715.

Shadow Computation for 3D Interaction and Animation

by

Yiorgos Chrysanthou

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
Queen Mary and Westfield College
University of London

January 1996

To my parents

Ὁ βίος βραχύς, ἡ δέ τέχνη μακρή

Ἱπποκράτης

ABSTRACT

The presence of shadows in an image helps viewers to better understand the spatial relationships between objects, is vital for interactive applications such as Virtual Reality, and, in general increases the appearance of reality that a picture provides. Many shadow algorithms have been devised that adequately solve the problem, ranging from the very simple (point light sources and local illumination) to the very detailed and realistic (Radiosity and Ray-tracing). None of these existing algorithms, however, is suitable for interaction, at least on the standard hardware, because they require a total recalculation of the shadows for any change in the scene geometry.

In this thesis various methods are proposed for providing shadows in dynamic scenes, illuminated by point or area light sources. These methods exploit the temporal and spatial coherence present in interactive environments to provide incremental updates to the shadow information in a fraction of the time required by other algorithms.

A data structure used by all shadow algorithms in this thesis, because it provides an efficient space partitioning and searching tool, is the BSP tree. One of the perceived problems of the BSP trees is that they are only suitable for static scenes. An investigation is made into the use of BSP trees for representing dynamic scenes and practical solutions are suggested.

Using the results of the above study, two methods are presented for shadows in dynamic scenes illuminated by point light sources. The first uses a regular space subdivision by means of a tiled cube placed around the source. The second uses a Shadow Volume BSP tree built from the set of unsorted polygons.

For area light sources an algorithm is presented which is a combination of a tiling cube similar to that used for the point sources and the discontinuity meshing (DM) method used in Radiosity. The shadow boundaries as well as other irregularities in the illumination function of each surface in the scene are found and built into a mesh using BSP tree merging. The combination of the space subdivision provided by the tiling and the structured mesh building provided by the merging lead to a significantly faster DM algorithm compared the previous methods, which in addition allows for incremental updates after a change in the scene geometry.

Experimental results have shown that near real-time frame rates can be achieved using the above methods on commonly used workstations which do not have specialised 3-D graphics hardware.

Acknowledgements

First of all, I would like to thank my supervisor, Mel Slater, for his indispensible guidance and friendship, not only during my Ph.D. years but throughout my studies, since my early days as an undergraduate. I would also like to thank him for introducing me to Computer Graphics and for sharing with me his profound knowlledge on the subject.

I wish to acknowledge the *Committee of Vice-Chancellors and Principles, U.K.* and the University of London for providing financial support for the largest part of my study.

I would like to thank Erricos, for helping me out in a lot of difficult occasions and for making sure I never missed a tea break. The rest of the Ph.D. students in what used to be the ACE lab, who have been to me like a second family in England (Guan, Sarom, Shancy, Anthony, Adelene, Mark, Abdulahi, Amela, Alaster), and the staff of the Computer Science Department at QMW for making it such a friendly place.

A special thanks to Isabelle, for putting up with me and keeping me sane during the writing of my thesis, and for teaching me that "efficiency" is not only for shadow algorithms.

Finally, I would like to thank my parents. Without their love, support and encouragement I would have never made it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are two strands to modern Computer Graphics research: photorealism and real-time. Ideally one would like to have both at the same time but in general a trade off is essential. With graphics hardware becoming common and Virtual Reality interactive applications demanding real-time realistic images, the need to bring the two together is ever more apparent.

One of the key components of photorealism is the correct simulation of illumination. A very important characteristic of light is that it can be blocked and produce shadows. The computation of shadows requires determination of the relative positions of the modeled objects and the sources of light. This can be an expensive operation and in general is omitted from interactive applications. At best some shadows are rendered as crude approximations.

The absence of shadows is one of the burdens of modern interactive graphics applications. This is not only because it makes images stand out as "computer generated", immediately dispersing any sense of realism or "being there", but it also makes the task of interaction much harder. Shadows can give vital clues for the spatial relations of the objects which are essential for tasks like object placement. Relevant research in the field of Virtual Reality [93] supports this statement.

The major purpose of this thesis is to show that interactive 3-D graphics including shadows is feasible, without special hardware, on standard workstations. We take the Sun SPARCstation2 as our standard here.

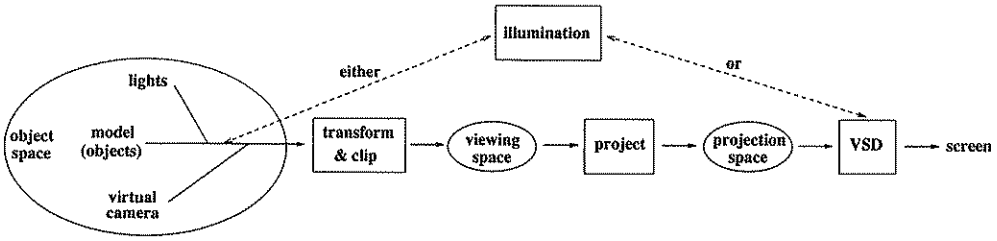


Figure 1.1: The graphics pipeline

1.1 Viewing Pipeline in Computer Graphics

To put the reader in the context of the work in this thesis, we give a brief description of the process for producing an image. This is summarised in Figure 1.1.

First a model is created by giving a mathematical description of its primitives, where by primitives we mean the polygons or curved surfaces that make up the model. The algorithms studied in this thesis work only on convex polygons so we will restrict our models to them. This is not a real limitation as everything else can be reduced to or be approximated by them and also they form the standard basic unit of most graphics hardware and software platforms. Polygons are assumed to have a front and back side which are conventionally specified by the order of their vertices. When viewed from the front the vertices of a polygon are arranged in a counter-clockwise order. The back side of a polygon is not visible to an observer but one should never need to see the back as we assume the polygons to be in groups forming closed polyhedra, with the polygons facing outside, away, from the volume they enclose. So when viewed from the outside, a back facing polygon is blocked by a front facing polygon of the same polyhedron. An object is made up of one or more polyhedra and the model is a collection of objects.

The objects are defined in a coordinate system particular to the model, we call this the *object space.*

In object space we also define the light sources, if any, by specifying their geometry and intensity. In the first graphics images created and in some of the simpler ones used today there are no light sources, the scenes are illuminated by an ambient light.

The *virtual camera* is defined by giving its position, direction and orientation. This represents the viewer in world coordinates and at the same time defines a new coordinate system (*viewing space*) with origin at the camera position and

axis orientation defined by the camera orientation.

As a viewer in real life has a limited field of view, so in our camera model we add some *clipping planes* that define the limits of what is visible, from the camera (*viewing volume*).

The objects are transformed to the viewing coordinate system clipping away any parts falling outside the viewing volume. Once in viewing space they then have to be projected onto the screen. This can be done either by projecting directly to the 2-dimensional display coordinates using the camera position as center of projection, or by transforming into a third space called *projection space* (or *box space*) which is equivalent to the 2-D display coordinates except that it has depth. Then the standard graphics viewing pipeline performs the *visible surface determination* (VSD) to decide which parts of the polygons are visible and which are blocked by others closer to the camera [31].

This whole process from the camera definition to the final display is called the graphics pipeline. In most modern interactive systems it is repeated in its entirety for any change in the model or in the camera parameters.

1.2 Illumination

One of the decisive factors for the realism of an image is how faithfully the transport of light is modeled. In the physical world light radiates from some emitting surface (light source) and travels in a straight path until it reaches another surface. There it is partially absorbed, partially reflected and, sometimes, partially transmitted making this other surface a new, lower intensity, light source.

The illumination algorithms can be classified into two categories depending on how far they go into modeling the above behavior:

Local illumination algorithms: These are only concerned with the light arriving directly from the principle light sources. They do not account for light reflected from other surfaces. To compensate for the loss of the reflected energy an ambient term is added to the intensity of each surface.

Global illumination algorithms: These provide more realistic images by accounting, as much as possible, for the transition of light between objects during illumination.

Another measure for classification that is often used is when the illumination

takes place in the graphics pipeline. As shown in Figure 1.1, this can be done in two places giving two classes:

Object space algorithms: The illumination calculations are done for the whole model without taking in account the viewpoint. The calculated intensities can be used for display from any camera position.

Image space algorithms: The illumination is done after the viewpoint is defined and the visible sections of the scene determined. Typically a change in the viewpoint requires recalculation of the illumination.

Object space algorithms are better suited for applications where the camera is likely to change often since the solution is given for the whole scene and it is stored in object space. On the other hand, image space methods tend to give faster results for a single image since the computation is concentrated only on the relevant parts of the image.

Part of the illumination is the determination of the areas that receive no direct light from at-least one of the sources (in shadow). The shape and appearance of the shadows depends heavily on the geometry of the light source. In the real world light sources have non-zero area and the shadows have edges that change gradually from dark to lit (penumbras). These penumbras are computationally difficult and expensive so in many applications the sources are represented by mathematical points. In this case the problem of finding the shadows reduces to a VSD problem with the source acting as viewpoint. The resulting shadows (umbras) have sharp edges with no transitional "grey" area.

1.3 Scope and Objectives

In this thesis we investigate methods for incorporating shadows into interactive applications by employing spatio-temporal coherence techniques that build upon an initial pre-computation to achieve real-time performance. The applications targeted will require multiple views of the model from different viewpoints so the algorithms involved here are all object space solutions. We also assume that the models consist of planar convex polygons and that direct illumination is sufficient.

Even though we aim to account only for light arriving directly from the light source, i.e. local illumination (*page 17*), we acknowledge the importance of inter-reflections. Thus the solution for area light sources presented in *Chapter 5* can form the basis of an efficient global illumination model.

To enable fast updates of the shadow information we use space subdivision techniques to localise the operations. These are based on two main ideas:

1. The Tiling Cube [92], which uses a regular subdivision of the space as seen from the point of view of the light by means of a cube with a grid on each side.

2. The BSP tree [34], which uses a hierarchical binary partitioning of space using the planes stored at its nodes. It can be used for visible surface determination if the planes of the scene polygons are used for the partition or for shadow determination if the shadow planes (the planes bounding the volume in shadow behind a polygon) are used for the partition. This method was perceived to be only well suited for applications, like walkthroughs, where the camera moves continuously but the model does not change. This is because as soon as the geometry of the model changes then the whole tree had to be recomputed. Later in this thesis we show that this is not necessarily true (see *Section 3.1*).

1.4 Contributions

The overall contribution of this thesis can be summarised as: a demonstration that real-time generation of images including correct shadows is achievable on workstations without specialised hardware. More specifically the contributions are:

1. A study in the use of BSP trees for dynamically changing scenes, with practical solutions suggested.

2. Using the results of this study, two algorithms for calculating shadows from point light sources were extended for use in dynamic scenes. The first one using the Tiling Cube and the second using a Shadow Volume BSP tree. The implementation of both algorithms showed that fairly complex scenes, can be maintained in near real-time on what we can refer to as common workstations.

3. Finally, an efficient method for finding shadows from area sources in dynamic scenes has been developed. The shadow boundaries (umbra and penumbra), as well as any edges where major irregularities in the illumination value occur, are found. The model polygons are divided along those

boundaries and irregularities with each resulting cell holding a reference to the polygons occluding its view from the source. This allows for a fast and accurate computation of the direct illumination as well as a structured way of incrementally updating this information during interaction. The implementation of this algorithm indicates that interactive frame rates can be achieved for moderately complex scenes on higher-end workstations.

1.5 Organisation of the Thesis

This thesis is organised as follows.

In *Chapter 2* we review previous related work: BSP trees, local and global illumination methods using point and area light sources. The limitations and suitability for interaction are discussed.

In *Chapter 3* we investigate methods for the use of BSP trees to represent dynamic models as well as for producing an invariant ordering in respect to an area light source.

In *Chapter 4* we describe how the Shadow Tiling and SVBSP tree point source shadow algorithms can be extended to accommodate moving objects. Evaluations and comparisons of the algorithms are given at the end of the chapter.

In *Chapter 5* we present a new discontinuity meshing algorithm that uses space subdivision and BSP tree merging and is suitable for interaction.

In the concluding chapter we summarise the accomplishments of this work and provide guidelines for future work.

Chapter 2

BSP Trees and Shadows

The focus of this thesis is on algorithms for calculating shadows in dynamic scenes. A fundamental choice for any algorithm is the underlying data structure. The data structure used throughout this thesis, either for representing the initial model, the shadow planes or the illumination discontinuities, is the Binary Space Partitioning (BSP) tree. The reason for this choice will become apparent as we describe each method. It is therefore important to overview previous work related to both shadow algorithms and the BSP trees.

2.1 Binary Space Partitioning Trees

The Binary Space Partitioning (BSP) tree algorithm was developed as an efficient method for solving the visible surface determination problem [34]. Based on Schumacher's work [89], on ordering linearly separable static sets, it uses an initial pre-processing step to give a linear display algorithm. It was later shown that BSP trees can be used to represent polyhedra and perform set operations upon them [103, 73]. Currently they are widely used for applications ranging from motion planning [104] to image representation [85] and shadow generation. In this thesis we will be concerned with their use for shadow generation. This also involves visible surface determination as well as combining BSP trees together (merging).

2.1.1 Definitions

The BSP tree is a hierarchical subdivision of n-dimensional space into homogeneous regions, using (n-1)-dimensional hyperplanes.

A hyperplane h (line in 2-D and plane in 3-D) is defined by an expression of the form $f_n = a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n + a_{n+1}$. The set of points in space that make $f_n > 0$ define the front (or positive) half-space of h (h^+), while those that make $f_n < 0$ define the back (or negative) half-space of h (h^-). The points $f_n = 0$ are on the hyperplane.

The BSP tree is stored in a binary tree structure. Each node t on the tree corresponds to a region in space (subspace), denoted by $r(t)$. Each internal node holds a hyperplane h_t that partitions the region at that node into front and back subspaces. The two subspaces are represented by the left ($t.front$) and right ($t.back$) "children" of the node. A leaf node, in contrast to an internal node, holds no hyperplane and corresponds to an unpartitioned region of space, which we call a *cell.*

The root node of a tree corresponds to the whole of the n-dimensional space. The region ($r(t)$) of each other node is defined by the intersection of the open half-spaces determined by the hyperplanes associated with the previous nodes on the path to the node t. To be more precise, given a node t_i at depth i which lies along the path $\{t_0,...,t_i\}$, where the subscripts denote depth, if $i = 0$ then $r(t_0) = \Re^n$, otherwise if t_i is the front child of t_{i-1} then $r(t_i) = h^+_{t_{i-1}} \cap r(t_{i-1})$ else $r(t_i) = h^-_{t_{i-1}} \cap r(t_{i-1})$.

The intersection of the hyperplane h_t of a node t with its region $r(t)$, is called the sub-hyperplane ($shp(t)$). Notice that since $r(t_0)$ is unbounded, so is $shp(t_0)$. Also any other $r(t_i)$ and $shp(t_i)$ may only be partly bounded.

In this thesis we will use BSP trees in 2-D and 3-D spaces. The following description on building and using the tree for various tasks is given for a 3-D environment but as the concepts behind the BSP trees are dimension independent the same is valid for 2-D.

2.1.2 Building a Tree

Given a set of polygons $S = \{s_1, .., s_n\}$, we can construct a BSP tree and partition 3-D space using the following simple recursive algorithm. A polygon is selected from S; the plane defined by this polygon is used to make the root of the tree and to partition space into front and back half-spaces. A reference to the polygon is also stored on the node. The rest of the polygons are compared against this plane and depending on which side they lie they are placed into two sets, *front* and *back.*

Any polygon lying partly in both subspaces is split along the intersection with the plane and the two fragments are placed in the corresponding sets. Any polygon found to be coplanar with the root plane is stored at the root, Figure 2.1.

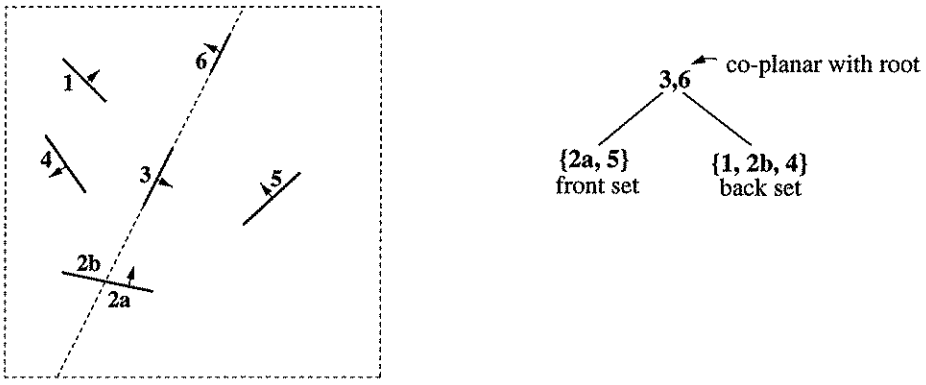


Figure 2.1: Partitioning space and the polygons with a polygon-plane

A polygon is selected from each of the two sets and its plane forms the root of the corresponding subtree that further subdivides space and the rest of the polygons. This is repeated until all polygons have been used and the original space has been subdivided into homogeneous regions (cells). In Figure 2.2 the cells are labeled (a to f) and shown as leaves on the tree. In general we will not show the cells on the tree and we may refer to the last internal nodes as leaves. Whether a 'leaf' is a cell or the last internal node splitting into empty subspaces, will always be clear by the context.

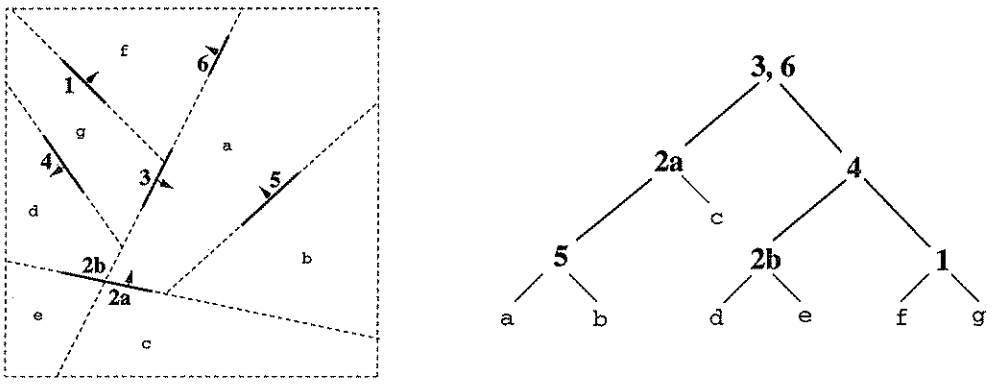


Figure 2.2: A complete subdivision

An alternative (incremental) approach for building a BSP tree from a set of polygons was suggested by Thibault and Naylor [103]. Each polygon is inserted in turn into the, initially empty, tree until it reaches a cell. The plane of the polygon then defines a node that splits the cell in two. Inserting a polygon into the tree is again a recursive procedure that compares the polygon against the

root plane and sends it into the appropriate subtree. As before polygons might be split by the root plane or stored at the root if they are coplanar.

Both ways of building the tree have the same complexity. The advantage of the latter is that it allows for polygons to be added to the tree after it has been built. This is useful for performing incremental changes to the tree (see *Section 3.1*).

Selecting the appropriate root polygon at each iteration can be crucial for the efficiency of the resulting tree. Some efficiency issues are discussed in *Section 2.1.5.*

2.1.3 Visible Surface Determination

Given a BSP tree such as the one built above in Figures 2.1 and 2.2, we can use it to solve the visible surface determination problem by traversing it from any given viewpoint to get the back-to-front order of the polygons stored at the nodes. The polygons can then be displayed in that order using over-painting to cover hidden surfaces.

```
void displayTree(Tree node, 3DPoint viewpoint)
{
        if (viewpoint in-front of node plane)
                displayTree(back subtree, viewpoint);
                draw polygons on the node;
                displayTree(front subtree, viewpoint);
        else
                displayTree(front subtree, viewpoint);
                draw polygons on the node;
                displayTree(back subtree, viewpoint);
        endif
}
```

Figure 2.3: Traversing the tree to get a back-to-front order

The traversal is based on the fact that given a viewpoint and two sets of polygons separated by a plane, the polygons on the same (*near*) side as the viewpoint can obstruct but cannot be obstructed by polygons on the other (*far*) side. So to get the reverse, back-to-front, order from the tree the simple recursive algorithm of Figure 2.3[1] can be used: compare the viewpoint against the root plane, traverse the far subtree first then display the root polygon(s) and then traverse near the subtree.

When the shading of the polygons is defined by a complex function, the multiple over-painting of each pixel performed by the above algorithm could be costly.

[1]The pseudocode notation is explained in *Appendix A*

This can be avoided by using the scan-line algorithm suggested by Gordon [47]. The tree in this method is traversed in front-to-back order. As the polygons are displayed the filled segments at each scan-line are recorded. These segments are never overwritten and the algorithm terminates when the whole screen has been covered, or when the tree is fully traversed.

2.1.4 Merging BSP Trees

A very popular modeling method is constructive solid geometry (CSG). In CSG polyhedra are combined by means of boolean set operations (*union, intersection and difference*) to form more complex objects. The description of the polyhedra and objects is usually by boundary representations. The use of this representation has many drawbacks such as being able to deal only with closed sets and requiring different data structures or complex algorithms to perform the different operations (spatial search, model modifications, rendering).

Naylor and Thibault [103, 73] presented an alternative way of representing and combining polyhedra that is simple and efficient and also allows for open sets.

Thibault described how a BSP tree can be used to represent any arbitrary polyhedron [103]. This is done by giving an IN or OUT value to each leaf node depending on whether the corresponding cell is inside or outside the polyhedron. A simple example can be seen in Figure 2.4. The grey area corresponds to the inside of the polyhedron which is denoted by the IN cells of the tree on the right.

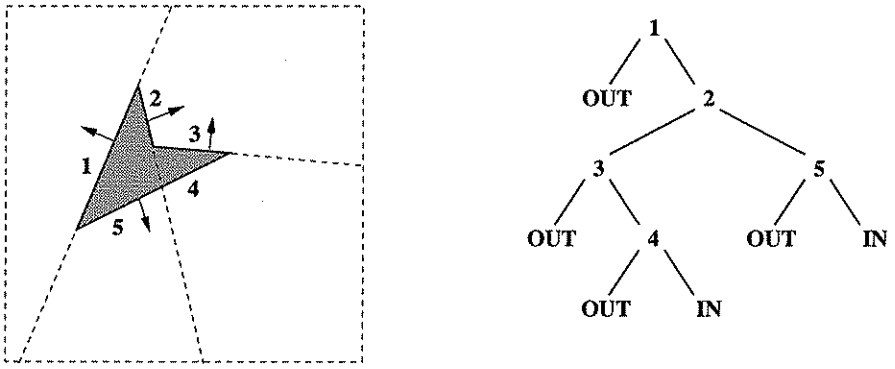


Figure 2.4: Representing a polyhedron by a BSP tree

Given two polyhedra P_1 and P_2 represented as BSP trees, T_1 and T_2, any boolean set operation can be performed on them by merging their trees [73]. Merging the partitionings of space, induced by T_1 and T_2 produces a third par-

titioning, T_3, that includes the two first. The values of the cells of the new partitioning depend on the operation used. The merging process however is always the same.

Merging T_1 and T_2 can be seen as inserting T_2 into T_1. In principle this is similar to the way we inserted the polygons in the tree during the incremental construction: starting at the root of T_1, T_2 is inserted recursively into T_1 until it reaches the leaves (cells) of T_1. At each step of the recursion, T_2 is compared against the plane h_{t_1} of each node t_1 of T_1 and is split into T_2^+ and T_2^-, where T_2^+ is the intersection of T_2 with the front half-space of h_{t_1} and T_2^- with the back half-space. Then T_2^+ is inserted in $t_1.front$ and T_2^- in $t_1.back$. Once it reaches a cell, an external routine is called that combines T_2 and the cell.

```
Tree merge(Tree t1, Tree t2)
{
        if (leaf(t1) or leaf(t2))
                return treeOpCell(t1, t2);
        endif

        {t2+, t2-} = partitionTree(t2, shp(t1));
        t1.front = merge(t1.front, t2+);
        t1.back = merge(t1.back, t2-);
        return t1;
}
```

Figure 2.5: Merging two trees

op	Cell	Tree	Cell <op> Tree
$\cap$	IN	t	IN
	OUT	t	t
$\cup$	IN	t	t
	OUT	t	OUT
—	IN	t	~t
	OUT	t	OUT

Table 2.1: Combining a cell and a tree

The pseudocode for the merging is given in Figure 2.5. Comparing and splitting the tree by the plane of a node is performed by the function *partitionTree* which we will explain shortly. Combining a tree and a cell is done by *treeOpCell*. This depends on the set operation being performed. In general this routine will return either the cell or the tree or the complement of the tree (denoted by ~t). The complement of the tree is found by reversing the attributes of its leaves, IN becomes OUT and OUT becomes IN. In this thesis only the *union* operation is used but the full table of the resulting values for all operations is given in Table 2.1.

If there are other attributes involved such as colour, texture etc, then these also have to be merged. How this is done depends on the application. An example of trees augmented with additional values at the cells can be seen in *Chapter 5*. There each cell holds a list of the polygons that occlude it from the source. In the *treeOpCell* function when the tree is retained, the list of occluders from the cell is added to the cells of the tree.

Partitioning a Tree with a Plane

As T_2 is inserted into T_1, at each node t_1 it is partitioned by the plane h_{t_1}, into T_2^+ and T_2^-. This partitioning is a recursive procedure that involves inserting h_{t_1} into T_2. Here again we differentiate between leaf nodes and internal nodes. When inserting h_{t_1} into T_2, if T_2 is a cell then T_2^+ and T_2^- are just copies of that cell. If not then three steps are performed:

1. h_{t_1} is compared against h_{T_2} to find their relative positions,
2. the subtrees of T_2 in which h_{t_1} lies are partitioned,
3. the resulting subtrees of the above partition are combined to form T_2^+ and T_2^-.

For step 1 the important thing to notice is that h_{t_1} and h_{T_2} are not defined over the whole of 3-D space but rather in the subspace formed by the intersection of $r(t_1)$ and $r(T_2)$. So the relation of h_{t_1} and h_{T_2} that we need is with respect to this region. Since h_{t_1} and h_{T_2} are infinite planes we need to find their intersections with the region in consideration. This intersection is what we earlier called the sub-hyperplane (*shp(t)*). The sub-hyperplanes are represented as polygons and their relative position is determined by comparing one against the plane of the other.

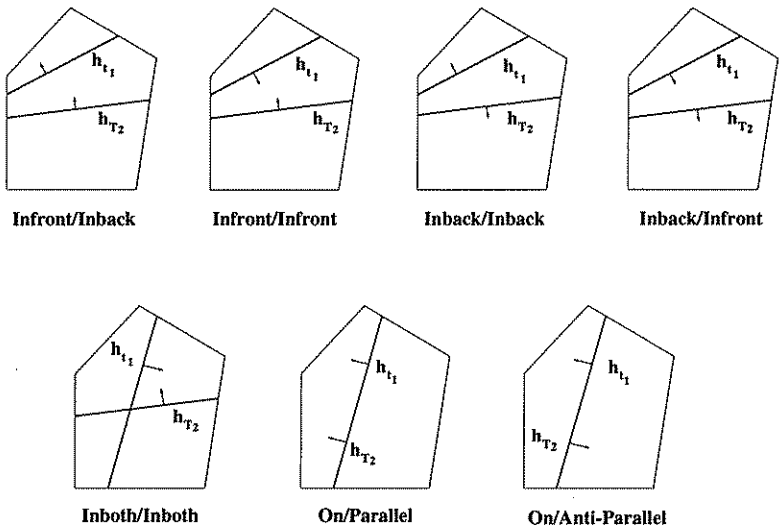


Figure 2.6: The sub-hyperplane of t_1 in respect to the sub-hyperplane of T_2

One problem with this approach is that $r(t_1) \bigcap r(T_2)$ may be open (if for example t_1 and T_2 are near the root), then the sub-hyperplanes will be open.

The solution, as suggested by Thibault in [103], is to represent 3-D space as a bounded set, for example, as a large enough bounding box containing the model. Any hyperplane is first clipped against this box to form a polygon and then is intersected against the node planes.

There are 7 possible classifications between the two sub-hyperplanes, which can be grouped into three sets (in one subtree, in both, coplanar), shown in Figure 2.6.

Each classification has two parts, the first is found by comparing $shp(t_1)$ against h_{T_2} and it shows the subtree of T_2 in which $shp(t_1)$ lies, possibly in both subtrees. The general idea is that only the subtrees in which $shp(t_1)$ lies will be partitioned while the others will be left unchanged.

For example in the case where we have *Infront/Inback* (Figure 2.7(a)) the front subtree of T_2 is partitioned to give $T_2.front^+$ and $T_2.front^-$ while $T_2.back$ remains unpartitioned. In the *Inboth/Inboth* case of Figure 2.8, both subtrees of T_2 are partitioned. In the case where h_{t_1} and h_{T_2} are coplanar no subtree is partitioned.

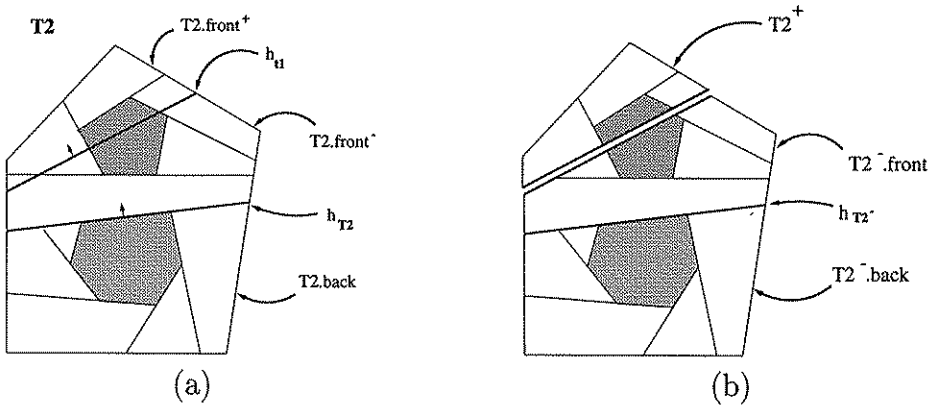


Figure 2.7: Infront/Inback (a) h_{t_1} partitions $T_2.front$ into $T_2.front^+$ and $T_2.front^-$ and (b) T_2^- and T_2^+ after partitioning

The third step is to put the pieces resulting from the partitioning of T_2 together to make T_2^+ and T_2^-. For each of the three sets of classifications we proceed as follows:

- $shp(t_1)$ falls entirely in one side h_{T_2}. For this we also need the second part of the classification, $shp(t_2)$ against h_{t_1}. For the case of *Infront/Inback* the resulting trees are (Figure 2.7(b)):

  $$T_2^-.front = (T_2.front)^-$$

$$T_2^-.back = T_2.back$$

and

$$T_2^+ = (T_2.front)^+$$

the other three cases are analogous.

- for the *Inboth/Inboth* case the two new trees are (Figure 2.8(b)):

$$T_2^+.front = (T_2.front)^+$$
$$T_2^+.back = (T_2.back)^+$$

and

$$T_2^-.front = (T_2.front)^-$$
$$T_2^-.back = (T_2.back)^-$$

- when the two hyperplanes are coplanar:
  if they are parallel and facing the same direction then

$$T_2^+ = T_2.front$$
$$T_2^- = T_2.back$$

otherwise (anti-parallel)

$$T_2^+ = T_2.back$$
$$T_2^- = T_2.front$$

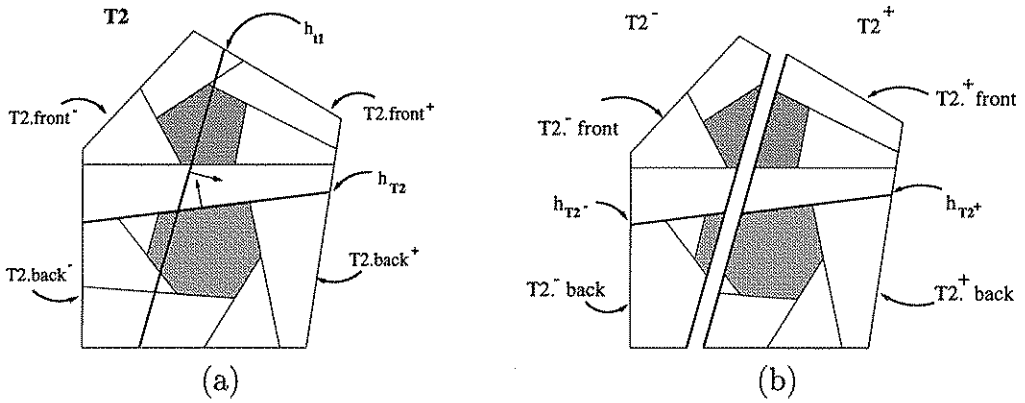


Figure 2.8: Inboth/Inboth (a) h_{t_1} partitions both $T_2.front$ and $T_2.back$ and (b) T_2^- and T_2^+ after partitioning

As we said at the beginning of the section, BSP merging provides a fast and simple way of combining polyhedra, open or closed. In the following chapters this will be a very useful facility for combining shadow volumes as well as other data.

2.1.5 Efficiency Considerations

The issue of creating trees that can be used efficiently is a problematic one, this is because there is no single notion of "efficiency".

As shown by Paterson and Yao [80, 81] for a set of n initial polygons the upper bound for space and time complexity for building a BSP tree is $O(n^2)$, although the expected case is closer to $O(nlogn)$. There can be great variation depending on the partitioning polygon selected as the root at each iteration.

One method that is often used for controlling the size and shape of a tree is to select a few candidate polygons at each iteration and find the best of these to use as root. The evaluation is done by comparing them against the rest of the polygons in the subspace and computing the weighted sum of two quantities, size (number of resulting splits) and distribution (difference in the number of polygons in each of the resulting subsets).

The weights used depend on the application. For visible surface determination the balance of the tree is not important, since every node is only visited once, but the size is very important. On the other hand for ray tracing or algorithms involving classifications, balance is more important than size. Also balanced trees are generally faster to build (if the number of splits created is not overwhelming) even though this doesn't reflect the run-time performance.

A different measurement of efficiency, based on expected cost of various operations given by probability models, is presented by Naylor [72]. The idea is to keep the largest cells (with a great probability of being visited) on shorter paths and the smaller cells on longer paths. In a sense this is a sequence of approximations similar to bounding volumes. This method builds trees well suited for merging since in effect the objects are wrapped with the minimal number of sub-hyperplanes extending outside.

Another problem with the existing algorithms for building efficient trees is that they are static in nature. For example, the method of sampling to select a suitable root for each subtree assumes that we have all polygons already at our disposal and also that once a selection is made it is *permanent* in the sense that it cannot be changed when better knowledge is acquired, without rebuilding the

subtree involved [64].

In most of the algorithms presented in thesis we are not so much concerned with building efficient trees. However, for the algorithms involving merging it is very important that the trees involved are "efficient". By efficient in this case we mean Naylor's probabilistic efficiency [72].

2.1.6 BSP Trees in Dynamic Scenes

The non-applicability of BSP trees to dynamic scenes has been a problem that researchers from the very infancy of the algorithm have tried to solve. The earliest work, even though not as yet on BSP trees, was that of Schumacker [89]. In his algorithm, which is considered to be the predecessor of the modern BSP trees, the tree was built using manually defined separating planes between objects. Each of these objects would have its faces sorted into a visibility order valid from any viewpoint (after back-face elimination). The ordering between the objects as seen from each tree cell was pre-calculated and stored and so at run time locating the position of the viewpoint was all that was needed, to get the priority order. In this structure the objects were allowed to move, without any recalculation of the tree, as long as they did not cross any of the separating planes.

The first partial solution for dynamic changes of BSP trees was given by Fuchs [32]. If we know in advance the objects that will be moving and the region in which they will do so then a tree can be constructed such that the relevant region is enclosed in a tree cell. Then the objects can move in that region independently with regard to the rest of the tree.

A different method that again involves knowing the objects that will be moving in advance but not their path, was used by Naylor [69, 70]. A tree of the static objects is built which is merged with the tree of the moving object at each frame to produce a complete scene tree. No removal is ever necessary since the original copy of the static tree is used for merging at each frame.

Torres [105] presented a BSP tree with several optimizations over the standard structure. Each object has its own single BSP tree which is built by considering the polygons of the object alone. These single trees form the leaves of the scene BSP tree of which the internal nodes are separating planes between the objects. If such separating planes cannot be found then user defined partitioning planes are required. To make the single trees of convex objects more balanced and speed up their construction, *halving planes* are sometimes used, which are planes

chosen to be parallel to as many object polygons as possible to minimise splits and positioned so as to have an almost equal number of the polygons on each side.

Also wrapping planes are sometimes used over complex objects in order to minimise the splitting of objects that are not linearly separable. Speed ups are obtained over the initial building of the tree and the moving of certain objects but the general idea as far as interaction is concerned is not much different from that of Schumacker.

None of these algorithms are general enough. Also the theoretical question of efficiently removing nodes from an existing tree still remains. Some solutions to this problem are given in *Section 3.1.*

2.2 Shadows from Point Light Sources

Modeling the light source as a mathematical point is usually the only viable way for producing, shadows when speed is important. The shadow problem is then reduced to a binary decision for each point on a surface in the scene, whether or not it is visible from the source. A decision which is simple to make and fast to compute. But as a result the shadows have sharp boundaries which give an artificial look to the image.

Several algorithms have been developed for calculating sharp shadows (umbras) and following the classifications of Crow [28] and Bergeron [9] they can be grouped into 5 classes:

Scan line algorithms: The shadow computation is performed as the image is rendered by raster scanning. The edges of the potential shadowing polygons are projected, from the light source as centre of projection, onto the polygon being scanned. These shadow edges then mark changes in colour on the scan segments, [3, 11].

