Boids that Sprint:
Optimising an AI Algorithm using Asynchronous Compute and GPGPU in DirectX12

Michael Ennis

BSc (Hons) Computer Games Applications Development

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures

Acknowledgements

I would like to thank my mentor, Ruth Falconer, for helping supervise my project, and all the lecturers at Abertay university who have helped me along the way.

I would like to thank my mother, stepfather and sister for their unyielding love and support throughout the hard times. I could not have finished this without all of you.

I want to dedicate this dissertation to the memory of my father, thank you for believing in me. I hope this has made you proud, dad.

Abstract

Artificial intelligence (AI) has become a fundamental component of many modern video games to complement their core game loops. Having large numbers of AI constantly execute their core functionality at run-time could negatively impact performance, lowering framerates and providing a frustrating experience for the end user. However, by offloading this to the graphics processing unit (GPU), performance could be improved, leading to these downsides being significantly negated.

The aim of this project is to evaluate the impact of utilizing general purpose computation on the graphics processing unit (GPGPU) and advanced DirectX 12 synchronization techniques to optimize performance and increase efficiency of a simple boids algorithm.

To address the core goals of the project, a DirectX 12 graphics application was developed to provide a foundation for three different iterations of the boids algorithm. The first iteration provided a control, by executing purely on a single thread within the CPU. The next offloaded the boids update functionality to the GPU, to fully utilize its parallel processing capabilities. The final version further optimizes the previous one by providing an asynchronous compute approach that executes both the compute and render pipelines simultaneously, to increase GPU resource utilization and overall performance.

Various tests results were gathered to measure the performance of all three iterations. The results indicated that the GPGPU and asynchronous compute approach significantly outperformed the CPU at all sample sizes. They also demonstrate a clear performance improvement when utilizing the asynchronous compute approach over the standard GPU version, which gradually declined as the sample size increased.

The project aims were appropriately met, as the results demonstrate the clear performance improvement of utilizing the GPGPU and the asynchronous compute approach. Further development of the project could implement spatial partitioning or individual asynchronous compute shaders per rule to further optimize performance.

Abbreviations, Symbols and Notation

GPGPU – General Purpose Computation on Graphics Processing Units

CUDA – Compute Unified Device Architecture

GPU – Graphics Processing Unit

CPU – Central Processing Unit

CBV – Constant Buffer View

UAV – Unordered Access View

SRV – Shader Resource View

ECS – Entity Component System

Boid – Bird-oid Object

API – Application Programming Interface

SM – Streaming Multiprocessor

AI – Artificial Intelligence

Compute Pipeline – refers to pipeline state object with single compute shader stage, specifically for compute tasks

Render Pipeline – refers to pipeline state object with vertex shader and pixel shader stages, specifically for graphics rendering tasks

Chapter 1 - Introduction

Artificial intelligence (AI) is commonly featured within video games to provide an engaging user experience, making games feel more alive and responsive. They can take on many different roles, including adversaries, helpful companions or simple quest givers. They are a conduit to interesting gameplay mechanics, responding to the player in entertaining ways.

AI agents can often be used as obstacles for the player to overcome, one example being large groups of enemies attempting to prevent the player from progressing through a level. These enemies, by acting as blockers, allow the player to feel engaged as they provide challenge and a sense of progression when overcome. One game that utilizes AI within this role is World War Z, by Saber Interactive. This cooperative multiplayer game provides an interesting experience by challenging their players to fight through a level full of antagonistic AI agents. This game supports up to 1000 entities attacking at run-time (Takahashi, 2019), representing large hordes of zombies that chase the players.

While this game stands out for its significant maximum run-time entity count, several others feature similar mechanics, requiring large entity numbers as a core gameplay feature to provide an engaging player experience. For the developers of these types of games to achieve this on their target platforms and minimum specifications, these AI agents must operate efficiently to ensure they do not significantly harm performance of the overall game.

As the demand for video games has increased, so has the complexity and performance requirements (Gach, 2025). Optimization within the games industry is vital to hitting average framerate expectations of the modern audience. One avenue to potential performance gains exists within the operations of gameplay AI. Large numbers of entities executing their behaviour at runtime results in a performance decline of the overall game, however by utilizing the parallel processing capabilities of the GPU, this could be significantly improved.

This project aims to analyse the performance improvements of optimising a simple boids algorithm using the GPU. This can be achieved through using GPGPU techniques to offload the functionality of the individual agents to the GPU for parallel processing. This algorithm was chosen due to it being an excellent candidate for this, as every agent performs the same three rules and operates on similar data. This aim is outlined by the research question:

- What performance improvements can be achieved by optimizing a single-threaded CPU based boids algorithm using GPGPU?

The project also aims to analyse a further optimisation of the GPGPU implementation by utilizing an asynchronous compute approach, outlined by the second research question:

- How can a GPGPU implementation of the boids algorithm be further optimized using an asynchronous compute approach in DirectX 12?

To answer these questions, three separate approaches to the boids algorithm are present within the application. The first implementation is a single-threaded CPU version, acting as a benchmark to visualize the performance increase obtained by the other iterations. The second implementation utilizes GPGPU techniques by offloading the AI operations of the boids algorithm to the GPU. The third implementation

applies an asynchronous compute approach to the standard GPGPU version for further optimization.

The remaining chapters of the dissertation attempt to address the key research questions. Chapter 2 investigates existing literature to determine appropriate GPU optimizations that could be implemented to increase performance. Chapter 3 meticulously describes the development process and justifications for specific design decisions regarding the application. Chapter 4 presents the results gathered through testing. Chapter 5 analyses the results gathered across all three boids implementations and describes observations found within the data. Chapter 6 describes an overview of the project and critically evaluates its effectiveness, alongside outlining potential future development paths.

Chapter 2 - Literature Review

To ensure that appropriate optimizations for the Boids model are implemented, several directions for improving performance should be taken into consideration. By exploring the current literature available and related work in this field, these options can be properly investigated and their impacts analysed before making any final decisions on which ones should be brought forward into the project. This section will give an overview of the boids model, general-purpose computation on graphics processing units, as well as any performance bottlenecks and potential optimizations that have been explored previously.

2.1 - The Boids Model

The Boids Algorithm, developed by Craig Reynolds (Reynolds, 1987), is an artificial intelligence behavioural model that simulates animal flocking behaviour, as seen in flocks of birds or schools of fish. There are three simple rules that enable this: separation, alignment and cohesion. Each of these rules encourage the boids agents to seamlessly traverse as a flock, basing their next decisions on the others around them. This results in emergent behaviour, as the individuals seem to move in directions that are coordinated, despite never being directly informed to do so.

The first rule, separation, encourages the boids to remain at a set distance from one another, forcing them to spread out. This rule is intended to prevent the boids from collapsing in on themselves, as this would cause them to collide with each other and dismantle the overall structure of the flock. This rule calculates the distance of any other boids in the model from the current one and applies force in the opposite direction if they are deemed to be too close.

The second rule, alignment, encourages nearby boids to move in a similar direction. This rule helps structure the flock by ensuring that they move alongside each other instead of colliding or drifting apart. This rule

calculates the average velocity of all nearby boids and applies force in that direction.

The final rule, cohesion, encourages the boids to move closer together. This rule is intended to act as a counterbalance to the separation rule, ensuring that no boid strays too far away from one another. This causes them to remain clustered in groups, rather than separating. This rule calculates the average position of all nearby boids and applies force in that direction.

The boids algorithm has been widely used in computer graphics to simulate flocks of animals or massed crowds of people. One such example is in the movie *'Batman Returns'* by Tim Burton, in which the algorithm effectively simulated swarms of bats and flocks of penguins. Another example is within the video game *'Half Life',* developed by Valve, to simulate a flock of birds near the end of the game which were coincidentally named "Boid" within the game files.

While this algorithm is excellent for simulating animal flocking behaviour, the performance of the naive model scales rather poorly given the worst-case time complexity is O(n^2), where 'n' represents the number of agents present within the model. This is because each boid takes into consideration all the others nearby to determine their next direction. When determining which boids are within range to impact the current one, a distance calculation needs to be performed.. This calculation between the two boids still needs to occur even if a boid is determined to be too far away to influence the current one, impacting performance negatively.

A potential optimization that could improve the performance of the model given the costly worst-case time complexity is to implement spatial partitioning. One research paper explores this topic in depth by testing the performance impact of different data structures on the computation time of a ranged-based search (Kratz, Luthman, 2021). This found that the basic grid data structure performed the most efficiently when the boids simulation contained fewer agents. As the number of agents

increased, the k-d and quad tree performed far better due to the trees adapting to the distribution of the agents dynamically, instead of being statically partitioned like the basic grid data structure.

Another research paper explores the topic in a similar way, by comparing the performance impact of the bin spatial partitioning data structure against a quadtree for a two-dimensional boids algorithm. However, it found that the bin data structure, similar to the basic grid from the previous paper, outperformed the quadtree data structure (Myszkier, 2023). This is likely due to a smaller test group of boids being used for the application, as the maximum dataset of only 5000 entities was used as compared to 11000 entities of the previous study. This implies that the previous hypothesis that grid-based data structures work more effectively at smaller entity counts while quad trees perform better at higher counts is likely still accurate (Kratz, Luthman, 2021).

Another potential optimization could be to process the calculations of the boids model on the GPU, increasing performance by executing them simultaneously. One research paper explores this topic by examining the efficiency of three separate implementations of the boids algorithm; one using multithreading on the CPU, another using a single core as a control subject and the final one using GPU resources on a compute shader in the graphics API DirectX11. The study found that the GPU implementation performed the best, reaching the highest frames per second and lowest update time per frame as compared to the other methods, when testing with an agent group size of between 64 and 4096 (Lindqvist, 2018). Similarly, the CPU multithreaded approach performed better than the control, however it did not outperform the GPU approach at any of the group sizes tested.

A similar paper also explores the topic of utilizing the GPU for performance improvements in the boids model but simulates multiple different flocks of boids instead (Husselmann, Hawick, 2011). Their algorithm is developed using CUDA, instead of using a graphics API. This

allows for the implementation to be separate from the graphics processing pipeline and allows for more fine-grained control over GPU memory management than traditional compute shaders. The raw performance of this method is competitive to a compute shader (Sörman, 2016), as such these methods are somewhat interchangeable. This research paper finds that the performance gains from implementing the model using the GPU is satisfactory, indicating that this implementation could be an effective method to optimizing the model.

A more unique paper focuses on creating a new feature that explores lowering the time complexity of the model by performing self-occlusion (Silva et al., 2009) that effectively reduces the number of neighbours used in the calculations of the boids model. This worked by determining if a specific cell in a grid-based data structure was within a certain angle and range of an individual boid. If so, any boids contained within those cells in the data structure were taken into consideration for the calculations.

Two versions of this model were then developed: a CUDA and a Cg shader language version with a compute shader implementation for the GPU. The study found that the Cg shader language version was faster than the CUDA variant when simulating the algorithm itself with both variants using self-occlusion, however the grid construction was far quicker on CUDA. One caveat to this research paper is that by implementing this visibility culling approach, the behaviour of the boids are somewhat altered. This is because there are far less boids being taken into consideration for the calculations, leading to divergent behaviour from the original model. To combat this, the researchers changed the parameters to encourage it to act more like the original, however the exact same behaviour cannot be expected given the new feature added to the model.

2.2 - GPGPU

General purpose computation on the graphics processing unit (GPGPU) involves using the parallel processing capabilities of the GPU to complete tasks that are generally processed by the CPU. The individual cores found within the GPU are efficient at dealing with mathematical computations due to having more transistors dedicated to them (Wang, 2023). In comparison, CPU cores are far more powerful, but are large and limited in number, with more transistors dedicated to control logic and caching for tasks that are more general purpose. This difference in architecture makes certain tasks with high degrees of parallelism excellent candidates for processing on the GPU, as fully utilizing the numerous cores available could result in significant performance gains compared to the CPU-based approach.

To utilize the benefits of GPU architecture for general purpose computation, tasks are often executed via compute shaders within a graphics API. The compute shader is a programmable stage in the graphics pipeline that can effectively utilize the GPU for general purpose computation, as it is not limited to graphics rendering. Individual tasks sent to the GPU are referred to as “Kernels”, which are functions designed to be executed in parallel by many threads.

The GPU architecture follows the SIMT (Single Instruction, Multiple Threads) model, which allows many threads to operate concurrently. All the threads in the model operate on the same instruction simultaneously but each individual thread works on different data elements, ensuring the efficient parallel processing of large data sets.

Threads on the GPU are organized into groups of 32, known as thread warps. Each thread in a warp executes the same instruction simultaneously in the kernel. This is known as lock-step execution (Wang, 2023). When conditional statements are reached and some of the threads go down a different branch, the thread warp executes both branches sequentially, with the threads that do not meet the conditions becoming

inactive when that branch is reached. This is known as warp divergence and can have significant impacts on the performance of the model, as the different branches could increase execution time of the kernel for each warp.

The streaming multiprocessor is a fundamental component of GPU architecture that handles the parallel execution of many threads. It accepts these in the form of thread blocks, which are made up of multiple thread warps. A streaming multiprocessor can run multiple thread blocks at once and can be assigned more than it can execute simultaneously. To ensure the streaming multiprocessors are running at maximum efficiency, it is recommended to over-subscribe them (Wang, 2023), assigning more thread blocks than there are streaming multiprocessors, to increase the occupancy of the GPU and utilize as many of the GPU resources as possible.

A research paper investigates two techniques for improving the performance of both conditional and loop statements on the instruction level for a CUDA algorithm (Carrillo, Siegel, Li, 2010). These techniques primarily revolve around the idea of improving the occupancy of a kernel by reducing the register pressure of the control statements. The second technique attempts to combat warp divergence by ‘Branch Splitting’. This involves splitting the multiple branches of a conditional statement into their own individual kernels for execution. This reduces pressure on the register space allocated for the threads and increases occupancy of the kernel, that could result in a performance increase. In cases where register pressure is low while occupancy is high, the impact could be negligible and potentially negative if the repetition of the loop or the launching of multiple kernels simultaneously is more expensive than the performance gain.

