Ray Tracing: Accelerated by Bounding Volume Hierarchies

Roshan Pasupathy, Ben Haderle, Zack Houghton, Wei-Han Chang

Abstract

Ray tracing is a highly effective and popular approach for rendering scenes with accurate global illumination. However, these results come at the cost of significantly longer rendering times. To address this issue we implemented bounding volume hierarchies (BVH) for faster intersection testing. We implemented top down (sorting based) and bottom up (clustering based) algorithms for constructing Hierarchies of higher quality and investigated their effectiveness. Our implementation of agglomerative clustering improved performance by a factor of 9 when compared to non-accelerated ray tracing. Detailed results are provided in Section 4.

1. Introduction

Ray tracing is a rendering technique that is able to render shadows and calculate global illumination for surfaces to render realistic images. It achieves this by shooting a ray from each pixel and following the scattered ray to light sources and colored objects. The downside of ray tracing is that it has a much longer render time than traditional methods. We first built a raytracer which ran on the CPU. To speed up render times we ported our CPU Ray tracer to CUDA for GPU execution. With this we were able to render realistic images of scenes with limited number of objects within seconds on the GPU. However, we wanted to render several objects, like the scene shown in Figure 1, in shorter time frames on the CPU. Thus the goal of our project was to algorithmically achieve faster render times. The following sections cover the background material and the algorithms we implemented to achieve accelerated ray tracing.

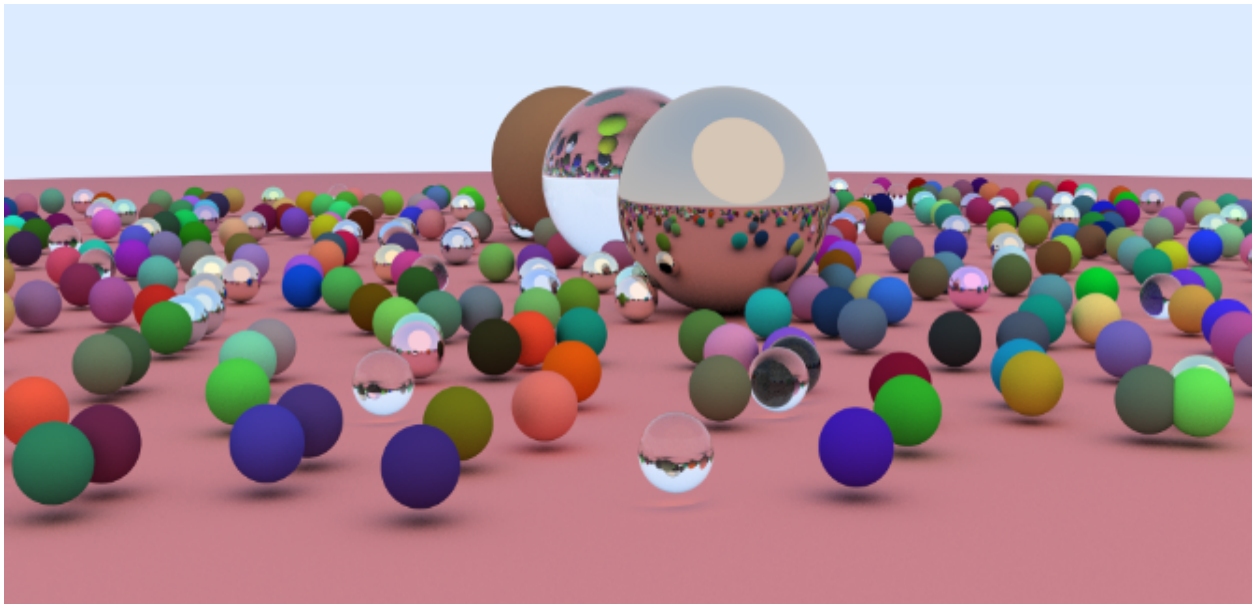


Figure 1: Scene with 499 spheres of various material types rendered by our CPU Ray Tracer

