Eurographics Symposium on Rendering - Experimental Ideas & Implementations (2016)
E. Eisemann and E. Fiume (Editors)

Deep Partitioned Shadow Volumes Using Stackless and Hybrid Traversals

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

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

Abstract

Computing accurate hard shadows is a difficult problem in interactive rendering. Previous methods rely either on Shadow Maps or Shadow Volumes. Recently Partitioned Shadow Volumes (PSV) has been introduced. It revisits the old Shadow Volumes Binary Tree Space Partitioning algorithm, leading to a practicable and efficient technique. In this article, we analyze the PSV query algorithm and identify two main drawbacks: First, it uses a stack which is not GPU friendly; its size must be small enough to reduce the register pressure, but large enough to avoid stack overflow. Second, PSV struggles with configurations involving significant depth complexity, especially for lit points. We solve these problems by adding a depth information to the PSV data structure, and by designing a stackless query. In addition, we show how to combine the former PSV query with our stackless solution, leading to a hybrid technique taking advantage of both. This eliminates any risk of stack overflow, and our experiments demonstrate that these improvements accelerate the rendering time up to a factor of 3.

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

Real-time and pixel-accurate shadows remain a challenging problem in computer graphics. Shadows are important for realistic rendering because they unveil spatial relationship between objects. Many algorithms can compute accurate shadows, but few of them can do it in real-time. Currently, most of the real-time shadow algorithms rely on shadow mapping or shadow volumes. While Shadow Maps are not pixel-accurate, they are widely used in practice because they are fast. Shadow Volumes are less efficient but they are still investigated because they are pixel-accurate. A huge amount of work tried to improve the shadow maps accuracy and the shadow volumes efficiency. However, both techniques still have drawbacks, which leads to search for different solutions.

In this respect, Gerhards *et al.* have recently proposed a novel approach, the Partitioned Shadow Volumes (PSV) algorithm [GMAG15]. This method relies on an old idea [CF89] which was completely different from the original Shadow Volumes algorithm proposed by Crow [Cro77]. Thanks to a specific partitioning strategy, PSV have many advantages and allow real-time and pixel-accurate shadows. This makes the algorithm an interesting option which has been little explored, contrary to shadow mapping or shadow volumes.

Thus, in this article, we study the PSV technique to find issues that could still hinder practical applications. We identify two main problems. Firstly, PSV can struggle with some specific geometric configurations. Secondly, the PSV implementation uses a stack whose size need to be carefully tuned to garantee a correct output. We propose three solutions to improve both robustness and efficiency. More precisely, our contributions are:

- We add a depth constraint to improve the PSV stability, whatever the geometric configuration.
- We propose a new stackless algorithm to eliminate all the drawbacks related to the former stack-based algorithm.
- We propose a hybrid algorithm (a short stack-based query combined to a stackless one) which benefits from both techniques.

We provide an extensive performance evaluation to highlight the algorithms behavior and their efficiency.

This paper is organized as follows: Section 2 recalls previous works related to real-time shadows; Section 3 briefly analyses the PSV algorithm and presents our contributions; Section 4 reports all our experiments and comparisons to the former PSV algorithm as well as a z-pass implementation of Shadow Volumes.

2. Related works

For a comprehensive survey of real-time shadows, we refer the reader to the book by Eisemann *et al.* [ESAW11].

Shadow mapping [Wil78] is a mature technique which has become a standard for real-time shadows. Shadow maps are very

DOI: 10.2312/sre.20161212

fast but they are not without drawback. They are not pixel accurate and the shadow quality strongly depends on the texture resolution. Aliasing or jagged shadows appear if the shadow map resolution does not match the eye-space samples. A depth bias has te be tuned to avoid self-shadowing. At grazing angles, it is barely impossible to avoid artefact without dramatically increasing the map resolution. Many works have focused on these problems and different solutions were proposed such as adaptive shadow maps [FFBG01] or irregular z-buffer [JLBM05, WHL15]. But adaptative or irregular structures are less GPU friendly. Cascaded shadow maps [Eng06, ZSXL06] are often preferred because they rely on traditional shadow maps. However several shadow maps at different resolutions have to be rendered. In practice, a shadow map often exceeds the screen resolution to guarantee shadow quality. But the resolution increases year after year, and 4K monitors are already available as high-end products. Today, 4096×4096 is a very common map resolution and 8192×8192 is not exceptional. The memory cost becomes a concern. Scandolo et al. [SBE16] are interested in compressing shadow maps because some situations already exceed the available memory on commodity GPUs. Moreover, six shadow maps are usually required to support one omnidirectional light source.

Shadow volumes were introduced by Crow [Cro77]. Contrary to shadow maps, it is an object based and pixel-accurate technique. Heidmann [Hei91] has provided the first hardware implementation, known as Z-PASS. A shadow volume appoints the region of space hidden from the light by an object. Silhouette edges are extruded from the light to create quads bounding the shadow volumes. The shadow quads are rasterized from the camera to count if image points are covered by more front-facing quads than back-facing ones. This is like comparing how many times a ray cast from the camera enters and exits a shadow volume, until it reaches an image point. Shadow volumes support directional as well as omni-directional light sources without any modification. They also have well-known drawbacks. The mesh connectivity is required to compute silhouette edges. And silhouette computation is not straightforward using general meshes [AW04]. The "eye-in-shadow" position is a well known issue of the Z-PASS implementation, because the counters need to be initialized to the number of shadow volumes containing the camera. A solution is to count shadow quads from a point at infinity instead of from the camera [BS99, Car00]. But, this process, known as Z-FAIL, is usually slighty less efficient than Z-PASS. Shadow volumes do not scale well with the geometric complexity, because more and more shadow quads are created and rasterized. Useless shadow casters [LWGM04, SWK08] can be culled to limit this problem to a certain extent. More importantly, according to the camera and light positions, large shadow quads may be rasterized, saturating the fill-rate. This leads to significant time variations, which are not acceptable for real-time applications, unless the geometric complexity is restricted. This explains why shadow mapping is still widely prefered to shadow volumes in game engines, where the computation stability is essential. However shadow volumes are still investigated because they are pixel-accurate.

For example, Sintorn *et al.* [SOA11, SKOA14] have proposed another approach and implementation. They compute per triangle shadow volumes eliminating the need for connectivity. Instead of rasterizing shadow quads, each shadow volume traverses a 3D hierarchical cluster built over image points to determine their visibility from the light. The method consistently outperforms a shadow volume hardware implementation.