While instruction level techniques can potentially provide a performance increase in certain cases, a simpler approach would be to avoid branching altogether where possible. An online CUDA lecture (Mills, 2024) explores the impact of several different optimization techniques on

kernel efficiency. One of the optimization case studies found in the lecture discusses the impact of warp divergence by implementing a simple kernel program in two separate ways; one with a conditional statement and one without. The case study found that by removing the conditional statement, the time taken to execute the kernel decreased by 66%. Although a single change is unlikely to have this large of an impact on more complicated kernels, the performance increase of avoiding branch conditions should not be overlooked and should play an important role in the layout of the kernels.

Other optimizations for GPU kernels include utilizing shared memory. This is found directly on the streaming multiprocessors, allowing quick data access for the individual threads. One downside to shared memory is that it is limited, meaning to use it appropriately for larger data sets, other techniques must be implemented. One paper investigates the effects of the tiling approach to utilize shared memory efficiently. Tiling involves splitting the data set across several different segments that can each fit within the limited shared memory space of the streaming multiprocessor. This study focuses on a simple matrix multiplication kernel, that can be processed into several tiles and placed in shared memory. The study found that the performance of the kernel increased by over 4 times when operating with 16x16 tiles, highlighting that effectiveness of shared memory usage. This can lead to large decreases in execution time, although in more complex cases the improvement would likely not be as significant.

2.3 - Summary

The methods explored in the GPGPU and boids field have impacted the decisions made regarding the optimizations used. The tasks can easily be implemented in a highly parallelable approach, as such using the GPU to implement boids seems like the optimal path. Other optimizations, such as avoiding branch conditions and implementing effective thread and block layouts to ensure proper occupancy of the streaming multiprocessors also seem appropriate due to the performance benefits gained, as such they will be built into the design philosophy behind the GPU implementation of the boids algorithm.

Chapter 3 - Methodology

To appropriately test different performance optimizations of the boids algorithm using GPGPU and advanced Direct X12 techniques, several different boids model iterations and applications were created. This section covers the development process from the initial Skateboard implementation to the final Direct X12 graphics application used for testing purposes, demonstrating how the core boids functionality evolved through development. This section will also cover the justifications for each iteration alongside the specifics of their implementation.

3.1 - Skateboard Application

The first implementation of the boids algorithm was initially created using the proprietary game engine Skateboard, that was developed by Abertay University. This version was the baseline used for all future iterations, as it appropriately handled all three boids rules; separation, cohesion and alignment, to allow the model to function appropriately.

The Skateboard engine was chosen for the initial boids implementation due to the engine providing a simple framework to begin development. It contained several libraries such as the OpenGL Mathematics library (Riccio, 2005) and in-built data types, such as transformation matrices. This decreased development time due to them being immediately available within the framework. Also, the engine had simple vertex and pixel shaders that instantly supported rendering objects within the scene, further decreasing development overhead. Another factor that impacted this decision was that the DirectX 12 graphics application programming interface (API) (Microsoft, 2015) was utilized, allowing for more control over memory and GPU synchronization that could be leveraged to improve performance.

3.1.1 - Entity Component System

The Skateboard engine utilizes an entity component system (ECS) to manage all objects within the world. This system works by changing the fundamentals of how objects are managed within game engines; the functionality of an object is split into separate systems that handle any appropriate logic, while object data is stored within components. Objects themselves are converted into handles, known as entities, that associate them with different data components. These components are then sent through different systems that apply functionality to them, instead of being operated upon directly by the objects themselves.

The main reason the ECS system is used for this boids implementation is due to the modularity and readability improvements associated with it. By decoupling the logic from the objects and encapsulating it within separate centralized systems, debugging becomes simpler. This is by design, as no system relies on another to function appropriately. Any bugs found within one system are more easily isolated, as there is no need to untangle any unrelated logic from the source of the issue itself.

Another reason the ECS system was chosen for this iteration is due to its potential performance improvements. The data components operated on by the systems can be stored contiguously in memory as values. This more efficiently utilizes the cache line when retrieving the data, as pointers do not always allocate memory contiguously, resulting in potential cache-misses.

Later iterations of the boids model do not use an ECS system. This is because the added development time of building an ECS framework from scratch was not outweighed by the minimal performance benefits gained. Despite this, the boids logic is still separated into different systems within future implementations. This seemed appropriate, as to apply the rules, knowledge of all boids within the model must be available. Storing this within each entity would quickly take up large amounts of memory and sending the data as parameters would become expensive, as the data

would need to be copied to each boid every frame to perform the rule calculations. As such, this data can easily be stored within a few centralized systems instead.

This also makes it easier to debug any logic errors, as the update functionality would be separate from rendering or any other key systems, as such any issues can be isolated more efficiently. Finally, reimplementing the update logic for any future boids model iterations also becomes simpler due to it being centralized, lowering the chance of bugs being introduced during this process.

3.1.2 - Initial Boids Application

The Skateboard boids implementation effectively utilizes the design philosophy behind ECS by splitting the rendering and physics functionality from the boids themselves, alongside all data associated with them. A third system is also present within the original model; the spawning system, which creates the entity handles and registers components with them. These components are then stored in arrays that are operated on by either the physics, rendering or spawning systems.

The three components associated with each entity handle are the movement, shape and transform components. These determine the direction, the vertices and indices, and the transformation of the boid, respectively. The transformation and movement components are primarily handled by the physics system, to position the boid entity and apply the rules appropriately, while the shape component is handled by the rendering system to effectively render them within the world. The pyramid component is inherited from the shape component, as a polymorphic implementation was developed to allow for the entity rendering system to determine which type of shape the vertices and indices should be assembled into.

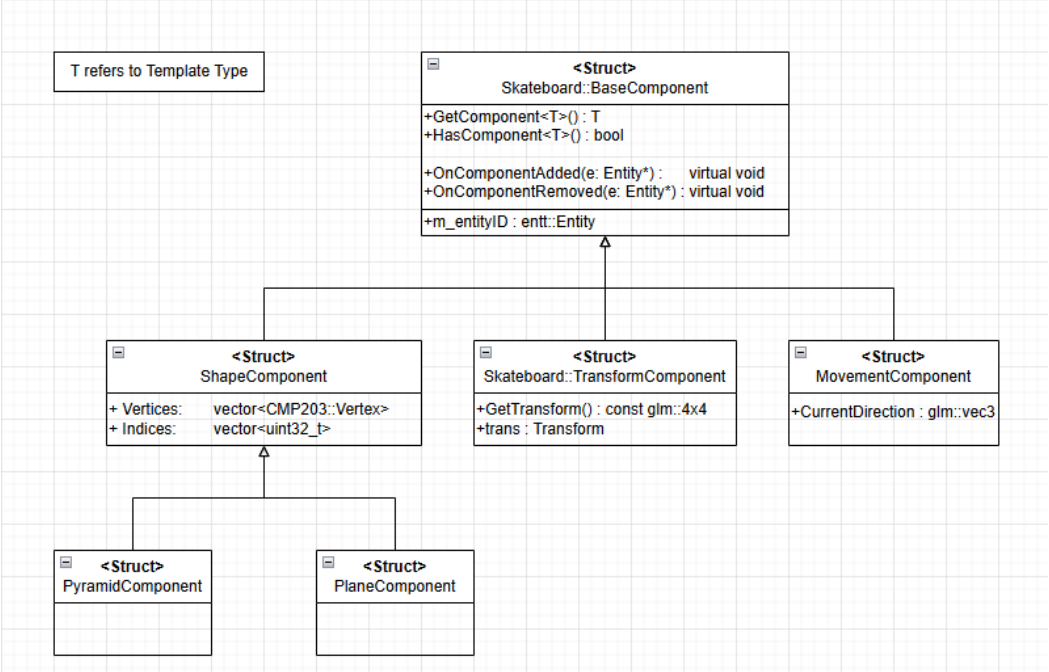


Fig 3.1 - UML diagram representation of boids components

The spawning system allows for the initialization of all boid entities by applying the appropriate components to each handle. The main methods; add entities, and remove entites, found within the spawning system, enable it to keep track of the memory allocated for entity handles, allowing them to be increased or decreased dynamically. This ensures external tools such as Imgui (Omar, 2025) can modify the number of boids present in the model.

The primary role of the physics system is to implement the three pivotal rules required for the boids model. These are applied during the update entities function within the physics system. This function begins by looping through every entity handle that exists within the model and retrieves the movement and transform components associated with them, then sends them through three separate functions that represent each of the boid rules. These functions also take in delta time as a parameter, using it to modify the final vectors that are applied to the boids themselves.

Each of the separate rule functions follow a similar flow; first, they loop through all boids within the model to access their movement and transform components, depending on the specific boid rule. Then, the functions check if any of the other boids are within range of the current one. If so, that function calculates a direction vector based on the specific rule and the weighted distance between them. The next step is to divide the resulting vector by the number of boids that have influenced it, generating an average vector. Finally, the result is normalized, then multiplied by delta time and the overall weight of the function to be applied to the current boid.

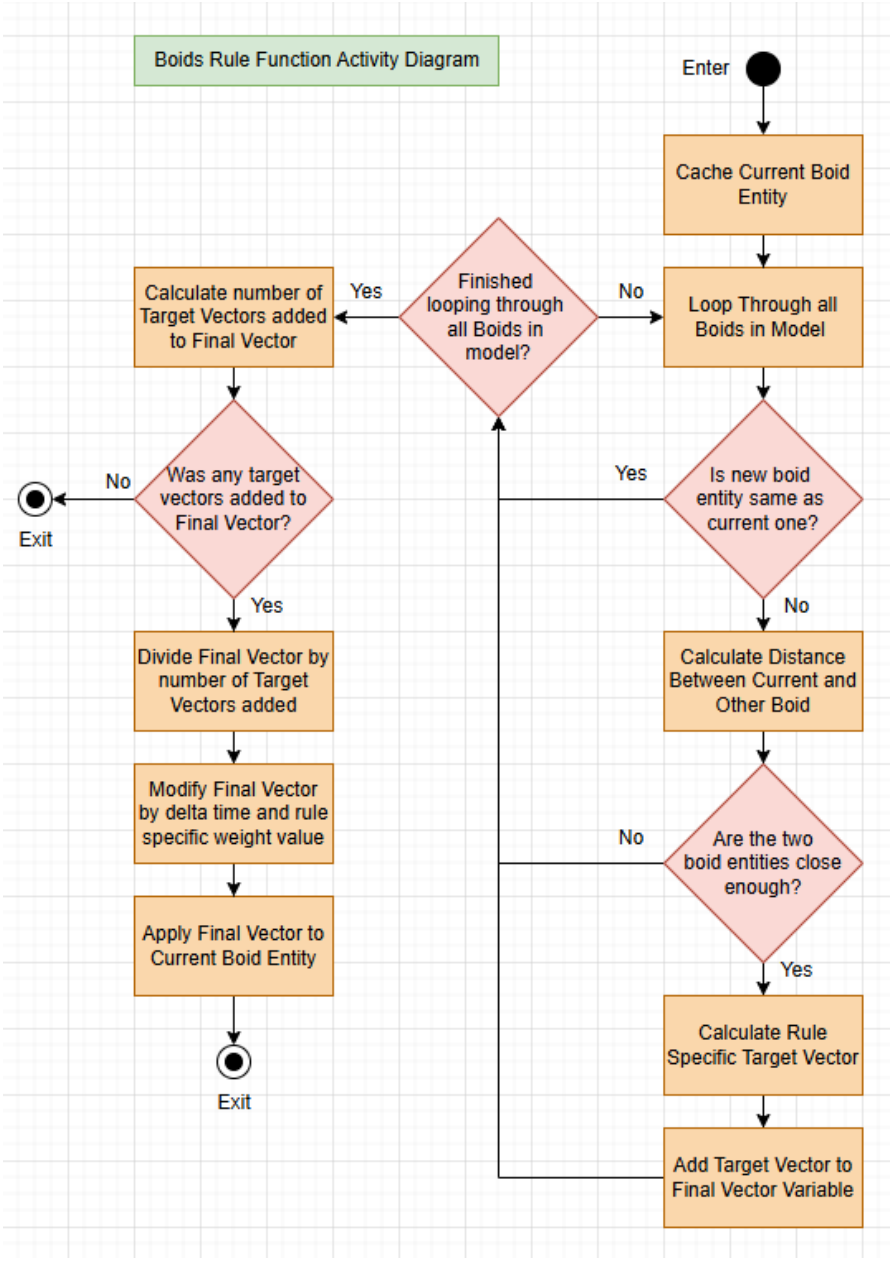


Fig 3.2 - Activity diagram representation of boids rule functions

The physics system is also responsible for limiting movement of the boids to within a small cube using Axis Aligned Bounding Box (AABB) collision detection. The usage of AABB was decided upon because the computational overhead of such a method is relatively cheap, ensuring high efficiency of the model, while forcing the boids to stay within the viewport. The method developed clamps the boids within the space and forces them to reflect off the walls. This bouncing effect can sometimes look unnatural; however, given the scope of the application, the efficiency

of this approach outweigh the minor loss in realism. This collision detection method is present in all future implementations. Alongside these previous two responsibilities, the physics system also moves and rotates them along their current direction vector, allowing them to move fluidly.

The rendering system allows the boids to be appropriately placed within the scene by sending the vertices and indices to the vertex shader. The shape used is based on the type of shape component associated with the entity handle. This was developed to allow for different shapes to represent the boids, however currently only the pyramid shape component is in use. This system also sends draw calls to the graphics API.

This implementation of the boids algorithm provided an overview of how the future implementations could be created, showcasing the splitting of logic into several systems and the handling of many different variables for each of the rules at runtime using ImGui. This served as an excellent proof of concept for the project, however there were several glaring issues with this version, which were later refined in future iterations of the model.

Originally, the three boids rules were split into separate functions within the physics system. This decision was made to improve readability. However, this unfortunately led to unintended performance consequences, due to the functions individually looping through all of the boids within the model. This led to an increase in the time complexity of the model from O(n^2) to O(3n^2), where n is the number of boids present in the model. At larger sample sizes, this performance decrease was significantly detrimental.

Another performance bottleneck of the implementation was also noticed in the the render system. All boid entities were rendered within their own draw call. This lead to thousands of draw calls per frame at larger entity counts, significantly lowering the performance of the model. Also, the

vertices and indices were updated every draw call, despite all boids being represented by the same shape. These rendering flaws were a result of not manually editing the vertex shader stage in the default graphics pipeline found within the engine.

The decision to not to fix these issues was made due to development time primarily being spent on creating a prototype model that functioned appropriately as a proof of concept, rather than a fully optimized version. As such, the primary purpose of this first model was still accomplished as the feasibility of the project was proven successful. Future implementations of the model steadily increased performance, refining the original model and providing excellent conditions for gathering results.

