Improving Boids Algorithm in GPU using Estimated Self Occlusion

Alessandro Ribeiro da Silva[1]
Universidade Federal de Minas Gerais

Wallace Santos Lages[2]
Universidade Federal de Minas Gerais

Luiz Chaimowicz[3]
Universidade Federal de Minas Gerais

Abstract

Behavioral models are used in games and computer graphics for realistic simulation of massive crowds. In this paper, we present a GPU based implementation of Reynolds [1987] algorithm for simulating flocks of birds and propose an extension to consider environment self occlusion. We performed several experiments and the results showed that the proposed approach runs up to three times faster than the original algorithm when simulating high density crowds, without compromising significantly the original crowd behavior.

Keywords:: Boid simulation, GPGPU

Author's Contact:

[1]alessandrosilva@ufmg.br, [1]www.alessandrosilva.com
[2]wlages@ufmg.br
[3]chaimo@dcc.ufmg.br

1 Introduction

The simulation of a large number of individuals has applications in many different games, whether to compose the background scene in games (GTA, Rockstar Games, 1997) or as part of the gameplay itself (Pikimin, Nintendo, 2001). As members of the crowd meet each other, they interact by coordinating their motion accordingly to the goal of each individual. As examples we may cite the motion of flocks of birds, banks of fishes, herds of land animals, or even groups of human characters.

The first behavioral models appeared as extensions of particle systems used to model water, fire, grass and atmospheric effects [Reeves 1983]. Other extensions soon followed. In 1987, Reynolds presented a distributed model for controlling flocks of birds that considered interactions between agents [Reynolds 1987]. Although every agent, or *boid*, takes decisions considering only its local perception of the world, the sum of the behaviors enable the flock to present a very real-like motion.

However, to simulate local perception, one must be able to identify neighbors among all existing agents. The naive option (comparing each boid to the other) leads to a $O(n^2)$ behavior that becomes prohibitive for a large number of boids. So, to obtain an interactive system, we must have both fast implementations and low complexity algorithms.

To speedup the process of finding neighbors, researchers have used different spatial structures [Shao and Terzopoulos 2005], [Reynolds 2006]. Instead of searching in the whole population, this enables a local search in a pre-sorted structure, thus lowering the asymptotical complexity. Another approach used is to avoid the computation when neighbors do not change much [Chiara et al. 2004] or even update only a small percentage of the population per frame [Reynolds 2006].

On the other hand, current graphics architectures exhibit a large degree of parallelism which can be used for a highly efficient boid computation and display. GPU implementations are presented by Court and Musse [2005] and Chiara et al. [2004]. A fast implementation on the Playstation3 hardware was presented by Reynolds [2006].

In this work we present an implementation of the model proposed by Reynolds [1987], [1999] for a Geforce 8800 GPU. We also present an extension to estimate self occlusion in the neighbor computation and show how it can be used to improve the simulation performance. Our idea is to estimate the number of boids occluding the view cone of each boid and avoid considering *invisible* boids in the behavior calculation. This technique is orthogonal to the one mentioned above and specially useful for very dense populations.

The remainder of this paper begins with a review of the original boid model and other related work. We then present the graphics hardware mappings and algorithms we used to estimate density and behavior. Finally, we present results and conclusions.

2 Related Work

Agent simulation for large crowds is very computing expensive. Some techniques used to alleviate the problem are: parallelization, use of spatial structures, and heuristics to reduce the update rate of the crowd.

