ÉCOLE POLYTECHNIQUE
Promotion X 99
David MARCHAL

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

Simulating pedestrian crowd behaviour in virtual cities

NON CONFIDENTIEL

Option	Informatique INF 591
Champ de l'option	Géométrie et synthèse d'images
Directeur de l'option	Jean Marc Steyaert
Directeur de stage	Céline Loscos
Dates du stage	15 avril - 5 juillet 2002
Adresse de l'organisme	UNIVERSITY COLLEGE LONDON Department of Computer Science Gower Street WC1E 6BT London Royaume-Uni

Abstract

In games, entertainment, medical and architectural applications, the creation of virtual city environments has recently became widespread. Populating these models raises problem such as real-time rendering and simulation. A populated 3D virtual city model has previously been developed at UCL, that simulated up to 10,000 pedestrians walking in real-time. However, they stressed on the rendering aspects rather than on the behaviour of the pedestrians, which only perform collision detection; therefore the simulation ends to be fairly unrealistic. To that extent, this report presents a pedestrian crowd simulation method aiming at improving the local and global reactions of the pedestrians. On the one hand, we enhanced the pedestrian-to-pedestrian collision avoidance method based on a subdivision of space into a 2D grid. On the other hand, pedestrians are given goals to reach making their trajectories smooth and straight. Goals are computed automatically and connected into a graph that reflects the structure of the city and triggers a spatial repartition of the density of pedestrians. In order to engender realistic reactions when areas become crowded, local directions are stored and updated in real-time, allowing the apparition of pedestrians streams. Combining the different methods proposed in this project contributes to a realistic appearance of the model. In this project, we successfully provide a framework for a more realistic crowd behaviour while keeping a real-time frame rate for up to 5,000 simulated pedestrians.

Résumé

Des domaines aussi variés que les jeux vidéos, l'architecture, ou la médecine ont vu récemment l'apparition d'un grand nombre d'environements virtuels urbains. Insérer une population dans ces environements pose de nombreux problèmes comme le rendu en temps réel et la simulation de comportements humains. Une équipe de UCL a développé une cité virtuelle en trois dimensions simulant jusqu'a 10,000 piétons se déplaçant en temps réel. Toutefois, elle a mis l'accent sur l'efficacité du rendu en 3 dimensions plutôt que sur le comportement des piétons: ceux-ci savent uniquement éviter les collisions, ce qui rend la simulation assez irréaliste. Ce rapport présente donc une méthode de simulation d'une foule de piétons visant à améliorer leurs réactions locales et globales. D'une part, la méthode de détection de collision, qui utilise une subdivision de l'espace dans une grille a deux dimensions, a été améliorée. D'autre part, comme les agents cherchent à atteindre un but, leur trajectoires sont régulières. Un graphe calculé automatiquement relie les buts entre eux et reflète bien la structure de la ville. Cela permet une répartition non uniforme des densités de piétons qui correspond à la réalité. Enfin, le fait que les agents soient capables de suivre un champ de directions mis a jour en temps réel engendre l'apparition de files de piétons, ce qui contribue au réalisme du modèle. Ce projet permet donc une simulation plus réaliste d'une foule de piétons tout en procurant une animation fluide jusqu'à 5,000 humains virtuels.

Contents

Chapter 1

Introduction

Like the car traffic, the pedestrian traffic may be quite congested in large cities. The design of most public facilities should often enable the passage of hundreds of thousands pedestrians per day. Railway stations, shopping centres or tourist places most of the time attract high density crowds. So as to cut designing or architectural expenses, it could be really interesting to be able to simulate the movement of a pedestrian crowd in such environments. Besides, architectural theories such as Space Syntax [7] link the physical aspects and the pedestrian business of a place.

Moreover, seeing the recent success of new technological devices like cell phones, palm devices or on-board GPS, simulating a pedestrian crowd in a city could have utterly different aims. Thus, the work presented in this report takes part of a wide project named *Equator* [12], involving eight English universities. The central goal of the Equator project is to promote the integration of the physical with the digital. It indeed aims at improving the quality of everyday life by building and adapting technologies for a range of user groups and application domains. An example of applications is creating new forms of play, performance and entertainment that combine the physical and digital so as to promote learning, participation and creativity.

Virtual cities are a large part of Equator project. It focuses on combining physical and digital cities to improve people's understanding of the world within which they live, and to enhance way-finding and access to physical and digital artifacts, information and people. In this framework, a virtual city simulation has been developed at UCL. The aim is to allow the user to interactively navigate through a complex, polygon-based city, in which thousands of virtual humans are moving. So far, though, the random behaviour of the pedestrians is far from giving the impression of intelligent

agents. Some realistic behaviours such as awareness of the pavements or smooth and straight trajectories are clearly missing.

This report therefore presents a way of simulating an intelligent crowd without using too complex techniques such as Artificial Intelligence. Indeed, as the city must be displayed in 3D and in real-time with 10,000 people moving in it, the behaviour of each individual cannot be too complicated. However, so that the crowd looks like a real pedestrian crowd, our goal is to manage to make pedestrians walk in flows and in a coherent way. Whereas most available models which describe the behaviour of a crowd usually deal with macroscopic variables like average and flow, we developed an individual-based model.

Our work is essentially based on the previous project developed at UCL by Tecchia et al. [2, 3, 4, 13]. We also used the crowd observations made by F. Feurtey [6] to assess the realistic appearance of our simulation. Finally, some collision detection algorithms developed by Musse et al. [8, 9] and Reynolds [10] contribute to the methods used in this project.

In Chapter 2, we present the related work dealing with virtual cities, real crowd features and former work on crowd modeling. Then, in Chapter 3, we explain the methods used to perform the four main developed aspects: making the agents aware of the city environment, performing a real-looking and fast agent-to-agent collision avoidance, managing the self-emergence of flows and making the agents walk in small groups. In Chapter 4, some interesting details of implementation are followed by statistic and results of the algorithm. This report ends with a conclusion and a discussion on future work in Chapter 5.

Chapter 2

Related work

Considering Equator project's aims, the project is probably bound to be at least a part of an Intelligent Virtual Environment; though, we do not deal with any Artificial Intelligent topics in this chapter contrary to [1]. Our main purpose is to give agents realistic trajectories at an individual level and to give the crowd movement a realistic look at a macroscopic level. There has been a lot of work intending to improve individual features of agents: in [5], for example, they add subconscious actions (like walking stooped when one is sad) so that there may be a greater diversity of behaviours. Although such work is significant, we will not mention it any further, since it is not related to the goal of the project.

In the following, we first present the previous simulation developed in UCL. Then, we enumerate some real crowd characteristics that, first, correspond to what is expected of the agents[1] and, second, enable us to assess the results given by our algorithm. Finally, examples of former implementations of crowd enable us to compare different methods used to perform collision detection or group behaviour.

2.1 Real-time visualisation of densely populated urban environments

This project intends to improve one of the aspect of a large project of populated virtual cities started by Tecchia et al. In a first attempt [2], they proposed a simple way to run real time collision detection, enabling the simulation of around 10,000 humans. In order to perform the collision detection

[1] In the following, the term agent is used as well as humans to describe virtual pedestrians. It does not refer to any Artificial Intelligence topics.

as quickly as possible, they used an approximated method based on space **discretisation**: the virtual humans, here considered as simple particles, are moving on a *2D grid map* [2], which stores the height at each point of the environment. As only one agent at a time is allowed to occupy a tile, the collision detection is all the more accurate as the resolution of the map is higher. However a trade off needs to be found between the precision and the speed of the simulation and the memory cost.The way the collision detection works is quite straightforward. First a height map is captured to represent the height of the obstacles in the city. When a particle moves from one tile to another, the height stored at the next aimed position is checked and compared to the current. If it is not too different from the current height, the cell is considered accessible. If not, the particle changes its direction by gradually rotating until an obstacle-free direction is found. So as to obtain a smoother animation, this collision detection algorithm can predict the i^{th}-next movement. This method, thanks to its simplicity, gave good results: the computation time used for collision detection increases linearly with the number of particles and is all the less significant as the complexity of the 3D scene is high. However, trajectories were not smooth because a random direction was chosen when a collision was detected.

The precedent algorithm was further improved in [3]. Using the same method of discretisation, the authors tried to ameliorate the impression of intelligence for the agents. Whereas the precedent platform had only one layer storing the height map, they increased this number to four: each layers corresponding to one kind of behaviour. The first layer stores the position of the building and is used for detecting the collision with the buildings. The second layer stores the agents positions and is consequently used for inter-collision detection, allowing agent to avoid each other. In addition, a behaviour layer was implemented for more complex behaviours, such as walking on a pavement or deciding to change direction. Finally, a call-back layer was added to create interactions with other objects such as calling elevators or climbing into a bus. Thanks to its 4 layers, the platform provided overall a simple way to create and test new kinds of agent behaviours. However, the four layers were not fully exploited although the structure of such a platform should provide a good framework for developing macroscopic behaviour.

