

# The power of a two-sided depth test and its application to CSG rendering and depth extraction

Category: Research Paper

## Abstract

Shadow mapping is a technique for doing real-time shadowing. Recent work has shown that shadow mapping hardware can be used as a *second depth test* in addition to the z-test. In this paper, we explore the computational power provided by this second depth test, by demonstrating its utility in two separate applications.

We first examine the problem of rendering objects described using CSG (Constructive Solid Geometry) operations. This problem has been studied since the mid-80s, and we provide an algorithm that asymptotically improves the number of rendering passes required to display a CSG product by a factor of  $n$  by exploiting the two-sided depth test.

We then examine the problem of selecting specific depth layers in a scene, and show how this can be done in  $O(\log n)$  passes. Further, we demonstrate for the first time lower bounds on the computations that can be performed by the graphics pipeline. Specifically, we show that the two-sided depth test is crucial for the above depth picking application, by demonstrating that without this operation, most natural hardware-based algorithms would require  $n$  rendering phases.

**Keywords:** Constructive solid geometry, z-buffer, OpenGL, shadow mapping, stream processing, lower bounds

**CR Categories:** I.3.3 [Computer Graphics]: Picture/Image Generation I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

## 1 Introduction

In recent years, the increased power of graphics rendering hardware has led to the use of the graphics pipeline for general purpose stream computations. One of the early examples of this was the use of hierarchical z-buffering for visibility calculations [9], and subsequently in programmable vertex shaders [19, 14, 20]. Other uses of the graphics pipeline as a general purpose stream computing engine have been demonstrated in computational geometry [12], robotics [11], and numerical analysis [13].

In a recent development, work by Everitt *et al.* [6] has shown that shadow mapping hardware (supported in the nVidia GeForce3 and newer architectures) can be used to perform order-independent transparency. They demonstrate this by using the shadow mapping phase in the pipeline to filter out fragments that have a z-value less than (or greater than) values stored in a depth texture. This operation, combined with the standard z-test, provides a *two-sided depth test* on fragments, and is exploited in a technique they call *depth peeling* that can *peel* off layers of a scene one by one. Interestingly enough, the idea of using two-sided depth tests to implement depth peeling was proposed earlier by Mammen [15], who used the idea of *virtual pixel maps*. The key observation by Everitt *et al.* was that existing shadow mapping hardware can be used to simulate this test. In this paper, we study the computational power that the *two-sided depth test* provides over the standard pipeline.

## 1.1 Our Contributions

**Rendering CSG Trees** We first examine the fundamental problem in solid modelling of rendering objects presented as CSG (constructive solid geometry) trees. Goldfeather *et al.* [7, 8] demonstrated how graphics hardware could be used to render CSG trees. The Trickle algorithm [3, 21] used depth-interval buffers (a kind of two-sided depth test) to render CSG trees efficiently. More recently, the Goldfeather *et al.* algorithm was implemented on modern graphics systems by Wiegand [25], followed by improvements due to Stewart *et al.* [22, 23, 24] and Erhart and Tobler [4].

We show that the two-sided depth test can be used to render CSG trees with a factor of  $n$  fewer passes than the best known OpenGL-based algorithms<sup>1</sup>. Our algorithm can render arbitrary simple objects, and does not require the explicit precalculation of *levels* that prior results did. Our algorithm works by performing a *topological sweep* over the arrangement of the objects; this technique may be of independent interest.

**Depth Picking** In our second application of the power of the two-sided depth test, we generalize the *depth peeling* technique of Everitt *et al.* to the problem of selecting an arbitrary level in a scene. Depth peeling uses the two-sided depth test to extract fragments at different distances from the viewpoint (the nearest, second nearest etc.). Using this method, one may extract the  $k^{th}$  level (the fragments that are  $k$ -closest) of a scene in  $k$  passes. However this approach is inefficient in many applications where  $k$  may be large. For example, a common problem in computational geometry is extracting the *median*  $= (n/2)^{th}$  level of an arrangement of objects. In this case, depth peeling would require  $n/2$  passes.

We show that we can *pick* any level in a collection of objects with a randomized algorithm. The algorithm computes the level without any error and the number of passes required varies, but is  $O(\log n)$  with high probability. Moreover, we show that randomization and the second depth test are **crucial** to the performance of this algorithm, by proving that if either of the two features are omitted, then any algorithm (from a very general class) must take  $\Omega(n)$  passes. Note that the use of the second depth test gives us an *exponential* speedup in the number of passes required.

The two problems above are closely related; walking through an arrangement level by level is a generalization of finding the “next” level. The problem of finding the  $k^{th}$  level is the problem of “jumping” to a specific level in the arrangement.

## 1.2 Paper Outline

We discuss prior work in Section 2. We define the problem of rendering a CSG tree in Section 3, and present our solution in Section 4. In Section 6, we demonstrate a lower bound for depth peeling in the absence of a two-sided depth test. In Section 8 we present our algorithm for depth picking, and prove corresponding lower bounds in Section 9.

---

<sup>1</sup>Epstein *et al.* [3] and later Rossignac and Wu [21] were the first to suggest the use of two-sided depth tests for performing CSG rendering. Although their algorithms pre-date OpenGL-based architectures, we believe that the number of rendering passes required by their algorithm would be similar to that used by our algorithm.

## 2 Prior Work

There has been extensive work on the problem of rendering solid objects defined in terms of CSG trees. A general survey of CSG methods is beyond the scope of this paper. We will focus solely on methods that make use of the z-buffer (and graphics hardware in general).

Goldfeather *et al.* [7] presented an algorithm for rendering a CSG tree of *convex* objects (and subsequently [8] for non-convex objects) using an extension of the Pixel-Planes graphics hardware. This algorithm was refined and implemented on modern graphics hardware by Wiegand [25]. The running time of the algorithm, expressed as the number of rendering passes required, is essentially quadratic in the number of objects (the running time also includes a quadratic term that depends on the *convexity* of the objects).

The Trickle algorithm [3, 21] developed by Epstein *et al.*, and later refined by Rossignac and Wu, takes a different approach, using “depth-interval buffers” (which essentially provide the functionality of a two-sided test) to do the rendering. Their approach requires three depth buffers, the two sided test and two color buffers, and thus is not readily adaptable to current OpenGL-based architectures. Although they do not analyse their algorithm in terms of rendering passes, we believe that their approach requires a number of passes for each product proportional to its depth complexity from the given viewpoint.

Stewart *et al.* [22] presented an improvement to the Goldfeather *et al.* algorithm that takes into account the fact that objects may be disjoint and thus can be rendered in parallel. Let the depth complexity of the collection of  $n$  objects being rendered be  $k$ . Then the modification proposed by Stewart *et al.* requires  $O(kn)$  rendering passes. In the case when objects do not intersect greatly (and thus  $k < n$ ), this algorithm is superior. Erhart and Tobler [4] demonstrated that in certain situations when the algorithm of Stewart *et al.* might run into problems as a result of inaccurate depth calculations, a modification yields more accurate results. However, in the worst case, their algorithm again requires  $O(n^2)$  passes.

More recently, Stewart *et al.* [23, 24] present improvements that compute a CSG product in a constant number of passes when the objects to be rendered are all convex. They do this with an ingenious technique of using a universal sequence to model the depth ordering of a set of objects without having to compute an explicit front-to-back ordering.

All of the algorithms above compute a union of objects by blending in the partial depth buffers obtained for each product. As we shall see later, our algorithm avoids this blending step.

