GPU Accelerated Path-planning for Multi-agents in Virtual Environments

Leonardo G. Fischer, Renato Silveira, Luciana Nedel
Institute of Informatics
Federal University of Rio Grande do Sul

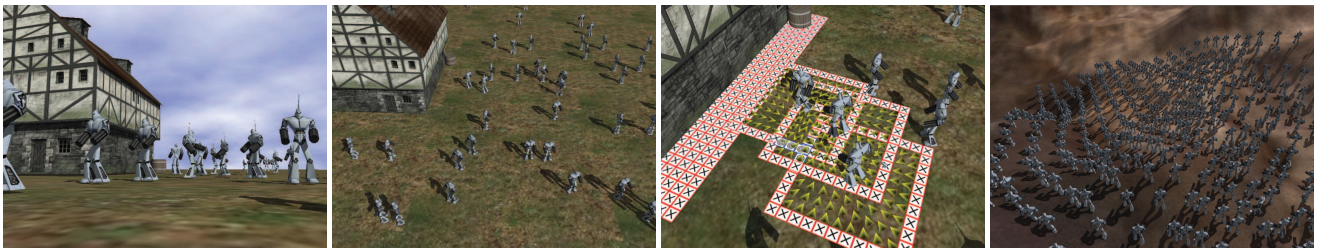


Figure 1: *Virtual characters controlled by the BVP Planner in a virtual environment.*

Abstract

Many games are populated by synthetic humanoid actors that act as autonomous agents. The animation of humanoids in real-time applications is yet a challenge if the problem involves attaining a precise location in a virtual world (path-planning), and moving realistically according to its own personality, intentions and mood (motion planning). In this paper we present a strategy to implement – using CUDA on GPU – a path planner that produces natural steering behaviors for virtual humans using a numerical solution for boundary value problems. The planner is based on the potential field formalism that allows synthetic actors to move negotiating space, avoiding collisions, and attaining goals, while producing very individual paths. The individuality of each character can be set by changing its inner field parameters leading to a broad range of possible behaviors without jeopardizing its performance. With our GPU-based strategy we achieve a speed up to 56 times the previous implementation, allowing its use in situations with a large number of autonomous characters, which is commonly found in games.

Keywords:: Path-planning, GPGPU, NVIDIA CUDA, Agent Simulation

Author's Contact:

{lgfischer,rsilveira,nedel}@inf.ufrgs.br

1 Introduction

Many types of games, specifically First Person Shooters (*FPS*) and Real Time Strategy (*RTS*) are populated by synthetic actors that should act as autonomous agents. Autonomous agents, also called *non-player characters*, are characters with the ability of playing a role into the environment with life-like and improvisational behavior. To behave in such way, the agents must act in the virtual world, perceive, react and remember their perceptions about this world, think about the effects of possible actions and finally, learn from their experience [Funge 2004]. In this complex and suitable context, navigation plays an important role [Nieuwenhuisen et al. 2007]. To move agents in a synthetic world, a semantic representation of the environment is needed, as well as the definition of the agent initial and target position (goal). Once these parameters were set, any path-planning algorithm can be used to find a trajectory to be followed.

However, in the real world, if we consider different persons (all in the same initial position) looking for achieving the same target position, each path followed will be unique. Even for the same task, the strategy used for each person to reach his/her goal depends on his/her physical constitution, personality, mood, reasoning, urgency, and so on. From this point of view, a high quality algorithm to move characters across virtual environments should generate expressive, natural and unexpected steering behaviors.

In contrast, the high performance required for real-time graphics applications compels developers to look for most efficient and less expensive methods that produce yet good and almost natural movements. To illustrate how performance is a crucial problem, it is known that to be playable, a game must run at least at a rate of 30-100 frames per second. This implies in 0.02 seconds per frame. Each frame (or step of an animation) includes the updating of the game status, handling user inputs, graphics processing, physics computations, strategic AI, path-planning, among others. Then, we can easily consider something as one millisecond per step for path-planning (with multi-core architectures, this restriction is relaxed).

Many researchers are working on methods to improve the quality of the steering behavior of synthetic agents with a minimal cost. One way to improve the performance is taking advantage of massively parallel architectures, as multi-core CPUs and GPUs (*Graphics Processing Unit*). In this work we propose a GPU implementation of the BVP Planner recently proposed by us [Dapper et al. 2007]. The BVP Planner is a method based on the numeric solution of the boundary value problem (BVP) to control the movement of pedestrians allowing the individuality of each agent.

Our main contributions in this paper are:

- A parallel version of our previously technique [Dapper et al. 2007], implemented on the GPU using nVIDIA CUDA (Compute Unified Device Architecture) [NVIDIA. 2009]
- A strategy to reduce the number of memory transactions between CPU and GPU
- Several tests showing that the GPU implementation improves up to 56 times the CPU sequential version, allowing the real-time use of this technique even in scenarios with a large number of autonomous characters

Despite *humanoid*, *autonomous agent*, and *behavior* are terms used in many different contexts, in this paper we limit its use in order to match our goals. For the sake of simplicity, we consider *humanoids* as a kind of embodied *autonomous agent* with reactive behaviors (driven by stimulus), represented by a computational model, and capable of producing physical manifestations in a virtual world. The term *behavior* will be used mainly as a synonymous of *animation* or *steering behavior* and intend to refer the improvisational and personalized action of a *humanoid*.

The remaining of this paper is structured as follows. Section 2 reviews some related works on path-planning techniques applied to virtual agents simulation. Section 3 describes the fundamentals of the path-planning method proposed by us. In Section 4 we detail the strategy used to handle the information about the environment and other agents. In Section 5 we present our strategy to implement

this technique on GPU. Section 6 shows our results, including several comparisons between the CPU and GPU version, and exposes considerations about performance. Finally, Section 7 presents our conclusions and some ideas for future works.

