Spatially Efficient Tree Layout for GPU Ray-tracing of Constructive Solid Geometry Scenes

D.Y. Ulyanov[1,2], D.K. Bogolepov[2], V.E. Turlapov[1]

University of Nizhniy Novgorod[1], OpenCASCADE[2]

A novel GPU-optimized CSG ray-tracing approach is presented that is fast and accurate, and allows achieving real-time frame rates at full-screen resolutions. It has no limitations on the maximum number of primitives, and produces final image in a single pass. We propose an efficient procedure to transform an input CSG tree into equivalent spatially coherent and well-balanced form. Through various experiments, we show that our solution allows interactive rendering of CSG models consisting of more than a million CSG primitives on consumer graphics cards.

Keywords: Constructive solid geometry, rendering, ray-tracing, GPU, optimization.

1. Introduction

Constructive Solid Geometry (CSG) is the geometric method that forms complicated shapes from simpler 3D primitives using the Boolean operations *union* ($\cup$), *intersection* ($\cap$), and *subtraction* (\). The (sets of) primitives involved in each operation and the sequence of operations create a so-called CSG tree. Thus, CSG tree is a binary tree with leaf nodes as primitives and interior nodes as Boolean operations. CSG is often used as a fundamental modeling approach in CAD/CAM/CAE applications. However, computation of the geometry corresponding to the CSG expression can be a slow process, which is often unacceptable for interactive scene editing. In some cases, a near real-time rendering of a CSG shape can be achieved by using multi-pass image-based techniques taking advantage of hardware depth and stencil buffers. But these algorithms impose limitations on the maximum depth complexity and, in general, are bounded by memory bandwidth. The main contribution of this paper is a novel GPU-optimized CSG ray-tracing algorithm, as well as an efficient procedure for conversion of input CSG tree into spatially coherent and well-balanced form. The proposed solution is relatively fast, can be easily integrated into existing ray-tracing systems and, as we show in our experiments, outperforms previously available algorithms.

2. Previous Work

In general, there are two basic approaches to render a CSG model. The first one is based on precomputing of the boundary of a CSG shape which can be tessellated into a triangular mesh and then rendered using conventional graphics methods. Since evaluation of CSG boundary is computationally expensive, these algorithms are mainly limited to static models and do not allow interactive editing. The second approach involves so-called *image-based* algorithms which generate just the image of a CSG model without expensive computation of the full shape geometry. Most of these algorithms are designed for graphics hardware and based on multi-pass, view-specific techniques making extensive use of depth and stencil buffers. The typical algorithms in this class are *Goldfeather* algorithm [1, 2] and the *Sequenced Convex Subtraction* (SCS) algorithm [3]. The first one allows handling all types of CSG primitives, while the second one is optimized for models consisting of convex primitives only. However, none of these algorithms is capable of rendering arbitrary CSG trees directly. Instead, an input tree is transformed into a *sum-of-products* (*normal*) form that can lead to exponential growth of the number of CSG operations and significantly reduces the performance for complex CSG shapes.

An alternative approach has been proposed in the later work [4]. The so-called *Blister* algorithm does not require a conversion to the sum-of-products form. Instead, it converts an arbitrary Boolean combination of primitives into the *Blist form* [5] containing each input primitive only once. To render a CSG shape, Blister uses peeling technique to produce layers of the entire primitive set in depth order

(each layer is the Z-buffer representation of a 3D scene that allows only one fragment stored at each pixel). Each peel is classified according to its CSG expression and then combined.

The above algorithms can achieve interactivity for relatively complex CSG shapes (thousands of primitives). However, all these techniques use many rendering passes, and thus are bandwidth limited. For many years, GPU memory bandwidth grows slower than computing performance, resulting in a data transfer bottleneck for many GPU-accelerated applications. A completely different approach was adopted in [6]. In this work, an attempt has been made to distribute the workload between a CPU and a GPU, by performing spatial decomposition of input CSG tree on a CPU and ray-tracing of its simple parts on a GPU. The algorithm has proven to be effective for relatively simple CSG shapes (hundreds of primitives). Whereas more complex models require subdivision into a larger number of parts that leads to a huge number of draw calls and performance decrease.

