SIMULATION OF LARGE AND DENSE CROWDS ON THE GPU USING OPENCL

MASTER THESIS

BY

Helmut Duregger

Institute of Computer Science
Faculty of Mathematics, Computer Science and Physics
University of Innsbruck

SUPERVISOR

Assoz.-Prof. Dr. Radu Prodan

October 18, 2011

I dedicate this to my mum and dad. Without their continuous patience and support nothing of this would have been possible, and I would not be the free-thinking person I am today. Dad started my interest in nature and technology, and taught me to be curious about how the world functions, to always question everything. Mum taught me to be nice to people, how never-ending love can make you stronger, and if you believe in something you can achieve your dreams. And I think they also gave me a good portion of humor on the way.

I also dedicate this to my brother and sister. My sister is so full of energy and life, and she never hesitates to share it with others. This put me back on my feet several times, and taught me that you have to share what you enjoy. My brother fueled my interest for the sciences, but also for the arts. Music has become my passion, and so has the movies and visual arts.

I dedicate this to all my friends, who accompanied me through the many years, sharing passion and dedication. They accept my flaws and quirks, and would never let me down.

Without you, I would not have made it.

Thank you!

ABSTRACT

This work presents the simulation of dense crowds of more than a million agents on the GPU. The freely available implementation is based on the OpenCL framework and runs primarily on the GPU without any significant participation of the CPU. The system is based on the continuum theory of flow fields. The simulated agents exhibit smooth trajectories and even at high densities intersections are prevented. Furthermore, emergent phenomena that have been observed in human crowds are simulated. The implementation is highly configurable and embedded into a GUI system, which allows easy extensibility and experimentation.

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Innsbruck, October 18, 2011

Helmut Duregger

Submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck in partial fulfillment of the requirements for the degree of Master of Science.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1D	- one-dimensional
2D	- two-dimensional
3D	- three-dimensional
ALU	- arithmetic logic unit
AMD	- Advanced Micro Devices
API	- application programming interface
APP	- Accelerated Parallel Processing
APU	- accelerated processing unit
BSP	- binary space partitioning
CAL	- Compute Abstraction Layer
Cell	- Cell Broadband Engine Architecture
Cg	- C for graphics
CPU	- central processing unit
CU	- compute unit
CUDA	- Compute Unified Device Architecture
FIM	- Fast Iterative Method

FMA	- fused multiply-add
FMM	- Fast Marching Method
FPS	- Frames Per Second
GFLOPS	- giga floating point operations per second
GLSL	- OpenGL Shading Language
GPGPU	- General-Purpose Computation on the GPU
GPU	- graphics processing unit
GUI	- graphical user interface
HLSL	- High Level Shading Language
ID	- identifier
IK	- inverse kinematics
ILP	- instruction-level parallelism
ISA	- instruction set architecture
LDS	- local data store
LOD	- level of detail
MRT	- Multiple Render Targets
NaN	- Not a Number
NDRange	- N-dimensional Range
OpenCL	- Open Computing Language
OpenGL	- Open Graphics Library
OUM	- Ordered Upwind Method
PC	- personal computer
PE	- processing element
PLE	- Principle of Least Effort
PNG	- Portable Network Graphics
RAM	- random-access memory
RGBA	- Red Green Blue Alpha
RVO	- Reciprocal Velocity Obstacle
SaP	- Sweep and Prune

SDK	-	software development kit
SIMD	-	Single Instruction, Multiple Data
SPE	-	synergistic processing element
SPMD	-	Single Program, Multiple Data
UAV	-	Unordered Access View
USP	-	unique selling proposition
VLIW	-	very long instruction word
VO	-	Velocity Obstacle
VRAM	-	video random-access memory
WC	-	write-combining

1 INTRODUCTION

This thesis investigates how crowd simulations with up to one million agents can be performed primarily on the graphics cards of modern personal computers (PCs) through the use of the Open Computing Language (OpenCL). Crowd simulation techniques and applications are surveyed and put into the context of developing parallel programs on the graphics processing unit (GPU). An example implementation has been created that runs the simulation of large and dense crowds on the graphics hardware without any significant participation of the central processing unit (CPU). The simulation runs in near real-time for the intended one million agents and in real-time for a medium to high number of agents. This document contains a detailed explanation of the program's method of operation, and ends with experimental results, summarizing the outcome of this endeavor. This document and the implementation is made publicly available on the web at

- https://github.com/hduregger/crowd

so that others may learn from it. This chapter continues with a general overview about crowd simulation and ends with the document overview.

Wikipedia defines crowd simulation as "*... the process of simulating the movement of a large number of objects ...*" [91]. This very broad definition already suggests the size of this topic. Depending on the application the objects (agents) could be humans, animals, vehicles, alien lifeforms, or something else completely. The number of agents may adept a theoretically infinite number of values. The methods applicable for displaying movement on computers encompass the listing of position vectors in a spreadsheet, as well as the detailed animation and visualization of three-dimensional (3D) models. Crowd simulation encompasses many topics rooted in psychology, artificial intelligence, robotics, animation, rendering, physics, navigation, and more. Hence this chapter can only scratch the surface and give a short overview of the numerous aspects of simulating crowds on computers. Nonetheless some information about applications in the film industry, biology, civil engineering, emergency scenarios and computer games will be given.

Simulations fall into the two categories *offline* and *real-time*. An offline simulation computes the result of the simulation and presents it afterwards, or it exposes intermediate results in time steps that are greater than the time steps in the simulation. This allows for computations to take longer than the steps being simulated. Such

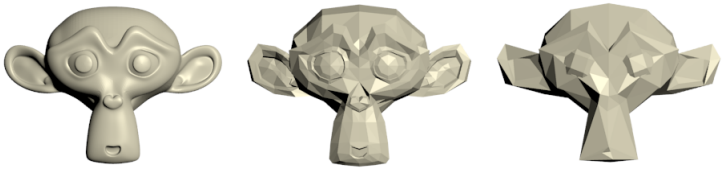


Figure 1: Models with different level of detail.

simulations often run on dedicated hardware and server farms. This is common in the film industry and scientific community.

In a real-time simulation the current intermediate result can be inspected immediately and the program might even take input and allow interaction with the simulation. The time steps between updates are equal to or smaller than the time spent for computation. This extra requirement of real-time presentation and interactivity makes it more difficult to avoid compromises in the quality of the movement and visualization. Even more so, if the underlying platform is consumer-grade hardware, like PC, gaming consoles or even mobile devices, that features less computational power than dedicated hardware might offer.

City designers profit from crowd simulation when planning to link residential, commercial, and industrial areas with transportation. An example simulation of a busy street crossing [23, 84] is given in the video accompanying [26]. Some simulations deal with the psychological and social aspects of human behavior as the designers need to know where people move to and at what time. In a wide-scale city simulation the architect does not need a detailed visual representation of each individual agent. Instead, statistical data and graphical overviews, as presented in [17], are more meaningful.

The computations required to simulate the large number of agents in a city are infeasible for real-time applications like computer games. The computer game Grand Theft Auto IV simulates a lively city including pedestrians and vehicles [19]. The environment and agents are visualized with detailed and animated 3D models. In order to lower the computational workload, level of detail (LOD) methods are used. In close-ups, detailed models are being rendered, in the distance the game renders objects with lower triangle counts. Figure 1 shows an example of the technique. From left to right the models feature decreasing LODs.

The concept of a *bubble* further allows for optimization. The bubble is a shape surrounding the player character in the game. It separates agents that are located inside the bubble from those that are situated outside of the bubble. Figure 2 shows a player character surrounded by a circular bubble inside a two-dimensional (2D) environment. Only agents inside the bubble are simulated with very detailed methods. Agents outside of the bubble are either simulated in some abstract fashion or not at all. Usually this poses no problem because the player

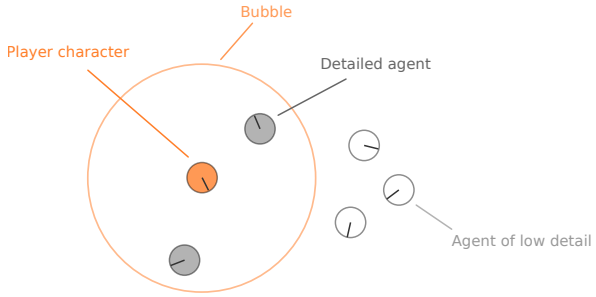


Figure 2: A player character surrounded by a bubble.

can't tell the difference. But this concept of a bubble can become obvious when users start to follow agents around and analyze their behavior. For instance, cars might loop on simple routes. This is unproblematic when they only drive through the player's bubble. But the player might discover this unnatural behavior by following them on their tour. Even worse, agents might sometimes vanish in front of the player's eyes, because their preplanned route and lifetime ended, and the developers did not include a solution that adepts to the situation. Another difficulty lies in the necessity to have agents move into the bubble, while other agents leave that area. If this diffusion of agents is not balanced, over time the bubble will either be drained of or filled up with agents which can ruin the player's immersion.

In order to prevent such problems Sunshine-Hill and Badler [72] use a detailed offline simulation that simulates a whole city to gather data. The agents follow a number of daily routines that involve actions like moving from home to the work place, or from the work place to a café for lunch break. Then they approximate that simulation with statistical methods to get a real-time simulation that is perceptually indistinguishable from the detailed simulation without the need for computing all the rigorous agent planning and simulation steps.

The design of a subway station with paths leading on and off the platforms focuses on optimizing the flow speed of the crowd, and how many people can get on and off the trains in limited time and space. This notion of flow indicates a similarity between crowd movement and liquids. An occurrence of dense crowd flows in the Moscow subway is presented in [56], and a concrete example application of crowd simulation is given in [48], where London Underground conducted studies about how the passenger flow through ticket gates can be improved.

When planning evacuations and emergency exits space is an even more limiting factor and the density of the crowd is of great interest as high densities can lead to injuries or even fatalities [44]. Therefore architects attempt to minimize the density at choke points to lower the risk to human life. Real-time simulation and response would be very beneficial when dealing with emergency scenarios. Optimally, surveillance staff could be notified of dangerous situations before

they occur [74]. Still, such simulations often run offline on dedicated hardware and only afterwards can the operator analyze the results. In interactive media like computer games crowd simulations must respond immediately to input while still maintaining a minimal degree of accuracy. This puts additional constraints on the implementation of the simulation because PCs and gaming consoles don't feature the computational power of dedicated simulation hardware.

Biologists can use simulations to study the behavior of animals in herds, swarms or schools. The impressive displays generated by starlings [21] have been studies in works such as [31].

Large battle scenes like the ones presented in the Lord of the Rings [9] or Star Wars films draw the audience deeper into the storyline. The employed systems allow designers and artists to rapidly prototype different scenes with placeholder agents before rendering the final sequence, all without the need to organize a large number of actors.

In the computer game StarCraft 2 the agents of the fictional Zerg race exhibit a distinctive flow behavior when attacking in large numbers [32]. While this looks impressive, there are also issues with the path finding system where agents get stuck, seemingly because they can not decide on what direction to take [73]. Another game, Supreme Commander 2 supports lane formation when groups of agents cross [24]. In Left 4 Dead the zombie agents attacking in hordes dynamically circumvent obstacles by casting rays into the environment [11]. This makes level design easier because designers don't need to tag objects so that agents can climb over them. But it also poses problems where agents climb over objects they should not climb over like street lamps [49]. Heavy Rain simulates human crowds in a believable manner [77].

This short excursion into a small number of selected applications and techniques should emphasize the need to compromise while developing crowd simulation systems. Negative examples just highlight this necessity, and should be seen as an educational opportunity for improvement. Chapter 2 continues with specific algorithms and methods related to crowd simulation. Chapter 3 mentions how GPUs can be used for computations unrelated to graphics and how the development system, an Advanced Micro Devices (AMD) Radeon HD 6950, functions. The central goal of this thesis is given in Chapter 4. The three subsequent chapters describe how agents find to their goal in the implementation (Chapter 5), how they move and how intersections are prevented (Chapter 6), and how the simulation results are presented to the user (Chapter 7). Experimental results are listed in Chapter 8, Chapter 9 summarizes related work and Chapter 10 provides a final conclusion.

2 CROWD SIMULATION

This chapter deals with three fundamental topics that were of interest during the development of the application described in this thesis, navigation, collision detection and resolution, and visualization. The examples and techniques mentioned herein are far from complete, but should lead to a basic understanding of crowd simulation related algorithms and approaches.

2.1 NAVIGATION

Applications of multi-agent navigation can have several requirements. For cases where a detailed visualization of the simulated agents is given, they generally should

- reach their destination in reasonable time along a reasonable path,
- avoid colliding or even intersecting with other agents or obstacles,
- exhibit a reasonable movement scheme.

Whether the quality of the resulting simulation is acceptable and what is determined reasonable depends on the type of agent and application. Backwards moving agents in a pedestrian simulation could give reason for concern while the same movement scheme might be totally acceptable for alien lifeforms in a computer game. Likewise a policeman hunting a thief should not wander off in a random trajectory, instead it should follow the thief, anticipate its movement and take shortcuts. Such notions of predictive planning or even learning are beyond the scope of this thesis and lie in artificial intelligence research areas.

A number of concepts such as

- global path planning,
- local obstacle avoidance,
- steering,
- flow-fields

can be found in the topic of agent navigation. Their responsibilities overlap. For instance, the problem of circumventing a dynamic obstacle is solved with global planning or with local obstacle avoidance. The following sections explain these concepts in more detail.

2.1.1 *Global Path Planning*

Pathfinding allows an agent to plot the path from its current position towards its goal location. When done as a global planning step, it can incorporate the motivation an agent has when selecting its path to a destination and model the decision process that weighs the pros and cons of path traits. In general an agent will try to minimize the distance it has to move and lower its effort in order to conserve energy [81, 26].

To find a shortest path in a weighted graph *Dijkstra's* search algorithm [18] is applicable. The weight (cost) on an edge may be given as

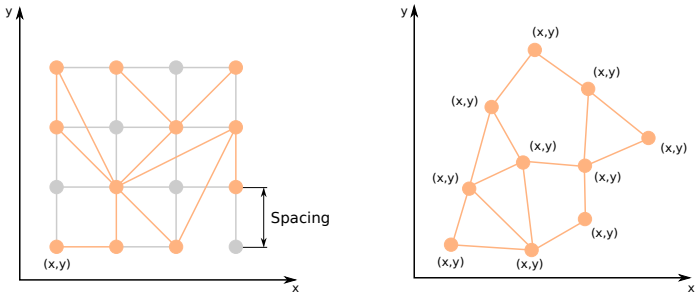
$$EdgeCost_e = Distance_e + TerrainCost_e$$

where $EdgeCost_e$ is the cost assigned to edge e, $Distance_e$ may be the Euclidean distance between the two nodes of e, and $TerrainCost_e$ reflects the property of the terrain. The lower the distance and the lower the cost caused by the terrain, the lower the cost along the edge. As a result, it is possible to express terrain properties like inclines and surface qualities like pavement or grass in the navigation data. A downside of Dijkstra's algorithm is that apart from the edge cost it has no information about how the nodes are laid out. Therefore, when deciding between outgoing edges, it treats those with the same cost equally. This causes it to inspect nodes in a circular fashion around the start node. Which includes nodes that need not be visited.

The A^* algorithm [29] is a generalization of Dijkstra's algorithm. While the latter only sums the edge costs leading up to the current node n, the former also adds a heuristic value that has been assigned to n. This heuristic is an estimation of the remaining path cost leading to the goal and can be useful for biasing the search algorithm into the direction of the goal nodes. If the Euclidean distance to the goal location is used as a heuristic, the total cost assigned to the path leading up to the current node may be given as

$$CurrentPathCost_n = \sum_{e \in E} EdgeCost_e + DistanceToGoal_n$$

where e is an edge in the set of edges E included in the path, and $DistanceToGoal_n$ is the heuristic assigned to the current node. The algorithm will then inspect those nodes that lie closer to the goal node first. In order to find the optimal path the heuristic should be admissible, which means that it does not overestimate the remaining cost. If the developer wants to save processing time by replacing the Euclidean distance computation with the computationally simpler Manhattan distance, A^* might not find the optimal path to the goal any longer. But it will still find a path and in many cases it is sufficient.



(a) Implicit node positions through grid location, shape and spacing.

(b) Explicit node positions.

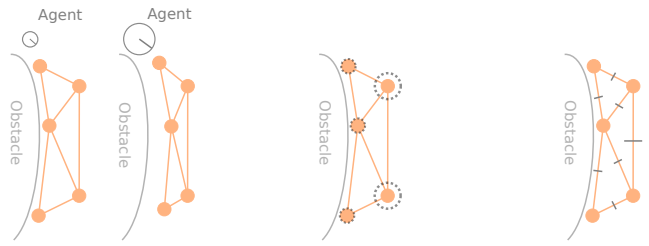
Figure 3: Different grid node storage schemes.

A discussion on the optimization of A* implementations is given in [6].

There are different ways for discretization of a continuous environment into a set of *waypoint* nodes that can be analyzed by path finding algorithms. One way is to use a regular grid of nodes where the position of the nodes are stored implicitly in the position, shape and spacing of the grid (Figure 3a). As a consequence it is not required to explicitly store the position of all nodes, because it can be derived from the grid information. A downside is that areas which can never be reached by agents will still contain unnecessary nodes that are marked as inactive and not reachable over any edges. Another waypoint technique stores the position of all nodes explicitly, thereby creating free-form grids and saving memory where no nodes should be located (Figure 3b). This improvement has to be weighed against the increased memory requirement per node. An example of the free-form waypoint grid used in the computer game Alien Swarm is given in [15]. Both types of waypoint grids (sometimes called roadmaps) are usually interpreted as only allowing paths leading along edges.

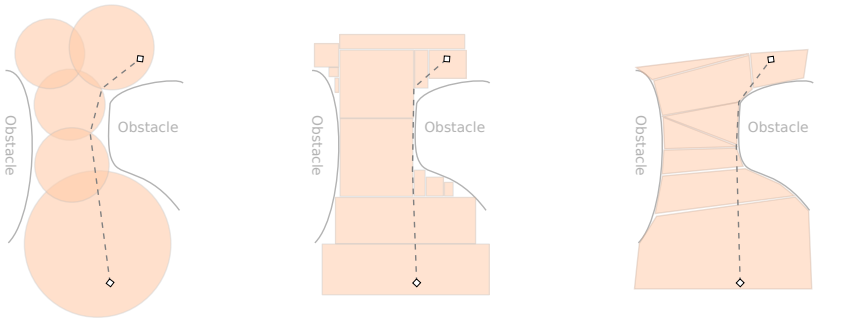
Next there is the differentiation of what a node represents. Some methods just see it as an infinitesimal point that represents only a location. One way to support agents of variable dimension is to use different grids for each supported agent size (or range of sizes) (Figure 4a). As an alternative, a radius stored per node can define the width of agents allowed to traverse over the node (Figure 4b). This radius can also be stored per edge instead (Figure 4c). Again the developer has to weigh off the storage requirements of additional grids with the memory and computational time needed for storing and processing (ranges of) radii.

The area around a node can be further interpreted. A node may cover its surrounding area with shapes such as circles (Figure 5a), rectangles (Figure 5b) or polygons (Figure 5c) that closely approximate the environment. An agent is then allowed to move freely inside these



(a) Separate grids. (b) Radius per node. (c) Radius per edge.

Figure 4: Different ways to define the allowed agent size on the grid.



(a) Circles. (b) Axis-aligned rectangles. (c) Convex polygons.

Figure 5: Several methods of environment discretization.

areas. A sample path is indicated by the dotted black line, terminated with diamonds. Any path inside the shape is valid. These *navigation meshes* allow to cover environment areas with a lower number of nodes by allowing nodes to take up space. As a downside it requires a more complex path construction algorithm, but it can lead to smoother paths and additionally save memory by requiring a lower number of nodes. A manually constructed navigation mesh system is being described by Valve Corporation [85], Booth [10], Champandard [14]. It is implemented in the Source game engine middle-ware and uses rectangles for its shapes. The game Fallout 3 uses another system with polygons [67]. Mononen [52, 51] developed a system to automatically generate polygonal navigation meshes from environment geometry.

There are many ways of how to define edges in waypoint grids. Figure 6 shows a path through grids of different connectivity and the result of path smoothing. A basic idea is to connect each node to all other nodes that are reachable in a straight line (Figure 6a). This results in many edges to store and process in the search. A compromise is to store only edges between neighboring nodes in a specific distance. This lowers the fidelity of the navigation information and agents sometimes have to take a diversion although it would be valid to move in a straight line (Figure 6b). The effect can be limited by

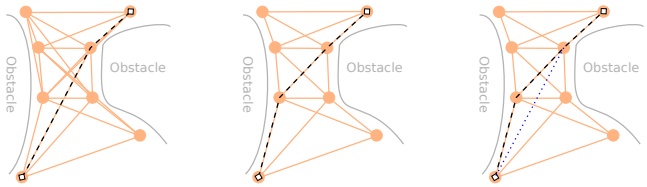


Figure 6: (a, b) Variable grid connectivity. (c) Path smoothing.

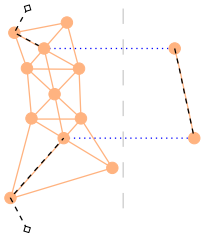


Figure 7: Hierarchical path-finding with high resolution grid on the left, and low resolution grid on the right.

first computing the path along the edges stored in the navigation data and then smoothing it by performing ray-obstacle intersection tests (*ray-casting*) between nodes in the path to probe for valid shortcuts through the environment. In Figure 6c the dotted blue line indicates the derived shortcut. Problems arise when the ray casting scheme does not respect the shape of the agent, potential collisions are missed and the resulting path leads the agent through an obstacle. To lower the likelihood of that problem simple shapes that correspond to the form of the agent, like spheres, capsules or rectangular boxes can be swept through the environment and tested for intersection with obstacles.

The complexity of the path search algorithm is proportional to the number of nodes and edges, therefore it is beneficial to lower the node count. In order to preserve the resolution when using grids *hierarchical* approaches are helpful [69, 12]. As an example, when two grids of different resolution are utilized, the high resolution grid is only used near the entry and exit points to the navigation grids. The remaining path segments are derived from the low resolution grid. Thus, the search algorithm does not need to inspect as many nodes and edges, and it saves computational time by performing larger hops. Figure 7 shows an example. At special transition nodes, the search algorithm can switch between the grids, as indicated by the dotted blue lines. Again, the start and end points of the path are indicated by diamonds.

A big challenge are dynamic obstacles. If an obstacle suddenly blocks an edge it must be invalidated, else agents might collide with the object. Sometimes the environment might still provide enough space to circumvent and therefore the system should still allow agents to reach their goal. If there is no other valid path, that leads around the obstacle, represented in the navigation data, then it must be updated

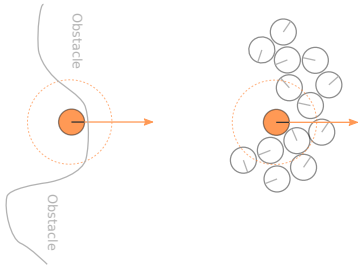


Figure 8: U-shaped obstacles. Static obstacle on the left and dynamic obstacle on the right. The arrow indicates the general heading. The circle covers the limited sensor range.

by introducing new nodes and edges. This can be done for waypoint nodes [70] and also for navigation meshes [50]. The developers of Uncharted combined a static navigation mesh, that covers the whole environment, with a dynamic grid that is constructed in limited range around the agent by rasterization of the static mesh [7]. During the rasterization process they can take dynamic obstacles into account.

These techniques require additional computations and algorithmic finesse. Therefore some navigation systems fall back to cheaper local obstacle avoidance methods that are less reliable than global planning approaches.

2.1.2 *Local Obstacle Avoidance*

In local obstacle avoidance schemes agents commonly only sense those objects situated in a local environment around them. Consequently the information they have available is very limited and they need to react quickly when other objects come into the range of their sensors to prevent collisions. Agents still need a global heading as input into the local decision cycle. Therefore global planning methods often feed their desired direction of motion into them. If the movement of an agent is largely based on local obstacle avoidance then it may get stuck at obstacles that are of *U-shape* (Figure 8), because it is difficult to find a detour with only local information. Nonetheless local obstacle avoidance can constitute a useful addition to global planning methods. In the resulting hybrid schemes it is possible to derive valid paths that avoid collisions, without the need to update the navigation information and search for a new path as soon as dynamic obstacles are encountered.

A method often applied in a local scope is the *Velocity Obstacle* (VO) algorithm [22]. Figure 9 shows how VOs are constructed. First, the size and velocity of agent A and obstacle B are known (Figure 9a). Then object A is reduced to a point, while B is enlarged by the radius of A (Figure 9b). Next, two tangents are constructed that enclose the *Collision Cone*. Any relative velocity $v_a - v_b$ that lies inside the cone

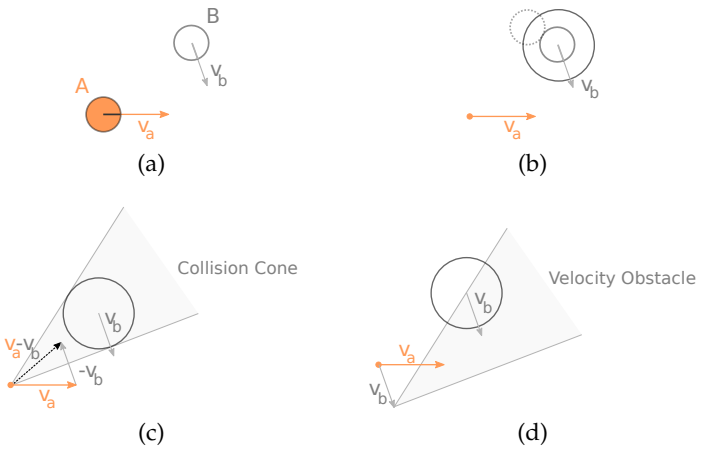


Figure 9: Velocity Obstacle construction. (a) Agent A and obstacle B with velocities. (b) A reduced to a point and B enlarged by radius of A. (c) Relative velocity and Collision Cone. (d) Velocity Obstacle.

will lead to a collision (Figure 9c). Shifting the Collision Cone by v_b results in the VO (Figure 9d). To prevent a collision, A must choose a velocity that lies outside of the VO. Multiple obstacles can be included in the decision process by combining the VOs.

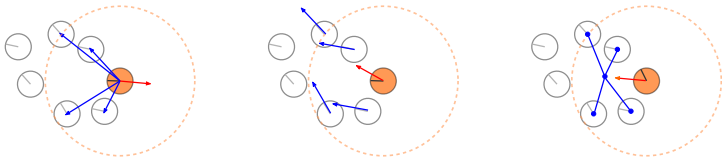
The technique is commonly used in computer games and closely tied to robotics research. In systems where each robot works independent of a central planning authority, such algorithms are a means of preventing collisions between agents. Unfortunately it can suffer from oscillations, where agents repeatedly attempt to evade each other. Extensions such as the Reciprocal Velocity Obstacle (RVO) technique [86] can help to address this problem. Parallel implementations are available [68].

More complex approaches allow to express sophisticated behavior like interactions between and inside groups of agents while trying to maintain formation [61, 39], or prioritizing individual agents [94] and groups of agents [61] at choke points.

2.1.3 *Steering*

Basically, steering describes how agents turn and move when transforming the global and local movement decisions into actual motion. This incorporates agent traits like minimum and maximum values for speed, turn rate, acceleration and deceleration, whether the agent can move backwards or sideways, and how it is oriented in space.

Sometimes Velocity Obstacle methods are named steering methods. Particle-based approaches like the one by Reynolds [57] are also being referred to as steering methods. Here the agents, named *boids*, are controlled by *steering forces* to derive a final velocity. Each agent only takes other agents inside a certain radius into consideration. Figure 10



(a) Separation. (b) Alignment. (c) Cohesion.

Figure 10: Different boid steering forces.

shows some resulting force vectors as red arrows and their cause in blue. The use of multiple forces allows to express *separation* where agents avoid other agents (Figure 10a), motion *alignment* with agents moving into the same general direction (Figure 10b), and *cohesion* for keeping groups of agents together (Figure 10c). Leader-follower relationships are also possible. A force can additionally guide an agent along a path that has been planned in a previous global step. However, the weights and priorities of steering forces are difficult to control.

As an emergent phenomenon, groups of agents simulated under these simple rules can exhibit complex behavior. This allows to approximate animal herd, school and swarm motion, like the impressive aerobatics displayed by starlings, which were also modeled in more detail in [31].

2.1.4 *Flow-Fields*

The path-finding systems mentioned so far work by searching for a valid path for each individual agent. While this allows to incorporate per-agent decision processes into the simulation, the process becomes too expensive as the number of agents increases [89]. When circumventing dynamic objects local avoidance schemes can be used as a work-around, instead of constantly updating the navigation data. But these methods have problems with limited perception and may cause agents to get stuck at U-shaped obstacles. Fortunately there is a different approach that can help dealing with these limitations.

The process of deriving paths can also be expressed similar to finding the solution to problems in *wave propagation* or *fluid dynamics*. In these models the environment and agents are represented as continuous entities. Individual agents don't have point-like or circular representations with strict boundaries, instead their influence fades out with increasing distance to their center. The environment itself also remains continuous and is not explicitly modeled as a grid. Domain[1] properties are then defined as scalar or vector fields. For instance, the agent density is a scalar that may be retrieved for every location in the domain, while the average agent velocity is an example of a vector field.

1 Throughout this document a domain is considered to be a connected open set [90, 93].

Still, on a computer the algorithms for solving these problems require a discretization of the continuous representation. In theory it is possible to compute very accurate solutions with highly detailed discretization schemes, in reality the available processing power often becomes a limiting factor. The computational complexity now stems from solving the problem on a reasonably accurate discretization, and the environment and agent discretization replace the agent count as the dominant quantity controlling the complexity.

A prominent example is *Continuum Crowds* by Treuille et al. [81]. What follows is an overview of the technique, a more detailed explanation and an example implementation is given in Chapter 5. The domain is discretized into a rectangular grid of cells. The underlying continuous model can be seen as featuring an endless amount of infinitely small cells. The smaller the cell size, the more accurate the solution will be.[2] Each cell of the grid is considered to be 4-connected. Thus, it has four neighbors and does not know about the cells in the diagonal direction.

A cost value is then mapped to each of the four sides (north, east, south, west) of a grid cell. This defines a direction-dependent (*anisotropic*), scalar field. It describes how costly it is to traverse from the current cell into the neighboring cell in one of the four directions. This is similar to the cost assigned to edges in Dijkstra's algorithm mentioned in Section 2.1.1.

The costs are derived from environmental properties (*discomfort*) and from the agents themselves (*density*, *velocity*), by mapping (*splatting*) agent information onto the cell grid. For instance, it is cheaper for an agent to move into the same direction as neighboring agents. And it is also cheaper to avoid areas of high agent density or discomfort.

Each cell is then either assigned to the set of cells belonging to a goal area or to the set of remaining cells. The goal cells are assigned a *potential* of zero. The authors then apply an *Ordered Upwind Method (*OUM) [64] to compute the solution. A comparison between Dijkstra's algorithm and OUMs is given in [83, 64]. In each successive step the algorithm expands the goal area outwards, thereby computing the paths from all cells to the goal cells in reverse. This expanding wave-like front also increments the potential of visited cells through a function depending on the cost and potential of neighboring cells. Thus the scalar potential field is zero at goal areas and increases towards those cells that are further away. Finally, the algorithm computes the *gradient* field base on this potential field. This gives the steepest ascents. The agents can then sample this vector field locally and move "downhill" in the direction of the negative gradients towards the goal. This can be imagined as a mountainous area where agents will later wander downhill towards the goal.

2 This is not necessarily true for all algorithms based on discretization schemes. Some will not give better solutions with increased resolution.

As an advantage of this technique it becomes quite simple to incorporate dynamic obstacles into the global path planning step. The objects can be splatted just like agents. If they contribute a very high or even infinite cost, then the agents will avoid the affected areas automatically. The method constantly updates the gradient field, thus agents will also find their way around congested areas. For all agents in the field there is a path starting at their locations, which makes supporting large numbers of agents feasible. Unfortunately, the cell size has to be small to reflect the shape of dynamic obstacles. In order to fit such a simulation into tight real-time constraints, a compromise between the fidelity of the cell grid and the size of the covered environment has to be made.

March of the Froblins by Shopf et al. [65] is an implementation that runs shader programs on the GPU. It combines the technique with Velocity Obstacles. [47] presents another GPU-based implementation. Similar techniques are being used in games like StarCraft 2, Supreme Commander 2 and Heavy Rain [16]. Heavy Rain features an implementation optimized for the synergistic processing elements (SPEs) of the Cell Broadband Engine Architecture (Cell) microprocessor. Another continuum-based approach that performs no path-finding but allows to limit the agent density and thereby helps to prevent collisions is given in [53].

2.2 COLLISION DETECTION AND RESOLUTION

Navigation systems can hardly ever prevent collisions completely. Even more so if a high number of agents are operating in limited space. Artifacts caused by intersecting agents can have a negative impact on simulation results, and are often obvious to the human eye. Collision detection and resolution methods help to prevent agents from intersecting with other agents, obstacles or the environment.

2.2.1 *Broad- and Narrow-phase*

In order to detect intersections the spatial overlap between the involved parties needs to be determined. If the positions of all agents, static, and dynamic obstacles in the environment are known *pair-wise checks* will reveal collisions. In physics engines this search is commonly separated into a *broad-* and a *narrow-phase*. In the broad-phase a subset of potential colliders is filtered out of all participating objects. This is also called collision *culling*. During the narrow-phase the actual pair-wise comparisons are performed. A number of algorithms is suitable for application on the GPU, for instance [27] and [25].

During the broad-phase step space partitioning methods help to control the computational complexity by lowering the number of

objects that must be checked per agent. Examples are binary space partitioning (BSP), *quadtrees*, *octrees*, and *binning*.

Computer game engines commonly further simplify the process by using simple geometric shapes like spheres, capsules, or cubes as aliases for the actual 3D models, which typically consist of triangle meshes. The reduced precision is often not noticed by players. Some game developers even choose to disable inter-agent collisions altogether. In StarCraft this is done for the sake of predictability and in Left 4 Dead it permits smooth motion for large numbers of agents [1].

2.2.2 *Resolution*

Finally, when an overlap was found the colliders can be separated. Sometimes it is not possible to find a perfect resolution. A simple pair-wise separation might cause the potentially resolved agents to intersect with other agents or environment structures. In very crowded scenes with many agents this can lead to oscillations.

