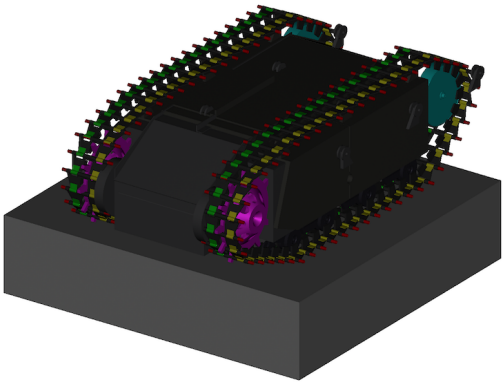
TÉCNICO LISBOA



Parallel GPU Boolean Evaluation for CSG Ray-Tracing

Marco da Silva Domingues

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João António Madeiras Pereira
Eng. Vasco Alexandre da Silva Costa

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. João António Madeiras Pereira
Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

October 2017

Acknowledgments

I would like to express my deep gratitude to Dr. Vasco Costa, my research supervisor, for his amazing mentoring and guidance through the year. His willingness to give his time so generously has been very much appreciated.

I would also like to thank Professor João Madeiras Pereira, my research supervisor, for giving me the chance to work on this research subject, and for the valuable feedback on my work.

Special thanks to the BRL-CAD community, for accepting my proposal to work with them on this topic under the Google Summer of Code program, and for their willingness in helping me when necessary.

On a more personal note, I would like to thank my parents for always supporting me, and for always making everything they could to make sure I could finish my studies successfully.

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.

Keywords

CSG, Boolean Evaluation, GPGPU, Ray-Tracing, OpenCL

Resumo

Apresenta-se um novo algoritmo para realizar a avaliação booleana em paralelo para o ray-tracing de Geometria Sólida Construtiva em GPUs, utilizando OpenCL. Usando um paradigma de travessia de raios que guarda todos os pontos de intersecção ao longo do raio, em conjunto com uma lista contendo todas as intersecções entre o raio e os objetos sólidos na cena, nós determinamos as secções do raio que realmente pertencem ao objeto composto, em dois passos. Primeiro, combinamos todas os segmentos de intersecção em partições do raio, ordenados por distância à origem do raio. Depois, usando álgebra booleana simples, avaliamos os objetos nas partições com todas as árvores CSG envolvidas. Demonstra-se que ao usar a nossa solução, cenas CSG complexas podem ser renderizadas efetivamente quando comparado com um algoritmo do estado da arte que corre no CPU. Nós alcançamos acelerações de 42% no mesmo CPU hardware, e até 31% ao executar o algoritmo no GPU, em algumas cenas.

Palavras Chave

CSG, Avaliação Booleana, GPGPU, Ray-Tracing, OpenCL

Contents

List of Figures

List of Tables

Abbreviations

CSG Constructive Solid Geometry

CPU Central Processing Unit

GPU Graphics Processing Unit

CNC Computer Numerical Control

Blist Boolean List

CST Constructive Solid Trimming

BVH Bounding Volume Hierarchy

OpenCL Open Computing Language

KMA Kernel Memory Allocator

GPGPU General-Purpose Computing on Graphics Processing Units

SDK Software Development Kit

GSoC Google Summer of Code

1

Introduction

Contents

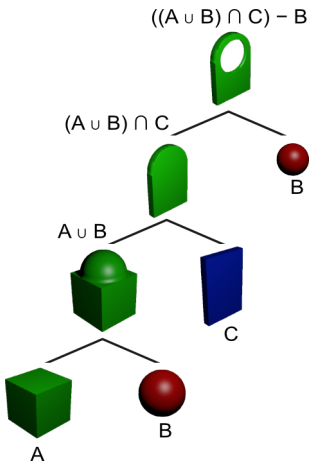


Figure 1.1: CSG Tree.

Constructive Solid Geometry (CSG) is a solid modeling representation that combines simple primitive objects, such as a cube, a sphere or a cylinder for example, 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. We illustrate the representation of a CSG object using a CSG tree in Figure 1.1.

A CSG scene may be rendered with ray-tracing, by traversing the CSG tree starting at the leaf nodes. Traversal of each node yields a list of line segments of the ray that pass through the solid object. This list is then passed to the parent node and the line segments are merged according to the Boolean operation of that internal node.

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 Control (CNC) lathes, it becomes important to study ways of accelerating the rendering of CSG scenes for modeling purposes. By implementing the Boolean evaluation on the Graphics Processing Unit (GPU) we can take advantage of the computational power and parallelism of this hardware to accelerate the rendering of CSG objects.

1.1 Problem

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 quicker than it would be by using the Central Processing Unit (CPU).

1.2 Objectives

The main purpose of this work is to research and implement a parallel algorithm to render CSG objects using ray-tracing with the Open Computing Language (OpenCL) compute API, exploiting the massively parallel architecture found in modern GPUs, which are designed to handle multiple tasks simultaneously.

With this work, we aim to investigate and answer to the following questions:

- How can the Boolean evaluation of CSG trees be implement on the GPU.
- What is the performance of this implementation?
- How does our solution perform compared to a state of the art CPU-based implementation?

1.3 Contributions

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.

1.4 Thesis Outline

The remainder of this document is organized as follows:

CHAPTER 2 describes some related work that has been done to render CSG scenes, both using ray-tracing and rasterization techniques, and it shows some algorithms to optimize and evaluate CSG trees. We also present some methods to create data structures on the GPU and we describe the core of the OpenCL architecture.

CHAPTER 3 presents an overview of our solution, and shows in detail how each step of the algorithm was implemented.

CHAPTER 4 describes how the solution was evaluated and presents the results obtained when rendering CSG scenes using our algorithm. It is also made a comparison with the results obtained by a state of the art CPU-based algorithm.

CHAPTER 5 summarizes the conclusions of our work and presents some ways to further improve our solution.

2

Related Work

Contents

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. In this chapter, we will present some of those studied solutions to render CSG models.

We also present some methods to optimize the CSG tree associated with a CSG object, with the removal of unnecessary nodes, which leads to reduced tree traversal time and faster Boolean evaluation.

To implement our algorithm on the GPU, we will have to implement efficient data structures in OpenCL. With that being said, we will also describe some implementations of data structures on the GPU and present the OpenCL framework.

2.1 CSG Rendering

In the scope of computer graphics, the term rendering is often used to refer to the process of generating a photorealistic or non-photorealistic 2D image from a 3D model. To achieve photorealistic representations of 3D objects, it is important to take in consideration some properties of the material that we are trying to replicate and understand how the lighting can affect the color and brightness of the object that we intend to display on screen.

Two rendering techniques that are commonly used in 3D computer graphics are ray-tracing and rasterization. The former technique is able to achieve a tremendous level of realism, but it has a very high computational cost associated and because of that is not common in applications that require interactivity. The latter is an extremely fast technique to render 3D objects when compared to ray-tracing algorithms, and although the same level of realism not being always accomplished, it is still a very popular rendering technique in real-time applications.

In ray-tracing algorithms the image is created by shooting rays from the eye position through each pixel of the window and intersection tests are made to check if the rays do intersect with the objects in the scene or not. The goal is to find the closest intersection point as it represents the object that is in front, and then use the material properties of the object and the lights on the scene to determine the shading of that specific object. This method of shooting rays against the scene is illustrated in Figure 2.1, where a ray is shot from the eye position and it goes through a pixel of the picture plane until it intersects the object.

The first ray-tracing algorithm was introduced by Arthur Appel [2] in 1968, where a ray was shot from the camera position through each pixel, and in case of intersection with an object, extra rays were cast from the intersection point to the position of each light in the scene, to determine if the point was in shadow or not.

Later in 1979, Turner Whitted [20] came with a recursive ray tracing algorithm that allowed to render more realistic images, by recursively casting extra rays at the intersection point to calculate shadows, reflections and refractions, if those properties made sense to the object being rendered.

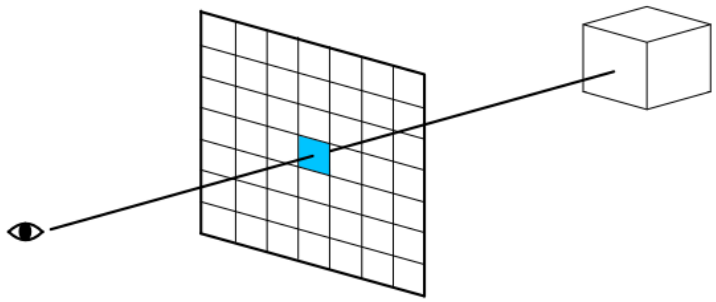


Figure 2.1: Ray-tracing example.

2.1.1 CSG Ray-tracing

Rendering CSG scenes with ray-tracing works in a similar way, but usually not only is the closest intersection point calculated, but all the intersections points along the ray. This allow us to define ray intervals, that can be later processed according to the Boolean operations involved with the intersected objects, in order to determine the hit point that should be used to shade the pixel.

