Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2011)
J. Bender, K. Erleben, and E. Galin (Editors)

Introducing congestion avoidance into CUDA based crowd simulation

F. Wockenfuss and C. Lürig

University of Applied Science Trier

Abstract

The simulation of larger crowds of peoples at interactive frames rates get more and more important in games and interactive simulations. Crowds contribute significantly to an immersive urban environment. The nature of the problem where one has to simulate a lot of individuals makes it very accessible for parallelization strategies. As graphics hardware today is a very accessible and powerful parallel computation hardware, the problem lends itself to be adapted to graphics hardware. It is implemented in CUDA in our case.
CUDA and other parallel implementations of social force models usually have several numerical and parallelization issues that are finally based on too inhomogeneous population distributions. These are often also undesirable from a game developers point of view. In this paper we introduce the concept of congestion avoidance into the crowd simulator that solves those technical issues and makes the whole simulation also look more natural for gaming applications.

Categories and Subject Descriptors (according to ACM CCS): I.6.8 [Simulation and Modeling]: Parallel—Gaming

1. Introduction

Using crowd simulation has become increasingly popular in video games over the recent years as one can see in games like [Ubi07,EA10,SCE10]. One can expect this trend to continue in the future. Implementing crowd simulation in games requires a high degree of efficiency and the usage of consumer level hardware. Therefore standard GPUs (graphics processing units) lend themselves well to that problem. If implementing a crowd simulator one has to observe that numerical stability in gaming applications is far more important than accuracy. Additionally for aesthetics reasons continuous crowd movements and a mostly homogeneous density is highly desirable.

The approach to crowd simulation we take in this paper is based on a social force model. The idea behind that approach is that the objective of the individual and the situation around it exert several forces on the simulated individual that result in acceleration and therefore movement. The most prominent early publication is the paper of Reynolds [Rey87] on flocking behavior. In this paper repulsive and attractive forces between individuals are used to model crowd behavior. The approach we follow here is more oriented towards the social force model of Helbing [HM95]. We only use repulsive forces between individuals. Those however also take orientation and velocities of people also into account, which results in a more naturally looking pedestrian simulation in crossing traffic simulations. Alternative approaches to crowd simulation are simulations based on queuing models like the one followed by Strippgen and Nagel [SN09] or simulations based on cellular automates like used by Klüpfel et al. [KMKWS00]. The queuing based approach would be more suited for traffic simulation and the cell based one for evacuation scenarios. For general pedestrian simulation in games variations of the social force model are generally the most suited.

The implementation of the simulation is done in CUDA [NVI08]. CUDA is a library from NVIDIA for NVIDIA graphics cards that provides possibilities to use graphics processing units for general purpose computing. Therefore it is called GPGPU computing. Graphics cards processors are massive parallel streaming processors that put a high amount of transistors in actual data processing and a very low amount if it in caching logic. Therefore those processors are optimized for high bandwidth at the cost of latency. CUDA and GPGPU computing in general provide the possibility for very efficient implementations of highly

DOI: 10.2312/PE/vriphys/vriphys11/101-110

parallelizable and compute intensive problems on consumer level hardware. As crowd simulation itself is highly parallelizable it lends itself very well to CUDA implementation. CUDA provides the opportunity to spawn many threads that are grouped into thread blocks. Threads of one block share a certain amount of memory and can be synchronized between each other. As with all parallel architectures an efficient implementation within this system depends on the efficient use of the local memory and keeping the amount of synchronization points minimal.

CUDA has already been successfully used for crowd and pedestrian simulation. Strippgen and Nagel [SN09] have implemented a queue bases simulation approach for traffic simulation. There are also already a couple of publications available that are based on social force models. Richmond et al. [RCR09] describe a simulation framework for pedestrians based on CUDA. As they use a social force based system they analyze all the relations between the different agents and therefore obtain a problem of quadratic complexity. Their argument is that even though they have quadratic complexity the algorithm suits itself for efficient parallelization on CUDA. In their predecessor paper [RR08] they have used traditional pixel and fragment shaders to address the issue. Traditional fragment shaders have also been used in the work of Rudomin et al. [RMH05] and Souza et al. [DLR07]. Rudomin stores the individual states of the agents in different textures. Souza uses a cell based approach in simulation.

Erra et al. [EFSC09] make use of the fact, that the influence between crowd agents usually diminishes with distance and therefore can be culled outside a certain radius. This reduces the problem complexity significantly. A uniform cell structured is applied that covers the whole area of pedestrians that should be simulated. Only a certain amount of neighboring cells must be regarded when simulating one agent. The cell structure provides the required data locality for the parallelization solution in CUDA. As for memory reasons only a certain amount of people can be simulated per thread block and therefore per cell. To achieve this a global sorting process is included before every simulation update to provide the necessary load balancing mechanism for every thread block.

As one can also see from those experiences implementing a crowd simulator in CUDA based on a social force model usually comes with three main difficulties.

- The first difficulty is the reduced data locality of the problem that originates from too varying crowd densities. In crowded areas there should me more threads used per square meter space. The compute source requirements depend on the numbers of persons to simulate. If the crowd density can vary significantly a dynamic load balancing scheme has to be introduced to readjust the cell sizes in a global processing step.
- The second problem is the numerical instability that stems from the explicit integration of social force models. This usually appears also in areas of high crowd density. High densities result in high repulsive forces and therefore result in stiff equations.
- And last but not least a too high crowd density is often not suitable for games as the visibility of a single crowd member to the player get too much reduced. Local behavior may look twitchy even if the simulation is still numerically stable.

