EUROGRAPHICS 2015 / O. Sorkine-Hornung and M. Wimmer
(Guest Editors)

Volume 34 (*2015*), *Number 2*

Partitioned Shadow Volumes

J. Gerhards[1], F. Mora[1], L. Aveneau[2], and D. Ghazanfarpour[1]

[1]University of Limoges - XLIM-CNRS, France
[2]University of Poitiers - XLIM-CNRS, France
{julien.gerhards, frederic.mora, lilian.aveneau, djamchid.ghazanfarpour}@xlim.fr

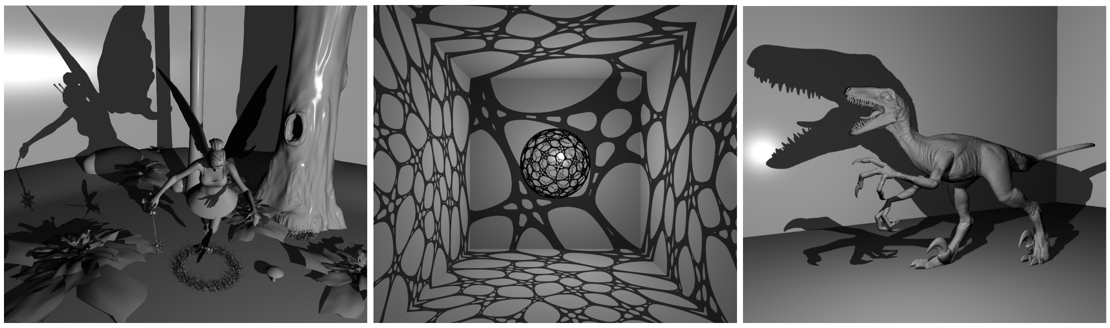


Figure 1: *Image rendered to a resolution of* 1024×1024 *with our novel shadow volume algorithm on a GTX 980 GPU. From left to right: Fairy (*4.14*ms, 174k triangles), Geodesic (*2.27*ms, 200k triangles), Raptor (*8.96*ms, 1 000k triangles).*

Abstract

Real-time shadows remain a challenging problem in computer graphics. In this context, shadow algorithms generally rely either on shadow mapping or shadow volumes. This paper rediscovers an old class of algorithms that build a binary space partition over the shadow volumes. For almost 20 years, such methods have received little attention as they have been considered lacking of both robustness and efficiency. We show that these issues can be overcome, leading to a simple and robust shadow algorithm. Hence we demonstrate that this kind of approach can reach a high level of performance. Our algorithm uses a new partitioning strategy which avoids any polygon clipping. It relies on a Ternary Object Partitioning tree, a new data structure used to find if an image point is shadowed. Our method works on a triangle soup and its memory footprint is fixed. Our experiments show that it is efficient and robust, including for finely tessellated models.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Computing shadows is a problem almost as old as the computer graphics history. Light and shadow are two sides of the same coin, as both should be rendered with accuracy to create realistic pictures. This often comes down to a visibility problem. If the light source is not visible from a point, it is in shadow. Thus, computing hard shadows from a point light source is easier than computing soft shadows from an area light source. However, easier does not mean easy. Despite a large amount of works in the literature [ESAW11], computing high quality hard shadows in real-time remains challenging for complex models. On graphic hardware, most of the methods rely on two different techniques: Shadow Maps and Shadow Volumes. If both are simple concepts, they are not easy to implement with robustness and efficiency. In this paper, we revisit the Shadow Volume BSP algorithm, a class of shadow volume algorithms which has received little attention. Our main contribution is a novel real-time shadow algorithm using a partition over the shadow volumes. In addition:

- It relies on a new data structure which can be computed efficiently in $O(n\log(n))$. Moreover, its memory footprint is in $O(n)$ and can be predicted.
- It does not require any polygon clipping and remains robust even on complex models.
- It works on a triangle soup and can support transparent or semi transparent occluders.
- The algorithm implementation is quite straightforward.

The paper is organized as follows: Section 2 recalls the main previous works on real-time hard shadows. Next, our approach is presented in Section 3, describing our new data structure and its usage to compute hard shadows. Section 4 analyzes our algorithms on different scenes, representing various configurations.

2. Related works

This section reviews the main real-time techniques with a focus on shadow volumes.

Shadow mapping [Wil78] is an image space technique. First, the geometry is rasterized from the light to create a depth map (the shadow map). Second, the geometry is rasterized from the eye and each visible point is projected onto the shadow map: If its distance from the light is larger than the depth stored in the shadow map, the point is in shadow. Shadow mapping is very fast because it relies on the z-buffer implemented as hardware on graphic card. Since it is an image space technique, it is not very sensitive to geometric complexity. This makes shadow mapping still more popular than shadow volumes, in spite of several drawbacks inherent to the rasterization. For example, 6 shadow maps are needed to support an omnidirectional light source. In particular, aliasing artifacts arise as soon as the shadow map resolution does not match the image points distribution. Many works address these issues. A detailed survey can be found in Real-Time Shadow by Eisemann *et al.* [ESAW11]. Despite aliasing problems, shadow mapping remains the reference technique for real-time applications. In contrast, shadow volumes are pixel-accurate but are generally slower and do not scale very well with the geometric complexity.

Shadow volumes are defined in object space. The former algorithm was presented by Frank Crow in 1977 [Cro77]. A shadow volume is the region of space occluded by an object with respect to the light source. At first, the algorithm computes the silhouette edges of the object. Next, they are extruded from the light to create oriented quads that define the shadow volume boundaries. A visible surface from the eye is shadowed if it is covered by more front-facing quads than back-facing quads. The shadow volume algorithm became popular with Heidmann's implementation [Hei91] on modern graphic hardware: Shadow quads are rendered in the stencil buffer; The stencil counters are incremented by a front-facing quad and decremented by a back-facing quad; Then, pixels with a non zero value are shadowed. However, several steps must be handled with a close attention to get an efficient and robust implementation.
At first, computing the silhouette is not straightforward. It requires the mesh connectivity information. A silhouette edge is shared by two triangles, one facing the light, the other back-facing the light. This definition holds for watertight objects and can be extended to support non-closed objects for manifold geometry [Ber86] or general geometry [AW04].
Heidmann's algorithm counts the shadow quads from the eye to the viewpoint. This process is called *z-pass* because it considers the quads passing the depth test. However, the stencil counters must be initialized to the number of shadow volumes containing the eye. The shadow quads may also be sliced by the near or far view frustum plane. This is a problem because shadow volumes must be closed to count properly the shadow quads. Different solutions have been proposed, involving additional capping planes, but they are not robust enough [EK02]. Notice that a "reasonably robust" solution is also proposed in [ESAW11], but its correctness can not be guaranteed. Another solution is to count the shadow quads from a point at infinity (and thus always lit) to the view point. This process is called *z-fail* [BS99, Car00] because it considers the quads failing the depth test. As a counterpart, *z-fail* is generally slower than *z-pass*, because it generates a higher fill rate.
Increasing geometric details or image resolution penalizes the shadow volume algorithm. Silhouette edges become more costly to compute, and they generate a lot of shadow quads. This saturates the fill rate because the stencil buffer requires too many updates. To support complex models, useless shadow casters can be culled [LWGM04]. Both a shadowed caster or a caster whose shadow is not visible from the eye can be culled safely. The same process can be applied to an object hierarchy [SWK08] to quickly culled objects instead of individual triangles.
Sintorn *et al.* [SOA11] show that a software hierarchical shadow volume rasterizer can outperform the hardware z-buffer: First, a 2D hierarchy is built over the view samples; Second, each shadow volume traverses the hierarchy to find the contained samples. In particular, the shadow volumes are computed per triangle, called shadow frustums. This avoids the silhouette computation, and enables support for transparent occluders. In [SKOA14], the authors notice that this algorithm may fall short in some situations. Thus they replace the former 2D hierarchical structure with a 3D cluster hierarchy, improving the previous method. Both algorithms are parallelized using CUDA and outperform consistently a *z-pass* implementation.

