Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Efficient Ray Tracing of CSG Models

Markéta Karaffová

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

May 2016

Acknowledgements

I would like to express my gratitude to my supervisor doc. Ing. Jiří Bittner, Ph.D. for the continuous guidance and encouragement. I would also like to thank doc. Dr. Alexander Wilkie from Charles University in Prague for the valuable insight into CSG rendering.

Last but not least I would like to thank my parents for supporting my studies. Without them I would not be able to get this far.

Declaration

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 23. května 2016

Abstract

This work explores ray tracing of constructive solid geometry (CSG) and its acceleration in combination with ray tracing triangles. It proposes a way how to exploit Embree, a highly optimized library using bounding volume hierarchy for ray tracing triangle meshes, for rendering CSG with triangle meshes.

Keywords: Ray tracing, CSG, Embree, triangle meshes, BVH

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Tato práce zkoumá metody sledování paprsku v kombinaci s konstruktivní geometrií těles (CSG). Dále navrhuje způsob využití Embree, vysoce optimalizované knihovny používající hierarchii obálek pro sledování paprsků v trojúhelníkových sítích, pro zobrazení CSG v kombinaci s trojúhelníkovými sítěmi.

Klíčová slova: Ray tracing, CSG, Embree, triangle meshes, BVH

Překlad názvu: Efektivní sledování paprsků v CSG modelech

Contents

Figures

Tables

Chapter 1

Introduction

This work explores ray tracing of constructive solid geometry (CSG) and its acceleration in combination with ray tracing triangles. The main idea behind this work is to use a framework optimized for ray tracing triangles and implement a support for CSG trees, which would for example enable "slicing" models and exposing their inner parts.

This work follows master degree thesis of Petr Zajíček [1]. In his work, he considered ray tracing of CSG using acceleration data structures. After evaluating some of the data structures he decided to apply kd-trees for reasons mentioned in section 2.3.1. This work takes a different approach and uses Embree, a highly optimized library using bounding volume hierarchy for ray tracing triangle meshes. I propose a way how to exploit Embree for rendering CSG with triangle meshes.

Chapter 2

Background

This chapter covers the necessary basics related to rendering CSG models. First, it depicts geometry types used in the work, then it explains ray tracing techniques, their accelerations and how they work with constructive solid geometry. Finally, it describes Embree, the library used in this work.

2.1 Solid representations

There are various solid representations, however this work focuses only on two of them: polygon meshes and constructive solid geometry.

2.1.1 Polygon meshes

Polygon meshes belong to the boundary representation group, which delimits some of its characteristics. Each object is defined by a collection of connected surface elements, which corresponds to the boundary of the solid. This results in easy ray intersection tests and displaying surfaces, however it does not offer any information about insides of solids. As a result, this representation is widely used when only surfaces are needed and information about the object interior is unnecessary, in other words, when solids are only observed from the outside as in the real world.

Polygon meshes consist of two parts: geometry and topology. Geometry defines positions of vertices and by its changes polygons change shape, position, size or orientation in space. Topology defines connections between

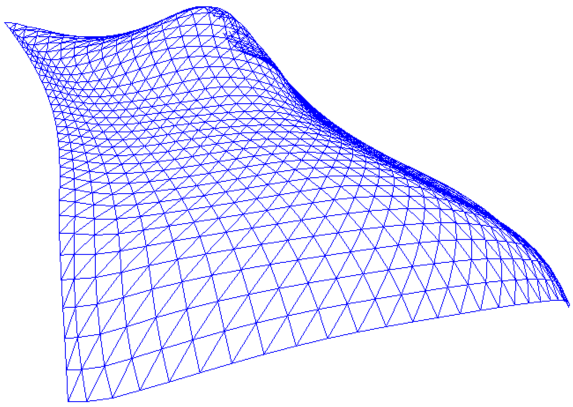

Figure 2.1: Example of a triangle mesh cloth.

vertices. A connection between two vertices is called an edge. Vertices and edges together with faces define polygons. Four or more polygons can form a polyhedron.

Polygon meshes use two-dimensional polygons to create shapes of three-dimensional polyhedral objects. The main idea behind this representation is that every object can be modeled using polygons, but to get the precise shape, a large number of polygons would be required. Reducing the amount of polygons leads to a lower precision, but it is often imperceptible for human eyes.

Furthermore, when rendering polygon meshes, interpolations of normals between vertices can be used to create smooth transitions between polygons and therefore hide sharp edges on round surfaces. Other ways of enhancements are also possible, such as using textures for bump mapping.

The most common type of polygons is a triangle, but quadrilaterals or other simple convex polygons can be also used. Triangles have one advantage over all other polygons: they are defined by three points in space. As it was already mentioned, all points of a polygon have to be on one plane. This holds true for every three points in 3D space, but for polygons with more than three points this rule has to be secured additionally.

The main disadvantage of triangle meshes is the high amount of triangles needed to plausibly describe the scene. However, since triangle meshes are the most popular scene representation, graphics cards are optimized for triangle rasterization. Furthermore, because each scene consists of triangles only, every triangle is processed in the same way, which allows parallel processing.

2.1.2 Constructive Solid Geometry

One of solid representations used primarily for designing objects is called constructive solid geometry (CSG). In this representation all objects are stored in a tree structure, the CSG tree. The tree can have three types of nodes, inner nodes with transformations, inner nodes with set operations and leaf nodes with primitives.

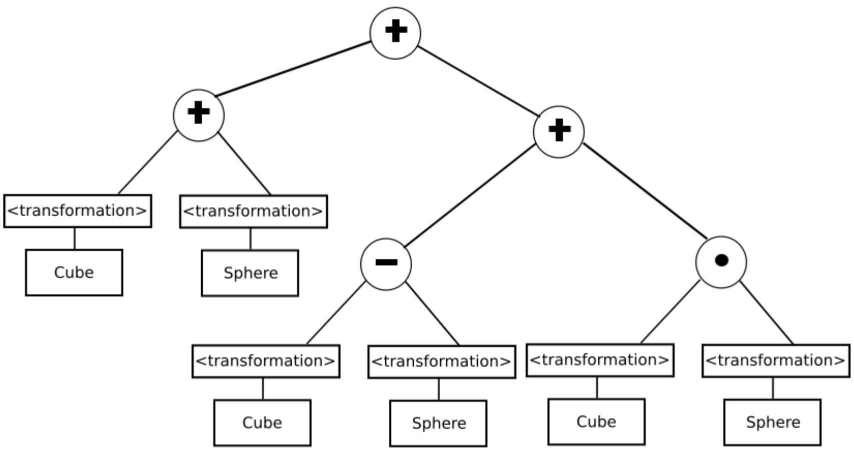

Figure 2.2: Example of a CSG tree

Leaf nodes contain simple solids as cubes, spheres, tori, cylinders and cones, or curves, planes and polygon meshes. Transformations are inner nodes with only one child and apply on the whole branch of their child. The last type of nodes has two children and represents binary set operations, such as union, intersection or difference of two solids. Examples of set operations are shown in figure 2.3.

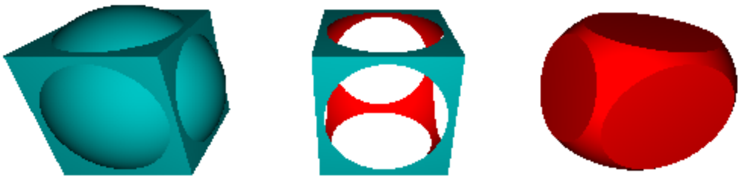

Figure 2.3: Basic operations of a cube and a sphere. From left: Union, difference and intersection

The main advantages of CSG are [1]:

- Testing, if a point is located inside an object, is easy. We simply need to test, if the point is inside every primitive, then traverse the tree and apply set operations.
- In contrast to triangle meshes, there is no need to consider the level of detail. When using triangle meshes, the more triangles are used, the better the final approximation is, but also the longer it takes to process the triangles. For CSG primitives an analytical solution for finding intersections with rays usually exists regardless of the scale used.
- Even complex scenes can be described by a number of primitives relatively small. This is important because the less primitives we have, the less intersections we need to compute.

While this representation offers natural tools for designing objects, rendering objects is not simple since this representation does not store any information about surfaces of solids. For this reason constructive solid geometry is often converted to another representation, for example polygon mesh. However, there are ways to directly render constructive solid geometry and one of them is ray tracing.

2.2 Ray tracing

Ray tracing was first used in 1968 [2] and since then it is quite popular for its high level of visual realism. Ray tracing is a group of techniques

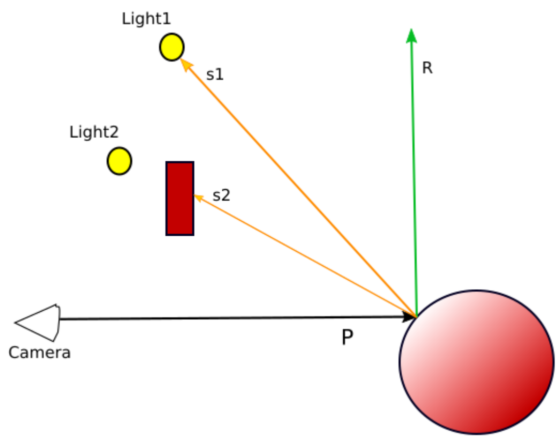

Figure 2.4: Example of ray tracing.

based on geometrical optics. However, the better light simulation they offer, the higher is their computational cost. As a result, they are mainly used when photo-realistic result is needed and computational time is not an issue.

The basic idea is to follow rays of light from light sources as they reflect, transmit and hit objects in the scene. Unfortunately, the overwhelming majority of rays never hit the camera and tracking them only slows down the whole process. This is why the most used way to implement ray tracing is in the reverse order, we cast rays from the point of view into the scene to find out, which objects they hit on the way through the scene.

There are various methods how to perform this task. The simplest way of the realization classical backward ray tracing was introduced by Whitted Turner in 1979 [3]. In this method only the first hit in the scene is found and then a local shading model, such as Phong, is used to compute the local shading. To check light source visibility, shadow rays are casted from hit places towards each of light sources. If the ray hits any object on its way, this object casts a shadow on the place of the hit and therefore the light source is not included in the evaluation of the shading model for the current point.

To get more realistic pictures we can continue further by tracing rays reflected or transmitted. Additionally, we can apply this method recursively and get up to two secondary rays per level of recursion.

