glEngine : Technical Specifications

Gaël Jochaud du Plessix
Loick Michard

2012

Contents

Chapter 1

Introduction

This document describes the technical specifications for the glEngine project. It comes with the Specifications document that explains the functional specifications. The technical specifications document describes the retained solutions for the glEngine project. These specifications are divided in two parts: the description of the scene management and how the 3D rendering works.

Chapter 2

Scene management

In order to render a virtual world in 3D, the engine must manage a lot of data describing a scene. This part of the specifications describes the different elements that can be part of a scene and how they are arranged hierarchically by the engine.

2.1 Scene elements

A scene is composed of various kinds of elements that describe all the aspects of the virtual world to render. All these elements inherit the class gle::Scene::Node that describe the properties of a scene element:

- A position (gle::Vector3f)
- A rotation matrix (gle::Matrix4f)
- A scale matrix (gle::Matrix4f)
- A name (std::string)
- A parend node (gle::Scene::Node*)
- A list of child nodes (sd::vector<gle::Scene::Node*>)

Scene elements are hierarchically linked with each other with their parent and children properties. The children are stored in an std::vector, because the scene graph is not supposed to change often but must be accessed efficiently.
In addition, nodes have a isDynamic property that indicates wether or not they are supposed to change after being added to the scene. (Cf. section 2.2)

2.1.1 Materials

Materials allow to specify how an object visually appear when rendered. It contains the following information:

- Ambient color (gle::Colorf)
- Diffuse color (gle::Colorf)
- Diffuse intensity (GLfloat)
- Specular color (gle::Colorf)

- Specular intensity (GLfloat)
- Shininess (GLfloat)
- Color map (gle::Texture)
- Normal map (gle::Texture)
- Environment map (gle::EnvironmentMap)
- Reflection intensity (GLfloat)

A material also have a name, in order to identify it.
All these informations are stored in a uniform buffer, in order to be efficiently used by the GPU.

Creation

The engine allows the creation of materials programmatically via the gle::Material class. All the properties of a material can be accessed and set via accessors.

Import

Materials can be imported from .mtl files through the .obj importer (Cf. section 1.2.2). The universal loader (gle::UniversalLoader) can also import materials.

2.1.2 Models

Models are the virtual representation of 3D objects. They inherit all the properties of a scene element and are characterized by:

- A geometry (vertex coords, normals, tangents, etc...)
- A rasterization mode (Fill, Line, Point)
- A material (gle::Material)

Creation

The user can create a model programmatically. Thus, glEngine provides the gle::Mesh class.
To specify the mesh geometry, there are several accessors that store their data directly on the hardware thanks to the Buffer Manager:

- gle::Mesh::setVertexes
- gle::Mesh::setNormals
- gle::Mesh::setTangents
- gle::Mesh::setTextureCoords
- gle::Mesh::setIndexes

All these data can be provided in two ways: simple (GLfloat|GLuint)* arrays or gle::Array<(GLfloat|GLuint)> arrays.

Import

The engine allows the import of a mesh from a Wavefront OBJ file, thanks to the gle::ObjLoader class. Its function gle::ObjLoader::load returns a mesh imported from a .obj file. If the file includes .mtl files describing the model materials, they will also be imported (Cf. Section 1.1). In addition, glEngine provides an other way to load assets, the gle::UniversalLoader class which is more usefull because it uses the ASSIMP loader to import scene elements from a lot of file formats. These two classes inherit the gle::FileLoader interface, so they can be used the same way.

Update

Models can be updated at any time, while their isDynamic property is set to true. To set new values on a gle::Mesh, the user can simply use the corresponding accessors functions.

2.1.3 Lights

Lights describe how a scene is illuminated. There are three types of lights:

- Point light: omnidirectional light characterized by a position
- Directional light: directional light characterized by a direction vector
- Spot light: directional light characterized by a position, a direction vector and a cut off angle

All light classes inherit the gle::Light class.

Creation

Lights can be created programmatically by the user through the classes gle::PointLight, gle::DirectionalLight and gle::SpotLight, specifying all their properties manually.

Update

It is possible to update the properties of dynamic lights at any time.

2.1.4 Cameras

Cameras permit to specify the point of view of the observer in a scene. They have several properties common to other scene elements such as a position and an orientation (specified by a target). A camera also have specific informations: a field o view, an aspect and the distances to near and far plane.
glEngine provides two kinds of camera:

- Perspective Camera: A camera that draw far objects smaller than near objects, as the eye see the world
- Orthographic Camera: A camera that applies a parallel projection, where all objects have the same size.