2 Related Work

The path-planning problem has been deeply explored in game development. The generation of a path between two known configurations in a bi-dimensional world is a well-known problem in robotics, artificial intelligence, and computer graphics field. However, to find the path is not enough when we want to endow artificial characters with natural and realistic movement similar to the ones found and followed by real human beings. When it comes to a game with many autonomous characters, for instance, these characters must also present convincing behavior. It is very difficult to produce natural behavior by using a strategy focusing on the global control of characters. On the other hand, taking into account the individuality of each character can be a costly task. As a consequence, most of the approaches proposed in computer graphics literature do not take into account the individual behavior of each agent.

An example is the technique proposed by Kuffner [James J. Kuffner 1998]. Kuffner proposed a technique where the scenario is mapped onto a 2D mesh and the path is computed using a dynamic programming technique like Dijkstra. Then, the motion controller is used to animate the agent along the path planned. Kuffner argue that his technique is fast enough to be used in dynamic environments. Another example is the work developed by Metoyer and Hodgings [Metoyer and Hodgins 2004]. They proposed a technique where the user defines the path that should be followed by each agent. During the motion along this path, it is smoothed and slightly changed to avoid collisions using force fields that act on the agent.

The development of randomized path-finding algorithms – specially the PRM (Probabilistic Roadmaps) [Kavraki et al. 1996] and RTT (Rapidly-exploring Random Tree) [LaValle 1998] – allowed the use of large and more complex configuration spaces to generate paths efficiently. Thus, the challenge becomes more the generation of realistic movements than finding a valid path. For instance, Choi et al. [Choi et al. 2003] use a library of captured movements associated to the PRM to generate realistic movements in a static environment, that is, live-captured motions are used insofar the agent tracks the path computed from a roadmap. Despite the fact the path is computed in a pre-processing phase, results are very realistic. Pettré et al. [Pettre et al. 2002] improved this idea adding one more step in this process. This step consists of smoothing the path computed by the PRM using Bézier curves. Hereinafter, the already captured motions are associated to the agent position during the path execution. As in previous works, the motion is also performed on a 2D environment.

Differently, Burgess and Darken [Burgess and Darken 2004] proposed a method based on the *principle of least action* which describes the tendency of elements in nature to seek the minimal effort solution. Authors claim that a realistic path for a human is the one that requires the smallest amount of effort. The method produces human-like movements, through very realistic paths, using properties of fluid simulation.

Tecchia et al. [Tecchia et al. 2001] proposed a platform that aims to accelerate the development of behaviors for agents through local rules that control these behaviors. These rules are governed by four different control levels, where each one reflects a different aspect of the behavior of the agent. Results show that, for a fairly simple behavioral model, the system performance can achieve interactive time.

Pelechano et al. [Pelechano et al. 2005] described a new architecture to integrate a psychological model into a crowd simulation system in order to obtain believable emergent behaviors. The architecture achieves individualistic behaviors through the modeling of the agent knowledge, as well as the basic principles of communication between agents.

Treuille et al. [Treuille et al. 2006] proposed a crowd simulator driven by dynamic potential fields which integrates both global navigation and local collision avoidance. Basically, this technique uses the crowd as a density field, and, for each group, constructs a unit cost field which is used to control people displacement. The method produces smooth behavior for a large amount of agents at interactive rates.

Recently, Reynolds [Reynolds 2006] implemented a high performance multi-agent simulation and animation for the Playstation® 3. Basically, his technique uses a spatial partitioning that divides the simulation into disjoint jobs which are evaluated in an arbitrary order on any number of Playstation® 3 Synergistic Processor Units (SPUs). A fine-grain partitioning suits SPU memory size and provides automatic load balancing. This approach allows a scalable multi-processor implementation of a large and fast crowd simulation, achieving good frame rates with thousand of agents.

In 2008, Bleiweiss [Bleiweiss 2008] implemented the Dijkstra and the A* algorithms using CUDA. Differently from our work, these algorithms are used in the path finding problem with pre-computed graphs. After several benchmarks, he observed that the Dijkstra implementation reached a speed up of 27 times compared to a C++ implementation without SSE instructions. The A* implementation reached a speed up of 24 times compared to the C++ implementation with SSE instructions.

Based on local control, van den Berg [van den Berg et al. 2008] proposed a technique that handles the navigation of multiple agents in the presence of dynamic obstacles. He uses an extended *velocity obstacles* concept to locally control the agents with few oscillation. Kapadia [Kapadia et al. 2009] presented a framework that enables agents to navigate in unknown environments based on *affordance fields* that compute all the possible ways an agent can interact with its environment.

As mentioned above, most of the approaches do not take into account the individual behavior of each agent, his internal state or mood. Our assumption is that realistic paths derive from human personal characteristics and internal state, thus varying from one person to another. As a consequence, we [Dapper et al. 2006; Dapper et al. 2007] recently proposed a technique that generate individual paths. Our path is smooth and is dynamically generated while the agent walks. In the following sections, we will explain the concepts of our technique and our strategy to implement it on the GPU.

