Implementing AND Benchmarking Different Acceleration Data Structures FOR Ray Tracing

Master Project Rendering Track

Author

Alhajras Algdairy

Advisor

Prof. Dr.-Ing. Matthias Teschner

Albert-Ludwigs-University of Freiburg
Chair of Computer Graphics

June 17, 2022

Acknowledgments

This project results from hard work and cumulative knowledge gained through seminars, labs, and lectures in the chair of computer graphics at the University of Freiburg under Prof. Dr.-Ing. Matthias Teschner guidness and supervision. Those materials and their resources were the fundamental building blocks to reaching this point with the proper feedback from the professor.

Previous work

This project is based on a simple raytracer that I have been developing in a Lab course at the University of Freiburg. The raytracer provides basic functionality, but it is not a commercial raytracer; it still needs some improvement from different perspectives. This project will focus only on the acceleration data structures and explore different methods, and tests their performance against each other.

Abstract

This report aims to implement and analyze different acceleration data structures integrated into a raytracer. Three types will be discussed: Bounding Volume Hierarchies (BVH), Linear Bounding Volume Hierarchies (LBVH), and Kd-Tree. The report shows the proof of concept and how to implement each data structure. Furthermore, we will explore each method's strengths and weaknesses by using different scenes and models.

Keywords: [ray tracing] [acceleration structure] [BVH] [LBVH] [object subdivision] [spatial subdivision] [KD-Tree]

Contents

1 Introduction to Ray tracing

1.1 Overview

Ray tracing is an algorithm to simulate how light behaves in a 3D scene to generate real-life digital images in a computer. This process is called rendering. Rendering is used in various applications, such as Gaming, Animation, Engineering, and Moviemaking.

Raytracing has three different implementations: Forward Raytracing, Backward Raytracing, and Hybrid Raytracing; regardless of which implementation is used, the general idea of the algorithm is quite simple but extremely powerful; it is to project the 3D scene into a 2D plane (Image), to do so, a resolution must be chosen beforehand to divide the plane into small squares named (Pixels), then we try to evaluate the color and illumination of each pixel. We cast several rays from the sensor/camera toward the scene for each pixel. We search for any intersection with all primitives, and if we hit one, we save it in a list. Depending on the distance between the sensor and the hit primitive, we can evaluate the nearest one to the sensor and pick it up. We can retrieve the primitive color from its original color property and start shading it. This includes knowing if the primitive is in a dark or bright region in the scene. This expensive process is recursively executed until we hit the light source or reach a predefined depth. For thousands of primitives and more, testing the intersection is a performance challenge.

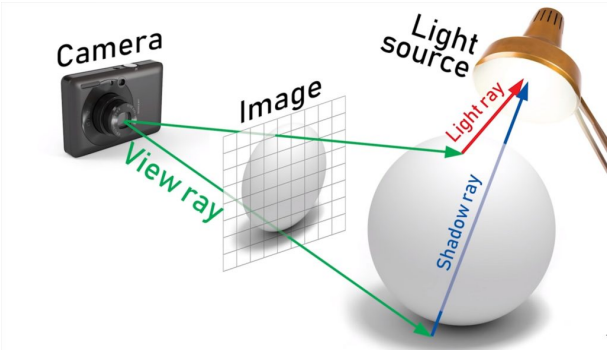


Figure 1: Ray casting illustration, where rays are travelling from the camera/sensor towards the scene.(GEORGE KIMATHI, 2020)

Other approaches such as Rasterization can be used to overcome these obstacles; however, Raytracing algorithms can provide more realistic images than Rasterization even though they take more time to render a scene. This is the reason why Rasterization is used for real-time applications such as gaming more than Raytracing; on the other hand, Raytracing is more often used in offline applications such as simulations and interior design software.

