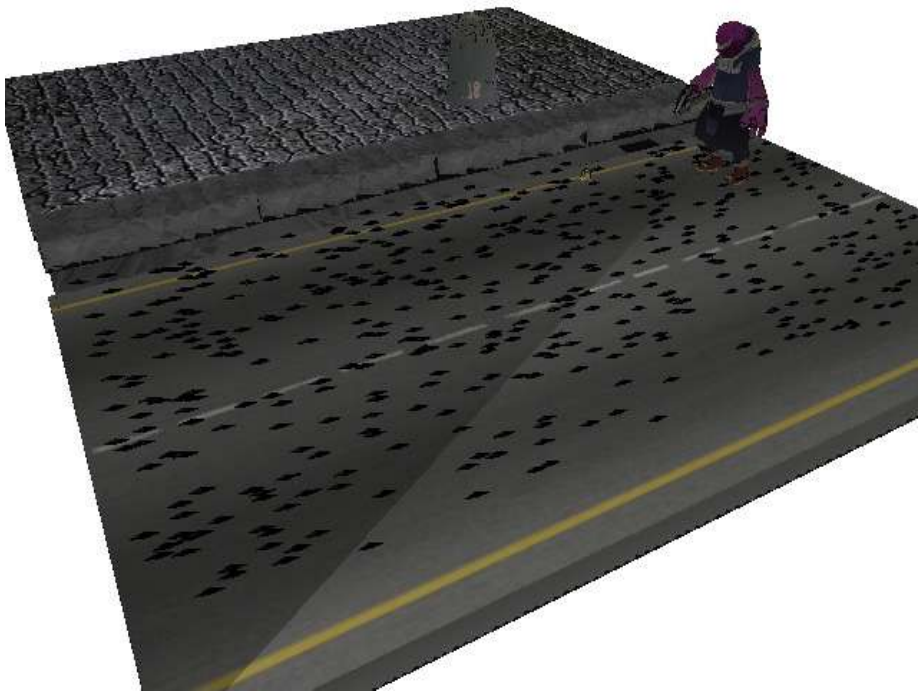
Scriptable Scene-Graph based OpenGL Rendering Engine



Masters Thesis

Colin Wheatley

Msc Computer Animation
N.C.C.A. Bournemouth University

11 September 2005

Contents

1 Introduction

1.1 Need for performance optimisations in real time/interactive applications

Unlike film and production animation, interactive applications such as video games and visual simulations are dictated by the real time performance of CPU's and graphics processors. In the early days of gaming and visual simulation, titles such as Pong and Space Invaders amazed audiences with their simple but effective graphics. Early computer generated flight simulators also demonstrated the possibilities of interactive environments to the world, albeit in a crude manner. The importance of realistic graphics and worlds weren't so important at the time. As the games and visual simulation sectors progressed however, vast improvements were made in the graphical performance of hardware allowing for bigger and more detailed environments, especially in the field of video games. Now as each new game was released to the market, game players expected more and more in every area, especially in the graphics department. As theatre audiences around the world enjoy full scale wars between possibly thousands of creatures in glorious detail in films such as the Lord of the Rings trilogy, games players are seldom allowed the same pleasure when confronted with real time generated battles found in games such as the Civilization series and Battle for Middle Earth.

Fig 1.1 Screenshot from the PC game Battle for Middle Earth [17]

In interactive applications, compromises must often be made between the realism of the world and the amount of objects in any scene. This makes sense when we think of the different approaches that are found in interactive simulations and production rendered animation. In production animation, rendering is an off-line process where each frame could take anywhere from minutes to hours to produce, where as in video games and simulations we are expected to produce anywhere from 25 to 60 frames a second. As technology continues to advance this compromise is becoming less and less of a problem for video games and interactive simulations alike, but for the time being developers must still find ways to optimise their applications to get the most out of the current technology available.

The aim of this project is to design and implement a performance driven, scene based rendering engine that can be utilised in the development of video games and interactive visual simulations, which can also be extended by the end user. There are many optimisations that can be applied to improve rendering times and our goal is to implement a set of key features, which will complement the design and performance of the application. Most rendering engines include some form of culling and state sorting as a standard, as well as features such as level of detail nodes and separate render passes. The organisation of objects in a scene can also be vitally important, a popular design approach for scene based rendering engines found in modern video games and simulations is through the use of a scene-graph, an approach that we will be implementing in the development of our system.

1.2 What is a Scene-Graph?

"The scene-graph is an object-oriented structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene." [18]

Scene-graphs are organised as a collection of nodes in a graph or tree structure, were nodes represent different objects and relationships in the scene. Nodes generally have a single parent and zero or multiple children, an operation applied to a node propagates the effect to its children. Most commonly a node can either be an internal node or a leaf node, where a internal node refers to some logical relationship or procedure and leaf nodes define the actual geometry of the object or more generally an object that can be viewed in a scene [10].

1.3 How can scene-graphs aid in interactive rendering?

Scene-graphs provide a logical and intuitive means of organising objects in a scene or world, establishing a good basis for transformation and object hierarchies. This can greatly aid our optimisation techniques and prepare the objects in the scene for advanced culling procedures such as Octree culling and occlusion culling techniques such as Hierarchical Z-Buffering [13]. These processes can improve the rendering performance of applications using graphics libraries such as OpenGL or Direct3D, which although based on low level commands for drawing to the screen contain no specific routines or processes to improve rendering performance on the client side. Scene graph based toolkits such as SGI's Performer [20] and the OpenSceneGraph [8] project provide a framework and library which make it easier for developers to produce 3D applications, allowing the developer to concentrate more on the creative aspect and not so much on underlying details such as culling and rendering optimisations.

2 Previous and Current Work

Many scene-graph based rendering engines exist in the computer animation sector, including commercial, open source, application specific and generic toolkits. Scene graphs are commonly found in vector-based graphics editing applications including programs such as AutoCAD, Adobe Illustrator and CorelDraw[?], but have also found great acclaim in visual simulations and many modern video games. Scene-graph implementations can also be found in modelling packages such as Alias Maya, in the form of the Hypergraph. Current scene graph based toolkits include SGI's Performer [20], the OpenSceneGraph project [8], opensg and more recently ogre3d [9], each supplying developers intuitive means for production of 3d applications.

After being one of the earliest on the market, SGI's Performer, built atop of the OpenGL graphics library, has now established itself as one of the industry standards for scene graph based development toolkits. SGI's Inventor existed at roughly the same time as Performer, but took a different approach to the performance driven Performer, ranking re usability as it's main priority, resulting in a system that has found more use in academic research or prototyping then high performance applications [7].

Fig 2.1 Visual Simulations implemented with the OpenSceneGraph toolkit [8]

More recently there has been a lot of research and input into the OpenSceneGraph project. An open-source high performance 3d toolkit, providing an object oriented framework on top of OpenGL freeing the developer from implementing and optimising low-level graphics calls. The toolkit contains some of the following key features, multiple culling techniques, state sorting, level of detail nodes, vertex arrays, display lists and as of this document supports 45 different model formats such as Alias (.obj), DirectX (.x) and 3DstudioMax (.3ds). Although the project is entirely written in standard C++, there is a large community of developers and users providing Java and Python bindings. The OpenSceneGraph has been used to develop many applications including numerous visual simulations and interactive environments.

Another open source toolkit that has become popular in recent years, especially in the video game sector, is Ogre3d. A scene-oriented, flexible 3D library written in C++ designed to make it easier and more intuitive for developers to produce applications utilising hardware-accelerated 3D graphics. The abstracted library features support for OpenGL and Direct3D, state management, spatial culling, opaque and transparent render passes, flexible mesh support, material/shader support, special effects and many other features. Ogre3D has been used in the development of games such as Pacific Storm and Ankh.

Fig 2.2 Screenshot from Pacific Storm in development and implementing Ogre3D [19]

3 Technical Background

3.1 Identifying the main aspects of a real time rendering engine

Although much of the work discussed previously documented generic libraries and toolkits to aid in computer generated applications, scene-graphs and scene based systems exist in many computer graphics applications which are designed specifically for that project. It is true that a generic toolkit may not meet the needs of all projects, requiring some aspects to be tailored to satisfy the needs of that specific application. It can also be seen however that most scene based systems possess many of the same key elements. These elements can be seen as the main data structures and processes that make up generic toolkits such as Ogre3d and the OpenSceneGraph project. The most notable are as follows;

3.2 Graph/Tree Based Data Structure

The core data structure of any scene-graph. Many different data structures have been implemented as scene-graphs including n-ary trees and directed acyclic graphs, all however are based on the idea of a graph or tree representing the spatial or logical hierarchy of objects in a scene. The most common is an n-ary tree, consisting of single or shared parents, possessing zero or more children, with at most n children at each inner node [10].

