Algorithms for Efficient Dynamic Hidden Surface Removal for Curved Surfaces

BTech Project Final Report

Final Semester 1998-99

by

Alok Kumar (95106)
Vivek Kwatra (95157)

under the guidance of

Dr. Sanjiv Kapoor
Dr. K.K.Biswas

Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Certificate

This is to certify that the thesis titled *Algorithms for Efficient Dynamic Hidden Surface Removal for Curved Surfaces* , being submitted by *Alok Kumar* and *Vivek Kwatra* in fulfillment of the requirements for the award of *Bachelor of Technology* in Computer Science & Engineering at the *Indian Institute of Technology* is a record of bona fide work carried out by them under our supervision.

The matter and results presented in this thesis are original and have not been submitted elsewhere, wholly or in part, for the award of any degree or diploma.

Dr. Sanjiv Kapoor
Dept. of Computer Sc. & Engg.
I.I.T. Delhi
New Delhi 110016, INDIA

Dr. K.K. Biswas
Dept. of Computer Sc. & Engg.
I.I.T. Delhi
New Delhi 110016, INDIA

Acknowledgement

We would like to take the opportunity to express our deep and sincere gratitude to our guide Dr. Sanjiv Kapoor and our co-guide Dr. K.K. Biswas for their continuous inspiration, guidance, invaluable advice and constant encouragement throughout the course of this project.

We sincerely appreciate the assistance offered by the staff of Microsoft Lab and Intel lab. We express our thanks to all our friends for their valuable help.

Alok Kumar
Vivek Kwatra

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Normal animation techniques involve storing each frame, and then successively displaying it. In many applications such as computer games or flight simulation, online animation techniques are used which involve storing of all the objects in the scene as original 3D objects and computing the new frame by changing the configuration of the actual objects. Scenes in which the objects are made up of polygons are easy to re-compute and render, but real-life objects are essentially curved objects and the dynamic rendering of real-life scenes containing these objects is of great significance. An important aspect of this problem is the efficiency with which the scene can be updated and the accuracy with which it is rendered. With this objective in mind, we have tried to devise data structures and algorithmic techniques which would facilitate fast updation of dynamic scenes.

1.2 Preview

In many applications, like engineering design of large equipment, and walkthroughs of ships and submarines, models are traditionally represented as millions of polygons. It is important to be able to visualize these models to speed up their design, prototyping and evaluation. While graphics systems have reached the capability to render millions of transformed, shaded and z-buffered triangles per second, they are not able to render such complex and detailed engineering models. For example, a high-end commercial graphics system, like an SGI Onyx with a Reality Engine II graphics accelerator and 2 Gigabytes of main memory, is not able to transform, Gouraud-shade, z-buffer and display the storage and handling room of a notional submarine 1 (see Figure 1.1) even once per second. Furthermore, empirical evidence shows that model sizes tend to grow ten fold every two years while the rendering speed only about doubles. These large models not only throttle the rendering pipeline, but also encounter system memory limitations. In fact, researchers like Clark [Cla76] realized quite early that not all triangles need be sent to the graphics sub-system every frame. He advocated hierarchically grouping objects into clusters spend only as much effort rendering each cluster as is needed for . The Sub model consists of 900,000 triangles.

The need to navigate through increasingly complex models, for simulation based design or design evaluation, has recently motivated considerable research in software techniques to speed up rendering. The main thrust of these techniques is to reduce the number of polygons passed

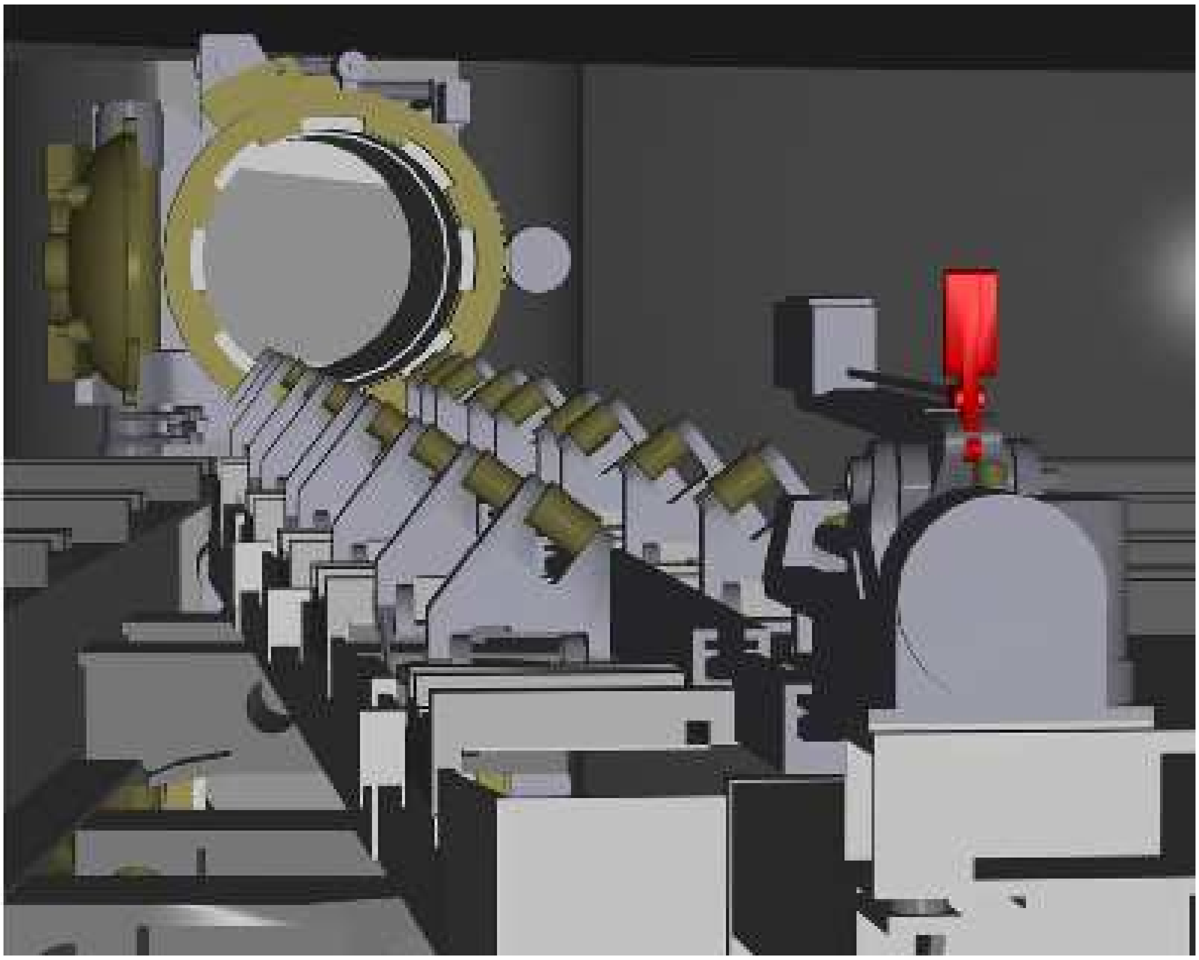

Figure 1.1: A Submarine storage and handling room: 900K Polygons

to the pipeline.

For large models, the number of off-screen polygons is likely to be high. Spatial partitioning with the help of data structures like Octrees [Hun78] and BSP trees [FKN80] are regularly used by rendering systems to reduce the number of off-screen polygons passed to the graphics pipe. Recently Teller et al. and Luebke et al. described techniques to subdivide models into cells and portals and only render cells visible through these portals. Funkhouser et al. use similar computation to fetch visible (and close to visible) parts of a model into memory, if the complete model does not fit into the main memory. These techniques are well suited for architectural environments but are not as effective for engineering models that are not easily partitioned into cells and portals.

1.2.1 Model reduction:

Graphics literature contains description of a number of techniques to reduce the number of polygons used to represent a model. In general, these methods generate a sequence of discrete approximations of parts of a model, each at a coarser level of detail from the previous one. At rendering time, the system picks and displays the appropriate version of each part based on its distance from the viewer. Switching smoothly between versions is a tough problem for these methods. The methods of progressive meshes alleviates this problem by varying detail more gradually. However, there remain difficult problems like handling degenerate models and efficiently switching detail to the desired level.

1.2.2 Image-based rendering:

Taking the concept of images further, some recent techniques dispense with geometry altogether and represent models as images [Che95, MB95, LH96, GGSC96]. The basic principle of image-based rendering is to generate multiple views from a small number of photographs. Such computation is quite slow on current graphics systems. Moreover, design of new geometry and its physical analysis is precluded by these methods due to the lack of exact geometry. The application of these techniques to computer aided design and manufacturing (CAD/CAM) remain to be proven. In many applications involving CAD/CAM, virtual reality, animation and biomedical engineering, the object models are designed in terms of tensor product trimmed non-uniform rational B-spline (NURBS) surfaces, which can exactly represent many geometries. This class includes Bezier surfaces and other rational parametric surfaces like tensor product and triangular patches. Large-scale models of shapes of automobiles, submarines, airplanes, etc. typically require tens of thousands of such surfaces to represent them. Constructive solid geometry, surface fitting over scattered data or surface reconstruction are some other techniques that produce many thousands of NURBS surfaces. NURBS offer the benefit of exact geometric representation and require significantly less memory than polygonal or image representation. Exact geometry ensures that we are able to render images at as high levels of detail as allowed by pixel discretization of images. NURBS are a popular tool to model surfaces and have found their way into graphics standards like PHIGS+ and OpenGL. Unfortunately, current renderers of these sculptured models on commercial graphics systems, while faster than ever before, are not able to render them in real-time for applications requiring walkthroughs and immersive design. Most systems perform a one time, high-resolution approximation of the NURBS in terms of polygons, thus giving rise to the enormous database sizes alluded to earlier, and all the associated problems. This dissertation seeks to change that by speeding up surface rendering.