Figure 2: NVIDIA RTX Technology Realizes Dream of Real-Time Cinematic Rendering.(Brian Burke, 2018)

In Game Developers Conference - NVIDIA announced NVIDIA RTX™, a ray-tracing technology that brings real-time, cinematic-quality rendering to content creators and game developers. (Brian Burke, 2018), the results where promising and shows that having a real-time raytracer is possiable with the advancement of GPUs.

The lighting and the shadows are so detailed in the scene that they look like a picture taken from a camera and not generated from a computer. The real challenge is real-time raytracers, which require rendering approximately 50 high-quality frames per second.

We should understand how challenging to render one frame that consists of thousands of primitives in less than one second.

1.2 Motivation of using acceleration data structures

In the Raytracing pipeline, we mentioned that each ray must execute an intersection test with all the primitive contained in the scene. This means if we have N number of pixels and P number of primitives, this will produce a complexity of $O(N.P)$, this means even if one pixel only contain a premitive we would have to go to all other pxiels and still test them against all premitives.

Brute forcing complexity is high, increasing linearly with the number of primitives. Moreover, some features such as anti-aliasing require more rays per pixel which increases the complexity to $O(N.P.S)$ where S is number of samples per pixel.

Using more samples will often produce a higher quality frame; additionally, for better illumination results, more recursion (Shadow rays) must be cast after each intersection. We need more rays but with fewer intersection tests as possible.

Let us illustrate this with a simple example, giving a raytracer that uses S: number of samples per pixel = 5, N: number of primitives = 100000, and P: number of pixels = 1280 x 1024, maximum depth = 3. This will give us a number of intersection tests, approximately = 1280 x 1024 x 100000 x 5 x 3 = 1.96608×10^{12} intersection test, assuming the machine we are using will spend 0.01ms in each ray. This means for this simple scene the raytracer will render the frame in 220 days. Some scenes contain millions of primitives making, rendering them by brute-forcing imposable.

Luckily one can use preprocessing algorithms to reorder and group the primitives to make them quickly tested. Spatial subdivision and object subdivision are the two basic types of data acceleration structures. Spatial subdivision algorithms partition three-dimensional space into areas and keep track of which primitives overlap which regions. On the other hand, object subdivision algorithms gradually subdivide the scene's objects into smaller groups. This way, we can only test the primitives that have a higher probability to intersect with the ray rather than testing all primitives that are not relevant to the region the ray passes through.

2 Bounding Volume Hierarchies (BVH)

2.1 Concept

The basic idea of the BVH is simple yet powerful. It is to wrap all primitives in a virtual bounding box. This box will act as metadata to show the limits of the primitive and has no idea of how the shape of the primitive inside it. This concept will make it easier to test the primitives because one can wrap a complex model with one bounding box, and if the ray intersects the bounding box of the model, then and only then can we test its primitives.

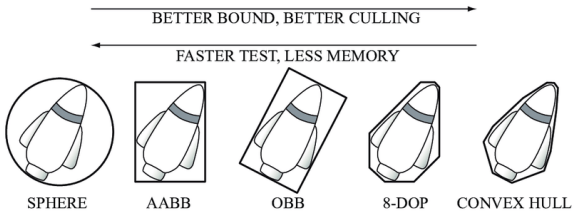


Figure 3: Bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), convex hull (Ericson, 2004)

The bounding box can have different shapes, as shown in Figure. The simpler the bounding box is, the faster and easier to test the intersection. However, the less tightened it becomes and space between the primitive the bounding box boundaries. A compact bounding volume will assist us have the fewest overlaps with other bounding volumes in our scene, while a quick intersection test will help reduce time complexity. In this report, we will be using AABB, because it is easy to implement and easy to test its intersection, and less memory consumption because it only needs to save two double points the minimum and maximum edges of the bounding box. One can use a combination of all of them but in this report we will only use the AABB.

