Developing a massive real-time crowd simulation framework on the GPU

October 13, 2016

Guillaume Payet
gvp13@uclive.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury, Christchurch, New Zealand

Supervisor: Dr. Ramakrishnan Mukundan
mukundan@canterbury.ac.nz

Abstract

Crowd simulations are used to imitate the behaviour of a large group of people. Such simulations are used in industries ranging from video-games to public security. In recent years, research has turned to the parallel nature of GPUs to simulate the behaviour of individuals in a crowd in parallel. This allows for real time visualisation and interaction with massive crowds and/or their environment. There are, however, no tool that could facilitate the integration of GPU-accelerated crowd simulations in an existing program like a game.

This paper proposes an OpenCL port of the PedSim library. It is a tool that allows CPU-run crowd simulations to be used in another application. It uses the social forces algorithm which is a crowd simulation method using a set of forces to compute each agents motion. Our implementation is compared in performance and functionality with the original PedSim. Finally, we discuss the functionalities missing from our implementation to provide a more complete crowd simulation.

Acknowledgements

I would like to thank Dr. Mukundan Ramakrishnan for all his supervision, support, feedback and patience throughout this year. I would also like to thank Prof. Andrew Cockburn and Dr. for their feedback and advice which might not seem much to them but were much appreciated. Lastly, I would like to thank all my friends and family who kept faith in me, supported me and cheered me on when I faltered.

Contents

1 Introduction

In this paper, we will investigate the tools available for crowd simulation development and assess whether or not they provide some sort of GPU acceleration and if not, how simple it would be to integrate that feature into them. We will then develop our own tool based on one of the non GPU-capable tools.

1.1 Crowd Simulations

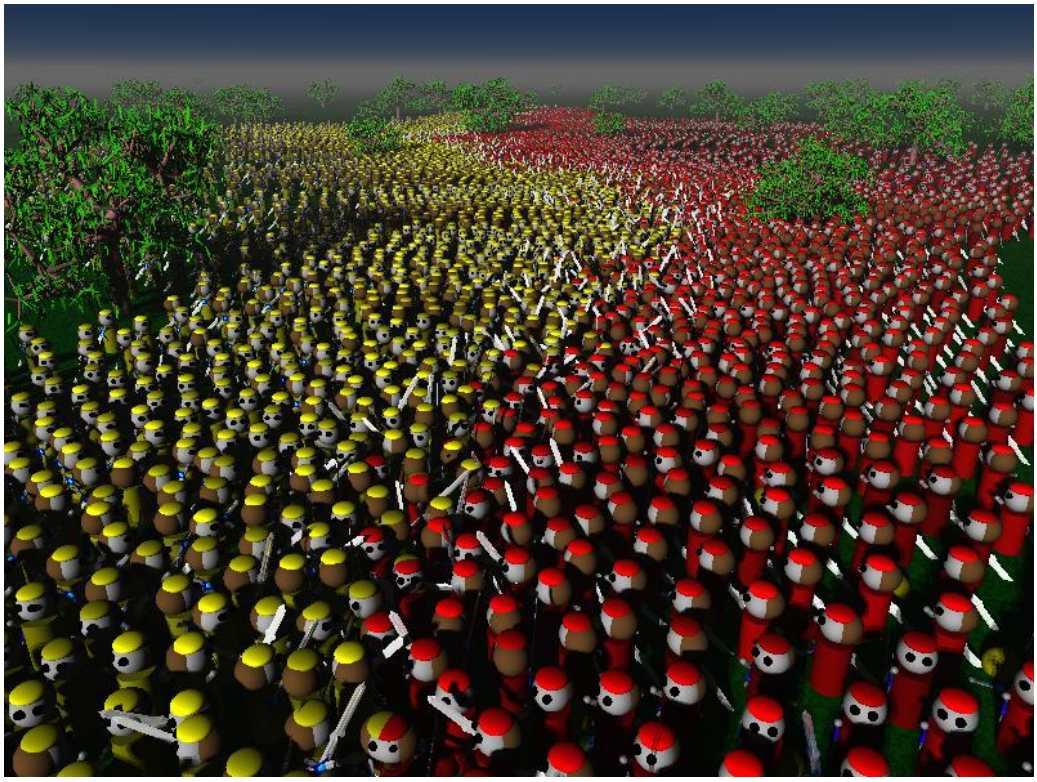

Figure 1.1: Screenshot of a film made using crowd motion precomputed by PedSim.

The goal of crowd simulations is to realistically imitate the behaviour and motion of large numbers of people. Such simulations have already shown their uses in a wide range of scenarios. For example, in video-games and cinema [3], they can provide a convincing sense of life in a scene. Another use of crowd simulations is to plan evacuation plans for large gatherings.

Despite the fact that crowd simulations have been studied and researched for a long time, it is a very young field of research. The first few mentions of crowd simulations date back to the 70s (e.g.: [27]). Since then and until now, research has been largely directed towards modelling realistic motion and behaviour.

1.2 General Purpose GPU

In recent years, There has been a large amount of interest towards accelerating general purpose algorithm using Graphics Processing Units (GPUs). This concept is called General Purpose GPU (GPGPU). GPUs are vector processors and as such are designed to process data in parallel. This is what makes them so efficient at accelerating some algorithms.

The scenarios which benefit the most from GPGPU are data-parallel problems [10]. These are made of iterative computations processing mutually exclusive data. Crowd simulations belong in that category. The computation of an agent's next move does not affect any other agents (only its current state does). Crowd simulations can therefore greatly benefit from GPGPU. As well as accelerating simulations and allowing more dynamism in them (i.e.: a player in a game could interact with the crowd or environment and see an immediate reaction), this can allow more complex and realistic (and slow) models to be used in real-time.

1.3 Crowd Simulation Tools

Crowd simulations can be considered a combination of several fields in computer science and social sciences. These are two different types of research fields which have little in common. Tools are designed to facilitate research from both sides. Here, we will look at two categories of tools:

Development: Tools used to develop new crowd simulation techniques by providing an environment with all required elements of crowd simulations and a basic, expandable structure. The "computer science" aspects are abstracted to allow the user to focus on the model.

Integration: Libraries used to easily integrate a crowd simulation scenario in an application. These tools provide ways to describe an environment, agent count, starting positions, goals and handle the simulation process. Everything is abstracted to allow a user to simulate crowds without any knowledge of the science involved.

There are also standalone crowd simulations available. Even though some of them are GPU-implemented, they are not designed as reusable libraries so have little relevance to this paper.

	Still active	Type	Algorithm(s)	Rendering API	Parallel
SteerSuite [32]	Yes	Framework (Development)	Social forces, RVO2 and PPR	OpenGL	No
Menge [12]	Yes	Framework (Development)	Social forces, Johansson, Karamouzas and Zanlungo	OpenGL	No
Brainiac [16]	No	Application (Development)	Sound-based custom algorithm	OpenGL	No
PedSim [18]	Yes	Library (Integration)	Social forces	None	No
CORPSE [26]	No	Application (Development)	Continuum crowds	OpenGL	No

Table 1.1: Crowd simulation tools found on the web. Few are freely available, fewer are active but none provide GPU acceleration.

Table 1.1 shows the list of tools available. It is obvious that none of them are GPU-implemented. Due to the implications of GPU programming, none of them are GPU capable either (more details on that in the background chapter).

There is therefore a need for a GPU-accelerated tool. This research attempts to fill this need by developing a crowd simulation library that can be integrated in any OpenGL application. This tool is a GPU implementation of PedSim. This library is based on a popular crowd simulation algorithm, the social forces

model. This method defines the motion of each agent in the crowd as a set of forces. By assessing the performance of this algorithm in a GPU implementation, we can draw conclusions about more optimised methods.

1.4 Research Goals

The goal of this project is threefold:

- To review GPGPU APIs in the context of crowd simulation. Additionally, since both operate of the same device (i.e.: the GPU), the interoperability between GPGPU APIs and rendering APIs will be investigated.
- To identify within an existing codebase which aspects, elements or computations can benefit from GPGPU.
- To develop a GPU-accelerated tool for crowd simulation development or integration and assess its performance in terms of framerate and how much improvement is achieved from the CPU counterpart.

The expected outcome of this research is a GPU-accelerated tool derived from one of those listed above which benefit from GPGPU at a level which could render a previously unusable algorithm in real-time fast enough to be interactive.

The results will also include analysis of the variability of the framerate when using different parameters in the GPGPU API (i.e.: we discuss optimisation in regards to the API).

1.5 Structure of the Report

This report has six chapters and this is the first one. In the second chapter, "Background", we describe the concepts behind GPGPU and crowd simulations. There is also a brief comparison of GPGPU APIs and a description of the one used in our experiments, OpenCL. The algorithms being implemented and PedSim are also described in that chapter.

The third chapter, "Design and Implementation", the process of development is detailed. We describe how each force is computed in PedSim, then how the library was modified to be "parallel-friendly", finally we explain the hurdles met and the optimisations performed while converting from a C++ codebase to an OpenCL kernel. Additionally, we describe how the data was recorded for analysis.

The "Results and Evaluation" chapter contains analysis of the data gathered and brief conclusions about what they mean. We proceed in steps from the original PedSim implementation to the final OpenCL version running in parallel on the GPU with optimisations.

In the chapter "Discussions and Future Work", we discuss what the results mean in more detail and assess whether they match with our expectations. The possible additions to the library are briefly explored. Future research related to space partitioning and implementing other algorithms on GPU are discussed.

Finally, in the last chapter, "Conclusions", we conclude the paper by summing up why this project was carried out, how we proceeded, what results were obtained and whether they matched our expectations.

2 Background

On a single-core CPU, running a program across multiple software threads will make the operating system switch between each thread to provide concurrent execution. However, on a multi-core CPU, the operating system will dispatch the software threads across the CPU cores, providing a parallel execution [33]. This may provide significant advantages for some applications.

2.1 GPGPU

CPUs are generally designed with few physical cores (two and four in most cases) (e.g.: the CPU used in this paper has 4 cores at 2.3 GHz [9]) running at high frequencies (around 2 to 4 gigahertz). They are designed for general purpose computing and can manage a limited amount of parallelisation. GPUs, on the other hand, were designed to compute and display graphics. Computer animations require to show at least 50 frames per second to present fluid motion. These images are made of millions of pixels (i.e.: picture elements, points). For example, a standard Full High Definition (FullHD, 1920 by 1080 pixels) screen is made up of 2 million pixels. In other words, the GPU requires to compute the colour of at least 100 million pixels per second. Each computation being largely independent, GPUs were designed with a much larger number of cores compared to CPUs (generally ranging from 500 to 2000) and these cores run at lower frequencies (between 200 and 1000 megahertz) (e.g.: the GPU used in our work has 640 cores at 936 MHz [11]).