Ray-tracing of the entire CSG tree is possible and used quite widely. However, most approaches to render CSG scenes require computing all intersections of a ray with a primitive. The ray is broken into intervals corresponding to the intersected primitives. After that the Boolean operations are applied to find out the first interval that is actually inside a CSG object. Due to a large amount of computation and significant memory consumption this approach can be extremely expensive. Moreover, it is poorly suited for a GPU, the effective use of which requires tens of thousands of threads running in parallel. Since GPU hardware resources are divided among threads, low resource usage is crucial to support a plurality of simultaneously-active threads. However, the implementation of interval CSG ray-tracer on the GPU is still possible as shown in [7]. Unfortunately, this approach tends to be limited by the number of primitives and maximum depth complexity due to the necessity of storing interval representation of the whole scene in GPU memory.

```
function INTERSECT(node, min)
 min_L ← min
 min_R ← min
 (t_L,N_L) ← INTERSECT(L(node), min_L)
 (t_R,N_R) ← INTERSECT(R(node), min_R)
 state_L ← CLASSIFY(t_L, N_L)
 state_R ← CLASSIFY(t_R, N_R)
 while true do
  actions ← table[state_L, state_R]
  if Miss ∈ actions then
   return miss
  if RetL ∈ actions or (RetLIfCloser ∈ actions and t_L ≤ t_R) then
   return (t_L, N_L)
  if RetR ∈ actions or (RetRIfCloser ∈ actions and t_R ≤ t_L) then
   if FlipR ∈ actions then
    N_R ← −N_R
   return (t_R, N_R)
  else
   if LoopL ∈ actions or (LoopLIfCloser ∈ actions and t_L ≤ t_R) then
    min_L ← t_L
    (t_L, N_L) ← INTERSECT(L(node), min_L)
    state_L ← CLASSIFY(t_L, N_L)
   else
    if LoopR ∈ actions or (LoopRIfCloser ∈ actions and t_R ≤ t_L) then
     min_R ← t_R
     (t_R, N_R) ← INTERSECT(R(node), min_R)
     state_R ← CLASSIFY(t_R, N_R)
    else
     return miss
```

Fig. 1. Recursive CSG intersection

A quite different approach based on *single-hit* ray-tracing (finding only nearest intersection) has been proposed in [8]. The algorithm uses a concept of state machine to calculate the intersection with a CSG model. The only limitation is that the basic CSG primitives should be closed (can be relaxed to handle orientable surfaces), non-self-intersecting and have consistently oriented normals. This elegant idea makes it quite easy to integrate CSG rendering into existing ray-tracing systems. Although the paper does not contain any characteristics of the algorithm, it looks suitable for the GPU and inspired our work. In the remainder of this section, we outline the main steps of this algorithm and point out some inaccuracies in the original state tables.

Let T be a CSG tree, and let $L(T)$ and $R(T)$ be the left and right sub-tree of T. To find the nearest intersection of ray R and tree T the ray is shot at sub-trees $L(T)$ and $R(T)$, and then the intersection with the each sub-tree is classified as one of entering, exiting or missing it. Based upon the combination of

these two classifications, one of several actions is taken: (*a*) returning a hit; (*b*) returning a miss; (*c*) changing the starting point of ray *R* for one of sub-trees and then shooting this ray again, classifying next intersection. In latter case, the state machine enters a new loop (see Figure 1). Kensler proposed 3 state tables (one for each Boolean operation) needed to ray-trace a CSG shape. Unfortunately, these state tables are not complete and lead to incorrect visualization. In this paper we provide refined state tables allowing correct visualization in all cases (see Table 1). Kensler's algorithm is recursive and poorly suited for a GPU. While recursion is supported on CUDA-enabled GPUs, the iterative version with a manually-managed state stack provides a much better performance, and can be implemented in the environment without recursion support (e.g., OpenGL, OpenCL). However, transforming of the algorithm into iterative form is not trivial due to a large number of parameters and local variables.