Creation

Cameras are created programmatically by the user via the classes gle::PerspectiveCamera and gle::OrthographicCamera that inherit gle::Camera, specifying all properties at object construction.

Update

It is possible to update the properties of a dynamic camera at any time, through accessors methods provided by gle::Camera and it child classes.

2.2 Structural information

Scene elements are structured hierarchically and have particular meta-data in order to improve the rendering process.
These data include information about whether the element is static or dynamic, its position in the scene graph and a representation of the world based on spatial partitioning.

2.2.1 Elements hierarchy

Each element of the scene is represented as the node of a graph, in order to hierarchy them. Thus, each element have a parent and a list of child elements.
In addition, it is possible to name the elements and retrieve them easily within the scene graph, thanks to the functions gle::Mesh::getChildren and gle::Mesh::getChildrenByName.

2.2.2 Static vs Dynamic

In the scene, there are two main types of elements: static ones and dynamic ones.
As their name suggest, static elements are not subject to change with time. Thereby, the renderer can improve its performance when processing these elements.
Dynamic elements may rather be updated very often, so the engine provides ability to efficiently update the data of these elements, through the gle::Scene::Node accessors functions.

2.2.3 Spatial partitioning

In order to improve the rendering performance, the scene have a representation of its elements based on spatial partitioning. Thanks to that representation, the engine is able to efficiently treat a lot of elements.

Octree

The most effective spacial scene representation for a 3D environment is octree. When static meshes are added or removed from the scene, a spacial octree representation is generated. This generation uses maximum number of threads for increase performance. For this, each node division in 3D are performed using a pool of threads.

Bounding Volume

In order to check the intersection between a mesh and an octree node, all meshes have a corresponding bounding volume. This bounding volume is a simplify space representation of the mesh. It enables to build a complete tree really faster. There are two kind of bounding volume available:

- Bounding box
- Bounding sphere

Chapter 3

3D Rendering

This part of the technical specifications describes how the 3D rendering work.
It lists the features of the renderer and describes the techniques retained for performance improvements.

3.1 Features

This section describes the rendering features that are supported by the glEngine renderer.

- Rasterization of triangles, lines and points using the OpenGL API's
- Texturing
- Diffuse shading
- Phong shading
- Reflection in an environment map
- Drawing of an environment map
- Bump mapping
- Fog
- Shadow mapping
- Rendering to texture

3.2 Performance

In order to provide an efficient rendering process, several techniques are used by the glEngine to improve its performance capabilities.

3.2.1 Frustum culling

Frustum culling is a technique that allows to draw only visible objects.

3.2.2 Reduction of draw calls

When rendering a 3D scene, the engine must take the most advantage of the GPU as posssible. Thus, it may reduce the number of draw calls to limit data transfers between the CPU and the GPU.
To do that, glEngine use several techniques and process scene data in order to draw the maximum meshes in one draw call.

Buffer manager

The first step to draw multiple meshes at once is to put their vertex attributes in the same OpenGL buffer. Thus, glEngine uses a buffer manager to manage the data of all the meshes.
This buffer manager is a simple implementation of a memory pool, with a list of nodes and a list of free nodes. It can be used in the engine through the gle::BufferManager class.

Meshes factorization

Next step is to know which meshes can be drawn with each others. This operation is performed by the function gle::Mesh::factorizeForDrawing(). This one uses gle::Mesh::canBeRenderedWith() to compare the meshes.
Two meshes can be rendered together if they have the same "rasterization mode" and if their materials are compatible (if they use the same textures). Thanks to that, a scene that use a texture atlas can be rendered with a minimum number of draw calls.

Uniform buffers

Once meshes are factorized in groups, the engine must build uniform buffers in order to store the per-mesh datas (transformation matrix, skeleton id, etc...). This is done by the gle::Scene::updateStaticMeshes() function, which creates as many uniform buffers as necessary. If a mesh group doesn't fit into a single buffer (according to GL MAX UNIFORM BLOCK SIZE), several buffers are created and each mesh is assigned to a buffer.

Rendering algorithm

When everything is ready to be rendered (vertex attributes are in the BufferManager, uniforms are divided into several buffers, meshes are factorized), the renderer build, for each mesh group, the buffer of indexes to render. This is done by copying directly from the GPU to the GPU the indexes previously stored in OpenGL buffer objects.
Then, the uniform buffers and textures corresponding to the mesh group are binded and the draw call is performed, through the glDrawElements function.