It is only at the start of this century that GPUs were added programmable shaders. These were designed to add more flexibility to the rendering process. Researchers also realised that those shaders could be used in non-rendering context, like for example, matrix processing. This was how general purpose GPU was first introduced [13]. The field involves studying and developing algorithms to be run in parallel on the GPUs and optimising them to get as much benefit from the many parallel cores.

2.1.1 Workload Partitioning and GPU Cores

When executed on a vector device such as a GPU, workload is split into parallel-executed workgroups. High-end GPUs generally have clusters (or shader cores depending on manufacturer) of 16 SIMD units to allow for efficient processing of high throughputs [17]. SIMD (Single Instruction Multiple Data) units are ALU processors which can apply operations on several data with a single instruction. GPU architectures dictate optimal sizes for workgroups. For example, the AMD HD7970 uses clusters with 4 lanes of 4 SIMD units each (16 units) and are optimised to process optimally 4 clusters (64-item workgroups). NVIDIA's GTX 850M uses "shader cores" containing 2 lanes of 16 SIMD units each. That way each core can optimally handle 32-item workgroups (See figure 2.1 for the block diagram of the GPU used in our experiments).

2.1.2 GPGPU APIs

There are several application programming interfaces (APIs) to develop GPGPU programs. Among them, only four are widely used and supported. The most popular API is NVIDIA's CUDA (the Compute Unified Device Architecture). It is a proprietary API developed by NVIDIA and only works with NVIDIA GPUs. This API provides highest performance than the others on NVIDIA GPUs and can access some of the advanced features of the cards. Fang et al. [15] investigated the performance difference between CUDA and the second most popular option, OpenCL. Their work shows that if proper optimisations are used,

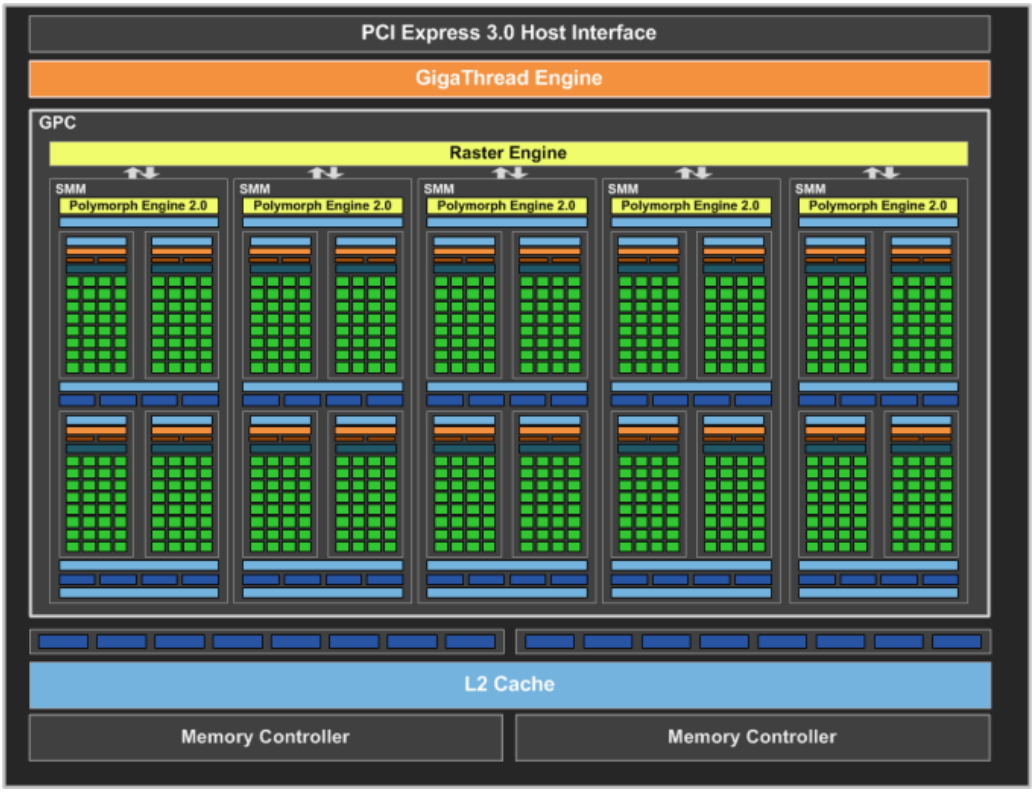

Figure 2.1: The NVIDIA GTX 850M contains of 5 streaming multiprocessors (nicknamed SMM for the Maxwell architecture) each containing themselves four clusters of 32 SIMD units (green blocks) for a total of 640 cores. [29]

CUDA provides a negligibly small performance advantage on OpenCL.

OpenCL is the one that is used in this paper and, thus, is described in the subsection below. The third most popular API is OpenGL. OpenGL, or the Open Graphics Library, is an open-source API which was exclusively designed for graphics rendering in 1992. With version 1.5 in 2003, extensions were added to allow shaders. Shaders are programs run on the GPU as part of a programmable pipeline. This gives more control to the developer. Later on, version 4.3 introduced compute shaders designed specifically for general processing unrelated to the rendering pipeline. As such, the shaders can be executed outside of the rendering cycle. OpenCL is generally preferred over OpenGL compute shaders as it provides more features and flexible memory models.

Finally, Vulkan is a compute and rendering API which was released this year. At the start of this research, it was unsure whether Vulkan would have been a candidate API. Now that it has been used, Vulkan has demonstrated better performances than OpenGL and can be used for GPGPU (through compute shaders, similarly to OpenGL). Vulkan uses an intermediate representation for shader code (called SPIR-V) which can be obtained by compiling either OpenCL kernels or OpenGL shaders. This means that the advanced features of OpenCL can be used along with Vulkan's better performance.

OpenCL

For our purposes, OpenCL was chosen as GPGPU API. OpenCL, the Open Computing Language, is an open-source standard designed and maintained by the Khronos group (who also maintain OpenGL and Vulkan). It is designed specifically for general purpose computing on the GPU (as opposed to OpenGL and Vulkan) [17].

Version 1.2 was released in 2011 and is the most widely supported across a wide range of manufacturers and devices. In this version, a variation of the C language is used to write the kernels which are then compiled at run-time by the GPU. This process is similar to the compilation of OpenGL shaders.

In OpenCL, NVIDIA's "shader cores" are referred to as "Compute Units" (CU). When running a parallel algorithm, the same program ("kernel" in OpenCL) is executed simultaneously on different inputs. Each instance of the kernel is called a workitem and they are bundled in workgroups. All workgroups have the same size in an execution and the total number of workitems must be divisible by the workgroups' size (i.e.: all workgroups must be full).

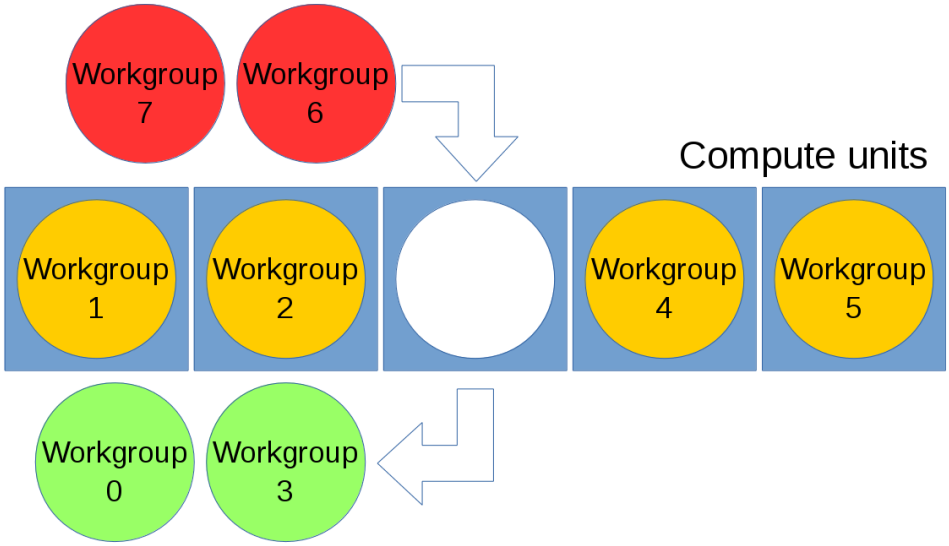

Figure 2.2: Structure of an OpenCL queue. The workgroups are queued and the workitems within them are executed in parallel by the streaming multiprocessors.

The workgroups are then queued in the GPU and processed by the compute units (see figure 2.2). There may be only one workgroup processed by one CU at a time. The compute units execute the workitems in parallel by queueing them through the SIMD units. When a workitem is completed, another is handled by the SIMD unit [17].

2.2 Crowd Simulation

Crowd simulation involves several different research areas, including computer graphics, artificial intelligence and sociology among others. This may make crowd simulation algorithm complex depending on the chosen method's level of realism. Though the research field is fairly new, there are basic principles and elements that can be found in all implementations. There are four basic elements interacting in a crowd simulation and these are the same as found in path planning algorithms [2]:

Agents: The simulation's actors. In path planning algorithms, there is only one agent but in crowd simulations, several navigate the same environment and must know of each other to avoid collisions by steering or adjusting their speed.

Goals: The agent's destinations. Unless an agent is to be static, it will have at least one goal. In path planning algorithms, these are the goal node (desired state in the search graph) and is treated similarly in crowd simulation algorithms. When several goals are available to an agent, a goal selection scheme may be used (just like in path planning algorithms).

Obstacles: Obstructions from the environment (walls, etc.). In a path planning context, one of the main goals of the algorithm is to compute a path avoiding all collision with these obstacles. In crowd

simulations, they are also to be navigated around. Since they are part of the environment, they generally do not move.

A scene: Container for the obstacles and agents. In both path planning methods and crowd simulation ones, these store the space partitioning structure (if any) (e.g.: kd-tree, quad-tree) and perform the space partitioning operations (finding which tree node to move an agent and moving it).

Along with the above elements, two types of algorithms have emerged, microscopic and macroscopic algorithms. A crowd simulation method is classified as microscopic if it considers the agents as individual entities. On the other hand, if the agents are merely part of a bigger body (such as a particle system), the algorithm is macroscopic.

