A Neighborhood Grid Data Structure for Massive 3D Crowd Simulation on GPU

Mark Joselli
UFF, Medialab

Erick Baptista Passos
UFF, Medialab

Marcelo Zamith
UFF, Medialab

Esteban Walter Gonzalez Clua
UFF, Medialab

Anselmo Montenegro
UFF, Medialab

Bruno Feijo
PUC-RIO, ICAD Games

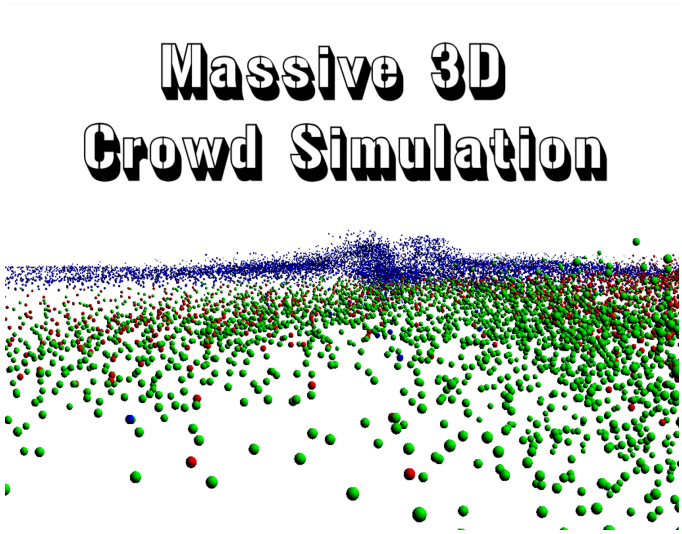

Figure 1: *A screenshot of the simulation.*

Abstract

Simulation and visualization of emergent crowd in real-time is a computationally intensive task. This intensity mostly comes from the $O(n^2)$ complexity of the traversal algorithm, necessary for the proximity queries of all pair of entities in order to compute the relevant mutual interactions. Previous works reduced this complexity by considerably factors, using adequate data structures for spatial subdivision and parallel computing on modern graphic hardware, achieving interactive frame rates in real-time simulations. However, the performance of existent proposals are heavily affected by the maximum density of the spatial subdivision cells, which is usually high, yet leading to algorithms that are not optimal. In this paper we extend previous neighborhood data structure, which is called neighborhood grid, and a simulation architecture that provides for extremely low parallel complexity. Also, we implement a representative flocking boids case-study from which we run benchmarks with simulation and rendering of up to 1 million boids at interactive frame-rates. We remark that this work can achive a minimum speeup of 2.94 when compared to traditional spatial subdivision methods with a similar visual experience and with lesser use of memory.

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

Author's Contact:

{mjoselli,epassos,mzamith,esteban,anselmo}@ic.uff.br
bruno@inf.puc-rio.br

1 Introduction

In a typical natural environment it is common to find a huge number of animals, plants and small dynamic particles. This is also the case in other densely populated systems, such as sport arenas, communities of ants, bees and other insects, or even streams of blood cells in our circulatory system. Computer simulations of these systems usually present a very limited number of independent entities, mostly with very predictable behavior. There are several approaches that aim to include more realistic behavioral models for crowd simulation such as [Reynolds 1987; Musse and Thalmann 1997; Shao and Terzopoulos 2005; Pelechano et al. 2007; Treuille et al. 2006].

Algorithms for massive crowd simulation are driven by the need to avoid the $O(n^2)$ complexity of the proximity queries between 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. As far as we know, no work until the present date has proposed a real time simulation of more than just a few thousands of complex entities interacting with each other. Applications for these computationally demanding algorithms range over crowd behavior prediction in emergency scenarios, street traffic simulation and enrichment of computer game worlds.

Non-graphics algorithms traditionally executed on the CPU, such as behavioral artificial intelligence algorithms, are sometimes suitable for parallel execution, which makes them appropriate to be implemented on the GPU. However, the first applications of GPUs performing general purpose computation (GPGPU) had to rely on the adaptation of graphics rendering APIs to different concepts, leading to a difficult learning curve and sometimes not very efficient data structures for the proposed solutions. The CUDA [NVidia 2009], CAL [AMD 2007] and OpenCL [Group 2009] technologies aim to provide a new abstraction layer on top of graphics hardware to facilitate its usage for non-graphics processing. Crowd simulation that explores this programming model on the GPU is a promising line of research.

