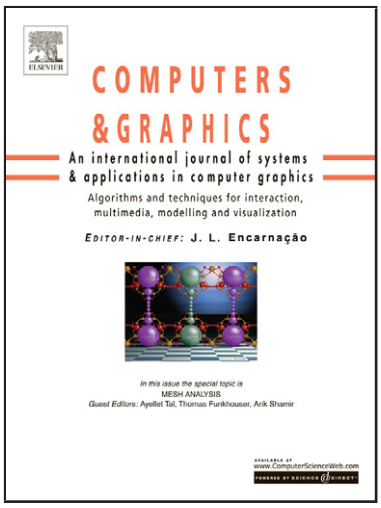


www.elsevier.com/locate/cag

Scalable GPU rendering of CSG models

Fabiano Romeiro, Luiz Velho, Luiz Henrique de Figueiredo

PII: S0097-8493(08)00074-5
DOI: doi:10.1016/j.cag.2008.06.002
Reference: CAG 1863

To appear in: *Computers & Graphics*

Received date: 30 October 2007
Revised date: 28 April 2008
Accepted date: 3 June 2008

Cite this article as: Fabiano Romeiro, Luiz Velho and Luiz Henrique de Figueiredo, Scalable GPU rendering of CSG models, *Computers & Graphics* (2008), doi:10.1016/j.cag.2008.06.002

Scalable GPU Rendering of CSG Models ⋆

Fabiano Romeiro [a,*], Luiz Velho [b],
Luiz Henrique de Figueiredo [b]

[a] *Harvard University, School of Engineering and Applied Sciences*
Cambridge, MA 02138 USA

[b] *IMPA-Instituto Nacional de Matemática Pura e Aplicada*
Estrada Dona Castorina 110, 22460-320 Rio de Janeiro, RJ, Brazil

Abstract

Existing methods that are able to interactively render complex CSG objects with the aid of GPUs are both image based and severely bandwidth limited. In this paper we present a new approach to this problem whose main advantage is its capability to efficiently scale the dependency on CPU instruction throughput, memory bandwidth and GPU instruction throughput. Here, we render CSG objects composed of convex primitives by combining spatial subdivision of the CSG object and GPU ray-tracing methods: the object is subdivided until it is locally "simple enough" to be rendered effectively on the GPU. Our results indicate that our method is able to share the load between the CPU and the GPU more evenly than previous methods, in a way that depends less on memory bandwidth and more on GPU instruction throughput for up to moderately sized CSG models. Even though the same results indicate that the present method is eventually becoming more bus bandwidth and CPU limited with the current state of the art GPUs, especially for extremely complex models, our method presents a solid recipe for escaping this problem in the future by a rescale of the dependency on CPU/memory bandwidth vs. GPU instruction throughput. With this, greater increases in performance are to be expected by adapting our method for newer generation of graphics hardware, as instruction throughput has historically increased at a greater pace than both bus bandwidth and internal GPU bandwidth.

Key words: CSG, Graphics Hardware, GPU, ray tracing

⋆ Based on "Hardware-assisted Rendering of CSG Models", by F. Romeiro, L. Velho and L. H. de Figueiredo, which appeared in Proceedings of SIBGRAPI 2006.

* Corresponding author.
Email addresses: romeiro@fas.harvard.edu (Fabiano Romeiro), lvelho@impa.br (Luiz Velho), lhf@impa.br (Luiz Henrique de Figueiredo).

1 Introduction

One of the most intuitive ways to model solid objects is by constructing them hierarchically: combinations of simple objects generate more and more complex objects. Several representations that incorporate this paradigm exist, and CSG [13] is the most popular.

In the CSG representation, solid objects are obtained by successive combinations of primitives, using boolean operations such as union, difference, and intersection. There, solid objects 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 whose nodes represent boolean operations. A geometric transformation is associated with each node, to allow translation, rotation and scaling of each part of the solid object. It is also possible to associate a complement operation with each leaf node, so that the complement of the primitive in question is considered, instead of the primitive itself.

1.1 Prior work and interactivity

Modeling under the CSG paradigm is extremely intuitive for the user. However, this paradigm would be much more useful if it would allow for interactivity in the modeling process—that is, realtime rendering of solid objects as their CSG representation is modified by the user. Moreover, it is highly desirable that this can be done for ever more complex CSG objects, as then the range of applications of this representation can be expanded. Since the introduction of CSG, several approaches have been devised towards achieving this goal.

Some of these approaches involve converting the CSG representation into a boundary representation and rendering that boundary, but these approaches are not really suitable for interactive performance. Another class of approaches to this problem is image-based: some of which run purely on the CPU, while others make use of special purpose hardware. Goldfeather et al. [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 [23] proposed an implementation of Goldfeather's algorithm on standard graphics hardware. Rappoport and Spitz [12] converted the CSG representation to a Convex Differences Aggregate (CDA) representation and then used the stencil buffer to render at interactive rates. Stewart et al.

[15] improved upon Goldfeather's algorithm and then introduced the Sequenced Convex Subtraction (SCS) algorithm [17], and later refined it [18]. 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 Dutré [1] presented an algorithm that performed 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 [14].

The subset of these algorithms that have reached interactivity on decently sized models ($\sim$ 500 primitives) are image-based, using either depth layering or depth peeling approaches. For this reason, (when not constrained by geometry throughput) they are bandwidth limited — and bandwidth of standard graphics hardware has historically improved at a rate that is much lower than that of instruction throughput. Another disadvantage of these algorithms in comparison to ours is their need to use multiple passes if one wants to render CSG objects with an unlimited number of primitives (due to the number of planes available in the stencil buffer) — although recently an Optimized Blist Form has been devised that lifts this limitation for Blister [16].

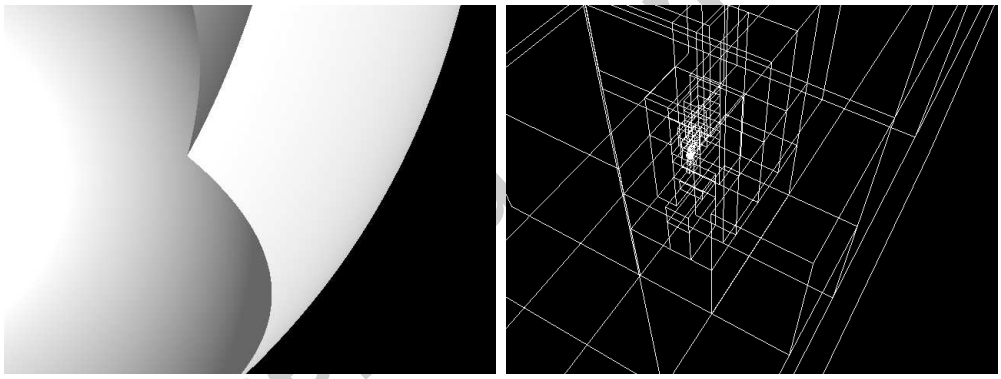
1.2 Our work

We present a model-based method inspired on traditional CPU CSG rendering approaches (e.g., [2]) in the sense that a spatial subdivision of the CSG object is performed. However, unlike in CPU based methods, we do not attempt to trace rays globally through the spatially subdivided structure. Instead, we use the GPU to trace rays locally on each part of the resulting subdivision structure. Because the spatial subdivision is performed until the regions of the CSG object inside each of these parts are "simple enough", this local ray tracing in the GPU can be performed efficiently. We show that this method is essentially instruction throughput limited, rather than bandwidth limited for up to moderately sized CSG models. We also argue that in the future, by changing the concept of "simple enough", it will be possible to shift the balance of this limitation towards using more GPU instruction throughput power. With this, we expect our method to maintain its greater dependency on GPU instruction throughput as newer generations of graphics are released.

2 Spatial Subdivision and CSG

The main insight of our approach is that surfaces of CSG objects are for the most part locally determined by single primitives or by the boolean operations of just two primitives. The exceptions are points of the CSG object that are in the intersection of surfaces of three or more different primitives, i.e., points of order 3 or higher (see Fig. 1). Because ray-tracing of primitives and of boolean operations of two primitives can be done efficiently on the GPU, as will be shown in Sect. 3, rendering the entire CSG object, with the exception of neighborhoods of points of order 3 or higher, reduces to:

(1) Subdividing the CSG object until each part either (i) is composed of a single primitive or a boolean operation of two primitives, or (ii) is insignificant enough to be ignored — where significance is measured by the number of pixels that the given part projects to on the screen — corresponding to either a part that contains one of the exception points, or to one that is still complex even though it does not contribute much to the image and hence can be ignored.
(2) Ray-tracing each part that falls in case (i) on the GPU.



(a) **(b)**

Fig. 1. **(a) Here we have a sphere subtracted by two other spheres. Note that the surface at the protruding point is determined jointly by all the 3 spheres, since it lies at the intersection of the surfaces of those spheres. This point has order 3. (b) Note that, no matter how much the octree is subdivided at the protruding point, there will always exist one cell, namely the one containing this point, which does not have a simple local representation of the CSG object. Hence, we impose a stopping criteria to the subdivision process, to disregard such cells when they are no longer significant.**

This rendering process will produce correct results at most pixels. The exceptions are those corresponding to parts which fall in case (ii). These, however, in most cases correspond to minimal artifacts, which can be minimized or enlarged by changing a threshold — the number of pixels that the given part of the subdivided structure projects to on the screen. Obviously, the

smaller the threshold the more complex the subdivision structure will become. The tradeoff here is quality over subdivision complexity and rendering time. The ability of our approach to regulate this tradeoff could be favorable: 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 Fig. 2).

2.1 Spatial subdivision structures

Subdivision of the CSG object is done using a modified octree. At each level of the octree we divide the existing cells in two instead of in eight, as in traditional octrees. This division is carried sequentially on the X, Y and Z axis, in a cyclic manner, making it a sequenced kd-tree. This binary subdivision greatly improves the CSG tree simplification procedure (see Sect. 2.2.1).

