Real-Time Shadow Generation Using BSP Trees and Stencil Buffers

HARLEN COSTA BATAGELO
ILAIM COSTA JÚNIOR

UNOESC – Universidade do Oeste de Santa Catarina
Campus de Videira
Rua Paese, 198, Bairro das Torres, 89560-000 Videira, SC, Brasil
{harlen, ilaim}@unoescvda.rct-sc.br

Abstract. This paper describes a real-time shadow generation algorithm for static polygonal environments illuminated by movable point light sources. The algorithm combines a technique of volumetric shadow rendering using stencil buffers with a Binary Space Partitioning (BSP) tree, and includes new easy-to-implement approaches to improvement techniques used in shadow volume algorithms, such as silhouette detection to reduce the number of redundant shadow polygons and the computation of capping polygons to handle cases where the shadow volumes are clipped by the eye-view near clipping plane. Such hybrid approach solves important limitations on the original shadow rendering algorithm, as well as achieves real-time frame rates when using modest size scenes (about 500 shadow polygons), according to measurements performed on personal computers using current graphics hardware. Per-phase timing results from the implementation are provided along the text and compared with those of the standard algorithm.

1 Introduction

The calculation of shadows has been a classical problem in 3D computer graphics. The presence of shadows provides to the viewer a better understanding of the spatial relationship among the objects, increasing the reality sense and coherence given by the image [23]. Unfortunately, there are difficulties on creating real-time shadow generation techniques for interactive applications, such as games and virtual reality, due to the high computational costs involved, especially when dealing with the current generation of personal computers.

Due to the recent popularization of 3D acceleration cards with stenciling capabilities (i.e., stencil buffers support) biased towards the personal computers market [1,2,18], techniques of volumetric shadow rendering using stencil buffers seem to be a feasible alternative to the real-time shadow generation problem [12,14], and have started to be used by a number of computer gaming industries [9,13]. On the other hand, this approach comes out with some important limitations, in special the fact that its efficiency is strongly dependent on the complexity of the shadowing objects. To minimize such problem, the number of shadow polygons must be kept at a minimum. A common solution [7] involves casting shadows only along the silhouette of the objects visible with respect to the light source position. Unfortunately, current approaches to these optimizations are either limited to convex polyhedrons [5,8] or to a particular field of view [14]. Our algorithm does not have these problems. We divide the optimization process in two independent tasks: hidden surface removal and silhouette edge identification. For each frame, we build an incremental BSP tree that discard any hidden shadow generator polygon through union boolean set operations. The remaining visible polygons are processed by a simplified approach to silhouette detection, thus producing a nearly optimal shadow volume that represents the union of all shadow volumes of the scene.

The standard shadow rendering algorithm also suffers from parity errors of the stencil buffer that happen whenever shadow volumes are clipped by the camera near plane, and which result in inverted shadows on the final image. Commonly the proposed solutions does not apply to the general case [7,17] or has difficult implementations when assuming the case of computing capping polygons to close clipped shadow volumes. In this work we include a new approach to this capping process, of easy implementation but also very efficient.

According to measurements performed on personal computers using current 3D graphics hardware, real-time frame rates (10-60 Hz) can be obtained when using scenes of modest size (up to about 500 shadow polygons per frame, though this obviously depend on hardware capabilities). The timing tests have also showed significant gains of efficiency over the standard algorithm.

The rest of the paper is organized as follows. Section 2 contains a brief taxonomy about similar shadow generation algorithms. The section 3 introduces the idea of BSP trees. The definition of stencil buffers, as well as their applications in shadow rendering, is given in section 4. The sections 5,6 and 7 describe the hybrid algorithm. The timing test results and conclusions are given in the sections 8, 9 and 10.

2 Shadow Algorithms

The shadow generation approaches were first surveyed by Franklin Crow [7], and after extended by Bergeron [3] and Woo et al. [26].

The classified algorithms used in interactive applications can be distinguished in three broad groups: projective shadow s algorithms, shadow map algorithms and shadow volume algorithms. We discuss briefly each one.

2.1 Projective Shadows

Projective shadow techniques are object-space precision algorithms that use projection transforms to simulate shadows in polygonal scenes. The algorithm works by projecting every polygon of the scene onto the plane of the other polygons, using an orthographic or perspective projection matrix to simulate, respectively, directional or point light sources. Each projected polygon is clipped by the target surface, thus producing a new polygon that can be rendered with the shadow color. The same principle applies to area light sources, where the shadows are given by *discontinuity meshes*. In the most fundamental level, the entire scene can be just compressed and projected on a ground plane, then rendered as a separate primitive with the shadow color [4].

To achieve real-time frame rates it is difficult to use projection shadows algorithms to shadow onto anything other than flat surfaces. Ideally, it is necessary graphic hardwares with capabilities to clip each projected polygon to the boundary of the target polygons, which is a process subjected to numerical robustness problems. Such hardware is also not common on PC computers.