Quinn et al. [2003] presented a parallel pedestrian movement model running over 11 processors Linux-based multicomputer with MPI. They were able to simulate and render the motion of tens of thousands of pedestrians in real time using a manager/worker organization. Other researchers used the powerful parallelism of graphics processors to speedup the processing and display of large crowds. Chiara et al. [2004] present a massive simulation and rendering of a behavioral model using graphics hardware. They rendered a 3D scene with a flock of 8000 animated bird models at 20 fps. They describe the use of vector fields to manage obstacle avoidance and a heuristic that avoids recomputing the behaviors when the list of neighbors does not change. Courty and Musse [2005] used the GPU to compute a physics-based animation model which considers the influence of gaseous phenomena in the behavior of the crowd. This system, called FastCrowd, ran a crowd of 10,000 individuals at 50 fps without visualization and at 35 fps using impostors. The behavior model is very complex and include new psycho-physical forces. In 2006, Reynolds published an implementation for the PLAYSTATION3 hardware [Reynolds 2006]. He was able to concurrently simulate and display simple crowds of 15,000 individuals at 60 frames per second.

Since the number of individuals is large and the global behavior changes slowly, many researchers decoupled simulation update from rendering [Reynolds 2006], [Treuille et al. 2006]. As long as the position is properly updated, errors are very difficult to observe. On Reynolds implementation, [Reynolds 2006] only 1/8 of the individuals are updated at each frame. We preferred not to take this approach. Every simulation is fully computed for every individual on every frame.

Another way to improve speed is to use spatial hierarchies to quickly exclude individuals too far to influence the one being computed. For GPU computation, the most common data structure is the regular grid [Shao and Terzopoulos 2005],[Reynolds 2006]. More sophisticated data structures are more complex to navigate and therefore, slower. Some works do not use spatial structures at all and rely solely on brute force [Drone 2007].

As mentioned before, the main contribution of this work is a GPU implementation of the original algorithm by Reynolds, that considers visibility into the behavior of each boid. The visibility is estimated by computing the density of boids in the field of view.

3 Background Information

Visibility Culling

The goal of visibility culling is to quickly reject parts of the scene that are not visible for a given viewpoint. In computer graphics, occlusion culling is used to avoid processing or drawing such parts

of the scene. In our work visibility is used to avoid computing the influence of occluded boids. From the taxonomy proposed by Cohen-Or et al. [2000] the more relevant classifications for this work are:

- Point vs. Region. Point algorithms performs the computation with respect to the location of the current viewpoint only whereas from-region performs a computation valid for a region of the space. From-region visibility has its cost amortized over time but usually requires a longer processing.
- Image precision vs. Object precision. Object precision methods use the raw objects in their visibility computations. Image precision methods, on the other hand, operate on the discrete representation of the objects after they are rendered into images.
- Conservative vs. approximate. Conservative techniques overestimate the visible set. Approximate techniques may fail to include the entire visible set as a trade off for speed.

The technique employed in this work can be described as an approximate from-point visibility algorithm. In particular, we approximate visibility based on the volumetric representation of the scene. Instead of performing geometric visibility computations, we compute for each voxel the density of boids and approximate the visibility between regions by computing the volume opacity between them. This idea was first proposed by Sillion [1995] in the context of a radiosity system. Volumetric visibility was also independently developed by Klosowski and Silva [2000].

Behavioral Models

In this paper we implement the original model proposed by Reynolds [1987], [1999], where each boid steering behavior is computed independently based on its field of view. The field of view can be described by a maximum viewing distance and a angle of view (Figure 1).

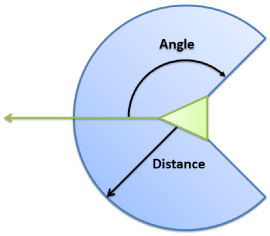

Figure 1: *Agent visibility (Adapted from Reynolds [1987]).*

Based on the visibility of each element, we consider three basic steering behaviors. The first, *separation*, is the behavior that prevents the boids from colliding. The steering force is computed as the average of the difference vector between current boid position and every neighbor. *Alignment* is the behavior that tends to align the boid with the average group direction. The *Cohesion* behavior moves the boid toward the center of his local neighborhood. The steering force is computed as the average position of the neighbors. An intuitive description of the behaviors is shown on Figure 2.