An example of ray tracing is shown in figure 2.4. Primary ray P hits the sphere and reflects as secondary ray R. Furthermore, two shadow rays, S1 and S2, are cast from the point of the hit. While S1 reaches Light1 without hitting any obstacle, and therefore Light1 will be used for computing of the final color, S1 hits a primitive, which casts a shadow on the sphere.

In the most scenes it is sufficient to set the depth of the recursion between three and five levels for optimal results. A higher depth would slow the evaluation and would not enhance the render enough to pay out. Due to the large number of rays needed, it is difficult to implement an effective algorithm.

For example, finding the nearest hit means that intersections with all objects in the scene should be computed for each of primary and secondary rays. Shadow rays need to find only one hit to stop, which can be in the worst case after checking hits with all objects, in case the shadow ray hits no objects.

Other ways of ray tracing are Monte Carlo methods that are based on stochastic sampling. Such a method is for example path tracing where the nearest his is found and then the ray is randomly scattered according to the bidirectional reflectance distribution function (BRDF) [4].

Path tracing can simulate some effects ray tracing cannot, such as soft shadows, caustics and indirect lighting. Since the direction of reflected rays is random, this method has high level of noise, which can be reduced using more ray samples per pixel (anti-aliasing). A great amount of rays is needed to render pictures with the minimum of noise.

This work is focused on extending ray tracing and path tracing to support CSG. Due computation of large number of intersection in ray tracing, it is important to use acceleration techniques, which are discussed in the next chapter.

2.3 Acceleration techniques

To accelerate rendering there are multiple issues, which should be addressed. Since computing intersections is the most expensive part of this method, the focus of all techniques is to reduce computational time or have less intersections to compute. [5]

The first option is to lower the computational time of each intersection by using methods that can quickly eliminate objects with no chance to be hit, and compute the analytical solution only if there is a chance the object can be hit. Reducing the number of intersections computed can be realized in multiple ways. First, we can have less intersection tests for each ray by excluding parts of the scene by dividing the scene into segments. Second, we can decrease the number of rays by using the coherence of rays or the adaptive termination of the recursion. And finally, it is possible to trace multiple rays at once.

2.3.1 Spatial division

By dividing given space into a hierarchy of subspaces, we can easily exclude all objects in the subspaces that are not intersected by the ray. The necessary condition for this method to work is that intersections between a ray and subspaces are computed fast.

There are multiple ways how to divide a scene into subspaces, each having its advantages and disadvantages. The most used are uniform grids, octal trees, kd-trees and bounding volume hierarchies.

Uniform grids.

Uniform grids are easy to implement and they work fast on regular scenes, but their memory requirements are high and they are unable to adjust to spatial distributions of objects in scenes. They divide the whole space into cells of the same size, store them all and note which objects belong to which cell. Since objects can be a part of multiple cells, they can be stored multiple times, causing already high memory requirements to grow even more.

Ray intersection tests with a grid are simple [7]. First, find the first cell intersected and test all objects in the cell. If no intersection is found, continue with the next cell (the previous cell's neighbour) and repeat testing. All objects in a cell need to be tested to find the closest intersection.

The cell size is uniform (hence the name) and must be decided before constructing the grid. In extreme cases this can lead to two problems with incorrect sampling. In the first case, larger cell size produces a low number of samples and in dense areas there are too many objects in one cell, causing intersection tests to be applied on all objects inside and resulting in lower efficiency. In the second case, the cell size is small and larger objects need to be stored in multiple cells. This causes memory requirements to grow significantly.

Octal trees.

Octal trees also divide space into parts of the same size, but do not store all cells. Instead, a tree structure octree is used. Octree allows an adaptive space sampling and therefore it avoids problems of uniform grids. The root of the octree is one cell encapsulating the whole scene. Each internal node has exactly eight children and divides its space between them equally. Leaf nodes store lists of primitives.

The depth of the octree is adaptively changed according to the scene. If there are too many objects in a leaf, it becomes an inner node and splits objects between its children. The same condition is recursively applied on its children.

Ray intersection tests are done by traversing given octree recursively, starting with the root node. [8] If a node is an inner node, it needs to be decided which children the ray intersects and in what order. It continues with the first one to be intersected and stores the rest on a stack. If node is a leaf, the algorithm computes intersections with all objects stored in the leaf.

If an intersection is found, the algorithm ends. Otherwise search continues with the first node on the stack.

In case all objects of a cell are in one corner and the node reached the maximum of objects, eight children are created, from which seven have no objects and one of them has all of the objects and needs to be split again. This way octrees generate many empty leaves, which leads to searching for a better method and inventing kd-trees.

Kd-trees.

Basic Kd-trees are similar to octrees, but their inner nodes have only two children rather than eight. The splitting plane is axis aligned and changes from node to node. If the axis does not change according to a regular schedule, it needs to be stored. The splitting plane axis and its position is based on the chosen strategy while building the tree. [9]

Building is also similar to octrees, starting with the root node that includes all objects. Every leaf node with too many objects applies the strategy for finding a splitting plane and saves its data and position if needed. Then it sort objects and splits them between its children. Both children are recursively checked in the same way.

Splitting can be also stopped by maximum depth or when surface area heuristic (SAH) decides splitting would no longer be profitable. [1] Trees built using SAH have faster traversals but longer build times. Axes of splitting plane can either be decided by round robin or use more complex ways, such setting the axis according to the widest spread of the cell.

Additionally, kd-trees can be improved by adjusting the position of the each of splitting planes, instead of placing it in the middle of the cell. For example, it can be placed according to the median of the points in the node, or use an adaptive way called sliding midpoint [9]. This strategy primary places the plane in the middle, but in case one of the children is empty, it moves the plane behind the closest object. This way there are no empty leaves.

The main disadvantages are objects on borders of two or more subspaces, which have to be saved into all subspaces they intersect or cut by the border into smaller objects.

Bounding volume hierarchy.

Bounding volume hierarchy (BVH) is also a tree structure, but instead of dividing space into cells that do not intersect, it encloses objects into bounding volumes (BV), which can intersect and do not necessarily cover the entire space. Each node of the tree corresponds to a bounding volume of node's children, which implies the root node is the bounding volume of the whole scene. Leaves are bounding volumes of individual objects or groups of objects.

Bounding volumes have various shapes and selecting the best one depends on our preference of memory requirements, query computational cost and how closely they enclose objects [5]. The more complicated shapes have higher computational cost, but they reject more intersection tests. This allows developers to chose what suits them the best.

BVH does not need to cover the entire scene and so in case of scenes with all objects situated close together, a ray might not pass even the root bounding volume intersection test. In addition, if scene consists of dense areas and sparse areas, traversing of the tree stops after fewer steps than with kd-trees[1].

Since bounding volumes can intersect, each primitive is only in one leaf and not in multiple as it is in case of kd-trees. Consequentially, there is also an advantage when using dynamic scenes. Moving an object within kd-trees requires checking, if the object crossed a boundary and therefore needs to be placed on both sides or moved to the other side. BVH does not need such tests, but moving objects can cause bounding volumes to grow and their efficiency to drop.

The disadvantage of BVH is overlapping of bounding volumes. When traversing a tree we need to decide if children overlap. If not, then we traverse the second child only if we don't find an intersection in the first one. However, if they do intersect, the second child needs to be checked regardless because it is possible for the second child to have an intersection closer than the first child's intersections. This manifests the most when dense scenes are used.

Compared to kd-trees, traversal algorithms tend to be slower[1] for some scenes, since ray intersection tests with bounding volumes are more costly than tests in kd-trees. Moreover, space requirements for storing bounding volumes in each node are higher than for storing planes in kd-tree nodes.

2.3.2 Decreasing the number of rays

The principle of using coherence of rays is the assumption that pixels next to each other have similar colors. Therefore we cast only some rays and interpolate colors for pixels between them. We adaptively change sampling frequency if rays hit different objects or the resulting color is too different.

Since using this method causes under-sampling, it should be used when quick rendering is preferred rather than a good quality. However, the same technique can be used for adaptive anti-aliasing when we sample a pixel by more rays to get more precise results [5].

Another way to reduce rays is stopping recursion of secondary rays when the contribution is lower than some threshold. The contribution is multiplied by a reflection constant, therefore when materials are not mirrors, it is possible to stop the recursion earlier than after the set maximum of levels. However, when the scene has many mirror-like surfaces, checking this condition can cause the opposite effect, because the recursion will not be stopped sooner and checking will only slow down rendering [5].

2.4 Ray tracing with CSG

Ray tracing methods can be adjusted for CSG. The first algorithm was presented in 1980 by Roth[6] and it is still used with some additional optimization.

The difference is that it collects all hits along the ray, including both enter points and exit points. Hits are recorded in the from of the parameter t of the ray and turned into one dimensional intervals between a pair of hits along the ray. After collecting all hits and creating intervals between them, set operations are applied on the intervals according to a CSG tree.

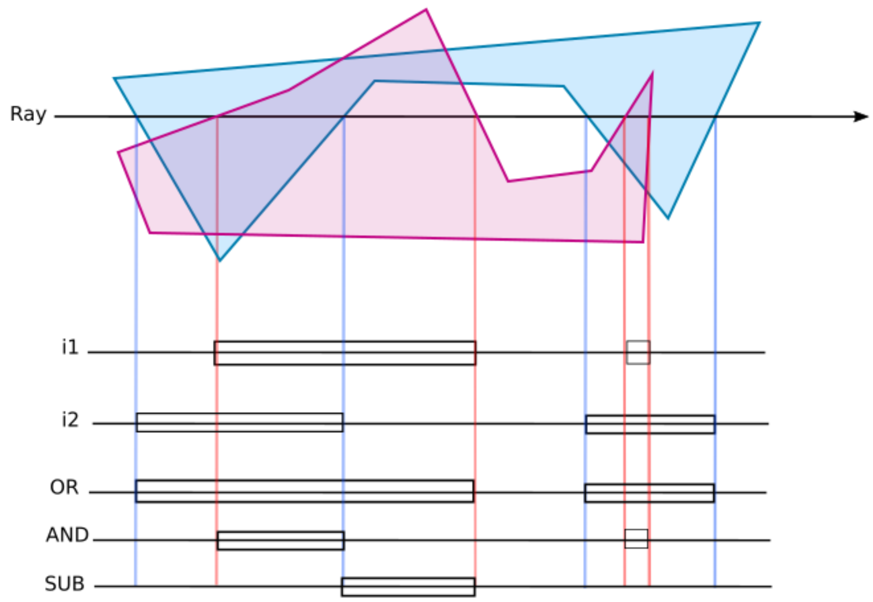

Figure 2.5: Example of ray tracing CSG.

