Rotating Cube

Code

```
/*****************************************************************
*
* RotatingCube_GLM.cpp
*
* Description: This example is a modified version of the original
* example code with a colored, rotating cube in shader-based
* OpenGL. Some of the original functionality is now implemented
* via the C++ mathematics library GLM.
*
* Note that the example requires the local installation of the
* header-only library GLM.
*
* Computer Graphics Proseminar SS 2015
*
* Interactive Graphics and Simulation Group
* Institute of Computer Science
* University of Innsbruck
*
******************************************************************/

/* Standard includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define GLM_FORCE_RADIANS  /* Use radians in all GLM functions */

/* GLM includes - adjust path as required for local installation */
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp" /* Provides glm::translate, glm::rotate,
                                         * glm::scale, glm::perspective */
#include "glm/gtc/type_ptr.hpp"         /* Vector/matrix handling */

/* OpenGL includes */
#include <GL/glew.h>
#include <GL/freeglut.h>

/* Local includes */
extern "C"
{
    #include "LoadShader.h"   /* Provides loading function for shader code */
}

/*---------------------------------------------------------------*/

/* Define handle to a vertex buffer object */
GLuint VBO;

/* Define handle to a color buffer object */
GLuint CBO;

/* Define handle to an index buffer object */
GLuint IBO;
```

```
/* Indices to vertex attributes; in this case positon and color */
enum DataID {vPosition = 0, vColor = 1};

/* Strings for loading and storing shader code */
static const char* VertexShaderString;
static const char* FragmentShaderString;

GLuint ShaderProgram;

glm::mat4 ProjectionMatrix; /* Perspective projection matrix */
glm::mat4 ViewMatrix;       /* Camera view matrix */
glm::mat4 ModelMatrix;      /* Model matrix */
glm::mat4 PVMMatrix;        /* Final combined transformation */

/* Transformation matrices for model positioning */
glm::mat4 TranslateOrigin;
glm::mat4 TranslateDown;
glm::mat4 RotateX;
glm::mat4 RotateZ;
glm::mat4 InitialTransform;


GLfloat vertex_buffer_data[] = { /* 8 cube vertices */
   -1.0, -1.0,  1.0,
    1.0, -1.0,  1.0,
    1.0,  1.0,  1.0,
   -1.0,  1.0,  1.0,
   -1.0, -1.0, -1.0,
    1.0, -1.0, -1.0,
    1.0,  1.0, -1.0,
   -1.0,  1.0, -1.0,
};

GLfloat color_buffer_data[] = { /* RGB color values for vertices */
   1.0, 0.0, 0.0,
   0.0, 1.0, 0.0,
   0.0, 1.0, 1.0,
   1.0, 1.0, 1.0,
   1.0, 0.0, 1.0,
   0.0, 1.0, 0.0,
   1.0, 0.0, 1.0,
   1.0, 1.0, 1.0,
};

GLushort index_buffer_data[] = { /* Indices of 6*2 triangles */
   0, 1, 2,
   2, 3, 0,
   1, 5, 6,
   6, 2, 1,
   7, 6, 5,
   5, 4, 7,
   4, 0, 3,
   3, 7, 4,
   4, 5, 1,
   1, 0, 4,
   3, 2, 6,
   6, 7, 3,
};

/*----------------------------------------------------------------*/
```

```
/************************************************************************
*
* Display
*
* This function is called when the content of the window needs to be
* drawn/redrawn. It has been specified through 'glutDisplayFunc()';
* Enable vertex attributes, create binding between C program and
* attribute name in shader
*
************************************************************************/

void Display()
{
    /* Clear window; color specified in 'Initialize()' */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnableVertexAttribArray(vPosition);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0, 0);

    glEnableVertexAttribArray(vColor);
    glBindBuffer(GL_ARRAY_BUFFER, CBO);
    glVertexAttribPointer(vColor, 3, GL_FLOAT,GL_FALSE, 0, 0);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);
    GLint size;
    glGetBufferParameteriv(GL_ELEMENT_ARRAY_BUFFER, GL_BUFFER_SIZE, &size);

    /* Associate program with shader matrices */
    GLint PVMMatrixID = glGetUniformLocation(ShaderProgram, "ProjectionViewModelMatrix");
    if (PVMMatrixID == -1)
    {
        fprintf(stderr, "Could not bind uniform ProjectionViewModelMatrix\n");
        exit(-1);
    }
    glUniformMatrix4fv(PVMMatrixID, 1, GL_FALSE, glm::value_ptr(PVMMatrix));