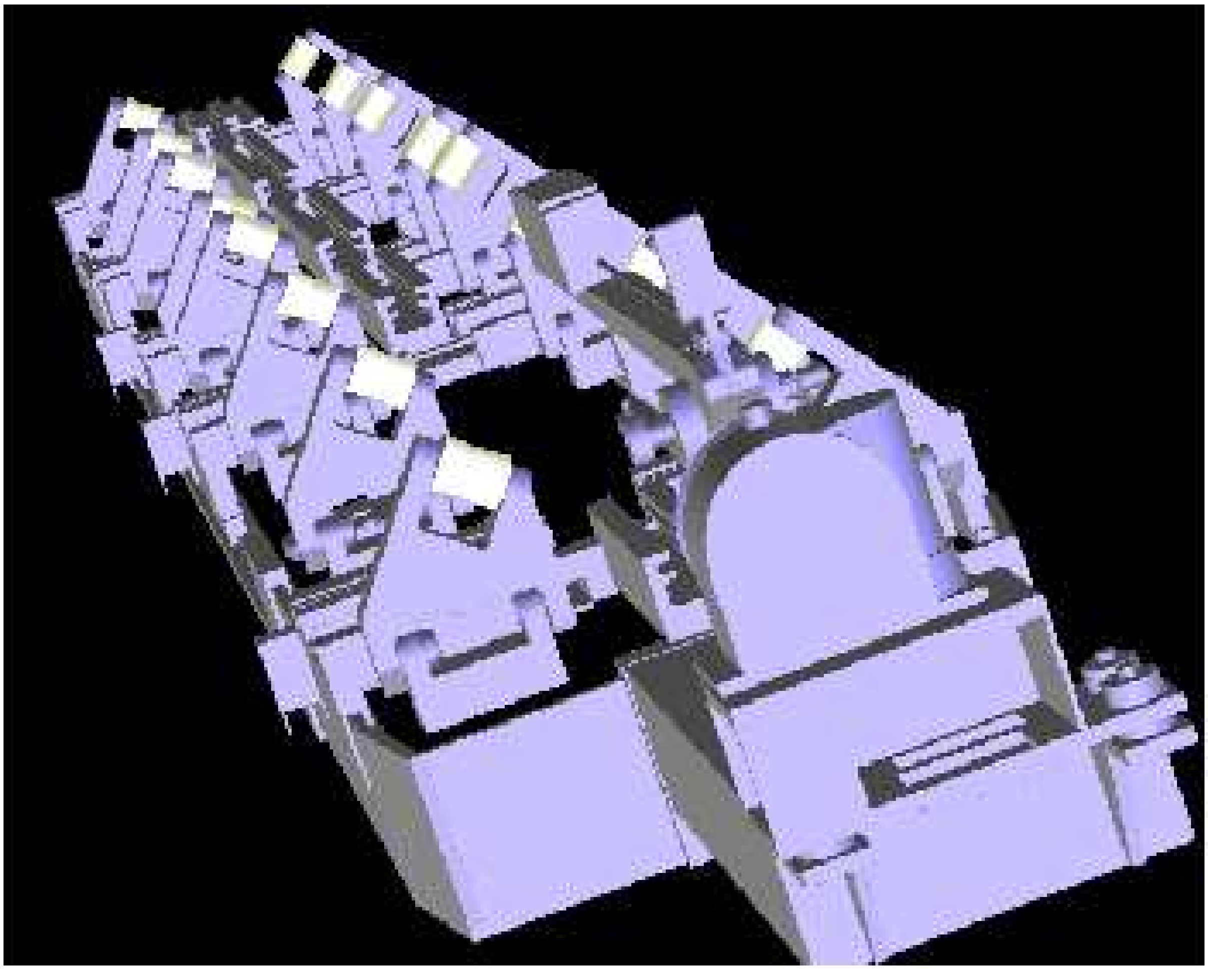

Figure 1.2: Pivot (from the Sub model): 4100 B'ezier patches

1.3 Surface rendering techniques

In this section we briefly review the current techniques to render curved surfaces. Due to its importance in design, curved surface rendering has been an active area of research for more than two decades. Forrest [For79] performed one of the earliest studies on benefits of shaded displays of curved surfaces. Forrest claims that simple wire-frame renderings do not lend themselves well to effective visualization of a surface, even with hidden line elimination and perspective projection. Forrest conjectures that real-time manipulation of smoothly shaded surfaces is too expensive to realize and advocates the usage of surface snap-shots from certain view points overlaid with wire-frames that are planar sections of the surface and garnished with a Gaussian curvature based shading. A number of techniques to display smoothly shaded surfaces have been devised in the past. We can broadly classify these techniques into four categories:

1.3.1 Pixel level subdivision

Catmull [Cat74] presents one of the first algorithms for displaying surfaces up to the accuracy of the resolution of display. This algorithm subdivides a surface into smaller and smaller sub-patches until each sub-patch is no bigger than the size of one screen pixel. The pixel is then assigned the color of that sub-patch. Catmull devised a fast subdivision algorithm for bicubic patches. While resulting in effective images, this method is quite slow due to the depth of subdivisions. The adaptive forward differencing of Shantz et al. [SL87, SC88] is similar in principle and handles trimming curves. It converts the NURBS basis to a forward difference basis and scales the difference up or down depending on the speed of parametric curves on the surface. Apart from having slow speed, this method also suffers from low accuracy, specially in case of high degree surfaces.

1.3.2 Scan-line Display

Scan-line based algorithms for display of curved surfaces are faster than pixel level subdivision in general. These algorithms process the screen-pixels in scan-line order. For each scan-line, the intersections of the scan-plane with the surfaces are used as spans. Whitted [Whi78] advocates subdividing surfaces at silhouettes and in areas of high curvature. This decomposes surfaces into elements, each of which are free of silhouettes except at their boundaries and are relatively flat. He defines surface elements using four boundary curves and decomposes each curve into segments such that only one intersection may occur with any scan-line. Whitted uses Newton's iteration to compute the extrema of spans and interpolates the normal across a span for fast shading. Blinn does not pre-process the patches but performs Newton- Raphson iterations to compute all extrema along each of the screen coordinates. He also presents heuristics to handle singularities and many other special cases. Lane et al. perform the scan-line intersection by recursively subdividing patches. If the convex hull of a patch intersects the current scan-line, the patch is subdivided into smaller sub-patches so that the convex hull of each sub-patch bounds it more tightly. Once the patches are flat enough, the convex hull almost coincides with the patch, and is used to compute the spans. Most scan-line based methods exploit spatial coherence to speed up span computation. However, due to the inherent complexity of scan-line intersection, these algorithms are too slow for real-time display of large models.

1.3.3 Ray Tracing

Whitted presents one of the earliest methods of producing photo-realistic images of bicubic surfaces using ray tracing. He uses bounding volumes to eliminate surfaces not intersected by a ray. If a ray intersects the volume bounding a surface, that surface is subdivided using the scheme of [Cat74] and the process is repeated until the volume is small enough to be approximated as the point of intersection. Whitted also performs anti-aliasing by filtering in the areas of high luminance gradient. Kajiya [Kaj82] presents a numerical solution and reduces the ray patch intersection in real projective 3-dimensional space to intersection of algebraic curves in 2-dimensional complex space. It uses results from elimination theory to compute the points of intersection. Nishita et al.'s algorithm is subdivision based, but performs the surface subdivision using an operation called B'ezier clipping. B'ezier clipping eliminates large parts of a surface that do not intersect a ray. This technique is applicable to trimmed surfaces as well. Although ray tracing can generate high quality images, it can involve millions of ray intersections, each needing thousands of operations. As a result, these methods are too slow for interactive display.

1.3.4 Polygonal Approximation

A number of algorithms for approximating curved surfaces with polygons are described in the literature [Cla79, Baj90]. As [Cat74], these methods subdivide the surfaces into smaller components, but only until each component is flat enough to be approximated by a polygon. (In this sense the algorithm of Lane et al. may also be classified as a polygonal approximation method.) Some apply the subdivision recursively, stopping when their flatness test is satisfied, while others first pick a number of sample points on the surface and then polygonize these points. Once the approximating polygons are computed, standard polygon rendering techniques are applicable. One common problem with these techniques are occurrence of cracks between adjacent patches if their triangular approximations do not match at the boundary. A common solution [Cla79] to this problem is to stop subdividing a boundary if the adjacent boundary reaches its subdivision limit. Most recent efforts for fast display of surfaces have concentrated on the approximation method. As opposed to the methods described earlier, these methods are able to make efficient use of recent advances in triangle rendering and hardware capabilities available on current graphics systems and hence come close to real-time rendering.

Chapter 2

Representing Curved Objects

2.1 Polygonal Representations

Curved objects have traditionally been modeled by first tessellating the curved surface by polygons and then using standard methods of hidden surface for polygonal surfaces. This method has been successfully used in static scenes, but in a dynamic scenario this may not be the best solution. The following disadvantages may be associated with this approach -

- The accuracy of the representation depends on the number of polygons used for approximation. As the number of polygons increases, it causes an increase in the space used for their storage and also in the size of the data structure (BSP in our case) representing the scene.
- The simulation of motion in a dynamic scene may be done through successive insertions and deletions of the moving object into the BSP tree. When the size of the BSP tree is large, with the polygonal representation, it may mean that we need to insert a large number of polygons in an already large tree for each movement of the object. Even if it is possible to insert the entire sub-tree of the object at once, instead of individual polygons, there may arise intermediate steps requiring merging of subtrees which would increase the time complexity considerably.
- An integral part of rendering for curved objects involves shading. To make the curved object image look free of polygonal edges, the shading model needs to be strong enough to do so (such as the Phong model which stores the normal at each vertex and is computationally intensive).

