Hardware-assisted Rendering of CSG Models

Fabiano Romeiro
Harvard University
romeiro@fas.harvard.edu

Luiz Velho
IMPA
lvelho@impa.br

Luiz Henrique de Figueiredo
IMPA
lhf@impa.br

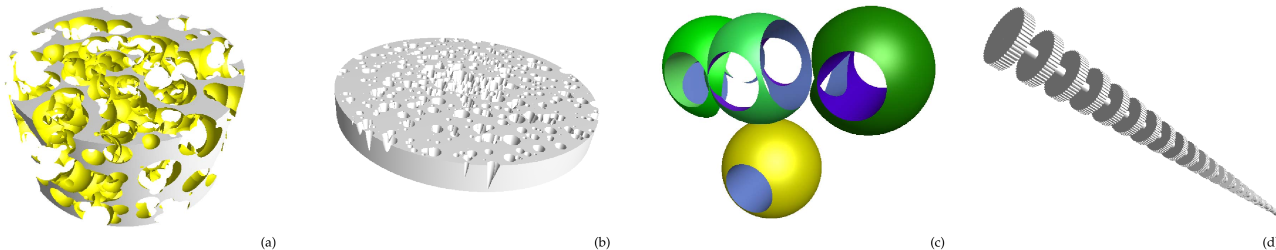


Figure 1: (a) CS1000, a cylinder subtracted of 1000 spheres (b) CC1000, a cylinder subtracted of 1000 cones, (c) R9, 9 primitives, (d) T1022, 1022 primitives

Abstract

Current methods that interactively render reasonably complex CSG objects are image based and are severely bandwidth limited. This paper presents a new approach to ray-tracing CSG objects composed of convex primitives that combines spatial subdivision and ray-tracing methods. By performing spatial subdivision on the CSG object until locally it is simple enough to be rendered effectively and efficiently on a GPU, we are able to share the load more evenly between the CPU and the GPU and depend less on bandwidth and more on GPU instruction throughput than current methods, hence obtaining better scalability with newer hardware.

Keywords: CSG, Graphics Hardware, GPU, ray tracing

1 Introduction

One of the most intuitive ways to model solid objects is by constructing them hierarchically, through combinations of simpler objects, creating more and more complex ones. Several representations that incorporate this paradigm exist, and CSG [11] is the most popular.

In the CSG representation, solid objects are obtained by successive boolean combinations of primitives, and are represented by the (CSG) expression corresponding to the sequence of boolean operations of primitives that led to them. These CSG expressions are stored as trees called CSG trees, whose leaves represent primitives and nodes represent boolean operations.

With each node is also associated a transformation to allow translation, rotation and scaling of each part of the solid object while it is being modeled. Alternatively these transformations can be applied directly to each leaf node of the subtree rooted at the node the transformation is to be associated with (in this case, no storing of transformations is needed).

1.1 Prior work and Interactivity

The CSG paradigm is highly advantageous and well suited for modeling, but much more useful if interactivity can be achieved as then models can be modified in realtime, greatly facilitating the design process. Scalability is a serious concern since rendering even reasonably complex CSG objects at interactive rates is difficult. Since the introduction of CSG, several approaches have been devised towards that goal (interactivity on ever more complex models). Some methods involve converting the CSG representation into a boundary representation and rendering that boundary, but these are not really suitable for interactive performance. Other methods are image-based, and some use special purpose hardware to render CSG objects. Goldfeather [4] presented an algorithm for rendering CSG models with convex objects (and later with non convex objects [5]) using a depth-layering approach on the Pixel Planes.

Wiegand [17] proposed an implementation of Goldfeather's algorithm on standard graphics hardware. Rappoport [10] converts the CSG representation to a CDA representation and then uses the stencil buffer to render at interactive rates. Stewart et al. [13] improved upon Goldfeather's algorithm and then introduced the SCS algorithm [14], an improvement upon [13] and later refined it [15]. Erhart and Tobbler [3], Guha et al. [6] and Kirsch and Dollner [9] implemented optimizations over either the SCS, the Goldfeather or the layered Goldfeather algorithms to better use newer graphics hardware. Adams and Dutre [1] presented an algorithm that perform interactive boolean operations on free-form solids bounded by surfels. More recently, Hable and Rossignac [7] used an approach that combines depth-peeling with the Blist formulation [12].

So far, the subset of these algorithms that have reached interactivity on decently sized models are image-based and use either depth layering or depth peeling approaches. For this reason they are bandwidth limited, and bandwidth of standard graphics hardware has historically improved at a rate that is at least an order of magnitude lower than the instruction throughput increase rate. They also impose limitations on the number of primitives (due to the number of planes available in the stencil buffer), unless they use multiple passes.

1.2 Proposed approach

We propose a method whose goal is to be more instruction throughput limited than bandwidth limited, and that has no maximum primitive number limit (being limited only by available memory). Our method also attempts to share the load between the CPU and the GPU, by performing spatial subdivision of the CSG object on the CPU and local ray-tracing of the CSG object on the GPU.

