# MASSIVE CROWD SIMULATION WITH PARALLEL PROCESSING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERDAL YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

FEBRUARY 2010

Approval of the Graduate School of Informatics

Prof.Dr.Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Asst.Prof.Dr.Tuğba Temizel Taşkaya
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

| | |
|---|---|
| Prof.Dr.Yasemin Yardımcı Çetin<br>Co-Supervisor | Assoc.Prof.Dr. Veysi İşler<br>Supervisor |

Examining Committee Members

| | | |
|---|---|---|
| Asst.Prof.Dr. Tolga Can | (METU, CENG) | ________________ |
| Assoc.Prof.Dr. Veysi İşler | (METU, CENG) | ________________ |
| Asst.Prof.Dr. Tolga Çapın | (BİLKENT, CS) | ________________ |
| Asst.Prof.Dr. Altan Koçyiğit | (METU, II) | ________________ |
| Asst.Prof.Dr. Alptekin Temizel | (METU, II) | ________________ |

## ABSTRACT

MASSIVE CROWD SIMULATION WITH
PARALLEL PROCESSING

Yılmaz, Erdal
Ph.D., Department of Information Systems
Supervisor: Assoc.Prof. Dr. Veysi İşler
Co-Supervisor: Prof. Dr. Yasemin Yardımcı Çetin

February 2010, 133 pages

This thesis analyzes how parallel processing with Graphics Processing Unit (GPU) could be used for massive crowd simulation, not only in terms of rendering but also the computational power that is required for realistic simulation. The extreme population in massive crowd simulation introduces an extra computational load, which is quite difficult to meet by using Central Processing Unit (CPU) resources only. The thesis shows the specific methods and approaches that maximize the throughput of GPU parallel computing, while using GPU as the main processor for massive crowd simulation.

The methodology introduced in this thesis makes it possible to simulate and visualize hundreds of thousands of virtual characters in real-time. In order to achieve two orders of magnitude speedups by using GPU parallel processing, various stream compaction and effective memory access approaches were employed.

To simulate crowd behavior, fuzzy logic functionality on the GPU has been implemented from scratch. This implementation is capable of computing more than half billion fuzzy inferences per second.

Keywords: Massive Crowd Simulation, CUDA, GPU Parallel Computing, Attribute Data Compaction

# ÖZ

PARALEL İŞLEM KULLANARAK
DEVASA KALABALIK BENZETİMİ

Yılmaz, Erdal
Doktora, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Veysi İşler
Ortak Tez Yöneticisi: Prof. Dr. Yasemin Yardımcı Çetin

Şubat 2010, 133 sayfa

Bu tez Grafik İşlemci Birimi kullanılarak yapılan paralel işlemlerin devasa kalabalık simülasyonu alanında kullanımını analiz etmekte olup, bu işlemi sadece grafik sunum açısından değil aynı zamanda gerçekçi benzetim için gerekli hesap gücü açısından da ele almaktadır. Devasa kalabalık benzetimlerinde kullanılan nüfusun sıradışı kalabalıklığı sadece Merkezi İşlem Birimi tarafından karşılanamayacak bir büyüklükte hesap yükü getirmektedir. Tez Grafik İşlemci Biriminin paralel hesap yeteneğinin devasa kalabalık simülasyonlarında ana işlemci olarak kullanılması esnasında faydayı arttırabilecek özel metot ve yaklaşımları göstermektedir.

Bu tezde tanıtılan metodoloji, yüzbinlerce sanal karakterin gerçek zamanlı olarak benzetiminin yapılmasına ve görselleştirilmesine olanak vermektedir. Grafik işlemci ile paralel işlem yaparak yüzlü rakamlar ile ifade edilen hızlanmaları sağlamak için, veri akımında indirgeme ve etkin hafıza erişimi yaklaşımları kullanılmıştır.

Kalabalık davranışının benzetimini yapmak amacıyla grafik işlemci üzerinde bulanık mantık uygulaması yapılmıştır. Bu uygulama saniyede yarım milyardan daha fazla sayıda bulanık mantık çıkarımı yapabilmektedir.

Anahtar Kelimeler: Devasa Kalabalık Benzetimi, CUDA, Grafik İşlemci ile Paralel Hesaplama, Öznitelik Veri Boyutu Küçültme

To Meltem and the twin boys.

# ACKNOWLEDGMENTS

The author is heartily thankful to the supervisors, Assoc. Prof. Dr. Veysi İşler and Prof. Dr. Yasemin Yardımcı Çetin who were abundantly helpful and offered endless assistance and guidance through the duration of this thesis. Their contribution to the author's vision and knowledge is beyond the content of this study. This thesis would not have been possible without their experience and supervision.

Special thanks to Can Ölçek, Eray Molla and Cansın Yıldız for the realization of archery physics, design and coding of fuzzy XML script. The author is also indebted to many friends for their support; Can Ardıç and Danielle Pooler for proof reading, Fatih Nar and Yasemin Çetin for contributions, Mehmet Erbaş and Hakan Şahin for building setups.

The author wishes to express his deepest love and sincere gratitude to Meltem for her endless support and understanding. By taking control of everything, she gave the time to complete this study. Lastly, the author offers his sincere apologies to his twins, Deniz and Demir for not spending enough time with them through the duration of this thesis.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Real-time simulation of massive crowds has always been a challenge, due to the limited computational resources. Such applications usually include virtual characters that interact with the environment and the other characters. Performing this amount of interaction in each simulation cycle is beyond the computing limits of commercially on-the-shelf CPUs and traditional serial programming techniques. Thus, massive crowds are used extensively on pre-rendered work such as movies, rather than real-time applications.

The level of realism and the quantity of the virtual population are among the major parameters that define this simulation's complexity. However, generating a life-like virtual environment with millions of inhabitants is rather a challenging task. Popular computer graphics techniques such as visibility culling and Levels of Detail (LOD) are usually employed to minimize computational complexity. On the other hand, these computational load reduction techniques cannot always be employed, as in the case of non-graphics simulation. In simulations where visualization is not a concern, it is usually required to update every virtual character in each simulation cycle without considering visibility or LOD. In such a case, alternative computational methods should be employed. Thanks to the state of the art GPUs, there is now commodity hardware, providing peak performances more than teraflops per second [1], and such a tremendous computational resource certainly helps achieve the goal of simulating massive crowds in real-time.

General purpose parallel programming can use GPUs not only for graphics but also for removing the burden of the non-graphical computational workload, which is traditionally handled by a CPU. Significant computational speedups have been achieved by various researchers from different disciplines using general purpose parallel programming [2-6]. Although GPU-based non-graphics computation is well suited to data-parallel tasks such as image processing kernels and matrix operations, it is also possible to accelerate many other applications by adapting existing algorithms to the general purpose parallel programming. Therefore it seems reasonable to exploit tremendous computing power of GPUs for massive crowd simulation, since computational power is an important concern. Additionally, data level parallelism, which is a must for efficient GPU implementation, can be achieved by assigning one thread per virtual character.

## 1.1 Motivation

This research addresses the possibility of using affordable parallel processing hardware to simulate massive crowds in real-time. The term "massive crowds" implies populations of up to several millions of virtual characters. Such a large number of agents is not common in real-time CPU-based serial programmed graphical applications, due to the limited computational resources. A typical crowd simulation application with graphics needs to share limited computational power for various tasks, such as rendering, behavior modeling and navigation. Computations get more complicated as the population and the level of realism increase. Considering these facts, real-time simulation and rendering of a large number of virtual characters usually require sacrificing some components such as visual quality or behavioral realism.

In the recent years, GPUs have demonstrated great progress as a non-graphics computing tool. It has already been shown that even two orders of magnitude speedups are possible, if the GPU takes computing responsibility [7,8]. Although the graphics processing architecture of the GPU is well suited to data-parallel applications, researchers from various disciplines have succeeded to port many different problems to this newly emerged programming environment to achieve significant computational speedups.

The announcements of immense computational speedups [7,8] and fascinating developments in GPU hardware inspired the use of general purpose parallel processing in massive crowd simulation. In literature, there exists limited work about the simulation and visualization of a large number of virtual characters using GPU parallel processing. However, there is significant amount of research regarding crowd simulation and parallel processing in different disciplines. This study considers adapting existing solutions in crowd simulation to general purpose parallel programming. Aspects of this research will introduce several practical approaches that can be helpful for future researchers in this field.

## 1.2 Scope

This thesis covers how general purpose parallel programming and the computational power of the GPUs can be used effectively to add more virtual characters in real-time applications by blending existing crowd simulation work and general purpose parallel programming techniques.

The issues mentioned in this thesis are explained by several case studies. The results look promising from the perspective of computational performance. The main scope of this thesis is crowd simulation, not crowd visualization; therefore, rendering quality and visualization related issues were not covered in detail. All of the visual outputs were given, to better explain the results of the case studies. However, it is certain that, higher visual quality can be achieved by spending more effort on graphics and by using a third party professional game-engine tool.

NVidia CUDA (Compute Unified Device Architecture) technology [9-11] was used in this thesis as the general purpose parallel programming tool. This technology lets software developers use C programming language to program the GPU, thus minimizing development platform learning curve. NVidia also plans to use CUDA with several other languages and application programming interfaces [9]. This technology has been available to the public since 2007. Starting with GeForce 8 series, all the NVidia GPUs are CUDA-enabled [10].

The following figures illustrate how general purpose parallel programming can help achieve the goal of simulating and rendering massive crowds in real-time. Figure 1 depicts a medieval combat scene. Figure 2 is related to massive crowds in video games. Figure 3 and Figure 4 show two screenshots that illustrate urban life respectively. The first figure depicts a special marathon event, while the second one shows a regular day.

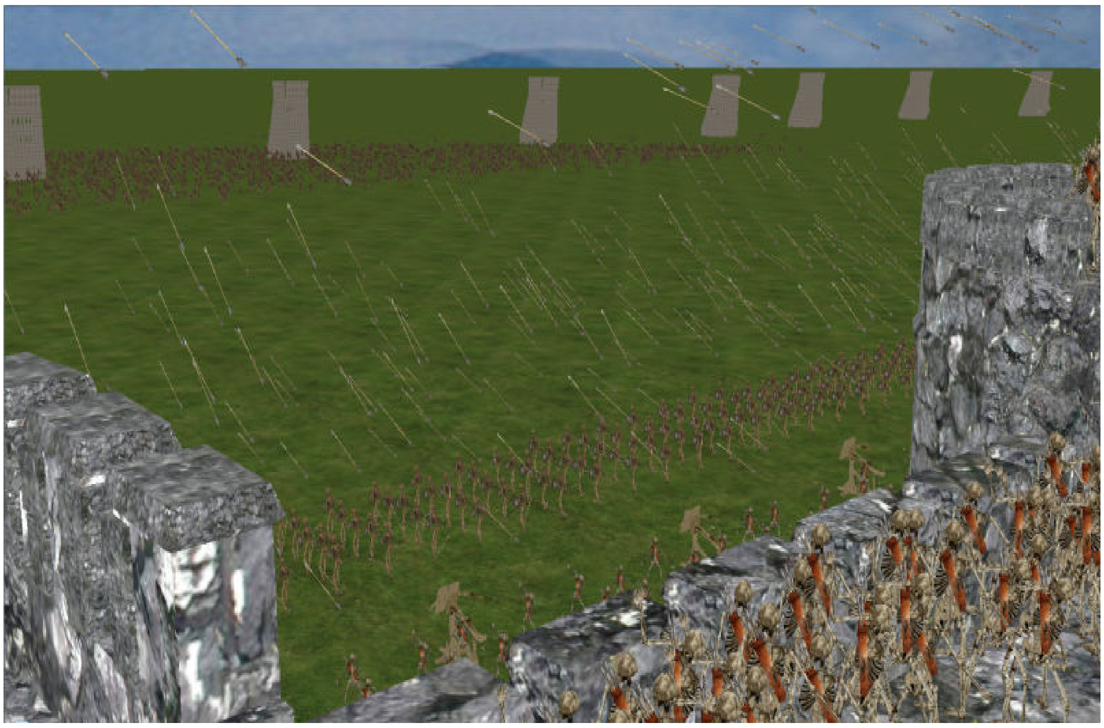


Figure 1: A crowded medieval-era combat scene.

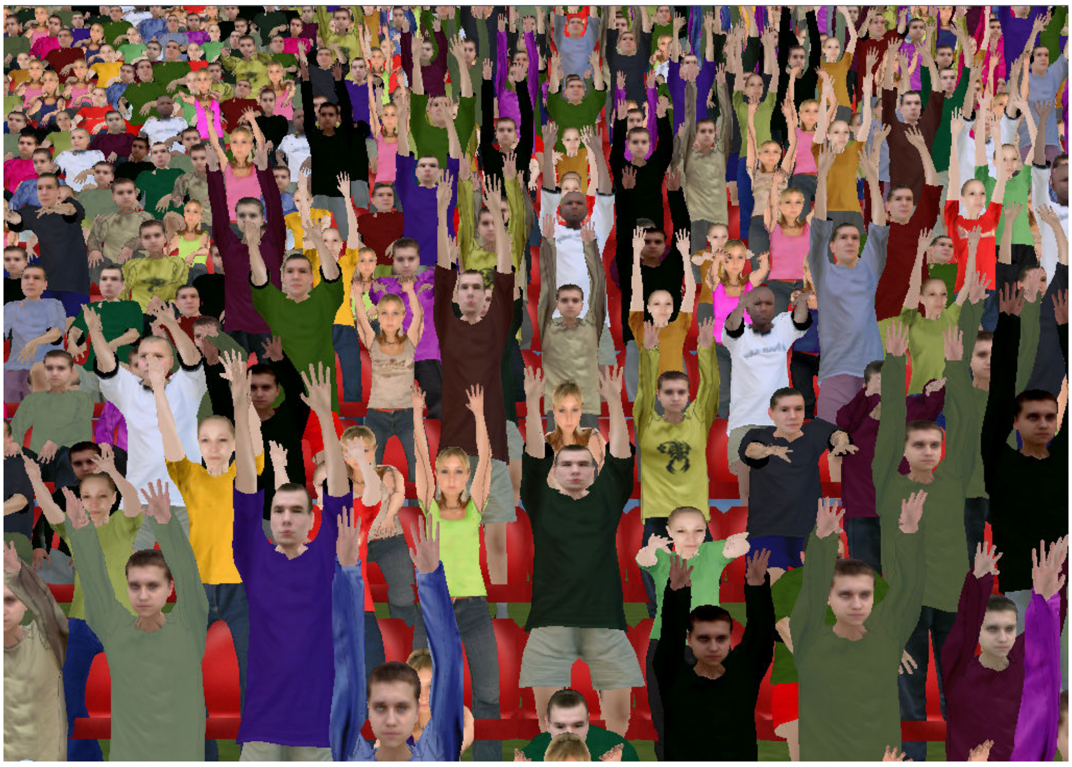


Figure 2: A crowded soccer arena scene.

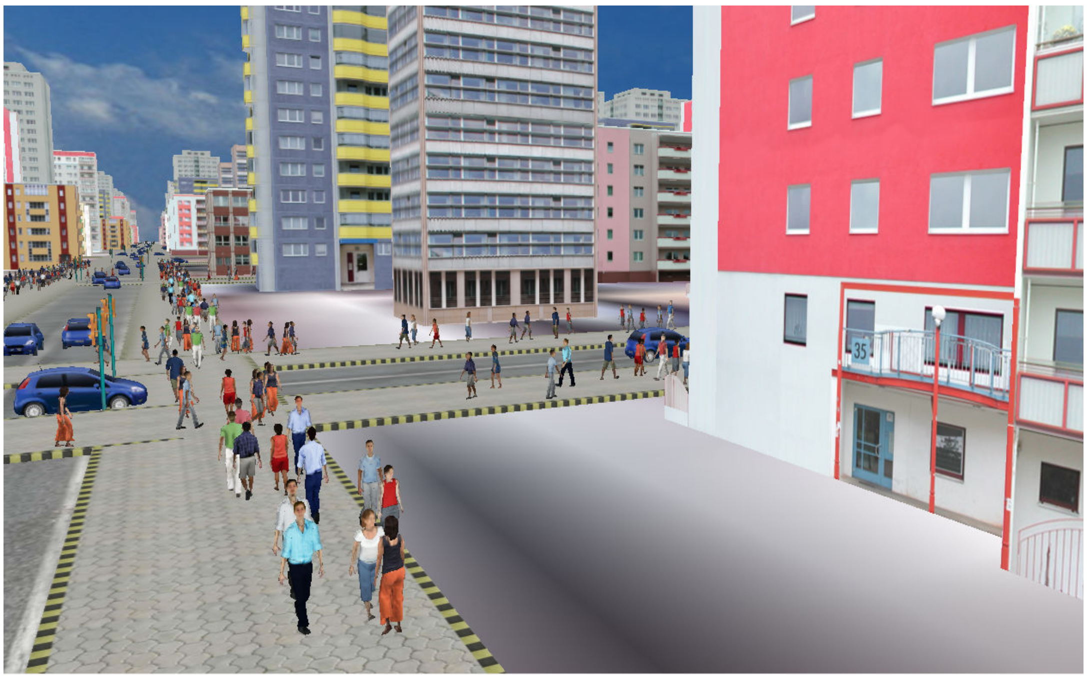


Figure 3: A crowded virtual city scene.

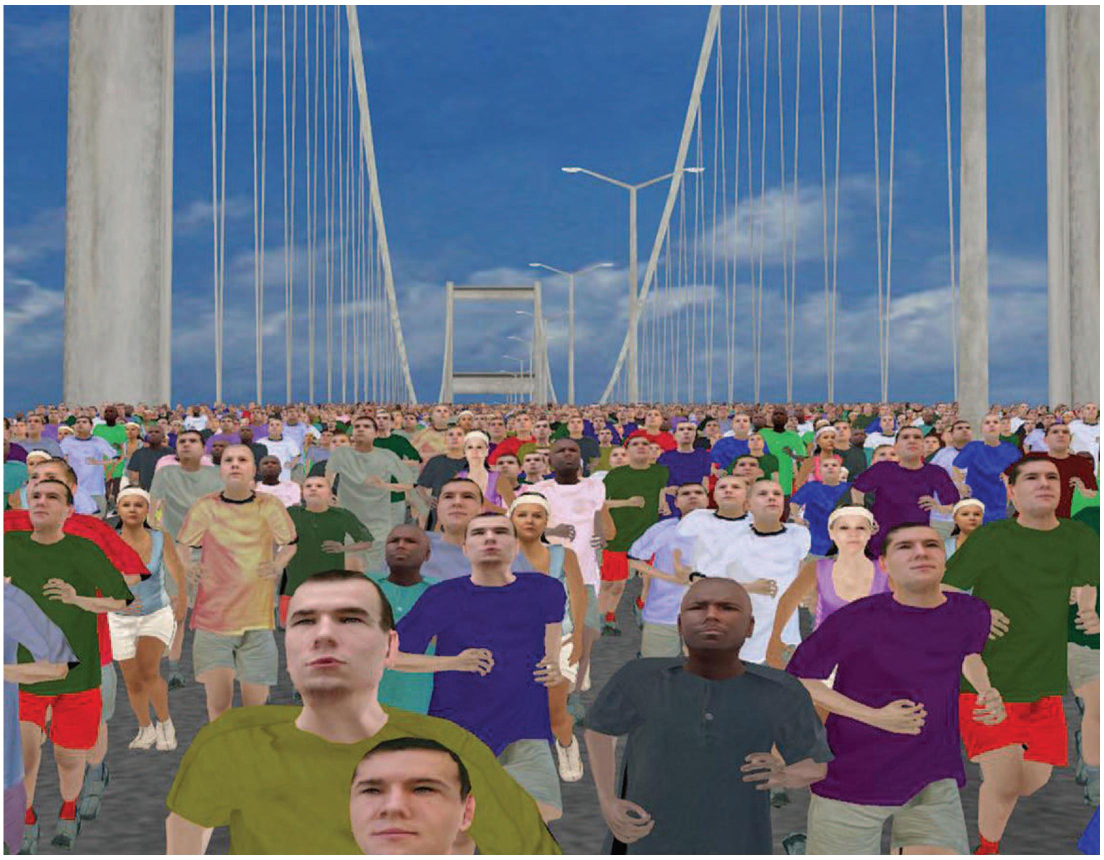


Figure 4: A crowded virtual marathon scene.

## 1.3 Significance

Most of the real-time crowd simulation research use limited populations to simulate and visualize virtual environments. The studies that try to visualize larger populations typically focus on visualization issues and rendering performance. Thus, the traditional approach usually deals with the army of clones that perform similar actions and movements. To the best of the author's knowledge, little attention has been paid to simulate massive populations in real time without sacrificing visual quality, behavioral variety or other computationally complex actions. The significance of this thesis is to bridge the gap between massive crowd simulation and the real-time constraints.

## 1.4 Contributions

This thesis addresses real-time massive crowd simulation using NVidia CUDA technology. Although this technology provides the computational power required by power hungry applications, special care must be paid while using this technology. The real-time constraints require using computational resources efficiently. In order to get the full benefit from general purpose parallel programming technology, the problem must be converted into data-parallel structure. It is also required to minimize data transfer between the CPU and the GPU. There are also several more issues that should also be considered for better performance. To the best of the author's knowledge, this research is one of the first attempts that used CUDA technology to try to simulate massive crowds. Therefore, the focus of this thesis is to provide some performance-enhancing information to the researchers in this field, through several simple but useful contributions. Some of the contributions are related with efficient usage of general purpose parallel programming considering the architecture of CUDA technology, and other contributions are related with the implementation of massive crowd simulation.

NVidia categorizes CUDA performance optimization issues in three strategies [11]. The first strategy is to maximize parallel execution. This strategy is quite important since massive crowd simulation contains many virtual characters. In order to implement this strategy effectively, each virtual character or processed entity is handled by a separate GPU thread. However, this action is not sufficient enough by itself to ensure best throughput. There are several outstanding issues to increase parallel processing performance such as using both the CPU and GPU resources considering the required processing power by the virtual characters. As in the analogy of LOD, some virtual characters may demand more processing power than the rest. In such a case, it is better to employ lightweight GPU resources for the majority while using heavy-duty CPU threads for minority that requires much more computations. Similarly, classifying and sorting virtual characters considering possible execution paths also helps achieve better speedups, since threads are ensured to follow similar execution paths.

The second strategy is to optimize memory usage to achieve maximum memory bandwidth. This issue will be examined in depth throughout this thesis. Considering this strategy, a solution based on data structures and data compaction has been offered. The proposed solution is to separate attributes and pack them into different data structures, so that the CPU-related attributes are not copied to the device, thus helps avoid extra data transfer overhead. To minimize device memory accesses, the transferred data is packed into GPU friendly data structures. The proposed solution minimizes data transfer and memory access costs as much as possible. Such an optimization helps achieve further speedups.

The final strategy is instruction-level optimization. Such an optimization can be done by simply following the details explained in CUDA documentation [9, 11]. Although details of this strategy will be covered in the following chapters, there is no contribution regarding instruction-level optimization.

Besides CUDA programming issues, there are several more contributions for the means of implementing crowd simulation related functionality by CUDA kernels (functions that run on the GPU). This thesis covers an in-depth look at implementing fuzzy inference using CUDA. The GPU kernels help by computing more than half billion fuzzy inferences per second. An infrastructure to handle such a huge amount of fuzzy inferences, provides developers the ability to generate scenes including massive virtual characters that perform distinct and non-deterministic actions. The contribution in this issue is to capture fuzzy knowledge-base and fuzzy rule-base via a simple GUI and transfer required parameters to the GPU with minimal coding effort. Expert knowledge is converted into an XML script, from which the fuzzy inference parameters are extracted. These parameters are consequently transferred to the constant memory of the device (GPU). Thus, it provides significant speedup since fuzzy logic kernels frequently access these parameters. The attributes of the virtual characters are also tried to be represented with device architecture friendly data structures such as 32, 64 and 128-bit words. The combination of such approaches not only lets developers include fuzzy logic functionality on the GPU with minimal effort, but also allows them to provide further speedups.

Another implementation-oriented contribution is to include many of the physical objects in the virtual environment with massive number of characters. The medieval-era case study covered in Chapter 6 realistically simulates the physics of thousands of arrows in real-time. It is certain that the interaction among the physical objects, the virtual characters and the realistic trajectory computations introduce an extra computational overhead. To overcome this power-hungry task, a solution that blends existing spatial hashing techniques and CUDA architecture is offered. Unfortunately, even the huge computational power offered by the GPU is still far away from creating this kind of simulation. In such a case, several assumptions can be made to minimize computational cost and architectural constraints. In fact, the offered solution contains some heuristics and assumptions that are specific to this case study. However, it seems possible to adapt this solution to similar applications.

## 1.5 Outline

The outline of this thesis is as follows:

- Chapter 2 starts with the literature survey about general issues in crowd simulation. This is followed by the detailed summary of crowd simulation studies using parallel computing.
- Chapter 3 explains the CUDA architecture as an introduction and gives details considering the aspects of this thesis.
- Chapter 4 demonstrates the details of using general purpose parallel computing for real-time massive crowd simulation in a step-by-step manner. In each step, the actions required to achieve further speedups are presented.

- Chapter 5 explains the details of fuzzy inference implementation on the GPU using CUDA.
- Chapter 6 covers three case studies. The first one demonstrates how massive crowd simulation can be used as a middleware for video sports games. In this case study, general purpose parallel programming is used to generate behaviors of the soccer game spectators. The second case study, which is inspired from the work of science-fiction writer H.G. Wells [12], covers the simulation of a medieval-era combat including nearly 250,000 warriors. The last case study gives the short summary of a work about virtual marathon.
- Chapter 7 concludes with results and summarizes the contributions. Potential follow-up work is also laid out in this chapter.

# CHAPTER 2

# RELATED WORK AND BACKGROUND

Simulating the behaviors, actions and movements of virtual characters in a crowd can be defined as crowd simulation. This research area has always attracted quite a significant interest from researchers of different disciplines, due to existence of large application fields such as military training, emergency planning, computer games, and architectural design. This chapter summarizes challenging issues in crowd simulation and gives literature background regarding crowd simulation using various parallel processing hardware and techniques.

## 2.1 Crowd Simulation

Real-time crowd simulation applications started in the mid and late nineties when commodity hardware started to meet the computational requirements of this community [13]. Since then, the studies are all focused on some or all of the challenging issues given below, which will be covered in detail throughout this section.

- Crowd Representation
- Crowd Navigation
- Crowd Variety
- Crowd Behavior

## 2.1.1 Virtual Crowd Representation

Although great progress has been achieved in graphics hardware in the last decade, crowd representation remains an important issue because of insatiable realism expectations. Depending on population, realism, perception, and hardware, various crowd representation techniques can be employed including:

- Geometric representation
- Image-based representation
- Point-based representation
- Hybrid representation

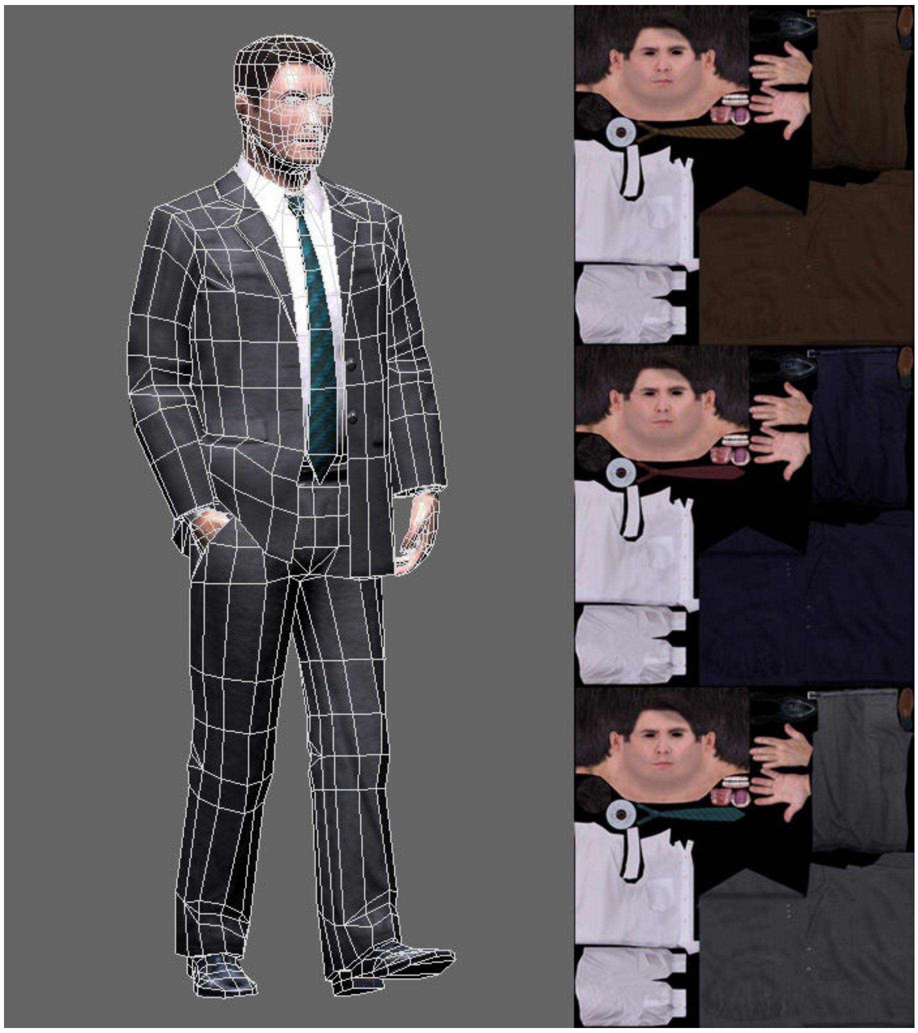


Figure 5: A geometric model and various texture maps.

Geometric representation uses virtual character models in 3D mesh. These models can be artist-made or scanned. The geometric model's visual quality is strongly related to the quantity of the meshes and resolution of the texture map. Figure 5 shows a typical geometric model.

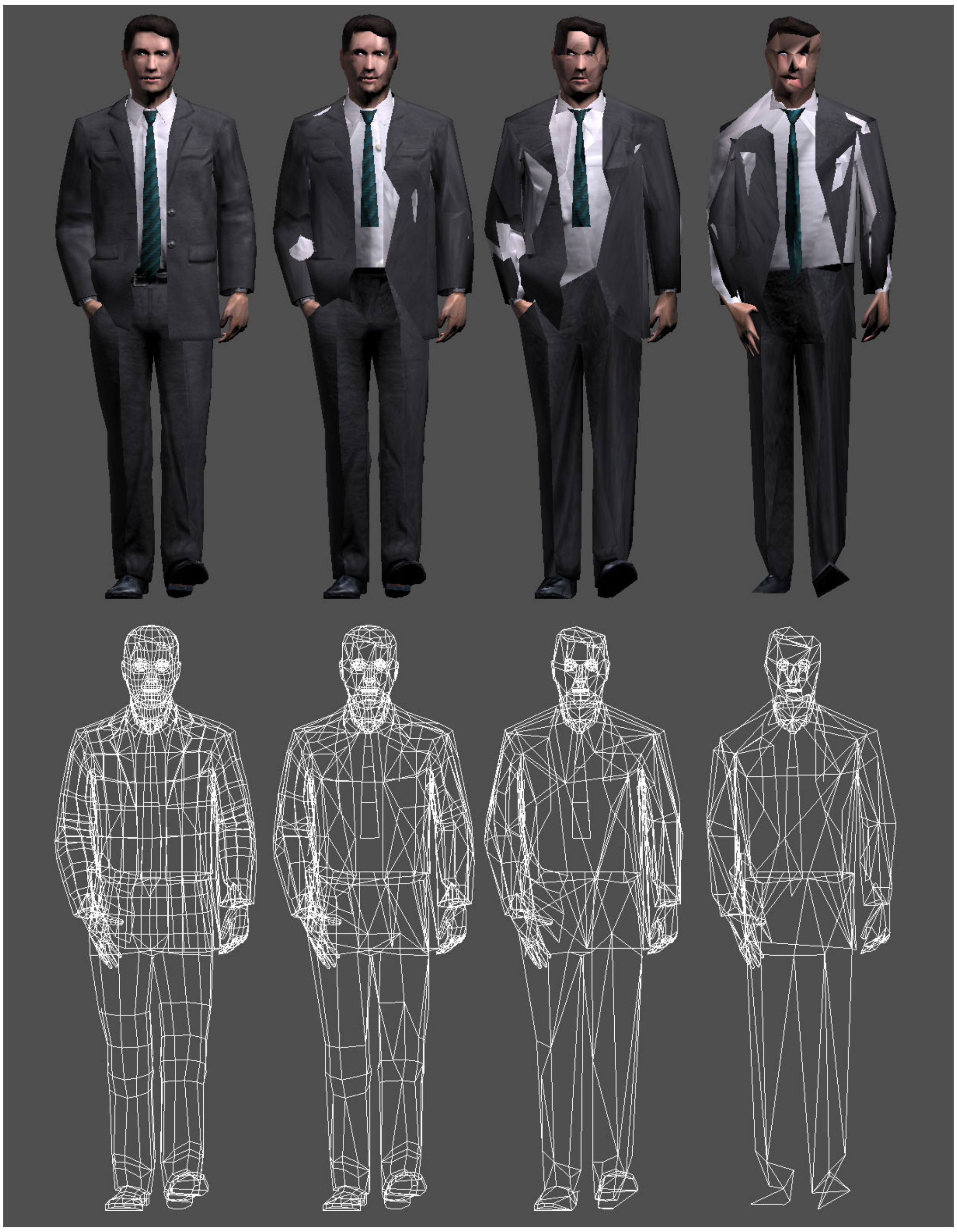


Figure 6: Auto-generated LOD models.

