Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Petr Zajíček

Acceleration of Ray-Casting for CSG scenes

Department of Software and Computer Science Education

Supervisor of the master thesis:	Dr. Alexander Wilkie
Study programme:	Computer Science
Specialization:	Computer Graphics

Prague 2012

I would like to take this chance and thank my supervisor Dr. Alexander Wilkie, for all the help and support he gave me, while working on this thesis.

Název práce: Acceleration of Ray-Casting for CSG scenes

Autor: Petr Zajíček

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Dr. Alexander Wilkie, Kabinet software a výuky informatiky

Abstrakt: Metody pro urychlení sledování paprsku jsou obvykle použité při práci se scénami které jsou definované trojúhelníkovými sítěmi.Tyto trojúhelníkové sítě však obsahují velké množství trojúhelníků. Na rozdíl od trojúhelníkových sítí obsahují CSG scény malé množství komplexních primitiv. V této práci představíme metodu Operační KD-strom. Tato urychlovací metoda aplikuje metodu KD-stromů — jedna z moderních urychlovacích struktur vyvinutých pro trojúhelníkové sítě — přímo na CSG scénu. Předpoklad je že velké snížení počtu primitiv urychlí renderování, pokud se místo trojúhelníkových sítí použije CSG scéna.

Klíčová slova: Urychlení sledování paprsku, KD-strom, CSG scéna

Title: Acceleration of Ray-Casting for CSG scenes

Author: Petr Zajíček

Department: Department of Software and Computer Science Education

Supervisor: Dr. Alexander Wilkie, Department of Software and Computer Science Education

Abstract: Ray tracing acceleration methods are usually applied to scenes defined by triangle meshes.These scenes contain a large number of triangles. In contrast, CSG scenes contain orders of magnitude less more complex primitives primitives. In this thesis we will present the Operation KD-tree. This acceleration method applies the KD-tree — modern acceleration method developed for triangle meshes — directly to the CSG scene. This is done on the premise, that the huge reduction in primitive count will yield enhanced performance, when rendering a scene using CSG instead of triangle meshes.

Keywords: Ray-casting acceleration, KD-tree, CSG scene

Contents

Introduction

0.1 Preface

Ray tracing is a rendering technique used in computer graphics. This technique has a wide field of applications and is constant subject of research. In most of its application to date it uses triangle meshes to define the geometry of the scene it is rendering. These triangle meshes are comprised of a huge number of triangles. Common scenes can be built out of over 100K triangles. To allow efficient rendering of these numbers of triangles sophisticated acceleration structures have been developed. One of the more most successful acceleration structures is the KD-tree. Since its introduction significant research was done into its application and optimization. There are other acceleration structures besides the KD-tree that are used to increase the rendering performance of ray tracing. Bounding volume hierarchies (BVH) and grids are such structures. Each of the three structures mentioned have different properties that make them more suitable in different usage cases.

Triangle meshes are however not the only way to describe the geometry of a scene. Constructive solid geometry (CSG) is an alternative way of scene description. It describes a model as a result of boolean operations on a given set of primitives. CSG is commonly used for modeling scenes and objects, however it is rarely used as a direct scene description for ray tracing. In most cases an object modeled using CSG is converted into a triangle mesh and the mesh is than ray traced. This is mostly done, because handling a scene with one type of primitive, the triangle, is more convenient and arguably also more efficient in many usage cases. The research presented in this thesis seems to indicate that this assessment is actually not as clear-cut as once thought. Still, at least in the past, this widespread assumption seems to have led to less research being invested into making ray tracers using SCG as efficient as the ones using triangle meshes.

There is a potential advantage to using CSG. The number of primitives used to describe a scene using CSG is orders of magnitude lower than the number of triangles needed to describe the same scene. Still, real-life CSG scenes contain a relatively high number of primitives. Considering that ray-primitive intersection tests for these primitives are generally more costly than the ray-triangle intersection test, an acceleration structure is still warranted.

The goal of this thesis is to explore the possibility of adapting the acceleration structures developed for triangle mesh ray tracing to serve as an acceleration structure for CSG ray tracing. The most problematic part of this adaptation is to make the acceleration structure work with the boolean operators present in CSG. These operators make CSG a powerful modeling tool, however they do not work well with acceleration structures.

To achieve this goal an adaptation of the KD-tree acceleration structure will be show. The choice to explore this option was made based on the success of the KD-tree as an acceleration structure. Also the implementation of the basic form of KD-tree is relatively simple, while it provides high performance benefits, which makes it perfect for testing purposes.

0.2 Outline

This thesis consist of 4 chapters.

In the first chapter is an introduction into ray tracing. It explains the key differences between triangle meshes and CSG and introduces some of the more commonly used acceleration structures with emphasis on KD-trees.

The second chapter describes the adaptation of the KD-tree for CSG ray tracing. It introduces the Operation KD-tree (OKD-tree). An adaptation of a simple optimization for the KD-tree is also described here.

In the third chapter the need to include triangle meshes into CSG as a primitive is discussed.

The fourth chapter deals with testing of the proposed algorithm. Test scenes are discussed one-by-one showing the strengths and weaknesses of the algorithm.

1. Ray tracing

1.1 What is ray tracing

Ray tracing is one of the image synthesis techniques used in computer graphics. It generates the image by tracing the path of a light-ray. This technique is capable of producing images with very high degree of visual realism. In almost all cases the images created by ray tracing look more real than those created by z-buffer or scan-line rendering methods. The drawback of the ray tracing method is that it has high computational costs compared to some other methods.

Ray tracing constructs the image by tracing the path of a a light-ray sent from the camera into the scene trough the pixels of the rendering plane. The rendering plane is the actual image that is being constructed. Sending the ray from the camera into the scene might seem contra-intuitive since in reality the light travels from the scene into the camera. However, the vast majority of the rays reflected by the scene never hit the camera. Reversing the direction of the followed rays eliminates this problem.

Rendering an image using ray tracing has the following phases:

- The ray is tested for intersection with the scene. If there are any intersections the nearest one to the camera needs to be found, because this would be the point visible from the camera. The basic way to find the intersections between the ray and the scene is to do a ray-object intersection for every object in the scene. This has a high computational cost, forcing the ray tracer to spend the overwhelming majority of rendering time doing intersection tests.

- When the nearest intersection to the camera has been found, the lighting calculations are done. The reason for this is to estimate the incident light from the intersection point. Different methods of doing lighting calculations range in their complexity. Generally, more realistic images require more complex lighting calculations.

  Almost all lighting methods use secondary rays sent from the intersection point to complete the lighting calculation. These rays are used to determine shadow, specular reflections, refraction, etc. If true realism is required, the secondary rays would be used to execute a monte-carlo approximation of the rendering equation. This equation is an integral equation that determines the equilibrium radiance leaving a point. It was simultaneously introduced into computer graphics by David Immel et al. [6] and James Kajiya [7] in 1986.

  The need to send secondary rays to complete the lighting calculation means that there is a recursive property inherent to ray tracing. Again, more realistic images require deeper recursion.

- Finally, based on the incident light the pixels of the rendering plane are colored.

The recursion inherent to ray tracing can be controlled in different ways. The simplest way is to use a non-adaptive method and stop, when the recursion has

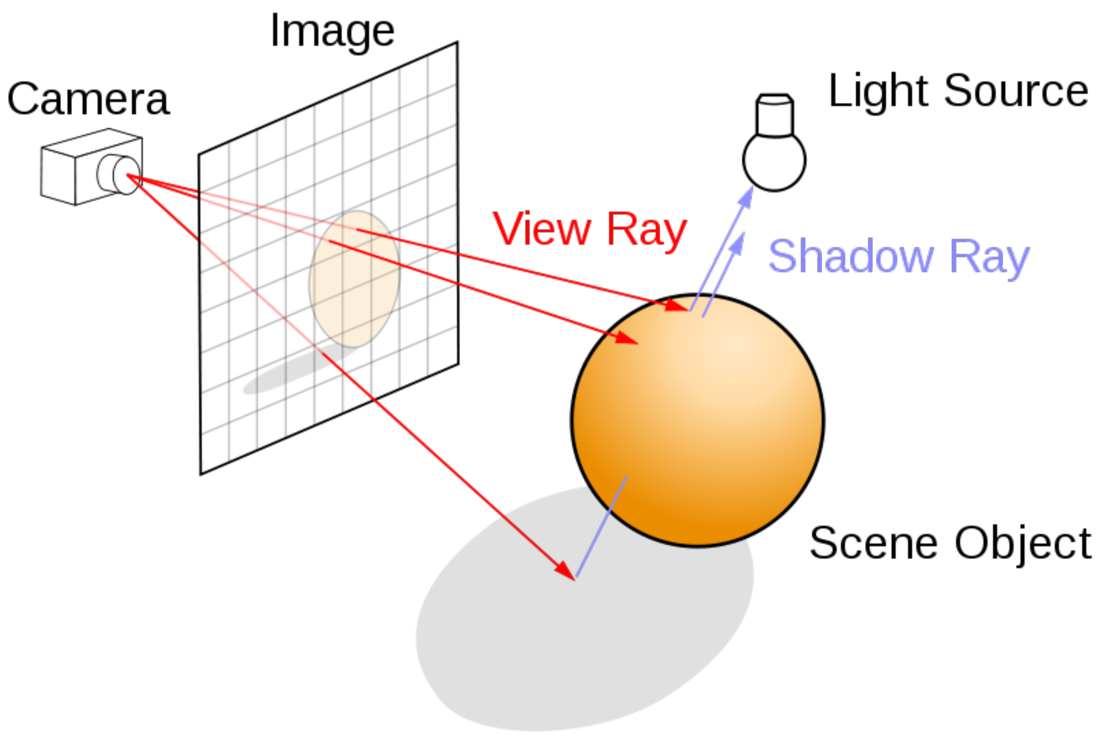

Figure 1.1: A diagram demonstrating ray tracing algorithm [15]

reached a certain fixed depth. A more sophisticated way stops the recursion when the importance of the secondary rays gets below a given threshold. The importance of the secondary rays is determined by the lighting method. Usually reflective surfaces assign high importance to the secondary rays sent from them, while diffuse surfaces assign less importance. This adaptive method still requires a maximum recursion depth limitation, otherwise for very reflective surfaces it might allow an extremely deep recursion.

Ray tracing has several advantages. The methods used to estimate incident light from a given intersection create a realistic image. Effects that are difficult to get from other rendering methods such as shadows, reflections, transparent and translucent materials etc. are a natural result of the ray tracing algorithm. However ray tracing has a significant drawback. The recursive nature of the algorithm adds up to a very high computational complexity. These properties of ray tracing make it an ideal method for applications where the image is rendered ahead of time and great demands are placed on its accuracy.

1.2 Scene descriptions for ray tracing

There are several ways to describe the a scene for ray tracing. In this thesis we will discuss the triangle mesh and the CSG tree.

1.2.1 Triangle meshes

Triangle meshes are the most commonly used method for describing the scene geometry. They are a boundary representation structure in which the boundaries of objects are represented by a high number of triangles. In computer graphics, triangle meshes are preferred for several reasons. One of the major reasons is that the processing complexity of triangles is relatively small compared to any higher order primitives. Also, having a scene where there is only one kind of primitive is advantageous. Graphics hardware can be optimized to have fast triangle processing. One can utilize parallel processing to a very high degree with the assumption that every object in the scene can be handled the same way. Using novel methods for storing triangle meshes and various level of detail implementation strategies make triangle meshes a practical way to describe a scene.

Triangle meshes have some properties that are interesting form the perspective of this thesis.

- Every face in the triangle mesh is actually part of the resulting scene. A face can be occluded by another face, however baring occlusion there are no instances where a ray intersection with a given face would not create a shaded point. A direct consequence of this is the fact that one can stop testing for intersections once the nearest intersection is found.

- To create a good looking model using a triangle mesh, the number of triangles has to be high. An obvious example for this is the sphere. To create a convincing approximation of a sphere using triangles one needs to tessellate the sphere into a lot of small triangles.

- The number of triangles needed to create a convincing approximation of any object is dependent on the circumstances under which the object is viewed. A specular object will need more triangles than an object that has a fully diffuse material. An object viewed from far away need not be so finely tessellated as an object that is close. This property of the triangle meshes forces one to consider implementing level of detail strategies to counter the rising number of triangles.

The second and third items mentioned on the list above have inspired the goal of this thesis. The first item is mentioned because it is the main problem faced when trying to use a different scene description method.

1.2.2 Constructive Solid Geometry

CSG is a method of constructing complex geometry by combining simple primitives using boolean operators.

Modeling using CSG has some useful properties.

- Checking whether the model is a solid is very easy. As long as one uses solid primitives, the models they create are also solid (water-tight). In contrast modeling through some kind of boundary representation like meshes might require consistency checks to make sure the resulting model is a closed solid.

- Determining whether an arbitrary point is inside a model is just a matter of determining this for every primitive used and running the boolean operations.

- No level of detail consideration needed. Finding ray intersections with these primitives is done analytically. This is important, because it means that no matter the scale one is working on, a precise intersection can be calculated. This eliminates the need for level of detail considerations, which are present when working with triangle meshes that are just approximations of an object that is modeled.

- Complex models can be described using only a small number of primitives. This is made possible by using boolean operations to combine different primitives. In fact for many scenes the number of primitives required to create them using CSG is orders of magnitude smaller than the number of triangles required to model the same scene using triangle meshes. Fewer primitives in the scene means fewer intersection test are executed during the ray-scene intersection test, which is beneficial for the rendering performance. This property of the CSG scenes is the focus of this thesis.

The above mentioned properties of CSG modeling are the reason why some applications use this kind of scene description for creating models of complex objects. For the actual rendering process however, the models created this way are usually tessellated and only the resulting triangle meshes are rendered. In this thesis we discuss the case when the CSG model is rendered directly.

CSG models use several different primitives. Every application has its own set of primitives. We use the following primitives:

- Sphere
- Cube
- Torus
- Cylinder
- Cone
- Hyperboloid
- Paraboloid

The operations used in CSG modeling are shown in figure 1.2.

- OR operator is the union of two objects.
- AND operator is the intersection of two objects.
- SUB operator is the difference of two objects.