2.2 Shadow Volumes

Shadow volume algorithms use polygonal volumes to represent enclosed shadow regions in the scene.

A *shadow volume* is a polyhedron formed by semi-infinite quadrilaterals named *shadow polygons*, which are projected for each edge of each shadow generator polygon. The two finite vertices of a shadow polygon correspond to the edge endpoints of the shadow generator polygon; the two infinite vertices point at the direction of the projected shadow, being limited by the view volume boundary in homogeneous coordinates. Figure 1 illustrates a shadow volume in 3D, as seen by the position of the camera.

The shadow polygons are defined so that their normals point outward its shadow volume (or inward, since all other shadow polygons follow the same rule). The regions in shadow can be determined by counting the number of front and back facing shadow polygons as seen by the camera position. For instance, if a ray from outside the shadow

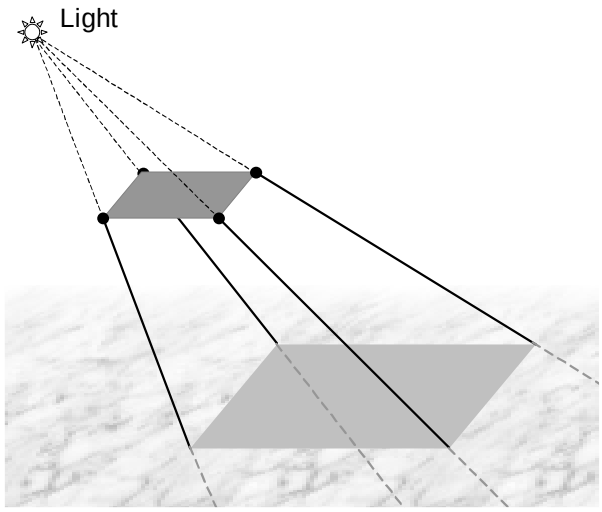


Figure 1: Shadow volume.

volumes to a point on a surface intercepts a different number of front facing or back facing shadow polygons, the point is in shadow.

In general, the efficiency of shadow volume algorithms is strongly dependent on the complexity of the shadowing objects. Ideally, the shadow volume should be generated only from the vertices along the silhouette of visible objects, as seen from the light source. The adjacent (non-silhouette) edges can be discarded since the shadow polygons generated from them will be coincident and will have opposed orientation, thus canceling the shadow polygon counting.

Shadow volumes can be used in conjunction with a number of algorithms, such as scanline, depth buffers algorithms [5] and BSP Trees [6,21]. In the pixel planes architecture [10], shadow volumes do not need explicit calculations of their shadow polygons. In addtion, the performance is not affected by the size of the shadows as commonly occurs, including our algorithm. Fortunately, this dependency is not significant by means of the fast fill rate provided by the current generation of graphic hardware.

2.3 Shadow Maps

Introduced by Williams [25], shadow map algorithms work at image-space precision, thus not being limited to polygonal models as occurs with projective and shadow volume algorithms. Its performance is stable, in a sense that it does not depend directly on the complexity of the shadowing objects. Moreover, it can be easily adapted to achieve soft shadows, simulating penumbra.

Shadow map algorithms work by updating depth buffers of the scene rendered from the point of view of the light sources. These depth buffers are actually the *shadow maps*,

and determine which pixels are in shadow by comparing if its values are smaller than the corresponding depth values of the pixels rendered from the point of view of the camera.

Shadow map algorithms can be stored as texture maps when using hardware with support to texture mapping and texture mapping comparison [20]. Using a texture coordinate transformation matrix, the shadow maps are mapped onto the original scene as viewed from the camera, then comparing its texels with the depth values of the corresponding scene pixels. The alpha value of each pixel is changed according to the result of the comparison.

Since shadow maps are essentially depth buffers, the quality of the shadows becomes directly proportional to the resolution of the map. Preciseness is also important to avoid aliasing artifacts and self-shadowing. In addition, in case of hardware implementations, texture management schemes should be used to prevent cache slowdown when using many maps.

3 BSP Trees

The BSP (Binary Space Partitioning) tree algorithm provides an elegant and efficient way to determine the visibility order of static groups of polygons with respect to an arbitrary viewpoint. It was introduced by Fuchs, Kedem and Naylor [11], based on Schumacker's work [19], and represent spatial relationships through recursive partitionings of n-dimensional spaces by hyperplanes. The tree is often built at an intensive pre-processing stage, choosing a polygon – in the 3D case – to act as the tree's root and partition the space in two sets. The remaining polygons are classified with respect to the normal of this root polygon, and stored as "front" or "back" set ramifications of this node. Polygons lying on both sides are split by the root and have their pieces assigned to the corresponding sets. Each set chooses a new partition plane, which becomes the children of the root. The half-spaces are divided in the same way, recursively, until each set contains no more polygons. The resulting structure, in binary tree form, can be traversed in a special in-order manner, establishing in linear time the back-to-front (or vice-versa) order of facing polygons as seen from any viewpoint.

