Parallel GPU Boolean Evaluation for CSG Ray-Tracing

Marco Domingues
Instituto Superior Técnico, Universidade de Lisboa, Portugal

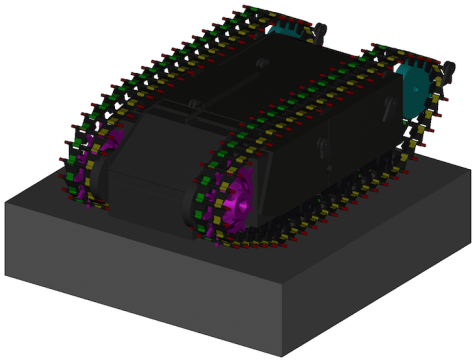


Figure 1: Render of the Goliath tracked mine, made of 3861 primitives and 1411 CSG trees.

Abstract

We present a novel parallel algorithm to perform Boolean evaluation for Constructive Solid Geometry ray-tracing on GPUs with OpenCL. By using a multi-hit ray traversal approach together with a list containing all the intersections between a ray and the solid objects in the scene, we are able to determine the sections of the ray that truly belong to the compound object, in two steps. First, we merge and sort all the intersection segments into partitions of the ray. Secondly, by using simple Boolean algebra, we evaluate the objects in the partitions against all the CSG trees. We demonstrate that our solution can efficiently render complex scenes when compared to a state of the art CPU-based algorithm. We achieve speedups of 42% on the same CPU hardware, and up to 31% when running the algorithm on the GPU, in some scenes.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Constructive solid geometry (CSG)

1. Introduction

Constructive Solid Geometry (CSG) is a solid modeling representation that combines simple primitive objects by using union, intersection and difference Boolean operators to create more complex geometry. In this modeling paradigm, a solid object is typically represented by a binary CSG tree, where the leaf nodes contain the primitive objects and the interior nodes hold the Boolean operators. The root of the tree describes the compound object that is formed by evaluating the Boolean expressions in the subtrees.

CSG solid modeling is often used in CAD/CAM/CAE (CAx) applications because it allows users to model complex solid objects from existing geometry. With the increased availability of desktop manufacturing tools, such as 3D printers and Computer Numerical Controlled (CNC) lathes, it becomes important to study ways of accelerating the rendering of CSG scenes for modeling purposes.

Rendering CSG objects with ray-tracing can be a very expensive task, since it requires computing multiple hits per ray to process the object. Despite being a time consuming algorithm, ray-tracing can be performed in parallel, as rays can be processed independently.

With that being said, we propose an algorithm to render CSG objects with ray-tracing on the GPU, taking advantage of the parallelism mechanisms found in this hardware to render CSG scenes quickly.

In summary, the contributions of our work include:

- an algorithm to perform CSG ray-tracing on the GPU using the SIMT programming paradigm.
- an implementation of a doubly-linked list data structure on the GPU.

- a compact linear CSG tree representation evaluated without the use of recursion.

2. Related Work

Several methods to render CSG objects have been studied and presented along the years, including ray-casting techniques, scan-line algorithms and z-buffer algorithms.

Roth [Rot82] presented an early method to ray-trace CSG scenes, where ray intervals are defined from intersecting the ray with the primitives and combined in a later stage according to the Boolean operators. The method that we propose is very similar to the Roth's algorithm, in the sense that we also combine the ray intervals in sorted order, forming partitions of the ray that we then evaluate. However, our solution is implemented in parallel on the GPU.

Jansen [Jan91] describes several methods of evaluating a CSG tree, and takes advantage of the coherence between rays to optimize the ray-tracing process of CSG scenes, skipping Boolean evaluation for adjacent rays that intersect the same primitives, in the same order of previous rays. Hizagi et al. [HKS*10] present an algorithm to render CSG of arbitrary primitives using interval arithmetic, and expressing the CSG object in terms of implicit functions. More recently, Mostajabodaveh et al. [MDG*17] organize the CSG model into layers and ray-trace each layer to determine the nearest hit point, on the GPU.