Two pass hidden-surface methods: As the first step the scene polygons are used for a visible surface determination algorithm, with the light source as the viewpoint. In this step the fragments of the scene polygons that are visible (lit) and invisible (shadowed) are found, and the shadowed fragments are added as detail polygons onto the original. The scene can then be rendered from the camera viewpoint using the same or a different visible

surface determination algorithm [5, 75]. The shadow calculations are only performed once for any series of images as long as the objects and the light source do not move.

Shadow Volumes: As described by Crow [28] the shadow volumes (the volume of space not seen by the light source, Figure 2.9(a)) are computed in world space but shadowing is left for the rendering stage. The shadow planes (the planes bounding a shadow volume, see *Section 2.2.1*) are added to the data as invisible surfaces and during the scan conversion of the polygons, each point is checked for containment in the shadow volumes. Extensions of this method [17, 92, 20, 21] calculate all shadows in object space and add them to the scene polygons as details as in [5]. More detailed description of the latter methods is given in *Section 2.2.1*.

Two pass Z-buffer: Two Z-buffers are used, one from the point of view of the light source and the other from the camera position [112, 56, 86]. First the image is rendered into the light source buffer, the light-buffer. Each entry of the light-buffer holds the distance of the object closest to the light source at that point, and hence lit. The second Z-buffer is used for rendering. Each point of the second buffer is mapped to the light-buffer: if the distance of the point is the same as that stored in the light-buffer then the point is lit otherwise it is in shadow.

Backwards Ray tracing: The path of the ray from the centre of projection through a pixel is traced and the closest intersecting object is found. To determine if the object-intersection point is in shadow, a ray is traced from there to the light source (shadow-ray). If an object is found that blocks the shadow-ray then the point is in shadow otherwise it is lit [3, 45]. This method can be easily extended to point-wise approximation of area light sources by sending multiple shadow rays.

Almost all of these methods can deal with multiple light sources by repeating the related calculations once for each source, or by having a separate pass for each source and using the output of the previous pass as input for the current.

The point source algorithms studied in this thesis are based on shadow volumes and therefore a more detailed description of this class follows. General reviews of shadow algorithms can be found in [28, 114, 31].

2.2.1 Shadow Volumes

Given a light source L and a polygon P defined by the vertices $\{v_1, v_2, ..., v_n\}$, a *shadow plane* is the plane defined by a triple (L, v_i, v_{i+1}) where $i = 1...n$ and $n+1 = 1$. The orientation of this plane is outwards, i.e. it has P behind it.

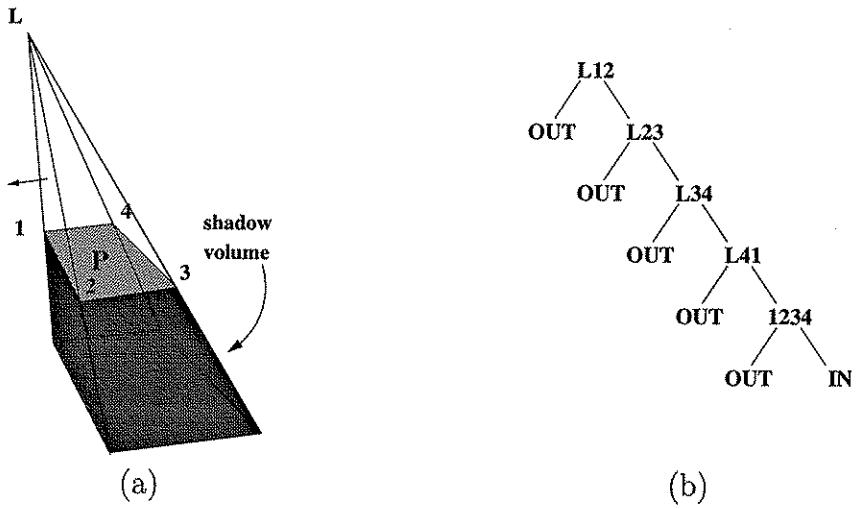


Figure 2.9: (a) A shadow volume and (b) its representation as a BSP tree

The shadow volume (SV) of P is then defined as the frustum enclosed by the shadow planes of (P, L) and bounded on top by P, Figure 2.9(a).

Shadow volumes can be used for whole objects instead of just for individual polygons. In such a case the shadow planes are defined by the contour edges of the object as seen from the light source.

In the algorithm described by Crow in [28] the shadow planes are included in the scene data and the shadow calculations are performed during rendering. Since a polygon representation of these planes was necessary they were bounded by clipping against the field of view or the sphere of influence of the light. In applications where multiple views of the scene are required the shadow calculations are repeated from each viewpoint. For such applications it is more efficient if all shadows are determined in object space and the results combined with the scene data before rendering.

The shadow between two polygons, an occluder (O) and a receiver (R), can be found by clipping R against the shadow volume of the O. Any part of R falling within the volume is in shadow. This shadow can either be represented as a detail polygon on top of R or by splitting R, using the shadow edges, to lit and shadowed pieces.

The brute force method for performing the calculations in object space would be to compare each, of say n, polygons facing the light against the shadow volume of each other such polygon. This, however is wasteful. An improvement to this n^2 algorithm can be achieved by observing that only a polygon closer to the light source can cast a shadow on one further away. The polygons can be sorted by building a scene BSP tree and traversing it in a front-to-back manner from the light position. The polygons can then be processed in this order and each one only needs to be compared against the SV of those before it. This method can reduce the number of comparisons involved, if the number of splits produced by the BSP tree is not too large, but much better performance can be achieved by using space subdivision. Two such methods are described below. In both of these methods the shadow volume of a polygon is represented as a BSP tree by assigning an *OUT* value to the front cells at each node and an *IN* to the back cell at the last node, Figure 2.9(b).

2.2.2 Shadow Volume BSP Tree

The Shadow Volume BSP (SVBSP) tree algorithm was introduced by Chin and Feiner in [17]. It uses a non-regular subdivision of space based on a BSP tree, but instead of the planes of the scene polygons it uses the shadow planes as partitions.

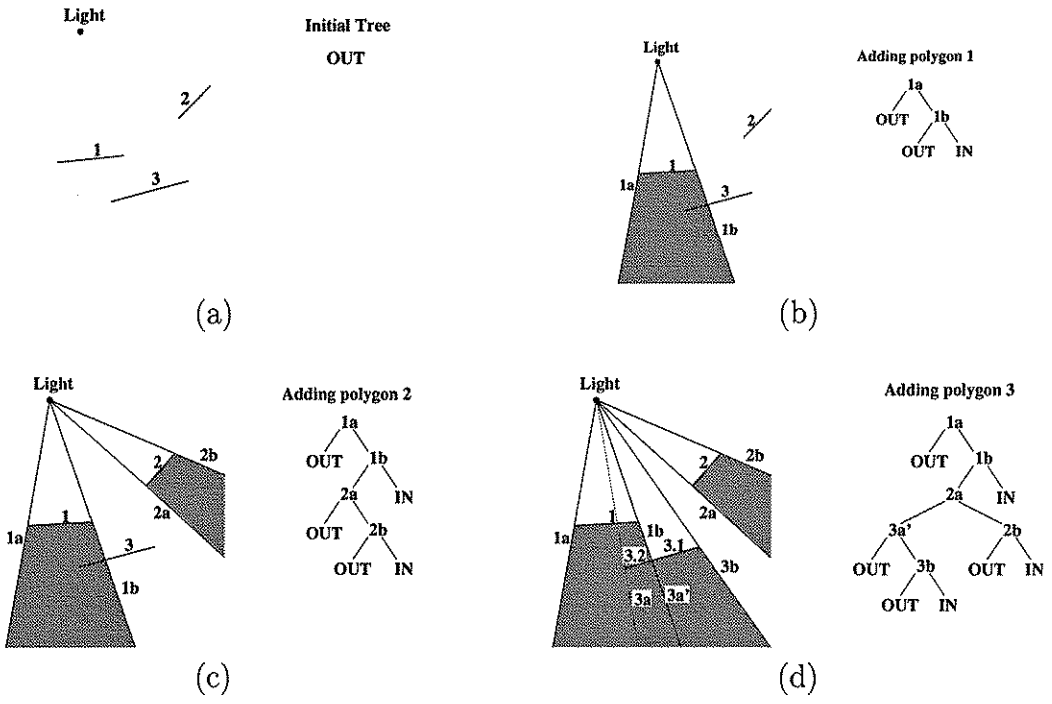


Figure 2.10: Building the SVBSP tree

A unified shadow volume of all the light facing polygons is built incrementally by inserting the front-to-back ordered polygons into an initially empty tree (Figure 2.10). This is done in a similar manner to the *Union* operation described in [103] with some important differences. The tree is not enlarged by adding the polygon plane when a fragment of the polygon ends up in an *OUT* node but it is enlarged with the shadow planes defined by the edges of that fragment. Such a fragment is classified as "lit". Any fragment reaching an *IN* node is classified as shadowed but does not change the tree (polygon 3.2 in Figure 2.10(d)). The polygons themselves do not need to be included into the tree since the order of processing ensures that they will always be behind those already in the tree. For ordering the polygons, typically, a BSP tree is used. A consequence of this is that there is an increase in the set of polygons. This can be avoided by including the scene polygons themselves in the SVBSP, see *Section 4.2* and [21]. For multiple light sources a different tree is built for each source which can be discarded before going to the next.

```
void buildSVBSP(Tree bsp, Light light)
{
      /* the svbsp tree is initially set to null */
      svbsp = OUT;
      /* the BSP gives the back-to-front order */
      order[] = traverseBSP(bsp, light);
      /* each polygon is inserted in order */
      for i = 0 to n do
            svbsp = insert(svbsp, order[i], light);
      endfor
      free (svbsp); /* discard tree */
}
```

```
Tree insert(Tree svbsp, Polygon poly, Light L)
{
      if (cell(svbsp)) /* svbsp is a cell */
            if (svbsp == IN)
            /* polygon in IN cell, in shadow */
                  add poly as a shadow polygon;
            else
            /* polygon in OUT cell, lit. expand tree*/
                  return constructSV(poly, L);
            endif
      else
            /* find which side of the root is polygon */
            classifyPolygon(svbsp.rootplane, poly, pf, pb);
            if (notNull(pf))
                  svbsp.front = insert(svbsp.front, pf, L);
            endif
            if (notNull(pb))
                  svbsp.back = insert(svbsp.back, pb, L);
            endif
      endif
      return svbsp;
}
```

Figure 2.11: Pseudocode for building the SVBSP tree

The pseudocode for this process is shown in Figure 2.11. In function *buildSVBSP* the SVBSP is first initialised to a single *OUT* node, then the polygons are ordered using the scene BSP tree and they are inserted into the SVBSP in that order.

A polygon is inserted using the recursive function *insert*. At each call of *insert* the SVBSP is checked, if it is an *IN* cell then the polygon is marked as shadowed. If the tree is an *OUT* cell then the cell is replaced by the shadow volume of the polygon, otherwise the polygon is classified against the plane at the root of the tree and sent to the appropriate subtree. At the end of the proceedure the tree

is deleted.

2.2.3 Shadow Tiling

The Shadow Tiling method presented by Slater in [92] uses a regular subdivision of space. A cube, with each of its faces subdivided into a rectangular grid, is placed around the light position. The polygons are projected onto the sides of the cube using the source as centre of projection. Any two polygons can have a shadow relation only if their projections overlap on at-least one side of the cube, Figure 2.12. Of course, since the grid elements (tiles) have non-zero area, polygons may share tiles and yet have no shadow relation.

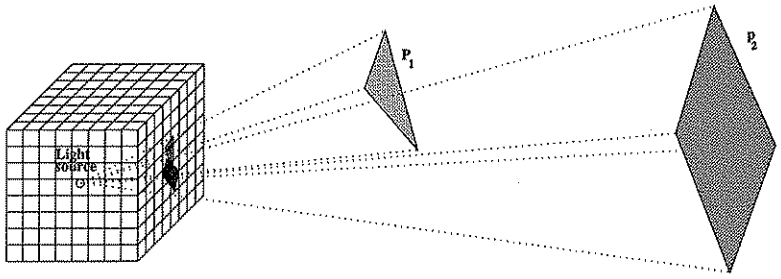


Figure 2.12: Polygons with overlapping projections on the cube have a shadow relation

The polygons are processed in front-to-back order from the light source. As each polygon P is scan converted onto the cube, its identifier is stored in the data structure representing the tiles. At the same time a list of all the polygons that are already stored in the tiles visited by P, is constructed. This list is called the *active polygon list* (APL) of P. Then P only needs to be compared against the shadow volume of the polygons in the APL.

This is summarised in Figure 2.13. After constructing the tiling cube, the BSP tree is traversed to get the order of the polygons as seen from the light position. Each polygon (P_i) is scan converted onto the cube sides, as the polygons that share tiles with it are found (the APL). Then, in the inner *for* loop, P_i is compared against the shadow volume of each of the polygons in the APL to find the shadows, if any.

Certain optimisations can be used to speed up this method. One is to make a distinction between tiles that intersect the boundary of a projection (*boundary tiles*) and those that are covered completely by the projection (*interior tiles*). Interior tiles correspond to a volume of space that is completely in shadow so

```
void shadowTiling(Tree bsp, Light light)
{
        /* first the tiling cube is constructed */
        tc = constructTilingCube(light);
        /* the BSP gives the back-to-front order */
        order[] = traverseBSP(bsp, light);
        /* each polygon is projected in order */
        for each polygon p_i in order[] do
                apl_{p_i}[] = projectOnCube(tc, p_i, light);
                /* apl[] (active polygon list) holds the polygons that */
                /* share tiles with the polygon. It will be */
                /* compared against the SV of each one of these*/
                for each polygon p_j in apl_{p_i}[] do
                        castShadow(p_i, p_j);
                endfor
        endfor
}
```

Figure 2.13: Shadow calculation using the tiling cube

there is no need to add more polygon identifiers to them. Any subsequent polygon that scan converts to interior tiles only, is fully in shadow.

A significant part of the processing required by this algorithm is taken up by the projection of the polygons onto the six sides of the cube. This can be improved by projecting only to the relevant sides. If for example the light source is outside the scene then the cube can be reduced to just one face. If it is not, a method described by Haines in [50] can be used: before projecting a polygon, one of its vertices is first projected and the side of the cube intersected is found. The polygon is projected onto this side and on any adjacent sides sharing an edge that was used to clip out part of the polygon.

A detailed evaluation and comparison of the SVBSP and Shadow Tiling methods can be found in [92]. In brief, as far as speed is concerned the two algorithms are almost equivalent (depending on the architecture of the machine used). But the number of output shadow polygons of the tiling method is considerably smaller than that of the SVBSP.

2.2.4 Point Light Sources in Dynamic Scenes

As already stated, shadows are very desirable in most computer generated images and especially so when these are part of an interactive system.

None of the existing algorithms are fast enough to be used at interactive speed for complex environments. The standard solution used in interactive applications is to use "fake" shadows [10]. These are just projections of the polygons onto a ground plane without clipping or checking for obstruction. In particular no inter-object shadows are considered. This problem is considered in *Chapter 4*.

2.3 Shadows from Area Light Sources

Shadows from point sources provide a lot of information about the spatial relations of the objects in the scene. In the real world most of the light sources have a non-zero area. To add to the realism of the images the effect of such sources should be modeled. Shadows due to area sources have soft edges, they are no longer defined by a singular sharp boundary (umbra), but also have partially lit areas (penumbra). A more precise definition of umbra and penumbra is given in *Section 2.3.1.*

The algorithms for finding shadows from area light sources can be classified into two broad categories:

Point sampling: The source is treated as a collection of point sources. The visible part of the source is taken as the proportion of point sources visible from the given position. This has been done using shadow volumes and a depth buffer [12], using hardware supported depth buffers [48] and ray casting. The latter is probably the most common with some impressive results in the context of Ray Tracing [26]. Campbell in [14] also used point sampling with shadow volumes before turning to an explicit calculation of the shadow boundaries for a faster and more accurate solution.

Analytical determination: With the exception of Amanatides's method [2], that analytically computes shadows from circular or spherical sources, the other methods in this category are restricted to models made entirely of planar polygons, including the source. The shadow planes are formed explicitly and traced into the scene to find the boundaries of the shadows on the model. This is done either by tracing each such plane separately [53, 61, 30, 37] or processing the set of shadow planes from a single occluder as a shadow volume [76, 22], or by putting these volumes together into an SVBSP tree [15, 18]. The calculations are done in object space.

In general, point sampling techniques are computationally expensive, especially if an accurate solution is required, but they work for all object geometries.

In this thesis we are dealing with interactive applications where the viewpoint moves, as well as the objects. So we will only be concerned with the object space algorithms where the shadows are pre-computed in world coordinates and stored in order to be displayed from any viewing position. These algorithms fall in

the analytical determination category. We assume the environment to be made entirely from polygonal diffuse surfaces.

2.3.1 Extremal Shadow Boundaries

Shadows from area light sources consist of a totally blocked area, the umbra, and a partially blocked area, the penumbra. The boundaries between lit and penumbra and between penumbra and umbra areas are celled the *extremal boundaries* of the shadow.

The first to compute the exact extremal boundaries were Nishita and Nakamae [76]. They described how to build the penumbra and umbra volumes formed by the *extremal planes* (Figure 2.14). Assuming that all shadow planes face outwards, away from the shadow volume, then these extremal planes are:

For penumbra: Planes defined by a pair of (source vertex, occluder edge) or (source edge, occluder vertex) and have the source totally in the front half-space and the occluder on the back half-space (Figure 2.14),

For umbra: Planes defined by a vertex of the source and an edge of the occluder and have both the source and the occluder in their back half-space (Figure 2.14).

The umbra shadow volume then is the intersection of the back (negative) half-spaces of the umbra planes and the back half-space of the polygon. Similarly, the penumbra shadow volume is the intersection of the back half-spaces of the penumbra planes and the back half-space of the polygon.

The shadow boundaries, umbra and penumbra, on the scene polygons are computed in object space. This is done by comparing each receiver polygon against the penumbra shadow volume of every other polygon; if there is an intersection then the receiver is also compared against the umbra volume of that polygon. It should be clear that where we say "every other polygon" we only refer to the polygons that face, at-least partly, the light source and which lie, at-least partly, in front of it (we call these *light-facing* polygons). Any polygon that is behind or has the source behind it is irrelevant to any shadow algorithm.

Once the boundaries are found they are transformed to image space where they are used in the illumination of the polygons during rendering. While the image is scan-converted to the screen the intensity is calculated whenever a shadow

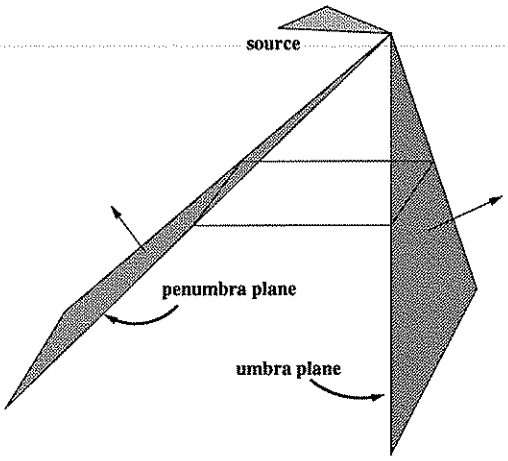


Figure 2.14: Extremal shadow planes

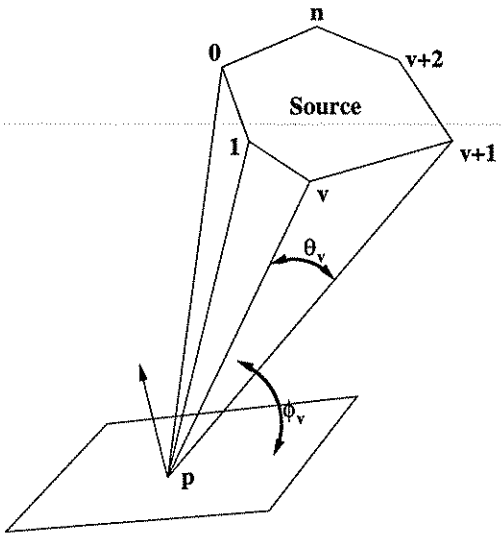


Figure 2.15: Illumination calculation from an area source

boundary is met and at regular intervals along the rest of the scan-line. The intensity of the unoccluded points is computed using *Equation 2.1*, explained in *page 42*. For points in the umbra this value is zero and only an ambient intensity is used. For points in the penumbra *Equation 2.1* is used on the visible parts of the source, from the point.

To find the visible parts of a source from a penumbra point p, first the set of occluders (O) of that point are found. Then for each polygon in O a pyramid using the polygon edges and with apex at p is constructed. This is very similar the point shadow volume described in *Section 2.2.1*, but with p as source. The source is compared against this pyramid and only the parts of it falling outside are visible. These visible parts are then compared against the pyramids of the rest of the polygons in O.

The shadow boundaries and illumination intensities are not explicitly stored in object space so whenever the viewpoint changes the whole process has to be repeated. Also the shadow boundary determination is an $O(n^2)$ process, where n is the number of light-facing scene polygons.

Campbell and Fussell [15] presented a more efficient algorithm that performs all the calculations in object space. The penumbra and umbra shadow volumes of all light-facing polygons are built as BSP trees and then put together to form two SVBSP trees, one for the penumbra and one for the umbra. The SVBSP trees are constructed by the shadow volumes using the BSP tree merging algorithm described in *Section 2.1.4*. Then each of the light-facing polygons is inserted into

the penumbra tree to determine the shadow boundaries. If it is found to lie even partly in an IN cell of the penumbra SVBSP then it is also tested against the umbra tree. The algorithm uses the two-pass process, constructing the SVBSPs and then inserting the polygons in them, because Campbell and Fussell failed to find a way of ordering the polygons with respect to the light source.

Chin and Feiner [18] dealt with the problem of ordering from an area by splitting the source whenever it is found to straddle the plane of a scene polygon. The scene BSP tree can then be unambiguously traversed from each resulting source fragment to give the front to back ordering. Two SVBSP trees are constructed for each source fragment and the shadows are calculated during the construction in a manner similar to the point SVBSP tree (*Section 2.2.2*).

Each of these latter methods have no means of knowing exactly which polygons are blocking the source at the penumbra vertices. Campbell and Fussell build the convex hull for each receiver and the source and clip all the scene polygons against it. The penumbra vertices on the receiver need only to test the source against the remaining polygons. Chin and Feiner use the BSP tree to find the set of polygons (O) that lie between the source and a receiver. For each penumbra vertex v_i on this receiver a point SVBSP tree is built using the polygons in O and v_i as the apex. Then the source is inserted into this tree and the visible parts are those reaching the OUT cells.

The visible parts of the light source as seen from a penumbra vertex, are calculated as convex polygons which can be used as separate sources with their sum giving the total illumination at that vertex.

The illumination I_p from a convex polygonal source with n vertices at point p is approximated by the following equation as described in [76]:

$$I_p = \frac{I_s}{2} \sum_{v=1}^{n} \theta_v \cos(\phi_v) \tag{2.1}$$

where

I_s = intensity of the light source

θ_v = the angle formed by source vertex v, p and source vertex $v+1$ (Figure 2.15)

ϕ_v = the angle between the plane on which p lies and the triangle v, p, $v+1$ (Figure 2.15)

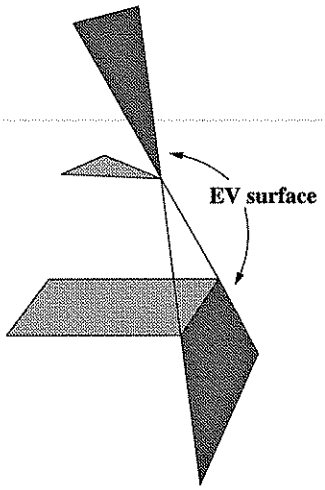


Figure 2.16: An EV surface is defined by an edge and a vertex

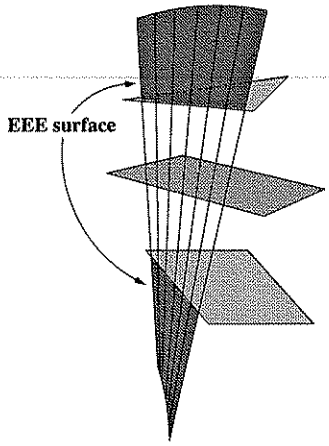


Figure 2.17: An EEE surface is defined by three non-adjacent edges

In all methods mentioned above only the umbra and penumbra boundaries are determined explicitly, but as Campbell notes the illumination function has maxima, minima and discontinuities within the penumbra regions. He used sampling to locate them.

2.3.2 Aspect Graphs

More insight into the problem of the variation within the penumbra, was gained through the aspect graph approach used for object recognition in computer vision. In this approach the 3-D objects in the scene are represented by a set of 2-D views and the viewpoint space is partitioned into regions such that in each region the qualitative structure of the line drawing does not change. The qualitative measure of the structure of each image is called the *aspect* [41]. By considering only views from the light source, that is finding the regions of space in which the visible part of the source is qualitatively constant, this becomes the same as the problem we are trying to solve.

In the aspect graph theory, the surfaces that bound these homogeneous regions are called *critical surfaces*, which when crossed produce *visual events*. The critical surfaces are defined by the interaction of edges and vertices in the scene and as described by Gigus in [42, 41] they can be grouped into two classes:

EV surfaces: The planes defined by an edge and a vertex, Figure 2.16,

EEE surfaces: The quadratic surfaces defined by three non-adjacent edges, Fig-

ure 2.17, [42].

For the shadow computation problem most of these surfaces are irrelevant. We are only interested in EV and EEE surfaces that contain a source feature (edge or vertex) and in EV and EEE surfaces that cut the source polygon, [29]. The intersection of these surfaces with the scene polygons generate critical curves which correspond to discontinuities in the illumination function. These critical curves are also called discontinuity curves (or edges for the EV events).

The penumbra volumes, as defined earlier, are made entirely of EV surfaces involving a feature of the source while the umbra volumes might consist of EV and EEE surfaces. All the discontinuities are enclosed by the penumbra.

2.3.3 Radiosity

In Computer Graphics the bulk of the work on computing and representing the exact structure of shadows has been carried out by researchers into Discontinuity Meshing Radiosity. To put this in context we will give a brief description of the Radiosity method in general before going into more detail in Discontinuity Meshing.

Radiosity is based on concepts from thermal engineering. First introduced into Computer Graphics by Goral in [46] it requires the assumption that the energy in a closed environment remains constant. It accounts for all inter-reflections among diffuse surfaces in the environment and specifies the resultant surface intensities independent of the viewpoint.

The environment is represented by a set of n discrete *patches* of finite size that can emit and/or reflect light uniformly over their entire area. The radiosity of each patch is given by

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \tag{2.2}$$

where

B_i = radiosity of the i^{th} patch

E_i = rate of energy emitted from the i^{th} patch

ρ_i = reflectivity of the i^{th} patch, the fraction of incident light reflected back into the environment

F_{ij} = form factor from patch i to patch j, the fraction of energy leaving patch i that lands directly on patch j

The radiosity of each patch depends upon the radiosity of all other patches in the environment, which results in a linear system of equations (*Equation 2.3*). In the *full matrix* radiosity solution the values are found by solving this matrix using an iterative algorithms such as the Gauss-Seidel.

$$\begin{pmatrix} 1 & -\rho_1 F_{1,2} & \dots & -\rho_1 F_{1,n} \\ -\rho_2 F_{2,1} & 1 & \dots & -\rho_2 F_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n,1} & -\rho_n F_{n,2} & \dots & 1 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{2.3}$$

This method for calculating the radiosities has many drawbacks. The whole matrix must be completely computed before any kind of image is produced which is a very time and space demanding process. An alternative, called *progressive refinement* radiosity, was proposed by Cohen in [23]. The single most bright patch is selected and it *shoots* its energy to all other patches in the scene. This is repeated with the next most bright patch (accounting for the received energy as well) and so on until the next most bright patch has an energy level below a threshold. This method converges more quickly, the scene can be drawn after each cycle and the process can be stopped when the result is satisfactory. Also only one matrix row needs to be stored at each step.

The most expensive part of every radiosity solution is the computation of the form factors and accounting for visibility between the patches. It is important that these computations are performed as efficiently as possible. The form factor is in a sense an expression of the visibility between two patches. For progressive radiosity where one emitting patch is processed at a time and hence the form factor between that patch and the rest of the environment is calculated, this is very much related to the problem of calculating the shadows from the emitting patch. One of the first and most popular solutions for finding form factors is the hemi-cube method [24]. This calculates the visibility using a z-buffer as space subdivision into cells, and approximates the form factors by counting the number of cells the projected patches cover. Alternative methods include ray casting [63, 109, 110] and the hemi-sphere [38, 95].

Another crucial factor for speed and accuracy in the radiosity solution is subdivision (meshing) of the surfaces. Traditional radiosity systems start with an initial mesh over the environment surfaces, usually a regular grid, and refine it

as more information about the intensities on the mesh is gained [25], sometimes requiring intervention from the user. Many different ways of refining the mesh have been suggested [6, 51, 94, 109]. A notable one is hierarchical radiosity [51] that automatically refines the mesh avoiding excessive accuracy where it would go to waste and thus speeding up the process by an order of magnitude.

This way of meshing the environment fails to account exactly for important intensity gradients such as shadows. Even though greater subdivision is usually performed in shadow areas, in general the subdivision is not aligned with the shadow boundaries. Many artifacts are produced as a result, such as *shadow/light leaks*, floating objects and jagged shadow edges.

Recent research has tackled this problem by building the initial mesh along the shadow boundaries and other important discontinuities before refining it any further. This method is called discontinuity meshing. At the same time as making a better mesh it also provides a means of calculating the exact visibility between the patches, accounting for more accurate form factors.

2.3.4 Discontinuity Meshing

A function f is said to be continuous (C^0) over an interval (t_1, t_2) if and only if

$$\forall a \in (t_1, t_2), \quad \lim_{x \to a-\epsilon} f(x) = \lim_{x \to a+\epsilon} f(x) = f(a), \quad as \ \epsilon \to 0 \tag{2.4}$$

A function that fails this criterion is said to have a zero degree (D^0) discontinuity. A function whose k^{th} derivative satisfies (2.4) is said to be C^k continuous. A function that is C^{k-1} but not C^k is D^k.

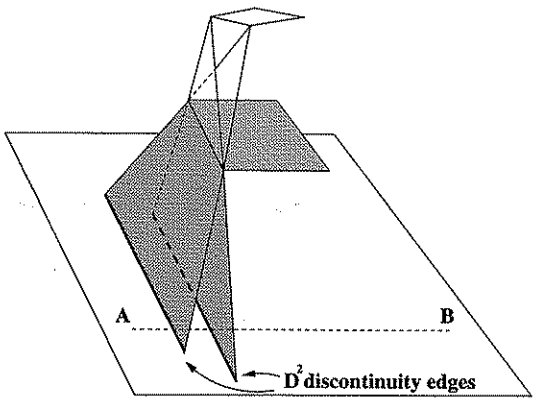


Figure 2.18: Discontinuities of the second degree (D^2)

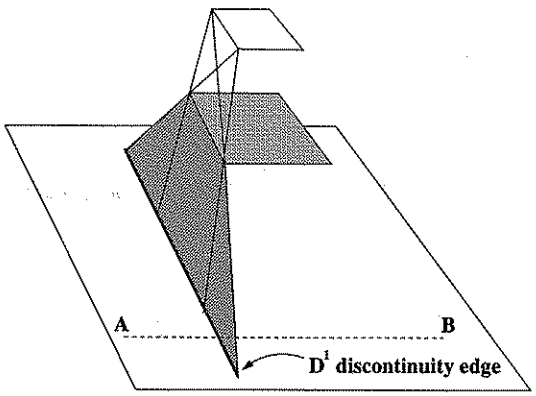


Figure 2.19: D^1 resulting from 2 overlapping D^2

Discontinuities in the illumination function of a polygon are caused at its intersection with the critical surfaces. As said before the only relevant critical events are those caused by EV or EEE surfaces involving a source edge or vertex, which are called *emitter events*, and those caused by EV or EEE surfaces not involving a source feature but intersecting the source with their surface, which are called *non-emitter events.*

As described by Heckbert [52] critical surfaces due to point, linear, area light sources cause in general D^0, D^1, D^2, respectively but when 2 discontinuities coincide the degree of discontinuity can decrease. As we are dealing with area sources, discontinuities from both EV and EEE surfaces will in general be of the second degree. For example in Figure 2.18 as we move along line AB on the polygon, the illumination function has D^2 discontinuities at the points where it crosses the marked edges. In cases where an edge of the source and an edge of the occluder are parallel then two EV surfaces have a combined effect and produce D^1 edges. In Figure 2.19 as we move along line AB on the polygon, the illumination function has a D^1 discontinuity at the point where it crosses the marked edge.

D^0 edges can also be generated. These occur along touching surfaces and they can cause some of the most severe artifacts if they are not represented explicitly. D^0 discontinuities were first correctly accounted for by Baum [6] by using a separate pass through the database to locate them before doing any further subdivision. Similar approaches have been adopted by all discontinuity meshing (DM) algorithms. This separate pass can, however, be avoided. Following the reasoning for D^1, D^0 can be seen as a set of co-linear D^2, (Figure 2.20), and can be treated through the same algorithm. The algorithm presented in *Chapter 5* of this thesis promotes to D^0 any edge where penumbra and umbra edges of the same occluder and source overlap.

Higher order discontinuities can be generated when the radiance of the source is non-uniform. This can be the case with secondary sources. In general a D^k on the source can cause D^{k+1} and D^{k+2} on the receiver [52]. Higher order discontinuities are less noticeable and anything above D^2 is not usually considered.

The first study on discontinuity meshing was presented by Heckbert [54] for a 2-D domain[2]. A complete mesh was constructed by considering every possible interaction between the edges and vertices in the scene, an operation with N^3 cost for N vertices. He later extended his work to a 3-D environment [53] by using

[2]In-fact Campbell [15] used subdivision along shadow boundaries (introducing the 2D BSP representation), before Heckbert, but he only considered extremal boundaries, see *Section 2.3.1.*

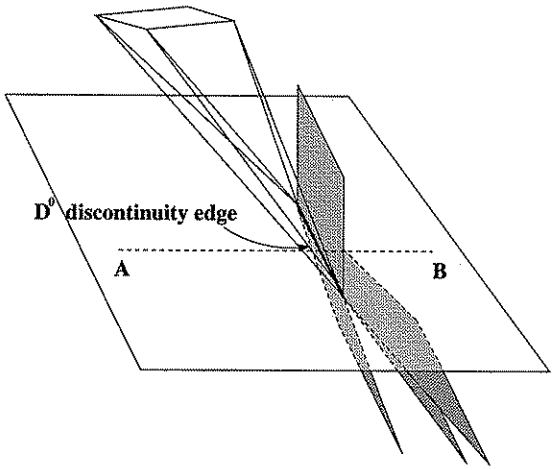


Figure 2.20: D^0 resulting from more than 2 overlapping D^2

a similar algorithm which traces every EV surface; EEE surfaces were ignored. At the same time a different 3-D algorithm was proposed by Lishinski *et al* [61]. They also considered only the EV discontinuities but used a more "progressive" approach for locating them. A separate computation is made for each emitting polygon. The highest energy polygon is selected to be the source at each pass, the discontinuities on the other polygons caused from this are found and the intensities are calculated. At the end, the resulting meshes are merged in order to produce the final subdivision. This approach was adopted in later DM research.

EEE surfaces were partly treated by Teller [100], in a related computation where the visible region of a source through a sequence of portals is calculated. This method finds what would be the extremal umbra boundary in our problem, but the algorithm is based on a 5-D Plucker coordinate representation that does not generalise so easily to a complete DM solution, and is computationally expensive.

Algorithms that include all EV and EEE events, even non-emitter ones, were later presented by Drettakis and Fiume [30] and Stewart and Ghali [96]. Most other researchers, including the author, have chosen to ignore EEE and non-emitter EV surfaces because the error produced by their exclusion is small compared to their cost.

Following Lishinski's progressive algorithm, discontinuity meshing can be broken down into four main operations:

1. The discontinuity curves are found by tracing the critical surfaces of the emitter through the environment.

2. These curves are used to construct the mesh, on each of the scene polygons/patches, due to the particular emitter.

3. The illumination intensities at the vertices of the mesh and other selected points is calculated. Calculating the intensities requires the determination of the visible part of the source from each point.

These three steps are repeated for each of the major emitters and then finally:

4. The meshes created on each surface are merged together to form the final subdivisions.

This last step is important only to the radiosity solution so it will not be described here. In most algorithms, the tasks of locating the discontinuities and constructing the mesh are interleaved but they will be discussed separately here.

Locating the Discontinuities

All DM-algorithms to date locate the D^0 edges and the D^1 and D^2 edges using different methods. The D^0, which lie at the contact between surfaces, are computed first by considering only object proximity. This involves visiting each object and comparing it against the adjacent ones. The efficiency of this operation depends on the efficiency of the method used for determining proximity. Tampieri [98] uses a hierarchy of bounding volumes while Drettakis [30] uses a voxel-based subdivision structure.

To find the rest of the discontinuities, which are EV emitter edges since we are not treating EEE or non-emitter EV, the method used again by all previous algorithms, is to form the semi-infinite *wedges* using a vertex of the source and an edge of an occluder or an edge of the source and a vertex of an occluder and find their intersection with the scene polygons. For the rest of this discussion we will differentiate between the wedges caused by a source vertex and those formed by a source edge by calling the former *ve* and the latter *ev* wedges. A *ve* wedge with its intersections in the environment can be seen in Figure 2.21.

