gVirtualXRay – Tutorial 01

Dr Franck P. Vidal

19th March 2014

Contents

List of Figures

Listings

1 Introduction

The complete source code of this tutorial is available in `example/src/tutorial_01.cxx`. It shows how to build a basic simulation pipeline and display the simulation setup using OpenGL. Basic knowledge of OpenGL and OpenGL Utility Toolkit (GLUT) is a pre-requisite. The gVirtualXRay library is built using modern programming paradigm. Errors are handles using C++ exceptions. In the OpenGL code, the fix rendering pipeline and direct rendering have both been avoided. Depreciated functions and features have also been avoided. In this tutorial, some of these functionalities are still in used to lighten the tutorial. For example, the matrix stacks are used and the fix rendering pipeline is used too.

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

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. How to set up all the main components of the simulation is describe in Section 4.

2 Header inclusion

```
// 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

// 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"

#include "gVirtualXRay/PolygonMesh.h"  // Handle 3D triangle meshes
#include "gVirtualXRay/XRayBeam.h"     // Manage X-Ray beams
#include "gVirtualXRay/XRayDetector.h" // Handle virtual X-Ray detectors
#include "gVirtualXRay/XRayRenderer.h" // Compute X-Ray images on the GPU
#include "gVirtualXRay/Shader.h"       // Handle GLSL programs

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

Listing 1: 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 [**GLFW**].

- `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.
- `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.
- `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.
- `glfw3.h` is the main GLFW header file.
- `iostream` is used for output streams.

- `exception` is used for C++ exceptions.
- `Types.h` defines new types, e.g. `RATIONAL_NUMBER`, `VEC2`, `VEC3` and `MATRIX4` to name a few.
- `Units.h` defines units such as metre, kilometre, electronvolt, kiloelectron volt, gram, kilogram, etc.
- `PolygonMesh.h` corresponds to a class used to handle 3D triangle meshes.
- `XRayBeam.h` corresponds to a class used to manage X-Ray beams. The beam spectrum is discretised into energy channels.
- `XRayDetector.h` corresponds to a class that handles virtual X-Ray detectors.
- `XRayRenderer.h` is used to compute X-Ray images on the Graphics Processor Unit (GPU).
- `Shader.h` corresponds to a class that handles OpenGL Shading Language (GLSL) programs.
- `buildCube.h` is used to create the triangle mesh of a cube.

3 Name Spaces

```
using namespace gVirtualXRay;
using namespace std;
```

Listing 2: Name spaces.

Listing 2 shows the name spaces that can be selected.

- `gVirtualXRay` is used to defined all the components of the X-ray simulation. It also includes graphics elements such as `PolygonMesh` and utilities such as `Exception`.
- `std` is used for output streams and exceptions.

4 Setting Components of the X-Ray Simulation

Figure 1 illustrates the main components of the simulation.

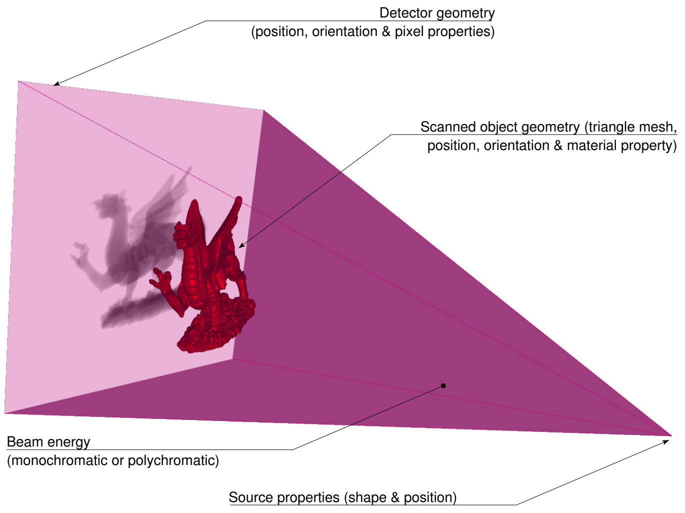
Detector geometry
(position, orientation & pixel properties)
Scanned object geometry (triangle mesh,
position, orientation & material property)
Beam energy
(monochromatic or polychromatic)
Source properties (shape & position)

Figure 1: Definition of the main components needed to simulated X-ray images.