In an other work, Tecchia et al. [13, 4, 14] have enhanced the 3D ap-

[2] In this report, the word "map" always describes a 2D array representing one aspect of the city in a discretised way. For example a "human map" describes the positions of every agent. A *tile* corresponds to a cell of this array.

Figure 2.1: The display of 10,000 avatars with shadows in real-time

pearance of the scene using an image-based rendering technique to display multiple avatars and shadows. Indeed, they focused more on the speed of the 3D display than on the real-looking behaviour of the pedestrians. Indeed, so far, few works were done to enable the interactive visualisation of crowd represented by realistic looking animated humans. The original idea was published in [13]. Rather than displaying a 3D polygon model for every pedestrian, which will limit the frame rate due to the high increase of the polygon number, each of them is represented by a single polygon oriented towards the viewpoint and mapped with an appropriate texture (image representing a virtual human) depending on the viewpoint and the walking steps of the pedestrians. Consequently, they stored one texture by avatar and by frame of animation. They used a compression method to decrease the memory cost. This image-based rendering method therefore enables great savings in polygonal number, that is the limiting factor of graphic cards. Moreover, it provides a simple way of touching up the environment's reality since it is quite easy to generate new appearances for human beings; human shadows can be generated in a similar way [14]. An example of simulation with this system can be seen in Figure 2.1.

2.2 Crowd characteristics

Frank FEURTEY [6] collected a great number of crowd features. First, these data can help finding what is significant in the way a crowd behaves so as

to be able to model this behaviour afterwards. Nevertheless, the purpose of these data is overall to check the validity of our model. In an experimental point of view, it may be valid only if it fit well with sociological observations. Among them, only a few concern the way people are walking; most of the others are about external constraints such as the weather or the time of the day.

2.2.1 Intrinsic features

Humans need space to walk. Whereas this free area is basically a way of preserving some intimacy, the individual area required to move in a city depends mainly on the density. The intimacy is preserved by other means such as not looking directly at each other. Without being compressed, the minimum space required by a still human is $0.14m^2$ (for a man) whereas $0.5m^2$ is necessary for a walking pedestrian. This **space requirement** lies at the roots of the approximation of the collision tiled-map of the previous work on virtual cities as described in the previous section 2.1. Though more precise studies showed that the shape of the free area around a walking human is a parabolic curve, whose apex is all the farther forward as the speed is higher. Besides, human speed ranges from 44 to 122 m/min. One characteristic that will really need to be checked on our model is the relationship between flow, speed and density of people. The flow q(in pedestrians per minute per metre width), the walking speed U (in metre per minute) and the density of pedestrians k (in pedestrians per squared metre) follow the rule:

$$q = k.U \tag{2.1}$$

Thus the speed is maximum when the density is the lowest (free flow). But whereas the density increases, the flow keeps on increasing (in spite of the decreasing speed) till a **critical value of the density**. Above this critical value, the flow decreases till being null when the traffic is congested. The links between these three features are summarised in Figure 2.2.

On an individual level, what may help us setting the behavioural rules of humans are the three main observed behaviours of real pedestrians: monitoring, yielding and streaming. **Monitoring** is the fact to assess the behaviour of other people to avoid collision or to evaluate their state of mind (in a hurry, angry...). This assessment is performed within an angle due to vision field. **Yielding** is the fact to change trajectory to avoid a collision with somebody else. The distance at which one starts detouring is all the longer as the density is slower. If both start detouring on the same side, a kind oscillation may appear. When density is quite high, people are likely to

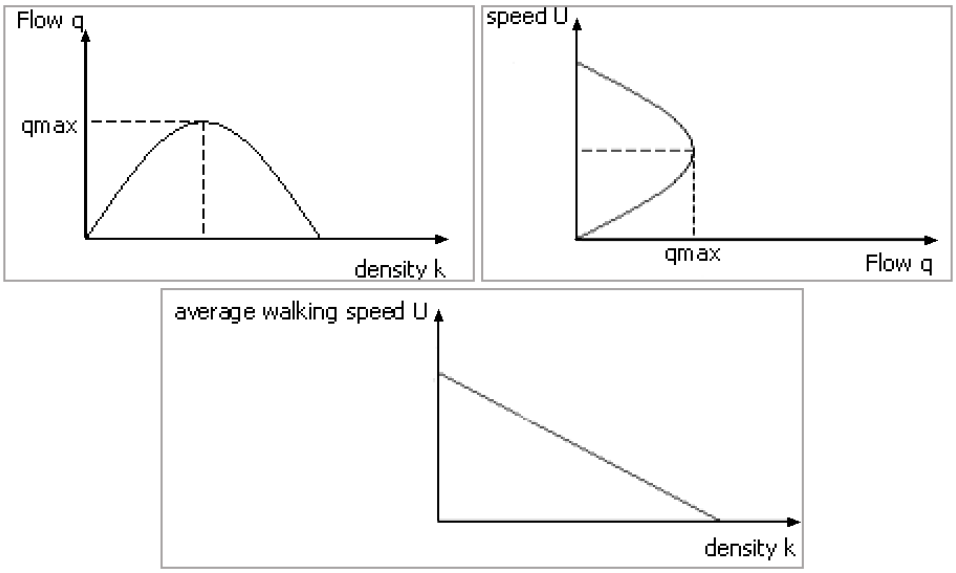

Figure 2.2: Flow, density and average speed relationships

follow the ones who walk in the same direction as them: this is **streaming**. Thus flow lines appear. Most of the time, one tries to walk so as to see above the shoulder of the one who walks ahead of him.

2.2.2 External contributions

Once the human way of moving is valid, we could incorporate some external contributions into the model to get a better reality. Among them, there is the average length of a walk in a city. People do not actually walk for a very long time: most daily walking trips do not last more than 6 min. In other words, 50 % of the people walk less than 1.4 km per day. Moreover the traffic deeply depends on external factors. On the one hand, the weather is really significant since pedestrian traffic decreases by 60% in case of rainy weather. On the other hand, there are daily cycles affecting the pedestrian density: for example the traffic is denser at peak time, for example in the morning when people go to work, or at lunch time. Besides, the frequentation of a street is utterly different whether it is a shop-oriented or office-oriented or residential-oriented street.

The study of traffic in cities (Jiang et al. [7])is termed as **space syntax**, which as a long tradition in urban studies. In space syntax, traffic concentrates on open spaces (streets, squares...) which are all connected with each

other. This is this **interconnection graph** that gives each node (most often crossroads) its unique position. In this approach, every open space is represented as the longest straight line as we can see on Figure 2.3.

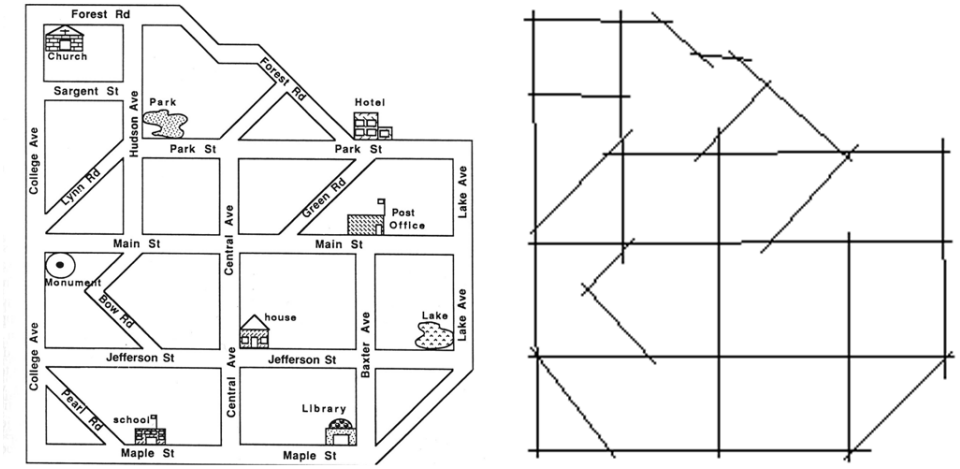

Figure 2.3: An urban environment and its axial representation

The connectivity C_i of a line i is defined as the number of lines connected to a line. The control value $Ctrl_i$ of a line i expresses the number of choices each line represents for its immediate neighbours as a line to move to. It is determined according to the following calculation:

$$Ctrl_i = \sum_{j=1}^{k} \frac{1}{C_j} \tag{2.2}$$

where k is the number of directly linked nodes of the considered node i, and C_j is the connectivity of the j^{th} directly linked node.

The **depth** can be defined as the number of steps from a considered node to all the other nodes. Thus a node may be said to be deep or shallow. The average depth MD_i of the node i is defined by

$$MD_i = \frac{\sum_{j=1}^{n} d_{i,j}}{n-1} \tag{2.3}$$

where n is the number of nodes of a whole graph, and $d_{i,j}$ the shortest distance between the node i and j.