Table 1. State tables for Boolean operations

∪	Enter *R*(*T*)	Exit *R*(*T*)	Miss *R*(*T*)	∩	Enter *R*(*T*)	Exit *R*(*T*)	Miss *R*(*T*)	\	Enter *R*(*T*)	Exit *R*(*T*)	Miss *R*(*T*)
Enter *L*(*T*)	*RetLIfCloser RetRIfCloser*	*RetRIfCloser LoopL*	*RetL*	Enter *L*(*T*)	*LoopLIfCloser LoopRIfCloser*	*RetLIfCloser LoopR*	*Miss*	Enter *L*(*T*)	*RetLIfCloser LoopR*	*LoopLIfCloser LoopRIfCloser*	*RetL*
Exit *L*(*T*)	*RetLIfCloser LoopR*	*LoopLIfCloser LoopRIfCloser*	*RetL*	Exit *L*(*T*)	*RetRIfCloser LoopL*	*RetLIfCloser RetRIfCloser*	*Miss*	Exit *L*(*T*)	*RetLIfCloser RetRIfCloser FlipR*	*RetRIfCloser FlipR LoopL*	*RetL*
Miss *L*(*T*)	*RetR*	*RetR*	*Miss*	Miss *L*(*T*)	*Miss*	*Miss*	*Miss*	Miss *L*(*T*)	*Miss*	*Miss*	*Miss*

2. GPU-Optimized CSG Ray-tracing

2.1 Stack-based CSG traverse

As our main contribution, we propose the iterative CSG ray-tracing algorithm that uses minimal state and is optimized for massively parallel architectures with limited (per thread) memory resources like GPUs. For that purpose we define a high-level state machine that manages the execution of the original algorithm in the iterative manner (see Figure 2).

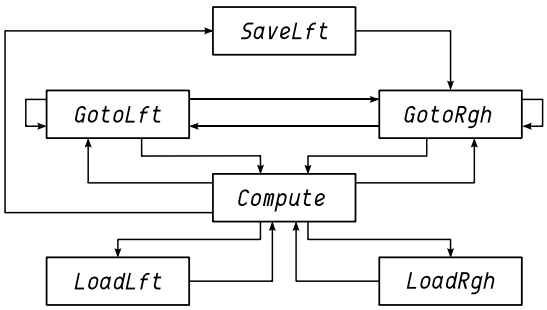

Fig. 2. High-level state machine

The use of state tables for each Boolean operation (let us call them CSG tables) is based on pre-computed intersections with child objects of the current CSG node. Thus, all of the states of high-level pushdown automata are divided into two categories: (*a*) calculation of intersections with child objects, and (*b*) applying CSG tables for classification of the processed CSG node. The first category includes the states `GotoLft` (finding the intersection with the left sub-tree), `GotoRgh` (finding the intersection with the right sub-tree), and `SaveLft` (storing the intersection parameters with the left sub-tree and then execution of `GotoRgh`). The last state is needed since the processing of the right sub-tree leads to trashing local variables. The second category includes the following states: `Compute` (applying CSG tables), `LoadLft` (loading intersection data for the left sub-tree and then execution of `Compute`), `LoadRgh` (loading intersection data for the right sub-tree and then execution of `Compute`). General scheme of transition between the states is shown in Figure 3. Here the `GoTo()` function (see Figure 4) calculates intersection points with left and right sub-trees, while the `Compute()` function (see Figure 4) classifies these points in order to detect the first intersection of a ray with the actual boundary of CSG shape. Note that `GoTo()` function enables the use of bounding boxes to improve the performance

of intersection function. Such bounds are calculated for each node of CSG tree to obtain a bounding volume hierarchy [9].

```
t_min ← 0
node ← V // virtual root whose left subtree is the real root
(t_L, N_L) ← invalid
(t_R, N_R) ← invalid
PUSHACTION(Compute)
action ← GotoLft
while true do
 if action ≡ SaveLft then
  t_min ← POPTIME()
  PUSHPRIMITIVE(t_L, N_L)
  action ← GotoRgh
 if action ∈ {GotoLft, GotoRgh} then
  GOTO()
 if action ∈ {LoadLft, LoadRgh, Compute} then
  COMPUTE()
```