All algorithms process each EV wedge separately to find its intersections with the environment, but differ mainly by whether they sort the scene polygons and compare it against them in that order or not. In the latter group we have Heckbert and Drettakis. Heckbert compares each scene polygon against each wedge,

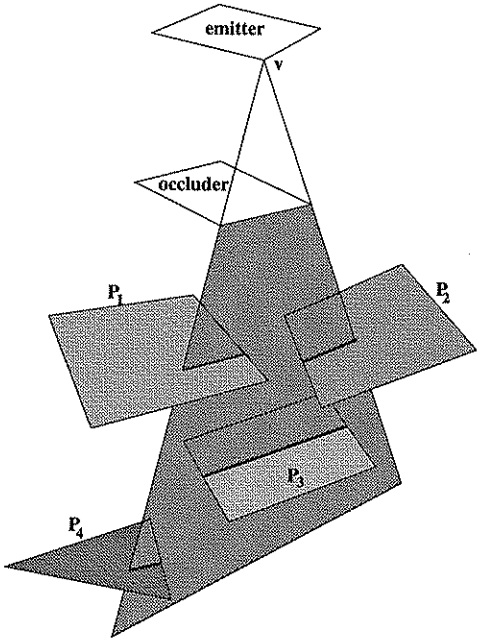


Figure 2.21: Intersecting the wedge with the scene polygons

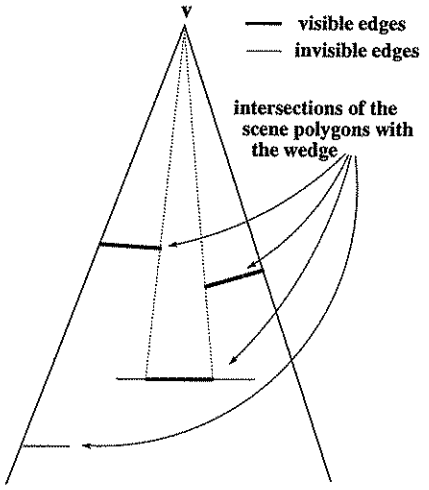


Figure 2.22: Intersections are transformed to the wedge space to determine visibility

requiring excessive computations. Drettakis greatly reduces the number of comparisons by using a voxel based subdivision which limits the candidate polygons to only those sharing voxels with a wedge.

One problem with processing the polygons in an unsorted manner is that it is not possible to tell immediately if the intersection of a polygon with a wedge is a discontinuity (critical edge). Only the intersection edges that are visible from the apex of the wedge form discontinuity edges and should be added to the mesh. For example the intersection of polygon 4 in Figure 2.21 cannot be seen by v, because it is blocked by polygons closer to v, and hence should not be added to the mesh. To find the critical edges from each wedge all the intersections are transformed into the wedge plane and put into a sorted list. A 2-D visibility test (a variant of the Weiler-Atherton visible surface algorithm [5, 111]) is performed from the point of view of the wedge apex and the critical edges are found. In Figure 2.22 only the thick edges correspond to discontinuities.

Alternatively the scene polygons can be built into a BSP tree which will provide the order from the apex of each wedge eliminating the need for the 2-D visibility test. This approach was taken by Lishinski [61] and Gatenby [37]. As the polygons are compared against the wedge in front-to-back order the intersections are found and at the same time the wedge is clipped against the intersected

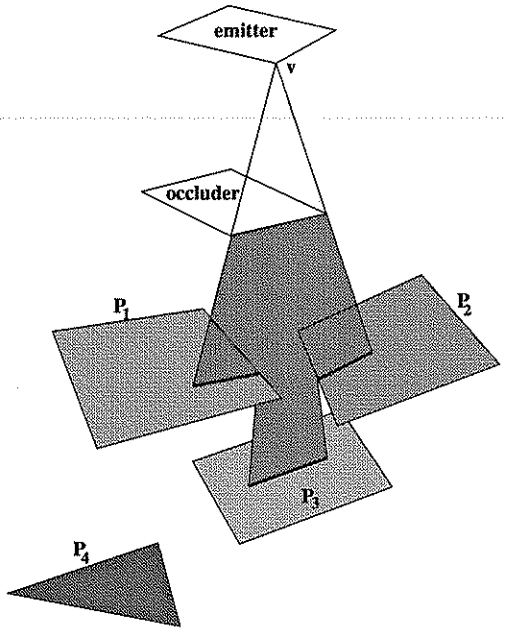


Figure 2.23: Clipping the wedge when compared against ordered polygons

polygons so that only the unobstructed parts of it are traced further. The tracing can stop as soon as the wedge is completely clipped avoiding unnecessary polygon/wedge comparisons. This can be seen in Figure 2.23: intersections that do not form discontinuity edges are not found and the tracing of the wedge stops at polygon 3.

In *Chapter 5* an alternative approach is given. The set of EV wedges corresponding to one occluder are treated as one entity (a shadow volume) and they are traced together in the scene. The shadow volume is compared against a candidate list of receiver polygons which is found using a tiling cube based method. The D^0 are found in the same pass.

Constructing the Mesh on the Faces

Once the discontinuity edges on each polygon are found they are combined to form the mesh on the polygons. A common data structure used for representing the mesh, which is also used in this thesis, is the *discontinuity meshing tree* (*DM-tree*). One such tree is used per scene polygon.

The DM-tree consists of two parts: a 2-D BSP tree and a Winged Edge Data Structure (WEDS)[8]. The WEDS is an edge based structure suitable for maintaining the consistency and accelerating access to the adjacency information. It has three basic elements: a vertex, an edge and a face structure. Most of the

topological information is held on the edge structure. This has pointers to the two faces lying on either side, to the two vertices on its endpoints and to four other edges sharing these vertices. Each vertex structure holds a 3-D point as well as a reference to an edge of which it is an endpoint and each face structure holds a pointer to one of the edges that define its boundary.

The WEDS is central to the DM-tree because it allows for the intensity of each vertex to be calculated only once and shared between the incident faces. Also it ensures that no T-vertices are introduced as the faces are split. The absence of such a structure was apparent in Campbell's work [15]. He was the first to use the BSP for the representation of shadows on the polygons but in his structure the edges were not shared between the neighboring elements which led to multiple illumination calculations and a problems with the T-vertices.

The 2-D BSP tree is an augmented version of the structure described in *Section 2.1.* As before each internal node holds a sub-hyperplane (edge) and it is defined by its hyperplane (line). Each node corresponds to a region of space which is partitioned by the node hyperplane with the leaf nodes corresponding to unpartitioned regions (cells or mesh faces). At the leaf nodes, however, in this case we store an explicit representation of the region of 2-D space that corresponds to the leaf by keeping a pointer to the mesh face.

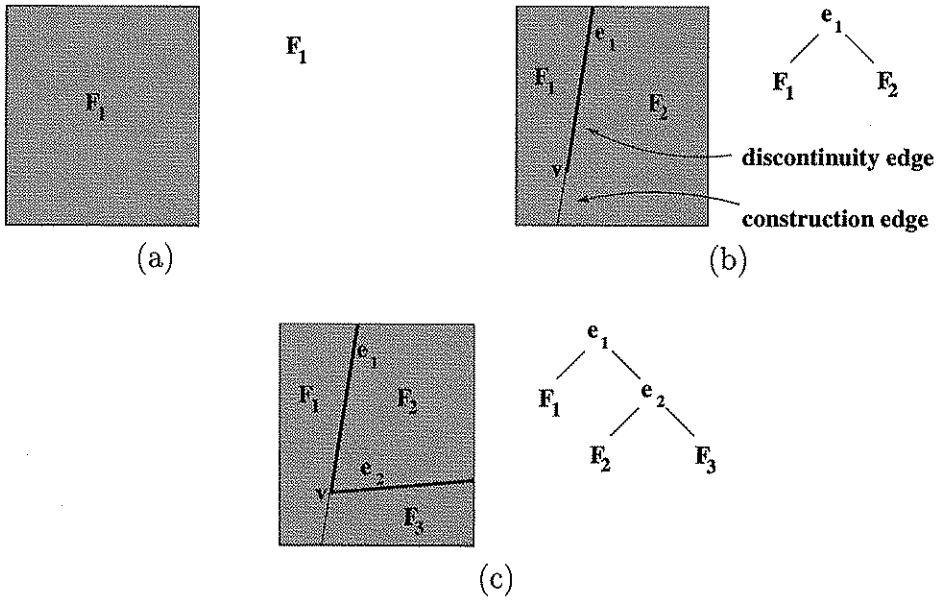


Figure 2.24: Building the DM-tree

Initially the DM-tree is a single leaf node holding one face (the whole polygon), as in Figure 2.24(a). When a discontinuity edge is added splitting the polygon, the tree is updated to store the edge at the root and the two new faces at its

leaves. If the edge does not span the face completely then it is augmented by adding another segment, called *construction* edge, to keep the subdivision convex (Figure 2.24(b)). As more discontinuity edges are added they are filtered down the DM-tree, possibly being subdivided on the way, until they reach the leaves where they subdivide the faces held there.

One of the potential problems of this method for building the mesh, is that it relies on machine precision to connect the discontinuity edges at the right point. Take for example edges e_1 and e_2 in Figure 2.24. These two edges were caused by wedges formed by consecutive edges of the occluder (or consecutive vertices) and that is why they share a common end-point (v). As each wedge is traced independently and the discontinuities are inserted into the structure one by one, they will only correctly connect at v if no precision errors occur in the calculations. This is a common problem, occuring in many applications and it is usually tackled by using a tolerance value that gives thickness to the lines. Of-course this creates other problems if we have very small or closely placed edges.

The only work that goes some way towards limiting this problem is that of Drettakis [29]. The *ev* wedges are created and traced in the order of the occluder's edges and each new discontinuity edge on the same receiver is connected to the end of the previous one. This is made possible because of the use of an extended WEDS to accommodate temporarily dangling edges without the use a 2-D BSP. Yet this can only correctly connect half the edges, and the *ve* events still rely on machine precision.

The algorithm presented in *Chapter 5* of this thesis correctly connects all EV discontinuities from an occluder before they are inserted into the the mesh of the receiver and thus minimises all such errors.

Illuminating the Mesh Vertices

Each vertex in the mesh created above (as well as any generated by further mesh refinement, like triangulation) must be illuminated. The illumination calculation of the mesh is usually the most costly part of DM. This is because for each vertex the visible parts of the source must be found before applying *Equation 2.1* on them. The usual method for finding the visible parts of a source (S) as seen from a vertex (v) is to project the occluders of v onto the source plane and then use them to clip away any hidden parts of the source. Of course only vertices in penumbra need to do this source/occluder comparison but most of the

DM-methods do not provide immediate information on which vertices are in the penumbra or which polygons are causing the penumbra if the vertex is found to be in one.

Using every polygon in the scene as a potential occluder would be extremely inefficient. Lishinski [61] limits the number of potential occluders by using a shaft culling technique [49]. For each vertex v in the mesh a pyramid is constructed using v as apex and the source as its base. The scene polygons are compared against this and only those that intersect the pyramid's interior are projected onto the source plane for clipping the source.

Gatenby [39] uses spatial coherence to provide a much smaller set of potential occluders for each vertex. It relies on the fact that if an occluder O does not obstruct any part of a given receiver R from the source then no vertex on R can be in the penumbra of O and hence O is excluded from the potential occluder set of the vertex. Before illuminating the vertices in the mesh of a receiver polygon R, a two step "pre-processing" is performed involving the whole of R, to find only the polygons that affect at least a part of it. First the BSP tree is traversed, front-to-back, from each of the source vertices until R is found. The sets of polygons encountered by the traversals are put together to form one set L'_o. The receiver is then compared against the penumbra shadow volume of each polygon in L'_o . When R is found to intersect the penumbra of a polygon in L'_o, this polygon is added to another list L_o. The potential occluders of vertices in the mesh of R is the set L_o. Those receivers totally in umbra or unoccluded are therefore treated particularly fast.

An altogether different way of calculating the visible parts of the light source, based on the aspect graphs, was suggested by Drettakis [30] and Stewart [96]. They use a data structure called the *backprojection* which stores the exact structure of the visible part of the source. This is computed once for each point on a surface and then it can be updated incrementally each time a discontinuity edge is crossed to get to a neighboring cell. It is fast compared to the alternatives but it has the disadvantage of requiring a full mesh to be constructed, one that includes non-emitter EV and EEE edges.

In the algorithm described in *Chapter 5* of this thesis the occluder polygons are identified and stored with each mesh element during the construction of the mesh so no searching is required at illumination time.

2.3.5 Area Light Sources in Dynamic Scenes

No one has addressed the problem of computing analytically, in object space, the shadows of moving objects. Some work involving dynamic scenes has been done in the context of radiosity [7, 16, 40, 65] but only in the form of altering the radiosities on the effected patches. The geometry of the mesh does not change to explicitly follow any shadow information.

2.4 Discussion

As it was described above, the shadow problem can be thought of as a visible surface determination (VSD) one. One of the main parts of VSD algorithms is to decide the precedence of overlapping faces, as seen from the viewpoint. A straightforward way to do this is by using a BSP tree. The BSP tree is seen as a static structure and also as one that can give an ordering only from a point (or region with same aspect). In *Chapter 3* it is shown that under the right circumstances the BSP tree can be modified at interactive speeds. A method is also described for ordering the scene polygons with respect to a restricted planar surface which can be used to provide the ordering from area light sources required in *Chapter 5*.

The SVBSP tree [17] and the shadow tiling [92] are the two fastest object space shadow algorithms. The resulting shadows can be easily used for successive frames where the viewpoint changes but the geometry remains the same. Ways of extending/modifying these methods to deal with moving objects are presented in *Chapter 4*.

The algorithms described for area light sources, especially the ones in the discontinuity meshing research, capture all major variations in the illumination function. When used with higher order interpolations they can produce very realistic effects. But the motivation for these algorithms was to produce an initial meshes for radiosity solutions which are concerned mainly with static scenes, they are not easily incorporated into dynamic scenes. In *Chapter 5* a method is presented that can still find the main variations in the illumination function but which also allows for modifications in the scene data.

Chapter 3

BSP Trees for Dynamic Scenes and Ordering

As discussed in the previous chapter, we will be using BSP trees in the shadow algorithms but before we can do that we need to look at certain issues regarding their use. In this thesis they have been used for ordering the scene polygons and for calculating and storing shadow information. Owing to the nature of our applications, i.e. dynamic scenes illuminated by point and area light sources, we came across two main problems which we will try to address in this chapter.

The first is how to update the BSP tree to reflect changes in the scene data. This is required by each of the algorithms to be described in the later chapters, as they all operate in non-static environments. For the point source tiling method (*Section 4.1*), the tree is built from the scene polygons to provide an ordering. For the SVBSP method the tree holds the polygons and the shadow planes for calculating the shadows. In the area source algorithm (*Chapter 5*), a 2-D tree is used for storing the shadow information on each polygon (DM-tree).

The second problem involves obtaining an ordering of the polygons from a planar polygonal area, rather than from a point. This is necessary for simplifying and speeding up the area source shadow algorithm.

3.1 Using the BSP Tree in Dynamic Scenes

Several methods have been proposed in earlier literature for using BSP trees in dynamic scenes (see *Section 2.1.6*). None of them, however, provides a simple and general solution. The merging algorithm is a notable solution but as described in [73, 69] it requires prior knowledge of the moving objects. Also the trees of the

static and moving objects need to be optimised [72] for the algorithm to operate at a reasonable speed.

In this section we describe an algorithm that places no restrictions on the moving objects or their paths. In its simplest form it can be added to an existing BSP implementation with minimal changes [20].

Without loss of generality we will assume that any change in the scene data can be modelled by two operations: deletion and/or addition of objects. An object transformation consists of the following steps: delete object, multiply its vertices by the transformation matrix, add object. As described in *Chapter 2*, a BSP tree can be built incrementally, either by adding polygons individually (*Section 2.1.2*) or by merging the *single* BSP tree of an object onto the total tree of the scene (*Section 2.1.5*). We can use this incremental building for the addition of objects to the tree. There remains only one issue for a complete algorithm: how to delete an object from the tree. When referring to a 3-D space, we will consider an object to be a set of oriented polygons. Thus removing an object is equivalent to removing its polygons.

Deleting a polygon from the tree seems a difficult operation, since the plane defined by the polygon may have been used to form a node which further subdivide the set of polygons in a subspace. In this case deleting the polygon and its node would split the tree into two separate pieces.

However this is not always true. A more careful study of the problem shows that in many cases the polygon can be deleted trivially. Any particular polygon can hold one of four positions in the tree. To delete the polygon, only one of these positions needs to be processed, and our algorithm will capitalise on this.

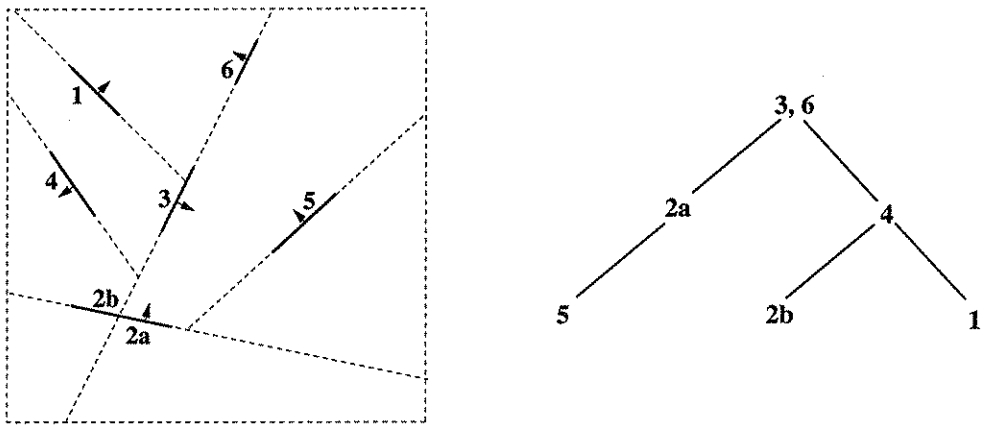


Figure 3.1: Possible positions of a polygon in the tree

The four positions a polygon can hold in the tree are:

(i) On a leaf node, polygon 5 in Figure 3.1.
The plane of the polygon is used to split the subspace into two empty subspaces. In this case we can simply replace it with an empty cell.

(ii) On a node with other faces, polygon 6 in Figure 3.1.
The polygon can be deleted from this node, since the plane is still defined by other polygons on that node. However, if the first two polygons in this node face in opposite directions, and we are deleting the first polygon, then the front and back subtrees of the node must be swapped in order to maintain the correct front and back ordering of the tree.

(iii) On a node with exactly one non-empty child, polygon 2a in Figure 3.1.
In this case, the plane was used to split the subspace into an empty region and a non-empty region. Thus if this node is removed, it can be replaced by the node representing the non-empty region. In other words, the child of the deleted node directly becomes the child of the parent of that node.

(iv) On a node with two non-empty children, polygon 4 in Figure 3.1.
This is the only case that needs more processing, as the polygon is used to split the subspace into two non-empty regions. Therefore if the polygon is removed we will be left with two unconnected subtrees.

In the following three subsections, ways of removing the nodes with two non-empty children (case (iv)) are examined. *Section 3.1.1* gives a simple general solution that requires no extra programming but makes no use of the information already built into one of the subtrees. *Section 3.1.2* uses merging of the two subtrees but it is limited to a 2-D domain since it assumes that the subspaces defined at the cells and internal nodes of the tree are explicitly stored into a Winged Edge Data Structure (WEDS). An example using this method can be found in *Chapter 5.* The last method is a generalisation of the ideas used in *Section 3.1.2* to work in 3-D without the WEDS.

3.1.1 Inserting Individual Polygons

The easier and in some cases the fastest method for joining two subtrees together after the root has been removed is simply to insert the faces of the smaller one into the larger.

The algorithm for removing an object from the tree (*removeFromTree* in Figure 3.2) has the following form. Each node of the BSP tree that holds a polygon of

```
Tree tranformObjects(Tree t, Object obj[])
/* t is the scene BSP and obj[] are the */
/* objects to be transformed */
{
        /* remove the object polygons from t */
        t = removeFromTree(t, obj[]);
        /* transform the object geometry */
        getNewObjectGeometry(obj[]);
        /* add the objects back to the tree */
        return insertObjectsInTree(t, obj[]);
}


Tree removeFromTree(Tree t, Object obj[])
{
        /* mark the nodes holding a polygon */
        /* of the transformed objects */
        for each object o_i in obj[]do
                markNodes(t, o_i);
        endfor
        /* remove marked nodes from tree */
        return restore(t);
}
```

Figure 3.2: Transforming a set of objects in the BSP tree

```
Tree restore(Tree t)
{
        if (empty(t)) return EMPTY;
        endif

        /* if the root of the tree is not marked */
        if (notMarked(t.root))
                /* recurse into the front and back */
                t.front = restore(t.front);
                t.back = restore(t.back);
                return t;
        else /* the root of the tree is marked */
                if (any subtree empty)
                        return restore(other subtree);
                else
                        /* find the smallest subtree */
                        ts = smaller subtree of t;
                        /* remove the marked nodes */
                        /* from the largest subtree */
                        tl = restore(larger subtree of t);
                        /* add the polygons of the small */
                        /* into the large, one by one */
                        return filterPolygonsOf(ts, tl);
                endif
        endif
}
```

Figure 3.3: Removing the marked nodes from the BSP tree

the transformed object is marked. This is done by following the location pointers that each polygon stores when it is inserted into the tree. If a node holds more than one polygon then instead of marking the node we delete the polygon in question from it directly (case (ii) above). The remaining coplanar polygons will still define the node but we must swap the subtrees if necessary. After all the relevant nodes have been marked a recursive function is called that goes through each node once and removes the marked ones, as shown in function *restore* in Figure 3.3.

In brief, at each iteration the root is checked and if it is not marked the function proceeds to the left and right subtrees. If the root is marked and it has only one non-empty subtree, the algorithm will return that, after it has been processed. If both subtrees are non-empty then it will perform three steps: find the largest of the two; call *restore* with it as an argument; and return the tree generated by inserting the polygons of the smaller into it. The polygons are inserted individually using the algorithm for incrementally building the tree (see *Section 2.1.2*). To determine which tree is larger, we go through both trees and count the polygons of the largest connected subtree in each (a marked node splits a tree into unconnected subtrees).

With *restore*, we have a way of removing the polygons from the tree which

can then be used by the *transformObjects* function (see Figure 3.2) to transform one or more objects.

3.1.2 2-D Using WEDS and Merging

In the method described above, no relation between the polygons in the subtrees is assumed. There are applications, however, where such a relation can be assumed and exploited. The 2-D BSP tree used to represent the discontinuity meshing in *Chapter 5* is one such case. The discontinuity edges created on a receiver from an occluder polygon are grouped together and bounded by the, relatively few, penumbra edges. Although on certain receivers a very large number of discontinuity edges may be present their *clustering* forms a very *efficient subdivision* of 2-D space [72]. Another important factor is the explicit representation of the binary partitioners in the tree. Each node holds an augmented discontinuity edge that spans the whole of the subspace represented at the node.

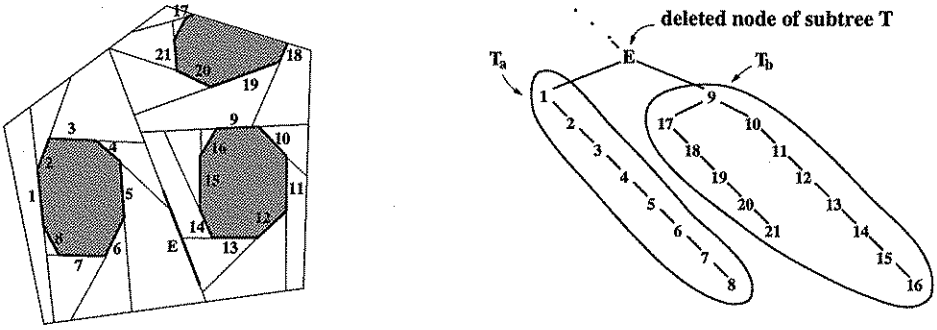


Figure 3.4: A 2-D scene and its tree representation

In such an application merging becomes very attractive, since its two main requirements (*efficient subdivision* and explicit representation of the sub-hyperplanes) are ready met. Thus we can use merging to put together the two subtrees of a deleted node.

First the root edge is removed from the WEDS. Then one of the subtrees (T_a) is chosen to form the basis of the new tree. This is expanded to span the whole subspace defined by both T_a and T_b and then (T_b) is merged into T_a. Unlike previous uses of merging that put two unrelated trees together, here we have two subtrees from the same tree. This means that they are defined in mutually exclusive subspaces. We also know the common boundary region of their subspaces, i.e. the edge defined at the marked node, which we shall call E. When this boundary edge is removed and one of the subtrees forms the basis

for the merging (T_a), this subtree may have some of its edges expanded to the whole extent of the space defined at the marked node. The edges that will be expanded are those that are bounded by E. These may now split the other subtree (T_b). Any edge of T_a that did not touch E, had E totally on one side (or it is bounded by edges in between, in which case the edge is irrelevant to our discussion). Consequently it must have the subspace over E on one side as well, since the BSP always subdivides space into convex cells.

Following the above reasoning when T_b is inserted into the expanded T_a it can use point classification for all nodes that did not get expanded and only apply the more expensive tree partition for the expanded nodes.

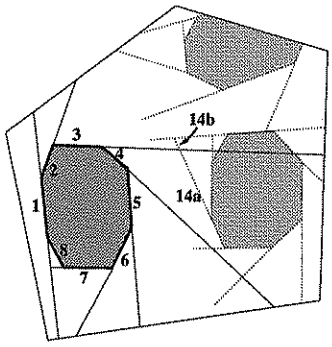


Figure 3.5: T_L is expanded to the whole subspace

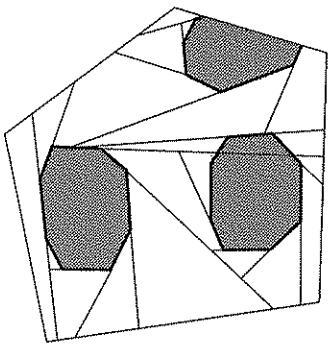


Figure 3.6: T_b is inserted into T_a to form one tree

The pseudocode for the algorithm is given in Figures 3.7 to 3.9. We will show how it works using the example in Figure 3.4. To delete the marked node of edge E that has two non-empty subtrees (T_a and T_b) the following three steps are performed:

1. The edge at the marked node is removed from the WEDS. For this we need to traverse E from end to end. The edges at its endpoints are joined together (these are boundary edges of the cell defined at the parent node) while anything else touching the edge is marked as *dangling*, and may need to be expanded later. The number of *dangling edges* on each side of E is counted. In our example the dangling edges would be {3, 4} on the left and {9, 12, 13, 19} on the right.

2. One of the subtrees is chosen to form the basis for the merging. Unlike the earlier method (*Section 3.1.1*) where the choice was based purely on size (largest connected subtree) here we can use a more elaborate function for predicting the cost of inserting each subtree. Most of the computation

involved in merging goes into the partitioning of the inserted tree. The fewer the dangling nodes, in the kept tree, the fewer partitioning operations will be performed. The cost of each individual partitioning depends mainly on the number of nodes of the partitioned tree visited. An accurate cost estimation can account for this number by comparing the lines of the dangling edges against the bounding areas of the subtrees of the inserted tree, or by some other method. We used a simpler but still adequate estimation of the cost:

$$E[cost_a] = D_a * size_b$$

where $a, b \in$ {front-subtree, back-subtree} and $a \neq b$, D_a = number of dangling edges in a and $size_b$ = number of unmarked nodes in b, (for the DM-tree we can use as size the number of unmarked penumbra edges).

Using this function T_a is selected, traversed from top to bottom and the dangling edges are extended to span the whole of the cell defined at the removed node. This creates a convex partitioning of the cell (Figure 3.5). To extend the dangling edges, the boundary of the cell defined at the parent node of E is traversed by always following edges from nodes that are ancestors of E. This is important since the edges of T_b are still in the WEDS but at the moment they should be ignored (Figure 3.5).

3. Finally, T_b is inserted into T_a to form a unified tree, as in Figure 3.6. The important thing here is that only the nodes that were expanded in step 2 ({3, 4}) may possibly split T_b, as they are the only ones that intersect T_b's subspace. For the rest of the nodes in T_a that will be encountered ({1, 2, 5}), classifying one point on T_b will suffice. When the merging is finished the dangling edges of T_b must be extended to span the whole of their subspace. Meanwhile, at partitioning, the subtrees created are condensed to avoid unnecessary fragmentation of homogeneous regions. An example of where the condensation can take place is edge 14 in Figure 3.5. The top part, 14b, does not contain any part of a discontinuity and so it will be removed.

This method is particularly fast if one of the subtrees has only a few dangling edges. An example of an extreme case can be seen in Figure 3.10. Here our algorithm can detect the left side of E having no intersection (no dangling edges recorded when deleting E). Hence T_b is inserted in T_a as a point.

In the particular application given in *Chapter 5*, apart from having our information optimised for merging there is another good reason for using merging

```
Tree restoreWeds(Tree t, Object obj)
/* remove all nodes in tree t belonging to object obj */
{
        if (leafNode(t)) return t;
        endif

        /* if root of tree was caused by the transformed object (obj) */
        if (fromObject(t.root, obj))
                /* remove root edge from WEDS and cound dangling edges */
                counter = removeFromWeds(t.root);
                if (one subtree empty)
                        return restoreWeds(other sub-tree, obj);
                else
                        /* use the number of dangling edges and */
                        /* size of subtrees to choose which to keep */
                        if (keepBackTree(counter, t))
                                ts = t.front;
                                tl = restoreWeds(t.back, obj);
                        else
                                ts = t.back;
                                tl = restoreWeds(t.front, obj);
                        endif
                        return insert2DTree(tl, ts);
                endif
        else /* root is not from obj */
                /* if any of the root end point are dangling */
                if (dangling(t.root))
                        /* expand root edge to span the whole space */
                        /* of the cell defined at the parent node */
                        expand(t.root, t.parent);
                endif
                t.front = restoreWeds(t.front, obj);
                tback = restoreWeds(t.back, obj);
                return t;
                endif
        endif
}
```

Figure 3.7: Removing the marked nodes from the 2-D BSP tree and the WEDS

instead of the method described in *Section 3.1.1.* Each vertex in our structure stores an illumination value which is quite expensive to calculate, and therefore we want to keep as many of the existing vertices as possible. If we insert each edge separately we may disturb them.

3.1.3 3-D Using Merging

There may be 3-D environments where merging is an appropriate way to put together the subtrees of a deleted node. As before the two subtrees are defined in mutually exclusive subspaces and the only nodes that may intersect the opposite subtree are those that intersect, with their plane at-least, the boundary between the two subspaces. In general we will not have an explicit representation like the DM-tree above to know which and how many they are.

We can still obtain this information by using the following simple procedure.

```
Tree insert2DTree(Tree t1, Tree t2)
/* inserts t2 into t1, t1 has already been restored but not t2 */
{
        if (leafNode(t1)) return restoreWeds(t2);
        endif
        if (leafNode(t2)) return t1;
        endif

        if (t1.root == t2.root) /*segments of the same line*/
                t1.front = insert2DTree(t1.front, t2.front);
                t1.back = insert2DTree(t1.back, t2.back);
        else if (t1.root has been expanded)
                partition(t1.root, t2, t2F, t2B);
                t1.front = insert2DTree(t1.front, t2F);
                t1.back = insert2DTree(t1.back, t2B);
        else
                c = classifyPoint(t1.root, any point of t2);
                if (c == FRONT)
                        t1.front = insert2DTree(t1.front, t2);
                else
                        t1.back = insert2DTree(t1.back, t2);
                endif
        endif
        return t1;
}
```

Figure 3.8: Inserting one of the subtrees into the other

First we construct a polygonal representation of the intersection of the root plane with the region of space represented by the root, the sub-hyperplane (shp) (see *Section 2.1.4*) and then we insert it in T_a and T_b while performing two actions into the process:

1. counting the nodes that intersect it in each tree, and hence might partition the other subtree when inserted, and

2. storing at each node encountered the classification of the shp (or the fragment that reached there) against the plane of the node.

For this operation to work properly the two subtrees must have any marked nodes removed first, so that the *restore* function of Figure 3.2 is re-arranged to recurse into the subtrees before processing the root (see Figure 3.11). After this measuring/marking step we can proceed with the merging of the subtrees. Now we use the cost estimation function as in the previous subsection, for deciding which subtree to use as base, depending on the number of times the sub-hyperplane was intersected by the two subtrees and depending on the size of the subtrees.

The subtree to be inserted is filtered using the classifications stored at nodes of the other tree. It can only visit the nodes that the shp has visited (possibly less but not any others) and it will have the same classification as the shp. Thus

```
int removeFromWeds(Edge e)
/* removes the edge; marks as dangling any edges lower */
/* down thet ree; returns a sum indicating which sides */
/* has more dangling edges */
{
        v = get left end of e;
        removeFromWeds e from vertex v;
        while (v != NULL) do
                v = next vertex to the right, along e;
                if (edges meeting at v are on higher nodes on tree)
                        connect the edges at v together;
                        v = NULL;
                else
                        mark edges meeting at v as dangling;
                        counter = add how many to left and right;
                endif
        endwhile
}
```

Figure 3.9: Removing a marked edge from the WEDS

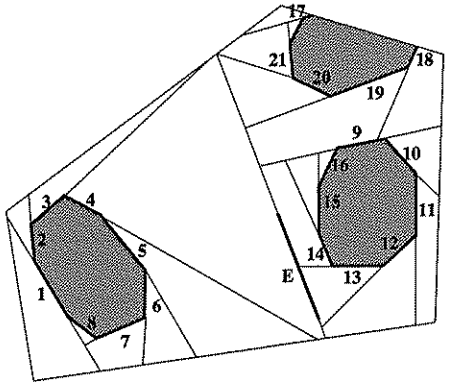


Figure 3.10: When E is removed, T_b can be inserted in T_a as a point because none of edges of T_a touch E

we will use the nodes that store a CUT value to partition the inserted subtree while traversing the others without any processing at all (Figure 3.12).

One benefit of this method compared to the original merging is that it does not need to store the polygon representation of the shp at each node. Storing it requires a lot of memory and processing every time a polygon is added to the tree. Here we only need to calculate shp that are known to cut the other subtree.

This algorithm has not been evaluated (as it was not required by any of the shadow algorithms discussed in this thesis) and requires further experimental exploration.

```
Tree restore3D(Tree t)
{
      if (empty(t)) return EMPTY;
      endif

      t.front = restore3D(t.front);
      t.back = restore3D(t.back);

      if (notMarked(t.root))
            return t;
      else
            if (any subtree of its empty)
                  return restore3D(other subtree);
            else
                  shp = findShp(t.root);
                  {ts, tl} = measureClassify(t.front,
                              t.back, shp);
                  return merge3D(ts, tl);
            endif
      endif
}
```

Figure 3.11: Removing the marked nodes from the tree

```
Tree merge3D(Tree t1, Tree t2)
/* insert t1 into t2 */ {
      if (empty(t1)) return t2;
      endif
      if (empty(t2)) return t1;
      endif

      /* the classification of the shp against */
      /* was stored and is used here */
      c = root.classification;
      if (c == FRONT)
            t1.front = merge3D(t1.front, t2);
      else if (c == BACK)
            t1.back = merge3D(t1.back, t2);
      else /* CUT, it cannot be ON */
            partition(t1.root, t2, t2F, t2B);
            t1.front = merge3D(t1.front, t2F);
            t1.back = merge3D(t1.back, t2B);
      endif
      return t1;
}
```

Figure 3.12: Merging the two subtrees

3.1.4 Discussion

In this section we will discuss some details about the operation and implementation of the algorithm.

First, it is important to realise that when a target object is being transformed in an interactive application, the *restore*, *restoreWeds* and *restore3D* functions are only relevant for the very first transformation. The reason for this is that after the object polygons have been deleted from the tree, when they are transformed and then reinserted, they will end up in one of three places: at the leaf-nodes of the tree; at a node shared with other coplanar polygons; and at a node near the leaf-nodes of the tree, on subtrees consisting wholly of polygons belonging to the target object. In each case only the simpler deletion cases discussed in *Section 3.1* will need to be used in subsequent transformations of this object in this particular interactive sequence.

Second, a consequence of the algorithm is that objects whose polygons are near the leaf-nodes of the BSP tree can be deleted in constant time. Therefore objects which are likely to be transformed frequently, for example a 3D cursor object in an interactive application, or smaller objects in the interior of a room in an application involving, say, room layout, should be placed into the tree last.

In the *restore* function as presented in Figure 3.3, in the case where the node to be deleted has two children, the smaller subtree is filtered into the larger one, regardless of the cost of the operation. An alternative strategy is to adopt some

criteria which determines when the filtering operation should be carried out, or when the nodes are simply marked as deleted but not actually deleted from the tree. The criteria we have adopted is to only do the filtering when the smaller subtree is less than some maximum size. It should be noted that in a scene there may be a very small proportion of polygons responsible for most of the splitting during the creation of the tree. It is precisely the nodes of these polygons which will, of course, have large subtrees. Therefore the operation of leaving marked nodes in the tree will not occur very often, and will not unduly increase the size of the tree. Furthermore, if these deleted polygons are left in the tree, subsequent transformations of other objects may diminish the size of the subtrees of the deleted polygons, and so they would eventually be deleted anyway.

An important requirement for keeping the number of rendering polygons under control will be to join fragments of a polygon together again, when they meet at a node while filtering the smaller subtree into the larger one. This is done by using a 2-D BSP tree and a WEDS to represent the subdivision on each polygon. This works in a similar way as for the DM-tree, *Section 3.1.2*. When two fragments of a polygon meet, their common edge is found and removed to form a single polygon.

3.1.5 Results

The programs for computing dynamic changes to BSP trees, were written in C and implemented on a SUN SparcStation 2+. The graphical output system used was X11 Release 5. Scenes were constructed in order to examine performance of the algorithms. Different scenes were used for the methods of *Section 3.1.1* and *Section 3.1.2* as they have different requirements. For the former a basic scene consisted of a room containing a desk with a computer, and a bookcase and books with further scenes constructed from this by adding additional desks and bookcases. For the latter the shadows on the floor from a collection of randomly placed cubes were used.

It is very difficult to quantify the performance of the algorithm for dynamic changes to BSP trees. This is because the time taken for an object transformation (not including rendering) depends heavily on the "alignment topology" of the polygon edges in the object rather than on the number of polygon edges in the object, or the particular geometry involved. By "alignment topology" we mean how many splits the target object polygons might cause in other polygons in the scene. If this number is very small, then the operation will be relatively fast.