The first algorithm for directly rendering a CSG scene without extracting a surface mesh was presented in 1982, by Roth [17]. Roth's algorithm can be divided in two essential parts: intersecting the rays with the primitives in the scene, and then combining left and right classifications. The algorithm starts at the top of the CSG tree and descends to the bottom of the tree recursively, classifying the ray with respect to the primitive objects, and then returning up the tree combining the classifications of left and right subtrees. The process of combining left and right classifications can be separated in three distinct steps. First, in the merging step, the intersection points from the left and right rays are combined in sorted order. Then, the segments of the composite ray are classified as *in* or *out*, depending on the Boolean operation associated and the classifications of the left and right rays along those segments. Finally, the composite ray is simplified by merging contiguous segments that have the same classification. In Figure 2.2, is illustrated an example of the combine process for the union operation between two primitive solids. The left and right ray segments are merged, by sorting their intersection points. Then, using simply Boolean algebra, each interval of the composite ray is classified with base on the Boolean operation involved and finally the composite ray is simplified.

In the research done by Jansen [7], different methods of evaluating a CSG tree by using bitwise operations were presented. These methods use the idea of representing the classification of a point with a bit-string, using a 0 to classify a primitive as *out* and a 1 to classify the primitive as *in*, which allows the use of bitwise operations to evaluate the CSG tree.

The Bit-Sequential is a CSG tree evaluation method in which the CSG tree is traversed in post order, and in each internal node, a bitwise operation is performed on the bits of the left and right subtrees. This method uses a $stack$ to store the intermediate results of performing the bitwise operations, and the final result of the CSG tree evaluation is stored in the $stack[n]$ position, where n represents

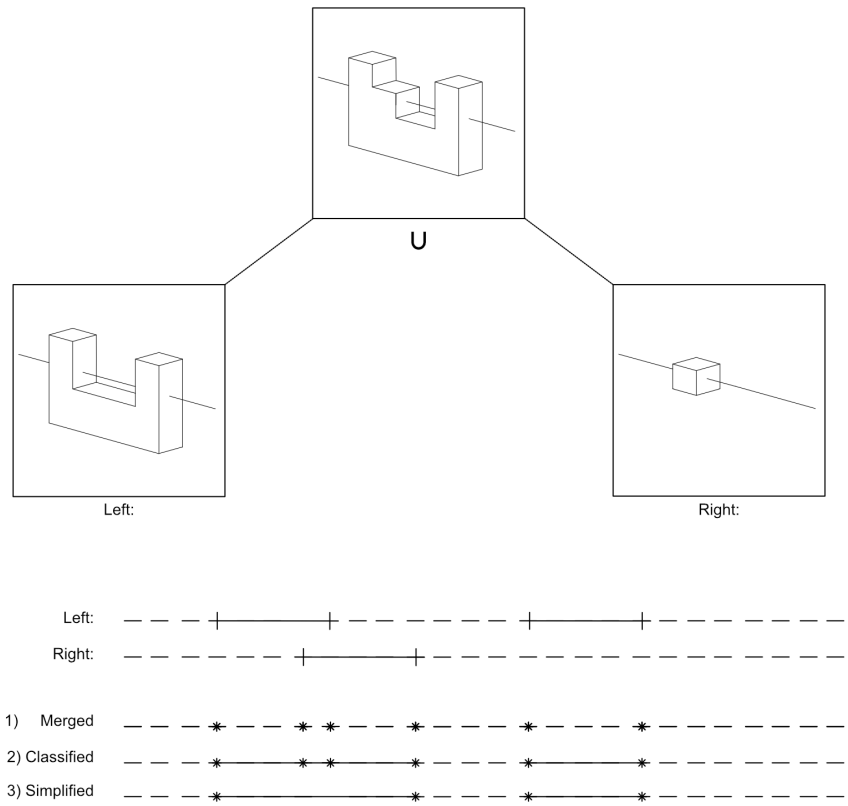


Figure 2.2: Example of the Three Step Combine Process (Source: "Ray Casting for Modeling Solids"[17]).

the depth of the tree.

The Bit-Parallel method is an alternative to the Bit-Sequential solution, in which all nodes on one level of the tree can be evaluated in parallel. In this method, a bitwise operation is performed between the bit-string representing all the nodes at one level and a shifted version of that same string. The number of left shifts that have to be applied to the second operand depends on the level of the tree that is being evaluated.

To evaluate a CSG tree using this method, first a bit-string is created for each level of the tree, in a pre-processing stage. Then, at each level of the CSG tree, a bitwise operation is performed between the bit-string of that level and the left shifted version of the same bit-string. Doing this for all the levels of the CSG tree will result in a bit-string at the root where the most significant bit represents the result of the CSG tree evaluation.

Jansen also describes how CSG coherence can reduce the processing time of CSG tree traversal. CSG coherence is about the idea of exploiting coherence between different rays, since for adjacent rays, the order in which primitives are intersected along the ray will be the same, thus the classification of intersection points will also be the same.

To take advantage of CSG coherence, all the sequences of intersection points have to be stored in a compact data structure. This data structure could be a list, in which each element corresponds to an intersection point. It stores an identifier for the primitive and the respective *in* or *out* classification of that intersection point.

If for a new ray, the order of intersection points is the same as the order of primitive identifiers on

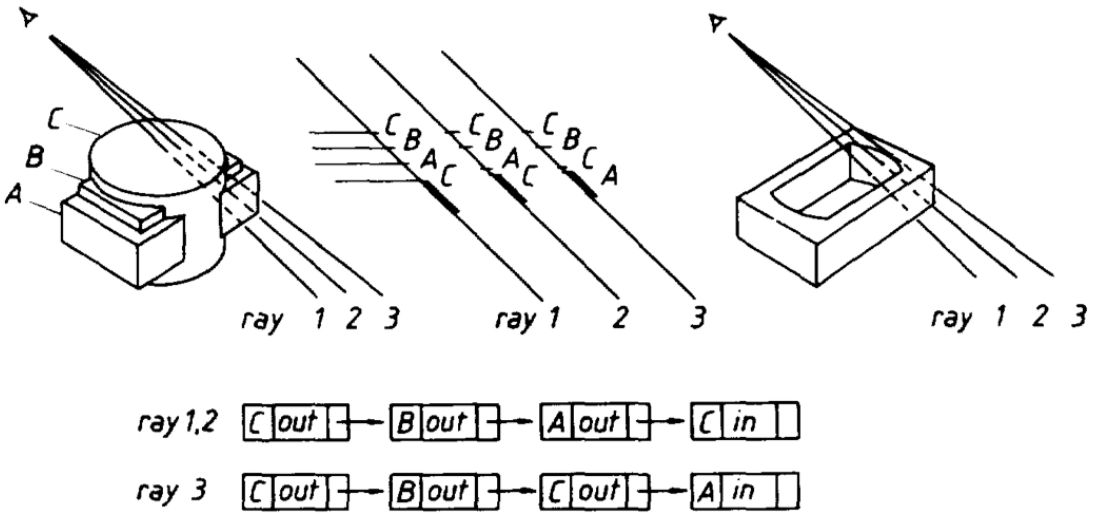


Figure 2.3: One List coherence method (Source: "Depth-order point classification techniques for CSG display algorithms"[7].)

the list, then the classification of that point can be read from the list. On the other hand, if the order of the primitives changes, or a new primitive is encountered along the ray, then the classification may change and the list must be modified.

Figure 2.3 shows an example of the One List CSG coherence method, where four intersection points of ray 1 are classified before an *in* interval was encountered. Because ray 2 encounters the primitives in the same order as ray 1, there is no need to reclassify the intersection points, as the classification can be read from the list. For ray 3, the sequence of intersection points changes, so the new points have to be classified and the list is modified.

Other data structure that is described to implement CSG coherence is the Sequence List. This data structure records all possible intersection sequences for a given solid object, by combining lists and providing access to each sequence start at the common root. In the Sequence List coherence method, the root node is a list of elements where each element has the information about the first intersection point classification for a primitive.

For the first intersection point of a ray, the intersected primitive has the reference of the respective list element. For subsequent rays, a list is created in the same manner as described in the data structure mentioned above. If sequences of classifications with a different ordering of primitives occur, then new branches in the list should be created to accommodate the changes.

Tests done in Jansen's research show that combining the status-tree traversal method with the first list data structure described may reduce significantly the time spent evaluating the CSG tree.

The status-tree is an extended version of the CSG tree that stores at each node the result of the combine operation of the node, applied to the classification of its two child nodes. In this method, bottom-up traversal is performed to the parts of the tree that may change. The traversal stops when no status change occurs or the root node is reached.

