Boids that See: Using Self-Occlusion for Simulating Large Groups on GPUs[1]

ALESSANDRO RIBEIRO DA SILVA, WALLACE SANTOS LAGES, LUIZ CHAIMOWICZ

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
alessandro.ribeiro.silva@gmail.com, wallacesantos@yahoo.com, chaimo@dcc.ufmg.br

Behavioral models have been used in the entertainment industry to increase the realism in the simulation of large groups of individuals. Unfortunately, the classical models can be very compute-intensive when very large groups are considered, reducing its applicability in games and other interactive systems. In this article we explore both search space reduction and parallelism to improve the performance of Reynold's *Boids* model. We propose a methodology that considers self-occlusion (visibility culling) to reduce the number of neighbors and we take advantage the parallelism present in common graphics processor units (GPUs) to allow the simulation of very large groups. We performed different GPU implementations (GPGPU and CUDA); the results show that visibility culling allows significant gains in performance without affecting the model's overall behavior.

Categories and Subject Descriptors: **Algorithms, Performance**

General Terms:
I.2.1 Applications and Expert Systems (Games)
I.3.1 Hardware Architecture (Graphics processors, Parallel processing)

Additional Key Words and Phrases: **Boids Simulation, Emergent Behavior, GPGPU**

1 INTRODUCTION

In the last couple of decades, the entertainment industry has relied on advanced computer graphics techniques to improve the impact of its movies and games. One technique that has attracted lot of interest is the simulation of large groups of individuals--for example, herds of land animals, flocks of birds, schools of fish, even crowds of humans. In these simulations, as members of the group meet each other, they interact by coordinating their motions according to the goal of each individual. As a result of these interactions, very realistic effects emerge, making movies and games more interesting and entertaining. Some examples in the movie industry are the wildebeest stampede in *Lion King* and the epic battles in *The Lord of the Rings* trilogy. It has also been extensively used in digital games, whether to compose the background scene (for example, in *GTA - Grand Theft Auto*) or as part of the gameplay itself (as in *Pikimin*, from Nintendo).

One of the most used models for simulating large groups of individuals was proposed by Reynolds [1987] to simulate a flock of birds. It is a completely distributed system in which agents take decisions based only on their local perception. As explained in Section 3.1, the behavior of each agent (called a *boid*) is influenced by its neighbors and the composition of these behaviors enables the flock to present a very realistic 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 $O(n^2)$ complexity, which becomes prohibitive for a large number of boids.

Hence, to use Reynolds' algorithm in games and other interactive systems, it was necessary to find ways of

[1] da Silva, A. R., Lages, W. S., and Chaimowicz, L. 2009. Boids that see: Using self-occlusion for simulating large groups on GPUs. ACM Comput. Entertain. 7, 4, Article 51 (December 2009), 20 pages. DOI = 10.1145/1658866.1658870 http://doi.acm.org/10.1145/1658866.1658870.

improving its performance. One path for improvement is to limit the search space when selecting the neighbors in order to avoid the $O(n^2)$ worst case. Another possibility is to use the graphics processor unit (GPU) to perform parts of the algorithm, taking advantage of the high level of parallelism in this specialized hardware.

In this article we propose a methodology that estimates self-occlusion, that is, the possible occlusion caused by a boid in view of the other boids during flight. In this way we can reduce the number of neighbors considered in the computations by discarding boids that would be invisible due to the presence of other boids. We then tested our methodology with two different GPU implementations: one using the Cg shader language and techniques for general-purpose computation on GPUs (GPGPU), and the other using the Nvidia's compute unified device architecture (CUDA), a software platform created by Nvidia for massively parallel high-performance computing on GPUs [Halfhill 2008]. We perform experiments with a large number of boids and discuss the main characteristics of these implementations.

This article is organized as follows: in the next section, we discuss some related work regarding the simulation of large groups of individuals. Section 3 explains Reynold's model and provides some background about visibility culling and GPU programming. In Section 4, we describe the GPU implementation, and in Section 5, we present the self-occlusion estimation methodology. Experiments are presented and discussed in Section 6, followed by the conclusions and possibilities for future work in Section 7.

2 RELATED WORK

The first behavioral models for the simulation of a large number of agents appeared as extensions of particle systems used to model water, fire, grass, and atmospheric effects [Reeves 1983]. Shortly after that, Reynolds presented a distributed model for controlling flocks of birds that considered interactions between agents [Reynolds 1987]. As will be explained in the next section, Reynold's model considers that each agent is subjected to forces that make it move together with its neighbors. Since this seminal work, several models and applications have been proposed in diverse areas such as human crowd simulation [Thalmann and Musse 2007], artificial life [Terzopoulos et al. 1994], and even robotics [Marcolino and Chaimowicz 2008].

Unfortunately, the cost of finding neighbors and computing the resultant forces in Reynolds' and other similar flocking models can be very expensive for larger groups. Thus, some researchers have been studying ways of alleviating this problem. One of the main research thrusts in this area is to distribute the computation to multiple processors / computers. 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. In 2006, Reynolds implemented his model using 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 updates from rendering [Reynolds 2006; Treuille et al. 2006]. For example, in the Reynolds [2006] implementation, only 1/8 of the individuals are updated at each frame. As long as the position is properly updated, errors are very difficult to observe.

Other researchers explored the high level of parallelism of graphics processors to speedup the processing and display of large crowds. Chiara et al. [2004] presented 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 frames per second (fps). They used vector fields to manage obstacle avoidance and a heuristic to avoid 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 includes new psychophysical forces. In [Drone 2007] there is a good overview about different techniques for the implementation of particle systems in GPU.

Another way to improve performance is to mix the parallel implementation with spatial hierarchies to quickly exclude individuals too far to influence the one being computed. Instead of searching the whole population, this

enables a local search in a presorted structure, thus lowering the asymptotical complexity. For example, Shao and Terzopoulos [2005] used a hierarchical data structure to simulate a large number of pedestrians moving in a virtual environment. For GPU computation, the most common data structure is the regular grid. Breitbart [2008] presents an implementation of Reynolds model running over the CUDA CuPP framework. Besides various strategies to create and process the grid, the author also proposes a dynamic grid, which uses CUDA shared memory to accelerate the neighbor search. Passos et al. [2008] developed a GPU implementation using CUDA that is able to simulate a large group of boids at near 30 fps without using an explicit data structure for searching.