2 Spatial Subdivision

The main insight of our approach is that surfaces of CSG objects can be (mostly) locally represented by single primitives (or their complements) or by boolean operations of two primitives (or their complements). The exceptions are points of the CSG object in the intersection of surfaces of three different primitives, e.g., vertices of order 3 or bigger (see figure 2). Since ray-tracing primitives and boolean operations of two primitives (for three or more the pixel shaders would be much more complex) can be done efficiently on the GPU (as will be shown in section 3), rendering the entire CSG object reduces to:

1. Subdividing it in the CPU until all parts are either (i) composed of a single primitive or a boolean operation of two primitives, or (ii) project to less than a given threshold of pixels on the screen (and hence either contain one of the exception points or is insignificant enough not to be subdivided further and ignored).

2. Ray-tracing each part that falls on case (i) on the GPU.

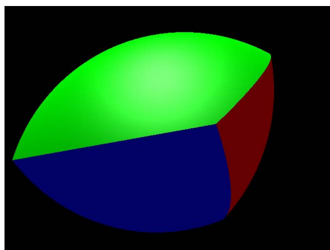


Figure 2: **Intersection of 3 spheres: The protruding point lies at the intersection of the surfaces of the 3 spheres, and hence there exists no open set containing it such that the representation of the CSG object, when restricted to that open set, reduces to a single primitive or boolean operations of just two primitives.Thus no matter how much we subdivide this object there will always exist one cell, namely the one containing this point, which does not have a simple local representation for the CSG object**

The above process produces correct results, except for possibly the pixels corresponding to parts which fell in case (ii). These, however, correspond to minimal artifacts in most cases, and can be minimised or enhanced by changing a threshold. Obviously, the smaller the threshold the more complex the subdivision will be. The tradeoff is quality over subdivision complexity and rendering time. For the initial design of a very complex CSG model, when small artifacts are of no concern, it might be desirable to use large thresholds in order to increase interactivity (see figure 3).

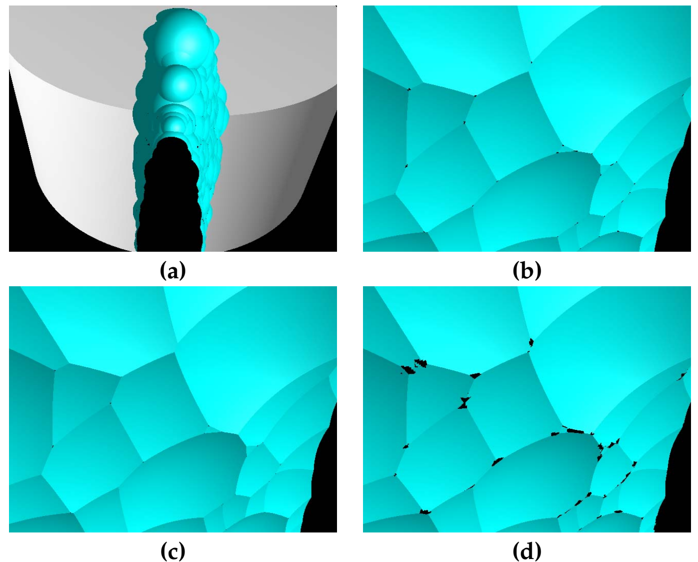


Figure 3: **(a) Using a threshold of 1 pixel usually produces no noticeable artifacts. (b) However in some scenes artifacts can be quite noticeable with that threshold, as evidenced by the zoom in on the walls. (c) Using smaller thresholds improves image quality by reducing artifacts and (d) increasing the threshold produces the opposite effect.**

2.1 Spatial subdivision structures

A modified octree structure is used to perform the subdivision of the CSG object. On each level we divide the cells in two instead of in eight, as in traditional octrees. This division is carried sequentially on the x, y and z coordinates, in a cyclic manner, making it a sequenced Kd-tree. This binary subdivision greatly improves the CSG tree simplification procedure (see section 2.2.1).

Along with each cell of the octree we keep a CSG tree structure that holds the simplest local representation of the surface of the original CSG object. All these structures are stored in main memory and the subdivision is performed on the CPU.

2.2 Octree Subdivision

The subdivision starts with the (axis aligned) bounding box of the CSG object as the initial cell of the octree. We set this cell's CSG tree to a copy of the CSG tree representing the CSG object. The process proceeds by subdividing the cell, setting each of its children's CSG trees to a copy of its own CSG tree, and then simplifying each child cell's CSG tree to obtain the simplest CSG tree that still represents the surface of the original CSG object when restricted to each child cell.This is repeated for each of the children recursively until one of the following conditions are met:

1. The cell's CSG tree is empty (i.e., the surface of the CSG object does not intersect with the cell).

2. The cell's CSG tree is exclusively composed of unions of primitives. (Rendering union of an arbitrary number of primitives on the GPU can be done efficiently as in section 3.3.3, hence we do not further subdivide and render

right away these cases)