Others have found success rendering CSG by extracting a surface mesh and using Z-Buffer rasterization and depth-peeling techniques. Despite ray-tracing algorithms having the potential to achieve more photo-realistic images than rasterization methods, the former technique is often chosen in applications that require some interactivity, as it can be more efficient than using ray-tracing algorithms.

Goldfeather et al. [GMTF89] present a method to directly render CSG objects using a Z-Buffer rasterization approach, after restructuring the CSG tree to its disjunctive normal form (DNF). This solution has $O(n^2)$ complexity, where n represents the number of primitives in the scene. Rossignac [Ros99] renders CSG models by converting the CSG tree into a Boolean list formulation (Blist) that can be evaluated in a pipeline fashion on the GPU. Later, Hable & Rossignac [HR05] present the *Blister* algorithm, combining the Blist representation with depth peeling techniques to render CSG objects on the GPU. Using this approach, pixels with the same depth might be displayed with the wrong color. However, this issue is resolved in later work, by the same authors [HR07], where they present a method called Constructive Solid Trimming (CST). In the CST method, the depth-buffer is used to trim and peel the CSG object against the Blist tree representation.

The rendering of CSG objects can be further optimized by simplifying the CSG tree, which leads to reduced tree traversal times. In addition, Jansen [Jan91] mentions a method to prune the CSG tree according to its active zones [RV88].

To implement dynamically allocated data structures on the GPU, Mulder [Mul15] proposes a solution using the Kernel Memory Allocator (KMA), which reuses memory that is no longer needed, while Yang et al. [YHGT10] use two memory buffers to implement a linked list on the GPU. Yang's approach is easier to implement than Mulder's, but has the problem of allocating more memory than necessary, since memory buffers have to be allocated before they can be used by the kernels. To implement our data structure, we follow a similar approach, albeith with a single memory buffer.

3. Overview

When rendering a CSG object with ray-tracing it is common practice to intersect all the objects along the ray, instead of stopping at the first intersection point. This produces ray intervals that we label as intersection *segments*. Each segment has an entry and exit point, representing a section of the intersected primitive. Intersecting a ray with an object may lead to more than one segment, depending on the topology of the object. The torus, as seen in Figure 2, is an example of a primitive that may produce two ray intersection intervals.

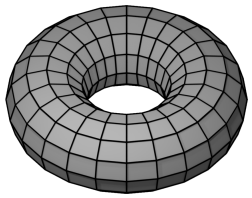


Figure 2: Torus.

Having the list with all the intersections segments of the ray, we "weave" the segments. At this stage of the algorithm, we combine the segments into ray *partitions*, according to the distance between the segment entry point and the ray origin. While a segment is relative to a single primitive, a ray partition describes sections of the compound object along the ray, and it may be formed by one or more segments.

To evaluate a partition, we first build a table with all the CSG trees involved with the partition. Then we traverse each of the CSG trees in the table by using Boolean algebra in order to verify if the partition represents geometry of the composite object.

Since more than one CSG tree traversal can lead to a partition being evaluated, it is necessary to resolve any overlaps that may occur during the process. Finally, we shade the closest partition relative to the ray origin, using the normal of the partition entry point, and the material associated with the respective CSG tree.

We illustrate the aforementioned process in Algorithm 1:

3.1. Weave of Segments

To weave segments, we iterate over the segments of the ray (see Figure 3), and then we compare the distance between the segment entry point and the partition exit point. If no partition is created, we start a new partition with the segment. For the subsequent segments, we basically iterate over the partition, and we fit the segment in one of the partitions created by extending the exit point of the partitions or, if necessary, by appending new partitions to the end of the list.

```
for each pixel do
    ray ← GenerateRay();
    Segments ← ShootRay(ray);
    Partitions ← BoolWeave(Segments);
    // Evaluate Partitions
    for each partition in Partitions do
        Regiontable ← BuildRegiontable(partition);
        for each region in Regiontable do
            Eval(partition, region);
        end
        OverlapHandler(partition, Regiontable);
    end
    Shade(Partitions);
end
```

Algorithm 1: CSG ray-tracing.

