Boids optimization in Unity

Bradley Bath

December 25, 2020

# Introduction

For this worksheet I’m creating a boids flocking algorithm, testing to see how much boids I can simulate before the framerate goes below 60FPS and then optimizing the algorithm until I reach a point where I can no longer optimize.

You can find the Unity project for this here. There’s two branches, main, which contains the algorithm with no optimizations, and optimized, which contains the optimizations.

**https://github.com/BradFitz66/UnityBoids/**

# Baseline

To start, I created a new Unity project and found a post describing the Boids algorithm written by the original author Craig Reynolds [1]. This describes how to implement the 3 stages, alignment, cohesion, and separation. I created two new scripts in Unity, “Flock” and “Boids”. The flock script contains a list of boids, and the Boids script also contains a list of Boids, and in then in the Start function of the flock script, it sets each boid it controls list of boids to the list it has. This is so each boid knows what other boids it can interact with (it won’t interact with boids from other flocks)

And then, using the post by Craig Reynolds, I implemented the flocking algorithm stages and also added an extra ‘goal’ stage, which steers each boid towards a point called ‘goal’. The boids use a simple velocity movement style which translates the transform of the boid by the current velocity, with no rigidbody component(to save on the overhead that rigidbodies cause). This means we’re not relying on the performance of Unity’s physics engine.

As a baseline, I created a single flock of 216 boids. At this current point, it ran at around 17FPS with the bottleneck being the CPU. The profiler reported that 88% of the CPU was being used by Boids.Update() inside Update.ScriptRunBehaviourUpdate.BehaviourUpdate To make sure that the bottleneck stayed on the CPU, I created a new material and applied them to each boid. This material has GPU Instancing checked which means that it will render each boid in a single draw call, making the GPU do less work.

**3 Optimization**

After getting a baseline performance reading, I started on optimization. The first thing I wanted to do was implement an octree. The reason for this is so that instead of each boid checking over every other boid in it’s flock, we only check boids that are in the same octree partition. This limits the amount of boids a single boid is iterating over which should increase performance. Instead of implementing my own octree (due to time constraints), I found a free open-source implementation on github[2].

To implement the octree, I created a new PointOctree in the flock script and added all the boids to it on start. I also added some new variables to the Boids script such as a reference to their flock (mostly so they can also reference the octree) and then in the update of the flock script, I remove and add back each boid to update the tree.

Then, to make the boids only care about boids in neighbouring octree nodes, I use GetNearbyNonAlloc with a distance of 2 and give Boids.boidsInRange as an argument to the function. This is faster than using the other function GetNearbyAlloc because it doesn’t allocate anything and just inserts data into an already initialized array.

One more small optimization I did was replace the Mathf.Pow function inside the separation function with a “faster” but slightly less accurate Pow function based on a paper by Nicol N. Schraudolp[3]. The function itself was written by Alexey Abramenko on twitter[4].

At this point, the algorithm was now operating at 60+ FPS with the octree optimization however if boids were close together, the FPS would drop down to 30~ FPS, which is still almost double the FPS without the octree. To make the FPS a constant 60+ FPS while testing, I created a script that moves the goal position randomly using UnitInsideSphere so the boids are always moving (this was specifically to help testing rather than an actual attempt to improve performance).

In the profiler, the amount of CPU time the Boids update was taking has dropped down to only 15%, with the flock script going up from 0% to 25% mostly due to the updating of the octree.

# 4 Conclusion

In conclusion, although there is still more performance on the table through a “better” octree implementation (the github page for the implementation I use has notes at the bottom of the README for how it could be improved) or through a compute shader to compute most of the algorithm on the GPU, the amount of time and the complexity to do either (especially the compute shader) outweighs the benefits for me.

There is guides on how to do GPU boids through Sebastian Lagues video on Boids[5] or through various open source boids projects on github. Unity’s DOTS/ECS is also a good option and there are open source boid implementations for those as-well. One other thing is that you can build the project to see how it performs without overhead from the Unity editor, but that is untested.

I am happy with the performance I have obtained through the use of the octree, and through tinkering of the various properties of the octree and boids, you could get probably get even better performance. For now, I think it’s satisfactory.

# References

1. https://www.red3d.com/cwr/boid
2. https://github.com/Nition/UnityOctree
3. https://nic.schraudolph.org/pubs/Schraudolph99.pdf
4. https://twitter.com/vapgames/status/1321827615225892864
5. https://www.youtube.com/watch?v=bqtqltqcQhw