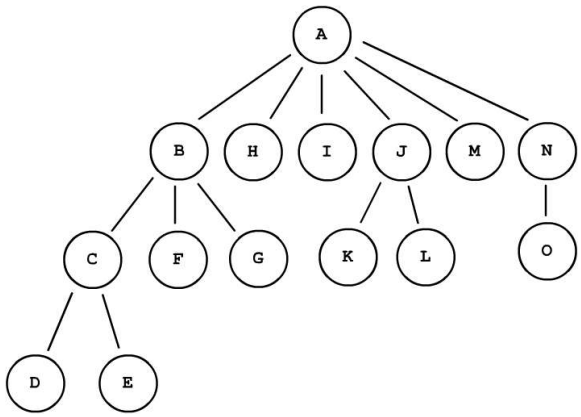


Fig. N-Ary Graph

3.3 Culling

In computer graphics, culling algorithms determine the visible objects in the scene. Any polygons which are deemed to be visible in the scene are sent down the graphics pipeline to be rendered, the rest are discarded or "culled". Many culling algorithms exist in computer graphics such as view frustum culling, which involves testing polygons against the viewing volume of the camera, back face culling, which removes any polygons that do not face the viewer. Occlusion based culling algorithms such as cell, portal, pvs and quad/octree based algorithms attempt to identify the visible parts of the scene, greatly reducing the number of primitives sent down the graphics pipeline, by culling many polygons at a higher level of abstraction then frustum culling [23].

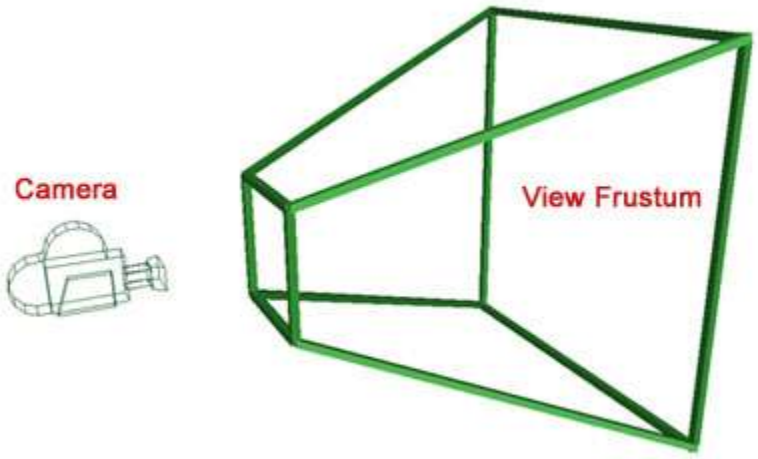


Fig. View Frustum Culling [24]

3.4 State Sorting

In computer graphics API's such as OpenGL and Direct3D, state changes can be very expensive, efficient state changes can greatly improve rendering times. State sorting is about minimising the number of state changes that go down the graphics pipeline. This is accomplished by reorganising the order that calls are made to the rendering API by grouping collections of similar objects together based on material/shader states such as texture, blend functions and surface material[11]. For example, if we have a scene composed of 300 objects rendered with a choice of three textures, in a worst case scenario, rendering the scene in random order could result in 300 texture state changes, which would be very expensive for our rendering time [10]. Through the use of state sorting however, we can guarantee the maximum number of state changes to be 3, which is obviously a vast improvement over the previous attempt. State sorting has often been described as a black art, some state changes are more expensive then others, for example, turning on and off depth buffer writing is far less expensive than turning on or off a light [11]. We must also contend with the fact that while developing specific projects or when using multiple graphics API's some state changes will not be comparatively expensive, so state sorting must often be tailored to a particular API or project.

3.5 Separate render passes

Implementing separate render passes, enables us to draw objects in an order which can efficiently optimise rendering. Many rendering API's are fill rate bound, which means that it is relatively expensive for them to draw pixels once everything has been setup[3]. So every pixel we don't draw saves us time, implementing this can be quite complicated and can involve culling which was discussed previously. An obvious approach however is to not draw any pixels which are behind other pixels. So by splitting the rendering into separate passes we can hope to draw pixels in an order that will minimise the number of pixels drawn.

For example we can first split the objects into two groups, those which are considered opaque and those which are transparent. Obviously the transparent objects must be drawn last, from back to front, for correct blending[3]. Opaque objects can be drawn from front to back or through separate passes, drawing large static objects at the foreground first, followed by dynamic objects such as actors and finally the background plane, which in itself could cover the entire screen[10].

3.6 Level of Detail Nodes

Imagine a scene were we are rendering a character or model. Obviously on close inspection the level of detail/complexity in the model is of importance. However, if we are to view that same model from far away, we cannot make out the same level of detail, yet still render the same complex model to the screen, thus wasting valuable rendering time. A better idea would be to draw the same model at different levels of complexity based on the model's distance from the viewer, this is were level of detail nodes play their part [3].

A level of detail or LOD node in a scene-graph is like a switch statement that is connected to several child nodes, were each child node represents a shape or model at different levels of complexity/detail. The LOD nodes job is to decide on a per frame basis, which child is to be sent to the next stage of the graphics pipeline. The LOD node bases this decision on the complexity of the child and the distance of the object from the viewer, the rest of the child nodes are said to be invisible for this frame. Popular modelling tools such as Alias' Maya incorporate user defined LOD nodes, allowing the designer an extra level of optimisation over Maya's built in techniques.

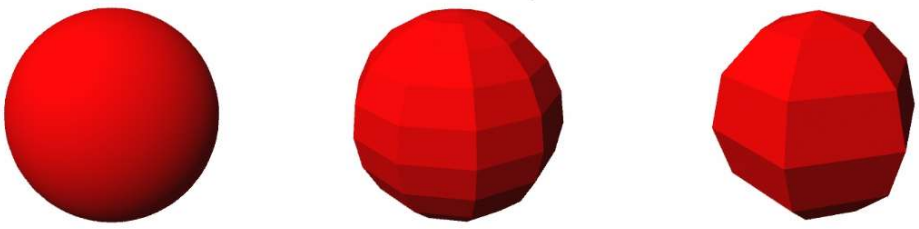


Fig. Spheres at reduced levels of complexity

4 Implementation

4.1 System Overview

The goal of this project was to develop a scriptable scene based rendering engine to be used for the creation of static scenes, that is not only useable but also extensible to include other features such as animation controllers at a later stage. The system has been developed using the C++ programming language in conjunction with the OpenGL graphics library, so as to encourage cross platform portability. Much like the Ogre3D engine discussed previously, our system interface is purely code/script driven and there is no GUI available for scene design. Embeddable scripting is provided by a LUA wrapper class, which provides a secondary interface to our application over the C++ application code. The main priorities of the application in order are; performance, extendible framework, user interface.

Performance: the application should considerably increase the rendering time of interactive applications, the application should also be capable of displaying reasonably complex models and scenes at interactive rates.

Extendible Framework: the system design should provide an intuitive and simple means to extend the application, making good use of object oriented frameworks and suitable design patterns were possible.

User Interface: the application should be user friendly and yet flexible enough to optimise and create multiple types of scenes. It should also incorporate a scripting language to aid in scene design.

4.2 Performance:

Performance as the main priority in our system design and implementation, is tackled with the following optimisation features.

4.2.1 Mesh Loader

The system includes a custom mesh loader which loads wavefront .obj files or a custom binary file. If the binary mesh file does not exist, the obj file is loaded as a preprocessing step, converted to vertex array data and finally written to binary form, ready for use on the applications next run. Vertex arrays are storage buffers, which can contain data such as vertices, texture and normal coordinates[4]. OpenGL provides built in support for vertex arrays, so by storing our mesh data in them, we can greatly enhance our rendering times, allowing for the depiction of reasonably complex models at interactive rates. Also, as we are storing our mesh data on disk in binary format, this allows us to improve the loading times compared to the parsing of .obj files and often results in smaller files.

