Lab 1

Getting started with Open GL

Kurs: Datorgrafik 2016

Labbrapport inlämnad: 8 November 2016

Gruppnummer:

1

Gruppmedlemmar:

Victor Svensson
920101-2714
Martin Hjalmarsson
860313-8218

Levererad till:

Daniel Canelhas

Uppgift 1.1 – Hello Gpu

Använd glGetString funktionen för att få namnet på rendreraren och vilken version av Open GL som stöds.

Utskriften från glGetString ger följande:

Renderer	AMD Radeon R9 200 Series
Open GL version supported	4.4.12874
Compability Profile Context	14.100.0.0

Hitta hårdvaruspecifikationen för rendreraren

AMD Radeon R9 200 Series säger bara vilken serie grafikkortet tillhör. Med hjälp av vendor ID och model ID funnet i enhetshanteraren kunde information om den specifika modellen hämtas.

Graphic card model	R9 290X
Video RAM	4GB
Number of processing units	2816
GPU Clock frequency	1000 mhz
Memory bandwith	352 / GB/s

Vilka funktioner saknas från den här versionen av Open GL jämfört med den senaste versionen?

Den nya Open GL versionen 4.5 har något som kallas ”Direct State Access”(DSA). Den medför att man kan modifiera OpenGL objekt utan att behöva binda dom till kontexten. I praktiken innebär det att istället för att kalla t.ex. **glGenVertexArrays**, **glBindVertexArray** osv så kan du kalla **glCreateVertexArrays** direkt. Det gör att man kan arbeta mer objektorienterat.

Uppgift 1.2 – Your first triangle

Beskriv funktionerna som används i programmet.

glGenVertexArrays	Genererar ett namn för vertex arrayen som skickas in
glBindVertexArray	Sätter given vertex array till den aktiva vertex arrayen
glGenBuffers	Genererar ett namn för buffern som skickas in
glBindBuffer	Sätter given buffer till den aktiva buffern

glBufferData	Laddar upp data till den aktiva buffern
glEnableVertexAttribArray	Aktiverar det givna attributet
glVertexAttribPointer	Specificerar ett attribut
glCreateShader	Skapar en shader av given typ
glShaderSource	Länkar den givna shadern till en shader-kod(sträng)
glCompileShader	Kompilerar shader-koden
glCreateProgram	Skapar ett tomt program som shaders kan bindas till
glAttachShader	Binder given shader med givet program
glLinkProgram	Länkar ihop programmet med bundna shaders. I gruppens fall: Skapar executables för vertex shader och för fragment shader.
glDeleteShader	Flaggar en shader för radering. Är shadern ej binden till något program så raderas den direkt. Annars så flaggas den för senare radering.
glUseProgram	Programmet installeras och läggs I current render state

Vilka dimensioner I screen-space mappas dina X,Y och Z koordinater till?

X	Positiv åt höger på skärmen
Y	Positiv uppåt på skärmen
Z	Positivt utåt från skärmen

Vad är gränserna för Z och vad händer om du går utanför den gränsen?

Gränserna är -1 och +1. Är du utanför detta intervall så clippas bilden.

Uppgift 1.3 – Introduction to shaders

Varför får fragment shadern olika värden på varje punkt I triangeln? Kan man kontrollera detta beteende?

Den får olika värden för varje fragment på grund ut av interpolation. Man kan kontrollera det här beteendet genom att använda olika interpolation qualifiers. Default är smooth. Ändrar man till flat så får hela triangeln färgen av den provokerande vertexen (den sista av de tre som bildar triangeln).

Uppgift 1.4 – Passing parameters to shader programs

Inget att redovisa under detta avsnitt.

Uppgift 1.5 – 3D geometry

Varför används index array?

Man sparar utrymme. Storleken på 60 shorts (index tabell) och 12 floats (vertex) är totalt 168 byte. Storleken på 60 floats, det vill säga om man skickar in alla vertex som används inklusive repetitioner, är 240 byte.

Uppgift 1.6 – Model, view & projection transformations

Varför är multiplikationsordningen viktig?

Multiplikation av matriser är inte kommutativ, dvs A*B != B*A.

Vad händer om man multiplicerar matriserna I omvänd ordning?

Om man utför multiplikation i omvänd ordning kommer resultatet inte att blir det förväntade. Om man tex utför perspektivmultiplikationen först så har man komprimerat z till värden mellan -1 och 1 vilket kommer att göra att resterande transformationer kommer ge ett konstigt resultat.

Uppgift 1.7 – Linear algebra isn't fun

Hittills har glDepthFunc varit sätt till GL_LESS. Vilka andra alternativ finns det för denna parameter?

Parametern används för att styra hur djupsorteringen fungerar. Ordet "godkänd" betyder I detta sammanhang att pixeln ritas ut och ett nytt Z-värde för den pixeln blir sparat I Z buffern.

GL_NEVER	Godkänns aldrig
GL_LESS	Godkänns om pixelns Z värde är mindre än det sparade Z värdet.
GL_EQUAL	Godkänns om pixelns Z värde är samma som det sparade Z värdet
GL_LEQUAL	Godkänns om pixelns Z värde är mindre eller lika som det sparade Z värdet
GL_GREATER	Godkänns om pixelns Z värde är större än det sparade Z värdet
GL_NOTEQUAL	Godkänns om pixelns Z värde inte är samma som det sparade värdet
GL_GEQUAL	Godkänns om pixelns Z värde är större eller lika som det sparade Z värdet
GL_ALWAYS	Godkänns alltid

Är det originala värdet att föredra? Varför? Vad kan hända annars?