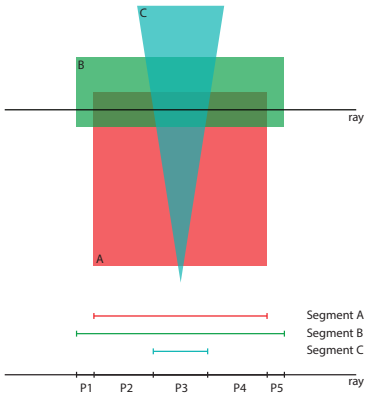


Figure 3: Ray-primitive intersection segments.

Doubly-Linked List

To merge the segments together into the solid ray partitions, we need to implement a data structure on the GPU to hold the partitions for each ray. We start by allocating a memory buffer whose dimensions are twice the number of segments, since this is the maximum possible extent required to store all the partitions. This method has a limitation since it can allocate more memory than the one it is effectively used. An alternative would be to run the weave of segments in two steps: firstly to determine the total number of needed partitions and secondly to allocate the memory and to perform the partitions storing.

We use a memory buffer to store a doubly-linked list of partitions for each ray. By using the thread's unique global ID to compute the memory offset, we guarantee that two different rays do not access the same location in memory.

Each partition stores the information of both entry and exit hit points as well as the segments of all the primitives that contribute to the partition. To implement the doubly-linked list, each partition also holds the index to the back and forward partition. Both appending and insertion operations are performed in $O(1)$ complexity, as we simply create the new partition at the end of the local buffer for the ray. This is possible by storing in variables the indexes to the head and tail of the list. These operations only differ on the way that they update the back and forward indexes. We illustrate our doubly-linked list representation in Figure 4.

Dynamic Bit Arrays

In order to know which segments contribute to the partition, we use a dynamic bit array where we simply set the bits corresponding to the segments. This has the advantage of using less memory than storing the actual segments and insertions can be made in constant time. If, instead, we used a list to store the segments of the partitions, it would require more memory and insertions would have $O(n)$ complexity because the list can not have duplicate elements. Despite the mentioned disadvantages, iterating over the list during the evaluation of partitions can be faster than our solution of dynamic bit-arrays. This is the case for very sparse bit arrays. We mitigate this issue by using the *clz* instruction to skip over zero entries.

3.2. CSG Tree Representation

Traversing and evaluating a CSG tree on the GPU can be a challenging task, since recursion is not allowed and the available memory is limited. To overcome this problem, we use a linearized tree representation and a stack to hold temporary values of the tree during evaluation.

We first attempted to represent the CSG tree in postfix notation, where each element of the array contained either the *id* of a primitive, or the Boolean operator associated. This method had the limitation of requiring all the elements of the tree to be processed to evaluate the compound object. When evaluating a CSG tree, the structure of the tree should be taken into consideration to skip subtrees that will not influence the outcome of the Boolean evaluation. For example, when intersecting two objects, if the left subtree has value $false$, then there is no need to process the right subtree, because the result will be $false$, following the Boolean algebra rules. Using a postfix tree representation does not allows us to easily implement this sort of optimizations.

We solve the early out problem by converting the CSG tree to an array form which is traversed in depth-first order. With this representation, we fit each element of the tree in 32 bits length, having a structure that requires less memory and that is able to skip unnecessary subtrees. To store each tree node in 32 bits, the following convention was adopted: the 3 most significant bits representing the operator, and the remaining 29 bits to represent either the position of the right child, or for leaf nodes, the *id* of the primitive. The operator zero indicates a leaf node. With this representation, a CSG tree can have up to 2^{29} nodes and can be used in scenes with 2^{29} primitives. These limits can be increased by representing each node with more bits and adopting a similar convention.

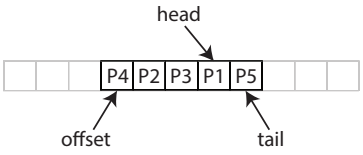
3.3. Evaluating Ray Partitions

To evaluate a ray partition, we first build a table with all the regions involved with a given partition. A region is basically a data structure that contains a CSG tree, and the material of the compound object. The material is used in a later stage to shade the partition. Since the memory on the GPU is a very limited resource, we use a dynamic bit-array to implement the table of regions, similar to what we did with the segments.