Fig. 3. Iterative CSG traversal

```
function GOTO()
 if action ≡ GotoLft then
  node ← L(node)
 else
  node ← R(node)
 if node is Operation then
  gotoL ← INTERSECTBOX(L(node))
  gotoR ← INTERSECTBOX(R(node))
 if gotoL and L(node) is Primitive then
  (t_L, N_L) ← INTERSECT(L(node), t_min)
  gotoL ← false
 if gotoR and R(node) is Primitive then
  (t_R, N_R) ← INTERSECT(R(node), t_min)
  gotoR ← false
 if gotoL or gotoR then
  if gotoL then
   PUSHPRIMITIVE(L(node), t_L)
   PUSHACTION(LoadLft)
  else if gotoR then
   PUSHPRIMITIVE(R(node), t_R)
   PUSHACTION(LoadRgh)
  else
   PUSHTIME(t_min)
   PUSHACTION(LoadLft)
   PUSHACTION(SaveLft)
  if gotoL then
   action ← GotoLft
  else
   action ← GotoRgh
 else
  action ← Compute
else
 // node is a Primitive
 if action ≡ GotoLft then
  (t_L, N_L) = Intersect(node, t_min)
 else
  (t_R, N_R) = Intersect(node, t_min)
 action ← Compute
 GOTOPARENT(node)
```

```
function COMPUTE()
 if action ∈ {LoadLft, LoadRgh} then
  if action ≡ LoadLft then
   (t_L, N_L) ← POPPRIMITIVE()
  else
   (t_R, N_R) ← POPPRIMITIVE()
 state_L ← CLASSIFY(t_L, N_L)
 state_R ← CLASSIFY(t_R, N_R)
 actions ← table[state_L, state_R]
 if RetL ∈ actions or
  (RetLIfCloser ∈ actions and t_L ≤ t_R) then
   (t_R, N_R) ← (t_L, N_L)
   action ← POPACTION()
   GOTOPARENT(node)
 if RetR ∈ actions or
  (RetRIfCloser ∈ actions and t_R < t_L) then
   if FlipNormR ∈ actions then
    N_R ← −N_R
   (t_L, N_L) ← (t_R, N_R)
   action ← POPACTION()
   GOTOPARENT(node)
 else if LoopL ∈ actions or
  (LoopLIfCloser ∈ actions and t_L ≤ t_R) then
  t_min ← t_L
  PUSHPRIMITIVE(t_R, N_R)
  PUSHACTION(LoadRgh)
  action ← GotoLft
 else if LoopR ∈ actions or
  (LoopRIfCloser ∈ actions and t_R < t_L) then
  t_min ← t_R
  PUSHPRIMITIVE(t_L, N_L)
  PUSHACTION(LoadLft)
  action ← GotoRgh
 else
  t_R ← invalid
  action ← POPACTION()
```

Fig. 4. GOTO stage (left) and COMPUTE stage (right)

2.3 Optimizing CSG Trees

It is obvious that the performance of our algorithm greatly depends on the topology of CSG tree that affects spatial coherence of primitives and height of the tree. However, the creation of a balanced, unbalanced, or a perfect CSG tree depends generally on the user. Thus, it is necessary to transform an input tree T into an equivalent well-balanced tree T' of roughly the same size as T.

