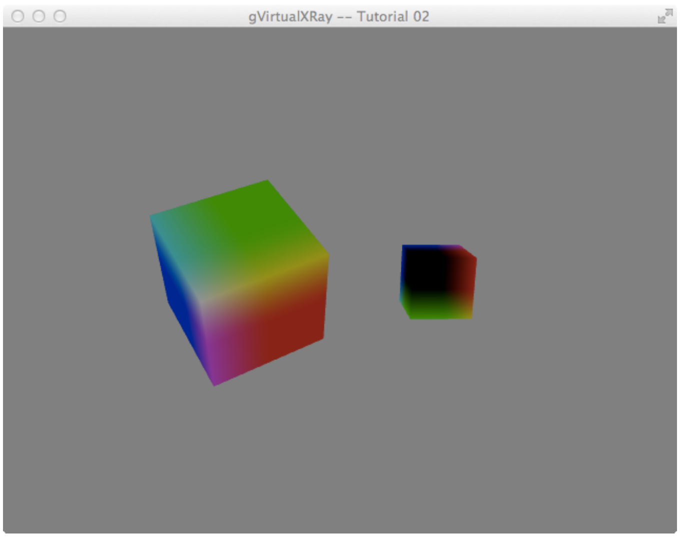
gVirtualXRay – Tutorial 02 – GLUT: Loading the shader program from a file compressed with the Z library. Adding an efficient mouse control to turn the 3D scene.

Dr Franck P. Vidal

27th 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 `tutorials/tutorial_02_glut/tutorial_02_glut.cxx`. It can be downloaded here: https://sourceforge.net/p/gvirtualxray/code/HEAD/tree/trunk/tutorials/tutorial_02_glut/tutorial_02_glut.cxx. It improves the 1st tutorial in two ways:

1. Loading the shader program from a file compressed with the Z library.
2. Adding an efficient mouse control to turn the 3D scene.

Two cubes are displayed (see Figure 1). First the fragment program is saved into `tutorial_02_gl2.frag`.

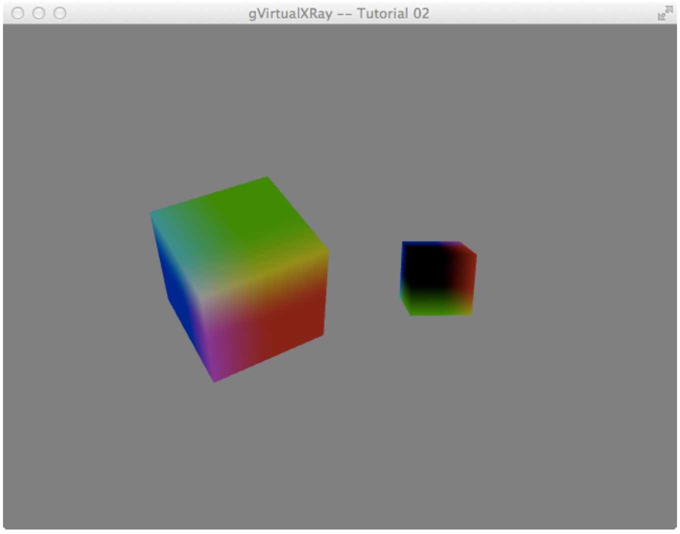


Figure 1: Screen capture of the second tutorial.

The vertex program is saved into `tutorial_02_gl2.vert`. Our implementation supports OpenGL 2.x only when GLUT is used. OpenGL 3.x and OpenGL 4.x or above are supported when GLFW is used. The files are individually compressed using *gzip* [1]. Finally each compressed file is converted into a C header file. It can be achieved CMake [2] or an external tool such as bin2c [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 functions have to be declared.
- Section 6 shows how to initialise the shaders from compressed data stored in C header files.
- Section 7 shows the callback for mouse button events. The mouse is used to rotate objects in an intuitive way.
- Section 8 shows how callback for cursor position events works.
- How to convert screen coordinates to arcball vectors is explained in Section 9
- The conversion from radian to degree is shown in Section 10.
- Section 11 shows how the corresponding rotation matrix is computed.
- Section 12 gives a preview of what the next tutorial will be about.
- Appendix A shows the source code of this tutorial.