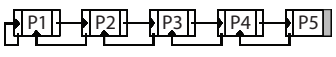
For each partition, we iterate over its segments, and then we check if the segment *id* is present on the CSG tree. This operation is extremely slow as a partition may be formed by several segments and a scene may contain thousands of large CSG trees. Having to repeat this procedure at runtime for each partition in the ray caused a huge bottleneck when rendering the scene. By pre-computing the list with all the regions associated with a given primitive, we were able to achieve a speedup of 80%.

By having the table with all the regions associated with the partition, we start with the traversal of the CSG trees. Note that a CSG tree containing only union operators does not need to be traversed, because all the segments of the partition will contribute to the compound object.

Since we evaluate a partition against all the regions involved, we might have occurrence of overlaps, i.e. where more than one region lead to a partition being evaluated. This is a problem because each region contains the material of the object and only one material can be used to shade the partition. Therefore, all the overlaps have to be resolved before a partition can be shaded. To resolve overlaps, we iterate over the region table, picking two regions at a time, and deciding which of the regions should claim the partition. To make the decision, we compare the *id* of the two regions with the region *id* of the previous partition in the ray. If there are no other partitions, we simply chose the region with smallest *id* number. We repeat this process until there is only one set bit in the region table bit array. We stop evaluating partitions when we find the first partition evaluated for the ray. This is possible because at this stage, we have processed all the hits, and the partitions are ordered by the distance to the ray origin. The first partition evaluated is guaranteed to be the partition with the nearest entry point, and it is shaded according to the region material.



(a) Section of the memory buffer.



(b) Doubly-Linked List representation.

Figure 4: Partitions from the example in Figure 3. a) How the partitions are stored in the memory buffer. b) How the partitions are represented with the doubly-linked list.

4. Implementation

To parallelize our algorithm on the GPU, we use the OpenCL 1.2 compute API. We reduce thread divergence by using a pipelined system, where we have one kernel per each stage of our algorithm: storing segments, weaving segments, evaluating partitions and shading partitions.

We use a Bounding Volume Hierarchy (BVH) [PL10] acceleration structure to optimize the intersection between rays and the primitives in the scene. We perform every calculation in double-precision, i.e 64-bit floating point, as it leads to more accurate results.

Our solution is implemented in the BRL-CAD [Muu95] solid modeling application, which is an open-source project since 2004. The source-code of our implementation is freely available at the BRL-CAD project website on SourceForge [Sou17].

5. Evaluation

To analyze the performance of our parallel CSG ray-tracer, we use a set of six CSG scenes with variable levels of complexity, both in the number of primitives and in the number of regions used. The exact number of primitives and regions used in each scene is shown in Table 1.

Table 1: Characteristics of the Test Scenes.

SCENE	PRIMITIVES	REGIONS
Boolean Ops	3	1
Operators	32	20
Truck	182	146
Tank Car	787	180
Havoc	2 427	308
Goliath	3 861	1 411

With these scenes we were able to compare our implementation with the legacy CPU-based ray-tracing algorithm in BRL-CAD, which provides the basis to the algorithm solution we have implemented. The legacy algorithm also creates ray partitions from intersection segments, and it uses the same method to evaluate partitions. However it uses different data structures.

For example the legacy algorithm implements sets with unique lists which have an $O(n)$ insertion complexity, while we implement them with bit arrays with $O(1)$ insertion complexity. In addition, we have implemented the algorithm in parallel on the GPU and use a Bounding Volume Hierarchy (BVH), while the legacy algorithm uses a space-partitioning kd-tree.

Since BRL-CAD supports highly complex primitives, ray-primitive intersection is more computationally expensive than in a ray-triangle ray-tracer. In a BVH the primitives are only intersected once per ray. To achieve the same with a spatial-partitioning scheme we would need to use mailboxes. However mailboxes, as implemented on the legacy algorithm, would severely increase

our per-thread memory requirements, thus reducing the amount of threads which can simultaneously be in flight on the GPU. This is not an issue with the legacy implementation since it is optimized to run over multi-threaded processors.