2.3 VISUALIZATION

The visual representation of crowd members can have high requirements in terms of quality. The film industry relies on lively depictions of crowd scenes and therefore generously invests into the final rendering process with dedicated server hardware, ray-tracing and post-processing techniques to mimic realistic appearance or skillfully create artistic look, all for the immersion of the audience.

In computer games impressive graphics effects have long been part of the unique selling propositions (USPs), which subsequently lead to an arms race between the creators of computer games, and also between the developers of game engine middle-ware. A large amount of research and development is spent on producing new visual rendering processes. The advent of shader programs introduced the programmability of GPUs and developers began to expose the visual effects supported by their graphics engines through tools, so that artists can tweak the appearance of agents and environments to achieve better visual quality and performance.

Researchers often utilize more abstract visualizations, for instance cylinders as aliases for agents, or curves plotting their trajectory in space. Where applicable, the results are only visualized in spreadsheets, graphs and colored fields.

2.3.1 *Rendering*

Today's consumer-level graphics cards can render large numbers of detailed agents to the screen in real time [24, 32, 77, 19]. The dedicated design of GPUs to stream and raster large numbers of triangles

makes this possible. While the computational power of GPUs has grown rapidly throughout the last few years, the increase in memory bandwidth could not keep up. Therefore modern graphics cards feature dedicated memory located close to and also inside the GPU. The programs then run on a client-server architecture, where the GPU serves the client program that is being executed on the CPU. In order to keep the latency down and bandwidth up, it is beneficial to place as much information as possible directly inside the memory on the graphics card. This is supported in todays programming standards like the Open Graphics Library (OpenGL) and *Direct3D* which expose that memory to the developer in the form of special data buffers. The client program can then instruct the GPU to retrieve vertex, primitive and texture data from the graphics card's memory. If the amount of data surpasses the amount of dedicated video memory, the graphics driver will automatically copy data back and forth between the usually larger client random-access memory (RAM) and the video memory.

Crowd simulations often visualize many instances of very similar agents. This can be achieved by placing the information required to draw each instance into a large buffer on the graphics card and then order the GPU to render with this data. Alternatively the programmer can exchange memory for GPU cycles. Modern graphics cards support *geometry instancing* where the GPU can be instructed to render several copies of the same model. Each replication may have different location, orientation or appearance. Dudash [20] presents a method that also supports animations. Vertex and geometry shader programs provide additional flexibility and the programmer can customize the geometry manipulation to specific needs.

2.3.2 *Animation*

For simulating living beings like humans or animals, simply visualizing the agents trajectory is often not enough. The movement of individual body parts can be as important. Example techniques for improving the visual quality are *forward* and *inverse kinematics (*IK), *motion capturing* and *animation planning*.

Forward kinematics allow to simulate smooth motion by providing snapshots (*key-frames*) of the position and orientation of body parts in predefined intervals and then interpolating them during program execution. A collection of related key-frames is often called *animation clip*. With motion capturing these clips are being generated by recording the movement of actors in a studio which makes it easier to recreate natural motion. The advanced facial motion capturing techniques applied in L.A. Noire [58] have recently gained recognition in the game development community.

IK techniques allow to model processes like a person reaching out to a drinking glass, without the need to provide a separate animation

clip for every possible start state[3]. It works by respecting constraints on the joints of the body and computes valid configurations that lead the hand to the glass. The technology is rooted in the robotics industry where grippers need to dynamically reach out to arbitrarily placed objects, but it is also applied for hand and foot planting in computer games and simulations [60, 8, 71]. With basic forward animation, the animation clips of all possible start states would be required. Yet, by blending several animation clips together it is sometimes possible to derive satisfying forward animations. *Animation blending* computes the weighted average of several input clips.

Even further goes animation planning as presented by Kovar et al. [43] where the trajectory of the agent depends on a collection of animation clips. Here each animation clip contains information about what trajectory the object experiences during the playback of the clip. If the data originates from motion capturing, the collection will contain very accurate information about possible trajectories. Based on this data a *motion graph* of different movement speeds and arcs is constructed. Then transitions between clips with similar key-frames are inserted. Based on this information an animation that results in a trajectory very close to the desired one can be derived, while strictly sticking to the allowed motion constraints. This can help to prevent *foot-skating* artifacts, where the agent's feet slide over the ground during movement, which are prevalent in pure forward kinematics based approaches. Similar works of Treuille et al. [82] and Lee et al. [45] promise real-time performance and improved quality.

3 The relation between agent and glass with respect to location and orientation in space.

3 GENERAL PURPOSE COMPUTATION ON THE GPU

Originally, GPUs were introduced to accelerate the rendering of 3D scenes with dedicated hardware for vertex transformation, lighting, and texturing. The application programming interfaces (APIs) OpenGL and Direct3D emerged and defined the way of programming this new hardware. Their fixed-function pipelines largely determined how the data was processed and only allowed tweaking a handful of steps through configuration variables and commands.

Over the years the hardware features and flexibility increased and assembly languages provided advanced programmability to certain steps in the graphics pipeline. Soon, high-level programming languages like C for graphics (Cg), High Level Shading Language (HLSL), and OpenGL Shading Language (GLSL) simplified the process and added functionality.

Eventually, developers started to use the GPU not only for graphics, but also for physics and other mathematical problems. This General-Purpose Computation on the GPU (GPGPU) in turn caused processor manufacturers to add more functionality to the hardware to serve the new requirements. Eventually dedicated GPGPU APIs like NVIDIA's Compute Unified Device Architecture (CUDA), AMD's Compute Abstraction Layer (CAL), and Microsoft's DirectCompute surfaced.

3.1 OPENCL

The various efforts accumulated into the Open Computing Language (OpenCL) [41] as an attempt at simplifying the software development process by unifying the parallel programming APIs for heterogeneous architectures. OpenCL is supported on multiple devices, for instance CPUs, GPUs, accelerated processing units (APUs), and also Cell.

3.1.1 *Platform Model*

OpenCL provides a standardized view on the components of a hardware system. It is based on a *platform* model consisting of a *host* with access to several *devices*, as illustrated in Figure 11a. The devices are divided into compute units (CUs), which again are segmented into the actual processing elements (PEs) performing the work.

Task-parallel and data-parallel programming paradigms are supported through Single Program, Multiple Data (SPMD) and Single Instruction, Multiple Data (SIMD) *commands* respectively. The commands execute in a *context* that defines memory access and synchro-

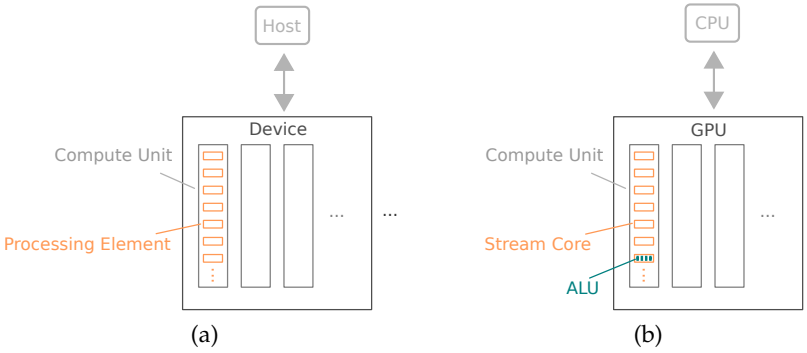


Figure 11: The OpenCL platform model. (a) OpenCL processing components. (b) Their analogy in the Radeon HD 6950.

nization over several devices. These commands are injected into a *command-queue* by the host application where they can be retrieved and executed by the device. The command-queue is defined with a context but limited to a single device. Special commands execute *kernels* which are designated functions of compiled *programs.* These programs are written in a language based on the C programming language.

3.1.2 *Virtual Index Space*

When requesting kernel executions, the application programmer can specify the number of spawned kernel instances by defining an index space with one, two, or three dimensions. This N-dimensional Range (NDRange) is a tuple and can cover different ranges in each direction. If the range for a direction is not specified, the range is implicitly defined as one. For instance, the NDRange $(4, 8)$ is identical to $(4, 8, 1)$, and defines a range of four entries along the x-axis, eight along the y-axis and one in the z-direction of the index space. All invoked kernel instances (*work-items*) hold a unique identifier (ID) that maps to an entry inside the index space. This *global* ID is a triple (x, y, z) with an entry for each direction. The entries are based on an offset that can be passed with the command. In applications that use matrix multiplications, this indexing scheme can be useful for defining which work-item processes what matrix entry. Such assignments are important for the optimization of memory access patterns. But the developer is also free to ignore the identifiers.

Multiple work-items are combined into *work-groups*. This is based on the work-group size passed to the command. The work-group size is again a tuple and unspecified components are implicitly defined as one. The NDRange value of each direction must be an integer multiple of the work-group size specified for that direction. With this information the number of work-groups in each direction can

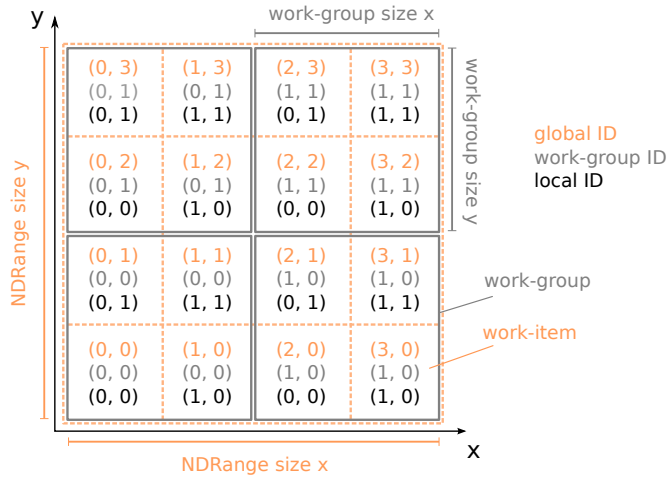


Figure 12: NDRange example in two dimensions, with its relation to work-group size, global and local identifiers.

be determined. The ID of a work-item's group and its index inside the group (*local* ID) can also be retrieved in a kernel. This allows for sophisticated work assignments and communication between work-items. In general, the programmer may not make assumptions about the execution order of work-items, but the elements of a work-group are defined to execute concurrently in a CU. The relation between the index space and the identifiers is depicted in Figure 12. The 2D example shows 16 work-items inside an NDRange of $(4, 4)$ and a work-group size of $(2, 2)$, resulting in four work-groups.

3.1.3 *Memory*

The storage on a device is primarily divided into *global* and *local memory*, but also features *constant* and *private* regions. Global can refer to larger storage areas, for instance, the RAM located on a graphics card's circuit board, sometimes called video random-access memory (VRAM), and local would be the smaller, but faster scratch pad memory installed inside high-end GPUs. This assignment is transparent to the application programmer and OpenCL also allows global and local memory to be regions inside the same hardware components. For instance, with a CPU device the global and local memory both reside in RAM, and in the OpenCL implementation for the Radeon HD 4000 series of GPUs the local memory is emulated in VRAM and does not provide any speed advantages [87]. This makes porting applications between platforms not necessarily a trivial task and it remains beneficial to study hardware specifications and implementation details for the purpose of finding opportunities for device specific optimizations. Constant memory can not be changed during kernel execution, and private memory is only visible to single work-items. Figure 13a shows an overview of the memory layout.

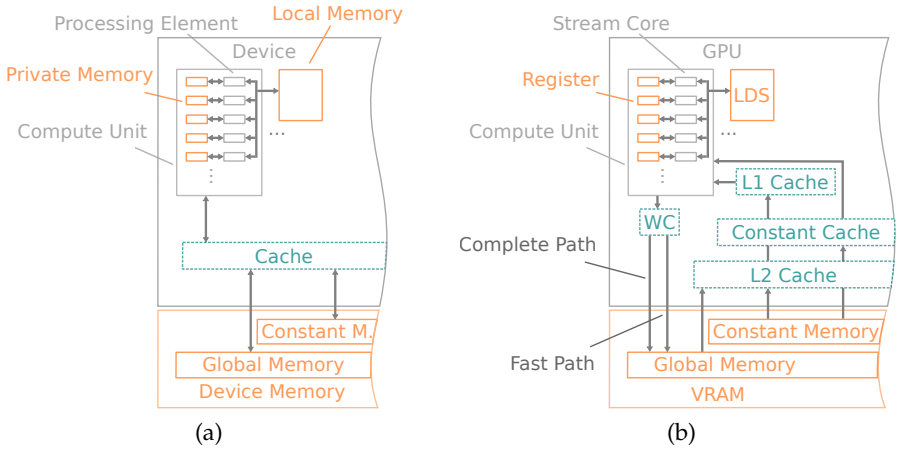


Figure 13: The OpenCL memory structure. (a) OpenCL memory regions. (b) Their analogy in the Radeon HD 6950.

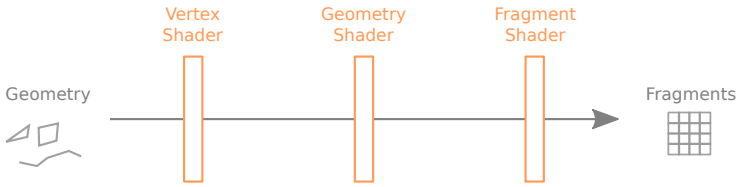


Figure 14: Simplified OpenGL pipeline.

Developers can access global memory through *buffer* and *image objects*. The former is a linear array and kernels can access the sequentially stored elements through pointers. The latter allows special, filtered access through *sampler objects*, and storage details are transparent to the programmer. An example for a tiled memory layout for image data is given in [3]. The implementation may further store the image data in a format different from how the data is presented in the kernel. For instance, an integer might internally be stored as a floating-point number. Again the underlying hardware may provide variable support. On graphics cards the image objects are sometimes mapped to dedicated hardware intended for texture manipulation. The standard further defines interfaces to data stored with OpenGL and Direct3D. OpenCL additionally simplifies the development of parallel programs on the GPU by exposing Unordered Access View (UAV).

UAV is a GPU hardware feature that allows PEs to store data into arbitrary locations inside buffers and images. In applications developed with OpenGL developers have to render geometry in order to run instances of their general purpose shader programs. A simplified representation of the OpenGL processing pipeline is shown in Figure 14. The geometry is transformed by vertex and geometry shaders and finally rasterized into several fragments. For each fragment an instance of the fragment shader program is run. Unfortunately the output of a fragment shader has a predefined storage location inside buffers and

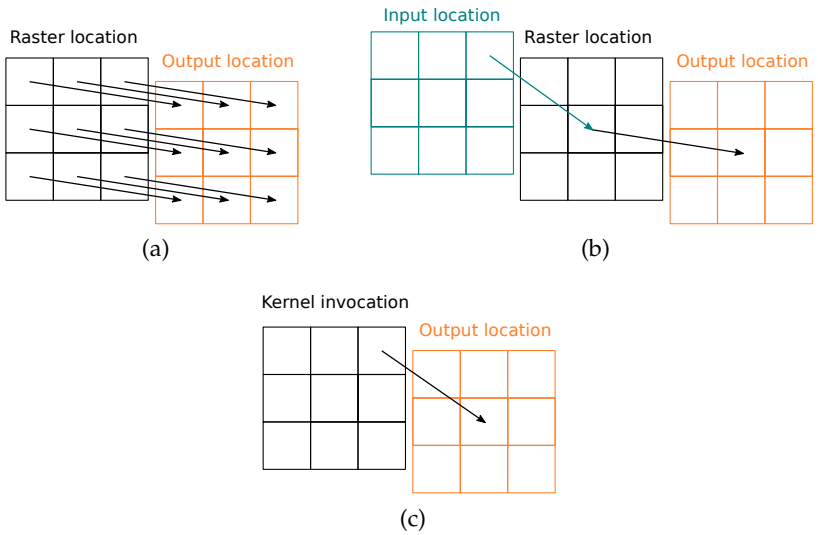


Figure 15: Different relations between program invocation and storage location.

textures.[1] In order to write into arbitrary regions, developers have to raster geometry so that the fragment output is mapped to the intended target region. In many cases algorithms must also be inverted into a gather scheme, where the rendered fragment defines the destination and fetches the data from other regions. Figure 15 visualizes the issue. In OpenGL the raster location directly corresponds to an output region (Figure 15a). By gathering data from an arbitrary location in an input texture, the information can be passed to another location (Figure 15b). With the exposure of UAV in OpenCL each kernel invocation can directly write to arbitrary positions in the buffer or image (Figure 15c).

Still, some memory operations are restricted. For instance it is not allowed to read and write to the same image inside a kernel. If an algorithm reads from a source image, processes the data, and then wants to write it back to the same image, this is not possible. Instead it can use double-buffering methods. They are also named *ping-pong* schemes, because the data is pushed back and forth between two identical images. Listing 1 illustrates the technique. In the pseudo-code a host application stores the handles to the two images in an array with two entries and retrieves them by their indexes. After each kernel invocation the indexes, and therefore images, can be swapped with a simple reassignment to $one - index$.

1 This OpenGL output restriction has been somewhat lifted by the very recent OpenGL 4.2 standard. Here pixel shaders may write to arbitrary locations and atomic operations are also supported. Still, OpenGL 4.2 features are likely only available on OpenCL capable hardware. Which means that GPGPU on older hardware must still be performed with workarounds. On newer hardware OpenCL remains more approachable and flexible for GPGPU tasks, because of how kernels can be issued and memory is accessed.

Listing 1: Pseudo-code illustrating ping-pong scheme.

```
Image[2] images;

int source      = 0;
int destination = 1;

loop
{
    process(images[source], images[destination]);

    source      = 1 - source;
    destination = 1 - destination;
}
```

Data that is repeatedly referenced inside a kernel invocation should be copied to local memory first. On modern GPUs the program can thereby benefit from the lower latency of the scratch pad memory. Shared access to local memory is restricted to work-items within a single work-group, but can be synchronized between participating items with special `barrier` instructions. The `barrier` forces any work-item inside a work-group to wait until all other items in the group have executed the instruction. If a `barrier` instruction is encountered by any work-item in a group, then all work-items inside the group must encounter it. Special care has to be taken so that this is also assured inside conditional statements and loops, else undefined behavior might occur. The programmer may also use *fence* instructions to ascertain that memory operations to global or local memory have finished.

Albeit not a magic wand, OpenCL is a functioning attempt at unifying the interface to current heterogeneous platforms. It allows the development of parallel programs, without the limitations of OpenGL on GPUs, and the intricacies of low-level threading on CPUs.

3.2 THE RADEON HD 6950

Application development for this thesis started out on a Radeon HD 4870. The developers of March of the Froblins already showed that crowd simulation is feasible on such hardware. But their implementation is limited to much lower agent numbers (65000 maximum, 16384 with simplified, and 3000 with detailed visualization [66]) than the one million intended for the application accompanying this thesis. And they apparently optimized the computation of the navigation data by spreading it over several frames, as the update every n frames in the demo indicates. Thus, the limited computational power and the restricted OpenCL support for local memory, as mentioned earlier, induced a switch to the more powerful Radeon HD 6950 graphics card.

This section discusses some of its hardware specific features and how they relate to OpenCL.

The AMD Radeon HD 6950 is part of the 6900 series. It's architecture also carries the name Cayman PRO, with Cayman being the name of the 6900 line's architecture, part of the Northern Islands family of GPUs (the 6000 series). But this document will stick to the numbered naming scheme for the graphics cards to emphasize the heritage between the products, instead of introducing additional names.

3.2.1 *Computation*

Most information provided in this section is based on [3] which provides a good insight into how AMD's current graphics cards function. Many examples directly refer to the Radeon HD 5870 design but sometimes this separation is not obvious. For the sake of discussion, it is assumed that many of the properties of the 5870 line carry over to the 6950 architecture. For most of the numbers listed in this document the exact specifications for the 6950 are given in the Programming Guide. Where this is clearly not the case, it will be explicitly mentioned. Based on the knowledge that the 6900 is a minimal redesign of the 6800, which in turn is based on the 5870 architecture, it can be considered unlikely that the functionality of the compute resources and the memory system differs much.

The card uses very long instruction words (VLIWs) in order to achieve instruction-level parallelism (ILP), as mentioned in AMD [4, 3]. While the 6800 GPUs still featured an instruction set architecture (ISA) with five-way VLIWs, the 6900 series was redesigned into a four-way VLIW architecture, allegedly without any practical performance impact [38, 5]. Figure 11b shows how the 6950's hierarchy of processing elements maps to OpenCL's platform model. The hardware features *Stream Cores*, each including four arithmetic logic units (ALUs). Instructions to these four ALUs can be issued in a single VLIW. The 6950 has 16 Stream Cores per CU and 22 CUs overall. This gives a total of 1408 ALUs running at 800 MHz clock speed. Thus it provides a theoretical peak performance of $\#ALUs \cdot ClockSpeed \cdot Operations = 1408 \cdot 800 \cdot 10^6\,\text{Hz} \cdot 2 = 2252.8$ giga floating point operations per second (GFLOPS). The number of operations is two because each ALU can execute a fused multiply-add (FMA) instruction in a single cycle. An FMA is a multiplication followed by an addition ($a \cdot b + c$). This computational power is comparable to what machines in the TOP500 achieved in the year 2000 [78]. Table 1 provides an overview of the number of compute resources, showing how many elements are contained per parent item, and the total number inside the device.

Unfortunately, the nomenclature used in the June 2011 edition of [3] is inconsistent. Sometimes the ALUs are referred to as PEs, in other sections the Stream Cores correspond to PEs. This is probably a relict

NAME	NUMBER PER PARENT	TOTAL NUMBER
GPU	-	1
Compute Unit	22	22
Stream Core	16	352
ALU	4	1408

Table 1: Number of compute resources in the AMD Radeon HD 6950 architecture.

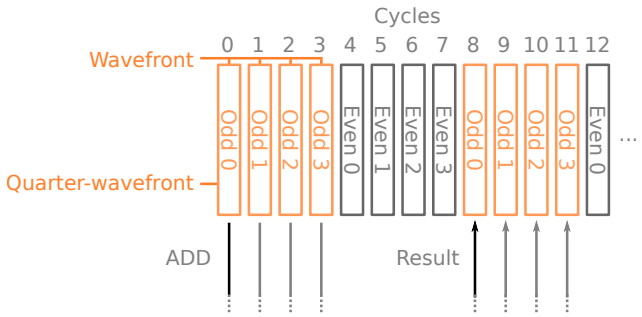


Figure 16: Wavefront processing for an ADD instruction.

of pre-OpenCL document versions (likely because [4] uses a similar naming scheme) and hopefully will be cleaned up. However, the OpenCL standard defines work-items with "*A work-item may execute on one or more processing elements.*" [41], and work-items can not be executed on individual ALUs in this GPU. Therefore, referring to the Stream Cores as PEs seems correct and this naming convention will be adopted throughout this document.

Work-items and work-groups are issued in the row-major order defined on the index space, filling the ranges first in x, then y, and finally z-direction. The maximum number of work-items per work-group is 256. Work-groups are further divided into *wavefronts* which consist of 64 work-items each, therefore four wavefronts constitute such a work-group.

The wavefronts are executed on compute units. A CU processes two wavefronts (*odd* and *even*) in an interleaved manner. This allows to hide the read-after-write latency of eight cycles for most ALU operations. Figure 16 shows an example for the floating-point add instruction. First the odd wavefront is executed. All 16 Stream Cores execute the same VLIW on a *quarter-wavefront* per cycle and the instruction for the whole odd wavefront can be handled in four cycles. Then the CU processes the even wavefront for four cycles, before it changes back to the odd wavefront. The result of the instruction is now available and can be used with the next instruction.

The compiler combines VLIW of the same type into *clauses* that are managed by flow control. ALU instructions are grouped with local

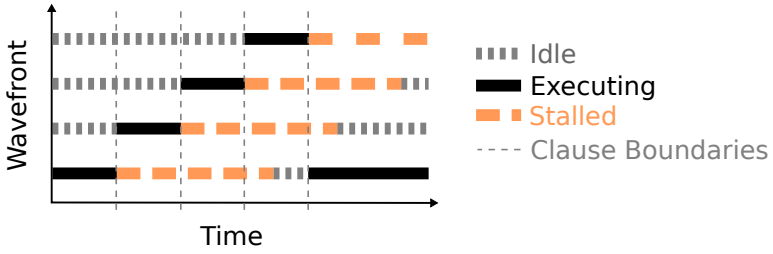


Figure 17: Hiding stalls with multiple wavefronts.

data store (LDS) operations. LDS is the low-latency memory inside the GPU, sometimes it is also named *local data share*. Global memory accesses need to be grouped into separate clauses. Clause switching causes an overhead and therefore clauses should contain as many instructions as possible. Apart from the odd-even scheme explained above, the CU does only switch clauses at flow-control boundaries. This means that a new clause can only be entered if all instructions of the previous clause have completed. If a LDS memory request can not be served immediately it will stall the CU until the memory operation has finished. The CU does not hide this stall by switching to a different clause or wavefront.

To hide the latency of global memory accesses, the CU does switch between wavefronts. The example in Figure 17 starts with the first wavefront executing while the other three are idle. It stalls on a global memory access and the CU schedules the next wavefront for execution. As soon as this one stalls the next one starts to execute. This continues until the first wavefront's memory access has completed and the CU can return execution to it at a clause boundary. It is beneficial to have many running wavefronts to increase the opportunity of hiding stalls. The maximum number of concurrent wavefronts that the GPU can manage depends on hardware limits and how much register space each kernel occupies [3, 30], because the register file is dynamically distributed between work-items.

The GPU may initialize resources during clause switches. For instance, it can load constants whose addresses are available at compile time into the constant cache. This typically includes single constant variables and function parameters.

Three example clauses (`01`, `02`, `03`) are given in Listing 2. The first one represents a read from global memory. The second clause includes three ALU operations. It shows an instance (`7`) where an instruction has been mapped to two (`x`, `y`) of the four ALUs (named `x`, `y`, `z`, `w`). The final example clause is a flow-control command. A more detailed definition of the structure of clauses and the ISA is given in [4].

Wavefronts are at the lowest granularity that instruction flow control can regard. If two work-items divert at a conditional statement then all participating work-items must evaluate both statements. Inapplicable results are masked out and only the appropriate ones are used. For

Listing 2: Instructions grouped into clauses.

```
...
01 TEX: ADDR(1216) CNT(1)
       6  VFETCH R6.xy__, R5.x, fc173  FORMAT(32_32_FLOAT)
          FETCH_TYPE(NO_INDEX_OFFSET)
02 ALU_PUSH_BEFORE: ADDR(272) CNT(4)
       7  x: SETGT_DX10  R0.x,  0.0f,  R6.x
          y: SETNE_DX10  R0.y,  R6.x,  R6.x
       8  z: OR_INT      R0.z,  PV7.x,  PV7.y
       9  x: PREDNE_INT  ____,  R0.z,  0.0f  UPDATE_EXEC_MASK UPDATE_PRED
03 JUMP  ADDR(6)
...
```

Listing 3: Conditional scalar assignment with `select` statement.

```
if (a) { i = 0; }
if (b) { i = 1; }
...
// becomes

i = select(i, 0, a);
i = select(i, 1, b);
...
```

instance, if the conditional branches into statements A and B (e.g. `if (isTrue){ A } else { B }`), and each statement would be executed by at least one work-item inside the wavefront, then the execution time of the wavefront for this conditional will be the sum of the execution times of A and B. These doubled executions are a reason why conditional statements should be avoided on GPUs. Sometimes it is cheaper to just compute a value across all work-items inside a wavefront instead of conditionally skipping the computation and branching. There are cases where the conditional can be translated into multiplications by zero and one. In circumstances where the operations concern vector types the OpenCL `select` statement can be used. But according to [3], the `select` statement should also be used for conditional scalar assignments, as shown in Listing 3. This prevents the creation of additional clauses which would cause unnecessary overhead.

Due to C language requirements the evaluation of a boolean expression with several terms inside an if statement must take short-cuts [40]. For instance, in the expression `a && b`, the left operand `a` is evaluated first and `b` is only evaluated if `a` does not result in zero. To support this in if statements (e.g. `if (a && b){...}`) the compiler must generate several clauses. This can be prevented by assigning the boolean

NAME	SIZE	TOTAL SIZE	PEAK READ BANDWIDTH
VRAM	2 GB / GPU	2 GB	160 GB/s
L2 Cache	512 kB / GPU	512 kB	410 GB/s
L1 Cache	8 kB / CU	176 kB	1126 GB/s
LDS	32 kB / CU	704 kB	2253 GB/s
Register	256 kB / CU	5632 kB	13517 GB/s
Constant Cache	48 kB / GPU	48 kB	4506 GB/s

Table 2: Memory resources in the AMD Radeon HD 6950 architecture.

expression to a variable first and then using that variable inside the if statement (`int i = a && b; if (i){...}`).

Of similar importance are loops. A loop will be executed until the last work-item inside the wavefront exits the loop. Therefore the execution time of the wavefront is the maximum execution time of its work-items for the loop.

3.2.2 *Memory*

The relation between the Radeon's memory structure and OpenCL is shown in Figure 13b. Individual PEs store their private variables in the register file. All PEs of a CU can exchange data through the LDS. Table 2 lists the different memory components with their size and peak read bandwidth [3]. The VRAM can store 2 GB of data on the Radeon HD 6950 but only part of that is available through OpenCL. The register file might seem large, but as mentioned above it has to be shared between the 16 Stream Cores (256 kB/16 = 16 kB) and multiple wavefronts.

Constant and global memory reads are cached through Level 1 (*L1*) and Level 2 (*L2*) caches, but there are also read instructions that explicitly bypass the caches. Moreover, according to [3] (August 2011), AMD's software development kit (SDK) with version 2.4 only caches reads from buffers if the buffer pointer is explicitly marked with the C language keywords `const` (value pointed to can not be changed) and `restrict` (no aliasing with other pointers). In [88] it is stated that the SDK 2.5 allows the GPU to cache loads from writable buffers in certain circumstances. During experiments with different kernels and the 2.5 SDK, caching was performed even if the `const` and `restrict` keywords were absent. [88] also indicates that load coalescing is supported, which lowers the number of addresses that need to be transmitted to global memory for lookup. Apparently, images are always cached. As mentioned earlier, special constant variables can reside in the constant cache.

Writes to global memory are managed through a write-combining (WC) cache. It caches multiple write requests before committing them to global memory in order to achieve better access patterns. The data then moves over two distinct memory lanes, the *Fast Path* and the *Complete Path*. On the Fast Path no complicated operations are performed and the data can be quickly committed to memory.

The Complete Path provides functionality necessary for OpenCL *atomic instructions*. For operations that return the old datum (e.g. *atomic_add*) it writes the old and the new entry into global memory at locations given in the instruction. Then it notifies the PE so that it can read the old value with special load requests that bypass the L2 cache. This notification is also used with OpenCL fence instructions. The Complete Path additionally manages data writes of less than 32 bits (e.g. `char` and `short`). Operations on the Complete Path suffer an extra delay compared to the ones on the Fast Path, therefore developers should try to use atomics with LDS and rewrite their algorithms to use data types with 32 bits. The global memory paths are wide enough to transmit the `float4` vector data type and programs should make use of it. The Fast Path supports coalesced writes, which lowers the number of destination addresses that have to be transmitted to the memory controller that supervises access to VRAM. Because the data items are laid out in a sequential fashion a single destination address is sufficient, instead of one address per participating work-item. The performance gain is however marginal because the memory controller takes far longer to handle the requests than it takes to transmit them.

Each address of a global memory access maps to a certain *channel* and *bank* controller. Figure 18a shows how memory addresses might map to the controllers on the 6950. According to [3] it features eight channels, but the exact location of the channel bits and the number and location of bank bits is taken from the description of the Radeon HD 5870. For the sake of discussion, it is assumed that the 6950 has the same address layout. If two PEs request memory that maps to the same channel (bank), a *channel conflict* (*bank conflict*) occurs and the controller serializes the access. As an example, two addresses A and B with

$$(A/256) \bmod 8 \equiv (B/256) \bmod 8 \equiv 0$$

will map to the first channel controller and cause a conflict. This problem is similar to cache line conflicts encountered in CPU architectures. It hardly ever occurs with memory accesses of *stride* one, with stride being the address spacing between memory accesses initiated by two work-items. But strides of larger powers of two that coincide with the address assignment as given in the example above are prone to this problem. In matrix processing two neighboring elements in a column are sometimes separated by a stride that is a power of two. Concurrent access to such neighbors may lead to a conflict. Sometimes this can be prevented by padding the matrix with an extra column, thereby

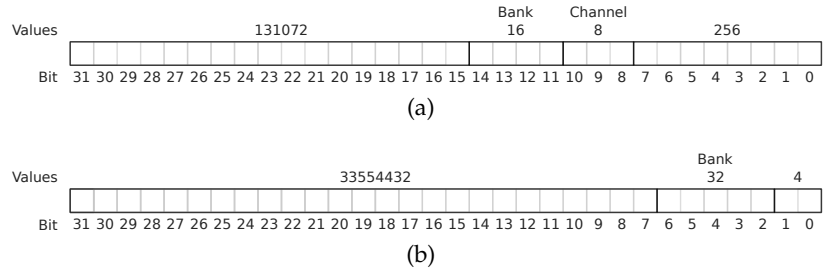


Figure 18: Address assignment to memory controllers. (a) Global memory address with to bank and channel bits. (b) Local memory address with bank assignment.

generating a stride that is not a power of two. In general it is difficult to predict bank conflicts for GPU programs, because many CUs are concurrently requesting memory accesses.

In the LDS each local memory address maps to a bank as illustrated in Figure 18b, and 32 concurrent accesses can be served per cycle. Nonetheless, if two work-items request data that maps to the same bank then the processing is stalled. As mentioned earlier the CU continues executing the wavefront and fills in the data as it arrives. Because 16 work-items (quarter-wavefront) are processed per cycle the LDS manages to deliver two values of four-bytes length (e.g. two `floats`) per cycle, as long as no addresses inside the wavefront map to the same bank. As an exception, when all 16 work-items of a CU access the same address the system can distribute the value to all Stream Cores in a single-cycle scatter operation. The LDS does not feature any special support for coalescing, but it has dedicated hardware that performs atomic operations much faster than the global memory.

The Radeon HD 6950's ancestry can be seen in its ability to stream large amounts of data through parallel programs with limited interdependence, as is typical for many applications in computer graphics. In GPGPU conditional statements and complicated flow logic make it more difficult to keep the throughput high and the ALUs working to capacity. Still, given the right application and the optimal implementation consumer-level graphics cards in general can provide an impressive performance at a reasonable affordability.

