# GPU-optimized Ray-tracing for Constructive Solid Geometry Scenes

Denis Bogolepov[1], Danila Ulyanov[1,2], Vadim Turlapov[2]
[1]OpenCASCADE SAS, [2]University of Nizhniy Novgorod
{denis.bogolepov, danila.ulyanov}@opencascade.com, vadim.turlapov@cs.vmk.unn.ru

## Abstract

A novel GPU-optimized CSG ray tracing approach is proposed that is fast and accurate and achieves real-time frame rates for complex CSG models. The algorithm is suitable for primitives defined by tessellation either analytically, has no limitations on the number of CSG primitives and produces the image in single pass. We also propose two-pass procedure for transforming the input tree into spatially coherent and well-balanced form. With these optimizations, the algorithm becomes compute-bound and scales well with additional GPU power (in contrast to multi-pass CSG algorithms). Through various experiments, we show that our solution allows interactive rendering of scenes with more than a million CSG primitives on consumer graphics cards, and as far as we know, this is the fastest general CSG algorithm.

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

## 1. INTRODUCTION

*Constructive Solid Geometry* (CSG) is a technique for combining simple 3D primitives to create a new complicated object using the Boolean operations *union*, *intersection*, and *subtraction*. The (sets of) 3D 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 pivotal modeling approach in CAD/CAM/CAE applications. CSG representations are concise, always valid (define a solid object or the empty set), and easily parameterized and edited. Finally, CSG provides high geometry accuracy and often has no alternatives in such areas as physical simulations (e.g., Monte Carlo transport for electrons, photons, protons and ions). However, actual computation of the geometry resulting from a CSG expression can be a slow process, which is often unacceptable for interactive scene editing.

The main contribution of this paper is the novel GPU-optimized CSG ray tracing algorithm, as well as the efficient procedure for transforming an input CSG tree into spatially coherent and well-balanced form. Our solution achieves high frame rates, can be easily integrated into existing ray tracing engines and, as we show in our experiments, outperforms previously available approaches.

## 2. PREVIOUS WORK

In general, there are two basic approaches to render a CSG model. The first one is based on pre-computing of the boundary of a CSG shape which can be tessellated and rendered using conventional graphics methods. Because evaluation of boundary is extremely expensive, these algorithms are mainly limited to static models and do not allow interactive scene editing. The second approach involves so-called image-based algorithms which generate just an image of CSG model without computation of full 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. Here, the widely used algorithms are *Goldfeather* algorithm [1, 2] and *Sequenced Convex Subtraction* (SCS) [3]. The first one handles arbitrary CSG primitives, while the second one supports convex primitives only. However, none of these algorithms is capable of rendering CSG shapes directly. Instead, an input CSG tree should be transformed into a *sum-of-products* (or *normal*) form that can lead to exponential growth of CSG operations and limits scalability and performance.

An alternative approach has been proposed in the later work [4]. The so-called *Blister* algorithm does not require conversion to the sum-of-products form. Instead, an input Boolean combination is converted into the *Blist form* containing each primitive only once. To render a CSG shape, Blister uses peeling technique to produce layers of the *entire* primitive set in depth order. Each peel is then classified according to its CSG expression and then combined.

The above algorithms can achieve interactivity for CSG shapes of medium complexity (thousands of primitives). However, all these techniques use many rendering passes, and hence are bandwidth limited. For many years, GPU memory bandwidth grows slower than compute performance, resulting in a data transfer bottleneck for many GPU accelerated applications. A completely different approach was adopted in [5]. Here, an attempt has been made to distribute calculations between a CPU and a GPU by performing decomposition of input CSG tree on a CPU and ray tracing of its simple parts on a GPU. This approach has proven to be effective for simple CSG models (hundreds of primitives). Whereas more complex shapes require subdivision into a larger number of parts which leads to a huge number of draw calls and low performance.

Ray tracing of entire CSG tree is possible and used quite widely. 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 the CSG object. Since each ray must be intersected with all primitives, this approach can be extremely expensive. Moreover, it is poorly suited for a GPU, because it is impossible to store a huge number of intervals for thousands of rays simultaneously. However, the implementation of interval CSG ray tracer on a GPU is still possible as shown in [6]. Anyway, this approach tends to be limited by the number of primitives and maximum depth complexity.

