Rendering Master Project
WS 2021/2022

Implementing and Analysing Acceleration Data Structures in Ray tracing

Report by Adil Rabbani
Supervised by Prof. Dr.-Ing. Matthias Teschner

Contents

1 Acknowledgments

I would like to thank **Prof. Dr.-Ing. Matthias Teschner** for giving me the opportunity to work on this topic and advising me throughout this lab. Moreover, this report extensively uses resources including, **Physically Based Rendering: From Theory to Implementation** (Pharr, Jakob, & Humphreys, 2016), **Peter Shirley's Ray tracing series** (Shirley, 2020a) and (Shirley, 2020b) and many other resources which will be cited where ever necessary. This report would not have been possible without the help from these resources.

2 Abstract

Acceleration structures are an important part of any ray tracer as without them tracing even a single ray would take time linear to the number of primitives in the scene since the ray would need to be tested against each primitive in the scene to find the closest intersection. But many of the cases, the ray misses majority of the primitives in the scene and hence doing so is extremely wasteful for time and compute power. Since the ray only cares about the closest intersection with a primitive, there are acceleration structures which help grouping as well as ordering many primitives in the scene and simultaneously checking if a ray hits any of them. This speeds up the rendering process as many of the ray-object intersection tests are too expensive to compute. Mainly, there are two broad categories of acceleration structures namely, **spatial subdivision** and **object subdivision**. Spatial subdivision algorithms divide 3D space into regions and record which primitives overlap which regions. On the other hand, object subdivision algorithms progressively break the objects in the scene into smaller sets of objects. This report implements and investigates 3 different kinds of acceleration data structures in ray tracing namely, **BVH (Bounding Volume Hierarchies)**, **Uniform Grid**, and **KD-trees (K-dimensional trees)**. BVH and LBVH are examples of object subdivision accleration structures while Uniform Grids and KD-trees are examples of spatial subdivision acceleration structures. This report would also be analysing when are these structures preferable from one another in different scenarios and what are the pros and cons of using either of them.

3 An overview of Ray tracing

4 Importance of acceleration data structures

5 Uniform Grid

5.1 Concept

Spatial subdivision or space subdivision is a technique to reduce intersections with the primitives in the scene by partitioning the space of the scene into regions or cells. Once the space is subdivided into cells, each cell has their own list of objects. This helps reducing the number of intersection tests because when a ray is traced in a particular region, we can ignore the objects that are not in the same region as the ray intersects. A very simple example is shown in the figure below:

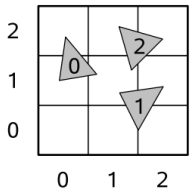


Figure 1: A grid with three triangles overlapping the cells. (Lagae & Dutré, 2008)

Figure.1 shows a 3x3 grid used to partition a scene consisting of 3 triangles. It is obvious that when our ray intersects cell (0, 1), we can just test the objects that are contained in that cell (triangle 0) and we need not care about the triangles in other regions. This way of partitioning space into cells or voxels (in 3D) is called a **uniform grid**. Uniform grid was one of the first proposed acceleration structures (Fujimoto, Tanaka, & Iwata, 1986) in rendering. As discussed previously, the idea is to divide the 3D space into a 3D grid. The approach is good as it automatically divides the objects into smaller parts which are faster to test than testing an entire model consisting of millions of triangles.

5.2 Implementation

The uniform grid implementation that is used in this project is from the paper **Compact, Fast and Robust Grids for Ray Tracing**(Lagae & Dutré, 2008). The paper presents a compact grid method which consists of a static data structure for representing a grid with minimal memory requirements, more specifically exactly one index per grid cell and exactly one index per object reference. This means that we will only be storing just one index for a cell in our grid and only index to reference an object in our scene. All acceleration data structures consist of two phases, i.e **constructing the structure** and **traversing the structure**. We will divide our discussion in the same way.

5.2.1 Construction

A uniform grid structure first require us to calculate the number of cells in our grid or more accurately the resolution of the grid denoted by M. The number of cells should be linear in the number of objects N in our scene (Devillers, 1988):

$$M = \rho N, \tag{1}$$

where ρ is called the grid density. The number of cells M is equal to the product of the resolution of the grid in each dimension. The resolution of the grid $M_x \times M_y \times M_z$ is therefore given by:

$$M_i = S_i \sqrt[3]{\frac{\rho N}{V}} \quad (i \in \{x,\ y,\ z\}) \tag{2}$$

where S_i is the size of the bounding box of the grid in dimension i i.e $(max_i - min_i)$ and V is the volume of the bounding box i.e $(S_x * S_y * S_z)$. The value of ρ in the above formula is debatable and often chosen between 5 to 10. Different papers suggest different values but the paper (Lagae & Dutré, 2008) suggests to use a value of 4 which is based on the **time to image** rather than the render time as shown below:

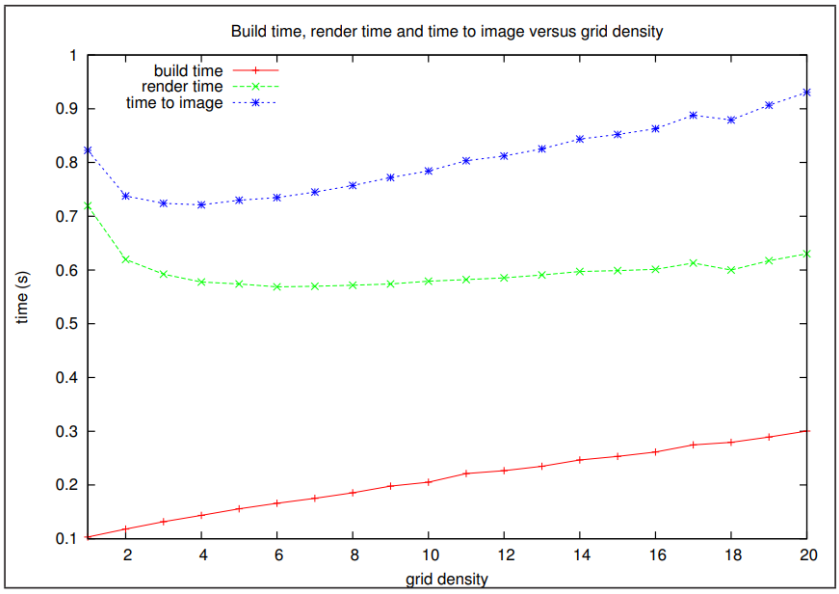


Figure 2: Grid density. Build time (red), render time (green) and time to image (blue) versus grid density for the Happy Buddha scene. A grid density of 4 minimizes the time to image. (Lagae & Dutré, 2008)

It can be seen that the value 4 gives a nice balance for the time it takes to build the structure and the time it takes to render the image.

The next question we need to answer when building uniform grids is how to insert objects in our cells. One way to do it is to use bounding boxes for each primitive in our scene and then test it if it overlaps with the cell in our grid. This is fast, however, some primitives end up in cells that overlap the bounding box but not the primitive itself which results in longer render times. More complex primitive cell tests exist but then they are slower to test. We therefore, use the bounding box method to insert primitives in our cell.

In order to get an intuition of how the structure is built, we will briefly discuss two other simpler methods of building uniform grids. One traditional way to represent grids with object references is using linked lists. This is illustrated in figure 2. Figure 2(a) shows a simple scene with three triangles which is subdivided into cells of a 2D grid. A linearized 1D version of the same grid is also shown. Here, it can be seen that a cell index (0, 1) in 2D is the same cell with the index (3) in 1D.

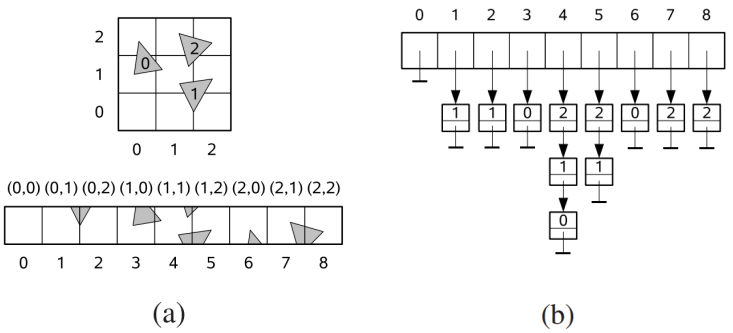


Figure 3: a) A grid with three triangles overlapping the cells and the linearized 1D version of the grid. b) A traditional grid data structure using linked lists. (Lagae & Dutré, 2008)

Figure 2(b) shows how the references are stored in each index of the linearized array. Each cell index points to a linked list which is empty if there are no objects in that cell and contains nodes if a cell contains objects. It can be seen that the cell index (1, 1) which is cell index (4) in the linearized array contains a linked list of 3 objects which are three triangles overlapping the cell in the scene.

Another method to build the uniform grid structure is to use dynamic arrays. This is illustrated in figure 3(c).

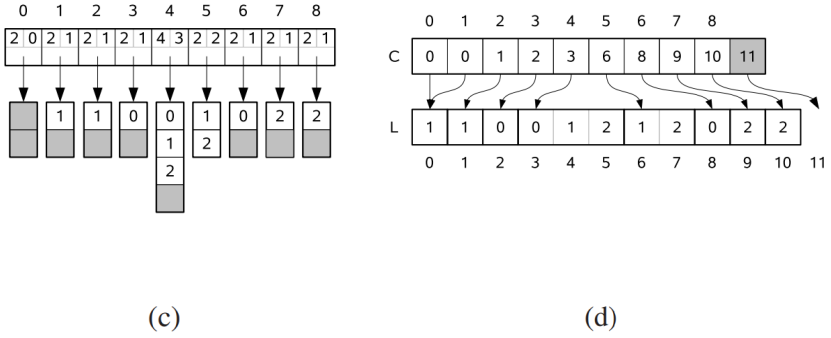


Figure 4: c) A traditional grid data structure using dynamic arrays. d) The compact grid data structure implemented in this report. (Lagae & Dutré, 2008)

The first array again represents the number of cells in our grid linearized, while each index of the array also maintains an array with a size depending on the number of objects in a specific cell. When the size is about to exceed the capacity, a new array with a larger capacity is allocated and the old array is copied and freed. We can observe that the array index 4 which represents cell (1, 1) in our grid contains an array with a size larger than other indexes hence the name, dynamic arrays.