Some things, however, are much less conventional such as the control flow of the algorithms. All methods iterate over the agents to move them and that is all they have in common. For this paper, the control flow of a "complete" crowd simulation is described using Menge's model [12]. Menge is a complete suite for crowd simulation development which divides the problem into 4 sub-problems which are abstracted so that anyone can implement solutions for them.

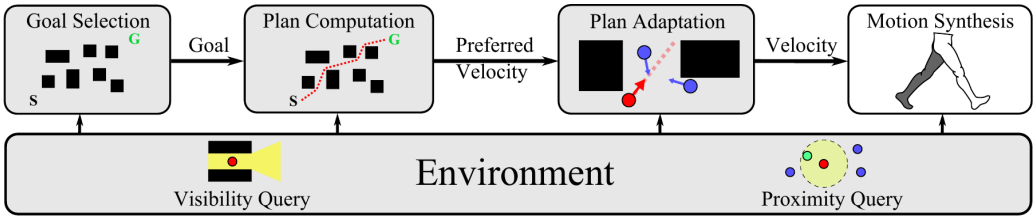

Figure 2.3: Menge splits the crowd simulation problem into sub-problem which can be dealt with separately. [12]

Goal selection: Agents must pick a goal if more than one are available (e.g.: closest exit). This step can be straightforward or complex depending on the scenario. For example, populating a train station or performing a formation may not be so simple. Solutions for this sub-problem include using finite state machines [1] and situation algebra [31].

Path planning: Agents construct a path around all of the static environment obstacles toward the selected goal [2]. Most path planning algorithms revolve around graph search where each node denote a valid move and the final node is the goal. A popular example of such algorithm is the A* graph search.

Path adaptation: Agents constantly steer and adjust their speed to avoid collisions with other agents (unforeseen conditions) during the simulation. In crowd algorithms, this sub-problem is generally where most of the agents' intelligence resides (some focus more on the path planning sub-problem).

Motion synthesis: Path, steering and speed adjustments are then converted to motion that can be visualised. This includes rendering the agents as well as animating them if needed (e.g.: human agents might have a walk animation).

Visual and proximity queries: The space partitioning method dictates how agents can access to each other and how fast this can be done. This is generally implemented as a grid or tree structure. Popular examples include quad-trees and kd-trees where the space is repeatedly divided until the nodes contain a set maximum number of agents.

The algorithm used in this paper is a path adaptation one. The agents only have one available goal at a time and the original path computed is a straight line to the goal. Motion synthesis is minimal and space partitioning is omitted to focus on the main algorithm.

2.2.1 Social Forces

The social forces model was introduced by Helbing and Molnár in 1995 [25]. The algorithm models crowd behaviour using a set of forces. It is a microscopic method where each agent is affected by forces due to the environment or other agents which contribute to the agent's acceleration. Computation of the acceleration at each time step is represented by the simple formula:

$$m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{\mathbf{v}_i^0 - \mathbf{v}_i}{\tau_i} + \sum_{j \neq i} \mathbf{f}_{ij} + \sum_W \mathbf{f}_{iW} \tag{2.1}$$

Where the LHS term is the agent's acceleration, the first term of the RHS represents a force pulling the agent toward its goal. The RHS's second term represents the "social forces", the effect of every other agent in the current one's neighbourhood on it. The last term sums the forces from the environment. This equation is executed for each agent and for each frame. The following descriptions of the computations are taken from the source code of PedSim [18] since they are the computations used in our implementations.

Computation for the desired force is quite straightforward, the difference between the goal's position and the agent's give a direction and the magnitude can be any value (we use the agent's maximum speed). The environment force (or obstacle force) is also straightforward but has an exponential component:

$$\mathbf{f}_{iW} = \frac{e^{-\frac{d_{iW}}{\sigma}} \mathbf{p}_{iW}}{d_{iW}}$$

Where d_{iW} denotes the distance between agent i and the nearest obstacle and $\mathbf{p}_{iW}$ is a unit vector representing the obstacle's direction from the agent (PedSim [18] only considers the closest obstacle while Helbing [25] sums all obstacles' forces). σ is a constant that scales the force and is defined by the developer. This force is computed once and only considers the closest obstacle.

Finally, the social force's computation between an agent i and a neighbour j is split into several part. The first part consists in calculating the difference in both agents' direction (or velocities to be more precise), $\mathbf{r}_{ij}$:

$$\mathbf{v}_{diff} = \mathbf{v}_i - \mathbf{v}_j \mathbf{r}_{ij} = \lambda \mathbf{v}_{diff} + \mathbf{p}_{ij}$$

Where λ, the lambda importance, is a constant used to scale $\mathbf{v}_{diff}$ and $\mathbf{r}_{ij}$ is normalised. θ can now be computed, it is the difference of angle from $\mathbf{p}_{ij}$ to $\mathbf{r}_{ij}$. If the difference is anticlockwise, θ is negative. The force itself is then computed in two parts, the velocity force and the angle force. Below is the former force's computation:

$$\mathbf{f}_{sv} = -e^{-\frac{d_{ij}}{B} - (n'B\theta)^2} \mathbf{r}_{ij}$$

Where n' is a user-defined constant, $B = \gamma a_{ij})$ is a scaling factor for the exponential where γ is also a user-defined constant and a_{ij} represents the length of $\mathbf{r}_{if}$ before it was normalised. B is largest when the agents point in opposite directions. The angle force is computed similarly to the velocity one except it is at right angle to it:

$$\mathbf{f}_{sa} = e^{-\frac{d_{ij}}{B} - (nB\theta)^2} (\mathbf{r}_{ij}(+/-)90\,\mathrm{deg})$$

Where the $(+/-)90\,\mathrm{deg}$ means that the vector points at 90 degrees from $\mathbf{r}_{ij}$ anticlockwise if *theta* is negative (anticlockwise to go from $\mathbf{p}_{ij}$ to $\mathbf{r}_{ij}$). The forces are then simply added together to make the social force that agent j exerts on agent i. This force is the method's core and can model some observed emergent behaviours of crowds such as following and grouping.

2.2.2 Lookahead Mental Layer

The social forces model describes a "physical" operation layer. Another aspect often present in crowd simulations is a "mental" layer. Such a layer describes some intelligence for the agents on top of their

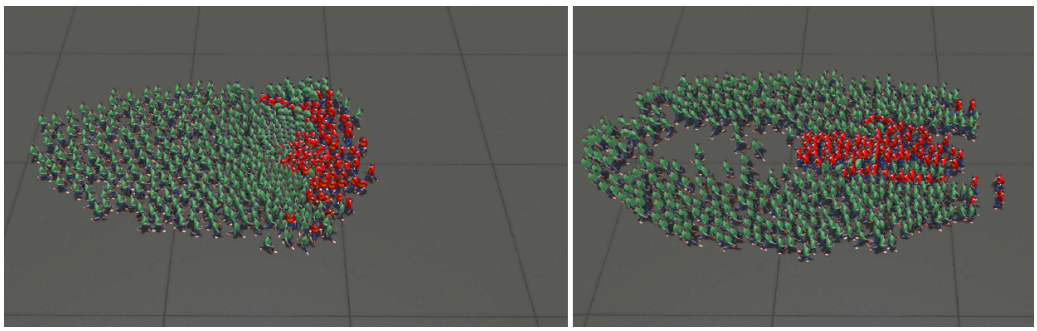

Figure 2.4: The lookahead mental model with two crowds colliding. The left image does not use the lookahead model. On the right image, the larger (green) crowd makes room for the smaller (red) crowd due to its density. [21]

desire to reach their goal. In this paper, we use the "lookahead" mental model [21] because it is simple to both understand and implement [20].

The model describes a long-range collision avoidance scheme where agents that are on the left and on the right of the agent are counted and a force is computed (in this case, because we are using social forces as main algorithm) which steers the agent away from the bigger density. This can be seen on figure 2.4.

2.3 Rendering

Rendering is the process of converting a 2-dimensional or 3-dimensional scene into an image to be displayed. The most popular rendering API is OpenGL which was already mentioned above as a compute API. OpenGL is developed by Khronos.

Since version 3.2 (released in 2009), OpenGL provides geometry shaders [4]. A geometry shader is a stage in the rendering pipeline that is executed in parallel to process geometry primitives to make them ready for drawing. The particularity of this shader stage is the ability to generate new geometry. Using this method, many objects can be rendered simultaneously, for example, agents in a crowd simulation.

2.3.1 Interoperability

In order to draw the agents in a simulation, OpenGL requires allocating a buffer in GPU memory containing all the agents' data. Then, that buffer needs to be updated every time the agents have moved. Since OpenCL also needs a buffer containing the agents' data, the information is sent to OpenCL for processing, then retrieved and sent to OpenGL.

There is an extension in OpenCL which allows access to the buffers allocated by OpenGL [23]. Using this extension, the data does not require to be moved, saving the time taken by the two transfers.

2.4 Related Research

Helmut Duregger has implemented the continuum crowds algorithm [34] in OpenCL for a master's thesis [14]. The continuum crowds model is a macroscopic algorithm which describes the scene using a set of fields. This method is more complex than social forces and, because of the fields, it can take advantage of image processing techniques and vector devices. At the time of writing, his implementation is the only one available using OpenCL.

2.4.1 Based on CUDA

Many papers use CUDA in their implementations which restricts them to NVIDIA GPUs. However, there are some advantages to using CUDA. One of them is being able to use C++ for writing the kernels. This allows more flexibility in designing algorithms. CUDA also provides a bit more performance than OpenCL

[15].

Chen et al. [8] developed a hybrid algorithm based on vector-fields. Their method is both microscopic and macroscopic. They focus on confrontation operation simulations involving large crowds.

Mróz and Was compare discrete and continuous methods. They assess the capabilities of GPGPU by comparing the social forces model (a continuous method) and a discrete algorithm based on Burstedde's cellular automata [5] called social distances.

Joselli et al. [28] implemented a crowd simulation which can take advantage of multiple GPUs. The algorithm used is a microscopic one based on boids and adjusted to simulate realistic crowds on GPU clusters.

Some other papers describe acceleration methods based on GPGPU to optimise only a certain aspect of crowd simulations. These papers only focus on a part of the problem.

Haciomeroglu et al. [24] present a GPU-assisted method where the simulation is split between the CPU and GPU. A clustering operation is performed on the GPU, then the cluster information is fed back to the CPU where a microscopic algorithm is executed.

Vigueras et al. [35] describe a GPU-accelerated collision avoidance method for crowd simulations. The authors identified that the biggest bottleneck in crowd simulations was the collision avoidance part so they propose a GPGPU method for it. The paper also compares the performance of their method on multi-core CPUs and on (many-core) GPUs.