The test scenes are freely available along with the BRL-CAD source code for third party use and evaluation.

Because we have implemented the solution with OpenCL, we are able to run our code in different hardware, ranging from CPUs to GPUs. For CPUS, we used an Intel Core i5 - 4790k and an AMD Ryzen 5 1600. We used the NVIDIA GeForce GTX 1060 and the NVIDIA GeForce GTX Titan to conduct the tests on the GPU. To execute the OpenCL code on the CPU, we used the Intel and AMD OpenCL SDK. We have only used NVIDIA GPUs to test our code. Therefore, we are only able to test the solution on the GPU with the NVIDIA OpenCL SDK.

The results presented in the remainder of the section were taken by rendering the CSG scenes with ray-tracing using a resolution of 1024x1024 pixels, and a perspective view with 35 degrees of elevation and 25 degrees of azimuth.

Memory Usage

With the objective of further reducing the wasted memory on the GPU, we have calculated the number of used partitions in each scene, and we compared that number with the total number of partitions allocated. Table 2 presents the total memory allocated on the GPU, for each scene, as well as the percentage of used partitions.

The results show that using the maximum possible number of partitions for this amount of intersection segments, i.e. twice the number of segments, is not an optimal solution as it leads to a considerable percentage of unused partitions by our algorithm. This gets worse as the depth-complexity of the scene increases. As mentioned earlier, it is possible to allocate the exact number of partitions in GPU memory by repeating the stage of weaving segments and determining the exact number of partitions required in each scene, which would result in higher rendering times when performing ray-tracing with our solution. We actually use this method to store the segments in memory since an estimation of the segments would result in a larger amount of wasted memory, since the partitions are calculated based on that number.

Time Efficiency

To determine the efficiency of our solution, we have measured the time to ray-trace each one of the six test scenes and we compared those values with the times obtained by rendering the scenes with BRL-CAD's legacy CPU-based algorithm. Because the depth-complexity of the scene is an important factor to understand the efficiency of our solution, we show in Figure 6 a color map representing the depth-complexity of each scene, i.e. the maximum number of segments per ray, along with a render of the scene and the execution times, with both our solution and with the legacy algorithm.

We can see that our solution is able to render scenes faster than the legacy algorithm, achieving speedups of 42% on the same CPU hardware, and up to 31% when running the code on the GPU. This

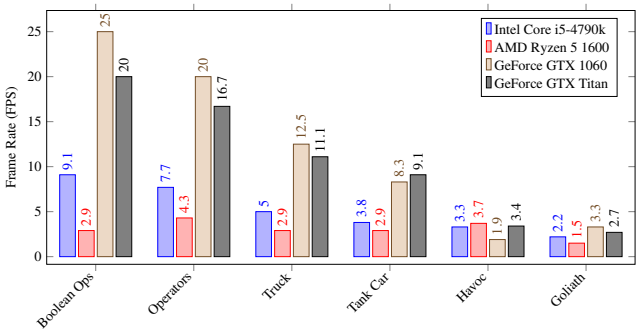


Figure 5: Frame rate when ray-tracing each scene on the Intel Core i5-4790k, on the AMD Ryzen 5 1600, on the NVIDIA GeForce GTX 1060 and on the NVIDIA GeForce GTX Titan.

is respectively observable, in the HAVOC test scene, and in the TANK CAR scene. However, it is still slower or similarly as fast as the legacy algorithm in some scenes.