To the best of our knowledge, there has been no work on extracting the  $k^{th}$  layer of a scene. The closest prior work on this topic seems to be the depth peeling work by Everitt *et al.*

## 3 Rendering CSG Trees

A three dimensional object can be described as the result of performing set operations ( $\cup, \cap, \setminus$ ) on a ground set of shapes. A *CSG tree* can be used to define an object by defining the sequence of operations that are performed. For example, consider the shapes A, B and C, and their combination  $A \cap B - C$  shown in Figure 1.

Figure 1: An example of a CSG tree. Internal nodes are labelled with the set operation being performed, and leaf nodes represent the ground set of shapes

### 3.1 Tree Normalization

The CSG tree is usually assumed to be in a canonical form to aid in rendering. A CSG tree is said to be in *sum-of-products* form if the expression it defines can be written as a union of intersections/subtractions (a sum of products). Such a tree is said to be *normalized*. Goldfeather *et al.* [7] provide an algorithm for normalizing a CSG tree; we use their technique, and in the rest of the paper assume without loss of generality that the CSG tree has been normalized.

If a CSG tree is normalized, then given a procedure to compute the product of a set of shapes, the union can be computed easily by merging the results of the product renderings in the depth buffer. The above mentioned algorithms all make use of this property, and thus focus on the problem of rendering a CSG tree denoted by a single product. Our algorithm, on the other hand, works on an *arbitrary* sum of products simultaneously. For clarity of presentation, we will explain the working of the algorithm on a single product, and subsequently we will show how the same idea can be used to render a sum of products. Hence, in what follows, we will consider the problem of rendering a CSG tree denoted by a single product.

### 3.2 Rendering A Product

Consider a single product  $((\mathbf{o}_1 \cap \mathbf{o}_2) - \mathbf{o}_3) \cap \mathbf{o}_4$ . By the transformation  $A - B = A \cap \bar{B}$ , where  $\bar{B}$  denotes the complement of  $B$ , we can rewrite the product as  $\mathbf{o}_1 \cap \mathbf{o}_2 \cap \bar{\mathbf{o}}_3 \cap \mathbf{o}_4$ . Thus in general, each product is the intersection of a set of (possibly complemented) objects.

The algorithm works by traversing the *arrangement* of the boundaries of the objects. It maintains a collection of sweep lines together move away from the viewing direction through the object, at each stage examining all cells of the arrangement in contact with the sweep lines.

Let the viewing direction be along the y-axis in the increasing direction from a viewpoint at  $y = -\infty$ . We call the *level* of a point in the arrangement the number of boundaries of objects crossed by a half-infinite ray parallel to the viewing direction emanating from the point towards the viewpoint (if the ray crosses multiple boundaries of a non-convex object, then each crossing is counted). For technical reasons that we explain below, the arrangement that we use for the purpose of constructing fronts only uses front faces of uncomplemented primitives and back faces of complemented primitives.

The  $k$ -level of the arrangement is defined as the set of points at level  $k$ . The *front* of a level is the set of points on the level closest to the viewpoint. We mention that for a single object, the collection of levels are precisely the *layers* that Goldfeather *et al.* and other researchers have used when dealing with non-convex shapes. Note that the maximum level of the arrangement of objects is precisely the depth complexity of the scene. The algorithm proceeds by a sweep approach. It first computes the front for level 0. It then checks to see whether any of the points in this front are in the desired product. Any point in the product is retained, and the *remaining portions* of the front are propagated to the next level.

**Testing product membership** Goldfeather *et al.* [7] made the key observation that if an object  $\mathbf{o}$  occurs uncomplemented in a product, the number of levels (or layers) of that object that lie between the viewpoint and a fixed point must be odd in order for that point to lie in the product. Similarly the number of levels must be even if the object is subtracted (occurs complemented). Let  $\text{parity}(\mathbf{o}, p)$  denote the parity of the number of levels of  $\mathbf{o}$  that lie between  $p$  and the viewpoint. For a product of the form  $\mathbf{o}_1 \cap \mathbf{o}_2 - \mathbf{o}_3$ ,  $p$  will lie in the product if  $\text{parity}(\mathbf{o}_1, p) = 1$ ,  $\text{parity}(\mathbf{o}_2, p) = 1$ ,

and  $\text{parity}(\mathbf{o}_3, p) = 0$ ). Another way of writing the third constraint is  $\text{parity}(\mathbf{o}_3, p) + 1 = 1$ ).

For a general product  $P$ , the condition that any point  $p$  lies in  $P$  can thus be written as

$$k + \sum_{\mathbf{o} \in P} \text{parity}(\mathbf{o}_i, p) = n \quad (3.1)$$

where  $k$  is the number of subtracted objects. Let  $f(\mathbf{o}, p)$  denote the number of front faces of  $\mathbf{o}$  between  $p$  and the viewpoint. Similarly let  $b(\mathbf{o}, p)$  denote the number of back faces of  $\mathbf{o}$  between  $p$  and the viewpoint. Now  $\text{parity}(\mathbf{o}, p)$  is precisely  $|f(\mathbf{o}, p) - b(\mathbf{o}, p)| \bmod 2$ . Since all objects are simple and thus have no self-intersections, each front face of  $\mathbf{o}$  is followed immediately by a back face of  $\mathbf{o}$ , and thus  $b(\mathbf{o}, p) \leq f(\mathbf{o}, p) + 1$ . This implies that we can rewrite the above equation as

$$\text{parity}(\mathbf{o}, p) = f(\mathbf{o}, p) - b(\mathbf{o}, p)$$

Summing over all all objects, the new condition that any point  $p$  lies in a product is

$$k + \sum_{\mathbf{o} \in P} f(\mathbf{o}, p) - \sum_{\mathbf{o} \in P} b(\mathbf{o}, p) = n \quad (3.2)$$

In the next section, we present a detailed implementation of this algorithm.

## 4 Algorithm Details

### 4.1 Advancing The Front

The front is maintained as depth values in the z-buffer. Initially, all the objects are rendered with the depth-test set to LESS. After this the z-buffer contains the front at level 1.

To advance the front, we copy the z-buffer to a depth texture, and invoke the *depth peeling* subroutine to filter only fragments whose depth is greater than the value in the z-buffer. We refer the reader to [5] for details of the implementation of depth peeling. This stage returns all fragments *behind* the front. In the next stage of the pipeline, the depth test is again set to LESS and thus the z-buffer now contains the front at level 2.

Observe the crucial role of the second depth test provided by the *depth peeling* routine. Without this, we would be unable to implement the two-sided depth test  $a \leq Z\text{value} < b$ , and would not be able to advance to the next level.

### 4.2 Testing for Membership

The replacement of Equation 3.1 by Equation 3.2 is crucial because it allows us to check membership for a point  $p$  in a constant number of passes (instead of  $n$ ). Recall that the current front is stored in the z-buffer. We first initialize the stencil buffer to  $k + 1$ , where  $k$  is the number of complemented objects in the product<sup>2</sup>. The depth test is set to LESS-OR-EQUAL with no z-buffer writes. In all these passes writes into the color buffer are disabled.

**Pass 1:** Render front faces of all uncomplemented objects, incrementing the stencil buffer if the depth test passes. Render similarly back faces of all uncomplemented objects, but this time decrementing the stencil buffer.

<sup>2</sup>This is to ensure that the stencil buffer will not contain 0 values as a result of counting faces. We accordingly modify the threshold for membership to be  $n + 1$ .

**Pass 2:** Render back faces of all complemented objects, incrementing the stencil buffer if the depth test passes, and render similarly their front faces, decrementing the stencil buffer in this case.

```
// N is set to 2b - 1 where b is the
// bitwidth of the stencil buffer
glDepthFunc(GL_LEQUAL);
glEnable(GL_STENCIL_TEST);

glStencilFunc(GL_NOTEQUAL, N, 0xff);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);

// Draw front faces
// Draw back faces

glStencilFunc(GL_NOTEQUAL, N, 0xff);
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);

// Draw back faces
// Draw front faces
```

### 4.3 Managing The Buffers

In between rendering phases, we need to re-initialize the stencil buffer at all pixels where a point of the product has not been found. After a rendering phases, we store a fixed value  $N = 2^b - 1$  (if the stencil buffer has  $b$  bits) in pixels where the stencil count is  $n + 1$ . The stencil test is set to  $\neq N$  and thus pixels in the color buffer for which the correct value has been found will not be modified.