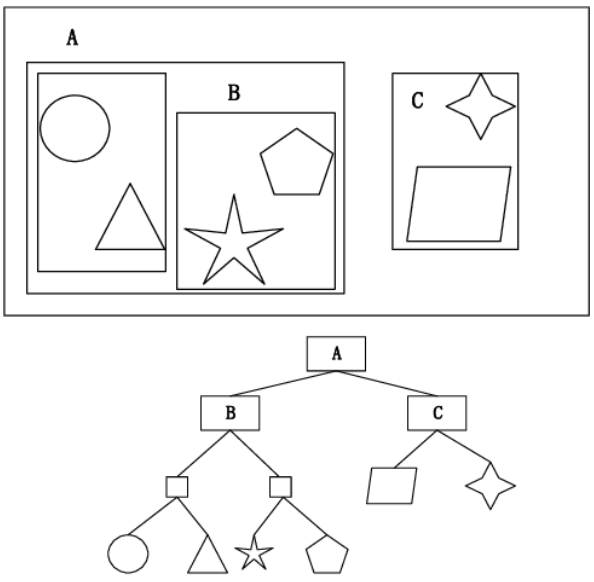


Figure 4: BVH tree result based on a simple scene. (Ericson, 2004)

Let us look at the figure as an example; this scene is consistent of 6 objects. Looking at

the generated binary tree, we can note that every two objects are bounded by one AABB, then each two AABB are bounded by a bigger AABB. We recursively do this until we reach the root node that covers the limits of the whole objects.

Without the BVH, we will have to go through all the 6 objects for each ray, even for the rays that do not intersect with any object. On the other hand, when we introduced BVH, we can note that for the rays that do not even intersect with the root AABB node, we do not go further to test its children. This means we will only test once rather than six times. Looking at the sphere, if a ray intersects with it, we will only have to go through a path that goes A -¿ B -¿ AABB sphere parent -¿ Sphere. These are four tests instead of 6.

We know that for a binary tree, the worse case complexity is O(n), but if it is a balanced tree, it becomes O(logn), this is significant, but the catch is we should try to build a balanced binary tree. This is where splitting criteria come in handy.

2.2 Implemntation

BVH has two main implementation pillars, Construction of the BVH tree and Traversal over the tree.

2.2.1 Construction

Before going through the construction details, we have to discuss one essential key in building the BVH tree. BVH tree comes in different flavors depending on its splitting criteria. Since we are building a tree, the critical question is, when do we split the node? and which axis to choose for splitting? Firstly we will answer the first qquestion. There are three popular strategies to split the node:

- **Median of the centroid coordinates (Object median)**:The median of primitives, meaning if we have the next primitive positions in the x-axis as follows $1, 3, 3, 6, 7, 8, 9$ the median will be the primitive in the middle, which is 6. This strategy will produce a well-balanced tree because it splits the primitives into two equal subtrees. Because this method is intuitive to implement and produces a well-balanced tree, this strategy will be adopted in this report.
- **Mean of centroid coordinates (Object mean)**: Using the man of the premitives going back the example $1, 3, 3, 6, 7, 8, 9$, the mean of this set is 5.2, where the left child node will contain $1, 3, 3$ and the right child $6, 7, 8, 9$
- **Spatial median**: Splitting the premitives volume into two equal parts. For example, if we have four spheres with the next radius $1, 1, 1, 4$, this strategy will split it as follow, the left child is $1, 1, 1$ and the right child is 4 because it will split based on the volume/area of the sphere and not the position as the previous strategies.

