ФГБОУ ВО

Уфимский Университет Науки и Технологий.

Кафедра вычислительной математики и кибернетики.

Отчет

по лабораторной работе №2

по теме “ Изучение преобразования для трехмерных объектов”

по дисциплине

**«Инженерная и компьютерная графика»**

Выполнил: ст.гр. ПРО-231Б

Семенов Г.Д.

Принял:

Котельников В.А.

Уфа 2023 г.

**Цель:** Изучение преобразования для трехмерных объектов.

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

**Ход работы:**

Урок 06 – перемещение.

Рисунок 1. Перемещение объекта по оси X.

Урок 07 – вращение.

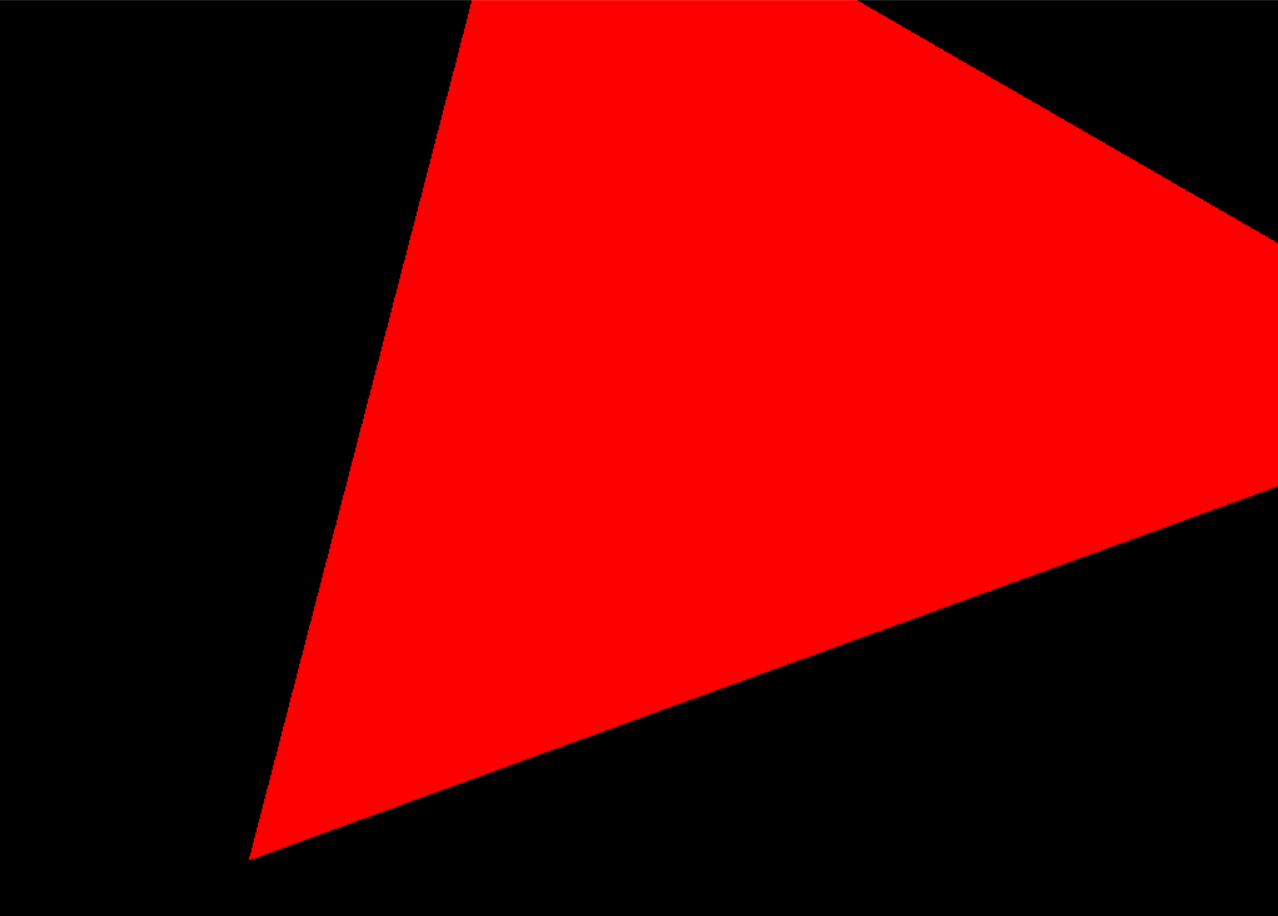
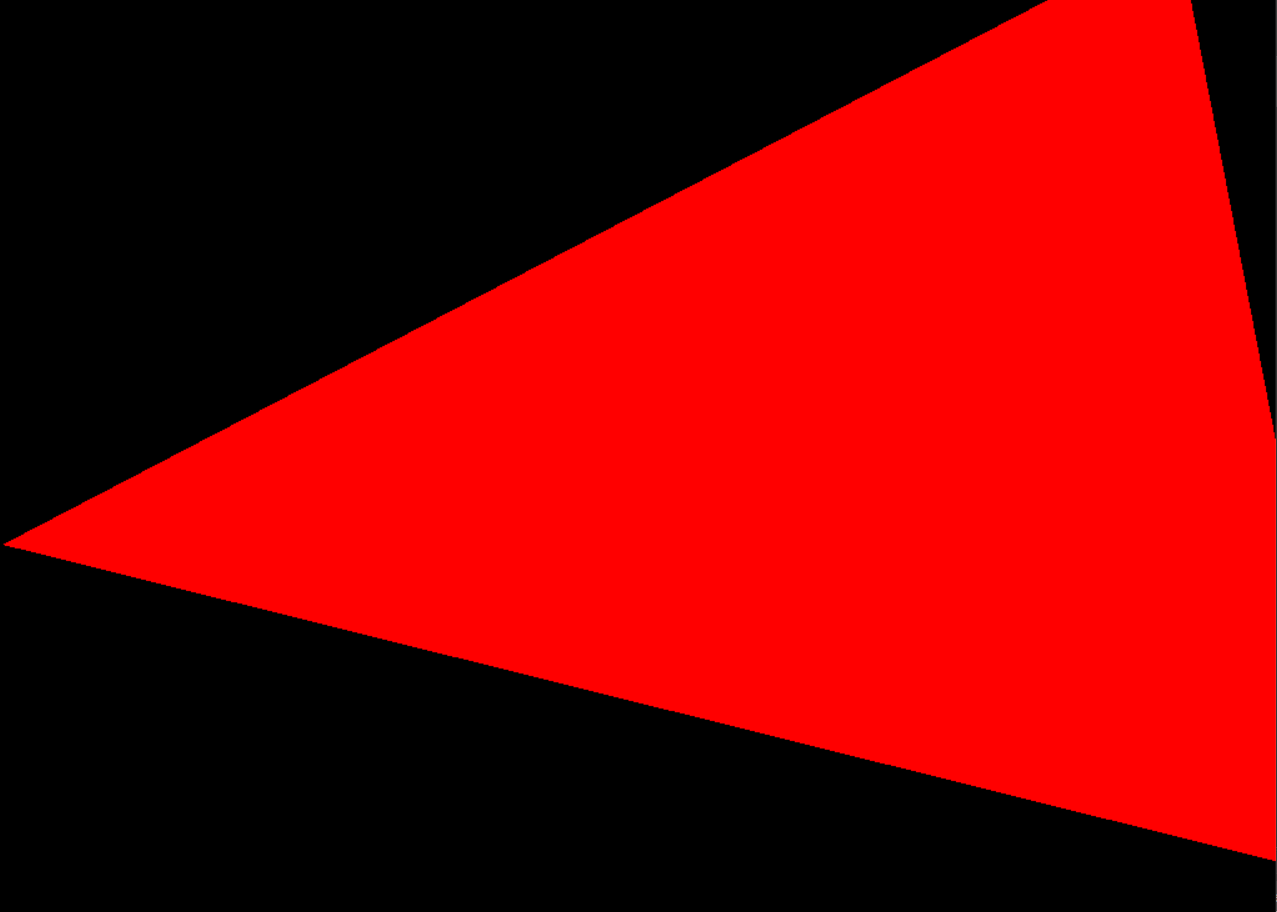
 