Thibault and Naylor [22] showed that a BSP tree could be used to represent an arbitrary polyhedral solid. In their work, each interior node contains a polyhedron's plane which embeds the set of polygons stored at this node. At the leaves are "in" and "out" cells indicating interior or exterior regions of the solid. Based on this principle, they show how to represent complex solids based on boolean set operations performed on other BSP trees and boundary representations of solids.

Relying on the work of Thibault and Naylor, Chin and Feiner [6] introduced a new algorithm, called SVBSP (*shadow volume BSP*) tree, which efficiently represents union of shadow volumes produced by traversing the original BSP tree from each light source position in a front-to-back order. The shadow planes of each shadow generator polygon are filtered down the tree, which initially contains a single "out" cell. At the leaves, the "in" and "out" cells indicate shadowed or lit regions, so shadow polygons contained in "in" regions could be discarded. On the contrary, the tree is enlarged by the "lit" polygons. As in the original BSP tree algorithm, shadow polygons with parts contained in "out" and "in" regions are split by the current shadow plane, but the shadowed fragments are discarded. After processing all shadow volumes, the tree represents a complex volume formed by the union of every shadow volume created from shadow generator polygons.

4 Stencil Buffers

A stencil buffer is an extra plane of bits in the frame buffer, used to enable or disable drawing to the rendering surface on a per-pixel basis. In an analogy to the stencil paper, the stencil buffer can be used to mask off regions of the color buffer so that it is not displayed.

The primitives are applied to the stencil buffer as we normally draw onto the color buffer, but instead of storing RGBA values, it stores only integer values within the range of 0 to 2^n-1 (inclusive), where n is the number of stencil bits. These values are assigned with respect to simple arithmetic operations, which are executed after passing through the evaluation of a condition between the current value of the stencil buffer pixel and a reference value defined by the user. Table 1 shows the main operations and conditions used.

Comparison Functions (same as a depth buffer):
<; ≤; >; ≥; =; ≠; always; never.

Operations to be applied to the current stencil value:
Set to zero; replace by reference value; increment; decrement; bitwise invert; keep.

Table 1: Stencil buffer operations and comparison functions (available under OpenGL and Direct3D).

The stencil buffer test is always executed before the depth buffer test, so it can be used to affect only the visible parts of the rendered scene. In fact, current hardware implementations store the stencil buffer and the z-buffer in the same memory chunk, thus yielding no extra penalty for

stencil test when already using depth test. The complete stencil test executed for each pixel is showed below in pseudo-code:

```
If  (ref AND mask) condition (stencil AND mask)
      If depth test passes
            Op1(stencil AND write_mask);
      Else
            Op2(stencil AND write_mask);
Else
      Op3(stencil AND write_mask);
```

where *ref* is the reference value, *stencil* is the current value of the stencil pixel and *mask*/*write_mask* are bit masks used to bitwise "AND" the reference value and the stencil buffer value before the evaluation of the condition, or to bitwise "AND" the current stencil value before the execution of the defined operation. Such operation depends on the test result, that can be: stencil test failed, stencil test passed and depth buffer failed, stencil test passed and depth buffer passed. All variables, except the current value of the stencil pixel, are defined by the user.

When the stencil buffer is ready, it can be used to mask off the drawing onto the rendering surface. For instance, allowing draw pixels only if the corresponding stencil buffer pixels are equal to a given value.

4.1 Volumetric Shadow Rendering

Stencil buffers can be used in conjunction with depth buffers in order to maintain the shadow counting/parity in image-space, marking the transition between front and back facing shadow polygons visible from the camera viewpoint [12,14].

Firstly, the original scene – without shadows – is rendered onto the color buffer, using the depth buffer for testing and writing. The shadow polygons are rendered onto the stencil buffer, this time not writing to the depth buffer, but still using the depth test. The pixels that pass the depth test modify the shadow counter for that pixels, increment ing these values if the current shadow polygon is facing the camera and decrementing otherwise. Consequently, at the final of this process the regions of the stencil buffer with non-zero pixels will correspond to shadowed pixels on the original image. This is sufficient to use the stencil buffer as a mask over the rendering surface and dark the appropriate pixels.

The complete algorithm is summarized in the following steps:

1. Enable the depth buffer for testing and writing.
2. Render the original scene onto the color buffer.
3. Clear the stencil buffer and disable the depth buffer for writing.
4. Draw onto the stencil buffer the shadow polygons that are facing the viewer, using the increment stencil operation.
5. Draw onto the stencil buffer the shadow polygons that are facing away the viewer, using the decrement stencil operation.
6. Dark the pixels of the color buffer using the stencil buffer to mask off regions that correspond to zero stencil buffer pixels.

Since this algorithm is based on shadow volumes, its efficiency depends directly on the complexity of the shadow casting objects, although the target geometry does not affect performance. Thus, this algorithm is best used when the shadow casting objects are simple but the shadowed objects are complex, possibly non-polygonal.