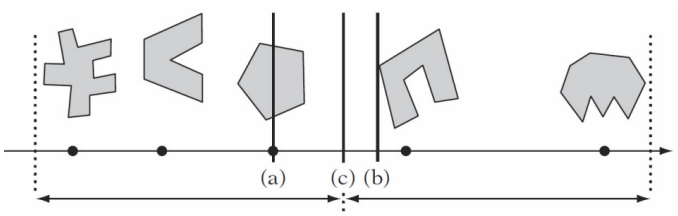
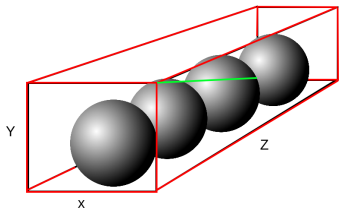


Figure 5: An example of (a) Object median splitting (b) Object mean splitting (c) Spatial median splitting based on (Ericson, 2004)

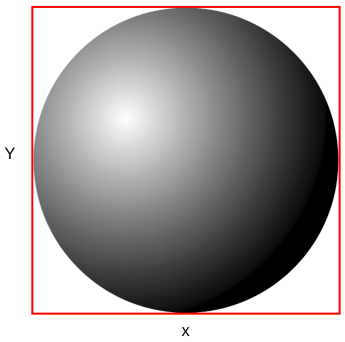
To answer the second question, which axis to choose for splitting? One can use any axis, but this is a naive way to split. What happens if all primitives have the same z-axis and y-axis, but only the x is changing, and we randomly choose to split by z-axis! This will usually produce an unbalanced tree. The optimal way is to calculate the longest axis for each node (AABB box). This way, we can try to split by the longest axis, which will produce a better-balanced tree. We can find the longest axis of the AABB box by using its diagonal d.

$$\mathbf{d} = \mathbf{AABB}_{max} - \mathbf{AABB}_{min} \tag{1}$$

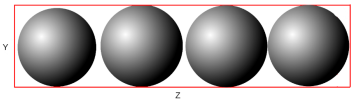
In the Figure, we have four spheres and an AABB covering all the spheres. If we would like to choose one axis to split from, we can randomly start with the x-axis. As it is illustrated, choosing this axis will not give us a middle point to split point since all spheres have the same x-axis. Same for the y-axis. On the other hand, we can see that the Z-axis will split the AABB box into two AABB boxes, each containing two spheres.



(a) AABB covers four spheres and its diagonal in green.



(b) Finding a split point for x and y axis is not useful for buel-ing a blaanced tree.



(c) Z-axis has the best premi-tives spread that build a bal-anced tree.

Figure 6: The scene contains four spheres with the same x-axis and y-axis but different z-axis.

2.2.2 Visual Illustration

For a better illustration of how to build the BVH tree by using the Median of the centroid coordinates splitting criteria and also by choosing the longest axis to split from, we will look at the next scene as an example:

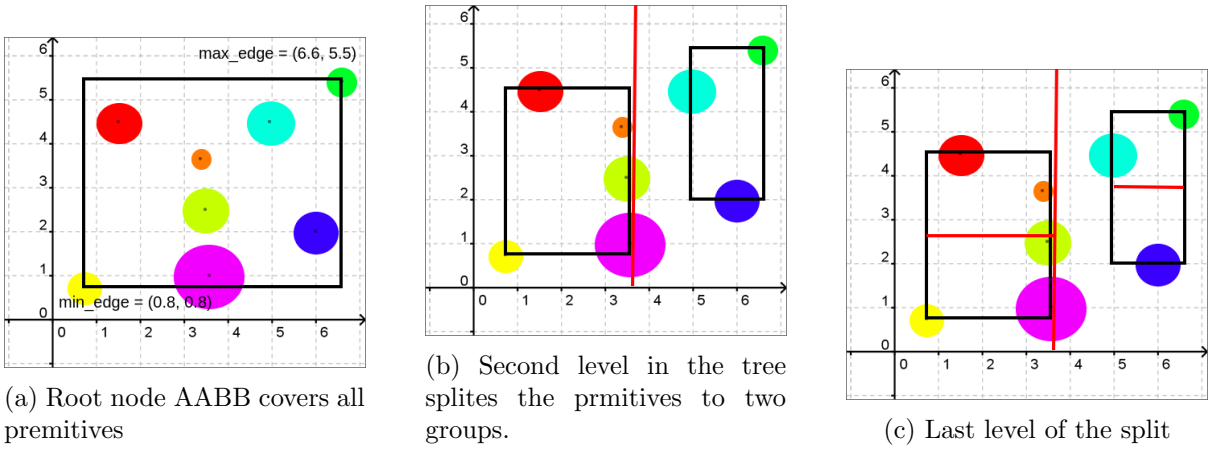
Firstly, we start creating the root node by calculating the *minimum_edge* (the smallest bounding point in our AABB) and the *maximumu_edge* point (the largest in the AABB box). This can be easily done by iterating through all primitives and assigning the smallest primitive centroid to the *minimum_edge* and the largest primitive centroid to the *maximum_edge*.