Рисунок 2. Вращение объекта относительно оси Z.

Урок 08 – преобразование масштаба.

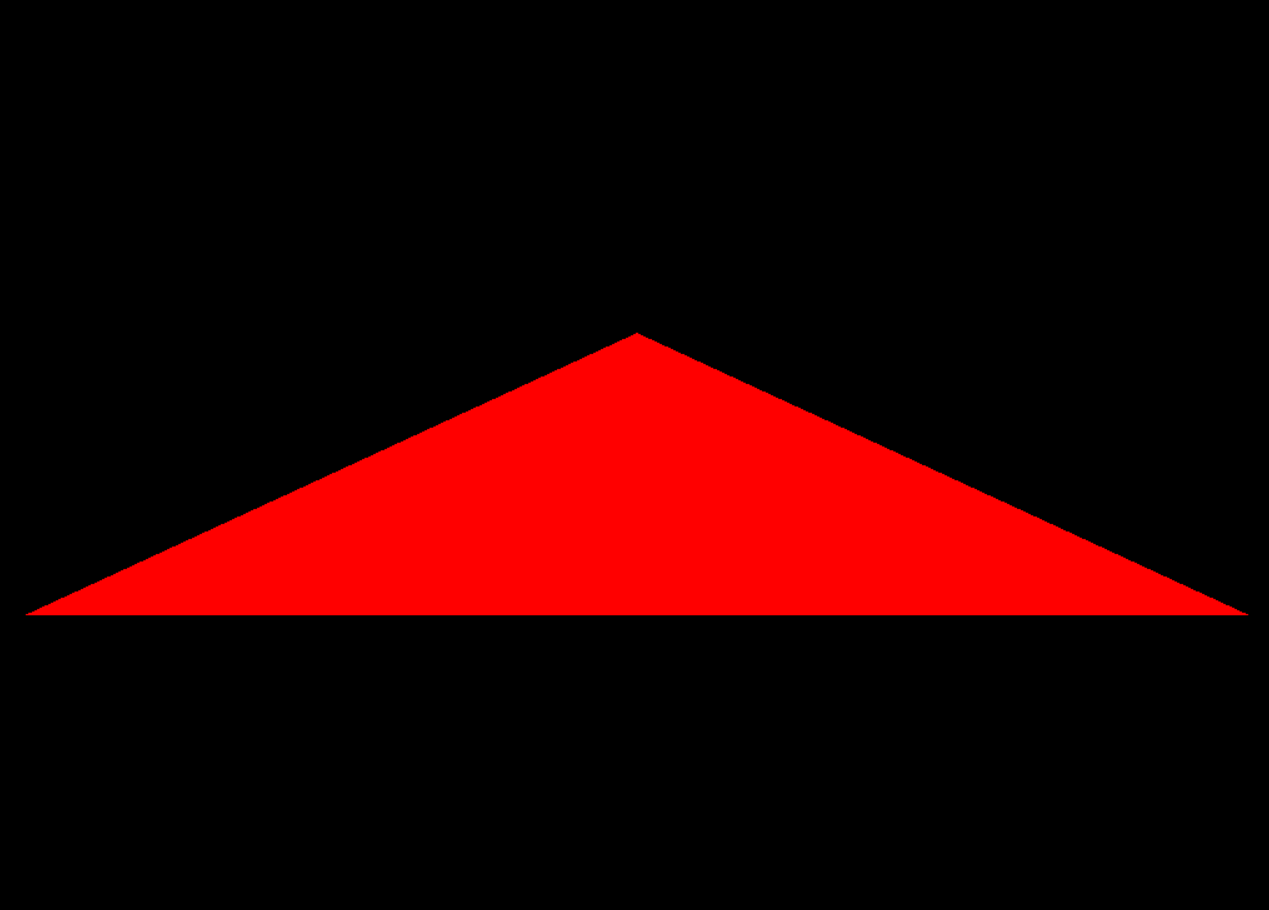
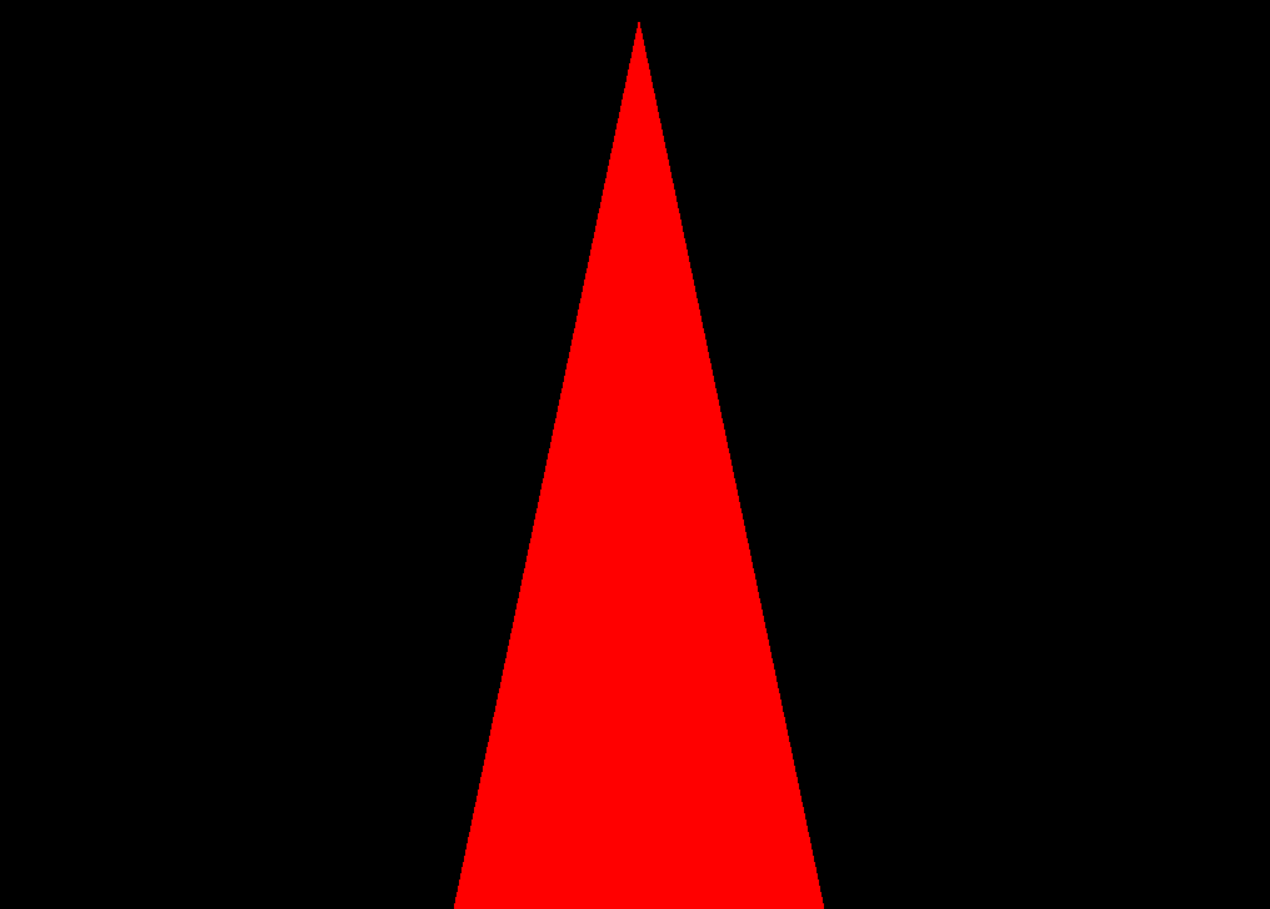
 

