Understanding and Using Scene graphs

David Woolford
June 15th 03

UNDERSTANDING AND USING SCENE GRAPHS

CONTENTS

1. INTRODUCTION

This document explores the concept of the scene graph and how it is applied in modern 3D graphics. The concept of the scene graph is introduced in Section 2, along with techniques for constructing scene graphs. In Section 3 modern implementations of the scene graph are illustrated. The tools mentioned include Java3D, OpenGL Performer, VRML and X3D. Section 4 discusses the advantages that scene graphs bring in the context of 3D graphical applications. In Section 5 a more in depth discussion is presented which addresses the significant differences that exist between a selection of modern scene graph implementations (mentioned above). In Section 6 Binary Space Partitioning trees are introduced and described as an alternative scene graph tool in 3D applications.

To begin building scene graphs the reader should have a good understanding of geometry and linear transformations. Similarly, experience in using a low level graphics API such as OpenGL is a decided advantage.

2. WHAT IS A SCENE GRAPH?

2.1 Introduction

Scene graphs are conceptual tools used to represent virtual three-dimensional (3D) worlds in computer graphics applications [Reiners 2002, Walsh 2002]. A scene graph is a hierarchical structure containing nodes connected by edges. The nodes of the scene graph manage the data describing a virtual scene and the edges that connect the nodes describe the relationships that exist between them in a meaningful way. The nodes are ideally arranged in a hierarchical manner that corresponds spatially and semantically to the modelled object/world.

We can divide the nodes in a scene graph into three categories – the root node, the intermediate grouping nodes called internal or group nodes, and leaf nodes which are located at the terminus of a branch. The root node is the first node ie, all other are nodes are connected either directly or indirectly to it. Internal nodes can have a variety of properties, with the most common use being 3D transforms performing rotation, translation, scaling and shearing. Internal nodes describe the position and orientation, or *state*, of the 3D world. Leaf nodes contain geometric data (also audio data).

Furthermore, a node that is spawned from another node is called a *child* node and the node that spawned it is known the *parent* node. Every node has a parent node except the root node which has none. A parent node may spawn from it any number of child nodes. Leaf nodes cannot be parent nodes, they may only be child nodes.

Originally a scene graph would be represented by a tree structure which permits nodes to have only one parent but an important evolutionary mutant known as the Directed Acyclic Graph (DAG) allows nodes to have more than one parent. Hierarchical scene graphs can use either of these approaches but the Directed Acyclic Graph is the more predominantly employed method. Directed Acyclic Graphs and trees are compared in **Figure 2.1**.

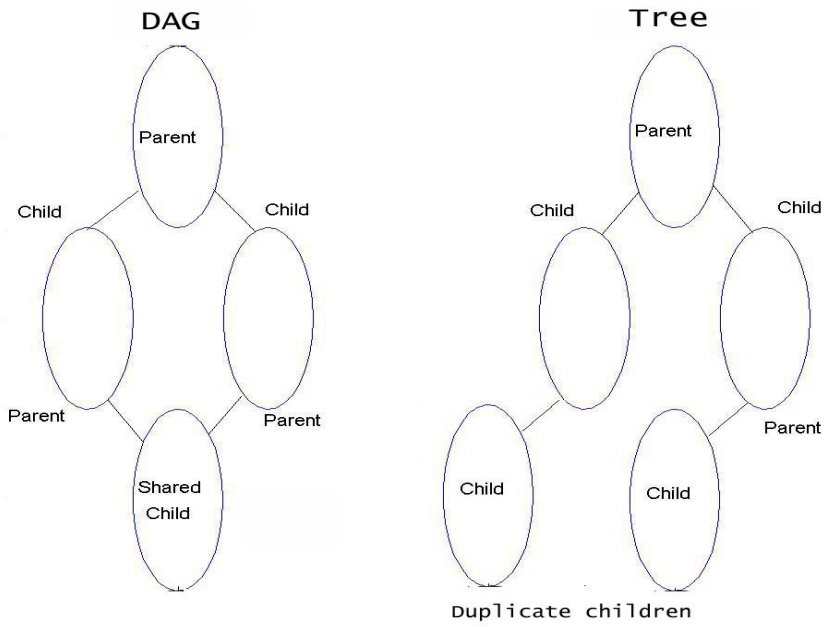

Figure 2.1 Comparing the structure of Directed Acyclic Graphs and trees.

2.2 Constructing a scene graph

We construct a scene graph by creating nodes and joining them together using edges. For instance let us model a house using the scene graph concept. A node that we conceptually label *House* might spawn from it four *Room* nodes and from each of these *Room* nodes we might spawn (say) numerous leaf nodes that describe the objects in each room. They do this by explicitly defining the geometries of each room. We construct the pseudo scene graph in **Figure 2.2**. Note how the edges connecting each of the nodes correspond to our intuitive notion of the relationships that exist between objects in the model.

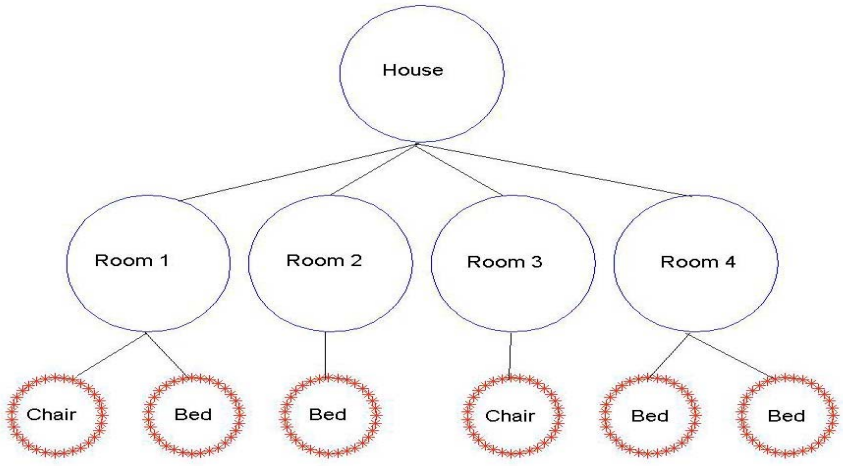

Figure 2.2 A well built pseudo scene graph of a house.

We say this model corresponds *spatially* with our modelled object because the house contains four rooms and each of these rooms is spatially separate from the other rooms. We say that this model corresponds *semantically* with the house because it is consistent with our human notion of the nature of objects and their constituent elements. Breaking objects down in a natural, hierarchical fashion is the essence of a scene graph.

We must be careful to avoid ambiguity. For instance in our house model we might be tempted to assemble the scene graph using a different strategy. We might instead notice that there are two chairs and fives beds in the house and so instead spawn two nodes from the *House* node namely, *Beds* and *Chairs*. Thus we have done away with our *Room* nodes. This idea is depicted in **Figure 2.3**.We could then say that our model is corresponding semantically with the house because the objects in the house fall into two categories, either bed or chair. But the geometric objects defined under the *Beds* and *Chairs* nodes are a jumbled set of spatially independent objects. This undesirable result is contrary to efficient usage of a scene graph, which will be discussed in more detail in Section 4. Thus we say that the ideal scene graph model corresponds both *spatially* and *semantically* with the modelled object.

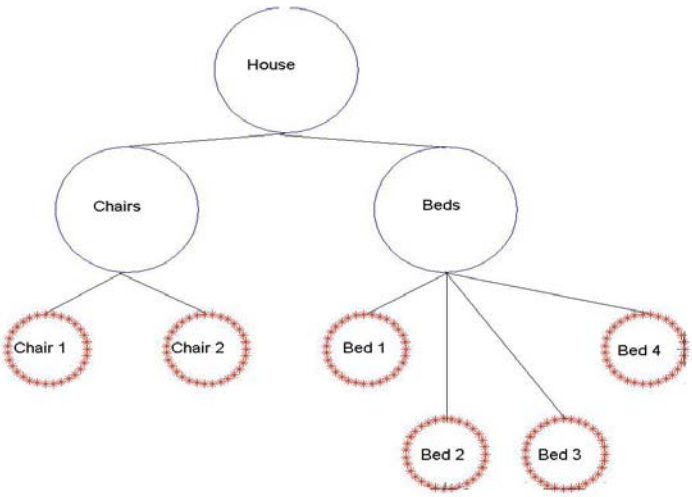

Figure 2.3 An alternative strategy that is contrary to efficient usage of a scene graph

2.3 Properties of a scene graph

Scene graphs implement a principle known as *state inheritance*. We have already mentioned that internal nodes convey the state of the system whereby state we mean the position and orientation of objects in the virtual environment (amongst other attributes). State inheritance is a property of scene graphs that dictates that each node inherits the state properties of all its ancestors, in a direct line to the root node.