The second step is to split the primitives in the node by using the AABB boundaries. As we mentioned, we will find the longest axis. To find the longest axis, we subtract the *minimum_edge* from *maximum_edge* and find the maximum of both axes $max(maximum\_edge - minimum\_edge)$. This will give us $max((6.6 - 0.8) - (5.5 - 0.8)) = max(5.8, 4.7) = 5.8$, which is the x-axis.

Now we calculate the splitting point by just halfing the distance between *maximum_edge* and the *minimum_edge*, $split\_point\_x = (0.8 + 6.6)/2 = 3.6$.

We then create a left node that contains the primitives less than 3.6 and a right node that contains the primitives with a centroid > 3.6.

Recustivly we keep doing this until we reach a $maximum\_leafs$ parameter which is predefined, and in this report, it is 1. This means the leaf node will only hold one primitive.



(a) Root node AABB covers all premitives

(b) Second level in the tree splites the prmitives to two groups.

(c) Last level of the split

Figure 7: Creating AABBs boxes

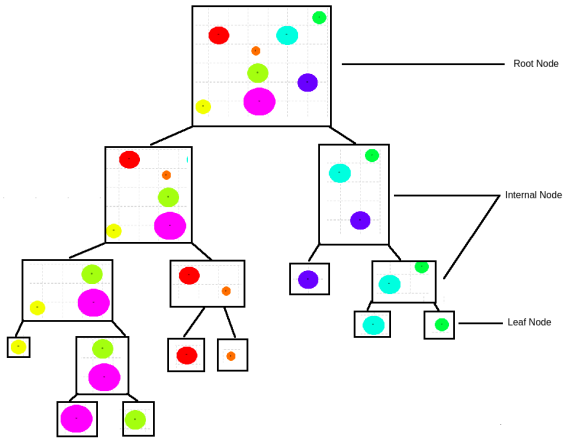


Figure 8: BVH Tree example by using the median split criteria

2.2.3 Traversal

The traversal phase of BVH is quite intuitive if you are familiar with how binary tree traversal works. After building the BVH tree, we start with its root node, which has the maximum and minimum boundaries of the biggest AABB that bounds the whole scene. We test if the ray intersects the AABB of the root node. If it is true, we go to its left and right children and do the same process; otherwise, we terminate. Recursively keep doing this until we reach the leaf node; the leaf node is the node that has no children, and it is the only node that contains a primitive inside it. Then we go through the list of primitives and test each one if it is intersected with the ray or not.

As discussed, the leaf node can include a list of primitives and not only one by setting a parameter called $max_leaf_primitives$ to 1; this depends on how you configure the BVH building tree method. In this report, I set the list to make it only contain one primitive. It is worth noticing that the bigger $max_leaf_primitives$ is, the smaller the depth of the tree becomes because we do not have to keep splitting until we reach a leaf that only contains primitives less or equal to $max_leaf_primitives$, hence less traversal time and less memory consumption because of the number of nodes saved, on the other having a long list of primitives inside the leaf node is bad for the performance because this will get us to the same challenge we would like to solve by introducing the BVH which is avoiding brute force algorithms.

