Introduction to Modern OpenGL

3

The success of OpenGL as a 3D graphics rendering framework attracted the interest of the mobile devices developers, as these have reached a computing power performance enough to support that. These new platforms have brought new challenges and performance concerns, so OpenGL ES 1.0 appeared in 2003 as an adaptation of OpenGL 1.3 by removing some of the "heavy" functionalities. The evolution of this framework both on desktop computers and mobile devices was clearly boosted from this moment on in particular by the evolution of the graphics processing engines (GPU) that were becoming more powerful every day. In fact architectural changes in hardware were introducing more and more computing power that could not be easily exploited by the traditional API. Progressively evolved from a fixed graphics pipeline in versions 1.X to one that brought the possibility of optionally supplying vertex and fragment shaders in version 2.0, to the mandatory use of these shaders from version 3.1 on.

3.1 From models to graphics

Despite all the possibilities that exist for modelling objects and scenes composed of these objects based on more or less complex and powerful mathematical formulations, most graphics engines require these models to be made of planar polygons, in the simpler case triangles. Thinking of a model as big collection of triangles and therefore of points, what is needed to create the scene where this model repeatedly appears is a sequence of afine transformations and then a projection to generate the view.

If we consider $\{House\}$ and $\{Tree\}$ as the two sets of points that compose the respective models of a house and a tree, then we can build some

scene by instantiating multiple copies of these point sets and moving them to the appropriate positions and orientations, by applying the corresponding modelling transformations $\mathbf{T}_n$.

$$\{House\}_1 = \mathbf{T}_{h1}\{House\}$$

$$\{Tree\}_1 = \mathbf{T}_{t1}\{Tree\}$$
$$\{House\}_2 = \mathbf{T}_{h2}\{House\}$$
$$\{Tree\}_2 = \mathbf{T}_{t2}\{Tree\}$$

$$...$$

$$\{House\}_N = \mathbf{T}_{hN}\{House\}$$
$$\{Tree\}_N = \mathbf{T}_{tN}\{Tree\}$$

3.2 Moving the camera and generating different views

One of the main motivations for building 3D models is the possibility of viewing them from different points. The easiest way to understand this is to think of a camera moving in a world or scenery, while taking pictures of it.

As for the objects, moving a camera to a given position in the world and directing it towards some place corresponds to translate it by some vector $\mathbf{t}_{cam}$ and rotate its attached frame axes by some rotation matrix $\mathbf{R}_{cam}$, with respect to the world reference frame.

Seen from the camera the world is suffering the inverse transformation, so if the camera is moved by

$$\mathbf{T}_{cam} = \left[\begin{array}{ccc|c} & \mathbf{R}_{cam} & & \mathbf{t}_{cam} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \begin{bmatrix} \mathbf{R}_{cam} & \mathbf{t}_{cam} \\ \mathbf{0} & 1 \end{bmatrix}$$

therefore its inverse is

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} \mathbf{R}_{cam}^{-1} & -\mathbf{R}_{cam}^{-1}\mathbf{t}_{cam} \\ \mathbf{0} & 1 \end{bmatrix}$$

Now going back to the houses and trees, we can compute their poses with respect to the camera as

$$\{House\}_{1c} = \mathbf{T}_{cam}^{-1}\mathbf{T}_{h1}\{House\}$$

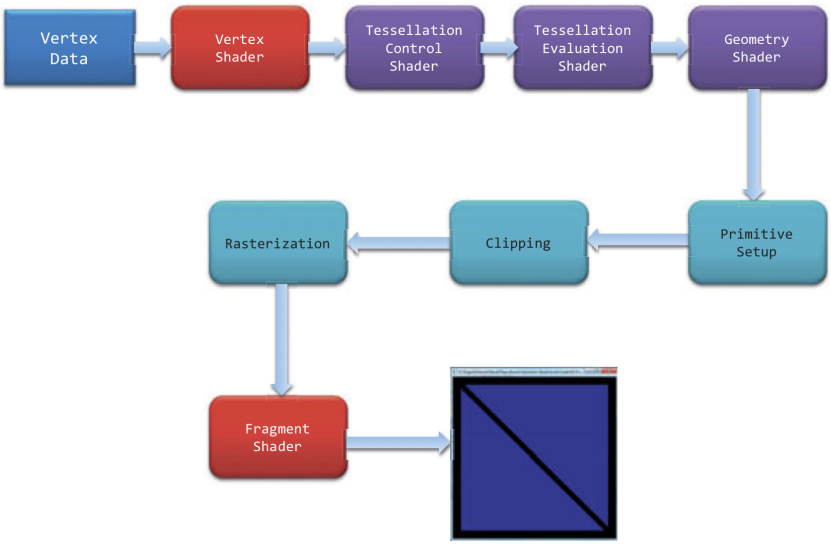

Figure 3.1: The OpenGL 4.6 pipeline

$$\{Tree\}_{1c} = \mathbf{T}_{cam}^{-1}\mathbf{T}_{t1}\{Tree\}.$$

The matrix $\mathbf{T}_{cam}^{-1}$ is called the *view matrix* and $\mathbf{T}_{h1}$ and $\mathbf{T}_{t1}$ the *model matrices*.