The CSG geometry is described by a tree structure, the semantic scene graph. The leaves of this tree hold the primitives, while the intermediate nodes are either the boolean operations or different attributes. The most commonly used attributes are:

- Transformations. (rotation, translation, scaling etc.)
- Materials descriptions

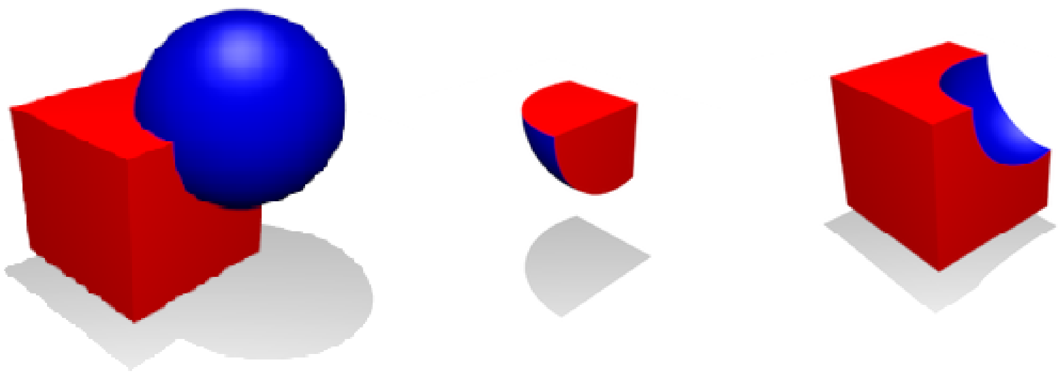

Figure 1.2: Boolean operations in CSG. Left to right: Union, Intersection, Difference [13, 12, 14]

- Surface descriptions (color, texture etc.)

The attributes are applied to the primitives in the subtree defined by the attribute node. Some attributes, like the the transformations are combined, while others supersede each other (a primitive should have only one material). In the case of attributes that supersede each other the one closest to the primitive is applied.

To execute the ray-scene intersection test the semantic scene graph is traversed in a dept-first manner. During traversal the attributes are applied. When a primitive is reached it is intersected and an intersection list is generated. This intersection list is then combined in in the operation nodes with the intersection lists generated by the other primitives. The intersection list that reaches the root of the tree will contain the intersections with the whole scene. The rules by which the intersections are combined are shown in figure 1.3.

Unfortunately ray tracing a CSG-tree directly gives rise to a problem that not all intersections with primitives yield a point that can be shaded. An example would be when the ray hits an object that is subtracted from another object. In this case the intersections on the subtracted part are going to be eliminated during the combination in the subtraction node. The direct consequence of this is that one might need to continue the intersection tests even after the nearest intersection to the eye or camera has been found.

1.3 Acceleration structures

As described the basic ray tracing algorithm needs to find the first intersection point with the scene. The naive approach of this is to intersect all the primitives in the scene with the ray, rank the intersections and chose the first one. This naive approach has performance issues. Test have shown that the most time consuming operation in ray tracing is indeed finding the intersection with the scene. The cost of executing an intersection test for each one of the primitives in the scene gets unreasonably high with the growing number of primitives in the scene. Considering that a finely tessellated triangle mesh scene can have well over 100K triangles this problem needs solving.

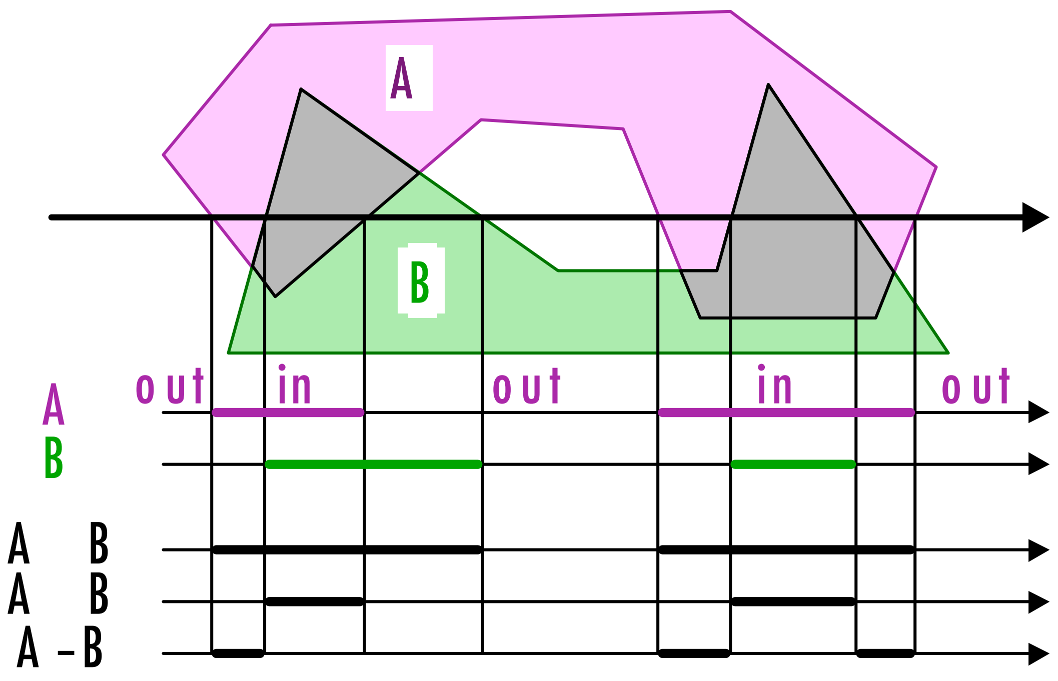

Figure 1.3: Combining Intersections in CSG

There are two ways of approaching the problem. The first one would be to decrease the number of primitives in the scene. This could be done by directly ray tracing the CSG scene, however even the relatively small number of primitives in the CSG scene can be problematic. More so when one considers that for some of the primitives used in CSG the intersection test involves more complex calculations than for a simple triangle.

The second way to make finding the intersection between the ray and scene more efficient is not to test every primitive in the scene. In every scene most of the primitives do not intersect the ray. There are ways to classify the primitives in a manner which makes it possible to decide for a large number of them whether any of them needs to be considered for intersection testing, using only one intersection test. The idea is to group primitives together according to some topological criteria. When this is done the whole group could be surrounded by a bounding volume (BV). When the ray-scene intersection test is being executed, this BV is tested for intersection before the primitives within are tested. If the ray does not intersect the bounding volume of the clump there is no reason to test the individual objects in the group.

There are several structures that do just this. These acceleration structures have been researched in detail in the past. Most of these structures have been developed for use with triangle meshes and are optimized for such use. In the following section a few of these structures will be described.

1.3.1 Bounding Volume Hierarchies

BVH is a tree structure. In this structures the primitives of the scene are surrounded by a BV. These BVs are the leaves of the tree. The inner nodes of the

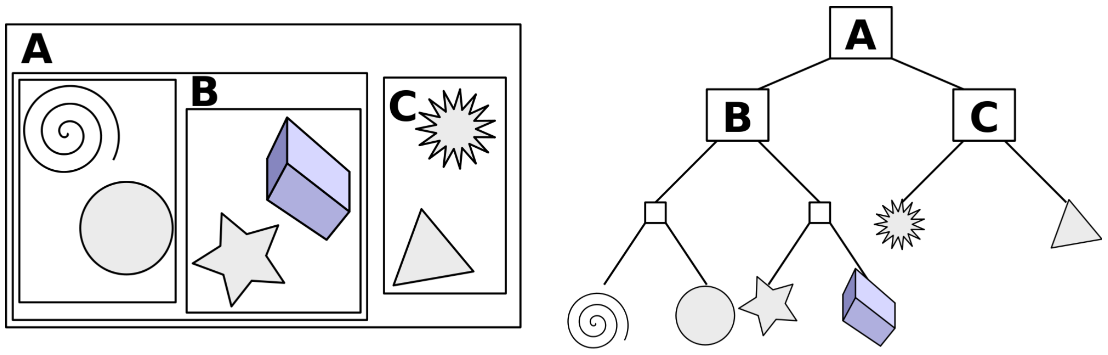

Figure 1.4: Example of bounding volume hierarchy (BVH) in two dimensions, where bounding volumes are AABB [16]

tree are BVs that encompass the BVs of their children. This results in the root being the BV for the entire scene. A BHV is shown in figure 1.4.

In this structure when a ray is intersection tested against the BV of an internal node of the tree and misses it, the children of this node need not be tested for intersection.

The matter of what kind of bounding volume is to be used is a matter of finding a balance. On the one hand a bounding volume that encompasses the primitive as tightly as possible would eliminate more intersection tests. On the other hand the intersection test for the bounding volume should be as cheap as possible and also the memory required to store the parameters of the BV should be as small as possible. The axis aligned bounding box (AABB) is good candidate for such a BV. Most implementations of BVHs use AABBs as bounding volumes.

There are several ways to build the BVH. Top-down building starts with one bounding volume for the entire scene. The primitives in the scenes are divided into groups — mostly two groups, if a binary tree is used — according to some criteria. BV for the groups are calculated and assigned as children to the node that is being divided. The building algorithm is called recursively on the children, each of them containing one of the groups.

Bottom-up methods start from the other end. In this case one starts with all the leaves and starts to group them together until one single node is created for the entire scene.

Top-down methods are more commonly used because they are easier to implement, however they might yield less efficient trees. The efficiency of the tree is greatly influenced by the strategy used to group the objects. Some of the grouping strategies will be mentioned later, because they are also used by other adaptive acceleration structures.

A BVH is an adaptive acceleration structure in the sense that the tree adapts to the geometry of the scene. If a good grouping strategy is used, dense areas of the scene produce their localized subtrees on higher levels of the tree. These dense areas which could be standing apart of other dense areas are then eliminated from ray intersection tests in the first few traversal steps through the tree.

A BVH groups its objects so that they belong into exactly one child of a node. This attribute of the BVH makes it possible to estimate of the memory cost of the tree which is useful. This same attribute however means that it is possible for the child nodes of a given node to overlap significantly. Generally one would traverse the child that has been intersected first by the ray before the other child. If the child nodes do not overlap any intersection found in the first child would be the closest intersection to the camera. If the children do overlap this assumption does not hold, thus one must traverse the other child too, if it is also intersected by the ray. This precludes any early exit possibilities.

The traversal algorithms for BVH tend to be slower than for other acceleration structures just because a the ray-AABB intersection test is more costly than a ray-plane intersection test used by the KD-tree. Also storing the whole AABB is more costly than storing a KD-tree node.

The BVH is mentioned here despite some of its flaws because a sort of BVH is very easily implemented into SCG ray tracing and so forms a natural comparison for the novel acceleration structure introduced in this thesis.

1.3.2 Surface Area Heuristic

There are several grouping strategies. Primitives are usually sorted according to their centroids. One of the simplest way to divide them into two groups is to do a median split. This grouping is however very inefficient. In most cases it fails to isolate empty space. Isolating empty space close to the root of any tree based acceleration structure stops the traversal of rays that miss the scene geometry as soon as possible. To create a more efficient splitting strategy this consideration was taken into account.

One of the most widely used grouping strategies is the surface area heuristic (SAH) [2, 9, 4]. This heuristic considers the geometry of the node that is being partitioned, and tries to chose the split that has the lowest cost of ray tracing. In order to calculate the cost of a given split one makes the following assumptions:

- The rays are uniformly distributed, they are infinite and go trough the node being partitioned.
- The cost of traversing a node C_t and the costs of intersecting the primitives C_p are known.

With these assumptions in mind the cost of a given split is the cost of intersecting the child nodes multiplied by the probability of hitting the child node plus the traversal cost.

$$C = C_t + P_1 * C_1 + P_2 * C_2 \tag{1.1}$$

The probability of hitting a child node when the parent node was hit is the ratio between the surface area of the child node and the parent node. The cost of intersecting a child node is the sum of the cost of intersecting the primitives in it. For triangle meshes where all primitives are triangles it's a simple multiplication.

$$C = C_t + \frac{SA(B_1)}{SA(B)} * N_1 * C_p + \frac{SA(B_2)}{SA(B)} * N_2 * C_p \tag{1.2}$$

The SAH can be also used to determine when to stop subdividing groups. If there is no split or grouping that has a lower cost than the intersection cost of the current node, one can stop the subdividing process.

With the use of the SAH the resulting acceleration structure can be very efficient, however the implementation of the SAH might be costly. In its basic form it calls for considering every possible grouping of primitives. However this can be avoided. In the section about KD-trees one such implementation will be shown.

1.3.3 Uniform Grid

Grids are a simple space partitioning data structures. They are discussed here to introduce the idea of space partitioning.

Space partitioning structures do not group the primitives as BVHs do. These structures divide the scene into areas while for every area they maintain a list of objects that are at least partially in this area. This yields two major differences. Unlike with BVHs the areas of the space partitioning structure do not overlap. The primitives however are not limited to be part of only one area.

As a consequence if the space partitioning structure can be traversed in the direction of the ray the first intersection achieved is the closest intersection to the camera. This gives opportunities for early acceleration structure traversal termination.

Uniform grids partition a the space into a uniform grid with a predetermined resolution. Building a grid structure is then a simple matter of finding the cells with which a given primitive overlaps and assigning the primitive to these cells. Also the traversal algorithm developed for the uniform grid is extremely fast. These two attributes are the main advantages of grid-like structures; fast building and traversal.

Basic grids are however non-adaptive structures. This meas that their resolution is given beforehand, and usually does not change during their construction. Because of their non-adaptive nature grids are susceptible to the "teapot in the stadium" problem. This problem arises when the scene has a high density or geometrically complex areas separated by relatively empty space. In this case the empty space will force the grid to have either a too fine resolution, thus larger primitives will be present in many cells to reasonably manage, or too coarse a resolution to partition the primitives in the high density areas.

Still, grids and their extensions are used for applications where the need for rebuilding the acceleration structure arises. These applications include dynamic scenes.

1.3.4 KD-tree

KD-trees are adaptive binary space partitioning tree structures. The root of the tree corresponds to the AABB of the whole scene. Each internal node of the tree introduces an axis aligned splitting plane that splits the current area into two non-overlapping areas, which are then considered the children of the node. The leaves nodes of the tree hold a list of scene primitives that are at least partially present in the area of space corresponding to the leaf.

The ray-scene intersection is solved by the KD-tree in the following manner. Assuming a node gets hit by the ray the traversal continues in one of two ways.

If the node is a leaf node, the list of the objects belonging to the leaf node is tested for intersections. The closest intersection inside the area is chosen and returned.

For internal nodes the child node bounding boxes are tested for intersection. The child nodes that are hit are traversed recursively. This may be one or both of the child nodes. It is important to note that it is possible to determine which child node gets hit first. This child is the near child, the other child is the far child. If the recursive traversal is then started with the near child early traversal termination is possible as long as returning the first intersection is sufficient. This is possible because the intersection returned by the near child will be always closer to the camera than intersection returned by the far child.

For the KD-tree to work efficiently it has to be built well. A well built KD-tree should split off empty space as close to its root as possible. This is advantageous because all rays that go through this empty space are immediately qualified as returning no intersections.

