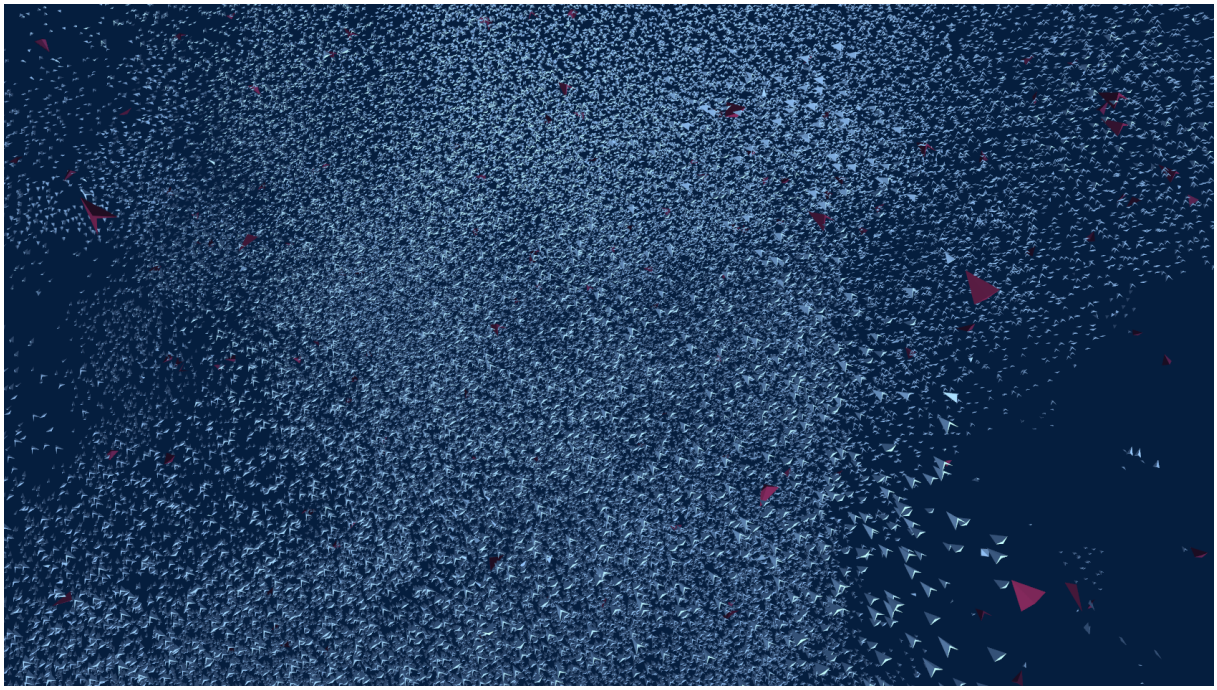
A COMPARATIVE STUDY OF MASSIVE FLOCK SIMULATIONS ON THE CPU AND THE GPU



Runvik, Arvid - IT
rarvid@student.chalmers.se

Karlsson, Leo - IT
leoo@student.chalmers.se

Hällqvist, Elias - D
haelias@student.chalmers.se

Lyrstrand, Oskar - CS
oskar@lyrstrand.se

Kraft, Jonathan - IT
kraftj@student.chalmers.se

June 9, 2019

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Abstract

This paper describes how large scale flocks can be simulated efficiently with modern computer graphics using parallelization techniques and spatial data structures. The results show that parallelism is essential in improving performance, both in CPU and GPU implementations. Best performance was achieved using CUDA, uniform grid, and batching which resulted in a simulation of two million agents at a frame rate of 30 frames per second. The simulation was extended with a graphical user interface, predators, colours and user interaction in order to create a simple game.

***Keywords*—** Craig Reynolds, Simulation, Flock, Herd, School, Multithreading, Parallelism, GPGPU, CUDA, TBB, Agent, Spatial Hashing

Sammandrag

Denna rapport beskriver hur storskaliga flockar kan simuleras effektivt med modern datorgrafik med hjälp av tekniker så som parallelisering och avancerade datastrukturer. Resultaten visar att parallelism är avgörande för att förbättra prestanda, både i CPU och GPU implementeringar. Den bästa prestandan uppnåddes med användning av CUDA, datastrukturen Uniform Grid samt batching vilket resulterade i en simulering av två miljoner agenter med en bildhastighet på 30 bilder per sekund. Simuleringen utökades med ett grafiskt användargränssnitt, rovdjur, färger och interaktivitet för att skapa ett enkelt spel.

Acknowledgements

We would like to thank Marco Fratarcangeli for supervising our project and giving us invaluable advice about the techniques used in this paper. Another person deserving of our gratitude is Joey De Vries, author of learnopengl.com, whose tutorials has guided us through the maze of learning OpenGL. Our thanks also extends to Microsoft for providing the library manager vcpkg which has saved us many hours of labor. The help we have received from other students, teachers and friends is highly appreciated and has elevated our work beyond our original capabilities. And of course, we would also like to thank Craig Reynolds for developing the original flocking algorithm which sparked the idea for this paper and gave us this inspirational challenge. Finally, we would like to thank flocks for existing, echoing Reynolds original flocking paper where he wrote that *"nature is the ultimate source of inspiration for computer graphics and animation."*

Contents

1 An Introduction to Huge Flocks

Flocking, schooling and herding are commonly seen throughout the animal kingdom, with congregations reaching millions of individuals. These gatherings exhibit complex behaviour, yet it seems that many of them do so without any central leader. In fact, it has been shown that similar behaviour can result from individuals following only a few simple behavioural rules [1]. It is not even necessary that individuals are aware of the group's behaviour as a whole. Instead they may only act upon local information, such as the position and velocity of their closest neighbours [2]. Flocking is therefore often described as a *emergent property* of these few rules, which can be loosely defined as *"features of a system that arise unexpectedly from interactions among the system's components."* [1]

Emergent properties are all around us, in the form of such disparate phenomena as memories stored in the brain [3] and traffic jams. In the latter example, individual car drivers can successfully be modelled as agents following a set of simple rules, analogous to flock members [4]. Traffic is just one example where agent-based modelling can help unravel a seemingly complex behaviour. Aside from understanding existing phenomena, knowledge about emergent behaviour can also help us in developing new technology, such as creating flock-like behaviour in drones to help in disaster areas.

As an emergent behaviour is the result of the interactions of many agents, it can be hard to accurately represent it with a single equation describing the system as a whole. An alternative approach is to model the individual agent, run a computer simulation of the interactions, and study the behaviour that emerges. However, large scale simulation can be computationally heavy since calculations must be made for every agent's interactions with other agents. The number of calculations needed can grow quadratically with the number of simulated agents if care is not taken when designing the program [5].

Larger flocks can be simulated by utilising the power of modern computer hardware. Graphics APIs such as OpenGL or Vulcan grants the programmer full control over the rendering process, in a way that every required step of the simulation can be optimised. And instead of calculating and updating the agents sequentially, which would be the naive approach, multi-threading can be used both on the CPU and GPU. Since today's hardware (especially graphics hardware) is built for parallelism, there are many tools and libraries making it easier to utilise this technique.

1.1 Purpose and scope

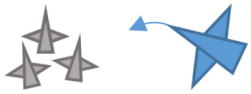
The primary purpose of this paper is to describe, and compare the performance of , different implementations of an application that simulate massive flocks of independent agents which follow simple rules. As a starting point, the three flocking rules first described by Craig Reynolds are to be used, namely: collision avoidance, velocity matching, and flock centring [6]. The goal is to achieve complex behaviour emerging from these simple rules with flocks of at least one million agents. Following this, a discussion and a comparison of which techniques grant the most agents are presented, as well as a subjective evaluation of how realistic the resulting flock behaviour is. As a way of presenting the result, the simulation is extended with more behaviours as well as some interactive features, making it into a simple game.

2 Flocking behaviour

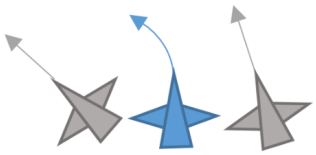
One basic algorithm that elegantly imitates the behaviour of a flock was first introduced by Craig Reynolds at the SIGGRAPH conference in 1987 [6]. Reynolds' conference paper used particle systems together with three different forces for each individual agent (or "boid"; bird-like object as he called them) to simulate flocking behaviour. Combining these forces with the agent's current velocity will result in an updated velocity. The three forces are: *flock centring*, *velocity matching* and *collision avoidance*. When these simple forces are applied to many agents, a complex behaviour will emerge and the agents will form groups of flocks.

2.1 Reynolds' original flocking algorithm

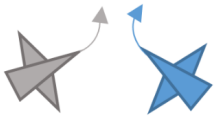
Each force of Reynolds' algorithm is calculated for, and applied to every agent. As seen in Figure 1, flock centring is the urge of an agent to steer towards the centre of the flock. Velocity matching causes each agent to align its speed and velocity with its surrounding neighbours. Collision avoidance acts as a counterbalance, repelling agents too close to each other in order to avoid collisions. Every force calculation requires information about an agents neighbours, where the neighbours are defined as the group of agents located within a certain range of the agent. Reynolds did not provide a detailed account of how the forces were calculated in his implementation. The mathematical formulas in the following sections instead reflects how the authors chose to implement the forces.



(a) Flock centring: agents adjust their heading towards the flock centre.



(b) Velocity matching: agents match their speed and heading to nearby agents.



(c) Collision avoidance: agents separate from nearby agents to avoid collisions.

Figure 1: Reynolds' three original flocking forces.

2.1.1 Flock centring

An essential part of flocking is that agents stay together instead of splitting up, which is the main responsibility of the *flock centring* force. This force creates a vector pulling the agent towards the centre of the neighbouring agents. Flock centring $\vec{v}_c$ velocity for agent i can be expressed as

$$\vec{v}_c = \frac{1}{n}\left(\sum_j^n \vec{p}_j\right) - \vec{p}_i,$$

where n is the number of neighbours and p_i is the position of agent number i.

2.1.2 Velocity matching

Another key part of flocking is the velocity matching, also called *alignment*. Velocity matching is the urge of an agent to match its velocity with the velocities of the neighbouring agents. Similarly to how

the flock centring force was calculated, velocity matching can be calculated by averaging the velocity of nearby agents. The alignment velocity v_a for agent i can be expressed as

$$\vec{v}_a = \frac{1}{n}\left(\sum_j^n \vec{v}_j\right) - \vec{v}_i$$

where n is the number of neighbours and v_i is the velocity of agent number i.