As mentioned before, this article explores both the reduction of the search space and the use of parallelism to improve the boids algorithm. Our main contribution is a novel methodology that considers only "visible" boids in the computation, that is, each boid considers only those neighbors that are not occluded by others. This is especially useful for very dense populations. We implement this approach in GPU using different techniques: GPGPU and CUDA, and discuss some characteristics of both implementations.

3 BACKGROUND

3.1 Reynolds Boid Model

In this article we implement the original model proposed by Reynolds [1987] for simulating a flock of flying creatures, called *boids*. This model consists of three basic steering behaviors (*separation*, *alignment,* and *cohesion*) that describe how an individual boid moves according to the positions and velocities of its neighbors. An intuitive description of the behaviors is shown on Figure 1.

The first behavior, *separation*, prevents the boids from colliding. The steering force is computed as the average of the difference vector between the current boid position and every neighbor (Figure 1(a)). The *cohesion* behavior moves the boid towards the center of his local neighborhood. The steering force is computed to move the boid towards the average position of the neighbors (Figure 1(b)). Finally, *alignment* is the behavior that makes the boids fly in the same direction. The steering force is computed in such a way that the boid will align itself in the same direction as the average group direction (Figure 1(c)).

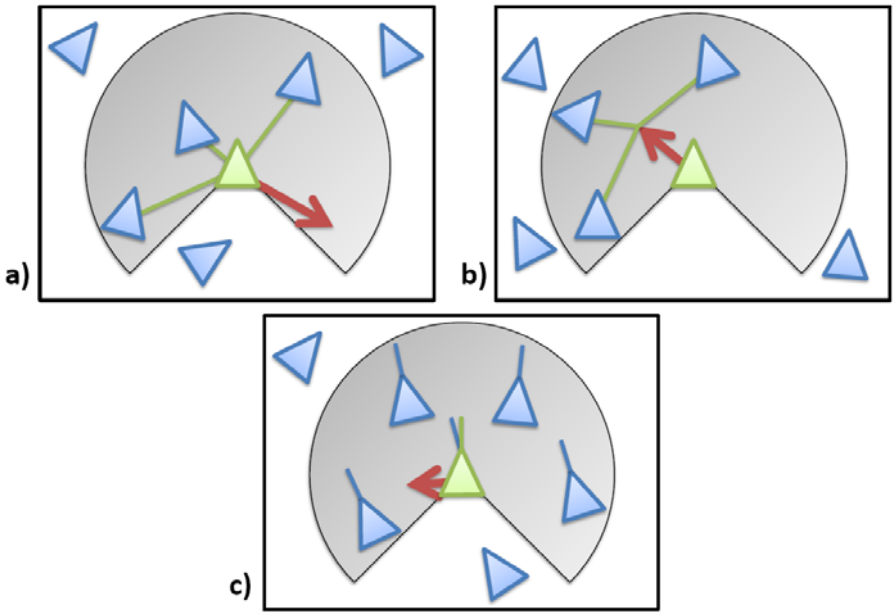
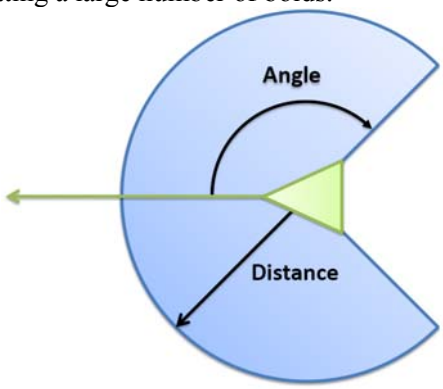


Fig. 1. The steering behaviors: (a) separation; (b) cohesion; (c) alignment (adapted from Reynolds [1987]).

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. Since this is a dynamical system with lots of interactions, it is very difficult to predict the final group behavior. Hence, it is normally called *emergent behavior*.

An important characteristic of this model is to determine the neighborhood of each boid. This is normally implemented using a "field of view" that can be described by a maximum viewing distance and viewing angle, as shown in Figure 2. Only boids inside this field of view are considered for computing the steering behaviors. As will be explained in Section 4, we augment this concept of "field of view" to also consider occlusion in order to improve performance when simulating a large number of boids.



Fig, 2. Agent visibility (adapted from Reynolds [1987]).

3.2 Visibility Culling

The goal of visibility culling is to quickly reject those parts of the scene that are not visible from 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 as follows:

Point vs. Region: Point algorithms perform 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 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].

3.3 GPU Programming

There are two different ways to use GPU computing power for generic programming. The first option is to

encode the data into textures or geometry and write the algorithm as a standard graphics shader [Owens et al. 2007]. In this case, the algorithm is implemented as an OpenGL or DirectX program and executes as a graphics application. This approach is called *general-purpose computing on GPU* (GPGPU), and has been used in other, similar work [Kolb et al. 2004; Chiara et al. 2004; Courty and Musse 2005].

Another option is to use the recently developed architectures that give access to the GPU directly from standard programming languages. Among them we may cite CUDA (compute unified device architecture) from Nvidia, Stream from AMD or the OpenCL (Open Computing Library) from the Khronos group. This approach does not require mapping the input data to graphics primitives, and enables more flexibility such as scattered reads and writes.

4 IMPLEMENTATION

We implemented the boids model and our visibility algorithm using both shaders (GPGPU) and CUDA. In the shader implementation, we used the Cg (C for graphics) language with the OpenGL API. In the next section we describe the main characteristics of both implementations.

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 axes) and a 3D force vector.

In the GPGPU programming model, we must encode the input information as graphics primitives to send them into the GPU. Hence the state of each boid is 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 1D texture length allowed in our GPU is about 8K, we used a function to map 1D address to a 2D address. The mapping function is defined by the equation below. Figure 3 shows this mapping in a graphical form. This mapping was applied to every data addressed by a one-dimensional coordinate. It will be referred as a virtual index, since it does not represent the real address sent to the graphics API.

