A Comparison of Three Shadow Volume Algorithms

Mel Slater[1]
Department of Computer Science,
QMW, University of London,
Mile End Road,
London E1 4NS,
UK.

Abstract

This paper describes and compares three different approaches to computing shadows each based on the idea of shadow volumes, a basic algorithm, the Shadow Volume BSP Tree algorithm, and a third based on 2D space subdivision, Shadow Tiling. Binary Space Partition trees are used to organise the polygons in the scene in a front-to-back order from the point of view of the light source, and then shadows on a polygon are computed by clipping the polygon to the shadow volumes of polygons closer to the light source. The three algorithms differ in their approach to minimising the number of comparisons of a polygon with the shadow volumes of its predecessors, and one of the algorithms represents the total shadow volume itelf by a BSP tree. The algorithms are compared analytically and statistically.

Keywords

shadows, shadow volume, BSP trees, 3D graphics

1. Introduction

This paper is concerned with the problem of generating shadow umbras formed from point or directional light sources, in the context of polygonally described scenes. In 1977 Franklin Crow's classic survey of shadow algorithms [10] distinguished three approaches to this problem known at that time:

(1) A polygon scan-line approach: This is due to Appel [1] and Bouknight and Kelley [4] where shadow edges are projected onto the polygon being displayed, and during the scan-line process such shadow edges mark a transition into or out of shadow.

(2) A two pass hidden surface algorithm: The first pass uses the light source position as viewpoint, distinguishing fragments of polygons which are visible to the light source, and those which are hidden, and therefore in shadow. These shadow polygons are added to the original scene, and in the second pass the scene is traversed from the camera viewpoint and hidden surface elimination and rendering are performed. An exemplar of this approach was given by Atherton, Weiler and Greenberg [2] who used polygon area sorting for the hidden surface computation.

[1] e-mail address: mel@dcs.qmw.ac.uk

(3) Shadow volumes: For each shadow generating polygon facing the light source, shadow planes are formed by each edge of the polygon, from the triangle of the edge vertices and the light source position (and clipped by the view volume). Such shadow planes, termed *shadow polygons* by Crow, are added as invisible polygons to the scene data structure. During the scan-conversion process, the shadow polygons mark transitions into or out of shadowed regions.

In this paper the term *shadow plane* refers to a semi-infinite shadow polygon formed from edge vertices and light source position. The volume formed from a shadow generating scene polygon by the "insides" of its shadow planes and its "back" side is a *shadow volume* (umbra). This volume can be used as a clipping region in object space for all other polygons which are in front of the shadow generating polygon. Those polygon fragments which are inside the clipping region are in shadow, whereas those outside are lit. Chin and Feiner [6] used a Binary Space Partition (BSP) tree to efficiently and elegently represent the total shadow volume formed by a set of polygons. Their algorithm is discussed in detail below.

Bergeron [3] generalised Crow's shadow volume algorithm to non-planar polygons and open polyhedra. He also extended Crow's classification of algorithms to include:

(4) z-buffer computation: This was introduced by Williams [33], who created z-buffers for light source positions. For each visible (x,y,z) position in image space there is an equivalent position (x',y',z') in the image space for the light source. The z value at location (x',y') in the light source z-buffer is compared to z' to determine whether or not the pixel corresponds to a point in shadow. Shapiro Brotman and Badler [26] used a modified z-buffer in combination with shadow volumes as a method for computing soft shadows, simulating penumbra. The shadow polygons are rendered into a data structure stored at the z-buffer locations. This data structure modifies the shading during the rendering phase. Area light sources were simulated by sampling points on the light source surfaces, and treating each as a point light.
(5) Ray Tracing: This is the first of the global illumination models, and was introduced into computer graphics by Whitted [32]. A primary ray is traced from the viewpoint through a "pixel location" to the nearest intersection with an object. "Shadow feeler" rays are traced from the object intersection point to the light source positions, and the object intersection point is in shadow with respect to a light source, if the corresponding shadow feeler intersects any object prior to meeting the light source. The original ray is traced recursively through reflected and refracted rays. A primary ray is passed through each pixel location, or pixels are sub-sampled for anti-aliasing.

Another more recent global illumination model is that of radiosity. This was introduced by Goral et. al. [16] and involves the computation of diffuse interreflections between surfaces in a closed environment. The algorithm involves computation of form-factors, which measure the proportion of energy leaving one surface which reaches another (taking into account object occlusions). The hemi-cube method of form-factor computation [8],[9] is similar to one of the methods of shadow computation (Shadow Tiling) considered later in this paper. In ray tracing there is an explicit step involved in the computation of shadows, though no geometrically specified shadows are ever explicitly computed. In the radiosity method there is no explicit shadow computation step - shadows are implicitly

discovered as those parts of the scene which receive a smaller fraction of light energy directly from the light sources, and indirectly from other non-light emitting objects in the scene, compared to neighbouring parts of the scene. Note that the radiosity method directly models area light sources, so that penumbras are automatically generated. Other methods for generation of soft shadows are described in [20][24].

Woo, Poulin and Fournier [34] provide the most comprehensive survey of shadow algorithms to date, including some theoretical assessment of algorithm performance. The present paper concentrates more narrowly on a comparison of three methods which combine ideas from (2) and (3). They are two pass algorithms, and they employ shadow volumes as object space clipping volumes to compute polygon shadows in the first pass. Each algorithm employs Binary Space Partition (BSP) trees to order the polygons with respect to light sources, and for hidden surface rendering. The Chin and Feiner algorithm [6] mentioned earlier also uses a Shadow Volume BSP (SVBSP) tree to represent the total shadow volume. The purpose of the present paper is to compare this algorithm with two others, a basic (brute force) one, and one using a 2D space subdivision, Shadow Tiling (ST).

In section 2, there is a brief overview of BSP trees. In section 3 the basic shadow algorithm is introduced followed by a description of the SVBSP tree approach in section 4. The Shadow Tiling algorithm is described in section 5. Sections 6 and 7 describe an implementation and discuss the results of a comparison of the three approaches, with conclusions in section 8.

It should be noted that the algorithms discussed in this paper are appropriate for rapid display of 3D scenes based on a relatively small number of polygons (of an order up to about 1000, though this obviously depends on hardware limitations). The speed of shadow computation is at the expense of photo-realism, which can at present only be achieved with global illumination methods - often requiring hours of processing on a serial computer. The algorithms are for use in a context where the shadow computation time must be at most in seconds rather than minutes or hours, and where the scene is to be viewed from a variety of viewpoints, with the camera position and orientation possibly changed interactively.

2. BSP Trees

BSP trees were first introduced by Schumuker et. al. [25] who discovered that separating planes in an environment can be arranged to determine clusters of objects, so that objects in a cluster on the same side of a plane as the viewpoint cannot be obscured by those on the far side. This insight was later used by Fuchs et. al. [12][13] to construct an efficient hidden surface removal algorithm for polygonally defined scenes.

The BSP tree method relies on the fact that polygons can be defined to have a "front" and a "back". If the plane equation of a polygon is $l(x,y,z) = 0$, where $l(x,y,z) = ax+by+cz+d$, then the front facing normal is (a,b,c). For any point (x,y,z), if $l(x,y,z) > 0$ then the point is in "front" of the plane, if $l(x,y,z) < 0$ then the point is "behind" the plane, and on the plane when $l(x,y,z) = 0$.

Given a scene defined by a set of polygons, one is chosen to be the root polygon. All other polygons are partitioned into three sets - a front set and a back set, according to whether the polygon is behind or in front of the root, and a set consisting of those in the same plane as the root. Polygons which have some vertices behind and some in front of the root are split into two by the plane of the root. The tree construction algorithm proceeds recursively on the front and back sets, until the entire polygon set has been exhausted. This procedure is carried out in object space.

The authors of [12] suggest a heuristic approach to reduce the number of splits. A split is caused each time a target polygon intersects the plane of the root. Whenever a root is to be selected, choose a number of polygons (they chose up to 15) and for each of these find the number of other polygons in the list which are intersected by its plane. Choose as the root the one which intersects the smallest number of others. Since for a scene of a thousand polygons the tree construction time might take several minutes, the additional work to do this splitting reduction would be worth while since it reduces the final number of polygons and therefore the actual rendering time of the scene. Other approaches for reducing the number of splits have been proposed [30].

