GpuWars: Design and Implementation of a GPGPU Game

Mark Joselli
Jose Ricardo Silva Junior
Marcelo Zamith
Esteban Clua
MediaLab, IC-UFF

Eduardo Soluri
Nullpointer Tecnologia

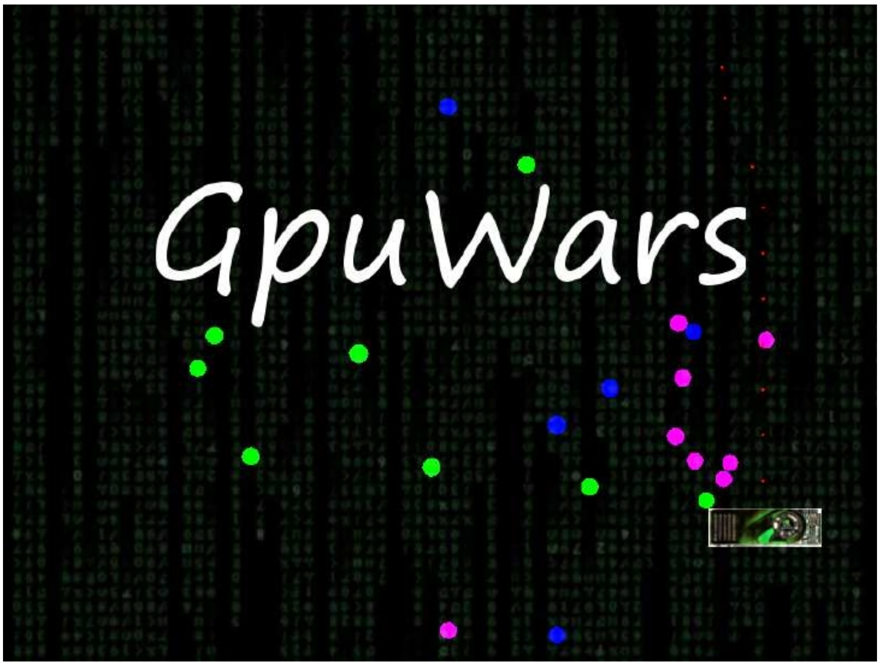

Figure 1: *Teaser of the GpuWars Game.*

Abstract

The GPUs (Graphics Processing Units) have evolved into extremely powerful and flexible processors, allowing its usage for processing different data. This advantage can be used in game development to optimize the game loop. Most GPGPU works deals only with some steps of the game loop, allowing to the CPU to process most of the game logic. This work differ from the traditional approach, by presenting and implementing practically the entire game loop inside the GPU. This is a big breakthrough on game development, since the CPUs are evolving to multi-core, and future games will need similar parallelism as the GPUs programs.

Keywords:: Digital Games, Game Architecture, GPGPU, Game Physics, Game AI

Author's Contact:

{mjoselli, jricardo, mzamith, esteban}@ic.uff.br
esoluri@nullpointer.com.br

1 Introduction

The increase of the level of realism in games depends 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.

Computers, new video game consoles (such as the Microsoft Xbox 360 and the Sony Playstation 3) and GPUs feature multi-core processors. For this reason, paralleling the game tasks is getting more and more important. This work has make a game with its tasks execution in parallel, with the sequential execution kept to a minimum.

A lot of games and works that uses GPGPU to process some parts of its tasks in the GPU and another on the CPU. This causes limitation on the simulation, because it requires a lot of data transfers between the CPU and GPU, and this can be the bottleneck of the simulation [Krueger 2008]. This work implements all the methods of the game entirely on the GPU with the use of CUDA architecture keeping the GPU-CPU communication to a minimum.

This work is particular important in order to present a paradigm that can be used in currently GPUs and video games (Xbox 360 and Playstation 3), but also in future CPU architectures [Intel 2009], where a massively cores are available.