#### Before Rendering Objects

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_NOTEQUAL, N, 0xff);
glStencilOp(GL_KEEP, GL_ZERO, GL_ZERO);
```

```
DrawQuad();
// now stencil buffer has either 0's or N
```

```
glStencilFunc(GL_NOTEQUAL, N, 0xff);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
for (i=0; i<val; i++)
```

```
    DrawQuad();
// now stencil buffer has either val's or N
```

#### After Rendering Objects

```
glDisable(GL_LIGHTING);
glDisable(GL_DEPTH_TEST);
```

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_EQUAL, val, 0xff);
glStencilOp(GL_KEEP, GL_ZERO, GL_ZERO);
DrawQuad(); // now val's are made to 0
```

```
glStencilFunc(GL_EQUAL, 0, 0xff);
glStencilOp(GL_KEEP, GL_INVERT, GL_INVERT);
DrawQuad(); // now 0 is made to 255
```

## 4.4 Running Time Analysis

From the current viewpoint, let the depth of the scene be  $k$ . As observed earlier, the number of levels the algorithm walks through is at most  $k$ . In order to advance a level, the algorithm must make six rendering passes. Thus, the total number of rendering passes is  $6k$ . It is important to observe that the multi-pass technique we use to maintain the correct stencil values is essential to avoid readbacks and auxiliary buffer management.

This running time is asymptotically superior to all prior techniques (by a factor of  $n$ ). Moreover, the running times of the previous algorithms is “worst-case”: the number of rendering passes required in any run is always the same. In our algorithm, we only render while there are pixels whose correct depth is yet to be determined. Therefore, while in the worst case our algorithm would take  $4k$  passes, in practice it might take far less, depending on the scene.

## 4.5 Computing A Union

As we mentioned earlier, there are two ways that our algorithm could be extended to compute a union of products. The first way, employed by previous algorithms, is to accumulate the depth values, retaining the smallest values.

The second technique allows us to compute the entire sum of products simultaneously. This technique is best explained by comparing it with the technique for one product. Let  $k(P)$  denote the number of complemented primitives in a product  $P$ .

```
Initialize z-buffer to first front.
Initialize stencil buffer with 0
while( stencil buffer contains 0 ) {
  Reset portions of
    stencil buffer to  $k$  where value  $\neq N$ 
  Test points on front for membership
    in P
  Set stencil buffer to value  $N > n$  for
    pixels passing test
  Advance front
}
```

Instead of running the above algorithm once for each union, we can interleave the computations for each union; the sentinel  $N$  ensures that pixels are not modified after they have attained the correct depth value. Let the products be  $P_1, P_2, \dots, P_m$ .

```
Initialize z-buffer to first front.
Initialize stencil buffer with 0
while( stencil buffer contains 0 ) {
  for  $i = 1$  to  $m$  {
    Copy  $k(P_i)$  into portions of
      stencil buffer  $\neq N$ 
    Test points on front for membership
      in  $P_i$ 
    Set stencil buffer to value  $N > n$ 
      for pixels passing test
  }
  Advance front
}
```

The arrangement  $\mathcal{A}(U_i P_i)$  is a strict refinement of  $\mathcal{A}(P_i)$ , and thus for any fixed product  $P_i$ , all its levels will eventually be encountered. However, consider a pixel  $p$  that takes a depth value from a front of  $P_i$  in the correct solution. It might happen that before this front is encountered, the depth of  $p$  (and thus the stencil value) is already set, preventing  $p$  from being updated correctly. However, note that each depth value encountered while advancing the fronts is either on a level of  $\mathcal{A}(P_i)$  or is an interior point of a

cell of  $\mathcal{A}(P_i)$ . In the first case, the correctness of the algorithm for a single product guarantees that the membership test will be computed correctly. In the second case, there must exist a depth value encountered at  $p$  prior to this front which (a) lies on a level of  $\mathcal{A}(P_i)$  and (b) yields the same result for a membership test. In Figure 2, the arrangements of two products  $P_1, P_2$  are drawn in red and black respectively. If the blue point (contained in a cell of  $P_2$ ) is contained in the union of  $P_1$  and  $P_2$ , then so is the green point on the boundary of this cell. This point will be tested at a prior level of the arrangement, and thus in all cases the depth values are computed correctly.

All prior CSG algorithms (with the exception of [3, 21]) do not propagate the layers in a strict front-to-back order. This property is crucial for our union algorithm to work.

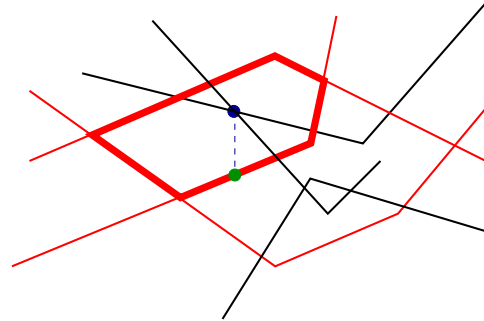


Figure 2: If an interior point is used to test for membership, there exists a boundary point with lesser depth that will yield the same result.

Figures 3 and 4 present a detailed explanation of the algorithm on the example shown in Figure 3(a). Figure 3(b) shows the arrangement of the objects. Notice that since objects C and D are subtracted, their *back* faces are included in the arrangement, and for all other objects, the *front* faces are included in the arrangement.

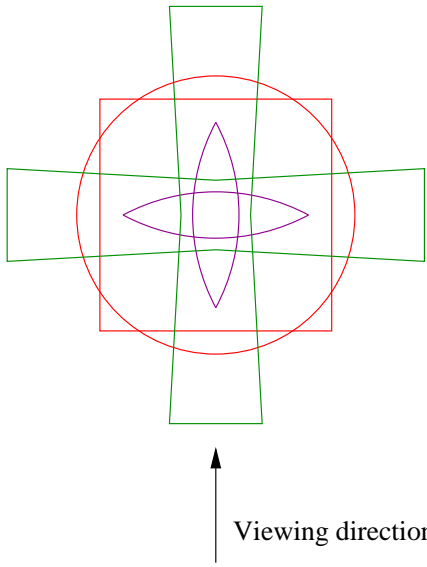
In the next four sets of figures, the current front is drawn on the left hand column, and the resulting output in the color buffer is shown in the right hand column. In each stage, the rendering passes described above are performed to determine which pixels are contained in the product, and then the front is advanced to the next stage. Finally, in Figure 4(c) we show the rendered output in the right column; Figure 4(d) displays all the regions lying in the CSG product.

## 4.6 Other Advantages Of Our Approach

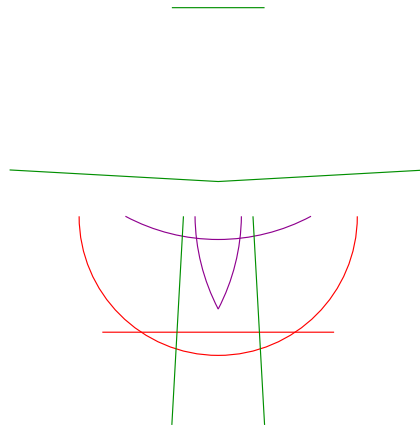
As we noted earlier, our algorithm uses a factor of  $n$  fewer rendering passes to compute a product of shapes. In addition to this, it has other desirable properties:

- Our algorithm computes layers in a strict front-to-back order, similar to Trickle. While this requires the ability to sort depths, it also allows us to compute the entire sum of products simultaneously in a seamless fashion. This allows us to avoid maintaining auxiliary depth buffers. It also allows us to exit as soon as the correct rendering has been achieved.
- Many readbacks are required to compute the levels used in the algorithms (either for each object [8], or over the entire scene [22]). In addition, the intermediate depth buffers must be stored and finally rendered to obtain the true depth values of the solution. In our solution, no such buffer transfers between the card and main memory are required, thus avoiding the use of costly low bandwidth buses.