Let us return to the house model and, for the sake of simplicity, let us construct the objects in each room relative to the origin. At each *Room* node we introduce a translation transformation that positions the objects of the room correctly in the house. Likewise, let's say that we want the house rotated to face a certain direction and so we specify a rotation transform at the *House* node. This is depicted in **Figure 1.4**. Because of state inheritance all the geometries identified at the leaf nodes will inherit the state properties of their ancestors and hence be positioned correctly. State inheritance is thus a useful tool that streamlines the process of constructing a 3D scene.

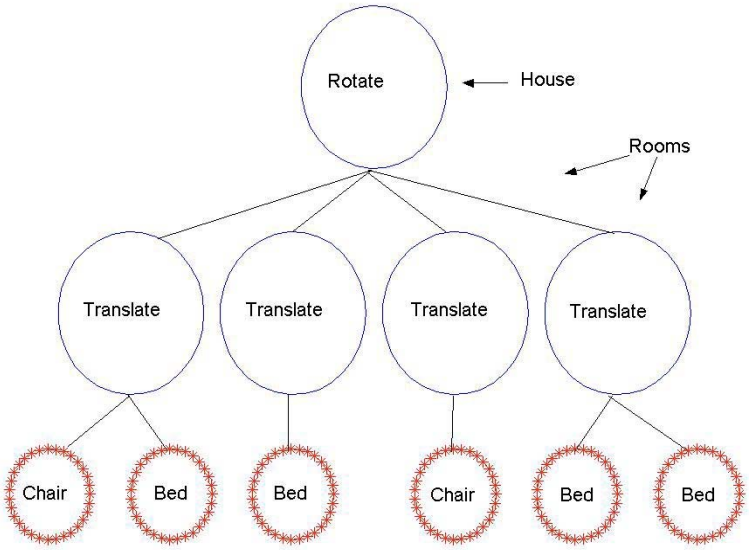

Figure 2.4 Using transforms to position the rooms and rotate the house.

3. BEGINNING TO CONSTRUCT SCENE GRAPHS USING AVAILABLE TOOLS

3.1 Introduction

As mentioned in the previous section, scene graphs are data storage tools that are used to describe 3D scenes. In the world of computer graphics there exists many scene graph implementations that build on the scene graph concept. They may come in the form of an Application Program Interface (API), designed to be integrated into standard programming languages such as C/C++ or Java. Examples of two such scene graph implementations are Java3D [The Java3D API Specification, 2000] and OpenGL Performer [Koh et al, 2000; Eckel 1997]. Alternatively scene graphs may come in the form of a text or binary file. Such files usually adopt a *markup* language format and are generally synonymous with web applications. Examples of two such implementations are the Virtual Reality Markup (or Modelling) Language (VRML) and Extensible 3D (X3D). VRML is an International Standard similar to HTML [Ames 1997; Carey et al 1997], whereas X3D works within the constructs of the XML file format [Blais et al 2001; Walsh et al 2001].

In this section, we will begin to construct scene graphs using the aforementioned tools. We will use a test scene (see **Figure 3.1**) to create small examples with each scene graph tool. Later, in Section 5, we will investigate more thoroughly the differences and similarities of each of the example scene graph implementations.

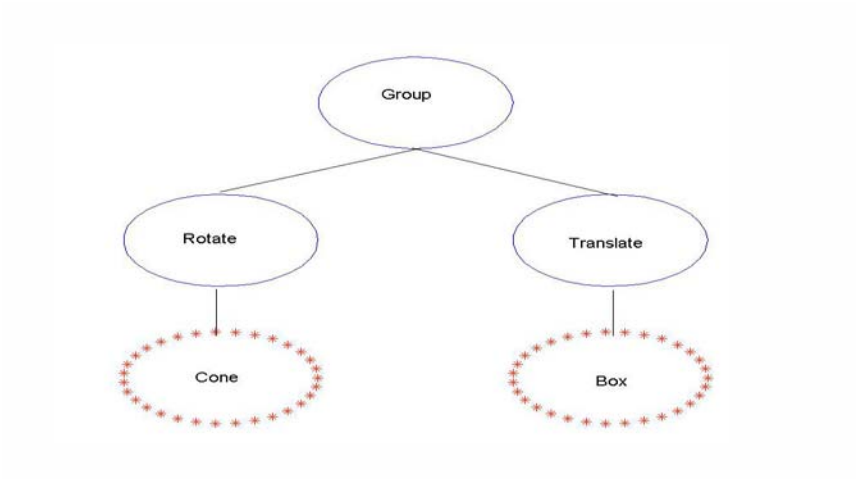

Figure 3.1 The scene graph which will be the theme of this section

3.2 VRML and X3D

VRML and X3D files can be created and edited in any text editor. They chiefly address 3D graphics on the World Wide Web. Both implementations were developed by the Web3D Consortium, an international authority in the field of 3D Web applications (http://www.web3d.org).

VRML and X3D are often viewed in a Web browser with an appropriate plug-in or viewer. These viewers construct a programmatic scene graph using the text file as a blueprint. This programmatic scene graph could possibly be constructed using a Java3D or OpenGL Performer. This characteristic of file format scene graphs will be discussed in more detail in Section 5. Finding such a viewer is a simple case of searching the Web with your favourite search engine. Once you have the plug-in you are ready to view VRML or X3D files. Some examples of such viewers are Cortona and Cosmoplayer and they are available on the World Wide Web.

3.2.1 VRML 97

VRML 2.0 (or VRML 97) is the successor to VRML 1.0. VRML uses the Directed Acyclic Model of scene graph. VRML is similar to HTML in that users can employ traditional cut and paste methods to build files.

There is but one mandatory element in a VRML file - all VRML files must begin with the following line; **#VRML V2.0 utf8.** Once specified as a VRML file in this way, we are free to use a host of simple tools that VRML makes available. We have already seen that scene graphs are technically anchored to a root node. VRML however does not require any specific reference to a root node and indeed, it has no root node syntax. The first node in a VRML is ambiguous, it can be one of the many group or leaf nodes that are available with VRML.

As we have mentioned previously scene graphs have group nodes and leaf nodes. Some VRML group nodes are illustrated in **Table 3.1**

Group Node	Semantics
Group	A generic grouping node
Transform	A geometry transform node
DirectionLight	A light source position infinitely far away, facing a specific direction
PointLight	A positioned light that emanates light in all directions
SpotLight	A positioned light that cast light in a specific direction and area
Inline	A node that reads a file and attaches it to the scene graph

Table 3.1 VRML group node syntax and semantics.

Similarly, there are a variety of leaf nodes available. They are illustrated in **Table 3.2**.

Leaf Node	Semantics
Shape	A node containing geometry
Background	Specifies a background
Sound	A node containing sound

Table 3.2 VRML leaf node syntax and semantics.

VRML supports simple graphics primitives such as spheres, cones, cubes, and cylinders. Such geometries are created by making the appropriate call within the **Shape** node. At this point we must address the VRML element **Children**. We have already described some generic group and leaf nodes, however the **Children** element falls into neither of these two categories. The **Children** elements are a linking mechanism that defines the parent-child relationship. For an illustration of a simple VRML file, see **Figure 3.2**. For the corresponding scene graph interpretation of this file see **Figure 3.1**.

```
#VRML V2.0 utf8
# This is an example of a simple VRML file
# It draws a cone rotated about the z-axis, centred at the origin
# and a box translated a distance 8 in the z-direction

Group {
  Children [
    Transform {
      rotation 0 0 1 1.57 # 180 degrees about z-axis
      Children [
        Shape {
          appearance Appearance {
            material Material {
              diffuseColor 0 1 0 # RGB scheme (Green)
            }
          }
          geometry  Cone {}
          }
      ]
    }
    Transform {
      translation 0 0 8 # move 8 units in the z-direction
      Children [
        Shape {
          appearance Appearance {
            material Material {
              diffuseColor 1 0 0 # RGB scheme (Red)
            }
          }
          geometry  box {}
        }
      ]
    }
   ]
}
```

Figure 3.2 An example of a VRML file.

Note how the **Transform** node facilitates the implementation of geometric transforms. VRML achieves this by allocating user modifiable fields to its nodes. If these fields are left unchanged then default settings are loaded. Similarly, the **Shape** node has fields associated with, namely *appearance* and *material*. For more information regarding the VRML file format see [Walsh et al 2001, Ames et al 1997].

