2019

Graphics Engine

COMPUTER GRAPHICS 2018/ 2019

Gregorio Meyer, Cem Koca and Julian Petralli

INDEX

Introduction

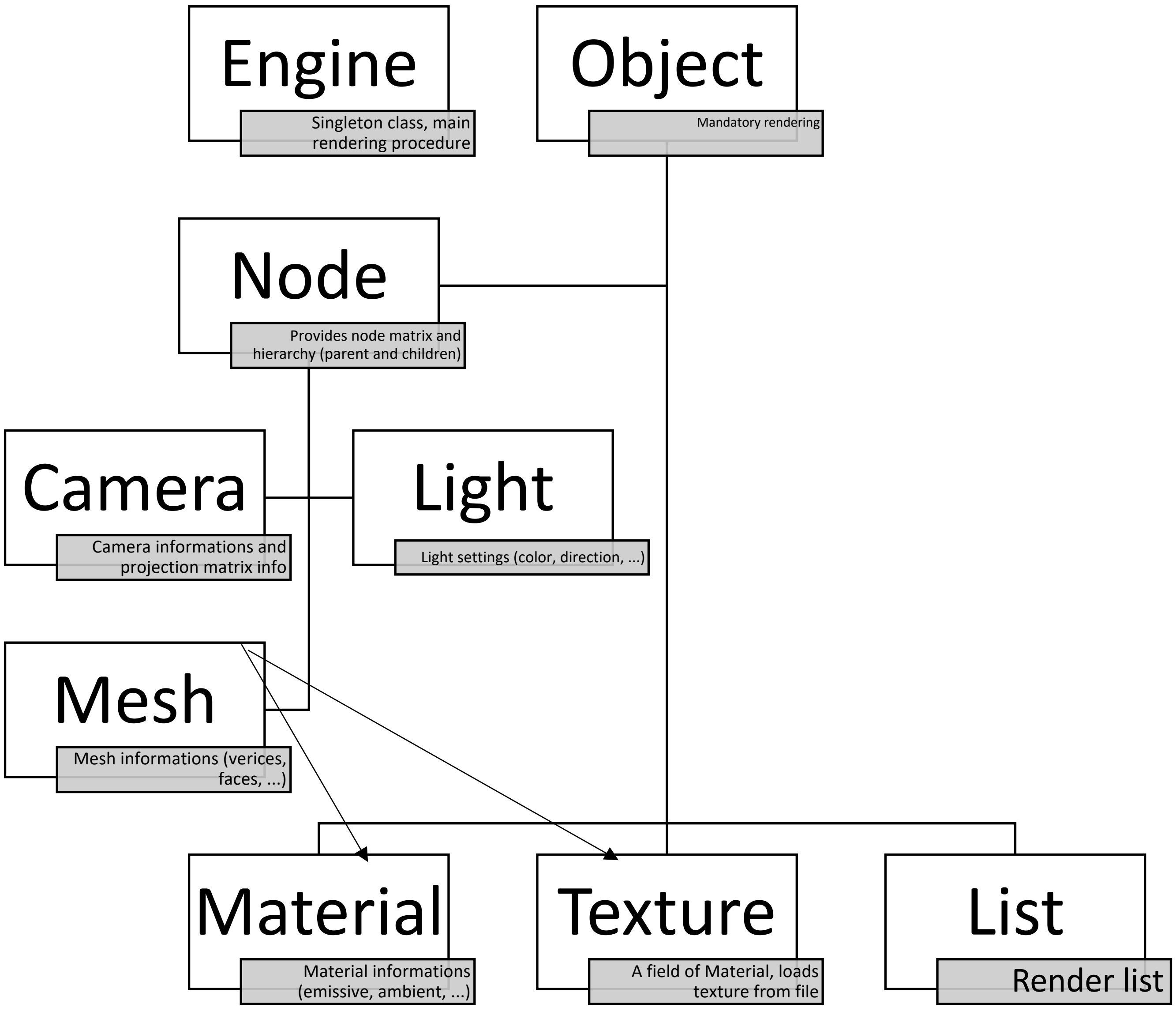
This project aims to develop two distinct components: a 3D graphics engine (as a library) and an animated gauntlet (as an application that uses the 3D graphics engine).
The 3D graphics engine uses some external libraries which aren't visible on the client side. The libraries used are:

- **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:


Engine
Singleton class, main rendering procedure
Object
Mandatory rendering
Node
Provides node matrix and hierarchy (parent and children)
Camera
Camera informations and projection matrix info
Light
Light settings (color, direction, ...)
Mesh
Mesh informations (verices, faces, ...)
Material
Material informations (emissive, ambient, ...)
Texture
A field of Material, loads texture from file
List
Render list

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.