Most of the research on crowd simulation tries to avoid the high complexity of proximity queries by applying some form of spatial subdivision to the environment and classifying entities among the cells based on their position. To accelerate data fetching in a parallel hardware (such as GPUs) the entities list must be sorted in such a way that all entities on the same cells are grouped together. This approach helps lowering the number of proximity queries but is very sensible to the maximum number of entities that can fit in a single cell. In this paper instead of using a similar approach,

we propose a novel simulation architecture that maintains entities into another kind of proximity based data structure, which we call "neighborhood grid". In this data structure, each cell now fits only one entity and does not directly represent a discrete spatial subdivision. The "neighborhood grid" is an approximate representation of the system of neighborhoods of the environment that maps the N-dimensional environment to a discrete map (lattice) with N dimensions, so that entities that are close in a neighborhood sense, appear close to each other in the map. Another approach is to think of it as a multi-dimensional compression of the environment that still keeps the original position information of all entities.

The entities are simulated and sorted as Cellular Automata with Extended Moore Neighborhood [Sarkar 2000] over the neighborhood grid, which is an ideal case for the memory model of GPUs. We argue and show that this approximate simulation technique brings a new bound to crowd simulation performance, maintainning the believability for entertainment contexts. The high performance and scalability are achieved by a very low parallel complexity of the model.

To keep the "neighborhood grid" aligned this work shows a previous implementation of a partial sorting mechanism, a partial odd-even sort, and a new sorting scheme, a bitonic sort, that can keep a much better visual experience with similar performance.

To illustrate and evaluate the "neighborhood grid", we implement a traditional emergent behavior model of flocking boids [Reynolds 1987] that has a minimum speedup of 2.94 over the tradition spatial hashing methods [Reynolds 2000; Reynolds 1999], with similar visual experience. The architecture can be further extended to any other simulation model that rely on dynamic autonomous entities and neighborhood information.

Summaryzing, this work is an extension of the work [Passos et al. 2008], with the following enhancements, which are the main contributions of these paper:

- Extension of the data structure for 3D environments;
- Presentation of a new sorting scheme that keeps a better visual experience with similar performance;
- Comparison of performance between our method and the tradition spatial hashing method [Reynolds 2000; Reynolds 1999] which was also implemented by this work.

The paper is organized as follows: Section 2 discusses related work on crowd simulation. Sections 3 explain the proposed "neighborhood grid", the data structures, the simulation steps in 3D and a simplification of the "neighborhood grid" for 2D systems. Section 4 describes the particular behavior model used to validate the proposed architecture. Section 5 brings the experimental results and analysis of the implemented simulation model. Finally, section 6 concludes the paper with a discussion on future work.

2 Related Work

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 accordantly 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. This was such an issue at the time 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 generic simulated flocking creatures ever since.

Musse and Thalmman [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 latter 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. The data structures and algorithms used by these works are not suitable for real-time simulation of very large crowds, which is one of the goals of this work.

Reynolds further enhanced 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 60 fps in a Playstation 2 hardware. By using the spatial hash to classify the boids into 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 leads to a sequential complexity that is closer to $O(n)$. This complexity, 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 neighborhood grid 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 up to 10,000 individuals at 45 fps 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 15,000 boids in Playstation 3 hardware at interactive framerates, but with a reduced simulation frame rate of around 10 fps. Due to the distributed memory of both architectures, it is necessary to copy compact versions of the buckets/cells of boids to the individual parallel processors before the simulation step could run, copying them back at the end of it to perform the rendering, which leads 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 in massive crowd simulation is very promising but brings another issue, related to its intrinsic dependency on data-locality to achieve high performance in this kind of hardware. For agent-based simulations that rely on spatial hashing, it is desired that the individuals should be sorted through the 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 task is triggered only when a boid departs from its group, which is detected by the use of a scattering matrix. This system could simulate 1,600 boids at 60 fps including the rendering of animated 2D models. Also the work by Silva et al. [Silva et al. 2008] implement a similar work, but it focus on the optimization of the algorithm by doing occlusion based on the vision of the boids. The FastCrowd system [Courty and Musse 2005] was also implemented with a mix of CPU and GPU computation to simulate and render a crowd of 10,000 individuals at 20 fps as simple 2D discs. Using this simple rendering primitive, the GPU was also capable of simultaneously computing the flow of gases on an evacuation scenario. A more recent work in the GPGPU field by Shopf et. al. [Shopf et al. 2008] presents an implementation that runs entirely on the GPU and can simulate and render 3,000 high detailed animated models or 65,000 simple primitives at real-time frame rates. Our implementation also runs entirely on the GPU and makes use of the fact that groups tend to move as blocks and uses a parallel sorting algorithm on the GPU to achieve even higher performance, as explained in the next sections.

The simulation architecture and data-structures proposed by [Treuille et al. 2006] depart from the agent-based models presented so far. These authors 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. Treuille et al argue that locally controlled agents, while providing for complex emergent behavior, are not an appropriate model for goal-driven individuals, such as human pedestrians. The im-

plemented system could simulate up to 10,000 humans at 5 fps (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 predetermined size.