[1] http://www.gzip.org/
[2] http://www.cmake.org/
[3] http://sourceforge.net/projects/bin2c/

2 Header inclusion

Listing 1 only shows the new i) the macros that have to be defines to include OpenGL core profile headers and ii) the header files that need to be included to decompress the data using the **zlib** library [4]:

- `gVirtualXRay/Utilities.h` is the new header file that is used to handle decompression using the Z library.
- `tutorial_02_gl2.frag.h` is the compressed fragment shader for OpenGL 2.x.
- `tutorial_02_gl2.vert.h` is the compressed vertex shader for OpenGL 2.x.
- For other header files, see the GLUT version of Tutorial 1.

```
//****************************************************************************
//  Include
//****************************************************************************
#include <GL/glew.h>  // Handle shaders

// Create an OpenGL context and attach a window to it
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <iostream>   // Print error messages in the console
#include <sstream>    // Generate error messages
#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 "gVirtualXRay/Utilities.h" // Handle decompression using the Z library

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

#include "tutorial_02_gl2.frag.h" // Fragment shader for OpenGL 2.x
#include "tutorial_02_gl2.vert.h" // Vertex shader for OpenGL 2.x
```

Listing 1: Header inclusion.

3 Name Spaces

See Tutorial 1.

4 Global Variables

Listing 2 shows the global variables that are used:

- `g_rotation_speed` controls the speed of rotation of the cubes.

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

- `g_tutorial_02_gl2_frag` is declared in `tutorial_02_gl2.frag.h`. It is the compressed fragment shader for OpenGL 2.x.
- `g_tutorial_02_gl2_vert` is declared in `tutorial_02_gl2.vert.h`. It is the compressed vertex shader for OpenGL 2.x.
- `bool g_use_arc_ball` is used to know if arc ball rotation is currently in use.
- `int g_button` stores which button generated an event last.
- `int g_button_state` stores the new state of this button.
- `GLint g_last_x_position` is the previous last known X position of the cursor.
- `GLint g_last_y_position` is the previous last known Y position of the cursor.
- `GLint g_current_x_position` is the current X position of the cursor.
- `GLint g_current_y_position` is the current Y position of the cursor.
- For other global variables, see the GLUT version of Tutorial 1.

```
//*****************************************************************************
//  Constant variables
//*****************************************************************************
const GLfloat g_rotation_speed(2.0);


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

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

// GLUT window ID
int g_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;
MATRIX4 g_scene_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);

// Use the arc ball rotation
bool g_use_arc_ball(false);

// States of the mouse
int g_button(-1); // Button
int g_button_state(-1); // Button state
GLint g_last_x_position(0); // Previous x position of the cursor
```

```
GLint g_last_y_position(0); // Previous y position of the cursor
GLint g_current_x_position(0); // Current x position of the cursor
GLint g_current_y_position(0); // Current x position of the cursor
```

Listing 2: Global variables.

5 Function Declarations

One function has been added to load shaders from compressed files and five functions have been add to control the rotation using the mouse:

- `initShader` is used i) to decompress the shaders, and ii) to compile the shaders (see Section 6).
- `mouseButtonCallback` is called when a mouse button event occurs (see Section 7).
- `cursorPosCallback` is used when the mouse moves (see Section 8).
- `getArcballVector` computes the arc ball vector given screen coordinates (see Section 9). Details about arc ball rotation can be found at http://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Arcball and http://nehe.gamedev.net/tutorial/arcball_rotation/19003/.
- `radian2degree` converts an angle value in radians into degrees (see Section 10).
- `computeRotation` generates the corresponding matrix rotation (see Section 11).

Listing 3 shows the declaration of all the functions.

```
//****************************************************************************
//  Function declaration
//****************************************************************************
void initGLUT(int argc, char** argv);
void initGLEW();
void initGL();
void initShader();
void load3DObjects();
void displayCallback();
void idleCallback();
void quitCallback();
void framebufferSizeCallback(int aWidth, int aHeight);
void keyCallback(unsigned char aKey, int aXPosition, int aYPosition);
void mouseButtonCallback(int aButton, int aState, int aXPosition, int aYPosition);
void cursorPosCallback(int aXPosition, int aYPosition);
VEC3 getArcballVector(int x, int y);
float radian2degree(float anAngle);
void computeRotation(MATRIX4& aRotationMatrix);
```

Listing 3: Function declarations.

