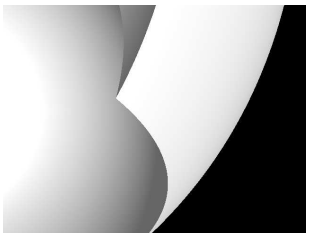
Hardware-Assisted CSG Rendering

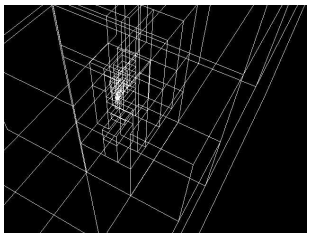
Fabiano Romeiro
Harvard University

Luiz Velho
IMPA

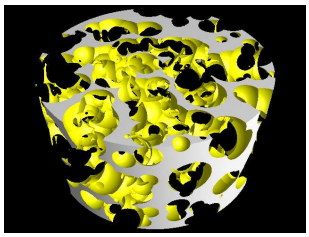
Luiz Henrique de Figueiredo
IMPA



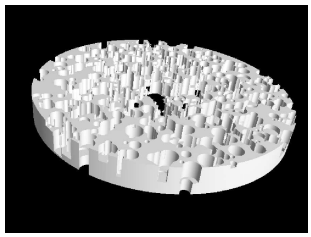
(a) 2 spheres subtracted of a sphere



(b) octree corresponding to (a)



(c) 1 cylinder minus 1000 spheres



(d) 1 cylinder minus 1000 cylinders

One of the most intuitive ways to model solid objects is by constructing them hierarchically, through combinations of simpler objects. Several representations that incorporate this paradigm exist, of which CSG is the most popular. In the CSG representation, solid objects are obtained by successive boolean combinations of primitives, and are represented by the 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.

The CSG paradigm is 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. Since CSG's introduction, several approaches have been devised towards achieving that goal on ever more complex objects. 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, most recently [2]. 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.

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). By performing spatial subdivision of the CSG object on the CPU and local ray-tracing of the CSG object on the GPU, our method attempts to share the load between the CPU and the GPU.

The main insight of our approach is that surfaces of CSG objects can be (mostly) locally represented by single primitives or by boolean operations of two primitives. 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 as in (a). 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, rendering the entire CSG object reduces to:

1. Subdividing it 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.

A modified octree structure is used to perform the subdivision of the CSG object (b). Along with each cell of the octree we keep a CSG tree structure that stores the simplest local representation of the surface of the original CSG object. The subdivision starts with the bounding box of the CSG object as the initial cell of the octree, whose CSG tree corresponds to the entire CSG object. The process proceeds by subdividing the cell, 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 above conditions are met or the cell no longer intersects the CSG objects surface.

Once the octree has been generated, it is traversed recursively in a view-dependent front-to-back manner and as leaf cells are reached the restriction of the cell's CSG tree to its interior is rendered in the GPU. This last step is an extension of [1], to ray-trace not only a set of primitives, but also boolean operations of them.

To ray-trace primitives in the GPU, as introduced in [1], we bind the appropriate vertex and pixel shaders first, and render some object (e.g. a bounding box) whose projection on the screen covers the projection of the intended primitive. 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 and finally performs the appropriate shading. In the case of a boolean operation of primitives, the pixel shader intersects the ray with both primitives, determines the boolean operation of the intersection sets, and performs the shading.

Our results are comparable performancewise with the currently top performing algorithms, and as newer generations of GPUs come out, since our algorithm relies more on instruction throughput power than on bandwidth as the previous ones, as infered by performance analysis on different GPUs, we expect its performance to improve at a greater pace.

References

[1] TOLEDO, R., AND LEVY, B. 2004. Extending the graphic pipeline with new gpu-accelerated primitives. Tech. rep., INRIA.

[2] HABLE, J., AND ROSSIGNAC, J. 2005. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. ACM Trans. Graph. 24, 3, 1024-1031.