In 1989, Chin and Feiner [CF89] have proposed the Shadow Volumes Binary Space Partitioning (SVBSP) tree. This technique is completely different from the former approach provided by Crow. It builds a BSP tree over the shadow volumes cast by each triangle. Thus the silhouette computation is not needed. A shadow volume is considered as an open and oriented convex bounded by four planes. Each shadow volume is filtered down and merged in a BSP tree using polyhedral set operations [NAT90]. Next points are located in the SVBSP tree to find if they are inside a shadow volume. An extension to support dynamic environments [CS95] of moderate complexity was also proposed. The SVBSP algorithm has several advantages: Like the original shadow volumes, it is pixel-accurate and supports naturally omni-directionnal light sources. In addition, mesh connectivity is not needed and the "eye-in-shadow" position is not a problem. Nevertheless, the SVBSP algorithm was left aside because of a major limitation: It lacks robustness and efficiency because it uses polygon clipping operations, which are prone to numerical inaccuracy. Since the number of clipping operations is not predictable, the memory footprint of the tree can not be predicted either.

Recently, Gerhards *et al.* [GMAG15] introduced the Partitioned Shadow Volumes. This methods inherits all the advantages of the SVBSP algorithm but not the disadvantages. PSV uses a different partitioning strategy and replace the BSP tree by a Ternary Object Partitioning (TOP) tree. Its construction only requires to compute the triangle positions with respect to a shadow volume plane, without any clipping. This eliminates the robustness issues, and improves memory efficiency. A TOP tree has a predictable and fixed memory footprint in $O(n)$, for n triangles. It can be built in parallel in $O(n\,log(n))$, and locating a point in the ternary structure costs $O(log(n))$. Thanks to these algorithmic properties, PSV allows real-time shadows.

In spite of many works in the literature, Shadow Mapping and Shadow Volumes retain several drawbacks. Innovative approach such as PSV are interesting because they offer new options. However PSV have been little explored. Thus, in the next section, we analyse this technique and identify two issues that may limit practical applications.

3. Analysis and solutions

This section starts with a brief overview of the PSV. Next, we focus on the TOP tree query. As noticed by the authors [GMAG15], this is the most computational part of the technique with models up to one million triangles. We deepen our analysis to explain why PSV can struggle with lit points in some situations. We also recall the problem related to the stack used in the original implementation. Then, the section presents our three solutions to solve these problems. At first, we add to the TOP tree nodes a depth value allowing to accelerate the most computational visibility queries. Next we introduce a new stackless algorithm to query a TOP tree. At last, we define a hybrid query, as the combination between a short stack query and a stackless query, to take advantage of both solutions.

3.1. PSV overview

The PSV method relies on the TOP tree which stores the visibility information from a light source. For each frame, PSV uses two steps. The first one builds the TOP tree over the shadow volumes cast by each triangle from the light. The TOP tree is stored in a buffer on the GPU. The second step queries the TOP tree to compute the shadows (see Figure 1). The technique runs completely on the GPU: the first step is done in a Compute Shader, the second one in a Fragment Shader. A TOP tree node contains a plane and three children:

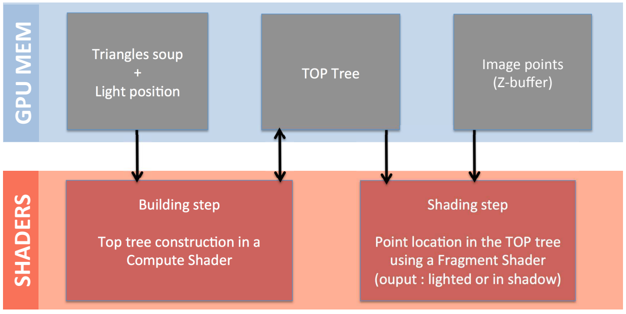

Figure 1: *Per frame overview of the PSV method: In a Compute Shader, a TOP tree is built over the shadow volumes cast by the triangles. Next a Fragment Shader is applied to filter down each image point in the TOP tree and to find if it is visible from the light.*

- A positive child containing all the geometry completely in the positive side of its plane.
- A negative child containing all the geometry completely in the negative side of its plane.
- An intersection child containing all the geometry intersected by its plane.

Shadow volume representation: Each shadow volume is represented by a TOP subtree as illustrated by Figure 2. The negative leaf is the volume in shadow, while the positive leaves are outside the shadow. Since the intersection leaves contain geometry that overlap the positive and negative half-space of a plane, this geometry may intersect the shadow volume boundaries.

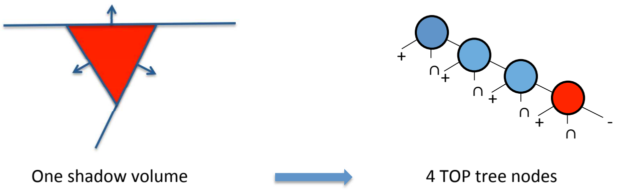

Figure 2: *Left: a shadow volume seen from the light. Right: the corresponding TOP subtree. The three first nodes contain a shadow plane defined by the light position and one triangle edge. The fourth node contains the supporting plane of the triangle.*

Building step: The TOP tree is built in a Compute Shader. Each instance filters down a triangle from the TOP tree root, initially empty. At each node, the triangle descends in one of the 3 children according to the triangle position with respect to the node plane. When a leaf is reached, it is replaced by the subtree representing the shadow volume cast by the triangle.

Algorithm 1 PSV query: It locates a point in a TOP tree to find if it is inside at least one shadow volume.

```
1:  PSV_query(Node $root$, Point $p$)
2:  NodeStack $stack$
3:  Node $n = root$
4:  while $n$ is not $null$ do
5:      int $location \leftarrow$ sign($n.plane$, $p$)
6:      // if the intersection child needs to be visited, it is
7:      // pushed on the stack to be processed later
8:      if $n.inter$ is not a leaf AND isInsideWedge($n.inter$, $p$) then
9:          push( $stack$, $n.inter$ )
10:     end if
11:     if $location > 0$ then
12:         if $n.pos$ is not a leaf then
13:             $n = n.pos$ // continue in the positive child
14:         else
15:             // n.pos is a leaf, pop the stack to search for occlusion
16:             $n = stack$ is empty ? $null$ : pop($stack$)
17:         end if
18:     else
19:         if $n.neg$ is not a leaf then
20:             $n = n.neg$ // continue in the negative child
21:         else
22:             return 0 // early termination case: p is in shadow
23:         end if
24:     end if
25: end while
26: return 1 // the stack is empty, p is outside any shadow volume
```

Shading step: Once the TOP tree is built, the shadows are computed using a Fragment Shader. Each image point is located in the TOP tree to find if it is inside a shadow volume. For the clarity of the presentation, Algorithm 1 recalls this query without detailing the so called "wedge optimization" proposed by Gerhards *et al.* [GMAG15]. While we still use it, we will not make any reference to this optimization since our work does not depend on it. Starting from the root node, the query continues in the positive or negative child (line 5) according to the point position with respect to the plane. However, since the intersection child overlaps, it may be pushed on a stack (lines 8-9) to be processed later. As soon as the point reaches a negative leaf (line 22), it is in shadow. But if the point reaches a positive leaf (line 16), the query keeps searching for an occlusion among the intersection nodes remaining on the stack.