Fig. Mesh Loader Demo

4.2.2 Frustum Culling

Graphics API's such as OpenGL provide their own clipping algorithms to remove any primitives which are not currently viewable on screen, this obviously improves the efficiency of rendering. The problem is that clipping is one of the last operations performed in the rendering pipeline, meaning that primitives which are not on screen are processed the same as any which are viewable, this results in wasted rendering time. In a small scene this may not be a problem but as the complexity and size of the scene increases this can be detrimental to the frame rate. Frustum culling determines whether or not an object is viewable from a given viewpoint, by testing if that object is in the current view frustum. The view frustum is the volume of space that includes everything that is currently visible from a given viewpoint. By only rendering objects which are in the view frustum we can reduce the number of primitives which are sent down the rendering pipeline and thus greatly improve our rendering times[11].

This system implements an OpenGL frustum culling technique proposed by Mark Morley's "Frustum Culling in OpenGL" tutorial[16]. The frustum is defined by six planes, which form the shape of a pyramid with the top cut off. The test is simple, If a point is inside this volume then it's in the frustum and it is visible, If the point is outside of the frustum then it isn't visible. We extract the frustum coordinates from the current OpenGL projection and modelview matrices, combine them, extract the values from the resulting matrix and use them to define the six planes of the frustum. We then provide functions to test whether a point, sphere or cube is contained in or intersects the frustum, we also provide a function which returns a points distance from the near clipping plane, which we can use to find the z depth of an object to be used by the LOD nodes later in the pipeline.

4.2.2.1 OpenGL View Frustum Extraction

Algorithm 1

// GET THE CURRENT PROJECTION MATRIX FROM OPENGL

// GET THE CURRENT MODELVIEW MATRIX FROM OPENGL

// MULTUPLY THE MODELVIEW BY THE PROJECTION MATRIX TO
// COMBINE THE TWO

// EXTRACT THE COORDINATES FOR EACH OF THE SIX PLANES FROM
// THE RESULTING MATRIX

// NORMALIZE EACH PLANE

// TEST THE DISTANCE BETWEEN A POINT IN SPACE AND EACH OF THE
// SIX FRUSTUM PLANES

// IF THE RESULT IS POSITIVE WE ARE INSIDE A PLANE, NEGATIVE WE
// ARE OUTSIDE, ZERO ON THE PLANE

// IF ALL SIX PLANES RETURN INSIDE WE ARE COMPLETELY INSIDE THE
// VIEW FRUSTUM

// IF WE ARE INSIDE AT LEAST ONE OR MORE PLANES BUT NOT ALL
// PLANES, WE INTERSECT THE FRUSTUM

// IF WE ARE OUTSIDE ALL PLANES WE ARE COMPLETELY OUTSIDE THE FRUSTUM
// AND CAN BE DISCARDED

4.2.3 Octree Culling

Frustum culling helps to improve rendering times, but on a large scene with many objects, culling each object against the frustum indivdually may not suffice. This system implements another culling algorithm which operates on top of the basic view frustum previously discussed, this algorithm is known as Octree culling or Octree Space Partitioning [15]. Octree culling is an advanced culling technique that operates on a higher level of abstraction then frustum culling. An Octree is a hierarchical structure which encloses a scene or group of objects, it is usually defined as a cube.

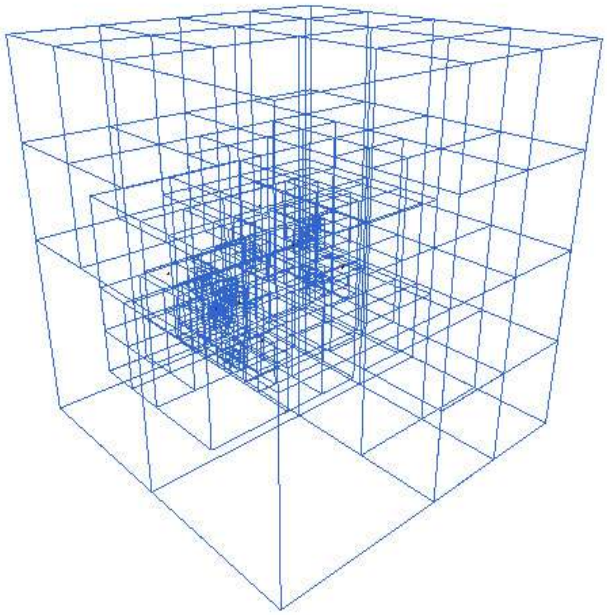


Fig. Octree

The creation of an Octree is generally a recursive procedure, first we insert each object at the root of the tree, following this we subdivide the root into eight child nodes represented by cubes an 1/8th the size of the root cube. We then test individually whether each object fits inside one of the child Octree nodes, if it is completely outside we move onto the next node, if it fits totally inside a node, the object is inserted into this node, this process continues recursively until we find a subdivision which is either too small to enclose the object or an intersection occurs. On intersections the object is attached to the last Octree node that could enclose the object.

4.2.3.1 Octree Creation Pseudo code

Algorithm 2

// CALCULATE THE CENTRE OF THE OCTREE ROOT FROM THE POSITIONS OF WORLD
// OBJECTS

// FIND THE RADIUS OF THE ROOT, THE DISTANCE FROM THE CENTRE TO THE
// FURTHEST OBJECT, DOUBLE THIS FOR THE CUBE WIDTH

// CREATE THE OCTREE CUBE ROOT FROM THE CENTRE AND WIDTH

// INSERT EACH OBJECT INTO THE ROOT

// SUBDIVIDE THE ROOT BY CREATING 8 OCTREE NODES AS CHILDREN INSIDE THE
// ROOT, EACH AN 1/8TH THE SIZE OF ITS PARENT

// TEST WHETHER EACH OBJECT CAN FIT INSIDE ONE OF THE ROOTS CHILD NODES

// IF THEY CAN FIT TOTALLY INSIDE A CHILD NODE, INSERT THE OBJECT INTO THIS
// NODE AND REPEAT THE PREVIOUS STEPS FOR THE CHILD OCTREE NODE

// THIS PROCESS CONTINUES RECURSIVLEY UNTIL WE FIND AN INTERSECTION OR A
// CHILD NODE WHICH IS TOO SMALL TO CONTAIN THE OBJECT, AT THIS POINT
// ATTACH THE OBJECT TO THE LAST OCTREE NODE THAT COULD FIT THE OBJECT
// WITHOUT INTERSECTIONS

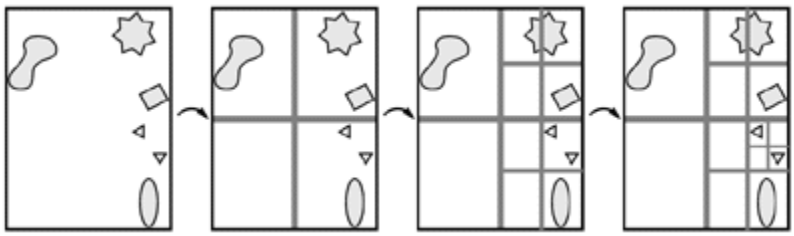


Fig. Octree subdivision & insertion [13]

The process of culling is to first take the root of the tree and test it against the view frustum, if it is visible, recursively test each of the root's nodes against the view frustum until we reach the leaf nodes of the Octree which are the objects themselves or we find a node which is outside the view frustum. In this case, discard the node and any of it's subset and move to the next Octree node. Recursive subdivision of the scene allows the algorithm to cull objects at a higher level of abstraction then with just frustum culling.