A KD-tree is built using the top-down method. One starts with a bounding box for all the primitives and the list of the primitives. From here the subdivision process starts. At every level the subdivision process has the following steps.

- Determine if another split is necessary. If no further split is necessary a leaf node is created with the list of current primitives.
- According to a given splitting strategy a splitting plane is chosen.
- Create the child nodes.
- Sort the primitives into the child nodes according to their position in respect to the splitting plane.
- Recursively subdivide the child nodes.

A combination of criteria is used to determine whether to split a node. The basic criterion is a predetermined depth limit. Once the recursion depth reaches this level no more splits are created. There is also no reason to split empty nodes. A more complex termination criterion can be created using the SAH. When a new split is no longer profitable in terms of traversal cost the recursive subdivision is stopped.

To create a well built KD-tree the SAH is used predominantly to determine the split plane. The SAH in its basic form is quite costly to implement, however there are methods to implement it in O(N log N) complexity [11].

The main idea of this SAH implementation is that the only splitting planes worth to consider are the planes given by the bounding boxes of the primitives in the node being split. This is because the number of primitives on either side of the splitting plane changes only in these positions. These possible split planes can be sorted according to their position along the axis perpendicular to their plane. The possible split planes are also flagged to determine whether the split plane is the start of a primitive or the end. This way when the possible split planes are visited in order, the number of the primitives left or right of the split plane can be continually updated in constant time. The sorting of the possible split planes, is needed only at the beginning of the whole building process. The sorted list of possible split planes can be divided into the child nodes without breaking its ordering. This way the possible split list can be reused in the recursive subdivision.

A fast traversal algorithm for KD-tree has also been presented [3]. This traversal algorithm is based on the idea of having a simple and fast inner loop that does not use true recursion but a stack to emulate it.

The nodes of the KD-tree can also be stored very efficiently [3]. In the node of the KD-tree one has to store a flag determining whether the node is leaf, the orientation and position of the splitting plane, pointer to child node or the list of primitives. The efficient way to do this is:

- leaf node flag (1 bit)
- orientation flag (2 bits)
- array index of the child or primitive list (29 bits of a float)
- position of the split plane (32 bit float)

If the nodes of the tree are arranged in an array so that the children of a given node are always next to each other, only the index of one of the children need be stored. To store the index of the child only 29 bits are used to make place for the flags. The 29 bits for storing the index of the child node still leaves enough possible nodes. Using more nodes would fill the RAM anyway and the ray tracing would stall. The size of the resulting data structure is only 64 bits. This allows for a high number of nodes being loaded into the cache line of the CPU speeding up the traversal of the tree.

To sum up, the KD-trees has a fast traversal algorithm, performs efficient spatial split, thus speeds up ray tracing considerably. The building algorithm for the KD-tree is still not suited for real-time rebuilding but creates a tree that can be reused for viewing the scene from arbitrary angles as well as tracing rays of general direction or origin.

Much research was done into KD-trees. There are several extensions to KD-trees such as multi-level ray tracing [10]. To date they are the most often used acceleration structure for ray tracing, however almost all the optimizations applicable to KD-trees use triangle meshes as the scene description method. The apparent success of the KD-trees when working with triangle meshes has prompted this thesis to investigate their usefulness while trying to accelerate direct ray tracing of a CSG scene.

2. Acceleration Structure for CSG

As described earlier a CSG scene has orders of magnitude less primitives than a triangle mesh. These primitives are however more complex shapes which in turn have more costly intersection tests and the CSG operations also require a more complex ray intersection acceleration structure that is inherently slower to traverse. This means that doing as few intersection tests as possible while finding the intersection between a ray and the scene is still a good idea. To fill the role of an acceleration structure that is capable to ensure that only the necessary intersection tests are done, we will investigate some of the acceleration structures discussed in the last section.

The acceleration structures developed and optimized for use with triangle meshes should be able to handle other primitives as well without extensive modification. There are however some problems inherent to the CSG tree which will require closer examination.

2.1 Native Bounding Volume Hierarchy

Building a BVH for the CSG scene is one of the ways to eliminate irrelevant primitives from the intersection process. This BVH is not a strict implementation of the method described earlier. It still groups primitives and creates BVs for these groups until in the root there is only one BV for the entire scene. AABBs are used as BVs in the implementation of this native bounding volume hierarchy (NBVH).

The main difference here is in the method of construction. For ray tracing purposes the CSG scene is stored in a tree structure. This tree structure can be conveniently used as the basis of the BVH construction. One would start with the bounding boxes for the leaves of the CSG tree. Every boolean operation would be considered a grouping of primitives, thus for every operation a new node in the BVH would be added. The children of this node would be the bounding boxes of the operands.

The semantic scene graph is a very versatile data structure. It allows for nodes that are not directly involved in the description of scene geometry to be attribute nodes in the graph. Such nodes would be the material or surface attribute, which do not modify the geometry as such and only supply data for the lighting calculations done when an intersection is already found. A different kind of additional data that can be inserted into the semantic scene graph could be precomputed optimization data. An example for this would be a node that stores the list of light sources in the scene, precomputed in some optimization step before ray tracing. The bounding boxes for the primitives and the operations could be inserted in the same manner directly into the semantic scene graph. The traversal algorithm for the semantic scene graph is only modified to include the BVH functionality. This means that when encountering a bounding box a test is done whether the ray actually hits this box. If the ray does not hit the box traversal in this branch is stopped.

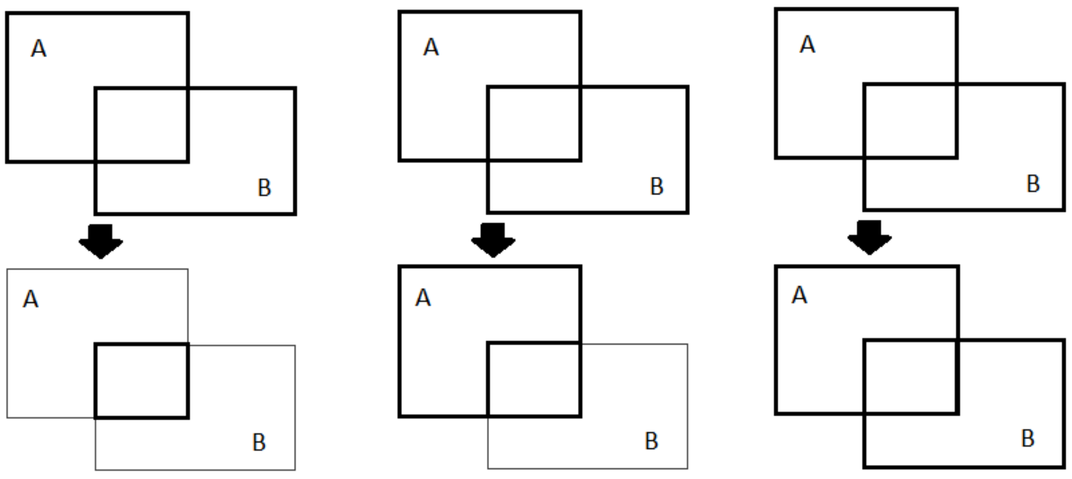

Figure 2.1: Illustration of bounding box clipping. Left to right: A and B, A sub B, A or B

A trivial optimization executed on the NBVH is to clip the bounding boxes according to the boolean operation. This means that the bounding boxes of the operands can be clipped so that only those parts that have a chance of producing relevant intersections are kept as shown in figure 2.1

- For the AND operation both operands bounding box would be clipped to the intersection of the two operands. Only in this area yield the AND operator non-empty results.
- For the SUB operation the left operand can be clipped to the part that intersects with the right operand.
- For the OR operation there is no clipping available.

This application of BVH has the advantage of being a natural byproduct of the CSG tree. Therefore implementing it is relatively easy. It does serve as a valid acceleration structure and in most cases it provides a significant performance enhancement.

There is however one drawback. The structure implemented this way is not a real adaptive structure. This is because it depends on the topology of the semantic scene graph, as it was modeled by the user. There are several cases when a semantic scene graph is created in a very unbalanced fashion. An example would be a procedurally created CSG model for a shell as seen in figure 2.2. A model like this creates a lopsided semantic scene graph, because the segments of the tree are strung together using a for loop. Using loops in the code creating the semantic scene graph is an intuitive way to create objects with repeating patterns. Code for such a shell is given in the attachments. In this case the depth of the tree forces the BVH to do intersection test with bounding boxes that could be avoided if a truly adaptive acceleration structure was used.

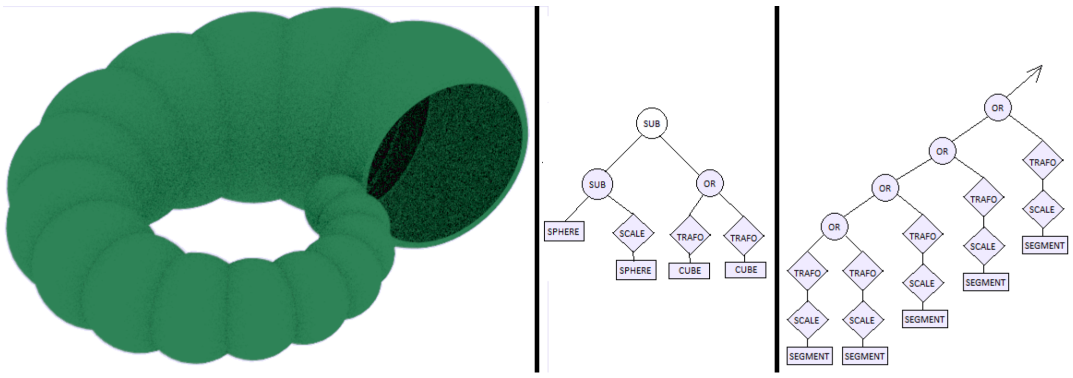

Figure 2.2: Procedurally created shell. Left to right: Rendered shell. Semantic scene graph for a segment of the shell. Lopsided semantic scene graph of the complete shell, the OR operation nodes have as the right operand only a shallow sub-tree, while the left operand is a deep tree with the same pattern repeating.

2.2 KD-tree

The non-adaptive nature of the natural application of BVH to CSG scenes has lead to investigating the possible use of a truly adaptive structures. Since KD-trees had been successful for triangle meshes. The choice was made to modify the basic KD-tree and adapt it to work in a CSG environment. This presents several challenges that need to be addressed.

A basic KD-tree is optimized to work for one kind of primitives. In the a CSG scene one works with several different primitives. This has influence on the building algorithm of the tree. The bounding boxes of the CSG primitives however, can still be used for the KD-tree construction. The bounding planes of the boxes are used as the split plane candidates. Even the bounding box clipping optimization introduced by the NBVH is still very useful. The intersection cost of a given primitive can be derived from the primitive directly or just use a global intersection cost for all primitives.

There is an inefficiency inherent in working with multiple types of primitives. Before the intersection test can be executed on the primitive its type has to be determined. This sort of polymorphism is exactly the thing one would not want in the core of a traversal algorithm. It slows down the KD-tree traversal by a fraction however, the massively reduced number of primitives in the scene should compensate for this inefficiency.

While the CSG primitives can be used as primitives for the KD-tree construction. It is important to note that the actual implementation of a semantic scene graph allows for transformations and other attributes in the nodes of the tree. These attributes have to be pushed to the leaves to make it possible to ray trace these leaves directly, without going trough the whole scene graph traversal. This is done by introducing a combined attribute node directly above the primitive.

Before the rendering an optimization step is executed on the scene graph, this "push attributes step" traverses the scene graph the same way as the ray would traverse the scene graph during ray tracing. The attribute nodes along the way are accessed and executed, which means the state of the governing structure of the traversal is changed accordingly. However, when a leaf node (a primitive) is reached instead of executing the intersection test, the state of the traversal (transformation, material, surface etc.) is saved into a combined attribute node. This combined attribute node is then inserted directly above the leaf. The accessed attribute nodes are deleted from the semantic scene graph, because their effect is saved in the combined attribute node.

The combined attribute node is then used as the primitive in KD-tree construction. When an intersection request is invoked on the combined attribute it restores the attributes stored in it into the governing structure of the ray traversal. Then the intersection test is executed on the primitive itself. The returned intersection can now have the right material and surface attributes. This optimization is useful even if no KD-tree is used. It eliminates the need to access and execute attribute nodes during semantic scene graph traversal.

The more important difference is the fact that in CSG scenes not all intersections of the ray with the primitive yield points that are actually shaded. This fact not only precludes the use of early ray traversal termination, but introduces the problem of tracking the boolean operations that need to be applied to the intersections.

A KD-tree is a space subdivision that allows one to very efficiently traverse a ray through space that is populated by geometric shapes. It is not possible, though, to directly introduce CSG functionality into it, in the same way as a scene graph can add BVH functionality directly into its CSG nodes. The information about the boolean operations has to be accessed trough the semantic scene graph. Accessing this information can be tricky, because of the order a KD-tree would access the primitives in its leaves. A KD-tree inherently accesses the primitives in random order with respect to the semantic scene graph. Pairing the intersected primitives to execute the boolean operations is therefore not possible. This means that the depth-first search used to combine intersections in the CSG tree can no longer be applied. The boolean operations are however the essence of the CSG scene and they need to be executed somehow. Solution for this problem is presented in following sections.

The benefits of adapting a KD-tree to work in a CSG environment include not just the performance enhancement of the basic KD-tree. It would also prove that all the advanced techniques used to make KD-trees one of the most successful acceleration structures could be used for CSG scenes as well.

2.3 Multiple KD-trees

One way to solve the problem of tracking the boolean operations is to use multiple KD-trees inserted into the semantic scene graph as attributes. These attributes would hold the data structure of the KD-tree and respond to the ray traversal by doing their own traversal then return the intersections.

The idea is that every operation that combines intersection points in such a manner that they eliminate some of them is considered as leaf primitive itself. These new primitives are inserted into the building process of the KD-tree above them. Sub KD-trees are then built for the primitives in the subtrees defined by the operands of these operations.

The operations that need to be handled this way are the boolean intersection and boolean difference.