At each simulation step, the steering force is applied to the current position of every boid, and a new position is computed according to the resultant velocities.

4 GPU Mapping and Implementation

The graphics processing unit (GPU) was developed to transform, light and texture map triangles. For this reason, in GPGPU algorithms, data is usually encoded into textures or geometry before being processed by the GPU. The GPU mapping used in this work was based in general GPU programming techniques [Owens et al. 2007]. It uses texture maps to encode vectors of speed, position etc. This same technique has been used in other similar works [Kolb et al. 2004], [Chiara et al. 2004], [Courty and Musse 2005]. The execution flow can be divided in three distinct steps, two of them are performed by the CPU.

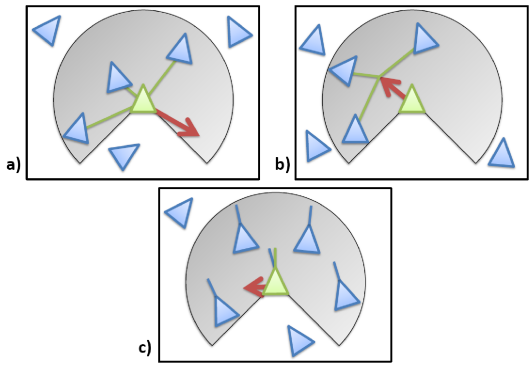

Figure 2: *The steering behaviors: a) Separation; b) Cohesion; c) Alignment (Adapted from Reynolds [1987]).*

Before delving into the algorithm, we shall first explain the data mapping and the data structure used.

4.1 Data Mapping

To describe the state of each boid, we need to store the following data: one translation vector, two orientation vectors (z and y axis) and a 3D force vector. This can be mapped into four RGBA textures. Since the graphics pipeline cannot be used to read and write at the same time, we used two copies of each texture. After each simulation step, input and output textures are switched (ping-pong buffering).

Since the maximum a 1D texture length allowed in our GPU is 8162, we used a function to map 1D address to a 2D address. The mapping function is defined by the Equation 1. Figure 3 shows the same mapping in a graphical form. This mapping was applied to every data addressed by an one dimensional coordinate. It will be referred as a virtual index since it does not represent the real address sent to the graphics API.

$$Tex2D = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1D\ index\ \mathbf{mod}\ TexWidth \\ \lfloor 1D\ index / TexWidth \rfloor \end{bmatrix}. \quad (1)$$

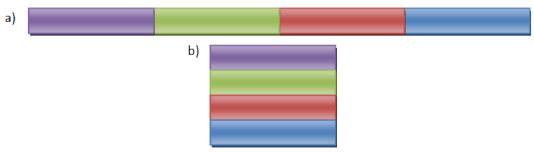

Figure 3: *Mapping of a 1D virtual index (a) to a 2D texture (b).*

4.2 Data Structure

The cost of neighbor search can be accelerated by using spatial indexing structures. We used a uniform grid since it has a constant cost to build and it is easy to evaluate inside the GPU. Other recursive structures, although more efficient, require a costly maintenance cost and are more complex to construct.

We encoded the grid structure as a 3D texture. Each position contains a virtual index to one boid. This index can be used to retrieve information about position, orientation or force in another texture. To be able to store more than one boid per cell, we used a linked list. This was implemented using the fourth coordinate (w) as a index to the next boid in the same cell. A value of -1 means the list has reached the end.

Figure 4 shows the mapping from the world space to the grid space, the mapping of the grid to a virtual index and the list of elements inside the position array.