The paper is organized as follows: Section 2 presents the GPGPU concepts. Section 3 presents some related works on GPGPU that can be applied to games. Section 4 presents the design of the GpuWars game. Section 5 presents the architecture and section 6 present the physics aspects of the architecture. Section 7 presents the game logic aspects of the architecture and section 8 presents the results. Finally section 9 presents the conclusions and future works.

2 Related Work

There are a lot of works that deals with the GPGPU field, but the application of these works on game fields are mostly concentrated on the game physics.

Physics on the GPGPU is a potential field and many works could achieve considerable speedup by taking the physics calculations from the CPU and processing on the GPU. All the major physics engine for games in the market has make, or is making, attempts to use of the GPU to process its calculations. The work of Green [Green 2007] presents an implementation on the GPU of some methods of the commercial physics engine called Havok FX which was being constructed to be a GPGPU version of Havok Physis [Havok 2009]. 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 GPU [Seiler et al. 2008]. Also the PhysX of nVidia [NVIDIA 2009b] is a physics engine that uses the CUDA architecture to optimizate its calculation [Harris 2009]. Also Bullet [Coumans 2009], an open source physics engine, is also investing in porting it to the GPU and has release some demos with some aspects of the engine running on the GPU. Also in [Joselli et al. 2008] a hybrid physics engine that has some of its calculations on the GPU is present. Besides the physics engines, there are other works related to the implementation of physics simulation processes on the GPU like: particle system [Kipfer et al. 2004], deformable bodies system [Georgii et al. 2005], and collision detection [Govindaraju et al. 2003].

Another field that could be implemented in the GPGPU and can be used by game is the game AI or game logic. This field is not very explored and there are very simple works on the field. There are implementation of finite state machine on the GPU [Rudomn et al. 2005], but this work implements very primitive behavior that cannot be used for games.

Another field that can be used for game that explores GPGPU is crowd simulation, like the works [Shopf et al. 2008; Passos et al. 2008; Chiara et al. 2004]. Crowd simulation can be used in games for simulating: the behavior of herbs of animals [Passos et al. 2008], people walking on the street [van den Berg et al. 2008], soldiers fighting in a battle [Jin et al. 2007], spectators watching a performance [nVidia 2008b] and also to populate game worlds [Shopf et al. 2008], like a GTA game [North 2008]. These works are particularly important since they propose a simple AI model implementation into a GPU architecture.

There are also some works that deals with the distribution of task between the CPU and GPU, like [Joselli et al. 2009]. These works concentrate on the GPU most the physics tasks of the game and these tasks can be distributed to the CPU. Even though these works presents some aspects of the game tasks inside the GPU, the present work differs from the latter, since it presents all the game tasks that needs to be processed developed inside the GPU.

There are no available work on the literature that use the GPU to process the entire game logic, like the one present in this work, just some tasks of the game.

3 The Design of the Game

The GpuWars is a massive 2D prototype shooter with a top-down 2D perspective. The game is similar to a 2D shooters like Geometric Wars [Creations 2009] and E4 [Inc. 2009]. 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" make physical contact with a enemy it looses time and in consequence it looses 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.

The GpuWars uses the keyboard as the input device of the game, one set of controls are used to control the movement of the "GPUship", and another set to control the direction of the shots.

4 The Architecture

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Network and others [Joselli et al. 2009]. More, computer games are also 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 as keyboards, mice and joysticks. Data processing tasks consist on 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.

This architecture was implemented using CUDA technology [nVidia 2009a] for GPGPU processing; OpenGL for rendering; GLSL (OpenGL Shading Language) for shaders; and GLUT (OpenGL Utility Toolkit) for window creation and input gathering.

The game loop of the GpuWars work as follows. First the CPU gather the input and sends it to the GPU. The GPU treat this data, making the necessary adjustments,i.e, the transformation of the player's position 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 on a VBO (Vertex Buffer Object) and sended to the shaders for rendering. The GPU also sends variables to the CPU in order to tell if it should terminate the application.