A quite different approach based on *single-hit* ray tracing (finding only nearest intersection) has been proposed in [7]. 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 practical experiments, it seems to be a good basis for a GPU algorithm and inspired our work. In the remainder of this section, we outline the main steps of Kensler's algorithm and point out some inaccuracies in the original state tables.

*26-я Международная конференция (GraphiCon2016), Россия, Нижний Новгород, 19–23 сентября 2016 г. 26th International Conference (GraphiCon2016), Russia, Nizhny Novgorod, September 19–23, 2016*

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 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.

**Table 1:** State tables for Boolean operations.

| ∪ | *Enter R(T)* | *Exit R(T)* | *Miss R(T)* |
|---|---|---|---|
| *Enter L(T)* | *RetLIfCloser*<br>*RetRIfCloser* | *RetRIfCloser*<br>*LoopL* | *RetL* |
| *Exit L(T)* | *RetLIfCloser*<br>*LoopR* | *LoopLIfCloser*<br>*LoopRIfCloser* | *RetL* |
| *Miss L(T)* | *RetR* | *RetR* | *Miss* |
| ∩ | *Enter R(T)* | *Exit R(T)* | *Miss R(T)* |
| *Enter L(T)* | *LoopLIfCloser*<br>*LoopRIfCloser* | *RetLIfCloser*<br>*LoopR* | *Miss* |
| *Exit L(T)* | *RetRIfCloser*<br>*LoopL* | *RetLIfCloser*<br>*RetRIfCloser* | *Miss* |
| *Miss L(T)* | *Miss* | *Miss* | *Miss* |
| \ | *Enter R(T)* | *Exit R(T)* | *Miss R(T)* |
| *Enter L(T)* | *RetLIfCloser*<br>*LoopR* | *LoopLIfCloser*<br>*LoopRIfCloser* | *RetL* |
| *Exit L(T)* | *RetLIfCloser*<br>*RetRIfCloser*<br>*FlipNormR* | *RetRIfCloser*<br>*FlipNormR*<br>*LoopL* | *RetL* |
| *Miss L(T)* | *Miss* | *Miss* | *Miss* |

Kensler proposed 3 state tables (one for each Boolean operation) needed to ray trace a CSG object. Here, we provide refined state tables allowing correct visualization in all cases (see Table 1). The pseudo code of this algorithm can be written as follow:

```
function INTERSECT(node, min)  { CSG node to traverse and ray offset }
 min_L ← min
 min_R ← min
 (t_L,N_L) ← INTERSECT(L(node), min_L)    { intersect ray with sub-trees }
 (t_R,N_R) ← INTERSECT(R(node), min_R)
 hit_L ← CLASSIFYHIT(t_L, N_L)            { classify intersection points }
 hit_R ← CLASSIFYHIT(t_R, N_R)            { hit types: enter, exit, miss }
 while true do
  actions ← StateTable[hit_L, hit_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 FlipNormR ∈ 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)
    hit_L ← CLASSIFYHIT(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)
     hit_R ← CLASSIFYHIT(t_R, N_R)
    else
     return miss
```

**Figure 1:** Recursive CSG intersection.

For more details, please refer to original paper [7]. The Kensler's algorithm is very poorly suited for a GPU, because it is recursive and requires too large stack frame. However, its conversion to an iterative form is not a trivial task demanding the identification of general patterns and relationships in the whole set of execution paths.

## 3. GPU-OPTIMIZED CSG RAY TRACING

As the main contribution, we propose an iterative version of CSG ray tracing algorithm that uses minimal state and is optimized for massively parallel architectures with limited per thread resources. For that purpose, we define a high-level state machine managing the execution of initial algorithm in an iterative way (Figure 2).