To render the polygons in the tree from any camera position the tree is traversed in a special in-order manner. The Centre of Projection is classified by the plane of the root polygon. If it is in front of the root, then the traversal algorithm is first applied recursively to the back set of the root, followed by displaying the root polygon, followed by applying the traversal algorithm to the front set. If it is behind the root then the traversal is applied to the front set, then the root is displayed, and then traversal applied to the back set. Back-face removal is achieved by not displaying the root in the second case, for in this case the root polygon clearly faces away from the COP. Note that the display function is responsible for clipping, projection and rendering. Gordon and Chen [17] have shown that a significant improvement in rendering speed can be attained when the tree is traversed in front-to-back order in conjunction with a dynamic data structure for scan-line polygon filling.

The BSP tree implementation [12] showed that a near real-time frame rate can be achieved with this method without extra-ordinary special graphics hardware, although the computation time for the tree itself could be significant. BSP trees are therefore particularly suited for applications where the scene itself is static, but where the camera frequently moves. This is the case for example, in architectural applications where the camera moves through a set of buildings [5], and other similar applications [23].

In this algorithm change of camera position only requires traversing the tree in a different order - it does not require reconstruction of the tree. However, a change in geometry of an object in the scene normally would require reconstruction of the tree. Hence the method is especially suitable for situations where a camera moves through an unchanging scene. However, Fuchs et. al. desribed in [12] a method for handling motion of objects in pre-determined paths and Torres [31] has introduced the notion of using auxiliary planes in a representation where a BSP tree is built for each object, with a combined scene BSP tree constructed such that objects can be rapidly added to and deleted from the scene.

BSP trees have also been used by Thibault and Naylor [30] for the representation of polyhedral solids and set operations on polyhedra in the representation and rendering of CSG models. This approach shows that BSP trees can also be used interactively in circumstances where the scene itself is changing - for example, Thibault and Naylor describe an application where a tool is interactively added or subtracted from a work piece. This is described in detail in [21]. The issue of computing changes to BSP trees dynamically, caused by changing object geometry, is taken up in section 8.

3. Base Line Algorithm for Computing Shadows

The crucial point of the BSP tree method is that the tree is constructed once and for all, and then for any viewpoint a unique back-to-front ordering of all the polygons is determined (given this particular tree). This suggests the following base line algorithm for the generation of shadows. (Convex polygons are assumed throughout for implementation purposes, but the algorithms in general do not depend on this assumption).

Assume for the moment that there is a single light source. Construct a BSP tree. Traverse the BSP tree from the position of the light source, thereby constructing a list of polygons in front-to-back order $p_0,p_1,...,p_n$. Polygons which face away from the light source are ignored. From the point of view of the light position, p_0 is the nearest polygon and p_n is the furthest.

In the following discussion, the term *shadow generating polygon* refers to a polygon which is considered as potentially casting shadows on other polygons which are behind it from the point of view of the light source. A *target polygon* is a polygon which is being considered as potentially having shadows cast upon it. All polygons (except the one furthest from the light source) will in turn be shadow generating polygons. All polygons (except the one nearest the light source) will in turn be target polygons for each shadow generator.

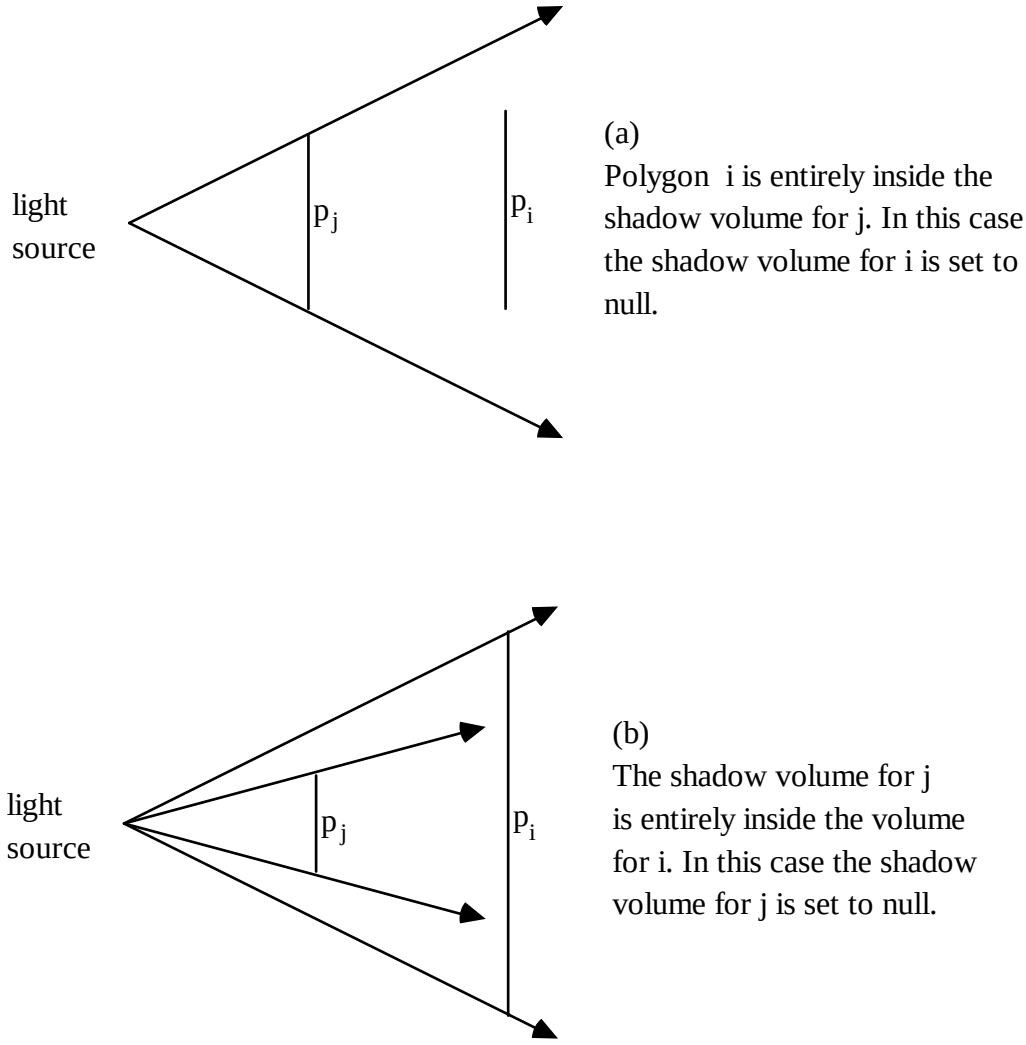

Figure 1
Shadow Volumes

The algorithm proceeds as follows: Let SV_i be the shadow volume generated by p_i. First, construct SV_0. Compute the shadow cast by SV_0 on p_1. If p_1 is entirely inside SV_0 then set the shadow volume for p_1 to be null, otherwise create SV_1. This is illustrated in Figure 1(a). If SV_0 is entirely inside, or identical to SV_1 then set SV_0 to be null. This is illustrated in Figure 1(b). Continue with p_2, p_3, ..., p_n. Hence p_i is compared against the non-null shadow volumes SV_0, SV_1, ..., SV_{i-1}. SV_i is created if p_i is at least partly outside at least one of these previous shadow volumes. Each of these shadow volumes will remain non-null provided that it is not contained within (or equal to) SV_i. This is illustrated in the pseudo code below:

Definition 3.1

```
proc CreateShadows(p)
begin
```

```
        Create(SV0);
        for i = 1 to n do
                polygonShadow = ∅;
                for j = 0 to i-1 do
                        if SVj ⊃ pi then
                                polygonShadow  = pi ;
                                 SVi = ∅;
                        else
                                polygonShadow = clip(pi,SVj):
                                Create(SVi);
                        endif
                        if polygonShadow - ∅ then
                                pi.shadows = pi.shadows ∪ {polygonShadow};
                                if  SVi ∈ SVj then SVj = ∅
                                endif
                        endif
                od
        od
end
```

There remains the question of adding the newly created polygon shadows (polygonShadow in the above code) to the data structure. This is achieved by each polygon keeping a list of pointers to those polygons which are shadows on it (p_i.shadows). Hence whenever a shadow fragment is computed, the pointer to it is added to the list of polygon shadows belonging to the target polygon. An alternative to this would be to store the shadow polygons in the same node of the BSP tree as the target (which is the method used in [6]).