Both the implementations described so far have a large memory overhead as they support insertion as well as removal of objects. However, if a grid is rebuilt from scratch every frame, dynamic structures like these are not needed. We now move towards the implementation described in the paper. This is illustrated in figure 3(d). The compact grid structure has two static arrays. The array L consist of the concatenation of all object lists i.e all the objects overlapped by cells are concatenated in this array. The array C stores for each cell, the offset of the corresponding object list in L i.e array C tells us where to index array L in order to fetch objects overlapped

by that cell in C.

Array L is a 1D array while array C is a 3D array of size $M_x \times M_y \times M_z$ which is linearized in row major order into a 1D array of size M. The array C can be indexed by:

$$C[z][y][x] = C[(((M_y z) + y)M_x) + x] \tag{3}$$

The number of objects contained in a specific cell can be given by $C[i+1] - C[i]$. This does not hold true for the last object list and so we extend array C by one position.

Now we'll discuss how to build the compact grid. First, the scene bounding box is computed. This is done by simply going through all objects in our scene and merging their bounding boxes. Once, we have the scene bounding box, we can compute S_i (size of the scene bounding box in dimension i) where $i \in x, y, z$, the V (volume of the scene bounding box) and by using equation (2) with $\rho = 4$, get the resolution of the grid M. Here M is the resolution of the grid in all 3 dimensions or $M_x \times M_y \times M_z$. Once, we have M, we can allocate the C array which will be of the same size and initialize all entries to 0. We also compute the cell dimension which is constant for all cells and given by:

$$D_i = \frac{S_i}{M_i} \quad (i \in \{x,\ y,\ z\}) \tag{4}$$

Next, we go through all primitives in our scene, computing their bounding box one by one, converting them to cell coordinates and checking if a primitive overlaps a cell or more than one cell. This is done by subtracting the minimum of the scene bounding box from the minimum of the bounding box of the primitive and dividing it by cell dimension D to get the first cell that overlaps that primitive. The same process is done but this time subtracting the the minimum of the scene bounding box from the maximum of the bounding box of the primitive and dividing it by cell dimension to get the last cell that overlaps that primitive. Now we iterate from the first cell to the last cell to get all cells overlapped by the primitive. By computing the linearized index using equation (3) we increment the index in cell array C for all overlaps.

There is a nice trick in the algorithm. Rather than computing for each cell the offset to its object list, the offset to the next object list is computed. That is, $C[i]$ records the offset of the object list of the cell with 1D index $i+1$. In simpler words, $C[i]$ points to one past the end of the object list of the cell with 1D index i. This is done by:

$$C[i] = C[i] + C[i-1] \quad (where\ \ i\ =\ 0\ \ to\ \ M) \tag{5}$$

The joint size of the object list is given by $C[M-1]$. And that is the value that is used to allocate array L. Primitive indices are now inserted in array L by reversely iterating over all primitives, and for each cell overlapped by the primitive, decrementing the offset of the cell and storing the object index at that offset. This is done by:

$$L[--C[j]] = i \quad (where\ \ i\ =\ N-1\ \ to\ \ 0)\ \ for\ \ each\ \ cell\ \ j\ \ overlapped\ \ by\ \ object\ \ i \tag{6}$$

After this operation the cell array C contains the correct offsets, since each offset was decremented the appropriate number of times.

5.2.2 Traversal

The traversal algorithm used to traverse the 3D grid is an implementation of the paper **A Fast Voxel Traversal Algorithm for Ray Tracing**(Amanatides, Woo, et al., 1987). The very first step in our algorithm is to verify if the ray even enters the bounds of our scene. This is done by checking if the ray intersects the bounding box of the scene at all, which is also the part where our bounds of the grid start. If there is no intersection, we return without doing any computation. However, if the ray does intersect the scene bounding box, we have to go through a two step process: initialization of the variables and incremental traversal of the grid.

1. **Initialization**
   The initialization phase starts with identifying which voxel the ray intersects with at first. This is the voxel which we will use to incrementally go through the grid. In order to know which voxel our ray starts in, we first need to know where does the ray actually start from and where does it end. To get $\boldsymbol{r_{start}}$ and $\boldsymbol{r_{end}}$:

$$\boldsymbol{r_{start}} = \boldsymbol{r_o} + \boldsymbol{r_{dir}} * t_{min} \tag{7}$$

$$\boldsymbol{r_{end}} = \boldsymbol{r_o} + \boldsymbol{r_{dir}} * t_{max} \tag{8}$$

   t_{min} and t_{max} are the t values which indicate from where do we need to start and end checking for objects in our scene or also known as the visibility area. These are typically set to 0.001 and $infinity$ respectively. When we have both $\boldsymbol{r_{start}}$ and $\boldsymbol{r_{end}}$ we can get the first voxel that intersects with our ray by:

$$i_{start} = \lfloor \frac{(r_{start_i} - scene_{AABB_{min_i}})}{D_i} \rfloor \quad (i \in \{X,\ Y,\ Z\}) \tag{9}$$

   Here, X_{start}, Y_{start} and Z_{start} is the x, y and z coordinate of our voxel and we can easily use equation (3) from above to get the voxel from linearized array C. The above equations (9), (10) and (11) need to be clamped in between 0 and grid resolutions $M_x - 1$, $M_y - 1$ and $M_z - 1$ so we always get a value which is inside our grid. Once we have X_{start}, Y_{start} and Z_{start}, we need to compute X_{end}, Y_{end} and Z_{end} as well by using $\boldsymbol{r_{end}}$ in equation (9), (10) and (11). After this, we need 3 more variables for each coordinate after which we can start the traversal. The 3 variables are the $step$, $tDelta$ and $tMax$ values:

   (a) **step**: This is the value which decides if the step in our traversal is positive, negative or zero. Or in other words, for stepX, are we going from left to right (positive), right to left (negative) or we don't want to change this coordinate at all (zero) in our traversal of voxels. It must be obvious that this is just the slope or the direction of our ray and we can set this value by checking if r_{dir_x} is positive, negative or zero. Same process is used for stepY and stepZ.