3. The cell's CSG tree has depth two (and is not an union of two primitives, since that falls on case 2).

4. The cell projects onto less than a threshold of pixels in the screen.

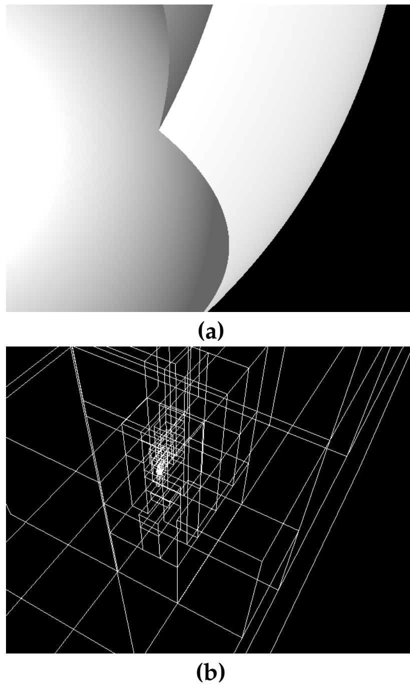


Figure 4: **(b) Note the adaptiveness of the octree and how it concentrates at the point of the (a) CSG object where the surfaces of three different primitives intersect: The octree keeps being subdivided at that region until the cell that still intersects the surfaces of three different primitives project to less than a threshold of pixels, 1 in this case, on the screen**

Cells that fall in either of these cases will be called leaf cells and all the others are called node cells.

Note that on each level of subdivision we do not start with the original CSG tree, but with the CSG tree of the parent cell, which is already a simplification of the original CSG tree (that restricted to the parent cell, and hence to the child cell, correctly represents the surface of the CSG object), thus the improvement of doing binary subdivision.

2.2.1 CSG tree simplification scheme

As has been said before, the goal of our CSG tree simplification scheme is to, given a cell and a CSG tree, simplify this CSG tree as much as possible, ending with the simplest CSG tree that, restricted to that cell, still represents the surface of the original CSG object correctly. This is done in a standard fashion, by pruning sub-trees of the CSG tree whose surface do not intersect with the current cell in a recursive bottom-up approach.

2.3 Traversal and rendering

Once the octree has been generated, it is traversed recursively in a view-dependent front-to-back manner. When a leaf cell corresponding to cases 1 or 4 in 2.2 is reached it is ignored, and when a leaf cell corresponding to cases 2 or 3 is reached, the part of the surface of the CSG object in the interior of that cell, namely the restriction of the cell's CSG tree to its interior, is rendered in the GPU, as detailed in the next section.

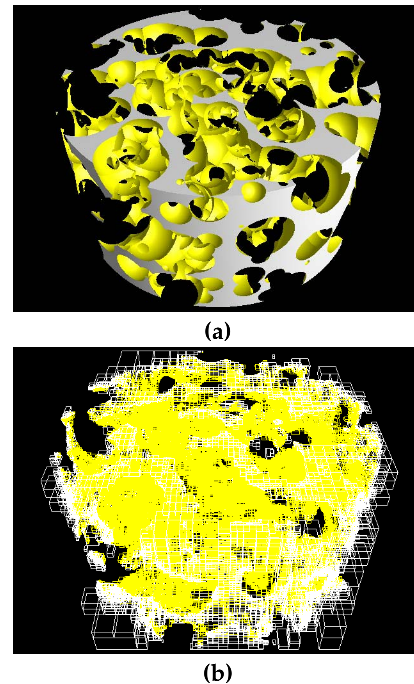


Figure 5: **(a) CS1000 and (b) its final octree: Each cell drawn in (b) has in its interior a part of the surface of the CSG object that can be represented by either a single primitive (or its complement) or by a boolean operation of just two primitives (complemented or not).**

It is also possible to render the CSG object while subdividing it. By doing so, paralelization of the subdivision on the CPU and the rendering on the GPU is achieved (see section 4). For vizualiation purposes however, once the subdivision has been completed for a CSG object we can view it from other angles and under different lighting conditions without re-subdividing from scratch.

The next section lays out the fundamentals of ray-tracing single, union, intersection and difference of primitives (or their complements) restricted to a given cell in the GPU.

3 Ray-tracing CSG objects in the GPU

The newer generations of GPUs have allowed the design of algorithms that transfer a substantial part of the workload to the GPU. Several developments have been made toward transfering ray-tracing of objects from the CPU to the GPU. We extend on the work developed by Toledo and Levy [16], to ray-trace not only a set of convex primitives, but also boolean operations of them.

Note that we render only the parts of primitives or boolean operations of primitives that are inside a given cell, as the restriction of the CSG object to each cell is rendered one at a time, and the local representation of the CSG object may not correspond to the original object outside the given cell. That being said, when ray tracing we must clip to the corresponding cell. Section 3.1 is an introductory presentation to the important concepts of ray-tracing GPU primitives, and for the sake of clarity does not incorporate this clipping. Sections 3.2 and 3.3 constitute the core of this section and detail at length how we ray-trace single and boolean operations of primitives, respectively.

3.1 Concepts of ray-tracing GPU primitives

The basic idea of ray-tracing primitives in the GPU, as introduced by Toledo and Levy [16], is to bind the appropriate vertex and pixel shaders first, and render some object (e.g., a bounding box) whose projection on the screen (the ray-tracing area, or RTA) covers the projection of the intended primitive (see figure 6). The shader then runs for every point of the faces of the rendered object, and traces a ray from that point, in the direction of the camera, determining if that ray intersects the primitive or not (see section 3.1.2).