All those previous works share a common point. Similarly to Crow's algorithm, they render the shadow volumes from the eye. The work presented in this paper falls in another class of algorithms which has received much less attention. In contrast to Crow's algorithm, the visibility problem is considered from the light, not from the eye. Chin and Feiner [CF89] have presented such an approach. They use the shadow volumes planes to compute a Binary Space Partitioning (BSP) tree that represents the view from the light. The data structure is called the Shadow Volume BSP (SVBSP) tree. In this early work, shadow volumes are considered per visible polygons, and silhouette edges are not computed. Geometry is filtered down the BSP tree in a front-to-back order from the light. Each polygon is split in several fragments along shadow volume planes. A fragment located outside of every shadow volume is lit, otherwise it is shadowed. The lit fragments grow the BSP tree by unioning their shadow volumes. Once all the geometry is subdivided into lit and shadowed fragments, they are used to render the scene from the eye. Basically, this approach is a modified version of the BSP algorithm presented by Thibault and Naylor [TN87, NAT90] to handle polyhedral set operations: Here, the polyhedra are the shadow volumes. Equivalent algorithms have been applied to image representation [Nay92] or occlusion culling [BHS98]. The SVBSP algorithm has been applied on static scenes as a precomputed step, but it has also been extended to support dynamic scenes [CS95] of modest complexity. To build a BSP tree, geometric operations are required (*i.e.* polygon/plane clipping). This is expensive and prone to numerical issues and geometric degeneracies. As a consequence, this kind of approach has been left aside because it is not considered practicable for real-time shadows using complex models. In this paper, we hope to demonstrate the contrary.

3. Partitioned Shadow Volumes

This section gives a brief overview of our approach and presents our data structure. Next it details its construction, and how it is used to query whether a point is outside (lit) or inside (shadowed) a shadow frustum (*i.e.* a shadow volume cast by a single triangle). At last it presents a simple optimization, and exposes our implementation.

3.1. Overview

Our approach has two steps: First it builds an acceleration data structure to represent the visibility from a light source; Second, it uses the former to compute if each image point is lit or shadowed.

In the first step, a partition is built using per-triangle shadow frustums, each one being defined by four equation planes. Doing so, our approach is clearly in the same class of algorithms as Chin and Feiner [CF89]. In particular our data structure is view-independent. Of course, if the light is moved, or if the geometry is moved or modified, the partition has to be built again. But, contrary to prior works, our partition is built without computing any polygon clipping. Thus, for each frame, the partition can be computed efficiently avoiding robustness issues. In addition, the partition has a fixed and predictable memory footprint, an interesting property, especially since we target a GPU implementation.

In the previous BSP approaches, the construction of the partition aims at subdividing all the triangles into lit or shadowed fragments. Instead, our second step uses the partition as an acceleration data structure to determine, for each image point, if it is inside at least one shadow frustum. Possibly, it can find all the shadow frustums containing an image point, and so can handle transparent occluders.

3.2. Data structure

At first, we analyze the previous BSP methods to justify the need for a different data structure. We start with the following observation: Each BSP plane splits the 3D space into two separate half-spaces, one being positive, the other negative. It is easy to locate a point in a BSP tree, because a point is always contained in one of the two half-spaces: The input fits the data structure. However, if the input contains planar or volumetric geometry, it can be intersected by a plane. A BSP tree does not account for this third case. Thus intersections have to be computed to make the input fits the BSP tree.

To avoid such geometric operations, we use a different data structure representing the 3 configurations that may occur when a triangle is tested against a plane. This is the starting point of our work. Thus, we define the *Ternary Object Partitioning* (TOP) tree. Its inner nodes store one equation plane and have 3 child nodes:

- one containing all the geometry included (strictly) in the positive half-space of the plane;
- one containing all the geometry included (strictly) in the negative half-space of the plane;
- one containing all the geometry intersected by the plane.

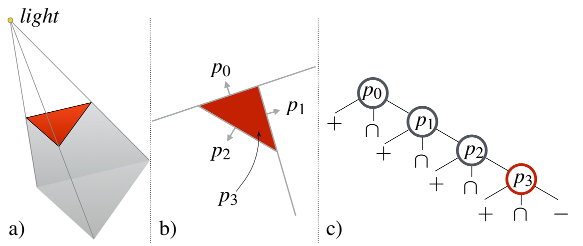


Figure 2: *a) A shadow frustum cast by a triangle. b) The shadow frustum seen from the light and its 4 bounding planes: p_0, p_1, p_2 are the 3 shadow planes defined by the light and an edge of the triangle, while p_3 is the supporting plane of the triangle. By convention, the inside is negative, the outside positive. c) The TOP tree representing the shadow frustum.*

Figure 2 shows a TOP tree representing a single shadow frustum. We need 4 inner nodes to store the 4 bounding planes of the shadow frustums. Each shadow frustum is oriented so that its inside is negative and its outside is positive. The 3 shadow planes correspond to the 3 first nodes while the triangle supporting plane, or capping plane, is always stored in the fourth node. This is not restrictive but this convention will be useful in section 3.6. Filtering a triangle in

