# Designing the Architecture of a Rendering Engine

**Răzvan Tănasie***

**University of Craiova, Faculty of Automation, Computers and Electronics, Romania (e-mail:tansie_razvan@software.ucv.ro).*

**Abstract**: Sometimes developers do not consider feasible the purchase of a pre-made rendering engine for creating a computer game so they prefer to create their own engine. Moreover, if the engine might be used for several games or applications that may run on different operating systems, it is desirable to be reusable and to have platform-independent functionality. This paper focuses on presenting the theoretical aspects involved by the construction of a portable and reusable rendering engine and on how they can be applied to design the architecture of the system. Although the author only intends to offer an overview of the subject, a prototype for a rendering engine is used for exemplification purposes. It has been implemented using the C++ programming language and the OpenGL API (Application Programming Interface), which confirm the claims of portability.

*Keywords*: Computer graphics, Real-time rendering, Rendering engine, OpenGL.

## 1. INTRODUCTION

Rendering is used in the area of computer graphics to describe the process of drawing three dimensional models on a two dimensional surface represented by the screen.

A rendering engine is a piece of software which takes as input meaningful rendering data associated to a group of objects and "outputs" the corresponding digital image on the monitor. The information that the engine needs consists in: objects geometry, camera position, textures, lighting and shading.

The transformations that must be carried on the initial objects until they can appear as simple pixels on the screen involve a considerable amount of processing which is supported by both CPU and GPU. As a result, one of the most challenging aspects of creating a rendering engine is to use as little GPU and CPU resources (memory, number of operations) as possible without affecting the quality of the rendered scene. That is, the viewer must perceive it as a real-world scene.

There are two types of rendering: pre-rendering and real-time rendering. Pre-rendering refers to a computationally intensive process that is typically used for movie creation or digital image processing (Marsh D. et al., 2006) while real-time rendering is often used for 3D video games which rely on the use of graphics cards with 3D hardware accelerators.

In what follows there will be presented the aspects involved by the creation of a real-time rendering engine. Thus, Section 2 discusses the graphics rendering pipeline by presenting an outline of each of its three stages. Section 3 illustrates the architecture of the proposed engine which aims at a clear separation between the pipeline stages by using object oriented design patterns while Section 4 presents the conclusions of the current research.

## 2. THE GRAPHICS RENDERING PIPELINE

The graphics rendering pipeline is considered to be the core component of real-time graphics because its main function is to generate a two dimensional image given a virtual camera, three dimensional objects, light sources, shading equations, textures and more (Akenine-Moller T. et al., 2008).

A pipeline consists of several stages (Hennessy et al., 2012). For example, in a factory that produces integrated circuits, a single circuit cannot go on for encapsulation in a plastic casing until all the circuits before it , handled by the same robot, have been encased.

The three stages of the graphics pipeline are: the application stage, the geometry stage and the rasterizer stage.

### *2.1 The Application Stage*

The application stage is entirely executed on the CPU and its result consists in a set of rendering primitives such as points, lines and triangles that represents the input for the next stage.

Unlike other stages, the present one cannot be decomposed into substages because it relies completely on a software program. Nevertheless, the performance can be increased by running it on several processor cores.

The main problems handled at this stage are: collision detection, texture animation, animations via transforms, user input handling.

### *2.2 The Geometry Stage*

The Geometry Stage is responsible for most of the perpolygon and per-vertex operations and is divided into the following functional stages: model and view transform, vertex shading, projection, clipping and screen mapping.

### *2.3 The Rasterizer Stage*

The Rasterizer Stage computes and sets the colours for the pixels that cover the objects in a scene. It takes as input the transformed and projected vertices with their associated shading data from the previous stage. Its functional substages are: triangle setup, triangle traversal, pixel shading and merging.

