Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance

Ugo Erra, Rosario De Chiara, Vittorio Scarano, Maurizio Tatafiore

ISISLab
Dipartimento di Informatica ed Applicazioni "R.M. Capocelli",
Università degli Studi di Salerno, 84081 Baronissi, Italy
Email: {ugoerr, dechiara, vitsca}@dia.unisa.it
m_tatafiore@yahoo.it

Abstract

In this work, we present a massive simulation of a behavioral model using graphics hardware. In particular we took a well established model to simulate complex flocks and we focused our attention on its implementation using techniques to manage efficiently large amount of data. Using the recent programmability of GPUs and recent extensions of computer graphics programming, we implemented on the graphics hardware the model capable of managing a large aggregate motion of birds in a virtual environment as well as to avoiding both static and dynamic obstacles. We demonstrated the effectiveness of our GPU implementation when compared with CPU according to recent trends that show graphics hardware capable of also working outside of its natural application field.

1 Introduction

Graphics hardware is a fascinating new field of research that gloves a new shine on the traditional number crunching computation. Due to large data computation in computer graphics, manufacturers have designed a stream architecture capable of high computational power. As a side effect there is a growing interest in using graphic boards also as general purpose stream processing engine [17]. The importance of this evolution has been emphasized by the name GPU (Graphics Processing Unit) given to this processor.

Part of research efforts have been focused on the simulation of complex models on GPUs. In fact, an accurate simulation shows two important aspects: 1) is computationally expensive and 2) manages a large aggregate of data. A GPU has the capability of running programs, called fragment programs, in highly efficient way as well as the capability to perform extremely fast read/write operations as stream data in memory locations arranged as textures. To take maximum advantage from GPU, the effort is devoted to a method to map the application as a stream process, hence the data must appear as stream data.

Of course, all the computational power available for free in consumer graphics boards comes with some "strings" attached, i.e. the complexity in using it for general purpose computations instead of 3d graphics rendering for which it has been designed for. Reviews show that the effort to use of GPUs for complex simulation pays back with high speedups of the GPU compared to the CPU.

We focus our attention on a distributed behavioral model of a flock. We aim to port this simulation on GPU by mapping this model as a stream computation. In nature every element of a flock uses a simple local behaviors to take a decision about what direction is to be followed. This decision is based on its natural behavior but also from simple information that are perceived from neighbors. The information about elements of flock are managed as stream data and all behaviors are implemented as fragment programs on a GPU. The CPU is only used to provide spatial sorting to GPU when needed and to manage scene rendering of flock.

In this work we show how to map a distributed behavioral model of a flock into GPU as a stream process and managing the data as stream data. Besides we describe how to use and improve the vector fields for obstacle avoidance using graphics hardware for linear interpolation. Finally, we discuss our implementation of a spatial data structure for

neighbor searching and we describe our heuristic to skip same updates of this structure.

This work is organized as follow: in Section 2 we present some related work about simulations on graphics hardware. Section 3 present a distributed behavioral model with the introduction the data structure for boid's local vision and the description of the mechanism for obstacle avoidance using graphics hardware. Section 4 describes our architecture of the behavioral model and how it was mapped on graphics hardware with some implementation details. Some final comments with the description of future work conclude the paper in Section 6.

2 Related works

In recent years non-graphics applications like simulations on GPUs are becoming more popular. Harris et al. [5] present a method for real-time visual simulation using an extension of cellular automata, known as coupled map lattice, that is useful to simulate various phenomenas as boiling.

Several works have been presented for fluid simulations solving Navier-Stokes equations on GPUs. Wei et al. [8] show a physically-based flow simulation which support complex boundary conditions running on GPUs. Boltz et al. [1] have also developed a conjugate-gradient solver for Navier-Stokes equations. Also Kruger [6] has presented the GPU implementation of several algebra operator used to solve that equations. All these works are all motivate to make simulation GPU driven.