3.3 Shaders

Figure 3.1 depicts the processing pipeline of OpenGL current versions. There is place for 5 shaders with different purposes, where the vertex and fragment shaders are mandatory and the remaining optional.

Starting with the vertex shader we may say that its primary purpose is to apply all the necessary geometric transformations and compute the projection of the vertices that compose each part of the models or scene. In reality this projection is not performed as expected from 3D space to a 2D image plane, but we may think of it like that by now.

This shader will receive each of the vertices, and may receive also the model matrix, the view matrix and camera matrix (projection matrix). Its output is the position of the corresponding projections in the image plane. This means that the C/C++/Python program will prepare these matrices, and will pass them to the GPU along with the model vertices to be projected.

```
GLfloat vertices[NumVertices][2] = {
{ -0.90, -0.90 },{  0.85, -0.90 }, { -0.90,  0.85 },// triangle 1
```

```
  {  0.90, -0.85 }, {  0.90,  0.90 },{ -0.85,  0.90 } // triangle 2
};
```

Let us start with a simple example of drawing a pair of triangles defined by their 2D coordinates above. These two triangles will fill almost the whole window despite of its size. This happens because instead of providing space coordinates, we are going to consider that these coordinates are already normalized device coordinates (NDC) with the third component (here omitted) equal to zero. In fact after the vertex shader every vertex should be transformed so that they fit inside a cube centred on (0,0,0) and each coordinate range from -1 to 1.

So lets prepare the buffer that will receive the above vertices, and copy the vertices data into it.

```
GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];
const GLuint NumVertices = 6;
glGenVertexArrays(NumVAOs, VAOs);
glBindVertexArray(VAOs[Triangles]);
glGenBuffers(NumBuffers, Buffers);
glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),vertices, GL_STATIC_DRAW);
```

The next step is to load and compile the shaders, and define that we are passing 2 components of each vertex to the first shader

```
myprog->loadShaders("triangles.vert","triangles.frag");
myprog->linkShaderProgram();
glVertexAttribPointer(vPosition, 2, GL_FLOAT,
                                    GL_FALSE, 0, BUFFER_OFFSET(0));
glEnableVertexAttribArray(vPosition);
```

The drawing function will start by activating the shaders that are to be used with the object. Note that different objects may use different shaders, as long as the corresponding is made active prior to drawing each of them. The clean the screen, select the desired *Vertex array object* and call the appropriate primitive drawing function.

```
  myprog->startUsing();
  glClear(GL_COLOR_BUFFER_BIT);
  glBindVertexArray(VAOs[Triangles]);
  glDrawArrays(GL_TRIANGLES, 0, NumVertices);
  glFlush();
  myprog->stopUsing();
```

Exercise 3.1

Download the examples archive L3.tgz from Inforestudante. Extract the files and compile them with

```
  make -f Makefile.linux
```

Run the program triangles, and analyse the code of files triangles.cpp and deecshader.cpp and deecshader.h. Explain each function or method.

3.4 Uniforms

As we have seen before the vertex shader receives a single vertex data for performing its computations. So it may be seen as having a different program (or thread) for each vertex. However it is common to have the processing of the whole set of vertices controlled by some parameter that may change during the run time. Although there was the possibility of adding this parameter repeated for each vertex, this would represent unnecessary overloading both in preparing the data and on the communication between CPU and GPU. For this a different type of data exists which are called uniforms, and consists of one input parameter that is shared between (and common to) the whole set of vertex processing threads.

Exercise 3.2
Add to the triangles vertex shader the following line

```
uniform float timepar;
```

add the following line somewhere after having linked the shader program.

```
GLuint timeLoc   = glGetUniformLocation(myprog->shaderprog, "timepar");
```

and the following on the display function

```
glUniform1f(timeLoc, timevariable);
```

Create a new version of the shader (in a new file) so that the y coordinates of the vertices are added to the sinus of some factor of time, e.g. $y = y + K_1 \sin(timepar/K_2)$.

Exercise 3.3
Create a new version of the shader so that it varies the scale of the object displayed.

Exercise 3.4
Create a new version of the shader so that it rotates the triangles about the screen centre.

Exercise 3.5
Now similarly add a uniform input variable as a vector of 3 components to the triangle fragment shader, and do the modifications necessary to vary the

3 color components along time. You can apply this to any of the previous developed examples.