Real-Time Motion Planning, Navigation, and Behavior for Large Crowds of Virtual Humans

THÈSE N^{O} 4401 (2009)

PRÉSENTÉE LE 22 MAI 2009

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE RÉALITÉ VIRTUELLE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Barbara YERSIN

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. D. Thalmann, directeur de thèse
Prof. M. Bierlaire, rapporteur
Dr S. Donikian, rapporteur
Prof. M. Zwicker, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2009

CONTENTS

Remerciements

Je tiens à remercier mon directeur de thèse, le Pr. Daniel Thalmann, ainsi que le Fonds National Suisse de la Recherche Scientifique pour m'avoir offert l'opportunité d'effectuer ce doctorat. Un grand merci également à Stéphane Donikian, Matthias Zwicker, Michel Bierlaire et Claude Petitpierre pour avoir accepté de faire partie de mon jury de thèse.

Merci à toute l'équipe présente et passée du VRLAB pour son soutien et sa bonne humeur. Je tiens à remercier plus particulièrement Helena Grillon, Damien Maupu, Ronan Boulic, Daniel Raunhardt et Benoît Le Callennec pour leurs conseils, nos discussions animées, et leur amitié.

Un grand merci également à Mireille pour son travail de design toujours excellent et efficace, même dans les moments les plus stressants, ainsi qu'à Julien Pettré, sans qui cette thèse ne serait pas la même.

Merci à Josiane Bottarelli, notre maman à tous au VRLAB, pour son aide et sa rapidité, sa bonne humeur et sa cuisine !

Je tiens à remercier toute ma famille pour son soutien inconditionnel : Jean-Robert, Dominique, Cathy, Alexandre, Idette, Jean-David, mais aussi Dina, Yaël et Alexandre.

Finalement, merci de tout coeur à Jonathan, sans qui la force de continuer m'aurait fait défaut plus d'une fois. Merci de m'avoir soutenue, aidée, conseillé durant toutes ces années et d'avoir partagé avec moi cette grande étape de ma vie, dans les moments heureux comme dans les périodes de doute.

Résumé

La simulation de foules en temps réel est un problème stimulant qui touche à beaucoup d'aspects différents de l'Infographie : visualisation, animation, planification de chemins, comportement, etc. Notre travail a été principalement dédié à deux aspects particuliers des foules en temps réel : la planification de mouvements et le comportement.
La planification de mouvements en temps réel pour une foule requiert des méthodes à la fois efficaces et réalistes afin de trouver un chemin adéquat, ainsi que d'éviter les obstacles sur ce chemin. Trouver un compromis entre efficacité et crédibilité est particulièrement difficile, et les techniques précédemment étudiées se concentrent souvent sur une seule approche. Nous avons développé deux approches pour complètement résoudre en temps réel la planification de mouvements pour une foule.

Notre première approche est une architecture hybride capable de gérer en temps réel la planification de chemins pour des milliers de piétons, tout en assurant l'évitement dynamique de collisions. Cette architecture est échelonnée de façon à permettre la création et distribution de régions d'intérêt variable, où la planification de mouvement est gérée par différents algorithmes. Concrètement, les régions de grand intérêt sont gouvernées par une approche à long terme basée sur un flux de potentiel, tandis que les autres zones exploitent un graphe de l'environnement et des techniques d'évitement à court-terme. Notre architecture permet aussi d'assurer la continuité de mouvements lors d'un changement entre deux algorithmes. Les tests et comparaisons effectués montrent que notre architecture est capable de planifier de manière réaliste et en temps réel le mouvement de milliers de personnages dans des environnements variés. Notre seconde approche est basée sur le concept de "zones de mouvement" (motion patches en anglais) [Lee et al., 2006], que nous étendons pour peupler de manière dense de grands environnements. Nous construisons une population à partir d'un ensemble de blocs contenant une simulation de foule pré-calculée. Chaque bloc est appelé une "zone de foule" (crowd patch en anglais). Nous résolvons les problèmes liés au calcul d'une zone, leur assemblage pour créer des environnements virtuels, et le contrôle de leur contenu afin de répondre aux besoins des designers. Notre contribution majeure est de procurer une diminution drastique des coûts de calcul pour simuler une foule d'humains virtuels en temps réel. Nous pouvons ainsi gérer des populations denses dans des environnements de grande échelle, et ceci avec des performances jamais atteintes auparavant. Nos résultats illustrent la population en temps-réel d'une ville potentiellement infinie avec des foules variées d'humains interagissant les uns avec les autres et avec leur environnement.
Appliquer des comportements intelligents et autonomes aux foules est un problème difficile, car la plupart des algorithmes sont trop coûteux en temps de calcul pour être exploités sur de grandes foules. Notre travail a été concentré sur la recherche de solutions pour simuler des comportements intelligents dans une foule, tout en restant peu coûteux. Nous contribuons dans ce domaine en développant des comportements basés sur la position des humains virtuels : des comportements sont déclenchés selon la situation et la position des piétons. Nous avons aussi étendu notre architecture de planification de mouvement avec un algorithme capable de simuler le comportement de groupes, ce qui améliore la perception de la scène par les utilisateurs.

Mots-clés :

Foules, temps-réel, planification de mouvements, comportement, évitement de collision, navigation

Abstract

Simulating crowds in real time is a challenging problem that touches many different aspects of Computer Graphics: rendering, animation, path planning, behavior, etc. Our work has mainly focused on two particular aspects of real-time crowds: motion planning and behavior.

Real-time crowd motion planning requires fast, realistic methods for path planning as well as obstacle avoidance. The difficulty to find a satisfying trade-off between efficiency and believability is particularly challenging, and prior techniques tend to focus on a single approach. We have developed two approaches to completely solve crowd motion planning in real time.

The first one is a hybrid architecture able to handle the path planning of thousands of pedestrians in real time, while ensuring dynamic collision avoidance. The scalability of this architecture allows to interactively create and distribute regions of varied interest, where motion planning is ruled by different algorithms. Practically, regions of high interest are governed by a long-term potential field-based approach, while other zones exploit a graph of the environment and short-term avoidance techniques. Our architecture also ensures pedestrian motion continuity when switching between motion planning algorithms. Tests and comparisons show that our architecture is able to realistically plan motion for thousands of characters in real time, and in varied environments.

Our second approach is based on the concept of motion patches [Lee et al., 2006], that we extend to densely populate large environments. We build a population from a set of blocks containing a pre-computed local crowd simulation. Each block is called a *crowd patch*. We address the problem of computing patches, assembling them to create virtual environments (VEs), and controlling their content to answer designers' needs. Our major contribution is to provide a drastic lowering of computation needs for simulating a virtual crowd at runtime. We can thus handle dense populations in large-scale environments with performances never reached so far. Our results illustrate the real-time population of a potentially infinite city with realistic and varied crowds interacting with each other and their environment.

Enforcing intelligent autonomous behaviors in crowds is a difficult problem, for most algorithms are too computationally expensive to be exploited on large crowds. Our work has been focused on finding solutions that can simulate intelligent behaviors of characters, while remaining computationally inexpensive. We contribute to crowd behaviors by developing situation-based behaviors, *i.e.*, behaviors triggered depending on the position of a pedestrian. We have also extended our crowd motion planning architecture with an algorithm able to simulate group behaviors, which much enhances the user perception of the watched scene.

Keywords:
Crowds, real-time, motion planning, behavior, collision avoidance, navigation

Relevant Publications

1. **Crowd Patches: Populating Large-Scale Virtual Environments for Real-Time Applications:** Barbara Yersin, Jonathan Maïm, Julien Pettré, and Daniel Thalmann; I3D'09.

2. **Real-Time Crowd Motion Planning: Scalable Avoidance and Group Behaviors:** Barbara Yersin, Jonathan Maïm, Fiorenzo Morini, and Daniel Thalmann; The Visual Computer 24(10), 2008.

3. **Real-Time, Scalable Motion Planning for Crowds:** Fiorenzo Morini, Barbara Yersin, Jonathan Maïm, and Daniel Thalmann; Cyberworlds International Conference, 2007.

4. **Populating Ancient Pompeii with Crowds of Virtual Romans:** Jonathan Maïm, Simon Haegler, Barbara Yersin, Pascal Müller, Daniel Thalmann, and Luc Van Gool; In the 8th International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST), 2007.

5. **Real-Time Navigating Crowds: Scalable Simulation and Rendering:** Julien Pettré, Pablo De Heras Ciechomski, Jonathan Maïm, Barbara Yersin, Jean-Paul Laumond, and Daniel Thalmann; Computer Animation and Virtual Worlds (CAVW) Journal - Special issue CASA 2006.

6. **Steering a Virtual Crowd Based on A Semantically Augmented Navigation Graph:** Barbara Yersin, Jonathan Maïm, Pablo De Heras Ciechomski, Sebastien Schertenleib, and Daniel Thalmann; VCROWDS'05.

CHAPTER 1

Introduction

Figure 1.1: A virtual crowd simulated in an urban environment by our crowd engine.

The possibility to simulate large crowds attracts the interest of many different domains. In Architecture, a crowd simulation allows to virtually populate 3D construction models

before they are built and inhabited in the real world; in the movies industry, virtual crowds are used to simulate epic battles; in the games industry, crowds are naturally exploited to populate entire virtual worlds; in History, reviving an ancient population and simulating its habits has a wide interest; in the medical domain, immersing patients in a virtual crowd allows psychiatrists to expose and train them against their fears; in the security domain, being able to simulate evacuations in confined environments is paramount too. Virtual crowds can offer a valuable contribution to all these domains. For this reason, the Computer Graphics community has been working for several years on this topic, and tremendous progress has been done.

Figure 1.2: A battle simulated with Massive^{TM} in the movie: "The Chronicles of Narnia: Prince Caspian".

Nevertheless, simulating crowds remains a particularly difficult topic, because it is situated at the intersection of many different research domains: rendering, animation, navigation, behavior, etc. If one wants to perfectly mimic a real crowd, all these subjects need to be treated with extreme care, and for real-time solutions, painful trade-offs need to be made. Software systems working offline, such as Massive^{TM} have provided impressive results over the last years, including the epic battles simulated for the movie "The Chronicles of Narnia: Prince Caspian" (Figure 1.2). Real-time solutions use more and more GPU-based techniques to lighten the burden of the CPU, as illustrated in the latest demo by ATI (Figure 1.3).

Our main interest lies in the motion planning of crowds. This subject has become a fundamental research field in the Computer Graphics community over the last years. The simulation of urban scenes, epic battles, or other environments that show thousands of people in real time require fast and realistic crowd motion. In this Chapter, we first introduce motion planning for crowds and its main challenges in Section 1.1. Then, in Sections 1.2 and 1.3, we present our main motivations and the contributions we have made in this domain. Finally, in Section 1.4, we summarize the organization of the following Chapters.

Figure 1.3: Froblins are mostly computed by the GPU of the latest graphics card by ATI.

1.1 Real-Time Crowd Motion Planning

In our everyday life, it sometimes happens that we find ourselves sitting on a bench in the middle of a busy pedestrian place, observing people and their attitudes. Most of them are walking. They have a goal that they want to reach, while avoiding bumping into other people, or tripping on an obstacle. Some of them are in groups, talking together as they walk, others are alone, but talking on the phone, some are just trying to avoid spots where the crowd is too dense and find a shorter path to reach their goal as quickly as possible, others don't hesitate to run, because they are already late. In order to correctly simulate such a situation (like the one illustrated in Figure 1.1) in real time, it is important to plan the movement of each individual: what is the goal? how to get there? are there people to stay close to? Are there people to avoid? All these questions regarding every single individual in the crowd is managed by what we call *motion planning*.

We distinguish three aspects of motion planning that need to be addressed if we want to obtain realistic results:

- *Path planning:* how to get from point A to point B. Given a goal, we usually plan our path according to various criteria: avoid zones where the traffic is too dense, reduce the distance to cover, minimize travel time, etc.

- *Obstacle avoidance:* to safely reach a goal, one also needs to avoid static and dynamic obstacles in the environment. Static obstacles usually are objects that do not move, such as trash cans, streetlights, signboards, etc. Dynamics obstacles typically are all other moving entities, including cars, animals, and especially, other pedestrians who

are constantly moving in unpredictable ways.

- *Group cohesion:* if a pedestrian moves in a group with his friends or family, he tries to remain as close to them as possible, while progressing on his path and avoiding obstacles.

Many approaches have been introduced to solve these problems. The main challenge is to find a method that provides realistic results, while keeping high frame rates.

1.2 Motivations

There are three main motivations to our work.

Firstly, most motion planning techniques today offer solutions too expensive to simulate very large crowds. To keep high frame rates, such methods either require to lower the number of characters, or to reduce their rendering to 2D points. We want to integrate a motion planning solution in a complete crowd engine, so that the whole simulation, *i.e.*, motion planning, navigation, animation, and rendering of thousands of highly detailed virtual humans runs in real time.

Secondly, when populating a virtual environment with crowds, motion planning is not a sufficient component to make them behave realistically. Indeed, if pedestrians are simply walking alone, with no motivational goal, and do not react to events happening in their environment, they have more similarities with zombies than with real crowds. A user observing such a scene quickly loses interest because of the lack of realism. We want to provide a solution to have intelligent behaviors in the crowd.

Finally, there are limitations to the number of people that can be simulated in real time. These restrictions come from the limited computational resources. On the other hand, virtual worlds constantly become larger and larger. Simulations extend to wider and wider environments, that could potentially reach the entire Earth, *e.g.*, Google Earth^{TM}, see Figure 1.4. Our goal is to be able to simulate crowds of sizes never reached so far, so that populating such environments becomes feasible.

1.3 Contributions

Our contribution is threefold.

Firstly, in order to handle realistic crowd motion planning in real time, we have developed a hybrid architecture that is plugged to an existing crowd engine. Our approach is based on levels of detail, usually employed for crowd rendering. We here use this approach to hierarchically plan the motion of pedestrians: we divide a scene into multiple regions that we classify with different levels of interest. Zones of high interest exploit accurate motion

Figure 1.4: A satellite view of New York, with Google Earth^{TM}.

planning techniques, while regions of lower interest for the user are ruled by faster, coarser methods. Additionally, a short-term collision avoidance algorithm is provided to avoid last-minute collisions and to ensure that switching between two regions ruled by different algorithms is seamless.

Our solution is complete and can handle the three aspects of motion planning introduced above, *i.e.*, path planning, obstacle avoidance, and group cohesion. By using a level of detail approach, it is possible to obtain high performances, even when integrating our algorithm with a complete crowd engine: we can simulate thousands of highly detailed characters that are animated and moving in real time. Moreover, regions with different levels of interest are directly set according to the user directives. He can thus choose the performance he wants to achieve, and distribute computation time accordingly.

Our second contribution concerns crowd behavior. In order to simulate intelligent behaviors at low cost, we propose to provide virtual humans with a knowledge of their environment by exploiting a *Navigation Graph*. Based on this knowledge, we induce special behaviors in the crowd, depending on their location in the scene. We have created several types of

behaviors:

- Make pedestrians acquire objects when entering special places, *e.g.*, people enter a shopping mall empty-handed, but get out with shopping bags.

- Simulate group behaviors, *i.e.* making some pedestrians walk close to each other at all times. This has been implemented as an additional layer to our motion planning technique, and can be enabled at initialization if desired.

- Trigger look-at behaviors, *i.e.*, make a pedestrians look at specific points in the environment when passing near them, *e.g.*, open doors or windows.

Based on our graph structure, many different situation-based behaviors can be triggered, thus making the crowd act in a more intelligent way than simply walking.

Our last contribution has been oriented towards the idea of populating very large-scale scenes. In such environments, simulating interactive crowds in real time becomes too expensive. On the other hand, completely pre-computing paths for every character is too memory demanding. We offer a solution that pre-computes the simulation task for a small patch of environment. By interconnecting such patches together, like dominos, we can simulate very large crowds at low cost: indeed, collision avoidance is pre-computed in the patches, and thus, the remaining work to achieve at run-time is much reduced. The simulation can last as long as wished, for patches are implemented to be periodic: the computed trajectories can be endlessly and seamlessly replayed over time. Using these patches, we can populate vast environments, of a potentially infinite size: patches are only set where the camera is looking. Their combination is very fast and can be achieved at runtime, depending on the camera position. Finally, since trajectories are pre-computed, local collision avoidance can be achieved with highly accurate, time consuming methods to provide excellent results at run-time.

1.4 Summary of Chapters

The next Chapters are distributed as follows.

Chapter 2. In this Chapter, we present the previous work achieved in crowd simulations. We first provide a general overview in the domain of crowd rendering. Then, we thoroughly detail previous work related to crowd motion planning and behavior.

Chapter 3. To fully understand our contributions in this research, we detail the background that represents its supporting foundations: the Navigation Graph, a structure that is used to help motion planning, and a crowd engine that takes care of animating and rendering characters. Our work on motion planning is directly integrated in this crowd engine.

Chapter 4. In this Chapter, we present our hybrid architecture used to rule motion planning of crowds with a level-of-detail approach. Pedestrians in regions of high interest are ruled by a potential field-based technique, while people at farther distances use less accurate methods.

Chapter 5. To complete our crowd motion planner, we have implemented several techniques allowing to avoid inter-pedestrian collisions in the short-term. This Chapter presents these methods, their functioning, and their implementation.

Chapter 6. An additional layer to our crowd engine has been developed to enhance the behavior of crowds: this layer includes situation-based actions, triggered depending on the position of pedestrians, but also grouping behaviors, *i.e.*, people walking in groups of 2 to 4, which are often encountered in urban scenes.

Chapter 7. In a final contribution, we have taken a new approach to crowd motion planning in order to minimize the computational costs at runtime. This approach prevents the user from having direct interactions with the crowd. However, it allows to populate very large environments for an unlimited amount of time, at a low cost.

Chapter 8. This Chapter brings a conclusion to our work. We here summarize our contributions, and discuss future work.

CHAPTER 2

Related Work

Populating virtual environments with large crowds of virtual humans is a challenging task. It encompasses many different aspects: rendering the environment, rendering the virtual population, controlling and designing content, simulating virtual humans' behaviors, their navigation and actions, and animating them accordingly. In the case of real-time simulations, the solutions to these problems are constrained by the need for interactivity: online real-time techniques are required.

In this chapter, we present the previous work that has been achieved in the domain of real-time crowd simulations. First, we introduce the major previous work on crowd rendering (Section 2.1). Then, in Section 2.2, we present in detail the related work in the domain of crowd motion planning, navigation, and collision avoidance. Finally, we explain how environment and navigation structures have been semantically annotated to rule crowd behavior in Section 2.3.

2.1 Crowd Rendering

Rendering a digital actor consists in several important steps: updating the animation, deforming the mesh accordingly, texturing the mesh, shading it, etc. The 2 main challenges when rendering crowds of such virtual humans in real-time are:

1. Keep a high frame rate wherever the camera is situated and whichever the number of visible humans.

2. Make each individual composing the crowd unique in appearance.

Rendering thousands of virtual humans at high frame rates is today possible if a level-of-detail approach is used. Indeed, a brute force execution of these tasks using a dynamic mesh [Lander, 1998], even with specialized hardware, does not scale to large crowds. A first optimization is to decrease the geometric model complexity according to the distance to the camera, which is the key idea of a level-of-detail approach [Hoppe, 1996]. In the case where such models are animated using a dynamics-based technique, simulation should be scaled [Safonova et al., 2004].

A second optimization to reduce the cost of animation updates as well as mesh deformations, is to employ static or baked meshes [Ulicny et al., 2004; Coic et al., 2005]. We call "static meshes" meshes whose vertices are deformed in a pre-process with respect to a set of animations. These postures are then stored in memory to be reused online during the simulation. Another representation of static meshes is through surfels [Pfister et al., 2000; Wand and Straßer, 2002]. Static meshes have the advantage of avoiding to perform the costly mesh deformation online. However, they have a large memory requirement, and thus, only a limited set of pre-computed animations can be used.

A third level of detail that is widely used in the domain of crowd rendering is image-based. Indeed, image-based rendering bypasses the need for animation updates and mesh deformations, as it works directly on the final projected image on the screen. Billboarding is the de facto standard for image-based rendering of crowds [Tecchia et al., 2002a] (see left of Figure 2.1). Billboards can also be dynamically updated: parts of the body define a billboard hierarchy [Aubel et al., 2000]. Dobbyn et al. managed to efficiently combine billboards and geometrical meshes to render an impressive number of virtual humans [Dobbyn et al., 2005], as illustrated in Figure 2.1 (right). Unfortunately, the use of billboards for rendering humans is highly restrictive as the range of possible motions is limited to the image content; generally, behaviors are limited to navigation and humans perform locomotion only. This partly explains why interactive environments made of humans exhibiting rich and varied behaviors [Shao and Terzopoulos, 2005] have a limited population size.

Figure 2.1: *(Left)* The use of billboards allows Tecchia et al. to simulate large crowds [Tecchia et al., 2002a]. *(Right)* Dobbyn et al. combine static meshes and billboards to have a higher quality rendering in front of the camera [Dobbyn et al., 2005].

The approach developed in the Yaq crowd engine [Maïm, 2009] combines dynamic meshes, static meshes, and billboards to render very large crowds of virtual humans [Maïm

et al., 2009]. This approach allows to have highly detailed meshes at the forefront, dynamically animated and thus able to perform procedural facial animation and looking at each other or at the camera.

To make each individual in a crowd look unique, an ideal solution would be to design a different mesh with a different texture for every single character. It seems obvious that such an approach becomes quickly infeasible with an increasing number of people to simulate. One would require an army of designers to perform such a task, and an inexhaustible memory to store and use all these meshes and textures at run-time.

The first step to ensure the rendering variety of a crowd is to have a small set of human meshes, and several textures for each of them (within the limits of the designer/memory capacities). In our case for instance, with the Yaq crowd engine [Maïm, 2009], we were typically able to use up to 6 or 8 different humans, each compliant with 2 to 5 different textures. From such a set of templates, additional techniques to increase variety have been developed. Tecchia et al. [Tecchia et al., 2002b] proposed to assign to each texture different colors for different body parts in a multipass process. De Heras et al. showed that color variety can be achieved with shaders, using the alpha channel as a tool for differentiating the body parts [de Heras Ciechomski et al., 2005]. In terms of performance, they were able to deal with hundreds of virtual humans in real time. To improve on previous techniques, Maim et al. used segmentation maps [Maïm et al., 2009]. They also introduced accessories, *e.g.*, hats, backpacks, glasses, wigs, attached to the human meshes to further vary each profile. In Figure 2.2, we show the results obtained when differentiating body parts using the alpha channel (top) or segmentation maps (bottom).

2.2 Crowd Motion Planning

Motion planning is a very important aspect of real-time crowd simulations. Virtual humans walking in the environment need specific goals, and the means to reach them while avoiding any collision with the environment or the other pedestrians. In this Section, we detail the previous work achieved in this domain. We first introduce previous work on local navigation methods, that control the pedestrian's moves on a short-term basis to avoid each other (Section 2.2.1). Then, we climb one level higher and consider techniques used for global motion planning and navigation, *i.e.*, in the long term, in Section 2.2.2.

2.2.1 Individual Local Navigation and Steering

At the local level, solving interactions between pedestrians and making them avoid any collision are paramount, because they have a great impact on the motion realism.

A first solution to solve interaction between pedestrians is to simulate each individual's perception of the environment: behavioral systems range from agent models based on physical laws [Helbing et al., 2000], particle systems [Bouvier and Guilloteau, 1996], perception-action loops [Terzopoulos, 1999], or sociology knowledge [Musse and Thalmann, 1997;

Figure 2.2: Recent results in crowd variety using different techniques to determine body parts: *(top)* using the alpha channel of the human texture [de Heras Ciechomski et al., 2005], and *(bottom)* using dedicated segementation maps [Maïm et al., 2009].

Jager et al., 2001]. Such models can be hierarchical in order to satisfy performance requirements: Musse and Thalmann simulated crowds with a 3-level hierarchy: agents, groups, and crowds [Musse and Thalmann, 2001]. In Figure 2.3, we show how they steer the crowd as groups.

The main problem when using an agent-based approach is that models cannot simulate large crowds (several thousands of people), unless the rendering task is much simplified, *e.g.*, using 2D points instead of human meshes.

Moving several entities simultaneously using waypoints can be achieved in a very efficient way, using steering or flocking methods [Reynolds, 1987; Hodgins et al., 1995; Reynolds, 1999] (see Figure 2.4). A sequence of short-term waypoints leading to a prescribed goal can be computed automatically at run-time.

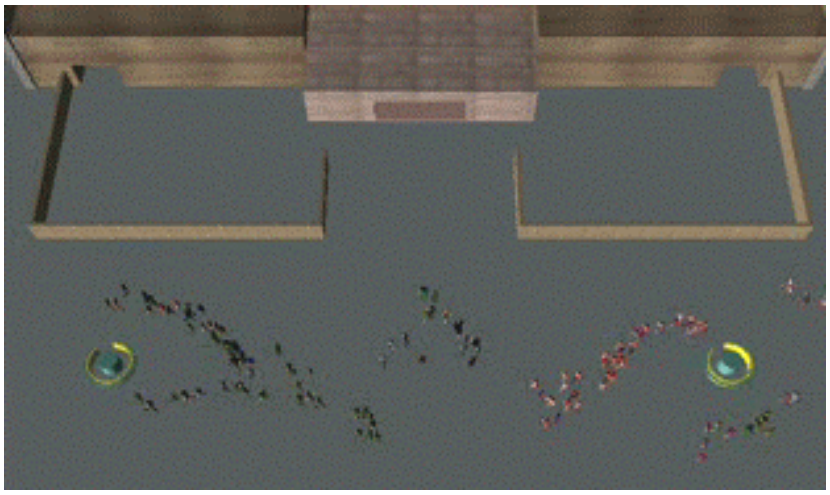


Figure 2.3: A hierarchical behavioral model for crowds [Musse and Thalmann, 2001]. Crowd behaviors are distributed to the groups, and then to the individuals.

Figure 2.4: Hodgins et al. used a grouping algorithm extended from the work of Reynolds [Reynolds, 1987] to make bicyclists move as a group while avoiding obstacles [Hodgins et al., 1995].

Another approach is to use local reactive methods. With a cell decomposition of the environment, it is possible to represent a pedestrian as a particle in a cell. Each cell can be occupied by at most one pedestrian [Gipps and Marksjo, 1985; Klüpfel et al., 2000]. Loscos et al. smoothed the trajectories by allowing several pedestrians in a same cell, but in a discreet number of positions [Loscos et al., 2003].

Khatib tested the use of a force field on mobile robots and manipulators [Khatib, 1986]: the goal to reach is represented as an attractive force, while obstacles are repulsive forces. Helbing and colleagues presented a model based on social forces, or the internal motivation of each pedestrian to perform specific actions [Helbing and Molnár, 1995]. This model has been extended to overcome some artifacts, such as oscillations in trajectories [Helbing et al., 2005; Pelechano et al., 2005]. In a recent work, Van den Berg et al. extended the Velocity Obstacle concept from robotics to solve human interactions [van den Berg et al., 2008] (see Figure 2.5).

2.2.2 Spatial Subdivision and Global Path Planning

At a global level, the autonomous navigation of crowds can be considered as a path planning problem, *i.e.*, finding the best path for an individual to reach a goal. This problem has been widely explored by the robotics community [Latombe, 1991]. Different techniques that sub-

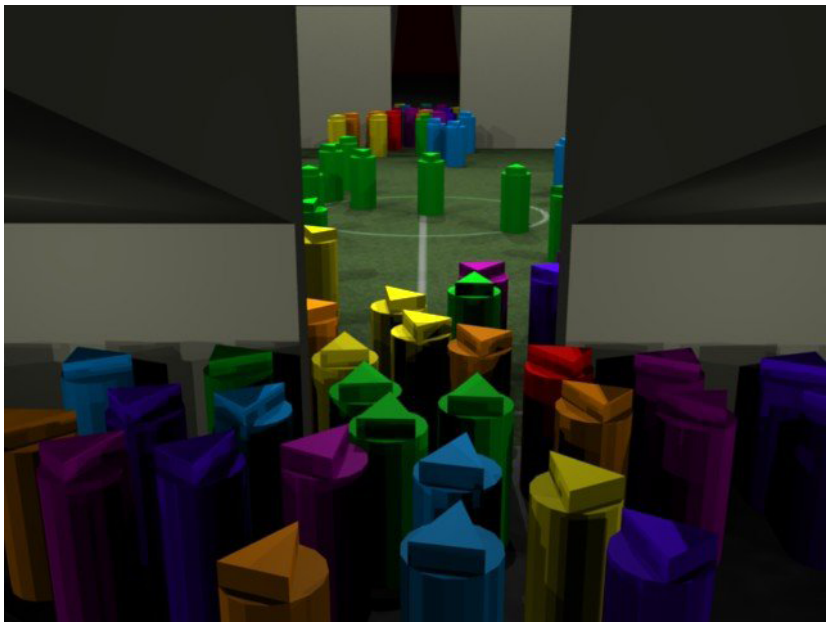


Figure 2.5: In their recent work, Van den Berg et al. simulate agents in crowded environments [van den Berg et al., 2008]. Each agent senses the environment independently and computes a collision-free path based on extended Velocity Obstacles and smoothness constraints.

divide the environment in various ways have been proposed.

The most intuitive way is to use a uniform cell decomposition in 3D [Bandi and Thalmann, 1998], or with 2D bitmaps, taking advantage of graphics hardware [Kuffner, 1998]. With such a decomposition, a pedestrian's global motion usually results from successive local motions from cell to cell, guided by probabilistic rules [Kirchner and Shadschneider, 2001].

Loscos et al. [Loscos et al., 2003] presented a behavioral model based on a 2D map of the environment: after identifying buildings, sidewalks, and crosswalks in a pre-process, characters followed successive goals based on probabilistic rules. Their method is suited when the spectator does not focus on one pedestrian in particular. Figure 2.6 illustrates the results obtained with their approach.

It is also possible to analyze navigable parts of environments and plan the best paths using probabilistic roadmaps (PRM): the nodes of the roadmap represent collision-free key positions, while edges are feasible paths between the nodes. Kavraki et al. applied this approach to find collision-free paths for robots in static workspaces [Kavraki et al., 1996], while Arikan et al. computed a visibility graph for simulating virtual agents [Arikan et al., 2001]. The PRM-based approach was later adapted to plan individual path and the underlying motions of characters [Choi et al., 2003; Pettré et al., 2003; Hauser et al., 2005].

Improvements on the PRM-based approach have been proposed: to better handle navigation, Bayazit *et al.* [Bayazit et al., 2002, 2003] integrated flocking techniques with roadmap-based path planning. The flock members store global information, such as dead-ends, in

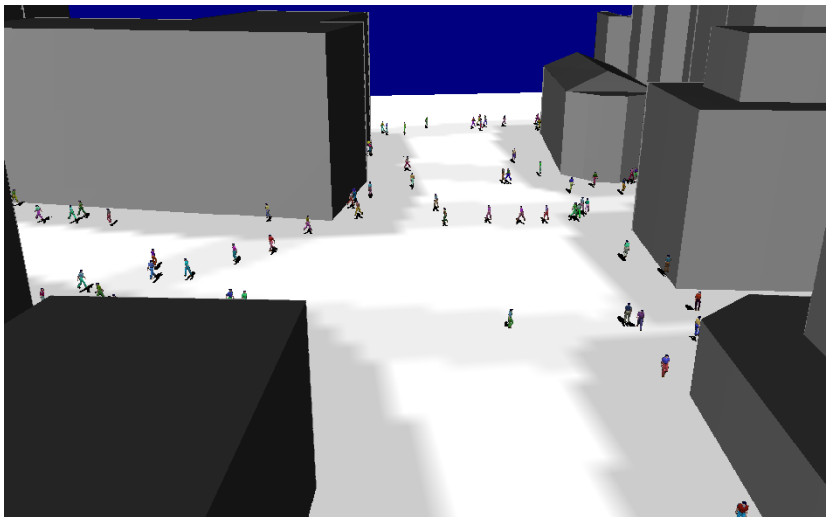
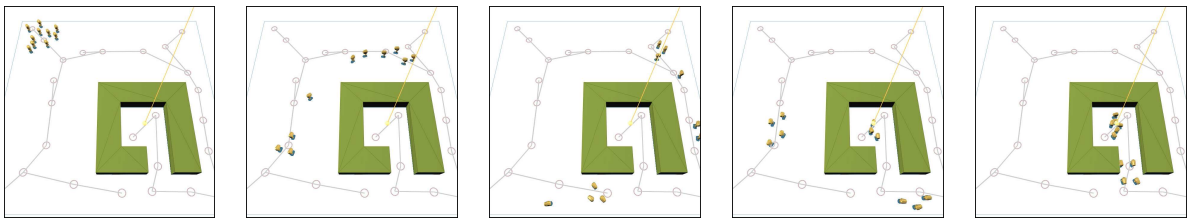


Figure 2.6: Example of simulation obtained based on a 2D map and probabilistic rules [Loscos et al., 2003] .

the roadmap nodes in order to warn other members (see Figure 2.7). Sung et al. combined probabilistic roadmaps with motion graphs to find paths and animations to steer characters towards a goal [Sung et al., 2005].