To resume, the CPU is responsible for: creating a window; gather the players input and send it to the GPU; make the GPU calls; execute the music and sound effects; and terminate the application, i.e, destroy the windows and release the data.

While the GPU is responsible for: applying the physics on the bodies; process the artificial intelligence; determinate the game status, like the player scores; and determinate the end of the game.

The data that is exchanged between the CPU and GPU is encapsulate in special structure, in order to keep the communication between the CPU and the GPU to a minimum, since this process can be a bottleneck of any simulation that has communication between CPU and GPU [Krueger 2008].

GPGPU programs are divided in threads. In order to process the main game logic which needs to be executed sequentially, the proposed architecture have a special CUDA thread which is responsible for it, and is the same that treats the "GpuShip" data and inputs. This processing includes: update the position of the "GpuShip" accordingly to the input; creation of shots, which are created in other CUDA threads; determinate the scores; determinate the game over; and determinate the creation of new enemies. The others threads are responsible for updating the enemies and the shots, like collision detection and response and the individuals behavior. The positions and type are put in a VBO and sent to a vertex shader in order to render the individuals without using the CPU. Also to deal with the creation of the shots and enemies, the architecture keeps a list with the values to indicate available positions for individuals creation. Using this structure the GPU processes some empty threads, threads that practically does not process anything, and also different codes in different threads, which can affect the performance because of the threads synchronization inside the CUDA block. In order to avoid this, the architecture groups similar threads together in a CUDA block, avoiding the lost in performance caused by the thread synchronization.

GPGPU programs does not have native pseudo random number generation. In order to fulfill that need this work developed a pseudo-random number generation based on nVidia demo [Podlozhnyuk 2007].

This architecture is build in a way that it can be also used, with proper modifications, in 3D games. In the next sections the most important steps there are processed on the GPU, the physics step and the AI step, are present.

5 Physics Step

This step is responsible for the physics behavior, i.e, how the bodies process and resolve all bodies collisions and response. The physics of this architecture is based on the physics on particle systems [nVidia 2008a; Microsoft 2007; Kipfer et al. 2004] and in a hybrid physics engine [Joselli et al. 2008].

Collision detection is a complex operation. For n bodies in a system, their must be a collision detection 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 algorithm to do the collisions tests are performed between the pairs of bodies that passed by the broad phase.

The physics step is responsible for: make the broad phase of the collision detection; Calculate the narrow phase of the collision detection, i.e, apply the collision in each body; and Forwarding the simulation step for each body by computing the new position and velocities according to the forces and the time step, i.e., integrating the equations of motion;

5.1 The broad phase

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

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

This work has based its implementation on the spatial hashing with sort of the nVidia particles demo [nVidia 2008a] and the CUDA broad phase implementation [Le Grand 2007]. This work differs from such implementation because it is adapted and optimize the structure and methods to be used with the GPGPU game loop process and to fill the requirements of the GpuWars game, which needs bigger grids and larger number of objects in the grid in order to be faster for the AI steps, which uses the grid to simulate the vision of the enemies.

This work implements also the neighborhood grid. That uses the position information of each entity to perform a lexicographical sort based on the two dimensions coordinate of the vector. The goal is to store in the bottom-left cell of the grid the entity with the smaller values for Y and X, and in the top-right cell the entity with highest values of Y and X respectively. Using these values to sort the matrix, the top lines will be filled with the entities with higher values of Y and the right columns will store those with higher values for X and so on. This kind of sorting provides data for the approximate neighborhood query, which is optimal in terms of data locality. More information on this structure can be see in [Passos et al. 2008].

5.2 The narrow phase of the 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 individuals, it is implemented a basic primitive area element, that complex models are put inside.

There are two types of bounds that this work implements, used to surround every model, simplifying the narrow phase of the collision detection: a circle bounds and a bounding rectangle. The circle bound is used whenever 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.