6 Shader Initialisation

- `g_display_shader` is the instance of the Shader class that will store our GLSL programs.
- `g_tutorial_02_gl3_vert` is declared in `tutorial_02_gl3.vert.h`. It is the compressed vertex shader for OpenGL 3.x or OpenGL 4.x. To generate the file, the text file containing our vertex shader was compressed using Gzip. A C header was then generated from the compressed file. It can be done using bin2c or CMake. There is no need to distribute text files with the executable file that reads the code at runtime. The code of the shader is embedded in the executable file.
- The same is done for the fragment shader. `g_tutorial_02_gl2_frag` is declared in `tutorial_02_gl2.frag.h`. It is the compressed fragment shader for OpenGL 3.x or OpenGL 4.x.

- `g_tutorial_02_gl3_vert` is decompressed into `p_vertex_shader`, `g_tutorial_02_gl3_frag` into `p_fragment_shader`, using the `inflate` function that will invoke Z library.
- Error codes are stored in `z_lib_return_code_vertex` and `z_lib_return_code_fragment`. They are used to make sure the decompression has been successful.
- The source of the shaders are then saved into two separate `std::string` instances (`vertex_shader` and `fragment_shader`).
- If a decompression error occurred, an exception is thrown.
- To help debugging, it is possible to give the vertex and fragment programs labels: `g_display_shader.setLabels("display.vert", "display.frag")`.
- Finally, the Shader instance `g_display_shader` loads the source code.

Listing 4 shows how to perform all these steps.

```
//-----------------
void initShader()
//-----------------
{
    // Initialise the shaders
    char* p_vertex_shader(0);
    char* p_fragment_shader(0);

    int z_lib_return_code_vertex(0);
    int z_lib_return_code_fragment(0);

    std::string vertex_shader;
    std::string fragment_shader;

    // L-buffer
    z_lib_return_code_vertex   = inflate(g_tutorial_02_gl2_vert, sizeof(
g_tutorial_02_gl2_vert),   &p_vertex_shader);
    z_lib_return_code_fragment = inflate(g_tutorial_02_gl2_frag, sizeof(
g_tutorial_02_gl2_frag), &p_fragment_shader);

    vertex_shader   = p_vertex_shader;
    fragment_shader = p_fragment_shader;
    delete [] p_vertex_shader;     p_vertex_shader = 0;
    delete [] p_fragment_shader; p_fragment_shader = 0;

    if (z_lib_return_code_vertex <= 0 || z_lib_return_code_fragment <= 0 || !vertex_shader.
size() || !fragment_shader.size())
    {
        throw Exception(__FILE__, __FUNCTION__, __LINE__, "Cannot decode the shader using
ZLib.");
    }
    g_display_shader.setLabels("display.vert", "display.frag");
    g_display_shader.loadSource(vertex_shader, fragment_shader);
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
}
```

Listing 4: Shader initialisation.

7 Callback for Mouse Button Events

This callback stores the mouse states in global variables, i.e. which button generated an event, the type of events and the cursor's position in screen coordinates) (see Listing 5). If the mouse button is pressed, then arcball rotation is in used. If the mouse button is released, then arcball rotation is stopped.

In `void initGLUT()`, the callback is registered as follows: `glutMouseFunc(mouseButtonCallback)`.

```
//-----------------------------------------------------------------------------
void mouseButtonCallback(int aButton, int aState, int aXPosition, int aYPosition)
//-----------------------------------------------------------------------------
{
    g_button = aButton;
    g_button_state = aState;

    // This is a click event
    if (g_button == GLUT_LEFT_BUTTON || g_button == GLUT_RIGHT_BUTTON || g_button == GLUT_MIDDLE_BUTTON)
    {
    // Use the arc ball
    if (g_button_state == GLUT_DOWN)
    {
      g_use_arc_ball = true;
    }
    // Stop using the arc ball
    else
    {
      g_use_arc_ball = false;
    }

    double xpos(aXPosition);
    double ypos(aYPosition);

    g_last_x_position = xpos;
    g_last_y_position = ypos;

    g_current_x_position = xpos;
    g_current_y_position = ypos;
    }

  // The button is down
  if (aState == GLUT_DOWN)
  {
    // This is a wheel event
    if (aButton == 3)
    {
      // Change the zoom
      g_zoom += 5 * cm;

      // Update the projection matrix
      framebufferSizeCallback(g_window_width, g_window_height);
    }
    // This is a wheel event
    else if (aButton == 4)
    {
      // Change the zoom
      g_zoom -= 5 * cm;

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

  // Redisplay the scene
  glutPostRedisplay();
}
```

Listing 5: Callback for mouse button events.

8 Callback for Cursor Position Events

Listing 6 shows i) how to store the cursor's position in global variables,and ii) how the matrix rotation is updated accordingly.