In 2010, Hijazi et al. [6] presented an algorithm to render CSG of arbitrary primitives using interval

arithmetic [9] and real-time ray casting. The principal idea behind Hijazi's algorithm is to represent solid objects with implicit functions and then use interval arithmetic to determine if a given ray interval contains the object's surface. An implicit surface in 3D space can be defined as the set of solutions of an equation

$$\mathbf{f}(x, y, z) = 0,\ f : \Omega \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}. \tag{2.1}$$

If we consider two primitive solids A and B to be respectively represented by the implicit functions f_a and f_b, and following the convention that $f < 0$ represents inside the solid, $f > 0$ outside the solid and $f = 0$ to define the solid's surface, then the Boolean operators used in CSG can be expressed in terms of implicit functions. Therefore, the union between A and B can be defined by

$$\mathbf{A} \cup \mathbf{B} = \mathbf{min}(f_a, f_b). \tag{2.2}$$

The intersection between A and B by

$$\mathbf{A} \cap \mathbf{B} = \mathbf{max}(f_a, f_b). \tag{2.3}$$

Finally, the difference between A and B is defined by

$$\mathbf{A} \setminus \mathbf{B} = \mathbf{max}(f_a, -f_b). \tag{2.4}$$

Using this method, a CSG object can be defined by an implicit function $CSG_t(X, Y, Z)$ substituting the ray equation

$$\vec{p}(t) = \vec{o} + t\vec{d} \tag{2.5}$$

on the function $CSG(x, y, z)$, and a given ray interval $\bar{\underline{t}} = [\underline{t}, \bar{t}]$ can be classified as *in* if $0 \in CSG_t(\bar{\underline{t}})$, making this a viable and efficient solution to render CSG scenes in real-time.

More recently, Mostajabodaveh et al. [10] demonstrated an algorithm based on Roth's algorithm [17] that we described before, in the sense that it stores all entry and exit points in sorted order for each primitive that was intersected by the ray, forming the ray intervals that represent the ranges where a ray is inside or outside of a primitive. These ray segments are then merged according to the Boolean operation involved and the resulting intervals represent the entry and exit points of the composite object. Although using the Roth's algorithm to determine the resulting ray intervals of the composite object, this algorithm presented by Mostajabodaveh et al. excels by the efficient ray-intersections calculations.

The algorithm uses the Bounding Volume Hierarchy (BVH) [8] acceleration structure and an optimized hit-point calculation method that consists in organizing the CSG model into a number of layers $L_i(i = 1, ..., l)$. Each layer is composed of a number of positive solids $P_{i,j}(j = 1, ..., p_i)$ and negative solids $N_{i,k}(k = 1, ..., n_i)$. By using this method, the CSG operations for a scene S can be described as

$$S = \sum_{i=1}^{l} L_i, \text{ with } L_i = \sum_{j=1}^{p_i} P_{i,j} - \sum_{k=1}^{n_i} N_{i,k} \tag{2.6}$$

where $+$ and $-$ represent union and difference set operations, respectively. Essentially, a layer can be seen as the difference between two compound objects (positive and negative objects), and to ray-trace a single layer they basically track when the ray runs within a positive or negative medium. They use two counters ($posDepth$ and $negDepth$) that are attached as parameters of each ray, and each time a ray enters or leaves a positive or negative primitive, the counters are increased or decreased, respectively. If the ray is inside a positive medium but not inside a negative medium, then the layer's correct hit-point has been found. Otherwise ray traversal continues.

With this method, the final global hit-point of a CSG scene can be found by sequentially testing the primary ray against all layers and choosing the nearest hit-point from the set of layer hit-points.

By using this ray-tracing approach, the authors claim to render massive CSG scenes with a decent frame rate, which allows to interact with the scene in real-time.

2.1.2 Rasterization Techniques

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 photorealistic 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. [3] present a method to directly render CSG objects using a Z-Buffer rasterization approach. In their method, the CSG tree is first converted to its normalized form to enable fast combinatorial logic and then geometric pruning is applied on the tree to remove many primitive leaf nodes that might be generated during the normalization stage.

The process of normalizing a CSG tree consists in rewriting the Boolean expression that the tree represents into a sum-of-products form, by using the eight basic set equivalences, as shown in Figure 2.4. If the Boolean expression in each internal node matches the left side of the set equivalence, then it is replaced by the right side of the set equivalence. Figure 2.5 illustrates an example of a CSG tree converted to its normalized form.

Because the normalization process can add many subtrees with primitives that might not be intersected, performing geometric pruning on the normalized CSG tree to remove those subtrees that will not contribute to the final image is an important operation to reduce the time spent rendering the CSG object. In their algorithm, they use bounding-box pruning to find the primitives that will not be intersected.

Using Goldfeather's algorithm, a CSG object can be rendered directly into a frame buffer containing sufficient memory to store two color values, two depth values and three one-bit flags.

To render a convex object with this algorithm, each product has to be broken into separate terms. Then each term is rendered into the $(zTemp, cTemp)$ image buffer and finally the terms are merged into the $(zFinal, cFinal)$ image buffer, using a standard Z-Buffer algorithm.

To render a k-convex primitive, the boundary surface of the primitive should be divided into front-facing and back-facing sub-surfaces. Then, in the front and back routines, the polygons of the k-

1. $X - (Y \cup Z) = (X - Y) - Z$
2. $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$
3. $X - (Y \cap Z) = (X - Y) \cup (X - Z)$
4. $X \cap (Y \cap Z) = (X \cap Y) \cup Z$
5. $X - (Y - Z) = (X - Y) \cup (X \cap Z)$
6. $X \cap (Y - Z) = (X \cap Y) - Z$
7. $(X \cup Y) - Z = (X - Z) \cup (Y - Z)$
8. $(X \cup Y) \cap Z = (X \cap Z) \cup (Y \cap Z)$

Figure 2.4: Set equivalences for normalization (Source: "Near real-time CSG rendering using tree normalization and geometric pruning"[3]).

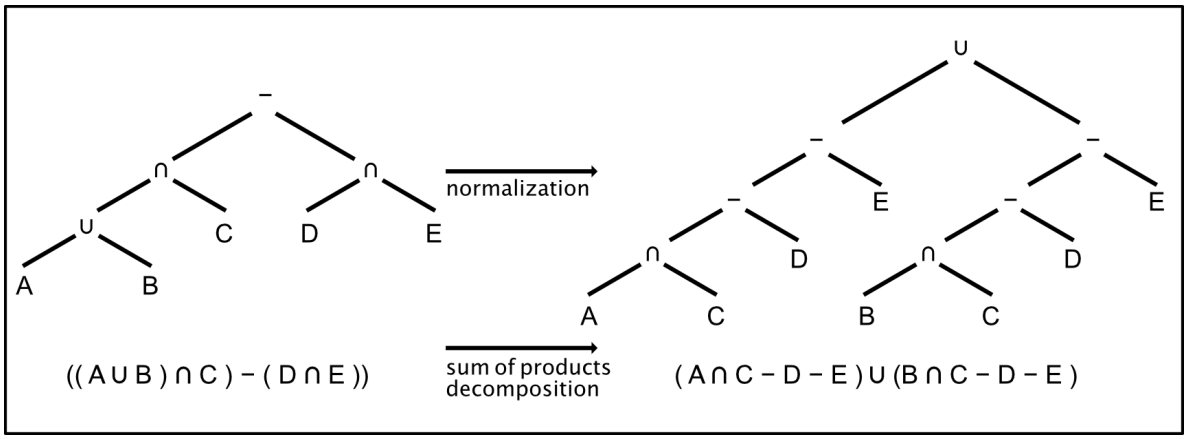


Figure 2.5: CSG Tree before and after normalization (Source: "Near real-time CSG rendering using tree normalization and geometric pruning"[3]).

convex primitive are sent to the frame buffer k times. In each step, a portion of the primitive's surface is retained, and *in*/*out* classification is performed with respect to the other primitives in the product terms. Finally, the portions of the primitive's surface that remain, are merged into the final image. A solid is k-convex if a ray intersecting the solid can enter and exit the object at most k times.

By using Goldfeather's algorithm is possible to render convex and non-convex CSG objects with a fixed number of bits per pixel with $O(n^2)$ complexity, for a CSG object with n primitives.

A different approach to represent CSG trees and Boolean expressions was presented by Rossignac [15], where the author introduces the concept of a Boolean List (Blist) formulation of CSG trees and provides an efficient algorithm to convert CSG expressions into Blists. The Blist method represents a CSG tree as a list of primitives that can be evaluated in a pipeline fashion. At each step, the result of classifying the cell against the current primitive is merged with the result of previous classifications. Converting a CSG tree to the Blist form can be done by first transforming the tree into its positive form and then rotating the tree by switching the left and right children at each node, making the tree left heavy. Finally, the leaves of the tree should be visited from left to right, and the Blist table that represents each CSG expression should be filled according with the information of each leaf.

Evaluating a CSG tree in the Blist representation can be done efficiently by updating a label when its value matches the primitive's name. The label is attached to each cell and it is passed to the successive primitives in the Blist during set membership classification, and it requires at most $log(H+1)$ bits, where H is the height of the tree. This shows how this representation reduces the storage needed to evaluate large collections of cells in parallel architectures, as the ones we can find in modern GPUs.