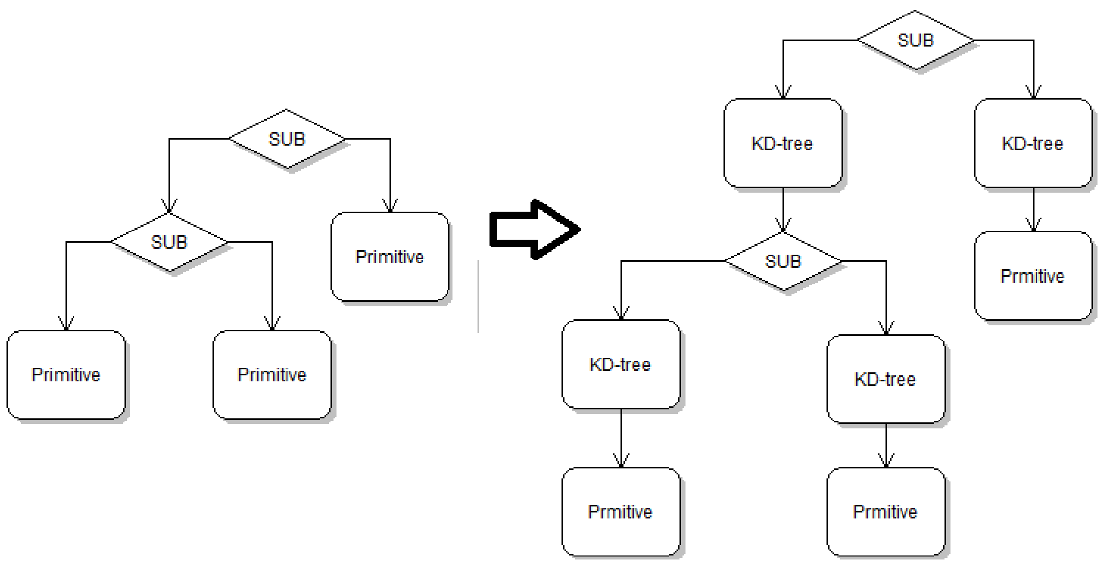

Figure 2.3: Left: A scene graph describing on object from which two different objects were subtracted. Right: The same scene after the KD-trees were added according to the Multiple KD-trees method.

Implementing this solution generally, creates problems. In many cases a boolean difference is done by subtracting one simple primitive from a compound object. The KD-tree created for the left operand of this operation would essentially be empty.

This solution of the operation tracking problem does work, but the number of KD-trees created for a realistic CSG model is high. For the shell model shown in figure 2.2 it would be 4 KD-trees for every segment. The memory requirements to maintain this high number of KD-trees seems unrealistic.

Other reasons why this solution is not optimal include the following. KD-trees work best when there is a relatively high number of primitives to store in them. In the figure 2.3 the wasteful nature of this approach is demonstrated. The scene graph illustrated there could be describing a simple object such as a single segment from the large shell model, figure 4.5. However even in this case 4 KD-trees would be added to the scene graph and each of them would be built over a single primitive. Also this use of multiple KD-trees conforms to the semantic scene graph in the cases where subtrees are created giving up a some of the adaptivity inherent in the KD-tree structure.

2.4 Operation KD-tree

The multiple KD-tree method was not implemented during the course of this thesis. The short theoretical analysis given in the previous section however, presents a compelling reason for the use of one global KD-tree.

A global KD-tree is built the same way as any regular KD-tree over all objects that are present in the scene. The only difference is the possibility to use different intersection cost values for different kinds of primitives. This different cost function can be implemented by reusing the open/close flag in the possible split data used in finding the best split plane with the SAH. This implementation

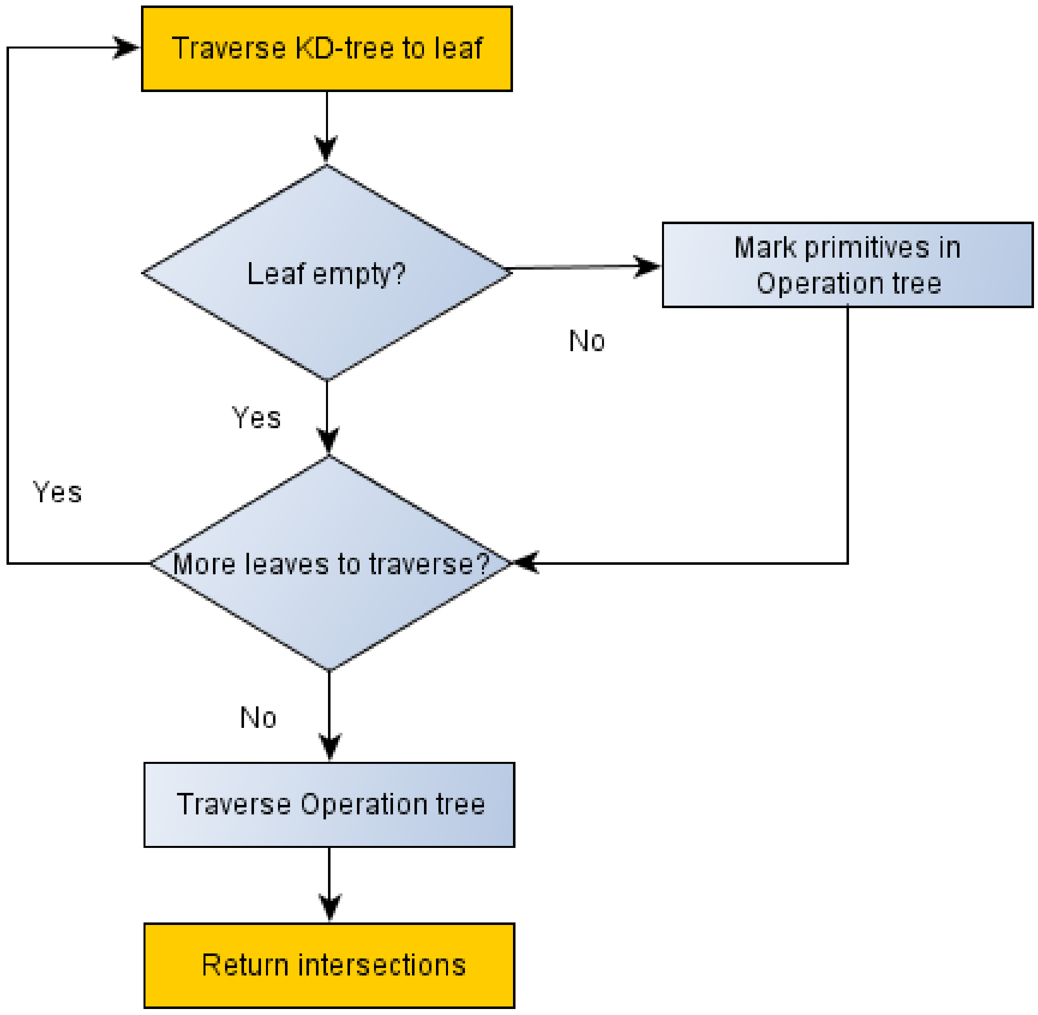

Figure 2.4: Flowchart for the three phases of the OKD ray-scene intersection test

does not increase the computation complexity of building the KD-tree.

If a global KD-tree is used as an acceleration structure the information about the CSG structure has to be maintained in addition to the KD-tree. The semantic scene graph itself could be used for this, however the semantic scene graph has many non-essential nodes in it. These nodes should be avoided in order to get as fast traversal as possible. As solution to this a small compact single purpose data structure with fast traversal is introduced, the operation tree. The operation tree in conjunction with the KD-tree form the Operation KD-tree (OKD-tree).

The ray-scene intersection test as executed by the OKD-tree consists of three phases. The three phases are illustrated on figure 2.4

- In the first phase the KD-tree is traversed using the classic traversal algorithm down to a leaf. The fact that the whole traversal algorithm does not have to be changed allows for an alternate perhaps better optimized algorithm to be adapted as well.

- When the traversal reaches a leaf node in the KD-tree the second phase is executed. In this phase one would normally do the intersection tests for the primitives in the list held by the node and return any intersections that are found. In this case however, the intersection tests are not executed. The execution of intersection tests is pushed back after the whole ray traversal trough the KD-tree is done. Since the nature of CSG scene precludes any early ray termination anyway, there is no harm in doing this. Instead of executing the intersection tests the primitives in the list are just marked for testing and the KD-tree traversal algorithm continues as if no intersections were found in this leaf.

- The third phase of the traversal starts when the KD-tree traversal has finished completely. Now the intersection tests are executed by the operation tree on the primitives marked by the KD-tree traversal algorithm. The operation tree is responsible for combining the found intersection so that they give the expected result.

The operation tree implemented in this thesis is a small tree structure optimized for a single task. The only task it has to fulfill is to get and combine the intersection from the marked primitives. This operation tree is constructed before ray tracing starts and is unchanged during ray tracing. The construction of the operation tree mimics the ray traversal in the semantic scene graph. It essentially creates a smaller version of the semantic scene graph, which contains only the pure boolean operations. As a result the leaves of the operation tree are pointing to the actual primitives in the scene.

For the representation of this operation tree a similar scheme has been chosen as for the cache line optimized representation of the KD-tree nodes. There are however some differences. The nodes in the operation tree have to implement a limited form of polymorphism. The different operations handle the intersections differently. To avoid the overhead of polymorphism a function pointer to a static function that is supposed to handle the intersection combination is stored in the node. This pointer is assigned during the construction of the operation tree. For leaf nodes this function calls the intersection function on the primitive pointed to from the leaf. This eliminates the overhead of figuring out the actual type o the primitive, because this identification is done during the construction of the operation tree. Storing the function pointer is possible because this structure has only this one purpose and no other polymorphic functions are expected.

As a results all the nodes share the same structure. They are organized in one sequential array where the children of a given node follow each other. Each node stores the index of their parent node, the index of the first child or the pointer to the primitive in the leaf and the function pointer for the combination function. If the SCG scene has operation nodes that have more than two children their number has to be saved in the node too. An example of such a node is the UNION node which is a node that does the boolean union operation on variable number of operands. The resulting structure is:

```
STRUCT OperationNode
{
  int IndexOfParentNode
  int NumberOfChildren

  //shared space for two attributes
  int IndexOfFirstChildeNode | PointerToPrimitiveInLeaf

  functionPointer IntersectionFunction.
}
```

As is apparent the nodes have no flag that could be checked to mark nodes for intersection testing. To allow for multiple threads to ray trace the structure the marking has to be done in a structure belonging to only one thread. This structure is in the case of this implementation the traversal structure itself. This structure maintains the state of the traversal. This state incorporates the active material, surface, transformation attributes and with the OKD-tree a structure that can determine for an index in the operation tree if the corresponding node is marked. In this case, it is an array of boolean values. It could be optimized to use single bits as flags.

When a leaf node of the operation tree is marked by the KD-tree traversal this mark is propagated upwards in the operation tree. This mark indicates that a given node of the operation tree might be relevant to the intersection tests. By propagating the mark upwards the tree, one can effectively ensure that whole branches of the tree are ignored when the tree is later traversed. These would be the branches that did not get marked by the KD-tree. Pseudo code for the marking follows.

```
FUNCTION mark(
  OperationNode nodeToMark,
  BoolArray flagArray
  )
{
  Mark the nodeToMark in the flagArray;
  Exit if nodeToMark is root;
  Exit if parent of nodeToMark already marked;
  Call marking on the parent of nodeToMark;
}
```

When the operation tree is traversed in the third phase it does the depth-first search as the semantic scene graph would do it. The one important difference is that the nodes that are not marked by the KD-tree do not get traversed. This eliminates the intersection test of the primitives that would not by tested by the KD-tree. The intersection combination operations for nodes that have no marked children are eliminated this way too. When a node is accessed during the third phase of ray traversal it clears its flag. This ensures that after the traversal is done the governing structure of the traversal is in consistent state with the flags for the operation tree all unmarked. Pseudo code for operation tree ray traversal:

```
FUNCTION traverse(
  OperationNode nodeToTraverse,
  BoolArray flagArray
  )
{
  FOR all children of nodeToTraverse
  {
    IF child is marked in flag array
    THAN traverse child;
  }
  Execute intersection function stored in nodeToTraverse
     to combine intersection returned by children;
  Unmark nodeToTraverse in flagArray;
  Return combined intersections;
}
```

This acceleration method should fully utilize the spatial partitioning done by the KD-tree and the efficient traversal algorithms developed for KD-trees. It does impose a significant overhead required to maintain and traverse the operation tree, but it is a trade-off for making the direct CSG ray tracing possible with global adaptive acceleration structures.

As shown, the method is implemented for the KD-tree but in theory one should be able to modify any acceleration structure to work on this basis.

2.5 Modified SAH and Mail-boxing

With spatial subdivision structures like the KD-tree, primitives can fall into more than one of the subdivided space areas. This fact has been explained in previous sections, however there is an implication that has not yet been explored here. When a ray traverses multiple leaves of the subdivided space in which the same primitive is present, this primitive gets tested for intersection in each of these leaves.

These multiple ray intersection tests are counter-productive. The whole reason to use a spatial subdivision structure is to eliminate intersection test on primitives that are not going to be hit anyway, yet this same structure introduces multiple intersection tests on the same primitive.

To solve this contradiction a method called Mail-boxing [1, 8] is used. This adds a structure —the mail-box — to all primitives to track the last ray that have been intersected with the primitive. If then a ray is about to be intersected with the a given primitive multiple times, the mail-box can identify this ray and no additional intersection tests are executed.

Mail-boxing has however significant hardware cache drawbacks. The mailbox structure for the primitives can take up a large amount of memory. This memory is accessed at completely random which causes cache line misses and is detrimental to performance. In fact it has been determined that in some cases mail-boxing is more harmful than useful for ray tracing performance.

Several improvements have been introduced to this method to make it viable. Some of these aimed to minimize the size of the mail-boxing structure. Also the inverse mail-boxing was introduced where the ray keeps track of a few primitives that have been visited by it. Inverse mail-boxing was proved to be successful and is in common use.

The method of adapting the KD-tree for CSG scenes described here does not use mail-boxing directly. However, by pushing the intersection test of the primitives back to be performed only after the whole KD-tree has been traversed, the result is the same. No primitive gets tested more than once. This is the result of the marking system, because marking an already marked primitive does not introduce a new intersection test. In effect the marking system used to track which primitives need to be tested during CSG traversal works as a mail-boxing system.

Since mail-boxing is a natural byproduct of the operation tree, the construction of KD-tree should be adjusted to take this fact in account. Warren A. Hunt introduced a small modification to the SAH in respect to mail-boxing [5]. The modification introduces the consideration that for rays that traverse both sides of a split the primitives present in both children do not increase the split cost. The resulting metric is:

$$C = C_t + \frac{SA(B_1)}{SA(B)} * N_1 * C_p + \frac{SA(B_2)}{SA(B)} * N_2 * C_p - \frac{SA(Split)}{SA(B)} * N_{1,2} * C_p \quad (2.1)$$

Here $SA(Split)$ denotes the the surface area common to the children of the node. This is effectively the splitting plane. $N_{1,2}$ denotes the number of primitives present in both children.