3.2. PSV analysis

We now focus on the TOP tree query used in the shading step. Algorithm 1 highlights a first problem: As soon as the point is found inside a shadow volume, the query ends (line 22). This is an early termination case. But if the point is not inside any shadow volumes, it is only when the stack has been exhausted that the point can be claimed visible from the light (line 26). As a consequence, the query is better at finding points in shadow than lit points. To visualize this behavior, we have calculated heat maps illustrating the number of visited nodes per query (*i.e.* per image points). These heat maps are shown in the PSV column in Figure 6.

Clearly, lit points are much more expensive than points in shadow. And surprisingly, the most expensive ones are often closer to the light than any other geometry. Obviously, those points can not be occluded. In section 3.3, we propose to limit this behavior by using a depth value per node: the minimum distance of a subtree to the light source.
A second problem is related to the PSV query implementation, especially on a parallel architecture. As noticed by Gerhards *et al.*, it uses a stack whose size is problematic. It must be large enough to avoid a stack overflow, but as small as possible to alleviate the register pressure. In section 3.4, we define a stackless algorithm by connecting the TOP tree nodes to their common intersection node ancestor. Then, in section 3.5 we explain why it is worth combining both stackless and stack-based queries.

3.3. Adding depth

Among the two problems emphasized by Section 3.2, the first one concerns the computational cost of lit point queries. To prove that a point is in shadow, it is sufficient to find at least one shadow volume containing the point. But to show that the point is lit, the query has to determine that it is outside all the shadow volumes. Such a behavior is well-known in logic programming: to demonstrate a theorem consists in invalidating all the negation cases. Then a theorem failure is answered faster than the theorem proof. In such a case, a solution is the constraint logic programming: it consists in adding some constraints to avoid exploring all the possible cases [JL87].

In the same spirit, we propose to add a depth constraint on each nodes to avoid visiting subtrees which only contain geometry further from the light than the queried point. Figure 3 explains why the PSV query struggles with lit points, especially those in front of many triangles with respect to the light source. In this case, a depth information is missing which would allow an early detection that no occlusion can be found in a subtree. Hence, we add to each PSV node its depth from the light: Since a TOP tree is a partition of the shadow volumes cast by a set of triangles, its depth is defined as the shortest distance between those triangles and the light. This requires a minor modification to the TOP tree construction as detailed in Algorithm 2. Thanks to this depth information, it becomes possible to compare the point distance to a subtree distance. If the point distance is smaller than the subtree distance, it is outside all the shadow volumes contained in the subtree. For example, on Figure 3, point p would be found lighted from the start, at the root node.

Notice that this modification is compatible with all our query algorithms presented in this paper (stack-based, stackless and hybrid). This optimization is included in Algorithm 3 (blue lines) which presents our stackless query.

3.4. Stackless query

To our best knowledge, no stackless algorithm for TOP tree exists so far. In a ray-tracing context, stackless ray traversals were proposed for binary data structures. Indeed, traditional stack-based ray traversals require a stack for each ray with an increase in memory, especially on parallel architectures. For sparse trees, parent pointers are required in each node to ascend in the trees [HDW*11]. An ar-

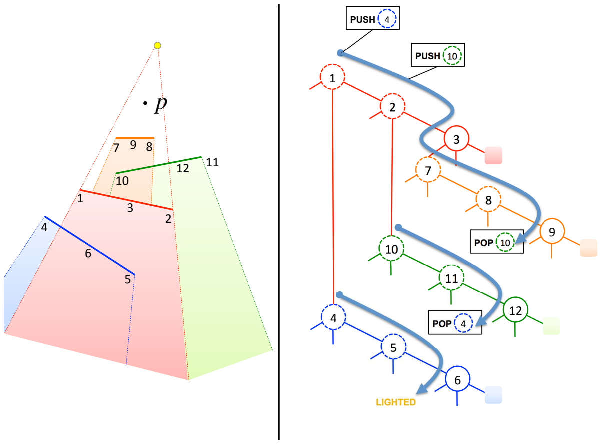

Figure 3: *Left: A 2D example with line segments instead of triangles. Obviously, the point p can not be shadowed since it is closer to the light than any other occluders. Right: a TOP tree corresponding to this geometric configuration, and the path followed by the PSV query to compute the visibility of p. The data structure does not contain any depth information from the light. Thus the query can not quickly detect that p is in front of all the occluders. In this example, the tree is entirely scanned to determine that p is lighted.*

Algorithm 2 Modified TOP tree construction. It enables depth test and stackless query and it is still compatible with a stack-based query. The first change (in green) adds a depth value to each node, i.e. the shortest distance from the light to the geometry in the subtree (see 3.3). The second change (in blue) allows the TOP tree to support our stackless query (see 3.4).

```
TOP_mergeSV(Node $root$, Triangle $t$, Light $l$)
Node $node = root$
// [depth] shortest distance from the light to the triangle
float $d = minDistance(t, l)$
// [stackless] null is a sentinel value
Node $ancestor\_parent = null$
while $node$ is not a leaf do
    // [depth] t belongs to the subtree whose root is node. The subtree
    // distance is updated if the triangle distance is smaller.
    $node.distance = min(node.distance, d)$
    float $location$ = position( $node.plane$, $t$ )
    if $location > 0$ then
        $node = node.pos$
    else if $location < 0$ then
        $node = node.neg$
    else
        // [stackless] t goes in an intersection subtree.
        // We set "node" as the parent node of this subtree.
        $ancestor\_parent = node$
        $node = node.inter$
    end if
end while
Node $sv$ = initSubtreeRepresentation($t$, $l$, $ancestor\_parent$)
replaceLeafBySubtree($node$, $sv$)
```