3 Design and Implementation

In this report, the implementation of a simple crowd simulation library in OpenCL is described. The implementation is based on an existing CPU library and will extend it by adding a set of flexible renderers. The library may then be used as a base for crowd simulation experiments and can be integrated into an existing application (e.g.: in a game).

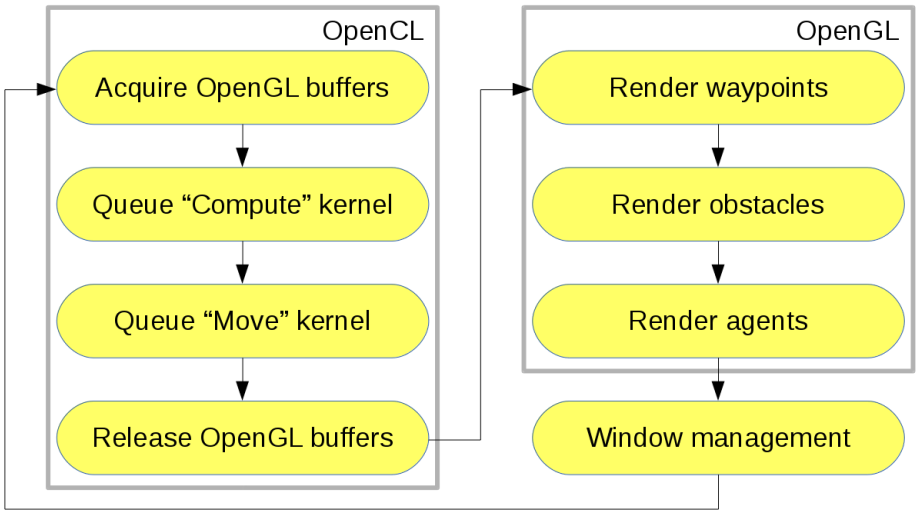

Figure 3.1: The control flow of our application. The OpenCL and OpenGL to optimise the buffers’ access. Window management is not part of the library and should be handled by the application.

3.1 PedSim

The original CPU implementation is named PedSim. It is a very lightweight and minimalistic crowd simulation library based on the social forces model discussed above. Being designed as a library, it can be used inside any kind of application. For example, PedSim could be integrated in video-games or films for simulating crowds in a single scene or it could be used as the core of an evacuation scenario testing application [19]. It has already been used as a simulation generator, generating motion files to be imported in a 3D animation application to produce crowd or war scenes [19].

The issue with PedSim is its restriction to a serial execution. The main path adaptation step is made up of two loop. The first one iterates through each agent and computes the forces affecting them. The second iteration moves the agents one by one. The first loop performs all the computation and is therefore a bottleneck. For that reason, it is the main focus of this paper.

3.2 OpenCL

Before being able to use OpenCL, it has to be initialised. The API is an open-source standard. This means that the Khronos Group does not provide the implementation but only the specifications. An API is provided by the hardware manufacturers and contain the headers and an Installable Client Driver (ICD) loader. Both are part of the standard and are largely the same [6]. The loader is then used to load an ICD which will contain the implementations of OpenCL functions. ICD's are generally part of the device driver and are readily available on computers.

The experiments in this paper were run on an HP laptop equipped with an Intel i7 CPU (quad core at 2.3GHz) with an integrated Intel HD 4600 GPU and a dedicated NVIDIA GTX850M GPU. There are, therefore two ICD's available on this computer. The choice of ICD is made at runtime where the first few OpenCL commands query the platforms available (command from the loader) and one is picked to create an OpenCL context (command from the ICD).

Once the OpenCL context is ready, a command queue is prepared for the devices. Commands can only be executed when sent to a queue and are handled by the device owning said queue (that could be a CPU, GPU, FPGA or any OpenCL-compatible device). The queue is required because the CPU and GPU are two separate chips running at two frequencies and synchronising them can takes time. Queueing the commands removes a need to wait.

3.2.1 Workgroups

Workitems to be executed on the GPU are cut into workgroups to take advantage of the GPU's compute units. Some architectures are optimised for certain sizes of workgroups. For example, NVIDIA advises developers to use multiples of 32 while AMD claims their GPUs provides better performance when using multiples of 64 (since it is a multiple of 32, compatibility is not a big issue).

As shown on figure 2.2, the data is first sent to the GPU, then an execution command is sent with two N-dimensional ranges representing the global and local space. These may be 1-, 2-, or 3-dimensional depending on the program but they both must have the same cardinality and the global space must be divisible by the local space. Examples of 2-dimensional problems include matrix manipulation and image processing.

The local range is synonymous to the workgroup size. Global space must be divisible by the local range. Additionally, some drivers may add some restrictions on the workgroup sizes. For example, the GPU used in the experiments further down cannot handle 1-dimensional workgroups of more than 1000 workitems.

For the purpose of this paper, the workgroups are 1-dimensional and sized using the size which yields maximum framerate. The effects of different workgroup sizes are discussed in the evaluation chapter below. The size varies depending on the agent count and the number of compute units so the optimal workgroup sizes chosen below may be different on another device.

3.3 OpenGL

If rendering is to be performed by OpenGL then a context must be obtained for it. Furthermore, this must be done before initialising the OpenCL context if data is to be shared between the two contexts. For simplicity reasons, the OpenGL context creation is to be created outside of the library presented here and will be fetched through the operating system when needed.

Sharing data between the two APIs means there is no need to send the agents' buffer to OpenCL and then recover it only to send it again to OpenGL. Since both APIs handle data on the Video RAM (VRAM, the GPU's memory), some implementations provide a zero-copy sharing of buffers. This means that no data is copied and both API's address the same space in memory.

3.4 Computing the Forces

In PedSim, the forces were computed inside a for loop containing a single method call on each agent independently. The mutual exclusivity of each computation made the loop an ideal candidate for a parallel implementation.

The first kernel computes the forces and combines them to find each agent's acceleration as per equation 2.1 (the "Compute" kernel on figure 3.1). Since the agents are being processed in parallel from the same buffer, the modification from one instance of the kernel must not affect the others. For that reason, in the kernel, all attributes of the agent *struct* are considered read-only except the acceleration attribute which is regarded as write-only.

3.4.1 Desired Force

The desired force is a simple force driving the agent toward its goal. The force points in the direction of the goal and is always of the same magnitude (which depends on the agent's maximum speed).

3.4.2 Social Forces

There is one social force acting on the current agent from each other agent in its neighbourhood. The force is inversely proportional to the distance between the agent and its neighbour. For that reason, a neighbourhood is used to prevent needlessly computing forces that will not significantly affect the agent. PedSim originally uses a quad-tree to obtain the neighbours from a square neighbourhood, since no space partitioning is implemented here, the neighbourhood is defined as a radius around the agent.

The computation is designed as a loop iterating through all agents. The distance between the current agent and the iteration's agent is computed and the agents outside a neighbourhood radius are discarded. A social force is computed for the agents that remained (as described in the background section) and all forces (vectors) are added to a sum (vector) (See Appendix A.1 for the kernel code).

3.4.3 Lookahead Force

This force is computed in a straightforward way. Like the social forces loop, the first operation is to discard all agents outside some neighbourhood radius (which is much larger than the one for social force computation). The second operation is to discard all agents which are not moving in the opposite direction (angle difference ¡ 2.5 radians). A counter is then incremented for each agent on the left and is decremented for each agent on the right (within 0.3 radians to simulate a field of view). Finally, if the counter is negative (more agents on the right), a force to the left is computed, to the right otherwise (See Appendix A.2 for the kernel code).

3.4.4 Obstacle Force

There is only one obstacle force in this implementation. A loop iterates through all available obstacles and picks the closest one to the agent. The force exerted by the obstacle is inversely proportional to the distance between it and the agent.

3.5 Moving the Agents

In PedSim, the agents were moved using a loop containing a single method call per agent (similarly to the forces computation). Therefore a second OpenCL kernel was developed (the "Move" kernel in figure 3.1). The operations performed by this loop was greatly simplified from the PedSim counterpart. Originally, an agent's acceleration is computed from the forces previously obtained (for simplicity, the acceleration is equal to the resultant force), then the velocity is adjusted using that acceleration (i.e.: the agent turns), the position is changed (i.e.: the agent moves) and finally the agent is moved to the correct node in the quad-tree. Since no space partitioning in our implementation, the kernel only performs the first three operations.

3.6 Rendering

The rendering is implemented in OpenGL 3.3. As will be discussed later, geometry shaders are an important part of the renderer and were introduced in version 3.2 making it the minimum requirement but 3.3 was used for compatibility with Mac OSX.

A separate renderer was developed for each object type (i.e.: agents, obstacle, waypoints) but their structure is similar. The renderer first initialise a shader program (different shaders for each renderer),

then, when the scene is set up, the renderer is sent the list of objects it needs to render. That list is directly sent to the GPU and these are the buffers which are shared with OpenCL. Finally, during the simulation, each renderer renders its objects in parallel as points. This allows better flexibility for the geometry shaders.

3.6.1 Geometry Shaders

Geometry shaders allow the generation, dismissal and manipulation of geometry. All renderers mentioned above make use of geometry shaders. In the OpenGL rendering pipeline, the data is first sent to the vertex shader where each vertex is processed separately, then the data is passed to the geometry shader as primitives (points, lines, triangles) and, finally, the data is rasterised and processed in the fragment shader as fragments (pixels).

As mentioned earlier, the renderers draw their list of objects as points. This allows the geometry shader to receive each object as a point primitive for which any kind of geometry can be generated.

The geometry shader for agents generates a triangle within the bounds of a circle with the same radius as the agent's and pointing in the same direction as the velocity vector of the agent. For obstacles, since they are defined as lines, the shader simply generates a line from the start point of the obstacle to its end point. For waypoints, the shader generates a circle with the same radius as the waypoint.