The major work involved in this algorithm is computation of the shadow volume, and the clipping of target polygons to this volume. When a plane belonging to the shadow volume is computed it is crucial that the correct "back" and "front" sides are represented by the polygon plane equation, so that a consistent definition of what is inside and outside the shadow volume is achieved. The convention has been adopted here that the "front" side of the shadow plane is on the shadow side.

The clipping of a target polygon to a shadow volume is similar to the Sutherland-Hodgman clipping algorithm [29]. For a given shadow plane defining the volume, if all vertices of the polygon lie at the front, back or on the shadow plane, then the result is straightforward. Similarly, if one edge is "on" and the remainder are all in front, or at the back, then again the result is clear. If, however, some of the vertices are behind and some in front of the plane, then the polygon must be split by the plane into a front and a back fragment. Only the front fragment is kept, and passed on to the next shadow plane constituting the shadow volume. The final fragment (if any) after processing by all of the shadow planes bounding the volume is the polygon shadow. This is added to the list of polygons belonging to the original target, as described above.

It is important to realise that the primitive computation involved in this is exactly that needed for the construction of the original BSP tree. This too, of course, requires the splitting of polygons by planes (in this case - of the root). It follows that the shadow

generating algorithm can easily be constructed given the software tools required for the BSP tree.

In the implementation, to be discussed more fully in section 6, all original scene polygons have their colour values computed before the shadows are generated (only flat shading was used for simplicity). When a shadow is generated, its colour is modified by subtracting away the contribution of the current light source.

Multiple light sources can be incorporated into this algorithm straightforwardly. There is one pass for each light source. The first light source is treated as above. After this has been processed, the position of the second light source is used to traverse the original BSP tree, to obtain the list of polygons in front-to-back order from the point of view of this light source. These polygons are treated as exactly as in the case of the first light source. The only additional requirement now is that when each target polygon is encountered, the shadows on it (ie, in the list of pointers to the polygon shadows) must also be considered as targets. Since these are ordinary polygons, these too will have a list of pointers to the shadows on them as computed during this process. Care must be taken to ensure that a polygon shadow is treated as a target only when it was produced by a different light source than the one currently being processed. Hence each polygon also stores the set of lights which have been "removed" from contributing to its colour. In the case of the original scene polygons, these sets will be empty.

Finally, the scene is rendered using the in-order traversal for BSP trees outlined above. This is possible since the BSP tree is itself unchanged throughout the shadow generating algorithm. The information associated with the polygons stored in the nodes of the tree has been changed, since they each now have a list of pointers to shadow polygons, and these in turn may contain a list of pointers to the "shadows on shadows", and so on, depending on the number of light sources. Therefore, displaying a polygon becomes more complex than when rendering without shadows.
What is required is that during the rendering process, polygon shadows associated with a scene polygon are displayed in the order necessary to achieve the correct shadow effect. First, the original scene polygon must be displayed. Next all of the first level shadow polygons must be displayed. First level shadow polygons are those produced directly from intersecting the scene polygon with the shadow volumes from the different light sources. Next all of the second level shadow polygons must be displayed. These are those generated by intersecting the shadow volumes with the first level polygons - and so on.

The process is illustrated in Figure 3. This shows three light sources throwing shadows onto a polygon labelled 0. The shadows are labelled 1, 2 and 3. It is assumed that the lights are processed in order 1,2 and 3. First light source L1 is processed, producing shadow 1. Next L2 is processed producing shadow 2 and $1 \cap 2$. Finally L3 is processed, producing shadows 3, $1 \cap 3$, $2 \cap 3$ and $1 \cap 2 \cap 3$. The ownership relations between the various polygons is shown in Figure 4. (Polygon a "owns" polygon b when b is in the list of pointers to shadows on a). When polygon 0 is displayed, it is displayed in the order of levels shown in Figure 3, ie, 0, 1, 2, 3, $1 \cap 2$, $1 \cap 3$, $2 \cap 3$, $1 \cap 2 \cap 3$. In general when there are k light sources, a single shadow generating polygon can generate up to 2^k -1 shadow polygons on a target.

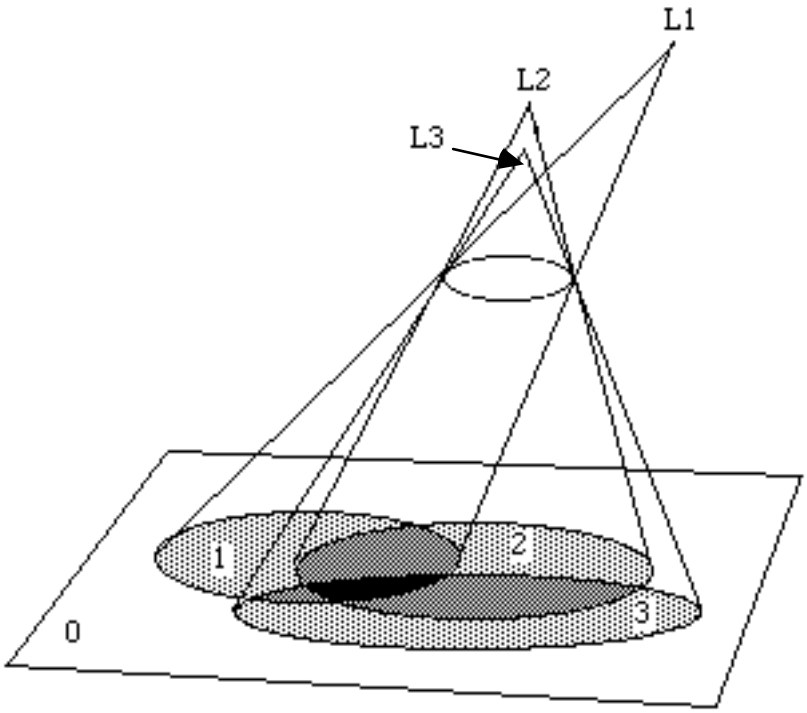

Figure 3
Shadows for Multiple Light Sources

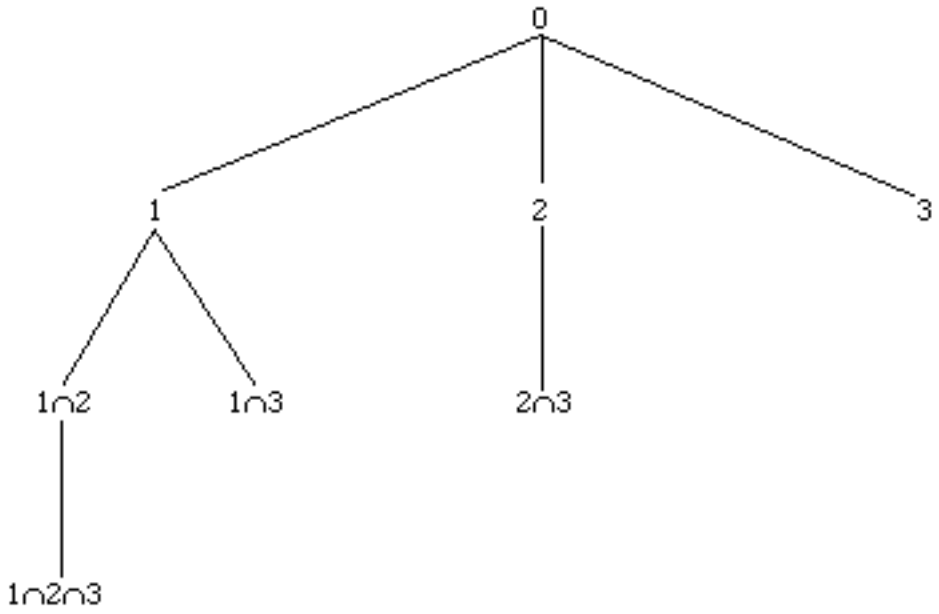

Figure 4
Ownership Relations Amongst

the Polygons of Figure 3

The basic algorithm described in this section requires n passes over the list of polygons for each light source. In the ith pass polygon p_i is tested against each of the i shadow volumes SV_0, SV_1, ..., SV_{i-1}. Therefore, the number of tests made for the first light source is n(n+1)/2. Suppose that s_i is the number of shadows on p_i after the first light source is processed. Then the number of comparisons for the second light source is:

$$\f(n(n+1),2) + \i\su(i=1,n,i\ s_i)$$

Since $0 \le s_i \le i$, the upper bound for the number of tests for two light sources is $O(n^3)$. This argument can be extended to show that the upper bound for k light sources is $O(n^{k+1})$.