The Blister algorithm was a GPU-based renderer of Boolean combinations introduced by Hable and Rossignac [4] that used the Blist approach to represent CSG expressions. The name Blister stands for Blist-Expression Renderer, and this implementation works by combining depth-peeling with the Blist formulation of Boolean expressions. Using this algorithm, which has a complexity time of $O(kn)$, one can render an arbitrary CSG model of n primitives in at most k steps, where k is the number of depth-layers in the arrangement of the primitives.

To render a CSG object, the Blister processes one depth-layer at a time, obtained by using a variation of the Depth Interval Buffer. It peels the entire arrangement of primitives in depth-order. Each peel is classified according to the CSG expression and then is combined. This classification is possible with the Blist formulation to represent CSG expressions. The peeling process stops when all pixels are behind a successfully classified opaque surface element or outside of a pre-computed superset of the CSG solid.

Despite the classification of surface elements being correct, in some cases the Blister implementation yields incorrect values of color for some pixels. This may happen if two surface elements that project onto the same pixel have the same quantized depth, because both surfels are tested against the same Blist expression. Consequently, there may exist two surfels with different colors or normals, and the Blister will use the first that encounters, hence the incorrect values.

An optimized approach to render CSG scenes using the Blist formulation was later introduced by Hable and Rossignac [5]. This new CSG rendering algorithm, also GPU-based, has the name of Constructive Solid Trimming (CST). The CST algorithm is faster than the Blister approach and eliminates the problem of incorrect values of color in some pixels.

The CST method uses the concept of active-zones that was first introduced by Rossignac and Voelcker [16], where they show that for depth-order classification, the relevant primitives in the CSG tree are the ones that are part of subtrees that form an intersection combination with the primitives in the positive form of the CSG tree. Using the CST method, the boundary of each primitive is peeled and trimmed, one at a time, against the Blist of its active-zone only, instead of trimming the boundary against the whole CSG expression.

In this method, the active-zone of each primitive is derived from the recursive traversal of the CSG tree, where the active-zone is defined by the intersection of the universal set with the intersection of i-nodes, minus the union of u-nodes. Then, it peels each primitive while trimming the primitive against its active-zone and merges the results into a global Z-Buffer, to later select the points according to the depth-order classification.

The authors also present some methods to optimize the CST rendering. The CST rendering can be faster for situations where the CSG object is opaque, because one could only peel and trim the front faces of positive primitives and the back faces of negative primitives. Also, for these situations where opaque CSG models are being rendered, there is no need to trim the boundary of the primitives against the u-nodes, it is only necessary to trim against the i-nodes. This reduces the total time spent trimming the primitives. Finally, leaf nodes that share the same Boolean operator should be grouped, because instead of trimming the layer of pixels against a single primitive, this can be done faster by trimming against a subtree of primitives that share the same parent operator.

The CST rendering method is a good example of how CSG solids can be rendered using the Blist approach to represent Boolean expressions, without having wrongly colored pixels as the ones that the Blister algorithm may produce.

2.2 Optimizing CSG Trees

Complex CSG scenes with a high number of solid objects and Boolean operations between them may produce CSG trees that take too long to traverse. Those trees can be highly unbalanced and may contain nodes that will be irrelevant in the tree traversal.

To reduce the time spent traversing the trees, several approaches can be taken to transform complex trees in easier trees to process, which will decrease the total time needed to render the CSG scene associated.

A method to obtain a better tree layout for GPU ray-tracing was presented by Ulianov et al. [19], where they propose an efficient pipeline for optimizing CSG trees that can be divided in four distinct phases:

1. **Converting the tree to the positive form**

A tree in the positive form is a tree that contains only union and intersection operations. Converting a tree to the positive form can be done by rewriting Boolean expressions containing difference operators, replacing the difference operations by intersection operations and complementing their right subtree, according to Equation 2.7. Finally, de Morgan's laws should be recursively applied to the complemented trees, in a pre-order traversal to propagate the complements to the leaf nodes.

$$A - B = A \cap \overline{B} \tag{2.7}$$

2. **Spatial optimization of tree topology**

   In this step of the optimization, they try to restructure the tree by repeatedly selecting *treelets* consisting of nodes with the same Boolean operation. They define a *treelet* as the collection of immediate descendants of a given CSG tree node.

3. **Minimizing height of tree**

   The principal goal of this stage of the pipeline is to end with a well-balanced CSG tree, by minimizing the height of the tree using local transformations.

4. **Reverse converting to a general tree form**

   At this stage, the resulting tree is converted to its general tree form.

Applying these actions on CSG trees will result in equivalent trees with a better topology, making them easier and faster to traverse in posterior phases of the CSG ray tracing pipeline, and could be an interesting approach to take in consideration in our solution.

Other methods to optimize the CSG trees were presented by Jansen [7]. Those methods consist of pruning the tree, by removing subtrees that are classified as *out*. If one of the leafnodes of the tree can be classified as *in* or *out*, then its parent node can be replaced by the other child or by an *in* or *out* classification. Following the rules of the Table 2.1 [21] it is possible to end with an equivalent CSG tree, but containing fewer nodes.

Next, Jansen introduces two different contexts in which tree pruning is often applied: spatial locality and structural locality.

In the context of spatial locality, a regular space is subdivided in cells, and a given cell can be intersected by one of the primitives in the CSG scene or not. If there is no intersection between a cell and a primitive, then that primitive is classified as *out* in the CSG tree. On the other hand, a primitive is classified as *in* if it is totally overlapped by a cell. Repeating this process for all the primitives in the scene will result in a CSG tree with nodes classified as *out* that can be removed.

Structural locality within the context of tree pruning is a technique to determine which subtrees could be removed, by taking into consideration the structural relations in the tree, or in other words, the Boolean operations involved between primitives. For example, if two primitives are combined with a union operation, then the depth-order classification of points in one primitive is independent of the

Table 2.1: Rules for CSG Tree Pruning.

OPERATION	LEFT	RIGHT	RESULT
Union	In	Tree	In
	Out	Tree	Tree
	Tree	In	In
	Tree	Out	Tree
Difference	In	Tree	Not Tree
	Out	Tree	Out
	Tree	In	Out
	Tree	Out	Tree
Intersection	In	Tree	Tree
	Out	Tree	Out
	Tree	In	Tree
	Tree	Out	Tree

classification of the other primitive involved, because the first primitive intersected will be classified as *in* independently of the position of the other primitive. This idea explores the concept of active-zones previously mentioned in this document.

To perform tree pruning using this method, the CSG tree must be first transformed into its positive form, and then reordered as the intersection of primitives with a subtree defining its Intersection-zone (I-zone), combined with a subtree contained in its Union-zone (U-zone). The removal of nodes using this method is possible because only the intersection of a primitive and the subtree defining its I-zone is relevant in depth-order classification. Finally, the tree can be pruned considering spatial locality.

Exploring the spatial and structural locality of a CSG tree, allow us to remove irrelevant tree nodes and end with a simpler CSG tree that is faster to traverse.

2.3 Data Structures on the GPU

In OpenCL the host program performs the configuration of kernel programs on the computing devices. The host is the main CPU used to manage kernel execution and the device is the component which contains the processing units that will execute the kernel programs. This will be the GPU, in our case. GPUs do not have access to the memory of the host but OpenCL offer mechanisms to transfer the data between the host system and the device.

Once the data is available on the GPU, the use of good concurrent data structures would be an important step to facilitate the process of parallelizing the data along the multiple work-items available on the GPU.

2.3.1 Semi-Linked Lists

Research in the implementation of dynamic data structures on the GPU has been done by Mulder [11]. They used the Kernel Memory Allocator (KMA) [18] in their solution to reuse memory that was no longer needed and implemented a Semi-Linked list using OpenCL.

In their solution, the nodes in the list contained a key and a pointer to the next node. The end of

the list was marked with $NULL$ and they were able to perform *add*, *get* and *delete* operations on the list. The concurrency on the list was guaranteed by using a compare-and-swap (CAS) instruction to synchronize the changes on the list.

Using the KMA to allocate memory on the GPU and implement dynamic data structures in OpenCL could lead to better memory management and less space used by our solution. Despite the memory improvements, the use of dynamic data structures could cause an overhead in time performance, because the data has to be transferred from the host to the device, and then processed in order to create the desired dynamic data structure with the KMA. This may be more time consuming than using the data directly as it came from the host.

2.3.2 Linked Lists

Research in the implementation of concurrent linked lists on the GPU has also been done by Yang et al [22]. In their solution, they use two memory buffers to construct the concurrent linked list. In one buffer, they store all linked list node data while the other buffer stores head pointers that reference the start of the linked lists in the first buffer. They use atomic memory operations to guarantee that two different working-items do not access the same data, at the same time.

In this solution, the two memory buffers have to be created with enough size to store all the possible nodes, or verifications to ensure that overflow does not occur should be implemented.

This method to create linked lists on the GPU is worth to consider in our solution, as we can determine the maximum number of ray partitions to be stored in the memory buffer.

Using this data structure implementation may lead to unused GPU memory, since the GPU memory has to be allocated before it can be used, which is a disadvantage of this method. Despite that limitation, this solution allow us to create a linked list on the GPU in a similar way to CPU implementations and it is easier to implement than using the KMA.