Along with each cell of the octree we keep a CSG tree structure that holds the simplest representation of the surface of the original CSG object when restricted to that cell (i.e., it holds the simplest local representation of the surface of the original object, where here the concept of locality is in the sense of restriction to a given cell).

2.2 Octree subdivision

The subdivision starts with the axis-aligned bounding box of the CSG object as the initial cell of the octree. This cell's CSG tree is set to a copy of the CSG tree representing the CSG object. We proceed by subdividing the cell, setting each of its children's CSG trees to a copy of the cell's own CSG tree, and then simplifying each child cell's CSG tree to obtain the simplest CSG tree that still correctly represents the restriction of the surface of the original CSG object to each child cell. This cell subdivision followed by CSG tree simplification is repeated recursively for each of the cell's children, 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 the union of an arbitrary number of primitives on the GPU can be done efficiently — see Sect. 3.3.3 — hence we do not further subdivide and render these cases right away).
(3) The cell's CSG tree has depth 2, but is not an union of two primitives (which would fall into case 2), i.e., the part of surface of the CSG object that is inside this cell is determined by two primitives and is thus

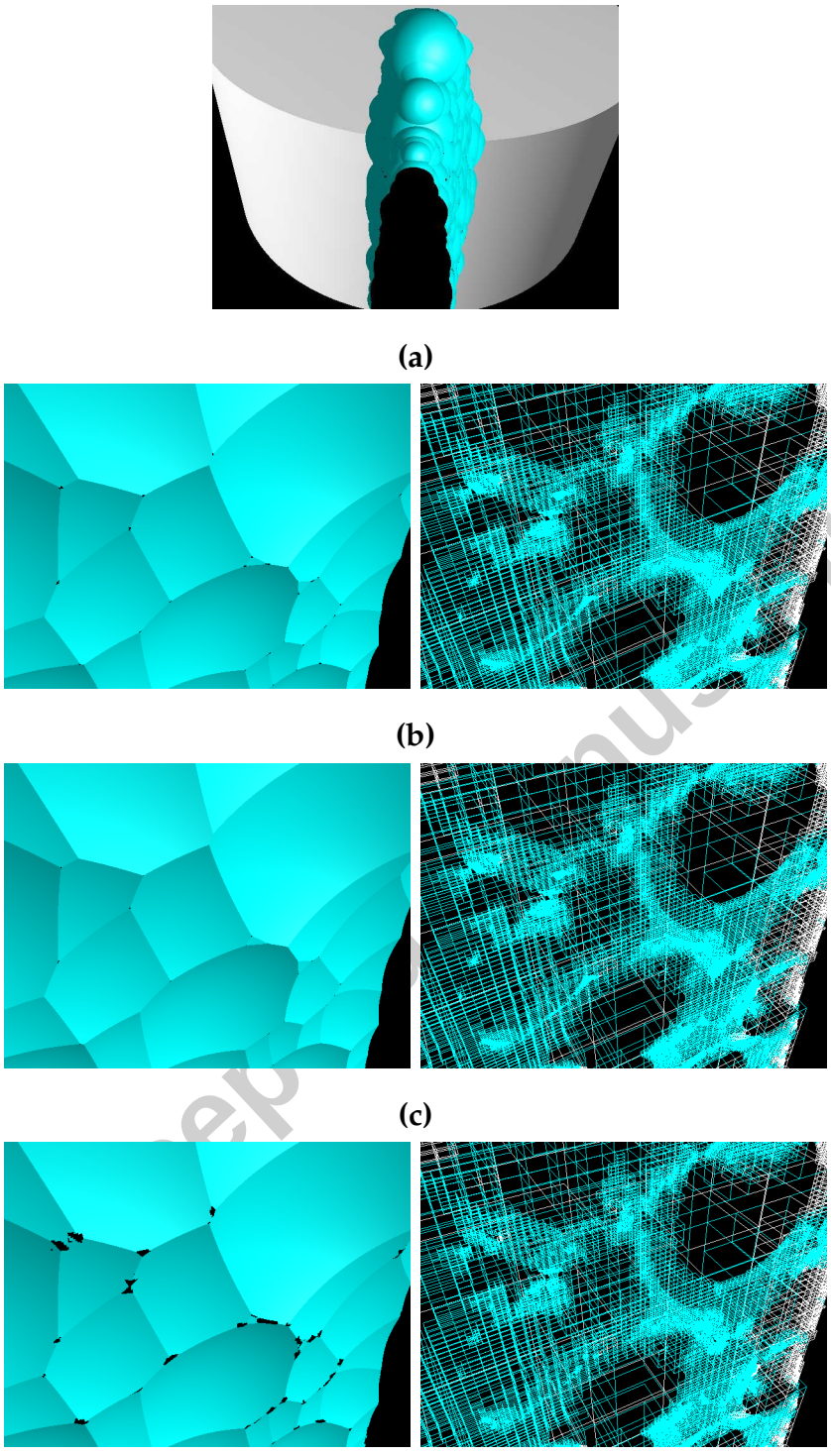
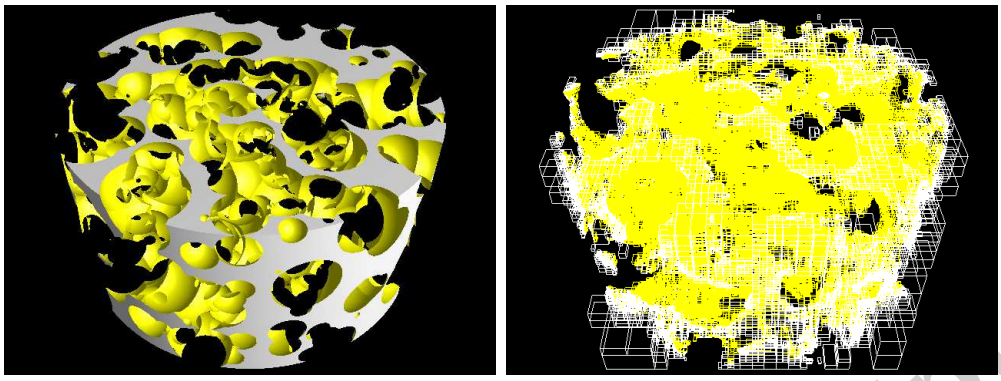


Fig. 2. **(a) Using a threshold of 1 pixel usually produces no noticeable artifacts. (b) However in some scenes artifacts can be quite noticeable with that threshold, as evidenced by the zoom in on the walls. (c) Using smaller thresholds (here, 0.5 pixels) improves image quality by reducing artifacts and (d) increasing the threshold (5 pixels) produces the opposite effect. Note how changing the threshold affects the complexity of the octree around the artifacts. (Please zoom in for better visualization of artifacts.)**

simple enough to be rendered efficiently on the GPU.

(4) The cell projects onto less than a threshold of pixels in the screen, i.e., the cell is insignificant.



(a) **(b)**

Fig. 3. **(a) A reasonably complex CSG model, composed of a cylinder subtracted by 1000 randomly placed spheres and (b) its final octree: each cell drawn here 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).**

Cells that fall in either of these cases are called *leaf cells* and all the others are called *node cells*. Also, in practice we approximate the threshold test of the projected area of a cell by threshold tests on the lengths of the projections of its edges onto the visible screen (i.e. the parts of the edges that lie within the visible screen): if all edges of a cell project to lengths of less than the given threshold we claim the projected area to be less than the threshold.

Note that at each level of subdivision we do not start with the original CSG tree, but rather with the CSG tree of the parent cell, which is already a simplification of the original CSG tree (which 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: a faster simplification process.

2.2.1 CSG tree simplification scheme

As mentioned before, the goal of our CSG tree simplification scheme is to, given a cell and a CSG tree, simplify the CSG tree as much as possible, ending with the simplest CSG tree such that the restriction of the surface of the original CSG object to that cell is correctly represented by the simplified CSG tree. This is done in a standard fashion, by pruning sub-trees of the CSG tree whose surfaces (of the object they represent) 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 Sect. 2.2 is reached it is ignored. 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 object given by the cell's CSG tree to the cell's interior) is rendered in the GPU, as detailed in the next section.

It is also possible to render the CSG object while subdividing it. By doing so, parallelization of the subdivision on the CPU and the rendering on the GPU is achieved (see Sect. 4). In our analysis, we consider both cases (generating octree and then traversing, as well as rendering while octree is being generated), since for visualization purposes, once the subdivision has been completed for a given CSG object, it need not be recomputed in order for the object to be viewed under different lighting conditions. In a similar fashion, to visualize the object under a new angle, only minimal updates to the octree are required, provided the new angle is close to the previous angle (which is a valid assumption in the case of interactive visualization), because the only cells that possibly need to be updated are the ones that contain an intersection of three or more objects and no longer project to less than a threshold of pixels.

3 Ray-tracing CSG objects in the GPU

Recent generations of GPUs have allowed the design of algorithms that have a substantial part of their workload performed by the GPU. Several developments have been made toward transferring ray-tracing of objects from the CPU to the GPU. Some of the early efforts in this direction addressed volume rendering on the GPU [10]. Also, the use of proxy geometry to obtain ray parameters on the GPU has been suggested in the hybrid approach of Westermann and Sevenich [22]. Here, we extend on the work developed by Toledo and Levy [19], to ray-trace in the GPU not only a set of convex primitives (or their complements), but also boolean operations of them (or their complements).

