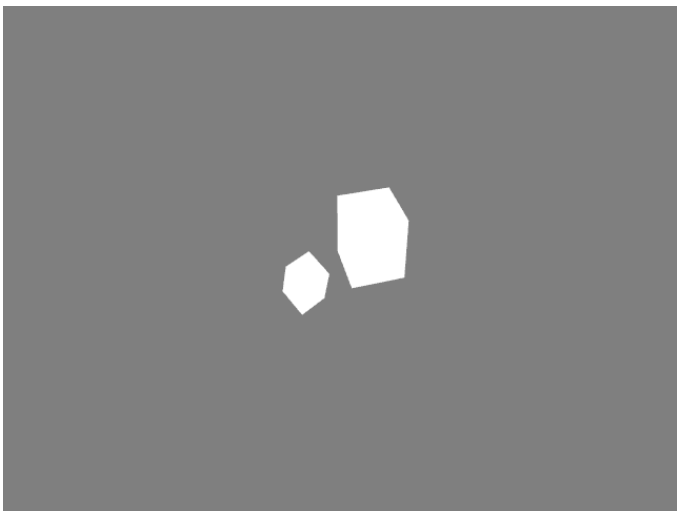
gVirtualXRay – Tutorial 01: Creating a Window and an OpenGL Core Profile 3.2 Context Using GLFW

Dr Franck P. Vidal

2nd September 2014



Contents

List of Figures

Listings

1 Introduction

The complete source code of this tutorial is available in Appendix A and on the Subversion (SVN) repository at `example/tutorial_01_glfw/tutorial_01_glfw.cxx`. It can be downloaded here: https://sourceforge.net/p/gvirtualxray/code/HEAD/tree/trunk/tutorials/tutorial_01_glfw/tutorial_01_glfw.cxx. It shows how to create a window with GLFW [1] and attach an OpenGL context to it. Two cubes are displayed (see Figure 1). They both rotate automatically. The rendering

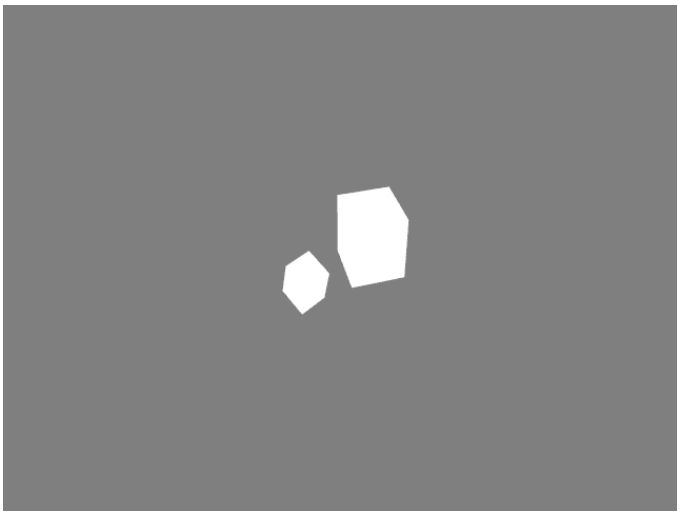


Figure 1: Screen capture of the tutorial.

makes use of OpenGL Shading Language (GLSL) as the fix rendering pipeline and direct rendering are both depreciated in any modern computer graphics applications.

It is an introductory tutorial, for more details, the reader may refer to the code (it is well documented) and the Doxygen [2] documentation of the project [3] .

The tutorial is organised as follows:

- Section 2 shows the header files to include.
- Section 3 shows some of the name spaces that can be included to lighten the code.
- Section 4 shows the global variables that are used.
- Section 5 what typical functions have to be declared in a basic GLFW program.
- Section 6 shows how to initialise GLFW.
- Section 7 shows how to initialise GLEW [4].
- Section 8 shows how to initialise OpenGL.
- How to load 3D objects is explained in Section 9
- They are displayed in Section 10.
- Section 11 shows the idle callback.
- Section 12 shows the routine that is called when the program shuts.
- Section 13 shows what is done when the size of the frame buffer changes, i.e. when the window is resized.
- Section 14 shows how to handle the keyboard.

[1] http://www.glfw.org/
[2] http://www.doxygen.org/
[3] http://gvirtualxray.sourceforge.net/documentation.php
[4] glew.sourceforge.net/

- Section 15 shows how to handle the scroll button.
- Section 16 deals with the error callback.
- Section 17 gives a preview of what the next tutorial will be about.
- Appendix A shows the source code of this tutorial.

2 Header inclusion

Listing 1 shows i) the macros that have to be defines to include OpenGL core profile hears and ii) the header files that need to be included to build a simple program based on the GLFW library:

- `GL/glew.h` is the GLEW header file. It can be found at http://glew.sourceforge.net/. GLEW is used to ensure that there is no undefined references when the Windows executable is created.
- `GL3_PROTOTYPES` is a macro that ensures that we are using opengl's core profile only. It has to be included before any other OpenGL header inclusion (Mac users only).
- `GL_GLEXT_PROTOTYPES` is a macro that ensures that we are using OpenGL's core profile only. It has to be included before any other OpenGL header inclusion (Windows or Linux users).
- `GLFW_INCLUDE_GLCOREARB` is a macro that tells GLFW to include the OpenGL core profile header. It has to be included before any other OpenGL header inclusion (all users).
- `glfw3.h` is the main GLFW header file.
- `iostream` is used for output streams.
- `exception` is used for C++ exceptions.
- `cstdlib` defines return status (`EXIT_SUCCESS` and `EXIT_FAILURE`).
- `gVirtualXRay/Types.h` defines new types, e.g `RATIONAL_NUMBER`, `VEC2`, `VEC3` and `MATRIX4` to name a few.
- `gVirtualXRay/Units.h` defines units such as metre, kilometre, electronvolt, kiloelectron volt, gram, kilogram, etc.
- `gVirtualXRay/OpenGLUtilities.h` defines some utility functions about OpenGL, e.g. matrix stacks, how to set the projection matrix, etc.
- `gVirtualXRay/PolygonMesh.h` corresponds to a class used to handle three-dimensional (3D) triangle meshes.
- `gVirtualXRay/Shader.h` corresponds to a class that handles (GLSL) programs.
- `buildCube.h` is used to create the triangle mesh of a cube.

```
1  //////////////////////////////////////////////////////////////////////////////
2  //  Include
3  //////////////////////////////////////////////////////////////////////////////
4  // Fix undefined references on Windows
5  #if defined(_WIN32) || defined(_WIN64)
6  #include <GL/glew.h>
7  #endif
8
9  // Ensure we are using opengl's core profile only
10 #ifdef __APPLE__
11 #define GL3_PROTOTYPES 1
12 #else
13 #define GL_GLEXT_PROTOTYPES 1
14 #endif
15
```

```
16 #define GLFW_INCLUDE_GLCOREARB 1 // Tell GLFW to include the OpenGL core profile header
17
18 #include <GLFW/glfw3.h> // Create an OpenGL context and attach a window to it
19
20 #include <iostream>   // Print error messages in the console
21 #include <exception>  // Catch C++ exception
22 #include <cstdlib>    // Define return status (EXIT_SUCCESS and EXIT_FAILURE)
23
24 // Define new types, e.g.~RATIONAL_NUMBER, VEC2, VEC3 and MATRIX4 to name a few
25 #include "gVirtualXRay/Types.h"
26
27 // Define units such as metre, kilometre, electronvolt, gram, kilogram, etc.
28 #include "gVirtualXRay/Units.h"
29
30 // Some utility functions about OpenGL, e.g. matrix stacks,
31 // how to set the projection matrix, etc.
32 #include "gVirtualXRay/OpenGLUtilities.h"
33
34 #include "gVirtualXRay/PolygonMesh.h"  // Handle 3D triangle meshes
35 #include "gVirtualXRay/Shader.h"       // Handle GLSL programs
36
37 #include "buildCube.h" // Create the triangle mesh of a cube
```

Listing 1: Header inclusion.

3 Name Spaces

Listing 2 shows the name spaces that can be selected:

- `gVirtualXRay` includes graphics elements such as `PolygonMesh` and utilities such as `Exception`.
- `std` is used for output streams and exceptions.

```
1 //========================================================================
2 //  Name space
3 //========================================================================
4 using namespace gVirtualXRay;
5 using namespace std;
```

Listing 2: Name spaces.

4 Global Variables

Listing 3 shows the global variables that are used:

- `GLsizei g_window_width` keeps track of the window width.
- `GLsizei g_window_height` keeps track of the window height.
- `GLFWwindow* g_p_window_id` is the GLFW window ID.
- `Shader g_display_shader` is the shader program used to display the 3D scene
- `PolygonMesh g_polygon_mesh_1` is the polygon mesh of the first 3D object.
- `PolygonMesh g_polygon_mesh_2` is the polygon mesh of the second 3D object.
- `MATRIX4 g_object_1_rotation_matrix` corresponds to the transformation matrix of the first 3D object.
- `MATRIX4 g_object_2_rotation_matrix`corresponds to the transformation matrix of the second 3D object.

- `vector<double> g_p_vertex_set_1` is an array containing the vertices of the first 3D object.
- `vector<unsigned char> g_p_index_set_1` is an array containing vertex indices to build triangles from `g_p_vertex_set_1`.
- `vector<float> g_p_vertex_set_2` is an array containing the vertices of the second 3D object. Note that no index is used in this case.
- `RATIONAL_NUMBER g_zoom` controls the zoom.
- `const GLchar* g_vertex_shader` is the source code of the vertex shader.
- `const GLchar* g_fragment_shader` is the source code of the fragment shader.

```
//****************************************************************************
//  Global variables
//****************************************************************************
// Keep track of the window width
GLsizei g_window_width(640);

// Keep track of the window height
GLsizei g_window_height(480);

// GLFW window ID
GLFWwindow* g_p_window_id(0);

// Shader program used to display the 3D scene
Shader g_display_shader;

// 3D objects as VAOs and VBOs
PolygonMesh g_polygon_mesh_1;
PolygonMesh g_polygon_mesh_2;

// Transformation matrices
MATRIX4 g_object_1_rotation_matrix;
MATRIX4 g_object_2_rotation_matrix;

// Geometric data
vector<double> g_p_vertex_set_1;
vector<unsigned char> g_p_index_set_1;
vector<float> g_p_vertex_set_2;

// Control the zoom
RATIONAL_NUMBER g_zoom(50.0 * cm);

// Vertex shader
const GLchar* g_vertex_shader = "\
\n#version 150\n \
\n \
in vec3 in_Vertex;\n \
\n \
uniform mat4 g_projection_matrix;\n \
uniform mat4 g_modelview_matrix;\n \
\n \
void main(void)\n \
{\n \
    gl_Position = g_projection_matrix * g_modelview_matrix * vec4(in_Vertex, 1.0);\n \
}\n \
";

// Fragment shader
const GLchar* g_fragment_shader = "\
\n#version 150\n \
precision highp float;\n \
\n \
out vec4 fragColor;\n \
void main(void)\n \
```

```
{\n \
    fragColor = vec4(1.0, 1.0, 1.0, 1.0);\n \
}\n \
";
```

Listing 3: Global variables.