PRM-based approaches suit high-dimensional problems well. However, they are not the most efficient ones for navigation planning when the problem is reduced to three dimensions.



(a) (b) (c) (d) (e)

Figure 2.7: *"Ten flock members are searching for an unknown goal. (a) The flock faces a branch point. (b) Since both edges have the same weight, the flock splits into two groups. (c) After dead ends are encountered in the lower left and upper right, edge weights leading to them are decreased. (d) As some members find the goal, edge weights leading to it are increased. (e) The remaining members reach the goal."* [Bayazit et al., 2003].

Other structures to discretize environments have been explored. Lamarche and Donikian [Lamarche and Donikian, 2004] used triangulation, as illustrated in Figure 2.8. Given a start and a goal position in an environment, Kamphuis and Overmars defined a walkable corridor that ensured sufficient clearance to allow a given group of units to pass [Kamphuis and Overmars, 2004]. Pettré et al. [Pettré et al., 2006, 2007] presented a novel approach to automatically extract a topology from a scene geometry and handle path planning using a Navigation Graph. Their method to analyze the terrain is inspired from Voronoï diagrams [Fortune, 1997], and from methods using graphics hardware to compute them [Hoff et al.,

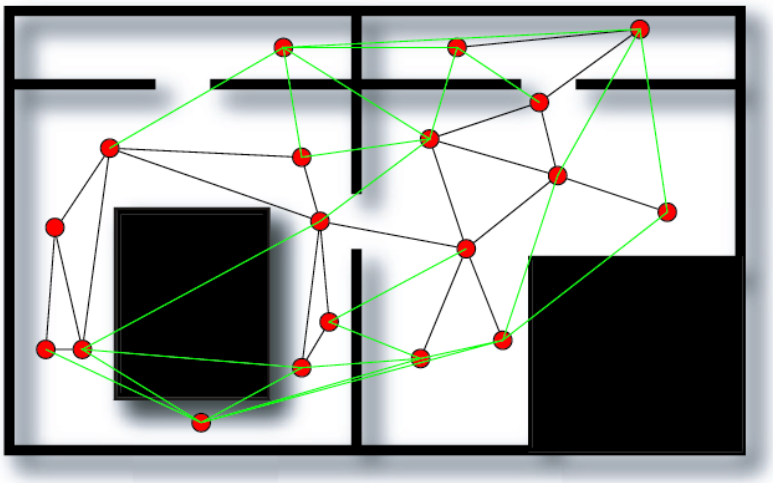


Figure 2.8: Example of neighborhood graph obtained by triangulation [Lamarche and Donikian, 2004].

1999]. Their structure is able to handle uneven and multi-layered terrains. As the Navigation Graph is the basis of our motion planning architecture, we fully detail it in the next Chapter.

In the case of large and complex environments, a hierarchical representation preserves fast path search times [Shao and Terzopoulos, 2005; Paris et al., 2006]. Sud et al. prefer using first and second order Voronoï diagrams to solve path planning queries and agent-to-agent neighboring queries [Sud et al., 2007]. Both cell-decomposition and Voronoï diagram-based techniques have demonstrated their ability to interactively handle large crowds. Path planning allows the user to have control over the population by distributing destination points to individuals composing the crowd.

Helbing *et al.*, using their social force model, handled agent-based motion planning, mainly focusing on emergent crowd behaviors in particular scenarii, *e.g.*, trail formation [Helbing et al., 1994], or panic escape [Helbing et al., 2000]. Most advanced crowd simulation softwares, as for instance $Massive^{TM}$, $Legion^{TM}$ or $Simulex^{TM}$ implement such agent-based models, *i.e.*, each individual has its own goal. However, because of their low performance, they better fit offline accurate simulations, *e.g.*, for security or architecture applications. Using these methods to simulate thousands of pedestrians in real time requires the use of particular machines supporting heavy parallelizations [Reynolds, 2006].

Some approaches can solve both navigation and pedestrians inter-collision problems simultaneously. For instance, with prioritized motion planning techniques [Lau and Kuffner, 2005], navigation is planned successively for each entity which then becomes a moving obstacle for the following ones. The growing complexity of this approach limits the number of people composing the crowd. The obtained results are illustrated in Figure 2.9 for 100 characters.

Physics-based algorithms have also been explored to solve both path planning and dynamic collision avoidance. Hughes [Hughes, 2002, 2003] interprets crowds as density fields

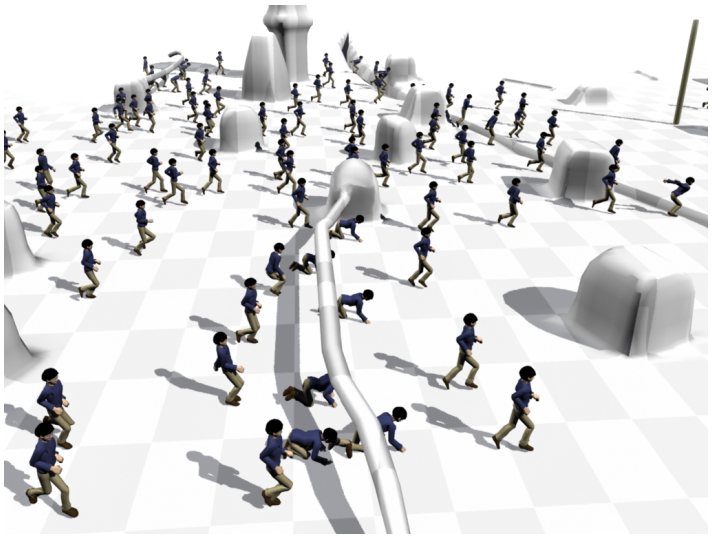


Figure 2.9: Resulting behavior of 100 characters using prioritized motion planning in a dynamic environment [Lau and Kuffner, 2005].

to rule the motion planning of pedestrians. The resulting potential fields are dynamic, guiding pedestrians to their objective, while avoiding obstacles. Chenney [Chenney, 2004] developed a model of flow tiles that ensures, under reasonable conditions, that agents do not require any form of collision detection at the expense of precluding any interaction between them. Heïgeas et al. [Heïgeas et al., 2003] introduced a model based on cellular automata and the physical properties of the environment to simulate emergent crowd phenomena. Kirchner and Shadschneider [Kirchner and Shadschneider, 2001] simulated the interaction between pedestrians evacuating a room. Their model discretizes the space into cells and two floor fields (one static, one dynamic) provide the pedestrians with probabilities of passing from one cell to another.

More recently, Treuille *et al.* [Treuille et al., 2006] proposed realistic motion planning for crowds, making an analogy with potential fields. Their method produces a potential field from the addition of a static field (goal) and a dynamic field (modeling other people). Each pedestrian then moves against the gradient towards the next suitable position in space (a *waypoint*) and thus avoids all obstacles.

Compared to agent-based approaches, these techniques allow to simulate thousands of pedestrians in real time, and are also able to show emergent behaviors. However, they produce less believable results, because they require assumptions that prevent treating each pedestrian with individual characteristics. For instance, only a limited number of goals can be defined and assigned to sets of pedestrians. The resulting performance depends on the size of the grid cells and the number of sets.

Hybrid approaches have also been explored. Pelechano *et al.* [Pelechano et al., 2007] combined psychological, physiological, and geometrical rules with physical forces to simulate dense crowds of autonomous agents. One of our main contribution is a hybrid architecture that adapts the concept of levels of detail from crowd rendering to crowd motion planning: different motion planning techniques are used to steer pedestrians, depending on

the user's point of view. We fully detail this architecture in Chapters 4 and 5.

A recent effort has been oriented toward solving interactions from examples, *i.e.*, from real captured data [Lerner et al., 2007; Paris et al., 2007; Lee et al., 2007]. However, performances do not fit to applications such as interactive virtual worlds.

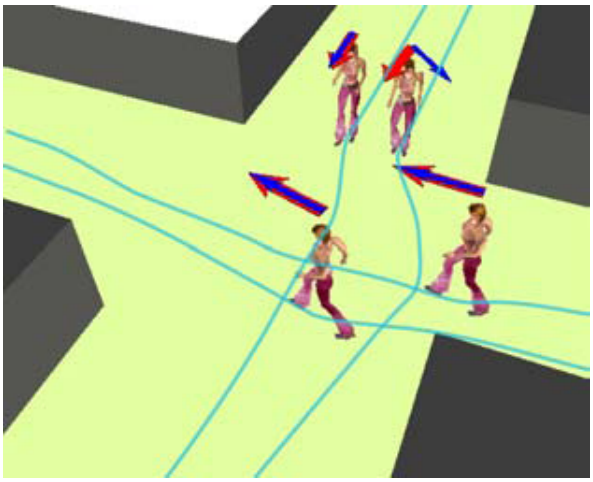


Figure 2.10: Pedestrians try to extrapolate the others' movements in order to prevent collisions [Paris et al., 2007]. The model is calibrated using motion capture data to obtain realistic results. The red arrows are desired directions, while blue arrows are the directions proposed by the model to prevent collisions.

In conclusion to this overview, different techniques - that are efficient enough to fit interactivity requirements - are available to provide motion and action autonomy to a virtual population: action decision results from behavioral simulation, or is defined directly at design. Global navigation is possible from path planning; individuals are then steered locally in order to avoid each-other. However, several limitations remain. First, it is difficult to mix various behaviors with real-time rendering techniques. Second, online generated worlds cannot be handled, because path planning is based on specific pre-computed data structures: environment size and complexity are limited, and a fixed, known in advance geometry is required. Finally, steering human locomotion is increasingly time-consuming with the number of interactions to solve.

2.3 Crowd Behavior

2.3.1 Environment Semantic and Spatial Behavior

In any location, and particularly in a city, virtual humans need to be aware of their environment in order for them to show an intelligent behavior, *i.e.*, act in accordance with their surroundings and react to sudden events that happen close by. Various methods have been studied on how to control crowd behavior by adding semantic data to their virtual environment: Farenc et al. presented a method for creating an informed environment, *i.e.*, an

environment able to provide pedestrians with information on their location, the position of various objects, the animations they should play, etc [Farenc et al., 1999]. Based on its geometry, an environment is hierarchically segmented into entities, such as quarters, blocks, parcels, or junctions. Musse et al. classified the environment information into 2 different types [Musse et al., 1998] :

- goals, that represent points through which the virtual humans should pass, or where the virtual humans should stop.
- obstacle positions, which indicate where obstacles are situated and allow the pedestrians to avoid collisions with the environment.

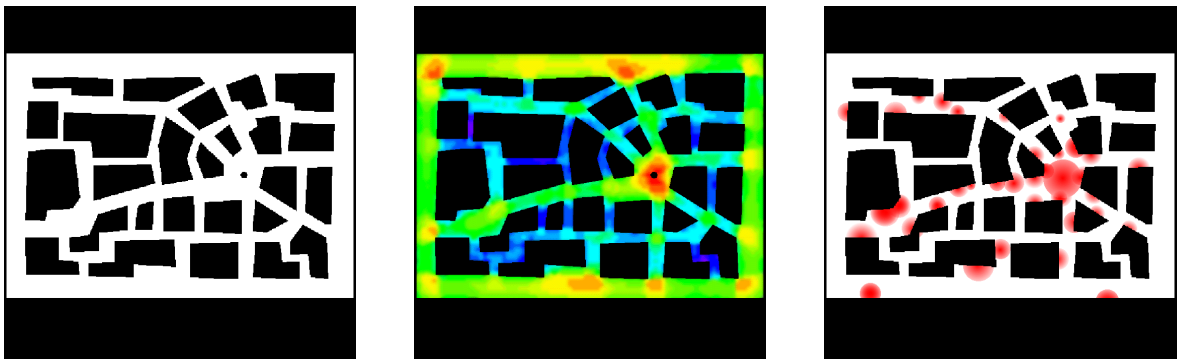


Figure 2.11: Collection of maps used to plan individual pedestrian actions [Tecchia et al., 2001]: *(left)* An example of a collision map. The regions where agents can move are encoded in white and inaccessible regions in black. *(center and right)* Examples of behaviour maps: Visibility map and Attraction map.

Tecchia et al. [Tecchia et al., 2001] developed a segmentation of the space into a 2D-grid, composed of 4 different layers. Each layer has a specific task: one takes care of inter-collision detection, another detects collisions with the environment, the third takes care of the behavior, and the last layer manages the callbacks for complex behaviors. The behavior layer, in particular, allows to associate a behavior to a cell. For instance, a cell can contain the information: ”turn left”, or ”wait”. We show in Figure 2.11 examples of such maps. Sung et al. [Sung et al., 2004] developed a painting interface that allows a designer to draw the regions where special events should occur. This interface is based on a layered structure, where each layer contains the regions of a particular situation, saved as a bitmap file, as illustrated in Figure 2.12. Based on these regions, a scalable behavioral model was used to have characters react according to a probabilistic action-selection mechanism.

Inspired from the work of Lamarche and Donikian [Lamarche and Donikian, 2004], Shao and Terzopoulos developed an autonomous pedestrian model where virtual humans individually plan their actions, based on a hierarchical collection of maps [Shao and Terzopoulos, 2005]. They create:

- Topological maps, which contain nodes and edges. The nodes are bounded volumes in 3D-space, and the edges represent the accessibility between these regions. A walkable 2D-surface is deduced from each volume.

- Perception maps, which provide the pedestrians with a sense of the ground height and of the surrounding static and mobile objects.

- Path maps, which take care of path planning for navigation.

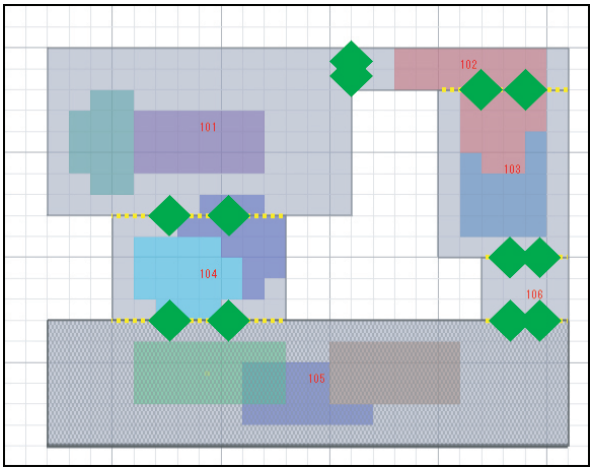


Figure 2.12: The painting interface [Sung et al., 2004]: Spatial situations can be easily set by drawing directly on the environment. Situation composition can be specified by overlaying regions.

More recently, Da Silveira and Musse [da Silveira and Musse, 2006] introduced a semi-automatic city generation process where semantic information such as population density was used to create the environment. However, they were limited to a fixed number of different buildings, previously stored in a repository. Moreover, although they exploited semantic data to choose where to put which kinds of buildings, this information was not further used to populate the generated environment or apply corresponding behaviors.

We have a different approach, based on the work of Pettré et al. [Pettré et al., 2006, 2007]: instead of storing the environmental information in the scene's geometry or in external structures, we directly employ the vertices of our Navigation Graph structure. The way we use them to add semantic to the environment and to steer the crowd behavior is explained in Chapter 6.

2.3.2 Groups

In real cities, many pedestrians are part of a group, whether they are sitting, standing, or walking toward their shared goal. They behave differently than if they were alone: they adapt their pace to the other members, wait for each other, may get separated in crowded places to avoid collisions, but regroup afterwards. Several approaches have been taken in order to simulate such behaviors. The first approach to offer impressive results is the one of Reynolds [Reynolds, 1987, 1999, 2006], who devised intelligent rules to simulate flocks of birds and fishes.

Bayazit et al. [Bayazit et al., 2003] created additional behaviors based on the work of Reynolds to further improve group behaviors. They introduced 3 group behaviors:

- The homing behavior is locally similar to the flocking behavior of Reynolds. Globally however, the group is steered toward its goal by a roadmap-based approach.
- The narrow passage behavior requires to designate a leader, who follows the assigned path. The other members follow him, arranged into a queue.
- The shepherding behavior nominates one group member as the shepherd dog. The dog's goal is to move the other group members (sheep) toward a goal, while the group members' only rule is to move away from the dog. If a subgroup gets separated from the flock, it is the dog's job to regroup them.

Musse and Thalmann directly defined crowds as a set of groups. Knowledge is hierarchically shared [Musse and Thalmann, 2001]: knowledge on positions of static obstacles and goals is shared by the whole crowd, while intelligence, memory, intention and perception are focalized in each group structure, stored in their unique leader. In order to remain close to each other, group members walk at similar speeds, follow the same path, and wait for each other on arrival at a goal. Finally, agents have a simpler structure than groups. They avoid collisions with each other and may decide to change group or become a leader. O'Sullivan et al. presented *ALOHA*, an architecture able to simulate group behaviors with a level-of-detail approach [O'Sullivan et al., 2003]. Their technique is focused on social interactions, *i.e.*, agents talking and listening to each other.

Niederberger and Gross introduced a generic system for autonomous and reactive agents, organized in hierarchical groups. Abstract groups are defined with special patterns, and instantiated with a given number of members. This process can be repeated to obtain a hierarchy of groups, *e.g.*, a family is composed of the father, the mother, and the children: the father is the leader, and the mother follows the father, but she is also the leader of the children [Niederberger and Gross, 2003].

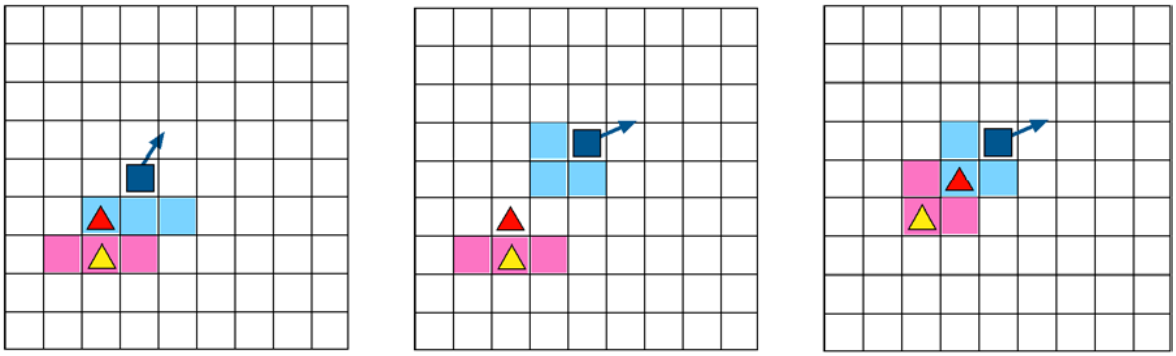


Figure 2.13: Group members (red and yellow triangles) following their leader (dark blue square) [Loscos et al., 2003]. *(Left)* The leader has tagged 3 cells behind it (light blue), and so has its first follower (light red). *(Center)* The leader has moved to a new cell and updated the tagged cells behind it to advise its followers. *(Right)* The two members have moved to follow their leader.

Loscos et al. introduced a simple model to simulate walking groups, based on a regular cell decomposition of the environment. Groups of 3 to 10 people are created and a leader is nominated for each group. At each time step, the leader moves first in a neighbor cell, and flags 3 cells behind it to advise its followers where they should move. The following members then try to reach the advised cells, and tag 3 cells behind them to show the way to the next members. This process is illustrated in Figure 2.13.To prevent the leader from constantly walking ahead of everybody else, the authors decided to not display it [Loscos et al., 2003].

If members of a group act individually (no leader makes the decisions), they may decide to reach a common goal using differents paths, resulting in a division of the group. Such a case is illustrated in Figure 2.14. To avoid this issue, Kamphuis and Overmars maintained a group cohesion by limiting the longitudinal and lateral dispersion of its members. The group's movement toward its goal and the inter-collision avoidance are managed using social forces [Kamphuis and Overmars, 2004].

More recently, Kwon et al. proposed a solution to edit the motion (animation and navigation) of a group while maintaining its formation. They create a graph whose vertices are positions of characters at precise frames. Edges define the neighborhood formation between individuals, and their trajectories. When a user edits this graph, care is taken to minimize the distortion of local arrangements among adjacent vertices [Kwon et al., 2008].

Figure 2.14: Typical problem encountered when members of a group look individually for a path in a complex environment. *(Left)* The group has for goal to attack the site indicated with a green arrow. *(Right)* Each member individually plans a path to reach the common goal, resulting in the division of the group [Kamphuis and Overmars, 2004].

CHAPTER 3

Background

The main work that has been achieved over the last 4 years has been focused on building a robust platform in order to simulate crowds in all aspects: display them in large numbers using levels of detail and appearance variety, control the pedestrians' behavior, animate them, and plan their motions while avoiding their collisions with the environment and each other. This work has been achieved in collaboration with Jonathan Maïm [Maïm, 2009]. While his contribution was targeted at the animation and rendering variety of crowds, our own main focus has been oriented towards crowd motion planning and behavior, which will be further detailed in the next Chapters.

In this Chapter, we present the background of our work. First, we introduce the *Navigation Graph* [Pettré et al., 2006, 2007], which is one of the main pillars of our crowd engine. It is used as a basic structure to select the levels of detail, plan the paths of pedestrians, and trigger situation-based behaviors. The construction of a Navigation Graph in itself, detailed in Section 3.1, has been developed by Julien Pettré. Our participation to this work has mainly been focused on integrating the graph structure into the crowd engine and to make it usable by our virtual crowds. The Navigation Graph has then been extensively exploited as a basis to develop our motion planning algorithms, detailed in the next Chapters. Second, in order to fully understand our crowd engine, called Yaq, we briefly describe its general architecture and pipeline in Section 3.2. Finally, in Section 3.3, we show the time required to compute a Navigation Graph and the general performance obtained when fully simulating a crowd with Yaq.

3.1 Navigation Graph

When simulating crowds in any environment, a first and inevitable step is to find paths where pedestrians can navigate. Many solutions to structure an environment for such purposes have been introduced above. We want here to focus on the Navigation Graph approach [Pettré et al., 2006, 2007]. A Navigation Graph is a simple structure that represents an environment topology by distinguishing navigable areas from impassable obstacles. The Navigation Graph is composed of vertices and edges: the vertices are vertical cylinders representing areas where a pedestrian can freely walk without colliding with its environment. Edges are gates allowing a pedestrian to cross from one cylinder to another one. They are introduced between vertices wherever two cylinders overlap. Examples of Navigation Graphs computed on various environments are illustrated in Figure 3.1. Based on this structure, it is possible to answer path requests from a given point to another, and provide a set of waypoints to steer pedestrians towards their goal.

3.1.1 Construction

The construction of a Navigation graph is an automatic computation, performed in a preprocess, based on an environment geometry. The resulting graph has the advantage of supporting both path planning and simulation by capturing the environment topology, and distinguishing navigable areas from impassable obstacles and terrains. The requested inputs for the construction of the Navigation Graph are minimal:

- the environment geometry,
- a slope angle threshold, over which it is estimated that a pedestrian cannot pass,
- the smallest bounding box that can contain any pedestrian of the simulation, defined by its height h_c and radius r_c
- a computational precision

Once the key parameters have been entered, the Navigation Graph computation is performed in 4 steps, illustrated in Figure 3.2: first, an intermediate *navigation grid* is computed, based on the environment geometry (top left). The grid identifies the locations where characters are able to stand (top center). Second, navigability between these locations is captured by a grid mesh (top right). Third, the clearance map provides the distance from each point to the nearest obstacle (bottom right). Finally, the Navigation Graph is deduced from a correctly selected set of points belonging to the medial-axis of the clearance maps; *graph vertices* are cylindrical areas where navigability is guaranteed as shown in Figure 3.2 (bottom center), while *edges* are gates between these areas (bottom left). The following paragraphs detail each step.

Navigation Grid Points. Navigation grid points are deduced from a 3D regular sampling of the environment mesh, as illustrated in Figure 3.2 (top center). Note that this is not a 2.5D

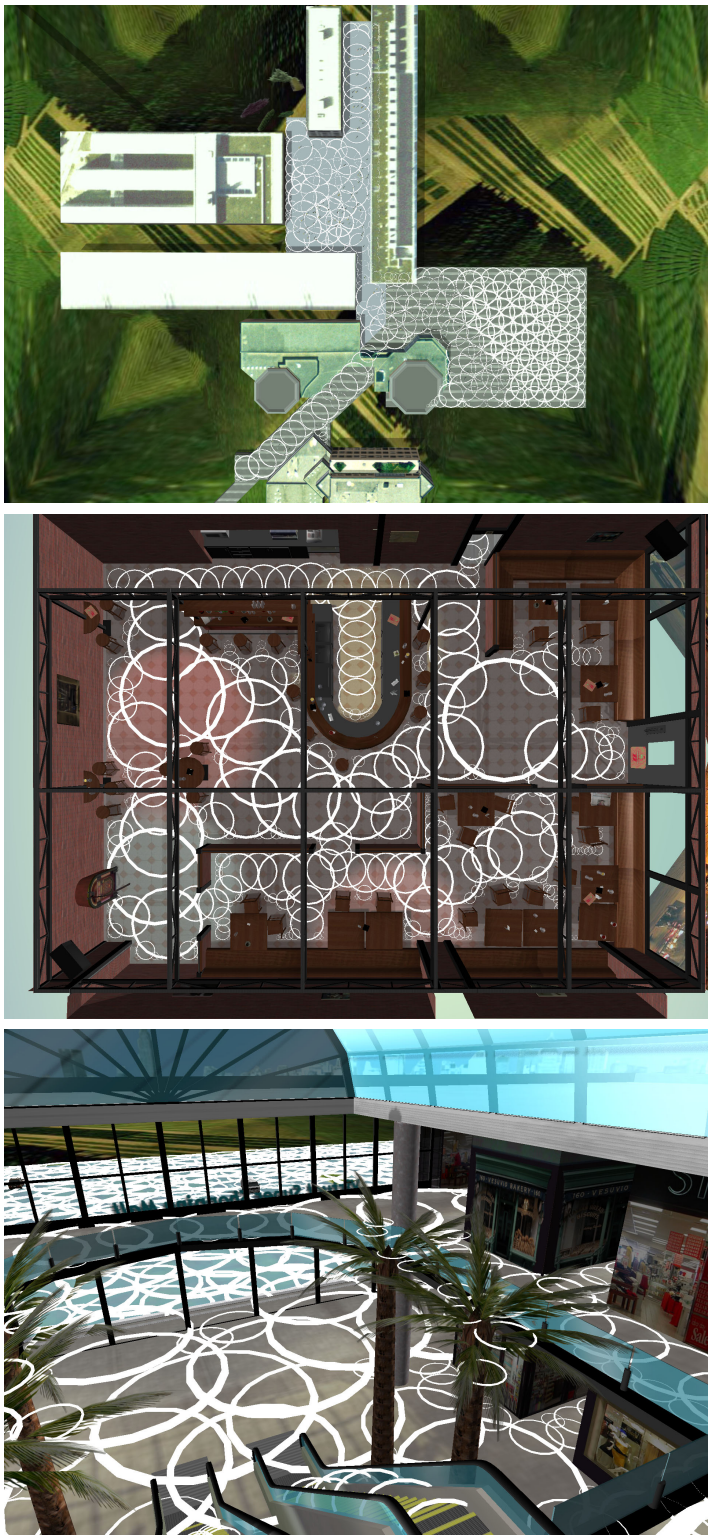


Figure 3.1: Some examples of Navigation Graphs automatically computed from the geometry of virtual environments. Note how the Navigation Graph is able to handle multi-layered terrains.

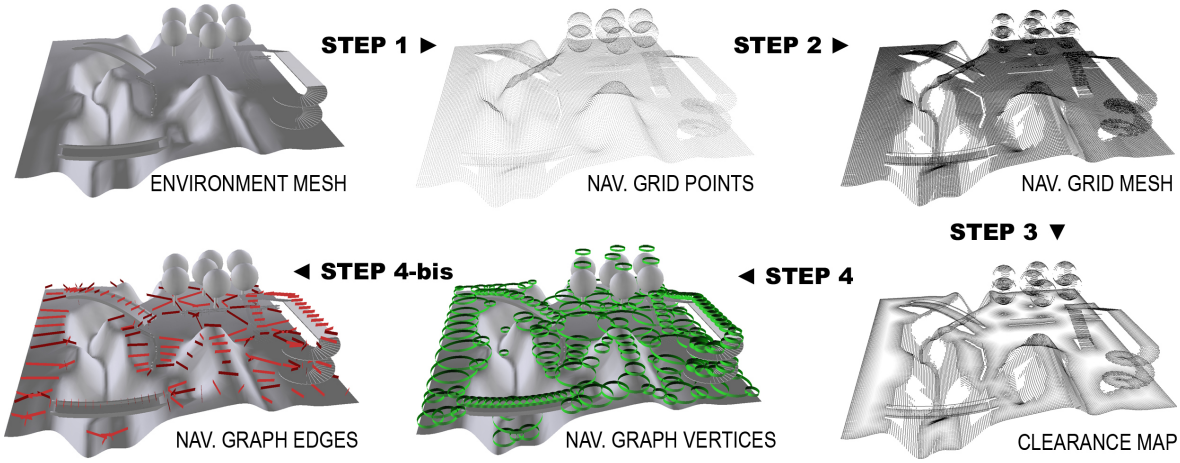


Figure 3.2: The construction of a Navigation Graph is a 4-step process based on an environment geometry [Pettré et al., 2006].

elevation map only: horizontal coordinates may refer to several altitudes. The size of the grid step is defined by the computational precision parameter input by the user. Grid points must correspond to collision-free locations where characters are able to stand; the points that are superposed vertically and separated with a too small distance are thus filtered (according to the user-defined maximum height h_c of the characters).

Navigation Grid Mesh. The second step consists in connecting each point to its valid neighbors: two neighboring grid points are connected if a character is able to move from one to the other, *i.e.*, if the terrain is flat enough, and the space between the points is obstacle-free. The slope of the terrain, formed by the pair of grid points is first compared to the slope angle threshold input by the user. Then, since vertical faces of obstacles have not been captured yet, we check if an obstacle stands between the two points. This can be done using any collision-checker. The solution used in the work of Pettré et al. [Pettré et al., 2006] is based on an OpenGL method processing the grid row by row, still in the orthographic mode, and observing vertical slices of the environment from the front and the side, as seen in Figure 3.2 (top right). As a result, each point is potentially connected to its 4 neighbor points. When a point is connected to less than 4 neighbors, this means that the point borders an obstacle or a too steep area. We call such a point a *border point*.

Clearance Map. The clearance map is the last required information in order to deduce the Navigation Graph. The clearance of a point is its distance to the nearest obstacle or impassable slope, *i.e.*, its distance to the nearest border point. Note that the border point computation takes the layered structure of the grid into account, so that a distance between points belonging to superposed surfaces is not considered. The image at the bottom right of Figure 3.2 illustrates the resulting clearance map: the shorter the distance to a border point, the darker the point.

Navigation Graph Construction. A *Navigation Graph* captures all the previously computed information in a compact manner. Its vertices represent navigable areas, modeled as cylinders lying on the walk surface and where absence of obstacles is guaranteed. Cylinders are easily deduced from the navigation grid and the clearance map: a cylinder can be centered on any navigation grid point. Its radius is set to the clearance value and its height to h_c: intrinsically, navigability is guaranteed inside it. However, to avoid redundant information, only a small subset of navigation grid points is used to construct the graph vertices. We first select navigation grid points belonging to the clearance map medial axis. Then, the grid point having the maximum clearance is selected to deduce a first cylinder (vertex). All medial axis points included in this cylinder are disabled, and the process is iterated until no point remains available for selection. The graph vertices are represented as cylinders in Figure 3.2 (bottom center), or navigable areas of the terrain. The graph edges are directly deduced from the previously computed cylinders: when two cylinders overlap, navigation from one to the other is possible, and an edge is then created between the two corresponding vertices. Edges are modeled as vertical gates delimited by the intersection of the cylinders in Figure 3.2 (bottom left). We illustrate in Figure 3.1 a few examples of Navigation Graphs automatically computed for various environments.