3.7 Gathering Data

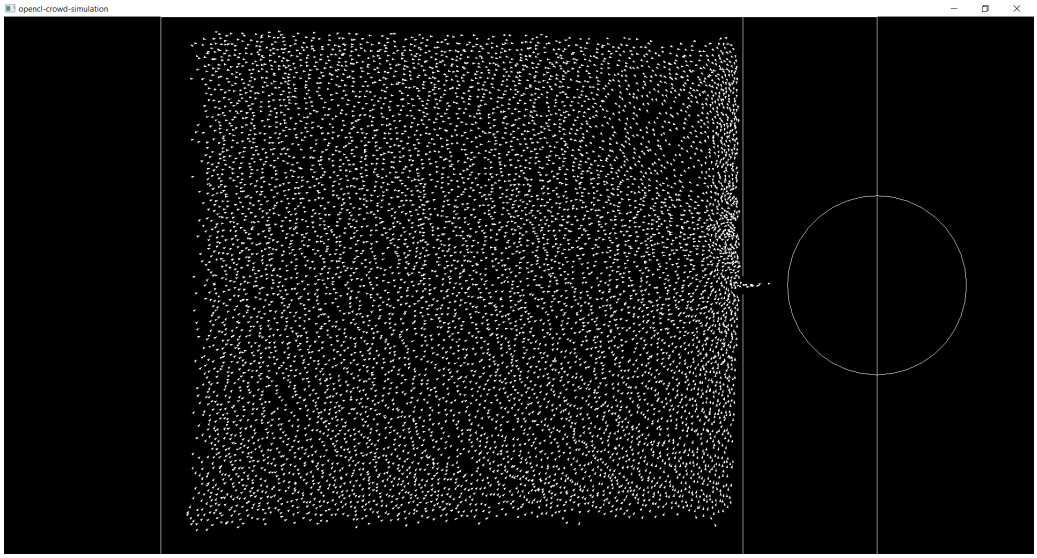

Figure 3.2: The experiment' program, used to record the data (showing the start of an 8,000 agent simulation). Rendering was kept simple, each agent is a triangle directed according to the agent's velocity vector, obstacles are lines and waypoints are circles.

Figure 3.2 shows the application used to gather the data analysed in the next section. This is a very simple application which opens a window, runs several simulations with different parameters, records the required data and closes. The rendering was kept simple which should make its impact on the performance negligible.

The first few tests are run without OpenCL, as a baseline for the comparisons. These tests only have the agent count as variable. The program runs ten simulations (the ten bars on the column charts below) for ten seconds each and outputs the average framerate in the console. Framerates are calculated by counting the frames displayed within the ten seconds and dividing by ten.

The other tests were run with OpenCL and therefore were run in parallel. As such, there is an optimal workgroup size which depends on the number of compute units, the number of SIMD units in a compute

unit and the total number of workitems. The workgroup size is therefore also a variable. For each agent count, a simulation is run as many time as there are factors dividing that agent count and only the one which resulted in the highest framerate is recorded (i.e.: the optimal workgroup size).

In the case of a CPU, there is no limit to the size of a workgroup (as long as the device does not run out of resources). But, in the case of GPUs, there is a maximum number of workitems that a compute unit can process. In the case of our GTX 850M (and I believe most NVIDIA GPUs), the limit is 1024 so the workgroup sizes were restricted to that when running. Each batch of simulations took approximately one hour.

4 Results and Evaluation

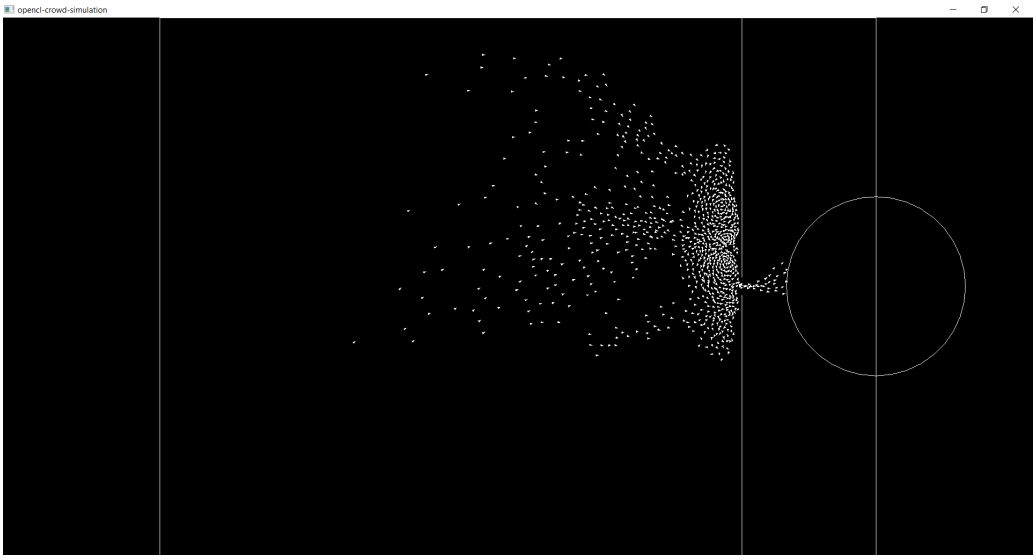

Figure 4.1: Example of our program towards the end of a simulation. It can be seen that agents accumulate against the wall and move in some swirling formation which is an emergent behaviour resulting from the social force model.

Experiments were run to assess the performance gain of our implementation. We only look at the simulation's speed, represented as the application's framerate in this paper (behavioural performance is not expected to change). The experiments were executed on a laptop equipped with an Intel quad-core CPU running at 2.3GHz [9] and an NVIDIA GPU with 640 processing cores running at 936 MHz [11]. The results therefore reflect real-time performance on middle- to high-end laptops.

As previously mentioned, the original implementation of PedSim had to be modified in order to make it "parallel-friendly". Changes include removing the space partitioning, storing all items of the same type in large arrays and removing all pointers in favour of array indices. Some of these changes have slowed the algorithm and some have accelerated it. Figure 4.2 is the comparison between PedSim and our implementation.

Note that there is a ceiling at approximately 180 frames per seconds either imposed by the operating system, GPU or display. It can be clearly seen on figure 4.3. Framerate above 170 are therefore considered as equal for these experiments.

The performance gain seen on figure 4.2 is mainly due to the change from double-precision floating-point numbers to single-precision. Some functions were also changed for more efficient ones (e.g.: $atan2()$ to $atan2f()$, $sqrt()$ to $sqrtf()$). The expectation was a generally slower due to the removal of space partitioning but the halving of data transferred outweighed the loss.

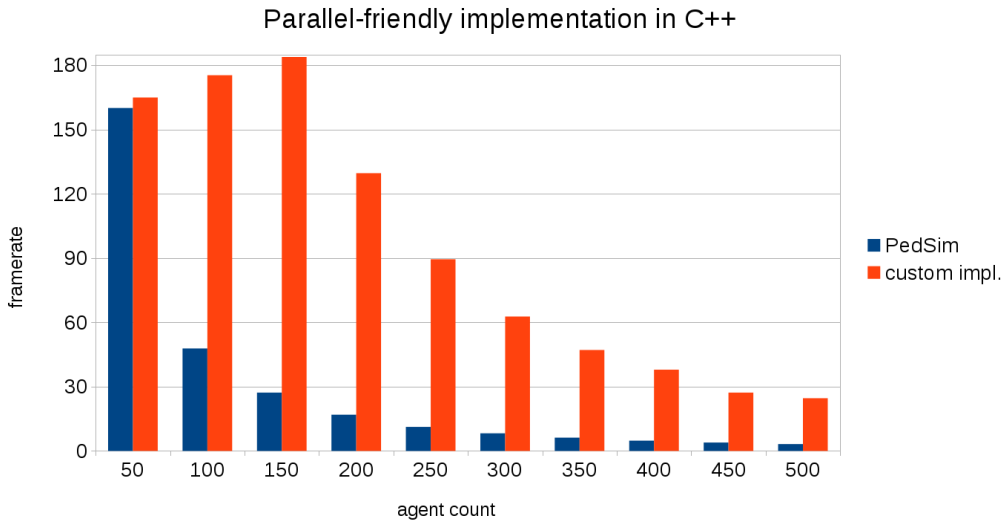

Figure 4.2: C++ variation of PedSim that is parallel-friendly

4.1 Running in OpenCL

Our custom implementation was specifically designed to be ported as OpenCL kernels with minimal changes. The code was converted and some minor OpenCL-specific optimisations were added (e.g.: $exp()$ were replaced with $native_exp()$ and $sqrtf()$ changed into $native_sqrt()$). The kernel was run in serial on a single CPU thread. This was done by bundling all workitems into one workgroup, a CPU thread is a compute unit and so can only process a single workgroup. Figure 4.3 shows the performance gain from merely running the algorithm in OpenCL.

The gain shown on figure 4.3 is due to low-level optimisations performed by OpenCL and the direct access to the CPU's processing pipeline and extensions (e.g.: here the Streaming SIMD Extensions (SSE4.2) or Advanced Vector Extensions (AVX 2.0) [9]) of the chip (as opposed to through the operating system).

4.2 Gain from Parallel

The next experiment looks into taking advantage of the 8 threads (OpenCL compute units) provided by the CPU. In cases where a GPU is not available, running a kernel in parallel on CPU can provide a performance advantage. Figure 4.4 below shows the difference between running the OpenCL kernel in serial and parallel. Note that, for this experiment and the following others, an optimal workgroup size was selected as explained in the design section.

The optimal workgroup sizes were generally low (mostly 2, 5 and 8) which means that there are not many processing units within a CPU thread (likely two but cannot verify). This still provides a significant acceleration over using only one CPU thread.

4.3 Gain from GPGPU

The purpose of this experiment is to evaluate the gain obtained from a GPU execution. Figure 4.5 below graph shows the comparison between running the OpenCL kernel on CPU and GPU. Note that there is no data sharing performed here, the only difference here is the device being run on, the buffers are retrieved in CPU memory after processing only to be transferred again for rendering in OpenGL.

The gain from using a GPU is obvious. In this case, the workgroup sizes that yielded best framerates were high compared to those of the CPU case. This shows that the GPU's compute units contain more threads (SIMD units) than the CPU.