We propose an efficient pipeline for optimizing CSG trees that runs in four phases: (*a*) converting the input tree T to a *positive* form; (*b*) spatial optimization of tree topology; (*c*) minimizing height of the tree; (*d*) reverse converting to a *general* form giving the output tree T'.

2.3.1 Converting to positive form

A CSG tree T is represented in the positive form using only $\cup$ and $\cap$ operations and negation of leaf nodes. This conversion can be easily done using the following transformations:

$$\overline{x \cup y} = \overline{x} \cap \overline{y}, \quad \overline{x \cap y} = \overline{x} \cup \overline{y}, \quad x - y = x \cap \overline{y}$$

The above transformations are applied to the tree in a pre-order traversal, and thus all complements are propagated to the leaf nodes. The reverse conversion to general form can be performed using a post-order traversal (in this case all negations are first removed from the children of each node).

2.3.2 Spatial optimization

For optimal performance, the tightness bounds of CSG tree nodes should be used which minimize the probability of ray intersection. For this purpose, we propose the spatial optimization procedure allowing minimizing the bounds of CSG nodes. Let us define *treelet* as the collection of immediate descendants of the given CSG tree node. Our optimization procedure is based on repeatedly selecting of treelets consisting of nodes with the same Boolean operation and their subsequent restructuring (in positive form, we are free to change the order of treelet nodes). Treelets are constructed during a pre-order traversal of CSG tree by expanding child nodes that have the same Boolean operation as the treelet root. The resulting treelet is reorganized by means of surface area heuristic (SAH), widely used for construction of accelerating structures such as *k*-d tree or Bounding Volume Hierarchy (BVH). Thereafter, the traversal of CSG tree continues with the outer treelet nodes.

The restructuring of treelet is based on the same binned technique as is used for construction of BVH [10]. Binned BVH is constructed over all treelet leaves bounded by axis-aligned boxes pre-computed for the input tree T given in general form. Because of this, for treelet leaves corresponding to negative CSG primitives we use their original (non-complemented) bounds. This strategy produces slightly better trees, because infinite bounding boxes do not provide any useful information related to primitive positions.

2.3.3 Minimizing tree height

Our algorithm evaluates intersection point in iterative manner by maintaining a stack. However, on massively parallel architectures like GPUs managing full per-ray stacks leads to significant storage and bandwidth costs. To reduce the traversal state size we desire a *well-balanced* CSG tree. Our next optimization stage is aimed to address this problem by minimizing the height of CSG tree using *local* transformations. At this stage, two types of treelets are considered. For brevity, let us call the child node with a greater height (in the whole tree T) the *heavy* child. The first type is formed of treelets which have the same Boolean operation ($\cup$ or $\cap$) in root node N_1 and its heavy child N_2 (see Figure 5*a*). Let T_3 be a heavy child of the node N_2. Obviously if $h(T_3) > h(T_1) + 1$ it is beneficial to transpose these subtrees. As with the rotations for binary search trees these result in elevating subtree T_3 and demoting subtree T_1. Thus, the height of the treelet, rooted at N_1, is decreased by one.