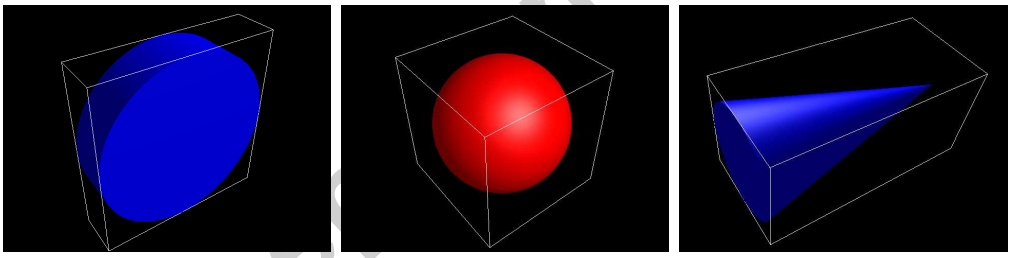
Because we render the restriction of the CSG object to each cell, we also deal with the more specific problem of rendering only the parts of primitives or boolean operations of primitives that are inside a given cell. This is essentially done by subsequently clipping the results of the original ray-tracing algorithm to the corresponding cell, as will be detailed later on. Sect. 3.1 is

an introductory presentation to the concepts of ray-tracing GPU primitives; for the sake of clarity, it does not incorporate this clipping. In Sects. 3.2 and 3.3 we detail at length how to ray-trace single primitives and boolean operations of primitives, respectively, when this simplification is lifted.

3.1 Concepts of ray-tracing GPU primitives

The basic idea of ray-tracing primitives in the GPU, as introduced by Toledo and Levy [19], is to render some object (e.g., a bounding box) whose projection on the screen (called the ray-tracing area or RTA for short) covers the projection of the desired primitive (see Fig. 4) after having bound appropriate vertex and pixel shaders. With this, the shader will run for every point of each of the faces of the rendered object. Then, in the shader, the appropriate ray-tracing can be performed to determine which color value should be used in each of these points to achieve the rendering of the desired primitive.

More specifically, given a point on one of the faces of the rendered object, the shader will trace a ray from that point, in the direction of the camera, determining whether the ray intersects the primitive or not, and performing the shading, if appropriate (see Sect. 3.1.2).



(a) **(b)** **(c)**

Fig. 4. **The front faces of the bounding box of a (a) cylinder (b) sphere (c) cone is rendered. The pixel shader then runs on each pixel of these front faces ray-tracing the respective primitives.**

3.1.1 Bounding box as the ray-tracing area

In this paper, we use Axis Aligned Bounding Boxes (AABBs) as the rendering objects (i.e., the cells of the octree structure defined in the previous section), and their front faces as RTAs. While this works, we realize that it might not be the most efficient rendering object for every case, because depending on the primitive and the camera position a different rendering object might still circumscribe the desired primitive while having a smaller

projected area (RTA). For example, a better suited rendering object 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 rendering object, while it would still be able to ray-trace the cone properly. A more thorough analysis of the efficiency of different rendering objects and their RTAs is provided by Toledo and Levy [19].

3.1.2 The roles of vertex and pixel shaders

In this context, when rendering the front faces of the bounding box of a primitive, 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. For each pixel of the projection of this bounding box, the pixel shader receives this information, as well as parameters containing information on the primitive in question (e.g., for a sphere: center, radius), calculates the ray direction, traces the ray starting from the point at the bounding box corresponding to the pixel in question, and finally calculates the ray-primitive intersection that is closest to the camera, as well as the primitive's normal at the intersection. Given the latter two, the pixel shader computes the pixel's color and depth, as detailed in Sect. 3.2.

3.2 Ray-tracing of simple convex primitives

While the approach described above works for non-convex primitives, from here on we will focus solely on convex primitives. With this restriction, we are assured that any given ray-primitive intersection result will either be an empty set or consist of a single segment (which can be stored with only two scalar variables). Thus, we can have an efficient implementation of the ray-primitive intersection and other necessary algorithms on the pixel shader.

In this scenario, given a cell L and a primitive P, in order to ray-trace P restricted to L we use the framework described in Sect. 3.1 with a pixel shader that performs the following steps:

(1) $o \leftarrow$ Object space coordinates of point for which shader is running
$v \leftarrow$ Camera direction
$r(t) = o + t \cdot v \leftarrow$ Ray in camera direction starting at o
(2) **Ray-cell intersection:** Intersect L with $r(t)$.
Obtain a segment $S_L = [0, t_f]$ as a result (i.e., the set of all t such that $r(t)$ and L intersect). [1] ,

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

(3) **Ray-primitive intersection:** Intersect P with $r(t)$.
Obtain S_P, a segment (degenerate or not) or an empty set. Calculate surface normals at the ray-surface intersection points (if any).
(4) **Complementing:** If P is complemented, complement S_P and the respective ray-surface intersection normals.
(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_P for some $\varepsilon > 0$, discard the pixel. [2]
(9) Else, the ray intersects the surface of P inside L at $r(t') = o + t' \cdot v$, and then we:
- **Depth Calculation:** Calculate the correct depth value for this pixel (this was originally set to the depth of o, which lies in one of the faces of the cell, and must be updated to the depth of $r(t')$).
- **Shading calculation:** Calculate the color of the pixel, given t''s associated color and surface normal (i.e., perform the shading given our lighting model and surface properties).

Next, we describe in further detail each part of the above algorithm.

Ray-cell intersection: Since a cell is nothing more than an AABB, specified by its lower left corner *ll* and its upper right corner *ur*, intersecting rays with AABBs is relatively easy and is done with a simplified version of an algorithm described by Haines [8] (see Alg.1). Note that we only need to compute t_f. Note that even though in practice a division by 0 might occur (if one or more of the components of v are 0), it is always the case that at least one of the 3 max operations will produce a finite result, and thus the result is always correct even without a check for division by 0.

Algorithm 1 rayCellIntersection(ll, ur, o, v)

$v_l \leftarrow (ll - o)/v$
$v_u \leftarrow (ur - o)/v$
$t_f \leftarrow \min(\max(v_l.x, v_u.x), \max(v_l.y, v_u.y), \max(v_l.z, v_u.z))$

Ray-primitive intersection: We calculate the boundaries of the ray-primitive intersection set by replacing the ray equation ($r(t) = o + t \cdot v$) into the equation for the surface of the primitive in question (e.g., $||x - c|| = a$ for a sphere centered in c with radius a) and solving for t. At the same time, the primitive normals at these boundary points are also calculated. Even though in

[2] If this is the case, the object extends outside the cell towards the camera, since the cell is in the range $[0, t_f]$. This means that the object is being 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 part of the surface that was clipped, is rendered.

our system we only implemented ray-primitive intersection algorithms for cylinders, cones, and spheres, it would be easy to implement similar ones for other implicit surfaces defined by low degree polynomials, such as more general quadrics or cubics. We provide pseudo-code for the case of a sphere in Alg. 2.

Algorithm 2 raySphereIntersection(o, v, a, c)

$t_1 \leftarrow \infty$
$t_2 \leftarrow \infty$
$o \leftarrow o - c$
$\delta \leftarrow (o \cdot v)^2 - o \cdot o + a \cdot a$
if $\delta \geq 0$ **then**
  $\delta \leftarrow \sqrt{\delta}$
  $t_1 \leftarrow -o \cdot v - \delta$
  $t_2 \leftarrow -o \cdot v + \delta$
  $normal_1 \leftarrow o + t_1 \cdot v$
  $normal_2 \leftarrow o + t_2 \cdot v$
end if
return $(t_1, normal_1), (t_2, normal_2)$

Depth Calculation: Depth calculation is performed by transforming $r(t')$ from object-space to eye-space coordinates, dividing by the homogeneous coordinate and re-scaling from $[-1, 1]$ to $[0, 1]$ (Alg. 3).

Algorithm 3 calcDepth(o, t, v)

$MV \leftarrow$ ModelView matrix.
$otv_{eye} \leftarrow MV \cdot (o + t \cdot v)$
$depth \leftarrow 0.5 + 0.5 \cdot (otv_{eye}.z / otv_{eye}.w)$

Shading calculation: We use positional lighting and the Blinn reflection model to perform the shading. Alg. 4 describes the details of this calculation. There, n is the surface normal at $r(t')$, l is the unit vector from the positional light source to $r(t')$, aC is the ambient color, dC is the diffuse color and sC is the specular color. Note that although in Alg. 4 we use Blinn-Phong shading, any other local shading model could be applied.

Algorithm 4 calcLighting(l, n, v, aC, dC, sC)

halfVec $\leftarrow (l + v)/||l + v||$
color $\leftarrow aC + dC \cdot \max\{n \cdot l, 0\} + sC \cdot \max\{n \cdot \text{halfVec}, 0\}^{16} \cdot (n \cdot l > 0)$

Complementing: Depending on the set S_P to be complemented, there are two possible cases (in either case, the complement of S_P will consist of two semi-intervals). We take different actions in each case, as specified below.

(1) S_P is an empty set.
   - In this case, we set the new S_P to be $S_P = (-\infty, t] \cup [t, +\infty)$ for some t (any t).

(2) S_P is a segment (i.e., $S_P = [t_0, t_1]$).
   - In this case, we set the new S_P to be $S_P = (-\infty, t_0] \cup [t_1, +\infty)$, and compute the new normals at the intersection points of the complemented primitive (corresponding to t_0 and t_1) by inverting the sign of the normals calculated in step 3.

3.3 Ray-tracing boolean operations of simple convex primitives

Now that we have described how to ray-trace convex primitives or their complements, we tackle the problem of ray-tracing boolean operations of them (or their complements). In Sect. 3.3.1, we describe how to efficiently perform the ray-tracing when the boolean operation is an intersection (in the sense that the pixel shader implementation is efficient). In Sect. 3.3.2, we show how to adapt this algorithm to handle the case where the boolean operation is a difference. And finally, we detail how we handle the case where the boolean operation is a union in Sect. 3.3.3.

