

Supermassive Crowd Simulation on GPU based on Emergent Behavior

Erick Baptista Passos
UFF, Medialab

Mark Joselli
UFF, Medialab

Anselmo Montenegro
UFF, Medialab

Marcelo Zamith
UFF, Medialab

Aura Conci
UFF, Medialab

Jack Rocha
UFF, Medialab

Bruno Feijo
PUC-RIO, ICAD Games

Esteban Walter Gonzalez Clua
UFF, Medialab

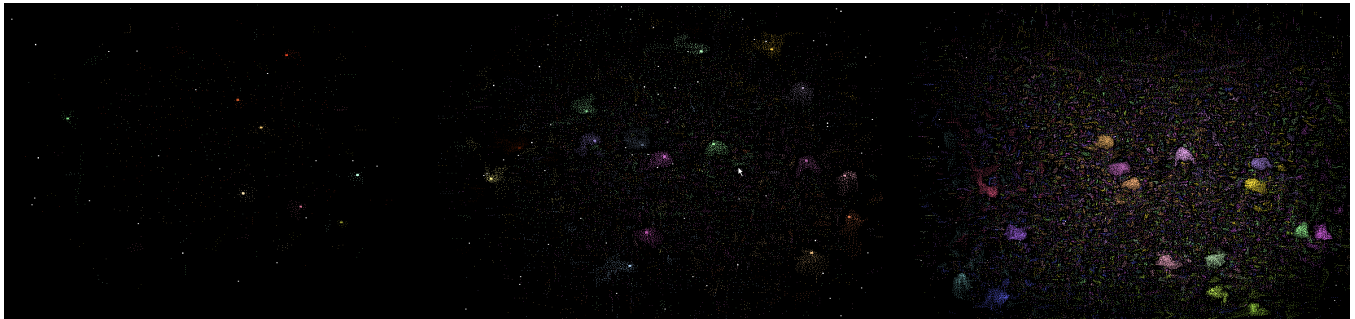


Figure 1: Screenshots of a boids simulation with variable number of entities

Abstract

Computing and presenting emergent crowd simulations in real-time is a computationally intensive task. This intensity mostly comes from the $O(n^2)$ complexity of the traversal algorithm needed for the interactions of all elements against each other based on a proximity query. By using special data structures such as grids, and the parallel nature of graphics hardware, relevant previous works reduces this complexity by considerably factors, making it possible to achieve interactive frame rates. However, existent proposals tend to be heavily bound by the maximum density of such grids, which is usually high, yet leading to arguably inefficient algorithms. In this paper we propose the use of a fine grained grid and accompanying data manipulation that leads to scalable algorithmic complexity. We also implement a representative flocking boids case-study from which we run benchmarks with more than 1 million simulated and rendered boids at nearly 30fps. We remark that previous works achieved not more than 15,000 boids with interactive frame rates.

Keywords:: GPGPU, CUDA, Crowd Simulation, Cellular Automata, Flocking Boids

Author's Contact:

epassos,mjoselli,mzamith,esteban,anselmo,aconci@ic.uff.br
jack.f.rocha@gmail.com
bruno@inf.puc-rio.br

1 Introduction

Visual simulations and 3D games are growing fast in terms of content and visual accuracy due to the increasing power of graphics hardware and computers architecture. One consequence of this evolution is that users expectations are now much more sensible when it comes to judging if a simulated entity behavior is overall believable. At the same time, visual applications that were only available at not real time applications regarding to the technology limitations are now becoming possible to run in real-time systems. For this reason the seek for more detailed graphics and more realistic appearance have been a easy to grasp trend, but there has also been a growing interest in more complex animation and Artificial Intelligence algorithms.

While the real world environments can be composed by thousands, or even millions of moving entities, a simulated one is usually constrained by a limited number of them. In a typical environment in the nature one can find a huge number of different animals or even cells interacting between them and with physics elements. This can happen in very different situations, like in a sports arena that is full

of autonomous individuals, ants or bee communities or even cells in a blood system. When simulating a similar virtual and real time scene, bound by available computational power, it is common to find a very limited number of independent entities, most of them behaving very predictably. There are several approaches that aim to include more unpredictable behavioral models in simulated environments, such as [Reynolds 1987; Musse and Thalmann 1997; Shao and Terzopoulos 2005; Pelechano et al. 2007; Treuille et al. 2006].