4. The SVBSP Tree Algorithm

Chin and Feiner [6] introduced a new algorithm which efficiently represents the shadow volumes produced by traversing the original BSP tree from each light source position. This relied on work by Thibault and Naylor [30] who showed that a BSP tree can be used to represent an arbitrary polyhedral solid. Each interior node represents the plane which embeds the set of polygons stored at this node. The plane partitions space into a "front" (or "out") half-space, and a "back" (or "in") half-space, and all descendants of the node are partitioned accordingly (splitting polygons if necessary). The empty regions at the leaves are labelled with "in" or "out" regions, depending on whether they represent the back or front of their parent. A two-dimensional example is shown in Figure 5.

Thibault and Naylor also showed that a BSP tree can be constructed incrementally. Briefly, one polygon supplies the plane for the root, and then subsequent polygons are filtered down the tree. For a given polygon, if it lies wholly in front or behind the root then it is added (recursively) to the appropriate child of the root, otherwise it is split by the plane of the root and each fragment is added recursively to the corresponding child. The final nodes are labelled as "in" or "out" as described above.

In Chin and Feiner's shadow algorithm the list of polygons for a given light source position are processed in front-to-back order as above. Consider first the situation of a single light source. Using the light source position construct the shadow planes for polygon p_0 and form a BSP tree *for these shadow planes* (this is called the SVBSP tree). Next consider polygon p_1. This is filtered down the tree as described in the previous paragraph though with some important differences. The fragments which reach "out" nodes (ie, in front of all the shadow planes - using the convention of section 3 above) are in shadow. These polygon shadows are not added to the new tree, but may be referenced as polygon shadows of p_1, as in section 3, or else stored

in the same node of the scene BSP tree as p_1 (which is the method preferred by Chin and Feiner).

The fragments which reaches "in" nodes (behind all of the shadow planes) are lit by this source. However, it is the *shadow planes formed from these lit polygons* which are now added to the (Shadow Volume) BSP tree under construction. Polygons p_2, p_3, ...,p_n are processed in the same way. After polygon p_i has been dealt with, each of the polygons p_1, p_2, ... ,p_i have had their shadows computed and stored, and the current SVBSP tree represents the total shadow volume so far.

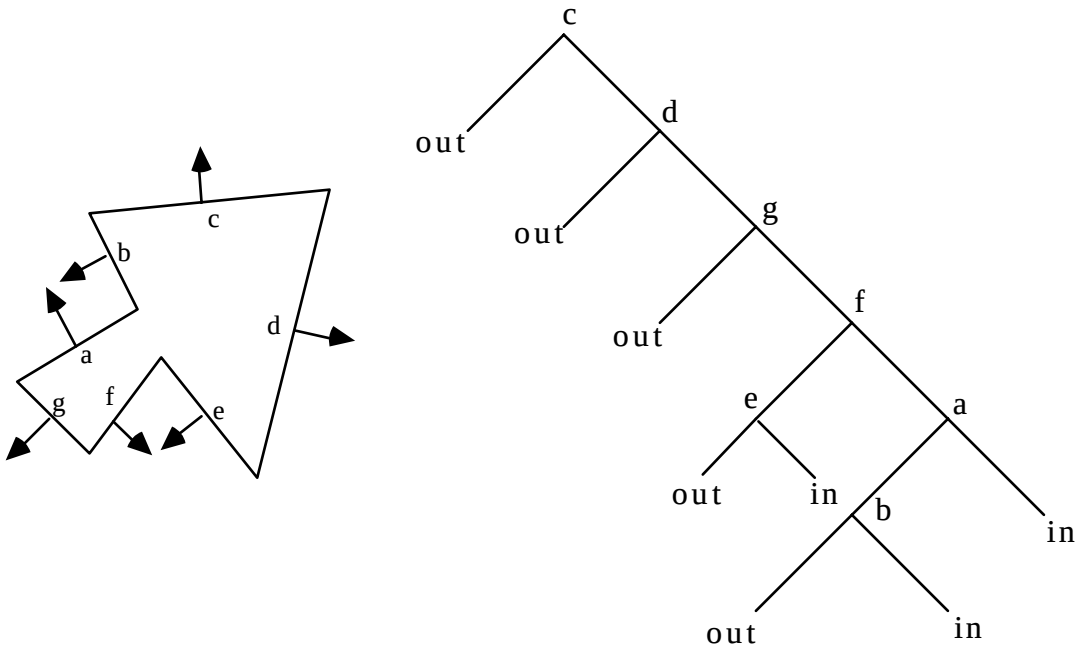

Figure 5
A Concave Polygon is Represented by a BSP Tree
(The arrows point to the outside of the polygon)

Multiple light sources are treated in much the same way as in the basic algorithm. For each subsequent light source, the old SVBSP tree can be discarded and a new SVBSP tree constructed, except that now in addition to the original scene polygons, the shadows computed by the earlier light sources must be taken into account.

This algorithm clearly has an improved efficiency compared to the base line one. Polygon p_i is tested against the SVBSP tree formed from the first i polygons, in time (at best) proportional to log(i). Therefore the complete set of polygons is treated in time proportional to $\Sigma\log(i) = \log(n!) \sim n \log(n)$. This is for a single light source. For two light sources, with s_i shadow polygons on polygon p_i, the time is proportional to:

$$\i\su(i=1,n,\B(\log(i) + s_i \log(i)))$$

which is bounded above by $O(n^2 \log(n))$. For k light sources, an upper bound would be $O(n^k \log(n))$.

