CS267 Applications of Parallel Computers

CUDA-based Spatial Hierarchical Data Structure for Computer Graphics Acceleration

Team 26

Zixi Cai, Hsin-Pei Lee

Gang Yao, Chuan-Jiun Yang

Abstract

A number of methods for constructing bounding volume hierarchies on the GPU are inherently sequential, usually one level at a time, since every node depends on its ancestor. Karras et al. proposed a novel parallel radix tree construction algorithm that avoids sequential bottlenecks by establishing a connection between node indices and keys. This way, the overall performance scales linearly with the number of available cores. Based on the algorithm of Karras et al., we implement a CUDA-based parallel BVH library and benchmark the performance on ray tracing and particle interaction applications.

Contents

1. Introduction

In modern computer graphics applications, spatial hierarchical data structures such as kd-trees and bounding volume hierarchies (BVHs) are widely used to accelerate calculation. Particularly, it is almost a requirement to take advantage of them to implement an efficient ray tracing-based rendering pipeline. Kd-trees offer high ray tracing performance, but are costly to build; BVHs offer a compromise between performance and the ability to handle complex scenes and secondary rays. These spatial hierarchical data structures also have a wide range of use cases like collision detection, particle simulation, surface reconstruction, and voxel-based global illumination.

Before long, the common practice of interactive ray tracing was to build the acceleration data structures like BVHs with CPU before launching any graphics pipeline on the GPU. However, while this may be sufficient for rendering a single frame, most real-time applications cannot afford to communicate frequently between the CPU and GPU. Thus, constructing these data structures dynamically on the fly and running them in parallel on GPU should be the preferred method of how such applications.

For some dynamic applications, updating only the bounding box when primitives move can result in a degradation of the quality of the BVH. For ray-tracing, this will result in a dramatic deterioration of rendering performance. The typical method to avoid this degradation is to rebuild the BVH when a heuristic determines the tree is no longer efficient. However, the heuristic not being properly defined can result in a disruption of the interactive system response [1]. We came across this classic paper by Tero Karras [2]. This parallel construction method claims to achieve efficiency improvement of 65% compared to that of Garanzha [3]. In that case, reconstruct the BVH completely in parallel will not be a big cost and should be the preferred method when implementing applications with high real-time demand.

Given an unsorted list of geometric primitives, constructing the BVH sequentially typically takes O(NlogN) time and O(N) space. Lauterbach et al. were the first to present a parallel method for constructing so-called linear BVHs [4], which was later improved by Pantaleoni and Luebke [5], as well as Garanzha et al. [3].

In this project, we based on the algorithm of Karras et al. [2] and implement the parallel version of BVH. We achieve parallelism across nodes and levels and cut down time-complexity to O(Nlog(h)), where h is the depth of the tree. We focus primarily on implementing the BVH data structures on CUDA. After that, we benchmarked its performance with a physically-based rendering pipeline. Then by extending the base data structure for a parallel quadtree, we are able to apply it to a non-uniform N-body simulation where the efficiency depend heavily on the construction speed.

2. Implementation

This section discusses in detail our implementation of parallel BVH. First, we go over the serial and parallel algorithm as well as the underlying concepts. Then, we introduce the data structures.

2.1 Parallel Construction of BVH

Traditional tree construction algorithms are inherently serial, typically requires recursion to construct all the nodes, meaning every child node's information is dependent on knowing about its parent.

Algorithm 1: Serial BVH Construction

```
Function build_bvh(primitives):
    if number of primitives is small then
        return
    end
    for each axis (x, y, z) do
        partition primitives
        generate the bounding boxes for the left and right nodes
    end
    partition the primitives into left and right nodes
    build_bvh(primitives in the left node)
    build_bvh(primitives in the right node)
```

The most promising parallel construction algorithm up until now is the linear BVH (LBVH) [6]. The idea is to choose the order in which the leaf nodes appear in the tree, and then generate the internal nodes in a way that respect the order. [2] An ideal BVH tree would have objects that located close to each other in 3D space to also reside close in the hierarchy. (Figure 2.1) So the first step would be to sort them along a space-filling curve, commonly known as Z-ordered curve.

