Graphic Engine

Computer Graphics 2018/ 2019

## Gregorio Meyer, Cem Koca and Julian Petralli

2019

# Introduction

This project aims to develop two distinct components: a 3D graphics engine (as a library) and

an animated gauntlet (as an application that uses the 3D graphics engine).

The 3D graphics engine uses some external libraries which aren’t visible on the client side. The libraries used are:

* **FreeGLUT**: is an open-source alternative to the OpenGL Utility Toolkit (GLUT) library it allows the user to create and manage windows containing OpenGL contexts on a wide range of platforms and also read the mouse, keyboard and joystick functions.
* **GLM**: OpenGL Mathematics, a header only C++ mathematics library for graphics programming
* **FreeImage**: FreeImage is an Open Source library project for developers who would like to support popular graphics image formats
* **OvoReader**: plugin given by our teacher to export a scene created with 3ds Studio Max to an .OVO file.

The 3D graphic engine allows to read various scenes created in 3ds Studio Max complete of lights, materials and texture and provides a series of features like:

– Scene-graph manipulation.

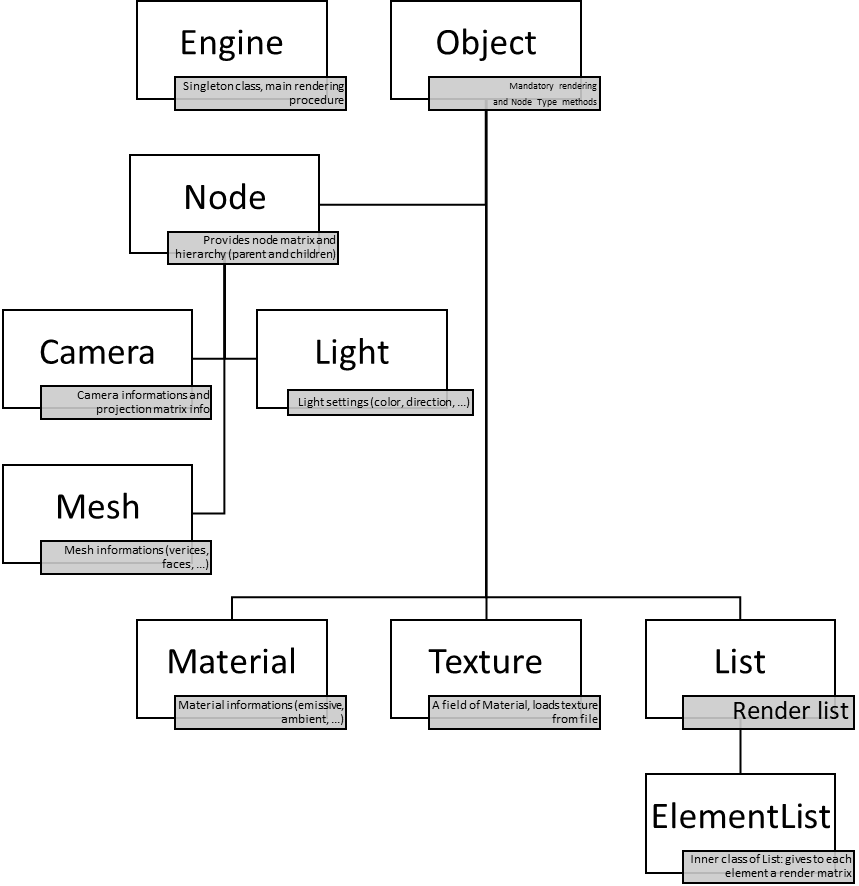
– Dynamic light sources and cameras.

– Texture mapping and loading.

– Transparency

# Engine structure

The structure of our engine is represented in the following hierarchy:



The base of our structure is the Object class, used by all the derived classes. This class is responsible for forcing some required API methods, keeping track of existing objects, and providing a unique ID to each object.

Information of all Objects come from the ovo file except Camera. The ovo file is read recursively.

Initially we read the TYPE\_NODE case in which we create e new node and we set name and id, then we call a method named readChildren() in a for loop that has the size of the children number. It will add all children to this node and for each child (if present) will call recursively the same method for reading all his children. This happen for types: Node, Light, Mesh. For each Mesh we control if there is an associated material and texture data and we assign them to their mesh (this is possible because materials are the first extracted from the scene).

