Parallel implementation of large scale crowd simulation

Thomas A. Grønneløv*

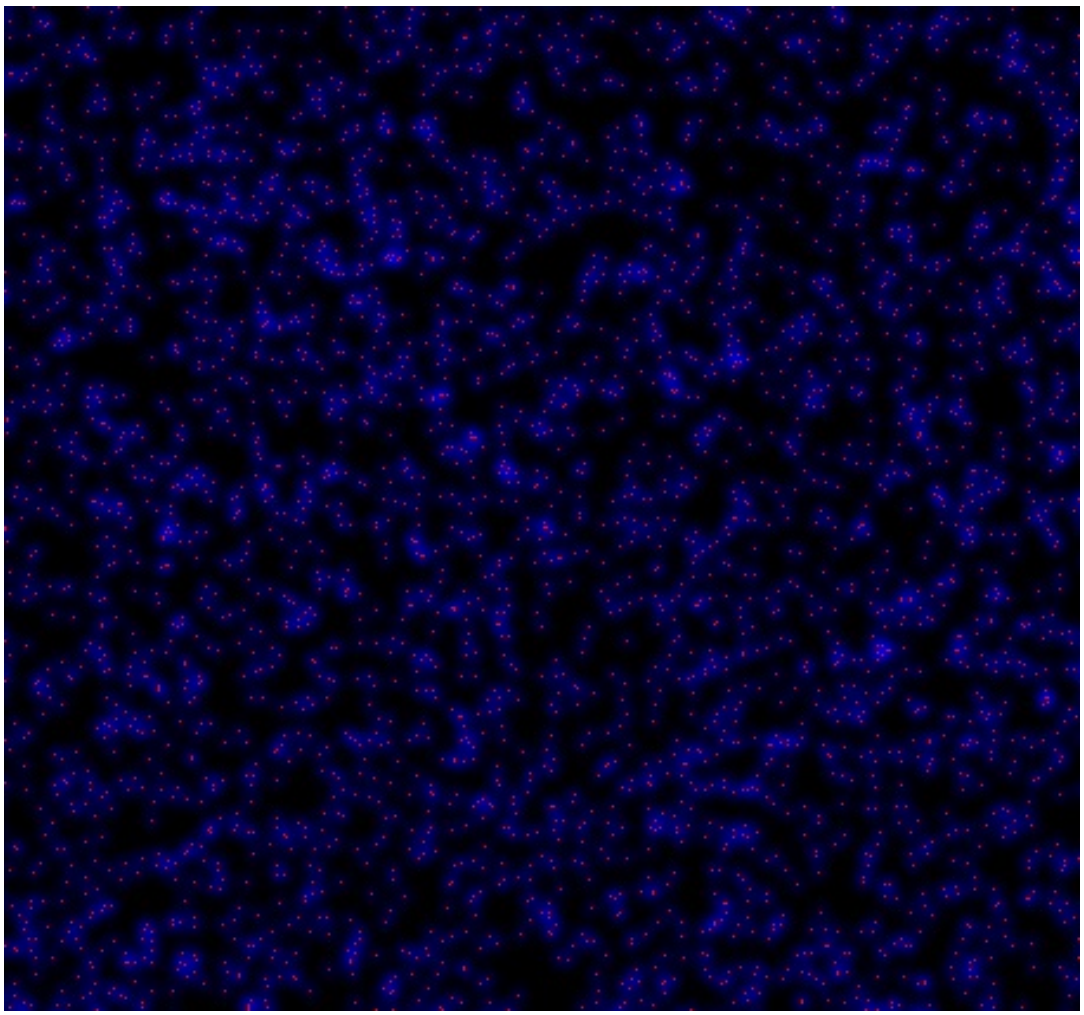

Figure 1: Crowd Simulation

*e-mail:Tag@Greenleaf.Dk

Abstract

When designing entrances and exits for a stadium, a transit station, ships or similar structures, it is important to optimize the flow of people which in turn requires that flow to be simulated. In addition to that, the police, the military, firefighters and others who have to deal with various types of crowds either to contain and control them or to evacuate them, are more and more using simulated training in their education and they also have a need for virtual crowds.

Various tools exist to perform crowd simulation, but they are generally not real-time and can be quite time consuming for a large scale simulation. The national football stadium, Parken, has room for over 40.000 persons so a full scale simulation dealing with an evacuation scenario will be computationally expensive. When used for interactive virtual training, real-time is an actual requirement and not just a convenience.

In order to make large scale simulation possible in real-time, or almost real-time, we will implement a model for human crowd behavior on a parallel processing platform using CUDA (a c based API for parallel processing) and evaluate the accuracy of the model along with its ability to simulating full scale scenarios with very large crowds.

Accuracy will be evaluated by comparison with various text book crowd behaviors such as the formation of ”lanes” and ”vortices” when two or more crowds moving in different directions meet.

We contribute with a clear relation between fluid simulation and human crowds where the controlling forces inside a crowd are connected with their physical fluid counterparts. The implementation results in a a method with a time complexity that depends on the size of the domain and not on the number of agents.

Contents

1 Conventions used in this paper

A large group of people is called a crowd, while a smaller group is just called a group.

An individual person in a group or crowd is called an agent. When talking about human crowds the term ”person” is used interchangeably with ”agent”.

The domain is the area containing the crowd and the simulation. It will generally be a room, a building or part of a city.

A goal is a position that the agents want to reach. It can be exits from or entrances to buildings or simply just attractive positions.

Human flocking

For training of emergency workers, military and police there is a need for realistic real-time crowd simulation. For designers of areas and structures with a large flow of people there is the same need, although not real-time. There exist texts such as the green, purple and primrose guides, further details in [Still(2000)], which describe in details the possible problems with crowds and the recommended

For those reasons software simulating crowds is a valuable tool. In the design phase, crowds can be simulated and problems can be identified and corrected immediately.

2 Flocking and crowds - previous work

Previously the behavior of flocks of birds or schools of fish were considered the result of a very complex system which somehow allowed the thousands of individual animals to coordinate their movement so that they avoided collision among themselves as well as against external objects and fleeing predators. The underlying ”programming” was apparently very complex.

In 1986 Craig Reynolds described his initial BOID[1] model[Reynolds(1987)] describing flocks of birds. The model employed only three steering forces among the individual ”birds”. Each force was based on the closest ones of the other BOIDS. The forces were ”separation”, do not get too close to others, ”alignment”, steer into same direction as the others and ”cohesion”, stay near others. With only those forces, the flocking behavior was easily seen.

Later in 1999 Craig Reynolds expanded his original work by adding general steering behavior[Reynolds(1999)] which adds goals, path following and some strategy. This helps making BOIDS more human-like and it introduces obstacle avoidance by directly changing the motion vector of a BOID away from stationary objects using the ”steering behavior”. The forces described in this work are the forces that we will implement, although from another perspective.

Dijkstra et. al. published a new method of crowd simulation[Dijkstra *et al.*(2000)Dijkstra, Timmermans, & Jessurun] used in simulation of pedestrians. It was different from the BOID model in that it was a grid based cellular automate. This method is simpler, but the the larger the grid cells, the less accurate and the smaller the cells, the more computational expensive the model is. The method can also have problems with irregular obstacles due to the regular cells. We take a somewhat similar approach in how we managed the simulation domain, though we will not make it entirely a discrete grid but more of a hybrid.

The previously mentioned methods were all local models in that the BOIDS/agents each reacted to their immediate neighborhood and were controlled by attractive or repulsive forces exerted by that neighborhood.

[1] Bird-like object

Simulation based on static (global) potential fields[Kirchner & Schadschneider(2002)] were used to test the evacuation of buildings, but failed to deal with changes in the environment. While the scene may be static, the crowd moving in it is not, and that crowd is also an obstacle.

The alternative is to consider the agents part of a whole, much like the individual particles in a fluid, and perform physical flow calculations on that whole. In [Clements & Hughes(2004)], a set of differential equations is derived and used to simulate the evolving potential field of a moving crowd.

Later this evolving field was used for simulation[Treuille *et al.*(2006)Treuille, Cooper, & Popović] by adding agent particles and letting them be moved by the field. This work was shown to have many of the properties seen in real crowds and our approach is quite similar to what they did, in that we will also consider crowds a fluid and calculate the fields describing the crowd and its motion.

A PhD thesis by Keith Still [Still(2000)] takes a very detailed look at crowd dynamics and goes to great lengths to calibrate simulations with actual observed behavior on a large scale. The simulation results clearly show that one can in fact create very accurate simulations of human crowd behavior. This demonstrates that crowd simulations are not just for fun and games but can be used to make life or death decisions about various structures and evacuation procedures.

3 Human flocking

The behavior of a human flock, henceforth a crowd, is quite different from that of a flock of birds or a school of fish. There is generally more planning involved and even though each member of the crowd, each agent, is influenced by the position and motion of nearby agents, there is a stronger individuality. However intelligent humans are compared to a school of fish, we are still all animals and are all subjected to the same fundamental programming. Research into human behavior and human crowd behavior [Krause & Dyer(2008)] has shown that we behave very much like flocking animals if the large scale individual goals are removed and that we are physically designed to mimic the behavior of other agents in a crowd by specific mimicking neurons in the brain. In other words, the basic flocking behavior stems from our own brains wanting to mimic the behavior of those around us. Human flocking behavior thus consists of a basic animalistic part, just as with the BOIDS, and a more complex purpose driven part. When people have been placed in large rooms [Krause & Dyer(2008)] and asked to walk around without any real purpose apart from just moving and not bumping into each other, then they behave like flocking animals.