2.4 General-Purpose Computing On Graphics Processing Units

General-Purpose Computing on Graphics Processing Units (GPGPU) is a term that refers to the use of a GPU to perform non-specialized calculations that would typically be handled by the CPU. The term has grown in popularity since it was first introduced, around 2001, responding to the necessity of computing parallelized tasks in a faster way.

Until recently, the usage of a GPU was limited to just computer graphics, but considering that the GPU has a vastly number of small processing units that operate in different data simultaneously, people start realizing the advantages of using the computing power of the GPU to perform many parallel processing tasks faster than they would be if performed on the CPU.

The process of ray-tracing a CSG scene is a highly parallelized task, since the rays can be processed independently. Therefore, there is only so much to gain by taking advantage of the parallelism of the GPU to render our CSG scenes with a ray-tracing algorithm.

Two popular GPGPU frameworks are NVIDIA's CUDA and OpenCL. Since the former is a proprietary framework that supports only NVIDIA hardware, we will present in more detail the OpenCL framework as it is the framework that we intend to use in our work.

2.4.1 OpenCL

OpenCL™ is an open royalty-free standard for general purpose parallel programming [12] designed to leverage CPUs, GPUs and other processors in order to accelerate parallel computations. This leads to great speedups in computationally intensive applications. Being a non-proprietary and multiplatform framework, it also has the advantage to allow OpenCL code to be portable across different devices and architectures.

We can use an hierarchy of models, which contains the Platform Model, the Execution Model, the Memory Model and the Programming Model to describe the architecture used in OpenCL.

The Platform Model consists of a host that is connected to one or more OpenCL devices. In our case, the device will be the GPU, but it could be a multi core CPU or other device that could provide processing power for OpenCL. The host is the CPU in which the application is being executed and OpenCL commands are submitted from the host to execute computations on the device's processing elements.

The execution of an OpenCL program occurs in two parts: the host program defines the kernels and manages their execution, and then those kernels are executed on the OpenCL device. Kernels are the basic unit of executable code and they can be data parallel or task parallel.

The Execution Model is defined by how the kernels execute on the device. Kernels are executed across a global domain of work-items, which is the term to reference a kernel instance and each work-item is executed in parallel. The global domain is a N-Dimensional domain, where N can be one, two or three. This global domain can be further grouped into local work-groups. Work-items from each local work-group are executed together on the device for synchronization and to allow the local memory to be shared, but there is no synchronization between global work-items, only within local work-groups.

To coordinate the execution of the kernels on the devices, the host creates a command-queue where commands are scheduled to be executed in-order or out-of-order.

Figure 2.6 illustrates how the Memory Model in OpenCL works for a simple example where the host is connected to a single OpenCL device. The host and the device memory models are independent of each other. The host has its own memory, which is the memory that it is on the CPU and that is accessible by the CPU. The device also has its global memory and constant memory. Although the memory models being independent, they can interact and data can be transferred between host and device through OpenCL API functions.

Each work-item executing a kernel has access to four distinct memory regions of the device memory. Firstly, work-items can access the *Global Memory*, which is a memory region that permits read-/write access to all work-items in all work-groups. Secondly, there is a region of the global memory that

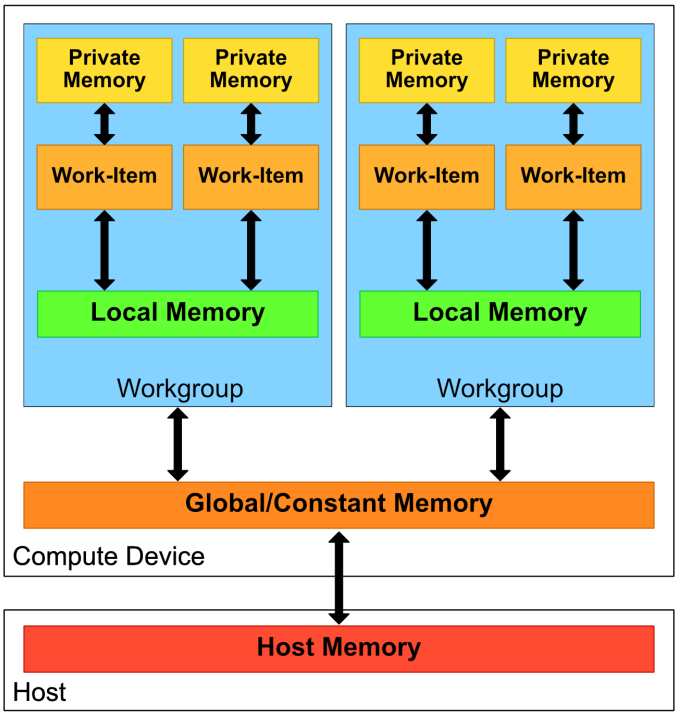


Figure 2.6: OpenCL™ Memory Model.

remains constant during the execution of a kernel, and it is named *Constant Memory*. Then, we have the *Local Memory* that represents a memory region local to each work-group, and this local memory is shared by all the work-items of that work-group. Finally, each work-item has a *Private Memory*, and any variable defined inside this private memory region is not visible to the other work-items.

To transfer data between the host and the device we have to use OpenCL memory objects that can be of two different types: memory buffer objects or image objects. A memory buffer object is basically a block of sequential memory while an image object can be a 2D or 3D image. The kernels receive memory objects as input and they can write the output for the memory object, considering that the memory object can be accessed to read and write operations.

In our solution, we will only use buffer objects to transfer data between the host and the device.

2.5 Summary

In this chapter we presented several algorithms to render CSG models, including ray-tracing algorithms and rasterization techniques. Since we intend to render CSG with ray-tracing on the GPU, we will mostly consider the ray-tracing solutions that we have mentioned early. Although the rasterization methods not contributing directly to our solution, their study was very important to better understand the different methods and concepts involved in the rendering process of CSG objects.

From the ray-tracing solutions, we will focus mostly in Roth's work, since our solution will be processing the ray intervals in a similar way. However, we will be implementing the solution on the

Table 2.2: Pros and Cons of the Semi-Linked List and Linked List data structures implementations on the GPU.

	PROS	CONS
Semi Linked List	- The data can be managed in a dynamic way - All the memory allocated is used - Memory is reused	- To create the list, the data has to be first stored in the GPU - Setup process decreases the time efficiency - More complex to implement
Linked List	- Easier to implement - Faster to process	- Lists have to be created with fixed size - Requires two buffers - Memory that is no longer needed cannot be released

GPU, and taking advantage of the BVH acceleration structure to compute intersections in an optimized way.

Our solution will process the ray intersections in a similar way to Roth's algorithm [17], in the sense that we will be forming ray intervals from all the intersection points along the rays. For each ray, we will sort the hit points, and merge the ray segments together into ray partitions, in a procedure that we define as *weaving* of segments. The ray partitions will be later evaluated against the CSG tree to determine the nearest partition entry point of the ray.

We have also presented some methods to optimize the traversal of CSG trees, as it can reduce considerably the time to render the CSG scene.

Our CSG tree traversal implementation will take advantage of the structural relations in the tree, i.e the Boolean operations involved between primitives. Structural locality within the context of tree pruning is mentioned in Jansen's work, and we will consider this method to stop the CSG tree evaluation earlier and avoid processing subtrees that will not influence the outcome of Boolean evaluation.

We also optimize the CSG tree with the objective of reducing the memory used per thread on the GPU, ending with a compact linear tree representation that can be traversed without recursion on the GPU. The details of the implementation can be see in the next chapter of the document.

To store data efficiently on the GPU, we will be implementing a doubly-linked list, in a method that has some similarities with the linked list solution presented by Yang et al., since it has some advantages in relation to the semi-linked list presented by Mulder (see Table 2.2). However, we will only use one memory buffer to implement the list data structure, as we can offset the elements in the buffer using the unique kernel *ID*, which also guarantees that different kernels do not access the same locations in memory.

3

Solution

Contents

In this chapter we will present our solution to perform Boolean evaluation in parallel on the GPU to render CSG objects using ray-tracing. We start by describing our algorithm, the requirements and the tools that we will use. Then, we show in a detailed manner the key aspects of our implementation.

3.1 Overview

Our solution will use a multi-hit ray traversal algorithm to allow us to find all the intersection points along a given a ray. Multi-hit ray traversal is a class of ray traversal algorithms that finds all the intersection points along the ray, and not only the nearest intersection point to the ray origin, as first-hit ray traversal algorithms do.

Given the list with all the intersections points between the rays and the solid objects in the scene, we will start by merging the intersection segments into ray partitions, in sorted order. Then, we will evaluate each partition against all the CSG trees involved with that specific partition, using simply Boolean algebra. The partitions evaluated with value $true$ are the partitions that represent sections of the compound object. The closest partitions to the ray origin from these partitions is the partition that will be shaded, according to the material properties of the object.