In this paper we solve this problem by controlling the crowd densitys in the first place. As a too high crowd density is the root of all three problems we enforce that only a certain maximum of people can be allocated in a cell of a uniform grid we put over the simulated surface area. Additionally we add an extra force to our simulation mode that make crowd members steer away from almost completely filled cells. We call this behavior congestion avoidance. This is analogue to the force component that make crowd members steer away from individuals and is usually called collision avoidance. Congestion avoidance works on a longer range and more coarse granular scale than collision avoidance. It prevents people from getting into too densely populated areas. This effect can further be enhanced by introducing crowd dispersion. Crowd dispersion adds an attractive force towards areas of low crowd density.

This approach eliminates the global sorting and preprocessing step. The cell size is adjusted to the influence range used in the simulation. This way all cells only need to know the situation on the neighboring cells. Therefore before each simulation step every cell is informed by its neighboring cells what happened within them during the last simulation step. Afterwards every cell updates its agents. If an agent is about to leave a cell this information gets transfered to the corresponding neighbor cell afterwards. The simulation responsibility of this gets transfered to the neighboring cell also. The communication between the different cells and therefore thread blocks is done over a blackboard architecture as described in [BMR*96]. The data exchange and overlapping memory access is therefore always limited. The blackboard is allocated in the device memory.

As a side effect we avoid the A-stability problems of explicit Euler integrators that are usually applied in social force models. When used on too stiff differential equations explicit integrators cause stability problems where the whole system starts oscillating increasingly. In the case of crowd simulation equations get stiff in conditions of low distances between actors. In this case the repulsive forces get very high. As we limit the crowd density to a controlled and known value this can not happen in our case. Also limiting the crowd density avoids twitchy and unnatural looking reactions which often break the suspension of disbelieve in a gaming context.

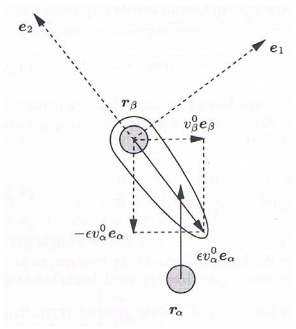
The outline of the remains of this paper is as follows: After discussing the related work in the following section the implementation of the social force model in CUDA will be described. The exact force calculation will be discussed as the distribution strategy of the crowd members among the different thread blocks. This discussion include the black board architecture that is used to provide communication between the different blocks. Afterwards follows the description of the real innovation of this paper that makes this blackboard approach feasible which is the introduction of the congestion avoidance component. Then we will discuss some results with performance values for different avoidance situations. Here we will also show that congestion avoidance and crowd dispersion in pedestrian simulations provides visually pleasing results. Finally we conclude with some possibilities for future work to extend the pedestrian simulation.

2. Related Work

Pelechano et al. [PAB07] explain that crowd simulation approaches are either rule based, cell based or based on social forces. The social force model is the most suited one for simulating large amount of pedestrians. This is the objective of this paper. Social force models contain elements of repulsion and attraction like used in the work of Erra et al. [EFSC09]. The component of attraction is used for flocking behavior. In this work we focus on the most important force component in biological simulation which is the repulsion component [KR02].

The social force model used by Helbing [HM95] also concentrates on repulsion to simulate the movement of pedestrians. This model calculates the reactions of simulated pedestrians as reactions to a social forcefield. This forcefield is created by social forces, which act much like physical forces. While agent based simulations try to calculate the expected reaction as the result of different conditions and prerequisites, the social force model adds up all forces created by the environment and simulates the person being moved by these forces much like a physical particle in a fluid or gas would be. One primary force for the simulated pedestrian is the motivation to move to a certain goal, which creates an attractive force pulling the pedestrian straight to his goal. Obstacles like walls do not only block the movement of a pedestrian, but also create a repelling force. This simulates believable behavior, since real pedestrians leave a certain amount of free space to obstacles when moving around freely.

In Helbings work pedestrians create a forcefield, which depends on the velocity and relative position to the target and the orientation of both the target and the pedestrian itself. These factors account for the fact that pedestrians need more space when moving faster, and pedestrians react stronger to others in front of them than behind them. Another factor is the early avoidance of pedestrians walking straight at each other, which results in a spike in the forcefield right in front of the pedestrian, which can be observed in figure *(fig. 1)*. This forcefield is scaled by the current speed, the crowd density and orientation of the target. For each pairing of two pedestrians a unique force field is created.



e_2
e_1
r_β
$v_\beta^0 e_\beta$
$-\epsilon v_\alpha^0 e_\alpha$
$\epsilon v_\alpha^0 e_\alpha$
r_α

Figure 1: *The forcefield created by a pedestrian for a certain target as extracted from Helbing [HM95]*

The force representing the influence on the target by this pedestrian can be calculated by sampling the forcefield at the position of the target. This leads to the following formula:

$$f_{\alpha\beta} = -\nabla_{r_\alpha} S e^{-\sqrt{[(r_\alpha - r_\beta)\cdot e_1]^2 + [(r_\alpha - r_\beta)\cdot e_2]^2/(\gamma_\alpha)^2}/R}$$

This formula encloses the term γ_α which is calculated in accordance to the pedestriance orientation and velocity:

$$\gamma_\alpha(t) = \begin{cases} \vartheta & if \quad [r_\alpha(t) - r_\beta(t)] \cdot e_2(t) \geq 0 \\ 1 + \Delta t \vartheta_\alpha(t) & if \quad [r_\alpha(t) - r_\beta(t)] \cdot e_2(t) < 0 \end{cases}$$

The factor S is a general strength factor (in our case it is 1.0). R is the influence radius of a crowd member, which is usually $4m$ in our case. The meaning of the remaining symbols can be deduced from figure *(fig. 1)*.