The result can be one or more intervals, as it is shown in figure 2.5, but for opaque objects we need only the start of the first interval, which is the first hit of the final object. For each of secondary and shadow rays we have to repeat the whole process.

There are known cases of ray intersections that could complicate this method. These are when there is only one intersection with a primitive (such as a ray intersects a corner of a cube or a ray is a tangent of a sphere) or there is infinite amount of intersections (a ray is a tangent of a cube). However, these two cases are in reality so rare we do not have to consider them.

Roth's work uses explicit representation of primitives, however it is also possible take an alternative approach and represent primitives as implicit functions and use interval arithmetic to realize set operations[10]. This work also uses explicit primitive representation, therefore implicits are only mentioned as an alternative.

Roth proposed accelerating his algorithm by using "early outs"[6] technique, which is based on traversing CSG trees in left first order. If the set operation evaluated is difference or intersection and there is no hit in the left branch, there is no need to evaluate the right branch.

Intersections with primitives are costly, therefore acceleration methods focus on decreasing the number of intersection tests. Since all intersections along the ray are needed, acceleration methods such as back-face culling are not possible. However, other methods described in the previous section can be used. When using spatial subdivision it is possible to use for each subspace a pruned version of CSG tree, which would only include primitives that can be found in the subspace.

2.4.1 BVH for CSG

The simplest way to construct BVH for CSG tree is to use the tree itself and add bounding boxes in the bottom-up way. This structure is easy to create and it gives the option of creating bounding volumes according results of the set operations. For example in case of AND operation, the resulting bounding volume would not include both children bounding volumes, but only the bounding volume of the set operation result.

Nevertheless, this structure depends on the CSG tree, which is often created by a user and is not balanced. With the depth of the CSG tree grows the BVH's depth and therefore a structure built in this manner is not optimized. A better option is to create a BVH as for scenes without CSG.

Using BVH for ray tracing CSG trees has one disadvantage. Intersections can be found in an incorrect order. This is because bounding volumes can overlap, as was mentioned in the previous section. As a result, we need to sort the intersections before creating intervals, especially if one object can have more than two intersections with the ray (for example tori).

Figure 2.6: The Utah Fairy (174k triangles), rendered with textures, transparency, and shadows. Model originally modelled using DAZ3D's DAZ Studio. Rendered with the OSPRay high-fidelity visualization toolkit.[11]

2.5 Embree

Embree is a collection of high-performance ray tracing kernels, developed by Intel. [11] The kernels are optimized for photo-realistic rendering on the latest Intel processors with support for SSE, AVX, AVX2, AVX512, and the 16-wide Intel Xeon Phi co-processor vector instructions. It supports Windows (both 32 bit and 64 bit), Linux (64 bit) and Mac OS X (64 bit). It runs on any CPU through well defined ISA and has no special hardware requirements.

Embree is targeted at professional developers to help them accelerate their work. Users reported 1.5x – 6x rendering speedup [12] when using

Embree. It has large memory capacity for rendering complex models. Embree is optimized for both incoherent (e.g. Monte Carlo) and coherent workloads (e.g. primary visibility and hard shadow rays). Embree also supports dynamic scenes.

Embree operates on bounding volume hierarchies for BVH's low build times and generally shallow depth enables fast traversal [13]. BVH can be optimized for either memory consumption or performance according to user's settings.

Other features according Embree documentation are:

- Finding either the closest or any hit.
- Single rays or ray packets with size of 4, 8 or 16 rays.
- High performance hierarchy builders.
- Intel SPMD Program Compiler (ISPC) support.
- Triangles, instances, hair and linear motion blur support.
- Extensibility (User Defined Geometry, Intersection filter functions, Open Source).
- Support for Intel Threading Building Blocks (TBB).
- Catmull clark subdivision surfaces.
- Vector displacement mapping.

While Embree is Open Source, it is recommended to use it through its API to benefit from future updates. As for date of writing this paper, Embree is version 2.10.0 and has regular releases of new versions multiple times a year

2.5.1 Embree API

To ensure this work could use any future version of Embree, there will be no changes in code of Embree itself. Instead this work uses Embree API. A complex manual of how Embree works can be found on the official Embree web page, as well as instructions how to run Embree. For purpose of this work this section contains a quick introduction into Embree API.

Devices.

First, a new Embree device needs to be created by calling:

```
RTCDevice device = rtcNewDevice(NULL);
```

Before the application exits, `rtcDeleteDevice(device)` needs to be called in order to execute all destructors properly. Usually, only one device is needed, but having multiple devices is also possible.

Next, at least one new scene has to be created by calling `rtcDeviceNewScene` function and before exiting destroyed by `rtcDeleteScene` function. Scenes are containers for geometries, one scene can contain different types of geometries. After adding geometries to a scene, `rtcCommit` with the scene as a parameter has to be called for Embree to build its internal data structure. Otherwise, all changes will be disregarded. Function `rtcCommit` has to be called every time any changes were performed on the data.

Scenes.

Function `rtcDeviceNewScene` takes as the first argument a device, the second argument is a flag describing the type of the scene, dynamic or static, and the third is a specification of ray queries to be used for the scene. Ray queries flag enables corresponding `rtcIntersect` and `rtcOccluded` functions for the specified ray packet size.

Static scenes allow changes in geometries only until the first `rtcCommit` call. The only way how to change geometries inside a static scene after calling the first `rtcCommit` is to delete the whole scene and create a new one. On the other hand, dynamic scenes can be disabled, enabled, deleted or modified. After each change, `rtcCommit` has to be called again to prevent undefined behavior.

Geometries.

As mentioned before, Embree supports various types of geometries that can be combined. This work will focus on user defined geometries, which allow us to define solids in our own way and use them in the CSG tree.

Since there is no predefined behavior for user defined geometries, we have to implement a bounding function, an intersect function and an occluded function. Furthermore, the user has to provide a user data pointer and pass it to every call of the functions mentioned.

The bounding function is needed to define a bounding box for each geometry in order to build an effective structure over the data. The function sets axis-aligned bounding box for the given object. Similarly, intersect and occluded functions are called to test if the given geometry is intersected or occluded by a ray. Since multiple sizes of ray packets are supported, different intersect and occluded functions should be provided for sizes of ray packets used in the application.

A user defined geometry is created by calling `rtcNewUserGeometry` function with parameters the scene and a number of geometries to generate. Function returns index of the first created geometry, which needs to be stored with the corresponding geometry and will be used as an identification.

As it was already mentioned, a data pointer has to be provided and that is done by calling `rtcSetUserData`, having a scene, geometry ID and a data pointer as parameters in this order. Once data are specified, functions are set by calling `rtcSetBoundsFunction` for bounds, `rtcSetIntersectFunction` for intersection and `rtcSetOccludedFunction` for occlusion. Their third parameter is the name of the function to be called.

Instances.

Embree supports creating instances of a scene inside another scene using some transformation to pass between two scenes. Creating an instance does not mean duplicating geometries, hence this system is very useful when scenes include some objects multiple times. However, only one level of instancing is natively supported, although the documentation suggests it is possible to implement more levels using user defined geometry. Nevertheless, there is no native support and users would have to realize this manually.

Instances are created using the `rtcNewInstance` function call and, as Embree documentation states, "potentially deleted" [11] by calling `rtcDeleteGeometry` function. Before creating an instance, two scenes has to be created first. For adding sceneB into sceneA following code is used:

```
unsigned instID = rtcNewInstance(sceneA, sceneB);
rtcSetTransform(sceneA,instID,RTC_MATRIX_COLUMN_MAJOR,&mat);
```

The instanced scene has to be committed before the scene it belongs to and they both have to be in the same device. Instances are automatically checked for intersections with the main scene. In case of a hit, `geometryID` and `primitiveID` relate to the instanced scene and the `instID` is set to the number returned when creating the instance.

Rays.

A ray is in Embree represented by a `RTCRay` structure. The structure has attributes that are set when a ray is created and should not be changed during the `rtcIntersect` call. These values are: `org` (ray origin), `dir` (ray direction), `tnear` (where ray enters the scene), `mask` (for packets of rays) and `time` (for motion blur).

Additionally, the ray structure contains attributes to store the hit information. While ray tracing the scene, geometries should not be changed to avoid thread-related problems, so the ray itself carries the hit information. Value `tfar` is initially set to infinity and when an object is hit, its value is set to parameter `t`, where the ray intersect the object. Values `u` and `v` are local hit coordinates, `Ng` is the geometry normal and `geomID` is the ID of the geometry that was intersected. Since a geometry can contain more primitives, rays also include `primID` value, which can be set to the index of the primitive in the geometry. If `geomID` is set to anything other than `RTC_INVALID_GEOMETRY_ID`, Embree evaluates it as a hit and ends ray tracing.

Embree leaves creating rays and evaluating results of ray tracing to users. As a result, a user can decide how many rays will be created, from where and in which direction they will be cast. A ray is cast by calling `rtcIntersect` function with the scene and the ray as parameters. Any secondary or shadow rays have to be cast by user after the return of this function.

Filter Functions.

Embree also supports so called filter functions, which are called when a hit is found. This gives user an opportunity for an additional evaluation, such as collecting all hits, counting hits or accumulating opacity.

There are two kinds of filter functions, intersection and occlusion. An intersection filter function is set by `rtcSetIntersectionFilterFunction` and if set, it is called automatically when the intersection function finds a hit. An occlusion filter function is set by calling `rtcSetOcclusionFilterFunction` and its calling is analogous. Both filter functions have four different versions for different packet sizes, which needs to be implemented if packets are used.

Chapter 3

Implementation

This chapter explains how to exploit Embree for CSG rendering. The main part of the program can be split into two separate parts: collecting all intersections along the ray, and evaluating them according to given CSG tree. The first part uses Embree to get all intersections, second does not depend on Embree. Figure 3.1 shows how Embree works with my C++ code.