The algorithm will receive as input the list with all the intersection segments between the rays and the objects in the CSG scene and a list with all the regions involved. It will return the evaluated partitions and for each ray, it will shade the closest evaluated partition to the ray origin, if there is any. Otherwise, the pixel will be colored with the defined background color.

In the context of our work, we define the following terms as:

- **Segments:** A segment is a line segment that represent *in* and *out* intersection points, and is derived by performing intersection calculations between a ray and a single object. A ray may produce multiple segments when intersected with an object, depending on the shape of that object. The torus, as seen in Figure 3.1, is an example of a primitive that may produce two ray intersection intervals. An intersection segment will contain information about both endpoints of the intersection, and the *id* of the object that originated the segment.

- **Partitions:** A partition is a structure that contains information about intervals of the ray that pass through geometry and it may be formed by one or more segments that are combined in a stage of our algorithm. A ray partition differs from a segment in the sense that it represents sections of the compound object and not sections of a single primitive that is part of the compound object.

- **Regions:** A region is a data structure that contains the information about the Boolean operations involved between the primitives within the region and it also contains information about how the objects of the region should be shaded. Therefore, each region contains a CSG tree, the material properties of the region and auxiliary variables to optimize the Boolean evaluation. For example, it helps to know if a region contains only union operations or not, since we could skip the CSG tree traversal in those cases.

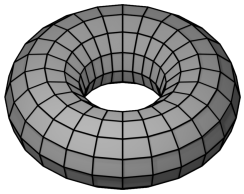


Figure 3.1: Torus.

- **Regiontable:** The regiontable is a structure that contains the list with all the regions involved in one partition. If the primitive *id* of the segments in the partition is also present in the region CSG tree, then we consider the region to be involved in the partition. Each partition has a regiontable associated, and this regiontable is built for each partition just before the start of the Boolean evaluation procedure. Once the regiontable has been created, we can start to evaluate the partition against each one of the regions present in the regiontable.

- **Overlaps:** Because a partition is evaluated against one or more regions, depending on the number of regions inside the partition's regiontable, it may occur situations where two or more regions claim the partition. We define this as an overlap.

  In the shading process, only one region can claim the partition. Otherwise, we would not know which region's material we would have to use to calculate the color of the pixel. Therefore, it is important to check for overlaps after evaluating a partition, and resolve any occurrence of overlaps before the shading stage.

In Algorithm 1, we present a top level description of our algorithm, and we will explain the implementation of each step in Section 3.3 of this document.

```
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.

It is important to highlight that every calculation in the algorithm will be performed in double-precision, i.e 64-bit floating point values, as it leads to more accurate results.

3.2 Tools

3.2.1 OpenCL

We will use OpenCL to parallelize our algorithm, more specifically, the OpenCL 1.2 version.

As mentioned before in this document, OpenCL is a non-proprietary framework. As we intend to contribute to an open-source project with our solution, we are required to use tools under the same license and distribution. Besides that, OpenCL is a multi-platform framework, which means that we will be able to run our OpenCL code on the GPU, and on the CPU. This will allow us to compare the efficiency of our algorithm in these different devices.

To run the OpenCL program on the device, the computer must have the OpenCL Software Development Kit (SDK) of the device manufacturer installed. It is our intention to use Intel and AMD CPUs and different NVIDIA GPUs to evaluate our solution. Therefore, we will have to use the Intel, AMD and NVIDIA OpenCL SDK to run our solution.

3.2.2 BRL-CAD

Our parallel solution to render CSG scenes with ray-tracing will be implemented in the BRL-CAD [13] application.

BRL-CAD is an open-source solid modeling application that was originally developed by the US Army Ballistic Research Laboratory, in 1979. It has been the primary CAD solid modeling system used by the US Military to model weapons systems and to perform lethality analyses. The BRL-CAD solid modeling system is often used in a wide range of military, academic and industrial applications, and it became an open-source project on 21 December 2004, when it started receiving contributions from developers across the world, who helped the tool to maintain its quality and to get its new features.

The BRL-CAD ray-tracer was credited as the first implementation of a real-time ray-tracer at the 2005 SIGGRAPH computer graphics conference, since its ray-tracing systems were able to achieve several frames per second in rendering performance.

By using BRL-CAD, we had access to a modeling environment that allowed us to create simple CSG scenes, as well as using some more robust test scenes that were available with the application. This was fundamental to help us debug and test our algorithm during the development phase.

Besides that, the BRL-CAD ray-tracing library, named *LIBRT*, already offers the common ray-tracing components, as ray generation, ray traversal, routines to check for collision between rays and solid objects and methods to perform shading, which allowed us to focus on the parallel Boolean evaluation system. It is also important to mention that the BRL-CAD tool already offers support for GPU ray-tracing. This means that the tool has code to shoot rays, ray-intersection methods for most of the primitives supported by the application and code to shade the intersected objects implemented in OpenCL.

With our contribution, the BRL-CAD application will have a complete CSG ray-tracer implemented in OpenCL. Therefore, a ray-tracer that can be executed in different kinds of hardware, as CPUs and/or GPUs.

BRL-CAD is freely available for the public and is distributed in both binary and source code form through the BRL-CAD project website on Sourceforge [1].

As previously mentioned, we have contributed to the BRL-CAD open-source project with our algorithm to render CSG models with OpenCL, under the Google Summer of Code (GSoC) 2017 project.

As the name refers, Google Summer of code, often abbreviated to GSoC is a project hosted by Google every summer since its inception in 2005.

The principal objective of GSoC is to introduce students from the entire world to open-source development, while helping the organizations to have new contributions and offering to students an opportunity for them to apply their skills in real-world projects.

Being a participant of GSoC17 under the BRL-CAD organization, and finishing the program with success, helps to validate the usefulness of our work, that was accepted and it is already fully integrated in BRL-CAD release branch, offering to users of the application an alternative to render CSG objects with ray-tracing using OpenCL.

The source code of our implementation is available for free at the BRL-CAD project website on Sourceforge [1] and it may be used for any purpose or furthermore improved, perhaps with the improving guidelines that we present in the last chapter of this document.

3.3 Implementation

Our OpenCL ray-tracer uses a BVH [14] acceleration structure to optimize the intersection between rays and the primitives in the scene and every operation in our algorithm is calculated using double-precision. With the use of this object-partitioning acceleration structure, we compute ray-intersections only once, which is not guaranteed with the use of a spatial partitioning structure like a KD-tree without using a technique such as mailboxes. By using mailboxes, a substantial amount of GPU memory would be needed to avoid intersecting a primitive more than once, hence the initial choice of a BVH structure to reduce the memory per thread.

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: counting hits, storing segments, weaving segments, evaluating partitions and shading partitions.

We use a global dimension with the size of the image we want to produce, divided in local work-groups of 64 working-items. For example, to produce a render with resolution of $1024x1024$ pixels, we will have one kernel executing for each pixel of the image. We employ the analogy of a kernel instance corresponding to a ray. Therefore, each working-item will represent an unique ray.

3.3.1 Kernels

In our solution we use a total of five OpenCL kernels to perform each stage of the algorithm, having the following rendering loop:

1. **Count Hits Kernel:** Kernel to determine the total number of intersection points. Knowing the

number of intersections between rays and objects in the scene is important to allocate the memory on the OpenCL device, in a way that minimizes the waste of resources.

2. **Store Segments Kernel:** In this kernel, the segments from ray-primitive intersections are stored in one OpenCL memory buffer allocated a priori, with its size being half the number of total hits calculated in the previous kernel.

3. **Weave of Segments Kernel:** This kernel processes the list of all segments for each ray, combining those segments into ray partitions, in sorted order. The created partitions are stored in an OpenCL memory buffer, as well as the index to the first partition of each ray.

4. **Evaluate Partitions Kernel:** Kernel that iterates over the partitions of the ray, and evaluates each partition against all the regions involved with that specific partition. In this kernel, the occurrence of overlaps is verified, and the kernel stops execution when a given partition is evaluated, or when all the partitions in the ray have been processed.

5. **Shade Partitions Kernel:** This kernel shades the closest partition evaluated to the origin of the ray, if there is any. Otherwise, the color of the pixel is set to the background color. The shading of partitions is performed according to the normal of the entry point of the partition, and also according to the material of the region associated with the evaluated partition.

The code of the kernel to count the hits and the code of the kernel to store the segments, is in fact, very similar. Both kernels perform ray-primitive intersection calculations, the only difference is that the first kernel only counts the number of hits, while the second stores the segments in the OpenCL buffer.

Having to perform intersection calculations twice for each scene is a disadvantage, as it is a time consuming task, but since dynamic allocation of memory on the device is not possible in OpenCL, all the OpenCL memory buffers must be allocated before they can be used by kernels. Here we are basically benefiting the memory of the device over the time efficiency of the algorithm, as the GPU memory available for allocation can be very limited.

An alternative to not having to intersect all the primitives in the scene twice, would be to estimate the maximum number of intersections that a scene could have, basing this number on the total number of primitives and on the type of objects in the scene. For example, intersecting a cube will always produce two hits, but intersecting a torus could yield two intersection segments, hence four intersection points. Estimating the maximum number of hits between a ray and a mesh may not be trivial, and if we used this method to avoid repeating intersection calculations, we possibly would not have a perfect fit of segments in memory.

Knowing the exact number of intersection points is important because we estimate the total number of necessary partitions based on that number. More precisely, we use the maximum possible number of partitions to allocate the memory buffers, i.e twice the number of segments minus one. Because of this, we end having some wasted memory as we show in the next chapter. If we also had estimated the total number of hits in the scene, the memory wasted would be even bigger.

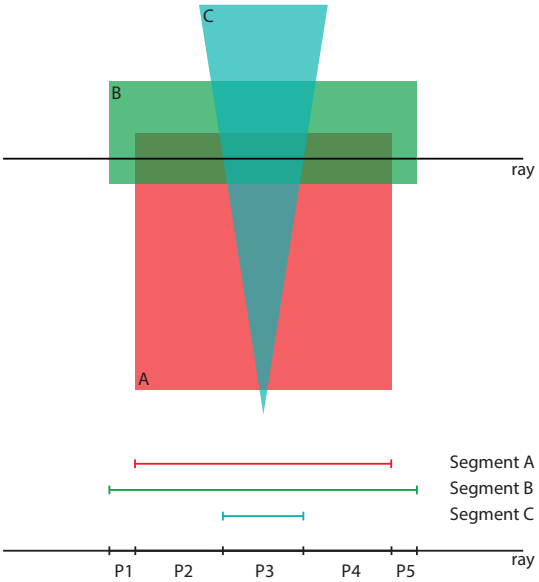


Figure 3.2: Ray-primitive intersection segments.

To minimize the waste of GPU resources, and to guarantee that all the memory allocated would be utilized by the algorithm, we could follow the same approach we used to store the intersection segments, and repeat the kernel that merges the segments into partitions. Firstly to determine the exact number of necessary partitions, and then to actually store the partitions in memory.

3.3.2 Weave of Segments

To weave segments, we iterate over the segments of the ray, 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. In Figure 3.2, we illustrate an example of the partitions that result from weaving the segments of a given ray. The ray intersects the primitives A, B and C, producing a segment for each primitive. The three segments are passed as input to the weave of segments kernel, and are merged according to the segment entry point, forming the five partitions, $P1, P2, ..., P5$.

3.3.3 Doubly-Linked List

To store the partitions in memory during the weave of segments stage, we have implemented a doubly-linked list in OpenCL. First, we allocate the OpenCL buffer in the host program, whose dimensions are twice the number of segments, since this is the maximum possible extent to store all the partitions. Then, in the weave of segments kernel, we use the unique global *ID* of each work-item to offset in the OpenCL buffer and access the correct buffer element. Each ray has its section of the OpenCL memory buffer starting at

