Simple Real-Time Ray Tracing using Octrees in OpenGL

Tiago Cruz
up202108810@up.pt

Tomás Xavier
up202108759@up.pt

I. Introduction

In this project, we set out to implement a simple real-time ray tracer that uses the **Octree data structure** to **optimize scene traversal** during **collision detection**. Our implementation uses OpenGL. This choice was made due to our familiarity with WebGL from previous courses (SGI), which shares many similarities.

However, using OpenGL for real-time ray tracing presented several challenges, as it is **designed for rasterization rather than ray tracing**. This forced us to consider three potential approaches: **(1)** abandoning OpenGL altogether, **(2)** implementing the entire ray tracer within GPU shaders, or **(3)** splitting the workload, using C++ on the CPU for static scene setup and GPU shaders for the dynamic ray traversal and intersection logic.

We ultimately chose the third option, as it provided a **balanced trade-off** between performance, flexibility, and leveraging our existing knowledge.

Due to the nature of this course, the implementation of the ray tracing algorithm itself was not our primary focus. To facilitate our efforts, we chose to **adapt an existing ray tracer** and integrate it with our custom Octree implementation.

As such, this report will not focus on any Computer Graphics development. Instead, we **focus on the Octree data structure**, its **integration** with the ray tracing pipeline, and **how it improves performance**, aligning with the objectives of the course.

II. Implementation

Diving into our implementation, we can subdivide it into 4 parts, **(1)** the process of building the Octree on the CPU, **(2)** flattening the Octree data structure into vectors to facilitate the conversion into the GPU, **(3)** structuring the data so it can be actually passed onto the GPU and finally **(4)** the Octree traversal and collision detection on the GPU.

A. Building the Octree

a) *Preparing the Scene:*

Before constructing the Octree, we first **generate the objects** to be included within it. To maintain the project's focus on data structures rather than computer graphics, we chose to work exclusively with **spheres**.

To add some complexity and diversity to our scene we generated *N* spheres at random positions within x,y and z and used 3 different materials, Lambert, Metal and Dielectric, which can be seen in Fig. 1

Fig. 1: The 3 used materials. From left to right: Lambert, Metal, Dielectric

Fig. 2: Example of the random scene with 100 spheres

b) *Creating the Octree:*

To create the Octree we started with a root node that covers the entire scene. Now, given a **Max Octree Depth** and **Min Spheres per Node** we subdivide recursively our nodes.

- **Max Octree Depth**: How many recursive calls/subdivisions our Octree structure is allowed to make at maximum
- **Min Spheres per Node**: Limit on how few spheres a node can have until it is forced to stop subdividing. A node with 0 spheres isn't affected by this as they won't be subdivided, but if we have Min Spheres per Node at 2 and a node has 2 spheres, even if it isn't at the max depth it won't subdivide further.

To **minimize space**, the Octree only makes a subdivision on a node if it has spheres, but when it does it:

1) Creates 8 children.
2) Checks for intersections between the spheres in the parent and the new children. If they intersect add them to the child.
3) If the children have spheres, try to subdivide them again.

c) *Child Creation Optimization:*

In order to facilitate traversal later on, we define a child creation order that is always followed. The order of priorities

is: Bottom - Left - Back. We can see this in the code snippet in Listing 1

```
enum OctantPosition {
  // Binary: zxy (0=min, 1=max for each
dimension)
  BottomLeftBack  = 0,  // 000: (min.z,
min.x, min.y)
  BottomLeftFront  = 1,  // 001: (min.z,
min.x, max.y)
  BottomRightBack  = 2,  // 010: (min.z,
max.x, min.y)
  BottomRightFront = 3,  // 011: (min.z,
max.x, max.y)
  TopLeftBack    = 4,  // 100: (max.z,
min.x, min.y)
  TopLeftFront    = 5,  // 101: (max.z,
min.x, max.y)
  TopRightBack    = 6,  // 110: (max.z,
max.x, min.y)
  TopRightFront   = 7  // 111: (max.z,
max.x, max.y)
 };
```

Listing 1: Code Snippet of Child Creation Order

B. Flatten the Octree

a) *Theory:*

After building the Octree we are left with the following **problem**:

- Octree is built using **pointers** and Shaders **don't allow it**.
  - Class OctreeNode has field OctreeNode* children[8]

To fix this particular problem we converted our OctreeNode class into GPUOctreeNode, which **doesn't store pointers** but stores **index values** for a vector of nodes, which we will have to create by flattening the Octree.