Other works show how it is possible to use graphics hardware for non-graphics applications. In [7] Lengyel describes a method for robot motion planning using rasterization hardware. Recently several methods have been used for ray tracing computation as in [2] and [9].

The work of Reynolds is our starting point about the study of a distributed behavior model. This model simulates the complex aggregate motion of a flock of birds, a herd of land animals, or a bank of fishes. In [12], Reynolds focused his study about a model that is apparently plausible without considerations about how to manage efficiently large aggregate motion of elements. This work is similar to particle systems presented by [11] to simulate natural effects like fire, clouds, smoke, etc., the main difference is that every particle is completely independent and there is not interaction between them. Later on, Reynolds presented some results on the implementation of the model on a PlayStation 2 [14].

3 A behavior model

In this paper we will aim to simulate a flock of birds starting from Reynold's work [12], but our considerations can be applied to others scenarios too.

Every element in this group is called *boid* (the contraction of *birdoid*). Every boid has some limitations: it has a strictly local knowledge of the space it occupies and its knowledge comes from a simulated vision from its current position, in other words there is no centralized control. The flock takes its decisions in a totally distributed manner in order to obtaining a synchronized movement.

This distributed decisions mechanism is borrowed from nature. Indeed, Reynolds observes that, in nature, none of the creatures being part of a group has a full knowledge of the entire group. Hence the decisions must be taken by every single element, local perceptions of the world as well as from information that is perceived from its neighbors. The sum of all this elementary behaviors is usually deep enough to enable the flock to present the complex aggregate motion that we can see in nature. The keystone of the simulation of this model is the imitation of this distributed partial knowledge of the group. This distributed mechanism jointly with a reasonable simulation of the physics of flight produces a very natural behavior.

3.1 Boid definition

Every boid in the system is defined as follow: a set of associated parameters used for the simulation of the flight as mass, maximum speed, maximum acceleration, global position, the current speed and a view reference system used to represent the point of view of the boid. In this reference system we used the vectors *forward*, *side* and *up* to indicate the orientation of boid. Part of this information is constant and defined at the "birth" of the boid while another part is updated at every frame of the simulation; this means that the simulation is *discrete*.

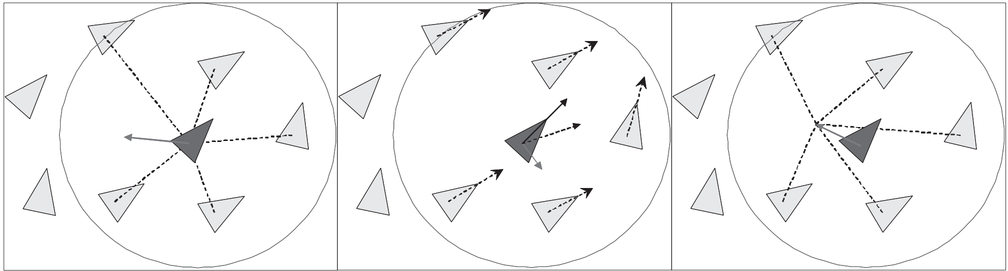

Figure 1: The three different types of steering behaviors. The first shows the *separation*, the boid tends get farer from the others nearby. The second shows the *alignment*, the boid tends to align itself to other boids nearby. The third shows the *cohesion*, the boid tends to stick together with the others nearby.

3.2 Local view

To support the local perception but also to manage a large set of boids we use a data structure (1) that allows to quickly obtain information about neighbors of every boid and (2) must be updated efficiently at every frame as the flock moves. The idea (similar to the one presented in [14]) is to use a simple space partitioning data structure in which boids are sorted in a regular cell grid, every cell keeps a list of boids that are flying in it. Given a boid, in constant time it is calculated in which cell it is flying and by exploring the adjacent cells, it is possible to gather information about boids around it. This stage is the only part of the simulation whose execution is still run on CPU because the sorting involves random access to memory for both reading and writing and these operations are currently not simultaneously available on graphics hardware.