We consider the case of a sequence of transformations applied to an object. This involves a simulation of the interactive situation where an object is selected and transformed (e.g. translated or rotated). The time for one complete transformation consists of three main components:

(a) marking the nodes in the BSP tree containing object polygons,

(b) using the *restore* or *restoreWeds* function to restore the tree, and then

(c) filtering the transformed polygons back into the tree.

However, as noted in *Section 3.1.4*, the full version of *restore/restoreWeds*, which can involve removing a node with two nonempty subtrees, is only needed for the deletion part of the first transformation in the sequence, since after this the object polygons will be at or near the leaf-nodes. Hence all the transformations involve (a) and (c) but only the first involves (b). In the implementation (a) also involves a "cleaning" process when used after the first transformation, that is the marked leaf-nodes are directly deleted in constant time. (a) is a constant and negligible time operation per polygon and is not discussed further. (c) re-uses the same function as was used for building the tree. The time taken for this step depends on the size of the object (number of polygons), as well as on the "alignment topology".

In our experiment, on average the percentage of time taken to insert an object against the total time to rebuild it was almost equal to the percentage of the number of polygons in the object against the total number of polygons in the scene. We will not give the timings for inserting an object back into the tree because we do not consider it to be very relevant. To justify our decision we use the following reasoning. Consider the two alternatives, rebuilding the tree or using our delete/add method:

rebuild:	rebuild without object;		**add object**
delete/add:	mark nodes;	restore tree;	**add object**

Rebuilding the tree can be seen as first building a tree with all other scene polygons and then adding the object polygons, while in our method we remove the object polygons from the tree and then add them back. As one can see adding the object is an operation that will take place regardless of the method used. What we will argue with the tables and discussion in this section is that marking the

scene	scene polygons initial	scene polygons after BSP	Build BSP time (ms)
office 1	133	164	86
office 2	211	258	150
office 3	333	537	318
office 4	745	1816	1240
office 4*	745	3521	2488

Table 3.1: Timings for initial building of the BSP trees

nodes plus restoring the tree take a fraction of the time for rebuilding the tree without the object.

The reader is reminded that the following results are only relevant for the first transformation of a selected object while for the following transformations the object can be deleted in constant time.

Evaluation of restore

The implementation was used with a test scene consisting of a room, a bookcase with two books, and a desk with a computer on top. A number of test scenes were created from this basic one: scene office1 consists of a room with a bookcase and two books, a desk and a computer. Scene office2 is a room with two bookcases and two books, two desks and a computer. Scene office3 is a room with three bookcases, two books, three desks and one computer. Scene office4 is a room with three bookcases, two books, ten desks and ten computers. Scene office4* has the same objects as office4 but with each one of them randomly rotated by a small angle. Images of these scenes are given in *Appendix C.*

The statistics for building the BSP trees of these scenes are given in Table 3.1, along with the number of polygons in each scene, initially and after the subdivision of the BSP tree. In the tables of this chapter, the times are given in milliseconds.

In Table 3.2 we give the results for the *restore* function (including tree + marking) of different objects for the first transformation in a sequence. For each object we show the number of polygons initially defined for the object, and the number after polygon-splits caused in the BSP construction. Under *restore tree* we give times for marking and restoring the tree based on actually deleting all marked nodes (*remove*), and the times (*partial*) if a node is deleted only when the

scene	object moved	object polys initial	final	restore tree (ms) all	partial		rebuild without	% rebuild all	partial
office1	computer	20	27	2	2		67	2.9	2.9
	desk & comp	56	87	1	1		43	2.3	2.3
	bookcase	54	66	9	1	(1)	38	23.7	2.6
office2	computer	20	27	2	2		147	1.4	1.4
	desk 2	36	52	2	2		112	1.8	1.8
	desk 1 & comp	56	87	3	3		109	2.7	2.7
	bookcase 1	54	54	13	2	(1)	130	10.0	1.5
	bookcase 2	54	54	3	3		115	2.6	2.6
office3	computer 1	20	27	24	4	(1)	309	7.7	1.3
	desk 1 & comp	56	87	38	7	(3)	300	12.7	2.3
	bookcase 1	54	66	28	6	(1)	480	5.8	1.5
	bookcase 2	54	130	15	7	(3)	244	6.1	2.9
	any other obj	6-56		2 - 3	2 - 3			<2	<2
office4	computer 1	20	27	70	20	(6)	1150	6.1	1.7
	desk 1 & comp	56	79	198	30	(13)	1270	15.6	2.3
	desk 2 & comp	56	115	22	12	(1)	1070	2.0	1.1
	bookcase 1	54	66	219	16	(1)	1131	19.3	1.4
	bookcase 3	54	221	118	23	(13)	899	13.1	2.5
	book	6	66	6	6		1220	0.5	0.5
	any other obj	6-56		8 - 20	8 - 11	(≤1)		<3	<2
office 4*	computer 1	20	27	495	38	(10)	2328	20.2	1.6
	desk 1 & comp	56	79	1620	61	(24)	1800	90.5	3.3
	desk 2 & comp	56	245	97	49	(9)	2179	4.4	2.2
	bookcase 1	54	72	588	54	(4)	2684	21.9	2.0
	bookcase 3	54	234	334	45	(16)	2206	15.1	2.0
	book	6	6	14	14		2460	0.5	0.5
	any other obj	6-56		18 - 40	18 - 32	(≤2)		<3	<2

Table 3.2: Timings for initial transformations of objects in the BSP tree

smaller subtree is not too large (the criteria used was a maximum of 5 polygons in the smaller subtree). Together with *partial* we give in brackets the number of marked nodes actually retained. This is important because if there were a large number of retained nodes it would significantly increase the size of the tree, but in fact the figures show that only a very small number of polygons (never greater than 1% of the total) cause much of the splitting.

The timing given under *rebuild without* is the time taken to build the tree without the selected object. More important than the timing is the percentage given in the two last columns, under *%rebuild.* This shows the saving involved in this algorithm by comparing the time to restore against the time to rebuild $(\frac{restore}{rebuild} \cdot 100)$.

To give a thorough evaluation of the algorithm, we computed the timings for

scene	cube faces		number of	Build BSP
	total	active	cells in BSP	time (ms)
4 cubes	26	11	595	140
7 cubes	44	21	1231	350
15 cubes	92	44	2541	800

Table 3.3: Timings for initial building of DM-trees

transforming each and every object in the scene. The selected objects in Table 3.2 are the ones that occupy the higher nodes in the tree and hence give the worst performance. For scenes office3 and office4, this can be confirmed by looking at the position of the selected objects on the trees which are shown in Figures 3.13 and 3.14 respectively. All other objects take considerably less time, less than 3% of rebuilding, which is mainly the time taken to traverse the tree and to do the size measurements when a marked internal node is found. While doing the experiment, we noticed that for certain objects the results were much better than we expected. Take for example desk 2 in office4 or bookcase 1 in office3, even though they have nodes with very large subtrees (see Figures 3.13 and 3.14), they were deleted relatively fast. The reason for this is that some of the object polygons are coplanar with polygons from other objects in the scene that are not moving. Even though this is normal and expected in most scenes, we ran another set of experiments (office4*) with a scene that has no coplanar faces between objects, to see the power of the algorithm in such a case. Here we found that the number of retained nodes, when retaining those with large subtrees, increased, even though it never exceeded 1% of the total number of nodes. Also the time for the worst case increased, with one object taking for its deletion as much as 90% of the rebuilding time. But in-fact the majority of the objects were deleted in less than 2-3% of the time with some as low as 0.5%.

In general we can see, that restoring the tree using this method takes a small fraction of the alternative.

Evaluation of restoreWeds

The *restoreWeds* function as described in *Section 3.1.2* only works for 2-D domains and hence the scenes used above are not suitable. Rather the data we used for evaluation are the discontinuity edges on a polygon created by a set of randomly placed cubes and a rectangular light source.

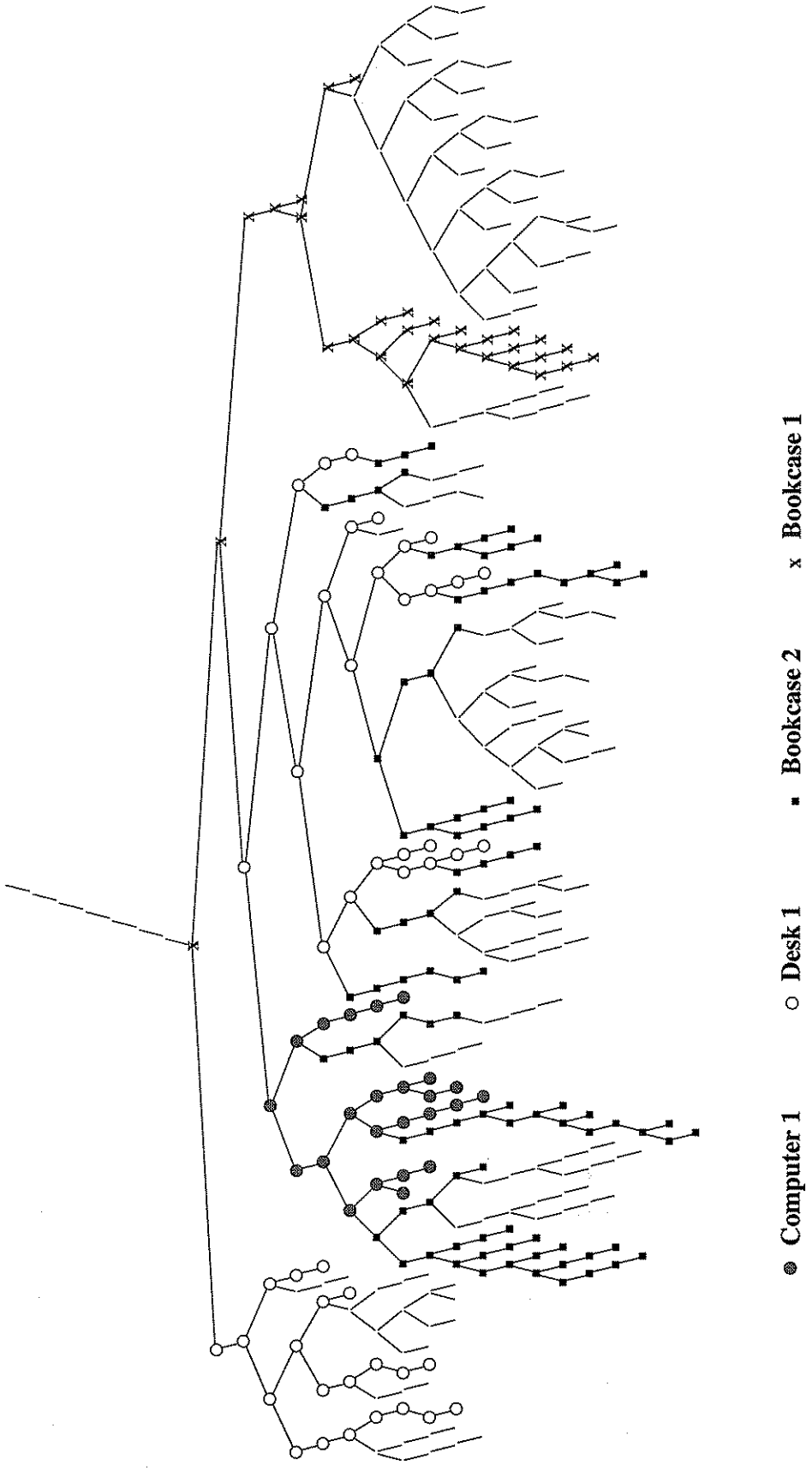


Figure 3.13: Tree for office3 with the objects shown in Table 3.2 marked out

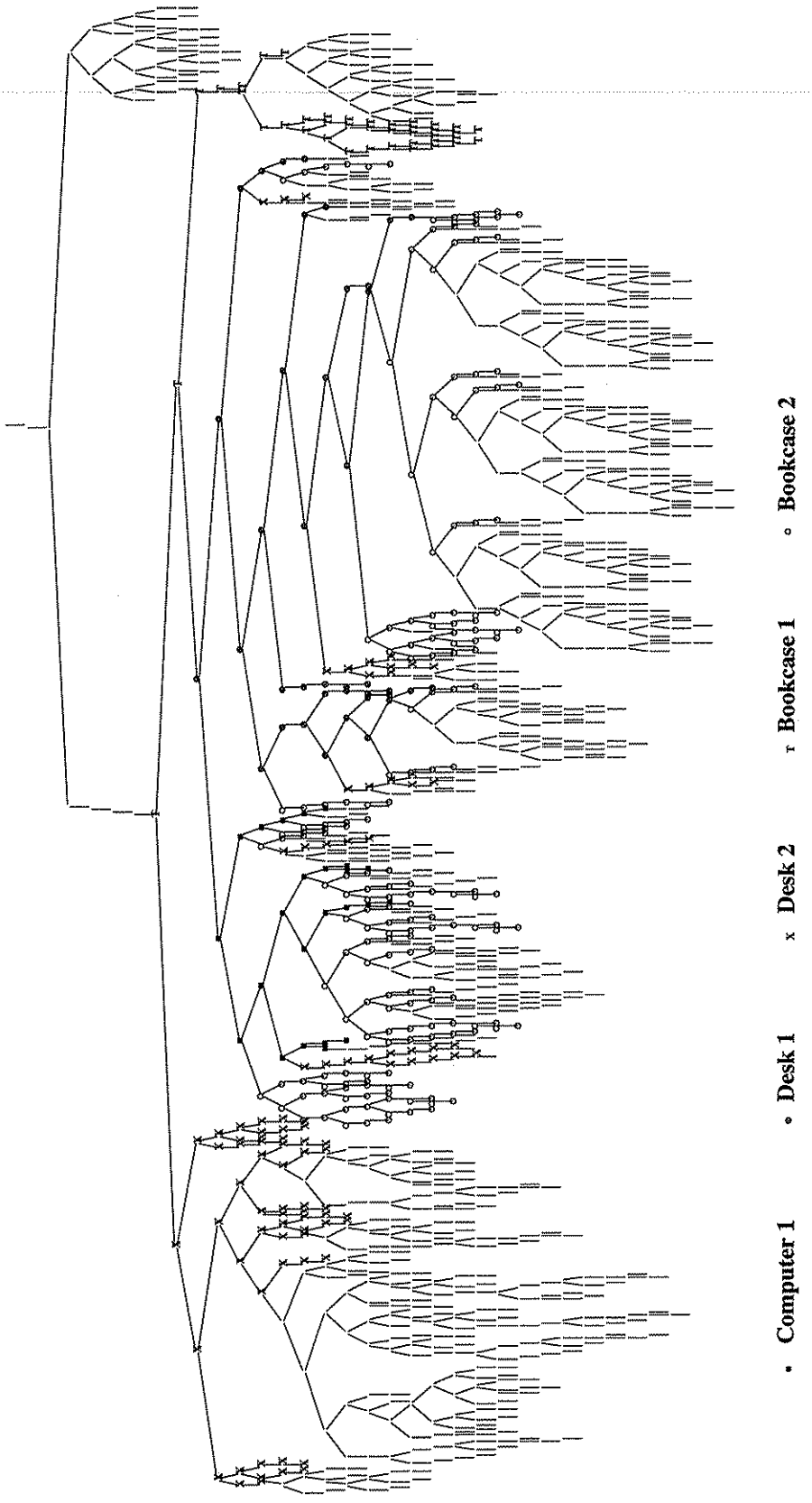


Figure 3.14: Tree for office4 with the objects shown in Table 3.2 marked out

scene	object moved	restoreWeds (ms)	rebuild without	%rebuild
4 cubes	cube 1	21	92	23
	cube 2	11	105	10
	cube 3	10	90	11
	cube 4	9	120	7
7 cubes	cube 1	15	300	5
	cube 3	48	318	15
	cube 4	10	340	3
	cube 5	9	310	3
15 cubes	cube 1	88	732	12
	cube 2	22	760	3
	cube 3	97	745	13
	cube 5	21	750	3

Table 3.4: Timings for initial transformation of objects in the DM-tree

The scenes have 4, 7 and 15 cubes with their shadows generating fairly complex BSP trees with up to 2541 nodes.

The subdivision on the polygon from the cube scenes can be seen in the Figures 3.15 to 3.17. In Table 3.3, we give for each scene the number of cells in the tree and the time spent for building it. Under *cube faces* is the number of polygons in the cube scene *total* and the number of them that created any discontinuities (*active*), even though this is not very relevant for the exercise. Since this example is taken from the algorithm in *Chapter 5*, each set of edges (from each active cube face) is first built into a BSP tree and then merged to the total tree of the polygon. The times given under *Build BSP* are actually only for merging the single trees and does not include their building. A definitely more favorable evaluation of this method can be achieved by including the times for building the single trees as well. These were not included however since it could be argued, that a copy of the single trees from the static objects could be stored and reused when required.

Following the evaluation of *restore*, we compute the performance of this method by comparing the time it takes to restore the tree against rebuilding it without the set of edges created by the transformed object. The results are shown in Table 3.4. Various objects in each scene were transformed and the time to remove their discontinuity edges from the polygon were recorded under *restoreWeds*. In the next column (*rebuild without*), we give the times for rebuilding the tree by

merging the single trees of the sets of discontinuities from the other objects. Finally the percentage time used by the function *restoreWeds* against the function rebuild is given under *%rebuild.*

However large the subtrees were, no nodes were retained. This explains why we have one column of timings under *restore, rebuild without* and *%rebuild.*

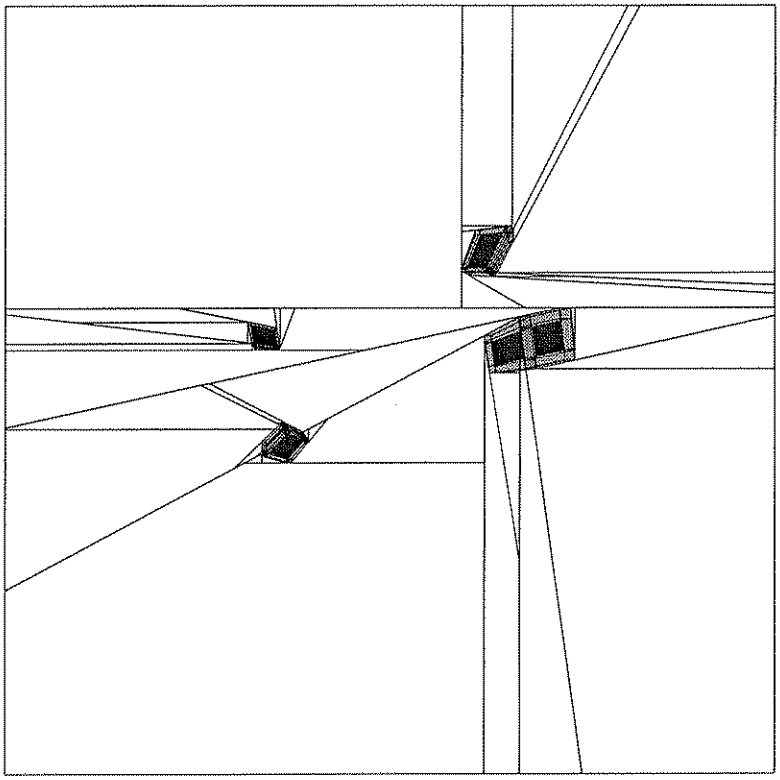


Figure 3.15: The subdivision generated by the shadows of 4 randomly placed cubes

Unlike the previous function (*restore*), where we had coplanar polygons between the static and the transformed object, here we have no collinear edges between the sets. This is because the cubes were randomly placed in space above the floor. In applications showing collinear edges, the performance of the algorithm improves.

Comparison of the Two Algorithms

A direct comparison of the two methods is inappropriate since they operate under very different circumstances. However some observations can be made.

The more complex method used in the *restoreWeds* function shows its strength when the objects are transformed for the first time in a sequence. Even when deleting nodes with large subtrees, it gives considerable gains. For subsequent

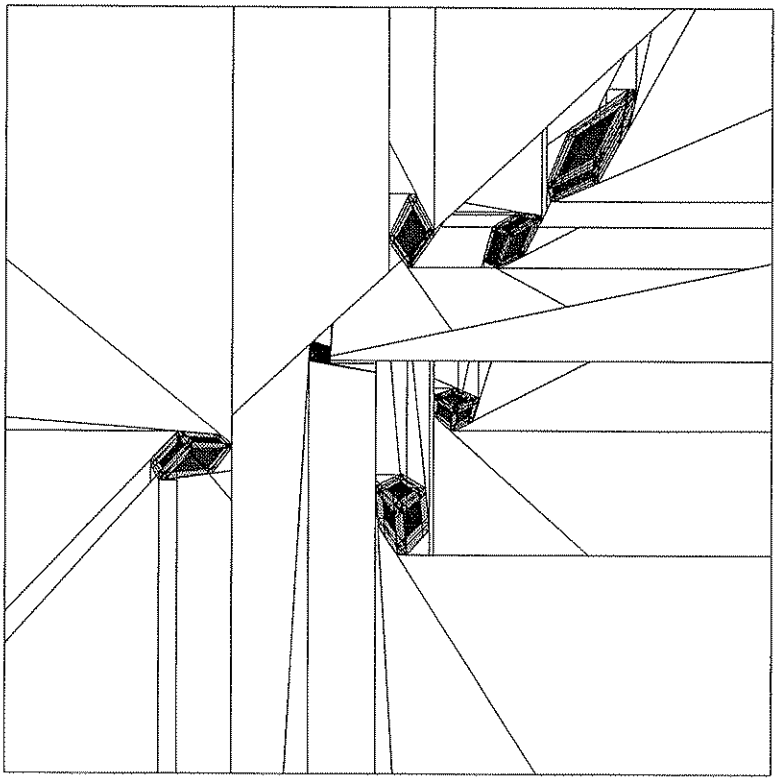


Figure 3.16: The subdivision generated by the shadows of 7 randomly placed cubes

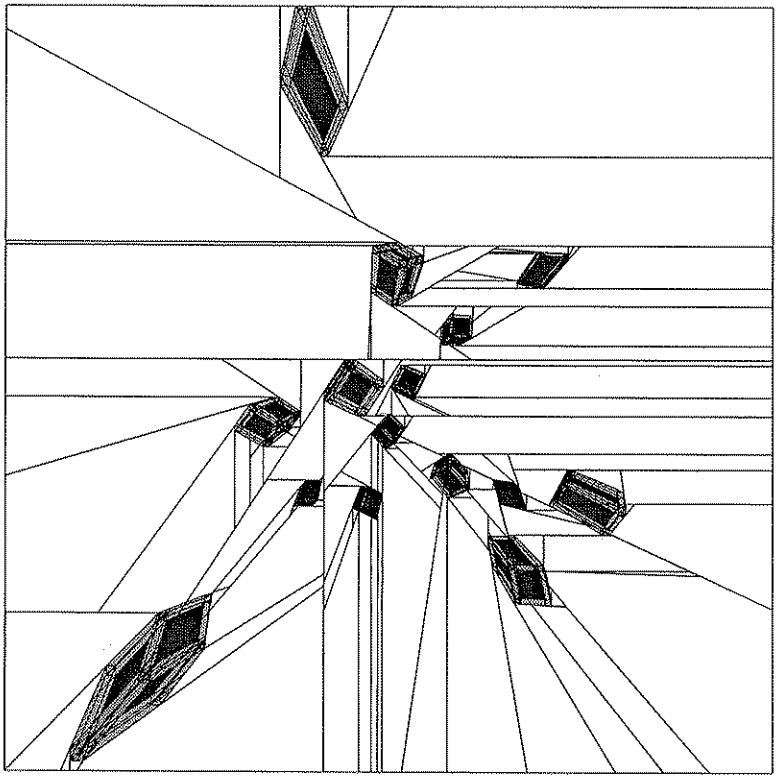


Figure 3.17: The subdivision generated by the shadows of 15 randomly placed cubes

transformations, maintaining the WEDS is an overhead, whereas the simple deletion algorithm used by *restore* works more efficiently.

The major expense for both of these algorithms is the insertion of the object back into the tree. In our experiments this takes a fraction of time proportional to the fraction of polygons (or edges in the 2-D case) in the object moved, compared to the total number of polygons in the scene, regardless of the total numbers involved. As the number of objects and the size of the scene data grow, the gain from these methods also grows, when the size of the moving object remains small. These results are only valid for the range of data and the particular scenes considered. There is no reason not to expect this to generalise, however this remains to be shown.

3.2 Determining the Order with Respect to an Area

One of the principle uses of the BSP tree is for ordering the scene polygons, either for visible surface determination from the viewpoint or from the point light source for shadow calculations. Ordering from area light sources is not as easy. We are not dealing with just one point but a finite area that might fall on both sides of a plane defined at a BSP node.

Previous researchers dealing with area light sources realised this problem but being unable to find a satisfactory solution, their methods resulted in unnecessary processing (see *Section 2.3.1*):

- Campbell and Fussell [13] did not attempt to find an order, assuming that it is not possible, their method requires two passes over the polygon set. One for building the unified shadow volumes, umbra and penumbra, from an unsorted set of polygons using merging and the second for calculating the shadow boundaries by inserting the polygons in these volumes.

- Chin and Feiner [18] simplify the problem of ordering in respect to the area source to one of ordering from a point by splitting the light source whenever it is found straddling the plane of a scene polygon. Thus they can combine the two passes mentioned above into one but performed multiple times, once for each source fragment.

- Gatenby and Hewitt [38], use the BSP tree to find the polygons lying be-

tween the source and a receiver but not by getting an absolute order. Rather they take the union of the polygon sets derived by traversing the BSP tree from each source vertex to the receiver and get an over-estimation of the set of polygons between the receiver and the whole of the source.

The reason traversing a BSP from an area light source (or viewing area) is not the same as for traversing the tree from a viewpoint is because the viewing area cannot necessarily be unambiguously classified against the root plane at each node. Take for example the simple scene of Figure 3.18(a). Traversing the tree of Figure 3.18(b) front-to-back from different points on A gives different orderings: a1 gives $\{2, 1, 3\}$ while a2 gives $\{3, 1, 2\}$. But the different orderings do not imply that there is a cycle or that an order valid for all points on A cannot be found. In fact because we are dealing with oriented polygons and we are only considering a limited viewing space, commonly there will be an invariant ordering, for example $\{1, 3, 2\}$ here.

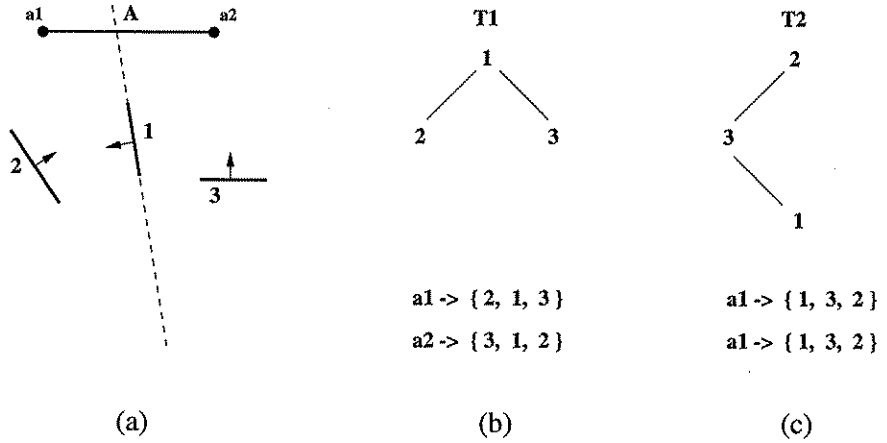


Figure 3.18: (a), (b) When the plane of an internal node cuts the area (A) different orderings are produced for different points on the area. (c) If the cutting node is placed at the leaves then the ordering is the same for every point on A

As we can see in Figure 3.18(c) a different tree can be constructed that when traversed gives that ordering from any point on A. The reason why a tree like T_1 does not work is because different points on A lie in different subspaces of polygon 1 and hence produce different orderings on the children of 1. In T_2 this still holds but since both children of 1 are now empty their ordering is irrelevant.

So an apparent solution to the problem of ordering from a viewing area, is to push all polygons that intersect the viewing area to the leaves of the tree and the traverse it from any point on the area.

3.2.1 Definitions

An interesting study related to this problem can be found in the list priority algorithms for visible surface determination that use pre-processing to induce priority relations among the scene polygons. Of particular interest is the work of Schumacker [89, 97]. He applied graph theory to find an invariant order for a set of polygons (a *cluster*). Along the same lines was also the work of Fuchs [33] and Naylor [67].

Before proceeding let us rephrase some adapted definitions and theorems from the above literature. In what follows s_i and s_j denote polygons:

Definition 1 *Actual visibility obstruction: s_i can have an actual visibility obstruction on s_j, with respect to a viewing vector v, if the vector goes through points t_i on s_i and t_j on s_j, and t_i is closer to the origin of v than t_j, and both polygons are front facing w.r.t. the origin of v.*

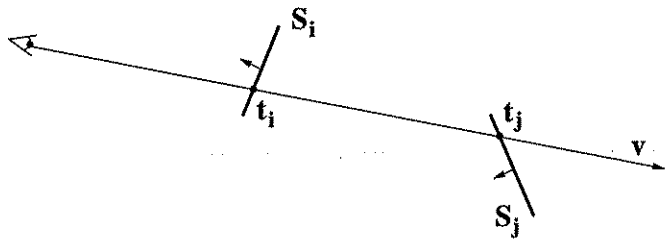


Figure 3.19: Polygons s_i and s_j are related under actual visibility obstruction with respect to v

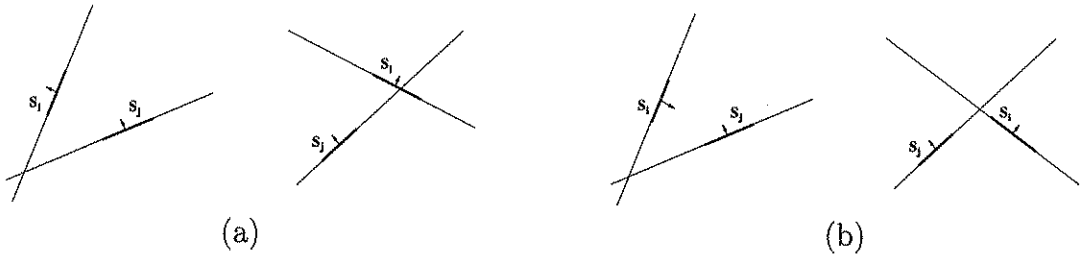
The actual obstruction gives us the visibility priority of two polygons from a particular viewpoint, Figure 3.19. This can be generalised by the following definition to give the priority relation of two polygons from any point in space:

Definition 2 *Potential visibility obstruction: s_i can have a potential visibility obstruction on s_j if $\exists$ a vector v that relates s_i and s_j under actual visibility obstruction.*

Potential visibility obstruction makes no assumptions on the position of the viewpoint so any ordering obtained under that relation is valid for the whole space. This relation between two polygons can be determined using the following theorem:

Theorem 1 *s_i can have a potential visibility obstruction on s_j if and only if $\exists$ a point on s_i in the front half-space of s_j and $\exists$ a point on s_j in the back half-space of s_i.* (See [67] for proof of the theorem).

Applying *Theorem 1* to the arrangements in Figure 3.20 we find that the two pairs on the left are related under potential visibility obstruction while the two pairs on the right are not.



(a) (b)

Figure 3.20: The pairs in (a) are related under potential visibility obstruction while those in (b) are not

3.2.2 Small Viewing Areas

In the research mentioned above, the aim was either to find an invariant visibility order for the whole of 3-D space, or one order for each viewpoint (or sets of viewpoints). Here we are dealing with a sub-problem since we are only interested in the set of viewpoints lying on a pre-defined planar polygonal space. Note that in what follows we assume this area to be convex. For a concave area, it will still hold if we use its convex hull.

Let us now state clearly what it is we will try to achieve:

Given a set of polygons $S = \{s_1, ..., s_n\}$ and a viewing area A we want to build a tree T^* that when traversed from left to right[1] will give a front-to-back priority ordering $<\sigma_1, ..., \sigma_k>$ valid from any point on A, where $\forall\sigma_i$, $\sigma_i \subseteq s_j$, some $1 \leq j \leq n$. For convenience we will call such a tree *ordered*.

This ordered tree is equivalent to a linear list of the polygons as seen front-to-back from A. As we will see later it is not always possible to generate this tree without splitting A since cycles may be present. In general, however, under the conditions we will be assuming it will be feasible. Any unbreakable cycles will be reported.

The method has two stages. First a BSP tree T' is constructed in the conventional way but only with the polygons in S that have A totally in their front half-space. The polygons that intersect A with their plane are stored in a list L_o while those that embed A or have A totally behind their plane are ignored. We will refer to the polygons that have A totally in-front as *area-facing* and to those that cut A with their plane as *offending*. Clearly traversing T' from any point on A gives the same ordering. Any point on A is in-front of the plane defining each node so traversing the tree from left-to-right is equivalent to a front-to-back traversal.

[1] Assuming that the left is the front subtree and the right is the back

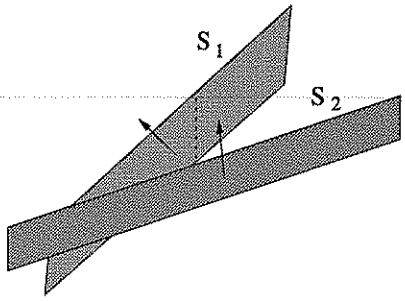


Figure 3.21: Cycle of two polygons

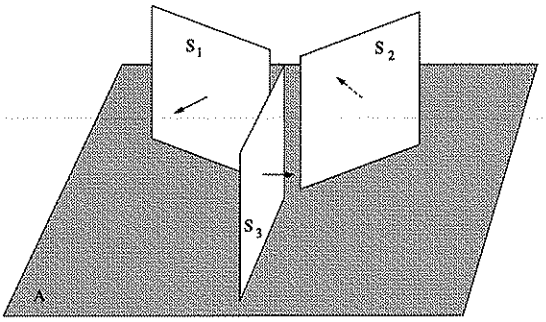


Figure 3.22: Cycle of more than two

The second step is to construct T^* by inserting the polygons in L_o into T'. It is easier to understand why T^* is ordered by thinking of T' as an ordering of subspaces C_i (cells) as seen from A. Any additional polygons inserted into T' will further subdivide the cells but cannot change their relative priorities. Thus it suffice to show that polygons within each C_i are ordered.

Since the area we are considering is the light source, which is relatively small, the number of polygons cutting it with their plane are expected to be few compared to those facing it. So the polygons in L_o will probably reach different cells or maybe few in the same cell. When an offending polygon is the only one to reach a leaf node of T' then we just create a tree node and add it there. For cells with more than one offending polygons, we use a graph theoretical approach, described below, to order them and produce a subtree to replace the cell.

Each polygon in the cell forms a node of the graph and has *arcs* connecting it to all other graph-nodes that it has priority over. The priority relation is determined using *Theorem 1.* We will denote this relation using $\rightsquigarrow$, $s_i \rightsquigarrow s_j$ meaning that s_i has a potential visibility obstruction (PVO) on s_j. Once the graph is built it is searched using a depth first search (DFS) to order the nodes and check for cycles. If cycles are detected a second DFS is performed on the transpose of the graph (the arcs are reversed) to locate the cycles and reduce it to its strongly connected components [27]. In the scenes we used for *Chapter 5* this second search was never needed. If the cycle is a trivial one involving only two polygons (Figure 3.21) then it can be resolved by cutting one of the polygons along the plane of the other. A method for dealing with other cycles is suggested in *Section 3.2.4.*

Building a tree using the arcs in the graph is an $O(n^2)$ process, where n is the number of polygons involved, but as these are very few this is not detrimental to the speed of the whole algorithm.

Such a graph theoretical approach was used by Schumacker [89] to order the polygons in each cluster. It was also used in [67] for generating a total ordering on the whole set of polygons but it was found impractical due to the large number of cycles involved. Our problem, however, is not as demanding since we are only interested for visibility priority from a limited area.

The tree we have constructed here excludes any back-facing polygons, with respect to A. This is not a requirement of the algorithm and if the tree is to be used for other operations then the complete set of polygons can be included. In this case the back-facing polygons are marked so as to traverse the right subtree before the left, when getting the order with respect to A.

The order for visible surface determination from any point in space, can be obtained by a normal back-to-front BSP traversal (classifying the viewpoint at each node) on the nodes of T' and right-to-left on the offending subtrees.

To use such a complete ordered tree from a different viewing area A', all that is needed is to identify the offending polygons for A' and push them down to the leaves. They do not have to be reinserted at the root of the tree but rather from the node where they lie. The priorities on the previously-offending subtrees are still valid but if the addition of the now-offending polygons creates cycles they will have to be rebuilt.

3.2.3 Maintaining the Order During Interaction

For interaction we assume the use of the algorithm described in *Section 3.1.1*: the polygons of the transformed object are removed from the tree and they are added back at their new positions. When removing polygons from leaf-nodes or from nodes with one non-empty subtree it can be shown trivially that the ordering in the tree is not affected. But while putting two subtrees together in the *restore* function or as the new polygons are added, area-facing polygons might reach an offending subtree. In order to maintain the assumptions we made earlier, area-facing polygons must go before any offending polygons in the tree. So when inserting an area-facing polygon, if it meets a subtree with offending polygons then the area-facing one is placed at the root of the subtree and the offending polygons are pushed down past it one by one.

3.2.4 A Generalisation of the Method

As mentioned earlier the visibility is restrained to a particular area, even if the area is large, the fact that it is limited and planar places an extra constraint on the visibility relation which can be used to resolve most of the cycles.