We will have to apply the **same logic for the spheres** as now each GPUNode will need to store **indexes for their spheres**. But since this is a value that can change we will create a fixed vector (objectIndices) with all the object indices from each node, then each GPUNode will have 2 attributes:

- **ObjectOffset**: The offset of the first sphere(object) in the objectIndices vector.
- **ObjectCount**: How many spheres(objects) the GPUNode has.

For Example:

- Given GPUNode *n* that has ObjectOffset = 2 and ObjectCount = 2
- Given vector objectIndices = [0,3,1,4,5,2,0,5,3]

We know that GPUNode *n*'s objects are 1 and 4, which represent 2 objects starting from offset 2.

b) *Practise:*

To implement this solution we will:

1) Do a **BFS-type traversal** through the nodes in our Octree. We will **store all nodes in a single vector** and **save the index** in a map, mapping the node to its index.
2) Iterate through our new Octree vector and:
   - Each **OctreeNode will be translated into a GPUOctreeNode** (see: Listing 2) and the **final result** of this iteration will be the **final flattened tree**.
   - In case of a **Non-Leaf** node:
     - Save in childrenOffset the 1st child location in the vector, using the map for quick access to the index.
   - In case of a **Leaf node** with objects:
     - Insert in the shared objectIndices vector the local object indices of the node.
     - Save in objectOffset the offset of their first object in the objectIndices vector.
   - When a value isn't used, for example childrenOffset for Leaf nodes, it defaults to −1.

```
struct GPUOctreeNode {
 glm::vec3 min; // Bottom Left Back
 glm::vec3 max; // Top Right Front
 int childrenOffset;
 int objectsOffset;
 int objectCount;
};
```

Listing 2: Code Snippet of GPUOctreeNode

C. Send the Flattened Data

Now that we have our flattened tree with the same structure of nodes that will be used in the shaders we need to actually **pass this information onto the Shaders**.

But, there is still a problem before doing this, the shaders accept buffers with values such as int, floats, vec4, …, not structures, objects, etc.

What we will have to do is **create many vectors of vec4**, where **each index** in the vector **corresponds to certain info** of a Sphere or Node. This can be seen in Listing 3.

```
 std::vector<glm::vec4>
sphereCentersAndRadii;    // center.xyz,
radius
 std::vector<glm::vec4>
sphereMaterialsAndAlbedo;  //
materialType, albedo.xyz
 std::vector<glm::vec4>
sphereFuzzAndRI;       // fuzz,
refractionIndex, 0, 0

 std::vector<glm::vec4>
octreeMinAndChildren;    // min.xyz,
childrenOffset
 std::vector<glm::vec4>
octreeMaxAndObjects;     // max.xyz,
objectsOffset
 std::vector<int>
octreeObjectCounts;         // objectCount
```

Listing 3: Code Snippet of vectors to pass onto the Shaders

Now we will **iterate through the flattened Octree and the spheres vector** and **fill these vectors**.

Then, in the Shaders we can **easily access every value we need** to calculate the ray-sphere intersection by accessing all the vectors using the index we want.

For example, if we need the childrenOffset of Octree node 5, we can just access the correct vector with index (octreeMinAndChildren[5].w – w because we only want the offset, not min).

D. Ray Intersection & Octree Traverse

Base Implementation:

- For each ray (Number of Samples), while the ray is alive, we **check for intersections** with the Octree up until Max Rays Depth times.
- If there is an intersection with the root node bounding box, then start a **DFS** along the Octree.
- Check if ray **intersects the bounding box** of the selected node, if so, add it to the iteration stack, otherwise ignore node.
- If on a Leaf Node, **check it's child objects for collisions** and **keep track of the collision closest to the ray origin**.
- After emptying the iteration stack, **only consider the intersection closest to the camera**, calculate the reflection/refraction, add 1 to Ray depth, and refer back to the first step.

This basis is good, and in theory can yield better results in much more complex scenes than the naive approach.