The synchronized aggregated motion of the flock is achieved by fixing one or more spatial goals which the boids have to reach, these goals are the result of the sum of every boid's steering behaviors. The sum performed is weighted in order to give a characterization to every boid, a sort of *personality*. Every decision is taken considering a certain number of neighbors. This number is fixed to four in order to store all of them in one pixel of a texture. We have implemented the three different types of steering behavior presented by Reynolds in [13] and called *flocking behavior* (an intuitive representation of them is shown in Figure 1):

- the *separation* behavior tends to keep distance from other neighbors. This behavior is necessary to prevent boids collision. A repulsive force is calculated as the difference vector between current boid position and every neighbors while the steering force is calculated as the average vectors between all the repulsive forces.
- the *alignment* behavior tends to align the boid with other neighbors computing the steering force as difference between the average of the forward vectors of the neighbors and the forward vector of the boid itself.
- the *cohesion* behavior tends to move the boid toward the center of his local neighborhood. This behavior is useful in order to give to the flock a aggregated aspect. The steering force is obtained computing the average position of neighbors.

Another behavior that has been implemented in the simulation has its justification in the search for a better visual effect: the *leader following* behavior. This behavior constrains every boid to follow a fixed leader inside the flock, this leader can follow both a fixed path or a random path. Finally, since the flock can not move inside a scene infinitely large, we also implemented a *containment* behavior: it constrains every boid to remain inside a bounding box that surrounds the entire scene and hence constrains the flock to remain in the scene.

3.3 Obstacle avoidance

In the simulation we also considered an important aspect: the interaction with objects in the environment. An expected behavior is that a flock of boids

manages to avoid obstacles it meets on its own path. A possible collision adds new information in the knowledge of every boid that sees it and the model has to take it into account. One approach is to use the work of Egbert and Winkler [3] about the *force fields*. In this method a discrete force field surrounds every object present in the environment, approaching to an obstacle the forward vector is summed with the vectors of force field and the boid feels a growing opposing forces on its path towards it. Two problems arise with this solution: first, when the boid is flying perpendicularly to a wall the force field just modifies the module of speed not the direction. Second, a boid flying parallel to a wall feels the influence of force field even though it does not flying towards it.

For our purpose we improved the force field solution tackling the problem of perpendicular or parallel flight towards an object by using the force field. Using the dot product to compute the angle between the force field vector and forward vector we distinguish three case to calculate the steering force:

- $= 0$, the boid is flying parallel to an obstacle and nothing is done.
- > 0, the boid is flying away from an obstacle and we use as steering vector the force field vector.
- < 0, the boid is flying toward an obstacle and we use as steering force the sum of forward vector more vector force field.

Another problem that appears by using the force field solution is what we called "lack of time": it is the situation in which a boid may be safe at time i and at time $i + 1$ it may be inside a wall because of the approximations due to the discrete time simulation. To resolve this problem we try to foresee future positions in order to verify if a boid is on a collision route.

The force field is built for every object in the scene using the normal vectors of the objects itself. This field is built using a simplified geometry model because in the obstacles avoidance more than details is enough to use the shape of the objects. We store the force field as a three dimensional discrete vector field and linear interpolation is used to compute intermediate vectors. In fact we append addition layer outside the field with all vectors to zero, hence the linear interpolation permit to obtain values from a position where the field has maximum influence to another position where has no influence. This solution is particularly efficient when implemented on graphics hardware because vector fields are stored using three dimensional textures and intermediate vectors are computed using the highly efficient hardware linear interpolation. Besides this solution is a good framework to manage static and dynamic obstacles.

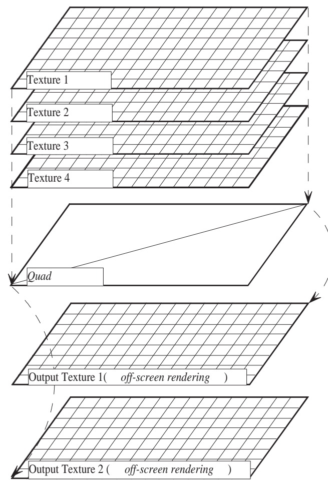

Figure 2: The streaming model on GPU. Our implementation uses a quad to drive data stream stored into textures, the result is one or more textures.