3.2.2 X3D

X3D may be conceived as the successor to VRML. X3D works within the XML file format. The XML file format is an infant Web language and data storage tool that is significantly different from its predecessor HTML [Ladd E et al 2001]. XML allows you to define your own syntax and specify how it is displayed. This means that any XML document can be presented in an arbitrary number of ways. X3D embraces the idea of ambiguous terminology allowing itself to be a highly versatile development tool that be can be fine-tuned to suit individual needs [Walsh et al, 2001 934-1033]. X3D will support the Directed Acyclic model of the scene graph.

At the time of writing X3D is still under development. It will support VRML file structure so that any VRML file will successfully run within the frameworks of an X3D file. More significantly, it will probably disclude geometric primitives such as spheres, cubes, cylinders, and cones (as well as a host of other VRML nodes) and instead support only basic geometry sets and a refined list of group nodes. This will not limit X3D, for such graphics primitives can be made readily available by the inclusion of the appropriate interpreters in the XML file structure (probably developed as an optional add-on referenced in the first few lines of the file structure). The power of such an approach is that any number of interpreters can be developed to suit specific needs.

The Web3D Consortium currently has a test toolkit intended for integration with X3D called Xj3D [Blais et al, 2001]. Xj3D is an interpreting tool that supports file formats similar to VRML, amongst other things. Below is an example of what will probably be a typical X3D file. Note the similarity between the X3D and VRML file format and syntax. This X3D file describes the same 3D scene described in **Figures 3.1** and **3.2**.

```
<?xml version="1.0" encoding="UTF-8"?>

<X3D>
  <Scene> #The root node
    <Group>
      <Transform rotation = "0 0 1 1.57">
        <Shape>
          <Cone/>
          <Appearance>
            <Material diffuseColor = '0 1 0'/>
          </Appearance>
        </Shape>
      </Transform>
      <Transform translation = "0 0 8">
        <Shape>
          <box/>
          <Appearance>
              <Material diffuseColor = '1 0 0'/>
          </Appearance>
        </Shape>
      </Transform>
    </Group>
  </Scene>
</X3D>
```

Figure 3.3 A example of a hypothetical X3D file

Notice how no child-node syntax is used. Instead the parent child relationship is inferred from the file lay out. The file shown above is using the concept of user-modifiable fields in a similar fashion to VRML.

3.3 Java3D

The Java3D API is referenced in any Java program by the inclusion of the appropriate packages. Every Java3D program is constructed using a combination of the following classes (**Table 3.3**).

Class	Description
javax.media.j3D	Essential classes that make up Java3D. Required in every Java3D program.
javax.vecmath	A collection of core math classes. Required in any Java3D program
com.sun.j3D	Java3D 'convenience' classes that provide geometry primitives and loaders. Optional.

Table 3.3 The Java3D classes and their uses.

Java3D supports the Directed Acyclic model of the scene graph. At the top level of every Java3D scene graph is a *Virtual Universe* attached to which is the *Locale* object. Together they comprise what is referred to as scene graph superstructure in Java3D literature [The Java3D API Specification, 2000 Walsh 2002, Walsh et al 2001]. *Branch group* nodes are attached to the *Locale* and acts as root nodes. In the initial stages of scene graph specification, nodes are first attached to the *Branch group* nodes. *Branch group* nodes are initiated by the call to Java3D's **BranchGroup** object. The top-level structure of the Java3D scene graph is depicted in **Figure 3.4**.

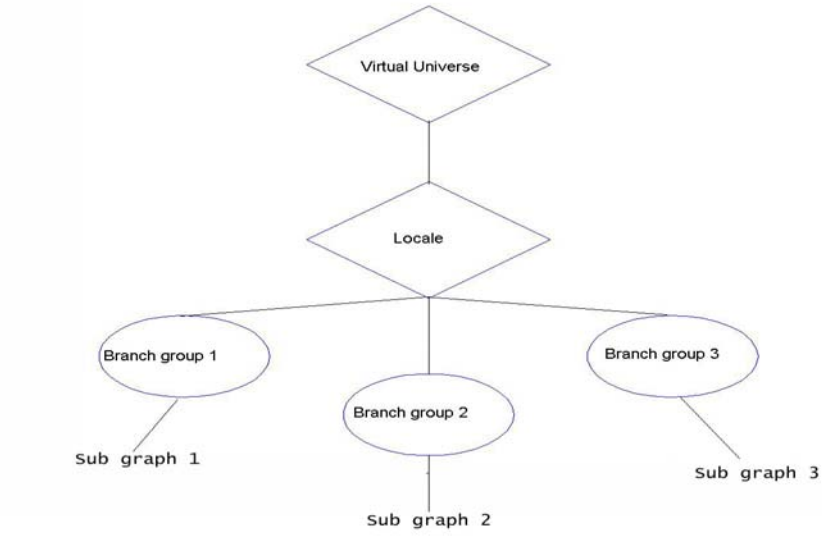

Figure 3.4 Top level structure of the Java3D API

Java3D has a selection of predefined group nodes. They are shown in **Table 3.4**.

Group Node	Semantics
BranchGroup	Acts a as a root node for a sub graph
OrderedGroup	A node that specifies the order in which its children are drawn
SharedGroup	A node that can be shared among sub graphs
Switch	A node that can choose which of its children will be rendered
TransformGroup	A node that may confer spatial transformations upon its children

Table 3.4 Java3D group nodes

There are also available generic leaf nodes, shown in **Table 3.5**.

Leaf Node	Semantics
Shape3D	A node that contains geometry objects
Sound	A node for sound sources
Link	A node referencing another leaf node in accordance with the DAG model

Table 3.5 Java3D leaf nodes

The core Java3D doesn't come with a suite of primitive shapes, however an add-on defines some basic primitives for implementation. This is supplied by Sun's (**web page**) utility package *com.sun.j3d.utils.geometry*. It defines *Box*, *Sphere*, *Cylinder*, and *Cone* classes that create the associated geometric primitives with specified shape and size. Java3D programmers have the option of creating elective shapes using points, lines, and polygons. There are also available a host of loaders which import other scene graph file formats (such as VRML). Below is an example of a Java3D file. It draws the same scene previously described in **Figure 3.1**.

```
Import java.applet.Applet;
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.utils.geometry.Box;
import com.sun.utils.geometry.Cone;
import com.sun.j3D.utils.applet.MainFrame;

public class HelloUniverse extends Applet {
   public Branchgroup createSceneGraph () {
        // Create the root branch group
        BranchGroup newRoot = new BranchGroup();

        // Create group nodes with transformational properties and add them to the root
        // node
        TransformGroup newTrans_1 = new TransformGroup();
        newRoot.addChild(new_Trans1);
        TransformGroup newTrans_2 = new TransformGroup();
        newRoot.addChild(new_Trans1);

        // Add the geometric primitives
        newTrans_1.addChild(new Cone());
               newTrans_2.addChild(new Box());

        // We must add transformational properties to the Transform nodes
        Transform3D rotate_example = new Transform3D();
        // Rotate around the z-axis 180 degrees
        rotate_example.setRotation(new Vector3D(0, 0, 1.57));
       newTrans_3.setTransform(rotate_example);


        Transform3D translate_example = new Transform3D();
        // Translate in the z-direction
        translate_example.setTranslation(new Vector3f(0,0,8);
        newTrans_2.setTransform(translate_example);

        newRoot.compile();

        return newRoot();
   }
}
```

Figure 3.5 An example of a Java3D file

As you can see, the process of creating a scene graph is now a more abstract process. The structure of the scene graph is no longer being conveyed *visually* as it did in VRML/X3D examples. Instead, a number of function calls are being utilised to construct the scene.

Note also how geometric transforms are first created and then applied to the **TransformGroup** node. This differs significantly from the previous implementations we have come across, which used fields instead of function calling.

3.4 OpenGL Performer

OpenGL Performer (Performer) is an advanced API. Performer is made available by the inclusion of the Performer libraries (shown below) into the C/C++ programming framework. Performer supports the Direct Acyclic model of the scene graph.

Library	Description
pf.h	Main Performer library. Contains essential libraries libpf and libpr
pfdu.h	Library of scene and geometry tools
pfutil.h	Utility functions library
pfbd.h	A library containing load, convert and store routines for different file formats
pfui.h	User interfaces library

Table 3.6 The performer libraries

A complete Performer scene graph is one that is rooted by a **pfScene** node. Performer supports the Directed Acyclic model of the scene graph. As before creating a scene graph is an iterative process of adding child nodes, of which at least one is a child of the root node. Performer comes with a generic list of group nodes that may be used to assemble a scene graph at the programmers discretion.