3 Path Planner based on Boundary Value Problems

Recently, we [Dapper et al. 2006; Dapper et al. 2007] developed a technique that produces natural and individual behaviors for virtual humanoids. This technique is based on an extension of the Laplace's Equation that produces a family of potential field functions that do not have local minima. This family is generated through the numeric solution of a convenient partial differential equation with Dirichlet boundary conditions, i.e., a boundary value problem (BVP). Boundary conditions are central to the method indicating which regions in the environment are obstacles and which ones are targets. Our method uses the following equation

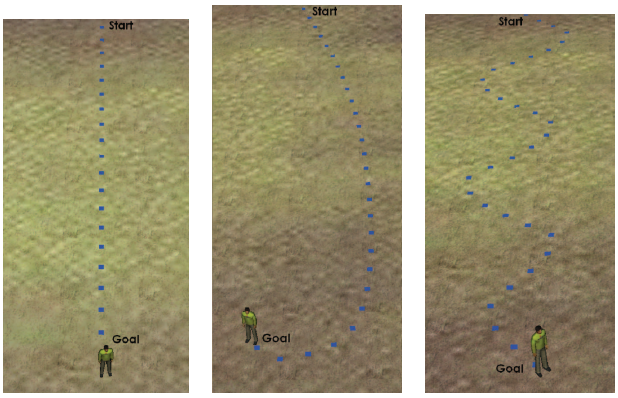
$$\nabla^2 \, p(\mathbf{r}) + \epsilon \mathbf{v}.\nabla p(\mathbf{r}) \; = \; 0 \qquad (1)$$

where $\mathbf{v}$ is a bias unity vector and ϵ is a scalar value.

The use of terms ϵ and $\mathbf{v}$ distort the potential field providing a preferred direction to be followed. This distortion allows the production of individual behaviors for humanoids illustrated through the path followed by each one during navigation tasks.

To generate realistic steering behaviors, we need to conveniently adjust both parameters ϵ and $\mathbf{v}$. The vector $\mathbf{v}$, called *behavior vector*, can be thought as an external force that pulls the agent to its direction always as possible whereas the parameter ϵ can be understood as the *strength* or *influence* of this vector in the agent behavior. The allowed values of parameters ϵ and $\mathbf{v}$ permit to generate an expressive amount of action sequences – *displacement sequences* – that virtual humanoids can use to reach a specific target position.

Figure 2 shows three different paths followed by an agent using the Equation 1 and changing the parameters ϵ and $\mathbf{v}$.



(a) (b) (c)

Figure 2: *Different paths followed by an agent using Equation 1: (a) path produced by harmonic potential, i.e., with $\epsilon = 0$; (b) with $\epsilon = -1.0$ and $\mathbf{v} = (1, 0)$; (c) with $\epsilon = -1.0$ and $\mathbf{v} = (1, sin(0.6t))$*

.

Two action sequences are not statically defined for a same pair ϵ and $\mathbf{v}$, i.e., the path generated vary according to the information gathered by the agent to allow it to dynamically react against unexpected events (e.g. dynamic obstacles). In other words, the configuration of the obstacles has an important role in the generation of the path.

Besides, this pair is not constrained to keep constant during the execution of tasks. They can vary insofar the agent displaces in the environment to obtain the desired behavior. Figure 2(c) shows a situation where the behavior vector varies according to a sin function. It is not natural for human beings to walk based on a sin function. However, the path based on a sin function illustrates the flexibility of Equation 1. Any function can be associated to $\mathbf{v}$ and ϵ to generate a behavior.

When $\epsilon = 0$, Equation 1 reduces to $\nabla^2\ p(\mathbf{r}) = 0$ which corresponds to Laplace's Equation. This equation is used as core of the path planner based on harmonic function developed by Connolly and Grupen [Connolly and Grupen 1993] on Robotics context. This planner produces paths that minimize the hitting probability of the agent with obstacles, i.e., in an indoor environment the agent will tend to follows a path equidistant to the walls, as shown in Figure 2(a). This behavior is not always adequate to simulate humanoid motion since it looks very stereotyped because humans do not always walk equidistant to the walls. Hence the importance of using these parameters ϵ and $\mathbf{v}$.

The common approach to numerically solve a BVP is to consider that the solution space is discretized in a regular grid. Each cell (i, j) is associated to a squared region of the real environment and stores a potential value $p^t_{i,j}$ at instant t. Each cell is distant from each other 1 unit. The Dirichlet boundary conditions previously associate a specific potential value to some cells, before the relaxation process is performed. That is, cells associated to obstacles in the real environment store a potential value equal to 1 (*high potential*) whereas cells containing the target store a potential value equal to 0 (*low potential*). The high potential value prevents the agent from running into obstacles whereas the low potential value generates an attraction basin that pulls the agent. The potentials of the other cells are computed using the Gauss-Seidel relaxation method, as discussed in [Prestes et al. 2002]. By considering the Equation 1, the potentials of the free space cells are updated through the following equation

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \quad (2)$$

where $p_c = p^{t+1}_{i,j}$, $p_b = p^t_{i,j+1}$, $p_t = p^{t+1}_{i,j-1}$, $p_r = p^t_{i+1,j}$, $p_l = p^{t+1}_{i-1,j}$ and $\mathbf{v} = (v_x, v_y)$. Figure 3 shows a representation of these cells.

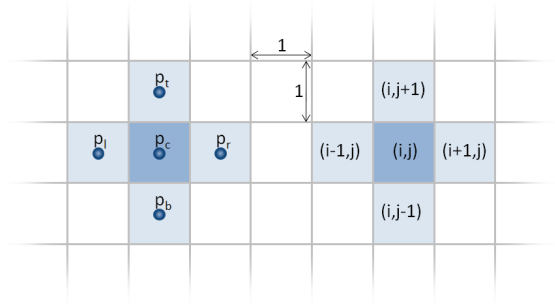


Figure 3: *Representation of p_c, p_b, p_t, p_r and p_l on the grid.*

The parameter $\mathbf{v}$ must be a unit vector and ϵ must be in the interval $(-2, 2)$. Values out of this range generate oscillatory and unstable paths that do not guarantee that the agent will reach the target or will avoid obstacles. This happens because the boundary conditions – that assert the agent is repelled by obstacles and attracted by targets – are violated.