As stated before problems may occur with this approach in areas of high population densities as the resulting equations get stiff to avoid overlapping pedestrians. This makes the approach not suitable for an explicit differential equation integrator. Lakoba et al. [LKF05] solve this problem by implementing a special numerical algorithm that treats the overlapping of persons separately. As we are less interested in panic situations as Lakoba et al. but more in general moving crowds in game situations we choose a different approach, where we try to avoid the situation by introducing congestion avoidance.

In order to implement this model efficiently, strategies for creating thread blocks as described by D. Kirk and W. W. Hwu in [KmWH10] are used in this work. As depicted in various case studies, the efficiency of a CUDA kernel depends heavily on the right usage of the various memory locations and hardware limitations of the GPU (see for intances [NVI11]). Access to global memory is one of the bottlenecks in parallel GPU programming, since these accesses impose a delay of several cycles and suffer from limited bandwidth. The goal is to minimize accesses to global memory and align the reading of global memory to gain the benefits of coalesced memory access. The fastest memory for communication between threads is the shared memory, accessible by all threads in a single threadblock. But the maximum size of shared memory available on a single Multiprocessor on the GPU require a detailed balancing of several limits. To gain optimal occupancy each multiprocessor should work on a maximum number of threads at one time. Occupancy is the percentage of computational resources actually used in calculation of the kernel. Without several tests and balancing a significant amount of calculation power is wasted.

As an alternative to CUDA and PC graphics cards Reynolds [Rey06] has implemented a Crowd simulator on the PS3 for gaming purposes. This approach is of course restricted to a specific console.

3. Development of Crowd Simulation

In order to reduce the high complexity of Crowd Simulation we use a cell based approach for storing all individuals. A uniform grid structure harbors a lot of advantages compared to dynamic grids or nearest-neighbor searches. A uniform grid offers good data-locality for entities close to each other. No preprocessing or sorting is required. New local forces like attractive elements can easily be implemented on a per-cell basis using all benefits of data-locality. The trade-offs of a uniform grid structure are the massive memory-requirement, since each cell in the grid needs to reserve enough memory for the maximum number of entities it can sustain at any one time, and the load balancing for almost empty cells.

The uniform grid structure is defined by two parameters: the dimensions of a single cell in world space (e.g. meters) and the maximum number of entities a cell can affiliate. These parameters are dominated by two requirements, the first being the actual requirements of the expected behavior of pedestrians, second being the hardware requirements for efficient parallel processing on the GPU. Since our approach uses a uniform grid where only the 8 adjacent cells are considered for computing the social forcefield, the dimensions of a single cell have to be at least the size of the cognition range of a pedestrian. Following [HM95] the rejective social force created by an individual has a maximum range where it can influence others. This range can be reduced further by ignoring minimal forces which are not feasible in a running crowd simulation in games. Altogether we calculate a maximum range of 4 to 5 meters which is even smaller as crowd density increases. The cell dimensions are hence set to $4 * 4m^2$ for the standard crowd simulation.

The maximum number of pedestrians per cell is predetermined by two factors. Modern CUDA-GPUs process kernels in a fashion where the massive number of parallel threads is divided into thread blocks. Each thread block has to be executed on a single multiprocessor and shares a certain amount of fast memory, only accessible by threads in this block. Each thread block is further splitted into warps of 32 threads each, which are executed in parallel by one multiprocessor, while other warps of the same threadblock are scheduled to run when the current warp executes multi-cycle operation like loading data from global memory. Thus for optimal performance a maximum number of 32 individuals per cell would be desirable, so each cell can be computed by one warp which provides optimal performance if thread coherence is high. The second factor is the expected density of simulated crowds. Following Herbert Jacobs and related works like [SV99] the estimated density for crowds can be approximated by 1.07 people per m^2 for lose crowds, where individuals can still navigate to any place. A density of $2.39/m^2$ for dense crowds, which are already to dense to move around without major problems and and a maximum of $4.3/m^2$ for a tightly-packed mob, which usually only erupts in panic scenarios. These proportions are summarized in table 1. With our cell size of $16m^2$ we can calculate a maximum value of 17 to 38 individuals per cell, where 38 people would already be too dense for a moving crowd. Since the overall goal of this work is to provide a simulation for games, where the crowd should always be distinguishable and smoothly animated, a maximum number of 32 People per cell is quite enough if we want to avoid heavy congestion, which roughly starts at 30 people per cell given the chosen dimensions.

Table 1: *Cell size for different Crowd-Densities*

Crowd Type	Density per m^2	No. People per Cell
Lose Crowd	1.07	17
Tight Crowd	2.39	38
Packed Mob	4.3	69

Each individual in our simulation needs to store at least its current position and velocity to save its current state. But the individuals also need a personal goal, which can be specified by the AI of the game and serves as an attractive force for the individual. Another additional component needed for rendering is the direction in which the character is currently facing, which not always equals the direction of the current velocity, since people tend to move sideways or even backwards without rotating their body much when evading smaller obstacles in their way. Each space in a cell also needs a flag to indicate if the current space is free or occupied, because storing this information in a central register would lead to too much synchronization overhead. This is also the reason for storing all state-information per individual immediately in the cell space which the person currently occupies. Although this enlarges the memory requirements, current graphics hardware has more than enough memory to accommodate for this. The data layout can be designed with different compression, which always brings down memory requirements, but adds calculation time for decompression.