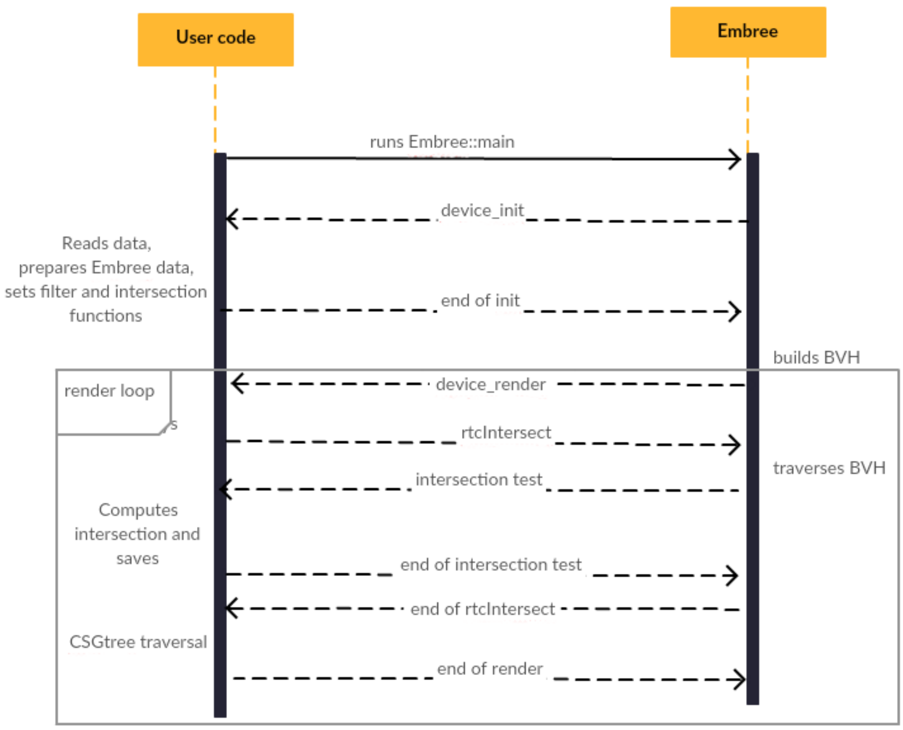

Figure 3.1: Diagram of communication between Embree and user code

Embree provides three functions for users to modify. The first one is `device_init` and, as its name suggests, it is for initialization of Embree and it runs only once at the beginning of the program. Section 3.1 describes my code of this function. The second function is `device_render` and it runs repeatedly for each frame. My code for this function is explained in sections 3.2 and 3.3. The last function is `device_cleanup`, which is called once when the program ends and its purpose is to free all allocated memory.

3.1 Setting up

Before starting ray tracing, first we need to set up the scene. As mentioned before, an Embree device and an Embree scene need to be created first. After that, a CSG tree is read from a file. Additionally, it is recommended to check, if there is at least one light when using a visualization method that requires lights, unless lights are set elsewhere.

3.1.1 File format

Since there is no official standard for representing CSG trees, I have decided to use a simple format that is easy to understand and edit. This format supports inner nodes with set operations, leaf nodes with primitives and a special kind of recursive nodes with additional CSG trees from other files.

To keep the format simple, transformations are kept in leaf nodes because Embree supports only one level of instances and needs only the final transformation. However, this format could be adjusted to include transformation nodes. In this case transformations would have to be collected while traversing the tree and again, applied on leaves.

The format uses ASCII and every file starts with a version of the format, currently `CSG 1.0`, followed by a space and a CSG scene definition. Each node definition starts with a vertical dash followed by a space.

Inner nodes have recursive definition:

```
| CSG <set opeation> <left child> <right child>
```

Set operations are "or" for union, "and" for intersection, "sub" for subtraction and "Union <d>", where <d> stands for the number of nodes in the union. The last one is a special case since it is not a set operation as such, but a short way to express a part of a tree where all operations are union. This is useful because union is the most used set operation in regular scenes and so users do not have to write a tree structure for connecting multiple parts of the model together.

Recursive leaves have following syntax:

```
|  CSGtree <file name>
```

After encountering `CSGtree` keyword, reading from the file stops, another file is opened and its content is read in place of the `CSGtree` node in the first tree. This can work recursively for multiple files. After finishing reading the second file, reading of the first file continues.

A leaf definition starts with a key word defining the type of the primitive stored in the leaf. Currently supported primitives are spheres, cubes, cylinders, cones and tori. Furthermore there is also one more keyword for triangle meshes, `Scene`. Following a keyword is a space, additional data, another space and a node transformation.

Additional data are defined only for tori and `Scenes`. Tori need to have one parameter defined explicitly since a torus is defined by two radii and no homogeneous transformation can preserve one while changing the other. `Scene`'s additional parameter is a path to a scene definition file.

The transformation is a set of twelve numbers surrounded by parentheses. Numbers are separated by a comma.

For example, intersection between two spheres is written as:

```
CSG 1.0 | CSG and | Sphere (1.00,0.00,0.00,0.00,1.00,0.00,
0.00,0.00,1.0,0.0,0.00,1.00) | Sphere (1.00,0.00,0.00,0.00,
1.00,0.00,0.00,0.00,1.00,0.00,0.00,-1.00)
```

Two Cylinders substracted and the result is added to a sphere:

```
CSG 1.0 | CSG or | CSG sub | Cylinder (<transformation>) |
Cylinder (<transformation>) | Sphere (<transformation>)
```

Union definition of five spheres is set in the following way.

```
CSG 1.0 | CSG Union 5 | Sphere (<transformation>) |
Sphere (<transformation>) | Sphere (<transformation>) |
Sphere (<transformation>) | Sphere (<transformation>)
```

Another feature this simple format lacks, aside from transformation nodes, is a material definition for primitives. Triangle meshes use materials from their files, but for primitives it is necessary to add material definitions manually when creating primitives.

3.1.2 CSG tree

CSG tree has the same structure as a binary tree with data in leaves. The basic node is virtual and has two attributes, pointer to its parent and a flag if the node is a leaf or not. From this node two types of node are derived.

Inner nodes have two pointers, one for each child, and the type of the set operation to be applied for this node. Leaves have only one attribute and that is the index of the primitive. All primitives are stored in an array aside from the tree itself, but the array is a part of `CSGtree` class.

Primitives are defined as:

```
class Primitive{
public:
    P_Type type;
    Material material;
    bool isSolid;
    int scene;
    unsigned int indexOfPrimitive;
};
```

`P_type` is an enum of primitive types in following order: sphere, cube, cylinder, cone, torus, mesh. The order is important only to sort primitives with the maximum of two possible intersections with a ray from primitives that can have more intersections, such as tori and meshes.

Structure `Material` are two Embree `OBJMaterial` instaces for the inside and outside materials. The next parameter, `isSolid`, is a flag to mark primitives that are solids and have defined inside for the reasons that are explained in section 3.4. `Scene` is the index of the mesh scene for mesh primitives. If no mesh scene is attached, this attribute is set to -1. The last attribute is the index of the primitive, which corresponds with its index in the array and ID of the instance of the primitive.

Transformations are not stored in the CSG tree because there is no need for them there. Embree needs transformations when creating instances and it manages all word and model coordinate transfers. Using instances and transformations, Embree transforms rays into a model space, computes intersections and returns results in the word coordinates. Therefore there is no need for storing transformations, unless they change during rendering.

3.1.3 Reading data

Building a CSG tree from a file is simple, when a CSG command is read, an inner node is created and then recursively the two children are read. In case of a Union command, the program attempts to create a balanced subtree. The resulting subtree has the maximum difference of depths between its leaves equal one.

There are two ways of handling leaves. For both of them a primitive is created and added to the primitive array. The first way is the same for all primitives except mesh scenes. When a geometry for Embree is created, a new Embree scene is also created. This scene must be stored so it could be properly deleted at the end of the program. Not deleting scenes properly can lead to memory errors mentioned in section 3.4, which is why it is necessary not to skip this step.

To save space, instead of creating an Embree scene for each primitive in the tree, scenes are reused. The idea is to create only one Embree scene for each type of CSG primitives and use Embree instances to generate instances of primitives with set transformations.

As a consequence, all primitive geometries must be created in a standard way. E.g. unit sphere with the center in the origin of coordinate system, unit cube axis-aligned in the positive octant, cylinder centered around the origin with height and radius equal one, cone with the same size and tip in the origin, and torus centered around the origin in the x/y plane and major radius equal one. Torus cannot be stored this way for the reason of its additional data.

All created Embree scenes are stored in a `std::vector`. Furthermore there is a short array of indices into the vector of scenes for each reusable primitive. Primitives are in order given by `P_type`. If no Embree scene was created for given primitive, the default value is -1.

Creating leaves proceeds as following algorithm:

```
if (scene for a primitive is not created yet){
    Create Embree scene g_scene0;
    Store the scene into the scene vector;
    Create Embree geometry for g_scene0;
    Commit g_scene0;
}
Create a new instance of the scene in the main scene;
Set transformation for the instance;
```

Creating a torus skips the step with checking, if a scene was already created, and creates a new scene for every torus in the scene.

Reading mesh scenes follows Embree tutorial Pathtracer. This tutorial only shows how to have one mesh in the scene, therefore the code had to be adjusted to support more meshes at the same time. Following structure saves all data needed for the path tracer, as it is in the tutorial.

```
struct MyScene{
    ISPCScene * scene;
    OBJScene * scene2; // a pointer for memory release
    void** geomID_to_mesh = nullptr;
    int* geomID_to_type = nullptr;
};
```

Since these are also dynamically allocated, they need to be freed at the end of the program. Following code shows how to set up an `OBJscene` with its path in variable `file`.

```
MyScene * ms = new MyScene;
ms->scene2 = new OBJScene();
Ref<SceneGraph::Node> node = loadOBJ(file,g_subdiv_mode !="");
ms->scene2->add(node);
ms->scene = (my_set_scene(ms->scene2));
scenes.push_back(ms);
g_scene0 = convertScene(scenes.at(scenes.size()-1));
```

After this code follows the same procedure as with other primitives, `g_scene0` is saved, committed and a new instance is created. Since instances do not have an explicit destructor and they are supposedly deleted when their related Embree scenes are deleted, there is no need to store instances.

3.1.4 Embree

The last thing, that is required before starting ray tracing CSG, is to set up the Embree data. How to set up devices and scenes was already mentioned in the chapter about Embree. For ray tracing CSG we need to find all intersections and the Embree documentation promised an easy way to do so without restarting rays, that is by using filter functions.