Future iterations of the boids algorithms were developed in a simpler DirectX 12 graphics application instead. One of the factors behind this decision was due to the Skateboard engine still being in development, leading to constant updates. These frequently changed the layout of the folders and other functionality within the engine which caused unintended bugs within the model. This required constant development time to ensure that it continued functioning appropriately.

Another factor behind this decision was that using the framework provided little fundamental knowledge on DirectX 12. This is because it is a complicated graphics API and without proper understanding of how it functions, it would be relatively difficult to appropriately utilize it. DirectX 12 usage within the Skateboard engine was complex and would require a large amount of development time to fully understand. As such, moving to a smaller graphics application provided a smaller framework to develop all future boids implementations. The application originated from an intricate tutorial series which methodically described the fundamentals of DirectX 12 and how they are used across several sample projects. These

provided the required knowledge base necessary to consider different approaches to improving performance using the DirectX 12 graphics API.

3.2 - DirectX12 Boids Application

The second application was developed from an in-depth tutorial series for a DirectX 12 renderer (Jeremiah, 2017). This framework provided the tools necessary to create a boids algorithm for both the CPU and the GPU, with access to the compute shader. The tutorial series also further explained the fundamentals of the graphics API in detail which laid the groundwork for future development approaches to be considered, notably the asynchronous compute method.

Several key features of DirectX 12 were utilized in the development process, including both memory management and GPU synchronization. These allow data to be efficiently transferred between pipelines and shader stages as well as allowing for appropriate coordination between the CPU and GPU, ensuring that the CPU waits for commands to finish executing before continuing. Synchronization is more heavily utilized in the async compute and GPGPU approaches to the boids algorithm, however it is still effectively used with the swap chain to ensure that all required commands have finished executing before swapping the current texture present on screen.

This application provides three different versions of the boids algorithm; one operating on a single thread within the CPU, one utilizing only GPGPU techniques and the final variant using GPU synchronization to allow the compute pipeline to run asynchronously with the render pipeline. These versions all provide a unique look into different optimization methods available for the boids algorithm, showcasing different results and allowing for unique conclusions to be drawn.

The core boids functionality present within the application is very similar in concept to the previous iteration. It effectively separates the logic into separate systems, following the systems part of the ECS design

philosophy. This ensured a sturdy foundation, as any bugs could be isolated and fixed appropriately. There are two systems currently present: the boids physics and rendering systems. Both are used by the CPU based implementation of the boids algorithm, however the physics system is reimplemented within the compute shader for the GPGPU versions, to appropriately utilize the many threads available within the GPU, though the overall functionality remains virtually the same.

3.2.1 - DirectX 12 Memory Management and Synchronization

GPU memory management is one of the primary features leveraged in this application. In DirectX 12, managing resources, including heap allocation and state transitions, becomes the responsibility of the programmer, allowing for more control and performance improvements if managed efficiently (Jeremiah, 2017). Individual GPU resources store data that can be operated upon by shaders within different stages of a pipeline state object. The GPU interacts with this data through descriptors, otherwise referred to as views, that are bound as root parameters in the root signature of a pipeline. These resources are required to be in specific states depending on the shader stage and the type of access required, such as read or write, before being utilized by the GPU.

Within this application, two types of descriptors are used: the unordered access views (UAV) and constant buffer views (CBV). Two UAV buffers are present in the application. These store boids data, however only one is used for the CPU and standard GPU versions, while both are required for the double buffering approach in the asynchronous compute implementation. Three constant buffers are required for the compute pipeline. These send the delta time, properties for the overall model including rule specific values, and the dimensions of the bounding box to the compute shader. Another one is required by the rendering pipeline to

send the camera view and projection matrices to the vertex shader for calculating the world view projection matrices on the GPU.

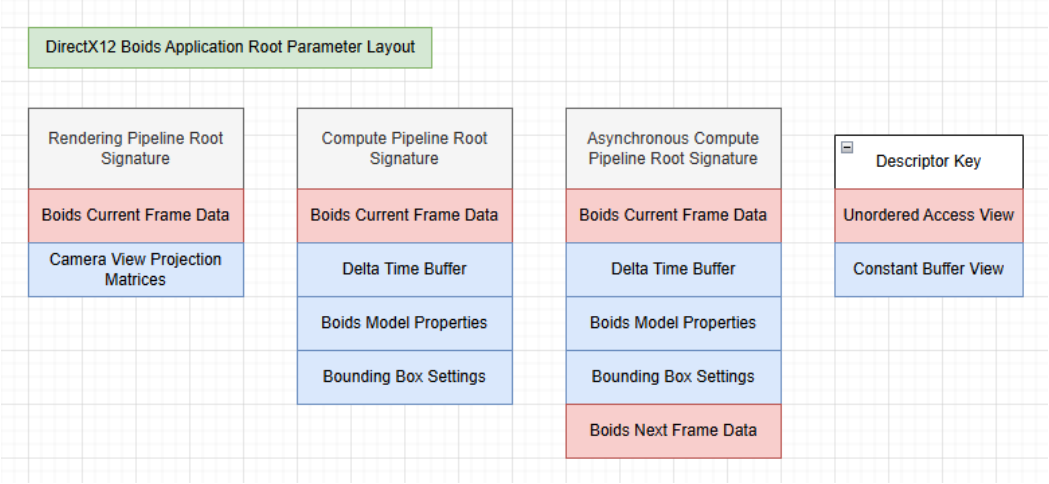


Fig 3.3 - Root parameter layout of different root signatures present within application

GPU synchronization is another primary feature leveraged within this application. In DirectX 12, this is also the responsibility of the programmer. To execute specific operations on the GPU, a command list must be used. Unlike DirectX 11, commands issued to command lists are not executed immediately (Jeremiah, 2017). Instead, commands are deferred until explicitly informed to execute on a command queue. Execution of these commands does not inherently block the main thread on the CPU, leading to potential issues if it relies on data generated by these GPU commands. Also, command queues can execute simultaneously, allowing for complete control of GPU synchronization that could lead to potential performance improvements if utilized correctly.

To properly synchronize the command queues, fences must be used. In DirectX 12, fences are synchronization objects used to halt execution of other GPU command queues or the CPU thread. They store a single integer value that reflects the last signal value sent to the fence. Signal commands are used to update a fence value. Command queues and the main CPU thread can be forced to wait for a specific fence value to be

reached, allowing for appropriate synchronization of GPU executions and the CPU. They can also be stalled until the next time the fence is signalled, for a simpler synchronization approach.

GPU synchronization is utilized throughout the boids application, primarily in the GPU and async compute variants. The standard GPU version requires the rendering pipeline to execute after the compute pipeline has finished writing to the UAV buffer, otherwise data could be read by the rendering pipeline that is currently being operated upon, leading to memory conflicts. The asynchronous compute version requires the rendering and async compute pipelines to finish executing before moving on to the next frame, otherwise the next frame's boids data could be incomplete before it begins rendering, leading to incorrect behaviour.

3.2.2 - Core Model Improvements from Initial application

The main improvements in functionality of the boids model from the previous Skateboard implementation are the rendering process and the rule functions. The rendering system was optimized to increase performance by lowering the number of draw calls needed to render the boids. The previous version required a draw call per boid, greatly increasing the performance cost of the rendering process. This version instead renders all boids within a single draw call, through mesh instancing.

To implement mesh instancing, the boids physics system sends the position and direction of all boids through a structured buffer to the vertex shader. The rendering system then sets the vertex and index buffer once, meaning that the mesh remains the same for all boids within the model. Finally, the vertex shader indexes into the structured buffer with the "SV_InstanceID" shader parameter, effectively determining the placement and rotation of each individual entity.

The other improvement for the model focuses on the core rule functions. Originally, the first application looped through all the boids per rule, leading to O(3n^2), where 'n' refers to the number of agents within the

model. This version refactors the core functionality to enforce each boid to loop through all others once, resetting the big O notation back to O(n^2). This allowed for appropriate results to be gathered, as inefficiencies within the core model itself could affect the final performance of all versions, since the core boids rule functionality remains the same between all three implementations.

3.2.3 - CPU Implementation

The CPU version of the boids algorithm within DirectX 12 was developed to easily accommodate future iterations, as such it utilizes an unordered access view to send data to the vertex shader from the boids physics system. As mentioned previously, the core functionality of the physics system for the CPU version changes slightly from the previous Skateboard model, only iterating through all the entities once per boid. This is implemented by changing the structure of the boids physics system from the initial Skateboard implementation; the new system now generates a target vector for each of the three rules from a single other boid entity, instead of having to loop through the entire array of boids for each rule. Essentially, the decision was made to sacrifice some readability in exchange for a large performance increase, as the calculation of the final vector for each rule is completed directly within the update loop itself, rather than in separate functions.

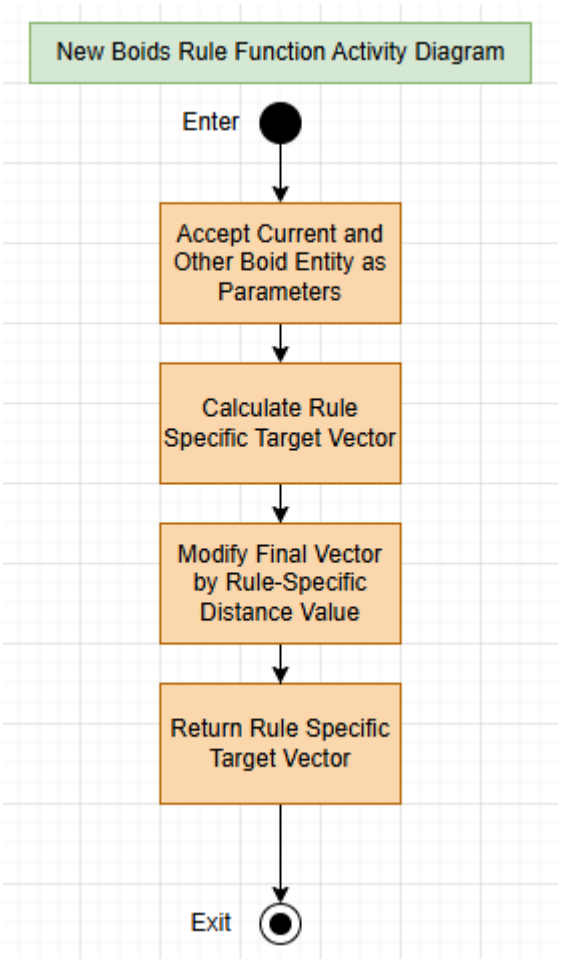


Fig 3.4 - Activity diagram representation of new boid rule functions, after extracting functionality from previous version

The update function within this iteration of the boids physics system is responsible for more of the calculations originally contained within the rule functions. First, it iterates through all boids within the model and caches the current boid, then iterates again caching the other boid. If the they have the same iterator, they are the same boid and as such, the other boid loop continues to the next boid in the model. Next, the distance between the current and other boid is calculated and if this is lower than any of the three specific rule's maximum distance, that rule function is called, passing the current boid and the other boid as parameters to

determine a target vector. The resulting vector is then added to the final vector for the specific rule.

The next section occurs after all other boids have been looped through, but still within the first loop of boids which the current boid is cached from. This section effectively divides the final vector by the number of resulting vectors added for each individual rule. Each of these final vectors are then multiplied by delta time and their respective rule's specific weight values, then normalized and added to the current boid's direction vector. This effectively forces the boids algorithm to perform with O(n^2) as each rule no longer individually loops through the entire array of boids, instead this functionality is handled by the update function itself. Finally, after all of the logic has been calculated, the boids data is then copied into an unordered access view (UAV) for use within the rendering pipeline. While the data could have been passed into a structured buffer using a shader resource view (SRV), the decision to use a UAV was made because it allowed for simple integration of future variants, as the buffer sent to the rendering pipeline also needs to be read from and written to within the compute shader, which is a feature that shader resource views do not provide.

A variant of the functionality found within the boids physics was placed within the compute shaders for the two GPU implementations. This ensured that the logic remained the same as the CPU approach, enabling appropriate testing while also keeping the associated performance improvements. As such, virtually all changes explained here also apply to those two variants. Due to this, the activity diagram below is applicable to all versions present within the DirectX12 application.

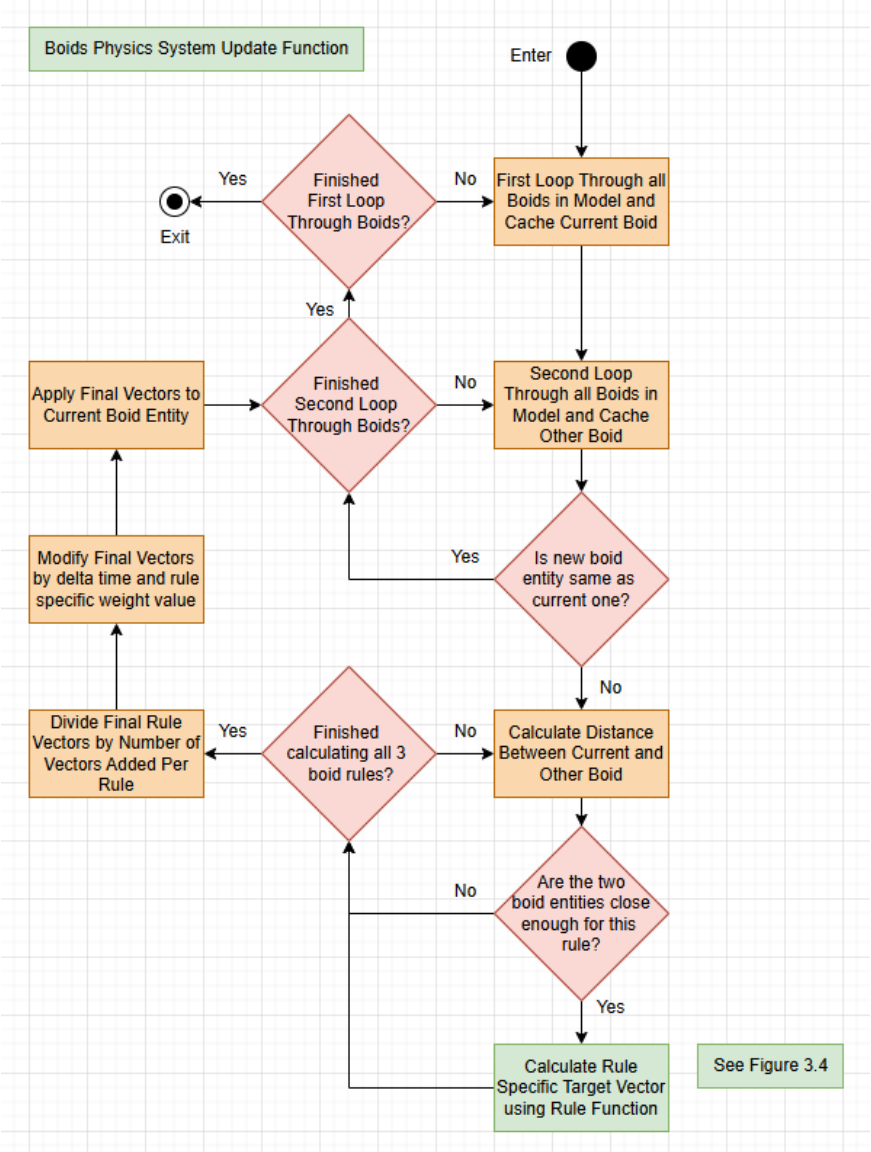


Fig 3.5 - Activity diagram representation of boid logic – applicable to all three versions of the model present in the DirectX12 application

3.2.4 - GPGPU Implementation

The GPGPU implementation of the algorithm was developed within the same DirectX12 graphics application. This version heavily utilized the compute shader to multithread the boids algorithm using the GPU to increase the overall performance of the model. The compute shader boids update logic is very similar to the previous implementation, except it

reads the data from an unordered access view buffer (UAV). This buffer is then written to after all rule functions have been calculated.