We tried several layouts as shown in table 2. While Set A

Table 2: *Different Data Layouts for State Data*

Set	A	B	C
State	int (4)	short (2)	char (1)
Goal	int (4)	ushort (2)	short (2)
Position	float2 (8)	ushort2 (4)	char2 (2)
Velocity	float2 (8)	short2 (4)	char2 (2)
Direction	float2 (8)	short2 (4)	char (1)
Overall Size	32 Byte	16 Byte	8 Byte

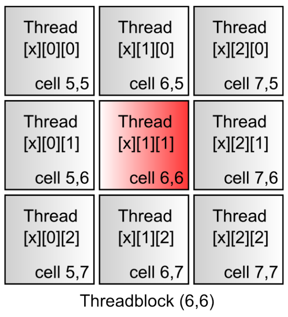


Figure 2: *Thread and Cell indices for thread block 6,6*

consumes 8 times as much memory as a 32-bit Pointer or an Integer array index, Set B and Set C come at a reasonable size. The best overall Performance can be achieved with Set B, providing a compromise between size and compression complexity. All layouts need to be 8-Byte aligned for coalesced memory access, whereas an overall alignment of 128 Bytes per cell provides the best per warp memory access. Set B provides $32 * 16 = 512$ Bytes per cell, so the whole data for one warp can be loaded in 4 coalesced operations. The overall Memory footage is also negligible small with 1 MB for every 2000 Cells so even a huge city could be simulated with the memory consumption of a small texture.

To process the simulation, in each step every individual has to be influenced by the forcefield created from all individuals around him. The influence of all individual forcefields are added up to a single force vector which modifies the current velocity of the individual. It is therefore a gathering approach. To achieve this, each of the up to 32 individuals of each cell has to be influenced by the 31 individuals in the same cell and the 32 individuals of the 9 surrounding cells. This could be solved by 32 Threads in a thread block, each calculating the accumulated influences of all possible 287 individuals. But the GPU design of current CUDA hardware demand certain factors considering the overall number of threads, number of thread blocks and grid dimensions in dependency of the number of registers and shared memory each thread uses. The best performance can be attained by reaching maximum occupancy on the GPU with a high number of threads in a thread block. To use this extra number of threads, each thread will calculate the influence of all individuals in one cell of the 9 cells which can influence someone for a single individual. So the first thread in a thread block will calculate the influence of all individuals in the upper left neighbor cell on the individual in the first space of the current center cell. This has the advantage that the first warp (the first 32 threads of a thread block) all follow the same code path, the only exception being threads which terminate immediately, because their corresponding space is not occupied. In our case a thread block consists of 9 warps, each warp processing the influence of one complete cell serially for each individual in this cell and parallel for all individuals which are influenced.

The 2d grid of cells is mapped directly to the grid of thread blocks in CUDA, so each thread block has the index of the central block which it will be computing influences for. A thread block only changes the values for the individuals in its central block, but it needs reading access to the surrounding 8 blocks as well. Each thread in a thread block has a three dimensional index, the X-Coordinate represents the index of the space in the central block, so all 9 threads with the X-Coordinate 0 will calculate the forces for the individual at space 0 in the central block. The Y-Coordinate represents the column index of the 3x3 grid of cells which surround the central cell. The Z-Coordinate represents the row in this grid *(fig.2)*. So all 32 threads with Y = 1 and Z = 1 will compute the influence of the individuals in the central block on each other. Since warps are always grouped linearly starting with the X coordinate, we always have all Threads concerning the influence of a single block in one warp. This warp loops over all 32 Individuals in this block and computes the influence of one individual on all 32 individuals in the central block parallel with one thread each. This results in 9 accumulated force-vectors for each individual. After computing these 9 vectors, all warps of a thread block need to be synchronized and write these values into shared memory. A single warp proceeds to add up these 9 values for each individual and compute its new velocity and the resulting position. The 8 other warps of the block terminate immediately after their values are written into shared memory.

Because each cell needs access to all individuals of the 8 neighboring cells, changes to these individuals need to be synchronized. Especially the case for individuals traversing to a new cell. Because a number of thread blocks is executed in parallel on different multiprocessors the traversal of a cell could lead to individuals being computed twice, thus creating the problem of two individuals occupying the same space, or an individual being not computed at all in an update for a certain cell, so other individuals could pass right through them. To access this problem a notifying blackboard architecture is implemented. Each individual which is going to traverse to a new cell is marked for traversal by setting its State-Flag to 'traversing'. All computations still consider this individual as an active part of the cell it still inhabits. At the same time an empty space in the desired cell is reserved for the individual. Only one individual can reserve a single space. After the simulation is updated for all cells, a second kernel is executed and writes the marked changes through. So all reserved spaces get occupied by the new individuals and all individuals which were marked as 'traversing' are emptied to free spaces.

This blackboard information is stored directly in the cells. The second kernel could run a lot faster, if the blackboard information was collected in one place. The right amount of threads could be computed beforehand so each thread would eventually access one blackboard-entry and write the changes through in the simulation. But because these threads would still need access to certain cells in the grid, using one thread for each space in each cell is only a little slower, since the memory-access to the grid is coalesced with our method. Another benefit is less synchronization, which is only needed once using the cell-space itself as a blackboard, instead of a single static blackboard for all cells. Access to a single blackboard would have to be synchronized in the first kernel, which is much more computationally expensive. Overall the blackboard kernel only uses a fracture of the overall computation time, so optimization is focused on the simulation kernel.