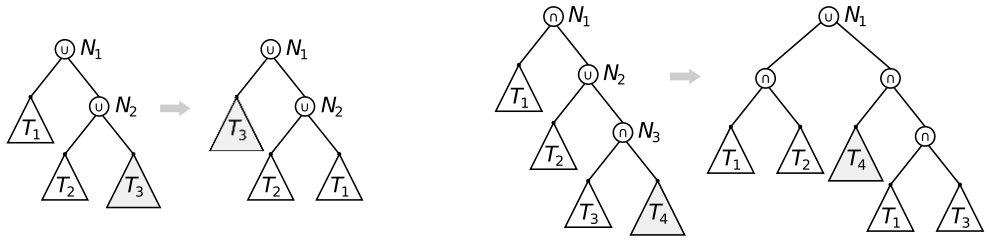

Fig. 5. Optimizations of first (*left*) and second (*right*) type

The second type of treelets corresponds to the case where the operations in the root node N_1, its heavy child N_2 and heavy grandchild N_3 are interleaved (i.e. $\cup$–$\cap$–$\cup$ or $\cap$–$\cup$–$\cap$). Let us consider the $\cup$–$\cap$–$\cup$ sequence (see Figure 5*b*). In this case, the treelet rooted at N_1 can be described by expression: $T_1 \cap (T_2 \cup T_3 \cup T_4) = (T_1 \cap T_2) \cup (T_1 \cap T_3 \cap T_4)$. Let T_4 be a heavy child of the node N_3. Therefore, if $h(T_4) > h(T_1) + 2$, then the normalization of the given treelet allows reducing its height by one. Please note that this normalization is localized, and thus has no effect on other tree nodes. However, even this optimization is undesirable because it results in duplication of the subtree T_1. For this reason we use such transformations only when optimizations of the first type have been exhausted. We use the multi-pass scheme, where at each pass a CSG tree is traversed in post-order, and appropriate restructuring patterns are applied.

4. Results and Discussion

For this study, all results have been measured using an NVIDIA GeForce GTX 680, AMD Radeon HD 7870 and Intel HD 4000 GPUs. All timings correspond to rendering in a 1280 × 720 window. The first test scene shows a CSG model of the city at different scales (see Figure 6). In all below cases the whole City scene is modeled as a single CSG tree. In a simple configuration (*a*), the model contains 3385 primitives. More complex configurations (*b* and *c*) contain 343K and 987K CSG primitives correspondingly. Scene *b* from upper row shows the case with extreme number of depth layers that is rather challenging for other approaches. Therefore, this test allows analyzing the performance depending on the complexity of the CSG model. For each GPU results are represented by two columns (see Table 2): left one corresponds to measured FPS without spatial optimization (−), and the right one was obtained with enabled spatial optimization (+). N/A markers shown were performance clearly cannot be considered to be interactive.

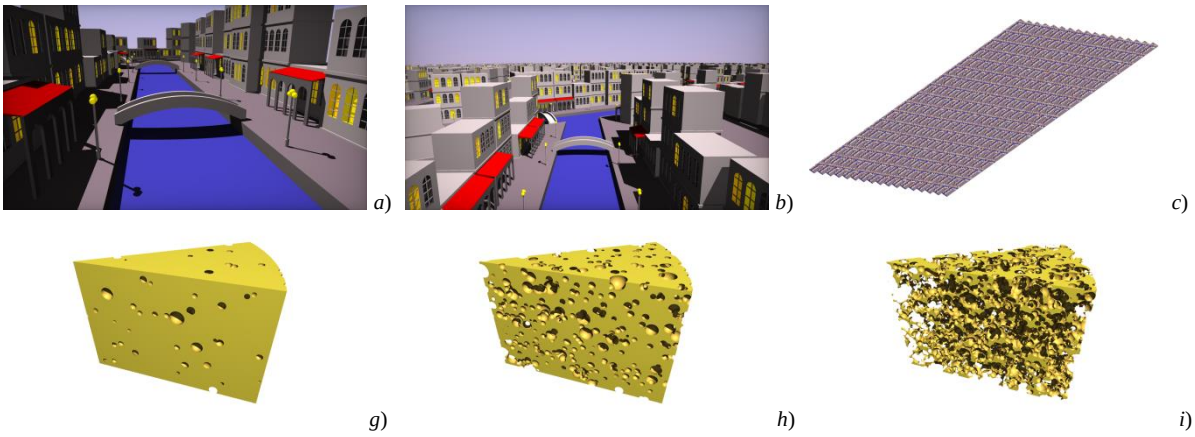

Fig. 6. City scene and Cheese scene

The second test scene represents a procedural Swiss cheese CSG model with the holes of varying radius (see Figure 6). Number of holes increases from 1000 (left) to 8000 (middle), and then to 32000 (right) resulting in a larger number of overlapped primitives and greater depth complexity. Thus, unlike the City model, the performance of the Swiss cheese model is affected greatly by spatial optimization.