2.1.1 Morton Code and Z-ordered Curve

The Morton code for a 3D point is defined as $X_0Y_0Z_0X_1Y_1Z_1...$, where the sequence $X_0X_1X_2...$ and $Y_0Y_1Y_2...$ represents an integer that marks the point's grid position the world space. (Figure 2.2) In practice, we limit the length of Morton codes to 32 bits and use 10 bits each for X, Y and Z coordinate. For Bounding Volume Hierarchy (BVH), we can use the centroid of the primitive's axis-aligned bounding box (AABB) as the point for calculating its morton code. Once

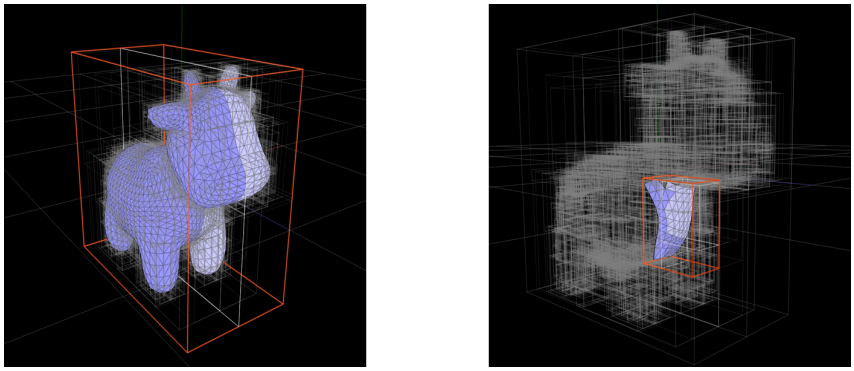

Figure 2.1: Visualization of BVH

the morton code for all points are generated, we can sort the them based on the unsigned int value of their morton code. This groups these points whose morton code are numerically close in spatial proximity. This brings maximize locality for geometric primitives in applications like ray tracing.

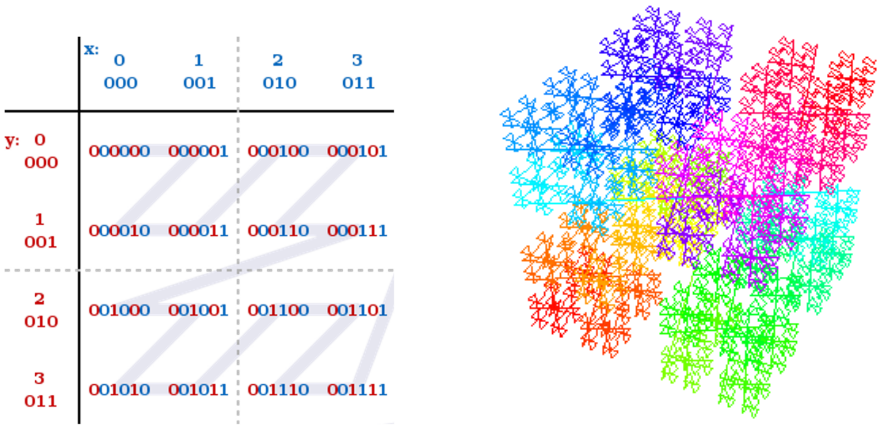

Figure 2.2: Z-ordered curve in 2D and 3D (wikipedia)

2.1.2 Construction of Binary Radix Trees

Since every leaf node is assigned a morton code as their key, a binary radix tree is defined as a hierarchical representation of their common prefixes. The keys are represented by the leaf nodes, and each internal node corresponds to the longest common prefix shared by the keys in their subtree. In addition to being a prefix tree, which contains one internal node for every common prefix, a radix tree omits nodes with only one child. Thus, every binary radix tree with n leaf nodes contains exactly $n - 1$ internal nodes.

After sorting the primitives according to their keys (Figure 2.3), keys covered by an internal node can be represented as a linear range $[i, j]$. In fig , the range of keys covered by each node is indicated by a horizontal bar, and the read circle shows the split position, representing the first differing bit

between keys. Tero Karras et al used $\delta(i, j)$ to denote the length of the longest common prefix between keys k_i and k_j, the sorted predicate implies $\delta(i', j') \geq \delta(i, j)$ for $[i', j'] \in [i, j]$. Thus, an internal node key is computed by comparing its first and last key, and the keys in between are guaranteed to shared the same prefix.

