Computer Graphics – Lab Session 1

Introduction:
In this session you will revisit some of the concepts of OpenGL explained during the last two lectures.

- You will learn a bit more about shaders, their behaviours, attributes and communication from the C ++ code.
- You will play with different OpenGL primitives (points, lines, vertices), which will help you understand the rendering pipeline a bit better.
- You will see when/how different tests are applied (depth buffer, back face culling) and their impact in an application.
- You will practice describing some geometries and modifying their transformation matrices to produce several effects.
- You will implement some advanced effects, which will help you understand how flexible OpenGL is when dealing with the data describing an object.

Exercise 1 : Framework
In this first exercise we want you to familiarise a bit with the framework that we will use during our lab sessions. We actually want you to get to understand this code in depth, playing with it, adding and removing settings, new types of objects, etc.

Start by looking at *templateFramework.cpp*. This is the main file and describes how the framework is initialised/stopped (*createWindow*, *setupOpenGL*, *programEnded*, *destroyWindow*). It also identifies the main tasks that you need to complete when creating any program (creating, rendering, updating and destroying the scene).

You also need to familiarise with two important classes in the framework: OpenGLContent and RenderBufferObjects.

- Subclasses of OpenGLContent can be added to the framework to describe objects with a specific rendering technique. You will find the framework already contains a good part of the objects/shaders that we used during the lectures.
- (Check out by yourself what RenderBufferObjects are for...).

Here is a list of the steps you can follow:
a) Download the Framework from the unit website (or directly https://www.cs.bris.ac.uk/Teaching/Resources/COMS30115/Lectures/LabFramework.zip)
b) Extract it on your Desktop
c) Start the Eclipse IDE with /usr/local/eclipse-luna/eclipse
d) Set the Workspace to your Desktop
e) Open the project using Eclipse : Menu File->Import->Import->General->Existing projects ... and select the LabFrameworkLinux64 folder as the Root Directory
f) Build the project (Ctrl-B), Refresh it (F5) and Run it (Ctrl-F11)

You will see a red triangle, built using the class ManualObject_SingleColor. The data describing the triangle can be found in templateFramework.cpp.
You can also see the three axes that represent the centre of the scene (how are these defined?). As we move around the scene, the axes remain in a fixed point in space, while the triangle is always in front of our camera.

Exercise 2 : Model View Projection
The triangle (Object B) is for now defined in clip space. That is why when we move the camera, the triangle stays in front of us. The problem is that the view and projection matrices gathered from the (mouse and keyboard) inputs are not taken into account. The purpose of this exercise is to add the view and projection matrix to enable moving around the triangle.

a) Add the Model View Projection matrix to Content/ManualObject_SingleColor: Check its shader and decide whether anything needs to be added. Modify the C++ to retrieve the required attributes and send the camera data to the shader.
b) Try moving the camera around the scene.
c) Play with the projection parameters in templateFramework::updateWorldScene() to understand the perspective projection
d) Try using glm::ortho instead

Exercise 3 : Primitives
During our lecture we paid special attention to the vertex and fragment shaders. We explained how both vertices and fragments shared their attributes and that the values in each fragment attributes are interpolated from those of the vertices that originated them.

In order to fully understand how this rasterizing happens, we need to be aware of the different types of primitives that OpenGL support. So far, we fed our input array of vertices to OpenGL without thinking twice about its internal format. And this is totally fine with vertex buffers, who process each vertex independently of all the others.

However, once the vertices have been processed, OpenGL takes into account the kind of primitive that the elements in the buffer describe. If we use triangles, it will take each three consecutive vertices as a *primitive*. If we use lines, it will consider each two vertices as a line. This will influence the way new geometries are included (tesselation), but also how the rasterizing takes place.