5 Function Declarations

Listing 4 shows the typical functions that have to be declared in a basic GLFW program:

- `initGLFW` is used i) to initialise GLFW, ii) to create an OpenGL context, iii) to create a window, and iv) attach the OpenGL context to the window (see Section 6).
- `initGLEW` is used to initialise GLEW (see Section 7).
- `initGL` is used to initialise some states of OpenGL, e.g. the background colour and enable the Z-buffer (see Section 8).
- `load3DObjects` loads the 3D geometry of two cubes (see Section 9).
- `displayCallback` render the two cubes on the screen (see Section 10).
- `idleCallback` is an idle callback (see Section 11). It is called once every event loop and can be used to perform animation.
- `quitCallback` is call when the program terminates (see Section 12). It closes the window and cleans up GLFW.
- `framebufferSizeCallback` is called every time the frame buffer size changes (see Section 13). It initialises the viewport size and the projection matrix.
- `keyCallback` is called every time a key is pressed or released on the keyboard (see Section 14). It can be used to close the window when the `escape` key is pressed.
- `scrollCallback` processes the mouse scroll button. It can be used to zoom in and out (see Section 15).
- `errorCallback` is called to throw an exception when GLFW generates an error (see Section 16).

```
//********************************************************************************
//  Function declaration
//********************************************************************************
void initGLFW();
void initGLEW();
void initGL();
void load3DObjects();
void displayCallback();
void idleCallback();
void quitCallback();
void framebufferSizeCallback(GLFWwindow* apWindow, int aWidth, int aHeight);
void keyCallback(GLFWwindow* apWindow, int aKey, int aScanCode, int anAction, int
    aModifierKey);
void scrollCallback(GLFWwindow* apWindow, double xoffset, double yoffset);
void errorCallback(int error, const char* description);
```

Listing 4: Function declarations.

6 Initialise GLFW

Listing 5 shows how to initialise GLFW to create an OpenGL core profile 3.2 context and how to attach a window to it. There are nime main steps:

- Register an error callback
- Initialise the GLFW library.
- If it cannot be initialised, an exception is thrown.
- Enable OpenGL 3.2 (this is compulsory)
- Enable anti-aliasing (this is optional).
- Create a windowed mode window and its OpenGL context
- If the window has not been created, an exception is thrown.
- Make the window's context current
- Register GLFW callbacks

```
1  //--------------
2  void initGLFW()
3  //--------------
4  {
5      // Set an error callback
6      glfwSetErrorCallback(errorCallback);
7
8      // Initialize GLFW
9      if (!glfwInit())
10     {
11         throw Exception(__FILE__, __FUNCTION__, __LINE__,
12                 "ERROR: cannot initialise GLFW.");
13     }
14
15     // Enable OpenGL 3.2 if possible
16     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
17     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
18     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
19     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
20
21     // Enable anti-aliasing
22     glfwWindowHint(GLFW_SAMPLES, 4);
23
24     // Create a windowed mode window and its OpenGL context
25     g_p_window_id = glfwCreateWindow(g_window_width, g_window_height,
26             "gVirtualXRay -- Tutorial 01", NULL, NULL);
27
28     // The window has not been created
29     if (!g_p_window_id)
30     {
31         throw Exception(__FILE__, __FUNCTION__, __LINE__,
32                 "ERROR: cannot create a GLFW windowed mode window and its OpenGL context.")
       ;
33     }
34
35     // Make the window's context current
36     glfwMakeContextCurrent(g_p_window_id);
37
38     // Register GLFW callbacks
39     glfwSetFramebufferSizeCallback(g_p_window_id, framebufferSizeCallback);
40     glfwSetKeyCallback(g_p_window_id, keyCallback);
41     glfwSetScrollCallback(g_p_window_id, scrollCallback);
42 }
```

Listing 5: Initialise GLFW.

7 Initialise GLEW

Listing 6 shows how to initialise GLEW when a Windows platform is used.

```
//--------------
void initGLEW()
//--------------
{
#ifdef _WIN32
  GLenum err = glewInit();
  if (GLEW_OK != err)
  {
    std::stringstream error_message;
    error_message << "ERROR: cannot initialise GLEW:\t" << glewGetErrorString(err);

        throw Exception(__FILE__, __FUNCTION__, __LINE__, error_message.str());
  }
#endif
}
```

Listing 6: Initialise GLEW.

8 Initialise OpenGL

Listing 7 shows how to initialise some OpenGL states (Z-buffer, and background colour) and check OpenGL's error status.

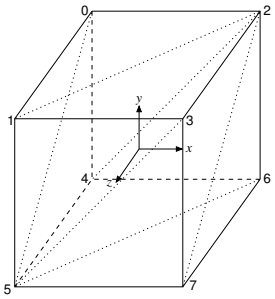
```
//------------
void initGL()
//------------
{
    // Enable the Z-buffer
    glEnable(GL_DEPTH_TEST);

    // Set the background colour
    glClearColor(0.5, 0.5, 0.5, 1.0);

    // Check if any OpenGL error has occurred.
    // If any has, an exception is thrown
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
}
```

Listing 7: Initialise OpenGL.

9 Load 3D Objects

Listing 8 shows how to create 3D objects. It implements two cubes. The first one makes use of a vertex list and an index list (see Figure 2). The second cube only makes use of a vertex list. Vertices are repeated several times. This is because each vertex is shared by different triangles. The second cube is therefore much bigger in term of memory usage. Listing 8 shows that our implementation supports both type of topology.