(Automatic polygon reduction usually produces visually poor results.)

In order to improve the rendering performance, a geometric model is mostly represented with different LODs, as illustrated in Figure 6. In crowd simulation applications, geometric models comprised of thousands of polygons and several LOD models are commonly used [14-16]. Although this approach can be easily implemented, rendering background characters that are far away from the camera is difficult because representing them with few polygons may not produce visually convincing results (Figure 6), while using high-polygonal models consumes too much of the GPU's resources. McDonnell et al. worked on perceptual issues and reported the facts about mesh simplification for virtual characters [17].

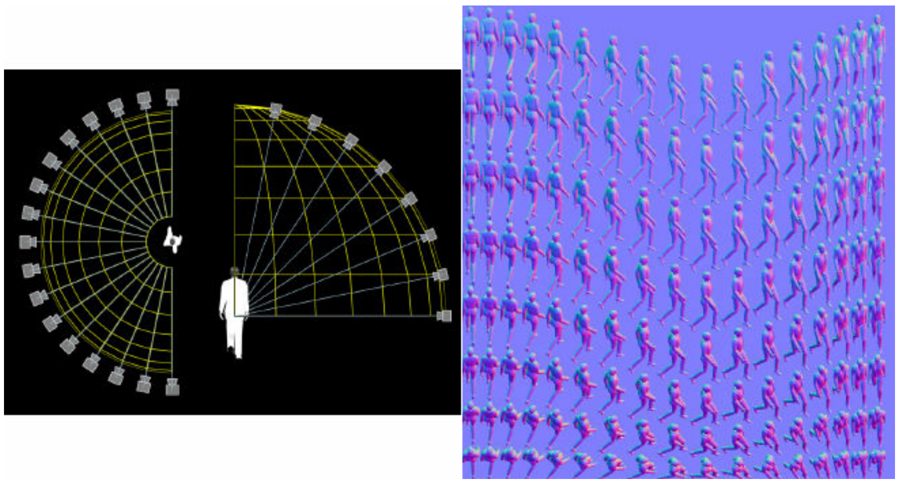


Figure 7: Illustration of an impostor and walking animation.
(Courtesy of Simon Dobbyn)

Image-based representation of crowds, called impostors, is very popular in crowd visualization [18-20]. Basically, impostors are many 2D images pre-rendered for all possible camera angles, thus trying to give the visual impact of 3D models. Figure 7 shows an impostor of a single human. The number of images increases in a linear fashion when more animation models are employed. Kavan et al. introduced polypostors to overcome this issue [21]. Polypostors are 2D polygonal characters that support a greater variety of animations without introducing any overhead since the rendering cost of a quad (two triangles) or several polygons are almost the same. But the authors reported that polypostors are limited to basic actions such as walking. It has already been demonstrated that impostors help generate perceptually realistic crowd scenes if they are away from the camera [22]. However, they suffer from low visual quality due to pixelization and flat view when models are close to the camera, as shown in Figure 8. Hybrid techniques, which will be explained later, are proposed to overcome this problem [23].

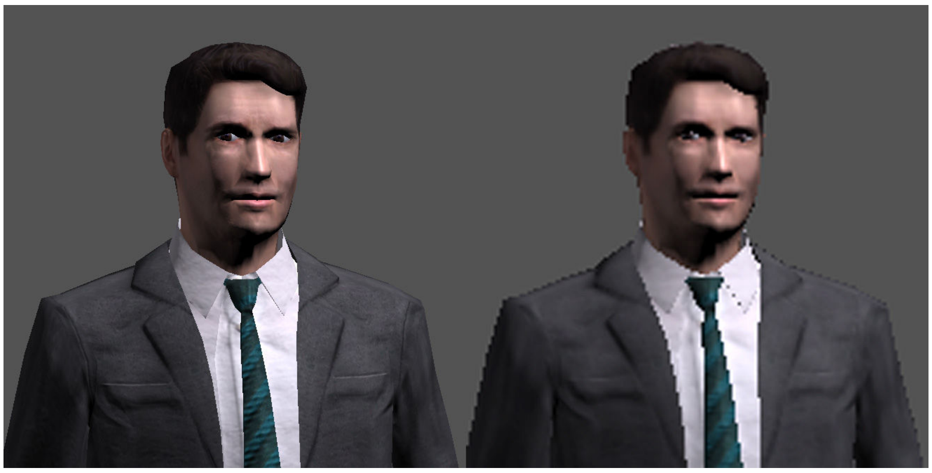


Figure 8: Visual quality comparison of 3D model and impostor image.
(Pixelization problem occurs due to use of raster impostor (right image).
This problem is similar to the vector vs. raster issue in many other disciplines.)

Grosman and Dally [24] implemented point-based rendering, in which point primitives are used to render geometric objects instead of polygons. The points should have normal, color and depth information, and thus better representing the original model. Several researchers have proposed improvements to the point-based rendering technique [25-27]. Rudomin and Millan used point-based rendering to visualize virtual crowds [28]. They also compared impostors and point-based rendering in crowd simulation, considering various parameters such as animation smoothness, rendering performance, and texture memory usage [29]. Neither method was superior, since both techniques have similar performance results and comparable pros and cons.

To overcome the bottlenecks introduced by geometric, image-based and point-based representation techniques, a hybrid approach in which high-cost/high-quality representation (geometric) and low-cost/low-quality representation (image-based or point-based) can be employed to render a virtual crowd. Dobbyn et al. proposed using geometric models and impostors (called as geopostors) together to increase rendering performance [23]. In this approach, geometric models render virtual humans within a certain threshold distance, while impostors are employed for the rest. Hamill et al. improved this approach with a perceptual metric [30], researching when to switch the geometric model and impostor. The geometric model can be switched to an impostor representation when one texture element (texel) corresponds to a pixel on the screen. McDonnell et al. validated this result by examining the participants' perception [22]. This hybrid approach has also been employed in latter studies. In a recent work, Maim, Yersin, and Thalmann used the hybrid approach in three levels [31]. The first level supports facial and hand animation with high-detail mesh models. This level is used to render the virtual characters close to the camera view-point. The second level represents virtual characters far from the camera with pre-computed static meshes. The final level uses impostors for the virtual characters that occupy the very limited space in the view-frustum.

### 2.1.2 Virtual Character Navigation

Navigation is another challenging issue in crowd simulation, since avoiding collisions in real time among thousands of virtual people, the entities, and the structures in the virtual environment is difficult. Additionally, the virtual characters' movement path should be visually convincing to meet realism expectations. There are a lot of parameters that determine a person's path. For example, people usually prefer the shortest path while they are rushing and crowded streets when it is dark. The computations get more complicated when the collision avoidance system of humans is included. Human path planning is updated frequently, considering many factors such as congestion. While moving, humans interact with other people, observe the surroundings and the other people's actions, and modify their paths accordingly, adjusting their orientation to avoid possible collisions. Basically, trying to implement path updating mechanism for every single entity requires huge processing power and good algorithms. This agent-based approach is computationally expensive and produces less realistic crowd behavior such as sharp paths. The literature covers many studies regarding the navigation of virtual characters in synthetic environments. Some of the outstanding works are summarized as follows.

Treuille, Cooper and Popovic implemented a new approach in crowd navigation [32]. They examined this problem from a perspective based on continuum dynamics. The fluid dynamics community inspired this study by introducing the use of a continuous density field to represent pedestrians. Treuille et al. presented a real-time motion synthesis model for large crowds without agent-based dynamics. They described a new type of crowd simulator driven by dynamic potential fields. These fields integrated the global navigation and local collision avoidance. The approach is based on two simple terms: a velocity-dependent term inducing lane formation, and a distance-based term stabilizing the flow. Furthermore, Treuille et al. showed how individuals produced more intelligent behavior with their knowledge of future. Their final renderings demonstrated a smooth flow under a variety of conditions and exhibited emergent phenomena observed in real crowds [32].

Yersin et al. tried to steer virtual crowds by using a semantically augmented navigation graph [33]. They addressed the problems arising from the lack of intelligent and realistic behaviors of virtual characters. To overcome this problem they provided the knowledge of the environment by exploiting a navigation graph. The algorithms employed to prepare navigation graphs were defined in another study [34]. After capturing the environment's topology, the navigable areas determined from structures, slopes, and several other parameters are delimited. Passing from one area to another is only allowed where an intersection exists. Finally, the graph is prepared by employing a deterministic graph building method based on the voronoi diagram. The paths in the scene are constructed from this graph. Cylindrical volumes simply define the walkable paths reconstructed in a multi-level manner. For example, there are two paths in one staircase: down to up and up to down. To give information to the virtual pedestrians, the nodes are labeled with semantic information relative to their region (e.g., Park, Hotel, Circus) [34].

Bayazit, Lien and Amato studied regarding flock behavior [35]. For simplicity, flock behaviors can also be employed to visualize human groups. There are many methods to simulate flocking behaviors [36], [37]. Flocking behavior studies mostly use local environment for decision making. Bayazit et al. investigated the contribution of global information, as an environmental roadmap that enables more sophisticated flocking behaviors, supports global navigation, and planning. They embedded several behavioral rules for individuals to modify their actions according to environment and changing states. This yielded different patterns that have not been observed in previous studies such as forming an unordered group in open areas and follow-the-leader fashion in narrow passages. Use of global information provided by their rule-based roadmaps improves the behavior of autonomous characters, and particularly enables more sophisticated behaviors than traditional flocking algorithms that only use local information [35].

Lamarche and Doinikan proposed an approach that enables the real-time simulation of hundreds of virtual pedestrians. They produced an accurate environmental model from the virtual environment's underlying geometry. This vector structure was used to compute optimized paths. Most importantly, Delaunay triangulation is refreshed automatically according to the population density. This method is similar to the load-balancing approach in parallel processing. Additionally, their model includes reactive navigation architecture inspired from physiological studies [38]

Some applications require less realistic navigation, such as collision-free, meandering people. In such a case, several assumptions can be made to minimize complexity of the navigation problem. In an immersive virtual reality game experience, Ulhaas et al. implemented boids algorithm and represented human agents as particles [37]. Collision avoidance between characters was achieved by calculating physically based repulsion forces between particles, which are added to the steering force determining proximate position. They used a three-zone model to set the strength of the repulsion forces. The resulting force function was adjusted by the physical parameters stiffness and viscosity.

Lerner, Chrysanthou and Lischinski studied crowd simulation in a data-driven approach, constructing a trajectory database from real videos [39]. They captured several videos and extracted individual trajectories manually. In the simulation phase they tried to find the best trajectory for each agent considering their spatial location, surrounding agents and obstacles. The characteristics of this approach are summarized as follows:

- Produces more realistic and better looking simulations with many actions as seen on videos
- Chooses AI from the virtual character's location without determining individual behaviors and assigning rules
- Creates different crowd animations from various data sets (panic, normal)
- Synthesizes new actions from existing data sets

Maim, Yersin and Thalmann employed a level-based navigation model [31]. Virtual character navigation in the foreground was computed from interaction and collision avoidance parameters. The background people were navigated using a simpler approach. Finally, they distributed virtual characters in available zones using statistical methods.

### 2.1.3 Population Variety

Virtual environments need to be populated with different looking virtual characters because a scene with lots of clones is far from realistic and the clones that look identical can easily be identified by the users [40]. It has already been shown that the users more easily detect the clone models that look alike. A proper data set needs to be used to populate scenes. For example, a large collection of adult Native Americans cannot be used to simulate the streets of Shanghai.

There are two main approaches for 3D virtual human model generation:

- Creative (Artistic Process): The process of 3D modeling that uses special software such as 3D Max, Maya, Poser and Lightwave. The creative approach gets harder and consumes time when a higher level of realism is desired. Since model generation effort is repeated for each virtual character, this approach is not practical to achieve large population variety.

- Parametric Construction (Reconstruction): This method uses base models to generate automatically new virtual humans by applying several functions and changing body shape parameters. To generate realistic models, the science of anthropometry (the measurement of living human individuals for the purposes of understanding human physical variation) is required.

Allen, Culles and Popovic [41] demonstrated a system for synthesizing high-resolution, realistic 3D human body shapes according to user-specified anthropometric parameters. They used whole-body 3D laser range scans of 250 different people. The authors used an artist-made human model with 60,000 vertices as a basis to produce other human models from laser scan data. In the first step, they deformed the base model with each body scan data to make a model without gaps, ready to be used by computer graphics applications. In this step 74 landmark positions were used to make exact matches, very similar to 3D registration. Principal Component Analysis (PCA) was used to minimize the data set by removing human models above a certain threshold. Then, relationships between the body parts such as height were defined to synthesize new body shapes. They used six anthropometric measures (stature (height), bitragion breadth (head breadth), shoulder breadth, arm length, bi-cristale breadth and leg length) to create a body type of the desired proportions [41].

Seo, Cordien, Philoppen and Thalmann generated animation ready models quickly [42]. They introduced several deformers for each body part to automatically adapt the model to different sizes and proportions. They employed a generic 3D body model and H-Anim skeleton hierarchy. To attach a skin to a body, they simply assigned a weight to each vertex using related bones. This structure lets the user easily deform a bone, and consequently the attached skin also deforms and fits. But there are some parts in the human body that are not suitable for bone deformation, such as breasts, bellies and bottoms. The authors assigned free-form curves (NURBS, Bezier curves, B-Splines) to deform these parts.

Thalmann and Seo introduced a framework for time-saving generation of realistic, animation-ready population body models while keeping as much distinctiveness between individuals as possible [43]. Similar to the study of Allen and his colleagues, Thalmann and Seo also used 3D scanned body models as prototypes. To generate populations they proposed a method which smoothly interpolates among these prototypes by using scattered data interpolation techniques.

A practical approach for model variation is to apply different texture maps to the same model, as demonstrated by Ciechomski et al. [44]. The important point of this technique is the choice of appropriate hair, skin, and eye colors. Similar approach has also been employed by McDonnell et al. to generate crowd variety [40]. Dobbyn et al. and McDonnell et al. used cloth textures to generate crowds with different clothes in real-time [45,46]. Similar approach has been employed in thesis as illustrated in Figure 9. Since the focus of this thesis is not to generate on-the-fly textures, artist-made textures were used. In this figure, blue, brown and gray suits are used to generate three different virtual characters. Figure 10 shows crowd variety using different textures and scaling.

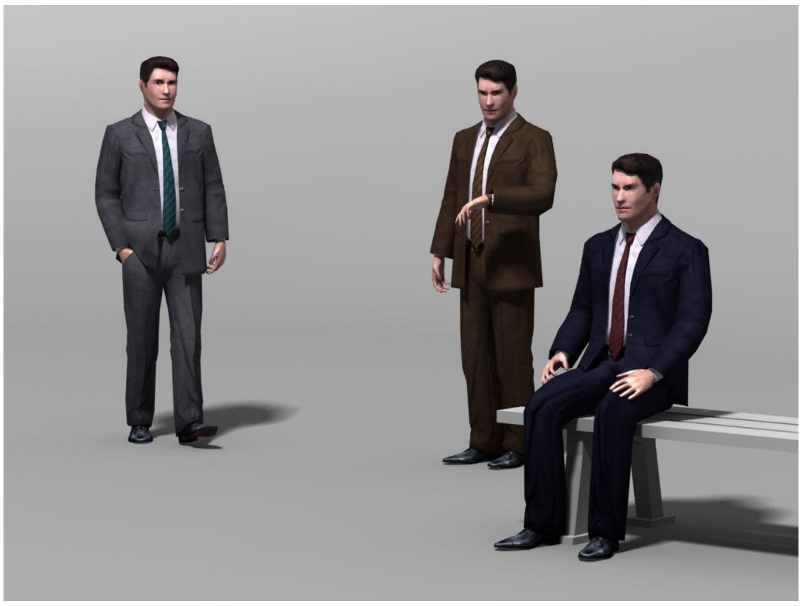


Figure 9: Population variety using different texture maps.

Maim and his colleagues introduced YaQ, an architecture for crowd simulation [31]. In YaQ, crowd variation is one of three components, and it was designed to generate unique individuals from a small set of templates. This component can change people's shape and assign different textures, colors and accessories. It also provides variety through different animations.

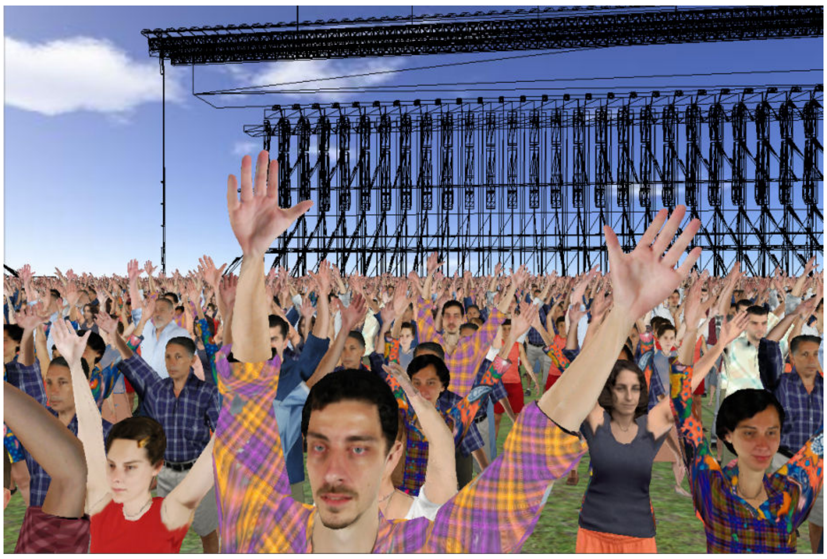


Figure 10: Population variety in a virtual concert.

### 2.1.4 Virtual Character Behavior Modeling

Behavior modeling is one of the most extensively-studied areas in virtual crowd simulation. There have been different approaches and studies dealing with virtual character decision making and behavior modeling in virtual worlds, but no method has been proven to be the best. The choice mostly depends on the goals and priorities of the study. In virtual worlds, characters simply perceive the surrounding world and react accordingly. Crowd simulation methods may produce repetitive and predictable motions, therefore, are easy to implement, while some methods are very power-hungry. Several approaches are:

- Rule-based Systems.
- Physics-based Systems
- State Machines
- Fuzzy Inferences
- Decision Trees
- Neural Network/Genetic Algorithms
- Real-life Data Extraction from Videos (Data-driven)

Ulincy and Thalmann studied crowd behaviors in emergency situations [47]. The behavior model described in their paper is simple enough to allow the real-time computation of many characters. However, the model based on rules and finite state machines is also capable of generating interesting behaviors. This study also provided script based interface to manage actions of the crowd.

Thalmann, Musse and Kalmann proposed the distribution of autonomy among the simulation's entities to achieve realistic behaviors [48]. The method they explained employs perception and emotion to shape behaviors and ultimately actions. This paper mainly focuses on humanoid autonomy. Thalmann et al. defined three types of crowd behavior that reflect three Levels of Autonomy (LoA):

- Guided crowds: Behaviors explicitly defined by users.
- Programmed crowds: Behaviors programmed in a scripting language.

- Autonomous crowds: Behaviors specified by rules or complex methods.

Essentially, the LoA concept decreases complexity by assigning autonomy to groups rather than individuals. Furthermore, Thalmann et al. also offered similar structure for objects in the scene. Similarly, to minimize complexity, objects were also given an autonomy they do not possess in real life [48].

Heigas et al. worked on realistic crowd simulation in the ancient Greek agora of Argos [49]. They visualized a social theater where two kinds of phenomena took place: interpersonal interactions (such as small group discussion and negotiation) and global interactions (such as flowing and jamming). This paper focused on the collective human phenomena called non-deliberative emergent crowd phenomena, typical of collective emergent self-organization such as flowing, avoiding, jamming, and collapsing.

Sung, Geicher and Chenney showed how scalable behaviors can be used in crowd simulation using a two-level decision model [50]. At the high level, they adapted a situation-based distributed control mechanism that gives specific detail to each individual about how to react to its local environment. At the low level, a probability scheme computes probabilities over state transitions and produces samples to move the simulation forward. They tried this approach in several situations, such as theatre and street environment. Ultimately, the described framework created complex crowd behaviors through the composition of situations and behaviors while minimizing the data stored in each character [50].

Braun et al. researched individual characteristics' impact in the evacuation from emergency situations [51]. In this study, they implemented physically-based model for crowd simulation in panic situations. The authors also added individual behaviors to get a more realistic simulation. Their approach successfully visualized the individual parameters of altruistic people, who tend to rescue dependent people before themselves.

Sakuma, Mukai and Kuriyama used both psychological issues and pedestrian speed and density measurements for crowd simulation [52]. They showed that particle system based approaches are inappropriate for crowd simulation, since complicated human perception mechanism requires the inclusion of human perception and psychology. This study also explains collision avoidance according to psychological issues. They indicated that from a psychological viewpoint, the neighboring agents impose mental stress on each other, which can be estimated on the basis of a personal space model. This model experimentally showed that mental stress increases as other people get closer. Within a certain threshold this stress becomes critical.

There exist different studies that involve additional parameters to crowd simulation. In one such study, Pelechano et al. employed psycho-socio-physio-logical parameters (emotions, stress, personality, nationality, cultural background, psychological models, roles, and communication) with existing crowd simulation [53]. This approach produces more realistic results compared to the rule-based or force-based methods. They proposed a structure, called PMFServ, to take into account psychological elements that affect human behavior. PMFserv was conceived as a software system that would expose a large library of well-established and data-grounded Performance Moderator Functions (PMFs) and Human Behavior Representations for use by cognitive architectures deployed in a variety of simulation environments. As a result they succeeded in getting more behavioral variety by including agent psycho-socio-physio-logical parameters into decision making system.

Later, Pelechano et al. presented the HiDAC (High-Density Autonomous Crowds) system [54]. They combined rules with physical forces, and determined agents' behaviors individually by employing a two-level approach, which models actions such as navigation, learning, communication, decision-making, perception of the environment and collision avoidance.

O'Sullivan et al. [55] used LOD not only for visualization and motion but also to simplify behavior of crowds. They reported that LODAI reduces computational costs by offering very low-cost behavioral model for the background individuals, and a detailed model for the foreground individuals.

Chittaro and Serra demonstrated autonomous virtual characters with behavioral differences [56], bringing distinct behaviors from probabilistic influence on behavior selection. Badler et al. also used personality in EMOTE (Expressive Motion Engine), which influences the character's perception and actions [57]. Bécheiraz and Thalmann introduced the basis of this behavioral modeling [58]. They used perception to generate emotion, then both perception and emotion to invoke a behavior corresponding to an action. Ayesh et al. tried fuzzy individual modeling (FIM) [59]. They also used perceptions to update the emotions that trigger different behaviors.

Rudomin and Millan used XML scripting to specify behaviors and employed a Finite State Machine as a processing method [59]. Furthermore, they implemented probabilistic FSMs, hierarchical FSMs and layered FSMs to produce non-deterministic results [60,61].

Similar to the data-driven study of Lerner et al. [39], Peters and Ennis have also used pre-recorded video sequences to extract real-life crowd behaviors and generate real life-like simulations [62].

## 2.2 Crowd Simulation Using Parallel Computing

In the literature there exist various research studies regarding crowd simulation, using various parallel computing platforms. Depending on the underlying hardware, various approaches have been implemented. This section summarizes some of the outstanding work related with the focus of this thesis.

### 2.2.1 Crowd Simulation Using Multi-CPU/Multi-Core Parallel Computing

In PSCrowds study, Reynolds implemented a virtual underwater environment with thousands of inhabitants, running in real-time by using Playstation3 game console that contains eight processors (one Power PC processor (PPU) and seven Synergistic Processor Units (SPUs)) [63]. Reynolds modeled fishes as interacting particle systems, an approach that requires having information about the surrounding individuals. Using this information, the simulation model computes the behavior of a fish. Basically, an interacting particle system computes the behavior by using the distance information with the other agents. If the distance test is performed for all the agents, the computational complexity becomes $O(n^2)$. Although this approach is simple and works well for small populations, it becomes useless for large populations. In order to minimize computational load, spatial hashing techniques can be employed. While using spatial hashing, Reynolds designed an algorithm to make effective use of the underlying parallel processors, and ultimately to achieve higher performance [63]. Reynolds used static and regular sub-cubes as spatial cells, represented by a software class called “bucket.” Each bucket is processed by a single SPU. Such a granularity allows the workload to be divided into jobs, executed independently or in parallel by multiple processors. These buckets are also processed in an order-independent way to avoid scheduling overhead [63]. In this study PPU is used as a coordinator. Some of the major responsibilities of the PPU are simulation cycle update, synchronization, communication, and task assignment to the SPUs. In the beginning of each rendering, individuals are assigned to a single bucket. After this step each SPU updates the simulation on bucket-base. A software class named “NearestN” is responsible for handling possible collision in the neighbor buckets. Reynolds has not used any load-balancing algorithm because of the homogenous distribution of individuals in the aquarium. PSCrowds was repeated for various scenarios. In the 2D version, 15,000 individuals were rendered at 60 fps. In the 3D version the population was 10,000. Finally, 5000 highly-detailed fish models were rendered in 3D at 30 fps [63].

In a similar crowd simulation that visualized thousands of chickens, RapidMind Inc. described the use of multi-processor architecture, Cell BE (Cell Broadband Engine) which contains nine CPUs [64]. In Cell BE, one processor is IBM Power PC Processing Element and the rest are specialized processors (SPE) tuned for high-performance floating point and integer math on short vectors. Since much of the Cell BE's impressive performance resides in the SPEs, the key to obtain high performance on the Cell BE processor is to use SPEs efficiently. In this study, crowd simulation mainly covers the following tasks [64]:

- Neighbor finding: Each character needs to perceive the state of its nearest neighbors.
- Environment access: Each character needs to perceive its local environment.
- State update: Each character needs to update its state over time.
- Visualization: The current state of the each character needs to be rendered to the screen.

Quinn, Metoyer and Zaworski worked on pedestrian simulation in a PC cluster made up of eleven computers [65]. In this system, they succeeded to move 10,000 pedestrians with 1:50 seconds intervals. Their main goal was to make a system that could simulate and visualize large crowds in real-time. In this study, individuals were represented with 2D dots; hence no realistic rendering approach was employed. That is why the rendering cost was not important. The power-hungry part of this study was updating pedestrian locations based on social-powers. In this method, the distance between pedestrians would cause repulsive or an attractive forces on every other pedestrian. In real life this distance is a few meters away. In order to meet real-time constraints, Quinn et al. used PC cluster and assigned a manager-worker style architecture for this task. The manager was responsible for communication with other PCs in the cluster (workers). It collected current positions of each individual after updating cycle and passed this information to the rendering engine. Each worker process was responsible for simulating pedestrian movement within a rectangular region of the building. Since the social-powers approach works on a few meters distance, grouping people in given boundaries minimized the required processing time as Reynolds described [63]. Similarly, Quinn et al. also used regular, fixed grids as spatial cells. In this study, they divided the simulation environment into 4m$^2$ square cells. With nine neighboring cells forming a 6m. by 6m. square, the greatest possible distance is 5.26 meters in the diagonal. In order to communicate PC cluster, they used Message Passing Interface (MPI) library, which is very popular for parallel processing. During their study, they observed a linear performance increase as they added more PCs to the cluster. This scalability means that it is possible to increase the number of interactive actors by adding more PCs to the cluster. Since there is no inter-communication between workers, this really helps minimizing network traffic overhead. Similar to the Reynolds' study they did not employ any load balancing.

Steed and Haidar focused on dynamic allocation of regions considering crowd distribution and spatial partitioning algorithms [66]. They aimed to minimize network overhead when a cluster of servers is used to simulate large crowds. Unlike the studies summarized above, Steed et al. tried to minimize the load balancing problem, which likely occurs due to a larger crowd occupying a specific zone such as city center [66]. They used pre-recorded activity data to compare the effectiveness of the investigated partitioning methods. Among the four spatial partitioning schemes (quad-tree, k-d tree unconstrained, k-d tree constrained and region growing), region growing, used mostly in image processing, achieved the best result. This algorithm simply starts with a selected seed point and enlarges until a certain threshold is reached. In this study, Steed and Haidar picked seed points close to the mass populations [66]. The resulting partitioning became an even and irregular shape reflected the complex road structure and the population distribution. They defined the problem of using a priori regular partitioning as not "reflect[ing] how participants will actually use the space. Certain regions might be very crowded and thus they become failure points at run-time" [66].

Zhou and Zhou tried to use a PC cluster to partition flock [67]. Their aim was to minimize O($n^2$) complexity and to increase the entities in the simulation. Initially, they investigated two possible communication methods between PCs in the cluster. The first technique was "all-to-all" communication; the second was "near-neighbor-communication." In all-to-all communication each PC can communicate with the rest and collect information about dynamic entities on other zones. Although this approach guarantees global vision, it significantly increases network traffic which causes an overhead in the simulation system. Taking the limited vision of boids in the flock into consideration, a better approach which is "near-neighbor-communication" can be used to decrease network traffic. Besides network communication, Zhou and Zhou also examined even-distribution and dynamic-load-balancing issues. They observed due to migration, un-even distribution frequently happened in their simulation system. In order to overcome this improper work-load distribution, they considered population distributions to redefine zones, and they used dynamic partitioning when necessary. Up to 512 boids were rendered in real time using different cluster configurations. They used a proper load-balancing algorithm to obtain a significant performance. However, it was indicated that frequently invoked load-balancing may lead to inefficiency. They offered to use suitable threshold value to invoke load-balance as a solution.

The commodity multi-core CPUs offer researchers a way to achieve speedups by exploiting multi-thread programming. Berg and colleagues demonstrated a collision avoidance approach in crowded scenes that also contain moving obstacles [68]. They used a pre-computed road map for global path planning. This approach increases the computational performance of the simulation with multi-core CPUs. Their approach was built upon the concept of velocity obstacles, a technique found in robotics for motion planning to avoid dynamic obstacles. In this velocity-obstacles based implementation, each agent senses the environment independently to compute a collision free path. Agents acquire information on the positions and velocities of other agents and obstacles. This information is used to decide how to move locally. However, this approach ensures no oscillatory behaviors. The study employed Intel

Xeon X7350 2.93 GHz with 16 cores. Their approach is fully parallelizable since each agent requires independent computation. This parallel solution showed that simulation performance increases almost linearly with the number of cores. They achieved interactive rates on virtual environments containing several thousands to tens of thousands of agents.