On the other hand, given its computational requirements, visual improvements have been made possible by the clever use of evolving parallel graphics hardware technology. Even being graphics content driven, many today 3D applications have bottlenecks stipulated by the CPU, which is responsible for non-graphics calculations. The use of graphics processing units (GPUs) for general purpose computing has become a new and interesting research area that continuously gains attention from industry and also academia, in order to resolve in a huge parallel architecture problems that are not related to graphics. Behavioral Artificial Intelligence algorithms, although traditionally sequential and executed on the CPU are sometimes suitable, but not easily, to parallel execution. In dynamic emergent crowd simulation, algorithms are driven by the need to avoid the $O(n^2)$ complexity of the proximity queries between active entities. Several approaches have been proposed to cope with this issue [Reynolds 2000; Chiara et al. 2004; Courty and Musse 2005] but none of them has reached an ideal level of scalability. It is also important to notice that no work until the present date has proposed the real time simulation of more than just a few thousands of complex entities that interact with each other. Applications for such technique ranges from crowd behavior prediction in case of a stadium fire or street traffic simulation, interactions between different blood cells and enrichment of computer game worlds.

GPUs are specialized devices for graphics computation, often comprising a set of SIMD (Single Instruction, Multiple Data) processing units due to the parallel nature of this task. The constant development of these devices, pushed mainly by the computer games industry, turned them fast enough to be appropriate for solving other computational intensive problems. The broad adopt of the term GPGPU (General Purpose computation on GPUs) to name this new field of research shows its importance. However, the first applications of GPUs to do general purpose computing had to rely on the adaptation of graphics rendering APIs to different concepts, representing a difficult learning curve to developers. The CUDA technology [Nvidia 2007] aims to provide a new abstraction layer on top of (former) graphics hardware to facilitate its use for non graphical processing. While being an important tool to enable a faster modeling of problems to a more general parallel machine concept, CUDA abstracted hardware still is very specialized, carrying a dif-

ferent memory model, with huge impacts on the performance of the developed application, depending on its memory access patterns.

GPUs are SIMD processors that take advantage of the streamed nature of graphics computation, where the processing of each subsequent pixel would require localized data from textures. This specialized hardware uses a great deal of read-ahead and caching techniques to accelerate this computation based in this localization. There are rules of thumb to create efficient streamed applications, the most important being to structure the data in a way to maximize memory reads based on locality and avoiding random writes. These rules enable an efficient use of available cache memory and read ahead mechanisms of these devices.

In this paper we propose a novel simulation architecture for crowds of autonomous geometric entities that are based on a partially sorted matrix. The entities are simulated as Cellular Automata with Extended Moore Neighborhood [Sarkar 2000] over this matrix, which is ideal for the memory model of GPUs. The high performance and scalability are achieved by a very low parallel complexity and independence of the sorting and simulation algorithms. We call this architecture Supermassive Crowd Simulation. To model the data structures and the simulation technique we use a traditional emergent behavior model of flocking boids [Reynolds 1987] but the architecture can be further extended to other simulation models that rely dynamic autonomous entities.

The rest of the paper is organized as follows: Section 2 discusses related work on general purpose computation on GPU and emergent behavior models. Section 3 explains the data structures and simulation steps while section 4 describes the particular behavior model used to validate the proposed architecture. Section 5 brings benchmark results and performance analysis compared to the same simulation on a CPU. Finally, section 6 concludes the paper with a discussion on future work.

2 Related work

This work has two different goals and contributions in crowd behavior simulation. The first one is to provide data structures and an architecture that are suitable for believable modeling of observable behavior in the real world. The second goal is to extract the best performance possible of current hardware. These two goals are shared among the majority of the cited research. Hence, this section is organized in approximate chronological order.