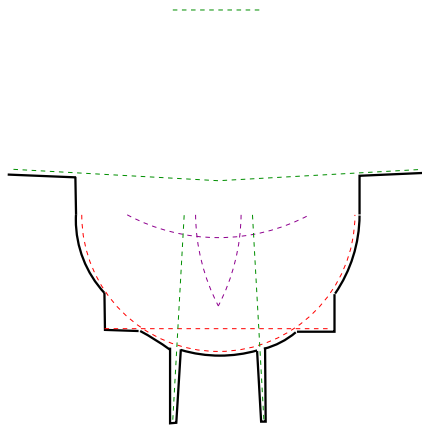
$(AB - C - D) \cup EF$



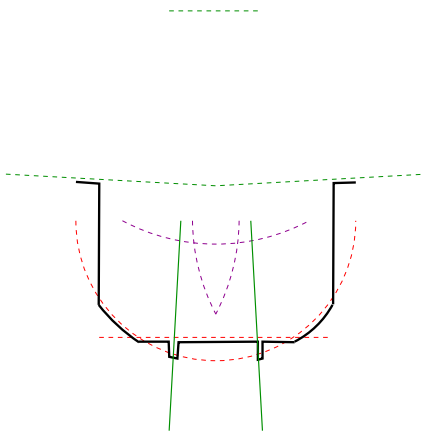
(a) The input shapes



(b) The arrangement of the input

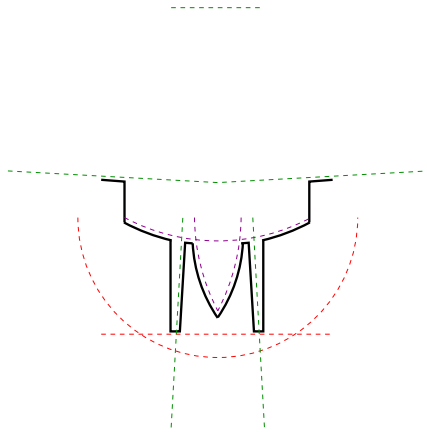


(c) Stage 1: The first front (on the left) and the retained pixels (on the right)

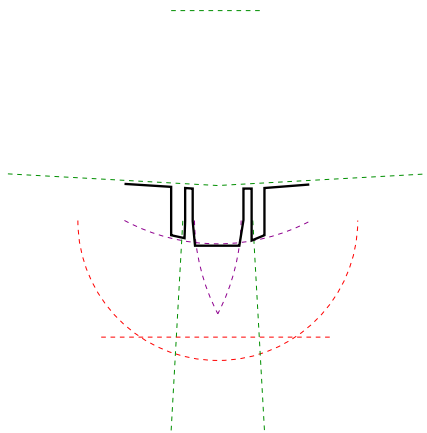
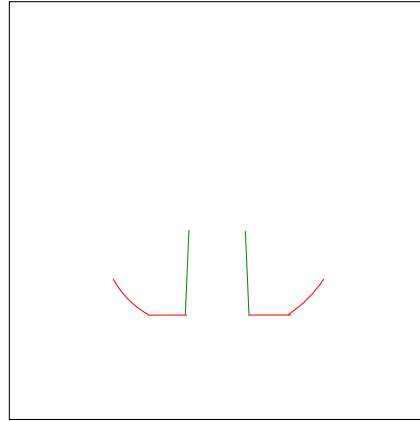


(d) Stage 2

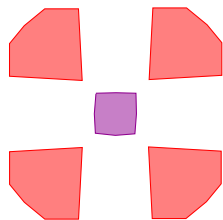
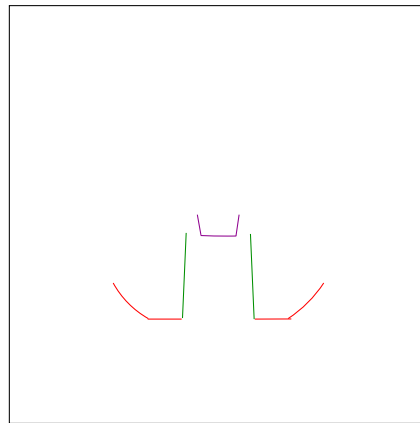
Figure 3: Stages in the Algorithm



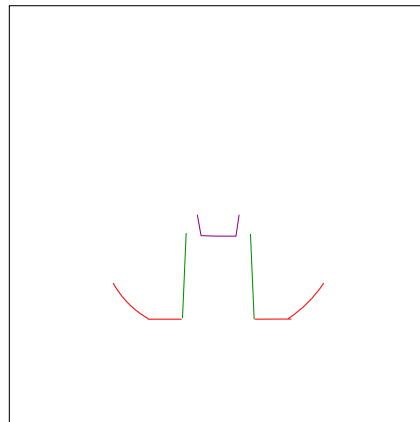
(a) Stage 3



(b) Stage 4



(c) The set of all regions in the product



(d) The final rendered output

Figure 4: Stages in the Algorithm continued...

We note here that the use of the two-sided depth test can be avoided. By applying our technique for testing product membership to the Stewart *et al.* algorithm, their level computations can be done in two passes, and then testing membership at each level can be done in a manner similar to ours, thus requiring only  $O(k)$  passes, instead of  $O(nk)$  passes. However, the above problems with buffer management still remain.

## 5 Implementation Details

All our code was implemented using C++/OpenGL on a 933Mhz/128 MB PC running Red Hat 7.3. The graphics card is an nVidia GeForce4 Ti4600. Our system performs no readbacks and uses no intermediate software buffers, while being able to handle *arbitrary* sums of products.

We will illustrate the performance of our system with the examples described in Table 1.

Name of object	Number of Products	Number of Primitives	CSG Rendering Frame Rate (fps)
CUBE	1	3	100
HELIX	1	4	50
GRID	1	26	17
DRIVEWHEEL	8	32	16

Table 1: CSG Models used in our implementation

Figure 5 demonstrates the working of our system on the above models. For each object, the lefthand-most image displays all the primitive objects involved in the CSG operations. As we go from left to right, each image displays the portion of the final answer rendered at that layer. In the case of DRIVEWHEEL, the original CSG object is very complex and so we render the set of primitive as two distinct figures (the two left-most ones) for ease of viewing. We also emphasize that we place *no* convexity restrictions on our primitives; HELIX contains nonconvex objects.

## 6 Depth Peeling and the Power of the Two-Sided Depth Test

In the previous sections, we demonstrated the power of the two-sided depth test in the context of rendering CSG trees. We now illustrate the crucial nature of this test, by considering the depth peeling application of Everitt *et al.*. The precise operation they perform is: “Given a layer of a scene, find the layer right behind it”. We will show that without the two-sided depth test, it is impossible to perform this operation in less than  $n$  rendering passes, where  $n$  is the total number of layers of objects in the scene.

**A Note on Lower Bounds** The standard method for proving a lower bound of this nature is to fix a model of computation with fixed operation costs, and demonstrate that an imagined adversary can supply inputs in such a way as to fool the algorithm into performing many operations. The model chosen typically does not reflect the entire range of operations of the computational system, but attempts to capture those operations believed to be at the heart of the problem being solved. We note that any lower bound is only as good as the computational model being used; if the model is changed, the lower bound does not in general continue to be true unless proved otherwise.