What the documentation did not mention was it only works for some types of geometries and not for user defined geometries. This was only discovered after debugging Embree code, because Embree did not provide any warning or suggestion. Since user defined geometries also require to provide bounding boxes, intersection tests and occlusion tests, the filter was implemented directly into intersection and occlusion functions.

Triangle meshes.

For triangle mesh geometries Embree already has optimized intersection and occlusion tests, therefore we only need to provide a filter function, which stores intersections and refuses them by setting `ray.geomID = RTC_INVALID_GEOMETRY_ID;`

Filter functions are set after providing Embree the geometry data. For example, for triangles it is after function `rtcSetBuffer` in the following way:

```
rtcSetIntersectionFilterFunction(scene_out, geomID,
    triangleFilterFunc);
rtcSetOcclusionFilterFunction(scene_out, geomID,
    triangleOccFilterFunc);
```

where `triangleFilterFunc` and `triangleOccFilterFunc` are the provided functions. Note that when using ray packets, different filter functions have to be provided as well.

User Defined Geometries.

This part explains how to set up Embree user defined geometries on one example: the sphere. Other primitives are set up in a similar way. Geometries are created as:

```
void createAnalyticalSphere (RTCScene scene)
{
    unsigned int geomID = rtcNewUserGeometry(scene, 1);
    Sphere* sphere = new Sphere;
    sphere->geomID = geomID;
```

```
    rtcSetUserData(scene, geomID, sphere);
    rtcSetBoundsFunction(scene, geomID,
            (RTCBoundsFunc)&sphereBoundsFunc);
    rtcSetIntersectFunction(scene, geomID,
            (RTCIntersectFunc)&sphereIntersectFunc);
    rtcSetOccludedFunction(scene, geomID,
            (RTCOccludedFunc)&sphereOccludedFunc);
}
```

This code is taken from the Embree tutorial with one change. In the tutorial, sphere's center and radius were set. Since my work uses standard settings for primitives, all spheres are stored with the center in the [0,0,0] coordinates and the radius is always equal one. As a result there is no need to store those values. Furthermore, in this work there is only one primitive per scene, which makes `geomID` for spheres useless.

Regardless of that, Embree needs some data to be provided only so it could offer them back to user when computing intersections. In conclusion, the structure `Sphere` can contain whatever the user wishes to use in intersection or occlusion tests.

Now we will take a closer look at the three functions provided, `sphereBoundsFunc`, `sphereIntersectFunc` and `sphereOccludedFunc`. The first named is for defining an axis-aligned bounding volume for our data and it has to have its signature as:

```
sphereBoundsFunc(const Sphere* spheres, size_t item,
    RTCBounds* bounds_o)
```

The first argument is a pointer to the data provided in `rtcSetUserData`. In our case it is a pointer to one sphere, but in general, it can contain multiple spheres and so thr second argument is the index of the primitive for which the bounding volume applies. The last argument is an Embree structure with members for lower and upper bounds for each of axes to store the bounding volume.

The signature of the intersection test function is:

```
sphereIntersectFunc(const Sphere* spheres, RTCRay& ray,
    size_t item)
```

The first and the last arguments are the same as in the previous function, the second one is the Embree ray structure. In this function, a precise hit is analytically computed. Since the filter function is a part of this function as well, after computing the intersection, intersection normal and texture coordinates, all intersection data are stored and the hit is refused.

The occlusion function does the same thing, except it does not need to compute additional data, such as normals. Its signature is also the same, except for the name of the function.

3.2 Collecting intersections

After providing Embree with all data needed, Embree builds its internal BVH and starts the rendering loop. Embree leaves all ray tracing management on the developers, therefore the way rays are created and in what amount is completely up to developers.

Before ray tracing, each ray has to set its origin, its direction, the closest and the farthest point in the main Embree scene and `geomID` set to `RTC_INVALID_GEOMETRY_ID`. After everything is set, function `rtcIntersect`, with the main Embree scene and the ray as arguments, has to be called to start ray tracing. Then Embree traverses its inner BVH and searches for intersections.

3.2.1 Thread related problems

Since we need all intersections for each ray, results stored in the Embree ray structure are not used. Instead, all intersections are stored elsewhere and evaluated after `rtcIntersect` returns.

When designing a storage for intersections, one has take into consideration that Embree uses threads for efficiency. Specifically, it uses one thread for ray tracing one pixel. Since the storage needs to be accessible from filter and intersection functions, it cannot be a local variable, which would prevent all thread-related issues, such as two threads rewriting one memory block at the same time.

The problem is Embree does not give any information about threads and rays do not have any identification that could be used as an index for storing the intersections. The structure of Embree rays should not be modified to keep the application effective and not dependent on one version of Embree, since the ray structure could change in future releases, as stated in Embree documentation.

Using given ray attributes to store an identification would limit options for ray tracing, but it could be used for a specific cases, when it is known that some features (for example motion blur) will not be used. This work shows how to use CSG with Embree in general, without limiting the capability of Embree.

Various approaches, such as using a hash function based on ray's origin and direction, were considered, but since we cannot guarantee there would not be two rays with the same origin and the same direction (for example in case of methods where ray directions are randomly generated) these methods were rejected. The final version uses addresses of `RTCRay` objects as identifications. This can be used only because `RTCRay` object are passed by reference and therefore their addresses are constant within the run of `rtcIntersect`.

Furthermore, it was discovered that each thread allocates rays at the same address every time during one render, therefore the number of unique addresses for primary rays is equal to the number of threads used. The default number of threads is, according to my observation, equal to the number of processor cores, but that is not guaranteed by Embree.

This is a big advantage, because instead of allocating a huge storage for every pixel, we can predict how many rays will run at the same time, allocate intersection storage only for a small amount of rays and reuse the storage when ray tracing of a pixel ends.

The hash created using ray reference is used as a key into a `std::map` object, where indices into the intersection storage are saved. The indices are generated using an atomic counter every time a new ray needs to store intersections.

3.2.2 Intersection Storage

With a ray identification method, it is possible to store intersections into separate places. It is convenient to use a structure to hold all data for one intersection. My `Intersection` structure copies `RTCRay` with one additional member, a flag `isInside`, which is used in case there is a different material for insides and outsides of solids. By default, this flag is set as false and changed when traversing the CSG tree, which is explained in the next section.

The first version had one two-dimensional array of pre-allocated intervals with one row for each thread. This was changed to a more intuitive way of storing all ray related data together in one class, `RayData`. Instances of these classes are stored in a short array according to indices of rays stored in the hashmap mentioned earlier.

`RayData` class is implemented as:

```
class RayData
{
public:
    Intersection buffer[maxIntersections];
    int lastIntersection;
    std::list<Interval> list;
    std::list<Interval> pool;
};
```

where `maxIntersections` is a constant of estimated maximum number of intersections per ray. Intersections are stored in `buffer`. This method has high memory requirements, but it is the fastest solution available since any kind of dynamic allocation slows down the implementation significantly.

Integer `lastIntersection` is the index of the last valid intersection in the buffer, because intersections are not deleted, only rewritten, therefore we need to remember the last intersection stored. Last two members of `RayData` are lists of `Interval` used for creating intervals from intersections, which is explained in the next subsection.

3.2.3 Intervals

Structure `Interval` is two pointers into the intersection buffer defined as:

```
struct Interval{
    Intersection * min;
    Intersection * max;
}
```

Intervals are stored in a `std::list<Interval>` container for more convenient way to work with them. It allows to have a sorted list and add items in the middle without moving other items and therefore it is easy to merge two lists together.

Dynamic allocation is costly and should be avoided, which is why this code uses a pool of preallocated intervals and function `std::list::splice`, which moves items from one list to another without allocation. Once all pools are initiated, no more intervals are allocated during the entire run of the code. Pools are a part of `RayData` structure, as mentioned before. Each pool is set as:

```
pool = std::list<Interval>(maxIntersections / 2);
```

Before creating intervals from intersections, first it is necessary to sort all intersections along the ray. Since bounding volumes can overlap, it is possible to find intersections in an incorrect order. In case of solids, all intersections are computed together and it is possible to sort them before saving. It does not matter in which order intervals from different primitives are created, important is only the order of intersections for one primitive, so intervals could be created correctly.

However, the problem is when triangle meshes are considered as one primitive in CSG. Intersections of triangles are stored separately in the filter function in the order Embree choose to use. For this reason all intersections are sorted before creating intervals.

After sorting, all intersections up to the index set in `lastIntersection` are sequentially read and paired according to their `instID`. Pointers of `Interval` are set and the new interval is moved from the pool to the `RayData::list` to avoid the allocation of the new list for every ray.

Creating intervals is followed by evaluating intervals according the CSG tree, which is explained next. After obtaining results it is imperative to set `lastIntersection` back to a negative value and move all items from `RayData::list` back to its pool.

3.3 Evaluating CSG

Tree traversal is triggered by calling function `CSG_tree::traverse`, which takes two arguments, a reference to the list of the data and the index of the ray, and returns the number of intervals in the resulting list. The return value can be used to display the number of intervals for the ray.

The list of the data received should not change since all items need to be moved back it its pool. The ray index is needed to store and retrieve data tools for each traversal. These tools are defined as:

```
class Tools{
public:
    std::list<Interval>::iterator intervals[maxPrimitives];
    bool flags[maxPrimitives];
    std::list<Interval>::iterator it1;
```

```
    std::list<Interval>::iterator it2;
    std::list<Interval>::iterator temp;
    std::list<Interval> pool;
    std::list<Interval> result;
    std::list<Interval> lists[maxPrimitives];
    int listCounter;
};
```

`MaxPrimitives` is an estimated maximum number of the primitives. `Flags` and `intervals` are related to each of primitives. Flags are boolean values answering the question if the primitive was intersected by the ray. If the value is true, `intervals` hold an iterator to the first interval that belongs to the primitive. Both arrays are set before traversing the CSG tree. This way the list of data is searched only once and not when all intersections for a primitive are needed.

Iterators `it1`, `it2` and `temp` are used in set operations functions, but are defined here to avoid dynamic allocations. `Result` is a list for storing the final intervals and `pool` is again a pool of pre-allocated intervals. `Lists` is an array of auxiliary lists and `listCounter` is a counter of used `Lists`.

3.3.1 Tree traversal

After all tools are set, `traverseTreeRek` is called with the original data, the `result` list from the `tools`, the pointer to the root node and the index of the ray as arguments. This function is a recursive, depth-first traversal of the whole CSG tree which stores its result into the list given on the second position. No tree primming is applied.