3. Triangle Meshes in CSG

3.1 Testing with Triangle Meshes

The new concept of using a KD-tree and tracking CSG operations in parallel had to be tested in some working application. For the purposes of testing this concept the Advanced Rendering Toolkit (ART) was used. This is a rendering toolkit in which CSG scenes were already used as the main form of scene definition, however it was not a toolkit built primarily for real-time ray tracing research purposes, rather for photo-realistic and predictive rendering purposes.

Even under the premise of focusing on predictive rendering having an efficient ray tracer is valuable. The rendering performance however can not be expected to hold up to the standards set by modern rendering engines optimized toward rendering speed. Comparing results with these engines would not yield a reasonable comparison between ray tracing triangle meshes and ray tracing CSG scenes. In this case the comparison between the two methods has to be made within the same engine.

Since updating the ray tracing of ART was initially a goal of the work, instead of implementing a full CSG scheme into any of the state-of-the-art ray tracers, the ability to ray trace triangle meshes was implemented into ART itself. As long as one limits the testing to the basic forms of the above mentioned acceleration methods, the comparison between the results holds and has bearing on the relative efficiency of the algorithms.

This introduces the problem of including triangle meshes into a full CSG scheme. The variability of the SCG scheme however deals with this quite well. In essence, one only needs to introduce a new primitive, the triangle mesh, into the scene. When this primitive is implemented it is easy to create a scene that is otherwise empty but contains a single triangle mesh primitive. The result of such a scene is pure triangle mesh ray tracing, because the primitives in a CSG scene effectively ray trace their own geometry once ray intersection tests are invoked on them. Since the scene was empty to begin with except for the triangle mesh, the only operations executed during ray tracing are the operations associated with triangle mesh ray tracing.

On this basis any other operations done before and after the actual ray tracing are executed in both triangle mesh or CSG cases thus bear no effect on relative rendering time. These operations in a predictive rendering environment may be more complex than in a real-time rendering environment which is why comparing the two directly is undesirable.

3.2 Implementing the Triangle Mesh Primitive

As explained triangle meshes are implemented into the standing CSG scheme as new primitives. They are a parametric primitive in contrast to the sphere or the cube which are created without any parameters and manipulated only trough transformations. A vertex array and an index array serve as the parameters of the triangle mesh. These define their actual geometry.

The vertex and index arrays are imported from a commonly used file format, the Stanford PLY format. The reasoning behind this is the fact that most commonly used testing scenes are available from the Stanford 3D scanning repository in this format. Also most of the commonly used modeling applications are able to export into this format. This is a useful property. For testing purposes models had to be created that can be defined using CSG modeling as well as triangle meshes.

The PLY file format describes the scene in a simple manner. In the header of the file the elements used in the file are defined. Different properties can be attached to these elements beyond their position such as color, normals etc, but they will be disregarded by the current implementation of the triangle mesh primitive. For every element the number of its uses is provided in its description. The header is than followed by the a simple listing of these elements.

During construction of the triangle mesh primitive the PLY file is read. To read the PLY file format the simple lightweight RPly library is used. When the data is extracted the actual triangles are created and stored in the data structure of the primitive. Right at this point the data needed for fast triangle ray intersection are calculated and stored.

To actually achieve the pure triangle mesh ray tracing the mesh primitive has to be encapsulated so that it can be considered a scene in itself. This is achieved by separating the internal ray tracing from the rest of the CSG scene. This means that the outside scene does not know about the triangles in the mesh. In fact from the outside the triangle mesh primitive is just another primitive. All the information the outside scene may have is the bounding box of the whole mesh.

This separation of the mesh allows for local ray tracing acceleration to be used within the mesh. To provide usable comparison between triangle mesh ray tracing and CSG ray tracing a KD-tree is used as a local acceleration structure. This KD-tree has the same optimizations implemented as the global KD-tree used for the CSG scene. The local acceleration structure of the mesh is built when the mesh is read from the file.

In the previous chapter an argument against local acceleration structures was made. It was based on the fact that they would introduce unmanageable memory costs, while possibly handling only a small number of primitives. In the case of triangle meshes this argument does not hold. Every triangle mesh introduces only one local acceleration structure, while this structure handles all the triangles in the mesh.

The resulting primitive is in essence quite simple. It stores the set of triangles, accessed trough the local acceleration structure. When ray intersection is invoked, it executes the internal ray tracing. The intersections are however returned as if they were intersections with a hull, not a solid object. This difference means that the boolean operators will not work with this basic implementation.

3.3 Additions to the Triangle Mesh

While the previously described implementation of the triangle mesh primitive is sufficient for testing purposes, the mesh can be adapted to serve as a fully operational CSG primitive. This would mean that boolean operations should be applicable to them.

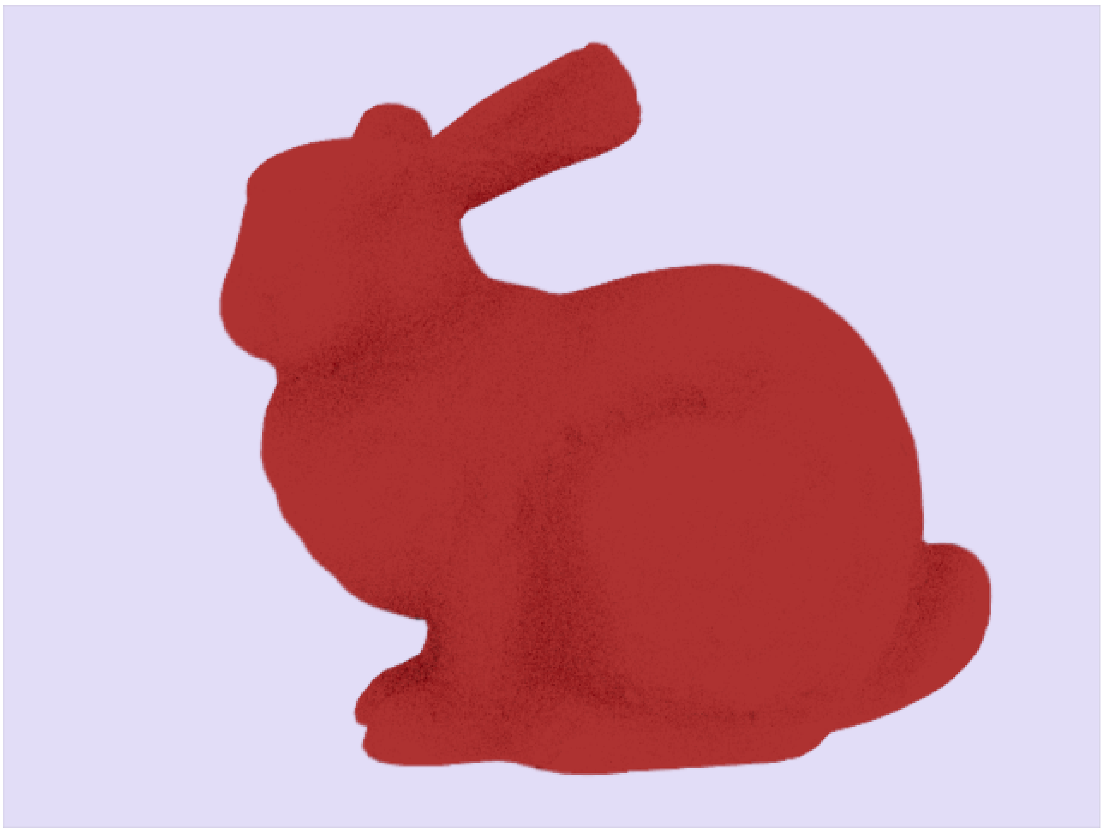

Figure 3.1: The stanford bunny rendered.

The main reason for implementing the triangle mesh in such way is to expand the variability of the SCG scene. One must admit that CSG modeling while powerful, has its own weaknesses. Architectural scenes or recursively defined objects like the shell shown in figure **??** can be defined very well trough CSG modeling. However some objects are too complex or irregular to be constructed efficiently from CSG primitives. An example for this would be a human face, or many of the objects in the Stanford 3D scanning repository. Figure 3.1 illustrates the bunny.

Combining the two modeling schemes gives more freedom in creating scenes. Of course using triangle meshes does increase the number of primitives in the CSG scene even if they are encapsulated. On the other hand using CSG operations allows for reusing one model several times. When a transformation is applied to a primitive, the primitive is not modified. The transformation is applied to the ray which is traversed trough the scene. In the same sense when two transformations of the same primitive are coupled together by a boolean operation the primitive is not duplicated. This means that if the same triangle mesh is transformed several times over the scene, the memory requirements of the scene are not increased significantly. Figure 3.2 shows how a bunny mesh can be reused.

The fact that reusing the triangle mesh does not increase memory requirements to store the scene, does not influence the ray tracing speed directly. However it allows for bigger geometry to be stored in memory.

To add full CSG functionality to the triangle mesh one must ensure that the intersections returned by the triangle mesh are paired correctly to simulate a solid object. Solid primitives of the CSG scene return intersections that are actually

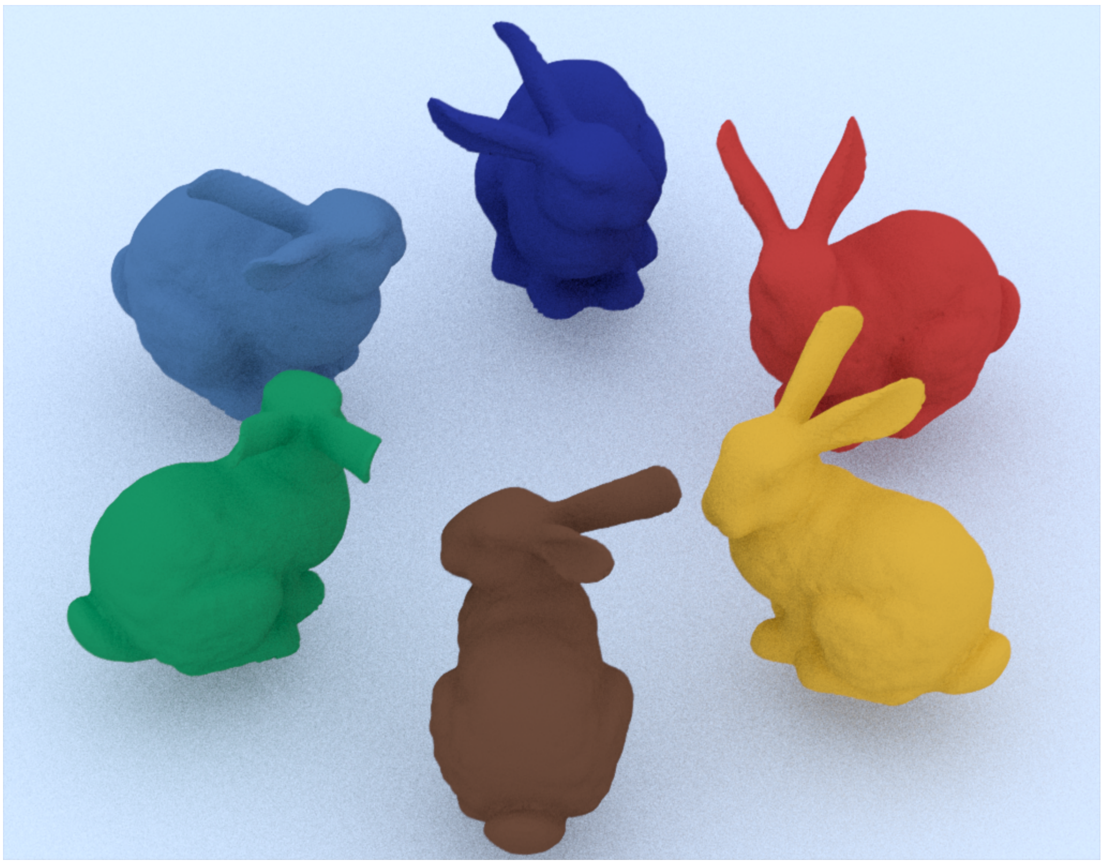

Figure 3.2: Reusing the bunny mesh.

dividers between different materials.

Pairing the intersections presents a problem. For most meshes used for this work no additional information about the faces is available. If the back face or front face information of the triangles were available it would be a simple matter of checking the orientation of the face being intersected to determine whether it's an entering or leaving intersection.

In absence of the face information one is forced to use the heuristic that entering end exiting intersections will alternate. Based on this observation the intersections can be paired up and assigned the appropriate materials.

However the assumption that entering and exiting intersections will alternate does not hold generally. When a ray grazes the mesh it creates only one intersection yet it still remains outside of the mesh. Multiple grazing hits of a ray on the mesh might therefore misqualify inside and outside space.

The assumption of alternating intersections also fails if the triangle mesh is not closed. In this case if the hole is hit either the entering or exiting intersection will be missing. The cases of intersection pairing are shown in figure 3.3.

The problem of not-closed meshes is left to the user. The user is responsible for supplying meshes that are closed if the full functionality of the CSG tree is to be used. In other cases the implementation allows for the user to specify primitive as a hull. This property can be specified for the meshes too, in which case the basic version of the meshes is used, doing no intersection pairing and thus rendering only the hull defined by the mesh. This hull based triangle mesh intersection is used for testing purposes to avoid weighting down the triangle

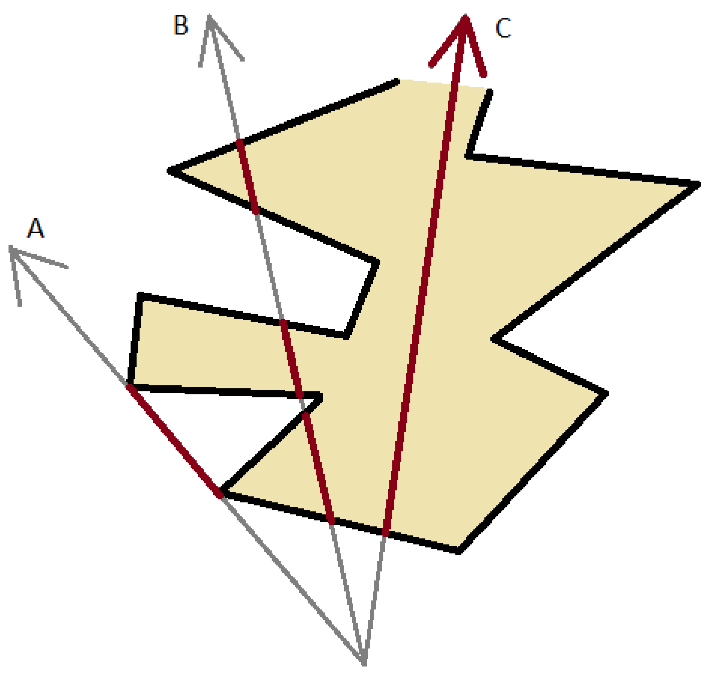

Figure 3.3: Pairing intersections on a triangle mesh. Gray are the rays outside the objects. Red are the rays inside the object. A) shows the grazing ray problem, B) shows a good pairing, C) shows problem with holes in the mesh.

meshes by the overhead of the pairing operations.