4

PROBLEM STATEMENT

The previous chapters provide a frame for understanding crowd simulation related challenges and techniques. They also explain the functionality of the development system that provides OpenCL on a contemporary GPUs. Now that this base has been set, the goal of the experimental part of this thesis can be stated.

The objective is to simulate up to a million agents, that navigate through an environment without intersecting with other agents or obstacles. Preferably the agents should be visualized and animated in a 3D scene and the program should allow some form of interactivity with them. The simulation should run in real time or at least close to real time for the interaction to remain enjoyable and it should use OpenCL to perform its processing on a GPU with hardly any interaction of the CPU, in order to keep that processor available to other tasks.

Based on these extensive requirements and the information provided in the introductory chapters a number of assumptions can be stated. Global path planning with methods like A* are unlikely to succeed as they have been termed too costly for large numbers of agents, even if the cost could be controlled with a hierarchical approach. They still require conditional tracking of multiple solution paths for a single agent, which seems inappropriate for GPU architectures that prefer simple data streams instead of complex decision logics. Steering based approaches with their difficult predictability and controllability seem less suitable as a solution to the problem. As hinted previously, flow-fields seem to map well to the kind of data processing that GPUs favor and they have been applied on graphics cards before. Their regular cell structure is similar to pixel grids in graphics processing. It should be possible to process large numbers of cells concurrently and dynamic objects are implicitly dealt with. Simple object placement, like the one in the March of the Froblins demo, should provide interactivity. Collision detection and VO like methods remain problematic because they indicate random memory access patterns occurring when an agent checks its neighbors. Still, the previous chapters mention GPU-based approaches for both so this seems achievable. Visualization is the domain of graphics cards therefore no problems should be expected.

Figure 19 shows the basic building blocks that constitute the intended simulation loop. The agents and the environment information is fed into the navigation planning step that derives movement directions for each agent. Next the agents are offset from their current

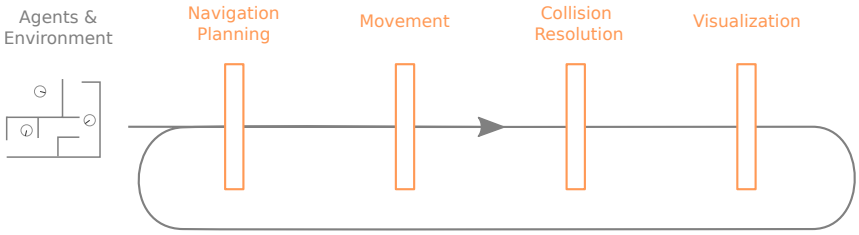


Figure 19: Basic building blocks of the simulation loop.

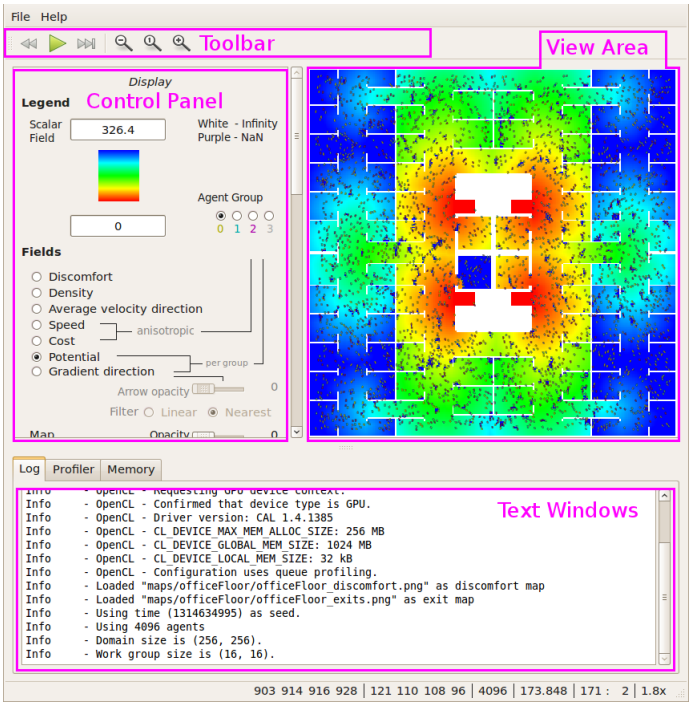


Figure 20: An instance of the program.

position into their movement direction. Eventual collisions and intersections are resolved and the agents and the environment can be visualized. Finally, the updated information reenters the loop.

If this undertaking succeeds, developers will have a working example of the given approaches that is available to everyone. This is in contrast to the existing solutions that are unavailable to the general public and where the CPU involvement is unclear. Furthermore the implementation should provide enough opportunity for experimentation to allow a comparison with other techniques. Last but not least, it will provide a base for further development. It reveals possibilities for improvement and students can learn from its faults.

Because it is based on OpenCL it will be easier to extend the functionality and apply it to specific applications. For instance it could be transformed into a tool for architects. This would allow the use of

consumer-grade hardware to model scenarios involving large crowds, as is relevant for building design. The large interactivity would allow rapid prototyping of choke points and evacuation paths. It could also be incorporated into a computer game, thereby providing interactivity and gameplay with a huge number of agents.

The subsequent chapters investigate how the assumptions mentioned above have been translated into a working application and what decisions influenced the course of the experiment. Figure 20 shows a running instance of the program.

5 COMPUTING THE NAVIGATION DATA

To lead the agents towards their goal the method in [81] is used. It does not explicitly compute and maintain paths consisting of nodes and edges like A* related methods do. Instead, each agent implicitly derives the shortest path from its current position towards the goal location by moving in the direction of vectors inside a gradient field. The way in which this navigation information is derived has several advantages over A* like methods because

- the complexity of the computation primarily depends on the size of the domain and the resolution of the cell grid,
- it equally supports small and large numbers of agents,
- dynamic obstacles can be easily integrated and
- when mapped to work-items it contains less interdependence

which makes it suitable for applications on the GPU. This suggests that agent numbers higher than those feasible with A* like approaches can be used. But it does also highlight the domain size and resolution as the limiting factor.

The introductory section about flow-fields (Section 2.1.4) already gives an overview of how the method works. The program uses environmental information and the influence of the agents to derive a cost function over the domain. Based on the cost a potential field can then be computed. The gradients on that field will later serve as indicators to the agent headings. Figure 21 shows which components in the program developed for this thesis are involved in the computation of the gradient field. Each outer shape corresponds to a C++ class that applies OpenCL kernels or OpenGL shader programs to the data buffers. The orange rectangles are OpenCL buffer objects and the teal rounded boxes are images. The data type inside the OpenCL containers and their usage is given in the figure. The black rectangles are OpenGL textures. The dotted lines indicate data passing between components and buffers. The following sections go into further detail by describing how the program operates and what each component does step-by-step.

5.1 INITIALIZATION

First the program reads several configuration parameters from a text file. This includes options like the number of agents to simulate, the

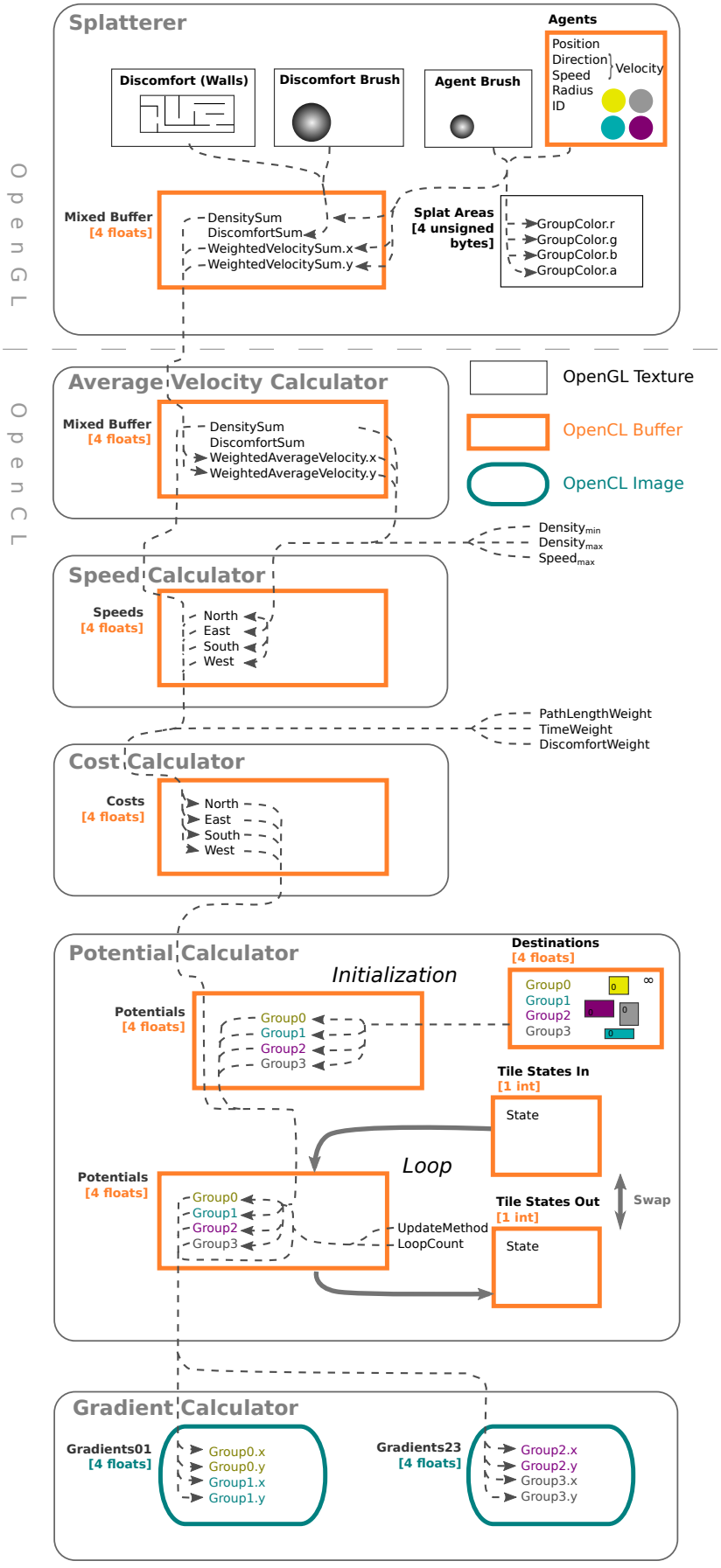


Figure 21: Overview of the navigation pipeline.

minimum and maximum agent radius, an optional random seed, the location of reentry areas, the work-group size to use, and the names of image files used as the discomfort and exit (goal) maps. The discomfort file contains the walls of the map, areas with increased discomfort, and also provides the size of the cell grid to the simulation. Valid sizes are limited to square images with a side length that is a power of two. The red color channel is used as discomfort and any red intensity greater than or equal to 0.99 is later transformed to a wall with infinite discomfort. The exit map defines where the agents are heading to, by storing zeros at destination cells. All distance and size units used in the implementation relate to the cell spacing. If an agent has a radius of one, then it is two cell spacings wide. Thereby the user can choose a relation between the spacing and the simulated domain. This allows to experiment with different resolutions by readjusting the agent size.

Next a data buffer is created that contains the information stored for each agent. Like all other buffers it is placed into VRAM. Starting position and radius are randomized and each agent is assigned to one of four agent groups. This is because of a limitation of the flow-field method. A single gradient field only provides the paths to a single collection of destination cells. If different agents should move to completely different goal destinations then multiple gradient fields need to be computed. Similar to March of the Froblins [65] four such gradient fields are computed concurrently. This is accomplished by utilizing the ILP provided in the four ALUs by issuing operations on the float4 data type. Thus, four sets of goal cells can be processed simultaneously and all agents inside one of the four groups will head for the goal cells assigned to their group.

The initialization phase additionally creates the contexts required for OpenGL and OpenCL. Some suitable computation and the visualization is performed through OpenGL. There is also a graphical user interface (GUI) system involved that provides some interactivity with the simulation and opportunity for experimentation.

5.2 DENSITY AND VELOCITY

The so called *Splatterer* exclusively uses OpenGL shader programs because they are especially suited to all its operations based on rendering. First it prepares two destination buffers, the *Mixed Buffer* and the *Splat Areas*. Three of the Mixed Buffer's components are cleared to zero, and the y-component is initialized with the base discomfort provided by the image file inside $[0, 1]$ (inclusive). Here the cells surpassing the wall threshold (0.99) are converted to infinite discomfort (wall). The Splat Areas buffer is initialized to all zero (transparent black).

OpenGL's blending is then enabled and the blend equation is set to additive blending. The equation becomes

$$Color_{i,j} = 1 \cdot Source_{i,j} + 1 \cdot Destination_{i,j},$$

Listing 4: Additive blending with OpenGL.

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
glBlendEquation(GL_FUNC_ADD);
```

Figure 22: The cone texture.

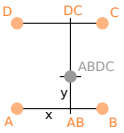


Figure 23: Bilinear interpolation.

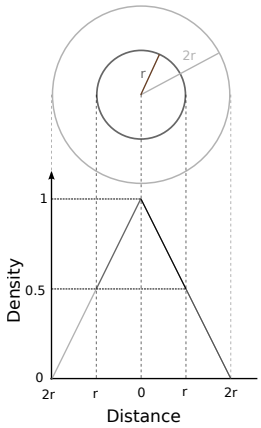


Figure 24: The agent density cone.

where $Source_{i,j}$ is the color of the rendered fragment, $Destination_{i,j}$ is the color of the current fragment in the buffer, and $Color_{i,j}$ is the new color in the buffer at the cell with coordinates (i, j). In the equation above the ones are explicitly listed to indicate the parameters to the corresponding OpenGL function, as shown in Listing 4. The Mixed Buffer and the Splat Areas are set as destination buffers in a Multiple Render Targets (MRT) configuration. This will cause all subsequently rendered data to add up inside these buffers.

Then the Splatterer draws a single *quad* to render the *Discomfort Brush* into the Mixed Buffer's y-component. The quad consists of two aligned triangles that form a rectangle. The Discomfort Brush is a user placed obstacle and its position can be selected with the middle mouse button. The amount of discomfort and the radius of the brush are configurable through GUI elements. The Splatterer uses a geometry shader to create a quad centered at the location of the brush and uses its diameter as the side length for the quad. When the brush is rendered the given discomfort is multiplied with a texture that causes the discomfort to fall off radially from its full intensity towards zero. Figure 22 shows the texture. It can be imagined as a cone. In the center, on the peak, the highest discomfort is rendered, here the texture corresponds to one. It then falls off linearly towards the base where it is zero. During movement the agents will try to circumvent this area of additional discomfort where the brush has been placed. This allows to test how the agents evade dynamic obstacles that contribute discomfort.

Finally, the per-agent information is rendered into the buffers. For each agent position and radius the Splatterer again uses a geometry shader to render a cone-textured quad. This time the texture directly represents the density for the agent which lies in $[0, 1]$ (inclusive). The agent is rendered with its radius doubled. The minimal allowed agent radius is 0.4, because for smaller quads the information is poorly mapped to the buffer. The doubling of the radius causes the density to be 0.5 at the agent's actual radius as illustrated in Figure 24. The rationale behind this is given at the end of Section 5.4. The density is written into the x-component of the buffer. The agent's velocity is weighted by the density and written into the z and w components. Concurrently each written fragment is colored in the group color and added to the Splat Areas which are used to visualize the agent contributions to the fragments (cells). The agent's group is encoded in its ID, as described in Chapter 6. Further information about this visualization is given in Chapter 7. In order to take up less space in the figure, the Agents buffer in Figure 21 actually stands for several OpenCL buffers that store all the agent properties as indicated in Figure 34.

After all agents of all groups have been processed the Mixed Buffer contains the sum of all density contributions, the sum of discomforts, and a weighted sum of the agent velocities. This gives a summed discomfort at cell (i, j) of

$$DiscomfortSum_{i,j} = \begin{cases} \infty & \text{if } D_{Map_{i,j}} \geq 0.99 \\ D_{Map_{i,j}} + Cone_{i,j} \cdot Intensity & \text{else} \end{cases},$$

where $D_{Map_{i,j}}$ is the density contributed to the cell (i, j) by the map file, $Cone_{i,j}$ is the factor enforced by the cone texture, and $Intensity$ is the discomfort brush intensity set in the GUI. The density at cell (i, j) is summed over all agents as

$$DensitySum_{i,j} = \sum_{a \epsilon Agents} Density_{a_{i,j}}, \tag{5.1}$$

and the weighted velocity sum is given as

$$WeightedVelocitySum_{i,j} = \sum_{a \epsilon Agents} Density_{a_{i,j}} \cdot Velocity_a, \tag{5.2}$$

where a is an agent in the set of agents, $Density_{a_{i,j}}$ is the density contributed by the agent to cell (i, j) taken directly from the cone texture, and $Velocity_a$ is the current velocity of the agent. Thus each agent's velocity contribution is weighted by its density and its influence fades out towards the doubled radius. The closer an agent moves toward another agent, the stronger the influence of that other agent becomes.

Whenever an arbitrarily positioned, scaled, and textured quad is rendered the texture elements (*texel*s) hardly ever align directly with

the centers of the fragment grid (the cells). Therefore the value contributing to the cell (i, j) is computed with bilinear interpolation of the values of the four closest texels in order to approximate the value at the intermediate position.[1] Figure 23 illustrates how this works. If V_A, V_B, V_C, and V_D are the values at the four contributing texels A, B, C, and D, and the fragment, with normalized texture coordinates (u, v), lies at the coordinates $(u', v') = (u \cdot TextureWidth, v \cdot TextureHeight)$. Then by using the floor function the fractional parts of the texture coordinates become $(x, y) = (u', v') - \lfloor (u', v') \rfloor$. The value V_{ABDC} at the sample location $ABDC$ for the fragment can now be computed with the two intermediate values V_{AB} and V_{DC} as

$$V_{AB} = xV_B + (1 - x)V_A, \tag{5.3}$$

$$V_{DC} = xV_C + (1 - x)V_D, \tag{5.4}$$

$$\begin{aligned} V_{ABDC} &= yV_{DC} + (1 - y)V_{AB} \\ &= (1 - x)(1 - y)V_A + x(1 - y)V_B + (1 - x)yV_D + xyV_C \quad \text{by (5.3)(5.4).} \end{aligned} \tag{5.5}$$

For vector values component-wise multiplication and addition is used.

5.3 AVERAGE VELOCITY

The *Average Velocity Calculator* bridges between those components that use OpenGL shader programs and those that use OpenCL kernels. The data in the Mixed Buffer is shared between both APIs. Buffers that have been created in the context of OpenGL need to be acquired by OpenCL before they can be used in a kernel. After the processing they need to be released so that the control is returned to OpenGL.

A one-dimensional (1D) NDRange is used for the kernel invocation. The kernel divides the $WeightedVelocitySum_{i,j}$ by the $DensitySum_{i,j}$, resulting in

$$\begin{aligned} WeightedAverageVelocity_{i,j} &= \frac{WeightedVelocitySum_{i,j}}{DensitySum_{i,j}} \\ &= \frac{\sum_{a \epsilon Agents} Density_{a_{i,j}} \cdot Velocity_a}{\sum_{a \epsilon Agents} Density_{a_{i,j}}} \quad \text{by (5.1)(5.2)} \end{aligned}$$

as the weighted mean agent velocity at each grid cell. Extra care is taken, not to divide by zero in cells where no density has been contributed by any agent. The result is then written back into the Mixed Buffer.

1 The name bilinear interpolation (also bilinear filtering) is misleading. The procedure is not a linear function in the strict mathematical sense.

5.4 SPEED

The *Speed Calculator* computes an anisotropic speed field that stores four values at each cell. The values stand for the speed an agent can achieve when moving in one of the four directions, north, east, south, or west. This is based on the idea that it is easier for an agent to move in the same direction as the agents surrounding it, instead of moving against the crowd flow. Again, a 1D NDRange is used to process all cells as work-items, and the kernel derives a speed for each direction based on the $WeightedAverageVelocity_{i,j}$ and $DensitySum_{i,j}$ computed in the previous steps.

First the $FlowSpeed_{i,j_d}$ is computed per direction $d \epsilon \{N, E, S, W\}$, corresponding to up, right, down, and left in the 2D coordinate system of the program. This is just the dot product of the direction and the $WeightedAverageVelocity_{i',j'}$ in the neighboring cell (i', j') in that direction. For instance, for the direction north it would be

$$FlowSpeed_{i,j_N} = (0,1) \cdot WeightedAverageVelocity_{i,j+1},$$

with $FlowSpeed_{i,j_N}$ being the flow speed at cell (i, j) in the up direction (north), and $WeightedAverageVelocity_{i,j+1}$ is the weighted mean of the velocity in cell $(i, j+1)$, which is the cell above cell (i, j). If the neighboring cell lies outside of the grid, the velocity is assumed to be the zero vector. But the resulting four dot products just extract a single component of each velocity and therefore it is easier to just compute them as

$$\begin{aligned}
FlowSpeed_{i,j_N} &= \quad WeightedAverageVelocity^{y}_{i,j+1}, \\
FlowSpeed_{i,j_E} &= \quad WeightedAverageVelocity^{x}_{i+1,j}, \\
FlowSpeed_{i,j_S} &= -WeightedAverageVelocity^{y}_{i,j-1}, \\
FlowSpeed_{i,j_W} &= -WeightedAverageVelocity^{x}_{i-1,j}, \\
&\text{with } FlowSpeed_{i,j_d} \geq 0.001,
\end{aligned}$$

with the superscript providing the x or y component of the velocity vector. From here on the kernel uses operations on the `float4` vector data type to process all four directions simultaneously. A minimal speed of at least 0.001 is enforced in order to prevent unwanted infinite values from occurring during later computations.

In open areas agents should move freely at the maximum speed possible. As indicated in Continuum Crowds, the maximum speed allowed could depend on the topography of the terrain. As an example, agents could slow down when moving up a slope. In the implementation accompanying this thesis no height information is used and the maps are all flat. Still, such a height map could be easily incorporated into the speed calculation, and the maximum speed can be set in the GUI. In congested zones the agents should slow down, therefore

density constraints are applied. Based on the density at the grid cell and user-provided minimum and maximum densities, an interpolated $MediumSpeed_{i,j_d}$ is computed that provides a slow-down for agents experiencing increased crowd density. This speed is given as

$$MediumSpeed_{i,j_d} = Speed_{max} - \frac{DensitySum_{i',j'} - Density_{min}}{Density_{max} - Density_{min}} \cdot (Speed_{max} - FlowSpeed_{i,j_d}),$$

with $Speed_{max}$, $Density_{min}$, $Density_{max}$ being the maximum speed, minimum density, and maximum density set in the GUI. The maximum speed is restricted to values greater than zero. $FlowSpeed_{i,j_d}$ is the flow speed for the direction as extracted above and $DensitySum_{i',j'}$ is the density sum in the neighboring cell (i', j') in the relevant direction. Again, if (i', j') lies outside of the grid, the sum is assumed to be positive infinity. The density term with the fraction bar gives the percentage of how far the density sum reaches into the range between the minimum and maximum density. The term to the right of the multiplication provides the variability in terms of speed. By subtracting it from the maximum speed the medium speed increases inversely proportional to the density. Figure 25 illustrates the behavior. In the example $Density_{min}$ is 0.5, $Density_{max}$ is 0.8, and $Speed_{max}$ is set to twelve. The maximum speed is illustrated with the coarsely dotted orange line, $FlowSpeed_{i,j_d}$ with the cyan dashes, and $MediumSpeed_{i,j_d}$ is given by the fine gray dotted curve. The flow speed is assumed to be five.

Finally, the achievable speed for a direction is computed as

$$Speed_{i,j_d} = \begin{cases} FlowSpeed_{i,j_d} & \text{if } DensitySum_{i',j'} \geq Density_{max} \\ \begin{cases} MediumSpeed_{i,j_d} & \text{if } DensitySum_{i',j'} > Density_{min} \\ Speed_{max} & \text{else} \end{cases} & \text{else} \end{cases}$$

$$\text{with } Speed_{i,j_d} \geq 0.001,$$

in two `select` statements. This results in a speed that is limited by the speed of the crowd if the density is high, linearly increases inside the minimum and maximum density range, and reaches its maximum if the density in the neighboring cell (i', j') is low. The resulting black speed curve in Figure 25 is clamped to the extrema, and linearly interpolated inside the density range.

To ensure that an agent actually can achieve $Speed_{max}$ at low density, the GUI system prevents $Density_{min}$ from being set to a value smaller than 0.5. This is connected with the agent density at its radius, which is also 0.5 as explained in Section 5.2 and illustrated in Figure 24. Later, when calculating its movement, each agent samples the speed field at an offset from its center. The magnitude of this offset vector is at least as large as the radius of the agent. Thus the movement will not be influenced by the agent's own density contribution to the speed computation, except for the error introduced by the grid discretization. Due to all the restrictions the speed can never be zero.

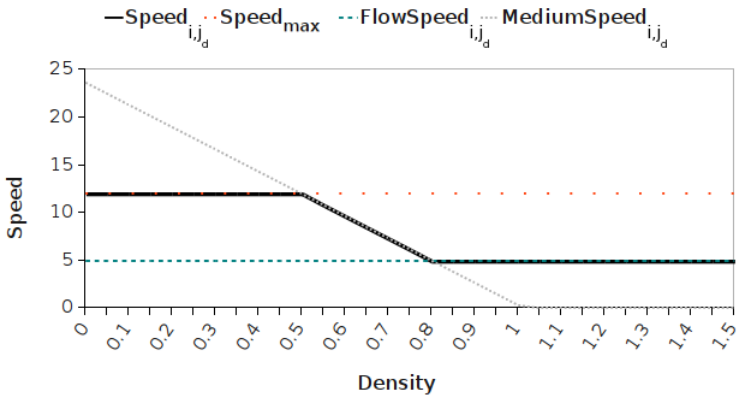


Figure 25: Speed calculation.

5.5 COST

The cost function defines the preferred path. Usually it should be short, so the path *length* is of concern. This is related to the Principle of Least Effort (PLE) as explained in [26], because agents don't want to waste energy. Additionally, agents typically don't like to spend a lot of *time* to get to a destination. The time is a function of path length and *speed*. And agents try to move at the maximum comfortable speed. Furthermore, agents might prefer certain areas over others. For example, someone might want to avoid a dangerous neighborhood at night, and an army has less difficulty crossing a river on a bridge than anywhere else. This notion of *discomfort* should also be reflected in the cost function and agents want to be exposed to unusual discomfort for as short an amount of time as possible.

All these path traits are combined in the continuous cost function with three integrals over the path

$$\begin{aligned} Cost_P &= W_L \int_P 1\,ds + W_T \int_P 1\,dt \qquad + W_D \int_P Discomfort\,dt \\ &= W_L \int_P 1\,ds + W_T \int_P \frac{1}{Speed}\,ds + W_D \int_P \frac{Discomfort}{Speed}\,ds \quad \text{by } dt = \frac{ds}{Speed} \\ &= \int_P \frac{Speed \cdot W_L + W_T + W_D \cdot Discomfort}{Speed}\,ds, \end{aligned}$$

where the $W_{\{L,T,D\}}$ are the length, time and discomfort weight respectively. These weights can be set by the user in the GUI. P is the path, and $Discomfort$ is the discomfort suffered on the path, while $Speed$ is the speed greater than zero that the agent achieves along the path. An integral with ds means that it is taken with respect to the path length, and dt indicates the same for the time spent on the path. According to the requirements an agent will pick the path that minimizes this function.

As explained in the previous section, the speed is anisotropic and depends on the flow direction of the crowd. Therefore the *Cost Calculator* computes a cost value for each of the four directions in the discretized domain. The equation can then be rewritten for a single grid cell and direction into

$$Cost_{i,j_d} = \frac{Speed_{i,j_d} \cdot W_L + W_T + W_D \cdot DiscomfortSum_{i',j'}}{Speed_{i,j_d}}, \quad \text{with } Cost_{i,j_d} > 0,$$

where the meaning of the cost, speed, and weight terms is identical to the ones above, and the $DiscomfortSum_{i',j'}$ is the sum of discomfort in the neighboring cell (i', j') in the relevant direction. If the cell (i', j') is outside of the grid then an infinite discomfort is assumed. With the restrictions in the speed computation, the denominator can never be zero. The employed kernel is invoked on the elements of a 1D NDRange and performs the computation for all four directions simultaneously. By moving across the grid the agent will later sum up the cost of multiple grid cells along the path, implicitly performing the integration in a discretized manner.

The cost function given here is the same as in Continuum Crowds. March of the Froblins uses a different cost function based on static terrain cost, agent density and dynamic hazard. This does not include a direction-dependent term and therefore lane formation, where agents follow other agents heading in the same direction, is not expressed in the cost function.

5.6 POTENTIAL

Now that the cost for leaving each grid cell in a certain direction has been computed, the *Potential Calculator* can deduce the potential field of each of the four agent groups. The potential is set to zero at goal areas, based on the provided exit map image file, and increases outwards. The equation that the algorithm is based on is called *eikonal equation* and has the following form

$$\|\nabla Potential\,(L)\| = Cost\,(L, D)\,, \tag{5.6}$$

where $Potential\,(L)$ is the potential at location L, ∇ is the gradient operator, $\|\ldots\|$ is the Euclidean norm, and $Cost\,(L, D)$ is the direction-dependent cost at location L for direction D. This partial differential equation says that the length of the gradient vector at any location L inside the domain is the same as the cost at that location. Which means, at areas with high cost the potential increases more than at areas of low cost. The potential can be seen as the *first arrival times* of a *wavefront* emanating from the goal area and flooding the rest of the domain. At locations where the cost is high, the wave will take longer

to propagate and the first arrival time will increase. Since the cost depends on the direction, the wavefront will expand faster in certain directions than in others. As the term first arrival time suggests, this is not a full representation of wave behavior though, because waves can not be reflected off boundaries and cause interference. So optical caustics can not be simulated. Another aspect is that the domain can be seen as a *level set*. The set contains groups of cells that have the same potential field value. This concept is similar to the contour lines describing equal height in a topographical map. This makes it easy to understand how agents can move "downhill" against the gradient to reach their goal.

5.6.1 *Discretization*

Because the potential is needed for the subsequent gradient computation the equation has to be solved. For each grid cell a discretized approximation can be used. As with other partial differential equations this is done by providing a finite difference scheme of the continuous equation. This can be transformed to extract the required variable. There are many ways to find a discrete gradient representation for the left side of Equation 5.6. They require multiple samples of the potential field, so that their difference in the x- and y-direction can be computed, to derive the gradient vector. These samples can be taken with different directions, distances, and weights.

The discretization schemes that belong to the family of Ordered Upwind Methods (OUMs) [64] proved especially useful for solving the eikonal equation. OUMs use a one-sided differential to discretize the gradient. This means that a total of three samples are taken from the current cell, a horizontal, and a vertical neighbor. More specifically, the gradient is taken in the *upwind* direction. Meaning that the neighboring cells used, lie in the direction of where the wave is coming from. This again reflects the path of the agent, which will move against the wavefront's propagation direction.

A difference scheme based on [55] turns out to be a working discretization technique. The authors simulate the trajectory of underwater robots, including the anisotropic influence of currents. This thesis uses a modified variant that includes the cost in the decision process, as mentioned in Continuum Crowds. The difference scheme is

$$\left(P_{i,j} - P_x\right)^2 + \left(P_{i,j} - P_y\right)^2 = \left(C_x + C_y\right)^2, \tag{5.7}$$

$$\begin{aligned}
&\text{where } P_x = P_{i-1,j} \text{ and } C_x = C_{i,j_W} \text{ if } \left(P_{i-1,j} + C_{i,j_W}\right) \leq \left(P_{i+1,j} + C_{i,j_E}\right), \\
&\quad \text{else } P_x = P_{i+1,j} \text{ and } C_x = C_{i,j_E}, \\
&\quad \text{and } P_y = P_{i,j-1} \text{ and } C_y = C_{i,j_S} \text{ if } \left(P_{i,j-1} + C_{i,j_S}\right) \leq \left(P_{i,j+1} + C_{i,j_N}\right), \\
&\quad \text{else } P_y = P_{i,j+1} \text{ and } C_y = C_{i,j_N}.
\end{aligned}$$

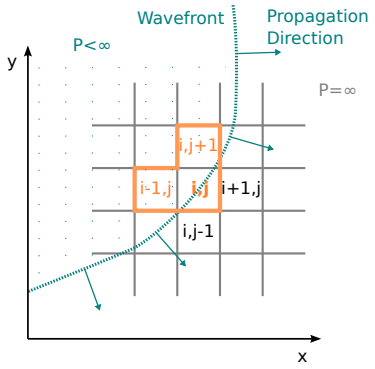


Figure 26: Neighbor selection during cell update.

$P_{i,j}$ is the potential in cell (i, j) and C_{i,j_d} is just shorthand for $Cost_{i,j_d}$. If a neighbor is outside of the grid, the potential is assumed to be infinite. Hence, the potential in the current cell is connected to the two potentials in the neighboring cells and the cost incurred under the chosen direction. The selection of the x- and y-neighbors is based on which sum of neighboring potential and cost, for the direction leading into the neighbor, is smaller, just as in Continuum Crowds. The squares in the scheme can be understood by looking at how the length of a 2D vector (x, y), with regard to the Euclidean norm, is computed as $\|(x, y)\| = \sqrt{x^2 + y^2}$. By multiplying the length to the left and right side, to get rid of the square root, and mirroring the equation it becomes $x^2 + y^2 = \|(x, y)\|^2$, which has the same shape as the difference scheme in Equation 5.7. The vector is the gradient and therefore the components are the differences along both axes, with the grid spacing in x- and y-direction (Δx and Δy) set to one. The cost term includes the direction-dependent cost, similar to what [55] uses. Thus, the resulting potential will reflect the headings of agents, allowing an agent to follow other agents on their path.

Figure 26 shows an example of how a cell (i, j) chooses the neighbors based on the cell propagation direction. Initially the potential in cells belonging to the set of goal cells is zero, in all other cells it is infinite. Cells in the dotted area have already been updated and their potential is less than infinity. The wavefront propagation direction determines which neighboring cells contribute to the current cell's potential, and which directional costs are used. The north and west cells have a much lower potential than the east and south cells, therefore they are used for the update, as highlighted by the orange selection.

To get the potential $P_{i,j}$ at the current grid cell from the difference scheme in Equation 5.7, the quadratic equation must be solved. It is

of the same type as the equation $ax^2 + bx + c = 0$ as can be seen by extracting the coefficients a, b, and c as in

$$\begin{aligned}\left(P_{i,j}-P_x\right)^2+\left(P_{i,j}-P_y\right)^2-\left(C_x+C_y\right)^2 &= P_{i,j}^2-2P_{i,j}P_x+P_x^2+P_{i,j}^2-2P_{i,j}P_y+P_y^2-\left(C_x+C_y\right)^2\\ &= 2P_{i,j}^2-2P_{i,j}P_x-2P_{i,j}P_y+P_x^2+P_y^2-\left(C_x+C_y\right)^2\\ &= \underbrace{2}_{a}P_{i,j}^2\underbrace{-2\left(P_x+P_y\right)}_{b}P_{i,j}+\underbrace{P_x^2+P_y^2-\left(C_x+C_y\right)^2}_{c}. \end{aligned} \quad (5.8)$$

By using the formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (5.9)$$

the two solutions of the quadratic equation can be calculated as

$$\begin{aligned}P_{i,j_{1,2}} &= \frac{-\left(-2\left(P_x+P_y\right)\right)\pm\sqrt{\left(-2\left(P_x+P_y\right)\right)^2-4\cdot2\left(P_x^2+P_y^2-\left(C_x+C_y\right)^2\right)}}{2\cdot2} \quad \text{by (5.8)(5.9)}\\ &= \frac{2\left(P_x+P_y\right)\pm\sqrt{4\left(P_x+P_y\right)^2-4\cdot2\left(P_x^2+P_y^2-\left(C_x+C_y\right)^2\right)}}{2\cdot2}\\ &= \frac{P_x+P_y\pm\sqrt{\left(P_x+P_y\right)^2-2\left(P_x^2+P_y^2-\left(C_x+C_y\right)^2\right)}}{2}\\ &= \frac{P_x+P_y\pm\sqrt{P_x^2+P_y^2+2P_xP_y-2P_x^2-2P_y^2+2\left(C_x+C_y\right)^2}}{2}\\ &= \frac{P_x+P_y\pm\sqrt{2P_xP_y-P_x^2-P_y^2+2\left(C_x+C_y\right)^2}}{2}\\ &= \frac{P_x+P_y\pm\sqrt{2\left(C_x+C_y\right)^2-\left(P_x-P_y\right)^2}}{2}\end{aligned}$$

by consistently choosing $+$ or $-$ for $\pm$. As long as the term under the square root *(discriminant)* is not zero (only one solution), or negative (no valid solution), the two solutions to the equation P_{i,j_1} and P_{i,j_2} will be found. But if there is only a single solution to the equation then it will not be greater than both neighboring potentials P_x and P_y which is a requirement, because the arrival time of the wavefront (e.g. the potential) has to increase. And if the discriminant is negative

the computation in $\mathbb{R}$ will result in a Not a Number (NaN).[2] The alternative and working solution for the potential

$$P_{i,j} = \begin{cases} \frac{P_x + P_y + \sqrt{2\left(C_x + C_y\right)^2 - \left(P_x - P_y\right)^2}}{2} & \text{if } \left(C_x + C_y\right) > \left|P_x - P_y\right| \\ \min\left\{P_x, P_y\right\} + \left(C_x + C_y\right) & \text{else} \end{cases}, \tag{5.10}$$

$$\text{with } P_{i,j} > P_x \geq P_y,$$

is similar to the one given by Kimmel and Sethian [42] and Zhao [95]. It either computes the larger root of the quadratic equation, to guarantee an increasing arrival time of the wavefront, or a backup solution if the discriminant would be negative or zero. To see that this is a working condition for a valid solution it is useful to analyze the different cases.

Larger root case with $P_x = P_y$ – Here the equation

$$\frac{P_x + P_y + \sqrt{2\left(C_x + C_y\right)^2 - \left(P_x - P_y\right)^2}}{2} = P_x + \underbrace{\frac{\sqrt{2\left(C_x + C_y\right)^2 - 0}}{2}}_{m}$$

holds and the term m must be greater than zero to find a valid potential. Thus, $\frac{1}{2}\sqrt{2\left(C_x + C_y\right)^2} > 0$ which implies $\left(C_x + C_y\right)^2 > 0$. This is already the case by enforcing costs greater than zero during the cost computation phase.

Larger root case with $P_x \neq P_y$ – There are again two cases.

Case 1: $2\left(C_x + C_y\right)^2 = \left(P_x - P_y\right)^2$ – In this case the discriminant is zero and the quadratic equation has only one solution. The conditional in Equation 5.10 asserts that this case is never chosen, because it would not give a potential greater than the neighboring potentials.

Case 2: $2\left(C_x + C_y\right)^2 < \left(P_x - P_y\right)^2$ – Here the discriminant is negative, but this case is also ruled out by the conditional.

2 A similar problem of NaNs appearing in the solution range could not be mitigated in the attempt to directly implement the methods mentioned in Continuum Crowds [81] and March of the Froblins [65]. These invalid values made it impossible to derive optimal paths. The problem could be partially corrected by keeping the old value until a valid solution could be found, but this caused jumps in the potential which resulted in visible straight edges in the gradient field (often emerging near doorways at the corners of walls). This caused agents to move in unnatural zigzag patterns and form up in lines instead of taking the shortest route. Furthermore, the example code provided in [65] contains unused and uninitialized variables which is detrimental for understanding the technique. So far it is unclear whether all of this is a limitation of the discretization schemes given in these works or if it was an error in the attempted implementation for this thesis. The authors of [47] also report problems with artifacts, although it is unclear whether they are of the same nature.

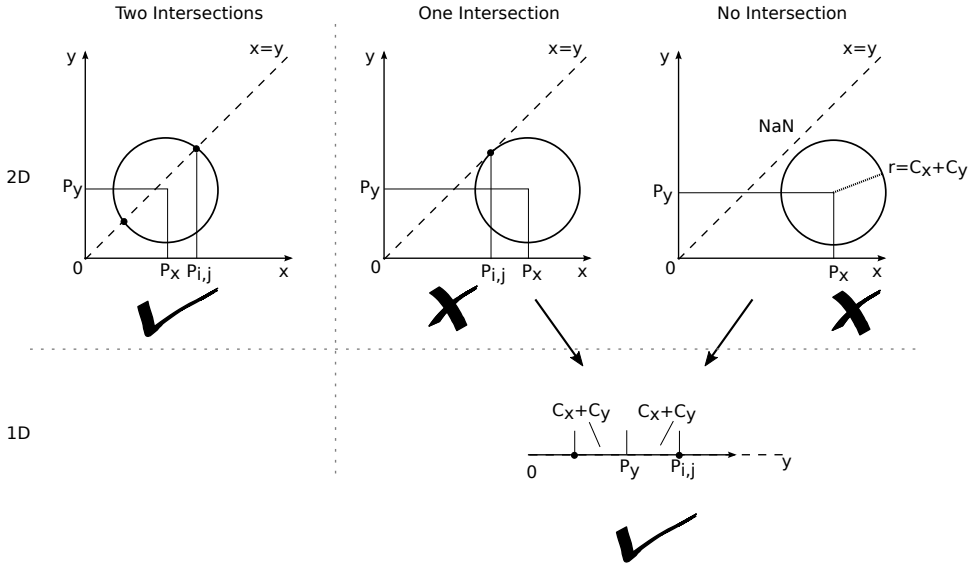


Figure 27: Circle-line intersection.

By looking at Figure 27 it is easier to understand which solution is chosen and why only the larger solution to the quadratic equation is accepted. The equation of a circle centered at (x_0, y_o) with radius r is

$$(x - x_0)^2 + (y - y_0)^2 = r^2.$$

Hence, geometrically Equation 5.7 can be seen as the intersection of the straight line $x = y$ with the circle of radius $r = C_x + C_y$ centered at (P_x, P_y). If the circle intersects with the line at two locations, then the larger solution is used, as depicted on the left side. The smaller solution would be less than the neighboring potentials and therefore it can not be considered. If there is only one intersection, as in the center image, then the potential would also be too small. The case to the right which would result in a NaN is also invalid. The backup value is the greater solution to the intersection done in a single dimension for the smaller of the two neighboring potentials, as shown in the lower section of the figure for the y candidate. The equation for the 1D sphere is

$$\begin{aligned}(x - x_0)^2 &= r^2,\\ \underbrace{1}_{a} x^2 \underbrace{-2x_0}_{b} x + \underbrace{x_0^2 - r^2}_{c} &= 0.\end{aligned}$$

Using Equation 5.9 the solutions to the quadratic equation for the 1D sphere become

$$x_{1,2} = x_0 \pm r,$$

of which only the greater one is used. [3]

3 In [42, 95] the general procedure for deriving the solution in n dimensions is provided, but no clear explanation of why the backup solution works is given. Maybe it is based on the smaller neighbor potential because the alternative would be an overestimation.

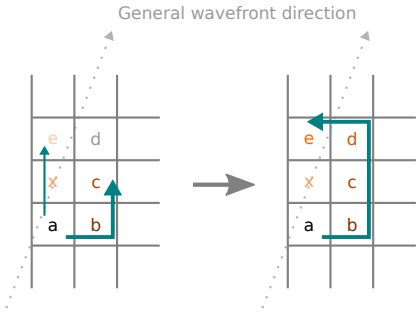


Figure 28: Subsequent potential update.

When updating the $P_{i,j}$ of a grid cell, the new solution $P_{i,j}^{k-1'}$ is only used if it is less than the current one $P_{i,j}^{k-1}$, as in $P_{i,j}^{k} = \min\left\{P_{i,j}^{k-1}, P_{i,j}^{k-1'}\right\}$ for the update step k. This is required, because the solution is always greater than the potentials it is based on and reasonably good solutions must be preserved in the cells. Else, all potentials would grow indefinitely based on the surrounding potentials. Still, it allows cells to acquire a smaller potential after they have been already updated. Situations in which a small potential has to propagate over several steps and cells to reach a previously updated cell are covered, as depicted in Figure 28. In this simplified example the wavefront propagates in the general direction from the lower left to the upper right, as indicated by the dotted arrow. Cell x features an extraordinary high cost. The potentials are illustrated in tones of orange with increasing brightness. When cell e gets updated for the first time in two steps through the cells $\{a, x\}$ it is assigned a high potential (bright orange). The value in d has not been determined yet. As soon as the update through the cells $\{a, b, c, d\}$ reaches it, the unnecessary large potential can be replaced by a lower one (darker orange). Thus, the wavefront moves faster around the costly obstacle x. Agents will later circumvent it in the opposite direction. Thicker teal arrows indicate faster wavefront propagation speed, due to lower cost.

5.6.2 *Cell Update Order*

Now that the difference scheme has been established, the cell update order can be discussed. Additional to working in an upwind manner, the OUM is an ordered method because of how it updates individual cells in a specific order.

One variant, the Fast Marching Method (FMM) by Sethian [63] defines an update order that allows to approximate the solution to the eikonal equation on a grid. It works as follows

1. Define *Alive* to be the set of goal cells and assign a potential of zero to them. Define *NarrowBand* to be the set of the 4-connected neighbors of the cells in Alive, which are not themselves in Alive.

Define *FarAway* to be the set of all other grid points and assign an infinite potential to them.

2. Update the potential in the cells of NarrowBand.

3. Find the cell c in NarrowBand with the lowest potential. Assign c to Alive. Assign all its neighbors that are not in Alive into NarrowBand

4. Recompute the potential of all neighbors of c that are not in Alive.

5. Repeat from step 2 until all cells are contained in Alive.

Thus, starting from the goal cells with zero potential the narrow band will march forward over the grid until all cells have been visited. But to find the cell with the lowest potential, a search over the cells in NarrowBand, or some ordered data structure is required. Also only a single cell is updated at each step in a sequential fashion, which makes this algorithm unsuitable for running on the GPU.

The Fast Iterative Method (FIM) by Jeong and Whitaker [34] is a related algorithm that can use the parallelism provided by the GPU architecture. It combines the idea of FMM to selectively update a single cell at a time, with just updating all cells of the grid until the potentials have converged to the solution as done by Rouy and Tourin [59]. FIM's GPU variant processes blocks (tiles) of cells at the same time. The procedure is as follows

1. Group neighboring cells into tiles (e.g. square subregions on a 2D domain). Assign all tiles that contain cells inside goal areas into the active list L.

2. Update each tile t in L

   a) Update all cells in t for n times (*inner loop*). During the last update step compute the difference between the current and the new potential. If the difference is below some threshold ε, mark the cell as converged.

   b) Perform a reduction over all cells in t, determining whether t has converged. If a single cell in t has not converged yet, then the whole tile is considered as not converged.

3. Check each tile t in L

   a) If t has converged, update all 4-connected neighbors of t as in step 2 with $n = 1$ (only one update), to find out if new information is available to the neighbors.

4. Set L to contain all non-converged tiles. Repeat from step 2 for as long as L is not empty (*outer loop*).

The list of active tiles L is similar to a narrow band of coarse granularity. The number of inner loop iterations n must not be too small, else the information inside a tile might not spread over the tile in a single outer step. It must also not be too larger either, or the inner loop will unnecessarily spend GPU cycles on a tile that is not changing anymore. The structure of each tile also plays a role here. If the tile features complicated walls, the potential might require many steps to converge. If it is just a plain map, less iterations are sufficient. The threshold ε must be similarly chosen to allow accurate solutions without wasting time on computations that don't improve the result in a reasonable way.

March of the Froblins [65] uses HLSL shader programs for their implementation, [33] applies OpenGL together with Cg, in [34] and [35] the authors use CUDA. The originators of the algorithm report speedups around 100 for GPU-based FIM over CPU-based FMM [35]. Detailed comparisons for different applications and with other similar techniques are given in [34].

March of the Froblins does not perform the reductions and management of an active tile list in order to safe computation. Instead an estimation of the required number of updates is used, and all tiles are updated until the solution has converged. The implementation in [35] uses the CPU to manage the list of active tiles. Although the authors mention that the amount of data transmitted from the GPU to the CPU and back is only small, the whole process still requires synchronization between the devices. In an experiment a `clFinish` was introduced into the outer potential update step of the program developed for this thesis. The resulting synchronization between the GPU and the CPU caused the execution time for each whole simulation step to double from $\sim 30\,\text{ms}$ to $\sim 60\,\text{ms}$ on a 256×256 grid. This overhead is of lower importance if larger grid sizes are used. For applications in medicine that work on data provided by computer tomography like [35], it is necessary to only terminate the computation after the best solution has been found. But if no exact solution is required and the number of update steps can be estimated, a fixed number of outer steps can lead to shorter execution times with reasonably good results. Thus it has been decided not to use any CPU sided management for the list of active tiles in this thesis. The required number of steps depends on the complexity and size of the map. Complicated wall structures and large maps require more iterations. Open areas with no walls require fewer steps.

As shown in Figure 21, the Potential Calculator uses a preinitialized buffer called *Destinations* which contains a potential of zero in cells belonging to the set of goal cells and infinity in all other cells. This buffer is never changed. Its content is copied to the working buffer *Potentials* before the potential computation starts for the current simulation step. The class then invokes a kernel on Potentials for each iteration of the

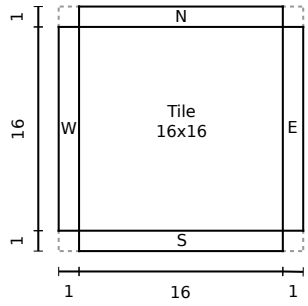


Figure 29: Potentials in local memory.

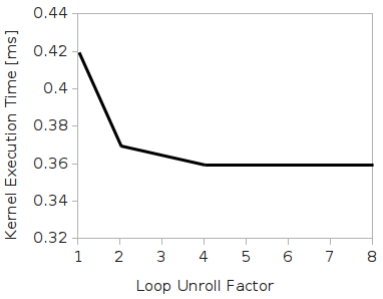


Figure 30: Loop unrolling and kernel execution times.

outer loop. It processes the grid based on a 2D NDRange. The cell tiles nicely maps to OpenCL work-groups and the potential data can be placed into local memory for rapid updates in the inner loop. Again, the four potential fields for the four agent groups (*Group0*, *Group1*, *Group2*, *Group3*) are processed in parallel by using operations on the `float4` data type as illustrated in the figure. Two update methods have been implemented, *Update All* and *Selective Update*, and the user can select in the GUI controls which update method is applied.

5.6.2.1 *Update All*

Update All processes all tiles during every outer step, just like March of the Froblins, in a brute force manner. The number of outer and inner steps (n) can be specified in the GUI.

In the implementation of [35] the potentials are arranged in an organized manner to allow coalesced access to global memory. In the program for this thesis all the `float4`s are laid out in a sequential fashion along the x-direction of the grid. During the fetch to local memory, as explained below, each quarter-wavefront of a 256 elements work-group loads $16 \cdot 4 \cdot 4 = 256$ coalesced bytes of `float4`s. This seems to be a large enough block of sequential data to support coalescing and caching. Hence, the specialized data layout in [35] has not been tested.

The kernel loads the potentials into local memory for subsequent processing inside the inner loop. Here the layout of the data is especially important because it will be accessed many times. Each work-item requires the potential of its cell and the four neighbor cells. If the work-group contains $16 \times 16 = 256$ items, the 256 inner potentials plus the $4 \cdot 16$ potentials along the northern, eastern, southern, and western border of the tile must be loaded into local scratch memory as shown in Figure 29. This results in a total of $256 + 4 \cdot 16 = 320$ `float4` values or 5120 bytes of data. Each work-item loads the values

corresponding to its index in the NDRange. The work-items at the tile edges additionally load the border potentials. The `float4`s

$$\left(P_{Group0_{i,j}}, P_{Group1_{i,j}}, P_{Group2_{i,j}}, P_{Group3_{i,j}}\right)$$

are stored as `float2`s inside the local memory, first all the potentials of Groupo and Group1 (components x and y of the `float4`*s* paired as `float2`s), then those of Group2 and Group3 (components z and w) as in

$$\begin{aligned}&\left(P_{Group0_{0,0}}, P_{Group1_{0,0}}\right), \left(P_{Group0_{1,0}}, P_{Group1_{1,0}}\right) \ldots \left(P_{Group0_{s,s}}, P_{Group1_{s,s}}\right),\\ &\left(P_{Group2_{0,0}}, P_{Group3_{0,0}}\right), \left(P_{Group2_{1,0}}, P_{Group3_{1,0}}\right) \ldots \left(P_{Group2_{s,s}}, P_{Group3_{s,s}}\right),\end{aligned}$$

where s is the side length of a tile. The reason is that each Stream Core can load 32 consecutive `float`*s* during a cycle, as mentioned in Section 3.2.2. Thus, the 16 work-items of a quarter-wavefront can load 16 `float2`s in a single cycle without any channel conflicts. In two consecutive instructions the 32 `float2`s, that store the potential of 16 cells, can be loaded.

The kernel then updates the potentials of all the cells of the tile inside the scratch memory for a user defined number of inner steps by the method given in Section 5.6.1. The inner update loop that performs the computation is unrolled into two steps. Hence, it performs two updates per inner step. Different unroll factors have been tested. Figure 30 shows the execution times for the potential kernel with different unroll factors (measured for factors one, two, four, and eight). Unrolling the loop to perform two updates per step provides a good performance increase. But the gain of higher values is only marginal. Additionally, the value two still allows detailed inner choices for experimentation (e.g. is a tile completely updated in k steps?). This is the reason why two has been chosen as the factor in the implementation related to this thesis.

5.6.2.2 *Selective Update*

The Selective Update works similar to Update All in how it executes inner loops, but it applies a more sophisticated algorithm during the outer loop steps. As indicated in Figure 21 it uses two buffers that track the tile states *Tile States In* and *Tile States Out*. Both contain integers for each tile, storing the state of the tile. There are three possible values for the integers, representing one of the states

- *Update*,
- *Sleep*,
- *IsConverged*.

Listing 5: Reduction function for the logical AND operation.