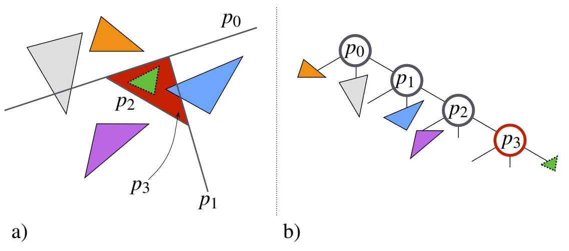


Figure 3: *a) A shadow frustum seen from the light and 5 triangles in various configurations wrt the 4 bounding planes. The shadow frustum is generated by the red triangle. $p_0, p_1, p2$ are the 3 shadow planes, p_3 is the capping plane. Notice the smallest triangle fully inside the shadow frustum. b) The TOP tree representing the shadow frustum and the location of the 5 triangles according to their positions wrt the planes.*

a TOP tree is not more difficult than locating a point in a BSP tree. Its vertices are tested with respect to a node plane. If they are all in its positive (resp. negative) half-space, the triangle belongs to the positive (resp. negative) child node. Otherwise it belongs to the intersection child node. Figure 3 illustrates the location of several triangles in a TOP tree corresponding to a single shadow frustum. The four nodes representing a shadow frustum do not have the same role. The first 3 nodes (containing the shadow planes) behave like a directional sort to partition the geometry as seen from the light. The fourth node (storing the capping plane) contains all the geometry (strictly) inside the 3 shadow planes, and so behaves like a depth sort from the light. Its positive (resp. negative) child node contains geometry in front of (resp. behind) the triangle, while its intersection child node contains geometry that intersects the triangle. Thus, the structure naturally supports triangles that intersect each other.

Notice that a TOP tree is not a BSP tree with 3 branches: Using a TOP tree completely changes the partition nature. A BSP tree represents a Euclidean space partition, while a TOP tree represents an object-space partition. Each triangle is classified according to its position with respect to a plane, and it is inserted into exactly one of the three child nodes. We do not have to compute geometric intersection anymore. In a different context (target tracking, vector correlation...), a comparable ternary tree structure has been described by Jeffrey Uhlmann in 1991 [Uhl91, Uhl08] using axis-aligned planes and bounding boxes as input. Surprisingly, we are not aware of previous works using a similar data structure in computer graphics.

The next section details our algorithm to build a full TOP tree using a set of shadow frustums.

3.3. Building a TOP tree

Algorithm 1 Merging algorithm: Given a triangle t, the algorithm filters the triangle down the tree to find the leaf it belongs to. Then, this leaf is replaced by the subtree representing the shadow frustum cast by t. Blue lines (5-7, 19-22) are related to the optimization explained in section 3.5.

```
 1: Node {
 2:    // A shadow plane or a capping plane
 3:    Plane plane
 4:    Node pos, inter, neg // the 3 child nodes
 5:    // Optimization: angle that defined the (conservative)
 6:    // wedge spanning the intersection child node
 7:    float angle
 8: }
 9: TOP_mergeShadowFrustum(Node root, Triangle t, Light l)
10: Node n ← root
11: while n is not a leaf do
12:    float location ← position( n.p, t )
13:    if location > 0 then
14:       n ← n.pos
15:    else if location < 0 then
16:       n ← n.neg
17:    else
18:       // n.plane intersects t
19:       if n.plane is a shadow plane then
20:          // update n.angle if t makes a greater angle wrt n.p
21:          updateWedge( n, t )
22:       end if
23:       n ← n.inter
24:    end if
25: end while
26: replaceLeafByShadowFrustum(n, t, l)
```

We build a TOP tree using a randomized incremental algorithm. It is randomized since the triangles are filtered down the tree in a random order. It is incremental because each triangle reaching a leaf grows the tree. Algorithm 1 details how a shadow frustum is merged with a TOP tree (this section skips over the optimization which is discussed in section 3.5). Given a random triangle, the algorithm starts from the root node and tests the triangle position with respect to the node plane (line 12). According to this position, the process continues iteratively in one of the three child nodes (lines 14, 16, 23). When a leaf is reached, it is replaced by the TOP subtree representing the shadow frustum cast by the triangle (line 26). The TOP tree is complete when all shadow frustums are merged, following the same process. Notice that testing a triangle with respect to a plane is the only required geometric computations, and that dynamic memory allocations are not needed.

Building a TOP tree is really straightforward. This is interesting because in dynamic environments, the data structure has to be built at each frame. Given a scene with n triangles, inserting a triangle is expected in $O(\log(n))$. Thus the time complexity for a full TOP tree is $O(n\log(n))$. This can be connected to the quicksort algorithm: Another way to see this construction is to consider that for each node, we have to sort all the remaining triangles into 3 sets. This is comparable to the classic "Dutch national flag problem" proposed by E. Dijkstra [Dij97], which can be solved using a quicksort algorithm variation (sometimes called "fat pivot" or "fat partition" [BM93]) treating values equal to the pivot.

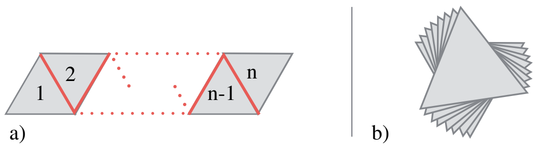


Figure 4: *Pathological cases. All triangles are seen from the light. a) Given a triangle i, $1 < i < n$, its red edges create shadow planes rejecting all triangles $j \neq i$. Thus, inserting triangles from 1 to n or n to 1 leads to the worst case complexity. b) All shadow frustums intersect all triangles, leading to the worst case complexity whatever the insertion order.*

Building a TOP tree is expected in $O(n\log(n))$. But if all the triangles are always inserted in the same subtree, it leads to the worst case complexity $O(n^2)$, growing a tree with height n. This may happen with two configurations, illustrated by figure 4. Let us notice that a BSP tree has the same problem with the first configuration. With the second one, it is even less robust because it would quickly run out of memory. Indeed, 2^n polygons would be created, contrary to a TOP tree whose memory is fixed.

The first pathological case is artificial (see figure 4.a), but it resembles a triangle strip, which is rather common. Inserting triangles following their strip order approaches the worst case scenario. Since our algorithm inserts triangles in a random order, it is avoided because the probability to retrieve the (reverse) strip order is 2 out of $n!$, for n triangles. This is similar to the quicksort algorithm: both algorithms run at worst in $O(n^2)$, but thanks to randomization, they run in practice in $O(n\log(n))$. There is nothing we can do about the second pathological case (see figure 4.b) since it does not depend on the insertion order. But this is a very singular case involving both specific geometry *and* light position.

The memory complexity for a TOP tree is $O(n)$. Since the geometry is not duplicated, split or intersected, we can easily predict the size of a TOP tree. Assuming an input of n triangles, this generates n shadow frustums; each one grows the tree by 4 inner nodes. Thus n triangles lead to a TOP tree with exactly $4n$ inner nodes.