3.3.1 Intersection of primitives

In this case, 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 use the same framework described in Sect. 3.1.2, with a pixel shader that performs the following steps:

(1) **Ray-cell intersection:** Intersect L with $r(t)$.
Obtain a segment ($S_L = [0, t_f]$) as a result.

(2) **Ray-primitive intersection:** Intersect P_1 with $r(t)$.
Obtain S_{P1}, either a segment (degenerate or not) or an empty set as a result. Calculate surface normals at the ray-surface intersection points (if any).

(3) **Ray-primitive intersection:** Intersect P_2 with $r(t)$.
Obtain S_{P2}, as in the previous step.

(4) **Complementing:** Complement S_{P1} and the respective normals if P_1 is complemented and likewise for S_{P2}.

(5) **Intersection of ray-primitive result sets:** Intersect S_{P1} with S_{P2}.
Obtain S. Associate with each boundary of S the appropriate normals and colors, either from S_{P1} or S_{P2} — more on this in Sect. 3.3.4.

(6) Clip S to L, obtaining S_R.

(7) Perform steps 6 to 9 of Sect. 3.2.

3.3.2 *Difference of primitives*

The pixel shader to ray-trace the difference of primitives is exactly the same as the one in Sect. 3.3.1, with one exception. 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*

Ray-tracing the union of primitives constitutes a special case, as it can be performed easily due to the z-buffer of the GPU. Given a cell L and a collection of primitives $P_1, \ldots, P_k$, to ray trace $\cup P_n$ restricted to L one needs only to render $P_1, \ldots, P_k$ restricted to L independently, as explained in Sect. 3.2. The z-buffer will then ensure that the correct color values get assigned to each pixel, given the depth of each of the primitives.

3.3.4 *Intersection of ray-primitive result sets*

Most of the steps of the algorithm of Sect. 3.3.1 were already detailed in Sect. 3.2. In this section we describe step 5, the only step missing an explanation. Here, we show how to intersect S_{P1} and S_{P2} in a way that its implementation on a pixel shader is efficient.

Depending on whether P_1 and P_2 are complemented, we have four possible cases 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

Instead of handling these four cases in a single extremely complex pixel shader, we implement each of them in a different pixel shader. Thus, depending on the primitives being complemented or not, we load up a different pixel shader. The main reason for this is that branching currently causes a big impact on performance (more so than switching shaders), hence it is more desirable to have one shader for each case than fours cases in a single

shader. In addition to that, some of these cases can be handled in much simpler ways than others, and this allows us to have simpler and more efficient pixel shaders for these cases. Namely, calculating the intersection of two segments is simpler than calculating the intersection of two semi-intervals and 1 segment, which in turn is simpler than calculating the intersection of two pairs of semi-intervals.

It is very possible that in the future, with more efficient branching in GPUs, the performance difference between having separate shaders or a unified one will be negligible and thus a unified shader will be preferable.

Next, we provide the details of the algorithms for each of these four cases.

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 specific situation).

Upon closer inspection, it is only necessary to calculate $t1$. This is because $t1$ alone determines whether the given ray intersects a visible surface of the CSG object when restricted to a cell (see Fig. 5). That is, we need only $t1$ to perform steps 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 Fig. 5). If that is the case, depth and shading are calculated at $o + t1 \cdot v$ with the normal and color associated to $t1$, otherwise the pixel is discarded.

Fig. 6 provides a graphical description of how $t1$ is calculated, while Alg. 5 contains the corresponding pseudo-code.

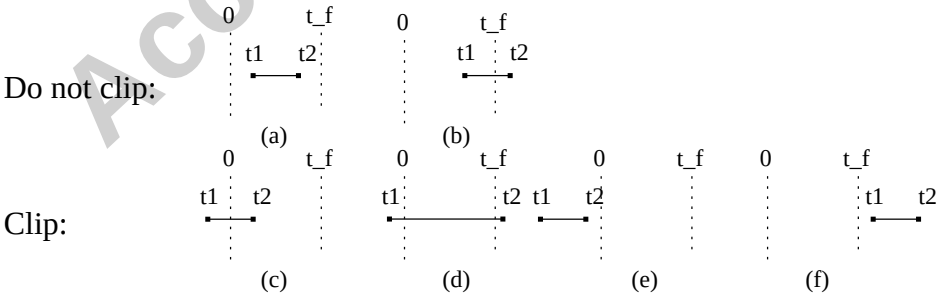


Fig. 5. **This figure describes the 6 possible cases of how $S = [t1, t2]$ relates to $S_L = [0, t_f]$, i.e., it describes the possible cases of the clipping operation. Obviously, cases (a) and (b) should not be clipped (these are determined by $0 < t1 < t_f$). Cases (e) and (f) trivially should not be rendered, since $[t1, t2] \cap [0, t_f] = \emptyset$. And in cases (c) and (d) we have $[-\varepsilon, \varepsilon]$ contained in $[t1, t2] \cap [0, t_f]$ for $\varepsilon = \min(|t1|, |t2|)$ (see Sect. 3.3.1, step (7)).**

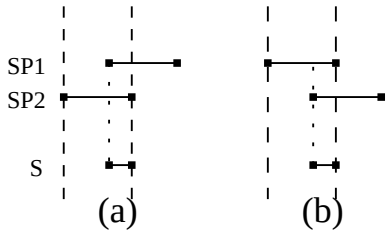


Fig. 6. S **is non-empty if (a) the left boundary of** S_{P1} **is inside** $S_{P2} = [t2_1, t2_2]$ **or (b) the left boundary of** S_{P2} **is inside** $S_{P1} = [t1_1, t1_2]$**. Hence the intersection is non-empty if and only if** $t1_1 \leq t2_2$ **and** $t1_2 \geq t2_1$**. In that case** $t1$ **is the greatest of** $t1_1$ **(case (a)) and** $t2_1$ **(case (b)).**

Cases 2 and 3: Segment(S_{P1}) ∩ two 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 $S = S_{P1} \cap S_{P2}$. Here, S might be an empty set, a segment or two segments, depending on S_{P1} and S_{P2}. Fig. 7 details each of the possible cases.

Only the left boundaries, $t1$ and $t2$, of each of the two possible segments that might constitute S, are necessary. After calculating $t1$ and $t2$ we check if the smallest of them, which we define to be $t1$ in our algorithm, is in $[0, t_f]$ (see Fig. 8). 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 that is the case, we proceed to calculate the depth and shading with $o + t2 \cdot v$ as the intersection point and with the appropriate normal and color. Finally, if that is not the case, the pixel is discarded (see Alg. 6).

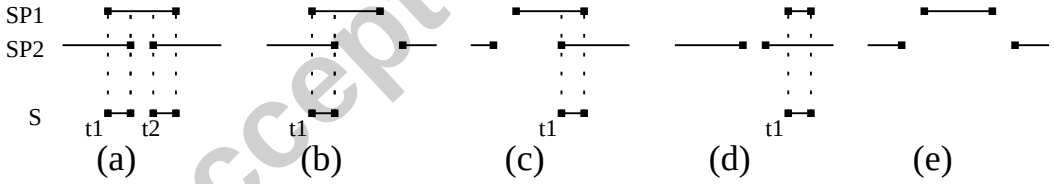


Fig. 7. **The four possible cases of combinations of** S_{P1} **and** S_{P2} **are depicted above. As can be seen by (a), (b), (c) and (d), t1 exists if either** $t1_1 < t2_1$**, or** $t1_1$ **is in** $[t2_1, t2_2]$ **and** $t1_2 > t2_2$**, or** $t1_1 > t2_2$**. In the first and third cases, t1 is set 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: two semi-intervals(S_{P1}) ∩ two 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 $S = S_{P1} \cap S_{P2}$. Here, S will have two semi-intervals, and zero or one segment. Fig. 9 exposes cases where S contains a segment as well as a case where S does not contain a segment, and is the inspiration behind Alg. 7.

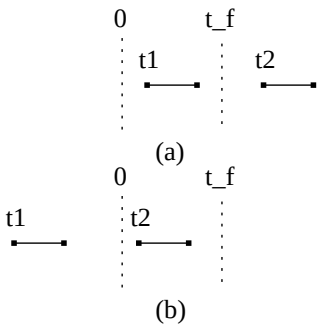


Fig. 8. **Case (a) shows a situation where t1 (with t1 in $[0,t_f]$) defines the intersection point. Case (b) depicts a situation where t2 (t1 not in $[0,t_f]$ and t2 in $[0,t_f]$) defines the intersection point. Note that this figure does not go into all possible cases of how S (in this case it has two segments) relates to $[0,t_f]$. Its purpose is only to explain the general idea of the algorithm.**

It is straightforward to note that 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 Fig. 10). 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 $t2$. At this point, the output parameters will either contain a value for t that is not in $[0,t_f]$ (since we initially set $t = t1$ without checking for that), in which case the pixel will be discarded in step 6, or a value for t that is in $[0,t_f]$, in which case we proceed to calculate the depth and shading with $o + t \cdot v$ as the intersection point.

Note that, with this algorithm, we may end up with a situation as in Fig. 11, where a pixel is rendered incorrectly. However, in this case the object extends outside the cell towards the camera along the ray, and thus that pixel is bound to be rendered correctly when the cells in front of the current cell, from the point of view of the camera, are rendered.

4 Results

4.1 Performance analysis

Since the difference operation is the costliest of the boolean operations in our implementation, the majority of the test CSG models chosen were made mainly with these operations. Additionally, most of the models used in our testing setup were chosen to be somewhat similar to models that would be

Algorithm 5 calcIntersection(SP_1, SP_2)

$t1 \leftarrow \infty$
if $(t1_1 \leq t2_2)$ and $(t1_2 \geq t2_1)$ **then**
  if $(t1_1 \geq t2_1)$ **then**
    $t1 \leftarrow t1_1$
    $color \leftarrow P_1$'s $color$
    $normal \leftarrow t1_1$'s $normal$
  else
    $t1 \leftarrow t2_1$
    $color \leftarrow P_2$'s $color$
    $normal \leftarrow t2_1$'s $normal$
  end if
end if
return $(t, color, normal)$

Algorithm 6 calcIntersection2(SP_1, SP_2)

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

associated with CAD applications, since this is perhaps where CSG models are most commonly used. In Figs. 12 and 13, we see a few of the different procedural models used in our testings.