Data regarding the rule-specific weights, usually present within the boids physics system, is provided through a constant buffer within the compute shader. These values, alongside delta time, are then used to calculate the next position and direction for the boids. A group sync and memory barrier is placed after calculating these to ensure that the same UAV buffer is written to only after all other threads have read from it and have finished all previous operations up until this point. This same buffer is then sent to the rendering pipeline, to be read from and indexed into via mesh instancing.

To improve efficiency of the GPGPU model, all data relating to the boids themselves remains entirely within the GPU. The initial UAV buffer containing the boids data is first sent to the compute pipeline to be updated based on the rule calculations, then it is sent directly to the render pipeline avoiding any costly data transfers from the GPU to the CPU. Accessing the data on the CPU is not necessary, as the world view projection matrices for the boids are calculated inside of the vertex shader within the render pipeline, requiring only the direction and positions of the boids to place their vertices appropriately within the world.

The tutorial framework was consistently utilized throughout development of the application. Several of the classes provided were used extensively for this variant of the boids model. One of the framework classes used was the custom command list. This essentially acts as a wrapper for a regular DirectX 12 command list, providing additional functionality to make utilizing one simpler. This included the creation of an upload heap to allow dynamic updating of constant and structured buffer data, providing the ability for the CPU to update data that exists within the GPU. This was utilized by all versions of the boids algorithm for the camera view and projection matrices within the vertex shader in the rendering pipeline. These constant buffers were dynamically updated

during runtime, as they could change if the user transforms the camera. Dynamic uploading was also utilized by the CPU version of the algorithm to copy the updated boids data to the rendering pipeline.

Other functionality included the transitioning of certain GPU resources from one state to another. The method “SetUnorderedAccessView” initializes several transition barriers within the command list to ensure that any unordered access views are in the appropriate state before sending them to specific shader stages. This method ensured that the buffer resource would be in the appropriate stage for reading and writing when sent to the compute shader pipeline. A custom unordered access view buffer class was created to initialize a GPU resource and an unordered access view, so that the GPU can easily retrieve and modify the data stored within. This class is inherited from a custom buffer and resource class, found within the framework. These classes essentially provide the appropriate function signatures to initialize the resource and views properly and initialize views, so that the data can be accessed appropriately.

Another class utilized was the custom command queue. This class also acts as a wrapper, but around a standard DirectX 12 command queue instead of a command list. This class provides a fence object and the ability to process several command lists simultaneously. It also provides several methods for using that fence object and posting signals after completing execution of all command lists. For the development of this version, two different command queues were required. Each queue processes a single type of command list. These command list types determine what GPU operations can be executed.

While both compute and direct command list types can execute a dispatch command required for execution of the compute pipeline, the custom command queue class only implements a single fence object. Because of this, future async compute implementations would be difficult to manage, as there would be no way to differentiate between when the dispatch or render command lists have finished executing. Therefore, the

decision was made to create two separate command queues with their own types of command lists, with each having their own separate fence object. This ensured that each command queue is given only a single responsibility; the render pipeline, or the compute pipeline, making it easier to implement a future asynchronous approach due to it being more explicit when either pipeline has finished execution.

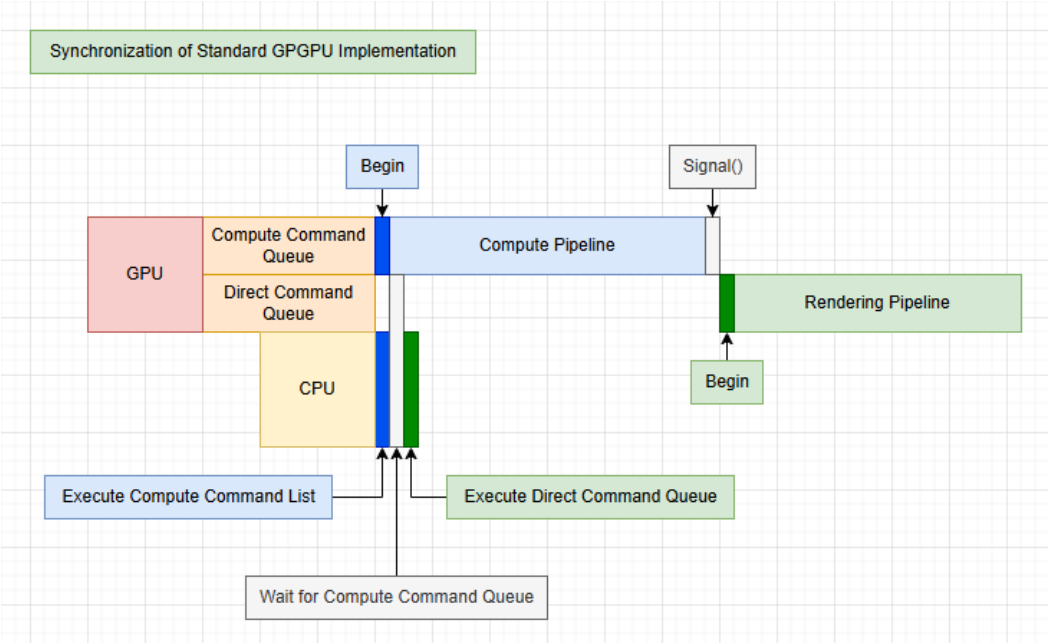


Fig 3.6 - Diagram of GPU synchronization for standard GPGPU boids implementation.

The lines next to the CPU represent individual commands sent to the GPU; showcasing how it plays a direct role in synchronization

3.2.5 - Asynchronous Compute GPGPU Implementation

The asynchronous compute version was developed similarly to the previous GPGPU implementation. This variant also leveraged the multithreading capabilities of the GPU to increase performance of the boids model, however it is taken one step further by enabling both the compute and render pipelines to execute simultaneously. This parallelism theoretically enables better allocation of GPU resources, increasing

occupancy and minimizing thread downtime to increase the overall performance of the boids model.

The compute shader used in this approach follows a different design pattern to the synchronous GPU implementation; the current boids are used to calculate the next frame of boids data. Instead of calculating the current boids data then rendering it sequentially, this approach enables the compute shader to calculate the next data while the graphics pipeline renders the current frame's boids data.

To accomplish this, two UAV buffers are used within the async variant of the compute shader. The first buffer stores the current frame of boids and is read-only, while the second buffer stores the next frame's boids data and is write-only. This avoids the same UAV buffer being read from and written to within the compute shader, as this would result in memory conflicts with the rendering pipeline. This could occur if it tries to read data that is being written to by the compute shader in the same UAV buffer, hence the separation of the read-only and write-only buffers.

These two UAV buffers are stored within an array of pointers on the CPU side. This setup allows them to be swapped every frame, so the next buffer becomes the current one for rendering, and previous buffer becomes the next one for the compute pipeline to write to. The timing of the swap is crucial, as it must occur after the compute and render pipeline have finished execution. If the swap occurs earlier, then the next frame's data will not have been fully calculated, leading to incorrect behavior of boids themselves, or the current boids will not have been rendered correctly. This double buffering approach enables proper synchronization of the boids data across the two pipelines, despite them operating in parallel.

Similar to the previous iteration, the asynchronous compute method utilizes two separate custom command queues to execute both the compute and render pipelines. Both queues implement their own fences, allowing for easy differentiation between when the render and compute

pipelines finish executing their respective GPU operations. Initially, both pipelines are executed in parallel, however once the rendering of the boids has completed, the render pipeline pauses execution of any further instructions until the compute pipeline has finished calculation of the next frame's boids data.

To accomplish this, the direct queue, which executes the render pipeline, waits upon the fence stored within the compute queue. Immediately after execution has finished, it sends a signal to its internal fence. This enables the direct queue to recognize when the compute queue has completed its operations, allowing for it to continue execution of any further commands, such as presenting the back buffer to the screen. This synchronization approach ensures that the next frame does not begin rendering until all boids data for it have been calculated by the compute shader.

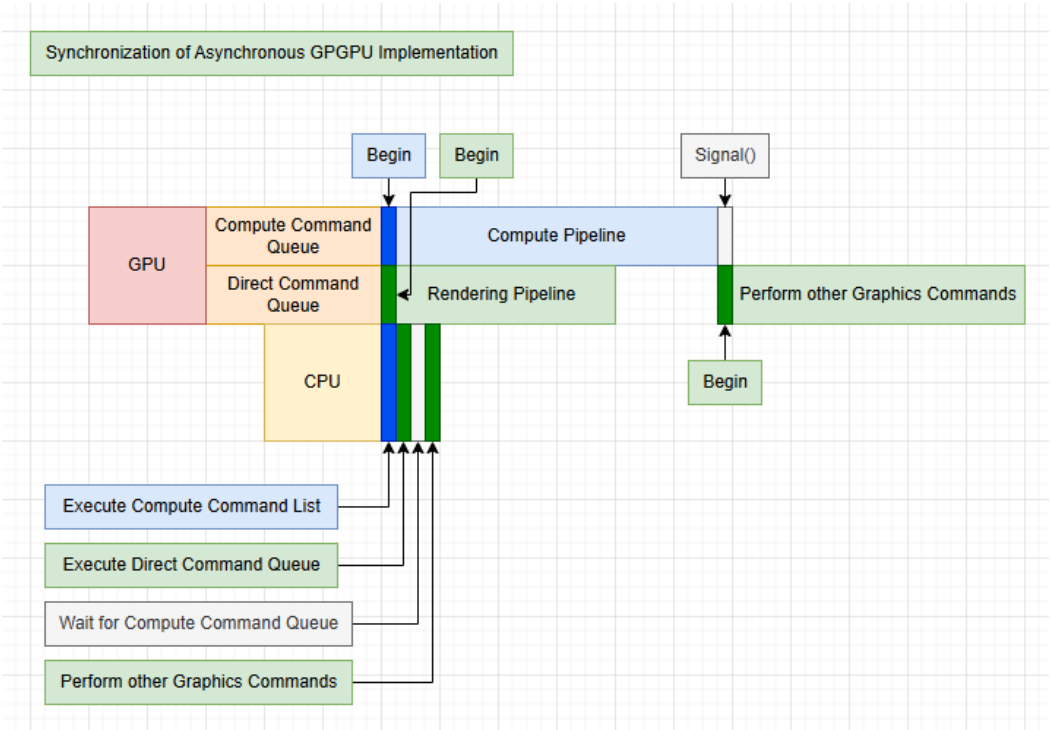


Fig 3.7 - Diagram of synchronization for asynchronous GPGPU boids implementation

Similar to previous synchronization figure, lines next to CPU represent instructions for GPU, while two distinct green lines on Direct Command Queue represent the pause and continuation of GPU operations.

Chapter 4 - Results

Testing was performed by measuring the execution time of both the render and compute pipelines, as well as the boids physics system when gathering results on the CPU version. Results were also gathered regarding the execution time of the entire frame to determine the overall performance impact. All testing was performed entirely on the second application, as all three iterations; CPU, GPU and asynchronous compute were present, allowing for an easier gathering process. Results were gathered using a NVidia 3060ti graphics card, alongside an i7 12700f processor.

4.1 - Results Gathering Process

To appropriately gather result data regarding the execution times of the two pipelines, two query heaps were used. Query heaps are GPU resources used to store performance or analytical data regarding the execution state of command lists. By placing queries between GPU operations, data can be gathered and analysed on individual commands that are executed on the GPU.

After this, the query heaps need to be resolved, which involves writing their data into a readback buffer, for the CPU to access. This must occur after the resolving process has finished, otherwise the data stored in the readback buffer may be inaccurate or incomplete.

Several types of queries can be chosen to query the command list. The decision to use timestamp queries was made because it provides accurate timing results for individual GPU operations. These timestamps are generated as soon as the GPU is finished with any proceeding workloads. This allows for appropriate timings to be measured by determining the difference between the latest and the previous timestamps, depending on the frequency of timestamps generated by the GPU. This calculation to determine the timing in milliseconds is

performed by the calculate GPU query time method within the main “tutorial3” class.

These query heaps effectively measure the time taken by the GPU to execute both the render and compute pipelines for both GPU implementations. The CPU version also utilizes a query heap for the render pipeline; however, it uses a far simpler timing method for the boids physics system update. As this function is within the CPU, a standard library clock can be utilized to determine the time difference between the beginning and end of the update method.

The decision to use query heaps was made because using standard library clocks to measure the performance of the pipelines would require splitting the GPU operations into separate execution calls for the command queue. This would incur a performance cost, as it would require two separate command list submissions, instead of combining the allocation of resources and execution for a pipeline into one. This additional overhead would affect the overall frame time, lowering accuracy of any results gathered.