Finally, the **integration** of a node is expressed by a value that indicates the degree to which a node is integrated or segregated from a system (global

integration) or from a part of a system (local integration). It is measured with Relative Asymmetry (RA) as follows:

$$RA_i = \frac{2(MD_i - 1)}{n - 2} \tag{2.4}$$

Thus, space syntax aims at describing some areas in a city in the sense of integration and segregation. A location is more integrated if all the other places can be reached from it after going through a small number of intermediate places. With such parameters, the movement pattern in a city can be understood and predicted: for example, the better a line is local-integrated, the busier it will be. Some space syntax analyses even show correlations between predicted segregated areas and burglary event places.

2.3 Related work on crowds

2.3.1 The big issue: collision detection

Bouvier and Cohen [11] implemented a simulation of a crowd involving 45,000 persons. Their model stems from Newtonian mechanics, under the influence of various forces that describe the human behaviour. The particles are able to avoid collision just before they occur and to have and change goals. Although the model is robust enough to reach several thousands of particles, the behaviours remain quite simple. The other methods presented below do not reach such big number of agents.

C. Reynolds [10] presents a set of **steering behaviours** that could be useful in animation or games. The author used an autonomous simple vehicle-based model on a point mass approximation, which trajectory is computing using simple mechanics laws. Each vehicle has a desired steering force and its position is computed by taking different elements into account. Thus, Reynolds implemented a large set of individual or group behaviours such as pursuing a moving goal, obstacle avoidance, path following, or flow field following. A crowd behaviour simulation could be realised by combining these numerous kinds of individual behaviours. For example, to model a caribou fleeing through a forest, *evasion* and *collision avoidance* are combined. Although the results are really impressive, they cannot be applied in the modeling of a crowd of several thousands agents for real-time simulation, since all these behaviours are based on an exact mathematical computation. One of the main differences between this work and Tecchia et al.'s one [2, 3] is the discretisation of space. Reynold's vehicles avoid each other in an exact way: they compute the distance between them and perform an exact circle

collision detection. On the contrary, there is no extra-cost for the discretised method: only one agent at a time may occupy a tile.

Frank Feurtey [6] developed another exact computing method simulating the collision avoidance behaviour of pedestrians. He proposed a new approach of collision detection based on predicting and modifying trajectories in a (x, y, t) space. Given the different velocity vectors of the other agents, one can draw their future trajectories in the (x, y, t) space, providing they will keep the same speed. With a limited speed, one has to find a path into a future cone (still in (x, y, t) space), free from any intersection with the other agents predicted paths. To perform this task, Feurtey used the projection of all the predicted trajectories onto a right section of the future cone. After this, each virtual human computes its new velocity by minimising a cost function involving the cost of moving away from goal, changing direction, acceleration and decelerating. With 12 pedestrians, he simulated a crossroad and compared his results with experimental data.

Musse et al. [9] proposed a **multi-resolution collision detection** based on two different collision avoidance algorithms. The first one is the simpler: after predicting the collision mathematically, the slower of two agents stops just before the collision occurs. This can be implemented as in Algorithm 2.1.

With this algorithm, we cannot be sure that, in case of a front collision, the agents will not bounce one against the other. On the contrary, a second algorithm enables the two agents to go round each other. As the second method is much more expensive in computational time, it is used only when the observer position is really nearby the collision place. The multi-resolution term stems from this use of either one or the other collision detection method, which enable Musse et al. to save 20% of computing time. Once again, this model has only been run with about ten agents.

2.3.2 Group behaviour

In a city, less than a half of the pedestrians walk alone. Indeed most people walk by two. To simulate a realistic environment, it is necessary to implement a group behaviour.

Thanks to his steering behaviours, Reynolds [10] invented the well-known **boids** (for bird-oid). Boids abide by a flocking rule that is simulated using the three *Separation*, *Cohesion* and *Alignment* basic steering behaviours. *Separation* means that they cannot be too close one from the other; *Cohesion* means that they all try to reach the centre of the flock; and *Alignment* forces them to set their direction as a mean of the surrounding agent directions.

Algorithm 2.1 agent-to-agent collision avoidance

```
For each pair of agents do
    If ( vectors are not collinear ) then
        If ( linear velocities are different ) then
            stop the slower agent
        Else
            choose one agent in a random way
        End if
    Else
        If (  vectors are convergent ) then
            choose one agent in a random way
            change its direction
        Else
            increase the linear velocity of the agent
            which walks ahead
        End if
    End if
End for
```

The result of the combination of these three laws was very good to model flocks, herds or schools but not ordinary humans (see Figure 2.4).

S.R. Musse et al. proposed an intermediate method involving group behaviour and 3D rendering in [8]. Indeed, they defined a crowd as a set of groups formed by human agents, each group having a list of goals. Agents from a same group share the same list of goals but social effects can occur: agents may change group. The collision avoidance was performed by using a list of well-defined goals and computing the Bézier curve, which has been improved afterwards as we saw it in section 2.3.1. To generate the group behaviour, they used 3 main laws: the agents from the same groups walk at the same speed, follow the same predefined path and can wait for each other when one agent is missing. In another paper [9], S.R. Musse and D. Thalmann made a point of the crowd behaviour analysis with a more sociological point of view. To enable more human-like reactions, each agent is specified by a level of dominance, a level of relationship and an emotional status and is ruled by seeking and flocking laws. Using the same list of goals as before, the behaviour of the crowd is thus really improved: for example, they have implemented the simulation of 4 groups visiting a museum, each

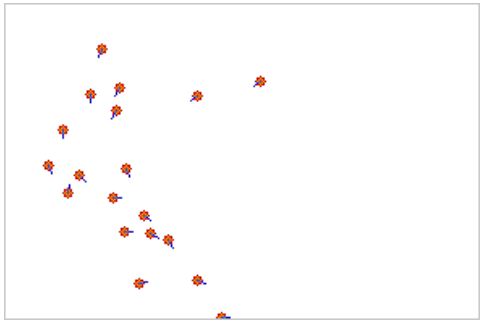

Figure 2.4: Boids simulation

of the group aggregating gradually as time elapses. This is an example of global behaviour generated by local laws.

2.4 Discussion

In this chapter, after describing the basic work that we want to ameliorate, we have made a point of the main features of real pedestrian crowds that we would like our simulation to be in accordance with: agent-to-agent behaviours such as yielding or streaming lie probably at the roots of the real appearance of the simulation. Then, we have enumerated some related work about pedestrian crowd simulation. It is however quite significant to notice that none of these works manage both features of being really realistic concerning the pedestrian behaviour and fast enough for large scale simulation. They will nevertheless be useful in the next chapter insofar as some of them can be adapted in a simple and fast enough way for our purpose.

Chapter 3

Toward realistic pedestrian behaviour

More than giving the main ideas developed in this project to simulate the pedestrian behaviour, this chapter also aims at highlighting the route that has led us to the final algorithm. This route, far from being a straight line, has been meandering between good and less good ideas, and it therefore illustrates the experimental aspect of the project.

3.1 Implementing town rules

3.1.1 The rules

We are so accustomed to walking in a town that we are not aware of the number of implicit rules that such walks refer to. As for virtual agents, it is obvious that they have to avoid penetrating into buildings as if they did not exist. But more complicated rules as walking on the pavements are also really necessary if we want the town to look like a real one.

As it has been done before and for efficiency reasons [2, 3, 13], we use a 2D array with a certain resolution to represent the physical city. The input data is a map with only buildings and ground symbolised on it. An example of such a map is shown in Figure 3.1. This map is used in parallel with a function allowing or not the access to the different kind of static objects, for instance BUILDING and GROUND. Giving access to GROUND only would enable the agents to avoid penetrating into the buildings. Yet, the streets are unfortunately not fully for pedestrians: we have therefore to prevent virtual humans from walking in the middle of the street; as a simplification,

pavements could be thus defined as the transition areas between buildings and streets. The instance PAVEMENT is therefore added to the static map and regions covered by the tag PAVEMENT are accessible by the agents. Thus, the streets are considered as forbidden areas. This should improve the accuracy of the simulation. Though, if design only pavements, it is quite unlikely that the pavements of two different blocks of buildings would have a common area: most agents would consequently be bound to walk around the same building. This implies that we have to add **pedestrian crossings** as well. The instance CROSSING is the second type of accessible areas for agents.