3.4. Point query

Algorithm 2 Point query algorithm: The algorithm locates a point in a TOP tree and finds if it is inside at least one shadow frustum. Blue lines (9-11, 13) are related to the optimization explained in section 3.5.

```
 1: TOP_pointQuery(Node root, Point pt)
 2: NodeStack stack ← root
 3: while stack is not empty do
 4:    Node n ← pop(stack)
 5:    if n is not a leaf then
 6:       int location ← sign(n.plane, pt)
 7:       if n.plane is a shadow plane then
 8:          // shadow plane case: check intersection node
 9:          // is pt inside the intersection wedge?
10:          float ptAngle ← computeAngle( n.plane, pt )
11:          if ptAngle < n.angle then
12:             push(stack, n.inter)
13:          end if
14:          push(stack, location > 0 ? n.pos : n.neg )
15:       else
16:          // capping plane case: check intersection node
17:          if location < 0 then
18:             return 0 // pt inside at least one shadow frustum
19:          end if
20:          push(stack, n.inter)
21:          push(stack, n.pos)
22:       end if
23:    end if
24: end while
25: return 1 // pt is outside any shadow frustum
```

To check if a point is shadowed, we need to locate it in the TOP tree and find if it is inside at least one shadow frustum. This query can be seen as a hybrid algorithm between the location of a point in a space partition and the location of a point in an object-space partition. Indeed, in a TOP tree, the positive and negative child nodes do not intersect in space since they are separated by the plane stored in their parent node. This is similar to a BSP tree. However the intersection child node may have an intersection (in space) with the other two. This is similar to a Bounding Volume Hierarchy.

Algorithm 2 details how a point is located in a TOP tree (once again, the optimization is explained in the next section). It distinguishes between the two kind of planes. As explained in section 3.2, a shadow plane behaves like a directional sort while a capping plane behaves like a depth sort. At first, a stack is initialized with the tree root (line 2). For each stack node, it always computes the position of the point with respect to the node plane (line 6). If it is a shadow plane (line 7), the intersection node needs to be inspected (line 12). But only either the positive or negative node needs to be explored according to the position of the point (line 14). If the node stores a capping plane (line 15) we already know that the point is inside the 3 shadow planes of a frustum. We just have to check if it is before or behind the related triangle. If the point is behind the triangle, it is inside the shadow frustum (line 18). This is an early termination case: The point is shadowed. Then, the intersection node is pushed (line 20). This case appears with self-intersecting triangles. At least, the point is in the positive half-space of the capping plane, and so before the triangle. Thus, the process continues in the positive child node (line 21). Obviously, the algorithm ends when the stack is empty. In this case the point is visible from the light since it is not inside any shadow frustum (line 25).

Notice that this algorithm can be slightly modified to handle transparent triangles. Instead of ending the algorithm as soon as the point is found inside a shadow frustum (line 18), we can compute the light attenuation due to a transparent triangle. Then the algorithm continues in both sides of the capping plane to find if the point is inside other shadow frustums. In this case, the algorithm always ends when the stack is empty, and returns the total light attenuation.

In this first version, intersection child nodes are always pushed onto the stack. This may decrease the efficiency because the cost is linear with respect to the number of visited nodes. Thus we now present an optimization to avoid exploring all the intersection child nodes.

3.5. Optimization

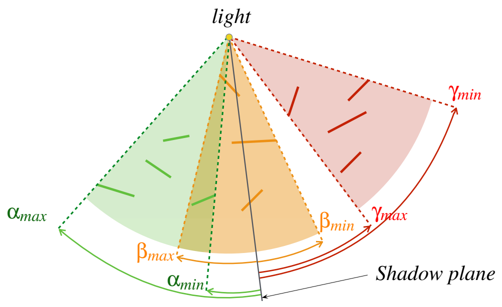


Figure 5: *A shadow plane subdivides the geometry into 3 sets. Each one is bounded by an unclosed wedge. Such a wedge can be defined by two angles with respect to the shadow plane. For example β_{min} and β_{max} are the angles that define the wedge enclosing the intersection child node. Thus, when a point is tested against a shadow plane during a query, a child node must be visited only if its wedge contains the point.*

As illustrated by figure 5, the geometry inside the 3 child nodes of a shadow plane can be bounded by two planes which define an unclosed wedge. If a point is not inside a wedge, the corresponding child node does not need to be visited. We can use this property to decrease the number of visited nodes. In particular this is interesting for the intersection nodes because their wedges are generally small. Thus we can expect a lower probability for a point to be inside the wedge of such an intersection node.

We define a wedge with two angular values, the minimum and the maximum angles between the shadow plane inside the node and the triangle vertices. A node only needs to be visited if the angle between the point and the shadow plane is included into the two wedge angles. This rejection test could be done for the 3 child nodes, requiring to store 6 additional floats per node. But, for positive and negative child nodes, our experiments have shown that the improvement was negligible. In this case the trade-off between memory and efficiency is not interesting. Thus the wedge test is restricted to the intersection nodes. Finally, we adopted a conservative test: We define the wedge of an intersection node as the maximum of the absolute values of its two angles. This conservative test is slightly less efficient but only requires one float per node. We find out this solution to be the best trade-off.

This optimization implies two minor modifications in the previous algorithms. In algorithm 1, each time a triangle is intersected by a plane, the maximum absolute angle spanning its vertices is computed and, if it is greater than the wedge value already stored in the node, the latter is updated (lines 19-22). In algorithm 2, we now push the intersection nodes onto the stack if and only if the absolute angle between the point and the plane (line 10) is smaller than the angle stored in the node (line 11). Let us notice that such a test is only relevant for a shadow plane, and not for a capping one.

3.6. Implementation

Our implementation is compatible with any graphic hardware with support for OpenGL 4.3. Both algorithms are implemented using OpenGL Shading Language (GLSL). Algorithm 1 is implemented with a compute shader. At each frame, it builds the TOP tree stored in a Shader Storage Buffer Object (SSBO). Algorithm 2 is implemented with a fragment shader. It queries the TOP tree in the SSBO to determine the visibility of a fragment from the light. We give the main keypoints for each shader.

Shader Storage Buffer Object: This is an array of nodes. Each node is referenced by its index in the array, and requires 32 bytes: 16 bytes are used for the plane equation, 12 bytes are used for the indexes of the 3 child nodes, and 4 bytes are used for the wedge value of the intersection child node. Notice that we do not need to represent explicitly the leaves: In our implementation, any leaf index equals zero and the nodes are written from index 1. As a consequence, for a scene made of n triangles, $4n+1$ nodes are allocated for the SSBO.