Eftersom att man vill rita ut det som ligger närmare kameran överst så vill man att objekt med ett mindre z-värde ska ritas ut över objekt med ett högre z-värde. Väljer man t.ex. GL_GREATER så kan det bli så att små bakgrundsobjekt ritas över stora förgrundsobjekt och det ser konstigt ut.

Kod

Uppgift 1.1

C++

```
/* include statements removed */

static void error_callback(int error, const char* description)
{
  std::cerr << description;
}

int main(int argc, char const *argv[])
{
  // start GL context and O/S window using the GLFW helper library

  glfwSetErrorCallback(error_callback);
  if( !glfwInit() )
    exit(EXIT_FAILURE);
  GLFWwindow* window = glfwCreateWindow (20, 20, "Hello OpenGL", NULL, NULL);

  if (!window) {
    glfwTerminate();
    exit(EXIT_FAILURE);
  }
  glfwMakeContextCurrent (window);

  // start GLEW extension handler
  glewExperimental = GL_TRUE;
  glewInit ();

  //----------------------------------------------------------------------------
-------------------------------------------------------------------------------/
/
  //Query renderer and version using glGetString;

  std::cout << "Renderer:" << std::endl << glGetString(GL_RENDERER) <<
std::endl;
  std::cout << "OpenGL version supported:" << std::endl <<
glGetString(GL_VERSION) << std::endl;
  //----------------------------------------------------------------------------
-------------------------------------------------------------------------------/
/
  system("pause");
  // close GL context and any other GLFW resources
  glfwTerminate();
  exit(EXIT_SUCCESS);
}
```

Uppgift 1.2

C++

```
/* include statements */

static void error_callback(int error, const char* description)
{
  std::cerr << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
  if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action == GLFW_PRESS)
    glfwSetWindowShouldClose(window, GL_TRUE);
}

static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
  glViewport(0, 0, width, height);
}


int main(int argc, char const *argv[])
{
        // start GL context and O/S window using the GLFW helper library

        glfwSetErrorCallback(error_callback);
        if (!glfwInit())
                exit(EXIT_FAILURE);

        GLFWwindow* window = glfwCreateWindow(640, 480, "Hello Triangle", NULL,
NULL);
        glfwSetKeyCallback(window, key_callback);
        glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

        if (!window) {
                glfwTerminate();
                exit(EXIT_FAILURE);
        }
        glfwMakeContextCurrent(window);

        // start GLEW extension handler
        glewExperimental = GL_TRUE;
        glewInit();

        //------------------------------------------------------------------------
---//
        // Set up geometry, VBO, VAO
        //------------------------------------------------------------------------
---//

        /* verticies of two triangles */
        static const GLfloat vertex_array[] = {
                -0.5f, -0.5f, -0.1f,
                0.5f, -0.5f, -0.1f,
                -0.5f, 0.5f, -0.1f,
                -0.9f, 0.1f, -0.1f,
                0.3f, -0.2f, -0.1f,
                -0.5f, 0.1f, -0.1f };

        /* generate name and bind vao */
        GLuint VAO;
        glGenVertexArrays(1, &VAO);
        glBindVertexArray(VAO);

        /* generate name, bind vbo, upload data */
        GLuint VBO;
        glGenBuffers(1, &VBO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_array), vertex_array,
GL_STATIC_DRAW);

        /* specify and enable vertex attribute */
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(
                0,
                3,
                GL_FLOAT,
                GL_FALSE,
                0,
                (void*)0
        );


  const char* vertex_shader =
    "#version 400\n"
    "in vec3 vp;"
    "void main () {"
    "  gl_Position = vec4 (vp, 1.0);"
    "}";

  const char* fragment_shader =
    "#version 400\n"
    "out vec4 frag_colour;"
    "void main () {"
    "  frag_colour = vec4 (0.5, 0.0, 0.5, 1.0);"
    "}";

  GLuint vs = glCreateShader (GL_VERTEX_SHADER);
  glShaderSource (vs, 1, &vertex_shader, NULL);
  glCompileShader (vs);

  GLuint fs = glCreateShader (GL_FRAGMENT_SHADER);
  glShaderSource (fs, 1, &fragment_shader, NULL);
  glCompileShader (fs);

  GLuint shader_programme = glCreateProgram ();
  glAttachShader (shader_programme, fs);
  glAttachShader (shader_programme, vs);
  glLinkProgram (shader_programme);

  glDeleteShader(vs);
  glDeleteShader(fs);

  glUseProgram (shader_programme);

  while (!glfwWindowShouldClose (window))
  {
    // update other events like input handling
    glfwPollEvents ();
    // clear the drawing surface
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //-------------------------------------------------------------------------/
/
```

```
    // Issue an appropriate glDraw*() command.
    //-----------------------------------------------------------------------/
/
        glDrawArrays(GL_TRIANGLES, 0, 6);
    //refresh the displayed image
    glfwSwapBuffers (window);
  }

  // close GL context and any other GLFW resources
  glfwTerminate();
  exit(EXIT_SUCCESS);
}
```