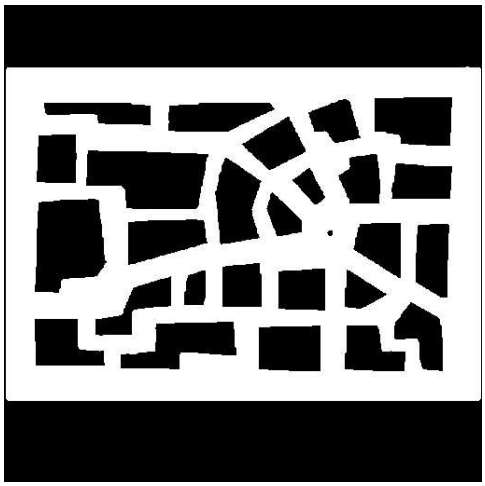

Figure 3.1: This is the input image used to represent the city. Buildings are black and streets white.

However simple all these concepts may be for ordinary citizens, it is not so easy to make virtual people understand these rules. Our aim is in fact to design automatically pavements and pedestrians crossings from a map with building and ground only. Giving access rights to certain areas is not enough: in the real streets, it is quite scarce to see anybody (even on the pavement) walking around a telephone pole. People may wait in the street, yet it would not be realistic if everybody kept walking round! This seems to lead us to goals setting. Agents can be given goals to reach. We have therefore worked on an automatic settings of the goals according to the static map.

The above reflections led us to design a set of methods to automatically define the regions where the agents can go. This is described in the section

3.1.4. In order to make the pedestrians walk more straight and in a more decisive way, we implemented two different techniques. The first one explained in section 3.1.2 is designed so that pedestrian are guided by levels. The second described in section 3.1.3 assigns to each agent a goal.

3.1.2 Gradient solution

At the beginning, we did not intend to implement all these rules at once: the most significant one was the pavement rule. Rather than simply forbidding buildings, we tried to develop a method that could naturally bring pavements without having to model them, and a good way of making pedestrians walk around buildings. The geographic maps and their relief representation lie at the roots of this technique: beginning with a binary map, black on the buildings areas and white for the ground, we simply apply a blurring filter on it as shown on Figure 3.2. Using an appropriate **convolution blur** (several times if necessary), it was possible to transform the two-level map into a multi-level (that we will call altitude) one with values going gradually from white to black, the whitest areas being the middle of the streets and the darkest ones corresponding to the buildings. By setting two thresholds (maximal altitude and minimal altitude allowed), we can thus force humans into walking in the grey areas that correspond to the edges of the buildings (see Figure 3.3 [1]). But it enables another interesting feature: if virtual humans try to minimise the difference of altitude between two consecutive points of their trajectory, they are forced to follow the iso-level lines of the altitude map, thus avoiding bumping directly into buildings.

There are though several drawbacks concerning this method. It is quite difficult to perform a good blur that shift our two initial levels into 256 different ones. All the more so as we would like, in the same time, to preserve the acute angles of the map (otherwise, people would be allowed to bump into the corners of the buildings). In practice, there are about 15 levels of altitude and often large areas at the same level. That means that one cannot use only the difference of height to guide the humans. Moreover, even if the height levels were well ranged, another force would be necessary to prevent all the people from following the same trajectories. On Figure 3.3, an agent was tracked and its trajectory displayed.

[1] As afterwards, the buildings are displayed in black.

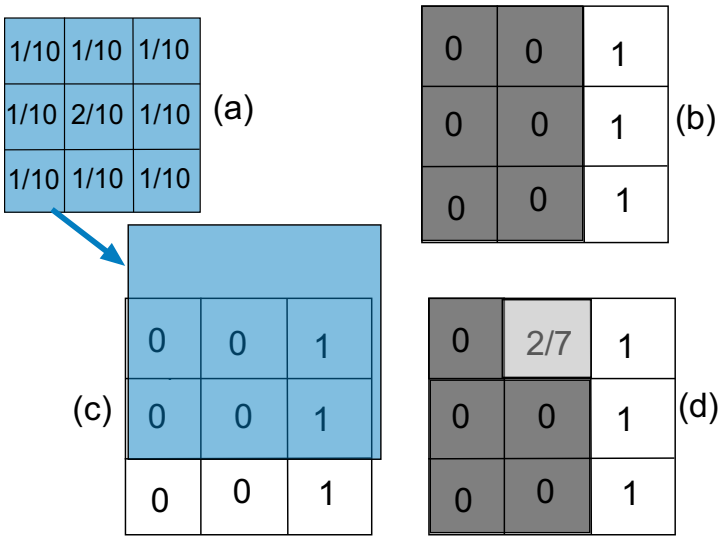

Figure 3.2: Convolution blur method: (a) represents the convolution matrix that is applied on the image (b). On (c), the matrix is applied on the upper centre pixel. The result is shown on (d). If the matrix is applied to every pixel, a gradient from white to black is created.

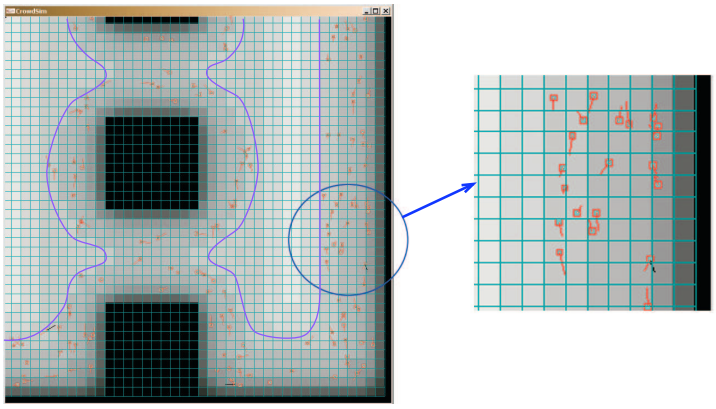

Figure 3.3: Illustration of the gradient method: the program was running with 600 humans following the level lines and avoiding each other in a very simple way. On the left, we can see the building surrounded by a gradient from dark grey to white. The limit in purple defines a kind of pavement. As we can see on the right, the directions of pedestrians follow the iso-level lines.

3.1.3 Gradient solution with goals

The limited results of the method described in the previous section (section 3.1.2) encourage us to introduce goals. For the same reason as for the pavement, we had to find a way of placing the goals on the map automatically. It is quite important to notice that the problems of setting the goals and reaching them are utterly different: in the first case, we want to find a method with no time constraints as it is only pre-processing. In the second case, the method will have to be performed for each human and every frame. It would have to be a really quick function.

In our system, a goal is a point defined by its position and its radius. A goal will be considered as reached by a virtual human as soon as the human stands closer from it than the radius. Besides, goals are not to be taken exactly like real human goals such as shopping or picking up somebody at the station. Although the idea is to make the observer believe that agents are acting so, our goals turn to be some "temporary places to go". Unless there is a straight path between the agent and its goal and provided there is no other agent to hinder its movement, reaching a goal is not only keeping getting closer to it, but it is also accepting to do a detour if necessary. We observed that a lot of ways of computing a distance are available to perform such a task. In addition, every agent has the memory of the three former goals it has reached so that it may not always walk around the same ones.

We made several successive improvements concerning goals setting. First, considering where real people usually intend to go in the streets, we realised that their goals are likely to be situated anywhere in a street. Thus our first method set the goals randomly on the accessible areas. When an agent reaches a goal, it is randomly given a new one. Unfortunately this new goal may be as far or hidden as possible from the former one and the human is bound to walk randomly, unable to reach it. Consequently, the goals have been linked into a **graph**. In this graph, two goals are neighbours only if they are visible the one from the other: that is there is no building or ground between them. Once a virtual human reaches a goal, it is given a new one which is one of the neighbour of its former goal. But even with this method, some agents get lost: indeed, this approach cannot avoid a goal to be so isolated from the other that no one can reach it or, worse, no one can get away from it because it does not belong to a connected part of the graph.

Finally, to be sure that every goal would not be too remote from each other, a good way of setting them is to put them on the obstacle (building or ground) **corners**. Besides, in every day life, it is quite usual to indicate strangers a route by giving them the direction they have to follow at each

crossroad. As our first map were "hand made", all the buildings simply laid with horizontal or vertical edges. The corners were consequently all right angles. Using a convolution-like method presented in Figure 3.4, it was quite easy to determinate where the corners were and thus to set the goals. They were then linked into a visibility graph.

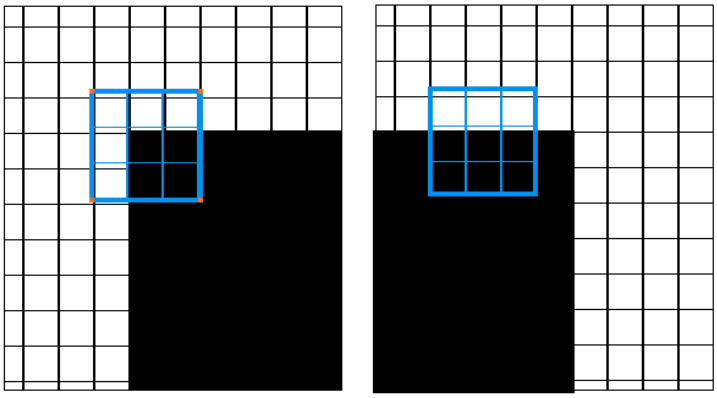

Figure 3.4: In this approach of corner detection, the buildings (black) are set up the value 1 whereas the streets (white) have the value 0. The 3 by 3 matrix of 1, shown highlighted on the top of the main grid, is convoluted with each pixel. When the result of the convolution is 4 (left), we have found a right corner; otherwise, it is not a corner (right). When a corner has been found as for the picture on the left hand side, the goal is set in the middle of the convolution mask. For this method to work properly, every building has obviously to be horizontal-vertical shaped.