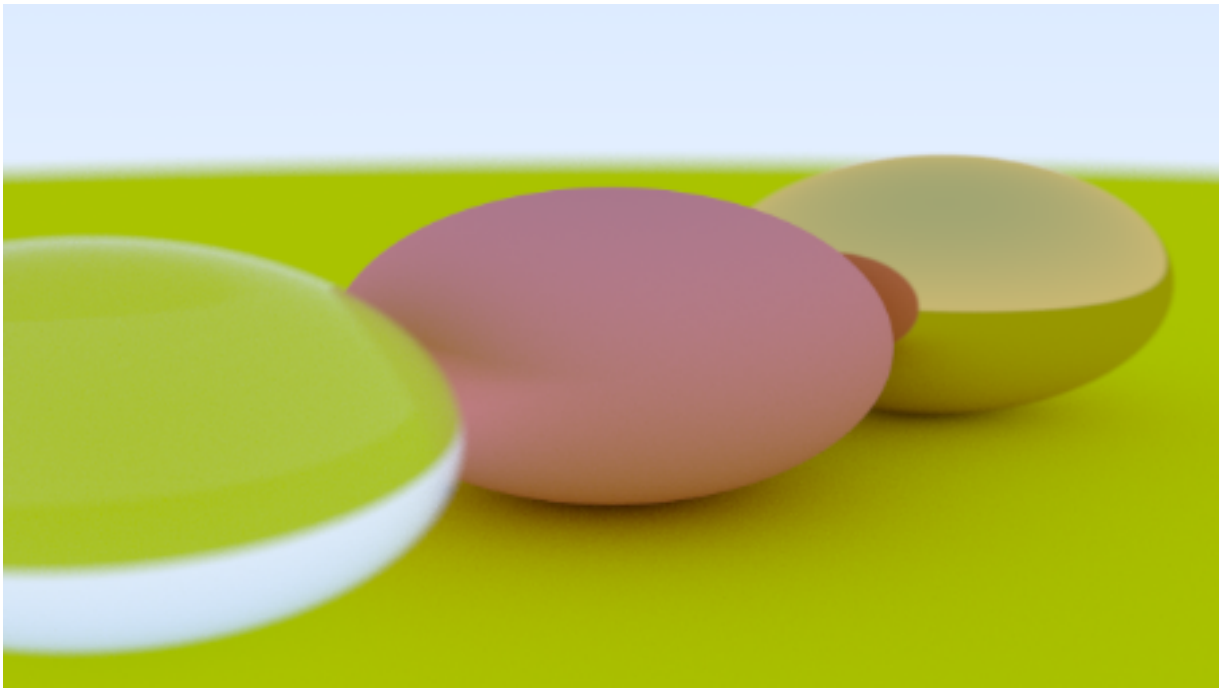


Figure 2: A Dielectric, Metal and Lambertian ball rendered by our GPU Ray Tracer

2. Background

The most expensive computation in the ray tracing pipeline tends to be linearly iterating through the objects in the world and performing ray intersection tests. However, logarithmic time complexity is achievable using Tree or Grid data structures to eliminate multiple checks at once. Two types of algorithms are generally used for accelerating ray intersection - spatial subdivision and bounding volume hierarchies.

2.1 Bounding Volume Hierarchy (BVH)

A bounding volume hierarchy utilizes axis aligned bounding boxes to group primitives and arrange them hierarchically. If a ray does not intersect a bounding box, we can conclude that it does not hit any of the objects contained within that bounding box. The major advantage here is that ray intersection testing with bounding boxes is computationally inexpensive.

2.2 Spatial Subdivision Methods

Grids, Octrees, K-d trees and BSP trees are the most researched spatial subdivision methods for accelerated ray tracing. Spatial subdivision methods as the name suggests splits the scene into smaller parts called cells. The idea being that the further away an object is from the ray, the earlier its cell should be ruled out from further intersection testing.

For our project we chose bounding volume hierarchies instead of spatial subdivisions due to their lower memory footprint, better performance for our use-cases and overall ease of implementation [1][2]. We focused on BVH construction and iterated through 3 algorithms to improve the quality of the resulting trees.

3. Algorithm

3.1 Our Implementation of Ray Tracing

Ray intersection is determined by the shape of the object - sphere, triangle and cuboid. To achieve a variety of visual detail, we implemented three material models for our objects: a Lambertian diffuse model, a metallic model, and a dielectric model. The models are defined by a small number of attributes that control the ray scattering. Each scattering function takes an input of an incoming ray, the world space coordinate at which that ray has hit an object, the normal of that point on the object, and the material properties of the object. The functions output an RGB color used as attenuation and a new ray emanating from the hit point.

The Lambertian material's output ray is the hit normal added with a random direction, of length less than one, which is then normalized. The color is just a provided albedo attribute. The metallic material reflects the incoming ray about the hit normal and then adds a similar random direction which is weighted by a "fuzziness" attribute. A perfectly reflective material would have a fuzziness value of 0. The returned color is again a provided albedo attribute. Finally, the dielectric material first determines if the incoming ray is exiting or entering the object. We use this information to define the sign of the normal, the angle between the incoming ray and our signed normal, and the incidence-refraction ratio. The ratio, as defined in Snell's law, is provided by the material's refractive index. If the incoming ray is exiting the object, the ratio is equal to the index, and if the ray is entering, then the ratio is equal to the reciprocal of the index. We then use the refractive index and the angle between the signed normal and the ray in Schlick's polynomial to determine the probability of reflection [3]. Using the probability, we either return a refracted ray or a reflected ray. The color attenuation returned is always pure white for our dielectrics.

3.2 Bounding Volume Hierarchy Implementation

The effectiveness of BVH greatly depends on the quality of tree constructed. For example an unnecessarily large bounding box would provide positive ray intersection tests for several rays which do not intersect any of the contained objects. This is illustrated in Figure 3 below where a bounding box grouping of A, B and C, D would yield worse results than say a bounding box which grouped A,C.

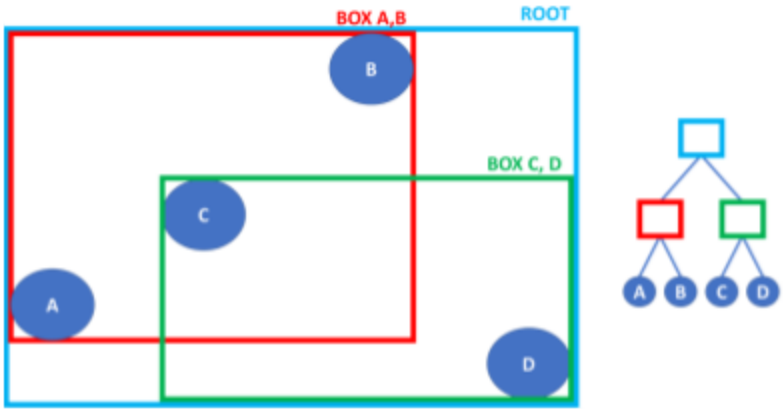


Figure 3: 2D illustration of effects of poor Tree quality on performance of the BVH

Thus we primarily focused on BVH construction for improving the quality of the Tree. Detailed results are provided later in the Section 4. For a visual representation of the tree quality in our rendered images (bounding box size and tree depth), we rasterized the bounding boxes of the hierarchy and color mapped them based on their position in the tree as shown in Figure 4.