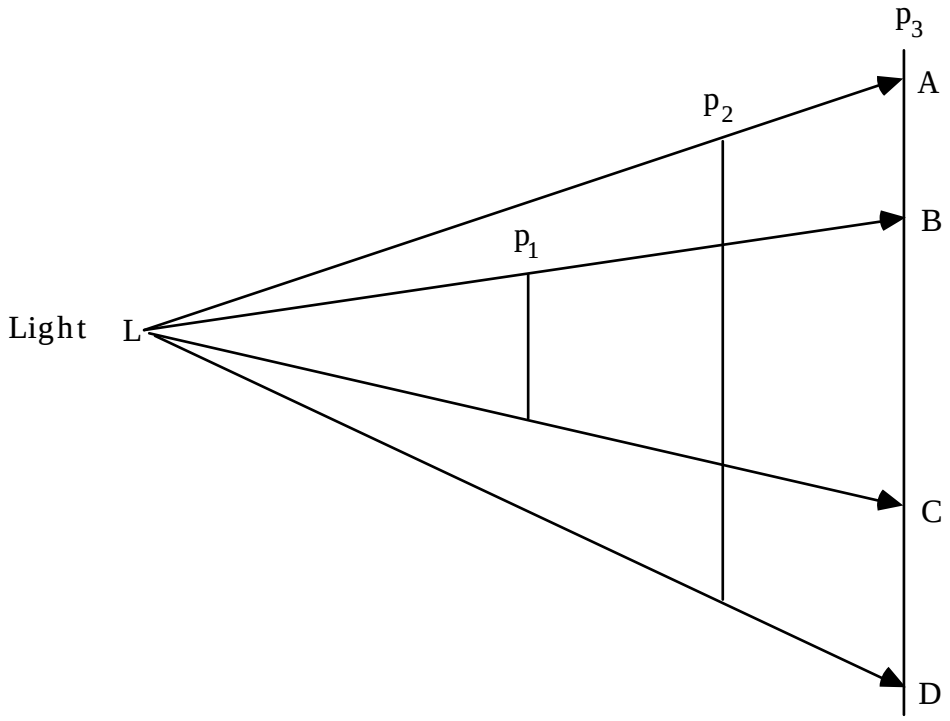

Figure 6
The Need to Merge Adjacent Fragments in the Same Region

An implication of the SVBSP tree algorithm as described so far is that it would generate more polygons for rendering than the base line one. This is illustrated in Figure 6 which shows two polygons p_1 and p_2, casting shadows on the third, p_3. In the base line algorithm, only one shadow AD would be cast on p_3. In the SVBSP tree algorithm three shadows might be cast. This is because after processing of the first two polygons, the SVBSP tree represents the three volumes LAB, LBC and LCD since polygon p_2 is split into three fragments by p_1. However, this would be avoided if adjacent fragments that were both in "out" or "in" regions were merged on their common edge. Moreover, if all fragments of a polygon are lit, or all in shadow, then the fragments can be discarded in favour of the original polygon. Using these techniques only one shadow would be cast in the example of Figure 6.

5. The Shadow Tiling Algorithm

The SVBSP tree algorithm improves the speed of the base line algorithm by reducing the time spent on the inner loop of Definition 3.1 from linear to logarithmic. This is achieved by the binary tree data structure. An alternative approach is to reduce the time of this inner loop by only testing each new polygon against the shadow volumes which are likely to intersect it. This can be achieved by a tiling data structure to be described next.
Consider a single light source. Construct a cube around the light position, and divide each face of the cube into a rectangular grid, where each grid element is called a tile.

The scene polygons are projected onto the faces of the cube, for example, as shown in Figure 7, and their identifiers are stored in the appropriate elements of the two dimensional array associated with each face. Now two polygons which do not overlap in any of the six tiling coordinate systems cannot cast shadows on each other, with respect to this light source. Polygons which do overlap are potentially in a shadow relationship to each other. The probability of polygons sharing tiles actually casting shadows on each other depends on the resolution of the tiling coordinate systems. Obviously, the greater the resolution, the higher the probability. This Shadow Tiling (ST) structure can be used to improve the performance of the basic algorithm.

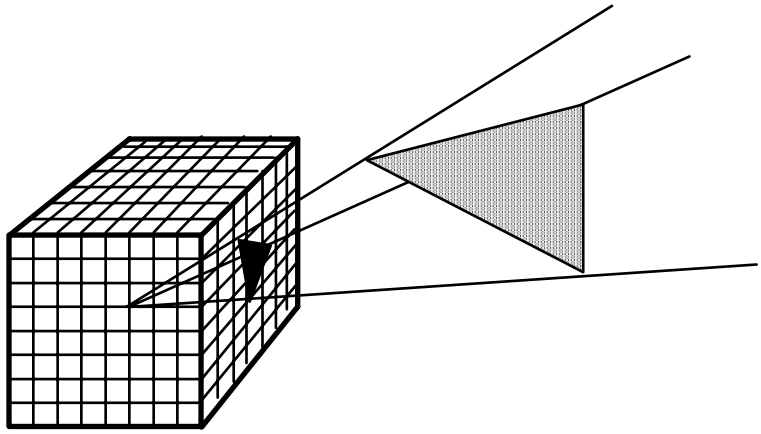

Figure 7
The Shadow Tiling

This idea is not entirely new. It is similar to the method employed by Bouknight and Kelley [4], where all polygons were projected to a spherical coordinate space around the light source. Each polygon was then tested for overlap against all of the others to determine which polygons could cast shadows on which. In the context of ray tracing, Haines and Greenberg [19] describe a shadow testing accelerator using this idea of a surrounding cube. The scene is projected onto the faces of the cube, which are uniformly divided into tiles. They used this method as a way of limiting the the number of object intersection tests for shadow feeler rays. Each shadow feeler ray intersects a particular tile, and only the objects stored at that tile need to be tested for intersection with the shadow feeler ray. Cohen and Greenberg [8], use a similar structure for computing form-factors in the radiosity method. A hemi-cube is centred on a polygonal patch, where each face of the hemi-cube is tiled, and all all other patches in the environment are projected to it. Such space subdivision techniques go back a long way in computer graphics, an early use was Encarnacao's scan-grid method for hidden surface computation [11]. Another use is in damage repair algorithms for raster displays [27], and volume space subdivision is used as a ray tracing acceleration technique [7][14][15].

The ST structure is built incrementally. The first polygon p_0 is put into the tiling and its Shadow Volume SV_0 is created. Polygon p_i is put into the tiling, but this process also delivers the set S_i of identifiers of polygons already in the tiling which share tiles with p_i. The shadow volumes corresponding to the polygons in S_i are then used to

generate shadows on p_i. Essentially the algorithm is the same as Definition 3.1, except that the inner loop does not consider all polygons so far processed, but only that subset which share tiles with the current target.

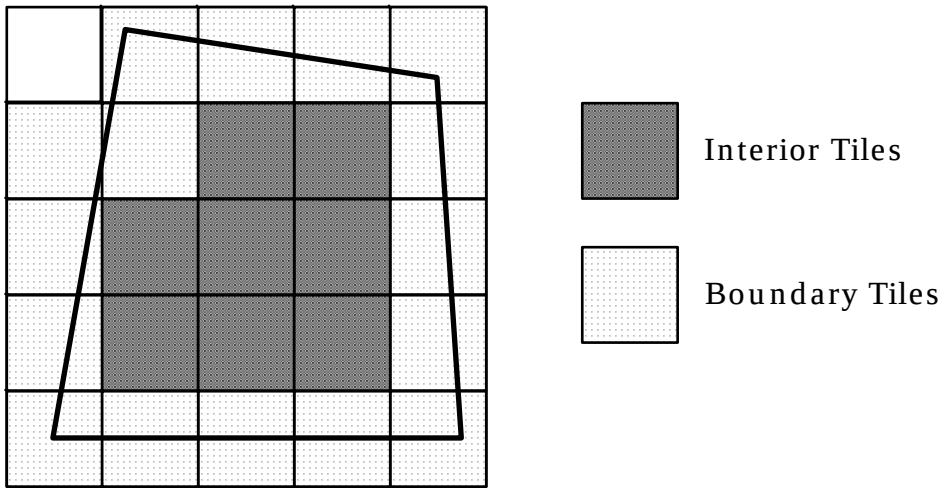

Figure 8
The Boundary and Interior Tiles of a Polygon

For any particular face of the ST cube, the tiles of a polygon are divided into two sets - interior and boundary, as shown in Figure 8. The boundary tiles are those found by tiling the polygon edges. The interior tiles are those tiles which are not on the boundary, but inside the polygon. This distinction is useful in the shadow computation context, since once an interior tile corresponding to a polygon has been set in the tiling, there is no point in ever adding another polygon identifier to this tile. This is because such a tile already represents an area which is in shadow with respect to this light source. Adding subsequent polygons into the tile can not change this fact.

Hence once a tile has been set which belongs to the interior of a polygon it can never be overwritten by subsequent polygons. Polygon identifiers are therefore added to the tiling according to the following rule:

Suppose (i,j) is the position in the Tiling Array to which a new polygon identifier is to be written. Let Tiling[i,j] represent the set of polygon identifiers stored at tile (i,j). Suppose boundary tiles of a polygon have the negative of the polygon identifier put into them, whereas interior tiles have the positive identifier stored. Let the polygon identifier be pI (>0).

```
if Tiling[i,j] contains a single positive element then donothing
else
        if (i,j) is an interior tile of pI then Tiling[i,j] = {pI}
        else
                Tiling[i,j] = Tiling[i,j] ∪ {-pI}
        endif
endif
```

It follows from this that at any time Tiling[i,j] will either be empty, contain a set consisting of a single positive element, or a set of negative elements. Once Tiling[i,j] contains a set with a single positive element it will never be changed.