In `void initGLUT()`, the callback is registered as follows: `glutMotionFunc(cursorPosCallback)`.

```
//------------------------------------------------------------
void cursorPosCallback(int aXPosition, int aYPosition)
//------------------------------------------------------------
{
    g_current_x_position = aXPosition;
    g_current_y_position = aYPosition;

    computeRotation(g_scene_rotation_matrix);

  // Redisplay the scene
  glutPostRedisplay();
}
```

Listing 6: Callback for cursor position events.

9 Conversion from Screen Coordinates to Arcball Vector

Listing 7 shows how to compute the arc ball vector given screen coordinates. Arcball rotation allows the user to rotate objects in a natural and intuitive way. Details about arc ball rotation can be found at http://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Arcball and http://nehe.gamedev.net/tutorial/arcball_rotation/19003/.

```
//----------------------------------------
VEC3 getArcballVector(int x, int y)
//----------------------------------------
{
    VEC3 P(2.0 * float(x) / float(g_window_width) - 1.0,
           2.0 * float(y) / float(g_window_height) - 1.0,
           0);

    P.setY(-P.getY());

    float OP_squared = P.getX() * P.getX() + P.getY() * P.getY();
    if (OP_squared <= 1.0)
    {
        P.setZ(sqrt(1.0 - OP_squared));   // Pythagore
    }
    else
    {
        P.normalise();   // Nearest point
    }

    return (P);
}
```

Listing 7: Conversion from screen coordinates to arc ball vector.

10 Radian to Degree Conversion

π radians corresponds to 180 degrees. Listing 8 shows how to perform this simple conversion.

```
//----------------------------------------
float radian2degree(float anAngle)
//----------------------------------------
{
    return (180.0 * anAngle / gVirtualXRay::PI);
}
```

Listing 8: Convert an angle value from radian to degree.

11 Compute Rotation Matrix

If the arcball rotation is currently in used, if the cursor's position has changed, then the rotation matrix is updated using the arcball technique (see Listing 9).

```
//----------------------------------------------------------------
void computeRotation(MATRIX4& aRotationMatrix)
//----------------------------------------------------------------
{
    if (g_use_arc_ball)
    {
        if (g_current_x_position != g_last_x_position || g_current_y_position !=
    g_last_y_position)
        {
            VEC3 va(getArcballVector(g_last_x_position,    g_last_y_position));
            VEC3 vb(getArcballVector(g_current_x_position, g_current_y_position));

            float angle(g_rotation_speed * radian2degree(acos(min(1.0, va.dotProduct(vb))))
    );

            VEC3 axis_in_camera_coord(va ^ vb);
            //axis_in_camera_coord.normalize();

            MATRIX4 camera2object(aRotationMatrix.getInverse());
            VEC3 axis_in_object_coord = camera2object * axis_in_camera_coord;
            //axis_in_object_coord.normalize();

            MATRIX4 rotation_matrix;
            rotation_matrix.rotate(angle, axis_in_object_coord);
            aRotationMatrix = aRotationMatrix * rotation_matrix;

            g_last_x_position = g_current_x_position;
            g_last_y_position = g_current_y_position;
        }
    }
}
```

Listing 9: Compute the arcball rotation.

12 Next Tutorial...

In the next tutorial:

- Display two cubes with a better shader program and with transparency (one of the cubes is within the other one). Do you see where we are going?
- Display the cubes in wireframe.
- Compute the X-ray attenuation of the cubes.

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_02_glut.cxx
*
*   @brief      Improve the first tutorial by:
*               - Loading the shader program from a file compressed with the Z library.
*               - Adding an efficient mouse control to turn the 3D scene.
*
*   @version    1.0
*
*   @date       17/08/2014
*
*   @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
*               (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
//******************************************************************************
#include <GL/glew.h>  // Handle shaders

// Create an OpenGL context and attach a window to it
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <iostream>   // Print error messages in the console
#include <sstream>    // Generate error messages
#include <exception>  // Catch C++ exception
```

```cpp
#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 "gVirtualXRay/Utilities.h" // Handle decompression using the Z library

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

#include "tutorial_02_gl2.frag.h" // Fragment shader for OpenGL 2.x
#include "tutorial_02_gl2.vert.h" // Vertex shader for OpenGL 2.x


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


//******************************************************************************
//  Constant variables
//******************************************************************************
const GLfloat g_rotation_speed(2.0);


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

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

// GLUT window ID
int g_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;
MATRIX4 g_scene_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);
```