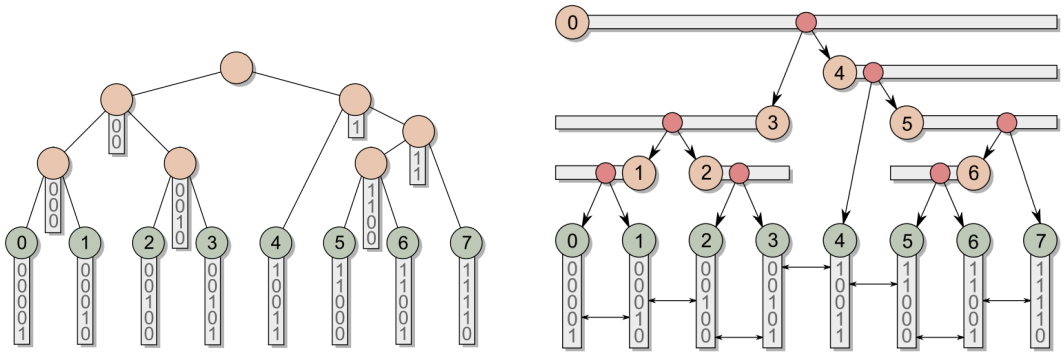

Figure 2.3: Binary Radix Tree and Linear Binary Radix Tree. Leaf nodes numbered 0-7 stores a 5 bit key in lexicographical order, and internal nodes represent their longest common prefixes. Each internal node covers a linear range of keys.

Garanzha el al proposed to construct each level of the linear radix tree in parallel. It is efficient but not as fast as constructing all nodes in parallel. The key insight from the paper of Tero Karras is to establish connection between node indices and keys through a specific tree layout. It assigns indices for the internal nodes in a way that allows it to find their children without knowing about previous nodes.

In order to constuct a binary radix tree, we need to determine the range of keys covered by each internal node and their children. One end of the node is its starting index, the other end can be determined by looking at nearby keys. Then its children can be defined by finding the proper split position. The algorithm listed below is an excerpt from Tero's paper.

Leaf nodes and internal nodes are stored in two separate arrays, L and I. The root node is at I_0, and the indices of its children are assigned according to its split γ. The index of every internal node coincides with either its first or last key. We process each internal node I_i in parallel. First, determine the direction of its range by looking at the neighboring keys, the direction with the longer common prefix should be the range of its internal indices. Then the other end must belong to a sibling node. Once we have the direction of the range, the lower bound for the length of the prefix is given by $\delta_{min} = \delta(i, i - d)$, so that $\delta(i, j) > \delta_{min}$ for any k_j belonging to I_i. Then by comparing all the pairs, we always choose d so that $\delta(i, i + d)$ corresponds to the larger one. The same principle is used to find the other end of the range by searching for the largest l that satisfies $\delta(i, i + ld) > \delta_{min}$.

Algorithm 2: Parallel Construction of ordered Binary Radix Tree

for each internal node with index [0, n-2] in parallel `build_node(`*primitives*`):`

// determine the direction of the range (+/-)

d = sign($\delta(i, i+1) - \delta(i, i-1)$)

// compute upper bound for the length of the range

$\delta_{min} = \delta(i, i-d)$

$l_max = 2$

while $\delta(i, i + l_{max} * d) > \delta_{min}$ **do**
| $l_{max} = l_{max} * 2$

end

// find the other end using binary search

$l = 0$

for $t \in [l_{max}/2, l_{max}/4, ..., 1]$ **do**
| **if** $\delta(i, i + (l + t) * d) > \delta_{min}$ **then**
| | $l = l + t$
| **end**
| $j = i + l * d$

end

// find the split point using binary search

$\delta_{node} = \delta(i, j)$

$s = 0$

for $t \in [l/2, l/4, ..., 1$ **do**
| **if** $\delta(i, i + (s + t) * d > \delta_{node}$ **then**
| | $s = s + t$
| **end**