The first known agent-based simulation for groups of interacting animals is the work proposed by Craig Reynolds [Reynolds 1987], in which he presented a distributed behavioral model to perform this task. His model is similar to a particle system where each individual is independently simulated and acts accordingly to its observation of the environment, including physical rules such as gravity, and influences by the other individuals perceived in the surroundings. The main drawback of the proposed approach is the $O(n^2)$ complexity of the traversal algorithm needed to perform the proximity tests for each pair of individuals. At the time, this was such an issue that the simulation had to be run as an offline batch process, even for a limited number of individuals. In order to cope with this limitation, the author suggested the use of spatial hashing. This work also introduced the term *boid* (abbreviation for birdoid) that has been used to designate an individual in animal crowd simulations ever since.

Musse and Thalmann [Musse and Thalmann 1997] propose a more complex modeling of human motion based on internal goal-oriented parameters and the group interactions that emerge from the simulation, taking into account sociological aspects of human relations. Others include psychological effects [Pelechano et al. 2007], social forces [Cordeiro et al. 2005] or even knowledge and learning aspects [Funge et al. 1999]. Shao and Terzopoulos [Shao and Terzopoulos 2005] extend the latest including path planning and visibility for pedestrians. It is important to mention that these proposals are mainly focused on the correctness aspects of behavior modeling. While serving as foundations on the subject, these structures and algorithms are not suitable for real-time simulation of very large crowds, which is one of the goals of this papers.

Reynolds further developed his behavioral model to include more

complex rules and to achieve the desired interactive performance by the use of spatial hashing [Reynolds 2000; Reynolds 1999]. This implementation could simulate up to 280 boids at 60fps in a playstation 2 hardware. By using the spatial hash to distribute the boids inside a grid, the proximity query algorithm could be performed against a reduced number of pairs. For each boid, only those inside the same grid cell and at adjacent ones, depending on its position were considered. This strategy led to a complexity that is close to $O(n)$. This complexity value, however, is highly dependent on the maximum density of each grid cell, which can be very high if the simulated environment is large and dense. We remark that the complexity of our data structure is not affected by the size of the environment or the distribution of the boids over it.

Quinn et al. [Quinn et al. 2003] used distributed multiprocessors to simulate evacuation scenarios with up to 10000 individuals at 45fps on a cluster connected by a gigabit switch. More recently, a similar spatial hashing data-structure was used by Reynolds [Reynolds 2006] to render up to 15000 boids in playstation 3 hardware at interactive framerates, with reduced simulation rates of around 10fps. Due to the distributed memory of both architectures, it is necessary to copy compact versions of the buckets/grids of boids to the individual parallel processors before the simulation step and copying them back at the end of it, leading to a potential performance bottleneck for larger sets of boids. This issue is evidenced in [Steed and Abou-Haidar 2003], where the authors span the crowd simulation over several network servers and conclude that moving individuals between servers is an expensive operation.

The use of the parallel power of GPUs to this problem is very promising but brings another issue, related to its intrinsic dependency on data-locality to achieve high performance. For agent-based simulation that relies on spatial hashing, it is desired that the individuals should be sorted through the containing data-structure based on their cell indexes. The work by Chiara et al. [Chiara et al. 2004] makes use of the CPU to perform this sorting. To avoid the performance penalty, this sorting is triggered only when a boid departs from its group, which is detected by the use of a scattering matrix. This system could simulate 1600 boids at 60fps including the rendering of animated 2D models. The FastCrowd system [Courty and Musse 2005] was also implemented with a mix of CPU and GPU computation that could simulate and render a crowd of 10000 individuals at 20fps as simple 2D discs. Using this simple rendering primitive, the GPU was capable of simultaneously computing the flow of gases on an evacuation scenario. We also make use of the fact that groups tend to move as blocks in crowd simulations and, as will be explained in next sections, use a parallel partial sorting algorithm on the GPU to achieve even higher performance.

The simulation architecture and data-structures of [Treuille et al. 2006] depart from the agent-based models presented so far. It uses a 2D dynamic field to represent both the crowd density and the obstacles of the environment. The individuals navigate through and according to this continuum field. It is argued that locally controlled agents, while providing for complex emergent behavior, are not an appropriate model for goal-driven individuals, such as human pedestrians. The implemented system could simulate up to 10000 humans at 5fps (without graphics) even with the inclusion of a dynamic environment such as traffic lights. The continuum field is an interesting approach but limits the environment to a pre-determined size.