Our approach uses only the parametric representation of the curved surface and does not tessellate the surface into polygons at any step. The BSP tree structure in our approach does not store polygons in its nodes. It instead stores complete objects or their fragments in its nodes. The objects are stored in the form of a list of curved surface patches (which constitute the object). We proceed to explain the various data structures and algorithms used by us in our approach.

2.2 Parametric Representations

2.2.1 Bezier Curves and Patches

Before describing how we define curved objects, we present here a brief description of Bezier curves and patches, since we use them to define the component patches of our curved objects.We would start with the explanation of Bezier curves and then we would move on to Bezier patches.

Curves can be approximated by polylines formed by joining points interpolated on the curve. Polylines are first degree approximations to the curve. To obtain a higher degree of accuracy, we could join the interpolated points with curves of degree higher than one. Thus a complex curve is approximated using a piecewise polynomial curve instead of a piecewise linear curve. The higher degree curve we are talking about are normally quadrics or cubics. We use parametric representations for them, i.e.

$$x = x(t), y = y(t), z = z(t)$$

to represent x, y and z co-ordinates respectively.

Cubic polynomials are most often used because lower-degree polynomials give too little flexibility in controlling the shape of the curve, and higher-degree polynomials introduce unwanted wiggles increasing computation. The cubic polynomials that define a

curve segment,

$$Q(t) = [x(t)\ y(t)\ z(t)]$$

are of the form

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$
$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

where

$$0 \leq t \leq 1$$

so that the generated curve segments are finite.

Let

$$T = [t^3\ t^2\ t\ 1]$$

and C be the coefficient matrix in the above system, so that

$$Q(t) = T.C$$

The **Tangent Vector ($\boldsymbol{TV}$)** at a point is defined to be the derivative of $Q(t)$ $w.r.t$ t at that point (i.e. the point defined by t). The direction of TV gives the orientation of the curve while its magnitude tells how strongly the curve is pulled towards the tangent.

A complex curve consists of a collection of various cubic curve segments. For two segments to be adjacent segments in the collection, they need to fulfill certain boundary conditions. There has to be a certain level of continuity between two adjacent cur

ve segments and we accordingly have the following continuities among adjacent curve segments.

G_0 geometric continuity exists if the two curve segments join together

G_1 geometric continuity exists if the two tangent vectors. TV_1 and TV_2 at the join point for the two curves have the same direction. The magnitude may not be the same. In other words, we should have TV_1=$k.TV_2$ where $k > 0$.

If the magnitude as well as direction of the tangent vectors is same then the curve is said to be C_1 parametric continuous (first degree continuous in the parameter t). If the direction and magnitude of the nth derivative of $Q(t)$ are equal at the j

oin point then the curve is called C_n continuous.

A curve segment $Q(t)$ is defined by constraints on endpoints, tangent vectors, and continuity between curve segments. Each cubic polynomial has four unknowns, so four constraints are needed. Let G (which is known as the geometric vector) be a four element vector defining the four constraints. It basically gives the geometric conditions such as endpoints, tangent vectors etc. Now, we can write

$$Q(t) = T.C$$

as

$$Q(t) = T.M.G$$

here M and G are constants, which come out through specific choices in the type of constraints in G. There may be several classes which may define a curve in several ways depending upon the choice of G. A unique M characterizes any such class.

Bezier Curves refers to a class of curves in which the constraints in G are the two end points, P_1 and P_4 and two points not on the curve, P_2 and P_3. P_1P_2 and P_3P_4 are used to calculate the tangent vectors for the end-points of the curve. Thus the geometric vector for the Bezier curve is

$$G = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

The tangent vectors are defined as

$$R_1 = Q'(0) = 3(P_2 - P_1)$$

$$R_4 = Q'(1) = 3(P_4 - P_3)$$

Also,

$$Q(0) = P_1$$

$$Q(1) = P_4$$

Let the basis matrix for Bezier curves be M_B. Then from the above set of equations it is very clear that we can write the following

$$Q(0) = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}.M.G$$

$$Q(1) = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}.M.G$$

$$Q'(0) = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}.M.G$$

$$Q'(1) = \begin{bmatrix} 3 & 2 & 1 & 0 \end{bmatrix}.M.G$$

This is equivalent to writing

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} .M. \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

From this equation we get

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Therefore, we can represent a Bezier curve by the equation

$$Q(t) = \sum_{i=1}^{4} p_i B_i(t)$$

where

$$\begin{bmatrix} B_1 & B_2 & B_3 & B_4 \end{bmatrix} = T.M$$

We can move to Bezier patches by extending Bezier curves in two dimensions. Mathematically, a Bezier patch can be defined as

$$Q(u,v) = U.M.P_c.M^T.V$$

where
U and V are the row and column vectors for the parameters u and v.
P_c is the control matrix of 16 control points.
The above equation can alternatively be written as

$$Q(u,v) = \sum_{i=1}^{4} \sum_{j=1}^{4} p_{ij} B_i(u) B_j(v)$$

where B is defined similarly as for Bezier curves. In the above equation, if we fix at v=0, we get

$$Q(0,v) = \sum_{j=1}^{4} p_{0j} B_j(v)$$

which is simply a Bezier curve in the parametric space of u. Thus, by varying v from 0 to 1, we obtain a series of Bezier curves in u, which move across the patch. The properties of Bezier patches are similar to those of Bezier curves. We use P_c to store the 16 control points of the patch, which themselves when taken in rows and columns, form the control points of a Bezier curve.

2.2.2 Representing Objects

For the purpose of defining a curved object, we subdivide it into a set of Bezier patches which together build up the entire object. The need to store such patches comes from the fact that a complex surface behaves as a cubic polynomial curve (or any other finite degree polynomial curve in general) only in patches and thus cannot be represented by a single parametric equation. Thus, an object is defined as a list of Bezier patches.

Since we restrict ourselves to cubic patches, each such Bezier patch is defined by a set of 16 control points. Also since the adjoining patches in an object satisfy certain continuity constraints, they have certain common points and certain points which are dependent on other points. Consequently, in the definition of an object, each constituent patch is defined as a matrix of control points. The matrix actually contains pointers to the control points instead of actual points, so that we do not need to store duplicate control points for consecutive patches and any updation in a common control point is done automatically for all patches containing it.

2.3 Hidden Surface Removal for Curved Surfaces

2.3.1 The Dynamic Scenario

The problem of hidden surface removal in curved surfaces is dealt with keeping in mind the fact that they represent a dynamic scene in which recomputation of the scene is required at every frame. The data structure we have used for the storage of the entire scene, i.e. its constituent objects is a BSP tree. A BSP Tree is a commonly used data structure for storing polygons describing a scene with tessellated objects. In a dynamic scenario, the choice of BSP tree is a provident one, since it is an object space approach and facilitates the dynamic updation of a scene as it contains information about the original configuration of the scene. In an image space approach, we do not have any information left regarding the original 3D scene, and hence a dynamic updation requires computation of the entire scene again.

The curved objects stored in the BSP lie at the leaf nodes of the tree and the tree nodes contain separating planes, which are planes used to separate two linearly separable objects. Linearly separable objects are defined as objects which are completely separable with a plane. Here we are working with an assumption that any two objects in space are linearly separable. While this assumption may seem too simplistic for the scene, it can be explained as follows. Firstly, for an initial application of our approach, it is necessary to work with such a simplifying