For reading materials is a little bit different, we add them in a special list and we associated a Texture (if present). The list will be passed when a Mesh is recognised.

Follows a more detailed description of each element.

# Engine

For our library, this is the principal class: first of all, it is the only and unique method for the communication between client and engine. Secondly, it provides many wrapper methods around FreeGLUT like the initialization of his and FreeImage context, many wrapper functions for FreeGLUT callback, all keyboard functions and all method which aim the helicopter moving.

We provided these possibilities to move our helicopter:

* move up: Spacebar
* rotate left/right: l and r
* move front: w
* move back: z

The engine class has its internal pointer used to save the reference to the graph’s root,

consequently user has to set it explicitly by passing the global root as parameter before

starting the render phase.

Once the client has created the context and called necessary callback, he can build his own scene and with the function “readOVOFile(‘filename’)” he can automatically build the scene graph recursively. Every Object, Mesh, Light, etc. is ridden from the ovo file, except the Camera, this is managed at client side.

We decided to have a field with a reference to the active camera and a vector with all cameras to make easier the switch between those last.

Our engine provides also the possibility to set a camera to a specific node with this method:

void setCameraToNode(Node\* root, string cameraName, string nodeName);

In our case is used to set the camera to the Helicopter, cameraName will be used to find the desired camera and this will be set as children of the node found by nodeName.

Once this is done, the display callback is called by FreeGlut and many methods will be also called, like:

* pass(): it instances lists of objects, lights and transparent meshes with the scene graph traversal. Then it populates a single render list.
* renderElementsList(): it takes the list previously created and call the render method for each element.
* gravityEffect(): it simulate a gravity effect to the helicopter.
* rotateElise/Rotor(): it apply to elise and rotor a rotation.

Pass is called two times because of the reflection, once is called with an identity matrix and the second one is called with this scale matrix:

glm::scale(glm::mat4(), glm::vec3(1.0f, -1.0f, 1.0));

The difficult part of this big class was to build a correct tree from ovo file, at the beginning we tried to read it a linear way (since we knew the number of children of each element, we wanted to add them one to one in the tree in the correct position). But after many try, we implemented it in a recursive way. This has been detected the easier and the fastest approach.

# Camera

The camera class is the only which is not present in the scene graph, so it is independent.

To add a camera the client has to call the following method, passing by the Engine class:

Camera\* Engine::addCamera(string name, glm::vec3 eye, glm::vec3 center, glm::vec3 up);

This will create a new camera and will call the glm::lookAt method to set his position.

It is possible to instance infinite cameras and the engine class provides a method (called with ‘c’ key) with which the client can switch between cameras.

We created a method that sets the gl matrix mode:

void Camera::matrixMode(int type);

The type field can be 0 or 1; 0 for *GL\_PROJECTION* and 1 for *GL\_MODELVIEW*.

Camera inherit from Node so it has a personal matrix, but this class has also a projection matrix.

Since our camera isn’t in the scene graph, the complicated thing to do was, for the helicopter camera, to add the camera as child of helicopter.

# Object

Object class is the root of hierarchy tree, each class in tree derive from this one.

We provided only a .h file because there is only name and id getters and setters.

Id is a field of Engine class and each time a node is added, we have to increment this last.

In addition, this class has two virtual mandatory methods: render and getNodeType. The second is managed by a separate enum and each node returns one of this fields.

# Node

The Node Class works as the ruling of the scene graph, and at its internal has the positioning

matrix in world coordinates.

All the Nodes must be linked to the Node root directly or by a parent Node, so every Node

has a father, except the logic Node root which will have its father pointer to null.

Each Node has its internal vector of node pointers which represents its children.

We provided methods for: removing a child by passing an index or a node pointer, add children (which set automatically the parent).

The render method was overwritten because the inherited class possess the same static

method, so basically, the render method does nothing.

# List

This class is used to save all elements from the scene graph. Each element is saved in an inner class named “Element” which is composed by a pointer to an object and the final matrix. We decided to put an Object pointer because we want to be more generic.

This class is composed only by a std::list<Element\*> and it is possible to add, remove and find a list element.

The list is populated from the Engine class by this method:

void Engine::populateListFromTree(glm::mat4 fatherMatrix, Node\* root);

This will pass the scene graph recursively and to have automatically the final matrix already multiplied with the parent’s matrix, this is passed in the method.