   (b) **tDelta**: This as the paper describes it, is the value which determines how far along the ray we must move (in units of t) for a component of such a movement to equal the dimension of the voxel. Or in other words, it is the step length in units of t between grid planes. This can be calculated by dividing the cell dimension with the ray direction:

$$tDeltaX = \frac{D_x}{r_{dir_x}} \tag{10}$$

$$tDeltaY = \frac{D_y}{r_{dir_y}} \tag{11}$$

$$tDeltaZ = \frac{D_z}{r_{dir_z}} \tag{12}$$

The $tDelta$ needs to be negative when the ray direction is negative and it is t_{max} when the ray direction is 0 respectively.

(c) **tMax**: This is the value in units of t and represents at which t value, the ray crosses the first vertical voxel boundary ($tMaxX$), horizontal voxel boundary ($tMaxY$) or the boundary in z-axis ($tMaxZ$). This needs to be updated in each step of the traversal using the $tDelta$ value we computed above. The minimum of these three values indicate how far we can travel along the ray and still remain in the current voxel. This also determines if the next voxel should be in x-direction, y-direction or the z-direction.

$$tMaxX = t_{min} + \frac{(scene_{AABB_{min_x}} + (X_{start} + 1) * D_x - r_{start_x})}{r_{dir_x}} \tag{13}$$

$$tMaxY = t_{min} + \frac{(scene_{AABB_{min_y}} + (Y_{start} + 1) * D_y - r_{start_y})}{r_{dir_y}} \tag{14}$$

$$tMaxZ = t_{min} + \frac{(scene_{AABB_{min_z}} + (Z_{start} + 1) * D_z - r_{start_z})}{r_{dir_z}} \tag{15}$$

The $tMax$ needs to use the current index (X_{start}, Y_{start} and Z_{start}) when the ray direction is negative and it is t_{max} when the ray direction is 0.

2. **Traversal**
We now have all the elements to traverse the grid. A pseudocode of the traversal algorithm is given below:

Algorithm 1 Fast Voxel Traversal for Ray Tracing

while $(X_{start}, Y_{start}, Z_{start} \neq X_{end}, Y_{end}, Z_{end})$ **do**

TestVoxelForIntersections$(X_{start}, Y_{start}, Z_{start})$

if $(tMaxX < tMaxY \ \& \ tMaxX < tMaxZ)$ **then**
$tMaxX \leftarrow tMaxX + tDeltaX$
$X_{start} \leftarrow X_{start} + stepX$

else if $(tMaxY < tMaxZ)$ **then**
$tMaxY \leftarrow tMaxY + tDeltaY$
$Y_{start} \leftarrow Y_{start} + stepY$

else
$tMaxZ \leftarrow tMaxZ + tDeltaZ$
$Z_{start} \leftarrow Z_{start} + stepZ$

if $(X_{start}, Y_{start}, Z_{start} < 0$ **or** $X_{start}, Y_{start}, Z_{start} > M_x - 1, M_y - 1, M_z - 1)$ **then**
break

6 Bounding Volume Hierarchies

6.1 Concept

In order to get a nice intuition about **bounding volume hierarchies**, we first need to understand what a **bounding volume** means. The idea of **bounding volumes** in graphics, is to find enclosing volumes that are easier to test. It is quite obvious that instead of testing all triangles in a particular mesh with millions of triangles, it might be helpful to first test if the ray actually intersects a simple volume that encloses the mesh itself. There are different types of bounding volumes as shown in figure 5.

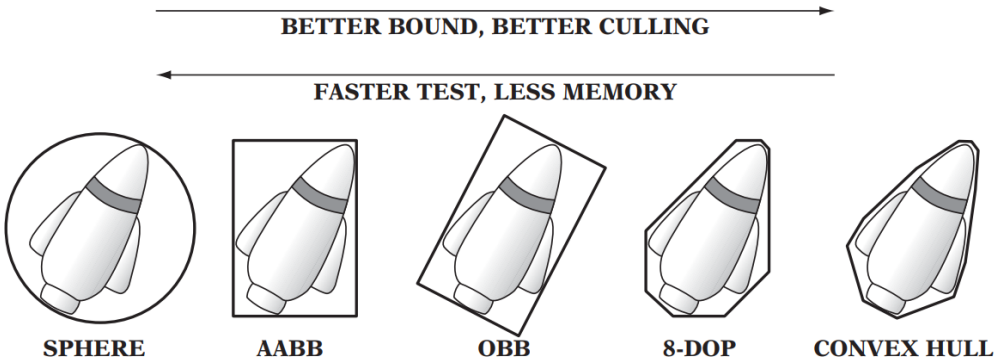


Figure 5: Different types of bounding volumes. (Ericson, 2004)

Ideally, we would like our bounding volume to be as tightly enclosed as possible but also easier to check for an intersection with a ray. A tight bounding volume would help us to have the least amount of overlaps with other bounding volumes in our scene while a fast intersection test would save up the computation time while the image is being rendered. One popular bounding volume type is **AABB** or **Axis-Aligned Bounding Box** (also shown in figure 5). This is due to the fact that they are faster to compute (linear run time) and also have a simple intersection test. AABBs are also the preferred bounding volume when building bounding volume hierarchies and so we will use the same in our implementation.