    /* Issue draw command, using indexed triangle list */
    glDrawElements(GL_TRIANGLES, size/sizeof(GLushort), GL_UNSIGNED_SHORT, 0);

    /* Disable attributes */
    glDisableVertexAttribArray(vPosition);
    glDisableVertexAttribArray(vColor);

    /* Swap between front and back buffer */
    glutSwapBuffers();
}


/************************************************************************
*
* OnIdle
*
*
*
************************************************************************/

void OnIdle()
{
    float angle = fmod((glutGet(GLUT_ELAPSED_TIME) / 1000.0), 360.0);

    /* Time dependent rotation matrix */
    glm::mat4 RotationMatrixAnim =
        glm::rotate(glm::mat4(1.0f),              /* Output matrix */
                    angle,                        /* Rotation angle */
                    glm::vec3(0.0f, 1.0f, 0.0f)); /* Rotation axis*/

    /* Apply model rotation; finally move cube down */
    ModelMatrix = TranslateDown * RotationMatrixAnim * InitialTransform;

    /* Set up single transformation matrix for complete transformation
       from model to screen space */
    PVMMatrix = ProjectionMatrix * ViewMatrix * ModelMatrix;

    /* Request redrawing of window content */
    glutPostRedisplay();
}


/******************************************************************
*
* SetupDataBuffers
*
* Create buffer objects and load data into buffers
*
*******************************************************************/

void SetupDataBuffers()
{
    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_buffer_data),
                 vertex_buffer_data, GL_STATIC_DRAW);

    glGenBuffers(1, &IBO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(index_buffer_data),
                 index_buffer_data, GL_STATIC_DRAW);

    glGenBuffers(1, &CBO);
    glBindBuffer(GL_ARRAY_BUFFER, CBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(color_buffer_data),
                 color_buffer_data, GL_STATIC_DRAW);
}


/******************************************************************
*
* AddShader
*
* This function creates and adds individual shaders
*
*******************************************************************/

void AddShader(GLuint ShaderProgram, const char* ShaderCode, GLenum ShaderType)
{
    /* Create shader object */
    GLuint ShaderObj = glCreateShader(ShaderType);

    if (ShaderObj == 0)
    {
        fprintf(stderr, "Error creating shader type %d\n", ShaderType);
        exit(0);
    }
    /* Associate shader source code string with shader object */
    glShaderSource(ShaderObj, 1, &ShaderCode, NULL);

    GLint success = 0;
    GLchar InfoLog[1024];

    /* Compile shader source code */
    glCompileShader(ShaderObj);
    glGetShaderiv(ShaderObj, GL_COMPILE_STATUS, &success);

    if (!success)
    {
        glGetShaderInfoLog(ShaderObj, 1024, NULL, InfoLog);
        fprintf(stderr, "Error compiling shader type %d: '%s'\n", ShaderType, InfoLog);
        exit(1);
    }

    /* Associate shader with shader program */
    glAttachShader(ShaderProgram, ShaderObj);
}


/*******************************************************************
*
* CreateShaderProgram
*
* This function creates the shader program; vertex and fragment
* shaders are loaded and linked into program; final shader program
* is put into the rendering pipeline
*
*******************************************************************/

void CreateShaderProgram()
{
    /* Allocate shader object */
    ShaderProgram = glCreateProgram();

    if (ShaderProgram == 0)
    {
        fprintf(stderr, "Error creating shader program\n");
        exit(1);
    }

    /* Load shader code from file */
    VertexShaderString = LoadShader("vertexshader.vs");
    FragmentShaderString = LoadShader("fragmentshader.fs");

    /* Separately add vertex and fragment shader to program */
    AddShader(ShaderProgram, VertexShaderString, GL_VERTEX_SHADER);
    AddShader(ShaderProgram, FragmentShaderString, GL_FRAGMENT_SHADER);

    GLint Success = 0;
    GLchar ErrorLog[1024];

    /* Link shader code into executable shader program */
    glLinkProgram(ShaderProgram);

    /* Check results of linking step */
    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);
    }

    /* Check if shader program can be executed */
