








1. Introduction

This document is a brief introduction to the main elements of the OpenGLFramework we will be using during our practices. You can download this framework from the course's web page (<https://www.cs.bris.ac.uk/Teaching/Resources/COMS30115/>).

We will introduce the most important parts of the framework. We will first describe its general structure, explaining their folders and what can be found in each of them. Then we will describe the main modules of the framework, describing their classes and interfaces. Finally, we will describe how to make use of the framework, that is, how to create a main application using it.

2. General Structure:

The framework is divided in four main folders:

 common	24/02/2015 16:49	File folder	
 Content	24/02/2015 16:49	File folder	
 external	24/02/2015 16:52	File folder	
 FBO	24/02/2015 16:49	File folder	
 shaders	24/02/2015 19:29	File folder	
 OpenGLFramework.cpp	24/02/2015 19:32	C++ Source file	4 KB
 OpenGLFramework.h	12/02/2015 16:26	C++ Header file	2 KB

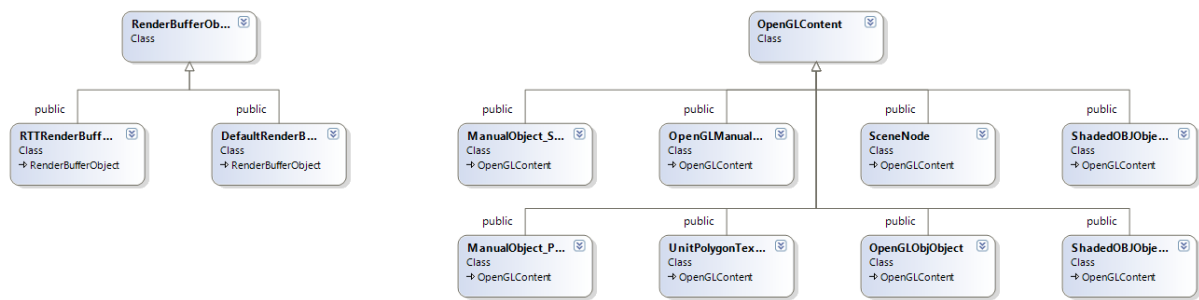
- **OpenGLFramework (.h and .cpp):** Basic entry point for the framework. This is all you need to #include from your main file to make use of all the functionality in the framework. It publishes some global methods required to manage the application (create window, set up OpenGL), that we explain in more detail later
- **Content:** Contains a collection of classes that represent elements that we can render using OpenGL.
- **FBO:** Contains a collection of classes that allow us to render a view of the 3D world. These can either allow us to a render to a monitor straightaway or render the view to a texture.
- **shaders:** As we explained in class, only a part of the code is actually C++ running in the CPU. The rest is run as shaders, which are executed in the GPU. These files contain the GLSL code
- **external:** contains the external libraries required to run the framework.

3. Class diagram:

There are two two base classes that require special attention: OpenGLContent and RenderBufferObject:

- **OpenGLContent:** Describes anything that can be rendered.
- **RenderBufferObject:** Describes a surface onto which we can render, let it be the monitor or a texture.

The framework makes extensive use of these two classes as the basic interface of most elements of the framework. We will focus on their internal details now.



OpenGLContent:

OpenGLContent is the basic interface for anything we render using OpenGL. Its interface is inspired in the explanations given in the lectures. As we mentioned then, in order to render anything in OpenGL, we need both GLSL code and C++ code. We also needed to copy whatever data we needed from main memory to the graphics card and handlers to specify how each chunk of data should be use (as vertex positions, colours, texture coordinates, etc.).

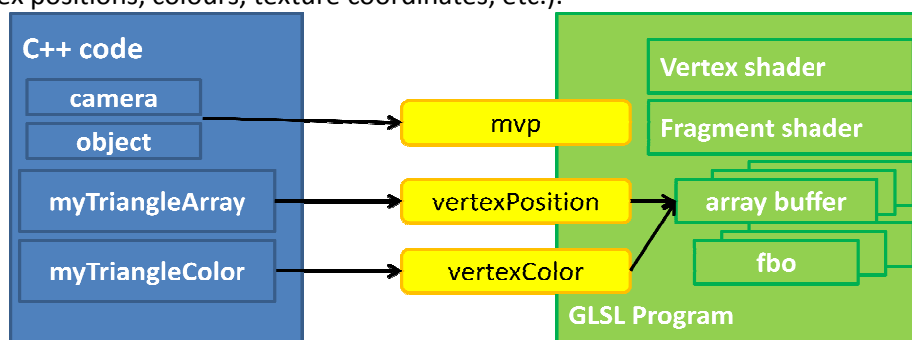


Figure 2: Main elements required to render something in OpenGL

Each subclass of OpenGLContent represents this small “ecosystem”. That is, it contains all the elements required to render something using a specific set of vertex and fragment shaders (we did not include any example using other types of shaders).