Since, now we have an intuition about what a bounding volume means, we can discuss what a hierarchy of such volume(s) would mean. The idea is to partition complex objects into a hierarchy of nodes or sets which contains only a smaller portion of the whole complex object. Looking at figure 5 (AABB) again, we can notice that we are using just one bounding volume to enclose the whole rocket model but we can surely divide the model into two parts. This can be done horizontally so that the nose of the rocket is in a separate AABB and the bottom portion with the wings and the body in a separate AABB. This can also be done vertically, separating the body and the wings in two different AABBs. Figure 6 visualizes the concept using a simpler scene.

In the figure we can see that the whole scene itself is first enclosed into one big bounding volume and then that BV encloses two other BVs. These two BVs contain the primitives themselves but those primitives are also enclosed using a simpler BV (AABB).

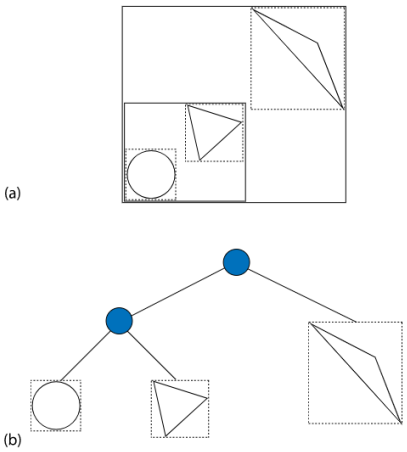


Figure 6: A hierarchy of bounding volumes of a scene consisting of 3 primitives. (Pharr et al., 2016)

The idea here is to partition complex objects into a hierarchy of simpler objects that are easier to test. If we send a ray into the scene shown in figure 6, our first test would be with the scene AABB itself, and if our ray does intersect with the scene AABB or the root node, we can then move further and test the same ray with the two AABBs in the left and right node. Now there's a chance that our ray hits the left node again where we would test the same ray again but this time only with the primitives' AABBs in the left node. And if our ray does not hit the right node, that would mean that we also do not intersect with any of the primitives in the right node or for more complex scenes, any of the further nodes in the right node. This results in a much more efficient way to test if we have an intersection with complex models in our scene. This way of dividing objects is known as **object subdivision**.

6.2 Implementation

We do have an idea of how to build the hierarchy or tree of primitives in our scene but there are still some important questions to answer. One of the most important part in a BVH is the **splitting criteria**. As we go through all the primitives in our scene, we need a criteria that decides how to partition our primitives in the left and right node. This is important because a bad criteria can result in a bad tree where our primitives in both left and right node could overlap. This would result in more ray-intersection tests when the tree is traversed and thereby resulting in a bad implementation of BVH. We will discuss 3 different splitting criterias in this report starting with the most trivial one.

6.2.1 Splitting Criterias

1. **Sorting axis:** One way to split the primitives is to sort them by picking a random axis. Once the primitives are sorted, the first half of the primitives can be placed into the left node while the second half can be placed in the right node. This ofcourse does not take into account many of the information in our scene that could be helpful but is still quite fast than not using the acceleration structure at all. Figure 7 illustrates this method for a simple scene.

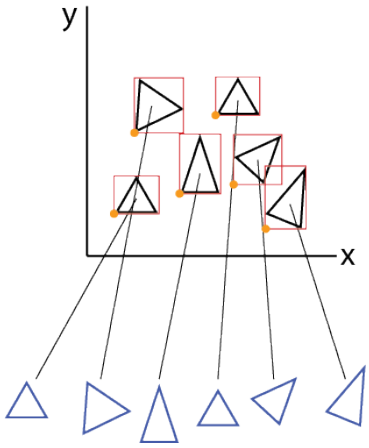


Figure 7: Sorting primitves in a scene based on x-axis.

One important rule here is to not use the same axis for sorting all the nodes in our tree as that would result in a tree biased towards a particular axis. Think of a scene where the objects are scattered more in the y-axis than the x-axis or z-axis. In that case, sorting primitives in the x-axis would result in a bad tree. One good rule is to randomly pick an axis at every node so that the tree is not biased towards a particular axis. Once we have just 1 or 2 primitives left in our node, we can initialize a leaf node and place the primitive in it.