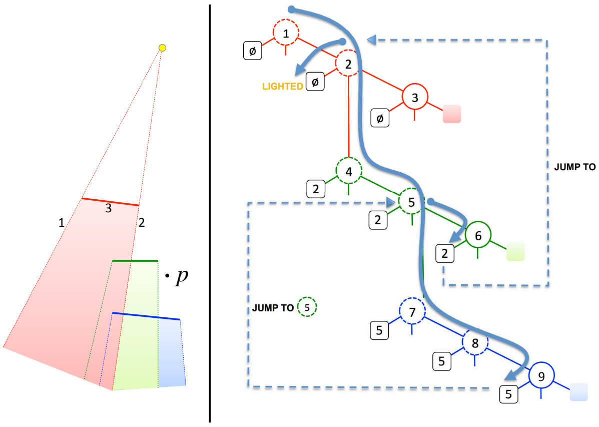

Figure 4: *Stackless query example. Left: A 2D simple geometric configuration using line segments instead of triangles. Right: A corresponding TOP tree. The green and the blue segments are intersected by the shadow plane* 2*, thus they belong to the intersection subtree of node* 2*. Similarly, the blue segment is intersected by the shadow plane* 5 *and thus it belongs to the intersection subtree of node* 5*. Each node points to the parent of the intersection subtree which it belongs to. To locate p in this TOP tree, the stackless query starts from the root node* 1*. The query descends at first in the intersection child, if needed. When the query reaches the positive/visible leaf of node* 9*, it jumps directly to the node* 5*. Since it is the second time that node* 5 *is tested, we know that the intersection subtree has already been checked, and so the query continues in the positive or negative subtree. Similarly, when the query reaches the positive leaf of node* 6*, it jumps to node* 2 *to test its positive and negative children. Finally the query locates p in a positive leaf with a null link since it does not belong to any intersection subtree. Thus p is visible from the light.*

ray of bits may also be used to keep track of the traversal [BAM13]. Another solution uses a short stack and encodes a restart trail in a bit mask [Lai10]. However, a TOP tree is a ternary data structure, not a binary one. This means that the solutions using an array of bits are not applicable. More importantly, the PSV query and a ray traversal are two different problems. The first one locates a point and outputs a boolean (lit/shadow), thus the scan order is not relevant. The second one uses a ray and searches for its first intersection, thus the scan order is important. In this section, we proposed two stackless queries tailored for a TOP tree. While some similarities exist with the previous works on stackless ray-traversals, our algorithms are not directly comparable.

Using the stack-based query, if the intersection child of a node n needs to be visited, it is pushed on the stack to be processed later. Without a stack, the query will have to backtrack until n after its positive or negative child have been explored. As mentionned above, a common solution to enable backtracking is to add a parent pointer to each node. Our first experiments used this solution, but they achieve very poor performances, below real-time rendering requirements. When such a query ascends in the tree, it visits too many nodes, and too many memory accesses are made, leading to a memory bandwidth overhead. Instead, we can take advantage of the TOP tree structure. We observe that the probability for a triangle to be intersected by a plane is low, especially with fine tessellated models. Thus, the probability for a node to be the intersection child of its parent is also small. We can save the memory bandwidth by jumping from a node to its first intersection ancestor instead of backtracking node by node. Figure 4 gives an example, assuming that each node is marked with a pointer to the parent of its first intersection ancestor.

In practice, we need to slightly modify the TOP tree building algorithm to mark each node with the appropriate link. This is detailed in Algorithm 2. To merge a given shadow volume cast by a triangle, the construction algorithm starts from the root node (line 2) which has no ancestor. Thus the ancestor link of the root is initialized with a sentinel value (line 6). Next the algorithm is similar to [GMAG15]. The triangle position is tested against the node plane (line 11). According to the test, the triangle continues in the relevant subtree. When the triangle is intersected by the plane (line 16), its ancestor link is updated (line 19). When the triangle reaches a leaf, the TOP tree representation is computed (line 23), marking its nodes with the parent of the intersection subtree which the triangle belongs to. At last the shadow volume cast by the triangle is merged to the data structure (line 24).

Algorithm 3 describes the stackless query using such a TOP tree. A state variable (line 3) indicates if a node is visited for the first time. In such a case, the intersection child is tested (line 16) and, if required, the query descends in the intersection subtree (line 17). When the node is visited for the second time (the query has just jumped following an ancestor link), we know that the intersection child has been tested already. Then, the query continues either in the positive child (line 24) or in the negative child (line 32). When a positive leaf is reached, the query ends visiting a subtree without finding any occlusion (line 25). Thus, following the ancestor link in the node (line 27), it jumps to the parent of the current intersection subtree. Eventually, the ancestor link is null, meaning that the subtree root is actually the TOP tree root, and so the point is lit (line 40). For shadowed point, the query still ends as soon as the point is located inside a shadow volume (line 34).

Notice that our stackless query does not visit the same sequence of nodes as the stack-based one. Indeed, the latter descends at first in the positive or negative subtree and next, it eventually visits the intersection subtree. Our stackless query does the opposite. We have experimented a modified stack-based query to use the same traversal order as our new stackless query. We did not noticed any significant performance differences. Thus we kept the original stack-based query as a reference for our comparisons.
In addition each node whose intersection subtree has to be visited is accessed twice: Once to descend in the intersection child, and once to descend in either the positive or the negative child. As a consequence, our stackless query is expected to visit more nodes than the stack-based query. The stackless algorithm requires more memory-bandwidth while the stack-based query has a higher register pressure. At first glance, a combination of the two methods does not seem to be a good idea. The following section explains why it is worth trying.

Algorithm 3 Stackless query algorithm (see 3.4). Blue lines correspond to the depth optimization (see 3.3).

```
TOP_stacklessQuery(Node $root$, Point $p$, Light $l$)
// Variable state: first (false) or second (true) visit to current node
bool $twice = false$
Node $node = root$
float $d = distance(p,l)$ // Distance from the light for depth test
while $node \neq null$ do
    // Depth test: point distance vs subtree distance
    if $d < node.distance$ then
        // p is closer to l than any shadow volumes in the subtree
        // Leave this intersection subtree, jump to the parent of its root
        $node = node.ancestor\_link$
        $twice = true$
        continue
    end if
    // First visit: intersection child test
    if $twice == false$ AND $node.inter$ is not a leaf then
        $node = node.inter$ // Descend in the intersection subtree
        continue
    end if
    // At this point, the intersection subtree is already checked
    // The algorithm continues in the positive or negative subtree
    $twice = false$
    int $location$ = sign($node.plane$, $p$)
    if $location > 0$ then
        if $node.pos$ is a leaf then
            // No occlusion found in this intersection subtree.
            $node = node.ancestor\_link$ // Jump to its parent node.
            $twice = true$ // This will be the second visit for node
        else
            $node = node.pos$
        end if
    else
        if $node.neg$ is a leaf then
            return 0 // p is inside a shadow volume
        else
            $node = node.neg$
        end if
    end if
end while
return 1 // p is outside any shadow volume
```

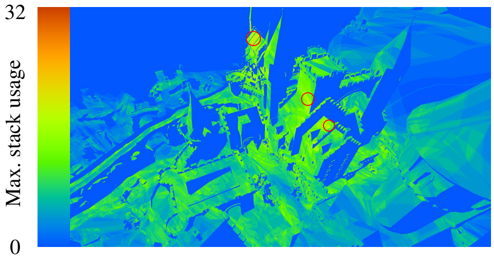

Figure 5: *Stack usage per image points. This heat map shows that the stack usage remains low for most image points. However it can get close to the maximum (32) (inside the red circles).*

3.5. Hybrid query

Figure 5 shows a heat map that represents the maximum stack usage per image point, using the former stack-based query proposed in [GMAG15]. This example is representative of the general behavior. For most queries, the stack usage does not exceed half of its size. As a consequence, using a short stack is enough in most cases. And this would also alleviate the register pressure. However, the red circles on the heat map draw attention to few locations where the stack is almost full. This leads us to combine the two methods, using a short stack and switching to the stackless query when this stack is full.