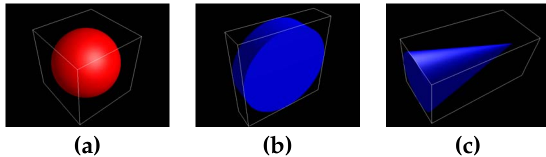


Figure 6: **The front faces of the bounding boxes of the (a) sphere (b) cylinder (c) cone are rendered. The pixel shader then runs on each pixel of those front faces ray-tracing the respective primitives**

3.1.1 Bounding box as the ray-tracing area

Front faces of Axis Aligned Bounding Boxes (AABB), i.e., the cells of the octree structure defined in the previous section, are used as RTAs. This might not be the most efficient, depending on the primitive and the camera position. For example, a better suited RTA for a cone would be a pyramid in which the cone is inscribed, as the pixel shader would run on a smaller number of pixels than with a bounding box as the RTA, and still be able to ray-trace the cone properly. [16] provides a more thorough analysis of RTAs and their efficiency.

3.1.2 Vertex and Pixel shader's roles

After rendering the front faces of the bounding box of a primitive with the appropriate vertex and pixel shaders bound, the vertex shader will be executed for every vertex of this bounding box, passing to the pixel shader vectors containing light, pixel and camera positions, all in object space. The pixel shader receives this information, as well as parameters containing information on the primitives in question (depending on primitive: center, radius, etc), calculates the ray direction, traces the ray starting from the point at the bounding box corresponding to the pixel in question, and calculates the closest (to the camera) intersection with the primitive, as well as the primitive normal at that intersection. It then computes the color and the correct depth of the pixel, as detailed in section 3.2.

3.2 Ray-tracing of simple convex primitives

The previous fundamentals will work for any class of primitives. From here on, however, we will focus on the subset of convex primitives, because with convex primitives we are assured that any given ray-primitive intersection result will either be empty or consist of a single segment (which can be stored with only two scalar variables), thus allowing for an efficient implementation of the ray-primitive and other necessary algorithms on the pixel shader.

Given a cell L and a primitive P, in order to ray-trace P restricted to L, we use the framework described in 3.1 with a pixel shader that performs:

1. Let $r = o + t \cdot v$ denote the ray starting at o (the object space coordinates of the point for which the shader is running), and going in the direction of the camera (v).
2. Intersect L with r - Obtain a segment [1], $S_L = [0, t_f]$ as a result (i.e., the set of all t such that r and L intersect).
3. Intersect P with r - Obtain S_P, a segment (degenerate or not) or an empty set. Calculate surface normals at the ray-surface intersection points (if any).
4. Complement S_P and the respective ray-surface intersection normals if P is complemented.
5. Clip S_P to L (i.e., intersect S_P and S_L), obtaining S_R.
6. If S_R is empty, this pixel of the RTA does not correspond to a pixel of the primitive, and is thus discarded.
7. Let t be the smallest value in S_R.
8. If $t = 0$ and $(-\varepsilon, \varepsilon)$ is in S_R for some $\varepsilon > 0$, discard the pixel.[2]
9. Else, the ray intersects the surface of P inside L at $o + t \cdot v$, and we then:
   - Calculate correct depth value for this pixel, which originaly contains the depth of o, which lies in one of the faces of the cell.(We must update this pixel's depth value to the depth of $o + t \cdot v$).
   - Calculate color of the pixel, given t's associated color an surface normal.

Follows a description of each part of the algorithm.

Ray-cell intersection: A cell is nothing more than an AABB. Intersecting rays with AABBs is relatively easy and is done as described in [8].

Ray-primitive intersection: The ray-primitive intersection set is calculated by replacing the ray equation ($r(t) = o + t \cdot v$) into the equation for the surface of the primitive in question

[1] The ray always intersect the cell, because the ray always starts at a point on one of the front faces of the cell

[2] The object then extends outside the cell towards the camera, since the cell is in the range $[0, t_f]$. This means the object must be clipped at this point by the cell, so the pixel is discarded since it will certainly be shaded when a cell closer to the camera, that contains the surface that was clipped, is rendered

and solving for t, finding the intersection points, if any. The normals at the intersection points are also calculated. Primitives implemented were the cylinder, cone, and sphere. Other implicit surfaces defined by low degree polynomials can be easily included in our system.

Depth Calculation: Depth calculation is performed by transforming $o+t \cdot v$ to eye-space coordinates, dividing by the homogeneous coordinate and rescaling from [-1,1] into [0,1].

Shading calculation: We implemented positional lighting, with ambient, diffuse and specular components.

3.3 Ray-tracing boolean operations of simple convex primitives

Section 3.3.1 describes how ray-tracing is implemented, when the boolean operation is an intersection, in a way that the resulting pixel shader is efficient. Other cases will either derive from this (difference, at 3.3.2) or will be handled differently (union, at 3.3.3).