3.1.2 Path Planning

Path planning using agent-based approaches can provide individual paths for each pedestrian, resulting in more realistic crowd simulations. In cases where the crowd is very large however, such a fine-grained solution is unfeasible, and crowds usually navigate in groups, rather than individually. With Navigation Graphs, it is possible to make a trade-off between these two approaches. First of all, a crowd can be divided into user-defined groups, that share the same starting and destination points. For each group, the Navigation Graph can provide a variety of solutions, *i.e.*, for each pair of points, the pedestrians can be spread out on different paths joining them. Variety is obtained at 2 distinct levels: with the path search, and with the path width.

Path Search. Given a pair of vertices (or starting/destination points), the goal is to obtain several paths to connect them. By first invoking Dijkstra's algorithm, it is possible to find the shortest path. To apply this method, edge costs are initialized as the distance between their connected vertices. The resulting shortest path is described as a sequence of edges, *i.e.*, a sequence of gates to cross, which delimits a corridor of varying width between the 2 distant vertices. To find a second path, one can then assume that a congestion point appears along the first path, where the most narrow gate is present. Therefore, the corresponding edge cost is edited and increased (multiplied by 10). A new Dijkstra search is then achieved, resulting in a new path. The process is iterated until no more edge costs can be modified (only one cost modification per edge is allowed). The union of all paths found for a single pair of starting/destination points is called: a *navigation flow*. Figure 3.3 displays several itineraries obtained with this method, for the query of going from the left-side of the environment up to the footbridge on the right. After the shortest path is found (top-left image), many variants are discovered. The corresponding navigation flow (union of all paths) is shown in

the bottom-center image of Figure 3.3. This way, members of a same group are spread on all the found paths, thus offering a first sense of variety in the navigation of pedestrians.

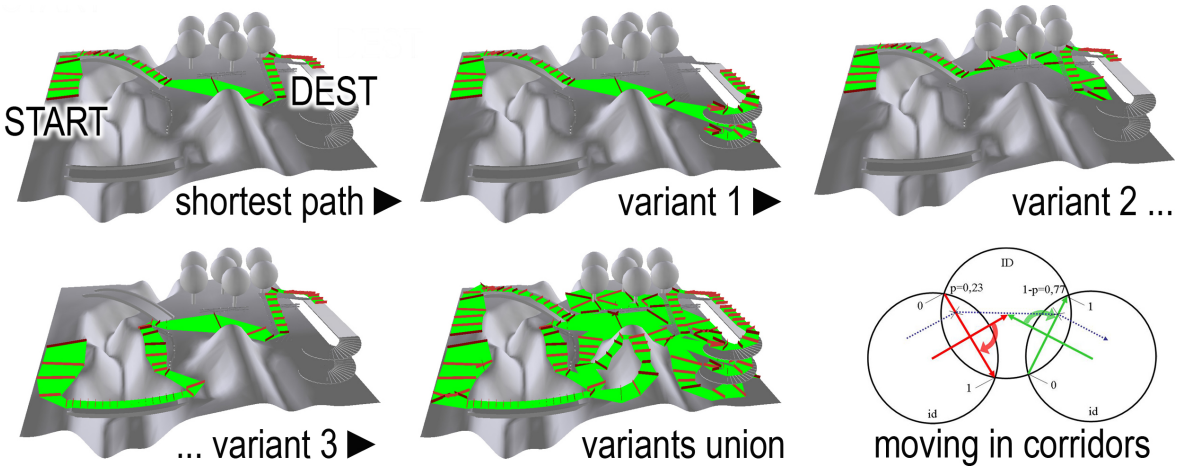


Figure 3.3: For a single query from the left to the right-side of the environment, several paths are successively found by increasing the cost of the most narrow edges. (*Bottom-right*) pseudo orientation of the graph and individualized trajectories [Pettré et al., 2006].

Path Width. A second step to further spread pedestrians sharing a same path is to exploit its corridor's width. For each pedestrian, the chosen path is individually transformed into a sequence of waypoints to follow, generated on each gate (edge) to cross. A parameter $p \in [0, 1]$ is defined for each pedestrian, and the exact position of a waypoint on a gate is determined, from left to right, as linearly dependent on p. As shown in Figure 3.3 (bottom-right), it is easier to compute waypoint coordinates if gates are oriented, thus allowing to distinguish the left from the right according to the crossing direction. A pseudo-orientation of the graph based on vertex numbering is used. For gates crossed in the reverse direction, $1 - p$ is used instead of p to compute the waypoint location. In Figure 3.3 (bottom-right), the vertex in the middle has a greater index than the two other ones, implying such an operation on p.

3.2 Yaq Architecture

Yaq is an engine dedicated to the simulation of large crowds in real time. This architecture has been built to integrate crowd rendering, animation, navigation, and behavior in real time. The work that we have achieved over the last 4 years is all integrated into Yaq. More specifically, our main focus has been oriented towards crowd motion planning and behavior, which will be further detailed in the next Chapters. For a better understanding of our work and its integration into this crowd engine, we start by providing an overview of Yaq in this Section. A more detailed presentation of the Yaq architecture, and how it handles crowd rendering and animation, is available in the work of Maim [Maïm, 2009].

To simulate crowds in real time, every component of Yaq's structure is optimized to treat the related data in an efficient manner. Yaq is composed of 6 main parts (illustrated in Figure 3.4), all treated successively in its runtime pipeline:

1. the *LOD setter* sorts all the virtual humans according to the level of detail in which they should be rendered;
2. the *simulator* is responsible for moving the pedestrians towards their goal;
3. the *animator* updates the animation of each character, whichever its level of detail;
4. the *renderer* displays the updated crowd;
5. the *path planner* is responsible for avoiding inter-pedestrian collisions;
6. the *behavior handler*, the final component of Yaq pipeline, takes care of updating the crowd behavior.

We detail the work achieved at initialization, and then at each consecutive step of the runtime pipeline in the following Sections.

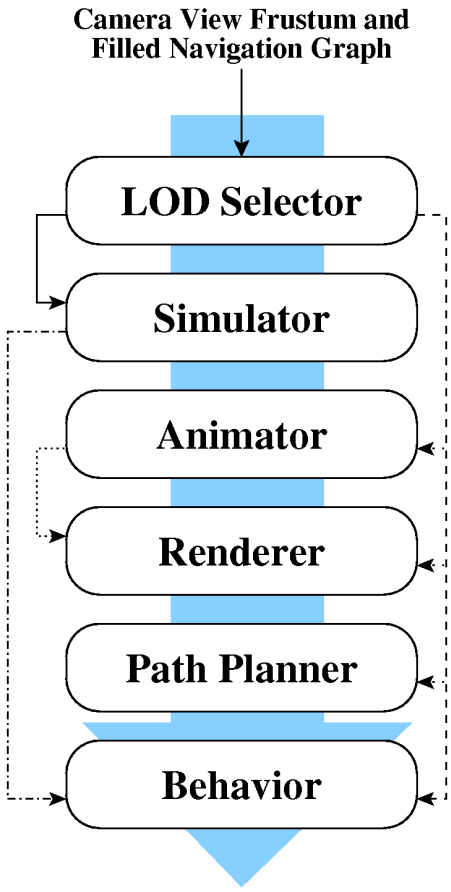


Figure 3.4: The runtime pipeline of Yaq is divided into 6 distinct parts, each responsible for a specific task.

3.2.1 Pre-Processes

There are a few important elements that need to be available in order for Yaq to correctly initialize and run the simulation.

Rendering

To render a crowd in Yaq, at least one human template has to be available, *i.e.*, a skeleton skinned with a mesh, and at least one texture mapped onto this mesh. To obtain realistic simulations, we usually set up Yaq to use at least five different human templates (as illustrated in Figure 3.5), and each of them possesses about 3 to 5 different textures. If color variety is to be achieved, as detailed in the work of Maim [Maïm et al., 2009], each human texture has to be complemented with 2 segmentation maps, that delimit the body parts associated to the texture. Finally, if all levels of detail, *i.e.*, dynamic meshes, static meshes, and billboards, are to be used in the simulation, the static meshes and billboards have to be pre-computed. The reader is advised to refer to the work of Maim [Maïm, 2009] for more details on this aspect.

Figure 3.5: Five human templates from Yaq. Each human template is composed of a skeleton skinned with a mesh to enable runtime animations and at least one texture mapped onto the mesh.

Animation

To be able to animate the characters, Yaq takes advantage of the work of Glardon et al. [Glardon et al., 2004b,a]. They introduced a PCA-based walk engine capable of animating on the fly human-like characters of any size and proportions by generating complete locomotion cycles. Their locomotion engine is based on the capture of several walk and run motions,

from which they have created a normalized model. This model is able to generate a new locomotion cycle based on 3 high-level parameters (see Figure 3.6):

- Personification weights : 5 people, different in height and gait have been captured while walking and running. This variable allows the user to choose how he wishes to parametrize these different styles.
- Speed : motions have been captured at many different speeds. This parameter allows to choose at which velocity the walk/run cycle should be generated.
- Locomotion weights: this parameter defines whether the cycle is a walk animation, a run animation, or a blend between them.

The locomotion engine is thus able to generate a whole range of varying walk/run cycles for a given character.

To efficiently animate the *dynamic meshes* in Yaq (the characters the closest to the camera, animated at runtime), a large number of locomotion cycles are generated in a pre-process for each human template. Other animations needed for idle humans, *e.g.*, sitting, standing, talking, are generally hand-designed. The total amount of animation clips is very cheap to store, because each animation can be represented with the skeleton movements only.

For *static meshes* and *billboards* however, only a few animations can be pre-computed, because they are much more costly in terms of memory storage: a static mesh animation is described by the positions of the mesh vertices at each keyframe, and each keyframe of animation for a billboard is represented with a 2D image of this posture from several points of view. For this reason, a limited subset of walk and run animations are usually selected to be pre-computed and sampled at 25Hz for static meshes and billboards. Other idle animations used in Yaq can often be replaced with a single frame of animation for static meshes and billboards. Since these representations are not used directly in front of the camera, a frozen representation is sufficient to preserve the illusion of moving for this type of quiet actions.

To store and manage all the generated data, an embedded database is used. This way, the data is efficiently packed and it can be augmented with meta-information, such as the duration of the animation, its type, and to which template the animation applies. For instance, a locomotion animation can be efficiently stored along with its speed and style.

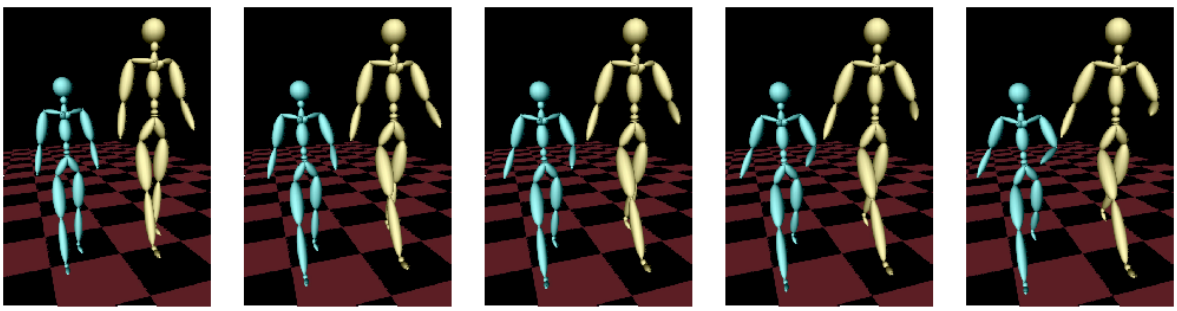


Figure 3.6: Results obtained in the work of Glardon et al. [Glardon et al., 2004a]: postures of two skeletons with different sizes playing a walk cycle at an identical speed *(from left to right)* 2.0, 4.0, 6.0, 8.0, and 10.0 km/h.

Navigation

In order to display the scene in which the crowd will be simulated, an environment geometry has to be provided in the COLLADA[1] format [The Khronos Group]. Also, to enable pedestrians to navigate in the chosen environment, a Navigation Graph has to be generated, as presented in the previous Section. An optional semantic model of the scene can also be provided to later trigger situation-based behaviors in the crowd. We do not further explain how this second model is treated in a pre-process, as it is fully detailed in Chapter 6.

3.2.2 Runtime Pipeline

At runtime, it is very important to optimize the different pipeline stages to be able to simulate crowds in real-time. Efficient crowd simulation is obtained by targeting computing resources where the attention of the user is focused. After initialization of Yaq, its pipeline, illustrated in Figure 3.4, starts running six important successive steps at each frame: the LOD setter, the simulator, the animator, the renderer, the path planner, and the behavior handler.

Initialization

When the simulation is launched, the initialization proceeds in four steps. First, all the pre-computed data is loaded, so that it is available for runtime requests. Second, based on the Navigation Graph and the user directives, *i.e.*, a list of starting/destination points between which the crowd should navigate, a series of paths are computed using Dijkstra's algorithm, as previously introduced. The third step is to instantiate the human templates many times to reach the desired size of the crowd. For each instance, a path, a direction, and a position on the path are assigned. Note that each instance knows in which graph vertex it is situated, and reciprocally, each vertex has a list of pedestrians currently in its area. Finally, the colors to apply to each instance's body parts are randomly selected within a constrained range, and uploaded onto the GPU. The pipeline can then start running, and receives two major inputs: the camera view frustum, and the Navigation Graph filled with virtual humans. This graph is used for crowd motion planning and as a general-purpose structure to hierarchically process virtual humans throughout the pipeline.

LOD Setter

The LOD setter is the first stage of the pipeline. User focus is determined by simple rules that allow to spread computing resources throughtout the environment. Depending on the camera frustum, each Navigation Graph vertex is categorized with two scores :

- *A representation score*, determined by finding the distance from the vertex to the camera and its eccentricity from the middle of the screen. This score determines the rendering level of detail of pedestrians inside the vertex.

[1] COLLADA is a royalty-free XML schema that enables digital asset exchange within the interactive 3D industry.

- *A score of interest*, resulting in an environment divided into regions of different interest (ROI). For each region, we choose a different motion planning algorithm. Regions of high interest use accurate, but more costly techniques, while regions of lower interest exploit simpler methods (more on this in the next Chapters).

A special scoring is applied to invisible vertices: those containing no humans are directly frustum culled without scoring and not further considered in the current frame. The invisible vertices filled with at least one character are kept in the pipeline, but for them, only the ROI score needs to be computed. Indeed, no rendering is needed for these humans, but they still require a minimal simulation to move along their path.

According to the scores obtained, we keep several lists of pedestrians up-to-date: one list sorts them by level of representation (or level of detail), and another sorts them according to their level of interest. Updating these lists may appear as time consuming, but it is worthwhile, since it enables a much faster processing in the next steps of the pipeline; the context switches are greatly reduced.

Simulator

The purpose of the second stage of the pipeline, the simulator, is to ensure that each pedestrian comes closer to its next waypoint. Indeed, each virtual human stores a waypoint, which is the position of its next short-term goal to reach. In other words, the simulator takes care of correctly updating the positions of all pedestrians. Note that the manner in which this update and the next waypoints are computed directly depends on the ROI in which the pedestrians are situated. Further explanations on these calculations are provided in Chapters 4 and 5.

Animator

The third stage is responsible for animating characters whichever the representation they are using. Yaq uses three different representations decreasingly costly to render and animate : dynamic meshes, which are deformed in real time using a skeleton and a skinning technique, static meshes, whose deformations are pre-computed for a chosen selection of animation clips, and billboards, extensively exploited in the domain of crowd rendering. The runtime animation process is similar for all representations: depending on the pedestrian's current animation time, the correct keyframe is identified and retrieved. Then, each representation is modified accordingly.

Concerning the walking and running animations, we would like to point out that the locomotion engine [Glardon et al., 2004b,a] generates animation cycles that include the global translation of the character, *i.e.*, its forward progression. However, this global translation is not used in Yaq, for we lack the animations of characters turning left or right. Thus, it is the simulator rather than the animator, that takes care of moving the walking characters forward. This approach offers more freedom to steer the virtual humans as one sees fit. On the other hand, care must be taken to limit cumbersome foot sliding effects. For dynamic meshes, satisfying results are obtained by changing the animation cycle of pedestrians several times per second, depending on their moving speed. For static meshes and billboards, which have

a much reduced animation repertoire, the gap between the played cycle and the actual speed of a virtual human is less noticeable, since these representations are used at farther distances.

Renderer

The renderer stage represents the phase where draw calls are issued to the GPU to display the crowds. Shadows are also handled at this stage using a shadow mapping algorithm [Reeves et al., 1987]. The whole process thus becomes a two-pass algorithm: first, dynamic meshes, static meshes, and billboards are sequentially rendered from the point of view of the main light. Then, the process is repeated from the point of view of the camera. Accessories, used to vary the shape of characters, are also rendered at this step.

There are three types of GLSL-based [Rost, 2004] shaders in Yaq's system corresponding to each rendering level of detail. The dynamic mesh shader deforms the mesh in the vertex program and uses Phong lighting in the fragment shader. Except for deformation, the static mesh uses the same set of shaders. For the billboards, the shader is kept simple, since it only displays textured quads, and most of the work is done on the CPU. During a frame update, the number of shader switches is kept to a minimum by rendering all the individuals of the same level of detail in the same pass. To this end, the updated sorted list of pedestrians from the LOD setter stage is used. In Figure 3.7, we show the distance at which the three levels of detail are used: the red characters are dynamic meshes, the green ones are static meshes, and the blue ones are billboards.

The renderer is the last stage processing the current frame. The last two pipeline stages operate on the subsequent frame, one step ahead.

Path Planner

The path planner stage performs the collision avoidance between pedestrians. Due to a complexity in $O(n^2)$, it runs at a significantly lower frequency than the previous stages. Regions of high interest, typically in the vicinity of the camera, are treated with a long-term collision avoidance strategy, while other ROI are treated with short-term algorithms (see next Chapters).

Behavior Handler

Finally, the last stage of the crowd pipeline is the behavior handler. During the entire pipeline, virtual humans cannot change their current animation type, *e.g.*, from locomotion to idle, because it would invalidate the various sorted lists of Yaq. This last stage is thus the only one which is allowed to change the motivation and current animation type of virtual humans. It is always achieved at the end of the pipeline, for the next frame.

In Yaq, crowd behavior is situation-based, *i.e.*, behavior is updated individually for each virtual human, based on their position in the Navigation Graph. At initialization, some graph vertices are associated to specific behaviors, *e.g.*, buy food, look at a specific point, play a special animation etc. Note that it is possible for a graph vertex to be associated with several behaviors. For each type of behavior, the corresponding graph vertices are identified, and the

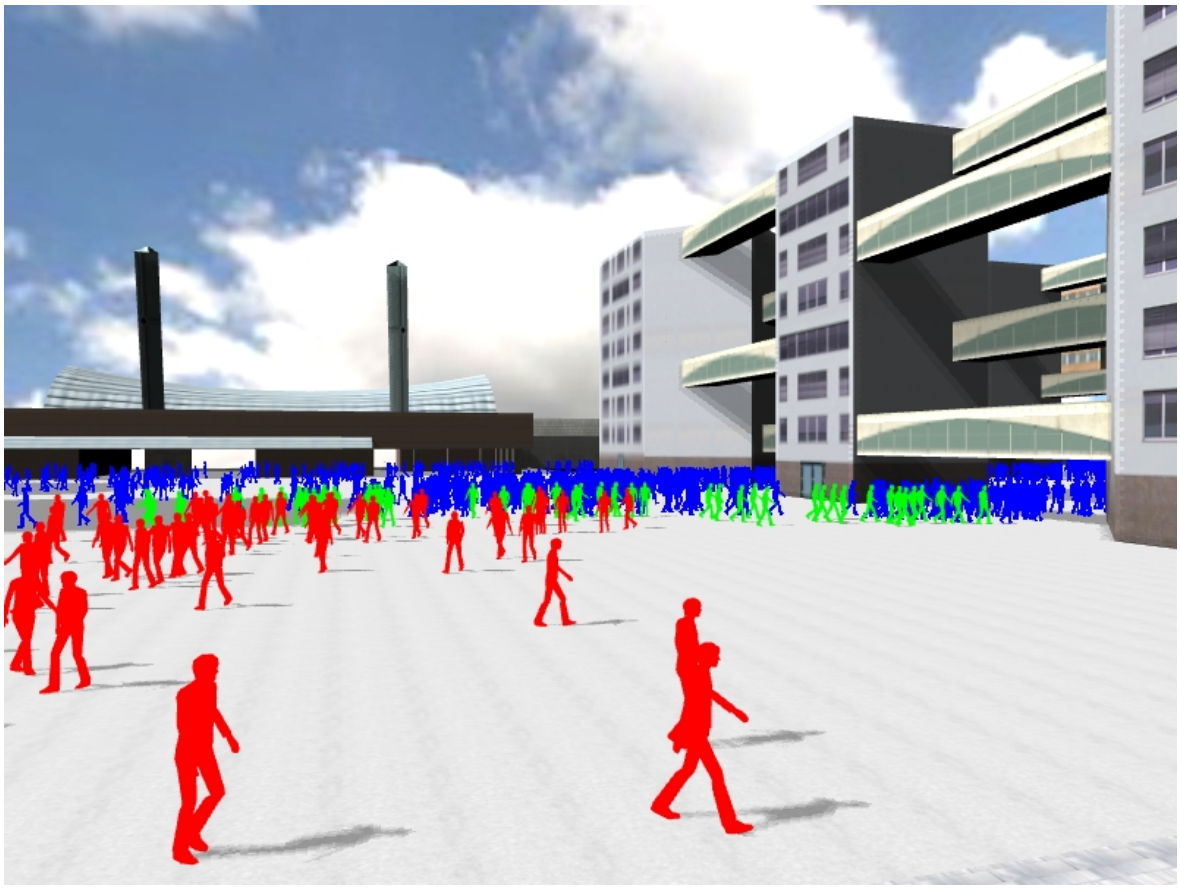


Figure 3.7: The 3 levels of detail used to represent virtual humans in Yaq: dynamic meshes in red, static meshes in green, and billboards in blue.

virtual humans in these vertices are urged to perform the corresponding actions. The changes triggered by the behavior handler do not require to be immediately taken into account. We thus limit this stage to a frequency of 10Hz. We fully explain how behaviors are applied to crowds in Chapter 6.

3.3 Performance of Yaq

The tests presented hereunder have been measured on a desktop computer with a Pentium Xeon 3.2 GHz processor, 1 GB of memory and a nVidia GeForce 7800 GTX with 256 MB of on-board video memory.

Note that these results are presented here for two reasons: first, to provide the reader with an idea on the time required to compute a Navigation Graph and generate paths with the algorithm presented in Section 3.1, and second, to give a global overview of the crowd engine's performance when all steps of the pipeline are processed. Several other tests, more focused on the sole motion planning of crowds, have been run individually, and are fully disclosed in the next Chapters.

The graph in Figure 3.8 illustrate the capacities of Yaq to simulate and render crowds with various levels of detail introduced earlier. In Table 3.1, we show how many triangles are rendered for each level of detail. Note that we have performed several tests with dynamic meshes by varying the number of triangles they are composed of. According to Figure 3.8, one can see that the system is able to simulate and render a large number of pedestrians with real-time performance. To simulate a crowd of 1,000 pedestrians or less, the lowest levels of detail are not even required. Interactive frame rates (10 fps) are obtained with up to 30,000 zoomed-out visible pedestrians. When a major part of the pedestrians remain invisible, due for example to a low density of population, Yaq can handle an even larger crowd.

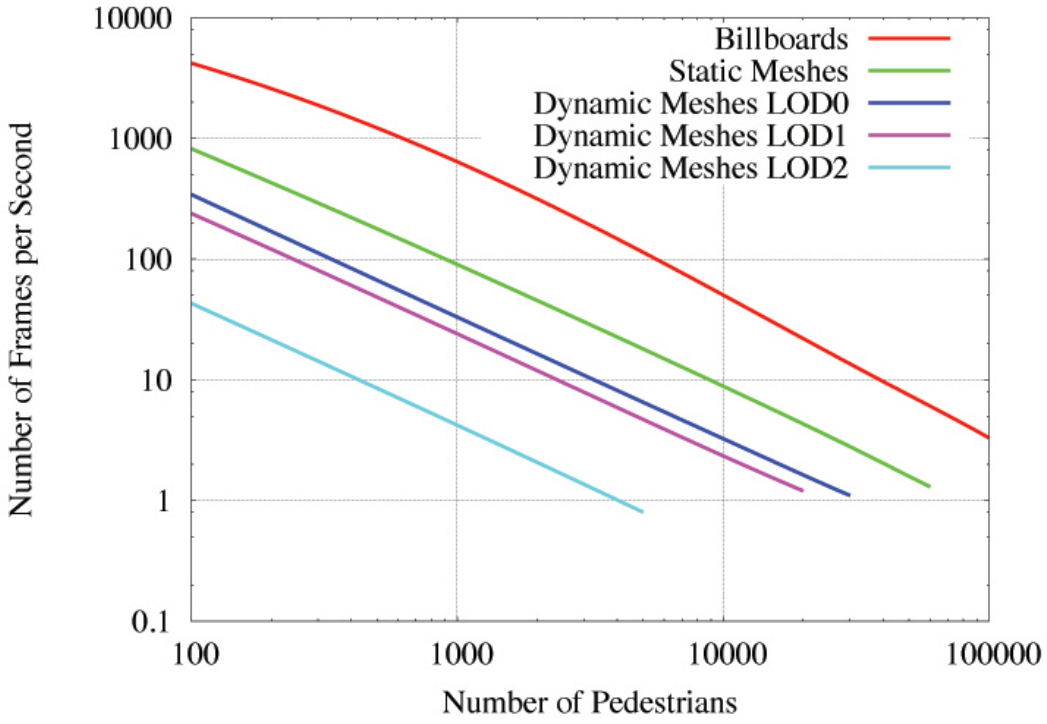


Figure 3.8: Frame rate obtained for each level of detail with a crowd of a growing size.

	Billboards	Static meshes	Dynamic LOD 0	Dynamic LOD 1	Dynamic LOD 2
Template1	2	243	243	1043	5945
Template2	2	102	102	960	5906

Table 3.1: Number of triangles to represent a virtual human in various levels of detail.

Table 3.2 gives the information and computation time measured on the three examples introduced in the following paragraphs. The computation time of Navigation Graphs mainly depends on the environment size and the defined precision: indeed, in the case of the city presented below, the intermediate navigation grid is made of a large number of points and many distance computations have to be performed to deduce the clearance map. The distance computation time could be improved using graphics hardware based methods [Hoff et al.,

1999]. The time needed for planning includes both path searches in the graph and waypoint computation for all pedestrians. Given the complexity of Dijkstra's algorithm, this duration essentially depends on the number of edges and vertices composing the graph.

	Itranarium	Planet Eight	City
Environment dimensions	$80 \times 80\ m.$	$175 \times 90\ m.$	$630 \times 430\ m.$
# Triangles	$20,646$	768	$35,871$
# Pedestrians	$1,000$	$10,000$	$35,000$
Graph computation time	$14\ sec.$	$6\ sec.$	$\sim 16\ min.$
Precision	$0.25\ m.$	$0.25\ m.$	$0.5\ m.$
# Vertices / #Edges	346 / 477	27 / 27	$5,017$ / $8,003$
Planning time (# paths)	$15.6\ ms$ (41)	$2.2\ ms$ (2)	$1.82\ sec.$ (350)

Table 3.2: Characteristics of 3 environments and the corresponding time required for Navigation Graph and path computations.

Itranarium

In this first example, we illustrate the abilities of the Navigation Graph to efficiently spread a crowd of people navigating between the same pair of starting/destination points. The environment is introduced in Figure 3.9: the itranarium is a crowded touristic place where 1,000 pedestrians walk between the indicated points 1 and 2 (left). The environment topology allows many itineraries (right). As the number of people is low and the environment size is small, we use high-quality settings: only two fidelity levels for the graphical models (dynamic and static meshes) and high frequency updates with collision avoidance for the simulation. The average frame rate of this simulation is of 30 fps.

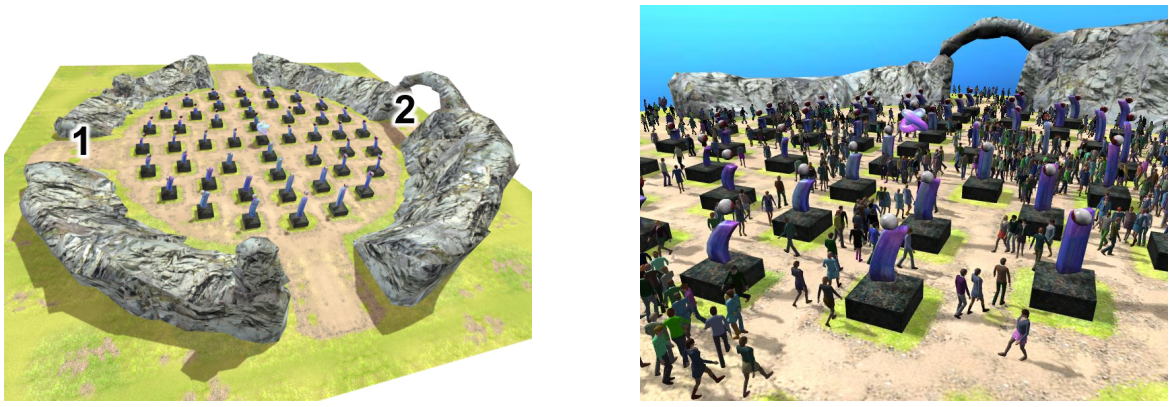


Figure 3.9: *(Left)* A Navigation Graph has been computed for the itranarium environment, and a single pair of starting/destination points was input. *(Right)* The use of Dijkstra's algorithm as presented in Section 3.1 allows to spread the crowd, even with a single pair of input points [Pettré et al., 2006].

Planet Eight

In this example, illustrated in Figure 3.10, we show the crowd engine capabilities. The environment is made of smooth slopes and superposed surfaces. The graphical models of the 10,000 pedestrians navigating between locations 1 and 2 range from dynamic meshes to billboards. Simulation is kept minimal; inter-pedestrian collision avoidance is deactivated. However, by tuning the distribution of parameter p (see Section 3.1), we separate the two flows of people in opposite directions. Thus, we illustrate the abilities of the navigation planning to provide variety in trajectories, with a low computational cost and a believable result. The average frame rate was of 20 fps.

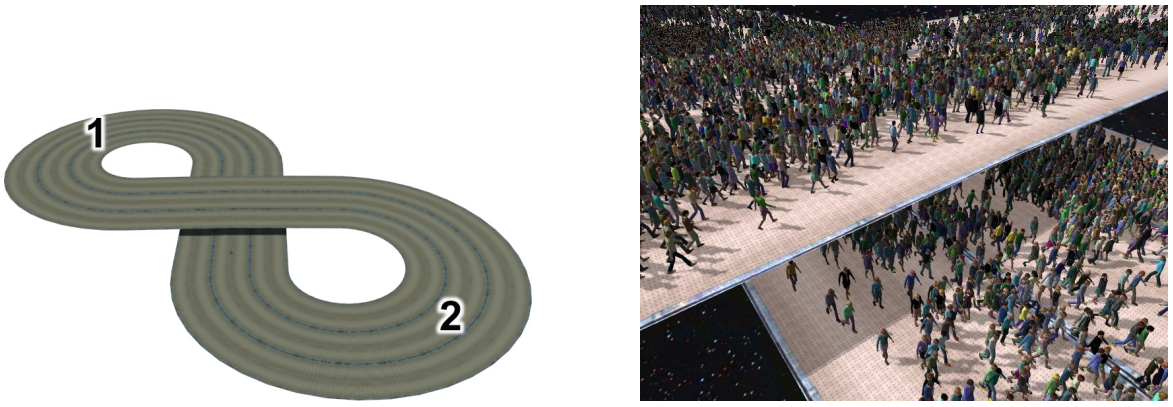


Figure 3.10: *(Left)* A Navigation Graph has been computed for the planet eight environment, and a single starting/destination points were input. *(Right)* The slopes in the environment are sufficiently gentle to allow the crowd to move on them. The two levels of the environment are correctly handled by the Navigation Graph [Pettré et al., 2006].