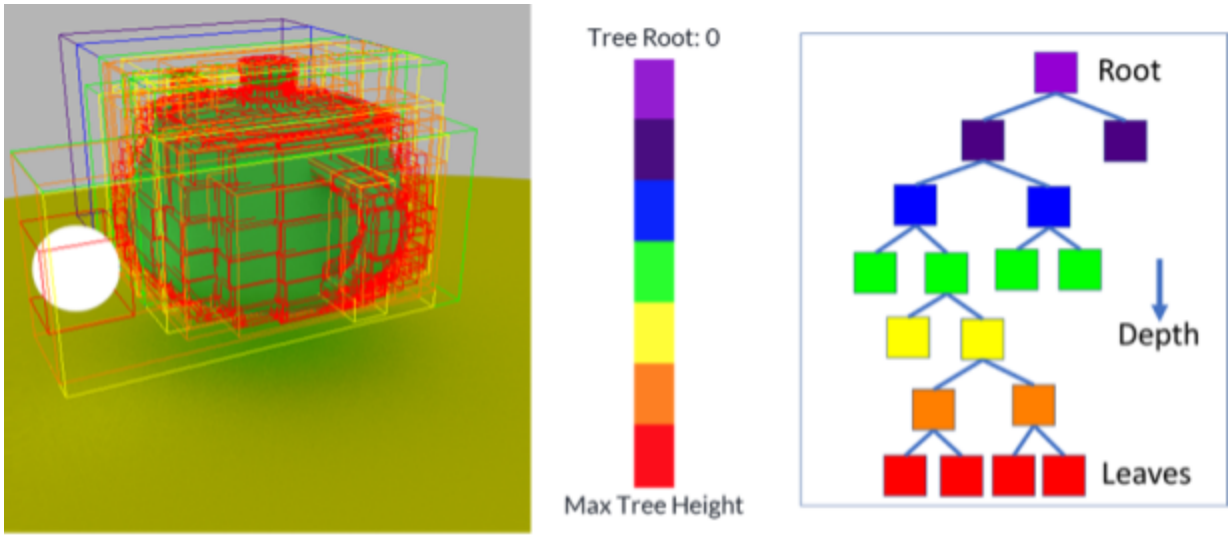


Figure 4: Visualization of the Bounding Volume Hierarchy

3.2.1 Top Down Construction

Algorithm 1 below shows how the BVH can be constructed recursively in a top down fashion. The time complexity is O(*nlogn*) due to sorting and takes negligible time in comparison to the ray tracing.

Algorithm 1 Top Down BVH construction

Input: list of primitives P
Output: a scene BVH (returns root node)
  Sort P
  $root$ = BuildTree(P)
  return $root$

BuildTree(P)

Input: sorted list of primitives P
Output: a node B which represents the root of the bvh of P
  if $|P| < |MaxLeafPrimitives|$ **then**
    Initialize B with its children set as P
    $B.BoundingBox$ = CombineBoundingBoxesOfPrimitives(P)
  else
    $\langle P_L, P_R \rangle$ = Split(P)
    $B.left$ = BuildTree(P_L)
    $B.right$ = BuildTree(P_R)
    $B.BoundingBox$ = CombineBoundingBoxesOfNodes($B.left$, $B.right$)
  end if
  return B

Sorting by Primitive Object Coordinates

In our initial implementation, we simply sorted by primitive coordinates choosing either the x, y or z coordinates. This improved performance by at least a factor of 2 for most scenes. However this approach is sensitive to cases where objects may be close in one dimension but far apart in another dimension effectively increasing the size of the bounding boxes. This is illustrated in Figure 5 where when we sort by x, the boxes are stretched in the z dimension.

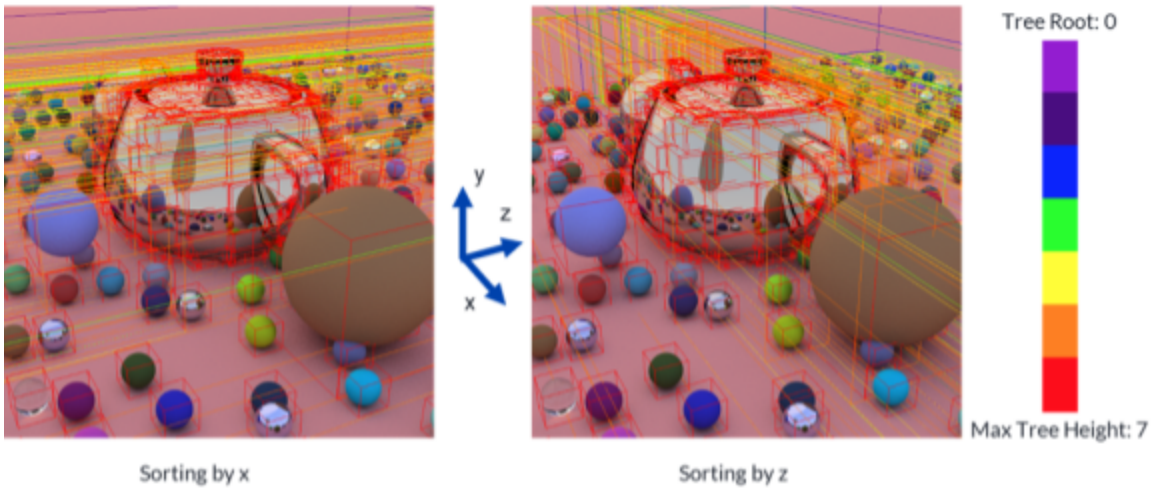


Figure 5: The reference rendered using a BVH constructed by sorting primitives

Sorting by Morton Code

The deficiencies of our initial implementation can be rectified by sorting objects by the Morton code of their centers [4][5]. The Morton code of an object provides its ordering along a space filling curve identical to the one shown in Figure 6 (left) but for a much larger grid determined by the range of maximum and minimum coordinates along each axis.