2. **Centroid method:** Another splitting criteria which makes use of some information in our scene in order to split primitives in a node is the centroid method. The method starts with computing the centroid/midpoint of all primitives' AABBs in a node. This is done by first calculating the AABB of a primitive, and then calculating centroid $\boldsymbol{c}$ as:

$$\boldsymbol{c} = (\boldsymbol{AABB_{min}} * 0.5) + (\boldsymbol{AABB_{max}} * 0.5) \tag{1}$$

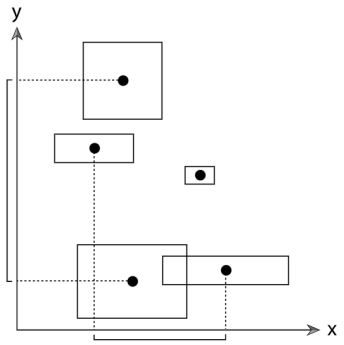


Figure 8: Splitting primitves in a scene based on centroids of their bounding volumes. (Pharr et al., 2016)

As we are computing centroids, we can get the minimum and maximum for all the centroids in a node. This minimum and maximum of all centroids is then used to compute an AABB which represents the AABB based on centroid of all the primitives in this node. Once we have this AABB, we then compute the diagonal $\boldsymbol{d}$ of this AABB as:

$$\boldsymbol{d} = \boldsymbol{AABB_{max}} - \boldsymbol{AABB_{min}} \tag{2}$$

This diagonal $\boldsymbol{d}$ can then be used to find the maximum extent in a given axis. What this means is that, we check the x, y and z value of this vector $\boldsymbol{d}$ and the one which is the highest among the three is the axis we will use to split our primitives. Intuitively, what we did is chose the axis which results in the least overlap of all AABBs in a given node.

In the example in Figure 8 above, this axis is y for that specific scene. Remember that previously we were choosing this axis randomly, so this means that with this method, we will surely end up with a better tree as we are using some of the information we have in our scene to select our axis. We still haven't decided the value which would partition our primitives. That part is quite simple. We just need to use the centroid AABB $\boldsymbol{c}$ we calculated previously, and calculate the midpoint of just the splitting axis we computed. So if the splitting axis was y we compute this split value s as:

$$s = (c_{min}[y] * 0.5) + (c_{max}[y] * 0.5) \tag{3}$$

Ofcourse, if the best axis was x or z in another node, we need to use that axis in the above equation. In Figure 8 above, this value we just computed is the midpoint of the extent in y so we end up with 2 primitives in the left node and 3 primitives in the right node. There is one slight issue with this approach. If all of the centroid points are at the same position (the centroid bounds have zero volume), we need to stop recursion and introduce a leaf node there. The condition is to check if the centroid AABB $\boldsymbol{c}$'s minimum value at the splitting axis is equal to it's maximum value at the splitting axis. For splitting axis y this is:

$$c_{min}[y] == c_{max}[y] \tag{4}$$

If the condition above is true for any splitting axis, we stop recursion and place half primitives in left node and half primtives in the right node.

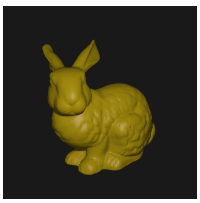
6.2.2 Traversal

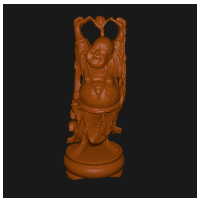
The traversal for a BVH is quite simple. We start with the root node and check if our ray intersects with the AABB of this node. If it doesn't intersect the node, we return from the function but if it does intersect with the node, we then call the function again passing the left and right nodes as parameters. We need a stopping condition for this recursion. We have already discussed one condition i.e we return $false$ if the ray does not intersect this node. One other condition is to check if we have reached a leaf node. This is done by keeping a boolean variable with each node and setting it to *true* when initializing a leaf node constructing the tree. Now if we reach a leaf node in our traversal we need to check if the ray intersects with any of the primitives in this leaf node. Unfortunately, since this is the last node in our tree it is necessary to check all primitives in this node to get the closest hit point for our ray. In the accompanied ray tracer, these hit points are saved separately and the closest one is choosen to be shaded. The array for hit points is then cleared and filled again in the next traversal. There is one condition that lets recursion stop early for shadow rays. If a ray is a shadow ray, the first hit found is enough and we can then return from the function. In order to achieve this a boolean is set to *true* specifically for shadow rays which lets the recursion stop earlier and return from the function.

7 Analysis

The data structures implemented in this report were tested on different models. Some obj variants of the scanned models from *The Stanford 3D Scanning Repository* were used. This includes *Stanford Bunny*, *Lucy*, *Stanford Dragon*, *Stanford Asian Dragon* and *Happy Buddha*. The famous *Utah teapot* is also used as well as *Suzanne* from Blender and *Spot* from Caltech.

The ray tracer as well as all the methods discussed in the report were implemented in C++. All of the methods also make use of STL. No low level optimizations were made. All results were obtained on a computer with a Quad Core AMD Ryzen 5 PRO 3500U CPU and 16GB of RAM. All images were rendered on a single core. And all images except the custom scene uses an aspect ratio of $1:1$ and a resolution of 500×500. Phong illumination model was used to shade all objects with just one point light in front of the camera. Only 1 ray per pixel was used to render all images.