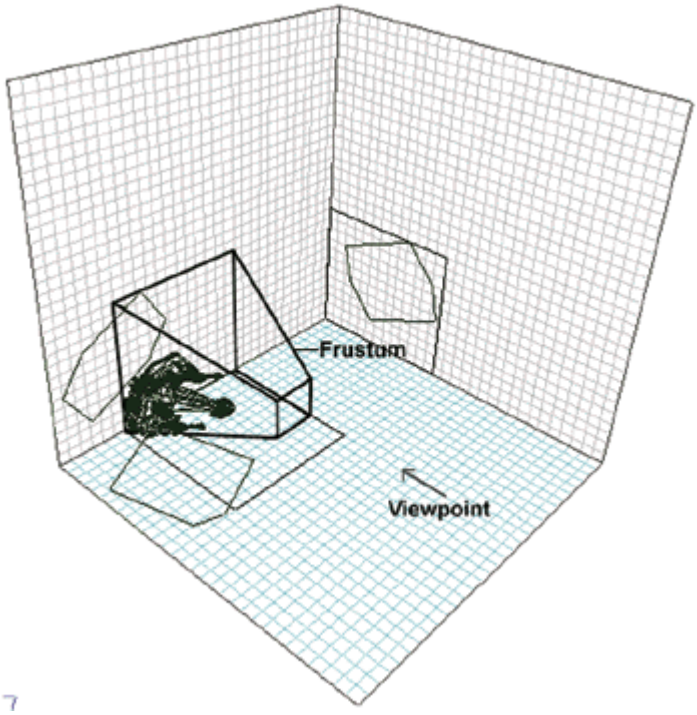


Fig. View Frustum Culling using the Octree [15]

Our implementation of Octree Space Partitioning is at the object level, meaning that objects rather then individual polygons are culled. Octree culling algorithms are best suited to static scenes, as the pre-process of inserting large numbers of objects into the tree can be quite expensive. With some techniques such as loose nodes however, the octree can be made more suitable for dynamic objects.

4.2.4 State Management, Materials & Render passes

Internally OpenGL acts as a state machine and is composed of many different state settings which instruct the API's operation. As previously discussed state changes in Graphics API's such as OpenGL can be very expensive and efficient state management can greatly improve rendering times [22]. In our system, state management is a process performed after culling has taken place. On completion of Octree culling, we possess a list of objects which have been deemed visible for that frame. Each object in the scene has either zero or one materials attached which describe how that object should be drawn, each material in the scene can have zero or many objects attached. In this system, what we are referring to by "Material" is a list of properties which describes how the object attached should be drawn, basic properties of our Material object are texture, specular, ambient, diffuse and emissive components and alpha type. Obviously the Material object can include any properties that could be desired, here we are using a basic set of properties shared by all objects in a scene, which can be extended at a later date to include other properties such as shaders. The process of state management is to sort the order in which objects are drawn by the material properties that define them.

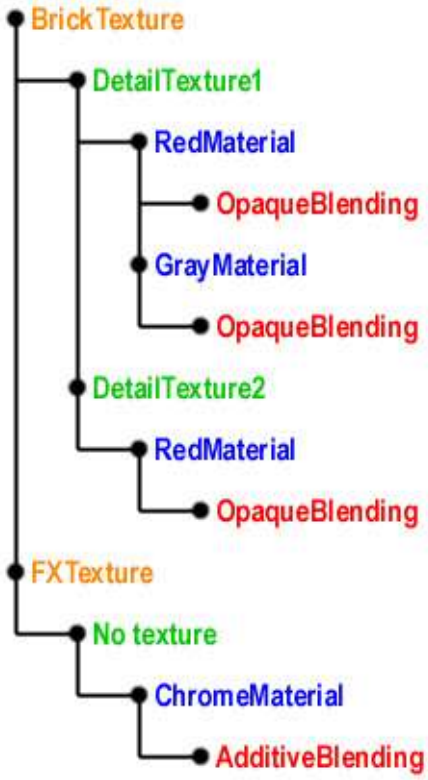


Fig. Material Tree Sort Example [22]

We are sorting materials using a binary sort algorithm, by doing so we can set a precedence on certain properties of our Material. For example we are sorting materials by texture first, as binding textures can be one of the most expensive state changes in a graphics API, this is followed by material components and finally blending function. If we were implementing shaders we would most likely sort materials by shader first, then texture, but as we have noted, state changes are not comparatively expensive on different systems and API's, so often we must tailor state management to a particular project or system, here we have implemented a more generic state management algorithm.

4.2.4.1 State Management Pseudo code

Algorithm 3

```
// STORE MATERIALS ATTACHED TO VISIBLE OBJECTS IN AN STL VECTOR

// OVERLOAD THE < OPERATOR TO SORT MATERIAL OBJECTS WITH THE FOLLOWING
// PRECEDENCE

    // TEXTURE

    // MATERIAL COMPONENTS

    // SHININESS

    // BLEND FUNCTION

// SORT THE MATERIALS USING THE STL VECTOR SORT FUNCTION
```

The process of rendering objects using materials involves setting the appropriate states described by the material properties, rendering the attached object and then releasing any state changes which will adversely effect the next object to be drawn. The Material object implemented by this system provides two functions which handle the setting and release of material properties, they are, use() and release() and act similarly to the OpenGL glPushMatrix()/glPushAttrib() and glPopMatrix()/glPopAttrib() block functions. Calling the use() function sets the material state properties based on the current settings stored by a static Material class member on the client, this may include such operations as enabling of texturing/blending and shader/texture binding. All objects rendered from this point will be drawn using the state settings specified by the use() function. After rendering of an object has been completed we call the release() function which returns OpenGL to it's previous state settings prior to the use function, this may involve calls to disable texturing and blending.

4.2.4.2 Rendering Objects with Materials Pseudo code

Algorithm 4

```
// SELECT A MATERIAL

// CALL THE USE FUNCTION WHICH WILL

// ENABLE BLENDING, DISABLE THE DEPTH MASK AND SET THE
// BLEND FUNCTION FOR TRANSPARENT OBJECTS

// SET THE SHADING MODEL

    // SET MATERIAL COMPONENTS

    // ENABLE TEXTURES IF THEY ARE SET

    // BIND THE TEXTURE

// RENDER THE ATTACHED OBJECTS

// CALL THE RELEASE MATERIAL FUNCTION WHICH WILL ENABLE THE

// DEPTH MASK AND DISABLE BLENDING

// DISABLE TEXTURES
```

Our state management algorithm can also carry out the task of designating objects to separate render passes. Rendering objects through multiple passes can be of benefit to rendering although it can be very scene specific. Here we have split our objects into multiple passes, through our state sorting algorithm. Before material state sorting can occur we must first divide the visible objects into opaque and transparent passes, by comparing the alpha type of each material. This is an important step as transparent objects must be drawn from back to front on a separate pass to avoid blending errors. We can then sort our opaque objects by render pass type, were the types include STATIC for large static objects, DYNAMIC such as actors or animated objects, TERRAIN another static object but one that is known to control more space in a scene, SCENE for a whole scene itself and BACKGROUND for background planes which can possibly take up the whole screen. The order in which the passes is sorted is of importance, here we have designated the following order, SCENE, STATIC, DYNAMIC, TERRAIN and BACKGROUND [3]. Although drawing the objects in this order gives us a good chance of not redrawing pixels on screen, the cost and benefits of the process can be very scene specific, often just sorting by material is more beneficial to render times as the problem of fill rate bound and pixel redraw is becoming less important. Designation of render pass type for each object is left to the end user and is a property found in the scene node itself, this allows for greater flexibility in scene design.

4.2.5 Level of Detail Nodes

A level of detail or LOD node is implemented in our scene-graph data structure which helps to minimize the complexity of objects drawn at different levels of z depth. The LOD node acts like a switch statement, it is connected to several child nodes, were each child node represents a shape or model at different levels of complexity. The LOD nodes job is to decide on a per frame basis, which child is to be sent to the next stage of the graphics pipeline, which in our case is culling. The LOD node bases this decision on the complexity of the child and the distance of the object from the viewer, the rest of the child nodes are set to invisible for this frame and thus are not sent for culling.

The process involves calculating the z depth of the object, which can be found by testing the distance of the object from the near clipping plane of the view frustum. We can then use this z depth to determine which level of detail should be drawn at this depth. So we draw different model's that represent the same structure based on how far away that object is from our viewpoint, up close we draw our highest complexity model and gradually draw less detailed modelled as we move our view further away from the object. The implementation of LOD nodes in the scene-graph is left to the end user.

4.2.5.1 Level Of Detail Pseudo code

Algorithm 5

// CALCULATE Z DEPTH OF LOD NODE AS DISTANCE OF OBJECT FROM VIEW FRUSTUM
// NEAR CLIPPING PLANE

// CALCULATE LOD DETAIL LEVEL BASED ON Z DEPTH

// SELECT APPROPRIATE SHAPE COMPLEXITY BASED ON LOD DETAIL VALUE

// SET REMAINING OBJECTS ATTACHED TO LOD NODE TO INVISIBLE FOR THIS FRAME

4.3 Extendible Framework

4.3.1 System Design

We have chosen to divide the rendering system into four sub systems consisting of, an interface wrapper providing embedded scripting using the LUA scripting language, the scene-graph data structure as the basis for our scene representation, the culling subsystem to provide culling at different levels of abstraction and the state management subsystem to organising state sorting and render passes.