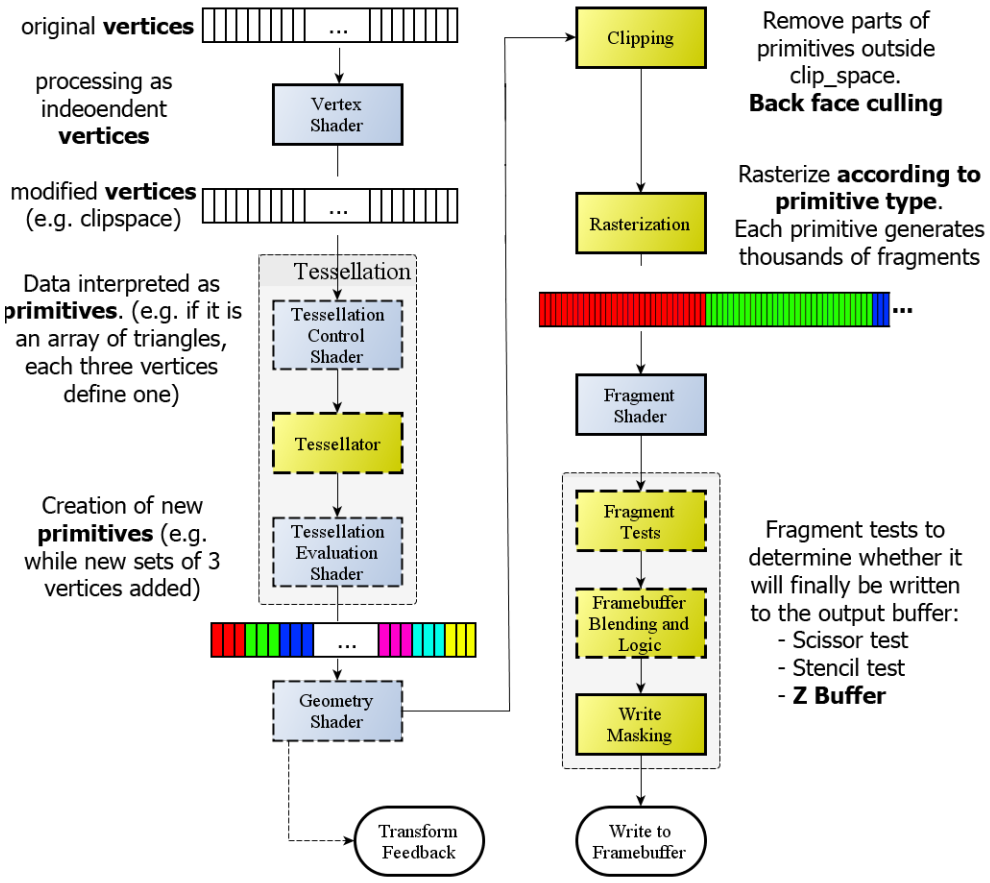


Figure 1: OpenGL rendering pipeline.

In this exercise we will play with the different OpenGL primitives.
a) Render the triangle with the different primitives (points, lines, triangles)
b) Experiment with `glPointSize`

Exercise 4 : Geometry

a) Turn the triangle into a tetrahedron (http://en.wikipedia.org/wiki/Tetrahedron) by adding vertices and their position attributes

b) Display it with the ManualObject_PerVertexColor object

Exercise 5 : Transformations

You are now going to manipulate your tetrahedron, e.g. translate and rotate it by manipulating the model matrix

a) Try to first translate and then rotate the pyramid

b) With the same values, try now to first rotate and then translate

c) How do you explain the difference? Use pen and paper and try to find an explanation.

Exercise 6 : Depth

a) Is there something wrong with the rendering of the tetrahedron ?

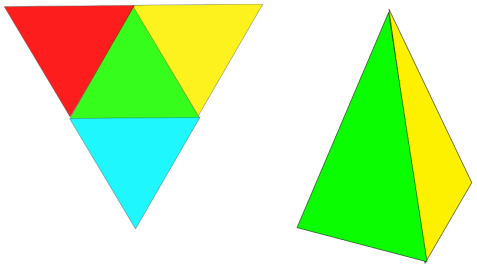
b) Go to the setupOpenGL in OpenGLFramework.cpp, what is missing ?

c) Experiment with the activation of the depth buffer and the face culling (front faces, back faces, both)

Exercise 7 : Attributes/Uniforms

In this exercise, you will see the difference between two types of data that can be passed from the C++ code to the Shaders in the graphics pipeline: attributes and uniforms. Attributes are defined for each vertex separately in buffers while uniforms are defined once and used by all the vertices. The use of both is not constrained by OpenGL. For example attributes are not limited to vertex positions and colors.

To understand that, you will create a shape morphing effect between the two shapes shown below.



a) Create a new class ManualObject_Morph by copying ManualObject_PerVertexColour

b) Define a "positionSecondShape" attribute and its the values for all vertices using an array and VBO. Add a uniform "morph" of GL_FLOAT. Set its value to 0.5.

c) Create MorphPerVertexColor.vertexshader

d) In it, add the "positionSecondShape" attribute and "morph" uniform. Use the morph variable to change the output vertex position to the interpolation between the "vertexPosition" and "positionSecondShape".

e) Animate the morphing by updating the morph uniform in the render method (for example use the cos function and gettimeofday)

Exercise 8 : .obj 3D Model File Format

a) Find and download a 3D model in .obj format that you want to display (try http://opengameart.org/ for example)

b) Open and analyse the file. Which attributes are used ?

c) Which object of the framework can you use to display it ?

d) Create the needed attributes and shader to display the object

e) You will present the object that you have displayed at the next session