The arcs in our graph denote the potential visibility obstruction. This relation is very broad, we are not interested in whether there exists a point in space for which two polygons can have an actual visibility obstruction but what we are interested in is if there exists such point on A. We will give a new name to this relation in the following definition:

Definition 3 *Distributed visibility obstruction: two polygons, s_i and s_j, are related under distributed visibility obstruction with respect to a planar polygon A, if $\exists$ a point p on A such that s_i and s_j are related under actual visibility obstruction with respect to that point.*

Obviously, distributed obstruction is a subset of potential visibility obstruction. It cannot create any extra arcs that are not present already. But by validating the existing arcs against this new relation their number can be reduced and cycles resolved. The existence of distributed visibility obstruction (DVO) can be determined by the following theorem:

Theorem 2 *Given 3 polygons A, s_i and s_j, s_i can have distributed visibility obstruction on s_j if s_i has potential visibility obstruction on s_j and $\exists$ a point on s_j behind all of the extremal penumbra planes from A and s_i.*

Proof: By definition (see *Section 2.3.1*), the extremal penumbra planes have the "source" totally in the front half-space and the "occluder" totally in the back half-space. Here the source is A and the occluder is s_i. If s_j falls in front of any penumbra plane then $\{A, s_j\}$ and $\{s_i\}$ are linearly separable by that plane, eg plane A_1 in Figure 3.23. This implies that there can be no line going through a point on A and a point on s_j that can be intersected by s_i. Thus there is no point on A from which s_i and s_j can have an actual obstruction. □

Using the same reasoning we can also show that DVO between A, s_i and s_j can exist only if A has some intersection with the subspace defined by the intersection of the extremal penumbra planes of s_i and s_j and the front half-space of s_i. This penumbra structure is similar to the *obstruction polyhedron* defined in [67].

The test for distributed obstruction is more expensive than checking for potential obstruction. If we are expecting only few cycles then we can build the

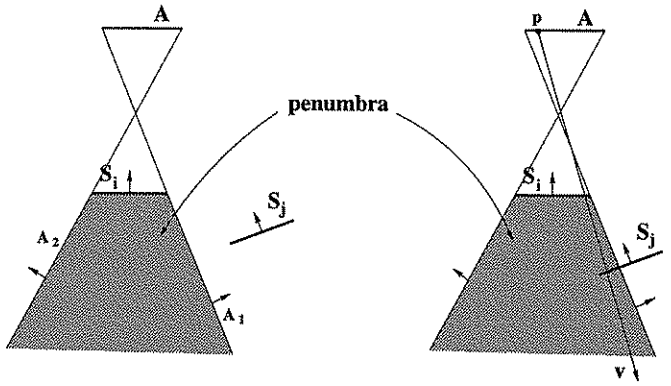


Figure 3.23: s_i can have distributed visibility obstruction on s_j with respect to A only if s_j lies, at-least partly, in the penumbra of s_i

arcs in the graph using only PVO, and only apply DVO to verify them after the polygons involved in the cycle are found. Also we can create fewer splits if we verify an obstruction before cutting a polygon involved in a cycle of size two, mentioned in the previous section.

We might find examples, such as the one in Figure 3.22, where cycles occur that can not be resolved by splitting the polygons or otherwise. In this case the actions taken depend on the application. For example in the area sources algorithm described in *Chapter 5*, we can assign the same priority number to all elements of a cycle. When deciding the shadow relations, polygons with the same numbers will be compared both ways (see the inner loop in Figures 5.1 and 5.2).

If this is to be used for rendering with the viewpoint moving on A, then we can find the polygon(s) whose removal will break all the cycles and use their planes to split A. The BSP nodes of these polygons are marked and their planes are checked when crossing from one fragment of A to another. The subtrees of the node whose plane is crossed are then switched (left goes right and vice-versa).

To use a tree constructed using DVO from a different viewing area A' the offending subtrees should be rebuilt. Also it can not be used for rendering from any viewpoint not on A.

3.2.5 Results

For evaluation of the method the office scenes were used. A relatively large light source was placed at the center of each room, near the ceiling and the tree was built with respect to that source. In Table 3.5 we give statistics on the total number of polygons (*N_pol*) in the scene, the number of polygons that have the

scene	N_pol	F_area	O_area	Max_O	N_cycles
office1	139	58	17	4	0
office2	211	96	19	3	0
office3	333	147	28	4	0
office4	751	336	65	4	0
office4*	751	327	70	6	0

Table 3.5: Visibility from the normal light source

scene	N_pol	F_area	O_area	Max_O	N_cycles*
office1	139	51	41	8	0
office2	211	74	64	11	0
office3	333	110	108	31	0
office4	751	240	241	66	0
office4*	751	227	259	46	1

Table 3.6: Visibility from a light source with half the dimensions of the ceiling

area totally in their front half-spaces (*F_area*) and number of polygons that cut the source with their plane (*O_area*). The next field (*Max_O*) gives the maximum number of offending polygons in any cell of T'. As we can see this number is very small, not exceeding 6 in any of the scenes. Finally we give the number of cycles found. For determining the visibility relations between the offending polygons, we used only potential visibility obstruction (*Theorem 1*). We can see that no cycles were present in any of the scenes.

The corresponding trees for these scenes office1, office3 and office4 are shown in Figures 3.24 and 3.25. To show clearly the double structure of the tree, the BSP (T') nodes are marked with white squares (these are from the area-facing polygons) while the graph theory based nodes are marked with black squares (the offending polygons).

To see how the algorithm behaves when we use large sources (or viewing areas), two more sets of experiments were run: one with a source having sides equal to half that of the ceiling and one with using the ceiling as the source. The data of these experiments are shown in Table 3.6 and Table 3.7. The columns in these tables are the same as in Table 3.5, apart from the last column (*N_cycles**). This corresponds to the number of cycles in the tree with the arcs created on distributed visibility obstruction (*Theorem 2*) rather than potential obstruction.

scene	N_pol	F_area	O_area	Max_O	N_cycles*
office1	139	26	91	32	0
office2	211	39	143	64	0
office3	333	59	225	96	0
office4	751	129	497	227	17
office4*	751	122	509	191	25

Table 3.7: Visibility from a light source with the dimensions of the ceiling

As the size of the source becomes larger the number of polygons intersecting it with their planes (offending) grows rapidly (*O_area*). Also the number of offending polygons reaching the same cell of T' grows (*Max_O*), making their ordering, using only potential obstruction impossible. This is the same problem recorded by Naylor in [67]. However, using the extra constraint provided by DVO, we can see that almost all of the cycles are resolved. Only in experiments with office4 and office4* do cycles remain. The reason for that is that the area used for visibility (the source) here is much larger than in the other experiments and in particular it is very large compared to its distance to the other objects in the scene. In further experiments using the same data but pushing the ceiling to twice the height we found that all the cycles disappeared.

The tree for office4 in Table 3.6 is shown in Figure 3.26 and for office1, office3 and office4 of Table 3.7 in Figures 3.27 and 3.28. As we can see the double structure of the tree is almost lost in the trees for Table 3.7. Also in the latter tree cycles were present. These cannot be seen on the tree because we stored all polygons in each cycle on the same node. So the nodes of the tree are ordered but some of them hold polygons that form a cycle.

We will not give the timings for building the tree as the purpose of the experiment was not to build the tree fast but to show that an ordering in respect to an area was attainable. Also in applications the tree would be built in the pre-processing stage so it would not affect the run-time performance. But to give an idea of the times involved we can say that for Table 3.5, where the reference area is a normal light source, the timings were less than building a normal scene BSP tree. This is because the back-facing polygons were excluded. For Table 3.6 it took slightly longer, while for Table 3.7 it took many times longer than for building a normal tree. These results were predictable since, as we said in *Section 3.2.4*, ordering the polygons in a cell is an $O(n^2)$ operation and in Table 3.7 n is very large.

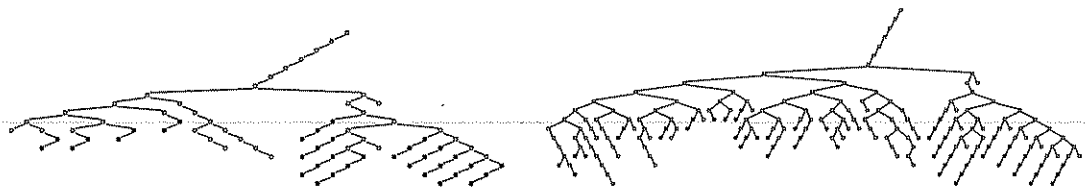


Figure 3.24: Trees of office1 and office3 from the normal light source

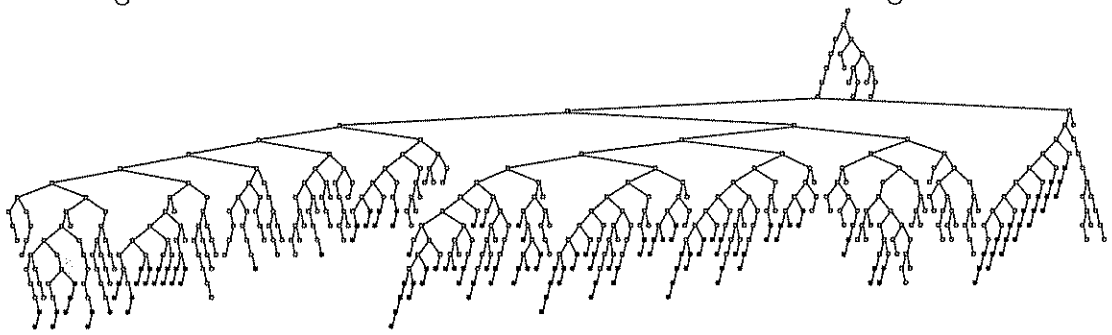


Figure 3.25: Tree of office4 from the normal light source

Interpolating from the above results we can draw the following conclusion: that even for complex scenes an ordering from an area light source can be derived using this method with few or even no splittings of the source.

3.3 Summary

In this chapter we have considered issues regarding the use of BSP trees for:

(a) Representing dynamic scenes. The BSP tree is incrementally updated at each scene modification by removing the polygons of the transformed object and reinserting them at their new position.

(b) Ordering polygons in respect to an area. A two level tree is constructed with respect to the area in consideration. The top level is a BSP tree of the polygons facing the area. The bottom level is a linear order of the polygons cutting the area lying in the leaves of the BSP tree.

Experimental results have shown that:

(a) The incremental update of the BSP tree can be performed in a small percentage of the time required to rebuild it, especially if only a small part of the scene changes.

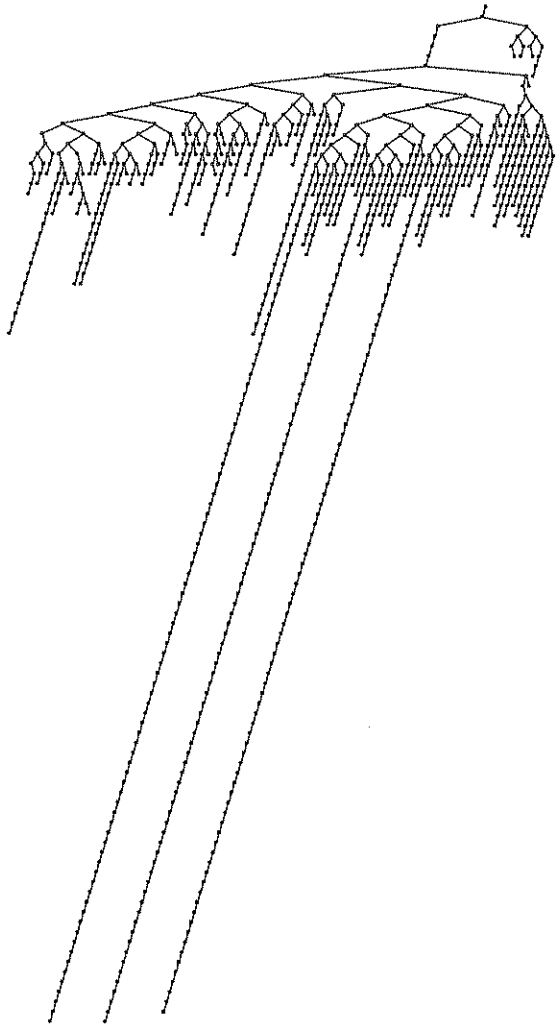


Figure 3.26: Tree of office4 with light having half the dimensions of the ceiling

(b) An ordering for small areas, such as the light source used in *Chapter 5*, always existed for the scenes tested. If cycles in the priority relation are present (e.g. for larger areas or more complex scenes), they will be found along with a minimal splitting set for the area to resolve them.

In the next chapter we will present algorithms that use the BSP tree in dynamic scenes to produce shadows from point light sources.

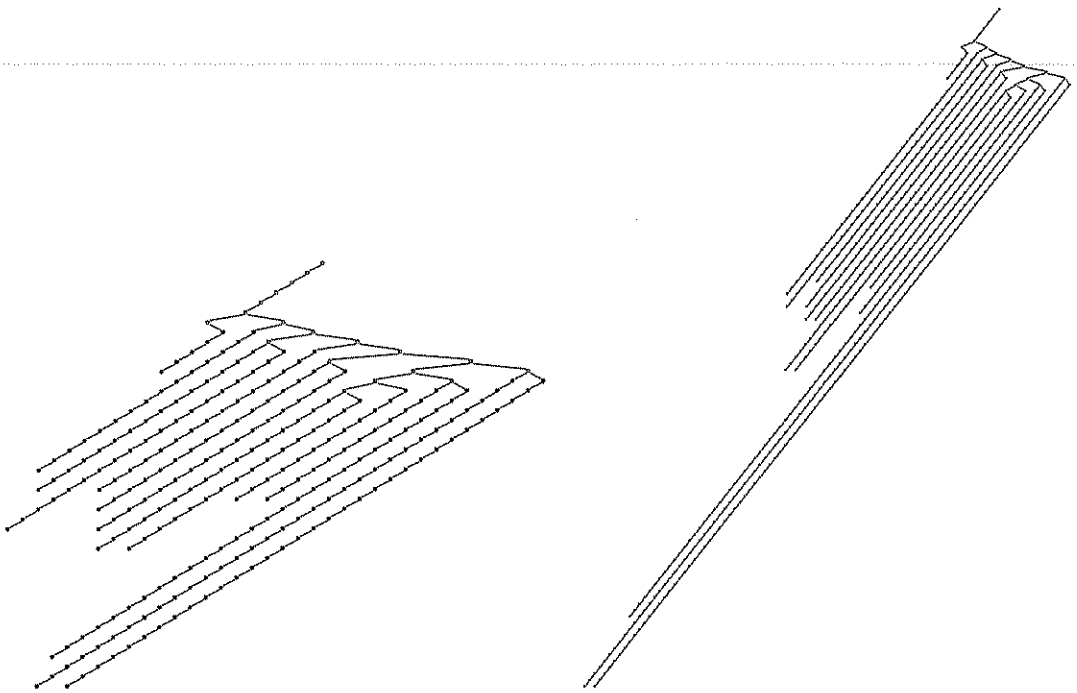


Figure 3.27: Trees of office1 and office3 with light having the dimensions of the ceiling

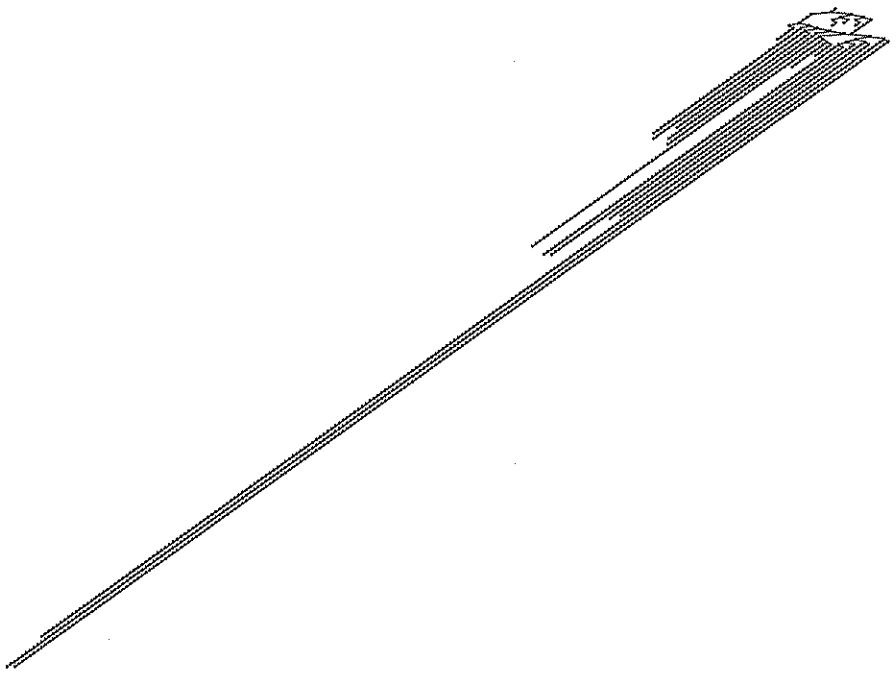


Figure 3.28: Tree of office4 with light having the dimensions of the ceiling

Chapter 4

Dynamic Scenes Illuminated by Point Light Sources

Many interactive applications could benefit from the addition of shadows, even if these are produced by point light sources. In this Chapter, two shadow volume algorithms are extended to deal with moving objects. Shadow volumes were chosen because they operate in object space and the solution is independent of the viewing position. This is a desirable attribute for animation/interaction, where the viewpoint changes frequently. We will assume that in the applications where this method is used speed is critical. In both algorithms, the shadows are stored as detail polygons on-top of the original scene polygons. This allows for easy deletion/addition of shadow polygons when objects are moved. Of course this is not inherent in the algorithms. A mesh such as the one used in *Chapter 5* could be used.

The general idea of shadow volumes is that any part of a polygon falling within the shadow volume of some other polygons is in shadow. Both of the algorithms to be described use subdivision of space to avoid the $O(n^2)$ comparison of all polygons against the SV of all other polygons.

4.1 Shadow Tiling

The shadow tiling method as described in *Section 2.2.3* makes use of the BSP tree to order the polygons in respect to the light source. Following the dynamic BSP trees in *Section 3.1* we can deduce a straight forward extension to the tiling algorithm for dealing with moving objects.

Initially the tiling is built as for the static case: the BSP tree is traversed to

get the front-to-back order and the polygons are projected and scan converted onto the sides of the cube. During scan conversion with each polygon (P) we record the identifiers of any other polygons already in the tiles it visits. These are stored along with the polygon in its active polygon list (APL). All polygons in the APL are closer to the source than P, since they are being processed in front-to-back order. Hence P is compared against the shadow volume of each of these for a shadow. The only difference from the static method is that some extra information has to be stored that will be used later for deletion from the tiling. Each polygon needs to hold the following:

(a) The tiles it intersects on the cube. If memory is a problem then we can re-scan the polygon to find the tiles when it will be deleted but it is faster if we just store them when the polygon is inserted.

(b) A list of faces it casts shadows on.

(c) As described earlier, shadows are stored as detail polygons on-top of each receiver. Each such shadow polygon needs to record the identifier of the occluder polygon that caused it.

When an object is transformed during interaction we need to remove its polygons from the BSP tree and the tiling cube and then reinsert them in both. The order in which this is done is seen in Figure 4.1. First each polygon in the object is marked on the BSP tree and removed from the tiling. To remove a polygon from the tiling we use the information described above:

(a) Is used to delete its identifier from the cube.

(b) Is used to delete the shadows it generates.

(c) Is used to delete its identifier from the set of polygons which cast shadows upon it.

Then the polygons are removed from the BSP tree using the functions described in *Section 3.1*, the object is transformed and the polygons are inserted back into the tree, marked as *new*.

Once the tree is complete, we can traverse it to obtain a new front-to-back list of the polygons with respect to the light source position. This list will contain some polygons, those marked as *new*, that do not have any shadows yet. The

```
Tree transformObject(Tree bsp, Object obj)
{
/* remove the object faces from the tiling */
/* cube and delete their shadows */
        for each polygon p_i in obj do
                mark p_i on bsp;
                remove p_i from tiling cube;
                remove shadows of p_i
        endfor
/* transform the polygons update the BSP */
        bsp = restore(bsp);
        getNewObjectGeometry(obj[]);
        for each polygon p_i in obj do
                add p_i to bsp;
        endfor
        /* get new polygon order from light */
        order[] = traverseBSP(bsp, light);
/* cast the shadows of the object faces */
        newShadows(order[]);
        return bsp;
}
```

Figure 4.1: Transforming an object using the Shadow Tiling

```
void newShadows(order[])
{
    for each polygon p_i in order[] do
        if new(p_i)
            project p_i on cube, and compute apl_{p_i}[];
            for polygon p_j in apl_{p_i}[] do
                if orderno(p_j) < orderno(p_i)
                    cast a shadow from p_j to p_i
                else
                    cast a shadow from p_i to p_j
                endif
            endfor
        endif
    endfor
}
```

Figure 4.2: Adding polygons to the tiling cube

function *newShadows* (Figure 4.2) is called that will go through the list and put each such polygon in the tiling cube.

Unlike the initial generation of the shadows, the APL of the new polygons can contain polygons both closer and further away from the source. We can distinguish between these from their positions in the new order list, which is given by the function *orderno(p_i)* in Figure 4.2. Polygons closer to the source will have lower order numbers. These will be used to cast shadow on the new polygons while others with greater order numbers will have shadows cast from the new polygons.

A disadvantage of this method compared to the original described in [92] is that no distinction is made here between the interior and the boundary tiles. All tiles are treated as boundary. If we use the idea of marking the interior tiles and do not process any further those polygons projecting only on them, then some shadows might not be cast. For the scenes this is fine since these uncast shadows will be over areas already covered by the polygons of the interior tiles. In dynamic scenes these polygons might move and uncover such areas causing errors in the shading.

It is possible, however, to have an algorithm that uses interior tiles. This algorithm can mark the polygons that scan convert totally these tiles as "potentially casting", but will not use them to cast any shadows and does not need to compare them against any SV since we know they are totally shadowed. When the

occluder of the interior tiles is removed then any "potentially casting" polygon in its APL will be processed again. To keep track of potentially casting polygons we would still need to enter all the identifiers in the tiling cube so the gain is limited and also the algorithm becomes more complicated. This has not been implemented.

Multiple light sources can be modelled by maintaining a separate tiling cube and the associated information for each source.

4.2 Unordered SVBSP Tree

The tiling algorithm described above provides a great speed-up compared to the alternative, which is calculating everything from scratch. The fact that it has to rely on the BSP tree for sorting the polygons from the light source is a disadvantage for large environments.

An alternative algorithm which has similar performance for static scenes [92] is the SVBSP tree. In its standard implementation it also uses the BSP tree for sorting from the light source. This is not, however, essential. In this section we show how to use a SVBSP tree built from an unsorted set of polygons for producing and maintaining shadows in a dynamic scene.

4.2.1 Building the Unordered SVBSP Tree

The standard SVBSP tree is built from an ordered set of polygons so there is no question as to which polygon is closer to the light source when the SVs of two polygons intersect. Building the tree using only the shadow planes is sufficient. For the unordered SVBSP tree the scene polygons themselves must be added to convey that information, so the SV of each individual polygon is complemented with the polygon plane. Since nodes containing shadow planes and nodes containing polygon planes are treated differently by the algorithm we will distinguish between them by calling the former SP-Nodes (Shadow Plane Nodes) and the latter PP-Nodes (Polygon Plane Nodes).

The algorithm uses a copy of the scene polygons in the tree for calculating the shadows which are then stored as detail on-top of the actual scene polygons.

The tree is built incrementally by inserting the light-facing polygons into an initially empty tree, a single OUT node (Figure 4.3). The first polygon just replaces that node with its SV (Figure 4.4). Subsequent scene polygons are

filtered into the tree by comparing them at each level against the plane at the root of the tree and recursively inserting them into the appropriate subtree. If they straddle the root plane then they are split and each piece is treated separately. When an OUT node is reached it is replaced by the SV. If the polygon was split its SV is built using the shadow planes of the original polygon (polygon 4 in Figure 4.6). This is necessary for dynamic modification and it also means that the SV needs to be calculated only once even if a polygon is split into many pieces.

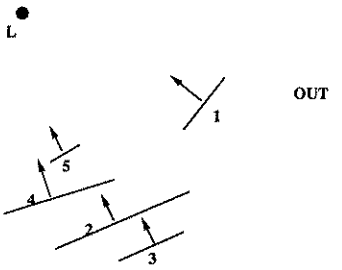


Figure 4.3: Initial scene

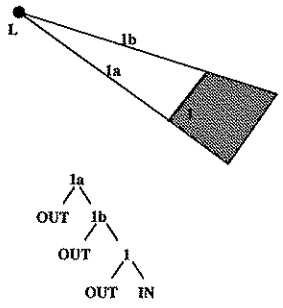


Figure 4.4: Insert poly 1

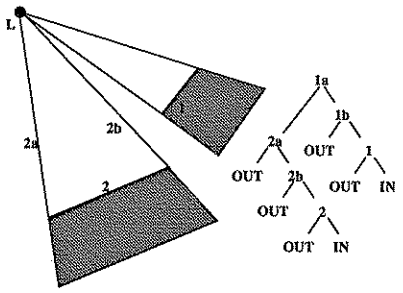


Figure 4.5: Insert poly 2

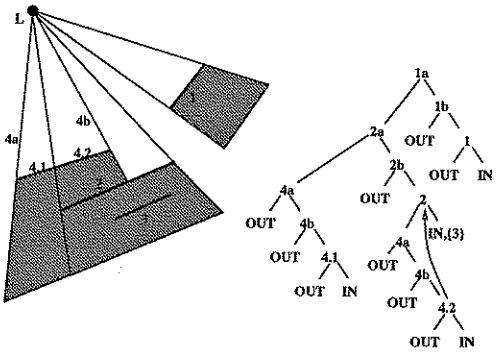


Figure 4.6: Insert poly 3 and 4

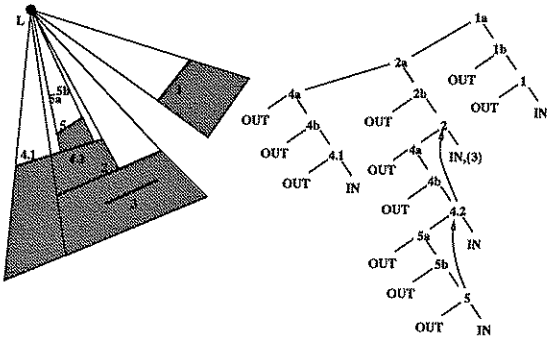


Figure 4.7: Insert poly 5

When a PP-Node is encountered, if the inserted polygon is classified as behind its plane then it is marked as shadowed and stored there (face 3 in Figure 4.6). If it is classified as in front then it takes note of the face at the root as a potential receiver and it is inserted into the front subtree. If it reaches an OUT node then a shadow is cast on the face stored as potential receiver (face 2 in Figure 4.6). If it comes in front of more than one potential receiver, only the last one is remembered and used (polygon 5 in Figure 4.7 comes in front of 2 and then in front of 4, a shadow is casted only on 4).

To cast a shadow onto a receiver, the original scene polygon of the receiver is clipped against the relevant SV. Likewise, as mentioned above, the SV is always

defined by original scene polygons which means that the fist time two polygons or, fragments of them, meet, the whole shadow between them is cast. This way a lot less shadow polygons are created, but to avoid duplication the occluder is marked so that it will not cast a shadow again on the same receiver.

The pseudo-code for building the unordered SVBSP tree is given in *Appendix B.*

4.2.2 Using the Tree for Dynamic Shadow Computation

In an interactive application where the scene geometry changes, the tree can be used to maintain the correct shadows.

During the building of the tree, each inserted polygon constructs a list of pointers to the locations it occupies on the tree. When an object is transformed, its polygons and their shadow planes on the tree are found using the location lists and are marked. After all relevant polygons have been marked, a recursive function is called that will iterate through the SVBSP tree once and remove all marked nodes. The result of this will be a valid SVBSP tree for the scene, but now without the transformed object. The object can then be reinserted into the tree using the algorithm described in *Section 4.2.1*, to get the shadows at its new position.

A detailed description of the procedure in pseudo-code form is given in the *Appendix B.*

Removing the Marked Nodes

The function used for removing the marked nodes works on the whole SV of polygons rather than on single nodes. There are 3 possible positions for each polygon and its shadow volume to consider:

(a) In the IN region, behind a PP-node (no shadow planes were attached here, just the polygon). This is the simplest case, the polygon is just removed (polygon 3 in Figure 4.7).

(b) At the leaves, subdividing an empty subspace. Again this is simple, the SV is replaced by an OUT node. Care must be taken if the PP-Node had a non-null receiver. This occurs when it is in front of some other PP-Node during insertion and it is now casting a shadow on this. In this case the

shadow must be removed. For example when deleting polygon 5 in Figure 4.7, the front (left) subtree of node labeled 4.2 should be replaced by OUT and the shadow on polygon 4 should be deleted (the arrows there show the receiver relation).

(c) Splitting a non-empty subspace, the SV forms the root of a larger subtree. This is the only relatively complex case. Removing it would result in unconnected sub-trees and these must be put together to form a new tree to replace the old one. This is similar to the fourth case in *Section 3.1.* If the deleted polygon was casting a shadow then that must be replaced by shadows from polygons that had the deleted one as target. These can only be in the front subtree of the deleted PP-node. For example if polygon 4 in Figure 4.7 is deleted then the shadow from 4 to 2 should be replaced by a shadow from 5. Any polygons that were in shadow, in the IN region behind the deleted polygon, must also be inserted into the new unified subtree.

Joining the Subtrees

The fact that the polygons and their shadow planes come in *clusters* led us to first try the merging algorithm as described in *Section 2.1.4.* After several experiments we came to the conclusion that merging is too slow for our purposes here. The main reason is that it requires calculation of the sub-hyperplanes of the shadow planes involved in the merging *Section 2.1.1.* Also it is very general, does not utilize the fact that all the shadow planes emanate from the same point (the light source).

A more specialized algorithm is used here. The largest of the trees to be merged is found, say T_1, and any possible marked nodes on this are removed. The inserted tree, T_2, is then treated as a set of shadow volumes. The polygon node (PP-Node) of the shadow volume forming the root of T_2 is found and filtered down T_1 along with its front and back subtrees. The filtering is done in a similar manner to a polygon. The fact that all shadow planes go through the light source position ensures that anything enclosed by a polygon's shadow volume can be split by another shadow plane, only if the polygon itself is split. This means that the front and back subtrees need to be checked for intersection with a plane only if the polygon is split by that plane. If the PP-Node meets another fragment of its own original polygon then it stops there and its front subtree is inserted into the front of the tree node (this is possible since the shadow planes used by the fragments are those of the original). When it reaches an OUT node

its SV is attached. After the 'root' SV and the subtrees of its PP-Node have been inserted, the algorithm is called recursively to insert the front subtrees of its SP-Nodes.

Note that the subtrees involved here are linearly separable by the deleted planes which embed the light source. No ray starting from the source and going in any one direction can intersect more than one of these subspaces so there is no shadow relation between them. Also, if polygons split or come together during the merging, the shadows on them or the shadows they cast do not change.

4.2.3 Further Discussion

As for the tree described in *Section 3.1*, when a target object is being continuously transformed, for example as a result of being dragged during an interactive application, the functions described in *Section 4.2.2* are only relevant for the very first transformation. After the first deletion and re-insertion, the faces will end up at the leaves and in subsequent frames can be deleted in constant time.

In the standard SVBSP tree the smaller objects which are usually placed on top of other larger ones tend to be higher up the tree because they tend to be closer the light source. This is the order that is obtained from the scene BSP tree traversed from the light position. Also their polygons may be widely distributed in the tree (Figure 4.10). Moreover these smaller objects are the ones most likely to be selected and transformed during an interaction.

In the method described here, the polygons may be grouped together according to the object to which they belong and are given to the SVBSP tree in that order. Therefore there is greater probability that those polygons belonging together will be grouped together in the SVBSP tree (Figure 4.11). Also the smaller objects can be inserted last. For Figure 4.11 the objects were inserted in depth-first order in relation to in the scene hierarchy (Figure 4.9).

Again, as in the case of the BSP trees in *Section 3.1*, a small proportion of shadow planes in the tree are responsible for most of the splitting. Removing these, when their polygons have moved, could be an expensive operation. This can be avoided by leaving these nodes in the tree as marked, and not removing them if their subtrees are found to be too large. They are removed eventually when later transformations make their subtrees sufficiently small.

More than one light source can be modeled by creating a separate SVBSP tree for each. The input for subsequent sources are the initial scene polygons and

scene	scene polygons		normal tiling		adjusted tiling	
	initial	after BSP	time (s)	shadow pol	time (s)	shadow pol
office1	133	164	.43	130	.49	161
office2	211	258	.63	215	.73	275
office3	313	537	1.26	502	1.52	624
office4	745	1816	4.57	1773	5.54	2386

Table 4.1: Timings for calculating the shadows using tiling

their shadows.

4.3 Results

Both algorithms were written in C and implemented on a SUN SparcStation 2+ with the same specifications as in *Chapter 3*. The scenes used for evaluation were again the office scenes as described in *Chapter 3*, shown in *Appendix C*.

To evaluate the methods we computed the time taken to update the shadows of a transformed object (delete the old shadows and find the new shadows) and compared this against the time taken for recalculating the shadows for the whole scene.

4.3.1 Tiling

In Slater's [92] initial tiling method where each projected polygon is scan-converted onto the cube sides, the tiles that are fully covered by the projection (internal) are marked as blocked, and no polygon identifiers are added to them. In our implementation this optimization was disabled since it would have caused omission of shadows that might be apparent after a modification. Due to this, the initial tiling method (we call it *normal tiling*) generates less shadow polygons and takes less time than our implementation (*adjusted tiling*).

The evaluation was done by comparing the update timings against the times taken by both methods.

In Table 4.1 we give the data for the initial calculation of shadows. Under *scene polygons* we show for each scene the initial number of polygons and the resulting number after the construction of the BSP tree. Under *normal tiling*, we show the time used and the number of output shadow polygons created by

scene	object moved	object polygons initial	final	%scene	move object (s)	compared to N-Tiling (%)	compared to A-Tiling (%)
office1	computer	20	27	16	0.07	16	14
office2	computer	20	27	10	0.08	13	11
	bookcase	54	54	21	0.25	40	34
office3	computer	20	35	7	0.07	5	4
office4	computer	20	27	2	0.21	5	4
	desk 1 & comp	56	79	4	1.05	23	19
	desk 2 & comp	56	115	6	0.95	20	17
	bookcase	54	66	4	0.50	11	9

Table 4.2: Transformation timings for the tiling method

calculating the shadows with the normal tiling method. Finally we give the corresponding timings and numbers of output shadow polygons resulting from our implementation (*adjusted tiling*).

A number of objects were selected and transformed. The timings for these transformations are given in Table 4.2. For each object moved, we show the number of polygons prior and after adding objects to the scene BSP tree and the percentage of the scene they represent (*%scene*). The time taken by each transformation is first given in seconds (*move object*), then as a percentage of the time taken to rebuild the shadows using the normal tiling method (*compared to N-tiling*) and finally given as a percentage of the time taken to rebuild the shadows using the adjusted tiling method (*compared A-tiling*).

It is apparent from these tables that small objects can be transformed particularly fast. In all scenes, the computers, for example, take a percentage of time comparable to the ratio of their polygons against the scene polygons.

Other objects however, like desk1 and desk2 in scene4 take a greater percentage of time than their corresponding ratio of polygons. We can see in Table 4.4, where the same objects were transformed using the SVBSP algorithm, these objects also take longer than their size alone explains. This is because the algorithm does not only depend on the size of the object but also on the number of shadows produced. These objects cast shadows on other objects, not only on the floor.

4.3.2 SVBSP Tree

As for the tiling, the algorithm we used for shadow generation is not the same as the original SVBSP algorithm. In the original method (*standard SVBSP*) the

tree is built by inserting the polygons in increasing distance from the source, while in the method we used here (*unsorted SVBSP*) the polygons can be added in any order. In the experiments the order in which the polygons are inserted is determined by the scene hierarchy.

Table 4.3 shows the timings and number of shadow polygons created when generating the shadows in the four test scenes. *S-SVBSP* refers to the standard SVBSP method, the time to build excludes the time to create the scene BSP tree.

scene	S-SVBSP		U-SVBSP after BSP		U-SVBSP no BSP	
	time (s)	shadow pol	time (s)	shadow pol	time (s)	shadow pol
office1	.46	332	.45	237	.28	172
office2	.72	526	.74	384	.51	292
office3	1.25	892	1.16	677	.63	389
office4	4.65	3820	3.70	2458	1.51	1063

Table 4.3: Timings for initial building of the SVBSP trees

The unordered SVBSP tree is created in two alternative ways, using the initial set of polygons (column marked *U-SVBSP no BSP*) and using the polygons after they have been split by the scene BSP tree (column marked *U-SVBSP after BSP*). In both cases however the order of insertion is determined by the scene hierarchy. The different ordering is partly responsible for the difference in timing between the S-SVBSP and the U-SVBSP after BSP. Timings include the calculation of the shadow geometry. The results suggest that even when (unnecessarily) using the polygons from the scene BSP tree, the unordered tree takes no more time to build, and results in less shadow polygons than the method described in [17].

Table 4.4 gives timings for transformations of various objects. In each case the number of polygons along with the proportion of the total scene accounted for by the object being transformed is shown. The timings for transformations differentiate between the first move and subsequent moves. The subsequent transformations always take less time, for the reasons given in Section 4. The column marked *compared to S-SVBSP* gives the proportion of time taken for the transformation in comparison with recreating the complete standard SVBSP tree and the column marked *compared to U-SVBSP* the proportion of time against recreating the complete unordered SVBSP tree.