- `buildCube(length_1, cube_centre, g_p_vertex_set_1, g_p_index_set_1)` creates the data required to model a cube using vertex data and index data. Its length is `length_1` and it is centred on `cube_centre` (0, 0, 0).
- `buildCube(length_2, cube_centre, g_p_vertex_set_2)` creates the data of another cube, but using vertex data only. Its length is `length_2` and it is also centred on `cube_centre`.



(a) Geometry.

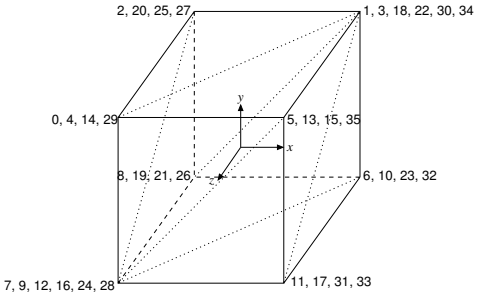
	Vertex List		
Vertex ID	**x**	**y**	**z**
0	-2.5 cm	2.5 cm	-2.5 cm
1	-2.5 cm	2.5 cm	2.5 cm
2	2.5 cm	2.5 cm	-2.5 cm
3	2.5 cm	2.5 cm	2.5 cm
4	-2.5 cm	-2.5 cm	-2.5 cm
5	-2.5 cm	-2.5 cm	2.5 cm
6	2.5 cm	-2.5 cm	-2.5 cm
7	2.5 cm	-2.5 cm	2.5 cm

(b) Vertex list.

	Face List		
	i	**j**	**k**
Top face	1	2	0
Top face	2	1	3
Bottom face	6	5	4
Bottom face	5	6	7
Front face	5	3	1
Front face	3	5	7
Back face	2	4	0
Back face	4	2	6
Left face	5	0	4
Left face	0	5	1
Right face	2	7	6
Right face	7	2	3

(c) Index list.

Figure 2: Topology of the first cube.



(a) Geometry.

	Vertex List		
Vertex ID	**x**	**y**	**z**
0	-2.5 cm	2.5 cm	2.5 cm
1	2.5 cm	2.5 cm	-2.5 cm
⋮	⋮	⋮	⋮
34	2.5 cm	2.5 cm	-2.5 cm
35	2.5 cm	2.5 cm	2.5 cm

(b) Vertex list.

Figure 3: Topology of the second cube.

- `g_polygon_mesh_1.setExternalData(GL_TRIANGLES, &g_p_vertex_set_1, &g_p_index_set_1, true, GL_STATIC_DRAW)` loads both the vertex (`&g_p_vertex_set_1`) and index (`&g_p_index_set_1`) data of the geometry. `GL_TRIANGLES` indicates that the mesh corresponds to a triangle mesh. `true` means that the vertex buffer object (VBO) should be created instantly if possible. `GL_STATIC_DRAW` is used because the (VBO) data will be set once and used many times.
- `g_polygon_mesh_2.setExternalData(GL_TRIANGLES, &g_p_vertex_set_2, true, GL_STATIC_DRAW)` loads the vertex (`&g_p_vertex_set_2`) data of the geometry. There is no index data for this mesh.

```
//--------------------
void load3DObjects()
//--------------------
{
    // Centre of the cubes
    VEC3 cube_centre(0, 0, 0);

    // Size of the cubes
    RATIONAL_NUMBER length_1( 5.0 * cm);
    RATIONAL_NUMBER length_2(10.0 * cm);

    // Create the cube using vertex data and index data
    buildCube(length_1, cube_centre, g_p_vertex_set_1, g_p_index_set_1);

    // Create the cube using vertex data only
    buildCube(length_2, cube_centre, g_p_vertex_set_2);
```

```
17
18      // Set geometry (using VAOs and VBOs)
19      g_polygon_mesh_1.setExternalData(GL_TRIANGLES,
20              &g_p_vertex_set_1,
21              &g_p_index_set_1,
22              true,
23              GL_STATIC_DRAW);
24
25      g_polygon_mesh_2.setExternalData(GL_TRIANGLES,
26              &g_p_vertex_set_2,
27              true,
28              GL_STATIC_DRAW);
29  }
```

Listing 8: Create 3D objects.

10 Display the 3D Scene

Listing 9 shows how to display the 3D objects.

- `// Clear the buffers glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` clears both the fragment and Z buffers.
- `pushShaderProgram()` adds the current shader program to the shader stack. It may become handy when multiple shaders are used.
- `g_display_shader.enable()` enables the given shader program.
- Its unique OpenGL ID can be retrieved with `g_display_shader.getProgramHandle()`.
- The status of the frame buffer object (FBO) can be checked with `checkFBOErrorStatus(__FILE__, __FUNCTION__, __LINE__)`. If an error had occurred, then an exception will be thrown.
- The status of the OpenGL's error flag can be checked with `checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__)`. If an error had occurred, then an exception will be thrown.
- `g_current_projection_matrix` is the current projection matrix. `g_current_modelview_matrix` is the current modelling-viewing matrix. Section 13 shows how it can be set. Using a shader program, when the fixed pipeline functions are disabled (as in modern OpenGL implementation), the programmer has to supply the projection and modelling-viewing matrices to the shader program. This is what Lines 22 to 30 are taking care of in Listing 9.
- To store the current transformation matrices, `pushModelViewMatrix()` and `pushProjectionMatrix()` are used. They replace the old `glMatrixMode` and `glPushMatrix`, which are no longer available in modern OpenGL.
- To translate the 1st object, write `g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(VEC3(8.0 * cm, 0.0, 0.0))`. This is similar to the old function `glTranslate`, which is now depreciated.
- To rotate this object, write `g_current_modelview_matrix *= g_object_1_rotation_matrix`. This is similar to the old function `glRotate`, which is now depreciated.
- To apply the changes to the shader program, call `applyModelViewMatrix()`.
- The 1st polygon mesh is displayed with `g_polygon_mesh_1.display()`.
- The transformation matrices are restored from the stack using `popModelViewMatrix()` and `popProjectionMatrix()`.