In the work [Passos et al. 2008], which this work extends, is implemented a crowd simulation system on the GPU where each boid is modeled as a cellular automaton [Sarkar 2000] in a 2D data structure. This work could achieve the simulation and renderization of up to 1 millions boids at interactive frame rates. The present paper extends that previous work to 3D environments keeping the same performance. As far as we are aware, no other work in the literature presents such a high performance.

This cellular automata model matches perfectly with the ideal locality for data fetching on graphics hardware but imposes that boids information have to be kept reasonably sorted over this data structure during simulation. Our proposal, such as most of the above work, is based on distributed agents to yield emergent behaviour, but the novel data-structures are suitable for unlimited environment size and better scalability over both the number of entities and neighbourhood reach.

3 Simulation Architecture

Individual entities in crowd behavior simulations depend on observations of their surrounding neighbors to decide which actions to take. The straightforward implementation of the neighborhood finding algorithm has a complexity of $O(n^2)$, for n entities, since it performs at least one proximity query for each entity pair in the crowd. Individuals are autonomous and can move during each frame, which leads to a very computationally intesive task.

Techniques of spatial subdivision have been used to group and sort these entities in order to accelerate the neighborhood finding task. Current implementations are usually based on variations of relatively coarse subdivisions techniques, such as a grid over the considered environment. After each update, all entities have their grid cell index calculated based on their latest locations. For GPU based solutions, some kind of sorting based on this index has to be performed in order to benefit from the read-ahead and caching mechanisms of such hardware. This way, neighbor entities in geometric space are stored near each other over the data structure. However, static subdivisions have some limitations when simulating large geometric spaces, where the size of each grid cell may fit a large number of entities. This issue limits the neighborhood finding problem by a hidden $O(n^2)$ complexity factor in the worst case scenario.

In this work we propose another approach for the neighborhood finding problem. This approach uses a grid data structure, which we call "neighborhood grid" that is used to store information about all the entities. In this "neighborhood grid", each entity is mapped in a individual cell (1:1 mapping) accordingly to its spatial location, so that entities that are close in a neighborhood sense, appear close to each other in the grid. In order to keep the "neighborhood grid" mapped accordingly to the spatial location, a sorting mechanism is needed. To fulfill that need, we present two sorting mechanism, one partial odd-even sort and one bitonic sort.

This simulation architecture can be described as a continuous loop with the following steps:

- Sorting pass (re-organizes the neighborhood grid);
- Simulation pass (updates position and orientation);
- Rendering pass (draws visible entities).

The following subsections describe the architecture. In the next subsection the "neighborhood grid" is explained. The role of sorting and the types of simulation algorithms suitable to the proposed architecture are also explained in following subsections. Also in the last part of this section we show a simplified version of the data structures for 2D simulations.

3.1 3D Proximity Data Structure: The Neighborhood Grid

The proposed architecture was developed with CUDA technology [NVidia 2009], and, in order to keep the processing entirely at the GPU, all information about entities is mapped as textures for the display-list and vertex shader rendering. The minimum information required for each entity are: position (a vector, representing the position of the entity), speed (a vector for storing the orientation and velocity in a single structure) and type (an integer that can be used to differentiate entity classes).

This information is stored in 3D arrays (grid), where each position holds the entire data for an individual entity. In this case two grids are required, one for the 3D position and another for the orientation with the entity type variable kept at a fourth value in one of these grids. The grid that contains the position vector for the entities is then used as a sorting structure. In this data structure, each cell fits only one entity. Figure 2 illustrates how a randomly distributed set of entities would be arranged in the "neighborhood grid" when correctly sorted. The smaller circles represent entities that are further away from the viewpoint.

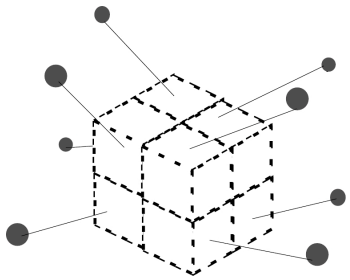

Figure 2: *An example of a distribution of entities in the neighborhood grid. Entities that are further away from the viewpoint are illustrated by small circles.*

In this work we use a form of neighborhood gathering that is known as Extended Moore Neighborhood [Sarkar 2000] in the Cellular Automata theory. Figure 3 illustrates this structure with a 2D matrix holding arbitrary information for 36 individual entities. To reduce the cost of proximity queries, each entity will only gather information about the entities surrounding its cell, based on a constant radius. In the example of Figure 3, this radius is 2, so the entity represented at cell (2,2) (in gray) would have access to the 24 highlighted surrounding cells/entities (in green) only.

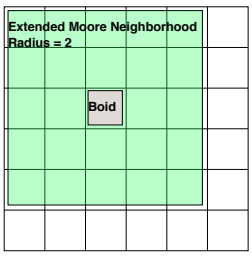

Figure 3: *Example of the Structure of the Extended Moore Neighborhood with 36 entities and radius = 2.*

Our work extends this matrix example to a 3D grid maintaining the same form of information gathering, only adding the extra dimension. Figure 4 illustrates our neighborhood grid with a neighborhood radius of 1.