```
XRayDetector g_xray_detector;
XRayBeam     g_xray_beam;
XRayRenderer g_xray_simulator;
PolygonMesh g_polygon_mesh_1;
PolygonMesh g_polygon_mesh_2;

```

```
MATRIX4 g_sample_rotation_matrix(MATRIX4::buildRotationMatrix(10.0, VEC3(1, 1, 1)));
```

Listing 3: Declaration of the simulation components.

Four of them have to be defined to simulate X-Ray images:

1. The detector (its position, orientation, number of pixels and the size of pixels); and
2. The source (it includes the shape and position of the source);
3. The beam (its energy spectrum);
4. The scanned objects (their topologies, positions, orientations and material properties).

To facilitate their integration into the GLFW example, they are declared as global variables (see Listing 3). `g_sample_rotation_matrix` is the modelling matrix that is used to transform the scanned object. By default it is a identity matrix. It could be used to move, rotate and scale the object. The `Matrix4x4` class provides such functionalities.

4.1 X-Ray Detector

```
//-----------------------
void setDetector()
//-----------------------
{
  // Number of pixels of the detector
  Vec2ui number_of_pixels(640, 640);
  g_xray_detector.setNumberOfPixels(number_of_pixels);

  // Size of pixels (in unit of length)
  RATIONAL_NUMBER resolution(320.0 * mm / number_of_pixels.getX());
  g_xray_detector.setResolutionInUnitsOfLengthPerPixel(resolution);

  // Position of the detector
  VEC3 detector_position( 0.0, 0.0, 10.0 * cm);
  g_xray_detector.setDetectorPosition(detector_position);

  // Orientation of the detector
  VEC3 detector_up_vector(0.0, 1.0, 0.0);
  g_xray_detector.setUpVector(detector_up_vector);
}
```

Listing 4: Setting up the X-ray detector.

Listing 4 shows how to initialise the detector. Four attributes have to be set:

1. Number of pixels of the detector;
2. Size of pixels (in unit of length);
3. Position of the detector;
4. Orientation of the detector.

Note: see how lengths are set in order to internally store them all in the same unit, e.g. `320.0 * mm`, or `10.0 * cm`.

4.2 X-Ray Source

```
//---------------------
void setSource()
//---------------------
{
  // Set the source position and shape
  VEC3 source_position(0.0, 0.0, -40.0 * cm);
  g_xray_detector.setXrayPointSource(source_position);
}
```

Listing 5: Setting up the X-ray source.

Listing 5 shows how to set the position and shape of the source. In this case, the shape corresponds to an infinitely small point. Other shapes can be chosen.

4.3 Beam Spectrum

```
//--------------
void setBeam()
//--------------
{
  // Set the energy
  RATIONAL_NUMBER incident_energy(80.0 * keV);
  g_xray_beam.initialise(incident_energy);
}
```

Listing 6: Setting up the spectrum of the incident beam.

Listing 6 shows how to set the position and shape of the source. In this case, the beam spectrum is monochromatic and the energy of all the photons is 80 kiloelectron volts (keV). Polychromatism can be chosen, in which case a beam spectrum has to be loaded.

Note: see how energies are set, e.g. `80.0 * keV` for 80 keV.

4.4 Scanned Object

```
//-----------------------
void setScannedObject()
//-----------------------
{
  // Create the cube
  VEC3 cube_centre(0, 0, 0);
  RATIONAL_NUMBER length_1( 5.0 * cm);
  RATIONAL_NUMBER length_2(10.0 * cm);
  buildCube(length_1, cube_centre, g_p_vertex_set_1, g_p_index_set_1);
  buildCube(length_2, cube_centre, g_p_vertex_set_2);

    // Set geometry
  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);