After the potential computation, the agent moves following the direction of the gradient descent of this potential at its current position (i, j),

$$(\nabla \mathbf{p})_{(i,j)} = \left(\frac{p_{i+1,j} - p_{i-1,j}}{2}, \frac{p_{i,j+1} - p_{i,j-1}}{2} \right)$$

This process is an intuitive way to control the agent motion. However, it can easily fail in producing realistic steering behaviors, as observed in real world. One of the reasons is that the agent changes its direction based solely on the gradient descent of its position. For instance, if the field of view of the agent is small, its reaction time will be very short to treat dynamic obstacles[1]. Then, these obstacles will produce a strong repel force that will change the agent direction abruptly. As we can see in Figure 4, if the agent uses only the gradient descent (*dgrad*) it will change its direction in nearly $\pi/2$.

We handle this problem by adjusting the current agent position by

$$\Delta\ \mathbf{d} = \upsilon(\cos(\varphi^t), \sin(\varphi^t)) \quad (3)$$

where υ defines the maximum agent speed and φ^t is

$$\varphi^t = \eta\ \varphi^{t-1} + (1 - \eta)\ \zeta^t \quad (4)$$

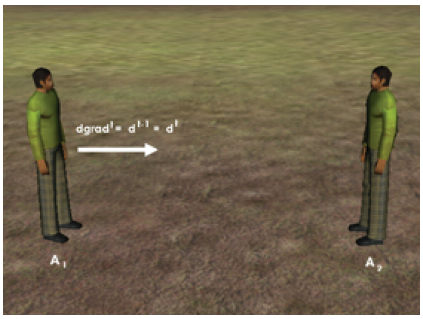
where $\eta \in [0, 1)$ and ζ is the orientation of the gradient descent at current agent position.

When $\eta = 0$, the agent adjusts its orientation using only information about the gradient descent. If $\eta = 0.5$, the previous agent direction (φ^{t-1}) and the gradient descent direction influence equally the computation of the new agent direction. Figure 4(b) shows the vector $\mathbf{d}^t$ with orientation φ^t computed with $\eta = 0.5$. The parameter η can be viewed as an inertial factor that tends to keep the agent direction constant insofar $\eta \to 1$. When $\eta \to 1$, the agent reacts slowly to unexpected events, increasing its hitting probability with obstacles. η is a flexible parameter that the user is able to control. However, a learning strategy could be used to specify what is the best η to a specific situation.

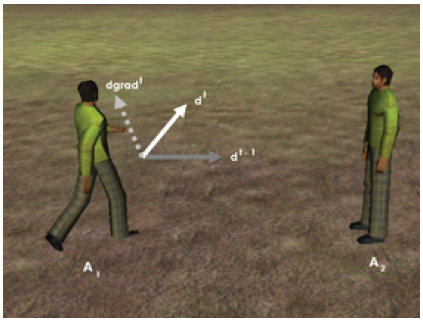
Despite Equation 3 produces good results and smooth paths in environments with few obstacles, when the environment is cluttered with obstacles, the agent behavior is not realistic and collisions can happen. To solve this problem, a speed control was incorporated into this equation,

$$\Delta\ \mathbf{d} = \upsilon\ (\cos(\varphi^t), \sin(\varphi^t))\ \Psi(|\varphi^{t-1} - \zeta^t|) \quad (5)$$

[1]We consider that dynamic obstacles (as other agents) are mapped in the environment only when they are inside the field of view of the agent, which almost corresponds to reality.



(a)



(b)

Figure 4: *Defining agent motion. (a) Situation before the agent A_2 enters in the field of view of A_1. (b) If the agent A_1 follows the direction defined by the gradient descent (*dgrad*), it will changes its direction in nearly $\pi/2$, what is undesirable. However, if the agent uses the vector d, it will achieve a smooth curve, what is more natural and realistic.*

where function $\Psi : \mathbf{R} \rightarrow \mathbf{R}$ is

$$\Psi(x) = \begin{cases} 0 & \text{if } x > \pi/2 \\ \cos(x) & \text{, otherwise} \end{cases} .$$

If $|\varphi^{t-1} - \zeta^t|$ is higher than $\pi/2$, then there is a high hitting probability and this function returns the value 0, making the agent stops. Otherwise, the agent speed will change proportionally to the collision risk. In regions cluttered with obstacles, agents will tend to move slowly. If a given agent is about to cross the path of another, one of them will stop and wait until the other get through. Furthermore, speed control allows the simulation of agents' mood through the variation of the speed magnitude, that is, it is possible to simulate a tired agent making it move slower and an agent that is anxious about its work making it move faster.

4 Implementation Strategy

As previously explained, our motion planning method requires the discretization of the environment into a regular grid. In this section we present the strategy that was used in our previous work [Dapper et al. 2006; Dapper et al. 2007] to implement it by using global environment maps (one for each target) and local maps (one for each agent), as well as the mechanisms used to control each agent steering behavior.

4.1 Environment Global Map

The entire environment is represented by a set of homogeneous meshes, $\{\mathcal{M}_k\}$, in which each mesh $\mathcal{M}_k$ has $L_x \times L_y$ cells, denoted by $\{\mathcal{C}^k_{i,j}\}$. Each cell $\mathcal{C}^k_{i,j}$ corresponds to a squared region centered in environment coordinates $r = (r_i, r_j)$ and stores a particular potential value $\mathcal{P}^k_{i,j}$. The potential associated to the mesh $\mathcal{M}_k$ is computed by the harmonic path planner, through the Equation 2, and then used by agents to reach the target $\mathcal{O}_k$.

In order to delimit the navigation space, we consider that the environment is surrounded by static obstacles. Global maps are built before simulation starts, in a pre-processing phase.