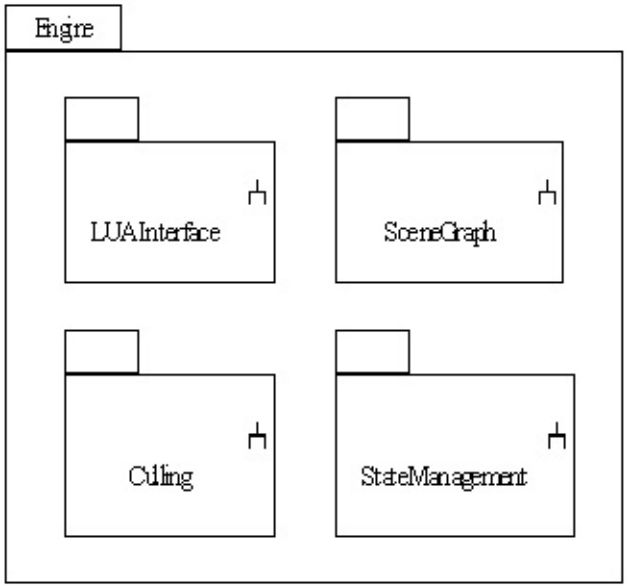


Fig 4.1 Sub-System Diagram

4.3.2 Scene-Graph Data Structure

The scene-graph, the core data structure of our rendering engine and basis for the logical and spatial relationships found in scenes. Here we are implementing the basic data structure as n-ary tree of nodes, consisting of single parents, possessing zero or more children. The scene-graph contains inner and leaf nodes, generally inner nodes are nodes that are not visible or renderable in the scene, who partake in some relationship or carry out an operation, such as Transform and Lod nodes. Leaf nodes represent renderable objects or the actual geometry to be displayed in the scene such as Shape nodes. Every node in the data structure is derived from the base class Node, which provides the basic functionality for creating, attaching and detaching nodes, as well as holding the member variable SceneNodeProperties, which stores the nodes unique identifier, node type (INNER or LEAF), world matrix, global position and render pass. Predefined scene-graph nodes are as follows; Transform, Lod, Shape, Scene, Mesh, Camera and Light. The layout and relationships of the nodes can be seen in the class diagram.

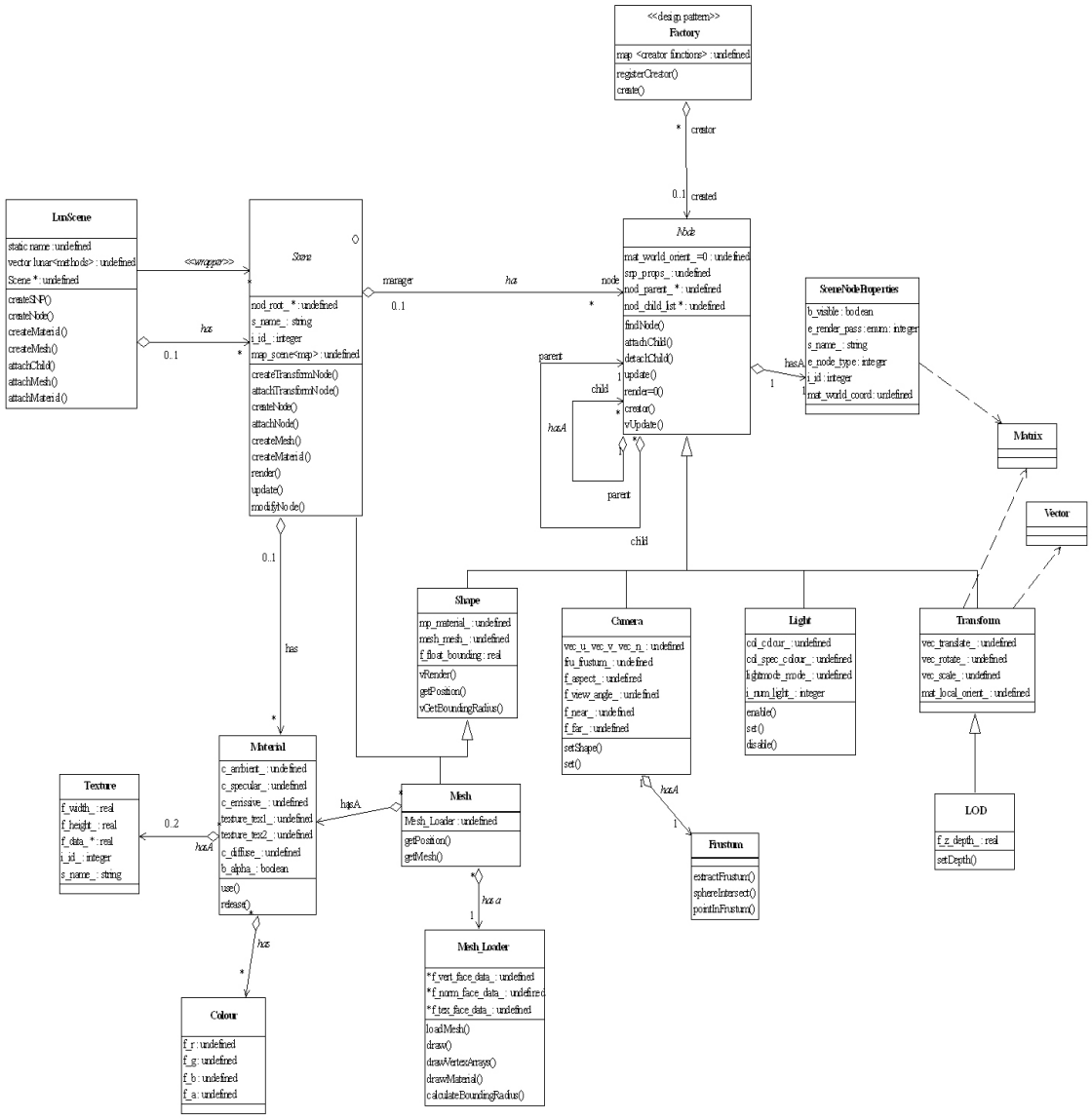


Fig 4.2 Scene-Graph Class Diagram

Although any node can be inserted in the scene-graph, there are some stipulations on the insertion, which are as follows;

- **Only INNER nodes may have children.**
- **LEAF nodes must be attached to an INNER node such as a Transform node.**
- **Nodes must posess a unique identifier for each scene.**
- **Nodes may only be connected to one parent at a time.**

Therefore each leaf node in a scene must be attached to a Transform or Lod node to be inserted in the scene-graph. The reasoning for this is simple, inner and leaf nodes play different roles in the scene-graph. The role of the Transform node is to calculate the local and world orientation and promote the logical and spatial relationships of objects in a given scene. Where as the role of a leaf node is to describe how the object is to be drawn, by seperating the functionality of each object we are promoting a better object oriented framework that will be of great benefit when extending the system in the future.

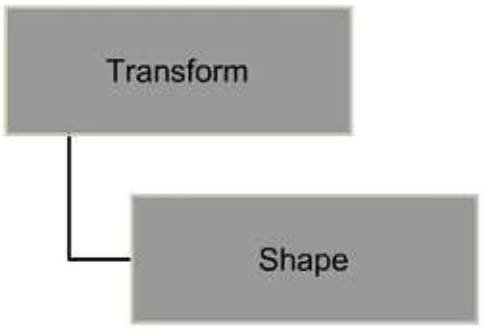


Fig 4.3 Shape attached to transform

Updating the scene-graph is a recursive process which starts at the root or at a desired node in the structure. Through a depth first traversal of each node in the graph we propagate the local and world transformations of each transform node to the leaf nodes, as well as carrying out any operations such as Lod node calculations.