- The transformation matrices are stored in the stack using `pushModelViewMatrix()` and `pushProjectionMatrix()`.
- To translate the 2nd object, write `g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(VEC3(-8.0 * cm, 0.0, 0.0))`.
- To rotate this object, write `g_current_modelview_matrix *= g_object_2_rotation_matrix`.
- To apply the changes to the shader program, call `applyModelViewMatrix()`.
- The 2nd object is displayed by `g_polygon_mesh_2.display()`.
- The transformation matrices are restored from the stack using `popModelViewMatrix()` and `popProjectionMatrix()`.
- `popShaderProgram()` disables the current shader and restores the previous shader from the stack.
- Finally, we check the error status of both OpenGL and the (FBO).

```
//--------------------
void displayCallback()
//--------------------
{
    // Clear the buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Add the current shader to the shader stack
    pushShaderProgram();

    // Enable the shader
    g_display_shader.enable();
    GLint shader_id(g_display_shader.getProgramHandle());

    // Check the status of OpenGL and of the current FBO
    checkFBOErrorStatus(__FILE__, __FUNCTION__, __LINE__);
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // A handle for shader resources
    GLuint handle(0);

    // Upload the projection matrix
    handle = glGetUniformLocation(shader_id, "g_projection_matrix");
    glUniformMatrix4fv(handle, 1, GL_FALSE, g_current_projection_matrix.get());
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // Upload the modelview matrix
    handle = glGetUniformLocation(shader_id, "g_modelview_matrix");
    glUniformMatrix4fv(handle, 1, GL_FALSE, g_current_modelview_matrix.get());
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // Store the current transformation matrices
    pushModelViewMatrix();
    pushProjectionMatrix();

    // Translate the 1st object
    g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(
            VEC3(8.0 * cm, 0.0, 0.0));

    // Rotate the 1st object
    g_current_modelview_matrix *= g_object_1_rotation_matrix;

    // Apply the change to the shader program
    applyModelViewMatrix();

    // Display the 1st object
    g_polygon_mesh_1.display();

```

```
49      // Restore the current transformation matrix
50      popModelViewMatrix();
51      popProjectionMatrix();
52
53      // Store the current transformation matrices
54      pushModelViewMatrix();
55      pushProjectionMatrix();
56
57      // Translate the 2nd object
58      g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(
59              VEC3(-8.0 * cm, 0.0, 0.0));
60
61      // Rotate the 2nd object
62      g_current_modelview_matrix *= g_object_2_rotation_matrix;
63
64      // Apply the change to the shader program
65      applyModelViewMatrix();
66
67      // Display the 2nd object
68      g_polygon_mesh_2.display();
69
70      // Restore the current transformation matrix
71      popModelViewMatrix();
72      popProjectionMatrix();
73
74      // Disable the shader and restore the previous shader from the stack
75      popShaderProgram();
76
77      // Check the status of OpenGL and of the current FBO
78      checkFBOErrorStatus(__FILE__, __FUNCTION__, __LINE__);
79      checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
80  }
```

Listing 9: display the 3D scene.

11 Idle Callback

In the idle callback the rotation matrices are updated to create an animation (see Listing 10). `g_object_1_rotation_matrix.rotate(1.0, VEC3(1.0, 0.0, 0.0))` multiplies the transformation matrix (`g_object_1_rotation_matrix`) by a rotation matrix defined by angle in degrees (`1.0` in this example) and a rotation axis (`VEC3(1.0, 0.0, 0.0)`).

```
1  //--------------------
2  void idleCallback()
3  //--------------------
4  {
5      // Rotate the objects
6      g_object_1_rotation_matrix.rotate(1.0, VEC3(1.0, 0.0, 0.0));
7      g_object_2_rotation_matrix.rotate(2.0, VEC3(0.0, 1.0, 0.0));
8  }
```

Listing 10: Idle callback.

12 Quit Callback

Listing 11 shows how to clean up GLFW. If the window exists, then it is destroyed. Finally, the GLFW application is terminated.

```
1  //--------------------
2  void quitCallback()
3  //--------------------
4  {
```

```
    // The window exists
    if (g_p_window_id)
    {
        // Close the window
        glfwDestroyWindow(g_p_window_id);
        g_p_window_id = 0;

        // Cleanup GLFW
        glfwTerminate();
    }
}
```

Listing 11: Quit callback.

13 Frame Buffer Size Callback

This callback is called when the size of the frame buffer changes, that is to say when the size of the window changes. In this function three main things are performed (see Listing 12).