All objects data (except Camera settings) are acquired through the OvoReader class

This class read the scene details from an OVO file and populate a list of Node according to their type. For each type a different sets of properties needs to be saved. For example when reading an object of type Node we save its name, type, children size and matrix.
Some of the features of the OvoReader are not used by our Client App (like the physics properties of the Mesh for example). But since they could be useful in another application we decided to keep them. When reading nodes of type Material we need to associate them with their textures, if present. In a similar way when reading a Mesh we need to assign them their material. In the following pages there is a more detailed description of all classes of our 3D graphic engine.

Engine

The engine class is the main component of the API. It's a single class (singleton) responsible for initializing the OpenGL context and main modules. It contains the main rendering procedure and a series of wrapper functions for FreeGlut callbacks and functions, and for the initialization (or deinitialization) of FreeImage context. It also contains the methods to move our gauntlet and camera. A list of available commands follows:

- 1, 2, 3, 4 turn on / off light 1, 2, 3 or 4
- c change the current camera
- up, down, left, right move the camera in the desired direction
- wheel up and wheel down to moves camera up and down
- I, j, k, l, u, o moves light directions
- R for autorotate and r for rotate
- H for open hand and h for open hand
- Mouse movement up and down change angular movable camera
- Mouse movement right and left change angular fixed camera

Once the client creates context and sets the necessary callbacks he can add some cameras to the scene and specify the background color. After this preliminary phase we can pass the ovo file to our engine through the method getScene. This method will return a pointer to our scene graph. Passing this pointer to the setList method will recursively populate our render list. After preparing the scene we enter the main rendering loop. Here we clear some buffers bits, set the perspective matrix for 3d rendering then render the scene. After that we switch to 2d rendering to render some text containing information about commands and performance, swap buffers and display the scene.
To make it easier to switch between different points of view, we kept a pointer to current camera and a vectors of cameras available.

Camera

Cameras are the only objects that is not read from the OvoReader. Camera settings are specified on the client side. To add a camera the client has to call the following Engine method:

```
Engine::addCamera(std::string name, bool movable, glm::vec3 eye, glm::vec3
center, glm::vec3 up)
```

This will create a new camera add it to the list of cameras and set it has the current one.
It is possible to switch between cameras since the engine class provides a method for this purpose (called with 'c' key). Cameras can either be static or dynamic according to their movable property. If the camera is set as movable it can be moved with arrow key and mouse wheel inside. Walls serves as constraints for the camera movements. Camera contains both a matrix inherited through Node and the projection matrix set with the setProjectionMatrix method.

Object

Object is an abstract class responsible for keeping track of existing objects forcing the render method and setting id and name.
This class is the root of our hierarchy tree, and it's used by all the derived classes.
Incremental id are setted through the setId method by the Node constructor.
Id is a field of Engine class and each time a node is added, we have to increment this last.

Node

The Node class extends the Object class with the required functions to locate the object in the 3D space (through a matrix) and in a hierarchy (through a hierarchical structure).
This matrix is saved world coordinates. Each node has a pointer to his father (except root which father is nullptr) and a vector containing its children (if he isn't a leaf).
The class provides methods for adding a children (this method also sets the parent) or for removing it by passing its index or its pointer. The render method is overwritten but not used.
SetChildrenSize is used by OvoReader and simply reserves memory for the upcoming childrens. The method getCapacity is used during the construction of the scene graph and returns the reserved memory size. In this manner we are sure that getChildrenSize always return the correct value. Node class also provides a method to obtain the final matrix which multiplies the father matrix (if present) for the current node matrix. This comes useful when rendering the list.

List

This class is used to save a list of the elements to render.
Internally it uses a std::vector of Node pointers called list. Through its methods it is possible to add, remove or get one of its list elements. The list is populated by passing the scene graph to the constructor. This will call the getTreeAsList method and set the list.
The render method of list renders all the elements contained in the list. It's possible to render the scene backwards simply by passing a scaling matrix and multiplying it for the first element of the list (the root node)

Mesh

This class derives from Node (which derives from Object). So it implements a render method. This method was, initially, implemented by drawing each triangle with glBegin() and glEnd() using vertices and faces which came from the ovo file. We later found out about a better technique so we decided to use "*vertexArray*"
This method uses arrays to save vertices, but also other information as: vertices position, normal vector, texture coordinates and colour information. With this method you don't have to pass each vertices in an individual mode, saving many resources and time.