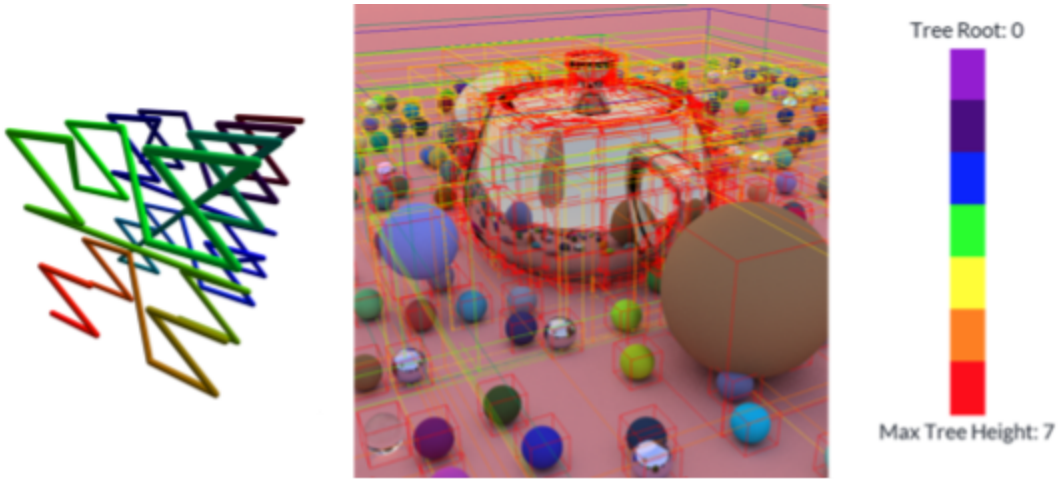


Figure 6: (left) Morton code for a 4x4x4 grid http://asgerhoedt.dk/?p=276 and (right) bounding boxes generated when sorting by morton code

Sorting by Morton code provided more consistent results with improvement of ~1.5x over coordinate sorting for the same scene. The improvement in the Tree quality is illustrated in Figure 6 (right).

3.2.2 Bottom Up Construction: Agglomerative Clustering

Top down approaches are still sensitive to the presence of large objects in the scene - like the Red Lambertian Ball below our scene. These objects greatly increase the size of the bounding boxes resulting in wasted intersection checks. To counter this, we adopted a bottom up approach wherein we clustered primitives greedily based on the minimum Surface Area Heuristic (SAH) as Proposed by Water et al. Algorithm 2 shows the construction process [6].

Algorithm 2 Agglomerative Clustering

Input: scene primitives $P = \{P_1, P_2, \cdots, P_N\}$.
Output: a scene BVH (returns root node)

```
Clusters C = P;                    // initialize with singleton clusters
while Size(C) > 1 do
    Best = ∞;
    for all C_i ∈ C do
        for all C_j ∈ C do
            if C_i ≠ C_j and d(C_i, C_j) < Best then
                Best = d(C_i, C_j);
                Left = C_i; Right = C_j;
            end if
        end for
    end for
    Cluster C' = new Cluster(Left, Right);
    C = C − {Left} − {Right} + {C'};
end while
return C;
```

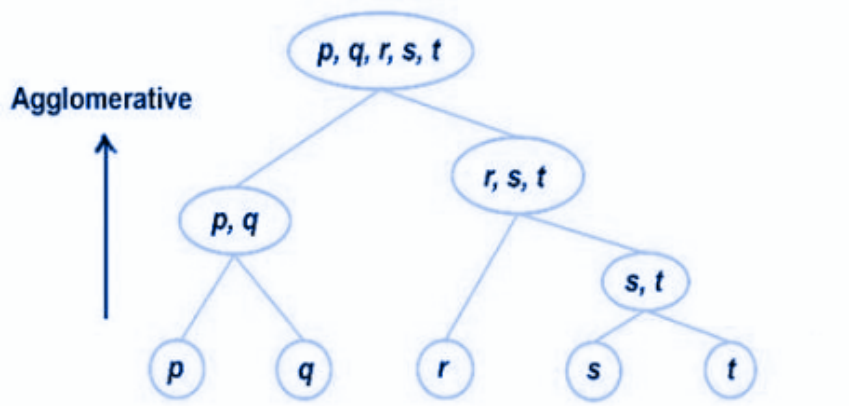


Figure 7: Agglomerative clustering tree

This resulted in slower build times of about 6 seconds for the reference scene as opposed to milliseconds for top down approaches. However, the significantly improved tree quality (shown in Figure 8) resulted in the rendering being performed 2.1 times faster than when sorting by morton code.

