TÉCNICO LISBOA

# Coherent Ray-Space Hierarchy via Ray Hashing and Sorting

**Nuno Tiago Lopes dos Reis**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisor(s): Prof. João Madeiras Pereira
Dr. Vasco Costa

### Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Full Name 1 (or 2)
Member of the Committee: Prof. Full Name 3

**October 2015**

# Acknowledgments

I would like to thank my two supervisors, Professor João Madeiras Pereira and Dr. Vasco Costa first for giving me the opportunity to work on a subject that I enjoy. Secondly for all the support over the course of these past few months, without their help I would not have gotten as far I as did.

Finally I would also like to thank you my family and my closest friends, for their support and encouragement during these hard months of work.

## Resumo

Nós apresentamos um algoritmo para a criação de uma hierarquia multinível no espaço dos raios de raios coerentes que corre no GPU. O nosso algoritmo usa rasterização para processar os raios primários e posteriormente usa esses resultados como entrada para a criação da hierarquia de raios secundários. O algoritmo gera um conjunto de raios; cria índices para esses raios, de acordo com os seus atributos; e ordena-os. Deste modo geramos uma lista de raios que são coerentes com os seus adjacentes. Para melhorar o desempenho ainda mais, subdividimos a geometria da cena num conjunto de esferas envolventes que são posteriormente intersectadas com a hierarquia de raios para diminuir o número de falsos positivos nos testes de intersecção com primitivas. Demonstramos que a nossa técnica diminui de forma notável o número de intersecções entre primitivas necessárias para renderizar uma imagem, em particular até cerca de 50% menos do que outros algoritmos nesta classe.



**Palavras-chave:** Rasterização, Ray-Tracing, Indexaçao de Raios, Ordenamento de Raios, Cone Envolvente, Esfera Envolvente, Hierarquias, GPU, GPGPU

## Abstract

We present an algorithm for creating an n-level ray-space hierarchy (RSH) of coherent rays that runs on the GPU. Our algorithm uses rasterization to process the primary rays, then uses those results as the inputs for a RSH, that processes the secondary rays. The RSH algorithm generates bundles of rays; hashes them, according to their attributes; and sorts them. Thus we generate a ray list with adjacent coherent rays to improve the rendering performance of the RSH vs a more classical approach. In addition the scenes geometry is partitioned into a set of bounding spheres, intersected with the RSH, to further decrease the amount of false ray bundle-primitive intersection tests. We show that our technique notably reduces the amount of ray-primitive intersection tests, required to render an image. In particular it performs up to 50% better in this metric than other algorithms in this class.



**Keywords:** Rasterization, Ray-Tracing, Ray-Hashing, Ray-Sorting, Bounding-Cone, Bounding-Sphere, Hierarchies, GPU, GPGPU

# Contents

# List of Tables

# List of Figures

# Nomenclature

**Abreviações**

GPGPU General-purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

RSH Ray-Space Hierarchy

# Chapter 1

# Introduction

In Naive Ray-Tracing (RT) each ray is tested against each polygon in the scene, this leads to N x M intersection tests per frame, assuming that we have N rays and M polygons. Performance is thus low, especially with moderately complex scenes due to the sheer amount of intersection tests computed. To optimize this naive approach (and RT in general) there are two common approaches to reduce the number of intersection tests, which are the bottleneck of the algorithm, Object Hierarchies and Spatial Hierarchies. Our work instead focuses on Ray Hierarchies and how to optimize them. This is a less well explored area of the RT domain and one that is complementary to the Object-Spatial Hierarchies.

This paper presents the Coherent Ray-Space Hierarchy (CRSH) algorithm. CRSH builds upon the Ray-Space Hierarchy (RSH) [1] and Ray-Sorting algorithms [2]. RSH, described by Roger et al., uses a tree that contains bounding sphere-cones that encompass a local set of rays. The tree is built bottom-up and traversed top-down. Our CRSH algorithm adds Ray-Sorting, described by Garanzha and Loop, to the mix in order to achieve higher efficiency in each tree node and then expands on this basis with whole mesh culling and improved hashing methods.

## 1.1 Objectives

We hypothesize that improving the coherency of the rays contained within each tree node shall lead to tighter bounding sphere-cones which, in turn, should reduce the amount of ray-geometry intersections. We use specialized ray hashing methods, tuned to the ray types we enumerated (e.g. shadow, reflection and refraction), to further improve the efficiency of the hierarchy. Finally we also introduce whole mesh bounding spheres to reduce even further the number of intersection tests at the top level of the hierarchy. This shallow spherical BVH allows us to further reduce the amount of ray-primitive intersection tests. We note that our technique uses rasterization to determine the primary intersections thus reserving the use of RT for secondaries.

Our main contributions are:

- a compact ray-space hierarchy (RSH) based on ray-indexing and ray-sorting.
- the novel combination of ray-sorting [2] with ray-space hierarchy techniques [1] to reduce the amount of ray-primitive intersections.
- culling whole meshes from the RSH prior to performing the final per primitive traversal.

## 1.2 Outline

This dissertation is subdivided into 5 different chapters. The first chapter introduces ray-tracing and ray-space hierarchies as well as the goals of this work. The second chapter provides some insight on previous work in the area. The third chapter describes the algorithm, first by giving an overview and then explains each step in detail. The fourth chapter provides the evaluation methods and results as well as discussing those results. The final section concludes this work by providing a final overview on the work and introducing ideas for future work.

# Chapter 2

# Background

## 2.1 Ray-Tracing

Ray-tracing [3] is a global illumination [4] technique that is used for the synthesis of realistic images which employs recursive ray-casting [5].

The ray tracing algorithm casts primary rays from the eye but does not stop there. When the rays intersect geometry they can generate extra secondary rays: e.g. shadow, reflection and refraction rays.

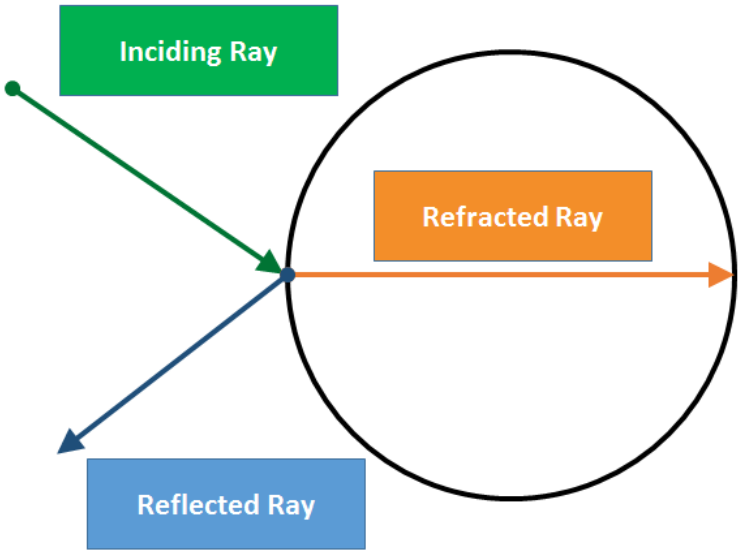



Figure 2.1: Secondary Rays

These rays differentiate ray-tracing from the more traditional rasterization algorithm since they naturally allow realistic and complex effects like reflections, refractions and shadows without any need for additional techniques. However this comes with at price. Ray-tracing is computationally expensive.

There is extensive research and ongoing to try and optimize it. Much of this research involves creating hierarchies in either the Object or Spatial domains to decrease the number of intersection tests in a divide and conquer fashion.

Object and Spatial Hierarchies (like Octrees and Bounding Volume Hierarchies) help reduce the amount of intersections by reducing the amount of scene geometry required to test by culling many polygons and objects away from the ray paths at once (see Figure 2.2 and Figure 2.3).

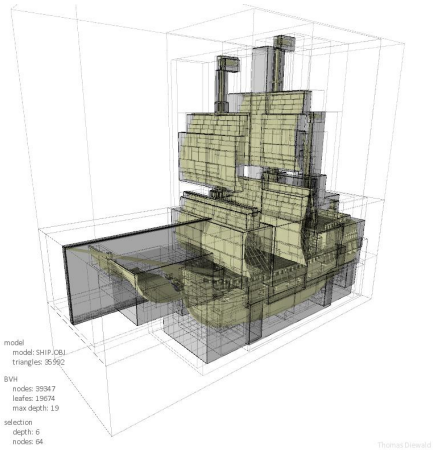


Figure 2.2: Octree

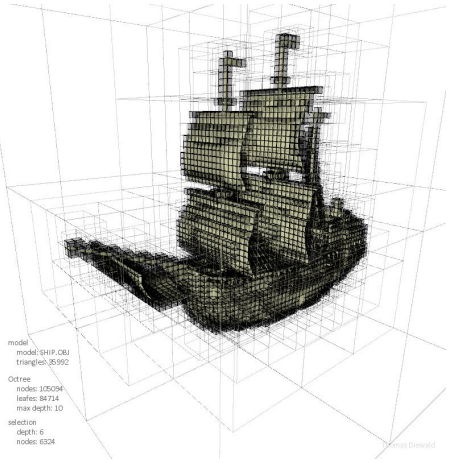


Figure 2.3: Bounding Volume Hierarchy

Ray-Space Hierarchies, use ray bundles or ray caching mechanisms to achieve the same goal, reduce the number of intersections. Instead of creating hierarchies based on the scenes geometry, they are based on the rays being cast in each frame. This is the approach we decided to take is based on ray bundling but also uses a feature of ray caching, which is ray hashing [6] [7].

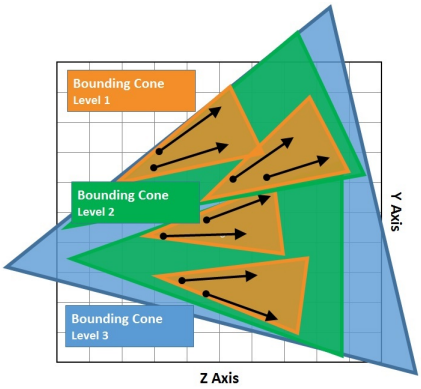



Figure 2.4: Bounding Cone

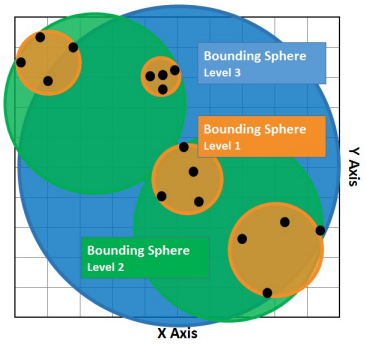



Figure 2.5: Bounding Sphere

## 2.2 Fast Ray Tracing by Ray Classification

Arvo and Kirk [6] introduced a 5-dimensional ray-space hierarchy. They subdivided the ray-space in such a way that all relevant rays are distributed among equivalence classes. These classes are associated with a candidate object set. For example, the candidate set $C_i$ contains the objects that the rays in the equivalence class $E_i$ can intersect. The focus of this algorithm is to decrease the number of intersection tests.

Since this algorithm works in a 5-dimensional space, the first thing we need to know is which are the degrees of freedom that a ray actually has. Traditionally rays are represented by a point and a vector in a 3-dimensional space, representing its origin and its direction. However a ray only has 5 degrees of freedom. These 5 degrees are its origin, which accounts for 3 degrees, and its direction, which accounts for 2 degrees when using spherical coordinates. This allowed Arvo and Kirk to parametrize rays and thus create subsets of the rays 5-dimensional space in such a way that rays with similar origins and directions would be in the same subset.

The algorithm was broken into 5 steps that I will now describe. It is also important to note that the first step is only carried out once while the second and third steps are consistently carried out in order to refine the hierarchy.

#### 1. Bounding Volume Creation

The first step is to find the 5-dimensional bounding volume that contains the equivalent of each ray that can interact with the environment in said space. Since it's important to use volumes that don't take up too much space they chose to store 5 ordered pairs that represent intervals in the 5 orthogonal axes, which they labeled x,y,z,u and v. These 5 ordered pairs represent a bounding volume. Each of these bounding volumes, denominated hypercubes, have a representation in 3-dimensional space which they call a beam. This is vital to the algorithm because this beam comprises the points in 3-dimensional space that the set of rays contained in the hypercube can reach. One of the issues however is that the beams associated with the hypercubes are not generally polyhedra. To account for this fact, they used 6 different mappings that have the desired properties locally and together contain the whole ray-space. Each of the 6 mappings correspond to a dominant axis, x+,x-,y+,y-,z+ or z-, and one sixth of the direction space. The bounding volume is then created by creating six copies of the bounding volume created from the bounding box containing all the geometry. The rays are then mapped onto these bounding volumes.

#### 2. Space Subdivision

The second step is to subdivide the 5-dimensional space into equivalence classes. The smaller each hypercube is, the more likely rays are to behave coherently. However, the smaller each hypercube is, the more classes we will have. The goal is to have hypercubes sufficiently small so that the rays of the corresponding beams intersect approximately the same objects, allowing the rays in that beam to share a set of candidate objects. Starting with the 6 bounding volumes created earlier, the hierarchy is created

by subdividing each hypercube in half at each level, cycling each of the 5 axis. This continues until either the hypercubes size or the sets number of candidate objects fall below a fixed threshold. It is important to note that hypercubes are only subdivided if they contain rays generated during the ray-tracing process.

#### 3. Candidate set Creation

For the third step each objects bounding box is intersected with the beam associated with the hypercube being tested. If there is an intersection then the object is added to that hypercubes candidate set. As the space is subdivided the candidate sets for the child hypercubes must be updated. Since there is an overlap between parent and child hypercubes they found that re-evaluating the candidate sets should only be done only after the hypercubes have been divided on all of the 5 axis, meaning a re-evaluation only after 5 subdivisions have occurred.

#### 4. Ray Classification

The fourth step consists of finding the 5-dimensional hypercube that each ray corresponds to. This is done by mapping the ray to the 5-dimensional space and then traversing the ray-space hierarchy, beginning with the hypercube that is indexed by the rays dominant axis (this hypercube is one of the 6 copies created at the start of the algorithm) until the leaf containing the ray is reached. If the candidate set corresponding to this leaf is empty then the ray intersects nothing. Otherwise, we proceed to the next step.

#### 5. Intersection Tests

Finally the ray is tested with each object for intersections. Initially a coarse intersection test is carried out by using the objects bounding volume instead of the actual object, in order to reject rays that don't intersect the object.

### 2.2.1 Considerations

This algorithm works similarly to a Hash Tree, in the sense that each ray is converted to a key (in this case the corresponding 5-dimensional point) and after finding out which class it belong to (the classes can be associated with the equivalence classes) it is then mapped to a value (in this case the candidate object set). For dynamic scenes it would be necessary to recalculate the candidate sets each time the objects moved, which would be very costly. This means that the ray-space hierarchy is tightly connected to the objects and their positioning in the scene. This suggests that the algorithm is not a very good fit for ray-tracing of dynamic scenes, which is my goal. The ray caching scheme is interesting however if used differently, i.e. for the sorting of secondary rays.

## 2.3 Five-dimensional Adaptive Subdivision for Ray Tracing

Simiakakis and Day [8] introduced some improvements to Arvo and Kirks work [6]. These improvements consist of using a directional subdivision method, which adapts automatically according to the scene. They also introduced a memory saving scheme but the main issues with Arvo and Kirks paper with dynamic scenes still remain, which does not make this algorithm a good fit for dynamic scenes.

## 2.4 Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

Roger et al. [1] presented an algorithm for interactive rendering of animated scenes with Whitted Ray-Tracing. This algorithm ran fully on the GPU and focuses on the secondary rays, using the GPU rasterizer to process the primary rays. They use a ray-space hierarchy that bundles rays together using cones and spheres. The algorithm is mapped on the GPU using textures to pass information between each of the algorithms phases.

The algorithm consists of 5 steps that i will describe. It is important to keep in mind that the first step uses the GPU rasterizer since it has the same effect as tracing the primary rays but rasterizing them is faster.

#### Render the Scene using the Rasterizer

This step is done using the traditional rasterizer (for example using modern OpenGLs rasterization pipeline) that outputs the pixel color, without any kind of secondary ray effects like shadows, reflections and refractions. Like it was stated before, the primary rays give us the same pixel color as the standard rasterizer, so they took advantage of the performance of the rasterizer to speed-up the algorithm.

#### Generate the first set of Secondary Rays

Since fragment shaders can have multiple render targets, they take advantage of this fact to output the initial set of secondary rays as well. These rays can consist of the shadow rays, reflection rays and refraction rays.. Since the algorithm is generic enough, it does not matter which kind of secondary rays are used. It is also worth noting that the shadow rays are very coherent locally (if we're using soft shadows) so they fit cone hierarchy quite well.

