A New Architecture for Games and Simulations Using GPUs

Mark Joselli
Escola Politécnica
Pontifícia Universidade Católica do Paraná - PUCPR
Paraná, Brazil
mark.joselli@pucpr.br

Cristina Nader Vasconcelos, Esteban Clua
MediaLab, IC-UFF
Brazil

***Abstract*—Multi-thread architectures are the current trends for both PCs (multi-core CPUs and GPUs) and game consoles such as the Microsoft Xbox 360 and Sony Playstation 3. GPUs (Graphics Processing Units) have evolved into extremely powerful and flexible processors, allowing its use for processing different data. This advantage can be used in game development to optimize the game loop. As reported in the literature, GPGPUs have been used in processing some steps of the game loop, while most of the game logic is still processed by the CPU. This proposal differs by presenting an architecture designed to process practically the entire game loop using the GPU. Two test cases, a crowd simulation and a 2D game shooter prototype called GpuWars, are presented to illustrate the proposed architecture.**

Keywords—Digital Games, Game Architecture, GPGPU, Game Physics, Game AI, Flocking Boids

I. INTRODUCTION

The increasing levels of realism in digital games depend not only on the enhancement of modeling and rendering effects, but also on the improvement of different aspects such as animation, artificial intelligence of the characters and physics simulation.

Multi-thread architectures on PCs are becoming more and more popular with the development of multi-core processors and the new GPU architectures that can be used for generic processing. In addition, top of the line video game systems, like the Microsoft Xbox 360 and the Sony Playstation 3, feature multi-cores processors. Therefore, game architectures have to make their tasks parallel, demanding the adoption of concepts from parallel systems in order to take full advantage of the hardware. This work presents a game architecture that has been developed so that most of its tasks are executed in parallel, and the sequential execution used is kept to a minimum.

The development of programmable GPUs has opened new possibilities for general-purpose computation (GPGPU), which can be used, for instance, to enhance the level of realism in virtual simulations. Some examples in GPGPU that address these issues include Quantum Monte Carlos [1], finite state machines [2] and ray casting [3].

Games are interactive real-time systems, similar to multimedia applications, they have time constraints to execute its tasks in order to present to end users the results properly. In general, the main loop of a game falls into three categories:

(1) data acquisition, which gets data from user input, and can only be executed by the CPU;

(2) data processing, where the game logic is processed, and can be processed by the CPU or the GPU (with the corrective adaptation); and

(3) data presentation, where the results are presented to the end user though images and audio, which is processed by the GPU (images) and by the CPU (audio).

The main contribution of this work is an architecture model and its implementation that processes the entire game logic using the GPU.

Several games and previous works used GPGPU to process some selected subtasks on the GPU, while the remaining tasks are processed on the CPU. Such processing partition is frequently pointed out as the bottleneck in these simulations, because it may induce several expensive data transfers between the CPU and GPU [4]. This work implements all the methods of the game entirely on the GPU with the use of a new GPGPU architecture, keeping the GPU-CPU communication to a minimum. This work is particularly important as it proposes a new paradigm that can be used in GPUs and video games (Xbox 360 and Playstation 3), and also in future CPU architectures [5], where massively cores are expected to be available.

This paper is organized as follows: Section 2 presents a set of GPGPU concepts. Section 3 presents some related works on GPGPU that can be applied to games. Section 4 presents the architecture and section 5 presents the physics aspects of the architecture. Section 6 presents the game logic aspects of the architecture. Section 7 and 8 present the two test cases, a crowd simulation and the GpuWar game, respectively. Finally, Section 9 presents the conclusions and future works.

II. GPGPU

GPUs are powerful processors originally dedicated to graphics computation. They are composed of several parallel processors, achieving much better performance than modern CPUs in a number of application scenarios. An nVidia 8800

Ultra [6], for instance, can sustain a measured 384 GFLOPS/s against 35.3 GFLOPS/s for the 2.6 Ghz dual-core Intel Xeon 5150 [7].

The GPU architectures have been specially designed for processing tasks that require high arithmetic rates and data bandwidths. Because of the SIMD parallel architecture of GPUs, the development of this kind of applications requires a different programming paradigm than the traditional CPU sequential programming model. As an example, the nVidia GeForce 9800 GX2 [8] has 256 unified stream processors. In order to take advantage of the GPU processing power in a game, the developer needs to adapt the game tasks to this kind of parallel paradigm, like the architecture presented in this paper.

nVidia and AMD/ATI have implemented unified architectures in their GPUs. Each of them is associated with a specific language. nVidia has developed CUDA (Compute Unified Device Architecture) [9] and AMD has developed CAL (Compute Abstraction Layer) [10]. One main advantage of these languages is that they allow the use of the GPU in a more flexible way (both languages are based on the C language) without some of the traditional shader language limitations, such as scatter memory operations, i.e. indexed write array operations, and offer other features that are not even implemented in those languages, such as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts [11]. Nevertheless, the disadvantage of these software architectures is that they are vendor specific, i.e., CUDA only works on nVidia and CAL only works on AMD/ATI cards. In order to have GPGPU programs that work on both GPUs it is necessary to implement them in shader languages like GLSL (OpenGL Shading Language), HLSL (High Level Shader Language) or CG (C for Graphics) with all the vertex and pixel shader limitations and idiosyncrasies. According to the vendors, in the near future it will be possible to use OpenCL (Open Computing Language) [12], which is available for both nVidia and AMD graphics cards.

The GPGPU is getting more and more common and it is being applied in the geologic area [13], the medical area [14] and in computer vision [15]. The websites from CUDA [16] and gpgpu.org [17] show the latest development in the field.

III. Related Work

There are a lot of GPGPU applications in many fields. In the field of games GPGPU has been mostly concentrated in game physics. Game physics using GPU is a field of potential, and many previous works have achieved considerable speedup by moving the physics calculations from the CPU to the GPU. All the major physics engines for games available in the market had made, or are making, attempts to use the GPU to process its calculations. The work of Green [18] described an implementation on the GPU of some methods of the commercial physics engine called Havok FX, which was constructed to be a GPGPU version of Havok Physics [19]. The Havok FX was discontinued when Havok was bought by Intel, but there are rumors that it will be continued with the release of Intel new architecture for multi-core processing [20]. Several other examples can be found in the literature. PhysX of nVidia [21] is a physics engine that uses the CUDA architecture to optimize its calculation [22]. Bullet [23], an open source physics engine, is also investing in porting it to the GPU and has released demos with some aspects of the engine running on the GPU. Also in [24] a hybrid physics engine that has some of its calculations on the GPU was presented. Besides the physics engines, there are other works related to the implementation of physics simulation processes on the GPU like: particle system [25], deformable bodies system [26], fluids [27, 28] and collision detection [29].

Physics simulation works very well on the GPU because of the high performance of the stream processors, which allows high parallelism of the physics problems that can be solved in this structure. With that, it is possible to have faster physics simulation on games, and also more physics realistic games.

Another field that could be implemented in the GPGPU and can be used by games is game AI or game logic. This field has not been explored and there are very simple works on the field. There is an implementation of Finite State Machine on the GPU [2], but it only implemented very primitive behavior and cannot be used for games.