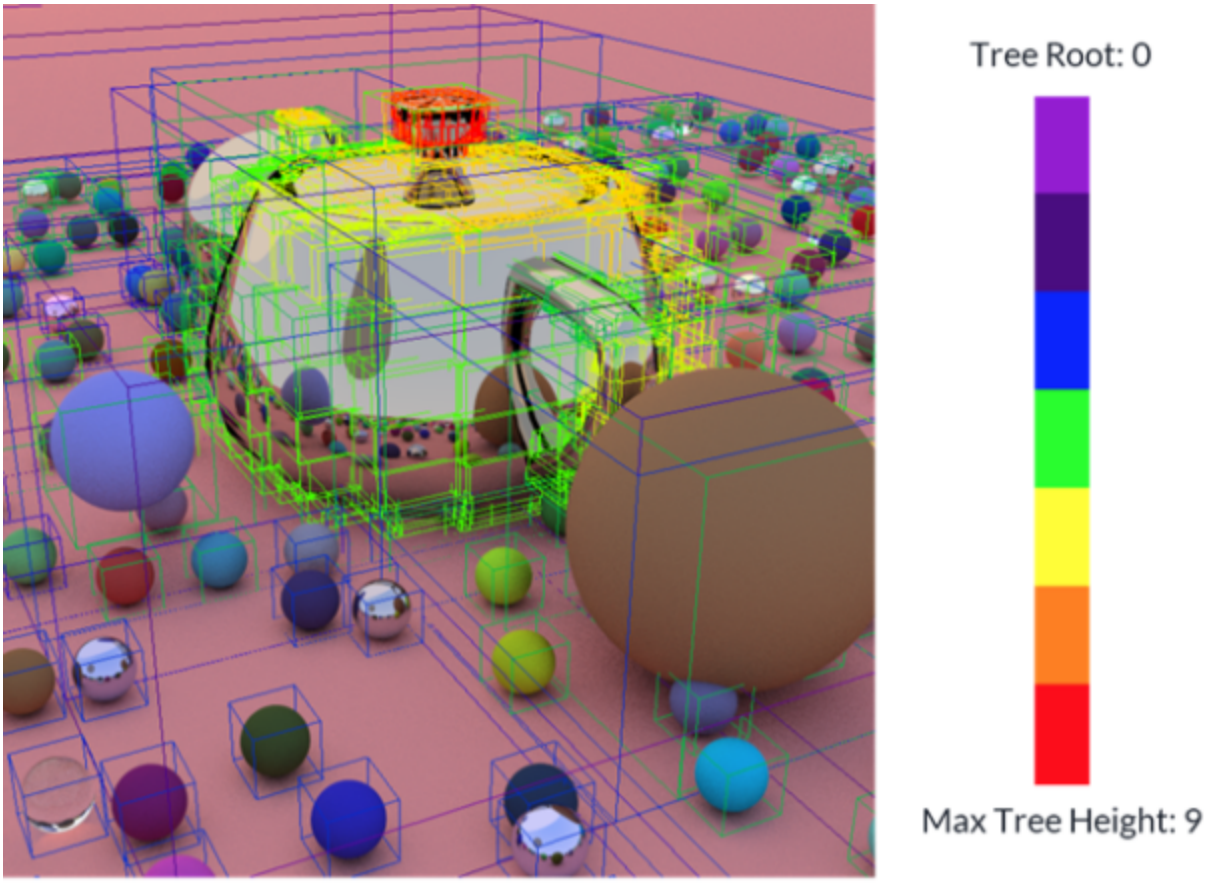


Figure 8: Bounding boxes generated by the agglomertive clustering algorithm

3.2.3 Varying Tree Node Sizes

Dammertz et al. proposed shortening the height of the BVH for faster detection [7]. They achieved this by increasing the number of child nodes for each interior node beyond 2. We implemented this and observed very minor improvements when increasing the number of child nodes to 4. Further increase of the child node size resulted in larger bounding boxes and flattening of the tree. There was a clear tradeoff between tree height and size of the bounding boxes in our tests. For better performance, further optimization of the node data structure packing would be required for better cache performance as described in Dammertz's paper. For now, the added flexibility is a useful feature to have in our BVH.

4. Evaluation

4.1 Reference Scenes

To validate the performance improvement from using bounding volume hierarchies, we created 6 test scenes. These scenes were obtained by varying the objects in the scene and the camera position as shown in Figure 9. Scene 1 and 2 has 489 spheres, Scene 2 and 3 has 896 triangles and 3 spheres and Scene 5 and 6 has 896 triangles and 492 spheres. Each of them were rendered at a resolution of 480x480 with a maximum of 4 ray bounces and 128 anti-aliasing samples per pixel. The testing process and result generation was automated using a Python Script.