Group Node	Semantics
PfGroup	A generic empty group node
PfScene	The root node
PfSCS	Stores a static coordinate system in which to place its children
PfDCS	Stores a dynamic coordinate system in which to place its children
PfSwitch	A node that directs traversals to all, none or just one of its children
PfPartition	A node that optimises very flat terrains

Table 3.7 Some Performer Group nodes

Similarly, there are preset leaf nodes available (**Table 3.8**).

Leaf node	Semantics
PfGeode	A node that encapsulates geometry
PfBillboard	Contains a slice of geometry that always faces the viewer
PfText	A node that displays 3D text

Table 3.8 Some Performer leaf nodes

Adding geometry to the scene graph is generally done using lines, points and polygons. Performer uses **pfGeoSets** to create these graphics primitives. Typically one creates a shape (or invokes a generic shape) and attaches it to the leaf node **pfGeode**. A variety of generic

3D shapes is available by the inclusion of the *libpfdu* library. Shapes available include spheres, cones, cubes, cylinders and pyramids.

There are other coding semantics associated with Performer that need to be addressed before we can start programming with the toolkit. Before building the scene graph the first thing one needs is a **pfChannel**, which is equivalent to a camera moving through the scene. Secondly we need a **pfPipe**, which renders the data to a window. Finally, there needs to be window that the scene is drawn to. This will be of type **pfPipeWindow**. An example of a Performer file that displays our theme scene is given below in **Figure 3.6**.

Take note of how the process of adding geometries to the scene graph is more complicated than any of the three previously mentioned tools. First, the geometric primitive is created using **pfGeoSet**. Second, a leaf node is created using **pfGeode**. Third, the geometric object is added to the lead node using **pfAddGSet**. Finally, the leaf node is attached to the scene graph using **pfAddChild**.

For more detail on the Performer API see [Eckel G ,1997; Koh et al 2000].

```
#include <Performer/pf.h>     // Libraries are in a Performer folder
#include <Performer/pfdu.h>

main {int argc, int *argv[])
{
   pfInit();
   //pfConfig must go at before creating a pipe
   pfConfig();

   //Create a pipe and configure it
   pfPipe *pipe = pfGetPipe(0);

   //Get a pipe window and associate it with the pipe
   pfPipeWindow *pWin = pfNewPWin(pipe);

   //Create a channel and connect it to the pipe
   pfChannel *chan = pfNewChan(pipe);

   //Initialise the root node
   pfScene *root_node = new pfNewScene();

   // Initialise transformational group nodes and add them to the root node
   pfDCS *dcs_rotate = new pfDCSRot(0, 0, 1.57);
   pfAddChild(root_node, dcs_rotate);
   pfDCS *dcs_transl = new pfDCSTrans(0, 0, 8);
   pfAddChild(root_node, dcs_transl);

   // Create the geometries and add them to a pfGeode
   // The cone
   pfGeoSet *new_cone = new Cone(100, arena);
   // A cone made up of 100 triangular polygons
   pfGeode *shape_cone = new pfGeode();
   pfAddGSet(shape_cone, new_cone);
   pfAddChild(dcs_rotate, shape_cone);
   // The box
   pfGeoSet *new_box  = new Cube(arena);
   pfGeode *shape_box = new pfGeode();
   pfAddGSet(shape_box, new_box);
   pfAddChild(dcs_transl, shape_box);
```

```
   //Light the scene
   pfAddChild(root_node, new pfLightSource);

   //Associate the root node with the channel
   setScene(chan, root_node);
}
```

Figure 3.6 A sample Performer file that creates a scene graph composed of a cone and a box.

3.5 Summary

In this section it has been shown how one might begin to construct scene graphs using some of the tools currently available. We have seen how scene graph implementations use group and leaf nodes, and we have explored the syntax of each of the scene graph tools by creating simple programs. We have also noted that the file formats rely on a viewer to construct the associated programmatic scene graph, and that this programmatic scene graph is actually constructed using an API such as Java3D or Performer.

4. Advantages of the scene graph

4.1 Scene graphs are high-level graphics tools

The scene graph came into existence as an implementation tool for handling large 3D graphics applications. So what advantages do scene graphs bring? Firstly, they streamline the scene design process. Scene graphs are high-level graphics tools that avoid the procedural aspects of low-level graphics tools such as OpenGL and DirectX [Walsh 2002].

Scene graphs are data storage structures that are managed and created using a scene graph modelling language. The scene graph modelling language stores the data, updates the data and tells a low level API (such as OpenGL) to render the scene. Thus rendering is abstracted out of the scene graph programming model. When using a scene graph tool, all low-level concerns are taken care of by the underlying low level API.

An example of such a programming alleviation concerns state inheritance. It is already been mentioned that state inheritance is a property employed by scene graphs. However, in the formulation stages of the scene graph the state properties of a node's ancestors are *not* referenced explicitly or stored as a property of the descendant node. State inheritance is a property exploited during the rendering phase; as the renderer traverses the scene graph from root to leaf, transformational affects are permuted from parent to. Transformational affects are typically accrued in a transform matrix that is used to position the vertices described in leaf nodes using straightforward matrix-vector operations. In other words, scene graphs can be thought of as storage structures that are interpreted and rendered using low-level graphics APIs.

State inheritance is an ideal tool for animation. The introduction of time-dependent transformational effects that animate entire sub graphs is achieved via the inclusion of the appropriate command at the ancestor node, from which the object originates. This framework provides an efficient base for building animated scenes.

4.2 Culling using bounding volumes

Bounding volumes are calculated and stored at each node of the scene graph. The bounding volume encapsulates each geometric object defined in a leaf node that is descendant from the node for which the bounding volume is associated. Scene graphs use the concept of bounding volumes to significantly improve the efficiency of the graphics pipeline. [Eberly, 2000] See **Figure 4.1** for an illustration of a graphics pipeline in the context of a scene graph application. As opposed to state inheritance, which can be thought as a process that proceeds from root to leaf, bounding volumes are calculated in the reverse order – from leaf to root.

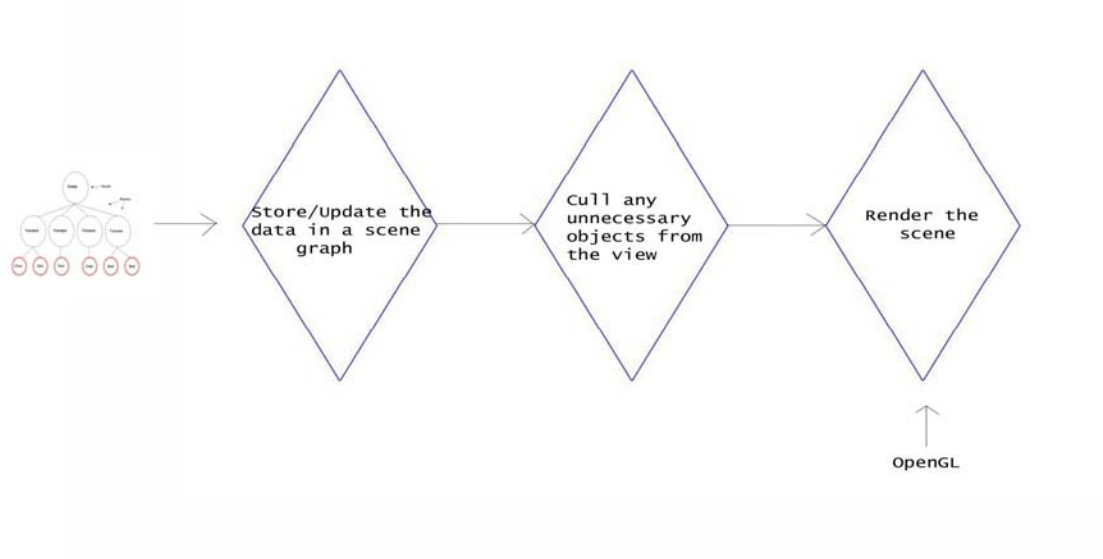

Figure 4.1 A simplified scene graph graphics pipeline

The application first calculates the bounding volumes of the leaf nodes and stores them. The bounding volumes are then merged (typically) two at a time until an entire set of children node-bounding volumes are merged. In this way, the bounding volume of the parent node is established. Consider again the house model that was constructed in section one. To establishing the bounding volume of each node in the scene graph, bounding volumes are first calculated at the leaf nodes. We establish the bounding volume for each object (leaf node) in each room and merge them until there are none left to merge; when this happens the bounding volumes of each room are known. The bounding volumes of the *Room* nodes are then merged to produce the bounding volume associated with the *House* node. Bounding volumes can vary but are typically oriented boxes, spheres, ellipsoids and lozenges.

