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:

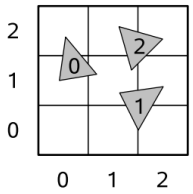

Figure 1: A grid with three triangles overlapping the cells. (Lagae & Dutré, 2008)

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:

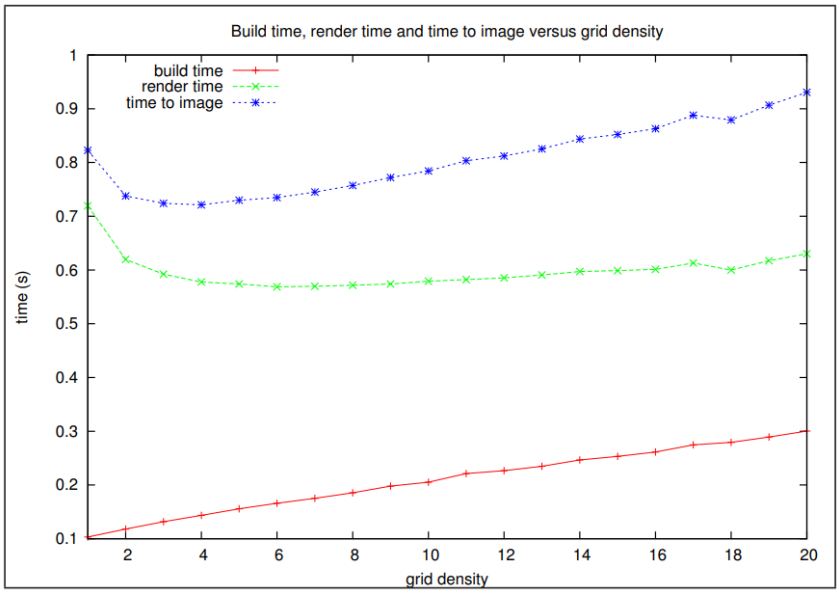

Figure 2: Grid density. Build time (red), render time (green) and time to image (blue) versus grid density for the Happy Buddha scene. A grid density of 4 minimizes the time to image. (Lagae & Dutré, 2008)

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.

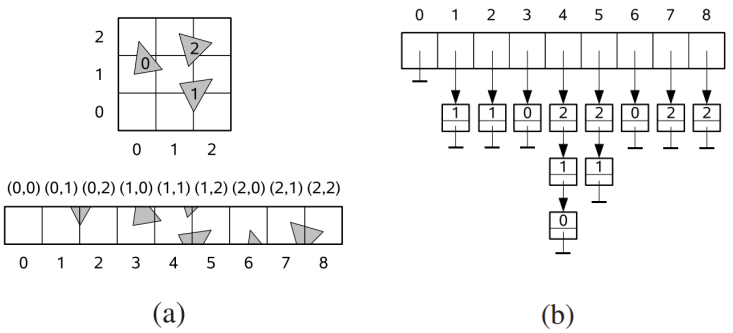

Figure 3: a) A grid with three triangles overlapping the cells and the linearized 1D version of the grid. b) A traditional grid data structure using linked lists. (Lagae & Dutré, 2008)

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).

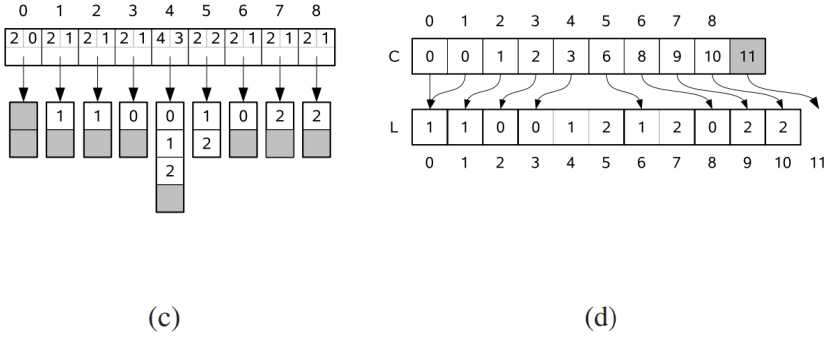

Figure 4: c) A traditional grid data structure using dynamic arrays. d) The compact grid data structure implemented in this report. (Lagae & Dutré, 2008)

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

5.3 Analysis

Now that we have a uniform grids implementation, we can compare it with the naive method for ray tracing i.e for every ray sent into the scene, we go through all objects in the scene and test if that ray intersects with any of the objects. The ray tracer used to test both the implementations was already implemented in a previous lab project. The ray tracer implements ray-sphere and ray-triangle intersection methods with support to add triangular meshes. For shading, it implements **Phong illumination** to support diffuse, ambient and specular highlights and adds texture mapping for spheres and triangles. Anti-aliasing is also supported and the number of samples can be changed according to user's needs. It also supports axis-aligned bounding boxes for meshes to slightly speed up ray-mesh intersection tests. Area lights were also implemented by reusing point light implementation.