4.2 Agent Local Map

Each agent a_k has one map m_k that stores the current local information about the environment obtained by its own sensors. This map is centered in the current agent position and represents a small fraction of the global map, usually about 10% of the total area covered by the global map.

The map m_k has $l^k_x \times l^k_y$ cells, denoted by $\{c^k_{i,j}\}$ and divided in three regions: the update zone (*u-zone*); the free zone (*f-zone*) and the border zone (*b-zone*), as shown in Figure 5. Each cell corresponds to a squared region centered in environment coordinates $r = (r_i, r_j)$ and stores a particular potential value $p^k_{i,j}$.

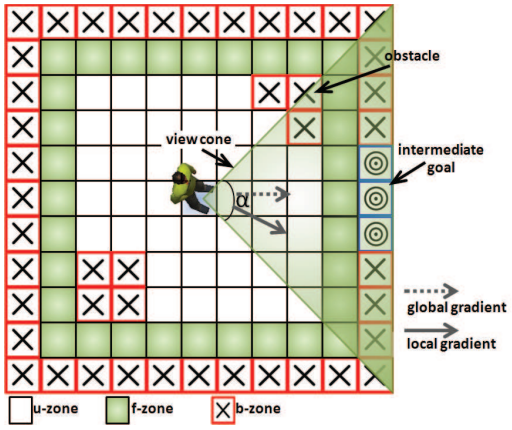


Figure 5: *Agent Local Map. The update (u-zone), free (f-zone) and border zones (b-zone) are shown. Blue and red cells correspond to the intermediate goal and obstacles, respectively.*

The area associated to each agent map cell is smaller than the area associated to the global map cell. The main reason is that the agent map is used to produce refined motion, while the global map is used only to assist the long-term agent navigation. Hence, the smaller the size of the cell on the local map, the better the quality of motion.

4.3 Updating Local Maps from Global Maps

For each agent a_k, a goal $\mathcal{O}_{goal(k)}$[2], a particular vector $\mathbf{v}_k$ that controls its behavior, and a ϵ_k should be stated. The same goal, $\mathbf{v}$, and ϵ can be designated to several agents. If $\mathbf{v}_k$ or ϵ_k is dynamic, then the function that controls it must also be specified.

To navigate into the environment, an agent a_k uses its sensors to perceive the world and to update its local map with information about obstacles and other agents. The agent sensor sets a view cone with aperture α.

Figure 6 exemplifies a particular instance of the agent local map where we can see the obstacles mapped from the global map. The *u-zone* cells $c^k_{i,j}$ which are inside the view cone and correspond to obstacles or other agents have their potential value set to 1. In Figure 7, as there is an agent in the *u-zone* of the agent local map, inside of his view cone, it is mapped as an obstacle into his local map. This procedure assures that dynamic or static obstacles behind the agent (out of his view cone) do not interfere in his future motion.

For each agent a_k, the global descent gradient on the cell in the global map $\mathcal{M}_{goal(k)}$ that contains his current position is calculated. The gradient direction is used to generate an intermediate goal in the border of the local map, setting the potential values of a couple of *b-zone* cells to 0, while the other *b-zone* cells are considered as obstacles, with their potential values set to 1. In Figure 7, the agent calculates his global gradient in order to project an intermediate goal in its own local map. As the agent local map is delimited by obstacles, the agent is pulled towards the intermediate goal using the direction of his local gradient. The intermediate goal helps the agent a_k to reach its target $\mathcal{O}_{goal(k)}$ while allowing it to produce a particular motion.

[2]Function *goal()* maps the agent number k into its current target number

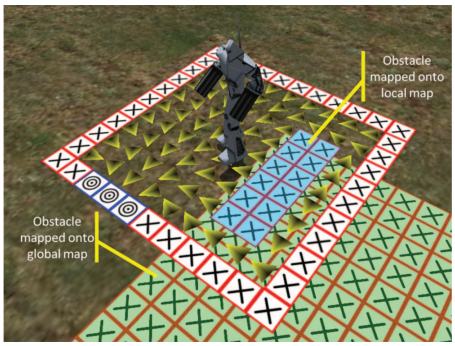


Figure 6: *Global map mapped onto the agent local map.*

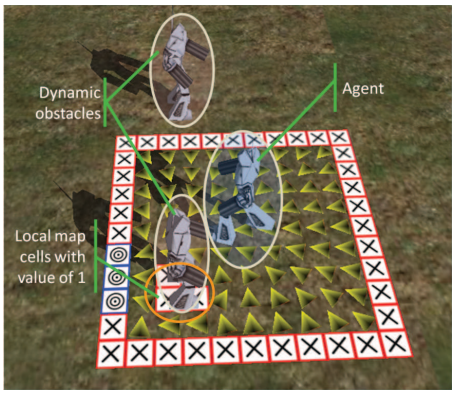


Figure 7: *The cells which are inside the agent's view cone and correspond to obstacles or other agents have their potential value set to* 1.

In some cases, the target $\mathcal{O}_{goal(k)}$ is inside both view cone and *u-zone*, and consequently local map cells associated are set to 0. The intermediate goal is always projected, even if the target is mapped onto the *u-zone*. Otherwise the agent can easily get trapped because it would be taking into consideration only the local information about the environment, in a same way as traditional potential fields [Khatib 1980].

F-zone cells are always considered free of obstacles, even when there are obstacles inside. The absence of this zone may close the connection between the current agent cell and the intermediate goal due to the mapping of obstacles in front of the intermediate goal. When this occurs, the agent gets lost because there is no information coming from the intermediate goal to produce a path to reach it. *F-zone* cells handle the situation always allowing the propagation of the goal's information to the cells associated to the agent position.