In the virtual index implementation, all indexes are stored with a $+1$ increment. In this way, it is possible to use a *memset* function

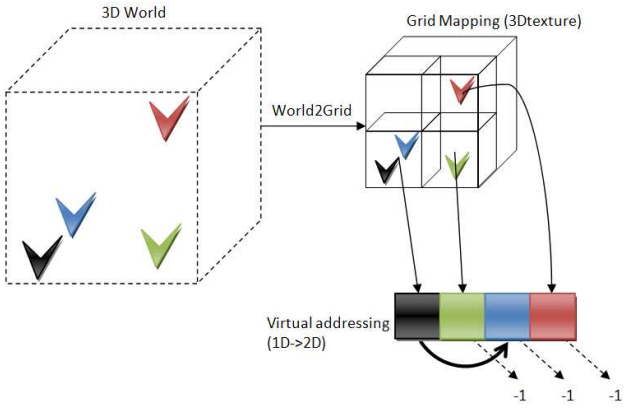

Figure 4: *Mapping of the 3D world to grid space and linked list indexing.*

to clear the grid content to zero. Inside the shader, all access and index conversions must subtract 1 from the elements to be accessed. A value of -1 after this operation means an invalid or null index.

4.3 Algorithm

As mentioned before, the execution flow can be divided in three distinct steps. The first step is the update of the grid structure (1). This is done on the CPU. Following, boids simulation is computed in the GPU (2) and finally they are rendered on the screen (3).

Step One: Grid update The first step objective is to associate each boid to a grid cell. This is necessary since at each step boids move among them. The application downloads the texture containing the position from the GPU and uses the values to update the internal grid indexes and the indexes of the position list. The grid construction is done in $O(m^3 + n)$, where m is the grid dimension and n the number of boids. After the construction, the application uploads the updated textures back to the GPU(Algorithm 1).

Algorithm 1 Grid structure construction algorithm.

```
1: Pos ← download positions from GPU
2: Grid ← clear grid content
3: for i ← 0 to n do
4:     Pos[i].w ← 0
5:     GridIndex ← Compute cell of pos[i]
6:     if Grid[GridIndex]exists then
7:         Pos[i].w ← next address pos + 1
8:     else
9:         Grid[GridIndex]← n + 1
10:    end if
11: end for
12: GPU_Positions ← upload Pos from CPU
13: GPU_Grid ← upload Grid from CPU
```

Step Two: Simulation The simulation step is done entirely inside the GPU, including the search for neighbors and the calculation of the vector for each behavior. The grid is used to estimate the visibility for each boid (Algorithm 2).

Step Three: Rendering To render the position of each boid, we used a static 3D model of a bird without texture. The model has 268 triangles and normals. The geometry was compiled in a *display list* [Opengl et al. 2005]. Since an OpenGL display list is static, we added a parameter to index the information of each boid in the position array. Using this method, it is possible to render all the static models with only one API call, reducing considerably the overhead due to matrix calls.

4.4 Grid cell visibility

The estimative of the visibility uses three levels of tests to avoid unnecessary processing of grid cells and individuals inside them.

Algorithm 2 Simulation algorithm.

```
1: GridPos ← Calculate the boid grid position
2: for i ← all the neighbor grid cell do
3:     GridIndex ← Compute cell of pos[i]
4:     if i is visible then
5:         for j ← all the neighbor in grid cell i do
6:             if j is visible then
7:                 Update Cohesion,
                   Alignment and Separation
8:             end if
9:         end for
10:    end if
11: end for
12: Desired force ← force based on vectors
13: lerp ← linear interpolation factor
14: FinalForce ← PreviousForce + (DesiredForce −
    PreviousForce) ∗ lerp
15: FinalTranslation ← Translation + FinalForce
16: Update axis 'y' and 'z'
17: Store FinalTranslation, FinalForce, 'y' and 'z'
```

First level: Maximum grid level

In the first test we select potential grid cells based on the max vision distance parameter. The output is a cube of cells with the local maximum cell count in all grid's direction (x,y,z) according to the Equation 2 (Figure 5a).

$$CellCount = \left\lceil \frac{visionDistance \times (GridSize - 1)}{WorldGridDimention} \right\rceil + 1. \quad (2)$$