3.1 Basic behavior

This is BOID-like behavior. Agents do not overlap and will be pushed away from each other when they are very close by separating forces. Only when there is a high external pressure will two agents stand right next to each other. The magnitude of the repulsive force is largely culture dependent as described in [Chattaraj(2009)] and so is the attractive force which makes individual agents stay in the vicinity of each other through cohesion forces. When agents move about, they tend to move more or less in the same direction as their neighbors, thus keeping the change of neighbors low.

3.2 Individual goals within a crowd

When considering a scene, such as a building evacuation, filling a stadium or a crowded transit station, there will generally by a limited number of goal positions which the agents try to move towards. They do so by finding an optimal, under certain constraints, path towards the goal and then move along this path. While navigating the path

they will have to consider other agents and try to stay close to the optimal path, maintain a high velocity and avoid bumping into other agents as well as fixed obstacles.

An alternative to trying to reach a specific goal position is to get away from somewhere. This could be a fire, a madman with a bomb or a wild animal.

The optimal path is described by Kieth [Still(2000)] as ”path of lest effort”. This means that the optimal path may very well be longer than the shortest path, but it will still cost the person the least effort by avoiding congested areas or avoid moving against the local motion of the crowd.

3.3 Level of agent's knowledge

When looking for the optimal path, one has to consider every possible influence along the way. This is obviously a problem for agents with limited knowledge. They may know the static domain, but they generally do not know exactly what is around the next corner. Therefore they may opt for an apparently optimal path and then turn the corner and run straight into a highly congested area where they can only move very slowly and with great effort.

A real agent in a human crowd will generally be able to gather information from observing the behavior of other agents. This advanced knowledge is a problem when doing real time simulation and it will often be simplified.

There are however things which are not know and cannot be inferred. Unless the agent has an (unrealistic) dynamic map of the domain showing every static and dynamic obstacle, or can see everything from its current position, then it will make decisions based on imperfect knowledge and those decisions will often be more or less flawed as seen in figure 2 where one of two apparently equal paths is blocked at a location which is out of sight.

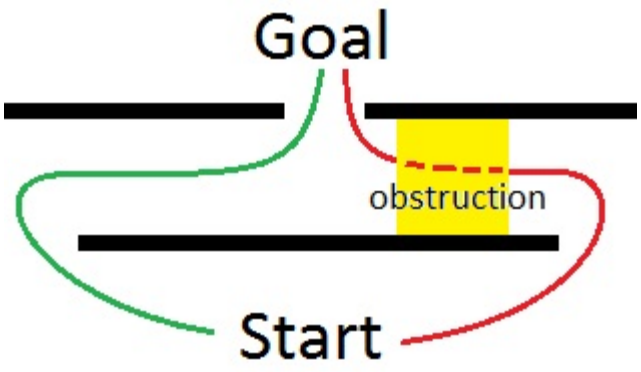

Figure 2: Two paths but one is obstructed at a position which is out of view

Reasoning about effectiveness of a given behavior An actual person will make decisions based on its observations of the surroundings and knowledge about static paths, but it will also learn from the results of its actions. If an apparently good strategy just doesn't work, then the person will realize that the strategy for some reason is flawed and then make a new one. This is not something that we have found in the literature, and we will unfortunately also not implement it here.

3.4 Implemented subset of crowd behavior

This section will look at a few specific complex behavioral patterns that we will attempt to support in our implementation - in addition to the basic behavior.

3.4.1 Lane formation

This is also sometimes referred to as fingers. When two groups of agents collide head on, one might expect chaos to ensue when the front agents collide with each other and cannot move around to the sides. After the initial collision collisions would then repeat over and over while the groups pass each other. What does in fact happen, as described in [Still(2000)], is that quite quickly the

front agents are pushed in between each other, but the following agents do not repeat this collision when they enter the other group. Instead they follow the agents in their own group which happens to move in the same general direction as they want to themselves. This behavior opens up a desirable path behind an agent which others follow. Shortly after the initial collision, the groups will have formed into variable size lanes and the agents will generally move at close to their desired top speed.

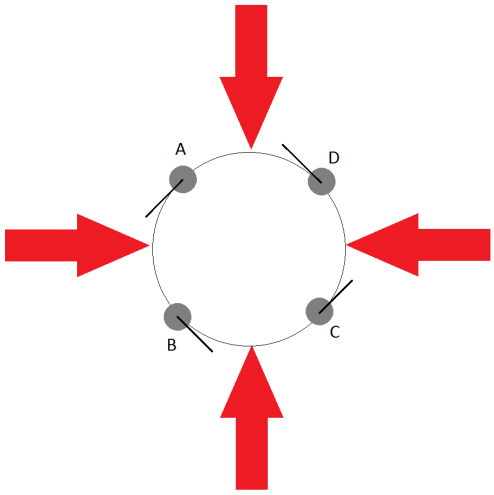

Figure 3: This agent will constantly be pushed by the pressure force from the four groups in a way so that the agents movement vector, seen as a thin line, is turned counter clockwise. unless its own path following force is strong enough to push it through the opposing groups it will keep circling the center of collision

3.4.2 Vortices

When groups of agents meet at steep angles they tend to form vortices. Rather then just move straight through each other, and forming lanes, they begin to spin slightly around the center of the collision as mentioned in [Treuille *et al.*(2006)Treuille, Cooper, & Popović]. Individual agents may even perform complete revolutions, as seen in figure 3, if the pressure is high enough from a large number of groups all crossing at once.

3.4.3 Braess's paradox

Braess's paradox is an example of behavior seen in crowds but not in fluid dynamics. This is one of the reasons that simple fluid dynamics is not enough in itself to accurately describe crowd mechanics. The paradox itself is concerned with a routing problem in which it worsens the congestion to add more routes. It is described in general terms relating to routing in [Wainwright(2007)] and more related to crowds in [Still(2000)].

In a crowd context you can describe it with a group of people trying to exit a room through two doors, as seen in figure 4. Each half of the agents will select their closest door and exit without much congestion. Then a third door opens and most agents will see that this door is closer and will then attempt to exit through it. Before all agents would exit through two doors while they will now mostly exit through one door.

In this simplistic setup most real human crowds would have enough common sense to avoid the congestion and exit through all three doors, but if the upper and lower doors were moved a little farther away and if the room was on fire and the crowd panicked, then that common sense would quickly be lost.

For this to be simulated, the crowd would have to employ a less than optimal pathfinding which does not take local congestion into consideration. If the pathfinding is perfect and is looking at congestion as a parameter then the agents will be very clever and avoid Braess paradox all together.

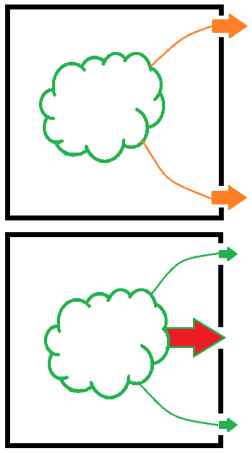

Figure 4: Braess's paradox in action. First a crowd tries to leave a room through two doors. Then a third door opens in the center and now most agents in the crowd will try to leave through that door. Congestion builds up and the exit rate drops.

4 Massive parallelism and CUDA

Some algorithms consist of a large number of sequential steps. Step 2 cannot be started before step 1 is completed etc. Those algorithms have no use for more than one processor. There is only one small job to work on at a time.

Other problems are highly parallel and consist of a number of smaller jobs which can be completed in any order. This means that if more than one processor is available, then each can handle a subset of the many small jobs and thereby the entire process can be speeded up. How much of a speedup depends on the overhead associated with partitioning the entire problem into a set of smaller problems and gathering the sub solutions to one whole solution.

Luckily one example of a problem which is parallel in nature is that of simulating a crowd. Each individual agent in a crowd will consider the current state of its surroundings and decide on what to do next. Then every agent will move at the same time. Each such observation and action will depend only on the current state of the system and not on what other agents are planning to do right now.

This is generally how complex systems are simulated. A current state is evaluated and the next state is calculated based on the current. A system is described through a set of state variables which for a crowd at time t might be $state(t) = [x_0\ y_0\ vx_0\ vy_0\ x_n\ y_n\ vx_n\ vy_n]$ describing position and velocity for n agents. In order to advance from $state(t)$ to $state(t+\Delta)$, Δt in the future, a set of calculations will have to be done, but the important part is that that all calculations will depend on $state(t)$ which is already available. This way no calculation will depend on the result of any other calculations and $state(t+\Delta)$ can be calculated in any order.

Amdahl's Law Amdahl's Law (1) describes the expected performance gain from changing serial to parallel code. This is based on the fraction of serial code that can be parallelized and the number of processors it can be distributed on. This equation does not take into considerations how much time will be used to split the problem and gather the solutions. Often having a large number of processors will mean having a architecture where the processors and their local memory is separated by high latency and therefore this overhead can be substantial. The larger the sub problems are, the less the comparative overhead from sending problems and solutions around there system will be.

When the parallel implementation is on a modern graphics card, or a super computer cluster, the value of N can be considered a very high number essentially removing the term P/N. Now the speedup depends only on P, the fraction of code being parallelized. It is clear from the equation that increasing the value of N does little for the speedup while increasing P has a much greater influence. If P is 0.90 then the speedup will be 10 times for an infinite N.