We have 6 test scenes to test our implementation of data structures. All of the scenes have a similar setting. They are all illuminated with 3 point light sources with the camera being in the same position, 3x anti-aliasing is used to render all scenes. Different meshes are used for each of the scenes but they all lie on a plane with one plane as a background for better visualization. The scenes used are:

1. **Scene 1** which consists of a cube in the middle of the scene with 12 triangles with a total of 16 triangles in the scene.
2. **Scene 2** which consists of blender monkey suzanne (OpenGLInsights, 2012) which consists of 968 triangles with a total of 972 triangles in the scene.
3. **Scene 3** which consists of 5 pine trees (Archive, 2017) each consisting of 527 triangles with a total of 2, 639 triangles in the scene.
4. **Scene 4** which uses stanford bunny (Stanford, 2021) with 4, 968 triangles with a total of 4, 972 triangles in the scene.
5. **Scene 5** which consists of utah teapot (Wikipedia, 2021) with 6, 320 triangles with a total of 6, 324 triangles in the scene.
6. **Scene 6** which consists of 4 pine trees and the utah teapot with a total of 8, 432 triangles.

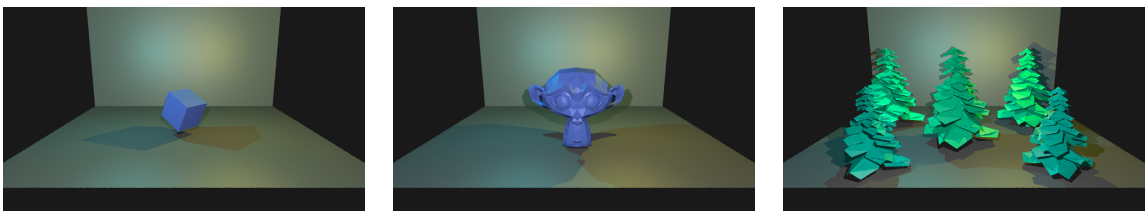

Figure 5: Scene 1, Scene 2 and Scene 3 from left to right.

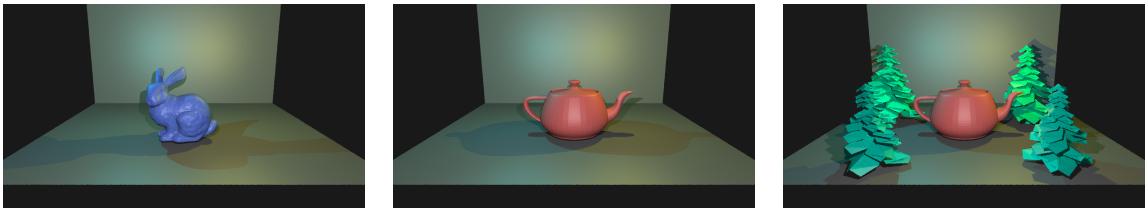

Figure 6: Scene 4, Scene 5 and Scene 6 from left to right.

Scene 1		
Type	Ray-triangle intersection tests	Time to render
Brute Force	23, 722, 309	5.382 seconds
Uniform Grid	1, 380, 343	5.95 seconds

Scene 2		
Type	Ray-triangle intersection tests	Time to render
Brute Force	2, 597, 291, 856	215.412 seconds
Uniform Grid	4, 709, 496	15.245 seconds

Scene 3		
Type	Ray-triangle intersection tests	Time to render
Brute Force	5, 082, 250, 516	413.224 seconds
Uniform Grid	3, 648, 710	14.44 seconds

Scene 4		
Type	Ray-triangle intersection tests	Time to render
Brute Force	9, 319, 100, 223	876.264 seconds
Uniform Grid	2, 618, 474	18.748 seconds

Scene 5		
Type	Ray-triangle intersection tests	Time to render
Brute Force	17, 926, 727, 145	1532.55 seconds
Uniform Grid	3, 876, 189	18.072 seconds

Scene 6		
Type	Ray-triangle intersection tests	Time to render
Brute Force	16, 289, 624, 734	1369.32 seconds
Uniform Grid	3, 205, 779	14.2628 seconds