Problem is – the added **overhead** of any Octree implementation means that to see these improvements we would need to render a scene that is **so complex that we get very little frames per second**.

Yes, the Octree method would be better but it wouldn't be a very pleasant viewing experience. Thus we formulated a **heuristic** to **speed up** the Octree so that we can see **better results** than a naive approach in simpler scenes:

- Given that **there is not any overlap between node's bounding boxes**, we can assume that are **26 different ray directions** - different combinations of the ray vector's x,y,z each being being >0, =0, or <0.
- Fig. 3 represents these different directions (the arrows and the two colored circles). **For each direction** there is a **set of optimal Octree child visiting orders**, that guarantees that we check each node **from closest to farthest** from the node origin.
- Knowing this, in theory, the **first node that contains a ray-object intersection** is the node with the **closest intersection**, so we only need to check between those node's objects for the closest intersection. We can then **discard the rest of the DFS**, but still adding 1 to ray depth and returning to step 1.
- We don't need to make 26 different execution paths as the optimal ordering sets overlap a lot so we can reduce this to **8 different sets** (each corresponding to a particular child node of the root Octree). There isn't a particularly optimal way of dividing each ray direction between the 8 sets, but the division we used is the one represented by the ray direction in Fig. 3. For example, in the red ray directions case we iterate through the Octree nodes in the order: 3,1,2,0,7,5,6,4.
- This heuristic we developed is **extremely efficient**, we got **much better performance** across the board as we are able to **terminate the DFS much earlier** than the base Octree Traversal. We will touch on this performance gain against a more naive approach on the next chapter.
- This heuristic isn't perfect though and can cause some **occasional visual glitches**.

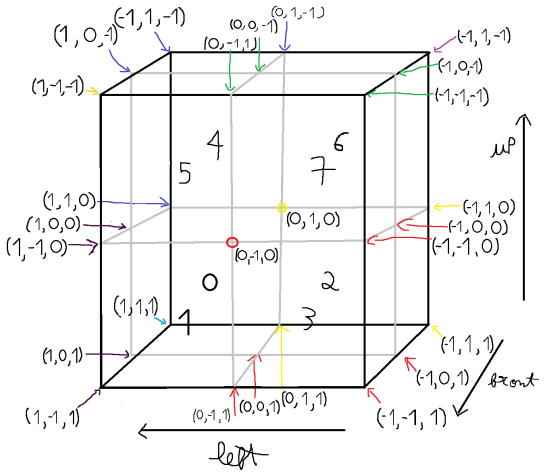

Fig. 3: Visual Representation of the different ray directions

III. Assessment

To assess our implementation, we created a python script that updates the values, compiles, runs for **50 frames** and saves all values.

- The 50 frame count and value saving is handled by the RayTracer itself.
- Outliers are eliminated based on standard deviation and z-score to avoid storing frames where the Shader was still initializing.
- Initial frames are skipped so that if the initialization process takes longer, they aren't counted towards our result.

We created a Jupyter notebook using data analysis libraries to study our results.

As there isn't a baseline approach to this current problem, we assumed naive to be: For every ray, test for intersections with every sphere We tested with **different quality levels**:

- **Low Quality** (Num Samples 4, Rays Depth 4) Fig. 4
- **Standard Quality** (16, 8) Fig. 5
- **High Quality** (32,16) Fig. 6