If the function receives a pointer to a leaf node, it retrieves the index of the primitive from the node and uses it to check if value of `flags` on this index is true and the ray did intersect the primitive. If it is, it retrieves the iterator from `intervals` and reads sequentially intervals in `data` from this iterator. Each time the interval read belongs to the primitive, an item from `pool` is moved to `result` and both `Interval` pointers are set according to those in `data`. After the function ends, the list given on the second position holds all intervals from `data` that belong to the primitive.

In case the primitive is a sphere, a cube, a cone or a cylinder, there can be only one interval stored, since each of these primitives has only the maximum of two intersections with a ray. Therefore, for these primitives, it is sufficient to find only the first interval and end the function. For meshes it is necessary to go through all the intervals for obvious reasons.

If `traverseTreeRek` function receives a pointer to an inner node, it calls itself on its left child first and passes it `result`. In `result`, there are now resulting intervals of the node's left child. The next step is decided according to the node's operator.

If the operator is Difference, the right branch is only traversed if the `result` is not empty. Otherwise, the result would be an empty set regardless of the left branch. Since `result` is already empty, the function can end. The traversal of the right branch is called with a new list for storing its results. This list is also checked and only if it is not empty, the right branch result is subtracted from the left one.

When the operator is Union, the right child is traversed in any case, but both results are only merged if the right child returned something to be merged. Otherwise, the result would be intervals from the left branch that are already stored in `result`. The last operator, Intersect, traverses the right child only if `result` is empty after traversing the left child for the same reason as Difference. Then it runs the set operation.

This method uses lists to save and merge intervals. A list is needed every time the traversal reaches a leaf. This can happen in one traversal as many times as many there are primitives. Allocating a list each time was slowing down the program significantly, therefore all lists are pre-allocated in `tools::lists`. Every time the traversal of the right child is called, a new list is passed as a reference. In the end of the function, all intervals from this list are moved back to `pool`.

The function is described by following pseudo-code:

```
void traverseTreeRek(std::list<Interval> &data, std::list<Interval> &result,  C
if (n is a leaf){
    retrieve index of the primitive;
    if (flags[index] ==true)
        get the iterator it1 from intervals
        for each interval of the primitive:
            copy pointers to an interval in the pool;
            move the interval from the pool to result;
        end;
    else{
        //traverse the left branch
        traverseTreeRek(data, result, n->left, index);
    switch (n->setOperator){
        case OP_Type::Difference:
            if (result is not empty){
                get an unused list from lists;
                //traverse the left branch;
                traverseTreeRek(data, list, n->right, index);
                if (if list not empty)
                    myDiff(result,list, index);
            }
            break;
        case OP_Type::Union:
            get an unused list from lists;
            traverseTreeRek(data,list, n->right, index);
            if (if list not empty)
                myUnion(result,list, index);
            break;
        case OP_Type::Intersect:
            if (result is not empty){
```

```
                get an unused list from lists;
                traverseTreeRek(data,list, n->right, index);
                myIntersect(result, list, index);
            }
            break;
        }
        return all items from list to the pool;
    }
}
```

3.3.2 Union

Union function can end quickly if the first list (`i1`) is empty. In this case the result of the union is in the second list (`i2`), but since this method gives results in `i1`, all intervals from `i2` are moved to `i1`. Otherwise, there are three cases of interval relation. In the first case, the interval from `i1` ends before the other starts, but since `i1` is the result of this union, no intervals are moved. In the opposite case, we need to move the interval from `i2`.

The most interesting is the third case (figure 3.2), when intervals intersect. The first step is to extend the first interval to cover the second one. Next, we need to check if the new extended interval does not cover more intervals from `i1`. Those need to be moved from `i1`. Also, it is necessary to update the maximum of the new interval in case one of deleted intervals ends after the `i2` interval.

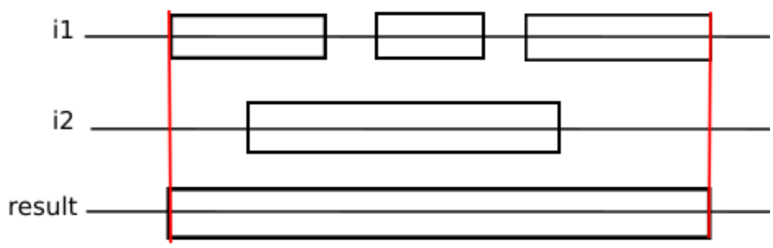

Figure 3.2: Union - case 3.

All three cases are implemented as:

```
it1 = i1.begin();
it2 = i2.begin();
while (it1 is not end of i1) and (it2 is not end of i2){
    if (it1 ends before it2 starts)
        it1++;
    else (it2 ends before it1 starts){
        move it2 to i1 on position it1;
        it2++;
    }else { //intervals intersect
        if (it2 starts before it1)
            (*it1).min = (*it2).min; //set min
        if (it2 ends after it1) {
            (*it1).max = (*it2).max; //set max
            if(it2 intersects more intervals from i1)
                update it1.max;
                move intersected intervals from i1 to pool;
        }
        it2++;
    }
    if (it1 reached the end before it2)
        move the rest i2 intervals to i1;
}
```

3.3.3 Difference

Difference follows an algorithm similar to the previous one but with five cases of interval positions. If they do not intersect, the algorithm moves to another interval on one of the lists. In case `it2` is in the middle of `it1` (as shown in figure 3.3), the result are two intervals and therefore we need to move one interval from the pool. Additionally, cutting an interval short means we need to set its `isInside` flag. Furthermore, the normal of the inside intersection needs to be in the reverse direction to point out of the interval.

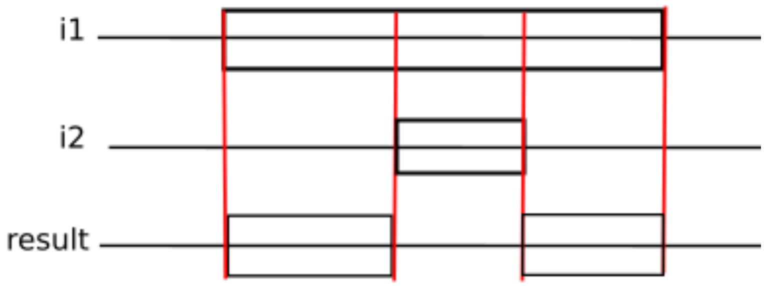

Figure 3.3: Difference - case 2.

This case can be described with pseudocode:

```
move an interval from the pool at it1;
set its min as it1.min;
set its max as it2.min;
set its max->isInside true;
reverse normal of max;
set max->geomID and max->primID according min;
set it1.min to it2.max;
set it1.min->isInside true;
reverse normal of it1.min;
set min->geomID and min->primID according max;
it2++;
```

If `it1` starts before `it2` and also ends before `it2`, `it1.max` is set to `it2.min`. Again, the normal has to be flipped and `isInside` flag is set true. As in the previous case, `geomID` and `primID` should be reseted.

If `it2` starts before `it1`, there are two cases of what can happen. In the first one `it2` also ends after `it1` and therefore the whole `it1` is not a part of the result and it is moved back to the pool. Otherwise, `it1.min` is set to `it2.max` and again, the normal is flipped, `isInside` flag set and `geomID` and `primID` are reseted.

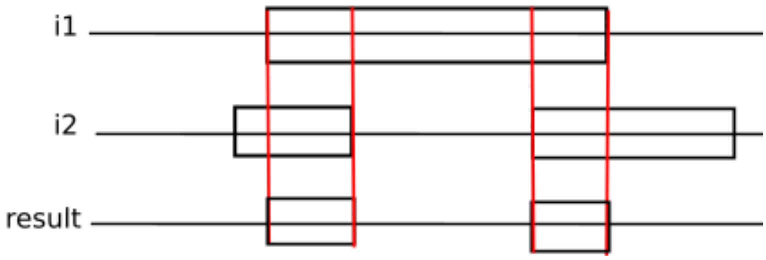

Figure 3.4: Intersection.

3.3.4 Intersect

If either of lists is empty, the result of this method is also an empty set. When the algorithm finds an interval that does not intersect any interval of the other list, it is moved to the pool.

If intervals do intersect, the final interval is defined as `[max(it1.min, it2.min),min(it1.max,it2.max)]` . Normals are not flipped, but `isInside` flag is set true for both ends of the interval. However, if `it2` ends before `it1` , we still need to test the original `it1` interval with other `i2` intervals, as shown in figure 3.4

When the second list is completely traversed but there are some intervals left in the first list, all remaining intervals from the first list are moved to the pool.

3.3.5 After traversing

Since usually ray tracing methods need only the first intersection, there is a method to be called after traversing the tree:

```
Intersection * getFirst(int index)
```

Its argument is the ray index for retrieving the correct result list. It returns the first intersection from the first interval in the result. When the result is no longer needed, it is flushed by calling `flushResult(int index)` , which returns all items from result to the pool.

3.4 Problems encountered

Last but not least, I would like to note two problems encountered that were not mentioned in the text so far. The first one is a problem of meshes and solids, and the second one is a memory problem.

3.4.1 Non-manifolds in CSG

Combining triangle meshes with solids brings problems of meshes into CSG. CSG requires primitives in leaves to be solids for intervals to create properly. We can grant this by using only models specifically created as manifolds, but that limits us only on models we make. When using models by other designers we cannot guarantee this condition is fulfilled.

Usually, designers do not create meshes with holes, caused by disconnected vertices and edges, and areas with no thickness are also not a problem, since the probability of hitting the area is nearly zero. The real problem are internal faces. Those faces can lead to an odd number of intersections or pairing intersections in an incorrect way. For example, if two cubes have both an internal face, it can happen the back face of the first cube and the front face of the second become a pair.

The quickest way to fix this problem is to add a virtual intersection closely behind every found intersection of a mesh marked as not solid. The virtual intersection has the same ray data as the original one, but it has its normal reversed. This ensures there is always an even number of intersections and it simulates behaviour of a thin shell of the primitive. While this fixes the problem, it doubles the space needed for storing intersections, as well as the space for intervals.

The more sophisticated way to solve this problem would be to check normals of every intersection and pair intersections with normals pointing the opposite direction. This method would give better results, but checking normals would be costly and therefore it would slow down the application significantly.

3.4.2 Memory problems

During my work I encountered problems with access violation that were only triggered after the program run previously in the release mode. Despite using Visual Studio, which should clean the used memory after every run, this error still happened.