2.1.3 Collision Avoidance

In contrast to the cohesive properties of flock centring, collision avoidance is responsible for making sure that no agents collide. The natural instinct of avoiding collisions should be stronger when the agents are closer to each other and weaker the further away from each other they are. To achieve this, first decide the magnitude of this force by calculating the inverse square of the distance between the agents. Secondly, let the direction of the force point away from the agents neighbours. The separation velocity v_s for agent number i can be expressed as

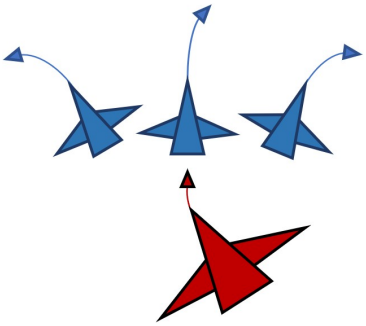
$$\vec{v}_s = \frac{1}{n}\left(\sum_j^n \frac{\vec{p}_i - \vec{p}_j}{\|\vec{p}_i - \vec{p}_j\|^2}\right),$$

where n is the number of neighbours and p_i is the position of agent number i.

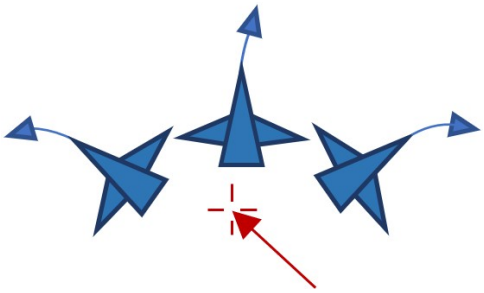
2.2 Extensions to flocking

Flocking animals typically show a range of behaviours not covered by Reynolds' model. Extensions to the model have been developed by several other authors: with predators [7], flock leaders [8], mutations, food, different species [9], to name a few.

Figure 2 shows the extended rules considered in this paper, predators and a repelling line. All agents other than predators are regarded as prey and will flee from predators. In turn, predators hunt and kill prey agents, which removes them from the simulation. And as the name suggests, the repelling line repels agents away from the line. Implementation details of these extensions are described later, in section 4.2.



(a) Agents vs. Predator: Agents spreading out to avoid an approaching predator.



(b) Repelling line: Agents repel from a point when being targeted by a line resembling a laser.

Figure 2: Extensions to Reynolds' three original flocking rules.

2.3 Resulting velocity

Each force produces a vector which affects the agents' velocity in the next timestep. One way of combining the different vectors is to simply take a weighted average of them. Each force is weighted and summed resulting in an output vector that constitutes the agent's new velocity. How the forces are weighted against each other has a large impact on the flock's emerging behaviour. The updated velocity $\vec{v}'$ for an agent becomes:

$$\vec{v}' = a \cdot \vec{v}_c + b \cdot \vec{v}_a + c \cdot \vec{v}_s,$$

where a, b, c are constants.

The drawback with this approach is that conflicting "urges" can make the agents indecisive in certain situations. For example, if an agent at the edge of a flock is heading towards an obstacle, a collision avoidance rule may produce a vector aimed to the left, while the flock centring rule produces a vector in the opposite direction. The two vectors cancel each other, and the agent crashes into the obstacle.

An alternative to the weighted average approach is to allocate acceleration to the different rules according to a priority order. In each time step, each agent starts off with some predefined maximum amount of acceleration left. In order of priority, the different rules are then allocated their requested amount of acceleration, as long as there is acceleration available. The scaling of each vector produced by the rules is described by the following formula, where the variable m_a has been initialised to the predefined maximum amount of acceleration:

$$\vec{v}_i \leftarrow \min\left(\|\vec{v}_i\|, m_a\right) \cdot \hat{\vec{v}}_i$$

$$m_a \leftarrow \max\left(0, m_a - \|\vec{v}_i\|\right).$$

Here, $\vec{v}_i$ is the vector produced by rule i. After scaling each vector, the new velocity of the agent is simply

$$\vec{v}' = \vec{v}_c + \vec{v}_a + \vec{v}_s.$$

This approach ensures that the most pressing "urges" are fulfilled first, possibly leaving some rules completely ignored. For example, if obstacle avoidance is the highest prioritised rule, an agent that is very close to an obstacle may "use up" all available acceleration in order to dodge the obstacle.

It goes without saying that in the physical world there are limits to how high acceleration an agent can produce, and how high velocity it can reach. Therefore, for a more realistic simulation one can choose to limit these values to predefined constants in a simulation. Mathematically it can be expressed as

$$\vec{a} \leftarrow \vec{v}' - \vec{v}$$

$$\vec{v}' \leftarrow \vec{v} + \min\left(\|\vec{a}\|, a_{max}\right) \cdot \hat{\vec{a}}$$

$$\vec{v}' \leftarrow \min\left(\|\vec{v}'\|, v_{max}\right) \cdot \hat{\vec{v}}'$$

where $\vec{v}$ is the old velocity, $\vec{v}'$ is the updated velocity, a_{max} is the maximum magnitude of acceleration, v_{max} is the maximum magnitude of velocity.

3 Technical Description and Utilising the Hardware

Simulating a large number of individual agents in real time requires a powerful programming language, such as C++. Doing this also requires utilising the language in a powerful way, requiring an investigation of hardware, libraries, algorithms and data structures. As such, the focus in this chapter is twofold: the first sections centre around utilising the hardware as efficiently as possible. Towards the end of the chapter, the focus changes towards more pure software approaches, such as data structures and algorithms.

3.1 CPU parallelism with TBB

Even though *Central Processing Units* (CPUs) are mainly optimized for sequential code [10], they often have multiple cores - meaning that they are capable of running multiple processes in parallel. Taking advantage of this resource can be done by using Thread Building Blocks (TBB), a library for C++ developed by Intel Corporation [11], which simplifies the process of creating parallel tasks on the CPU. The library does so by dividing the workload onto multiple CPU cores, mainly by using algorithms and data structures adapted for parallel executions. If a system has a CPU with four cores a program could theoretically be up to four times faster with TBB by utilizing all of the cores in parallel.

One drawback with trying to maximise performance by parallelisation on the CPU alone is the fact that CPUs often only have a few cores. Performance can be improved more on a modern *Graphics Processing Unit* (GPU), which often have thousands of cores. Writing GPU code has, and can still be, quite cumbersome, but with tools such as OpenGL, it has been made a lot easier.

3.2 Accessing the GPU with OpenGL

OpenGL is an Application Programming Interface (API) used for writing computer graphics applications that access and control the GPU on the device it runs on. It provides an abstraction layer between the graphics hardware and software, essentially making it easier for developers to focus on quality and performance instead of having to worry about different implementations for different platforms. To achieve great performance on modern graphics processors, OpenGL uses a combination of *pipelining* and parallelism [12].

Pipelining, or more specifically the *graphics pipeline*, consists of several stages that need to be performed whenever something is being rendered to the screen. There are multiple stages in the graphics pipeline, an example is depicted in Figure 3. In this example, three vertices represent a triangle that will be rendered on screen. First, these vertices are sent to the *vertex shader*. This first stage runs parallel programs for every vertex and can be used to transform or colour every vertex of the triangle. Each vertex is then sent to the next stage where the vertices are processed together and merged into shapes. The shapes are then sent to the *rasterizer* which basically makes a pixelated representation of each shape. The *fragment shader* then receives these rasterized shapes as input and runs a program for each fragment of the total picture in parallel. The fragment shader is often responsible for calculating the final colour of each pixel. Before the image is rendered on the screen, all parts of the picture that will not be visible in the final result are removed and blending is applied. The final image is rendered to something called the *framebuffer* and the image is shown on the display.

The small programs that run on each stage of the graphics pipeline are called *shaders*, and in contrast

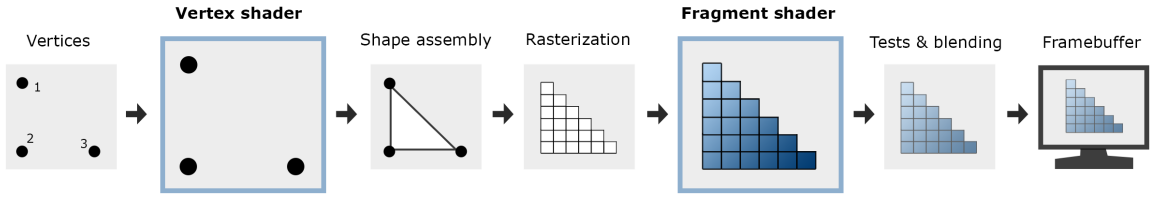


Figure 3: The graphics pipeline: Each stage has different responsibilities and each stage takes an input and output from adjacent stages

to coding as usual or with TBB, the shader code only resides on the GPU. The shader for each stage runs its code in parallel for every input. This parallelism is what grants the shaders its power. For example, if you have a screen resolution of 1980x1080 pixels and you draw two triangles filling the whole screen, the fragment shader will run the same code for all two million pixels in parallel. This efficiency is so powerful that it becomes interesting to use the GPU and the shaders for other purposes than rendering, something that is made possible by the OpenGl Shading Language (GLSL).

3.2.1 Coding shaders with GLSL

The OpenGL Shading Language (GLSL) was a response to a growing trend of replacing fixed functionality with programmable functionality in areas of the graphics hardware that had grown exceedingly complex [13]. The new language granted programmers with the ability to write their own shaders. Being able to customize shaders allows for more control and also the potential of improving performance in general. For example, calculations such as transformations could easily be moved to the *vertex shader*, where all vertices can be transformed in parallel. The main advantage of calculating the transformations in parallel on the GPU instead of sequentially on the CPU is that the parallel implementation is more efficient. And as the graphics hardware grew more powerful, so did the interest of utilizing its performance for other things than graphics.

3.3 Utilizing the GPU with GPGPU

The performance boost in graphics hardware over the last few decades has been dramatic, with it probably being the most cost-efficient computational hardware today [10]. General purpose computing on graphics processing unit (GPGPU) takes advantage of this efficiency, by utilizing the graphics card for computations that are normally done by the CPU. The cores in GPUs are generally simpler than those in CPUs in that they have a smaller instruction set and a smaller cache, but in turn there are many more of them. While CPUs may have a few cores, a GPU can have thousands of them. This makes the GPU very powerful for processing many smaller computations in parallel. With flocking as an example, updating every agent's position according to a few forces is a task that a GPU might complete faster than a CPU since there are many simple computations. However, sending data between the CPU and the GPU can however introduce significant latency in an application[14], so where the data (that is to be processed) resides must be taken into consideration. Ideally, all data that is to be processed by the GPU should be transferred there once, and then stay there for the full duration of the program.

3.3.1 Simplifying GPU programming with CUDA

CUDA is a parallel computing platform that simplifies GPGPU programming. It features language extensions that enable programmers to write functions that are executed on the GPU directly in C or C++ code. These GPU accelerated functions are often referred to as *kernels*. A drawback with CUDA is that it can only be used with CUDA-enabled GPUs manufactured by Nvidia[15].