Bounding volumes facilitate the introduction of an efficient culling stage into the graphics pipeline. In computer graphics culling usually refers to a process of disregarding objects before rendering occurs, because they are not visible. Culling methods are often based on the idea of testing vertices or objects for presence within the viewing volume. The viewing volume is the section of model space that is to be displayed. Objects outside of the viewing volume will not be seen and hence should not be rendered. The naïve approach of testing every vertex in the scene for presence within the viewing volume can be an inefficient process that impacts detrimentally on frame refresh rates. Using the idea of bounding volumes scene graphs help to resolve this problem.

Scene graph programming models first test bounding volumes for presence with the viewing volume. Because the bounding volume of a particular node encapsulates all the geometries described by all the nodes that descend from it, if the bounding volume is outside the viewing volume then the entire sub graph is disregarded from the rendering stage (culled). In scene graph terminology this is often referred to as *pruning*.

The culling process is a top down traversal of the scene graph that starts at the root node. At each node there are a finite number of logical possibilities, and they are outlined as follows.

1. The bounding volume is completely outside the viewing volume, in which case the branch is pruned and has no further consideration in the drawing process.
2. The bounding volume is completely within the viewing volume. In which case the entire sub graph beginning at the node in question is passed to the next stage in the graphics pipeline.
3. The bounding volume is partially within the viewing volume. In this case step down the scene graph to the next layer of nodes and query their bounding volumes – one of these four scenarios will be true.
4. If the bounding volume is a leaf node and it is partially within the viewing volume than it is handed to the next stage of the graphics pipeline

Clearly then every element of the scene graph will be accounted for by this process. Leaf nodes that are partially within the viewing volume will be clipped accordingly. The essential result from this process is the potential to dramatically improve the speed of the graphics pipeline. Indeed, in large 3D scenes containing animation and a dynamic viewpoint the efficiency of that bounding volumes deliver to the cull process is vital.

4.3 Collision detection

Intersection (collision) methods use bounding volumes in similar ways to those already mentioned. Querying a scene graph structure for intersection with an object can be achieved via a top-down traversal starting at the root node. Assuming our object has a bounding volume (perhaps it is an animated object within the scene graph itself) we can employ a strategy similar to the culling scheme mentioned previously.

1. If the bounding volume of the object does not intersect the bounding volume of the node than disregard the sub graph that originates at the node. Collision can not occur.
2. If the bounding volume of the object intersects the bounding volume of the node (which is not a leaf node), query the child nodes for collision.
3. If the bounding volume of the objects intersects the bounding volume of a leaf node defining geometry then a collision has occurred.

The scene graph acts as a querying structure that supports both view volume culling and collision detection. As in the case of the culling process, the scene graph brings with it the potential to increase the efficiency of the collision detection process. Without the scene graph collision detection would be an exhaustive process of $O(n^2)$, where n represents the number of autonomous objects in the scene.

The potential contribution that bounding volumes can make to the overall improvement of refresh rates cannot be underestimated. This is precisely why we say that the ideal scene graph representation of a scene corresponds spatially and semantically to the modelled world. "The efficiency of both (culling and intersecting) traversals is largely dependent on the depth and balance of the scene graph hierarchy" [Rohlf et al 1997].

4.4 Parallel processing

Scene graphs are complimented by parallel processing techniques and so in cases where scene complexity is too great for a single processor, parallel opportunities are available. For more information see [Rohlf et al, Bethel].

5. 5. AN IN DEPTH LOOK AT SCENE GRAPH APPLICATIONS

5.1 Introduction

Thus far we have defined a scene graph, overviewed some core scene graph implementations, and described the issues that a scene graph addresses. In this section we aim to compare and contrast VRML, X3D, Performer and Java3D

5.2 Discussion

As we have observed in section two, each of the scene graph applications build scene graphs in similar ways. They all support inclusion of geometries, sound and light sources. In addition, the scene graph is assembled using a range of group nodes that are similar in each implementation. The process of adding child nodes to parent nodes is similar even though the syntax is different. Performer and Java3D use the functions **pfAddChild** and **addChild** respectively, whilst VRML uses the element syntax **children.** X3D has no specific child-node syntax, but rather infers the parent child relationship from the file lay out.

We can broadly partition our scene graph implementations into two categories – the APIs (Java3D and Performer) and the file formats (X3D and VRML). The APIs are generic tools intended for use in a wide range of fields whereas the file formats are application specific- they are designed chiefly as data storage structures that can be passed easily and quickly across the Web.

The problem if displaying 3D scene on the Web was solved when developers realised that they could pass 3D data to machines in text format and let the machine take care of the graphics primitives and rendering. VRML was designed to be simple, robust and yet at the same time capable of producing attractive looking scenes that interact with the user. The process of getting VRML/X3D working on your machine starts by installing a viewer on your system. VRML viewers interpret the VRML file and construct a scene graph for displaying purposes.

One of the most fundamental differences of our two categories is that VRML and X3D are actually interpreted and rendered by an API such as Performer or Java3D. VRML./X3D viewers implement an API that constructs a scene graph representation mimicking the

specifications in the text file. File formats are way of communicating to an API, which constructs the programmatic scene graph representation using its own semantics

Thus, we can pass a viewer a simple and potentially very small text file across a computer network that tells the viewer what to draw. The viewer utilises and API such as Java3D or Performer to construct and render the 3D scene. Because VRML is an international standard, all its nodes and capabilities are predefined and viewers can account for all possibilities and implementations of the scene graph. In a similar way viewers will be able to accommodate for any syntax referenced in an X3D file by referencing the appropriate X3D file interpreter.

We have already seen that VRML and X3D are very similar in a number of ways. They are both file formats that can be created using basic text editors. They do not require advanced knowledge of a programming language. Yet they are also different in a number of ways. X3D is implemented in the framework of the Extensible Markup Language (XML). XML may be conceived as the successor to HyperText Markup Language (HTML) [Ladd et al 2001]. HTML has a predefined list of available elements and every HTML document (technically) requires the inclusion of mandatory elements such as the beginning <html> and ending </html> elements. In addition the user has a host of tools for creating Web documents and these deliver control of test styles and the ability to create tables, to name a few. XML is significantly different in that is has no predefined elements. XML is a language *without* words. The Web page designer creates an XML document using his own syntax and then applies a *transformation sheet* that specifies how the XML document is to be presented (if at all). XML files can also act as data storage files that can be passed between businesses and institutions using their own syntax.

X3D embraces the ideas of XML. For instance, VRML can be expressed in XML documents using X3D [Blais et al 2001]. X3D addresses some of the limitations of VRML. VRML files can be embedded into HTML documents, but the embedding process requires that VRML worlds be separate entities in the Web page. Typical embedded VRML is isolated from the rest of the page, requiring its own window and its own navigational capabilities. X3D has the added advantage of giving 3D objects the capability to interact fluidly with other (2D) objects in the web page. 3D objects can be integrated as part of the background or even as 3D interacting elements using X3D.

Java3D and IRIS Performer are available for use as soon as the appropriate classes/libraries have been included in the file header. Java3D and Performer are flexible in that they have a

wide range of application. This is derived from the fact that they are both APIs designed with no explicit purpose other than to act as generic scene graph constructors and renderers. Because Java3D and Performer are so tightly coupled with their fostering programming languages they require a higher level of programming sophistication, as opposed to VRML and X3D. This brings the potential to include the available tools that already exist for each language. For instance a programmer can use any other class/library for either language whilst simultaneously creating a scene graph application by simply referencing the appropriate classes/libraries.

Along with the capability to build simple scene graph structures, Performer supports frame rate control, management of system stress and load, multiprocessing, and facilitates a dynamic scene graph structure that can be queried and altered in real time. In a similar way Java3D also supports advanced features.

5.3 Summary

As you can see, scene graph APIs and scene graph file formats are two entirely different realisations of the abstract scene graph concept. Scene graph APIs are intricate graphics tools that are used to create 3D programs. They deliver greater control over the final product. A file format language such as VRML is limited in the context scene graph APIs. It does not deliver any extensible capabilities, it is a strict language with little room for manoeuvre. Nevertheless it was designed to be an International Standard Web application, similar in many regards to HTML, and shows no signs of becoming redundant. On the contrary, VRML has been very successful in initiating web 3D. The next evolutionary step is X3D which is designed to improve on the drawbacks of VRML. Although X3D will be a powerful tool for creating 3D web content it will still merely be a method of communicating to scene graph APIs such as Java3D and Performer.