Uppgift 1.3

C++

```
/* include statements */

Shaders myShaders("../lab1-3_vs.glsl", "../lab1-3_fs.glsl");
GLfloat x_offset = 0;
GLfloat y_offset = 0;
GLfloat blue_offset = 0;


static void error_callback(int error, const char* description)
{
  std::cerr << description;
}

static void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
        blue_offset += yoffset/100;
        glUniform1f(myShaders.get_modifier_location(), blue_offset);
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
  if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action == GLFW_PRESS)
    glfwSetWindowShouldClose(window, GL_TRUE);

  if ((key == GLFW_KEY_R) && action == GLFW_PRESS)
  {
    //-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-//
    // Reload shaders
    //-------------------------------------------------------------------------
-------------------------------------------------------------------------------
-//
          myShaders.experimental_reload();
  }
}

static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
  glViewport(0, 0, width, height);
}

int main(int argc, char const *argv[])
{
  // start GL context and O/S window using the GLFW helper library

  glfwSetErrorCallback(error_callback);
  if( !glfwInit() )
    exit(EXIT_FAILURE);

  GLFWwindow* window = glfwCreateWindow (640, 480, "Hello Triangle", NULL,
NULL);
  glfwSetKeyCallback(window, key_callback);
  glfwSetScrollCallback(window, scroll_callback);
  glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

  if (!window) {
    glfwTerminate();
    exit(EXIT_FAILURE);
  }
  glfwMakeContextCurrent (window);

  // start GLEW extension handler
  glewExperimental = GL_TRUE;
  glewInit ();

  //--------------------------------------------------------------------------
-----------------------------------------------------------------------------/
/
  // Set up geometry, VBO, VAO
  //--------------------------------------------------------------------------
-----------------------------------------------------------------------------/
/
  static const GLfloat vertex_array[] = {
          -0.5f, -0.5f, -1.0f,
          0.5f, -0.5f, -0.1f,
          -0.5f, 0.5f, -0.0f };

  GLuint VAO;
  glGenVertexArrays(1, &VAO);
  glBindVertexArray(VAO);

  GLuint VBO;
  glGenBuffers(1, &VBO);
  glBindBuffer(GL_ARRAY_BUFFER, VBO);
  glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_array), vertex_array,
GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(
          0,
          3,
          GL_FLOAT,
          GL_FALSE,
          0,
          (void*)0
  );


  myShaders.load();

  while (!glfwWindowShouldClose (window))
  {
    // update other events like input handling
    glfwPollEvents ();
    // clear the drawing surface
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//
    // Issue an appropriate glDraw*() command.
    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//
        glDrawArrays(GL_TRIANGLES, 0, 3);
    glfwSwapBuffers (window);
  }

  // close GL context and any other GLFW resources
  glfwTerminate();
  exit(EXIT_SUCCESS);
}
```

Vertex shader

```
#version 400
in vec3 vp;
out vec3 position;

void main () {
  gl_Position = vec4 (vp, 1.0);
  position = (vp + vec3(1,1, 0)) / 2;
 };
```

Fragment shader

```
#version 400
out vec4 frag_colour;
in vec3 position;

void main () {
  frag_colour = vec4 (position, 1.0);
}
```

Uppgift 1.4

C++

```
/* include statements */

Shaders myShaders("../lab1-4_vs.glsl", "../lab1-4_fs.glsl");
GLfloat x_offset = 0;
GLfloat y_offset = 0;
GLfloat blue_offset = 0;


static void error_callback(int error, const char* description)
{
        std::cerr << description;
}
```

```
static void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
        blue_offset += yoffset / 100;
        glUniform1f(myShaders.get_modifier_location(), blue_offset);
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
        if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action ==
GLFW_PRESS)
                glfwSetWindowShouldClose(window, GL_TRUE);

        if ((key == GLFW_KEY_R) && action == GLFW_PRESS)
        {
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Reload shaders
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                myShaders.experimental_reload();
        }
        if ((key == GLFW_KEY_LEFT) && action == GLFW_PRESS)
        {
                x_offset -= 0.05f;
                glUniform2f(myShaders.get_offset_location(), x_offset,
y_offset);
        }
        if ((key == GLFW_KEY_RIGHT) && action == GLFW_PRESS)
        {
                x_offset += 0.05f;
                glUniform2f(myShaders.get_offset_location(), x_offset,
y_offset);
        }
        if ((key == GLFW_KEY_UP) && action == GLFW_PRESS)
        {
                y_offset += 0.05f;
                glUniform2f(myShaders.get_offset_location(), x_offset,
y_offset);
        }
        if ((key == GLFW_KEY_DOWN) && action == GLFW_PRESS)
        {
                y_offset -= 0.05f;
                glUniform2f(myShaders.get_offset_location(), x_offset,
y_offset);
        }
}

static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
        glViewport(0, 0, width, height);
}

int main(int argc, char const *argv[])
{

        // start GL context and O/S window using the GLFW helper library

        glfwSetErrorCallback(error_callback);
        if (!glfwInit())
                exit(EXIT_FAILURE);
        GLFWwindow* window = glfwCreateWindow(640, 480, "Hello Triangle", NULL,
NULL);
        glfwSetKeyCallback(window, key_callback);
        glfwSetScrollCallback(window, scroll_callback);
        glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

        if (!window) {
                glfwTerminate();
                exit(EXIT_FAILURE);
        }
        glfwMakeContextCurrent(window);

        // start GLEW extension handler
        glewExperimental = GL_TRUE;
        glewInit();

        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        // Set up geometry, VBO, VAO
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        static const GLfloat vertex_array[] = {
                -0.5f, -0.5f, -1.0f,
                0.5f, -0.5f, -0.1f,
                -0.5f, 0.5f, -0.0f };

        GLuint VAO;
        glGenVertexArrays(1, &VAO);
        glBindVertexArray(VAO);

        GLuint VBO;
        glGenBuffers(1, &VBO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_array), vertex_array,
GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(
                0,
                3,
                GL_FLOAT,
                GL_FALSE,
                0,
                (void*)0
        );


        myShaders.load();

        while (!glfwWindowShouldClose(window))
        {
                // update other events like input handling
                glfwPollEvents();
                // clear the drawing surface
                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Issue an appropriate glDraw*() command.
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
```

```
                glDrawArrays(GL_TRIANGLES, 0, 3);
                glfwSwapBuffers(window);
        }

        // close GL context and any other GLFW resources
        glfwTerminate();
        exit(EXIT_SUCCESS);
}
```