5.3 The Integrator

This method is responsible for integrating the equations of motion of a rigid body. In this work it consist on a simple step, since it does not takes into account the angular velocities and torque. This method updates crowd individual 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. Euler integration is one of the simplest form 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.

6 AI Step

Game AI is used to produce the illusion of intelligence in the behavior of non-player characters (NPC), and in the case of GpuWars, the enemies. There are a lot of ways to implement the game AI such as finite state machines, fuzzy logic, neural networks, and many others . This work uses finite state machine (FSM). Finite state Machines are powerful tools used in many computer game implementations [Rankin and Vargas 2009], like the NPC behavior, the characters animation states and the game menu states.

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. With the modification of these values, this work implements seven different types of enemies.

6.1 Kamikaze Behavior

The kamikaze approach is a behavior that simulates suicidal attacks. It uses a state machine that has only four state, wandering, attacking, checking energy and dead, and can be seen on figure 4.

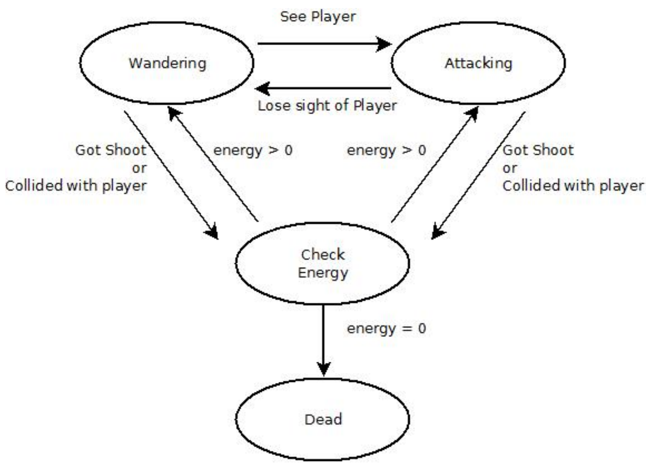

Figure 2: *The Kamikaze State Machine.*

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 a GPU architecture, since few information about the scene is necessary.

6.2 Group Behavior

The group behavior is a behavior that make groups, avoid bullets and attacks. It has a state machine that has six state, wandering, grouping, attacking, checking energy, avoiding bullets and dead.

This behavior is also very simple. The individual wanders trying to find similar individuals, i.e, individuals of the same type, and the "GPUShip". If it sees a similar individual, it goes close to it and make a group. And if it can see the player, it attacks the player by throwing itself against it. If the individual sees a bullet coming in its direction it tries to avoid it.

6.3 Tricky Behavior

The tricky behavior is the most complex behavior of the game. This behavior tries also groups similar individuals and it is the only that recoveries energy. It has a state machine that has seven states, wandering, grouping, attacking, avoiding bullets, checking energy, escaping and dead.

The enemy wanders trying to find the "GPUShip" or similar individuals. If it sees a similar individual, it goes close to it and make a group. If it is seeing the player, it throws itself against it. If the individual sees a bullet coming in its direction it tries to avoid it. If it has little energy it tries to scape to recover the lost energy.

7 Results

The number of enemies determines the performance of the game. This work has decided to have a maximum bound of 8192 enemies. A screenshot of the game can be seen of figure 1.

The game was tested with a quad-core with a nVidia GeForce 8800GS GPU card (which has 96 stream processors), with the FPS ranging from 130 to 170.

8 Conclusions and Future Work

The GPUs have evolved and can be used to process different tasks of the game loops. Most works deals with some aspects of the game loop, with more focus on the game physics. This work differ from the related GPGPU works, presenting a game that has all the game logic inside the GPU. This can make a new trend on game development.

Future works will focus on creating more complex behavior of enemies, by implementing other game AI techniques, like hierarchical state machines, fuzzy logic and neural networks. Also the authors will proceed by evolving the architecture so it can be used in other type of games.