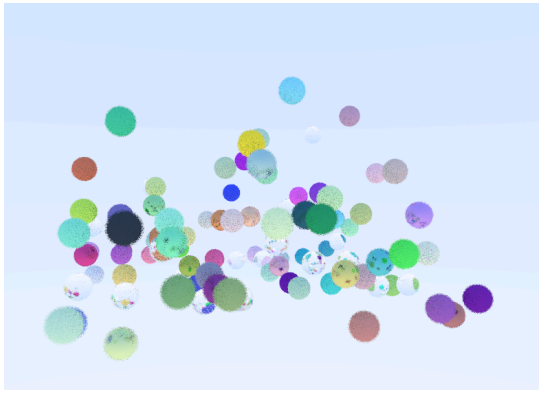

Fig. 4: Low Quality

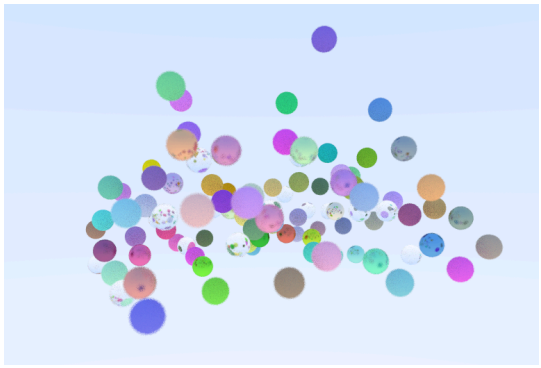

Fig. 5: Standard Quality

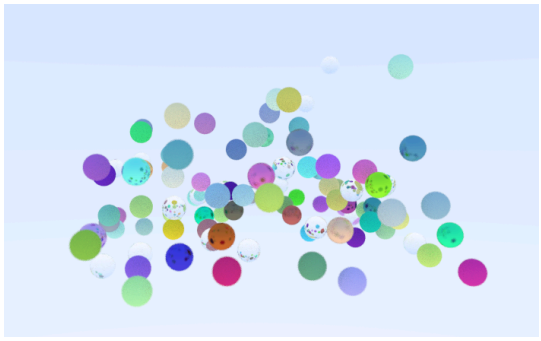

Fig. 6: High Quality

A. *Naive vs Octree*

In this section we will compare our Octree approach with the previously mentioned naive one. We will also test with different amount of Spheres in scene and different Resolutions but fixed Min Spheres Per Node at 1.

a) *Most impactful variable:*

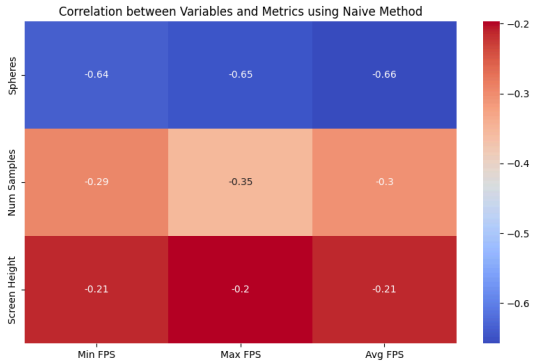

Fig. 7: Naive heat map

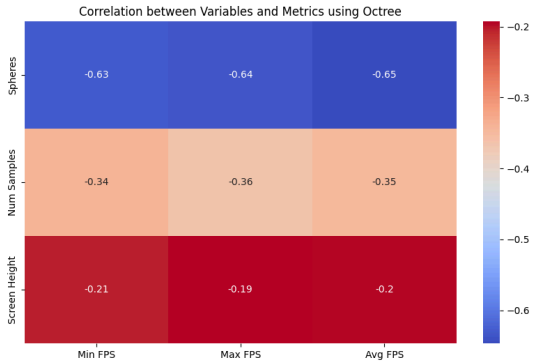

Fig. 8: Octree heat map

First, making a heat map per implementation of the FPS, depending on the Screen Height, Num Samples and spheres reveals that the former is the one with the biggest impact in performance (Fig. 7, Fig. 8)

b) *Number of Spheres:*

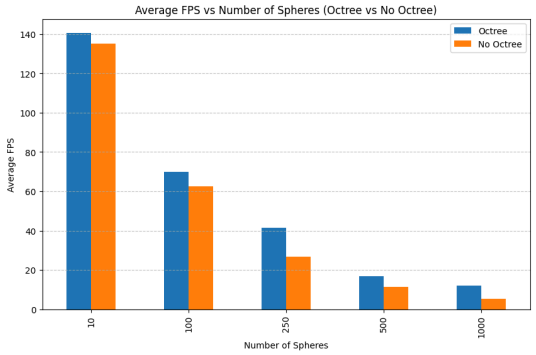

Fig. 9: Performance comparison dependent on Number of Spheres

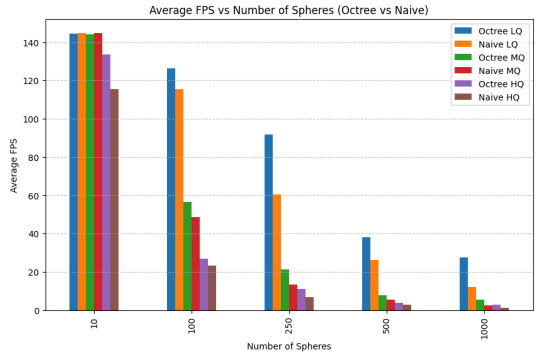

Fig. 10: Performance comparison dependent on Number of Spheres but grouped by quality levels

- In Fig. 9 and Fig. 10 we notice that **our Octree is in general always better than the Naive implementation**, but if we verify only the low quality and medium quality versions for 10 spheres, the naive version is actually slightly better. This is due to the added **overhead** of the Octree.