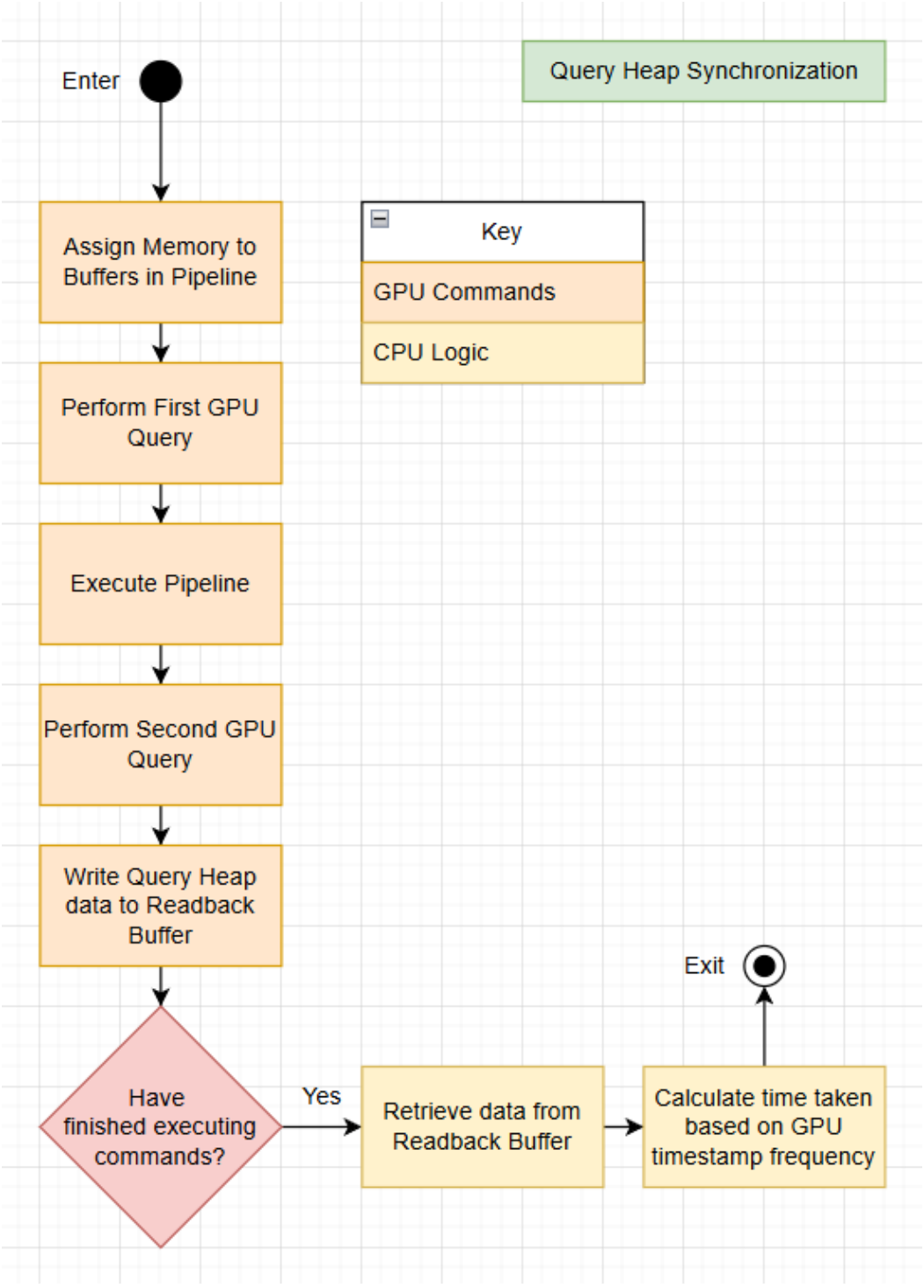


Fig 4.0 - Activity diagram showcasing appropriate Query Heap Synchronization

Three different types of tests were performed to analyse the performance impact of specific optimizations on the boids algorithm. Two of these focused on only the GPU and asynchronous compute versions, while one includes the CPU iteration alongside them. Different variants of testing

were achieved through the implementation of ImGui. All results gathered were in milliseconds (ms).

The method of gathering results involved recording the time taken per frame for either the entire application or an individual pipeline to a vector. Then, an average is generated based on the elements of the vector. This average is then sent to a separate queue. The custom variable “m_AmountOfCaptures”, modifiable within the ImGui user interface, determines how many averages will be generate for the queue, determining how long results are gathered for. After generating these results, the elements present within the queues are individually popped from them and printed to a text file, to be read by the end user. This process can be partially automized by selecting the “Automatic” radio button under the “Capturing Results” section. However, “Stop Capture” still needs to be pressed manually to print the results to a text file.

4.2 - Finalized Results

4.2.1 - General Performance Results

The first test was performed on all three algorithms to determine the performance difference between variants. This test gathered results on varying numbers of boids, ranging from 500 to 5000. All test cases utilized 128 thread group sizes, as the second set of results determined this size to be the most efficient in many cases. The CPU frame results were split from the GPU and async graph due to the significant difference in scale. Without this, the results would be difficult to properly demonstrate within a single graph.

Also, the render time differences between the two GPGPU implementations were minimal, as such the decision was made to only include the render time performance of the GPU model in Figure 4.4.

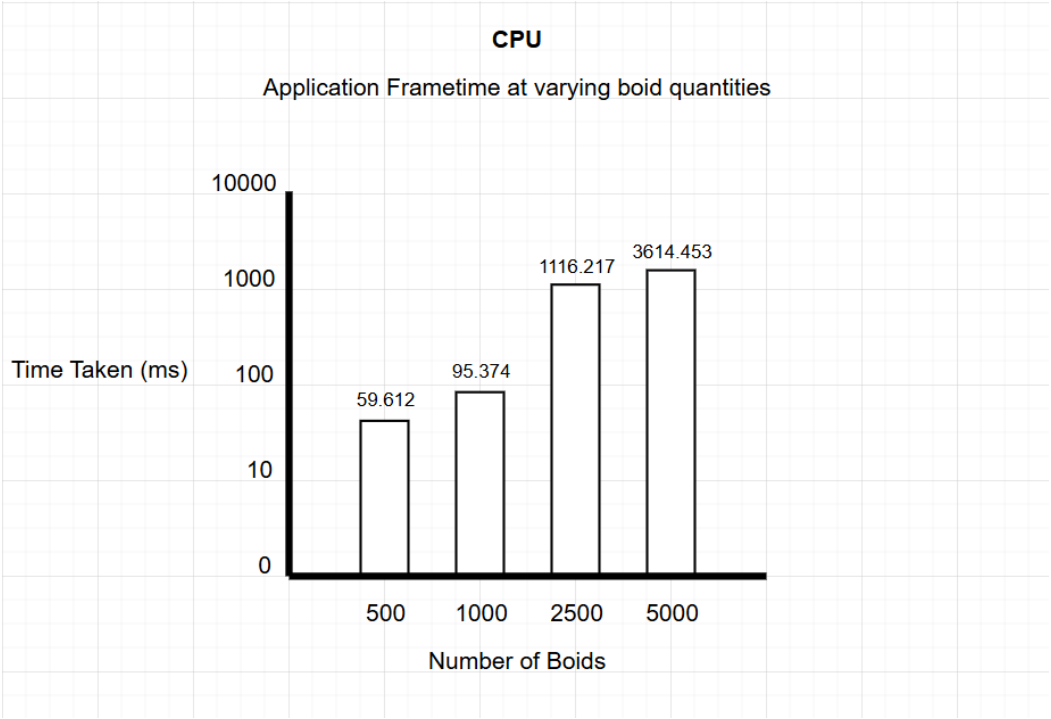


Fig 4.1 - Logarithmic graph of average time taken per frame for CPU boids implementation

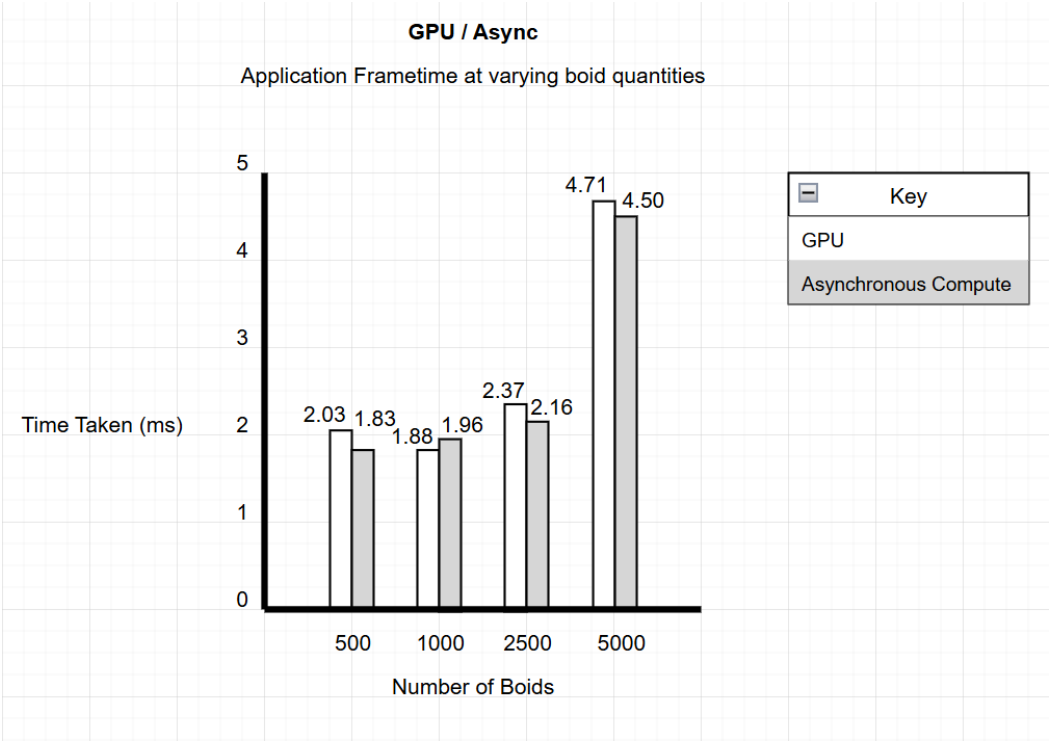


Fig 4.2 - Comparison of average time per frame between GPGPU and Asynchronous Compute boids implementations

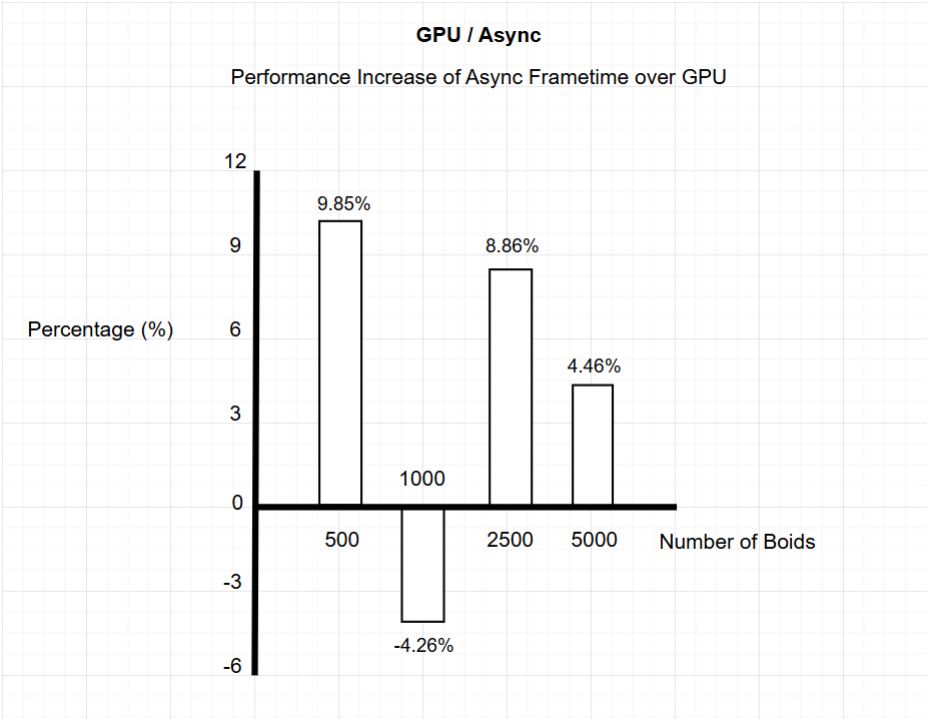


Fig 4.3 - Average performance increase of Asynchronous over standard GPU implementation

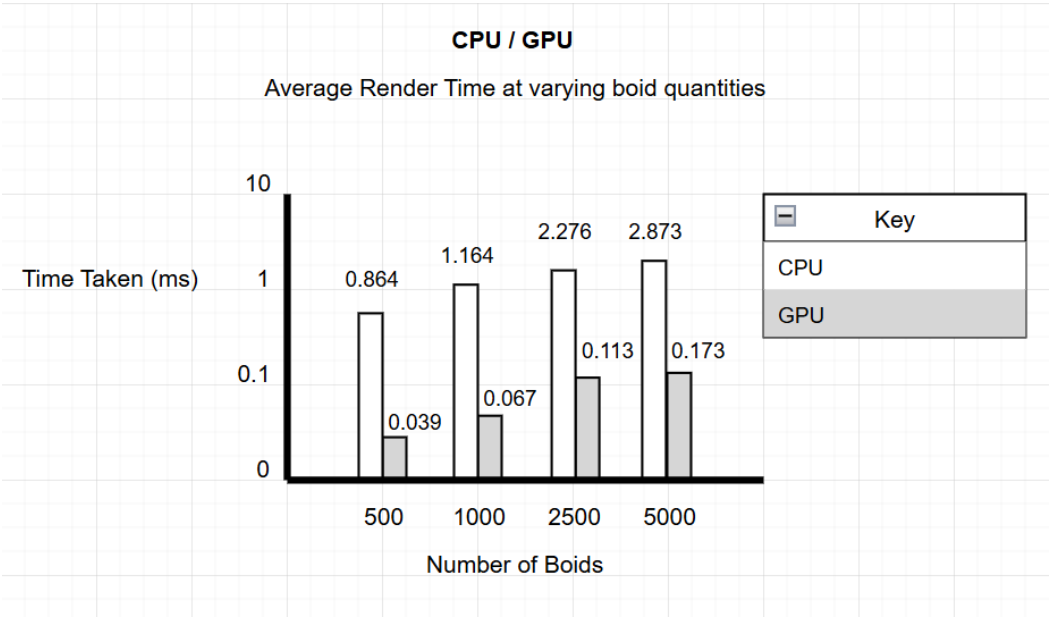


Fig 4.4 - Logarithmic graph of comparison between average render times per frame for CPU and GPU boids implementations