6. BINARY SPACE PARTITIONING TREES.

6.1 Introduction

Scene graphs and Binary Space Partitioning trees are tools used chiefly in 3D graphics applications. They are both storage structures for representing 3D-scene geometry. BSP trees are just one method of implementing a scene graph. Whereas scene graphs are excellent modeling tools that work most effectively for environments containing abundant amounts of animated objects, BSP trees are more commonly applied in game development.[Naylor 1996, 2000]. BSP trees were implemented in the first wave of immersive 3D game engines such as *Doom* and *Quake*. Their use is still widespread and fundamental to most modern game engines [Hadwinger, 2003].

6.2 Building a BSP tree

To understand the concept of a BSP tree, let us first consider a scene consisting of two-dimensional objects partitioned by lines. The following discussion is in reference to **Figs 6.1** and **6.2**.

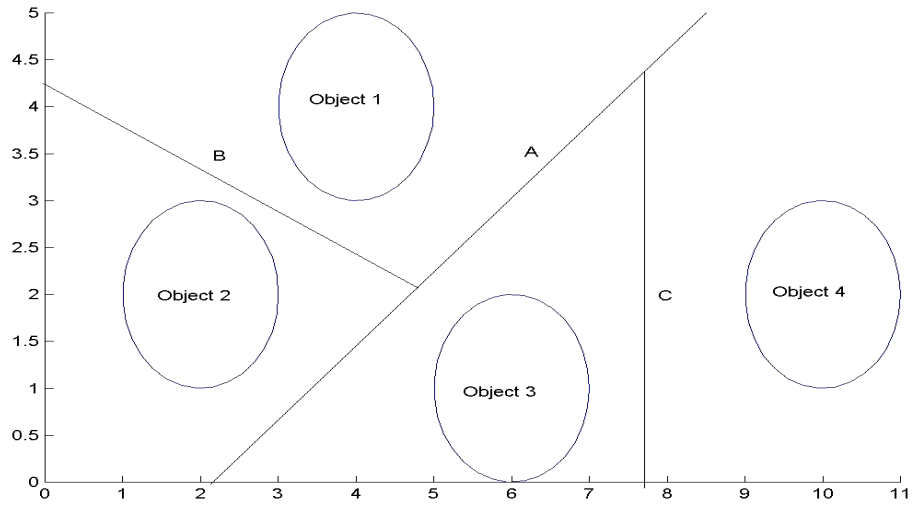

Figure 6.1 Spatially partitioning a scene

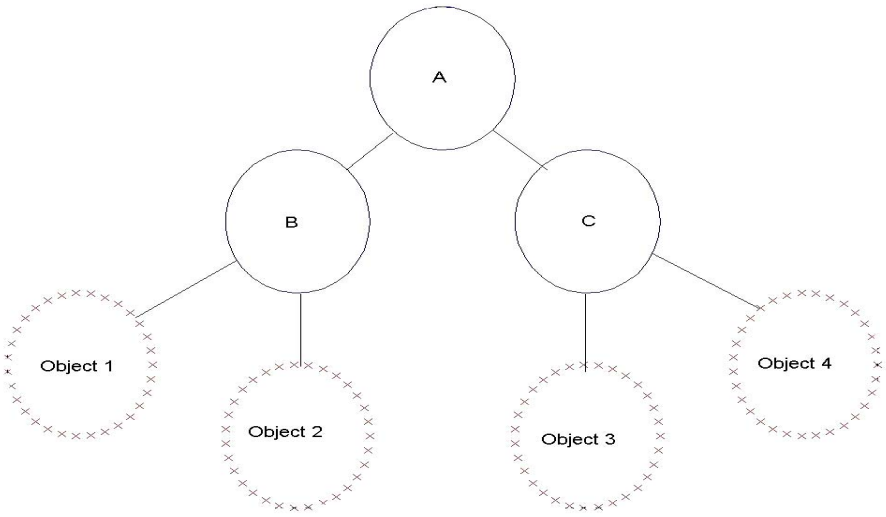

Figure 6.2 the partitioning tree describing the scene in Figure 6.1

The goal of partitioning is the representation of distinct regions in space containing *one* object only. First, we partition the (binary) space with partitioning lines as demonstrated in **Fig 6.1**. The second step involves recording this process in a tree like structure as in **Fig 6.2**. We can revive the concepts of nodes and leaves from the previous section excepts in this case a *node represents a partitioning line and a leaf represent an object*. Proceeding to the left or right of a node is logically equivalent to designating coordinate vectors of all descendants as either greater than or less than the line defined by B. Thus if we traverse a BSP tree from root object, we will have accrued sufficient information to define a bounding volume that encapsulates one object in the scene.

A graphical representation of the accruing of logical data to define specific boundaries is depicted in **Figs 6.3** and **6.4**. Assume that each line is a necessary addition that partitions the objects (not displayed) in a meaningful way and that we are interested in finding logical bounds that define the highlighted areas

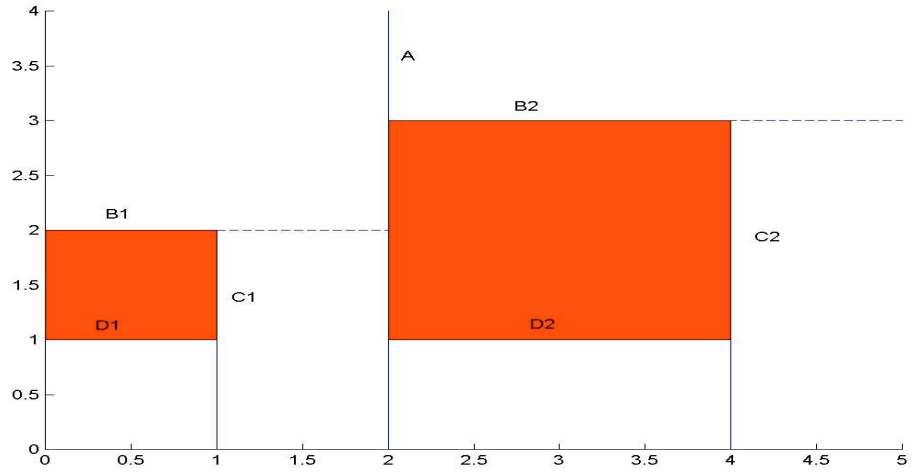

Figure 6.3 A partitioned 2D space

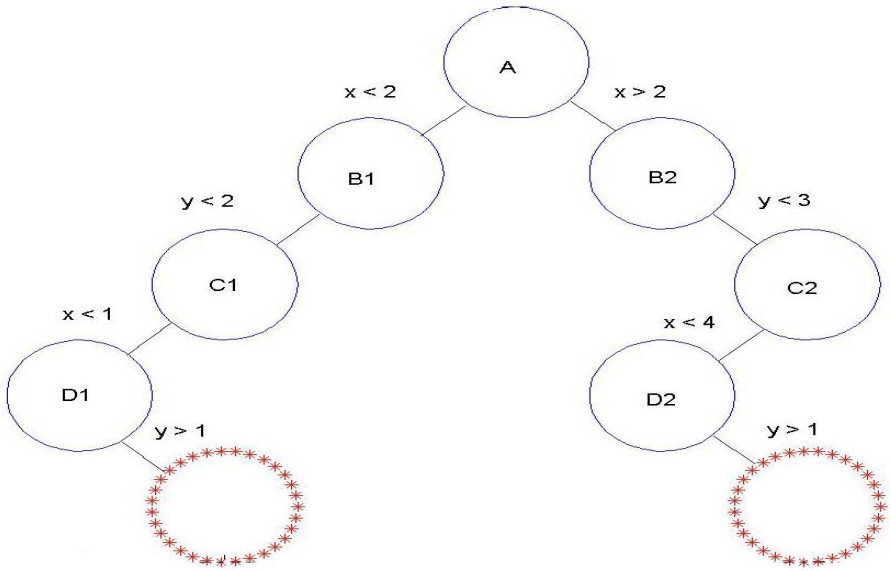

Figure 6.4 The partitioning tree used to describe the regions in Figure 6.3

We construct the Partitioning Tree of **Fig 6.3** in **Fig 6.4**. We may now proceed to use the tree to obtain a logical bounding volume. For example the smaller of the two red regions exists for x belonging to the interval [0 1] and y belonging to [1 2]. This is assuming that x is always greater than or equal to zero. The larger of the two spaces is defined by the bounds x in [2 4] and y in [1 3]. In both cases traversing the Partitioning Tree from the first node to the leaf

node containing the object attains the bounding volume. The bounding volume is attenuated at each step.