The problem of the grazing hits remains. These cases are however extremely rare. Considering that in the current application of this method several rays are sent for every pixel, if one of the rays is compromised its influence gets diluted. Testing shows that this problem does not present itself in practice.

4. Results

4.1 Testing

In the previous chapter it was established that the testing of the OKD-tree driven acceleration structure will be done within ART with emphasis on relative rendering performance to triangle meshes. The rendering performance of the triangle meshes will be used as the baseline.

The actual acceleration structures tested will be:

- Triangle meshes. These form the baseline for the testing. The implemented acceleration structure for the triangle meshes is the basic KD-tree using the cache line optimization, built using the modified SAH and traversed using Vlastimil Havran's traversal algorithm [3].
- CSG with NBVH. The NBVH is used as a control to the OKD-tree acceleration structure.
- CSG with OKD-tree, built with modifiead SAH (MSAH). This is the novel acceleration structure to be tested. In this case the metric used to build the KD-tree is the MSAH.
- CSG with OKD-tree, built with SAH. This test uses the original SAH as a metric for building the KD-tree.

The testing is done for the acceleration structure built using SAH an MSAH separately to investigate the assumption that improvements to the KD-tree can be carried over into the CSG context. If there are cases where the tree built using the MSAH outperforms the tree built using the SAH, it will suggest that one can reap the benefits of improving the KD-tree algorithm in the traditional triangle mesh context. This would mean that research into acceleration structures does not need to be done separately for CSG and triangle mesh context, which would be an interesting realization.

During testing the following data are recorded:

- Build time for the acceleration structure. This time is usually orders of magnitude smaller than the rendering time itself. However it is still an interesting data point. In certain applications for example ray tracing dynamic scenes it would play a greater role.
- The absolute render time. As explained earlier this time should not be compared to the state-of-the-art ray tracer. ART is not designed to be a real-time ray tracing platform, also the test scenes are rendered using full path tracing not just first order visibility as is done by most RTRT tests.
- The relative improvement in render time compared to the triangle mesh ray tracing.

All data shown are an average of rendering a given scene 6 times. This is done to mitigate factors beyond the control of the ray tracing software, mostly the state of the system and to provide more accurate results.

All images are rendered in 640x480 resolution using 8 samples per pixel and a maximal recursion of depth of 20. The testing is done on a MacBook Pro with 2,66GHz Intel Core 2 Duo processor, 3MB L2 cache, 4GB RAM.

4.2 Test Scenes

Finding or creating suitable scenes or even just models for testing purposes in this thesis was a relatively complicated task. With triangle meshes being used so dominantly in computer graphics, CSG scene definitions are hard to come by. Finding scenes that are described in CSG and triangle mesh formats at the same time is even harder.

As a result models had to be created from scratch. This means that the tests are run only on a limited number of simple models. However these models were chosen each to test a specific aspect of the algorithm, thus they should serve as sufficient examples and testbeds. Each model will be discussed in depth in its own section together with the rendering results that have been achieved using them.

The models used for testing are:

- Grooved sphere. This is one sphere with 6 grooves. Shown in figure 4.2.
- Checkered cube. This is a cube that has a lot of small cubes subtracted from its sides. Shown in figure 4.4.
- Large shell. A shell like model, with several turns. Shown in figure 4.5.
- Small shell. A shell like model, with only one turn. Shown in figure 4.6.

The reason for using two shell models stems from the way they are constructed. A segment of the large shell is constructed by subtracting two slightly translated spheres from a central sphere. A segment of the small shell is constructed by subtracting a smaller sphere from a concentric larger sphere and the result is cut by planes or in this case cubes to form a segment. The construction of these segments is shown in figure 4.1. In both cases the shell itself is then constructed by rotating and scaling the segments in a FOR loop. Note that this FOR loop is a very intuitive way to describe the scene, but it creates an unbalanced semantic scene graph.

The two different methods of constructing the shells were not chosen at random. The method used for the large shell makes more sense in CSG context, because less primitives are used in a less contrived way. However, the need for the construction method used for the small shell arose from the need to create a corresponding triangle mesh.

Triangle meshes for testing were created using the Netgen Mesh Generator. Effort was made to use Blender or 3D Studio Max applications. The functionality of boolean operator needed to create a shell model is however suspect in these applications. Even the Netgen Mesh Generator created artifacts when the construction method for the large shell was used. The problems there are the thin

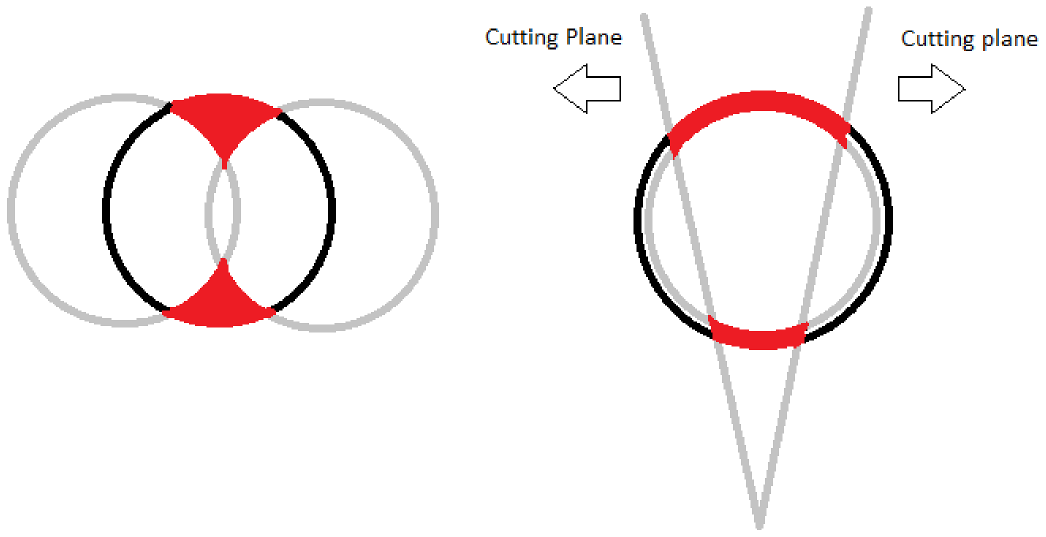

Figure 4.1: The two different modes of constructing a single segment of the shell. Left is the large shell segment, right is the small shell segment. The segments are displayed in cross section. Black outline is the original sphere, gray are the subtracted parts, while red is the resulting segment.

edges created by the sphere subtraction. These edges came out jugged and uneven. Using the small shell construction method results in an empty sphere with constant wall width. Subtraction from this empty sphere is handled correctly by the meshing software.

An observation here is in order. While an experienced graphic artist might have been able to create even the shell object with far less trouble, the fact remains that its description from the CSG scene is more intuitive and easier to realize.

4.3 Grooved Sphere

The grooved sphere model is a sphere from which 6 toruses were subtracted to form the grooves. The toruses are concentric with the sphere and in pair lay in axis aligned planes. The model is shown in figure 4.2. The semantic scene graph describing the scene is a completely balanced tree. The code describing this model can be found in the attachments.

This model was chosen because it presents several worst case scenarios.

For the triangle meshes this model is a worst case scenario based on the fact that spheres and toruses require a high number of triangle subdivisions to create a relatively smooth surface. Combining these high polygon count primitives using subtraction makes the resulting object even more expensive. For testing purposes where a fully diffused surface was used a 15K triangle count mesh was used. For surfaces with specular reflections an even higher polygon count would be needed.

This model also presents a worst case scenario for the OKD-tree acceleration structure when rendering in CSG context. The bounding boxes of the toruses and the sphere completely overlap in this model. The faces of these bounding boxes are used as possible split planes by the SAH. As a result there is no way to

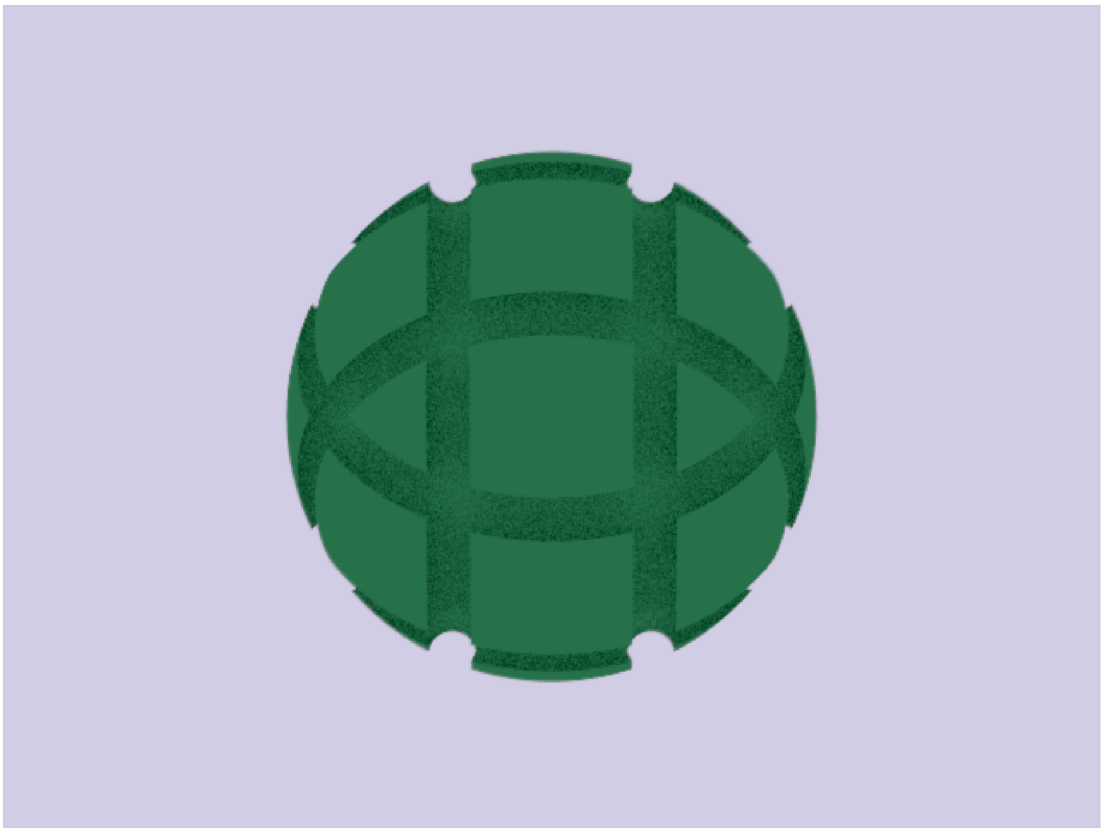
Figure 4.2: Grooved sphere test image.

do space partitioning to reduce the number of primitives in any given partition of the space. This problem is illustrated in figure 4.3. In this figure it is easy to see that whichever split plane is chosen the number of primitives is reduced at most by 2 for one side of the split but stays the same for the other side. During this the surface area of the children does not decrease sufficiently to warrant a split.

In the figure 4.3 a split is shown which partitions of 2 primitives. This would be the most likely split to keep, because there is no empty space to cut off. Even in this case the cost of splitting is:

$$C_{split} = C_t + \frac{5}{8} * 3 * C_p + \frac{7}{8} * 5 * C_p = C_t + 9\frac{4}{8} * C_p \tag{4.1}$$

The cost of creating a leaf is:

$$C_{leaf} = 7 * C_p \tag{4.2}$$

It is clear that splitting the node which in this case is the root node is just not cost-effective.

In this case modified SAH does calculate the cost of split lower than the original SAH, but it still concludes that splitting would be too costly. As a result for both SAH and MSAH heuristics the created KD-tree is a degenerated tree with only one leaf that includes all the primitives.

The test result shown in table 4.1 indicate that the CSG scene outperforms the triangle mesh quite clearly. It is also worthwhile to note that the acceleration structure construction time is negligible for the CSG scene.

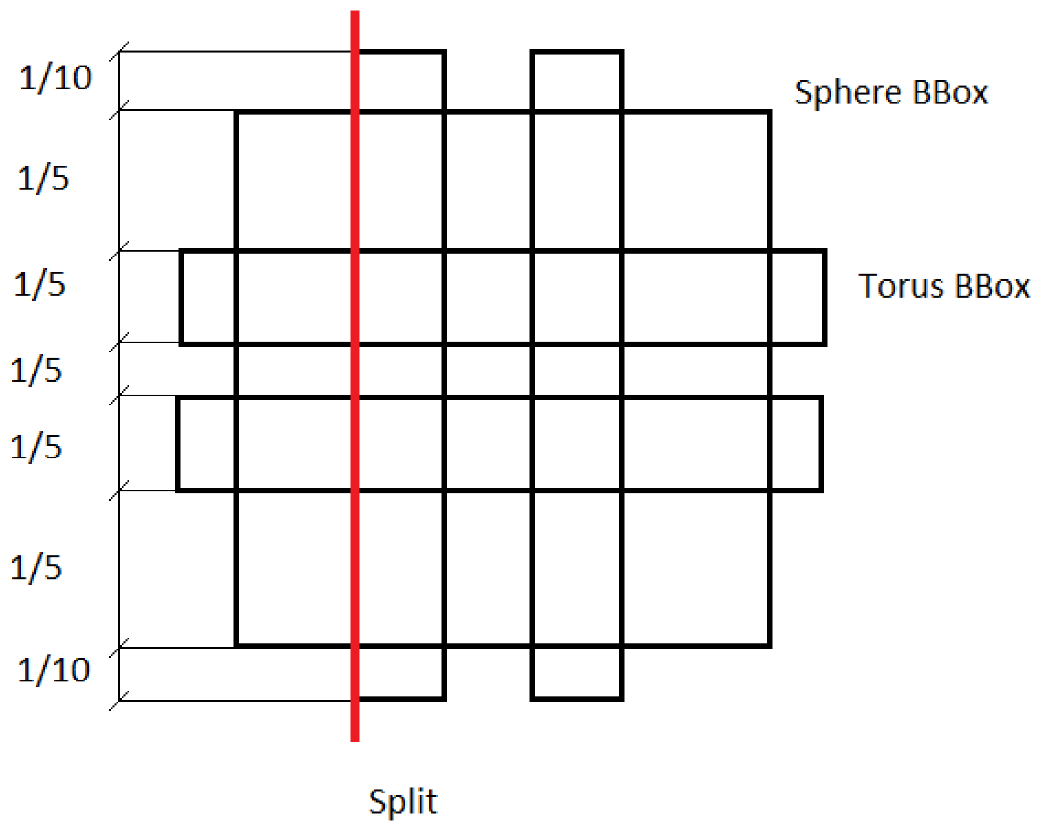

Figure 4.3: Illustration of the most sensible split in grooved sphere.