Another field that can be incorporated into a game that explores GPGPU is crowd simulation [30–36]. Crowd simulation can be used in games for simulating the behavior of herds [31, 35], people walking on streets [37], soldiers fighting in a battle [38], spectators watching a performance [39] and also to populate game worlds [30], like a GTA game [40]. These works are particularly important since they propose a simple AI model implementation into GPU architecture. This kind of simulation can also be implemented using the architecture proposed in this paper, as shown in the crowd simulation test case.

There are also some works that deal with the distribution of tasks between the CPU and GPU, like [24, 41–45]. These works concentrated on the GPU processing most of the physics tasks and distributing other tasks to the CPU. Even though these works presented some aspects of the game tasks processed by the GPU, the present work differs in a way that all game tasks are processed by the GPU.

Another research topic is related to the challenges for turning the game tasks parallel for multi-cores CPU, on which two works can be highlighted. The first presented the idea of a game loop, which is called Data Parallel Model [46]. In this model, the data is grouped in parallel sections of the application where they are processed. The Data Parallel Model proposed to use separate threads for sets of data (like game objects), instead of using a main loop with concurrent parts that process all data. The second approach presented the design of a parallel game engine for multi-cores Intel processors [5]. The main idea is to divide the data processing in tasks (like AI, physics, graphics and audio) and divide those tasks between the available cores. The presented work uses some similar concepts, like grouping the tasks according to the game object and dividing the tasks according to task itself.

The authors of this paper do not know any previous work in the literature that uses the GPU to process the entire game

logic, like the one presented in this work. An earlier version of this architecture can be seen in [47].

IV. The Architecture

Computer games are multimedia applications that require knowledge of many different fields, such as computer graphics, artificial intelligence, physics, networking and others [44]. Computer games are interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition in games is related to gathering data from input devices such as keyboards, mice and joysticks. Data processing tasks include applying game rules, responding to user commands, simulating physics and artificial intelligence behaviors. Data presentation tasks relate to providing feedback to the player about the current game state, usually through images and audio. In this architecture practically all game logic is processed in the GPU, i.e. all the data processing tasks, only using the CPU for tasks that need to make use of CPU, like data acquisition.

The game loops are the underlying structure that games and real time simulations are built upon. These loops are regarded as real-time because games and simulations (and similar kinds of multimedia applications) have time constraints to run the tasks that rely on them. This means that if those tasks do not run fast enough, the experience that the application must provide will be compromised.

Most GPGPU works use a sequential game loop called single-thread game loop with a GPGPU stage [24], which processes some of the data processing tasks of the game loop inside the GPU. Figure 1 illustrates this game loop. There are some variations of the same loop by putting the tasks in multi-threads [44].

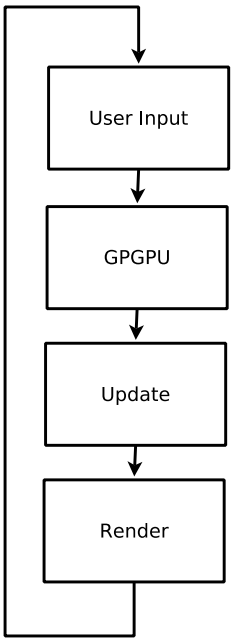

Fig. 1. Single-thread game loop with a GPGPU stage.

The game loop of this architecture differs from those works as it does not have an update stage in the CPU. This loop works as follows. First the CPU gathers the input and sends it to the GPU. The GPU processes this data, making the necessary adjustments, like the transformation of players' positions and the creation of the players' shots. The GPU starts updating the bodies by applying the physics behavior on them and their logic behavior, which corresponds to the artificial intelligence step. These updates are put in a VBO (Vertex Buffer Object) and sent to the shaders for rendering. The GPU also sends variables to the CPU in order to execute sound effects and to tell if it should terminate the application. This game loop is illustrated in Figure 2.

To resume, the CPU is responsible for:

- creating a window;
- gathering the users input and sending it to the GPU;
- making the GPU calls;
- executing the music and sound effects; and
- terminating the simulation/application/game, i.e., destroying the window and releasing the data.

And the GPU is responsible for:

- applying the physics on the bodies;
- processing the artificial intelligence;
- determining the game status, like the player scores; and
- determining the end of simulation/application/game.

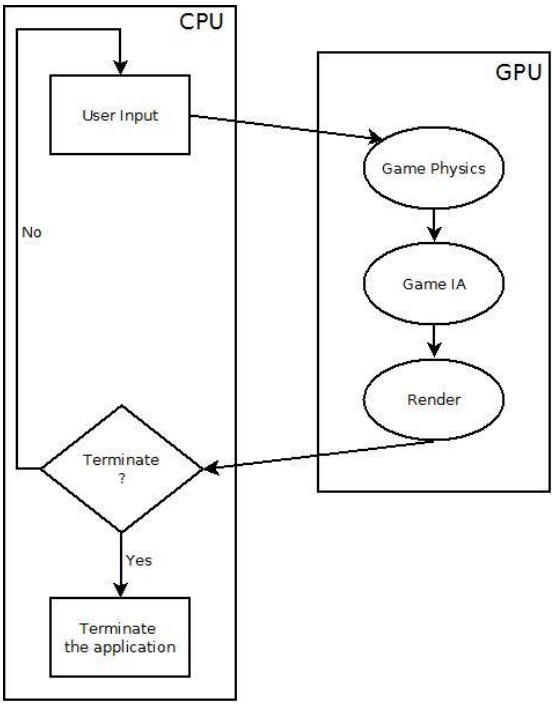

Fig. 2. Game loop of architecture.