3.3.1 Intersection of primitives

Given a cell (L) and two primitives (P_1 and P_2), we want to ray trace ($P_1 \cap P_2$) restricted to L. Again we perform the same procedure described in 3.1.2, with a pixel shader that performs as follows:

1. Intersect L with r - Obtain a segment ($S_L = [0, t_f]$) as a result.
2. Intersect P_1 with r - Obtain S_{P1}, either a segment (degenerate or not) or an empty set as a result. While calculating intersections, calculate surface normals at the ray-surface intersection points (if any).
3. Intersect P_2 with r - Obtain S_{P2}, as in the previous step.
4. Complement S_{P1} and the respective normals if P_1 is complemented and likewise for S_{P2}.
5. Intersect S_{P1} with S_{P2} - Obtain S (more on this in section 3.3.4). Associate with each boundary of S the appropriate normals and colors (either from S_{P1} and S_{P2}).
6. Clip S to L - Obtain S_R
7. Perform steps 6 to 9 in section 3.2.

3.3.2 Difference of primitives

The algorithm is exactly the same as the one in section 3.3.1, but we modify step 4 to instead complement S_{P1} and the respective normals if P_1 is complemented and complement SP_2 and the respective normals if P_2 is NOT complemented.

3.3.3 Union of primitives

The union operation is a special case, as it can be handled by the z-buffer of the GPU. That is, given a cell (L) and a collection of primitives (P_1, ..., P_k), if we want to ray trace ($\cup P_n$) restricted to L, we need only render P_1, ..., P_k restricted to C, one at a time, as explained in section 3.2.

3.3.4 Intersection of ray-primitive result sets

This section describes how to intersect S_{P1} and S_{P2} in a way that allows for an efficient shader implementation.

Depending on whether P_1 and P_2 are complemented, we have 4 possible combinations for S_{P1} and S_{P2}.

1. P_1 not complemented, P_2 not complemented:

   S_{P1}: 1 segment or empty-set
   S_{P2}: 1 segment or empty-set

2. P_1 not complemented, P_2 complemented

   S_{P1}: 1 segment or empty-set
   S_{P2}: 2 semi-intervals

3. P_1 complemented, P_2 not complemented (identical to previous case with P_1 and P_2 interchanged)
4. P_1 complemented, P_2 complemented:

   S_{P1}: 2 semi-intervals
   S_{P2}: 2 semi-intervals

Since calculating the intersection of 2 segments is simpler than calculating the intersection of 2 semi-intervals and 1 segment, which in turn is simpler than calculating the intersection of 2 pairs of semi-intervals, we break those 4 cases in different algorithms so as to create the simplest possible pixel shader for each case, (e.g., to render the intersection of a complemented primitive and another primitive we would load a shader that uses an algorithm for case 2).

Case 1: Segment(S_{P1}) $\cap$ Segment(S_{P2})

Let $S_{P1} = [t1_1, t1_2]$ and $S_{P2} = [t2_1, t2_2]$.

We want to find $S = S_{P1} \cap S_{P2} = [t1, t2]$ and associate with $t1$ and $t2$ the appropriate normals and colors (either from S_{P1} or S_{P2}, depending on the case).

Upon closer inspection, it is only necessary to calculate $t1$, because $t1$ alone determines whether the given ray intersects a *visible* surface of the CSG object when restricted to that cell (see figure 7). That is, we need only t1 to perform step 6 and 7 of the algorithm for ray-tracing the intersection or difference of primitives. Step 6 is performed by checking whether $0 \leq t1 \leq t_f$ (again, see figure 7). If so, calculate the depth and shading at $o + t1 \cdot v$ with the normal and color associated to $t1$, otherwise the pixel is discarded.

Figure 8 provides a graphical description on how $t1$ is calculated.

Cases 2 and 3: Segment(S_{P1}) $\cap$ 2 semi-intervals(S_{P2})

Let $S_{P1} = [t1_1, t1_2]$ and $S_{P2} = [-\infty, t2_1] \cup [t2_2, \infty]$.

We want to find $S = S_{P1} \cap S_{P2}$. S might be an empty set, a segment or two segments. Figure 9 details the conditions necessary for each of these cases.

Only the left boundaries ($t1$ and $t2$) of each of the two possible segments S might be constituted of is necessary. After calculating $t1$ and $t2$ we check if the smallest, which is always $t1$, of them is in $[0, t_f]$ (See figure 10). If so, we proceed to calculate the depth and shading with $o + t1 \cdot v$ as the intersection point, and with the appropriate normal and color. Otherwise we check whether $t2$ lies in $[0, t_f]$. If so, we proceed to calculate the depth and shading with $o + t2 \cdot v$ as the intersection

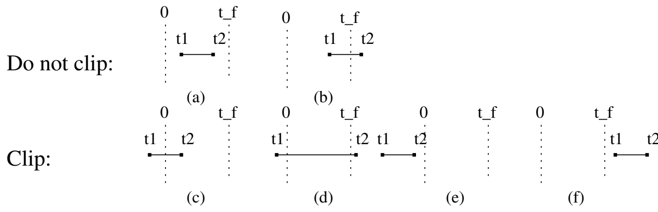


Figure 7: **This figure describes the 6 possible cases of how $S=[t1,t2]$ relates to $S_L=[0,t_f]$. Obviously, cases (a) and (b) should not be clipped (these are determined by $0<t1<t_f$). Cases (e) and (f) also trivially should not be rendered, since $[t1,t2]\cap[0,t_f]=0$. And in cases (c) and (d) we have $[-\varepsilon,\varepsilon]$ contained in $[t1,t2]\cap[0,t_f]$ for ε=min($|t1|$,$|t2|$) - see section 3.3.1 item 7**

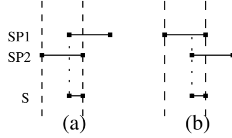


Figure 8: **S is non-empty if (a) the left boundary of S_{P1} is inside $S_{P2}=[t2_1,t2_2]$ or (b) the left boundary of S_{P2} is inside $S_{P1}=[t1_1,t1_2]$. Hence there is an intersection if and only if $t1_1 \leq t2_2$ and $t1_2 \geq t2_1$. In that case $t1$ is the greatest of $t1_1$ (case (a)) and $t2_1$ (case (b)).**

point and with the appropriate normal and color. Otherwise the pixel is discarded.