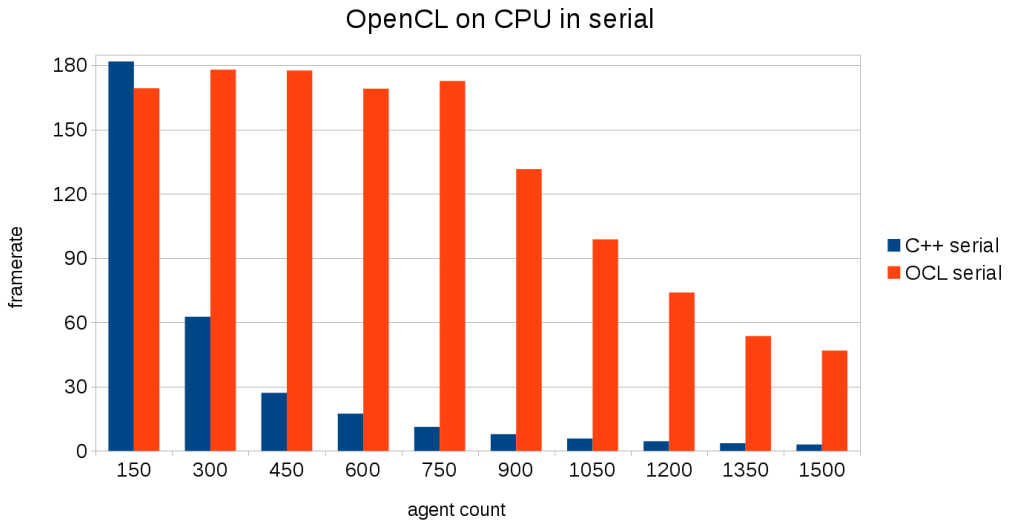

Figure 4.3: OpenCL implementation running in serial on a single CPU thread

From figure 4.5, it may be possible to infer a relationship between the GPU's structure 2.1 and the workgroup sizes. There are 128 SIMD units in a streaming multiprocessor so ideal workgroup sizes are (theoretically) multiples of 128. For example, a multiple of 125 workitems minimises the unused SIMD units to 3 per compute unit.

4.3.1 Data Sharing

An OpenCL extension allows sharing memory with OpenGL (e.g.: buffers, textures) [23]. Zero-copy sharing is used which allows OpenCL to access the OpenGL buffers and textures without moving any information. This extension is used which removes the need to retrieve the data from OpenCL and send it to OpenGL. The difference is shown on figure 4.6.

Since the waypoints and obstacles do not change, only the agents buffer had to be transferred repeatedly before. This experiment therefore shows the gain from not having to transfer the buffer of agents twice for every frame between GPU and CPU (e.g.: for 10,000 agents of 40 bytes, a 400kB buffer is transferred twice or 800kB is exchanged every frame). As figure 4.6 shows, there is little benefit gained from this.

4.4 Workgroup Sizes

It was shown that workgroup size affects the execution's performance. There is a sweet spot of workgroup sizes where the performance is maximal. This phenomenon is due to the execution pipeline of OpenCL, workgroups are processed in parallel by compute units and workitems within workgroups are processed in parallel by SIMD units (generally). Figure 4.7 shows how workgroup sizes affect a simulation of 5,000 agents.

As figure 4.7 shows, there is a "sweet spot" when it comes to workgroup sizes. For a simulation of 5,000 agents on a GPU with five streaming multiprocessors, the optimal workgroup size is 125. This means that each SM processes 125 agents simultaneously and there are 80 workgroups queued across 5 SMs. The (theoretical) worst case scenario makes each compute unit process 16 workgroups in series. The significant drop from grouping 200 agents can be explained in that each compute unit contains 128 SIMD units, the workgroup has to be processed in two batches (worst case).

Knowing how workgroup sizes affect the performance of computations, a conclusion can be drawn on how an optimal workgroup size can be chosen. The "sweet spot" is the maximum factor of the total computation space (e.g.: here 5,000 agents) which does not exceed the number of SIMD units in a compute

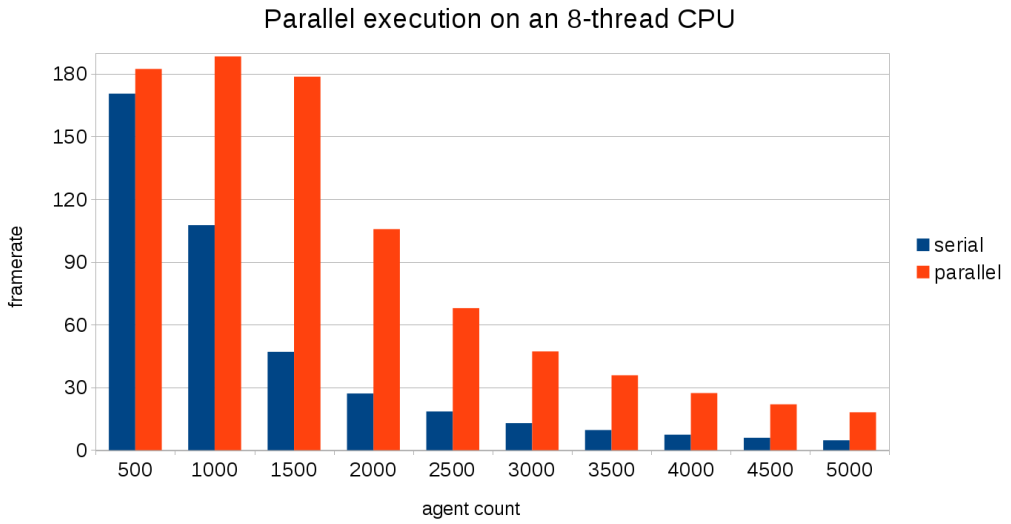

Figure 4.4: OpenCL implementation running in parallel on 8 CPU threads. The optimal workgroup sizes for each agent count respectively were 5, 125, 5, 8, 2, 2, 25, 8, 3 and 2

unit (e.g.: here 128). Although logically sound, this theory does not hold for the experiments shown on figures 4.5 and 4.6.

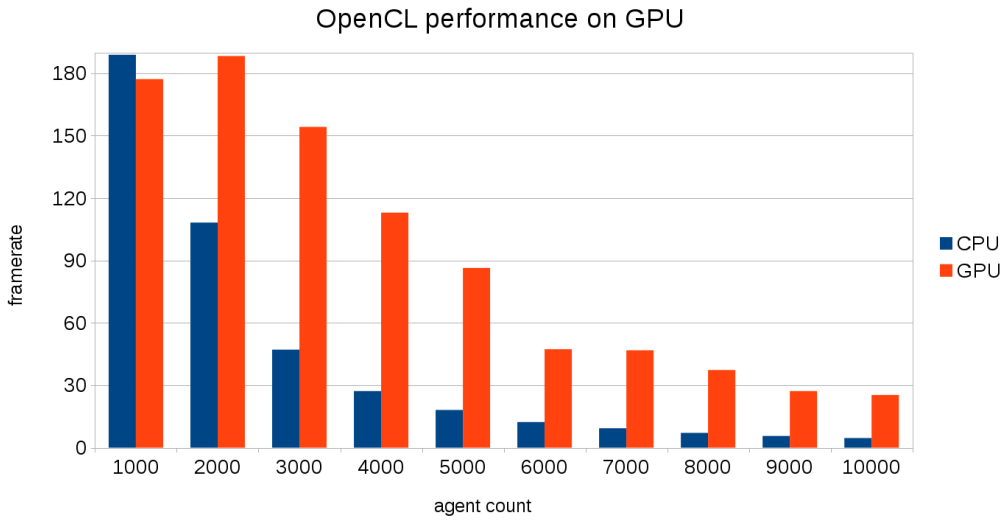

Figure 4.5: OpenCL implementation running on GPU. The workgroup sizes for the GPU execution were 200, 40, 300, 125, 125, 48, 350, 64, 60 and 125

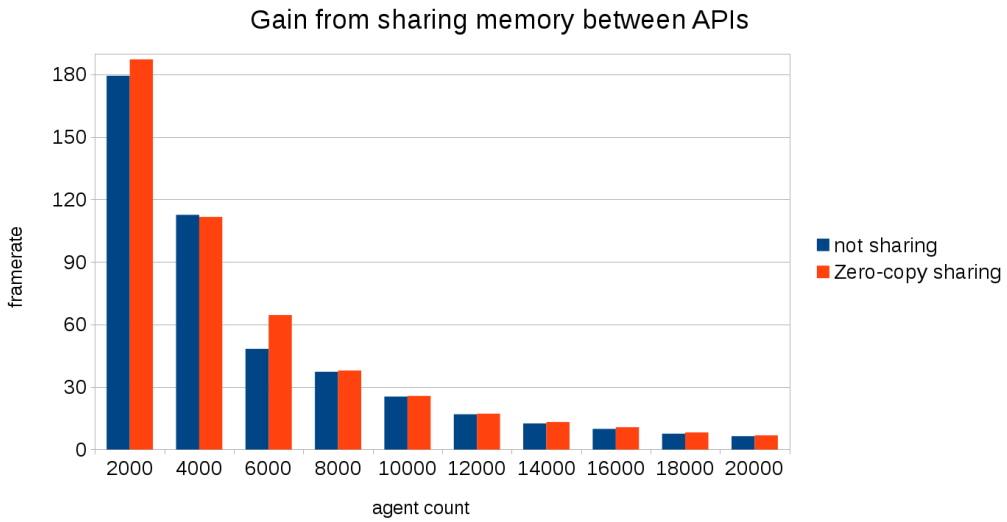

Figure 4.6: Memory sharing between OpenGL and OpenCL provide marginally better performance

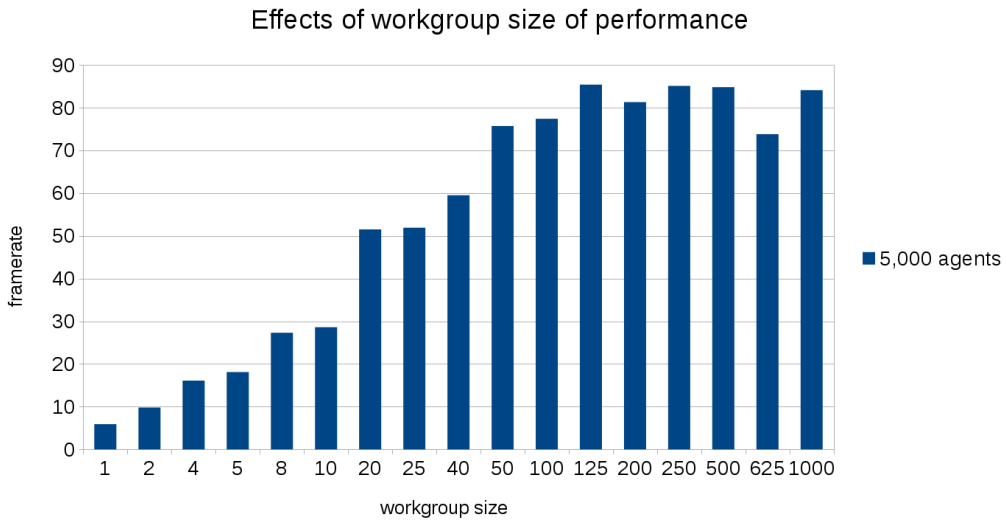

Figure 4.7: There is an optimal workgroup size. Both lower and higher sizes will provide lower performances. 5,000 agents (workitems) were split every way they could be (the number 5,000 has 17 factors below 1,024).