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

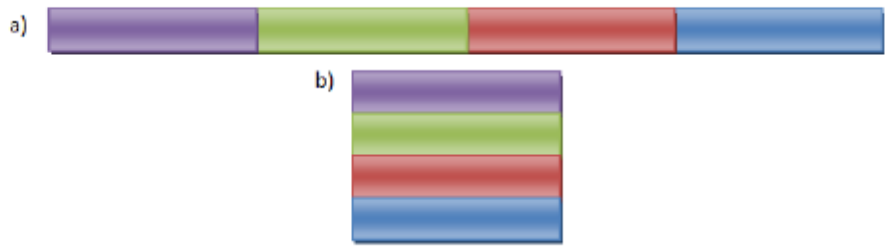


Fig. 3. Mapping of a 1D virtual index (a) to a 2D texture (b).

CUDA allows a different mapping process. Basically, there are three main types of memory in CUDA: global,

texture, and shared memory. The global memory is not cached, and is therefore slow. Texture memory implements cache logic in hardware and is persistent across thread blocks. Finally, shared memory is a very slow latency (1 cycle) memory divided into 16k chunks. However, it is not persistent across blocks and must be managed by the application.

We implemented two versions of the algorithm: The naive implementation uses the global memory to store several contiguous arrays representing the boid's information. Each array is allocated twice, so that the current step can use information from the previous step.

In the optimized implementation, there is only one set of arrays in global memory. Each array, however, has a corresponding copy in texture memory that is used for kernel reads. The data in the texture memory is kept updated by using a CUDA operation to copy it from global to texture memory. The data is read from the textures using the same mapping shown in Figure 3.

4.2 Data Structure

The cost of neighbor search can be accelerated by using spatial indexing structures. We used a uniform grid because it has a constant cost to build and is easy to evaluate inside the GPU. Other recursive structures, although more efficient, require costly maintenance and are more complex to construct. The structure used is shown in the Figure 4: there is a regular division in the space in the three axes that indexes lists of boids.

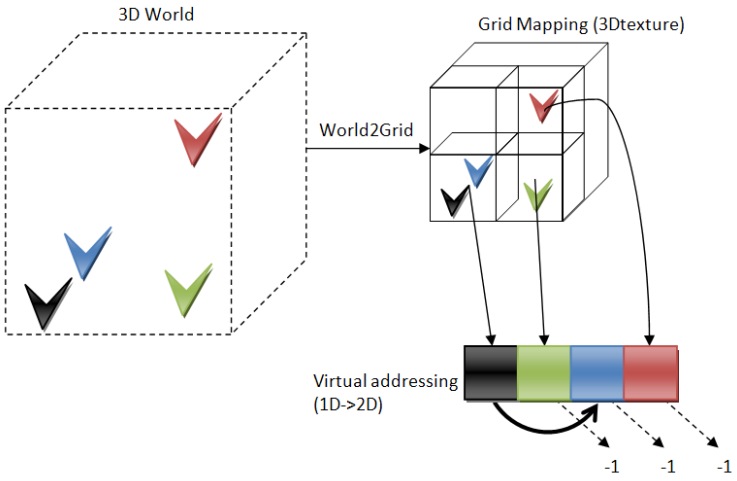


Fig. 4. Mapping of the 3D world to grid space and linked list indexing.

In the GPGPU programming model, 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 an 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 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.

In CUDA, we used an integer array as one grid. The access is possible according the mapping from 3D coordinate to a 1D index according to the following formula:

$$GridIndex1D = x + y \times gridWidth + z \times gridWidth \times gridHeight \ .$$

4.3 Algorithm

The execution flow of the algorithm can be divided into three distinct steps: (1) update the grid structure; (2) boids simulation is computed in the GPU; and (3) boids are rendered on the screen. These three steps are described next.

Step One: *Grid Update*. The objective of the first step is to associate each boid to a grid cell. This is necessary, since at each step boids move among them. In the shader version, 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. So in this model, the grid construction is done on the CPU 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(a)).

Algorithm 1(a). Shader grid structure construction algorithm.

```
1: Pos ← download positions from GPU
 2: Grid ← clear grid content with zeros
 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
```

In the CUDA model, the grid is computed directly inside the GPU via the atomic operation *atomicExch* (available on CUDA version 1.1), as shown in Algorithm 1(b).

Algorithm 1(b). CUDA grid structure construction algorithm.

```
1:void computar_grid(int* grid,  int* boid_next){
2:  unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
3:  unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
4:  int boidIndex = y*boidW+x;
5:  int3 gridIndex = POS2GRID(readBoidPosition(boidIndex)); //the read changes if the program use textures
6:  int gridPosition = GRIDindex1D(gridIndex);
7:  int currentCell2boid = atomicExch(&grid[gridPosition], boidIndex);
8:  boid_next[boidIndex] = currentCell2boid;
9:}
```

In the optimized CUDA version, after the grid kernel execution, there are two memory transfers: one from the

array with the grid and the other from the array “next_result” to their correspondent textures. Both algorithms use the same steps: (1) clear the indexes in the grid structure (Figure 5(a)); and (2) update it according to the boids position in the world space with the grid division. The update is made for each boid, as shown in Figure 5(b).