Although this two kernel method provides a way to handle marking and actually traversing, it still leaves a problem of reserving a new empty space in a cell. If two individuals want to traverse to the same cell and both want to reserve the same space, synchronization needs to guarantee only one individual gets the reservation. This is done by using the AtomicCAS (atomic compare and swap) function of the CUDA-API.

```
for (int j=idx*32; j < (idx+1)*32; ++j)
{
  // Check each space in the new cell
  if (atomicCAS((int*) &people[j].State,
        FREE, RESERVED) == FREE)
  {
    // Mark the space to be freed
    people[i].State = LEAVING
    i = j;
    // Mark the current person
    person.State = RESERVED
    cellchanged = true;
    break;
  }
}
```

The case of an individual wanting to traverse to a new cell which is completely full still remains a problem. In this case the individual is simply stopped from moving to the space of the new cell by clamping its position to the current cell and setting its velocity in the corresponding direction to zero. This can easily lead to an accumulation of individuals wanting to traverse to a cell which is completely full. To avoid this we introduce congestion avoidance in the next step.

4. Introduction of Congestion Avoidance

Following the standard Social Force Model introduced by Helbing in [HM95] pedestrians only react to attractive forces by their goal and other socially interesting targets and repulsing forces by single pedestrians and hindrances. The repulsing forcefield created by other pedestrians has a small radius, which is even smaller in crowded environments, since the clutter obstructs vision and other possibilities to perceive other pedestrians. This leads to a problem, if a pedestrian walks into a crowded mass of people. The high crowd density slows him considerably down, and he is pushed around by a varying and chaotic number of forcefields by the pedestrians in his direct neighborhood. This phenomenon has also been described by Pelechano et al. [PAB07]. The A-stability problem of the explicit integrator can also be observed under those conditions. Although this immediate reaction looks natural and believable at first, the pedestrian is expected to avoid this clutter of people and search a way to his goal traversing territory with lower crowd density, so he can move faster.

Following this reasoning and applying it to our cell based approach a pedestrian needs to recognize cells with high crowd density and try to avoid these. This could be realized by implementing an algorithm to recognize these situations and change the immediate goal of a person. But the seamless way to integrate it is to represent congestion avoidance by a repelling force. This force is emitted by neighboring cells, whose population exceeds a certain limit and gets stronger until it reaches the maximum of 32 individuals in one cell. The lower limit at which a cell starts to project this force is at 21 individuals, which was described by [SV99] as the crowd density at which individuals are starting to get recognized as a lose crowd instead of single pedestrians. Each individual can get repelled by the eight neighboring cells which surround its own cell. Since we already have a warp of threads for each of these cells, each thread of each warp checks if the crowd density is above the congestion limit and if it is adds a repulsive force, pointing straight away from the crowded cell to its force vector.

```
if ( (blockppl > 20 && ppl > 26)
   && (tIdx.y!=1 || tIdx.z!=1))
{
  force.x -= (tIdx.y-1)*(blockppl-20);
  force.y -= (tIdx.z-1)*(blockppl-20);
}
```

Measuring the crowd density of a cell only costs minimal extra calculation time, since each of our cells is already read by 32 threads in a threadblock. The CUDA API provides a function __ballot(bool b) which checks a certain condition for all threads in a warp in a single computation (see

[NVI08]). The result of this condition check is returned to all threads in the warp, which can easily determine the number of threads fulfilling the condition and hence the number of active pedestrians in a cell.

Although this implementation creates the desired effect for situations with small clutters of people, which are avoided by other individuals, it still lacks another phenomenon observed in tight crowds. If people are in a crowd, they are not only repelled by groups with even higher density, but they actively search less crowded space to walk as it has also been explained by Latif et al. [LW04]. To simulate this behavior we introduce an additional attractive force, which is calculated exactly the same way as the congestion avoidance force. But this force draws individuals to less crowded cells and we call it congestion dispersion. This force only activates if the current cell of an individual has at least a certain crowd density. The limit for activation of this force is 26 people in a cell, which is half way between the congestion avoidance limit and the maximum number of people in a cell. This attractive force has the benefit of not only letting new individuals avoid a congested space, but also draws pedestrians out of the congested environment so congestions are dispersed quickly.

5. Results

In this section we discuss visual and performance results of the CPU and GPU implementation of our crowd simulator with and without congestion avoidance. Furthermore we will analyze technical impact factors. One of them is the amount of thread blocks we use per cell. As a further performance indicator for the congestion avoidance we analyze the average effective velocity of the simulated pedestrians. The effective velocity of a pedestrian is defined by the euclidean distance between its start and ending position divided by the time needed for the journey.

Introducing congestion avoidance and crowd dispersion results in visually smoother simulations with more homogeneous crowds. Twitchy and unstable behavior is less likely to occur. In figure *(fig. 3)* we show simulation results with and without congestion avoidance and crowd dispersion. The first picture shows the simulation of a dense crowd with randomly placed goals after 1 Minute of simulation time. Black arrows indicate pedestrians in densely crowded areas and gray arrows in light crowded areas. Densely crowded areas are defined as 1.07 persons per m^2 or more (see table 1). Although the crowd starts at random positions with random goals, most of the paths from an individual to its goal cross in the center of the simulated environment. This leads to many individuals meeting in the center all on their way to opposing directions, meaning they have to find a way through or around the crowd. The social force model without congestion avoidance leads to a huge clutter of people, which get almost completely stuck and block each other from advancing. Since we have a constraint of a maximum crowd density,

