Scene Graph Management for OpenGL based 3D Graphics Engine
Adnan M. L. Karim, M. Sazzad Karim, Emdad Ahmed, Dr. M. Rokonuzzaman
Department of computer Science, North South University, Dhaka 12, Bangladesh
Emails: adnan@esophers.com, remon316@hotmail.com, emdad@northsouth.edu, rzaman@northsouth.edu

ABSTRACT
Scene Graph is high level representation of a 3D world that can be used to manage objects in a 3D graphics engine. As the number of polygons and entities in the 3D scene increases, so does the complexity for efficient management of the scene. Sending all the geometry to the graphics hardware is a huge waste of processor power and memory bandwidth. It also hinders the quality of the 3D scene. Therefore, efficient management of different type of objects is the key to fast rendering. Use of Scene Graphs for graphics engines differs in design and implementation but the general features of Scene Graphs are used in most engines. In this paper we present a Scene Graph management technique utilizing different hidden surface removal methods for faster and more efficient rendering of interactive 3D environments.

Keywords: OpenGL, API, Renderer, Scene Graph, Space Partitioning, Frustum, BSP, Octree, Quad tree, Zone, Portal.

1. INTRODUCTION
3D graphics scenes basically comprise of vertices, faces, textures, camera, lighting, environment effects etc. For simple scenes, rendering directly without any data management does not pose any problem. But as the scene becomes more complex and advanced, interactivity and performance becomes an issue. Then simply rendering all the data available in a scene is not the best way to do it. Here scene graph can provide an efficient solution in managing scene data. A Scene Graph is basically a ‘Graph’ data structure which represents an object hierarchy of a 3D scene [3]. It can be either a Directed Acyclic Graph (DAG) or simply an n-node tree. Organizing objects in a scene graph itself needs careful consideration and we will discuss some techniques to manage objects and render them using OpenGL.

2. GENERAL SCENE GRAPHS
Scene graphs contain Nodes and Arcs. A node contains the data and arcs are relationships between nodes. The content of nodes can be static or dynamic and arc determines the organization of these data in the scene graph. A traversal algorithm is applied to the graph to determine what to render, collision checking and other information. Applying different traversal algorithm would output different results. Also, change in the organization of the nodes or the changed data in the nodes would also cause a different output.

2.1 Node Types
Scene graph contains different types of nodes. As mentioned above, a scene usually comprises of varieties of information.

The nodes must contain all those information. What is contained in each node varies from implementation to implementation. Following are the standard node types found in most APIs:

- Geometric node contains the vertices data, surface normal or description of the shape of the objects.
- Appearance node holds the material description as well as texture information.
- Dimension node have object position, transformation, size and orientation data.
- Group nodes determine the hierarchical organization and composition of other nodes.
- Non-visual nodes can contain other information like sound and behavior data.

In some implementations, the nodes are self contained, that is, each node defines an object entity and has all the necessary of information defined within the node. This particular implementation is convenient for most graphics engines.

2.2 Organization of a Scene Graph
Nodes in the graph are usually attached in a hierarchical manner. Arcs in the graph are directional. They represent parent-child or sibling-sibling relationships. Each object is divided into smaller sub-objects. For example, a 3D world can contain buildings, forests, and houses etc as its children. Each child node is attached to the parent node (e.g. a House attached to a world/terrain node). In this representation, the position and orientation of the children are determined by the position and orientation of the parent. As shown in Figure 1, we can see the partial scene graph of an imaginary world. Here we have two buildings and have shown the expansion of the nodes of “Building 1” and “Forest”.

![Figure 1: Hierarchical Object Scene Graph](image-url)
The organization may vary in different implementation. The figure above is for a generic scene graph. The nodes are self-contained and has all the information needed to render an object. The scene traversal algorithm starts from the roots and goes on a depth first basis, rendering a complete object. For instance, here, it first goes to the root, then to “Sky”, then “Terrain”. Here Terrain first renders itself then its children: “House”, “Building1 etc. And within “Building1” it renders “Table”, “Chair”, “Statue” and “Man” entity after it renders itself.

2.3 Rendering the Objects in Scene Graph

We go through each object in the scene graph and render them each frame. Only the Draw function of the root object need to be invoked to draw the whole scene as each node recursively draws its children. The generic traversal algorithm is as follows [1],

```c
void Draw ( CAgniCamera *camera )
// Draw this object and objects in its subtree
{
    glPushMatrix( );
    OnDraw ( camera ); // Render this object
    if( HasChild( ) )
        ((CAgniObject*)m_pChild) -> Draw ( camera );
    glPopMatrix();

    // Draw siblings
    if ( HasParent( ) && !IsLastChild( ) )
        ((CAgniObject*)m_pNext) -> Draw ( camera );
} // Draw calls the OnDraw function of a node and its subtree
```

OnDraw function actually renders the polygons of the object, preferably using glDrawElements( ... ) to maximize performance using OpenGL that draws a group of polygons with same material properties [4].