Our architecture stores the entities/boids data over a matrix, with an individual cell corresponding to exactly one boid and the simulation occurring in parallel on the GPU. In the implemented system, each boid is modeled as an agent that, based on an Extended Moore Neighborhood [Sarkar 2000], perceives a constant number of other surrounding boids. This cellular automaton model matches perfectly with the data-locality dependency of graphics hardware but imposes that boids data have to be kept spatially sorted over the matrix during simulation. Our proposal, such as most of the above work, is based on distributed agents to yield emergent behavior, but the novel data-structures is prepared for unlimited environment size and better scalability.

3 Simulation Architecture

Individual entities in crowd behavior simulations depend on observations of their surrounding neighbors in order to decide actions to take. The straightforward implementation of the neighborhood finding algorithm has a complexity of $O(n^2)$, for n entities, since it depends at least on distance tests performed for all entity pairs in the crowd. Individuals are autonomous and move during each frame, which leads to a very computationally expensive task.

Techniques of spatial subdivision have been used to group and sort these entities to accelerate the neighborhood finding task. Current implementations are usually based on variations of relatively coarse Voronoi subdivisions, such as a grid. After each update, all entities have their grid cell index calculated. For GPU based solutions, some kind of sorting has to be performed, so geometric space neighbors are grouped together in the used data structure. However, static Voronoi structures have some limitations related to the simulation of very large geometric spaces, where each cell node may have a large number of entities inside. This issue makes the neighborhood finding problem again limited by a hidden $O(n^2)$ complexity factor.

Our architecture is built around a fine grained dynamic data structure that is used to store information about all entities and is sorting a simulation agnostic, meaning one can use it with different sorting strategies and simulation algorithms. The following subsections describe these data structures, the role of sorting and the types of simulation algorithms suitable to the proposed architecture.

3.1 Neighborhood Matrix

The proposed architecture was developed with the CUDA technology [NVIDIA 2007], in order to keep the process entirely at the GPU. To assure the desired high performance, all information about entities is organized in matrixes that are mapped as textures. Follows the minimum information required for each entity: Position - a 2D vector, representing the position of the entity; Speed - a vector for storing the direction and velocity in a single structure; Type - an integer that can be used to differentiate entity classes.

These entities information are stored in matrixes, where each index contains values for an individual entity. Since is possible to be required a variable number of data about the entities, it may be necessary to use more than one matrix. However it is a good practice to avoid wasting GPU memory by sharing vectors to store more than one single piece of information in each index.

The matrix containing the position vector for the entities is then used as a sorting structure. In Figure 2, one can see an example of such matrix where information about the position of 36 individual entities. To reduce the cost of proximity queries, each entity will have access to the ones surrounding its cells based on a given radius. In the example, the radius is 2, so the entity represented at cell (2,2) would have access to its 24 surrounding entities only. In Cellular Automata, this form of information gathering is called Extended Moore Neighborhood [Sarkar 2000].

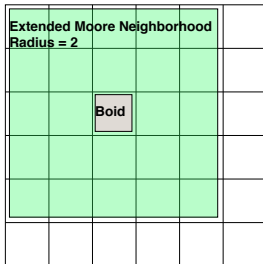


Figure 2: Neighborhood matrix

This structure enables the exact prediction of the performance, since the number of proximity queries will be constant over the simulation. This happens because instead of making distance queries, taking as parameters all entities inside its own coarse Voronoi cell

and the ones in the adjacent regions, as in traditional implementations, each entity would query only a fixed number of surrounding individual matrix cells. However, this matrix has to be sorted continually in such a way that neighbors in geometric space are stored in cells close to each other. This guarantees that this extension of cellular automata may gather information about close neighbors.

One can notice that the use of a matrix is tied to the two dimensions of this particular case. For a 3D simulation the data structure could be a 3 dimensional array, with no loose of generality, similar to the usage of quadrees or octrees in other applications of spatial subdivisions. Since in this work the implementation of every entity is mapped to one CUDA thread in both the sorting and simulation steps, it is important to mention that these matrixes are double buffered, so that each of these tasks does not write data over the input structures that can still be read by other CUDA threads.