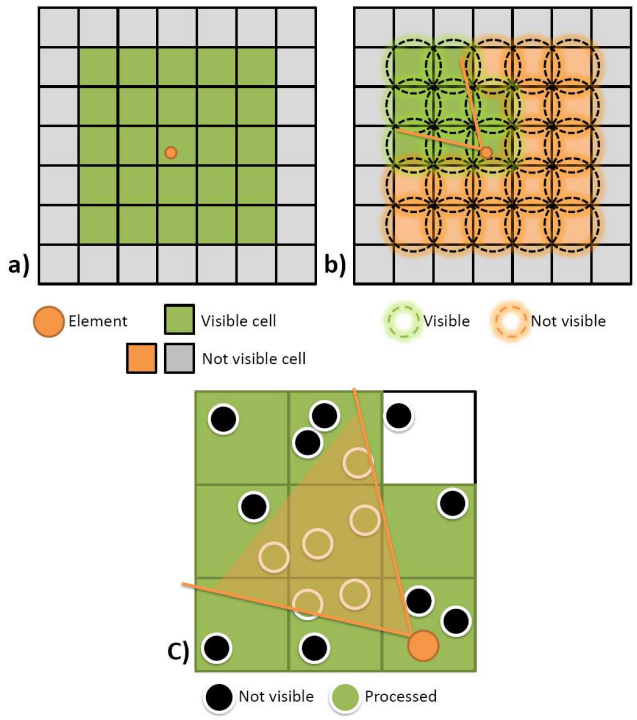

Figure 5: *Visibility tests: a) Maximum grid range; b) Sphere-cone test; c) Element test.*

Second level: Sphere-cone test

This test filters the grid cells that are not visible using sphere-cone collision. First we define spheres for each cell using the grid center as the sphere center and half of cell diagonal as the sphere radius. From the element orientation we construct a inverse rotation matrix that puts each grid cell sphere in the element local space. The cone-sphere test executes as a 2D test using the length of the sphere from local Z axis.

Figure 6 shows the result of each calculation. In a) we have the grid-sphere in grid space, b) shows the sphere after the inverse transformation, c) shows the 2D test space. Notice that the scale is not important to determine the cone visibility test since the ratio between the cone and the sphere will be the same.

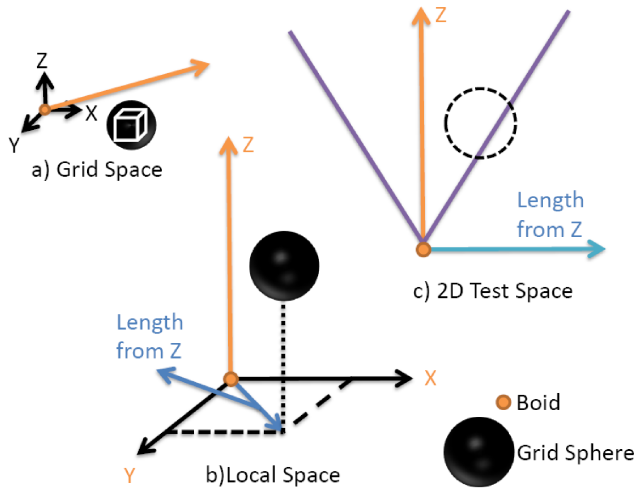

Figure 6: *a) The grid cell sphere in grid space; b) The grid cell sphere in the element local space; c) The 2D space used for the cone test.*

The output of this test is a grid cell filtered accordingly the element view (Figure 5 b).

Third level: Element test

After we know all visible cells, we iterate over all lists inside of them and test each element against the boid field of view. As output we have a list of the boids inside the field of view. This is the list used in the behavior calculation (Figure 5c).

4.5 Estimating self occlusion

Note that although many individuals are inside the field of view, not all of them would be really seen in a real situation. To estimate the visibility we iterate over the grid cells from inside to outside. Figure 7 shows one iteration example. As the grid becomes lighter the layer is increased.