Actually, each of the classes in “/Content” simply represent this, a particular way in which you can render something. At the very top of the “.h” files of each class you can find comments describing what the class does, the GLSL shaders it uses and relevant details about it. The classes in the framework are pretty much the examples that we gave during the lecture in “Advanced OpenGL”, and should be easy to follow. A suggested order to go through them (with increasing complexity and following what was presented in the slides) could be:

- ManualObject_SingleColour
- ManualObject_PerVertexColour
- UnitPolyginTextured
- OpenGLManualObject
- ShadedOBJObjectDirectionalLight
- ShadedOBJObjectPointLight

Methods:

OpenGLContent describes the methods that describe the “life cycle” of an object. Again, this is inspired in the structure we used during the lectures when we described the C++ code.

- *loadContentsToMainMemory*: This method prepares the contents in main memory that will be required to render the object. This can include describing arrays of vertices, textures, etc. In Fig 2., this would correspond to creating the C++ data-structures *myTriangleArray*, *myTriangleColour*, *camera* and *object*.

- *allocateOpenGLResources*: This method copies all the data to the graphics card and gather the required handlers that will allow us to use the data later, feeding it to our shaders. In Figure, this method would: a) compile the GLSL program that uses our fragment and vertex shader, b) retrieve all the yellow handlers and c) create the buffer objects and fill them with our data. This all corresponds with the steps we repeatedly showed in the slides as “Preparing the *attributes* and *buffers* storing their values”.
- *Render*: This methods sets the object’s program and attributes as active and triggers rendering. This all corresponds with the steps we repeatedly showed in the slides as “Set our *attributes* and *buffers* active and ...**Render**”. The method has two arguments, the View and Projection matrices to use for rendering. Some objects are always rendered in “Clipspace coordinates” instead of “world coordinates” and do not use these attributes.
- *unallocateResources*: It deletes the resources allocated, both from the graphics card (buffers, textures, unload the program) and from main memory.

Some auxiliary methods are also included that allow you read each object’s unique identifier (*getID*) and to change/retrieve the model matrix of the object (the transformation matrix that describes its position and orientation).

RenderBufferObject

The class *RenderBufferObject* represents a "surface" where you can render. There are two subclasses for it, one representing the default monitor (*DefaultRenderBufferObject*), and another one to be used to render to texture (*RTTRenderBufferObject*). All these classes can be found in the FBO folder of the framework.

Users are not allowed to create these objects directly, but they need to do this indirectly through the base class *RenderBufferObject*, using the methods:

```
static RenderBufferObject* createDefaultRenderBufferObject();
static RenderBufferObject* createRTTBufferObject(int width, int height);
```

After creation, these classes define two methods (*preRender* and *postRender*) that need to be called before and after rendering the contents to prepare the rendering surface (clearing colour and depth buffers, etc.).

4. Client Application

The file *templateFramework.cpp* serves as an example of what a client application needs to do in order to build the framework. The comments should help you follow this explanation.

First, we declare *FrameBufferObject*’s and *OpenGLContent*’s variables that will contain the objects that build our application. We define them globally in the main so that they are visible in all the methods in the file. This is fine if you have a small amount of objects, but it can get tricky as it scales up. An alternative would be to have the variables and methods in this file implemented as a class.

Then we move into the four main methods that will build our application:

```
//1. Methods required to build our program:
//1.1. Create the contents and place them where appropriate
void createWorldScene();
//1.2. Render our world
void renderWorldScene(RenderBufferObject* rf, glm::mat4 P, glm::mat4 V);
//1.3. Update the world according to: user events (inputs), time, etc...
void updateWorldScene();
//1.4. Deallocate the resources once our program is finished.
void destroyWorldScene();
```

The method *createWorldScene* is called only once during the initialization of the program. In this method, we create the actual *FrameBufferObjects* and *OpenGLContents* by calling their constructors. We also make sure all necessary resources are allocated into the graphics card and ready to render, by calling *loadResourcesToMainMemory* and *allocateOpenGLResources* on each *OpenGLContent*

The method *renderWorldScene* is called every frame, once the application is started. Its body is real simple: the *FrameBufferObject* is prepared and render is called on all *OpenGLContents*.

The method *updateWorldScene* is also called every frame, and this is where all the changes to the world happen (due to user's interaction, events, animations, AI algorithms, ...). At the minute, this is where you will find the code that allows use the mouse and keyboard to control the camera and to detect when ESC is pressed, to exit the program.

The method *destroyWorldScene* is called at the end of the program and simply makes sure all resources are de-allocated.