One very useful CUDA feature is *unified memory*. It is a way of allocating data in a single memory address space that can be accessed by both the CPU and the GPU. For example, it is possible to initialize an array in unified memory, let the CPU fill it with data, and then launch a CUDA kernel that manipulates the data in the array. Of course, the data still needs to be transferred on the bus to the GPU, but there is no need for the programmer to explicitly instruct the CPU to do so.

CUDA also features OpenGL interoperability, which means it can share data on the GPU with OpenGL [16]. Used right, this feature makes it possible to greatly reduce the amount of data that is being transferred on the bus between the GPU and CPU, by keeping and updating agent data in GPU memory and then instructing OpenGL to render it directly.

3.3.2 Compute shaders

Libraries such as CUDA are very popular when it comes to general purpose programming because they are great for data-parallel computing. One drawback is that they can be quite cumbersome to use and initialise since they require dedicated drivers and installation of additional toolkits. Compute shaders, on the other hand, is part of the OpenGL core. They use the GLSL language so OpenGL programmers will be familiar with the code. Compute shaders can also share data directly with other shaders in the graphics pipeline but most importantly they are not part of the pipeline. This means that they can be directly invoked whenever it suits the application.

3.4 Avoiding congestion with batching

As has been mentioned earlier, sending data between the CPU and the GPU can introduce significant latency in an application [14]. This is because the bus between the CPU and the GPU can become congested, which will lower the efficiency of any program. *Batching* is a way of decreasing this communication between the CPU and GPU by grouping the data and sending it in larger chunks. The time it takes to send a small amount of data is similar to the time it takes to send a large chunk of data in one go, which is why it is more efficient to send few large packets of data instead of many small.

For example, position data for agents in a game world are typically initiated on the CPU and then sent to the GPU for rendering. A naive way of rendering these agents would be to loop through each of them and sending them one at a time together with a transformation matrix to the GPU. But because the CPU is responsible for sending this data the performance of the application would depend on how fast the data can be sent, rather than by depending on the GPU [17]. With *batching*, a chunk of data is instead sent once every frame and the pressure on the CPU is decreased significantly for larger amounts of agents. This concept is shown here in pseudocode:

```
# rendering without batching
for each agent a in agents
{
    // send transformation matrices to gpu
    render(a);
}

# rendering with batching
for each agent a in agents
{
    // transform agent a on the cpu
}
render(agents);
```

Notice how the render function is only called once in the batching example, while in the first example is called for every agent in the simulation. This is an example of how the software has to be optimised in order to be efficient in cooperation with the hardware, which naturally leads us to the part of this chapter that focuses on software optimisation.

3.5 Optimising using data structures and algorithms

When simulating large scale flocking behaviour in real time, one of the bottlenecks that arise is the process of comparing agents with one another. Comparing agents in flocking is needed in order to decide whether they are close enough to affect one another. And as the number of agents grows, the number of comparisons needed for flocking grow quadratically - which quickly turns ineffective. This problem is shared with many different simulations and programs, and this chapter aims to describe data structures and algorithms that mitigate and prevent this bottleneck.

3.5.1 Efficient neighbour search using a uniform grid

As mentioned before, agents in Reynolds' flocking model only reacts to neighbouring agents that are within a certain distance. The naive method of determining which agents meet these criteria is to simply compute the distance between each and every agent. The time complexity for this is $\mathcal{O}(n^2)$, which means it is not a suitable approach for flocks with more than a few agents. A more efficient approach is to use spatial data structures, which is an umbrella term for techniques that organize spatial data, such as 3D coordinates.

A *uniform grid* is an example of a simple spatial data structure. It works by dividing space into equally sized *cells* of some size. Each cell has discrete coordinates in the grid. For example, assume a 2D space that is divided into cells with the width a and height b. Then, a point with coordinates (x, y) is in a cell that has the coordinates $(\lfloor x/a \rfloor, \lfloor y/b \rfloor)$. Agents can then be grouped together based on which cell it is positioned in.

3.5.2 Efficient look-up using Spatial Hashing

Grouping agents together can be done by storing them in a hash table, using cell coordinates as keys. This means that agents that reside in the same cell will end up in the same hash table cell, or "bucket".

This technique is called *spatial hashing*. For an agent to find the agents that are in the same cell, it is now a matter of calculating its cell's coordinates, do a hash table lookup with the coordinates as a key, and iterate over the agents contained in the bucket. The same goes for finding neighbours in adjacent cells, but using different cell coordinates. Returning to the 2D example: to find neighbours in the cell just to the right of the agent's own cell, you just use the cell coordinates $(\lfloor x/a \rfloor + 1, \lfloor y/b \rfloor)$ as the key when doing the hash table lookup.

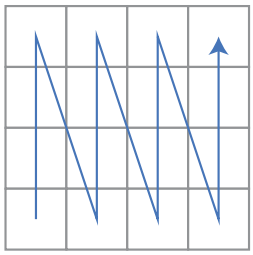
Another way of grouping agents is to simply sort them according to their cells' coordinates. This approach has been suggested by Simon Green [18], but for particle collision instead of agents. With this method, each cell is identified by a unique integer. After sorting the array of agents by their cell ID, two additional arrays are used to store the beginning and end indices of each cell. The method is perhaps best understood through an example. Consider an instance where agents have been sorted by their cell ID. Agents a, b both resides in cell number 4, and agents c and d resides in cell number 2 and 3 respectively. Cell number 0 and 1 is empty, so cell start/end indices is `null` for these cells. Cell number 2 contains only one agent (c), and that agent is found at index 0 in the agent's array. So both the cell start and end indices for this cell is 0. The same goes for cell number 3, it contains one agent that is found at index 1 so start/end indices are 1. Finally, cell number 4 contains two agents, the first one is at index 2 and the last one at index 3. Thus the start index for this cell is 2 and the end index 3. For a more detailed example of actual implementation, see section 4.1.3.

Note that each cell in the grid requires one entry in cell start/end arrays. Therefore the space in which the agents are simulated must be limited. By contrast, the hash table in a spatial hashing implementation only requires as many entries as there are occupied cells. In other words, it has at most n entries, where n is the number of agents.

3.5.3 Generating grid cell identifiers

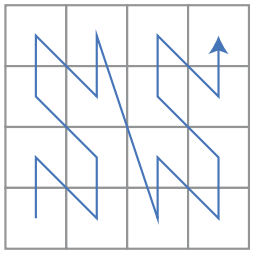
When using a uniform grid with sorting, the cell identifiers can be constructed in a number of ways. Perhaps the most straightforward way is to simply concatenate the bits of the cell coordinates into a single integer. For example, a cell in a 3D grid with binary coordinates $(x_3x_2x_1x_0, y_3y_2y_1y_0, z_3z_2z_1z_0)$ would be identified by the integer $x_3x_2x_1x_0y_3y_2y_1y_0z_3z_2z_1z_0$, where each variable represents a single bit. An alternative way is to interleave the bits: $x_3y_3z_3x_2y_2z_2x_1y_1z_1x_0y_0z_0$. The latter method is called *Z-order* or *Morton order* [19]. See Figure 4 for a graphical example.

The main advantage of Z-order is that cells that are close to each other in a 2D or 3D grid, on average, get ID integers that are closer to each other than what you would get with simple concatenation. This, in turn, means that when the agents' data has been sorted by their cell IDs, agents that are close to each other in 3D space will be close to each other in memory. The more formal way of expressing this is that locality is preserved when multidimensional data is mapped to one dimension. Consider for example two adjacent cells in a 3D grid with coordinates $(0, 0, 0)$ and $(1, 0, 0)$. Suppose the coordinates are represented by 4 bits each. With bit concatenation their binary identifiers are 0000 0000 0000 (0 in decimal) and 0001 0000 0000 (256 in decimal.) The Z-order identifiers are, by contrast, 0000 0000 0000 and 0000 0000 1000 (8 in decimal.)



3	7	11	15
2	6	10	14
1	5	9	13
0	4	8	12

(a) Column major order as a result of bit concatenation.



5	7	13	15
4	6	12	14
1	3	9	11
0	2	8	10

(b) Z-order as a result of bit interleaving.

Figure 4: Two different ways of encoding cell IDs in a 2-dimensional grid

4 Creating the Application and Gathering Data

The two previous chapters introduced the mechanics of flocking, extensions of flocking, graphics hardware and some software approaches of optimising flocks. Drawing from these lessons, this chapter focuses on how the simulation was implemented using these techniques. This includes implementations with and without using GPGPU and efficient data structures. It also includes the extensions of flocking mentioned in section 2, which has been developed into a game with some visual effects. Finally, the chapter concludes with a small analysis of what ethical aspects have been taken into account.

4.1 Different implementations of neighbour searching

In order to maximise the performance of the flocking simulation, multiple combinations of data structures and techniques were implemented, as seen in table 1. The label "naive" will hereafter refer to the versions that use a pair-wise comparison of all agents for the neighbour search. Note that the flocking simulation (and the game as a whole) behaves exactly the same across all versions. Only performance differences would be noticeable to an end user.

Table 1: Different implementations of the flocking simulation. "Algorithm" here refers to how agents find their neighbours

Technique \ Algorithm	Naive	Spatial hashing	Uniform grid with sorting
CPU, single-thread	✓	✓	
CPU, multi-thread	✓	✓	
GPU, using CUDA	✓	✓	✓
GPU, using compute shaders	✓		

In the following subsections, implementation details for the different versions are described. All versions use GLAD libraries, GLFW (for window creation and user input), GLM (for math operations on matrices and vectors). Although vcpkg is not needed for creating this project, it was used to handle the installation of libraries in order to save time. Batching was used for all implementations since it would be essential for performance improvements in general.

4.1.1 Naive versions

The naive approach does not present any particular challenges in terms of data structure choices. All agents can be stored in a single array, and each agent loop through the whole array when searching for its neighbours. In the multi-threaded CPU version, parallelism is achieved by iterating over the array using the TBB library's `parallel_for`. In the CUDA version, a kernel is launched with one thread for each agent. Each thread iterates over the whole array.

4.1.2 Spatial hashing versions

In the spatial hashing versions, a uniform grid is used, as described in section 3.5.1. The grid cells have sides that are equal to the maximum distance at which agents reacts to each other. By choosing this size, an agent's neighbours are guaranteed to be found in the cells adjacent to the agent's own cell. Thus, for an agent to find it's neighbours, a total of 27 (3^3) cells needs to be checked.

Each agent is put in a "bucket" based on it's hashed cell coordinates. What is put in the hash table is actually a `struct` containing pointers to the first (head) and the last (tail) agent in the bucket. These pointers are used together with an array `nextAgent` that contains pointers pointing to the next agent in the bucket. If a bucket contains at least two agents, both the "head" and "tail" points to some agent. When a new agent is added to the bucket, it replaces the tail agent, and a pointer to this new agent is added to the old tail's entry in `nextAgent`. This creates something akin to a singly linked list. The reason an actual singly linked list (like `std::forward_list`) is not used is that we know beforehand that agents belong to exactly one bucket, so it is sufficient for each agent to point to only one other agent. This way, the extra performance overhead of dynamic memory allocation when creating the lists can be avoided.