I fixed this problem by ensuring Embree destructors are properly called by the end of each run. As it was mentioned before, methods for deleting Embree scenes and devices have to be called explicitly. Therefore the cleaning method has to be called before every possible end of the program.

While this problem is an implementation issue, it is mentioned in case any future work would follow this thesis and encounter the same problem.

Chapter 4

Results

This work was tested on HP EliteBook 840 G1 (Intel Core i5-4310, 8 GB RAM), with Windows 7 Professional 64bit version.

Reference times are taken from Embree tutorial Pathtracer with default settings (one sample per pixel, maximum recursion depth is eight). The same settings are used for the path tracer in my work. All renders are in 512x512 resolution.

Results are for two versions of display. The first is using simple Phong shading model and the second uses path tracer according to Embree tutorial. Additionally, there is also an illustrative mode, which maps number of intersections on grayscale.

Tested variables are render time, render time per ray and number of set operation per ray.

4.1 Austrian Imperial Crown

First tested scene is “Austrian Imperial Crown” model by Martin Lubich. This scene has 4 868 924 triangles. Embree path tracer average render time is 238 ms.

Tested cases were the model itself and randomly generated cylinders shooting through the model. CSG trees were generated as Union of cylinders subtracted from the crown. Test cases can be seen in figure 4.1.

Figure 4.1: Test cases. Top left: 0 cylinders, top right: 10 cylinders, bottom left: 100 cylinders, bottom right: 1000 cylinders,

	#OR/ray	#SUB/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Cylinders: 0	0	0	0	145	1.8079
Cylinders: 10	0.04	0.001	0.041	170	1.5451
Cylinders: 100	1.19	0.21	1.40	282	0.9282
Cylinders: 1000	13.62	0.27	13.89	1 512	0.1734

Table 4.1: Austrian Imperial Crown, rendering mode: Phong, triangles: 4 868 924, resolution: 512x512, pathtracer reference time: 238 ms

In table 4.1 there are values for testing cases rendered using Phong. The number of rays is in this case constant, 262 144 rays per frame, because only primary rays were used. First three columns show the number of union, difference and all set operation together per one ray. Intersection is not used in this case, therefore it is not mentioned in this table. The last two columns are render time per frame and trace speed.

There is one notable observation about this table. The number of operations does not grow linearly with the number of cylinders. This has two causes. In the first iteration of tests the union operation ran regardless of the emptiness of the given lists and subtraction needed only the first list not empty.

This resulted in a constant number of subtractions per ray for all cases. This was caused by the way the tree is build. There is only one subtraction in the entire tree, and it was only evaluated if the left side of the tree (the crown) was hit. Therefore, this number expressed how many rays hit the crown on scale from zero to one, where one meant all rays hit the crown. The value for this scene was 0.44, therefore we can say more than a half of the rays did not hit the crown.

When subtraction only runs if it receives two not empty lists, the number of operations grows in a non linear way, since cylinders are generated randomly and the more there are, the more likely there are to intersect each other and not add more subtracted space.

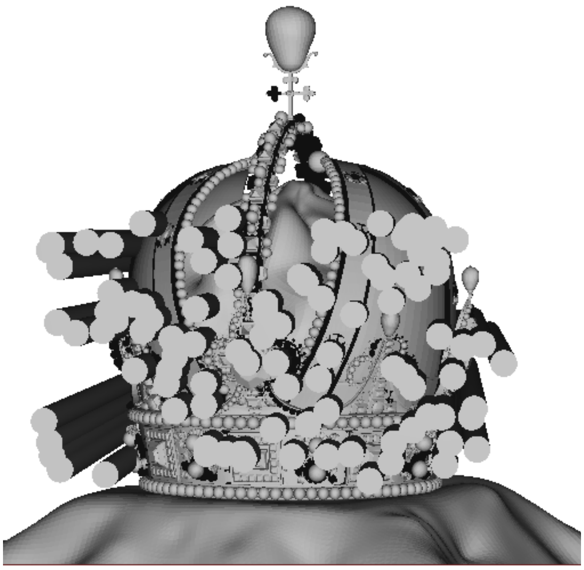

Figure 4.2: Render of the scene with 100 cylinders where difference was changed to union.

Second, cylinders are generated aligned with Z axis, not towards the camera. Together with the field of view limited to ten degrees (the default settings of this scene) this causes cylinders to face towards sides. This can be seen in figure 4.2. As a result, rays, which do not intersect the crown, still do intersect cylinders. As the number of cylinders grow, there are also more of cylinders on the sides of the crown and more rays intersecting them but not the crown. This causes the non-linear growth of the union set operation.

	#Rays	#OR/ray	#SUB/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Cylinders: 0	714 847	0	0	0	816	0.8760
Cylinders: 10	712 435	0.25	0.13	0.38	863	0.8255
Cylinders: 100	661 204	3.47	0.36	3.83	1 295	0.5106
Cylinders: 1000	407 174	21.52	0.28	21.80	3 203	0.1271

Table 4.2: Austrian Imperial Crown, rendering mode: pathtracer, triangles: 4 868 924, resolution: 512x512, pathtracer reference time: 238 ms

Table 4.2 shows the same measurements for path tracer. This mode uses secondary and shadow rays. Furthermore, secondary rays are generated randomly, therefore the number of rays changes from frame to frame. The number of rays drops gradually, since primary rays without hit do not generate any secondary or shadow rays.

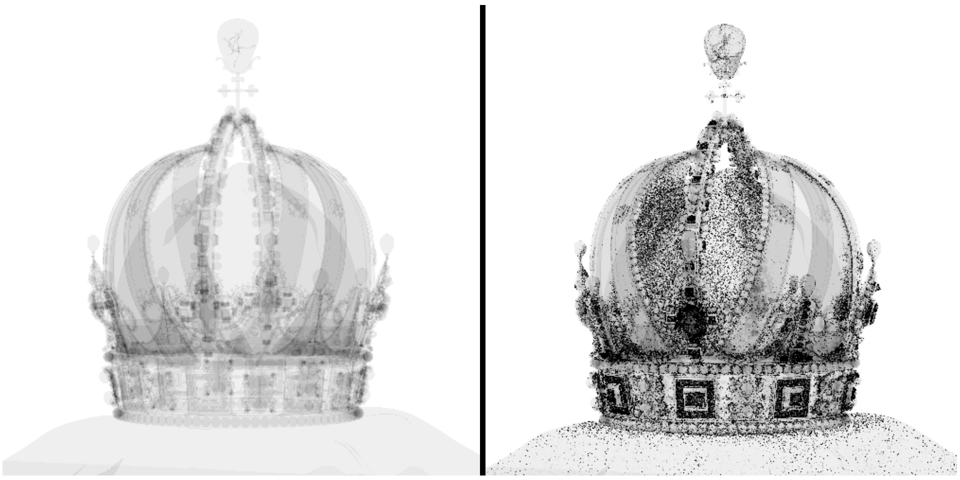

Figure 4.3: Number of intersections. Left: Phong, right: pathtracer

Image 4.3 compares number of intersections of both display modes. The darker the pixel is, the more intersections were found for the pixel. The noise in the right picture is due to the random ray generation.

At this point I would like to explain, why render time of the crown itself is 891 ms, which is almost four times more than the reference time (238 ms). Both are rendered with the same settings of path tracer. The difference is that the Embree pathtracer only finds the first hit, which is in the vast majority of cases the first or the second intersection tested. My implementation needs to collect all hits, which more than doubles my render time. Without rejecting hits in filters, my render time for the crown is 373 ms. The rest of the difference is the cost of saving intersections and creating intervals.

4.2 Asian Dragon

The second tested scene was Asian Dragon created by Stanford University Computer Graphics Laboratory. This scene has 7 349 978 triangles and the Embree reference time is 205 ms. For this test were used the same settings and test cases are also the same. Test cases are shown in figure 4.4.

Table 4.3 and table 4.4 are showing results of the measuring for both display cases. In both cases data can be explain in the same way as for the crown.

	#OR/ray	#SUB/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Cylinders: 0	0	0	0	135	1.9482
Cylinders: 10	0.10	0.06	0.16	157	1.6697
Cylinders: 100	2.57	0.28	2.85	296	0.8856
Cylinders: 1000	28.42	0.36	28.78	1 826	0.1436

Table 4.3: Asian dragon, rendering mode: Phong, triangles: 7 349 978, resolution: 512x512, pathtracer reference time: 205 ms

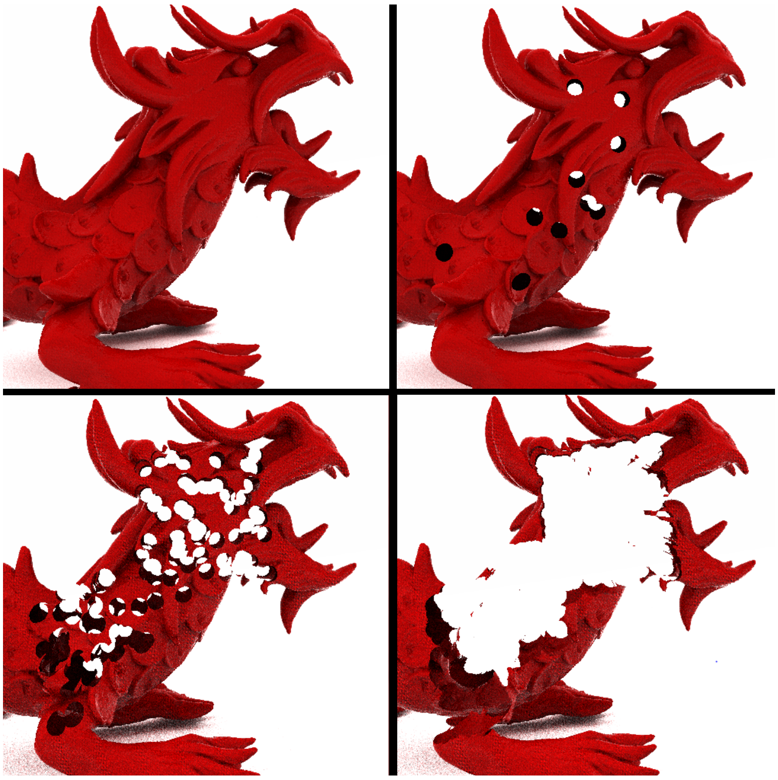

Figure 4.4: Test cases of the Asian Dragon.