The third test scene demonstrates a large number of satellites orbiting a planet (see Figure 8). In this case, each satellite is represented by a separate CSG tree. A plurality of independent satellites are placed into the scene as outer nodes of high-level BVH. Since we are able to interactively rebuild accelerating structure each frame (at least for tens of thousands of objects), we can arbitrarily modify transformations of particular CSG trees. As a result, it becomes possible to edit the scene or to animate arbitrary shapes. In our test case, the satellites move across the planet along randomly selected orbital tracks.

Table 2. Measured performance

Scene	Primitives	Tree Depth	Intel 4000		Radeon HD 7870		GeForce GTX 680	
			–	+	–	+	–	+
City (a)	3385	14	7	7.5	50	60	51	57
City (b)	343589	22	1.8	4.5	6.5	17	8	22
City (c)	987218	24	2.3	7	6.7	18	8.3	21
Cheese (a)	1002	11	0.4	17	4.6	110	5.8	128
Cheese (b)	8002	14	N/A	6.5	0.5	28	0.5	32
Cheese (c)	32002	17	N/A	0.5	N/A	3.7	N/A	4
Satellites (a)	87565	7	5	9	26	67	29	65
Satellites (b)	1120065	7	2.8	4.5	8	18	7	15
Satellites (c)	1120065	7	2.5	4.5	4.2	9	5.6	12

We found that our implementation scales well with increasing the GPU clock speed (Figure 7 shows linear dependence on clock speed). Therefore, we can expect further performance increase on later generations of GPUs. In contrast, the memory clock does not affect performance, which confirms the assumption that the algorithm is not memory-bound.

From practical point of view, there are several factors which can affect the rendering performance. The first one is *screen resolution* as for over ray-tracing methods. The frame rate decreases almost linearly increasing the total number of processed pixels. The second important factor is the *number of primitives*, but however, it does not affect performance directly. Experiments show that we can easily render City scene containing more than 1 million CSG primitives while having trouble with 32K Swiss cheese model. This is due to extensive overlaps between the primitives in cheese model which force the algorithm to iterate over the CSG sub-trees intensively in order to classify intersection points. Moreover, the efficiency of spatial optimizer also suffers from a large number of overlapped primitives. However, even in this stress scenario, we can show near linear performance degradation depending on the number of primitives.

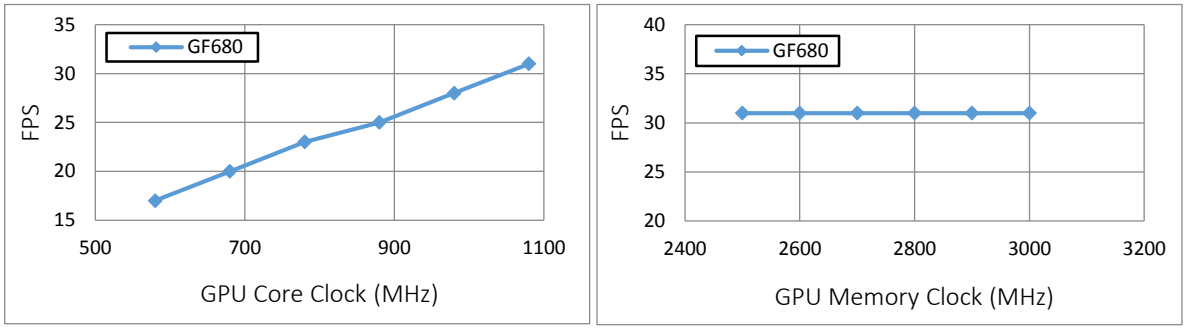

Fig. 7. Rendering performance on Cheese 8K scene depending on GPU clock speed (GF GTX 680).