In addition to the fact that this method gave quite good trajectories, it enables us to notice something important: the altitude map, that we had used so far, is less and less significant. As the agents are guided from corner to corner they do not need any longer to have a altitude level to follow. That is why we decided to stop to use the gradient map and to use only the goals instead to help the navigation of the agents. Figure 3.5 shows a trajectory that definitely does not take the level lines into account.

However, this method has drawbacks. The corner detection algorithm implemented here placed the goal inside corners, that is in an area with a forbidden access for agents. The goal was nonetheless reachable since its radius was large enough to spread till the accessible area. The problem was that all the agents were strangely moving along the walls as they got near the goals. Because of this affluence along the walls (whereas other routes

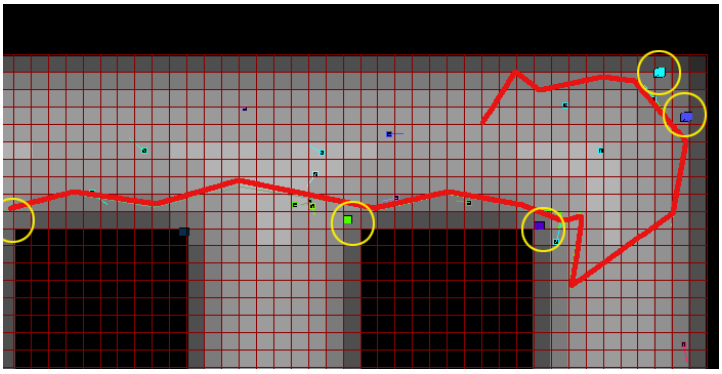

Figure 3.5: Gradient solution with goals: This is the trajectory followed by a virtual human. The yellow circles represent every goals it has reached so far. It shows how useless the static blurred map is.

would have been possible by walking far from the wall), unexpected traffic jams occurred with quite low densities.

3.1.4 Designing pavements and pedestrian crossings

This section presents the final algorithm used to set goals, design pavements and pedestrian crossings. First, we build pavements, set goals on the corners of the pavements, then we simplify the graph of goals and finally build pedestrian crossings.

As our interest is to make the method work for any type of city, the input binary map representing the location of buildings is no longer made by hand but loaded from a file representing the ground of an existing 3D city virtual model. Thus the ground of our simulation corresponds to a single textured polygon. Rather than using a blur-like method to **design the pavements**, they can be more simply defined by enlarging the building areas. However, the former **corner detection** method used to assign goals is no more valid with such a city model. As in a real town, streets are not necessarily oriented vertically or horizontally and corners can be acute or obtuse angles. To place the goals on the corners, we therefore use a new more flexible method that needs to detect the succession of three different levels: GROUND-PAVEMENT-GROUND or BUILDING-PAVEMENT-BUILDING (see on Figure 3.6). This very succession has to be found several times for the same corner with different angles. Thus the corner detection is more or less precise and can locate more or less corners (see in section 4.2.1 for a more

precise algorithm). But in order to have goals all around the buildings, we prefer to set them so as to get first more corners than necessary. This corner detection is performed on the pavements once we have built them with half of their final size. Afterwards, pavements are widened to their full width and the detected corners thus appeared to be in the centre of the pavements.

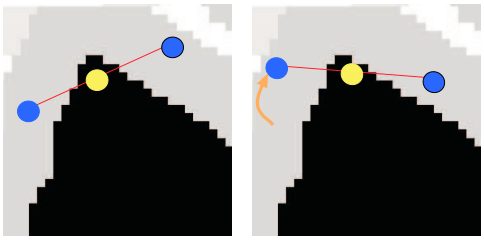

Figure 3.6: This illustrates the corner detection method. Once the centre point is on a building (black), the segment turns till the two extremities are on pavement (grey). Once such a case has been found, there must be another angle of the segment with the same succession PAVEMENT-BUILDING-PAVEMENT (right figure) so that the centre point may be registered as a corner.

Like before, the inter-visible corners are then connected into a **graph**. Although the number of goals is too large, the method is quite efficient insofar as there is no pavement piece without a goal (the goal-less pavements would be also agent-less) while everything is completed automatically. Besides, it is quite interesting to notice that the density of detected corners is higher when the angles are neither horizontal nor vertical shaped. This might be a drawback but we use it to increase the realism as explained in section 3.3.2. We will see later that the higher the goal density, the higher the people density. Fortunately, the areas that do not look vertical-horizontal shaped often look more like a pedestrian city centre.

Finally, the number of goals has to be decreased: this is done by **merging the closer goals** but keeping connectivity. If two close goals A and B are to be merged, they are replaced by one goal, whose position is the barycentre of A and B and whose neighbours are the neighbours of goal A and those of goal B, even if these neighbours are no more visible from the new position. These method is summarised in Figure 3.7.

Finally, as soon as the goals are well set, it is possible to design the **pedestrian crossings** automatically, by taking advantage of the simplified goals graph. For every pair of goals, it is possible to know if they are on

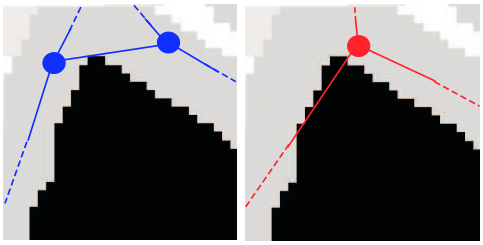

Figure 3.7: The goal merging method aims at merging the closest goals (left) without losing connectivity: on the right, the new goal is connected with the left hand corner although the building lies between the both.

opposite sides of a street by checking the accessibility of the tiles between both of them. Thus, every goal is linked by a pedestrian crossing with its two closest neighbours that lie on opposite sides of its street. Although it does not reflect exactly the way real pedestrian crossings lay, the pedestrian crossings created with this method correspond to a true necessity in terms of traffic (see on Figure 3.8).

Figure 3.8: This is an example of the automatic designing of pavements(in gray) and pedestrian crossings(in light gray).

Using a description of space in terms of pavements, pedestrian crossings and goals, pedestrians are able to compute their trajectory in real-time concerning the static obstacles. The problem is slightly different with the other humans since they are also moving. Besides, in exact computing methods such as Feurtey's 2.3.1, the position and velocity of each agent are assessed at each frame to re-compute the optimal trajectory. Although we will not take every agent into account, the position of humans has to be assessed indeed quite often.

3.2 Individual behaviour and inter-collision

Regarding a human flow, it is quite logic to compare it with a particle flow: thus, we should set local rules, from which global behaviours such as streaming are likely to stem. After setting those local rules in an efficient way, we will see that global behaviour sometimes needs further contributions to appear (in section 3.3).

The positions of every agent are stored in an array called **human map**, and only one agent can occupy a tile at a time. There are many ways to perform collision detection. One is to check one tile ahead of the agent. This check up is performed at each frame. If the agent is to change tile and the aimed tile is already occupied, the agent tries to avoid collision. According to Musse and Thalmann [9], it is a good method to switch behaviour with the angle between the two velocity vectors. We consider three main cases as shown on Figure 3.9 . If the two velocity vectors are perpendicular

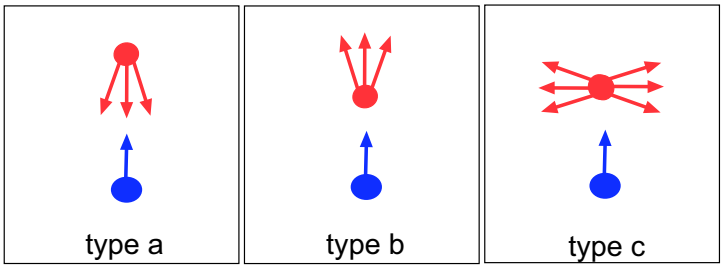

Figure 3.9: The three main collision cases: *front* collision (a), *following* collision (b) and *perpendicular* collision (c).

(Figure 3.9(c)), the agent gives way to the other. If they are basically in the same direction (Figure 3.9(b)), the rear agent tries to follow the leader one. Finally, if they are likely to bump into each other (Figure 3.9(a)), the agent turns to avoid collision. To get smooth curved trajectories, it is more efficient to perform the speed vector rotation with a pace of $\pi/8$ than $\pi/4$. Although this method allows the agents to bend their trajectories to avoid a front collision, such a behaviour is never observed because they do not have enough time to perform it well. Indeed checking just the next tile is not enough to implement a good collision avoidance.