City

This last example illustrates a large urban environment (Figure 3.11), where we define 8 significant destinations. A population of 35,000 pedestrians is composed of 7 groups (of 5,000 pedestrians each) navigating back and forth between pairs of significant destinations. All the available levels of detail are used. Due to path variety, even with such a reduced set of destinations, pedestrians are spread and no place remains empty. During exploration, the frame rate ranges from 10 (for large views of dense areas) to 20 fps. As expected, places defined as significant destinations are more densely populated than others. Mandatory passages such as the bridges around the hotel, are highly crowded.

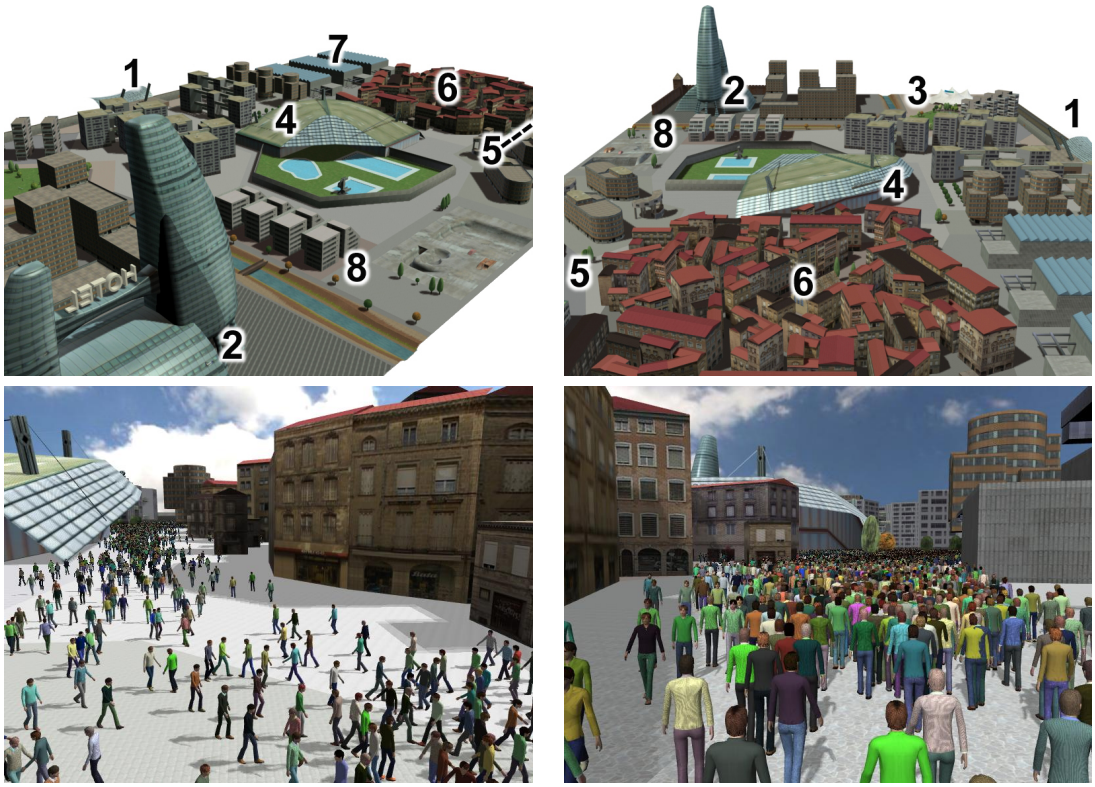


Figure 3.11: *(Top)* Seven pairs of starting/destination points were chosen between the 8 important places of the city for path computation. *(Bottom)* The crowd uses 3 levels of detail and is spread over the whole city [Pettré et al., 2006].

CHAPTER 4

Motion Planning Architecture

Realistic real-time motion planning for crowds has become a fundamental research field in the Computer Graphics community. The simulation of urban scenes, epic battles, or other environments that show thousands of people in real time require fast and realistic crowd motion. Domains of application are vast: video games, psychological studies, and Architecture to name a few. In this Chapter, we present the motion planning architecture of Yaq, offering a hybrid and scalable solution for real-time motion planning of thousands of characters in complex environments. Two illustrations of the results obtained using this hybrid solution are visible in Figures 4.1 and 4.9.

In Section 4.1, we first provide an introduction to our motion planning architecture. Then, in Section 4.2, we present the two main approaches we have taken to introduce levels of detail in the simulation. The method that we have adopted in the end is presented in Section 4.2.2. Our implementation of this solution is detailed in Section 4.3. Finally, we show the performance obtained in Section 4.4 and discuss our results in Section 4.5.

4.1 Introduction

In our perspective, crowds are formed by thousands of individuals that move in a bounded environment. Each pedestrian wants to reach his individual goal in space, avoiding obstacles, and remaining close to his friends or family. People perceive their environment, and use this information to choose the shortest path in time and space that leads to their goal. Emergent behaviors can also be observed in crowds, *e.g.*, in places where the space is small and very crowded, people form lanes to maximize their speed. Also, when dangerous events occur, pedestrians tend to react in very chaotic ways to escape.

Figure 4.1: Pedestrians using our hybrid motion planning architecture to reach their goal and avoid each other in a city environment.

Planning crowd motion in real time is a very expensive task, which can be divided into three distinct parts: path planning, obstacle avoidance, and group cohesion. Path planning consists in finding the best way to reach a goal. The path selection criteria are the avoidance of congested zones, and minimization of distance and travel time. Path planning must also offer a variety of paths to spread pedestrians in the whole scene. Obstacles to avoid can either be other pedestrians or objects that compose the environment. The goal of each individual is to inhibit collisions with such obstacles. As for groups, they are very often represented by 2 to 4 pedestrians walking side by side, and avoiding separation as well as inter-collision. In the context of real-time simulations, all three aspects of motion planning need to be efficiently addressed to produce believable results.

Multiple motion planning approaches for crowds have been introduced. As of today, several fast path planning solutions exist. Dynamic avoidance and high-level group behaviors however, remain expensive tasks. Agent-based methods offer realistic pedestrian motion planning, especially when coupled with global navigation. This approach gives the possibility to add individual and cognitive behaviors to each agent, but becomes too expensive for large crowds. Potential field approaches handle long and short-term avoidance. Long term avoidance predicts possible collisions and inhibits them. Short term avoidance intervenes when long-term avoidance cannot prevent a collision. These methods offer less believable

results than agent-based approaches, because they do not provide the possibility to individualize each pedestrian behavior. However, they have much lower computational costs.

In the remaining Chapters, we detail a motion planning architecture that realistically handles crowd motion planning in real time. Our approach provides a complete solution for all three aspects of crowd motion, *i.e.*, path planning (following Sections), short-term obstacle avoidance (Chapter 5), and group behaviors (Chapter 6).

To obtain high performance, our approach is scalable: we divide the scene into multiple regions of varying interest, defined at initialization and modifiable at runtime. According to its level of interest, each region is ruled by a different motion planning algorithm. Zones attracting the user attention exploit accurate methods, while computation time is saved with less expensive algorithms in other regions. Our motion planning architecture also ensures that no visible disturbance is generated when switching from an algorithm to another.

The results we have obtained by integrating this architecture into Yaq show that we can simulate up to $10,000$ pedestrians in real time with a large variety of goals. Also, small groups are created and their members keep close to each other, as in reality. Finally, the possibility to introduce and interactively modify the regions of interest in a scene allow the user to choose the performance and distribute computation time accordingly. We illustrate in Figures 4.1 and 4.9 pedestrians taking advantage of our architecture to plan their motion in two environments.

4.2 Environment Classification

The foundation of our motion planning architecture is based on Navigation Graphs, automatically extracted from the mesh of an arbitrary environment [Pettré et al., 2006]. This approach has the advantage of robustly handling path planning. Vertices represent cylindrical zones of the walkable space, while edges are the gates where pedestrians can cross the space from one vertex to another. To connect two distant vertices, it is possible to create a *navigation flow*, composed of a set of varied paths. An example of navigation flow is shown in Figure 4.2. Thanks to this approach, pedestrian spreading is ensured. During simulation, pedestrians are assigned one navigation flow, and one direction. When they reach an extremity of the flow, they reverse their direction, and choose a new path, minimizing their travel time, *e.g.*, avoiding congested areas. Vertices offer a suitable structure of the walkable space; they can be exploited to classify different regions of the scene. We here present two approaches that have been tested within the framework of Yaq, and discuss their advantages and drawbacks. The first approach classifies vertices to define several *levels of simulation*, that are updated at different frequencies, depending on their distance to the camera. The drawbacks encountered with this approach have been overcome with our second approach, implementing various navigation strategies, depending on the level of interest of each region.

4.2.1 Levels of Simulation (LOS)

The first step we have taken to introduce motion planning in crowds was to use a Navigation Graph for path planning. With a Navigation Graph, each individual is informed of the

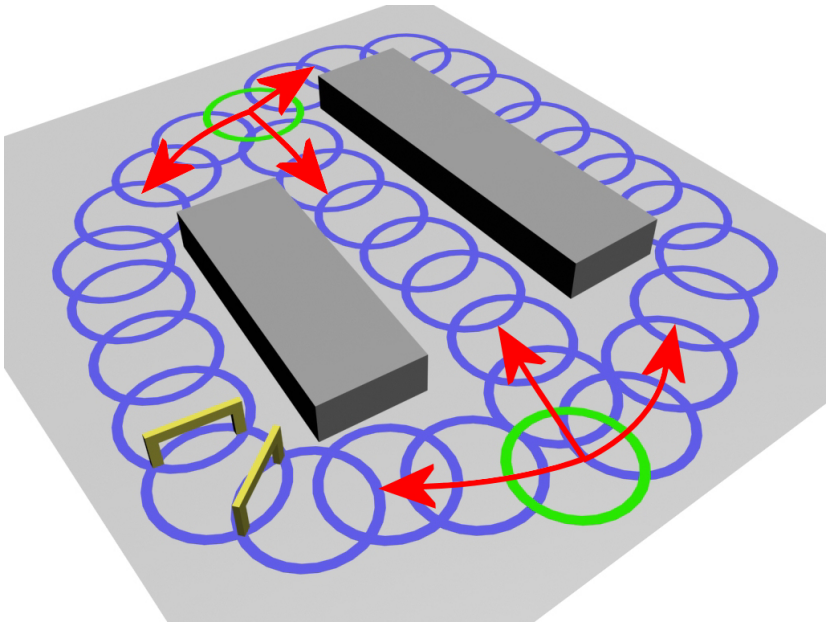


Figure 4.2: A Navigation Graph composed of a single navigation flow (in blue) connecting two distant vertices (in green). The navigation flow is composed of three paths that can be followed in either direction (red arrows). Two edges are also represented as gates (in yellow).

next waypoint to follow in order to reach a goal, while avoiding static obstacles within the environment. However, Navigation Graphs are not suited to handle inter-pedestrian collision avoidance. To manage this aspect of motion planning, we first implemented a simple force-field-based method, where waypoints are attractive forces, while pedestrians repulse each other, as introduced in the work of Khatib [Khatib, 1986]. Despite the simplicity of this simulation model, we soon realized that the potential size of crowds remained limited. At this point, to break limitations, we decided to introduce scalable algorithms. Our first attempt was to scale the frequency of simulation [Pettré et al., 2006].

Levels of simulation (LOS) allow to distribute the available computation time spatially and temporally according to the spectator's point of view: its quality is enhanced where attention is focused and progressively decreases toward invisible zones. First, we distribute levels of simulation for each Navigation Graph vertex according to the point of view, as illustrated in Figure 4.3. Note that the highest simulation quality is around the view point (number 4), including just behind the camera. The level remains high in front of the user (number 3), even at far distances, because the human eye is sensitive to motion continuity; on the side views, quality is progressively decreased (number 2 and 1), until it becomes the lowest one in invisible areas (number 0).

In invisible and very far areas, the pedestrians' position is updated at low frequencies. Also, different methods are used to steer pedestrians toward waypoints: a linear steering is used for low quality levels, while smoother trajectories are produced using the method proposed by Reynolds [Reynolds, 1999] for higher levels. Inter-collisions are checked between a limited number of pedestrians. Indeed, at a far distance, collisions are hardly detectable. As a result, we only solve them where the simulation level is the highest. Nevertheless, if a

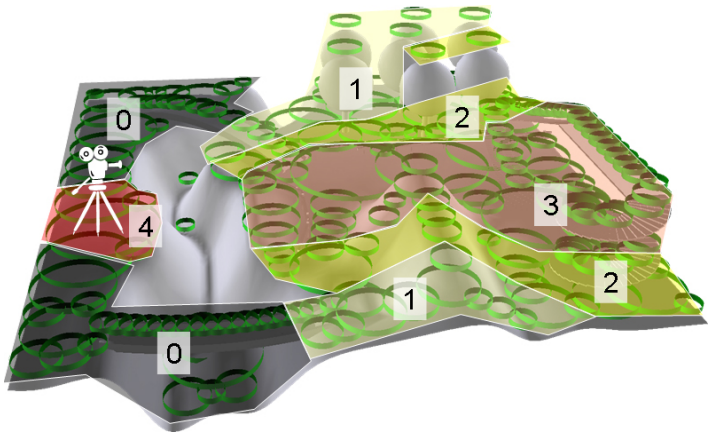


Figure 4.3: Five levels of simulation are distributed according to the user's point of view. The numbers decrease with the accuracy of simulation.

congested zone is observed, taking into account all the pedestrians contained in such high-level vertices would result in a prohibitive computation time. Thus, among them, only those that are nearer than a specified distance are selected. This distance is inversely proportional to the local population density. Note that we use a list referencing pedestrians for each vertex to execute all previous tasks efficiently.

With this first approach, we have encountered two main drawbacks. Firstly, frequency modifications can be exploited, but in invisible areas only. Indeed, we have noticed that a frequency drop is very noticeable when pedestrians are visible, even if they remain very far from the user's point of view. A second drawback we have noticed is that collision avoidance between pedestrians are necessary in all visible zones, except if the scene extends to the horizon. Indeed, the user's attention usually switches to different places of the scene, and do not necessarily focus just in front of the camera. Pedestrians moving through each other in the back of the scene may thus be noticed.

For these reasons, motion planning in Yaq has been transformed and adapted to scale the actual algorithm used for navigating crowds, rather than the frequency of simulation. We detail this approach, which is still used in Yaq, in the following Section.

4.2.2 Regions of Interest (ROI)

The goal of our architecture is to handle thousands of pedestrians in real time. We thus exploit the above mentioned vertex structure to divide the environment into regions ruled by different motion planning techniques. We classify these regions with a level of interest. The most interesting zones are ruled by realistic but expensive techniques, while others use simpler and faster solutions. Regions of interest (ROI) can be defined in any number and anywhere in the walkable space with high-level parameters, modifiable at runtime. Such flexibility is indeed desirable: it allows the user to first choose the wanted performance, and then distribute ROI, *i.e.*, computation time, as wished.

By defining three different ROI, we obtain a simple and flexible architecture for realistic results: **ROI 0** is composed of vertices of *high* interest, **ROI 1** regroups vertices of *low*

interest, and **ROI 2** contains all other vertices, of *no* interest.

With this classification, it is possible to divide the environment into many zones, each tagged with the appropriate level. In practice, we position the ROI with respect to the camera position and field of view. ROI 0 is usually directly in front of the camera, and/or in zones where important events occur. Two examples of such arrangements are shown in Figure 4.4. ROI 1 covers the remaining visible space, while ROI 2 includes all vertices outside the view frustum. Note that this choice is arbitrary, and that our architecture is versatile enough to satisfy any other environment decomposition.

Figure 4.4: Zones of the environment categorized as ROI 0 are indicated with yellow circles. *(Left)* ROI 0 can be defined close to the camera , or *(right)* near an event that is likely to attract the spectator's attention, like a threatening car.

For regions of no interest (ROI 2), path planning is ruled by the Navigation Graph. Pedestrians are linearly steered to the list of waypoints on their path edges. To use the minimal computation resources, obstacle avoidance is not handled.

Path planning in regions of low interest (ROI 1) is also ruled by the Navigation Graph. To steer pedestrians to their waypoints, an approach similar to Reynolds' is used [Reynolds, 1999]. Obstacles are avoided, thanks to an agent-based short-term algorithm providing efficient and realistic results.

In the regions of high interest (ROI 0), path planning and obstacle avoidance are both ruled by a potential field-based algorithm, similarly to the work of Treuill et al. [Treuille et al., 2006]. Compared to agent-based approaches, potential fields are less expensive, and still offer results more realistic than the ones of ROI 1 and 2, because collision avoidance is planned in the long-term. Nevertheless, in certain situations, this approach fails to avoid collisions. To overcome this problem, the same short-term algorithm as in ROI 1 is also activated in ROI 0.

Group behavior is an additional layer of the architecture that can be used in order to simulate pedestrians walking with friends or family. This algorithm, detailed in a later Chapter, is purely based on the edition of pedestrian waypoints. Thus, it can be used for all ROI introduced above.

An important concern when dealing with regions ruled by different motion planning algorithms is to keep smooth and unnoticeable transitions at their borders. The way we place

ROI implicitely solves this issue. Firstly, the border between ROI 2 and ROI 1 is always outside the view frustum, and thus does not require any specific attention: a pedestrian that may abruptly change directions when switching from ROI 2 to ROI 1 is still invisible to the spectators. Secondly, passing the borders between ROI 0 and ROI 1 is always smooth, because both regions use the same short-term avoidance algorithm: sudden changes in directions are most of the time triggered by the short-term avoidance algorithm. If such an abrupt orientation change is triggered for a pedestrian just before it leaves ROI 1, the same algorithm is used when it enters ROI 0, and thus, the avoidance strategy remains the same. The next Section detail our practical implementation of three ROI.

4.3 Implementation

We present below the details of our hybrid architecture implementation. We mainly focus on the initialization and runtime operations to construct and manage the scalable crowd motion planning. We first detail the initialization phase, *i.e.*, the grid construction over the graph space, the initialization of the structure of neighbor cells and of the ROI. Then, we describe our runtime pipeline. It is composed of five stages: the classification of graph vertices in correct ROI, the potential field computation, our short-term avoidance computation algorithm, the pedestrian steering phase, and finally, the continuity maintenance between grid and Navigation Graph. Note that we here detail all the pipeline stages, except the short-term avoidance algorithm, which is fully described in the next Chapter.

Initialization

First of all, for the given environment, a Navigation Graph is generated, and navigation flows created. We maintain a list of all active vertices, *i.e.*, vertices belonging to at least one path. The others are discarded, since no pedestrian will ever pass through them during simulation. Then, a grid is disposed on the scene, its size limited by the bounding rectangle containing all graph vertices. This grid is composed of an array of cells, each containing the link to its neighbor cells, and intrisic parameters used to compute a potential field.

Many of the cells that compose the grid are not needed in the simulation, because they represent zones that are not covered by graph vertices, and thus indicate static obstacles. Moreover, some vertices are not used by any navigation flow, and are not exploited by pedestrians, as illustrated in Figure 4.5. Thus, for each cell, we test whether its center is inside a vertex that composes a path. If not, the cell is deactivated. The main advantage of this preprocess is a drastic reduction of the number of cells in which the potential field computation is necessary. Finally, each cell is linked to its active neighbors only.

Classification of Vertices in ROI

To define a ROI, the user specifies three parameters: a position, a radius, and a level of interest. All vertices whose center is contained within this region are assigned the specified

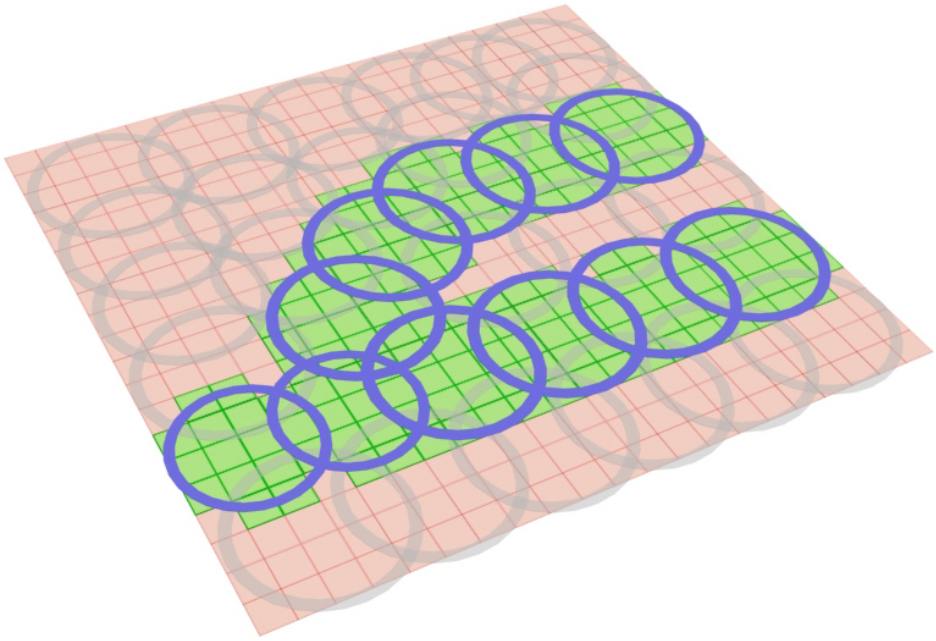


Figure 4.5: A grid is placed on top of the Navigation Graph. Only cells within a vertex that is part of a path stay active (in green).

level. These parameters can be modified at any moment, implying a simple re-classification of vertices that can be efficiently achieved offline, or in real time, during the simulation.

In our practical use of ROI, we create three vertex lists corresponding to our three levels of interest. At runtime, we first automatically detect vertices that are outside the view frustum, and insert them into the list with the lowest level of interest (ROI 2). We then iterate over the remaining vertices, testing whether they are inside ROI 0: since zones of high interest are described by a position and a radius, it is easy to test if the position of each remaining vertex is inside one of these areas. If it is the case, the vertex is classified as of high interest and put in the ROI 0 list. Otherwise, it is put in the remaining list, of low interest (ROI 1).

Potential Field Computation

To accelerate the potential field computation, it is possible to group pedestrians, as suggested by Treuille et al. [Treuille et al., 2006]. It is important here to note that the term "group" can be confusing when used on the one hand for potential field computation, and on the other hand, to describe grouping behaviors within the crowd. In order to avoid ambiguity, in the remaining of this document, the term "set" is used to describe pedestrians with the same goal. We refer to "groups" as small teams of pedestrians walking together. For the set creation, pedestrians in ROI 0 sharing the same navigation flow and direction, *i.e.*, having the same goal, are brought together in one set. Thus, there are two sets for each of these navigation flows (one per direction). Note that it is not required to create one set per path composing the navigation flow, because our sole interest is to group the pedestrians sharing the same goal, whichever the path their are using to reach this goal. Sets need to be re-computed at each time step, to correctly classify pedestrians that change ROI.

Once the sets are created, a potential field is computed for each of them. At the goal, the potential is set to 0, and increased as spread over the grid. Given the potential gradient, each pedestrian is assigned a new waypoint, corresponding to the center of a neighbor cell with the lowest potential. For further details on the potential field computation, see the work of Treuille et al. [Treuille et al., 2006]. Taking advantage of our architecture, we introduce a technique to reduce the computation time of potential fields: actually, the potential field is only required in regions of high interest (ROI 0). These regions only cover part of the scene, and thus, part of the grid. By computing the potential only for the cells located inside ROI 0, we can drastically decrease computation time.

Unfortunately, goals are often outside ROI 0. Such situations make it impossible to initiate the potential field computation, which should imperatively start at the goal cell with a 0 value. To overcome this problem and still limit the places where the potential is computed, for each set, we create subgoals, situated just outside ROI 0, as shown in yellow in Figure 4.6. We use the navigation flow structure to identify them: for every path of every flow leaving ROI 0, the first vertex of another ROI (1 or 2) met in the direction of the goal, is a subgoal. The potential computation is initiated in the central cell of every subgoal, and spread over all cells inside ROI 0. To obtain the same behavior as if the potential was computed all over the grid, we do not initiate the potential of the subgoal cells to 0, but approximate them. For each subgoal cell c inside subgoal vertex v_c, the potential ϕ_c is computed as:

$$\phi_c = C \cdot \sum_{v \in P(v_c)} (v.density + 1) \cdot v.radius \tag{4.1}$$

Where v is a vertex of path $P(v_c)$, starting at v_c and leading to the final goal. The density of v is given by the number of pedestrians in it per square meter. Thus, the contribution to the potential of each vertex v is defined as its radius, weighted by its degree of occupation. To avoid a null contribution from an empty vertex, we always add 1 to the computed density. Constant C is used to weight the sum so that values for ϕ_c are in the same range as if the potential was computed from the goal. Note that vertex v_c may be part of several paths at the same time. In this case, we compute Equation 4.1 for each path, and assign the lowest result to ϕ_c.

Short-Term Avoidance

At this point in the pipeline, pedestrians in both ROI 1 and ROI 0 (where collisions could not be avoided with the potential field), are ruled by a short-term collision avoidance method, detailed in the next Chapter.

Steering

Both Navigation Graph and potential field approaches provide waypoints which pedestrians must reach. A smooth steering algorithm is necessary to obtain a fluid movement toward these points. Reynolds' seek behavior [Reynolds, 1999] has the advantage of producing a believable steering toward a target point in space. We use this model for pedestrians of both ROI 0 and 1, and a linear steering in ROI 2.

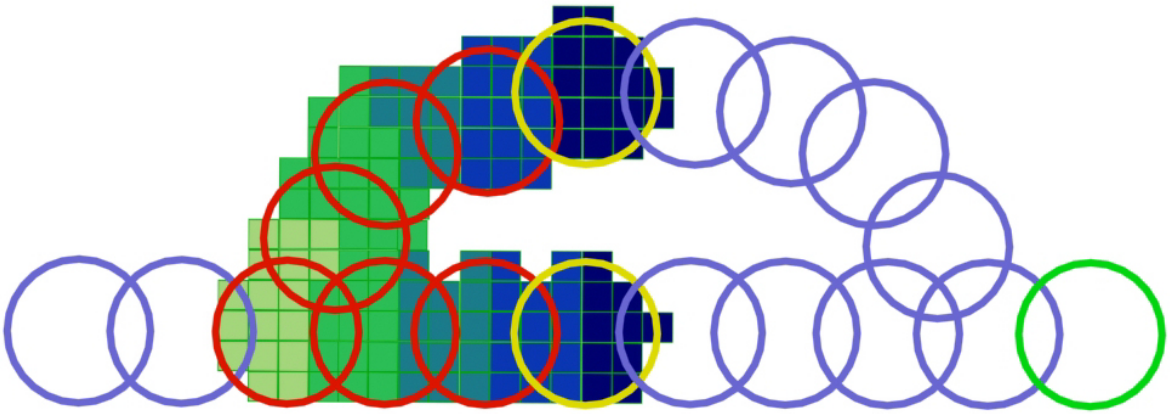


Figure 4.6: Potential is computed for vertices either in ROI 0 (in red) or identified as subgoals (in yellow). The final goal is displayed as a green vertex. Potential starts in the central cells of the subgoals with an approximated value.

Continuity Maintenance

In our architecture, we use two structures based on different spaces: a Navigation Graph composed of vertices and edges, and a grid of cells. This duality brings up two issues when switching from one space to the other, more precisely, when a pedestrian passes from ROI 0 to ROI 1.

The first issue arises when a pedestrian enters the active grid space (ROI 0). Its position is then only updated in the grid, but no longer in the graph. It implies that this character stays registered in the same vertex while progressing in the grid. Thus, its next waypoint on the graph also remains the same. If this issue is not solved, when the pedestrian eventually exits ROI 0, it turns back to meet the graph waypoint it has long since passed. To avoid this, we update the pedestrian position in the graph, even in ROI 0: if a pedestrian enters this region, we keep track of its distance to its next graph waypoint. When the distance is under a given threshold, the pedestrian is registered in the next vertex of its path.

The second issue occurs when two or more paths of the same navigation flow are present in ROI 0. Since path planning in that area is ruled by the potential field, a pedestrian is steered on the path where the potential is the lowest, as in Figure 4.7 (right). However, this path does not necessarily correspond to the one it is registered to in the graph (Figure 4.7 (left)). In the worst case, the pedestrian becomes completely lost when exiting ROI 0: it is within a vertex that does not belong to the path it should follow. To solve this problem, when a pedestrian exits ROI 0, we test whether it still is on the same graph path. If not, we look for a new path using this vertex and register the pedestrian to it.

In summary, there are three levels of interest, ruled either by a potential grid, or a Navigation Graph. Steering is achieved with either the seek behavior of Reynolds, or linearly. Finally, a short-term avoidance algorithm (detailed in the next Chapter) is used in some of the ROI. We present a summary of this classification in Table 4.1.

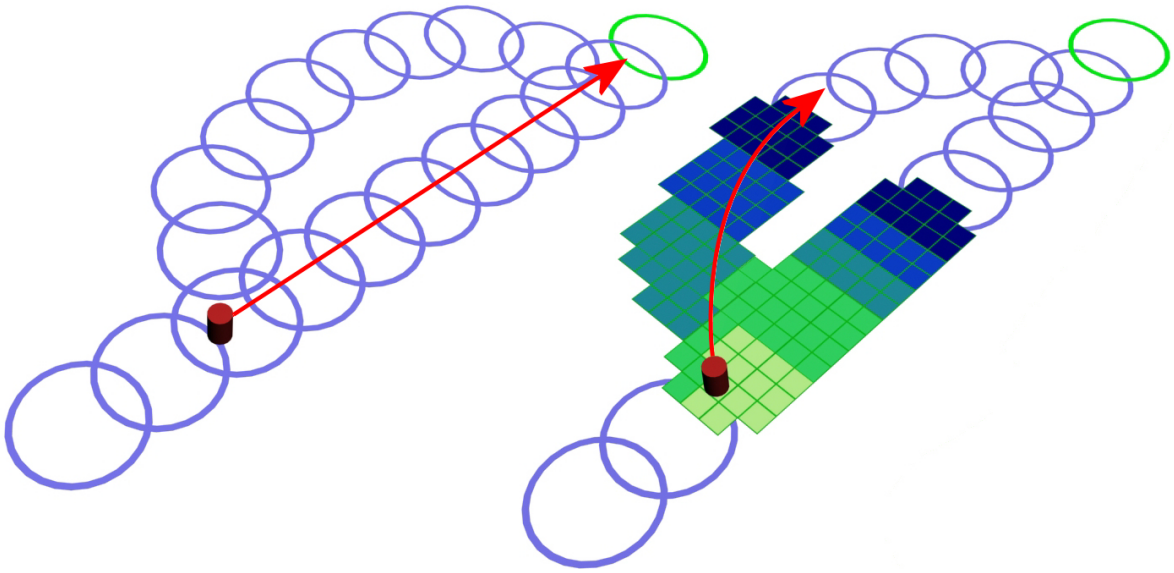


Figure 4.7: *(Left)* In graph space, the path followed by the pedestrian is the right one. *(Right)* In grid space, the potential field is lower on the left path. High potential is represented in light green and low potential in dark blue.

Level of Interest	No (ROI 2)	Low (ROI 1)	High (ROI 0)
Motion Planning	Navigation Graph	Navigation Graph	Potential Grid
Steering	Linear	Seek	Seek
Short-Term	No	Yes	Yes

Table 4.1: Summary of motion planning algorithms used according to the chosen level of interest.

4.4 Performance

Our performance tests have been run with an Athlon64 4000+, with 2 GB RAM and two NVidia 6800 ultra in SLI mode. For these tests, pedestrians are represented with two human templates using several textures, and exploiting color variety techniques [Maïm et al., 2009]. They are rendered as billboards, and use a walk animation, sampled at a frequency of 20 Hz. Note that in the following performance tests, we observe interesting emergent behaviors, *e.g.*, lane formations or panic effects, that make the crowd motion planning more realistic. The tests described below are illustrated in our video available online [Yersin et al., 2007].

We use a city pedestrian area (Figure 4.1) to test the performance of our motion planning architecture, compared with our implementation of the purely potential field-based approach [Treuille et al., 2006]. In this scene, the camera position is fixed at a predefined position. For our tests, we define three regions in the environment. The one with the highest level of interest (ROI 0) has a radius of 15 m, and is static, in the center of the scene. The remaining space inside the view frustum is of low interest (ROI 1), and the other zones are classified in ROI 2. We have tested the efficiency of both approaches with cells of 3×3 m^2, and an increasing number of pedestrians and sets, starting from 2 sets and 200 pedestrians

up to 12 sets, totaling $1,200$ characters. Figure 4.8 shows the results of this comparison. The performance of our approach logically decreases with the increasing number of sets, but much more slowly than with the purely potential field-based approach. We see two reasons for this difference.