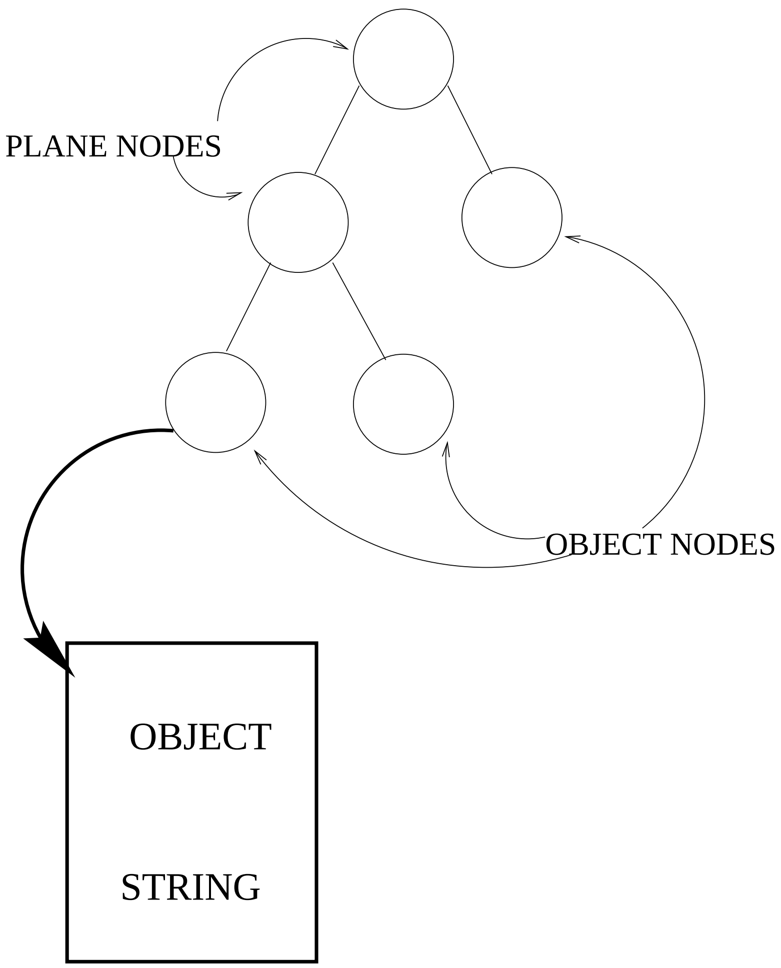

Figure 2.1: Basic structure of the BSP tree

assumption. Secondly, any two objects which are not linearly separable can be considered as part of a single object. In such a case we do not need to find any separating plane between the two. Note that the purpose of finding a separating plane and thus storing the objects in the BSP is to facilitate hidden surface removal by its appropriate traversal. Once the objects have been ordered by such a mechanism, we still need to render the curved object in a way such that the correct order among its constituent patches is maintained. Hence even if we are not able to find a separating plane between two objects and consider them to be part of the same object, we can find an ordering for the collective object and render it in a way that the correct order between the various patches is maintained. Since, unless the scene is too complex, such a situation would not occur very frequently, thus we can handle this as described above. Another situation which may arise using this approach may be that the collective object and a third object may not be linearly separable. To avoid such a situation, we dynamically change the order in which we insert objects in the tree so that any object for which it is not possible to find a separating plane up to a certain threshold depth in the tree, are appended to the end of the list of objects to be inserted. It would not be very helpful to do so, once we have percolated deep into the tree because then we would be incurring the computation cost for the insertion of the object twice.

2.3.2 Construction of the BSP Tree

We are initially given a list of objects describing the scene in a random order, and we need to construct the BSP tree from this list.

Let the list of objects be **L**. Each node of **L** would contain -

- one of the objects in the scene, **O**.
- a tree formed from some or none of the nodes of the BSP tree which we call $\boldsymbol{T_O}$. This tree would contain the planes which would cause the object to fragment while it is being inserted into the BSP. This would be help us rendering of the object, each fragment at a time, without explicitly storing separate fragments at the leaf nodes of the BSP tree. An appropriate traversal of this tree, would give us a region in space, whose intersection with **O** would give the required fragment we are interested in. This 'appropriate' traversal would be obtained by storing a binary number with every fragment in the BSP tree.
- a pointer to one of the fragments of the object in the BSP tree. This would help us in fast deletion of the objects because we will directly reach a node storing one of the object fragments in the tree and we can then traverse all such nodes and delete them one by one.

The structure of the BSP tree which would be constructed from the above list, **L** of polygons is described below -

- It is essentially a binary tree which partitions the space into regions. We name this BSP tree as **B**.
- All the nodes of **B** either contain planes or object fragments.
- Only leaf nodes of **B** contain object fragments.
- All other nodes of **B** contain planes which partition the space.
- Any node **T** has a particular region, say **R** in space associated with it.
- If **T** contains a plane, then this plane again divides **R** into two regions, with which the left and the right child of **T** are associated.
- If **T** contains an object fragment then **T** should be a leaf node and the fragment contained in **T** should lie completely within the region associated with **T** and no other fragment should lie in this region.
- The root of **B** node is associated with the entire 3-D space.
- The leaf nodes store the object fragments by keeping a pointer to the object, **O** in **L** and a binary string **s**, which tells how to traverse the fragment tree,$\boldsymbol{T_O}$, stored in **O**. A *0* in **s** $\Rightarrow$ traverse the left child, and a *1* in **s** $\Rightarrow$ traverse the right child. The leaf nodes also contain a pointer to the next fragment of the same object. This in fast deletion of the object.

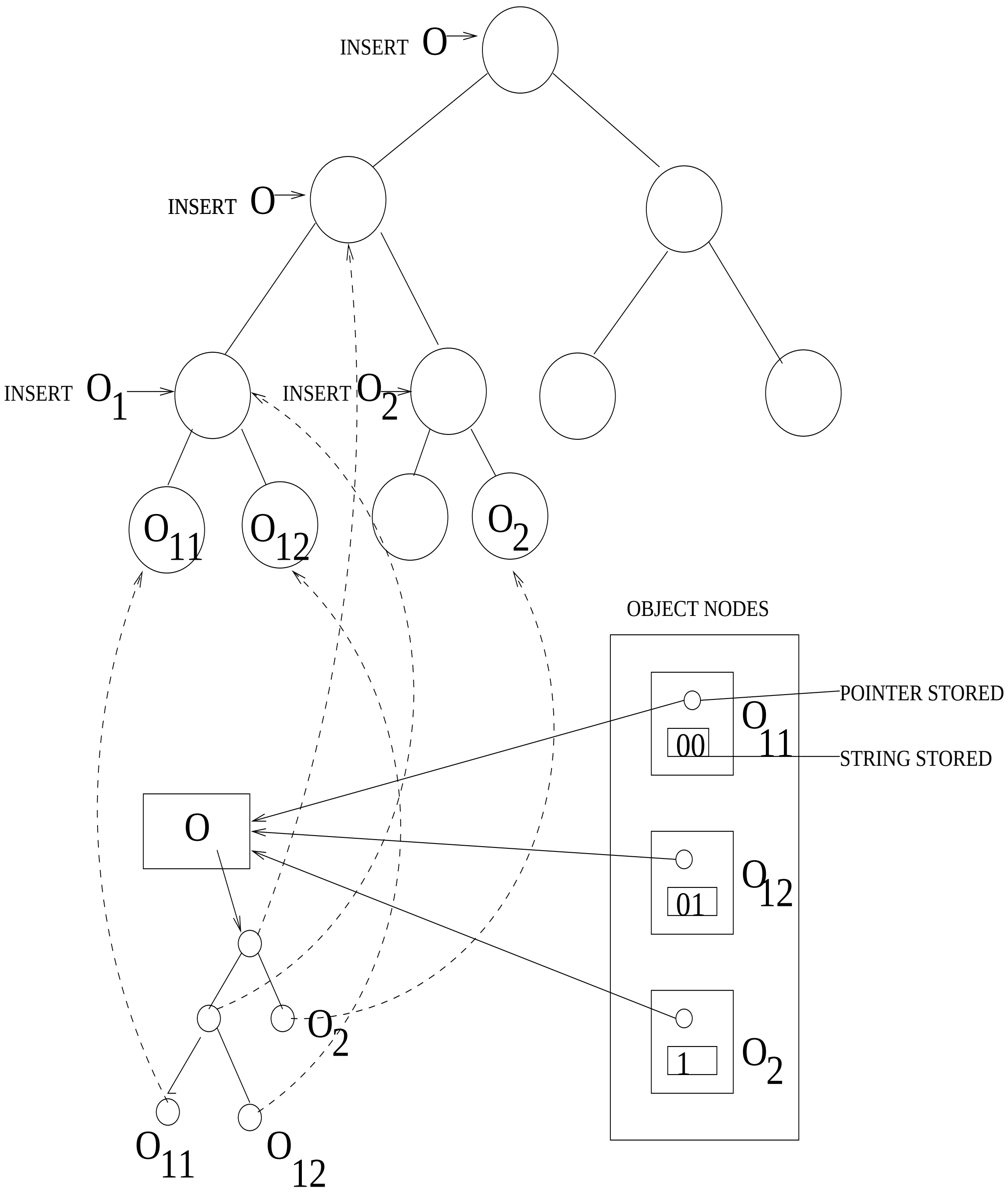

Figure 2.2: Storing information for fragments of objects

Chapter 3

Dynamic Updation Algorithms

3.1 Simulating Motion

In this chapter, we describe how we construct and manipulate the BSP tree structure described previously and use it for efficient simulation of the dynamic scene. The simulation of motion of objects is done basically by deleting the moving object, computing its new position and re-inserting the moved object into the BSP tree. The two basic updation algorithms, i.e. insertion and deletion are described in the remaining part of this section.

3.1.1 Inserting an Object

Now we describe how to insert an object **O**, from the object list **L** into the BSP tree **B**. The object fragments have to lie at the leaf nodes associated with the appropriate regions in which they lie. If we can find a region in which the entire object lies, we place it there, or else it is fragmented into smaller objects which can fit into single regions. The algorithm is described below -

We construct a new node for the BSP tree, $\boldsymbol{N_O}$, which contains a pointer to the object **O** in the list **L** and a null string **s**.

We take **R** as a pointer to the root node of **B**. **R** would either contain a plane or an object fragment (if it is a leaf node as well).

The node $\boldsymbol{N_O}$ is filtered w.r.t **R** as follows -

1. **R contains a plane :** Let this plane be **P**. In this case we see whether **P** cuts **O** or not.

   - In case it does not cut **O**, **O** would lie completely towards one side of the plane. If it lies towards the left of **L**, we insert it into the left subtree of **R** and vice-versa.
   - If **P** cuts **O** into two fragments, $\mathbf{O}_L$ and $\mathbf{O}_R$, the left and the right one respectively, we insert them into the left and right subtrees of **R** respectively. For this, we generate two new tree nodes, $\boldsymbol{N_L^O}$ and $\boldsymbol{N_O^R}$. $\boldsymbol{N_L^O}$ contains a pointer to **O** in the object list **L** and a string $\mathbf{s}_L$, given by

     $$s_L = s.0$$

     $\boldsymbol{N_O^R}$ contains a pointer to **O** in the object list **L** and a string $\mathbf{s}_R$, given by

     $$s_R = s.1$$

where $\mathbf{s}_L$ and $\mathbf{s}_R$ represent the fragments, $\mathbf{O}_L$ and $\mathbf{O}_R$, respectively.