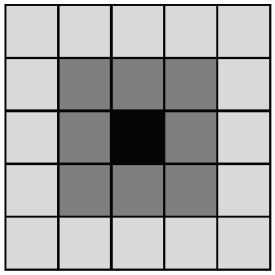

Figure 7: *Layer iteration. As the blocks become lighter the layer is increased.*

We stop the search when the number of processed neighbors reach the minimum visibility density or the maximum vision range is exceeded. In Figure 8, the minimum visibility density of four neighbors is reached before the maximum vision range and there are many neighbors discarded from the processing.

Grid size influence and estimation errors

Since the number of boids is only an estimative of the real occlusion, sometimes we may not consider boids that otherwise would be included in the behavior computation. This situation is depicted in Figure 9. This may change the behavior of the flock, specially if the number of boids being simulated is small.

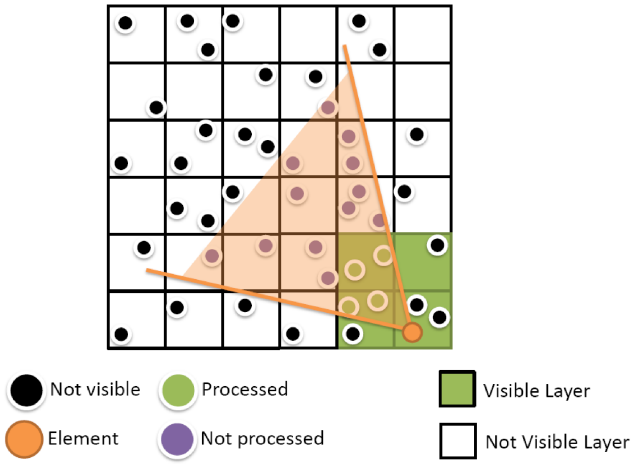

Figure 8: *Using visibility to cull boids.*

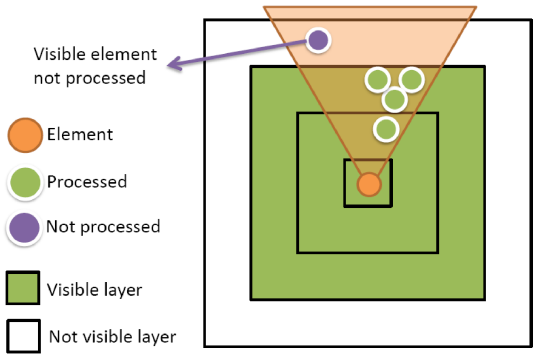

Figure 9: *Problem when the neighbors do not occlude another visible neighbor.*

Another important issue regarding the simulation is the grid density (elements per grid cell). When it uses on fixed number, if the density is too low the algorithm will discard a lot of empty cells and if the density is too high the algorithm will iterate over a lot of elements even if the most part of them are not visible.

We try to create a grid with a more uniform distribution by changing the size of the grid along the number of boids being simulated. The size of the grid is given by Equation 3, that considers the total number of boids and the desired density.

$$gridSize = \left\lceil \sqrt[3]{\frac{TotalBoids}{DesiredDensity}} \right\rceil + 1. \quad (3)$$

5 Results and Discussion

We performed a series of experiments in order to determine the effectiveness of the occlusion algorithm. The tests were executed in a PC with the following configuration:

- **Processor**: Athlon64x2 4200+
- **Memory**: 2Gb DDR1 - 400 Mhz
- **Graphics Processor**: GeForce8800GT - 512Mb DDR3 - PCI-e 16x

The shaders were written in Cg[Mark et al. 2003] and the application in C++.

We measured the time spent in the three steps described before. In the first step, we also measured the time spent transferring textures to and from the GPU and the time spent in grid construction. All times are in milliseconds.