Рисунок 3. Изменение масштаба объекта.

Урок 12 – Проекция перспективы.

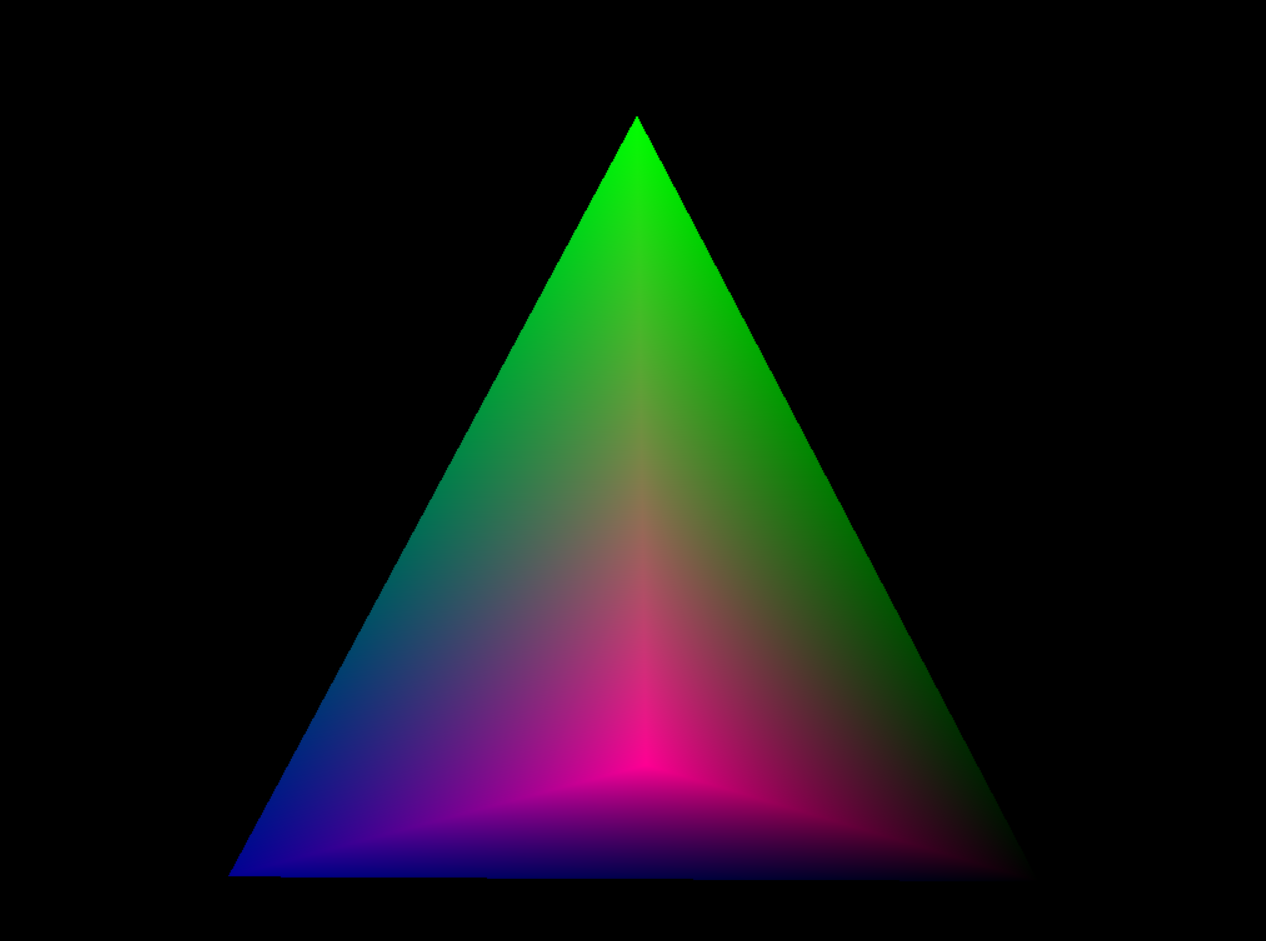
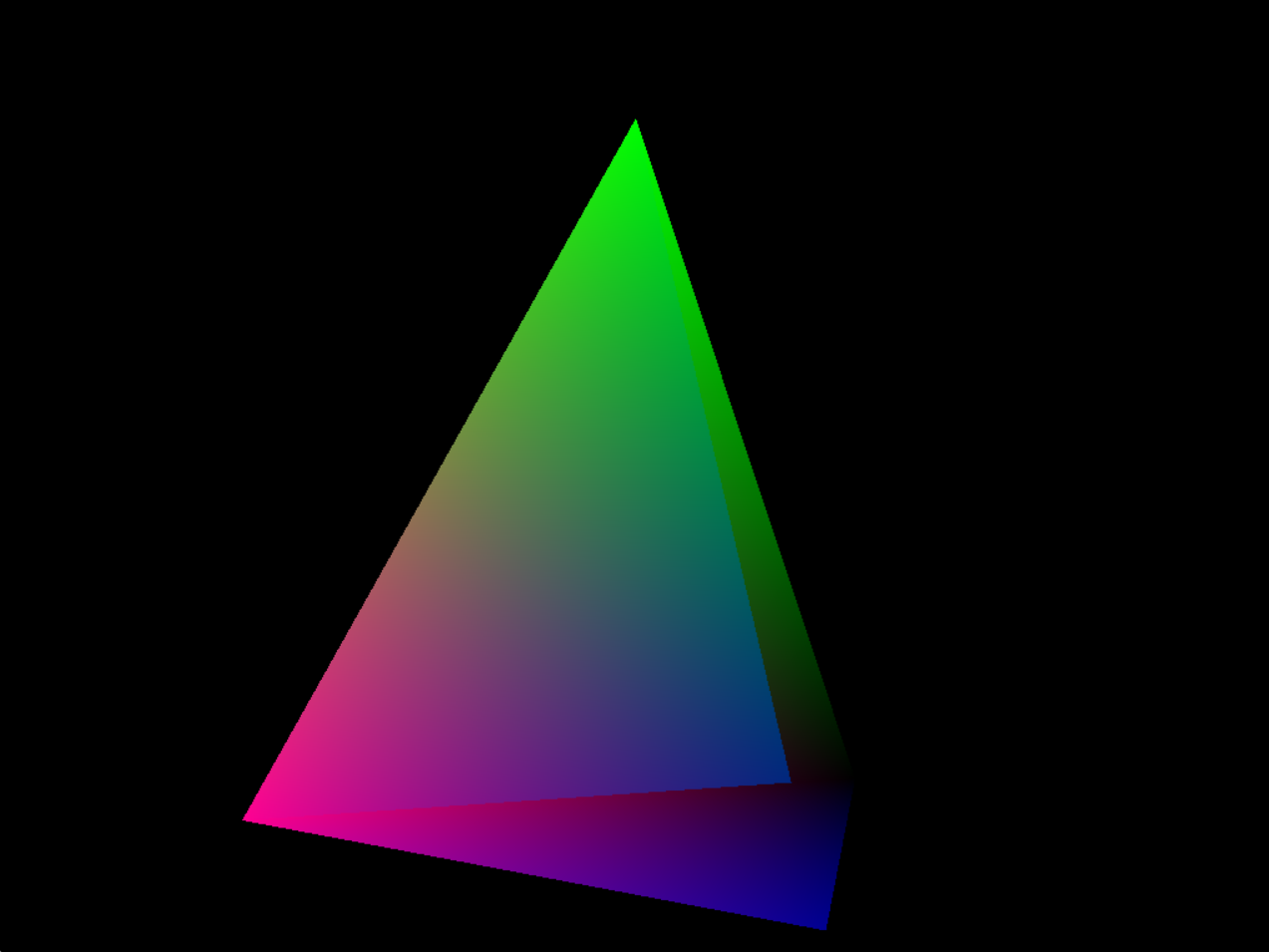
 