## 3. ARCHITECTURE

The prototype has a simple architecture based mostly on the object oriented design patterns of composition and inheritance and it implements the model-view-controller (MVC) concept (Gamma E., 1994). It was also influenced by the examples given in (Benstead L. et al., 2009) which helped integrate the OpenGL rendering pipeline.

It must be specified that this is only a draft of a future engine which aims to demonstrate that it is flexible and solid enough to serve as a framework for the development of 3D games.

Fig. 1 illustrates the classes that implement a part of the functionality of the engine that is, the creation and the rendering of simple objects that are also affected by light sources and that can perform simple actions such as: translation, scaling and rotation (this applies only to the instances of the DynamicObject class).

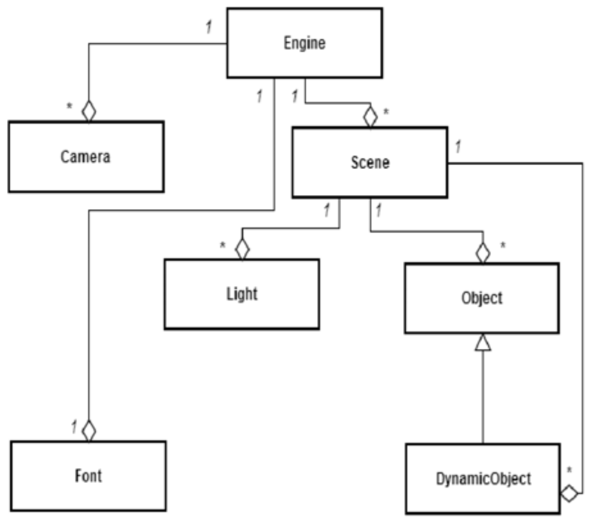

Fig. 1. UML class diagram for the main components of the engine

Next, the main characteristics of each class and how the rendering process is performed will be briefly described.

### *3.1 The Engine Class*

The Engine Class is responsible with the creation and destruction of objects, their initialization, their update based on user input and their display on screen. Therefore, it plays the role of the controller in the MVC architecture.

Firstly, in the initialization phase, the OpenGL context is set up by enabling culling and depth tests (Foley J., 1995).

Culling is a technique used in rendering for optimization purposes and it consists in eliminating the vertices that are not visible to the viewer from the group of vertices to be drawn on screen. There are several techniques used for culling. That implemented by OpenGL is called "back-face culling". By setting the GL_CULL_FACE state for the OpenGL context, the closed surfaces from the scene will not have all their polygons drawn but only those that face the camera. Thus, the number of rendered polygons decreases to half.

The depth test is associated to a depth buffer which contains a value for each pixel. This value represents the distance of each pixel from the eye and it is used to determine the order of drawing. The pixels with larger depth-buffer values are overwritten by pixels with smaller values.

After setting up the rendering context the scenes and their corresponding objects are created and initialized. There can also be created one or several cameras that allow the user switch between scenes or watch the same scene from different angles.

Handling user commands such as: key pressing, mouse clicks, mouse movement, wheel rotation and so on is also one of the most important aspects of an interactive application. Thus, the Engine class contains methods (functions) that enable: switching between cameras and scenes (the class contains the indices of the current active camera and scene objects), camera rotation (around OY and OX) and camera translation.

If the framework is used for creating a 3D game, the class can be easily extended to keep one or more references to the player(s) and to implement the needed functions for controlling player physics.

### *3.2 The Camera Class*

A camera in a virtual 3D world is actually the point and angle from which the user perceives the rest of the objects. As a result, the controller only needs to have access to the position of the camera, a 3D vector, and to its three angles of rotation: yaw (rotation around OY), pitch (rotation around OX) and roll (rotation around OZ).

Moving or rotating a camera object is one of the most interesting aspects in rendering because they actually involve the translation and rotation of all the objects in the scene in the opposite direction. For instance, if one wants to rotate the camera 30 degrees to the left, all the objects are rotated 30 degrees to the right. Also, if it should be moved 22 units forward, all the objects will move 22 units backward. This is necessary because rotation, translation and scaling are implemented via transformation matrices applied on the vertices of an object. Since the camera is not a physical object to be perceived but it affects the way the others are perceived, it only modifies those matrices used for rendering a scene.