#### Build the Ray-Space Hierarchy

The second step is taking the rays generated from the previous step and constructing the hierarchy. Each node consists of a structure composed by a sphere and a cone, which means that 8 floats are stored, 3 for spheres center and 1 for the spheres radius, 3 for the cones direction and 1 for the cones spread angle.i The hierarchy is constructed bottom-up, meaning that the leaves are constructed first.

These leaves consist of spheres with radii 0 and cones with spread 0, representing the exact rays. The upper levels are then constructed by computing the union of the child nodes until we have a top node that encloses the whole ray-space.

#### Intersect the Ray-Space Hierarchy with the Scene

In this step the ray-space hierarchy is intersected with the scene. This is done in a top-down manner, testing each parent node before going down the hierarchy. Only triangles that intersect the parent node will be tested for intersections with the children nodes. This means that at each level of the hierarchy several triangles are removed from the intersection tests. It is also important to note that for this preliminary intersection phase the tests are carried out using the triangles bounding spheres since they are simply exclusion tests before the final local intersection tests. At each level of the hierarchy the intersection information is stored in textures, one for each of the 4 children nodes. If there is an intersection, the IDs for the triangle and the cone are stored in a texture, otherwise an empty element is stored. Since the texture will have blanks when there are failed intersection tests. These 4 textures also need pruned of empty elements for the next pass.

This has two purposes, one being memory management and the other being the mapping of the algorithm to the GPGPU. Since each pass stores its output in textures we can process each pass using the same operations meaning that there won't be threads having different execution paths which in turn is one of the features necessary to make an algorithm fit the GPU. These pruning methods are called stream-reduction passes and they presented two different methods, one involving an hierarchical version of Horns [9] and a slower one using geometry shaders. The hierarchical version works by breaking down the textures in components of fixed size and concatenating them at the end.

#### Final Intersection Tests

Finally after traversing the hierarchy we have a texture with the triangle and ray IDs (the last level of the hierarchy are individual rays), which means we need to calculate the traditional intersection tests, keeping the closest intersection.

#### Considerations

This algorithm works as a bounding volume hierarchy of sorts, adapted to the rays, using cones and spheres to enclose the ray bundles. It is also very well adapted to GPGPU since each pass has a fixed number of operations and has the same execution path. The stream-pass reduction method is vital since it helps prune the textures of empty elements, allowing the threads processing the textures to execute the same operations. Finally it leaves the possibility of combining the ray-space hierarchy with a object-space hierarchy, possibly reducing the number of intersections even more. It is also an example of hybrid rendering since it uses the GPUs rasterizer to compute the initial rays and the initial set of secondary rays.

## 2.5 Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing

Garanzha and Loop [2] introduced an algorithm using various parallel primitives to adapt the ray-tracing algorithm to the GPU. Their algorithm sorts the generated rays and creates tight-fit frustums on the GPU and then intersects them with a Bounding Volume Hierarchy for the scenes geometry that is built on the CPU. One of the most interesting parts of their work is the fast ray sorting since it is done using parallel primitives which mitigates the overall cost of the operation when its executed on the GPU. I will now go into more detail regarding the four stages of the algorithm that they outlined.

#### Ray Sorting

The first step, Ray Sorting, is used to store coherent rays in coherent memory locations in order to have less divergence during the tracing routine. If any given set of rays are coherent, they are more likely to intersect the same geometry and thus access the same memory locations. The sorting also allows the frustums in the next step to be tighter fits and thus give us a smaller set of intersection candidates.

To sort the rays, they created a key-index pair for each ray, with the key being the rays ID and the index being the rays Hash value. This hash value is computed by quantizing the rays origin in a virtual grid that is within the scenes bounding box and then by quantizing its direction using a similar process. After creating this pair for every ray, a compression-sorting-decompression scheme is used. This compression step generates chunks out of the initial set of rays. These chunks are represented by three arrays, the hash array (which indicates hash each chunk), the base array (which indicates the index of the first ray in each chunk) and a size array. This compression step takes advantage of the fact that it is likely that rays that are generated sequentially will have the same hash value. These chunks are then sorted using radix sort. The final step is the decompression, which applies an exclusive scan on the chunks size array to generate an array that indicates where each chunk will start in the re-ordered rays set. After the exclusive scan two arrays are generated, the skeleton array that contains the base of the chunk in the positions from which the re-ordered rays set will be created from and the head flags array that indicates where new chunks start. An inclusive segment scan is then applied to these two arrays and the final re-ordered ray set is built. The final part of the decompression step is extracting the ranges of the packets of coherent rays. This is done by compressing the re-ordered ray set, using the same procedure that was used for the un-ordered ray set and then dividing those compressed chunks into packets, with a maximum size that is defined manually and then by decompressing using an inclusive segmented scan once more.

#### Frustum Creation

The frustum creation step picks up where the sorting left off. From the packet ranges calculated in the previous step a frustum is created for each one, using a dominant axis and two axis aligned rectangles to represent them, bounding each ray in the packet.

#### Frustum Traversal

The frustums are then intersected with the scenes Bounding Volume Hierarchy that was created on the CPU. This binary BVH has arity eight and 2/3rds of the tree levels are eliminated and an Octo-BVH is created. After traversing the BVH in a breadth-first order, each frustum will have a list of leaves assigned to it, which contain the primitives that will be used in the next step.

#### Local Intersection Tests

Finally the leaves are iterated on and for each of the primitives that they contain, localized intersection tests are calculated.

#### Considerations

This algorithm does not create a global Ray-Space hierarchy that contains every single ray. Instead, smaller local Frustums are created, taking advantage of the Ray-Sorting step to make them tight-fits. This reduces the number of leaves that each frustum will have assigned to it after traversing the Bounding Volume Hierarchy, which in turn reduces the number of localized intersection tests. The Ray-Sorting step is very interesting as it uses parallel primitives such as segmented scans to maximize its effectiveness on the GPU. This approach can be merged with Roger et al's [1] algorithm in order to create a more efficient RSH.

# Chapter 3

# Algorithm

Our algorithm (refered to as CRSH from now on) creates a Coherent Ray-Space Hierarchy that takes advantage of the properties of highly coherent rays to render photo-realistic images. It consists of seven major steps (see Figure 3.1).

We use the GPUs rasterizer to compute the primary rays and then output the necessary information to create the initial batch of secondary rays. We then sort those rays, create the hierarchy and traverse it. The final step consists in intersecting the bottom-level of the hierarchy with the geometry and accumulating shading.

In each frame steps 1 and 2 are executed only once while steps 3 through 7 are executed once per ray batch. A ray batch can consist of any combination of shadow rays, reflection rays or refraction rays. This means that we can use batches that only contain shadow rays, only reflection rays, only refraction rays or any combination of the aforementioned types.

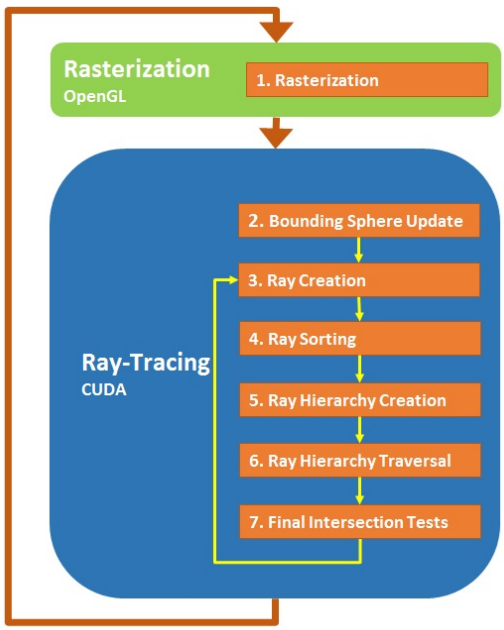



Figure 3.1: Coherent Ray-Space Hierarchy Overview.

## 3.1 Rasterization

Rasterization is the first step of the algorithm. There was an ongoing urban legend that Rasterization and Ray-Tracing are polar opposites and that a choice has to be made between these two techniques. This is not true at all. Although Rasterization solves the rendering problem conversely vs Ray-Tracing (i.e. projecting primitives to the screen, vs projecting rays backwards to the primitives in the scene), one can complement the other. The first set of rays, the primary rays, does not convey any of the global illumination effects that Ray-Tracing is well suited to achieve, such as Shadows, Reflections and Refractions. This means that Rasterization can convey similar visual results to tracing the primary rays, while being extremely faster and optimized in the graphics hardware. Supplementing the Rasterization of primary rays with the Ray-Tracing of secondary rays can get us the benefits from both techniques: the efficiency of Rasterization and the global illumination effects from Ray-Tracing.

In order to combine both techniques the output from the fragment shaders used to rasterize the scene must be more than just the traditional fragment colors. We need to create different render targets according to the information we want to store. In our case we output the fragment diffuse and specular properties, the fragment position and the fragment normal. In our implementation the fragment shader outputs 4 different textures containing four 32 bit floats per pixel each. These textures are generated with OpenGL/GLSL and are then sent to CUDA to create the first level of secondary rays.

The first pair of textures, the diffuse and specular properties textures, are used to calculate the shading of the primary rays (see Figure 3.2 and Figure 3.3). Although the primary rays are computed in the rasterizer, it is also important to note that after every batch of rays (primary or secondary) we need to trace shadow rays to determine the shading of those rays. We could store the triangle ID instead of the diffuse and specular properties but we chose the latter two instead. We store these two values instead of the triangle IDs so that we can prevent memory accesses and calculations. If we chose to output the triangle ID this would require us to calculate the triangle ID in the context of the scene (the rasterizer computes one object at a time therefore an extra step would be necessary). It would also require three different global memory accesses (the triangle IDs texture, the diffuse properties array and the specular properties array) compared to the two accesses if we just access the material values directly. The size of the two textures are also negligible when compared to the total memory requirements of the algorithm, therefore there is no reason not to use these two textures instead of one that contains the triangle IDs.

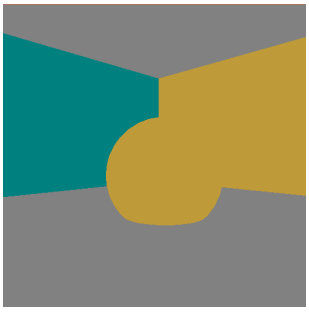


Figure 3.2: Diffuse Texture Render

Figure 3.3: Specular Texture Render

The second pair of textures, the fragment position and normal textures, contain the information necessary to calculate the secondary rays (see Figure 3.4 and Figure 3.5). These include the shadow, reflection and refraction rays. However it is important to note that to calculate the shadow rays it is only necessary to output the fragment position, since the shadow ray directions are determined by the light and fragment positions. This means that the fragment normal is only needed for the reflection and refraction rays.

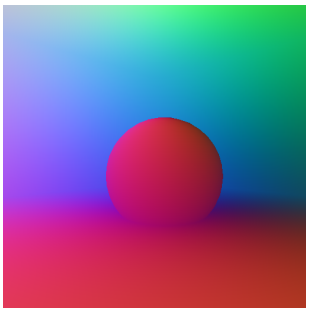


Figure 3.4: Fragment Position Texture Render

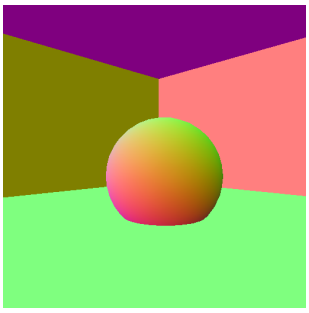


Figure 3.5: Fragment Normal Texture Render

## 3.2 Ray-Tracing

### 3.2.1 Bounding Volume Creation

Although this is a pre-processing step it makes sense to mention it now since in the following step we will update the object-based bounding spheres. We pre-compute the minimal bounding sphere of the object meshes, using an implementation based on [10]s algorithm (see Figure 3.6). It is important to use the minimal bounding spheres because these spheres will be used later on in the algorithm (in the hierarchy traversal step) to cull potential intersections in an early phase and thus reduced the total number of intersection tests computed. Since these are all pre-processed they have no impact on render time performance.

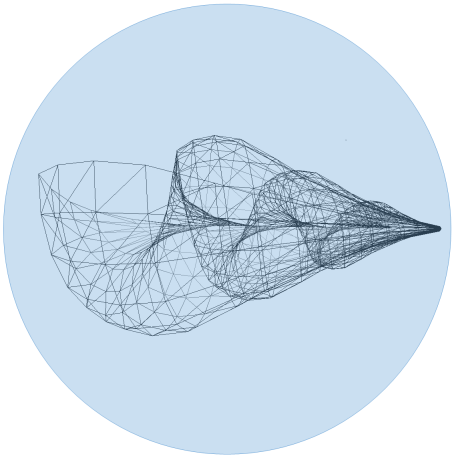


Figure 3.6: Object Bounding Sphere

### 3.2.2 Bounding Volume Update

In this step we update the object bounding spheres according to the transformations being applied to the object they envelop (e.g. translation, rotation, scale). Since we only update the center and the radius there is no need to recalculate the bounding spheres in each frame (the transformations do not invalidate the bounding spheres themselves).

### 3.2.3 Secondary Ray Creation and Indexing

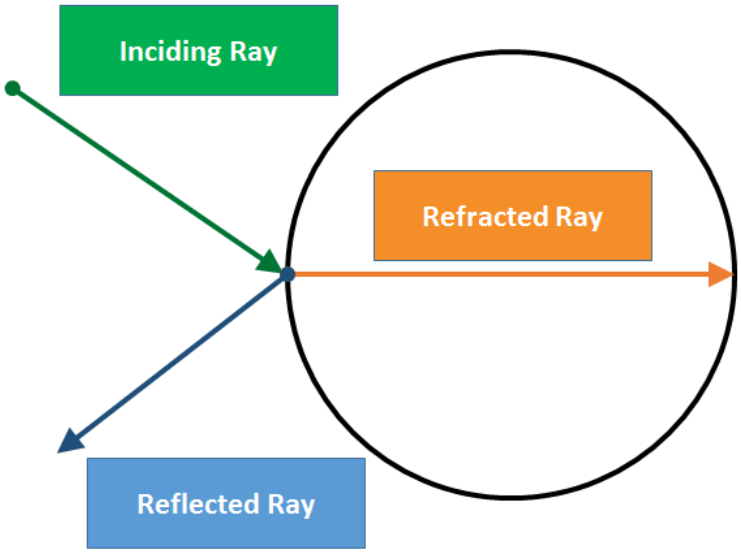



Figure 3.7: Secondary Rays

After the updating the bounding spheres we need to create the secondary rays. This step includes both the creation and the indexing of said rays. This step can be executed right after the rasterization step or after the final intersections step, depending on the ray-tracing depth being used. The ray index will be used later on in the algorithm in the sorting step. Since this is the first of the repeatable steps in our algorithm we will present both cases, the one where the rays are created right after the rasterization step as well as the one where rays are created after processing a ray batch.

#### Secondary Ray Creation

We will now present the formulas used to calculate the secondary rays. After the Rasterization step they are created by using the textures that contain the fragment positions and normals. If this is executed after processing another ray batch then the rays are created using the information obtained from the final intersection tests step (this information is still the fragment position and normals, it only means that the rays are created right after the shading step).

For that reason, the formulas are generic enough so that they can be used for both steps without any need for modifications. After this step it is irrelevant if we are processing the first batch of secondary rays or not since they are all processed in the same way, up until the final intersection tests.

The following formula is used to create the shadow rays:

$$o = l \tag{3.1}$$

$$\vec{d} = \frac{\vec{p} - \vec{o}}{|\vec{p} - \vec{o}|} \tag{3.2}$$

$$\mathbf{o} = Origin, \tilde{\mathbf{d}} = Direction, \mathbf{p} = FragmentPosition, \mathbf{l} = LightPosition$$

The following formula is used to create the reflection rays:

$$o = p \tag{3.3}$$

$$\vec{d} = \vec{i} - 2 \times \vec{n} \times (\vec{n} \cdot \vec{i}) \tag{3.4}$$

$$\mathbf{o} = Origin, \tilde{\mathbf{d}} = Direction, \mathbf{p} = FragmentPosition, \mathbf{i} = IncidentRayDirection$$

The following formula is used to create the refraction rays:

$$o = p \tag{3.5}$$

$$k = 1 - ri^2 \times (1 - (\vec{i} \cdot \vec{n})^2) \tag{3.6}$$

$$\vec{d} = \vec{i} \times ri - \vec{n} \times (ri \times (\vec{i} \cdot \vec{n}) + \sqrt{k}) \tag{3.7}$$

$$\mathbf{o} = Origin, \tilde{\mathbf{d}} = Direction, \mathbf{p} = FragmentPosition, \mathbf{i} = IncidentRayDirection, \mathbf{ri} = RefractionIndex$$

### Secondary Ray Indexing

We create an index for each individual ray in order to sort the secondary rays in the following steps. We use a different hashing function for each type of ray (see Figures 3.8, 3.9). Since each ray consists of an origin and a direction it would be straightforward to simply use these two parameters to create our hash. However for shadow rays it is sufficient to use the index of the light source to which it belongs and its direction. This is only possible however if we invert the origin of the shadow ray so that it originates at the light source rather than the originating fragment. To further reduce the size of the hash keys we convert the ray direction into spherical coordinates [11] and store both the light index and the spherical coordinates into a 32 bit integer, with the light index having the higher bit-value such that the shadow rays are sorted according to the light source a priori.