end

$\gamma = i + s * d + min(d, 0)$

// assign child node pointers

if $min(i, j) = \gamma$ **then** $left = L_\gamma$

else $left = I_\gamma$

if $max(i, j) = \gamma + 1$ **then** $right = L_{\gamma+1}$

else $right = I_{\gamma+1}$

2.1.3 Bounding Box Calculation

Now that we have a hierarchy of nodes in place, the only thing left to do is to assign a conservative bounding box for each of them. The approach of Teros' is to do a parallel bottom-up reduction, where each thread starts from a single leaf node and walks toward the root. To find the bounding box of a given node, the thread simply looks up the bounding boxes of its children and calculates their union. To avoid duplicate work, the idea is to use an atomic flag per node to terminate the first thread that enters it, while letting the second one through. This ensures that every node gets processed only once, and not before both of its children are processed. The bounding box calculation has high execution divergence, meaning only half of the threads remain active after processing one node, one quarter after processing two nodes, one eighth after processing three nodes, and so on. However, this is not really a problem in practice because of two reasons. First, bounding box calculation takes only 0.06 ms, which is still reasonably low compared to sorting the objects. Second, the processing mainly consists of reading and writing bounding boxes, and the amount of computation is minimal. This means that the execution time is almost entirely dictated by the available memory bandwidth, and reducing execution divergence would not really help that much.

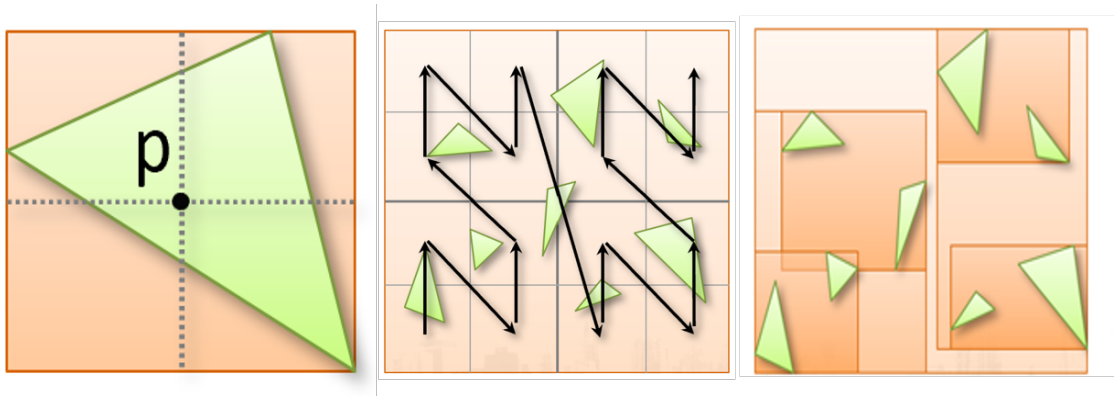

Figure 2.4: From Primitives to sorted Bounding Boxes

2.2 Parallel Construction of Octree/Quadtree

This node-level parallelization can also be applied to octree and quadtree. The difference is that the bounding boxes of octree and quadtree nodes are predetermined based on the point's grid position. Our project implements quadtree, where each $2k$-bit prefix of a given morton code maps to a quadtree node at level k. We can enumerate these prefixes by looking at the edges of a corresponding binary radix tree. We evaluate the counts from morton code during radix tree construction, and then perform parallel prefix sum to allocate the quadtree nodes. The parent of the quadtree nodes can be found by looking at the parent pointer of each radix tree nodes.

3. Performance

To measure performance, we benchmarked the construction time of our BVH and Quadtree. Then, we applied our BVH implementation to a CUDA-based physically-based rendering pipeline that we found online [7] to test its efficiency in real world ray-tracing applications. In addition, we extended the scope of homework 2-3 for a non-uniform particle simulation and applied our parallel Quadtree data structure to test out its performance.

3.1 Bounding Volume Hierarchy

In the following, we conduct parallel scalability experiments to investigate the behavior of our BVH hierarchy generation. We present the results of a strong scaling experiment as well as a path tracing rendering program.