This is interesting in that even with an infinite number of processors and code which is 90% par-

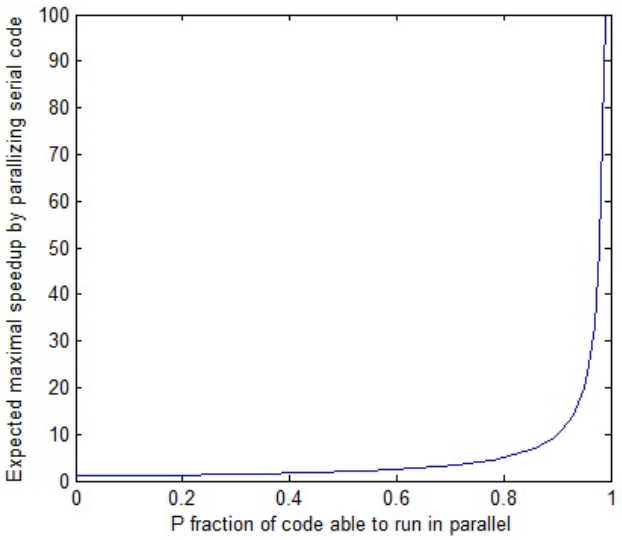

Figure 5: Amdahls law plotted as a function of P and N

allelizable, the speedup is only 10 times. In figure 5 the development of speedup as a function of P given an infite number of N is shown. The conclusion is easy to make and it is that it is good to have many processors, but the important thing is to make heavy calculations entirely parallel. If not then all the processors in the world does little good.

For our implemented crowd simulation both the number of processors and the fraction of parallel code is very high. All the simulation code will be parallel and only initialization and visualization will be serial.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Amdahls equation showing the expected performance gain by parallelizing previously serial code

4.1 Graphics card processing and CUDA

While super computer clusters are not available to most users, there is an alternative which is very common indeed. This is the Graphics Processing Unit GPU and its fast memory on modern graphics cards. The graphics card which currently holds the processing record is the ATI Radeon HD 5800 which reportedly delivers 2.700 GFlops. Ten years ago that would earned *one* such card a third place on the list of 500 fastest super computers [Top500Org(1999)]. Graphics cards of today generally support running in groups of more than one. This can boost performance further and Nvidia is also marketing their Tesla computation boards which are essentially graphics cards with more memory and no graphics output.

CUDA is an acronym for Compute Unified Device Architecture and was released by Nvidia in 2007. It is an API which allows easy access to the computing power on the graphics cards. Ever since the first graphics cards supported user defined shader code, those shaders have been used for computing unrelated to graphics, but the code would have to be masked as graphics by doing render calls from some graphics API. With CUDA the code is written in a subset of the c language and is merely compiled for and executed on the GPU as would it be on any other processor.

CUDA brings more control to the programmer. A shader cannot read from arbitrary addresses in memory but generally has to use texture lookups to take in the data it is going to work on. The concept of shared memory (section 4.6) is not exposed to shaders at all, and this memory is the fastest on the entire device. With a shader there was no way to synchronize the individual threads executing the shader code. In CUDA this is possible for certain groups of threads. Finally CUDA allows faster copying of data to and from the device than what was possible with shaders where everything had to pretend to be graphics.

CUDA only works on a CPU (emulation for de-

bugging) or Nvidia GPUs and lack support for other vendors. Currently two alternatives to CUDA are being rolled out. They are OpenCL and Direct Compute. OpenCL will work with all graphics cards supporting directX10 or higher while Direct Compute requires DirectX11 capable hardware.

Compute capability There are different levels of CUDA capable devices. Currently there exist only devices with compute capability 1.0 to 1.3. There are subtle differences between the different compute capabilities; mostly devices of higher capabilities place fewer restrictions on the code. Coalesced memory transactions section 4.8.4 on higher devices for example are easier to set up right since data alignment and access pattern is more free.

This project was done using a GeForce 9800 GT which has compute capability 1.1, as do most ordinary graphics cards currently, and the text is written with that level of capability in mind. Please refer to the CUDA programming guide [Nvidia(2007)] for further details.

4.2 CUDA applications

A CUDA application consists both of code running on the CPU, acting as the host, and code running on the GPU, being the device. The host code has to initialize CUDA, move relevant data to the device and call the device code. There are two types of device code, namely kernel code and pure device. The kernel is callable from the host while pure device functions are only callable from the device.

The host code can be ordinary c or c++[2] and can perform other tasks relevant to the application apart from running CUDA kernels. Device code on the other hand is a subset of c and does have a few restrictions which are generally not very limiting.

- No recursion
- Cannot declare static variables inside functions
- Does not support a variable number of arguments - default arguments are supported though
- Pointers to functions are not allowed
- Kernel functions must return void
- Kernel functions can only receive 256 bytes of arguments when called
- Limited to 2 million instructions

4.3 Device hardware

A device consists of a number of multiprocessors, hereafter called MP as seen on figure 6. Each MP is capable of running several threads at the same time without any time slicing. On a MP there are registers and a local memory with very low latency - as low as zero due to how instructions are executed.

There are also two caches. One for constant lookup, and one for texture lookups. The device is the GPU meaning that the device is one large processor. The global memory is off chip and is not considered part of the device itself. Each MP has direct access to global device memory and cached access to same memory when used for constants or textures.

4.4 Threads and blocks

Threads are collected in larger groups called blocks and the blocks are laid out in a grid. The total number of threads is defined by specifying how many blocks is used and how many threads there is in each block.

Blocks are conceptually laid out in a 1D, 2D or 3D pattern and threads are laid out in a similar way inside the blocks. This layout has nothing to do

[2] At the time of writing the c++ integration is considered an alpha feature and is not reliable

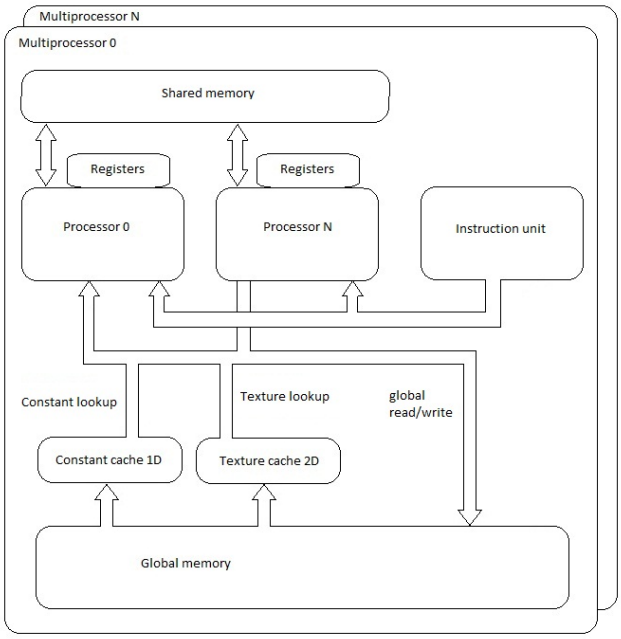

Figure 6: One device contains global device memory and N multiprocessors. Each multiprocessor contains shared memory and one instruction decoder as well as M processing cores and constant and texture caches. Each processing core contains its own registers

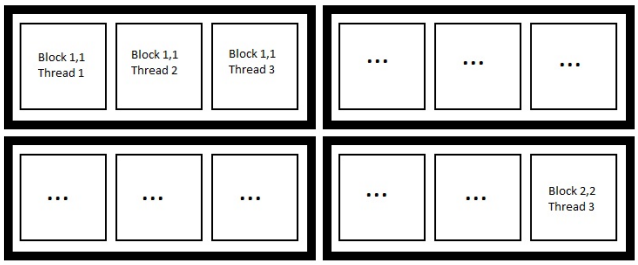

Figure 7: 4 blocks each containing 3 threads. The blocks are laid out in 2D while the threads are laid out in 1D. Note that real world blocks cannot contain only 3 threads!

with how they are positioned in hardware, it is only to make it easier to map from a problem domain to the thread domain. If the problem is simulation of heat diffusion in a 2D domain, then it would seem natural to place the blocks and threads in a 2D arrangement so that blocks and threads have 2D identifiers mapping easily to the 2D space of the problem.

Thread blocks The reasoning behind partitioning into blocks is that threads inside the same block are able to be tied strongly together. They can synchronize their execution and they can work on the same low latency shared memory. Threads from different blocks cannot synchronize in any way. While they can read from and write to the same variables in global memory, doing so will give rise to all sorts of race conditions or require the use of special simple atomic operations.

Block and thread IDs Inside the device code it is important to know exactly which of the many threads is the current one in code. This is done by looking at the predefined variables `threadId` and `blockid` section 4.4 as well as `gridDim` and `blockDim`. With that information it is easy to see exactly where in the grid the thread is located so that it can figure out exactly what part of the problem it is supposed to work on and where it should store its results.

While it is beneficial to cooperate among threads

of the same block, blocks can not be arbitrarily large since each MP has limited resources and a block cannot be shared among MP. For this reason, and some other performance considerations explained later, the code is generally divided into several blocks.

The threads in a block are grouped into so called warps which are smaller groups of 32 threads each. The reasoning behind this is that a MP can execute several threads at the same physical time. Each MP has a single instruction decoder and several processing cores. This means that the same instruction can run on different data at the same time. This is known as SIMD, single instruction multiple data, and as long as every thread in a warp performs the same operation at the same time, as seen in figure 8, they can do it simultaneously. If they do in fact diverge due to a conditional branch then they will not execute in parallel but will be serialized, as seen in figure 9 which is explained further in section 4.8 dealing with performance.

4.5 Thread synchronization

Only threads in the same block can synchronize. This is done by calling `__synchthreads` which is a thread barrier. No thread can pass it before all threads in the same block have arrived. `__synchthreads` is only allowed in conditional code if all threads in a block are guaranteed to execute the code. This is to prevent deadlocks with some threads waiting at the barrier and others never arriving.

It should be noted that while threads in the same block have the means to cooperate, it is still up to the programmer to ensure that there are no race conditions. If two threads read and write the same location in memory then the result is undefined and if two threads perform atomic operations such as increment on the same location the operations will execute as expected but the order in which they execute is undefined.