6.3 Binary Space Partitioning Tree guidelines

To extend the problem into n dimensions we introduce the terminology of hyperplanes and use it to denote a partitioning object in any dimension. Hyperplanes thus in 1D are points, in 2D are lines and in 3D are planes. Hyperplanes partition the binary space into logical sub compartments.

There are simple rules that govern the implementation of Partitioning Trees. The primary one being that all created bounding volumes must be convex (ie having no internal angles greater than 180 degrees). This is because BSP trees are used to speed up the visualisation process, a process that would be invalid if any bounding volumes were not convex. The second rule is that no geometric object (polyhedron, polygon, line, and point) can be on both sides of a hyperplane. Finally, no hyperplane may pass through another hyperplane.

6.4 Graphical efficiency

We have already seen that scene graphs use the concept of bounding volumes to quickly cull large portions of the 3D scene. BSP trees act in a different way. The first step of the visualisation process is to use the BSP tree as a *search structure* for locating the position of the viewer in the context of the 3D scene it represents. This process is quick and is visually demonstrated in **Fig 6.5**.

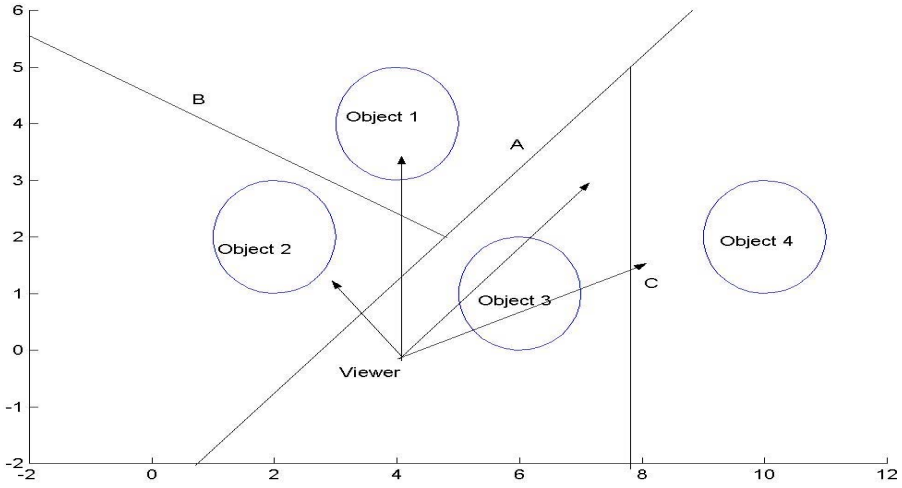

Figure 6.5 Locating the viewing position in a BSP tree

The next step of the viewing process is to assign visibility ordering to each of the objects in the scene. We do this with the so called 'Painter's Algorithm' in mind, a process of drawing polygons in a far to near order so that closer objects will paint themselves over objects located behind them. If we can assign a visibility ordering to every object then we have a way of rasterising the scene with an established, yet inefficient algorithm.

The BSP tree can be used to define the visibility ordering. For any given viewing position we first define on which side of the root hyperplane the viewer lies. Each object on the nearside of the hyperplane has assigned a higher priority than all objects on the other side of the hyperplane. We can apply the same operation to each hyperplane as they are encountered, always classifying those objects on the nearer side of the hyperplane a higher priority in the visibility ordering. Ordering the objects in the scene is equivalent to a single tree traversal of a BSP tree. Observe **Fig 6.6** which is the Partitioning Tree that describes the location of the viewer in **Fig 6.5**. Note the visibility ordering is displayed on the bottom of the figure.

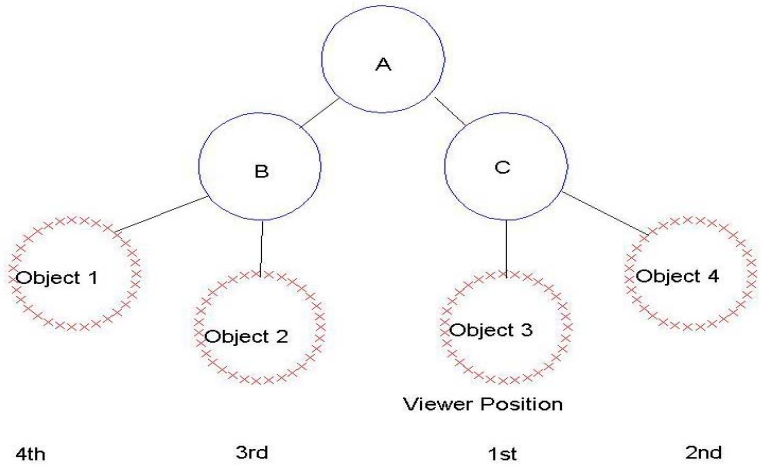

Figure 6.6 Visibility orderings.

We now have a method for efficiently employing the Painter's algorithm using only O(n) operations. However, we may improve efficiency by employing the back to front Painter's algorithm which paints the nearest polygons first and allows no pixel to change once it has been painted. Visibility ordering has supplied an efficient solution to the problem of nearest to furthest sorting. Note that these rasterising algorithms are not guaranteed to produce consistent results if bounding volumes are not convex. Visibility orderings are also useful in the context of ray tracing, beam tracing, shadow computations and radiosity.

BPS trees can speed up the display process by enabling the viewing application to disregard entire section of the scene in a similar way to scene graphs. If a nearby hyperplane is outside the viewing frustum then all objects on the far side of the hyperplane can be ignored.

Finally, collision testing can be done using a BSP tree. Using the BSP tree as a search structure, the region(s) of the binary space that the colliding object is within can be established. Only the objects cohabiting in the same region(s) that the candidate object is within are tested for collision. All other objects can be disregarded from the collision testing process.

6.5 Conclusion

This section has introduced and described the process of constructing Binary Space Partitioning trees. A BSP tree can be visualised as a graph containing nodes connected by edges, where the leaf nodes contain the geometries of the scene. Binary Space Partitioning

trees are similar to scene graphs in that they can be used for collision detection and for view volume culling, and they can be used to store the positional data of a scene. However, a BSP tree is not used to construct a scene, it is used to represent the space described by a scene in a tree structure that can be exploited as an efficient search structure. This tree structure can be used to improve the efficiency of the graphics pipeline and to provide visibility orderings, which are useful in terms of rendering scenes to display.

7. SUMMARY

This document has explored in detail the use of scene graphs in the field of 3D graphics. This part of the document aims to summarise the main points presented in the body of text in the preceding pages of this document.

In abstract form a scene graph is a collection of nodes connected by edges. A collection of nodes connected by edges and containing no cycles is known as a tree. Scene graphs can take the form of a tree, but most implementations of the scene graph employ the Directed Acyclic Graph (DAG) model. The DAG is just a variation of a graph that allows two (or more) nodes to share the same child node. This is because in 3D graphics the same objects are often used in different parts of the scene. By employing the DAG model, nodes can reference entire sub-trees of the scene graph and thus paint the same objects, but at different locations in the in scene.

Scene graphs are used to store the data describing a 3D scene. There are a variety of data types that are typically associated with 3D graphics. Linear transforms are used frequently to position and rotate the objects in the scene. Scene graphs facilitate the inclusion of linear transformations by allowing their association with certain internal nodes in the graph. Scene graphs typically store geometric objects at the terminus of a branch (or at a leaf node). A leaf node can spawn from it no other nodes. When traversing a scene graph from the root to leaf, the linear transformations associated with the internal nodes are accumulated. That is, a child node inherits the linear transformations associated with its parent node. Thus, the position of each geometric object (defined at the leaf nodes) is determined by the state properties of every ancestor node, in a direct line to the root node. This property of scene graphs is known as state inheritance.

A scene graph can be used to store the data describing a 3D scene in many ways however; there are ways of constructing a scene graph so that its structure can be used to increase the efficiency of graphics applications. The scene graph structure should correspond both spatially and semantically to the modeled object/world. Put another way, the scene graph should describe the modeled world in a natural, hierarchical fashion. This is largely due to the fact that bounding volumes can be used to speed up rendering processes.

Each node in the scene graph has associated with it a bounding volume that encapsulates all the objects defined by leaf nodes that descend from it. So the root node has associated with a bounding volume that encloses the entire scene, and a leaf node's bounding volume only encloses the object(s) defined therein. As the scene graph is traversed from root to the leaf, the bounding volumes encountered at the nodes are attenuated until finally they enclose a single object – which is equivalent to being at a leaf node and at the end of the traversal.