2.4 Collision Detection in Scene Graph

Scene graph can also be used to process collision between the different objects in its nodes. For collision detection, generally object’s bounding volume is checked against another object’s bounding volume. Just like the rendering traversal, collision traversal goes all the way down to leaf node once it finds the topmost predecessor of the object it is colliding with. The algorithm is as follows [1],

```c
void ProcessCollisions( CAgniObject *object )
{
    //Bounding volume check with object
    if ( Intersects((CAgniObject*)this->BoundingBox, 
        object->BoundingBox ) == true 
        && ( object != ((CAgniObject*)this) ) )
    {
        // this object collided with “object”
        OnCollision( object );// Perform collision with “object”

        //Child collision with object
        if ( HasChild( ) )
            ((CAgniObject*)m_pChild)->
                ProcessCollisions( object );
    }

    //Sibblings collision with object
    if ( HasParent() && !IsLastChild( ) )
        ((CAgniObject*)m_pNext)->
            ProcessCollisions( object );
}
```

2.5 Review of the General Scene Graph

Although, this is very easy and straight forward and does the basic work, it needs some modifications. Every object in the world is rendered each frame and with the same rendering technique whether we need it or not. The collision detection is also very simple and does not provide more precise per-face collision detection. We will discuss some methods that will improve rendering and will provide better support for occlusion and collision detection in the following sections.

3. OPTIMIZATION OF THE SCENE GRAPH – VISIBILITY CULLING

The generic scene graph implemented here can be optimized by using visibility culling techniques without changing the current organization, but just changing some properties of the nodes. We could add to the traversal algorithm the calculations for frustum culling [2].

3.1 FRUSTUM CULLING

The renderer would take the parameters of the view camera and calculate the view frustum (as shown in Figure 2) against the bounding volume of each object’s node, starting from the root node. So if the view frustum does not intersect the bounding volume of a parent node, then it does not intersect the bounding volume of its child nodes either. So the whole sub graph of that parent can be culled. So before rendering, the view frustum is checked every frame. This gives a significant performance boost for rendering huge worlds.
However, basic frustum culling is not effective enough for large scenes with complex objects. If we see a small part of a huge object the whole object will be drawn. For outdoor scenes, most of the objects in front of the viewer are within the viewing frustum, so we have to render all of them. Moreover, in both outdoor and indoor scene, the objects occluded behind another are also rendered, which is unnecessary. Object management in hierarchical scene graphs can be improved by using special object nodes resulting from the application of spatial-partitioning techniques like BSP-tree and Octree/Quad tree.

### 3.2 BSP TREE

BSP tree data structure can be used for complex structures (especially indoor) in the world. We can either manage the whole world’s static data with a BSP tree or we can use a tree for each of the complex structures. In the latter case we can still have our scene graph manage the world objects as it is designed by just using BSP for particular objects. The BSP tree is one of the slowest geometry structures to render. Therefore, an efficient way to optimize rendering performance is, defining Zones and Portals in the 3D structures at design time as discussed in [7].

#### 3.2.1 BSP Tree Construction

The BSP tree must be constructed before the scene is rendered. In contrast to Octree, BSP trees recursively divide space into pairs of subspaces, each separated by a plane of arbitrary orientation and position [5]. But the planes should be chosen such that they cut fewer polygons and also generates a balanced tree (more preferable for traversing). To start with the process the scene geometry or object geometry is compiled into a tree format that contains the structures of the computed BSP-tree such as: Nodes, Leafs, Faces (polygons, patches), and Vertices. Each Node and Leaf contains the information about which Zone it belongs to. A Zone is a convex section in the scene that can be rendered independently. This prevents the renderer from checking each polygon in the scene for visibility. Portals are used to divide Zones, which are nothing but invisible sheets between Zones. Here is a BSP tree construction technique we used based on the algorithm presented in [5].

```cpp
// BSP Tree Class and Construction Process:
class BSPTree
{
    private:
        Polygon3D nodePoly; // the polygon that defines the partition plane
        BSPTree front; // front child
        BSPTree back; // back child
        int totalSplit;

    public:
        BSPTree()
        {
            totalSplit = 0;
        } // constructor BSPTree
        BSPTree(Vector poly)
        {
            MakeBSP(poly);
        }

        // overloaded constructor
        BSPTree MakeBSP( vector<Polygon3D> pl )
        {
            vector<Polygon3D> inFront;
            // list of polygons in front
            vector<Polygon3D> behind;
            // list of polygons in back
            BSPTree retval; // new BSP tree
            int l = pl.size();
            // number of polygons in list pl
            int j = SelectPoly( pl );
            //* select best possible polygon from list
            retval.nodePoly = (Polygon3D)pl[j];
            // selected polygon from pl
            for( i=0; i < l; i++ )
            {
                if ( i != j )
                { // check other than root node
                    Polygon3D q = (Polygon3D)pl[i];
                    // other Polygon
                    if ( p.relation(q) == INFRONT )
                        inFront.push_back(q);
                    // add poly to front
                    else if ( p.relation(q)==BEHIND )
                        behind.push_back(q);
                    // add poly to back
                    else if ( p.relation(q)==INTERSECTS )
                    {
                        // split this convex Polygon into two
                        totalSplit++;
                        Polygon3D split[2];
                        split = p.splitPolygon(q);
                        // returns two split polygons
                        inFront.push_back(split[0]);
                        // first polygon add inFront
                        behind.push_back(split[1]);
                        // second polygon add in behind
                    }
                }
            }
            if ( !inFront.isEmpty() ) retval.front
            return retval;
        } // method MakeBSP
    }
}
```