Under OpenGL, the default decrement and increment operation saturates the stencil buffer values to a minimum of zero and a maximum of 2^n-1. To avoid shadow counting underflow, the front facing shadow polygons must be rendered before the back facing shadow polygons. In the special case of 1-bit stencil buffers, shadow volumes must be rendered in a front-to-back order (Direct3D can avoid this problem since it includes options to wrap values to 0 and 2^n-1). On the other hand, shadows may be still rendered incorrectly, as can be noticed in certain pathological cases of overlapped shadow volumes, even using stencil buffers with more than 1-bit. A common solution is to choose a "non-shadow" value different from the standard "zero", preferable in the middle of the stencil values range (e.g., 2^{n-1}). The algorithm is modified to initialize and test for this value instead of zero.

4.2 Shadow Rendering Modes

Once the stencil buffer is ready, all pixels on the color buffer that have corresponding stencil values greater than zero should be modified by the shadow color. This stage can be done by applying a large polygon that covers the entire viewport, using the stencil buffer as mask. A black polygon is enough when using a zero ambient light term and only one light source. Otherwise, translucent polygons with the ambient color should be used. Translucency can also be used to make composite shadows when using more than one light source. However, the translucency level (alpha component in a RGBA surface) is chosen empirically, which can result in inconsistent illumination. This can be noticed, for example, if an object that reflects specular light is in shadow. A common solution consists in simply maintain a low specular term [5]

or render the specular part separately [16]. This last one, using alpha blending to combine the shadows produced by each light source, can be done as follows:

1. Render the original scene onto the color buffer using only ambient illumination and update the depth buffer.
2. For each light source:
   - Render the shadow volumes generated from the current light source as described by the standard algorithm (section 4.1, passes 3--5).
   - Enable alpha blending with source and destination blend factors to one:one (*addition* mode).
   - To update only the visible pixels, set the depth test comparison function to "equal" and render onto the color buffer the original scene illuminated only by the current light source where the stencil values are zero.

5 Hybrid Algorithm

Our hybrid algorithm combines the techniques described in sections 3 and 4. To remove hidden shadow generator polygons, we build, for each frame, an incremental SVBSP tree as described lately, but with some differences. To improve efficiency, we do not attempt to keep a perfect CSG union of shadow volumes, which commonly needs to remove nodes already added to the tree. We also do not clip the original shadow polygons with fragments in shadowed regions, since we noticed that this process, done in software, is slower than the equivalent polygon rendering cost by hardware. Moreover, such polygons can be lately removed efficiently with our silhouette detection algorithm.

The complete hybrid algorithm is summarized in the following steps:

Pre-processing:

- Build the BSP tree of the scene.

For each frame:

1. Determine shadow casters: Traverse the tree with respect to the position of the light source, obtaining the set of visible polygons in a front-to-back order. This is necessary to build the SVBSP tree later.
2. Create shadow volumes: Generate a shadow volume for each shadow casting polygon.
3. Build SVBSP tree: Create the SVBSP tree and add each shadow volume generated from a shadow casting polygon in a front-to-back order. If an entire shadow volume is under an "in" cell, it is not added to the tree and its shadow casting polygon is discarded.
4. Identify silhouette edges: From the set of visible shadow casting polygons contained in the SVBSP tree, identify which edges are not shared by two polygons of same orientation. These are stated as silhouette edges.
5. Render shadows: Use the technique described in the standard algorithm (section 4.1, steps 1--5) to render onto the stencil buffer only the shadow polygons generated from silhouette edges.
6. Clip shadow volumes: Clip each shadow volume and its corresponding shadow casting surface with respect to the near clipping plane from the viewing frustum. This includes shadow volumes formed by non-silhouette edges.
7. Cap shadow volumes: For each clipped shadow volume, sort the vertices generated from the clipping stage in a polar order, then build a cap polygon. Draw them onto the stencil buffer as a usual shadow polygon that faces away the camera.
8. Dark the appropriate pixels: Dark the pixels of the color buffer that correspond to non-zero stencil buffer pixels (section 4.1, step 6).

The algorithm restricts the representation of the scene polygons as indices from an array of vertices. This is necessary to preserve the primitive adjacency information, used by the silhouette detection algorithm. Fortunately, that is the standard scene format used by most 3D modeler programs (e.g., 3D-Studio, LightWave 3D). Another disadvantage, inherited from the BSP tree, is its limitation to static scenes. Dynamic objects can be handled as set of polygons that are not part of the BSP tree and possibly not taken in account by the SVBSP tree.

In applications where the viewing frustum never clips away the shadow volumes by its near plane (in general, applications with isometric/third person perspectives), there is no need of the capping stage. Therefore, steps 6 and 7 previously described can be discarded.

Multiple light sources are handled in a similar with the standard algorithm. An alternative is to use accumulation buffers (available under OpenGL) instead translucency, in order to achieve accurate color precision. Our algorithm adapted to this method is described as follows:

For each frame:

1. Render the scene without shadows onto the color buffer using only ambient illumination and load it into the accumulation buffer.
2. For each light source:
   - Enable depth buffer for testing and writing.
   - Render onto the color buffer the scene without

shadows and illuminated only by the current light source.