5 Discussion and Future Work

Our library has some changes made from the original PedSim code. One such change was the addition of renderers. Rendering is not performed in the original PedSim because the library might not be appropriate for real-time use. The examples showcasing PedSim use the library to merely output the simulation as a file which is then loaded in a 3D animation application for presenting.

5.1 Limitations

Many aspects of the developed library go against prior expectations. The most important such point is that the quad-tree element of the original PedSim could not be ported to OpenCL, which prevents our library to be a 1:1 port of PedSim on GPU. A quad-tree implementation in the C language requires the manual allocation of memory to grow the tree when needed. Since OpenCL kernels are based on the C language, a quad-tree implementation would have the same requirements as in C, however, manual memory allocation is not allowed within an OpenCL kernel (it is in CUDA on some devices). In PedSim, the quad-tree is only used to accelerate the neighbourhoods' filtering but no collision detection is implemented. This means that our library does not lack some functionality but rather merely an optimisation.

5.1.1 Limitations of the Model

The social forces model itself presents a few limitations in both speed and behaviour of the agents. Improvements discussed in this section are changes that would need to be applied to both our code and PedSim itself.

The main speed limitation comes from the computations being quite naive. As such, Some complex functions such as e^x and tan^{-1} significantly increase the computation time. These could be replaced by (faster) approximations.

In terms behavioural realism, the social forces model also has a few limitations. It is only a path adaptation algorithm and as such does not compute a "best" path to the goal but merely a straight path until an obstacle or other agent is met. The current implementation would get stuck in a dead-end and would only solve a maze if enough agents are simulated which would "push" each other and eventually reach the exit.

Another missing element is a collision detection and resolution method. This would prevent the agents from being on the same position as another agent or even to be inside a neighbour's radius. This cannot be implemented yet as collision detection requires a space partitioning method to be usable in real-time, some even require it [22].

5.2 Future Research

As mentioned above, a space partitioning method is the most important element that could be added. A tree-based method could be implemented using preallocated nodes. The maximum size that can be reached is available. Additionally, the maximum number of agents in each node should be large enough to never be reached and the space for those agents (pointers or indices) should be allocated. This may require a large amount of memory depending on the scene.

Some work about space partitioning on the GPU include a grid-based approach from Wang et al. [36] and a bi-dimensional data structure from Passos et al. [30] which managed to boost a boids simulation from 15,000 to a million at interactive framerate.

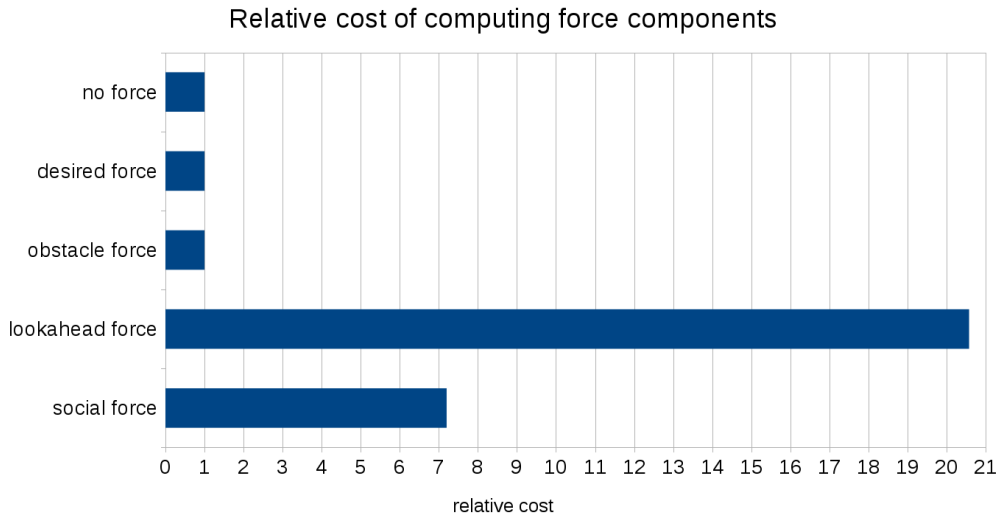

Figure 5.1: The relative computation time of each force. 1 means the force takes no time to compute.

Since space partitioning is required for it, a collision detection and resolution method would be a logical next step. Some algorithms are implemented in CUDA, studied and compared by Scott Le Grand in the GPU Gems (a NVIDIA book about CUDA) [22]. Figure 5.2 shows an algorithm sorting the objects according to their position. Such a method stores the agents in a sorted list which would imply a time complexity of $O(\log N)$ for moving an agent.

To make the simulation more realistic and complete, a path planning algorithm should be added. The agents would then follow a path as opposed to travelling in a straight line to their goal until a obstacle is met. The current implementation may only solve a maze if there are enough agents. The agents "repel" each other until the whole maze is visited and some agents eventually reach the exit.

This library also handles rendering the simulation. This part can be improved by adding texturing, lighting, animations or other improvements. The geometry shaders used could be extended to receive a uniform buffer filled with a 3D mesh to be rendered. Animations could also be added even though mesh skinning requires a vertex shader. In our implementation, the geometry generates an entire mesh one primitive at a time so could be used like a vertex shader. The only issue with this idea is that all vertices will be processed (generated) in series as opposed to in parallel in vertex shaders.

In its current state, the rendering uses geometry shaders to render the agents. This provides significant flexibility. Each agent is rendered separately and could be represented with different models, different poses (in the case of animated models) or different sizes. An issue arises, however, when the model become more complex. The generation is done procedurally for a single agent. This issue could be addressed using instanced rendering [7]. The OpenGL feature allows to render a large number of copies of the same model in parallel with all the variability already implemented (figure 5.3 shows one such example).

Vulkan is a new API for GPU computing and rendering which was released in February this year. The already has a very inclusive list of supported hardware and generally provides better performance than OpenGL for the same features. There are not many examples of Vulkan used for GPGPU yet but it is possible as it provides compute shaders. Furthermore, Vulkan's shaders are in an intermediate language called SPIR-V which can be obtained from OpenGL shaders or OpenCL kernels that are compiled either before compile time or at runtime. This means that the OpenCL kernels developed here could be re-used with minimal changes.

diag-space-partitioning.png diag-space-partitioning.pdf

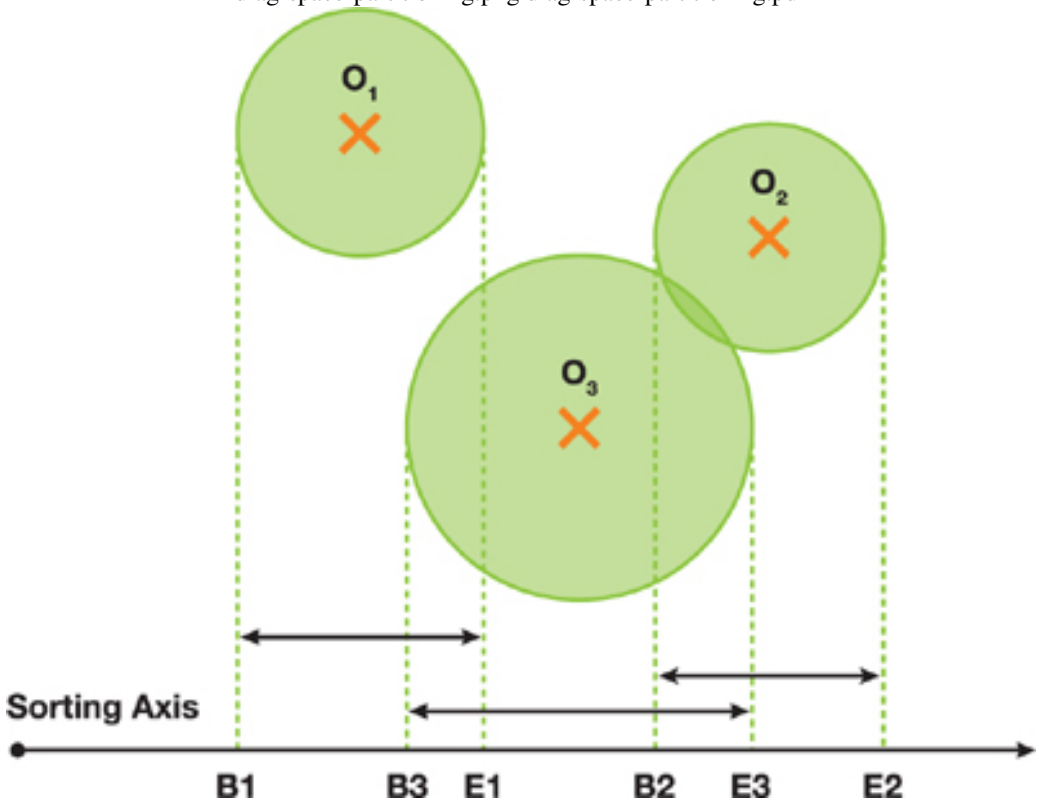

Figure 5.2: Collision detection in 2 dimensions. The broad-phase collision detection is reduced to an $O(n)$ (linear) search. [22]

Figure 5.3: Instancing is used here to render several models of grass and soldiers. Notice that the soldiers are not constrained to have the same pose. [7]

6 Conclusions

In this paper, we discuss the tools for crowd simulation and highlight that there is no open-source tool available taking advantage of a parallel implementation. This justifies the goal of this experiment to develop a tool revolving around a parallel implementation of a crowd simulation.

The tool takes form as a library due to the lack of reusable crowd simulations available. This library is essentially a parallel port of PedSim, a lightweight crowd simulation library that can be integrated in any program. PedSim was chosen for its lightweight nature and flexibility (anyone could modify any aspect of the simulation they wish without much hassle).

After investigating the recent interest toward GPUs for general processing as well as the APIs designed for it, some of the GPGPU platforms are briefly mentioned and compared, leading to the choice of OpenCL for the development of our simulation. This library was chosen for its popularity, hardware support and portability.

The GPU implementation is compared to the original PedSim and displays a large improvement but not as much as expected. This is due to some limitations in the model such as the use of trigonometry functions as well as some restrictions from OpenCL like the inability to implement a tree-based space partitioning method.

The library can be used in any program, similarly to PedSim but unlike its original source, requires OpenCL support. Another difference with PedSim are the renderers which were added and are required due to the use of OpenCL's OpenGL sharing extension. The data is sent once to the GPU and is shared between the two libraries. This greatly reduces the need to transfer data between the CPU and GPU and improves the performance.