That is why the number of checked tiles needs to be increased. It is still necessary to check the immediate next tile ahead in case of a perpendicular way-cutting collision. But we add a five tiles ahead check as we can see on Figure 3.10. In case there is somebody on this new tile, the behaviour of our

agent is changed, but in a slightly different way whether the human obstacle is near or far. The farther it is, the smoother the direction or speed changes are.

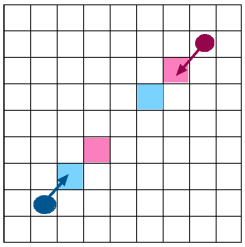

Figure 3.10: Each agent checks the next tile and the fifth next tile according to its velocity vector.

Even when we try to check 3 tiles (the next one, the third next one and the fifth next one) , there are still collisions that are not well avoided. In fact, in case of a front collision, it is likely that each one of the agents checks a free tile and that, due to certain speed value, they may walk towards each other without detecting each other before they both stand on the other next tile. If that kind of collision happens too often (and this is the case), the simulation is not realistic. Thus the method logically evolves to a **ray tracing approach**. Rather than checking only the first, third and fifth next tiles, we throw a ray ahead of the agent, which reacts differently if it is intercepted close or far from it. We check up to ten tiles ahead of the agent. Three obstacle-distance ranges are set, within which the reaction is different: for a close obstacle, the agent is allowed to turn a lot to avoid collision whereas it may just turn of PI/8 for a middle distance. For farther collision detections, the reaction is most of the time simply to slow down.

We have to highlight the fact that agents have a zero degree sight field. They check for obstacles presence just in the direction of their speed. Setting local rules in case of a non null sight field would be far more complicated but certainly more efficient. Agents blinkers make them react as surprised people when they have to avoid a perpendicular collision.

3.3 Managing flows

As we have just seen, the local rules are quite simple. Whereas they are fine enough to enable agents to avoid people in different ways considering the relative speed, they do not allow them to take their neighbours global

movement into account. That is why, for example, they do not change direction if another agent is walking in the same direction as them but one metre aside. To make the agents understand better the surrounding environment, we make them aware of flows as explained in section 3.3.1, and of the surrounding density (see section 3.3.2).

3.3.1 Taking flows into account

The local rules alone are unable to trigger streaming. Consequently, agents need to be aware of the flows. Compared to individual human movements, flow changes are quite a scarce process. That is why we can consider them as static components. We represent flows as a **direction field**. In a map such as the static map, the direction of each tile is stored. These directions are updated by every human each time they move. But direction intensities decrease linearly with time: indeed, if a human of direction $\overrightarrow{D_0}$ crosses a tile at an instant t_0, this tile is set at direction $\overrightarrow{D_0}$. The intensity $D(t)$ of this tile then decreases linearly till being null at time $t_0 + \tau$ where τ is the relaxation time.

$$\overrightarrow{D(t)} = \left(\frac{t_0 - t}{\tau} + 1\right).\overrightarrow{D_0} \tag{3.1}$$

The longer the relaxation time, the more static the direction field is. The other way for a direction to last long is to be updated by the arrival of another agent. If $\overrightarrow{D_1}$ is the direction of this second agent, the new direction $\overrightarrow{newD}$ stored in the tile at instant t is:

$$\overrightarrow{newD} = \frac{1}{2}\left(\left(\frac{t_0 - t}{\tau} + 1\right).\overrightarrow{D_0} + \overrightarrow{D_1}\right) \tag{3.2}$$

However, updating the direction field is not enough: agents have to follow it. Thus, when they check if their direction is adapted with their goals, they also take directions into account. Contrary to the human collision avoidance, the process of goal checking is carried out in different directions, reflecting a real sight field. Every virtual human tries to match the next tile direction with their own speed. The more recently the tile direction has been set, the more significant it will be for the human. We can see the visualisation of the direction field on Figure 3.11. Finally, the direction field method brings a fare improvement to the local rules. The local rules would certainly have been really more time consuming if we had tried to implement them so as to get a similar result without direction field.

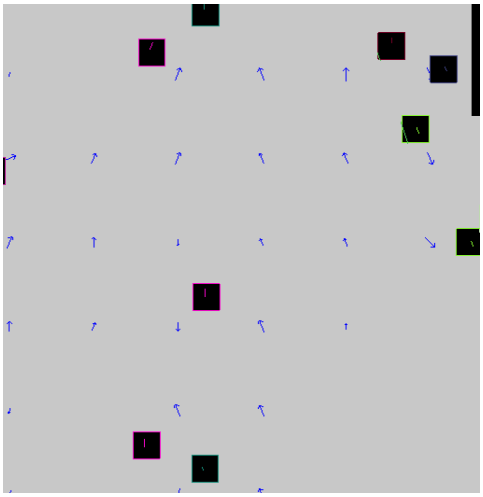

Figure 3.11: The blue arrows represent the direction field. Their intensity (so their length) decreases with time. Agents are represented by squares.

3.3.2 Density

On the one hand, agents have to be able to avoid getting bogged down on a tile because all the cases they check for lead to a deadlock. On the other hand, they have to adopt some human-like behaviours such as queuing when the density of people around them is really to high. These two abilities are quite contradictory: the more buoyant agents are, the less probably there may be a standstill in their movements. But, neither may they simulate a queueing behaviour. The trade-off actually relies on density. Besides, in reality, one has a very different behaviour if the traffic is congested or not: if the density is quite high, one do not try to overtake permanently; on the contrary, one adapt one's linear speed to the flow. That is why we make the agents follow different behaviour patterns in case the density is above or underneath a certain critical value.

Unfortunately, even with 10,000 people in our virtual city, there are still great discrepancies of density according to the region of the model. So, it is necessary to deal with local rather than global density. Computing the local densities in real-time could be really time-costing. Hence, we make an approximation relying on the fact that we can predict quite accurately the local densities of agents according to the **local density of goals**. In our simulation, we noticed that the higher the density of goals is, the higher the local density of people is. Using a low resolution map (that is a large tile

grid), local densities are pre-computed and stored into an array. Figure 3.12 shows how the approximation fits with the real agents density. In (d), we can see that the difference between the estimation of the density based on the number of goals (showed in (c)) and the pedestrian density measured during the simulation, is constant for the whole area of the virtual city, described by the static map (a).

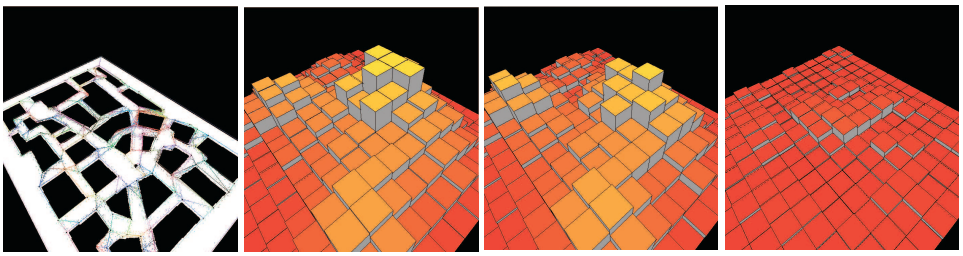

Figure 3.12: Local density assessment: (*a*) shows the global view of the city with 10,000 agents; on (*b*), we can see the human local average density computed during the simulation; (*c*) shows the prediction of local density according to the goals local density; finally, (*d*) represents the absolute value of the difference between (*b*) and (*c*).

The big tiles defined by the density map are said to be congested if the local density of goals is higher than a critical value. On congested tiles, agents adopt a quieter individual behaviour. These differences of behaviour are precisely described in section 4.3.2.

3.4 Group behaviour

The issue of making people walk in small groups has two main goals: the first one is obviously to improve the reality of the simulation. Owing to observations in the street, one can conclude that around half of the pedestrians walk by two or more. Moreover, using groups could be a mean to reduce the computing time for the same number of agents: indeed, the definition of a group [9] is often given by a set of people who have the same goals or list of goals and the same emotional parameter, that means, in our case, the same average speed and way of accelerating. Hence, one agent of the group only , that we will call the *Leader*, has to have its trajectory computed as before; the other agents, called *Members*, just have to follow it. Therefore, providing the number of operations necessary to compute the *members*' behaviour if

smaller than the *leader*'s , the average frame rate of the simulation is likely to increase.