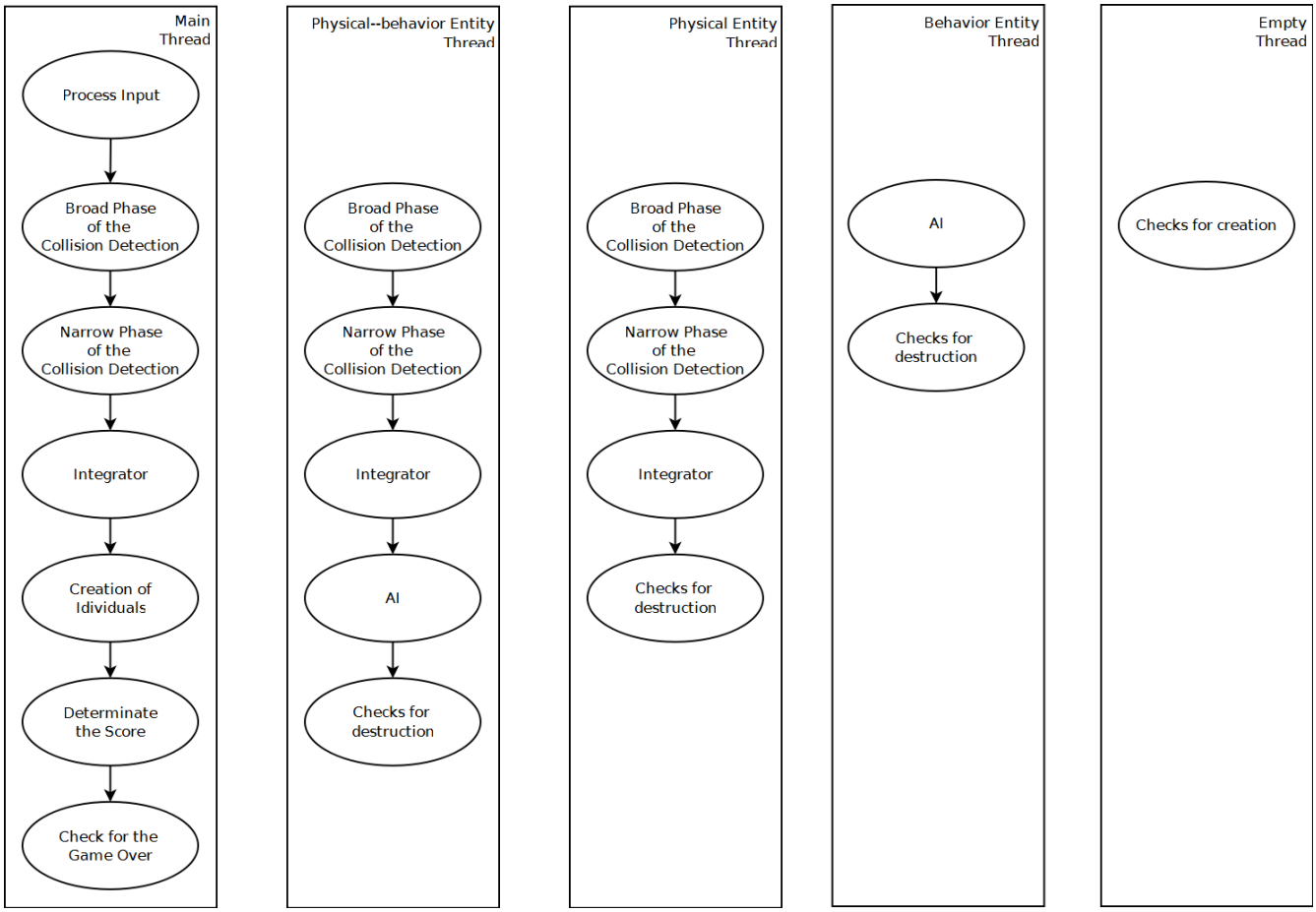

Fig. 3. The different processes of the architecture threads.

The data that is exchanged between the CPU and GPU is encapsulated in a special structure, in order to keep the communication between the CPU and GPU to a minimum, since this process can be a bottleneck of any simulation that requires communication between the CPU and GPU [48]. In order to implement this architecture some data structures are needed. The description of the data required for each entity follows:

- one vector (x, y, z) with the entity position;
- one vector (x, y, z) with the entity force;
- one vector (x, y, z) with the entity direction/orientation;
- one float as the entity type;
- one float with the entity energy; and
- one float for the entity mass.

The data is grouped into three vectors of size four in order to optimize the data exchange between GPU executions. To assure the desired high performance, all information, whenever possible, is organized and mapped as textures, using a ping-pong strategy.

Random numbers are used in games to avoid deterministic behavior. In the proposed architecture, random numbers are used to model random behavior of entities, including the creation of new entities, the initial status of these entities and the actions of the entities. Since GPUs do not have native pseudo random number generation, we developed a pseudo-random number generator based on a nVidia demo [49].

GPGPU programs are divided into threads. In order to process the main game logic that needs to be executed sequentially, the proposed architecture has a special GPU thread, which is responsible for it, and is the same that treats the user inputs. This processing includes tasks that updates the simulation according to the users input, i.e., threats the input; creates new entities, if necessary (which are created in other GPU threads); determines the scores (in case that the simulation is a game); determines the game over or the end of the simulation. The others threads are responsible for updating the entities, like collision detection and response, and the entities behavior. Even though some threads may wait for others threads to end, this approach shows an average speedup of 35 %.

There are three types of entities simulated by this work: physical entity, behavior entity and physical-behavior entity. The physical entity will only simulate the physical aspects of an entity related to collisions. The behavior entity will only

simulate the behavior of the entity and will not simulate collisions. And the physical-behavior entity will simulate both the physical and behavioral aspects.

The positions and type of an entity is gathered into a VBO (Vertex Buffer Object) and sent to a vertex shader for rendering without the use of the CPU. In order to deal with the creation of the entities, the architecture keeps a list with the values to indicate available positions for entities creation.

Using this structure, the GPU processes some empty threads (threads that practically do not process anything), and also different codes in different threads, which can affect the general performance, because of the threads synchronization mechanism inside the GPU block. In order to avoid this, the architecture proposed to group similar threads together into a GPU block, avoiding the loss in performance caused by thread synchronization. Figure 3 illustrates the processes of the different threads.

The proposed architecture was built in a way that it can be used, with proper modifications, for both 3D games and GPGPU particle simulations. It was implemented using the following technology: CUDA [9] for GPGPU processing; OpenGL for rendering; GLSL (OpenGL Shading Language) for shaders; and GLUT (OpenGL Utility Toolkit) for window creation and input gathering. But the concepts presented here could also be adapted to others technologies. In the next sections, the most important steps that are processed on the GPU, the physics step and the AI step, are present.

V. PHYSICS STEP

This step is responsible for the physics behavior, i.e., how the bodies process and resolve body collisions and responses. The physics of this architecture is based on the physics of particle systems [25, 50–52] and a hybrid physics engine [24].

Collision detection is a complex operation. For n bodies in a system, there must be a collision detection to check between the $O(n^2)$ pairs of bodies. Normally, to reduce this computation cost, this task is performed in two steps: first, the broad phase, and second, the narrow phase. In the broad phase, the collision library detects which bodies have a chance of colliding among themselves. In the narrow phase, a more refined collision algorithm are performed between the pairs of bodies that passed by the broad phase.

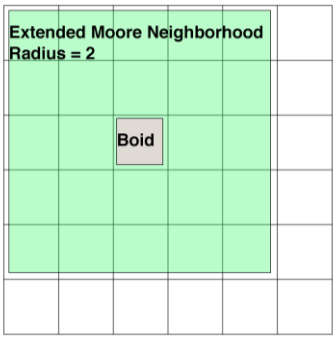

Fig. 4. The neighborhood matrix.

The physics step is responsible for:

- executing the broad phase of the collision detection;
- executing the narrow phase of the collision detection, by applying the collision test in each of the previously approved body pair; and
- forwarding the simulation step for each body by computing the new position and velocity according to the forces and the time step, i.e., solving the motion equations.

A. The Broad Phase

This phase is responsible for avoiding the n^2 comparison between all the entities, and also avoiding doing a narrow phase of the collision detection between the n^2 entities which is normally done by spatial hashing.

There are many ways to do a spatial hashing in the broad phase of collision detection. This work uses a uniform grid, which has a constant building cost, which makes the simulation more constant, and is highly suitable for the parallel structure of the GPU. This structure is also used in the AI step in order to determine the vision of the bodies.