```
struct partition {
    struct hit inhit;
    struct hit outhit;
    uint inseg;
    uint outseg;
    uint forw_pp;
    uint back_pp;
    uint region_id;
    char inflip;
    char outflip;
};
```

Figure 3.3: Partition structure.

$$partition\_index = 2 * h[id] \tag{3.1}$$

where the *id* represents the unique ID of the work-item, and $h[id]$ contains the sum of all segments of the previous rays.

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.

The method we used to implement the doubly-linked list data structure is similar to the method presented in the Section 2.3.2 of this document, where Yang et al. [22] use two OpenCL buffers to implement their list. However, we use only one OpenCL memory buffer, by storing the index to the back and forward partition in each element of the buffer. Each element of the buffer contains one partition, and the partition structure is defined as can be seen in Figure 3.3, where the hit structure contains the intersection point and the normal at the intersection point. The $inseg$ and $outseg$ have the indexes of the respective entry and exit segments of the partition. As mentioned above, the $forw\_pp$ and $back\_pp$ represent the indexes to the back and forward partition in the memory buffer, respectively. Lastly, the $region\_id$ has the *id* of the region that lead to the partition being evaluated, while the $inflip$ and $outflip$ are used in the shading process to change the signal of the partition entry point normal, for the cases where an interior surface is being shaded.

Our method has the same limitation of the referenced solution, 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.

Both appending and insertion operations 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. In Figure 3.4 and in Figure 3.5 we present the OpenCL code of the $insertion$ and $appending$ operations on the doubly-linked list, respectively.

In sum, we have a large buffer of memory that we use to create a doubly-linked list for each ray. We illustrate our doubly-linked list representation in Figure 3.6.

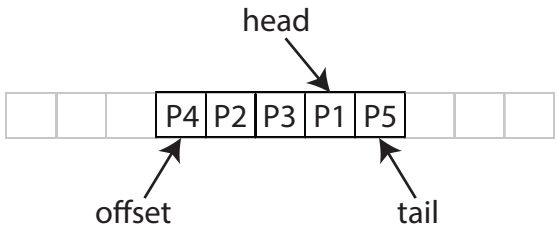
```
inline void
insert_partition_pp(global struct partition *partitions,
    global uint *ipartition, size_t id,
    uint *head, uint new, uint old)
{
    if (ipartition[id] == 0) {
        // No partitions on the list
        return;
    }

    if (*head == old) {
        // Inserting at the head of the list
        partitions[old].back_pp = new;
        partitions[new].back_pp = new;
        partitions[new].forw_pp = old;
        *head = new;
    } else {
        partitions[partitions[old].back_pp].forw_pp = new;
        partitions[new].back_pp = partitions[old].back_pp;
        partitions[new].forw_pp = old;
        partitions[old].back_pp = new;
    }
}
```

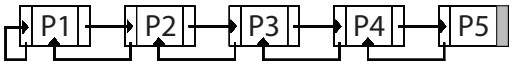
Figure 3.4: Method to insert partitions on the doubly-linked list.