3.2 Sorting Pass

The matrix that stores position information is used to perform a topological sorting over two dimensions of these vectors. The goal is to store in the top-leftmost cell of the matrixes the entity with the smaller values for X and Y, and the bottom-rightmost cell to the entity with highest values of X and Y respectively. Using both values to sort the matrix, the top lines will be filled with the entities with higher values of Y while the left columns will store those with lower values for X and so on. This kind of sorting allows builds automatic approximate proximity query based on data locality.

When performing a sorting over an one dimension array of float point values, the rule is that given an array A, the following rule must apply at the end:

- $\forall A[i] \in A, i > 0 \Rightarrow A[i-1] \leq A[i]$.

Extending this rule to a matrix M where each cell has two float point values X and Y:

- Eq.1: $\forall M[i][j] \in M, j > 0 \Rightarrow M[i][j-1].X \leq M[i][j].X$;
- Eq.2: $M[i][j-1].X = M[i][j].X \Rightarrow M[i][j-1].Y \leq M[i][j].Y$;
- Eq.3: $\forall M[i][j] \in M, i > 0 \Rightarrow M[i-1][j].Y \leq M[i][j].Y$;
- Eq.4: $M[i-1][j].Y = M[i][j].Y \Rightarrow M[i-1][j].X \leq M[i][j].X$;

The pass must be divided into four steps, one for odd and one for even elements for each of both directions. The first step runs the sorting between each entity position vector of the even columns against its immediate neighbor in the subsequent odd column, based first on the values of the X component. If it rules of Eq.1 or Eq. 2 are violated, the entities switch cells in the matrixes. It is important to notice that not only the cells in the position matrix have to be switched, but all data that is kept at the others as well, otherwise a violation of the data structure will occur. The other three sorting steps perform the same operation for even columns, odd lines and even lines, respectively.

As shown above, the sorting has to be performed in both directions. This process is sorting agnostic, which means that it is possible to use different sorting strategies, as long as the rules above are eventually or partially achieved during simulation. A partial sorting strategy was tested using only one pass of a parallel implementation for the odd-even transposition sort algorithm at each simulation step. The odd-even transposition sort is similar to the bubble sort algorithm and is possible to complete the pass, traversing the whole data structure, in a $O(n)$ sequential time. Because there are two steps, one for odd and other for even elements, this algorithm is suitable for parallel execution. In Figure 3 it is shown a schematic presentation of a complete odd-even transposition sort pass.

As seen from Figure 3, the four-step pass does one sort for odd and one for even elements on both directions. The first step runs the sorting between each entity position vector of even columns against its immediate neighbor in the subsequent odd column based first on the values of the X component. If it happens that sorting rules 1 or 2

Sorting Steps

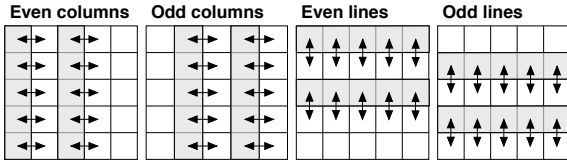


Figure 3: Partial sorting pass with 4 odd-even transposition steps

are violated, the entities switch cells in the matrixes. It is important to notice that not only the position vector have to be switched, but all data that is kept at the matrixes, otherwise violating the safety of the data structure. The other three sorting steps perform the same operation for even columns, odd lines and even lines, respectively.

At the present work it was sufficient to run only one complete odd-even pass for each simulation frame because we initialized the position matrix in an ordered state and, the flocking nature of the simulation algorithm imply that the entities do not overlap positions frequently. In practice, this means that in a very few simulation steps, the matrixes correctly represent the proximity relations between them. Depending on the simulation being performed, it may be necessary to perform a complete sorting at each frame step. In this case, it is recommended a sorting algorithm with better worst case complexity, such as a parallel merge sort. However, for all test scenarios built in this work, the incomplete odd-even transposition pass was enough to sustain a approximately correct simulation, with no visual noticeable artifacts, and with a parallel complexity (and performance) of $O(1)$ against $O(\log n)$ of a complete sorting based on a parallel merge sort.

3.3 Simulation Pass

The simulation pass can perform any kind of crowd emergent behavior of entities that are constrained to the knowledge of data from their neighborhood, such as flocking boids or even dynamic fluids. This pass must be implemented as a CUDA kernel function that receives as arguments at least the neighborhood matrixes (double buffered as input and output) and the time passed since the last step.