- Set the OpenGL's viewport with `glViewport`.
- Set the projection matrix. In this example; we use `loadPerspectiveProjectionMatrix` to set `g_current_projection_matrix`. It is similar to `gluPerspective`, which is not available in modern OpenGL.
- Set the modelling-viewing transformation. In this example; we use `loadLookAtModelViewMatrix` to set `g_current_modelview_matrix`. It is similar to `gluLookAt`, which is not available in modern OpenGL.

```
//------------------------------------------------------------------------
void framebufferSizeCallback(GLFWwindow* apWindow, int width, int height)
//------------------------------------------------------------------------
{
    // Avoid a division by zero
    if (height == 0)
    {
        // Prevent divide by 0
        height = 1;
    }

    int x(0), y(0), w(width), h(height);

    // Store the width and height of the window
    g_window_width = width;
    g_window_height = height;

    // Compute the aspect ratio of the size of the window
    double screen_aspect_ratio(double(g_window_width) / double(g_window_height));

    // Update the viewport
    glViewport(x, y, w, h);

    // Set up the projection matrix (g_current_projection_matrix)
    loadPerspectiveProjectionMatrix(45.0, screen_aspect_ratio, 0.1 * cm, 5000.0 * cm);

    // Set up the modelling-viewing matrix (g_current_modelview_matrix)
    loadLookAtModelViewMatrix(50.0 * cm, 50.0 * cm, g_zoom,
            0.0, 0.0, 0.0,
            0.0, 1.0, 0.0);
}
```

Listing 12: Change of frame buffer size callback.

14 Keyboard Callback

Listing 13 shows a typical GLFW keyboard callback. Here the window is closed when the user presses `Q` or `Esc`.

```
//--------------------------------------------------------------------------
void keyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
//--------------------------------------------------------------------------
{
    if (action == GLFW_PRESS)
    {
        switch(key)
        {
        // Close the program
        case GLFW_KEY_Q:
        case GLFW_KEY_ESCAPE:
                glfwSetWindowShouldClose(g_p_window_id, GL_TRUE);
                break;

        default:
            break;
        }
    }
}
```

Listing 13: Keyboard callback.

15 Scroll Button Callback

The zoom is updated with the mouse wheel (see Listing 14). To update the modelling-viewing matrix, the Frame Buffer Size Callback is called.

```
//--------------------------------------------------------------------------
void scrollCallback(GLFWwindow* apWindow, double xoffset, double yoffset)
//--------------------------------------------------------------------------
{
    // Scrolling along the Y-axis
    if (fabs(yoffset) > EPSILON)
    {
        // Change the zoom
        g_zoom += 5 * yoffset * cm;

        // Update the projection matrix
        framebufferSizeCallback(apWindow, g_window_width, g_window_height);
    }
}
```

Listing 14: Scroll button callback.

16 Error Callback

In the error callback an exception is thrown with the details of the error (see Listing 15).

```
//--------------------------------------------------------
void errorCallback(int error, const char* description)
//--------------------------------------------------------
{
    // Throw an error
    throw Exception(__FILE__, __FUNCTION__, __LINE__, description);
}
```

Listing 15: Error callback.

17 Next Tutorial...

In the next tutorial:

- Load the shader from a file compressed using the Zlib [5]
- We will see how to create an efficient mouse control to turn the 3D scene.

A Program Source Code

```
/*

Copyright (c) 2014, Dr Franck P. Vidal (franck.p.vidal@fpvidal.net),
http://www.fpvidal.net/
All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.

3. Neither the name of the Bangor University nor the names of its contributors
may be used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

*/


/**
*****************************************************************************
*
*   @file       tutorial_01_glfw.cxx
*
*   @brief      Creating a Window and an OpenGL Core Profile 3.2 Context Using GLFW.
*
*   @version    1.0
*
*   @date       06/05/2014
*
*   @author     Dr Franck P. Vidal
*
*   @section    License
*               BSD 3-Clause License.
*
*               For details on use and redistribution please refer
*               to http://opensource.org/licenses/BSD-3-Clause
*
*   @section    Copyright
```