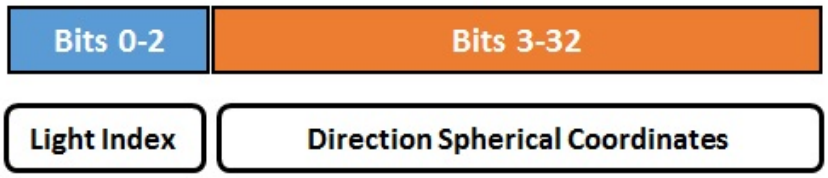



Figure 3.8: Shadow Ray Hash.

The Reflection and Refraction rays are also converted to spherical coordinates. However in this case the ray origin is used in the hash instead of being discarded, given that these rays are not as coherent in regards to their origin. These rays are only coherent locally, for example if they are all originating from the same area in the same object, unlike shadow rays which have the same origin for each light source. This also means that the indexing step is extremely important for our algorithm because if we can sort these rays optimally we will also get an optimal hierarchy, leading to a greatly reduced number of intersection tests.

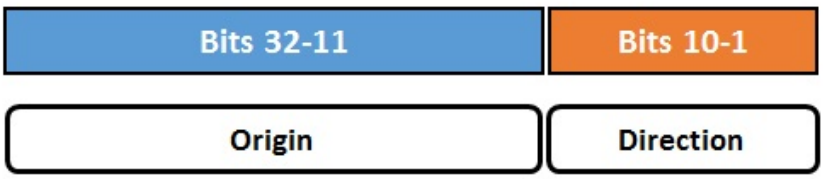



Figure 3.9: Reflection and Refraction Ray Hash.

### Secondary Ray Trimming

Once the secondary rays are created and indexed, we have four arrays, the ray array that contains the generated secondary rays, two arrays that contain the ray keys (the indices's we calculated) and the ray values (the rays position in the ray array) and a final array with head flags which indicate if there is a ray in the corresponding position within the key-value arrays, where we store either a 0 or a 1, indicating if there is a ray or not, respectively. This is an extremely important step for parallel programming since by

doing this we can deduce the number of rays (by checking the last position of the scan array) as well as having all the rays that need to be processed in adjacent positions, meaning there will be no threads processing the empty spaces in the arrays.

Using the information from the head flags array we then run a trimming operator on the key-value arrays (see Figure 3.10). This is done by first applying an inclusive scan operator [12] on the head flags array, which gives us the number of positions that each pair needs to be shifted to the left. This is done in order to trim the arrays [9].

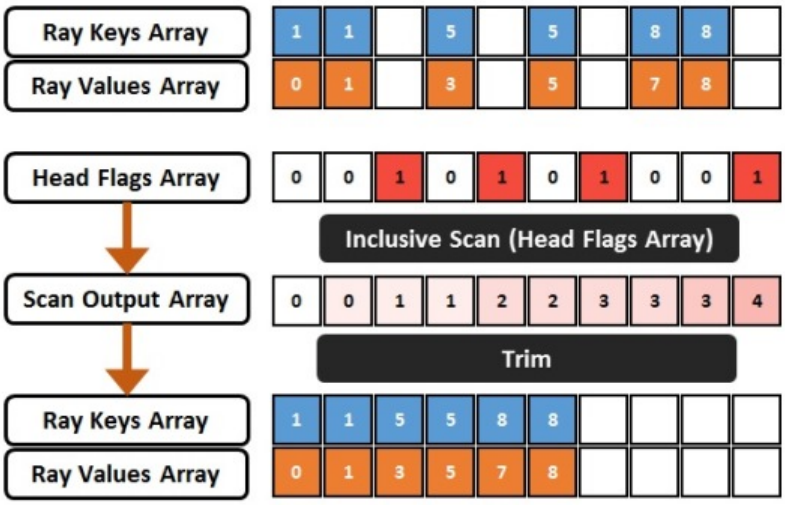



Figure 3.10: Array Trimming

### 3.2.4 Secondary Ray Compression

After trimming the secondary rays we used a compression-sorting-decompression scheme, expanding on prior work by Garanzha and Loop [2]. The compression step exploits the local coherency of rays. Even for secondary rays, the bounces generated by two adjacent rays have a good chance of being coherent (i.e. a sphere reflecting rays around an area, see Figure 3.11).

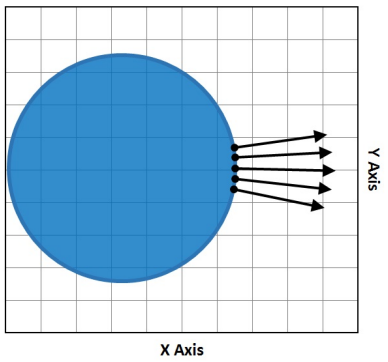



Figure 3.11: Ray Reflection

This coherency means that these rays can end up having the same hash value. Given these rays with the same hashes, we compress the ray key-value pairs into chunks, minimizing the number of pairs that need to be sorted. To compress the pairs we utilize a head flags array with the same size as the key-value pair array, initializing it with $0$s in every position and inserting $1$s into positions in which the key (hash) of the corresponding pair differs from the previous pair. After populating the head flags array we apply an inclusive scan operator on it [12]. By combining the head flags array with the scan output array we create the chunk keys, base and size arrays, which contain the hash, starting index and size of the corresponding chunks (see Figure 3.12). The chunk keys are represented in different colors at the image below. The chunk base array represents the original position of the first ray in the chunk while the chunk size array represents the size of the chunk, needed for the ray array decompression.

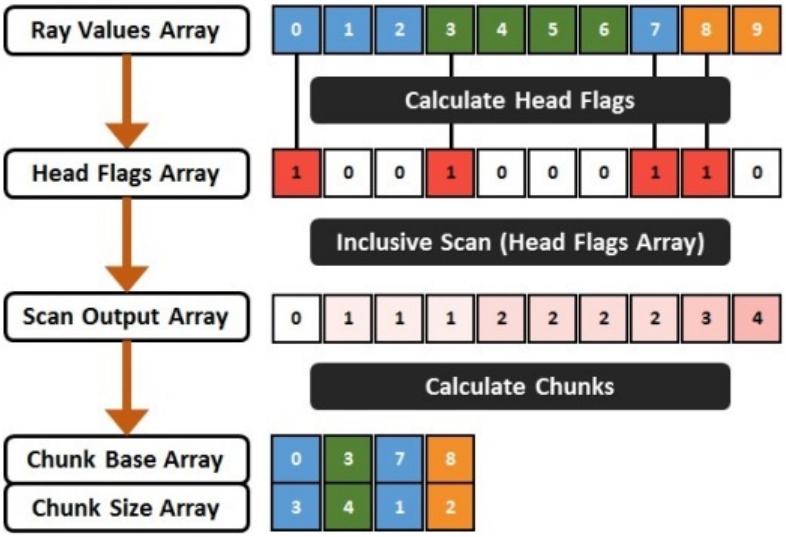



Figure 3.12: Ray Compression into Chunks

It is important to note that even though there are only the chunk base and size arrays in Figure 3.12 the compression step also creates two more arrays, the chunk keys array (that contains the indices's of the corresponding rays) and the chunk values array (that contains the position of the corresponding chunk key, initially a sorted array). These arrays will be used for the sorting step.

### 3.2.5 Secondary Ray Sorting

After compressing the rays we have the chunk base and size arrays that contain the necessary information to reconstruct the initial rays array and the chunk keys and values arrays that contain the necessary information for the sorting process. With the latter two we apply a radix sort [13] (see Figure 3.13) and obtain two sorted arrays containing the ray keys and values. The ray keys array is irrelevant after this stage since it was only necessary to sorting the ray values array.

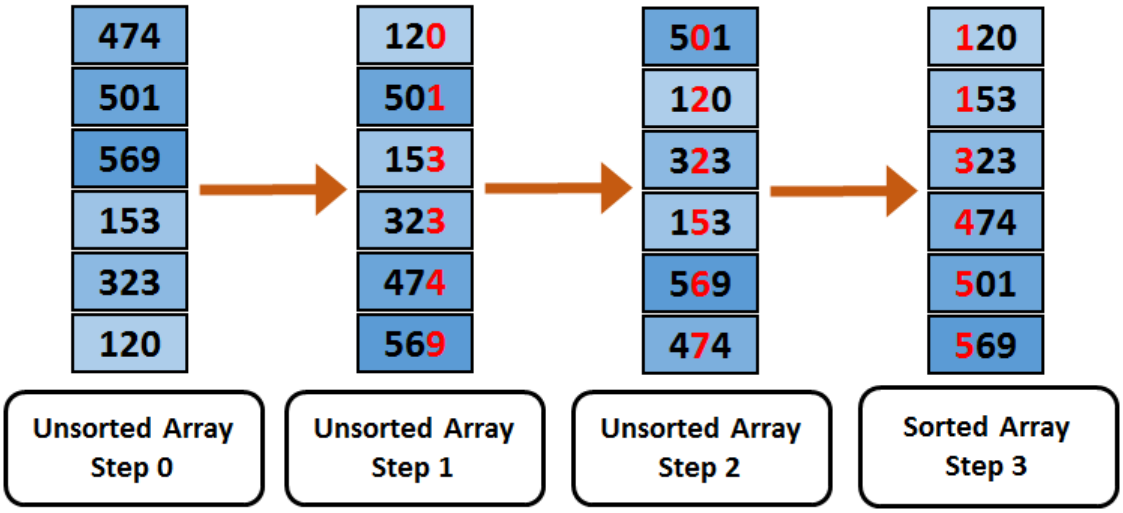



Figure 3.13: Radix Sort

### 3.2.6 Secondary Ray Decompression

Finally the chunk decompression begins with the creation of a skeleton array with the same size as the sorted chunk base and size arrays. This skeleton array contains the size of the sorted chunks in each position. Next we apply an exclusive scan operator on the skeleton array, originating a scan array. This scan array gives us the starting positions positions of each chunks in the sorted ray key and value arrays. After creating these two arrays (the skeleton and scan arrays) we fill the sorted ray array. We start in the position indicated in the scan array and finish after filling the number of rays indicated in the skeleton array. Although Figure 3.14 has the chunk size array with the exact same values as the skeleton array its important to note that this is simply a representation for easier understanding since at this point we do not have a sorted chunk size array but instead a sorted values array that when combined with the chunk size array we created when we compressed the rays into chunks will give us the sorted chunk sizes.

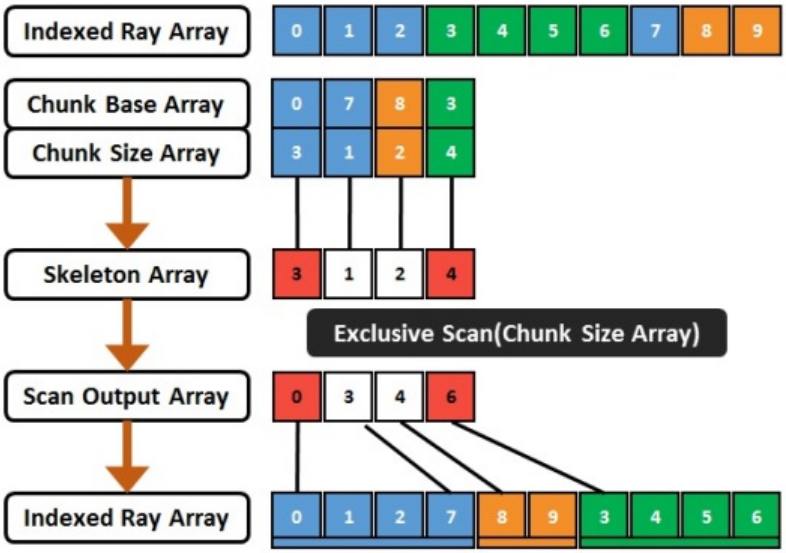



Figure 3.14: Ray Decompression from Chunks

To summarize these last three sections, we first compressed the rays into chunks, we sort the chunks are finally decompressed the sorted chunks (see Figure 3.15).

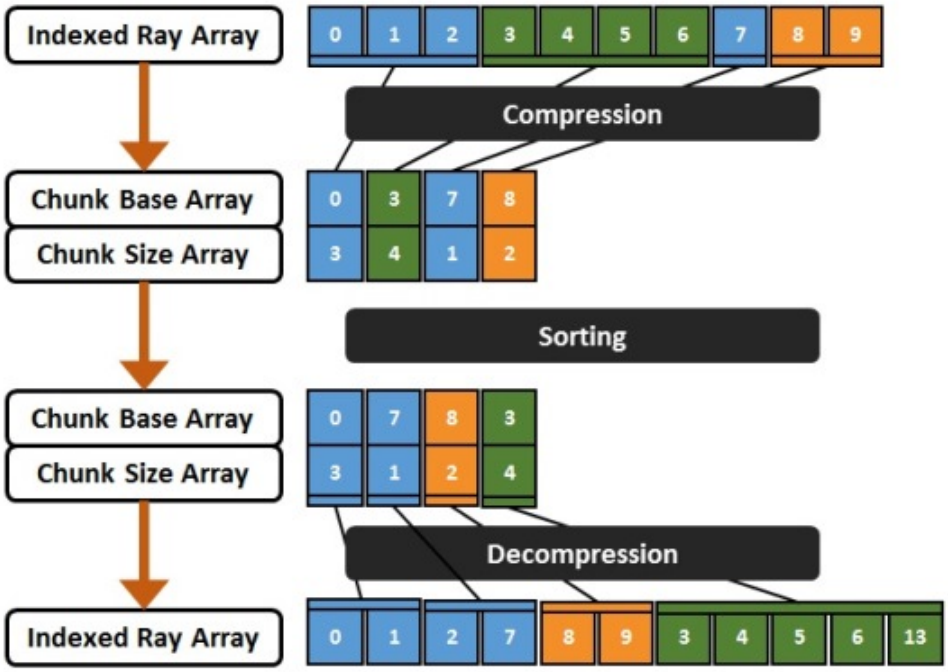



Figure 3.15: Sorting Overview

### 3.2.7 Hierarchy Creation

With the sorted rays we can now create the ray hierarchy. Since the rays are now sorted coherently the hierarchy will be much tighter in its lower levels, giving us a smaller number of intersection candidates as we traverse further down the hierarchy.

Each node in the hierarchy is represented by a sphere and a cone (see Figures 3.16, 3.17). The sphere contains all the nodes ray origins while the cone contain the rays themselves (see Figure 3.18). This structure is stored using eight floats: the sphere center and radius (four floats) and the cone direction and spread angle (four floats). The construction of the hierarchy is done in a bottom-up fashion. Thus we start with the leaves, with spheres of radius $0$ and a cone with spread angle equal to $0$. These leaves correspond to the sorted rays. The upper levels of the hierarchy are created by calculating the union of the child nodes. The number of children combined in each node can also be parametrized.

It is important to take into consideration that since the hierarchy is not tied directly to the geometry positions in the scene it does not matter for hierarchy creation whether the scene is dynamic or static. This means that creating the scene with a moving object does not impact performance at all. What does impact performance is the number of secondary rays generated and this can be affected by the number of objects in the screen (more specifically the number of fragments that each object creates, which directly correlates to the number of secondary rays generated). The only thing that does matter

directly is the number of bounces of each ray, meaning that if there are more pixels occupied in the screen, the hierarchy will forcibly have more nodes.