Limited Spreading Zone. An approach purely based on potential fields requires to spread the potential over the whole environment to englobe all cells. Our technique only computes the potential field in a limited region of high interest (ROI 0).

Reduced Number of Sets. With our hierarchical approach, it is highly unlikely that all sets of virtual humans happen to pass through the region defined as ROI 0; only a subset of the total number of sets passes in this region, reducing the number of different potential fields to compute. This test has also been performed with the ROI 0 dynamically moving on the city place (demonstrated in a video [Yersin et al., 2007]). Even so, the obtained results remain similar to those shown in Figure 4.8, because, once again, not all sets pass through the zone of high interest.

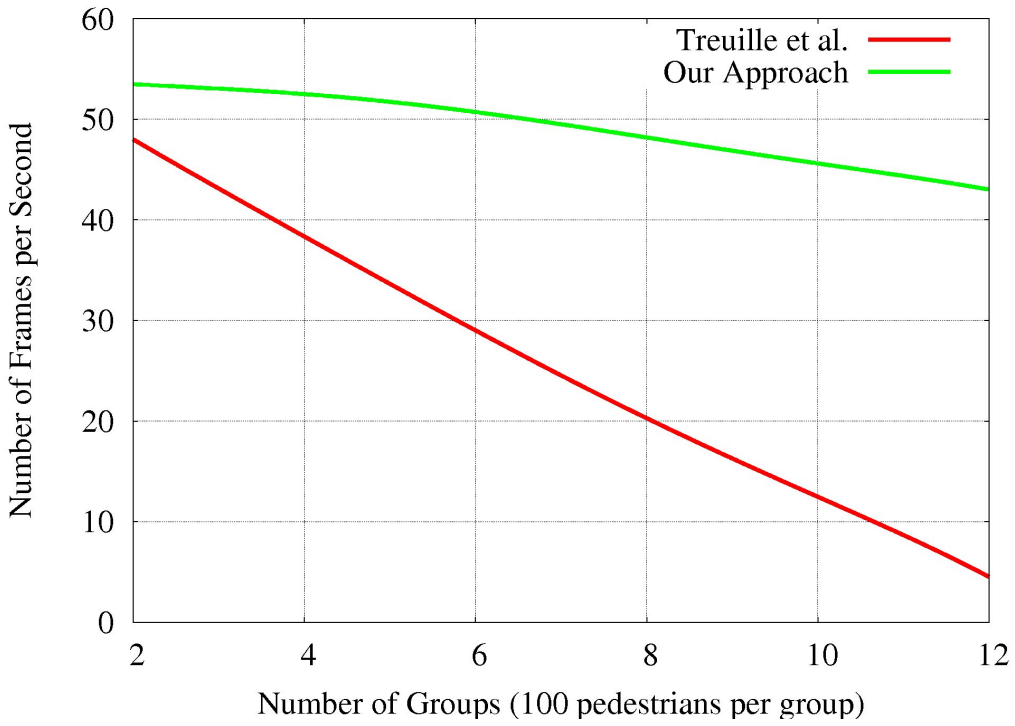


Figure 4.8: Comparison between our approach and our implementation of the approach of Treuille et al. [Treuille et al., 2006] for a varying number of sets, each composed of 100 pedestrians.

Our second test is achieved with $10,000$ pedestrians in a large scene with 12 navigation flows, *i.e.*, 24 sets, spread over the whole environment, as demonstrated in Figure 4.9. For this scenario, the ROI are placed according to the camera position. If the camera moves, the regions are also displaced. The cell size is set to 4×4 m^2, and the performance reaches 20 fps.

Figure 4.9: Pedestrians using our hybrid motion planning architecture to reach their goal and avoid each other in a large landscape of fields.

The third scenario uses the same city pedestrian area as in the first test, extended with several surrounding streets and buildings. There are $5,000$ pedestrians and some cars on the roads, as illustrated in Figure 4.10. Each cell of the grid covers a 3×3 m^2 area. Since the user attention is mainly drawn by the threatening cars, a region of high interest (ROI 0) is set around each of them. Moreover, to make pedestrians flee potential collisions, a high discomfort and speed increase are set in front of the cars, as in the approach of Treuille et al. [Treuille et al., 2006]. As a result, pedestrians close to a car are always in a region of high interest, and thus ruled by a potential field. In front of cars particularly, the pedestrians flee the zone of danger, demonstrating an emergent panic behavior. The remaining visible environment is classified as a region of low interest (ROI 1), so that pedestrians still take care to avoid each other, while the zone outside the view frustum is set as of no interest (ROI 2). The resulting fps varies between 15 and 30, depending on the number of visible cars (1 to 3), and the size of their surrounding ROI 0, (10 to 15 m radius). We note that the use of a single walk animation in our video [Yersin et al., 2007] generates foot sliding artifacts when pedestrians suddenly greatly increase their speed, to avoid a car for instance. This issue is related to the rendering level of detail we used, *i.e.*, billboards, which only have a few sampled animations available, that most of the time do not exactly match the actual moving speed of the characters. A first solution would be to solely use dynamic meshes in our simulation. They have many different locomotion animations at various speeds that can be used in Yaq to minimize foot sliding.However, this approach greatly reduces the potential size of the simulated crowd, and since our goal is to navigate very large crowds, this solution is not appropriate. An interesting lead for future work would be to use an alternate level of detail, *e.g.*, the polypostors of Kavan et al. [Kavan et al., 2008], with which many more

animations could be used than with billboards.

Figure 4.10: A city scene where pedestrians avoid a car surrounded by a ROI 0, where the potential is set as highly elevated.

Finally, we have tested the frame rate evolution with a fixed number of 24 sets and an increasing number of pedestrians. The test has been conducted in a large scene with cells of $3 \times 3\ m^2$, and 1 to 5 distinct ROI 0, each of a $15\ m$ radius. For the remaining of the scene, ROI 2 is not exploited; all vertices are classified as ROI 1. During the test, the scene and pedestrian rendering is deactivated to analyze the sole motion planning cost. The results, in Figure 4.11, show that even with 5 distinct ROI 0, our architecture manages the motion planning of $10,000$ pedestrians at interactive frame-rates (between 10 and $15\ fps$). Note that the increasing number of pedestrians does not influence the potential computation, which is more sensible to the number of sets, as much as the short-term avoidance, which has a complexity in $O(n^2)$ (see next Chapter).

4.5 Discussion

In this Chapter, we have presented the hybrid motion planning architecture integrated into Yaq. This system allows realistic real-time crowd motion planning for thousands of pedestrians. Our approach is scalable; it is possible to divide the scene into regions and exploit different motion planning algorithms according to their level of interest. The architecture flexibility allows the user to determine the performance he wishes and to select and distribute the regions of interest (ROI) accordingly.

Our implementation employs an accurate potential field-based method for pedestrians in ROI 0. A simple and efficient short-term avoidance algorithm is exploited in both ROI 0 and 1, thus ensuring no noticeable transition at region borders (see Chapter 5). Results show

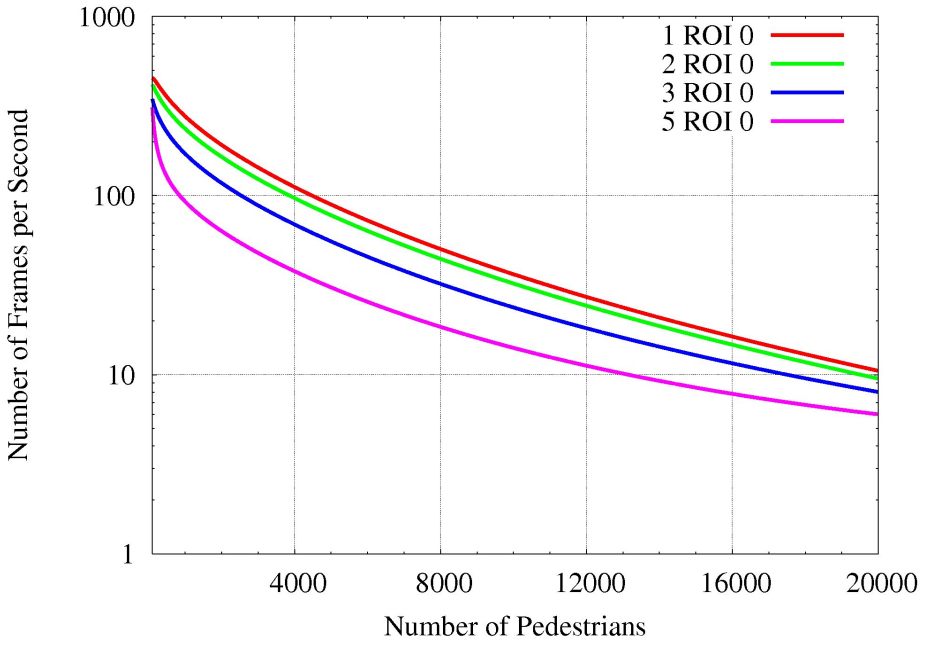


Figure 4.11: Performance for 24 sets with an increasing number of pedestrians (no rendering). 1 to 5 ROI 0 of a 15 m radius each are placed in the scene, while the remaining space is entirely in ROI 1.

that it is possible to simulate over $10,000$ characters in real time, while defining many more sets than in a purely potential field-based approach. Realism is further demonstrated with observable emergent behaviors, like lane formations, and panic escape.

There are some limitations to our architecture. Firstly, in too crowded narrow environments, severe bottlenecks may appear, making the use of our potential field-based approach a waste of computational time. However, it is possible to enforce a low level of interest in these regions, *e.g.*, ruled by the short-term avoidance algorithm introduced in the next Chapter. Another solution that we have recently explored, is to use *crowd patches*, or precomputed motions for crowds, as later detailed in Chapter 7.

A second limitation is our set-based approach: we are constrained to assign general goals for sets of pedestrians. One goal per pedestrian would be too prohibitive for real-time applications. Yet, we note that our architecture is able to handle many more sets than previous potential field-based methods. This is mainly due to our massive reduction of the number of cells in which the potential is actually computed, and implies the possibility to refine the grid for more accurate results.

CHAPTER 5

Short-Term Collision Avoidance

As introduced in Chapter 4, we use a hierarchical architecture in order to perform the motion planning of all individuals in the crowd. This hierarchy is composed of three levels, corresponding to three motion planning algorithms, all used in different regions of the environment, depending on their level of interest. Regions of high interest (ROI 0) are ruled by a potential field-based approach, detailed in the previous Chapter. In the regions of no interest (ROI 2), pedestrians are outside the view frustum and their simulation is limited to a linear steering towards their successive Navigation Graph waypoints. No inter-pedestrian collision is performed there.

In other regions, called of "low" interest, *i.e.*, visible zones that are situated at far distances, and/or where no special event is happening, we need an efficient short-term avoidance algorithm. Our ideal algorithm should be very fast, for it manages a large number of pedestrians, and usable in both ROI 1 and ROI 0 (in cases of very narrow or crowded places, where the potential field approach may fail).

In order to find the best solution for this task, we have tested a first algorithm, based on a grid structure, and consisting in the modification of the next waypoints of pedestrians about to collide. We describe this approach in Section 5.1. Then, in Section 5.2, we present the various improvements we have developed afterwards to obtain better avoidance.

5.1 Original Cell-Based Algorithm

In this Section, we detail our first approach to avoid inter-pedestrian collisions in the short-term. We use a simplified low-level agent-based approach. It can be used to efficiently avoid

local inter-pedestrian collisions in both ROI 0 and 1. Particularly, in ROI 0, it complements the potential field approach, which may fail when the available space is too small and too crowded.

Using an agent-based technique may seem like a bad approach. Indeed, it is well known that agent-based methods, although providing more realistic, individualized trajectories, are more costly than a potential field-based approach, for instance. In our case, the agent-based technique works at a low level (no individual long-term path planning), and on a portion of the crowd only. Moreover, our algorithm is kept simple and thus remains very efficient.

Algorithm 1 details every step of the short-term avoidance process. The first step is to find pedestrians that can potentially collide. Indeed, there is no need to apply this algorithm to safe pedestrians, which can keep following their next waypoint, whether its is based on the Navigation Graph (ROI 1), or on a potential field (ROI 0). To avoid an exhaustive search of high-risk pedestrians, we take advantage of the grid structure covering the whole environment: at runtime, every pedestrian in ROI 0 or 1 is registered in its current grid cell (line 3 of Algorithm 1). This way, we can reduce the search for possible collisions to a small set of neighbor cells. Although this simplification does not cut down the order of complexity in $O(n^2)$, it significantly decreases n, as compared to a brute force approach [Reynolds, 1987]. To keep the algorithm fast, the two steps mentioned above are alternated during simulation (line 1 of Algorithm 1): we first register the pedestrians to their cell at one time step, while the search for potential collisions and their avoidance is achieved at the next step (line 4). Given the low distance covered by a pedestrian in such a short time lapse, the algorithm robustness is guaranteed.

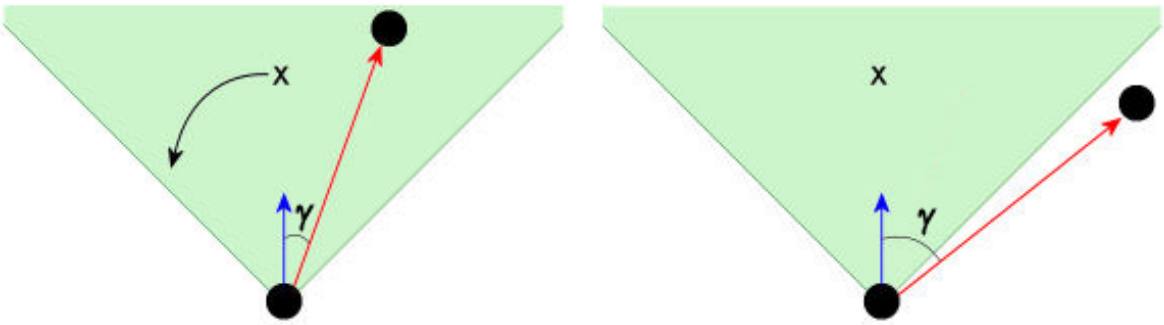


Figure 5.1: Two pedestrians are closer than the security distance α. Angle γ is computed between the first pedestrian's heading vector (in blue) and the two characters' distance vector (in red). *(Left)* γ is in the range $[-\frac{\pi}{4}, \frac{\pi}{4}]$ (green zone) and the first pedestrian's waypoint is thus rotated to avoid collision. *(Right)* The second character is outside the green zone, no avoidance procedure is required.

The avoidance itself is based on two values: a *security* distance α, fixed at 2 m, and an

emergency distance β, at $0.5\ m$. For each pedestrian p in ROI 0 or 1, we start by searching for its neighbor cells in an area of radius α (line 6). Then, for each pedestrian $p_{neighbor}$ contained in a neighbor cell, we test the angle between the heading direction of p and its distance vector to $p_{neighbor}$. If this angle is too small (line 10), the current waypoint of p is rotated away from its neighbor (line 11). An example is shown in Figure 5.1

In the case where a security avoidance is triggered (see left-side of Figure 5.1), and a pedestrian's waypoint has been rotated, we need to make sure that the pedestrian will resume its path after the avoidance. To do so, the rotated waypoint is set back to its original position at the next time step.

An emergency avoidance is sometimes required when the previous approaches (potential fields, security distance) fail. Such emergency cases also happen in reality, when people bump into each other because of a wrong evaluation of distance or speed. If p and $p_{neighbor}$ are at a distance of β or less (line 8), p is gently slided aside its neighbor by a repulsive force.

Algorithm 1: Short-term avoidance algorithm.

Data: set of pedestrians in $\{ROI\ 0 \cup ROI\ 1\}$, set of grid cells, distances α and β
Result: new set of pedestrians in $\{ROI\ 0 \cup ROI\ 1\}$

```
1  if isEven(frameNumber) then
2      for each pedestrian p ∈ {ROI 0 ∪ ROI 1} do
3          register p in its current cell c_p
4  else
5      for each pedestrian p in cell c_p do
6          Set_neighbors = findNeighbors(c_p, α)
7          for each pedestrian p_neighbor in Set_neighbors do
8              if distance(p, p_neighbor) < β then
9                  slide p away from p_neighbor
10             else if angle(p, p_neighbor) ∈ [−π/4, π/4] then
11                 rotateWaypoint(p, p_neighbor)
```

5.2 Improved Algorithm

In this Section, we introduce an improved version of our short-term avoidance algorithm, based on the assumptions that pedestrians mostly want to first maximize their speed and second, to minimize detours.

Note that the method presented in this Section is more generic and adjustable to fit special requirements than the previous one. For instance, if no implementation of a potential field approach is available, this algorithm could be adapted to predict possible collisions up to several meters ahead. Thus, it can perfectly be used as an intermediate or replacement solution for long-term and short-term avoidance methods. Its implementation is summarized

in Algorithm 2 and further detailed below.

Algorithm 2: Improved Short-term avoidance algorithm.

Data: set of pedestrians in $\{ROI\ 0 \cup ROI\ 1\}$, set of grid cells, distance β
Result: updated set of pedestrians in $\{ROI\ 0 \cup ROI\ 1\}$

```
1  if isEven(frameNumber) then
2      for each pedestrian p ∈ {ROI 0 ∪ ROI 1} do
3          register p in its current cell c_p
4  else
5      for each pedestrian p in cell c_p do
6          find n cells ahead to fill f_p, l_p, r_p.
7          // emergency check
8          for each pedestrian p_neighbor in {f_p, l_p, r_p}[i]
9          where i ∈ [0..1] do
10             if distance(p, p_neighbor) < β then
11                 repulse(p, p_neighbor)
12                 removeIntermediateWaypoint(p)
13         // security check
14         for each pedestrian p_neighbor in f_p[i]
15         where i ∈ [1..n] do
16             select intermediate waypoint:
17             l_p[i − 1] or r_p[i − 1]
18             slowDown(p)
```

As shown at lines 1-3 of Algorithm 2, we use the same frequency as before to register pedestrians in their current cell, *i.e.*, the registration in cells of ROI 0 and 1 is achieved at every other time step. Once pedestrians are registered in their current cell, we proceed in three important steps. Firstly, we identify cells ahead of the pedestrian, depending on its speed and direction. Secondly, a security check is performed for pedestrians in a certain range of cells. Finally, an emergency avoidance is performed if the pedestrians are closer than the emergency threshold: β.

Find Cells Ahead. For each pedestrian p in ROI 0 or 1, we identify cells in its forward direction, *i.e.*, in front of it. This is achieved by defining three different arrays of cells (Algorithm 2, line 6): f_p, for cells directly in front of p, l_p to the left of p, and r_p to its right. The front array f_p is filled with n iterations as follows:

$$f_p[i] = getCell(c.pos + i \cdot c.w \cdot p.fwd)\ with\ i \in (0..n). \tag{5.1}$$

Where $c.pos$ is the center position of cell c in which the current pedestrian p is situated. The normalized vector $p.fwd$ represents p's current forward direction, and $c.w$ is the width of a generic cell. We explain later how the parameter n is determined.

In other terms, this computation allows to find a number n of cells that are directly in front of the pedestrian's point of view. Similarly to f_p, the left and right arrays l_p and r_p are filled as follows:

$$l_p[i] = getCell(c.pos + l \cdot c.w + i \cdot c.w \cdot p.fwd) \; with \; i \in (0..n). \tag{5.2}$$

$$r_p[i] = getCell(c.pos - l \cdot c.w + i \cdot c.w \cdot p.fwd) \; with \; i \in (0..n). \tag{5.3}$$

Where l is the normalized vector perpendicular to $p.fwd$ on its left-hand side. The number of iterations n, used to fill the arrays, is arbitrary. In our implementation, we opted for arrays of variable size, depending on the pedestrian's speed. Thus, a slow pedestrian looks less far ahead than a fast one. Note that although there are n iterations for each array, this does not mean that all arrays will contain n cells. Indeed, depending on the forward direction of the pedestrian, it may happen that 2 consecutive computations of Equation 5.1, 5.2, or 5.3, *e.g.*, $f_p[i]$ and $f_p[i+1]$, produce the same resulting cell. We treat this case by checking that the result for a $(i+1)^{th}$ computation is not equal to the result obtained at the i^{th} iteration. If it is the case, the $(i+1)^{th}$ result is not saved, and the corresponding array will contain $(n-1)$ cells.

The resulting arrays form a corridor, which is always pointing in the forward direction of the pedestrian. An example is illustrated in Figure 5.2, where the identified cells ahead of the pedestrian are highlighted. Note that inactive cells are not taken into account when filling the vectors.

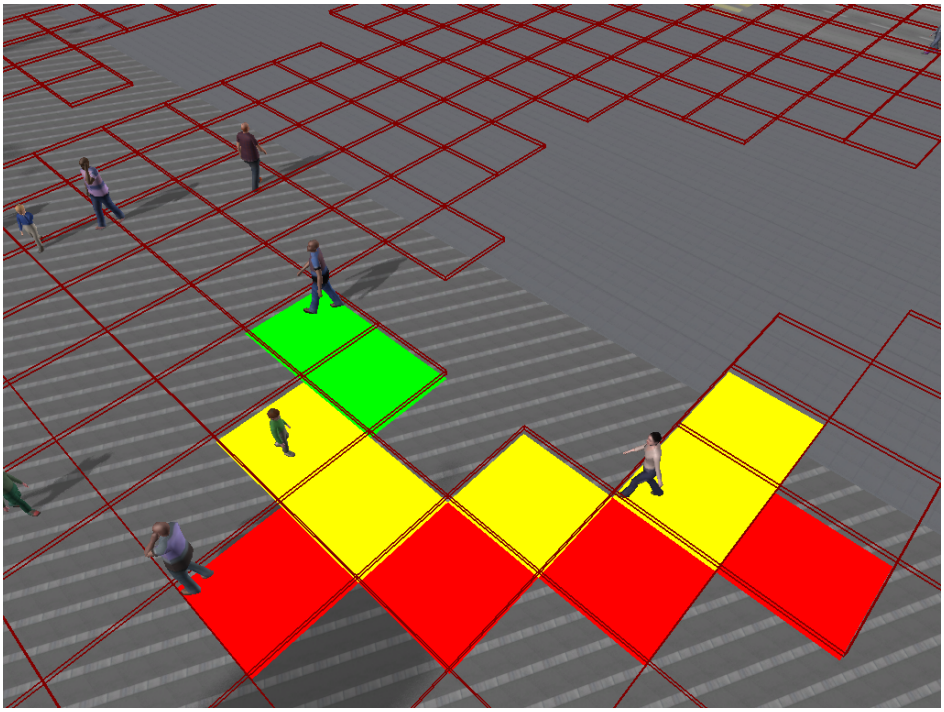


Figure 5.2: A corridor of cells where a child pedestrian (on the left-side) looks for potential collisions. Yellow cells are in its front array, green ones in its left array, and red ones in its right array. The number of cells depends on the speed of the pedestrian. Deactivated cells are not used to fill the arrays.

Security Check. Once the cells to check have been identified, care is taken to prevent collisions. This is achieved as detailed in Algorithm 2: at line 14, we iterate over cells in front of p, starting from the second one, *i.e.*, we iterate over $f_p[i]$ with $i \in (1..n)$, where n is the size of f_p. We consider security checks in $f_p[0]$ unnecessary, because neighbors within this cell are very close to p, and will require an emergency collision avoidance (which is treated below). For each of these front cells, if a neighbor pedestrian $p_{neighbor}$ exists, an intermediate waypoint $p.intwp$, preceding p's original waypoint is introduced to prevent the collision. The intermediate waypoint is either set to the right or to the left of p (lines 16-17), depending on two conditions:

1. Is $p.intwp$ in the same general direction as the original waypoint $p.wp$? We do not want the pedestrian to perform a U-turn. This condition is tested against the two possible directions: $l_p[i-1]$ and $r_p[i-1]$:
   $(p.wp - p) \cdot (p.intwp - p) > 0.0$

2. Is $p_{neighbor}$ rather on p's left side? This condition allows to find the fastest way to avoid $p_{neighbor}$:
   $[(p_{neighbor} - p) \times p.fwd].y < 0.0$

Condition 1 has precedence: if the left alternative returns $true$ while the right one returns $false$, then p will go left towards an intermediate waypoint situated at $l_p[i-1]$'s position. However, if the results of Condition 1 are the same for both the left and right alternatives, then the result of Condition 2 determines the final solution: p will go towards $r_p[i-1]$ if $false$, or towards $l_p[i-1]$ if $true$. Finally, on line 18 of Algorithm 2, p is slowed down to correctly avoid its neighbor. When the collision avoidance is over, the speed will increase again in order to maximize the pedestrian's progress.

Emergency Check. As presented in the previous Section, an emergency avoidance is sometimes required when the other approaches fail, *e.g.*, in over-crowded places. Our approach here is quite similar to the one presented in Algorithm 1, although the cells in which the check is performed are differently selected: to prevent virtual pedestrians from passing through each other, we check in cells close around p that a minimum distance is always respected. This is shown in lines 7 to 12 in Algorithm 2. First of all, note at lines 8 and 9 that the check is only performed over the two first rows of cells for all three arrays, *i.e.*, $f_p[0,1]$, $l_p[0,1]$, and $r_p[0,1]$. Then, for each pedestrian registered in these cells, if the distance between p and $p_{neighbor}$ is smaller than a constant value β (same parameter as in Section 5.1), both characters are moved away from each other, as if bumping against each other. Finally, on line 12, we avoid unnecessary detours by removing the intermediate waypoint if there was one, since it is no longer accurate: both pedestrians have been pushed away, and the situation has changed.

The second Algorithm presented here is more costly than the first one, especially because of the search for cells ahead. However, this approach is still sufficiently efficient to simulate very large crowds in real time, and offers much better results than the previous one. We illustrate a short-term avoidance behavior in Figure 5.3.

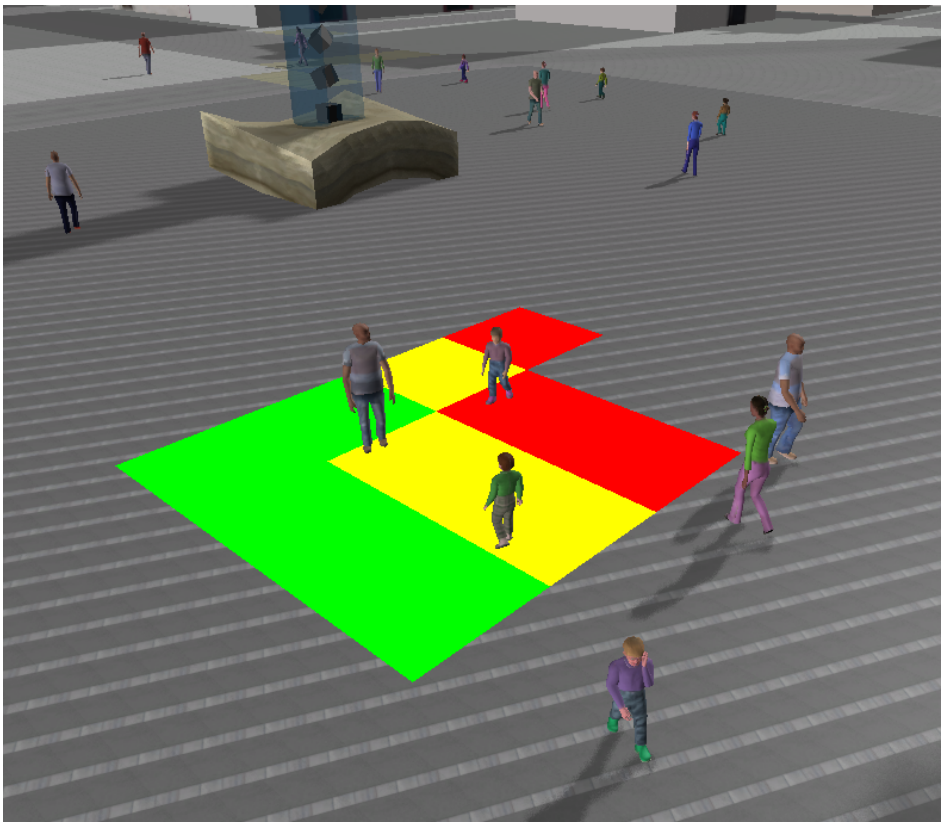


Figure 5.3: The characters present in either the yellow, green, or red cells are detected as a potential threat of collision for the child pedestrian. The short-term avoidance is triggered to take the adequate measures.

CHAPTER 6

Crowd Behavior

When simulating large crowds, it is important to introduce variety in the paths pedestrians are following. This is however not sufficient: if pedestrians are only walking back and forth from point A to point B, the spectator can quickly notice their lack of purpose. On the other hand, the available computational resources to trigger intelligent behaviors are very limited in crowds, for their navigation, animation, and rendering are already very expensive tasks that are absolutely paramount. Our approach to this problem is to find a trade-off that simulates intelligent behaviors, while remaining computationally cheap.

In this Chapter, we detail the various experiments we have achieved to improve pedestrians' behaviors. In Section 6.1, we first present a semantic tool that has been developed to allow a user to give high-level instructions to a crowd that follows his orders. Then, in Section 6.2, we present how the Navigation Graph structure can be used to trigger situation-based behaviors in crowds. Finally, in Section 6.3, we detail a new approach to create groups of 2 to 4 people walking together in an urban environment. The impact of using groups on spectators has been tested and we present the obtained results in the same Section.

6.1 Crowd Motion Planning Tool

To make crowd movements more realistic, a first important step is to identify the main places where many people tend to go, *i.e.*, places where there is a lot of pedestrian traffic. It can be a shopping mall, a park, a circus, etc. Adding meta-information to key places in an environment has been achieved in many different ways. Our approach is to use the Navigation Graph of an environment to hold this meta-information, which is a very advantageous solution: instead of tagging the meshes of an environment, or creating a new dedicated informational

structure, we directly work on the structure that is already present, and which is used for path planning and pedestrian steering.

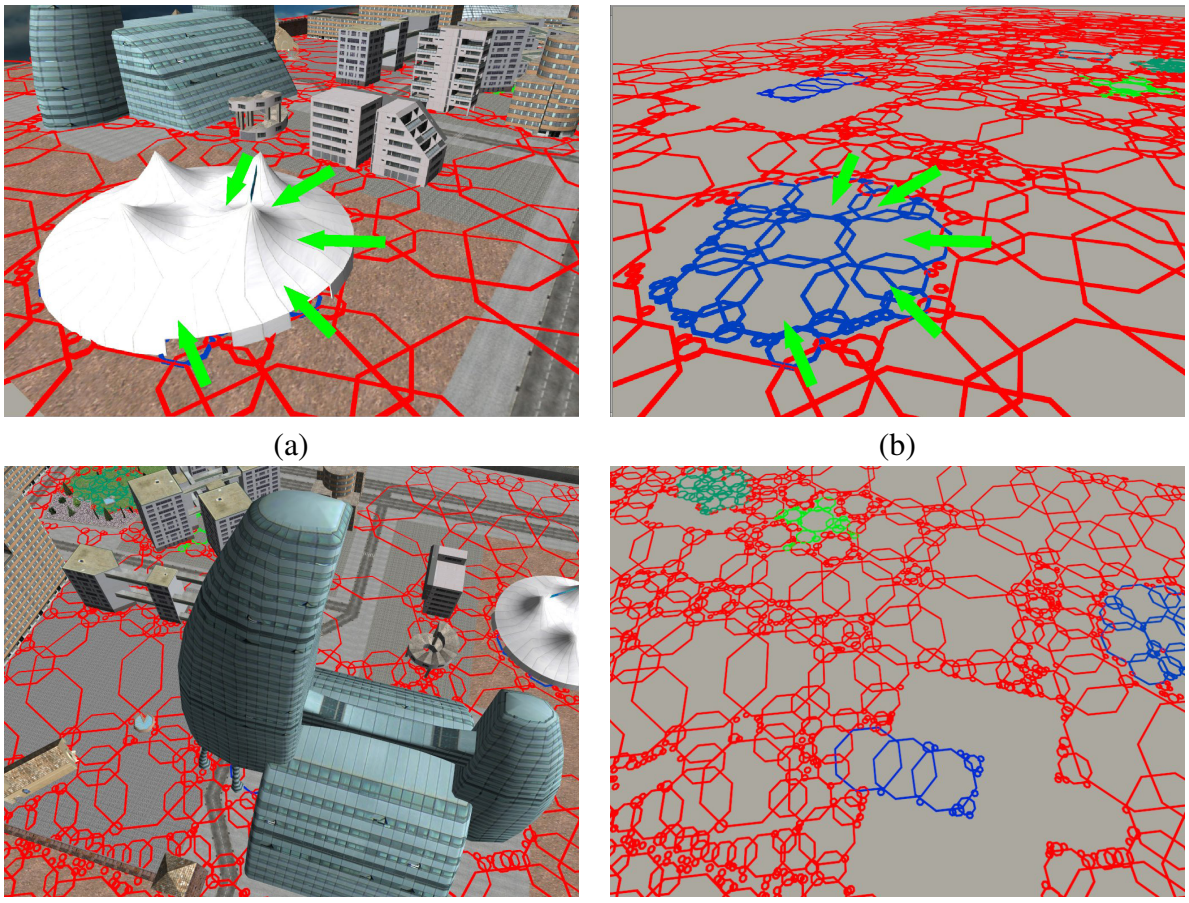


Figure 6.1: *(a)* A circus tent with 5 entrances highlighted with green arrows, and *(b)* its vertices (in blue), connected with the remaining of the graph only at these 5 entrance points. *(c)* A hotel with a single entrance at the left, and *(d)* the corresponding Navigation Graph. There are no vertices where the hotel mesh is present.