4.6 Memory

The graphics card has a GPU which contains several multiprocessors MP. Each MP has memory that is very close to the processing cores. This memory is very fast but it is also quite limited in size. On the card, but outside the GPU, there is a large DRAM memory. This memory is quite slow in comparison, but it is also plentiful On-chip memory is roughly 100 times faster than off-chip memory.

Those two types of memory are split into several subtypes.

- Registers - Multiprocessor memory, 8192 32 bit registers shared among all threads on a multiprocessor. They are very fast taking no extra clocks to access.
- Shared memory - Multiprocessor memory, very fast but has a limited size of 16 kB per processor
- Global memory - Device DRAM, large memory. This memory is not cached and is considered slow.
- Constant memory - Device DRAM, this memory is cached on the GPU and read only. It is limited to 64 kB.
- Thread local memory - Device DRAM, smaller uncached region of 16kB per thread. Registers spill into this memory when needed.
- Texture memory - Device DRAM, this compares to constant memory in that it is read only and is cached on the GPU. Where constant memory had an ordinary linearly address based cache, texture memory employs a 2D cache which is better suited to take advantage of 2D locality.

The host code has access to all DRAM memory except thread local memory.

4.7 Textures

Global memory can be used as textures and read through special texture fetches. A texture fetch is different from ordinary memory access in that the texture unit will, if so desired, perform indexing conversion, data value conversion and interpolation as well as a cache. A texture can be accessed by indexes from 0.0 to 1.0 or from 0 to width-1 and height-1. A byte value of 0xff can be scaled so that it maps to 1 or 0xff and texture lookups in between texels can be the linearly interpolated values of the closest texel neighbors.

Furthermore the texture unit can make the texture coordinates clamp to the edges or wrap around and repeat. oundary conditions are easily implemented using clamped coordinates by simply writing the appropriate values into the outer most texels.

4.8 Performance considerations

While code written for parallel execution on a CPU can pretty much be moved directly to the GPU using CUDA, the performance would not greatly increase unless the difference in hardware is taken into consideration. In [Nvidia(2009)] a very thorough walkthrough of performance considerations is presented.

4.8.1 Keep occupancy high

A MP has 8.192 registers available for all its threads. There can be 768 active threads active at the same time. If all threads are active and each one uses 10 registers, then 7.680 registers are used. If the threads require 11 registers, then fewer can be active at the same time and occupancy drops.

Additionally, if 5 thread blocks, each consisting of 128 threads, are loaded, that will use 640 threads or 84% of the maximum. Blocks cannot be partially loaded. It is all or nothing.

This way the register use as well as the size of the thread blocks can affect how effectively the thread blocks can be packed on the MP and how much ”space” is wasted.

4.8.2 Branching within warps

The multiprocessor executes warps of 32 threads at the same time using SIMD. This means that the instructions, though not the arguments, executed by each of the 32 threads should be identical. When they are, they run at the same time, but when the thread code has split into different code paths due to conditional branching, they cannot execute simultaneously. The MP handles this by taking each branch sequentially and disabling the threads that should not execute the current code. When the branches later converge, the threads will again run in parallel. If a warp is written so that all 32 threads run different paths then it will execute a single thread at a time and do this 32 times leading to a poor performance. It is therefore quite important to ensure that threads in the same warp branch as little away from each other as possibly. If threads in different warps branch differently, then that is not an issue. Each warp will run its turn and run its threads in parallel. This is something that can be exploited in the code since it is possible to calculate warp id based on thread id.

4.8.3 Global or local memory

Global memory is the only memory accessible from the cost and it is therefore this memory that contains the actual job and the eventual result, but it is also very slow. One should always try to avoid using the global memory when possible.

The way this is generally done is by organizing the code in such a way that threads in the same block work on more or less the same subset of the entire problem. This way the first task for the threads in a block is to fetch the relevant data from global memory once and for all and then let all threads work on that data. While they work, they will not access global memory and when they are done

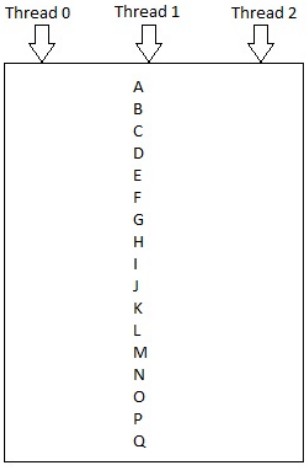

Figure 8: When all threads in a warp executes the same code, the threads will run in parallel at the same physical time. The operations are identical, only the data is different among threads

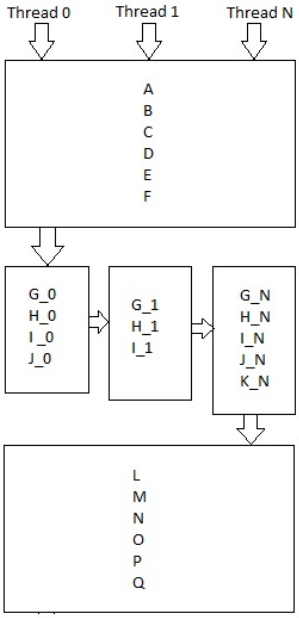

Figure 9: When the threads in a warp branch away from each other, then their different operations will be serialized and run one after another. Only after the threads merge into the same code path again will the threads resume the parallel execution

they can write the locally stored results into global memory.

4.8.4 Coalesce memory transactions

A single read or write operation to or from global memory can move up to 128 bytes if every *consecutive* thread access *consecutive* addresses with the *correct alignment* of the first address. If it is the case, then the threads perform a collective read-/write, and if it is not then every thread will perform its own read/write which again affects performance negatively.

The memory being accessed must satisfy certain conditions. Every thread in a half warp must access data of the following size.

- 4 byte words which results in one 64 byte transaction
- 8 byte words which results in one 128 byte transaction
- 16 byte words which results in two 128 byte transactions

Additionally the data needs to be aligned correctly and abide the following rules.

- All 16 words of data must be contained within a single continuous memory block starting and fill it completely.
- The continuous memory block must start at an address which is a multiple of its size. Thus N byte blocks must start at address 0,N,2N,...,kN
- Threads in the half warp must access the words in sequence $thread_n$ accesses $word_n$

4.8.5 Hide memory latency

When a warp accesses memory, there is a delay before the transaction completes. During this delay the warp is blocked. This allows other warps to execute in the mean time. They will eventually also

access memory and block which lets other warps execute and so on. Eventually the first warp will have completed its transaction and will be ready to run again.

What this means is that with a single warp the latency will have a profound effect, but with many warps the delay can be spent performing calculations.

4.8.6 Performance summary

Local memory Local memory is fast, global is slow, unless it has been cached. The local memory can be thought of as an on chip cache which the user is free to fill as he sees fit.

Branching Every warp of 16 threads should follow the same code path 4.8.2. If it is known that some threads need to follow one path while others follow another then they should be laid out so that threads of the same path are in the same warps.

Size of structures If every thread needs access to a structure 10 bytes large, then memory can be saved by aligning the structs with start addresses offset by 10 bytes, but that would prevent coalescing 4.8.4. It would therefore be better to waste 6 bytes following every struct thus making the structs start every 16 bytes and support coalesced memory transactions.

Have enough threads While it may sometimes be simpler to have a few complex threads it helps hide the fact that memory is slow when there are enough threads on a multiprocessor so that one thread can pause on a memory transaction while others carry on doing calculations.

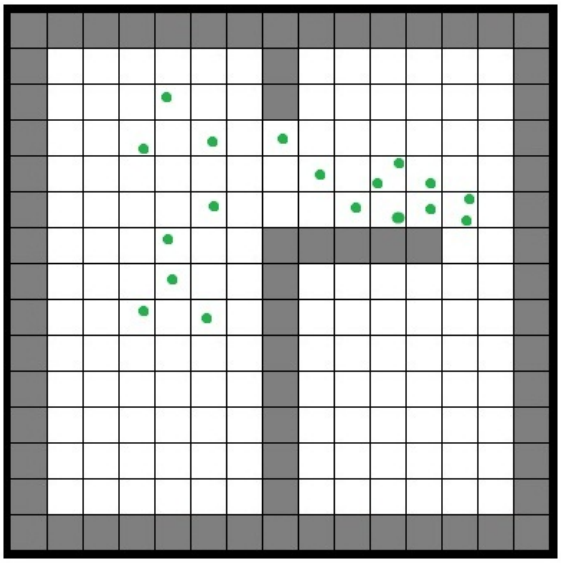

Figure 10: Domain is divided into tiles. White tiles are passable while gray ones are walls. Each agent exist in one tile but has a real number value position within the domain.

5 Implementation

The simulation will take place in a rectangular domain which is discretizised into a large number of square tiles as seen in figure 10. Each may or may not be large enough to hold multiple agents at the same time[3]. The domain always has a wall at the boundary cells to help generalize certain parts of the code.

Though the domain is partioned into discrete cells, the position of the agents are real values numbers. Agents cannot travel outside the domain and the inside of the domain will be made up of empty tiles which the agents can occupy and tiles with wall in them which agents may not occupy.

An agent is defined by the following parameters.

- Position, a 2D real valued vector which is always inside the domain.
- Velocity, a 2D real valued vector describing velocity and direction of the motion. The ve-

[3]This depends on how unwilling agents are to pack together

locity goes from 0 to the agents maximal velocity.

- Goal, an integer index defining which goal among a set of goals this agent seeks. A value of -1 means the agent has no goal.
- Maximal velocity, the maximal velocity this agent can move at. This value may be equal for all agents or it may be selected randomly at initialization.
- Inertia, a real valued number which acts as mass in calculating accelerations $a = \frac{F}{inertia}$. This value may or may not be the same for all agents. It can be seen as a combination of mass over strength and laziness.