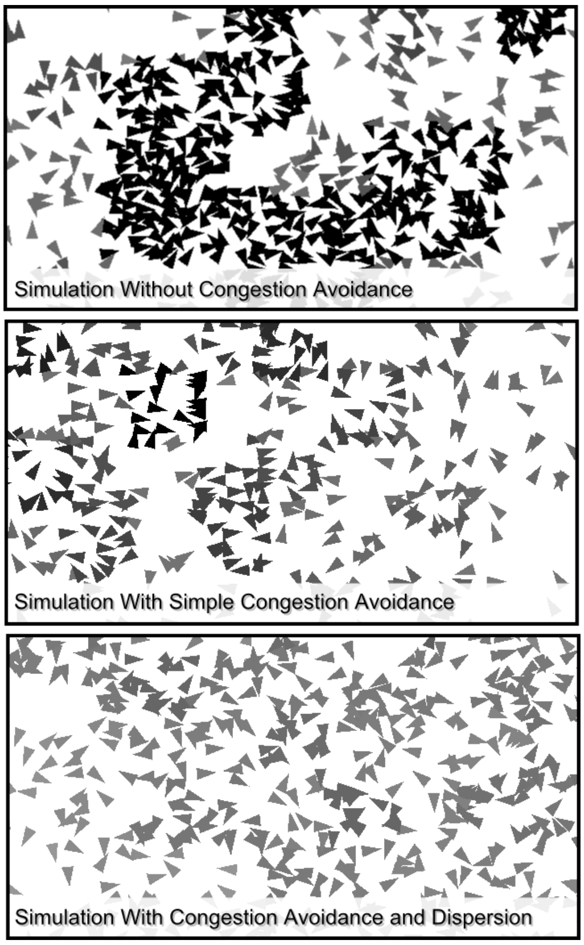

Simulation Without Congestion Avoidance
Simulation With Simple Congestion Avoidance
Simulation With Congestion Avoidance and Dispersion

Figure 3: *The results of congestion avoidance and congestion dispersion*

and limit the velocity of individuals in accordance to their cells crowd density, most of the forces cancel each other out and most individuals trying to traverse the center get stuck in a growing clutter and integration stability problems occur.

After implementing Congestion Avoidance we get a result as shown in the second picture, after running the simulation with equal starting conditions for one minute. Most of the individuals can traverse the center by evading small dense groups of opposing individuals. Although these small clutters are avoided by other individuals and hence can't grow, they still emerge randomly in the center and a small number of individuals gets stuck for a while. The random forces from other individuals passing by result in the breakup of these clutters by chance, but it sill doesn't seem very natural, since an intelligent agent could easily communicate with other individuals to breakup these clutters as soon as they emerge. To get this desired behavior Collision Dispersion is imple-

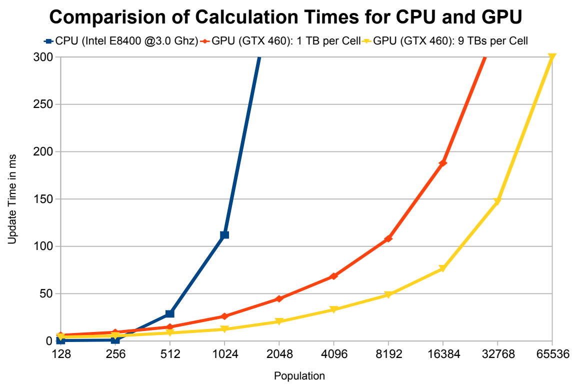


Figure 4: *Performance Chart for Random Start, Random Goal Simulation*

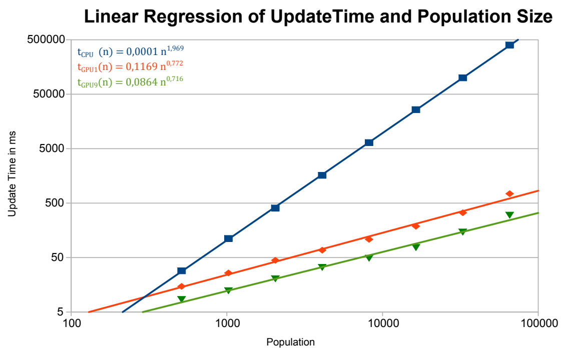


Figure 5: *Logarithmic Performance Chart with trend lines*

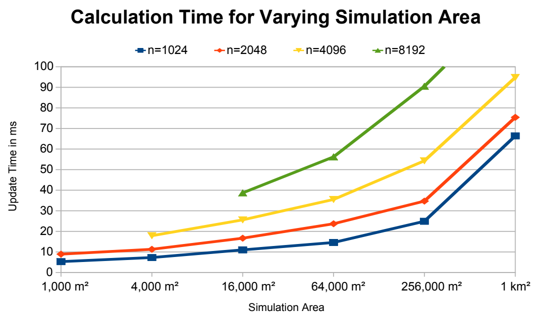


Figure 6: *Performance chart for varying areas and population sizes*

mented and most of the individuals evade situations with high crowd density, so congestions are quickly dispersed.

To analyze the performance of the algorithm we first measured simulation update times for the CPU and the GPU implementation under varying numbers of crowd participants. The used CPU implementation is a straight forward n body particle simulator. The resulting measurements are shown in figure *(fig. 4)*. As one can see from the graph the relation between the crowd size and the update time is of the form $t = a \cdot n^b$ where n is the amount of participants in the crowd. This correlation is expected, since the n-body simulation has a $O(n^2)$ complexity. This complexity remains the same even with a distance cut off in simulation. As the overall density increases linearly with the total amount of crowd participants, the amount of participants in the local vicinity increases also linearly. Therefore the problem remains of $O(n^2)$ complexity.