6.1.1 A Semantically Augmented Navigation Graph

As previously introduced, the vertices of a Navigation Graph are represented as circular navigable areas. Areas outside all circles are forbidden and thus, going from a vertex to another is possible only if they intersect. Since the Navigation Graph detects where obstacles lie, this structure also allows to delimit the borders of a building for instance. Two examples are illustrated in Figure 6.1, where we show the graph borders near a circus tent and a hotel. Indeed, the graph vertices (or circles) inside a building have no connection with the vertices outside of it (there is an obstacle between them), except where there is an entrance. From this graph structure, it is possible to add semantic data to its vertices, and to steer the crowd based on this information.

We have created a tool to make the semantic assignment job intuitive and fast : the user can simply click to select one or several vertices of the graph and label them with some semantic information, *i.e.*, choose a semantic description from a list: “park”, “hotel”, “station”, etc. The selected vertices then appear in a different color expressing their new meaning. In Figure 6.1 (b) and (d), it is possible to distinguish several places in the graph, where vertices have been tagged with a semantic label. They appear in various shades of blue and green. Our semantic tool also allows to save and load any semantically augmented Navigation Graph by serializing all its vertices and the meta-information they hold.

Once the process of tagging vertices is completed, the semantic information can be used to enhance path finding to a higher behavioral level: with a second tool, the user can easily select one or several virtual humans using the mouse, and give them instructions such as “go to the park”, or “go to the circus”. These instructions have the advantage of not interfering with the path planner’s efficiency and can be treated in real time. For instance, using the Navigation Graph, the high-level goal of reaching a shopping center can be divided into a simple path request from the current position of the selected pedestrian to one of the vertices tagged as a shopping center. This approach is particularly interesting when several shopping centers are present in the same environment: pedestrians instructed to go to a shopping center randomly pick one of them, and go there. The instructed crowd is thus spread over the environment, and provide the user with a feeling of individuality in the decision making process (pedestrians seem to decide on their own to which shop they prefer to go, independently from what the others do).

Figure 6.2: A group of virtual humans has been instructed to go to the courtyard of a building. They arrive to their destination through all the available entry points.

6.1.2 Results and Discussion

Some images of the results we obtained using our set of high-level tools are illustrated in Figures 6.2 and 6.3. A video illustrating the tools is also available online [Yersin et al., 2005]. The idea of augmenting the Navigation Graph with some semantic information has

proven to be very useful. Indeed, the semantic information of each place is directly stored inside the structure of the graph and thus, queries can be answered instantaneously as if the request was at a lower level.

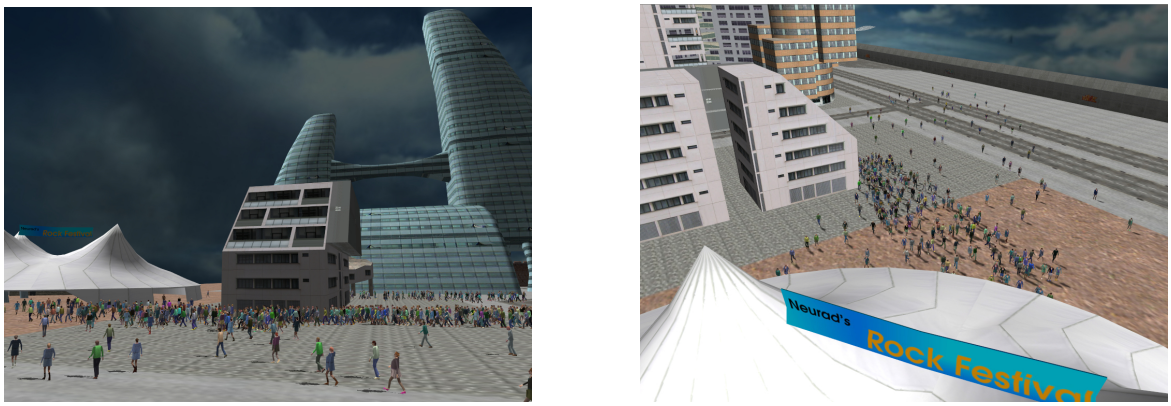


Figure 6.3: A large group of pedestrians have been instructed to go to the circus. They spread on their way, thanks to the variety of paths provided by the Navigation Graph. They also enter the circus tent through different entrances.

There are a few limitations related to this approach, that could be easily overcome in a future work. First of all, a single path request can be answered in real time with the Navigation Graph. However, if the user decides to select a large group of pedestrians to instruct them to go somewhere, there will be as many path requests as there are humans, because each one of them has its own starting position, and a randomly chosen destination. This brute force approach creates some lags, while the paths are computed. However, this problem could be easily overcome with the following solution: first of all, once the annotation of the graph is finished, graph vertices of a same place could be grouped into a higher level structure. Then, in a pre-process, a single set of paths could then be computed from each semantic point to the other ones. Finally, smaller paths (and thus faster to compute), from the current position of a pedestrian to one of the larger paths could be computed at runtime to individualize trajectories and the entrances used.

A second limitation in our basic approach is the random choice of destinations for pedestrians. If two parks are situated at the two ends of a long street, pedestrians close to one of them may choose to move to the one at the other end. This could be easily overcome by testing the distance from the current pedestrian position to the central position of each park.

To simulate crowds of autonomous individuals that select paths on their own, we have chosen to pre-compute a wide variety of paths joining important places of an environment. Thus, a pedestrian reaching an important destination can randomly choose a new path that will lead it towards another major place.

6.2 Situation-Based Behavior

In a recent project of cultural heritage, the Computer Vision Laboratory of ETHZ and the Virtual Reality Laboratory of EPFL have collaborated to revive a district of the ancient city

of Pompeii in a simulation. The ETHZ was responsible for reconstructing the city (Figure 6.4 (left)), while EPFL had to populate its streets with virtual Romans (Figure 6.4 (right)). Within this project, we have worked to improve the behavior of crowds, depending on the places where they were situated. We detail here how this work has been achieved.

Figure 6.4: *(Left)* The Pompeii districts generated with the city engine of the ETHZ Computer Vision Laboratory. *(Right)* Virtual Romans instantiated from seven templates (male and female nobles, plebeians, patricians, and a legionary).

6.2.1 System Overview

The main goal of our work is the real-time simulation of a crowd of virtual Romans exhibiting realistic behaviors in a reconstructed district of Ancient Pompeii. In an offline process, the city is first automatically reconstructed and exported into two different representations:

- A high-resolution model. It is used to generate a Navigation Graph, and is also rendered at runtime (see right-side of Figure 6.5).
- A low-resolution model labeled with semantic data. This model is only composed of annotated building footprints and door / window positions, as illustrated in Figure 6.5 (left). From this annotated geometry, we automatically label the Navigation Graph vertices with the correct actions to be performed by the virtual Romans.

In Table 6.1, we summarize the different semantics that have been embedded in the city geometry and the associated behaviors we want to trigger in the crowd. We here concentrate on this small set of behaviors, because we want to (1) explain the underlying mechanisms in great detail, and (2) emphasize these specific behaviors in the video we have shot, available online [Maïm et al., 2007]. However, Table 6.1 is not exhaustive, and many more behaviors could easily be added.

Figure 6.5: A building contains invisible geometry (checkerboard) which is used to store the semantic information.

geometry semantics	behavior	actions
shop	get amphora	walk inside, get out with amphora.
bakery	get bread	walk inside, get out with bread.
young	rich	only rich people go there.
old	poor	only poor people go there.
door	look at	slow down, look through door.
window	look at	slow down, look through window.
	stop look at	accelerate, stop looking.

Table 6.1: Summary of semantics and associated behaviors.

Shops and Bakeries. There are several buildings in the city model where virtual Romans can freely enter. Some of them are labelled as shops and bakeries, and the characters entering them acquire related accessories, *e.g.*, oil amphoras or bread. These accessories are directly attached to a joint of the virtual character's skeleton, and follow its movements when deformed. We can attach accessories to various joints, depending on their nature. Here, we illustrate this variety by empirically placing the amphoras in the hands or on the heads of the romans, depending on their social rank.

Young and Old Districts. The idea of rich and poor districts is based on *age maps* that were provided by archaeologists taking part in this project. These maps show the age of buildings in the city. Using them, it is possible to see where the districts that have been most recently built are situated. Although this age difference cannot yet be expressed with various textures on the buildings, we have empirically decided to set the rich Roman templates in the most recent districts and the poor people in older areas. From this, virtual characters know where they belong and while most parts of the city are accessible to everybody, some districts are restricted to a certain class of people: rich Romans in most recent districts and slaves in the oldest zones.

Doors and Windows. When the virtual Romans walk in the city, they may pass near an open door or a window. In this case, we make the characters slow down and look at them.

For convenience, and as previously stated, the semantic information associated to the buildings is provided to us in a separate, simplified scene file. The Navigation Graph on the other hand, is created based on a highly detailed scene file.

6.2.2 From Semantic Map to Behaviors

From the semantic labels created in the city generation, a simple file is extracted, containing the data strictly necessary to identify places where specific behaviors should be applied. This file is only composed of labeled building footprints and quads representing windows and doors (see Figure 6.5).

Extract Geometry Semantics

The first step to navigate the virtual Romans in the city is to compute a Navigation Graph, based on the detailed scene geometry file. Then, we can easily find paths between different areas of the city. Note that, at runtime, there is no direct interaction between the Romans and the city geometry. Instead, they interact with the Navigation Graph. In order to make them behave differently given their surroundings, the semantic data contained in the geometry need to be transfered into the Navigation Graph vertices. This is achieved in an offline preprocess, where the behavior data and their location are output into a dedicated script.

Depending on the behavior we want to trigger and its location, graph vertices are identified in different ways. For instance, we would like people to slow down and look at the doors and windows when walking past them. In such a situation, we need to identify the graph vertices that are within a certain distance from the doors and windows. For the shop and bakery buildings, the virtual Romans should be able to find their way to get indoors. In this case, we have to find the graph vertices that are contained inside each building's footprint; outside vertices, even if close to the building, should not be considered. We have designed an algorithm to read the simplified city model and automatically extract its semantic data to identify the influenced Navigation Graph vertices. This automatic process is detailed in Algorithm 3:

Algorithm 3: Semantic Mapping.

Data: set of simple geometries with semantic labels, set of graph vertices with their position and radius.

Result: subset of graph vertices GV where to apply corresponding behavior.

```
1 for each geometry g do
2     extract semantic label s from g.name
3     switch s do
4         GV = identifyGraphVertices(s, g)
5         computeBehaviorParameters(GV, s)
```

In our specific scenario of Pompeii, we have developed two methods to identify the graph vertices (line 4) which will adopt a special behavior. The first one is used for windows and doors, the second one for shops and bakeries:

- If a geometry g of the simple model has a semantic s of type "window" or "door", the method at line 4 will return all the graph vertices situated at a distance closer than a given range r to the geometry g.

- If a geometry g of the simple model has a semantic s of type "shop" or "bakery", it means that g is a building footprint (left of Figure 6.5). We then detect all the graph vertices contained inside g. For this particular function, we can reduce the problem to a 2D approach (working on the OpenGL XZ plane). Based on the work of Bourke [Bourke, 1987], we easily determine when a point p is contained inside the geometry by computing and summing the angles $\widehat{v_i p v_{i+1}}$ between each pair of geometry vertices (v_i, v_{i+1}) and the graph vertex position p. If the sum of these angles equals 2π, then the graph vertex is contained inside the polygon. In our particular Pompeii case, the buildings all have simple footprints which allow to exploit this algorithm.

We illustrate the various steps and the resulting annotated Navigation Graph in Figure 6.6. As shown in Algorithm 3 (line 3), it is possible to treat each semantic label with different methods, depending on their location and effects. At line 5, there is also a second important function $computeBehaviorParameters$ that can be developed differently, depending on the semantic label and the corresponding wished behavior. The purpose of this method is detailed in the next section.

Semantics to Behavior

We know that each semantic label corresponds to a specific behavior. For instance, the window and door semantics trigger a "look at" behavior that makes virtual Romans slow down and look through the window (see Table 6.1). To keep our crowd engine as generic as possible, each graph vertex triggering a special behavior also receives a series of variables used later on for parameterization. For our "look at" example, this means that each graph vertex associated with this behavior should make Romans look through the window or the door. In order to know exactly where Romans have to look, each of these graph vertices also receives a target point, computed as the center of the window/door quad. Thus, for each behavior, it is possible to compute specific parameters during the graph vertex identification in Algorithm 3. More specifically, at line 3, we test the semantic labels to know which parameters need to be computed, and how the graph vertices should be identified.

In our Pompeii scenario, only the "look at" semantics requires extra parameters. Since a graph vertex may be close to several windows or doors at the same time, we make sure to save a set of target points (one per door/window) as its behavior parameters. When a virtual Roman crosses such a graph vertex, it looks at the target point that is closest to him and facing him. For other semantics, no parameter is required. However, the engine has been developed to accept variables in any number.

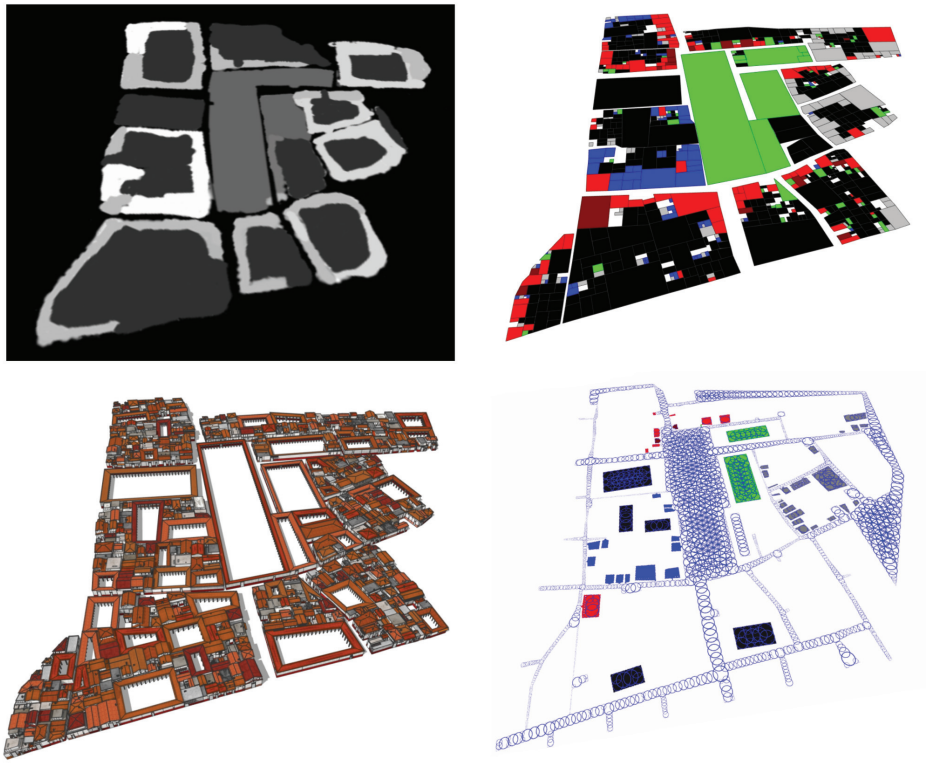


Figure 6.6: Four stages in the offline pre-process: *(top-left)* the manually designed map with five different gray levels for the building types (used to generate the city), *(top-right)* output of the annotated simplified geometry (labeled footprints), *(bottom-left)* the high-resolution model of Pompeii that is used for rendering, and *(bottom-right)* the final Navigation Graph, computed with the high-resolution model and labeled using the low-resolution one.

Finally, this process outputs a script describing which behaviors apply to which vertices and with which parameters. This script is later used at the initialization of the crowd simulation to assign behaviors to graph vertices.

Long Term vs Short Term Behaviors

There are many behaviors that can be triggered when a virtual Roman passes over a graph vertex. Some of them are permanent, *i.e.*, once they are triggered for a Roman, they are kept until the end of the simulation, while others are temporary: once the Roman leaves their area, the behaviors are stopped. For instance, a Roman entering a bakery acquires some bread and will keep it when leaving the bakery until the end of the simulation. However, a Roman passing close to a window will slow down to look through it until it is too far away and then resume a faster walk.

The permanent behaviors are not complex to manage. Once triggered, they modify parameters within the Roman data that will never be set back to their previous value. For temporary behaviors however, it is important to detect when a virtual Roman leaves an area with a specific behavior, and set its modified parameters back to normal values. We have chosen to treat this specific case as follows: when Algorithm 3 is called, for each temporary

behavior b, a new behavior $stopb$ is created as its opposite. To identify the graph vertices that should trigger $stopb$, we iterate over all vertices invoking b, and check the behavior of their neighbors. If a vertex triggering b has a neighbor that is not triggering b, we have found a vertex that borders the b area. We thus flag it with $stopb$. Note that it is possible for a graph vertex to trigger several behaviors. An example of the "look at" behavior is illustrated in Figure 6.7.

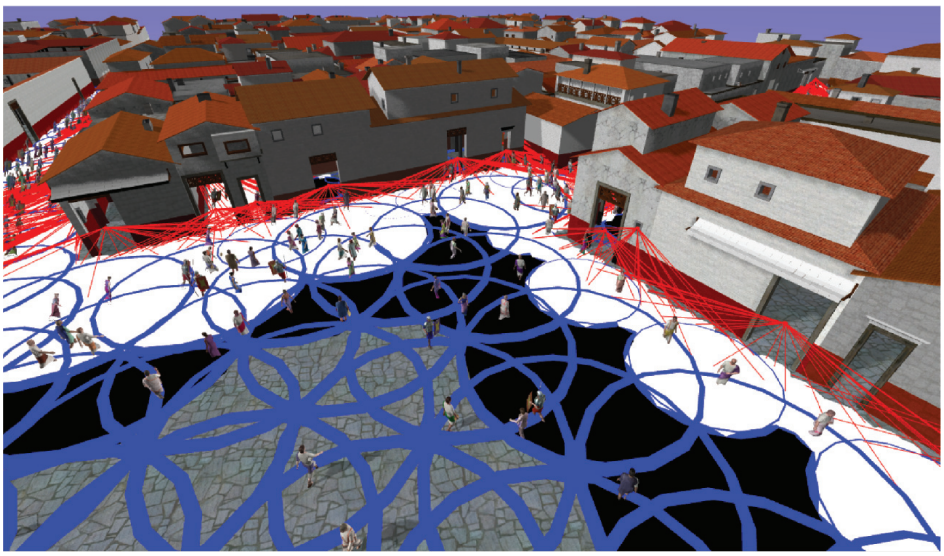


Figure 6.7: Graph vertices are marked with special behaviors: "look at" (in white), and "stop look at" (in black). The target points where the Romans should look are indicated in red.

6.2.3 Results

A video demonstrating the results of our work in reviving the ancient city of Pompeii is available online [Maïm et al., 2007]. First, its streets and buildings have been automatically reconstructed and annotated. Second, they have been populated with crowds of virtual Romans, presenting several different behaviors, and thus offering a realistic and varied simulation (Figures 6.8 and 6.9).

For the crowd, seven human templates have been exploited to create the images as well as the video: male and female nobles, plebeians, patricians and finally, a male legionary. These seven templates are instantiated several hundred times to generate large crowds. To ensure a varied and appealing result, per body part color variety techniques are exploited.

The simulation of the crowd in the city has been achieved with an Intel core duo 2.4 GHz 2 Gb RAM and a Nvidia Geforce 8800 GTX 768 Mb RAM. The crowd engine is mainly implemented in *C++*, but to ease the definition of behavior actions, we use the *Lua* scripting language. One of its main advantages is the fast test/fix cycles while programming behavior functions.

The city part used for the simulation (illustrated in Figure 6.6 (bottom left)) is composed of about 700,000 triangles and 12 Mb of compressed textures. For the crowds, combining the different LOD, it is possible to simulate in this environment 4,000 Romans, *i.e.*, about

Figure 6.8: Crowds of virtual Romans in a street of Ancient Pompeii.

600,000 triangles and 88 Mb of compressed textures, with real-time performance (30 fps in average). Note however that we have reduced the number of Romans to 2,000 in the video sequences, due to the requirements of the real-time capture software.

6.2.4 Conclusion and Future Work

Based on archaeological data, we have presented the different steps of our work to populate the ancient city of Pompeii with virtual Romans. Thanks to the semantic data labeled in the geometry, crowds are able to exhibit particular behaviors relatively to their location in the city. Our results and video show that we are able to simulate several thousands virtual characters in the reconstructed city in real-time.

Figure 6.9: Crowds of virtual Romans simulated in a reconstructed part of Pompeii.

One possible follow-up work would be to make the virtual Romans interact with the

model, *e.g.*, opening doors. This would allow to create more intelligent and varied behaviors for crowds. From an archaeological point of view the simulation would also benefit from a validation of the virtual Roman behavior based on historical data.

6.3 Group behavior

Our most recent effort in improving the crowd behavior has been focused on creating groups (Figure 6.10). In our everyday life, it is rare to observe people in an urban scene walking all by themselves. Indeed, it is easy to notice that pedestrians often evolve in groups of 2 or more. For this reason, we introduce an additional and optional layer to our motion planning architecture. This layer takes care of creating small groups of people, which try to remain close to each other during simulation. We first explain our implementation of groups in Section 6.3.1. Then, in Section 6.3.2, we assess the impact groups have on spectators.

Figure 6.10: Crowds in an urban environment form small groups to improve the simulation realism.

6.3.1 Implementation

Our algorithm is completely based on the update of pedestrian waypoints, similarly to the short-term avoidance approach presented before. This choice is based on several criteria: first, all the motion planning algorithms that are used in our architecture are waypoint-based, which makes our grouping scalable to all levels of interest (ROI). Second, using a potential field to steer pedestrians close to each other is not possible; it would require a potential field

computation for each small group, instead of each set[1], as previously stated. Moreover, it would not solve the scalability issue for other regions of lower interest, *i.e.*, ROI 1 and 2: we need a solution that can be adapted to all regions of various interest. Using a waypoint-based method is simple as well as scalable.

At initialization, groups are created in the numbers and sizes chosen by the user. Typically, they are composed of 2 to 4 pedestrians. For each of these groups, the first member is identified as the leader. The followers are assigned to the same path flow as the leader, they are registered in the same initial vertex, at the same speed. Note that special care is taken to use a different human template for each member of the same group. This is achieved in order to avoid a too important feeling of resemblance between two group members. Each created group is assigned a unique ID, and is stored as a list containing all its members' IDs.

At runtime, several consecutive operations are called. As detailed in Chapter 4, the waypoint of each pedestrian is first updated according to its ROI, if necessary. Then, the short-term avoidance is handled, potentially introducing an intermediate waypoint to prevent collisions. This algorithm remains the same for groups, except that members of a same group do not trigger a security avoidance. Otherwise, it would imply that group members try to avoid each other and remain together at the same time. The emergency check however is kept in order to inhibit inter-penetrations. Once the short-term avoidance with other pedestrians is achieved, groups are handled on an extra layer in 2 steps.

1. Speed Adaptation. For group members to stay close to each other, it is important that they react to the others' behavior. We start by computing the distance between them projected on the leader's forward vector:

$$leader.fwd \cdot distance_{2D}(leader, follower). \tag{6.1}$$

This value provides the distance between the members in the direction of the leader's forward vector. If this distance is larger than a given threshold (in our implementation: $0.5m$), then the follower is too far behind and needs to catch up. The speed of the leader is thus decreased, and the follower's one increased. On the other hand, if the result of Equation 6.1 is smaller than the negative threshold ($-0.5m$), the leader accelerates while the follower slows down. In the case where the distance is within the correct range, care is taken that both speeds are set to a same intermediate value. This is how we ensure that 2 members of a group remain at the same level. In order for their lateral distance to remain small, their waypoints are updated.

2. Waypoint Modification. It is important to note that in ROI 1, our group algorithm only updates the pedestrian's gate waypoints, *i.e.*, waypoints situated on the intersection of two vertices. An intermediate waypoint that may be introduced during a security check is not modified. Thus, if 2 group members get dangerously near other pedestrians, they can separate to prevent a collision. They will get back close to each other by the time they reach

[1]Remember that we call a "group" a small set of 2 to 4 people, while a "set" represents a large amount of virtual humans sharing the same goal, *i.e.*, for which a single potential field is computed.

their next gate waypoint. In ROI 0, the system is slightly different. At all times, the follower's waypoint is updated to be set close to the leader's one (in a neighbor cell). However, if a security avoidance happens, then, similarly to ROI 1, waypoints are modified according to our short-term avoidance algorithm. Thus, group members can become separated. Once the potential collision is avoided, the follower's next waypoint is set back next to the leader's one. In ROI 2, which is outside the view frustum, no short-term avoidance is performed, and members of a same group have the same speed and their waypoints are set close to each other on the vertices' gates.

Note that for groups of 2 members, it is important that both members adapt their speed to stay together. In larger groups however, we do not update the leader's speed according to the other members' positions. We then consider that the leader sets the pace and followers adapt their speeds to catch up.

6.3.2 Assessment

In order to assess our implementation of small group behaviors, we have created eight short movies that have been shown to twelve subjects with no Computer Graphics background. Four of the movies represent a pedestrian city center (Figure 6.10), while the other four were shot in a Halloween theme park (Figure 6.11). For these environments, each of the four movies has a specific scenario: *alone*, where pedestrians only walk by themselves, *small*, where pedestrians are in groups of 2 only, *large*, for larger groups of 3 to 4 people, and *mix* for a varied distribution of size from 1 to 4 group members. Note that an illustrative montage based on the videos used for testing is available online [Yersin et al., 2008].

We asked the twelve subjects to watch the videos and pay particular attention to the pedestrians and their behavior. They were all shown the eight videos of 30 seconds each in different order to avoid a bias in the answers. After each viewing, several questions were asked:

- Indicate what you like / dislike about this scene.
- Who are these people and what are they doing ?
- Do they look sympathetic ?
- Do you feel at ease ?

We intentionally hid the topic of the study and asked questions not directly concerning the groups to get a feeling of subjects' reactions, and to see if they would notice the different scenarii. In the movies, the human templates used were 3 adult women, 2 adult men and 1 boy. Rendering variety techniques were employed to further diversify the crowd [Maïm et al., 2009].

Contrary to our expectations, only a small number of subjects directly noticed that the size of groups was changing with the videos. However, through naive comments, they showed that the feeling was different for changing scenarii. For the city center, in the *alone*

Figure 6.11: Virtual humans are visiting a theme park n groups of 1 to 4 people.

scenario, people found that pedestrians looked stressed out, in a hurry, going to work. Especially concerning the child template, some people wondered where his parents were, or assumed he was going home from school. Several people thought that the child was in fact a person of short stature when no adult was accompanying him. However, when groups were introduced, the subjects talked about an easy-going ambiance, less strict. The few lonely pedestrians in the *mix* scenario were going to work, the groups were visitors, tourists, families, or friends shopping together. In the Halloween theme park, the same type of comments were observed. In the *alone* scenario, people thought that the point of view was close to the exit. People noticing lonely children expressed the same feeling as in the city. In the grouped scenarii, pedestrians were seen as shoppers, families and friends, or visitors. In the *mix* scenario, pedestrians walking alone were interpreted as park staff. These results show that it is possible to completely modify a scene atmosphere by introducing groups of different sizes.

In conclusion, we have shown that it is possible to insert an additional and scalable layer to our motion planning architecture, in order to simulate realistic small groups of pedestrians. Our study shows that such an addition offers a larger variety of interpretations, and a more sympathetic ambiance in the scenes.

Based on our study, we have gathered many interesting leads to improve group behaviors: care should be taken when choosing group members to improve realism, *e.g.*, a businessman wandering with a casually clothed person may raise questions, children should be accompanied with at least one adult, etc. Also, interactions between group members and among different groups should be introduced. For instance, children should occasionally look at their parent, group members should talk to each other, people from different groups could

show interest in their surroundings, maybe recognize other people from time to time and wave.

CHAPTER 7

Crowd Patches

Figure 7.1: *(Left)* A procedurally generated pedestrian street, successfully populated with crowd patches. *(Right)* The same image showing apparent pre-computed trajectories and patch borders.

In this Chapter, we explore a new solution to populate virtual worlds of a potentially infinite size. The hybrid approach we have introduced in Chapter 4 is particularly adequate when we want to simulate a crowd exhibiting intelligent behaviors, and when a direct interaction with the crowd is desired. Unfortunately, crowd simulation remains computationally expensive, even when virtual humans' behavior is limited to navigation. This is especially due to the costly avoidance of collisions. One possibility to lower the computation costs is to pre-compute, store and replay a simulation. Nevertheless, the content is then fixed, and limited in duration and size for obvious memory storage reasons. Moreover, virtual environments (VEs) can reach huge dimensions, like entire cities or even up to the entire Earth, making the pre-computation of an animation for a corresponding virtual population unthinkable.

In this Chapter, we present a new way to address the problem of densely populating any virtual environment, without any assumption or explicit limitation on its dimensions, the possible duration of its exploration, or its population size. An example is illustrated in Figure 7.1.

7.1 Motivations and Contributions

Our motivations in this work are threefold. First, we want to consider environments of any dimensions, from small public areas to entire cities, and potentially more. The problem raised by these environments is the underlying need for large-scale virtual populations. Second, we want to handle environments procedurally generated online and in real-time. It is difficult for current techniques to populate such environments, because crowd engines very often require the environment to be pre-processed, before being able to populate it. This is typically the case of Navigation Graphs. Third, we want to lower the relative cost of motion synthesis and navigation simulation, with respect to the whole VE resource needs, with no visible impact on the resulting motion quality. Our contribution consists of four points:

1. **A technique that pre-computes simulation tasks, stores the resulting motion trajectories, and reuses them on the fly:** obviously, such pre-computations allow us to drastically decrease the online resources needed to animate virtual populations. We break classical crowd simulation limitations on the environment dimensions: instead of pre-computing a global simulation dedicated to the whole environment, we independently pre-compute the simulation of small areas, called *crowd patches*. To create virtual populations, the crowd patches are interconnected to infinity from the spectator's point of view. We also break limitations on the usual durations of pre-computed motions: by adapting our local simulation technique, we provide periodic trajectories that can be replayed seamlessly and endlessly in loops over time.

2. **A user-guided design technique to control the environment content in an easy and efficient way:** our technique is based on a set of *patch templates*, having specific constraints on the patch content, *e.g.*, the type of obstacles in the patch, the human trajectories there, etc. A large variety of different patches can be generated out of a same template, and then be assembled according to designers' directives.

3. **The ability to populate any environment, whether it is already provided, or procedurally modeled in real time:** patches can be pre-computed to populate the empty areas of an existing virtual environment, or generated online with the scene model. In the latter case, some of the patches also contain large obstacles such as the buildings of a virtual city.

4. **A major improvement on the difficult trade-off between simulation quality, memory consumption, and performances:** concerning the quality of the crowd simulation, since it is pre-computed, fine-grained offline techniques for motion synthesis and simulation can be used. Secondly, since the resulting trajectories are periodic, the

memory consumption is no longer directly related to motion duration: potentially infinite motions are achievable from a limited set of patches. Finally, run-time operations are limited to motion replay, resulting in high performances.