### 6.1 Computational Model

Consider a *comparison-based* model where we are allowed only to compare the values of objects. Assume all values are distinct and denoted by the set  $I$ . Define  $\text{PASS}(v_1, v_2) = \{x \in I \mid v_1 \leq x \leq v_2\}$ ,  $\text{PASS}_>(v) = \{x \in I \mid v < x\}$ , and  $\text{PASS}_<(v) = \{x \in I \mid x < v\}$ . Define  $\text{PASS}_\leq(v)$  and  $\text{PASS}_\geq(v)$  analogously. For a set  $S$ , let  $\max(\min)S$  represent  $\max_{x \in S}(\min_{x \in S})x$ . Finally, let  $\text{RANK}(v, S) = |\{x \mid x \in S \mid x \leq v\}|$ . The class of algorithms we will consider are algorithms that can ask questions of the form “What is  $|\text{PASS}_t(v)|$ ?”, where  $t$  is one of the tests  $<, >, \leq, \geq$ . The algorithm can also ask the question “Name an *arbitrary element* in  $\text{PASS}_t(v)$ ?”

Intuitively, the sets  $\text{PASS}_\leq(v)$  and  $\text{PASS}_\geq(v)$  reflect the results of performing a z-test on a fragment; the sizes of these sets can be computed using the stencil buffer. The operation  $\text{PASS}(v_1, v_2)$  reflects the result of performing a two-sided depth test. The restriction to comparison-based operations means that we disallow the arithmetic blending operations that the graphics pipeline is capable of performing; as we discussed earlier, we believe that these operations are not critical in computing the layers of scenes with depth complexity.

### 6.2 The Lower Bound Strategies

We will show that any algorithm of the above described form must ask  $\min\{\Omega(n), W\}$  questions where the size of the buffers are  $W$  bits. We will allow the algorithm to be randomized in its strategy; since any deterministic algorithm is trivially a randomized algorithm (which uses 0 random bits), randomized algorithms are at least as powerful as deterministic algorithms and so a lower bound for randomized algorithms automatically implies a lower bound for any deterministic algorithm.

**Deterministic Strategies** Most lower bounds on deterministic algorithms work in an *evasive fashion*. The adversary constructs an input which is almost completely specified except for some small pieces of information. The deterministic algorithm asks questions in a fixed order and the adversary avoids disclosing the information as long as possible. The adversary has to design the concealed information and answer questions truthfully in such a way that the result of the computation cannot be determined until the algorithm has been forced to ask a certain number of questions.

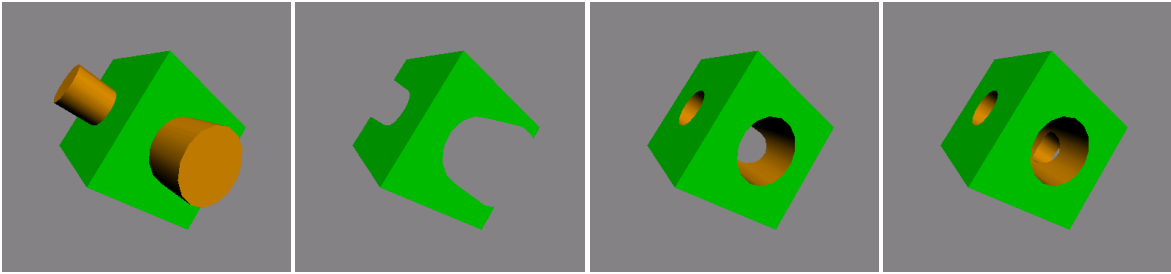
In the setting of depth peeling, the adversary constructs a set of  $n = 2k + 1$  fragments such that  $k$  fragments have depth value  $< \frac{1}{4}$  at a pixel and  $k$  fragments have depth value greater than  $\frac{3}{4}$ . One fragment has a depth value  $v_m$  which satisfies  $\frac{1}{4} < v_m < \frac{3}{4}$ . The adversary does not reveal the order in which the fragments appear. The goal of the algorithm is to find the value  $v_m$ , the result of *depth peeling* when the current depth value is set to  $\frac{1}{4}$ .

We will show that finding  $v_m$  requires  $\Omega(\min\{W, n\})$  passes. If depth peeling could be performed in a fewer number of passes, then we could compute  $v_m$  faster than the bound above, a contradiction. We will thus conclude that depth peeling requires the specified number of passes<sup>3</sup>.

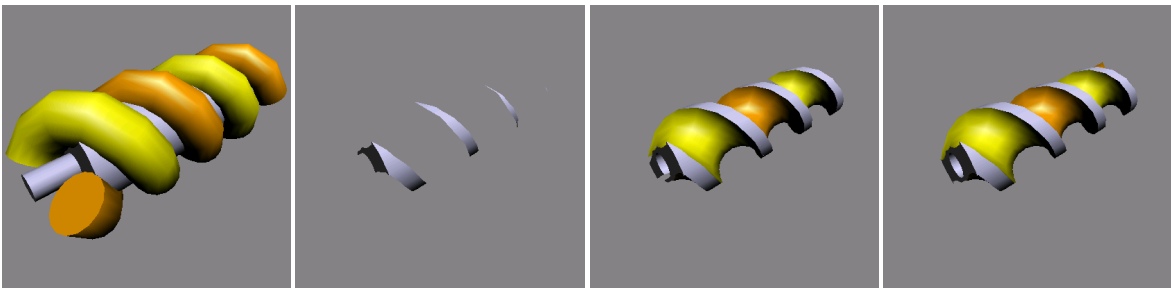
In addition to the queries described earlier, the algorithm can also ask “tell me the depth value of the  $i^{\text{th}}$  fragment in the order the fragments appear”. Such questions can be answered in a single pass using the stencil buffer.

Consider an evasive strategy that tries to maximize the length of the unresolved interval in which  $v_m$  lies. Initially the interval is  $(\frac{1}{4}, \frac{3}{4})$ . If the algorithm asks question “how many fragments have depth greater than  $d$ ” where  $d \leq \frac{1}{4}$  or  $d \geq \frac{3}{4}$  the adversary answers

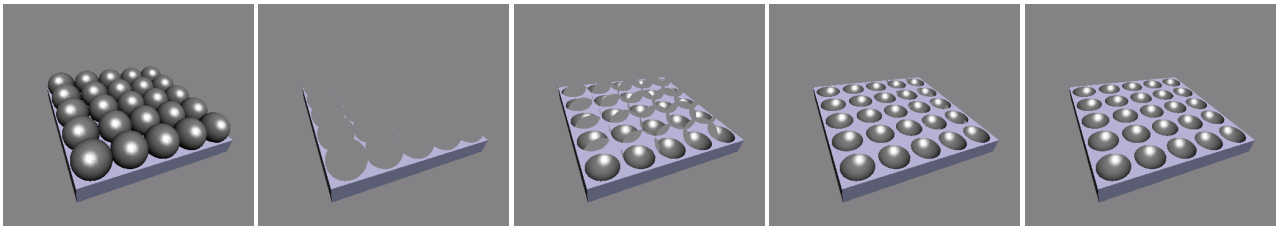
<sup>3</sup>Essentially we are using the fact that finding  $v_m$  in the special input above reduces to *depth peeling*.