```
int reduceAnd(local int* booleans,
              size_t     localId,
              size_t     localSize)
{
    for (int offset = localSize / 2; offset > 0; offset >>= 1)
    {
        if (localId < offset)
        {
            int isTrueA = booleans[localId         ];
            int isTrueB = booleans[localId + offset];

            booleans[localId] = isTrueA && isTrueB;
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    return booleans[0];
}
```

Update means that the tile must be updated in the next outer iteration. Sleep says that it should sleep and no update is to be performed. IsConverged indicates that it converged during the previous step. This third case is required in order to notify neighboring tiles about new information. Without it there would be no way to consider the case where the convergence happened during a single step. For example, a tile is woken up from Sleep, updates and immediately converges again. Without this third state the transition would originate in Sleep and end in Sleep. Here, it ends in IsConverged, and a neighboring tile can wake up to process the new information.

The Crowd Calculator prepares the input buffer with initial tile states. The entries in Tile States In are initialized to Update if there is a goal cell inside the tile, and Sleep else. This is determined by performing a reduction over the cells in the tile. The implementation of the parallel reduction is based on [13] and uses local memory. Listing 5 provides the function used by the kernel for reducing operands to the logical *AND* operation. The code processes the boolean values stored inside the local array `booleans`, the local ID and the size of the array and work-group are also provided. In the first iteration half of the participating work-items compute the result of two operands each. With each subsequent iteration less and less work-items participate in the reduction step. It is interesting to note, that all work-items, including the ones not participating in the computation, execute the `barrier` instruction. This is necessary as explained earlier. The `barrier` ensures that all items have written their results into the array before the

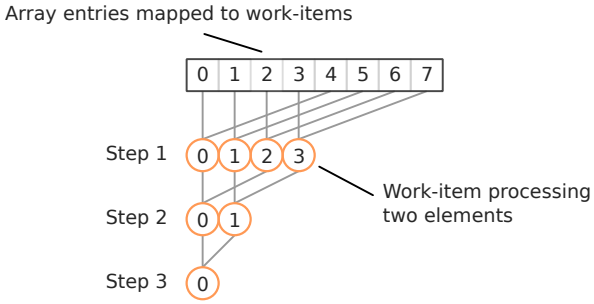


Figure 31: Relation of array elements and work-items in the logical AND reduction.

next iteration. Finally, the result resides in the first array entry and can be returned. The reduction requires $\log n$ steps where n is the number of entries in the array. The program uses a similar function for the reduction of the logical *OR* operation. To allow the parallel reduction operation to work, an operation must be associative (to allow pairings) and commutative (to allow parallel computation for the pairs), as mentioned in [13]. Figure 31 illustrates how the work-items relate to array entries, and which items participate in each step for an example with eight elements and items. The result can be computed in three steps. By limiting the computation to work-items with low local ID the CU can skip the computation for whole wavefronts that don't participate.

During each outer loop step the kernel then uses the states in Tile States In to determine how it has to process each tile. For this the first work-item in a work-group that processes a tile retrieves the current state α into local memory. The first item is identified by its local ID equal to zero. Each work-item in that work-group then continues to process its cell depending on the state. As mentioned earlier, if all 16 work-items in a quarter-wavefront access the same address the LDS can distribute the state value in a single cycle. The tile transitions into the next state β and the first work-item in the group writes that state into Tile States Out. These state transitions are subsequently denoted as $\alpha \rightarrow \beta$. The first work-item in a group also performs the neighbor checks mentioned below. In the following description of state transitions, a *Large Step* means that multiple inner update steps are performed as explained for Update All (Section 5.6.2.1). *Small Step* is similar but computes only a single inner update iteration. $\Delta P_{i,j} = \left| P^n_{i,j} - P^{n-1}_{i,j} \right|$ is the absolute difference in potential between the last and previous update step in either a Large Step or a Small Step. The neighboring tiles are $Tiles_{NESW} = \{Tile_N, Tile_E, Tile_S, Tile_W\}$, where $Tile_d$ is the neighbor of the current tile in direction d. For each outer iteration the algorithm retrieves the current state α, and performs actions and transitions depending on what state it represents

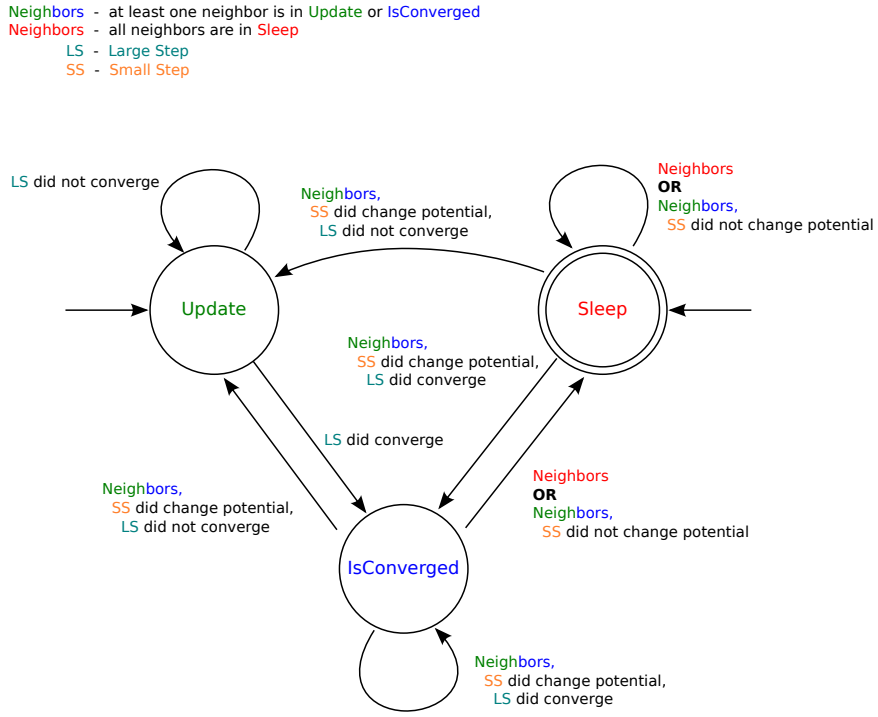


Figure 32: Tile state diagram.

Update – Perform a Large Step. Compute $\Delta P_{i,j}$. If $\Delta P_{i,j} \leq \varepsilon$ or $P^n_{i,j} = \infty$, the cell has converged. Determine the convergence of the tile by reducing the result of all work-items in the work-group in local memory. If all cells have converged then $Update \rightarrow IsConverged$ else $Update \rightarrow Update$.

Sleep – If all tiles in $Tiles_{NESW}$ are in Sleep then $Sleep \rightarrow Sleep$, because nothing has changed since this tile was set to sleep.

Else, find out whether the tile really needs to be updated with the new information in the neighboring tiles. Perform a Small Step. If $\Delta P_{i,j} \leq \varepsilon$ then the data in the neighboring tile did not influence this tile and $Sleep \rightarrow Sleep$.

Else, the information flowing in from a neighboring tile actually caused a change. Perform a Large Step. If all cells converged then $Sleep \rightarrow IsConverged$, else $Sleep \rightarrow Update$.

IsConverged – Same as for Sleep, but with $\alpha = IsConverged$ in all state transitions.

In this implementation $\varepsilon = 0.001$. The work-groups have to write state changes without changing the current input states for other work-groups. While logically all tiles are processed in parallel, some are updated sequentially in praxis. During an outer update step, a work-group must not change its input state before related tiles have read it. Hence, the tile state buffers are managed in a ping-pong scheme and swapped after each outer iteration. The work-groups can

write their new states at will, without influencing other work-groups. Additionally, four potential fields are computed simultaneously (one for each agent group) but only one tile state is managed for four overlapping tiles. During the checks, even if the situation of only one of the tiles diverts, all four tiles are treated equally. For instance, if the potential in one of the four potential fields has not converged in a tile yet, then the potentials for all fields in that tile are updated again in parallel utilizing the four ALUs inside the Stream Cores.

Figure 32 gives an overview of the states and the transitions involved in the algorithm. Update and Sleep are the start states based on how the tile is initialized. Sleep is the state that all tiles reside in after the potential field has been computed, if the number of outer iterations has been sufficiently defined by the user. The colors make it easier to understand the actions and neighbor states. The condition for most transitions is based on the state of neighboring tiles. Only the Update state has transitions, that don't depend on any neighbor state, leading out of it. Subsequent actions and their results are indented in the transition conditions to indicate the decision process involved. The *OR* means that one of the two possible conditions can lead to the transition. "*Neighbors*" represents the state of the neighboring tiles, "*did change potential*" indicates that the test changed the potential enough to warrant a transition, "*did converge*" points out that all cells in the tile passed the convergence test.

The algorithm calculates the potential field with the FIM on the GPU without any synchronization with the CPU. With the exception of the first work-item deviating in some cases, and the reductions, all work-items on a CU follow the same control flow. In cases where the tile is not to be updated and all neighbors are sleeping no additional computation is performed. Thus, the hardware can schedule another tile to be processed on the CU. This is often the case because in the beginning many tiles contain only infinite potentials and stay in Sleep, and later on many converged tiles will remain in Sleep. Section 7.1.2 explains how the tile state changes can be visualized. This Selective Update scheme saves GPU cycles and leads to higher performance than the rather brute force Update All method. Experiments showed that larger map sizes like 1024×1024 allow greater improvements than smaller maps with only 256×256 cells, resulting in speedups of ~ 1.5 versus ~ 1.25. Additionally, by managing the algorithm solely with the GPU, the CPU remains free to perform other tasks after filling the OpenCL command-queue.

5.7 GRADIENT

To store a 2D gradient vector for each of the four agent groups, eight `float` values are required. The *Gradient Calculator* manages two OpenCL images, *Gradients01* and *Gradients23*, to store two vectors

for each entry, as shown in Figure 21. The reason is that OpenCL (and OpenGL) supports at most four components per image (texture) element. With OpenCL buffers `float8`s could be used. The images are OpenGL textures used for the visualization of the gradient field and shared with the OpenCL context. Another reason why images are used is that the gradients are sampled with bilinear filtering (Equation 5.5) during agent movement computations and visualization of the gradient field, which is supported by the underlying hardware through OpenGL and OpenCL samplers for textures and images. Because bilinear interpolation just computes a linear combination of the values per component, the two 2D vectors inside a `float4` are correctly filtered even though the filtering is performed on the whole `float4` variables.

The gradient needs to be computed in the upwind direction, in order to allow the agents to move against the propagation direction of the wavefront. The Gradient Calculator uses a 1D NDRange for its kernel invocation. The kernel computes two gradient vectors concurrently. First the potential of the current cell and the neighbors is retrieved. Then three vectors are prepared with the potentials required for the gradients

$$Positive^{01}_{i,j} = \left(P^0_{i+1,j}, P^0_{i,j+1}, P^1_{i+1,j}, P^1_{i,j+1}\right), \tag{5.11}$$

$$Negative^{01}_{i,j} = \left(P^0_{i-1,j}, P^0_{i,j-1}, P^1_{i-1,j}, P^1_{i,j-1}\right), \tag{5.12}$$

$$Center^{01}_{i,j} = \left(P^0_{i,j}, P^0_{i,j}, P^1_{i,j}, P^1_{i,j}\right), \tag{5.13}$$

where the superscript indicates the relation to the agent groups. The backward and forward difference quotients in the x- and y-direction are computed in parallel for Group0 and Group1 as

$$\begin{aligned} DifferenceQuotient^{01-}_{i,j} &= Center^{01}_{i,j} - Negative^{01}_{i,j} \\ &= \left(P^0_{i,j} - P^0_{i-1,j}, P^0_{i,j} - P^0_{i,j-1}, P^1_{i,j} - P^1_{i-1,j}, P^1_{i,j} - P^1_{i,j-1}\right) \\ &\qquad \text{by (5.13)(5.12),} \\ DifferenceQuotient^{01+}_{i,j} &= Positive^{01}_{i,j} - Center^{01}_{i,j} \\ &= \left(P^0_{i+1,j} - P^0_{i,j}, P^0_{i,j+1} - P^0_{i,j}, P^1_{i+1,j} - P^1_{i,j}, P^1_{i,j+1} - P^1_{i,j}\right) \\ &\qquad \text{by (5.11)(5.13),} \end{aligned}$$

with component-wise operations. Because the grid size is considered to be one, no division is required to get to the difference quotients. Now it needs to be determined which difference quotients are required for the upwind direction. There are two choices per x- and y-direction. For a single dimension three classes of cases can be given. Figure 33 shows an illustration of these relations between neighboring potentials in a single dimension. The lines indicate relative differences in potential.

In the *Common* class on the left are potentials where $P_{i,j}$ is either equal to both neighboring potentials, to one potential, or in the middle

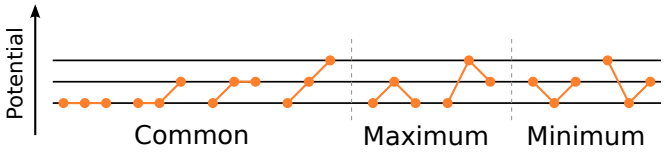


Figure 33: Three classes of neighboring potential configurations in one dimension.

of both. By checking which of the two neighboring potentials is smaller, the correct difference quotient can be found. It will always reside on the upwind (downhill) side. If $P_{i,j}$ is equal to the lower potential, as in the second case from the left, then the difference quotient will be zero, which is correct. In the case on the far left where both neighboring potentials are equal it does not matter which side is chosen. Both will result in a difference quotient of zero.

The *Maximum* case in the middle features a central potential that is larger than either of the neighboring potentials. This edge case where a potential peak or crest exists is also reasonably covered by the selection performed in the Common case. The crest appears where wavefronts coming from one or several goal areas meet. To be more accurate, it would be necessary to also check the cost in the case on the right, because the cost also decides from which side the wavefront reached the central grid cell. But since this is an edge case and agents near potential crests appear to choose one of the two directions randomly, no extra check is performed. In the experiments the potential crests also moved around quite a bit in each simulation step, therefore the implications of this compromise between performance and accuracy are small.

In the *Minimum* class on the right $P_{i,j}$ is smaller than any of the two neighboring potentials. Because the first arrival time (potential) of the wavefront increases with distance to the goal areas, the central potential must be inside a destination cell. This edge case is also treated identical to the Common class, because goal areas that are only one cell wide are problematic anyway due to how the agent movement is performed below (the goal cell might not be hit by the sampling, to detect that the agent is touching the goal area). This class should not occur in a reasonably well designed map.

All other potential configurations are isomorphic to the ones in these classes and therefore they also fit into the explanation above. The gradient is then computed as

$$Gradient^{01}_{i,j_s} = \begin{cases} DifferenceQuotient^{01-}_{i,j_s} & \text{if } Negative^{01}_{i,j_s} \leq Positive^{01}_{i,j_s} \\ DifferenceQuotient^{01+}_{i,j_s} & \text{else} \end{cases},$$

with the component based `select` statement, where the subscript s indicates one of the components $\{x, y, z, w\}$ of the four-element vector. A final check determines if both neighboring potentials in a single

dimension are infinite. In this case the component of the gradient for that direction is set to zero, instead of the NaN resulting from the subtraction between infinite potentials. The whole procedure is then repeated to compute the gradients of Group2 and Group3 ($Gradient^{23}_{i,j}$).

6 AGENT MOVEMENT

After the navigation pipeline has successfully derived the gradient field for all four agent groups, the agents can finally use that information to plot their paths over the map. The movement phase is separated into two steps, the update of agent position, orientation, speed, and ID, and the resolution of collisions occurring between agents. Figure 34 shows the involved components. The *Agent Mover* manages all the individual buffers storing the agent properties, that have been depicted as a single buffer in the top right corner of Figure 21. Additionally to the previously illustrated properties they also store the *MaximumSpeeds* and *TurnRates* of each agent. The agent *Positions* are randomized at application start to any location inside the domain, that has a distance of at least five units to the borders of the map. The seed for the random number generator can be set in the configuration file. If it is less than zero, the system time is used as seed. Initially all agent *Directions* face to the east ($(1, 0)$). The MaximumSpeeds are randomized inside $[4, 12]$ (inclusive), and the current *Speeds* are initialized to the maximum speeds. The minimum and maximum boundary for the randomly generated *Radii* can be specified in the configuration file. Agents are evenly distributed between agent groups in the initialization phase of the program, and the group assignment is stored in the two least significant bits of the values in *IDs*. The idea is to save memory by allowing further properties to be stored in the higher bits of an ID. For instance, the agents could be further classified for different looks during visualization, but this has not been implemented yet. The TurnRates are currently set to 0.15. This is not an angle but a length-related value because of how the agent's direction update step works with offset vectors.

6.1 META MOVEMENT

The meta movement scheme, which can be changed at any time in the GUI, allows agents to perform actions that extend beyond simply updating their movement trajectories. This allows the user to model situations where the agents constantly leave the map area as is typical in evacuation scenarios (*Park*), re-plan after arriving at a goal and move to the new destination area (*Change Group*), or teleport from the goal area to a location where they can reenter the domain and continue to participate in the simulation (*Respawn*). The currently selected method is illustrated as *ExitMode* in Figure 34.

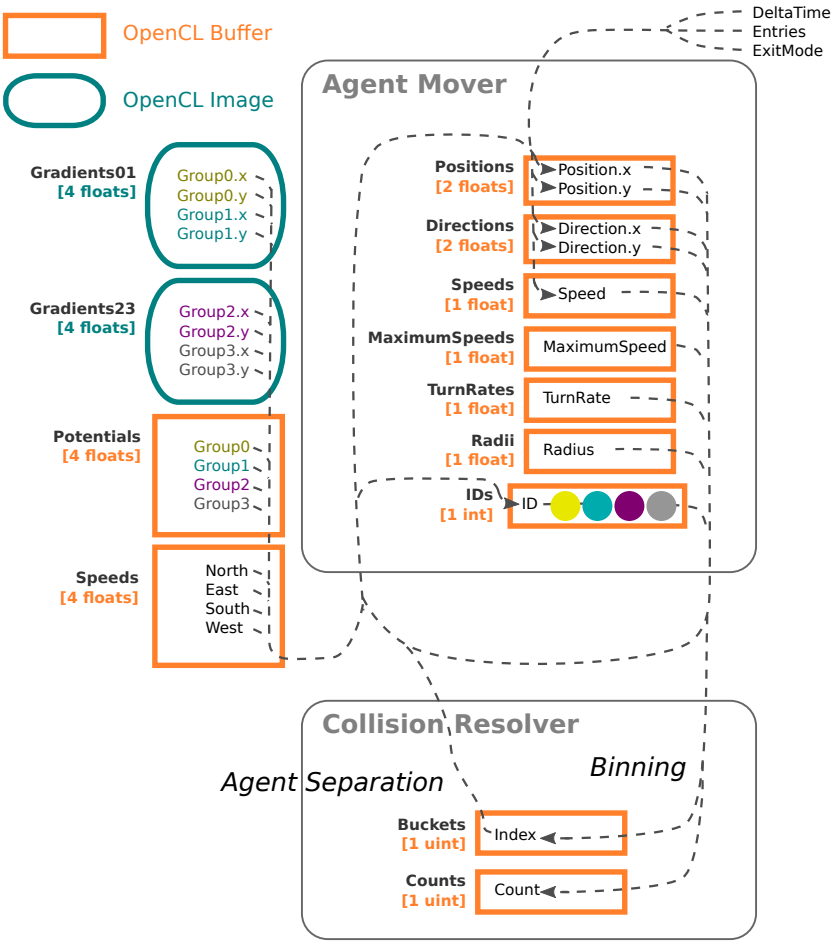


Figure 34: The agent movement phase.

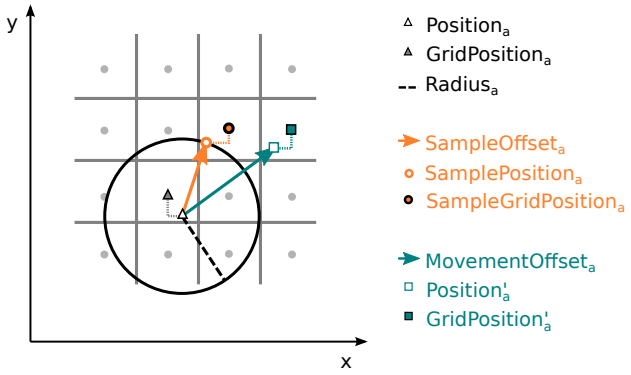


Figure 35: Sample positions during agent movement update.

6.1.1 *Park*

Each agent can be situated inside the domain and active, or parked outside of the area, indicated by a negative position. In the parking position they are invisible and disabled (their trajectories are not updated). The program features counters in the GUI that list how many agents of each group are currently active and parked. These counters are incremented or decremented with atomic operations to reflect the current state of the agents. If parking is enabled, the agents stream will continuously drain the map of active agents. This can be used to simulate evacuation scenarios to see how fast all agents can leave the map through the given exit zones. Additionally, agents may start inside a wall due to how the initial position is selected randomly. These agents and those that end up there erroneously are placed into the parking position. The situation is detected by sampling the potential at the agent's $GridPosition_a$, as shown in Figure 35. If the potential at the triangle marking the grid position is infinite, the agent is inside a wall. Extra care has been taken so that agents don't end up inside walls during the simulation, as mentioned below, and so far no occasions have been witnessed.

6.1.2 *Change Group*

Similar to the Froblins in the demo of [65], the agents can switch groups if this meta movement scheme is activated. After arriving at their destination the agents continue into the new direction to the goal area of their new group. Currently the agent groups are changed in a circular fashion, but this could be easily adapted to a random algorithm or other, more advanced logics. This scheme provides a means to simulate situations where agents have to visit predefined locations in a certain endless sequence, thereby continuously moving across the map.

6.1.3 *Respawn*

When the movement scheme is set to Respawn, agents that arrive at their destination will first be parked by the kernel program. In the next simulation step they respawn together with all the other parked agents. Hence, agents that have been parked because they were situated inside walls will be revived. The *Entries* (Figure 34) are a axis-aligned, rectangular areas, that can be defined per agent group in the configuration file. The agents spawn in a pseudo-random location

inside that area based on the global ID of the work-item updating the agent, as in

$$\begin{aligned} EntryWidth &= Entry_x^{\urcorner} - Entry_x^{\llcorner} + 1, \\ EntryHeight &= Entry_y^{\urcorner} - Entry_y^{\llcorner} + 1, \\ Index &= GlobalID \bmod (EntryWidth \cdot EntryHeight), \\ SpawnPosition_a &= Entry^{\llcorner} + \left(Index \bmod EntryWidth, \frac{Index}{EntryWidth} \right), \end{aligned}$$

where $Entry_x^{\urcorner}$ denotes the x-component of the entry's upper right corner, $Entry_x^{\llcorner}$ is the same for the lower left corner, and similar for the y-direction. This selects an $Index$ into the whole number of available grid locations for this entry, $EntryWidth \cdot EntryHeight$. An agent will always spawn at the same location inside the spawn entry area.

6.2 MOVING THE AGENTS

The Agent Mover invokes a kernel based on a 1D NDRange to update all agents. By retrieving the current agent position the kernel can decide which further steps need to be taken. If the position is negative, the agent is currently parked and can be respawned as explained above. On the other hand, when the position is positive, the agent is active and must be further processed.

First a $SamplePosition_a$ is computed that is offset by the $SampleOffset_a$ from the agents center in the direction it is facing, as indicated in Figure 35. The distance is chosen to be the agent's radius. Any smaller offset would result in a sample that is affected by the agent's own density and velocity influence. Then the potential at the $SampleGridPosition_a$ is retrieved. The grid position of any floating-point position is determined by the component-wise rounding to negative infinity and conversion to integer. If the position lies outside of the grid, an infinite potential is retrieved. If the potential at the grid location is zero, the sample is situated inside the agent's goal area, and because the sample is taken at the agents boundary, the agent is said to have reached its destination. Based on this information the agent can be parked for deactivation or respawning, or be reassigned to a different agent group. It will be updated again during the next simulation step.

If the agent has not reached its goal yet, the kernel uses the gradient at $SamplePosition_a$ to compute a new agent position. The gradient is stored in the OpenCL images and can be filtered with bilinear interpolation (Equation 5.5), to get directions of higher fidelity at locations situated between grid node centers, as indicated in the figure. If $SamplePosition_a$ is outside of the domain, the null vector with both components equaling zero is returned.

The components of the filtered gradient vector can be NaN because of subtractions between infinite values during the interpolation process. This can happen at wall edges and is sometimes visible in the

interpolated visualization of the gradient as mentioned in Chapter 7. Therefore the gradient is checked for NaN and infinite values that indicate that it was sampled inside a wall. Furthermore, if both components of the gradient vector are zero, it was sampled from a location even further in the center of walls. These zeros have been set in Section 5.7 to make the inner cells of walls easier detectable. In the case when the gradient is sampled outside of the domain this is also treated like a wall. Another case where both components are zero is inside goal areas, but this situation has been handled above already. Thus, if the gradient contains NaNs, infinite values, or all zeros, the agent must be steered away from a wall, else, it would move into it and be parked automatically during the next simulation step. To prevent this the agent is forced to turn in place with zero speed. The opposite of the current movement direction becomes the new target direction and the turn rate is increased to lower the amount of time during which agents are facing walls. The condition for when the sample is inside a wall can be given as

$$\begin{aligned} IsWall_a = &(Gradient_x = \infty) \vee \left(Gradient_y = \infty\right) \\ &\vee (Gradient_x = NaN) \vee \left(Gradient_y = NaN\right) \\ &\vee \left(Gradient = \overrightarrow{0}\right), \end{aligned}$$

where $Gradient_x$ and $Gradient_y$ are the x- and y-component of the interpolated gradient vector $Gradient$, and $\overrightarrow{0}$ is the null vector.

If the agent is not confronted with a wall, the negative gradient is normalized and becomes the new target heading. The length of the gradient vector is of no interest. Thus, it is not important how steep the potential field at that location is, only the direction of the wavefront propagation is required. The $TargetDirection_a$ of agent a thus becomes

$$TargetDirection_a = \begin{cases} -Direction_a & \text{if } IsWall_a \\ -\frac{Gradient}{\|Gradient\|} & \text{else} \end{cases},$$

based on the agent's current $Direction_a$ and the $Gradient$ sampled. Next the dot product between the current and the intended direction is computed as

$$Cosine_a = Direction_a \cdot TargetDirection_a,$$

to get a measure of how far the agent intends to turn. If it wants to turn far or if it is confronted with a wall, the turn rate is increased by a factor of 4 as in

$$TurnRate'_a = \begin{cases} 4 \cdot TurnRate_a & \text{if } IsWall_a \text{ OR } Cosine_a < 0.6 \\ TurnRate_a & \text{else} \end{cases},$$

where $TurnRate'_a$ is a temporary agent turn rate. Figure 36 shows how the direction vectors relate. If the length of the difference vector

$$DirectionOffset_a = TargetDirection_a - Direction_a$$

is less than or equal to the $TurnRate'_a$, the $NewDirection_a$ becomes the $TargetDirection_a$, because the agent can turn towards it in a single simulation step, as illustrated on the far left. If that length is too large, as indicated by the red arrow in the center of Figure 36, an intermediate direction

$$Direction_a^{\measuredangle} = Direction_a + TurnRate'_a \frac{DirectionOffset_a}{\|DirectionOffset_a\|}$$

must be computed. If the $TargetDirection_a$ is close to the opposite of the current $Direction_a$, the computation of $Direction_a^{\measuredangle}$ might result in the null vector. This case can be detected by checking the $Cosine_a$. If it is smaller than some negative threshold (e.g. -0.8), then an alternative direction computation resulting in $Direction_a^{\curvearrowleft}$ is used. Here, the agent's rotation is based on a vector orthogonal to the current direction as in

$$\begin{aligned} Direction_a^{\perp} &= \left(-Direction_y, Direction_x\right), \\ Direction_a^{\curvearrowleft} &= Direction_a + \frac{Direction_a^{\perp} - Direction}{\|Direction_a^{\perp} - Direction\|} TurnRate'_a, \end{aligned}$$

as illustrated on the far right in Figure 36. Hence, all agents using this rotation scheme will turn towards the left. Finally, the new agent direction computation can be summarized with

$$IsTooFar = \|TargetDirection_a - Direction_a\| > TurnRate'_a$$

as

$$NewDirection_a = \begin{cases} \frac{Direction_a^{\curvearrowleft}}{\|Direction_a^{\curvearrowleft}\|} & \text{if } Cosine_a < -0.8 \\ \begin{cases} \frac{Direction_a^{\measuredangle}}{\|Direction_a^{\measuredangle}\|} & \text{if } IsTooFar \\ TargetDirection_a & \text{else} \end{cases} & \text{else} \end{cases}.$$

In all cases it is assured that the vector is of unit length. This turning scheme with offset vectors is used instead of rotations by angles to prevent the computation of costly trigonometric functions. $NewDirection_a$ is written to back into the corresponding agent property buffer.

Next the speed is computed. As mentioned earlier, the speed becomes zero if the agent is facing a wall, so that it can turn in place. If the agent is free to move, the anisotropic speed field is sampled at the $SampleGridPosition_a$ as shown in Figure 35. Again this is outside of the agent's own influence. The $NewDirection_a$ is then used to derive the speed. The speed field provides only speeds for one of the four

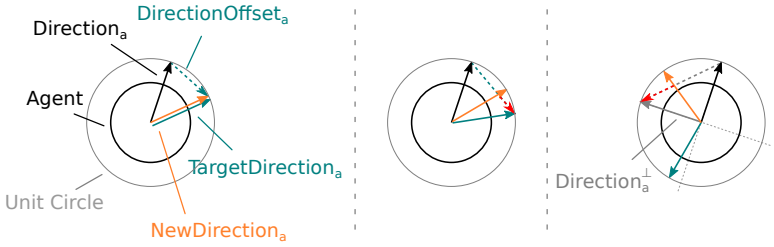


Figure 36: The three direction update cases.

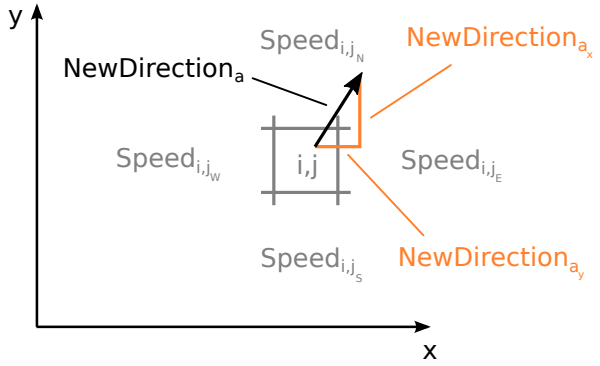


Figure 37: Anisotropic speed contribution to the agent speed.

directions north, east, south, and west, but $NewDirection_a$ most likely does not directly point into one of these headings. Therefore, the contribution of each direction to the actual speed must be determined. By checking if the sign of the x- and y-component of the direction vector is positive or negative, the contributing speed along each axis can be found. This is $Speed_{i,j_W}$ or $Speed_{i,j_E}$ for the x-, and $Speed_{i,j_N}$ or $Speed_{i,j_S}$ for the y-direction. $NewDirection_a$ is a vector of length one and its components enclose a right triangle as shown in Figure 37. With the relation

$$NewDirection_{a_x}^2 + NewDirection_{a_y}^2 = 1$$

coming from the *Pythagorean theorem*, the percentage of contribution of each component becomes known. They correctly add up to 1. The unbounded agent speed thus is

$$Speed_a^U = \begin{cases} 0 & \text{if } IsWall_a \\ NewDirection_{a_x}^2 Speed_{i,j_x} + NewDirection_{a_y}^2 Speed_{i,j_y} & \text{else} \end{cases},$$

where $Speed_{i,j_x}$ and $Speed_{i,j_y}$ are the anisotropic speeds related to the components of $NewDirection_a$. In the example in Figure 37 these are $Speed_{i,j_N}$ and $Speed_{i,j_E}$. The speed is then limited to the maximum speed for the agent $MaximumSpeed_a$ and the bounded agent speed becomes

$$Speed_a^B = \min\left(Speed_a^U, MaximumSpeed_a\right).$$

Now the temporary velocity

$$Velocity'_a = NewDirection_a Speed_a^B$$