The intersection test is split into two main steps. First, we go through all the nodes in the tree, and if we hit a leaf node, we add all its primitives into a list called *hit_list*. Afterward, we go through these candidates and find which are intersected and which one is the closest. The following two pseudo-codes explain the intersection test for the BVH.

Algorithm 1 Pseudocode of the method *BVH_intersect*

```
Require: ray_p, ray_d, current_node, hit_list
  if current_node → boundingBoxIntersection(ray_p, ray_d, current_node) = false then re-
  turn None
      premitives ← current_node → premitives
      if premitives → size ≠ 0 then
          hit_list → push(premitives)
      else
          BVH_intersect(ray_p, ray_d, current_node → left_child, hit_list)
          BVH_intersect(ray_p, ray_d, current_node → right_child, hit_list)
```

Algorithm 2 Pseudocode of the method *find_intersected_premitive*

```
Require: ray_p, ray_d, hit_list
  closest_premitive ← hit_list[0]
  for premitive in hit_list do
      if premitive → intersect then
          if premitive → isClosest then
              closest_premitive ← premitive
  return closest_premitive =0
```

3 Linear Bounding Volume Hierarchies (LBVH)

3.1 Concept

While BVH boosts the renderer's performance and makes it possible to render millions of primitives in minutes, it introduces an extra step in the rendering pipeline, which is a preprocess for building the tree. The building tree process can take half of the rendering time; however, creating this tree will make the render faster to find the intersected primitive. The question now is, can we do something about it?

One way to solve this is to parallelize this process. This can be done if the subtrees are independent of each other. This is what Linear Bounding Volume Hierarchies do.

The first step is to transform the building tree problem into a sorting problem. We map the 3D centroid points for each primitive to one value that can be sorted. Morton codes do this mapping. Morton code map multidimensional data to one dimension while preserving locality of the data points.[WIKIPEDIA]. The transformation is done by interleaving the bits of the premitive centroides in base 2. for example taking a 3D point (x,y,z) the result mortone code will be

$$...z_3y_3x_3z_2y_2x_2z_1y_1x_1z_0y_0x_0$$

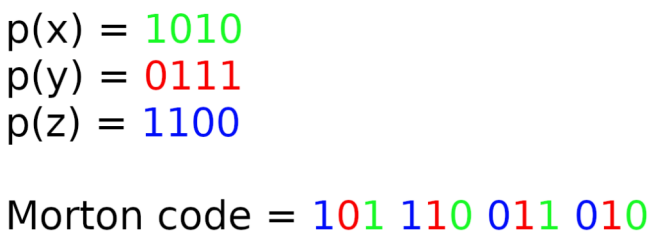


Figure 9: Example of mapping a 3D point to a morton code.

After mapping the 3D point to one 1D Morton code, we can use a sorting algorithm as a Radix sort.

3.1.1 Visual Illustration

To better understand how the LBVH constructs the binary tree, we will illustrate this with a scene as shown in Figure. The scene is made out of 16 primitives. The first step is to generate the Morton code for each centroid point. Looking closely into the generated Morton codes of each primitive, we can note that each bit acts as a splitting plane that splits the primitives into more than one region.

The first bit will split the y-axis into two regions. The first region with the first bit equals 1, and the second region with the first bit equals 0. The second bit will split the scene into two areas by an x-axis plane. The third bit will split the y-axis into four regions.