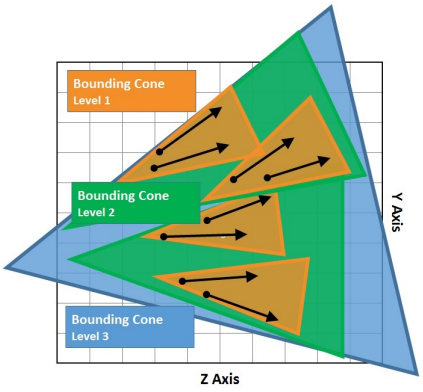



Figure 3.16: Bounding Cone - 2D View

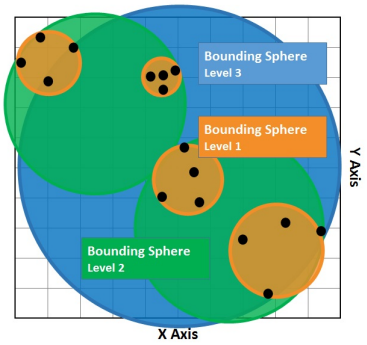



Figure 3.17: Bounding Sphere - 2D View

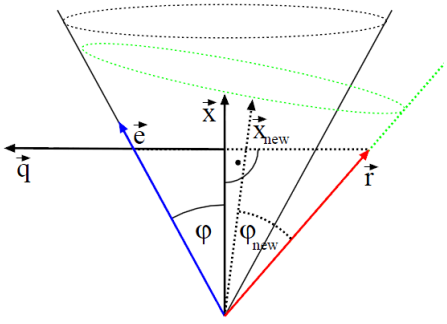



Figure 3.18: Cone-Ray Union - 2D View. courtesy of [14].

We will now present the formulas used for the creation of each individual node. For the first level of nodes we use a different formula for the creation of the cones. This is done because the first level is a particular case. The leaves of the hierarchy are simply rays therefore it is more efficient to used a different formula ir order to create more compact cones [14].

The following formulas are used to create the first level nodes:

$$\vec{q} = \frac{(\vec{x} \cdot \vec{r}) \cdot \vec{x} - \vec{r}}{|(\vec{x} \cdot \vec{r}) \cdot \vec{x} - \vec{r}|} \tag{3.8}$$

$$\vec{e} = \vec{x} \cdot \cos(\phi) + \vec{q} \cdot \sin\phi \tag{3.9}$$

$$\vec{x}_{new} = \frac{\vec{e} + \vec{r}}{|\vec{e} + \vec{r}|} \tag{3.10}$$

$$\cos\phi_{new} = \vec{x}_{new} \cdot \vec{r} \tag{3.11}$$

For the remaining levels we use the following formulas for combining cones:

$$\vec{x}_{new} = \frac{\vec{x_1} + \vec{x_2}}{|\vec{x_1} + \vec{x_2}|} \tag{3.12}$$

$$\cos\phi_{new} = \frac{\arccos(\vec{x_1} + \vec{x_2})}{2} + \max(\phi_1, \phi_2) \tag{3.13}$$

Finally for the union of the spheres we use this formula:

$$center_{new} = \frac{center_1 + center_2}{2} \tag{3.14}$$

$$radius_{new} = \frac{|center_2 - center_1|}{2} + \max(radius_1, radius_2) \tag{3.15}$$

Roger et al. [1] also noted that some nodes might become too large as we travel higher up into the hierarchy. To mitigate this problem we decided to limit the number of levels generated and subsequently the number of levels traversed. Since rays are sorted before this step, there is much higher coherency between rays in the lower levels. If we focus on these rays and ignore the higher levels of the hierarchy we will have better results (this will be demonstrated later in the evaluation section). There is a possibility that we might end up having more local intersection tests but since the quality of the nodes in the higher levels of the hierarchy starts to degenerate, we would most likely end up having intersections with every single triangle in the scene while traversing those nodes and thus have no real gain from calculating intersections on these higher level nodes to begin with.

### 3.2.8 Hierarchy Traversal

For the top level of the hierarchy we intersect the hierarchy nodes with the object bounding spheres to cull intersections even further. After this initial step we traverse the hierarchy in a top-down order, intersecting each node with the scenes geometry. Since the top level nodes of the hierarchy fully contain the bottom level nodes, triangles rejected at the top levels will not be tested again in the bottom level nodes. Let us say we start traversing the hierarchy with the root node. If a certain triangle does not

intersect the root node then this means that that specific triangle will not intersect any of its children. Since it is the root node, it also means that no ray in the scene will intersect it so we do not have to test it for further intersections. After traversing each level of the hierarchy we store the intersection information in an array so that the child nodes will know the sets of triangles they have to compute intersections against.

The intersection tests being run at this stage are only coarse grained tests. They use the triangles bounding spheres since we will have to do the actual intersection tests in the final stage anyway. The intersection tests are being ran in a parallel manner so there is an issue regarding empty spaces in the arrays that contain the intersection information. As such these arrays need to be trimmed using the same procedure that we used after the ray generation.

These hits are stored as an int32 in which the first 18 bits store the node id and the last 14 bits store the triangle id. This is not a problem for larger scenes since those processed in triangle batches. Each hit only needs to store the maximum number of triangles per batch. This means we can process up to 16384 triangles per batch. This compression into a single integer was necessary due to the large memory constraints of the algorithm. For a scene with 100.000 triangles and an hierarchy with 10.000 nodes we might need to store up to 1.000.000.000 hits. We already allocate less memory than those 1.000.000.000 hits since the test results indicate that that maximum is never reached but we still have to compress the hits otherwise the necessary memory for this array would be far too great.

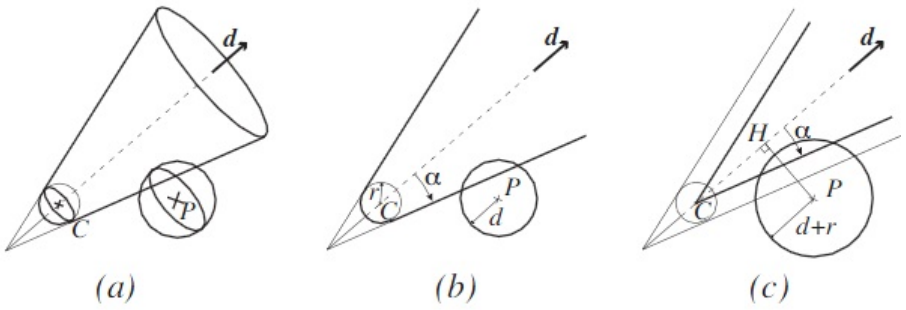



Figure 3.19: Cone-Ray Union - 2D View. courtesy of [1].

To calculate the intersection between the node, which is composed of the union of a sphere and a cone, we simplify the problem by enlarging the triangles bounding sphere [15] and reducing the cones size (see Figure 3.19). The original formula for cone-sphere intersections was described in the Amanatides paper [16]. The current formula, which expands on the work of Amanatides [16], was described by Roger et al. [1].

$$result = |C - H| \times \tan\alpha + \frac{d + r}{\cos\alpha} \geqslant |P - H| \tag{3.16}$$

### 3.2.9 Final Intersection Tests

After traversing the hierarchy we have an array of node id and triangle id pairs. The candidates for the local intersection tests [17]. This final step differs according to two things, the type of ray and the depth of ray-tracing. For shadow rays all that is necessary is to indicate whether or not the ray was blocked by any other object. In contrast, for reflection and refraction rays we need to find the closest intersection and store the corresponding triangle.

Shadow rays are only traced after tracing a batch of primary, reflection or refraction rays. As such, they use the triangles that those rays intersected instead of storing triangle intersections. Reflection and refraction rays unlike shadow rays need to store the triangle they intersected. This is necessary because after each batch of such rays we need to trace shadow rays to check whether or not the triangle has any other objects blocked its path to the light sources.

#### Barycentric Interpolation

Before calculating the shading for each ray is necessary to interpolate the hit points normal in order to have smooth shading. To interpolate the normal we use barycentric interpolation (see Figure 3.20).

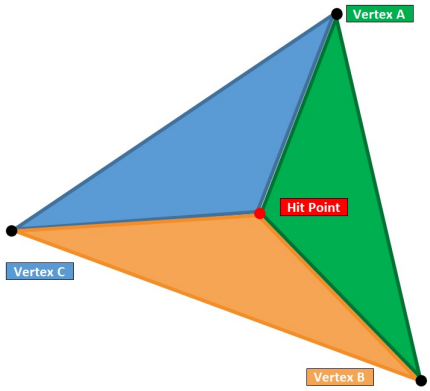



Figure 3.20: Barycentric Coordinates.

These are the equations used to calculate the interpolated normal:

$$area_{ABC} = \frac{(v_b - v_a) \times (v_c - v_a)}{|(v_b - v_a) \times (v_c - v_a)|} \tag{3.17a}$$

$$area_{PBC} = \frac{(v_b - hitP) \times (v_c - hitP)}{|(v_b - hitP) \times (v_c - hitP)|} \tag{3.17b}$$

$$area_{PCA} = \frac{(v_a - hitP) \times (v_c - hitP)}{|(v_a - hitP) \times (v_c - hitP)|} \tag{3.17c}$$

$$\vec{hitN} = \frac{area_{PBC}}{area_{ABC}} \times n_a + \frac{area_{PCA}}{area_{ABC}} \times n_b + (1 - \frac{area_{PBC}}{area_{ABC}} - \frac{area_{PCA}}{area_{ABC}}) \times n_c \tag{3.18}$$

$$\mathbf{hitP} = HitPointPosition, \mathbf{hitN} = HitPointNormal$$

$$\mathbf{v_a}, \mathbf{v_b}, \mathbf{v_c} = VertexPositions, \mathbf{n_a}, \mathbf{n_b}, \mathbf{n_c} = VertexNormals$$

#### Shading

To calculate the shading we used a Lambertian reflection and Phong illumination. This is the implementation we used:

$$light_{direction} = \frac{light_{position} - hit_{position}}{|light_{position} - hit_{position}|} \tag{3.19a}$$

$$light_{distance} = |light_{position} - hit_{position}| \tag{3.19b}$$

$$light_{attenuation} = \frac{1}{att_{constant} + light_{distance} \times att_{linear} + light^2_{distance} \times att_{exponential}} \tag{3.19c}$$

$$view\_vector = \frac{hit_{position} - camera_{position}}{|hit_{position} - camera_{position}|} \tag{3.19d}$$

$$halfway\_vector = \frac{light_{direction} - view\_vector}{|light_{direction} - view\_vector|} \tag{3.19e}$$

$$diffuse\_factor = \max(0, \min(1, light_{direction} \cdot hit_{normal})) \tag{3.19f}$$

$$specular\_factor = \max(0, \min(1, \max(0, halfway\_vector \cdot hit_{normal})^{shininess})) \tag{3.19g}$$

$$color+ = hit_{diffuse} * light_{color} * diffuse\_factor * light\_intensity * attenuation \tag{3.19h}$$

$$color+ = hit_{specular} * light_{color} * specular\_factor * light\_intensity * attenuation \tag{3.19i}$$

In this final step all that remains is to find out which is the closest intersected triangle for each ray and accumulate shading. Depending on the depth that we want for the algorithm we might need to output another set of secondary rays. Since the algorithm is generic, all that is necessary for this is to output these rays onto the ray array that we used initially and continue from the ray compression step.

# Chapter 4

# Evaluation

## 4.1 Test Methodology

We implemented our CRSH algorithm in OpenGL/C++ and CUDA/C++ then compared it with our implementation of RAH [1] over the same architecture. We map our algorithm onto the GPU, parallelizing it there fully. We achieve this mainly by the use of parallel primitives, like prefix sums [18]. We used the CUB [12] [13] library to perform parallel radix sorts and prefix sums.

We measure the amount of intersections, including misses and hits, to evaluate ray hierarchy algorithms proficiency at reducing the amount of ray-primitive intersection tests required to render an image. All scenes were rendered at $512 \times 512$ resolution. We also vary the depth of the hierarchy and the number of nodes we combine to create the upper levels of the hierarchy. We also measured the rendering time of each scene using a baseline configuration as well as the individual steps in each algorithm relatively to the overall rendering time.

The test information was collected using a NVIDIA GeForce GTX 770M GPU with 3 GB of RAM. Our algorithm is completely executed on the GPU (including hierarchy construction and traversal) so the CPU has no impact on the test results.

## 4.2 Test Scenes

We used three different scenes, OFFICE, CORNELL and SPONZA.

The OFFICE scene (36K triangles) is representative of interior design applications. It is divided into several submeshes therefore it adapts very well to our bounding volume scheme. For this scene the emphasis was on testing shadow rays.

We selected CORNELL (790 triangles). as it is representative of highly reflective scenes. It consists an object surrounded by six mirrors. On this scene we focused on testing reflection rays although it also features shadow rays in it.

SPONZA (66K triangles), much like OFFICE, is representative of architectural scenes. For this scene the emphasis was also on testing shadow rays but for scenes that do not conform with our bounding volume scheme. This scene does not adapt well to our scheme as is not divided into submeshes.

## 4.3 Test Hypothesis

We hypothesised that our more coherent RSH hierarchy needs to compute fewer intersection results to render a scene. We expect more expressive results for shadow rays. As shadow rays have low divergence the sorting step should create a more coherent RSH than for reflection rays. In addition our hierarchy should also be more coherent with reflection rays than one based on the RAH algorithm due to the hashing we use. However by the very nature of reflection rays they will never be as coherent as shadow rays resulting in a lower quality hierarchy.

By varying the depth of the hierarchy we also expect that with hierarchies of a certain depth the upper nodes in the hierarchy will have degenerated so much that processing them leads to no benefit at all. In other words, these nodes will be so wide that they will intersect all the geometry in the scene, which means that we just wasted time processing them since we didn't prevent any future intersection tests. This should happen because as we go up in the hierarchy the nodes will encompass so many rays that they will be too large.

By varying the number of nodes used to create the upper levels of the hierarchy we expect a similar outcome to varying the depth. If we use more lower level nodes to create the upper level ones, the upper level nodes will be larger therefore after a certain level they will be too wide (i.e using 16 level 0 nodes to create a level 1 node). However there is a trade-off related to the memory consumption of the algorithm. If we use more nodes to create the upper nodes we will have less nodes overall and thus need less memory to store the potential intersection hits.

## 4.4 Test Results and Discussion

### 4.4.1 Hierarchy Traversal Results - Subdivision 8 Depth 2

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 63,79% less intersections than RAH on this scene. 98,14% less than a brute force approach (see Table 4.1 and Table 4.4).

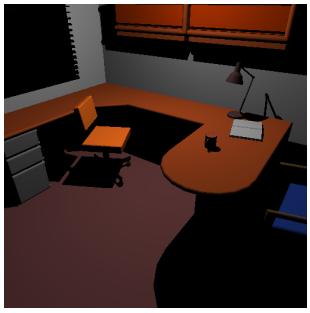


Figure 4.1: OFFICE

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 142726748 | 202025920 |
| # SH MISSES | 117473508 | 186409563 |
| # SH HITS | 25253240 | 15616357 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 11559388 | 85572416 |
| # SH MISSES | 862836 | 76455023 |
| # SH HITS | 10696552 | 9117393 |

TABLE 4.1: OFFICE DIVISION 8 DEPTH 2 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.2 and Table 4.4).

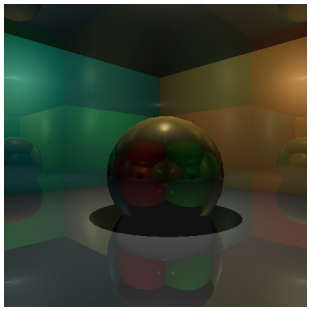


Figure 4.2: CORNELL

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 2995344 | 5167632 |
| # SH MISSES | 2349390 | 3606077 |
| # SH HITS | 645954 | 1561555 |
| # RE INTERSECTIONS | 6488064 | 17802296 |
| # RE MISSES | 4262777 | 14311812 |
| # RE HITS | 2225287 | 3490484 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 750384 | 2375896 |
| # SH MISSES | 453397 | 981144 |
| # SH HITS | 296987 | 1394752 |
| # RE INTERSECTIONS | 2737872 | 10269256 |
| # RE MISSES | 1454215 | 6983410 |
| # RE HITS | 1283657 | 3285846 |

TABLE 4.2: CORNELL DIVISION 8 DEPTH 2 TEST RESULTS.

**Sponza**

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.3 and Table 4.4).