This kind of spatial data structure and extremely regular information gathering enables a good prediction of the performance, since the number of proximity queries will always be constant over the simulation. This happens because instead of making these proximity queries over all entities inside a coarse grid bucket/cell (variable quantity), such as in traditional implementations, each entity would query only a fixed number of surrounding individual neighbors. However, this matrix has to be sorted continually in such a way that

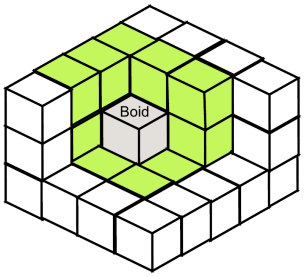

Figure 4: *Example of the neighborhood grid with radius = 1.*

those entities which are neighbors in geometric space are stored in individual cells that are close to each other. This guarantees that each entity should gather information about its closest neighbors. Depending on the simulation (and the sorting step), some misalignment may occur over the data structure, causing that some of the neighbor entities are missed by the gathering step. However, the larger the Moore radius is, less likely it is to happen such issue, which we could observe during the experiments.

3.2 Sorting Pass

The position information of each entity is used to perform a lexicographical sort based on the three dimensions of this vector. The goal is to store in the closer-bottom-leftmost cell of the grid the entity with the smaller values for Z, Y and X, and in the far-top-rightmost cell the entity with highest values of Z, Y and X respectively. Using these three values to sort the matrix, the farthest lines will be filled with the entities with the higher values of Z while the top lines will be filled with the entities with higher values of Y and the right columns will store those with higher values for X and so on. This kind of sorting provides for the approximate neighborhood query that is optimal in terms of data locality.

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

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

Extending this rule to a grid **G** where each cell has three float point values *X*, *Y* and *Z*:

1. $\forall$*G[i][j][k]* $\in \mathbf{G},\ k > 0,$ *G[i][j][k].Z* $\leq$ *G[i][j][k-1].Z*;
2. $\forall$*G[i][j][k]* $\in \mathbf{G},\ k > 0, G[i][j][k-1].Z = G[i][j][k].Z \Rightarrow$ *G[i][j][k].X* $\leq$ *G[i][j][k-1].X*;
3. $\forall$*G[i][j][k]* $\in \mathbf{G},\ k > 0, G[i][j][k-1].Z = G[i][j][k].Z\ AND\ G[i][j][k].X \leq G[i][j][k-1].X \Rightarrow$ *G[i][j][k].Y* $\leq$ *G[i][j][k-1].Y*;
4. $\forall$*G[i][j][k]* $\in \mathbf{G},\ j > 0,$ *G[i][j][k].Y* $\leq$ *G[i][j-1][k].Y*;;
5. $\forall$*G[i][j][k]* $\in \mathbf{G},\ j > 0, G[i][j-1][k].Y = G[i][j][k].Y \Rightarrow$ *G[i][j][k].Z* $\leq$ *G[i][j-1][k].Z*;;
6. $\forall$*G[i][j][k]* $\in \mathbf{G},\ j > 0, G[i][j-1][k].Y = G[i][j][k].Y\ AND\ G[i][j-1][k].Z \leq G[i][j][k].Z \Rightarrow$ *G[i][j][k].X* $\leq$ *G[i][j][k-1].X*;
7. $\forall$*G[i][j][k]* $\in \mathbf{G},\ i > 0,$ *G[i][j][k].X* $\leq$ *G[i-1][j][k].X*;
8. $\forall$*G[i][j][k]* $\in \mathbf{G},\ i > 0, G[i-1][j][k].X = G[i][j][k].X \Rightarrow$ *G[i][j][k].Y* $\leq$ *G[i-1][j][k].Y*;
9. $\forall$*G[i][j][k]* $\in \mathbf{G},\ i > 0, G[i-1][j][k].X = G[i][j][k].X\ AND\ G[i][j][k].Y \leq G[i-1][j][k].Y \Rightarrow$ *G[i][j][k].Z* $\leq$ *G[i-1][j][k].Z*;

The architecture is independent of the sorting algorithm used, as long as the rules above are always, eventually or even partially achieved during simulation, depending on the desired neighborhood precision. In this work we show a partial odd-even sort, which makes a partial sort in each dimension and a bitonic sort [Batcher 1968], which make a full sort in each dimension.

3.2.1 Partial Odd-Even Sorting

Here we present an inherently parallel (but not optimal) partial sort strategy: an odd-even transposition sort, with only one odd-even pass per update. The odd-even transposition sort is similar to the bubble sort algorithm and it is possible to complete a partial pass, traversing the whole data structure, in $O(n)$ sequential time or $O(1)$ parallel complexity when running on n CUDA threads (if available on the GPU). Because there are two steps, one for odd and other for even elements (for each axis), this algorithm is suitable for parallel execution.