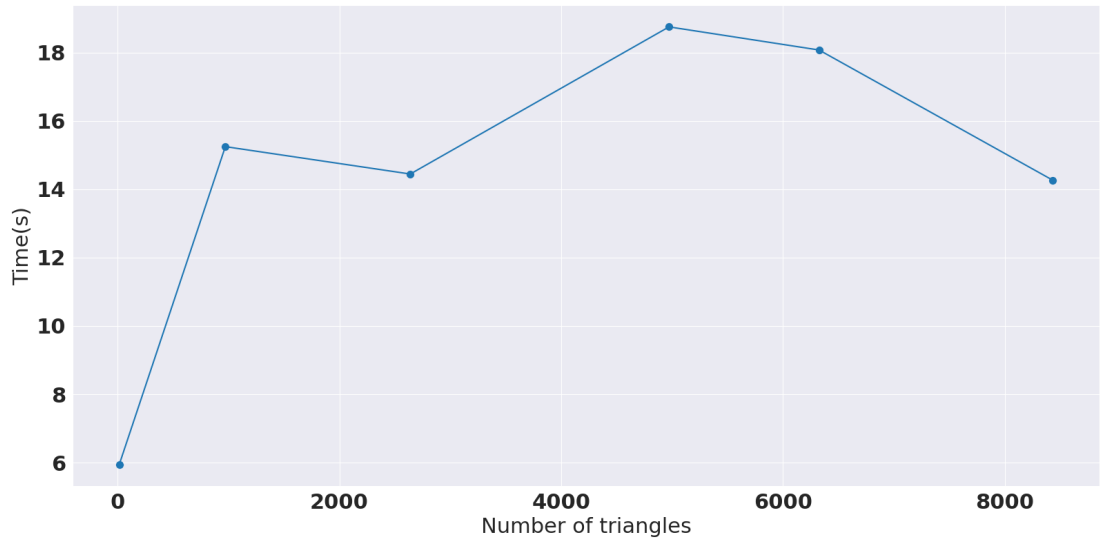

Figure 7: (Uniform Grid) A graph with number of triangles in the x-axis and time taken to render the scene in y-axis.

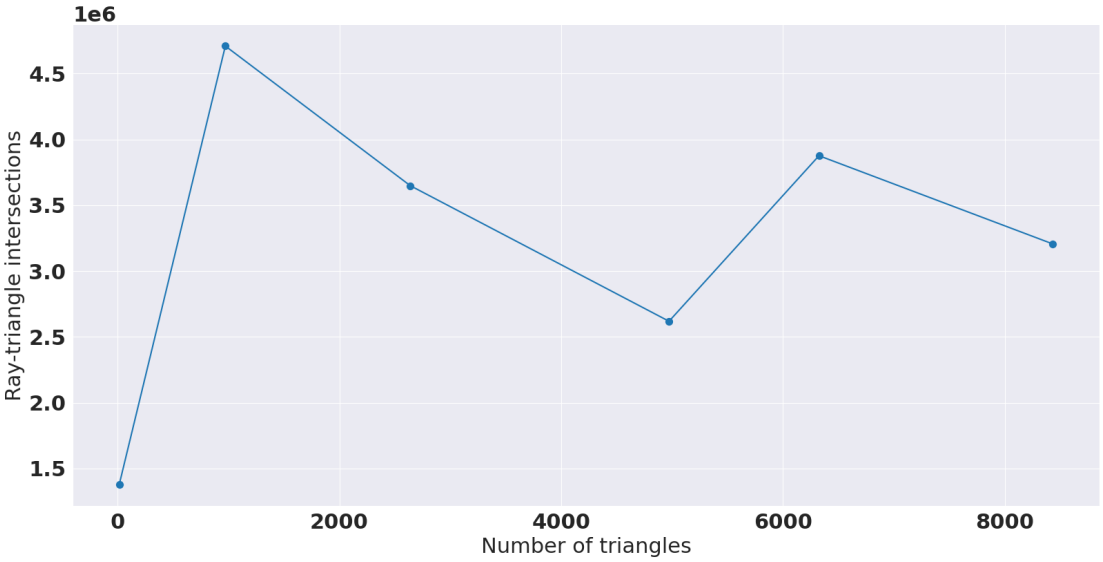

Figure 8: (Uniform Grid) A graph with number of triangles in the x-axis and ray-triangle intersections in y-axis.