All versions follow the same basic steps:

1. Calculate each agents' cell ID and put them in the hash table bucket accordingly
2. Fetch each agent's neighbours from the hash table by calculating the IDs of its own cell and adjacent cells.
3. Apply the flocking rules and update the agents' position and velocity.
4. Reset the hash table.

The multithreaded CPU version performs step 2 in parallel using, again, TBB's `parallel_for`. In contrast, all steps in the CUDA version is done in parallel. It performs step 1 in parallel by launching a kernel with one thread per agent. To avoid race conditions (where several threads tries to update the same variable at the same time), the tail and head pointers of the hash table buckets are atomically updated using CUDA's `atomicExch`. There is no need to make `nextAgent` thread-safe, since each entry is updated at most once. Step 2 and 3 are parallel, again by launching a kernel with one thread

per agent. Step 4 is carried out by launching a kernel with one thread per hash table entry.

For the CPU single thread version, the `std::unordered_map` class is used as a hash table. Unfortunately, there is no GPU implementation of this class available in CUDA, so for the CUDA implementation a custom hash function is used together with an array. The cell hash h is computed as

$$h = [(x \cdot u) \oplus (y \cdot v) \oplus (y \cdot w)] \bmod n$$

where x, y, z are the cell coordinates, u, v, w are large prime numbers, and n is the size of the array.

4.1.3 Uniform grid with sorting

As mentioned in section 3.5.1, a uniform grid can be constructed using sorting. This was implemented with CUDA using a parallel radix sort from the CUB library (`cub::device_radix_sort`) which has a time complexity of $\mathcal{O}(n)$. This version works by following these steps, as visualised in Figure 5.

1. Calculate each agent's cell ID using Z-order and store it in an array, `cellIDs`.
2. Initialise an array, `agentIDs`, that hold the agents ID's (here, the agent ID is simply the agent's index in the `agents` array.)
3. Sort the agent IDs by their corresponding cell IDs.
4. Copy each agent to a an alternate agent array, `agentsAlt` according to the now sorted `agentID` array. `agentsAlt` now contains all agents sorted by their cell ID.
5. Check where cells start and end in the `cellIDs` array and store the start and end indices in two arrays, `cellStartIndex` and `cellEndIndex`, respectively.
6. Calculate the new position and velocity of the agents in `agentsAlt` according to the flocking rules. The result is stored by simply overwriting the data in the original `agents` array. In this step, each agent finds it's neighbours by looking up the start/end indices of the adjacent cells.
7. Reset the `cellStartIndex` and `cellEndIndex` with null values.

Each step here corresponds to a kernel that is executed in parallel. Thanks to the use of two arrays holding the agents, one can be used for reading and one for writing (as is done in step 4 and 6.) This eliminates the need for atomic operations.

The reason for sorting agent IDs (which are unsigned integers) instead of sorting the agent data directly is that `cub::device_radix_sort` doesn't support sorting of custom types.

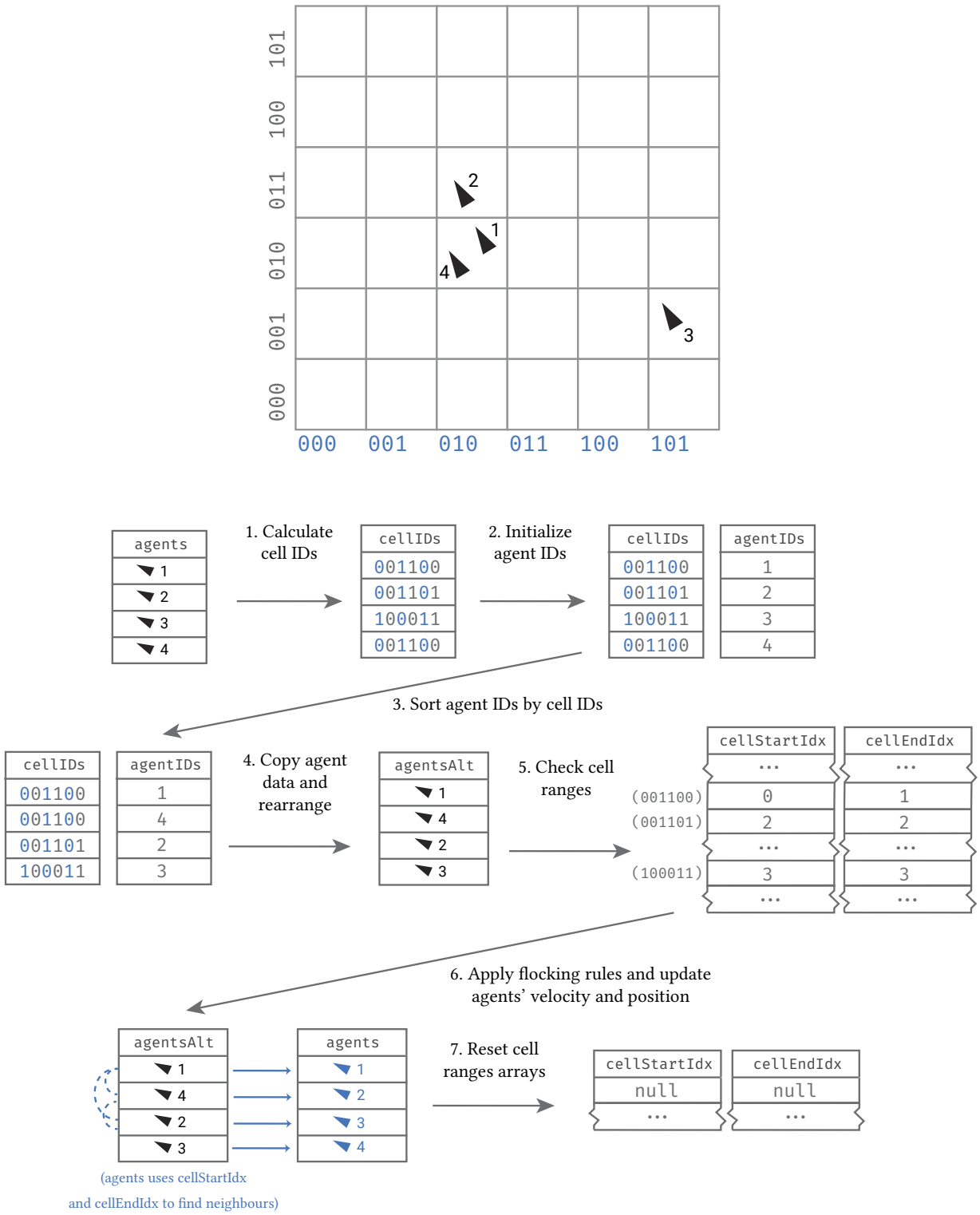


Figure 5: The different steps of the "uniform grid with sorting" method.

4.2 Game implementation

In order to present the application in an appealing and more immersive way, a game was implemented. Extensions to flocking as described in section 2.2 were added. The added behaviour include predators, predator avoidance, walls and a repulsive line. The goal of the game is to prevent the predators from killing the preys by shooting the predators with a laser. In addition to choosing when to use the laser, the user may also choose to navigate through the three-dimensional space by looking around using a mouse and traversing by pressing the WASD keys.

4.2.1 Repulsive force

When the laser is fired, agents will repel from the laser ray's path. In order to find the direction in which an individual agent is repelled, the closest point $\vec{p}_c$ on the ray's path is calculated as follows

$$\vec{p}_c = \left(\left(\vec{p}_a - \vec{p}_{laser}\right) \bullet \vec{v}_{dir}\right) \cdot \hat{\vec{v}}_{dir},$$

where p_a is the position of the agent, p_{laser} is the position of the laser, and $\vec{v}_{dir}$ is the direction in which the laser is pointing. The force on the agent is then calculated as

$$\vec{v} = \frac{\vec{p}_a - \vec{p}_c}{\|\vec{p}_a - \vec{p}_c\|^2}.$$

As seen, the magnitude is inversely proportional to the distance from the point, i.e the further away from the ray's path, the less an agent is affected by it.

4.2.2 Predators

It has been suggested that one of the reasons animals swarm is that it confuses predators. When prey animals swarm in large numbers, it gets harder for predators to single out individuals to attack. This is called the *confusion effect*[20]. In order to model this indecisiveness, predator agents are attracted to the perceived center of neighbouring agents', rather than to an individual agent. This is similar to the flock centering rule for prey agents described in section 2.1.1, and can be mathematically expressed as

$$\vec{v} = \frac{1}{n}\left(\sum_j^n \vec{p}_j\right) - \vec{p}_i,$$

where n is the number of neighbouring prey agents and p_i the position of the predator agent i.

Prey agents are repelled by predators. The vector produced by this rule is an average of normalised vectors pointing in the opposite direction relative to neighbouring predators, expressed as

$$\vec{v}_s = \frac{1}{n}\left(\sum_j^n \frac{\vec{p}_i - \vec{p}_j}{\|\vec{p}_i - \vec{p}_j\|}\right),$$

where n is the number of neighbouring predator agents and p_i is the position of the prey agent i. This is similar to the collision avoidance rule described in section 2.1.3. However, for that rule, each normalised vector was divided by the distance, as to weaken the urge to avoid collisions when the distance is large. The division by distance is omitted in the case of predator avoidance since a prey agent is acutely concerned with avoiding predators as soon as they are within the agent's scope.

4.2.3 Light, colours and flight animation

In order to improve the visual appeal and sense of depth, light was added to the simulation. This was done by adding a light source position and a light colour. When the model is rendered for each agent, light is calculated for each side of the model by calculating the dot product between the ray vector and the surface normal, see Figure 6. The dot product is then multiplied with the light colour which in turn is multiplied with the colour of the surface. This method of calculating the light of each side of a model is called *diffuse lightning* [21].

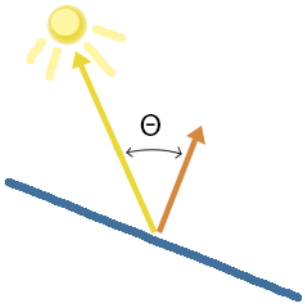


Figure 6: Diffuse lighting: How much light that is cast on a surface depends on its angle relative to the light source.

Besides light, colours were tweaked for improved appearance and a flight animation that moves the wings of an agent up and down according to their velocity was added.