This work has based its implementation on the neighborhood gathering method using neighborhood matrix which were presented in [31–34].

1) Neighborhood Matrix

The entity information is stored in matrices, where each index contains values for an individual entity. Since is possible to be required a variable number of data about the entities, it may be necessary to use more than one matrix. However it is a good practice to avoid wasting GPU memory by sharing vectors to store more than one single piece of information in each index.

The matrix containing the position vector for the entities is then used as a sorting structure. In Figure 4, one can see an example of such matrix where information about the position of 36 individual entities. To reduce the cost of proximity queries, each entity will have access to the ones surrounding its cells based on a given radius. In the example, the radius is 2, so the entity represented at cell (2, 2) would have access to its 24 surrounding entities only. In cellular automata, this form of information gathering is called extended Moore neighborhood [53].

This structure enables the exact prediction of the performance, since the number of proximity queries will be constant over the simulation. This happens because instead of making distance queries, taking as parameters all entities inside its own coarse Voronoi cell and the ones in the adjacent regions, as in traditional implementations, each entity would query only a fixed number of surrounding individual matrix cells. However, this matrix has to be sorted continually in such a way that neighbors in geometric space are stored in cells close to each other. This guarantees that this extension of cellular automata may gather information about close neighbors. For maintaining the grid aligned we use a bitonic sort [54], which makes a full sort in each dimension. The

bitonic sort is a simple parallel sorting algorithm that is very efficient when sorting small number of elements [55], which is our case since our sorting algorithm is applied to each dimension separately. Our implementation is an optimized and adapted version based on the previous work of nVidia [56]. More on this data structure can be seen in [32].

B. The Narrow Phase of Collision Detection

The narrow phase of the collision detection is responsible for doing the collision detection among the rigid bodies. In this work, instead of doing the collision check between all the polygons of the entities, it is implemented a basic primitive area element, that complex models are put inside. The bounds are used to surround every model, simplifying the narrow phase of the collision detection. Two types of bounds were implemented: a circle bound and a bounding rectangle. The circle bound is used whenever it is possible. This is done in order to save memory, since the circle bound only needs the position vector and a radius, while the bounding rectangle needs four variables.

C. The Integrator

This method is responsible for integrating the equations of motion of a rigid body [57]. In the proposed architecture, it consists of a simple formulation; since it does not take into account the angular velocities and torque. This method updates crowd entity velocity based on the forces that are applied to it, which are sent to the integrator, and then it updates the position based on its velocities, using an integration method based on Euler integration (this type of integration is one of the simplest forms of integration). Mathematically, it evaluates the derivative of a function at a certain time, and linearly extrapolate based on that derivative to the next time step.

VI. AI Step

Game Artificial Intelligence (AI) is used to produce the illusion of intelligence in the behavior of non-player characters (NPC), as, for instance, in the case of the test case GpuWars, of the enemies. The algorithms used for Game AI are typically built upon known methods from the AI field, but game AI focuses more on the gameplay instead of precision. Besides, game AI has more computational constraints than pure AI applications, since the game needs also to process the game physics and to render the results.

There are several ways to implement the game AI, such as Finite State Machines (FSM), fuzzy logic, neural networks, and many others [58]. This work uses Finite State Machine. FSMs are powerful tools used in many parts of computer games [59–61], like the NPC behavior, the characters animation states and the game menu states.

A FSM models structured behavior and is composed by states, the transitions between those states, and the actions. The architecture can be used to implements agent-based behaviors like finite state machines and crowd behavior.

The behaviors are affected by the size of vision (which uses the grid made by the broad phase of the collision detection), velocity, energy and type, which are variables available for each type of entity.

VII. Test Case I: Crowd Simulation

In order to validate and show the scalability of the architecture proposed in this work, we implemented the well known distributed simulation algorithm flocking boids (bird-like object) [62]. This algorithm was selected because of its good visual results, proximity to real world behavior observation of animals and understandability. The implementation of the flocking boids model using the architecture proposed enabled a real time simulation of up to 131 thousands animals of several species, with a corresponding visual feedback.

The test case simulated a crowd of animals interacting with each other and avoiding random obstacles around the space. This simulation can be used to represent from small bird flocks to huge and complex terrestrial animal groups or even thousands of thousands of different cells in a living system. Boids from the same type (representing the species) try to form groups and avoid staying close to other type of species.

To achieve a realistic simulation we try to mimic what is observed in nature. Many animal behaviors resemble that of state machines and cellular automata, where a combination of internal and external factors defines which actions are taken and how they are made. A state machine is used to decide which actions are taken. With this approach, internal state is represented by the boid type and external ones correspond to the visible neighbors, depending from where the boid is looking at (direction), and their relative distances.

Based on these ideas, our simulation algorithm uses internal and external states to compute these influences for each boid: flocking (grouping, repulsion and direction following); leader following; and other boid types repulsion (used also for obstacle avoidance). An illustration can be seen in Figure 5.

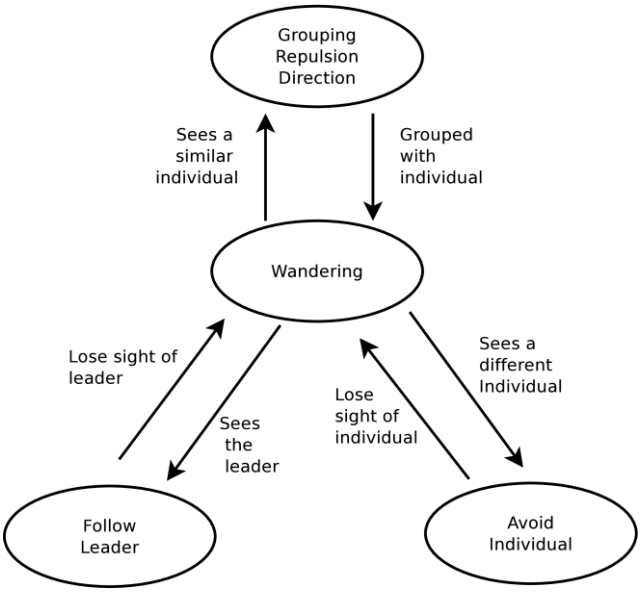

Fig. 5. The Boid state machine.

A. Results

Every boid was implemented as a physical-behavior entity, and it was render as a simple sphere. All the tests of this test case were made with an Intel quad-core 2.4 GHz with a nVidia Geforce 8800 GTS (which has 96 stream processors) running on Windows Vista. Each instance of the test ran for 300 seconds. The average time to compute a frame (and subsequent frames per second) was recorded for each experiment. To assure the results are consistent, each test was repeated 10 times and the standard deviation of the average times confirmed to be within 3%. A screenshot of the simulation can be seen in Figure 6.

Table 1 shows the results of the simulation in milliseconds and in frame per seconds, and Figure 7 show the evolution of the simulation with the different numbers of entities in the simulation.

This result shows that the architecture presents a good evolution, and can simulate and render up to 131 thousands interactive boids in real-time. A similar work of Reynolds [63] implements a similar algorithm, without the physics collision and response, in a Sony Playstation 3 architecture and can simulate and render up to 15 thousands interactive boids.