	Suzanne	Spot	Teapot	Bunny
				
#tri's	968	5, 856	15, 704	69, 451
BVH_C build(s)	0.014 s	0.091 s	0.26 s	1.3 s
BVH_R build(s)	0.03 s	0.3 s	0.84 s	5.1 s
C.G build(s)	0.004 s	0.025 s	0.15 s	0.36 s
BVH_C render(s)	1.1 s	1.34 s	1.36 s	1.5 s
BVH_R render(s)	2.8 s	23.2 s	5.7 s	25.5 s
C.G render(s)	1.17 s	1.5 s	1.4 s	2.4 s
BVH_C avg # isect tests / isect ray	56.004	64.107	98.9	76.1
BVH_R avg # isect tests / isect ray	185	488	703	2100
C.G avg # isect tests / isect ray	34.31	41.088	41.35	71.6

	Lucy	Asian Dragon	Dragon	Happy Buddha
				
#tri's	99, 970	249, 999	871, 306	1, 087, 474
BVH_C build(s)	1.97 s	5.4 s	20.8 s	26.17 s
BVH_R build(s)	8.1 s	22.5 s	N/A	N/A
C.G build(s)	0.4 s	0.82 s	3.4 s	5.854 s
BVH_C render(s)	1.18 s	1.68 s	2.62 s	1.91 s
BVH_R render(s)	31.4 s	24.3 s	N/A	N/A
C.G render(s)	2.1 s	3.5 s	5.7 s	4.28 s
BVH_C avg # isect tests / isect ray	85.965	100.97	95.64	98.06
BVH_R avg # isect tests / isect ray	2728.2	3264.3	N/A	N/A
C.G avg # isect tests / isect ray	59.12	115.89	155.68	116.7

For all of the scenes we record the time it takes to construct the data structure and the time it takes to render the image. Another metric which is quite helpful is to compute the average number of intersection tests per intersecting ray. So, for every ray that intersects with the scene bounding box, we compute the number of intersection tests, sum them all together, and then take the average of it. It can be seen that both compact grids as well as the BVH centroid split method perform really well on a variety of models with different amount of triangles.

7.1 Construction time analysis

7.1.1 Why compact grid is fast to construct?

Overall, it can be seen that compact grids take quite less time to build the structure as compared to the binary tree in BVH centroid split method. This is because compact grid is constructed by first going through all primitives in the scene, check which cells they overlap to get the linearized C array so it is a loop over all primitives in the scene. Then the linearized C array is corrected by going through the same array itself. This array has approximately the size of the density ρ times the primitives in our scene. Next, we get the object list L by going through

the corrected C array once again. Once we have both these arrays, we then get the corrected L and C array by going through the primitives once again, getting the cell they overlap and then getting the final L and C array. We can ofcourse, save much of the information previously when we go through the primitives array the first time, but it would require a lot of additional memory because one triangle for example can be in more than one cell so this array which would store this information would be quite large. Overall, most of the time is spent checking for triangles that overlap but since we only have to go through them 2 times, it is still not that slow.

7.1.2 Why BVH is slow to construct?

Comparing this with the BVH centroid split method, we start with the scene bounding box which is the root node of our tree. This contains all of the primitives in our scene, so we go through them one by one, get their bounding boxes, get the centroid of those bounding boxes, and then compute the union of all those centroids to get a resultant bounding box. This resultant bounding box is then used to get the best split for the primitives in the current node and then we partition them into left and right nodes. Note that for both left and right nodes, we again want the correct bounding box that encloses the primitives in those nodes, so we again want a union of all bounding boxes of primitives in each of those nodes. This process is repeated for the left and right nodes again until we reach our base condition. So, most of the time is spent computing this resultant bounding box, which requires us to loop through all primitives in a node. This makes the BVH slow when constructing the structure. One important thing to mention here is that the better the split criteria, the better the quality of our tree, and the less the depth of our tree. So as a result, if we use a better criteria, we would be able to partition the primitives in a much better way, and this would result in a tree with less overlapping primitives in both left and right nodes of our tree. This can help in cutting down the time it takes to construct the tree in some scenarios (depending on the complexity of the splitting criteria) but would always help in cutting down the render time.

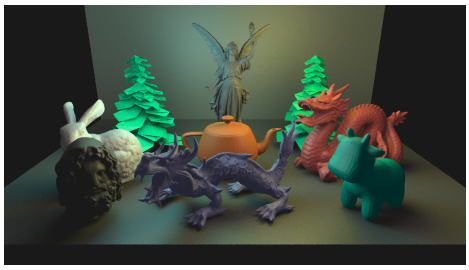
The BVH random split method performs poorly compared to other structures. This is quite obvious since we choose the split axis randomly without using any valuable information our scene can give to us. We sort the primitives and then place the first half in left node and the second half in the right node. Choosing the split axis randomly also results in random trees. In some cases, the tree is satisfactory while in other cases the tree can be much worse, increasing both the time complexity as well as the space complexity for tree construction. Resulting in a bad tree also increases the render time since then there might be a case that many of the primitives end up in both left and right nodes of our tree with wasted repeated intersection tests.

7.2 Render time analysis

Let's discuss the render time now. This is the amount of time it takes to render our image given that we have already built our data structure. It can be seen that both compact grids and BVH centroid split method perform quite nicely and render the image in similar times. One deviation to notice is the fact that the render time for compact grids is increasing slightly as the number of primitives in our scene increase. This is not true for BVH. The centroid split method renders the Happy Buddha scene in about 1.91 seconds, which is almost the same amount of time it takes for all other scenes. The compact grids however, while still very fast for less complex scenes, start to take more time to render the image for more complex scenes. The Happy Buddha scene takes about 4.28 seconds, while the Suzanne scene takes just about 1.17 seconds to render.