4 Mapping the behavioral model on the GPU

This model is suitable to be implemented on graphics hardware because every element can be managed as a single computation task and the decisions are taken in a distributed flavor and no centralized control is needed. Furthermore just a little amount of information must be maintained for every boid. We use the fragment processor to do all the necessary computations because it permits textures to be taken in input and writes the results in another texture.

For our purposes we adopt the model used by Harris [4] in which the geometry drives the processing. All the information about the boids is encoded into pixels of a texture, to force the fragment processor to do all the computations about one boid, we map all textures into a quad that has the same size

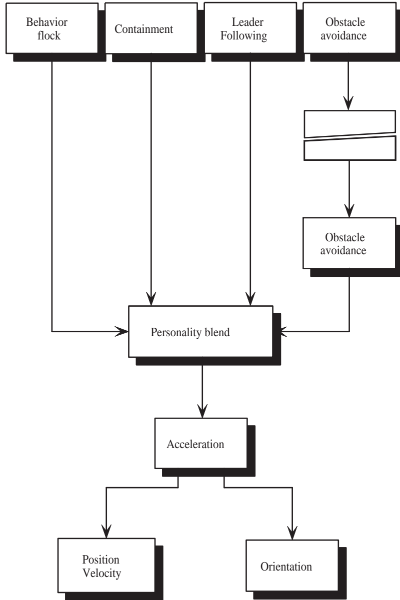

Figure 3: Architecture of behavior model on the GPU.

of frame buffer as shown in Figure 2. In this way the fragment program processes every boid fetching all data about it in only one execution. To avoid simultaneous rendering from one texture into itself not available on current hardware, we use a double-texture scheme. The data stored in the input texture, are used to update the data stored in the output texture. When the process is completed, we send the output texture to the CPU or to the next fragment program, and the textures are swapped for next rendering. The implementation of the entire process has been designed considering that every fragment program has a limit on the number of instructions as well as a limit on the number of register outputs hence the entire process is done using a multi pass scheme, the output of a fragment program is sent as input to the next fragment.

4.1 Implementation details

The entire process is shown in Figure 3. We implemented all modules as Cg fragment programs which are invoked using OpenGL Cg run-time functions described in the Cg toolkit user's manual [18]. To avoid overhead inside the GPU pipeline we devoted particular attention to switching textures and switching GL contexts between successive modules in the architecture.

The entire process takes as input four textures: 1) a constant texture T_c to store scalar information as mass, maximum velocity and maximum acceleration, 2) a texture T_o for orientation (three scalar values), 3) a texture T_p for position (three scalar values) and current velocity (one scalar value), 4) a texture T_n to store the four nearest boids.

We define the execution of a fragment program $[T_1, \ldots, T_n] \mapsto$ FRAGMENT $\mapsto [T]$ as the operation that binds the fragment, sets up the texture parameters $T_1,\ldots,T_n$ and draws the geometry onto the output texture T using off-screen rendering (called p-buffer). The entire process is executed in the following steps:

1. Prepare the input texture T_c, T_o, T_p, T_n
2. $[T_p, T_o] \mapsto$ CONTAINMENT $\mapsto [T_{s1}]$
3. $[T_p, T_o] \mapsto$ LEADER FOLLOWING $\mapsto [T_{s2}]$
4. $[T_p, T_o] \mapsto$ FLOCKING BEHAVIOR $\mapsto [T_{s3}]$
5. For every obstacle i with texture field T_{fi}
   - $[T_p, T_o, T_{s4}, T_{fi}] \mapsto$ OBSTACLE AVOIDANCE $\mapsto [T_{s4}]$
6. $[T_{s1}, T_{s2}, T_{s3}, T_{s4}] \mapsto$ PERSONALITY BLEND $\mapsto [T_s]$
7. $[T_c, T_s] \mapsto$ ACCELERATION $\mapsto [T_a]$
8. $[T_c, T_a] \mapsto$ POSITION VELOCITY $\mapsto [T_P]$
9. $[T_c, T_a] \mapsto$ ORIENTATION $\mapsto [T_O]$

