Optimization of 6 Degrees-Of-Freedom Large Scale

Decentralized Multi-Agent Formations.

Jacob Diaz

COMP 3190

Professor John Anderson

December 20th, 2020

**TABLE OF CONTENTS:**

* **ABSTRACT**
* **INTRODUCTION**
* **MATERIALS AND METHODS**
  + **THE SOFTWARE / LANGUAGES USED**
  + **THE GOAL**
  + **THE SYSTEM**
* **RESULTS**
  + **SYSTEM OPTIMIZATIONS**
    - **METHOD #1**
    - **METHOD #2**
    - **METHOD #3**
* **DISCUSSION**
  + **MOVEMENT OBSERVATIONS**
  + **DATA INCONSISTENCIES**
* **BIBLIOGRAPHY**

**ABSTRACT:**

**The natural behaviours of flocks, herds, and schools of animals have been studied for years, and many behavioral models have been created using simple, decentralized controllers. Many of these systems depend on the “boid” model. In this paper we work to expand the behaviours within the “boid” model and look into possible optimization methods that can be implemented to greatly expand the size and complexity of these multi-agent simulations.**

**INTRODUCTION:**

For many years many papers have focused on analyzing the mechanics of the highly dynamical systems present in animal flocks. It is widely accepted that the common formations that animals with 6 degrees of freedom of movement (birds, aquatic life) offer some form of aerodynamic improvement to at least some of the animals in the group, and we have found that this can be extended into man-made machines, “These studies have demonstrated the true effectiveness of the boid control concept for managing conflicting goals, such cohesion and separation, in a flock of UAVs while also achieving core mission objectives” (Glen A. Dimock, 2003). This paper chooses to take the notions of individual control and heighten them. We analyze how we can find and optimize formations and movement inside large scale multi-agent systems. We will incorporate some natural movement constraints, such as constant velocity and limited mobility, to see how this affects the behaviour. Finally, we will develop methods for optimizing the computation of these large formations.

**MATERIALS AND METHODS:**

* **THE SOFTWARE / LANGUAGES USED:**

This paper primarily utilizes Unity3D (version 2018.2.0f2), as well as C# and HLSL. Unity was chosen because it is an industry standard for 3D graphics in games, and general media applications, and therefore it would be easy to put together visualizations as well as simulations hand-in-hand. Unity also offers some great tools for smoothing out the workflow such as it’s interface system between C# and relevant compute shaders that were used to optimize the project.

* **THE GOAL:**

We will be implementing the Boid model introduced at SIGGRAPH ‘87 (Reynolds, 1987) that implements common agent behaviours such as separation, alignment, and cohesion (illustrated later). The main constraint we will consider is that of a roughly constant velocity. This constraint is observed in many natural systems such as flocks of birds that must maintain lift, or schools of fish that always maintain their speed. We will also consider how this constraint acts with other constraints such as various forms of input, and with various external forces.

Our goal is to analyze and expand the classical Boid model of multi-agent dynamics to support other actions like swarm targeting, rough swarm formations, and the handling of external forces. There will be centralized processing done on within the systems in this paper as a means of optimization, but the agents will all still act on their individual sensory input. All centralized computation is strictly for getting and setting global test parameters.

* **THE SYSTEM:**

We begin by selecting a simulation model to base our work on, the boid is a very good starting point for the decentralized dynamics we’re looking for, and so we go forward introducing a framework for this system. We wish to create a list of only the essential behaviours as motivated by (David C. Brogan, 2002). The core concepts are as follows:

* **Neighborhood:** The partial spherical volume around a single agent that defines its “sight” range. Any agent within another agent’s neighborhood is a **neighbor.**
* **Separation:** The behaviour that determines how much the agent steers away from other agents in their neighborhood
* **Alignment:** The behaviour that determines how much the agent steers towards the average heading of its neighbors.
* **Cohesion:** The behaviour that determines how much the agent steers towards the average position of its neighbours.
* **Tracking:** The behaviour that determines how strongly the agent wants to be close to a goal location

We wish to take on a localized realistic ‘vision’ system for our agents, similar to the *neighbor-referenced* sensing control scheme, “each robot maintains a position relative to one other predetermined robot.” (Jakob Fredslund, 2002). The key difference here is the system in this paper references the single closest neighbor for very fine-turned formation control, our system modifies this to account for all neighbors.