We now filter $\boldsymbol{N_L^O}$ w.r.t the left child of **R** and $\boldsymbol{N_O^R}$ w.r.t to right child of **R**.

2. **R contains an object fragment :** Let this object fragment be **O'**. In this case we try to find a separating plane, $\mathbf{P_s}$ between **O** and **O'**. We will be able to find the separating plane, if the two objects are linearly separable in space. If we are able to find such a plane, we create two new tree nodes, $\boldsymbol{N_1^O}$ and $\boldsymbol{N_2^O}$. We copy $\boldsymbol{N_O}$ in $\boldsymbol{N_1^O}$ and **R** in $\boldsymbol{N_2^O}$, and put the separating plane in the **R**.

   In the case that the two fragments intersect or are not linearly separable, we merge the two objects into the same node. We keep a tag in the node, indicating that it contains more than one fragment, and a list of the fragments is maintained. The hidden surface removal for these objects may be done in the rendering stage. Note that since the number of objects in a single node would not be too many, a pixel by pixel computation of the visible surface at this stage would not be too time consuming.

   Also, whenever we filter a node $\boldsymbol{N_O}$, containing an object fragment **O**, w.r.t this node, we would have to look for linear separability of **O** with all the objects in the node. To avoid this, we can do the following -

   Let the node containing the list of objects be $\boldsymbol{N_L}$. We see if the object fragment **O**, which we are inserting is linearly separable w.r.t any of the objects contained in $\boldsymbol{N_L}$. If we can find any such object(s), we name the set of these objects as **S**. We now find a separating plane between **O** and the objects in **S**. We then create two new nodes $\boldsymbol{N_P}$ and $\boldsymbol{N_S}$, and put the separating plane in $\boldsymbol{N_P}$ and the set **S** in $\boldsymbol{N_S}$. Also we remove the elements of **S** from $\boldsymbol{N_L}$. Then we replace $\boldsymbol{N_L}$ with $\boldsymbol{N_P}$ and make $\boldsymbol{N_S}$ and $\boldsymbol{N_O}$ its children. After this, we insert all the elements left in $\boldsymbol{N_L}$ into the subtree rooted at $\boldsymbol{N_P}$.

   What we basically do above is that to reduce the number of objects in a node, while inserting a new object, we partition the set into two such that one of these sets can be separated from the new object with a plane. On re-insertion of the elements of the other set, they get partitioned into smaller fragments w.r.t the just computed plane and thus can possibly be separated linearly from other fragments, which are either the elements of the former set or the new object.

Analysis

In the above algorithm, described for inserting an object, it traverses down the tree unless it reaches a leaf node. If the object does not fragment, the maximum time it takes for insertion is $O(d)$, where d is the depth of the BSP tree. In the worst case, the object gets fragmented at every node. In such a case, the time complexity can be computed from the following recurrence -

$$T(d) = 1 + 2T(d-1)$$

$$T(0) = 1$$

$$\Rightarrow T(d) = O(2^d)$$

where d is the depth of the BSP tree. The BSP is an ordered tree and hence we can keep it balanced, so that

$$d = O(log(n))$$

where n is the number of nodes in the tree, or the number of object fragments, i.e. the number of leaf nodes in the tree (which is half the total nodes in the tree). So, we have

$$T(n) = O(n)$$

In this analysis, we assume that an object gets fragmented at every non-leaf node of the tree, which is rather improbable. Instead, if we assume that an object gets cut at a node with a probability of 0.5, then the following expected time complexity may be derived -

$$T(d) = 1 + 1/2(T(d-1) + 2T(d-1))$$

$$\Rightarrow T(d) = O(1.5^d)$$

$$\Rightarrow T(n) = O(n^{.405})$$

which is sub-linear. In fact the actual probability may be even less than 0.5, and the lesser the object gets fragmented, the more we tend to lower bound, given by

$$T(n) = \Omega(log(n))$$

3.1.2 Deleting an Object

Deleting an object from the BSP tree, is of critical importance in our structure because it greatly expedites the process. The object node in the object list stores a pointer to the first object fragment, and each subsequent node in the BSP contains a pointer to the next fragment. So, while deleting we traverse the list of fragments and delete each node one by one. Also the tree-structure stored in the object node in the object list is removed.

Analysis

The time complexity for the deletion is simply $O(f)$, where f is the number of fragments of the object. Again, if the object gets cut at every node in the tree, f would grow to n, where n is the total number of object fragments already present in the tree. This means that the number of leaf nodes in the tree would double at almost every iteration. If we instead take the probability of fragmentation as 0.5, the number of fragments of the object would grow with the depth d as

$$f(d) = 3/2f(d-1)$$

This implies that the expected number of fragments would be $2/3n^{0.405}$ finally. Although we have given this figure, the actual number of fragments is highly non-deterministic and would vary largely with the arrangement of object sin the scene. In a real case, infact, the number of fragments should intuitively be much less. In many cases, the object would not be fragmented at all, so that the leaf nodes increase just by one. The figure of n for the number of fragments above is highly non-realistic.

3.2 Dynamic Rendering

Parametric representations of curved objects have been used for their rendering, but any rendering technique has always used polygonization of the bicubic patch at some stage. Our approach is to eliminate the intermediate use of polygonization at any stage and directly work with the parametric representation. For the rendering of a curved object, shading is an implicit requirement since we need a two-dimensional image to give the effect of 3D space. This means that for all the pixels in the image space, we need to compute the intensity value at that pixel. When rendering a single object, we need to compute the visible surface of the object, which would be projected at that pixel and then compute the intensity value. Calculating the visible surface is not a very big overhead because the number of surface between the viewpoint and the pixel would not be very large. If the object is a convex object, the number of such surfaces is just two.

In the dynamic case, when an object moves , we first delete the object from the BSP tree, compute the parameters of the object based on its new position and then insert the new object into the BSP. After this, we render the scene by traversing the BSP appropriately depending on the position of the viewpoint. Whenever we reach an object fragment, we compute its projection on the image plane. We first search for boundaries on the patches contained in the object, by finding the angle between surface normals and the line joining the surface with the viewpoint. An angle of 90^0 means the point on the object surface lies on the boundary of its projection in the image plane. An angle greater than 90^0 implies the surface is not visible from the viewpoint. Once we know the boundary for the fragment in the image plane, we compute the visible surfaces for the pixels inside the boundary and place the appropriate intensity value on the pixel.

We consider only that part of the surface in the fragment which lies in the region obtained by traversing the tree stored in the object list according to the binary string stored in the fragment node. Note that when we render different fragments separately, we may get cracks within the object. This is partially avoidable by merging any fragments which get rendered in succession. Finally, since we are using the painter's algorithm for rendering the entire scene, we may render the same pixel more than once. Techniques to avoid this also need to be explored and integrated.

Chapter 4

Computation of Separating Planes

To find separating plane between curved objects, there are several approaches considered , two of which are

1. To cover control points of Bezier Patches, constituting the curved object, with an ellipsoid and then find the separating plane between the ellipsoid covers.
2. To use linear programming techniques for finding separating plane between the control points of the Bezier Patches.

4.1 Algorithm Using Ellipsoidal Cover

In this algorithm, we use the fact that convex hull of control points of a Bezier Patch covers the Bezier Patch itself. So , we first find the ellipsoidal cover (which is itself an convex hull and also easy to deal with because of the algebraic simplicity) of the control points of all Bezier Patches constituting the curved object. Then , we find the separating plane between two ellipsoids, assuming they are non-intersecting.

4.1.1 Finding the Ellipsoidal Cover of control points

We would like to cover the control points with an optimal ellipsoid so as to assist determination of separating plane. An ellipsoid consists of three radii $\boldsymbol{R_1}$, $\boldsymbol{R_2}$,$\boldsymbol{R_3}$ in mutually perpendicular directions and a **Center**. Center of the ellipsoid is found as the center of the minimal cuboidal cover of the control points, such that sides of the cuboid are parallel to coordinate axes,The ellipsoidal cover, then is constructed using the following algorithm -

1. Find the most distant point $\boldsymbol{C_1}$ among the control points from the center of the object. The line joining $\boldsymbol{C_1}$ - **Center** is chosen as axis and $\boldsymbol{R_1}$ is assigned its length.
2. For all points, p, apart from $\boldsymbol{C_1}$
   (a) Construct an ellipse with "this" point p and $\boldsymbol{C_1}$ lying on the ellipse and $\boldsymbol{C_1}$-center as major axis.
   (b) Determine the corresponding length of minor axis.
3. Choose the point $\boldsymbol{C_2}$ with maximum minor axis length and fix corresponding minor axis length as $\boldsymbol{R_2}$.

4. For all points, p, apart from $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$
   (a) Construct an ellipsoid with "this" point p, $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$ lying on the ellipsoid and **Center** as its center.
   (b) Determine the corresponding length of the third axis.
   Choose the point $\boldsymbol{C_3}$ with maximum length of third axis and fix corresponding third axis length as $\boldsymbol{R_3}$.