We evaluated the influence of visibility estimation on rendering time. The number of boids changed from 256 to 207936 and the grid size changed according to the Equation 3. The vision angle was fixed in 45 degrees and the vision distance in 100 units. Our virtual environment extends from -500 to 500 units in the three axis. The model used for each boid has 268 triangles. Data was collected over a range of 45 different population sizes. The time for each one was obtained as the average of 30 frames.

The grid was an important factor for reducing the complexity of neighbor search in the algorithm, which would otherwise be $O(n^2)$ where n is the number of the boids.

Figure 10 shows the processing times of the different operations in each simulation step. Notice that only the grid construct time does not exhibit a linear increase. Since the grid construction needs to clear the entire grid in main memory, and the grid increases according the Equation 3, the total time increases accordingly to the grid storage complexity ($O(n^3)$).

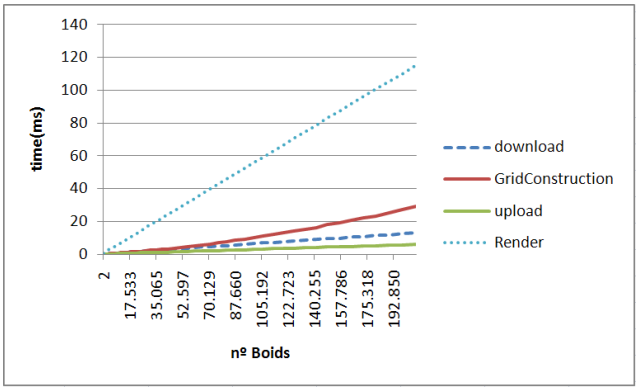

Figure 10: *Time to download, grid construction, upload and render in milliseconds.*

The upload time would have the same behavior, since it uploads the contents of the grid to the GPU, however it remains below all the other times as the number of boids increases. The render time and download time increases linearly with the number of boids. The grid construction time very low compared to the render time.

The simulation time is shown separately in Figure 11 because it is the only metric affected by the visibility estimation algorithm. Notice that the time near 70k and 192k boids increases abruptly, probably because these values affects some GPU internals, like cache. As expected, the simulation with visibility estimation consume less time than the algorithm without it.

The steep jumps on Figure 11 are similar to the ones observed on the NV4X Nvidia GPU architecture when resolving dynamic branches [Harris and Buck 2005]. However we are not sure if this also applies to the newer Tesla architecture (G8X, G9X). More tests would be required to settle down this point.

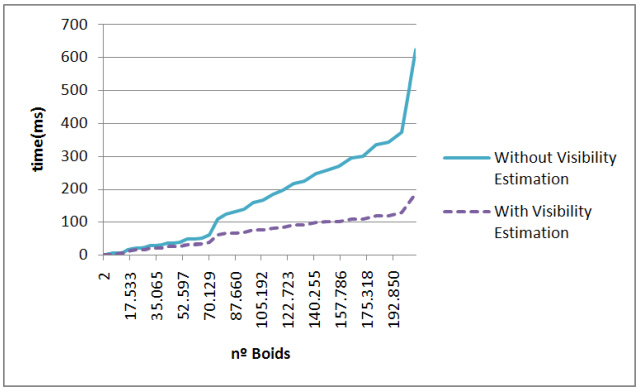

Figure 11: *Simulation time in milliseconds with and without the visibility estimation.*

To evaluate the time improvement of our approach, we computed the ratio (speedup) between the time taken by our implementation of Reynolds algorithm divided by the time taken by same algorithm with visibility estimation. Figure 12 shows the speedup obtained for an increasing number of boids. Notice that for few boids the speed up is smaller than 1, but for 10k boids or more, the speed up increases in almost a constant rate.