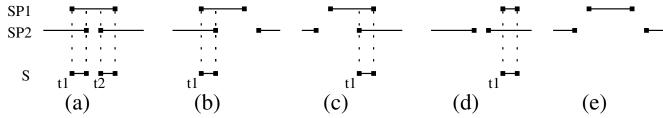


Figure 9: **The four possible cases of combinations of S_{P1} and S_{P2} are depicted above. As can be seen by (a), (b), (c) and (d), t1 exists if either $t1_1 < t2_1$, or $t1_1$ is in $[t2_1,t2_2]$, or $t1_1 > t2_2$. In the first and third cases, t1 is set to $t1_1$ ((a), (b) and (d)). In the second case t1 is set to $t2_1$ (case (c)). As can be seen in (a), t2 exists if $t1_1 \leq t2_2$ and $t1_2 \geq t2_2$.**

Case 4: 2 semi-intervals(S_{P1}) $\cap$ 2 semi-intervals(S_{P2})

Let $S_{P1}=[-\infty,t1_1]\cup[t1_2,\infty]$ and $S_{P2}=[-\infty,t2_1]\cup[t2_2,\infty]$.

We want to find $S=S_{P1}\cap S_{P2}$. S will have 2 semi-intervals, and zero or one segment. Figure 11 exposes the cases where a segment exists in S and the case where there is no segment in S, and is the inspiration behind Algorithm 1.

Only the left boundaries of the left bounded semi-interval and of the possible segment in S are necessary. After calculating $t1$ and $t2$ we set the output parameters $(t, color, normal)$ to the parameters associated with $t1$ and then check whether $t2$ lies in $[0,t_f]$ (Note that we may have either $t1 \leq t2$ or $t1 > t2$. See figure 12). If that is the case, and either $t2 < t1$ or $t1$ is not in $[0,t_f]$, then we set the resulting parameters $(t, color, normal)$ to the the parameters associated with $t1$. At this point, the output parameters will contain either $t = t1$, in which case the pixel may discarded in step 6 (depending on whether t1 is in $[0,t_f]$ or not), or a value in $[0,t_f]$, in which case we proceed to calculating the depth and shading with $o+t\cdot v$ as the

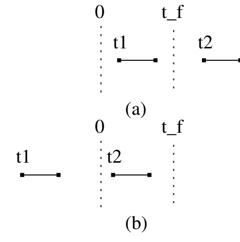


Figure 10: **Case (a) shows a situation where t1 (with t1 in $[0,t_f]$) defines the intersection point. Case (b) depicts a situation where t2 (t1 not in $[0,t_f]$ and t2 in $[0,t_f]$) defines the intersection point. Note that this figure does not go into all possible cases of how S (in this case it has two segments) relates to $[0,t_f]$. Its purpose is only to explain the general idea of the algorithm**

intersection point.

Note that with Algorithm 1 we may end up with a situation as in figure 13 where a pixel is rendered incorrectly, however since the object extends outside the cell towards the camera along the ray, that pixel is bound to be rendered correctly when the cells in front of the current cell (from the camera point of view) are rendered.

Algorithm 1 calc_intersection3(SP_1, SP_2)

$t1 \leftarrow \infty$
$t2 \leftarrow \infty$
if $t2_2 \leq t1_1$ **then**
  $t2 \leftarrow t2_2$
end if
6: **if** $t2_2 > t1_2$ **then**
  $t2 \leftarrow t2_2$
else
  $t1 \leftarrow t1_2$
end if
if $t2_1 \geq t1_2$ **then**
12:   $t1 \leftarrow t1_2$
end if
$t \leftarrow t1$
$color \leftarrow P_1$'s $color$
$normal \leftarrow t1_2$'s $normal$
if ($t2 \in [0,t_f]$) and (($t2 < t1$) or ($t1 \notin [0,t_f]$)) **then**
18:   $t \leftarrow t2$
  $color \leftarrow P_2$'s $color$
  $normal \leftarrow t2_2$'s $normal$
end if
return $(t, color, normal)$

4 Results

4.1 Performance analysis

A few different procedural models were used to analyze the scalability of the algorithm with respect to primitive, depth, area and octree complexities. These models have mainly difference operations because this is the costliest. We also experimented with some models we believe represent the most