Given that our solution is based on ray-tracing, it can be naturally extended to produce various visual effects such as transparency, shadows, reflections, refractions, etc. Using OpenGL/GLSL as the main API for GPU computations allows seamless interoperability between CSG rendering engine and standard OpenGL pipeline. For that purpose we calculate the depth value for each processed fragment, based on the intersection time and camera projection matrix. For example, the rendering can be extended with text annotations, axes, or generic triangulated objects drawn by OpenGL. Finally, our solution allows implementing the hardware-accelerated selection mechanism. To this end, we write unique IDs of intersected objects into a separate texture allowing to identify a CSG primitive, or even its particular face, which lies under the given pixel.

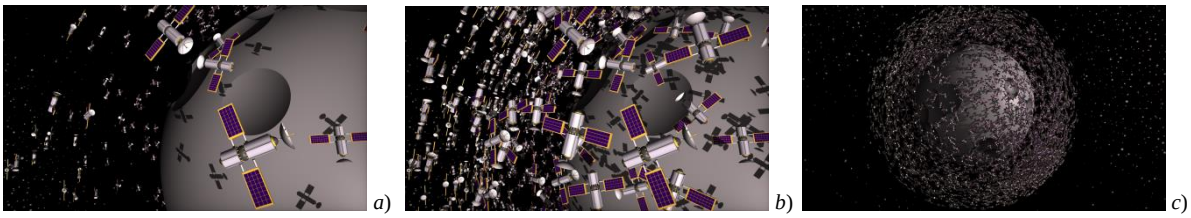
a) *b)* *c)*

Fig. 8. Satellites scene

4. Conclusion

We proposed a GPU-optimized CSG rendering approach, which is fast and accurate, and allows achieving real-time frame rates at full-screen resolutions. Unlike alternative image-based CSG algorithms, our solution is more compute-bound than bandwidth-bound, and does not impose restrictions on the maximum number of CSG primitives being limited only by available GPU memory. We also proposed the efficient pre-processing stage to convert an input CSG tree into equivalent spatially coherent and well-balanced form. As a result, our CSG rendering system provides interactive or even real-time performance for CSG models consisting of more than a million CSG primitives on consumer graphics cards.

References

1. Goldfeather, J., Monar, S., Turk, G., Fuchs, H. Near real-time CSG rendering using tree normalization and geometric pruning // IEEE Symposium on Computer Graphics and Applications. 1989. P. 20-28.
2. Kirsch, F., Döllner, J. Rendering techniques for hardware-accelerated image-based CSG // Journal of WSCG. 2004. Vol. 12, No. 1-3, P. 269-276.
3. Stewart, N., Leach, G., Sabu J. Linear-time CSG rendering of intersected convex objects // Journal of WSCG. 2002. Vol. 10, No. 1-2, P. 437-444.
4. Hable, J., Rossignac, J. Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes // ACM Transactions on Graphics. 2005. Vol. 24, No. 3, P. 1024-1031.
5. Rossignac, J. BLIST: A Boolean list formulation of CSG trees // Technical Report GIT-GVU-99-04 available from the GVU Center at Georgia Tech. http://www.cc.gatech.edu/gvu/reports/1999/ 1999.
6. Romeiro, F., Velho, L., De Figueiredo L. H. Hardware-assisted rendering of csg models // SIBGRAPI'06. 19th Brazilian Symposium. 2006. P. 139-146.
7. Lefebvre, S., Grand-Est, L. I. N. IceSL: A GPU accelerated CSG modeller and slicer // AEFA'13, 18th European Forum on Additive Manufacturing. 2013.
8. Kensler A. Ray tracing CSG objects using single hit intersections URL: http://xrt.wdfiles.com/local--files/doc%3Acsg/CSG.pdf.
9. Cameron, S. Efficient bounds in constructive solid geometry // IEEE Computer Graphics and Applications. 1991. P. 68-74.
10. Wald, I. On fast construction of SAH-based bounding volume hierarchies // IEEE Symposium on Interactive Ray Tracing. 2007. P. 33-40).