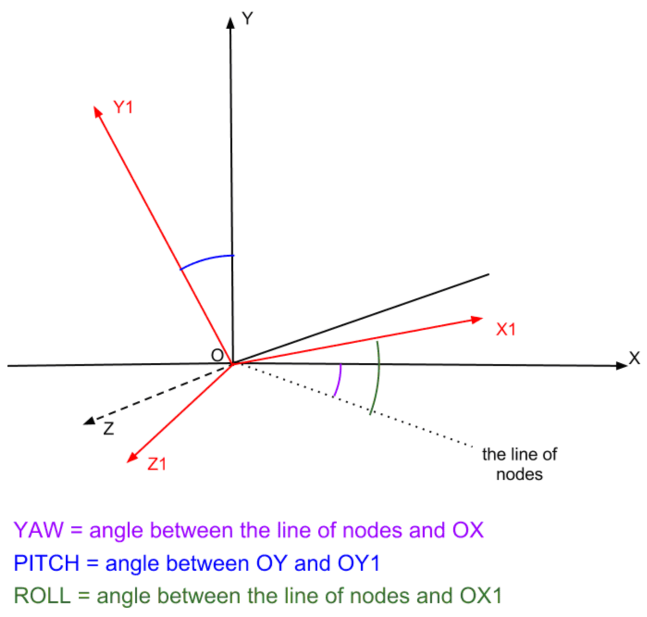
Y
Y1
X1
X
O
Z
Z1
the line of nodes
YAW = angle between the line of nodes and OX
PITCH = angle between OY and OY1
ROLL = angle between the line of nodes and OX1

Fig. 2. The angles for general rotation

Rotation around OZ is rarely used. As a result, the roll angle was neglected which implies that the position of the camera is determined based on the other two angles. For a camera that moves in the XOZ plane the following equations determine its position based on the varying yaw and pitch values:

$$\begin{aligned} \vec{p}_{t+1} &= \vec{p}_t + dz * \vec{f} + dx * \vec{r} \\ \vec{f} &= \left[-\sin(yaw) * \cos(pitch), \sin(pitch), \cos(yaw) * \cos(pitch)\right] \\ \vec{r} &= \left[\cos(yaw), 0, \sin(yaw)\right] \end{aligned} \quad (1)$$

Where*:*

$\vec{p}_t$ = current position of the camera

$\vec{p}_{t+1}$ = next position of the camera

$\vec{f}$ = front vector

$\vec{r}$ = right vector

*dx* = translation on OX

dz = translation on OZ

The *dx*, *dz, yaw* and *pitch* values are set according to user input.

*3.3 The Scene Class*

The Scene Class contains an array of light objects, an array of static objects, instances of the class Object, and another array of dynamic objects, instances of the DynamicObject class. The difference between these two object types is that the latter can change position and orientation based on some speed variables whose values may change in time.

The class also keeps a reference to the view frustum which is no more than a collection of six planes that define the view volume. This variable is used in order to introduce another culling technique known as “frustum culling”. Thus, before going on to the geometry phase, all the objects in the scene are firstly checked to be partially or totally inside the view volume. If they are exclusively outside, then they will not appear on screen so it is not feasible to apply all the vertex and pixel transformations on them.

This verification is performed using only CPU resources during the application stage and its success depends on the frustum being updated at each frame in order to be consistent with the position of the camera.

The Scene class acts as an organizer for all the entities the user sees or are about to see and it conveys to them the messages sent by the engine. Thereafter, it represents the model of the MVC architecture along with its building components.