- Determine shadow casters.
- Create shadow volumes.
- Build SVBSP tree.
- Identify silhouette edges.
- Use the standard algorithm (section 4.1, passes 3--5) to render onto the stencil buffer only the shadow polygons generated from silhouette edges, with respect to the current light source.
- Clip shadow volumes.
- Cap shadow volumes.
- Dark the appropriate pixels.
- Accumulate the color buffer onto the accumulation buffer.

3. Turn back the accumulation buffer to the color buffer (no scaling is needed).

Accumulation buffers can also be used to simulate anti-aliased or soft shadows. This can be done by rendering the shadow volumes multiple times onto the accumulation buffer, each one with a light source positioned in a difference place along the surface of the area light source. Unfortunately, to obtain good results it is necessary to jitter a great number of point light sources, which cause considerable slowdown on the algorithm.

6 Silhouette Detection

With respect to the set of polygons that are facing the light source (i.e., the shadow casting surfaces), we define by silhouette the edges of these polygons that are not shared by other polygon of identical orientation.

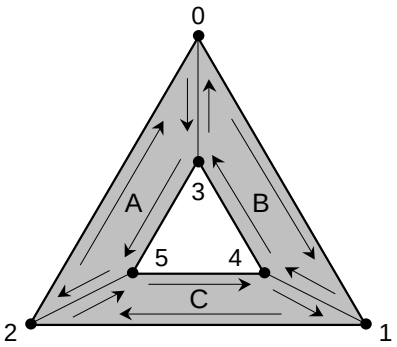
Our silhouette detection approach works only on the topology of polygons meshes, based on similar structures used in graph adjacency algorithms such as adjacency matrixes and edge lists. It works incrementally, storing the indices of the edge endpoints of each polygon in lists that describe the connected vertices according to the orientation of the current polygon.

The algorithm requires, as input, a set of polygons represented as indexes from an array of vertices. As output, returns an array of lists containing the identified edges. Each list will contains the indexes that, altogether with the index corresponding to the number of the list, form the silhouette edges. Figure 2 contains the pseudo-code of the described algorithm. Figure 3 illustrates an example.

An important drawback of our algorithm is that the index-based polygon representation also considers as silhouette those edges formed by T-vertices. Such problem can be significant, since this kind of edge frequently arises during the BSP tree construction when the original polygons are split by partition planes. Fortunately, this can be corrected still during the pre-processing stage.

```
Find_Silhouette(polygon set P, list set L)
{
    for each polygon p of P
        for each side n of p
        {
            i = first vertex of n
            j = second vertex of n
            if L[ j ] contains i
                delete i from L[ j ]
            else
                add j to L[ i ]
        }
    returns L
}
```

Figure 2: Pseudo-code of the silhouette detection algorithm.



Input polygons (clockwise order):
A=0,3,5,2 B=0,1,4,3 C=4,1,2,5

Output:

Vertex	Elements	
0	3	1
1	4	2
2	0	
3	5	
4	3	
5	2	4

Figure 3: Silhouette edge identification example. The grayed cells contain the indexes of the vertices that form non-silhouette edges.

6.1 Efficiency

The time and space complexity of the described silhouette detection algorithm is bounded above by $O(n^2)$, for n input polygons of n vertices. In most cases, the time complexity is $O(n)$ for n polygons.

There is a trade-off between efficiency and memory consuming. Since each list is constructed incrementally, it is not initially known the number of elements required by each one. Linked lists are not recommended, since the efficiency should decays due to the high number of dynamic memory accesses. Static lists does not have this problem, though it is necessary to pre-allocate n elements for each list, which increases the space complexity to $O(n^3)$, assuming n polygons of n vertices. We suggest the pre-allocation of a low number of cells on each static list that correspond to the maximum expected number of edges sharing a single vertex.

7 Capping

Since shadow volumes are polygonal objects added to the scene and hence pass through the geometry pipeline, it is common that some parts of them are clipped away by the camera's viewing frustum. This clipping stage can discard fragments of shadow volumes that contains important shadow counting information, thus producing shadow counting/parity loses on the stencil buffer that turns in grossly inverted shadows on the final image. The standard solution [7,17] of inverting the parity globally whenever the viewpoint is into a shadow volume does not work in the general case. In most cases these parity loses happens locally (i.e., only in some portions of the viewport), even if the viewpoint is not actually in shadow. In general, the inversion happens whenever there are shadow volumes lying partly or wholly in front of the camera's near clipping plane. Another solution resides in computing capping polygons to close the shadow volume of the scene. A cap polygon is a polygon created from the intersection of a shadow volume with a frustum plane. They are handled as any other shadow polygons, with the normal pointing outside the shadow volume. This capping process may be very difficulty, since the shadow volume of the scene is often a complex polyhedron, concave and with holes.