Two sets of experiments were performed for each scene. In the first set (first row for each scene), a BSP tree was built representing the scene and the resulting

scene	object moved	object polygons number	% scene	time to move (s) first	next	%U-SVBSP first	next	%S-SVBSP first	next
office1	computer	27	16	0.08	0.06			17	12
		20	15	0.03	0.07	12	9	7	5
office2	computer	27	19	0.12	0.06			16	8
		20	15	0.07	0.03	14	6	10	4
	bookcase	54	21	0.25	0.14			34	19
		54	26	0.19	0.9	37	18	26	13
office3	computer	51	9	0.21	0.07			17	5
		20	6	0.05	0.03	8	5	4	2
office4	computer	27	2	0.25	0.14			5	3
		20	3	0.15	0.07	10	5	3	2
	desk 1 & comp	79	4	2.20	0.82			47	18
		56	8	0.43	0.22	28	15	9	5
	desk 2 & comp	115	6	0.83	0.43			18	9
		56	8	0.23	0.12	15	8	5	3
	bookcase	66	4	0.70	0.18			15	4
		54	7	0.17	0.09	11	6	4	2

Table 4.4: Transformation timings for the SVBSP method

polygons were used as input for building both SVBSP trees. This was done to obtain a measure of the performance when the input polygons for the SVBSP trees are the same. In the second set of experiments (second row), the unordered SVBSP tree was built from the scene polygons, which is why the same object has less polygons and it takes less time to move. The standard SVBSP tree used for comparison is the same throughout.

4.3.3 Comparison of the Tiling and the SVBSP Algorithms

When generating the initial set of shadows for the two methods, we observed that given the same input polygons, they perform equally well in terms of speed. In terms of output shadow polygons, the tiling method produces a much smaller set. This is in accordance with the observation in [92].

The advantage of the unordered SVBSP is that it allows us to use the initial scene of polygons as input, instead of the polygons after having built the scene BSP tree. Because of this, the time taken by the unordered SVBSP algorithm is about one third of the time taken by the tiling (and the standard SVBSP)

algorithm. Another advantage is the low number of output shadow polygons in the unordered SVBSP method.

The reason for using the BSP tree for the tiling was mainly because the ordering allowed for marking of interior files which resulted in fewer shadow polygons and greater speed. For interaction we do not use the marking, so a better performance could probably be obtained by using an "unordered tiling". Even though we have not implemented this, we believe that times comparable to the unordered SVBSP algorithm could be obtained with definitely fewer shadow polygons.

For interaction, comparing the two algorithms directly from the tables given here is unfavorable for the tiling method. This is due to the fact that the code for the SVBSP was highly optimized (at a low level) for interaction, while the tiling was not. However we can still make some observations. For both, the transformation time depends on the size (number of polygons) of the object moved and on the number of shadows produced. For the SVBSP algorithm the time for the first transformation in a sequence also depends on the position of the object in the tree. This may cause considerable delay when certain objects are first selected. So this method is more suitable for scenes where a selected object is likely to move for more one than one frame in a sequence and/or when the objects likely to move are known in advance and they are added to the tree at the end. For the tiling the interaction times are the same for all steps in a sequence which makes it more suitable for environments where the selected object changes frequently.

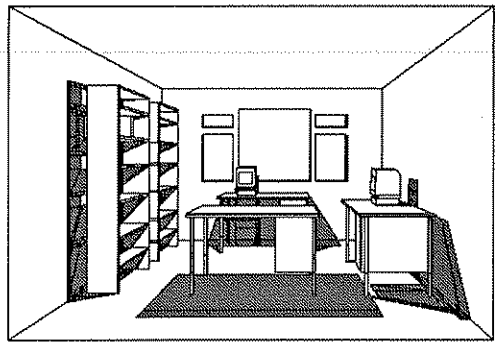


Figure 4.8: Office2

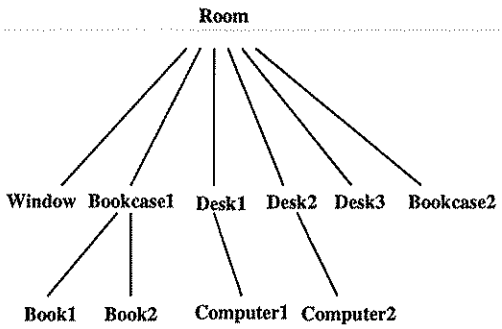


Figure 4.9: Model hierarchy

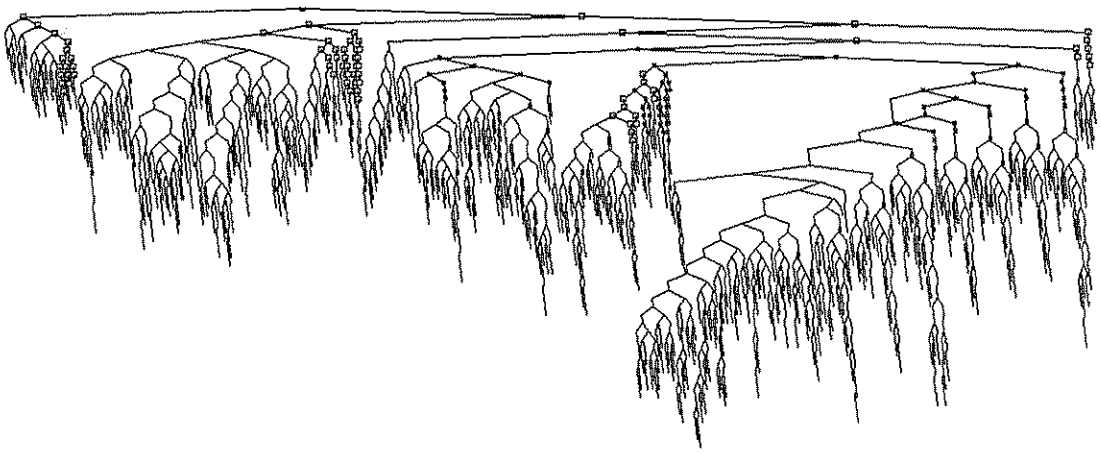


Figure 4.10: Position of computers in standard SVBSP

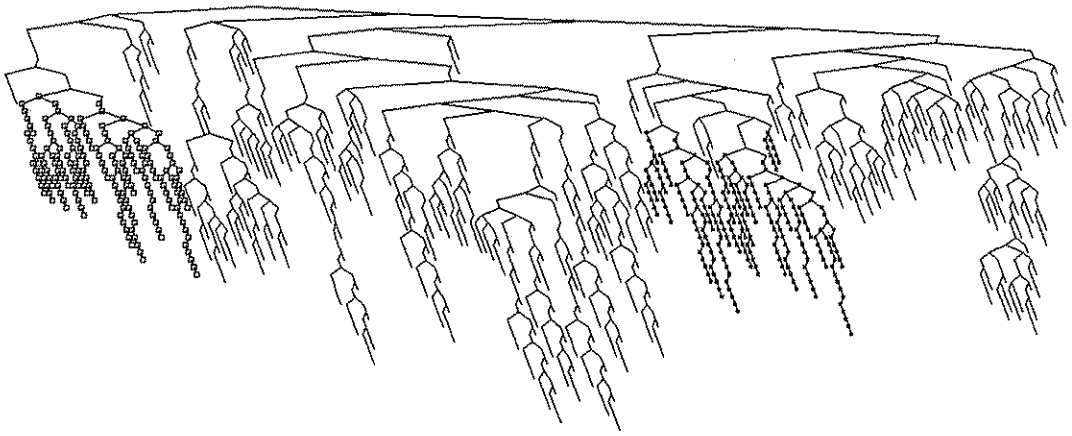


Figure 4.11: Position of computers in unordered SVBSP

The faces of computer1 in the tree are marked by a □ and those of computer2 by *

Chapter 5

Dynamic Scenes Illuminated by Area Light Sources

In the previous chapter we have described two methods for generating shadows from point light sources in dynamic polygonal scenes. Although the shadows produced have provided much information about the relationships between objects in the scene, and made the task of interaction much easier, they are too simplistic and unrealistic for many applications. Instead, a better representation of real lighting can be achieved using area light sources. However these require more complex algorithms if they are to represent the illumination function closely: the boundaries of the umbras and penumbras must be located, and any other abrupt changes (discontinuities) in the gradient of the illumination function must be identified.

Previous research, in the context of discontinuity meshing radiosity, has provided a number of ways for identifying the illumination discontinuities but only for static scenes. In this chapter we will describe a new method for constructing a discontinuity mesh that allows for fast updates after a modification in the scene geometry. Since we assume that speed is critical for the applications concerned, we will not consider any EEE or non-emitter EV events. Using the same reasoning we will avoid any further subdivision other than that necessary. The method can be extended to the use of EEE and non-emitter EV-events and adaptive subdivision if it is be to used for static scenes.

5.1 The Need for a New Algorithm

All existing methods for discontinuity meshing share a common fundamental problem that makes them unusable for interaction. They trace each discontinuity surface in the scene separately, so even though they find all critical edges they cannot find the areas covered in shadow. If they were to be used for interaction, these methods, would have no way of knowing which vertices or edges have a modified visibility with respect to the source, when a polygon is added or removed from the scene. An example of a case where existing algorithms would fail is shown in Figure 5.19. To determine which vertices are covered by the newly added polygon (Figure 5.19(b)) an exhaustive search would have to be performed on the entire set of existing vertices.

In the method described in this chapter we deal with this by taking a step backwards and treating the discontinuity meshing problem as a shadow problem.

The boundary of an area light source shadow is defined by the penumbra. To find the discontinuities we trace the penumbra volumes of each occluder in the scene and when an intersection is found we cast the whole set of discontinuities as one, instead of tracing each one separately. Shadow volumes for area light sources have been suggested before [13, 18] but the difference is that we do not only find the umbra and penumbra boundaries, as in those methods, but the complete set of EV edges.

Another problem with existing methods is their speed. Usually the construction time of the DM for a simple scene is of the order of tens or hundreds of seconds, on very powerful machines. Even if the mesh could be modified in a fraction of this time, still it would not give real-time rates. Several acceleration techniques are suggested here.

We speed up the process of finding the shadow relations between the polygons in the scene by using an efficient space subdivision scheme based on the tiling cube and by ordering the polygons using the method described in *Section 3.2*. To avoid the increase in the number of input polygons we do not split these when building the BSP tree and show how to use the tiling cube with such a tree.

Once a shadow relation is identified, the discontinuity edges from the occluder to the receiver are found and built into a *single DM-tree*. To accelerate the insertion of these edges into the receiver's DM-tree we use BSP merging. Note that at this stage any D^0 edges due to a touching occluder and receiver are also found so no extra pass is required to locate them, as in the other methods.

The most important benefit of the merging is apparent at the illumination step. At the construction of the single DM-tree each of its cells is assigned the identifier of the occluder. As the single DM-tree is merged into the total DM-tree of a receiver this information is passed to the cells of the total tree. During illumination we use this information to reduce the amount of occluder/source clipping needed to the minimum.

In addition to the speed advantage our method is also more accurate since the discontinuities from an occluder are connected by the adjacency information of the EV surfaces rather than by relying only on machine precision (see *page 53* for related discussion).

5.2 Overview of the Algorithm

The description of the method is divided into two parts, the initial building and illumination of the mesh and the incremental modification step.

```
void buildDM()
{
/* construct the tiling cube and the scene BSP */
        bsp = buildOrderedBSP(light, polygons[]);
        tc = constructTilingCube(scene bounding box, light);

/* project polygons in order on cube and find the discontinuities */
        order[] = traverseBSP(bsp);
        for each polygon p_i in order[] do
            if not already processed p_i
                /* build the penumbra, umbra and internal shadow volumes */
                {PSV_pi, USV_pi, ISV_pi} = constructShadowVolumes(p_i, light);
                /* project on the tiling cube using the penumbra */
                apl_pi[] = projectOnCube(tc, PSV_pi);
                /* find the shadows between p_i and polygons in its apl_pi[] */
                for each polygon p_j in apl_pi[] do
                    castShadow(p_j, p_i);
                    if last orderno of p_j ≥ i
                        castShadow(p_i, p_j);
                    endif
                endfor
            endif
        endfor

/* illuminate mesh vertices */
        for each vertex v_i in the mesh do
            illuminateVertex(v_i);
        endfor
}
```

Figure 5.1: Initial building of the discontinuity meshing

The algorithm for constructing the discontinuity mesh is summarised in Figure 5.1. The basic structure is very similar to the Shadow Tiling for point light sources (*Section 4.1*). First the scene BSP tree and the tiling cube are built. Then, the

polygons are projected into the cube in the front-to-back order as seen from the light source and the active polygon list (APL) is found. Shadows are then cast between the current polygon and the polygons in its APL. Finally, the vertices of the mesh are illuminated.

Of course most of the operations performed at each of the above steps are very different from the point source tiling: the BSP tree is specially built to give an invariant ordering for any point on the light source (*Section 3.2*); the tiling is not a small cube placed around the source but more like a bounding box placed around the scene and the polygon projections are found by clipping the cube sides against the polygons penumbra volume and the cube sides (*Section 5.3.1*); and the shadows cast from occluder to receiver are complete sets of discontinuities pre-built into a DM-tree and then merged into the receiver's total DM-tree rather than detail polygons stored on-top (*Section 5.3.2*).

```
void transformObject(obj)
{
/* remove the object */
        /* remove its polygons from the tiling and the scene BSP tree*/
        for each polygon p_i in obj do
                mark p_i on the scene bsp;
                remove p_i from the tiling;
                add the id of p_is receivers to invalidDMT[];
        endfor
        /* remove the shadows of the object from the relevant DM-trees */
        for each polygon in invalidDMT[] do
                remove discontinuities due to obj;
                add any disturbed vertices to toIlluminate[];
        endfor

/* update the BSP tree */
        bsp = restore(bsp);
        getNewObjectGeometry(obj);
        bsp = addToOrderedBSP(bsp, obj);
        order[] = traverseBSP(bsp);

/* add the object back */
        /* Build shadow volumes, add to the tiling and cast shadows */
        for each polygon p_i in obj do
                {PSV_pi, USV_pi, ISV_pi} = constructShadowVolumes(p_i, light);
                apl_pi[] = projectOnCube(TC,PSV_pi);
                for each polygon p_j in apl_pi[] do
                        if first orderno of p_j ≤ last orderno of p_i
                                castShadow(p_j, p_i);
                        endif
                        if last orderno of p_j ≥ first orderno of p_i
                                castShadow(p_i, p_j);
                        endif
                endfor
        endfor

/* illuminate any new or disturbed vertices */
        for each vertex v_i in toIlluminate[] do
                illuminateVertex(v_i);
        endfor
}
```

Figure 5.2: Modifying the discontinuity meshing

The information produced from this method is in object space and can be used for rendering the scene from any viewpoint. Thus all that is required is a way to modify the scene objects for a fully dynamic environment. As before modifications are modeled by a deletion and/or an addition of objects.

The dynamic process is described in Figure 5.2. When an object is transformed, its polygons are deleted from the tiling cube and the BSP tree similarly to the algorithm in *Section 4.1.* Then the DM-trees of all relevant polygons are traversed for removing all shadow information due to this object. This information includes mesh edges forming nodes in the 2-D BSP. These are removed using the method described in *Chapter 3.*

Once the object is transformed to its new state, it is added back to the scene BSP tree which is traversed to get the new front-to-back order. Each of its polygons is projected into the tiling cube to find its APL. The precedence of polygons in respect to their visibility from the source can be determined by their first and last order numbers. For any polygon further away we need to find the discontinuities on it from the added polygon, if any, and for any polygon that lies closer the reverse.

Finally any newly created vertices and any existing vertices that were uncovered when the object was deleted or covered when it was added back will have their illumination value calculated.

In the rest of this Chapter we present and evaluate the algorithm for dynamic scenes. The next section gives a more detail description of the main steps for the building of the discontinuity meshing followed by the section on modifying the mesh in dynamic scenes and closing with some examples and results.

5.3 Constructing the Mesh

As a first step the scene BSP tree is built, using the algorithm presented in *Section 3.2*, so as to give an order valid from any point on the light source. One of the main problems in using BSP trees is the increase in the number of polygons. This is especially true in applications, such as the present, where expensive operations depend heavily on the number of polygons. To reduce the problem but retain the benefits of the ordering given by the BSP tree, we do not split the polygons during the construction of the tree. Whenever a polygon is found straddling the plane of a root node two copies, pointers to the original polygon are inserted down the

subtrees, one in each side. The resulting tree will have exactly the same structure as if had been built normally only that at nodes where a fragment would be held now we have a pointer to the original. A tree built like this can not be used for visible surface determination but, as we describe in the next section, it can be used for determining the priority in the tiling.

Since the polygons are processed as a whole, the first time a polygon is encountered all its critical surfaces are created. For convenience we group the critical surfaces of a polygon into three sets the penumbra (PSV), umbra (USV) and internal (ISV) shadow volumes. As their names suggest, the first two are collections of the extremal surfaces, penumbra and umbra. The third set (ISV) does not define a volume of space but is simply the rest of the critical surfaces which fall between the umbra and the penumbra.

The criterion for classifying the surfaces into the aforementioned sets is the same one proposed by Nishita and Nakamae [76]: penumbra surfaces are those having the source fully in their front half-space and the occluder fully in the back half-space; while umbra surfaces are those having both the source and the occluder fully in their back half-space. The rest (internal) intersect either the source or the occluder.

A special case occurs when a polygon, lying in the front half-space of the source, cuts the source with its plane. Here we use as source the part that falls in the front half-space of the polygon and there is no USV.

5.3.1 Determining Shadow Relations Between Polygons

In most algorithms where tiling cubes (or hemi-cubes as here) are employed, they are usually placed closely around the "source" and the polygon intersections with the cube are found by a projection through the centre of the source (Figure 5.3). This is sufficient for applications where the source is a point ([50, 92], *Section 4.1*) or where an approximation by a point is acceptable ([24]). In our case we cannot accept such an approximation as it would under-estimate the relations between some polygons and cause the omission of shadows. See for example Figure 5.4 where even though P_2 intersects the penumbra of P_1 their projections on the cube do not intersect.

A method that identifies all shadow relations is the *shaft volume* (Figure 5.5), where the projection is taken as the intersection of the cube with the shaft of the source and a scene polygon. For this volume we can either use the axis aligned

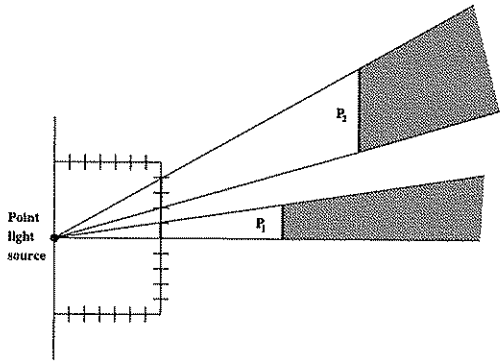


Figure 5.3: Tiling cube for point light source gives a minimal super-set of the shadow relations

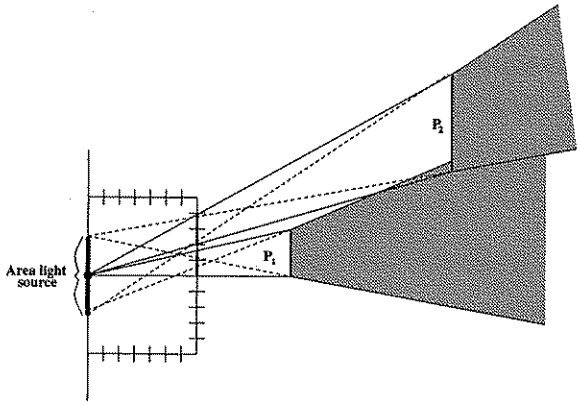


Figure 5.4: Projection by point approximation underestimates the shadow relations for area light sources

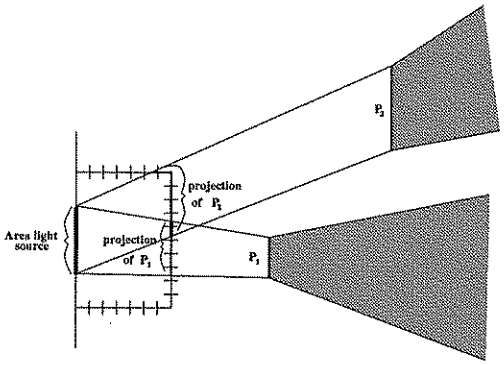


Figure 5.5: Using the *shaft volume* greatly overestimates the shadow relations

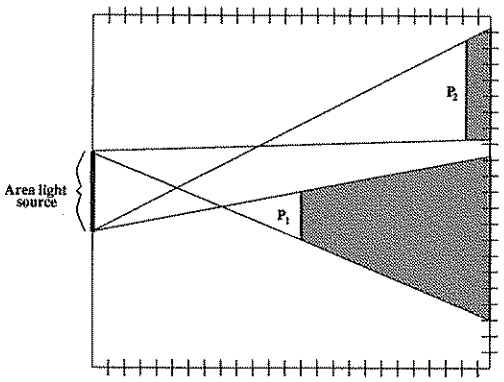


Figure 5.6: Placing the tiling cube around the scene gives a minimal super-set of the shadow relations

bounding boxes of the two polygons [49] or their convex hull [13]. However, apart from requiring extra processing for finding the planes that form the volume, it also greatly over-estimates the number of possible interactions.

We found that a much smaller super-set, which at the same requires no additional calculations, can be found by letting the tiling cube enclose the whole scene and using the intersection of the cube faces with the penumbra as projection (Figure 5.6). Assuming that the source is relatively small compared to the distances involved, this estimation can be very close to the set of actual relations. The closer the sides of the tiling cube are to the polygons, the smaller the set of excess classifications. An example can be seen in Figures 5.6 and 5.7, making the tiling cube larger causes the projections of P_1 and P_2 to overlap.

In our implementation the size of the cube is determined by two factors: the

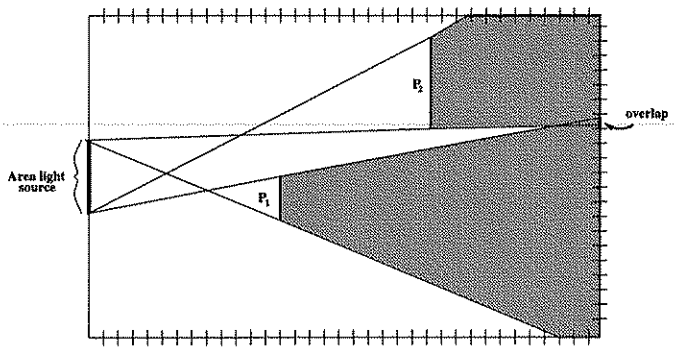


Figure 5.7: Larger cube gives larger overestimation

bounding box of the scene, including the volume where objects may possibly move, and the sphere of influence of the light source. We take the minimum of the two in each direction.

Once the cube is built, the polygons are projected onto it. As mentioned earlier, the polygons are not split during construction of the BSP tree and thus when the tree is traversed to get the front-to-back order from the source some will inevitably hold more than one position in the ordering. While the BSP is traversed the smallest and largest order numbers of each polygon are stored. The smaller the order number the closer it is to the light source.

As each polygon P_i is processed, the first time it is encountered it is put into the tiling and its APL is found, then it is marked as "processed" and next time it is found is not processed again. Any polygon P_j in the APL of P_i may be, at-least partly, closer to the light source, which is why it is already there. So there is a possible shadow relation from P_j to P_i. But since P_j may have many positions in the tree, one part of it may be behind P_i. This is checked by comparing the last order number of P_j against the first of P_i.

Before actually casting any shadows we apply a first verification test, by testing if the occluder and receiver have a potential visibility obstruction (see *Theorem 1, page 79*).

To give an example of how this works we use Figure 5.8. This in fact is a worst case scenario for the simple scene of Figure 5.8(a). The BSP tree representation at the top of Figure 5.8(b) is very inefficiently built, but it serves well the purpose of showing the steps involved in the method. Notice that the polygons in the tree are not split. When this tree is traversed to get the order from the source some polygons occupy more than one entry in that order. The order is shown below the tree. When processing the polygons in this order we do the following operations:

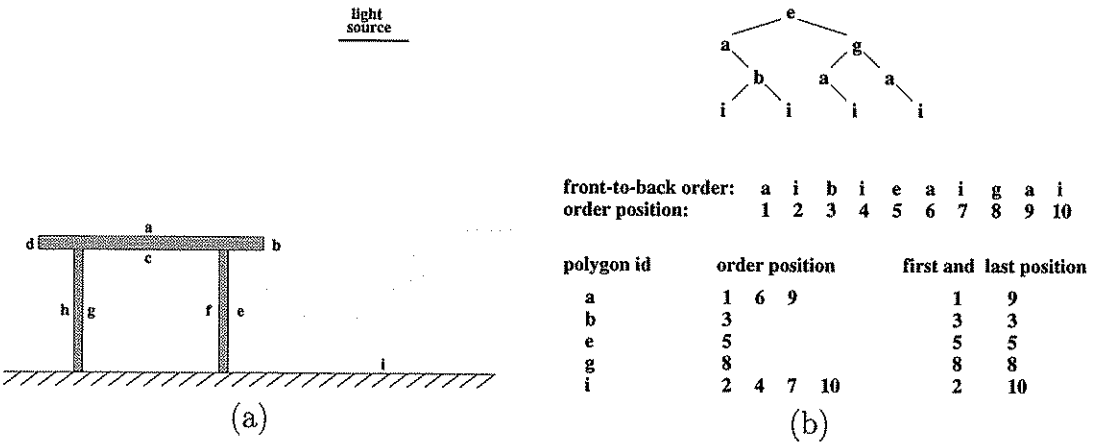


Figure 5.8: (a) A simple scene with a light source and (b) the BSP representation of the scene (top), the order derived by traversing the BSP from the source (middle) and a table of the order numbers of each polygon (bottom)

Note that here $\{P_i \rightarrow P_j\}$ means that there is a shadow relation from P_i to P_j and *cancelled* refers to the relation failing the verification test just mentioned.

step = 1	**a** is added	APL = $\emptyset$		
step = 2	**i** is added	APL = {**a**}		**a** → **i**
			last(**a**) > first(**i**)	**i** → **a** (*cancelled*)
step = 3	**b** is added	APL = {**a**, **i**}		**a** → **b** (*cancelled*)
			last(**a**) > first(**b**)	**b** → **a** (*cancelled*)
				i → **b** (*cancelled*)
			last(**i**) > first(**b**)	**b** → **i**
step = 4	**i** has already been projected			
step = 5	**e** is added	APL = {**a**, **i**, **b**}		**a** → **e**
			last(**a**) > first(**e**)	**e** → **a** (*cancelled*)
				i → **e** (*cancelled*)
			last(**i**) > first(**e**)	**e** → **i**
				b → **e**
step = 6	**a** has already been projected			
step = 7	**i** has already been projected			
step = 8	**g** is added	APL = {**a**, **i**}		**a** → **g**
			last(**a**) > first(**g**)	**g** → **a** (*cancelled*)
				i → **g** (*cancelled*)
			last(**i**) > first(**g**)	**g** → **a**
step = 9	**a** has already been projected			
step = 10	**i** has already been projected			

By the end of this process only seven pairs of (occluder, receiver) are found that may have a shadow relation and survive to the more expensive operations. Note that the above example can be made more efficient if the scene BSP tree is more carefully built, for example if the two large polygons (floor and top of table) are added to the tree earlier.

Optimisation

The same optimisation used in *Chapter 4* for speeding up the scan-conversion of polygon projections in the tiling cube can be used here as well: we can avoid comparing all 5 sides of the hemi-cube against the penumbra planes by first projecting one of the penumbra vertices onto the cube to find which side it falls on. During scan-conversion of the projection on this side, if any boundary edge is crossed then we continue with the cube-side over that edge.

5.3.2 Casting a Shadow Between two Polygons

The potential shadow relations computed in the previous section are used for casting the shadows between the polygons. In this section we will describe how, given two polygons (an occluder O and a receiver R), we find the discontinuities and the regions on the receiver, covered by the umbra or penumbra of the occluder. This is summarised in Figure 5.9.

```
void castShadow(Polygon occluder,Polygon receiver)
/* cast a shadow from the occluder to the receiver*/
{
        /* project the penumbra vertices onto the plane of the receiver */
        pv[] = projectPenumbraVertices(occluder, planeOf(receiver));
        /*compare the receiver polygon against the penumbra vertices */
        compareAgainstPenumbra(receiver, pv[]);
        if there is no intersection return ;
        endif

        singletree = constructSingleTree(occluder, pv[], receiver);
        receiver.dmt = merge(receiver.dmt, singletree);
}
```

Figure 5.9: Casting a shadow from one polygon to another

First we apply an additional verification test on the shadow relation: the penumbra vertices are projected on R's plane and the area they define is compared against R, Figure 5.11. Since all critical surfaces from an occluder are enclosed by the penumbra, if the receiver has no intersection with the penumbra then it

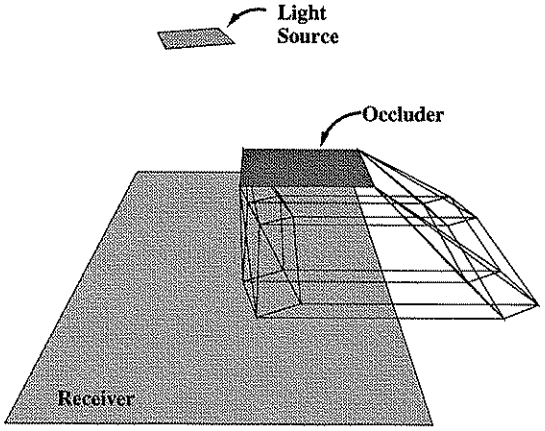


Figure 5.10: The source, the receiver and the occluder with the complete set of EV planes.

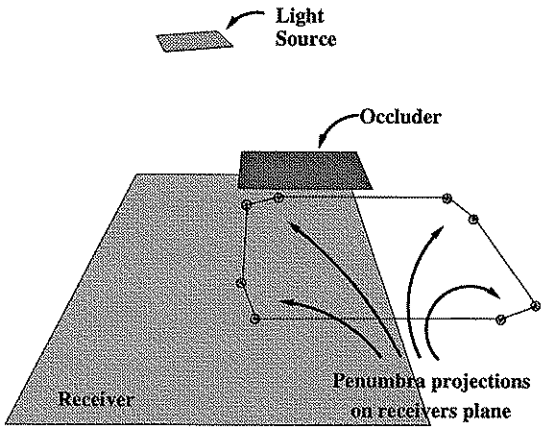


Figure 5.11: The penumbra vertices are cast on the receivers plane and checked for intersection with the receiver.

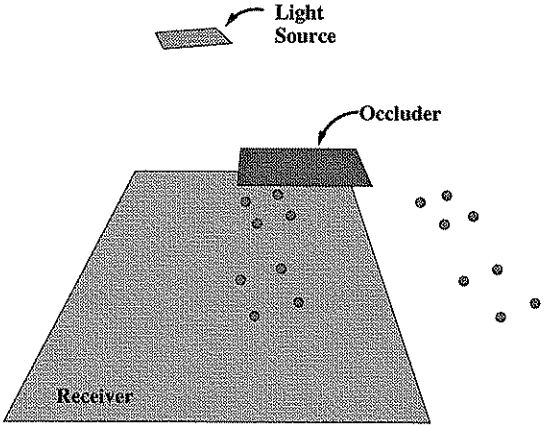


Figure 5.12: When an intersection is established the rest of the vertices are also projected.

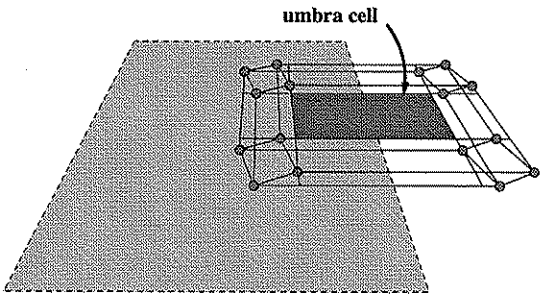


Figure 5.13: The single-tree is built using the adjacency information in the shadow planes.

cannot have an intersection with any of the other surfaces. The function execution terminates here if no intersection is found, and we proceed to the next (R, O) pair. If there is some intersection then the rest of the vertices (umbra and internal) are projected onto R's plane, Figure 5.12, and they are joined to make a DM-tree of discontinuities from O, Figure 5.13. We call this tree the *single DM-tree* of O on R (or simply *single-tree*). This single DM-tree is then merged into the total DM-tree of R, Figure 5.14.

Constructing the Single DM-Tree

After studying the structure of the set of discontinuities on a receiver for a variety of occluder/source pairs (Figure 5.15) and for different orientations, we made

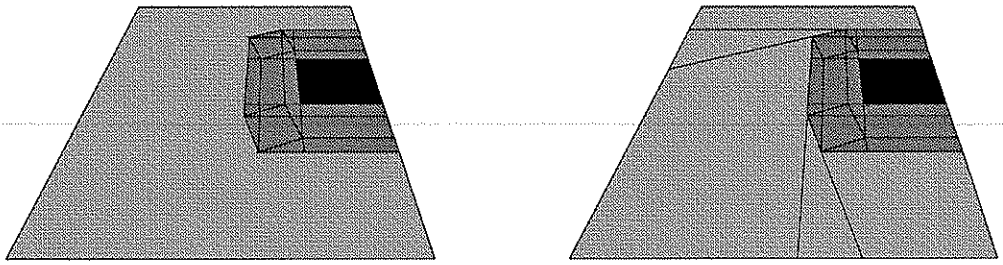


Figure 5.14: The single-tree is merged into the total-tree of the face, clipping anything outside and adding construction edges to any penumbra edge not spanning its subspace.

certain observations which we thought could help to build the single tree more effectively. The four edges meeting at each of the vertices, two *ev* (source edge, occluder vertex) and two *ve* (source vertex, occluder edge), will always connect in a certain order depending only on the relative geometries of the occluder and the source, not the receiver. Also the overall structure of the single tree for a given pair (occluder, source) will be qualitatively equivalent on most receivers. Certain simple tests can be applied to indicate whether this structure holds for a given receiver or not. One such test could be to project the umbra vertices on the receivers plane and check if the umbra cell is defined and it has n_o edges or if the non-consecutive umbra edges cross, where n_o is the number of occluder edges.

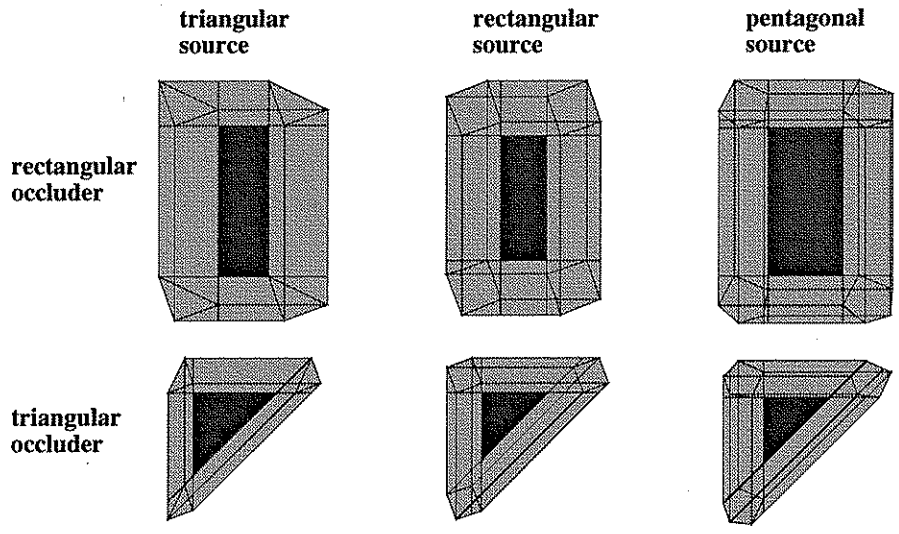


Figure 5.15: Shadows and discontinuities from pairs of different geometries.

Initially it was anticipated that a parameterised DM-tree could be built for each occluder. This would have both the 2-D BSP tree and the WEDS structure pre-computed, including the intersection of the edges, the only variable being here the exact co-ordinates of the mesh vertices which would depend on the plane equation of the receiver. For the majority of shadow relations the criteria men-

tioned above are met and so the single-tree could be found by just plugging the receiver's plane equation into the parameterised tree. For receivers not meeting the criteria a more general algorithm is needed.

Given such a parameterised tree, when building a mesh for a static scene, we can clip it as we go along by keeping only the part falling in the *OUT* cells when merging it with the receivers total DM-tree.

Such a tree was partly implemented with some success for the general case of rectangular occluder and source, but it was later abandoned as it required very complex and specialised functions for each different pair of (occluder, source) geometries and because of the large number of special cases.

The current implementation employs a more generalised algorithm that uses only some of the available information: first the umbra and internal vertices are projected onto the plane of the receiver; recall that the penumbra vertices have been projected already in the previous step. In general there will be $n_o \cdot n_s$ vertices, including the penumbra, where n_s is the number of source vertices. Each of these vertices has its own id-number indicating the vertices (one from the occluder and one from the source) that caused it. The penumbra subtree is constructed by traversing the penumbra vertices and connecting them. We know that none of these edges intersect so they form a linear tree. The umbra subtree is then built using the umbra edges but some comparisons are needed here as there may be intersections between them. This subtree is attached at the back of the penumbra. At this point the umbra cell is identified, if it exists, and is marked.

The edges due to surfaces in the ISV are added one by one starting at the first umbra node since they are definitely behind all penumbra nodes. As they are filtered down the leaves of the tree, if a vertex with the same id-number as one at their end-points is met then they make a connection with it. This ensures that each vertex connects to the correct four edges without depending on machine precision and with minimal computation.

Some notable special cases of this method are:

D^0 edges: If the receiver and occluder are touching then some of the umbra vertices will coincide with penumbra vertices. In such case these (D^0) vertices are marked as both umbra and penumbra and any edge defined by two such vertices is marked as D^0 edge.

Undefined vertices: When the receiver cuts the occluder with its plane then not all of the $n_o \cdot n_s$ vertices will project correctly. Dummy vertices are used

to replace those undefined which are clipped away later during the merging with the total DM-tree of the receiver.

Merging the Single DM-Tree into the DM-Tree of the Receiver

After constructing the single-tree we merge it with the total DM-tree of the receiver. We use the algorithm for BSP tree merging proposed by Naylor [73] with some modifications to allow for trees not spanning the entire subspace in which they reside. Nodes with such a property are the boundary edges of the receiver and also the penumbra nodes of the single-tree. As seen in Figure 5.14 the latter are only expanded after they reach a cell.