Vertex shader

```
#version 400
in vec3 vp;
out vec3 position;
uniform vec2 position_offset;

void main () {
  gl_Position = vec4 (vp, 1.0) + vec4(position_offset, 0, 0);
  position = (vp + vec3(1,1,1)) / 2;
 };
```

Fragment shader

```
#version 400
out vec4 frag_colour;
in vec3 position;
uniform float modifier;
void main () {
  frag_colour = vec4 (position, 1.0) + vec4(0, 0, modifier, 0) ;
}
```

Uppgift 1.5

C++

```
/* include statements */

Shaders myShaders("../lab1-5_vs.glsl", "../lab1-5_fs.glsl");

static void error_callback(int error, const char* description)
{
    std::cerr << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
    if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);

        if ((key == GLFW_KEY_R) && action == GLFW_PRESS)
        {
                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Reload shaders
                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                myShaders.experimental_reload();
        }
}
static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}

int main(int argc, char const *argv[])
{
  // start GL context and O/S window using the GLFW helper library

  glfwSetErrorCallback(error_callback);
  if( !glfwInit() )
    exit(EXIT_FAILURE);

  GLFWwindow* window = glfwCreateWindow (640, 480, "Hello Icosahedron", NULL,
NULL);
  glfwSetKeyCallback(window, key_callback);
  glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

  if (!window) {
        glfwTerminate();
        exit(EXIT_FAILURE);
        }
  glfwMakeContextCurrent (window);

  // start GLEW extension handler
  glewExperimental = GL_TRUE;
  glewInit ();

//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// Set up geometry, VBO, EBO, VAO
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//

  float t = (1.0f + sqrtf(5.0f))*0.25f;
  float points[] = {
          // An icosahedron has 12 vertices
          -0.5, t, 0,
          0.5, t, 0,
          -0.5, -t, 0,
          0.5, -t, 0,
          0, -0.5, t,
          0, 0.5, t,
          0, -0.5, -t,
          0, 0.5, -t,
          t, 0, -0.5,
          t, 0, 0.5,
          -t, 0, -0.5,
          -t, 0, 0.5
  };

  unsigned short faces[] = {
          // ... and 20 triangular faces, defined by these vertex indices:
          0, 11, 5,
          0, 5, 1,
          0, 1, 7,
          0, 7, 10,
          0, 10, 11,
          1, 5, 9,
          5, 11, 4,
          11, 10, 2,
          10, 7, 6,
          7, 1, 8,
          3, 9, 4,
          3, 4, 2,
          3, 2, 6,
          3, 6, 8,
          3, 8, 9,
          4, 9, 5,
          2, 4, 11,
          6, 2, 10,
          8, 6, 7,
          9, 8, 1
  };

  GLuint VAO;
  glGenVertexArrays(1, &VAO);
  glBindVertexArray(VAO);

  GLuint VBO;
  glGenBuffers(1, &VBO);
  glBindBuffer(GL_ARRAY_BUFFER, VBO);
  glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(
          0,
          3,
          GL_FLOAT,
          GL_FALSE,
          0,
          (void*)0
  );

  GLuint EBO;
  glGenBuffers(1, &EBO);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(faces), faces, GL_STATIC_DRAW);


//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
// load and compile shaders  "../lab1-5_vs.glsl" and "../lab1-5_fs.glsl"
//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
// attach and link vertex and fragment shaders into a shader program
//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
  myShaders.load();

  while (!glfwWindowShouldClose (window))
  {

    // update other events like input handling
    glfwPollEvents ();

    // clear the drawing surface
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
// Issue an appropriate glDraw*() command.
//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
        glDrawElements(
                GL_TRIANGLES,
                sizeof(faces) / sizeof(faces[0]),
                GL_UNSIGNED_SHORT,
                (void*)0
        );
    glfwSwapBuffers (window);
  }

  // close GL context and any other GLFW resources
  glfwTerminate();
  exit(EXIT_SUCCESS);
}
```