```
// Use the arc ball rotation
bool g_use_arc_ball(false);

// States of the mouse
int g_button(-1); // Button
int g_button_state(-1); // Button state
GLint g_last_x_position(0); // Previous x position of the cursor
GLint g_last_y_position(0); // Previous y position of the cursor
GLint g_current_x_position(0); // Current x position of the cursor
GLint g_current_y_position(0); // Current x position of the cursor


//******************************************************************************
//  Function declaration
//******************************************************************************
void initGLUT(int argc, char** argv);
void initGLEW();
void initGL();
void initShader();
void load3DObjects();
void displayCallback();
void idleCallback();
void quitCallback();
void framebufferSizeCallback(int aWidth, int aHeight);
void keyCallback(unsigned char aKey, int aXPosition, int aYPosition);
void mouseButtonCallback(int aButton, int aState, int aXPosition, int aYPosition);
void cursorPosCallback(int aXPosition, int aYPosition);
VEC3 getArcballVector(int x, int y);
float radian2degree(float anAngle);
void computeRotation(MATRIX4& aRotationMatrix);


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

    // Register the exit callback
    atexit(quitCallback);

    try
    {
        // Initialise GLUT
        initGLUT(argc, argv);

        // Initialise GLEW
        initGLEW();

        // Initialise OpenGL
        initGL();

        // Initialise the shader from files compressed using the Z library
        initShader();

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

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

        // Launch the event loop
        glutMainLoop();
    }
    // 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 initGLUT(int argc, char** argv)
//-------------------------------------
{
  // GLUT initialisation
  glutInit (&argc, argv);

  // Create a windowed mode window and its OpenGL context
  glutInitWindowSize(g_window_width, g_window_height);
  glutInitDisplayMode ( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    g_window_id = glutCreateWindow("gVirtualXRay -- Tutorial 02");

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

    // Register GLUT callbacks
    glutDisplayFunc(displayCallback);
    glutReshapeFunc(framebufferSizeCallback);
    glutKeyboardFunc(keyCallback);
    glutMouseFunc(mouseButtonCallback);
    glutMotionFunc(cursorPosCallback);
    glutIdleFunc(idleCallback);
}


//-----------------
void initGLEW()
//-----------------
{
  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());
  }
}


//-------------------
void initShader()
//-------------------
{
    // Initialise the shaders
    char* p_vertex_shader(0);
    char* p_fragment_shader(0);

    int z_lib_return_code_vertex(0);
    int z_lib_return_code_fragment(0);
```

```
    std::string vertex_shader;
    std::string fragment_shader;

    // L-buffer
    z_lib_return_code_vertex   = inflate(g_tutorial_02_gl2_vert, sizeof(
    g_tutorial_02_gl2_vert),   &p_vertex_shader);
    z_lib_return_code_fragment = inflate(g_tutorial_02_gl2_frag, sizeof(
    g_tutorial_02_gl2_frag), &p_fragment_shader);

    vertex_shader   = p_vertex_shader;
    fragment_shader = p_fragment_shader;
    delete [] p_vertex_shader;     p_vertex_shader = 0;
    delete [] p_fragment_shader; p_fragment_shader = 0;

    if (z_lib_return_code_vertex <= 0 || z_lib_return_code_fragment <= 0 || !vertex_shader.
    size() || !fragment_shader.size())
    {
        throw Exception(__FILE__, __FUNCTION__, __LINE__, "Cannot decode the shader using
    ZLib.");
    }
    g_display_shader.setLabels("display.vert", "display.frag");
    g_display_shader.loadSource(vertex_shader, fragment_shader);
    checkOpenGLErrorStatus(__FILE__, __FUNCTION__, __LINE__);
}


//-----------
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();

    // Rotate the 3D scene
    g_current_modelview_matrix *= g_scene_rotation_matrix;

    // Store the current transformation matrices
    pushModelViewMatrix();

    // 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();
    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__);

    // Swap front and back buffers
    glutSwapBuffers();
}


//-------------------------------------
void idleCallback()
//-------------------------------------
{
    // Rotate the objects
    g_object_1_rotation_matrix.rotate(.01, VEC3(1.0, 0.0, 0.0));
    g_object_2_rotation_matrix.rotate(.02, VEC3(0.0, 1.0, 0.0));

  // Redisplay the scene
  glutPostRedisplay();
}


//-------------------------------------
void quitCallback()
//-------------------------------------
{
    // The window exists
    if (g_window_id)
    {
        // Close the window
        glutDestroyWindow(g_window_id);
        g_window_id = 0;
    }
}