VIII. Test Case II: GpuWars Game

The GpuWars is a massive 2D game prototype shooter with a top-down 2D perspective. The game is similar to 2D shooters like Geometric Wars [64] and E4 [65]. The main enhancements of GpuWars is that it uses GPU to process its calculations, allowing to process and render thousands of enemies, while similar games only process hundreds.

The game play is very simple. The player plays as a GPU card (which is called "GPUShip") inside the "computer universe", and he needs to process (by shooting them) polygons, shaders and data (the enemies) from a game. Every time the "GPUShip" makes physical contact with an enemy it loses time and in consequence it loses FPS. The objective is to process the maximum number of data in the smaller amount of time, and keep the game interactive with a minimum 12 frames per second.

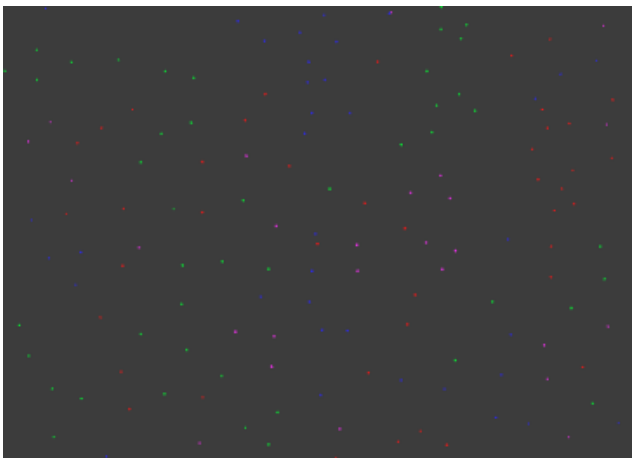

Fig. 6. A screenshot of the simulation.

TABLE I. Results of the crowd simulation test case

Number of bodies	Time GPU	GPU FPS
512	2.26	442
1024	2.69	371
2048	3.44	291
4096	4.57	219
8192	5.26	190
16384	8.20	122
32768	13.89	72
131032	55.56	18

The GpuWars uses the keyboard as the input device, where one set of controls are used to control the movement of the "GPUShip", and another set to control the direction of the shots. The shots are implemented as physical entities and the enemies are simulated as physical-behavior entities.

A. The GpuWars Game AI

This game implements 3 different behaviors using FSM: the kamikaze, the group and the tricky behaviors, which are present in the next subsections.

The behaviors are affected by the size of vision (which uses the grid made by the broad phase of the collision detection), velocity and energy (which are variables available for each type of enemy). By modifying these values, this work implements seven different types of enemies.

1) Kamikaze Behavior

The kamikaze approach is a behavior that simulates suicidal attacks. It is created by using a state machine that has four states: wandering, attacking, checking energy, and dead. It is illustrated in Figure 8.

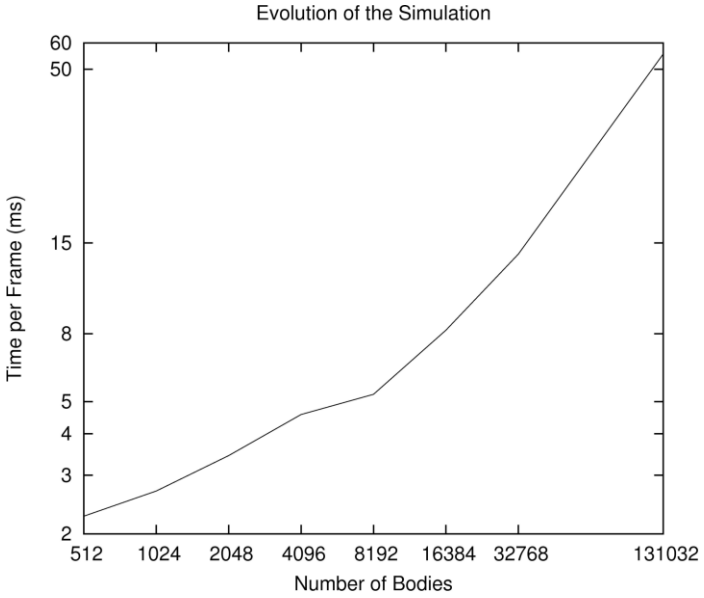

Fig. 7. Elapsed time of the simulation in milliseconds.

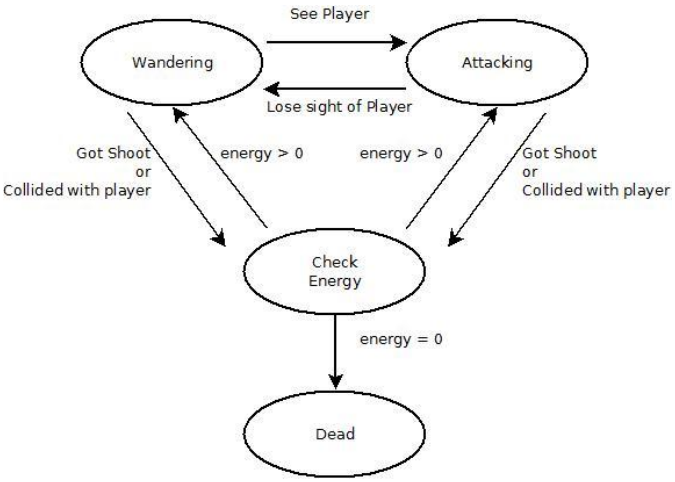

Fig. 8. The Kamikaze state machine.

The kamikaze is a very simple behavior. It wanders until it sees the "GPUShip", then it goes attacking it by throwing itself against it. This approach is well suited for GPU architecture, since little information about the scene is necessary.

2) Group Behavior

The group behavior creates a conduct pattern that makes groups, avoid bullets and attacks. It was modeled with a state machine that has six states: wandering, grouping, attacking, checking energy, avoiding bullets and dead, as shown in Figure 9.

This behavior is also very simple. The entity wanders trying to find similar entities, i.e., entities of the same type, and the "GPUShip". If it sees a similar entity, it goes closer to it and makes a group. In cases where it can see the player, it attacks the player by throwing itself against it. If the entity sees a bullet coming in its direction, it tries to avoid it.

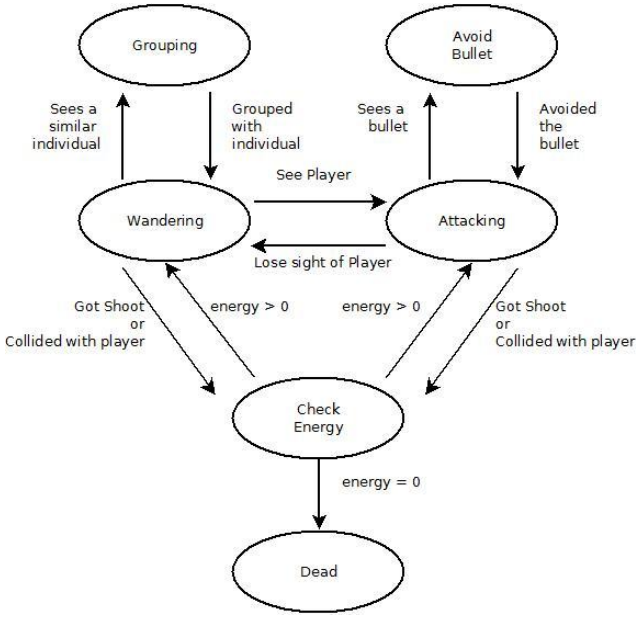

Fig. 9. The group state machine.