4 Case-Study: Flocking Boids

For the purpose of this work, we chose to validate the proposed technique by implementing a well known distributed simulation algorithm called, flocking boids [Reynolds 1987]. This is a good algorithm to use because of its good visual results, proximity to real world behavior observation of animals and understandability. The implementation of the flocking boids model using our algorithm enables a real time simulation of up to one million animals of several species, with a corresponding visual feedback. The number of different species is limited only by the number of animals in the simulation.

Our model simulates a crowd of animals interacting with each other and avoiding random obstacles around the space. This simulation can be used to represent from small bird flocks to huge and complex terrestrial animal groups or either thousand of hundreds of different cells in a living system. Boids from the same type (representing the species) try to form groups and avoid staying close to the other type of species. The number of simulated boids and types is limited only by technology but, as demonstrated in the next section, our method scales very well due to the data structures used. In this section we focus at the extension of the concepts of cellular automata in the simulation step, in order to represent emergent animal behavior.

To achieve a believable simulation we try to mimic what is observable in nature: many animal behaviors resemble that of state machines and cellular automata, where a combination of internal and external factors defines which actions are taken and how they are made. A state machine is used to decide which actions are taken.

The actions themselves performed by a cellular automaton algorithm. With this approach, internal state is represented by the boid type and external ones corresponds to the visible neighbors, depending from where the boid is looking at (direction), and their relative distances.

Based on this ideas, our simulation algorithm uses internal and external states to compute these influences for each boid: Flocking (grouping, repulsion and direction following); leader following; and other boid types repulsion (used also for obstacle avoidance). Additionally, there are multiplier factors which dictate how each influence type may get blended to another, in each step. In order to enable a richer simulation, these factors are stored independent for each type of boid in separate arrays. These arrays are indexed by the boid type, meaning that the element at position 1 represents the information that the array keeps for all boids of type 1. These arrays size corresponds to the number of different boid types. Below is a summary of the information kept for each boid type:

- Multiplier factors, one for each type of influence;
- Neighborhood matrix cell index for the leader of the type.

4.1 Vision

In nature, each animal species has a particular eye placement, evolved based on its survival needs such as focusing on a prey or covering a larger field of view to detect predators. To mimic this fact, our boids have a limited field of view, parameterized by an angle. Obstacles and other boids outside this field of view are not considered in the simulation. Figure 4 shows a comprehensible representation of this field of view.

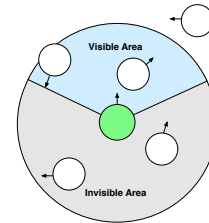


Figure 4: The visual field of a boid

When two boids are very close to each other, up to collide, corresponds to a special case where a boid takes into account a neighbor even if it is outside of the its field of view. If collisions were allowed to happen, the simulation could become unstable since neighbor boids coming from behind would suddenly appear in front of another. It is possible to think of this as a collision detection for a prevention system, having the same effect as a movement made by animals that, even not seeing each other, would have gotten into a sudden contact.

4.2 Flocking Behavior

A boid keeps on moving by watching his visible neighbors and deciding what direction to take next. Each neighbor influences this direction in different conflicting manners, depending on its type and distance from the simulated boid. From neighbors of the same type, the simulated one receives three simultaneous influences: grouping, repulsion and direction following.

4.2.1 Grouping Influence

By grouping we mean the tendency that animals from the same species have to keep forming relatively tight groups. To simulate this behavior we compute the group center position by averaging the positions of all visible neighbors of the same type as the one being simulated. This grouping influence will be multiplied by a grouping factor, unique for each type, and by the distance from the centre. The last factor will make the influence stronger to boids that are far from the group. Figure 5 illustrates grouping and repulsion influences.

4.2.2 Repulsion Influence

If only the grouping influence was taken into account, boids would tend to form very dense groups, resulting in very frequent collisions, not representing what we see in nature. To balance this grouping tendency a collision avoidance influence is computed. For each simulated boid, the relative distance to its neighbors is computed and divided by its length. This weighted vector is then multiplied by a specified repulsion factor and added as an influence to the desired motion vector. One can notice that the parameterized factors of both the grouping and distance influences play a major role in determining the density of the groups, since one cancels each other at a certain distance when equilibrium is reached between them.