This sorting pass must be spread into six steps, one for odd and one for even elements for each axis. The first step runs the sorting between each entity position vector of the even columns against its immediate neighbor in the subsequent odd column. If the rules described by Eq.1, Eq. 2 or Eq.3 are violated, the entities switch cells in the grids. The other six sorting steps perform the same operation for the odd column of the Z and the similar steps over the Y and X axis.

From tests we have seen that with this partial sort more than 10% of entities are in the wrong place on the "neighborhood grid" when comparing with a full sort on the entire grid. So this sorting mechanism only seems viable on simulation that does not need a lot of precision, or that the entities does not change position very often. Otherwise the use of the bitonic sort is advised, which is present next.

3.2.2 Bitonic Sorting

The bitonic sort [Batcher 1968] is simple parallel sorting algorithm that is very efficient when sorting small number of elements [Blelloch et al. 1998], which is our case since our sort strategy is divided by dimensions. Our implementation is an optimized and adapted version based on a demo from nVidia [nVidia 2008]. This sort is divided in 3 passes, one for each dimension (X,Y and Z).

The complexity of this algorithm is $O(nlog(n)^2)$ being n the number of elements to sort in sequential time. This comparisons are divided in n CUDA threads making the algorithm in this parallel implementation with a complexity of $O(log(n)^2)$, if there is n stream processors on the GPU.

This sorting does not make a full sort on the "neighborhood grid" only a full sort on each dimension (X,Y and Z) of the grid. So, for example, if a change in one entity position on the Y pass, another pass for the X would be needed in order to keep the "neighborhood grid" with an full sort. But from tests we have seen that this misaligned is very small, less than 1% of the entities changes place in one step of the simulation, and in the next step this error will be fixed, and the use of a full sort on the "neighborhood grid" would impose some lost in performance without visible gain in the simulation.

3.3 Simulation Pass

The simulation pass can perform any kind of emergent crowd behavior for entities that are constrained to the knowledge of data in their surrounds, such as flocking boids, swarms or pedestrian groups. This pass must be implemented as a CUDA kernel function that receives as arguments at least the position and orientation of each entity (double buffered as input and output) and the time elapsed since the last step. This kernel function is then executed in parallel with one CUDA thread for each entity. This function uses the data from the previous step for the respective entity and its neighbors and calculates new values for its entity only, which must be written to the same cell in the output grid.

In Section 4, an example of a flocking boids simulation pass is described. The implementation of such simulation in our architecture is evaluated in Section 5. The following subsection is dedicated to explain the 2D version of the presented data structures.

3.4 Simulation Architecture with 2D Proximity: The Neighborhood Matrix

The 2D proximity is a simplification of the 3D proximity with only the X and Y (or Z) dimension. In this case, the proximity data structure used is a "neighborhood matrix" instead of the 3D "neighborhood grid", but with similar extended Moore neighborhood as showed in figure 3. The sorting pass is a simplified one with just the X and Y passes.

We remark that the term 2D mentioned refers only to the spatial nature of the data structure, which is still suitable for a simple 3D simulation where the entities do not traverse the third dimension too much such as a pedestrian crowd. For more complex simulations where entities move freely over the third axis, such as swarm of bees, we recommend using the 3D version of the proximity data structure.

4 Case-Study: Flocking Boids

For the purpose of validating the proposed technique, we choose to implement 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 "neighborhood grid" enables real time simulation with up to one million animals of several types, with a corresponding visual feedback as shown in the experiments described the next section.

Our model simulates a crowd of animals interacting with each other and avoiding random obstacles around the continuous 3D space. This simulation can be used to represent from small bird flocks to huge and complex terrestrial animal groups. Boids from the same type (representing species) try to form groups and avoid staying close to the other types. The number of simulated entities/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 on 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 cellular automata, where a combination of internal and external factors (from neighbor cells) defines which actions are taken and how they are done. With this approach, internal state is represented by position, speed (also orientation) and the boid type, and external information reffers to visible neighbors, depending on where the boid is looking at (orientation), and their relative distances.

Our simulation algorithm computes these influences for each boid: flocking (grouping, repulsion, and direction following); leader following; and repulsion from other types of boids (that can be used also for obstacle avoidance). Additionally, there are constant multiplier factors which dictate how each influence type may get blended with another. In order to enable a richer simulation, these factors are stored independently for each type of boid in separate arrays. More information about the behavior used in this work refer to [Passos et al. 2008].

5 Performance and Analysis

In this work, we implemented and tested the flocking boids case-study using the "neighborhood grid" and also evaluated the rendering of all boids. The rendering consists of a simple display list that is repeated for each entity/boid using the position and orientation information gathered from a texture that is bound from the output VBO of the CUDA simulation in a vertex shader as can be seen on Figure 5.