For each of these models we performed a number of measurements to help

Algorithm 7 calcIntersection3(SP_1, SP_2)

$t1 \leftarrow \infty$
$t2 \leftarrow \infty$
if $t2_2 \leq t1_1$ **then**
  $t2 \leftarrow t2_2$
end if
if $t2_2 > t1_2$ **then**
  $t2 \leftarrow t2_2$
else
  $t1 \leftarrow t1_2$
end if
if $t2_1 \geq t1_2$ **then**
  $t1 \leftarrow t1_2$
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**
  $t \leftarrow t2$
  $color \leftarrow P_2$'s $color$
  $normal \leftarrow t2_2$'s $normal$
end if
return $(t, color, normal)$

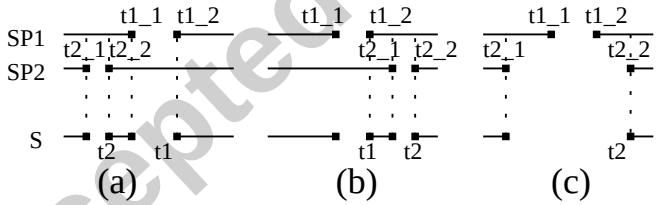


Fig. 9. **Case (a) shows a situation where S has 1 segment (this situation is defined by $t2_2 \leq t1_1$). In this case, the left boundary of the segment (t2) will be set to $t2_2$ and t1 will be set to $t1_2$.Case (b) also shows a situation where S has 1 segment (this situation is conditioned by $t2_1 > t1_2$). 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 ∞.**

us analyze the scalability of our algorithm with respect to primitive and octree complexities. We measured: the time to perform the octree subdivision, S; the setup time, that is, the time to traverse the octree and pass the parameters to the GPU, but without ray-tracing (i.e., with the fragment and vertex profiles disabled), T_S; the time to traverse the octree and render it, T_R; the time to perform the octree subdivision and render while subdividing, S_R; and the time to render the object in povray T_{pov} (with the settings

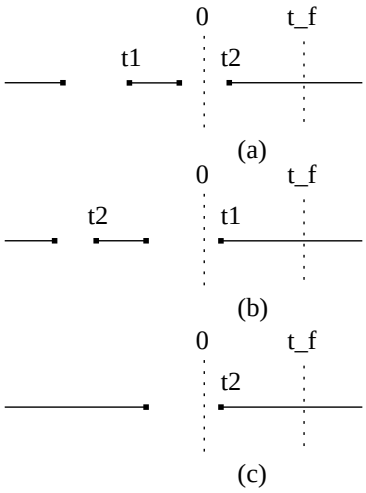


Fig. 10. **Case (a) shows a situation where t2 is in $[0,t_f]$ and t1 (t1=∞) is not. Case (b) shows a situation where t2 is not in $[0,t_f]$, so t will retain its initial value ($t = t1$). Case (c) depicts a situation where t2 is in $[0,t_f]$ and t1 ($t1 = \infty$) is not.**

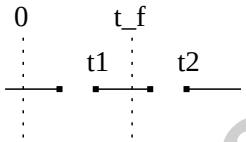


Fig. 11. **Algorithm finds incorrect intersection point (should find o but finds $o + t1 \cdot v$)**

detailed in Sect. 4.2).

All tests were performed on four 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; C_3, a Pentium D 2.8Ghz with 1GB of RAM and a Geforce 7950 GT GPU; and C_4, a 2x Quad-core Xeon 2Ghz with 16GB of RAM and a Quadro FX 5600 GPU (approximately equivalent to a Geforce 8800). A threshold of 1 pixel was used in all tests for the spatial subdivision's stopping criteria. From Table 1 we observe that the increase in performance from C_1 through C_4 far surpasses the increase in CPU performance and CPU/GPU memory bandwidth. To see this, compare the increase in performance in S and T_S versus T_R in Figs. 14 and 15 - while subdivision performance (CPU) increases only twofold between C_1 and C_4 and CPU/GPU bus bandwidth remains almost unchanged (T_S, even though CPU performance also impacts this measure), traversal and rendering times increase up to 20 times from C_1 to C_2, 3 times from C_2 to C_3 and 2 times from C_3 to C_4.

Also noticeable, is the fact that more complex models show less improvement from C_1 through C_4 (in both time to traverse and render, T_R and time to subdivide and render, S_R). From gpubench analysis, the Quadro FX5600

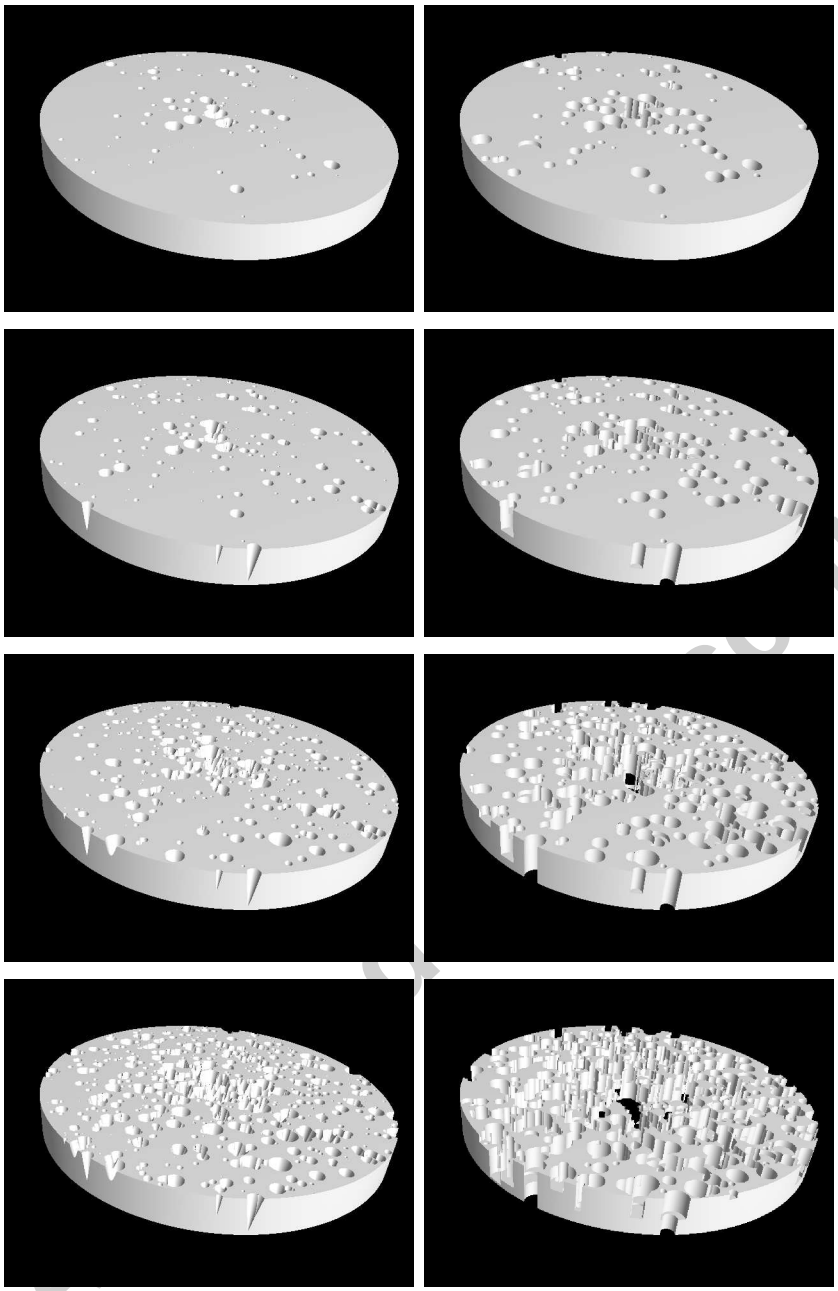


Fig. 12. **First column: a cylinder subtracted by 100 (CONE100), 200 (CONE200), 500 (CONE500) and 1000 (CONE1000) cones, respectively (CSG tree given by (Cyl \ Cone) \ Cone ...). Second column: a cylinder subtracted by 100 (CYL100), 200 (CYL200), 500 (CYL500) and 1000 (CYL1000) cylinders, respectively (CSG tree given by (Cyl \ Cyl) \ Cyl ...).**

should have around 2 times the instruction throughput of the Geforce 7950GT which in turn should have around 4 times the instruction throughput of the Geforce 6800GT, which finally should have around 20 times the instruction throughput of the Geforce FX Go5200. In fact, as we have discussed, for the most part we observe these kind of increases in performance with less com-