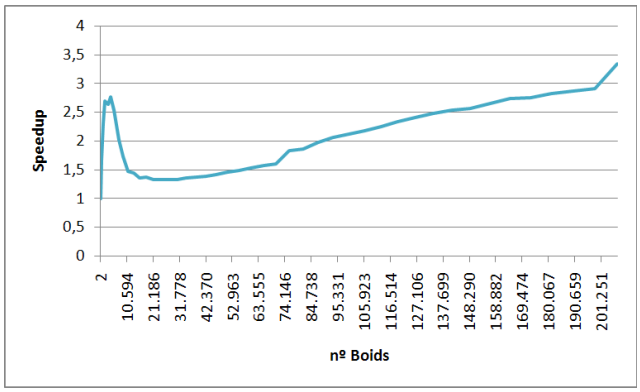

Figure 12: *Simulation with visibility speedup.*

When simulating boids, the expected result is a believable group simulation. However, after we turned on the visibility estimation, the simulation diverged from the behavior of the Reynolds original algorithm. This was expected since we change the dynamical parameters of the system. However very similar results were obtained by adjusting the steering constants. With the new parameters we could achieve a reasonable simulation with an speedup of more than 3 times of update rate. Since most crowd models usually require a lot of tweaking we do not see this as a real concern.

Figure 13 shows one example of a simulation with 20164 boids.

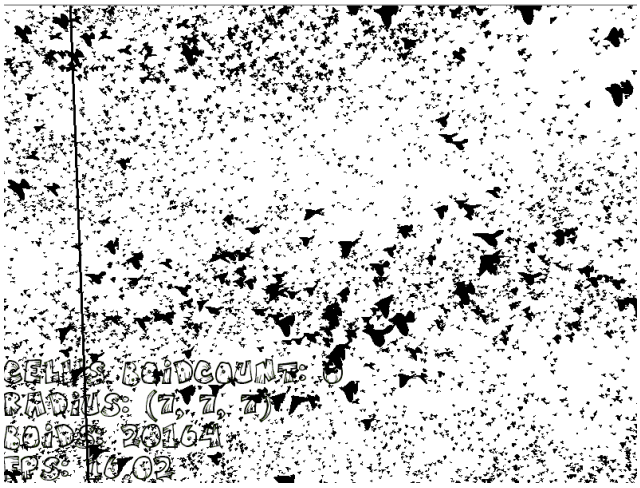

Figure 13: *Simulation example of 20164 boids.*

6 Conclusion and Future Work

We presented a mapping of a behavioral model to run in a GPU and proposed one extension to optimize the neighbor search by using a visibility estimation. The experimental results showed that this technique can be very effective in reducing the total complexity of crowd simulation. Some of the future extensions for this work include:

- Evaluate other types of occlusion estimation.
- Optimization of the linked list structure for cache access.
- Use a 2D texture for encoding the grid structure instead of a 3D
- Evaluate other spatial subdivision structures

- Include environment obstacle avoidance.

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.

COHEN-OR, D., CHRYSANTHOU, Y., AND SILVA, C. 2000. A survey of visibility for walkthrough applications. *Proc. of EUROGRAPHICS'00, course notes*.

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.

DRONE, S. 2007. Real-time particle systems on the gpu in dynamic environments. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 80–96.

HARRIS, M., AND BUCK, I. 2005. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, ch. GPU Flow-Control Idioms, 547–555.

KLOSOWSKI, J. T., AND SILVA, C. T. 2000. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics 6*, 2, 108–123.

KOLB, A., LATTA, L., AND REZK-SALAMA, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, New York, NY, USA, 123–131.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM Press, New York, NY, USA, 896–907.

OPENGL, SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2005. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1, 80–113.

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

REEVES, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph. 2*, 2, 91–108.

REYNOLDS, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th Annual Conference on Computer Graphics and interactive Techniques*, 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference 1999*, Miller Freeman Game Group, San Francisco, California, 763–782.

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.

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.

SILLION, F. X. 1995. A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. *IEEE Transactions on Visualization and Computer Graphics 1*, 3, 240–254.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. *ACM Trans. Graph. 25*, 3, 1160–1168.