3.4.1 Size of the groups

In real situations, the size of pedestrians groups is scarcely bigger than 3. However, tourist groups are quite often composed of around 20 people. These differences are though often leveled due to the density of the traffic. It is quite scarce to see the ten people of a group walking on the same line; the group is more likely to split into smaller sub-groups of 2 or 3 pedestrians, each sub-group following the other. That is why there is no group size bigger than 10 in our simulation. The repartition of the group is initialised randomly following the probability law presented on the Figure 3.13: 95% of the groups have a size of 1 or 2; 5% have a size between 3 and 10. Consequently, if the *member* moving method computation is far less time-consuming than the *leader*'s, we can save nearly **half of the computing time** spent for human behaviour.

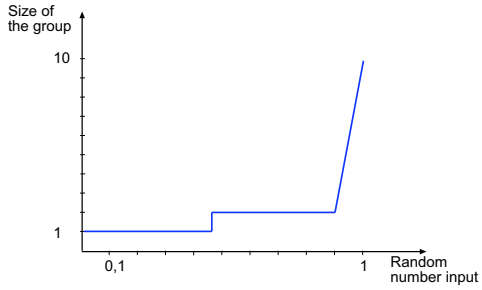

Figure 3.13: Probability law of the group size. One random number is generated by the computer and this graphic provides the corresponding size of the group.

3.4.2 Group motion

To make the group move in a time-spending way, we use the following method: the *leader* of a group always moves first. Once it has moved, it sets 3 tiles behind it with an *advised tile* tag. Then, the *members* of the group move one by one in the same frame: they have to move to an *advised tile*. In order to have enough *advised tiles* for the whole group, every member, after moving, advises the 3 tiles behind it with the tag. When

every member has moved, all the *advised tile* tags are deleted. Figure 3.14 illustrates this method.

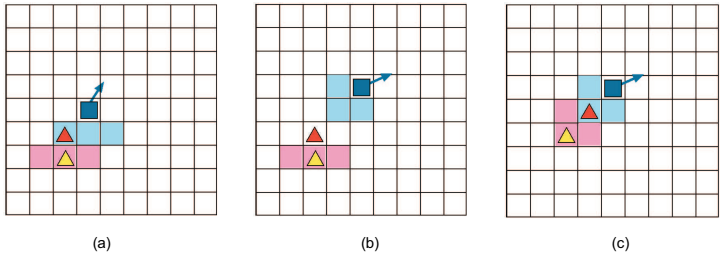

Figure 3.14: Group moving method: on (*a*), the square is the *leader* and there are two *members* (triangles); the *leader* advised 3 tiles (light blue) and so did the first member (light red). On (*b*), the *leader* has moved and its advised tiles as well. On (*c*), the two *members* have moved to keep the connectivity of the group. The second *member* (yellow) could have chosen either the blue or the red advised tiles.

However, the point is that a group has to remain **connected** for the method to work: at each frame, for every agent of the group, one of their neighbour tile has to be occupied by another agent of the same group. Otherwise, in case a *member* loses the connection with the rest of the group, it does not any more *advised tiles* to move to and is likely to stand still. Unfortunately, this constraint of connectivity brings about an unrealistic behaviour of the members of a group. When the *leader* turns sharply, the positions of the advised tiles are most probably going to change a lot so that most *members* would have to "jump" from one tile to another. Moreover, *members* tend to follow each other, rather than moving aligned, which gives the user the feeling that the *members* are more children or pets than adults.

The development of this method is at an early stage and we are aware of its limitation. This is the last contribution brought to the algorithm and we did not have enough time to improve it. However, we think that there must be a way to cope with these awkward jumps.

3.5 discussion

On the one hand, this chapter depicted the way the agent-to-agent collision avoidance is performed using a ray tracing method. But in order to keep simple enough local laws, we have introduced two main global entities: the goals and the flows. The goals, computed thanks to a corner detection

method, are linked into a visibility graph. They will enable agents to have straight trajectories. Thanks to their repartition, the pedestrian density is far from being uniform. Besides, the higher densities correspond to a deep integrated (regarding space syntax terminology) place. As for the flows, they are emphasised using a direction map and they bring about the streaming behaviour. We have also found a potential method to create a group behaviour which would save computing time.

The next chapter deals with implementation details and aims at underlining the performance and the weak points of our algorithm.

Chapter 4

Implementation, results

4.1 General presentation of the algorithm

The algorithm is implemented in $C++$ using the *OpenGL* library. It is based on the interactions between a *Crowd* class and different *Maps* (see on Figure 4.1).

A map is defined by an array and a resolution: the bigger the resolution, the smaller the size of the city area matching with one cell of the array. The three main maps make up three layers, each one of them triggering different kinds of behaviour for the *Crowd*. The first map is the *static map* and contains indications for the agents to know where they have the right to go: it corresponds to the map related in section 3.1. Only the pavements and the pedestrian crossings are accessible areas. Second, the inter-agent collision detection is performed using the *human map* that stores the positions of every human (see section 3.2). The third map stores directions and enables the streaming (see in section 3.3). These three maps have the same size of 512 by 512. One more less important map has a smaller resolution: this is the map of density (section 3.3.2), whose size is 30 by 30 tiles. The last significant structure is the graph of goals also presented in section 3.1. It is implemented with an array indexed by the goals, and stores the lists of neighbours (visible goals).

An interface has also been implemented. It offers several possibilities of parametrisation, display and result processing. Different city models can be loaded. The graph of goals can be computed or loaded to save pre-processing time. The main human parameters such as their number or their default speed can be modified. The display is in three dimensions and enables the user to track every human and their trajectory can be drawn. This is a really

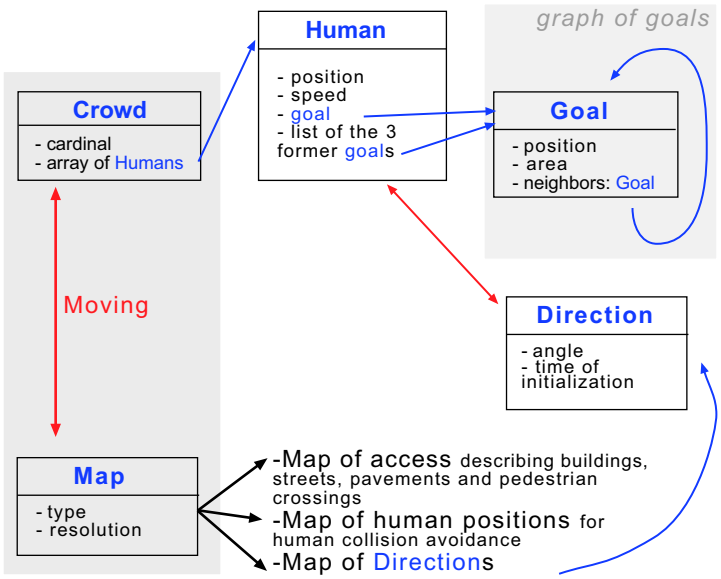

Figure 4.1: Architecture of the program: the blue arrows show the structure dependencies whereas the red ones indicates strong interactions during the execution of the program.

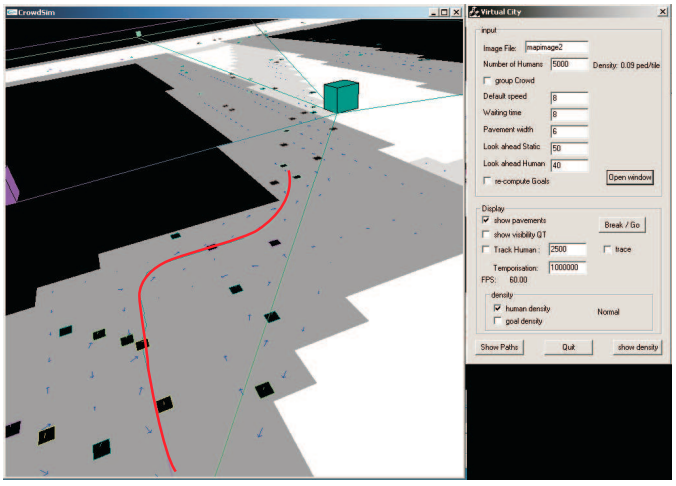

Figure 4.2: Graphical user interface of the program: on the left window, we can see the agents (squares) moving on a pavement. One of them has its trajectory displayed. The cube illustrates a goal.

useful tool to judge if the way they move is realistic or not. The interface is presented on Figure 4.2.

4.2 Initialisation

The initialisation concerns two actions: the precise analysis of the placement of the goals and the initialisation of the positions of pedestrians.

4.2.1 Corners detection and merging for the goal graph

The difficulty in detecting corners is that the corners to detect do not have any particular properties: they are not supposed to be right angles or horizontal or vertical oriented. We present here the details of the method explained in section 3.1.4: the following algorithm(4.1) is called after having built pavements around buildings. There are three constants used to describe the city called BUILDING, PAVEMENT and GROUND (part of the streets that are not accessible for agents). These constants relates to the ideas developed in section 3.1.