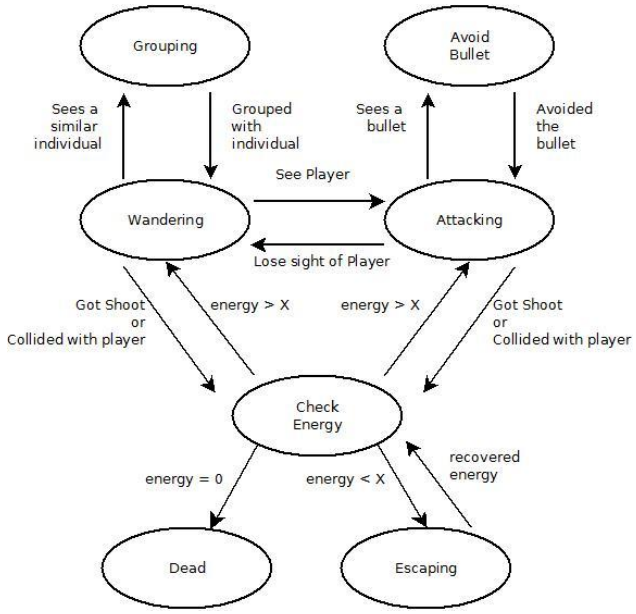

Fig. 10. The tricky state machine.

3) Tricky Behavior

The tricky behavior is the most complex behavior of the game. Similar to group behavior, this behavior also tries to group similar entities. For the tricky behavior, an entity may recover its energy. It has a state machine with seven states: wandering, grouping, attacking, avoiding bullets, checking energy, escaping, and dead, as shown in Figure 10.

This type of enemy wanders trying to find the "GPUShip" or similar entities. If it sees a similar individual, it goes closer to it and makes a group. If it is seeing the player, it throws itself against it. If the entity sees a bullet coming in its direction it tries to avoid it. If it has little energy it tries to escape from the player neighborhood to recover the lost energy.

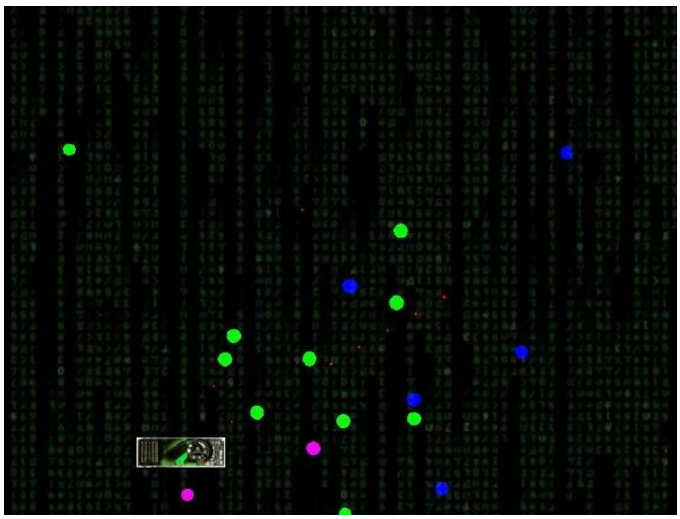

Fig. 11. A screenshot of the game.

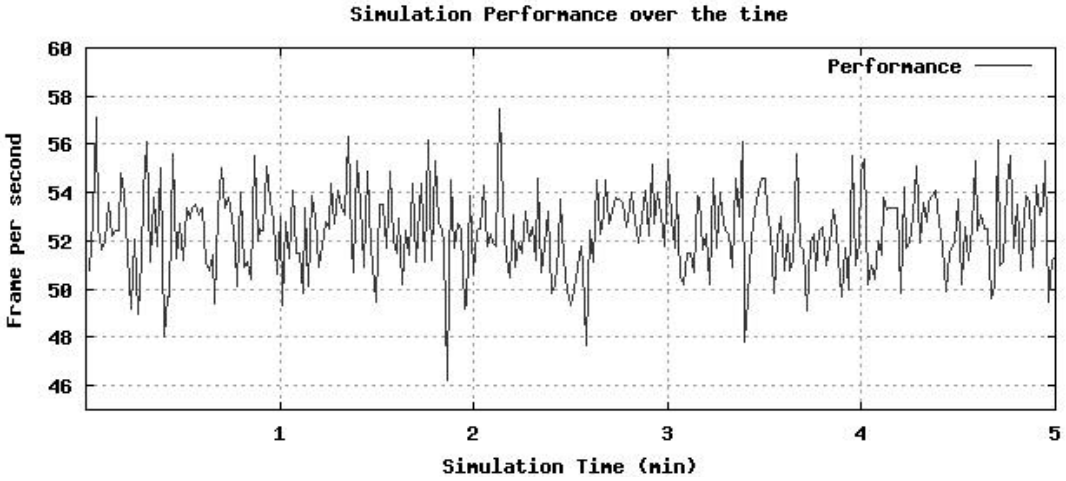

Fig. 12. Performance of the game.

B. Results

For the sake of repeatability, we ran the tests over the minimum hardware that can run CUDA. We used a notebook with an AMD Turion Dual-core, 3GB of RAM, with a nVidia GeForce mobile 8200M GPU card, which has only 8 stream processors, running on Windows Vista.

The number of enemies determines the performance of the game. This game has a maximum of 8192 enemies. A screenshot of the game can be seen of Figure 11. To better view the performance, Figure 12 shows a graph with the performance in FPS of the game for 5 minutes of the game. From this figure it can be seen that the performance of the game ranges from 45 to 58 frames per second (FPS). This performance is considered optimal in a game [43].

We have also tested the GpuWars game with the CPU determining the game status (players and enemies energies, and players scores) and the performance of the game has decreased to 30-40 FPS. Showing a bottleneck on the CPU's processing of the game status. This test also shows that the use of the architecture with all the game processing inside the GPU, can considerably speedup the game.

The game was also tested over a more powerful hardware, a quad-core with a nVidia GeForce 8800GS GPU card similar to the one used in the crowd test case, with similar results but with a speedup of three times (the FPS ranges from 130 to 170).

IX. Conclusions and Future Work

The development and evolution of multi-cores processors, GPUs and video games indicates that multi-thread architectures are a trend. Besides, the GPUs have evolved into more generic processors allowing them to be used to process different tasks of the game logic. Most works deals with some aspects of the game logic, with more focus on the game physics, and allowing the CPU to process others tasks, like the game AI. This work differs from the related GPGPU works, presenting an architecture that has all the game logic inside the GPU. The concepts of the proposed architecture could also be applied to others multi-cores processors like the Playstation 3, Xbox 360, and clusters. One drawback of the architecture is that, in order take full advantage of the GPU power, all game task must be implemented in GPGPUs kernels.