Looking back at the original goals of the project, all objectives were met. The usability of OpenCL was assessed and studied. There was no aspect of a path adaptation algorithm that could have been separately GPU-accelerated so the entire algorithm was re-written as an OpenCL kernel. The library that was designed is based of PedSim and allows interactive framerates for the social force model for up to tens of thousands of agents (as opposed to a few hundreds in PedSim).

The library is usable but includes several limitations. The largest limitation is the lack of collision detection and resolution. Several agents can be located at the same position or some can enter their neighbour's radius (one is inside another). The integration of one such method requires the implementation of a space partitioning algorithm. This presents a few hurdles because manual memory allocation is not possible in OpenCL.

Other further research that could be carried out include a GPU implementation of a path planning algorithm, an adjustment to the social forces to remove the slow functions or improvements to the renderers such as lighting, textures or animations. According to the data currently available on the subject, it seems that a port on Vulkan could show better performances as well as more flexibility.

Bibliography

[1] Stefania Bandini, Mizar Luca Federici, Sara Manzoni, and Giuseppe Vizzari. Towards a methodology for situated cellular agent based crowd simulations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3963 LNAI, pages 203–220, October 2005.

[2] Jérôme Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, December 1991.

[3] Basefount.

[4] Pat Brown and Barthold Lichtenbelt. Arb_geometry_shader4, January 2011. Available: `https://www.opengl.org/registry/specs/ARB/geometry_shader4.txt` Accessed: 2016-10-10.

[5] Carsten Burstedde, Kai Klauck, Andreas Schadschneider, and Johannes Zittartz. Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3-4):507–525, January 2001.

[6] Christopher Cameron, Benedict Gaster, Michael Houston, John Kessenich, Christopher Lamb, Laurent Morichetti, Aftab Munshi, and Ofer Rosenberg. cl_khr_icd, March 2010. Available: `https://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt` Accessed: 2016-09-28.

[7] Francesco Carucci. Chapter 3. inside geometry instancing, April 2005. Available: `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter03.html` Accessed: 2016-10-10.

[8] Dan Chen, Lizhe Wang, Mingwei Tian, Jian Tian, Shuaiting Wang, Congcong Bian, and Xiaoli Li. Massively parallel modelling & simulation of large crowd with gpgpu. *Journal of Supercomputing*, 63(3):675–690, March 2013.

[9] Intel Corporation. Intel©core^{TM}i7-4712hq processor, 2014. Available: `http://ark.intel.com/products/78932/Intel-Core-i7-4712HQ-Processor-6M-Cache-up-to-3_30-GHz` Accessed: 2016-09-27.

[10] Intel Corporation. Using data parallelism, July 2016. Available: `https://software.intel.com/en-us/node/540422` Accessed: 2016-10-11.

[11] NVIDIA Corporation. Geforce gtx 850m specifications, 2014. Available: `http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-850m/specifications` Accessed: 2016-09-27.

[12] Sean Curtis, Andrew Best, and Dinesh Manocha. Menge: A modular framework for simulating crowd movement. "pre-published" paper available at: `http://gamma.cs.unc.edu/Menge/files/mengeCDMain.pdf` Accessed: 2016-05-25, 2013.

[13] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, and Gregory Peterson. From cuda to opencl: Towards a performance-portable solution for multi-plaform gpu programming. *Parallel Computing*, 38:391–407, Octobre 2011.

[14] Helmut Duregger. Simulation of large and dense crowds on the gpu using opencl. Master's thesis, University of Innsbruck, October 2011.

[15] Jiambin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Proceedings of the International Conference on Parallel Processing*, pages 216–225, September 2011.

[16] Dirk Fuchs. Brainiac, October 2014. Available: https://github.com/difu/Brainiac Accessed: 2016-10-09.

[17] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, opencl 1.2 edition, 2013.

[18] Christian Gloor. Pedsim, February 2010. Available: http://pedsim.silmaril.org/ Accessed: 2016-10-09.

[19] Christian Gloor. Pedsim examples, February 2010. Available: http://pedsim.silmaril.org/examples/ Accessed: 2016-09-28.

[20] Christian Gloor. Pedsim documentation, February 2010. Available: http://pedsim.silmaril.org/documentation/ Accessed: 2016-10-02.

[21] Abhinav Golas, Rahul Narain, and Ming Lin. Hybrid long-range collision avoidance for crowd simulation. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 29–36, March 2013.

[22] Scott Le Grand. Chapter 32. broad-phase collision detection with cuda, July 2009. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html Accessed: 2016-10-03.

[23] The OpenCL Working Group. cl_khr_gl_sharing, March 2010. Available: https://www.khronos.org/registry/cl/extensions/khr/cl_khr_gl_sharing.txt Accessed: 2016-10-09.

[24] Murat Haciomeroglu, Oner Barut, Cumhur Y. Ozcan, and Hayri Sever. A gpu-assisted hybrid model for real-time crowd simulations. *Computers & Graphics*, 37(7):862–872, November 2013.

[25] Dirk Helbing and Péter Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51 (5):4282–4286, 1995.

[26] Folkert Huizinga. C.o.r.p.s.e., July 2011. Available: https://github.com/Error323/CORPSE Accessed: 2016-10-09.

[27] N. Johnson and W. Feinberg. A computer simulation of the emergence of consensus in crowds. *American Sociological Review*, 42:505–521, 1977.

[28] Mark Joselli, José Ricardo Da Silva Junior, and Esteban Clua. An architecture for real time crowd simulation using multiple gpus. In *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, pages 1–10, December 2014.

[29] Hassan Mujtaba. Nvidia geforce gtx 860m features maxwell gm107 core and 45w tdp benchmarks unveiled, March 2013. Available: http://wccftech.com/nvidia-geforce-gtx-860m-features-maxwell-gm107-core-45w-tdp-benchmarks-unveiled/ Accessed: 2016-10-03.

[30] Erick Baptista Passos, Mark Joselli, Marcelo Zamith, Esteban Walter Gonzalez Clua, Anselmo Montenegro, Aura Conci, and Bruno Feijo. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. *Computers in Entertainment*, 7(4), December 2009.

[31] Wei Shao and Demetri Terzopoulos. Autonomous pedestrians. In *Computer Animation, Conference Proceedings*, pages 19–28, July 2005.

[32] Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn D. Reinman. An open framework for developing, evaluating, and sharing steering algorithms. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5884 LNCS, pages 158–169, November 2009.

[33] Borut Robic Theo Ungerer and Jurij Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.

[34] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. In *ACM SIGGRAPH 2006 Papers, SIGGRAPH '06*, pages 1160–1168, July-August 2006.

[35] Guillermo Vigueras, Juan Manuel Ordu na, Miguel Angel Ngel Lozano, José María Cecilia, and José Manuel Rodriguez García. Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *International Journal of High Performance Computing Applications*, 28(1):33–49, February 2014.

[36] Yongwei Wang, Michael Lees, and Wentong Cai. Grid-based partitioning for large-scale distributed agent-based crowd simulation. In *Winter Simulation Conference*, December 2012.

A Appendices

A.1 Computing Social Forces

```
float2 get_social_force(const __global agent_t* agent, int num_agents, const __global agent_t* agents)
{
    float2 force = 0.0f;

    for (int i = 0; i < num_agents; i++) {
        const __global agent_t* other = &agents[i];

        if (!other->isActive || other == agent) continue;


        float2 diff = other->position - agent->position;
        float squared_distance = SQUARED_LENGTH(diff);

        if (squared_distance > SQUARED(SOCIAL_RADIUS)) continue;


        float distance = native_sqrt(squared_distance);
        float2 diff_direction = diff / distance;

        float2 diff_velocity = agent->velocity - other->velocity;

        float2 interaction = LAMBDA_IMPORTANCE * diff_velocity + diff_direction;
        float interaction_length = LENGTH(interaction);
        float2 interaction_direction = interaction / interaction_length;

        float theta = atan2(diff_direction.y, diff_direction.x) - atan2(interaction_direction.y, interaction_direction.x);

        if (theta >  PI) theta -= 2.0f * PI;
        if (theta < -PI) theta += 2.0f * PI;

        float theta_sign = (theta < 0.0f) ? -1.0f : 1.0f;

        float B = GAMMA * interaction_length;
        float force_velocity_amount = -native_exp(-distance / B - SQUARED(N_PRIME * B * theta));
        float force_angle_amount = -theta_sign * native_exp(-distance / B - SQUARED(N * B * theta));

        float2 force_velocity = force_velocity_amount * interaction_direction;
        float2 force_angle = force_angle_amount * (float2)(-interaction_direction.y, interaction_direction.x);

        force += force_velocity + force_angle;
    }

    return force;
}
```

Figure A.1: Piece of kernel code that computes the social forces for an agent. First, the force used for the sum is set to zero. In the loop, the first operation is to discard the agent if it is the current one, then if it is too far. A social force is then computed for each agent (according to formulas given in the “Background” section) and added to the sum.

A.2 Computing Lookahead Forces

```
float2 get_look_ahead_force(const __global agent_t* agent, float2 direction, int num_agents, const __global agent_t* agents)
{
    int count = 0;

    for (int i = 0; i < num_agents; i++) {
        const __global agent_t* other = &agents[i];

        if (!other->isActive || other == agent) continue;


        float2 diff = other->position - agent->position;

        if (SQUARED_LENGTH(diff) > SQUARED(NEIGHBORHOOD_RADIUS)) continue;


        float direction_angle = atan2(direction.y, direction.x);
        float vv = direction_angle - atan2(other->velocity.y, other->velocity.x);

        if (vv >  PI) vv -= 2.0f * PI;
        if (vv < -PI) vv += 2.0f * PI;

        if (fabs(vv) > 2.5f) {
            float s = atan2(diff.y, diff.x) - direction_angle;

            if (s >  PI) s -= 2.0f * PI;
            if (s < -PI) s += 2.0f * PI;

            if ((-0.3f < s) && (s < 0.0f)) {
                count--;
            }
            else if ((0.0f < s) && (s < 0.3f)) {
                count++;
            }
        }
    }

    float2 force;

    if (count < 0) {
        force.x =  0.5f * direction.y;
        force.y = -0.5f * direction.x;
    }
    else {
        force.x = -0.5f * direction.y;
        force.y =  0.5f * direction.x;
    }

    return force;
}
```

Figure A.2: The part of our "Compute" kernel which computes the lookahead forces. Similarly to the social forces, the first operations of the loop is to discard the agent if it is the current one or if it too far. The loop then filters the agents which are moving in opposing direction to the current agent ($fabs(vv) > 2.5f$ in radians) and simply weigh the agent count on the left and right. After the loop, the force direction is decided from the agents counted.