4.3 Ethical aspects

The process of creating an application is usually not inherently unethical, but the resulting application might be used for malicious practices. Models for flocking behaviour can be applied in many different areas. It might be wise to imagine different employments of our simulation to prevent that it is being used in harmful ways. For example, any group of organisms where each organism follow a simple set of rules can probably be simulated with some minor tweaks to our implementation. As an example, flocking behaviour has been implemented for swarms of automated drones [22]. Since automated drones could be used for warfare or mass surveillance, it can be argued that flocking behaviour has contributed to the development of something that can be used for malicious purposes.

For our work, it is safe to assume that the result will not impact the research in the area in such a drastic way that it could become harmful in any way, shape or form. The worst thing that could possibly happen is that the simulation is too heavy for public property computer hardware so much that it breaks or becomes slow.

5 Visual results

This chapter details the visual specifications of a flocking simulation that is implemented according to the techniques described in earlier chapters. It contains pictures of the simulation ranging from game mechanics, visual effects and the visual representation of all game elements.

5.1 Images from the simulation

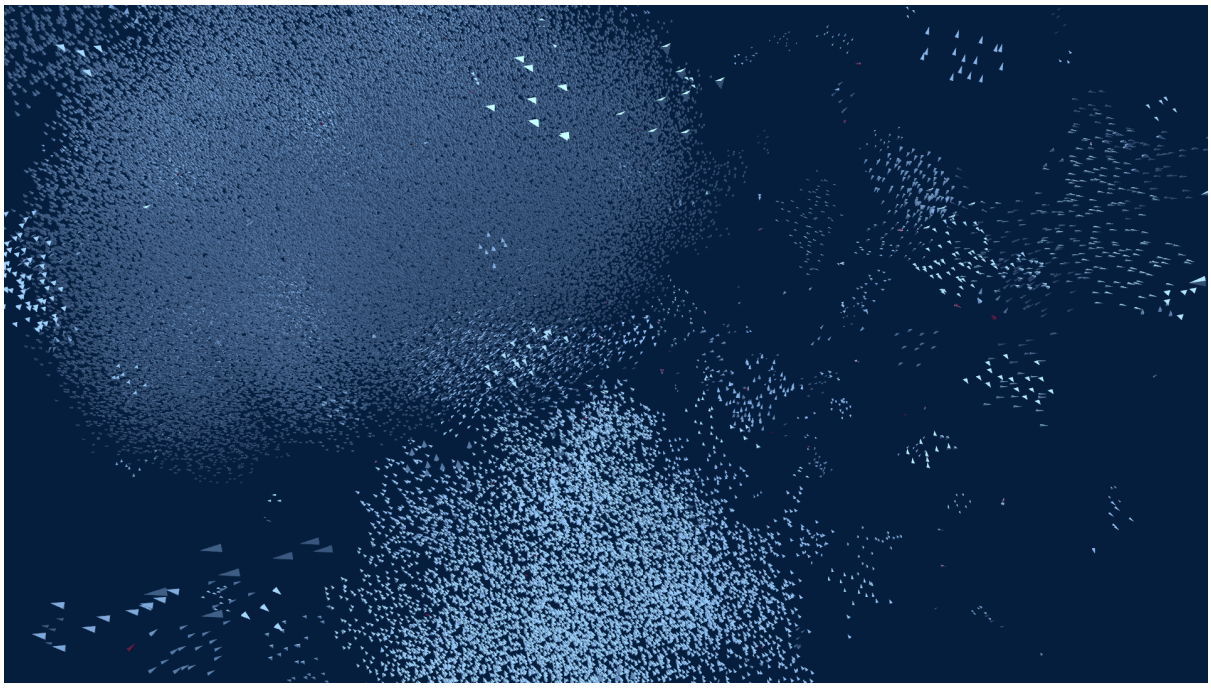


Figure 7: 100k agents simulated using CUDA

It is hard to show the flocking behaviour using only still images, but aspects to consider (Figure 7) include the coherent grouping of agents, the agents pointing in the same direction and the uniform distances between one agent to the another.

A simulation of light can be seen more closely in Figure 8. Each side of the model that is directed towards the light is lit up more than sides that are directed away from the light.

While the player presses the left mouse button a laser will appear, destroying agents and repelling any agents that are close enough. An example of this can be shown in Figure 9, where agents steer away from the laser line.

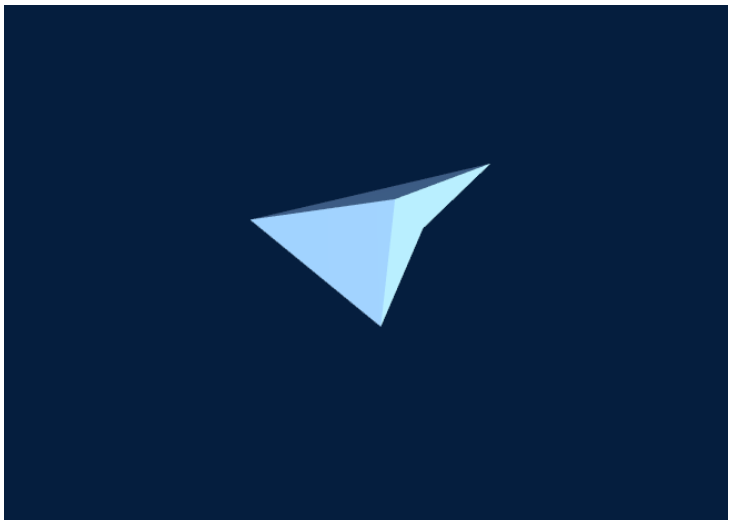


Figure 8: Light shed on the agent model.

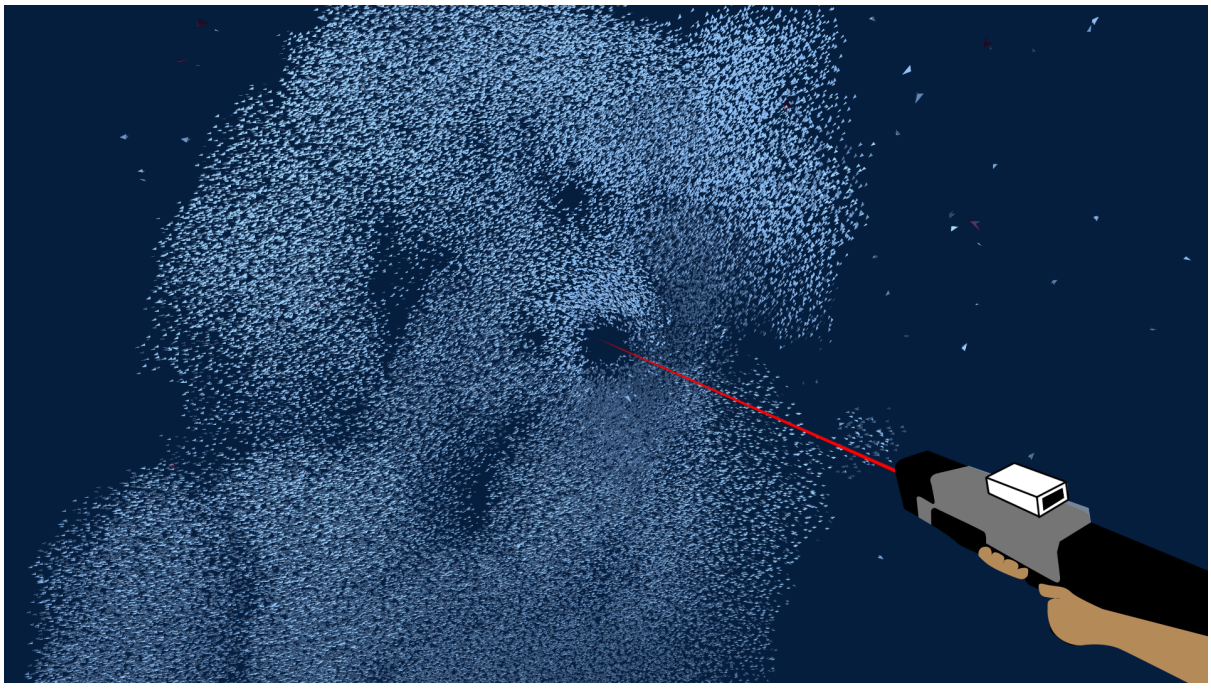


Figure 9: A laser beam projected from the player gun towards the center of the screen. Agents are repelled from the laser.

(a) Predator model (b) Prey model

Figure 10: Models of predator and prey

The predator and the prey agents differ in a few ways. Figure 10a shows how the predators are red and slightly larger than the prey illustrated in Figure 10b. It is harder to see how the prey flee away from the predators, but with a bit of imagination one can get an idea of how it works by looking at Figure 11.

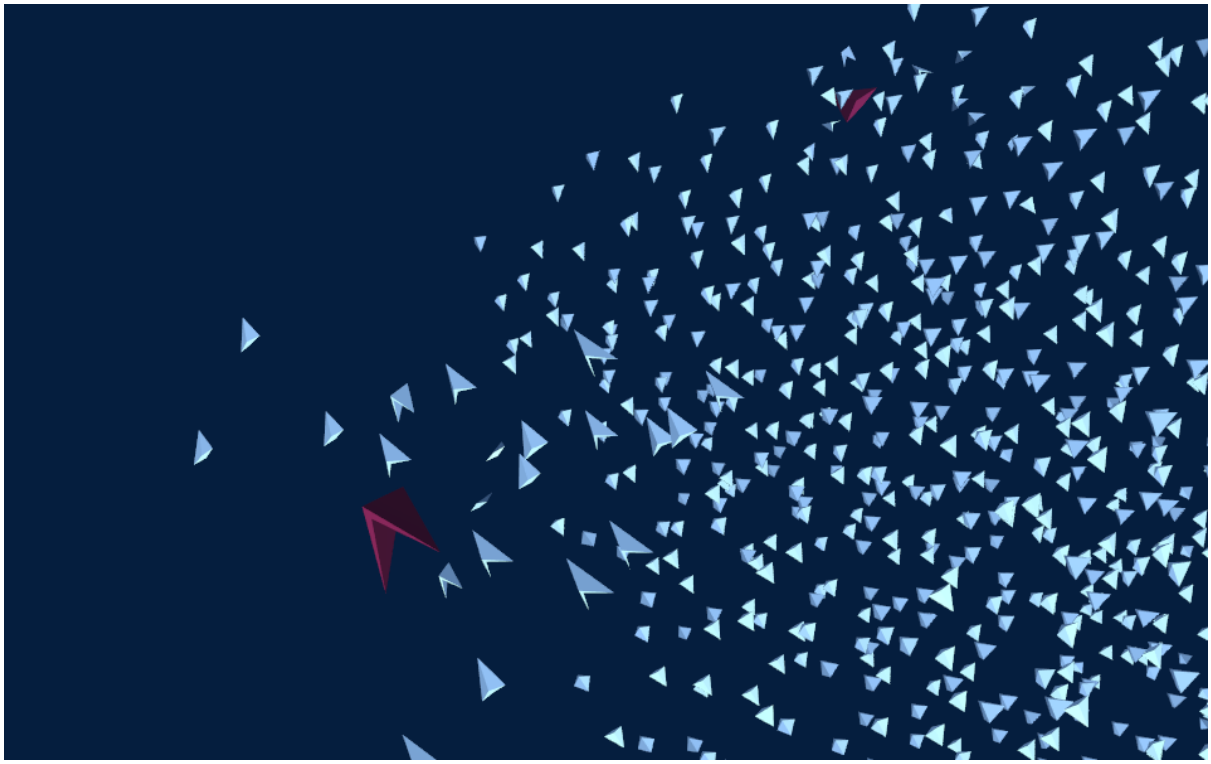


Figure 11: Two predators disrupting the ordinary flocking behaviours by instilling a bit of fear in nearby agents.

