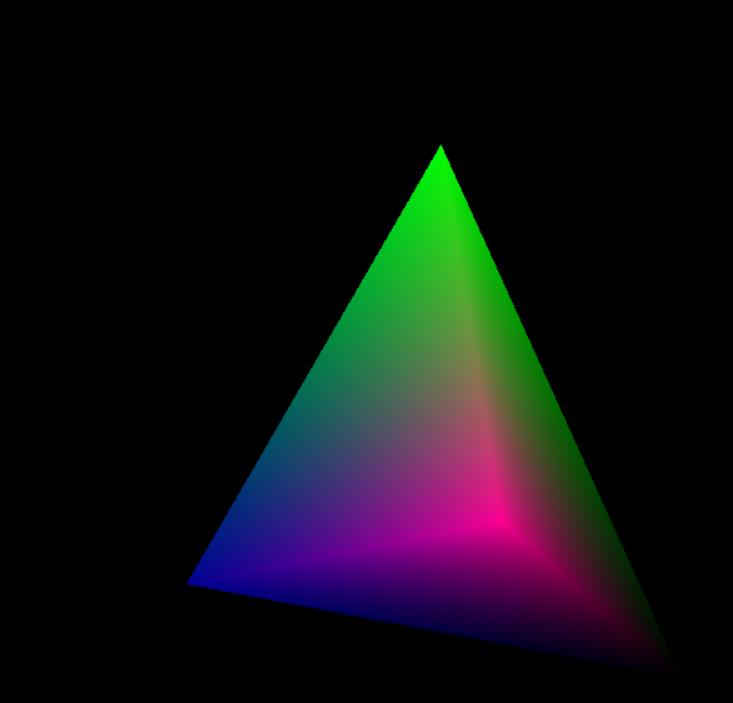


Рисунок . Пространство камеры

**Вывод**

В результате выполнения данной лабораторной работы были выполнены уроки 6-8, 12-13, в ходе которых были разработаны программы, демонстрирующие движение, изменение размеров и вращение объектов, а также реализованы проекция перспективы и пространство камеры.

Полный код находится в Github:

https://github.com/karimarm/Labs\_ECG.git

**Приложение 1**

L2.1\_ECG.cpp

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <cmath>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "math\_3d.h"

GLuint VBO; // Указатель на буфер вершин

GLuint gWorldLocation; // Указатель для доступа к матрице

// Вершинный шейдер

static const char\* pVS = " \n\

#version 330 \n\

\n\

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

\n\

//uniform mat4 gWorldScale, gWorldRotate, gWorldPosition, gWorld[2]; \n\

uniform mat4 gWorld[3]; \n\

\n\

void main() \n\

{ \n\

gl\_Position = gWorld[2] \* (gWorld[1] \* (gWorld[0] \* vec4(Position, 1.0))); \n\

}";

// рагментный шейдер

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); // Очистка буфера

static float Scale = 0.0f;

Scale += 0.0005f;

// Матрица преобразования

float matrix4[3][16];

matrix4[0][0] = cosf(Scale); matrix4[0][1] = -sinf(Scale); matrix4[0][2] = 0.0f; matrix4[0][3] = 0.0f;

matrix4[0][4] = sinf(Scale); matrix4[0][5] = cosf(Scale); matrix4[0][6] = 0.0f; matrix4[0][7] = 0.0f;

matrix4[0][8] = 0.0f; matrix4[0][9] = 0.0f; matrix4[0][10] = 1.0f; matrix4[0][11] = 0.0f;

matrix4[0][12] = 0.0f; matrix4[0][13] = 0.0f; matrix4[0][14] = 0.0f; matrix4[0][15] = 1.0f;

matrix4[1][0] = sinf(Scale); matrix4[1][1] = 0.0f; matrix4[1][2] = 0.0f; matrix4[1][3] = 0.0f;

matrix4[1][4] = 0.0f; matrix4[1][5] = sinf(Scale); matrix4[1][6] = 0.0f; matrix4[1][7] = 0.0f;

matrix4[1][8] = 0.0f; matrix4[1][9] = 0.0f; matrix4[1][10] = sinf(Scale); matrix4[1][11] = 0.0f;

matrix4[1][12] = 0.0f; matrix4[1][13] = 0.0f; matrix4[1][14] = 0.0f; matrix4[1][15] = 1.0f;

matrix4[2][0] = 1.0f; matrix4[2][1] = 0.0f; matrix4[2][2] = 0.0f; matrix4[2][3] = sinf(Scale);

matrix4[2][4] = 0.0f; matrix4[2][5] = 1.0f; matrix4[2][6] = 0.0f; matrix4[2][7] = 0.0f;

matrix4[2][8] = 0.0f; matrix4[2][9] = 0.0f; matrix4[2][10] = 1.0f; matrix4[2][11] = 0.0f;

matrix4[2][12] = 0.0f; matrix4[2][13] = 0.0f; matrix4[2][14] = 0.0f; matrix4[2][15] = 1.0f;

glUniformMatrix4fv(gWorldLocation, 3, GL\_TRUE, &matrix4[0][0]); // Загрузка массива в uniform-переменную шейдера

glEnableVertexAttribArray(0); // Массив атрибутов вершин

glBindBuffer(GL\_ARRAY\_BUFFER, VBO); // Привязка буфера

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); // Восприятие данных внутри буфера

glDrawArrays(GL\_TRIANGLES, 0, 3); // Отрисовка

glDisableVertexAttribArray(0); // Отключить атрибуты вершин

glutSwapBuffers(); // Меняет буферы местами

//glutPostRedisplay();