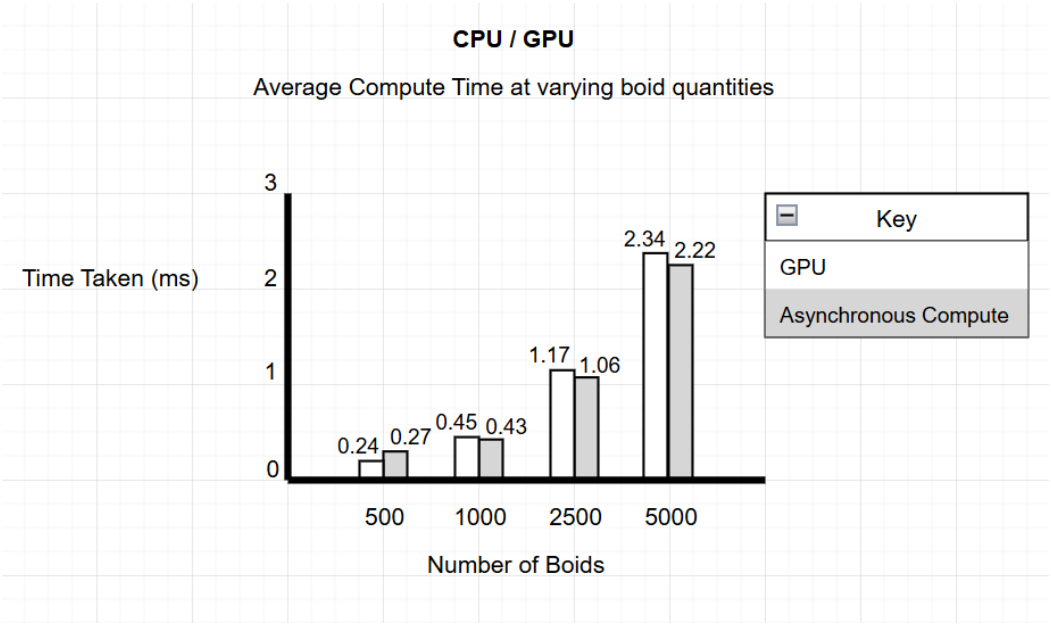


Fig 4.5 - Comparison of average compute time per frame for both GPU and asynchronous compute boids implementations

4.2.2 - Thread Group Size Results

The second test focused on the impact of different thread group sizes on the performance of the GPU model. The sizes were 128, 256, 512 and 1024. These were decided upon were based on previous literature that demonstrated the differences between them, allowing for a potential comparison to be drawn between the two studies (Falconer, 2015). Different sample sizes were used for testing, ranging from 2500 to 50000. These amounts allow for the most effective thread group size to be measured with varying degrees of stress on the GPU. All threads are in an [X, 1, 1] layout, where X is the number of threads.

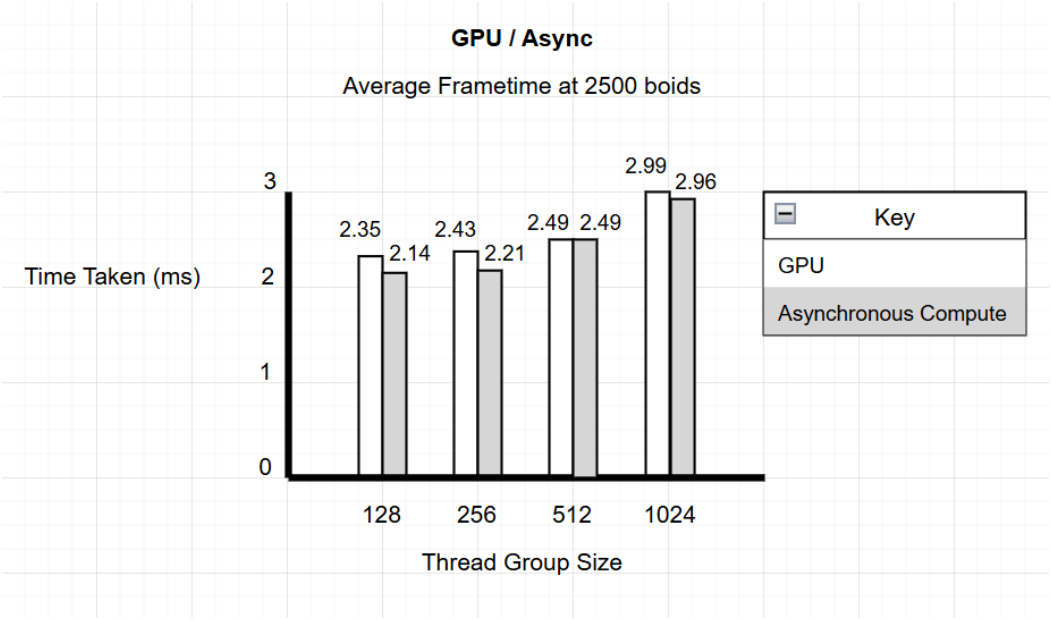


Fig 4.6 - Comparison of thread group size performance on computation time of the GPU and asynchronous compute boids models at 2500 boids

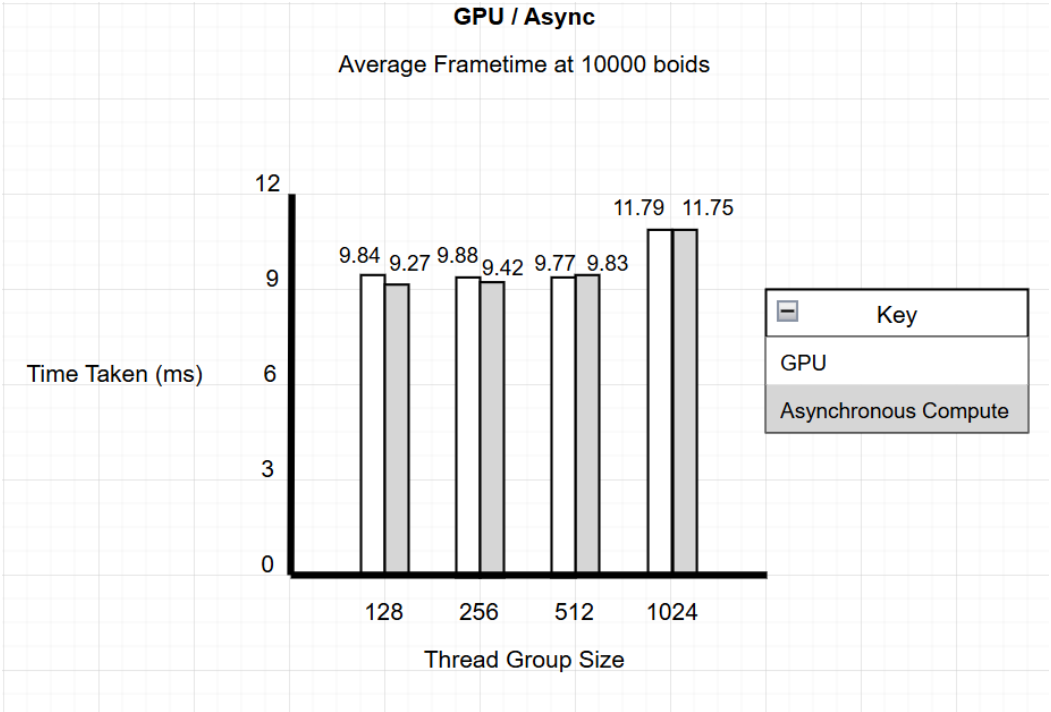


Fig 4.7 - Comparison of thread group size performance on computation time of the GPU and asynchronous compute boids models at 10000 boids

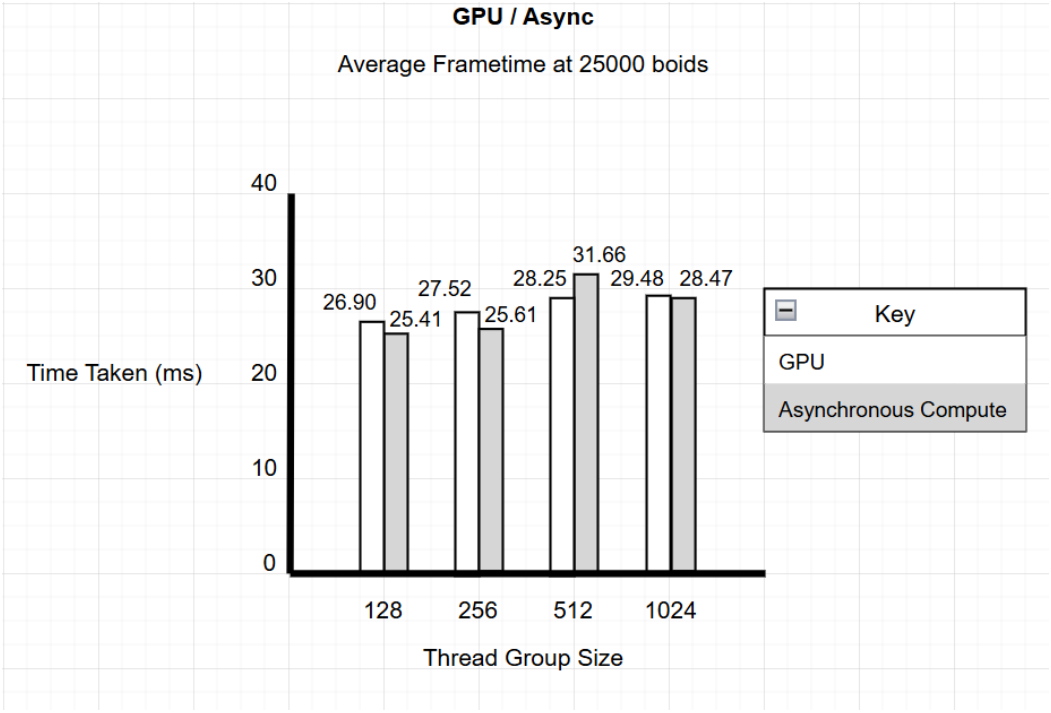


Fig 4.8 - Comparison of thread group size performance on computation time of the GPU and asynchronous compute boids models at 25000 boids

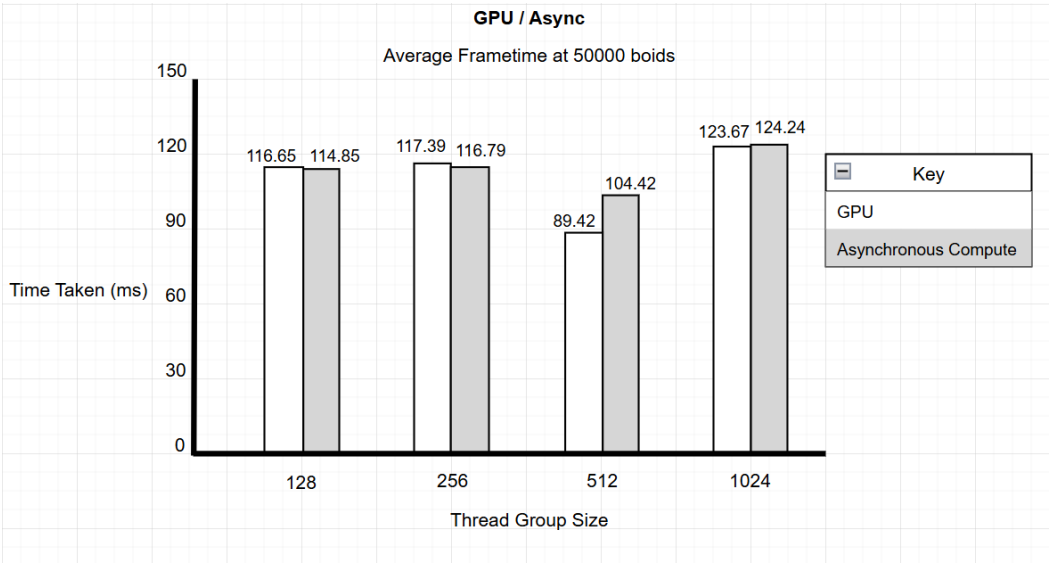


Fig 4.9 - Comparison of thread group size performance on computation time of the GPU and asynchronous compute boids models at 25000 boids

4.2.3 - Large Scale Performance Results

The final test demonstrated the performance difference between the GPU and asynchronous version at a larger scale, with increasingly large sample sizes up to 100000. All test cases utilized 128 threads per group.