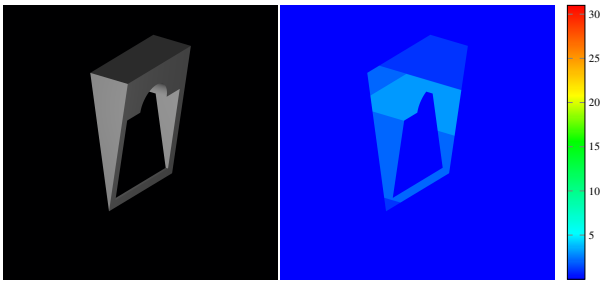
Performing ray intersection calculations twice to reduce the unused GPU memory is a limitation of our work that leads to higher times when rendering the scene, since intersecting rays with the primitives is an expensive operation. It is also important to mention that the legacy ray-tracing algorithm is optimized to evaluate the ray partitions in partial fashion, starting to process the segments as soon as they are created. In this way we might avoid having to compute all the intersection points along the ray, since the initially computed segments might lead to a partition being evaluated. This optimization is possible in the legacy algorithm since it uses a spatial partition acceleration structure, but this is not so easy to implement with the Bounding Volume Hierarchy (BVH), as it is an object partition acceleration structure, where the intersections are not computed in depth order.

Summarizing, the partial evaluation of hits, facilitated with a spatial partitioning kd-tree, leads to faster performance in scenes with high depth complexity, than an object partitioning BVH. This is particularly evident in the GOLIATH scene which has high depth complexity in a small amount of rays.

In Figure 5 we present a chart comparing the frame rates when rendering the test scenes on the Intel Core i5-4790k, on the AMD Ryzen 5 1600, on the NVIDIA GeForce GTX 1060 and on the NVIDIA GeForce GTX Titan. Both NVIDIA's GPUs have vastly different processing power when performing calculations using double-precision, respectively, 120 and 1500 GFLOPS. As we can see in the chart, the GPU with more double-precision processing power will often outperform the less capable GPU. By using a workstation GPU, the rendering of CSG scenes could be extremely fast.

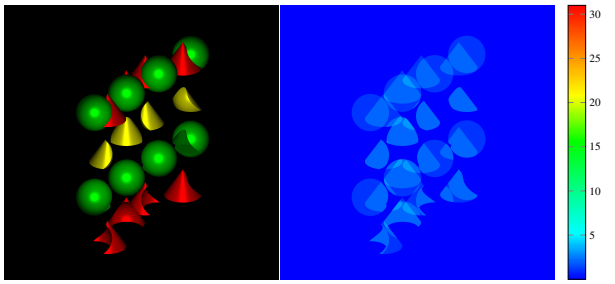
6. Conclusions and Future Work

We have presented an algorithm to efficiently render CSG models on the GPU using ray-tracing. Our solution requires little memory per thread, which is an advantage considering that the available memory per thread on the GPU is very limited, thus increasing the



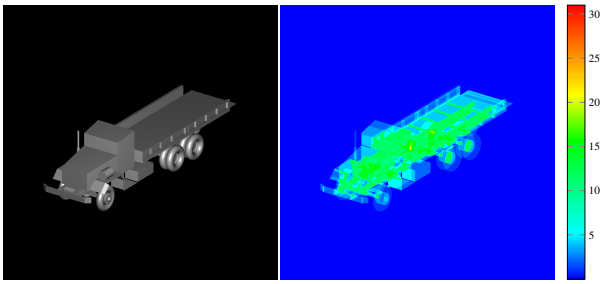
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	70 ms	110 ms	150 ms	
AMD Ryzen 5 1600	50 ms	340 ms	340 ms	
NVIDIA GeForce GTX 1060				40 ms
NVIDIA GeForce GTX Titan				50 ms

(a) Boolean Ops test results.



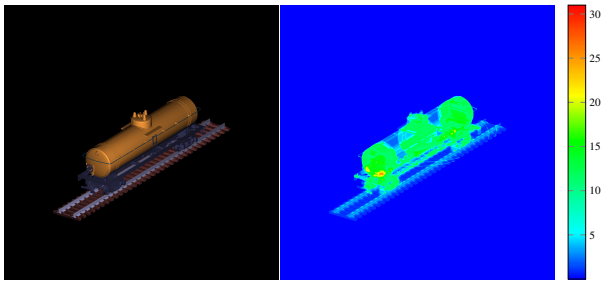
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	120 ms	130 ms	180 ms	
AMD Ryzen 5 1600	60 ms	280 ms	230 ms	
NVIDIA GeForce GTX 1060				50 ms
NVIDIA GeForce GTX Titan				60 ms

(b) Operators test results.