In a recent study Stephen and his colleagues presented high-performance collision avoidance for crowds [69]. Similar to the study by Berg et al., their approach was built-upon the concept of velocity obstacles. This newly introduced collision avoidance algorithm was called ClearPath. They also extended ClearPath with data-parallelism and thread-parallelism, and named this extended version as P-ClearPath. The results of the study showed that P-ClearPath achieves 8-15× speedup compared to the previous VO-based solutions. In the implementation, an Intel Quad-Core CPU that supports 16 threads was used. In this setup a 2× speedup was observed when dynamic partitioning was used to reduce load imbalance. Static partitioning causes the threads handling sparsely populated zones to finish computations earlier than the threads dealing with the crowded zones. It was also indicated that P-Clear uses 20% of the CPU resources for simulating 5000 virtual characters, meaning that the rest of the CPU resources can be used for other computations such as AI.

### 2.2.2 Crowd Simulation Using GPU

Recently, several studies have been published regarding crowd simulation with a GPU. Current studies employ NVidia CUDA instead of the previously used GPGPU.

An early work regarding crowd simulation on the GPU was demonstrated by Courty and Musse [70] in a study named FastCrowd. Significant speedup was achieved with the GPU implementation instead of the CPU implementation.

Erra et al. demonstrated massive simulation of a distributed behavioral model on the GeForce FX 5800 GPU [71]. Well-known boids implementation was chosen. GPU results were compared with CPU results, and better performance was reported. Although an early model of compute-capable GPUs was used, the speedups were very important because it showed that the GPUs began to take the compute responsibility from the CPUs.

Richmond and Romano presented a high performance agent based pedestrian simulation using GPGPU [72]. They indicated that, GPGPU was chosen because NVidia CUDA is a vendor dependent solution. Their work was designed to support a scripting based interface, thus defining more complicated agent behaviors. To increase simulation performance, the pedestrian data and simulation were kept in the GPU. This approach helped them remove data transfer overhead. Similarly they used real-time feedback to employ an LOD system on the GPU and to improve rendering and simulation performances. With the LOD system, the reserved resources could add more characters to the simulation. They ran several simulations using different populations and rendering elements. When triangles were used to represent pedestrians, higher frame rates were achieved. For example, 13 fps was reported for the population of 262,144 pedestrians. The frame rates were decreased significantly when polygonal human models were used. 40 fps was reported for the population of 1000 pedestrians. GeForce 8800 GT GPU was used for these tests. The authors later improved this study and implemented the new version using CUDA. They reported 250× speedup compared to the single CPU implementation.

D'Souza and his colleagues simulated a mega-crowd on the GPU [73]. They successfully implemented SugarScape model with a GPGPU. The simulation, which includes more than two million agents, achieved 50 updates per second. The SugarScape model shares properties with agent-based models. The authors used GPU texture memory to store agent attributes and GLSL to write shader code. The simplicity of the model and the minimal cost of the rendering primitives might also help achieve the goal of rendering huge populations at high simulation cycle rates. However, the actual issue was to compute and store everything on the GPU and thus take full advantage of the extreme computational power and the high memory bandwidth. Although no tests were performed, it was indicated that the implemented prototype would outperform even High Performance Clusters (HPCs).

In a recent study by Passos et al., it was shown that simulation and visualization of very large crowds in real-time is possible with NVidia CUDA technology [74]. Researchers succeeded in running a simulation of more than one million boids at nearly 30 fps, a great number compared to the 15,000 boids Reynolds simulated with a multi-CPU architecture. This significant result showed that successful casting of the existing algorithms to general purpose parallel computing can create very significant enhancements to real-time crowd simulation performance. Although they represented boids with very simple geometric primitives, more complicated geometries can be used in the rendering process. The high performance of this implementation relies on an effective sorting method used as a base to the spatial hashing approach. The simulation was repeated, using different populations ranging from 64 to 1048576. Passos et al. also implemented the simulation on the CPU and the GPU and showed that GPU bypasses the CPU at around 250 boids [74]. As the number increases, the frame rates of the CPU decreases quadratically. The algorithm was tested on 2.4 Ghz and NVidia GTS 8800 GPU. The study was extended with the addition of a 3rd dimension, a new data structure, and several other sorting methods. Using the previous hardware, comparable speedups were again obtained [75].

Karthikeyan used CUDA technology to render crowds of virtual humans in his master's thesis [76]. The computations for the animation were compared on the GPU and the CPU. The results showed that the GPU bypasses the CPU at around 1000 virtual characters.

# CHAPTER 3

# NVidia CUDA TECHNOLOGY AND GENERAL PURPOSE PARALLEL PROGRAMMING

This chapter presents detailed information about the CUDA architecture and general purpose parallel programming. The issues covered in this chapter will be used throughout the rest of this thesis, to show how CUDA technology and parallel programming helps achieve significant speedups in massive crowd simulation.

## 3.1 What is NVidia CUDA?

Although announcements were made earlier, NVidia introduced CUDA to the public in February, 2007 [9, 10]. This technology was designed to meet several important requirements for a wide audience's use. One of the most important requirement is the ability to program GPUs easily. Simplicity is necessary to ease GPU parallel programming and enable its use in more disciplines. Before CUDA, GPU parallel programming was limited to shader models of the graphics APIs. Thus, only the problems well-suited to the nature of vertex and fragment shaders were computed by using GPU parallel processing. Additionally, expressing general algorithms in terms of textures and GPU provided 3D operations by using only float numbers were among the issues that limit the popularity of the GPU computing [77]. To achieve the goal of making GPU parallel programming easy and practical, NVidia offered to use C programming language with minimal extensions [9]. Another important issue is the heterogeneous computing model, which makes it possible to use CPU and GPU resources together [9]. CUDA lets programmers divide the code and data into sub-parts, considering their suitability to the CPU/GPU architecture and respective programming techniques. Such a division is possible because the host and device have their own memories. In this sense, it also becomes possible to port existing implementations gradually, from the CPU to the GPU [9].

It should be noted that CUDA is NVidia specific. In order to make a GPU parallel processing application run on ATI GPUs as well, there exists another programming tool called OpenCL (Open Computing Language). Currently, this initiation is lead by Apple Inc.

## 3.2 CUDA Architecture

The competition in the video-game industry to present more life-like and highly-detailed 3D graphics has also started a competition between the GPU producers to offer better hardware to the gamers. Thanks to the nature of the graphics processing, the newly released products offer highly parallel processing units with high-memory bandwidth and computational power of more than teraflops per second [8,9,77]. Modern GPUs are designed to support data-parallel computations, which means many threads execute the same code for each data-element. Data-parallel processing can be shortly described as mapping data elements to parallel processing threads [11]. The performance increases if there is little to no branching, since exactly the same code is executed for all parallel running threads. Image processing kernels and matrix operations are among the typical applications that get the most benefit from this architecture. However, significant speedups can be achieved in many additional disciplines, by porting existing algorithms to this data-parallel architecture [1-6]. In the near future, the hardware developers will offer better GPUs with more parallel processing threads. It is the responsibility of software developers to use this incredible GPU processing power that is comparable to the processing power of the high-performance computing clusters.

The data-parallel and thread-parallel architecture introduces scalability. Since no extra effort is necessary to run the existing solution, the new GPUs are capable of running more processing threads. It means that the code designed for the NVidia 8 series runs faster in NVidia GTX series without any additional coding. Considering the nature of massive crowd simulations, data-parallelism should be provided by setting one thread per virtual character.

The three abstractions offered by NVidia ensure the granularity required for good data parallelism and thread parallelism [11]. These abstractions listed below are designed to make CUDA programmers' life easy.

- Thread Group hierarchy: Threads are packed into blocks which are also packed into a single grid.
- Shared memories: CUDA lets threads use six different memories that are designed to meet different requirements.
- Barrier synchronization: This abstraction synchronizes threads within a single block and makes a thread wait the others to finish related computing, before going further.

C for CUDA makes it possible to write functions that run on the GPU by using C language. These functions are called "kernels," which are executed for each thread in a parallel manner, unlike the conventional serial programming functions that run only once.

CUDA's architecture offers thread hierarchy in top-down order as follows:

1. Grid: A grid contains one or two dimensional blocks.
2. Blocks: A block contains one, two or three dimensional threads. Current GPUs allow a block to contain 512 threads at most. The blocks are executed

independently, and they are directed to available processors to provide scalability.

3. Thread: A thread is the basic execution element.

This hierarchy and the structure are depicted by Figure 11. For example if it is assumed that 1048576 virtual characters to be processed independently in parallel manner and the block size is determined as 512, then there are 2048 blocks.

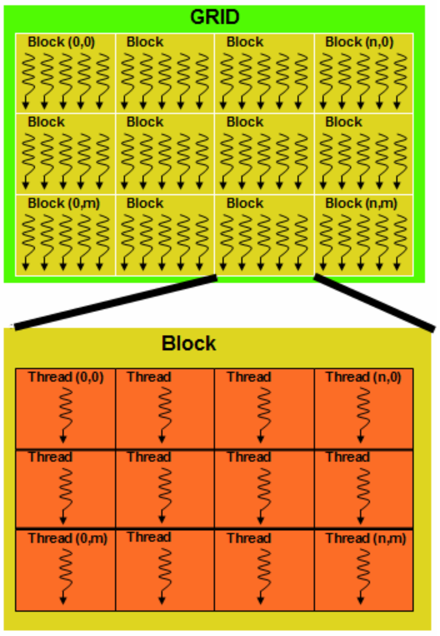


Figure 11: CUDA thread hierarchy.

## 3.3 GPU Processing

CUDA-enabled GPUs contain several multi-threaded streaming multi-processors (SM). For example, a GTX295 GPU contains 60 SMs. Each SM is comprised of eight Scalar Processors (SP). As depicted in Figure 12, SMs also have two special function units: a multithreaded instruction unit, and an on-chip shared memory. Each SP can run a single-warp (containing 32 threads) concurrently.

SIMT (Single Instruction Multiple Thread) architecture manages many concurrent threads. A SIMT unit, which handles every issue in warp-basis, exists for each SM. The execution of SIMT is illustrated by Figure 13. In each time step a common instruction is applied to the active threads. Thus, the performance improves when each thread follows the same execution path. Branching causes delay because the threads within a warp wait until a common instruction is reached. For the best performance, similar parallel-threads should be organized sequentially. This issue should be taken into consideration during the design phase.

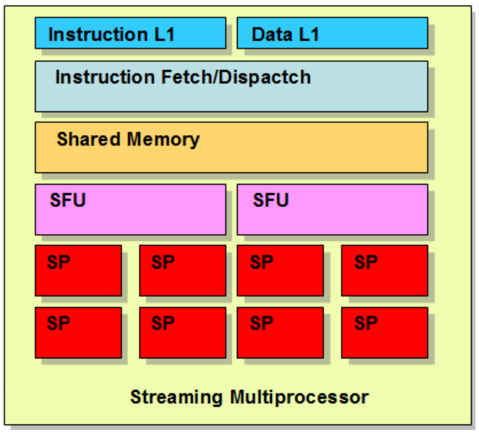


Figure 12 Streaming multiprocessor.

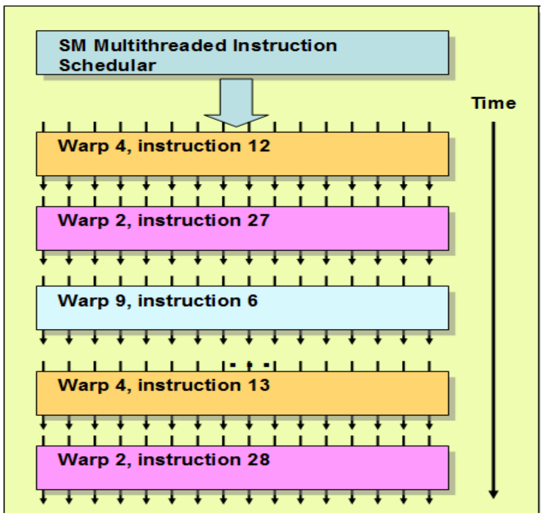


Figure 13: SIMT workflow.

## 3.4 Device (GPU) Memories

Efficient GPU parallel programming is based on proper usage of the device memory. Thus, the details of the device memory, comprised of six different memories with various characteristics, need to be known. The memory spaces illustrated in Figure 14 are all well-suited to different requirements. This section gives detailed information regarding these memory spaces and how to use them efficiently depending on implementation.

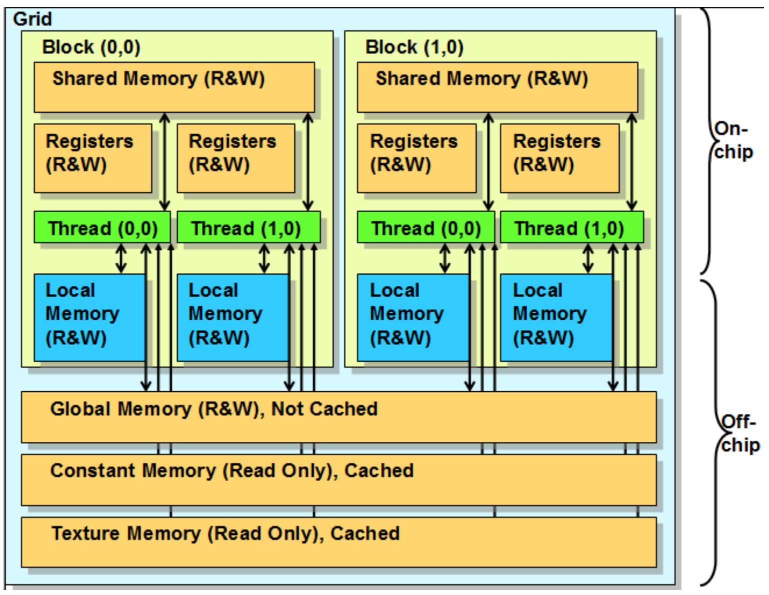


Figure 14: Device memory architecture.

- **Register:** The register is located on the chip and thus offers a fast access. However this memory is only accessible by a single thread and has limited space. For example, the GTX 200 GPU offers 16,384 32-bit registers per SM.
- **Local:** Local memory is used as extra space when registers do not meet the requirements. Since this memory space is not located on the chip and not cached, it suffers from bandwidth optimization.
- **Shared:** Shared memory is designed for performance. This on-chip memory is located close to the stream processor and offers very low-latency [9]. Shared memory is accessible by the threads within a block and much faster than the global memory. Currently the shared memory space is limited to

16KB per block. This memory space is accessible to all threads in a block. Although designed for fast access, shared memory only offers limited capacity. Thus, when memory size matters, fewer threads should be chosen over the maximum limit of 512 threads per block. In addition to fast access, shared memory also allows synchronization, required to keep threads aside for certain cases. Replacing global memory accesses with shared memory saves significant global memory bandwidth. Therefore, it is certainly worth the extra effort to re-implement algorithms to take the most advantage of shared memories.

- **Global Memory:** This off-chip memory space can be considered as the most flexible memory space. It offers both read and write operations as well as provides access to all threads and the host. Unfortunately Global memory is very ineffective for access latency. Since this memory space is not cached, coalesced memory access is required for better performance. It also does not require too much memory space.
- **Constant Memory:** This memory space is read-only and, similar to the global memory, off-chip. However, constant memory is cached and offers better performance. Currently, the available constant memory space is limited to 64KB.
- **Texture Memory:** Texture memory is actually a read-only global memory. Compared to global memory, this memory space offers better performance from a cached-structure. Texture memory allows data representations in one of these combinations:
  - Dimensionality: 1, 2 or 3.
  - Data Component: 1, 2 or 4.
  - Data Elements: Signed/Unsigned Integers (8,16 and 32-bit), Floats (16 and 32-bit)

  Read-only memory accesses should be employed via texture memory whenever possible to offer better performance.

The concurrent execution between the host and the device can be utilized by using asynchronous functions. When these functions are employed the control is returned to the host before the device complete whole process. Thus the performance of heterogeneous computation can be improved. Asynchronous tasks are listed as follows [9]:

- Kernel launches,
- Memory copy functions with Async suffix,
- Host to Device and Device to Host memory copy functions,
- Memory setting functions

It is also possible to employ page-locked host and device memory copies concurrently with kernel executions [9]. However, not all CUDA enabled devices support this functionality.

## 3.5 Enhancing Computational Performance

To improve CUDA implementation performance, many actions can be employed. However, the overall effect of each technique is unique. Some points should always be taken into consideration while the rest depend on the nature of the problem and the computational content. Non-optimized CUDA codes always suffer from performance bottleneck. Thus, special care must be given to certain issues. NVidia has defined three strategies regarding performance optimization [11]:

1. The first strategy achieves better throughput from maximizing parallel execution. The problem must be converted into data-parallel structure as often as possible. For massive crowd simulation, an actor-based solution (where each virtual character is handled as a single and autonomous agent) fits best. Thus, a thread per character can be assigned for these computations.
2. The second strategy uses the bandwidth efficiently. This strategy minimizes data transfer between the host and the device/s and uses device memories

appropriately. If possible, high-performance shared memory or cached texture memory should be chosen instead of global memory. However, due to the nature of crowd simulation, global memory is almost indispensable. Therefore, the structure of device data must be well suited to the CUDA architecture.

3. The third strategy is the optimization via instruction usage. This strategy covers choosing alternative approaches to offer high performance. For example, single precision arithmetic could be enough for character navigation. Consequently, there is no reason to use double precision which offers lower computational throughput. The instruction level optimization also minimizes branching within a warp. This ensures that all of the threads follow a similar execution path and the computational resources are used efficiently. CUDA also provides several math functions for better performance, which should be used whenever possible. The overall performance becomes significant when instruction level optimization is employed due to the huge number of characters.

## 3.6 FERMI: The Upcoming CUDA Architecture

NVidia will further improve GPU parallel computing with the upcoming FERMI architecture, which is built upon on previous G80 and G200 architectures. The new FERMI-enabled G300 GPU will introduce several advances. While designing FERMI, the developers focused on improvements such as double precision performance, error-checking, memory operations, and memory space [78, 79]. FERMI will also introduce some significant innovations to its hardware. The main innovation, the 3rd generation SM, is illustrated in Figure 15. Firstly, the number of cores in SMs will be raised from 8 to 32. The G300 GPU will contain 512 CUDA cores. The new double precision performance speedup will be 8× [78]. Similarly shared memory capacity increment will be 4 times larger, at 64 kb. The new 64 bit memory-space will also offer more GPU memory. Two other improvements worth mentioning are the 10x context switching performance and the NEXUS development environment. The details of these new improvements can be found on the NVidia FERMI white paper [78].

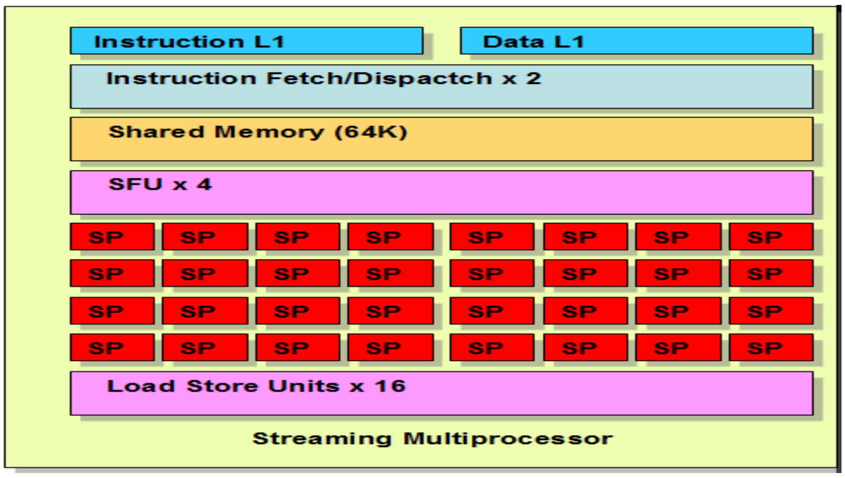


Figure 15: 3rd generation SM.

# CHAPTER 4

# MASSIVE CROWD SIMULATION WITH GENERAL-PURPOSE PARALLEL PROGRAMMING

This chapter explains how general purpose parallel programming can be used for massive crowd simulation studies in a step-by-step approach, using NVidia's CUDA technology. Firstly, a generic massive crowd simulation application, which is assumed to run on a single CPU core, will be ported to the GPU. In order to improve computational performance, various approaches will be introduced in the following sections. These approaches will utilize parallel computing basics and CUDA architecture.

## 4.1 Massive Crowd Simulation Scenario

It is assumed that over one million virtual pedestrian (actual population is 1,048,576) live in a virtual city. The aim is to simulate the navigation and behavioral modeling of these virtual people and try to improve computational performance using GPU parallel processing. This case scenario only focuses on navigation and reasoning simulation, and thus excludes rendering process. Consequently, there is no 3D graphics and no filtering operations like visibility or culling tests and LOD. In each simulation loop, the application computes the virtual character's navigation and reasoning processes. The average computational cost per character is 1200 arithmetic operations (+:560; -:80; /:240; *:240; sqrt: 80) as given by Listing 1. The aim of this generic function is to perform same amount of arithmetic operation to compare CPU and GPU performances, not to simulate realistic behavior and navigation model. Such functions were given in Chapter 5 and Chapter 6. The application performs these processes via the "processAvatar" function, which is called for all characters during each simulation loop. Table 1 lists the attributes of a virtual character.

Listing 1: Pseudocode of the "processAvatar" Function

```
for (idx =0; idx <populationCount;++ idx)
        for (i=0;i<80;++i)
                position[idx].x/=i;
                position[idx].y/=i;
                position[idx].z/=i;
                heading[idx].x-=sqrt(heading[idx])+i;
                mood[idx]+=personality[idx]+gender[idx]+age[idx]+weight[idx]+i;
                agility[idx]+=gender[idx]*age[idx]*height[idx]*i;
```

Table 1: The attributes of a virtual character.

| **Attribute Name** | **Data Type** | **Data Range** | **Constant** |
|---|---|---|---|
| position | float3 | float range | no |
| heading | float | 0-360 | no |
| age | integer | 0-120 | yes |
| height | integer | 0-255 | yes |
| weight | integer | 0-255 | yes |
| personality | integer | 0-7 | yes |
| mood | integer | 0-9 | no |
| agility | integer | 0-100 | no |
| gender | integer | 0-1 | yes |
| model_index | integer | 0-100 | yes |
| cloth_index | integer | 0-100 | yes |

## 4.2 Test Implementation Setup and Results

In this test implementation, host-side computations were done using a single CPU core (Intel I7 920 @2.67 GHz) and device-side computations were done using an NVidia GTX 295 GPU. Actually, NVidia GTX 295 GPU is made up of two GPUs. In this thesis multi-GPU has not been utilized. Thus only a single GTX 200 series GPU has been used to compare single GPU performance with single CPU core performance throughout the thesis. The speedup is almost 100% scalable when multi-GPUs are employed [9, 10]. Table 2 gives performance results of the crowd simulation implementation and the improvements. Figure 16 shows computational enhancement using logarithmic representation, since the overall speedup is nearly 475×. It can be seen that it is possible to achieve two orders of magnitude speedups using CUDA parallel computing architecture. Please note that the achieved 475× speedup is totally dependent on this simulation. However, it is even possible to achieve better speedups depending on the computational model and implementation by using functionalities such as asynchronous concurrent execution as discussed in Chapter 3.

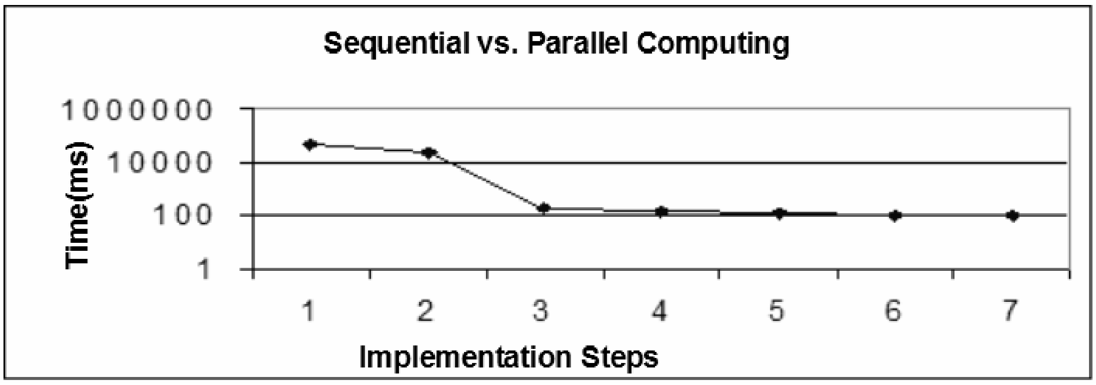


Figure 16: Speedup via parallel processing and further improvements through computational performance considerations (logarithmic representation).

The following chapters will also introduce comparable results. But, it must be noted that this test compares many-core parallel computing with single core sequential computing. Therefore, this test does not compare CPU computing with GPU computing. Such a test should compare optimized multi-thread CPU implementation with optimized GPU implementation, which is not within the scope of this thesis.

Table 2: Computational enhancement of the generic massive crowd simulation. (CPU: Intel I7 920 @2.67 GHz, GPU: . GTX 295)

| **Description** | **Time (ms)** | **Step Speedup** | **Cumulative Speedup** |
|---|---|---|---|
| **Step1 :**The simulation was computed on the CPU | 49,393.7 | - | - |
| **Step 2:** The simulation was computed on the CPU and the GPU. The computational load was shared equally. The simulation time also included host-to-device and device-to-host data transfer. Due to bad design, all attributes were transferred in each simulation loop. Memory pattern includes several non-coalescent accesses. | 22,737.4 | 2.17 | **2.17** |
| **Step 3:** The simulation was computed on the GPU. The simulation time also included host-to-device and device-to-host data transfer. Due to bad design, all attributes were transferred in each simulation loop. Memory pattern includes several non-coalescent accesses. | 183.6 | 123.84 | **269.02** |
| **Step 4:** Unnecessary attribute transfer was prevented. The design was improved by dividing data structure. Details can be found in section 4.3.4 | 128.1 | 1.43 | **385.59** |
| **Step 5:** Data compaction was used for the transferred data. In this example decode/encode cost was found 4 ms. | 113.1 | 1.13 | **436.73** |
| **Step 6:** Memory-level optimization implemented. | 104.4 | 1.08 | **473.12** |
| **Step 7:** Instruction-level optimization implemented. | 104.2 | 1.002 | **474.03** |

## 4.3 Transferring Computational Load from a CPU to a GPU

Transferring computational load from a CPU to a GPU helps achieve good speedups. This transfer actually ensures the parallelism, since the serial execution on the CPU is replaced with a highly parallel solution on the GPU. CUDA technology names the CPU as "Host" and the GPU as "Device". Hereinafter, this naming convention will be used.

As previously explained, three strategies must be employed to achieve maximum speedup. Section 4.3.1, 4.3.2 and 4.3.3 give details of the first strategy, which is to maximize parallel execution. Section 4.3.4 covers memory and bandwidth issues, while section 4.3.5 mentions the last strategy, instruction level optimization.

### 4.3.1 CPU Computing

In CPU computing, the application was assumed to run on a single CPU core. The result of this step was used as base value to compute the cumulative and step speedups. As previously mentioned, visualization issues were excluded in this example scenario and the total computational cost for a virtual character in a single simulation loop was 1200 flops. Using single CPU core (Intel i7920 @2.67 GHz), the average simulation cycle was found 49393 milliseconds. The simulation workflow is illustrated in Figure 17. The name of the simulation function is "processAvatar_CPU".

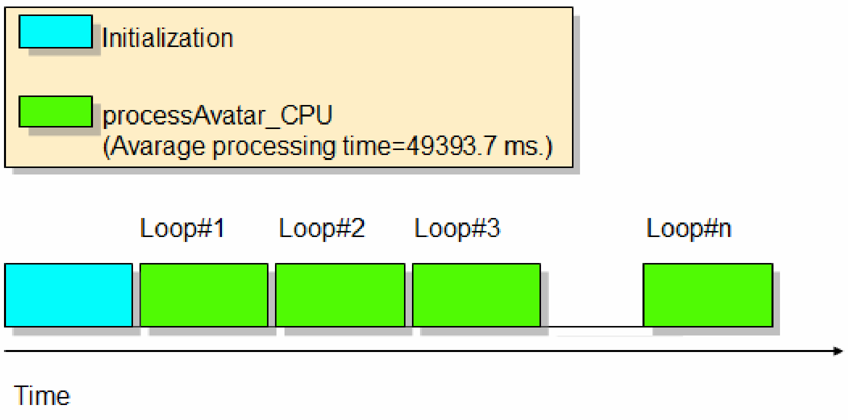


Figure 17: The workflow of the simulation running on a single CPU core.

### 4.3.2 CPU&GPU Computing

In this step, the computations were performed both on the CPU and GPU. It was assumed that programmers successfully divided "processAvatar_CPU" function into two parts ("processAvatar_CPU," and "processAvatar_GPU"), suitable to run on the CPU and GPU. The computational cost for these functions was nearly 600 flops for both. The simulation workflow illustrated in Figure 18 shows that including the GPU as a co-processor, introduced a new data-transfer overhead between the host and the device. Despite the newly introduced data transfer cost, porting even half the computational load to the GPU achieved nearly 2× speedup. In this step, the average simulation cycle was found 22737.4 milliseconds.

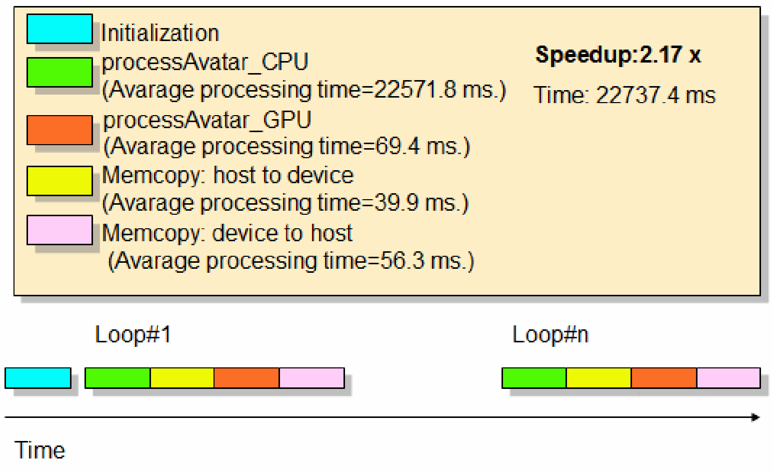


Figure 18: Workflow of the CPU and GPU computing.

### 4.3.3 GPU Computing

This step assumed that the whole simulation was computed on the GPU. Thus CPU resources were freed for other purposes. The new simulation workflow, illustrated in Figure 19, achieved 269.02× speedup, which is tremendous. Please note that this initial design was very poor due to unnecessary memory copy operations. Figure 20-

a shows that 47% of the GPU processing is occupied by memory copy operations. This issue will be fixed in the following sub-sections.

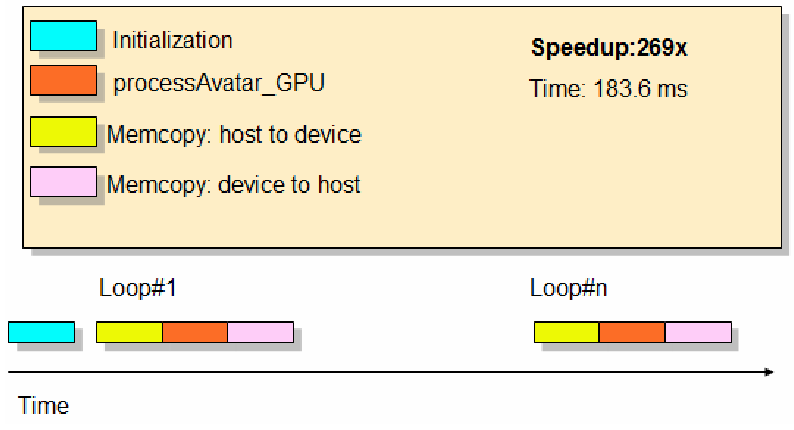


Figure 19: Workflow of the GPU computing.