The time efficiency of this algorithm depends on the coherence of the scene with respect to the light sources. When polygon p_i is being processed there are i polygons already in the tiling, each with a corresponding shadow volume. A proportion π_i of these will intersect p_i in the tilings. Hence, for a single light source, the time is proportional to:

$$Kn + \i\su(i=1,n,i\ \pi_i)$$

where K is the constant time to put a polygon into the tiling. The best case performance is obtained when the π_i values are close to zero. This, however, corresponds to a rather uninteresting scene where no polygon casts a shadow on any other. At the other extreme the worst performance occurs when π_i is close to 1. In this case the time will be greater than that for the basic method, because of the overhead represented by K. This corresponds to a scene where the polygons are stacked one in front of the other from the point of view of the light source.

This algorithm is likely to have a comparable or better performance than the SVBSP method in circumstances where

$$\pi_i < \f(log(i),i)$$

Obviously, the distribution of the π_i values is an empirical question, varying from scene to scene. In order to test the algorithms in practise a number of scenes were investigated, and the results discussed in the next section.

6. Implementation

All three algorithms were implemented using a common software base for the construction of BSP trees. Hence, in so far as possible, differences in performance between the three algorithms are unlikely to be the results of the vagaries of implementation. The implementation was written entirely in C, and the compiler used was GNU/gcc [28].

In constructing the original scene BSP tree, no special attempt was made to reduce the number of polygons splits, other than presentation of the polygons to the tree making procedure in an approximate "outside" to "inside" order. This means that polygons furthest from the centre of the the scene were presented to the BSP tree construction algorithm earlier than those nearest the centre. Since the main scenes consisted of a room with furniture, this avoided the problem of the smaller interior polygons splitting the larger exterior polygons (eg, the walls of the room).

There are three details where the implementation of SVBSP may differ from Chin and Feiner. The first is that no attempt was made to keep the SVBSP tree balanced, although it is not clear whether this attempt was made in Chin and Feiner's actual

implementation. The second is that the shading values for shadows were computed as the shadows were found, rather than outside of the SVBSP construction procedure. The third is that polygon shadows constructed were stored as "children" of the corresponding target polygons, rather than in the same node of the BSP tree as the target. This allowed the original BSP tree to remain unchanged throughout the shadow generating procedure. As is the case with the Chin and Feiner implementation only convex polygons were used, and polygons were processed individually rather than using edge connectivity to identify silhouettes of polyhedra. Adjacent polygon fragments in the same type of region ("in" or "out") were not merged - nor were they merged in the Chin and Feiner implementation.

The tiling structure was represented as a two dimensional array of sets of integers, where a "set" was represented as a linked list. However, the set of polygon identifiers corresponding to the polygons intersecting a given target was represented as integer bit-masks. This data structure ensured that set operations such as union could be computed easily and efficiently. The size of the tiling array could be varied, but the figures in the next section are all based on an array size 128×128. This turned out to be the optimal resolution amongst the range considered - all other sizes resulted in a slower time.

The program was implemented on a SUN SparcStation 1+ with 8mb of memory, which has a RISC architecture. As a comparison, the program was also executed on a SUN 3/160, also with 8mb memory, which has a CISC architecture and a 68881 floating point co-processor.

7. Results

A number of scenes were constructed in order to examine differences in performance between the three algorithms. A basic scene consisted of a room containing a desk with a computer on top, and a bookcase. Further scenes were constructed from this by adding additional desks and bookcases (up to two further desks and two further bookcases) with ten scenes selected. Each scene was generated with one and two light sources. The first light source was inside the room near the centre of the ceiling, and the second outside the room shining through a doorway into the room. Each combination of scene with each of the first and both light sources was executed five times. Some examples are shown in the plates. The timing results were averaged over the five executions.

The same scenes were generated using the light sources in different positions, but did not lead to qualitative differences in the results, so that only the set of results for one set of light source positions is presented here.

To provide a contrast, scenes were also generated using the Haines data set [18]. These were three examples of the recursive tetrahedron based on size factors 3, 4 and 5 casting shadows on themselves and on another single polygon.

The dependent variable was time to compute shadows (in seconds) averaged over five executions. The time to compute shadows does not include the time to compute the scene BSP tree, which is the same for all three algorithms. The independent variable

was the number of polygons after construction of the initial scene BSP tree (Nbsp). This was considered more important than the original number of input polygons (N) since different numbers of such output polygons can result depending on the root selection strategy in scene BSP tree construction. Nearly all polygons were four sided so that the number of polygons is almost an exact proportion of the number of edges.

The results are given in Tables 1-3, and summarised in Figures 9-11. These figures show the scatter plots of shadow generation time against number of (BSP tree generated) input polygons, together with least squares regression lines (interpolated lines for Figure 11). In the case of the room scenes the results suggest that the tiling algorithm consistently performs marginally better than the SVBSP algorithm, however, the rate of change of the time with number of input polygons is the same for the two algorithms. It is as if there is a constant (marginal) overhead which operates against the SVBSP algorithm.

The recursive tetrahedron provides scenes which do not favour the tiling algorithm - they are highly coherent scenes in the sense of polygons being packed behind each other from the point of view of the light source. Table 2 shows the SVBSP approach performing faster than the tiling in the case of 1025 input polygons. It is also comforting that the figures in Table 3 and 4 are similar to those of Table 1 in the Chin and Feiner paper. The ratio of execution time of Tetra1024 to Tetra256 is 6.1 in their paper compared to 4.6 in this paper. Their number of output polygons in the Tetra256 scene is 723 compared to 773 in this paper, and 3345 compared to 3268 in the Tetra1024 case.

Table 1
Shadow Computation Time (seconds)
Room Scenes

N	Lights	Nbsp	basic(s)	tiling(s)	svbsp(s)
138	1	228	0.6	0.8	1.2
174	1	308	0.8	0.9	1.3
192	1	358	1.4	1.2	1.7
210	1	412	1.0	1.1	1.4
228	1	438	1.6	1.3	1.7
264	1	542	1.8	1.5	1.9
246	1	614	2.7	1.8	2.4
264	1	668	2.4	1.7	2.2
282	1	694	3.1	2.0	2.5
318	1	798	3.5	2.1	2.6
138	2	228	2.5	2.6	4.0
174	2	308	3.2	3.2	4.7
192	2	358	5.8	4.9	6.3
210	2	412	4.3	3.9	4.8
228	2	438	6.8	5.5	7.0
264	2	542	8.2	6.3	7.0
246	2	614	10.8	7.9	9.6
264	2	668	9.4	7.0	8.3

282	2	694	12.2	8.7	10.4
318	2	798	14.0	9.6	10.7

Table 2
Shadow Computation Time (seconds)
Recursive Tetrahedron

N	**Nbsp**	**basic** (s)	**tiling** (s)	**svbsp** (s)
65	83	0.2	0.3	0.4
257	299	1.9	1.6	1.6
1025	1135	21.3	10.8	7.3

Table 3
Total Number of Output Polygons
Recursive Tetrahedron

N	**Nbsp**	**Nbasic**	**Ntiling**	**Nsvbsp**
65	83	158	158	178
257	299	649	649	773
1025	1135	2672	2672	3268