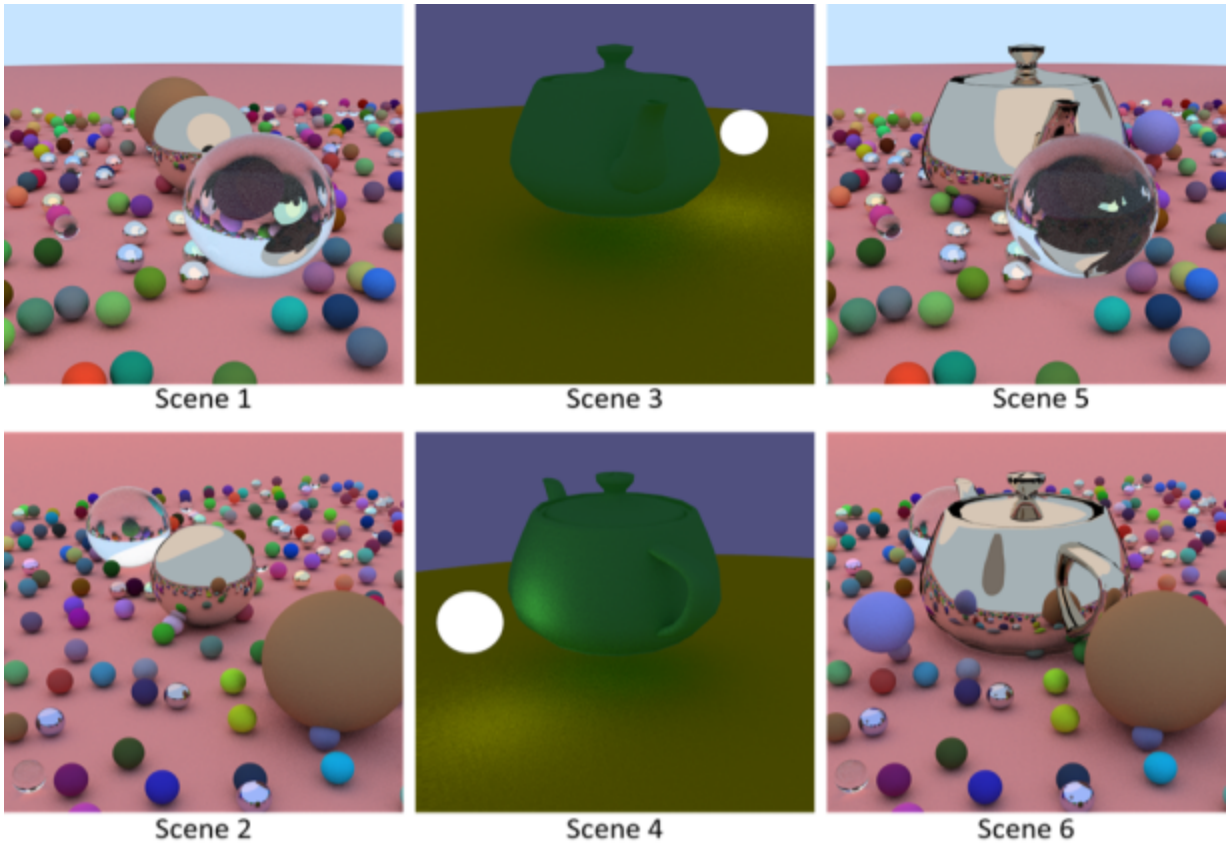


Figure 9: The reference scenes for evaluating the performance of our algorithms

4.2 Results

Table 1: Mean and Standard Deviation for Render Times

	Scene 1	Scene 2	Scene 3	Scene 4	Scene 5	Scene 6
No BVH	102.9s ±1.40	106.7s ±4.11	336.2s ±4.57	335.1s ±7.19	451.2s ±7.50	471.4s ±9.23
Top Down x sorted	48.2s ±0.49	47.3s ±0.73	49.9s ±1.15	65.5s ±0.45	155.7s ±0.90	177.7s ±1.39
Top Down Morton	62.3s ±0.47	62.7s ±0.61	47.7s ±0.25	57.8s ±1.28	96.1s ±0.46	116.2s ±1.70
Agglom. 2 children	30.4s ±0.17	34.1s ±0.40	25.6s ±3.64	25.1s ±3.00	50.2s ±4.11	55.6s ±4.75
Agglom. 4 children	30.3s ±0.34	33.1s ±0.39	24.0s ±2.17	27.6s ±2.66	48.9s ±5.60	56.2s ±8.32
Agglom. 8 children	28.3s ±0.07	29.7s ±0.18	28.6s ±4.07	30.3s ±4.18	50.5s ±5.36	58.7s ±7.95

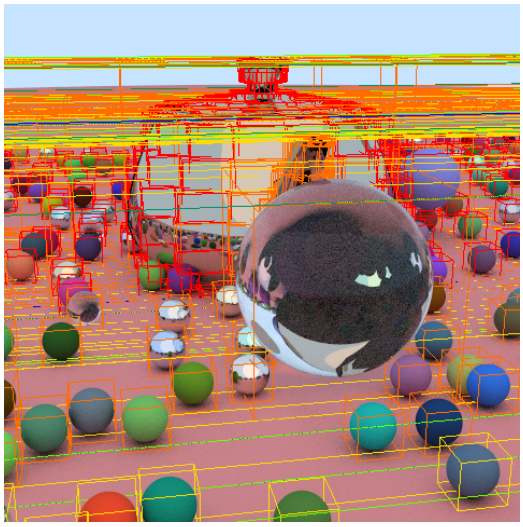
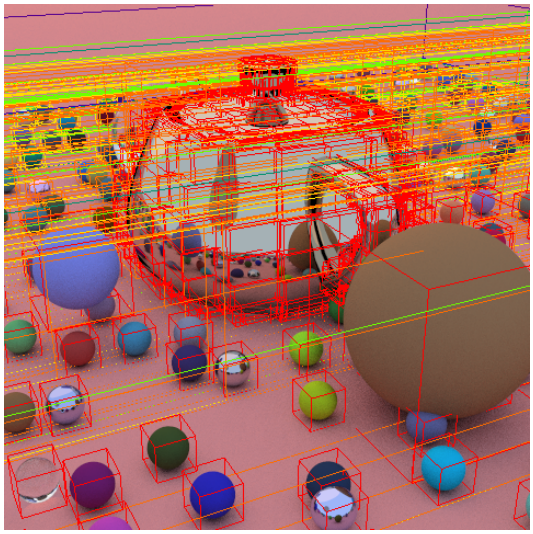
Table 2: Speedup