4.1.2 Determination of separating plane between the ellipsoidal covers

Now, we need to determine separating plane between two ellipsoids ($\boldsymbol{E_1}$,$\boldsymbol{E_2}$) in arbitrary orientation. We transform one of the ellipsoids, say $\boldsymbol{E_1}$, into a sphere by compression of the axes. The other ellipsoid $\boldsymbol{E_2}$ is altered using the same transformation matrix. Now, we bring $\boldsymbol{E_2}$ to the origin with its axes aligned with the coordinate axes. So as to find the separating plane, a point $\boldsymbol{P_2}$ is selected on $\boldsymbol{E_2}$, such that the line **L** joining $\boldsymbol{P_2}$ to the center of transformed $\boldsymbol{E_1}$ is normal to $\boldsymbol{E_2}$. The line **L** is also normal to $\boldsymbol{E_1}$ (transformed) and intersects $\boldsymbol{E_1}$ at $\boldsymbol{P_1}$. Mid_plane is selected as perpendicular to **L** at the mid_point of segment($\boldsymbol{P_1} - \boldsymbol{P_2}$).

Let coordinates of $\boldsymbol{P_2}$, which is unknown, be (x,y,z) and center of $\boldsymbol{E_1}$(known) be (u,v,w) and length of axes of $\boldsymbol{E_2}$(known) be (a,b,c). The equations derived from conditions under which a line is simultaneously normal to $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$ are :-

$$(a-b)xy - avx + buy = 0$$
$$(b-c)yz - bwy + cvz = 0$$
$$(a-c)xz - cuz + awx = 0$$
$$ax^2 + by^2 + cz^2 = 1$$

One of the equations of the first three has to be eliminated as they are not independent , then we get three equations and three variables. As there are three polynomial equations and three variables, a solution exists, hence we will always be able to find a solution of above equations. Octave libraries can be used to solve the above equations.

4.2 Algorithm Using Linear Programming

In this algorithm, we again use the fact that convex hull of control points of a Bezier Patch covers the Bezier Patch itself. So, here we take a general equation of a plane and apply constraints for each control of one object to lie of the negative side of the plane and then similarly all control points to lie on the positive side of that plane. Thus, we get a simple linear programming equation, where we try to maximize the average distance of a control point from the plane , thus the plane is the plane of maximum separation between the objects.

Let the equation of the plane be

$$ax + by + cz = d$$

Then, for a control point (h,k,j) of the first object we will get a constraint

$$ah + bk + cj - d \leq -1$$

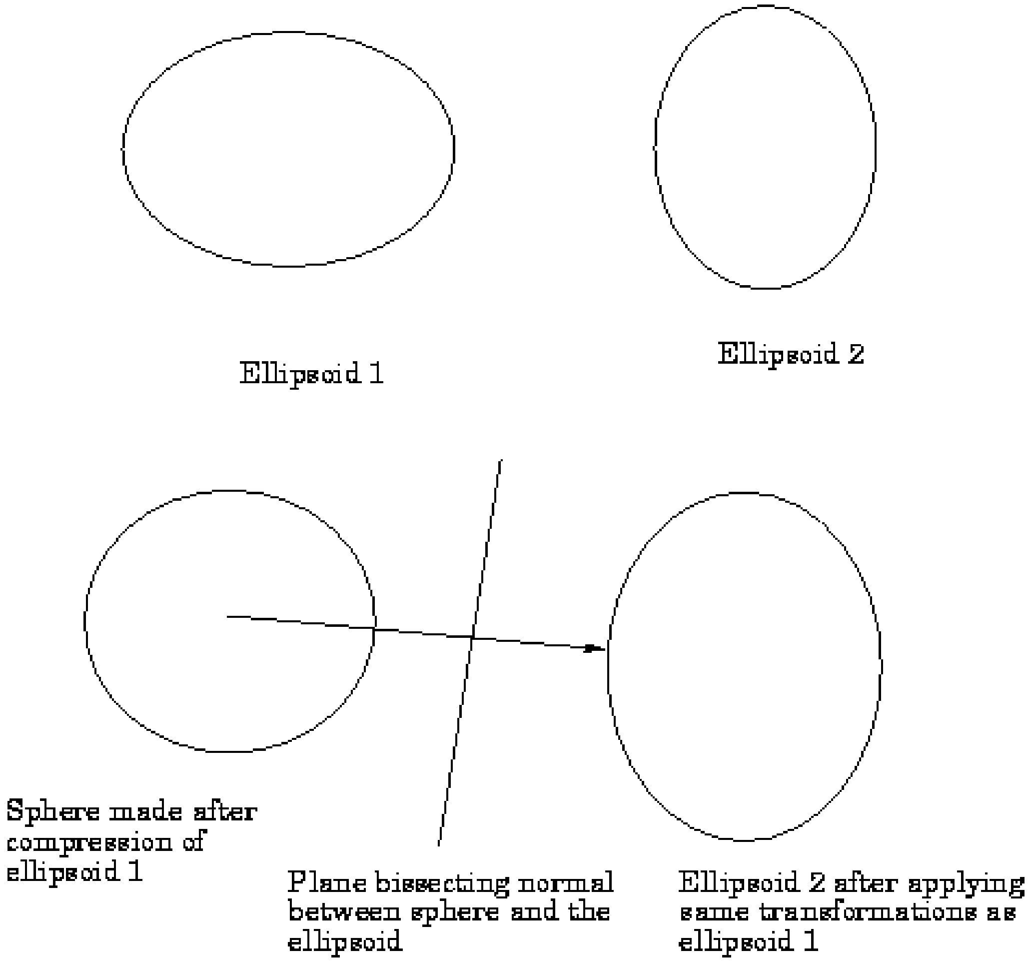

Figure 4.1: Finding Separating plane using ellipsoid

For the other object, the constraint will be

$$ah + bk + cj - d \geq 1$$

The function to be maximized is

$$\sum \left|\frac{ax + by + cz - d}{\sqrt{a^2 + b^2 + c^2}}\right|$$

which is the sum of distances of all the control points from the separating plane. This linear program can be solved in order of number of control points.

Chapter 5

Algorithm for Space Partition Trees Using Curved Separating Surfaces

Till now, we have always used straight surfaces for finding space partition trees. But as now, the objects in our world are not just simple polygons, so it lead to possibility of using extended curved surfaces as the separating surfaces. But, when we use curved surfaces to separate objects, it does not directly divide objects in two parts, where each part are distinct and one part has priority over another just depending upon where the viewpoint lies. Here, in the case of curved partition surfaces, we have to calculate dynamically the priorities of the objects, using the position of viewpoint and the separating curve.

5.1 Algorithm for 2 Dimensional objects

Here, the 2D curves are Bezier curves, the extension of which in two dimensions are the separation curves for the partition tree. We find the partition tree in the similar fashion as BSP tree, that is to put all the objects one side of the separating curve at the left child of the root node and the objects on the other side at the right child. We create the whole tree in similar fashion recursively.

5.1.1 The Data Structure

Each node in the partition tree is either a Bezier patch or a line. And its left and right subtrees are the partition trees of the objects in the left and right side of the partitioning curve.

5.1.2 The Algorithm

While displaying the objects, we first traverse the whole tree. The variation from the traversal of BSP trees here is that, while traversing we dynamically divide the left and right subtrees into groups, such that the priorities can be defined explicitly. We draw tangents on the separating curve from the viewpoint and thus, the whole region is divided in some groups, the number of regions depending on the number of tangents from the viewpoint to the curve. Now, separate queues indication priorities in each region are created and the queues are displayed in order, depending on the position of the viewpoint (fig 5.1).

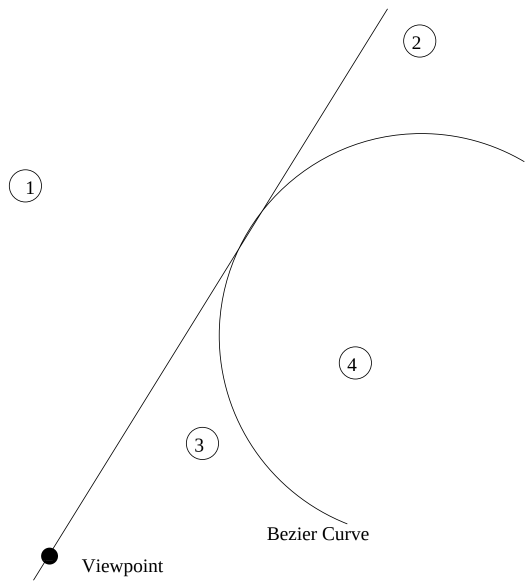

Figure 5.1: Partition Tree for Two Dimensions

In the figure, the Bezier curve is at the root of the partition tree and suppose the part on convex side of the curve is as left subtree and the part on the concave side is as right subtree. So, we traverse the left subtree and make three queues for each of region 1, 2 and 3. Before traversing the subtree, we have empty queues for each region and while traversing the tree as we get any object, we find which region or regions among these it lies and put the object is the queues of regions, where it lies. If it lies is more than one regions, then it is fragmented, such that one fragment lies in one region only and then each fragment is put in the respective queue of the region it lies. Hence, we will have queues for region 1,2,3 and we will get the queue of region 4 by traversing the right subtree. Now, we display queues in the order, 2,1,4 and 3. There are many other viable combinations, which will give same result, which is correct priorities. However, for optimization, we can merge queues for region 1 and 2 or for region 1 and 3. Thus, we will avoid some extra calculation. This optimization is possible as display in object 1 is independent of display in any other region.

5.2 The 3-D Algorithm

The algorithm described above for the 2-D case is not applicable for the 3-D case because the concept of a tangent plane dividing the outer partition of a curve into distinct regions is not applicable to the 3-D case. In a 3-D case we have patches instead of curves, and the analog of tangents would be tangent planes. Such a tangent plane would intersect the patch at one unique point and not a line. Thus a region around that point would be created but that would not be split into distinct parts as in the 2-D case. The 2-D case can be obtained by slicing the patch and the tangent plane along a plane perpendicular to the tangent plane and passing through the point of intersection. This plane perpendicular to the tangent plane has properties equivalent to the 2-D case.