is computed and with the *Euler method* the position

$$Position'_a = Position_a + DeltaTime \cdot Velocity_a$$

is found, based on the old position $Position_a$, the velocity, and the time step. This can be the time difference between the last two simulation steps, or some fixed value, as used in the implementation for this thesis, where 30 ms has been chosen for the simulation of 33 steps per second. With the risk of the agent ending up inside a wall, another check needs to be performed. The potential at the grid location related to $Position'_a$ is retrieved. If it is outside of the domain, infinity is returned. The position is illustrated as $GridPosition'_a$ in Figure 35. In the case where the potential is infinite, the agent would again step into a wall or outside of the domain. Thus, if this is the case, the current agent position is kept and the speed is set to zero. Else, the new values can be used. The boolean variable $IsNewWall_a$ is set to true if the potential at $GridPosition'_a$ is infinite. Finally, the new agent speed and position are

$$NewSpeed_a = \begin{cases} 0 & \text{if } IsNewWall_a \\ Speed_a^B & \text{else} \end{cases},$$

$$NewPosition_a = \begin{cases} Position_a & \text{if } IsNewWall_a \\ Position'_a & \text{else} \end{cases},$$

and can be stored in the corresponding property buffers.

This movement method approximates the agent trajectory with linear segments computed by the Euler method. If the time step is too large or the speed too high, the agent might move through other agents or even a wall in a single simulation step. With the given technique of *discrete* (*a posteriori*) collision detection, this can not be prevented. A solution would require *continuous* (*a priori*) collision detection, as explained in [92]. Hence, reasonable time steps and speeds must be used. This is also the reason why the implementation features a fixed time step of 30 ms, to get believable trajectories, even if the computation slows to a crawl for examples with large domains and many agents. In such cases, using the time difference between simulation steps in wall clock time, might cause large agent offsets and the problems mentioned above.

6.3 COLLISION RESOLUTION

During the agent position update by the Agent Mover extra care is taken not to place any agents inside walls. Additionally agents slow

down in areas of high density in order to minimize collisions. Still, agents may end up intersecting with each other. The visual artifacts can be easily identified by the human eye and the absence of rigid boundaries between agents lowers the quality of the simulation of agent trajectories. Therefore, a separate discrete collision detection and resolution phase is executed by the *Collision Resolver* depicted in Figure 34. For the collision resolution only intersections that go beyond an infinitely small touching point are of interest. Collisions where agents merely touch each other, need not be handled, because elastic collisions are not simulated, and inappropriate for many agent types. For experimentation purposes the collision resolution can be disabled in the GUI options.

6.3.1 *Collision Detection*

In order to know which agents violate the constraints of the simulation, pairs of agents that overlap must be identified. It is highly inefficient to check each agent against all other agents. Even with the sharing of agent information in local memory, the implementation of the brute force approach was about 100 times slower than other methods, with an execution time of 45 s for $2^{20} = 1048576$ agents on a 1024×1024 grid. Hence, sophisticated broad-phase algorithms are usually applied that lower the number of inter-agent checks that must be performed. Not all are suitable for GPU architectures. Complicated flow-control and memory accesses can limit the achieved performance. Apart from the brute force method, for this thesis two algorithms have been implemented and tested, *Sweep and Prune* (SaP) and *Binning*.

6.3.1.1 *Sweep and Prune*

SaP (also called Sort and Sweep) has been previously implemented for GPUs, for instance by Liu et al. [46]. Its original sequential algorithm works by projecting the objects onto an axis, this could be the x-axis for example, and afterwards checking which images overlap. The sequence for agents is as follows

1. Project the extent of each agent a_i onto the chosen axis resulting in a set of intervals $I_i = [start_i, end_i]$ (inclusive). Each I_i describes where along that axis the agent starts and ends.
2. Sort all $start_i$ and end_i into a common list L of start and end points.
3. Define the active set of agents as $A = \emptyset$. Sweep L by stepping through the entries
   a) If $start_i$ is encountered, add a_i to A.
   b) Add all a_i in A to the set of potential pairs of colliders.

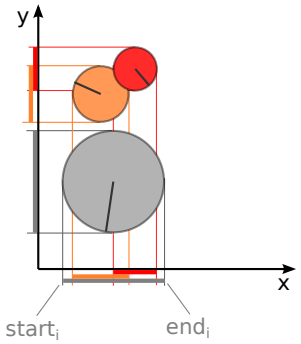


Figure 38: Projections for 2D Sweep and Prune.

c) If end_i is encountered, remove a_i from A.

This will find all potentially colliding pairs of agents that overlap when projected onto the given axis. In a 2D space this must be done for two axes, for instance the x- and y-axis, to determine if the agents really collide, as in Figure 38. Then the results for both axes can be combined and the pairs of colliders are defined. Only if the agents overlap on both axes, a pair of colliding agents has been found. In the example the orange and red agents in the top right corner intersect.

For simple circular objects like the agents in this thesis, it is easier to perform Sweep and Prune only along one axis, and check potential pairs by comparing the radii and positions. There is a collision if the inequality

$$\begin{aligned} \left\| PositionOffset_{a_i,a_j} \right\| &= \left\| Position_{a_i} - Position_{a_j} \right\| \\ &< Radius_{a_i} + Radius_{a_j} \end{aligned} \tag{6.1}$$

holds, which can be simplified to

$$\left(PositionOffset^x_{a_i,a_j} \right)^2 + \left(PositionOffset^y_{a_i,a_j} \right)^2 < \left(Radius_{a_i} + Radius_{a_j} \right)^2,$$

in order not to compute the square root. The start and end entries are not necessarily the actual position values along that axis, they could also be pointers referencing the agents. If the agents don't move much, the relative position of the pointers in the list remains the same. Also limited reordering can be implemented efficiently with swapping. Still, this algorithm is not suitable for parallel implementations on the GPU because of how the checks are performed in a sequential fashion and the memory accesses due to swapping [46].

Fortunately it can be adapted to parallel architectures. The authors of [46] created the following algorithm

1. Project each agent a_i onto the chosen axis resulting in a set of intervals as in the original algorithm.

2. Sort only the $start_i$ into the list L_P.

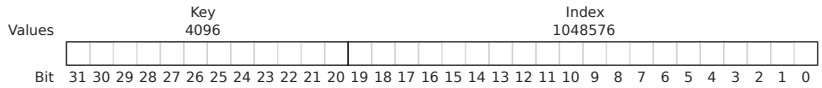


Figure 39: Entry pair with agent key and index for radix sort.

3. For each $start_i$ in parallel, beginning at $start_i$, step through L_P and process each $start_j$, until a $start_j$ is encountered that is greater than end_i

    a) Check if a_i and a_j really intersect with the radius and position test in Equation 6.1. If so, add them to the list of colliding pairs.

Hence, several work-items can concurrently scan a subset of the list for colliders in parallel. The work mentions *radix sorting* as the sort algorithm of choice because efficient parallel implementations are available.

For this thesis the algorithm is further changed. Instead of the starts of the agent extents, the agent position itself is sorted. The decision to manage one 32-bit value per agent during the sorting process, makes it impossible to use a single precision floating-point variable for the position. This decision was partially made on the assumption that radix sort only works with integers, although variants for floating-point values do exist [75].

The position of each agent is converted to an integer value, with rounding to negative infinity, that is used as the input to the radix sort implementation for integers. This is called the *key*. The integer maps the agent to an interval on the axis. The choice of how wide the interval is affects the search for collisions as explained below. If the agent's position is negative, the resulting integer value will look like a large positive integer to the radix sort algorithm. Later, during the actual collision resolution, the floating-point position of the agent is retrieved, and agents with negative positions are ignored. Each *entry pair* in the list also has to provide the *index* to the buffers containing all the agent positions and radii, for the subsequent offset test. Hence, the 32-bit integer value used, stores the key and the index, as in Figure 39. The index uses the 20 lower bits to support up to $2^{20} = 1048576$ agents. The most significant 12 bits containing the key allow up to $2^{12} = 4096$ integer positions along the axis, designed for a maximum grid size of 4096×4096.

Parallel radix sort works by counting the occurrence of each bit b_i equal to zero in the keys of all agents [28]. This requires as many passes as there are bits, 12 kernel invocations for the 12-bit key. This is another reason why no 32-bit floating-point type has been used. The first pass starts with the least-significant key bit, b_{20} in Figure 39. The kernel counts how many keys have $b_{20} = 0$. Then all the keys containing a one are moved before those containing the zeros. Otherwise the order does not change. In this regard, radix sort is a stable sort, as it does not

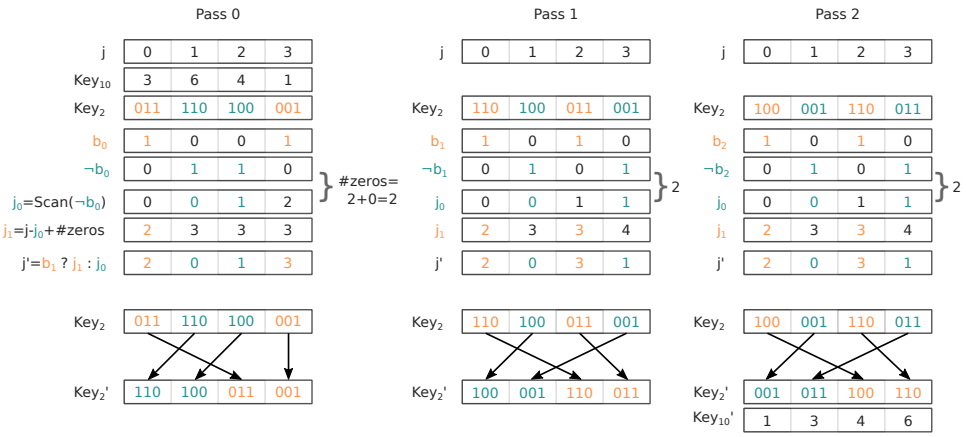


Figure 40: Radix sort passes for three-digit keys.

reorder values of the same size. Then the algorithm continues with the next bit $i+1$ until all bits have been processed. In the implementation a bit mask is provided to the kernel, which marks the relevant bit for the current pass. Over time, bits 20 to 31 in Figure 39 are processed.

For counting the bits the parallel *exclusive scan* primitive is used [28, 62]. By choosing addition as the associative operation, with zero as the identity element, scan computes the *prefix sum* over an input array. Each subsequent entry of the result contains the sum of all previous entries of the input, excluding the input with the same index as the result. For instance, the prefix sums of $\{3,2,6,4\}$ are $\{0,3,5,11\}$.

The *split* primitive [62] extends scan and rearranges the array elements as needed for the radix sort. An example sort with the numbers $\{3,6,4,1\}$ is depicted in Figure 40. The sort is performed in three steps. In each step split separates the elements where the relevant bit is zero from those where it is one. The line j shows the current index of the element in the array. Key_{10} is the key in the decimal number system and Key_2 is its binary representation. First the bit b_i is extracted and its negation $\neg b_i$ computed. For each pass, the elements that have $b_i = 1$ are illustrated in orange, and those with $\neg b_i = 1$ in teal. Then a prefix sum is performed over the negated bits with scan, resulting in what will become the new array indexes (j_0) of elements with the bit equaling zero. The total number of counted zeros (#*zeros*) can be found by adding the prefix sum for the last entry with the value of its inverted bit. The index of entries with a one bit is then calculated as $j_1 = j - j_0 + \#zeros$. Finally, the actual new index j' of each element is selected by checking the bit b_i. It serves as the destination for the scatter operation. If $b_i = 1$, then $j' = j_1$, else $j' = j_0$. The whole procedure is repeated twice and the reordered list $\{1,3,4,6\}$ emerges out of this radix sort.

A single sort pass is performed on the elements of a work-group. To sort more keys than fit into a single group, each group must know the #*zeros*′ of all the previous groups. The solution to this is given

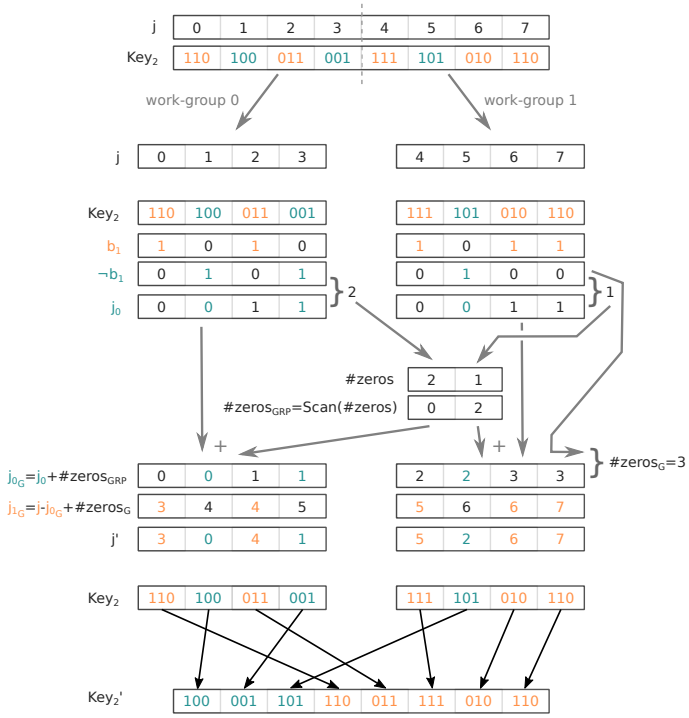


Figure 41: Radix sort for two work-groups.

in [28]. Figure 41 shows an example for two work-groups and the second pass ($b_i = b_1$). The *#zeros* in each work-group is written to a separate global buffer by the prefix sum kernel so that another kernel invocation can compute the prefix sum over these *#zeros'*. These are the two-element arrays in the figure. The second one contains the number of zero elements before each group $\#zeros_{GRP}$.

Then the $\#zeros_{GRP}$s are added to the original prefix sums computed locally inside each group, the j_o entries. This results in the global prefix sums j_{o_G} of the whole array, which again is the number of elements with $\neg b_i = 1$ before the current element for each of the entries. Next, the global number of zero bits $\#zeros_G$ for the current pass is found by adding the negated bit state of the very last element with its prefix sum. Now the j_{1_G}s can be derived as given in the image and the computation of the new indexes and the scattering can be performed. Things are more complicated if the $\#zeros_{GRP}$s don't fit into a single work-group, then the prefix sum computation for $\#zeros_{GRP}$ has to be separated into several work-groups and the kernels must be invoked in a recursive fashion. In the work [28] an alternative variant is mentioned. Therein, the radix sort is performed efficiently in local memory for a work-group over all bits. The locally sorted keys of all groups are afterwards merged into the complete array by another sort algorithm. With this method no separate kernel instantiations must be performed per bit, because it is all done inside a loop in local memory. Because it requires the implementation of another sort algorithm, this has not

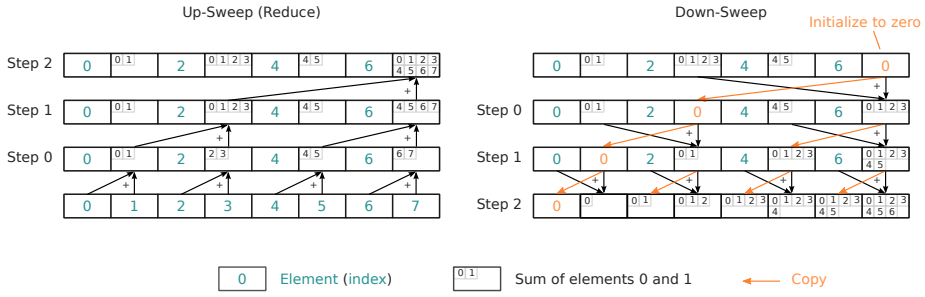


Figure 42: Parallel sum scan.

been compared with the previously mentioned variant that performs the radix sort in a global manner.

With complex algorithms like these, it is often beneficial to add helper functions during development that allow to inspect the values by copying the buffer contents to the CPU for logging to a console. AMD's implementation also supports the OpenCL extension that allows to print messages from a kernel running on the CPU. Kernels can even be debugged with the debugging program gdb when executed on the CPU. All this methods can help to find errors inside the program.

The actual parallel sum scan kernel is based on the CUDA C code given in [28]. It executes two passes, *Up-Sweep* and *Down-Sweep*, as shown in Figure 42. The teal numbers are indexes in the array. The Up-Sweep is a reduction that sums up all elements, and keeps the partial sums. In the Down-Sweep the last entry is initialized to zero, the total sum is not needed. In each subsequent step the values are then summed up and copied in a special pattern. The orange zeros are actual values (sums). Finally, the exclusive prefix sums are stored in the original array. The computation is done per work-group in local memory for optimal speed.

After all agent entry pairs that contain the keys have been sorted, collisions can be searched. This is done by searching through the agents inside the integer intervals they have been assigned to. With the conversion of the agent's position to an integer information has been lost. Based on the entry pair each agent is only known to be situated inside the interval of the projection axis that is related to the integer. For instance, if the agent's position is $(3.65, 5.74)$, the projection to the x-axis and conversion to the integer 3 will only tell that the agent is somewhere inside $[3, 4)$ (left inclusive, right exclusive), if the interval spacing is 1. If the interval spacing is wider, in order to cover a larger range of position values and a larger domain, the information becomes less accurate, and more agents need to be checked per interval. For instance, if the spacing was 4, the example agent would be somewhere inside the interval $[3, 7)$ along the x-axis, and all other agents in that interval need to be checked.

By defining a maximum agent radius, it becomes possible to describe a search pattern that will reveal overlapping agents. For each entry in the sorted list of entry pairs, a work-item linearly searches the neighboring entries to the left and right in the array, as shown in Figure 43, until the sampled entry pair mentions a key that is situated in an interval outside of the reasonable distance resulting from the maximum radius defined over all agents. This kind of abort condition is required, because the algorithm can not stop based on the position of other agents, as done in Step 3 of the algorithm of [46] listed on page 75. In [46] the extents are sorted based on their floating-point position which provides the correct order of entries up to floating-point precision. In the implementation for this thesis, the integer radix sort causes several agents to reside inside an interval, because of the reduced integer precision. The actual order of these agents is not provided by the sort. If the algorithm would abort on the first agent that falls outside of the rule, another agent afterwards in the list and same interval, but actually, before the aforementioned agent, in floating-point precision, might be missed. For instance, the interval 3 might contain four agents at the x-positions $\{3.65, 3.21, 3.8, 3.01\}$, in that order. If the algorithm is searching through the entries of that interval, it can not abort at the agent with x-position 3.65 if it is outside the reasonable distance, because the agents with 3.01 and 3.21 might still be in reach. Only on the integer and therefore interval level can this decision be made.

In addition to its own interval, the work-item must check the intervals below and above the current one. The exact number of intervals can be found by analyzing the situation for different examples as in Figure 44. The horizontal lines indicate the interval boundaries. The Orange arrows show the agent extents projected onto the single axis. The extents are of identical size because only $Diameter_{max}$ is considered. The gray arrows indicate how the agent $start_i$s are mapped to the integer interval boundaries. The analysis is the same for the agent positions mapped for the algorithm instead of the starts. The mapping to the interval boundaries can be done with rounding to negative infinity, as long as the diameter and spacing have the same unit, and if the spacing is one. This is the case in the implementation for this thesis, where the interval spacing corresponds to the grid spacing of one.

In the first example the interval spacing is twice the diameter. For agents lying in the first interval, all agents in the next interval need to be also considered, because they could stretch across the interval border. Any agent that is actually situated just short of the edge of the first interval, might extend into the next interval. This also holds for the previous interval, and one extra interval needs to be checked in both directions. Let k be the number of extra intervals, then $k = 1$. In the second example, the ratio is one and still one additional interval

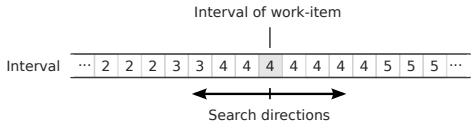


Figure 43: Intervals represented by the sorted entry pairs and search directions.

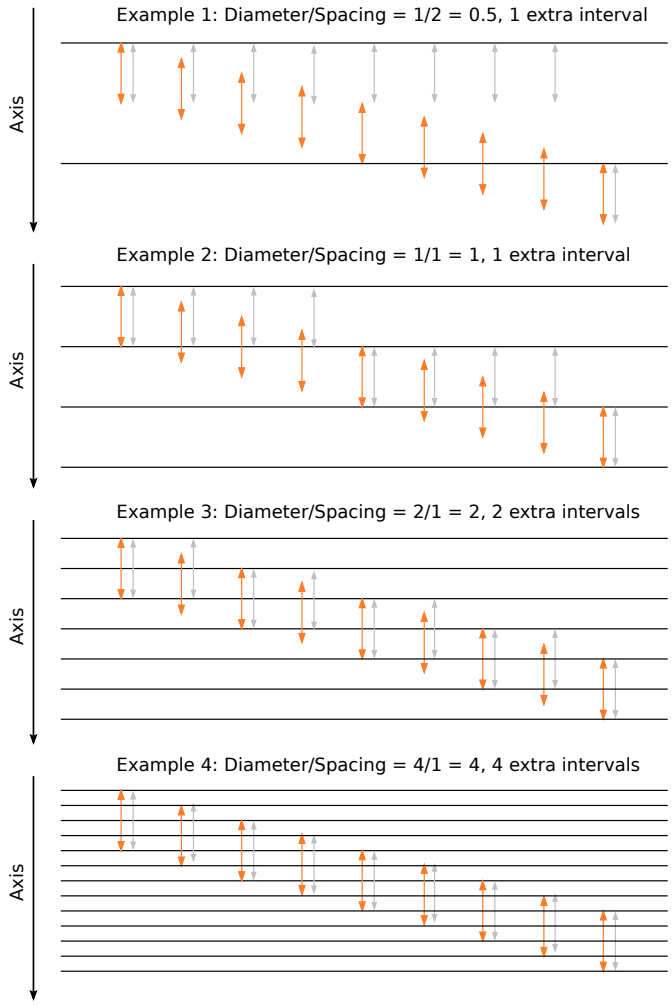


Figure 44: Relation between maximum agent diameter and interval spacings.

needs to be checked in each directions, thus, k is again one. In the third example, the ratio is two, and each agent can extend into the interval two steps further than the current interval, hence, $k = 2$. This behavior continues in the fourth example, and reveals the pattern

$$k = \lceil Diameter_{max} / IntervalSpacing \rceil . \tag{6.2}$$

$\lceil \ldots \rceil$ is the ceiling function. Therefore, the total number of intervals that need to be checked per work-item is $2k + 1$. If $Diameter_{max}$ is large

in relation to the interval spacing, many intervals must be searched, and vice versa. All agents inside the range of interesting intervals are further considered with the detailed position and radius check to see if an overlap occurred. The algorithm works as follows

1. For each entry pair E_i, representing an agent in the interval I_i, in the sorted list of entry pairs L_E, in parallel
    a) Step through L_E to the left by decrementing the index i. Check each entry pair E_j until its key describes an interval less than $I_i - k$.
        i. Perform a detailed position and radius check on the agent described by E_j. If the agents intersect, add them to the list of intersecting agent pairs.
    b) Step through L_E to the right by incrementing the index i. Check each entry pair E_j until its key describes an interval greater than $I_i + k$.
        i. Perform a detailed check as in item 1(a)i.

Experiments revealed that the radix sort is reasonably fast with an execution time of 10 ms, with 1048576 agents randomly spread over a 1024×1024 domain. But the collision search step performs poorly with an execution time of 2.8 s in the same example. This can be partially explained by looking at how the agents in that example are projected onto the x-axis and mapped into the intervals. Because more than a million agents are arranged all over the domain with 1024 intervals along the axis, about 1000 agents will be mapped to each interval. With the maximum agent radius causing the agents to extend over multiple intervals, the number of agents that need to be checked per work-item, can be estimated as a multiple of 1000. With several random memory accesses per work-item this is too much. The authors of [46] employ multiple optimization techniques. One of them is choosing a different projection axis, that is not the x- or y-axis. They compute an axis based on how the objects are arranged in the domain. The optimal axis causes less false positives with potential collider reports. For the algorithm using integer intervals, it also causes a smaller number of agents to be projected into each interval. For instance, if the agents are spread out in an elongated cloud of agents, the axis of choice would be the principal axis leading through that cloud. But because the agents are uniformly distributed, at least at the start, a different choice of axis does not generally improve the situation in the application for real-time crowd simulation. Using two projections on two orthogonal axes, like the x- and y-axis, does not help either. This would require twice the amount of work for searches without any improvement, because the sweep over one axis can not be sped up with information from the other axis. Even with additional spatial subdivision and workload balancing the performance achieved in [46] is insufficient. Apparently,

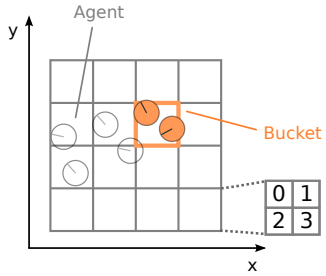


Figure 45: Agents assigned to buckets with binning.

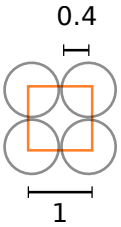


Figure 46: Fitting four agents with smallest radius into a bucket.

Sweep and Prune is not a suitable broad-phase optimization technique for collision detection on the GPU. As a consequence, the Sweep and Prune implementation has been removed and is not available in the application. Another algorithm using binning provides far superior performance.

6.3.1.2 *Binning*

Binning techniques have been previously implemented for applications on the GPU, for instance, with shader programs [27, 65], and CUDA [25]. Binning works by subdividing the domain into a regular arrangement of bins, sometimes also called *buckets*. This results in a grid structure similar to the cell grid used for the navigation data computation in Chapter 5. Figure 45 shows how a domain can be subdivided into buckets and how the agent position relates to their assignment to a bucket. The two orange agents are assigned to the bucket of the same color. The buckets can contain several agents. The extent of each bucket is known from the subdivision of the domain. Hence, broad-phase collision culling can be performed similar to the interval searches in the custom Sweep and Prune method. Actually, the intervals correspond to 1D buckets. But now the agent's interval information is defined for both dimensions in a 2D environment. Thus, there are far less agents per bucket than there were agents per interval.

For this thesis a straight forward OpenCL implementation was developed. The bucket size is defined to be one, which is the same as the cell width for the navigation computation. The Collision Resolver in Figure 34 maintains two buffers. *Buckets* contains one *bucket entry* per agent inside the bucket. There is a limit of four entries per bucket, as indicated in Figure 45 for a single bucket on the right. Hence, there can never be more than four agents assigned to a bucket. Each entry is an unsigned integer providing the agent's index to the property buffers. The second buffer *Counts* manages the number of used bucket entries per bucket. For instance, if there are three agents in the bucket, Counts will say that three of the four entries in Buckets are currently in

use. The relation between agent radius and the size of the bucket area is chosen to make it unlikely that more than four agents reside inside a bucket at any time. Still, if there are more agents, the additional agents will be ignored during collision detection. This effect is reduced over time, because agents will get separated and the density should decrease. Therefore, the number of agents inside a bucket will decrease, in highly congested areas, as long as there is room for the separation step to displace agents. As mentioned previously, the minimal allowed agent radius is 0.4, which allows about four agents to fit into the area of a bucket without intersection, as shown in Figure 46.

In each simulation step, the Collision Resolver first clears the count of used buckets to zero with a kernel. Then the agents are assigned to the buckets by another kernel with a 1D NDRange. Each work item-processes an agent. First the agent's position is retrieved. If it contains negative components, the agent is parked and no additional computation is performed. Else, the conversion to the bucket index is done by first rounding each component of the position vector to negative infinity. Then the index into the bucket entry buffer is computed as

$$\begin{aligned} Bucket &= Position_{a_x}^{-\infty} + Position_{a_y}^{-\infty} \cdot SideLength_{BG}, \\ BucketEntry &= 4Bucket, \end{aligned}$$

where $Position_{a_x}^{-\infty}$ and $Position_{a_y}^{-\infty}$ are the rounded position components, $SideLength_{BG}$ is the number of buckets along the side of the bucket grid, $Bucket$ identifies the bucket being accessed, and the factor four originates from the four possible entries per bucket. Incrementing this index by one, steps over four consecutive bucket entries.

The actual entry is found by atomically increasing the current count for the bucket. This also returns the previous count $count_{old}$. The kernel then checks if $count_{old} < 4$. If this is the case, it writes the agent's index into the buffer, and thus, the bucket. This procedure gives valid indexes into the buffer, because the count written previously by any work-item while the bucket is not full, correctly provides the index for the next entry. For instance, if the current count is three, it will be incremented to four, and the work-item writes to the bucket entry with sub index three, inside the list of entries for the bucket, as shown in Figure 45. The next work-item accessing the count will increment it to five, and because it was four previously, the work-item will not write to the buffer.

After the binning phase is complete, the Collision Resolver searches for potential colliders inside the buckets. This search is performed similarly to the one for Sweep and Prune where intervals have been inspected. The same logic applies and the number of additional buckets that have to be checked in each direction is given by Equation 6.2, with the $IntervalSpacing$ replaced by the bucket's side length. As the

bucket size has been chosen to be one for this implementation the equation changes to

$$k_B = \lceil Diameter_{max}/1 \rceil = \lceil Diameter_{max} \rceil .$$

Now the number of directions is four, which creates a square of buckets that must be searched for agents colliding with the current agent. This square covers $(2k_B + 1)^2$ buckets. Because of the domain boundaries the square is truncated if it extends beyond the simulated environment. The range of buckets in the x-direction can thus be given as $[LowerBound_x, UpperBound_x]$ (inclusive), where

$$\begin{aligned} LowerBound_x &= \max\{Bucket_x - k_B, 0\}, \\ UpperBound_x &= \min\{Bucket_x + k_B, SideLength_{BG} - 1\}, \end{aligned}$$

$Bucket_x$ is the column of the agent's bucket in the grid of buckets. For the y-direction the interval is defined in a similar way.

Again a kernel is invoked with a 1D NDRange. Each work-item processes an agent a_i and retrieves its position $Position_{a_i}$. If the position hints that the agent has been parked, no further computation is performed. Else, the kernel recomputes the agent's bucket, which, together with k_B and the cutoffs at the boundary, provides the intervals above, defining the rectangular selection of buckets that must be searched. In two nested loops the work-item iterates over the buckets and one-by-one inspects the four entries. If the stored index is not the same as the index of the current agent, the entry stems from a different entity a_j. The kernel can now retrieve its position $Position_{a_j}$ and compute the difference