}

static void InitializeGlutCallbacks()

{

glutDisplayFunc(RenderSceneCB); // Функция обратного вызова отрисовки одного кадра

glutIdleFunc(RenderSceneCB); // Ленивая функция рендера

}

static void CreateVertexBuffer()

{

// Массив вершин треугольника

Vector3f Vertices[3];

Vertices[0] = Vector3f(0.0f, 0.5f, 0.0f);

Vertices[1] = Vector3f(-0.433f, -0.25f, 0.0f);

Vertices[2] = Vector3f(0.433f, -0.25f, 0.0f);

glGenBuffers(1, &VBO); // Функция для генерации объектов

glBindBuffer(GL\_ARRAY\_BUFFER, VBO); // Привязка буфера

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW); // Наполнение буфера данными

}

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

{

// Создание шейдера

GLuint ShaderObj = glCreateShader(ShaderType);

if (ShaderObj == 0) {

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

exit(0);

}

// Указание исходного кода шейдера

const GLchar\* p[1];

p[0] = pShaderText;

GLint Lengths[1];

Lengths[0] = strlen(pShaderText);

glShaderSource(ShaderObj, 1, p, Lengths);

// Компилирование шейдера

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); // Присоединение скомпилированного объекта шейдера к объекту программы

}

static void CompileShaders()

{

// Создание программного объекта в который будут слинкованы все шейдеры

GLuint ShaderProgram = glCreateProgram();

if (ShaderProgram == 0) {

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

exit(1);

}

// Создание шейдеров

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

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

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

// Линковка и проверка на ошибки

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);

}

// Проверка программы

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); // Назначение отлинкованной программы шейдеров конвейеру

// Запрос и проверка uniform-переменной

gWorldLocation = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation != 0xFFFFFFFF);

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv); // Инициализация GLUT

// Настройка опций GLUT

// GLUT\_DOUBLE - двойная буферизация

// GLUT\_RGBA - буфер цвета

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

// Задание параметров окна и его создание с заголовком "Tutorial 03"

glutInitWindowSize(600, 600);

glutInitWindowPosition(100, 100);

glutCreateWindow("Lab 2.1");

// Отрисовка кадра

InitializeGlutCallbacks();

// Инициализация GLEW и проверка на ошибки

GLenum res = glewInit();

if (res != GLEW\_OK) {

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

return 1;

}

// Цвет очистки буфера кадра

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

// Создание массива вершин треугольника

CreateVertexBuffer();

// Создание и компиляция шейдеров

CompileShaders();

// Передача контроля GLUT'у

glutMainLoop();

return 0;

}

L2.2\_ECG.cpp

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include "pipeline.h"

#define WINDOW\_WIDTH 600

#define WINDOW\_HEIGHT 600

GLuint VBO; // Указатель на буфер вершин

GLuint IBO; // Указатель на буфер индексовч

GLuint gWVPLocation; // Указатель для доступа к матрице

// Вершинный шейдер

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\

}";

// рагментный шейдер

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); // Очистка буфера

static float Scale = 0.0f;

Scale += 0.01f;

// Создаёт объект конвейера и настраивает его

Pipeline p;

p.Scale(1.0f, 1.0f, 1.0f);

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

p.WorldPos(sinf(Scale\*0.01f), 0.0f, 0.0f);

/\*Vector3f CameraPos(0.0f, 3.0f, 0.0f);

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

Vector3f CameraUp(0.0f, 0.0f, 1.0f);\*/

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(gWVPLocation, 1, GL\_TRUE, (const GLfloat\*)p.GetTrans()); // Загрузка массива в uniform-переменную шейдера

glEnableVertexAttribArray(0); // Массив атрибутов вершин

glBindBuffer(GL\_ARRAY\_BUFFER, VBO); // Привязка буфера вершин

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); // Восприятие данных внутри буфера

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO); // Привязка буфера индексов

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

glDisableVertexAttribArray(0); // Отключить атрибуты вершин

glutSwapBuffers(); // Меняет буферы местами

}

static void InitializeGlutCallbacks()

{

glutDisplayFunc(RenderSceneCB);

glutIdleFunc(RenderSceneCB);

}

static void CreateVertexBuffer()

{

// Массив вершин треугольников

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, &VBO); // Функция для генерации объектов

glBindBuffer(GL\_ARRAY\_BUFFER, VBO); // Привязка буфера

glBufferData(GL\_ARRAY\_BUFFER, sizeof(Vertices), Vertices, GL\_STATIC\_DRAW); // Наполнение буфера данными

}

static void CreateIndexBuffer()

{

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

1, 3, 2,

2, 3, 0,

0, 2, 1 };

/\*unsigned int Indices[] = { 0, 2, 1,

0, 3, 1,

1, 3, 2,

2, 3, 0 };\*/

glGenBuffers(1, &IBO); // Функция для генерации

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO); // Привязка буфера индексов

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(Indices), Indices, GL\_STATIC\_DRAW); // Наполнение буфера данными

}

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);

gWVPLocation = glGetUniformLocation(ShaderProgram, "gWVP");

assert(gWVPLocation != 0xFFFFFFFF);

}

int main(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 1;

}

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

CreateVertexBuffer();

CreateIndexBuffer();

CompileShaders();

glutMainLoop();

return 0;

}