The test results also show that in this worst case scenario the OKD-tree preforms slightly worse than the NBVH. This has to do with the fact that there is no way to partition this model, in which case the overhead introduced by tracking the operations in addition to the KD-tree is just too high. The KD-tree traversal is generally faster than the BVH traversal, however in this case it does not yield any reduction in intersection test numbers thus it can not offset the operation tree. The performance drop is however below 1% which is not so significant.

Using the MSAH heuristic did not yield better partitioning of the model, therefore the result are virtually the same as when the SAH was used.

4.4 Checkered Cube

The Checkered cube is a cube with a surface indentations. Form each of its faces a 5 x 5 grid of small cubes is subtracted. The rendering of the model is shown in figure 4.4. The semantic scene graph describing the scene is a completely balanced tree, as implemented. The code describing this model can be found in the attachments.

This model was chosen to present a best case scenario for the triangle mesh approach, however this is also a good scene for the OKD-tree method.

This model is a best case scenario for the triangle mesh method because it is full of flat surfaces which can be very well triangulated. For this model the absolute minimum of triangles that can describe it is just as good as any finer

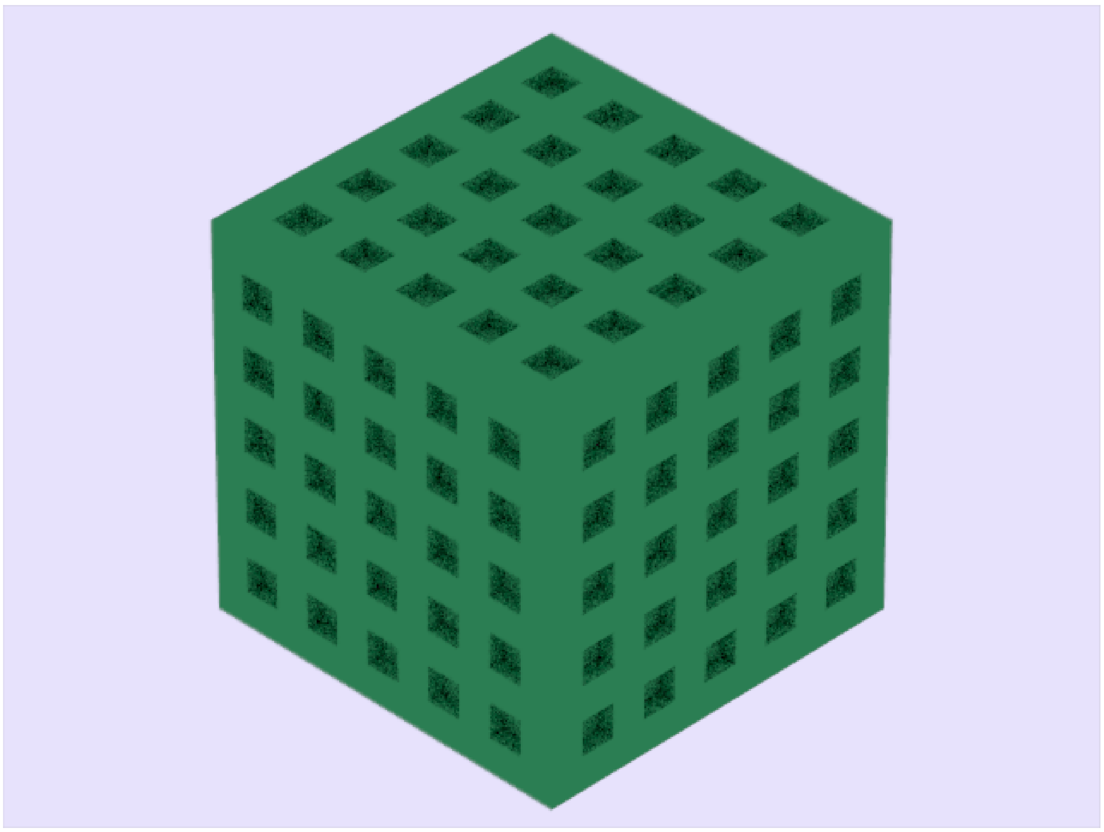

Figure 4.4: Checkered cube test image.

subdivision. The mesh of this model contains only 2538 triangles. This might still look like a lot of triangles, but one should considers the fact that just the faces of the small cubes require 1500 triangles.

In contrast one needs only 151 primitives for the whole model (the number of small cubes plus the big cube) when it is created using CSG modeling. This simple calculation shows the real difference between the number of primitives used in describing the scene by triangles or CSG, even if it is a best case scenario for the triangle meshes.

The reason why this is a good scene for the OKD-tree is because even though it uses quite a high number of CSG primitives, these primitives do not overlap to a high degree. This should imply that a reasonably good KD-tree can be built above it. The primitives in this scene are axis aligned cubes, which means no empty space is wasted during space partitioning (cubes fit perfectly into their bounding boxes).

The OKD-tree for this scene is built using the SAH as well as using the MSAH. Building the scene using the MSAH creates an almost twice as deep tree as with the original SAH. This implies a possible difference in rendering performance.

The test results for the checkered cube scene are shown in table 4.2. Even thou this is a very good case for the triangle mesh all the CSG approaches do outperform it. The improvement is not as big as with the grooved sphere scene, but it is still there.

The more interesting result in this test is the comparison between OKD-tree and the NBVH. The NBVH still outperforms the OKD-tree built using the SAH. This is stemming from the fact that the KD-tree built is still not efficient enough

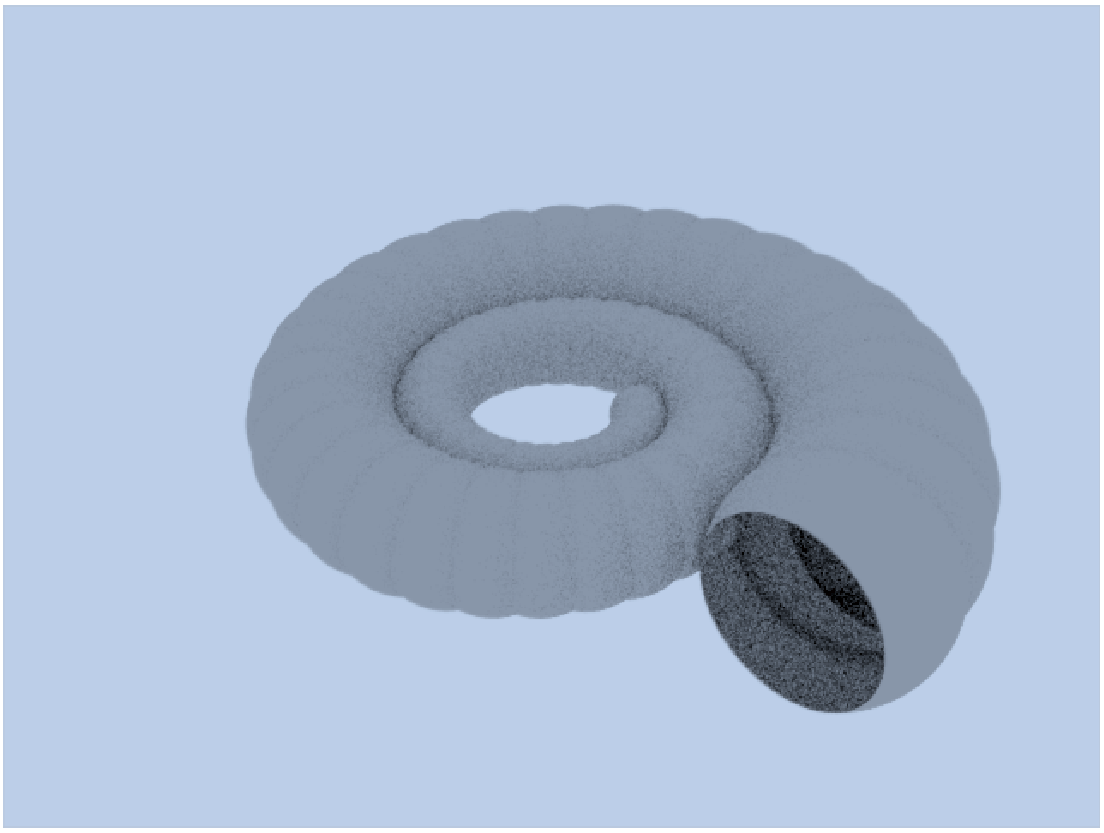

Figure 4.5: Large shell test image.

to offset the overhead of tracing the operations. On the other hand the OKD-tree built using the MSAH does outperform the NBVH. The improvement compared to the NBHV is still only about 1.5%, but it proves that the OKD-tree is at least a zero sum method. There are cases as this, where it is more efficient than the NBVH. In other cases it might be less efficient then the NBVH, but the variance is within tolerable limits.

Another interesting result is the fact that with this scene implementing a simple optimization initially developed for triangle meshes ray tracing also enhances the performance of the OKD-tree. As mentioned at the beginning of this chapter this gives credit to the assumption that transplanting optimizations developed for KD-trees in triangle mesh ray tracing does retain the benefits of these optimizations.

In this scene the build time for the acceleration structure of the triangle mesh is still one order of magnitude larger than the build times of any of the CSG methods.

4.5 Shell

The shell is an example of a procedurally created model. It is created by defining one segment of the shell in some way. Two of those ways were discussed earlier. Once the segment is defined it is transformed and added to the already finished shell in a loop. In this case the transformations required to make the shell are rotation and scaling. The code describing both shells used for testing can be

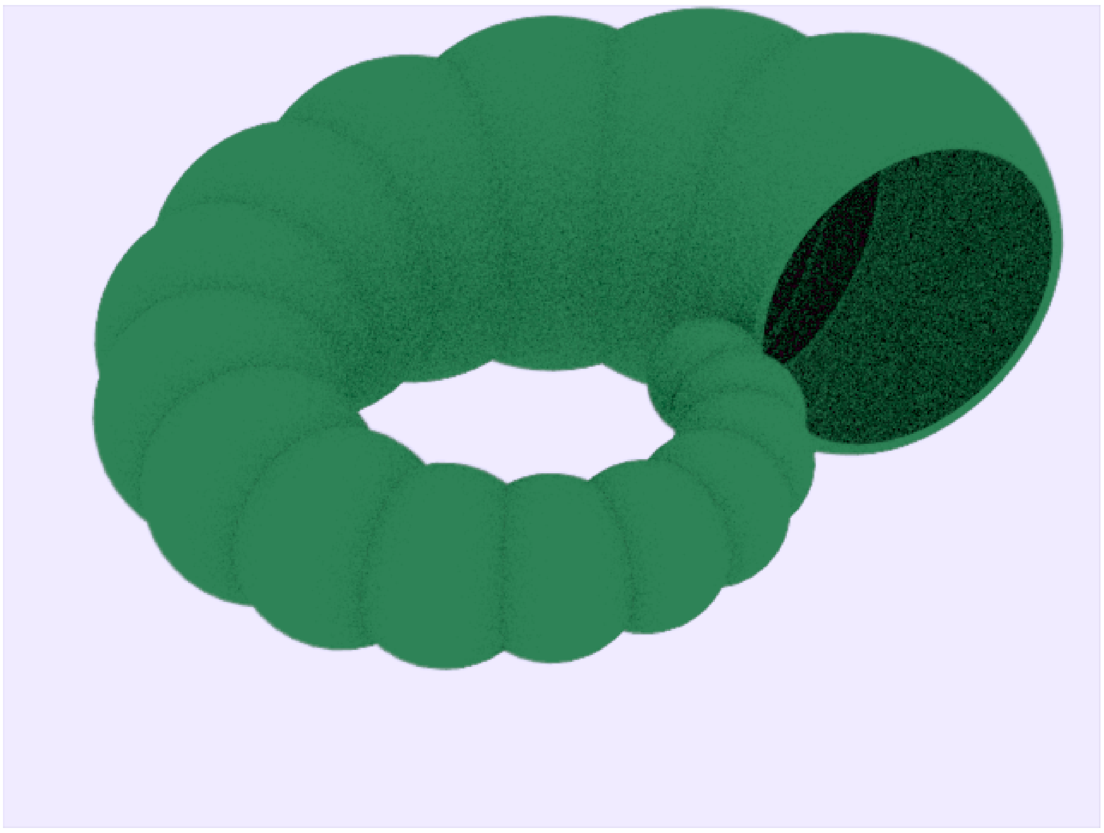

Figure 4.6: Small shell test image.

found the attachments.

Two different ways of constructing the shell are used. The small shell is shown in figure 4.6, while the large shell is shown in figure 4.5. The reasons for using two shell models stemming from the necessity of creating a triangle mesh approximation of the shell have already been discussed. However, the shell model has some properties that make it interesting just from the perspective of comparing the NBVH and the OKD-tree.

Two attributes of the shell play a mayor role in the efficiency of ray tracing it.

- A single segment of the shell is created using primitives that have bounding boxes that overlap to a significant degree. As with the grooved sphere model this will hinder the efficiency of the OKD-tree acceleration structure.

  From this perspective the construction mode of the small shell is more problematic. It uses 4 primitives and all of these primitives have almost fully overlapping bounding boxes. In fact the planes are simulated by enlarged and rotated cubes thus they do overlap fully with the main sphere. The smaller sphere used in creating the hole in the shell does not overlap fully but just by a little.

  In contrast the segment created by the construction method of the large shell uses 3 primitives that do not overlap to such a high degree.

- The FOR loop used to string the segments together creates a very unbalanced CSG-tree. Every OR node created by this loop has at least one leaf operand, the other is either another OR node or a leaf if it is the deepest OR node. Traversing this semantic scene graph while using the NBVH acceleration structure is inefficient. At every level a bounding volume intersection test is done which can usually eliminate only a small number of primitives.

This property of the shell should favor the OKD-tree structure. The operation tree used in the OKD-tree will be unbalanced the same way as the semantic scene graph itself. However, while traversing the operation tree to do the intersection tests, no bounding volume intersection test are done. The lookup in the marking array is cheaper than doing the full ray-AABB intersection test. This however does not guarantee a more efficient ray tracing, since if the KD-tree cannot separate space properly the operation tree traversal will have to visit more branches.