The merging algorithm is recursive and terminates only when one of the trees involved reduces to a cell. The function *treeOpCell* described in *Section 2.1.4* is called to apply the union operation on the tree and cell with a result depending on the value of the cell.

In Table 2.1 the cell can have only two values, IN or OUT, indicating the containment of the cell in a polyhedron. However here, because the trees we are merging are not defined over the whole of 2-D space but rather over the limited subspace enclosed by the boundary of the receiver, we have an additional value OUT^*. This value shows that the cell is outside of the space of interest and is assigned to those cells lying on the outside of polygons boundary edges. The other two values are still used and they refer to the containment of a cell in shadow. In addition each cell carries extra information showing the list of polygons limiting its view from the light (occluders).

When merging polyhedra, if a cell is in both then it is assigned an IN value. This is shown in the first line of the Table 2.1 where the tree added to an IN cell is compressed. Here this reasoning is only valid for the umbra cells. For the penumbra we need to keep the subdivision because it matters if a cell is in one shadow or more.

Actually in our implementation we keep the subdivision even in the umbra regions since during interaction the occluder of the umbra cell may be removed. So the *treeOpCell* function has the three cases shown in Table 5.1.

The values a cell can take and the result when adding a tree to the cell are:

- OUT^*, only for cells lying on the outside of boundary edges
  $OUT^* + tree = OUT^*$.

op	Cell	Tree	Cell <op> Tree
union	OUT*	t	OUT*
	OUT	t	t
	IN	t	t′

Table 5.1: Combining a cell and a tree

- *OUT*, for unoccluded cells
  $OUT + tree = tree.$

- *IN*, for occluded cell, along with this is stored a list with the occluding polygons, umbra ones first.
  $IN + tree = tree'$, where *tree'* has the same structure as *tree* but each of its cells has the list of polygons in the cell added to its own list.

Optimisations

Since we are building this mesh for interaction and not as a static mesh, we avoid clipping the critical surfaces as they are processed. However, we can still apply certain optimisations without affecting the possibility for modifications. Here are some of the optimisations we have implemented:

- Any polygon that has its penumbra completely blocked by a receiver R does not cast shadows on any polygons further away than R. However, it needs to record these potential relations in case R is moved.

- Any polygon that is completely covered by the umbra of an occluder above it, casts no shadows. If the occluder of this polygon is removed then we treat it as new and process it again.

- Using the assumption that the polygons are grouped into objects and that they will only be transformed as part of them, we can condense the subdivision in umbra cells that is caused by critical edges from polygons of the same object. In fact since most often neighboring polygons are part of the same object this results to a compression of almost all umbra regions. This can be seen in Figure 5.16. Here we see the shadows of a desk on the floor before and after umbra compression.

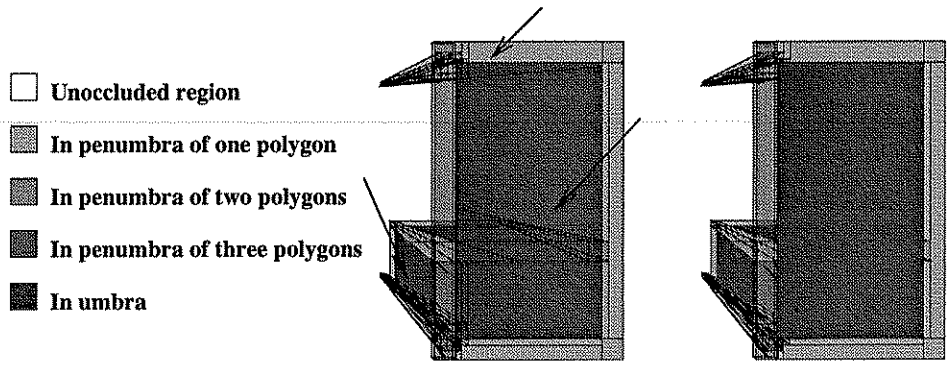


Figure 5.16: Discontinuities in the umbra of faces from the same object can be compressed.

5.3.3 Computing Illumination Intensities on the Vertices

An illumination intensity must be calculated for each vertex in the mesh. For this we use *Equation 2.1*, described in *Chapter 2*. This equation assumes that the source is totally visible from the vertex in consideration. For any vertex v_o which may be partly blocked, the visible parts of the light source must be found.

For the illumination we need to know, for each vertex, if it is occluded or not, and if it is, which are the occluders. In existing DM algorithms there is no means of knowing the state of each vertex (if it is unoccluded, in umbra or in penumbra) so the visibility of the source from each must be determined. This is done by projecting the potentially occluding polygons onto the source plane and clipping away any part of the source that is hidden. The visibility determination is usually the most time consuming operation of the whole algorithm and is also very wasteful since many of the mesh vertices are either completely unobstructed or in umbra (see Table 5.3).

In our method, however, not only do we know the state of each vertex but we also know exactly which occluders block each penumbra vertex before we begin to illuminate it, so there is no searching and no redundant occluder/source comparisons.

An overview of the illumination step is given Figure 5.17. As a result of using a Winged Edge Data Structure, each vertex v_i holds a pointer to one of the edges of which it forms the end-point. From this edge the set of mesh cells C sharing v_i can be found. Each of these cells holds an *occluder-list* (O_{C_j}) which is a list of the faces that block the light source from its view, either partly or fully. The occluders that block the source fully, are stored (and flagged) at the head of the occluder-list.

```
void illuminateVertex(Vertex v_i, Light light)
{
        C = set of n mesh cells sharing v_i
        if ∃ c_j ∈ C such that c_j is unoccluded
                illuminateUnoccluded(v_i, light);
        else if ∃ c_j ∈ C such that c_j is in umbra and
                        the edge of c_j through v_i is not D^0
                illuminateInUmbra(v_i);
        else
                O = set of n occluder lists of the cells in C
                O_covering = ⋂_{j=1}^{n} O_{C_j}, where O_{C_j} ∈ O
                if O_covering = ∅
                        illuminateUnoccluded(v_i, light);
                else
                        illuminatePenumbra(v_i, O_covering, light);
                endif
        endif
}

void illuminateUnoccluded(Vertex v_i, Light light)
{
        v_i.intensity += result of Equation 2.1 using v_i and light;
}
void illuminateInUmbra(Vertex v_i)
{
        v_i.intensity = ambient;
}
void illuminatePenumbra(Vertex v_i, Polygon O_covering, Light light)
{
        lightregions[] = projects the polygons in O_covering onto the
                light's plane to find the regions of light visible from v_i;
        for each region r_i in lightregions[] do
                illuminateUnoccluded(v_i, r_i);
        endfor
}
```

Figure 5.17: Illumination of a vertex in the mesh

Using these occluder lists we determine the visibility of the source for v_i. We have three cases to consider:

1. Any of the O_{C_i} are empty: The vertex is illuminated as unobstructed. This can happen in three cases: (i) when the vertex is shared only by unoccluded cells, vertex v_a in Figure 5.18, (ii) when the vertex is shared by lit and penumbra cells, vertex v_b in Figure 5.18 or (iii) when shared by umbra, lit and maybe penumbra cells v_c in Figure 5.18. To avoid light leaks, because of the later case, umbra cells are always displayed with ambient light regardless of the vertex colour value.

2. One of the O_{C_j}s contains an umbra element: The vertex is given an ambient colour value (vertex v_d in Figure 5.18). In the rare case where a D^0 vertex is covered by the penumbra caused by a different face then the vertex is treated as penumbra. These cases can be easily identified by the elements in the occluder lists.

3. All the O_{C_j}s are non-empty and contain no umbra elements: The occluder sets O_{C_j} of all the cells in C are put together using an intersection operation and the active subset $O_{covering}$ of the occluders that cover the vertex, is found. The polygons in set $O_{covering}$ are then used to determine the visible parts of the source, from the vertex, and calculate the intensity using *Equation 2.1* on them. Examples of these cases are vertices v_e and v_f. In v_e all touching cells are covered by the face 1 of the cube so taking the intersection of the cell occluder-lists gives $O_{covering}$ of v_e = face 1. For v_f all cells are covered by face 1 but some by face 2 so again $O_{covering}$ of v_f= face 1.

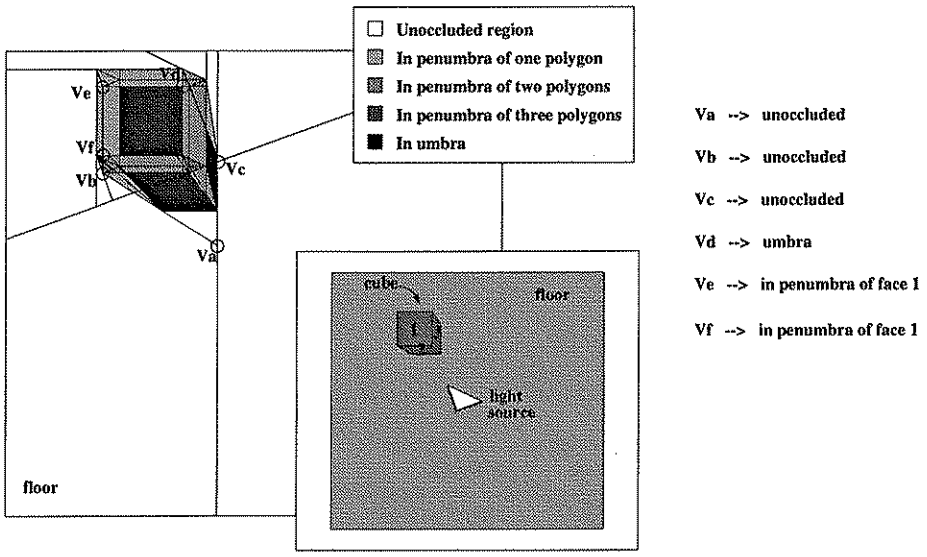


Figure 5.18: Possible classifications of a vertex during illumination.

It is important that the above tests are performed in the given order otherwise shadow leaks may occur. An example of this is vertex v_c in Figure 5.18. If we test for umbra occlusion before checking for no occlusion, then the vertex will be given an ambient value causing the umbra to leak into the lit cell on the right.

5.3.4 Further Subdivision

With the mesh created in the above process we can be certain that the major discontinuities in the illumination function are captured. There are cases, however, where further subdivision is required for the interpolation function to give a sufficient approximation. This could be caused by several factors like badly shaped cells or the presence of a maximum or a point of inflection not covered by the mesh.

In most radiosity solutions this problem is treated by performing a triangulation on the mesh elements and then further adaptively subdividing these until certain termination criteria are met.

Our method is particularly well suited for this kind of refinement since the information held by each cell (the occluder list) can help to provide better termination criteria and speed up the illumination of new vertices. But, since we aim for real-time modifications of the mesh, we can not afford an expensive subdivision method.

More research is required to find an efficient dynamic triangulation that can be updated with minimum re-computation. Meanwhile we can eliminate the most important artifacts by locating the global maximum on each surface.

Given any pair of convex polygons, an emitter and a receiver, the radiance function on the receiver has some well-defined properties. It can have only one maximum and it is monotonically non-increasing at increasing distance from that maximum [29]. Furthermore, the position of the global maximum depends only on the relative geometries of the pair. An intervening polygon (an occluder) can only obstruct and make it disappear it or create less significant local maxima [13]. As the receiver is convex, minima will lie on its vertices, so they will be computed.

The position of the maximum on each source-facing polygon can be located and stored during the mesh construction. At the end, if the neighborhood of the maximum is still unoccluded then we can explicitly subdivide the significant region. During interaction we need to reconstruct this extra triangulation only if the cells in which the maximum lies change.

Finding the maximum analytically, in the general case, is a very difficult task but a good approximation can be found using a method proposed by Drettakis in [29].

5.4 Dynamic Modifications

As the results presented in *Section 5.5* indicate, the algorithm constructs the discontinuity meshing with considerable speed. However, that was not the main purpose of this research. The aim was not just to build another, faster, DM-algorithm but to build one that can take advantage of the spatio-temporal coherence in interactive applications and allow for the necessary modifications to be

performed in a fraction of the normal construction time.

Incremental modifications are made possible due to a combination of certain aspects of the algorithm:

1. The space subdivision scheme significantly localises the operations performed to only a small superset of the affected polygons. Drettakis[30] also uses a (voxel based) space subdivision scheme but as his algorithm traces each discontinuity surface independently, it fails to identify all polygons concerned during scan-conversion (small polygons fully in umbra or penumbra are only found on a separate step).

2. The use of BSP tree merging for adding the discontinuities from an occluder to a receiver polygon. This induces an explicit classification of the cells which provides a means for identifying the concerned vertices during interaction.

   An example of this in a dynamic sequence can be seen in Figure 5.19. In the initial scene we have two objects (object-a and floor) as-well as a light source. The mesh is constructed and the illumination value at each vertex is calculated. At the second frame a third object moves in. The discontinuities due to this are found. A traditional method can find the newly created vertices and pass them for illumination, but as it treats each discontinuity edge independently it will have no fast way of knowing about the three existing vertices that are now covered.

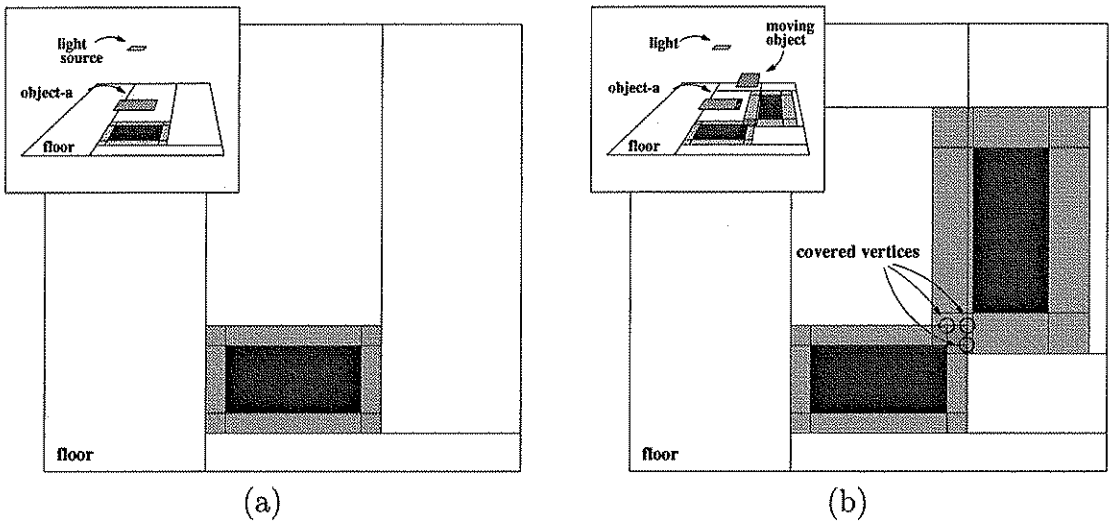


Figure 5.19: Merging allows for easy identification of the vertices with changed intensity when a polygon is added or deleted

As in the previous methods we will perform the object transformation in two steps, a deletion and an addition of the object.

5.4.1 Removing an Object

As described in *Section 5.2*, to transform an object, we first remove it from the tiling cube and the BSP tree as we did for the point light sources. Each polygon holds a list of references to the receiver polygons upon which it has cast a shadow during the construction of the mesh. When removing an object polygon, its receivers are added to a list called *invalidDMT-list*. The polygons added to this list contain information in their DM-trees generated by the moving object which must be removed. So after removing all object polygons the DM-tree of each polygon in the list is traversed and scanned for two things:

1. Nodes holding discontinuities due to polygons in the removed object. These nodes are removed using the method described in *Chapter 3*. As the subdivision defined by these discontinuities is removed, any references to the object in the remaining cells must also be removed.

2. Subtrees marked as completely covered by a polygon of the removed object. The cells of such subtrees are visited and any reference to the object in question is deleted (every vertex of these cells is added to the illumination list).

5.4.2 Adding an Object

Adding an object to the scene requires similar steps to the initial construction of the mesh but only involves the polygons of the added object. First the polygons are added to the scene BSP which is the traversed to get the new front-to-back order. Then they are added to the tiling cube and the other faces sharing tiles with it are found (the APL). Once the shadow relations are decided upon the *castShadow* function is used to generate them.

5.4.3 Illumination of Vertices

After the deletion and/or addition of objects, new vertices will be created and some of the existing ones will have changed visibility, due to objects covering or uncovering them. However most of the vertices will remain unaffected and

it would be extremely wasteful to recalculate the illumination for all of them. Instead a list is maintained during the deletion or addition of objects, which holds the relevant vertices. The vertices added to this list include the following:

During deletion of an object:

1. Existing vertices on cells that have one or more of their occluders removed from their occluder-list.

2. New vertices created by *restoreWeds*, either by extending dangling edges or by partitioning during merging of subtrees after deletion of a node.
   In fact it is not essential to recalculate the illumination value of these vertices from scratch, since the shadow information remains the same. Their value could be determined by interpolation from the end-points of the edge they partition, but we recalculate them for greater accuracy.

During addition of an object:

1. Existing vertices covered by added polygons.

2. All vertices on the mesh of added polygons.

3. Any other new vertex created by the discontinuities caused by the added polygons on the existing.

The illumination of the vertices is done in the same way as described in *Section 5.3.3.*

5.4.4 Optimisations

Several optimisations can be achieved when we consider the fact that, in general in an interactive application the selected object will move for more than one successive frame.

- Removing the object from the scene data involves traversing the DM-trees of the receivers completely from top to bottom. As described in *Chapter 3* this is wasteful since after the first iteration the inserted discontinuities will at or near the leaves. We can take advantage of this by creating a list of pointers to nodes on the DM-trees during the dynamic insertion of the moving object. This list points to the top node of each subtree of

discontinuities due to the moving object and to the top node of each existing subtree covered completely by a shadow of the object.

- One attribute of dynamic environments is that the attention of the user is distracted by the movement so a lot more imperfections can go unnoticed. In cases where the performance of the algorithm is not sufficient such as when the dynamic objects are large or moving over complex parts of the scene, a speed up can be obtained by using only extremal discontinuities for the dynamic objects (umbra and penumbra). We can return back to the full algorithm on release of the object.

5.5 Results

In this chapter we have presented an algorithm for calculating and maintaining the discontinuity meshing in dynamic scenes.

To evaluate the performance of the incremental updates, the same concept is used as in the two previous chapters: we compute the time taken for updating the DM after an object transformation has occured, and compare it against the time it takes to rebuild the whole DM from scratch.

The difference from the previous chapters is that the algorithm for building the DM is not a pre-existing, already tested one. So for our argument to be valid we have to evaluate the building of the mesh and then compare the incremental changes against it.

5.5.1 Statistics for Initial Construction of the DM

scene	scene polygons			construct DM
	total	front facing	offending	(sec)
15 cubes	92	33	20	1.70
officeA	114	42	17	1.23
officeB	128	51	11	1.77
officecubes	184	82	10	2.24

Table 5.2: Total mesh construction time

The algorithm is written in C and implemented on a SUN SparcStation 20, 75MHz, Model 71 with 160M of RAM. Four different scenes were used in the

experiments. The first (*15 cubes*) consists of 15 randomly placed cubes, the second (*officeA*) of a desk, a bookcase, a computer and a large polyhedral cursor and the third (*officeB*) of two desks, one of them raised above the floor, and a bookcase. For the last scene we used three desks a bookcase and six randomly placed cubes. The desks and bookcases in these scenes are the same as in the office scenes in *Appendix C.*

The time for building the mesh (including illumination) for these scenes is given in Table 5.2. Under *scene polygons* we show for each scene the total number of polygons, how many of these are facing the source (*front facing*) and how many cut the source with their plane (*offending*). Times are given in seconds.

scene	mesh vertices				source/occluder comparissons		
	total	lit	umbra	penum	total	av. penum	av. total
15 cubes	3676	676	21	2979	6239	2.01	1.69
officeA	2857	457	716	1684	3830	2.27	1.37
officeB	3074	405	464	2185	6421	2.93	2.08
officecubes	4664	822	674	3168	7829	2.47	1.67

Table 5.3: Illumination of the mesh vertices

In Table 5.3 we see the effectiveness of the method during the illumination phase. The first four columns for each scene, tell us about the number and type of vertices in the mesh. The first gives the total number while the other three give the number of unobstructed (*lit*), in umbra and in penumbra (*penum*) respectively. A large percentage of these vertices, the unoccluded and those in umbra, do not need any source visibility determination. The number of source/occluder comparisons given in the next column under *total* were performed entirely for the penumbra vertices. The two last columns in this table show the number of comparisons performed averaged over the penumbra vertices (*av. penum*) and over the total vertices in the scene (*av. total*). The values in these last two columns are particularly important since they show that the number of occluder/source comparisons is not expected to increase very much with an increase in the model size. The average source/occluder comparisons over the total number of vertices is smaller for scene *officecubes* than for scene *15 cubes*, even though the number of polygons in the former is double that of the latter.

Certain important observations can be made if we examine where exactly the time given in Table 5.2 is spent. Table 5.4 shows the individual timings for each major operation of the algorithm, for the scenes used. For each scene under the

	15 cubes		officeA		officeB		officecubes	
	sec	%	sec	%	sec	%	sec	%
build BSP	0.01	1	0.01	1	0.01	1	0.01	0
build mesh	0.57	33	0.62	50	0.60	34	1.07	48
make SVs	0.02		0.03		0.03		0.04	
add to TC	0.06		0.02		0.05		0.12	
single-DM	0.16		0.22		0.17		0.29	
merge DMs	0.32		0.33		0.34		0.60	
misc	0.02		0.02		0.01		0.02	
illuminate	1.10	65	0.58	47	1.15	64	1.14	51
misc	0.02	1	0.02	2	0.01	1	0.02	1

Table 5.4: Analytical times for the construction of the mesh

column labelled *sec* the absolute time is shown, and under % the percentage of time each routine takes of the total is shown. (Notice that the rows and columns of this table are transposed compared to the earlier tables). The first row (*build BSP*) shows for each scene the time taken to build the scene BSP tree using the method described in *Section 3.2.* The data here support the claim we made in that section that the time for building an *ordered* tree from small areas will not be excessive.

In the second row we have the total time for constructing the mesh, which is analysed further in the following five rows. *make SVs* is the time to construct the shadow volumes (create the EV critical surfaces) which is not significant. *add to TC* is the time taken by the shadow tiling, to find the shadow relations and test for potential obstruction. This row provides evidence of the efficiency of our subdivision system. It helps to identify and process almost only the related pairs of polygons, and yet it takes an amount of time never exceeding the 4% of the total computation. The next two rows show the times for building the single DM-trees (*single-DM*) and for merging them to the total DM-tree of the receiver (*merge DMs*). These are both significant values taking up to almost a half of the total processing in certain scenes (e.g. officeA). *misc* refers to various secondary routines of the mesh construction.

The row labelled *illuminate* shows the illumination computation. Following the discussion on Table 5.3 one might have expected this to be less expensive than recorded here. One of the reasons for this is the matching efficiency of the rest of the operations, another is the size of the source. In all the scenes used here the light source is very large, this can verified by the width of the penumbras in

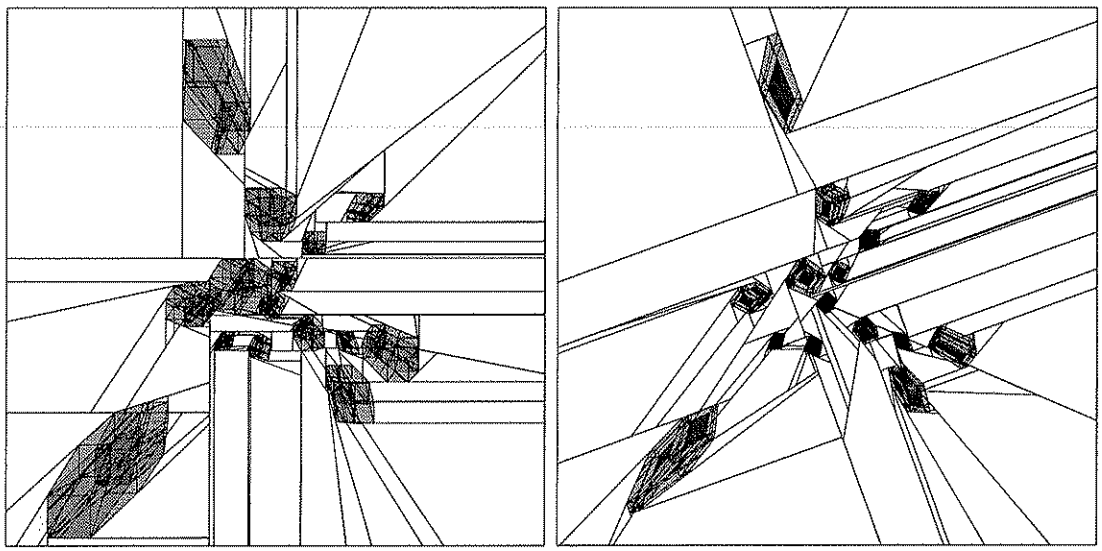


Figure 5.20: The mesh of *15 cubes* scene from (a) a large light source and (b) a source 5 times smaller

the meshes shown in Figure 5.21 and Figure 5.22.

source	mesh vertices				source/occluder comparissons		
	total	lit	umbra	penum	total	av. penum	av. total
large	3676	676	21	2979	6239	2.01	1.69
small	2628	707	213	1708	2750	1.61	1.05

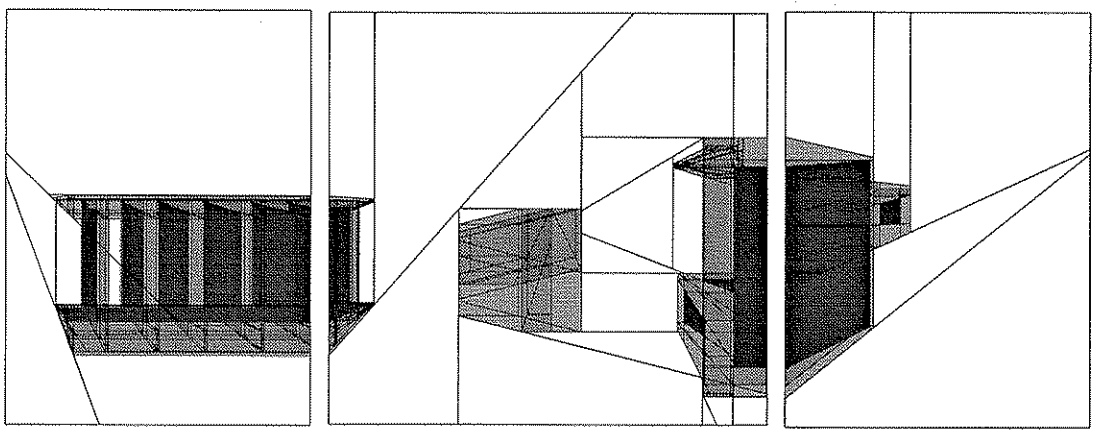
Table 5.5: Difference in mesh vertices by making the source smaller

To give an example of how source size influences the performance, we ran the *15 cubes* scene with a source 5 times smaller than the original. The resulting mesh is less complex (Figure 5.20), with fewer vertices and in particular much fewer penumbra vertices, the illumination time drops dramatically from 1.10s to 0.35s. The construction time also drops since there are fewer intersecting edges in the mesh, but it does not decrease as much since the number of shadow relations remain almost unchanged. The data are shown in Table 5.5 and Table 5.6. The first shows the difference in the number of vertices and the second the times. In both tables *large* refers to the first run, with large source and *small* to the second. These tables indicate that the algorithm is output sensitive, meaning that the amount of computation depends more on the resulting mesh than in the number and geometry of input polygons.

One of the problems reported by other researchers [98] is that the use of DM-tree creates badly shaped cells with excessive subdivision. This is mainly due

source	total time	construct mesh		illuminate mesh	
	sec	sec	%	sec	%
large	1.70	0.57	33.5	1.10	64.7
small	0.77	0.41	53.2	0.35	45.4

Table 5.6: Difference in mesh computations by making the source smaller

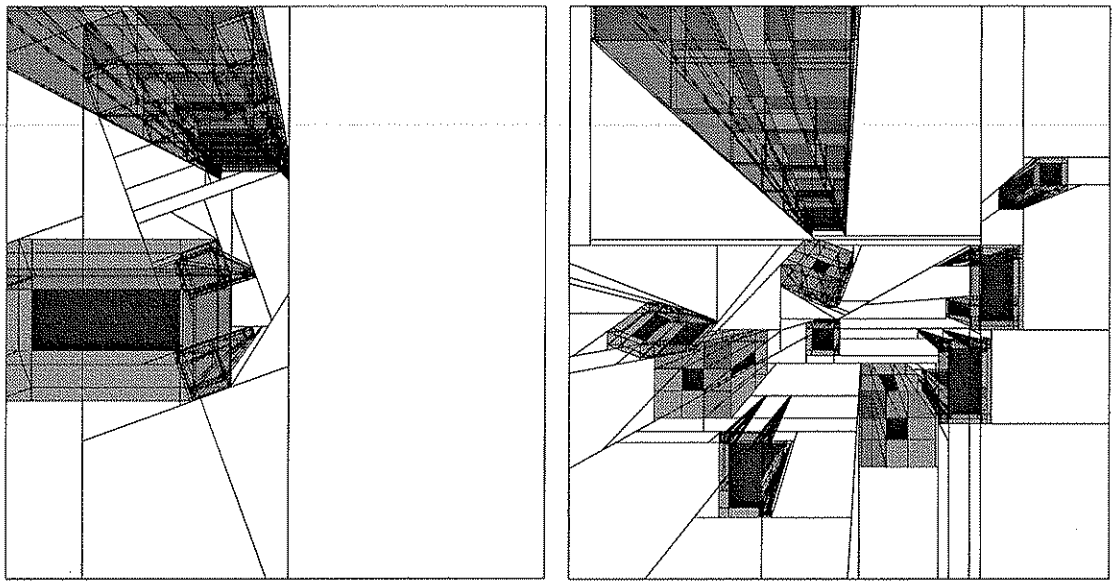


(a) (b) (c)

Figure 5.21: The mesh of (a) the left wall, (b) the floor and (c) the right wall for *officeA*

to the construction edges added to the discontinuities when forming the binary subdivision. One of the benefits of our method is that without any user intervention this problem is very limited. In Figure 5.21 we have the mesh resulting from *officeA* on the left wall, the floor and the right wall, in that order. Here we can see almost no extra subdivision than the necessary. Of course this scene is well suited for our example, since the objects are rectangular with sides parallel to a rectangular source (apart from the cursor), however this pattern is present in all our experiments. In Figure 5.22(a) the source is rotated so that it is not parallel to anything and in Figure 5.22(b), which shows the mesh on the floor from *officecubes*, some objects are randomly placed. In both of these again the subdivision is small.

Another common problem of the existing DM-algorithms is the time complexity. In general this is more than linear, even when the number of shadows grows linearly. The only other work that reports close to linear growth is that of Drettakis [30] but even there the slope is steep. To give a rough idea of the growth rate of our method, we run a set of experiments using the cube scenes. We computed the construction/illumination times for scenes consisting of one to



(a) (b)

Figure 5.22: The mesh on the floor from (a) two objects and a rotated source and (b) the *officecubes* scene

fifteen cubes, by steps of one. As we can see from Table 5.7 and Figure 5.23, in the experiments carried out, not only did we have linear growth but also the marginal cost of adding each extra object is always the same throughout the range of the data.

cubes	1	2	3	4	5	6	7	8
time (ms)	51	115	177	224	256	290	352	409
cubes	9	10	11	12	13	14	15	
time (ms)	423	510	560	614	710	734	766	

Table 5.7: Mesh construction times for scenes with one to fifteen cubes

One of the reasons we have such a reduced growth in this particular experiment is that the cubes are randomly placed without much overlap, as seen from the light source. We cannot interpret these results as showing the algorithm to be linear in the worst case. Arrangements could be constructed where all the projections overlap on the tiling cube and where the number of shadows is quadratic with respect to the number of scene polygons. We can expect, however, that in a large number of scenes where the objects are evenly distributed and the number of shadows we be close to a linear function of the number of polygons, so also will the performance of the algorithm be linear.

Larger scale experiments are required before we can have any conclusive ev-

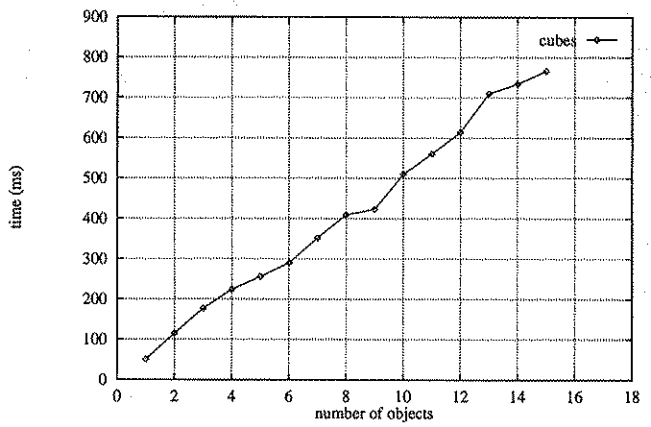


Figure 5.23: Time against number of objects for the cube scenes

idence on the performance of the method. However, by comparing the present results with the results reported by other algorithms [61, 30, 37], and especially the rate of growth, we speculate that this method could be up to an order of magnitude faster. For example Drettakis [30] reports 6.4s for 55 polygons, going up-to 40s for 187 polygons (these times include the identification of EEE emitter and EV non-emitter events but no illumination). In Gatenby's method [37] it takes 1s for 34 polygons jumping up to 116s for 214 polygons (these times are for two sources, including triangulation).

5.5.2 Evaluation of the Incremental Modifications

Having established that our method is at least as good as the existing DM methods, we can now continue with the evaluation of dynamic scenes.

Selected objects from each scene were moved, the results are given in Table 5.8. For each scene we show the objects that moved followed by the translation times (under *transformation*). The transformation is broken into its two components the deletion of the object (*delete*) which includes removal from the tiling and from the DM-trees of the objects receivers, and the addition (*add*) of the object back to the scene. In the next two columns we give the times for rebuilding the scene mesh without the moved object (*rebuild without*) and the total time to rebuild the whole mesh, including the object (*rebuild total*). The values of the latter are taken from Table 5.2. Finally under *del as %rebuild* we give the percentage of time used for removing the object against rebuilding the mesh with out it.

As already stated the addition of a object in the mesh is performed using the same method as for the initial creation of the mesh. So the time taken to add

the object (*add*) should be similar to the difference of the columns *rebuild total* and *rebuild without*. This is approximately the case. The important column in this table is the last one.

scene	object moved	tranformation delete (s)	add (s)	rebuild without (s)	rebuild total (s)	del as %rebuild
15 cubes	cube 3	0.02	0.10	1.62	1.70	1
	cube 5	0.01	0.11	1.58	1.70	1
officeA	cursor	0.01	0.09	1.13	1.23	1
	bookcase	0.02	1.02	0.45	1.23	4
officeB	desk1	0.02	0.19	1.52	1.77	1
	bookcase	0.02	1.20	0.59	1.77	3
officecubes	cube 1	0.02	0.11	2.12	2.24	1
	cube 3	0.02	0.11	2.15	2.24	1

Table 5.8: Timings for mesh computation after transforming objects in the scene

Depending on the positions of the discontinuities in the mesh, the first deletion of certain objects may take longer than normal. This issue is discussed in *Chapter 3*. As we have seen in *Section 5.5.1* in most scenes illuminated by our algorithm the edges of an object are grouped together without many construction edges extending to split other objects. This makes it fast even for the first deletion.

5.6 Summary

In this Chapter we have presented a new method for discontinuity meshing. It uses a tiling cube based subdivision for finding the shadow relations and BSP tree merging for constructing the mesh. The combination of the two provides a means for fast incremental updates in the mesh.

Early experimental results suggest that this method not only can be modified in a fraction of the time of other methods but also it is faster to construct initially and scales better. However, this statement is tentative and requires further experimentation to confirm.

Chapter 6

Conclusion

Shadows are very important in computer generated images. They provide important spatial clues and enhance realism, particularly for interactive applications. Although a great deal of research has been carried out into accurate shadow computation, little has been done concerning interactive applications.

In this thesis we have investigated methods for interactive shadow computation. We have shown that this is achievable for both point and area light sources. In a pre-processing step the complete set of shadows is computed which is incrementally updated on each frame during interaction. Space subdivision methods are used for speeding-up the pre-processing and, more importantly, the incremental updates.

The algorithms have been implemented and tested on a number of different size models confirming that the goal is achievable for scenes of considerable complexity. To give an example of the frame rates that can be achieved we implemented the Unordered SVBSP tree algorithm (*Section 4.2*) on an entry-level Silicon Graphics Indy R4000 100MHz under GL. The scene *office3*, which has 313 initial polygons and 389 shadow polygons, was rendered with an average frame rate per second of 7.3. This is just for repeatedly re-rendering the scene without any changes. If we pick one of the computers and move it about, with its shadows being constantly updated, then the frame rate per second drops down to 7.1. As we ca see, the bottleneck of our system here is the rendering speed of the machine and not the shadow calculations.

6.1 Main Contributions

Then main contributions of this thesis are as follows:

6.1.1 BSP Tree Representation for non-Static Scenes

A study into the use of BSP representation of dynamic scenes has been carried out. The results led to the conclusion that BSP trees are suitable for interactive applications where only a small part of the model is changing. Of course BSP trees are also eminently suitable to changes in view, their original and major purpose. Assuming that, in general, in such applications the moving object will be the same over several frames then only in the first frame is there any time spent on rearranging the tree. In subsequent frames the time needed for updating the tree is mainly dependent upon the proportion of the model changing. Three methods were proposed in *Section 3.1* for minimising the time taken in removing the selected object in its first frame. The suitability of each one depends on the application.