Larger Scene with LQ:

- We then measured much more complex scenes, but we had to only look at scenes in LQ and in a resolution of 800x600 so that the performance was measurable.

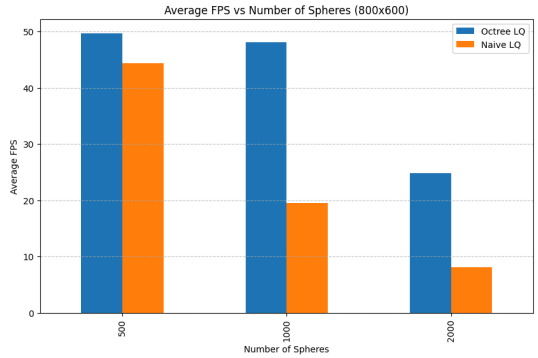

Fig. 11: Performance comparison dependent on Number of Spheres

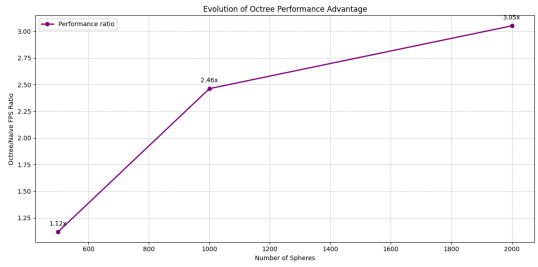

Fig. 12: Performance advantage dependent on Number of Spheres

- In Fig. 11 and Fig. 12 we can infer that the Octree provides a **bigger performance advantage the bigger the scene is**.

c) *Screen Height:*

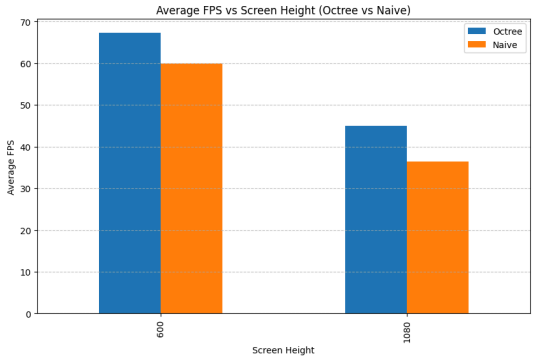

Fig. 13: Performance comparison dependent on screen resolution

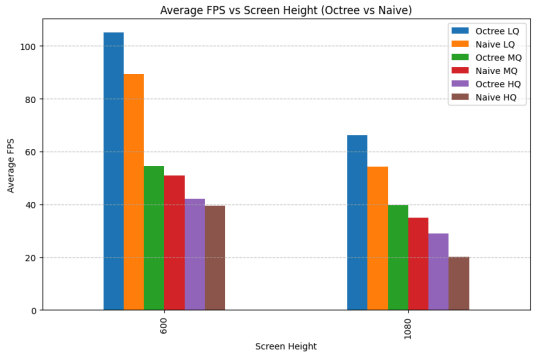

Fig. 14: Performance comparison dependent on screen resolution but grouped by quality levels

In Fig. 13 and Fig. 14 we see that our Octree implementation keeps performing better than the Naive approach.

B. Different Octree Depths and Minimum Spheres per Node

We will now compare different versions of our implementation. To do this, **we will vary between different values of Max Octree Depth and Min Spheres per Node**. We will always test using the same quality level and resolution in order to standardize, but use a varying amount of spheres.