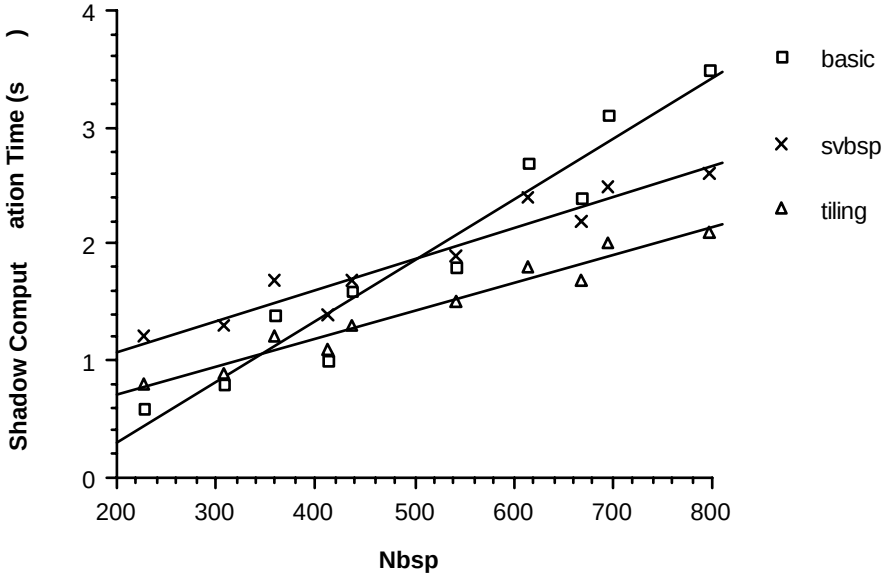

Figure 9
Timings for the Room Scenes with One Light

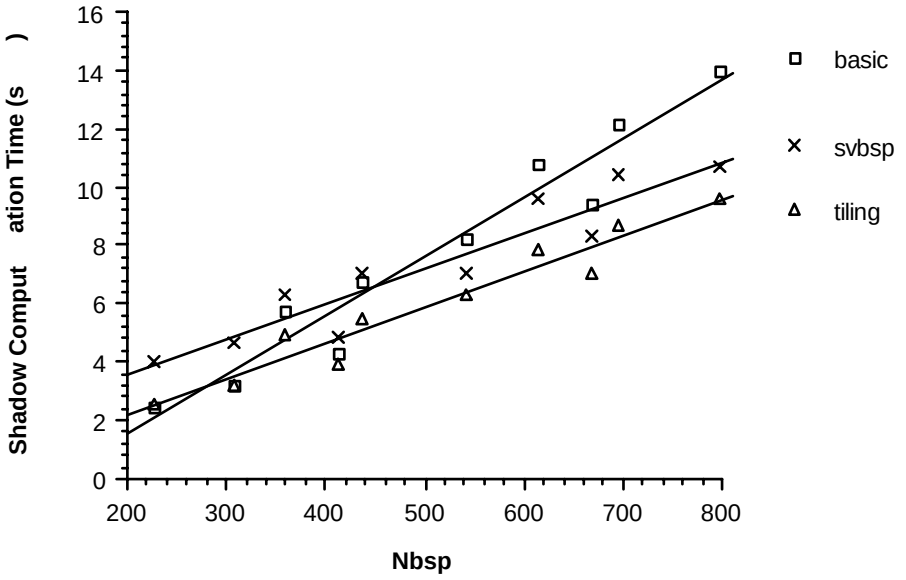

Figure 10
Timings for the Room Scenes with Two Lights
(Lines are Least Squares Regression)

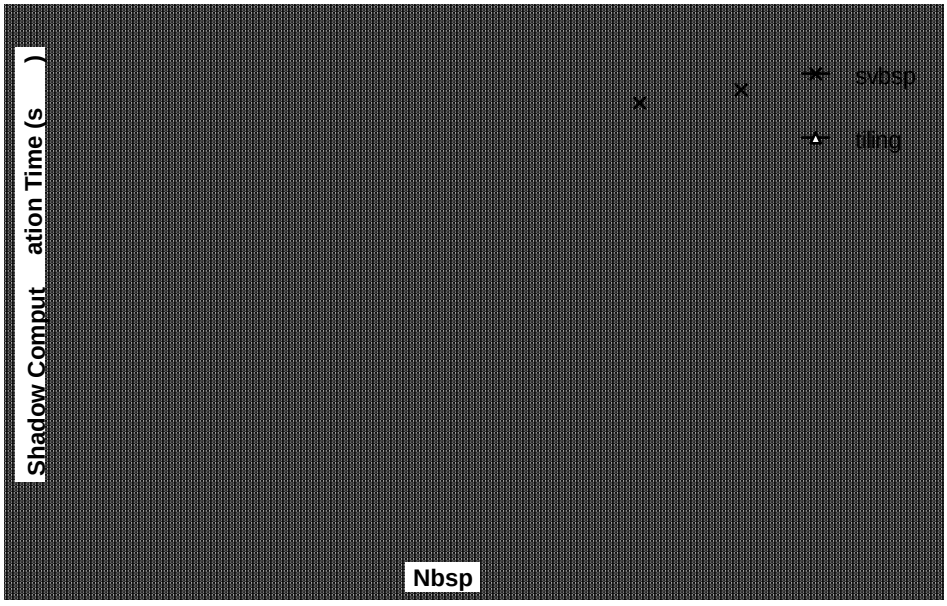

Figure 11
Timings for the Room Scenes with Two Lights
(Lines are Linear Interpolations)

Figure 11 shows an alternative representation of the shadow generation times for the room scenes with two lights, comparing only the tiling and SVBSP algorithms. This highlights the roughly constant differences between the two results over the range of scenes. Both algorithms are memory intensive, the tiling one in order to store and access the six tiling arrays, and the SVBSP in accessing and augmenting the SVBSP tree. However, the tiling algorithm has the advantage that the memory associated with the tiling array may be stored in contiguous blocks, whereas the SVBSP tree may be fragmented. The differences in performance may simply be due to this.

The identical programs were also executed on a SUN 3/160. Interestingly, the relationship between the tiling and SVBSP algorithm changes for the room scenes with two lights. Figure 12 shows this. This may be due to the different memory allocation architectures between the two types of machine. The ST algorithm stores the shadow information in the tiling arrays which are contiguous blocks of memory, whereas the SVBSP tree would be stored in a fragmented manner.

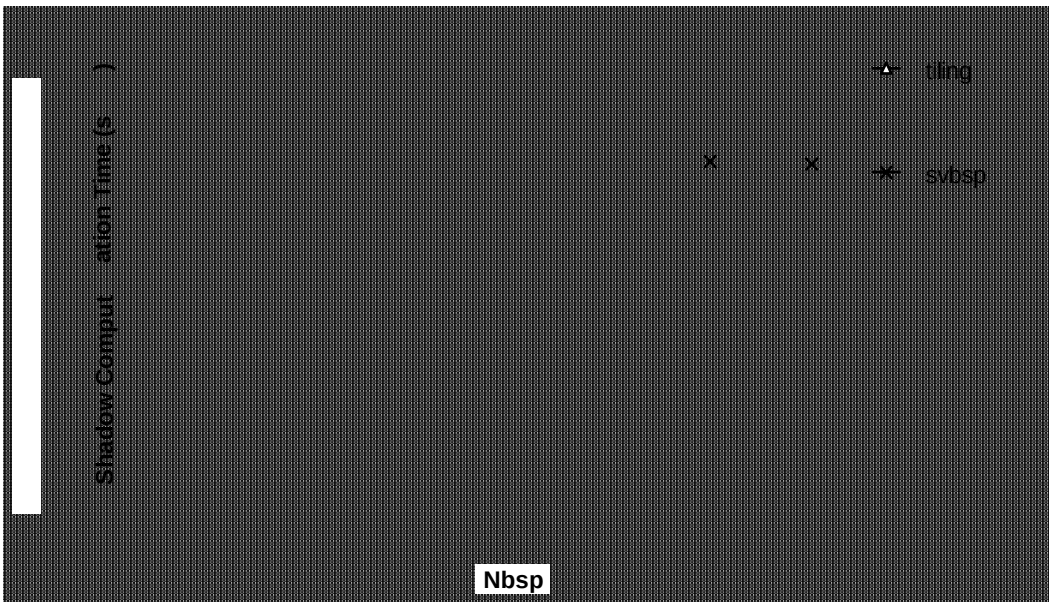

Figure 12
SUN 3 Timings for the Room Scenes with Two Lights
(Lines are Linear Interpolations)

8. Conclusions

The results discussed in the previous section show no decisive advantage for either the SVBSP or tiling algorithm in comparison to each other with respect to execution time. The timing results cannot be generalised beyond the range of scenes considered, and vary with machine architecture. In any case, even in the with the results on the SparcStation, the times are at most approximately 2 seconds difference so that this alone would not determine the choice of which of these two algorithms to use.

The SVBSP has an advantage in the fact that it never has to transform the scene polygons. In the case of the tiling algorithm the scene polygons have to be transformed to the tiling display space - which involves going through the entire

viewing pipeline, including clipping. Overall, the SVBSP algorithm is a more elegant approach than the tiling.