The construction of the BSP Tree shown above depends on the function SelectPoly( list of polygons ). The algorithm for SelectPoly is usually based on a heuristic that chooses a polygon
for the partition plane based on, a) the polygons in both sides of the plane, b) number of polygons that are split by the plane. The former criteria create a balanced tree if there are almost same numbers of polygons in both sides, and the later creates a tree by splitting minimum polygons.

3.2.2 BSP Tree Rendering

Rendering starts by determining the Zone where the view camera is (if Zone/Portals are used). For each portal seen inside the camera’s view frustum, the connected Zones are rendered along with the Zone the camera is in. Each Zone contains a list of BSP-tree leafs; when a Zone is selected to be rendered all the polygons in containing leafs are send to the graphics pipeline. Here polygon sorting may not be necessary as OpenGL uses hardware Z-Buffer [5].

3.2.3 Advantage of Using BSP

BSP does have significant advantages over other space partitioning techniques. First, zoning, if properly done significantly removes a significant chunk of polygons (that are in other zones, not visible) from being passed into the graphics rendering pipeline. Second, BSP geometry handles collision detection very efficiently, which is a vital part of a graphics engine that deals with an interactive world.

3.3 OCTREE/QUADTREE

Octree is a special kind of data structure where every node has exactly eight children (hence the name Octree). Octree is fairly easy to construct and can be used for rendering geometry, occlusion culling and collision detection. A Quad tree is similar to an Octree, with only 4 children for each node instead of 8. For larger and more open terrains, Quad trees some time provide better culling. In general, the construction and rendering methods of Octree can also be used with Quad trees.

3.3.1 Octree Construction

The whole world geometry data is subdivided into eight octants. Recursively each octant further derives eight more children. This goes on until a predetermined tree depth is reached or each octant reaches a maximum polygon threshold. Each octant is actually the bounding cube of a group of data; the polygons are generally stored in the leaves. The root node contains the bounding cube of the whole 3D world. Figure 3 shows a demonstration of the Octree generated subdivisions.

3.3.2 Rendering Data in Octree

While rendering, the view frustum is checked recursively through the Octree nodes. As before, if the camera is not positioned in an octant or view frustum does not intersect a parent zone or octant, none of its children geometry gets rendered. Since the grouping is based on polygons, many polygons outside of view are removed from rendering.

Here is the Octree rendering code that we used in our graphics engine; the rendering is done using OpenGL function glDrawElements( ... ) [4]:

```c
// Draw Octree: usually drawing the Root node recursively
// draws all the nodes in the tree that are attached with Root
void CAgniOctreeNode::Draw ( CAgniCamera *camera )
{
    // First check if this Node is inside the viewing frustum
    if ( !camera->m_Frustum.ContainsAABBox ( m_Min.x, m_Min.y, m_Min.z, m_Max.x, m_Max.y, m_Max.z ) == OUTSIDE ) return;

    // Otherwise ___
    if ( !m_bIsLeaf )
    {
        m_pChildNode[TOP_LEFT_FRONT]->Draw ( camera );
        m_pChildNode[TOP_LEFT_BACK]->Draw ( camera );
        m_pChildNode[TOP_RIGHT_BACK]->Draw ( camera );
        m_pChildNode[TOP_RIGHT_FRONT]->Draw ( camera );
        m_pChildNode[BOTTOM_LEFT_FRONT]->Draw ( camera );
        m_pChildNode[BOTTOM_LEFT_BACK]->Draw ( camera );
        m_pChildNode[BOTTOM_RIGHT_BACK]->Draw ( camera );
        m_pChildNode[BOTTOM_RIGHT_FRONT]->Draw ( camera );
    }
    else
    {
        // It is a leaf
        // If there isn't a list of valid face groups, return
        if ( !m_pGFaces ) return;

        // Draw the faces assigned to this leaf
        DrawLeafFaces ( );
    }
} // end Draw( … )
```