Polygons inserted into a BSP tree may be split into several fragments. During interaction, as the structure of the tree changes polygons may be further split or have their fragments come together under the same node. A method was suggested, using a 2-D BSP tree and a Winged Edge Data Structure on each of the scene polygons, for keeping the number polygons in the tree under control by joining up any fragments coming together on the same node.

6.1.2 Point Light Sources in non-Static Scenes

Using the results of the methods mentioned above, two existing point source shadow algorithms are extended to support dynamic transformation of objects (*Chapter 4*). Both are object space algorithms, one based on a regular space subdivision and the other on Shadow Volume BSP trees.

Experimental results have shown that savings of greater than 90% are achievable during interaction, depending on the size and complexity of the objects involved. This makes the methods suitable for interactive applications with fairly complex scenes, even on low-end workstations.

6.1.3 Visibility Ordering

Experimental results suggested that the problem of finding an invariant order for a set of polygons as seen from an area light source is solvable. A BSP tree based algorithm was developed that can find such an ordering, if it exists, or indicate a minimal splitting set of planes for reducing the viewing area into regions that

allow invariant ordering (*Section 3.2*).

The implementation of the algorithm has shown that indeed for the relatively small polygonal light source and the set of scenes tested, the ordering was always possible.

6.1.4 Area Light Sources in non-Static Scenes

In *Chapter 5* a fast discontinuity meshing algorithm was presented. A spatial subdivision based on the tiling cube, along with the ordering produced by the method mentioned above, were used for identifying potential shadow relations between model polygons.

While traditional DM algorithms trace each discontinuity surface separately through the model, our algorithm traces the whole set of surfaces (shadow) from an occluder together. The intersections of this set of surfaces with the plane of each receiver are found and they are connected together to form a DM-tree which is merged into the DM-tree of the receiver. This process has several advantages over previous methods, such as: reduced time complexity, increased accuracy and explicit classification of each resulting mesh cell in respect to its occluders leading to faster illumination calculations.

Due to the structured creation of the DM, incremental updates are made possible. The shadow information for moving objects can be computed using only a fraction of the computation required to compute the whole shadow information.

6.2 Future Directions

In this thesis solutions to the dynamic shadow problem have been given. Even though these were shown to be sufficient for the requirements of a large class of applications, they by no means offer a complete or optimal solution to the general problem. There are numerous ways in which this research could be extended or improved.

6.2.1 BSP Trees

Efficiency of Dynamic BSP Trees

The method used throughout this thesis for dynamically changing a BSP tree of the scene polygons was the first of the methods described in *Section 3.1*. Even though this algorithm has proved to be very fast, it does not address the problem of maintaining the efficiency of the tree when an internal node is deleted. In our implementation this efficiency issue was not apparent since the trees we used were not optimised. In other applications however a great deal of effort may go into building the tree efficiently. An alternative, which will maintain the efficiency of the tree is the third method. Further investigation and experimental evaluation of this method is required. In particular an additional evaluation of the cost function is needed.

Visibility Ordering

The algorithm presented in *Section 3.2* for ordering the model polygons with respect to an area can be expanded to produce a set of orderings that together can account for the whole of 3-D space. For example by subdividing space and using one order for each subspace.

6.2.2 Point Light Sources

Unordered Shadow Tiling

The shadow tiling method used for interaction, *Section 4.1*, was found to perform less efficiently than the unordered SVBSP tree algorithm. The main reason for this is that we used the BSP tree for ordering the scene polygons, which considerably increased the number of input polygons. This is not a requirement of the algorithm. We suspect that performance comparable to the unordered SVBSP tree algorithm can be achieved by using an unordered tiling, while at the same time retaining the extra benefit of the tiling (i.e. no internal nodes to delete). The methods used in *Chapter 5* for speeding up the shadow tiling could be used here, i.e. building a BSP without splitting and ordering the polygons based on their first and last positions and marking completely blocked polygons to avoid unnecessary shadow generation. Implementation and further experimental evaluation is required to test this idea.

SVBSP for Large, Densely Occluded Environments

Currently the SVBSP algorithm builds a single tree for the whole model, regardless of the model data. This can be extremely wasteful when used for a large densely occluded environment such as an architectural building, where most of the objects are hidden from the source. Methods used for visibility pruning in walkthroughs can be applied here [1, 36, 102, 101]. The model can be partitioned into cells (rooms) connected by portals with an adjacency graph giving information on which cells are visible. Hence their objects can have shadow relations.

A different SVBSP tree can be built for each cell (assuming there is a source in each) with the portals added as special entities connecting the neighboring trees.

6.2.3 Area Light Sources

Building Single DM-Trees

From Table 5.4, where the timings for each module of the DM-algorithm were given, we can see that a significant percentage of the time was taken by the construction of the single DM-trees. Some observations on the structure of the single DM-tree were made in *Section 5.3.2*. These led to a faster implementation of the single tree but the initial goal for which the study was made was not achieved. Further study and a more accurate implementation are needed to produce parameterised DM-trees. Using these parametrised trees, we can eliminate the time taken in the individual construction of the tree.

Another way of speeding up the construction of the tree, as well as the merging and the illumination, is to use the contour lines of the object, as seen from the source instead of individual polygons, for generating shadows. However, in this case self-shadows can be a problem if the objects are not convex.

Dynamic Triangulation

The DM-algorithm presented in this thesis provides an efficient way for finding and maintaining the discontinuity information during interaction. In addition to these discontinuities, the illumination function contains other important information, such as maxima that are usually captured by further triangulation at the DM-cells. In our implementation the triangulation method used for interaction is very straight forward. For every cell that changes, even in the slightest, its whole

triangulation is recalculated. A better approach could be found.

Diffuse Inter-reflections

Finally, as one would expect from a DM-algorithm, we can extend this method to a full radiosity solution ([51, 62]). The heart of our method is the propagation of the discontinuities in groups, forming shadows. This could be very useful when dealing with secondary sources. In general a secondary source S will be subdivided into cells along some discontinuity edges. Instead of processing each cell separately we can process S as one source generating one single DM tree that includes all the information of the cells. The discontinuities on S, of course, are enclosed by the boundary of S. So when creating the shadow volumes from S and an occluder, the PSV and USV are made entirely using boundary edges and vertices of S. We can then use a selection of the important discontinuity edges and vertices to create additional shadow planes which are added to the ISV of the occluder. In this way all cells on S can be added to the tiling together and generate one single DM-tree on each receiver which at the same time will encode the information about their individual geometry.

6.3 Conclusion

In this thesis we have presented methods for rapidly incrementally updating the representation and shadow information of a scene model.

No matter how fast computer hardware becomes the desire for more complex and more realistic models will always overtake the speed advantage. Probably the only way of keeping up with model complexity and providing interaction is with techniques like those presented here. We hope this work will be useful and stimulate further research in this area.

Appendix A

Pseudocode Notation

The pseudocode used in this thesis is presented in a *C*-like syntax. All operators have the same functionality and structure as in the *C* language. Those that differ or do not exist in *C* are listed below.

list[]	square brackets indicate that *list* is a list or an array
$\{a_1, .., a_n\}$	curly brackets denote a set
for (condition) **do**	the repeating statements in a **for** loop are
statements	enclosed between the **do** and the **endfor**
endfor	
if (condition)	multiple statements are allowed between the
statements	condition and the **else** and the **else** and
else statements	**endif** , the **else** may be omitted
endif	

Functions can have as return value any structure, list, array or set.
Which one it is should be clear by the context.

Also the following set operators were used with the usual meaning:

$\exists$	there exists
$\forall$	for all
$\in$	is an element of
$\cap$	intersection
$\cup$	union
$\emptyset$	empty set

Appendix B

Pseudocode for the Unordered SVBSP Tree

In the implementation described here the tree consists of a collection of shadow trees. A shadow tree is a set of $(n+1)$ nodes one at the back of the other, where n is the number of edges in a polygon. The first n nodes hold the shadow planes of the the polygon while the last node holds the polygon itself.

The empty cells have a *NULL* value rather than an *IN* or *OUT* value because these are trivial to compute when required: cells lying in front of any node are *OUT* cells and cells behind a node holding a polygon are *IN*. Note that it is not possible to have empty cells at the back of a shadow plane node.

```
typedef struct svbsp {
        struct svbsp        front;
        struct svbsp        back;
        Plane               rootplane;
        Polygon             face[1];
        Polygon             below[1];
        /* this holds the reference to the last polygon it came infront */
        Polygon             INregion[][1];
        /* the list of polygons falling into the node's IN region */
} Tree ;
/* [1] these fields are only relevant for PP-Nodes */


Polygon initial(Polygon poly)
{
        returns the initial scene polygon of whose poly is a fragment,
        poly == initial(poly) if it was not split while inserting in svbsp
}

void castShadow(Polygon occluder, Polygon receiver)
{
        uses the initial polygons of occluder and receiver for casting
        the shadows
```

```
}

void addShadowPolygon(Polygon receiver, Polygon shadowpoly)
{
        shadowpoly is added as a detail polygon ontop of the receiver
}

Tree shadowTree(Polygon poly, Polygon below)
{
        the shadow volume of poly is build into an SVBSP with a node
        for each shadow plane followed by a node for poly
        the shadow planes used are those of initial(poly) which are
        calculated only once at the begining
        the below field of the polygon node is set to below
}
```

Building the tree

```
Tree builtUSVBSP(Polygon faces[], Light light)
{
        svbsp = NULL;
        for each polygon p_i in faces[] do
                if (p_i is facing the light)
                        svbsp = addPolygonToTree(svbsp, p_i, NULL);
                endif
        endfor
}

Tree addPolygonToTree(Tree svbsp, Polygon p, Polygon below)
/* below is the polygon of the last PPNode p came infront */
{
        if (empty(svbsp))
                if (below != NULL)
                        if (notAlreadyCasting(p, below))
                                castShadow(p, below);
                        endif
                endif
                return shadowTree(p, below);

        c = classifyPolygon(svbsp.rootplane, p, pf, pb);
        if (polygonNodeRoot(svbsp))
                if (c == FRONT)
                        svbsp.front = addPolygonToTree(svbsp.front, p, svbsp.face);
                else /* p is behind the root polygon */
                        addToInRegion(svbsp.INregion, p);
                        addShadowPolygon(initial(p), p);
                endif
        else /* root defined by a shadow plane */
                if (notNull(pf))
                        svbsp.front = addPolygonToTree(svbsp.front, pf, below);
                endif
                if (notNull(pb))
                        svbsp.back = addPolygonToTree(svbsp.back, pb, below);
                endif
```

```
        endif
        return svbsp;
}
```

Using the tree for incremental changes

```
Tree transformObject(Tree svbsp, Object obj)
{
        markFacesOnSVBSP(svbsp, obj);
        svbsp = restoreSVBSP(svbsp, NULL);
        getNewObjectGeometry(obj);
        for each polygon p_i in obj do
                svbsp = addPolygonToTree(svbsp, p_i, NULL);
        endfor
        return svbsp;
}

void markFacesOnSVBSP(Tree svbsp, Object obj)
{
        using pointers created during the building of the tree, the
        position(s) of each object polygon are found on the tree
        if the polygon isin an IN region, it is deleted otherwise the
        nodes where the polygon and its shadow planes are held are
        marked
}


Tree restoreSVBSP(Tree svbsp, Polygon newbelow)
{
        if (empty(svbsp))
                return NULL;
        endif
        if (marked(svbsp))
                if (polygonNodeRoot(svbsp))
                        if (svbsp.below && notDeleted(svbsp.below))
                                removeShadowFrom(svbsp.face, svbsp.below);
                                newbelow = svbsp.below;
                        endif
                        temp = restoreSVBSP(svbsp.front, newbelow);
                        for any polygon p_i in svbsp.INregion[] do
                                temp = addPolygonToTree(temp, newbelow);
                        endfor
                        return temp;
                else /* root is from a shadow plane */
                        large = restoreSVBSP(largest subtree of svbsp, newbelow);
                        return insertTree(large, other subtree of svbsp, newbelow);
                endif
        else /* not marked */
                if (polygonNodeRoot(svbsp))
                        if ((newbelow != svbsp.below) && notAlreadyCasting(svbsp.face, newbelow))
                                castShadow(svbsp.face, newbelow);
                                svbsp.below = newbelow;
                        endif
```

```
                svbsp.front = restoreSVBSP(svbsp.front, svbsp.face);
            else /* root is from a shadow plane */
                svbsp.front = restoreSVBSP(svbsp.front, newbelow);
                svbsp.back = restoreSVBSP(svbsp.back, newbelow);
            endif
            return svbsp;
        endif
}

Tree insertTree(Tree large, Tree small, Polygon newbelow)
/* large has already been restored, marked nodes removed, but not small */
{
        if (empty(small))
            return large;
        else if (polygonNodeRoot(small))
            if (marked(small))
                if (small.below && notDeleted(small.below))
                    removeShadowFrom(small.face, small.below);
                endif
                large = insertTree(large, small.front, newbelow);
                for any polygon p_i in small.INregion[] do
                    large = addPolygonToTree(large, newbelow);
                endfor
                return large;
            else /* root of small is not marked */
                small.below = newbelow;
                return inserPolygonNode(large, small, newbelow);
            endif
        else /* root of small holds a shadow plane */
            insertTree(insertTree(large, small.back, newbelow), small.front, newbelow);
        endif
}

Tree inserPolygonNode(Tree large, Tree small, Polygon newbelow)
/* root of small is a polygon node while root of large can be anything */
{
        if (empty(large))
            if (newbelow != small.below)
                castShadow(small.face, newbelow);
            endif
            small.front = restoreSVBSP(small.front, small.face);
            return shadowTree(small, newbelow);
        else if (polygonNodeRoot(large))
            return insertPPNodeToPPNode(large, small);
        else /* root of large holds a shadow plane */
            if (fromSamePolygon(small, large))
                joinTrees(large, small);
                return large;
            else
                c = classifyNode(large.rootplane, small, sf, sb);
                if (c == FRONT)
                    large.front = inserPolygonNode(large.front, small, newbelow);
                else if (c == FRONT)
                    large.back = inserPolygonNode(large.back, small, newbelow);
                else /* CUT */
```

```
                        large.front = inserPolygonNode(large.front, small, newbelow);
                        large.back = inserPolygonNode(large.back, small, newbelow);
                        large = insertTree(large, small.front, newbelow);
                    return large;
                endif
        endif
}

void joinTrees(Tree large, Tree small)
/* the subtrees are defined by fragments of the same polygon */
/* root of small is a polygon node, root of large can be either */
{
        the polygon held at small is merged with the polygon on large;
        the polygons in the IN region of small are added to the IN region
        of large and the front subtree of small is added to the front
        of the polygon node of large
}

Tree insertPPNodeToPPNode(Tree large, Tree small)
/* since the two subtrees are defined in mutually exclussive spaces this */
/* should never be called, the only reason it may be is because we used the */
/* shadow volumes of the initial polygons. Whatsmore it will only happen */
/* when subtrees have been re-arranged after restoring the tree, so small */
/* must hold, in an INregion[], a fragment of the same polygon as root */
/* of large */
{
        if (fromSamePolygon(small, large))
                joinTrees(large, small);
                return large;
        else
                c = classifyPolygon(large.rootplane, small.face);
                if (c == BACK)
                        add small.face and small.INregion[] to large.INregion[];
                        return insertTree(large, small.front, large.face);
                else
                        if (notAlreadyCasting(small.face, large.face))
                                castShadow(small.face, newbelow);
                        endif
                        compare small.INregion[] against large.rootplane,
                        if any of them are from same initial polygon as
                        large.face then remove that shadow, if any of them
                        are behind large then move them from small.INregion[]
                        to large.INregion[]
                        return inserPolygonNode(large, small, large.face);
                endif
        endif
}
```

Appendix C

Images

The office images of Figures C.1 to C.5 are those used throughout this thesis for evaluation of the methods. These scenes were generated using the Unordered SVSBP algorithm (*Section 4.2*) using one point light source located near the center of the ceiling. Special thanks should go to Mel for coding the original model (*office1*) himself and making it freely available.

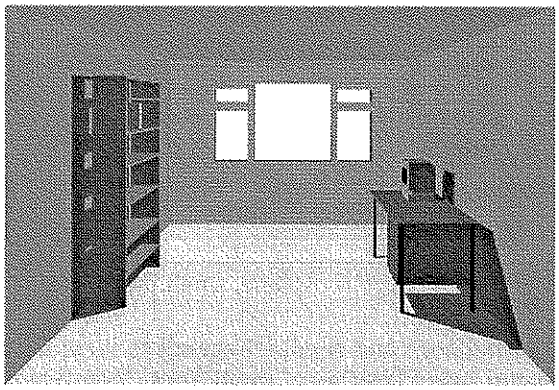


Figure C.1: *office1*, a room with a bookcase, two books, a desk and a computer; 136 polygons

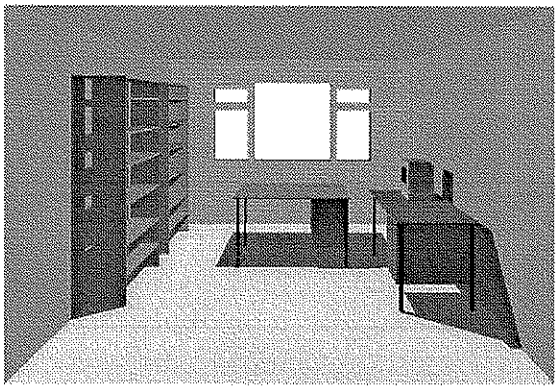


Figure C.2: *office2*, a room with 2 bookcases, two books, two desks and a computer; 211 polygons

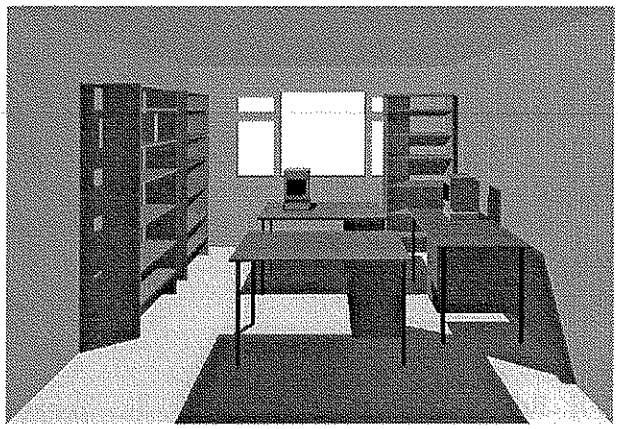


Figure C.3: *office3*, a room with 3 bookcases, two books, three desks and two computers; 333 polygons

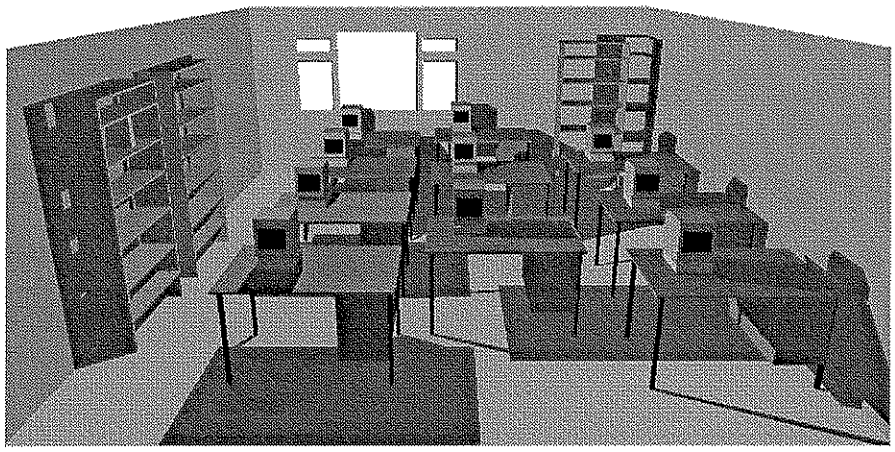


Figure C.4: *office4*, a room with 3 bookcases, two books, ten desks and ten computers; 745 polygons

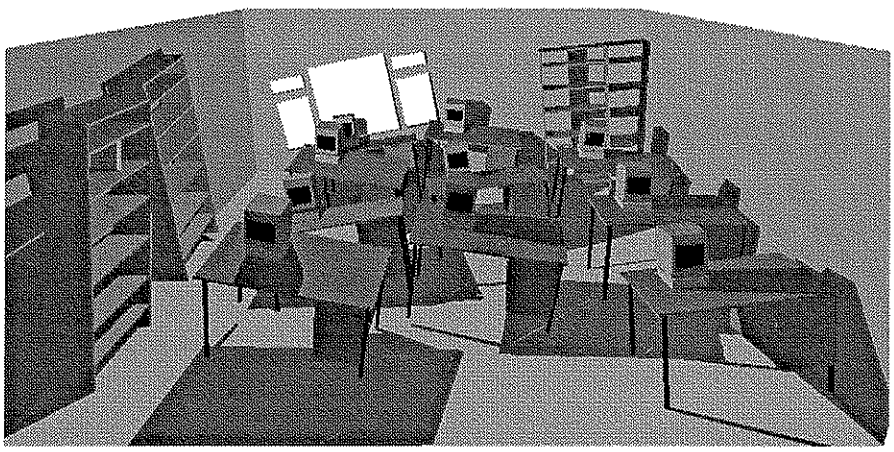


Figure C.5: *office4**, the same as *office4*,but with each object randomly rotated by a small degree

Bibliography

[1] J. M. Airey, J. H. Rohlf, and P. Frederick Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[2] J. Amanatides. Ray tracing with cones. In Hank Christiansen, editor, *ACM Computer Graphics*, volume 18, pages 129–135, July 1984.

[3] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[4] J. Arvo and D. B. Kirk. Fast ray tracing by ray classification. In Maureen C. Stone, editor, *ACM Computer Graphics*, volume 21, pages 55–64, July 1987.

[5] P. Atherton, K. Weiler, and D. Greenberg. Polygon shadow generation. In *ACM Computer Graphics*, volume 12, pages 275–281, August 1978.

[6] D. R. Baum, S. Mann, K. P. Smith, and J. M. Winget. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. In Thomas W. Sederberg, editor, *ACM Computer Graphics*, volume 25, pages 51–60, July 1991.

[7] D. R. Baum, J. R. Wallace, and H. F. Cohen. The back-buffer algorithm: an extension of the radiosity method to dynamic environments. *The Visual Computer*, 2(5):298–306, 1986.

[8] B. G. Baumgart. Geometric modeling for computer vision. AIM-249, STA -CS-74-463, CS Dept, Stanford U., October 1974.

[9] P. Bergeron. A general version of crow's shadow volumes. *IEEE Computer Graphics & Applications*, 6(9):17–28, 1986.

[10] J. F. Blinn. Jim blinn's corner: Me and my (fake) shadow. *IEEE Computer Graphics & Applications*, 8(1):82–86, January 1988.

[11] W. J. Bouknight and K. C. Kelly. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. In *Proc. AFIPS JSCC*, volume 36, pages 1–10, 1970.

[12] L. S. Brotman and N. I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics & Applications*, 4(10):71–81, October 1984.

[13] A. T. Campbell. *Modelling Global Diffuse Illumination for Image Synthesis.* PhD thesis, Department of Computer Science, University of Texas at Austin, December 1991.

[14] A. T. Campbell and D. S. Fussell. Adaptive mesh generation for global illumination. *ACM Computer Graphics*, 24(4):155–164, August 1990.

[15] A. T. Campbell, III and D. S. Fussell. An analytic approach to illumination with area light sources. Technical Report R-91-25, Dept. of Computer Sciences, Univ. of Texas at Austin, August 1991.

[16] S. E. Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. In Forest Baskett, editor, *ACM Computer Graphics*, volume 24, pages 135–144, August 1990.

[17] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. *ACM Computer Graphics*, 23(3):99–106, 1989.

[18] N. Chin and S. Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *ACM Computer Graphics (Symp. on Interactive 3D Graphics)*, pages 21–30, 1992.

[19] H. K. Choi and C. M. Kyung. Pysha: a shadow-testing acceleration scheme for ray tracing. *Computer-Aided Design*, 24(2), February 1992.

[20] Y. Chrysanthou and M. Slater. Dynamic changes to scenes represented as BSP trees. *Computer graphics Forum, (Eurographics 92)*, 11(3):321–332, 1992.

[21] Y. Chrysanthou and M. Slater. Shadow Volume BSP trees for fast computation of shadows in dynamic scenes. In *Proceedings of the ACM Symposium of Interactive 3D Graphics*, pages 45–50, Monterrey, California, March 1995.

[22] Y. Chrysanthou and M. Slater. Incremental changes to scenes illuminated by area light sources. In *Submitted for publication*, January 1996.

[23] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. In John Dill, editor, *ACM Computer Graphics*, volume 22, pages 75–84, August 1988.

[24] M. F. Cohen and D. P. Greenberg. The Hemi-Cube: A radiosity solution for complex environments. In B. A. Barsky, editor, *ACM Computer Graphics*, volume 19, pages 31–40, August 1985.

[25] M. F. Cohen, D. P. Greenberg, D. S. Immel, and P. J. Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics & Applications*, 6(3):26–35, March 1986.

[26] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In Hank Christiansen, editor, *ACM Computer Graphics*, volume 18, pages 137–145, July 1984.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[28] F. Crow. Shadow algorithms for computer graphics. *ACM Computer Graphics*, 11(2):242–247, 1977.

[29] G. Drettakis. *Structured Sampling and Reconstruction of illumination for Image Synthesis*. PhD thesis, Department of Computer Science, University of Toronto, January 1994.

[30] G. Drettakis and E. Fiume. A fast shadow algorithm for area light sources using back-projection. In Andrew Glassner, editor, *ACM Computer Graphics*, pages 223–230, July 1994.

[31] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics : Principles and practice*. Addison-Wesley Publishing Company, second edition, 1990.

[32] H. Fuchs, G. D. Abram, and E. D. Grant. Near real-time shaded display of rigid objects. *ACM Computer Graphics*, 17(3):65–72, 1983.

[33] H. Fuchs, Z. M. Kedem, and B. Naylor. Predetermining visibility priority in 3-D scenes (preliminary report). *ACM Computer Graphics*, 13(3):175–181, August 1979.

[34] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *ACM Computer Graphics*, 14(3):124–133, 1980.

[35] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, pages 16–26, April 1986.

[36] T. A. Funkhouser, C. H. Sequin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. *ACM Computer Graphics (Symp. on Interactive 3D Graphics)*, 25(2):11–20, March 1992.

[37] N. Gatenby. *Incorporating Hierarchical Radiodity into Discontinuity Meshing Radiosity.* Ph.D. thesis, University of Manchester, Manchester, UK, 1995.

[38] N. Gatenby and W. T. Hewitt. Radiosity in computer graphics: a proposed alternative to the hemi-cube algorithm. In *Second Eurographics Workshop on Rendering*, 1991.

[39] N. Gatenby and W. T. Hewitt. Optimizing discontinuity meshing radiosity. In *Fifth Eurographics Workshop on Rendering*, pages 249–258, Darmstadt, Germany, June 1994.

[40] D. W. George, F. X. Sillion, and D. P. Greenberg. Radiosity redistribution for dynamic environments. *IEEE Computer Graphics & Applications*, 10(4):26–34, July 1990.

[41] Z. Gigus, J. Canny, and R. Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13(6):542–551, June 1991.

[42] Z. Gigus and J. Malik. Computing the aspect graph for the line drawings of polyhedral objects. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(2):113–133, February 1990.

[43] A. S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics & Applications*, pages 60–70, March 1988.

[44] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[45] Robert A. Goldstein and Roger Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, January 1971.

[46] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In Hank Christiansen, editor, *ACM Computer Graphics*, volume 18, pages 213–222, July 1984.

[47] D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics & Applications*, 11(5):79 –85, 1991.

[48] P. E. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In Forest Baskett, editor, *ACM Computer Graphics*, volume 24, pages 309–318, August 1990.

[49] E. Haines and J. Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, 1991.

[50] E. A. Haines and D. P. Greenberg. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics & Applications*, 6(9):6–16, 1986.

[51] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In Thomas W. Sederberg, editor, *ACM Computer Graphics*, volume 25, pages 197–206, July 1991.

[52] P. Heckbert. *Simulating Global Illumination Using Adaptive Meshing.* PhD thesis, CS Division (EECS), Univ. of California, Berkeley, June 1991.

[53] P. Heckbert. Discontinuity meshing for radiosity. *Third Eurographics Workshop on Rendering*, pages 203–226, May 1992.

[54] P. Heckbert. Radiosity in flatland. *Computer graphics Forum, (Eurographics 92)*, 11(3):181–192, 1992.

[55] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In Hank Christiansen, editor, *ACM Computer Graphics*, volume 18, pages 119–127, July 1984.

[56] J. C. Hourcade and A. Nicolas. Algorithms for antialiased cast shadows. *Computers and Graphics*, 9(3):259–265, 1985.

[57] H. Hubschman and S. W. Zucker. Frame to frame coherence and the hidden surface computation: Constraints for a convex world. *ACM Computer Graphics*, 1:129–162, April 1982.

[58] S. E. Hudson. Adding shadows to a 3D cursor. *ACM Transactions on Computer Graphics*, 11(2):193–199, April 1992.

[59] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In David C. Evans and Russell J. Athay, editors, *ACM Computer Graphics*, volume 20, pages 269–278, August 1986.

[60] R. Krishnaswamy, G. S. Alijani, and S. Shyh-Chang. On constructing binary space parition trees. In *ACM 18th Annual Computer Science Conference Proceedings*, pages 230–235, 1990.

[61] D. Lischinski, F. Tampieri, and D. P. Greenberg. Discontinuity meshingfor accurate radiosity. *IEEE Computer Graphics & Applications*, 12(6):25–39, November 1992.

[62] D. Lischinski, F. Tampieri, and D. P. Greenberg. Combining hierarchical radiosity and discontinuity meshing. In James T. Kajiya, editor, *ACM Computer Graphics*, volume 27, pages 199–208, August 1993.

[63] G. M. Maxwell, M. J. Bailey, and V. W. Goldschmidt. Calculations of the radiation configuration factor using ray casting. *Computer-Aided Design*, 18(7):371–379, September 1986.

[64] S. McPartlin. A foundation for BSP trees. Second year progress report. Technical report, University of Edinborough, October 1994.

[65] S. Muller and F. Schoffel. Fast radiosity repropagation for interactive virtual environments using a shadow-form-factor-list. In *Fifth Eurographics Workshop on Rendering*, pages 325–342, Darmstadt, Germany, June 1994.

[66] K. Mulmuley. *Computational Geometry, An Introduction Through Randomized Algorithms.* Prentice-Hall, Inc., Prentice Hall, Englewood Cliffs, NJ 07632, 1994.

[67] B. F. Naylor. *A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes.* PhD thesis, University of Texas at Dallas, May 1981.

[68] B. F. Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer-Aided Design*, 22(4):138–148, May 1990.

[69] B. F. Naylor. Sculpt: An interactive modeling tool. In *Proceedings of the Graphics Interface '90*, pages 138–148, 1990.

[70] B. F. Naylor. Interactive solid geometry via partitioning trees. In *Proceedings of the Graphics Interface '92*, pages 11–18, 1992.

[71] B. F. Naylor. Partitioning tree image representation and generation from 3d geometric models. In *Proceedings of the Graphics Interface '92*, pages 201–212, 1992.

[72] B. F. Naylor. Constructing good partitioning trees. In *Proceedings of the Graphics Interface '92*, pages 181–191, 1993.

[73] B. F. Naylor, J. Amandatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *ACM Computer Graphics*, 24(4):115–124, 1990.

[74] T. Nishita, Y. Miyawaki, and E. Nakamae. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In Maureen C. Stone, editor, *ACM Computer Graphics*, volume 21, pages 303–310, July 1987.

[75] T. Nishita and E. Nakamae. An algorithm for half-tone representation of three-dimensional objects. *Information Processing Society of Japan*, 14:93–99, 1974.

[76] T. Nishita and E. Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. *Proc. COMPSAC 83: The IEEE Computer Society's Seventh Internat. Computer Software and Applications Conf.*, pages 237–242, November 1983.

[77] T. Nishita and E. Nakamae. Continuous tone representation of tree-dimentional objects taking account of shadows and interreflection. *ACM Computer Graphics*, 19(3):23–30, July 1985.

[78] T. Nishita and E. Nakamae. Continuous tone representation of three-dimensional objects illuminated by sky light. In David C. Evans and Russell J. Athay, editors, *ACM Computer Graphics*, volume 20, pages 125–132, August 1986.

[79] T. Nishita, I. Okamura, and E. Nakamae. Shading models for point and linear sources. *ACM Transactions on Computer Graphics*, 4(2):124–146, April 1985.

[80] M. S. Paterson and F. F. Yao. Binary partitions with applications to hidden surface removal and solid modeling. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry*, pages 23–32, 1989.

[81] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *Discrete Computational Geometry*, 5:485–503, 1990.

[82] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *Journal of Algorithms*, 13:99–113, 1992.

[83] A. Pearce and D. Jevans. Exploiting shadow coherence in ray tracing. In *Proceedings of Graphics Interface '91*, pages 109–116, June 1991.

[84] P. Poulin and J. Amanatides. Shading and shadowing with linear light sources. In C. E. Vandoni and D. A. Duce, editors, *Computer graphics Forum, (Eurographics 90)*, pages 377–386. North-Holland, September 1990.

[85] H. Radha, R. Leonardi, M. Vetterli, and B. Naylor. Binary space partitioning tree representation of images. *Journal of Visual Communication and Image Representation*, 2(3):201–221, 1991.

[86] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In Maureen C. Stone, editor, *ACM Computer Graphics*, volume 21, pages 283–291, July 1987.

[87] E. M. Reingold and J. S. Tilford. Tidier drawing of trees. *IEEE Transaction on Software Engineering*, SE-7(2):223–228, March 1981.

[88] D. Salesin, D. Lischinski, and T. DeRose. Reconstructing illumination functions with selected discontinuities. *Third Eurographics Workshop on Rendering*, pages 99–112, May 1992.

[89] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX,, September 1969.

[90] W. B. Seales and C. R. Dyer. Shaded rendering and shadow computation for polyhedral animation. *Proceedings of the Graphics Interface '90*, pages 175–182, 1990.

[91] M. Sharir and M. H. Overmars. A simple output sensitive algorithm for hidden surface removal. *ACM Transactions on Computer Graphics*, 11(1):1–12, January 1992.

[92] M. Slater. A comparison of three shadow volume algorithms. *The Visual Computer*, 9(1):25–38, October 1992.

[93] M. Slater, M. Usoh, and Y. Chrysanthou. The influence of shadows on presence in immersive virtual environments. In M. Goebel, editor, *Eurographics Workshop on Virtual Environments (Accepted for publication in a Springer Verlag book on Virtual Environments, after a further refereeing process)*, 1995.

[94] B. E. Smits, J. R. Arvo, and D. H. Salesin. An importance-driven radiosity algorithm. In Edwin E. Catmull, editor, *ACM Computer Graphics*, volume 26, pages 273–282, July 1992.

[95] S. Spencer. The hemisphere radiosity method: a tale of two algorithms. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 127–135, 1990.

[96] A. J. Stewart and S. Ghali. Fast computation of shadow boundaries using spatial coherence and backprojections. In Andrew Glassner, editor, *ACM Computer Graphics*, pages 231–238. ACM SIGGRAPH, July 1994.

[97] I. E. Sutherland, R. F. Sproull, and R. A. Schumaker. A characterization of ten hidden surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

[98] F. Tampieri. *Discontinuity Meshing for Radiosity Image Synthesis.* Ph.D. thesis, Cornell University, Ithaca, NY, 1993.

[99] T. Tanaka and T. Takahashi. Shading with area light sources. In Werner Purgathofer, editor, *Eurographics '91*, pages 235–246. North-Holland, September 1991.

[100] S. J. Teller. Computing the antipenumbra of an area light source. In Edwin E. Catmull, editor, *ACM Computer Graphics*, volume 26, pages 139–148, July 1992.

[101] S. J. Teller and P. Hanrahan. Global visibility algorithms for illumination computations. In James T. Kajiya, editor, *ACM Computer Graphics*, volume 27, pages 239–246, August 1993.

[102] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. *ACM Computer Graphics*, 25(4):61–69, July 1991.

[103] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partition trees. *ACM Computer Graphics*, 21(4):153–162, 1987.

[104] A. O. Tokuta. Motion planning using Binary Space Partitioning. In *IEEE International Workshop on Intelligent Robots and Systems IROS '91*, pages 86–90, Osaka, Japan, November 1994.

[105] E. Torres. Optimization of the binary space partition algorithm (BSP) for visualization of dynamic scenes. *Computer graphics Forum, (Eurographics 90)*, 9(3):507–518, 1990. C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland.

[106] D. Tost. An algorithm of hidden surface removal based on frame-to-frame coherence. In Werner Purgathofer, editor, *Eurographics '91*, pages 261–273. North-Holland, September 1991.

[107] D. Tost and P. Brunet. A definition of frame-to-frame coherence. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation '90 (Second workshop on Computer Animation)*, pages 207–225. Springer-Verlag, April 1990.

[108] J. van Ee and C.W.A.M. van Overveld. Casting shadows with approximated object space accuracy by means of a modifies Z-buffer. *The Visual Computer*, 10:243–254, 1994.

[109] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. In Jeffrey Lane, editor, *ACM Computer Graphics*, volume 23, pages 315–324, July 1989.

[110] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A ray tracing solution for diffuse interreflection. In John Dill, editor, *ACM Computer Graphics*, volume 22, pages 85–92, August 1988.

[111] K. Weiler and K. Atherton. Hidden surface removal using polygon area sorting. *ACM Computer Graphics*, 11(2):214–222, July 1977.

[112] L. Williams. Casting curved shadows on curved surfaces. *ACM Computer Graphics*, 12(3):270–274, August 1978.

[113] A. Woo and J. Amanatides. Voxel occlusion testing: A shadow determination accelerator for ray tracing. In *Proceedings of Graphics Interface '90*, pages 213–220, May 1990.

[114] A. Woo, P. Poulin, and A. Fourier. A survey of shadow algorithms. *IEEE Computer Graphics & Applications*, 10(6):13–31, 1990.