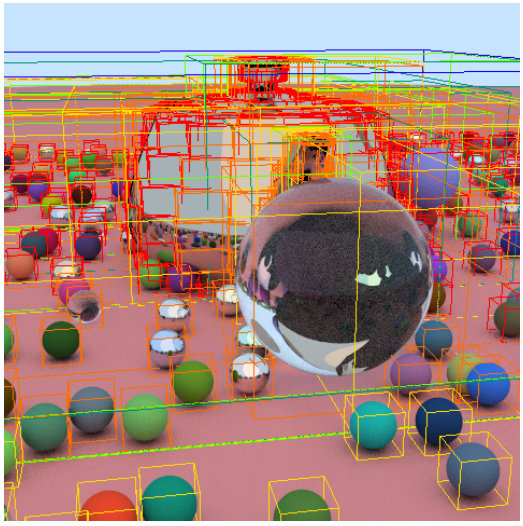
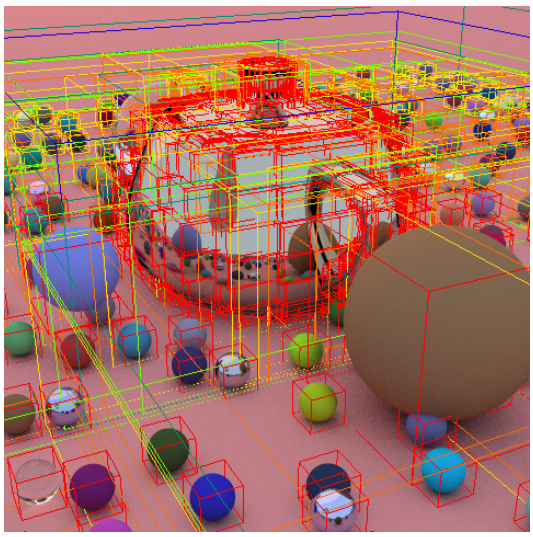
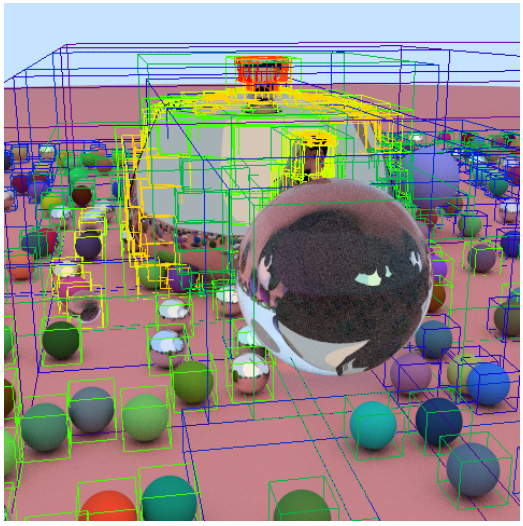
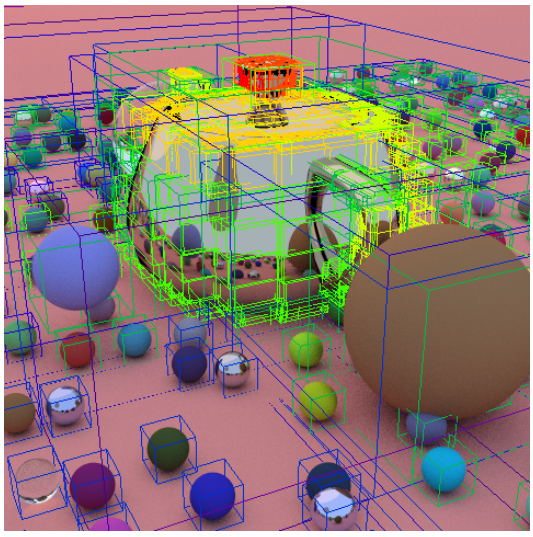
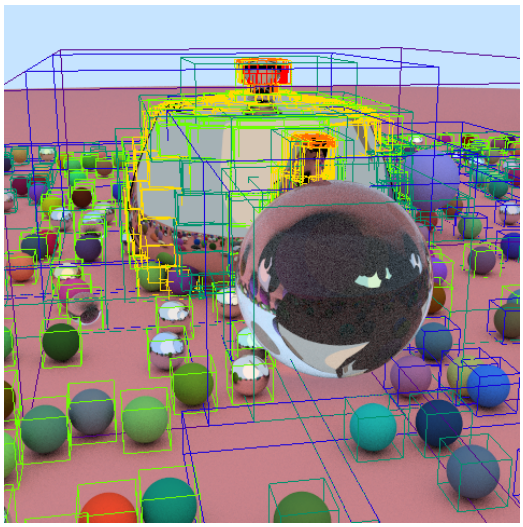
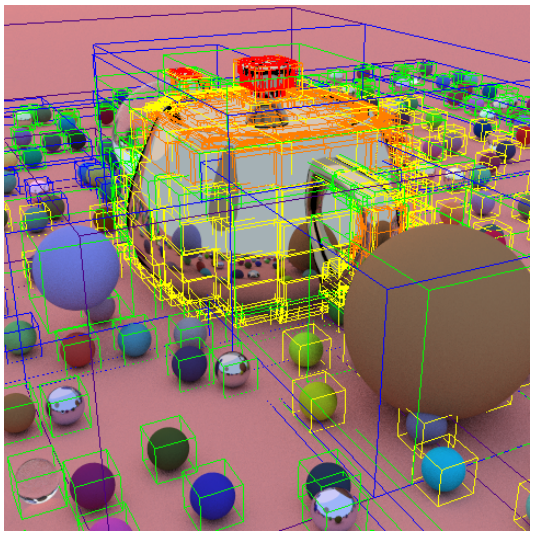
	Scene 1	Scene 2	Scene 3	Scene 4	Scene 5	Scene 6
No BVH	1.000x	1.000x	1.000x	1.000x	1.000x	1.000x
Top Down x sorted	2.134x	2.257x	6.740x	5.120x	2.897x	2.652x
Top Down Morton	1.651x	1.701x	7.046x	5.797x	4.693x	4.057x
Agglom.2 children	3.384x	3.131x	13.140x	13.363x	8.996x	8.476x
Agglom. 4 children	3.398x	3.219x	13.999x	12.121x	9.235x	8.400x
Agglom. 8 children	3.632x	3.593x	11.741x	11.051x	8.938x	8.030x