All tests in this work were performed on an Intel Core 2 Quad 2.4GHz CPU with 3GB of RAM and equipped with an NVidia 8800 GTS GPU (that has 96 stream processors) and the operating system is Windows Vista. Each instance of the test ran for 300 seconds. The average time to compute a frame (and subsequent frames per second) was recorded for each experiment. To assure the results are consistent, each test was repeated 10 times and the standard deviation of the average times confirmed to be within 3%. All tests results includes the simulation calculation and also the renderization to the screen.

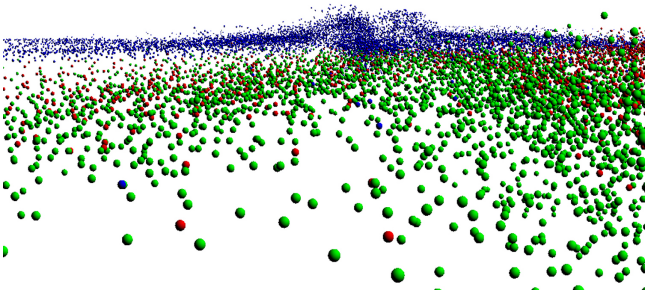

Figure 5: *Simulation with 32K boids.*

To evaluate the scalability of the architecture, we varied the number of entities/boids being simulated (from 1 thousand to 1 million) and the Moore neighborhood radius (from 1 to 4). At preliminary tests, we observed that the number of different boid types had no observable influence on the performance, so a fixed number of 4 types was used for all experiments. In order to fully evaluate the speedup of this architecture for crowd simulation over the traditional spatial hashing method based on [Reynolds 2000; Reynolds 1999], we have implemented the spatial hasing scheme in GPU with the use of CUDA. This implementation has the same flocking behavior as the one implemented in our architecure.

Table 1 shows the results of the simulation in frames per second, for all experiments in 3D with the partial odd-even sort compared with the traditional spatial hashing. We can notice that the simulation runs at interactive frame-rates even with 1 million boids. With the use of the partial odd-even sort, we see the best visual performance with radius 4, but we can still see some strange behavior some times, i.e, collision and behaviors that should not happen. With this radius we have a minimum speedup of 1.86 when compared with the traditional spatial hash method.

Table 2 shows the results of the simulation in frames per second, for all experiments in 3D with the bitonic sort compared with the traditional spatial hashing. With the use of the bitonic sort, we see the best visual performance with radius 2. With this radius we have a minimum speedup of 2.94 when compared with the traditional spatial hash method.

From this results we can see that the bitonic sort is faster than the partial odd-even sort when there are less than 32,768 entities. This happens mainly because the bitonic sort does 1 pass for each dimension while the partial odd-even sort does 2 passes for each dimension. Also using the best radius for visual experience (radius 2 for the bitonic sort and radius 4 for the partial odd-even sort), we can see that the bitonic sort have minimum speedup of 1.16 over the partial odd-even sort. We suggest that for the best visual and performance crowd simulation, to use the presented architecture with bitonic sort and the radius 2.

Table 3 shows how much memory for the presented architecture, which is the same for both sorting mechanisms and different neighborhood radius, and for the spatial hash. From this results we can see that this architecture spends much less memory since it does not needs a lot of memory to keep the data structure having consuming memory in a linear form, while the spatial hash does needs at least 2 MB for keeping the data structure.

Figure 6 shows how the time are spent in % with each step of the simulation (with the data structure, the behavior and the memory copy) during the simulation with 32,769 boids for the bitonic sort (with radius 2), odd-even sort (with radius 4) and spatial hash. This shows that the spatial hash uses 35 % of its time processing the its data structure while the bitonic sort spends 25% and the odd-even

Table 1: *Numerical results of the architecture running with a partial odd-even sort compared with the spatial hash.*

# Boids	Spatial Hash Fps	Partial Odd-Even Sort							
		Radius=1		Radius=2		Radius=3		Radius=4	
		FPS	Speedup	FPS	Speedup	FPS	Speedup	FPS	Speedup
1,024	370	860	2.35	710	1.92	695	1.87	688	1.86
32,768	72	222	3.08	200	2.78	185	2.57	166	2.30
131,072	18	68	3.78	63	3.50	57	3.17	51	2.83
524,288	4.00	19	4.75	17	4.25	15	3.75	12	3.00
1,048,576	0.50	9.72	19.55	8.60	17.20	7.41	14.84	6.25	12.50

Table 2: *Numerical results of the architecture running with a bitonic sort compared with the spatial Hash.*

# Boids	Spatial Hash Fps	Bitonic Sort							
		Radius=1		Radius=2		Radius=3		Radius=4	
		FPS	Speedup	FPS	Speedup	FPS	Speedup	FPS	Speedup
1,024	370	1,155	3.12	1,118	3.02	1,109	3.00	1,099	2.97
32,768	72	212	2.94	197	2.74	178	2.47	164	2.28
131,072	18	62	4.50	58	3.22	53	2.94	48	2.67
524,288	4.00	18	4.50	16	4.00	14	3.50	12	3.00
1,048,576	0.50	8.45	16.90	7.49	14.98	6.64	13.28	6.20	12.40

Table 3: *Use of the memory when using the Spatial Hash and this architecture.*

# Boids	Use of Memory	
	Spatial Hash	Neighborhood Grid
1,024	2.1 MB	5.6 KB
32,768	2.3MB	180 KB
131,072	3 MB	721 KB
524,288	5.5 MB	2.9 MB
1,048,576	9 MB	5.8 MB

only 14%.