(a) CUBE



(b) HELIX



(c) GRID



(d) DRIVEWHEEL

Figure 5: CSG trees, intermediate results and final output of our system

truthfully. Asking these questions does not give any information to the algorithm about  $v_m$ . If the algorithm asks such a question for a value  $\frac{1}{4} < d < \frac{3}{4}$ , the adversary has to choose an answer more carefully. For example, if the first question is “how many fragments are at depth greater than  $\frac{5}{8}$ ”, the adversary observes that the interval  $(\frac{5}{8}, \frac{3}{4})$  is smaller and thus  $v_m$  should be in the other (longer) subinterval. Thus until the algorithm asks  $W - 2$  questions of this form, the adversary has a candidate value  $v_m$  which is consistent with all its answers. After  $W - 2$  questions, all bits of  $v_m$  are specified and the algorithm knows the value.

For questions of the form “What is the depth of the  $i^{\text{th}}$  fragment in the input order”, the adversary always answers with a value other than  $v_m$ . Note that this is possible since the order of the input is not revealed to the algorithm. Eventually, the adversary will be forced to reveal  $v_m$ , but only after all other answers have been exhausted, i.e. after  $n - 1$  other queries. Putting the two facts together give us a  $\Omega(\min\{W, n\})$  lower bound on the number of queries that the algorithm must ask (and thus, the number of passes).

**Lemma 6.1.** Any deterministic algorithm must perform  $\Omega(\min(W, n))$  passes to compute the next layer in a scene, given the current one.

**Randomized Lower Bound Strategies** The method traditionally used to prove lower bounds for randomized algorithms is Yao’s MinMax Principle [18, Chapter 2]. For a deterministic algorithm, the adversary can inspect the algorithm and carefully choose inputs to fool it. This does not work for a randomized algorithm since the algorithm behaviour is not predictable on a given input. Yao’s MinMax principle says that instead of letting the adversary choose any input and having the algorithm be randomized, we can restrict the adversary to choosing inputs from a probability distribution and restrict the algorithm to being deterministic, and the lower bound applies to the more general setting. However in all cases the deterministic lower bound sets the tone for proving a randomized lower bound.

In the above setting suppose the adversary chooses  $v_m$  uniformly at random in the interval  $(\frac{1}{4}, \frac{3}{4})$  and chooses a random permutation of the fragments as the input. Now we need to investigate the expected number of steps of the algorithm. One such way is to quantify “how many bits of  $v_m$  does the algorithm know” after every question. Thus we can define the *information gained* by the algorithm in the number of bits. We defer the details of the argument to Appendix A.

**Lemma 6.2.** The expected number of rendering passes required by a randomized algorithm to compute the *next* layer of a scene from the current one is  $\Omega(\min(W, n))$ .

## 7 CSG Lower bounds with Two sided Tests

We now address the problem of computing CSG renderings. In this section, we demonstrate that asymptotically, any algorithm that renders CSG objects must perform  $\Omega(\delta)$  rendering passes, where  $\delta$  is the depth complexity of the scene.

We will use the same comparison based model as above. In this case the discussion of the lower bound gets more complicated since questions of the form “what is  $|\text{PASS}_t(v_1, v_2)|$ ”. “what is  $\min \text{PASS}_t(v)$ ” are admissible. etc.

**Theorem 7.1.** Any deterministic or randomized algorithm that executes two-sided depth tests (in addition to the regular depth test) requires  $\Omega(\delta)$  passes to compute the visible faces of a CSG object.

*Proof.* As before we will first give a deterministic lower bound. The adversary constructs a set of  $n$  objects each with 4 faces. The

front faces of object  $i$  are at depths  $i$  and  $i + n$  (for convenience, objects are numbered from 0). The back faces of object  $i$  are at  $i + n - 1$  and  $i + 2n - 1$ . We will refer to this configuration as the clean configuration (See Figure 6). Note that there is no point at which all the objects overlap, and the depth complexity of the scene is  $4n$ .

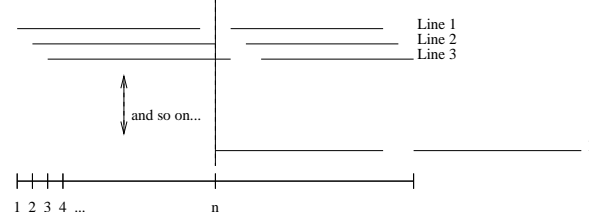


Figure 6: The clean configuration

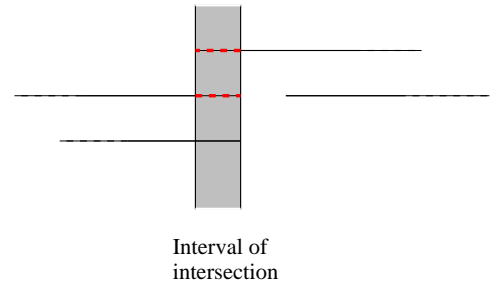


Figure 7: The modification by the adversary

Now the adversary picks an arbitrary  $i > 0$ . It then extends the interval  $[i, i + n - 1]$  to  $[i, i + n]$ , and extends the interval  $[i - 1 + n, i + 2n - 2]$  to  $[i - 2 + n, i + 2n - 2]$  (see Figure 7). Now, the configuration contains an intersection (the interval  $[i - 1, n, i + n]$ ). If we could solve CSG intersection fast, we can determine which object the adversary changed. We will show that we cannot determine the latter with less than  $\Omega(n)$  queries and thus cannot solve CSG intersection in less than  $\Omega(n)$  passes.

Since two-sided depth tests are more powerful than single-sided depth tests, we will restrict our attention to questions involving a two-sided depth test. For questions of the form “what is  $|\text{PASS}_t(v_1, v_2)|$ ” the clean and the modified configurations give the same answer as long as  $v_1, v_2 \neq i + n - 2$  or  $i + n - 1$ . Likewise computing  $\min \text{PASS}_t(v)$  gives no extra information if  $v \notin \{i + n - 3, i + n - 2, i + n - 1, i + n\}$ . We refer to these queries as posing a question for face  $v_1$  and  $v_2$  (or  $v$ ) in these cases.

Thus, as long as there are 4 consecutive planes consisting of the first back face and second front face of the objects for which the algorithm has not asked a question, the adversary can “hide” the intersection in that region. done there. There are  $2n$  faces of interest and each question asks about at most two faces. Thus the algorithm must ask  $\frac{n}{2}$  questions to ensure that there are no consecutive four faces for which no questions have been posed. Thus the lower bound is  $\Omega(n)$ .

To prove a randomized lower bound, the adversary picks  $i$  at random. Once again to discover  $i$  the algorithm must probe at least one of the 4 faces in  $\{i + n - 3, i + n - 2, i + n - 1, i + n\}$ . The probability of such an event is  $\frac{1}{r}$  where  $r$  such “blocks” of 4 faces remain. The expected running time of the algorithm is

$$1 + \frac{4}{2n} + 2 \left(1 - \frac{4}{2n}\right) \frac{4}{2n-4} + 3 \left(1 - \frac{4}{2n}\right) \left(1 - \frac{4}{2n-4}\right) \frac{4}{2n-8} + \dots$$

which evaluates to  $\Omega(n)$ . □

## 8 Depth Picking

We now examine a generalization of depth peeling that we call *depth picking*. In this problem, we wish to select the  $k^{\text{th}}$ -level of a scene, rather than just the next one, as in depth peeling. The operation of selecting a particular level in an arrangement is a key operation in a variety of geometric algorithms[2]. For example, picking the median level ( $k = n/2$ ) of a set of planes has applications to problems in clustering and data fitting[17]. Depth peeling can be used to solve this problem: to pick the  $k^{\text{th}}$ -level, we “peel off”  $k$  layers of the scene, and this can be done in  $k$  passes using the two-sided depth test. In this section, we will present a randomized algorithm that makes use of a two-sided depth test to compute the  $k^{\text{th}}$  level of an arrangement in  $O(\log n)$  passes<sup>4</sup>, an exponential improvement over using just depth peeling.