$$Offset_{a_i,a_j} = Position_{a_i} - Position_{a_j}$$

between the potential colliders. If $Offset_{a_i,a_j} = \overrightarrow{0}$, the two agents are in the same spot. This situation is handled in a special way during the separation step below. If the offset is not the null vector, the squared distance between the two agents

$$Distance^2_{a_i,a_j} = \left(Offset^x_{a_i,a_j}\right)^2 + \left(Offset^y_{a_i,a_j}\right)^2,$$

is calculated with the x- and y-component of the distance vector. If

$$Distance^2_{a_i,a_j} < Diameter^2_{max}$$

an intersection is possible. This situation is illustrated in Figure 47a. Only after this has been ascertained, the actual radius $Radius_{a_j}$ of the other agent is retrieved from global memory. This saves memory accesses and improves the performance. In addition, no distance and square root has been computed yet. The comparison of the squared distance with the squared sum of both radii reveals if the agents intersect, even if they are of different size. This condition can be given as

$$Distance^2_{a_i,a_j} < \left(Radius_{a_i} + Radius_{a_j}\right)^2.$$

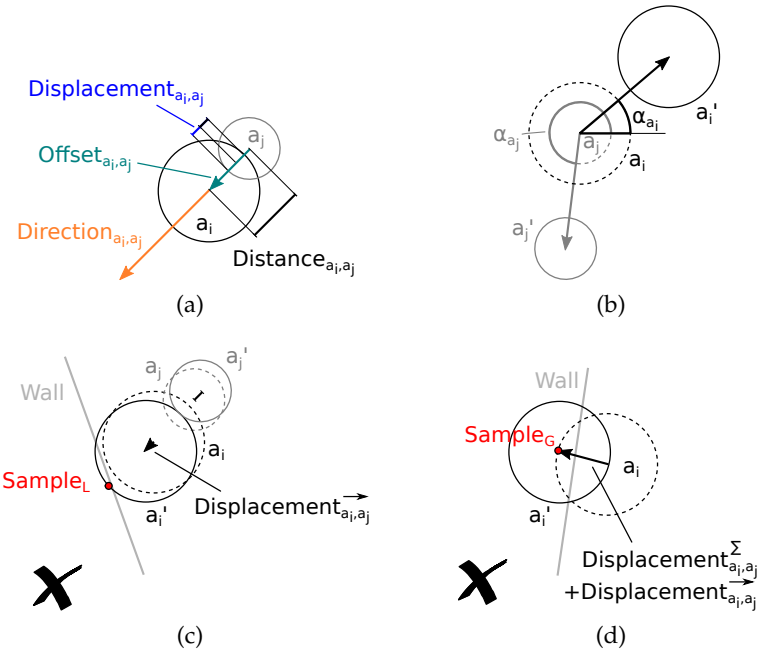


Figure 47: Agent intersection and separation. (a) Separation by distance and direction. (b) Separation by angle. (c) Failed local displacement because of intersection with wall. (d) Failed global displacement due to wall.

6.3.2 *Agent Separation*

Two kinds of intersecting agents can be found during the previous search, those that intersect only partially and those that are exactly in the same location. These cases must be handled separately because of how the agent displacement is performed, but never is an explicit list of colliding agent pairs maintained. Instead, the same kernel that searches for colliding agents does also perform the separation step in one go, by managing a displacement sum $Displacement^{\Sigma}_{a_i}$. This vector is initialized to the null vector. For each intersecting agent, the required displacement is computed and added to the sum. Thus, for a pair of overlapping agents, each agent mutually performs half of the collision resolution offset.

If the agents are not in the same location, the kernel computes the actual distance between the two agents by taking the square root of the previously computed squared distance

$$Distance_{a_i,a_j} = \sqrt{Distance^2_{a_i,a_j}}.$$

It is required to compute the necessary displacement of the agent. With the distance the unit direction vector

$$Direction_{a_i,a_j} = \frac{Offset_{a_i,a_j}}{Distance_{a_i,a_j}}$$

can be given. Each agent then contributes

$$Displacement_{a_i,a_j} = \frac{Radius_{a_i} + Radius_{a_j} - Distance_{a_i,a_j}}{2}$$

to the resolution. But the corresponding offset vector

$$Displacement^{\rightarrow}_{a_i,a_j} = Displacement_{a_i,a_j} Direction_{a_i,a_j}$$

can only be added to the vector sum $Displacement^{\sum}_{a_i}$, if it does not push the agent into a wall. Hence, the potential field is sampled in the displacement direction at what would be the edge of the agent if it were at the new position, as in Figure 47c. This sample position is the grid position corresponding to

$$Sample_L = Position_{a_i} + Displacement^{\rightarrow}_{a_i,a_j} + Radius_{a_i} Direction_{a_i,a_j}.$$

If the potential is infinite, a wall has been hit. In the figure both agents have been moved to the theoretical positions a'_i and a'_j. The current positions are indicated by the dotted circles. This would resolve the collision, but a'_i is intersecting with a wall, as detected by sampling at the red dot. As a consequence, the agent is not moved. a_j might still be able to move to a'_j, so the intersection will be partially resolved. If the agents are additionally being separated from other agents, and if they are moving, the situation will be fully resolved eventually.

If a_i can be moved, another check is performed to ensure that the sum of displacements will not push the agent into a wall either. After all, the separation from other agents could influence the position to cause an intersection with a wall, but it might also allow the movement $Displacement^{\rightarrow}_{a_i,a_j}$, even though the single offset test above would prohibit it. This test primarily assures that the center of an agent does not end up inside a wall, because of the collision resolution. This would cause the agent to be parked and suddenly disappear, which must be prevented. Hence, the potential at the grid location corresponding to

$$Sample_G = Position_{a_i} + Displacement^{\rightarrow}_{a_i,a_j} + Displacement^{\sum}_{a_i}$$

is sampled. If a wall is found, as in Figure 47d, the local displacement will not contribute to the sum, otherwise $Displacement^{\rightarrow}_{a_i,a_j}$ is permanently added to $Displacement^{\sum}_{a_i}$. This technique gives good collision resolution. Only if many agents intersect with a_i the summed offset might be large and cause an irrational offset. In the experiments this case only appeared when the situation was artificially designed for it, in normal circumstances the algorithm performs reasonably. If this is of concern, the displacement vector could be limited to a maximum length.

The second case revolves around agents that have identical positions, where $Position_{a_i} = Position_{a_j}$. Here, the offset vector $Offset_{a_i,a_j}$ would be the null vector, and the division by its length would lead to infinity.

Hence, a different approach to separate the agents must be taken. As before, each work-item changes only one of the two involved agents. In order to make sure that the agents don't get moved into the same direction, some kind of convention has to be used. The system designed for this thesis uses the index of the agent to choose a direction for the displacement based on the angle

$$\alpha_{a_i} = \frac{i}{AgentCount} 2\pi,$$

where $AgentCount$ is the total number of agents. If the number of agents is high, it is unlikely that two agents have similar angels when they are separated. In the case where the agent count is low, there are less possible angles, and therefore they differ more. Consequently, this should provide good separation directions, as in the example in Figure 47b. The direction is then given as

$$Direction^0_{a_i,a_j} = (\cos\alpha, \sin\alpha),$$

and radially points away from the agents current position. The local displacement is then

$$Displacement^{\rightarrow 0}_{a_i,a_j} = Radius_{max} Direction^0_{a_i,a_j},$$

to move the agent as far as reasonable. Because all the $Direction^0_{a_i,a_j}$ are identical, the displacement replaces $Displacement^{\sum}_{a_i}$ instead of being added to it. The reason is, if other agents would also be in the exact same position, all the identical offsets would add up and move the agent over a large distance. Again, this offset could be clamped to some maximum. But this case of agents ending up in the exact same spot is very rare. Furthermore, the computation of the trigonometric functions can be skipped by many wavefronts, because hardly any work-item will take that branch of the conditional.

After all work-items have processed their agents, most intersections have been resolved. If the congestion is high and very limited space is available, the collision resolution algorithm is unable to resolve all collisions. With each simulation step agents will then be displaced back and forth between positions, that cause intersections, resulting in oscillating agents and an overall turbulent crowd behavior. This usually happens when suddenly many agents are spawned into a tight area. In the common case, agents don't move up too close because of the speed constraint caused by agent density, and such extreme situations don't occur.

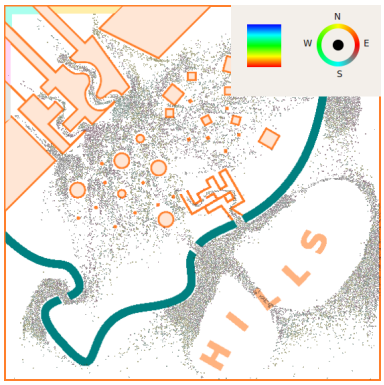
7 VISUALIZATION

After the computation steps have completed everything is displayed to the user. The visualization given by the application provides immediate insight into the result of the simulation, and thus, the behavior of the agents. Simultaneously, it can be used to inspect the separate computation steps for study, experimentation and debugging. Additionally, the GUI system, as shown in Figure 20, allows to enter properties to the simulation, and shows internal values to the user. Most elements feature tool tips that explain the functionality and list the keyboard or mouse shortcut if available.

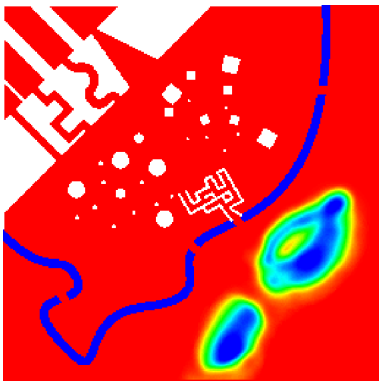
A view area shows the actual scene and agent movement. It can be panned by holding the right mouse button and zoomed with the mouse wheel. In addition, a number of different views can be displayed or partially overlaid. The *toolbar* at the top of Figure 20 provides options for pausing the simulation, running and single stepping it, as well as, buttons replicating the zoom functionality of the mouse wheel. On the left a large *control panel* contains all the options and input fields grouped into two sections, *display* and *computation*. At the bottom *text windows* show the application log, profiling information, and memory usage data.

7.1 FIELDS

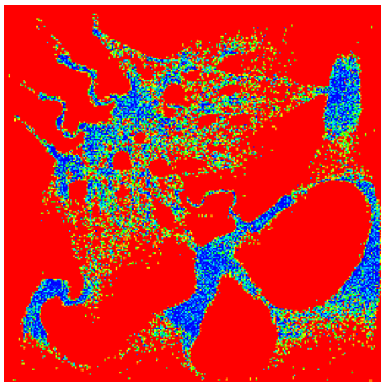
The control panel provides options to visualize the content of the buffers listed in Figure 21. By sharing the information between OpenGL and OpenCL, the application can easily render the data as a textured quad to the view area. For the inspection of anisotropic fields, the user can select one of the four directions. If buffers contain information per agent group, the required group can be select. For easier understanding, a legend automatically updates whenever the selection changes between a scalar or a vector field. Figure 48a shows the legends in the upper right corner. For scalars a color scale shows how the values are visualized according to minimum and maximum values that can be picked from the view, as explained in the tool tips, or entered in a text field. Any value smaller than the minimum boundary is colored in red, any larger than the maximum one, is colored in blue. The boundary values are displayed in the GUI, as in Figure 20. In between that, values are interpolated between, red, yellow, green, cyan, and blue, providing a distinguishable range of colors. Infinity is indicated by white and NaN by purple areas, in order to stick out of the regular colors. Whenever the boundary values change, the coloring in the



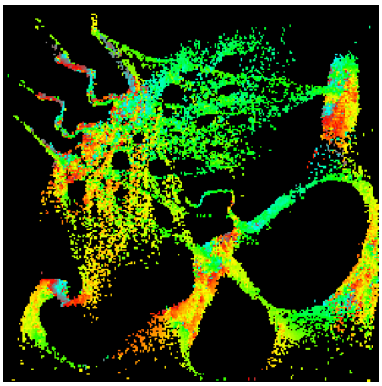
(a) Map image, agents, and legends.



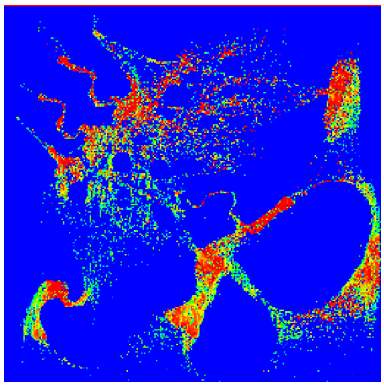
(b) $DiscomfortSum_{i,j}$ scalar field.



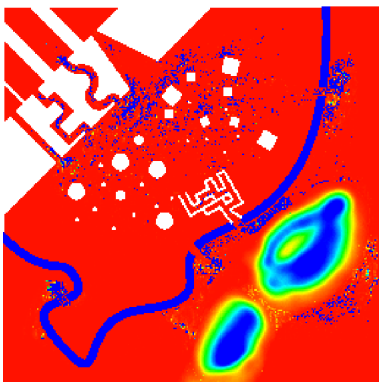
(c) $DensitySum_{i,j}$ scalar field.



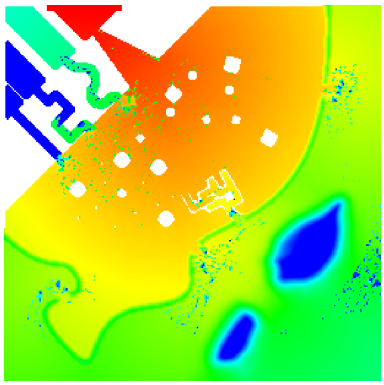
(d) $WeightedAverageVelocity_{i,j}$ direction vector field.



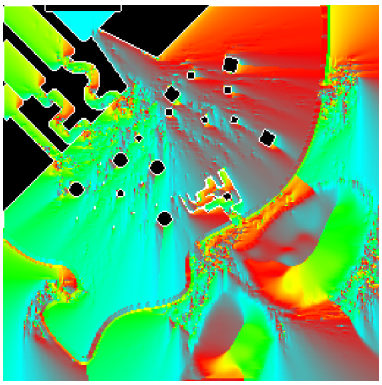
(e) $Speed_{i,j_N}$ scalar field.



(f) $Cost_{i,j_N}$ scalar field.



(g) Potential scalar field.



(h) $Gradient^0_{i,j}$ direction vector field.

Figure 48: Different overviews of a scene.

view area updates dynamically. In Figure 48 the scalar fields are all displayed with different minimum and maximum legend boundaries. Vector fields are similarly colored. The choice of colors influences how easy it is to visually distinguish values and spot jumps. Furthermore the color scheme should allow to identify agents, even if drawn on a colorful background field. Hence, the appropriate selection of a color scale and group color for agents is a difficult task.

7.1.1 *Navigation Data*

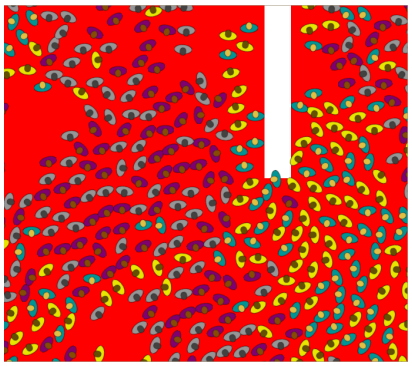
Figure 48 shows several example views of the same abstract scene. Figure 48a contains the map, which can be overlaid with variable opacity over the other field views. At this zoom level, the agents appear as a gray mass. In fact they are colored by their group color, in a yellow tone (Group0), cyan (Group1), purple (Group2), and gray (Group3). The map shows their goal areas in the top left corner colored in the individual group colors. The legends have been added to the screenshot with an image manipulation program. Figure 48b shows the variable discomfort defined by the exit file. No additional discomfort from the discomfort brush is contained. The white areas are of infinite discomfort and therefore considered as walls. The blue line could represent a river, and is segmented by bridges. The colorful area in the lower right corresponds to the hills in the map image. Figure 48c shows that the density at choke points is increased (blue). In Figure 48d the agents in the open area in the north are heading to the west as indicated by the green color. Agents that want to move north are confronted with areas of low speed near congested regions, as shown in Figure 48e. Open cells, where no agents are situated, allow the maximum speed. Figure 48f depicts the cost contribution of the agents, walls, river, and the hilly region. As with the previous figure, the directional penalty for moving north is not clearly visible at this zoom level. Figure 48g gives the potential for Group0 as computed for this scene. The exit area lies in the triangular region on the top, just where the yellow bar is shown in Figure 48a. Finally, Figure 48h displays the corresponding gradient direction vectors.

The previous views showed an overview of a scene and how the various fields appear at a global scope. This hides some of the details as mentioned for the speed and cost fields. Figure 49 shows a different scene, with larger agents, and a higher zoom step. Hence, various details become visible. Figure 49a illustrates the discomfort. On the right a wall separates the agents, on the left many gray and purple (Group2 and Group3) agents are streaming towards the south. To the right of the wall the agents are also heading downwards. Between the wall and the large flow on the left, a few yellow and cyan agents (Group0 and Group1) move towards the north. Figure 49b indicates that the density inside the flows is higher than in the cells where no

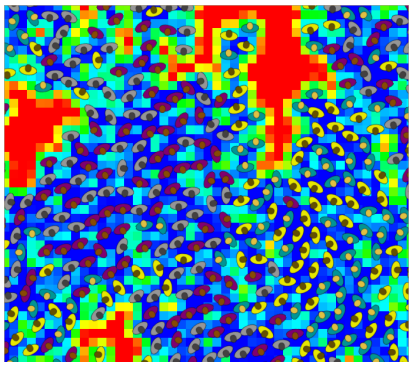
agents are. In this view the discretization size of the cells is visible. In this scene it is one unit and the same as the agent radius. Figure 49c reveals the average velocities inside the crowd, indicated by color and small arrows. In this view it becomes apparent how the views show the results of the current navigation computation, while the displayed agents have already turned based on this information. The yellow agent in the black region on the lower left, is heading to the north-west. The direction indicated in the field by green color and the arrows shows a slightly different previous heading, leading more towards the north. The visualization in Figure 49d communicates how the average direction influences the anisotropic speed field. The image shows the field for heading to the north. The red areas mostly contain agents heading south. Consequently, agents moving against them are affected by low speed. The small number of yellow and cyan agents heading north next to the wall in the upper area of the view, experience higher speeds, as indicated in green. The two leaders on the top see even higher speeds (blue), because no other agents influence their movement, by contributing density and velocity. Figure 49e shows how the discomfort at the wall and the different speeds result in the cost for heading north. Moving against the flow of agents streaming towards the south is costly (blue), the small group of agents heading north uses the area of lower cost (orange). Figure 49f provides the potential of Group0 (the yellow agents). They will follow the potentials that decrease slowly, with small changes of potential in the color scale legend. This is visible in Figure 49g, where the arrows point in the opposite direction of where the agents are heading.

7.1.2 *Selective Update Tile States*

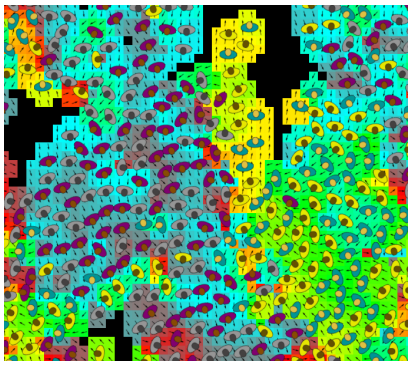
If the selective tile update mode is active, the option to show the "*Selective update tile states*" overlay is available. By enabling the GUI option "*Break after each solver step*" the tile update order can be visualized, if the single step button is used. Figure 50 shows how the tiles relate to the cell grid, the work-groups during potential computation, and the update state. In Figure 50a the chosen opacity of the work-group overlay still allows the map and agents to be visible. Each square in the checkerboard pattern corresponds to a work-group (tile) during the potential computation, as illustrated in Figure 50b. The granularity of the potential update algorithm only covers tile sized cell areas. Only whole tiles can be updated by a work-group. In Figure 50c the current tile states are visible. Tiles in state Sleep are red, those in Update are green, and the tiles that just converged (IsConverged) are shown in blue color. The block of green tiles will expand towards the bottom right, as shown in Figure 51. The update starts in the corner where the goal areas are, and expands over the domain towards the lower right,



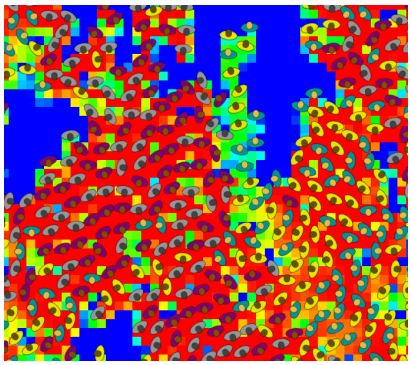
(a) $DiscomfortSum_{i,j}$ scalar field.



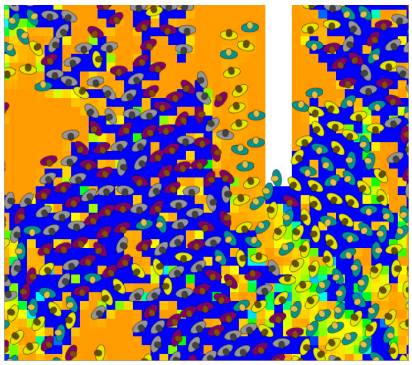
(b) $DensitySum_{i,j}$ scalar field.



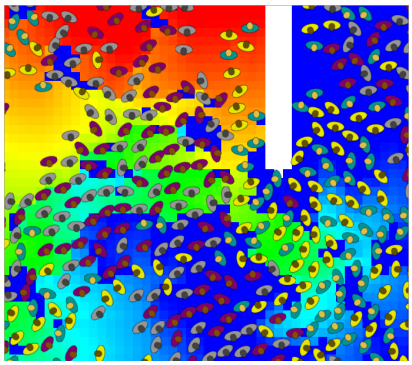
(c) $WeightedAverageVelocity_{i,j}$ direction vector field.



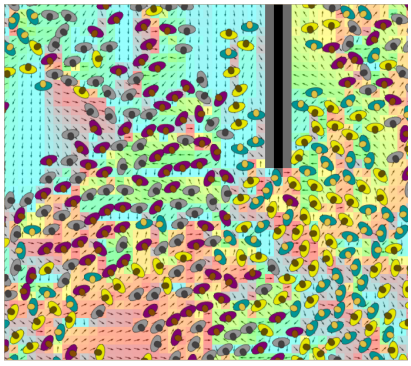
(d) $Speed_{i,j_N}$ scalar field.



(e) $Cost_{i,j_N}$ scalar field.

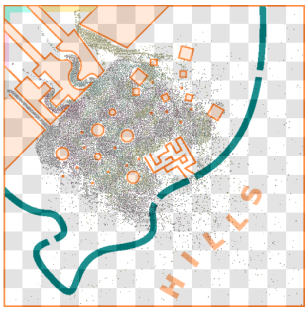


(f) Potential scalar field.

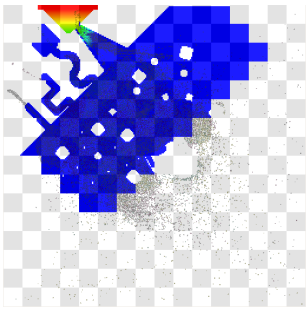


(g) $Gradient^0_{i,j}$ direction vector field.

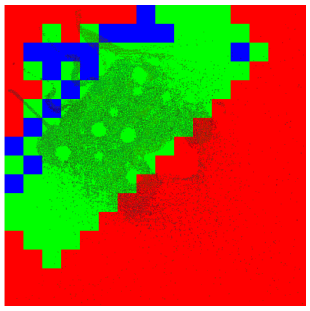
Figure 49: Zoomed-in views of a scene.



(a) Map, agents, and tiles.



(b) Potential, agents, and tiles.



(c) Selective update tile states, and agents.

Figure 50: Selective tile update states.

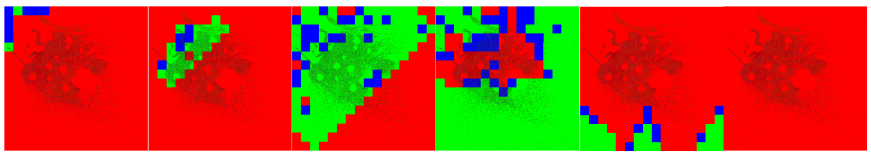
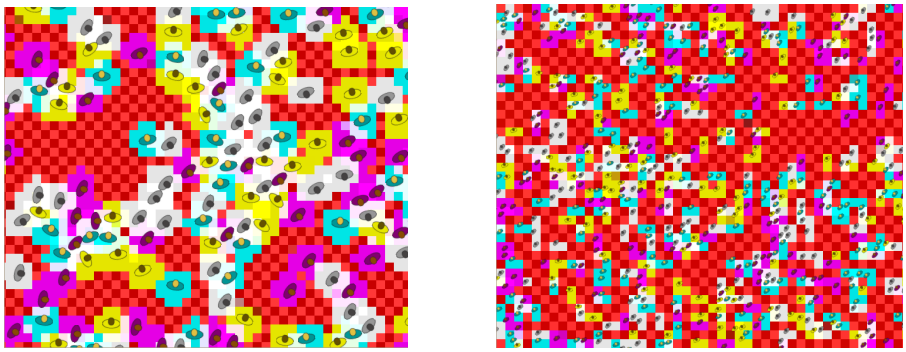


Figure 51: Tile update sequence.

until all tiles have converged. The number of outer steps between the images in the sequence varies.

7.2 AGENTS

The agents are visualized as sprites. In a separate pass they are rendered as quads based on the current zoom scale. This allows them to be of higher resolution than their representation used during the splatting phase. The textures used for the quads can be set per agent group in the configuration file, which makes it possible to use different agent looks per simulated scenario. During the agent splatting process a separate render target stores which cells are touched by which agents. These *"splat areas"* can be enabled for display in the GUI, as can the checkerboard pattern for work-items, both depicted in Figure 52. In Figure 52a the agents have a radius of 1, the doubled agent radius during the splatting is visible, and the splat areas cover a larger area than the actual agents do. Each agent splat covers several cells. Therefore,



(a) (b)

Figure 52: Agent sprites, splat areas, and work-items.

the conic density texture can at least be represented partially by this rough discretization. In Figure 52b the agent radius is only 0.4, and the splatting contributes to less cells. Experimentations showed, that even lower radii result in an insufficient contribution and no usable navigation data can be derived. Hence, in this implementation 0.4 is enforced as the smallest allowed radius.

Inspired by the impressive agent visualizations of other works, it was originally intended to include some form of animated agent meshes, and a 3D environment. Because writing a renderer and animation system is a lot of work, various open-source renderers were compared. Of those capable enough, the Ogre library [79] seemed to be the only one allowing to interface with the Qt GUI system [54]. Still, Ogre manages a scene graph on the CPU side. An extension exists that improves performance with instantiated and animated objects [80], but at the time this was tested in the application, it could not work directly with agent properties stored on the GPU. The round-trip of transferring the agent positions, orientations, and radii from the GPU to the CPU, updating the scene graph, and then rendering the agents on the GPU with Ogre, was just too costly. The rendering slowed the simulation down significantly, and only a few number of agents could be visualized with this approach, in an acceptable time frame. As a consequence, the 3D rendering is not available in the program. Nonetheless, a custom solution based on such works as [20] might still allow a 3D scene with acceptable performance.

After the visualization the agent properties are fed back into the Splatterer. This closes the simulation loop depicted in Figure 19.

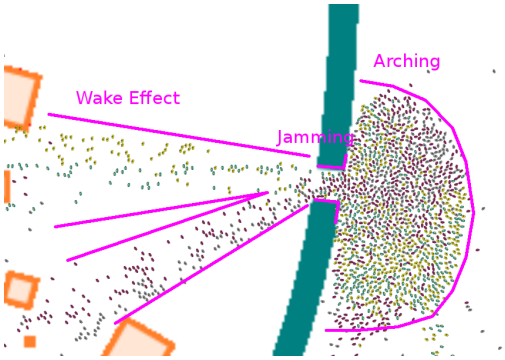
8

EXPERIMENTAL RESULTS

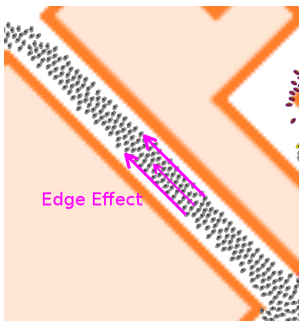
During the development of this thesis, little work has been spent on studying the actual behavior of human agents. Most certainly it is a highly complex topic. While the behavior of the agents simulated in the implementation can not be described as realistic, especially when analyzing the trajectory of individual agents, the overall flow exhibits some of the properties that can be seen in real crowds [26].

8.1 QUALITY OF THE SIMULATION

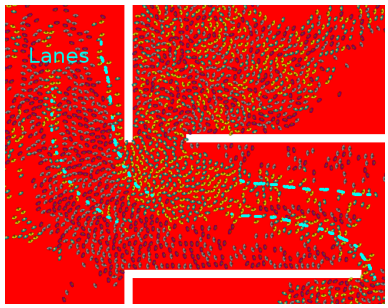
Figure 53a shows how agents *jam* tight openings. The subsequent agents form *arches* around these congested areas. As the agents continue through the opening, they spread out in a pattern called *wake effect*. Of course the direction of this pattern also depends on the location of the goal area. In Figure 53b the agents at the edge of the flow move faster than those in the middle, exposing the *edge effect*. Agents moving in opposite directions form *lanes* as shown by the dotted lines



(a) Jamming, arching, and wake effect.

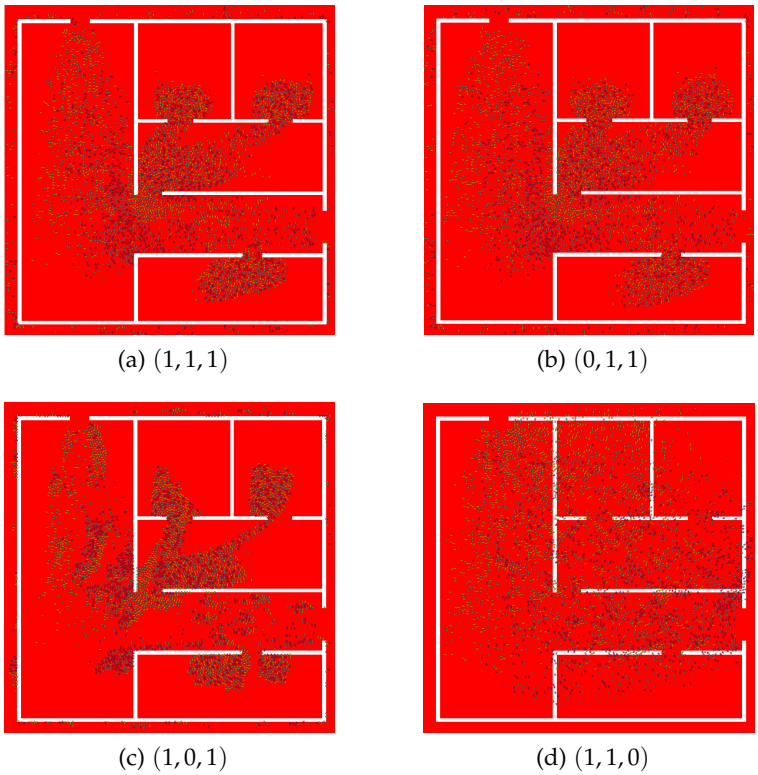


(b) Edge effect.



(c) Lane formation.

Figure 53: Emergent phenomena in crowds.



(a) $(1, 1, 1)$ (b) $(0, 1, 1)$

(c) $(1, 0, 1)$ (d) $(1, 1, 0)$

Figure 54: Effect of cost weights. The three weights are given as $(Length, Time, Discomfort)$.

in Figure 53c. Furthermore, if a passage gets blocked by too many agents or by the user-placed discomfort brush, some agents will take a different route, exhibiting *congestion avoidance*.

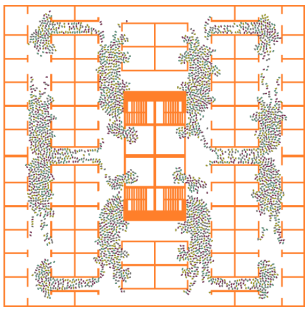
Much of the exposure of these effects and the crowd movement in general, depends on the choice of cost weights. Figure 54 provides a comparison of four extreme cases. The exits are located in the outer wall openings at the upper left and lower right. Each image shows the same scene after approximately ten seconds, but with different cost weights. The weights are given as a triple. In Figure 54a the costs are all the same. Agents exhibit the emergent phenomena mentioned above. Agents waiting at the rear of arches usually wander and turn around very much. Figure 54b shows the scene with zero cost for path length. This causes the agents to wander around much more and form loser structures. After all, it does not cost more to move, as long as the agents can get to the goal fast. In Figure 54c the opposite can be seen. The agents don't take detours anymore to evade other agents and congestion. They will wait as long as it takes to move along their chosen path. Unfortunately, some will never reach their goal, because they are blocked by other agents with similar stubbornness. In Figure 54d walls are meaningless and agents can take the direct route to the exits. These examples highlight the importance of the individual cost weight factors. Many different crowd behaviors can be simulated

by tweaking these values. For instance, if agents, waiting at a small passage, tend to move around too much, lowering the cost incurred by waiting in place, can lead to a more relaxed crowd movement.

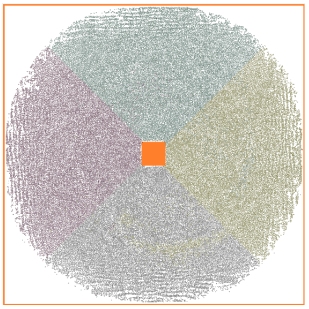
Still, very detailed individual agent behavior can not be replicated with this simulation model, as the whole scene is affected by changes to the weights. The system could be modified so that each agent group has different costs, but for controlling individuals other approaches like those using VOs are more suitable. VOs could also be added as an additional layer that controls the agent movement. In the Froblins demo of [65] the authors used this local navigation technique for avoiding other agents. This would be a useful extension to the current implementation for this thesis, because sometimes two agents get stuck at each other because the granularity and symmetry of the discretization grid, does not provide a resolution. In this case it looks as if the agents can not decide on which side to pass the other agent, so they just stop. In [65] the evasion direction is coded into the VO system. The navigation system used in this thesis already prevents agents following other agents to move up too close. This lowers the probability of intersections. Still, if two agent streams cross orthogonally, many intersections occur and need to be resolved by the collision detection process. This can be seen by temporary disabling the collision resolution in the GUI options.

Three scenarios have been simulated with the given implementation, an office evacuation (*Office*), a circular crowd movement (Circle), and a scene with paths leading through hills and other obstacles (Hills), as depicted in Figure 55. The Office scene (Figure 55a) shows an evacuation scenario with 4096 agents of radius one leaving the 256×256 grid area, for testing the Park meta movement scheme. In Figure 55b 65536 agents with a radius of 0.4 start out randomly distributed over the 256×256 map. After several minutes a circle forms and the agents move counter-clockwise around the central obstacle. This scenario has been realized by setting diagonal goal areas where agent groups meet in the image. With the Change Group movement method, agents move from sector to sector switching agent group and destination. Figure 55c shows an environment with multiple obstacles of different discomfort. The 16384 agents of radius 0.4 traverse this landscape towards the exit areas in the upper left corner. Then they respawn along the lower and right border of the 256×256 map.

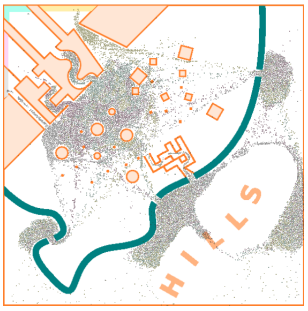
Overall, the simulation system provides a reasonable but not accurate simulation of human crowd movement. Similar systems have been used in computer games [16, 77], and if the agent density is not too high, to prevent agents from starting to wander rapidly behind arches, a believable experience can be presented. In very crowded environments, the simulation of individual agents is still suitable for non-human agents. The overall crowd flow could also be used for background scenes in movies, or for the analysis of architectural designs.



(a) Office.



(b) Circle.



(c) Hills.

Figure 55: Three simulated situations.

While no accurate simulation of human psychological is performed, the consequences of emergency exit width and location choices can be witnessed. As such, it can be used as a rapid iteration tool during the design process. Longer running and more accurate simulations could afterwards verify the results.

8.2 EFFICIENCY

To analyze the efficiency of the program, profiling information must be gathered. Four methods have been used to estimate the efficiency of the algorithm on the Radeon HD 6950, client-side time measuring, OpenCL command-queue profiling, the AMD APP Profiler tool, and memory bookkeeping.

8.2.1 *Client-side Profiling*

Client-side profiling works by storing the current wall clock time at the instant before the component that needs to be measured executes, and then comparing it with the time after the execution. A system of classes and C++ macros ensures that the profiling code does not clutter the regular execution code more than necessary. The program uses the function `clock_gettime` with the `CLOCK_MONOTONIC` parameter to get the current system time, as recommended in [3]. For each part

of the program that needs to be profiled, ten time measurements are taken and averaged afterwards. This only happens if the printing to the profiling text window is enabled.

Because the execution of OpenCL and OpenGL commands on the GPU happens asynchronous to the time measurement on the CPU, the two processors must be synchronized, in order to get valid times. Otherwise, the CPU would only measure the time it takes to issue the commands to the command-queue. Hence, before and after each region that should be measured, a synchronization command is added in the code. For this thesis `clFinish` and `glFinish` are used. These instructions guarantee that after their execution all pending OpenGL and OpenCL commands have executed on the GPU. While this synchronization allows to measure the performance of the individual components involved in the simulation, it obviously increases the overall execution time of the application. Therefore, detailed profiling of individual computation components is only performed if explicitly requested in the GUI options. Otherwise, only whole simulation steps are measured. This does not incur performance penalties because the CPU and GPU do already synchronize to present and swap rendering buffers after each simulation step and frame for double-buffered animation.

Listing 6 shows an example output with the detailed profiling option enabled, for the Office scene with 30 outer iterations.

Listing 6: Client-side execution times.