  // Set the material properties
  g_polygon_mesh_1.setHounsfieldValue(50.0);
  g_polygon_mesh_2.setHounsfieldValue( 0.0);
}
```

Listing 7: Setting up the properties of the scanned object.

In most simulation cases, the topology of the scan objects will be stored in the main memory as an array of triplets of floating point numbers. Each triplet corresponds to a vertex. An index of vertices can also be given to reduce the memory consumption, and eventually speed-up the computations. This is a well known practice in computer graphics. Two objects are considered here (see Listing 3). The first one makes use of two arrays (one for vertices and one for indices); the second one makes use of a single array (for vertices).

Listing 7 shows how to set the properties of the scanned objects. `buildCube(...)` is used to initialise the topology of the two scanned objects as cubes. The topology of each cube is then passed on to instances of PolygonMesh. Note that in this case, the memory is managed by the calling program, not by the class instance. It is also possible to load STereoLithography (STL) binary files, in which case the PolygonMesh class instance will manage the memory.

The next step is to set the material property (here the Hounsfield value).

4.5 Simulation Environment

```
//-------------------
void setSimulator()
//-------------------
{
  // Initialise the X-ray renderer
  g_xray_simulator.initialise(XRayRenderer::OPENGL,
      GL_RGB16F,
      &g_xray_detector,
      &g_xray_beam);

  // Add the geometry to the X-ray renderer
  g_xray_simulator.addInnerSurface(&g_polygon_mesh_1);
  g_xray_simulator.addOuterSurface(&g_polygon_mesh_2);
}
```

Listing 8: Setting up the simulation environment.

Listing 8 shows how to set the simulation environment. Only the OpenGL simulation has been implemented. For fast computation, use GL_RGB16F_ARB; for accurate results, use GL_RGB32F_ARB. This parameter controls the number of bits used by floating point number. Finally, the detector, source and geometries have to be registered. One may note that the biggest cube, which totally enclose the smaller one is used as OuterSurface (it would be the case of the skin surface in the case of a medical simulation). The smaller one is used as InnerSurface (it would be the case of bones and internal organs in the case of a medical simulation).

5 Running the Simulation

```
  // Compute the X-Ray image
  if (g_xray_simulator.isReady())
  {
    g_xray_simulator.computeImage(g_sample_rotation_matrix);

    // Normalise the X-ray image
    g_xray_simulator.normalise();
  }
```

Listing 9: Running the simulation.

To run the simulation, call `XRayRenderer::computeImage(...)` with the scanned object transformation matrix as parameter (see Listing 9). It may be useful to call `XRayRenderer::normalise()` to make sure that the X-ray image is normalise before it is displayed. The image can be displayed in negative mode, or not. To enable and disable this mode, use `XRayRenderer::useNegativeFilteringFlag(boolean_flag)`.

```
    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);
    if (!g_p_window_id)
    {
        glfwTerminate();
      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);

}


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

    g_window_width = width;
    g_window_height = height;
```

Listing 10: Display callback.

Listing 10 shows the GLUT display callback. One may note that the X-ray beam is displayed using transparency. It is the reason why it is displayed last. Figure 2 shows the corresponding rendering.

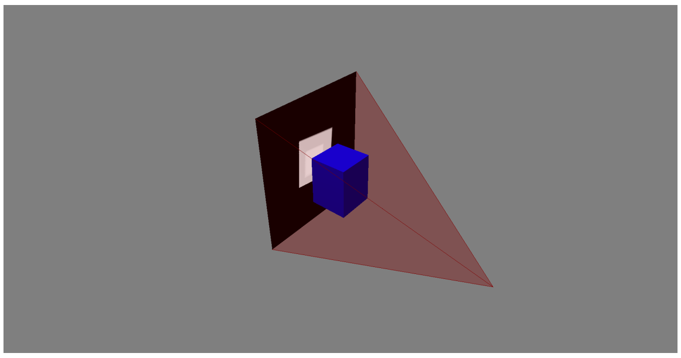

Figure 2: OpenGL rendering.

6 Next Tutorial

In the next tutorial:

- You will see how to get rid of OpenGL's fixed pipeline (which is now depreciated).
- You will see how to get rid of OpenGL's matrix stack (which is now depreciated).
- We will see how to create an efficient mouse control to turn the 3D scene, object, and detector (the detector does not have to be centered and orthogonal to the beam).
- You will also see how to update the topology of the triangle meshes. This is important in medical simulations with soft tissue deformations.