After the sensing and mapping steps, the agent k updates the potential value of its map cells using Equation 2 with its pair $\mathbf{v}^k$ and ϵ^k. Hereinafter, it updates its position according to Equation 5 using the gradient descent computed from the potential field stored on its local map in the position $p_x = \lceil l_x^k/2 \rceil$ and $p_y = \lceil l_y^k/2 \rceil$.

5 Implementation on GPU

In the real world, people walking inside a room react to what they perceive from the environment based on their own personality, mood and reasoning, i.e., they think in parallel. So, a technique that handles several agents should be parallelized in the same way.

According to Section 3, during the update phase of our technique, each agent must update its local map with the environment obstacles which are inside this region. Note that, in this step, we consider that for a given agent a_i, each other agent $a_j, i \neq j$, is also an obstacle. Then, each cell in the agent local map inside his view cone is updated as an obstacle, with the potential value equal to 1. After the update of these cells, we update the cells which correspond to the agent goal, with the potential value of 0.

Note that each one of these updates can be made in parallel between the agents. The only dependency here is that obstacle cells must be updated before goal cells. It must be done sequentially, otherwise, if an agent has a goal very close to an obstacle, both obstacle and goal may be mapped to the same cell. In this case, if goal cells are updated before obstacle cells, the agent will become lost, without a goal to achieve. All other cells are updated as free cells.

Afterwards, the Equation 2 is evaluated for each agent local map. Since it is difficult and needs to be evaluated independently for each agent, it is a good candidate for a parallel implementation. The Gauss-Seidel relaxation method (previously used in Equation 2) is not suitable for a parallel implementation because it uses values from the current and previous iterations. In a sequential approach, it is very simple to implement and fast to execute, but a parallel implementation will require some kind of synchronization, which may cause degradation in performance. A better approach for a parallel implementation is to use values only from the previous iteration. This is exactly what the Jacobi method does. The update rule is described below.

$$p_c = \frac{p_b + p_t + p_r + p_l}{4} + \frac{\epsilon((p_r - p_l)v_x + (p_b - p_t)v_y)}{8} \quad (6)$$

where $p_c = p^t_{i,j}$, $p_b = p^t_{i,j+1}$, $p_t = p^t_{i,j-1}$, $p_r = p^t_{i+1,j}$, $p_l = p^t_{i-1,j}$ and $\mathbf{v} = (v_x, v_y)$.

We implemented the parallel version of our technique using the nVIDIA® Cuda [NVIDIA. 2009] language, which allows us to use the graphics processor without using shading languages. In the context of CUDA, the CPU, here called *Host*, controls the graphics processor, called *Device*. It sends data, calls the *Device* to execute some functions, and then copies back its results.

Each graphics processor of a nVIDIA graphics card is divided into several multiprocessors. Cuda divides the processing in blocks, where each block is divided in several threads. Each block of threads is mapped to one multiprocessor of the graphics processor. When the *Host* calls the *Device* to execute a function, it needs to inform how the work will be divided in blocks and threads. Maximum performance is achieved when we maximize the use of blocks and threads for a given graphics processor.

Each of the multiprocessors is a group of simple processors that share a set of registers and some memory (the *shared memory* space). The shared memory size is very small (16KB on graphics cards up to *Compute Capability 1.3*), but it is as fast as the registers. The communication between two multiprocessors must be done through the Device Memory, which is very slow if compared to the shared memory. There is also the *Constant Cache* and *Texture Cache memory*, which has better access times than the Device memory, but it is read-only for the *Device*.

Before the execution of the code in the *Device*, the *Host* must send the data to its Device Memory to be processed later. The memory copy from the Host Memory to the Device memory is a slow process, and should be minimized. Besides, the nVIDIA Cuda Programming Guide [NVIDIA. 2009] says that one single call to the memory copy function with a lot of data is much more efficient than several calls to the same function with a few bytes. We can improve the performance of our application making good use of these restrictions of Cuda.

As previously mentioned, each agent a_k has several attributes: the scalar ϵ_k, the vector $\mathbf{v}_k$, and its current objective $\mathcal{O}_{goal(k)}$. The local map also has some attributes, like its width l_x^k and height l_y^k. All these attributes must be sent at least once to the *Device*. The agent goal and the local map position in the world, for instance, will be frequently updated. To avoid several memory transactions between the *Host* and the *Device*, we store all these attributes in contiguous memory areas, and treat it like an array. At the position k we store an attribute of the agent a_k. Proceeding this way, we avoid several unnecessary copies, improving the overall performance.

Figure 8 shows our data structure for a set of 3 agents. The array m with all local map cells is illustrated with its cell's index. Each position k of the array $\mathbf{s}$ contains an index to the first position in the array $\mathbf{m}$ in which the agent a_k local map information is stored. Each position k of the array $\mathbf{l}$, $\mathbf{O}$, ϵ, $\mathbf{v}$ contains the information of the

local map dimension and goal, as well as the behavioral parameters ϵ and $\mathbf{v}$ of the agent a_k, respectively.

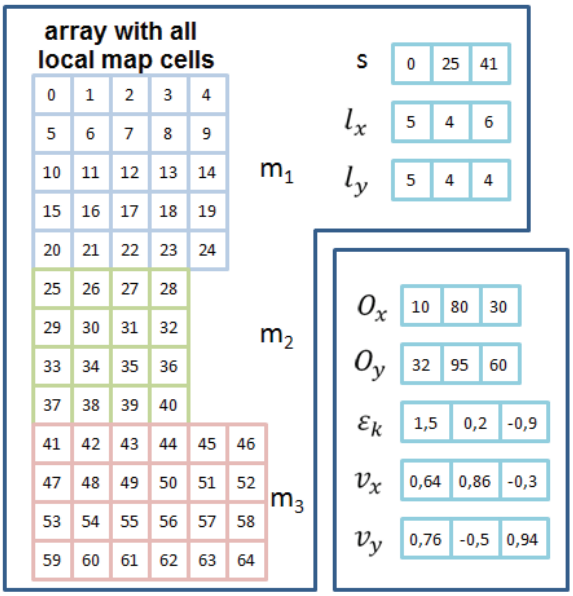


Figure 8: *Data structure used on GPU.*

There are situations in which the size of the agent local map must be changed. Any update on the size of an agent local map will require the modification of the array m, which implies in the entire data structure reconstruction. In these cases, the *Host* must reallocate the entire array in the Host Memory, and send it again to the Device Memory. These attributes should not only be copied once to the Device memory, but they should be sent to the Constant Memory or Texture Memory.