Ideally on parallel architectures, switching between two algorithms should be avoided, because it can generate SIMD divergence in thread groups. But as illustrated in Figure 5, most of the threads should only keep executing the short-stack based query. In addition to their small number, the "pathological" image points are not spread over the image but always grouped together. Thus when a thread switches to the stackless query, it is probably not the only one in its group, limiting the divergence cost.

The combination of the stack-based and stackless queries is quite straightforward. The result is a hybrid query defined as follows: As long as it is possible to push nodes on the stack, the hybrid query runs the stack-based query. When the intersection child of a node n can not be pushed, the intersection subtree is immediately visited using the stackless query. When it jumps back to node n, the hybrid query switches back to the short stack query to visit either the positive or the negative subtree of n.

3.6. Implementation

Our implementation relies on the former PSV code, available at `https://github.com/PSVcode/EG2015`, and compatible with OpenGL 4.3 and above. Let us recall that, for each frame, the TOP tree is constructed using a Compute Shader and stored in a Shader Storage Buffer Object. The TOP tree queries are implemented as Fragment Shaders applied to a deferred image buffer.

Memory layout: A TOP tree is stored in an array of nodes. The depth information is added to the TOP tree nodes without increasing the node size. Indeed, the former PSV implementation uses 32 bytes per node and stores a shadow volume as 4 consecutive nodes. Thus their negative children are always the node index plus one. Hence, we can spare 16 bytes per shadow volume.
Compared to the PSV memory layout, we need to add a distance and an ancestor link to enable both our depth test and our stackless query. We can notice that a triangle depth holds for the four nodes of its shadow volume. Similarly, a shadow volume belongs to one intersection subtree, so its four nodes share the same ancestor link. Thus it is sufficient to store the depth and the link information once every four nodes. This required 8 of the 16 bytes available without increasing the memory cost.

Compute Shader: Each Compute Shader instance merges one triangle into the tree. To solve concurrency, Gerhards *et al.*'s building algorithm relies on atomic operations. Our modified construction creates new concurrent accesses between instances to update the depth values. We solve this using an atomic minimum. As explained in the next section, this generates a small overhead compared to the former construction.

Fragment Shader: Our depth, stackless and hybrid queries are

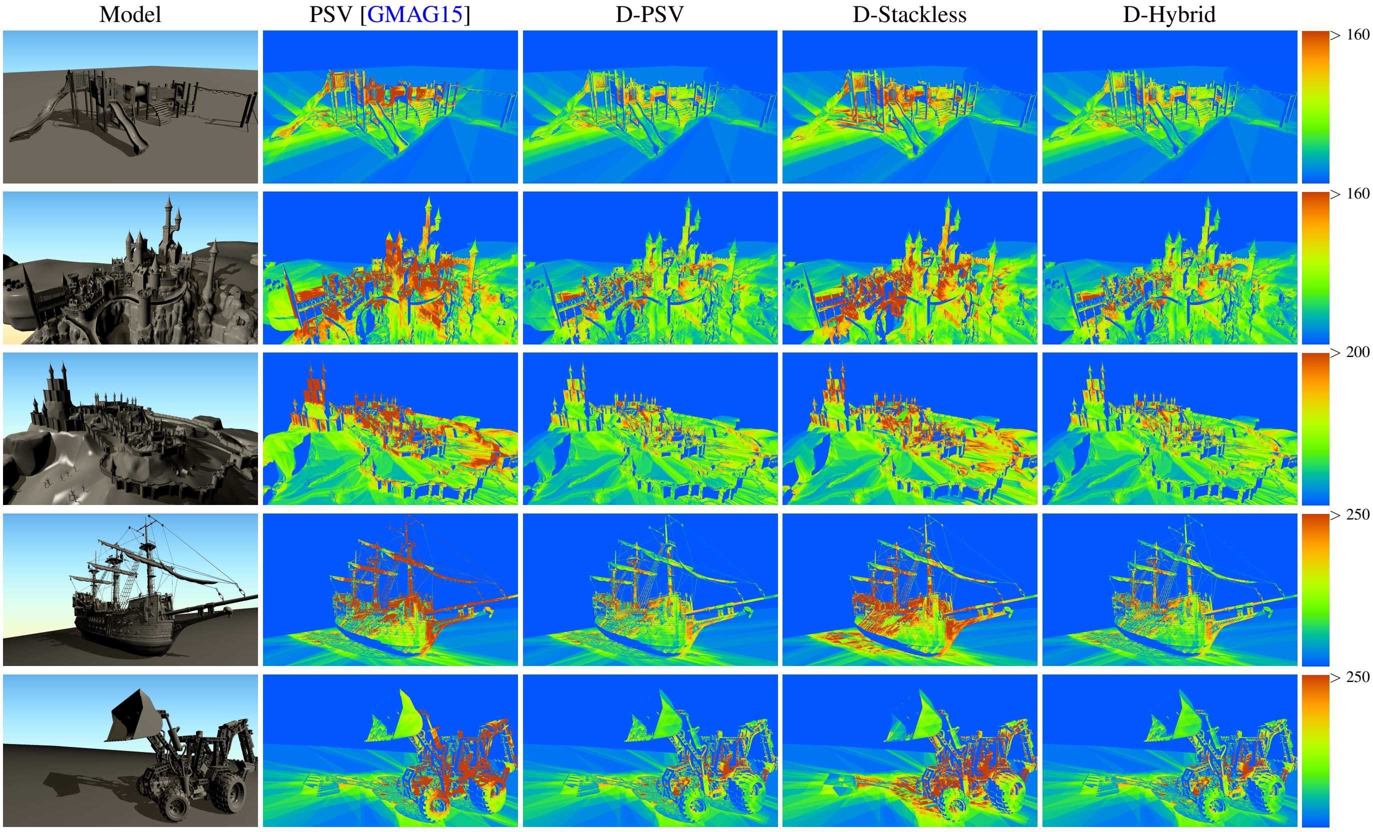

Figure 6: *Heat maps visualizing the number of visited nodes per pixel/query, using the former PSV method, and our three new algorithms.*

implemented in Fragment Shaders. Since the depth is unique per shadow volume, our depth test is only done when the query descends in a positive or an intersection child and thus, when it encounters a new shadow volume.