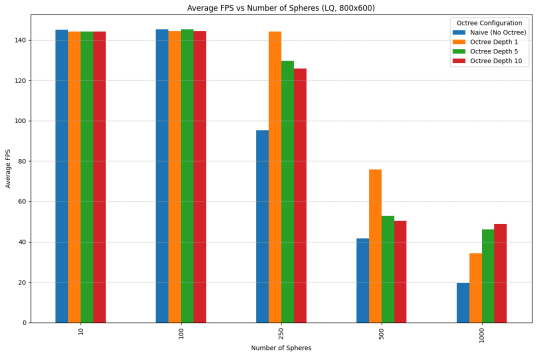

Fig. 15: Average FPS vs Number of Spheres (LQ, 800x600)

In Fig. 15 we can see that first, for a small number of spheres and with a limited value for max FPS, **small depths** (for example Depth 1, where the scene only has 9 Octree Nodes – the Root and it's 8 subdivisions) are **better for small scenes**, while when we start adding **more complexity more depth is**

needed, as we can see by the change of best depth at 1 with 500 spheres and best depth at 10 for 1000 spheres (Fig. 15).

Now to verify if bigger values of depth are necessarily better, even if the Min Spheres per Node remains unchanged, we will test for depth depending on the number of spheres, using half, same and double. This can be seen in Fig. 16, where we reach the conclusion that **if we do not adapt the Min Sphere per Node value we won't notice any major change** besides some small FPS variations due to scene randomness.

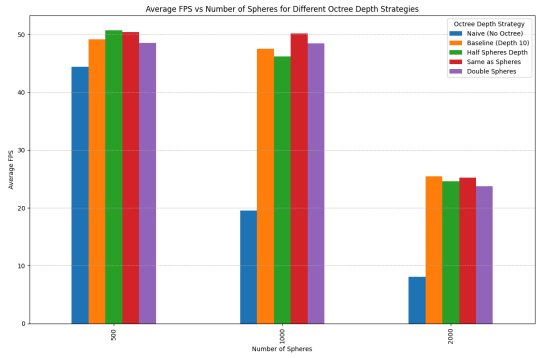

Fig. 16: Average FPS vs Number of Spheres (LQ, 800x600) – Depth based on Sphere amount

Now we will test without FPS limit, small values for Max Octree Depth (0-5) and for each depth, test with 0 and 1 as the value for Minimum Spheres per Node. The result can be seen in Fig. 17. Since we will test up until 2000 spheres, Max Octree Depth 4 should be more than enough since it can have 8^4 = 4096 leaf nodes, even if we assume that it will redistribute the spheres evenly among the nodes.

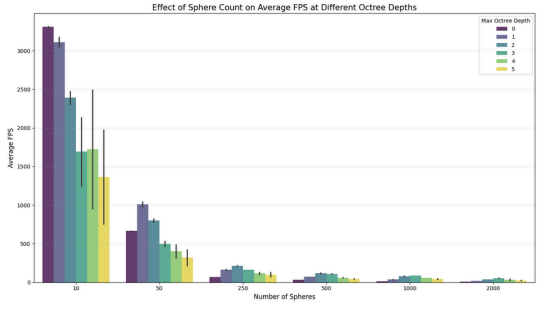

Fig. 17: Average FPS vs Number of Spheres (LQ, 800x600) – Small Depth, No FPS Limit, Using both 0 and 1 in Min Spheres per Node

In Fig. 17 there is a line in the middle of each bar. This displays the difference between 0 and 1 Min Spheres per Node. By verifying our data we verify that the **better value is 1**. This will be seen in more detail later.

We can also note a huge reduction from 10 to 50 spheres, but every FPS is over 144, so it's **barely noticeable for the human eye**, therefore this isn't of much relevance performance-wise.

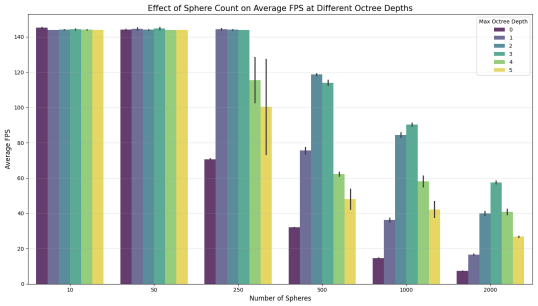

Fig. 18: Average FPS vs Number of Spheres (LQ, 800x600) – Small Depth, 144 FPS Limit, Using both 0 and 1 in Min Spheres per Node

In Fig. 18 we can see that for low sphere counts any value of Max Octree Depth does equally good, but on bigger scenes we see that **Depth 0**, which is as bad as a naive solution as it only filters out rays that go towards the sky, **has a fast decrease in performance**. We can see this difference in performance better in Fig. 19, where the best ratio starts at depth 1 in 250 spheres, goes to 2 in 500, goes to 3 at 1000 and is evenly more noticeable at 3 again at 2000 spheres.