	#Rays	#OR/ray	#SUB/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Cylinders: 0	577183	0	0	0	365	1.5813
Cylinders: 10	578396	0.11	0.07	0.17	397	1.4569
Cylinders: 100	565 760	2.65	0.24	2.89	750	0.7543
Cylinders: 1000	463 290	27.16	0.29	27.45	3 322	0.1395

Table 4.4: Asian dragon, rendering mode: pathtracer, triangles: 7 349 978, resolution: 512x512, pathtracer reference time: 205 ms

4.3 Happy Buddha

The third tested scene is Happy Buddha statue also created by Stanford University Computer Graphics Laboratory. This scene has 1 087 474 triangles and pathtracer reference time is 41 ms.

This scene is made from a 3D scan and it is a manifold. In this test, it was traced as a solid, that is without virtual points. While searching for all ray intersections in previous scenes slowed down the algorithm, in this

Figure 4.5: Path tracer render of Happy Buddha statue.

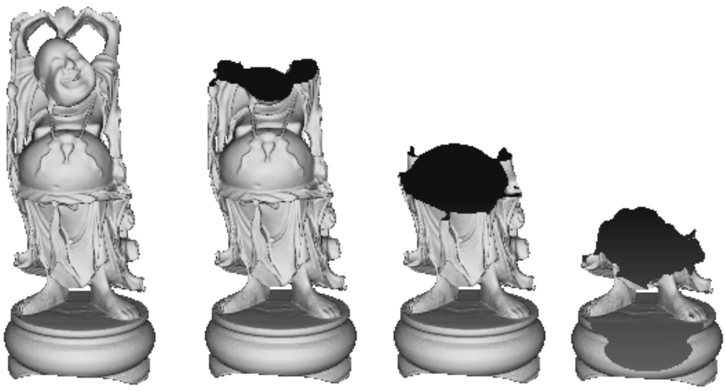

Figure 4.6: Test cases from left to right: 0%, 25%, 50%, 75%.

case there are mostly only two intersections with a ray. The render time of the scene without any additional primitives is 94 ms.

Testing cases are different for his scene. Instead of cylinders, there is only one cube, which moves in the scene and covers Happy Buddha statue. The cube is subtracted from the statue. The independent variable is how much of the status is "cut out" by the cube. Test cases are shown in figure 4.6

In table 4.5 and table 4.6 are measured results. The first column is how much of the statue was approximately covered by the cube. Measured render times grow with higher coverage, but drop as the statue gets covered entirely. This is caused by faster no hit evaluation. Tree traversal returns number of intervals in the result and if it returns zero, the result is not retrieved and returned. Additionally, there is no color evaluation, which cases path tracer to lower render times earlier.

	#SUB/ray	Render time [ms]	Trace speed [MRay/s]
0%	0	34	7.71
25%	0.03	40	6.55
50%	0.06	52	5.04
75%	0.08	56	4.68
100%	0.08	46	5.70

Table 4.5: Happy Buddha, rendering mode: Phong, triangles: 1 087 474, resolution: 512x512, pathtracer reference time: 40.96 ms.

	#Rays	#SUB/ray	Render time [ms]	Trace speed [MRay/s]
0%	350 713	0.001	93	3.77
25%	349 395	0.043	97	3.60
50%	328 753	0.085	106	3.10
75%	306 053	0.106	94	3.26
100%	262 144	0.083	60	4.37

Table 4.6: Happy Buddha, rendering mode: Pathtracer, triangles: 1 087 474, resolution: 512x512, pathtracer reference time: 40.96 ms.

Figure 4.7: CSG scene: Biplane.

4.4 CSG models

After testing performance on scenes with triangle meshes, this last test uses purely CSG scenes. All models in this part are courtesy of doc. Dr. Alexander Wilkie from Charles University in Prague. Three scenes were used for tests, Biplane, Snail and Rotonda. Biplane is the smallest one of them, with only

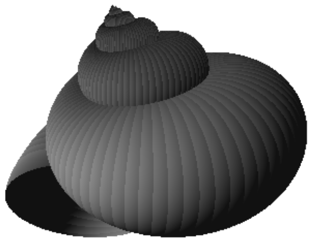

Figure 4.8: Scene Snail.

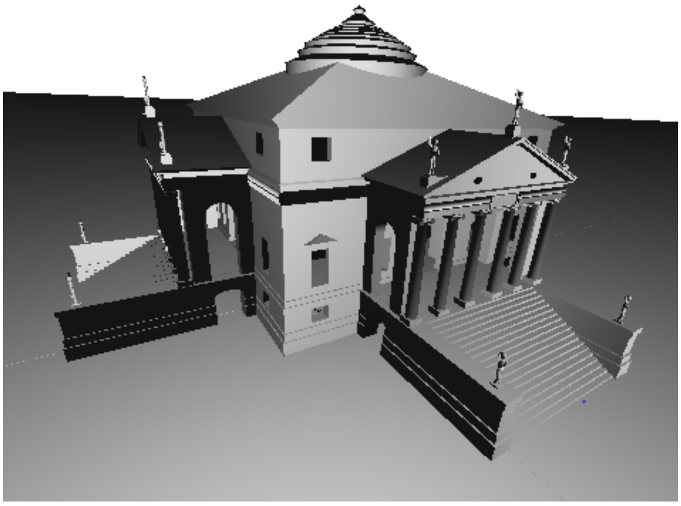

Figure 4.9: Scene Villa Rotonda.

336 primitives. Snail consists of spheres subtracted from each other creating a snail shell. This scene has 2 144 primitives and it is lopsided. The last one is the Villa Rotonda and it has 1 255 primitives.

	#OR/ray	#SUB/ray	#AND/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Biplane (336 primitives)	1.036	0.234	0.001	1.271	243	0.927
Snail (2 144 primitives)	10.107	0.312	0	10.419	1 980	0.132
Rotonda (1 255 primitives)	4.096	0.746	1.386	817.853	2 746	0.095

Table 4.7: resolution: 512x512.

	#Rays	#OR/ray	#SUB/ray	#AND/ray	#all/ray	Render time [ms]	Trace speed [MRay/s]
Biplane (336 primitives)	799 762	4.26	0.72	0.02	5.00	2 173	0.368
Snail (2 144 primitives)	934 637	27.83	1.03	0	28.86	20 812	0.045
Rotonda (1 255 primitives)	1 446 151	7.09	1.95	4.49	13.53	18 371	0.787

Table 4.8: resolution: 512x512.

4.5 Test summary and optimization proposals

Since tests did not bring results as expected, I looked more closely at running times of Embree ray tracing times and my tree traversal. The current ratio measured for Biplane is approximately 1:3. Embree traced a ray in 0.0055 ms on average. To make this part run faster there are two possible optimizations. First, better intersection tests with quick tests before computing the point of the intersection. Second, not using instances of scenes, but have all geometries in one scene.

Instances are very useful and easy way to implement CSG, however, Embree documentation does not give any additional information about how the BVH tree is build. It is ambiguous how BVH is built in case of instances and therefore it might be less effective than having all geometries in one scene. This would require creating a geometry for each primitive, store transformations and apply them manually before every bounds, intersection and occluded test.

However, traversing CSG tree took three times longer and therefore there is more space for optimization. For example creating own structure to store intervals instead of std::list, or using cache-friendly implementation of CSG trees.

Chapter 5

Conclusion

This work explored ray casting in combination with CSG and proposed a way how to exploit Embree for CSG ray tracing. I was able to implement the CSG support in Embree successfully. As expected, the cost for extending Embree was significantly higher render times due to collecting all intersections and evaluating CSG trees.

At the end of the previous chapter, I suggested possible improvements for the future work. First, it is possible to use geometries instead of instances to guarantee Embree builds its internal structure effectively. Since the tree traversal took three times as long as collecting intersections, further optimization is of this part is also possible. For example using tree priming or other methods excluding parts of the tree from traversing.

This work shown how it is possible to use Embree for CSG ray tracing and provided its results. While there is still space for optimization, it is left for any future work that would follow this thesis.

Bibliography

[1] ZAJÍČEK, Petr *Acceleration of Ray-Casting for CSG scenes.* Master thesis, MFF UK, 2012.

[2] APPELl A. *Some techniques for shading machine renderings of solids.* AFIPS Conference Proc. 32 pp.37-45, 1968

[3] WHITTED Turner *An improved illumination model for shaded display.* Proceedings of the 6th annual conference on Computer graphics and interactive techniques, 1979

[4] *Monte Carlo Ray Tracing*, Siggraph 2003 Course 44. Available on: http://www.cs.odu.edu/ yaohang/cs714814/Assg/raytracing.pdf

[5] ŽÁRA, Jiří. *Moderní počítačová grafika.* 2., přeprac. a rozš. vyd. Praha: Computer Press, 2004, 609 s., ISBN 80-251-0454-0.

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

[7] AMANATIDES John, WOO Andrew. *A Fast Voxel Traversal Algorithm for Ray Tracingl* , Eurographics 1987.

[8] REVELLES J., URENA C., LASTRA M. *An Efficient Parametric Algorithm for Octree Traversal.* pp. 212-219, WSCG'2000 conference, February 2000.

[9] MANEEWONGVATANA, Songrit, David MOUNT, *It's okay to be skinny, if your friends are fat.* Department of Computer Science, University of Maryland, December 18, 1999

[10] HIJAZI, Younis, Aaron KNOLL, Mathias SCHOTT, Andrew KENSLER, Charles HANSEN, Hans HAGEN. *CSG Operations of Arbitrary Primitives with Interval Arithmetic and Real-Time Ray Tracing* SCI Institute, University of Utah, November 19, 2008

[11] *Embree:High Performance Ray Tracing Kernels* [online]. Available on: https://embree.github.io/

[12] WOOP, Sven. *Embree Ray Tracing Kernels* [online]. Available on: https://embree.github.io/data/embree-siggraph-2015-final.pdf

[13] WALD, Ingo, Sven WOOP, Carsten BENTHIN, Gregory S. JOHNSON a Manfred ERNST. *A Kernel Framework for Efficient CPU Ray Tracing.* ACM Transactions on Graphics (proceedings of ACM SIGGRAPH). 2014.

[14] ROSSIGNAC, Jarek. *Blist: A Boolean list formulation of CSG trees.* GVU Center Georgia Institute of Technology, 1999