Our easy-implementation approach to capping consists simply in clipping *every* shadow volume with respect to the eye-view near plane, without any silhouette optimization, i.e., also including shadow volumes formed by shadow polygons generated from non-silhouette edges. This process is easily done converting the plane equation of the near plane to world coordinates, then clipping each volume separately. The generated vertices of each volume are sorted in a polar order and turned into a cap polygon. When the standard algorithm have finished the shadow volume rendering, the cap polygons are rendered as separate shadow polygons that are facing the camera, thus correcting the parity inversion either locally or globally.

8 Implementation

The algorithm was implemented in C++ using the Direct3D 6 API. The timing tests were performed using a PC Pentium Celeron 466Mhz, equipped with a 3D acceleration card (chipset Riva TNT). Figures 4--8 show the five scenes tested. The two scenes of tetrahedrons (figure 6 and 7) are actually the same, but with the "base polygon" positioned in a different vertical coordinate. All scenes were rendered with a resolution of 640x480 pixels, 32 bits of RGBA colors, Gouraud shading and a stencil buffer of 8-bits.

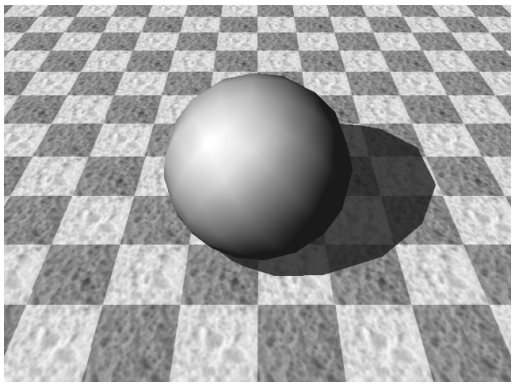


Figure 4: Sphere – 256 polygons

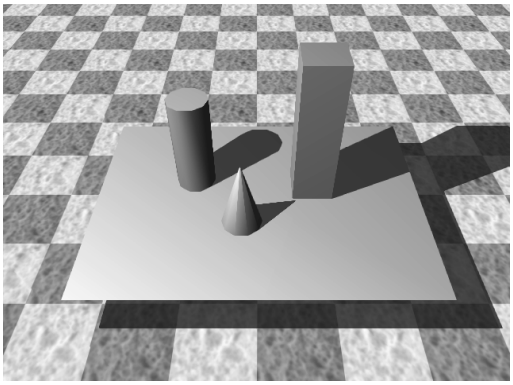


Figure 5: Solids – 96 polygons

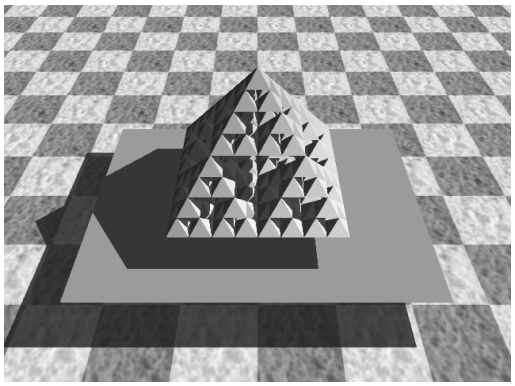


Figure 6: Tetrahedron #1 – 752 polygons

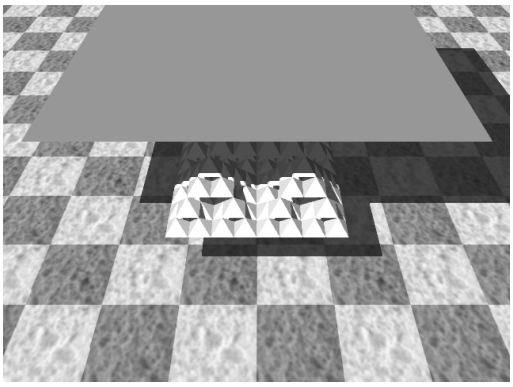


Figure 7: Tetrahedron #2 – 752 polygons

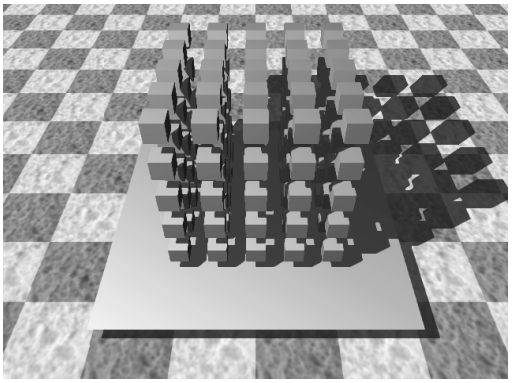


Figure 8: Cubes – 1502 polygons

The timing tests were computed over ten executions of each scene, each execution with a single point light source positioned in a distinct place.

The BSP tree pre-processor was configured to achieve the less possible number of polygon splits, though we observed that different configurations did not influence considerably the efficiency of the shadow generation.