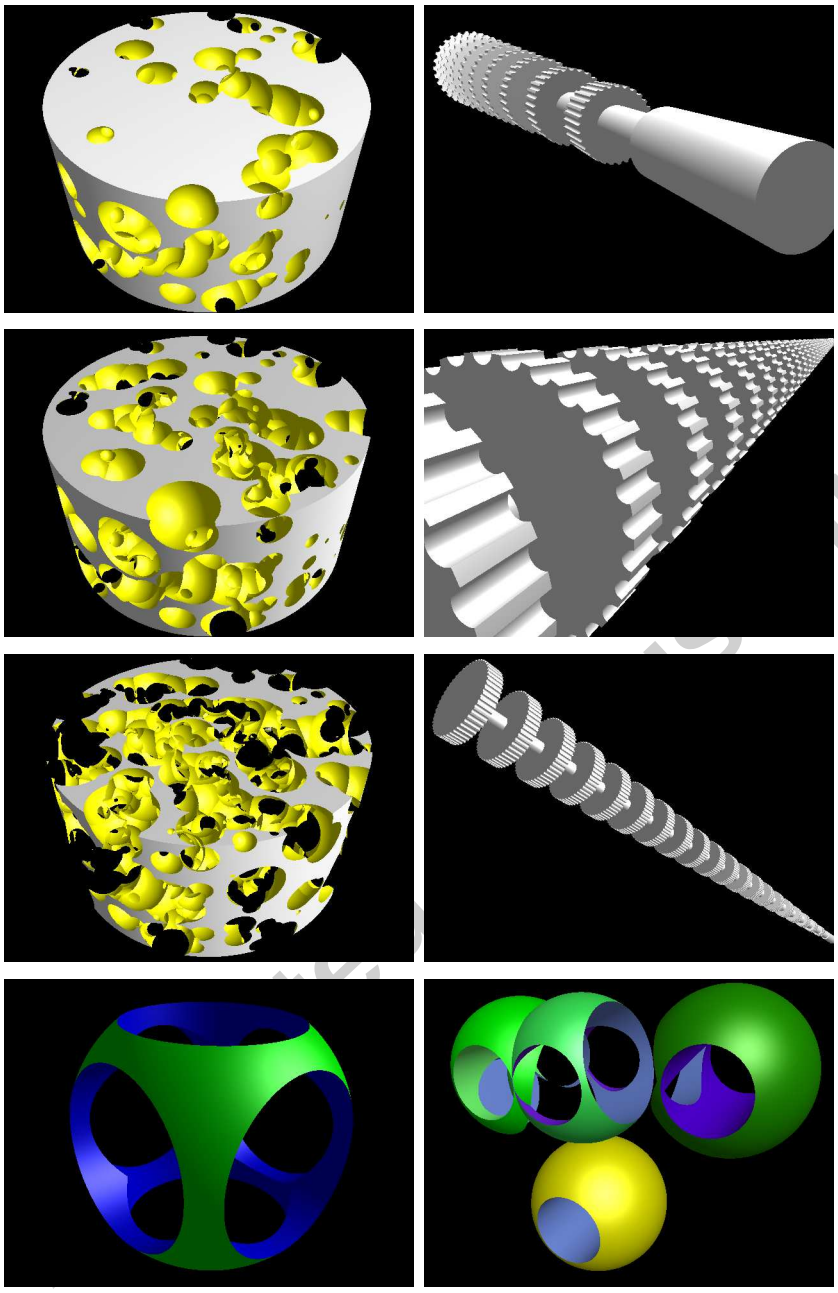


Fig. 13. **First row: a cylinder subtracted by 250 spheres (CHEESE250); the union of 30 cylinders, each subtracted by 10 cylinders, forming a tube (TUBE300). Second row: a cylinder subtracted by 500 spheres (CHEESE500); the union of 30 cylinders, each subtracted by 20 cylinders (TUBE500). Third row: a cylinder subtracted by 1000 spheres (CHEESE1000); the union of 50 cylinders, each subtracted by 20 cylinders (TUBE1000). Fourth row: a sphere subtracted by 3 cylinders (WIDGET); object formed by 9 primitives, among which spheres and cylinders (R9). For all the cheese models, CSG tree given by ((Cyl \ Sphere) \ Sphere ...).**

Model	T_R			S_R			T_{pov}	
	*Go*5200	6800	*Q*5600	*Go*5200	6800	*Q*5600	PentM1.4	Ath3800
CHEESE250	5501	567	161	5506	631	361	62s	33s
CHEESE500	9070	1402	400	9287	1521	982	157s	134s
CHEESE1000	10776	2086	1001	11243	2314	1627	431s	373s
CYL500	23074	2833	234	23231	3240	1097	159s	136s
CONE1000	34975	4324	762	35233	4820	2472	413s	388s
R9	1626	78	17	1677	80	31	2s	1s
WIDGET	2105	46	17	2102	46	17	2s	1s
TUBE1000	2464	319	84	2466	324	265	3s	2s

Table 1
Results for several of the models in Figs. 13 and 12, in milliseconds (unless otherwise noted). Each value corresponds to an average over 10 measurements.

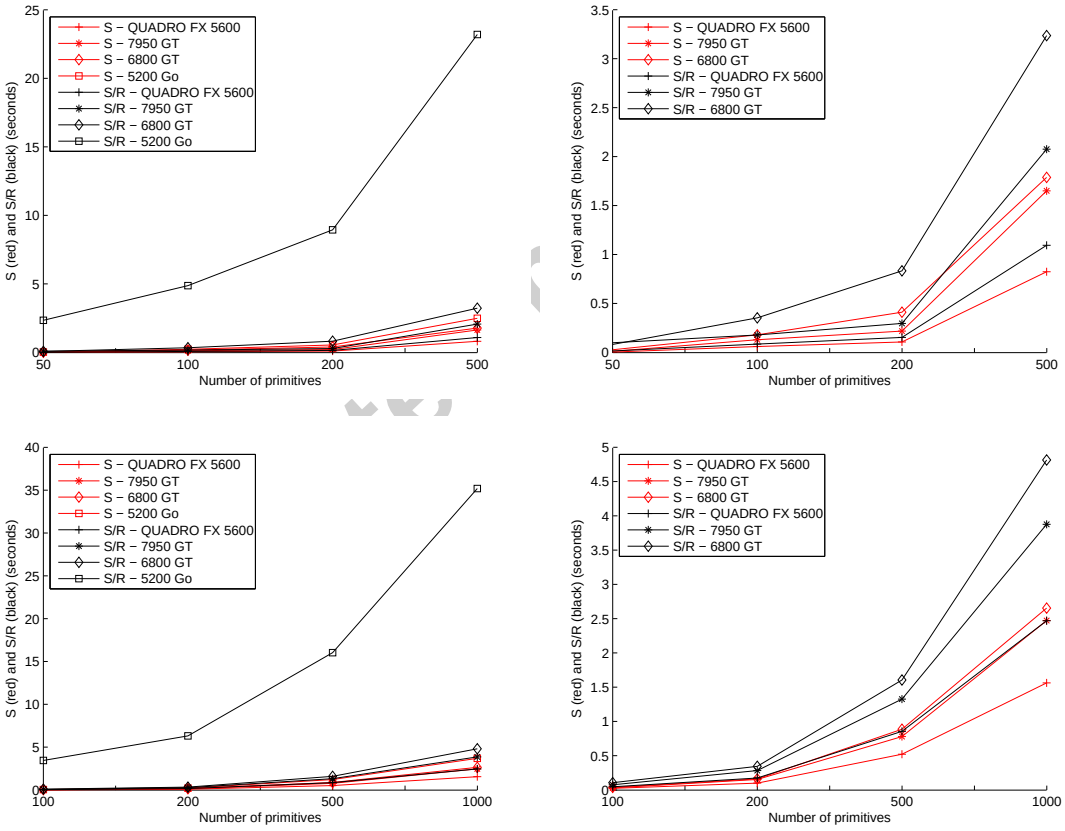


Fig. 14. **Time to subdivide (S, in red) and subdivide and render (S/R, in black). First row: timings for cylinder subtracted by 50, 100, 200 and 500 cylinders (Fig. 12, second column). Second row: timings for cylinder subtracted by 100, 200, 500 and 1000 cones (Fig. 12, first column). In each row, the first column presents timings for all configurations, while the second column presents timings for the 3 faster configurations with more detail.**

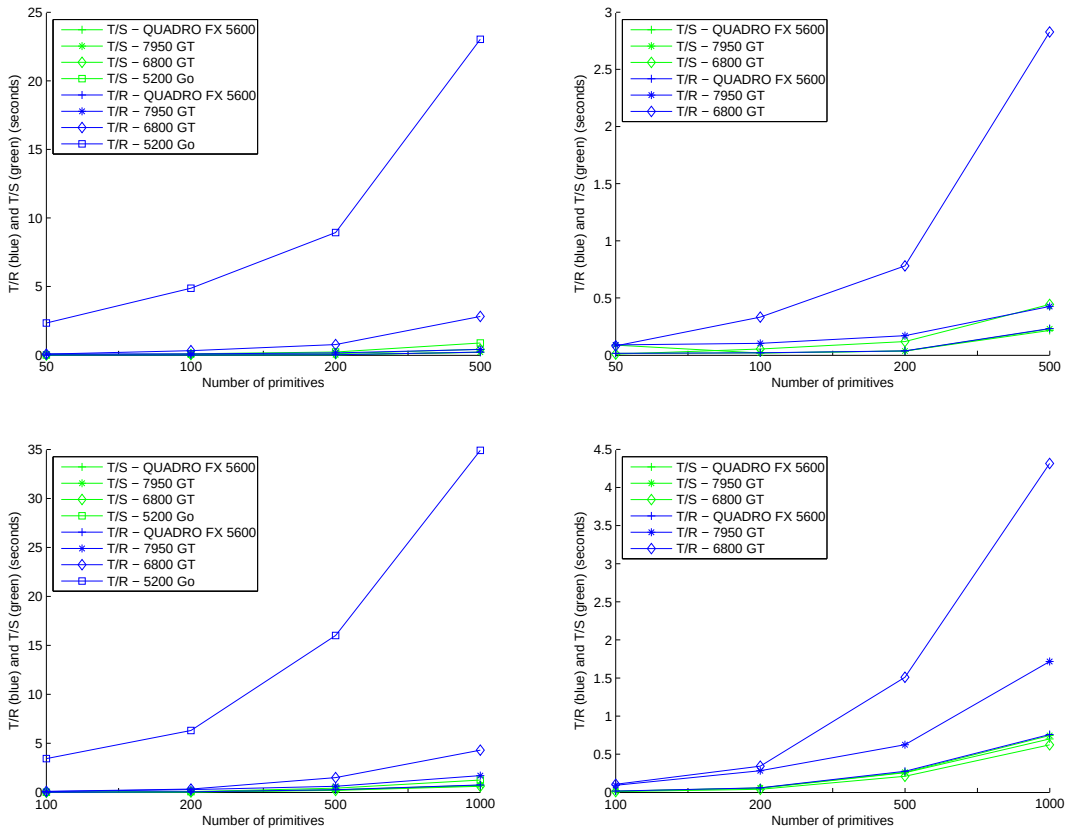


Fig. 15. **Time to traverse and setup (T/S, in green) and traverse and render (T/R, in blue). First row: timings for cylinder subtracted by 50, 100, 200 and 500 cylinders (Fig. 12, second column). Second row: timings for cylinder subtracted by 100, 200, 500 and 1000 cones (Fig. 12, first column). In each row, the first column presents timings for all configurations, while the second column presents timings for the 3 faster configurations with more detail.**