Also we can note that **as the amount of spheres increases, the best value for Octree depth also increases**. This is probably due to the fact that a deeper Octree allows for **more effective spatial partitioning** in scenes with a higher number of objects.

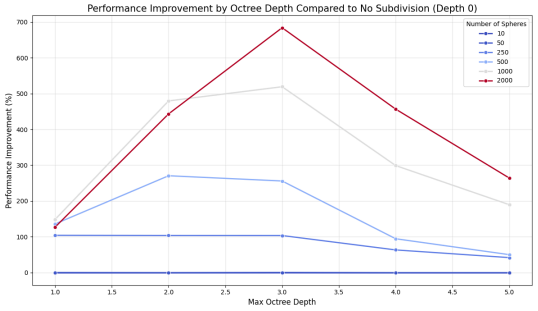

Fig. 19: Difference Ratio between >0 Max Octree Depth and 0, based on values in Fig. 18

We can look more in depth onto the difference between Min Spheres per Node in Fig. 20, where we can see that for **small scenes there isn't much of a noticeable difference**.

With a medium-sized scene (250-1000) we can see that **Min Spheres per Node 1 is better**. This is due to the fact when we don't limit the variable, the Octree is allowed to subdivide more freely, which results in **more leaf-nodes** we will have to traverse, sometimes many leaf-nodes sharing parts of a single sphere.

For our biggest scene (2000 spheres) we notice that the values are basically **the same**. This we could attribute to the maximum amount of Octree Depth, which doesn't allow Min Spheres per Node 0 to further subdivide beyond the leaf-node limit.

Although **not limiting may give better spatial partitioning**, as the algorithm has more freedom, limiting to 1 sphere per node gives the algorithm a **simplification** that allows it to **improve performance**.

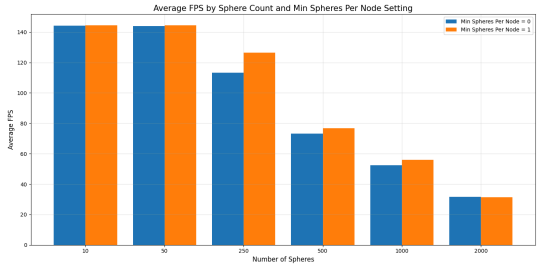

Fig. 20: Comparing Average FPS vs Sphere Count with different Min Spheres per Node (0 and 1)

Finally we can verify that **there isn't a big correlation between Octree Depth and Quality level**, as for different quality levels, despite the different FPS values, they follow the same trend of a bigger value at depth 1 and a similar decreasing factor up until depth 5 and staying about the same until depth 10, as seen in Fig. 21.

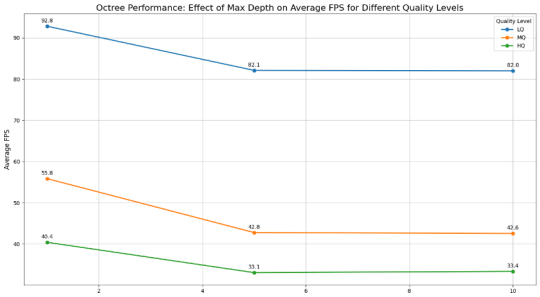

Fig. 21: Effect of Max Depth on Average FPS for Different Quality Levels

IV. Conclusions

Based on our performance analysis and visual inspection, we reached the following conclusions:

- We noticed an improvement across the board when comparing our implementation with the naive solution with bigger scenes in particular showing stronger improvements.
- In small scenes Octree causes some expected overhead and we felt that, due to OpenGL's limitations, we weren't able to reach the full potential of our implementation.
- Bigger clusters of spheres per node are better for smaller scenes.