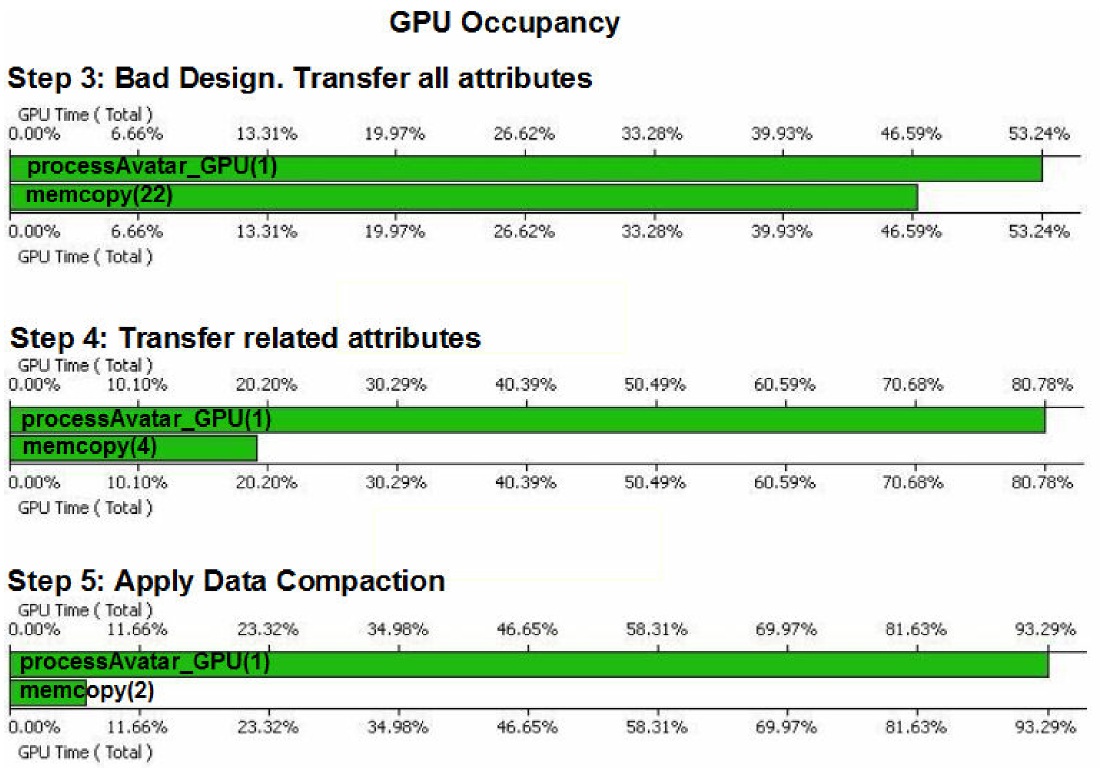


Figure 20: GPU occupancy plot.

(Top, 20-a: Shows bad CUDA implementation. Each attribute was transferred between the device and the host. Middle, 20-b: Shows improvement by eliminating the transfer of unused attributes. Bottom, 20-c: Data compaction minimizes the size of the transferred data, thus provides further enhancement).

### 4.3.4 Minimizing Data Transfer and Data Access Cost

One major bottleneck in general purpose parallel programming is the data transfer time between the host and the device. Therefore, GPU vendors try to improve bus-width. Table 3 shows the bandwidths of various CUDA capable GPUs. The previous sections showed that I/O time limits better speedups. Therefore, the size of transferred data needs to be reduced. Figure 20-a depicts the GPU occupancy of the test simulation. In each simulation loop, almost half of the GPU resources were occupied by memory copy operations. It should be noted that stream reduction makes sense when data transfer frequency is high. In other words, it is not worth trying to reduce data size in a very compute intensive application when data is transferred to the GPU only a few times because data transfer time becomes negligible compared to processing time. However crowd simulation requires data transfer with every simulation cycle or at similarly high frequencies, since it is required to update virtual characters' action and position in each frame. Thus, stream reduction is a significant issue in massive crowd simulation applications on general purpose parallel programming.

Table 3: Bandwidth of various CUDA-enabled devices.

| **Model** | **Bandwidth max (GB/s)** | **Bus width (bit)** |
|---|---|---|
| GeForce 8300 GS | 6.4 | 128/256 |
| GeForce 8800 GTX | 86.4 | 384 |
| GeForce 9500 GT | 25.6 | 128 |
| GeForce 9800 GTX | 70.4 | 256 |
| GeForce GTX 260 | 111.9 | 448 |
| GeForce GTX 295 | 2*111.9 | 2*448 |

Stream reduction, also known as stream compaction, can occur with several steps. Firstly, only the required data sets should be transferred. The data structure may contain all of a virtual character's attributes but some may not be processed on the GPU. Thus, unnecessary traffic will occupy a limited bandwidth. To accomplish this, the data structure should be divided into categories of CPU related, GPU related and CPU/GPU related, limiting the transferred data to the CPU/GPU related category.

- CPU related attributes: These attributes are usually related with visualization or pre/post processing and have no effect on GPU processed functions such as navigation or behavior modeling. Texture IDs, hair color, eye color are some of these attributes. Since transferring these attributes into the device consumes memory space and bandwidth, keeping them on the host memory helps achieving more speedup.
- GPU related attributes: These attributes are used by GPU to produce results to be used by the CPU. The mentioned attributes are transferred to the GPU at the initialization phase and stored in the device memory as long as used by GPU kernels. The transfer time in the initialization phase is negligible since it is a one-time task. However significant amount of device memory accesses are required in each simulation cycle. Thus using appropriate device memory and memory access strategy lies at the center of this optimization.
- CPU/GPU related attributes: These attributes are both used by the CPU and GPU throughout the simulation. Since there might be extensive number of transfers it becomes important to minimize the transfer time.

Figure 20-b shows the improvement achieved by preventing unnecessary memory copy operations. In this step, the constant attributes and CPU related attributes (model and texture ids) were removed from the main data structure and thus there was no need to transfer these attributes in each simulation loop.

Secondly, the stream size can be reduced with a decrease in data transfer frequency. Occasionally, some of the attributes may not be required in each simulation loop. In massive crowd simulation, an avatar's act can be modeled using combination of several attributes. The weight and refreshment rate of these attributes might be different, allowing further speedup. However, this solution is implementation dependent and requires extra coding effort.

Finally, data compression, an approach that deals with the transfer of large amounts of data using low bandwidth can provide further stream reduction. The two methods of data compression techniques are lossless and lossy compression. Lossy compression is usually employed for perceptual content such as audio and video [80]. In massive crowd simulation, lossy compression can be used for the positional data (translation and rotation values) required to be transferred frequently, since the loss of few centimeters of accuracy probably is not a problem. Lossless compression is employed when data content, such as behavioral attributes, is very important and should not be changed.

Numerous research studies dealt with minimizing the size of transferred data; some even focused on increasing the efficiency of GPU parallel processing. These GPU parallel processing studies recognize stream reduction as a well-known approach in which unnecessary data is removed from the output stream before it is transferred. Roger, Assarsson and Holzschucz implemented stream reduction in GPGPU using the parallel structure of the underlying hardware [81]. They simply divided output data and processed them in parallel, providing significant speedup compared to the sequential data reduction. Horn also studied data reduction on GPGPU to run faster collision detection on the GPU [82]. Unlike the previous studies on GPGPU, Balevic et al. employed CUDA and introduced a novel approach to reduce the size of simulation data on massively parallel GPGPUs through arithmetic coding [80]. They evaluated Huffman coding and arithmetic coding and preferred the arithmetic encoding since it does not often require non-aligned memory accesses. As previously explained, the cost of non-aligned memory accesses on the GPU side significantly reduces the computational throughput. Balevic et al. used light scattering data and improved data transfer time nearly 30× compared to the uncompressed data transfer. Although this speedup seems significant, the overall performance enhancement needs to be evaluated including the newly introduced overheads such as encoding/decoding and extra memory accesses in the GPU. They ultimately showed that arithmetic encoding significantly reduces data transfer and storage costs while ensuring no data loss. Harris, Sengupta and Owens implemented another stream reduction study, using GeForce 8800 GTX which offers native scatter on the hardware. They obtained more efficient results compared to the study of Horn that used GeForce 6800 without native scatter capability [83]. Recently, another stream reduction study reported a 3× speedup compared to the previous published algorithms using SIMD architecture and global barrier synchronization [84]. Although the authors used CUDA, they indicated that the algorithm also suits AMD GPUs and the expected Intel Larrabee, since they only used implicit atomicity in SIMD architecture and barrier synchronization [84].

Although these researchers have demonstrated various approaches for stream reduction, other data reduction techniques can be employed for massive crowd simulations on the GPU. The case study can better explain this approach. As seen in Table 1, the GPU-updated attributes are mostly integer scalars between 0-100 to represent output of the behavioral model. Instead of using 32-bit integers to transfer these attributes, shorter bit representations can be used. Similar bit reduction can also be applied for position or angular information. For example for avatar direction, 9 bits are enough to cover 360 degrees since rotations less than a degree does not matter in visualization. Similarly, when elevation difference is not large, two less bits can be employed instead of 32-bit float. The approach can be named as bit compaction. The data size reduction becomes significant in highly populated crowd simulations. The first advantage of this approach is the minimal cost of the newly introduced decode and encode operations performed on the CPU and the GPU respectively. Unlike the previously mentioned compression techniques, bit level extraction and insertion introduce almost no overhead compared to the other algorithms. Another advantage is the minimization of data access operations on the GPU, which is costly especially when using global device memory. The global device memory access cost increases when non-coalesced data structures are used. For example using a data structure comprised of 7 integer values significantly reduces GPU computing performance since this structure does not ensure coalesced memory access as previously described. Since any four-byte or eight-byte data structures are processed in single instruction, converting reduced data structure into a four-byte or eight-byte data structure is better if the reduced data size is shorter. In this case, a 29-bit can be inserted into a 32-bit envelope by just padding 0s to the end. Since this would be part of existing decode/encode functions, almost no instruction cost is introduced.

Figure 20-c shows the additional speedup achieved by just employing this simple bit compaction technique. The achieved step speedup and cumulative speedup (1.13 and 436.73 respectively) show that this approach is certainly worth implementing in cases where fewer bits represent the same integer values. Equation 1 shows the case when data reduction should be employed.

$$T_{dataTransfer} + T_{memoryAccess} > \widetilde{T}_{dataTransfer} + T_{decode} + T_{encode} + \widetilde{T}_{memoryAccess} \qquad \text{(Equation 1)}$$

where,

$T_{dataTransfer}$ denotes the uncompressed data transfer time (device→host),

$T_{memoryAccess}$ denotes the memory access time for uncompressed data at the GPU,

$\widetilde{T}_{dataTransfer}$ denotes the compressed data transfer time (device→host),

$T_{decode}$ denotes decode time of the compressed data on the CPU,

$T_{encode}$ denotes decode time of the compressed data on the GPU,

$\widetilde{T}_{memoryAccess}$ denotes the memory access time for compressed data at the GPU.

To the best of the author’s knowledge, the above mentioned bit compaction technique, which can be evaluated in means of data transfer and providing aligned memory access, has not been offered for massive crowd simulation studies using parallel processing. The simplicity of the proposed approach is the main advantage while the main disadvantage is the dependency on the implementation since this approach only works when many small integer numbers are required to transfer between the device and the host frequently. Fortunately, this is not an unusual case in massive crowd simulation studies, and this approach is one of the novelties within this thesis.

### 4.3.5 Using Device Memory Efficiently

In the previous chapter, the specifications of the six device memories were discussed in detail. Using shared memories becomes critical when excessive use of global memory can be replaced. When global memory is employed, providing coalesced memory accesses ensures extra speedup. In CUDA architecture the global memory access costs one or two memory transactions for all threads of a half-warp (16 threads) depending on the bit-length,. However, for non-coalesced access patterns the bandwidth is around an order of magnitude lower than coalesced patterns [9]. This issue becomes significant in massive crowd simulation, where excessive memory access requirement is very common. In this simulation, using coalesced pattern further improvement was achieved (step speedup: 1.08, cumulative speedup: 473.12).

### 4.3.6 Instruction Level Optimization

Although instruction level optimization does not provide as noticeable speedups, it is easier to implement and, depending on the content and accuracy expectations, it is definitely worth trying. The main rule of thumb for this type of optimization is to prefer single-precision. Therefore, double-precision arithmetic must be avoided whenever possible; however single-precision is definitely enough for all crowd simulation applications.

To improve computational performance, CUDA architecture offers several computationally low-cost functions that easily replace existing operators or functions. One of these utilities is ***__fdividef(x, y)***, which performs division twice as fast as the standard single-precision floating point division [9]. Similarly, ***__sincosf(x,sptr,cptr)*** performs better than the respective trigonometric functions. The details of such utilities can be found in the CUDA reference guide [9].

Instruction level optimization is not only limited to the type of precision or utility functions. The computational performance enhancement tricks can be employed in GPU parallel programming as well. A typical example is the use of the squared distance metric to avoid expensive square root operations. Similarly, bit-wise operations should be used whenever possible instead of division operation [9].

# CHAPTER 5

# FUZZY INFERENCE WITH GPU PARALLEL PROCESSING

As previously mentioned, one of the contributions of this study is high-performance fuzzy inference with CUDA, which utilizes GPU parallel processing architecture. Design considerations provided significant computational performance. For example, it is possible to make nearly half billion fuzzy inferences per second, using single GTX 295 GPU, 300× faster than an average CPU core.

The reasons for choosing fuzzy logic for behavioral modeling or simulating some actions for massive crowd simulation are as follows:

- Ability to produce realistic and less predictable reactions.
- Ability to capture a real human knowledge-base and use it extensively with minimal coding.
- Use of an AI technique that is more suitable to model complex virtual character behavior.

This chapter summarizes the background of fuzzy logic and explains the implementation details.

## 5.1 Fuzzy Logic

Fuzzy sets were first introduced by Lotfi Zadeh in 1965 [85], and over the last 40 years, fuzzy logic has been used extensively in many application areas, including crowd simulation [86-88] and agent behavior modeling [59,89]. As stated by Zadeh, imprecise inputs are important for human thinking, or other perceptual actions such as pattern recognition, communication and abstraction. For example, humans can easily process the imprecise information such as tall, very tall, hot, and too hot. However, computers require precise inputs and programmers are very familiar with this yes/no (Boolean) logic. In classical set theory (crisp) there is a distinct borderline between two categories. As shown in Figure 21-a, a human is either tall (>1.80 cm) or short (<1.80 cm). Thus, there is no difference between two men who are 1.79 cm. and 1.47 cm. tall respectively, since they are both considered as short. A fuzzy set has been generated when the borderline is modified to reflect fuzziness, as shown in Figure 21-b. In fuzzy set theory, there is no Boolean logic. Instead, multi-valued logic is used (a membership degree between 0 and 1). In Figure 21-b, for a given height of 1.79 cm, the membership for "Fuzzy Set Tall" is found 0.4. This computation is done by using Equation 2.

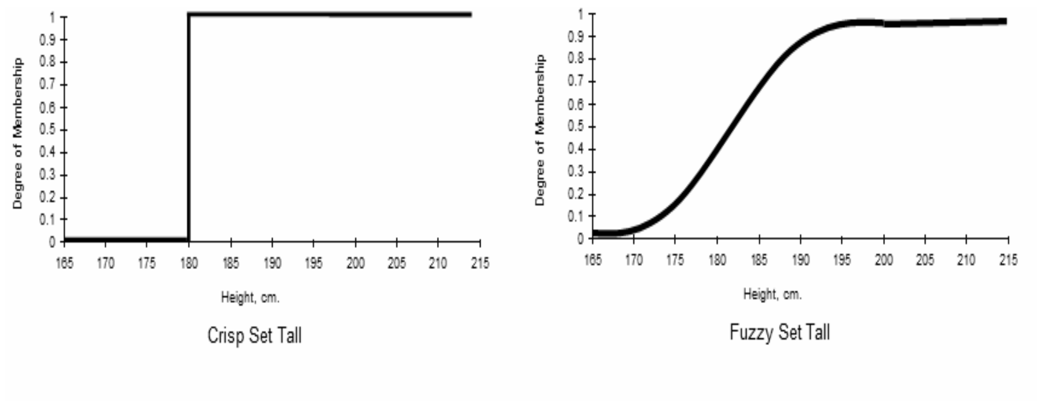


Figure 21: Classical set for tall (left: a) and tall fuzzy set (right: b).

$$\delta = \mu(x) = \begin{cases} 0 & \textit{if } x \textit{ is not in } A \\ (0,1) & \textit{if } x \textit{ is partially in } A \\ 1 & \textit{if } x \textit{ is totally in } A \end{cases} \qquad \text{(Equation 2)}$$

where,

$\delta$ denotes the degree of membership, $\mu(.)$ denotes the fuzzy function

$x$ denotes the input value and $A$ denotes the fuzzy set.

As shown in Figure 21-b, the given function (μ) is continuous. However, as stated by De Byl, such continuous functions are not efficient in means of computational cost [90]. Therefore, it is better to represent these functions with corresponding linear fit functions, as depicted by Figure 22.

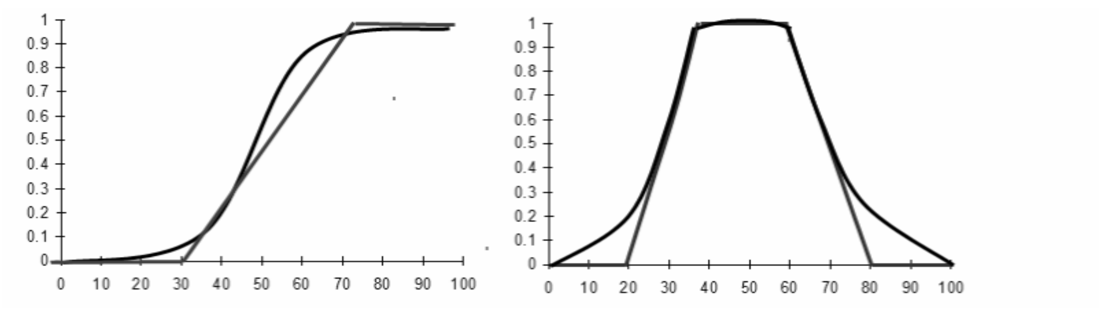


Figure 22: Typical fuzzy continuos functions (black) and respective linear-fit functions (red).

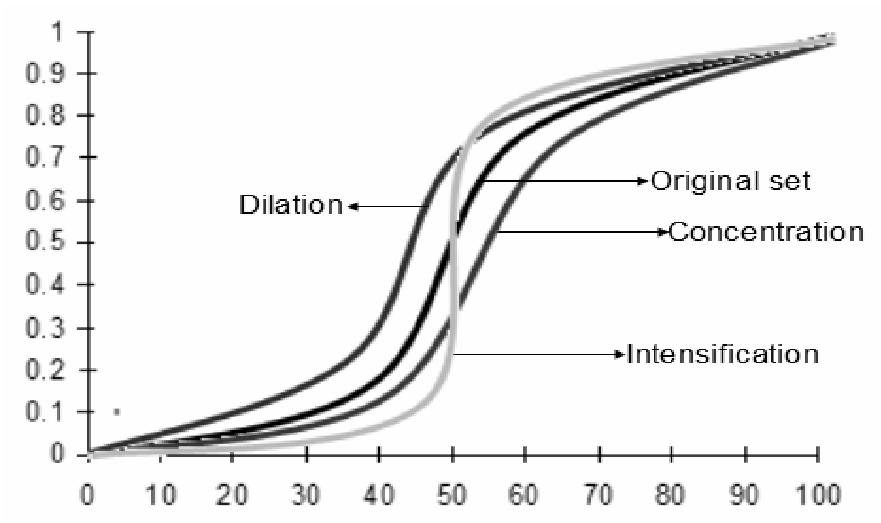


Figure 23: Modification of the linguistic hedges.

It is also possible to modify the shape of the fuzzy sets by using linguistic variables like adjectives and adverbs that are used in daily life. Typical examples are "very", "somewhat", "fairly", "slightly", and "moderately". These adverbs are called as Linguistic Hedges which dilate, concentrate or intensify the original fuzzy set [91].

These modifications are depicted by Figure 23. Dilation stretches a fuzzy set by increasing the membership, while concentration does the opposite. Intensification behaves like the combination of the dilation and concentration. Such hedges dilate the membership if the original membership is less than 0.5 or concentrate the membership if the original membership is greater than 0.5. For example, the hedge "very" modifies the fuzzy set $[\mu(x)]^2$, while the hedge "very, very" modifies $[\mu(x)]^4$. Similarly, "somewhat" can be equated to $[\mu(x)]^{1/2}$. Table 4 lists some of the popular linguistic hedges.

Table 4: Linguistic hedges.

| **Name** | **Equation** | **Effect** |
|---|---|---|
| very | $[\mu(x)]^2$ | Dilation |
| very, very | $[\mu(x)]^4$ | Dilation |
| plus | $[\mu(x)]^{1.25}$ | Dilation |
| extremely | $[\mu(x)]^3$ | Dilation |
| slightly or somewhat | $[\mu(x)]^{0.5}$ | Concentration |
| minus | $[\mu(x)]^{0.75}$ | Concentration |
| indeed | if $0 \leq \mu \leq 0.5$<br>$2[\mu(x)]^2$<br>if $0.5 < \mu \leq 1$<br>$1-2[1-\mu(x)]^2$ | Intensification |

## 5.2 Fuzzy Inference

Fuzzy inference simply performs a processing by using scalar input values and fuzzy control elements (fuzzy variables, fuzzy sets and fuzzy rules), and produces an output represented by a scalar value. There are three popular fuzzy inference methods: Mamdani [92], Sugeno [93], and Tsukamoto [94]. Among these three methods, Mamdani-style is commonly applied. Thus, in this study, a fuzzy controller approach proposed by Mamdani has been implemented for fuzzy inference process. A

Mamdani-style fuzzy inference is a four-step process (see Figure 24) given as follows [90]:

- Fuzzification
- Rule Evaluation
- Rule Aggregation
- Defuzzification

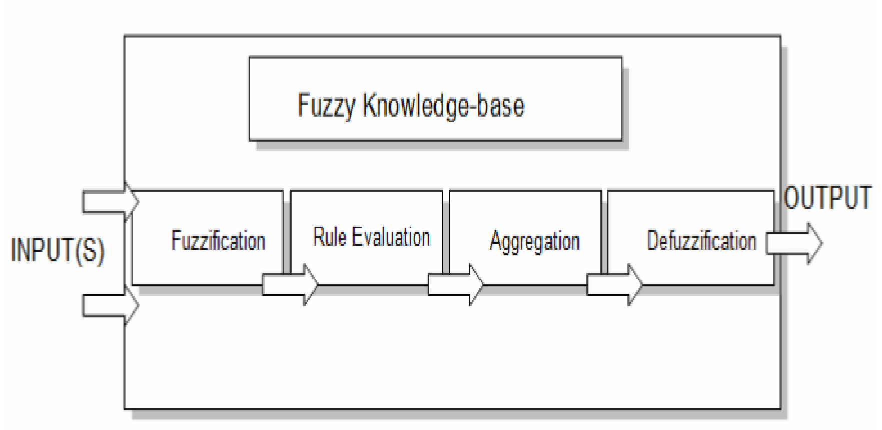


Figure 24: Mamdani-style fuzzy inference.

To better explain the Mamdani style fuzzy inference, an example is provided. It is assumed that the fuzzy inference is used to determine enthusiasm level of the soccer spectators when a goal is scored. Although there exists numerous parameters to define enthusiasm-level, the inference is built upon the significance of the game and the quality of the goal. Clearly a goal makes the fans happy. However, not every goal makes the same effect. For example, a goal scored in the last minute of the championships final is not the same with a goal scored in season preparation match played against a very weak opponent. Additionally, a very low quality goal by chance is not equal to a reverse shot goal. This list can be extended with more examples. The following rules (Table 5) and fuzzy sets (Figure 25) will be used to explain the four steps of Mamdani style inference.

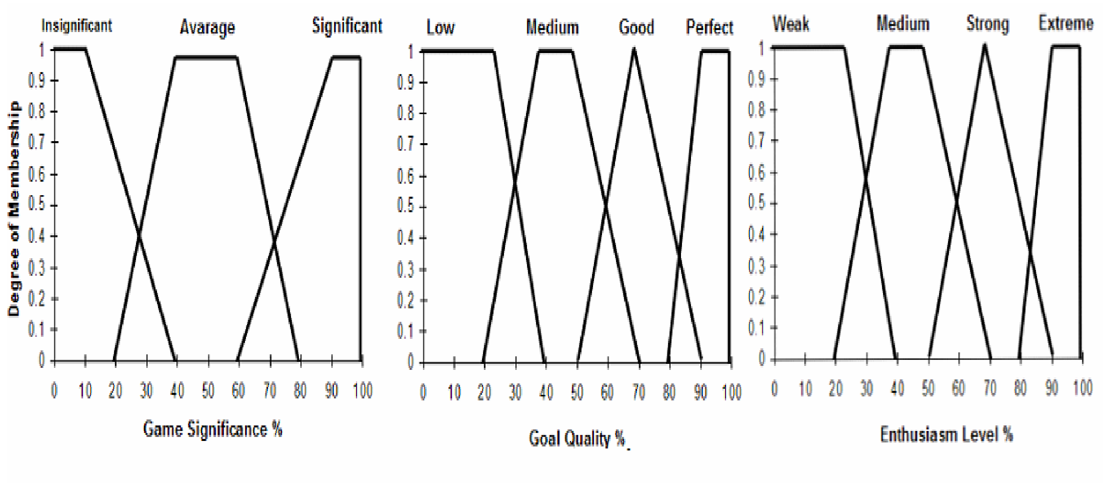


Figure 25: Fuzzy sets for soccer spectator example.

Table 5: Fuzzy knowledge-base for soccer spectator example.

| **Rule Number** | **Rule** | **Input#1** | **Fuzzy Operator** | **Input #2** | **Output** |
|---|---|---|---|---|---|
| 1 | If | **Game** is **Significant** | **AND** | **Goal quality** is **Perfect** | then **Enthusiasm** is **Extreme** |
| 2 | If | **Game** is **Average** | **OR** | **Goal quality** is **Medium** | then **Enthusiasm** is **Average** |
| 3 | If | **Game** is **Insignificant** | **AND** | **Goal quality** is **Low** | then **Enthusiasm** is **Low** |
| 4 | If | **Game** is **Significant** | **OR** | **Goal quality** is **Good** | then **Enthusiasm** is **Strong** |

### 5.2.1 Fuzzification

Fuzzification is defined as the process of making a scalar quantity fuzzy [91]. This step takes input value(s) and obtains corresponding degree(s) of membership to the related fuzzy set(s). The employed approach is illustrated in Figure 26. The input value is limited to the range of x-axis of the related fuzzy sets, while the degree of membership is always between 0 and 1. For example if the goal quality is 63 (input value), the corresponding degrees of membership are: 0 for low, 0.35 for medium, 0.65 for good, and 0 for perfect fuzzy sets.

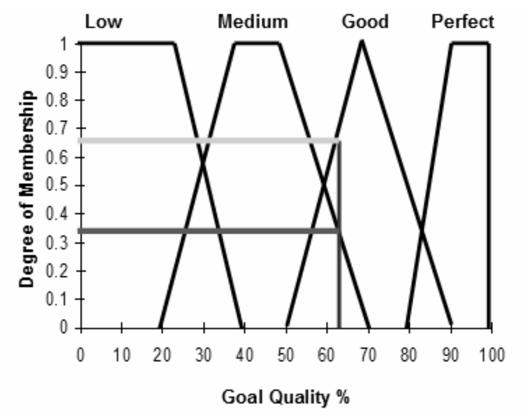


Figure 26: Fuzzification process.

### 5.2.2 Rule Evaluation

Rule evaluation is a two-step operation. The first step is taking the fuzzified values obtained from the previous step and choosing one of them according to given fuzzy operator. The two most commonly used fuzzy operators are: "AND" and "OR" operators. The fuzzy "AND" operator selects the minimum, while the fuzzy "OR" does the opposite and picks the maximum. Thus, this step generates a single output value from the given membership values. The second step is to generate a subset from the corresponding fuzzy set which is written in the else part of the rule. The output value obtained in the first step is used to cut-off the original fuzzy set to make a new subset. This new sub-set is also known as alpha-cut [91]. Rule evaluation of the fourth rule is depicted by Figure 27. In this example the input value for goal quality is 63% and the input value for game significance is 85%.

### 5.2.3 Aggregation

In this step, the fuzzy subsets (alpha-cuts) generated in the previous section are aggregated as illustrated in Figure 28. Therefore, a new fuzzy set which will be used for fuzzy decision making is obtained. This new set is the union of the cut-off fuzzy sets (the sets included in fuzzy rules) according to given fuzzy rules. Since the aggregation step is built upon union operation, the order of rule evaluation has no effect on the output fuzzy set.

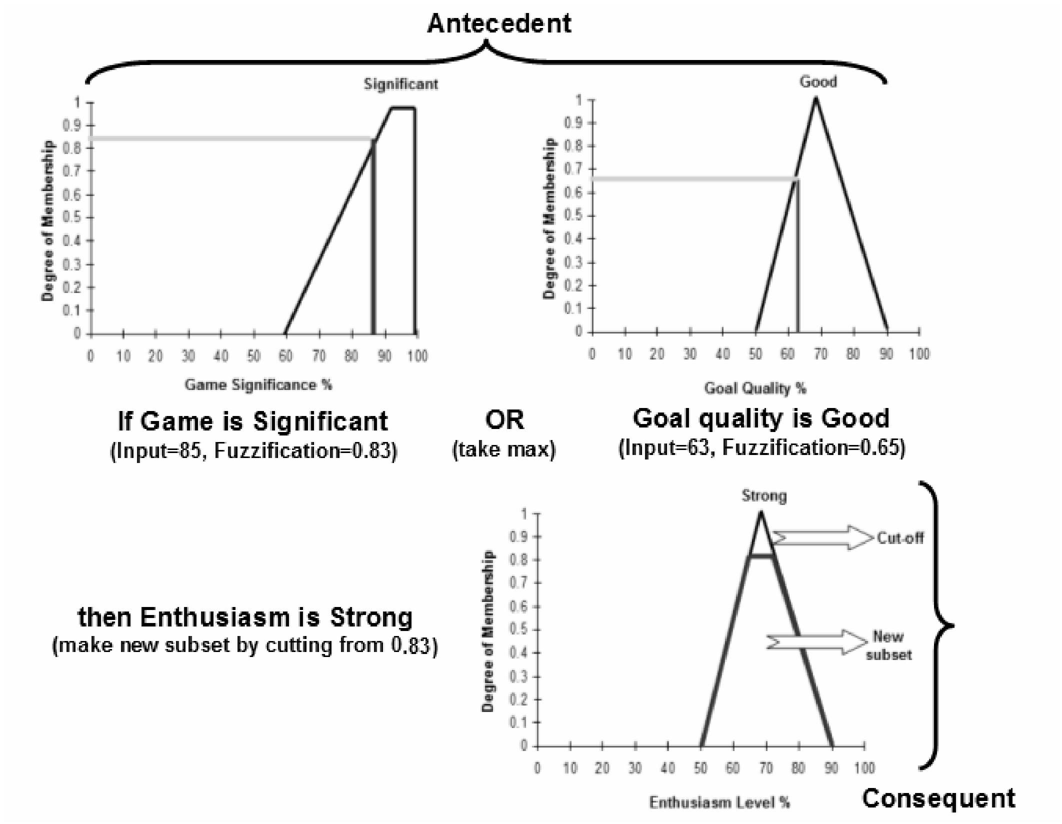


Figure 27: Rule evaluation.