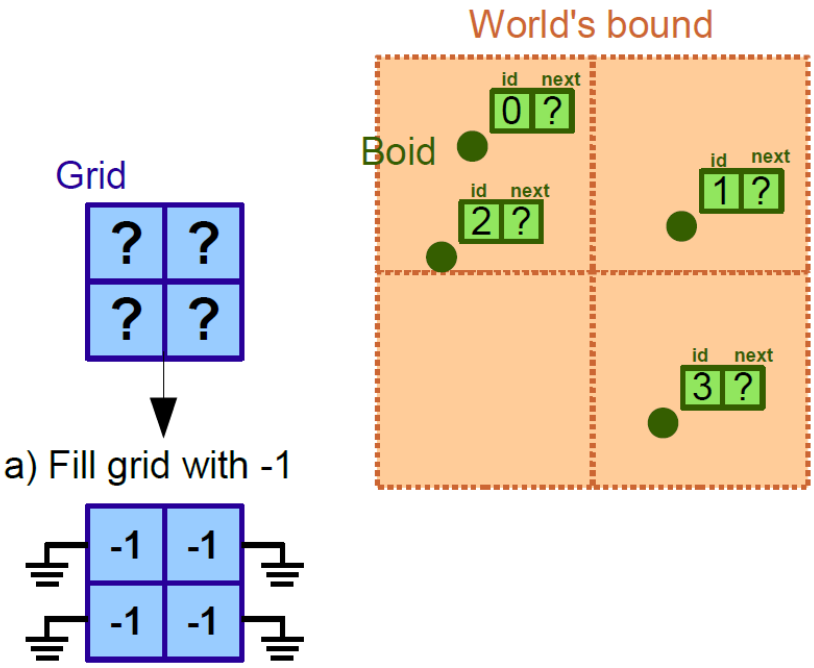


Fig. 5(a). Clearing the indexes in the grid structure.

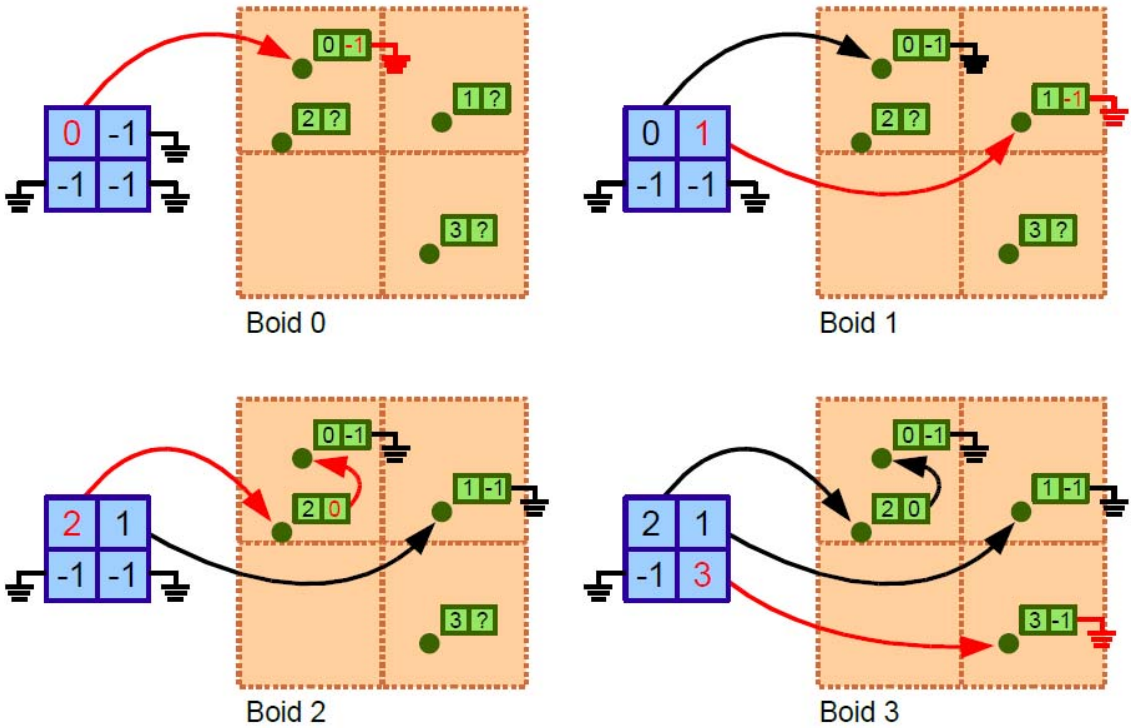


Fig. 5(b). Updating according to the boids position in the world space.

Step Two: *Simulation.* The simulation step is done entirely inside the GPU in both models, 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).

Algorithm 2. Simulation algorithm.

```
GridPos ← Calculate the boid grid position
for i ← all the neighbor grid cell do
  GridIndex ← Compute cell of pos[i]
  if i is visible then
    for j ← all the neighbor in grid cell i do
      if j is visible then
        Update Cohesion, Alignment and Separation vectors
      end if
    end for
  end if
end for
Desired Velocity ← force based on vectors
lerp ← linear interpolation factor
FinalVelocity ← PreviousForce + (DesiredVelocity □ PreviousVelocity) lerp
FinalTranslation ← Translation + FinalVelocity
Update axis y and z
Store FinalTranslation; FinalVelocity; y and z
```

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 [Shreiner 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

To estimate visibility, three levels of tests are used to avoid unnecessary processing of grid cells and the individuals inside them.

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 for all the directions of the grid (x,y,z) according to Equation 2 (Figure 6(a)).

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

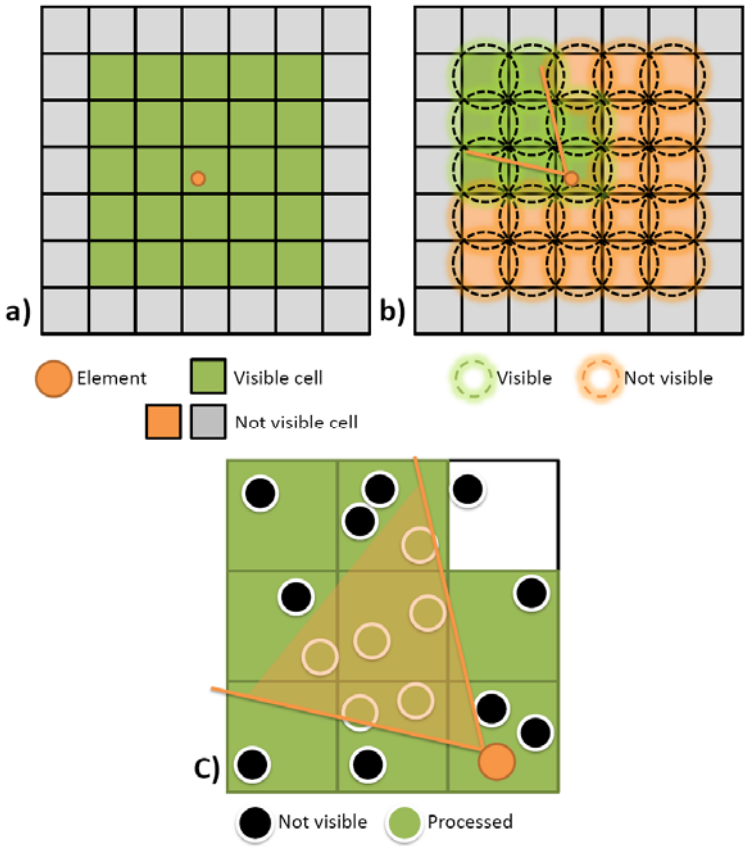


Fig. 6. Visibility tests: (a) maximum grid range; (b) sphere-cone test; (c) element test.