Simulation Simulation is done by numerical integration with Explicit Euler as seen in (2). The state variable S is a vector describing the combined *dynamic* state variables of every agent $S = [x_1, y_1, vx_1, vy_1, ..., x_n, y_n, vx_n, vy_n]$. Δt is the time step used and $S'(t)$ is the first derivative of the state variables at time t i.e. $[vx_1, vy_1, ax_1, ay_1, ..., vx_n, vy_n, ax_n, ay_n]$. Explit Euler integration is not a very accurate numerical integration scheme but it is very simple to implement and as Δt goes to zero, so does the truncation error. We are not interested in very large time steps since we will visualize the simulation step by step and the system is not a very stiff one.

Walls An agent cannot be in a tile that holds a wall. There are however no forces keeping the agent from entering the walls. If an agent after a simulation step is found to be inside a wall, then it will simply be moved back out as seen in figure 11. The agent is at 1 before the simulation step. It is at 2 after the step and is then moved out to 3 to abide by the do-not-enter-a-wall constraint. The component of the agents motion vector which points into the wall is set to zero.

The reason walls are not implemented as forces keeping agents out is that an agent should not have

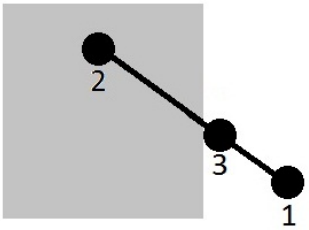

Figure 11: An agent penetrating a wall will be moved back against its movement vector until it is no longer inside the wall

a problem with moving right next to a wall but should never be inside the wall. This would require a repulsive force which was zero outside the wall and infinite inside the wall. This would mean that the force should be a discontinuous step function going from zero to infinity. Such functions will make the simulated system ”stiff” and very hard to work with.

$$S(t+\Delta t) = S(t) + S'(t)\Delta t \tag{2}$$

5.1 Crowds as fluids

As mentioned in section 2 and in [Treuille *et al.*(2006)Treuille, Cooper, & Popović] a crowd can *to a certain extent* be handled by variations over methods designed for fluid simulation. We will let us inspire by just that and the selected fluid simulation model is Smoothed Particle Hydrodynamics SPH which is covered in detail in [Liu & Liu(2003)] and [Gingold & J.(1977)]. Refer to those sources for detailed information.

The fluid based model is attractive in that there exist a clear mathematical tool set for fluids. While a BOID-model, where the controlling forces are described in more abstract terms, can give the same results, we feel that a fluid model with some support from pathfinding will lead to a clearer definition of the crowds motivation and action.

What we want from the fluid method is the derivative S' of the systems state variable i.e. the forces

acting on the agents due to the influence of the other agents.

A very brief summary of the SPH elements which we will be using is, when put in crowd perspective, the following.

SPH A crowd is considered a fluid. Each agent is a macroscopic fluid particle. The agents do have specific point positions, but their physical boundaries are smoothed so that they are not point impulses in the simulation domain but rather Gaussian smoothed points centered at the agent's position.

Smoothing by convolution Convoluting an impulse with any other function will result in that function translated to have its origin located at the position of the impulse. In other words, convoluting an impulse with, for example, a Gaussian function gives the same result as drawing a Gaussian with mean at the impulse. It is just another way of drawing the Gausian indirectly using convolution. For n impulses we write n impulses and then perform one convolution, thereby drawing the sum of n Gaussians. As seen in figure 14 two impulses and one convolution results in the sum of two Gaussians. This is not a method we have seen used in the literature, but it seems like the best way of doing it when using CUDA.

Ordinarily SPH implementations do not discretize the domain and explicitly smooth with a kernel. Instead they sum the contribution over all other particles and weigh each other particle with the distance based kernel function - at the location of all other particles. The reason we did not follow that path is twofold. Such an implementation has a timecomplexity which depends on the number of agents and it requires some support code like spatial hashing to make it reasonable fast. The primary reason, however, is that we do not only need to evaluate the field at the location of the agents. We need the entire (discretezised) field to be evaluated for the dynamic pathfinding, so in the end there is little choice.

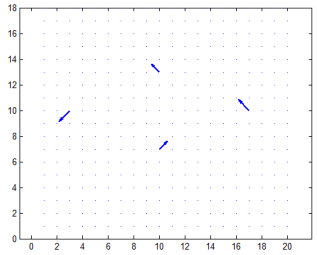

Figure 12: Velocityfield with four agents velocities written as impulses

This method is ok for 2D, but in 3D the domain would have to be rather rough to not use an excessive ammount of memory and to not take too long to convolute.

Density The density ρ of a crowd is the sum of the smoothed density impulses of all agents as seen in figure 14. Density is mass divided by area (in 2D) $\rho = \frac{kg}{m^2}$.

Separation and cohesion Pressure P relates linearly to density as $P = \rho c$ where c in some texts is the speed of sound in the fluid material. For crowds the value of c is just a scalar describing the unwillingness to be compressed.

The separation forces is then the negative gradient of the pressure field $-\nabla P$ and the cohesion force is the positive gradient ∇P.

Steering Simple BIOD-like agents will not only push each other (forward or backwards) when moving into one another, they will also influence the neighbors that they move along side, to speed up or slow down to match each other's speed. This neighbor influence is controlled by viscous forces in a fluid. Viscous force is the vector laplacian of the velocity field $\nabla^2 v$ scaled with the viscosity coefficient, or in crowd terms: the agents tendency to steer together with the neighbors. An overview of pressure and viscous forces in a fluid can be found in [Tritton(1988)]

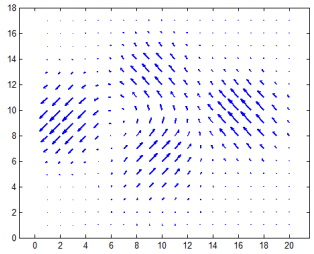

Figure 13: Velocityfield with four agent velocities smoothed with a Gaussian

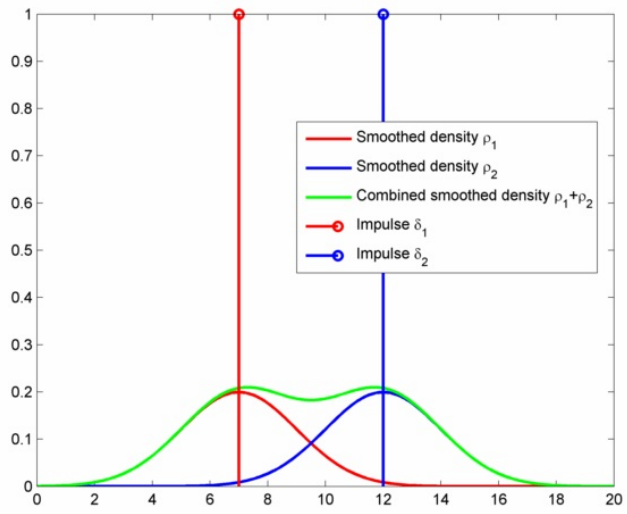

Figure 14: Two impulse densities, located at the agents positions, are smoothed with a Gaussian kernel resulting in the sum of two Gaussians. The individual Gaussians are shown as illustration but are not explicitly calculated

Crowd related fluid properties The properties of a fluid that are relevant for the crowd model are.

- Density ρ which is important for finding a path which avoids congestion and for visualizing the crowd
- Velocity field v which is required for finding paths that move in the same direction as the other agents and avoid agents moving in oposite direction
- Vector Laplacien of velocity field $\nabla^2 v$ which is used in viscosity force calculations that make the agents steer together
- Pressure gradient ∇P which is needed to find the direction and magnitude of the separation - as well as for cohesion, which is not a true fluid property but one that can also be derived from the pressure gradient

5.1.1 Smoothing agents

In figure 14 impulses and their smoothed counterpart are shown. The smoothing is done with a smoothing kernel that scales the contribution of the agents to the field, at a specific position, by considering the gents distance and direction from that position. A Gaussian smoothing kernel is seen in (3) where r is the distance from the position in the field for which we seek a contribution, σ^2 is the variance and a is a scaling factor ensuring that the function has unit integral. There are other requirements for smoothing kernels which are detailed in [Liu & Liu(2003)].

$$a\, e^{-\frac{r^2}{2\sigma^2}} \tag{3}$$

Unit integral All kernels should have unit integral. This ensures that whatever function is smoothed by the kernel, the smoothed function will have the same integral. In other words; the ”material” is distributed differently but there is no

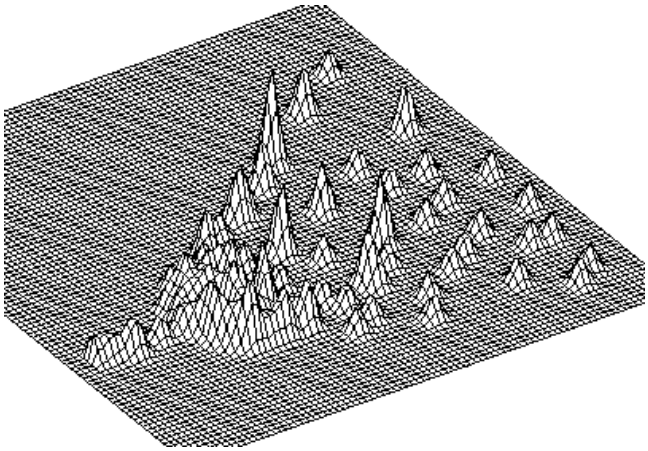

Figure 15: Impulse functions on top of agents were smoothed by a Gaussian

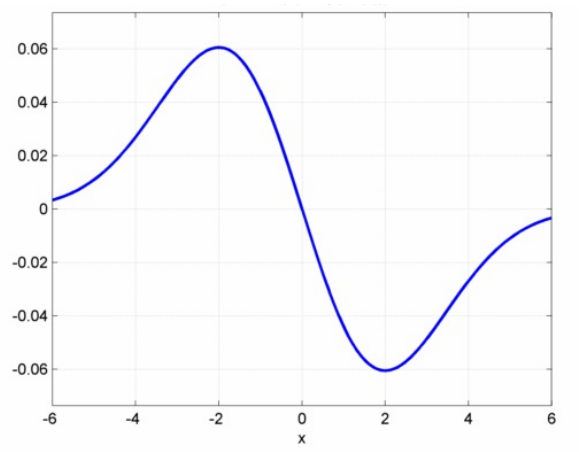

Figure 16: The first derivative of a Gaussian function in 1D is zero when the distance from the mean is zero

more and no less of it after smoothing[4]. The result of smoothing several impulse functions located at the agents positions in 2D is seen in figure 15.