*3.4 The Light Class*

In computer graphics, light can be classified into four types: ambient (light reflected off many surfaces so that its source cannot be perceived), diffuse (light from a certain source that reflects equally in all directions), specular (comes from a specific source and reflects in only one direction) and emissive (which is emitted by an object).

When combined, all these types result in different light effects depending on how more powerful is one type from another. The translation to computer language means that the data encapsulated by the Light class consists in: the position of the light in the rendered world and the colours of its ambient, specular and diffuse components. The emissive component is more related to materials than to light sources so it has been neglected.

The lighting model of the engine is the one proposed in (Benstead L. et *all* , 2009) . Here GLSL shaders are used to compute the colour of each vertex based on its initial colour and the light sources that act on it. Then, the vertex colours are interpolated across the surfaces of the polygons to create realism.

A light is also supposed to illuminate objects less intensely if they are farther from the source so the Light class also contains variables (C-constant attenuation, L-linear attenuation, Q-quadratic attenuation) that serve to creating this attenuation effect by multiplying the diffuse, specular, and source-specific ambient light colours to the attenuation factor. The latter is obtained from the relation:

$$\frac{1}{C + L * d + Q * d^2} \quad (2)$$

where *d* represents the distance of the vertex from the source of light.

All the data encapsulated by this class along with the information from the Object class related to the material properties and vertices positions and colours form the input for the vertex shaders used by OpenGL.

*3.5 The Object and the DynamicObject Classes*

As previously specified, the DynamicObject class only adds new members to the Object class for quantifying the rotation or the translation speed of one of its instances. Therefore, only the properties and behaviour of the base class will be described into more detail.

The Object class is one of the most complex structures as illustrated in Fig. 3 because it encapsulates all the necessary data for rendering, i.e.: position of the entity, yaw, pitch and roll angles, radius of the bounding sphere used for culling, vertex, index, texture coordinates and normals buffers and arrays used by OpenGL, a reference to the current shader and another reference to the current skin of the object.

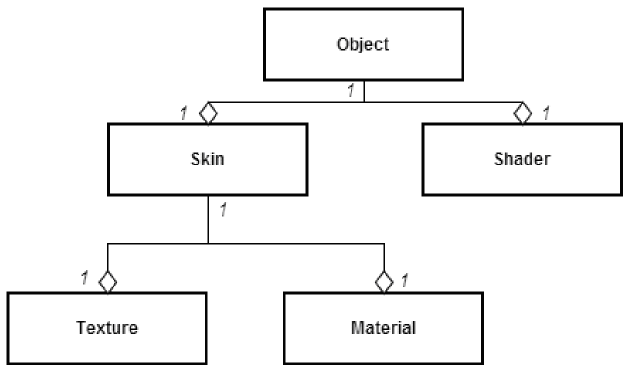

Fig. 3. UML class diagram for the Object class and its members

The class has been designed such as the shader and the texture (or skin) of an object can be changed at runtime based on the conditions of the game. The Texture class provides the access means to the .tga file that stores the image which will be mapped on an object based on its texture coordinates. The Material class contains the properties of the texture that describe how a light beam is reflected. These properties are: diffuse colour, ambient colour, specular colour, emissive colour and shininess.

Taking into consideration all the above information, the evolution of the rendering process for an object can be presented.

Firstly, the shader is enabled. Different objects may use different shaders so it must be ensured that OpenGL uses to correct shader each time. The same operation is performed for textures but in this case the literature uses the term "bind" to specify to OpenGL what texture it should use otherwise it will use the last one that was bound.

Next, the OpenGL rendering pipeline receives all the necessary information for each vertex: position, colour, texture coordinates, normals and then the identifier of the index buffer which gives the order in which vertices should be traversed is specified.

Further on, the vertices undergo several transformations before rasterization such as: modelview transform, projection, clipping and viewport transforms whose output is a set of triangles that fits in the viewport and hence, that the user will see. These transformations are performed by the vertex shader, which can also compute an intermediary colour of a vertex based on its initial colour, material properties and light sources.