9 Results

Per-phase timing results from the hybrid algorithm are given in table 2. Timing results from the standard shadow volume algorithm (as described in section 4.1), are given in table 3. All units are milliseconds, except FPS.

Figure 9 helps to visualize the gain of efficiency provided by the hybrid algorithm. Unit are frames per second, limited to a 60Hz refresh rate.

Figure 10 shows the reduction of the number of shadow polygons provided by the SVBSP tree in conjunction with the silhouette detection algorithm. Finally, figure 11 shows the shadow rendering time against the cost of the SVBSP construction (both in milliseconds).

	Sphere	Solids	Tetra #1	Tetra #2	Cubes
A	1,75	0,40	3,40	3,25	6,30
B	0,10	> 0,00	0,30	0,30	1,20
C	0,35	0,15	2,05	1,35	3,90
D	2,40	0,55	8,50	2,15	16,70
E	0,40	0,35	0,70	0,35	1,10
F	0,15	0,05	0,75	0,15	1,70
G	0,60	0,75	12,75	2,20	18,00
Total	5,75	2,25	28,45	9,75	48,90
FPS	55,1	57,7	14,9	34,8	9,9

	Phase Description
A	Eye-view rendering
B	Front-to-back sorting of shadow casting surfaces
C	Shadow volume generation
D	SVBSP construction
E	Silhouette detection
F	Shadow volume clipping and capping
G	Shadow rendering

Table 2: Per-phase timing results from the hybrid algorithm.

	Sphere	Solids	Tetra #1	Tetra #2	Cubes
A	1,75	0,60	5,30	5,40	10,35
B	0,05	> 0,00	0,25	0,20	0,60
C	0,65	0,30	3,10	3,15	6,55
D	6,15	3,20	43,85	48,80	148,50
Total	8,60	4,10	52,50	57,55	166,00
FPS	23,9	34,3	9,8	8,8	5,5

	Phase Description
A	Eye-view rendering
B	Shadow casting surfaces determination
C	Shadow volume generation
D	Shadow rendering

Table 3: Per-phase timing results from the standard algorithm.

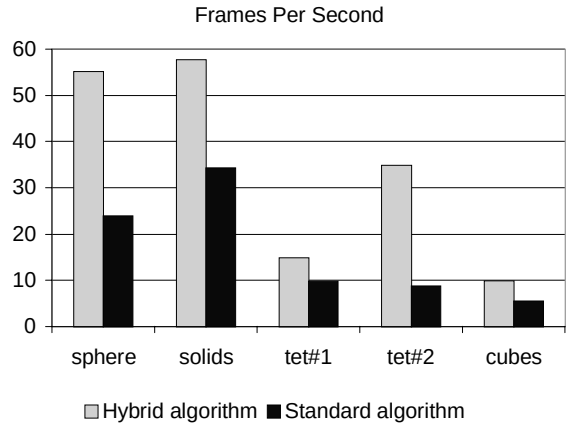


Figure 9: Visualization of FPS measurements in table 2 and 3.

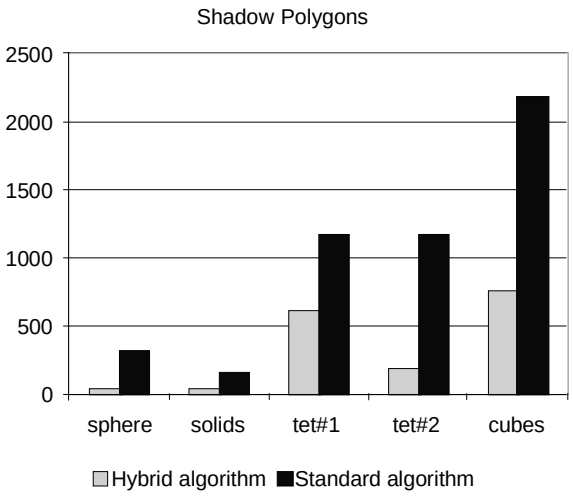


Figure 10: Average number of shadow polygons produced in each frame.

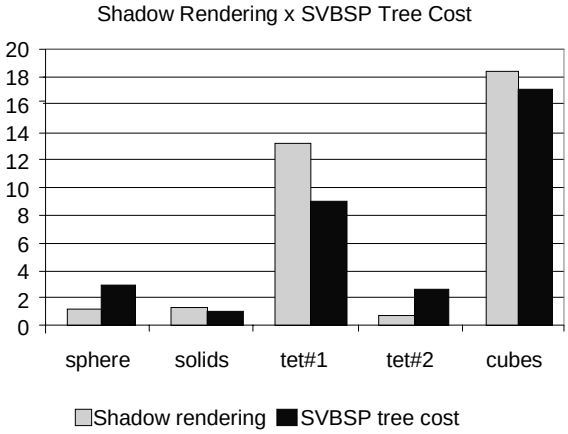


Figure 11: Shadow rendering time against SVBSP tree construction time (ms).