Vertex shader

```
#version 400
in vec3 vp;
out vec3 position;

void main () {
  gl_Position = vec4 (vec3(vp), 1.0);
  position = vp;
};
```

Fragment shader

```
#version 400
out vec4 frag_colour;
in vec3 position;

vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main () {

        frag_colour = vec4(-position.z);
}
```

Uppgift 1.6

C++

```
/* include statements */

//--------------------------------------------------------------------------------
--------------------------------------------------------------------------------//
// You can store the rotation angles here, for example
//--------------------------------------------------------------------------------
--------------------------------------------------------------------------------//
float g_rotation[2] = { 0, 0 };

Shaders myShaders("../lab1-6_vs.glsl", "../lab1-6_fs.glsl");


void MUL_4x4 (float (*C)[4], const float (*A)[4], const float (*B)[4])
{
  //computes C = A x B
        for (int i = 0; i < 4; ++i)
        {
                for (int j = 0; j < 4; ++j)
                {
                        float sum = 0;
                        for (int k = 0; k < 4; ++k)
                        {
                                sum += A[i][k] * B[k][j];
                        }
                        C[i][j] = sum;
                }
        }
}

void invertMatrix (float (*C)[4], const float (*A)[4])
{
  //computes C = A^(-1) for a transformation matrix

  //The rotation part can be inverted separately from the translation part
  //and the last row is the same

        for (int i = 0; i < 3; ++i)
        {
                for (int j = 0; j < 3; ++j)
                {
                        C[j][i] = A[i][j];
                }
        }
        for (int i = 0; i < 4; ++i)
        {
                C[i][3] = A[i][3] * -1;
                C[3][i] = A[3][i];
        }
}


void checkShaderCompileError(GLint shaderID)
{
  GLint isCompiled = 0;
  glGetShaderiv(shaderID, GL_COMPILE_STATUS, &isCompiled);

  if(isCompiled == GL_FALSE)
  {
    GLint maxLength = 0;
    glGetShaderiv(shaderID, GL_INFO_LOG_LENGTH, &maxLength);

    // The maxLength includes the NULL character
    std::string errorLog;
    errorLog.resize(maxLength);
    glGetShaderInfoLog(shaderID, maxLength, &maxLength, &errorLog[0]);

    std::cout << "shader compilation failed:" << std::endl;
    std::cout << errorLog << std::endl;
    return;
  }
  else
    std::cout << "shader compilation success." << std::endl;

  return;
}
```

```
static void error_callback(int error, const char* description)
{
    std::cerr << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
    if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GL_TRUE);

        if ((key == GLFW_KEY_R) && action == GLFW_PRESS)
        {
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Reload shaders
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                myShaders.experimental_reload();
        }

//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// Update rotation angle here, for example
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//

        if ((key == GLFW_KEY_RIGHT) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[1] += 0.10;
    }
        if ((key == GLFW_KEY_LEFT) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[1] -= 0.10;
        }
        if ((key == GLFW_KEY_UP) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[0] += 0.10;
        }
        if ((key == GLFW_KEY_DOWN) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[0] -= 0.10;
        }
}

static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}


int main(int argc, char const *argv[])
{
  // start GL context and O/S window using the GLFW helper library

  glfwSetErrorCallback(error_callback);
  if( !glfwInit() )
```

```
    exit(EXIT_FAILURE);

  GLFWwindow* window = glfwCreateWindow (800, 600, "Hello Icosahedron", NULL,
NULL);
  glfwSetKeyCallback(window, key_callback);
  glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

  int w_height = 800;
  int w_width = 800;

  if (!window) {
        glfwTerminate();
        exit(EXIT_FAILURE);
        }
  glfwMakeContextCurrent (window);

  // start GLEW extension handler
  glewExperimental = GL_TRUE;
  glewInit ();

  // tell GL to only draw onto a pixel if the shape is closer to the viewer
  glEnable (GL_DEPTH_TEST); // enable depth-testing
  glDepthFunc (GL_LESS); // depth-testing interprets a smaller value as "closer"

//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
// Set up geometry, VBO, EBO, VAO
//--------------------------------------------------------------------------------
------------------------------------------------------------------------------//