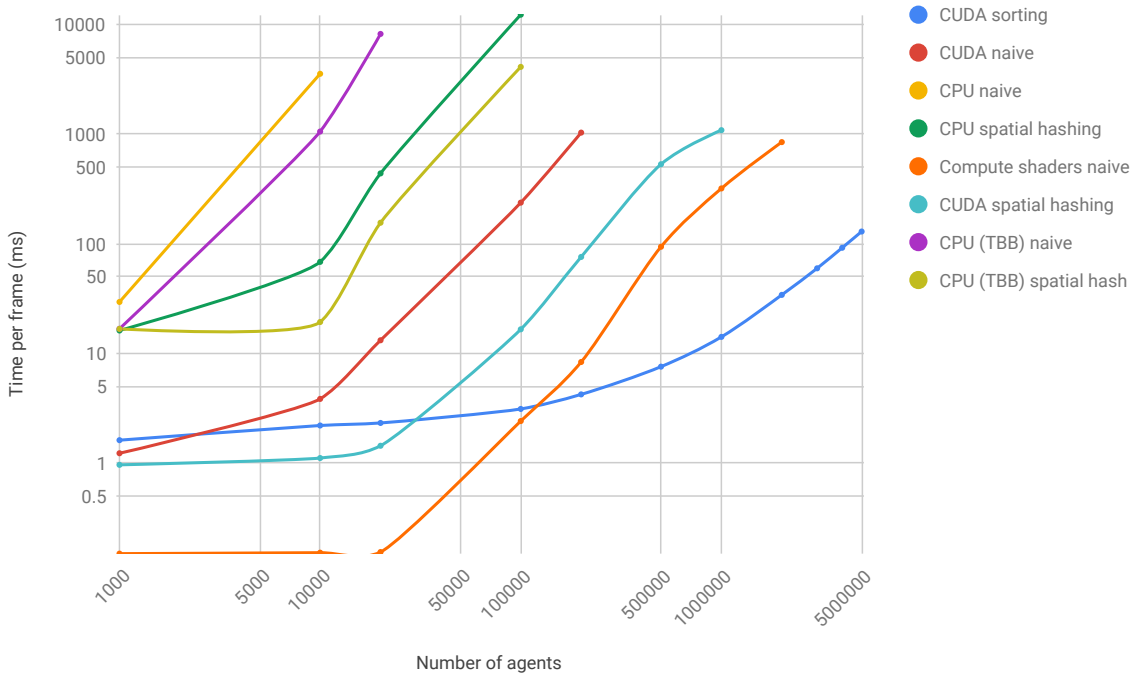


Figure 12: Measured performance of the different implementations. Note the logarithmic scales.

6 Performance of Different Implementations

In contrast to the visual presentation of the simulation in the previous chapter, this chapter focuses only on the performance of the different implementations of the simulation. The different implementations were detailed in chapter 4.

The different implementations were benchmarked using a computer with the following specifications:

- CPU: Intel Core i5-7600 3.50GHz, 4 cores
- Physical Memory (RAM): 32 GB
- GPU: NVIDIA GeForce GTX 1060 6GB
- CUDA cores: 1280
- CUDA driver version: 9.1.84
- Operating system: Microsoft Windows 10 Education

For each implementation, an average of 200 frames was recorded. All recorded times include both computing the flocking behaviour and rendering of the scene, if not stated otherwise. Only the flocking behaviour was tested; all interactive features were disabled in order to prevent any accidental interference.

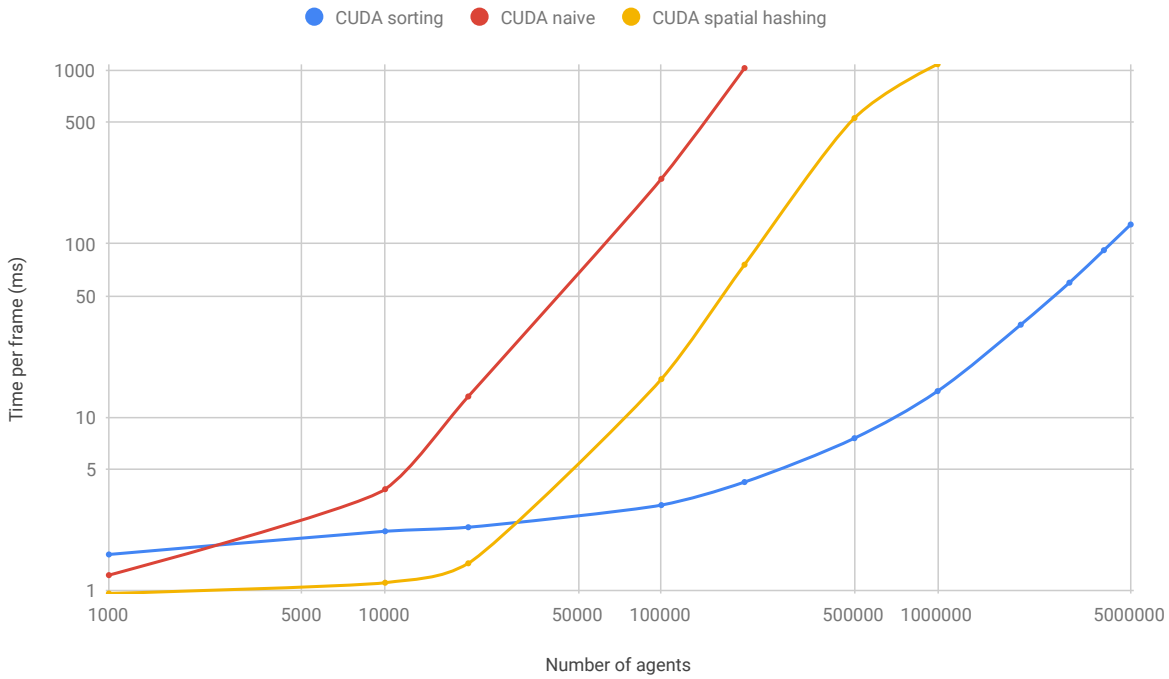


Figure 13: Performance of the CUDA implementations. Note the logarithmic scales.

6.1 Timing Compute Shaders

Whenever the CPU sends a task to the GPU, the CPU will do so and then directly continue its own work - not waiting for the GPU to finish all of the computations. This means that a timer on the CPU might not be accurate when we want to time processes on the GPU. Instead of using a timer residing on the CPU we used *OpenGL queries* to achieve asynchronous benchmarks for the Compute Shaders implementations. Querying for the results from the GPU gives a more accurate result but can stall the application if the results are not directly available [23]. Since we did not resolve this issue the results for Compute Shaders might seem somewhat worse than they are. Query objects were not needed for CUDA however, which has a built-in synchronization function that can be used instead.

6.2 CUDA implementations

Figure 13 shows the results for the CUDA implementations. All implementations utilise CUDA/OpenGL interoperability in order to keep the agent data on the GPU. Transfer of agent data between the CPU and the GPU therefore only occurs once, during initialization of the simulation. Batching is used during rendering.

6.2.1 Timing of individual tasks

Figure 14 shows the portion of time spent on each step of the "CUDA uniform grid with sorting" method. Each slice corresponds to a step described in section 4.1.3.

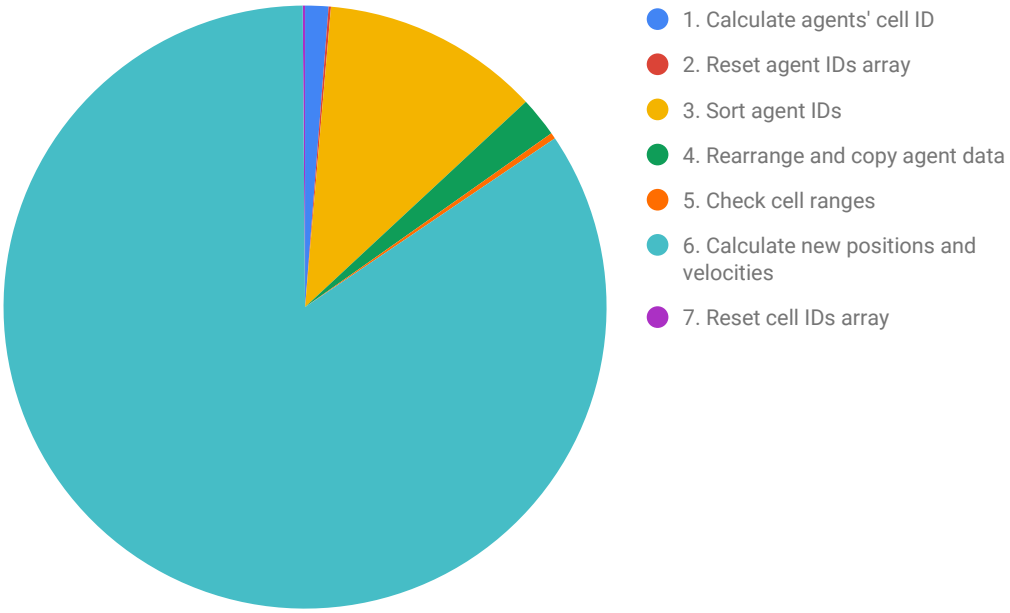


Figure 14: Breakdown of steps of the “CUDA uniform grid with sorting”. Number of agents: 2,000,000.

6.2.2 Z-order vs coordinate concatenation

Figure 15 shows performance for two different ways of constructing the cell ID in the “CUDA sorting” method. The recorded times are averages for a full step in the simulation, including rendering.