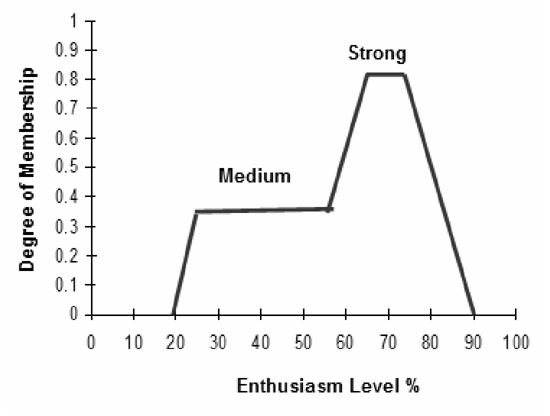


Figure 28: Aggregation.

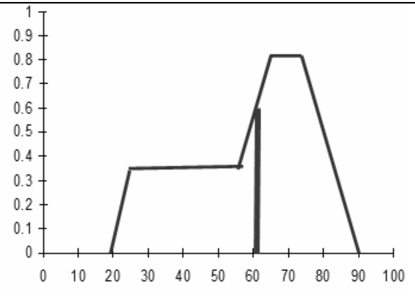
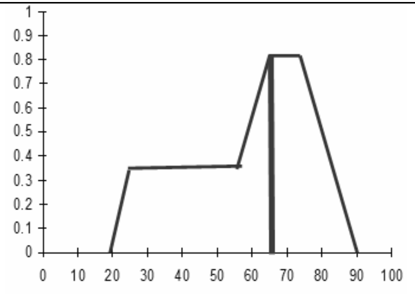
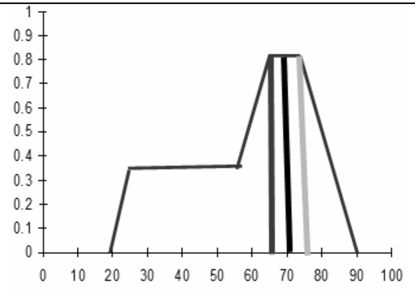
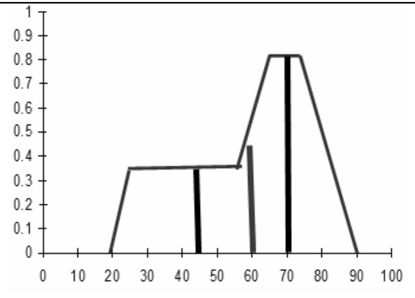
The fourth rule given in Table 6 is evaluated using the given input values; 63% for goal quality and 85% for game significance. The first and the third rules have no impact on the aggregation, since the result of fuzzification is 0 for these rules. The fuzzification results for the second and the fourth rules are 0.35 and 0.83, respectively. Considering these results the above given new fuzzy set is aggregated.

### 5.2.4 Defuzzification to a Scalar

The final step is to produce a scalar value using the aggregated fuzzy set generated in the previous step. This scalar quantity is the final output of the fuzzy inference process and may easily be used by the application or a tool that needs the output in this format. Therefore, defuzzification can be defined as conversion of a fuzzy set to a precise quantity [91]. To produce a scalar quantity from a fuzzy set, there exist various approaches. Probably the most common defuzzification method is getting the center of the given shape, called centroid, given by Equation 3. Using this equation, the result for the example fuzzy inference regarding enthusiasm level is computed as 61%.

$$I = \frac{\sum \mu_I x}{\sum \mu_I} \qquad \text{(Equation 3)}$$

Table 6: Well-known defuzzification methods.

| **Name** | **Description** | **Illustration** |
|---|---|---|
| Centroid | Gives the center of the fuzzy set. This method is also referred as Center of Gravity (CoG) or Center of Area (CoA). |  |
| Maximum Membership | As the name indicates this method gives the maximum membership value for defuzzification. |  |
| First Maxima, Last Maxima, and Mean Maximum Membership | First Maxima (blue) and Last Maxima (green) gives the first and the last maximum values, respectively. Mean Maximum Membership (black) gives the average of the maximum values. |  |
| Weighted Average | This method takes the weighted average of independent subsets, not the union. In the illustration the result is shown with blue line. |  |

Popular defuzzification approaches and their functionalities are given in Table 6. The current implementation only supports centroid, the others are considered as future work. Although the centroid method gives an approximate result due to discrete computation, it ensures computationally efficient solution which is crucial for real-time massive crowd simulation.

## 5.3 GPU Implementation

To employ fuzzy inference on a CPU using traditional sequential programming, there exist several software libraries, third party components or similar tools. However, these tools could not be used directly in CUDA implementation. Thus a fuzzy inference on the GPU was written from scratch. To achieve better computational performance, several design considerations were taken into account. For example, a fuzzy rule was supposed to consist of two input sets, a fuzzy operator and a resultant set. Additionally, rule-base, knowledge-base, fuzzy sets and fuzzy rules are stored within an array of data structures that were comprised of sequential scalar values.

GPU implementation starts with the transfer of rule-base and knowledge-base to the device. This operation simply passes the related values to the device's constant memory. Employing device constant memory significantly improves computational time, considering the other alternative, passing fuzzy knowledge-base to the main kernel. Such an approach may cause non-coalesced data load, and thus result poor performance. Fuzzy inference on the GPU is carried out via several kernels given by Listing 2. These kernels are designed for computational efficiency. The name of the main kernel is "fuzzyInference" which takes an index (indicates the fuzzy inference), a set of input values, fuzzy set group index (a combined array) and input count. The index is used to get the required information such as fuzzy rules, fuzzy sets and operators from constant memory. The count parameter indicates the number of input values and the index parameter(s) to map the given value with the corresponding fuzzy set group.  Fuzzy inference process is started by calling "fuzzyInference" kernel, and the returned value is the output of the fuzzy inference. This kernel calls "evaluateRule" kernel for each fuzzy rule. The "evaluateRule" kernel takes the corresponding input values and calls "getMembership" kernel to compute membership values. Consequently aggregation is performed and in the end "getCentroid" kernel is used to obtain final output.

Listing 2: Pseudo code of Fuzzy Inference Kernels

```
fuzzyInference //main GPU kernel for fuzzy inference
  for each rule
    evaluateRule //an independent GPU kernel that performs rule evaluation.
      for each FuzzySet
        getMembership // an independent GPU kernel that computes the degree of
        membership
aggregate // an independent GPU kernel that performs aggregation
getCentroid //an independent GPU kernel that performs defuzzification
```

Although the fuzzy inference on GPU looks cryptic, the computational performance seems impressive. Table 7 gives results of fuzzy inference test, run on different CPUs and GPUs. Each fuzzy inference operation performs a Mamdani-style inference by evaluating three fuzzy rules. The results show that GTX 295 GPU is capable of making more than half-billion fuzzy inferences per second, almost 300× of an average CPU core. The CPU and the GPU test functions are almost identical and optimized for high-performance. Please note that this performance test does not include CPU-GPU data transfer.

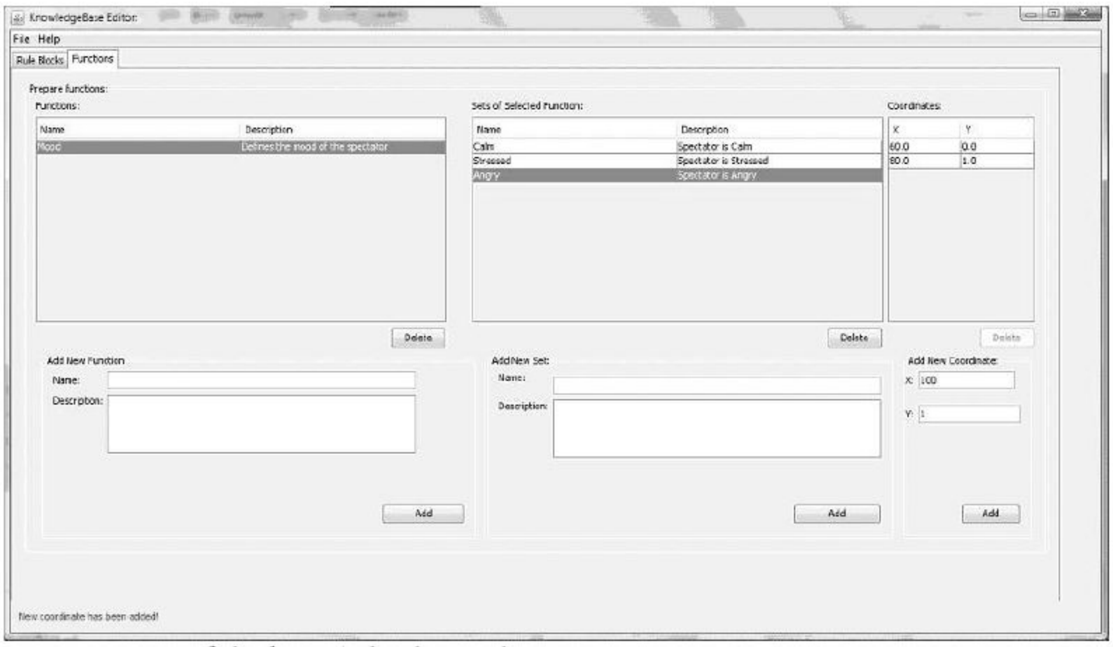
## 5.4 Capturing Knowledge Base

The success and the reality of the fuzzy inference is strongly related with the reality and content of the underlying knowledge-base, provided by human domain expert. Therefore, it is necessary to capture this knowledge and convert it into a format that can be used by fuzzy inference tool. It is possible for programmers to include fuzzy knowledge-base by hard coding. However, this approach is not practical, because even a minor revision requires recompilation of the source code. Furthermore, this approach requires good programming skills, especially when complicated fuzzy knowledge is employed.

Table 7: Fuzzy inference test results.

| **Processing Unit** | **Time required to make 1,048,576 inferences (millisecond)** | **Number of inferences per second** |
|---|---|---|
| GeForce GT 120M | 14.85 | 70,611,179 (~ 70 million) |
| GeForce 9500GT ( | 13.36 | 78,486,227 (~ 80 million) |
| GeForce GTX 295 | 1.863 | 562,842,727 (~ 560 million) |
| Intel 920 @ 2.67 GHz (Single CPU Core) | 671 | 1,562,706 (~ 1.5 million) |
| Intel T 9550 @ 2.67 GHz. (Single CPU Core) | 601 | 1,744,719 (~ 1.8 million) |
| Intel E 9550 @ 2.34 GHz. (Single CPU Core) | 704 | 1,489,455 (~ 1.5 million) |

Another approach is to capture domain expert's knowledge via predefined syntax and rules by using simple text file or better organized XML script. International Electrotechnical Commission (IEC) defined a text-based Fuzzy Control Language (FCL), which became popular for fuzzy inference applications [95]. FCL helps creating knowledge-base and defines fuzzy rules as well as choosing fuzzy operators and defuzzification method among several options. Acampora and Loia proposed Fuzzy Markup Language (FML) which offers functionality similar to the FCL, but in a more organized way due to the employment of XML structure [96]. In this thesis an XML script is proposed, which is partly inspired by the legacy FCL and by the FML. These studies were used as templates to maintain a similar terminology to the existing systems. The proposed XML script is designed to be captured via a user friendly GUI as shown in Figure 29. The captured content is converted into values that can be transferred to the GPU and can be used by GPU fuzzy inference kernels. Figure 30 and Figure 31 show XML structure of a sample fuzzy knowledge-base.



Prepare functions:

Functions:

| Name | Description |
|---|---|
| Mood | Defines the mood of the spectator |

Sets of Selected Function:

| Name | Description |
|---|---|
| Calm | Spectator is Calm |
| Stressed | Spectator is Stressed |
| Angry | Spectator is Angry |

Coordinates:

| X | Y |
|---|---|
| 60.0 | 0.0 |
| 80.0 | 1.0 |

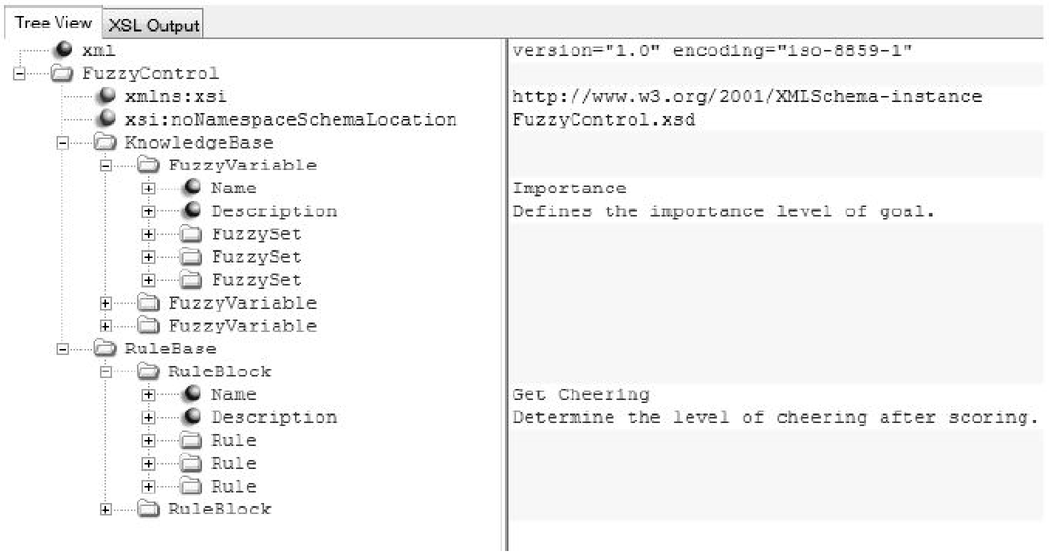
Add New Function
Name:
Description:

AddNew Set:
Name:
Description:

Add New Coordinate:
X: 100
Y: 1

New coordinate has been added!

Figure 29: A GUI to capture human expert's knowledge.



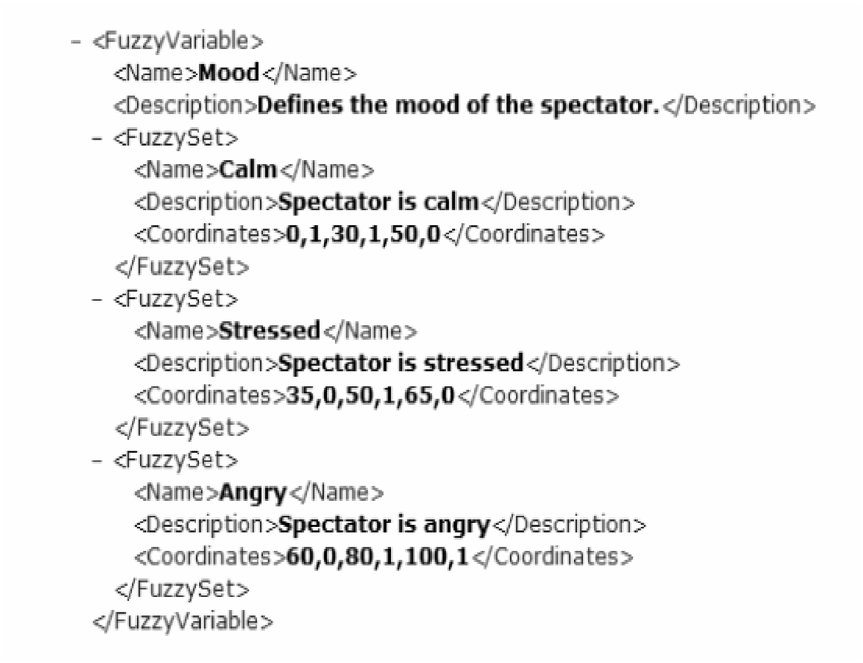
```
xml                                   version="1.0" encoding="iso-8859-1"
FuzzyControl
  xmlns:xsi                           http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation       FuzzyControl.xsd
  KnowledgeBase
    FuzzyVariable
      Name                            Importance
      Description                     Defines the importance level of goal.
      FuzzySet
      FuzzySet
      FuzzySet
    FuzzyVariable
    FuzzyVariable
  RuleBase
    RuleBlock
      Name                            Get Cheering
      Description                     Determine the level of cheering after scoring.
      Rule
      Rule
      Rule
    RuleBlock
```

Figure 30: XML structure in tree view.

The "KnowledgeBase" contains a single, or a set of, "FuzzyVariable(s)" which corresponds to a term used in fuzzy inference operation. This fuzzy element, which may be composed of a single or several “FuzzySet(s)” has “Name” and “Description” attributes. The shapes of the fuzzy sets are given within the “Coordinates” tag. The “RuleBase” section covers “RuleBlock(s)”, with each “RuleBlock” containing only two input terms and one output term. This is preferred for the sake of easy implementation. The rules within the “RuleBlock” are evaluated using only fuzzy “AND” and fuzzy “OR” operators. The details and hierarchy of XML elements are given Table 8.

Table 8: XML elements.

| **Name** | **Description** | **Attributes** |
|---|---|---|
| FuzzyControl | The container for Knowledgebase and Rulebase | |
| KnowledgeBase | A container for fuzzy varible(s) | - |
| FuzzyVariable | Defines a single fuzzy variable, comprised of fuzzy set(s). FuzzyVariable is a container of fuzzy set(s) | Name:<br>Description: |
| FuzzySet | Defines a single fuzzy set. | Name:<br>Description:<br>Coordinates: |
| Coordinates: | Defines the shape of the fuzzy set. The vertices are given in order (x-y pairs). | |
| Rulebase | A container for fuzzy Ruleblock(s) | Name:<br>Description: |
| Ruleblock | A container for fuzzy rule(s) | Name:<br>Description: |
| Rule | Defines a fuzzy rule and contains input(s), operator and output. | |
| Operator | Indicates which operator is supposed to be used. Two options available: AND/OR. | |
| Input | Refers to input FuzzySet and container FuzzyVariable elements. | Function:<br>Set: |
| Output | Refers to output FuzzySet and container FuzzyVariable elements. | Function:<br>Set: |



```
- <FuzzyVariable>
    <Name>Mood</Name>
    <Description>Defines the mood of the spectator.</Description>
  - <FuzzySet>
      <Name>Calm</Name>
      <Description>Spectator is calm</Description>
      <Coordinates>0,1,30,1,50,0</Coordinates>
    </FuzzySet>
  - <FuzzySet>
      <Name>Stressed</Name>
      <Description>Spectator is stressed</Description>
      <Coordinates>35,0,50,1,65,0</Coordinates>
    </FuzzySet>
  - <FuzzySet>
      <Name>Angry</Name>
      <Description>Spectator is angry</Description>
      <Coordinates>60,0,80,1,100,1</Coordinates>
    </FuzzySet>
  </FuzzyVariable>
```

Figure 31: XML structure in text view.

# CHAPTER 6

# CASE STUDIES

This chapter covers three case studies that illustrate the use of GPU parallel processing for real-time massive crowd simulation. The first case study simulates and visualizes sports game spectators, and the second simulates a medieval era combat field. The last case study illustrates a virtual marathon event. These case studies deal with thousands or even hundreds of thousands of virtual characters.

## 6.1 12th Man

This case study computes behaviors of the soccer spectators on the GPU then transfers the results back to the CPU to be used by an existing game engine. It is assumed that a game engine that is capable of visualizing spectator actions using scalar values provided by the application or another third party tool handles all the renderings. The game engine processes the scalar values representing the virtual character's current emotion or behavior and converts the values into an action. This case study puts the spectator's computational load on the GPU providing more CPU resources to the game engine.

### 6.1.1 Motivation and Background

Soccer, due to its very nature, has always attracted millions of fans, not only in the real world, but also in the virtual world. Spectators play a highly significant role during real soccer matches, and it has been shown that the support of an excited crowd can offer significant advantages to a home team [97]. The term "12th man" is often used to highlight the importance of fan support. Sports-based game developers have always taken spectators into account, visualizing them either as static bitmap images or dynamic geometric models since the early 1980s. In Figure 32 and Figure 33, two screenshots from the online soccer game "I Can Football" show the representation of virtual spectators in a typical soccer game; the spectators visualized in this game are similar to those found at most popular soccer games.

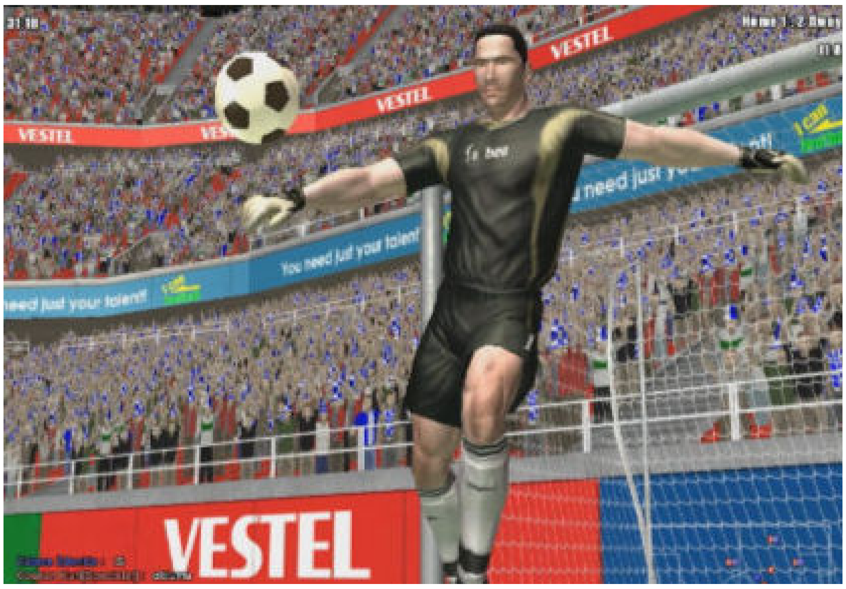


Figure 32: Comparison of the visuals of spectators and players (courtesy of Sobee).

Recently, significant progress has been achieved in spectator realism; however, the visual and behavioral realism of the spectators still lags far behind that of the players as shown in Figure 32 and Figure 33. Two priorities have taken from the realism of the spectators: the prioritization of limited computational resources for the field of play, and the prioritization of development efforts in the game field, which takes precedence over action in the background. Because of these priorities, spectators should demand very small CPU resource and also very little development effort.

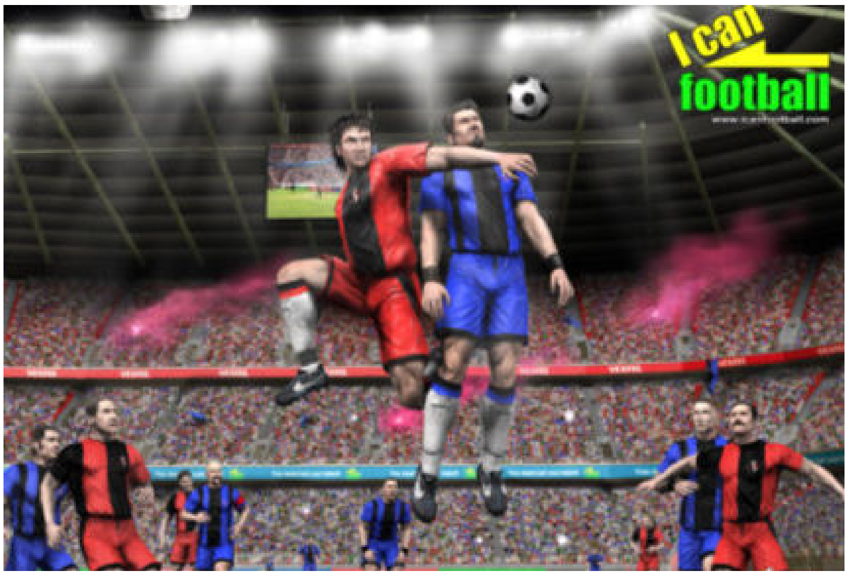


Figure 33: Comparison of the visuals of spectators and players.

### 6.1.2 Spectator Behavior Engine

In this case study, a software module was designed to compute spectator using an agent-based approach. Each spectator was considered as an independent individual. This module is called as Spectator Behavior Engine (SBE) and designed to run on the GPU using NVidia CUDA technology. Figure 34 shows the interaction of SBE with an existing game engine.

Firstly, the game engine initializes SBE and transfers spectator attributes to the device via "cudaMemcpy" function. The fuzzy knowledge-base is also transferred to the device's constant memory to improve computational performance, as discussed in Chapter 5. To be processed by the SBE, a spectator must have several distinct attributes. These attributes are related with behavioral modeling and transferred to the GPU during the initialization of the application. Thus the spectator related non-graphical attributes are stored on the device memory during run-time. This action saves significant time, since it is not required to occupy bandwidth to transfer these attributes. The fuzzy inference engine uses these attributes as spectator dependent local fuzzy inputs (see Figure 34).

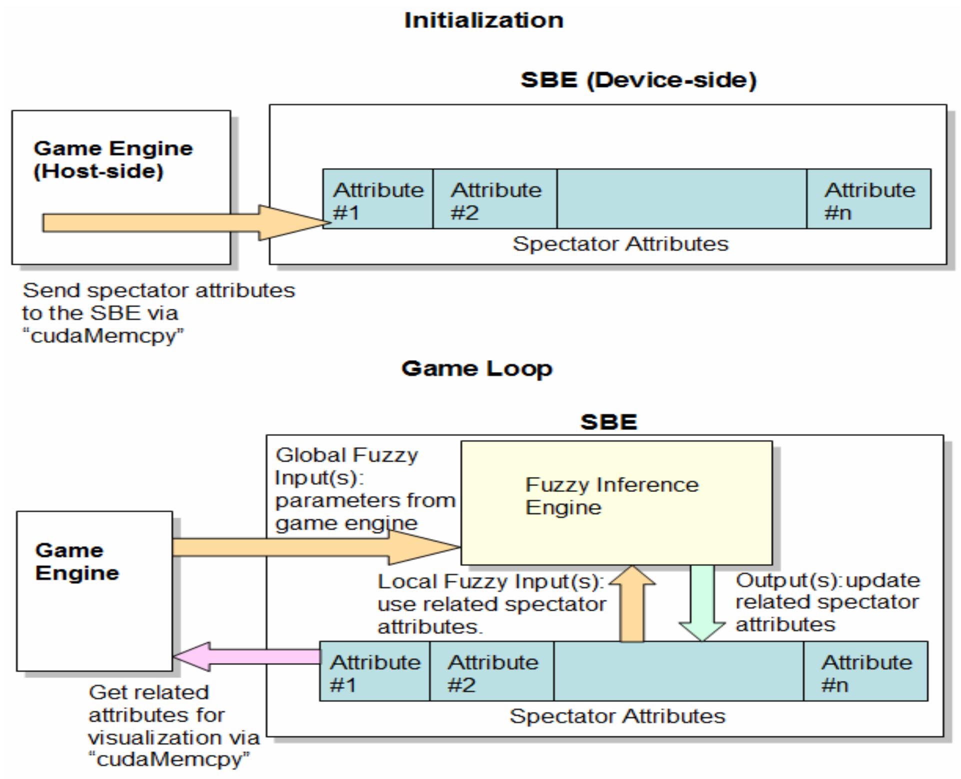

Initialization
SBE (Device-side)
Game Engine (Host-side)
Attribute #1
Attribute #2
Attribute #n
Spectator Attributes
Send spectator attributes to the SBE via "cudaMemcpy"
Game Loop
SBE
Global Fuzzy Input(s): parameters from game engine
Fuzzy Inference Engine
Game Engine
Local Fuzzy Input(s): use related spectator attributes.
Output(s): update related spectator attributes
Attribute #1
Attribute #2
Attribute #n
Spectator Attributes
Get related attributes for visualization via "cudaMemcpy"

Figure 34: SBE architecture.

The game engine can request spectator processing anytime to reflect any change in the behavior of the spectators. The game engine corresponds to master and SBE to worker. The fuzzy inference engine uses the inputs provided by the game engine as global fuzzy inputs. The fuzzy inference engine generates scalar outputs to reflect any spectator change of behavior, by using global fuzzy inputs (from game engine) and spectator-dependent inputs (using related attributes of the spectator). The generated output(s) updates the respective spectator attribute(s). The fuzzy inference engine is also capable of running hierarchical inferences, which means the output of a fuzzy inference can be used as an input for the next fuzzy inference. Finally whenever required, the game engine gets the updated spectator outputs to the host-side to animate and render the spectators. Therefore, the game engine has all the control in this architecture and SBE has no direct connection to the game field, it only produces outputs. This approach provides flexibility, since any application that processes spectators can use SBE with little effort. Additionally, there is no need to update spectators in each frame and return the results immediately.

### 6.1.3 Performance Test

As previously mentioned, computational and rendering resources are very valuable for achieving life-like visualization of the game field. Minimal resources must be used to process spectators at the background. In this case study, it was assumed that several fuzzy inferences were enough to update spectator behaviors. Therefore a two minutes test (120,000 ms) was run using different spectator populations. In this test the game engine was requested to update spectator behaviors in different intervals. Depending on the call parameters, the related spectator attributes were updated by SBE using fuzzy inference engine (1-10 inferences per request). Finally, the related attributes were copied to the host, whenever needed. The size of the related attribute(s) was assumed 32 bits. Figure 35 shows the results of this performance test for 65,536 spectators using NVidia GTX 295 GPU. Figure 35-a indicates that memory copy operations occupied only %7 of the total GPU processing time. This performance was achieved by getting updated results whenever required, not in every frame. Figure 35-b shows that during two-minute simulation period, SBE was called 120 times (at different periods) and the attributes were copied to the host 47 times (at irregular intervals). Additionally, the quantity of the fuzzy inferences changed in each SBE call (Figure 35-b; "fuzzyInferenceTest" bar size is different in each run). This figure indicates that SBE processing completed in 103.77 ms (96.11 ms for fuzzy inferences, 7.66 ms for memory copy from device to host). Since the total simulation time was 120,000 ms, this processing time corresponded to only 0.09% of the GPU time, which is negligible in whole process chain. The same test was also run on a single CPU core to show the benefit of running SBE on the GPU. Table 9 gives the CPU and GPU performances. For the population of 65,536 spectators, it took

16% of the single core resource to run the same simulation on the CPU, an unacceptable value for sports-based video games.

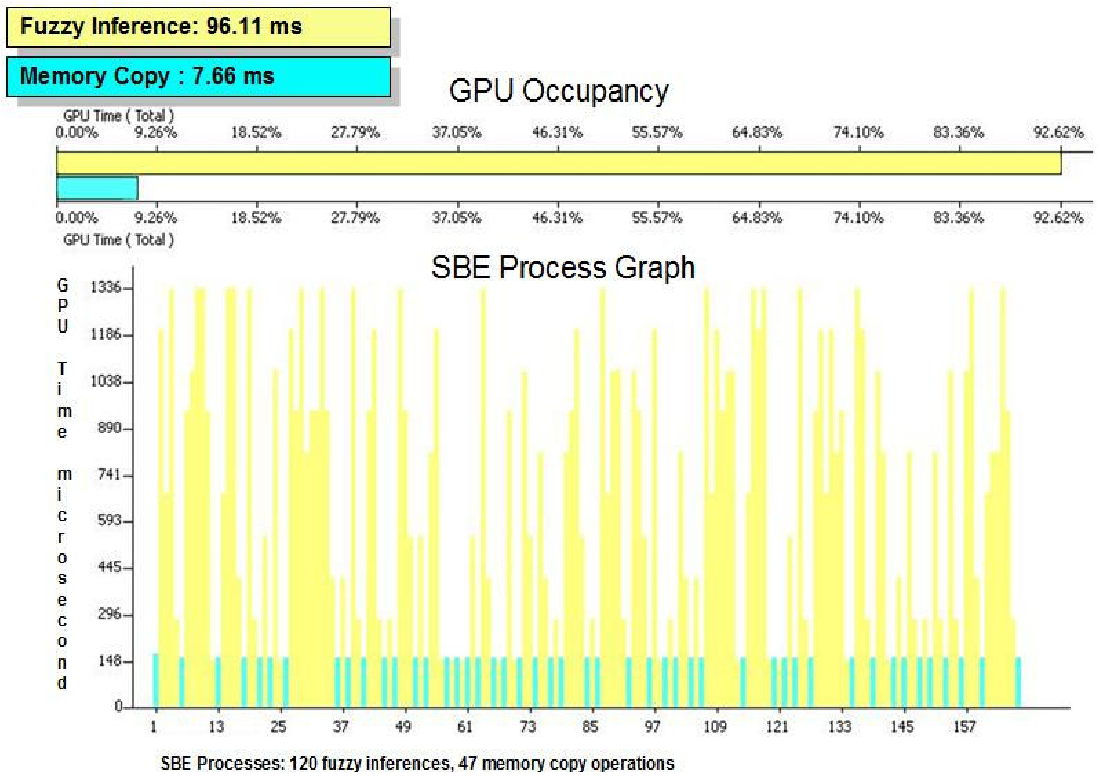


Figure 35: Simulation of 65,536 spectators on the GTX 295 GPU.