We can notice the benefit of the use of SVBSP trees when comparing the results of the scenes of tetrahedrons (figure 6 and 7) with the results of the standard algorithm. On the other hand, the scene containing the cubes formation (figure 8) does not benefits from the SVBSP tree, and has its performance decreased. In fact, it is desirable to use the SVBSP tree only is scenes or objects where shadow volumes are potentially ocluded by other volumes, so the cost of tree construction does not affect efficiency substantially.

10 Conclusion

We have presented a hybrid algorithm of real-time shadow generation, based on a technique of shadow volume rendering using stencil buffers and benefiting from BSP trees to perform hidden surface removal through boolean set operations.

As observed in experimental tests, the SVBSP tree predominantly improves performance on scenes with potentially many hidden shadow volumes. Otherwise, it should not be used. Since the shadow rendering is the most time consuming task, optimizations should be aimed to reduce the number of shadow polygons. We suggest the use of LOD (level-of-detail) management algorithms for complex scenes, which could also reduce the cost of the SVBSP tree construction. The algorithm can also be extended to handle large scenes by using portals [15] or cells algorithms in conjunction with BSP trees.

Acknowledgements

We gratefully acknowledge Kade Criddle, who promptly provided us resources for performing the timing tests, and the SIBGRAPI referees for their helpful comments and sugestions.

References

[1] 3dfx Interactive, Inc., http://www.3dfx.com.
[2] ATI Technologies Inc., http://www.atitech.com.
[3] Bergeron, P. (1986) “A General Version of Crow’s Shadow Volumes”, *IEEE CG&A*, 6(9), 17-28.
[4] Blinn, J. (1988) “Me and My (fake) Shadow”, *IEEE CG&A*, 8(1), 82-86.
[5] Brotman, L.S., Badler N.I. (1984) “Generating Soft Shadow with a Depth Buffer Algorithm”, *IEEE CG&A*, 4(10), 5-12.
[6] Chin, N., Feiner, S. (1989) “Near Real-Time Shadow Generation Using BSP Trees”, *Computer Graphics* 23(3), 99-106.
[7] Crow, F. (1977) “Shadow Algorithms for Computer Graphics”, *Computer Graphics* 11(2) 242-247.
[8] DirectX 6.1 SDK (1999) “Shadow Volume Sample”. Microsoft Corp., http://www.microsoft.com/DirectX
[9] Epic Games Inc., http://www.epicgames.com.
[10] Fuchs, H., Goldfeather, J., Hultquist, J., Spach, S., Austin, J., Brooks, F., Jr., Eyles, J., Poulton, J. (1985) “Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes.”, *In Proc. SIGGRAPH*, 19(7), 111-120.
[11] 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.
[12] Heidmann, T. (1991) “Real Shadows Real Time”, *Iris Universe*, 18, 28-31, Silicon Graphics Inc.
[13] Id Software Inc., http://www.idsoftware.com.
[14] Kilgard, M. (1997) “OpenGL-based Real-Time Shadows”, Silicon Graphics Inc., (http://reality.sgi.com/

mjk_asd/tips/rts/).
[15] Luebke, D. and Georges, C. (1995) “Portals and mirrors: Simple, fast evaluation of potentially visible sets”, *ACM Interactive 3D Graphics Conference*, Monterey, CA.
[16] McCool, M.D., (1998) “Shadow Volume Reconstruction”, University of Waterloo, http://www.cgl.uwaterloo.ca/~mmccool/.
[17] McReynolds, T., Blythe, D., (1998) “Advanced Graphics Programming Techniques Using OpenGL”, *SIGGRAPH’98 Course*.
[18] NVIDIA Corporation., http://www.nvidia.com.
[19] Schumacker, 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.
[20] Segal, M., Korobkin, C., Van Widenfelt, R., Foran, J., Haeberli, P. (1992) “Fast Shadows and Lightning Effects Using Texture Mapping”. *In Proc. SIGGRAPH*, 26(7), 249-252.
[21] Slater, M. (1992) “A Comparison of Three Shadow Volume Algorithms”, *The Visual Computer*, 1, 25-38.
[22] Thibault, W.C., Naylor, B.G. (1987) “Set Operations on Polyhedra Using Binary Space Partitioning Trees”, *Computer Graphics*, 21(4), 153-162.
[23] Wanger, L. (1992) “The Effect of Shadow Quality on the Perception of Spatial Relationships in Computer Generated Imagery”, *In SIGGRAPH Symposium on Interactive 3D Graphics*, 39-42.
[24] Whitted, T. (1980) “An Improved Illumination Model for Shaded Display”, *CACM*, 23(6), 343-349.
[25] Williams, L. (1978) “Casting Curved Shadows on Curved Surfaces”, *Computer Graphics*, 12(8), 270-274.
[26] Woo, A., Poulin, P., Fourier, A. (1990) “A Survey of Shadow Algorithms”, *IEEE CG&A*, 10(6), 13-31.