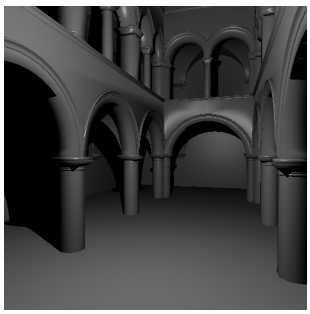
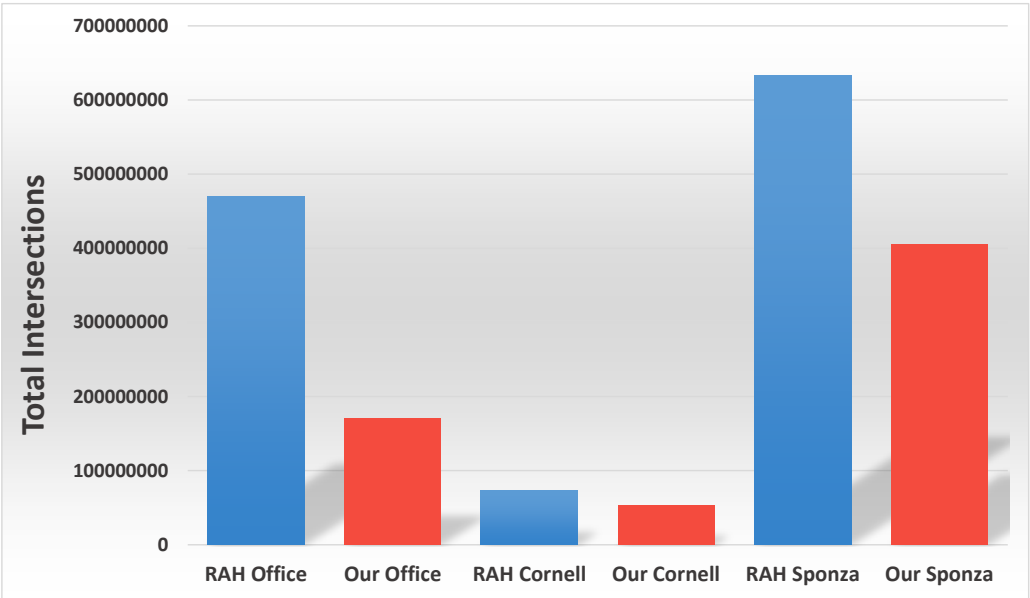


Figure 4.3: SPONZA

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 266597400 | 261494752 |
| # SH MISSES | 233910556 | 248455163 |
| # SH HITS | 32686844 | 13039589 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 266597400 | 62665496 |
| # SH MISSES | 258764213 | 53120871 |
| # SH HITS | 7833187 | 9544625 |

TABLE 4.3: SPONZA DIVISION 8 DEPTH 2 TEST RESULTS.

### 4.4.2 Hierarchy Traversal Discussion - Subdivision 8 Depth 2




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 469683524 | 5.14% | 72869648 | 12.01% | 632408864 | 3.71% |
| *Our Algorithm* | **170070948** | **1.86%** | **53578192** | **8.83%** | **405619896** | **2.38%** |

Table 4.4: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 8 and hierarchy depth 2.

For this set of tests we used a node subdivision of 8 and a hierarchy depth of 2. This means that every node in the upper levels of the hierarchy consists of 8 nodes in the level directly below (or 8 rays if we're constructing the first level of the hierarchy).

#### Office

Our initial expectations for Office were to get a much lower number of intersection tests with our algorithm than with RAH. The scene is a good fit to our bounding volume scheme and our highly coherent shadow ray hierarchy. Results confirm (see Table 4.1) our initial expectations: we compute 63,79% less intersections than RAH on this scene. 98,14% less than a brute force approach.

#### Cornell

For the Cornell scene we focused primarily on the reflection rays which are much more incoherent than shadow rays so we expected results to be less positive than with the Office scene. We compute 26,47% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 91,17% less than the brute force approach (see Table 4.2).

#### Sponza

The final scene, Sponza, is a whole mesh. We did not employ object subdivision in this scene. Hence we expected worse results than with Office since we would only get the benefit of the shadow ray hierarchy and none from the bounding volume scheme. We compute 35,86% less intersection tests than RAH and 97,62% than the brute force approach (see Table 4.3). This confirmed our expectations since the results are about 50% worse than they were with the Office.

However even without the object subdivision our algorithm still manages to outperform RAH, which is very positive since these results all stem from the sorting we applied on the rays before creating the hierarchy.

#### Configuration Comparison

Since we have not analyzed the results for different subdivision and depth configurations we will only compare this setup (as a baseline) with the remaining ones in the next sections.

### 4.4.3 Hierarchy Traversal Results - Subdivision 8 Depth 3

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 39,18% less intersections than RAH on this scene. 97,87% less than a brute force approach (see Table 4.4 and Table 4.8).

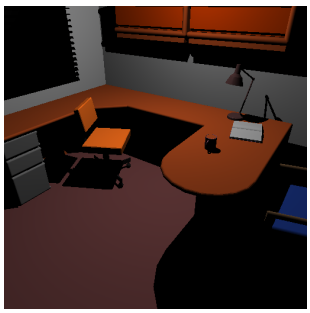


Figure 4.4: OFFICE

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 17863536 | 131228368 | 202025920 |
| # SH MISSES | 1459990 | 105975128 | 186409563 |
| # SH HITS | 16403546 | 25253240 | 15616357 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 4108164 | 31864128 | 85755984 |
| # SH MISSES | 125148 | 21144630 | 76679022 |
| # SH HITS | 3983016 | 10719498 | 9076962 |

TABLE 4.5: OFFICE DIVISION 8 DEPTH 3 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.5 and Table 4.8).

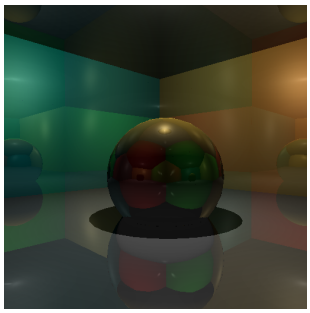


Figure 4.5: CORNELL

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 374616 | 2617976 | 5156880 |
| # SH MISSES | 47369 | 1973366 | 3606077 |
| # SH HITS | 327247 | 644610 | 1550803 |
| | | | |
| # RE INTERSECTIONS | 811008 | 6477632 | 17802296 |
| # RE MISSES | 1304 | 4252345 | 14311812 |
| # RE HITS | 809704 | 2225287 | 3490484 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 181656 | 807208 | 2366280 |
| # SH MISSES | 80755 | 511423 | 991955 |
| # SH HITS | 100901 | 295785 | 1374325 |
| | | | |
| # RE INTERSECTIONS | 614052 | 4143408 | 10278344 |
| # RE MISSES | 96126 | 2858615 | 6992777 |
| # RE HITS | 517926 | 1284793 | 3285567 |

TABLE 4.6: CORNELL DIVISION 8 DEPTH 3 TEST RESULTS.

**Sponza**

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.6 and Table 4.8).

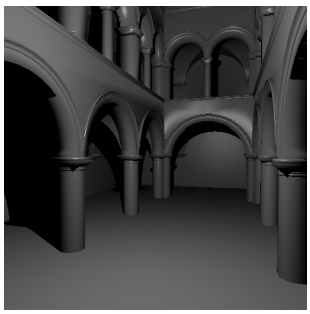
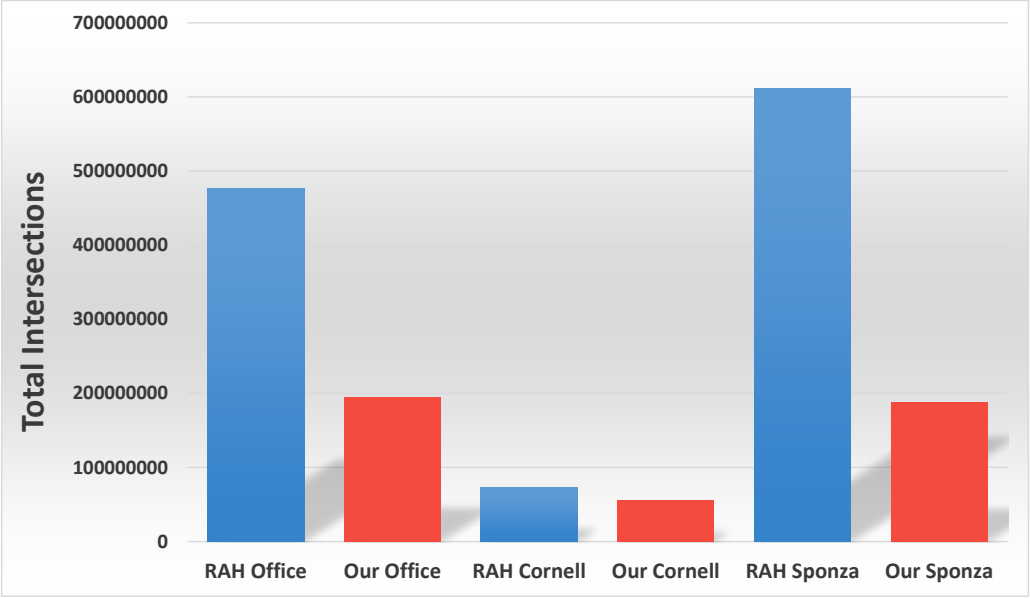


Figure 4.6: SPONZA

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 33357900 | 214961944 | 261132816 |
| # SH MISSES | 6487657 | 182320342 | 248455163 |
| # SH HITS | 26870243 | 32641602 | 12677653 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 33357900 | 23359968 | 61887672 |
| # SH MISSES | 30437904 | 15624009 | 53120871 |
| # SH HITS | 2919996 | 7735959 | 8766801 |

TABLE 4.7: SPONZA DIVISION 8 DEPTH 3 TEST RESULTS.

### 4.4.4 Hierarchy Traversal Discussion - Subdivision 8 Depth 3




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 476048680 | 5.21% | 73570704 | 12.12% | 632408864 | 3.58% |
| *Our Algorithm* | **194343972** | **2.13%** | **55670084** | **9.17%** | **188739948** | **1.11%** |

Table 4.8: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 8 and hierarchy depth 3.

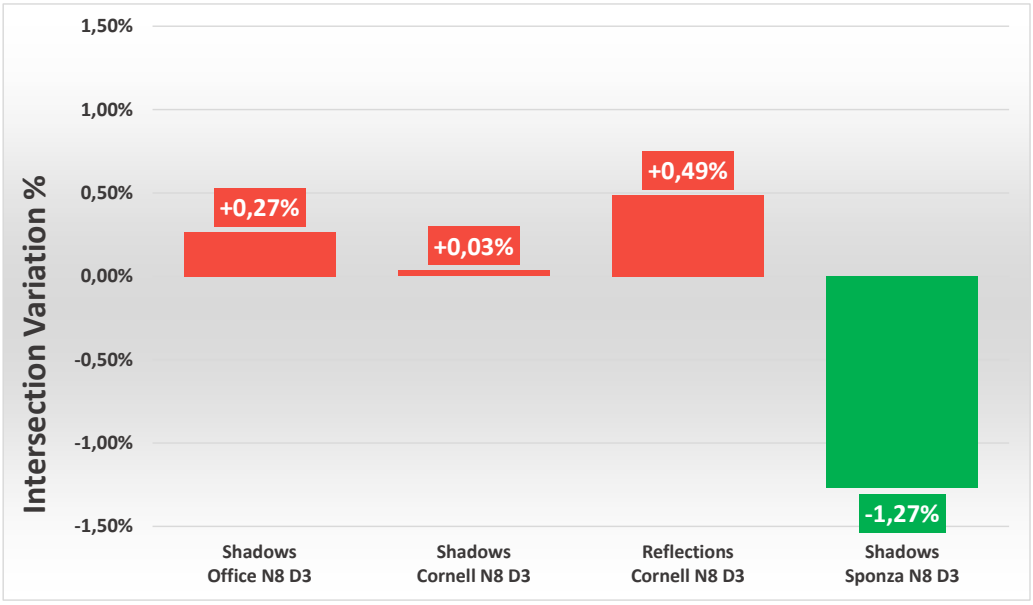



Figure 4.7: Configuration Comparison between Node Subdivision 8 with Hierarchy Depth 2 and Node Subdivision 8 with Hierarchy Depth 3.

For this set of tests we used a node subdivision of 8 and a hierarchy depth of 3. This means that every node in the upper levels of the hierarchy consists of 8 nodes in the level directly below. We compared this configuration with the baseline configuration of node subdivision of 8 and hierarchy depth of 2.

#### Office

Like our previous tests results, this configuration manages to outperform the RAH algorithm once more (see Table 4.4). We compute 59,18% less intersections than RAH on this scene and 97,87% less than a brute force approach. There is however an increase in the number of intersection tests compared to the baseline configuration. The baseline configuration computes 1.86% of the brute force intersection total while this configuration computes 2.13%, a difference of 0.27%, which amounts to about 13% more intersection tests (see Figure 4.7).

#### Cornell

For the Cornell scene we compute 24,33% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 90,83% less than the brute force approach (see Table 4.5). Like in the Office scene there is an increase in the intersection total, from 7.45% to 7.48% for shadow rays and from 9.46% to 9.95% for reflection rays (see Figure 4.7).

#### Sponza

In the Sponza scene we compute 69,10% less intersection tests than RAH and 98,89% than the brute force approach (see Table 4.6). Unlike previous scenes however the number of intersection tests was reduced from 2.38% to 1.11% (see Figure 4.7). This can be explained by the bigger depth of the

hierarchy. Since there aren't any object bounding spheres for this scene every single triangle is tested for intersections with the hierarchy's top nodes. This means that most of these triangles are culled at the first step of the traversal, leading to an overall lower number of intersection tests when compared with a hierarchy with lower depth. There should however be a break point after which the increase in depth will yield no further benefits.

### 4.4.5 Hierarchy Traversal Results - Subdivision 8 Depth 4

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 39,18% less intersections than RAH on this scene. 97,87% less than a brute force approach (see Table 4.8 and Table 4.12).

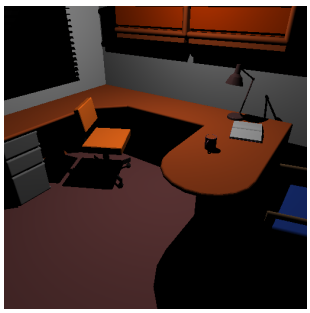


Figure 4.8: OFFICE

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 2251096 | 16736728 | 131228368 | 202025920 |
| # SH MISSES | 159005 | 333182 | 105975128 | 186409563 |
| # SH HITS | 2092091 | 16403546 | 25253240 | 15616357 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 1398238 | 11068016 | 31896768 | 85725152 |
| # SH MISSES | 14736 | 7080920 | 21181124 | 76679166 |
| # SH HITS | 1383502 | 3987096 | 10715644 | 9045986 |

TABLE 4.9: OFFICE DIVISION 8 DEPTH 4 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.9 and Table 4.12).

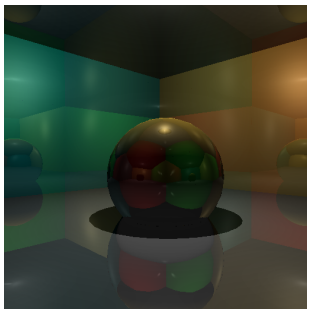


Figure 4.9: CORNELL

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 47520 | 335304 | 2617944 | 5150792 |
| # SH MISSES | 5607 | 8061 | 1974095 | 3600819 |
| # SH HITS | 41913 | 327243 | 643849 | 1549973 |
| | | | | |
| # RE INTERSECTIONS | 101376 | 809824 | 6477632 | 17802296 |
| # RE MISSES | 148 | 120 | 4252345 | 14311812 |
| # RE HITS | 101228 | 809704 | 1284793 | 3490484 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 35280 | 235248 | 807920 | 2358088 |
| # SH MISSES | 5874 | 134258 | 513159 | 991955 |
| # SH HITS | 29406 | 100990 | 294761 | 1366133 |
| | | | | |
| # RE INTERSECTIONS | 101376 | 809808 | 4144632 | 10278344 |
| # RE MISSES | 150 | 291729 | 2859839 | 6992777 |
| # RE HITS | 101226 | 518079 | 1284793 | 3285567 |

TABLE 4.10: CORNELL DIVISION 8 DEPTH 4 TEST RESULTS.

**Sponza**

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.9 and Table 4.12).