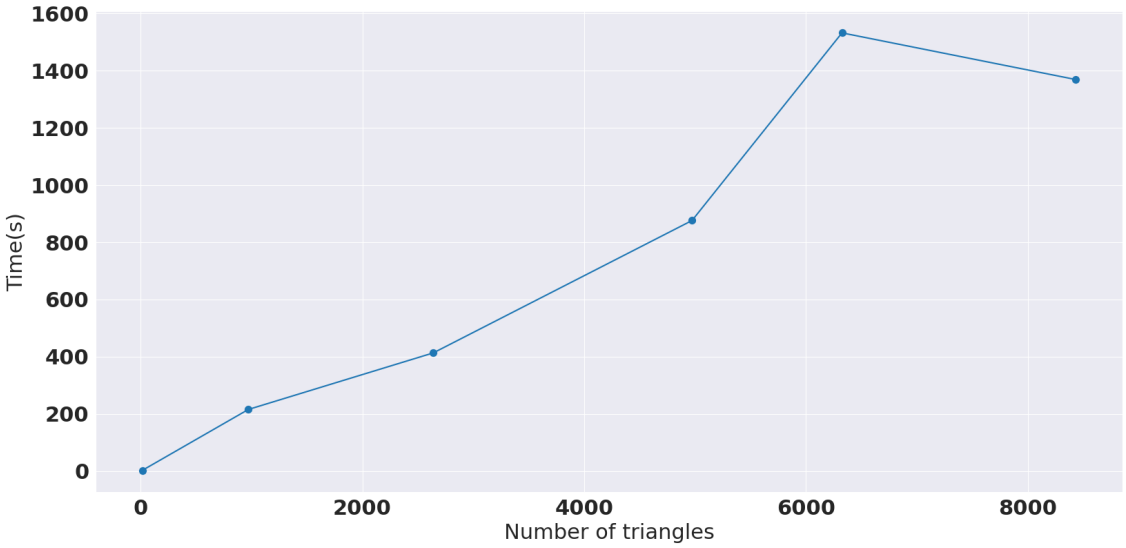

Figure 9: (Brute Force) A graph with number of triangles in the x-axis and time taken to render the scene in y-axis.

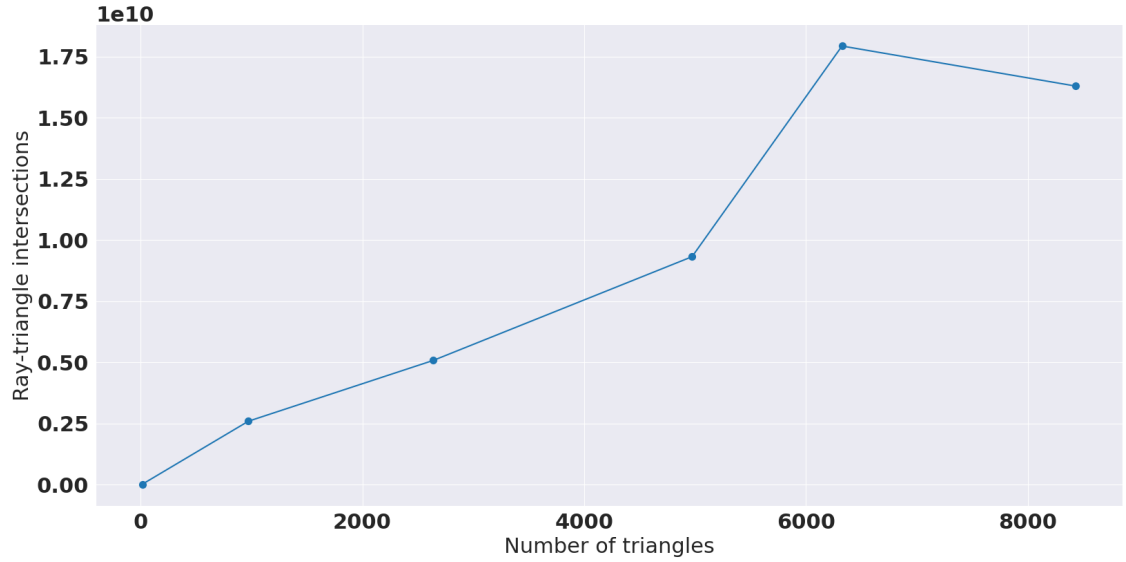

Figure 10: (Brute Force) A graph with number of triangles in the x-axis and ray-triangle intersections in y-axis.

References

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

Archive, M. C. G. (2017). *Pine tree.* Retrieved from `https://casual-effects.com/data/`

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

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).

OpenGLInsights. (2012). *Blender monkey.* Retrieved from `https://github.com/OpenGLInsights/OpenGLInsightsCode/blob/master/Chapter%2026%20Indexing%20Multiple%20Vertex%20Arrays/article/suzanne.obj`

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`)

Stanford. (2021). *Stanford bunny.* Retrieved from `http://graphics.stanford.edu/data/3Dscanrep/`

Wikipedia. (2021). *Article on utah teapot.* Retrieved from `https://en.wikipedia.org/wiki/Utah_teapot`