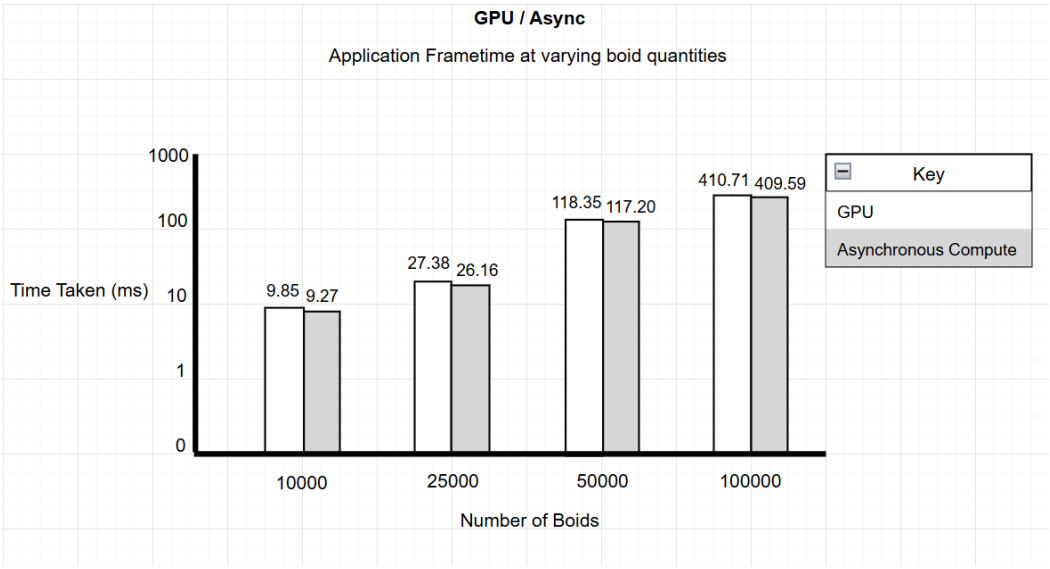


Fig 4.10 - Logarithmic graph of average time taken per frame for GPU and asynchronous compute boids implementations

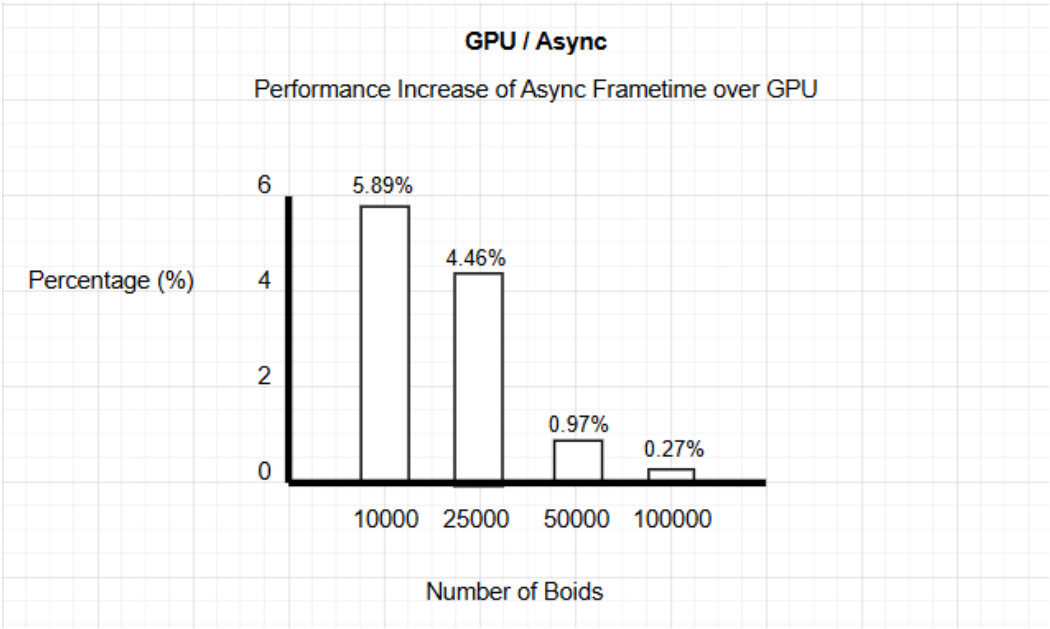


Fig 4.11 - Average performance increase of Asynchronous over standard GPU implementation

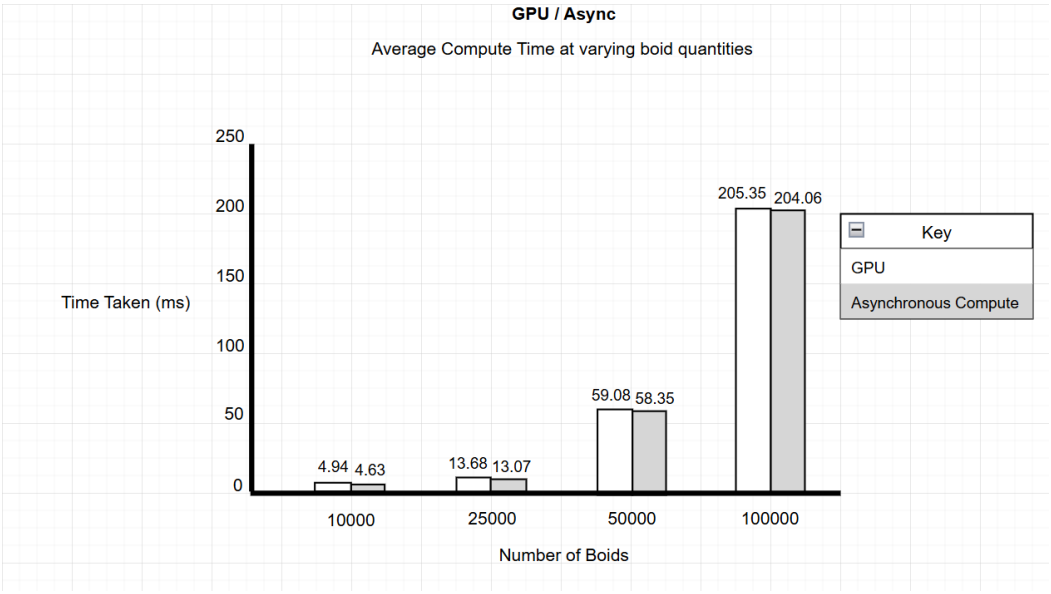


Fig 4.12 - Comparison of average compute time per frame for both GPU and asynchronous compute boids implementations

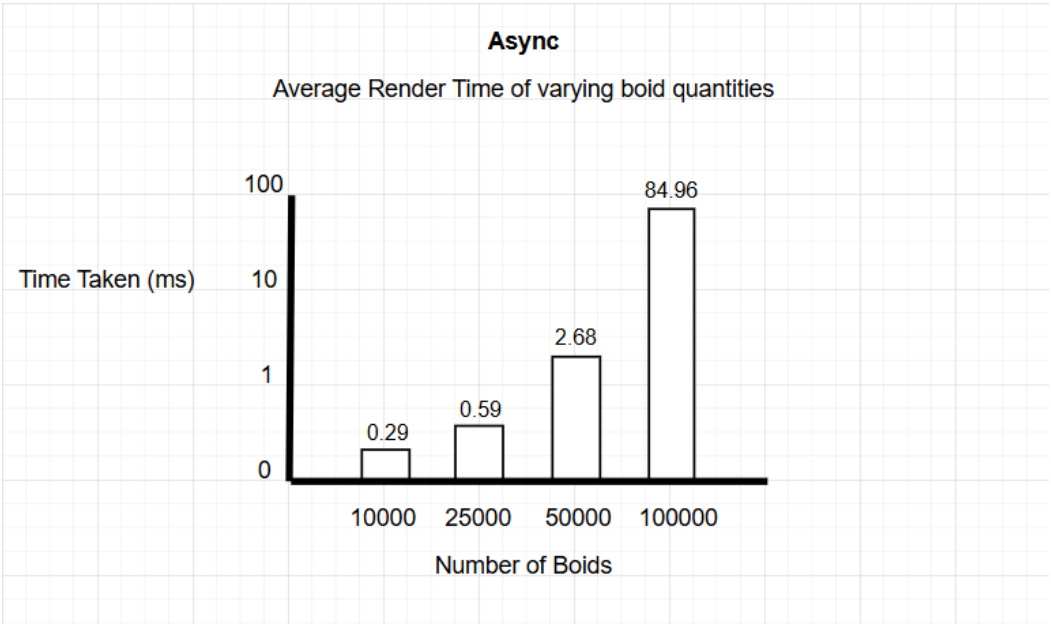


Fig 4.13 – Logarithmic graph of average render time per frame for asynchronous compute boids implementation

Chapter 5 - Discussion

This section provides an in-depth exploration into the reasons behind each of the results gathered from testing, to provide an answer to the research questions proposed in the introduction. The application is also analysed and critically evaluated to determine how effectively it aligned with the overall aims of the project.

5.1 - General Performance Analysis

The first series of tests focused on the comparison between the CPU, GPU and asynchronous implementations. These results proved that the use of GPGPU for the boids application significantly improved performance, by efficiently utilizing the parallel capabilities of the NVidia 3060ti graphics card. The CPU version acted as an excellent control, allowing the impact of the test results to be properly highlighted, showcasing the effects of offloading the boids logic to the GPU and the differences between the two GPU implementations.

5.1.1 - CPU Implementation Analysis

The CPU results present in Figure 4.1 represent the overall frame time performance of the application at different volumes of boid entities. At 500 boids, the average frame time for the CPU approach was 59.61ms, as opposed to the 2.03ms taken for the standard GPU method. As more boids were added to the model, the average frame time for the application also increased drastically. Notably, the increase was not linear; this is due to the naive boids algorithm operating with O(n^2), where n represents the number of boids. As more are added, additional calculations are required for all previous entities within the model. This leads to an exponential increase in operating time, impacting all approaches within the application; however, it impacts the CPU version

more heavily, due to its single threaded naive approach to updating the boids.

Individual CPU threads are more powerful than those present on the GPU, however they are also far less numerous. They are designed for high single thread performance and optimized for general purpose usage (Wang, 2023). However, the cost of switching between different tasks is higher than that of GPU threads, which are far more efficient at performing smaller mathematical operations. Due to the smaller size of GPU threads, their switch cost is significantly lower, enabling them to work more effectively with workloads that benefit from high parallelism, such as the boids model.

The render time results present in Figure 4.4 represent the differences in time taken for the render pipeline between the two GPGPU implementations and the CPU one. Across all boid entity counts, the CPU render time was significantly longer than the standard GPU implementation. At 500 boids, the render time for the CPU version was 0.86ms, as compared to the 0.039ms for the GPU version. This is because the CPU render time also includes copy times from the boids data stored on the CPU to the UAV buffer present within the render pipeline.

This CPU to GPU copy time provides excellent proof for the decision to keep the boids data entirely within graphics processing unit for the two GPGPU implementations. This avoided a costly GPU to CPU transfer, to calculate the world view projection matrices on the CPU, then another to move those matrices back to the GPU. Avoiding these two memory transfers likely decreased the overall frame time of the application for the GPGPU implementations.

5.1.2 - GPU and Async Implementation Analysis

The results of the comparison between the two implementations, present in Figure 4.2, represent the overall frame time of the application when executing each implementation at differing sample sizes. At 500 boids,

the average frame time for the GPU implementation was 2.03ms, as opposed to the 1.83ms time taken for the asynchronous approach. This highlights the effectiveness of executing the render and compute pipeline in parallel; resulting in a direct performance increase of 9.28%, visible in Figure 4.3. This far outweighs the individual performance gain of executing the rendering pipeline synchronously, as hypothetically the only time gained would be the rendering time, at 0.039ms, as seen in Figure 4.4. This is likely due to an increase in occupancy of the GPU streaming multiprocessors, with more thread warps being executed at once and less downtime between thread warp stalls, hiding latency (NVidia, 2015). Also, GPU idle time between finalizing one pipeline and executing the next is lowered, as both pipelines operate simultaneously.

The sample size used to demonstrate the performance difference between the GPU and asynchronous compute methods was chosen in an attempt to compare them with the CPU implementation. However, both versions completely outperformed the single threaded CPU approach in every case. At 5000 boids the GPU frame time averaged 4.71ms, while the CPU frame time averaged 3614.45ms, showcased in Figure 4.2 and 4.1 respectively. This demonstrates the overwhelming performance gain of utilizing the parallel capabilities of the GPU for the boids algorithm. This limited the overall size of the first set of test results to 5000 boids, due to the CPU model having difficulties executing, even though the two GPGPU implementations could handle significantly larger sample sizes.

The equation to generate the percentage performance increase for Figure 4.3 and 4.11 is visible below. “AsyncTime” refers to the overall frame time

of the asynchronous method, while “SyncTime” refers to the frame time of the standard GPGPU implementation.

$$Percentage\ Gain = \frac{AsyncTime - SyncTime}{SyncTime} \cdot 100$$

At 5000 boids, the overall frame time of the application increases to 4.71ms for the standard GPU implementation, as opposed to 4.50ms for the asynchronous approach. This highlights the effectiveness of the async approach dwindling, as the direct performance increase is 4.46%. This is less than half of the percentage at 500 boids, showcasing the decline in relevance of the asynchronous approach. However, at 2500 boids, the performance increase is still a respectable 8.86%, demonstrating that the async method is more efficient at lower sample sizes. Despite this, it still outperforms the standard GPU approach across many sample sizes, even as its percentage increase declines heavily. At 1000 boids, there is an overall performance decrease of –4.26%. This is likely due to a limited volume of results gathered, as every other sample size revealed a performance increase.

The higher percentage increase in performance at smaller sample sizes is likely due to the compute and render times being narrower. At 500 boids, the average time taken for the render pipeline was 0.039ms and the compute pipeline was 0.27ms. This results in a performance gap of only 0.231ms between the two pipelines. At 5000 boids, the time taken for the render and compute time was 0.173ms and 2.22ms respectively, resulting in a widening performance gap of 2.047ms. This is nearly 10 times the size of the 0.231ms of the 500 boids. This demonstrates that as the number of entities in the model grows, the importance of the rendering pipeline executing asynchronously decreases. This is due to the relative size of the render pipeline lowering in comparison to the

compute pipeline, leading to an overall decline in the performance increase of the asynchronous compute approach.

One of the factors impacting render time is mesh instancing. This optimization allows for all boids within the model to render within a single draw call, by indexing into a structured buffer using the variable “SV_InstanceID” inside of the vertex shader. Without this, it is possible that the asynchronous approach could perform better than the standard GPU version, however the overall performance of the application would be lower for both implementations due to each individual boid requiring their own draw call.

The comparison between different computation times of the two GPGPU implementations was also gathered. At almost all sample sizes, despite both implementations utilizing almost identical compute shaders, the asynchronous compute shader executed faster. At 5000 boids, the average computation time for the standard GPU implementation was 2.34ms, as opposed to 2.22ms for the asynchronous approach, resulting in a 5.13% performance increase. This is likely due to the standard compute shader utilizing a memory barrier with group sync, which effectively blocks execution of all threads until all memory accesses have completed, and they have all reached this function call. This is required when using a single UAV buffer, as write and read access could occur simultaneously, resulting in memory conflicts.

The asynchronous compute shader avoids this by design, as it reads from one UAV buffer, while writing to a second one. At 500 boids, the standard GPU computation time is lower than this one, at 0.24ms as compared to 0.27ms. This is likely due to the overhead of managing two separate UAV buffers. This is usually not a performance hinderance at

most sample sizes, however it is possible that due to the significantly small sample size, it becomes visible.

5.2 - GPGPU Thread Group Size Analysis

In the second set of results, showcased in Figures 4.6 to 4.9, the difference in frame time between the two GPU implementations of the boids algorithm at various thread group sizes was recorded. In Figure 4.6, At 2500 boids with 128 threads per group, the overall frame time for the GPU version was 2.35ms. At the same boid count, with 1028 threads per group, the time was 2.99ms. This implies that utilizing numerous small blocks allowed for an increase in occupancy of the GPU. This results in better GPU utilization, as more blocks were executed simultaneously across the streaming multiprocessors, resulting in a performance increase.