One can argue that an agent does not have a smoothed but a single well defined position, which is true. It is smoothed, however, to calculate the influence of an agent, which reaches beyond the actual space occupied by that agent. We also need to consider things like a crowd density and have it be defined by a differentiable function usable for calculating gradient based forces. An impulse function on top of every agent would give none of this.

Kernels While a Gaussian smoothing kernel is reasonable for density calculations, there is a problem with using it when calculating pressure forces. The derivative of a Gaussian goes towards zero when the distance from the mean goes towards zero as seen in figure 16. This means that the repulsive force is strongest for an agent distance of 2 (with the specific choice of variance for the Gaussian in the figure) and that agents closer than 2 will be less influenced to move away from each other, and will eventually not be pushed away at all when the agents overlap. This is not realistic for human agents in a crowd.

Spiky kernel An alternative kernel is the ”spiky” kernel function (4) described in [Müller *et al.*(2003)Müller, Charypar, & Gross]. In that text the kernel is defined for 3D space but we will convert it into a 2D kernel.

In (4) r is the distance between agents and h is the support radius. The support is the distance at which the kernel value drops to zero and agents farther than r apart have no influence on each other. The spiky kernel has a first derivative that does not approach zero as the distance between agents decreases. Instead it has a discontinuity which ensures that the calculated force will always increase when two agents move closer together and the force will change to the opposite sign if they pass each other. The kernel and its derivative is shown in figure 17.

Since kernels must have unit integral and we will be working in 2D, the basic function $(h-r)^3$ have been scaled by the 2D integral. The function is circular so this is best integrated in polar coordinates and is $\frac{\pi h^4}{2}$ and the scaling factor is therefore $\frac{2}{\pi h^4}$ as seen in (4). We also need the partial differential of the function which is seen in (5).

[4]An agent which is smoothed out over the edge of the domain will actually ”loose” mass. There exist schemes to deal with this [Liu & Liu(2003)], but in this implementation that problem is ignored

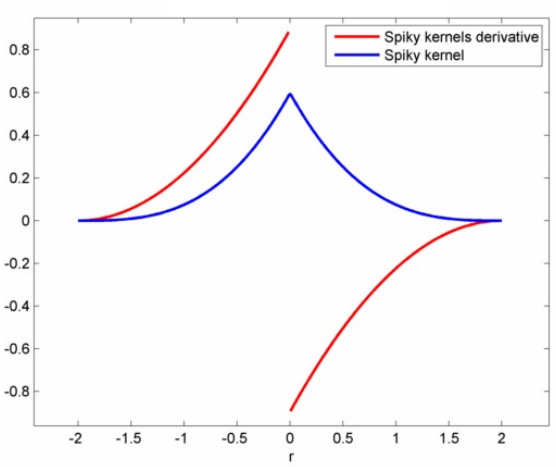

Figure 17: The spiky kernel along with its first derivative

$$spiky(r,h) = \frac{2}{\pi h^4}(h-r)^3 \quad for\ r \leq h,\ else\ zero \tag{4}$$

$$\nabla spiky(r,h) = -\frac{6}{\pi h^4}(h-r)^2 \frac{r}{\|r\|} \tag{5}$$

Seperabel convolution If a convolution kernel is separabel, then a 2D convolution can be done as the sum of two 1D convolutions. This is much less work but it does require the kernel to be separabel which means that it should be the outer product of only two vectors and therefore have only rank 1. While this is true for the Gaussian kernel and its derivatives, it is not true for the spiky kernel which has rank 20. Doing a SVD decomposition of the spiky kernel shows that the diagonal values are quite close to 0 except for the four largest. This means that even though the kernel is not separabel, it is primarily the sum of only four outer vector products. We could therefore still gain some of the performance benefits from separabel convolution using the spiky kernel, though it would be more complicated and not as accurate as a non separabel convolution using the actual kernel. In this paper we will, due to the dificulties with the spiky kernel, not use separabel convolution, but leave that as an option for further optimization later on.

Frequency domain convolution 2D convolution can be very time consuming for large kernels. The time complexity of non separabel 2D convolution on an impulse map size n^2 using a kernel size N^2 the is $O(n^2N^2)$. For this reason one can onvert both impulse map and kernel into the frequency domain, perform the convolution there and then convert back from frequency domain. With kernel and impulse map converted to frequencies, the convolution is a point wise multiplication of the two. This is not a large job even for large kernels. The conversion to and from frequency domain is done with a Fast Fourier Transform FFT and an inverse thereof IFFT. Both have previously been implemented efficiently on CUDA [Podlozhnyuk(2007)].

Convolution with CUDA In order to obtain the density field and the smoothed pressure gradient vector field, we will view the discretezised domain as a 2D matrix and perform convolution with this matrix and an appropriate smoothing kernel. The result is written into 2D textures for quick cached lookup later on. The domain matrix is initialized to zero and then each agent will add a single impulse to the matrix cell it occupies and the resulting matrix is then convolved using the method described in [Podlozhnyuk(2007)] which runs at 252 MPixels/s for a square separable kernel with a radius of 9 on a GF9800GT. With textures of 1024x1024 texels this is expected to take 3.96ms for each convolution.

Textures When the fields are stored inside 2D textures, the texture unit makes it easy to lookup field values at real valued coordinates, even though the texture is discrete. A texture unit can be set to return bilinearly interpolated values. It should be noted however that linear interpolation between discrete samples of a function, which is non-linear,

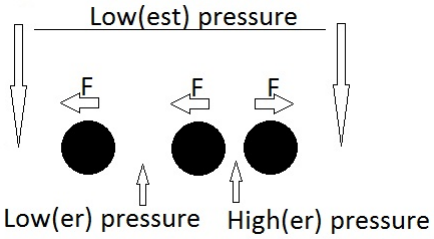

Figure 18: Pressure forces push away from higher pressure towards lower pressure. The magnitude depends on the difference in pressure

is a rough approximation. If for example the function $f(x) = x^2$ is sampled for $f(10) = 100$ and $f(20) = 400$ then the linearly interpolated value for $f(15)$ is 250 while the actual value for $f(15)$ is $15^2 = 225$. This is inaccurate, but it does not show itself to be a problem in our implementation.

5.2 Separation force

The pressure force provides separation. Agents are repulsed by high pressure, but are *not* attracted to low pressure. Low pressure exist only when compared to some other higher pressure. That high*er* pressure in another direction is what pushes agents towards low*er* pressure. This is basic but quite important. Presure forces, as seen in figure 18, *only* push. They never pull.

$$P = \rho c_{pressure} \tag{6}$$

$$F_{pressure} = -\nabla P \tag{7}$$

5.3 Cohesion force

As stated previously, pressure pushes away from high pressure towards low pressure. It does not attract. There is however nothing preventing us from making a fictional attractive pressure by using the positive pressure gradient rather than the negative. Fluids do not behave like this - but crowds do.

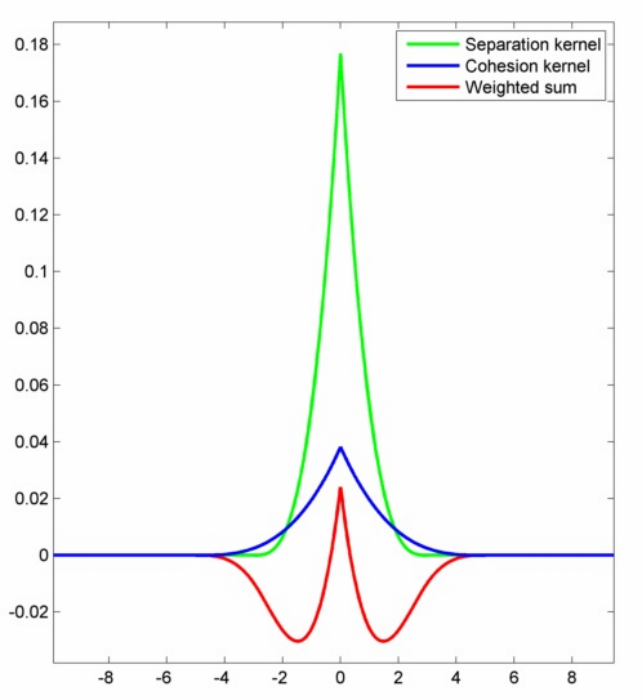

Figure 19: Kernels for separation and cohesion and their weighted sum $separation - cohesion$. An agent doing gradient descent on this sum will end up in the minima surrounding the common mean. In 2D there is one minimum forming a circle around the mean at a distance where the forces are equal in magnitude but opposite in direction

There is a distance at which cohesion and separation are equal in magnitude and opposite in direction. At this stationary point, agents will be content with keeping the distance constant. This means that the two forces will need to have a different falloff-rate since there will be no stationary points when just adding two differently scaled spiky kernels with the same support radius and the same center. One kernel will always be greater than the other. It is also obvious that the cohesion should be stronger than the separation at a distance and that close up it should be the other way around. Therefore the cohesion pressure gradient has to be calculated with a spiky kernel that has a larger support. Essentially we will make sure that the sum of the two kernels form a spiky mexican hat - a spiky version of a Difference of Gaussians.