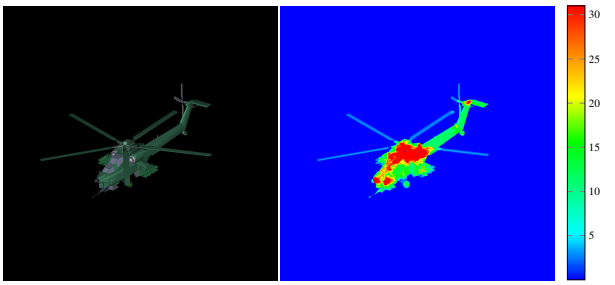
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	80 ms	200 ms	290 ms	
AMD Ryzen 5 1600	50 ms	350 ms	360 ms	
NVIDIA GeForce GTX 1060				80 ms
NVIDIA GeForce GTX Titan				90 ms

(c) Truck test results.



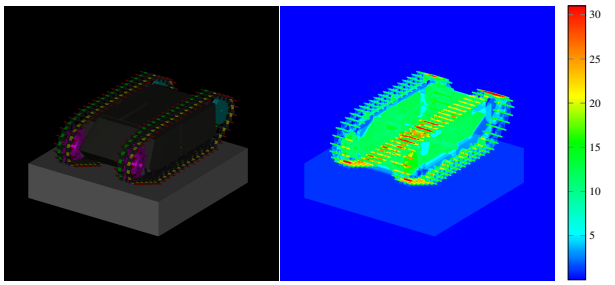
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	280 ms	260 ms	330 ms	
AMD Ryzen 5 1600	160 ms	350 ms	350 ms	
NVIDIA GeForce GTX 1060				120 ms
NVIDIA GeForce GTX Titan				110 ms

(d) Tank Car test results.



	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	520 ms	300 ms	320 ms	
AMD Ryzen 5 1600	290 ms	270 ms	270 ms	
NVIDIA GeForce GTX 1060				520 ms
NVIDIA GeForce GTX Titan				290 ms

(e) Havoc test results.



	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	Nvidia SDK
Intel Core i5-4790k	500 ms	460 ms	650 ms	
AMD Ryzen 5 1600	310 ms	880 ms	870 ms	
NVIDIA GeForce GTX 1060				300 ms
NVIDIA GeForce GTX Titan				370 ms

(f) Goliath test results.

Figure 6: Render image result with depth complexity color map and table displaying the time results, in milliseconds, for each scene. The results include the execution times of the legacy BRL-CAD ray-tracing algorithm, when executed in the Intel i5-4790k and in the AMD Ryzen 5 1600 CPUs, so it can be compared with the OpenCL results obtained by performing ray-tracing with our solution.

Table 2: Total memory allocated on the GPU for each scene.

SCENE	SEGMENTS	USED PARTITIONS	ALLOCATED PARTITIONS	PARTITIONS USAGE RATIO [%]	ALLOCATED MEMORY [MBS]
Boolean Ops	433 443	573 728	866 886	66.2	379.5
Operators	405 319	516 066	810 638	63.7	356.1
Truck	796 168	1 010 303	1 592 336	63.4	697.2
Tank Car	651 742	915 691	1 303 484	70.2	581.3
Havoc	701 340	958 081	1 402 680	68.3	650.0
Goliath	1 710 913	1 875 823	3 421 826	54.8	1 642.7

amount of threads concurrently in flight. The algorithm is of practical relevance since it can be used in modeling applications to render CSG objects, often used in CAD models, effectively in parallel on the GPU. It may also be used in analysis tools.

To minimize the amount of memory used in the GPU, we sacrifice execution time to accurately calculate the total number of segments in the scene. We do this by intersecting the objects in the scene twice. The first pass to count all the hits in the scene, to determine how much memory we need to allocate, and the second pass to store the segments, resulting in a perfect fit of segments in memory. This also means less GPU memory is wasted when creating the ray partitions. This is a limitation of our work, since intersection calculations are time consuming, and we also perform every calculation in double-precision floating point, which is slower than using single-precision. The efficiency of our work could be further improved by studying heuristics to determine the maximum number of segments in the scene, as well as heuristics to estimate the number of partitions from the calculated number of hits.