Compute Shader: Each invocation of the compute shader merges one triangle with the TOP tree. Basically, the *nth* invocation merges the *nth* triangle. However, to simulate a pseudo random insertion order, we use a permutation list so that the *nth* invocation merges the triangle referenced at the *nth* position in the list. The compute shader invocations run

concurrently, and each conflict is handled using atomic operations. A first concurrent access happens if two invocations try to merge a shadow frustum at the same time and at the same place in the tree (see algorithm 1, line 26). Replacing a leaf is equivalent to replace its zero index with the root index of the subtree representing a shadow frustum. This is done using an atomic *compare-and-swap* operation. A second concurrent access happens when an invocation wants to update the angle value in a node (see algorithm 1, lines 19-22). Each invocation computes the absolute angle between its triangle and a shadow plane. If this value is greater than the node wedge value, this latter is updated. Thus we need an atomic maximum. Unfortunately, such an operation is only available for integers, not floats. Proprietary extensions exist but this would restrict the method portability. Thus, we tried a standard solution: Each floating-point value is encoded as an integer. This is achieved using a GLSL function (`floatBitsToUint`) that preserves the bit-level representation. Next the atomic maximum operation is applied on integer representation. Comparison between floats encoded as integers remains valid as long as the values are positive. And we only deal with absolute values.
Atomic operations have become very efficient since Kepler architecture. Their overhead is almost negligible if no conflict occurs. However, at the beginning of the building process, a lot of atomic conflicts are expected because running threads are concentrated in the first levels of the TOP tree. But, the more the tree grows, the more the threads are distributed in different nodes, the less conflicts arise.

Fragment Shader: This shader is a straightforward implementation of algorithm 2. It is plugged in a deferred pipeline to ensure that it runs once per pixel. Nevertheless, it could also be used in a forward pipeline. Unless otherwise stated, we use a stack size (algorithm 2, line 2) of 32 nodes. The stack size should be carefully tuned. Large sizes consume registers and decrease efficiency. On the contrary, a stack overflow will crash the program if the size is too small. To find out if a node contains a shadow plane (algorithm 2, line 7), we only have to test if the node index is not a multiple number of 4.

4. Results

Unless otherwise stated, all experiments were run both on a NVIDIA GTX 780 GPU and a NVIDIA GTX 980, with an image resolution of 1024×1024. The compute shader (CS) and the fragment shader (FS) are always applied to each frame. Our scenes present different configurations as shown in figure **??**. From top to bottom: TREX (20k triangles) is our smallest model, it generates complex shadows. BUNNY (69k triangles) has more triangles but has a less complex shadow. FAIRY (174k triangles) presents various geometric details. GEODESIC (200k triangles) is used with a light located inside the model, casting shadows in all directions. CITADEL (393k triangles) is a game level scene from the Epic UDK. At least, RAPTOR (1 000k triangles) is a highly detailed dinosaur with lots of ripple on its skin.

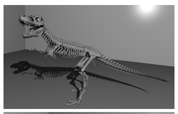
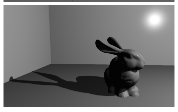
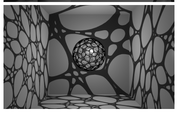
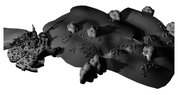
4.1. Memory

In all our experiments, the SSBO is always allocated to support all the geometry, even if the culling of front facing triangles is enabled. A benefit of our PSV algorithm wrt traditional BSP methods is that the memory footprint is linearly dependent on the number of triangles. Since a triangle requires 4 nodes, the SSBO size is $4 \times 32 \times (n+1)$ bytes for n triangles. Thus, memory consumption is not a serious limitation: 2.44 MB (TREX), 8.43 MB (BUNNY), 21.24 MB (FAIRY), 24.42 MB (GEODESIC), 48 MB (CITADEL) and 122MB (RAPTOR).

4.2. Shadows

To compute shadows, a front-face-culling step is always applied before building the TOP tree. Both the CS and FS times are measured per frame in milliseconds during a fly-through animation. The number of nodes visited on average by a FS invocation (*i.e.* per pixel) is also recorded. The table in figure 6 sums up the results. For comparison purpose, our PSV algorithm was run without the optimization presented in section 3.5 (No opt). The multiplication factors reported show that the wedge optimization is very efficient. For example, without optimization, the method is almost 23 times slower on the GEODESIC model. Notice that the CS is slightly faster without this optimization. Indeed, the angle value is not updated anymore, so the number of concurrent accesses decreases. However the overall times (CS+FS) are generally dominated by the FS. We have also evaluated our algorithm without a random insertion order of triangles in the TOP tree (No rand). The results are slightly slower for all the scenes. However we could have expected a more significant impact. The scheduling of the CS invocations by the graphic drivers affects the insertion order of the triangles. We suspect that it contributes to simulate a per-block randomized insertion.

We now focus on the PSV algorithm. The experiments run on a GTX 980 are 2 (RAPTOR) to 2.7 (CITADEL) times faster than on a GTX 780. This exceeds, by far, the difference in computational power between the graphics cards. Our method is not computationally intensive, but requires many memory accesses since each fragment/pixel queries the TOP tree (the traversal number equals the image resolution, 1024×1024). The GTX 980, based on the Maxwell architecture, improves the memory management over the GTX 780 which is based on the Kepler architecture. Our method automatically benefits from this evolution. This also illustrates that the memory access efficiency is the main bottleneck. As shown by figure 6, graph a), the CS time tends to be proportional to the number of triangles regardless of the model nature. The theoretical complexity to sequentially build a TOP tree is $O(nlog(n))$ (see section 3.3). This is co-

			GTX 780			GTX 980		
	Version	Nodes	CS	FS	CS+FS	CS	FS	CS+FS
	**PSV**	16.48	0.25	4.96	**5.21**	0.18	1.85	**2.03**
	No opt	×11.15	×0.76	×12.59	×12.02	×0.94	×14.98	×13.73
	No rand	×1.06	×0.88	×1.05	×1.04	×0.83	×1.0	×0.99
	No FFC	×1.41	×1.68	×1.27	×1.29	×1.67	×1.12	×1.17
	**PSV**	16.17	0.64	4.69	**5.33**	0.4	1.65	**2.05**
	No opt	×10.15	×0.81	×12.03	×10.68	×0.95	×14.91	×12.19
	No rand	×1.00	×0.98	×1.06	×1.05	×1.03	×1.69	×1.02
	No FFC	×1.12	×1.52	×1.25	×1.28	×1.80	×1.40	×1.40
	PSV	23.93	1.92	8.64	**10.56**	1.24	2.9	**4.14**
	No opt	×14.35	×0.87	×17.32	×15.79	×0.77	×10.41	×7.53
	No rand	×1.38	×0.98	×1.31	×1.25	×1.01	×1.34	×1.24
	No FFC	×1.44	×1.76	×2.14	×2.11	×1.37	×1.37	×1.37
	**PSV**	14.13	1.8	3.45	**5.25**	1.04	1.23	**2.27**
	No opt	×21.15	×1.13	×34.38	×22.98	×0.97	×43.93	×24.25
	No rand	×2.54	×1.27	×3.09	×2.47	×1.38	×1.17	×1.27
	No FFC	×1.66	×1.82	×2.84	×2.49	×1.91	×3.17	×2.59
	**PSV**	68.02	4.89	20.7	**25.59**	3.0	6.4	**9.4**
	No opt	×4.34	×0.84	×6.25	×5.06	×0.86	×8.49	×6.06
	No rand	×1.21	×0.96	×1.18	×1.14	×0.95	×1.18	×1.10
	No FFC	×1.17	×1.44	×1.25	×1.29	×1.57	×1.30	×1.39
	PSV	20.38	10.54	7.26	**17.8**	5.81	3.15	**8.96**
	No opt	×14.64	×0.87	×15.66	×6.90	×0.90	×16.72	×6.46
	No rand	×4.27	×2.15	×3.88	×2.86	×2.11	×3.37	×2.55
	No FFC	×1.09	×1.82	×2.63	×2.15	×2.72	×4.23	×3.25