    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);
    }

    /* Put linked shader program into drawing pipeline */
    glUseProgram(ShaderProgram);
}


/****************************************************************
*
* Initialize
*
* This function is called to initialize rendering elements, setup
* vertex buffer objects, and to setup the vertex and fragment shader
*
*****************************************************************/

void Initialize(void)
{
    /* Set background (clear) color to blue */
    glClearColor(0.0, 0.0, 0.4, 0.0);

    /* Enable depth testing */
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    /* Setup vertex, color, and index buffer objects */
    SetupDataBuffers();

    /* Setup shaders and shader program */
    CreateShaderProgram();

    /* Set projection transform */
    float fovy = 45.0*M_PI/180.0;
    float aspect = 1.0;
    float nearPlane = 1.0;
    float farPlane = 50.0;
    ProjectionMatrix = glm::perspective(fovy, aspect, nearPlane, farPlane);

    /* Set viewing transform */
    ViewMatrix = glm::lookAt(glm::vec3(0,0,10),    /* Eye vector */
                             glm::vec3(0,0,0),     /* Viewing center */
                             glm::vec3(0,1,-1) );  /* Up vector */

    /* Translate down */
    TranslateDown = glm::translate(glm::mat4(1.0f),
                                   glm::vec3(0.0f, -sqrtf(sqrtf(2.0) * 1.0), 0.0f));
    /* Initial transformation; translate and rotate cube onto tip */
    float RotAngleX = -M_PI * 45.0/180.0;
    float RotAngleZ =  M_PI * 35.0/180.0;

    InitialTransform = glm::rotate(glm::mat4(1.0f),
                                   RotAngleZ,
                                   glm::vec3(0.0f, 0.0f, 1.0f));
    InitialTransform = glm::rotate(InitialTransform,
                                   RotAngleX,
                                   glm::vec3(1.0f, 0.0f, 0.0f));
    InitialTransform = glm::translate(InitialTransform,
                                      glm::vec3(1.0f, 1.0f, 1.0f));
}


/****************************************************************
*
* main
*
* Main function to setup GLUT, GLEW, and enter rendering loop
*
*****************************************************************/

int main(int argc, char** argv)
{
    /* Initialize GLUT; set double buffered window and RGBA color model */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(400, 400);
    glutCreateWindow("CG Proseminar - Rotating Cube GLM");

    /* Initialize GL extension wrangler */
    GLenum res = glewInit();
    if (res != GLEW_OK)
    {
        fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));
        return 1;
    }

    /* Setup scene and rendering parameters */
    Initialize();


    /* Specify callback functions;enter GLUT event processing loop,
     * handing control over to GLUT */
    glutIdleFunc(OnIdle);
    glutDisplayFunc(Display);
    glutMainLoop();

    /* ISO C requires main to return int */
    return 0;
}
```

New Functions

GLint glGetUniformLocation(GLuint program, const GLchar *name)— Returns the location of a uniform variable

program - Specifies the program object to be queried.

namePoints - to a null terminated string containing the name of the uniform variable whose location is to be queried.

void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)— specify the value of a uniform variable for the current program object

location - Specifies the location of the uniform value to be modified.

count - Specifies the number of matrices that are to be modified. This should be 1 if the targeted uniform variable is not an array of matrices, and 1 or more if it is an array of matrices.

transpose - Specifies whether to transpose the matrix as the values are loaded into the uniform variable. Must be GL_FALSE.

value- Specifies a pointer to an array of count values that will be used to update the specified uniform variable.

void glGenBuffers(GLsizei n, GLuint * buffers)— returns n buffer object names in buffers. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to glGenBuffers.

n - Specifies the number of buffer object names to be generated.

buffers - Specifies an array in which the generated buffer object names are stored.