```
Client-side execution times:

Write discomfort:                  0.436167 ms
Splat densities:                   0.557921 ms
Read splat buffer:                 0.276342 ms
Compute average velocity:          0.750590 ms
Compute speed:                     0.950256 ms
Compute cost:                      1.280601 ms
Compute potential:                13.348872 ms
Compute gradient:                  0.898579 ms
-------------------------------------------------
Continuum sum:                    18.499328 ms
Move agents:                       2.897998 ms
Resolve collisions:                1.634542 ms
-------------------------------------------------
Computation sum:                  23.031868 ms
Render agents:                     1.508184 ms
-------------------------------------------------
Computation and render:           28.059699 ms
Overhead:                          8.271146 ms
Total:                            36.330845 ms
```

If the option is disabled, only the last four elements are computed and displayed. The first entry represents the time it takes to write the

information from the discomfort file, buffered in VRAM, to a copy on which the computation is performed. This is followed by the time for splatting all the agents, and the discomfort brush into the Mixed Buffer, and rendering the splat areas. Next the time for a copy of the Mixed Buffer is given. In this case the data must be copied so it can be used in OpenCL buffers. This might not be required if OpenCL images were used. The copy is executed on the GPU, which explains the short amount of time. The following five entries list the execution times of navigation components that use OpenCL kernel programs. Here, the large contribution of the potential computation becomes apparent. Below the first line is the sum of all previous computation steps used to derive the navigation data. Then the times for agent movement and collision resolution follow. Below the second line the complete sum for the agent navigation and movement is presented, and followed the time it takes to render the agent sprites at the current zoom scale. Below the last line the complete sum of the execution times for the simulation are given. The overhead it measured from the end of the computation to the next start, in order to measure the operating system and GUI involvement. The total time summarizes all results and provides the effective time used for simulation steps and visualization frames in the given computing environment. Without the detailed profiling the simulation performs twice as fast.

8.2.2 *OpenCL Command-Queue Profiling*

The profiling scheme just mentioned, allows to measure the execution times of whole program sections. For instance, it can be measured how long it takes for the Collision Resolver to fix all agent intersections, regardless of how many kernel programs are involved. For detailed kernel profiling OpenCL allows to deep profile the command-queue. Because it slows the execution down, this ability needs to be explicitly activated in the code at command-queue creation. By providing event objects to each kernel, invocation information about the runtime of the kernel can be gathered. This includes kernel start and end times. To ensure that the kernel has actually finished executing, another `clFinish` call must be used before evaluating the results.

In the program a pair of `clFinish` instructions encloses each OpenCL command that is profiled. This is necessary because AMD's implementation of the standard does not guarantee accurate kernel finish times [3]. If several kernels or commands are enqueued, they are batched by the runtime and issued in packets. The reported finish times of all commands in a batch are then the same. By calling a `clFinish` before and after the command, it is guaranteed that it is the only one being batched and executed.

This deep profiling is only performed if command-queue profiling is enabled in the configuration file, and the detailed profiling option is

selected in the GUI. Again, this is averaged over several steps. For the potential kernel ten times the outer loop count samples are collected, for all other kernels ten samples.

Listing 7 shows an example output for the command-queue profiling and the Office scene.

Listing 7: Kernel execution times.

```
Single kernel execution times:

  AverageVelocity:                0.019500 ms
  Speed:                          0.019344 ms
  Cost:                           0.030155 ms
  PotentialSolver:                0.202028 ms
  Gradient:                       0.039112 ms
  Move:                           0.311955 ms
  SortAgentsIntoBuckets:          0.022489 ms
  ResolveCollisions:              0.142722 ms
```

The time for a single execution of the potential solver kernel is not large, but the kernel must be invoked once per outer iteration, 30 times in the example. It is also noteworthy, that the kernel execution times are much lower than the times for whole components listed above. This comes from the additional setup of kernel parameters and other overhead contained in the body of the measured functions. Also the explicit processor synchronization required for command-queue profiling does cause a large performance hit. As mentioned previously in Section 5.6.2, this is especially costly for the repeated potential kernel invocation and synchronization. Hence, all these detailed profiling methods are only useful to gather information about a single component or kernel. No correlation between several parts can be analyzed based on this data. The whole program execution time decreases by a large amount if all the synchronization is disabled.

8.2.3 *AMD APP Profiler*

The AMD Accelerated Parallel Processing (APP) Profiler [2] is a tool that allows to gather additional information about a program. This includes the kernel execution times as can be gathered directly with OpenCL commands as mentioned above, but also detailed information about, memory and register usage, memory bandwidth occupation, ALU utilization, and more.

Figure 56 shows a selection of performance counter values gathered with the Profiler for the potential kernel running the Hills example. Each bar is related to a different property. The information is based on two profiling runs with the application, one with Selective Update and one with the Update All method. For the Update All scheme, the Profiler lists approximately the same results per kernel invocation for each

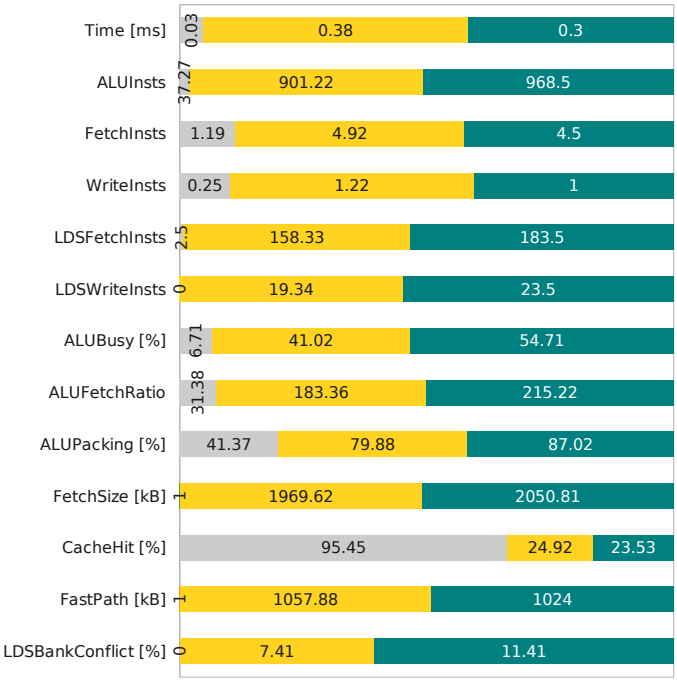


Figure 56: AMD APP Profiler information for potential kernel with Selective Update (fastest iteration in gray, slowest iteration in yellow) and Update All (teal).

outer iteration. For the Selective Update method, the results of each iteration differ. They start with short kernel execution times, when only a few tiles are active, increase over a peak execution time, where many tiles are computing solutions, until all tiles have converged, with very short execution time and hardly any computation performed. The values in gray represent the kernel run, where no tiles need to be updated because all have converged. This has the shortest execution time of 0.03 ms. The yellow values are from the longest running iteration of the Selective Update algorithm, with 0.38 ms. All other iterations have execution times that lie somewhere in between those two values. The other properties are not necessarily inside the ranges given by these two iterations. The teal colored values are from the Update All method for comparison. This iteration had an execution time of 0.3 ms.

Next, each property is explained based on the description in [3] and accompanied by a short analysis.

Time is the kernel execution time. This is the time it takes to update all tiles. The gray value has a very short time, because the whole grid has converged, and the kernel does not need to perform any inner iterations. Interestingly, the time for the Update All pass is smaller than the largest one of the Selective Update method. This can happen when all, or nearly all, tiles are active and in the

Update state. Then the kernel has to perform all the work that the Update All version has to do, plus the state management. This explains the increased time. Averaged over all outer iterations the execution time of the Selective Update scheme is smaller.

ALUInsts gives the average number of ALU instructions executed by a work-item. The yellow iteration executes less instructions than an Update All iteration. This indicates that not all tiles were active. Still, the execution time is longer. Hence, some memory transfers must have slowed the operation down.

FetchInsts and WriteInsts list the average instructions per work-item that access global memory. And as expected, the numbers in the yellow case surpass those in the teal iteration. More global memory accesses have been performed. The additional accesses come from the tile state management buffers.

LDSFetchInsts and LDSWriteInsts refer to the average local memory accesses per work-item. These counts are part of the ALUInsts. As mentioned previously, ALU and LDS instructions are grouped in the same clauses. The values in the gray iteration indicate, that the kernel does not perform any potential computations. Only the tiles and the state of its neighbors is retrieved by one work-item and shared with the other items through local memory.

ALUBusy refers to the percentage of the execution time that is spent on ALU instructions. This value should be high in order to optimally utilize the processing power of the GPU. The potential kernel only utilizes about 50% even in the Update All case. This suggests that the algorithm is not optimal for execution on the GPU, memory operations take up a large part of the process.

ALUFetchRatio provides $\frac{ALUInsts}{FetchInsts}$, and is a measure of how many ALU and local memory operations are performed in relation to fetches from global memory. This number should be large, so that the GPU can hide the global memory latency with computation.

ALUPacking with the high values around 80% indicate that the kernel compiler can fill the VLIWs with close to four ALU operations in the busy iterations. This indicates that the vectorization of the algorithm for the Radeon HD 6950 architecture is good. Still, the authors of [65] reported an ALU utilization of 98% with their shader implementation. This could suggest that additional optimizations in the kernel code or OpenCL runtime would be possible, even though their implementation differs.

FetchSize sums the total amount of memory loaded from VRAM, including reloading from caches. In the examples about 2 MB are transfered in one direction.

CacheHit gives the percentage of fetches that could be served by the data caches. About a quarter of the loads can be handled by caches in the busy kernel instantiations.

FastPath lists the number of kilobytes transfered over the fast memory access path. For this kernel all writes to global memory go over the Fast Path. The Profiler also provides a counter for the Complete Path, but it was zero in this case. The amount is approximately half of the FetchSize. Most likely this is because each work-item loads the potential and the cost at a grid location, but writes only the potential.

LDSBankConflict provides the percentage of the execution time spent on LDS bank conflicts. As mentioned earlier, the GPU can not switch processing to another wavefront or clause, if a local memory access is stalled. Thus, valuable execution time can be wasted. In the examples the amount of time that is lost to stalls is low, as is to be expected with the optimal local memory layout. The conflicts that still occur come from the additional loading of potential values along the edges of the tile.

It seems as if the value of ALUBusy is around 50% because the rest of the time is spent on LDS instructions. This coincides with the algorithm that does many local memory accesses but only few arithmetic operations. Additional performance counters list the percentage of execution time that is spent on waiting for global memory fetches. These values are less than 1%, and thus not explicitly listed here. This suggests that only few conflicts arise during global memory accesses.

8.2.4 *Memory Usage*

The application keeps track of how much global memory each component requires. Listing 8 provides the global memory occupation for an example with a 1024×1024 grid and over one million (2^{20}) agents.

Listing 8: Global memory usage.

```
Total VRAM usage. This is the sum of OpenGL texture, OpenGL
buffer, OpenCL buffer, and OpenCL image memory used by each
component in bytes.

Splatterer                          38797392
Cost Calculator                     16777216
Speed Calculator                    16777216
Potential Calculator                29360129
Gradient Calculator                 33554432
```