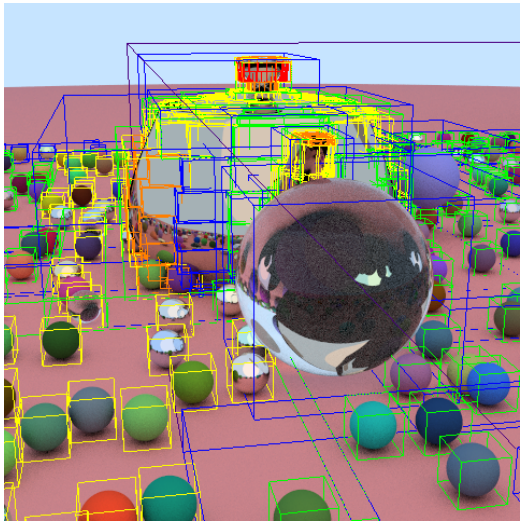
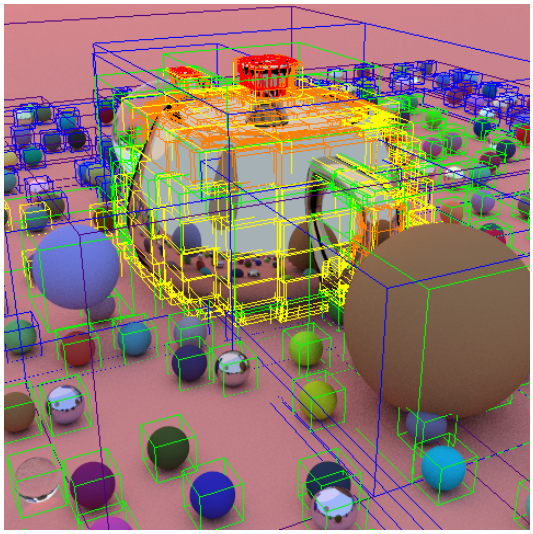
Results were averaged out over 6 runs. The tree build and renderer were multi-threaded using OpenMP and ran on 8 threads on an Intel Xeon X5570 with a clock speed of 2.93GHz.

It is interesting to note that sorting by x performs better than sorting by morton code for rendering the spheres but is much slower when rendering the triangle mesh. This can be explained by the fact that sorting by morton code clusters objects which are closer to each other spatially and hence reduces bounding box sizes. Thus for scenes where primitives are very close to each other, like in meshes, sorting by morton code will yield much better results

Table 3: Bounding Boxes for Scene 5 and 6

	Scene 5	Scene 6
Top Down x sorted		

Top Down Morton		
Agglom. 2 children		
Agglom. 4 children		

Agglom. 8 children		

5. Conclusion

In this project, we built a ray tracer capable of rendering scenes containing meshes and spheres made of three different materials with realistic global illumination. We also demonstrated that we can accelerate ray tracing significantly using bounding volume hierarchies. The performance gain increases rapidly as we increase the number of objects in the scene.

Based on our testing and visualization of the bounding boxes we can conclude that bottom up agglomerative clustering can produce high quality BVHs by minimizing the surface area of the boxes at each level of the tree. This proves to be more robust to changes in viewing orientation.

Our BVH accelerated raytracer is parallelized using OpenMP at the moment. However, the algorithm is highly parallelizable and can easily be extended to GPU execution. Further work will be required to reduce branching for GPU execution which is common in ray tracing algorithms due to varied number of ray reflections and refractions.

6. References

[1] D. Luebke and S. Parker, "Interactive Ray Tracing with CUDA", NVIDIA Technical Presentation, SIGGRAPH 2008, 2019.

[2] M. Stich, H. Friedrich and A. Dietrich, "Spatial splits in bounding volume hierarchies", Proceedings of the Conference on High Performance Graphics, New Orleans, Louisiana, 2009.

[3] C. Schlick, “An Inexpensive BRDF Model for Physically Based Rendering,” Computer Graphics Forum, vol. 13, no. 3, pp. 149–162, 1994.

[4] I. Gargantini, "An Effective Way To Represent Quadtrees", Communications of the ACM, pp. 905-910, 1982.

[5] M. Bern, D. Eppstein and S. Teng, "Parallel Construction Of Quadtrees And Quality Triangulations", International Journal of Computational Geometry & Applications, vol. 09, no. 06, pp. 517-532, 1999.

[6] B. Walter, K. Bala, M. Kulkarni and K. Pingali, Fast agglomerative clustering for rendering. 2008, pp. 81-86.

[7] H. Dammertz, J. Hanika and A. Keller, "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays", in Proceedings of the Nineteenth Eurographics conference on Rendering, Sarajevo, Bosnia and Herzegovina, 2008.