Displaying the logarithmic charts of the values transforming the formula into $ld(t) = ld(a) + b \cdot ld(x)$ gives us a linear correlation. Following a regression analysis of this data, we can approximate the measured values with trend lines. The coefficient of determination as the variance of our measured values to the trend line *(fig. 5)*. The values for 128 and 256 people are left out, because caching speeds up the calculation for small populations. However when analyzing these values separately, similar results are achieved. The results are $t(n) = 0.0001 \cdot n^{1.969}$ for the straight forward CPU implementation, which approximately fits the expected $O(n^2)$ complexity. Furthermore $t(n) = 0.1169 \cdot n^{0.772}$ for the Single-Threadblock GPU implementation and $t(n) = 0.0864 \cdot n^{0.716}$ for the 9-Threadblock GPU implementation. This suggests that the optimized GPU implementation not only calculates the simulation with less than linear complexity, but also a big speedup by full occupancy on the GPU achieved by multiple thread blocks. One can expect that once a certain threshold of total crowd participants has been reached the problem becomes quadratic again. This will be the case when the parallel computing capability of the used GPU has been totally exploited.

Although the cell based simulation excels in performance for increasing crowd sizes, it imposes a new limitation. Since the algorithm we use is based on a fixed size grid, its overall dimensions are determined by the size of the area to be simulated. 9 threadblocks are started for each cell in the whole simulation. Even though all threadblocks which are started for empty cells terminate immediately, they still require access to the information if this cell is currently empty and hence access the device memory, which needs some processing cycles. What's more, a huge area with only a few individuals to simulate still needs a high amount of GPU processing time, since all the threadblocks are still started and occupy the multiprocessors, even if only one execution path is valid, but this is a problem all parallel calculation approaches face. To measure the performance of the simulation hence depends on two factors: population size and simulation area. The performance for varying areas can be observed in figure *(fig. 6)*

This chart indicates a good performance for relatively small densely crowded areas, like pedestrian zones or side-

Table 3: *Average Effective Velocity in km/h on $64x64m^2$*

	1000	2000	3000	4000	5000
Std. Model	2.03	1.79	0*	0*	0*
C. Avoidance	2.01	1.45	1.14	0.80	0.62
C. Dispersion	2.02	1.69	1.24	1.05	0.85

walks. These measurements show that the cell based simulation can easily simulate up to 3000 individuals at 60 fps, if the simulation area doesn't exceed $5000m^2$ in a quadratic pedestrian zone or up to 1000 individuals in a zone up to $64000m^2$. These sizes are measurements of fully simulated quadratic areas, whereas pedestrian zones are usually a complex patchwork of small streets, junctions and areas. This increases the area which can be simulated in a virtual city, since only the pedestrian areas need to be mapped by the simulation grid.

Congestion avoidance has been introduced to make the simulation without dynamic cell resizing and sorting possible. Additionally we can also produce more visually pleasing results. Congestion avoidance improves the simulation by making it more natural looking and less likely to twitch or come to a halt. One way of estimating the quality of the congestion avoidance is watching the actual simulation for a natural looking behavior and steady movement and distribution of the crowd. Since these factors are hard to measure in numbers, we decided to measure the average effective velocity of the pedestrians in the simulation. This velocity is a good indicator of how often an individual got stuck on the way to his goal and how much of a hindrance the crowd was for each of its members. The simulation was launched with randomized starting positions and goals. But nonetheless several runs of the simulation all even out to the same average effective velocity, given the same simulation area, crowd size and congestion avoidance settings. These measurements can be observed in table 3

The velocity entries flagged $0*$ in Table 3 mark constellations where the crowd density is too high for the unmodified social force model. Most of the pedestrians eventually get stuck in areas with high trough traffic and can only escape by pure chance, if the forces of passing individuals free them. But these areas grow over time, so the effective velocity of these stuck individuals approximates zero, since they never reach their goal. This is already the case for a lose crowd of 3000 people on $4096m^2$ which get stuck without congestion avoidance. As expected congestion avoidance solves this problem and can achieve an average effective velocity of $1.14km/h$. Crowd Dispersion can even further enhance this effect by achieving an average effective velocity of $1.24km/h$ under the same conditions. While these additions improve the flow in dense crowds and have now measurable effect on the simulation of very few individuals they actually slow down individuals in a lose crowd of about 0.5 people per m^2. This can be attributed to the fact, that these individuals take slight detours to evade crowded spots, which could become congestions. In a lose crowd individuals can still pass right through these spots, without the danger of getting stuck, but when the crowd gets more dense, these spots become real congestions and the pedestrians get completely stuck.

The overall simulation looks very believable for all crowd densities, where pedestrians try to avoid congested spots and clutter before narrow points eagerly waiting to traverse. The simulation is also very stable and even after long runs or with gaps in performance the result looks believable and fitting for an in game simulation of pedestrians.

6. Conclusion and Future Work

In this paper we have shown that the introduction of congestion avoidance into CUDA based crowd simulation leads to significant advantages. Congestion avoidance provides a natural mechanism to limit crowd densities. The aspect of visually pleasing results has been additionally objectified by measuring the effective speed of crowd participants that gets significantly increased when those techniques are applied. The positive impact of congestion avoidance can further be enhanced by crowd dispersion that is a selective limit to congestion avoidance.