This work has implemented two tests cases using the proposed architecture: a crowd simulation, and a 2D game shooter. The crowd test case was a boid simulation which can achieve up to 131 thousands entities in interactive frame rate. This type of simulation can be applied to games, creating richer environments for the players.

The second test case is a design and implementation of the GpuWars game, which is a 2D shooter that has parallel tasks, AI, physics and game score, which are processed on the GPU. Also the concept of making the game logic parallel can be adapted for other multi-thread architectures. This can make a new trend on game development.

This architecture could be used for optimizing GTA [40] like games, by putting the simulation of pedestrian behavior and vehicles behavior on the GPU. It could also be used for the simulation of real time strategy game (RTS) characters behavior.

Future works will focus on creating more complex behavior for enemies, by implementing other game AI techniques, like hierarchical state machines, fuzzy logic and neural networks. Our plans are to proceed by evolving the proposed architecture, transforming it into a GPGPU game engine so that it can be used in other type of games.

References

[1] A. Anderson, W. G. III, and P. Schröder, "Quantum monte carlo on graphical processing units," Computer Physics Communications, vol. 177, no. 3, pp. 298–306, 2007.

[2] I. Rudomín, E. Millán, and B. Hernández, "Fragment shaders for agent animation using finite state machines," Simulation Modelling Practice and Theory, vol. 13, no. 8, pp. 741–751, 2005.

[3] C. Müller, M. Strengert, and T. Ertl, "Adaptive load balancing for raycasting of non-uniformly bricked volumes," Parallel Computing, vol. 33, no. 6, pp. 406–419, 2007.

[4] J. Krüeger, "A GPU framework for interactive simulation and rendering of fluid effects," 2008. [Online]. Available: http://www.sci.utah.edu/publications/krueger08/GPU_framework.pdf

[5] Intel, "Intel multi-core technology," 2009. [Online]. Avalible: http://www.intel.com/multi-core/

[6] nVidia, "Technical brief: nVidia GeForce 8800 GPU architecture overview," 2006. [Online]. Available: http://www.nvidia.com/page/8800_tech_briefs.html

[7] ——, "nVidia CUDA compute unified device architecture," Programming Guide, 2008.

[8] ——, "nVidia GeForce 9800 GX2 specification," 2009. [Online]. Avalible: http://www.nvidia.com/object/product_geforce_9800_gx2_us.html

[9] ——, "nVidia CUDA compute unified device architecture documentation version 2.2," 2009. [Online]. Avalible: http://developer.nvidia.com/object/cuda.html

[10] AMD, "AMD stream computing," 2008. [Online]. Avalible: http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf

[11] J. D. Owens, D. Leubke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," Computer Graphics Forum, vol. 26, no. 1, pp. 80–113, 2007.

[12] K. Group, "OpenCL - the open standard for parallel programming of heterogeneous systems," 2009. [Online]. Avalible: http://www.khronos.org/opencl/

[13] B. Kadlec, H. Tufo, and G. Dorn, "Knowledge-assisted visualization and segmentation of geologic features using implicit surfaces," IEEE Computer Graphics and Applications, vol. PP, no. 99, Preprint, 2009.

[14] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant, "Fast deformable registration on the GPU: a CUDA implementation of demons," in Proceedings of the 1st technical session on UnConventional High Performance Computing (UCHPC) in conjunction with the 6th International Conference on Computational Science and Its Applications (ICCSA), M. Gavrilova, O. Gervasi, A. Lagan, Y. Mun, and A. Iglesias, Eds., ICCSA 2008. Los Alamitos, California: IEEE Computer Society, 2008, pp. 223–233.

[15] TunaCode, "CUVIlib: CUDA vision and imaging library," 2010. [Online]. Avalible: http://www.cuvilib.com/

[16] nVidia, "CUDA zone," 2010. [Online]. Avalible: http://www.nvidia.com/object/ cuda_home_new.html

[17] M. Harris, "gpgpu.org," 2010. [Online]. Avalible: http://www.gpgpu.org, 2010.

[18] S. Green, "GPGPU physics," Siggraph07 GPGPU Tutorial, The 34th International Conference and Exhibition on Computer Graphics and Interactive Techniques, San Diego, California, USA, August 2007.

[19] Havok, "Havok physics," 2009. [Online]. Avalible: http://www.havok.com/content/view/17/30/

[20] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," ACM Transactions on Graphics, vol. 27, no. 3, 2008.

[21] nVidia, "nVidia physx," 2009. [Online]. Avalible: http://www.nvidia.com/object/nvidia_ physx.html

[22] M. Harris, "CUDA fluid simulation in nVidia physx," SIGGRAPH Asia 2009: Beyond Programmable Shading course, The 2nd ACM SIGGRAPH Conference and Exhibition in Asia, Yokohama, Japan, 2009.

[23] E. Coumans, "Bullet physics library," 2009. [Online]. Available: http://www.bulletphysics.com

[24] M. Joselli, E. Clua, A. Montenegro, A. Conci, and P. Pagliosa, "A new physics engine with automatic process distribution between CPU-GPU," in Proceedings of the 2008 ACM SIGGRAPH Symposium on Video games, 2008, pp. 149–156.

[25] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a GPU-based particle engine," in Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware, 2004, pp. 115–122.

[26] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on GPU," in Proceedings of Simulation and Visualization, 2005, pp. 247–258.

[27] J. R. da Silva Junior, E. W. G. Clua, A. Montenegro, M. Lage, M. de Andrade Dreux, M. Joselli, P. A. Pagliosa, and C. L. Kuryla, "A heterogeneous system based on GPU and multi-core CPU for real-time fluid and rigid body simulation," International Journal of Computational Fluid Dynamics, vol. 26, no. 3, pp. 193–204, 2012.

[28] J. R. da Silva Junior, M. Joselli, M. Zamith, M. Lage, E. Clua, and E. Soluri, "An architecture for real time fluid simulation using multiple GPUs," in Proceedings of SBGames, SBC, 2012.

[29] K. N. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "CULLIDE: interactive collision detection between complex models in large environments using graphics hardware," in Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware, 2003, pp. 25–32.

[30] J. Shopf, J. Barczak, C. Oat, and N. Tatarchuk, "March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU," ACM SIGGRAPH 2008: Advances in Real-Time Rendering in 3D Graphics and Games Course, New York, NY, USA, 2008, pp. 52–101.

[31] E. Passos, M. Joselli, M. Zamith, J. Rocha, A. Montenegro, E. Clua, A. Conci, and B. Feijo´, "Supermassive crowd simulation on GPU based on emergent behavior," in Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, 2008, pp. 81–86.

[32] E. B. Passos, M. Joselli, M. Zamith, E. W. G. Clua, A. Montenegro, A. Conci, and B. Feijo, "A bidimensional data structure and spatial optimization for supermassive crowd simulation on GPU," Computers in Entertainment (CIE), vol. 7, no. 4, p. 60, 2009.