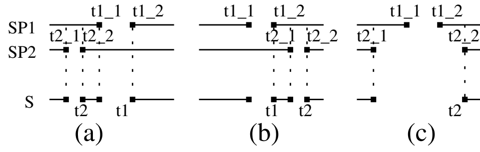


Figure 11: **Case (a) shows a situation where S has 1 segment (this situation is defined by $t2_2 \leq t1_1$). In this case, the left boundary of the segment (t2) will be set to $t2_2$ and t1 will be set to $t1_2$.Case (b) also shows a situation where S has 1 segment (this situation is conditioned by $t2_1 > t1_2$). In this case the left boundary of the segment (t1) will be set to $t1_2$ and t2 will be set to $t2_2$. Case (c) shows a situation where S has no segments (this case is conditioned by $t2_2 > t1_1$ and $t2_1 < t1_2$). In this case t2 is set to $t2_2$ and t1 remains at ∞.**

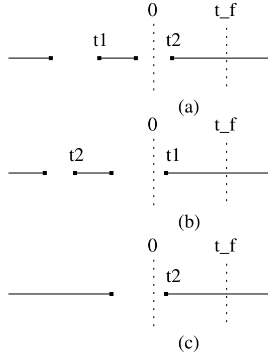


Figure 12: **Case (a) shows a situation where t2 is in $[0, t_f]$ and t1 (t1=∞) is not. Case (b) shows a situation where t2 is not in $[0, t_f]$, so t will retain its initial value ($t = t1$). Case (c) depicts a situation where t2 is in $[0, t_f]$ and t1 ($t1 = \infty$) is not.**

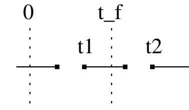


Figure 13: **Algorithm finds incorrect intersection point (should find o but finds $o + t1 \cdot v$)**

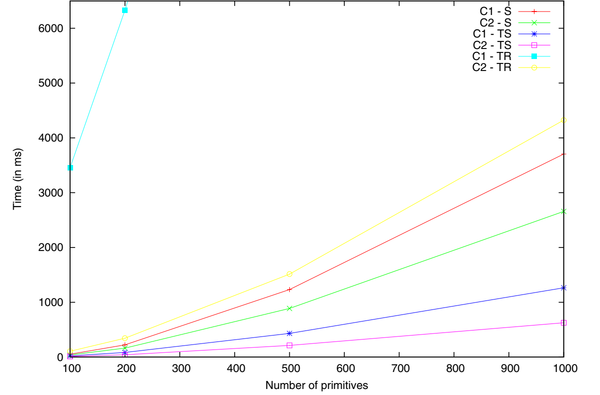


Figure 14: **(a) Graph showing times for CC100, CC200, CC500 and CC1000 (each CCx denotes a cylinder subtracted by x cones). Blue line clipped for visualization purposes - its values at 500 and 1000 are 16046 and 34975, respectively**

common use of CSG models (CAD) to analyze how our implementation scales with the complexity of these models. All timings correspond to rendering in a 640x480 window.

The tests were performed on two configurations: C_1, a Pentium-M 1.4GHz with 512MB of RAM and a Geforce FX Go5200 GPU; C_2, an Athlon 64 3800+ with 1GB of RAM and a Geforce 6800 GT GPU. All tests used a threshold of 1 pixel.

For each model we measured S (time to perform the octree subdivision), T_S (time to traverse the octree and pass the parameters to the GPU without the pixel shaders being active, i.e., without rendering), T_R (time to traverse the octree and render it), S_R (time to perform the octree subdivision and render while subdividing).

We can see from figure 14 that the algorithm scales in a linear fashion with the number of primitives on both test setups. The more complex models also show less improvement from C_1 to C_2 (in T_R). From gpubench analysis, C_2's GPU should have around 20 times more instruction throughput than C_1's GPU. That kind of increase in performance is seen in the less complex models, but for more complex ones that difference can drop to as low as 5 times. This is explained by the fact that these complex models have very large and complex octrees whose subdivision takes significant time, and for each cell of the octree a setup time (to pass primitive parameters, camera and light parameters, etc to the pixel shader) incurs, thus the process becomes more CPU and bandwidth bound, as can be evidenced by comparing T_S and S as the number of primitives increases in figure 14. This indicates that the factor by which the algorithm improves with better GPUs is instruction throughput bound for not so complex models, but gets more bandwidth and CPU bound as models get more complex. The results in table 1 also support these hypotheses. Another bottleneck as models need more complex octrees and thus have a large number of leaf cells is the number of draw calls made during the rendering. For example rendering R9 requires only 2068 draw calls, while CS1000 requires 85306 draw calls, and CC1000 requires 103250.