plex models, such as the moderately complex ones (involving up to 500 primitives). This shows up in the T_R timing in both Figs. 14 and 15, as well as in the T_R column of Table 1. The fact that for more complex models a more modest increase in performance is observed can be explained by the fact that these complex models have very large and complex octrees — whose subdivision and traversal process takes significant time. Also, for each cell of the resulting octree, a setup time incurs (the time it takes to pass primitive parameters, etc. to the pixel shader). Thus, for these highly complex models, the rendering process becomes more CPU and bandwidth bound, as evidenced by comparing the set up time T_S and the subdivision time S, as the number of primitives increases in Figs. 14 and 15.

While the increase in T_R performance when using faster GPUs is noticeable and follows the increase in instruction throughput for moderately sized models, we can see from the second column of Fig. 14 that for the configu-

ration with the Quadro FX5600 GPU, the traversal and rendering time (T_R) appears to be limited by the setup time T_S. This limitation is possibly preventing that configuration from performing even better. Hence, our results indicate that for the current state of the art GPUs, the CPU (time to traverse the octree) and the CPU/GPU bandwidth (time to pass the rendering parameters to the GPU), which together comprise T_S, are becoming limiting factors in the increase of performance. However, as we argue in Sect. 5, our method can be continuously adapted to rescale the dependency on GPU throughput vs. GPU/CPU bandwidth and CPU throughput.

Also, note that factors other than the number of primitives clearly influence the performance of our approach, among which are: the number of cells of the object that are actually visible on the screen — the smaller the number of cells inside the visible screen the faster the rendering is, as the cells that are not visible will be clipped and no pixel shader will be executed for them; the area the object occupies on the screen — the larger the area the object occupies on the screen the larger the number of pixels for which pixel shaders will be run, hence the slower the rendering is; the object's 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 slower the rendering is. Isolating the impact of each of these effects is considerably hard as there are usually a combination of these in effect.

4.2 Correctness analysis

We performed a qualitative correctness analysis by comparing renderings of all of our models produced by our approach against those produced by povray, when used with comparable quality settings (i.e., +Q3 — without inter-object shadows or anti-aliasing). In all cases tested, there were no visible discrepancies. In Fig. 16, we expose some of these renderings side by side. Our method is able to produce the same visual quality as povray, and at the same time it does so significantly faster (in some cases by more than one order of magnitude).

5 Conclusions

We have presented a method for rendering CSG objects composed of convex primitives that tries to get the best of both worlds — CPU and GPU — without suffering (at least as much as in previous methods) the curse of memory bandwidth limitations. In our approach, spatial subdivision of the

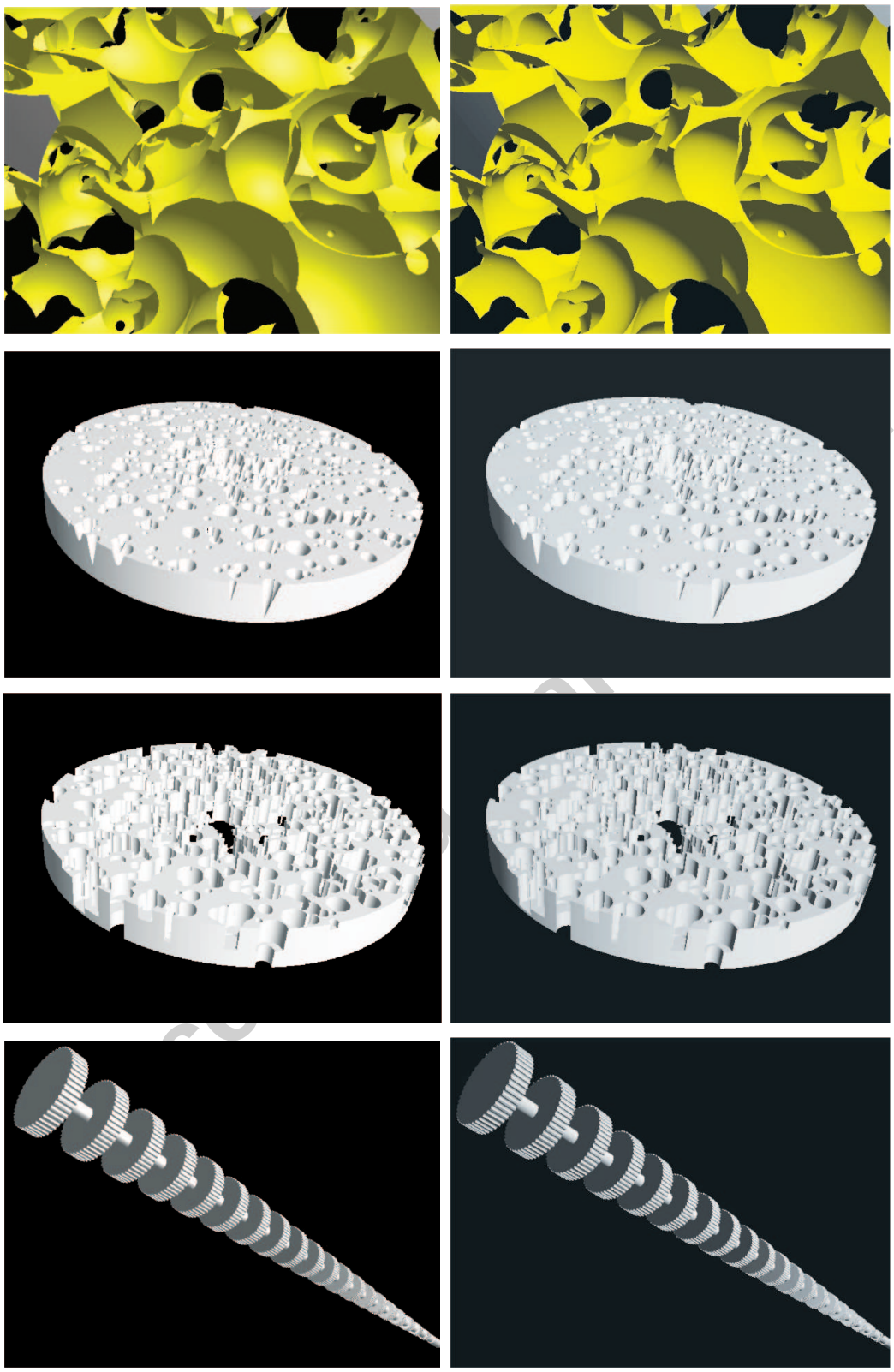


Fig. 16. **First column: CSG models rendered using our approach. Second column: CSG models rendered using povray. Note that while a change in bias exists, there are no noticeable artifacts when comparing our renderings to the povray renderings. Images have been adjusted in brightness and constrast for comparison purposes.**

CSG object is performed on the CPU, while parts of the CSG object that lie inside each cell of the resulting octree are ray-traced in the GPU.

Experimental results have demonstrated that with the latest GPUs state of the art performance is achieved, when comparing to results directly reported in other papers. Results have also shown that our algorithm is dominantly GPU instruction throughput bound for up to moderately complex objects (up to 500 primitives). Moreover, while the process becomes more CPU and bandwidth bound for more complex models, between each of the configurations tested, an improvement more significant than both the respective CPU throughput and the memory bandwidth increase is observed, even for the more complex models. This is a remarkable fact since instruction throughput has essentially doubled with every new generation of GPUs whereas bandwidth improvement has increased at a considerably lower pace over the last few generations of GPUs.

While, as GPUs become more powerful, it might seem that the process seems to be geared towards becoming entirely CPU and bus bandwidth bound (as our results for the Quadro FX 5600 seem to indicate), it will be possible to efficiently implement the ray-tracing of boolean operations of more than two primitives with newer GPUs, potentially even with the Quadro FX 5600. Thus, it will be possible to slightly adapt our method to rescale the dependency on memory bandwidth/CPU instruction throughput vs GPU instruction throughput towards using more GPU instruction throughput. With a continuous improvement (ray-tracing of boolean operations of more and more primitives on the GPU becoming feasible) the corresponding octrees would become less and less complex, consequently requiring less parameter passing to the GPU, decreasing the bandwidth requirements and allowing models with far greater complexity to be rendered at realistic rates as well as allowing the rendering of moderately sized models to remain GPU instruction throughput limited. In short, it is possible to adapt our approach to re-balance the load between the CPU and the GPU. In our view, this is an important direction of future work for our approach. For example, it would be interesting to investigate ways to dynamically change the concept of "simple enough" depending on the CPU/GPU load balance.

As one direction of future work, using more adaptive structures such as kd-trees would allow for better spatial subdivision — since cells could then be divided right on points of order 3 or larger, such as in Fig. 1, decreasing the complexity of the subdivision structures. Parallelizing the spatial subdivision is an easy modification to our method that would also make the subdivision process much more efficient, since CPUs seem to be geared towards having ever more cores (e.g., with such a modification, the subdivision S and subdivision and rendering S_R times with the 2x Quadcore

Xeon 2GHz configuration would be *much* faster).