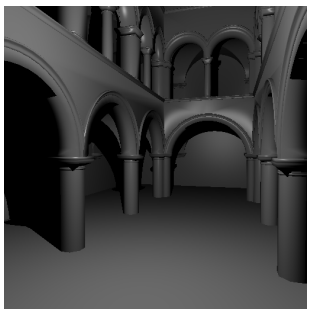
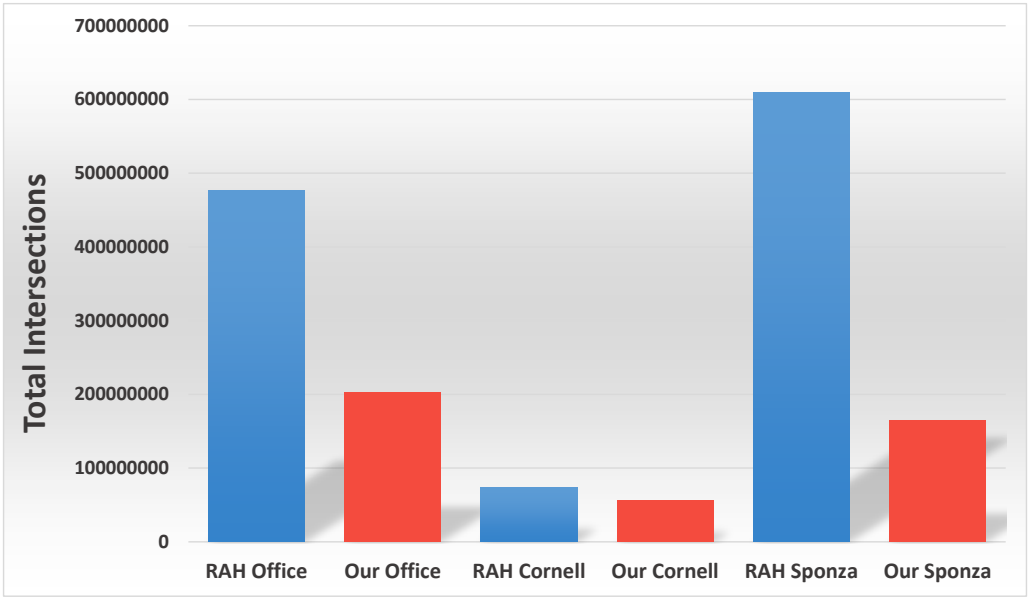


Figure 4.10: SPONZA

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 4186350 | 27458720 | 214961944 | 261132816 |
| # SH MISSES | 754010 | 588477 | 182320342 | 248455163 |
| # SH HITS | 3432340 | 26870243 | 32641602 | 12677653 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 4186350 | 12578168 | 23263056 | 61112376 |
| # SH MISSES | 2614079 | 9670286 | 15624009 | 53120871 |
| # SH HITS | 1572271 | 2907882 | 7639047 | 7991505 |

TABLE 4.11: SPONZA DIVISION 8 DEPTH 4 TEST RESULTS.

### 4.4.6 Hierarchy Traversal Discussion - Subdivision 8 Depth 4




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 477172968 | 5.22% | 73666344 | 12.14% | 609161054 | 3.57% |
| *Our Algorithm* | **202456062** | **2.22%** | **55984296** | **9.22%** | **165071990** | **0.97%** |

Table 4.12: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 8 and hierarchy depth 4.

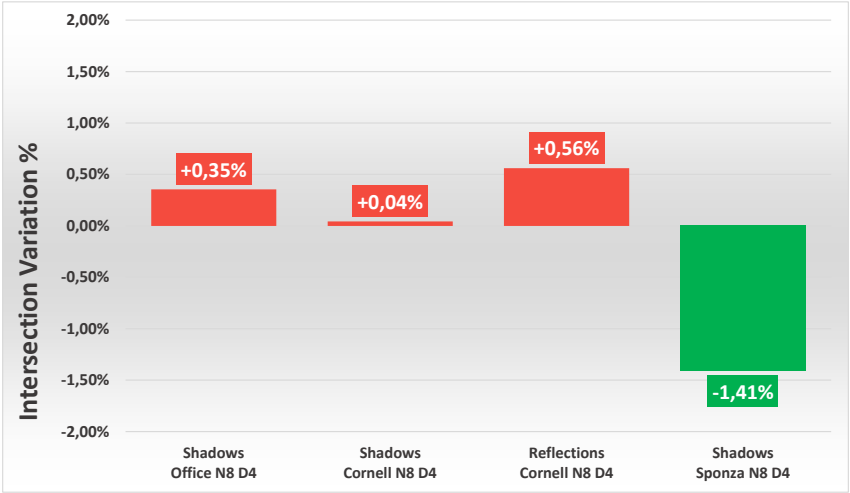



Figure 4.11: Configuration Comparison between Node Subdivision 8 with Hierarchy Depth 2 and Node Subdivision 8 with Hierarchy Depth 4.

For this set of tests we used a node subdivision of 8 and a hierarchy depth of 4. This means that every node in the upper levels of the hierarchy consists of 8 nodes in the level directly below. We compared this configuration with the baseline configuration of node subdivision of 8 and hierarchy depth of 2.

#### Office

Like our previous tests results, this configuration manages to outperform the RAH algorithm once more (see Table 4.8). We compute 57,57% less intersections than RAH on this scene and 97,78% less than a brute force approach. The tendency for an increase in the intersection total shown in the previous comparison continued for this test. There is an increase from 1.86% to 2,22% of the brute force intersection total (see Figure 4.11).

#### Cornell

For the Cornell scene we compute 24,00% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 90,78% less than the brute force approach (see Table 4.9). Like in the Office scene the tendency for an increase in the intersection total remains as we increase the hierarchies depth. The percentage rose from 7.45% to 7.49% for shadow rays and from 9.46% to 10.02% for reflection rays (see Figure 4.11).

#### Sponza

In the Sponza scene we compute 72,90% less intersection tests than RAH and 99,03% than the brute force approach (see Table 4.10). The number of intersection tests was reduced once again, going from 2.38% to 0.97% (see Figure 4.11). Our hypothesis remains consistent since as we increase the hierarchy's depth the number of overall intersections keeps dropping for this particular scene.

### 4.4.7 Hierarchy Traversal Results - Subdivision 16 Depth 2

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 39,18% less intersections than RAH on this scene. 97,87% less than a brute force approach (see Table 4.12 and Table 4.16).

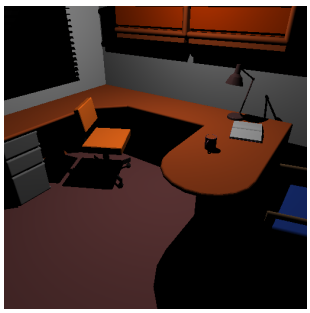


Figure 4.12: OFFICE

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 35690764 | 412453440 |
| # SH MISSES | 9912424 | 398662535 |
| # SH HITS | 25778340 | 13790905 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 7724430 | 119385600 |
| # SH MISSES | 262830 | 112687674 |
| # SH HITS | 7461600 | 6697926 |

TABLE 4.13: OFFICE DIVISION 16 DEPTH 2 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.13 and Table 4.16).

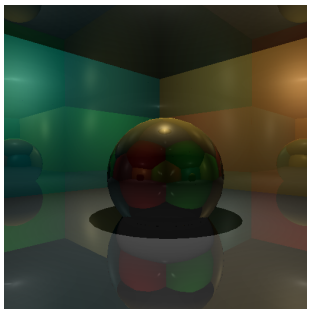


Figure 4.13: CORNELL

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 749232 | 7732000 |
| # SH MISSES | 265982 | 6808346 |
| # SH HITS | 483250 | 923654 |
| | | |
| # RE INTERSECTIONS | 1622016 | 23053680 |
| # RE MISSES | 181161 | 20614507 |
| # RE HITS | 1440855 | 2439173 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 343872 | 2571952 |
| # SH MISSES | 183125 | 1846363 |
| # SH HITS | 160747 | 725589 |
| | | |
| # RE INTERSECTIONS | 983928 | 11460592 |
| # RE MISSES | 267641 | 9368514 |
| # RE HITS | 716287 | 2092078 |

TABLE 4.14: CORNELL DIVISION 16 DEPTH 2 TEST RESULTS.

### Sponza

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.14 and Table 4.16).

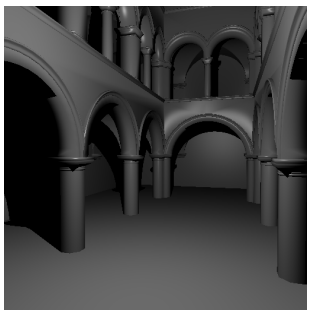
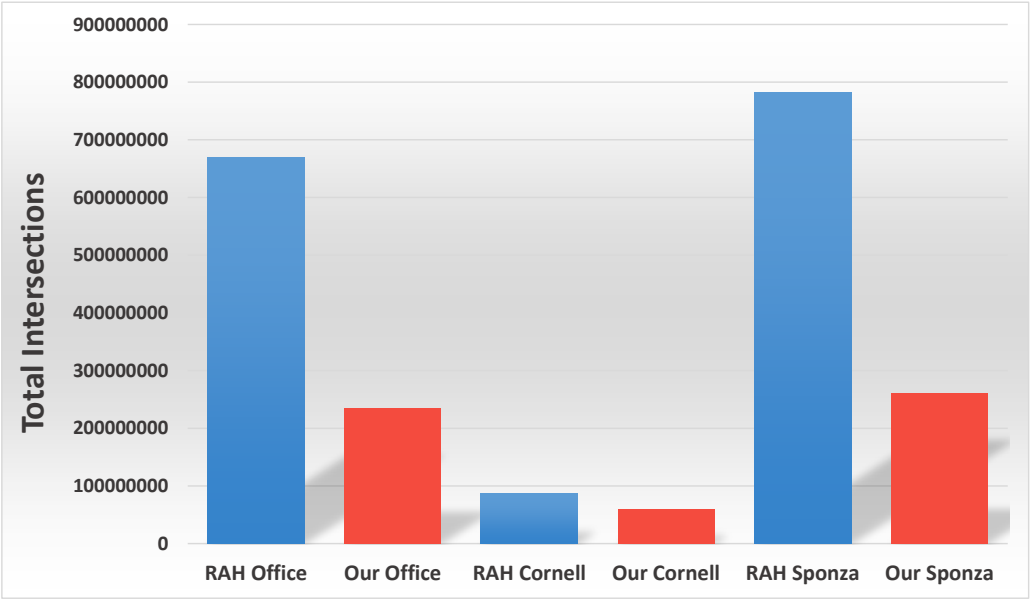


Figure 4.14: SPONZA

| OFFICE | LEVEL 2 | LEVEL 1 |
|---|---|---|
| *RAH Algorithm* | | |
| # SH INTERSECTIONS | 66649350 | 554494192 |
| # SH MISSES | 31993463 | 544455870 |
| # SH HITS | 34655887 | 10038322 |
| *Our Algorithm* | | |
| # SH INTERSECTIONS | 66649350 | 103145120 |
| # SH MISSES | 60202780 | 97483134 |
| # SH HITS | 6446570 | 5661986 |

TABLE 4.15: SPONZA DIVISION 16 DEPTH 2 TEST RESULTS.

## 4.4.8 Hierarchy Traversal Discussion - Subdivision 16 Depth 2




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 668798684 | 7.32% | 86962160 | 14.33% | 781756694 | 4.58% |
| *Our Algorithm* | **234276846** | **2.57%** | **60443016** | **9.96%** | **260386246** | **1.53%** |

Table 4.16: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 16 and hierarchy depth 2.

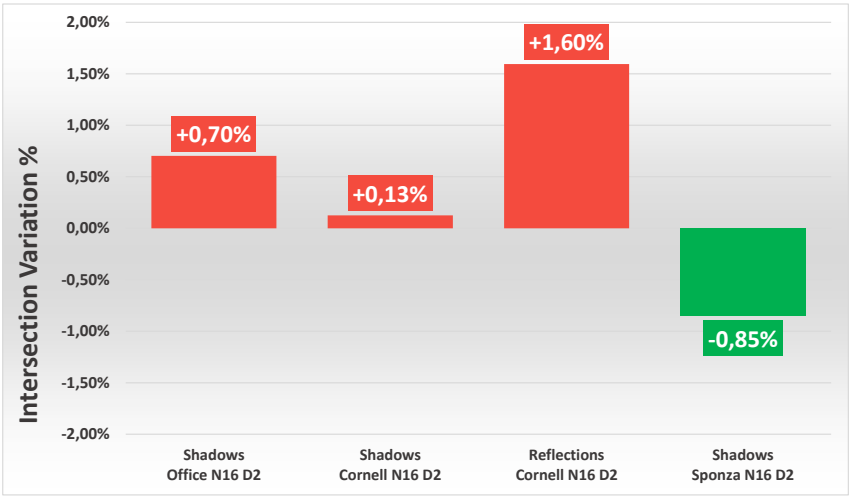



Figure 4.15: Configuration Comparison between Node Subdivision 8 with Hierarchy Depth 2 and Node Subdivision 16 with Hierarchy Depth 2.

For this set of tests we used a node subdivision of 16 and a hierarchy depth of 2. This means that every node in the upper levels of the hierarchy consists of 16 nodes in the level directly below. We compared this configuration with the baseline configuration of node subdivision of 8 and hierarchy depth of 2.

### Office

With the new node subdivision this configuration still outperforms the RAH algorithm (see Table 4.12). We compute 64,97% less intersections than RAH on this scene and 97,43% less than a brute force approach. These results are extremely close to the baseline configuration but there is still an increase of 0.70% when compared to the baseline (from 1.86% to 2.57%) (see Figure 4.15).

### Cornell

For the Cornell scene we compute 30,50% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 90,04% less than the brute force approach (see Table 4.13). When we compare these results with the baseline configuration we get an increase of 0.13%, from 7.45% to 7.58% for shadow rays and from 9.46% to 11.06% for reflection rays (see Figure 4.15).

### Sponza

In the Sponza scene we compute 66,69% less intersection tests than RAH and 98,47% than the brute force approach (see Table 4.14). The number of intersection tests was reduced once again, going from the baseline 2.38% to 1.53% (see Figure 4.15). This is slightly different from the previous cases but the explanation is very similar. Because every node consists of more rays the upper levels of the hierarchy discard even more test results for the traversal of the lower levels of the hierarchy, leading to the overall reduction in intersections.

### 4.4.9 Hierarchy Traversal Results - Subdivision 16 Depth 3

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 39,18% less intersections than RAH on this scene. 97,87% less than a brute force approach (see Table 4.16 and Table 4.20).

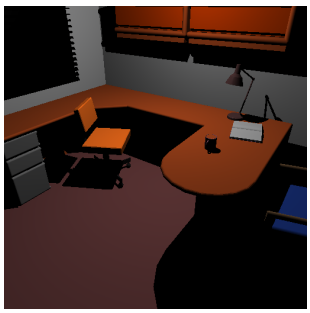


Figure 4.16: OFFICE

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 2251096 | 36017536 | 412453440 |
| # SH MISSES | 0 | 10239196 | 398662535 |
| # SH HITS | 2251096 | 25778340 | 13790905 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 1474802 | 23344000 | 119524928 |
| # SH MISSES | 15802 | 15873692 | 112848037 |
| # SH HITS | 1459000 | 7470308 | 6676891 |

TABLE 4.17: OFFICE DIVISION 16 DEPTH 3 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.17 and Table 4.20).

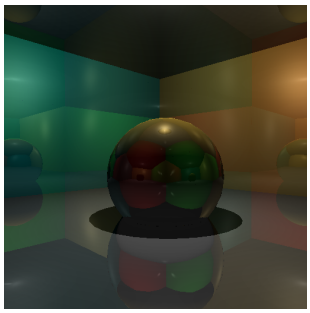


Figure 4.17: CORNELL

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 47520 | 760128 | 7732000 |
| # SH MISSES | 12 | 276878 | 6808346 |
| # SH HITS | 47508 | 483250 | 923654 |
| | | | |
| # RE INTERSECTIONS | 101376 | 1622016 | 23053680 |
| # RE MISSES | 0 | 181161 | 20614507 |
| # RE HITS | 101376 | 1440855 | 2439173 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 36720 | 498352 | 2577680 |
| # SH MISSES | 5573 | 337247 | 1856187 |
| # SH HITS | 31147 | 161105 | 721493 |
| | | | |
| # RE INTERSECTIONS | 101376 | 1620432 | 11466336 |
| # RE MISSES | 99 | 903786 | 9374256 |
| # RE HITS | 101277 | 716646 | 2092080 |

TABLE 4.18: CORNELL DIVISION 16 DEPTH 3 TEST RESULTS.

### Sponza

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.18 and Table 4.20).