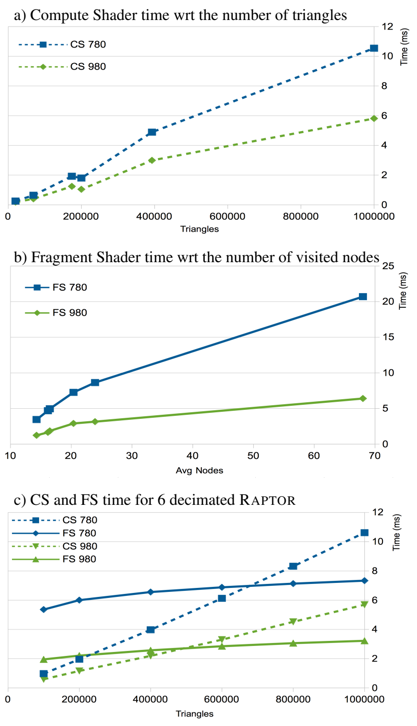


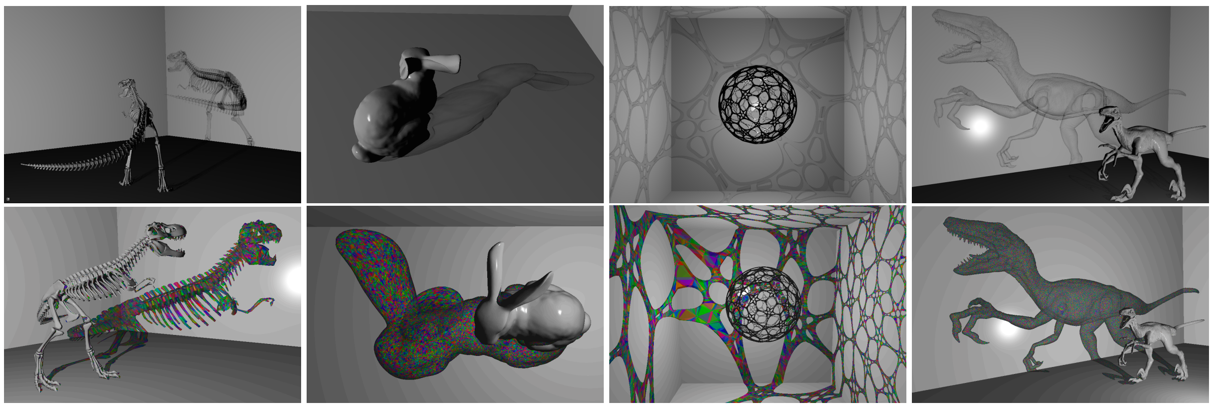
Figure 6: *Shadow computation. Left: the table sums up computation times for each model and for two GPU: a NVIDIA GTX 780 and a GTX 980. The PSV rows correspond to our method and the times are given on average in milliseconds. The CS column gives compute shader time per frame. The FS column gives fragment shader time per frame. The CS+FS column gives the total time. The Nodes column is the average number of nodes visited per fragment shader invocation (per pixel). This does not depend on the GPU. The table also gives the slowdowns in efficiency for different versions of the PSV algorithm (×N means N times the PSV algorithm time). No opt means without the wedge optimization. No rand means without a random insertion order of the triangles. No FFC means without front-face-culling. Right: a) The CS time with respect to the number of triangles for each model. b) The FS time with respect to the number of visited nodes for each scene. c) The CS and FS time with respect to geometric complexity (using 6 decimated versions of the* RAPTOR *model).*

herent with our experiments that bring forwards the dominant term *n*.

Graph b), figure 6, shows that the FS time is proportional to the number of visited nodes. It is not proportional to the number of triangles in the scene. For example the FS time on TREX (20k triangles) and the BUNNY (69k triangles) are equivalent. Since the TREX casts more complex shadows, it involves a complex TOP tree. Hence, FS finally visits on average 16 nodes on both models.

FS depends on both geometric complexity and shadow complexity. Thus, we have decimated the RAPTOR model at several resolutions to evaluate our PSV algorithm independently of the shadow complexity. Results are presented on figure 6, graph c). They confirm the linear behaviour of the CS. Moreover, the FS presents a logarithmic behaviour with respect to the number of triangles, corresponding to the expected complexity discussed in the section 3.4. Thanks to this property, the PSV algorithm is not sensitive to the geometric complexity. For example, 20.38 nodes are tested on average per pixel on the RAPTOR model. Since one triangle implies 4 TOP nodes, this is equivalent to test roughly 5 triangles per pixel. It is very few considering a TOP tree built per frame with 500k triangles on average (using back-facing triangles).

CITADEL is the most complicated scene in our tests. On average, 68 nodes are visited per pixel. Indeed, the citadel city geometry has highly variable depths from the light. It generates a lot of shadow volumes that intersect each others, increasing the intersection node sizes. This is the model that requires the most memory accesses. It explains why the difference between the GTX 780 and the GTX 980 is the greatest on this scene. In addition to the average number of visited nodes, the maximum number also conveys the visual complexity from the light source. For the 6 models presented in figure 6, the maximum numbers of visited nodes are 303



Avg nodes: 23.25
GTX 780: 15.35ms (0.45 + 14.9)
GTX 980: 3.39ms (0.30 + 3.09)

Avg nodes: 22.95
GTX 780: 14.86ms (0.97 + 13.89)
GTX 980: 3.67ms (0.72 + 2.95)

Avg nodes: 25.63
GTX 780: 26.87ms (3.27 + 23.60)
GTX 980: 8.41ms (1.99 + 6.42)

Avg nodes: 28.67
GTX 780: 46.23ms (19.18 + 27.05)
GTX 980: 33.90ms (15.78 + 18.12)

Figure 7: *Results with transparency, using the modified point query (see section 3.4). First row: x-ray shadows. Second row: colored x-ray shadows, using one color per triangle. Avg nodes is the average number of nodes visited by a FS invocation. The computation times are given on average per frame and come with the shader times in parenthesis (CS+FS). Notice the right images revealing a kind of stomach inside the* Raptor *body.*

(TRex), 189 Bunny, 471 (Fairy), 197 (Geodesic), 1264 (Citadel), 486 (Raptor).