Our modified Compute Shader and our Fragment Shaders implementing the stackless queries are also available at `https://github.com/PSVcode/EGSR2016`.

4. Results

Our experiments are done using a NVIDIA GTX 980 at a resolution of 1920×1080. Five models are used. The PLAYGROUND scene (131K triangles) is the smallest one but casts shadows of various complexities. The EPIC CITADEL scene (393K triangles) is a game level from the Epic UDK. The CLOSED CITY model (623K triangles) is a large open scene including large scale and detailed features. The SHIP model (956K triangles) has many fine details casting complex shadows. The EXCAVATOR model (1 130K triangles) is a complex modeling of a Lego® bricks excavator-bulldozer.

We compare different methods. Our reference method is PSV: It refers to the former PSV algorithm using the source code publicly available. D-PSV is the same algorithm, but using our depth test. This allows to evaluate the impact of the depth test regardless of the stackless query. D-Stackless is our stackless query including depth test. D-Hybrid refers to our hybrid algorithm with depth test. PSV and D-PSV use a stack size of 32 nodes, as in [GMAG15], except for EXCAVATOR which uses a stack size of 48. Indeed, stack overflows occur with a smallest value, creating visual artifacts. D-Hybrid uses a short stack of 12 nodes (we discuss this value in section 4.4). Front face culling is always enabled, and we also add a frustum culling test, defined as the convex hull of the view frustum plus the light. Triangles outside this volume can not cast shadows visible from the camera. Notice that this test is done in the compute shader on the GPU, triangles are never culled before being sent to the graphics card. The frustum culling test is only useful when all the geometry does not contribute to the picture.
At last, we have also implemented a second state of the art technique: A z-pass implementation of the traditional Shadow Volumes referred as Z-PASS. All our experiments are done using 2 500 animation frames per scene.

4.1. Memory

As explained in Section 3.6, the memory consumption does not change compare to the former PSV implementation. Thus, the memory cost is the same whatever the query algorithm. For each scene with n triangles, an array of $32 \times 4 \times n$ nodes is allocated (each triangle generates 4 nodes and each node costs 32 bytes). Thus the memory consumption for our models are: 16 MB (PLAYGROUND), 48 MB (EPIC CITADEL), 76 MB (CLOSED CITY), 117 MB (SHIP), and 138 MB (EXCAVATOR).

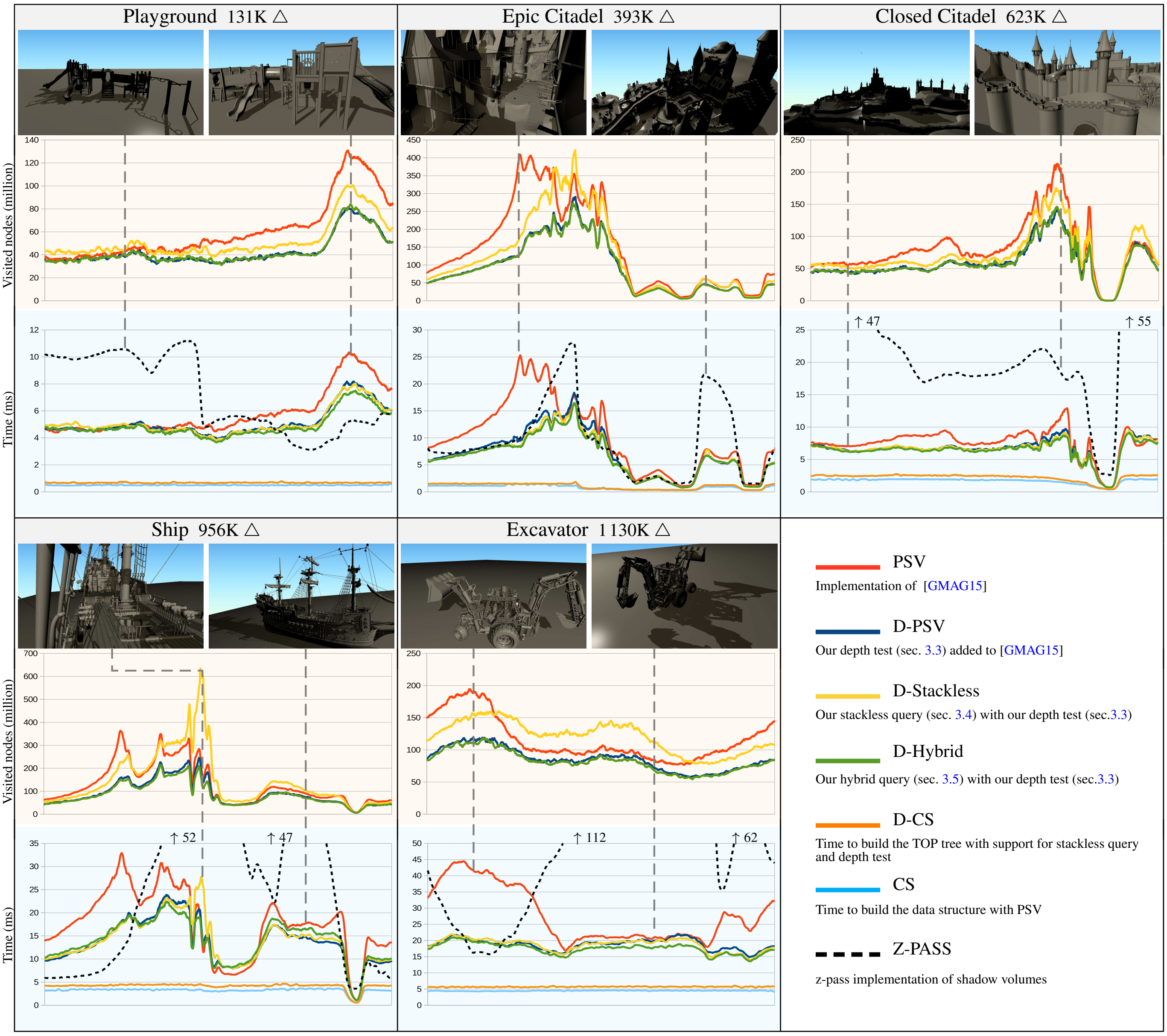

Figure 7: *Number of visited nodes and computation times measured over* 2 500 *frames animations, using our new query algorithms, and compared to former PSV method and a Z-PASS implementation.*

4.2. Algorithmic comparison

Our first experiments measure the number of nodes visited per query and per frame. Figure 6 presents the representative heat maps for each scene. The difference between the PSV and the D-PSV columns illustrates the depth test impact. For shadowed points, the number of visited nodes does not seem to change. But for lit points, it is significantly reduced. As expected, D-Stackless visits more nodes than D-PSV but less than PSV due to the depth test. At last D-Hybrid is comparable to D-PSV. However, the former uses a short stack of 12 nodes and switches to the stackless query to prevent any stack overflow, while D-PSV does not provide any guarantees.