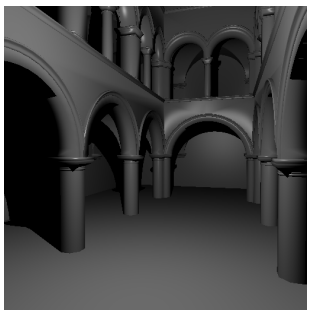
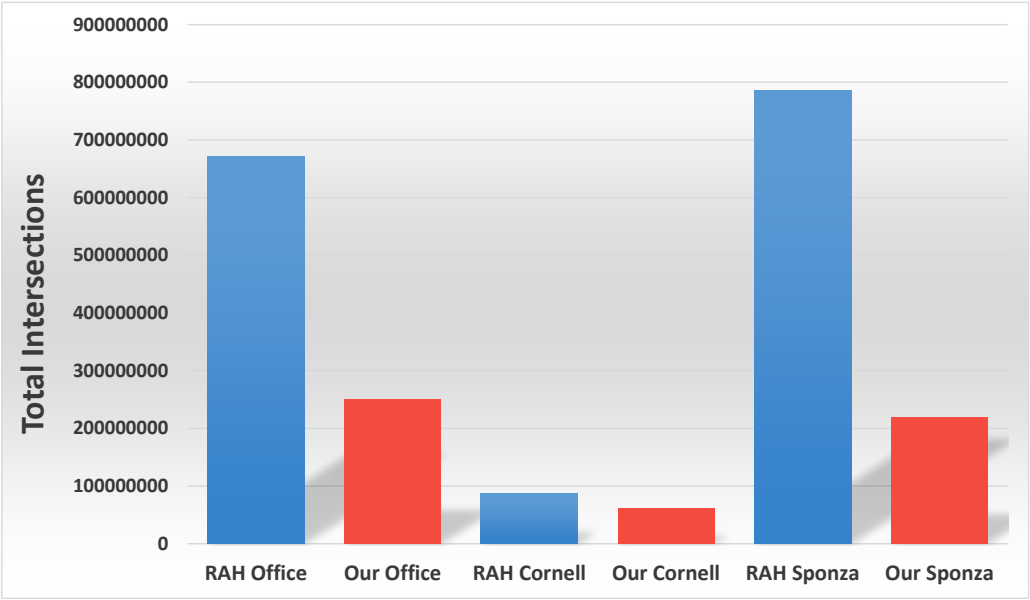


Figure 4.18: SPONZA

| OFFICE | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|
| *RAH Algorithm* | | | |
| # SH INTERSECTIONS | 4186350 | 66981600 | 554494192 |
| # SH MISSES | 0 | 32325713 | 544455870 |
| # SH HITS | 4186350 | 34655887 | 10038322 |
| *Our Algorithm* | | | |
| # SH INTERSECTIONS | 4186350 | 29037376 | 102667568 |
| # SH MISSES | 2371514 | 22620653 | 97483134 |
| # SH HITS | 1814836 | 6416723 | 5184434 |

TABLE 4.19: SPONZA DIVISION 16 DEPTH 3 TEST RESULTS.

## 4.4.10 Hierarchy Traversal Discussion - Subdivision 16 Depth 3




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 671376552 | 7.35% | 87121952 | 14.35% | 786275294 | 4.61% |
| *Our Algorithm* | **251173986** | **2.75%** | **61318064** | **10.10%** | **218842238** | **1.28%** |

Table 4.20: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 16 and hierarchy depth 3.

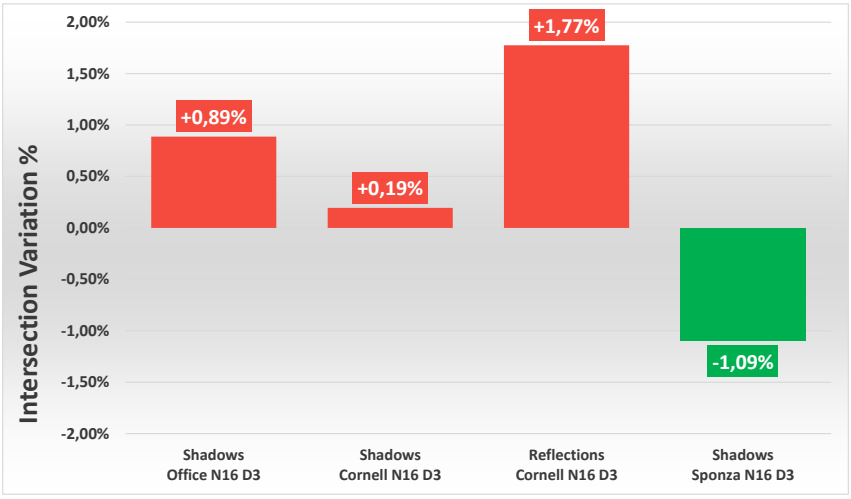



Figure 4.19: Configuration Comparison between Node Subdivision 8 with Hierarchy Depth 2 and Node Subdivision 16 with Hierarchy Depth 3.

For this set of tests we used a node subdivision of 16 and a hierarchy depth of 3. This means that every node in the upper levels of the hierarchy consists of 16 nodes in the level directly below. We compared this configuration with the baseline configuration of node subdivision of 8 and hierarchy depth of 2.

#### Office

Once more, increasing the depth of the hierarchy outperforms the RAH algorithm (see Table 4.16). We compute 62.59% less intersections than RAH on this scene and 97,25% less than a brute force approach. There is an increase of 0.70% when compared to the baseline (from 1.86% to 2.75%) (see Figure 4.19).

#### Cornell

For the Cornell scene we compute 29,62% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 89,90% less than the brute force approach (see Table 4.17). When we compare these results with the baseline configuration we get an increase of 0.13%, from 7.45% to 7.65% for shadow rays and from 9.46% to 11.24% for reflection rays (see Figure 4.19). This continues the trend seen in previous test results.

#### Sponza

In the Sponza scene we compute 72,17% less intersection tests than RAH and 98,72% than the brute force approach (see Table 4.18). The number of intersection tests was reduced once again, going from the baseline 2.38% to 1.28% (see Figure 4.19). Since we increased the depth from the previous configuration it is natural that the trend for a decrease in intersection tests remains until we reach the break point.

### 4.4.11 Hierarchy Traversal Results - Subdivision 16 Depth 4

#### Office

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 39,18% less intersections than RAH on this scene. 97,87% less than a brute force approach (see Table 4.20 and Table 4.24).

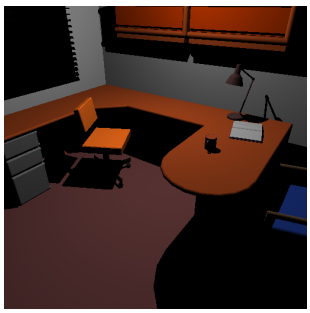


Figure 4.20: OFFICE

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 145232 | 2323712 | 36017536 | 412453440 |
| # SH MISSES | 0 | 72616 | 10239196 | 398662535 |
| # SH HITS | 145232 | 2251096 | 25778340 | 13790905 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 145232 | 2323712 | 23351216 | 119524928 |
| # SH MISSES | 0 | 864261 | 15880908 | 112848037 |
| # SH HITS | 145232 | 1459451 | 7470308 | 6676891 |

TABLE 4.21: OFFICE DIVISION 16 DEPTH 4 TEST RESULTS.

#### Cornell

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 26,47% less intersections than RAH on this scene. 91,17% less than a brute force approach (see Table 4.21 and Table 4.24).

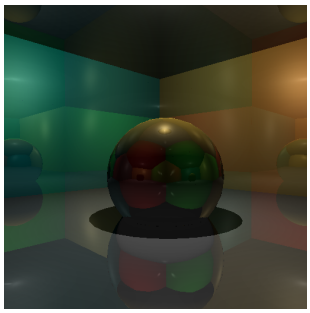


Figure 4.21: CORNELL

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 3168 | 50688 | 760128 | 7732000 |
| # SH MISSES | 0 | 3180 | 276878 | 6808346 |
| # SH HITS | 3168 | 47508 | 483250 | 923654 |
| | | | | |
| # RE INTERSECTIONS | 6336 | 101376 | 1622016 | 23053680 |
| # RE MISSES | 0 | 0 | 181161 | 20614507 |
| # RE HITS | 6336 | 101376 | 1440855 | 2439173 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 3168 | 50688 | 498352 | 2577680 |
| # SH MISSES | 0 | 19541 | 337247 | 1856187 |
| # SH HITS | 3168 | 31147 | 161105 | 721493 |
| | | | | |
| # RE INTERSECTIONS | 6336 | 101376 | 1620432 | 11466336 |
| # RE MISSES | 0 | 99 | 903786 | 9374256 |
| # RE HITS | 6336 | 101277 | 716646 | 2092080 |

TABLE 4.22: CORNELL DIVISION 16 DEPTH 4 TEST RESULTS.

### Sponza

Comparing our algorithm with the RAH algorithm we can see that our algorithm computes 35,86% less intersections than RAH on this scene. 97,62% less than a brute force approach (see Table 4.22 and Table 4.24).

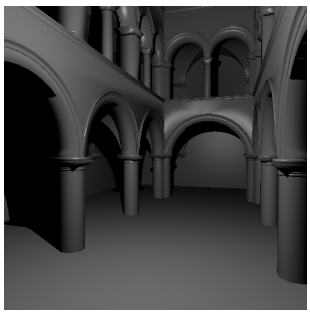
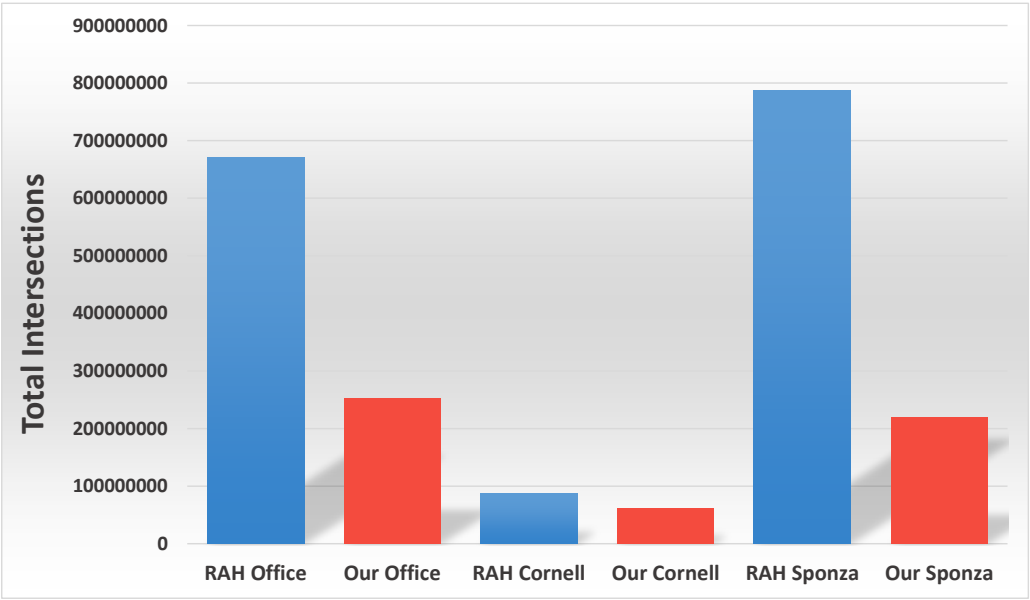


Figure 4.22: SPONZA

| OFFICE | LEVEL 4 | LEVEL 3 | LEVEL 2 | LEVEL 1 |
|---|---|---|---|---|
| *RAH Algorithm* | | | | |
| # SH INTERSECTIONS | 265800 | 4252800 | 66981600 | 554494192 |
| # SH MISSES | 0 | 66450 | 32325713 | 544455870 |
| # SH HITS | 265800 | 4186350 | 34655887 | 10038322 |
| *Our Algorithm* | | | | |
| # SH INTERSECTIONS | 265800 | 4252800 | 29037376 | 102667568 |
| # SH MISSES | 0 | 2437964 | 22620653 | 97483134 |
| # SH HITS | 265800 | 1814836 | 6416723 | 5184434 |

TABLE 4.23: SPONZA DIVISION 16 DEPTH 4 TEST RESULTS.

## 4.4.12 Hierarchy Traversal Discussion - Subdivision 16 Depth 4




| | OFFICE | | CORNELL | | SPONZA | |
|---|---|---|---|---|---|---|
| ALGORITHM | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % | TOTAL # ISECT | RELATIVE % |
| *Brute Force* | 9133132168 | 100% | 606911976 | 100% | 17058578850 | 100% |
| *RAH Algorithm* | 671594400 | 7.35% | 87134624 | 14.36% | 786607544 | 4.61% |
| *Our Algorithm* | **252175344** | **2.76%** | **61341536** | **10.11%** | **219174488** | **1.28%** |

Table 4.24: OFFICE (251546 shadow rays), CORNELL (242015 shadow & 524288 reflection rays), SPONZA (256713 shadow rays) rendering performance using node subdivision 16 and hierarchy depth 4.

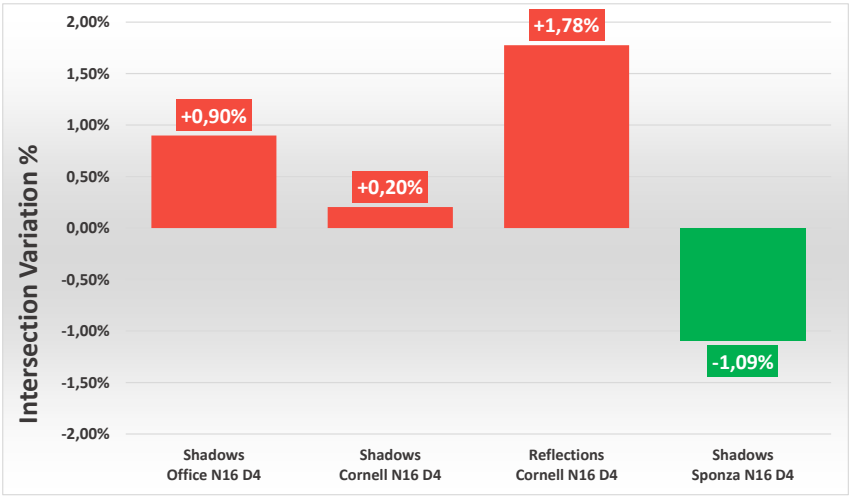



Figure 4.23: Configuration Comparison between Node Subdivision 8 with Hierarchy Depth 2 and Node Subdivision 16 with Hierarchy Depth 4.

For this set of tests we used a node subdivision of 16 and a hierarchy depth of 3. This means that every node in the upper levels of the hierarchy consists of 16 nodes in the level directly below. We compared this configuration with the baseline configuration of node subdivision of 8 and hierarchy depth of 2.

#### Office

For the final configuration the Office scene calculated less intersections the RAH algorithm (see Table 4.20). We compute 62.45% less intersections than RAH on this scene and 97,24% less than a brute force approach. There is yet another increase of 0.70% when compared to the baseline (from 1.86% to 2.75%) (see Figure 4.19).

#### Cornell

For the final test for the Cornell scene we compute 29,60% less intersections overall (shadow and reflection rays combined) than the RAH algorithm and 89,89% less than the brute force approach (see Table 4.21). When we compare these results with the baseline configuration we get an increase of 0.13%, from 7.45% to 7.66% for shadow rays and from 9.46% to 11.25% for reflection rays (see Figure 4.23).

#### Sponza

For the final test using the Sponza scene we computed 72,14% less intersection tests than RAH and 98,68% than the brute force approach (see Table 4.22). The number of intersection tests was reduced once again, going from the baseline 2.38% to 1.28% (see Figure 4.23). We increased the depth from the previous configuration but we got a small increase in the intersection tests. This indicates that the

previous configuration is the break point after which increasing the depth stops yielding benefits for this scene.

### 4.4.13 Rendering Time Results

For this set of tests we used a node subdivision of 8 and a hierarchy depth of 2. This means that every node in the upper levels of the hierarchy consists of 16 nodes in the level directly below. The tests were run over the course of 58 frames and the results for each phase are the average of these 58 frames.