3.1.1 Construction Efficiency

Figure 3.1 shows the performance of BVH hierarchy generation for Stanford Dragon. In this case the problem size stays fixed (871K triangles) but the number of blocks and the number of threads per block are increased. The x-axis corresponds to number of blocks (upper) and number of threads per block (lower). The y-axis corresponds to the time spent (left), parallel speedup (middle), and parallel efficiency (right) respectively. The dotted red line indicates the ideal performance, where the amount of parallelism is optimal. Our method, indicated by the solid blue line, scales somewhat linearly with the number of blocks and threads, and is pretty close to ideal case.

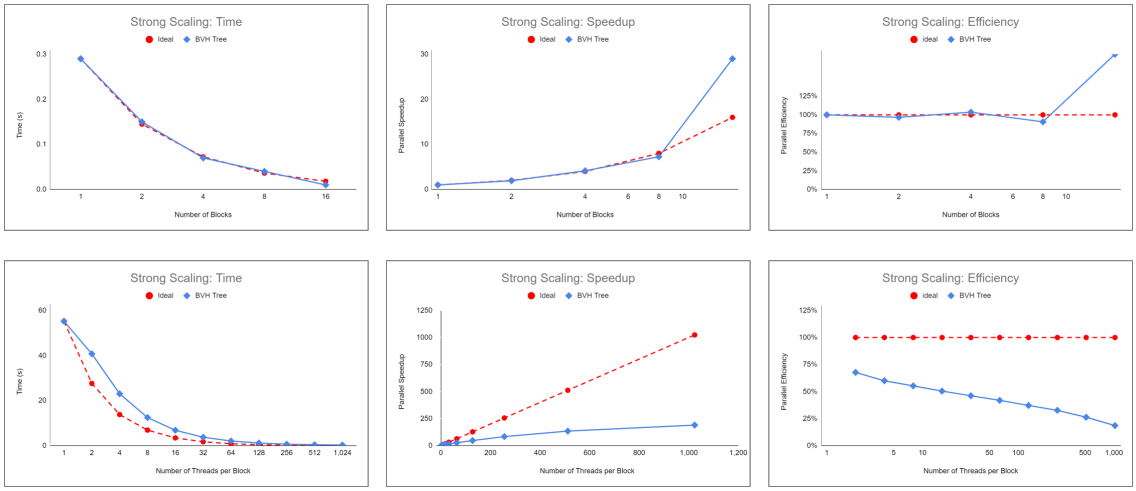

Figure 3.1: BVH construction time of Stanford Dragon as a function of the number of blocks and the number of threads per block.

3.1.2 Application - Ray Tracing Renderer

Bounding volume hierarchies are used to support several operations on sets of geometric objects efficiently, and real-time ray tracing is one of the most common use cases. BVHs eliminate potential intersection candidates within a scene by omitting geometric objects located in bounding volumes which are not intersected by the current ray.

Here, we combine our BVH data structures and Hong's open source GPU path tracer [7] to render several 3D models. (Figure 3.2) Then, we compare the render time for the serial and parallel BVH construction methods.

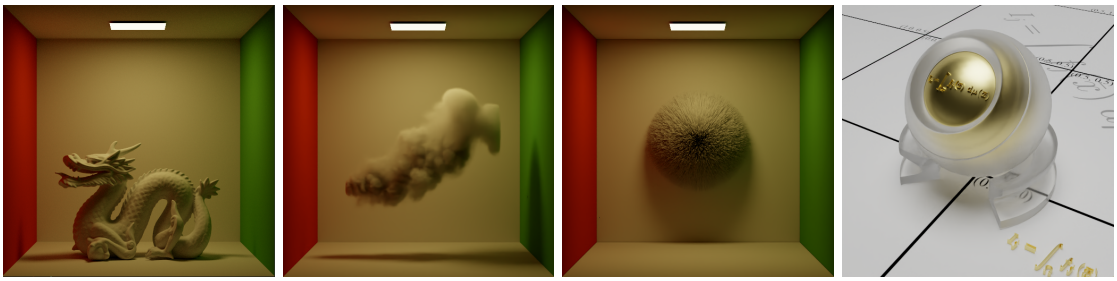

Figure 3.2: Sample models used in the ray tracing render-er experiment.