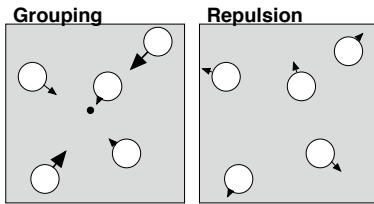


Figure 5: Grouping and repulsion influences

4.2.3 Direction Following Influence

Besides the tendency of forming groups, animals also tend to follow the same direction as its companions. To achieve this behavior we compute another influence every time a boid sees a neighbor of the same type. This influence is represented by the current velocity/direction followed by the neighbor. Figure 6 exemplifies this influence.

4.3 Leader Following

Besides from recognizing its neighbors of the same type and trying to move as a group, each type may have a leader to follow. Normal boids, when see the leader, have a stronger desire to follow it, represented by a larger multiplier factor, that gets blended with the other computed influences. Each leader is simulated at the same time as normal boids but also being identified as such and acting accordingly. However, the movement of this leader is not driven by the desire to keep grouping, but only trying to reach a desired location and avoiding obstacles and other boid groups.

Inside the data structures, the leaders are represented as normal boids. There is a small auxiliary array keeping the current matrix index exclusively for the leaders of each boid type. The array size is the number of different boid types. Element n of this array contains the cell index of the leader for the boids of type t . To be correct along the time, this array must be updated by the sorting pass if any of the leaders change its cell.

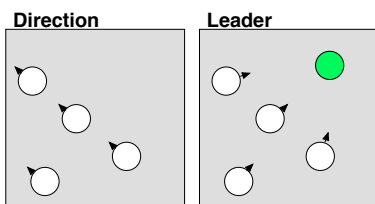


Figure 6: Direction and leader following

During the simulation step, for each boid the leaders array value for its type is fetched and the value returned identifies the leader index inside the matrixes. If the returned index corresponds to the boid being simulated, it means that it corresponds to the group leader and follows to an alternative and more random simulation algorithm.

For normal boids, this leader index is used to fetch its position and direction, so that the correct influence can be computed.

4.4 Obstacles, Enemies and Influences Composition

In this work, obstacles are also represented as boids inserted in the same data structures, also being sorted and simulated. To avoid movement during the simulation step, obstacles are initialized with a different type value, and are not simulated. However, if a neighbor of a specific simulated boid happens to be an obstacle, the only influence calculated is a repulsion force. This force is then multiplied by a factor that is stored in the unused direction vector of this still obstacle-boid, enabling the representation of obstacles of arbitrary sizes with a round repulsion field. Neighbors of different types that are not obstacles also have a strong repulsion influence calculated, but the multiplier factor is kept at the simulated boid type, representing an enemy-fearness factor. All calculated influences are added into an acceleration vector that is used to update the position and direction/speed vectors.

5 Performance and Analysis

In this work, we evaluated two versions of the described simulations. While the first was completely executed at the GPU, the second was built for the CPU. Both versions used the same partial sorting strategy, based on a single pass of the odd-even transposition sort algorithm. The tests were performed on an Intel Core 2 Quad 2.4GHz CPU, 4GB of RAM equipped with an NVidia 8800 GTS GPU. Each instance of the test ran for 300 seconds. The average time to compute a frame was recorded for each one. To assure the results are consistent, each test was repeated 10 times.

A total of 8 different test instances were executed for each implementation type varying only the number of boids, ranging from 64 up to 1,048,576. At preliminary tests, we observed that the number of boid types had little influence on the performance, so a fixed number of 8 types was used. Figure 7 brings a summary of the results showing how both implementations scale with the increasing number of simulated boids. As expected, the CPU version presents an early quadratical behavior, with more than one second to calculate and render each frame when more than 250,000 boids were present, which is another evidence of the better scalability of our GPU implementation.