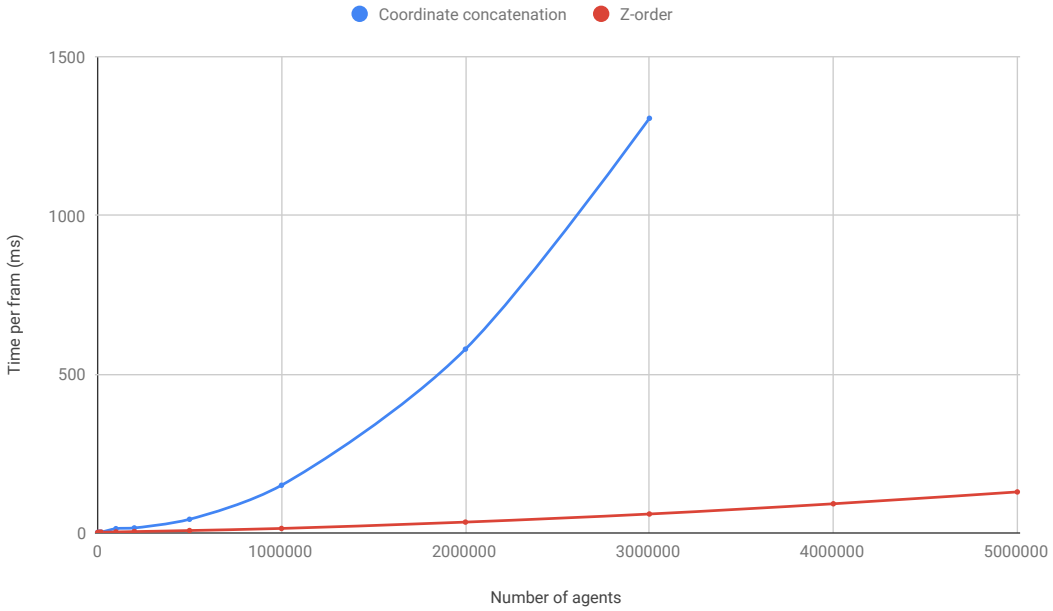


Figure 15: Simulation performance of two different ways of constructing the cell ID.

7 Analysis and Discussion

The results clearly shows that it's possible to increase the number of agents in an flock simulation by at least three orders of magnitude with the help of GPGPU and spatial data structures The results we achieved when we compared the different implementations met our expectations quite well. TBB increased the performance proportional to the number of CPU cores compared to the single threaded version, just as expected. Using spatial hashing instead of the naive approach always increased the performance. At 1,000 agents in CUDA, the naive approach was almost equal to the spatial hashing version. This is probably due to the fact that in that particular case, the number of CUDA cores is higher than the number of agents.

Up to 20,000 agents, spatial hashing outperforms sorting in CUDA. After that point, spatial hashing quickly becomes worse than sorting. One possible interpretation of this is that it's related to how neighbour data is stored and accessed. When agents are stored in the hash table, using the spatial hashing algorithm, their position in the array is basically random and will not correlate with their position within the world. Two agents that are close to each other in the world might be far away from each other in the hash table, thus they will be further away from each other in memory as well. When linking together cell sharing agents in the same bucket with the help of pointers in the `nextAgent` array, again, there is no correlation between the spatial distance of the agents and the distance in memory. For example, agents with ID a and b, where $a \ll b$ might be close neighbours in the scene, but their places in the `nextAgent` array is locked to indices a and b respectively. By contrast, in the sorting implementation there is a high probability that a and b's data will be close to each other in the memory after the sorting step.

When the number of agents is relatively low, the algorithmic simplicity of spatial hashing, compared to the more complex sorting, makes up for the fact that memory accesses are more scattered in the former case. But as the number of agents grows, inflating the arrays holding agent and cell contents,

the probability of cache misses increases. It seems like from somewhere around 200,000 agents and upwards, cache misses are so frequent that the spatial hashing simply cannot compete with the sorting approach.

Perhaps the most striking result is the performance impact of using Z-order encoded cells, instead of simple coordinate concatenation. Already at 500,000 agents, the Z-order performs better by an order of magnitude. Similarly to the spatial hashing versus sorting comparison, the performance of Z-order and concatenation are very similar up to 20,000 agents. Our interpretation is, again, that this is due to memory access patterns. As explained in section 3.5.3, spatially close cells will on average end up closer in memory after sorting when they are encoded by Z-order compared to coordinate concatenation. In other words, coordinate concatenation is worse at preserving relative distances between the agents. When array sizes increases, the absolute distances between agents whose locality isn't perfectly preserved increases, making the probability of cache misses during neighbour search increasingly higher.

We used OpenGL queries to synchronise the CPU and GPU so that we could measure performance for Compute Shaders correctly. Still the results that we acquired seemed to be to good to be true. For example, 50,000 agents resulted in an update time of less than 0.2 milliseconds per frame which is very low compared to about 10 milliseconds per frame for CUDA. This results suggests that either Compute Shaders are very efficient, or that the timer did not synchronize properly with the GPU. Sadly, we didn't have time to investigate this matter further.

7.1 Accurately representing a flock

The rules that govern the agent simulation are sufficient to give good flocking behaviour. However, if one only uses the three basic Reynolds rules, some problems arise. The simulation will be very rigid, since the agents have no other goal than to cluster, and follow the path of the rest of the flock. This means that they will merge into large spheres that move in a uniform direction.

The rigid motion of the flock is a result of the rigid motion of individual agents themselves. Simply put, once the rules are balanced, each individual agent will have no reason to ever move in a different direction than the flock, and this isn't accurate to how flocks move in nature. For example, it has been observed that birds can randomly shoot off in a playful manner whereafter the flock follows that bird as if it was a leader [8]. It is not clear to us how such a behaviour can be implemented as it seems like the agents would need to temporarily set aside some of the other flocking rules in order to act as, or follow, a leader. An attempt was made to "loosen up" the rigid behaviour by adding add a random vector each agent, that is added to its final velocity by a certain probability. However, it resulted in a flickering behaviour rather than a smooth, more dynamic movement as we had hoped for. It also had no effect on the behaviour of the flock at the macro level, probably because vectors in random directions over a large sample will result in a net change of close to zero.

These are problems that can be dealt with by introducing new rules to the simulation, but introducing new rules is an inherent problem in itself. Many complex systems are hard to accurately model. This means that what may seem like good behaviour may be completely unnatural. For example, we have added behaviour that ensures that the agents do not collide with the boundaries in the model, but the interaction that is done when this occurs is very sudden. This is also apparent in how we tried to add some randomness to the movement of the agents, as stated above.

Moreover, one very concrete example of this would be to add leading agents, that direct the flocks in

some way, either scripted or according to some AI system. However, this is inaccurate to how most flocking behaviour is done, where there is no leader at all, especially not one with more complex or different behaviour.

Hence, We need to be careful of what rules we should add to the simulation, and what the results of these rules represent. it is of our belief that the additional rules should act dynamically, and we should work on improving the simulation by indirect measures. I.e, rules that add other kinds of motivated behaviours, such as hunger, predators or other needs.

Another pressing problem is that the agents by design lack long-range perception. When an agent diverge so far from the flock that no neighbours are within its scope, the agent aimlessly flies around in the scene until it by chance encounters some flock mates again.

When it comes to combining the rules, the prioritised acceleration allocation approach yielded more pleasing results than the weighted average approach. When using the former, behaviour is more dynamic since individuals in the flock can choose to temporarily ignore rules with lower priority in certain situations. This is especially evident when a predator attacks the flock, the individuals closest to the predator make forceful attempts to avoid it, causing a disturbance that propagates throughout the flock.

The behaviour of the predators needs refining but is nonetheless interesting. When predators approach a flock or flies through it, it does look like they find it hard to single out individual prey to attack. But once they succeed in forcing an individual out of the flock, they often pursue that individual. So even if the rule dictates that the predator should aim for the average position of neighbouring prey, they are successful in catching prey as soon as their surrounding is not very crowded.

7.2 Challenges and difficulties with the project

Developing the application has been a lot harder than we expected. In order to have tight control of the optimisation techniques we use, we developed the application in C++ and OpenGL. We could have developed our application in a platform such as Unity or Unreal Engine, and completed it in a shorter time, but then we wouldn't have learned as much about low level optimisation techniques. Tasks that are trivial in game development platforms, such as rendering a simple triangle and applying transformations to it, took us some time to accomplish. But even though we had to deal with this slow progression of things, developing in this fashion also made the project more rewarding. Every little progress could be attributed to someone's improved knowledge. It was, however, a problem that some of the group members had previous knowledge with graphics programming while others didn't, this made the distribution of the programming contribution quite uneven.

One of the larger challenges has been where on where to put our focus, as many of the features are either working completely or not working at all, especially the GPU optimisation. This means that it has been a continuous issue on what parts of the project requires most effort for optimisation; some of the features that we have implemented have been made irrelevant, or had a very low impact, while other features, such as when we got CUDA to work for the first time had a tremendous impact on performance when simulating large numbers of agents.

Another one of our big issues have been dealing with the game aspects of the project. It is not obvious how massive flocks, with millions of agents, can be made relevant in a game setting. Already at 100,000 agents the flocks becomes hard to overview, and adding more agents does not exactly improve the gaming experience. In the end we settled for fewer agents in the game, but still benchmarked larger

flocks to get more data on how efficient our applications is.

Finally, and perhaps most crucially, we didn't have enough time to implement all combinations of techniques and neighbour search algorithms that we intended. Some combinations are more interesting than other. For example, it's highly unlikely that a single-threaded CPU implementation of the sorting method would outperform our current best-performing implementation, but it would be interesting to implement the sorting method using compute shaders.

8 Future Work

There are many possible optimisations that we did not have enough time to implement, or thoroughly research. In this section, we will describe these methods, and why we believe that they would be interesting for our project.

8.1 Improvements to spatial hashing approach

One drawback with our spatial hash implementations is that agents are spread out more or less random across the hash table. If we instead had used *locality-sensitive hashing*[24], we could have preserved spatial coherence, possibly resulting in fewer cache misses and better performance.

In our spatial hash implementations we used an array (`nextAgent`) with agent pointers to link together agents residing in the same bucket. This may be efficient in terms of memory usage, but it introduces a lot of pointer hopping, which can degrade performance. One possible improvement to this could be to copy and rearrange the `nextAgent` array so that linked together agent pointer are stored contiguously. Iterating over bucket sharing agent pointers would then be matter of starting at the bucket head, accessing the `nextAgent` entry for the head, and then access subsequent array entries until you reach the entry pointing at the tail. Such a rearrangement can be done in $\mathcal{O}(n)$ time, but would require an extra pointer for each head in the bucket that indicates the start of the chain in `nextAgent`.

Furthermore, if the `nextAgent` array were to be rearranged in the above suggested way, the agent data itself could be rearranged accordingly. In that case, at least agents residing in the same bucket would have its data stored contiguously in memory, and that would certainly improve memory access patterns.

8.2 Improvements to the sorting approach

As already mentioned, using Z-order encoded cells in the CUDA sorting implementation increased performance significantly by preserving some spatial locality in the agents data. When the agents perform neighbour search, however, they always check neighbouring cells in the same order: column major order. An alternative approach would be to check the cells in order by their Z-order ID. Doing it this way would ensure that potential neighbour agents are accessed in the order they are stored in memory, improving memory access patterns even further.