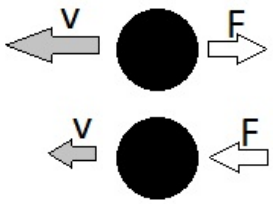

Figure 20: Viscous forces act to make the differences in velocity go to zero. A fast moving agent close to a slow moving agent will create viscous forces that will slow down the fast and speed up the slow

5.4 Steering force

The viscous force is used to implement the mimicking behavior of agents. Viscous forces will accelerate nearby agents towards equal velocity and direction. It will speed up the slow and slow down the fast until they have identical motion vectors, or until they have moved away from each other. This is seen in figure 20.

$$F_{viscous} = \nabla v c_{viscous} \tag{8}$$

5.5 Static pathfinding and global forces

While the fluid simulation view on a crowd can deal with local inter-agent forces, there is a need for pathfinding to support the larger goal of getting from here to there. A goal can be a door leading out of a map of a building floor plan or the free seats in a theater or something similar that the agents want to reach.

The discretized domain consists of square cells with each - apart from the ones at the boundary - 8 neighbors. This can easily be seen as a connected graph with vertices at the cells and edges between neighboring cells. We have chosen to the A* algorithm with no heuristic function.

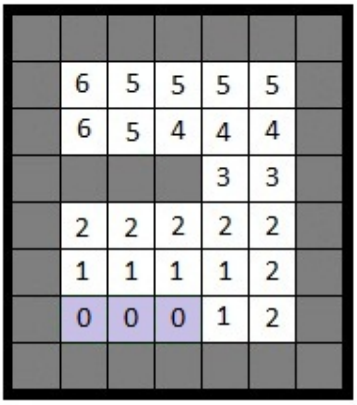

Figure 21: Pathfinding starts in the green goal nodes and spread out to the entire domain, filling it with the cost of going between a given cell and the goal

The A* algorithm is very well described so we will not do that in this text. We only point out two details that we are doing differently. Firstly we do not have an ordinary goal vertex to pathfind *to* in the graph. Instead we pathfind until every vertex has been reached and have had an actual travel cost from the source calculated. Secondly, we do not use a heuristic function which is not needed when we want to examine the entire graph. The heuristic function is normally used to avoid just that, and to only look at the direct path from start to goal.

Goals A goal position in the simulation is defined as a set of cells in the map as on figure 21 which are all considered goal positions. We pathfind *from* all such goal cells and fill the entire graph with calculated cost values from the goal cells. The cost is the linear distance from center of one cell to the center of the next. This means that diagonal moves cost $\sqrt{2}$ while non diagonal moves cost 1.

After such a pathfinding, every cell in the discretized domain will hold a cost of getting from it to the goal. If the simulation has multiple different goals then a pathfinding is calculated for each goal and the cels will now hold multiple cost values - the cost of getting to each goal.

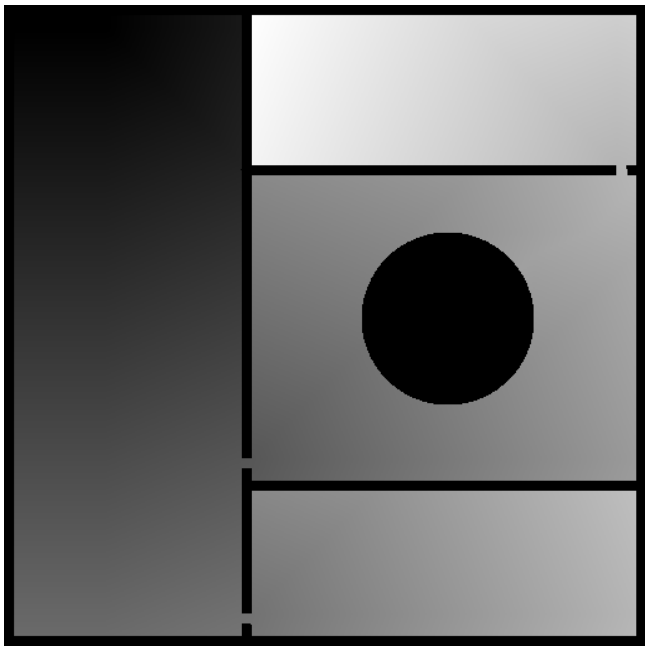

Figure 22: Static path cost for one exit located at top left corner

There has been some research into using the GPU for pathfinding but the focus seems to always be on how to calculate a very large number of paths simultaniously rather than calculating one path whith effective use of a large number of threads. Another form of pathfinding, allpairs-shortest-path, has been investigated using CUDA [Katz & Kider(2008)], but even though an all-pairs-shortest-path calculation could be usefull for us, it is an extreme overkill. It is fast compared to doing all-pairs on the CPU, but it is a large problem and it is taking too long to solve. In addition to that, the implementation is not trivial. For theese reasons we have currently implemented the pathfinding on the CPU.

Cost gradient After pathfinding, or rather just plain cost calculations, we can easily go from any cell to a goal by folowing the negative cost gradient which can be calculated using finite difference in the regular grid. Incidentally a finite difference can also be implemented using separable convolution. Since the agents have real values positions, the agent will need to lookup the cost gradient in-between cells using bilinear interpolation. For this reason, the costs gradient is stored inside a texture so that the texture unit can perform the interpolation for us.

Agents are controlled by forces and the pathfinding can be considered a complex force that changes in direction as the agent moves around. The force does not point linearly towards the goal but instead it points along the shortest path to the goal from the current position.

5.6 Dynamic pathfinding

Dynamic pathfinding takes the crowd itself into consideration. It tries to find the path of least effort by avoiding dense parts of the crowd and in particularly parts of the crowd that does not move in the direction the agent does.

The actual pathfinding works the same way as the static counter part, but where the static version was only concerned with the distance traveled, the dynamic one penalizes moves through areas with high density and areas with an oposing velocity field.

The cost of high density areas is just the density multiplied with a scaler which lets us change the preferences of the agent, by making that agent more or less concerned with congestion. If it is a dark night filled with dangers, then an agent could even be designed to prefer paths which were not too isolated from the rest of the crowd.

The cost of going against the flow is based on the observation that the (no special effort) motion vector at a certain position in the crowd will be equal to the average motion vector at that location. It will take extra effort to move in another direction and the more the desired motion vector points away from the smoothed field velocity vector, the more effort it takes .

It will however cost nothing extra to move orthogonal to the average velocity or even to move faster than the average velocity. Those will be handled

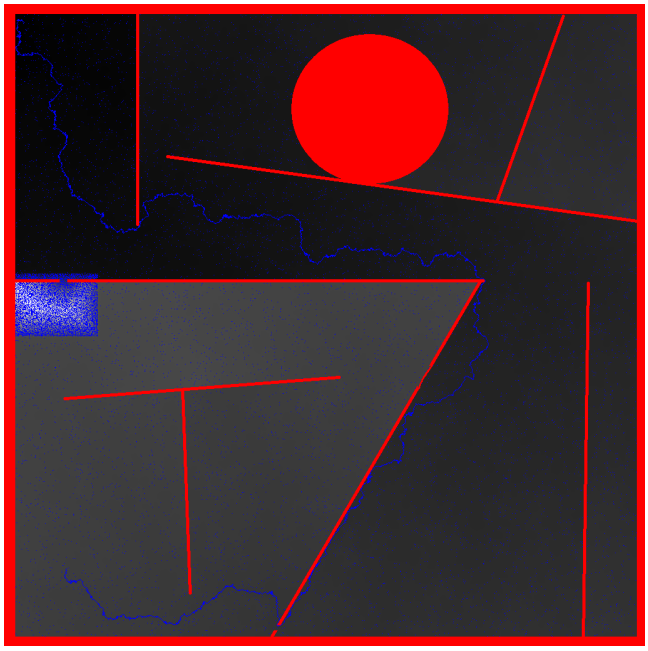

Figure 23: A square crowd is placed so it blocks the static shortest path. An agent moves around a number of static agents spread throughout the domain to get to the goal

in the density penalty. Only crowd velocity going against the agents own movement vector is penalized by a clamped dot product as seen in (9).

$$cost = -min(0, v_{field} \bullet v_{desired}) \tag{9}$$

Note that the velocity field is implicitly scaled with the density of the crowd through the smoothing. Away from the dense crowd the velocity field will approach zero and have little influence on movement cost[5].

Since the smoothing kernels have unit integral the sum of all the smoothed movement vectors will be equal to the sum of all the individual movement vectors.

Update frequency While the forces arising from physical collisions among agents is immediate and needs to be calculated every time step, the "forces" from dynamic pathfinding does not need to run quite as frequent. A real human agent *do* take some processing time to observe the crowd and find the areas towards the goal where the crowd moves the right away and the areas where the crowd is not overly dense. We can therefore relax the timing requirements quite a bit. This is fortunate, considering that the current implementation of pathfinding is on the CPU and is not real-time.

[5]This is why movement faster than the field velocity should not be penalized

5.7 Forces in a crowd perspective again

Local, fluid inspired, forces will provide separation, cohesion and steering. Global pathfinding force will pull an agent towards its goal position.

An agent has to weigh the influence vectors which will rarely point in the same direction. This is done by multiplying with four different scalars. The sum of weighed forces will then accelerate the agent by $a = \frac{F}{inertia}$.