```
Agent Mover                      37748800
Collision Resolver               20971520
Work Groups Calculator            4194304
Work Items Calculator             4194304
Crowd Calculator                 33554432
Discomfort Renderer                   144
Density Renderer                      144
Potential Renderer                    144
Work Groups Renderer                  144
Work Items Renderer                   144
Gradient Renderer                     144
Splat Areas Renderer                  144
Cost Renderer                         144
Average Velocity Renderer             144
Speed Renderer                        144
Map Renderer                          144
Group States Renderer                 144
Agents Renderer                   1048592
Sum                             236980065 bytes = 226 MB
```

A total of approximately 226 MB is being used by this application and most of the 2 GB of VRAM on the Radeon HD 6950 is free for other purposes. This could be used for a 3D visualization of the environment. First all the computation and afterwards the rendering related components are listed. The *Work Groups Calculator* and *Work Items Calculator* provide and manage the textures for the checkerboard pattern overlays. $1024 \cdot 1024 \cdot 4 = 4194304$ bytes is the amount of memory required to store an uncompressed 1024×1024 texture with four byte-sized components per texel. Most renderers only list a few bytes, because the texture that they render is actually shared with OpenCL, and thus listed with one of the computation related components. The 144 bytes come from additional data required to render the texture (two triangles of three vertices each, with four floating-point components per vertex position, plus six 2D texture coordinates), and could alternatively be shared between all the renderers. In other applications, where no visualization of the individual buffers is required, the content of the Mixed Buffer could be overwritten to save memory. In the example this would free up 16 MB.

8.2.5 *Component Comparison*

Some parts of the simulation primarily depend on the cell count of the domain grid and others on the number of agents. Figure 57 shows how the number of cells influences the execution time of the whole navigation pipeline, containing all the steps from splatting to gradient computation. Several examples with grid sizes between 16×16 and 1024×1024 have been profiled. The work-group size is 256, and the splatting of 2^{20} agents took about 16 ms. The scene features a constant discomfort of zero and goal areas along the south and west border.

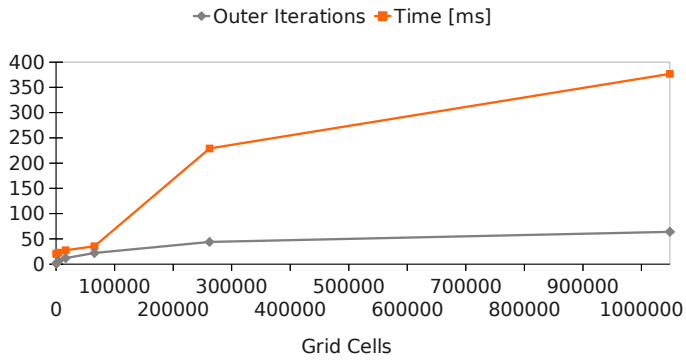


Figure 57: Execution times and outer potential iterations for different cell counts.

The agent movement is set to Change Group. The gray curve depicts the number of outer iterations that had to be set for the potential solver. These values originate from experimentations and visually judging the quality of the gradient field. If a tile has not been updated at all because the iteration count is too low, it will be visible as a black square in the gradient view. Low-quality results can be spotted as tile-sized, quadratic shapes in the visualization of the gradient directions. The required iteration count depends on the complexity of the map. Only the agent placement influences the complexity in these examples. In small maps many agents overlap, in larger maps they are more spaced-out in the environment. The appropriate choice of the number of steps can be difficult, if the gradient view changes rapidly between simulation steps. The orange curve shows the execution time. Between the grid sizes 256×256 and 512×512 a huge increase in execution time can be seen. The AMD APP Profiler does not list any significant difference in performance counters for these two cases. Still, it can be estimated that the increase memory traffic is the cause, although the stall counters don't reflect this.

Figure 58 continues the component analysis by presenting the timing results for movement (gray), collision resolution (orange), and visualization (yellow). In this case several experiments with different agent counts from 256 to $2^{20} = 1048576$ have been performed. The grid size is 1024×1024, the work-group size 256, and the agent radius 0.4. The movement shows a linear increase with agent count. This is as expected, because the agents are moved independently of each other. The visualization also increases linearly. In experiments, the time to visualize agents grows dramatically if the agent radius is increased. This is understandable, because the number of pixels that the rasterizer must produce increases with the square of the agent radius. The execution time curve for the collision resolution appears to be linear, although there is a bend at 262144 agents. In fact, the curve must reflect a function somewhere between a linear and a quadratic

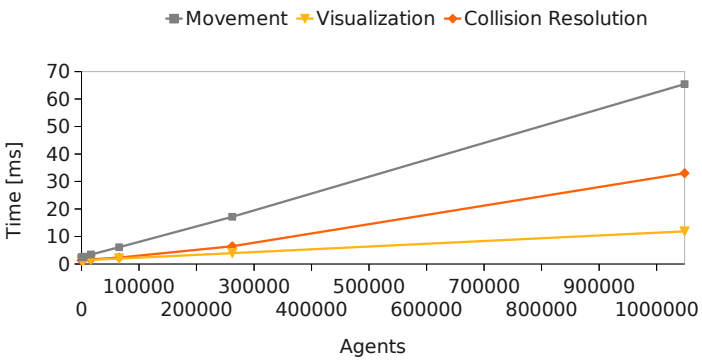


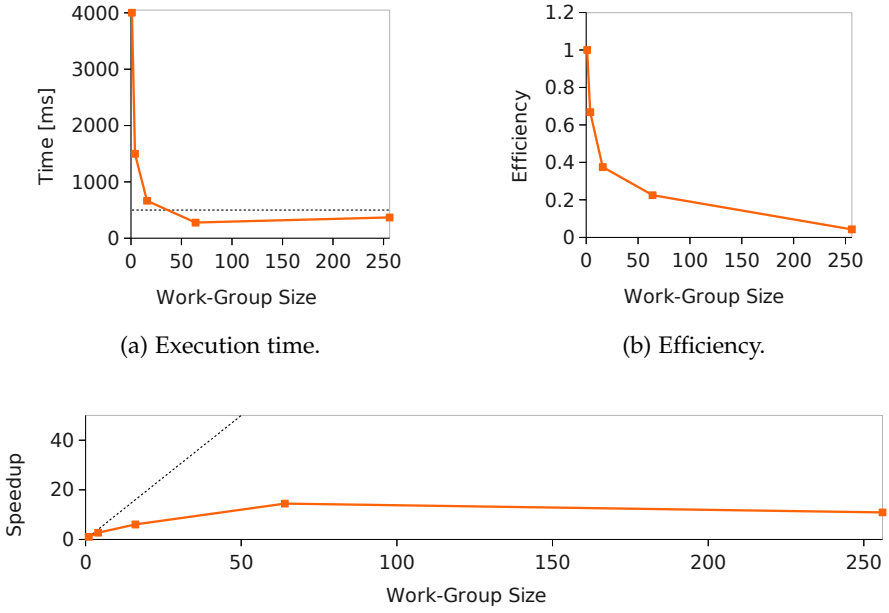
Figure 58: Execution times for different agent counts.

increase. It would be quadratic if each agent was checked against all other agents. Because of the binning method, the number is much lower. For each agent up to $8 \cdot 4 + 3 = 35$ other agents must be checked with the given radius of 0.4.

8.2.6 *Different Work-Group Sizes*

Figure 59a shows how the work-group size influences the execution time for the computational part of the simulation, from splatting to collision resolution. Again the empty map of size 1024×1024 with the south and west border goals has been used, together with 2^{20} agents to splat. The Splatterer is based on OpenGL shader programs, therefore the work-group size can not be changed for the splatting itself. All the other computational kernels are influenced by it though. The tested work-group sizes are in the range from 1 to 256. For 1D NDRanges this is just the given numbers, for 2D NDRanges these are the sizes $(1, 1)$ to $(16, 16)$. The number of outer iterations for the potential solving is 64 and has not been adjusted to the resulting tile sizes. For smaller work-group and therefore tile sizes the number of steps must be increased in order to compute a valid gradient solution. For a tile size of 1×1 approximately $2n = 2 \cdot 1024 = 2048$ steps would be required to get at least some result, where n is the side length of the grid. With four seconds the execution time is already large, as shown in Figure 59a. With the adapted iteration count it would be approximately two minutes. Still, the gradient solution is likely suboptimal and even more steps would be required.

Instead of adapting the step count, it is analyzed how the work-group size influences the execution time for a fixed amount of work. In the figure the time radically decreases until 64 work-items process a tile concurrently. This is the wavefront size of the Radeon HD 6950. Interestingly, with a larger work-group size the time slightly increases. This is probably caused by some overhead for managing several wavefronts per work-group. Section 3.2 mentions that the



(a) Execution time.

(b) Efficiency.

(c) Speedup.

Figure 59: Efficiency analysis for different work-group sizes.

graphics card processes two wavefronts per CU to hide the ALU latency. With a work-group size of 64, matching the wavefronts size, the wavefronts must be from different tiles. With a size of 256 items, they are from the same tile and additional synchronization inside the work-group has to be managed. Unfortunately a work-group with 128 items could not be tested because of the tile size constraints.

As mentioned above, for this application the outer iteration count would need to be increased for 64 work-items to provide a valid gradient solution. In experiments 130 iterations have been determined as sufficient. This raised the time to approximately 500 ms, which is more than the largest work-group size requires, as depicted by the horizontal dotted line. Consequently, 256 work-items per work-group and tile are optimal and achieve the highest performance for the application.

Figure 59c compares the different speedups resulting from dividing the execution time for a work group-size of one by the time for the work-group size, as in

$$Speedup_s = \frac{Time_1}{Time_s}.$$

It is important to note, that the execution times are not based on thread or wavefront counts, but on work-group sizes. The GPU still issues several wavefronts to run on the 22 CUs even if the work-group size is one. Nonetheless, the curve provides an interesting insight into the behavior. Linear speedup is indicated by the dotted black line.

The speedup actually achieved is much lower. For constant work no improvement can be seen beyond 64 work-items, where a speedup of 14.4 is achieved. With 256 items the speedup lies at 10.9.

The curve in Figure 59b shows the resulting efficiency, computed as

$$Efficiency_s = \frac{Speedup_s}{s}.$$

Again, this is linked to the work-group size and not the number of processors. The figure depicts how the efficiency rapidly falls off with increased work-group size. For 64 work-items it is 23%, with 256 items only 4%. Still, the parallelized algorithm with the largest work-group size brings the fastest performance for the required number of iterations, in this application.

Unfortunately, the program currently does not provide correct results when run on AMD's CPU implementation of the OpenCL standard. The problem seems to lie in the interoperability with OpenGL. Although this means that no meaningful performance comparison can be given, it can be said that the program ran much slower on an AMD Phenom II X4 920.

Furthermore, it has been attempted to speed up the simulation of the agent movement, by only updating the navigation information every other step, and distributing the outer potential iterations over several frames. This behaves badly if the agents move fast enough to reach another cell before the information is updated. In this case, the cell will suddenly update based on old information and the agents exhibit rapid turns. Hence, such an optimization can only work if the cell size is large in relation to the agent speed, as in the demo of [65].

8.3 MAP MAKING

The program requires a configuration file, based on Qt's settings system, that sets several parameters and also refers to the image files that define the map area. An empty default configuration file can be generated by running the program without any arguments. The file then must be specified as the single argument to the application.

The discomfort image file describes the discomfort distribution in the scene. The program only uses the red color component's intensity to load the values. For convenience different intensities of white can be used in the image manipulation program. Areas that have a value of larger or equal to 0.99 in the red channel will be walls with infinite discomfort in the simulation. The easiest way to draw walls is by using a full white intensity. Corridors must be at least two grid cells wide to allow the algorithm to derive a path.

The exit image file contains the goal areas for the four agent groups. This is encoded in the four color channels, red, green, blue, and alpha. If a pixel contains a value of zero in the red channel, this means, that agents in group Groupo will be heading towards that

location. Any value larger than zero will be initialized to infinity for the potential solver. The same holds for the other three channels and agent groups. To paint such exit maps, GIMP [76], or any equivalent image manipulation program, can be used as follows

1. Create a new square image that has a side length which is a multiple of the work-group side length set in the configuration file.
2. Select Colors > Components > Decompose.
3. Select Red Green Blue Alpha (RGBA) as the color model and tick the option to decompose to layers.
4. This results in four layers that can be painted with black and white color to create the exit maps for each group.
5. When done, select Colors > Components > Compose.
6. Again, select RGBA as the color model and assign the layers to the channels as required for the scenario.
7. Save the image, for instance as a Portable Network Graphics (PNG) file.

The configuration settings contain an entry for the map image file. This image can be viewed as an overlay in the application. For instance, showing a scenery from the top or a floor plan. Its resolution can be higher than those of the other images, but it also must be a square image.

9 RELATED WORK

Although related work has been mentioned throughout this document, no overview has been provided yet. Table 3 summarizes related work in the field of crowd simulation and compares it with this thesis. The table mentions the relevant platform, execution mode, and some example applications mentioned in the works. If no information is available, the corresponding entry is marked by a hyphen. The update rates of the algorithms are given in FPS. Here it means updates per second.

Continuum Crowds presented by Treuille et al. [81] is the central work around which the others have been developed. Their flow-field implementation for crowd simulation runs on the CPU. Although simulating a respectable number of agents, the performance of the algorithm on the given platform was insufficient for computer games or similar interactive applications.

The works by Jiang et al. [37, 36] extend the continuum approach to multi-storied buildings. This allows to simulate staircases and footbridges, which is not currently possible with the implementation for this thesis. Their simulations don't run in real-time, but they are able to playback the recordings of their offline simulation. They mention the intent to parallelize the technique for performance improvement and future publications.

GPU Continuum Crowds by Luke Zarko and Mark Fickett [47] is a parallel implementation that runs on the GPU using Cg and OpenGL. As mention in Section 5.6.1 they were confronted with artifacts in their solutions, potentially related to similar problems encountered during the development of this thesis. This caused them to add additional precautions which lowered the performance of their algorithm.

March of the Froblins by Shopf et al. [65, 66] is another parallel implementation running on the GPU. The authors use HLSL and Direct3D for their implementation. On contemporary hardware the algorithm ran in highly interactive rates. Although the mentioned update rate of 20 FPS seems to be the one of the rendering algorithm. When studying the output of the navigation algorithm, it appears to run in a slower rate. As mentioned earlier, this is not a problem because the Froblins move slowly in relation to the size of the grid cells. In comparison, this thesis uses the same update rate for rendering and movement simulation. Because of the problems encountered during the attempt to implement their algorithm (see Section 5.6.1), it can be speculated that their algorithm also suffered from artifacts. This theory would support their choice of environment, which is very open

without any walls and tight restrictions. It is important to note that this is highly speculative. Still, the algorithm used in this thesis is able to deal with tight environments containing walls, causing congestion, typical for office buildings. It remains unclear, if March of the Froblins has the same capability. Nonetheless, their visual representation of the environment and agents is far superior to the one presented in this thesis. Noteworthy is also their additional implementation of VOs, which would be a useful addition to the simulation in this thesis.

Although Champandard [16] mentions that Supreme Commander 2, StarCraft 2, and Heavy Rain use flow-field-based methods, it is unclear how closely related they are to Continuum Crowds. The agents in Supreme Commander 2 [24] exhibit lane formation, which this thesis is also capable of, but no additional information or numbers could be found. It is also unclear whether the implementation runs solely on the CPU or if some GPU acceleration is used. A similar situation exists with StarCraft 2 [73, 32, 16]. Not much information is available. Still, the simulation looks impressive, and at least 800 agents are possible. This includes the simulation of combat, not covered at all in this thesis. According to [16] the implementation for Heavy Rain [77] runs in parallel on the SPEs of the Cell processor.

The secretive nature prevalent in game development prevents the release of information and implementation details. It is unlikely that any of the game developers would provide the source code to their implementations. Similarly many researches don't provide their implementations together with easy access to source code and documentation, even though free hosting platforms make it easy to release information to the general public. This is where this thesis can make a significant contribution. By making all the source code and documentation available on the web, easily accessible to anyone with access to the Internet (see Chapter 1).

In the overall comparison, the application developed for this thesis can simulate more agents in interactive rates, than mentioned in related works. Compared with March of the Froblins, it seems to provide a more stable algorithm, allowing more restricted environments and higher agent speeds. Still, it would benefit from the addition of the VO algorithm. From the known implementations, this is the first one to use the combination of OpenGL and OpenCL for the computation of the navigation data.

Table 4 shows a number of works related to algorithms used for collision detection and compares the used platforms. Harada [27] uses shader programs and a concept similar to binning to detect collisions between objects. It requires multiple rendering passes per object. Grand [25] uses CUDA and explains an approach to parallel radix sorting. Harris et al. [28] define an alternative implementation of parallel radix sorting with CUDA. Liu et al. [46] provide a very sophisticated approach to collision detection with CUDA. It served as

WORK	PLATFORM	MODE	EXAMPLES
Continuum Crowds [81]	Intel Pentium 3.4 GHz	Sequential	10000 agents, 60 × 60 grid, 5 FPS
			2000 agents, 120 × 120 grid, 5 FPS
... complex environment(s) [37, 36]	Intel Core2 Duo E8300 2.83 GHz	Sequential	-
GPU Continuum Crowds [47]	NVIDIA GeForce 8800 GTX, Cg, OpenGL	Parallel	8000 agents, 256 × 256 grid, 15 FPS
March of the Froblins [65, 66]	AMD Radeon HD 4870, HLSL, Direct3D	Parallel	65000 agents, 256 × 256 grid, 20 FPS[a]
			3000 Froblins, 256 × 256 grid, 20 FPS[b]
Supreme Commander 2 [24, 16]	-	-	-
StarCraft 2 [73, 32, 16]	-	-	800 agents
Heavy Rain [77, 16]	Cell	Parallel	-
This thesis	AMD Radeon HD 6950, GLSL, OpenGL,	Parallel	1048576 agents, 1024 × 1024 grid, 2.5 FPS
	OpenCL		65536 agents, 256 × 256 grid, 30 FPS

Table 3: Comparison with related crowd simulation works.

a Apparently this is the update rate for rendering, the navigation simulation seems to run at a lower rate.
b Same as above.

WORK	PLATFORM
Real-Time Rigid... [27]	NVIDIA GeForce 8800 GTX, shaders
Broad-Phase Collision... [25]	NVIDIA GeForce 8800 GTX, CUDA
Parallel Prefix Sum... [28]	NVIDIA GeForce 8800 GTX, CUDA
Real-time Collision Culling... [46]	NVIDIA Tesla C1060, CUDA
This thesis	AMD Radeon HD 6950, OpenCL

Table 4: Comparison with works related to collision detection.

the basis for the Sweep and Prune implementation mentioned in this thesis, but also lists a number of additional optimizations.

Related FIM solvers are listed in Table 5. The scientific works rooted in medicine and oil exploration [33, 34, 35] deal with 3D domains. This is in contrast to the 2D domains commonly used for crowd simulation. Additionally, the solvers used in these applications don't terminate after some fixed number of iterations. Instead, the computation stops if the required accuracy of the result has been achieved. In [35] the CPU synchronizes with the GPU to control the algorithm's execution. This is required in order to terminate the computation. In this thesis the number of iterations is controlled by the user, and determined empirically. Therefore no CPU interference is required and the algorithm can be implemented solely on the GPU in OpenCL. All this makes performance comparisons less meaningful, therefore no execution times have been listed. For the interested reader, the works contain explicit information about execution times. It is noteworthy that no other OpenCL implementation of the FIM was encountered during the development of this thesis, but no dedicated search was performed.

WORK	PLATFORM	EXAMPLES
A fast eikonal... [33]	NVIDIA GeForce 7900 GTX, Cg, OpenGL	$256 \times 256 \times 256$ grid
A fast Iterative... [34]	NVIDIA Quadro FX 5600, CUDA	$256 \times 256 \times 256$ grid
Interactive Visualization... [35]	NVIDIA GeForce 8800 GTX, CUDA	$256 \times 256 \times 100$ grid
This thesis	AMD Radeon HD 6950, OpenCL	1024×1024 grid

Table 5: Comparison with FIM solvers unrelated to crowd simulation.

10 CONCLUSION

This thesis presented and compared a number of different techniques used in crowd simulation. A subset of these methods has been selected to run as a parallel program on the Radeon HD 6950 graphics card using the standardized OpenCL and OpenGL frameworks. The successful implementation is highly configurable and allows to study the continuum-based approach of simulating crowd movement on the GPU in various scenarios.

The FIM implementation developed for this thesis runs completely on the GPU without any significant CPU involvement, managed primarily by OpenCL kernel programs. Furthermore, two collision detection algorithms have been realized on the GPU. The superior one remains available inside the demo. The application is capable of simulating the movement of up to a million agents in near real-time. The quality of the simulation is sufficient for a rough approximation of the movement of human crowds, and for agents in computer games. The many visualizations make the program suitable for studying and experimentation. The public release of this document and the program's source code allows others to benefit.

Further research could focus on the support of platforms with multiple GPUs. The simulation of multi-storied buildings, as in [36, 37], would be an interesting extension to the current navigation system. Also, as the local obstacle avoidance based on the navigation information alone is insufficient, and causes agents to get stuck on each other in a few cases, adding VO-based local obstacle avoidance, as in [65], would improve the results. This would also reduce the need for small cell sizes in relation to the agent radii. The system could further regard vectorized walls for more precise collision avoidance. Additionally, the visualization could be improved with a custom 3D animation system.

Even though problems with the OpenCL implementation have been encountered during the development, complex parallel algorithms can be expressed and realized with this rather young standard. The quality of the implementation provided by hardware manufacturers will surely improve, as has been shown by the constant upgrades released during the past months. This makes consumer-grade hardware an affordable option for the scientific community.

BIBLIOGRAPHY

[1] akring. Disabling Collision Between AI, May 2010. URL http://www.gameplaydev.com/2010/05/disabling-collision-between-ai/. [Online; accessed 11-August-2011].

[2] AMD. AMD APP Profiler, June 2011. URL http://developer.amd.com/tools/AMDAPPProfiler/Pages/default.aspx.

[3] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, August 2011. URL http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/default.aspx.

[4] AMD. HD 6900 Series Instruction Set Architecture, June 2011. URL http://developer.amd.com/sdks/amdappsdk/assets/AMD_HD_6900_Series_Instruction_Set_Architecture.pdf.

[5] Chris Angelini. Radeon HD 6970 And 6950 Review: Is Cayman A Gator Or A Crock?, December 2010. URL http://www.tomshardware.com/reviews/radeon-hd-6970-radeon-hd-6950-cayman,2818.html. [Online; accessed 11-August-2011].

[6] Bobby Anguelov. Optimizing the A* algorithm, May 2011. URL http://takinginitiative.net/2011/05/02/optimizing-the-a-algorithm/. [Online; accessed 11-August-2011].

[7] Andrew Armstrong. Animating in a Complex World: Integrating AI and Animation, 2009. URL http://aarmstrong.org/notes/game-developers-conference-2009-notes/animating-in-a-complex-world-integrating-ai-and-animation. [Online; accessed 11-August-2011].

[8] Attila16. Assassin's Creed: Climbing in Damascus, April 2008. URL http://www.youtube.com/watch?v=voz6Xwwhezc. [Online; accessed 11-August-2011].

[9] b4t3. Lord of the Rings ROTK Extended Edition Calling for Grond, February 2009. URL http://www.youtube.com/watch?v=NzktLKinyCM&feature=player_detailpage#t=23s. [Online; accessed 11-August-2011].

[10] Michael Booth. The Official Counter-Strike Bot by Michael Booth, October 2008. URL http://aigamedev.com/insider/presentations/official-counter-strike-bot/. [Online; accessed 11-August-2011].

[11] Michael Booth. The AI Systems of Left 4 Dead, 2009. URL http://www.valvesoftware.com/company/publications.html.

[12] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004. URL http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf.

[13] Bryan Catanzaro. OpenCL™ Optimization Case Study: Simple Reductions, August 2010. URL http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study-Simple-Reductions_4.aspx. [Online; accessed 16-August-2011].

[14] Alex J. Champandard. There's a Hole in Your NavMesh, Dear Zombie, October 2009. URL http://aigamedev.com/open/articles/hole-navmesh-dear-zombie/. [Online; accessed 11-August-2011].

[15] Alex J. Champandard. Are Waypoint Graphs Outnumbered? Not in AlienSwarm!, July 2010. URL http://aigamedev.com/open/reviews/alienswarm-node-graph/. [Online; accessed 11-August-2011].

[16] Alex J. Champandard. This Year in Game AI: Analysis, Trends from 2010 and Predictions for 2011, January 2011. URL http://aigamedev.com/open/editorial/2010-retrospective/. [Online; accessed 11-August-2011].

[17] A. Crooks, C. Castle, and M. Batty. Key challenges in agent-based modelling for geo-spatial simulation. *Computers, Environment and Urban Systems*, 32(6):417–430, November 2008. URL http://dx.doi.org/10.1016/j.compenvurbsys.2008.09.004.

[18] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[19] DKTronics70. "City Life" RealityIV ENB HDR GTA IV Video, July 2010. URL http://www.youtube.com/watch?v=idERA_o6rwo. [Online; accessed 11-August-2011].

[20] Bryan Dudash. GPU Gems 3 - Chapter 2. Animated Crowd Rendering, April 2011. URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch02.html. [Online; accessed 11-August-2011].

[21] dylanwinter1. starlings on Otmoor, February 2007. URL http://www.youtube.com/watch?v=XH-groCeKbE. [Online; accessed 11-August-2011].

[22] Paolo Fiorini and Zvi Shillert. Motion Planning in Dynamic Environments using Velocity Obstacles. *International Journal of Robotics Research*, 17:760–772, 1998.

[23] fripoutapioka. foule à shibuya, July 2007. URL http://www.youtube.com/watch?v=ioaZ4tZoARA. [Online; accessed 11-August-2011].

[24] Gas Powered Games. Supreme Commander 2 - Flowfield Pathfinding, February 2010. URL http://www.gametrailers.com/video/flowfield-pathfinding-supreme-commander/62420. [Online; accessed 11-August-2011].

[25] Scott Le Grand. GPU Gems 3 - Chapter 32. Broad-Phase Collision Detection with CUDA, April 2011. URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html. [Online; accessed 11-August-2011].

[26] Stephen. J. Guy, Jatin Chhugani, Sean Curtis, Pradeep Dubey, Ming Lin, and Dinesh Manocha. PLEdestrians: A Least-Effort Approach to Crowd Simulation. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, 2010. URL http://gamma.cs.unc.edu/PLEdestrians/.

[27] Takahiro Harada. GPU Gems 3 - Chapter 29. Real-Time Rigid Body Simulation on GPUs, April 2011. URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch29.html. [Online; accessed 11-August-2011].

[28] Mark Harris, Shubhabrata Sengupta, and John D. Owens. GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA, April 2011. URL http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html. [Online; accessed 23-August-2011].

[29] Peter Hart, Nils Nilsson, and Bertrfam Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968. URL http://dx.doi.org/10.1109/TSSC.1968.300136.

[30] Justin Hensley. Throughput Computing: Hardware Basics, 2008. URL http://developer.amd.com/documentation/videos/InsideTrack/assets/hensley-gpu-architecture.pdf.

[31] H. Hildenbrandt, C. Carere, and C.-K. Hemelrijk. Self-organised complex aerial displays of thousands of starlings: a model, 2009. URL http://www.citebase.org/abstract?id=oai:arXiv.org:0908.2677.

[32] HuskyStarcraft. 800 ZERGLINGS, March 2010. URL http://www.youtube.com/watch?v=oZJXgA4aL6Q. [Online; accessed 11-August-2011].

[33] Won-Ki Jeong and Ross T. Whitaker. A fast eikonal equation solver for parallel systems. In *In SIAM conference on Computational Science and Engineering*, 2007. URL http://www.cs.utah.edu/~wkjeong/research.html.

[34] Won-Ki Jeong and Ross T. Whitaker. A Fast Iterative Method for Eikonal Equations. *SIAM Journal on Scientific Computing*, 30 (5):2512–2534, 2008. URL http://www.cs.utah.edu/~wkjeong/research.html.

[35] Won-Ki Jeong, P. Thomas Fletcher, Ran Tao, and Ross T. Whitaker. Interactive Visualization of Volumetric White Matter Connectivity in DT-MRI Using a Parallel-Hardware Hamilton-Jacobi Solver. *IEEE Transactions on Visualization and Computer Graphics*, 13:1480–1487, 2007. URL http://www.cs.utah.edu/~wkjeong/research.html.

[36] Hao Jiang, Wenbin Xu, Tianlu Mao, Chunpeng Li, Shihong Xia, and Zhaoqi Wang. A Semantic Environment Model for Crowd Simulation in Multilayered Complex Environment, 2009. URL http://vr.ict.ac.cn/paper_selected/vrst09_jh/vrst09_jh.htm.

[37] Hao Jiang, Wenbin Xu, Tianlu Mao, Chunpeng Li, Shihong Xia, and Zhaoqi Wang. Continuum crowd simulation in complex environments, 2010. URL http://www.sciencedirect.com/science/article/pii/S0097849310000804.

[38] David Kanter. AMD's Cayman GPU Architecture, December 2010. URL http://www.realworldtech.com/page.cfm?ArticleID=RWT121410213827&p=1. [Online; accessed 11-August-2011].

[39] Ioannis Karamouzas and Mark Overmars. Simulating the Local Behaviour of Small Pedestrian Groups. In *VRST '10: Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology (to appear)*. ACM, 2010. URL http://people.cs.uu.nl/ioannis/groups/.

[40] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall, 2 edition, April 1988. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0131103628.

[41] Khronos OpenCL Working Group. The OpenCL Specification, June 2011. URL http://www.khronos.org/registry/cl/.

[42] Ron Kimmel and James A. Sethian. Optimal Algorithm for Shape from Shading and Path Planning. *Journal of Mathematical Imaging and Vision*, 14:2001, 2001. URL http://www.cs.technion.ac.il/~ron/publications.html.

[43] Lucas Kovar, Michael Gleicher, and Frederic Pighin. Motion Graphs, 2002. URL http://www.cs.wisc.edu/graphics/Gallery/kovar.vol/MoGraphs/.

[44] Ris S.C. Lee and Roger L. Hughes. Prediction of human crowd pressures. *Accident Analysis & Prevention*, 38(4):712 – 722, 2006. URL http://www.sciencedirect.com/science/article/pii/S000145750600008X.

[45] Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. Motion Fields for Interactive Character Animation, 2010. URL http://grail.cs.washington.edu/projects/motion-fields/.

[46] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. Real-time Collision Culling of a Million Bodies on Graphics Processing Units. *ACM Trans. Graph.*, 29:154:1–154:8, December 2010. URL http://graphics.ewha.ac.kr/gSaP/.

[47] Luke Zarko and Mark Fickett. GPU Continuum Crowds. URL http://www.seas.upenn.edu/~cis565/index-2007.htm.

[48] Annie Mole. London Journeys - Value of Time, June 2010. URL http://london-underground.blogspot.com/2010/06/london-journeys-value-of-time.html. [Online; accessed 11-August-2011].

[49] MoNiiXMedia. Left4dead 2 Zombie climbing HUGE LAMPS !, October 2009. URL http://www.youtube.com/watch?v=gcwkoHeAMHA. [Online; accessed 11-August-2011].

[50] Mikko Mononen. Temporary Obstacle Progress, April 2011. URL http://digestingduck.blogspot.com/2011/04/temporary-obstacle-progress.html. [Online; accessed 11-August-2011].

[51] Mikko Mononen. Digesting Duck - Blog about game AI and prototyping, July 2011. URL http://digestingduck.blogspot.com/. [Online; accessed 11-August-2011].

[52] Mikko Mononen. recastnavigation - Navigation-mesh Construction Toolset for Games, July 2011. URL http://code.google.com/p/recastnavigation/. [Online; accessed 11-August-2011].

[53] Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C. Lin. Aggregate dynamics for dense crowd simulation. *ACM Trans. Graph.*, 28(5):1–8, 2009. URL http://gamma.cs.unc.edu/DenseCrowds/.

[54] Nokia Corporation. Qt - Cross-platform application and UI framework, August 2011. URL http://qt.nokia.com/. [Online; accessed 30-August-2011].

[55] Clément Pêtrès, Yan Pailhas, Yvan Petillot, and Dave Lane. Underwater Path Planing Using Fast Marching Algorithms. *IEEE Oceans Europe 2005*, June 2005. URL http://www.eps.hw.ac.uk/~ceeyrp/WWW/Research/Papers.html.

[56] prgmetro. Crowds in Moscow metro, August 2007. URL http://www.youtube.com/watch?v=GCZKXHsZo68. [Online; accessed 11-August-2011].

[57] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. In *Computer Graphics*, pages 25–34, 1987. URL http://www.red3d.com/cwr/boids/.

[58] Rockstar Games. L.A. Noire Official Launch Trailer, July 2011. URL http://www.rockstargames.com/lanoire/videos/7211/. [Online; accessed 11-August-2011].

[59] Elisabeth Rouy and Agnès Tourin. A Viscosity Solutions Approach to Shape-From-Shading. *SIAM Journal on Numerical Analysis*, 29(3):867–884, 1992. URL http://dx.doi.org/10.1137/0729053.

[60] runevision. Semi-Procedural Animation for Character Locomotion, August 2008. URL http://www.youtube.com/watch?v=0ea-CXxBEYg. [Online; accessed 11-August-2011].

[61] Matthew Schuerman, Shawn Singh, Mubbasir Kapadia, and Petros Faloutsos. Situation agents: agent-based externalized steering logic. *Comput. Animat. Virtual Worlds*, 21:267–276, May 2010. ISSN 1546-4261. URL http://dx.doi.org/10.1002/cav.v21:3/4.

[62] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, pages 97–106. Association for Computing Machinery, 2007. URL http://www.idav.ucdavis.edu/publications/print_pub?pub_id=915.

[63] James A. Sethian. A Fast Marching Level Set Method for Monotonically Advancing Fronts. In *Proc. Nat. Acad. Sci*, pages 1591–1595, 1995. URL http://math.berkeley.edu/~sethian/2006/Explanations/fast_marching_explain.html.

[64] James A. Sethian and Alexander Vladimirsky. Ordered Upwind Methods for Static Hamilton–Jacobi Equations: Theory and Algorithms. *SIAM J. Numer. Anal.*, 41:325–363, January 2003. URL http://dx.doi.org/10.1137/S0036142901392742.

[65] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU, 2008. URL http://developer.amd.com/samples/demos/pages/froblins.aspx.

[66] Jeremy Shopf, Christopher Oat, and Joshua Barczak. GPU Crowd Simulation, 2008. URL https://a248.e.akamai.net/f/674/9206/0/www2.ati.com/misc/siggraph_asia_08/GPUCrowdSimulation_SLIDES.pdf.

[67] Bethesda Softworks. Bethsoft Tutorial Navmesh, March 2011. URL http://geck.bethsoft.com/index.php/Bethsoft_Tutorial_Navmesh. [Online; accessed 11-August-2011].

[68] Stephen, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. ClearPath: highly parallel collision avoidance for multi-agent simulation. In *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–187, New York, NY, USA, 2009. ACM. URL http://gamma.cs.unc.edu/CA/.

[69] Nathan R. Sturtevant. Memory-Efficient Abstractions for Pathfinding. In *AIIDE'07*, pages 31–36, 2007. URL http://webdocs.cs.ualberta.ca/~nathanst/pathfinding.html.

[70] Avneesh Sud, Russell Gayle, Erik Andersen, Stephen Guy, Ming Lin, and Dinesh Manocha. Real-time Navigation of Independent Agents Using Adaptive Roadmaps. *ACM Symposium on Virtual Reality Software and Technology*, 2007. URL http://gamma.cs.unc.edu/crowd/aero/.

[71] SundownSFA. Realtime Procedurally Animated Spiders, October 2007. URL http://www.youtube.com/watch?v=I1P_B65XW4I. [Online; accessed 11-August-2011].

[72] Ben Sunshine-Hill and Norman I. Badler. Perceptually Realistic Behavior through Alibi Generation. 2010. URL http://repository.upenn.edu/hms/114/.

[73] sweYoda2. StarCraft 2 – Path Finding Bug, October 2010. URL http://www.youtube.com/watch?v=DtyBfdfBYvc. [Online; accessed 11-August-2011].

[74] technology review. The Problem of Predicting Crowd Crush, August 2010. URL http://www.technologyreview.com/blog/arxiv/25624/. [Online; accessed 11-August-2011].

[75] Pierre Terdiman. Radix Sort Revisited, January 2000. URL http://codercorner.com/RadixSortRevisited.htm. [Online; accessed 23-August-2011].

[76] The GIMP Team. GIMP – The GNU Image Manipulation Program, 2011. URL http://www.gimp.org/. [Online; accessed 03-September-2011].

[77] TokyoDomeSF4. Heavy Rain 021 Evil playthrough Lexington Station 1 of 1, February 2010. URL http://www.youtube.com/watch?v=cM2xc27Ha2s. [Online; accessed 11-August-2011].

[78] TOP500.Org. TOP500 List - November 2000 (1-100), August 2011. URL http://www.top500.org/list/2000/11/100. [Online; accessed 11-August-2011].

[79] Torus Knot Software Ltd. OGRE – Open Source 3D Graphics Engine, August 2011. URL http://www.ogre3d.org/. [Online; accessed 30-August-2011].

[80] Torus Knot Software Ltd. Ogre Forums – New InstanceManager: Instancing done the right way, August 2011. URL http://www.ogre3d.org/forums/viewtopic.php?f=4&t=59902&sid=955aaa40626641527cc9cdfb34749a61. [Online; accessed 30-August-2011].

[81] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum Crowds. *ACM Trans. Graph*, 25:1160–1168, 2006. URL http://grail.cs.washington.edu/projects/crowd-flows/.

[82] Adrien Treuille, Yongjoon Lee, and Zoran Popović. Near-optimal Character Animation with Continuous Control, 2007. URL http://grail.cs.washington.edu/projects/graph-optimal-control/.

[83] John N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40:1528–1538, 1995.

[84] Tyrouo. Shibuya's crossing, August 2007. URL http://www.youtube.com/watch?v=9qx31jHBd1g. [Online; accessed 11-August-2011].

[85] Valve Corporation. Navigation Meshes, July 2011. URL http://developer.valvesoftware.com/wiki/Navigation_Meshes. [Online; accessed 11-August-2011].

[86] Jur van den Berg, Ming C. Lin, and Dinesh Manocha. Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation. In *IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*, pages 1928–1935. IEEE, 2008. URL http://gamma.cs.unc.edu/RVO/.

[87] Micah Villmow. OpenCL performance issues, December 2009. URL http://forums.amd.com/devforum/messageview.

cfm?catid=390&threadid=123857. [Online; accessed 11-August-2011].

[88] Micah Villmow. Global memory bandwidth, July 2011. URL http://forums.amd.com/devforum/messageview.cfm?catid=390&threadid=152978. [Online; accessed 01-September-2011].

[89] Ko-Hsin Cindy Wang and Adi Botea. Tractable multi-agent path planning on grid maps. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 1870–1875. Morgan Kaufmann Publishers Inc., 2009. URL http://portal.acm.org/citation.cfm?id=1661445.1661745.

[90] Eric W. Weisstein. "Domain". From MathWorld–A Wolfram Web Resource, 2011. URL http://mathworld.wolfram.com/Domain.html. [Online; accessed 11-August-2011].

[91] Wikipedia. Crowd simulation — Wikipedia, The Free Encyclopedia, 2011. URL http://en.wikipedia.org/wiki/Crowd_simulation. [Online; accessed 11-August-2011].

[92] Wikipedia. Collision detection — Wikipedia, The Free Encyclopedia, 2011. URL http://en.wikipedia.org/wiki/Collision_detection. [Online; accessed 26-August-2011].

[93] Wikipedia. Domain (mathematical analysis) — Wikipedia, The Free Encyclopedia, 2011. URL http://en.wikipedia.org/wiki/Domain_%28mathematical_analysis%29. [Online; accessed 11-August-2011].

[94] H. Yeh, S. Curtis, S. Patil, J. van den Berg, D. Manocha, and M. Lin. Composite agents. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08. Eurographics Association, 2008. URL http://gamma.cs.unc.edu/CompAgent/.

[95] Hongkai Zhao. A FAST SWEEPING METHOD FOR EIKONAL EQUATIONS, 2004. URL http://www.math.uci.edu/~zhao/homepage/research.html.