It is possible that in interactive walkthrough the SVBSP might be slightly faster[2]. This is shown by Figure 3, where a single shadow generating polygon might generate up to 8 polygons one on top of another in the case of the ST algorithm for three light sources. The SVBSP tree approach generates a single covering of the display space. On the other hand, this is not an intrinsic property of the ST approach, since it would be possible to structure the algorithm also to provide a single covering of the display image, by only keeping those fragments of the original polygons which were not in shadow rather than painting the shadow polygons always on top of the original. Another drawback of this overpainting approach is that standard z-buffer hardware cannot be used for rendering, since the original polygons and their shadows are co-planar, and could lead to a "streaking" effect.

There is an application area where the tiling approach may be preferable to the SVBSP. When incremental changes are made to a scene (for example, translating or rotating an object), a frequently used approach is to recompute and redisplay the entire scene. However, incremental changes can be expressed as sequences of polygon deletions and additions. Some progress has been made [21][30][31] in dynamically changing BSP trees for this purpose, but such work does not include a consideration of shadows. Recent work[3] shows that the ST approach can be efficiently used for dynamic modifications of objects in scenes represented as BSP trees, including correct changes to shadows. We have found that SVBSP trees can also be dynamically modified, but at the time of writing it is not clear as to which approach has the advantage in this case.

Acknowledgements

This work has benefited from discussions with Allan Davison, Kieron Drake and Eliot Miranda. The author is indebted to the referees for their insights and suggestions. Thanks to T. Lin for commenting on an earlier draft. The work is a contribution to the ESPRIT funded project, The SPIRIT High Performance Technical Workstation.

References

[1] Appel, A. (1968) *Some Techniques for Shading Machine Renderings of Solids*, **Proc. AFIPS JSCC 1968**, 32,, 37-45.

[2] Atherton, P.R., Weler, K. and Greenberg, D. (1978) *Polygon Shadow Generation* **Computer Graphics** 12, 275-281.

[2]This was pointed out by a referee.

[3] G. Chrysanthou, M. Slater (1991) *Computing Dynamic Changes to BSP Trees*, submitted for publication.

[3] Bergeron, P., (1986) *A General Version of Crow's Shadow Volumes*, **IEEE CG&A**, 6(9), 17-28.

[4] Bouknight, W.J., Kelley, K. (1970) *An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources*, **AFIPS Conf. Proc.** 36, 1-10.

[5] Brooks, F.P. Jr (1987) *Walkthrough - A Dynamic System for Simulating Virtual Buildings*, **Workshop on Interactive 3D Graphics, Proceedings**, New York, Association of Computing Machinery.

[6] Chin, N., Feiner, S. (1989) *Near Real-Time Shadow Generation Using BSP Trees*, **Computer Graphics** 23(3), 99-106.

[7] Cleary, J.G., Wyvill, G. (1988) *Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision*, **The Visual Computer**, 4, 65-83.

[8] Cohen, M.F., Greenberg, D.P. (1985) *The Hemi-Cube: A Radiosity Solution for Complex Environments***, Computer Graphics** 19(3), 31-40.

[9] Cohen, M.F., Shenchang, Ec., Wallace, J.R. and Greenberg, D.P. (1988) *A Progressive Refinement Approach to Fast Radiosity Image Generation***, Computer Graphics** 22(4), 75-84.

[10] Crow, F. (1977) *Shadow Algorithms for Computer Graphics*, **Computer Graphics** 11(2) 242-247.

[11] Encarnacao, J., and Giloi W. (1972) *PRADIS - an Advanced Programming System for 3-D-display*, **AFIPS Conference Proceedings**, 40, 1972 Spring Joint Computer Conference.

[12] Fuchs, H. Abram G.D. and Grant, E.D. (1983) *Near Real-Time Shaded Display of Rigid Objects*, **Computer Graphics** 17(3), 65-72.

[13] Fuchs, H., Kedem, Z.M., and Naylor, B.F. (1980), *On Visible Surface Generation by A Priori Tree Structures*, **Computer Graphics** 14(3), 124-133.

[14] Fujimoto, A., Tanaka, T., Iwata, K. (1986) *ARTS: Accelerated Ray-Tracing System*, **IEEE CG&A**, 6(4), 16-26.

[15] Glassner, A.S. (1984) *Space Subdivision for Fast Ray Tracing*, **IEEE CG&A**, 4(10), 15-22.

[16] Goral, C., Torrance, K.E., Greenberg, D. (1984) *Modeling the Interactionof Light Between Diffuse Surfaces*, **Computer Graphics**, 18(3), 213-222.
[17] Gordon, D., Chen, S. (1991) *Front-toBack Display of BSP Trees*, **IEEE CG&A**, 11(5), 79-85.

[18] Haines, E.A., *A Proposal for Standard Graphics Environments*, **IEEE CG&A**, 7(11), 3-5.

[19] Haines, E.A., Greenberg, D.P. (1986) *The Light Buffer: A Shadow Testing Accelerator*, **IEEE CG&A** 6(9), 6-16.

[20] Meyer, Urs (1990) *Hemi-Cube Ray-Tracing: A Method for Generating Soft Shadows*, **Eurographics 90**, C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland, 365-376.

[21] Naylor, B. (1990) *SCULPT: An Interactive Solid Modeling Tool*, **Graphics Interface 90**, Morgan-Kaufmann Publishers, 138-155

[22] Newell, M.E., Newell, R.G., and Sancha, T.L. (1972) *A New Approach to the Shaded Picture Problem*, **Proc. ACM Nat. Conf**, 443-450.

[23] Plummer, M. and Penna, D. (1989) *Mass Market Applications for Real Time 3D Graphics*, **Computer Graphics Forum** 8(2) 143-150.

[24] Poulin, P. and Amantides, J. (1990) *Shading and Shadowing with Linear Light Sources*, **Eurographics 90**, C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland, 377-386.

[25] Schumucker, R., Brand, B., Gilliland, M., Sharp, W., (1969) *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.

[26] Shapiro Brotman, L. and Badler, N.I. (1984) Generating Soft Shadows with a Depth Buffer Algorithm, **IEEE CG&A**, 4(10), 5-38.

[27] Slater, M., Davison A., Smith M. (1988/89) *Liberation from Rectangles: A Tiling Method for Dynamic Modification of Objects on Raster Displays*, **Eurographics 88**, D.A. Duce and P. Jancene eds, pp.381-392 (North-Holland). Republished in **Computers and Graphics** (1989) 13(1) pp. 83-89.

[28] Stallman, R.M. (1989) *Using and Porting GCC*, version 1.37.1, Free Software Foundation, 675 Mass Ave, Cambridge MA 02139, USA.

[29] Sutherland, I.E. and Hodgman G.W. (1974) *Reentrant Polygon Clipping*, **Communications of the ACM** 17(1), 32-42.

[30] Thibault, W.C., and Naylor, B.F. (1987) *Set Operations on Polyhedra Using Binary Space Partition Trees*, **Computer Graphics** 21(4) 153-162.

[31] Torres, E. (1990) *Optimization of the Binary Space Partition Algorithm (BSP) For the Visualization of Dynamic Scenes*, **Eurographics 90**, C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland, 507-518.

[32] Whitted, T. (1980) *An Improved Illumination Model for Shaded Display*, **Comm. ACM**, 23(6), 343-349.

[33] Williams, L. (1978) *Casting Curved Shadows on Curved Surfaces*, **Computer Graphics** 12, 270-274

[34] Woo, A., Poulin, P., Fourier, A. (1990) *A Survey of Shadow Algorithms*, **IEEE CG&A**, 10(6), 13-31.

Plate 1

A view of a the test scene with two light sources. One light source is outside of a door, and the other at the centre of the ceiling. The large shadow on the wall is caused by the light passing through the door.

Plate 2

A different view of the same scene is shown.

Plate 3

The test scene has had more items added to it, and a different view is shown.

Plate 4

The recursive tetrahedron (tetra256) from the Haines data base is shown, with shadows caste on itself and another polygon, using a single light source.