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 the 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 7 shows the result of each calculation. In a) we have the grid cell 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. The output of this test is a grid cell filtered accordingly the element view (Figure 6b).

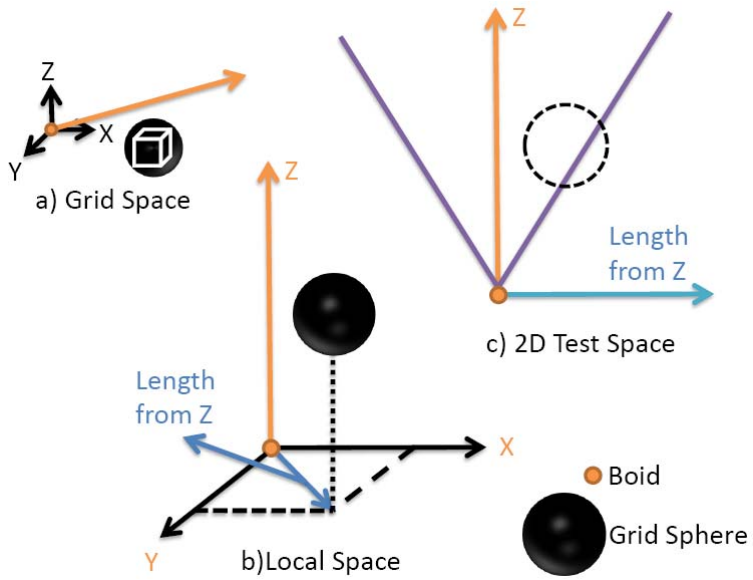


Fig. 7. (a) The grid cell sphere in grid space; (b) the grid cell sphere in the element local space; (c) the 2D space for the cone test.

Third Level: *Element Test.* After we know of all the 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 6(c)).

5 ESTIMATING SELF-OCCLUSION

In order to improve performance and allow the simulation of a very large number of entities, we propose a new methodology to compute neighbors with Reynold's algorithm. The basic idea is that although many individuals are inside a boids regular field of view, not all of them would actually be seen in a real situation, mainly in a much cluttered environment.

The algorithm to estimate the visibility iterates over the grid cells from inside to outside, as shown in Figure 8.

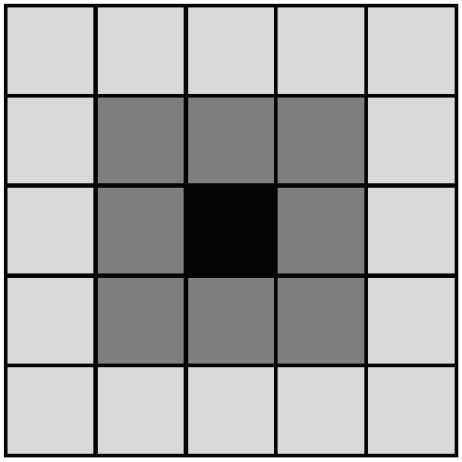


Fig. 8. Layer iteration. As the blocks become lighter, the layer is increased.

We stop the search when the number of processed neighbors reaches the minimum visibility density or when the maximum vision range is exceeded. In Figure 9, the minimum visibility density of four neighbors is reached before the maximum vision range, and there are many neighbors discarded from processing. Since the number of boids is only an estimative of the real occlusion, we may at times not consider boids that would otherwise be included in the behavior computation; this situation is depicted in Figure 10. This may change the behavior of the flock, especially if the number of boids being simulated is small. Another important issue regarding the simulation is grid density (elements per grid cell). When a fixed number is used, 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 when most of them are not visible.

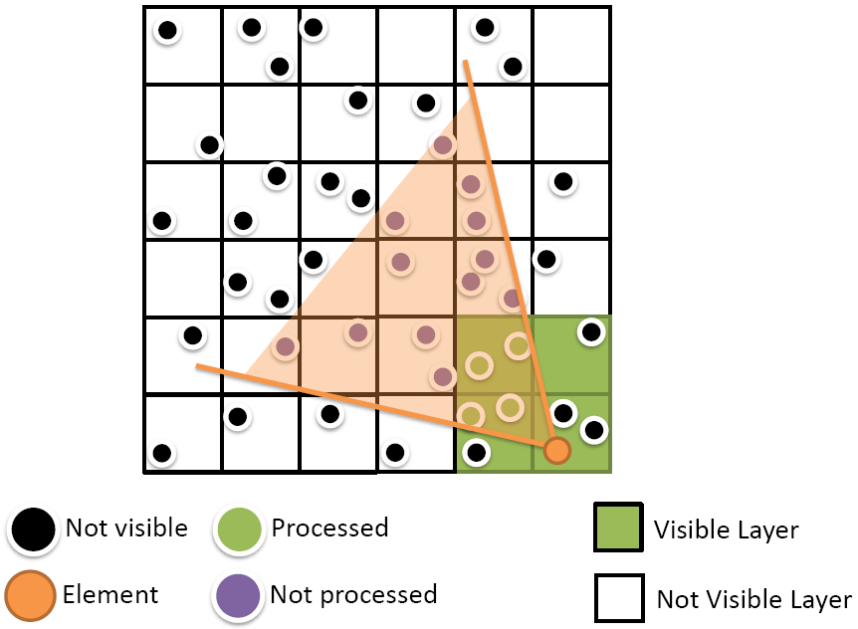


Fig. 9: Using visibility to cull boids.