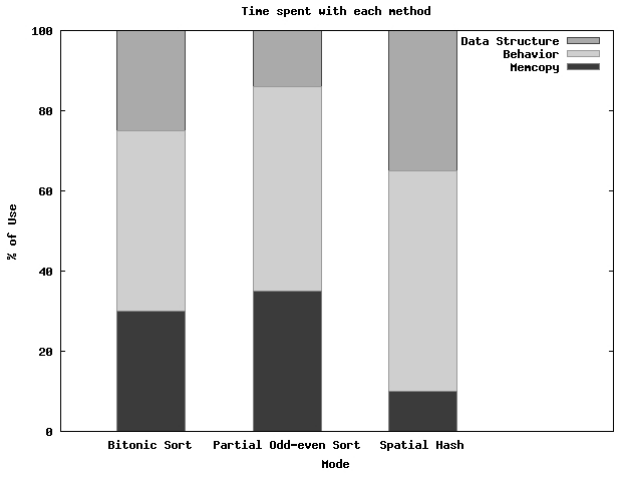

Figure 6: *Comparison of the % of use between the Bitonic Sort, Odd-even partial sort and Spatial Hash.*

Figure 7 shows comparison between the spatial hashing and the two sorting schemes, showing how the time to compute and render each frame grows with the number of boids using the the same radius 2 for the bitonic sort and radius 4 for the partial odd-even sort. These plot uses a logarithmic scale in both axis (due to the growth on the number of boids in the experiments) which shows that there is a linear (and not quadratic) relation between the number of boids and the computing cost, for our architecture.

Also we have tested the architecture with the simplified data structure for 2D simulations (the "neighborhood matrix"). The results show gains from 10 to 20 % in speedup when compared with the 3D simulation. The reason for this performance difference is not related to the simulation itself, but to the fact that in 3D the number of candidate neighbors is higher, leading to more memory reads for each boid. For instance, with a radius of 1 each boid agent in 2D reads its 8 immediate neighbors, while in 3D this number (with the same radius) grows to 26. The architecture is also implemented in the CPU so it can be used in computers that does not have GPU with CUDA. In this case it can only simulate and render up to 8,000 boids in real time.

6 Conclusion

In this paper we have shown an extension of a novel technique for simulating emergent behavior of dynamic entities in a densely populated environment. We have extended all of our data structure to higher dimension (3D) in order to deal with 3D scenes. We also have implemented and compared two sorting techniques to be used with the architecture, one partial odd-even sort and one bitonic sort. We have seen, from visual experience and numerical test, that for simulating flocking boids the partial odd-even sort is not the best approach when precision is needed, since it keeps a high error in the "neighborhood matrix" of more than 10 % on the grid and this error can be perceved visualy in the simulation by the boids' behavior and collisions. With the best radius for visual experience, radius 2 for the bitonic sort and radius 4 for the partial odd-even sort, we have a speedup of 1.16 with the use of the bitonic sort over the partial odd-even sort.

This architecture is capable of interactively simulating and rendering up to 1 million of individual flocking boids in real time, while the traditional spatial hashing methods expends 2 seconds for executing each frame. And with the use of our architecture with bitonic sort and a radius 2, we experience a similar visual simulation as with the spatial hashing method with expressive speedup. The authors of this work suggest using this configuration, the presented architecture with bitonic sort and the radius 2, to achieve best visual and performance crowd simulation.

The data structures developed for 3D and 2D approximate neighborhood queries lead to very low parallel complexity and are suitable for several different simulation algorithms, as long as they can be modeled as cellular automata with Extended Moore Neighborhood.

As future work we plan to extend this architecture to enable the representation of more complex geometric obstacles such as buildings, terrains or mazes. These augmented data structures and more complex algorithms are being designed in order to achieve more realistic simulations, and consequently providing for a even more believable virtual environment. This project is being developed as a crowd simulation framework where programmers can plug in their chosen sorting and simulation strategies.