To evaluate the overall performance of the parallel construction method for ray tracing, we record its BVH construction time and path tracing rendering time and compare them to serial method. We experiment on meshes with different number of primitives, shown in Table 3.1. As for BVH construction, parallel construction is much faster than serial construction, especially when number of primitives is large enough. However, the quality of resulting BVH structure in our method is always slightly worse than that in serial method, causing path tracer to spend more time on rendering. To further explore on this, we also count the average number of bounding box intersection function call per ray in two cases. In BVH constructed by our method, a ray would have to check intersection with nearly 1 time more bounding boxes, which explains the difference in rendering time.

Time break down above demonstrates that when there are no large enough number of primitives, our method will not show advantage in overall time cost over serial method. As number of primitives increases, its advantage will become more distinct since in this case, BVH construction time will become a significant factor affecting overall time cost.

	dragon-S (10K)		dragon-M (40K)		dragon-L (160K)	
	serial	parallel	serial	parallel	serial	parallel
BVH construction (s)	0.31	0.01	1.40	0.03	6.13	0.12
rendering (s)	5.35	7.58	6.26	8.70	7.16	9.97
avg bboxes intersection	38.25	73.72	38.66	73.01	43.19	78.42

Table 3.1: BVH construction time and quality of generating BVHs using CPU-version construction (serial) and our method (parallel) for different meshes (10K, 40K, 160K primitives respectively). Our method is much faster in construction. But the generated BVH is not as efficient as theirs, which can be shown by rendering time and average times per ray of bounding box intersection computation.

3.2 Quadtree

In the following, we conducted parallel scalability test to benchmark the performance of our quadtree implementation. We first tested the tree construction efficiency. Then we modified homework 2-3 for a non-uniform particle simulation and utilized our quadtree as the fundamental building block. Finally we tested how its performance scales with problem size.

3.2.1 Construction Efficiency

Figure 3.3 shows the performance of construction a quadtree with fixed problem size (1 million points) scales with the number of blocks with fixed number of threads per block (256). The left sub figure shows the execution time on the y-axis, the one in the middle shows raw speed up compared to one block, and the one on the right shows the percentage efficiency on the y-axis. Ideally, when the total number of threads do not exceed the total number of nodes constructed (approximated by the number of points here), the expected speed up should scale linearly with total number of threads. When the total number of threads exceeds the number of points, meaning the extra idle threads do not contribute, we then reach the roofline for possible speed up. The ideal situation is denoted by the red dotted line on the plot.

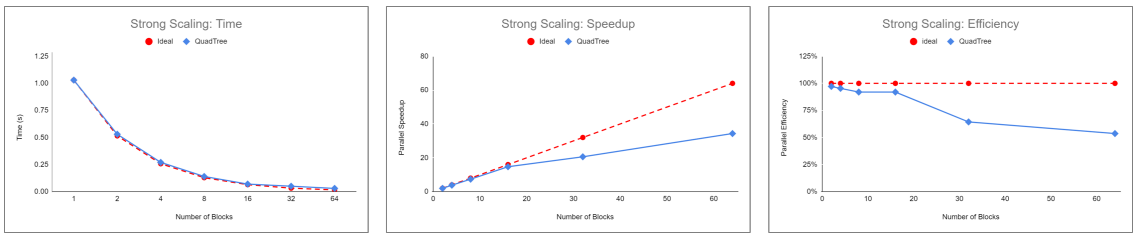

Figure 3.3: Quad-Tree Construction Performance

Figure 3.5 shows the construction speed scalability with number of particles from 10 to 100,000,000.

The number of blocks is fixed at 8, and the number of threads per block is fixed at 16.

3.2.2 Application - Boids Particle Simulation

Quadtrees are most useful in particle simulations when the average distribution of particles throughout the time span is not uniform. We found the ideas of boids interesting and particularly suitable in that case. Boids are a kind of artificial life that simulates the behavior of flocking birds or fish. Compare to the simple repulsive particle interaction, a boid usually have three types of force exerted on it: Separation, Cohesion and Alignment. For details for these forces, we referenced the paper by Silvia el al [8]. We modified the apply_force function from homework 2-3 to accommodate the changes.