Figure 7 provides a more detailed insight. For each scene, the upper graph (light orange background) corresponds to the number of nodes visited by each query per frame. It confirms that the behavior described above is consistent throughout all the animations. Notice the view frame associated to a peak for the PSV query: the light is behind and close to the camera. This generates a configuration similar to the pathological example given by the Figure 3 in section 3.3. PSV struggles with the lit points because it is unable to find that they are closer to the light than the triangles. On the contrary, our new queries benefit from the depth test in the same situation. While D-PSV and D-Hybrid always visit less nodes than PSV, D-Stackless can test more nodes. As mentioned in section 3.4,

	Average number of visited Nodes				Average computation time (σ) in ms				
	Percentage of PSV				worst / average / best acceleration factors compared to PSV				
	PSV	D-PSV	D-Stackless	D-Hybrid	PSV	D-PSV	D-Stackless	D-Hybrid	Z-PASS
Playground	61	43	52	43	5.53 (1.66)	4.89 (1.04)	4.88 (1.0)	4.66 (0.91)	6.79 (2.8)
	-	71	86	71	-	0.95 / 1.13 / 1.31	0.89 / 1.13 / 1.36	0.93 / 1.19 / 1.43	0.43 / 0.81 / 2.36
Epic Citadel	167	104	141	103	9.95 (6.57)	7.08 (4.40)	6.67 (3.98)	6.42 (3.79)	9.38 (6.93)
	-	62	84	62	-	0.93 / 1.41 / 2.67	0.98 / 1.49 / 2.96	1.06 / 1.55 / 3.05	0.30 / 1.06 / 2.54
Closed Citadel	85	60	72	60	7.61 (1.79)	6.60 (1.32)	6.53 (1.22)	6.34 (1.20)	24.6 (8.98)
	-	71	85	70	-	0.83 / 1.15 / 1.41	0.81 / 1.17 / 1.48	0.90 / 1.20 / 1.53	0.13 / 0.31 / 0.84
Ship	142	100	161	93	17.58 (6.62)	13.56 (4.22)	13.95 (4.40)	13.92 (3.72)	25.99 (16.34)
	-	71	114	66	-	0.81 / 1.30 / 1.81	0.45 / 1.26 / 1.86	0.78 / 1.26 / 1.82	0.13 / 0.68 / 3.19
Excavator	116	87	125	83	27.87 (9.09)	18.83 (1.51)	18.66 (1.57)	17.52 (1.49)	57.19 (33.29)
	-	75	106	72	-	0.99 / 1.48 / 2.15	1.02 / 1.49 / 2.10	1.13 / 1.59 / 2.17	0.13 / 0.49 / 2.64

Figure 8: *Number of visited nodes (on the left) and computation times (right) using our new queries, and using the previous implementation of PSV, and a classical Z-PASS. σ is the standard deviation of the average computation times per frame.*

this was expected. The depth test slightly hides this behavior on the first three models. It is more noticeable on SHIP and EXCAVATOR, our two most complex models.

Figure 8 summarizes our results and gives the average performances. The left part (light orange background) of the table corresponds to the number of visited nodes. D-Hybrid has the best average behavior and visits 62% to 72% of the nodes visited by PSV, while D-Stackless visits 84% to 114%.

4.3. Performance comparison

Figure 7 provides the computation times for each method, in the lower graphs (light blue background). Notice that Z-PASS aside, all the computation times include the TOP tree construction done for each frame. The time exclusively spent by the Compute Shaders to build the TOP trees are represented by the CS and D-CS plots. The former is the Gerhards *et al.*'s algorithm, while the latter is our modified algorithm to enable depth test and stackless queries. These two plots variations illustrate the contribution of the frustum culling test. Our modified construction has a small (but acceptable) overhead because it requires an additional atomic operation. Computation times for D-Hybrid, D-PSV and D-Stackless are very closed. They consistently outperform PSV, especially in the pathological cases discussed above. We could have expected D-Stackless to be slower particularly on the EXCAVATOR model, where it visits more nodes than any other query. D-Stackless has the minimum register pressure. In this case, it appears to balance the increase in visited nodes.

Computation times for Z-PASS vary widely and can become very important as the number of triangles increases. Shadow Volumes does not scale well with the geometric complexity. Comparing the Z-PASS behavior to the other methods, we notice a kind of duality. The two view frames from the EXCAVATOR animation are representative: The first one corresponds to a pathological case for the other methods, contrary to Z-PASS which is faster in the same situation. Indeed, when the light is behind the camera, the shadow quads projection get smaller and are processed efficiently by the hardware rasterizer. The second view frame is conversely a pathological case for Z-PASS, because it creates large and elongated shadow quads increasing the fill rate. The other methods remain efficient because many points are in shadows, leading to an early termination as explained in section 3.2. Shading a point involves a camera and a light. Z-PASS solves the problem from the camera, PSV solves the problem from the light. This duality is noticeable in our experiments.

With the SHIP model, we can notice that the original PSV query is slighty faster at the begining of the second part of the animation. This corresponds to frames where shadows are visible, but not the shadow casters (the camera focuses on the shadow of the ship cast on the ground, but the ship is outside the view frustum). In this specific configuration, the image points are all further from the light than the geometry. This means that the depth test in our new queries always fails: D-PSV or D-Hybrid visit exactly the same nodes than the original PSV query. This is shown by the upper graph for the corresponding frames. Thus, the small computation overhead for D-PSV and D-Hybrid is the cost of the depth test during the building step and the shading step. The same holds true for D-Stackless, although the number of visited nodes is slighty more important due to the different traversal order. However, all queries remain very efficient in such configurations, because most of the image points are in shadow, leading to an early termination.

Figure 8 provides the average computation times throughout the animation and different statistics in its right part (light blue background). Except for SHIP (we discuss this exception in the next section) D-Hybrid has the best performance on average, with a smaller standard deviation: It is the fastest and the most stable, which is also important in real-time rendering. Compared to PSV, the maximum acceleration factors recorded during the animations correspond also to D-Hybrid (from 1.43 on PLAYGROUND to 3.05 on EPIC CITADEL). However we can notice that D-Stackless and D-PSV have very close computation times.