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

Mode is the type of primitive, count is the number of indices in the index array, type is the data type of the index array and the last parameter is a pointer to index array.
To draw elements we used glDrawElement(). This method draws a sequence of primitives whose order is determined by the index array. Those indices correspond to the faces which we get from the ovo file (both describe vertices order). This function reduce the number of vertices to transfer by OpenGL maintaining a cache of the vertices previously rendered instead of passing them every time.
In our case the method has this format:
The Mesh class also contains data concerning indices, vertices, normal vector and texture stored in a pointer to float. Mesh also contains a pointer to its material. The material render method is called directly by the Mesh render method.

Material

This class represent a material used by our Meshes. It contains fields to store its emissive, ambient, diffuse, specular, shininess and alpha values. If alpha is not one the corresponding material will be transparent. The alpha value of materials can be modified by the Engine class method setAlphaToMaterial. Material also stores information about its texture, similarly to what done with the Mesh class when the render method is called its Texture render method is also called.

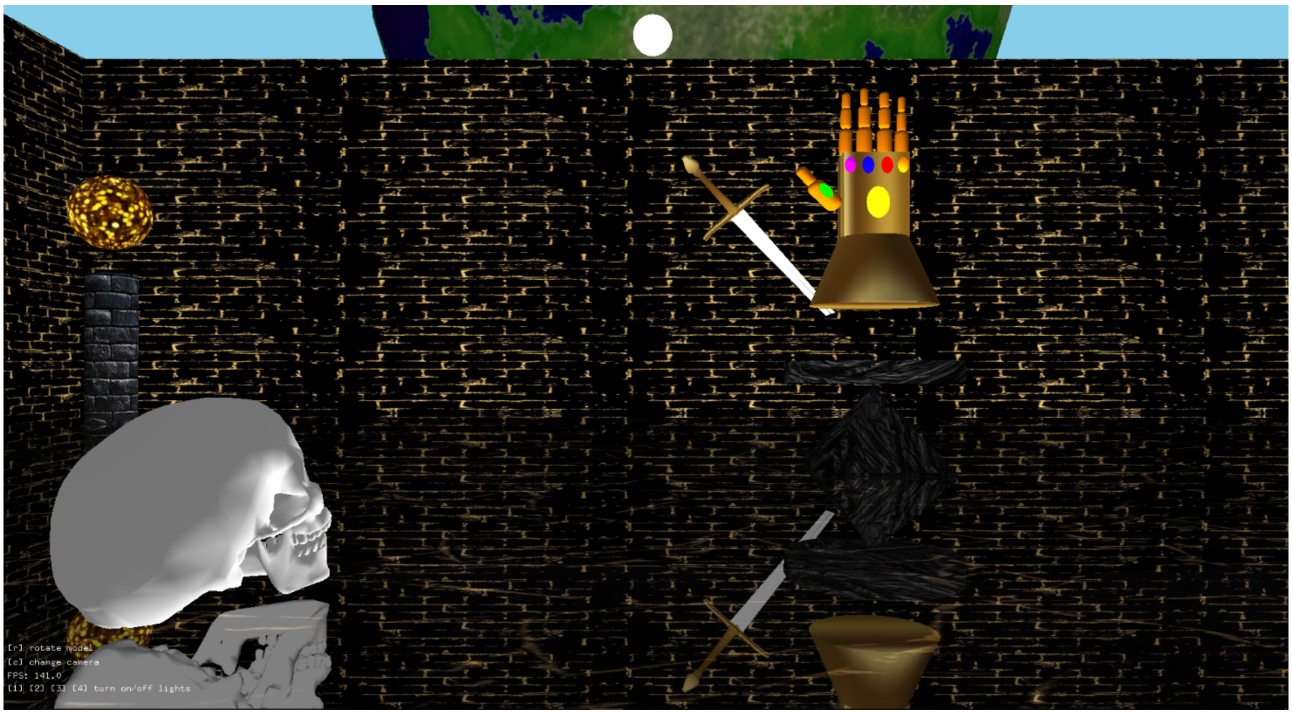
Texture

Texture are generated by passing the correct texture name to the Texture constructor. If the name is "[none]" the texture creation is cancelled. Texture are searched in the resources folder so the "../resources/" path is automatically prepended. Render method will simply use glBindTexture with the associated Texture id. For enabling the anisotropic filtering we use some quick defines as seen in class. The texture destructor is responsible for deleting the corresponding texture through the glBindTexture method.

Light

The Light class stores various information like ambient, diffuse, specular, color, direction and many others. All this settings are setted by the OvoReader class. The Engine class provides a method for turning on or off a light according to his name (though buttons 1, 2, 3, 4) To do this Light class contains a boolean value which indicates the state of the light. Lights number are managed separately by id so that it's easier to maintain compatibility with openGL enums (GL_LIGHT#)

Working Demo


[r] rotate model
[c] change camera
FPS: 141.0
[1] [2] [3] [4] turn on/off lights