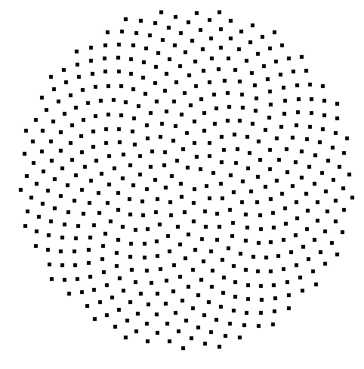
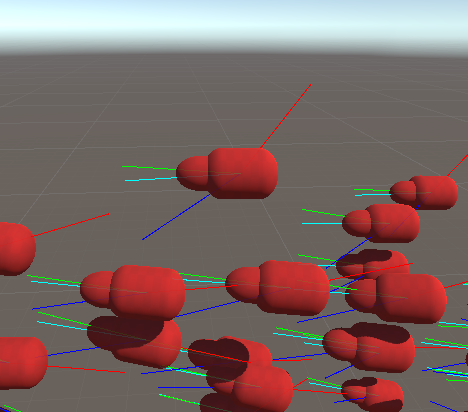
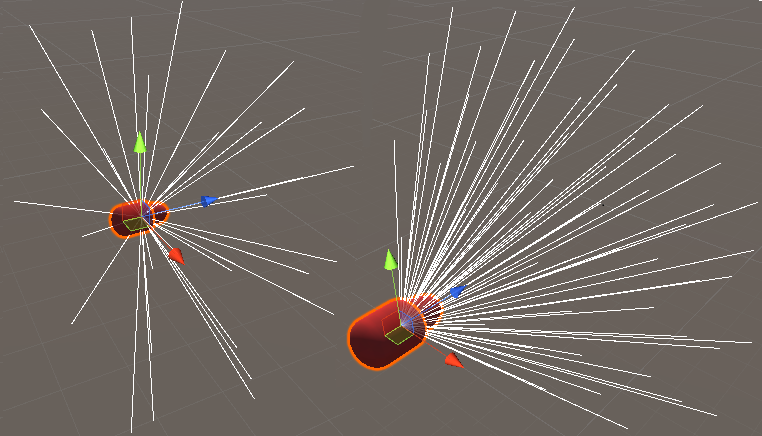
To achieve this vision system, we need to be able to sample obstacles in certain directions. For this we must choose equidistant points around the agent in (ideally) a pseudorandom way within a defined angle of sight. A simple parameterized sphere function is a good start, but we have issues that arise. The points on a parameterized sphere are not pseudorandom nor equidistant and a side affect of this placement gives increased resolution of the agent’s vision as it gets closer to the center. This behaviour might be useful in other applications, but for our technique we want equidistant points. There is also the possibility of loading in pre-set points from a regular icosahedron, while this method does supply us with equidistant points, most common formulas for the icosahedron subdivide exponentially, removing some aspects of scalability. Finally, there exists a method of mapping points of the Fibonacci spiral to a sphere as found in a paper regarding illumination integrals. (R. Marques, 2013) This is the method we are going to use. The mapped 3D vectors are calculated and stored at runtime to be referenced later. This method is superior as it gives equidistant points that can be fine-tuned, and it even gives us the added bonus of a pseudorandom formation order (This “pseudorandom sampling” is due to the formula relying on the “golden angle” *ϕ* ≈137.508). We can see the formation of the Fibonacci points as vision rays around a single agent in *Figure* 2. Finally, each agent also necessarily has multiple parameters dictating movement speed, turn speed, sight distance and sight angle. On every time step we sample the neighborhood of each agent and apply each behaviour according to assigned global behaviour strength parameters. Our agents also perform sphere-casts (modified raycasts that perform a spherical collision check at every timestep) from each defined sight line to appropriately detect static world objects and avoid them. All of these systems and behaviours return rotations, and the final output of a single timestep evaluation is a rotation, a Quaternion target(*Q*) that the agent attempts to match. *Q* is a linear combination of all behaviour goal orientations and their associated behaviour strengths. In *Figure 3* we can clearly see how the behaviours and their corresponding desired orientations work together. This weighted average of rotations with agents possessing 6 degrees of freedom is adapted from the herding behaviour mechanics seen in (Jessica K. Hodgins, 2003).

Figure : A Fibonacci Spiral (Duvenhage, 2019)

Figure 3: A Grouping of agents with their behaviour orientation goals displayed (Red = Separation, Green = Alignment, Blue = Cohesion, Cyan = Tracking)

Figure : 3D Mapped Fibonacci Points. (Left: 120deg FOV with less resolution, Right: 55deg FOV with greater resolution)

**RESULTS:**

* **SYSTEM OPTIMIZATIONS:**

This system performs a lot of unneeded iteration. If we are smart with our calculations this gives us an *O(n^2)* computational complexity at best (where number of agents = n). This is because we iterate across all agents (n), and then withing each agent we check all relative distances to other agents (n). We can greatly increase the computation speed by leveraging the power of the GPU with compute shaders. Compute shaders are useful for this optimization as floating-point operations across numerous threads are what the GPU is designed for, and so we will look at some GPU-based approaches. With Unity3D we can interface with the compute shaders in the hopes of bringing our computational complexity down. We first define the agent’s dataset:

**3D Vector – Position:** The 3D position of the agent within the system.

**3D Vector – Forward Direction:** The 3D normalized forward vector of the agent.

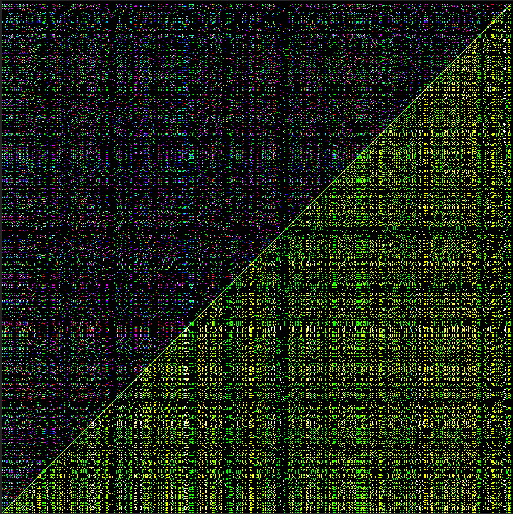
**3D Vector – Right Direction:** The 3D right direction (orthogonal to forward) for considering appropriate orientations with regards to roll. We do not use this value here but it can be easily added.

After loading the agent’s data into a structured buffer linked to our compute shader we can attempt optimizations. In these trials we tested with 300 agents (*n = 300)*, a sight distance of 3 units (*d = 3,* or 3x the length of one agent), and a view angle of 110 degrees (θ = 110). These values are constant throughout the tests, with the exception of performance testing in the final algorithm.

* **Method #1**

We first consider the GPU as a method of compacting and transferring data. We load the datasets into the GPU and write all information to a texture, allowing for easier addressing within each agent through a shared texture. On average this method performs the same as if each agent individually calculated their values, because each agent still is doing the majority work. This method also only stores positional data as color, and therefore is not enough.

* **Method #2**

Building on the first method we wish to compact our data, to fit more data into the same texture. Because the Euclidean distance is a symmetrical relationship, we can immediately leverage this to double our texture data density. We store position and forward direction data in color, with the distance between the agents stored as the alpha. This give each agent a “validity” value within the alpha to reduce computation further. To simplify each agent’s internal calculations further we also store relative position and direction data instead of absolute positions.

With this method we dispatch large groups of threads across minimal GPU wavefronts in the hopes of seeing performance increases from the small number of GPU cycles we take. This test had 16x16 thread groups of 32x32 threads each culminating in a 512x512 resolution texture, or support for 512 agents.

Figure 4: A visualization of the GPU's texture buffer

In *Figure 4* the upper triangle of data stores the position deltas, the lower triangle stores direction deltas. We can note that the diagonal separating these halves represents an agent testing against itself, and this data is discarded for now. This method serves as a strong framework for dense data calculation, but in practice when large number of agents are in close proximity, we still retain an *O(n^2)* computational complexity. It does however cut down on the number of object references required by each agent, replacing them with simple comparisons. This method gives a small performance increase of roughly 10% more frames rendered per second on average.

* **Method #3**

Our core issue at this point is the presence of iterations within each agent itself because each iteration across *n* values still retains the chance of a worst case *O(n^2)* computational complexity. We wish to bring all iteration into the compute shader. To do this we must first lower the vast parallelism that *Method 2* leveraged to a single dimension of threads. We run one thread per agent, and each thread iterates through all other agents. This method drastically lowers the size of the texture in favour of putting more work into each thread itself. We run a single grouping of 340x3 threads (grouped by the GPU, not manually by the user this time), and we get an output that is a resolution of 340x3. This texture supports 340 agents and essentially supplies a separation, alignment, and cohesion direction vector directly to each agent via a texture lookup. Unlike the many common bounded approaches (Jessica K. Hodgins, 2003), these output values are always dependent on every agent in the system and there is no local limit to the number of nearby agents to consider. This system more effectively models the complex, but still goal-based, dynamics found in large schools of fish, where we do not consider the influence of small, limited numbers of neighboring agents, but the motion of the system as a whole to give intelligent and believable movement similar to that of a school of fish or a flock of birds.