```
inline void
append_partition_pp(global struct partition *partitions,
        global uint *ipartition, size_t id,
    uint new, uint *tail)
{
    if (ipartition[id] == 0) {
        // No partitions on the list
        partitions[new].back_pp = new;
        partitions[new].forw_pp = UINT_MAX;
        *tail = new;
    } else {
        partitions[new].back_pp = *tail;
        partitions[new].forw_pp = UINT_MAX;
        partitions[*tail].forw_pp = new;
        *tail = new;
    }
}
```

Figure 3.5: Method to append partitions on the doubly-linked list.



(a) Section of the memory buffer.



(b) Doubly-Linked List representation.

Figure 3.6: Partitions from the example in Figure 3.2. a) How the partitions are stored in the memory buffer. b) How the partitions are represented with the doubly-linked list.

P1		
C	B	A
0	1	0

P2		
C	B	A
0	1	1

P3		
C	B	A
1	1	1

P4		
C	B	A
0	1	1

P5		
C	B	A
0	1	0

Figure 3.7: Bit arrays (bottom row) of each partition from the example in Figure 3.2.

3.3.4 Dynamic Bit Arrays

In order to know which segments contribute to the partition, we use one dynamic bit array per partition, where we simply set the bits corresponding to the segments that contribute to the partition. Figure 3.7 illustrates the use of bit arrays to represent the segments in each partition, where three bits are used per partition since the ray intersects three primitives producing the segments A, B and C. To implement the bit array, we allocate an OpenCL memory buffer with the dimension based on the number of segments per ray. 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.

3.3.5 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 limita-

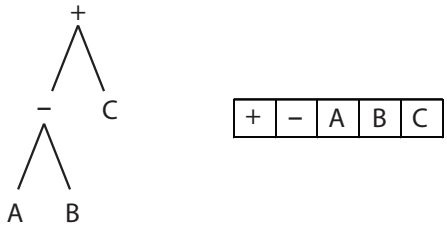


Figure 3.8: Linear CSG tree representation of the CSG object presented in Figure 3.2.

tion 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 (see Figure 3.8). 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.

3.3.6 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 primitive id of the segment 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.

4

Results

Contents

To study the performance of our implementation, we rendered several CSG scenes while recording the time taken to render the image and the resources used by the OpenCL device, and we compared those values with the BRL-CAD algorithm to render CSG. In this chapter, we describe the hardware utilized to run the OpenCL code and the test scenes that we used. We also present the results obtained.

4.1 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 4.1.

Table 4.1: 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 we use a 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

Table 4.2: 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

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.

4.2 Results

4.2.1 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 4.2 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.

4.2.2 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 4.2 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 is respectively observable, in the HAVOC test scene, and in the TANK CAR scene. However, it is

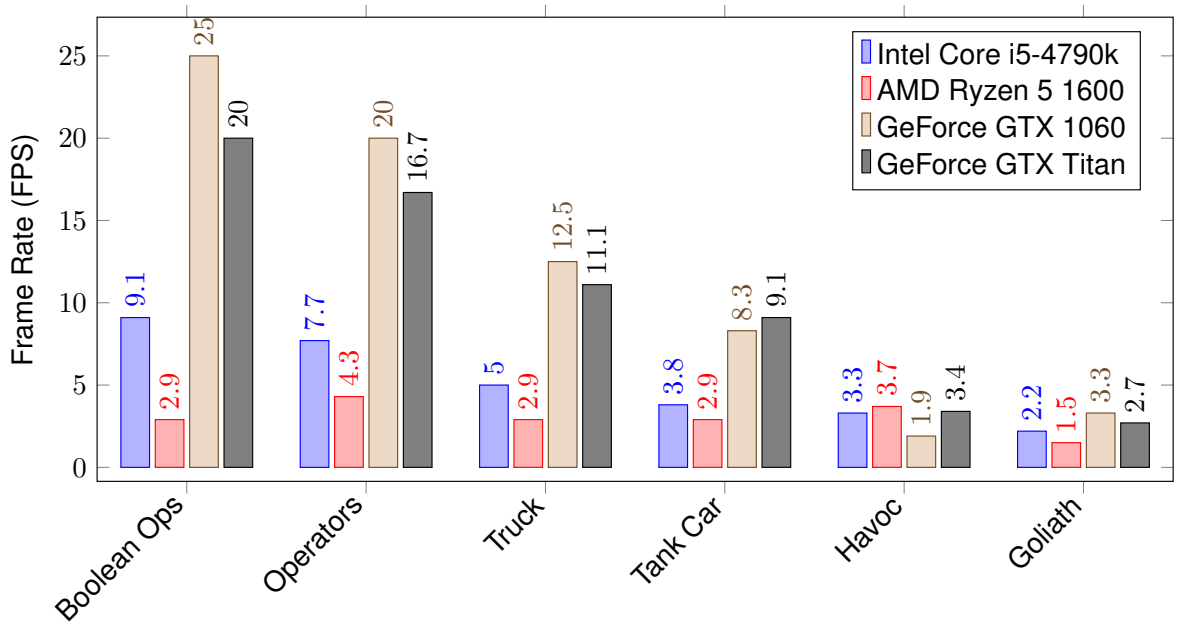


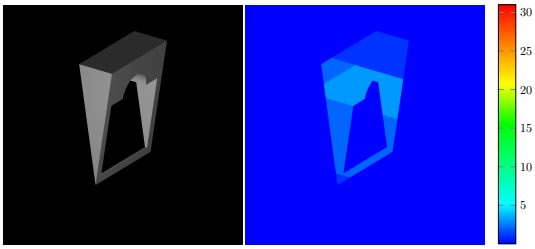
Figure 4.1: Frame rate when ray-tracing each scene on the Intel Core i5-4790k, on the AMD Ryzen 5 1600, on the NVIDIA GeForce GTX 1060 and on the NVIDIA GeForce GTX Titan.

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 BVH, as it is an object partition acceleration structure, where the intersections are not computed in depth order.

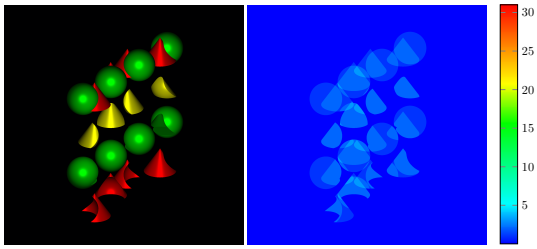
Summarizing, the partial evaluation of hits, facilitated with a spatial partitioning kd-tree, leads to faster performance in scenes with high depth complexity, than an object partitioning BVH. This is particularly evident in the GOLIATH scene which has high depth complexity in a small amount of rays.

In Figure 4.1 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.



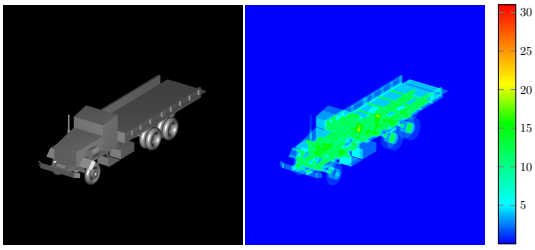
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	NVIDIA SDK
Intel Core i5-4790k	70 ms	110 ms	150 ms	
AMD Ryzen 5 1600	50 ms	340 ms	340 ms	
NVIDIA GeForce GTX 1060				40 ms
NVIDIA GeForce GTX Titan				50 ms

(a) Boolean Ops test results.



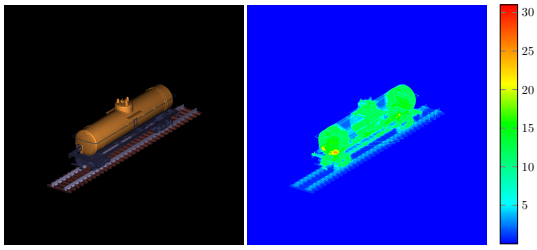
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	NVIDIA SDK
Intel Core i5-4790k	120 ms	130 ms	180 ms	
AMD Ryzen 5 1600	60 ms	280 ms	230 ms	
NVIDIA GeForce GTX 1060				50 ms
NVIDIA GeForce GTX Titan				60 ms

(b) Operators test results.



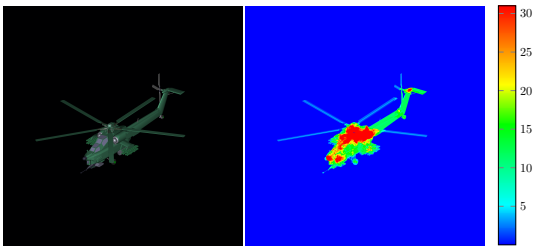
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	NVIDIA SDK
Intel Core i5-4790k	80 ms	200 ms	290 ms	
AMD Ryzen 5 1600	50 ms	350 ms	360 ms	
NVIDIA GeForce GTX 1060				80 ms
NVIDIA GeForce GTX Titan				90 ms

(c) Truck test results.



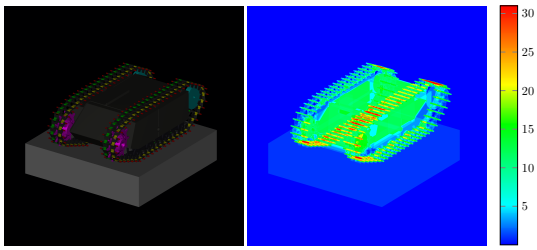
	BRL-CAD	OpenCL		
		Intel SDK	AMD SDK	NVIDIA SDK
Intel Core i5-4790k	280 ms	260 ms	330 ms	
AMD Ryzen 5 1600	160 ms	350 ms	350 ms	
NVIDIA GeForce GTX 1060				120 ms
NVIDIA GeForce GTX Titan				110 ms

(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 4.2: 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.

5

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 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.

Bibliography

[1] SourceForge project website. `https://sourceforge.net/projects/brlcad/`, 2017.

[2] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[3] J. Goldfeather, S. Monar, G. Turk, and H. Fuchs. Near real-time csg rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, 1989.

[4] J. Hable and J. Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1024–1031. ACM, 2005.

[5] J. Hable and J. Rossignac. CST: Constructive Solid Trimming for Rendering BReps and CSG. volume 13. IEEE, 2007.

[6] Y. Hijazi, A. Knoll, M. Schott, A. Kensler, and C. Hansen. Csg operations of arbitrary primitives with interval arithmetic and real-time ray casting. In *Dagstuhl Follow-Ups*, volume 1. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[7] F. W. Jansen. Depth-order point classification techniques for csg display algorithms. *ACM Transactions on Graphics (TOG)*, 10(1):40–70, 1991.

[8] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[9] R. E. Moore. *Methods and applications of interval analysis*. SIAM, 1979.

[10] S. Mostajabodaveh, A. Dietrich, T. Gierlinger, F. Michel, and A. Stork. Csg ray tracing revisited: Interactive rendering of massive models made of non-planar higher order primitives. In *VISIGRAPP (1: GRAPP)*, pages 258–265, 2017.

[11] H. Mulder. Concurrent manipulation of dynamic data structures in opencl. 2015.

[12] A. Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[13] M. J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. 1995.

[14] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.

[15] J. R. Rossignac. Blist: A boolean list formulation of csg trees. Technical report, Georgia Institute of Technology, 1999.

[16] J. R. Rossignac and H. B. Voelcker. Active zones in csg for accelerating boundary evaluation, redundancy elimination, interference detection, and shading algorithms. *ACM Transactions on Graphics (TOG)*, 8(1):51–87, 1988.

[17] S. D. Roth. Ray casting for modeling solids. *Computer graphics and image processing*, 18(2): 109–144, 1982.

[18] R. Spliet, L. Howes, B. R. Gaster, and A. L. Varbanescu. Kma: A dynamic memory manager for opencl. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 9. ACM, 2014.

[19] D. Ulyanov, D. Bogolepov, and V. Turlapov. Spatially efficient tree layout for gpu ray-tracing of constructive solid geometry scenes. 2016.

[20] T. Whitted. An improved illumination model for shaded display. In *ACM Siggraph 2005 Courses*, page 4. ACM, 2005.

[21] J. Woodwark and K. Quinlan. Reducing the effect of complexity on volume model evaluation. *Computer-Aided Design*, 14(2):89–95, 1982.

[22] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.