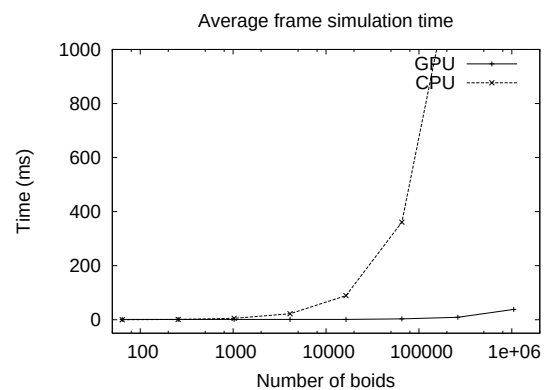


Figure 7: CPU vs. GPU implementations

From the raw results, shown in Table 1, it is possible to see that the performance of the GPU implementation bypassed the CPU at around 250 boids and sustained interactive frame rates for more than 250,000 boids. Impressive 25fps are achieved for a population of more than 1 million boids. In our performance experiments, each boid was rendered as a single point on the screen, which enabled us to measure the cost of the simulation steps in CUDA, instead since the rendering of complex models and other background elements

Table 1: Raw results

# of boids	time GPU	fps GPU	time CPU	fps CPU
64	1,06	946,13	0,30	3307,90
256	1,08	929,89	1,38	689,85
1024	1,11	904,07	5,56	174,40
4096	1,22	822,57	22,53	43,84
16384	1,66	594,86	89,89	10,92
65536	3,50	280,02	361,27	2,67
262144	9,67	101,96	1390,29	0,66
1048576	38,96	25,45	5394,65	0,17

would make this task the performance bottleneck. Figure 8 shows a screenshot of a running simulation with 65536 flocking boids at an average framerate of 280fps.

**Figure 8:** Simulation with 64K boids

6 Conclusion

In this paper we showed a novel technique for simulating emergent behavior of spatial dynamic entities called Supermassive Crowd Simulation. We showed an implementation capable of running up to 1,048,576 autonomous flocking boids at an interactive frame rate using current graphics hardware and CUDA technology. The data structures are suitable for several different simulation algorithms as long as they can be modeled as cellular automata.

As future work we plan to extend this model to include the representation of more complex geometric obstacles such as buildings or mazes. These augmented data structures and more complex algorithms are being developed in order to have more complex 3D boids representation and consequently more realistic simulations. The project is being built as a crowd simulation library where users can just plug in sorting and simulation strategies. Today our strategies do not take into account time required for rendering complex geometry. However, simple experiments are showing that more polygons for each boid will not compromise to much the achieved performance.

We also plan to further analyse the complexity of the data structures. We are specially interested in the distribution of perturbations of the sorting rules during simulation when applying partial sorting strategies such as the odd-even transposition sort pass used in the presented example. This analysis will not be limited to the neighborhood matrix presented but extended to tridimensional arrays.

References

- CHIARA, R. D., ERRA, U., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *VMV*, 233–240.
- CORDEIRO, O. C., BRAUN, A., SILVEIRA, C. B., AND MUSSE, S. R. 2005. Concurrency on social forces simulation model. In *Proceedings of the First International Workshop on Crowd Simulation*.
- COURTY, N., AND MUSSE, S. R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, IEEE Computer Society, Washington, DC, USA, 206–212.
- FUNGE, J., TU, X., AND TERZOPOULOS, D. 1999. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Siggraph 1999, Computer Graphics Proceedings*, Addison Wesley Longman, Los Angeles, A. Rockwood, Ed., 29–38.
- MUSSE, S. R., AND THALMANN, D. 1997. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In *Workshop Computer Animation and Simulation of Eurographics*, 39–52.
- NVIDIA, 2007. Cuda technology. <http://www.nvidia.com/cuda>.
- PELECHANO, N., ALLBECK, J. M., AND BADLER, N. I. 2007. Controlling individual agents in high-density crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 99–108.
- QUINN, M. J., METOYER, R. A., AND HUNTER-ZAWORSKI, K., 2003. Interaction with groups of autonomous characters.
- 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*, ACM, New York, NY, USA, 25–34.
- REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Game Developers Conference*.
- REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Game Developers Conference*.
- REYNOLDS, C. 2006. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM, New York, NY, USA, 113–121.
- SARKAR, P. 2000. A brief history of cellular automata. *ACM Comput. Surv.* 32, 1, 80–107.
- SHAO, W., AND TERZOPOULOS, D. 2005. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 19–28.
- STEED, A., AND ABOU-HAIDAR, R. 2003. Partitioning crowded virtual environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, 7–14.
- TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, 1160–1168.