  float t = (1.0f + sqrtf(5.0f))*0.25f;
  float points[] = {
          // An icosahedron has 12 vertices
          -0.5, t, 0,
          0.5, t, 0,
          -0.5, -t, 0,
          0.5, -t, 0,
          0, -0.5, t,
          0, 0.5, t,
          0, -0.5, -t,
          0, 0.5, -t,
          t, 0, -0.5,
          t, 0, 0.5,
          -t, 0, -0.5,
          -t, 0, 0.5
  };

  unsigned short faces[] = {
          // ... and 20 triangular faces, defined by these vertex indices:
          0, 11, 5,
          0, 5, 1,
          0, 1, 7,
          0, 7, 10,
          0, 10, 11,
          1, 5, 9,
          5, 11, 4,
          11, 10, 2,
          10, 7, 6,
          7, 1, 8,
          3, 9, 4,
          3, 4, 2,
          3, 2, 6,
          3, 6, 8,
          3, 8, 9,
          4, 9, 5,
          2, 4, 11,
          6, 2, 10,
          8, 6, 7,
          9, 8, 1
  };

  GLuint VAO;
  glGenVertexArrays(1, &VAO);
  glBindVertexArray(VAO);

  GLuint VBO;
  glGenBuffers(1, &VBO);
  glBindBuffer(GL_ARRAY_BUFFER, VBO);
  glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
  glEnableVertexAttribArray(0);
  glVertexAttribPointer(
          0,
          3,
          GL_FLOAT,
          GL_FALSE,
          0,
          (void*)0
  );

  GLuint EBO;
  glGenBuffers(1, &EBO);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(faces), faces, GL_STATIC_DRAW);
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// load and compile shaders  "../lab1-6_vs.glsl" and "../lab1-6_fs.glsl"
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// attach and link vertex and fragment shaders into a shader program
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
  myShaders.load();

  float n=1.0;
  float f=100.0;
  float a;
  float b;

  while (!glfwWindowShouldClose (window))
  {
          glfwGetFramebufferSize(window, &w_width, &w_height); //you might need
this for correcting the aspect ratio

//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// Define the projection matrix, rotation matrices, model matrix, etc. The
variable names and code structure is a simple suggestion, you may improve on it!
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//


          a = -(f + n) / (f - n);
          b = -(2 * f * n) / (f - n);
          GLfloat projectionMatrix[4][4] = {
                  { (double)w_height/w_width, 0, 0, 0 },
                  { 0, 1, 0, 0 },
                  { 0, 0, a, b },
                  { 0, 0, -1, 0 }
          };

          GLfloat rotate_y[4][4] = {
                  { cos(g_rotation[1]), 0, -sin(g_rotation[1]), 0 },
                  { 0, 1, 0, 0 },
                  { sin(g_rotation[1]), 0, cos(g_rotation[1]), 0 },
                  { 0, 0, 0, 1 }
          };
          GLfloat rotate_x[4][4] = {
                  { 1, 0, 0, 0 },
                  { 0, cos(g_rotation[0]), -sin(g_rotation[0]), 0 },
                  { 0, sin(g_rotation[0]), cos(g_rotation[0]), 0 },
                  { 0, 0, 0, 1 }
          };
          GLfloat modelMatrix[4][4];
          MUL_4x4(modelMatrix, rotate_x, rotate_y);

          GLfloat viewMatrix[4][4] = {
                  { 1, 0, 0, 0 },
                  { 0, 1, 0, 0 },
                  { 0, 0, 1, 2 },
                  { 0, 0, 0, 1 }
          };
          GLfloat inverseViewMatrix[4][4];
          invertMatrix(inverseViewMatrix, viewMatrix);

          GLfloat modelViewMatrix[4][4];
          MUL_4x4(modelViewMatrix, inverseViewMatrix, modelMatrix);

          GLfloat modelViewProjectionMatrix[4][4];
          MUL_4x4(modelViewProjectionMatrix, projectionMatrix, modelViewMatrix);



    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//
    // Send your modelViewProjection matrix to your vertex shader as a uniform
varable
    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//

         glUniformMatrix4fv(glGetUniformLocation(myShaders.get_shader_program(),
"modelViewProjectionMatrix"), 1, GL_TRUE, &modelViewProjectionMatrix[0][0]);

    // update other events like input handling
    glfwPollEvents ();

    // clear the drawing surface
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//
    // Issue an appropriate glDraw*() command.
    //--------------------------------------------------------------------------
--------------------------------------------------------------------------------
-//
        glDrawElements(
                GL_TRIANGLES,
                sizeof(faces) / sizeof(faces[0]),
                GL_UNSIGNED_SHORT,
                (void*)0
        );
    glfwSwapBuffers (window);
  }

  // close GL context and any other GLFW resources
  glfwTerminate();
  exit(EXIT_SUCCESS);
}
```