The steps 2,3, 4 and 5 apply all the steering behaviors described in 3.2 producing four steering textures. In particular step 5 about obstacle avoidance is applied for all texture field objects present in the scene producing a steering vector as described in 3.3, the result of every iteration is summed with previous result and then reused for the next. All four steering textures are blended in step 6 using a weighted average with weights set up as uniform parameters. This texture is used in step 7 to compute the acceleration using the parameters defined in section 3.1. Then this force is used in last two fragments to compute the new positions T_P and new orientations T_O.

Textures T_P and T_O are sent back to the CPU to prepare the new geometry data of all boids to be rendered. This texture readback from GPU to CPU can impose performance penalties but new OpenGL extensions would eliminate in our architecture this drawback. The texture T_P is also used to update the space partitioning data structure and to compute a new texture about the four nearest boids.

4.2 Neighbor searching

The three behaviors that have been implemented in the personality of every boid are heavily influenced by the neighbors of such boid: this tries to mimick the limited knowledge of the flock that a boid has through its eyes. Every boid has in the parameters that defines its "life" a list of neighbors used in vicinity considerations.

This is known as one of the critical time-consuming phases of the model (see [14]). Being the path and the general shape of the flock both very dynamic, the calculation of the list of neighbors has to be done every frame and it is usually a very expensive operation.

In order to avoid the calculation of this list at every frame, a useful characteristics ia based on the observation of trying to avoid the calculation when a list neighbors of a boid is the same between the frames. We give here an intuitive explanation: when a boid flights uniformly respect on its neighbors, its list of neighbors remains the same frame by frame. At this point it is useful to consider that the calculation of the neighbors list is performed in a grid-based flavor: every boid considers as neighbors just the boids belonging to the cells around it.

Our heuristic is the following: at the beginning of every step of the simulation every boid knows which cell of the grid it belongs to, at the end of the step, after performing all the calculation regarding its behavior, the boid knows in which cell of the grid it will belong on the next step (both this information can be obtained at the cost of a floating point division). Every boid can express the relative variation of cell position with a triples of values, one for every spatial dimension, took from the set $[-1, 0, +1]$. At the beginning of every step a 3-dimensional matrix of 27 ($3 \times 3 \times 3$) values, called *scattering matrix* (see figure 4 (a)), is cleared and it is used to keep track of *the amount of boids that, at the end of the step, has performed a certain change of cell*, the intent of the matrix is to measure the *scattering* of the flock, indeed every flock adds 1 in the cell of the scattering matrix, whose coordinates are given by the triple the boid calculates.

What we expect is that, as long as the flock flies in a wide empty space it will keep a quite stable shape also keeping constant the neighbors lists of the boids, and this information is reflected in the scattering matrix through a large value in one cell and almost 0 in other cells. On the other hand, whenever the flock reaches the bounds of the space, or intercept an obstacle to avoid, the matrix will present a certain number of cells containing small values, meaning that boids are changing direction with high variability. A large value in a cell of matrix means that a lot of boids are changing cell (in the spatial subdivision matrix) in the same direction (see figure 4 (b)). A sparse scattering matrix means that boids are changing cell (once again in spatial subdivision matrix) in various directions (see figure 4 (c)) and this means that the neighbors lists have to be recalculated.

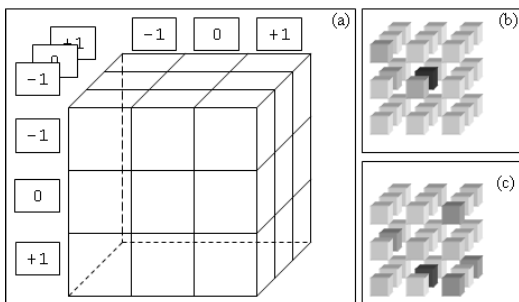

Figure 4: Scattering matrix: it is used to track down how much the flock has a uniform shape after each frame calculation. The cells are numbered with triple meaning indicating in which direction the boid is moving.