Across most sample sizes, the smallest thread group was the most efficient, at 128 threads per group. This is likely due to the increased number of blocks allowing for a higher occupancy rate of the GPU, utilizing its parallel capabilities more effectively. At 50000 boids in Figure 4.9, the 128-thread group size averaged 116.65ms. This is faster than both the 256 and 1028 sizes, as the frame times were 117.39ms and 123.67ms respectively. However, the 512-thread group size performed significantly better, at an average frame time of 89.42ms.

This is due to the underlying hardware that was available on the testing machine. The 3060ti graphics card was the primary GPU utilized, having access to 38 streaming multiprocessors (Moass, 2020) with a theoretical maximum of 2048 threads executing simultaneously. At 50000 boids, with 512 threads per block, the GPU implementation creates 98 blocks.

When assigning thread blocks, it is best to provide more than one block per streaming multiprocessor (NVidia, 2023). By oversubscribing them with more thread warps than can run simultaneously, overall performance can be increased through hiding latency (NVidia, 2015). Latency occurs

when threads are waiting for memory access or otherwise stalled, potentially due to thread sync barriers. In this case, if other thread warps are waiting for execution and are not stalled, the warp scheduler within the streaming processor can switch contexts to them, effectively hiding latency by constantly operating on non-stalled threads. However, if the number of blocks far exceeds the number of streaming multiprocessors, the opposite effect could occur, as the context switching overhead between blocks could become heavier, resulting in lower overall performance.

In the case of 50000 boids executing with a 512-thread group size, there are approximately 2.6 blocks per streaming multiprocessor. Since this results in just under maximum theoretical capacity of the streaming multiprocessors on the 3060ti, this an ideal number of blocks, allowing for increased occupancy and decreased idling time due to latency. This explains why the thread group size of 512 performs more efficiently at this sample size.

Notably, the asynchronous approach was outperformed by the GPU implementation at 50000 boids, as the overall frame time was 104.42ms. This is likely due to the render pipeline adding more thread groups for the streaming multiprocessors than they can execute simultaneously, decreasing performance.

5.3 - Large Scale Performance Analysis

The final set of results were gathered to analyse the performance of the asynchronous compute against the standard implementation with increasingly large sample sizes. The decision to test at these sizes was made due to the rapid performance decline measured in Figure 4.3. At 10000 boids, the lowest sample size in Figure 4.10, the average overall frame time of the application was measured at 9.85ms for the standard implementation, as opposed to 9.27ms for the asynchronous approach. This resulted in a performance increase of 5.88%, visible in Figure 4.11.

This is similar to the 5000 boids count percentage in Figure 4.3, at 4.46%.

Despite the higher sample size of 10000 boids, the overall percentage increase is slightly higher than the previous size. This is likely due to a limited number of results being gathered, though it could also be attributed with better occupancy of the streaming multiprocessors, by assigning larger numbers of thread groups.

At 100000 boids, the overall frame time of the application for the asynchronous approach was 410.71ms, as opposed to 409.59ms of the standard implementation. The performance increase of the asynchronous approach declines significantly to 0.27%. This is due to the GPU running out of resources to support the simultaneous execution of the compute and render pipeline. There are likely too many thread blocks assigned to the GPU for the streaming multiprocessors to execute simultaneously, decreasing performance despite high occupancy.

This is demonstrated by the significant increase in render time, highlighted in Figure 4.13. At 10000 boids, the render time measured for the asynchronous compute approach is 0.29ms. However, at 100000 boids, it increased drastically to 84.96ms. Despite this, the average performance increase remains low, indicating that the two pipelines are not running synchronously, as otherwise the performance increase would likely be higher due to the significant size of the render pipeline.

5.4 - Discussion Summary

The results gathered indicate that offloading the boids functionality to the GPU significantly improves performance over a single threaded CPU approach across all sample sizes. This is due to effectively leveraging the parallel capabilities of the graphics processing unit to optimize and

multithread the boids algorithm. These results align with the aim of the project and effectively answers the first research question:

- What performance improvements can be achieved by optimizing a single-threaded CPU based boids algorithm using GPGPU.

The results also highlight the performance improvements achieved by asynchronously executing the compute and render pipelines of the GPGPU implementation. They demonstrate that at lower boid counts, the highest improvements in performance are gained, by achieving a maximum performance increase of 9.85% with a 500-sample size. This is highlighted in Figure 4.3. However, at higher boid counts, this diminishes substantially, reaching a minimum performance increase of 0.27% with a 100000-sample size. These results also align with the aim of the project and effectively address the second research question:

- How can a GPGPU implementation of the boids algorithm be further optimized using an asynchronous compute approach in DirectX12.

Chapter 6 - Conclusion

6.1 - Project Overview

The project initially aimed to explore the real-time performance increase associated with implementing a boids model using GPGPU techniques. The concept of asynchronous compute was considered due to the capabilities of the DirectX 12 graphics API becoming more apparent during development. This concept was eventually integrated, providing an interesting optimisation to the classic GPGPU implementation of the boids algorithm.

The results of the project demonstrate the decline in efficiency of the single threaded CPU model as the sample size increases. However, it also highlighted the significant performance improvements achieved by utilizing the parallel capabilities of GPU for the boids calculations. These results imply that the GPU is an ideal candidate for optimising a boids model, especially at large entity counts, pushing the boundaries of how many entities can run simultaneously.

The project also discovered that the asynchronous approach does result in higher performance across virtually all sample sizes. This demonstrates the efficiency of simultaneously executing the compute and render pipelines. Naturally, this leads to a higher occupancy of the streaming multiprocessors by providing more tasks for it to execute, leading to better utilization of the available GPU resources.

However, as the entity counts increased, the performance improvement of the asynchronous compute method sharply declined. This is due it taking significantly longer to update the boids data using the compute pipeline than to render the boids themselves. At the largest sample sizes, the GPU runs out of available resources to support the simultaneous execution of both pipelines, resulting in only minimal performance gains.

After analysis of the performance impact of different thread group sizes, the most efficient one was determined to be 128 threads per group across

most sample sizes. This is due to the increased volume of thread groups, providing better utilization of the streaming multiprocessors within the GPU, increasing occupancy. This allows for more opportunities to hide latency (NVidia, 2023). However, at a very specific sample size of 50000, 512 threads per group outperformed previous expectations. This is due to the underlying GPU hardware used to gather test results, the NVidia 3060ti. This number of threads per group highlights the ideal quantity of blocks per SM for this GPU, resulting in a performance increase.

6.2 - Critical Analysis

The optimizations present within the application focus on using the naive implementation of the boids algorithm. This version is relatively inefficient, due to having an O notation of O(n^2) where n represents the number of boids present within the model. This limits the maximum performance of the overall application, especially at high entity counts. Test results were also limited to a single machine. Further analysis with results gathered from other variants of GPU hardware could improve the quality of this study, leading to more wide-ranging and impactful conclusions being drawn.

In real-world settings, a GPGPU approach to improving performance for an AI model within a video game would only be successful if the underlying GPU hardware had enough resources to sustain both the compute and render pipelines simultaneously. If the functionality of the AI is not suitable for multithreading, any performance improvements would be diminished. This could even result in a performance loss, if introducing the AI logic to the GPU results in resource contention, due to the high graphical demands of modern video games.

6.3 - Future Work

The project may be expanded in the future to accommodate spatial partitioning. Utilizing a spatial partitioning data structure, such as an octree, would allow for a ranged-based search to limit the number of entities considered for the rules calculations, significantly lowering the O notation.

This would directly improve the overall performance of the model. (Kratz, Luthman, 2021).

Another direction to expand the project in is to implement asynchronous compute with multiple pipelines. Each compute pipeline could perform one of the boids rule functions, spreading the workload between more thread blocks. This could increase the occupancy of the GPU, due to the streaming multiprocessors being heavily underutilized at low sample sizes, resulting in higher performance of the overall model.

List of References

Takahashi, D. (2019) *World War Z hands-on — More to do than just shooting zombie swarms*, *VentureBeat*. [Online] Available at: https://venturebeat.com/games/world-war-z-hands-on-more-to-do-than-just-shooting-zombie-swarms/ (Accessed: 10 October 2024).

Gach, E. (2025) *Monster Hunter Wilds' First Major Update Promises Much-Needed Performance Improvements On PC*, *Kotaku*. [Online] Available at :https://kotaku.com/monster-hunter-wilds-pc-stability-performance-mizutsune-1851773610 (Accessed: 25 April 2025).

Wang, Y. (2023) *GPU programming concepts — Introduction to GPU Programming documentation*, *Github.io*. [Online] Available at: https://enccs.github.io/intro-gpu-programming/4-gpu-concepts/ (Accessed: 13 March 2025).

Reynolds, C.W. (1987) *Flocks, herds and schools: A distributed behavioral model.* [Online] Available at: https://doi.org/10.1145/37401.37406 (Accessed: 15 March 2025)

Kratz, J. and Luthman, V. (2021) *Comparison of spatial partitioning data structures in crowd simulations*, *DIVA*. [Online] Available at: https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1595833&dswid=-1625 (Accessed: 14 March 2025).

Myszkier, G. (2023) *Comparing bins and quadtree optimizations of the boids simulation to the naïve approach.* [Online] Available at: https://doi.org/10.13140/RG.2.2.32538.21444. (Accessed: 14 March 2025)

Husselmann, A.V. and Hawick, K.A. (2011) *Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUS*. [Online] Available at: https://doi.org/10.2316/P.2011.757-012. (Accessed: 15 March 2025)

Sörman, T. (2016) *Comparison of Technologies for General-Purpose Computing on Graphics Processing Units*, *DIVA*. [Online] Available at: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A909410&dswid=-5273 (Accessed: 15March 2025).

Silva, A.R.D., Lages, W.S. and Chaimowicz, L. (2009) 'Boids that see', *Computers in Entertainment*, 7(4), pp. 1–20. [Online] Available at: https://doi.org/10.1145/1658866.1658870. (Accessed: 16 March 2025)

Carrillo, S., Siegel, J. and Li, X. (2010) *Impact analysis of conditional and loop statements for the NVIDIA G80 architecture* [Online] Available at: https://www.researchgate.net/publication/252069571_Impact_analysis_of_conditional_and_loop_statements_for_the_NVIDIA_G80_architecture. (Accessed: 18 March 2025)

Mills, C. (2024) *GPU MODE Lecture 8: CUDA Performance Checklist – Christian Mills*, *Christian Mills*. [Online] Available at: https://christianjmills.com/posts/cuda-mode-notes/lecture-008/ (Accessed: 18 March 2025).

Ryoo, S. *et al.* (2008) 'Optimization principles and application performance evaluation of a multithreaded GPU using CUDA', *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP '08* [Preprint]. Available at: https://doi.org/10.1145/1345206.1345220. (Accessed: 18 March 2025)

Falconer, Ruth, and Alasdair Houston. “Visual Simulation of Soil-Microbial System Using GPGPU Technology.” *Computation*, vol. 3, no. 1, 27 Feb. 2015, pp. 58–71, Available at: www.mdpi.com/2079-3197/3/1/58, https://doi.org/10.3390/computation3010058. (Accessed 16 Apr. 2025).

Lindqvist, S. (2018) *Performance Evaluation of Boids on the GPU and CPU, DIVA.* [Online] Available at:https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1191916&dswid=6159 (Accessed: 15 March 2025)

Omar, O. (2025) *Dear ImGui*. GitHub. [Online] Available at: https://github.com/ocornut/imgui (Accessed: 11 April 2025).

Riccio, C. (2005) glm. [Online] Available at: https://glm.g-truc.net/ (Accessed: 13 April 2025).

Microsoft (2015) *Direct3D 12 graphics - Win32 apps*. [Online] Available at: https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics (Accessed: 13 April 2025).

Jeremiah (2017) *Learning DirectX 12*, *3D Game Engine Programming*. [Online] Available at: https://www.3dgep.com/learning-directx-12-1/ (Accessed: 15 April 2025).

NVidia (2015) *Achieved Occupancy*, *docs.nvidia.com*. Available at: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm (Accessed: 24 April 2025).

Moass, D. (2020) *Nvidia RTX 3060 Ti Founders Edition Review*, *KitGuru*. Available at: https://www.kitguru.net/components/graphic-cards/dominic-moass/nvidia-rtx-3060-ti-founders-edition-review/ (Accessed: 25 April 2025).

NVidia (2023) *CUDA C++ Best Practices Guide Release 12.2 NVIDIA*. Available at: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (Accessed: 28 April 2025).

Appendix

Appendix 1

Abertay University

GDPR Research Data Management
Data Sign Off Form

For undergraduate or postgraduate student projects supervised by an Abertay staff member.

This form MUST be included in the student's thesis/dissertation. Note that failure to do this will mean that the student's project cannot be assessed/examined.

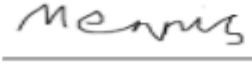
Part 1: Supervisors to Complete

By signing this form, you are confirming that you have checked and verified your student's data according to the criteria stated below (e.g., raw data, completed questionnaires, superlab/Eprime output, transcriptions etc.)

Student Name:	Michael Ennis		
Student Number:	1902141		
Lead Supervisor Name:	Ruth Falconer		
Lead Supervisor Signature			
Project title:	Boids that sprint: Optimising an AI Algorithm using Asynchronous Compute and GPGPU in DirectX12		
Study route:	PhD [Control]	MbR [Control]	MPhil [Control]
	Undergraduate [Control]	PhD by Publication [Control]	

Part 2: Student to Complete

	Initial here to confirm 'Yes'
I confirm that I have handed over all manual records from my research project (e.g., consent forms, transcripts) to my supervisor for archiving/storage	Yes
I confirm that I have handed over all digital records from my research project (e.g., recordings, data files) to my supervisor for archiving/storage	Yes
I confirm that I no longer hold any digital records from my research project on any device other than the university network and the only data that I may retain is a copy of an anonymised data file(s) from my research	Yes
I understand that, for undergraduate projects, my supervisor may delete manual/digital records of data if there is no foreseeable use for that data (with the exception of consent forms, which should be retained for 10 years)	Yes

Student signature: 

Date: 29/04/2025