[33] M. Joselli, E. B. Passos, M. Zamith, E. Clua, A. Montenegro, and B. Feijo, "A neighborhood grid data structure for massive 3d crowd simulation on GPU," in Proceedings, Brazilian Symposium on Games and Digital Entertainment, pp. 121–131, 2009.

[34] M. Joselli, E. B. Passos, J. R. S. Junior, M. Zamith, E. Clua, and E. Soluri, "A flocking boids simulation and optimization structure for mobile multicore architectures," in Proceedings of SBGames, 2012, pp. 83–92.

[35] A. R. Silva, W. S. Lages, and L. Chaimowicz, "Improving boids algorithm in GPU using estimated self occlusion," in Proceedings of SBGames '08 – VII Brazilian Symposium on Computer Games and Digital Entertainment, 2008, pp. 41–46.

[36] R. D. Chiara, U. Erra, V. Scarano, and M. Tatafiore, "Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance," in Proceedings of Vision, Modeling, and Visualization (VMV), 2004, pp. 233–240.

[37] J. van den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin, "Interactive navigation of multiple agents in crowded environments," in Proceedings of the 2008 Symposium on Interactive 3D graphics and games (I3D '08), ACM, New York, USA, 2008, pp. 139–147.

[38] X. Jin, C. C. L. Wang, S. Huang, and J. Xu, "Interactive control of real-time crowd navigation in virtual environment," in Proceedings of the 2007 ACM Symposium on Virtual reality software and technology (VRST '07), ACM, New York, NY, USA, 2007, pp. 109–112.

[39] nVidia, "Skinned instancing," 2008. [Online]. Avalible: http://developer.download.nvidia.com/SDK/10/direct3d/Source/SkinnedInstancing/doc/SkinnedInstancingWhitePaper.pdf

[40] R. G. North, "Grand theft auto IV, rockstar games," 2008. [Online]. Avalible: http://www.rockstargames.com/IV/

[41] M. Joselli, M. Zamith, L. Valente, E. W. G. Clua, A. Montenegro, A. Conci, B. Feijo´, M. Dornellas, R. Leal, and C. Pozzer, "Automatic dynamic task distribution between CPU and GPU for real-time systems," in IEEE Proceedings of the 11th International Conference on Computational Science and Engineering, 2008, pp. 48–55.

[42] M. Zamith, M. Joselli, L. Valente, E. Clua, A. Montenegro, R. C. P. Leal-Toledo, and B. Feijo, "A game loop architecture with automatic distribution of tasks and load balancing between processors," in Proceedings of SBGames 2009, pp. 5–8.

[43] M. Joselli, M. Zamith, L. Valente, E. W. G. Clua, A. Montenegro, A. Conci, and P. Feijo´, Pagliosa, "An adaptative game loop architecture

with automatic distribution of tasks between CPU and GPU," in Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment, 2009, pp. 115–120.

[44] M. Joselli, M. Zamith, L. Valente, E. W. G. Clua, A. Montenegro, R. Leal-Toledo, B. Feijó, and P. Pagliosa, "An architeture with automatic load balancing for real-time simulation and visualization systems," JCIS–Journal of Computational Interdisciplinary Sciences, Vol. 1, No. 3, pp. 207–224, 2010.

[45] M. Joselli, M. Zamith, E. W. G. Clua, A. Montenegro, R. C. P. Leal-Toledo, L. Valente, and B. Feijó, "An architecture with automatic load balancing and distribution for digital games," in Proceedings of 2010 Brazilian Symposium on Games and Digital Entertainment (SBGAMES), IEEE, 2010, pp. 59–70.

[46] V. Mönkkönen, "Multithreaded game engine architectures," 2006. [Online]. Available: http://www.gamasutra.com/features/20060906/monkkonen_01.shtml

[47] M. Joselli and E. Clua, "GpuWars: design and implementation of a GPGPU game," in Proceedings of 2009 VIII Brazilian Symposium on Games and Digital Entertainment (SBGAMES), IEEE, 2009, pp. 132–140.

[48] M. Joselli, J. Ricardo da Silva, M. Zamith, E. Clua, M. Pelegrino, and E. Mendonca, "Techniques for designing GPGPU games," in Proceedings of Games Innovation Conference (IGIC), 2012, pp. 1–5.

[49] V. Podlozhnyuk, "Parallel mersenne twister," 2007. [Online]. Avalible: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf

[50] nVidia, "CUDA particles," 2008. [Online]. Avalible: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/ particles/doc/particles.pdf

[51] Microsoft, "Advanced particles," SIGGRAPH 2007: Real-Time Rendering in 3D Graphics and Games course, ACM, 2007.

[52] M. Joselli, J. R. S. Junior, M. Zamith, E. Clua, and E. Soluri, "A novel data structure for particle system simulation based on GPU with the use of neighborhood grids," Proceedings of the GPU Computing Developer Forum 2012 (CSBC 2012 workshop), SBC, 2012.

[53] P. Sarkar, "A brief history of cellular automata," ACM Computing Surveys, vol. 32, no. 1, pp. 80–107, 2000.

[54] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the Spring Joint Computer Conference (AFIPS '68), New York, NY, USA, April 30–May 2, 1968, ACM, pp. 307–314.

[55] G. E. Blelloch, C. G. Plaxton, C. E. Leiserson, S. J. Smith, B. M. Maggs, and M. Zagha, "An experimental analysis of parallel sorting algorithms," Theory of Computing Systems, vol. 31, no. 2, pp. 135–167, 1998.

[56] nVidia, "Bitonic sort demo," Tech Report, 2007. [Online]. Available: http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html#bitonic

[57] D. H. Eberly, Game Physics. San Francisco, CA: Morgan Kaufmann Publishers, 2004.

[58] D. M. Bourg and G. Seemann, AI for Game Developers. Sebastopol, CA: O'Reilly Media, 2004.

[59] E. Dybsand, "A finite state machine class," in Game Programming Gems, M. Deloura, Eds. Hingham, MA: Charles River Media, 2000, pp. 237–248.

[60] J. R. Rankin and S. S. Vargas, "FPS extensions modelling ESGs," in Proceedings of the 2009 Second International Conferences on Advances in Computer-Human Interactions (ACHI '09), Washington, DC, USA, IEEE Computer Society, 2009, pp. 152–155.

[61] F. Li and R. J. Woodham, "Video analysis of hockey play in selected game situations," Image Vision Computing, vol. 27, no. 1-2, pp. 45–58, 2009.

[62] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87, New York, NY, USA, ACM, 1987, pp. 25–34.

[63] C. Reynolds, "Big fast crowds on ps3," in Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames, New York, NY, USA, ACM, 2006, pp. 113–121. [Online]. Available: http://doi.acm.org/10.1145/1183316.1183333

[64] B. Creations, "Geometry wars retro evolve," 2009. [Online]. Available: http://www.bizarrecreations.com/games/geometry_wars_retro_evolved/

[65] Q. E. Inc., "Every extend extra extreme," 2009. [Online]. Available: http://www.qentertainment.com/eng/2007/09/every_extend_extra_extreme.html