We also plan to include other geometric primitives into the system, such as half-spaces, height maps, cubics, quartics, and surfaces defined by Bezier patches. The use of linear half-space in CSG would make possible for example the representation of polyhedral surfaces. Height maps are useful for the representation of terrain data, and higher order implicits enable the description of more complex smooth surfaces. There has been a lot of recent work in this direction that can be incorporated into our framework. Some relevant references include [20,21,11].

Inter-object shadows can be incorporated in a two-pass approach, rendering to a shadow-buffer first, from the point of view of the light source. One may also look at performing occlusion queries to exclude occluded cells from the rendering process, and thus reduce the dependency of the rendering times on the depth complexity of the object.

Finally, it would be interesting to compare the performance of implementations of our approach using pixel shaders to a similar one using a more generic framework for GPU programming such as NVIDIA's CUDA, as that framework could provide more efficient ways of passing CSG model parameters to the GPU, further reducing the dependency of our method on bus bandwidth: in our current implementation, everytime the CSG object inside a given cell is raytraced, the full parameters for all the primitives that determine that object (e.g. radii, colors, centers, etc) are passed to the GPU. One way to overcome this limitation would be to assign IDs to all the primitives that compose the CSG object, upload their parameters once onto the GPU's memory and when handling a given cell simply pass the primitives' ids instead of the full parameters. It is worth noticing that recently geometry instancing and bindable uniforms have become available and it is likely possible to implement this functionality using these extensions, but it might be the case that a framework like CUDA provides tools to make this task easier.

References

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

[2] João Luiz Dihl Comba and Ronaldo C Marinho Persiano. Ray Tracing otimizado de sólidos CSG usando octrees. In *Proc. SIBGRAPI'90*, pages 21–30, Gramado, Brazil, May 1990.

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

[4] Jack Goldfeather, Jeff P M Hultquist, and Henry Fuchs. Fast constructive-solid geometry display in the pixel-powers graphics system. In *SIGGRAPH '86: Proceedings of the 13th annual conference on computer graphics and interactive techniques*, 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 Computer Graphics and Applications*, 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 *Proc. Symposium on Interactive 3D Graphics*, 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. 1989.

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

[10] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2001.

[11] Charles Loop and Jim Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25:664-670, 2006.

[12] Ari Rappoport and Steven Spitz. Interactive boolean operations for conceptual design of 3-d solids. In *SIGGRAPH '97: Proceedings of the 24th annual conference on computer graphics and interactive techniques*, pages 269–278, 1997.

[13] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Transactions on Graphics*, 12(4):437–464, 1980.

[14] Jarek Rossignac. Blist: A boolean list formulation of CSG trees. Technical report, January 08 1999.

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

[16] Jarek Rossignac. Optimized Blist Form (OBF). Technical report, GIT-GVU-07-10 May 23 2007.

[17] Nigel Stewart, Geoff Leach, and Sabu John. A CSG rendering algorithm for convex objects. *The 8th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '2000 - WSCG 2000*, II:369–372, February 2000.

[18] Nigel Stewart, Geoff Leach, and Sabu John. Linear-time CSG rendering of intersected convex objects. *The 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '2002 - WSCG 2002*, II:437–444, February 2002.

[19] Rodrigo de Toledo and Bruno Levy. Extending the graphic pipeline with new GPU-accelerated primitives. In *International gOcad Meeting, Nancy, France*, 2004. Also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil.

[20] Rodrigo de Toledo, Bin Wang and Bruno Levy. Geometry Textures. In *Proceedings of SIBGRAPI 2007 - XX Brazilian Symposium on Computer Graphics and Image Processing*, 79-86, 2007.

[21] Rodrigo de Toledo and Bruno Levy and Jean-Claude Paul. Iterative Methods for Visualization of Implicit Surfaces on GPU. *ISVC, International Symposium on Visual Computing*, to appear, 2007.

[22] Ruediger Westermann and Bernd Sevenich. Accelerated Volume Ray-Casting using Texture Mapping. In *IEEE Visualization 2001*, 2001.

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

A Exerpts of Cg Code

In this appendix, we provide Cg source code for vertex and pixel shaders that ray-trace the difference of two spheres. Given the description of our algorithm in the previous sections, modifying these to handle other primitives or other boolean operations should be fairly trivial.

A.1 Common vertex shader (used with all our pixel shaders)

```
vfconn vp_main(appdata IN, uniform float4x4 ModelViewI,
               uniform float4x4 ModelViewProj, uniform float3 light)
{
    vfconn OUT;
    // Output light position - object space
    OUT.l.xyz = light.xyz;
    OUT.l.w = 1.0f;
    // Output camera position - object space
    float3 viewpos = mul(ModelViewI, float4(0,0,0,1)).xyz;
    OUT.v.w = 1.0f;
    OUT.v.xyz = viewpos.xyz;
    // Output position - object space
    OUT.OPos.xyz = IN.position.xyz;
    // Output position - clip space
    OUT.HPos = mul(ModelViewProj, IN.position);
    // Output color
    OUT.Col0.xyzw = IN.color.xyzw;
    return OUT;
}
```

A.2 Pixel shader to ray-trace the difference of two spheres

```
void fp_main(in vfconn IN, out float4 color : COLOR0, out float depth : DEPTH,
             uniform float4x4 ModelViewProj, uniform float radius1,
             uniform float radius2, uniform float3 center1,
             uniform float3 center2, uniform float3 ll,
             uniform float3 ur, uniform float3 color1, uniform float3 color2)
{
    float3 o = IN.OPos.xyz;
    float t;
    half3 norm_l;
    float3 norm_v;
    float3 v = o - IN.v.xyz;
    intsec res;
    float3 obj_color;
    norm_v = normalize(v);
    // Perform ray-cell intersection
    float max_t = find_max_t(o,norm_v,ll,ur);
    // Intersect ray with first sphere
    intsec res1 = ray_intersect_sphere(center1, radius1, o, norm_v, max_t);
    // Intersect ray with second sphere
    intsec res2 = ray_intersect_sphere(center2, radius2, o, norm_v, max_t);
    // Complement second ray-primitive intersection set,
    // since this is a difference operation
    res2.normal1 = -res2.normal1;
    res2.normal2 = -res2.normal2;
    // Perform intersection of ray-primitive result sets res1 and res2
    // Note that this corresponds to case 2 in Sect. 3.3.4
    res = calc_intersection2(res1, res2, max_t);
    if (res.t1 < -EPS)
    {
        // object extends outside the cell (through faces of the
        // cell facing the camera - clip object)
        discard;
    }
    if ((res.t1-max_t)>EPS)
    {
        // object extends outside the cell (through faces of the
        // cell not facing the camera - clip object)
        discard;
    }
    // Calculate Depth
    float4 vtemp;
    vtemp.xyz = o + res.t1 * norm_v;
    vtemp.w = 1.0f;
    float4 depth_tmp = mul(ModelViewProj, vtemp);
    depth = 0.5 + (depth_tmp.z/depth_tmp.w)/2;
    // Perform Shading
    half3 normal = normalize(res.normal);
    norm_l = normalize(IN.l.xyz - vtemp.xyz);
    norm_v = normalize(IN.v.xyz - vtemp.xyz);
    if (res.object==false)
        obj_color=color1;
    else
        obj_color=color2;
    color.xyz = calc_lighting(normal, norm_l, norm_v, obj_color);
    color.w = 1.0f;
}
```

A.3 Intersection of ray-primitive result sets - case 2

```
intsec calc_intersection2(intsec i1, intsec i2, float mt)
```

```
{
    float t1, t2;
    intsec res;
    t1 = INFTY;
    t2 = INFTY;
    if (i1.t1 < i2.t1)
    {
        t1 = i1.t1;
    }
    if (i1.t1 > i2.t2)
    {
        t1 = i1.t1;
    }
        else if (i1.t2 >= i2.t2)
    {
        t2 = i2.t2;
    }
    if ((t1 >= -EPS) && (t1 <= (mt+EPS)))
    {
        res.object=0;
        res.t1 = t1;
        res.normal = i1.normal;
    }
    else
    {
        res.object=1;
        res.t1 = t2;
        res.normal = i2.normal2;
    }
    return res;
}
```

A.4 Ray-Sphere intersection

```
intsec ray_intersect_sphere(float3 center, float radius, float3 o, float3 v)
{
    intsec res;
    res.t1=INFTY;
    res.t2=INFTY;
    o = o - center;
    float oo = dot(o,o);
    float ov = dot(o,v);
    float delta = (ov * ov - oo + radius * radius);
    if (delta >= 0.0f)
    {
        delta = sqrt(delta);
        res.t1 = -ov - delta;
        res.t2 = -ov + delta;
        res.normal = o + (res.t1) * v;
        res.normal2 = o + (res.t2) * v;
    }
    return res;
}
```

A.5 Ray-Cell intersection

```
float find_max_t(float3 o, float3 v, float3 ll, float3 ur)
{
    float3 v_ll=(ll-o)/v;
    float3 v_ur=(ur-o)/v;
```

```
    return min(min(max(v_ll.x,v_ur.x), max(v_ll.y,v_ur.y)), max(v_ll.z,v_ur.z));
}
```

A.6 Auxiliary data structures

```
struct intsec
{
    fixed object;
    float t1;
    float t2;
    half3 normal;
    half3 normal2;
};
struct vfconn
{
    float4 HPos : POSITION;
    float4 OPos : TEXCOORD1;
    float4 Col0 : COLOR0;
    float4 l    : TEXCOORD2;
    float4 v    : TEXCOORD0;
};
struct appdata
{
    float4 position : POSITION;
    float4 color    : COLOR0;
};
```