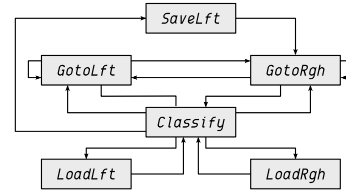


**Figure 2:** High-level pushdown automata.

The use of state tables for each Boolean operation is based on pre-computed intersections with children of current CSG tree node. Thus, all the states of high-level pushdown automata are divided into two classes: (*a*) finding intersections with the child objects, and (*b*) applying state tables for classification of hit points found. The first class includes the states `GotoLft` (find intersection with the left child), `GotoRgh` (find intersection with the right child), and `SaveLft` (store intersection parameters with the left child and then execute `GotoRgh`). The last state is needed because the processing of the right sub-tree overwrites intersection parameters for the left sub-tree, and thus they should be stored for later use. The second class includes the following states: `Classify` (apply state tables), `LoadLft` (load intersection parameters for the left sub-tree and then execute `Classify`), `LoadRgh` (load intersection parameters for the right sub-tree and then execute `Classify`). The pseudocode of the transition between high-level states is shown in Figure 3.

```
t_start ← 0
node ← V   { virtual root containing actual root at the left child }
(t_L, N_L) ← invalid
(t_R, N_R) ← invalid
PUSHSTATE(Classify)
state ← GotoLft   { current state of high-level pushdown automata }
while true do
 if state ≡ SaveLft then
  PUSHHIT(t_L, N_L)
  t_start ← POPTIME()
  state ← GotoRgh
 if state ∈ {GotoLft, GotoRgh} then
  GOTO(t_start)
 if state ∈ {LoadLft, LoadRgh, Classify} then
  CLASSIFY()
```

**Figure 3:** Iterative CSG traversal.

Instead of direct handling of primitive normals we store just the indices of CSG primitives ($N_L$ and $N_R$ variables). This modification decreases the size of stack frame and provides more information at the algorithm output (e.g. primitive indices can be used to access materials). The `GoTo()` function (Figure 4) calculates intersection points with left and right sub-trees, while the `CLASSIFY()` function (Figure 5) performs classification of the points found in order to detect the first hit with the actual CSG boundary. Note, that `GoTo()` function enables the use of bounding boxes of CSG tree nodes to improve the performance of intersection subroutine [8].

```
function GOTO(t_start)
 if state ≡ GotoLft then
  node ← L(node)
 else
  node ← R(node)
```

```
if node is Operation then            { node is Boolean operation }
 traverseL ← IntersectBox(L(node))
 traverseR ← IntersectBox(R(node))
 if traverseL and L(node) is Primitive then { L is CSG primitive }
  (t_L, N_L) ← Intersect(L(node), t_start)
  traverseL ← FALSE
 if traverseR and R(node) is Primitive then { R is CSG primitive }
  (t_R, N_R) ← Intersect(R(node), t_start)
  traverseR ← FALSE
 if traverseL or traverseR then    { traverse at least one child }
  if !traverseL then
   PushHit(t_L, L(node))
   PushState(LoadLft)
  else if !traverseR then
   PushHit(t_R, R(node))
   PushState(LoadRgh)
  else
   PushTime(t_start)
   PushState(LoadLft)
   PushState(SaveLft)
  if traverseL then
   state ← GotoLft
  else
   state ← GotoRgh
 else
  state ← Classify
else                                 { node is CSG primitive }
 if state ≡ GotoLft then
  (t_L, N_L) = Intersect(node, t_start)
 else
  (t_R, N_R) = Intersect(node, t_start)
 state ← Compute
 GoToParent(node)
```

**Figure 4:** *GoTo*() function to find intersections with sub-trees.

```
function Classify()
 if state ∈ {LoadLft, LoadRgh} then
  if state ≡ LoadLft then
   (t_L, N_L) ← PopHit()
  else
   (t_R, N_R) ← PopHit()
 hit_L ← ClassifyHit(t_L, N_L)
 hit_R ← ClassifyHit(t_R, N_R)
 actions ← StateTable[hit_L, hit_R]
 if RetL ∈ actions or
  (RetLIfCloser ∈ actions and t_L ≤ t_R) then
   (t_R, N_R) ← (t_L, N_L)
   state ← PopState()
   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)
   state ← PopState()
   GoToParent(node)
 else if LoopL ∈ actions or
  (LoopLIfCloser ∈ actions and t_L ≤ t_R) then
  t_start ← t_L
  PushHit(t_R, N_R)
  PushState(LoadRgh)
  state ← GotoLft
 else if LoopR ∈ actions or
  (LoopRIfCloser ∈ actions and t_R < t_L) then
  t_start ← t_R
  PushHit(t_L, N_L)
  PushState(LoadLft)
  state ← GotoRgh
 else
  t_R ← invalid
  state ← PopState()
```