4.3.2.1 Depth First Traversal Recursive Update Function

Algorithm 6

// STARTING AT THE ROOT NODE OR A DESIRED SUB NODE

// CALL THE NODES ACTUAL VIRTUAL UPDATE FUNCTION

// CHECK IF WE HAVE ANY CHILDREN

// IF WE DO, TRAVERSE THROUGH EACH CHILD AND CALL THIS
// RECURSIVE FUNCTION ON THEM

// IF WE HAVE NO CHILDREN OR HAVE REACHED THE END OF THE CHILD LIST WE ARE
// FINISHED OUR TRAVERSAL

The ability to update a subsystem of the scene-graph rather then the whole system is a functionality that will become important in future if we decide to extend the engine to include animation, as it will allow us to monitor the data structure and only update nodes in the system which have been altered since the last frame.

4.3.3 Spatial & Logical Relationships

The distinction between inner and leaf nodes allows us to exploit an important aspect of scene-graphs, the ability to model spatial and logical relationships in a scene [18]. For example, imagine a scene were we have multiple characters in a building, were the building consists of several rooms. In global space, designing this scene could be quite cumbersome, attaching each character to the world and then positioning them one by one in world coordinates. By using the logical and spatial relationships supplied by the scene-graph we can approach this scene in a different manner, first we create the building and attach each room locally to that object, then we create actors and attach each actor through local translations to a room. By modelling the scene in this manner we can orientate each object locally to a parent object, instead of defining world positions for each object.

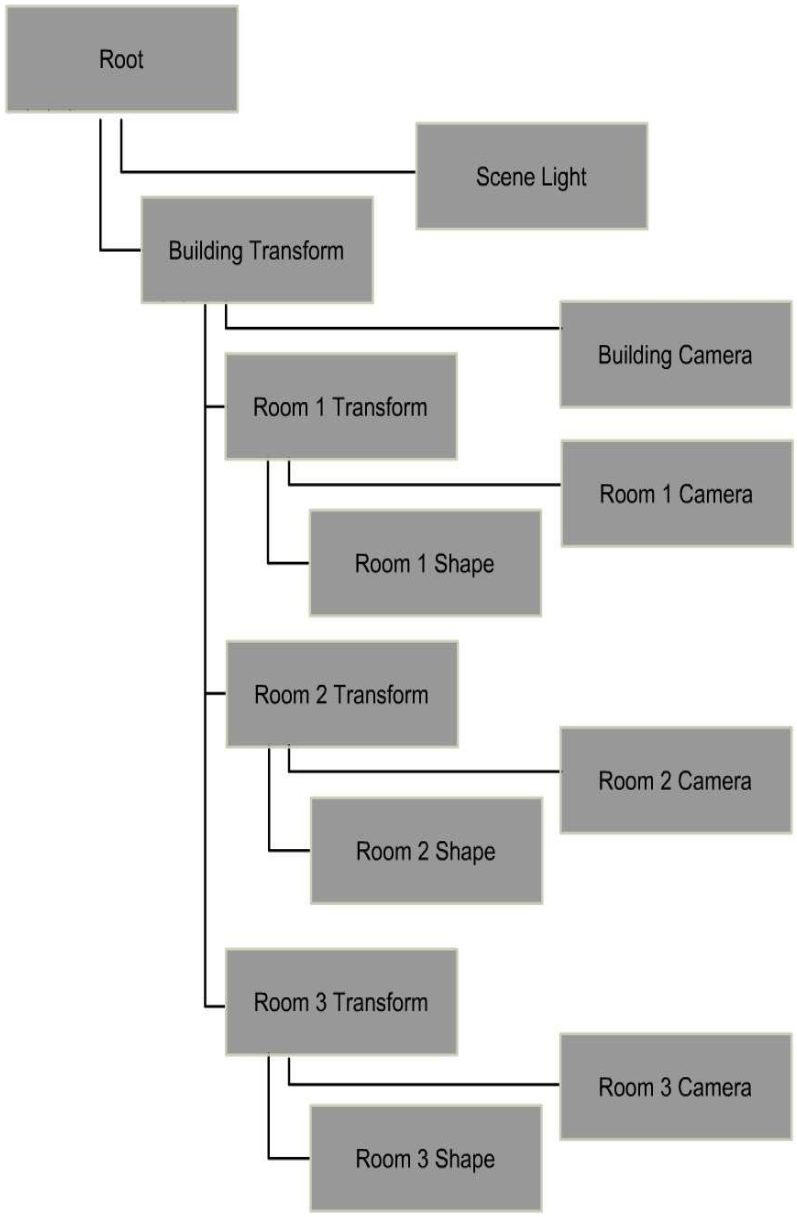


Fig 4.3 Spatial & Logical Relationship scene-graph example

4.3.4 Abstract Shape Class

The predefined abstract shape class provides the basis for renderable objects in the scene-graph, all geometry and visible objects in a scene are derived from the Shape class. The class declares four virtual functions that allow the derived objects to define how they are to be drawn. The functions are as follows; update(), getPosition(), getBoundingRadius() and render(). Separating these definitions from an abstract base class to derived objects allows for more flexibility when deriving user defined shapes. For example as part of the system we have derived a class Mesh, which defines methods for drawing models and meshes using the functionality of the Mesh Loader class. This class becomes very useful when we wish to draw models and meshes using the Mesh Loader class, however it does not supply us with an endless amount of options when rendering objects and models in our scenes. Suppose we want to draw something procedurally or create an effect such as a particle system, using the shape as a base class we can derive our own shapes and tell the scene-graph how they should be drawn and updated conviently, by defining the classes derived virtual functions. This allows for dynamic growth of functionality in the system and the ability to tailor the engine to a user's needs.

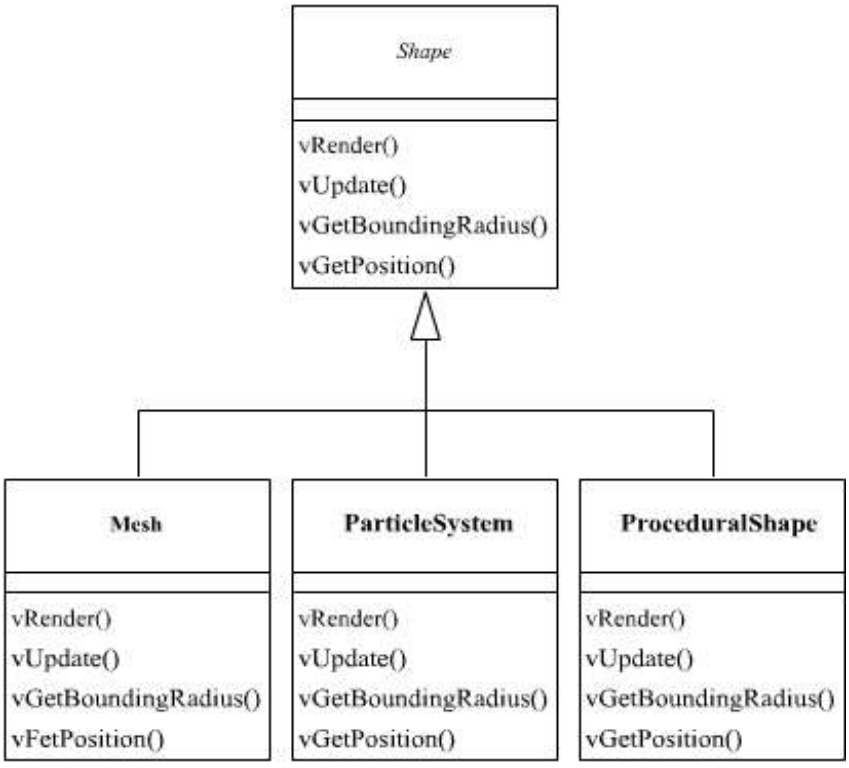


Fig 4.4 Derived Shapes example

4.3.5 Factory Pattern

One of the main goals when designing this application was to create a scene based rendering engine which was not only usable but also extensible. Through the design and implementation of the application we have tried to supply a set of basic nodes and operations which are required for a generic system. However, obviously we cannot supply a node for every desired operation or shape and must supply a means for user defined nodes to be derived from our base class with as little effort as possible. This design results in a problem, at runtime the application cannot anticipate the class of object that it must create. The Application may know that it has to instantiate classes, but it may only know about abstract classes, which it cannot instantiate. Thus the Application class may only know when it has to instantiate a new Object of a class, not what kind of subclass to create [1]. As a result there is an inherent need to not only manipulate user defined objects but also to create them.