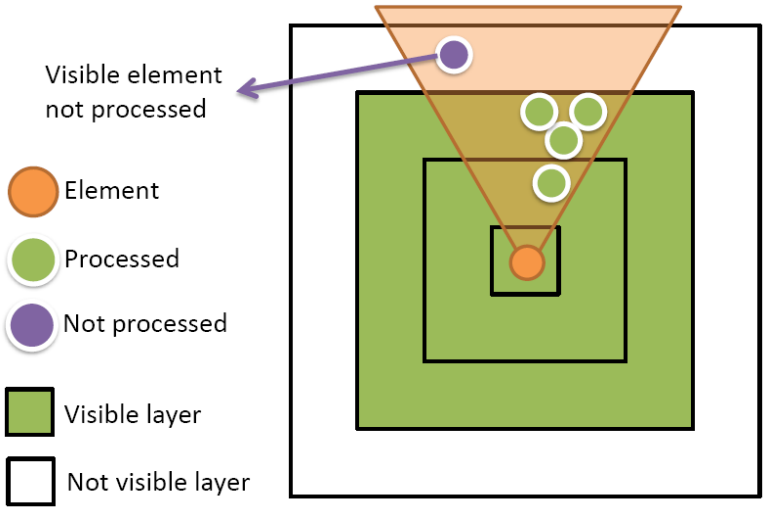


Fig. 10. A problem when the neighbors do not occlude another visible neighbor.

We tried to create a grid with a more uniform distribution by changing the size of the grid in accordance with the number of boids being simulated. The size of the grid is given by the equation below, which considers the total number of boids and the desired density.

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

6 RESULTS AND DISCUSSION

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

- Processor: Athlon64x2 4200+
- Memory: 2Gb DDR1 - 400 Mhz
- Operational System: Windows Vista 64
- Graphics Processor: GeForce8800GT - 512Mb DDR3 - PCI-e 16x
- Graphics Driver Version: 181.22
- Cg API: 2.0

We tested three implementations: the shader, naive CUDA, and an optimized CUDA with cache optimization through textures.

We measured the time spent in the three steps described before. All the measured times are in milliseconds. The first step (grid construction) in the shader and the optimized CUDA version includes the data transfer time (CUDA: from global memory to texture; shader: texture uploads and download). In the second step (simulation), the transfer time is only included for the optimized CUDA times.

In the experiments, we varied the number of boids from 1024 to 65536. This interval was chosen so that the square root of the number is a multiple of 16. Hence we could use the CUDA blocks of 16x16. The grid size changed according to Equation 3. The vision angle was fixed at 45 degrees and the vision distance at 200 units. Our virtual environment extends from -500 to 500 units on the three axes. The model for each boid has 268 triangles; the density parameter was 5 boids per cell; data was collected over a range of 6 different population sizes; and the time for each one was an average of more than 30 frames.

Figure 11 shows the time spent on grid construction for each implemented algorithm. Both CUDA algorithms performed better. Although the optimized CUDA had to transfer the result array from global to texture memory, the time was very close to the other CUDA algorithm. The shader grid construction was the slowest, mainly influenced by the CPU part. It was not possible to implement scattering as a programming strategy in the shader as it was in CUDA.

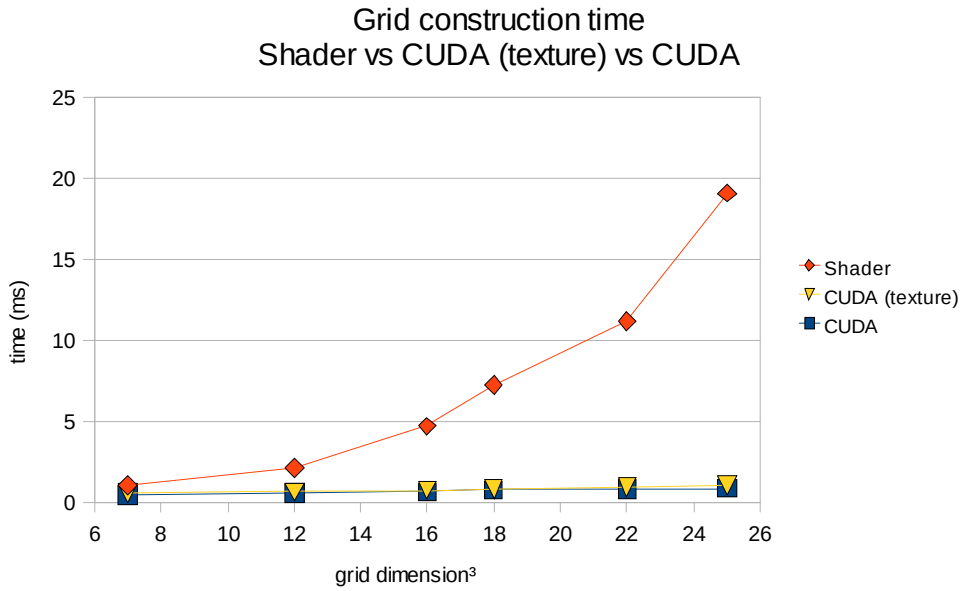


Fig. 11. Grid construction times: shader construction time is greater since it was executed on the CPU.

In Figures 12, 13, and 14 we can see the total simulation times for the three implementations with and without visibility culling. All of them show a significant gain when using visibility culling. The biggest improvement can be seen in the shader implementation.