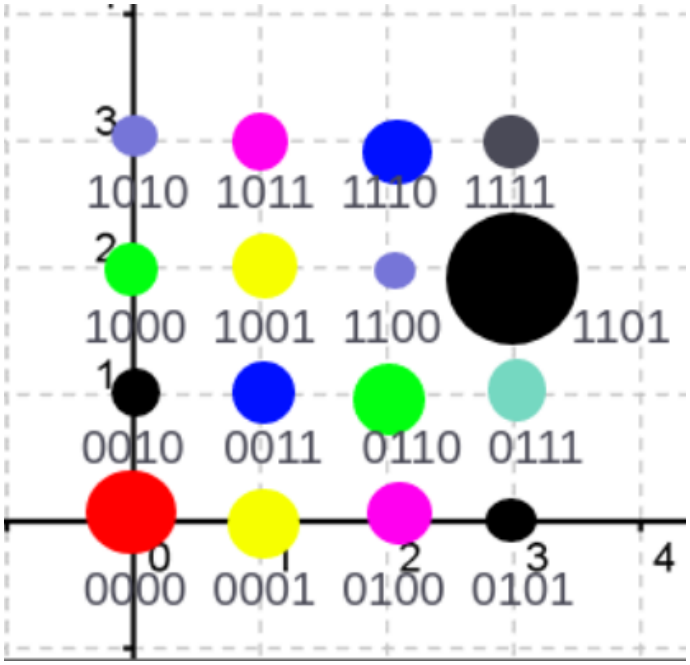
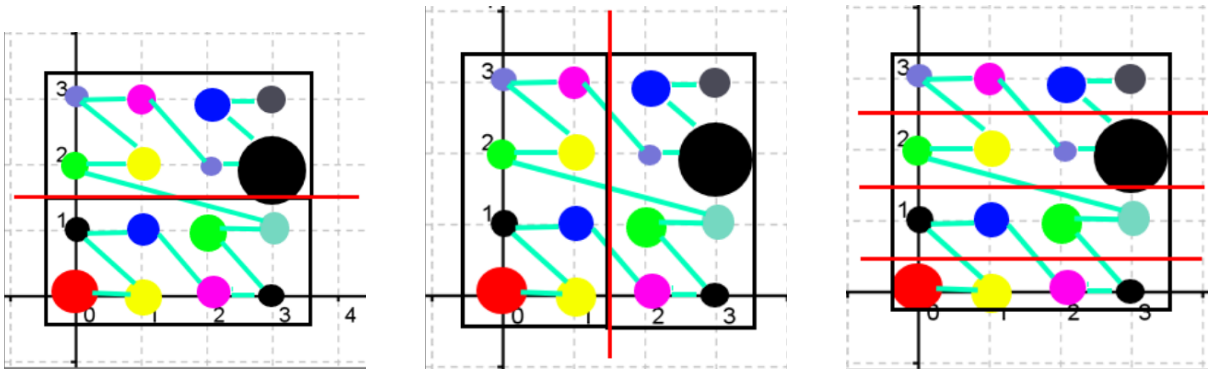


Figure 10: The values of various bits in the Morton value indicate the region of space that the corresponding coordinate lies in.



(a) First split on x-axis (b) Second split on y-axis (c) Third split on x-axis

Figure 11: (a) In 2D, the high bit of the Morton-coded value of a point's coordinates defines a splitting plane along the middle of the y-axis. If the high bit is set, the point is above the plane. (b) Similarly, the second-highest bit of the Morton value splits the x-axis in the middle. (c) If the high y bit is 1 and the high x bit is 0, the point must lie in the shaded region. (d) The second-from-highest y bit splits the y axis into four areas.

3.2 Implemntation

The implementation can be done in two main steps, building the tree and tree traversal. Traversal will not be repeated since the produced tree is a binary tree similar to the BVH. Hence the same traversal algorithm used in the BVH will be utilized for LBVH. But the main focus of the LBVH is building the tree.