- Separation, this is identical to the *negative* pressure gradient of the pressure from the narrow kernel $-\nabla P_{narrow}\ c_{separation}$
- Steering, this is the viscosity $\nabla v\ c_{steering}$
- Cohesion, this is the positive pressure gradient of the wide kernel pressure $\nabla P_{wide}\ c_{cohesion}$
- Pathfinding, this is the negative path cost gradient $-\nabla cost\ c_{path}$

6 Results

The described methods were implemented on CUDA and a limited[6] number of tests were performed. Timing for various number of agents and various sizes of domains were performed. Crowd simulation was executed and observed for the described behaviors. Some behaviors were clearly seen while others were not.

6.1 Timing

As expected, the time spent calculating each step of the simulation was dependant only on the size of the domain. While it does take slightly longer to write impulses for 10.000 agents than for 5.000 the influence of that part was not much of a factor in the measurements. Time spent calculating the forces and performing the updates each step is shown in table 1 and in table 2.

The time taken to perform dynamic pathfinding was measured, and is shown in table 3. While the pathfinding can run parallel with the simulation, working on an older state and returning a delayed optimal path, it will have to run on a coarser discretized domain or on a small domain in general. The time spent doing static pathfinding is irrelevant since it is done as preprocessing.

6.2 Behavior

The behavior of the agents were realistic in that they could be made to not intersect. This was tested by counting the number of cells with an impulse written into it and ensuring that each agent had been assigned to a single cell through a simulation run. For lower repulsive forces and larger cell sizes one should expect multiple agents in the same cells though.

[6] See section 6.3

Lanes We did see lane formation, but not consistantly. In figure 24 lanes are seen, but mostly for one group. The red group are mosly showing the lane behavior while the green groop is more loose. It is unclear why one group forms clear lanes and the other seems to move more loosely in-between but it was a quite consistent observation. One group forms lanes and the other fills in the space between lanes.

Vortices Both by visual inspection and by calculating the curl of the smoothed velocity field, we looked for the formation of vortices. We did not find any. It is unclear if the crowd need to have a specific relationship between speed, inertia and separation and cohesion forces for this to occur. The literature does not provide any clues to solve the puzzle.

Braess's paradox When testing how the exit rate of a room with two doors compare to a room with three dors, where one door is directly between the other two, we would see the expected drop in exit rate, but it required the dynamic pathfinding to be off.

When the agents performed dynamic pathfinding, they would avoid the expected congesting at the center door, and simply move out through the other two doors. Approximately one third of the agents exited through each door and three doors were faster than two doors.

If dynamic pathfinding was off, then the agents *all* moved to the center door closest to the goal and everyone exited through that door, but that was the case both with one, two and three doors. Therefore there was not a worsening of the exit rate with a third door opening.

The problem is that we could only toggle between entirely stuid or entirely clever agents. There was nothing in between. This could possibly have been implemented by letting some agents use the dynamic pathfinding while others only knew the static path.

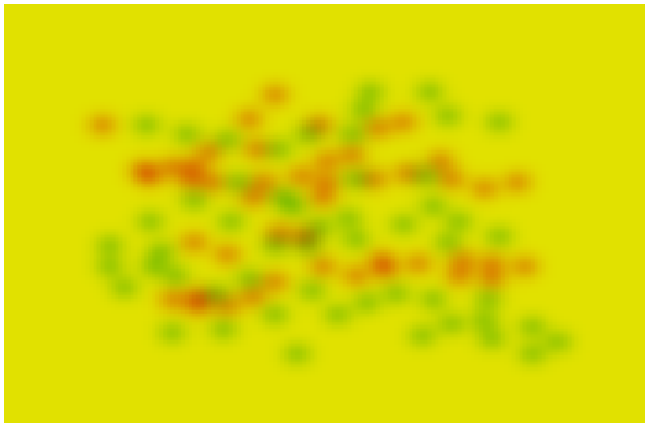

Figure 24: Lane formation. Red group moves right to left and green group moves left to right. Both groups are pushed from top and bothom to keep them from spreading out too much. Note that the density is visualized but extends farter away from the agents than what can be seen in the coloring of the smoothed densities

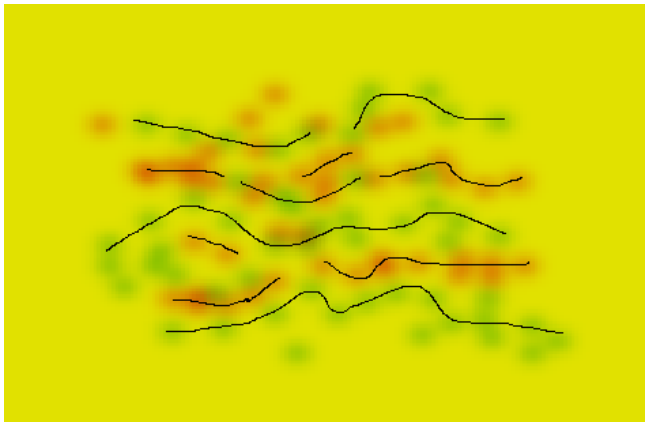

Figure 25: Same lane formation as in figure 24 with hand drawn lines marking the lanes

6.3 Stability of code

The implemented code was unstable. There were random crashes and simulations which sometimes, for no apparent reason, exploded. It is quite likely that is has to with hardware problems on the system used for implementation and test. The graphics card has been seen to run quite hot which may be the root of the problems.

It did however hinder thorough testing to some degree and it does leave a slight uncertainty about the correctness of the implementation.

7 Conclusion

We have had to conclude that while the methods described here do have merit, they also face some problems.

The convolution on GPU is quite fast indeed, but having a large domain and having to perform multiple convolutions to find the different fields does take time. It is in fact slower per simulation step than what expected from the speed seen in the fluid and n-body simulations released by Nvidia, which do not use convolution. It simply is faster to handle ordinary particle interaction, as is the norm for SPH, than having to smooth the entire domain. The smoothed densities and velocity fields does support the dynamic pathfinding, but again we see that the dynamic pathfinding is too slow on large domains the way we implemented it.

Unless the entire field is required for visualization or pathfinding or the domain is small compared to the number of agents simulated, the convolution method seems to be too out of the ordinary and too slow.

Num agents	512	1024	2048	4096	8192	10.000	20.000	40.000	60.000
Time pr. step	14.1ms	14.7ms	14.3ms	14.4ms	15.2ms	15.1ms	15.7ms	15.7ms	15.9ms

Table 1: Time pr. simulation step. Based on number of agents with constant domain size of 1024x1024. Time does not include dynamic pathfinding or visualization. This is implemented on GeForce 9800 GT

Domain size	256	512	1024	2048	4096	8192
Time pr. step	3.2ms	4.1ms	14.7	68.1ms	230.2ms	920.6

Table 2: Time pr. simulation step based on domain size with constant agent number of 1024. Time does not include dynamic pathfinding or visualization. This is implemented on GeForce 9800 GT

Domain size	256	512	1024	2048	4096
Time ms	37ms	252ms	630ms	2709ms	12754ms

Table 3: Time doing static or dynamic pathfinding on different domain sizes. This is currently implemented on Intel Core2 Quad Q9555 2.83GHz CPU

References

Chattaraj, U. (2009). Understanding pedestrian motion across cultures. URL `http://cms.shu.edu.cn/Portals/286/005-tgf09-Ujjal-Chattaraj.pdf`.

Clements, R. R. & Hughes, R. L. (2004). Mathematical modelling of a mediaeval battle: the battle of agincourt, 1415. *Mathematics and Computers in Simulation*, *64*(2), 259–269. URL `http://dx.doi.org/10.1016/j.matcom.2003.09.019`.

Dijkstra, J., Timmermans, H. J. P., & Jessurun, J. (2000). A multi-agent cellular automata system for visualising simulated pedestrian activity. In S. Bandini & T. Worsch (Eds.), *Theoretical and Practical Issues on Cellular Automata, Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry, Karlsruhe, 4-6 October 2000*. Springer, 29–36.

Gingold, R. A. M. & J., J. (1977). Smoothed particle hydrodynamic: theory and application to non-spherical stars.

Katz, G. J. & Kider, J. T., Jr (2008). All-pairs shortest-paths for large graphs on the gpu. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 47–55.

Kirchner, A. & Schadschneider, A. (2002). Cellular automaton simulations of pedestrian dynamics and evacuation processes.

Krause & Dyer, J. (2008). Sheep in human clothing. URL `http://www.leeds.ac.uk/media/press_releases/current/flock.htm`.

Liu & Liu (2003). *Smoothed particle hydrodynamics*. World scientific.

Müller, M., Charypar, D., & Gross, M. (2003). Particle-based fluid simulation for interactive applications. In D. Breen & M. Lin (Eds.), *Eurographics/SIGGRAPH Symposium on Computer Animation*. San Diego, California: Eurographics Association, 154–159. URL `http://www.eg.org/EG/DL/WS/SCA03/154-159.pdf`.

Nvidia (2007). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.

Nvidia (2009). *NVIDIA CUDA C programming - Best Practices Guide*.

Podlozhnyuk, V. (2007). Fft-based 2d convolution. URL `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf`.

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 25–34.

Reynolds, C. W. (1999). Steering behaviors for autonomous characters. URL `http://citeseer.ist.psu.edu/509294.html;http://www.red3d.com/cwr/papers/1999/gdc99steer.pdf`.

Still, K. (2000). *Crowd Dynamics*. Ph.D. thesis, University of Warvick.

Top500Org (1999). URL `http://www.top500.org/list/1999/06/`.

Treuille, A., Cooper, S., & Popović, Z. (2006). Continuum crowds. *ACM Transactions on Graphics*, *25*(3), 1160–1168.

Tritton, D. (1988). *Physical Fluid Dynamics*. Oxford university press.

Wainwright, M. (2007). Braess paradox. URL `http://www.crowddynamics.co.uk/Main/Braess%20Paradox.htm`.