**Office**

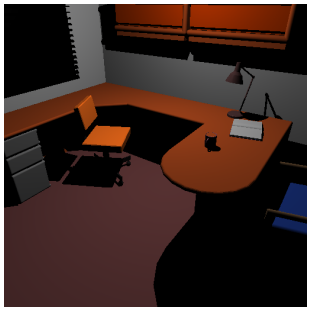


Figure 4.24: OFFICE

| OFFICE | TOTAL TIME (MS) | RELATIVE TIME (%) |
|---|---|---|
| *RAH Algorithm (618 ms per Frame)* | | |
| RAY CREATION | 40,240 | 0,11% |
| RAY COMPRESSION | 0,000 | 0,00% |
| RAY SORTING | 0,000 | 0,00% |
| RAY DECOMPRESSION | 0,000 | 0,00% |
| HIERARCHY CREATION | 122,165 | 0,34% |
| HIERARCHY TRAVERSAL | 30394,169 | 84,77% |
| FINAL INTERSECTION TESTS | 4428,185 | 12,35% |
| *Our Algorithm (316 ms per Frame)* | | |
| RAY CREATION | 40,433 | 0,22% |
| RAY COMPRESSION | 16,651 | 0,09% |
| RAY SORTING | 11,822 | 0,06% |
| RAY DECOMPRESSION | 99,721 | 0,54% |
| HIERARCHY CREATION | 130,626 | 0,71% |
| HIERARCHY TRAVERSAL | 13918,797 | 75,76% |
| FINAL INTERSECTION TESTS | 3355,308 | 18,26% |

TABLE 4.25: OFFICE RENDERING TIME RESULTS.

For this scene we can see that the major time consuming steps are in fact the hierarchy traversal and the final intersection tests. The biggest difference between our algorithm and the RAH algorithm resides in the time spent traversing the hierarchy. While our algorithm only needs 13918 milliseconds, the RAH algorithm requires 30394 milliseconds, which is about 215% more. This increased time traversing the hierarchy amounts to about 100% more time to render each frame (see Table 4.24).

**Cornell**

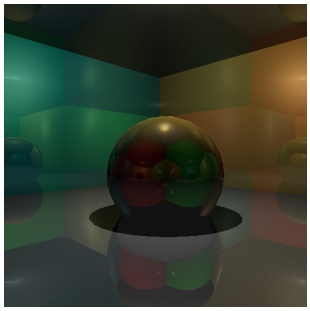


Figure 4.25: CORNELL

| CORNELL | TOTAL TIME (MS) | RELATIVE TIME (%) |
|---|---|---|
| *RAH Algorithm (093 ms per Frame)* | | |
| RAY CREATION | 125,731 | 2,16% |
| RAY COMPRESSION | 0,000 | 0,00% |
| RAY SORTING | 0,000 | 0,00% |
| RAY DECOMPRESSION | 0,000 | 0,00% |
| HIERARCHY CREATION | 366,984 | 6,29% |
| HIERARCHY TRAVERSAL | 2717,309 | 46,59% |
| FINAL INTERSECTION TESTS | 2208,936 | 37,88% |
| *Our Algorithm (100 ms per Frame)* | | |
| RAY CREATION | 128,134 | 2,38% |
| RAY COMPRESSION | 53,916 | 1,00% |
| RAY SORTING | 50,970 | 0,95% |
| RAY DECOMPRESSION | 251,946 | 4,68% |
| HIERARCHY CREATION | 407,090 | 7,57% |
| HIERARCHY TRAVERSAL | 1776,750 | 33,03% |
| FINAL INTERSECTION TESTS | 2175,057 | 40,43% |

TABLE 4.26: CORNELL RENDERING TIME RESULTS.

Much like Office, the Cornell scene takes most of its rendering time traversing the hierarchy and calculating the final intersection tests. However due to the lower geometric complexity of the scene the absolute values aren't as high. The traversal takes 2717 milliseconds for the RAH algorithm and 1700 for our algorithm, which is a significant reduction (see Table 4.25).

**Sponza**

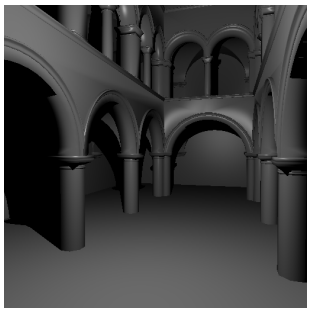


Figure 4.26: SPONZA

| SPONZA | TOTAL TIME (MS) | RELATIVE TIME (%) |
|---|---|---|
| *RAH Algorithm (957 ms per Frame)* | | |
| RAY CREATION | 41,215 | 0,07% |
| RAY COMPRESSION | 0,000 | 0,00% |
| RAY SORTING | 0,000 | 0,00% |
| RAY DECOMPRESSION | 0,000 | 0,00% |
| HIERARCHY CREATION | 126,499 | 0,23% |
| HIERARCHY TRAVERSAL | 49537,627 | 89,25% |
| FINAL INTERSECTION TESTS | 4995,089 | 9,00% |
| *Our Algorithm (901 ms per Frame)* | | |
| RAY CREATION | 41,500 | 0,08% |
| RAY COMPRESSION | 17,151 | 0,03% |
| RAY SORTING | 12,696 | 0,02% |
| RAY DECOMPRESSION | 98,617 | 0,19% |
| HIERARCHY CREATION | 137,835 | 0,26% |
| HIERARCHY TRAVERSAL | 46867,912 | 89,65% |
| FINAL INTERSECTION TESTS | 4367,193 | 8,35% |

TABLE 4.27: SPONZA RENDERING TIME RESULTS.

Finally for the Sponza scene we still see a similar relative time being spent in the traversal of the hierarchy as in the previous scenes. Even though the Sponza scene isn't subdivided into separate object meshes, we still manage to outperform RAH while traversing the hierarchy (see Table 4.26).

### 4.4.14 Test Results Global Discussion

#### Configuration Tests

For the Office scene the configuration tests indicated that the optimal configuration is in fact the baseline configuration used, using node subdivision of 8 and hierarchy depth of 2. However this is reliant on the hashing function used, which means that if a better hashing function is used the optimal configuration may be different.

For the Cornell scene there was always an increase for every configuration when compared with the baseline configuration. However this increase was more significant for the reflection rays. This is due to the fact that reflection rays are much less coherent than shadow rays, which leads to bigger nodes when combining such rays. Much like the shadow rays however using better hashing functions can mitigate this effect.

Finally for the Sponza scene we noticed a reverse trend compared to the remaining scenes. This is due to the fact that the scene itself isn't subdivided into several objects, making the use of bounding spheres useless. As such, only the higher levels of the hierarchy were culling the number of potential intersection tests, which means that as we increased the hierarchy's depth this mitigation effect would be increased until we reached a certain break point, like it was noted before. Since this scene is an outlier however it is safe to assume if it was subdivided into several objects, much like the Office scene, the optimal configuration would be the baseline configuration used for the tests.

#### Rendering Time Tests

Regarding the rendering time tests, it is notable that even though we managed to significantly reduce the total number of intersection tests the hierarchy traversal still takes up most of the rendering time. This leads us to believe that if we can optimize the traversal step, either by culling more intersection tests at the early stages or by making the implementation more efficient we can get a significant reduction in the time necessary for the two final phases of our algorithm.

# Chapter 5

# Conclusions

In this thesis we introduced a new algorithm that creates an efficient Ray-Space Hierarchy which greatly reduces the number of intersection tests necessary to ray-trace a scene due to its improved coherency and shallow bounding volume hierarchy.

We achieved our goal of reducing the number of intersection tests using a Ray-Space Hierarchy. This technique is orthogonal to the use of both Object and Space Hierarchies. They can all be be used together in the same application to obtain even better results. Our results show that we can expect a reduction in computed intersections of 50% for shadow rays and 25% for reflection rays compared to previous state of the art ray-space hierarchies.

## 5.1 Future Work

However there is still room for improvement, namely in three areas: The first area of improvement concerns the ray hashing functions. Since the hash determines how rays are sorted, the hierarchy will improve if we manage to enhance the ray classification accuracy (i.e. ray spatial coherency).

The second area of improvement relates to the object bounding-volumes. We used spherical bounding volumes in this paper and a shallow object hierarchy. In the future we aim to also combine our coherent ray hierarchy with a deeper object hierarchy that will further reduce the number of ray-primitive intersections (e.g. [19]).

The final area of improvement relates to the memory necessary to run the algorith. The arrays that are necessary to store the hierarchy traversal hits are very large therefore any reductions in the size of these arrays would allow for more triangles to be processed per batch.

# Bibliography


[1] D. Roger, U. Assarsson, and N. Holzschuch. Whitted Ray-tracing for Dynamic Scenes Using a Ray-space Hierarchy on the GPU. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR '07, pages 99–110, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-52-4.

[2] K. Garanzha and C. Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2), May 2010.

[3] T. Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, Jun 1980. ISSN 0001-0782.

[4] T. Ritschel, C. Dachsbacher, T. Grosch, and J. Kautz. The State of the Art in Interactive Global Illumination. In *Computer Graphics Forum*, volume 31, pages 160–188. Wiley Online Library, 2012.

[5] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[6] J. Arvo and D. Kirk. Fast Ray Tracing by Ray Classification. *SIGGRAPH Computer Graphics*, 21 (4):55–64, Aug 1987. ISSN 0097-8930.

[7] T. Aila and T. Karras. Architecture Considerations for Tracing Incoherent Rays. In *Proceedings of the Conference on High Performance Graphics*, pages 113–122. Eurographics Association, 2010.

[8] G. Simiakakis and A. M. Day. Five-dimensional adaptive subdivision for ray tracing. *Computer Graphics Forum*, 13(2):133–140, 1994. ISSN 1467-8659.

[9] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. GPU Gems. Addison-Wesley Professional, 2005. ISBN 0321335597.

[10] B. Gärtner. Fast and Robust Smallest Enclosing Balls. In *Proceedings of the 7th Annual European Symposium on Algorithms*, ESA '99, pages 325–338, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66251-0.

[11] A. W. Paeth, editor. *Graphics Gems V*. Academic Press, Inc., 1995. ISBN 012543457X.

[12] D. Merrill and A. Grimshaw. Parallel Scan for Stream Architectures. Technical report, University of Virginia, Department of Computer Science, 2009.

[13] D. G. Merrill and A. S. Grimshaw. Revisiting Sorting for GPGPU Stream Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7.

[14] L. Szécsi. The Hierarchical Ray Engine. In *WSCG Full Papers Proceedings*, pages 249–256. Václav Skala-UNION Agency, 2006.

[15] C. Ericson. *Real-Time Collision Detection*. Series in Interactive 3-D Technology. Morgan Kaufmann Publishers Inc., 2004. ISBN 1558607323.

[16] J. Amanatides. Ray Tracing with Cones. *SIGGRAPH Computer Graphics*, 18(3):129–135, Jan 1984. ISSN 0097-8930.

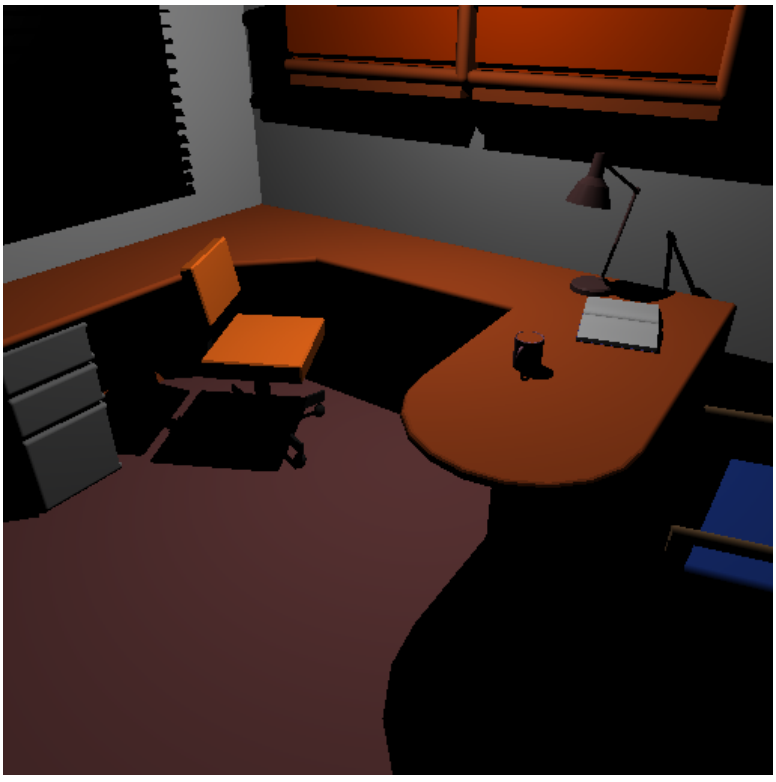
[17] T. Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphic Tools*, 2(2):25–30, Nov 1997. ISSN 1086-7651.

[18] G. E. Blelloch. Prefix Sums and their Applications. Technical report, 1990.

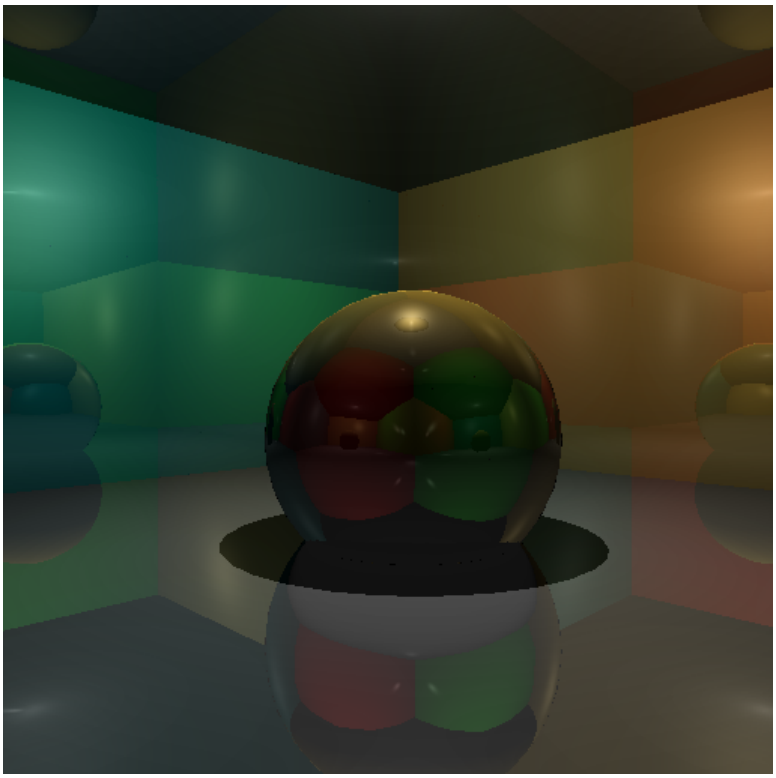
[19] G. Bradshaw and C. O'Sullivan. Adaptive Medial-Axis Approximation for Sphere-tree Construction. *ACM Transactions on Graphics (TOG)*, 23(1):1–26, 2004.

# Appendix A

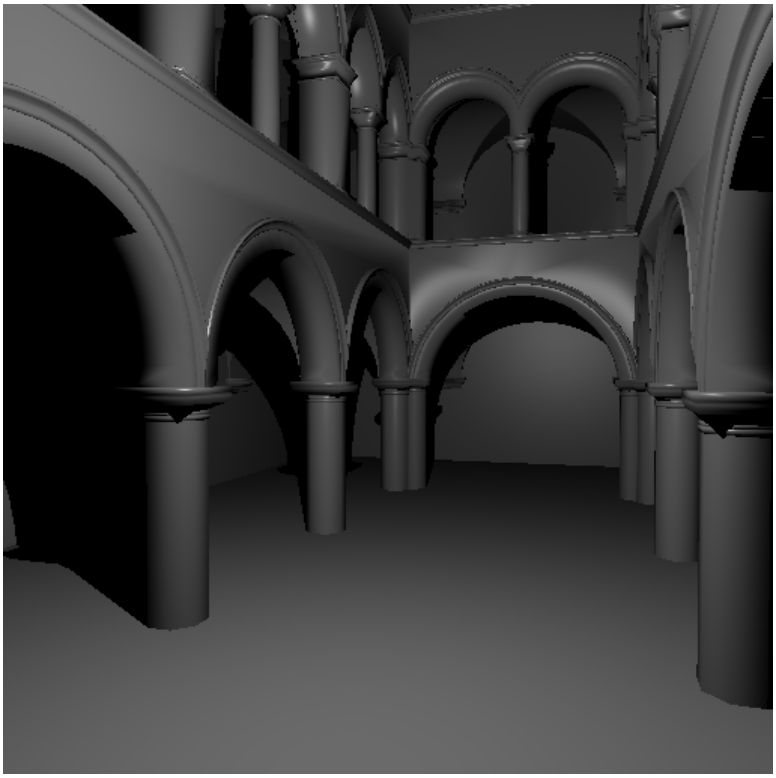
# Rendered Images



OFFICE



CORNELL



SPONZA