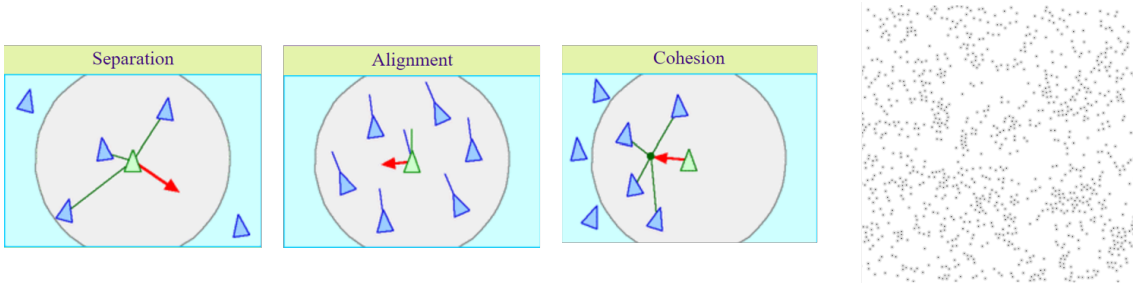

Figure 3.4: Rule 1: Each boid attempts to maintain a reasonable amount of distance between itself and any nearby boids, to prevent overcrowding. **Rule 2:** Boids try to change their position so that it corresponds with the average alignment of other nearby boids. **Rule 2:** Every boid attempts to move towards the average position of other nearby boids.

We tested the simulation time for fixed 64 blocks and 256 threads per block on varying problem size. Figure 3.5 shows how construction time scales with problem size when fixed amount of threads is given (8 blocks, 16 threads per block). Figure 3.6 shows how the complete boid simulation time scales with the number of particles.

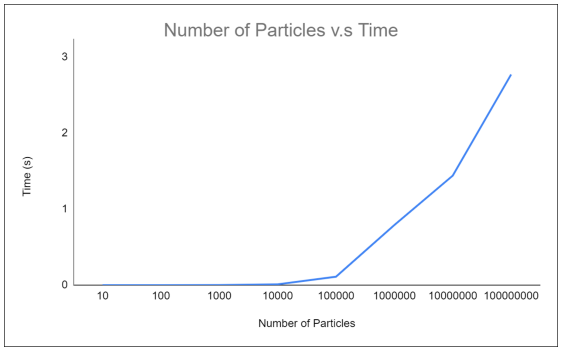

Figure 3.5: Quad Tree Construction speed scalability with number of particles

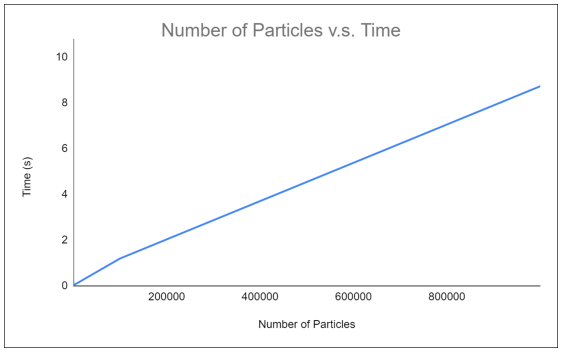

Figure 3.6: Boid simulation time (y-axis) as a function of the number of particles (x-axis)

References

[1] T. Ize, I. Wald, and S. G. Parker, "Asynchronous bvh construction for ray tracing dynamic scenes on parallel multi-core architectures," in *Proceedings of the 7th Eurographics conference on Parallel Graphics and Visualization*, pp. 101–108, 2007.

[2] T. Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees.," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pp. 33–37, 2012.

[3] K. Garanzha, J. Pantaleoni, and D. K. Mcallister, "Simpler and faster hlbvh with work queues," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 59–64, 2011.

[4] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, vol. 28 of *2*, (Oxford, UK), pp. 375–384, Blackwell Publishing Ltd, April 2009.

[5] J. Pantaleonil and D. Luebke, "Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, p. 87–95, 2010.

[6] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, vol. 28, pp. 375–384, Wiley Online Library, 2009.

[7] F. Hong, "gpu-pathtracer." `https://github.com/brickray/gpu-pathtracer`, 2019.

[8] A. R. da Silva, W. S. Lages, and L. Chaimowicz, "Improving boids algorithm in gpu using estimated self occlusion," *Proceedings of SBGames' 08: Computing Track, Computers in Entertainment (CIE)*, pp. 41–46, 2008.