References

CHIARA, R. D., ERRA, U., SCARANO, V., AND TATAFIORE, M. 2004. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *Vision, Modeling, and Visualization (VMV)*, 233–240.

COUMANS, E., 2009. Bullet physics library. Disponvel em: http://www.bulletphysics.com.

CREATIONS, B., 2009. Geometry wars retro evolve. Available at: http://www.bizarrecreations.com/games/geometry_wars_retro_evolved/.

GEORGII, J., ECHTLER, F., AND WESTERMANN, R. 2005. Interactive simulation of deformable bodies on gpu. In *Proceedings of Simulation and Visualization 2005*, 247–258.

GOVINDARAJU, K. N., REDON, S., LIN, M. C., AND MANOCHA, D. 2003. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, 25–32.

GREEN, S., 2007. Gpgpu physics. Siggraph07 GPGPU Tutorial.

HARRIS, M., 2009. Cuda fluid simulation in nvidia physx. Siggraph Asia 2009: Beyond Programmable Shading course.

HAVOK, 2009. Havok physics. Avalible at: http://www.havok.com/content/view/17/30/.

INC., Q. E., 2009. Every extend extra extreme. Available at: http://www.qentertainment.com/eng/2007/09/every_extend_extra_extreme.html.

INTEL, 2009. Intel multi-core technology. Avalible at: http://www.intel.com/multi-core/.

JIN, X., WANG, C. C. L., HUANG, S., AND XU, J. 2007. Interactive control of real-time crowd navigation in virtual environment. In *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, 109–112.

JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 149–156.

JOSELLI, M., ZAMITH, M., VALENTE, L., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., AND FEIJÓ, PAGLIOSA, P. 2009. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 115–120.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: a gpu-based particle engine. In *Graphics Hardware 2004*, 115–122.

KRUEGER, J. 2008. A gpu framework for interactive simulation and rendering of fluid effects. *IT - Information Technology 4*, (accepted).

LE GRAND, S. 2007. Broad-phase collision detection with cuda. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley Professional, August, ch. 32.

MICROSOFT, 2007. Advanced particles. Siggraph 2007: Real-Time Rendering in 3D Graphics and Games course.

NORTH, R. G., 2008. Grand theft auto iv, rockstar games. Avalible at: http://www.rockstargames.com/IV/.

NVIDIA, 2008. Cuda particles. Avalible at: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/particles/doc/particles.pdf.

NVIDIA, 2008. Skinned instancing. Avalible at: http://developer.download.nvidia.com/SDK/10/direct3d/Source/SkinnedInstancing/doc/SkinnedInstancingWhitePaper.pdf.

NVIDIA, 2009. Nvidia cuda compute unified device architecture documentation version 2.2. Avalible at: http://developer.nvidia.com/object/cuda.html.

NVIDIA, 2009. Nvidia physx. Avalible at: http://www.nvidia.com/object/nvidia_physx.html.

PASSOS, E., JOSELLI, M., ZAMITH, M., ROCHA, J., MONTENEGRO, A., CLUA, E., CONCI, A., AND FEIJÓ, B. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 81–86.

PODLOZHNYUK, V., 2007. Parallel mersenne twister. Avalible at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf.

RANKIN, J. R., AND VARGAS, S. S. 2009. Fps extensions modelling esgs. In *ACHI '09: Proceedings of the 2009 Second International Conferences on Advances in Computer-Human Interactions*, IEEE Computer Society, Washington, DC, USA, 152–155.

RUDOMN, T., MILLN, E., AND HERNNDEZ, B. 2005. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory 13(8)*, 741–751.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3.

SHOPF, J., BARCZAK, J., OAT, C., AND TATARCHUK, N. 2008. March of the froblins: simulation and rendering massive crowds of intelligent and detailed creatures on gpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, 52–101.

VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D., AND LIN, M. 2008. Interactive navigation of multiple agents in crowded environments. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 139–147.