As noticed in section 3.6, our FS implementation uses a stack whose size may affect its performance. While a stack of size 32 is enough for all our models, we have measured that doubling the size (*i.e* 64) multiplies the FS times by 1.8 (TRex) to 1.6 (Raptor). The impact is more noticeable on the smallest models. This shows that it is important not to use a stack larger than necessary.

4.3. Transparency

As explained in section 3.4, a slightly modification of the point query algorithm allows to find all the shadow frustums containing a given point. This is useful to support transparent or semi-transparent occluders. To test this modified version, we render the shadows as if all the triangles were transparent. Every time the point is found in a shadow frustum, we compute the light attenuation due to the related triangle, taking into account the angle between its surface and the point-to-light direction. While this is clearly not realistic, it creates a nice effect: It looks like x-ray views. This modified fragment shader uses a stack size of 64 nodes instead of 32. Indeed, since it searches for all the shadow frustums, more nodes have to be visited. Possibly, a material property can be used per triangle. Since each shadow frustum leads to 4 consecutive nodes, we do not have to store the negative child index for the 3 first nodes. It is the current node index plus 1. Doing so, we can spare 3 integers per shadow frustum or triangle. To reveal this possibility, we also render colored x-ray shadows using one color per triangle. Since the triangles are merged randomly into the TOP tree, notice that this query does not consider the geometry in a front-to-back order from the light. Nonetheless, it can be useful for order independent transparency techniques [MB13].

Figure 7 presents some results. Obviously, transparency is more difficult, and thus more expensive to compute. At first, front-face-culling is disabled and so all the geometry is used to build the TOP tree. It implies that the CS costs almost twice as much as in previous results. About the FS, the comparison with the previous tests is not straightforward, because the query algorithms to render shadows and x-ray shadows are different. In addition the increase in the stack size affects the FS times.

4.4. Increasing resolution

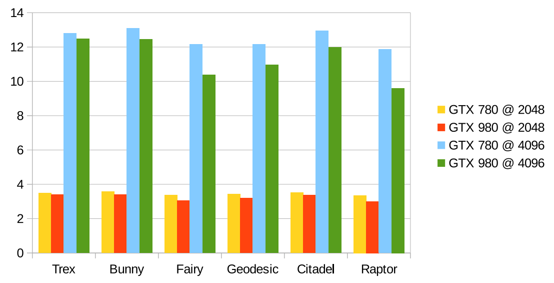


Figure 8: *The histogram represent the multiplication factors when rendering to a resolution of* 2048×2048 *and* 4096×4096*, compare to a resolution of* 1024×1024.

We have tested our PSV algorithm using different image resolutions. Notice that it only affects the FS computation. The CS and the memory consumption do not depend on the image resolution. Figure 8 shows that the computation time is multiplied by about 3.8 when rendering to a resolution of 2048×2048, whereas one would expect a factor 4. It is multiplied by a factor between 10 and 13 when rendering to a resolution of 4096×4096, less than the expected factor 16. Increasing the resolution improves the FS consistency: More FS follow the same path in the TOP tree. This explains why the computation time does not increase as much as the number of pixels.

4.5. Comparison with previous works

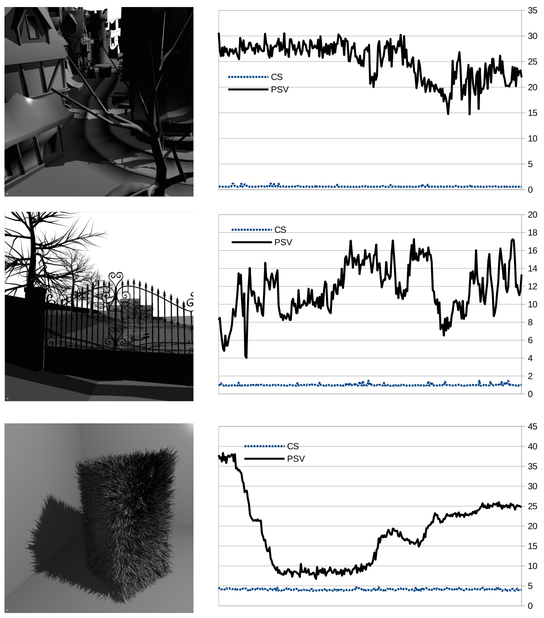


Figure 9: *From top to bottom: Small Citadel (60k triangles) is a part of the Citadel model; Villa (88k triangles) and Fuzzy (400k triangles). The graphs show the time per frame during a fly-through animation. The PSV line is the total time for our algorithm. The CS line is the compute shader time, and is almost constant. For the clarity of the presentation, the FS time does not appear. Nevertheless, it corresponds to the first minus the second.*

As explained in section 2, previous works that build a partition over shadow volumes are not enough robust and efficient to render our scenes, especially in real-time. Instead, we compare our approach to the algorithm recently presented by Sintorn *et al.* [SKOA14]. It can be considered as the fastest shadow volumes algorithm to date. This comparison is also interesting because the two algorithms are somehow dual: While we build an acceleration structure over the shadow volumes and query each image point, the algorithm of Sintorn *et al.* builds an acceleration structure over the image points and query each shadow volume. Both methods are dominated by the traversal cost of their data structure. Thus, given n triangles and p pixels, our method performs p queries and each one costs $O(\log(n))$. In contrast Sintorn *et al.* perform n queries, each one costing $O(\log(p))$.

We make an indirect comparison with the results presented in [SKOA14]. We use the same models and data, the same graphic card (NVIDIA Titan GPU) and the same image resolution (1024×1024). Figure 9 shows the results achieved by the PSV algorithm. Our algorithm is not as fast as the Sintorn one on these 3 test scenes. With the SMALL CITADEL model (60k triangles), the PSV algorithm is almost 10 times slower. Using the VILLA model (88k triangles), the PSV needs from 6 to 16ms which is 3 to 5 times slower. On FUZZY (400k triangles) the PSV is 1.3 to 1.5 slower. In these tests, $n \leq 400k$ is always smaller than $p = 1M$, which is not favorable to our approach. Hence, the experiments are coherent with the complexity of the two algorithms discussed above: The difference in efficiency decreases as n grows, because [SKOA14] scales linearly *wrt* n, while our method is logarithmic *wrt* n.