Let's discuss the metric we computed which is the average number of intersection tests for each intersecting ray. It is interesting to note that this metric has a direct relation to the render time. If the value is lower, this would mean that the image would also take less time to render. Higher values can result in an increased render time. So for example, the dragon scene has the value 155.68 for compact grids and the time it takes to render the scene with compact grids is about 5.7 seconds. The same scene has a value of 95.64 for BVH, and it takes about 2.62 seconds to render the image using BVH. This metric can also be used to come up with a better density ρ value for a specific scene when using compact grids. This value, as mentioned previously is used in equation (2) to get the desired grid resolution. Let's discuss this further for a more complex scene.

7.3 Further analysis

	Custom scene 1	Custom scene 2
		
#tri's	3, 362, 633 ≈ 3.36M	1, 401, 384 ≈ 1.4M
BVH_C build(s)	95.91 s	37.02 s
C.G build(s)	($\rho = 4$) 13.276 s, ($\rho = 6$) 23.271 s, ($\rho = 8$) 23.859 s	($\rho = 8$) 11.22 s
BVH_C render(s)	100.15 s	663.84 s
C.G render(s)	($\rho = 4$) 218.15 s, ($\rho = 6$) 183.961 s, ($\rho = 8$) 164.468 s	($\rho = 8$) 2076.93 s
BVH_C avg # isect tests / isect ray	320.96	79.6917
C.G avg# isect tests / isect ray	($\rho = 4$) 560.504, ($\rho = 6$) 466.015, ($\rho = 8$) 417.912	($\rho = 8$) 264.514

7.3.1 Scene 1

This scene uses an aspect ratio of 16 : 9 with a resolution of 1000 × 562. 3 rays per pixel were used to render this scene. 2 point lights, one in front of the camera with an intensity value of

30 and the other above the scene with a high intensity value of 60 were used. The scene uses random scaling for the grass so that it properly represents a natural scene. This affects the render times as well as build time slightly for different runs but not so much.

It is interesting to note that BVH outperforms compact grid in this scene. The build time is about 95 seconds and the render time is about 100 seconds with a total of about 195 seconds for time to image. Compact grids while still take less time to construct (13.276 seconds), the time taken to render the image is about 218 seconds with a total of about 231 seconds. Note that the average number of intersection tests per intersecting ray for compact grids is quite high (500.5) compared to BVH (320.96). Now we tried to increase the value ρ to get different resolution grids for compact grids and noticed that they start to get better results for increased density.

For a value of $\rho = 6$ the average number of intersection tests per intersecting ray went down to 466.015 which had a significant impact on render time as well. This time we were able to render the same image in about 184 seconds, which is about 34 seconds faster than before. We did spent some extra time (about 10 seconds more) on construction but even adding the total time to image gives us 207 seconds versus 231 seconds. Now increasing the ρ value further gives us a value of 417.912, which is not a huge difference from before. This means that we might have still improved our time but not so much, which is apparent from the values in the table. Since increasing this ρ value also increases the space complexity to keep our grid in memory, this new ρ value isn't worth it. The metric we computed can therefore be used to arrive at a better ρ value for specific scenes when using compact grids. Let us analyse one more scene, this time using area lights and having different objects of varied number of triangles scattered throughout the scene.

7.3.2 Scene 2

This scene uses an aspect ratio of $16 : 9$ with a resolution of 1000×562. 3 rays per pixel were used to render this scene. This time five area lights were used instead of point lights. All of them are basically point lights in a grid of 3×3 so a total of 45 point lights. The intensity value for a point is then averaged for each area light to compute shading. All of the lights are mostly around the center of the scene. As you can notice, the scene has different objects with varied amount of triangles; some of them are highly dense objects while others contain less number of triangles.

Compact grids perform very poorly in this scene. As obvious from the table, the construction time was quite low (about 11.22 seconds) but rendering this scene took about 2077 seconds or about 35 minutes. The reason behind this significant impact even though the scene contains almost half of the number of triangles in scene 1 is because this scene has a **non uniform primitive distribution**. This causes uniform grids to suffer from a problem known as **teapot in a stadium**. What this means is that, if all objects in the scene have approximately the same size, then uniform grids have generally good results, but if there are objects which are very small and some objects that are very large, then many of the triangles of small objects end up in many cells of the grid which results in greater number of intersection tests in some cells resulting in poor performance.

BVH however, do not suffer from this problem since it divides objects (object subdivision) instead of space (space subdivision). This is the reason why BVH was able to render the same scene in about 663 seconds or about 11 minutes.

References

Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics* (Vol. 87, pp. 3–10).

Devillers, O. (1988). *Méthodes d'optimisation du tracé de rayons* (Unpublished doctoral dissertation). Université Paris Sud-Paris XI.

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

Fujimoto, A., Tanaka, T., & Iwata, K. (1986). Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, *6*(4), 16–26.

Lagae, A., & Dutré, P. (2008). Compact, fast and robust grids for ray tracing. In *Computer graphics forum* (Vol. 27, pp. 1235–1244).

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

Shirley, P. (2020a, December). *Ray tracing: The next week.* Retrieved from `https://raytracing.github.io/books/RayTracingTheNextWeek.html` (`https://raytracing.github.io/books/RayTracingTheNextWeek.html`)

Shirley, P. (2020b, December). *Ray tracing: The rest of your life.* Retrieved from `https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html` (`https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html`)