**Figure 5:** *Classify*() function to classify intersections found.

In our GLSL implementation, we use only two stacks for storing the algorithm state. The first one is used for intersection times and primitives indices (PushTime and PushHit functions), while the second one is used for states (PushState function).

## 4. OPTIMIZING CSG TREES

The rendering performance of our algorithm greatly depends on the topology of input CSG tree. Unfortunately, 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 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 $T'$.

### 4.1 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 by applying the following transformations in a pre-order traversal:

$$\overline{x \cup y} = \bar{x} \cap \bar{y}\ ,\ \ \overline{x \cap y} = \bar{x} \cup \bar{y}\ ,\ \ x \setminus y = x \cap \bar{y}$$

### 4.2 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's sub-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 widely used for building accelerating structures, such as *k*-d tree or Bounding Volume Hierarchy (BVH). Thereafter, the traversal of CSG tree continues with the outer treelet's nodes.

The restructuring of each extracted treelet is based on the same binned technique as is used for construction of BVH [9]. Binned BVH is built over all treelet leaves bounded by axis-aligned boxes pre-computed for the input tree $T$ given in *general* form (without negations).

### 4.3 Minimizing tree height

To reduce the traversal stack 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.

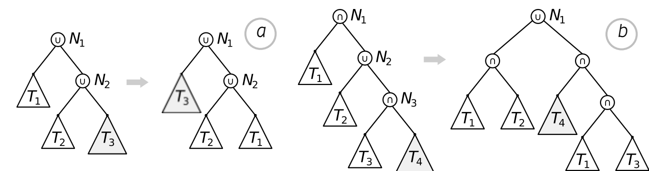


**Figure 6:** Optimizations of first (*a*) and second (*b*) type.

The first transformation is applied to treelets which have the same Boolean operation ($\cup$ or $\cap$) in root node $N_1$ and its heavy child $N_2$

(see Figure 6*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.

The second transformation is applied when the operations in the root node $N_1$, its heavy child $N_2$ and heavy grandchild $N_3$ are interleaved ("∪−∩−∪" or "∩−∪−∩"). Let us consider the case of ∪−∩−∪ sequence (see Figure 6*b*). 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 treelet $N_1$ allows reducing its height by 1. However, this operation also results in duplication of the sub-tree $T_1$. For this reason, we perform such transformations only when optimizations of the first type have been exhausted.

## 5. 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 in a 1280 × 720 window. The first scene (*a*) shows a CSG model of the city at different scales (see Figure 7). In all cases the whole city is modeled as a single CSG tree containing 3385, 343K and 987K primitives correspondingly. Scene (*b*) demonstrates the case with huge number of depth layers that is rather challenging for other approaches. Number of holes in cheese model increases from 500 to 8000, and then to 32000 resulting in a larger number of overlapped primitives and greater depth complexity. The third scene (*c*) contains a large number of satellites, each of which is represented by a separate CSG tree. Our geometry representation is based *Geometry Description Markup Language* (GDML) that supports such basic primitives as parallelepiped, hexagonal prism, second-order algebraic surfaces (like sphere, cylinder, cone), and some high-order surfaces (torus). All the primitives are handled in GLSL directly (without tessellation).