5 Results

We have experimented on an AMD Athlon XP 2000+ based machine with 1.67Ghz processor speed and 512Mb of RAM. The graphics card used was a GeForce FX 5800 with 128Mb of video memory, core speed of 300Mhz and memory speed of 600Mhz. For comparison we implemented both a CPU only version and a GPU based version of the behavioral model. The times showed in figures are averaged over 1000 steps.

The test scene used to render the flock is composed of a statically tessellated terrain, and six avoidable objects, each one with its vector field: four columns, a crossbeam and a moving sphere.

In Figure 5 the time weight of every phase of the frame is shown. We labeled with an asterisk the phases running on GPU. A result that came from

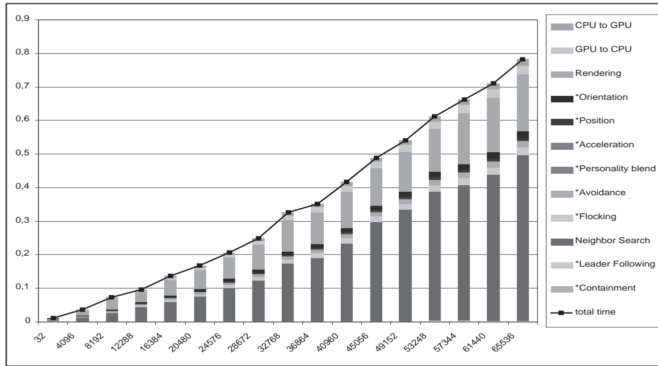

Figure 5: Total time composition. On the x-axis the number of boids. Time is expressed in seconds.

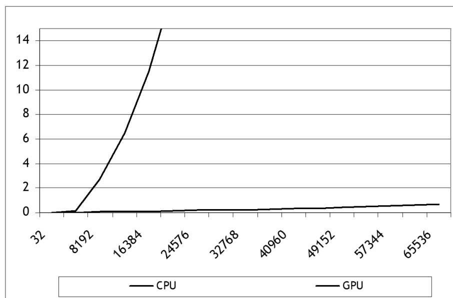

Figure 6: Performances of the two different implementations: CPU and GPU boosted. Time on the y-axis is expressed in seconds. The lower is better.

the analysis of the time distribution is that the heaviest phase is neighbors searching and that it yet runs on CPU. Something that can also be deducted, but a better analysis is given in Figure 6, is that the GPU boosted simulation scales well on increasing number of boids.

In Figure 6 we provided the performances comparison between the two implementations. The figure shows the number of seconds per iteration. It is clear that GPU implementation scales much better than CPU. This is an interesting point because we achieved this result in spite of the heavy phase of neighbors search yet running on CPU.

In Table 1 are reported the performances of the application running the GPU version expressed in *frames per second* (fps) which is a very common measure unit for graphics application. We would like to remark that fps is a measure prone to high variations depending on what is actually shown in the visualization. An interesting comparison should be done with results reported by Reynolds in [15, 16] where a performance of 60 fps was obtained with 300 boids. Our system animates 1600 boids at 60 fps with obstacle avoidance, but interactive frame rates, usually about 20 fps, can yet be obtained with more than 8000 boids.

# of boids	32	4096	16384	32768
FPS	115.29	30.07	8.01	3.45

Table 1: Frame per second performance. The higher is better.

6 Conclusions and future work

In this work we have shown:

- a method to map a distributed behavioral model onto GPU by organizing data as stream in order to manage them on GPU.
- the use of vector fields to manage obstacle avoidance improving the behavior of boids approaching an obstacle and how use the linear hardware interpolation to compute middle vectors.
- an heuristic to improve the update of spatial data structure.