Figure 3: Octree Demonstration
// Draw the faces/triangles in this leaf
void CAgniOctreeNode::DrawLeafFaces()
{
    glEnable(GL_TEXTURE_2D);  // Enable texture
    glColor3ub(255, 255, 255); // reset color

    // For all the Face Groups in this Leaf ...
    for (int i = 0; i < m_nGFaces; i++)
    {
        // Get the material ID
        int shader = m_pGFaces[i].shaderID;
        if (shader > -1)
        {
            // Apply material properties
            // Get the i’th Material
            CMaterial &pMat = m_pSource->m_Materials[shader];

            // If this material has a valid texture ID apply texture:
            if (pMat.textureID > -1)
            {
                glBindTexture(GL_TEXTURE_2D, pMat.textureID);
            }
            else // Apply the diffuse color
            {
                glColor3ub(pMat.diffColor[0], pMat.diffColor[1], pMat.diffColor[2]);
            }
        } //if (shader > -1)

        // Now draw the vertices in this Group via previously
        // enabled and initialized Vertex Arrays
        glDrawElements(GL_TRIANGLES, m_pGFaces[i].nFaceIndices, GL_UNSIGNED_INT, m_pGFaces[i].pFaceIndices);
    } // for (int i = 0 ...
}// end DrawLeafFaces( …

3.3.3 Advantage of Using Octree

Octree mechanism provides a balanced solution for both outdoor and indoor scenes. For large terrains it gives better performance than other data structures; a Quad tree can also be used in rendering large terrains. Octree can be computed within reasonable time so it can be recomputed if necessary in runtime. But still just frustum culling may choose some octants to render which are occluded from view. Therefore an efficient occlusion culling algorithm should be used to boost performance of an Octree based system, e.g. using Hierarchical Occlusion Maps [6].

4. INTEGRATING BSP/OCTREE WITH SCENEGRAPH

Our generic scene graph implementation uses hierarchical object representation whereas Octree/Quad tree and BSP/Portals use hierarchical geometric data representation. There are inherent design differences among them.

To integrate the BSP tree to our scene graph we define each of the BSP tree nodes as a special derivation of the scene node object. Then we attach the root node of the constructed BSP tree to any specific node of the scene graph.

We can integrate an Octree structure in two ways. One way is to derive the whole Octree as a special object of the scene graph and attach the Octree as an object node. Another way is, we can derive each node of the Octree from a scene graph object node and attach the root node of the Octree with the scene graph. The later way is better as the inherent structure of the Octree generate nodes that nicely fit into our scene graph structure.

Figure 4 shows a scene graph that uses both a Quad tree and an Octree. In this design, the scene graph uses hierarchical objects to represent the world, where each object can be an entity, general object, BSP tree node, Octree node etc. With advanced inheritance and abstraction mechanism, this would provide a highly optimized renderer.

5. CHOOSING THE BEST CULLING SOLUTION

Choosing the best space partitioning technique, or not to use at all depends on the 3D scene in question. Now a days graphics applications demand rendering scenes with thousands of polygons in complex outdoor and indoor environments. BSP trees were good solutions thus far, but its complex construction and use limits the number of polygons in the scene to be rendered. Moreover there is a design complexity to define Zones and Portals efficiently for every type of scene. But Octree and its derivations can handle large number of polygons and allows the use of efficient occlusion culling. So, Octree is the better solution of the two that also conforms to our scene graph design.

6. RECOMMENDATION AND FUTURE WORKS

Occlusion techniques should be used that can further reduce the number of polygons drawn. Now some graphic hardware vendors are providing occlusion features implanted in hardware e.g. OpenGL extension named GL_ARB_occlusion_query, and GL_NV_occlusion_query in some nVidia cards [8]. Hierarchical Occlusion Culling can be done in software efficiently as proposed in [6]. Octree and its other derivations (Quad tree, Loose Octree, Adaptive Octree etc.) are excellent structures to support occlusion features.
7. CONCLUSION

Scene graphs need to be very dynamic and employ abstraction from details. It should be designed such that its functionality can be later enhanced with minimum modifications. With the advancement of today’s computer graphics hardware, more advanced and optimized mechanism to manage 3D scenes are becoming available. Our design of the scene graph provides the flexibility to add objects with different properties and complexity and let the renderer use a simple process to draw the scene. Graphics engines developed for games tend to develop their own scene graphs suited for their specific requirements. There are already some well recognized scene graph based graphics toolkits developed using OpenGL available for application developers, namely Open Inventor™ by sgi [9] and OpenSceneGraph [10]. But any 3D graphics engine can utilize the scene graph management techniques we discussed to improve performance and flexibility.

REFERENCES