Table 9: CPU and GPU performances of the SBE.

| Population | CPU (Single CPU Core ,Intel T 9550 @ 2.67 GHz. ) | | GPU (GT 120M) | | GPU (GTX 295) | |
|---|---|---|---|---|---|---|
| | Process Time ms | Processor Occupancy | Process Time ms | Processor Occupancy | Process Time ms | Processor Occupancy |
| **16,384** | 4,762 | 3.97% | 188.97 | 0.16% | 39.20 | 0.03% |
| **32,768** | 9,470 | 7.89% | 335.74 | 0.28% | 61.81 | 0.05% |
| **65,536** | 18,992 | 15.82% | 651.32 | 0.54% | 103.77 | 0.09% |

### 6.1.4 Visual Results

This section gives the visual result of a specific case in which the spectators cheer a goal by using the fuzzy knowledge-base mentioned in Chapter 5 (see Table 5 and Figure 25-28). The visual outputs of this example are provided in Figure 36 and Figure 37. As can be seen in these figures, there are none of the repeated motion patterns or robotic actions that can usually be observed in soccer video-games. A comparison of Figure 38 and Figure 37 clearly highlights this, given the much greater variety of actions and reactions in Figure 37 than in Figure 38. Soccer games generally treat spectators as a whole rather than as individuals, which results in repeated and user-predictable spectator gestures that may annoy or disappoint the player as depicted by Figure 38. Although fans within a stadium usually make similar moves, it is not impressive to see exact clones. Consequently, this case study produces natural behavioral differences in the crowd. When each spectator is modeled as an intelligent individual, their reactions can be produced based on various factors, such as the characteristics of the game being played, the reputation of the players, the current score and cultural issues.

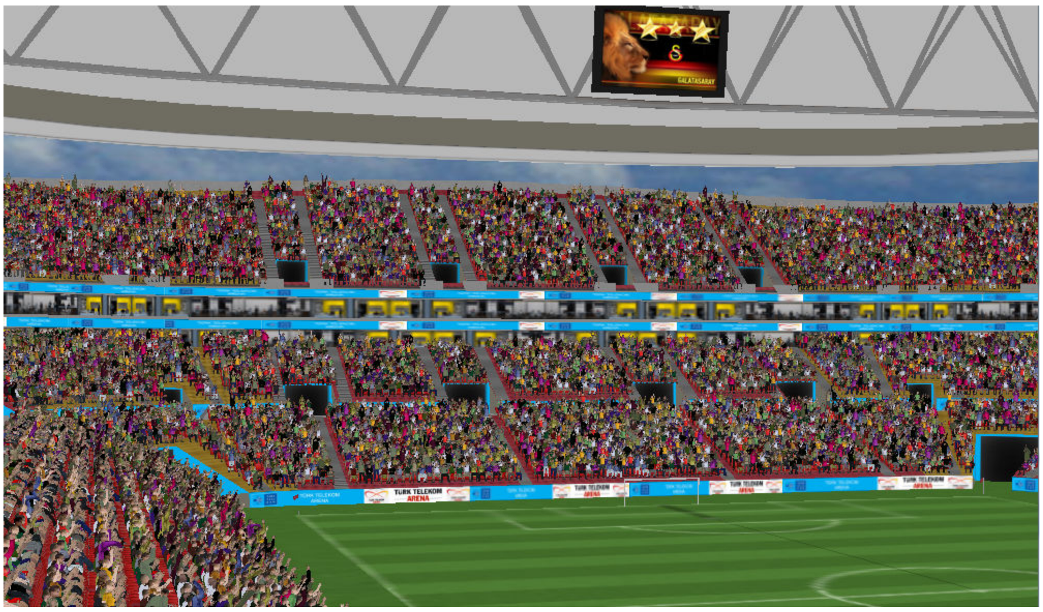


Figure 36: Individually-processed fans celebrate a goal.

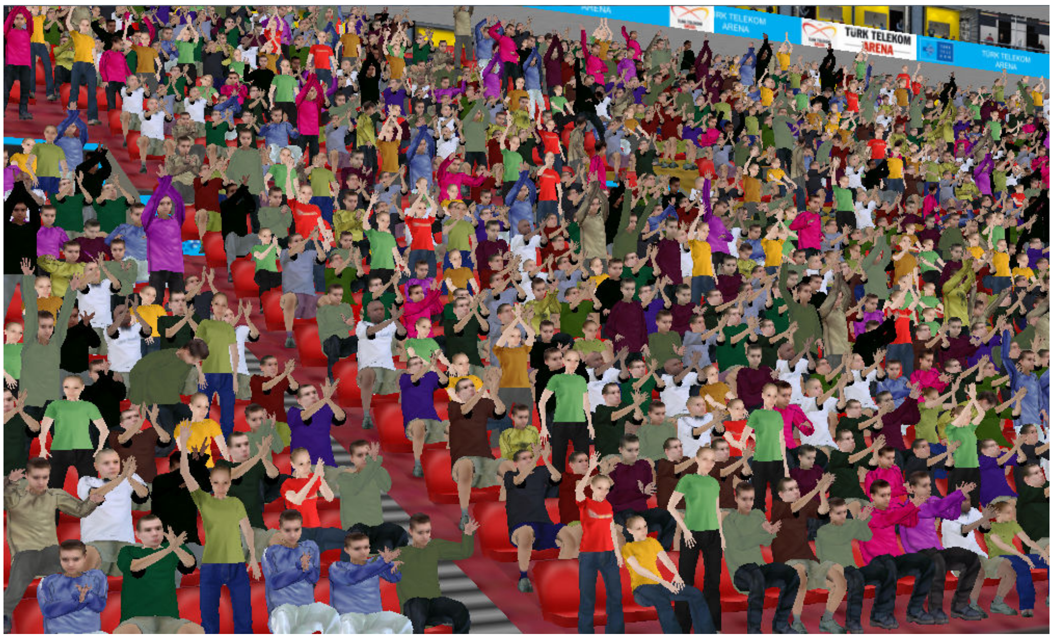


Figure 37: Close-up view. Individually-processed fans celebrate a goal.

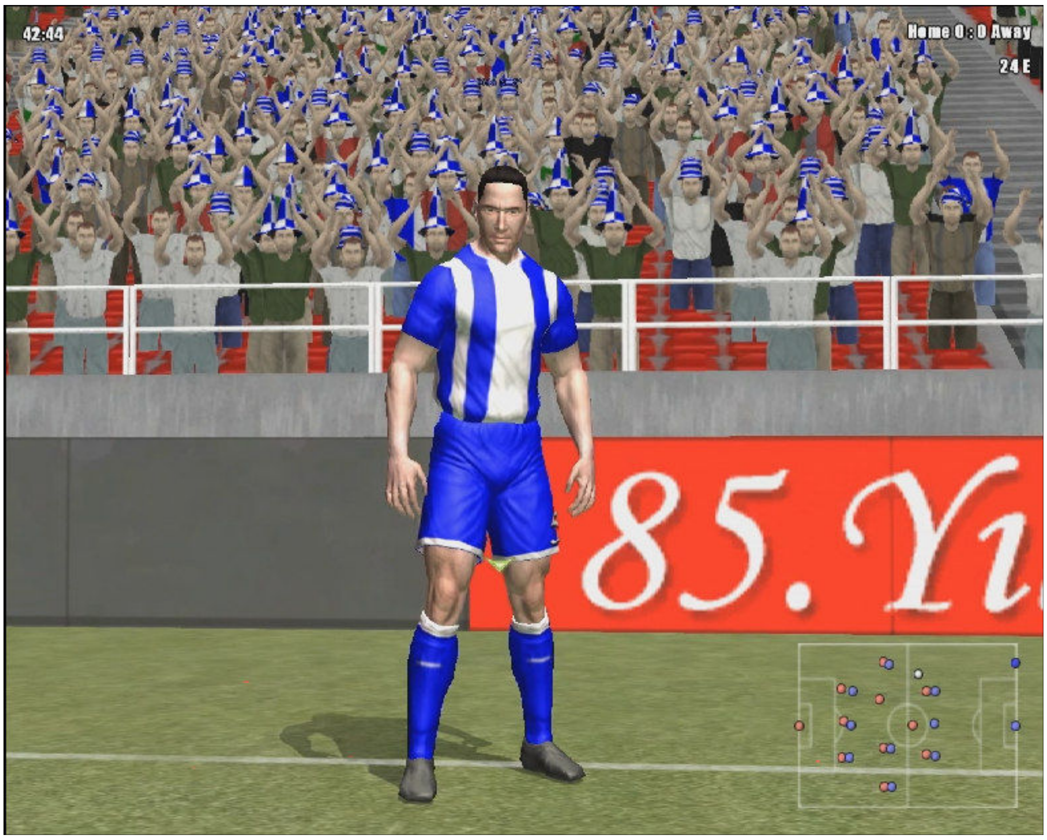


Figure 38: Clone spectators produce repeated actions.

### 6.1.5 Mexican Wave Visualization

In this specific case, SBE was used to simulate a Mexican wave, also known as a stadium or audience wave. The Mexican wave is a collective human behavior where the spectators in the neighbor columns stand up while raising their arms up and then sit down again. This action triggers the neighbors to do the same. If the wave is strong enough, it continues around the stadium several times. The employed fuzzy rules were given in Table 10. There are two inputs for fuzzification. The first one is the strength of the wave, which is derived from the neighbors and the second one is the mood of the spectator. Therefore this example also simulates the interaction with the neighbors. The Mexican wave phenomena was interpreted and quantified by Farkas, Helbing and Vicsek, with a variant of models originally developed to describe cardiac tissue [98]. They examined several Mexican wave videos and reported several results [98]:

- The wave usually rolls in a clockwise direction.
- The typical wave speed is 12 m/s. (Nearly 20 seats).
- The average width is 6-12 m. (Nearly 15 seats).
- The wave is generated by no more than a few dozen people.

Table 10: Fuzzy knowledge-base for Mexican wave example.

| **Rule** | |
|---|---|
| 1 | If ***Wave*** is ***Strong*** **OR** ***Mood*** is ***Normal*** then ***Involvement*** is ***Average*** |
| 2 | If ***Wave*** is ***Strong*** **OR** ***Mood*** is ***Excited*** then ***Involvement*** is ***High*** |
| 3 | If ***Wave*** is ***Weak*** **AND** ***Mood*** is ***Bored*** then ***Involvement*** is ***Low*** |

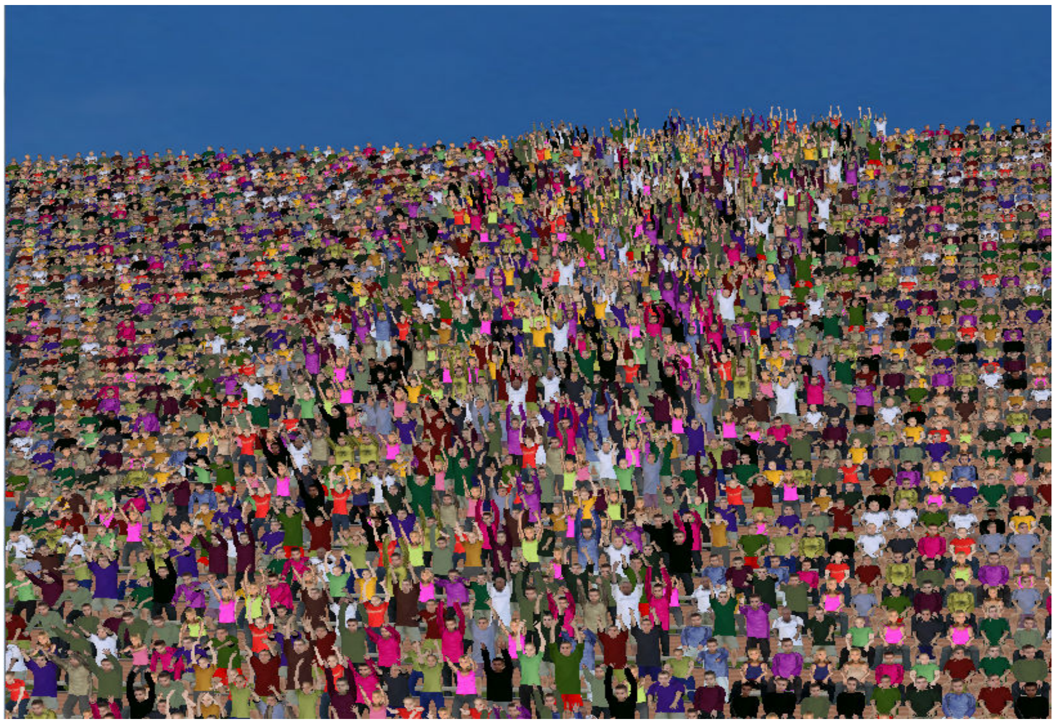


Figure 39: Mexican wave simulation.

The metrics provided by Farkas et al. were taken into account while designing fuzzy knowledge-base and animation speed. The results illustrated in Figure 39 reflect similarities to the above given metrics. As seen in this figure, some spectators do not join the wave, depending on their current mood. The status of the neighbors was used to determine the first fuzzy input which is the strength of the wave. Considering a clockwise rolling direction [98], the spectators on the right have more weight.

## 6.2 H.G. Wells' Little Wars

This case study covers a medieval era combat simulation and visualization, where more than 250,000 warriors fight under heavy arrow storm as shown in Figure 40. The implementation introduces various GPU kernels that handle combat simulation issues such as crowd navigation, combat physics, collision detection, and virtual character behavior modeling. To achieve real-time frame rates, most of the general-purpose parallel programming issues mentioned in the previous chapters are employed. The scene is rendered using OpenGL. Visualization is provided for evaluating the results of the implementation.

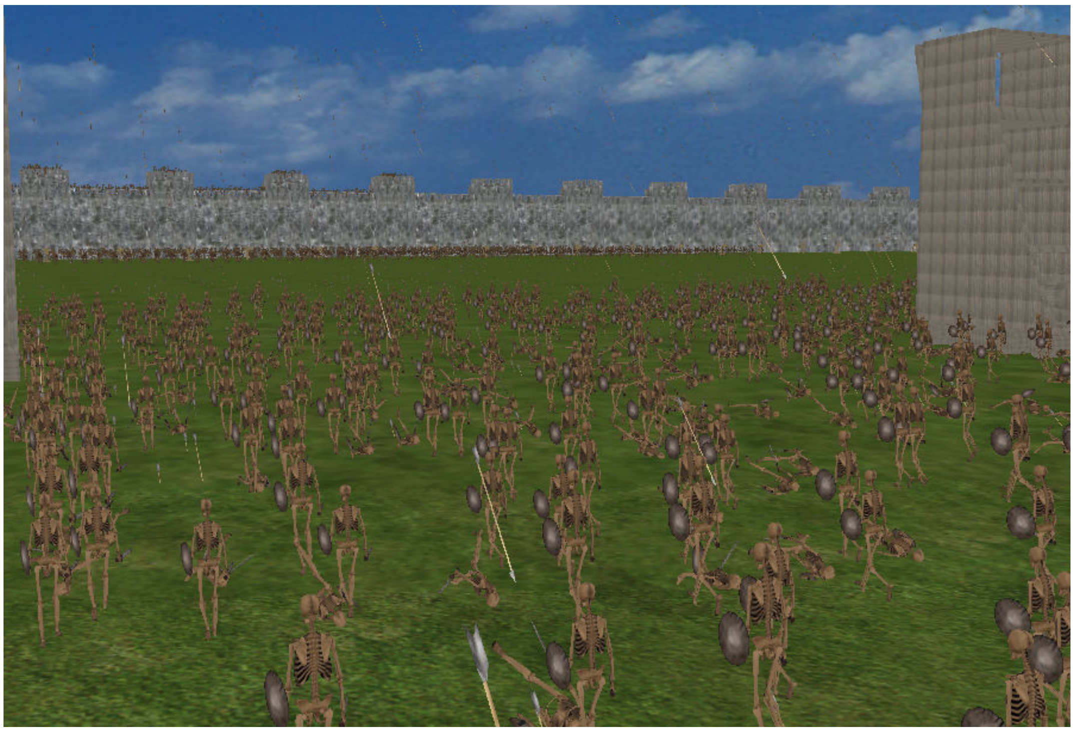


Figure 40: Invaders move toward the city walls.
The arrow sizes were exaggerated in order to be identified easily.

### 6.2.1 Motivation and Background

Tabletop war games and their computer game versions, such as the world wide known RISK, mostly simulate combats through a probabilistic approach. Typically, dice are rolled to decide the loser/winner, or to determine the quantity of casualties. In the computer versions of war games, the probabilistic model can be enriched with the predefined rules and the parameters such as power, hit ratio, training and morale. There are many computational combat models, ranging from simple linear equations to complicated systems. In addition to the probabilistic method, a physics- based approach is also available.

In 1913, English novelist H.G.Wells, famous for the books "Time Machine", "Invisible Man" and "The War of the Worlds", revealed a completely different tabletop war game [12]. He named this game as Little Wars and defined the rules and details in the book "Little Wars: a game for boys from twelve years of age to one hundred and fifty and for that more intelligent sort of girls who like boy's games and books". What he suggested was a new model, depending on training, practice and talent rather than computational approaches. He replaced the role of dice or similar tools with spring-breech loader toy cannons to shoot soldiers. The following quote describes the combat model idea of Little Wars: "Whenever possible, death should be by the actual gun and the rifle fire and not by computation. Things should happen, and not be decided [12]." The case study is inspired from this statement.

In the battle history, there exist some events, whose results cannot be predicted by probabilistic combat models. A typical example of such events is the battle of Agincourt in 1415, where outnumbered English army, mainly consisting of longbow men, won a spectacular victory against a French army of heavily armored soldiers and knights [99,100]. If we run a simple computational model using these inputs, it is certain that a great majority of the runs will be concluded with French victory. That is why researchers and battlefield detectives have been investigating this battle considering many other parameters (terrain, crowd dynamics, etc.) to determine what led to English victory [99,100]. The results of a physics-based combat simulation could better fit to the historical facts provided that the model includes the replicas of the actual arrow shots and correctly positioned units. Although this battle shows the significance of massive quantity of archers, in the other battles the results are not similar. In the battle of Thermopylae, a Spartan army of a relatively small size retarded the huge Persian army for a few days under an extremely heavy arrow storm [101]. Herodotus described this event as "the sun was blocked out by the Persian arrow storm" [101]. Wells' physics based combat simulation model could help to solve mystery behind such battles.

This case study tries to make things happen by simulating the physics of arrows. To determine the casualties, the arrow shots were employed rather than the hit ratio or similar computations.

### 6.2.2 Scenario

The scenario assumes that there is a war between two fantastic fractions showing similar characteristics of the medieval ages. The armies of these fantastic fractions consist of the skeleton archers, skeleton sword fighters and several other medieval siege weapons. An army of 40,000 archers and 10,000 infantry tries to intercept 200,000 invader warriors beyond city walls. The city is well fortified and surrounded by walls and towers. There is also an inner wall as the final defense line. The defender archers are positioned on the walls and towers. There are also archers and sword fighters in front of the city walls. The city lies over a flat plain. Inside the city walls, there are several medieval style buildings and facilities. The invaders launch a direct assault to the front walls. Attacker assault lines decelerate as arrow storm welcomes them. Figure 41 shows outer walls and defenders.

### 6.2.3 Archery Physics

Archery is involved in this case study to blend massive physics and massive crowd simulation. The employed archery model is not an original study and has no allegation such as contribution to medieval archery modeling. However, it is realistic enough to implement the movements of arrows. Compared to the modern bow and arrow, the medieval versions have their own characteristics. Unfortunately, there is not any proper arrow sample survived from that era to make wind tunnel experiments

[102]. However, the stories told about the medieval battles give us realistic data samples such as range [102,103].

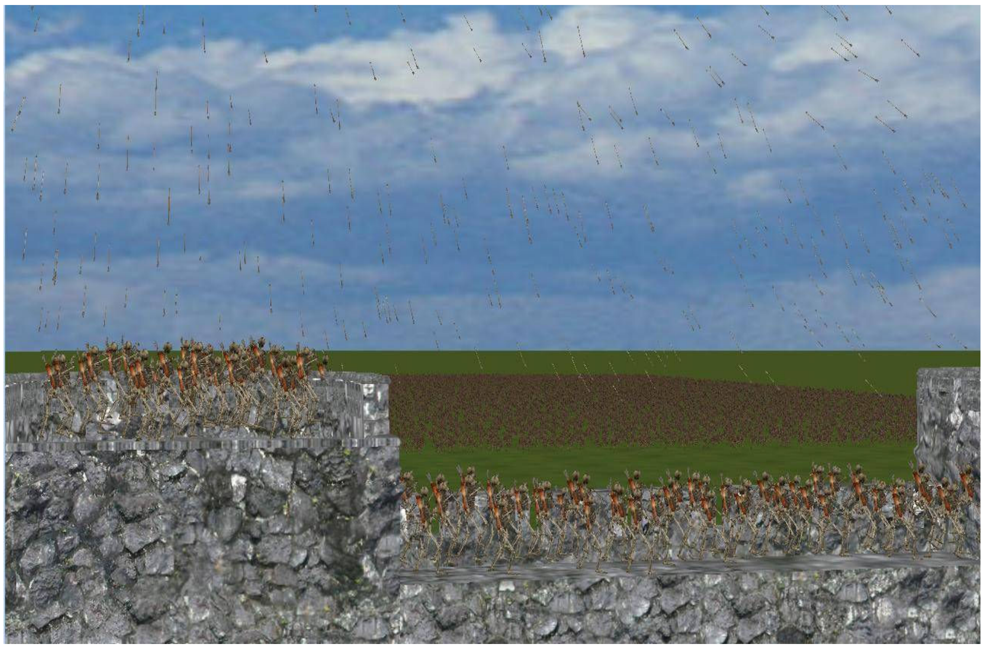


Figure 41: Defenders try to stop invaders beyond city walls.
This screenshot shows the beginning of an arrow storm.

As a starting point, the bow and arrow system is considered as a simple spring system. Since the structure of the medieval bow is well known, this spring system can be completely observed. The potential energy stored during the bow drawing can be precisely calculated. This potential energy is converted into the kinetic energy when the arrow is released. However, due to the oscillations and the moving parts in the bow structure, some of the potential energy is transferred to the bow instead of arrow. In fact, the ratio of the energy transferred to the arrow gives the efficiency of the system. While the modern bows are made up of composite materials giving efficiencies greater than 1, the best medieval longbows are made up of yew and have an efficiency of 0.9 [102]. Even though there is not a proper arrow sample, the existing ones tell us certain information like mass, length etc. Throughout the simulation, the mass of an average arrow is taken as 0.060 kg according to this information. Using these crucial parameters, the initial maximum speed of an arrow can be calculated as ~55 m/s [102]. This is important because during the combat calculations, the complex energy equations can be neglected.

$$\mathbf{x}(t) = \mathbf{x_0} + \int_0^t \mathbf{V}(u) \cdot du$$

(Equation 4.a)

$$\mathbf{V}(t) = \mathbf{V_0} + \int_0^t \left( -\frac{\mathbf{F_d}(\mathbf{V}(u))}{m_a} - g \cdot \hat{\mathbf{e}}_\mathbf{y} \right) \cdot du$$

(Equation 4.b)

Since the energy calculations are unnecessary, only the equations of motion are sufficient to calculate the trajectory of an arrow (Equations 4). To be more realistic, the drag force (wind resistance) is also included into these equations [104]. The characteristic of drag force for an arrow is found through the wind tunnel experiments as $F_d = cV^2$ [102]. The c value is also measured as $10^{-4} N \cdot S^2 m^{-2}$ [102]. The introduction of drag force makes the equation more complex to solve efficiently. However, by using posteriori physics technique, the next position of each arrow can be computed through the instantaneous speed (Equations 5). The effect of the gravity and external forces like wind are taken into account when calculating speed. In fact, the discrete values are calculated with linear approximation fashion. The difference between the estimates is related to the frame render time which is essentially around 30 ms. Thus, the posteriori calculations are repeated every 1.8 m of the arrow travel.

$$\vec{\mathbf{a}}_{\mathbf{t-1}} = -\frac{c_a}{m_a} \cdot \begin{bmatrix} V_x^2 \hat{\mathbf{e}}_\mathbf{x} \\ \operatorname{sgn}(V_y) \cdot V_y^2 \hat{\mathbf{e}}_\mathbf{y} \\ V_z^2 \hat{\mathbf{e}}_\mathbf{z} \end{bmatrix} - g \cdot \hat{\mathbf{e}}_\mathbf{y}$$

(Equation 5.a)

$$\vec{\mathbf{x}}_\mathbf{t} = \vec{\mathbf{x}}_{\mathbf{t-1}} + \vec{\mathbf{V}}_{\mathbf{t-1}} \cdot \Delta t$$

(Equation 5.b)

$$\vec{\mathbf{V}}_\mathbf{t} = \vec{\mathbf{V}}_{\mathbf{t-1}} + \vec{\mathbf{a}}_{\mathbf{t-1}} \cdot \Delta t$$

(Equation 5.c)

$V_x$, $V_y$, and $V_z$ are components of $\vec{\mathbf{V}}_{\mathbf{t-1}}$

After executing several trials, the range of the arrow with maximum initial speed and 45° release angle is measured as ~230 m. This value is consistent with Sir Roger Williams' writings [102]. The result is shown in Figure 42.

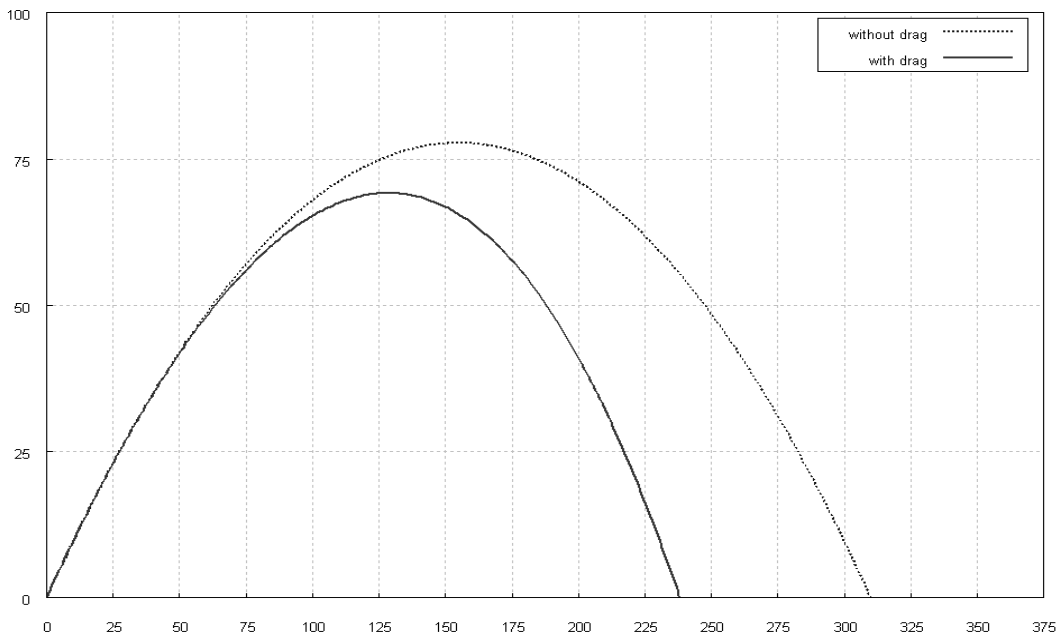


Figure 42: Arrow trajectory graph.

For the initial speed of 55 m/s and release angle 45º, the calculations are done at 0.03 second interval. The graph shows that the posteriori method is capable of simulating real arrow physics. The range is around 230 m, as expected.

### 6.2.4 Accurate Vertical Positioning

Accurate positioning of large number of virtual characters that are moving on an uneven terrain surface or collision detection of thousands of arrows with terrain in real-time is a power-hungry task. It is required to calculate actual elevation value of the underlying terrain for each dynamic entity in the simulation that interacts with the terrain. The accurate terrain elevation is calculated using plane geometry. The following method is employed, when horizontal coordinate pair is provided.

- Find the triangle on which the point lies.
- Get vertex coordinates of the triangle.
- Calculate the normal vector of the triangle by using the Equation 6.
- Calculate the plane-shift constant value of the triangle by using the Equation 7.
- Calculate the height value by using the Equation 8.

$$\vec{N} = (\vec{V}_2 - \vec{V}_1) \times (\vec{V}_3 - \vec{V}_1) \qquad \text{(Equation 6)}$$

$$D = \vec{N} \bullet \vec{V}_1 \qquad \text{(Equation 7)}$$

$$P_y = (N_x P_x + N_z P_z - D)/N \qquad \text{(Equation 8)}$$

where, $\vec{N}$ denotes normal vector of the plane; $\vec{V}_1$, $\vec{V}_2$, $\vec{V}_3$ are the vertices of the triangle in vector form, $D$ is plane-shift constant and $P_x$, $P_y$, $P_z$ are 3D coordinates of the point.

To decrease computational cost, texture memory is assigned. Pre-computed normal vectors and plane-shift constants of the triangles of the terrain model are stored in a texture memory and used whenever required. This functionality is called by the kernel “getTerrainHeight”.

### 6.2.5 Application Workflow

The workflow is illustrated by Figure 43. This application is a heterogeneous solution, in which both the CPU and GPU takes active roles regarding computations and rendering issues. The simulation starts just after the initialization. In each simulation cycle the same processing steps are repeated. The cycle begins with a pre-process on the CPU by organizing data in a way to be handled faster on the GPU, such as grouping considering spatial partitioning or computational cost. The GPU takes computing responsibility and starts processing with the archers, followed by the processing of the arrows and the infantry warriors. Some of the updated information regarding the warriors, archers and arrows are returned back to the CPU for rendering and grouping for the next loop. The details of these simulation steps are covered in the next parts.