This Chapter introduces the crowd patches approach, whose objective is to populate an environment from a set of blocks containing pre-computed motions for virtual humans. By constraining motion trajectories both in space and time at the limits of each block, we can connect them to build large environments with numerous humans. In Section 7.2, we first present the two main components that rule our method, *i.e.*, the *crowd patch* and the *pattern*. We show how to compute patches from patterns in Section 7.3: the most difficult point is to compute periodic motions while respecting continuity conditions on patch borders. The content of patches must fit the designers' requirements on the VE's content; we thus introduce *patch templates* in Section 7.4, with key-ideas for patching environments. Patch templates allow a designer to define the expected content of patches with respect to their location in the environment. They also provide control on the population motion and its distribution in different areas. We demonstrate our approach in Section 7.5 with two examples. Finally, we conclude with a discussion on this new approach, its limitations, and future work directions in Section 7.6.

7.2 Principles and definitions

Our solution is inspired from the work of Lee et al. [Lee et al., 2006]: *motion patches*[1] are building blocks annotated with motions. A virtual human can traverse or act in a block by using (replaying) one of the available motions. Motions are connected between different blocks by sharing common limit conditions. Blocks can be assembled to create new environments. We illustrate an environment constructed with motion patches in Figure 7.2. One limitation of this approach is that no interaction between several characters evolving in a same patch is possible. We extend the concept of motion patches by creating *crowd patches*, where several humans can move and interact simultaneously. The issue then grows from a geometrical problem to a geometrical *and temporal* one. A virtual population is created by assembling such blocks. As interactions are solved once for each block, and further reused during simulation, we save precious computation time. Assembling and disassembling blocks can be achieved at run-time, breaking the limitations on environment size: the virtual population is only generated in front of the spectator's point of view.

The overall objective of our approach is to populate virtual environments by composing small pieces of pre-computed animations. Such pieces, called *crowd patches*, can contain moving pedestrians, humans executing other tasks, animals, or objects. Inside a patch, animations are computed so that they are cyclic over a constant period, and can be seamlessly repeated. This allows animated content to appear in endless motion. Animated objects may cross the limits of a patch and move to a neighbor patch. The trajectories of such objects must then meet common limit conditions to allow going from one patch to another adjacent one. These limit conditions are defined using the concept of *patterns*. In the following Sections,

[1] For clarity purpose, care should be taken not to confuse *motion* patches and *crowd* patches.

Figure 7.2: An illustration of motion patches [Lee et al., 2006]: patches are building blocks annotated with captured motions that can be combined to generate the movements of a virtual character. On the top-right of the image, we can see the original jungle jym in which the motions have been captured.

we define more precisely *patches* and *patterns*.

7.2.1 Patches

Patches are geometrical areas with convex polygonal shapes. They may contain static and dynamic objects. Static objects are simple obstacles which geometry is fully contained inside the patch. Larger obstacles, such as buildings, are handled differently (explained in a following Section). Dynamic objects are animated: their position and internal configuration is moving in time according to a trajectory $\tau(t)$. As previously introduced, we want all the dynamic objects of patches to have a periodic motion in order to be seamlessly repeated in time. We note π this period and we define the patch periodicity condition for each dynamic object:

$$\tau(0) = \tau(\pi) \tag{7.1}$$

Two categories of dynamic objects may be distinguished: *endogenous* and *exogenous* objects. The trajectory of *endogenous* objects always remains inside the geometrical limits of the patch for the whole period π. An example of endogenous object is displayed in Fig-

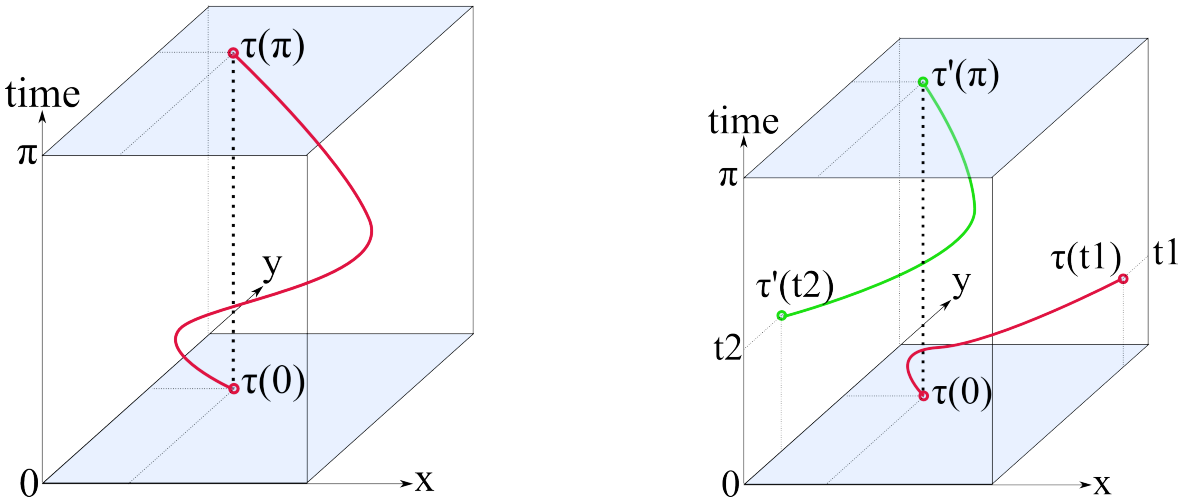


Figure 7.3: Example of square patches. The vertical axis represents time. *(Left)* The patch contains an endogenous point that performs a periodic animation: its start and end positions $\tau(0)$ and $\tau(\pi)$ are the same. *(Right)* The patch shows an exogenous point that performs a periodic motion, even if its trajectory leaves the patch at time t_1 to reappear at t_2 on the other side of the patch.

ure 7.3 (left) with a point moving in a square patch. The point's trajectory is fully contained in the patch and respects the periodicity condition 7.1. If the animation is looped with a period π, the point appears to be moving endlessly inside the patch. Note that static objects can be considered as endogenous objects, with no animation.

Exogenous objects have a trajectory $\tau(t)$ that goes out of the patch borders at some time, and thus, does not meet the periodicity condition 7.1. In order to ensure the respect of this condition, we impose the presence of another instance of the same exogenous object whose trajectory is $\tau'(t)$ (Figure 7.3 (right)). As the two objects are of the same type, *i.e.*, they have an identical kinematics model, their trajectories can be directly compared. Different cases are then to be distinguished:

- trajectory $\tau(t)$ is defined for $t : 0 \rightarrow T$ with $0 < T < \pi$: we impose the patch to contain another instance of the exogenous object with a trajectory $\tau'(t)$ defined for $t : T' \rightarrow \pi$ with $0 < T' < \pi$, and with condition $\tau(0) = \tau'(\pi)$.
- trajectory $\tau(t)$ is defined for $t : T \rightarrow \pi$ with $0 < T < \pi$: we impose the patch to contain another instance of the exogenous object with a trajectory $\tau'(t)$ defined for $t : 0 \rightarrow T'$ with $0 < T' < \pi$, and with condition $\tau'(0) = \tau(\pi)$
- trajectory $\tau(t)$ is defined for $t : T_1 \rightarrow T_2$ with $0 < T_1 < T_2 < \pi$. Then, condition 7.1 is implicitly respected, the presence of another instance of the exogenous object is not required.

In the example of Figure 7.3 (right), the point trajectory $\tau(t)$ is defined inside the patch for $t : 0 \rightarrow t_1 < \pi$. This trajectory is associated to another instance of a moving point $\tau'(t)$ so that $\tau'(\pi) = \tau(0)$. Note that $t_1 \neq t_2$ and no order relationship is required: the two instances of the moving point can be simultaneously outside or inside the patch.

7.2.2 Patterns

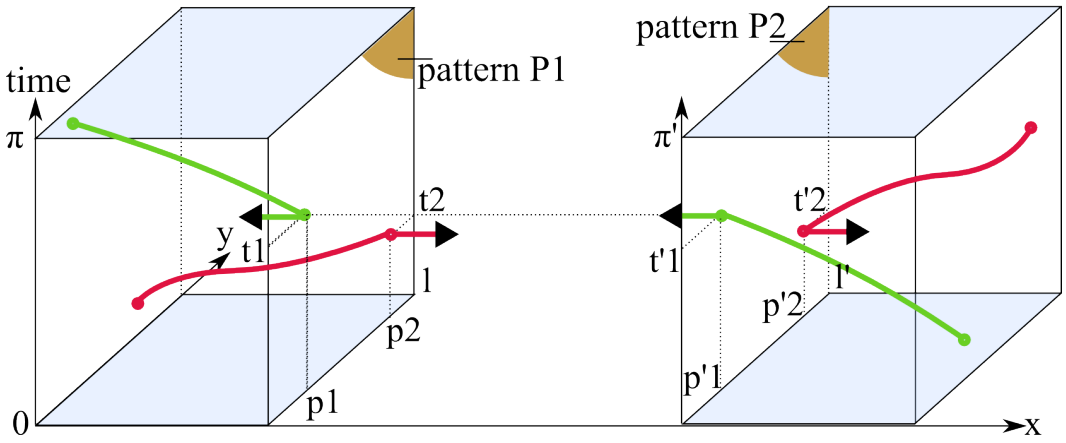


Figure 7.4: Example of 2 square patches sharing patterns with mirrored conditions and thus, connectible. Once connected, the point following the green trajectory passes from the right patch to the left one seamlessly, while the point with the red trajectory moves from left to right.

In the previous Section, we define exogenous objects: their trajectory exits the geometrical limits of a patch. The overall role of patterns is to register the limit conditions of exogenous object trajectories in order to enable the connection of patches. A pattern is defined for each face of a polygonal patch. As a result, patterns fully delimit patches. They are two-dimensional: a space dimension with length l, and a time dimension with duration (period) π. Patterns identify limit conditions for exogenous object trajectories. These conditions are either an input point I, or an output point O, at a specific position on the patch face $p \in [0, l]$, and at given time $t \in [0, \pi]$. Thus, a pattern is fully defined from its dimension l, its duration π and a set of inputs and outputs: $P = \{l, \pi, I_i[p_i, t_i], 0_j[p_j, t_j]\}$.

We build environments and their population by assembling patches. Thus, two adjacent patches have at least one common face. Two adjacent patches also share identical limit conditions for exogenous objects' trajectories. Indeed, when an exogenous object goes from one patch to an adjacent one, it first follows the trajectory contained by the first patch, and then switches to the one described by the second patch. These two trajectories have to be at least continuous C^0 in order to allow a seamless transition from the first patch to the second one. The patterns standing between the two adjacent patches allow to share these limit conditions. Let us consider the case of two adjacent patches with one common face as illustrated in Figure 7.4. In our example, the two patches each contain two exogenous objects going through their common face. The common face is delimited by pattern P_1 for the first patch on the left, and by pattern P_2 for the second patch on the right. we have: $P_1 = \{\pi, l, I[p_1, t_1], 0[p_2, t_2]\}$ and $P_2 = \{\pi', l', I[p'_2, t'_2], 0[p'_1, t'_1]\}$. In order to satisfy C^0 continuity for exogenous objects' trajectories we must ensure that:

$$\pi = \pi',\ l = l',\ p_1 = p'_1,\ t_1 = t'_1,\ p_2 = p'_2,\ t_2 = t'_2 \tag{7.2}$$

We then say that P_1 is the mirror pattern of P_2. As a summary, mirrored patterns have the same duration, length, and number of mirrored inputs and outputs.

7.3 Creating Patches

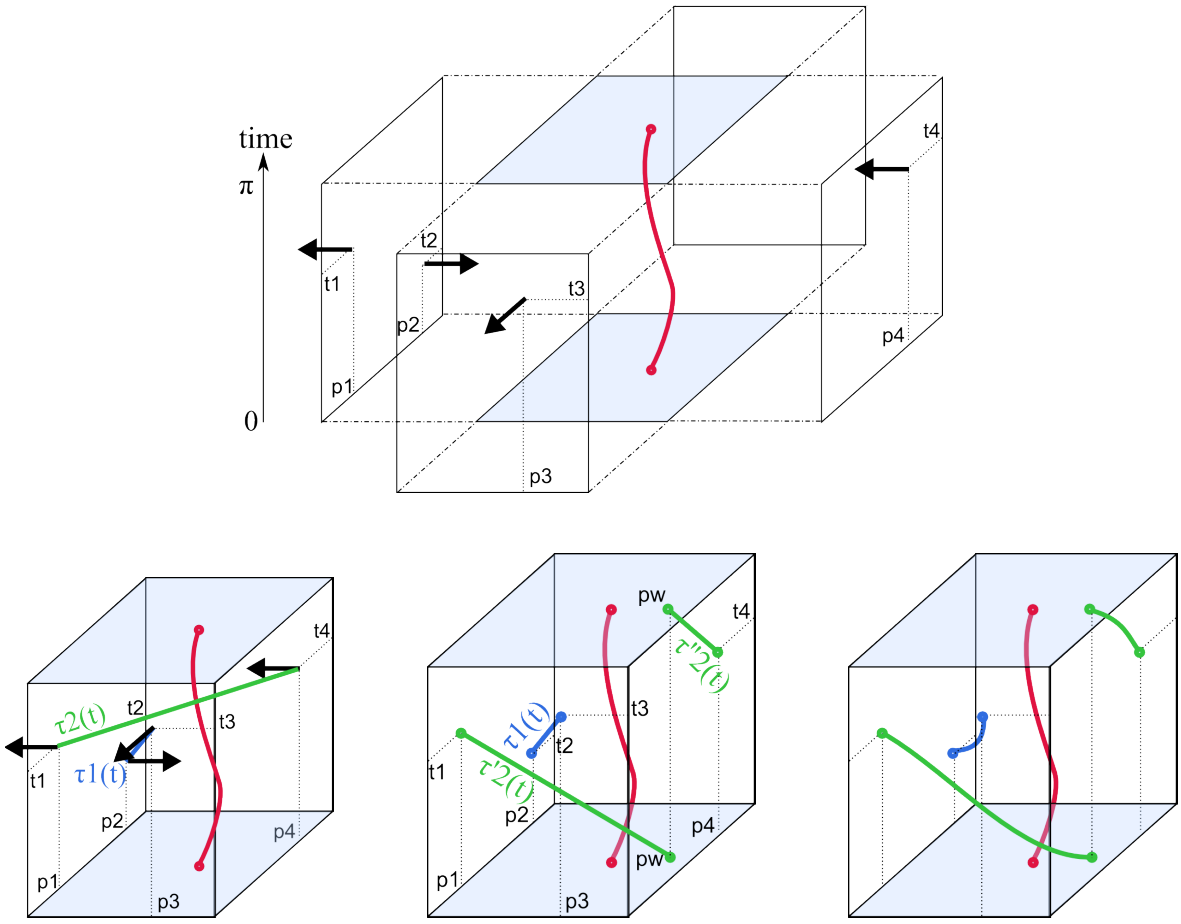


Figure 7.5: Computation of exogenous trajectories for walking humans in a square patch. *(Top)* the patterns composing the patch count 2 input and 2 output points. The patch also contains 1 endogenous pre-defined trajectory. *(Bottom-left)* input and output points from different patterns are randomly connected. The green trajectory is infeasible, since it goes back in time. *(Bottom-center)* The green trajectory is split into 2 trajectories at position p_w, meaning that 2 humans will use it over one period. *(Bottom-right)* The exogenous trajectories are updated with a particle-based technique to avoid collisions between objects.

In this Section, we describe the complete process for creating one patch. The proposed method is decomposed into 3 successive steps: first, create the patch geometry and define exogenous trajectories' limit conditions by assembling patterns, second, allow insertion of static and endogenous objects, and finally, automatically compute exogenous trajectories. For an easy understanding, we provide the example of a simple square patch, illustrated in Figure 7.5, that would typically populate a virtual shopping street.

7.3.1 Pattern Assembly

Our method for creating patches first starts by defining their shape. From the previous definitions, one understands that patches are delimited by patterns: each face of the convex polygonal shape of patches is defined by one pattern. As a result, patches are created by assembling patterns. Two conditions have to be respected when assembling patterns to create one patch:

1. all patterns must have a common period π which will be the patch's period too.
2. the total number of inputs defined by the set of patterns composing the patch has to be equal to the number of outputs. Indeed, each exogenous object entering the patch must, sooner or later, leave it. Obviously, if this condition is not respected, it is impossible to respect periodicity condition 7.1.

An example of patch assembly from 4 patterns is displayed on top of Figure 7.5: all patterns have the same duration π and length l, and are combined to delimit a square patch. Note that having identical lengths for patterns is not required, and patches may be composed by any number of patterns, 3 at least, with no theoretical maximum. These 4 patterns also define 4 limit conditions: 2 inputs of exogenous objects $I_1[t_2, p_2]$ and $I_2[t_4, p_4]$, as well as 2 outputs $O_1[t_1, p_1]$ and $O_2[t_3, p_3]$.

7.3.2 Static Objects and Endogenous trajectories

After defining its shape, we either define ourselves, or get the online information on all the static and endogenous objects contained in the patch. Static objects are located relatively to the patch so that their geometry is fully contained in the patch. Endogenous objects are animated so that they have a cyclic motion with period π. In the example provided in Figure 7.5 (top), we can see that one endogenous object is defined for the patch, with a red trajectory.

In the typical example of a patch built for populating a virtual shopping street, static obstacles are: trash cans, trees, public benches, streetlights, signboards, billboards, etc. Endogenous objects can be humans talking, sitting, phoning, watching shop windows, etc. They can also represent animated objects or animals, such as a merry-go-round, or dogs. Finally, note that once defined, static and endogenous objects are no longer modified. We respectively consider them as static and moving obstacles in the following step, that consists in automatically computing exogenous trajectories.

7.3.3 Exogenous trajectories: case of walking humans

Computing trajectories for exogenous objects is more complex than the previous steps: the limit conditions defined by patterns must be respected, and collision avoidance with static objects, endogenous objects (whose animations are now fixed), and other exogenous objects moving in the patch, must be achieved. We propose a method to automatically compute exogenous trajectories of walking humans. We consider the evolution of their global position

only; a trajectory is thus modeled as a moving 2D point. Computing exogenous trajectories for walking humans is done using a 3-step technique. Each step is illustrated in the bottom of Figure 7.5.

Initialization of Trajectories. We initialize exogenous trajectories by connecting each input to an output, as defined by patterns. We thus obtain initial limit conditions for the trajectories, and count as many trajectories as the total number of inputs (or outputs) defined for the patch. Inputs and outputs are connected at random, with the sole constraint that we avoid connecting inputs and outputs of the same pattern. This way, walking humans pass through the patch rather than head back. At first, linear trajectories are computed between a connected input $I[p_i, t_i]$ and output $O[p_o, t_o]$. We obtain for each connected pair:

$$\tau(t) = p_i + p_i\vec{p}_o.\frac{t}{t_o - t_i} \tag{7.3}$$

In the example of Figure 7.5 (left), 2 exogenous linear trajectories are defined: $\tau_1 : I[p_2, t_2] \rightarrow O[p_3, t_3]$ in blue, and $\tau_2 : I[p_4, t_4] \rightarrow O[p_1, t_1]$ in green.

Velocity Adaptation. Input and output points being arbitrarily connected, the resulting trajectories may be infeasible for walking humans: the average speed $\tilde{v} = \| p_i\vec{p}_o \| .(t_o - t_i)^{-1}$ along the trajectory may be too high, or even negative, if $t_i > t_o$, which would be nonsense. The key-idea to address this problem is to split an unacceptable trajectory τ into two trajectories τ' and τ'' passing by a new way-point p_w as follows:

$$p_w = p_i + (p_i\vec{p}_o.\frac{\pi - t_i}{t_o + \pi - t_i}) \tag{7.4}$$

$$\tau \begin{cases} \tau' : [p_i, t_i] \rightarrow [p_w, \pi] \\ \tau'' : [p_w, 0] \rightarrow [p_o, t_o] \end{cases} \tag{7.5}$$

In other words, instead of having one pedestrian joining the considered input and output points, we obtain two humans: one going from the input point to p_w, while the second one heads from p_w towards the output point. These two humans have an identical position p_w at different times: at $t = \pi$ for the first human, and at $t = 0$ for the second one, thus ensuring periodicity condition 7.1. The new average speeds of pedestrians are respectively $\tilde{v'} = \| p_i\vec{p}_w \| .(\pi - t_i)^{-1}$ and $\tilde{v''} = \| p_w\vec{p}_o \| .(t_o)^{-1}$. In the case where these speeds are still too high for humans, the process can be reiterated until they fall into an acceptable range (typically $[0, 2]\ m.s^{-1}$). In the example of Figure 7.5 (center), τ_2 is defined with $t_2 > t_3$ and is split into two trajectories.

Collision Avoidance. Trajectories may result in collisions between objects, static or dynamic. We consider static and endogenous objects unmodifiable, and refine exogenous object motions to avoid collisions. To reach this objective, we use a particle-based technique that, at each time step, steers exogenous objects from a set of forces:

- objects track an attraction point moving along their initial linear trajectory $\tau(t)$

- objects are repulsed by all other objects in their vicinity.
- in order to avoid dead-lock situations, attractive and repulsive forces are balanced: the farther an object is from its attraction point and the closer t is to t_o, the more paramount attraction forces are, as compared to repulsive ones.

Particle-based simulations may result in jerky motions. We thus smooth the resulting trajectories before storing them. In Figure 7.5 (right), we show the trajectories obtained after the collision avoidance and smoothing steps. Note that our approach allows to update exogenous trajectories with any method. Especially if patches are pre-computed, it is possible to use more fine-grained approaches to solve collisions. We here employ a particle-based approach for its simplicity and rapidity.

In conclusion, we described a technique for computing patches. The patch is first geometrically defined by assembling a set of patterns, which also provide a set of input and output points for exogenous objects. Then, static and endogenous objects are added to the patch. Finally, exogenous object trajectories are computed for walking humans, so that they respect limit conditions imposed by patterns, while avoiding object collisions.

7.4 Creating Worlds

In this Section we first describe how to assemble patches to create virtual worlds, whether they are pre-computed, or generated online. Then, we present the *patch templates* and *pattern types*, which help designers to efficiently control the environment content.

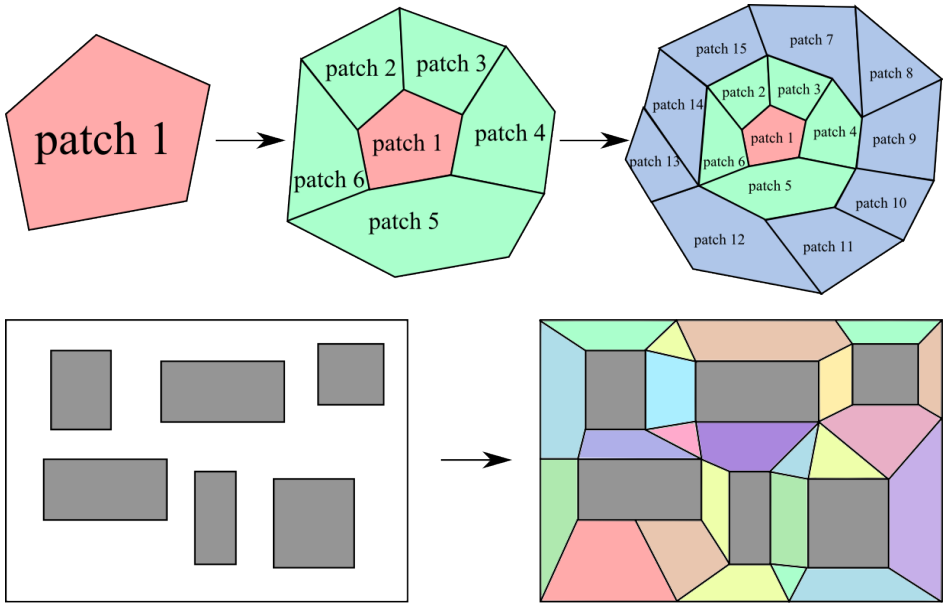


Figure 7.6: Creating Worlds using *(top)* a bottom-up technique, *e.g.*, procedural generation, or *(bottom)* a top-down approach, starting from a geometrical model of the environment.

7.4.1 Patch Assembly

We distinguish two major ways of using crowd patches for creating and/or populating virtual environments: a bottom-up, or a top-down technique.

The *bottom-up* approach starts from an empty scene. A first patch is created, and then, iteratively, new adjacent patches are progressively added. This process is illustrated in Figure 7.6 (top). Patterns of a newly created patch that are adjacent to the previously inserted patches are constrained: they are directly defined by mirroring patterns of the already existing patches. Then, the patch polygon is closed by other patterns, so that they satisfy the patch assembly conditions stated in Section 7.3.1. Note how patch assembly progressively grows in an eccentric manner.

A *top-down* approach starts from a geometrical model of the environment. The obstacle-free parts of the environment are decomposed into polygonal cells that are used as a blueprint for creating patches. This technique perfectly fits the case of virtual cities where large buildings are already present, and streets have to be populated by patches. Computational Geometry provides many techniques to decompose a surface into elementary cells: some provide exact decomposition, *e.g.*, triangulation, whereas others are approximated, *e.g.*, using 2D grids. It is possible to use any of them as long as convex cells are obtained. We provide an example of a hand-made decomposition in Figure 7.6 (bottom).

Both techniques can be used to define the geometrical shapes of patterns. It is possible for both of them to pre-compute all patches in order to create or populate a given environment. It is also possible to generate patches on-the-fly, from the spectator's point of view: only visible parts are patched. The next step is to define the patches' content, which is dependent on the desired appearance of a given area, and is thus a design problem. We describe patch templates and pattern types in the next Sections.

7.4.2 Patch Templates

The content of a patch is dependent on its precise location in the environment, and on the considered environment itself. Let us take the example of a shopping pedestrian zone: we want some people to walk there, some people looking at shop-windows, people walking their dog, people sitting on public benches, etc. The corresponding patches should then contain static obstacles such as benches, trees or flowerpots. Endogenous objects can be people standing in front of shop windows, talking together, sitting on benches, while exogenous objects are simply walking humans.

Designers want to have a certain control over the environment content, but accurately defining the objects in each patch is too time consuming. A solution is then to automatically generate patches from given templates. A *patch template* regroups patches meeting a set of constraints on their objects and patterns. In order to associate geographic zones with desired templates, designers provide a *template map*. The map defines which template to use at any point of the environment. We can also address the specific case of environments that are modeled online in real-time, using procedural methods. However, they need adaptation to generate the template map in parallel with the geometry.

When a patch of a given template is created, some static and endogenous objects are

randomly selected among the allowed set to compose its content. Designers also need to control the flow of walking humans going through patches, which implicitly defines the overall distribution of people in an environment. This is possible by constraining the pattern types to use. We detail in the next Section what a pattern type is.

7.4.3 Pattern Types

As mentioned above, controlling walking humans in our method is achieved by defining specific pattern types. These types allow to choose the specific distribution of input and output points in space, or time. We give here some examples of usage for specific distributions.

Empty Patterns. Environments are likely to contain large obstacles, such as buildings (as seen in Figure 7.6 (bottom)). In order to avoid collisions between exogenous walking humans and these obstacles, patterns delimiting them are empty of inputs or outputs. Indeed, a patch with an output at an obstacle border would result in an irremediable collision, whilst an input would result in a sudden apparition of a human through the wall.

One-way Patterns. One-way patterns are exclusively composed of inputs or outputs. They allow to simulate, for example, one-way flows of walking humans in corresponding patches.

Specific I/O Space Distribution. It is possible to limit the range of input and output positions in space on a given pattern. This allows to simulate the presence of a narrow passage, such as a door, between two patches for instance, or to simulate crowded streets of walking humans forming lanes.

Specific I/O Time Distribution. Users may want pedestrians to enter or exit patches irregularly in time. For instance, at zebra crossings, pedestrians leave sidewalks when the walk sign is green, and have to reach the opposite sidewalk before the light switches to red. Such temporal conditions can be respected by adequately distributing patterns' inputs and outputs in time.

In conclusion, we have introduced the concepts of patch templates and pattern types. A template allows to control the environment content by inserting various objects, static or moving, according to the designers' will. Pattern types gives control on the flows of walking humans. Templates and types are user-defined. In the following section, we put these concepts into practice and present our results.

7.5 Applications and Results

In this Section, we fully illustrate the concept of crowd patches with two concrete applications. First, we procedurally generate a potentially infinite pedestrian street, populated with

a bottom-up approach. Second, we use a pre-computed city environment complemented with a template map to populate it. Finally, we analyze the performances obtained for the generation of patches and the online simulation.

7.5.1 Applications

Bottom-Up Approach. The main advantage of a bottom-up approach is that environments are procedurally generated at run-time and can grow in unexpected ways. We illustrate this approach here with a simple example: a potentially infinite pedestrian street, procedurally generated at run-time with a rule-based algorithm. The infinite deployment of this street is demonstrated in a video available online [Yersin et al., 2009] and in Figure 7.1.

To use crowd patches in a bottom-up approach, several steps have to be followed in a pre-process. First of all, we design a library of typical static obstacles: streetlights, trash cans, trees, city maps, benches, etc. These static objects are illustrated in Figure 7.7 (left). Secondly, endogenous objects, like humans talking together, playing children, or seated people, have to be created (Figure 7.7 (center)). In a third step, the required pattern types are identified: an endless street requires among others, empty patterns for building borders, and specific I/O space distributions to simulate lane formations in highly crowded portions of the street. For each identified pattern type, we then generate various patterns. Note that the set of patterns does not need to be exhaustive; if a pattern is missing, it can be generated at run-time. However, the smaller the number of patches/patterns to generate online, the faster the simulation. Various pattern examples are illustrated on the right of Figure 7.7. Similarly to patterns, patch templates are identified according to the sort of environment to generate online. A first non-exhaustive set of patches for each template is created too, using the pattern library. Some of these patches are shown in the bottom of Figure 7.7. Finally, to further vary the patch content, we have also defined an additional set of grounds, or textures that can be applied on a patch: cobblestones, grass, asphalt, etc. The resulting variety is visible in the video and Figure 7.1.

At run-time, patches are introduced in the scene wherever the camera is looking. Specific patches are first looked for in the existing library. If no patch matches the requirements, it is generated on-the-fly. If the patch generation requires a specific pattern that has not yet been created, it is also created online. A second important step at run-time is to update each pedestrian on its trajectory. When a human reaches a patch border, we seamlessly make it move to the neighbor patch. To efficiently achieve this, each trajectory possesses a parameter pointing to the neighbor trajectory that should be followed next.

Top-down approach. A major advantage of pre-defined environments is the possibility to pre-compute many elements and dedicate the resources spared in this way for run-time rendering and animation of pedestrians. Our second example, illustrated in Figure 7.8 and in the video [Yersin et al., 2009], is a large city whose geometry has been previously designed, along with a template map to assist the populating process.

In such a case, the whole set of patches can be computed offline. Based on the template map, each patch is consecutively generated, along with the required patterns. Note that in the case of a city, we typically require patterns of all types to simulate streets, building entries,

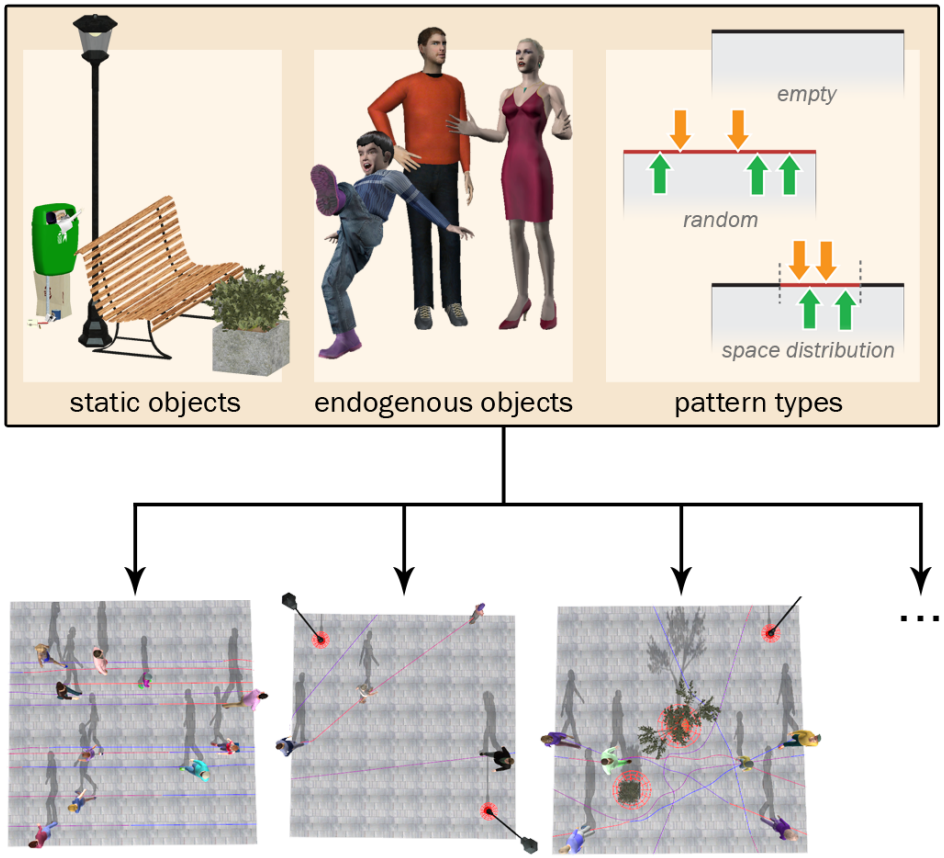


Figure 7.7: To build a lively pedestrian street, we use *(left)* static obstacles, *(center)* endogenous obstacles, and *(right)* specific patterns. *(Bottom)* From these sets, we first define several patch templates. Finally, each template is instantiated to result in a variety of patches.

public places, etc. Once the full map has been computed, the runtime execution can start. Since everything is pre-computed, the user can freely move in the whole city and observe the pedestrians evolving everywhere. At each time step, the work to achieve is reduced to the updates of pedestrians' animation, position, and rendering. We further detail the achieved performances in the next Section.