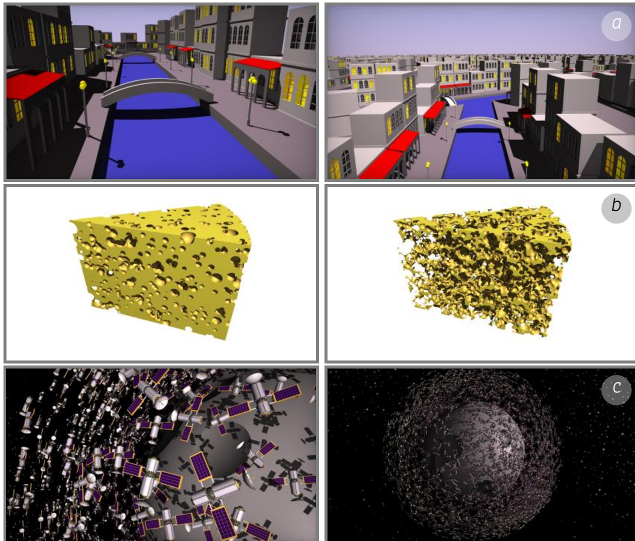


**Figure 7:** Test scene: City (*a*), Cheese (*b*), and Satellites (*c*).

For each GPU results are represented by two columns (see Table 2): left one corresponds to FPS without spatial optimization (−), and the right one was obtained with enabled spatial optimization (+). N/A markers show where the performance clearly cannot be considered to be interactive.

**Table 2:** Measured performance (in FPS) and comparison with OpenSG and IceSL [6].

| Scene | # Prims | Tree Depth | Intel 4000 | | Radeon 7870 | | GTX 680 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | − | + | − | + | − | + | OpenSG | IceSL |
| City | 3385 | 14 | 7 | 7.5 | 50 | 60 | 51 | 57 | | |
| | 343K | 22 | 1.8 | 4.5 | 6.5 | 17 | 8 | 22 | | |
| | 987K | 24 | 2.3 | 7 | 6.7 | 18 | 8.3 | 21 | | |
| Cheese | 502 | 10 | 1.2 | 31 | 7.5 | 193 | 11.2 | 211 | 21 | 1.1 |
| | 1002 | 11 | 0.4 | 17 | 4.6 | 110 | 5.8 | 128 | 6.5 | 0.3 |
| | 8002 | 14 | N/A | 6.5 | 0.5 | 28 | 0.5 | 32 | N/A | N/A |
| | 32002 | 17 | N/A | 0.5 | N/A | 3.7 | N/A | 4 | N/A | N/A |
| Satellites | 87.5K | 7 | 5 | 9 | 26 | 67 | 29 | 65 | | |
| | 1120K | 7 | 2.8 | 4.5 | 8 | 18 | 7 | 15 | | |

The main factors affecting performance are the *screen resolution* (as for over ray-tracing methods) and the *number of primitives*, but it does not affect performance directly. We can easily render scene (*a*) with more than 1 million CSG primitives while having trouble with 32K primitives in scene (*b*). This is due to extensive overlaps between the primitives in cheese model which force the algorithm to iterate over the CSG subtrees intensively. However, even in this stress scenario, we can show near linear performance degradation depending on the primitive number and outperform alternative solutions.

## 6. 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 algorithms our solution renders the model in single pass and does not impose restrictions on the complexity of CSG tree being limited only by the amount of GPU memory. We found that our implementation scales well with increasing GPU clock speed, while the memory clock does not affect performance. Thus, we can expect further performance increase on next-gen GPUs.

## 7. REFERENCES

[1] Goldfeather, J., Monar, S., Turk, G., Fuchs, H. *Near real-time CSG rendering using tree normalization and geometric pruning* // IEEE CG&A. 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 Trans. on Graph. 2005. Vol. 24, No. 3, P. 1024-1031.

[5] Romeiro, F., Velho, L., De Figueiredo L. H. *Hardware-assisted rendering of CSG models* // SIBGRAPI'06. 2006. P. 139-146.

[6] Lefebvre, S., Grand-Est, L. I. N. *IceSL: A GPU accelerated CSG modeller and slicer* // AEFA'13. 2013.

[7] Kensler A. *Ray tracing CSG objects using single hit intersections* (http://xrt.wdfiles.com/local--files/doc%3Acsg/CSG.pdf).

[8] Cameron, S. *Efficient bounds in constructive solid geometry* // IEEE CG&A. 1991. P. 68-74.

[9] Wald, I. *On fast construction of SAH-based bounding volume hierarchies* // IEEE Symposium on Interactive Ray Tracing. 2007. P. 33-40.