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 Feijó, PUC-RIO, ICAD Games
Eduardo Soluri, Nullpointer Tecnologia

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
esoluri@nullpointer.com.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].

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 [Joselli et al. 2008]. 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.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 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.

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.

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.

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.

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 1 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.

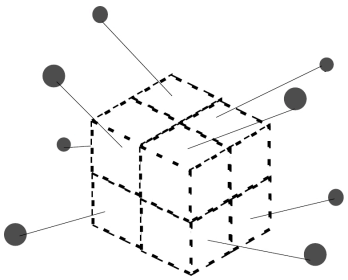

Figure 1: *An example of a distribution of entities in the neighborhood grid. Entities that are further away from the viewpoint are illustrated by small circles.*

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 2 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 2, 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. Our work extends this matrix example to a 3D grid maintaining the same form of information gathering, only adding the extra dimension.

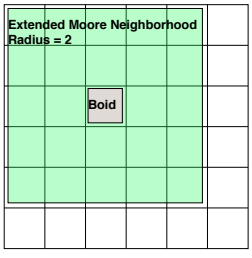

Figure 2: *Example of the Structure of the Extended Moore Neighborhood with 36 entities and radius = 2.*

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 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:

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

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 use a bitonic sort [Batcher 1968], which make a full sort in each dimension.

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).

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.

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.

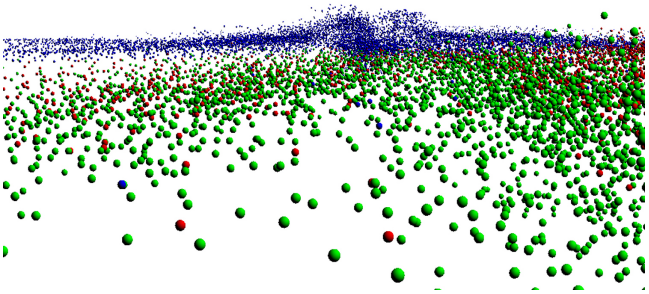

Figure 3: *Simulation with 32K boids.*

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. All tests results includes the simulation calculation and also the renderization to the screen.

Table 1 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.

Table 1: *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

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.

6 Conclusion

In this paper we have shown an architecture for simulating emergent behavior of dynamic entities in a densely populated environment. 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.

References

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.

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.

JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., AND FEIJÓ, PAGLIOSA, P. 2008. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 115–120.

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 FEIJÓ, 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.

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.

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.

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.