Bounding volumes can be queried in a number of ways. For instance, if the bounds of the viewing volume are known then it can be established whether or not a volume enclosing a set of objects is visible. If a bounding volume is not within the viewing volume then all the objects within that viewing volume do not have to be rendered. In graphical applications it is often the case that particular objects in a particular region/s of the scene cannot be seen; if the scene graph describes the scene in a natural and hierarchical fashion then a bounding volume enclosing these objects could exist at some node from which they descend. Using bounding volumes graphics applications can speed up the rendering process by efficiently disregarding objects that do not need to be rendered.

In a similar way bounding volumes can be used to speed up collision detection; instead of a viewing volume the application uses the bounding volume of a candidate object that is possibly colliding with objects in the scene. In similar way as above, the scene graph is traversed starting at the root node. Bounding volumes are be queried for possible intersection. The main result being that entire sections of the scene graph can be disregarded if bounding volumes do not intersect.

Armed with the theory behind scene graphs, it is a simple step to start constructing scene graph representations of 3D scenes using the available tools. VRML and X3D are file format scene graphs that can be constructed using text editors and can be viewed using Web-browser plug-ins. These plug-ins can be found on the World Wide Web. Constructing a scene graph using these tools is a relatively simple process, as long as some rules are adhered to. There is only a finite number nodes that can be implemented as internal nodes. These facilitate the inclusion of transforms and lights (amongst others). Only the leaf nodes can contain geometric data (and possible audio data), and the leaf nodes cannot have nodes spawning from them. Above all else, the scene graph must be constructed (using these tools) so that the parent-child relationships are defined. If the scene graph constructed using these tools were represented conceptually using nodes and edges (ie as a graph), then the shape of this graph would necessarily be a tree or a DAG (or a set of the same things). Not adhering to any of these rules will result in the plug-in failing to execute the scene.

These rules are similarly applied to Application Programming Interfaces (APIs) such as Java3D and OpenGL Performer, which construct the scene graph in similar ways. One major difference, though, is that the programming semantics of APIs are inherently more detailed. One has to be familiar with programming using a standard language such as C/C++ or Java. But there are advantages in using an API to construct a scene graph representation of a 3D scene. The APIs deliver greater control over the final product. This is because APIs typically give the programmer control over frame rates and rendering modes (amongst other things) that in turn allows the application to be fine-tuned to meet the specific needs.

We can see then, that there are basically two types of implementation of the scene graph. The first are file format scene graphs, and the second are the APIs, or programmatic scene graphs. At first glance they may seem unrelated, but the plug-ins that display file format scene graphs must first construct their own scene graph representation using a programmatic scene graph paradigm. The file format scene graphs are just a simplified means of communicating to an application that constructs its own programmatic scene graph. And this programmatic scene graph can be derived using API such as Java3D or Performer.

Finally, at the end of the document Binary Space Partitioning (BSP) trees were discussed. A BSP tree is just a different type of data storage structure. To understand a BSP tree it is necessary to understand concepts associated with spatial discretisation. The three-dimensional (binary) space defined by a scene is partitioned using a hyperplane. Two volumes result, which are in turn partitioned using another hyperplane. Every resulting volume is partitioned in a similar way and the process continues iteratively. The process stops when a unique volume containing no other object encloses every object. Constructing a BSP tree is the process of recording this spatial partitioning process in a tree structure. The nodes of this tree structure contain the equations defining the hyperplanes in the partitioned space. Each of the nodes spawns from it at most two edges. Following the BSP down either of these edges corresponds to being on one of the sides of the hyperplane. Leaf nodes are the objects, and when a node spawns a leaf node it spawns nothing else.

The key result of the BSP tree is that it can be used as a search structure. For instance, a BSP tree can be used to efficiently locate the position of the viewer in the context of the 3D scene. As each node is encountered on a traversal of a BSP tree, the next node is chosen according to which side of the current hyperplane that the viewer is located. Continuing along this reasoning, the BSP tree can be used to locate the position of the nearest object. Once the nearest object has been located the rest of the objects in the scene can be given a viewing priority using the BSP tree. A viewing priority corresponds to how far the object is from the viewer, relative to the other objects in the scene. Those with a higher viewing priority are closer to the camera than those objects with a lower viewing priority. Viewing priorities can be used to organise objects according to their distance from the viewer and this organizing can be exploited by rasterizing algorithms such as the Painter's algorithm and depth-buffering.

In similar ways to scene graphs, BSP trees can also be used for collision detection and they can be used to speed up the rendering stage of the graphics pipeline. This is because the BSP is an efficient search structure. The BSP tree structure can be used to quickly determine the closest objects and they can also be used for determining entire volumes that are outside the viewing volume. BSP trees are implemented more widely in graphical environments consisting of large amounts of inanimate geometries such as first person computer games.

So in some ways, a BSP tree is another form of scene graph. It is just a data storage structure that stores the bounding volumes encapsulating unique objects in the 3D scene, in a hierarchical fashion. In a similar way to scene graphs, BSP trees are used to improve the

efficiency of rendering processes. The main difference between the two methods is that a scene graph can be used to describe a complete 3D scene. Linear transformations can be incorporated, geometries can be defined and lights can be added to create a realistic scene that is stored in one data storage object – a scene graph. A BSP tree is not used to create a 3D scene however, it *is* used to store the position of each object within a tree structure in a hierarchical fashion, and this tree structure can be used to as an efficient search structure for locating the position of the viewer or an object.

8. ACKNOWLEDGEMENTS

The author would like to thank Geoff Ericksson and Laz Kastanis who both supplied invaluable input, discussion and encouragement for the duration of this paper.

References

Ames A, Nadeau D and Moreland J. **VRML 2.0 Sourcebook** (2nd edition). New York: John Wiley and Sons. 1997.

Bethel, W et al. **Hierarchical Parallelism in a Scene Graph**. See http://www.r3vis.com/RMScenegraph/hpsg.pdf, 2001.

Blais C, Brutzmann D, Horner D and Major Shane Nicklaus. **Web-Based 3D Technology for Scenario Authoring and Visualization: The Savage Project.** Naval Postgraduate School, Monterey California. See http://web.nps.navy.mil/~brutzmann/Savage/documents/WebBased3Dtechnology-Savage-IITSEC2001.pdf, 2001.

Carey R and Bell G. **The Annotated VRML97 Reference Manual. 1997**

Clay S, Zhao J and Insinger C. **IRIS Performer: Real-Time 3D Rendering for High-Performance and Interactive Graphics Applications.** Silicon Graphics Inc. See http://www.sgi.com/software/performer/presentations/perf_wp_clr.pdf, 1998.

Eberly D. **3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics.** Academic Press, United States of America. 2001

Eckel G. **IRIS Performer: Getting Started Guide.** Silicon Graphics Inc. See http://www.cs.trinity.edu/The_Coure/cs3353/performer.getting-started.pdf, 1997.

Kilthau, S. **Non-Realistic Rendering API.** See http://www.cs.sfa.ca/~kilthau/personal/downloads/nrrapispec.pdf, 2000.

Hadwinger, M. **Design and Architecture of a Portable and Extensible 3D Game Engine.** See http://openparsec.sourceforge.org, 2002.

Koh E and Wishinsky B.J. **OpenGL Performer Programming: Student Handbook.** SGI Global Education Services. Silicon Graphics Inc United States of America. 2000.

Ladd E, O'Donnel J, Morgan M and Watt A.H. **Platinum Edition: Using XHTML, XML and Java 2.** Que Corportation, United States of America. 2001.

Naylor B.F. **A Tutorial on Binary Space Partitioning Trees.** See http://www.cs.utexas.edu/users/lin/cs315h/bsp-tutorial.pdf, 2002.

Naylor B.F. **Partitioning Tree Image Representation and Generation from 3D Geometric Models.** See http://www.graphicsinterface.org/pre1996/92-Naylor3Dto2D.pdf, 1996.

Reiners D. **Scene Graph Rendering**. OpenSG Forums. See http://www.vrjuggler.org/pub/scenegraph-rendering.ieeevr2002.pdf, 2002.

Rohlf J and Helman J. **IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics.** See http://www.cs.virginia.edu/~gfx/Courses/2002/BigData/papers/Interfaces%20and%20Software%20Systems/IRIS%20Performer.pdf, 2002.

Sowizral H. **Scene Graphs in the New Millennium**. Vision 2000. January/February 56-57. 2000

The Java 3D API Specification. Sun Microsystems. See http://java.sun.com/products/java-media/3DforDevelopers/j3dguide/j3dToc.doc.html, 2000.

Walsh A.E. **Understanding Scene Graphs**. Dr Dobb's Journal, 27:7, 17-26. 2002.

Walsh A.E and Bourges-Sevenier M. **Core Web3D**. Prentice Hall, United States of America. 2001.