In the tested CUDA sorting implementation, the start and end indices indicating each cell's range in the agent array are stored in two arrays (`cellStartIndex` and `cellEndIndex`). The cell ID is used as an index when storing the range information. For example, the start index for a cell with ID i is stored at `cellStartIndex[i]`. This of course means that the number of cells, and in turn the space of the scene, must be limited. However, if one were to use a hash table for storing the cell ranges, it would be possible to use an unlimited space, much as in the case of the spatial hashing approach. Such a hash table would preferably use a locality-sensitive hashing, since the lookups would be clustered around the cells containing agents.

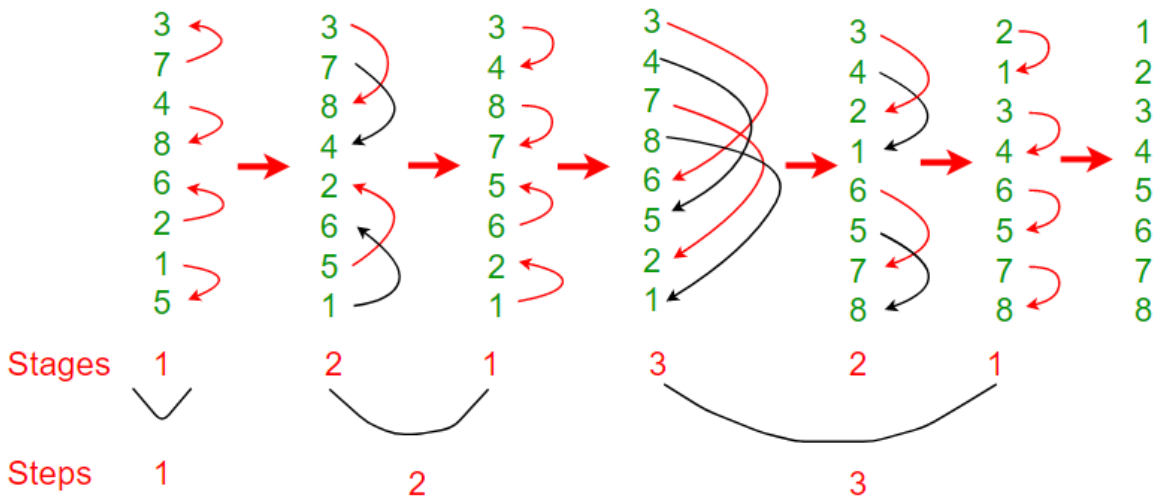


Figure 16: Illustration for the different steps in Bitonic sort.

8.2.1 Bitonic sort

A potential alternative to parallel radix sort is a sorting algorithm called Bitonic sort. This is a parallel algorithm suitable for the GPU, that sorts an array in $\mathcal{O}(\log^2 n)$ parallel time. This is done by merging sequences into larger bitonic sequences, starting by adjacent numbers, and then merging these bitonic sequences. This is done until the whole list has become a bitonic sequence - a bitonic sequence is a list with no more than one local maximum and one local minimum. Endpoints are considered to be wrap-around points for the sequence. If we then take this sequence and compare/replace the corresponding elements in one half to the second half, and recursively do this for the whole array, we will then get a sorted list, in efficient parallel time. An illustration of this can be seen in Figure 16.

This algorithm is more efficient at sorting large data sets ($n \geq 1M$) over the more standard radix sort for parallellised computing sort.**bitonic**

8.3 Framebuffer Objects

We tried to implement another GPU approach that uses something called *Framebuffer Objects*, but we did not get it to work properly. If we had more time we would spend more time on get this version to work. Framebuffers are normally used when something is being rendered on screen (see Figure 3) but they could also be used to render things that does not appear on screen. Basically we can utilise the fact that what is being rendered to the framebuffer objects are stored on the GPU. The data for each agent can be represented as pixels of these rendering targets. When the fragment shader processes the information of these rendering targets it would do so in parallel for each pixel, meaning that all agents would be updated in parallel.

8.4 Additional work with Compute Shaders

Compute Shaders became our last resort of another GPGPU approach than CUDA, something to compare CUDA with. But since it was quite hard to get Compute Shaders to work we only had time to

implement and test it with the naive neighbouring algorithm. If we had more time we would have tried different approaches just as we did with the CUDA version.

8.5 Reducing the number of re-calculations for positions and velocities

One way to improve the number of agents we can simulate would be to reduce the amount of times we calculate new positions and velocities. As seen in Figure 13, one can tell that much of the processing time per frame is spent on this. However, since each frame only amounts to minuscule movement, one alternative would be that the agents only update velocity & direction every few frames instead.

If one ensures that the agents update at different intervals, they could each ensure that they don't collide when they do re-calculate velocity and direction. One idea here would also be to have a value for how much the different rules affect the agent, and update more frequently if that number is high.

However, it may be an issue where more calculations are made than what is actually minimized. Still, the concept of letting their collective behaviour be more dominant than their individual influence should still be usable to minimize the time spent on unnecessary calculations.

References

[1] S. Camazine, N. R. Franks, J. Sneyd, E. Bonabeau, J.-L. Deneubourg, and G. Theraula, *Self-Organization in Biological Systems*. Princeton, NJ, USA: Princeton University Press, 2001, ISBN: 0691012113.

[2] A. Cavagna, A. Cimarelli, I. Giardina, G. Parisi, R. Santagati, F. Stefanini, and M. Viale, "Scale-free correlations in starling flocks", *Proceedings of the National Academy of Sciences*, vol. 107, no. 26, pp. 11 865–11 870, 2010.

[3] D. Galas, "Systems biology", *Britannica Academic*, [Online]. Available: https://academic.eb.com/levels/collegiate/article/systems-biology/601866#326371.toc (visited on 04/30/2019).

[4] Y. Kim, J. Falletta, and S. Kelly, "Traffic is complex, but modelling using deceptively simple rules can help unravel what's going on", 2018-06-07. [Online]. Available: https://theconversation.com/traffic-is-complex-but-modelling-using-deceptively-simple-rules-can-help-unravel-whats-going-on-92833 (visited on 05/16/2019).

[5] M. E. Goldsby, C. M. Pancerella, and Ieee, "Multithreaded agent-based simulation", in *Winter Simulation Conference on Simulation - Making Decisions in a Complex World*, ser. Winter Simulation Conference Proceedings, NEW YORK: Ieee, 2013, pp. 1581–1591, ISBN: 978-1-4799-3950-3; 978-1-4799-2077-8. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6721541 (visited on 05/16/2019).

[6] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model", in *14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*, M. C. Stone, Ed., Association for Computing Machinery, Inc, pp. 25–34, ISBN: 0897912276 (ISBN); 9780897912273 (ISBN). DOI: 10.1145/37401.37406. [Online]. Available: https://dl.acm.org/citation.cfm?id=37406 (visited on 05/16/2019).

[7] O. Blomqvist, S. Bremberg, and R. Zauer, *Mathematical modeling of flocking behavior*. 2012-05-25. [Online]. Available: http://kth.diva-portal.org/smash/get/diva2:561907/FULLTEXT03.pdf (visited on 05/16/2019).

[8] C. Hartman and B. Benes, "Autonomous boids", *Computer Animation and Virtual Worlds*, vol. 17, no. 3-4, pp. 199–206, 2006.

[9] L. Spector, J. Klein, C. Perry, and M. Feinstein, "Emergence of collective behavior in evolving populations of flying agents", *Genetic Programming and Evolvable Machines*, vol. 6, no. 1, pp. 111–125, 2005.

[10] J. D. Owens, "A survey of general-purpose computation on graphics hardware", 2005. [Online]. Available: https://www.cs.utexas.edu/~pingali/CS395T/2009fa/papers/GPUSurvey.pdf (visited on 05/16/2019).

[11] I. Corporation. (2019). Thread building blocks, [Online]. Available: https://www.threadingbuildingblocks.org/ (visited on 04/09/2019).

[12] N. H. Graham Sellers Richard S. Wright Jr., *OpenGL SuperBible: Comprehensive Tutorial and Reference, 7th ed.* Pearson Education, Inc., 2016, ISBN: 978-0-672-33747-5.

[13] R. R. John Kessenich Dave Baldwin, "Glsl language specification, version 1.10.59", 2004. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.1.10.pdf (visited on 05/16/2019).

[14] R. J. Hovland, "Latency and bandwidth impact on gpu-systems", 2008-12-17. [Online]. Available: https://pdfs.semanticscholar.org/8456/cd707e76842aafa396add87ccf7cff5ac0fe.pdf (visited on 05/16/2019).

[15] N. Corporation. (2019). Cuda gpus, [Online]. Available: https://developer.nvidia.com/cuda-gpus (visited on 04/04/2019).

[16] NVIDIA Corporation. (2019). Nvidia developer documentation, section 3.2.12. graphics interoperability, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#graphics-interoperability (visited on 04/30/2019).

[17] M. Wloka, "Batch, batch, batch", 2003. [Online]. Available: https://www.nvidia.com/docs/IO/8228/BatchBatchBatch.pdf (visited on 05/16/2019).

[18] S. Green, "Particle simulation using cuda", *NVIDIA whitepaper*, vol. 6, pp. 121–128, 2010-05.

[19] D. Guo and M. Gahegan, "Spatial ordering and encoding for geographic data mining and visualization", *J. Intell. Inf. Syst.*, vol. 27, pp. 243–266, Nov. 2006-10-21. DOI: 10.1007/s10844-006-9952-8.

[20] J. M. Jeschke and R. Tollrian, "Prey swarming: Which predators become confused and why?", *Animal Behaviour*, vol. 74, no. 3, pp. 387–393, 2007.

[21] J. de Vries. (). Learnopengl - basic lightning, [Online]. Available: https://learnopengl.com/Lighting/Basic-Lighting (visited on 05/16/2019).

[22] S. Hauert, S. Leven, M. Varga, F. Ruini, A. Cangelosi, J. C. Zufferey, D. Floreano, and Ieee, "Reynolds flocking in reality with fixed-wing robots: Communication range vs. maximum turning rate", in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, ser. IEEE International Conference on Intelligent Robots and systems, NEW YORK: Ieee, 2011, ISBN: 978-1-61284-455-8. [Online]. Available: %3CGo%20to%20ISI%3E://WOS:000297477505057 (visited on 05/16/2019).

[23] Khronos, "Query object", 2019. [Online]. Available: https://www.khronos.org/opengl/wiki/Query_Object (visited on 05/16/2019).

[24] S. Lefebvre and H. Hoppe, "Perfect spatial hashing", in *ACM Transactions on Graphics (TOG)*, ACM, vol. 25, 2006, pp. 579–588.