Additionally to obtaining visually pleasing results we get several stability and performance benefits. The most important aspect is the performance benefit as a global sorting and assignment of crowd members to worker threads is avoided. The change of crowd members between simulation cells can happen asynchronously over a blackboard architecture. The only point where one has to pay attention is that a cell does not get overcrowded by people entering asynchronously from different blocks. This is handled elegantly with a compare and swap (CAS) command.

We have proven that performance improvements and smoother crowd movement can be obtained by applying congestion avoidance with several measurements.

Experiments have shown that for optimal performance gain the optimization of data structures used in local memory for a thread block is important. One has to balance memory consumption and time needed for compresison and decompression. The technical constraints for optimizations on graphics hardware are complex. Experimentation with different configurations is required for optimal performance.

As future work there are several extension possibilities to this system. The easiest one should be the inclusion of obstacles. Static obstacles integrate well into the social force model, as they represent a repulsor to crowd agents. The only problem with this approach as with most steering behaviors is that one has to pay attention that crowd members do not get stuck on local minima.

The problem of local minima can usually be avoided in a game context, as coarse granular navigation can often be described by a navigation graph that gets edited by the level designer. This paradigm is usually combined with some steering behavior in todays games. The steering behavior is sometimes based on a social force model. Therefore it would be a possibility to combine the CUDA force model simulation with the queing simulation. Introducing a coarse granular navigation system also provides the possibility to restructure the uniform grid to accomodate the established link structure of the coarse granular navigation system. This way memory consumption can be reduced and additional performance can be gained.

When one takes a look at the CPU time consumed by crowd simulators in actual games, one can observe that a significant portion of those resources is devoted to the animation system. As we want to simulate many crowd actors implementing a special low fidelity and low resource hungry animation system would be an option here. In this case we do not need a full blown animation tree system. A simplified system would suffice in this case, that could also be moved to the GPU for efficient animation computation. Extending this framework for more complex simulations will be part of our future research.

References

[BMR*96] BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P., STAL M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996. 2

[DLR07] D'SOUZA R. M., LYSENKO M., RAHMANI K.: SugarScape on steroids: simulating over a million agents at interactive rates. 2

[EA10] EA: Army of two: The 40th day. [PlayStation 3, XBOX360], 2010. 1

[EFSC09] ERRA U., FROLA B., SCARANO V., COUZIN I.: An efficient gpu implementation for large scale individual-based simulation of collective behavior. *2009 International Workshop on High Performance Computational Systems Biology 0* (2009), 51–58. 2, 3

[HM95] HELBING D., MOLNÁR P.: Social force model for pedestrian dynamics. *Phys. Rev. E 51*, 5 (May 1995), 4282–4286. 1, 3, 4, 6

[KMKWS00] KLÜPFEL H., MEYER-KÖNIG T., WAHLE J., SCHRECKENBERG M.: Microscopic simulation of evacuation processes on passenger ships. In *Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry: Theoretical and Practical Issues on Cellular Automata* (London, UK, 2000), Springer-Verlag, pp. 63–71. 1

[KmWH10] KIRK D. B., MEI W. HWU W.: *Programming Massively Parallel Processors*. Elsevier Inc., Burlington, USA, 2010. 3

[KR02] KRAUSE J., RUXTON G.: *Living in Groups*. Oxford University Press, USA, 2002. 3

[LKF05] LAKOBA T. I., KAUP D. J., FINKELSTEIN N. M.: Modifications of the helbing-molner-farkas-vicsek social force model for pedestrian evolution. *Simulation 81* (May 2005), 339–352. 3

[LW04] LATIF M. S. A., WIDYARTO S.: The crowd simulation for interactive virtual environments. In *Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), VRCAI '04, ACM, pp. 278–281. 7

[NVI08] NVIDIA: *NVIDIA CUDA Programming Guide 2.0*. 2008. (NVIDIA Developer Site) `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`. 1, 7

[NVI11] NVIDIA CORPORATION: *CUDA C Best Practices Guide*, 4.0 ed. 2701 San Tomas Expressway, Santa Clara 95050, USA, May 2011. 3

[PAB07] PELECHANO N., ALLBECK J. M., BADLER N. I.: Controlling individual agents in high-density crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 99–108. 3, 6

[RCR09] RICHMOND P., COAKLEY S., ROMANO D. M.: A high performance agent based modelling framework on graphics card hardware with CUDA. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems* (Richland, SC, 2009), International Foundation for Autonomous Agents and Multiagent Systems, pp. 1125–1126. 2

[Rey87] REYNOLDS C. W.: Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 25–34. 1

[Rey06] REYNOLDS C.: Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), Sandbox '06, ACM, pp. 113–121. 4

[RMH05] RUDOMIN I., MILLAN E., HERNANDEZ B.: Fragment shaders for agent animation using finite state machines. simulation modelling practice and theory. *Journal Elsevier 13* (2005), 741–751. 2

[RR08] RICHMOND P., ROMANO D. M.: Agent Based GPU, a Real-time 3D Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the GPU. 2

[SCE10] SCE: Heavy rain. [PlayStation 3], 2010. 1

[SN09] STRIPPGEN D., NAGEL K.: Using common graphics hardware for multi-agent traffic simulation with CUDA. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (Rome, Italy, 2009). 1, 2

[SV99] S.A. VELASTIN R.A. LOTUFO A. M. L. D. F. C.: Estimating crowd density with minkowski fractal dimension. *Acoustics, Speech, and Signal Processing 6* (March 1999). 4, 6

[Ubi07] UBISOFT: Assasin's crreed. [PlayStation 3, XBOX360], 2007. 1