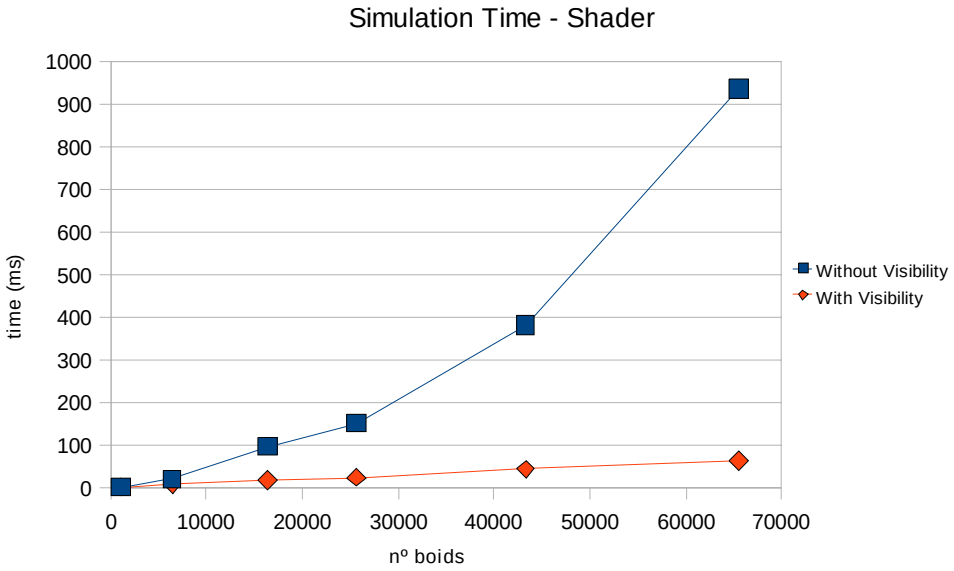


Fig. 12. Simulation time for the GPGPU implementation.

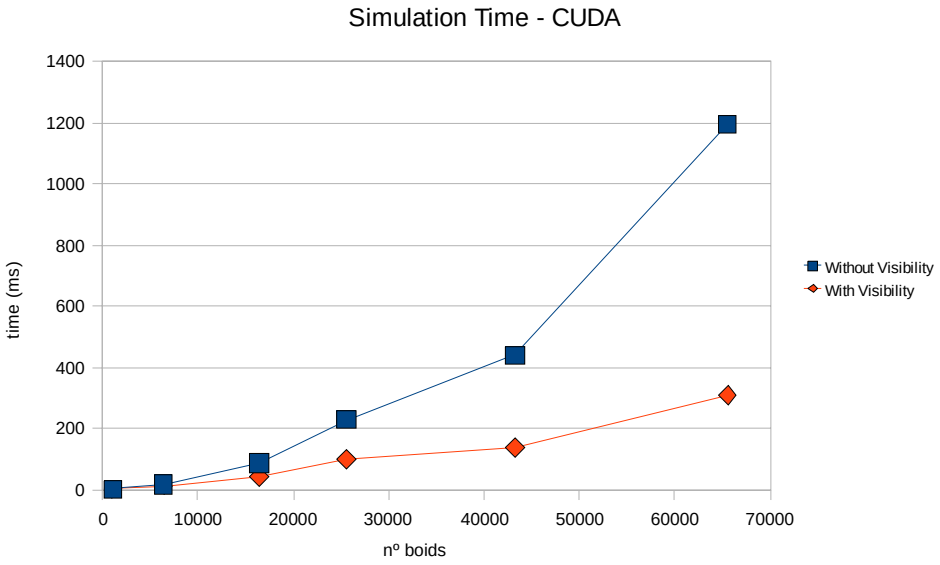


Fig.13. Simulation time for the CUDA with global memory implementation.

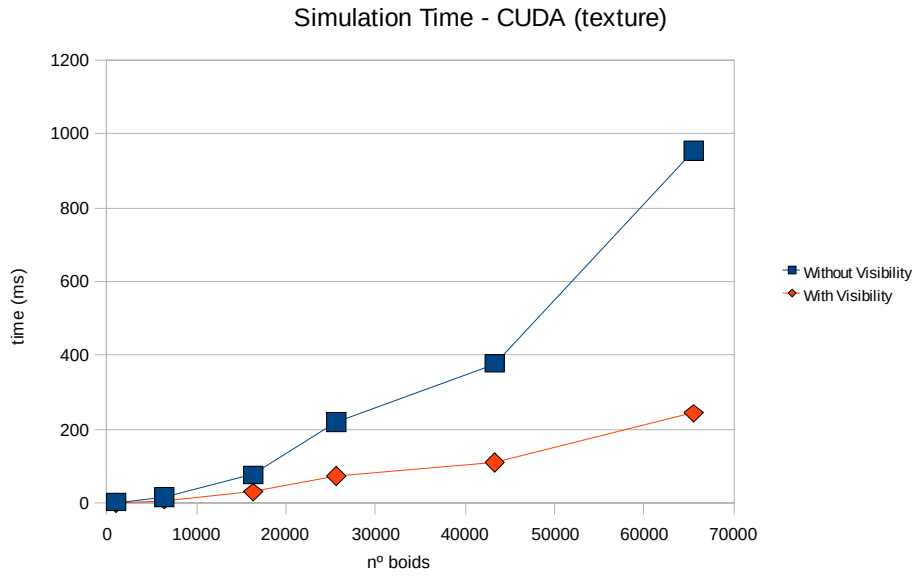


Fig. 14. Simulation time for the CUDA with texture memory implementation.

Figure 15 shows the total time for the three implementations with visibility culling. Comparing only the CUDA implementations, we can say that the use of texture as a cached memory worked well because the time spent decreased slightly. But among the three implementations, the shader gave the best performance, running more than three times faster than CUDA for a large number of boids.