Rendering CSG with ray-tracing can also be optimized by processing the segments and evaluating the ray partitions in partial fashion, since the first object intersected by the ray could lead to a evaluated partition, avoiding further primitive intersections to be calculated for the ray. To accomplish this objective, a space partitioning data structure would be required instead of the Bounding Volume Hierarchy (BVH) we used in our solution. We recommend any future work to start with this optimization, as it may reduce significantly the rendering time of complex CSG scenes and the total memory needed by the algorithm.

Acknowledgements

References

[App68] APPEL A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (1968), ACM, pp. 37–45.

[BRL16] BRL-CAD: Open Source Solid Modeling webpage. http://brlcad.org/, 2016.

[GMTF89] GOLDFEATHER J., MONAR S., TURK G., FUCHS H.: Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications 9*, 3 (1989), 20–28. 2

[HKS*10] HIJAZI Y., KNOLL A., SCHOTT M., KENSLER A., HANSEN C.: CSG Operations of Arbitrary Primitives with Interval Arithmetic and Real-Time Ray Casting. In *Dagstuhl Follow-Ups* (2010), vol. 1, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2

[HR05] HABLE J., ROSSIGNAC J.: Blister: GPU-Based Rendering of Boolean Combinations of Free-Form Triangulated Shapes. In *ACM Transactions on Graphics (TOG)* (2005), vol. 24, ACM, pp. 1024–1031. 2

[HR07] HABLE J., ROSSIGNAC J.: CST: Constructive Solid Trimming for Rendering BReps and CSG. *IEEE transactions on visualization and computer graphics 13*, 5 (2007). 2

[Jan91] JANSEN F. W.: Depth-Order Point Classification Techniques for CSG Display Algorithms. *ACM Transactions on Graphics (TOG) 10*, 1 (1991), 40–70. 2

[MDG*17] MOSTAJABODAVEH S., DIETRICH A., GIERLINGER T., MICHEL F., STORK A.: CSG Ray Tracing Revisited: Interactive Rendering of Massive Models Made of Non-planar Higher Order Primitives. In *VISIGRAPP (1: GRAPP)* (2017), pp. 258–265. 2

[Moo79] MOORE R. E.: *Methods and Applications of Interval Analysis*. SIAM, 1979.

[Mul15] MULDER H.: Concurrent Manipulation of Dynamic Data Structures in OpenCL. 2

[Mun09] MUNSHI A.: The OpenCL Specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE* (2009), IEEE, pp. 1–314.

[Muu95] MUUSS M. J.: Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. 4

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics* (2010), Eurographics Association, pp. 87–95. 4

[Ros99] ROSSIGNAC J. R.: *BLIST: A Boolean List Formulation of CSG Trees*. Tech. rep., Georgia Institute of Technology, 1999. 2

[Rot82] ROTH S. D.: Ray Casting for Modeling Solids. *Computer Graphics and Image Processing 18*, 2 (1982), 109–144. 2

[RV88] ROSSIGNAC J. R., VOELCKER H. B.: Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection, and Shading Algorithms. *ACM Transactions on Graphics (TOG) 8*, 1 (1988), 51–87. 2

[SHGV14] SPLIET R., HOWES L., GASTER B. R., VARBANESCU A. L.: KMA: A Dynamic Memory Manager for OpenCL. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (2014), ACM, p. 9.

[Sou17] SourceForge project website. https://sourceforge.net/projects/brlcad/, 2017. 4

[UBT16] ULYANOV D., BOGOLEPOV D., TURLAPOV V.: Spatially Efficient Tree Layout for GPU Ray-tracing of Constructive Solid Geometry Scenes.

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. In *Computer Graphics Forum* (2010), vol. 29, Wiley Online Library, pp. 1297–1304. 2