//------------------------------------------------------------------------------
void framebufferSizeCallback(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(0.0 * cm, 0.0 * cm, g_zoom,
            0.0, 0.0, 0.0,
            0.0, 1.0, 0.0);

  // Redisplay the scene
  glutPostRedisplay();
}


//----------------------------------------------------------------------
void keyCallback(unsigned char aKey, int aXPosition, int aYPosition)
//----------------------------------------------------------------------
{
  switch(aKey)
  {
  // Close the program
  case 27:
  case 'q':
        quitCallback();
    exit(EXIT_SUCCESS);
    break;

  default:
    break;
  }

  // Redisplay the scene
  glutPostRedisplay();
}


//-----------------------------------------------------------------------------------
void mouseButtonCallback(int aButton, int aState, int aXPosition, int aYPosition)
//-----------------------------------------------------------------------------------
{
    g_button = aButton;
    g_button_state = aState;

    // This is a click event
    if (g_button == GLUT_LEFT_BUTTON || g_button == GLUT_RIGHT_BUTTON || g_button ==
   GLUT_MIDDLE_BUTTON)
    {
    // Use the arc ball
    if (g_button_state == GLUT_DOWN)
    {
      g_use_arc_ball = true;
    }
    // Stop using the arc ball
    else
    {
      g_use_arc_ball = false;
    }
```

```
    double xpos(aXPosition);
    double ypos(aYPosition);

    g_last_x_position = xpos;
    g_last_y_position = ypos;

    g_current_x_position = xpos;
    g_current_y_position = ypos;
    }

  // The button is down
  if (aState == GLUT_DOWN)
  {
    // This is a wheel event
    if (aButton == 3)
    {
      // Change the zoom
      g_zoom += 5 * cm;

      // Update the projection matrix
      framebufferSizeCallback(g_window_width, g_window_height);
    }
    // This is a wheel event
    else if (aButton == 4)
    {
      // Change the zoom
      g_zoom -= 5 * cm;

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

  // Redisplay the scene
  glutPostRedisplay();
}


//----------------------------------------------------------------------------
void cursorPosCallback(int aXPosition, int aYPosition)
//----------------------------------------------------------------------------
{
    g_current_x_position = aXPosition;
    g_current_y_position = aYPosition;

    computeRotation(g_scene_rotation_matrix);

  // Redisplay the scene
  glutPostRedisplay();
}


//-----------------------------------------
VEC3 getArcballVector(int x, int y)
//-----------------------------------------
{
    VEC3 P(2.0 * float(x) / float(g_window_width) - 1.0,
           2.0 * float(y) / float(g_window_height) - 1.0,
           0);

    P.setY(-P.getY());

    float OP_squared = P.getX() * P.getX() + P.getY() * P.getY();
    if (OP_squared <= 1.0)
    {
        P.setZ(sqrt(1.0 - OP_squared));  // Pythagore
    }
    else
```

```
    {
        P.normalise();   // Nearest point
    }

    return (P);
}


//------------------------------------------------------------------
float radian2degree(float anAngle)
//------------------------------------------------------------------
{
    return (180.0 * anAngle / gVirtualXRay::PI);
}


//----------------------------------------------------------------------------------------
void computeRotation(MATRIX4& aRotationMatrix)
//----------------------------------------------------------------------------------------
{
    if (g_use_arc_ball)
    {
        if (g_current_x_position != g_last_x_position || g_current_y_position !=
    g_last_y_position)
        {
            VEC3 va(getArcballVector(g_last_x_position,    g_last_y_position));
            VEC3 vb(getArcballVector(g_current_x_position, g_current_y_position));

            float angle(g_rotation_speed * radian2degree(acos(min(1.0, va.dotProduct(vb))))
    );

            VEC3 axis_in_camera_coord(va ^ vb);
            //axis_in_camera_coord.normalize();

            MATRIX4 camera2object(aRotationMatrix.getInverse());
            VEC3 axis_in_object_coord = camera2object * axis_in_camera_coord;
            //axis_in_object_coord.normalize();

            MATRIX4 rotation_matrix;
            rotation_matrix.rotate(angle, axis_in_object_coord);
            aRotationMatrix = aRotationMatrix * rotation_matrix;

            g_last_x_position = g_current_x_position;
            g_last_y_position = g_current_y_position;
        }
    }
}
```

Listing 10: All the source code of this tutorial.