As the *Environment Global Map* is composed only of static obstacles, it can be copied to the Device Memory only once. Then, the update step can be done in the following way. First, each local map is mapped into a block of threads, in which each thread updates one cell of the local map. The thread will find the local map cell corresponding to the Environment Global Map, and will copy the information from the global map cell to the local map cell. This is done only to the cells in the *f-zone*. Figure 6 illustrates this situation.

Afterwards, each local map is mapped to a block of threads, and each thread is associated with a dynamic obstacle. This thread checks whether the obstacle appears inside the view cone. If yes, the local map cells occupied by the obstacle update its potential value to 1. Next, each cell in the *b-zone* is mapped as an obstacle, also updating its potential to 1, except for the ones that are goal cells. The remaining cells are updated as free cells. Then, the Equation 6 can be evaluated, starting one thread to each local map cell. A synchronization must be made between the iterations in order to guarantee that all cells are up to date to the next iteration.

The convergence of the Equation 6 is achieved through several *reads* and *writes* at the Device Memory during several iterations. In order to avoid the high latency of the Device Memory, this must be made in the shared memory of the multiprocessor. An implementation of the Jacobi method will require two copies of the potential map, where at each iteration the values are *read* from one of them and *written* to the other. However, the shared memory size is very limited. Then, we decided to use a combination of the Jacobi method with the Gauss-Seidel. In our implementation, only one copy of the potential map is stored in the shared memory. At each iteration t, a cell $c^k_{i,j}$ may be updated with the potential of the neighborhood cells at the iteration $t-1$ or t. We do not specify whether will be used values from iteration $t-1$ or t. It will depend on how the information will be arranged in the shaders, i.e., the synchronization between cells update is not needed.

6 Results

In order to verify that our parallel implementation can be executed faster than the sequential one, a couple of tests were accomplished. All the tests were executed in an Intel® Core 2 6300 1.86GHz, with 2Gb of RAM memory, a nVIDIA GeForce 9800 GX2 graphics card (the graphics processor has 600 MHz of clock) and Microsoft Windows XP SP3 operating system. We measured how many times per second the algorithm can be executed, and what is the impact of the memory copy between the *Host* and the *Device*, using three different sizes for the local maps.

The tests were executed in the following way. Initially, three sizes of local maps where chosen: 11×11, 16×16 and 21×21. We chose these sizes because previous tests [Dapper et al. 2006] showed that they generate animations with very good quality, being the most interesting for tests. Then, several scenarios were executed using the parallel and sequential versions of the algorithm, changing the number of agents in the scene. For each test, we recorded the frequency at which the algorithm can be executed, and the percentage of time spent in memory copies between the *Host* and the *Device*.

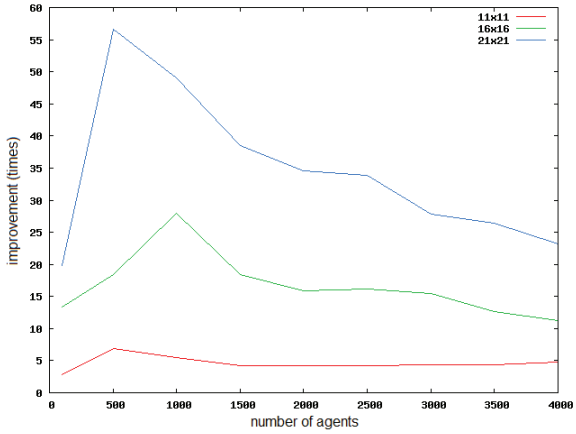


Figure 9: *Speed up achieved using the parallel implementation over the sequential version, with three different sizes of local maps.*

The graphic in Figure 9 shows the speed up achieved using the parallel implementation over the sequential version of the technique. As we can see, in all tests executed the parallel version was above twice faster than the sequential one (exactly the lowest point in the graphic is at 2.85 times). Besides that, the highest point in the graphic occurs at the point 56.60, meaning that in an optimal configuration the parallel version was 56 times faster than the sequential version.

Using bigger local maps means that more threads are needed for each local map in the several steps of the technique. The fact that the multiprocessor offers several running threads at the same time implies in a better use of the resources and in good improvements in performance.

On the other hand, for several reasons, with smaller local maps the speed up is not so high. On the side of the parallel version, a small local map does not make a good use of the resources of each multiprocessor. And on the side of the sequential version, a small local map may fit better in the processor cache. Moreover, the processor clock is three times higher than the graphics processor clock. If we combine all these factors in the same test, the speed up in the parallel version is minimized.

In addition, according to the nVIDIA Cuda Programming Guide [NVIDIA. 2009], the graphics processor cannot handle all the data in a parallel way. The division of the work in blocks of threads lets the graphics processor scheduler run some blocks of thread while others wait for execution. Because of this, the computation of 256 local maps in a parallel way does not give a speed up of 256 times.

To explain what is the cause of the graphics peaks, the nVIDIA Cuda Programming Guide says that each algorithm implemented

with Cuda has an optimal point, in which the amount of blocks and threads uses the most possible number of resources available in the graphics processor simultaneously. In our technique, this point is the one with 500 agents in the scene, each one with a local map of a size of 21×21.