Finally this experiments illustrate that our depth test improves significantly the PSV algorithm. This is true for the three query algorithms which use the depth test, regardless of their stack-based or stackless nature. They provide fast and close computation times with an advantage to D-Hybrid. Moreover D-Hybrid is also better for two other reasons: it visits less nodes than D-Stackless, and it cannot generate a stack overflow contrary to PSV and D-PSV.

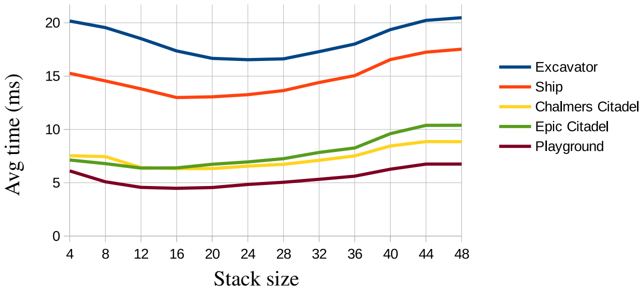

Figure 9: *Average times using D-Hybrid with different stack sizes.*

4.4. Limitations

For comparison purposes, our experiments with D-Hybrid use the same short stack size (12) with all the models. However we have also done the same experiments, increasing the stack size from 4 to 48. For each scene, Figure 9 shows the average time according to the stack size. While a size of 12 leads to the best results for our first three models, a size of 16 is more relevant for SHIP while a size of 20 will improve again the results on EXCAVATOR. With SHIP the average time per frame is 13.92 with a size of 12 (see Figure 8) but drops to 12.99 with a size of 16, improving over D-PSV (13.56). For EXCAVATOR, the average time is 17.52 with a size of 12 and 16.66 with a stack of 20. Obviously, more complex models require a more important stack usage. Using a fixed size of 12, D-hybrid has a loss of efficiency because it switches too many times in stackless mode. Hence the stack size needs to be increased to benefit from the best balance point. As a drawback, the register pressure increases and at some point, it will become equivalent to D-PSV. Nevertheless, D-Hybrid has always the advantage to prevent any stack overflow. Finally its stack size can also be used to favor either memory or computation time.

Our results illustrates the depth test efficiency. One may wonder if a similar or even better result could be achieved if triangles are inserted in the TOP tree following their depth order from the light source. Currently this is not compatible with the TOP tree construction proposed by Gerhards *et al.*, since it is randomized and can not support a constraint insertion order. In addition, this randomized insertion order of the triangles prevents the TOP trees from being completely unbalanced [GMAG15]. A constraint order will call into question this important property.

5. Conclusion

PSV is a recent algorithm allowing pixel-accurate and real-time shadows. While it has been little explored, it is an interesting option with respect to mature techniques such as shadow volumes or shadow maps. In this article, we have studied the PSV algorithm in order to improve it. We have underlined two drawbacks and we have provided several solutions.

At first, we have explained why PSV is by nature less efficient to process lit points than points in shadow. In some situations, this behavior can cause a significant computation overhead. Thus, we have proposed a modified algorithm to add a depth information to the TOP tree used by PSV. Thanks to this depth value, the algorithm is able to detect when a point is closer from the light than the geometry in a TOP subtree. In such a case, the point can not be occluded by the geometry and the algorithm can skip over the subtree. Our experiments show that this modification improves consistently the performances, especially in the most difficult situations for the original PSV.

Next, we have focused on the original PSV query implementation that requires to tune carefully a stack size. Instead, we have slighty modified the TOP tree nodes to enable stackless queries, without increasing the memory cost. We have proposed a stackless query and a hybrid query which takes advantage of the stackless query combined to a short stack one. It provides generally the best results in our tests.

Finally, PSV is more efficient and more stable using our improvements.

As a future work, it would be interesting to focus on the TOP tree construction. Since it is randomized, it does not allow any control over the process. Thus a different algorithm may introduce some heuristics to improve the data structure quality. The challenge will be to improve the quality without slowing down the TOP tree construction.

6. Aknowledgments

The PLAYGROUND and EXCAVATOR models are from the 3D Warehouse under the General Model License. The SHIP scene is available at Blend Swap under the creative commons license, and was modeled by Chris Kuhn and Greg Zaal. The EPIC CITADEL scene is distributed with the Unreal Development Kit by Epic Games. The CLOSED CITADEL is courtesy of Erik Sintorn.

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

[BAM13] BARRINGER R., AKENINE-MÖLLER T.: Dynamic stackless binary tree traversal. *Journal of Computer Graphics Techniques (JCGT) 2*, 1 (March 2013), 38–49. 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. 1, 2

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 242–248. 1, 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. 2

[Eng06] ENGEL W.: *Shader X5: Advanced Rendering Techniques*. Charles River Media, Inc., 2006. 2

[ESAW11] EISEMANN E., SCHWARZ M., ASSARSSON U., WIMMER M.: *Real-Time Shadows*. A.K. Peters, 2011. 1

[FFBG01] Fernando R., Fernandez S., Bala K., Greenberg D. P.: Adaptive shadow maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 387–390. 2

[GMAG15] Gerhards J., Mora F., Aveneau L., Ghazanfarpour D.: Partitioned shadow volumes. *Computer Graphics Forum 34*, 2 (2015), 549–559. 1, 2, 3, 5, 6, 7, 8, 10

[HDW*11] Hapala M., Davidovič T., Wald I., Havran V., Slusallek P.: Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics* (New York, NY, USA, 2011), SCCG '11, ACM, pp. 7–12. 4

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

[JL87] Jaffar J., Lassez J.-L.: Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1987), ACM, pp. 111–119. 4

[JLBM05] Johnson G. S., Lee J., Burns C. A., Mark W. R.: The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4 (Oct. 2005), 1462–1482. 2

[Lai10] Laine S.: Restart Trail for Stackless BVH Traversal. In *High Performance Graphics* (2010), Doggett M., Laine S., Hunt W., (Eds.), The Eurographics Association. 5

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

[NAT90] Naylor B., Amanatides J., Thibault W.: Merging bsp trees yields polyhedral set operations. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 115–124. 2

[SBE16] Scandolo L., Bauszat P., Eisemann E.: Compressed multiresolution hierarchies for high-quality precomputed shadows. *Computer Graphics Forum (Proc. Eurographics) 35*, 2 (May 2016). 2

[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

[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, USA, 2011), 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

[WHL15] Wyman C., Hoetzlein R., Lefohn A.: Frustum-traced raster shadows: Revisiting irregular z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2015), i3D '15, ACM, pp. 15–23. 2

[Wil78] Williams L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 270–274. 1

[ZSXL06] Zhang F., Sun H., Xu L., Lun L. K.: Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications* (2006), ACM, pp. 311–318. 2