The solution to this problem is in the use of a creational design pattern created by Gopalan Suresh Raj, called the Factory Method. The pattern helps to model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate. The major benefit of using this pattern is that the Factory Pattern promotes loose coupling by eliminating the need to bind application specific classes into the code. Thus when deriving user defined objects we do not have to alter or hard code tests to find the type of an object at creation or runtime [21].

"Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses. [1]"

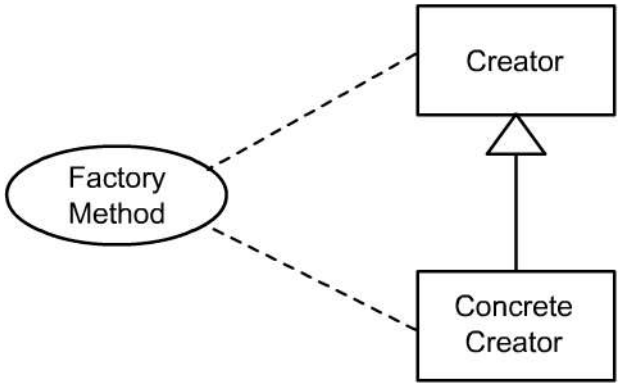


Fig 4.5 Factory Pattern Overview

Our Factory Pattern implementation supplies a factory object which stores a mapping between function pointers and a unique string describing the type of object being created. The function pointers point to corresponding creation functions which define how each class is created. So to create a new object type, we must register the creation function and a unique string describing the object to be created. For example, suppose we have a base node type which we are deriving classes from, to define a new class from this, we just define the class as usual but with one exception. The class must contain a creator function with a signature shared by all derived objects of the base class, which describes what object is being created [1].

4.3.5.1 Factory Pseudo code

Algorithm 7

```
// A FACTORY OBJECT STORES A MAPPING BETWEEN A UNIQUE IDENTIFIER NAMING A
// DERIVED CLASS & A FUNCTION POINTER, WERE THE FUNCTION DESCRIBES HOW
// THAT DERIVED CLASS IS CREATED

// DERIVE A NEW CLASS FROM THE BASE CLASS

// SPECIFY THE NEW CLASSES UNIQUE CREATOR FUNCTION

// REGISTER THE CLASS & CREATOR FUNCTION WITH THE OBJECT
// FACTORY

// TO CREATE A NEW INSTANCE OF THE DERIVED CLASS CALL THE CREATE OBJECT
// FUNCTION OF THE FACTORY BY SPECIFYING THE CLASSES UNIQUE IDENTIFIER
```

The design pattern has provided us with a dynamic scheme for object creation, moving the responsibility for creating objects from one centralized place to each concrete class [1]. Thus making the factory scalable and removing the need to modify code on creation of each new derived class.

4.4 User Interface

4.4.1 Interface Overview

As mentioned previously, our system does not supply a GUI for scene design, the project has been designed as a framework to be used by programmers and as such there is no need at this point to supply a graphical interface. However, as a priority of our system design, the user interface plays an important part in the success of the project, which is why we have supplied two public interfaces to the functionality of our system. The Scene class provides the main interface to our system through a number of member functions, while the LuaScene acts a wrapper class around this, providing embeddable scripting through the Lua interpreter.

4.4.2 Scene Interface

The Scene class provides the main user interface to our rendering engine, it provides functions for creating nodes, creating meshes and materials, updating and rendering scenes, creating and settings lights, attaching and detaching nodes and node subsystems from the scene-graph, creating the Octree, as well as running the culling and state management procedures. The scene also maintains all the nodes, materials and meshes created, as well as maintaining the scene-graph data structure itself.

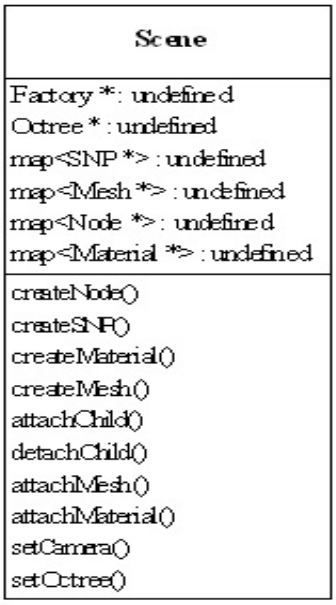


Fig 4.6 Scene Object Diagram

4.4.3 Lua Interface

Our system provides two forms of user interface, the application's C++ methods provided by the Scene object class and a LuaScene wrapper surrounding this class which binds the C++ code to the LUA scripting language, allowing for scriptable interface to the scene-graph subsystem.

"Lua is a powerful light-weight programming language designed for extending applications. Lua is a language engine that you can embed into your application. This means that, besides syntax and semantics, Lua has an API that allows the application to exchange data with Lua programs and also to extend Lua with C functions. In this sense, Lua can be regarded as a language framework for building domain-specific languages. [6]"

Lua's API is not designed to register C++ classes to Lua, only C functions that have the signature int()(lua_State*), it does however provide a low level C API and extension mechanism that makes it possible. Here we are using Lunar[25] to bind our C++ code to the Lua so that we can call our application methods from an external script run in the lua interpreter. Lunar is a template class created by Lenny Palozzi, which provides binding of C++ classes to Lua, allowing us to create and call functions from C++ objects in Lua. The lunar process involves four steps, class registration, object instantiation, member function calling and garbage collection.To bind a C++ class to Lua using this method, there a few requirements the class must meet.

- **The class must have a public constructor that takes a lua_State pointer as an argument**
- **Registered member functions must have the signature int(T::*)(lua_State*)**
- **It must have a public static const char[] member called className**
- **It must have a public static const Luna<T>::RegType[] member called Register**

Implementing Lunar, we first register our class and member functions using the Lunar register() function, specify a static class name so that it can be identified in Lua and then create an instance of our class or pass a reference to it through Lua user data. In our system we create an instance of our Scene class in the C++ application code and then pass it to Lua as user data, with a unique Lua global reference name. In the Lua script we can then obtain a reference to our object using this global reference.

4.4.4 Scenes as scene-graph nodes

Imagine we are designing a city scene, organising such a scene would be quite complicated to do in one scene-graph data structure, especially when we consider the number of nodes required to model a large city. It would be a better idea if we could design small subsets of the overall scene and then insert those subsets into a larger scene, a scene of scenes. Our system provides this functionality by allowing the scene object istelf to be part of a scene-graph, this functionality is provided by deriving the scene class from the shape class, so in essence when we create a scene we are really just creating another object or set of geometry to be rendered. As we have a scene of scenes, this means that we also have an octree of octrees as each scene contains it's own octree, this greatly benefits culling, as it gives us the ability to cull whole scenes as part of the Octree's recursive culling technique.

4.4.5 Camera & Light Classes

The engine also defines a simple camera object based on the hill "computer graphics using OpenGL" programming book [5], the object also includes a frustum that is used for view frustum culling. The class provides functions for setting camera projection and orientation. As a node in the scene-graph the camera has the functionality of the base node class and thus can be attached to other nodes such as the Transform node and using this parent to calculate it's orientation.

The user can also create lights and attach them to other nodes in the scene-graph in the same manner as the Camera. The Light class provides functions for setting light properties, as well as enabling and disabling the Light.

5 Using the System

The rendering engine is available for both Windows and Linux, through separate releases.

5.1 Installation

Linux

The following library dependencies must be installed prior to building applications using the system.

- **DevIL – Image Library**
- **Lua 5.1 – Lua library files**
- **OpenGL – OpenGL graphics library**

Once these dependencies have been installed, we can develop applications using the system by including the header and source files, a static library will be available shortly.

Windows

The following library dependencies must be installed prior to building applications using the system.

- **DevIL – Image Library**
- **Lua 5.1 – Lua library files**
- **OpenGL – OpenGL graphics library**

Once these dependencies have been installed, we can develop applications using the system by including the header and source files, a dynamic library will be available shortly.

5.1 The “Hello World” Application

To demonstrate a simple use of the rendering engine we will briefly describe a basic application.