Рисунок 4. Вращение объекта вокруг камеры.

Урок 13 – Пространство камеры.

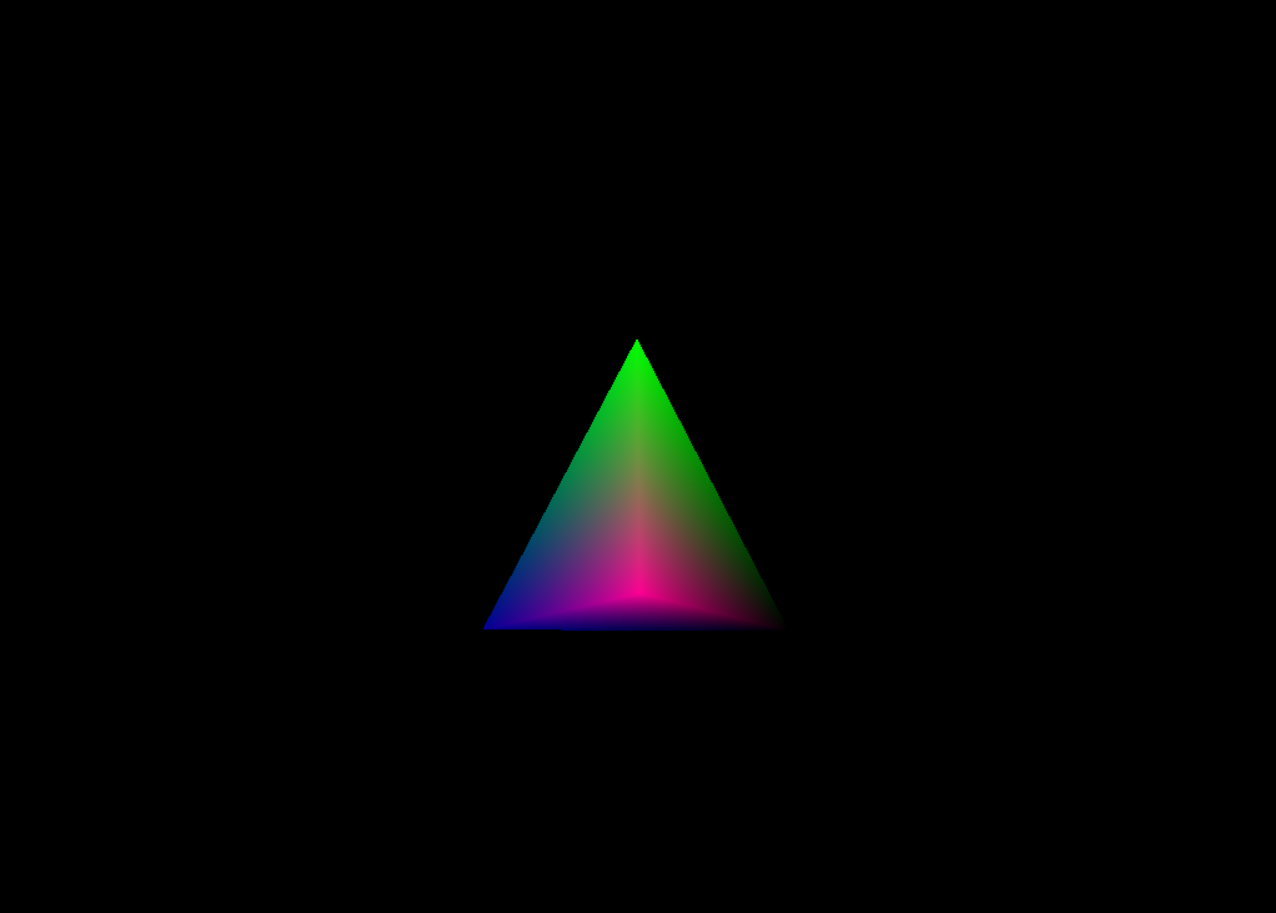