Vertex shader

```
#version 400

layout(location=0) in vec4 vp;
uniform mat4 modelViewProjectionMatrix;
out vec4 position;

void main () {

//-------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// Apply the model, view and projection transform to vertex positions and
forward the position to the fragment shader using an appropriate "out" variable
//-------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
  position = modelViewProjectionMatrix * vp;
  gl_Position = modelViewProjectionMatrix * vp;

};
```

Fragment shader

```
#version 400
out vec4 frag_colour;
in vec4 position;

vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main () {
        vec3 z = hsv2rgb(vec3((position.z+1), 1, 1));
        frag_colour = vec4(z.xyz,1);
}
```

Uppgift 1.7

C++

```
/*include statements */

//-------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// You can store the rotation angles here, for example
```

```
//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
float g_rotation[2] = { 0, 0 };

Shaders myShaders("../lab1-7_vs.glsl", "../lab1-7_fs.glsl");


void MUL_4x4(float(*C)[4], const float(*A)[4], const float(*B)[4])
{
        //computes C = A x B
        for (int i = 0; i < 4; ++i)
        {
                for (int j = 0; j < 4; ++j)
                {
                        float sum = 0;
                        for (int k = 0; k < 4; ++k)
                        {
                                sum += A[i][k] * B[k][j];
                        }
                        C[i][j] = sum;
                }
        }
}

void invertMatrix(float(*C)[4], const float(*A)[4])
{
        //computes C = A^(-1) for a transformation matrix

        //The rotation part can be inverted separately from the translation part
        //and the last row is the same

        for (int i = 0; i < 3; ++i)
        {
                for (int j = 0; j < 3; ++j)
                {
                        C[j][i] = A[i][j];
                }
        }
        for (int i = 0; i < 4; ++i)
        {
                C[i][3] = A[i][3] * -1;
                C[3][i] = A[3][i];
        }
}


void checkShaderCompileError(GLint shaderID)
{
        GLint isCompiled = 0;
        glGetShaderiv(shaderID, GL_COMPILE_STATUS, &isCompiled);

        if (isCompiled == GL_FALSE)
        {
                GLint maxLength = 0;
                glGetShaderiv(shaderID, GL_INFO_LOG_LENGTH, &maxLength);

                // The maxLength includes the NULL character
                std::string errorLog;
                errorLog.resize(maxLength);
                glGetShaderInfoLog(shaderID, maxLength, &maxLength,
&errorLog[0]);

                std::cout << "shader compilation failed:" << std::endl;
                std::cout << errorLog << std::endl;
                return;
        }
        else
                std::cout << "shader compilation success." << std::endl;

        return;
}



static void error_callback(int error, const char* description)
{
        std::cerr << description;
}

static void key_callback(GLFWwindow* window, int key, int scancode, int action,
int mods)
{
        if ((key == GLFW_KEY_ESCAPE || key == GLFW_KEY_Q) && action ==
GLFW_PRESS)
                glfwSetWindowShouldClose(window, GL_TRUE);

        if ((key == GLFW_KEY_R) && action == GLFW_PRESS)
        {
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Reload shaders
                //--------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                myShaders.experimental_reload();
        }

        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        // Update rotation angle here, for example
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//

        if ((key == GLFW_KEY_RIGHT) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[1] += 0.10;
        }
        if ((key == GLFW_KEY_LEFT) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[1] -= 0.10;
        }
        if ((key == GLFW_KEY_UP) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[0] += 0.10;
        }
        if ((key == GLFW_KEY_DOWN) && ((action == GLFW_PRESS) || action ==
GLFW_REPEAT))
        {
                g_rotation[0] -= 0.10;
        }
}
```

```
static void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
        glViewport(0, 0, width, height);
}


int main(int argc, char const *argv[])
{
        // start GL context and O/S window using the GLFW helper library

        glfwSetErrorCallback(error_callback);
        if (!glfwInit())
                exit(EXIT_FAILURE);

        GLFWwindow* window = glfwCreateWindow(800, 600, "Hello Icosahedron",
NULL, NULL);
        glfwSetKeyCallback(window, key_callback);
        glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

        int w_height = 800;
        int w_width = 800;

        if (!window) {
                glfwTerminate();
                exit(EXIT_FAILURE);
        }
        glfwMakeContextCurrent(window);

        // start GLEW extension handler
        glewExperimental = GL_TRUE;
        glewInit();

        // tell GL to only draw onto a pixel if the shape is closer to the
viewer
        glEnable(GL_DEPTH_TEST); // enable depth-testing
        glDepthFunc(GL_LESS); // depth-testing interprets a smaller value as
"closer"

                                                  //----------------------------
--------------------------------------------------------------------------------
-----------------------------------------------//
                                                  // Set up geometry, VBO, EBO,
VAO
                                                  //----------------------------
--------------------------------------------------------------------------------
-----------------------------------------------//
        float t = (1.0f + sqrtf(5.0f))*0.25f;
        float points[] = {
                // An icosahedron has 12 vertices
                -0.5, t, 0,
                0.5, t, 0,
                -0.5, -t, 0,
                0.5, -t, 0,
                0, -0.5, t,
                0, 0.5, t,
                0, -0.5, -t,
                0, 0.5, -t,
                t, 0, -0.5,
                t, 0, 0.5,
                -t, 0, -0.5,
                -t, 0, 0.5
        };
        unsigned short faces[] = {
                // ... and 20 triangular faces, defined by these vertex indices:
                0, 11, 5,
                0, 5, 1,
                0, 1, 7,
                0, 7, 10,
                0, 10, 11,
                1, 5, 9,
                5, 11, 4,
                11, 10, 2,
                10, 7, 6,
                7, 1, 8,
                3, 9, 4,
                3, 4, 2,
                3, 2, 6,
                3, 6, 8,
                3, 8, 9,
                4, 9, 5,
                2, 4, 11,
                6, 2, 10,
                8, 6, 7,
                9, 8, 1
        };

        GLuint VAO;
        glGenVertexArrays(1, &VAO);
        glBindVertexArray(VAO);

        GLuint VBO;
        glGenBuffers(1, &VBO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(
                0,
                3,
                GL_FLOAT,
                GL_FALSE,
                0,
                (void*)0
        );

        GLuint EBO;
        glGenBuffers(1, &EBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(faces), faces,
GL_STATIC_DRAW);
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        // load and compile shaders  "../lab1-6_vs.glsl" and "../lab1-6_fs.glsl"
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        // attach and link vertex and fragment shaders into a shader program
        //----------------------------------------------------------------------
--------------------------------------------------------------------------------
-----//
        myShaders.load();

        float n = 1.0;
        float f = 100.0;
        float a;
        float b;

        while (!glfwWindowShouldClose(window))
        {
                glfwGetFramebufferSize(window, &w_width, &w_height); //you might
need this for correcting the aspect ratio


//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//

// Define the projection matrix, rotation matrices, model matrix, etc. The
variable names and code structure is a simple suggestion, you may improve on it!

//------------------------------------------------------------------------------
-----------------------------------------------------------------------------//


                auto projectionMatrix = glm::perspective(90.0f,
(float(w_width)/w_height), n, f);

                auto rotate_y = glm::rotate(g_rotation[1], glm::vec3(0, 1, 0));
                auto rotate_x = glm::rotate(g_rotation[0], glm::vec3(1, 0, 0));

                auto modelMatrix = rotate_x * rotate_y;

                auto inverseViewMatrix =
glm::inverse(glm::translate(glm::vec3(0, 0, 2)));

                auto modelViewProjectionMatrix = projectionMatrix *
inverseViewMatrix * modelMatrix;


                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Send your modelViewProjection matrix to your vertex shader as
a uniform varable
                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//


glUniformMatrix4fv(glGetUniformLocation(myShaders.get_shader_program(),
"modelViewProjectionMatrix"), 1, 0, glm::value_ptr(modelViewProjectionMatrix));

                // update other events like input handling
                glfwPollEvents();

                // clear the drawing surface
                glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                // Issue an appropriate glDraw*() command.
                //----------------------------------------------------------------
--------------------------------------------------------------------------------
-------------//
                glDrawElements(
                        GL_TRIANGLES,
                        sizeof(faces) / sizeof(faces[0]),
                        GL_UNSIGNED_SHORT,
                        (void*)0
                );
                glfwSwapBuffers(window);
        }

        // close GL context and any other GLFW resources
        glfwTerminate();
        exit(EXIT_SUCCESS);
}
```