The more dominant of the previous two properties will tip the rendering performance in favor of either the NBHV or the OKD-tree.

Since there is no corresponding triangle mesh for the large shell, no tests could be done there. However there is no reason these tests would give much different results than the mesh for the small shell. The shell in general is a relatively high polygon count model. If a suitable mesh could have been created for the large shell the polygon count would be even higher that the polygon count of the small shell mesh. The small shell mesh contains 28K triangles.

The test results for the shell scenes are shown in tables 4.3 and 4.4.

The triangle mesh is outperformed in these scenes by all CSG acceleration methods.

The OKD-tree build using MSAH is outperforming the OKD-tree built using original SAH. This further proves the validity of transplanting optimizations into KD-trees.

As for the comparison between the NBVH and the OKD-tree, the tests show that a deeper unbalanced semantic scene graph can be rendered more efficiently by the OKD-tree. A deeper semantic scene graph in this case would be created by a shell that has more turns. For a shell with only one turn the the overlapping nature of the primitives favors the NBVH acceleration, but for a shell with two or more turns the OKD-tree wins.

Subsequent testing showed that even if the more complex segment construction method of the small shell is used for a shell with at least two turns the OKD-tree will render more efficiently. This is in spite of the fact that in this case the overlapping of primitives creates a degenerated KD-tree which has leaves that have over 90 primitives in them. The nature of the operation tree is such that when it has to be traversed deeply then the single task orientation compensates for all the faults of the other structures used.

Table 4.1: Rendering results of the grooved sphere model

Acceleration structure	Structure build time	Absolute render time	Improvement
Triangle Mesh	2.09 sec	44.85 sec	0%
CSG with NBVH	0.0 sec	34.155 sec	23.84%
CSG with OKD-tree (SAH)	0.0 sec	34.53 sec	23.01%
CSG with OKD-tree (MSAH)	0.0 sec	34.52 sec	23.03%

Table 4.2: Rendering results of the checkered cube model

Acceleration structure	Structure build time	Absolute render time	Improvement
Triangle Mesh	0.10 sec	51.26 sec	0%
CSG with NBVH	0.03 sec	48.57 sec	5.27%
CSG with OKD-tree (SAH)	0.04 sec	48.90 sec	4.62%
CSG with OKD-tree (MSAH)	0.04 sec	47.72 sec	6.94%

Table 4.3: Rendering results of the large shell model

Acceleration structure	Structure build time	Absolute render time	Improvement
CSG with NBVH	0.17 sec	91.045 sec	0%
CSG with OKD-tree (SAH)	0.18 sec	86.34 sec	5.1%
CSG with OKD-tree (MSAH)	0.18 sec	85 sec	6.63%

Table 4.4: Rendering results of the small shell model

Acceleration structure	Structure build time	Absolute render time	Improvement
Triangle Mesh	3.1 sec	134.21 sec	0%
CSG with NBVH	0.02 sec	101.21 sec	24.62%
CSG with OKD-tree (SAH)	0.02 sec	131.43 sec	2.071%
CSG with OKD-tree (MSAH)	0.02 sec	121.32 sec	9.604%

Conclusion

In this thesis we investigated a method that combines the leading acceleration structure developed for triangle mesh ray tracing with CSG scene description. This was done on the basis that a lower number of primitives required for defining a CSG scene coupled with the efficiency of this acceleration structure will achieve better performance during ray tracing. The results shown in this thesis do suggest that this proposition has its merits.

In all the test scenes the geometry defined by CSG outperformed the geometry defined by triangle meshes. However, one has to note that the rendering environment used to test the proposition while reasonably well optimized is not a state-of-the-art real-time ray tracer. Real-time ray tracers are hardwired to perform operations on triangles as fast as possible. As a result their performance can not be realistically compared to the performance of the renderer presented here, as long as we are concerned with raw rendering speed. Still, the result can not be dismissed because they show that on a level playing field the idea of minimizing the number of primitives to ray trace is solid.

The results also show that the build times for the acceleration structures are greatly reduced when they are used in CSG context. This reduction of build times is at least by one order of magnitude. It is caused simply by the reduction of the number of primitives. This result might seem self evident, nevertheless the build times have to be factored in when considering time-to-image. In fact, even though KD-trees are considered among the most efficient acceleration structures, they come up short for some applications because of the time needed to build them. These are usually dynamic scenes where the motion of the objects in the scene forces a rebuild of the KD-tree after each frame. In this context the build time reduction gained by using a CSG scene might be valuable.

We have also compared the performance of the OKD-tree introduced in this thesis and the NBVH usually used as a go-to acceleration structure for CSG ray tracing. In this comparison the OKD-tree appears to be on par with the NBVH.

We have identified the scenes that favor the NBVH. These are the scenes where the primitives overlap to a significant degree. In these cases the KD-tree, used as the main part of the OKD-tree, can not partition space efficiently enough. As a result the overhead imposed by the Operation-tree, used to carry out the boolean operations inherent in CSG ray tracing, hinders the efficiency of the OKD-tree.

We have also identified the scenes that favor the OKD-tree. These are the scenes where the overlap is minimal or where the semantic scene graph is highly unbalanced.

The scenes with minimally overlapping primitives can be efficiently partitioned by the OKD-tree. This lets the fast traversal algorithms developed for KD-trees to realize their full potential. As a result the overhead of the Operation-tree is mitigated and the ray tracing performance is increased. Scenes that belong to this category are not a rarity. Any time the surface of a large primitive is modified by a number of relatively small primitives a scene like this is created. The checkered cube shown in this thesis is but one of the possibilities.

In case of the scenes where the created semantic scene graph is unbalanced the OKD-tree benefits from the one task orientation of the Operation tree. Some of these scenes may be partitioned well, other not so much. In any case the NBVH has to traverse a large portion of the semantic scene graph, which is inefficient. In contrast the OKD-tree once finished with the KD-tree traversal phase, traverses only the Operation-tree. The traversal algorithm of the Operation tree is far more efficient that the traversal algorithm of the NBVH. This efficiency is due to the following reasons. Polymorphic behavior is excluded from the Operation tree. Also the nodes of this tree are as small as possible, which serves as a cache-line optimization.

The fact that unbalanced semantic scene graphs are no longer a hindrance to the ray tracing performance relaxes the need to control their creation. This makes automatic creation and manipulation with the scene that much easier.

In the thesis a basic optimization of the KD-tree has been transplanted to the OKD-tree, namely the MSAH. The result show that this optimization has maintained its benefits even after transplanting. The performance increase after transplanting the MSAH has averaged on about 3%. In the paper which introduces the MSAH the performance increase is comparable to the performance increase experienced in this implementation. The implication of this is that this combination of acceleration methods can benefit from the same optimizations as the KD-tree.

In conclusion, the use of the KD-tree in a CSG environment is valid idea on par with the existing acceleration structures for CSG. The possibility of transplanting other more sophisticated optimizations is implied by the success of transplanting the MSAH and could be subject to further research. The inability of partitioning significantly overlapping primitives is another subject for further research.

Bibliography

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[2] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, May 1987.

[3] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* PhD thesis, Czech Technical University, Praha, Czech Republic, April 2001. Available from http://www.cgg.cvut.cz/ havran/phdthesis.html.

[4] Vlastimil Havran and Jiri Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, February 2002.

[5] Warren Hunt. Corrections to the surface area metric with respect to mailboxing. In *IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 77–80. IEEE/EG, Aug 2008.

[6] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In David C. Evans and Russell J. Athay, editors, *SIGGRAPH*, pages 133–142. ACM, 1986.

[7] Jim Kajiya. The Rendering Equation. *SIGGRAPH '86 (Proceedings of the 13th annual conference on Computer graphics and interactive techniques)*, 1986.

[8] David Kirk and James Arvo. Improved ray tagging for voxel-based ray tracing. In James Arvo, editor, *Graphics Gems II*, pages 264–266. Academic Press, 1991.

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

[10] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[11] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). *Symposium on Interactive Ray Tracing*, 0:61–69, 2006.

[12] Wikipedia. Boolean difference. `http://en.wikipedia.org/wiki/File:Boolean_difference.PNG`, 2006.

[13] Wikipedia. Boolean intersect. `http://en.wikipedia.org/wiki/File:Boolean_intersect.PNG`, 2006.

[14] Wikipedia. Boolean union. `http://en.wikipedia.org/wiki/File:Boolean_union.PNG`, 2006.

[15] Wikipedia. Ray trace diagram. `http://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg#metadata`, 2008.

[16] Wikipedia. Example of bounding volume hierarchy. http://en.wikipedia.org/wiki/File:Example_of_bounding_volume_hierarchy.svg, 2011.

Nomenclature

AABB	Axis aligned bounding box
BHV	Bounding volume hierarchy
BV	Bounding volume
CSG	Constructive solid geometry
MSAH	Modifiead SAH
NBVH	Native bounding volume hierarchy
OKD-tree	Operation KD-tree
SAH	Surface area heuristic

Attachments

1 Semantic Scene Graph Codes

The code to create the grooved sphere model:

```
ArNode * createGroovedSphere(
        ART_GV  * art_gv
        )
{
  id sphere = SPHERE;

   id torus = [ TORUS( 0.1 ) apply
    : USCALE(0.9)
    : TRANSLATION( 0, 0.0, 0.374 )
    ];

  id groovePair = [ torus or : [ torus apply : ROT_Y( 180 DEGREES )
      ]];

  id allGrooves = UNION(
    groovePair,
    [ groovePair apply: ROT_Y( 90 DEGREES ) ],
    [ groovePair apply: ROT_X( 90 DEGREES ) ],
    UNION_END
    );

  id groovedSphere = [ sphere sub: allGrooves ];

    return groovedSphere;
}
```

Code to create the checkered cube model:

```
ArNode * createCheckerCube(
        ART_GV  * art_gv
        )
{
  // This is the big cube;
  id cube = [ CUBE apply :TRANSLATION(-0.5, -0.5, -0.5) :USCALE(2.2)
      ];

  // This is the basic small cube;
  id smallcube = [ CUBE apply
    :TRANSLATION(-0.5, -0.5, -0.5)
    :USCALE(0.2)
    :TRANSLATION(0.8,0.8,1.1)
    ];

  //Create a row of small cubes by translating the one small cube;
  id smallcubeRow = UNION(
    smallcube,
    [ smallcube apply :TRANSLATION(-0.4,0,0) ],
    [ smallcube apply :TRANSLATION(-0.8,0,0) ],
    [ smallcube apply :TRANSLATION(-1.2,0,0) ],
    [ smallcube apply :TRANSLATION(-1.6,0,0) ],
    UNION_END
  );

  //Create a grid of small cubes by translating the row of small
      cubes;
  id smallcubeGrid = UNION(
    smallcubeRow,
    [ smallcubeRow apply :TRANSLATION(0,-0.4,0) ],
    [ smallcubeRow apply :TRANSLATION(0,-0.8,0) ],
    [ smallcubeRow apply :TRANSLATION(0,-1.2,0) ],
    [ smallcubeRow apply :TRANSLATION(0,-1.6,0) ],
    UNION_END
  );

  //Create a lattice of small cubes by rotating the grid of cubes;
  //This lattice is not created by translation, because there is no
      need
  //for small cubes inside the big cube.
  id smallcubeLattice = UNION(
    smallcubeGrid,
    [ smallcubeGrid apply :ROT_X(90 DEGREES) ],
    [ smallcubeGrid apply :ROT_X(180 DEGREES) ],
    [ smallcubeGrid apply :ROT_X(270 DEGREES) ],
    [ smallcubeGrid apply :ROT_Y(90 DEGREES) ],
    [ smallcubeGrid apply :ROT_Y(270 DEGREES) ],
    UNION_END
  );

  //Subtract the small cubes from the big cube to get the checker
      cube;
  id checkerCube = [ cube sub: smallcubeLattice ];

    return checkerCube;
}
```

Code to create the small shell model:

```
ArNode * createSmallShell(
        ART_GV * art_gv
        )
{
  id mainSphere = [ SPHERE apply
    :TRANSLATION(0,-2,0)
    ];

  id smallSphere = [ SPHERE apply
    :USCALE(0.95)
    :TRANSLATION(0,-2,0)
    ];

  id plane1 = [ CUBE apply
    :TRANSLATION(0.0, -0.5, -0.5)
    :SCALE(10,10,10)
    :ROT_Z(18 DEGREES)
    ];

  id plane2 = [ CUBE apply
    :TRANSLATION(0.0, -0.5, -0.5)
    :SCALE(10,10,10)
    :ROT_Z(162 DEGREES)
    ];

  //To create a more balanced graph first the union of things
  //that are going to be subtracted is created.
  id thingsToSubtract = UNION(
    smallSphere,
    plane1,
    plane2,
    UNION_END
    );

  //The important subtraction to form the segment.
  id segment = [ mainSphere sub: thingsToSubtract ];

  //The shell starts with one segment.
  id shell = segment;

  //Now the loop creates the segments of the shell
  //by transforming the segment several times
  for( int i = 1; i< 16; ++i )
  {
    segment = [ segment apply
           :ROT_Z( 22.5 DEGREES)
           :USCALE(0.93)
           ];
    //Each new segment is added to the shell.
    //This is what creates the lopsided tree.
    shell = [ shell or: segment ];
  }

    return shell;
}
```

Code to create the large shell model:

```
id createLargeShell(
  ART_GV  * art_gv
  )
{
  id mainSphere = SPHERE;

  id leftSphere = [ SPHERE apply
    :TRANSLATION(0,-0.02,0)
    :USCALE(0.989227963463)
    ];

  id rightSphere = [ SPHERE apply
    :TRANSLATION(0,0.02,0)
    :USCALE(0.989227963463)
    ];

  id segment = [ mainSphere sub:
    [ leftSphere or : rightSphere ]
    ];

  segment = [ segment apply : TRANSLATION (3,0,0)];

  //The shell starts with one segment.
  id shell = segment;

  //Now the loop creates the segments of the shell
  //by transforming the segment several times
  for( int i = 0; i < 64; ++i )
  {
    segment = [ segment apply
             : ROT_Z( 12 DEGREES)
             : USCALE(0.979)
             ];
    //Each new segment is added to the shell.
    //This is what creates the lopsided tree.
    shell = [ shell or: segment ];
    }

    return shell;
}
```

2 Contents of the CD

The CD contains the source code for the ART rendering toolkit. As well as the User manual for ART and some scene description files. Since this thesis was implemented into a large toolkit there is no way of separating the code pertaining to the thesis alone. There is a readme.txt file in the source directory. It lists the most important files associated with this thesis.