The render method was overwritten because the inherited class possess the same static

method, so basically, the render method does nothing.

Difficult of this class was to pass from scene graph to the list because of the element’s order. For this reason we decided to do many Lists, for example for lights, meshes, transparent meshes and when we pass the scene graph each element will be put in the correct list. After this, each list is merged in a single list and this call render for each element.

# Mesh

This is a class which derives from Node which derives from Object, so must have render and getNodeType. Render method was, initially, implemented by drawing each triangle with glBegin() and glEnd() using vertices and faces which came from the ovo file.

Later, when we had almost finished this project, we noted that we didn’t arrive at a sufficient number of FPS and so our application could not be considered a realtime application; for this reason we decided to use “*vertexArray”.*

This technique uses some arrays that aims to save vertices, but also other information as: vertices position, normal vector, texture coordinates and colour information. And you can draw only a selection of geometric primitives by dereferencing the array elements with array indices. So you have not to pass each vertices in an individual mode, saving many resources and time.

In our engine we used glDrawElement(), this draws a sequence of primitives whose order is determined by index array. Those indices correspond to the faces which are ridden in the ovo file, in fact both describe the vertices order.

This function aims to reduce the number of vertices to transfer because OpenGL maintains the vertices previously rendered in a cache instead of passing them ever.

void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid\* indices);

The first parameter is the type of primitive, the second is the number of indices of index array, the third is data type of index array and the last parameter is the address of index array.

In our case the method has this format:

glDrawElements(GL\_TRIANGLES, indicesSize, GL\_UNSIGNED\_SHORT, indices);

In addiction of this, we have to enable and disable the arrays which we use and other methods that specify the exact position of those last in memory.

* **glVertexPointer():** specify pointer to vertex coordinates array
* glNormalPointer(): specify pointer to normal array
* glTexCoordPointer(): specify pointer to texture cords array

Because of this solution we could take a more complicated helicopter model.

As fields we have a pointer to float for data concerning indices, vertices, normal vector and texture.

Mesh has a field which is a pointer to a material and in rendering phase, after the type is controlled and returns “MESH” (as enum).

# Material

This is similar to light because of the fields for emissive, ambient, diffuse and specular informations.

More has shininess and alpha parameters. Alpha value is modifiable by a method in class Engine:

void setAlphaToMaterial(Node\* root, float alpha, string nodename);

Material has a reference to Texture and when it has to be rendered, it renders also the texture associated because of the state machine of OpenGL. In fact, in render phase, we render before the Material, then the Texture (those if existing) and at the end the relative Mesh.

# Texture

Texture provides a constructor which, passing the name of the texture, will load this last from the folder “../ovo\_files/” by the FreeImage library.

Render method will simply use glBindTexture with the associated id.

The render method works by setting the state machine on the current texture, this means

that every mesh called after the rendering of the texture will have that texture build up.

Mesh, Texture and Material are linked so it was difficult to link them (we know that our approach is not the best one but works…). Best was to call only render for mesh and, if a material and/ or a texture were present, automatically called their specific render methods.

For Texture and Material most of problems were to enable all and the correct settings, this caused much time invested in these classes.

# Light

Light also derives from node and all his data comes from the ovo file. His fields are all information common to the light model like color, direction, position, ambient, diffuse, specular, etc.

It has also a boolean which says if the light is activated or not. From the ovo file comes also an integer that describes the subtype of the light; the following table will describe the difference between the light type:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Subtype | Type | X | Y | Z | w | Cutoff |
| 0 | omnidirectional | 0.0 | 0.0 | 0.0 | 1.0 | 180 |
| 1 | directional/infinite | direction[0] | direction[1] | direction[2] | 0.0 | - |
| 2 | spotlight | direction[0] | direction[1] | direction[2] | 1.0 | 0> <180 |

We decided to manage the light id separately from the others id, so we can associate the light id with the number of light of openGL (GL\_LIGHT#), if the number is more than 7, the method getLightNumber will return the default light (GL\_LIGHT0).

It is also possible to enable or disable a precise light.

Light has a lot of settings to enable in the correct way, helpful is that the ovo reader give all the necessary information to create those last.

# Working Demo

Foto DEMO