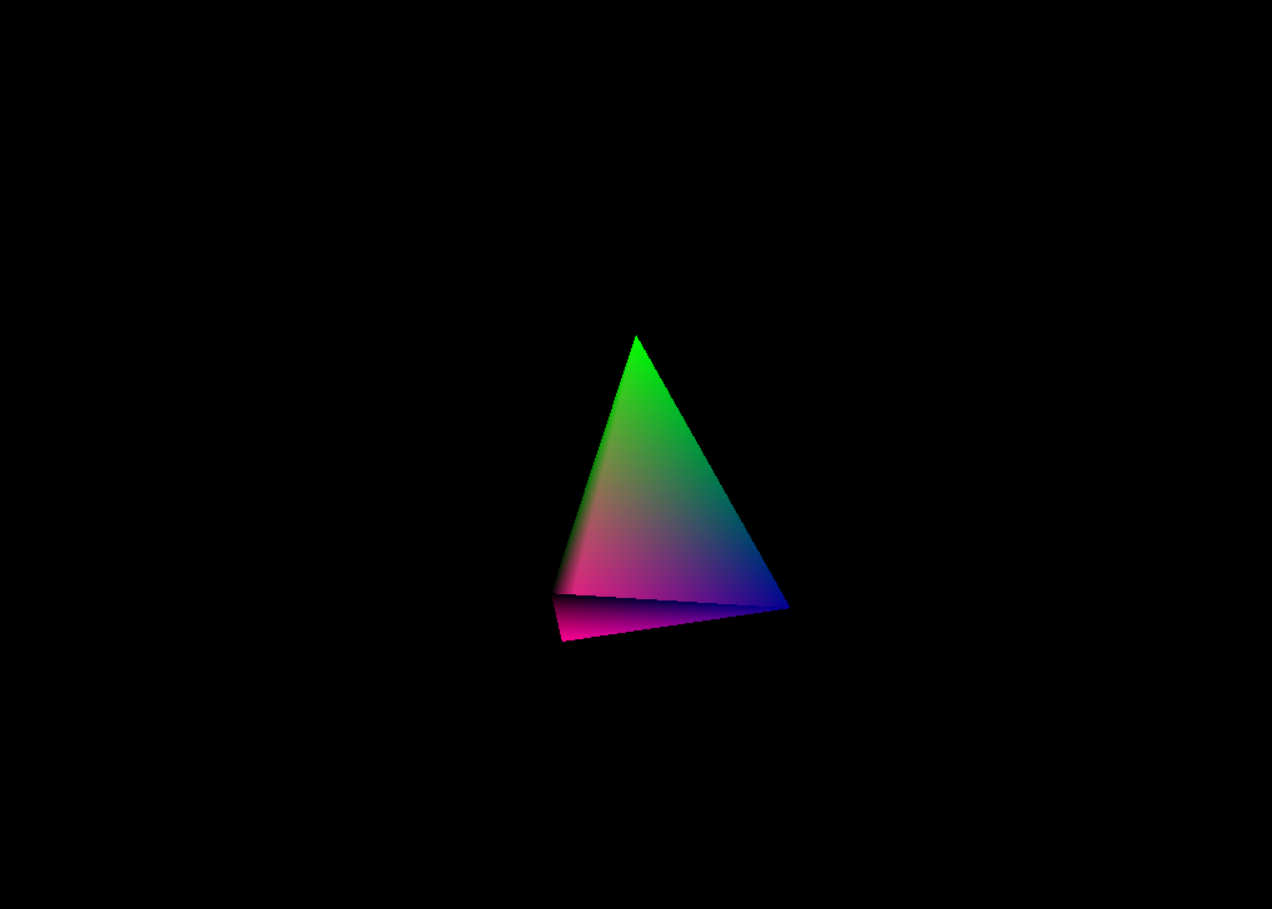
 

Рисунок 5. Вращение камеры вокруг объекта.