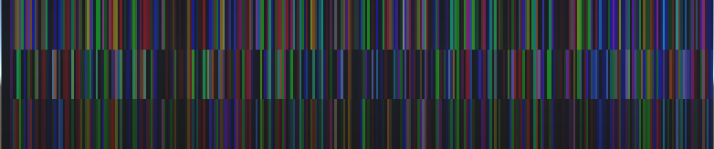
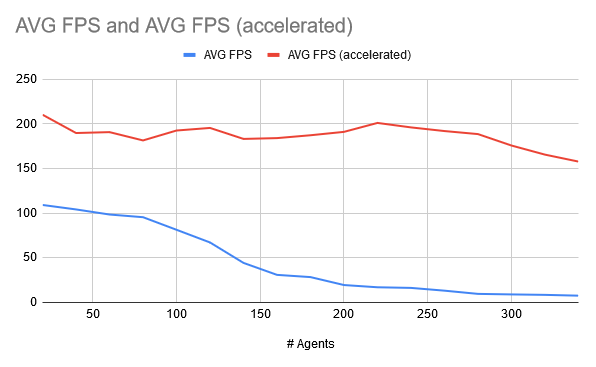
The important metric we note here is with the final method we have reduced computational complexity from *O(n^2)* to that of *O(n).* Each agent can simply intake their behaviour orientations through a texture lookup with minimal overhead. In *Figure 5* we can see the drastically reduced texture size, with a much higher density of important information when compared to previous methods. This method can also be easily expanded to accommodate for more data within the compute shader by simply increasing the number of threads we request and modifying the resolutions of the texture. We must note that there are thread dispatch limits and a system that incorporates much larger operations or more complex compute shader actions will need to accommodate for this with either multiple computer shaders, multiple thread dispatches, and/or a restructuring of the algorithm. We can see in *Figure 6* just how much greater the average frames per second are when compared to the original *O(n^2)* method on the CPU alone.

Figure 5: A visualization of the 340x3 GPU texture buffer

Figure 6: A graph relating the classic CPU-bound method of calculation with Method #3 on the GPU

**DISCUSSION:**

* **MOVEMENT OBSERVATIONS:**

All methods present in this paper do give realistic results, but it is worth noting that across all methods the same behaviour weight values do not apply. I believe that this could be due to the transfer from Color data to Floating-point data, and a mismatch of expected vector normalizations within the agent. It is also worth noting that if a greater weight is assigned to that of separation than that of tracking or bounds correction then each agent will essentially ignore tracking altogether is the right scenarios. I believe that this could be fixed by changing our behaviour weights from constant values to functions of relative distances, for example: Let the tracking behaviour weight be proportional to the distance from the goal.

The constant velocity constraint upon the agents is a key part of making the flock appear believable, as well as reduce the number of collisions. This algorithm does not take velocity into account and the behaviour constants are tuned for their set velocity. If we wanted to take into account new velocities and possible accelerations, we can simply establish a greater number of data buffers for the GPU to intake, there is no change in texture output size.

* **DATA INCONSISTENSIES:**

It must be noted that there is no direct relationship between each *Color* value and the standard *float* value. In this project we used a standard ARGB32 texture format, allocating 8 bits to each for floating point values within [0,1]. We note here that the RGB color values of the render texture are normalized and that this forces our vector output to also be normalized, but this works perfectly for our needs. There does exist a mismatch in floating point precision between the *Color* type and the *float* type but given the chaotic nature of this system this loss in precision of behaviour direction values does not negatively affect the behaviour of the agents.

One might finally notice the odd texture size of 340x3 in the final method of GPU computation. It is the only texture in the project that does not have power of 2 dimensions. This is due to a restriction of 1024 threads created per dispatch on the GPU. Since we have 3 values per agent to calculate we can perform at most: 1024/3 ≈ 340 agent calculations per dispatch. This could be easily increased through multiple GPU dispatches and/or a restructure of the compute shader code.

**BIBLIOGRAPHY:**

David C. Brogan, J. K. H., 2002. *Simulation Level of Detail for Multiagent Control,* Georgia: Association for Computing Machinery.

Duvenhage, B., 2019. *Bernardt Duvenhage's Blog.* [Online]   
Available at: https://bduvenhage.me  
[Accessed 25 November 2020].

Glen A. Dimock, M. S. S., 2003. *THE AERODYNAMIC BENEFITS OF SELF-ORGANIZATION IN BIRD FLOCKS,* Urbana, IL, USA: American Institute of Aeronautics and Astronautics (AIAA).

Jakob Fredslund, M. J. M., 2002. *1A General Algorithm for Robot Formations Using Local Sensing and Minimal Communication,* Los Angeles, CA, USA: s.n.

Jakob Fredslund, M. J. M., 2002. *1A General Algorithm for Robot Formations UsingLocal Sensing and Minimal Communication,* Los Angeles, CA, USA: s.n.

Jessica K. Hodgins, D. C. B., 2003. *Robot Herds: Group Behaviour for Systems with Significant Dynamics,* Atlanta, GA, USA: Georgia Institute of Technology.

R. Marques, C. B. M. R. L. K. B., 2013. Spherical Fibonacci Point Sets for Illumination Integrals. In: *Computer Graphic Form.* Malden, MA, USA: Blackwell Publishing, pp. 1 - 11.

Reynolds, C. W., 1987. *Flocks, Herds, and Schools: A Distributed Behavioral Model,* s.l.: s.n.