#include <stdio.h>

#include <iostream>

#include <string.h>

#include <assert.h>

#include <math.h>

#include <GL/glew.h>

#include <GL/freeglut.h>

#include <glm/glm.hpp>

#include "pipeline3.h"

#define WINDOW\_WIDTH 1024

#define WINDOW\_HEIGHT 768

using namespace glm;

GLuint VBO;

GLuint IBO;

GLuint gWorldLocation;

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\

}";

void RenderSceneCB()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

static float Scale = 0.0f;

Scale += 0.1f; //

Pipeline p;

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

p.SetPerspectiveProj(30.0f, WINDOW\_WIDTH, WINDOW\_HEIGHT, 1.0f, 1000.0f);

vec3 CameraPos(1.0f, 1.0f, -3.0f);

vec3 CameraTarget(0.45f, 0.0f, 1.0f);

vec3 CameraUp(0.0f, 1.0f, 0.0f);

p.SetCamera(CameraPos, CameraTarget, CameraUp);

glUniformMatrix4fv(gWorldLocation, 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()

{

vec3 Vertices[4];

Vertices[0] = vec3(-1.0f, -1.0f, 0.0f);

Vertices[1] = vec3(0.0f, -1.0f, 1.0f);

Vertices[2] = vec3(1.0f, -1.0f, 0.0f);

Vertices[3] = vec3(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);

glLinkProgram(ShaderProgram);

GLint Success = 0;

GLchar ErrorLog[1024] = { 0 };

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);

gWorldLocation = glGetUniformLocation(ShaderProgram, "gWorld");

assert(gWorldLocation != 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();

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;

}

#define M\_PI 3.14159F

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

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

#include "pipeline3.h"

/\*vec3 vec3::Cross(const vec3& 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 vec3(\_x, \_y, \_z);

}\*/

/\*vec3& vec3::Normalize()

{

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

x /= Length;

y /= Length;

z /= Length;

return \*this;

}\*/

void Pipeline::InitScaleTransform(mat4& m) const

{

m[0][0] = m\_scale.x; m[0][1] = 0.0f; m[0][2] = 0.0f; m[0][3] = 0.0f;

m[1][0] = 0.0f; m[1][1] = m\_scale.y; m[1][2] = 0.0f; m[1][3] = 0.0f;

m[2][0] = 0.0f; m[2][1] = 0.0f; m[2][2] = m\_scale.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 Pipeline::InitRotateTransform(mat4& m) const

{

mat4 rx, ry, rz;

const float x = ToRadian(m\_rotateInfo.x);

const float y = ToRadian(m\_rotateInfo.y);

const float z = ToRadian(m\_rotateInfo.z);

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

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

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

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

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

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

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

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

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

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

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

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

m = rz \* ry \* rx;

}

void Pipeline::InitTranslationTransform(mat4& m) const

{

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

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

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

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

}

void Pipeline::InitPerspectiveProj(mat4& m) const

{

const float ar = m\_persProj.Width / m\_persProj.Height;

const float zNear = m\_persProj.zNear;

const float zFar = m\_persProj.zFar;

const float zRange = zNear - zFar;

const float tanHalfFOV = tanf(ToRadian(m\_persProj.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;

}

/\*const mat4\* Pipeline::GetTrans()

{

mat4 ScaleTrans, RotateTrans, TranslationTrans, PersProjTrans;

InitScaleTransform(ScaleTrans);

InitRotateTransform(RotateTrans);

InitTranslationTransform(TranslationTrans);

InitPerspectiveProj(PersProjTrans);

m\_transformation = RotateTrans \* TranslationTrans \* PersProjTrans \* ScaleTrans;//тпрс было

return &m\_transformation;

}\*/

const mat4\* Pipeline::GetTrans()

{

mat4 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);

\*/

InitScaleTransform(ScaleTrans);

InitRotateTransform(RotateTrans);

InitTranslationTransform(TranslationTrans);

InitTranslationTransform(CameraTranslationTrans);

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

/\*PersProjTrans.InitPersProjTransform(m\_persProj.FOV, m\_persProj.Width,

m\_persProj.Height, m\_persProj.zNear, m\_persProj.zFar);

InitPersProjTransform(PersProjTrans);\*/

InitPerspectiveProj(PersProjTrans);

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

return &m\_transformation;

}

//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 vec3& Target, const vec3& Up);

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