The first requirement of the application is to include the appropriate system header files along with any other header files we wish to include:

```
#include "scene.h"
#include "luascene.h"
```

The next step is to register the lua wrapper class and functions, as well as the global material state:

```
//define lua class name
const char LuaScene::className[] = "LuaScene";

//define the methods we will expose to lua
#define method(class, name) {#name, &class::name}

Lunar<LuaScene>::RegType LuaScene::methods[] = {
        method(LuaScene, createSNP),
        method(LuaScene, createNode),
        method(LuaScene, translate),
        method(LuaScene, attachChild),
        method(LuaScene, attachChild2Parent),
        method(LuaScene, createMesh),
        method(LuaScene, attachMesh),
        method(LuaScene, createMaterial),
        method(LuaScene, createMaterialTex),
        method(LuaScene, attachMaterial),
        method(LuaScene, setLight),
        method(LuaScene, enableLight),
        method(LuaScene, setSpotLight),
        method(LuaScene, setCamera),
        method(LuaScene, setOctree),
        method(LuaScene, update),
        {0,0}
};
```

```
//set static material variable
Material *Material::mtl_current_state_ = new Material();
```

The next step is to create a scene and load it using a lua script called test.lua

```
//// LUA SCRIPT INITIATION ////////////////////////////////////////////////////////////////////////////////

//open the lua state
lua_State *L = lua_open();

//load default libs
luaL_openlibs(L);

//register lunar methods
Lunar<LuaScene>::Register(L);

//allocate scene memory for scenes
Scene *scn_scene1 = new Scene();

//push the objects into lua
lua_pushlightuserdata(L, scn_scene1);
lua_setglobal(L, "scene");

std::cerr << "-- Loading Lua Script: " << std::endl;
int i_status = luaL_loadfile(L, "test.lua");
std::cerr << "-- Execute Lua Script: " << std::endl;

//if status is 0
if(i_status == 0)
{
        //execute script
        lua_pcall(L, 0, 0, 0);
}       //end if

//report any errors
reportErrors(L, i_status);

//close script/lua
lua_close(L);
```

Finally in our OpenGL update function we call the scene update function and in our display function we call our render function.

```
void Update(int i)
{

	//call the update function
	scn_scene1->update();

	glutPostRedisplay();
	glutTimerFunc(60,Update,0);

}	//end method definition


void myDisplay()
{
	//clear the colour and depth buffers
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glColor3f(1.0f, 1.0f, 1.0f);

	//load identity matrix
	glLoadIdentity();

	//set global rotations
	setGlobalRotations();

	//render the scene
	scn_scene1->vRender();

	//swap buffers
	glutSwapBuffers();

}	//end method definition
```

Demo Applications have been included on the source CD.

6 Conclusion

The goal of this project was to develop a performance driven scriptable scene based rendering engine to be used for the creation of static scenes, that is not only useful to developers but also easily extensible, with the main priorities of the application being; performance, extendible framework and user interface.

The system has certainly achieved it's primary goals and priorities, although in some areas more then others. Rendering performance for demos has increased through the use of the system's culling, state sorting and level of detail procedures, with vast improvements for spacious scenes through the use of Octree Culling and notable improvements for densley populated scenes using level of detail nodes. State management provides improvement for large scenes, were multiple objects share the same materials. The system did not however show substantial increases through the use of multiple render passes, although it can be mentioned that separate render passes would greatly improve performance if the Octree had implemented polygon level culling as well as object level. Performance could also be improved by extending the Octree algorithm to include Z buffer occlusion, a routine discussed in Real Time Rendering - second edition [13]. Established rendering engines such as Ogre3d include many different forms of culling, as an extension, the system could also include portal/cell culling. Portal culling is a process which divides a scene into separate rooms or cells and connects them by a door or opening which is referred to as a portal, the parts of the scene which are deemed visible are the sections that can be seen through the portal, thus allowing us to clip the connected sectors against the portal boundaries[?], portals are best suited for indoor scenes such as mazes. An obvious development would see the inclusion of animation in scenes, which could be controlled by a keyframe node found in the scene-graph that acts as an operation on a shape much like the LOD node.

Possibly the more complicated and time consuming portion of development was the system design, as an important aspect of the project was extensibility much time was put into this section, which proved very successful. The system has provided an easily extendible framework through the use of design patterns such as the Factory method, which has been implemented in the scene class through object registration and creation. The object oriented layout of the scene-graph data structure has also benefited the derivation of new node types, including Shape instancing that has been implemented in the Mesh class.

The system provides completely automated culling and state management and the simple user implementation of level of detail nodes, allowing the end user to concentrate more on scene design and less on optimisation. The interface provides two forms of functionality, through the public Scene methods supplied and a lua scripting wrapper surrounding these

base methods. Although the Lua interface is an improvement over the hard coded Scene functionality, managing to separate testing and scene design from application code, some aspects could be improved such as error and debugging information which is quite sparse in some cases. In future, scripting could also be implemented to setup the basic OpenGL framework required to run an application, which would further abstract application code to Lua. The ability to include Scenes in the scene-graph has greatly improved the scene design interface, making it a much easier and less complicated process to create a large scene. Further developments in the system could see the inclusion of a mesh decimation tool that would automatically reduce the complexity of a model connected to a LOD node.

7 Reference

[1] Alexandrescu, A., Modern C++ Design – Generic Programming and Design Patterns Applied, 2001. Abstract Factory. Addison Wesley

[2] Eberly, David H., 3D Game Engine Architecture, 2005. Scene-graphs. Kaufmann Publishers Inc

[3] McShaffrey, M., Game Coding Complete - 2nd Edition, 2005. Scene-graphs. Paraglyph Inc.

[4] OpenGL Architecture Review Board, OpenGL Programming Guide, 2005. Addison Wesley

[5] Hill, F.S., Computer Graphics Using Open GL, 2000. Prentice Hall

[6] Leruselimschy, R., Programming in Lua, 2006. Lua.org

[7] [Anon], [no date]. Lua Tutorials [online]. Available from http://lua-users.org/wiki/ [Accessed August 2006]

[8] [Anon], [no date]. OpenSceneGraph Project [online]. Available from http://www.openscenegraph.org/ [Accessed July 2006]

[9] [Anon], [no date]. Ogre3d rendering engine [online]. Available from http://www.ogre3d.org [Accessed July 2006]

[10] [Bar-seev, A], [no date]. Scene-graphs past, present & future [online]. Available from http://www.realityprime.com/scenegraph.php [Accessed July 2006]

[11] [Anon], [no date]. State sorting tutorial [onlline]. Available from http://opengl.j3d.org/tutorials/statesorting.html [Accessed July 2006]

[12] Möller, T., Haines. E, Real time rendering, 2002. Hierarchical Z Buffering. AK Peters, Ltd

[13] [Anon], [no date]. , Octree Culling [online]. Available from http://www.distanthumans.info/programming/java/misc.php [Accessed August 2006]

[14] Greene, N., Kass, M., Miller, G. 1993, Hierarchical Z-Buffer Visibility.

[15] [Kelleghan, M.], [no date]. Octree Partitioning Technique [online]. Available from http://www.gamasutra.com/features/19970801/octree.htm [Accessed July 2006]

[16] [Morley, M.], [2000]. Frustum Culling using OpenGL [online]. Avilable fromhttp://www.crownandcutlass.com/features/technicaldetails/frustum.html [AccessedJuly 2006]

[17] [Anon], [no date]. Lord of the Rings Battle for Middle Earth IMG [online]. Avilable from http://www.ign.com [Accessed July 2006]

[18] [Anon], [no date]. Scene-graphs [online]. Avilable from http://en.wikipedia.org/wiki/Scene_graph [Accessed July 2006]

[19] [Anon], [no date]. Pacifice Storm IMG [online]. Available from http://www.pacificstorm.net/ [Accessed July 2006]

[20] [SGI], [no date]. SGI's Performer [online] Available from http://www.sgi.com/products/software/performer/ [Accessed July 2006]

[21] [Raj, G. S.], [no date]. Factory Pattern [online]. Available from http://gsraj.tripod.com/design/creational/factory/factory.html [Accessed August 2006]

[22] [Nuydens, T.], [no date]. State Management [online]. Available from http://www.delphi3d.net/articles/viewarticle.php?article=stateman.htm [Accessed August 2006]

[23] [Anon], [no date]. Culling [online]. Available from http://www.gamedev.net/reference/programming/features/culling/ [Accessed July 2006]

[24] [Anon], [no date]. View Frustum Culling [online]. Available from http://www.c-unit.com/tutorials/mdirectx/?t=45 [Accessed September 2006]

[25] [Palozzi, L.], [no date]. Lunar C++ Binding [online]. Available from http://www.lua.org/notes/ltn005.html [Accessed Ahugust 2006]