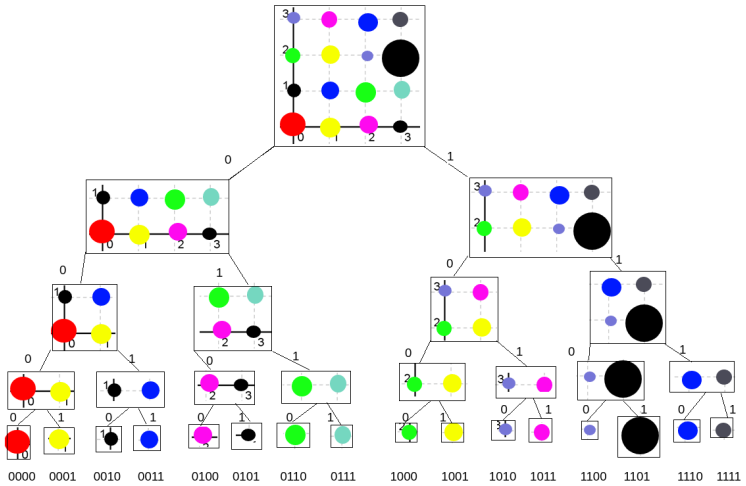


Figure 12: Generated tree by LBVH.

3.2.1 Construction

Building the tree needs to calculate the Morton code for each primitive. There are two methods for generating Morton code, magic bits, and a lookup table. We will use the magic bits method. After generating the Morton code for each primitive, we will use the Radix sort.

Next step to building the tree, we would like to find the splitting point. As we discussed before, LBVH splits the space into two regions based on the first-bit difference. This means the leftChild will contain all primitives with a Morton code with a first bit equal to 0, and the rightChild will contain the primitives with Morton codes with a first bit equal to 1. Since the Radix sort already sorts the primitives, we can do the splitting by simply using an Exponential search.

We start with a random split point. Let us say the last points. Then we start decreasing exponentially to search for the first bit differing from 0 to 1. One way to check this is to use some bit operations; however, another approach is to count the number of leading zeros and compare them with the previous one. If they differ, then we can split.

Until this point, we are not gaining much if using LBVH. As we discussed, LBVH special is how easily we can parallelize it. To achieve this, we will use a simple conditional multithreading mechanism. For building the tree we will only create a new thread of the number of premitives in the subtree is more than 100 premiitive. Note that this is a random number chosen, but we do not want to create a new thread for small subtrees because it is not worth it anyway.

Algorithm 3 Pseudocode of the method $LBVH\_construct$

function ConstructLBVH($allSceneObjects$, $sortedMortonCodes$, $sortedMortonCodes$, $startIndex$, $endIndex$, $totalNodes$)
    $totalNodes++$
    $node \leftarrow Node()$
    $aabbBoundries \leftarrow findBiggestAABBBoundries(allSceneObjects)$
    $objectsNumber \leftarrow endIndex - startIndex$
    if $objectsNumber = 0$ **then**
        $node \rightarrow isleaf \leftarrow true$
        $node \rightarrow objs \leftarrow allSceneObjects[startIndex]$
        return $node$
    $midPoint \leftarrow findSplitPoint(sortedMortonCodes, startIndex, endIndex)$
    $node \rightarrow leftchild \leftarrow constructBVHNew(allSceneObjects, startIndex, midSplitIndex, totalNodes)$
    $node \rightarrow rightchild \leftarrow constructBVH(allSceneObjects, midSplitIndex, endIndex, totalNodes)$
    return $node$

References

GEORGE KIMATHI (2020), *https://www.dignited.com/62084/how-ray-tracing-works/.*

Brian Burke (2018), *https://nvidianews.nvidia.com/news/nvidia-rtx-technology-realizes-dream-of-real-time-cinematic-rendering.*

Ericson, C. (2004).*Real-time collision detection.* Crc Press.

Teschner M (2022), *Advanced Computer Graphics Stochastic Raytracing.*

Pharr, M., Jakob, W., Humphreys, G. (2016). *Physically based rendering: From theory to implementation.*

Karras T (2012) *Thinking Parallel: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/.*

Jacobson, A. (2021). *Common 3d test models.* Retrieved from https://github.com/alecjacobson/common-3d-test-models

Wald, I. (2007). *On Fast Construction of SAH based Bounding Volume Hierarchies*

https://en.wikipedia.org/wiki/Z-order$_curve$