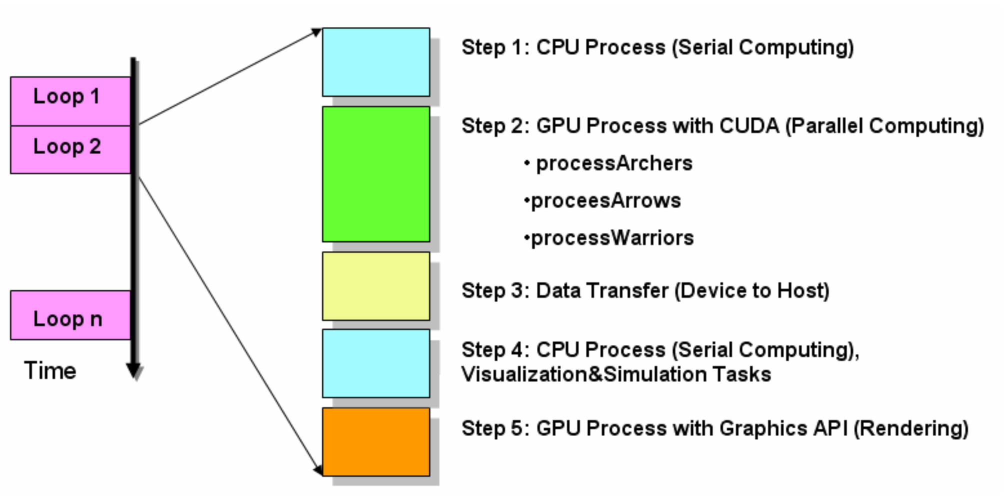

Loop 1
Loop 2
Loop n
Time
Step 1: CPU Process (Serial Computing)
Step 2: GPU Process with CUDA (Parallel Computing)
• processArchers
•proceesArrows
•processWarriors
Step 3: Data Transfer (Device to Host)
Step 4: CPU Process (Serial Computing), Visualization&Simulation Tasks
Step 5: GPU Process with Graphics API (Rendering)

Figure 43: Application workflow.

#### 6.2.5.1 Simulating Archers

The archers are simulated by kernel "processArchers". Neither a retreat nor an attack is assumed. Therefore the navigation of virtual characters is not included in this kernel. The following tasks are performed by this kernel:

- Compute the visibility and LOD of the archer.
- Launch an arrow.

##### 6.2.5.1.1 Visibility and LOD Computation

Computing the LOD and visibility on the GPU saves significant amount of computation time compared to the CPU, as mentioned in Chapter 4. The visibility and distance to the camera are determined by using view frustum and squared distance metric. The view frustum which is under control of the CPU is transferred to the constant memory, when the camera changes its position. If the camera is steady for a period, this approach provides computational performance. To implement effective bit compaction, three bits are assigned to store visibility and LOD (0 for outside view frustum, 1-7 for LOD).

#### 6.2.5.1.2 Launching an Arrow

The releasing parameters of an arrow are determined by a bow drawing model that utilizes a single fuzzy inference which is built on the strength and the experience of the archer. The output of this fuzzy inference determines the deviation from the corresponding releasing parameters computed by using a preprocessed look-up table that is stored in the constant memory. This model ensures that the stronger and more experienced archers make more accurate throw.

### 6.2.5.2 Arrow Processing Kernel

The computations regarding archery are handled by using “processArrows” kernel. This kernel tries to optimize previously mentioned CUDA performance enhancement issues as much as possible. This kernel performs the following tasks:

- Compute the visibility and LOD of the arrow.
- Update the position of the arrow.
- Perform collision detection (terrain, object and warrior).
- Perform data compaction.

To parallelize the problem, arrows are considered as the most basic computation element, and thus each arrow is processed via a single GPU thread. The trajectory of the arrow is related with the archer that launches it. Therefore, the output of the “processArcher” kernel is used by “processArrows” kernel. An average arrow is assumed to move 1 meters per 15 ms. This information is required for collision tests.

#### 6.2.5.2.1 Updating Arrows

The second task is to compute the new position, pitch and yaw angles of the arrow. This action is performed if the arrow is launched and not yet hit to the ground, a warrior or an obstacle. A one bit flag is set to true for launched arrows and set to false when it hits one of the above listed entities. Although this approach prevents further computations, it may also cause a path divergence, which should be avoided as much as possible. However, in practice it is not possible to employ all the practices offered for efficient GPU computation, at the same time.

To compute the new position and angles, equations 3,4 and 5 are used. The unique properties of the arrow together with the launching parameters provide unique trajectories. To the best of the author's knowledge, this amount of arrow simulation in real-time has not been demonstrated yet. Figure 44 shows unique arrow trajectories. Besides the unique properties of the arrow, there are also several external factors such as wind. To use memory efficiently and minimize memory accesses, such external factors are stored in constant memory. The new and the previous positions are stored as local variables for collision tests, since the arrow might have already been traveled several meters.

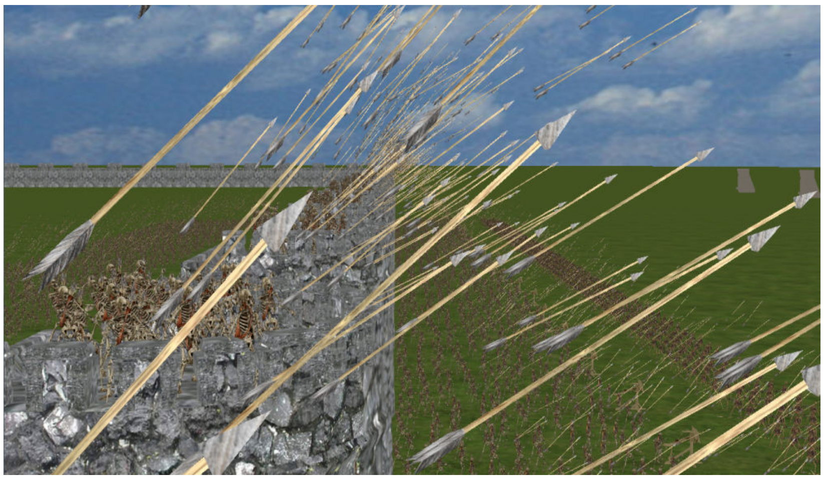


Figure 44: Thousands of parallel processed unique arrow trajectories.

### 6.2.5.2.2 Collision Detection

The hardest computation in this kernel is to perform various collision tests to check whether the arrow keeps travelling or not. To minimize computational load, the tests are done considering the order of complexity. Therefore the simplest one is performed first. Additionally, spatial partitioning is employed to minimize O($n^2$) complexity, as discussed in Chapter 2 [63, 65, 66].

The simplest test is to check the elevation of the arrow to determine whether it is above the cell ceiling or not. The cell ceiling is a scalar value stored in a constant memory which is 10 meters above the highest vertical structure in that cell. If there is no object, this value becomes 10 meters above the highest terrain elevation. Figure 45 illustrates cell ceiling. Further collision tests are performed when the arrow is below the cell ceiling.

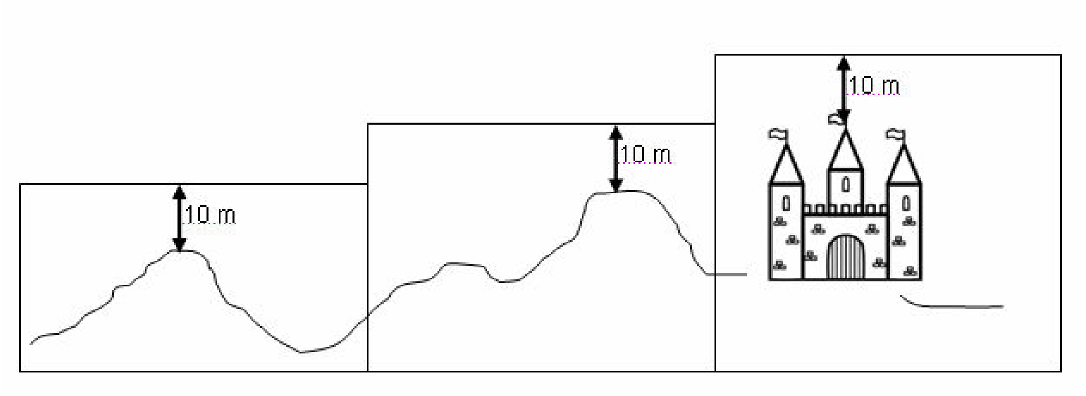


Figure 45: Illustration of the cell ceiling.
The ceiling is 10 m above the highest point in that spatial cell.

The next test is to check whether the arrow hits the dynamic or static entities in the scene. These obstacles are catapults, ballistas, city walls or buildings. To increase computational speedup, the bounding geometries are stored in the constant memory. When a potential collision is predicted, the detailed geometry is used if necessary. The full geometry is stored in the global memory and accessed only when necessary.

In this sense, LOD is used for collision detection. For a detailed collision test, the previous position is also used to compute impact point accurately. The squared distance is used to check whether the arrow is within the hit threshold or not.

This test is followed by collision detection with the warriors. Further bounding box and full-geometry tests are performed in a similar way as described above. In this implementation, no realistic collision response is employed. Spatial partitioning helps achieve faster collision-detection to meet real time requirements. If a collision occurs the warrior's status is changed accordingly (such as dead).

This test is followed by collision detection with the terrain. The hit point is computed using previously mentioned "getTerrainHeight" kernel.

#### 6.2.5.2.3 Data Compaction

The last step in arrow processing is to perform data compaction. In this step the empty spaces in the arrow position data (type: float4) are used to store heading and tilt angles (9 bits for each angle), visibility and LOD (3 bits), the arrow status (1 bit). Thus the size of the data that will be transferred to the host is reduced.

### 6.2.5.3 Warrior Processing Kernel

The computations regarding warriors are done by using "processWarriors" kernel which performs the following tasks: The first and the last tasks are very similar to the previous kernel, and have already been explained. The second and the third tasks are discussed in this section.

- Compute the visibility and LOD of the warrior.
- Fight.
- Update the position of the warrior (navigation).
- Perform data compaction.

#### 6.2.5.3.1 Combat Model

A fuzzy inference which uses the strength and the training-level of the warrior was employed to simulate combat model between the warriors. The output of this fuzzy inference updates the health of the warrior. The combat model ensures that in each combat the health decreases. Therefore there are no unbeatable super heroes in the simulation. Additionally, the combat model works within a certain distance threshold, thus ensures many-to-many warrior combats as well.

#### 6.2.5.3.2 Navigation

In this simulation, the invaders move toward the city walls while defenders try to keep their posts. Each warrior has speed and direction attributes that are required to compute the new position. Although the city is surrounded by a fairly flat terrain, accurate vertical positioning is also done to prevent sink or raise problem with the terrain surface. The path is updated, if it is blocked by an obstacle. Following position update, spatial hashing is applied to minimize the computational load of the arrow collision detection and combat model.

### 6.2.6 Simulation Performance

Figure 46 shows run-time results of the above mentioned kernels and respective memory copy operations. As this figure shows the GPU handles almost everything in 13 ms (5.1 ms for “processArrows”, 4.2 ms for memory copy, 3.3 ms for “processWarriors” and 0.5 ms for “processArchers”). The simulation achieved nearly 40 fps (25 ms per frame) on a PC with GTX 295 GPU. Therefore 12 ms is used by the rendering process. This high frame rate is also dependent on low polygon count of the models (nearly 1500 polygon) and the employed LOD and visibility culling. The achieved 40 fps speed indicates that real-time visualization of a massive combat simulation is possible by using GPU parallel processing. The results in this figure also show that possible optimizations should better focus on arrow processing kernel and memory copy operations.

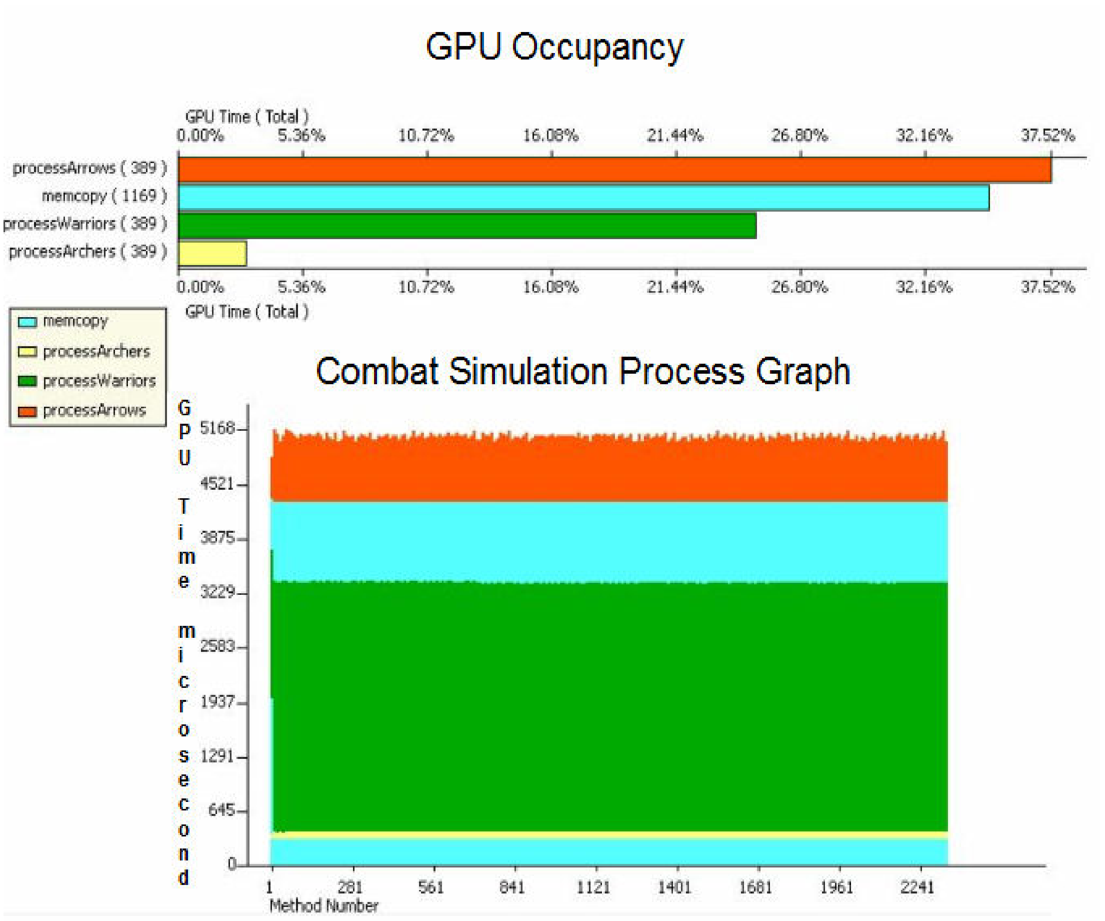


Figure 46: GPU performance graph.

### 6.3 Virtual Marathon

This case study summarizes a published work that covers several topics discussed in this thesis [88]. The work is about one of the most crowded events in city life; a marathon. The overall population in well-known marathon runs sometimes exceeds one million people. For example in the annual New York City Marathon nearly one million people supports 40,000 runners. Figure 47 compares the real and virtual marathon events on the İstanbul's Bosporus Bridge that connects Asia and Europe. In this virtual marathon event 32,768 runners and 1,015,808 spectators were simulated.

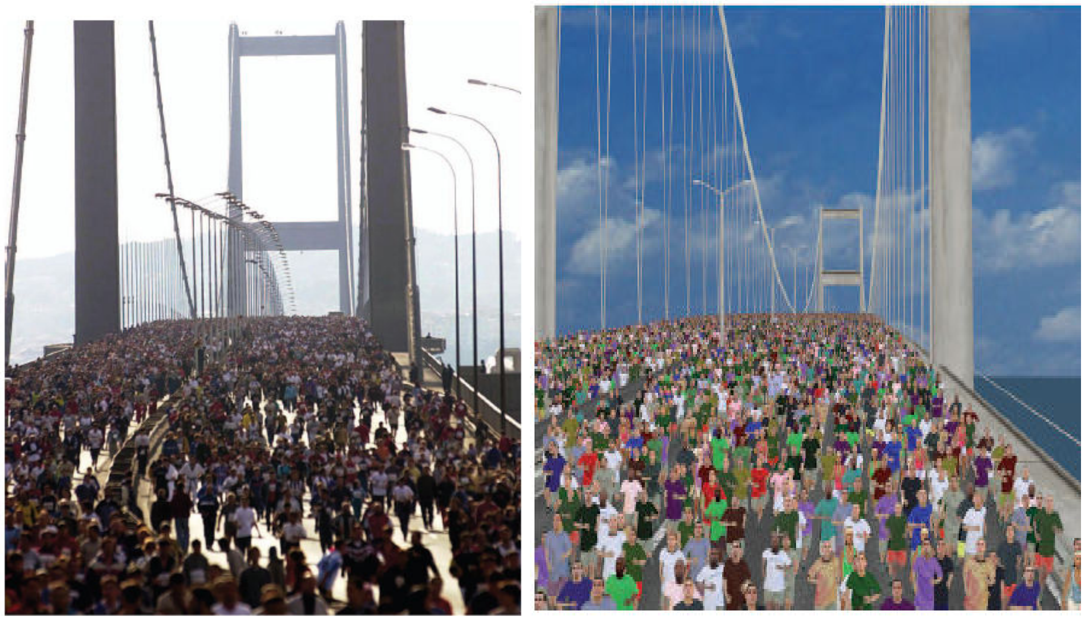


Figure 47: Marathon crowds.

Real photo (left, courtesy of İstanbul Municipality) and a screenshot of the application (right).

In this study multimonitor setups were used to simulate this event on large screens. Figure 48 depicts the employed systems. By using GTX 285 GPU 11-12 frames per second was achieved while simulating over one million virtual characters.

The fuzzy logic routines that were introduced in Chapter 5 were used to model the behaviors of the runners and the spectators. This simulation did not include collision detection process on the GPU. However limited collision detection that includes few runners that are close to the camera was employed on the CPU, only for visualization purposes.

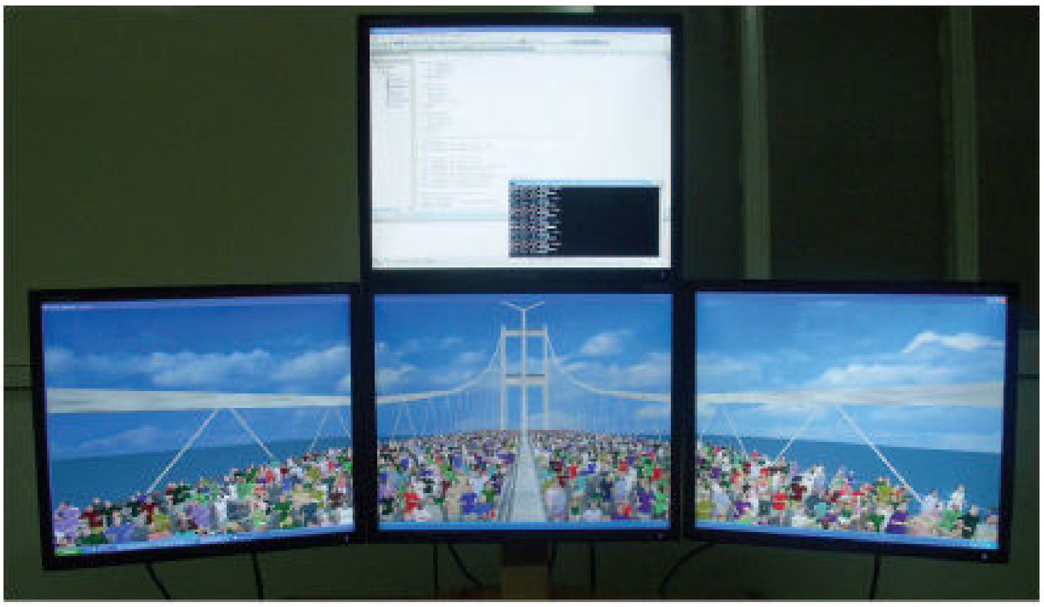
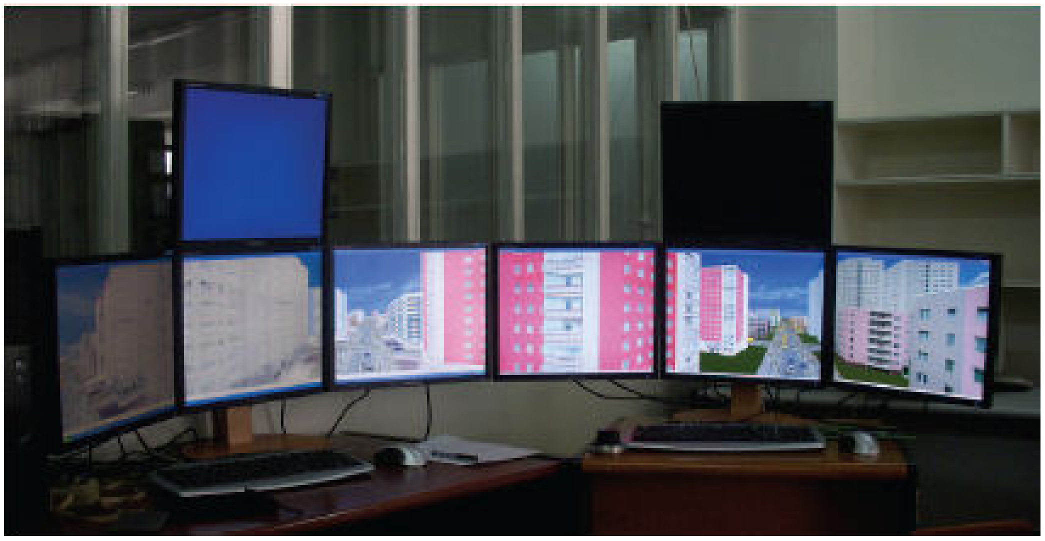



Figure 48: Multimonitor setups for the virtual marathon.

(Top) The basic setup consisted of three connected 19-inch LCD monitors, which produced 3,840 × 1,024 pixel resolution. (Bottom) An enhanced multimonitor setup provides increased resolution.

Table 11 gives the comparison of CPU and GPU performances of this simulation by using two different behavioral models; low-cost model and high-cost model. The first model includes one fuzzy inference, and the second model includes four fuzzy inferences and a more precise frustum-culling algorithm. Thus, it contains nearly five times more computations.

Table 11: CPU and GPU processing times for updates.

| **Number of people** | **Processing time (ms)** | | | |
|---|---|---|---|---|
| | **Low-cost model** | | **High-cost model** | |
| | **CPU (Quad Core @2.67 GHz)** | **GPU (GTX 285)** | **CPU (Quad Core @2.67 GHz)** | **GPU (GTX 285)** |
| 32,768 | 46.25 | 3.10 | 198.11 | 3.16 |
| 65,536 | 90.64 | 4.32 | 349.39 | 4.36 |
| 131,072 | 179.36 | 8.59 | 786.24 | 8.76 |
| 262,144 | 356.48 | 15.52 | 1,573.16 | 15.60 |
| 524,288 | 711.28 | 30.89 | 3,119.36 | 31.29 |
| 1,048,576 | 1,420.64 | 59.86 | 6,282.52 | 61.20 |

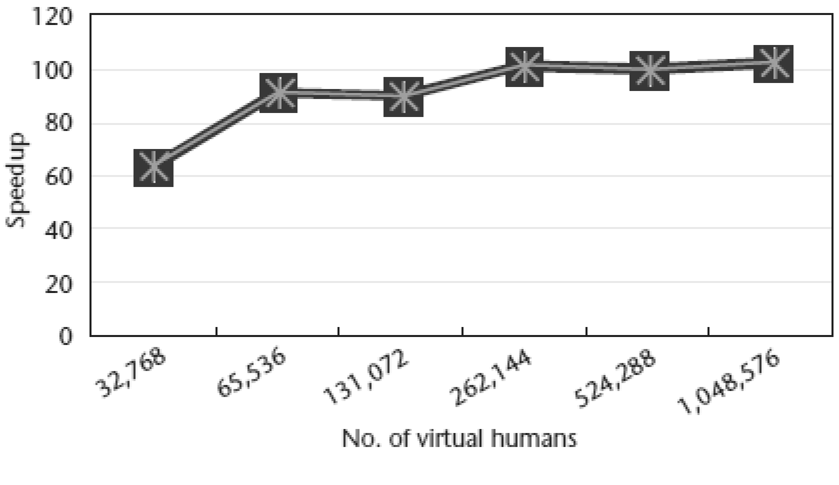


Figure 49: GPU speedup for the high-cost model

Figure 49 shows the results of the simulation repeated with various populations. The reported 100× speedup is far away from previously demonstrated speedups. Please note that, stream reduction techniques discussed in Chapter 3 and Chapter 4 were not employed in this case study. Therefore, memory bandwidth appeared as the major bottleneck. Figure 50 and 51 show details from this simulation.

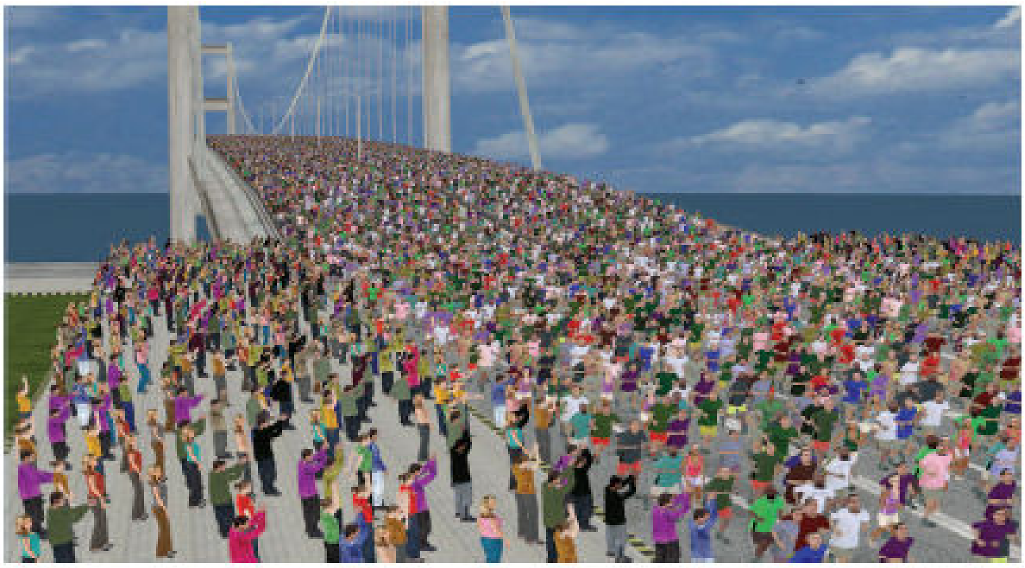


Figure 50: Vertical positioning of the virtual people.
The GPU precisely calculates the touch point of the virtual people to the ground.

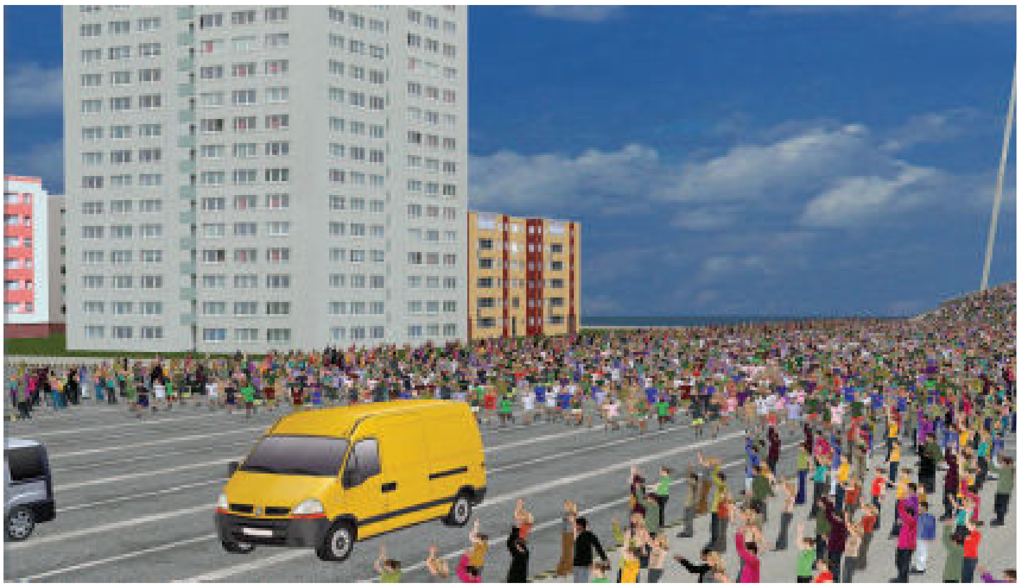


Figure 51: Spectators applauses the front-line runners.

# CHAPTER 7

# CONCLUSIONS AND FUTURE WORK

In this thesis various approaches were introduced that are useful for massive crowd simulation implementations on the GPU. This study is one of the first that handles massive crowd simulation using parallel processing with CUDA. The reported performance results show that the difficulty when creating massive crowd simulations in real-time can be solved with commodity PCs. The content and the contributions are mainly related to handling more virtual characters in less time, and thus using the freed resources to add more population to the virtual environment or to increase the realism level of the simulation. Due to the fact that this thesis is one of the first studies in this field, there are still many issues to improve and resolve. This chapter summarizes this thesis' contributions and provides information regarding the further directions.

## 7.1 Contributions

This thesis has demonstrated that it is possible to simulate massive crowds even comprised of hundreds of thousands virtual characters in real-time. The methodology covered in this study has demonstrated that by using GTX 295 GPU over 400× speedup is possible compared to the same massive crowd simulation that runs on a single CPU core. Therefore, the main contribution of this thesis is a methodology which can be employed to simulate massive crowds in real-time using parallel processing with CUDA technology. The methodology starts with re-arrangement of the data structures. The virtual characters' attribute data structure is divided into sub-parts with respect to the processor(s) that will use these attributes. This action prevents unnecessary data transfer which is the main bottleneck of the CUDA technology. Furthermore, the bandwidth problem is minimized by employing a data compaction that represents attributes with fewer bits whenever possible. The described data compaction also provides less memory access and may help ensure coalesced memory access. The encoding and decoding processes are almost negligible, since the bit extraction from the data envelope and bit insertion into the data envelope, are not computationally complex. Dividing data into sub-parts also makes it easy to handle and transfer the corresponding data portions whenever required. Basically, not all the data needs to be processed and transferred at the same frequency. Thus, handling the data sets as a whole sometimes means wasting limited bandwidth, because unchanged or unnecessary data is transferred within the larger data structure.

Another contribution is Fuzzy Inference implementation on the GPU using CUDA kernels to model behaviors of the virtual characters. These kernels make it possible to compute millions of inferences per second. To provide computational efficiency, the fuzzy knowledge-base and the rule-base are transferred to the constant memory of the device. The mentioned fuzzy parameters are converted into scalars and transferred to the constant memory. While designing these kernels, performance and flexibility were taken into consideration. Thus, the approach regarding the use of fuzzy inferences is not limited to massive crowd simulation; it can be used by the other disciplines as well. This thesis demonstrated that it is possible to compute over 500,000,000 fuzzy inferences per second, which is enough to model behaviors of many virtual characters in a shorter time, by using commodity hardware. To improve computational speedup, different device memories were assigned. Texture memory was used for fuzzification inputs that remain the same, such as personality, throughout the simulation while the global memory was used for inputs that change, such as fatigue. To capture the experience of the domain expert, a user-friendly GUI tool was developed. The captured knowledge-base and rule-base were used to generate an XML script which was converted into the variables and parameters required by the corresponding CPU and GPU functions.

Additionally, this thesis contributed to the blending of real-time massive crowd simulation and massive physics in the same application. To the best of the author’s knowledge, no other study that simulates extensive number of physical objects together with massive crowds has been found. In the case study “Little Wars,” tens of thousands arrows were physically modeled and simulated in the virtual combat field where more than 250,000 warriors involved. This case study deals not only with the collision detection of the arrows but also with the warrior collision avoidance, the combat models, and the bow drawing models.

## 7.2 Future Work

The topics covered so far about massive crowd simulation using GPU are only the visible part of the iceberg. Since crowd simulation is a wide research area, one must go deep beneath the surface to discover what else can be done to improve massive crowd simulation with performance and realism. Besides the crowd simulation research issues, another improvement could be to adapt the offered solutions to newer versions or variants of CUDA technology.