[5] http://www.zlib.net/

```
*               (c) by Dr Franck P. Vidal (franck.p.vidal@fpvidal.net),
*               http://www.fpvidal.net/, Dec 2014, 2014, version 1.0,
*               BSD 3-Clause License
*
*******************************************************************************
*/


//******************************************************************************
//  Include
//******************************************************************************
// Fix undefined references on Windows
#if defined(_WIN32) || defined(_WIN64)
#include <GL/glew.h>
#endif

// Ensure we are using opengl's core profile only
#ifdef __APPLE__
#define GL3_PROTOTYPES 1
#else
#define GL_GLEXT_PROTOTYPES 1
#endif

#define GLFW_INCLUDE_GLCOREARB 1 // Tell GLFW to include the OpenGL core profile header

#include <GLFW/glfw3.h> // Create an OpenGL context and attach a window to it

#include <iostream>   // Print error messages in the console
#include <exception>  // Catch C++ exception
#include <cstdlib>    // Define return status (EXIT_SUCCESS and EXIT_FAILURE)

// Define new types, e.g.~RATIONAL_NUMBER, VEC2, VEC3 and MATRIX4 to name a few
#include "gVirtualXRay/Types.h"

// Define units such as metre, kilometre, electronvolt, gram, kilogram, etc.
#include "gVirtualXRay/Units.h"

// Some utility functions about OpenGL, e.g. matrix stacks,
// how to set the projection matrix, etc.
#include "gVirtualXRay/OpenGLUtilities.h"

#include "gVirtualXRay/PolygonMesh.h"  // Handle 3D triangle meshes
#include "gVirtualXRay/Shader.h"       // Handle GLSL programs

#include "buildCube.h" // Create the triangle mesh of a cube


//******************************************************************************
//  Name space
//******************************************************************************
using namespace gVirtualXRay;
using namespace std;


//******************************************************************************
//  Global variables
//******************************************************************************
// Keep track of the window width
GLsizei g_window_width(640);

// Keep track of the window height
GLsizei g_window_height(480);

// GLFW window ID
GLFWwindow* g_p_window_id(0);

// Shader program used to display the 3D scene
Shader g_display_shader;
```

```
// 3D objects as VAOs and VBOs
PolygonMesh g_polygon_mesh_1;
PolygonMesh g_polygon_mesh_2;

// Transformation matrices
MATRIX4 g_object_1_rotation_matrix;
MATRIX4 g_object_2_rotation_matrix;

// Geometric data
vector<double> g_p_vertex_set_1;
vector<unsigned char> g_p_index_set_1;
vector<float> g_p_vertex_set_2;

// Control the zoom
RATIONAL_NUMBER g_zoom(50.0 * cm);

// Vertex shader
const GLchar* g_vertex_shader = "\
\n#version 150\n \
\n \
in vec3 in_Vertex;\n \
\n \
uniform mat4 g_projection_matrix;\n \
uniform mat4 g_modelview_matrix;\n \
\n \
void main(void)\n \
{\n \
    gl_Position = g_projection_matrix * g_modelview_matrix * vec4(in_Vertex, 1.0);\n \
}\n \
";

// Fragment shader
const GLchar* g_fragment_shader = "\
\n#version 150\n \
precision highp float;\n \
\n \
out vec4 fragColor;\n \
void main(void)\n \
{\n \
    fragColor = vec4(1.0, 1.0, 1.0, 1.0);\n \
}\n \
";


//*****************************************************************************
//  Function declaration
//*****************************************************************************
void initGLFW();
void initGLEW();
void initGL();
void load3DObjects();
void displayCallback();
void idleCallback();
void quitCallback();
void framebufferSizeCallback(GLFWwindow* apWindow, int aWidth, int aHeight);
void keyCallback(GLFWwindow* apWindow, int aKey, int aScanCode, int anAction, int
    aModifierKey);
void scrollCallback(GLFWwindow* apWindow, double xoffset, double yoffset);
void errorCallback(int error, const char* description);


//------------------------------
int main(int argc, char** argv)
//------------------------------
{
    // Return code
    int return_code(EXIT_SUCCESS);
```

```
    // Register the exit callback
    atexit(quitCallback);

    try
    {
        // Initialise GLFW
        initGLFW();

        // Initialise GLEW
        initGLEW();

        // Initialise OpenGL
        initGL();

        // Give a text ID to the vertex and fragment shaders, it can be useful when
    debuging shaders
        g_display_shader.setLabels("g_vertex_shader", "g_fragment_shader");

        // Load the source code of the shaders onto the GPU
        g_display_shader.loadSource(g_vertex_shader, g_fragment_shader);

        // Initialise the geometry of the 3D objects
        load3DObjects();

        // Set the projection matrix
        framebufferSizeCallback(g_p_window_id, g_window_width, g_window_height);

        // Launch the event loop
        while (!glfwWindowShouldClose(g_p_window_id))
        {
            // Render here
            displayCallback();

            // Swap front and back buffers
            glfwSwapBuffers(g_p_window_id);

            // Poll for and process events
            glfwPollEvents();

            // Idle callback
            idleCallback();
        }
    }
    // Catch exception if any
    catch (const exception& error)
    {
        cerr << error.what() << endl;
        return_code = EXIT_FAILURE;
    }

    // Close the window and shut GLFW if needed
    quitCallback();

    // Return an exit code
    return (return_code);
}


//--------------
void initGLFW()
//--------------
{
    // Set an error callback
    glfwSetErrorCallback(errorCallback);

    // Initialize GLFW
    if (!glfwInit())
```

```
    {
        throw Exception(__FILE__, __FUNCTION__, __LINE__,
                "ERROR: cannot initialise GLFW.");
    }

    // Enable OpenGL 3.2 if possible
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // Enable anti-aliasing
    glfwWindowHint(GLFW_SAMPLES, 4);

    // Create a windowed mode window and its OpenGL context
    g_p_window_id = glfwCreateWindow(g_window_width, g_window_height,
            "gVirtualXRay -- Tutorial 01", NULL, NULL);

    // The window has not been created
    if (!g_p_window_id)
    {
        throw Exception(__FILE__, __FUNCTION__, __LINE__,
                "ERROR: cannot create a GLFW windowed mode window and its OpenGL context.")
    ;
    }

    // Make the window's context current
    glfwMakeContextCurrent(g_p_window_id);

    // Register GLFW callbacks
    glfwSetFramebufferSizeCallback(g_p_window_id, framebufferSizeCallback);
    glfwSetKeyCallback(g_p_window_id, keyCallback);
    glfwSetScrollCallback(g_p_window_id, scrollCallback);
}


//--------------
void initGLEW()
//--------------
{
#ifdef _WIN32
  GLenum err = glewInit();
  if (GLEW_OK != err)
  {