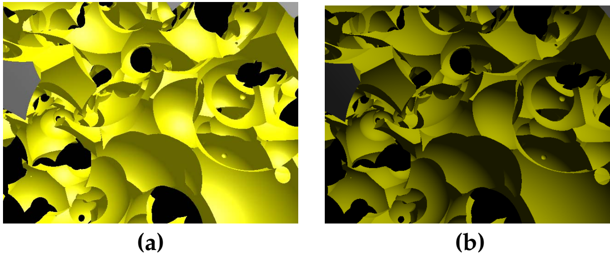
Model	S		T_S		T_R		S_R	
	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
CS1000	1739	856	964	505	10776	2086	11243	2314
CC1000	3705	2658	1265	626	34975	4324	35233	4820
R9	29	20	19	11	1626	78	1677	80
T1022	220	162	86	44	2464	319	2466	324

Table 1: Performance results for several models (all timings in miliseconds)

Also, given a CSG model, its rendering time when varying the camera position will vary with the number of cells that are actually on the visible screen (the smaller the number of cells inside the visible screen the faster the rendering time, as the not visible cells will be clipped and no pixel shader will be executed for the pixels on those cells' front faces), with the area the object occupies on the screen (the larger the area the object occupies on the screen the larger the number of pixels that pixel shaders will be executed for, hence the slowest the rendering times will be) and with its depth complexity (every pixel on the screen will have pixel shaders executed for it as many times as there are cells whose front faces contain that specific pixel, hence the larger the depth complexity the slowest the rendering time).

4.2 Correctness analysis

In order to verify the pixel accuracy of our approach, we compared all our renderizations with renderizations produced by povray, with quality settings comparable to our renderings (+Q3), i.e., without inter-object shadows or anti-aliasing (figure 15 shows one such comparison). In all cases tested there were no visible discrepancies.



(a) **(b)**

Figure 15: **(a) CS1000 viewed up close, rendered using proposed algorithm (b) CS1000 up close, rendered using povray**

5 Conclusions and Future Work

We have presented a new approach to rendering CSG objects composed of convex primitives that has as its main advantage the fact that it relies more on instruction throughput power than on bandwidth (at least for reasonably complex models) as opposed to other existing algorithms. This is a remarkable fact since instruction throughput essentially doubles with every new generation of GPUs whereas the bandwidth improvement has been almost stagnated over the last few generations. Our results are equiparable performance-wise with the currently top performing algorithms, and as newer generations of GPUs come out, from the results obtained, we expect our algorithm to improve its performance at a greater pace than previous algorithms.

Also, as newer GPUs are unveiled it may become feasible to implement efficiently non-convex primitives as well as boolean operations of more than two primitives, thus further unloading the CPU, decreasing the octree size, and requiring less parameter passing to the GPU, which would allow for models with far greater complexity to be rendered at realistic rates. Other structures such as kd-trees may be used to perform better spatial subdivision, on which cells are divided right on points such as in figure 2, thus decreasing the complexity of the tree structures.

Another future improvement would be inter-object shadows. This could be done with a two-pass approach to the problem, rendering to a shadow-buffer first from the light source point of view. Also, preliminary results indicate performing occlusion queries may reduce drastically the dependence of the rendering times on depth complexity and should be further investigated.

References

[1] Bart Adams and Philip Dutre. Interactive boolean operations on surfel-bounded solids. *ACM Trans. Graph.*, 22(3):651–656, 2003.

[2] Joao Luiz Dihl Comba and Ronaldo C Marinho Persiano. Ray tracing otimizado de solidos CSG usando octrees. In *Proceedings of the III SIBGRAPI*, pages 21–30, May 1990.

[3] Gunter Erhart and Robert F. Tobler. General purpose z-buffer CSG rendering with consumer level hardware. Technical Report 2000-003, VRVis, 2000.

[4] Jack Goldfeather, Jeff P M Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proceedings of SIGGRAPH '86*, pages 107–116, 1986.

[5] Jack Goldfeather, Steven Molnar, Greg Turk, and Henry Fuchs. Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Comput Graphics Appl*, 9(3):20–28, May 1989.

[6] Sudipto Guha, Shankar Krishnan, Kamesh Munagala, and Suresh Venkatasubramanian. Application of the two-sided depth test to CSG rendering. In *Proceedings of SI3D '03*, pages 177–180, 2003.

[7] John Hable and Jarek Rossignac. Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes. *ACM Trans. Graph.*, 24(3):1024–1031, 2005.

[8] Eric Haines. *Essential ray tracing algorithms*, pages 33–77. Academic Press Ltd., 1989.

[9] Florian Kirsch and Jurgen Dollner. Rendering techniques for hardware-accelerated image-based CSG. In *Journal of WSCG*, volume 12, pages 221–228, February 2004.

[10] Ari Rappoport and Steven Spitz. Interactive boolean operations for conceptual design of 3-d solids. In *Proceedings of SIGGRAPH '97*, pages 269–278, 1997.

[11] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.*, 12(4):437–464, 1980.

[12] Jarek Rossignac. Blist: A boolean list formulation of CSG trees. Technical Report 99-04, Georgia Institute of Technology, January 08 1999.

[13] Nigel Stewart, Geoff Leach, and Sabu John. An improved z-buffer CSG rendering algorithm. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–30, 1998.

[14] Nigel Stewart, Geoff Leach, and Sabu John. A CSG rendering algorithm for convex objects. *WSCG 2000*, II:369–372, Feb 2000.

[15] Nigel Stewart, Geoff Leach, and Sabu John. Linear-time CSG rendering of intersected convex objects. *WSCG 2002*, II:437–444, Feb 2002.

[16] Rodrigo Toledo and Bruno Levy. Extending the graphic pipeline with new GPU-accelerated primitives. Technical report, INRIA, 2004.

[17] T. F. Wiegand. Interactive rendering of CSG models. *Computer Graphics Forum*, 15(4):249–261, 1996.