The last stage, the rasterization stage, is performed with the help of the pixel shader which computes the final colour of each pixel based on texture coordinates and on the previous colour value from the vertex shader.

## *3.6. The Font Class*

Any graphics application uses labels for describing objects and phrases to create a user-friendly experience. Therefore, a rendering engine must offer the possibility of displaying text of different fonts, sizes and colours on the screen.

In order to make this possible, the Font class has been created (Fig. 4) and it uses a texture of Targa format (.tga) to select the type of the font. Other significant attributes of the class are the size, a real number, the colour, a composite data type consisting of four real numbers for the red, green, blue and alpha components and a shader program which is the same for all fonts since there are no differences in their rendering process.

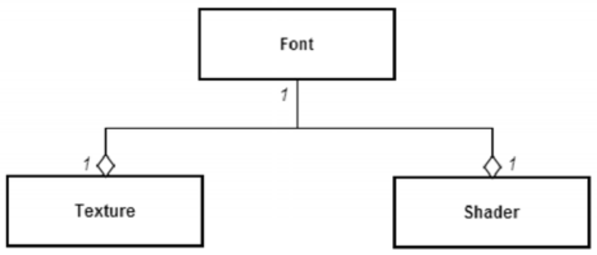

Fig. 4. UML class diagram for the Font class

Because the Engine class is the controller for this architecture and because, generally, texts appear on screen based on user input, it is the one that keeps a reference to the Font object in use.

To create such an object, the programmer only needs to specify the type of the font which is equivalent to the name of the .tga file, the size, the colour and the width and height of the viewport. The last two parameters are necessary to set the orthographic projection during font rendering thus making the texts appear two-dimensional.

To use a Font instance, the programmer must follow two rules: call the drawing routine for a certain text string and for its corresponding x and y coordinates after all the other objects in the scene have been rendered and reset the width and height used by the orthographic projection each time the window is resized. The first rule ensures that the text is actually visible because if it were rendered before everything else, then the objects in the scene would cover it. The second rule preserves the ratio between text size, objects size and window size as the window becomes larger or smaller.

The rendering process is similar to the one for objects the difference consisting in the fact that, for each letter in the string, there must be computed eight texture coordinates which correspond to the four vertices of the quadrilateral that contains the letter.

## 4. CONCLUSIONS

In this paper, the author has tried to illustrate how a rendering engine works and which are the theoretical aspects involved. For exemplification, it has been used a prototype developed using the C++ programming language and the OpenGL API which ensures a higher degree of portability on different platforms (Windows, Linux, Mac OS and even on mobile using OpenGL ES).

The aims for future work consist in extending the engine to support a solid animations system using 3D models exported from a modelling application and to provide implementations for several effects such as fog, smoke, water. Also incorporating optimization techniques to improve performance would be a great step forward, frustum culling being only the first one.

ACKNOWLEDGMENT

This work was supported by the strategic grant POSDRU/89/1.5/S/61968, Project ID61968 (2009), co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007-2013.

## REFERENCES

Akenine-Moller, T., Haines, E. and Hoffman, N. (2008). *Real-time rendering.* A K Peters, Wellesley, Massachusetts.

Benstead, L., Astle, D., and Hawkins, K. (2009). *Beginning OpenGL game programming (second edition)*. Course Technology, a part of Cengage Learning, Boston, Massachusetts.

Marsh, D., Ricard, D., White, S., and Xu, J. (2006). *Performing a pre-rendering pass in digital image processing*. Patent Application Publication.

Hennessy, J. and Patterson, D. (2012). *Computer architecture – a quantitative approach. 5th Edition*. Morgan Kaufmann. Waltham. Massachusetts.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series.

Foley, J., van Dam, A., Feiner, S., and Hughes, J. (1995). *Computer Graphics: Principles and Practice in C. 2nd Edition.* The Systems Programming Series.