Vertex shader

```
#version 400

layout(location=0) in vec4 vp;
uniform mat4 modelViewProjectionMatrix;
out vec4 position;

void main () {

//--------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
// Apply the model, view and projection transform to vertex positions and
forward the position to the fragment shader using an appropriate "out" variable
//--------------------------------------------------------------------------------
-----------------------------------------------------------------------------//
  position = modelViewProjectionMatrix * vp;
  gl_Position = modelViewProjectionMatrix * vp;

};
```

Fragment shader

```
#version 400
out vec4 frag_colour;
in vec4 position;

vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

void main () {
        vec3 z = hsv2rgb(vec3((position.z+1)/2, 1, 1));
        frag_colour = vec4(z.xyz,1);
}
```

Custom class Shaders.h

```
#pragma once
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>
#include "readfile.hpp"
class Shaders
{
        private:
                std::string vertex_shader_str;
                std::string fragment_shader_str;
                std::string vertex_filename;
                std::string fragment_filename;
                GLuint shader_program;
                GLuint vs;
                GLuint fs;
                GLint offset_location = 0;
                GLint modifier_location = 0;

                void update_locations()
                {
                        offset_location =
glGetUniformLocation(get_shader_program(), "position_offset");
                        modifier_location =
glGetUniformLocation(get_shader_program(), "modifier");
                }
        public:
                Shaders(){}
                Shaders(std::string vertex_filename, std::string
fragment_filename)
                {
                        this->vertex_filename = vertex_filename;
                        this->fragment_filename = fragment_filename;
                }
                GLint get_offset_location() { return offset_location; }
                GLint get_modifier_location() { return modifier_location; }

                void load()
                {
                        vertex_shader_str = readFile(vertex_filename.c_str());
                        fragment_shader_str =
readFile(fragment_filename.c_str());
                        const char *vertex_shader_src =
vertex_shader_str.c_str();
                        const char *fragment_shader_src =
fragment_shader_str.c_str();
                        vs = glCreateShader(GL_VERTEX_SHADER);
                        glShaderSource(vs, 1, &vertex_shader_src, NULL);
                        glCompileShader(vs);
                        fs = glCreateShader(GL_FRAGMENT_SHADER);
                        glShaderSource(fs, 1, &fragment_shader_src, NULL);
                        glCompileShader(fs);
                        shader_program = glCreateProgram();
                        glAttachShader(shader_program, fs);
                        glAttachShader(shader_program, vs);
                        glLinkProgram(shader_program);
                        glDeleteShader(vs);
                        glDeleteShader(fs);

                        glUseProgram(shader_program);

                        update_locations();
                }

                void experimental_reload()
                {
                        glDetachShader(shader_program, vs);
                        glDetachShader(shader_program, fs);
                        vertex_shader_str = readFile(vertex_filename.c_str());
                        fragment_shader_str =
readFile(fragment_filename.c_str());
                        const char *vertex_shader_src =
vertex_shader_str.c_str();
```

```
                        const char *fragment_shader_src =
fragment_shader_str.c_str();
                        vs = glCreateShader(GL_VERTEX_SHADER);
                        glShaderSource(vs, 1, &vertex_shader_src, NULL);
                        glCompileShader(vs);
                        fs = glCreateShader(GL_FRAGMENT_SHADER);
                        glShaderSource(fs, 1, &fragment_shader_src, NULL);
                        glCompileShader(fs);
                        glAttachShader(shader_program, fs);
                        glAttachShader(shader_program, vs);
                        glLinkProgram(shader_program);
                        glDeleteShader(vs);
                        glDeleteShader(fs);
                }

                GLuint get_shader_program()
                {
                        return shader_program;
                }

                void reload()
                {
                        glDeleteProgram(shader_program);
                        load();
                }
        };
```