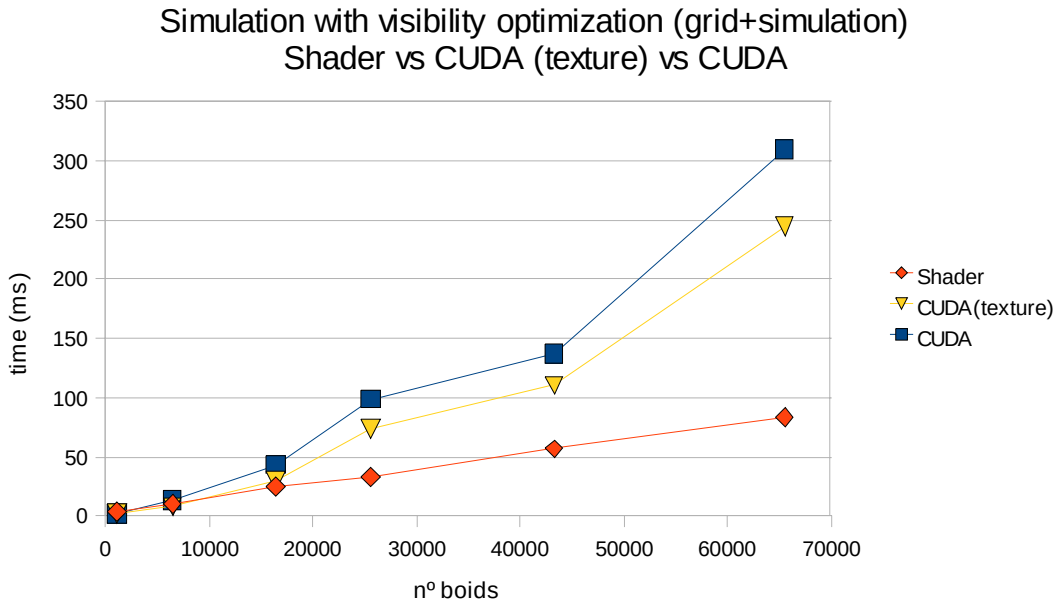


Fig. 15. Total simulation times (grid+simulation) with visibility culling.

Considering all the results, we can say that grid construction is very effective in the CUDA implementation, while simulation is faster in the shader implementation. Thus, the best of both worlds would be the CUDA grid construction with a shader simulation.

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 original Reynolds algorithm. This was expected, since we changed the dynamic parameters of the system. However, very similar results were obtained by adjusting the steering constants. With the new parameters we achieved a reasonable simulation with a speedup of more than three times the update rate. Since most crowd models usually require a lot of tweaking, we do not see this as a real concern. Figure 16 shows one example of a simulation with 20164 boids. A video of this simulation can be seen at http://www.youtube.com/watch?v=xM9DHHBqBCk.

Fig. 16. Simulation example with 20164 boids.

7 CONCLUSIONS AND FUTURE WORK

In this article we presented an extension of the boids algorithm to optimize neighbor-search by using visibility estimation. This extension was implemented on GPU using both GPGPU (shader) techniques and CUDA. The experimental results show that visibility culling can be very effective in reducing the total complexity in the simulation of large groups. When comparing different implementations of the algorithm, the CUDA version performed better than the GPGPU for grid construction. But considering the total time, the GPGPU version was significantly faster, mainly when our extension was included, even though the grid was constructed on the CPU. This means that the gain achieved by using the visibility estimation is bigger than the overhead for generating the grid on the CPU.

In the future, we would like to implement some optimizations such as using a linked list structure for cache access and using a 2D texture for encoding the grid structure instead of a 3D texture. We would especially like to develop a CUDA implementation using the much faster shared memory. On the algorithm side, we think that evaluating other strategies for occlusion estimation and to search for the nearest boids may provide more light on the subject.

ACKNOWLEDGMENTS

This work was partially supported by CNPq (National Council for Scientific and Technological Development - Brazil) and Fapemig (Research Support Foundation of the State of Minas Gerais - Brazil).

References

Breitbart, J. 2008. Case studies on GPU usage and data structure design. Master's thesis, Universität Kassel.

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), 233–240.

Cohen-Or, D., Chrysanthou, Y., and Silva, C. 2003. A survey of visibility for walkthrough applications. *IEEE Trans. Visualization and Computer Graphics 9*, 3 (July-Sept.), 412–431.

Courty, N. and Musse, S.R. 2005. Simulation of large crowds in emergency situations including gaseous phenomena. In *Proceedings of the Computer Graphics International* (CGI '05), IEEE, 206–212.

Drone, S. 2007. Real-time particle systems on the GPU in dynamic environments. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques* (SIGGRAPH'07), ACM, New York, 80–96.

Halfhill, T. 2008, Parallel Processing With Cuda. Microprocessor, In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, 89–97.

Harris, M. and Buck, I. 2005. *GPU Gems 2 – Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 547–555.

Klosowski, J.T. and Silva, C.T. 2000. The prioritized layered projection algorithm for visible set estimation. IEEE *IEEE Trans. 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 *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware,* ACM, New York, 123–131.

Marcolino, L.S. and Chaimowicz, L. 2008. No robot left behind: Coordination to overcome local minima in swarm navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1904–1909.

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 *Proceedings of the International Conference on Computer Graphics and Interactive Techniques,* ACM, New York, 896–907.

nVidia. 2008. nVidia CUDA programming guide 2.1.

nVidia Corp. 2007. nVidia CUDA computer unified device architecture programming guide. http://developer.nvidia.com/cuda.

Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Rger, 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.

Passos, E., Joselli, M., Zamith, M., Rocha, J., Clua, E., Montenegro, A., Conci, A., and Feijó, B. 2008. Supermassive crowd simulation on GPU based on emergent behavior. In *Proceedings of the Seventh Brazilian Symposium on Computer Games and Digital Entertainment* (SBGames'08), Sociedade Brasileira de Computação, SBC, 70–75.

Quinn, M.J., Metoyer, R., and Hunter-Zaworski, K. 2003. Parallel implementation of the social forces model. 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. Graphics 2*, 2, 91–108.

Reynolds, C.W. 2006. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, (Sandbox '06), ACM, New York, 113–121.

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

Shao, W. and Terzopoulos, D. 2005. Autonomous pedestrians. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (SCA '05), ACM, New York, 19–28.

Shreiner, D., Woo, M., Neider, J., and Davis, T. 2005. *OpenGL*(*R*) *Programming Guide*: *The Official Guide to Learning OpenGL*(*R*), Vers. 2 (5th ed.), Addison-Wesley, Reading, MA.

Sillion, F.X. 1995. A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. I*EEE Trans.Visualization Computer Graphics 1*, 3, 240– 254.

Thalmann, D. and Musse, S. 2007. *Crowd Simulation*, Springer, Berlin.

Terzopoulos, D., Tu, X., and Grzeszczuk, R. 1994. Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. *Artificial Life 1*, 4, 327–351.

Treuille, A., Cooper, S., and Popovi´c , Z. 2006. Continuum crowds. *ACM Trans. Graphics 25*, 3, 1160–1168. Received February 2009; accepted July 2009