The algorithm is a randomized variant of quicksort [10]. A high-level description of this algorithm is as follows:

---

Algorithm 1: QUICKPICK

**Input:**  $x_1, x_2, \dots, x_n, k$

**Output:** The  $k^{\text{th}}$  largest item

```
lo ← arg min(x1, ... xn), hi ← arg max(x1, ... , xn).
while hi-lo > 1 do
  Pick random element xmid between xlo and xhi
  clo ← number of xi between xlo and xmid
  if clo ≥ k then
    hi ← mid
  else
    lo ← mid
Output lo
```

---

With probability greater than 0.5, the element  $x_{\text{mid}}$  will lie in the middle third of the input range, and thus after one iteration of the `while` loop, the difference  $hi-lo$  will be at most  $n/3$ , thus implying that the expected number of iterations of the `while` loop is  $O(\log n)$ . This can be shown to hold with high probability [16].

The crucial test in this algorithm is the following operation:

*Pick a random element lying in the range [lo,hi]*

Using the two-sided depth test, we can extract all elements whose values lie between two endpoints. Once these fragments pass the two tests, they are rendered into the color buffer, and the last fragment encountered is stored (assuming we do no blending). In order to ensure that this element is random, we permute the input sequence randomly in each pass, thus guaranteeing that the *last* fragment passing the two depth tests is a random fragment among all those that pass the test.

The values  $lo$  are stored in the shadow buffer and the values  $hi$  are stored in the depth buffer at each stage. After rendering into the color buffer, we transfer the contents into the stencil buffer. Since every fragment is coded uniquely, the stencil buffer now contains the ID of the random fragment chosen (at each pixel). We save the depth buffer in texture memory, and then one more pass with stencil and depth writes enabled gives us the  $mid$  values. We then compute the counts  $clo$  in the stencil buffer, and use these to compute (at each pixel) the new  $lo$  and  $hi$  values.

<sup>4</sup>The algorithm is a Las Vegas algorithm [18]; the number of passes is  $O(\log n)$  with high probability, and the algorithm always returns the correct answer.

## 9 Why Randomization is Crucial

In Section 6 we demonstrated that the two-sided depth test is crucial to perform depth peeling. Since depth picking is a strict generalization of depth peeling, the two-sided depth test is crucial for depth picking as well. As observed earlier, depth peeling can be used to solve the depth picking problem in  $k$  passes (where  $k$  is the desired level), and our randomized algorithm takes  $O(\log n)$  passes.

In this section we demonstrate that randomization and the two-sided depth are *both* necessary to do depth picking in fewer than a linear number of passes. We do this by showing that if no randomization is permitted to an algorithm, it must take  $\Omega(k)$  passes to compute the  $k$ -level.

As before, we specify the computational model first. The algorithm is permitted to ask (see Section 6 for the definitions) questions of the form “What is  $\min \text{PASS}_>(v)$ ?”, “What is  $\max x \text{PASS}_<(v)$ ?” (which are essentially depth peeling questions), and “What is  $|\text{PASS}(v_1, v_2)|$ ?”, which is an interval-count operation. We can think of the quantity  $|\text{PASS}(v_1, v_2)|$  as a *generalized rank* of  $v_2$  (count elements below  $v_2$ , but above  $v_1$ ). Note that each such question can be answered in one rendering pass.

Once again, to prove a lower bound we will follow an evasive strategy. The strategy is fairly complex and we defer the details to Appendix B.

**Theorem 9.1.** Any deterministic algorithm that computes the  $k^{\text{th}}$  level in a scene using the two-sided test must perform  $\Omega(\min(n/2 - 1, W/2))$  rendering passes.

## 10 Conclusions

In this paper, we demonstrate that the two-sided depth test, as realized by using the shadow buffer, is a powerful operator in the graphics pipeline. We show this by demonstrating an implementation that renders CSG objects in a small number of passes, and show how to extract arbitrary levels from a scene in sublinear number of passes. Our algorithms are strengthened by the lower bounds that we prove, in many ways showing that the two-sided depth test is crucial to our ability to perform such operations efficiently.

It is likely that there are many other problems for which a two-sided depth test provides a tremendous advantage over single-sided tests. In general, with the increasing power of graphics hardware, theoretical studies that attempt to ascertain the potential and the limits of this pipeline as a general purpose stream engine will tie into the burgeoning trend of streaming algorithms in other areas (databases and networking being two notable ones [1]).

## 11 Acknowledgements

Suppressed for double-blind review process.

## References

- [1] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND J. W. Models and issues in data stream systems. In *Proc. 21st ACM Symposium on Principles of Database Systems* (2002).
- [2] DOBKIN, D. P., DRYSDALE, R. L., AND GUIBAS, L. J. Finding smallest polygons. In *Computational Geometry*, F. P. Preparata, Ed., vol. 1 of *Advances in Computing Research*. JAI Press, 1983, pp. 181–214.
- [3] EPSTEIN, D., JANSEN, F., AND ROSSIGNAC, J. Z-buffer rendering from CSG: The trickle algorithm. Research Report RC 15182, IBM, 1989.

- [4] ERHART, G., AND TOBLER, R. General purpose z-buffer CSG rendering with consumer level hardware. Tech. Rep. VRVis 003, VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH, 2000.
- [5] EVERITT, C. Interactive order-independent transparency. Tech. rep., Nvidia Corporation, 2002. [http://developer.nvidia.com/view.asp?IO=Interactive\\_Order\\_Transparency](http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency).
- [6] EVERITT, C., REGE, A., AND CEBENOYAN, C. Hardware shadow mapping. In *ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations using graphics hardware* (2002), pp. F38–F51.
- [7] GOLDFEATHER, J., HULTQUIST, J. P. M., AND FUCHS, H. Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), ACM Press, pp. 107–116.
- [8] GOLDFEATHER, J., MOLNAR, S., TURK, G., AND FUCHS, H. Near realtime CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications* 9, 3 (May 1989), 20–28.
- [9] GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-buffer visibility. *Computer Graphics* 27, Annual Conference Series (1993), 231–238.
- [10] HOARE, C. A. R. Partition (algorithm 63), quicksort (algorithm 64), and find (algorithm 65). *J. ACM* 7 (1961), 321–322.
- [11] HOFF, K., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proc. IEEE International Conf. on Robotics and Automation* (2000).
- [12] HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics* 33, Annual Conference Series (1999), 277–286.
- [13] LARSEN, E. S., AND MCALLISTER, D. Fast matrix multiplies using graphics hardware. In *Supercomputing* (2001).
- [14] LINDHOLM, E., KILGARD, M., AND MORETON, H. A user-programmable vertex engine. In *Proc. ACM SIGGRAPH* (2001).
- [15] MAMMEN, A. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications* 9, 4 (July 1989), 43–55.
- [16] MCDIARMID, C., AND HAYWARD, R. Large deviations for quicksort. *J. Algorithms* 21 (1996), 476–507.
- [17] MILLER, K., RAMASWAMI, S., ROUSSEEUW, P. J., SEL-LARES, T., SOUVAINÉ, D., STREINU, I., AND STRUYF, A. Fast implementation of depth contours using topological sweep. In *Proc. 12th Annual SIAM Symp. Discrete Algorithms* (2001), pp. 690–699.
- [18] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, 1995.
- [19] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proc. ACM SIGGRAPH* (2000), K. Akeley, Ed., ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 425–432.
- [20] PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HAN-RAHAN, P. A real-time procedural shading system for programmable graphics hardware. In *Proc. ACM SIGGRAPH* (2001).
- [21] ROSSIGNAC, J., AND WU, J. Correct shading of regularized CSG solids using a depth-interval buffer. In *Advanced Computer Graphics Hardware V: Rendering, Ray Tracing and Visualization Systems*, R. L. Grimsdale and A. Kaufman, Eds., Eurographics Seminars. Springer-Verlag, 1992, pp. 117–138.
- [22] STEWART, N., LEACH, G., AND JOHN, S. An improved z-buffer CSG rendering algorithm. In *Proc. Eurographics/Siggraph Workshop on Graphics Hardware* (1998), pp. 25–30.
- [23] STEWART, N., LEACH, G., AND JOHN, S. A Z-buffer CSG rendering algorithm for convex objects. In *8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media - WSCG 2000* (2000).
- [24] STEWART, N., LEACH, G., AND JOHN, S. Linear-time CSG rendering of intersected convex objects. In *10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2002* (2002), vol. II, pp. 437–444.
- [25] WIEGAND, T. F. Interactive rendering of CSG models. *Computer Graphics Forum* 15, 4 (1996), 249–261.