7.5.2 Results

The performance tests and the video have all been performed on a desktop PC with a dual core 2.4GHz, 2GB of RAM, and an Nvidia Geforce 8800 GTX. We have instantiated 6 different human templates, rendered with a level-of-detail approach, ranging from about 6'000 to 1'000 triangles. They are animated with a motion capture database. Shadows are computed on the environment and the humans. In our implementation, the computation of a patch takes approximately 60ms. This number fits perfectly to real-time procedurally generated environments, given that a first non-exhaustive library of patches has been pre-computed.

The infinite street environment has been built with one patch template, and a total amount

Figure 7.8: *(Left)* A pre-defined city populated with crowd patches, based on a template map. *(Right)* The same image with apparent trajectories and patch borders.

of 10 patterns only, of empty, space-constrained, and random types. The patterns all have a period of $10s$ and an $8m$ length . The average number of visible patches in front of the camera is 500, 75% of which are buildings with 2 entrances allowing people to come in and out. The remaining visible patches represent the street and its crossings. There are approximately 750 visible walking humans at all times (exogenous objects), 30 idle humans representing endogenous objects, and 80 static objects, such as benches, or streetlights. In average, we obtain 30 frames per second, including the rendering of the humans and the environment. The frame rate is relatively constant over the whole progression on the street, since the number of patches and humans which are computed and displayed remains the same. As illustrated in the video, the result is impressively varied, given that we have used 10 patterns only. In Table 7.1, we show the variation of the frame rate for the simulation only (rendering is deactivated), when the number of patches to update is modified.

# of updated patches	# of simulated humans	average frame rate
1	1-5	275-285
45	67-93	140-155
103	144-206	135-140
206	372-420	130-135
411	586-634	110-120
810	1185-1240	95-105
1845	2596-2693	40-45
3772	3572-3725	30-35

Table 7.1: Frame rate evolution (without rendering) for a varying number of patches to simulate.

The city environment was pre-computed with 20 different patterns of a $10s$ period and an $8m$ length. All the pattern types previously introduced were represented. The template map that was used has a size of 32x32 pixels, where each pixel represents one square patch. There are thus a total amount of 1024 patches. We used 4 patch templates (city place, street,

building entrance, public park). Open spaces represent 25% of this map. We have simulated approximately 2500 exogenous humans, 180 endogenous humans, and 300 static objects at 20fps (including rendering). In this case, the trajectories have all been pre-computed. The obtained frame rate is thus limited by the rendering of the whole city and its citizens, rather than the simulation itself, which only takes 5-10% of the resources. With this second example, we confirm that a small number of patterns can already bring a great variety of trajectories to the environment. The spectator has to focus in order to detect the periodicity of the simulation.

7.6 Conclusion, Discussion and Future Work

In this Chapter, we presented an alternate solution to navigate crowds at a very low cost. As compared to the use of online motion planning techniques (introduced in Chapter 4), crowd patches represent a faster method for populating large-scale interactive virtual environments with walking and idle humans, as well as animated and static objects. The key-idea of our solution is to build environments from a set of blocks, the crowd patches, that describe periodic motion for a small local population, as well as other environment details. Periodicity in time allows endless replay. Our solution provides an advantageous trade-off between memory usage, computation needs, and motion quality. Crowd patches allow to handle large-scale environments and to densely populate them with believable motions. Our method also provides a solution for designing environments from sets of templates. Crowd patches have some limitations, but there are solutions to alleviate them, as discussed in the following paragraphs.

Time Period. Patches and patterns have a user-defined period. Contiguous patches need to have an identical period to be connected. As a result, a limitation of our solution is the need for an identical period for all interconnected patches. Changing the period is easy when dealing with exogenous objects, as we proposed an automatic technique for computing their periodic animation. However, endogenous objects may have hand-designed animation, making the change of period duration more difficult.

Static Patches. Patches are created once at a given place: the overall distribution of the population in the environment is thus static in time. As a result, it is not possible to synthesize dynamic events, such as a small demonstration of people going through the environment. Also, due to the finality of the patches' layout, no goal-directed navigation can be achieved for a human instance: by following a specific pedestrian, a spectator may observe some incoherent navigation (roaming or going nowhere). To solve these limitations, an interesting direction for future work would be to allow dynamic changes of patterns and patches.

Variety in Space and Time. If spectators' point of view remains static for a long period of time, the animation periodicity may be detected. It is possible to make the detection more difficult by generating a variety of patches from identical patterns, instead of a unique in-

stance. This variety can be obtained by randomizing initial connections between inputs and outputs at the patch creation stage, whilst endogenous and static objects remain. It is also possible to imagine further variations when computing trajectories by modifying the steering method parameters when solving interactions between people. However, memory consumption is proportionally increased. Generating a variety of patches for identical patterns appears to be the most interesting direction for improving our approach.

Interactivity. As emphasized in the two previous paragraphs, character motions are pre-computed and fixed using crowd patches. This does not allow rich interactivity between the virtual population and the user. We add subtle details, such as humans looking towards the camera, that simulate interactivity. However, the pre-computed locomotion trajectories do not account for the presence of spectators. We propose two future directions to allow interaction between the users and the virtual population. The first one would be to locally edit locomotion trajectories in order to account for the presence of spectators. However, patterns define strict limit conditions for these trajectories. Managing edition and limit conditions would then need special care. A second solution would be to consider that patches are only used to simulate secondary characters, whereas interactive digital actors would be added to this background population. Interactions between digital actors and secondary characters would also need careful management.

Improving Motion Quality. Exogenous trajectories for walking humans are computed using a particle-based simulation. Such a technique is advantageous, because it very efficiently computes trajectories that remain reasonably believable. Recently, Kwon and colleagues proposed a solution to edit the motion of groups [Kwon et al., 2008]. To further improve our results quality, we could automate and use such editing techniques to compute trajectories inside patches and combine them with small group motions. As edition in space and time is achievable with Kwon's approach, group trajectories can be adapted in order to meet limit conditions imposed by patterns.

In conclusion, the proposed crowd patches approach has several paramount advantages. We ease the design process for virtual populations using a library of patch templates. Secondly, we make it possible to use time consuming animation techniques, as motions are pre-computed and stored. In the case of environments and patches generated online in real-time, our solution is more efficient than a complete crowd simulation. Finally, we allow the handling of large-scale environments and populations by drastically lowering the need for computation resources dedicated to simulation. Another major advantage of our approach is the ability to guarantee simulation content: trajectories are fully solved once patches are computed, and the resulting animation is reproducible: risks of deadlock situations or other artifacts are eradicated, and the simulation can be subtly locally edited (*e.g.*, by replacing some patches) at some given places without any impact on the remaining parts of the environment. Some drawbacks relative to the repetition of pre-computed motions have been pointed out in the previous paragraphs. However, several tracks have been proposed to alleviate these limitations.

CHAPTER 8

Conclusion

In this Chapter, we summarize our contributions in real-time crowd simulations, and discuss future work.

8.1 Summary of Contributions

Hybrid Motion Planning. We have presented a hybrid motion planning architecture. This system can handle realistic and real-time crowd motion planning for thousands of pedestrians. Our approach is scalable: the user can easily divide the scene into regions of varying interest and use different motion planning algorithms according to their level. One major advantage of this scalable approach is that it allows the user to determine the performance he wishes to achieve and to select and distribute the regions of interest (ROI) accordingly.

For the different levels of interests, we have developed several motion planning techniques: an accurate potential field-based method is used for pedestrians in ROI 0. Additionally, a simple and efficient short-term avoidance algorithm is exploited in both ROI 0 and 1, which ensures no noticeable transition when a pedestrian reaches a region border. With our architecture, results show that it is possible to simulate over $10,000$ characters in real time, including the computational costs of rendering, animation, and shading. Moreover, our architecture allows to define many more sets than a purely potential field-based approach. Realism is further demonstrated with observable emergent behaviors, like lane formations, and panic escape.

Crowd Behavior. We have presented the different steps of our work to add special behaviors in a crowd. First, based on an environment geometry annotated with semantic data, we showed how to extract this information, and use it at runtime to trigger intelligent behaviors in specific places of the environment.

Second, we have presented tools to help a designer to directly annotate the Navigation Graph with semantic information. These annotations allow a pedestrian to perceive its environment, and react to high-level requests issued by the user.

Finally, we have shown that it is possible to insert an additional and scalable layer to our motion planning architecture, in order to simulate realistic small groups of pedestrians. Our study shows that such an addition offers a larger variety of interpretations, and a more sympathetic ambiance in virtual scenes.

Crowd Patches. We have presented an alternate solution to navigate crowds at a very low cost. As compared to our online motion planning architecture, crowd patches allow to populate virtual environments at a much larger scale. The key-idea of our solution is to create a set of crowd patches, or small zones where periodic motion is pre-computed, and assemble them to build environments. Periodicity in time allows endless replay. This solution offers a trade-off between memory storage, runtime computations, and motion quality. With crowd patches, it is possible to populate environments at a very large scale and ensure no collisions between pedestrians and their surroundings. We make it possible to use time consuming animation techniques, as motions are pre-computed and stored.

Although the pre-computation of crowd patches prevents from having a direct interaction between a user and the crowd, patches allow to focus the computational resources on other demanding tasks, and to run a crowd simulation unlimited in time and size.

8.2 Future Work

Hybrid Motion Planning. We see two main improvements that should be explored in a future work. Firstly, potential field-based approaches are not suited in cases where the crowd is too densely gathered in a narrow place. Such situations should be automatically detected, and faster short-term collision avoidance techniques should be preferred over potential field computations. This automatic detection can be easily achieved using the Navigation Graph: graph vertices with especially small radius in certain places could be detected at initialization and watched at runtime. If the pedestrian density becomes too high in such vertices, the switching to another planning technique could be automatically achieved.

A second limitation is our set-based approach: we have to assign a limited number of goals shared by several pedestrians. This limitation is alleviated by the many different paths that pedestrians can use to reach their goal. However, a more individual approach would make each pedestrian unique. Following one person in a crowd would have a meaning, for its destination would be different from all the others. A solution that we should explore in a future work is to use an agent-based approach to steer pedestrians very close to the camera. This could be achieved by inserting an additional ROI of higher interest, in which such a

technique would be used. The main difficulty in this approach is to find a satisfying agent-based technique that is not too costly, and a solution to make transitions between this ROI and the others seamless.

Crowd Behavior. We have presented tools for the user to issue online high-level path requests to the Navigation Graph. Online path requests need to be answered in real time. To achieve this goal, an interesting lead for future work would be to make a Navigation Graph hierarchical: graph vertices in a same regions are grouped together to form a larger vertex. Then, sets of larger vertices are grouped to form an even larger vertex, etc. This would lead to much faster path finding. Moreover, with such a structure, many global paths could be pre-computed at initialization, *i.e.*, find paths between the major places of the environment; for instance in a city: between the station, the mall, the park, the bar, etc.

Concerning the group behaviors, from our study, we have seen that adding attentional behaviors to groups could much enhance their believability. Such behaviors include: talking to each other, look at each other from time to time, but also look at other pedestrians passing by. This can be easily achieved in the future by controlling the animation of virtual humans' upper body, depending on the situation. Attentional behaviors are currently investigated by Grillon and Thalmann [Grillon and Thalmann, 2009], where they are introduced in Yaq as an additional animation layer. A second important point is to take into account the type of pedestrians that are put in a same group: we expect businessmen to be on their own, or accompanied by other business persons. With the same idea, kids need to be accompanied with one adult. Such additional constraints on group constitution can be easily modeled and introduced.

Crowd Patches. One major improvement that we can easily add to crowd patches in a future work is to compute several paths per patch: when a patch is created, we first choose 4 patterns. From these patterns, inputs and outputs are randomly connected. If we connect them differently, we obtain new trajectories without changing the patch border constraints. At runtime, the trajectories to use in a given patch could be regularly modified to increase the feeling of motion variety.

In order to improve the runtime patch assembly, another interesting lead would be to investigate the work of Cohen et al. on Wang Tiles and see how it could be extended to assemble crowd patches [Cohen et al., 2003]. Indeed, Wang Tiles are squares with edges of various colors. To assemble two Wang Tiles, their edge colors must match. Square edges can be interpreted as patterns in our patches, and techniques developed to tile a plane with Wang Tiles can probably be adapted to crowd patches.

A third lead for future work is to integrate crowd patches as an additional level of interest within our hybrid motion planning technique. For instance, crowd patches would be ideal in the background, for inter-pedestrian collisions would be avoided, and the computational cost of the method is very low. However, special care would be required at the region border, to switch from a crowd patch-ruled algorithm to a Navigation Graph- or potential field-based algorithm.

LIST OF FIGURES

BIBLIOGRAPHY

O. Arikan, S. Chenney, and D. A. Forsyth. Efficient multi-agent path planning. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 151–162, New York, NY, USA, 2001. Springer-Verlag New York, Inc. ISBN 3-211-83711-6. 2.2.2

A. Aubel, R. Boulic, and D. Thalmann. Real-time display of virtual humans: Levels of details and impostors. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(2):207–217, 2000. 2.1

S. Bandi and D. Thalmann. Space discretization for efficient human navigation. *Computer Graphics Forum*, 17(3), 1998. 2.2.2

O. B. Bayazit, J.-M. Lien, and N. M. Amato. Roadmap-based flocking for complex environments. In *PG '02: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, page 104, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1784-6. 2.2.2

O. Burchan Bayazit, Jyh-Ming Lien, and Nancy M. Amato. Better group behaviors in complex environments using global roadmaps. In *ICAL 2003*, pages 362–370, 2003. ISBN 0-262-69281-3. 2.2.2, 2.7, 2.3.2, 8.2

P. Bourke. `http://local.wasp.uwa.edu.au/~pbourke/geometry/insidepoly/`, 1987. 6.2.2

E. Bouvier and P. Guilloteau. Crowd simulation in immersive space management. In *Proceedings of the Eurographics workshop on Virtual environments and scientific visualization '96*, pages 104–110, London, UK, 1996. Springer-Verlag. ISBN 3-211-82886-9. 2.2.1

S. Chenney. Flow tiles. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 233–242, 2004. 2.2.2

M.Geol Choi, J. Lee, and S.Y. Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics*, 22(2):182–203, 2003. 2.2.2

Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang Tiles for image and texture generation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 287–294, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. doi: http://doi.acm.org/10.1145/1201775.882265. 8.2

J.-M. Coic, C. Loscos, and A. Meyer. Three LOD for the realistic and real-time rendering of crowds with dynamic lighting. *Research Report LIRIS, France*, 2005. 2.1

L. Gonzaga da Silveira and S. R. Musse. Real-time generation of populated virtual cities. In *Proc. ACM VRST*, pages 155–164, 2006. ISBN 1-59593-321-2. doi: http://doi.acm.org/10.1145/1180495.1180527. 2.3.1

P. de Heras Ciechomski, S. Schertenleib, J. Maïm, D. Maupu, and D. Thalmann. Real-time shader rendering for crowds in virtual heritage. In *VAST'05*, pages 1–8, 2005. 2.1, 2.2, 8.2

S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry / impostor crowd rendering system. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 95–102, 2005. ISBN 1-59593-013-2. doi: http://doi.acm.org/10.1145/1053427.1053443. 2.1, 2.1, 8.2

N. Farenc, R. Boulic, and D. Thalmann. An informed environment dedicated to the simulation of virtual humans in urban context. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics'99)*, volume 18(3), pages 309–318. The Eurographics Association and Blackwell Publishers, 1999. 2.3.1

S. J. Fortune. Voronoi diagrams and delaunay triangulations. *CRC Handbook of Discrete and Computational Geometry*, pages 377–388, 1997. 2.2.2

P.G. Gipps and B. Marksjo. Micro-simulation model for pedestrian flows. *Mathematics and Computers in Simulation*, 27:95–105, 1985. 2.2.1

P. Glardon, R. Boulic, and D. Thalmann. PCA-based walking engine using motion capture data. In *Proc. of Computer Graphics International*, 2004a. 3.2.1, 3.6, 3.2.2, 8.2

P. Glardon, R. Boulic, and D. Thalmann. A coherent locomotion engine extrapolating beyond experimental data. In *Proceedings of International Conference on Computer Animation and Social Agents (CASA'04)*, 2004b. 3.2.1, 3.2.2

H. Grillon and D. Thalmann. Simulating attentional behaviors for crowds. In *submitted to International Conference on Computer Animation and Social Agents (CASA'09)*, 2009. 8.2

K. Hauser, T. Bretl, and J.C. Latombe. Non-gaited humanoid locomotion planning. *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, pages 7–12, 2005. 2.2.2

L. Heïgeas, A. Luciani, J. Thollot, and N. Castagné. A physically-based particle model of emergent crowd behaviors. In *Graphicon*, 2003. 2.2.2

D. Helbing and P. Molnár. Social force model for pedestrian dynamics. *Physical Review*, 51 (5):4282–4286, 1995. 2.2.1

D. Helbing, P. Molnár, and F. Schweitzer. Computer simulations of pedestrian dynamics and trail formation. *Evolution of Natural Structures*, pages 229–234, 1994. 2.2.2

D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000. 2.2.1, 2.2.2

D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: experiments, simulations and design solutions. *Transportation science*, pages 1–24, 2005. 2.2.1

J.K. Hodgins, W.L. Wooten, D.C. Brogan, and J.F. O'Brien. Animating human athletics. *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 29:71–78, 1995. 2.2.1, 2.4, 8.2

K. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoï diagrams using graphics hardware. *SIGGRAPH'99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 33:277–286, 1999. 2.2.2, 3.3

H. Hoppe. Progressive meshes. *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, 1996. 2.1

R. L. Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36:507–535(29), 2002. 2.2.2

R. L. Hughes. The flow of human crowds. *Annual Review of Fluid Mechanics*, 35(1):169–182, 2003. 2.2.2

W. Jager, R. Popping, and J.P. Van de Sande. Clustering and fighting in two-party crowds: Simulating the approach-avoidance conflict. *Journal of Artificial Societies and Social Simulation*, 4(3), 2001. 2.2.1

A. Kamphuis and M.H. Overmars. Finding paths for coherent groups using clearance. *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28, 2004. 2.2.2, 2.3.2, 2.14, 8.2

L. Kavan, S. Dobbyn, S. Collins, J. Zara, and C. O'Sullivan. Polypostors: 2d polygonal impostors for 3d crowds. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 149–155. ACM Press, February 2008. 4.4

L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Proceedings of IEEE Transactions on Robotics and Automation*, pages 566–580, 1996. 2.2.2

O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986. 2.2.1, 4.2.1

A. Kirchner and A. Shadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. In *Physica A*, pages 237–244, 2001. 2.2.2, 2.2.2

H. Klüpfel, T. Meyer-König, J. Wahle, and M. Schreckenberg. Microscopic simulation of evacuation processes on passenger ships. In *Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry*, pages 63–71, 2000. 2.2.1

J. J. Kuffner. Goal-directed navigation for animated characters using real-time path planning and control. In *CAPTECH '98: Proceedings of the International Workshop on Modelling and Motion Capture Techniques for Virtual Environments*, pages 171–186, London, UK, 1998. Springer-Verlag. ISBN 3-540-65353-8. 2.2.2

T. Kwon, K. Hoon Lee, J. Lee, and S. Takahashi. Group motion editing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, 2008. doi: http://doi.acm.org/10.1145/1399504.1360679. 2.3.2, 7.6

F. Lamarche and S. Donikian. Crowds of virtual humans : a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum (Eurographics'04)*, 23(3):509–518, September 2004. 2.2.2, 2.8, 2.3.1, 8.2

J. Lander. Skin them bones. *Game Developer Magazine*, pages 11–14, May 1998. 2.1

J.-C. Latombe. *Robot Motion Planning*. Boston: Kluwer Academic Publishers, 1991. 2.2.2

M. Lau and J. J. Kuffner. Behavior planning for character animation. *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 271–280, 2005. 2.2.2, 2.9, 8.2

K. Hoon Lee, M. Geol Choi, and J. Lee. Motion patches: building blocks for virtual environments annotated with motion data. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 898–906, 2006. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179352.1141972. (document), 7.2, 7.2, 8.2

K. Hoon Lee, M. Geol Choi, Q. Hong, and J. Lee. Group behavior from video: a data-driven approach to crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 109–118, 2007. ISBN 978-1-59593-624-4. 2.2.2

A. Lerner, Y. Chrysanthou, and D. Lischinski. Crowds by example. *Computer Graphics Forum (Eurographics'07)*, 26(3), 2007. 2.2.2

C. Loscos, D. Marchal, and A. Meyer. Intuitive crowd behaviour in dense urban environments using local laws. *Theory and Practice of Computer Graphics*, pages 122–129, 2003. 2.2.1, 2.2.2, 2.6, 2.13, 2.3.2, 8.2

J. Maïm. *Generating, Animating, and Rendering Varied Individuals for Real-Time Crowds*. PhD thesis, EPFL, 2009. 2.1, 3, 3.2, 3.2.1

J. Maïm, B. Yersin, M. Clavien, and D. Thalmann. http://www.youtube.com/watch?v=ZA3gzqG94JQ, 2007. 6.2.1, 6.2.3

J. Maïm, B. Yersin, and D. Thalmann. Unique instances for crowds. *To appear in IEEE Computer Graphics and Applications*, 2009. 2.1, 2.2, 3.2.1, 4.4, 6.3.2, 8.2

S. Raupp Musse and D. Thalmann. A hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001. 2.2.1, 2.3, 2.3.2, 8.2

S. Raupp Musse and D. Thalmann. A model of human crowd behavior: Group interrelationship and collision detection analysis. In *Eurographics worksop on Computer Animation and Simulation*, 1997. 2.2.1

S. Raupp Musse, C. Babski, T. Capin, and D. Thalmann. Crowd modelling in collaborative virtual environments. In *Proc. ACM VRST*, pages 115–123, 1998. ISBN 1-58113-019-8. doi: http://doi.acm.org/10.1145/293701.293716. 2.3.1

C. Niederberger and M. H. Gross. Hierarchical and heterogeneous reactive agents for real-time applications. *Computer Graphics Forum*, 22(3):323–331, 2003. 2.3.2

C. O'Sullivan, J. Cassell, H. Vilhjámsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang. Levels of detail for crowds and groups. *Computer Graphics Forum*, 21(4): 733–741, 2003. 2.3.2

S. Paris, S. Donikian, and N. Bonvalet. Environmental abstraction and path planning techniques for realistic crowd simulation. *Computer Animation and Virtual Worlds*, 17:325–335, 2006. 2.2.2

S. Paris, J. Pettré, and S. Donikian. Pedestrian steering for crowd simulation: A predictive approach. In *Computer Graphics Forum (Eurographics'07)*, 2007. 2.2.2, 2.10, 8.2

N. Pelechano, K. O'Brien, B. Silverman, and N. Badler. Crowd simulation incorporating agent psychological models, roles and communication. *First International Workshop on Crowd Simulation*, 2005. 2.2.1

N. Pelechano, J.M. Allbeck, and N.I. Badler. Controlling individual agents in high-density crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2007. 2.2.2

J. Pettré, J.-P. Laumond, and T. Siméon. A 2-stages locomotion planner for digital actors. *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 258–264, 2003. 2.2.2

J. Pettré, P. de Heras Ciechomski, J. Maïm, B. Yersin, J.P. Laumond, and D. Thalmann. Real-time navigating crowds: scalable simulation and rendering: Research articles. *Computer Animation and Virtual Worlds*, 17(34):445–455, 2006. 2.2.2, 2.3.1, 3, 3.1, 3.2, 3.1.1, 3.3, 3.9, 3.10, 3.11, 4.2, 4.2.1, 8.2

J. Pettré, H. Grillon, and D. Thalmann. Crowds of moving objects: Navigation planning and simulation. In *Proc. IEEE ICRA*, 2007. 2.2.2, 2.3.1, 3, 3.1

H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, 2000. 2.1

W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291, 1987. ISBN 0-89791-227-6. doi: http://doi.acm.org/10.1145/37401.37435. 3.2.2

C. W. Reynolds. Steering behaviors for autonomous characters. *Proc. of Game Developers Conference*, pages 763–782, 1999. 2.2.1, 2.3.2, 4.2.1, 4.2.2, 4.3

C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 25–34, July 1987. 2.2.1, 2.4, 2.3.2, 5.1, 8.2

C. W. Reynolds. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, 2006. 2.2.2, 2.3.2

R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, 2004. 3.2.2

A. Safonova, J. K. Hodgins, and N. Pollard. Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. *ACM Transactions on Graphics*, 23(3):514–521, 2004. 2.1

W. Shao and D. Terzopoulos. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28, 2005. ISBN 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073371. 2.1, 2.2.2, 2.3.1

A. Sud, E. Andersen, S. Curtis, M. Lin, and D. Manocha. Real-time path planning for virtual agents in dynamic environments. *Proceedings of IEEE VR*, pages 91–98, 2007. 2.2.2

M. Sung, M. Gleicher, and S. Chenney. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3):519–528, 2004. 2.3.1, 2.12, 8.2

M. Sung, L. Kovar, and M. Gleicher. Fast and accurate goal-directed motion synthesis for crowds. *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 291–300, 2005. 2.2.2

F. Tecchia, C. Loscos, R. Conroy, and Y. Chrysanthou. Agent behavior simulator (abs): A platform for urban behaviour development. In *In GTEC'2001*, pages 17–21, 2001. 2.11, 2.3.1, 8.2

F. Tecchia, C. Loscos, and Y. Chrysanthou. Visualizing crowds in real-time. *Computer Graphics Forum*, 21(4):753–765, December 2002a. 2.1, 2.1, 8.2

F. Tecchia, C. Loscos, and Y. Chrysanthou. Image-based crowd rendering. *IEEE Computer Graphics and Applications*, 22(2):36–43, 2002b. ISSN 0272-1716. doi: http://dx.doi.org/10.1109/38.988745. 2.1

D. Terzopoulos. Artificial life for computer graphics. *Communications of the ACM*, 42(8): 32–42, 1999. 2.2.1

The Khronos Group. http://www.collada.org/. 3.2.1

A. Treuille, S. Cooper, and Z. Popovic. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, 2006. doi: http://doi.acm.org/10.1145/1179352.1142008. 2.2.2, 4.2.2, 4.3, 4.4, 4.8, 4.4, 8.2

B. Ulicny, P. de Heras, and D. Thalmann. Crowdbrush: Interactive authoring of real-time crowd scenes. *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 243–252, 2004. 2.1

J. van den Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin. Interactive navigation of individual agents in crowded environments. *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 2008. 2.2.1, 2.5, 8.2

M. Wand and W. Straßer. Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum (Eurographics'02)*, 21(3):483–491, 2002. 2.1

B. Yersin, J.Maïm, P. De Heras Ciechomski, S. Schertenleib, J. Pettré, P. Glardon, M. Clavien, and D. Thalmann. `http://vrlab.epfl.ch/~maim/v-crowds05.high.mp4`, 2005. 6.1.2

B. Yersin, J.Maïm, F. Morini, M. Clavien, and D. Thalmann. `http://www.youtube.com/watch?v=ifimWFs5-hc`, 2007. 4.4, 4.4

B. Yersin, J.Maïm, M. Clavien, and D. Thalmann. `http://www.youtube.com/watch?v=QpwbhNE6Fe8`, 2008. 6.3.2

B. Yersin, J.Maïm, M. Clavien, and D. Thalmann. `http://www.youtube.com/watch?v=kQvBUCOen2s`, 2009. 7.5.1, 7.5.1

BARBARA YERSIN

Av. du Chablais 43, CH-1008 Prilly

Age 28, single
Nationality Swiss

Email :
barbara.yersin@epfl.ch
Tel : +41 21 693 52 48
Mobile : +41 21 629 25 54

EDUCATION

- 2005-2009 — **EPFL (Federal Institute of Technology of Lausanne)**
  PhD candidate, Virtual Reality Laboratory.
  Real-Time Motion Planning, Navigation, and Behavior for Large Crowds of Virtual Humans.
- 2004-2005 — **Université de Montréal**
  Master Project, Computer Graphics Laboratory.
  « Environmental Relighting with Precomputed Irradiance Cube Maps for Immersive Walkthroughs. »
- 1999-2005 — **EPFL (Federal Institute of Technology of Lausanne)**
  Master degree in Computer Science.
- 1999 — **Gymnase de Beaulieu, Lausanne**
  Maturity in Economics.

EXPERIENCE

- 2005-2007 — **Teaching Assistant, EPFL, Lausanne**
  Tutoring exercises in Computer Graphics and Advanced Computer Graphics courses.
- 2004-2005 — **Cinétoile Cinema, Lausanne**
  Barmaid.
- 2003 — **Forum EPFL, Lausanne**
  Commitee Member, Responsible for Student Relations.
  Annual manifestation regrouping over 100 companies and 400 EPFL master students to facilitate first job opportunities and contacts.

SKILLS

- **Programming** — C, C++ OpenGL, SQL, LaTeX, Java, ActionScript
- **Languages**

French	mother tongue
English	fluent
German	school knowledge, easily perfectible

PUBLICATIONS

- **2009** **Unique Instances for Crowds.**
Jonathan Maïm, Barbara Yersin, and Daniel Thalmann.
To Appear in Computer Graphics and Applications Journal.

  YaQ : An Architecture for Real-Time Navigation and Rendering of Varied Crowds.
Jonathan Maïm, Barbara Yersin, Julien Pettré, and Daniel Thalmann.
To Appear in Computer Graphics and Applications Journal – Special Issue on Virtual Populace.

  Crowd Patches : Populating Large-Scale Virtual Environments for Real-Time Applications.
Barbara Yersin, Jonathan Maïm, Julien Pettré, and Daniel Thalmann.
In proceedings of I3D'09.

- **2008** **Real-Time Crowd Motion Planning : Scalable Avoidance and Group Behaviors.**
Barbara Yersin, Jonathan Maïm, Fiorenzo Morini, and Daniel Thalmann.
The Visual Computer Journal 24(10) : 859-870.

- **2007** **Populating Ancient Pompeii with Crowds of Virtual Romans.**
Jonathan Maïm, Simon Haegler, Barbara Yersin, Pascal Mueller, Daniel Thalmann and Luc Van Gool.
The 8th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST'07), Brighton, UK, November 26-30.

  Crowd Simulation.
Daniel Thalmann and Soraia Raupp Musse.
Springer Editions, October, 30.

  Real-Time Scalable Motion Planning for Crowds.
Fiorenzo Morini, Barbara Yersin, Jonathan Maïm and Daniel Thalmann.
In proceedings of the 2007 International Conference on Cyberworlds, Hannover, Germany, October, 24-26.

- **2006** **Real-Time Navigating Crowds: Scalable Simulation and Rendering.**
Julien Pettré, Pablo de Heras Ciechomski, Jonathan Maïm, Barbara Yersin, Jean-Paul Laumond and Daniel Thalmann.
Computer Animation and Virtual World (CAVW) Journal - CASA special issue.

- **2005** **Steering a Virtual Crowd Based on a Semantically Augmented Navigation Graph.**
Barbara Yersin, Jonathan Maïm, Pablo de Heras Ciechomski, Sébastien Schertenleib and Daniel Thalmann.
First International Workshop on Crowd Simulation (V-CROWDS), Lausanne, Switzerland, November 24-25.