We have noticed that our approach does not benefit from the Titan GPU, based on the Kepler architecture, as the GTX 780 GPU. While the CS time is improved, the FS time is slower than using a GTX 780 GPU. The resulting overhead is 10 to 15%. Moreover, our GLSL implementation is probably not as finely tuned as the Sintorn CUDA implementation. The latter makes use of persistent threads to keep the CUDA cores fully utilized. The hierarchical structure uses a branching factor of 32 to fit CUDA warps. In contrast, our GLSL implementation does not allow as much control as CUDA. The shader execution is a matter for the graphic drivers, and we can not make any assumption on this process.

On the other hand, our approach requires less memory: 7.33 MB (SMALLCITADEL), 10.74 MB (VILLA) and 48.8 (FUZZY). The memory used by the Sintorn algorithm only depends on the image resolution. In this case (1024×1024), the authors report $198MB$ of memory for all their models. At the cost of a small overhead (at most 5%), they also give a more compact representation that requires 33 MB to 58 MB at worst.

5. Conclusion

In this paper, we present a new method to represent a partition over the shadow volumes, made from the light. It is an important evolution with respect to previous works that use BSP. Its main difference resides in the TOP tree, a ternary tree allowing to avoid any polygon clipping. The low and fixed memory consumption is a great advantage of our method. It is important to notice that such an algorithm will never exceed a predictable amount of memory. Our solution relies on two simple steps, implemented using a Compute and a Fragment shader. Our experimental results show

that it is very robust and renders real-time shadows. The increase in the number of triangles is generally not favorable to shadow volumes algorithms. But thanks to the partition we built over the shadow volumes, our algorithm is not very sensitive to the geometric complexity, because our data structure is queried in logarithmic time. This is an advantage for rendering highly detailed models. On the contrary, it is much more sensitive to the depth complexity from the light. This should be improved by computing several TOP trees according to the distance of the triangles from the light. We plane to explore this solution as a future work.
We have rediscovered an old class of algorithms that uses shadow volumes to build a partition from the light. We have shown that it can be raised to a high level of robustness and efficiency. While this kind of approach was left aside, we hope this work will generate a new interest. In addition, our method is pretty simple thanks to our data structure: the TOP tree. Beyond real-time shadows, we think that this structure may be useful in many other partitioning problems in computer graphics.

6. Aknowledgments

The authors especially thank Erik Sintorn for providing the models and data used in [SKOA14]. The Epic Citadel scene is distributed with the Unreal Development Kit by Epic Games. The Raptor model is provided courtesy of Sensable by the AIM@SHAPE-VISIONAIR Shape Repository.

References

[AW04] Aldridge G., Woods E.: Robust, geometry-independent shadow volumes. In *Proceedings of the 2Nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* (New York, NY, USA, 2004), GRAPHITE '04, ACM, pp. 250–253. 2

[Ber86] Bergeron P.: A general version of crow's shadow volumes. *Computer Graphics and Applications, IEEE 6*, 9 (Sept 1986), 17–28. 2

[BHS98] Bittner J., Havran V., Slavik P.: Hierarchical visibility culling with occlusion trees. In *Proceedings of the Computer Graphics International 1998* (Washington, DC, USA, 1998), CGI '98, IEEE Computer Society, pp. 207–. 3

[BM93] Bentley J. L., McIlroy M. D.: Engineering a sort function. *Softw. Pract. Exper. 23*, 11 (Nov. 1993), 1249–1265. 5

[BS99] Bilodeau W., Songy M.: Real time shadows. In *Creative 1999, Creative Labs Inc. Sponsored game developper conferences* (Los Angeles, California and Surrey, England, 1999). 2

[Car00] Carmack J.: Z-fail shadow volumes. Internet Forum, 2000. 2

[CF89] Chin N., Feiner S.: Near real-time shadow generation using bsp trees. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1989), SIGGRAPH '89, ACM, pp. 99–106. 3

[Cro77] Crow F. C.: Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 242–248. 2

[CS95] Chrysanthou Y., Slater M.: Shadow volume bsp trees for computation of shadows in dynamic scenes. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1995), I3D '95, ACM, pp. 45–50. 3

[Dij97] Dijkstra E. W.: *A Discipline of Programming*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. 5

[EK02] Everitt C., Kilgard M. J.: Practical and robust stenciled shadow volumes for hardware-accelerated rendering. http://developer.nvidia.com, 2002. 2

[ESAW11] Eisemann E., Schwarz M., Assarsson U., Wimmer M.: *Real-Time Shadows*. A.K. Peters, 2011. 1, 2

[Hei91] Heidmann T.: Real shadows real time, 1991. 2

[LWGM04] Lloyd D. B., Wendt J., Govindaraju N. K., Manocha D.: Cc shadow volumes. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (Aire-la-Ville, Switzerland, Switzerland, 2004), EGSR'04, Eurographics Association, pp. 197–205. 2

[MB13] McGuire M., Bavoil L.: Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (December 2013), 122–141. 9

[NAT90] Naylor B., Amanatides J., Thibault W.: Merging bsp trees yields polyhedral set operations. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 115–124. 3

[Nay92] Naylor B. F.: Partitioning tree image representation and generation from 3d geometric models. In *Proceedings of the Conference on Graphics Interface '92* (San Francisco, CA, USA, 1992), Morgan Kaufmann Publishers Inc., pp. 201–212. 3

[SKOA14] Sintorn E., Kämpe V., Olsson O., Assarsson U.: Per-triangle shadow volumes using a view-sample cluster hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, 2014), I3D '14, ACM, pp. 111–118. 2, 10, 11

[SOA11] Sintorn E., Olsson O., Assarsson U.: An efficient alias-free shadow algorithm for opaque and transparent objects using per-triangle shadow volumes. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (New York, NY, USA, 2011), SA '11, ACM, pp. 153:1–153:10. 2

[SWK08] Stich M., Wächter C., Keller A.: Efficient and robust shadow volumes using hierarchical occlusion culling and geometry shaders. In *GPU Gems 3*, Nguyen H., (Ed.). Addison-Wesley, 2008, pp. 239–256. 2

[TN87] Thibault W. C., Naylor B. F.: Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 153–162. 3

[Uhl91] Uhlmann J. K.: Adaptive partitioning strategies for ternary tree structures. *Pattern Recognition Letters 12*, 9 (Sept. 1991), 537–541. 4

[Uhl08] Uhlmann J. K.: *Handbook of Mutlisensor Data Fusion: Theory and Praction*, 2nd ed. CRC Press, 2008, ch. Introduction to the Algorithmics of Data Association in Multiple-Target Tracking, pp. 69–89. 4

[Wil78] Williams L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 270–274. 2