While our performance evaluations do not render a complex geometry for each entity, we argue that the performance penalty for adding

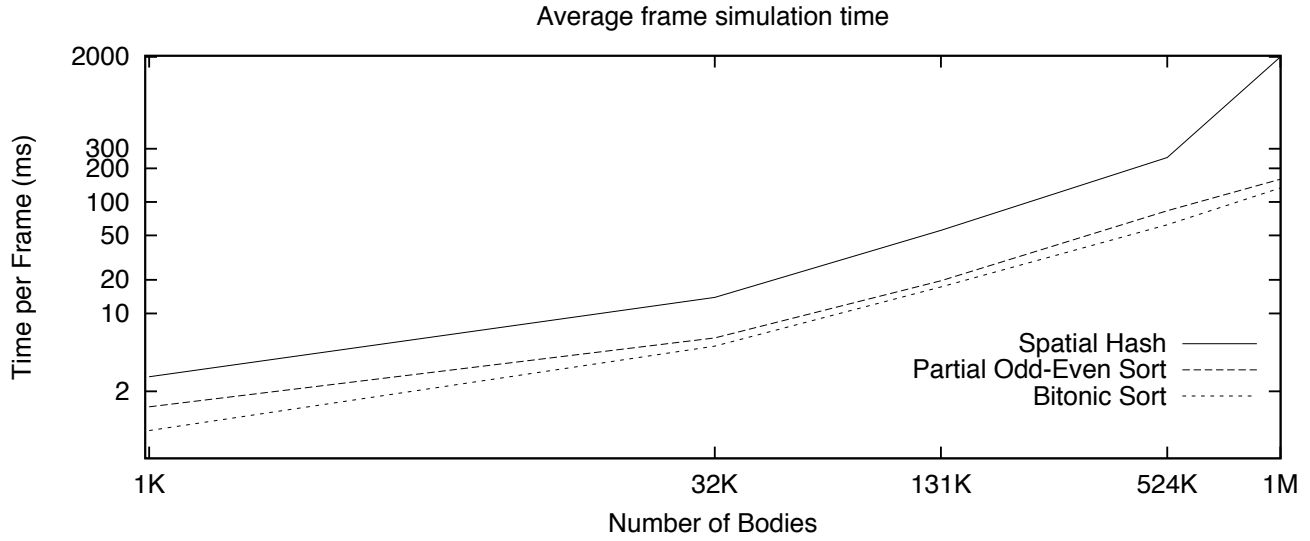

Figure 7: *Comparison of the evolution between the spatial hash and the bitonic and partial odd-even sort.*

such rendering at the end of the simulation can be easily predicted. To do so, we suggest to add the provided computational cost of the sorting and simulation passes (time spent in mili-seconds, with variable numbers of entities and neighborhood radius) to that of a VBO + vertex shader transforming and pixel shader lighting of several copies of the same display list, with more complex geometries, which can be found in published literature.

References

AMD, 2007. Amd stream computing. Available at: `http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf`. Acessed in 20/02/2009.

BATCHER, K. E. 1968. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, AFIPS, 307–314.

BLELLOCH, G. E., PLAXTON, C. G., LEISERSON, C. E., SMITH, S. J., MAGGS, B. M., AND ZAGHA, M., 1998. An experimental analysis of parallel sorting algorithms.

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 *Vision, Modeling, and Visualization (VMV)*, 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 (V-CROWDS)*, V-CROWDS.

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, CGI, 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., Siggraph, 29–38.

GROUP, K., 2009. Opencl - the open standard for parallel programming of heterogeneous systems. Avalible at: `http://www.khronos.org/opencl/`.

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*, Eurographics, 39–52.

NVIDIA, 2008. Bitonic sort demo. Avalible at: `http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html#bitonic`.

NVIDIA, 2009. Cuda technology. http://www.nvidia.com/cuda. Acessed in 20/02/2009.

PASSOS, E., JOSELLI, M., ZAMITH, M., ROCHA, J., MONTENEGRO, A., CLUA, E., CONCI, A., AND FEIJ, B. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, SBC, 81–86.

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, SCA, 99–108.

QUINN, M. J., METOYER, R. A., AND HUNTER-ZAWORSKI, K. 2003. Parallel implementation of the social forces model. In *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, PED, 63–74.

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, SIGGRAPH, 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, GDC.

REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Game Developers Conference 2000*, GDC.

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, Sandbox, 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, SCA, 19–28.

SHOPF, J., BARCZAK, J., OAT, C., AND TATARCHUK, N. 2008. March of the froblins: simulation and rendering massive crowds

of intelligent and detailed creatures on gpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, SIGGRAPH, 52–101.

SILVA, A. R., LAGES, W. S., AND CHAIMOWICZ, L. 2008. Improving boids algorithm in gpu using estimated self occlusion. In *Proceedings of SBGames'08 - VII Brazilian Symposium on Computer Games and Digital Entertainment*, Sociedade Brasileira de Computação, SBC, SBC, 41–46.

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, VRST, 7–14.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, SIGGRAPH, 1160–1168.