NVidia CUDA is a new vendor dependent technology. Although only two years have passed since the official release, it has already been updated several times. In the first quarter of 2010, there will be significant improvements and changes in the hardware and the development platform. FERMI (the new version of CUDA) will introduce more parallel cores, and thus offer more processing power [78]. Additionally, the software development environment will be easier than the existing environment. Moreover, NVidia is not the only player in this field; other CPU/GPU vendors will also release SIMD based hardware. In this sense, the future work should make offered solutions flexible enough to work on different vendor platforms and various massively parallel computing tools. These probable platforms must be considered while trying to implement future issues regarding massive crowd simulation.

The described fuzzy inference solution has the potential for many other disciplines. The current system was built upon several assumptions and limitations. Amendments and improvements are necessary to the described solution before its use in alternative research areas. In the future, linguistic hedge functionality, or the ability to add adverb like options such as “somehow” and “nearly,” needs to be added by revising the knowledge-base/rule-base GUI, XML definition and fuzzy inference kernels. Also, since the fuzzy operators are currently limited to “AND” and “OR,” more operators can be added. The last issue in this context is to enrich the employed defuzzification methods.

Furthermore, stream compaction should be better improved by employing the arithmetic coding algorithm that already works well with CUDA technology [80]. The proposed data compaction technique helps minimize the transferred data load when only floats can represent data and precision is not important. Stream compaction is a huge research topic and even the reduction of a single bit in massive crowd simulation will be a significant improvement. Therefore, more attention will be paid to the stream reduction.

The “Little Wars” case study will be improved by employing a better and more realistic physics model, collision detection, and collision avoidance algorithms. The employed algorithms must compute collisions and model the reactions more accurately. The combat simulation also needs to be more realistic, requiring a more complicated combat model and a set of combat Mo-Cap data. To enrich actions, a newly acquired Mo-Cap system in METU MODSIMMER Center will be used. Regarding the visualization of these applications, a 3rd party game engine will offer better quality rendering.

The "12th Man" case study needs to be validated as useful by comparing the GPU generated results with real spectator behaviors observed from real soccer game videos. Additionally, the generated results should be compared with existing soccer video games, thus proven to be better. As previously mentioned, more virtual spectators' actions generated by the GPU parallel processing to reflect behavioral variety should be added.

Finally, six high-end PCs equipped with 3-4 upcoming GeForce 300 series GPUs will be connected, making the processing power over 10 teraflops per second. The system will be connected to a newly built back-projection wall comprised of 6 projectors and the corresponding large screen to simulate massive crowds containing millions of virtual characters. This system will be used for the large-screen visualization of massive crowd simulation and other scientific visualization studies as well. The large-screen display solution is depicted in Figure 47.

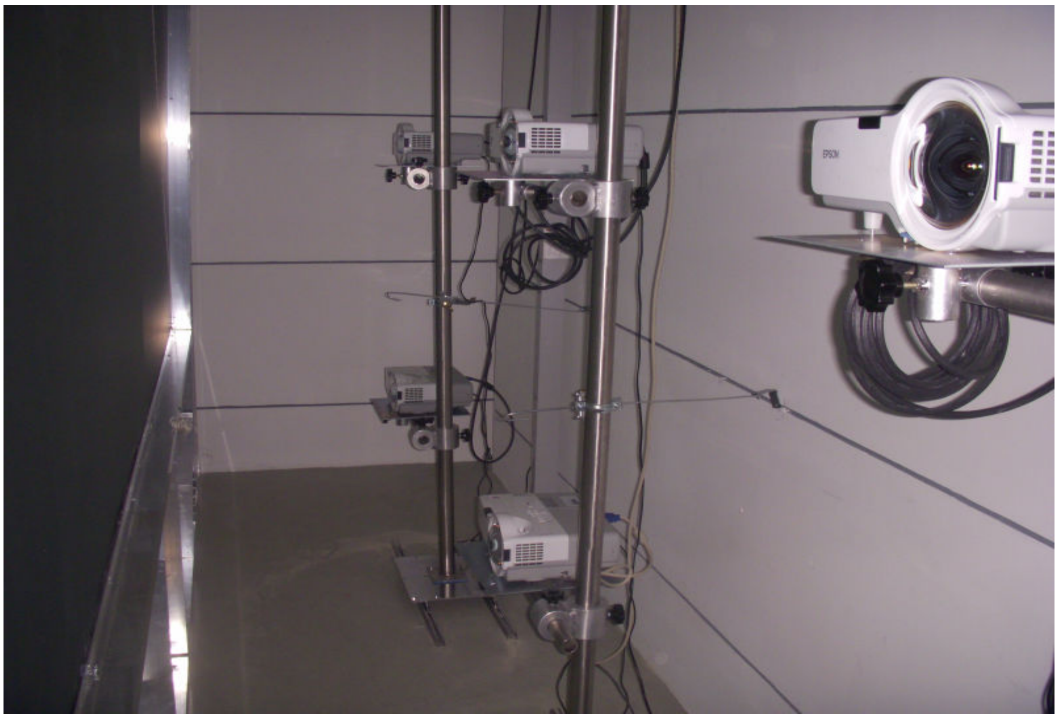


Figure 52 Back-projection large-screen display comprised of six back-projectors and seamless plexiglas wall.

# BIBLIOGRAPHY

[1] Mehrara, M., Jablin, T., Upton, D., August, D., Hazelwood, K., Mahlke, S. (2009). Multicore compilation strategies and challenges. IEEE Signal Processing Magazine, 26(6), 55-63.

[2] Weber, O., Devir, Y., Bronstein, A.M., Bronstein, M.M., Kimmel, R. (2008). Parallel Algorithms for Approximation of Distance Maps on Parametric Surfaces. ACM Transactions on Graphics, 27(4), 1-141.

[3]. Silberstein, M., Schuster, A., Geiger, D., Patney, A., Owens, J.D. (2008). Efficient Sum-Product Computations on GPUs Using Software-Managed Cache. In proceedings of the 22nd ICS, 309-318.

[4] Nyland, L., Harris, M., Prins, J. (2008). Fast N-Body Simulation with CUDA. Nguyen, H. (Eds.), GPU Gems 3 (pp.677-695). Addison-Wesley.

[5] Le Grand, S. (2008). Broad-Phase Collision Detection with CUDA. Nguyen, H. (Eds.), GPU Gems 3 (pp. 697-721). Addison-Wesley.

[6] Howes, L., Thomas, D. (2008). Efficient Random Number Generation and Application Using CUDA. Nguyen, H. (Eds.), GPU Gems 3 (pp. 805-830). Addison-Wesley.

[7] Lastra, M., Mantas, J.M., Urena, C., Castro, M.J., Garcia, J.A. (2009). Simulation of shallow-water systems using graphics processing units. Mathematics and Computers in Simulation, 80(3), 598-618.

[8] Thibault, J.C., & Senocak, I. (2009). CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. In 47th AIAA Aerospace Sciences Meeting, Retrieved January 11, 2010, from http://pdf.aiaa.org/preview/CDReadyMASM09_1811/PV2009_758.pdf

[9] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. Retrieved January 11, 2010, from http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[10] NVIDIA CUDA Compute Unified Device Architecture – Quick Start Guide. Retrieved January 11, 2010, from http://developer.download.nvidia.com/compute/cuda/2_3/docs/CUDA_Getting_Started_2.3_Windows.pdf

[11] NVIDIA CUDA Compute Unified Device Architecture – Best Practices Guide. Retrieved January 11, 2010, from http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf

[12] Wells, H.G. (1913). Little Wars: A Game for Boys From Twelve Years of Age to One Hundred And Fifty And for That More Intelligent Sort of Girl Who Likes Boys' Games and Books. London: Frank Palmer.

[13] Daniel, T., & Soraia, R.M. (2007). Crowd Simulation. Berlin: Springer.

[14] Ciechomski, P.H., Schertenleib, S., Maïm, J., Thalmann, D. (2005). Reviving the Roman Odeon of Aphrodisias: Dynamic animation and variety control of crowds in virtual heritage. VSMM, 601–610.

[15] Yılmaz, E., Yardımcı, Y., Erdem, Ç., Erdem, T., Özkan, M. (2006). Music Driven Real-Time 3D Concert Simulation. Günsel, B., Jain, A.K., Tekalp, A.M., Sankur, B. (Eds.), MRCS (pp. 379-386). Berlin: Springer.

[16] Dudash, B. (2008). Animated Crowd Rendering. Nguyen, H. (Eds.), GPU Gems 3 (pp. 39-52). Addison-Wesley.

[17] McDonnell, R., Dobbyn, S., O'Sullivan, C. (2005). LOD human representations: A comparative study. Proceedings of the First International Workshop on Crowd Simulation, 101–115.

[18] Tecchia, F., Chrysanthou, Y. (2000). Real-time rendering of densely populated urban environments. Proceedings of the Eurographics Workshop on Rendering Techniques, 83–88.

[19] Tecchia, F., Loscos, C., Chrysanthou, Y. (2002). Visualizing crowds in real-time. Computer Graphics Forum, 21(4), 753–765.

[20] Aubel, A., Boulic, R., Thalmann, D. (2000). Real-time display of virtual humans: levels of details and impostors. IEEE Transactions on Circuits and Systems for Video Technology, 10(2), 207–217.

[21] Kavan. L., Dobbyn, S., Collins, S., Zara, J., O'Sullivan, C. (2008). Polypostors: 2d polygonal impostors for 3d crowds. In proceedings of the 2008 symposium on Interactive 3D graphics and games, 149–155.

[22] McDonnell, R., Dobbyn, S., Collins, S., O’Sullivan, C. (2006). Perceptual evaluation of LOD clothing for virtual humans. In Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 117–126.

[23] S. Dobbyn, J. Hamill, K. O’Conor, C. O’Sullivan. (2005). Geopostors: a real time geometry/impostor crowd rendering system. International Conference on Computer Graphics and Interactive Techniques, 95–102.

[24] Grossman, J.P., & Dally, W.J. (1998). Point sample rendering. In Proceedings of the 9th Eurographics Workshop on Rendering, 181–192.

[25] Rusinkiewicz, S., & Levoy, M. (2000). Qsplat: a multiresolution point rendering system for large meshes. In proceedings of the 27th annual conference on Computer graphics and interactive techniques, 343–352.

[26] Pfister, H., Zwicker, M., Baar, J., Gross, M. (2000). Surfels: surface elements as rendering primitives. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pages 335–342.

[27] Marroquim, R., Kraus, M., Cavalcanti, P.R. (2008). Efficient image reconstruction for point-based and line-based rendering, Computers & Graphics, 32(2), 189-203.

[28] Rudomin, I., & Millan, E. (2004). Point based rendering and displaced subdivision for interactive animation of crowds of clothed characters. Virtual Reality Interaction and Physical Simulation Workshop, 139–148.

[29] Millan, E., & Rudomin, I. (2006). A comparison between impostors and point-based models for interactive rendering of animated models. In proceedings of the International Conference on Computer Animation and Social Agents.

[30] Hamill, J., McDonnell, R., Dobbyn, S., and O’Sullivan, C. (2005). Perceptual evaluation of impostor representations for virtual humans and buildings. Computer Graphics Forum, 24(3), 623–633.

[31] Maïm, J., Yersin, B., Pettré, J., Thalmann, D. (2009). YaQ: An Architecture for Real-Time Navigation and Rendering of Varied Crowds. IEEE Computer Graphics and Applications, 29(4), 44-53.

[32] Treuille, A., Cooper, S., Popovi´c, Z. (2006). Continuum crowds. Proceedings of ACM SIGGRAPH, 25(3) ,1160–1168.

[33] Yersin, B., Maim, J., Ciechomski, P.H., Schertenleib, S. Thalmann, D. (2005). Steering a Virtual Crowd Based on a Semantically Augmented Navigation Graph. In Proceedings of the First International Workshop on Crowd Simulation.

[34] Pettre, J., Laumond, J., and Thalmann, D. (2005). A navigation graph for real-time crowd animation on multilayered and uneven terrain.

[35] Bayazit, O.B., Lien, J.M., Amato, N.M. (2002). Better Group Behaviors in Complex Environments using Global Roadmaps. Artificial Life, 362-370.

[36] Reynolds, C.W. (1987). Flocks, herds and schools: A distributed behavioral model. In SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 25–34.

[37] Ulhaas, K.D., Erdmann, D., Gerl, O., Schulz, N., Wiendl, V., Andre, E. (2006). An Immersive Game - Augsburg Cityrun. André, E.; Dybkjær, L.; Minker, W.; Neumann, H.; Weber, M. (Eds.), Perception and Interactive Technologies (pp. 201-204). Berlin: Springer.

[38] Lamarche, F., & Donikian, S. (2004). Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. Computer Graphics Forum, 23(3), 509–518.

[39] Lerner, A., Chrysanthou, Y., Lischinski, D. (2007). Crowds by Example. Computer Graphics Forum, 26(3), 655-664.

[40] McDonnell, R., Larkin, M., Dobbyn, S., Collins, S., O'Sullivan, C. (2008). Clone attack! Perception of crowd variety. ACM SIGGRAPH, 27(3), 1–8.

[41] Allen, B., Curless, B., Popovic, Z. (2004) Exploring the space of human body shapes: data-driven synthesis under anthropometric control. (Tech. Rep. 2004-01-2188), USA, University of Washington, SAE Technical Papers.

[42] Seo, H., Cordier, F., Philippon, L., Thalmann, N.M. (2000). Interactive Modelling of MPEG-4 Deformable Human Body Models. IFIP Conference Proceedings, 196, 120-131.

[43] Thalmann, N.M., Seo, H., Cordier, F. (2004). Automatic modeling of virtual humans and body clothing. Journal of Computer Science and Technology, 19(5), 575-584.

[44] Ciechomski, P.H., Schertenleib, S., Maïm, J., Maupu, D., Thalmann, D. (2005). Real-time Shader Rendering for Crowds in Virtual Heritage. In VAST '05.

[45] Dobbyn, S., McDonnell, R., Kavan, L., Collins, S., and O’Sullivan, C. (2006). Clothing the masses: Real-time clothed crowds with variation. In Eurographics Short Papers, 103–106.

[46] McDonnell, R., Dobbyn, S., O’Sullivan, C. (2007). Pipeline for Populating Games with Realistic Crowds. International Journal of Intelligent Games and Simulation, 4(2), 1 - 15.

[47] Ulicny, B., Thalmann, D. (2001). Crowd simulation for interactive virtual environments and VR training systems. Proceedings of Eurographic workshop on Computer animation and simulation, 163-170.

[48] Thalmann, D., Musse, S.R., Kalmann, M. (2000). From Individual Human Agents to Crowds. Informatik Magazine, 1, 6-11.

[49] Heigeas, L., Luciani, A., Thollot, J., Castagné, N. (2003). A physically-based particle model of emergent crowd behaviors. In proceedings of Graphicon ‘03.

[50] Sung, M., Glechier, M., Chenney, S. (2004). Scalable Behaviors for Crowd Simulation. Computer Graphics Forum, 23(3), 519-528 .

[51] Braun, A., Bodmann, E.J.B., Musse, S.R. (2005). Simulating virtual crowds in emergency situations. Virtual Reality Software and Technology, 244-252.

[52] Sakuma, T., Mukai, T., Kuriyama, S. (2005). Psychological model for animating crowded pedestrians: Virtual Humans and Social Agents. Computer Animation and Virtual Worlds, 16(3-4), 343-351.

[53] Pelechano, N., O'Brien, K., Silverman, B., Badler, N. (2005). Crowd Simulation Incorporating Agent Psychological Models, Roles and Communication. First International Workshop on Crowd Simulation: V-CROWDS.

[54] Pelechano, N., Allbeck, J.M., Badler, N.I. (2007). Controlling individual agents in high-density crowd simulation. In SCA’07, 99-108.

[55] O’Sullivan, C., Cassell, J., Vilhjámsson, H., Dingliana, J., Dobbyn, S., McNamee, B., Peters, C., Giang, T. (2003). Levels of detail for crowds and groups. Computer Graphics Forum, 21(4): 733–741.

[56] Chittaro, L., Serra, M. (2004). Behavioral programming of autonomous characters based on probabilistic automata and personality. Computer Animation and Virtual Worlds,15(3-4), 319-326.

[57] Badler, N., Allback, J., Zhao, L., Byun, M. (2002). Representing and Parameterizing Agent Behaviors. In Proceedings of Computer Animation, 133-143.

[58] Bécheiraz, P., Thalmann, D. (1998). A behavioral animation system for autonomous actors personified by emotions. In Proceedings of First Workshop on Embodied Conversational Characters '98, 57-65.

[59] Ayesh, A., Stokes, J., Edwards, R. (2007). Fuzzy individual model (FIM) for realistic crowd simulation: preliminary results. In Proceedings of FUZZ-IEEE Conference '07, 1–5.

[60] Rudomín, I., Millán, E. (2004). XML scripting and images for specifying behavior of virtual characters and crowds. In Proceedings of CASA '04, 121-128.

[61] Rudomín, I., Millán, E. (2005). Probabilistic, layered and hierarchical animated agents using XML. In Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia GRAPHITE'05, 113-116.

[62] Peters, C., Ennis, C. (2009). Modeling Groups of Plausible Virtual Pedestrians. IEEE Computer Graphics & Applications, 29(4), 54-63.

[63] Reynolds, C. (2006). Big Fast Crowds on PS3. Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, 113-121.

[64] Cell BE Development Made Simple. RapidMind Inc. (2006). Technology Paper.

[65] Quinn, M.J., Metoyer, R.A., and H. Zaworski. (2003). Parallel Implementation of the Social Forces Model. Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, 63-74

[66] Steed, A., & Abou-Haidar, R. (2003). Partitioning crowded virtual environments. Virtual Reality Software and Technology, 7-14.

[67] Zhou, B., & Zhou, S. (2004). Parallel Simulation of Group Behaviors. Proceedings of the 2004 Winter Simulation Conference. 1, 364–370.

[68] Berg, J., Patil, S., Seawall, J., Manocha, D., Lin, M. (2008). Interactive navigation of individual agents in crowded environments. Proc. of ACM Symposium on Interactive 3D Graphics and Games, 139–147.

[69] Stephen, Chhugani, J., Kim, C., Satish, N., Lin, M., Manocha, D., and Dubey, P. (2009). Clearpath: highly parallel collision avoidance for multi-agent simulation. In

SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 177-187.

[70] Courty, N., Musse, S.R., (2004). FASTCROWD: Real-Time Simulation and Interaction with Large Crowds based on Graphics Hardware". Short Paper in ACM SCA 2004 - Symposium on Computer Animation, 2004. Tech. rep., INRIA, March.

[71] Erra, U., De Chiara, R., Scarano, V., and Tatafiore, M. (2004). Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In Proceedings of Vision, Modeling and Visualization 2004 (VMV).

[72] Paul, R. and Daniela, R. (2008). A high performance framework for agent based pedestrian dynamics on gpu hardware. In Proceedings of EUROSIS ESM 2008 (European Simulation and Modelling).

[73] D'Souza, R. M., Lysenko, M., and Rahmani, K. (2007). SugarScape on steroids: simulating over a million agents at interactive rates. Proceedings of Agent2007 conference.

[74] 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 Seventh Brazilian Symposium on Computer Games and Digital Entertainment, 81-86.

[75] Joselli, M., Zamith, M.P.M., Passos, E., Clua, E.W.G., Montenegro, A.A., Feijo, B. (2009). A Neighborhood Grid Data Structure for Massive 3D Crowd Simulation on GPU. SBGames, Rio de Janeiro.

[76] Karthikeyan, M. (2008). Real time crowd visualization using the GPU. USA, Virginia Polytechnic Institute and State University.

[77] Strippgen, D., Nageli K. (2009). Using common graphics hardware for multi-agent traffic simulation with CUDA. In proceedings of the 2nd International Conference on Simulation Tools and Techniques, Article No. 62.

[78] NVIDIA. Whitepaper:NVidia's Next Generation CUDA Compute Architecture, Fermi. Retrieved January 11, 2010, from http://www.nvidia.com/content/PDF/fermiwhitepapers/Whitepaper.pdf

[79] Glaskowsky, P.N. (2009). NVIDIA's Fermi: The First Complete GPU Computing Architecture, http://www.nvidia.com/content/PDF/fermi_white_papers -/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf.

[80] Balevic, A., Rockstroh, L., Wroblewski, M., Simon, S. (2008). Using Arithmetic Coding for Reduction of Resulting Simulation Data Size on Massively Parallel GPGPUs. Recent Advances in Parallel Virtual Machine and Message Passing Interface, 295-302.

[81] Roger, D., Assarsson, U., Holzschuch, N. (2007). Efficient stream reduction on the GPU. In: Kaeli, D., Leeser, M. (eds.) Workshop on General Purpose Processing on Graphics Processing Units.

[82] Horn, D. (2005). Stream reduction operations for GPGPU applications. Pharr, M. (Eds.), GPU Gems 2 (pp. 573-589). Addison-Wesley.

[83] Harris, M., Sengupta, S., Owens, J.D. (2007). Parallel prefix sum (scan) with CUDA. Nguyen, H. (Eds.), GPU Gems 3 (ch. 39). Addison-Wesley.

[84] Billeter, M., Olsson, O., Assarsson, U., Efficient stream compaction on wide SIMD many-core architectures. Proceedings of the Conference on High Performance Graphics, 159-166.

[85] Zadeh, L.A. (1965). Fuzzy sets. Information and Control, 8(3), 338-353.

[86] Chang, J, Li, T. (2008). Simulating virtual crowd with fuzzy logics and motion planning for shape template. In Proceedings of IEEE Conference on Cybernetics and Intelligent Systems, 131-136.

[87] Ahn, E.Y., Kim, J.W., Kwak, N.Y., Han, S.H. (2005). Emotion-based crowd simulation using fuzzy algorithm. AI 2005: Advances in Artificial Intelligence, 330-338.

[88] Yılmaz, E., İşler, V., Yardımcı, Y.Ç., (2009). The Virtual Marathon: Parallel Computing Supports Crowd Simulations. IEEE Computer Graphics & Applications, 29(4), 26-33.

[89] Zambetta, F. (2007). Simulating sensory perception in 3D game characters. In Proceedings of the 4th Australasian conference on Interactive entertainment, article no:7.

[90] Byl, P.B. (2004). Programming Believable Characters for Computer Games. Boston: Charles River Media.

[91] Ross, T. (2005). Fuzzy Logic with Engineering Applications. USA: WileyBlackwell.

[92] Mamdani, E. H. (1974). Application of fuzzy algorithms for control of simple dynamic plant. Procedings of IEEE, 121(12):1585-1588.

[93] Sugeno, M., & Kang, G. (1988). Structure Identification of Fuzzy Model. Fuzzy Sets and Systems, 28, 15-33.

[94] Tsukamoto, Y. (1979). An Approach to Fuzzy Reasoning Method. M.Gupta, R.Ragade, and R.Yager (eds.), Advances in Fuzzy Set Theory and Applications (pp.137-149), Amsterdam: Elsevier.

[95] Fuzzy Control Programming. International Electrotechnical Commission (IEC) (1997), http://www.fuzzytech.com/binaries/ieccd1.pdf

[96] Acampora, G., & Loia V. (2005). Using FML and Fuzzy Technology in Adaptive Ambient Intelligence Environments. International Journal of Computational Intelligence Research, 1(2), 171-182.

[97] Boyko, R.H., Boyko, A.R., Boyko, M.G. (2007). Referee bias contributes to home advantage in English Premiership football. Journal of Sports Sciences, 25(11), 1185-1194.

[98] Farkas, I., Helbing, D., Vicsek, T. (2002). Social behaviour:Mexican waves in an excitable medium. Nature 419, 131-132.

[99] Clements, R.R., Hughes, R.L. (2004). Mathematical Modelling of a Medieval Battle: The Battle of Agincourt, 1415. Mathematics and Computers in Simulation. 64(2), 259-269.

[100] Wikipedia, Battle of Agincourt,
http://en.wikipedia.org/wiki/Battle_of_Agincourt

[101].Wikipedia, Battle of Thermopylae,
http://en.wikipedia.org/wiki/Battle_of_Thermopylae

[102] Gareth, R. (1995). The Physics of Medieval Archery, Stortford Archery Club Newsletter, Issues 5 & 6.

[103] Longbow-Archers, http://www.longbow-archers.com/heavybowarchers.htm

[104] Bourg, D.M. (2002). Physics for Game Developers. Sebastopol: O'Reilly.

# CURRICULUM VITAE

## PERSONAL INFORMATION

Surname, Name: Yılmaz, Erdal
Nationality: Turkish (TC)
Date and Place of Birth: 2 February 1971 , Ankara
Marital Status: Married
Phone: +90 312 479 28 55
Fax: +90 312 210 22 91
email: erdal@ii.metu.edu.tr

## EDUCATION

| **Degree** | **Institution** | **Year of Graduation** |
|---|---|---|
| MS | METU Informatics Institute | 2003 |
| Certificate Program for Army Officers | METU Computer Engineering | 1999 |
| BS | General Command of Mapping, Survey Engineering School | 1995 |
| BS | Army College, Civil Engineering Dept. | 1993 |
| High School | Maltepe Military High School, Ankara | 1989 |

## WORK EXPERIENCE

| **Year** | **Place** | **Enrollment** |
|---|---|---|
| 1993- Present | General Command of Mapping | Project Officer |

## FOREIGN LANGUAGES
Fluent English

**PUBLICATIONS**

**Journal Papers:**

[1] Yılmaz, E., İşler, V., Yardımcı, Y.Ç. (2009). The Virtual Marathon: Parallel Computing Supports Crowd Simulations. IEEE Computer Graphics & Applications, 29(4), 26-33.

**Conference Papers:**

[1] Yılmaz, E., Maras, H.H., Yardimci, Y.Ç. (2004). PC based generation of Real-Time Realistic Synthetic Scenes for Low Altitude Flights. Proceedings of the SPIE, 5424, 31-39.

[2] Yılmaz, E., Maras, H.H., Yardimci, Y.Ç. (2004). Photorealistic Scene Realistic Scene Generation for PC Based Real-Time Outdoor Virtual Reality Applications. Proceedings of the XXth ISPRS Congress, 615-620.

[3] Yılmaz, E., Yardımcı, Y. (2005). Photorealistic Outdoor Scene Visualizations with PCs. Proceedings of the 50th Photogrammetry Week, 273-282.

[4] Yılmaz, E., Cagiltay, K. (2005). History of Digital Games in Turkey. Proceedings of the DIGRA Conf. 2005 (Changing Views: World in Play), p. Online.

[5] Karaahmetoğlu, C., Yılmaz, E., Yardımcı, Y.Ç., Köksal, G. (2006). Out-the-window scene properties in pc-based helicopter simulators. Enhanced and Synthetic Vision 2006.

[6] Yılmaz, E., Yardımcı, Y., Erdem, Ç., Erdem, T., Özkan, M. (2006). Music Driven Real-Time 3D Concert Simulation. Günsel, B., Jain, A.K., Tekalp, A.M., Sankur, B. (Eds.), MRCS (pp. 379-386). Berlin: Springer.

[7] Yılmaz, E., Ocak, M., Taştan, H. (2006). A Practical Approach for Serving Large Amounts of Geospatial Data via Computer Networks. Proceedings of the International Society for Photogrammetry and Remote Sensing (ISPRS) Second International Symposium on Geo-information for Disaster Management, Goa, India, p. Online , 2006.

[8] Kose, K., Grammalidis, N., Yilmaz, E., Cetin, E. (2008). 3D Forest Fire Propagation Simulation. 3DTV-CON 2008, Istanbul, Turkey, 369-372.

[9] Zabulis, X., Grammalidis, N., Bastanlar, Y., Yilmaz, E., Yardimci, Y.Ç. (2008). 3D Scene Reconstruction Based on Robust Camera Motion Estimation. 3DTV-CON2008, Istanbul, Turkey, May 2008, p. 53-56.

[10] K. Kose, N. Grammalidis, E.Yilmaz,E. Cetin: "3D Wildfire Simulation System", Proceedings of the ISPRS2008, Beijing, China, 1431-1436.

[11] Bastanlar, Y., Grammalidis, N., Zabulis, X., Yilmaz, E., Yardimci, Y.Ç., Triantafyllidis, G. (2008). 3D Reconstruction for a Cultural Heritage Virtual Tour System. Proceedings of the ISPRS2008, Beijing, China, 1023-1028.

[12] Köse, K., Yılmaz, E., Çetin, E. (2009). Progressive Compresion Of Digital Elevation Data Using Meshes. Proceedings of the IEEE International Symposium on Geoscience and Remote Sensing IGARSS 2009, Johannesburg, South Africa, p. Online.

**(National)**

**Journal Papers:**

[1] Taştan, H., Maraş, H., Şahin, K., Kurt, M., Ünlü, T., Çağlar, Y., Yılmaz, E. (2000). Sayısal Harita Destekli Askeri Uygulamalar Yazılımı. Harita Dergisi.

**Conference Papers:**

[1] Taştan, H., Maraş, H., Şahin, K., Kurt, M., Ünlü, T., Çağlar, Y., Yılmaz, E. (2001). Sayısal Harita Destekli Askeri Uygulamalar Yazılımı. 1nci Uluslararası Uzay Sempozyumu Bildiriler Kitabı, 301-310.

[2] Şahin, D., Yılmaz, E., Yardımcı, Y., Leblebicioglu, K. (2003). Hava Muharebesinin Modellenmesi ve Gorsellestirilmesi. MODSIM Calistayi, ODTÜ, Ankara.

[3] Yılmaz, E., Maraş, H., Yardımcı, Y.Ç. (2003). Sanal Kum Sandığı. MODSIM Calistayi, ODTÜ.

[4] Yılmaz, E., Çağıltay, K. (2004). Türkiye'de Elektronik Oyunların Tarihçesi. TBD 21. Ulusal Bilişim Kurultayı Bildiriler Kitabı, ODTÜ, 159-166.

[5] Karaahmetoğlu, C., Yılmaz, E., Leblebicioglu, K., Yardımcı, Y.Ç. (2005). PC Tabanlı Helikopter Uçuş Simülatörlerinde Alçak İrtifa Taktik Uçuş Eğitimi İçin Gereken Pencere Dışı Görsel Bileşenlerin Özellikleri. USMOS 2005, ODTÜ, Ankara, 237-246.

[6] Karaahmetoğlu, C., Yardımcı, Y.Ç., Köksal, G., Yılmaz, E. (2006). Helikopter Simülatörleri İçin Görsel Bileşen Özellikleri. UHUK (Ulusal Havacılık ve Uzay Konferansı) Bildiriler Kitabı , Ankara.

[7] Kose, K., Yılmaz, E., Grammalidis, N., Aktug, B., Cetin, A.E., Ilhami, A. (2008). 3D Forest-Fire Spread Simulation System (3-Boyutlu Orman Yangını Yayılımı Sistemi). SIU 2008 Bildiriler Kitabı, Didim, Aydın.

[8] Yılmaz, E., & Erbaş, M. (2008). Konumsal Bilgi Görsel Sunumu ve Örnek Bir Uygulama. SAVTEK 2008, ODTÜ, Ankara, 381-389.

[9] Yılmaz, E. Kantar, F., Erbaş, M. (2009). Harita Genel Komutanlığının Konumsal Veri Sunum Uygulamaları. 1. BHİKP Sempozyumu, Ankara.

[10] Çetin, Y., Yılmaz, E., Yardımcı, Y.Ç. (2009). Üç Boyutlu Sanal Çevre Görsel İpuçlarının Helikopter Simülatörleri Açısından İncelenmesi. USMOS 2009, Haziran 2009, Ankara.

**HOBBIES**
Computer Technologies, Collectible Figures

# VITA

Erdal Yılmaz was born in Ankara on February 2, 1971. He received his B.S. degrees in Civil Engineering Department of Army College in 1993 and Survey Engineering School of General Command of Mapping in 1995. He attended a four-semester training program in Computer Engineering Department of METU in 1999. He received a MSc. Degree in Information Systems Department of Informatics Institute, METU in 2003. Since 1995 he has been worked in various departments of General Command of Mapping as project officer. His main areas of interest are computer graphics, computer games and GIS.