7 Conclusion

This paper presented a strategy to implement on GPU a BVP Planner [Dapper et al. 2007] that produces natural steering behaviors for virtual humans, using a path-planning algorithm based on the numerical solution of boundary value problems.

The guiding potential of Equation 1 is free of local minima, what constitutes a great advantage when compared to the traditional potential fields method. Furthermore, the method proposed is formally complete [Connolly and Grupen 1993] and generates smooth and safe paths that can be directly used in mobile robots or autonomous characters in games. The local information gathered by agent sensors allows treating dynamic obstacles, such as other agents navigating in the environment.

We implemented a parallel version of this algorithm using the nVIDIA® Cuda [NVIDIA. 2009] language, which allows us to use the graphics processor avoiding the use of shading languages. The parallelism was explored, reducing the amount of memory transactions between CPU and GPU.

Our result shown that the GPU implementation improves up to 56 times the sequential CPU version, allowing the real-time use of this technique even in scenarios with a huge number of autonomous characters, which is a common situation often found in games.

As future work, we suggest the exploration of ADI Method [Peaceman D. W. 1995], obtaining a faster convergence of the relaxation process. The ADI Method is suitable to be used on parallel architectures and to explore the use of other shading languages. It would be interesting to compare the possible improvements in performance using other languages.

We have also proposed an extension of this technique to manage the movement of groups of agents in dynamic environments [Silveira et al. 2008]. We intend to implement a parallel version of this extension and release the project over an open source license.

Acknowledgements

The authors would like to thank Edson Prestes and Fabio Dapper for their valuable work on the CPU version of the path-planning algorithm, and Francele Carvalho Rodrigues for helping in the orthographic and grammatical revision of the text. This work was partially supported by grants from CNPq to Leonardo Fischer, Renato Silveira and Luciana Nedel.

References

BLEIWEISS, A. 2008. Gpu accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 65–74.

BURGESS, R. G., AND DARKEN, C. J. 2004. Realistic human path planning using fluid simulation. In *Proceedings of Behavior Representation in Modeling and Simulation (BRIMS)*.

CHOI, M. G., LEE, J., AND SHIN, S. Y. 2003. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph. 22*, 2, 182–203.

CONNOLLY, C., AND GRUPEN, R. 1993. On the applications of harmonic functions to robotics. *International Journal of Robotic Systems 10*, 931–946.

DAPPER, F., PRESTES, E., IDIART, M. A. P., AND NEDEL, L. P. 2006. Simulating pedestrian behavior with potential fields. In *Advances in Computer Graphics*, Springer Verlag, vol. 4035 of *Lecture Notes in Computer Science*, 324–335.

DAPPER, F., PRESTES, E., AND NEDEL, L. P. 2007. Generating steering behaviors for virtual humanoids using bvp control. *Proc. of CGI*.

FUNGE, J. D. 2004. *Artificial Intelligence For Computer Games: An Introduction*. A. K. Peters, Ltd., Natick, MA, USA.

JAMES J. KUFFNER, J. 1998. Goal-directed navigation for animated characters using real-time path planning and control. In *International Workshop on Modelling and Motion Capture Techniques for Virtual Environments*, Springer-Verlag, London, UK, 171–186.

KAPADIA, M., SINGH, S., HEWLETT, W., AND FALOUTSOS, P. 2009. Egocentric affordance fields in pedestrian steering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 215–223.

KAVRAKI, L., SVESTKA, P., LATOMBE, J.-C., AND OVERMARS, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration space. *IEEE Transactions on Robotics and Automation 12*, 4, 566–580.

KHATIB, O. 1980. *Commande dynamique dans l'espace opérational des robots manipulaters en présence d'obstacles*. PhD thesis, École Nationale Supérieure de l'Aéronatique et de l'Espace, France.

LAVALLE, S. 1998. Rapidly-exploring random trees: A new tool for path planning. Tech. Rep. 98-11, Computer Science Dept., Iowa State University.

METOYER, R. A., AND HODGINS, J. K. 2004. Reactive pedestrian path following from examples. *The Visual Computer 20*, 10, 635–649.

NIEUWENHUISEN, D., KAMPHUIS, A., AND OVERMARS, M. H. 2007. High quality navigation in computer games. *Sci. Comput. Program. 67*, 1, 91–104.

NVIDIA. 2009. Nvidia cuda. *http://www.nvidia.com/cuda, last acces at 07/2009*.

PEACEMAN D. W., R. J. H. H. 1995. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics 3*, 28–41.

PELECHANO, N., OBRIEN, K., SILVERMAN, B., AND BADLER, N. 2005. Crowd simulation incorporating agent psychological models, roles and communication. In *1st Int'l Workshop on Crowd Simulation*, 21–30.

PETTRE, J., SIMEON, T., AND LAUMOND, J. 2002. Planning human walk in virtual environments. In *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. 3, 3048 – 3053.

PRESTES, E., ENGEL, P. M., TREVISAN, M., AND IDIART, M. A. 2002. Exploration method using harmonic functions. *Robotics and Autonomous Systems 40*, 1, 25–42.

REYNOLDS, C. 2006. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, ACM Press, New York, NY, USA, 113–121.

SILVEIRA, R., PRESTES, E., AND NEDEL, L. P. 2008. Managing coherent groups. *Comput. Animat. Virtual Worlds 19*, 3-4, 295–305.

TECCHIA, F., LOSCOS, C., CONROY, R., AND CHRYSANTHOU, Y., 2001. Agent behaviour simulator (abs): A platform for urban behaviour development.

TREUILLE, A., COOPER, S., AND POPOVIĆ, Z. 2006. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM Press, New York, NY, USA, 1160–1168.

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