## A Randomized Lower Bound for Depth Peeling

As in the deterministic setting, asking the query  $\text{PASS}_{\geq}(d)$  does not give any information for  $d \leq \frac{1}{4}$  or  $d \geq \frac{3}{4}$ .

For a query of the form “Return the  $i^{\text{th}}$  element in the input stream”, the probability that the  $i^{\text{th}}$  fragment is the fragment with depth  $v_m$  is  $\frac{1}{n}$ . Likewise, if the algorithm asks “give an arbitrary fragment in  $\text{pass}_t(v)$ ” the probability of getting the fragment with depth value  $v_m$  is  $\frac{2}{n}$ . If the fragment queried is not the desired fragment the algorithm gains no information.

Consider any algorithm that takes less than  $\frac{n}{4}$  steps. Even if all questions probe distinct fragments as above, with probability  $(1 - \frac{2}{n})^{\frac{n}{4}}$  the algorithm does not stumble upon the correct fragment. The probability is greater than  $\frac{1}{2}$  for  $n > 4$ .

Therefore at least half of the time the algorithm must utilize questions like “What is  $|\text{pass}_t(v)|$ ?”. If  $v < v_m$  the count will be  $k$  else if  $v > v_m$  then it will be  $k + 1$ . The case  $v = v_m$  means that we stumbled upon the correct value of  $v_m$  already. Yao’s principle is used here to observe that we can assume the algorithm in question to be deterministic. As a consequence, contingent upon the fact that all queries hoping to probe the correct fragment were futile, the rest of the algorithm is deterministic. Thus the algorithm defines a *fixed* binary tree on the  $2^{W-1}$  possible values of  $v_m$ . The binary tree is fixed since the first question “is  $v < v_m$ ” asked by the algorithm is fixed, being deterministic. Contingent on the answer to the first question the second such question is also fixed and so on.

Now the expected number of queries is the average depth of the tree, since we know all the bits of  $v_m$  only when we reach a leaf. But any binary tree on  $2^{W-1}$  objects has an average depth at least  $W - 1$ . Thus with probability greater than  $\frac{1}{2}$  we must ask at least  $\Omega(W)$  questions if we asked no more than  $\frac{n}{4}$  questions. This proves the desired lower bound of  $\Omega(\min\{n, W\})$ .

## B Deterministic Lower Bound for Depth Picking

We will use sets  $S_{lo}$  and  $S_{hi}$  and a value UNDEF. Initially  $S_{lo} = \{0\}$ ,  $S_{hi} = \{1\}$  and  $\text{UNDEF} = n - 2$ . The construction is as follows:

**case I:** If we are asked  $\min \text{PASS}_{>}(v)$

- If  $v < \max S_{lo}$  return the smallest element in  $S_{lo}$  larger than  $v$ .
- If  $v \geq \min S_{hi}$  return the smallest element in  $S_{hi}$  larger than  $v$ . If this element is undefined then it means that all values are less than  $v$ ; return a null symbol.
- If  $v < \frac{\max S_{lo} + \min S_{hi}}{2}$ ,  $S_{lo} = S_{lo} \cup \{\frac{\max S_{lo} + \min S_{hi}}{2}\}$  and  $\text{UNDEF} - -$ . return  $\frac{\max S_{lo} + \min S_{hi}}{2}$ .
- If  $v \geq \frac{\max S_{lo} + \min S_{hi}}{2}$  return  $\min S_{hi}$  and  $S_{hi} = S_{hi} \cup \{\frac{\max S_{lo} + \min S_{hi}}{2}\}$  and  $\text{UNDEF} - -$ .

**case II:** If we are asked  $\max \text{PASS}_{<}(v)$

- If  $v > \min S_{hi}$  return the largest element in  $S_{hi}$  smaller than  $v$ .
- If  $v \leq \max S_{lo}$  return the largest element in  $S_{lo}$  smaller than  $v$ . If this element is undefined then it means that all values are greater than  $v$ ; return a null symbol.
- If  $v \leq \frac{\max S_{lo} + \min S_{hi}}{2}$  return  $\max S_{lo}$  and  $S_{hi} = S_{hi} \cup \{\frac{\max S_{lo} + \min S_{hi}}{2}\}$  and  $\text{UNDEF} - -$ .

- If  $v > \frac{\max S_{lo} + \min S_{hi}}{2}$ ,  $S_{lo} = S_{lo} \cup \{\frac{\max S_{lo} + \min S_{hi}}{2}\}$  and  $\text{UNDEF} - -$ . return  $\frac{\max S_{lo} + \min S_{hi}}{2}$ .

**case III:** If we are asked  $|\text{PASS}(v_1, v_2)|$ .

- If  $v_1 \leq \max S_{lo}$  and  $v_2 \geq \min S_{hi}$  we return UNDEF plus the number of elements in  $S_{hi}$  and  $S_{lo}$  that fall in the range.
- If  $v_1 > \min S_{hi}$  or  $v_2 < \max S_{lo}$  we inspect the items of  $S_{lo}, S_{hi}$  which fall in the range and answer appropriately.
- If  $v_1 \geq \frac{3 \max S_{lo} + \min S_{hi}}{4}$  set  $S_{hi} = S_{hi} \cup \{\frac{3 \max S_{lo} + \min S_{hi}}{4}, \frac{\max S_{lo} + \min S_{hi}}{2}\}$ ,  $\text{UNDEF} - = 2$  and the answer inspecting the number of items in  $S_{hi}$  in the range.
- If  $v_2 \leq \frac{\max S_{lo} + 3 \min S_{hi}}{4}$  set  $S_{lo} = S_{lo} \cup \{\frac{\max S_{lo} + \min S_{hi}}{2}, \frac{\max S_{lo} + 3 \min S_{hi}}{4}\}$ ,  $\text{UNDEF} - = 2$  and the answer inspecting the number of items in  $S_{lo}$  in the range.
- Otherwise set (storing  $\max S_{lo}, \min S_{hi}$ )  $S_{lo} = S_{lo} \cup \{\frac{3 \max S_{lo} + \min S_{hi}}{4}\}$  and  $S_{hi} = S_{hi} \cup \{\frac{\max S_{lo} + 3 \min S_{hi}}{4}\}$  and set  $\text{UNDEF} - = 2$  and return (new value of) UNDEF.

We can verify that  $\max S_{lo} < \min S_{hi}$  as long as  $\text{UNDEF} > 0$  and the number of bits of precision required to express the numbers is  $n - \text{UNDEF}$ . Thus we can force the algorithm  $A$  to run for at least  $\frac{n}{2} - 1$  steps (since before these many steps UNDEF is nonzero). If the word size is  $W$  then we can force the algorithm to ask  $\frac{W}{2}$  questions since two bits of precision are fixed every iteration.