**Вывод**

Я изучил реализацию перемещения, вращения, изменения масштаба объекта, а также перемещения камеры, вращения камеры вокруг объекта, параметры камеры (такие как угол обзора, расположение, направление).

Ссылка на GitHub:

<https://github.com/SemyonovGleb/Eacg>

**Приложение 1**

Код **main.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 1024

#define WINDOW\_HEIGHT 768

GLuint VBO;

GLuint IBO;

GLuint gWVPLocation;

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;

Pipeline p;

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

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

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

Vector3f CameraTarget(0.0f, 0.0f, 2.0f);

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());

glEnableVertexAttribArray(0);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

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

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

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, &VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

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, &IBO);

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

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);

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;

}

Код **math\_3d.h**:

#ifndef MATH\_3D\_H

#define MATH\_3D\_H

#include <stdio.h>

#include <math.h>

#define ToRadian(x) ((x) \* M\_PI / 180.0f)

#define ToDegree(x) ((x) \* 180.0f / M\_PI)

struct Vector3f

{

float x;

float y;

float z;

Vector3f()

{

}

Vector3f(float \_x, float \_y, float \_z)

{

x = \_x;

y = \_y;

z = \_z;

}

Vector3f Cross(const Vector3f& v) const;

Vector3f& Normalize();

void Print() const

{

printf("(%.02f, %.02f, %.02f", x, y, z);

}

};

class Matrix4f

{

public:

float m[4][4];

Matrix4f()

{

}

inline void InitIdentity()

{

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

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

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

}

inline Matrix4f operator\*(const Matrix4f& Right) const

{

Matrix4f Ret;

for (unsigned int i = 0 ; i < 4 ; i++) {

for (unsigned int j = 0 ; j < 4 ; j++) {

Ret.m[i][j] = m[i][0] \* Right.m[0][j] +

m[i][1] \* Right.m[1][j] +

m[i][2] \* Right.m[2][j] +

m[i][3] \* Right.m[3][j];

}

}

return Ret;

}

void InitScaleTransform(float ScaleX, float ScaleY, float ScaleZ);

void InitRotateTransform(float RotateX, float RotateY, float RotateZ);

void InitTranslationTransform(float x, float y, float z);

void InitCameraTransform(const Vector3f& Target, const Vector3f& Up);

void InitPersProjTransform(float FOV, float Width, float Height, float zNear, float zFar);

};

#endif /\* MATH\_3D\_H \*/

Код **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;

}

Код **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;

}

Код **pipeline.h**

#ifndef PIPELINE\_H

#define PIPELINE\_H

#include "math\_3d.h"

class Pipeline

{

public:

Pipeline()

{

m\_scale = Vector3f(1.0f, 1.0f, 1.0f);

m\_worldPos = Vector3f(0.0f, 0.0f, 0.0f);

m\_rotateInfo = Vector3f(0.0f, 0.0f, 0.0f);

}

void Scale(float ScaleX, float ScaleY, float ScaleZ)

{

m\_scale.x = ScaleX;

m\_scale.y = ScaleY;

m\_scale.z = ScaleZ;

}

void WorldPos(float x, float y, float z)

{

m\_worldPos.x = x;

m\_worldPos.y = y;

m\_worldPos.z = z;

}

void Rotate(float RotateX, float RotateY, float RotateZ)

{

m\_rotateInfo.x = RotateX;

m\_rotateInfo.y = RotateY;

m\_rotateInfo.z = RotateZ;

}

void SetPerspectiveProj(float FOV, float Width, float Height, float zNear, float zFar)

{

m\_persProj.FOV = FOV;

m\_persProj.Width = Width;

m\_persProj.Height = Height;

m\_persProj.zNear = zNear;

m\_persProj.zFar = zFar;

}

void SetCamera(const Vector3f& Pos, const Vector3f& Target, const Vector3f& Up)

{

m\_camera.Pos = Pos;

m\_camera.Target = Target;

m\_camera.Up = Up;

}

const Matrix4f\* GetTrans();

private:

Vector3f m\_scale;

Vector3f m\_worldPos;

Vector3f m\_rotateInfo;

struct {

float FOV;

float Width;

float Height;

float zNear;

float zFar;

} m\_persProj;

struct {

Vector3f Pos;

Vector3f Target;

Vector3f Up;

} m\_camera;

Matrix4f m\_transformation;

};

#endif /\* PIPELINE\_H \*/