No matter how much the GPU implementation beats the CPU implementation, we yet have a

time demanding phase kept on CPU, the neighbors search. We have planned further research related to the regular cell grid we used to partition the flock. The scattering matrix heuristic offered, in some very preliminary tests, promising results that deserves further investigations: in particular, we would like to find a schema in which every boid updates its position in the grid with no centralized coordination.

In general the searching for the k-nearest neighbors is a well studied problem and plenty of solution are folklore. What we find challenging is the implementation of one of the solutions on the GPU in order to take *the complete simulation* on GPU. In this context it seems interesting to consider the technique of using bitonic sort as shown in Purcell et al. [10] to sort boids into spatial structure.

Another scenario where we do expect great improvements is the use of new extensions of GPUs. In particular we can use displacement mapping into vertex program avoiding readback of output textures from GPU to CPU. Equally interesting could be the introduction of the novel PCI Express bus [19] which will provide bandwidth suitable to fast data movements between CPU and GPU.

References

[1] Bolz, J., Farmer, I., Grinspun, E., and Schroder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, SIGGRAPH 2003.

[2] Carr, N.A., Hall, J.D. and Hart, J.C. *The Ray Engine*, Graphics Hardware 2002.

[3] P. K. Egbert and S. H. Winkler, *Collision Free Object Movement Using Vector Fields*, IEEE Computer Graphics and Applications 1996, v.16 n.4, p.18-24, July 1996

[4] M. J. Harris, *Simulation and animation with hardware accelerated procedural textures*, Game Developers Conference 2003.

[5] M. J. Harris, Greg Coombe, Thorsten Schuermann and Anselmo Lastra, *Physically-Based Visual Simulation on Graphics Hardware*, Graphics Hardware 2002.

[6] Kruger, J., and Westermannm R. *Linear algebra operator for gpu implementation of numerical algorithms*, SIGGRAPH 1983.

[7] Lengyel, J., Reichert, M., Donald, B.R. and Greenberg, D.P. *Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*, Proceedings of SIGGRAPH 1990, p.327-335, August 1990.

[8] Wei Li, Zhe Fan, Xiaoming Wei and Arie Kaufman, *GPU-Based Simulation with Complex Boundaries*,Technical Report 031105, Computer Science Department, SUNY at Stony Brook, Nov 2003.

[9] Purcell, T.J., Buck, I., Mark, W.R. and Hanrahan, P. *Ray Tracing on Programmable Graphics Hardware*, ACM Transactions on Graphics, Volume 21, Issue 3, July 2002.

[10] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, *Photon mapping on programmable graphics hardware*, Graphics Hardware 2003.

[11] Reeves, W.T., *Particle System - A Technique for modeling a Class of Fuzzy Objects*, Proceedings of SIGGRAPH 1983.

[12] C. W. Reynolds, *Flocks, herds and schools: A distributed behavioral model*, SIGGRAPH, 1987.

[13] C. W. Reynolds, *Steering behaviors for autonomous characters*, Game Developers Conference (GDC), 1999.

[14] C. W. Reynolds, *Interaction with Groups of Autonomous Characters*, in the proceedings of Game Developers Conference 2000, CMP Game Media Group (formerly: Miller Freeman Game Group), San Francisco, California, pages 449-460, 2000.

[15] C. W. Reynolds, *Games Research: the Science of Interactive Entertainment*, SIGGRAPH, 2000, Course 39. Course description at `http://www.red3d.com/siggraph/2000/course39/`

[16] C. W. Reynolds, *Artificial Life for Computer Games*, Game Developers Conference, 2001, Course. Course description at `http://www.dgp.toronto.edu/~funge/gdc2001/`

[17] S. Venkatasubramanian, *The Graphics Card as a Stream Computer*, Proc. of the Workshop on Management and Processing of Data Streams, in cooperation with ACM SIGMOD/PODS and FCRF 2003, San Diego (USA), June 8 2003.

[18] NVIDIA Corporation, “Cg Toolkit User’s Manual”, Release 1.1, February 2003.

[19] http://www.pcisig.com/specifications/pciexpress