Hence in the 3-D case we first try to split each patch inside the tree into at most two patches w.r.t the viewpoint. We work under the assumption that a single patch does not curve around

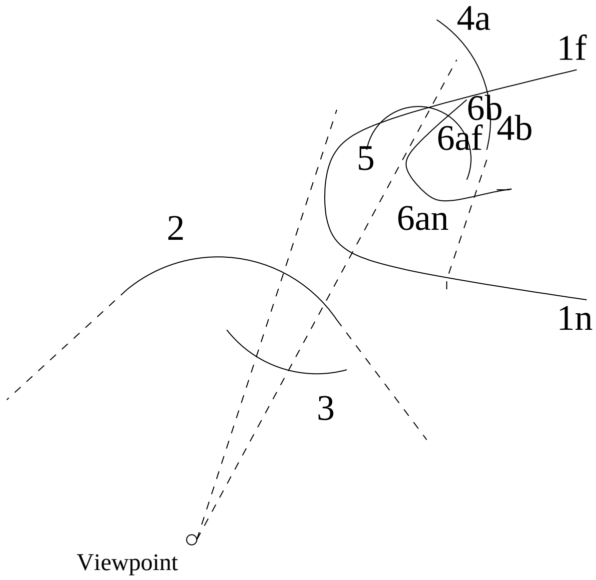

Figure 5.2: 3D scene with curved surfaces

for an angle greater than 180^0. We compute a boundary for each patch, i.e the curve along which the normal to the surface and the lines joining the curve to the viewpoint make right angles. This would give us a list of patches which do not obscure themselves or other patches by mere folding of the patch. Now we can obtain a list of such patches which when displayed in a correct sequence would give us a view of the scene as required. We start the description of the algorithm by first elaborating the BSP tree structure, followed by algorithms for visibility and boundary determination.

5.2.1 The Data Structure

The binary space partition tree representing the scene contains a bezier patch at each node of the tree. The left subtree contains all patches lying on the convex side of the extended patch (by extended patch, we mean the patch extended infinitely in all directions). The right subtree contains all patches lying on the concave side of the patch.

5.2.2 The Hidden Surface Removal Algorithm

For finding the correct sequence in which to paint the patches we do the following -

1. We first generate a sequence (back to front) of the patches in the convex-sided subtree, l_x, and the concave-sided subtree, l_v, of the root node.

2. Now, if the view point lies on the concave side of the patch in the root node, it implies that this patch would not have any boundary w.r.t the viewpoint. Also, however we choose the view volume, the patch in the root node does not obscure any of the patches in l_v. Similarly any of the patches in l_x do not obscure any of the patches in l_v. So firstly, we do not need to split the patch at the root node, and secondly, the back to front ordering is l_x, followed by the patch at the root node, followed by l_v.

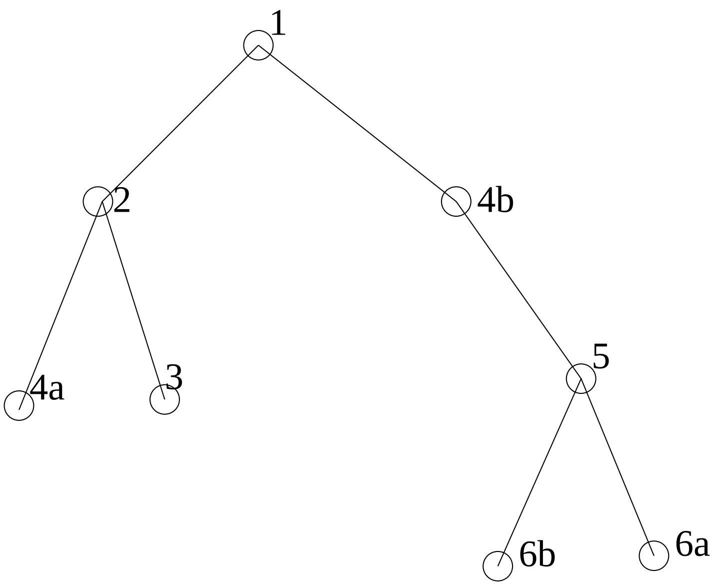

Figure 5.3: BSP Tree for Curved Surfaces in Tree Dimensions

3. If the view point lies on the convex side of the patch at the root node, the patch may have a boundary w.r.t the viewpoint. Also, this patch and those in l_v need to be inserted in l_x. So, we first compute the boundary of the patch at the root node and name the two resulting splits as p_f (for the one farther away from the viewpoint) and p_n (for the one nearer to the viewpoint). We now consider the patches in l_x.

   Note that if the line joining one of the control points of a patch and the view point cuts either p_f or p_n, it implies the patch lies behind the patch at the root node w.r.t the viewpoint. Since, we already split patches about their boundaries, it cannot obscure the patch at the root node by self-folding. Hence this patch and all patches before it in l_x need to be painted before either the patch at the root node, or any of the patches in l_v. So, we find all such patches in l_x, such that the line joining either of their control points with the viewpoint intersects either p_f or p_n. We take the patch closest to the view point and call it p_c. The new back to front list would be all elements of l_x in sequence upto p_c, followed by p_f, followed by l_v, followed by p_n, followed by the remaining elements of l_x in sequence.

4. If the root node is the leaf node as well, we split the patch at the node into two along its boundary w.r.t the view point and the resulting list is the patch away from the viewpoint followed by patch close to the viewpoint.

5.2.3 Algorithm to compute the boundary of the patch

To compute the boundary of the patch, we initially try to find a single point on the boundary of the patch and its corresponding projection in the image plane. We then consider the neighbourhood of the point computed in the image plane and try to find the point which corresponds to a boundary point. We keep computing the boundary until it is entirely covered (i.e. the

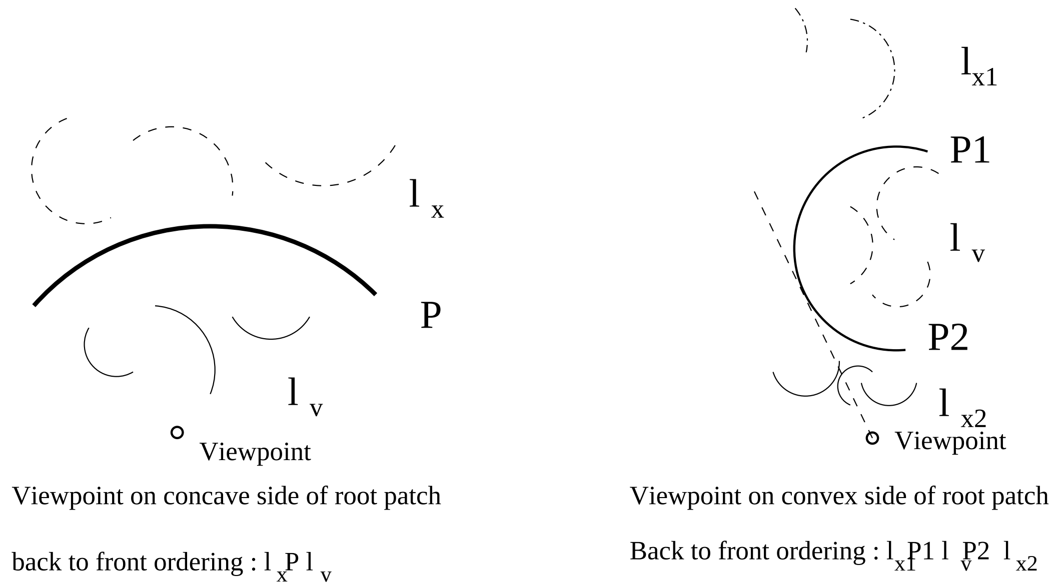

Figure 5.4: Determining Priorities

limits of the parameters of the curve are reached). The following approaches can be applied for computing a point on the boundary.

1. The boundary of a patch is a curve which is a collection of points which may be considered as the points of intersection of curves with tangents drawn to them, where each curve lies on a plane passing through the viewpoint and intersecting the image plane. So, finding a single point on the curve can be **reduced from 3-D to 2-D**. Therefore, we choose a plane which passes through the viewpoint and intersects the patch somewhere. We can choose another point for the plane on the patch itself. What remains of the patch on this plane is a curve. We can now find a tangent to this curve. The intersection of this curve with its tangent gives the boundary point on the curve while its projection is given by the intersection of this tangent with the image plane.

2. The above method for finding a boundary point turns out to be cumbersome when went about analytically, in the case of parametric curves, although it might be easier in case of implicit equations. So, we use the method of **binary search**. For a parametric curve in u and v, we span over the range of u to find a value at which the normals at $v = 0$ and $v = 1$ lie on the opposite sides of the boundary. Once we find such a u, we do a binary search over v to find the boundary point.

   Note that the failure to find a suitable u indicates non-existence of a boundary.

5.2.4 Summary and Improvements possible

To summarize the entire algorithm, we first construct the BSP tree using extensions of the available patches. While displaying the scene we traverse the tree as described above. We generate ordered lists of the subtrees and split the root patch if required. We then merge the lists and the root patch to form a new list representing the combined order of painting.

Improvements can be done over and above the above algorithm. Instead of painting back to front, we can paint front to back and mark the pixels if they have been already visited. While computing a boundary, we may take this into account and not compute all boundaries. Also, re-painting the same pixels may be avoided using this technique.