    std::stringstream error_message;
    error_message << "ERROR: cannot initialise GLEW:\t" << glewGetErrorString(err);

        throw Exception(__FILE__, __FUNCTION__, __LINE__, error_message.str());
  }
#endif
}


//------------
void initGL()
//------------
{
    // Enable the Z-buffer
    glEnable(GL_DEPTH_TEST);

    // Set the background colour
    glClearColor(0.5, 0.5, 0.5, 1.0);

    // Check if any OpenGL error has occurred.
    // If any has, an exception is thrown
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
}
```

```
//--------------------
void load3DObjects()
//--------------------
{
    // Centre of the cubes
    VEC3 cube_centre(0, 0, 0);

    // Size of the cubes
    RATIONAL_NUMBER length_1( 5.0 * cm);
    RATIONAL_NUMBER length_2(10.0 * cm);

    // Create the cube using vertex data and index data
    buildCube(length_1, cube_centre, g_p_vertex_set_1, g_p_index_set_1);

    // Create the cube using vertex data only
    buildCube(length_2, cube_centre, g_p_vertex_set_2);

    // Set geometry (using VAOs and VBOs)
    g_polygon_mesh_1.setExternalData(GL_TRIANGLES,
            &g_p_vertex_set_1,
            &g_p_index_set_1,
            true,
            GL_STATIC_DRAW);

    g_polygon_mesh_2.setExternalData(GL_TRIANGLES,
            &g_p_vertex_set_2,
            true,
            GL_STATIC_DRAW);
}


//----------------------
void displayCallback()
//----------------------
{
    // Clear the buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Add the current shader to the shader stack
    pushShaderProgram();

    // Enable the shader
    g_display_shader.enable();
    GLint shader_id(g_display_shader.getProgramHandle());

    // Check the status of OpenGL and of the current FBO
    checkFBOErrorStatus(__FILE__, __FUNCTION__, __LINE__);
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // A handle for shader resources
    GLuint handle(0);

    // Upload the projection matrix
    handle = glGetUniformLocation(shader_id, "g_projection_matrix");
    glUniformMatrix4fv(handle, 1, GL_FALSE, g_current_projection_matrix.get());
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // Upload the modelview matrix
    handle = glGetUniformLocation(shader_id, "g_modelview_matrix");
    glUniformMatrix4fv(handle, 1, GL_FALSE, g_current_modelview_matrix.get());
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);

    // Store the current transformation matrices
    pushModelViewMatrix();
    pushProjectionMatrix();

    // Translate the 1st object
```

```
    g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(
            VEC3(8.0 * cm, 0.0, 0.0));

    // Rotate the 1st object
    g_current_modelview_matrix *= g_object_1_rotation_matrix;

    // Apply the change to the shader program
    applyModelViewMatrix();

    // Display the 1st object
    g_polygon_mesh_1.display();

    // Restore the current transformation matrix
    popModelViewMatrix();
    popProjectionMatrix();

    // Store the current transformation matrices
    pushModelViewMatrix();
    pushProjectionMatrix();

    // Translate the 2nd object
    g_current_modelview_matrix *= MATRIX4::buildTranslationMatrix(
            VEC3(-8.0 * cm, 0.0, 0.0));

    // Rotate the 2nd object
    g_current_modelview_matrix *= g_object_2_rotation_matrix;

    // Apply the change to the shader program
    applyModelViewMatrix();

    // Display the 2nd object
    g_polygon_mesh_2.display();

    // Restore the current transformation matrix
    popModelViewMatrix();
    popProjectionMatrix();

    // Disable the shader and restore the previous shader from the stack
    popShaderProgram();

    // Check the status of OpenGL and of the current FBO
    checkFBOErrorStatus(__FILE__, __FUNCTION__, __LINE__);
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
}


//------------------
void idleCallback()
//------------------
{
    // Rotate the objects
    g_object_1_rotation_matrix.rotate(1.0, VEC3(1.0, 0.0, 0.0));
    g_object_2_rotation_matrix.rotate(2.0, VEC3(0.0, 1.0, 0.0));
}


//------------------
void quitCallback()
//------------------
{
    // The window exists
    if (g_p_window_id)
    {
        // Close the window
        glfwDestroyWindow(g_p_window_id);
        g_p_window_id = 0;

        // Cleanup GLFW
```

```
        glfwTerminate();
    }
}


//--------------------------------------------------------------------------
void framebufferSizeCallback(GLFWwindow* apWindow, int width, int height)
//--------------------------------------------------------------------------
{
    // Avoid a division by zero
    if (height == 0)
    {
        // Prevent divide by 0
        height = 1;
    }

    int x(0), y(0), w(width), h(height);

    // Store the width and height of the window
    g_window_width = width;
    g_window_height = height;

    // Compute the aspect ratio of the size of the window
    double screen_aspect_ratio(double(g_window_width) / double(g_window_height));

    // Update the viewport
    glViewport(x, y, w, h);

    // Set up the projection matrix (g_current_projection_matrix)
    loadPerspectiveProjectionMatrix(45.0, screen_aspect_ratio, 0.1 * cm, 5000.0 * cm);

    // Set up the modelling-viewing matrix (g_current_modelview_matrix)
    loadLookAtModelViewMatrix(50.0 * cm, 50.0 * cm, g_zoom,
            0.0, 0.0, 0.0,
            0.0, 1.0, 0.0);
}


//--------------------------------------------------------------------------
void keyCallback(GLFWwindow* window, int key, int scancode, int action, int mods)
//--------------------------------------------------------------------------
{
    if (action == GLFW_PRESS)
    {
        switch(key)
        {
        // Close the program
        case GLFW_KEY_Q:
        case GLFW_KEY_ESCAPE:
                glfwSetWindowShouldClose(g_p_window_id, GL_TRUE);
                break;

        default:
            break;
        }
    }
}


//--------------------------------------------------------------------------
void scrollCallback(GLFWwindow* apWindow, double xoffset, double yoffset)
//--------------------------------------------------------------------------
{
    // Scrolling along the Y-axis
    if (fabs(yoffset) > EPSILON)
    {
        // Change the zoom
        g_zoom += 5 * yoffset * cm;
```

```
        // Update the projection matrix
        framebufferSizeCallback(apWindow, g_window_width, g_window_height);
    }
}


//-----------------------------------------------------------
void errorCallback(int error, const char* description)
//-----------------------------------------------------------
{
    // Throw an error
    throw Exception(__FILE__, __FUNCTION__, __LINE__, description);
}
```

Listing 16: All the source code of this tutorial.