Chapter 6

Conclusions

The problem of rendering curved surfaces using polygonal approximations has been studied in detail, but directly using parametric representations for rendering has not been a topic of study in the past. The study is relevant to dynamic scenarios since we require the updation time to be as small as possible. In our approach we have come up with an appropriate data structure for such a scenario. Rendering of objects with such a structure avoids the use of techniques like *z-buffering* or *ray tracing*, which do not use any information of the previous scene to compute the new scene. Thus we get faster updations.

Possible Modifications

The technique we have used for hidden surface removal is the Painter's algorithm. This implies that if more than one object has its projection on a pixel in the image plane, then we need to compute the intensity of that pixel as many times. To avoid this we can use a front to back traversal of the BSP tree instead of the standard back to front traversal. We then need to maintain information whether a pixel's final intensity has already been computed or is yet to be computed. One technique for this is to maintain a dynamic clipping structure for the tree. In this structure, we maintain information about what all surfaces are hidden under a particular surface in a particular frame of the scene. This information is dynamically updated. Every time, we just paint that portion of the image for which the covering object has moved or changed. This would eliminate the overhead introduced by the painter's algorithm. One possible development would be to devise this clipping structure for curved objects and integrate it with the present BSP implementation.

Appendix A

BSP Trees

A.1 Introduction

A Binary Space Partitioning (BSP) tree is a standard binary tree used to sort and search for polytopes in n-dimensional space. The tree taken as a whole represents the entire space, and each node in the tree represents a convex subspace. Each node stores a **hyperplane** which divides the space it represents into two halves, and references to two nodes which represent each half. In addition, each node may store one or more polytopes.

It is common to see BSP trees which represent two and three dimensional space, but the definition of the structure is not constrained to these. For these two cases, the polytope stored is a line segment and a polygon, respectively.

A Binary Space Partitioning (BSP) tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.

A **hyperplane** in n-dimensional space is an n-1 dimensional object which can be used to divide the space into two half-spaces. For example, in three dimensional space, the **hyperplane** is a plane. In two dimensional space, a line is used.

BSP trees are extremely versatile, because they are powerful sorting and classification structures. They have uses ranging from hidden surface removal and ray tracing hierarchies to solid modeling and robot motion planning.

A.2 Construction of BSP Trees

Given a set of polygons in three dimensional space, we want to build a BSP tree which contains all of the polygons.

The algorithm to build a BSP tree is very simple:

1. Select a partition plane.
2. Partition the set of polygons with the plane.
3. Recurse with each of the two new sets.

A.2.1 Choosing the partition plane

The choice of partition plane depends on how the tree will be used, and what sort of efficiency criteria we have for the construction. For some purposes, it is appropriate to choose the partition plane from the input set of polygons. Other applications may benefit more from axis aligned orthogonal partitions.Choosing planes between middle of two objects can also be useful.

In any case, we want to evaluate how our choice will affect the results. It is desirable to have a balanced tree, where each leaf contains roughly the same number of polygons. However, there is some cost in achieving this. If a polygon happens to span the partition plane, it will be split into two or more pieces. A poor choice of the partition plane can result in many such splits, and a marked increase in the number of polygons. Usually there will be some trade off between a well balanced tree and a large number of splits.

A.2.2 Partitioning polygons

Partitioning a set of polygons with a plane is done by classifying each member of the set with respect to the plane. If a polygon lies entirely to one side or the other of the plane, then it is not modified, and is added to the partition set for the side that it is on. If a polygon spans the plane, it is split into two or more pieces and the resulting parts are added to the sets associated with either side as appropriate.

Partitioning a polygon with a plane is a matter of determining which side of the plane the polygon is on. This is referred to as a front/back test, and is performed by testing each point in the polygon against the plane. If all of the points lie to one side of the plane, then the entire polygon is on that side and does not need to be split. If some points lie on both sides of the plane, then the polygon is split into two or more pieces.

The basic algorithm is to loop across all the edges of the polygon and find those for which one vertex is on each side of the partition plane. The intersection points of these edges and the plane are computed, and those points are used as new vertices for the resulting pieces.

Convex polygons are generally easier to deal with in BSP tree construction than concave ones, because splitting them with a plane always results in exactly two convex pieces. Furthermore, the algorithm for splitting convex polygons is straightforward and robust. Splitting of concave polygons, especially self intersecting ones, is a significant problem in its own right.

A.3 Hidden Surface Removal

Probably the most common application of BSP trees is hidden surface removal in three dimensions. BSP trees provide an elegant, efficient method for sorting polygons via a depth first tree walk. This fact can be exploited in a back to front **painter's algorithm** approach to the visible surface problem, or a front to back scan-line approach.

BSP trees are well suited to interactive display of static (not moving) geometry because the tree can be constructed as a pre-process. Then the display from any arbitrary viewpoint can be done in linear time. Adding dynamic (moving) objects to the scene in efficient time was a major part of out project.

A.3.1 Painter's algorithm

The idea behind the painter's algorithm is to draw polygons far away from the eye first, followed by drawing those that are close to the eye. Hidden surfaces will be written over in the image as the surfaces that obscure them are drawn. One condition for a successful painter's algorithm is that there be a single plane which separates any two objects. This means that it might be necessary to split polygons in certain configurations.Therefore, cyclic polygons have to be treated separately by splitting the polygons to remove cyclicity.

One reason that BSP trees are so elegant for the painter's algorithm is that the splitting of difficult polygons is an automatic part of tree construction.

To draw the contents of the tree, perform a back to front tree traversal. Begin at the root node and classify the eye point with respect to its partition plane. Draw the subtree at the far child from the eye, then draw the polygons in this node, then draw the near subtree. Repeat this procedure recursively for each subtree.

A.3.2 Scan-line hidden surface removal

It is just as easy to traverse the BSP tree in front to back order as it is for back to front. We can use this to our advantage in a scan-line method method by using a write mask which will prevent pixels from being written more than once. This will represent significant speedups if a complex lighting model is evaluated for each pixel, because the painter's algorithm will blindly evaluate the same pixel many times.

The trick to making a scan-line approach successful is to have an efficient method for masking pixels. One way to do this is to maintain a list of pixel spans which have not yet been written to for each scan line. For each polygon scan converted, only pixels in the available spans are written, and the spans are updated accordingly.Storing the current picture in a pixmap and copy the final figure in the output window also solves the problem to some extent, because use of I/O devices, which are slow and critically determine the delay, is minimized here.

A.4 Making BSP Tree More Efficient

A.4.1 Optimal trees

Construction of an optimal tree is an NP-complete problem. The problem is one of splitting versus tree balancing. These are mutually exclusive requirements. You should choose your strategy for building a good tree based on how you intend to use the tree.

A.4.2 Minimizing splitting

An obvious problem with BSP trees is that polygons get split during the construction phase, which results in a larger number of polygons. Larger numbers of polygons translate into larger storage requirements and longer tree traversal times. This is undesirable in all applications of BSP trees, so some scheme for minimizing splitting will improve tree performance.One such scheme as used in our project is to sort all polygons, such that polygons with least numbers of cuts,from all other polygons, is inserted last.This reduces splitting to a large extent.

Minimization of splitting requires pre-existing knowledge about all of the polygons that will be inserted into the tree. This knowledge may not exist for interactive uses such as solid modeling.

A.4.3 Tree balancing

Tree balancing is important for uses which perform spatial classification of points, lines, and surfaces. This includes ray tracing and solid modeling. Tree balancing is important for these applications because the time complexity for classification is based on the depth of the tree. Unbalanced trees have deeper subtrees, and therefore have a worse worst case.

A.4.4 Balancing vs. splitting

If balancing is an important concern for your application, it will be necessary to trade off some balance for reduced splitting. If you are choosing your hyperplanes from the polygon candidates, then one way to optimize these two factors is to randomly select a small number of candidates. These new candidates are tested against the full list for splitting and balancing efficiency. A linear combination of the two efficiencies is used to rank the candidates, and the best one is chosen.

Bibliography

[Cla76] . H. Clark. Hierarchical geometric models for visible surface algorithms. Communications of ACM, 19(10):547–554, 1976.

[Hun78] . Hunt. Efficient Computation and Data Structure for Graphics. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, 1978.

[FKN80] . Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priority tree structure. In Proceedings of ACM SIGGRAPH, pages 124–133, 1980.

[Che95] .E. Chen. QuickTime vr – an imagebased approach to virtual invi ronment navigation. In Proceedings of ACM SIGGRAPH, pages 29–38, 1995.

[MB95] . Mcmillan and G. Bishop. Plenoptic modeling: An image based rendering system. In Proceedings of ACM SIGGRAPH, pages 39–46, 1995.

[LH96] . Levoy and P. Hanrahan. Light field rendering. In Proceedings of ACM SIGGRAPH, pages 31–42, 1996.

[GGSC96] . Gortler, R. Grzeszczuk, R. Szelski, and M. Cohen. The lumigraph. In Proceedings of ACM SIGGRAPH, pages 43–54, 1996.

[For79] .R. Forrest. On the rendering of surfaces. ACM Computer Graphics, 13(2):253–259, 1979. (SIGGRAPH Proceedings).

[Cat74] . Catmull. A Subdivision Algorithm for Computer Display of Curved Surfaces. PhD thesis, University of Utah, 1974.

[Baj90] .L. Bajaj. Rational hypersurface display. ACM Computer Graphics, 24(2):117–127, 1990. (Symposium on Interactive 3D Graphics).

[Foley] oley, van Dam, Feiner, Hughes, Computer Graphics- Principles and Practice, Addison-Wesley Publishing Company, 471-530 , 695-718

[Ihm91] hm, I., and Naylor, B., Piecewise linear approximations of digitized space curves with applications, Scientific Visualization of Physical Phenomena (Proceedings of CG International '91), 545–569, 1991.