It is important to precise that the two cases of acute and obtuse corner cannot occur with the same tile (i, j). Moreover the bigger the variable c is, the sharper the angle is. The limit of 1 and 5 have been set experimentally.

As this method detects on purpose more corners than necessary, we have to merge the closest ones while conserving connectivity between the goals. This method is presented in the algorithm (4.2). The constant ε refers to the radius below which, two goals are merged into a new one. This threshold is set experimentally.

Actually, the distance between the two goals is computed just once before the execution of this algorithm and we prefer dealing with the squared distance (it is less time-consuming). The mean used for computing the new position of the merged goal does not match with the barycentre if there are more than two goals merged together. However, it gave good results.

4.2.2 Initialising agents' positions and goals

Before starting the simulation, we need to initialise the position and attributes of every virtual human.

The speed is a characteristic of each human. Though, there are several possible **representations of the speed vector**: for instance, cartesian or

Algorithm 4.1 Corner detection

```
For each tile(i,j) on the PAVEMENT do
    //detection of obtuse corners
    c ← 0          //c represents the accuracy of the corner
    For each direction k around (i,j) do
        If ( (i,j)+k⃗ and (i,j)-k⃗ are on the GROUND ) then
            c ← c +1;
        End if
    End for
    If ( 1 < c < 5 ) then
        set Corner (i,j)          //we have found a corner
    End if

    //detection of acute corners
    c ← 0
    For each direction k around (i,j) do
        If ( (i,j)+k⃗ and (i,j)-k⃗ are on BUILDING ) then
            c ← c +1;
        End if
    End for
    If ( 1 < c < 5 ) then
        set Corner (i,j)          //we have found a
    End if
End for
```

Algorithm 4.2 Goal merging

```
For every pair (a,b) of not deleted goals do
    If ( distance between a and b ≤ ε ) then
        neighbour(b) ← neighbour(b) + neighbour(a)
        position(b) ← mean(position(a), position(b))
        delete(a);
    End if
End for
```

polar. As the most common way to use the speed (in our simulation) is to add it to the current position to get (and test) the new position, the best way to represent it in memory is a priori with cartesian coordinates. In that way, rotations are performed by multiplying the speed vector with a pre-computed matrix. However, this is quite inefficient a method for real-time computations purposes. In addition, we often have to assess the angle between two vectors (for the human collision avoidance, for example). Using a cartesian representation, it is quite heavy to compute this angle without computing a squared root. Thus, we prefer using an **indexed polar representation** of speed vectors presented on Figure 4.3. The different values of angles and norms are indexed on a range of sixteen values stored in arrays. The corresponding sines and cosines are also pre-computed and stored in arrays, which enables a fast computation (without cosines and sines) for the new position. With this method, we replace two float coordinates by one byte (16 angles, 16 norms, ie 256 possibilities) and every computation on speed is simpler or faster.

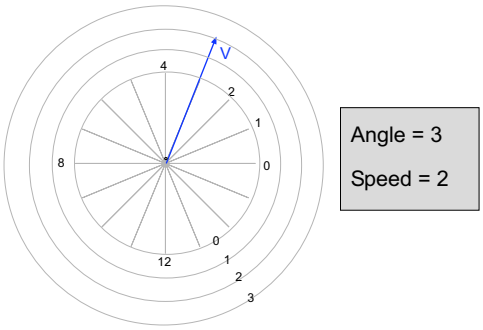

Figure 4.3: Indexed polar representation of speed: there are 16 angles and 16 norms values available.

Before the simulation, every human has to have their position and first goal initialised. These two features are linked because the agent has to be able to reach its goal from its first position: in other words, the initial goal has to be visible from the initial human position. To perform this task, we use the fact that, between two connected goals, there is always a pavement; this pavement part is in addition more or less rectangular-shaped. It is therefore quite easy to add an agent on this particular pavement. Insofar as it stands between two visible goals at least, the agent can take any one of them as a first goal (see algorithm 4.3).

Algorithm 4.3 Initialisation of position and goal

```
agenti ← 0                    //number of the agent
While agenti < cardinal of the crowd do
    For every pair (a,b) of connected goals do
        find a random position for agenti in the rectangle
        whose extremities are a and b
        set the goal of agenti with a or b
        agenti ← agenti +1
    End for
End while
```

4.3 Static and dynamic obstacles

The behaviour of an agent relies on four components: avoiding static obstacles, reaching a goal, following a flow and avoiding the other agents. We have to find a way to combine them in a function that gives valid direction and speed to follow. We chose to deal with static entities first and with agent-to-agent collision afterwards.

4.3.1 Static obstacles

Buildings, streets, goals, and flows belongs to static obstacles. We use them to get an advised direction for agents. Indeed the method used here is inspired by Feurtey's cost function of different contributions (see in Section 2.3.1). Our aim is to try every direction around the agent, to check each one of them according to criteria and to take the best one as a new direction. In this framework, we have to know how to give a **mark** to a direction concerning goal searching, flow following and static obstacle avoiding. Every mark is out of 100.

First, giving a mark concerning the **static obstacle** avoidance was not too difficult. Given a trial direction, we check if the next tile defined with this direction is accessible or not. If it is, the mark given is 100, otherwise it is 0. This assessment can be modulated by taking the further next tiles into account. Actually, the 5^{th} next tile's accessibility is checked as well, which modulates the mark. Moreover, this ahead-checking avoids any perpendicular collision into a wall and contributes to the smoothness of the trajectory.

The **flow-following** mark is based on the angle difference between the direction stored into the next tile and the trial direction of the agent. The indexed polar speed representation presented in section 4.2.2 is here particularly handy. The two speed vector angles are integer values between 0 and 15. Making the difference between these two values enables to give a mark, whose bounds are 0 if the two directions are opposed and 100 if they are the same. Sixteen other intermediate marks correspond to other angles.

A direction is a good direction if it moves the agent closer to its **goal**. Thus the issue of **distance** appears. There are several distances that have a great advantage upon the euclidian distance: it is quite long to compute because of a square root. Now, this computation is not only performed for every agent and every frame, but it is even done for several trial directions at each time. As the method has to be optimised for our kind of applications, we cannot afford using the euclidian distance. Moreover, in order to give a mark, it is necessary that we deal with a bound entity. Using the euclidian distance, we had the inequality:

$$\left|Dist\left(p+\overrightarrow{D_{try}}, Goal\right) - Dist\left(p, Goal\right)\right| \leq MaxSpeed \tag{4.1}$$

where p is the position of the agent, $\overrightarrow{D_{try}}$ its trial direction, *Goal* the goal it wishes to reach, and $MaxSpeed$ the maximum linear speed value for an agent, that is the maximum distance it can cover in one frame. For any distance D, we define:

$$\Delta_D(A,B) = \left|D\left(p,A\right) - D\left(p,B\right)\right| \tag{4.2}$$

with the same notations. Thanks to the former inequality, it is possible to give a mark between 0 and 100 for every direction. Besides, another distance has the same property:

$$d_1(A,B) = Max\left(\left|A_x - B_x\right|, \left|A_y - B_y\right|\right) \tag{4.3}$$

where A and B are two points and A_x, A_y, B_x, B_y their cartesian coordinates. Using distance d_1 would save a great amount of computing time. Nonetheless, it has the property of preferring diagonals: the value of $\Delta_{D_1}(A,B)$ is the greatest when the vector $\overrightarrow{AB}$ is oriented as a diagonal of the plane. Figure 4.4 shows the typical trajectories given when d_1 is used. The trajectories given by this distance are not really realistic insofar as every agent has the tendency to follow the wall (which is likely to cause traffic jams) whereas there is a large free area a bit aside from it.

Obviously, the best thing to do was to use the squared euclidian distance (hence avoiding the square root). Nevertheless, the $\Delta_{SquaredDist}$ obtained

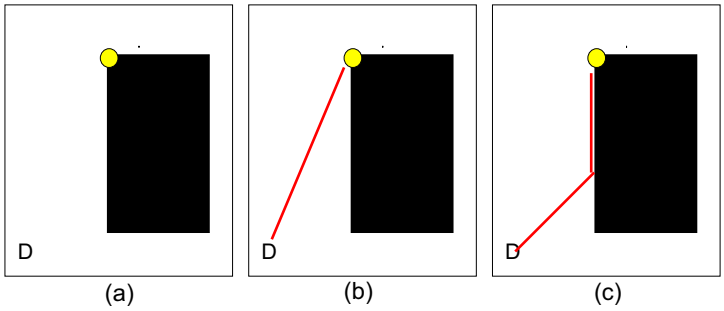

Figure 4.4: On (a), a building, a goal (yellow circle) and the departure position (D) of the agent is represented; on (b), we can see the trajectory in case the distance used is euclidian; (c) shows the trajectory if we use d_1.

with it is not bound any more: the more the agent is far from its goal, the bigger $\Delta_{SquaredDist}$ will be. In order to normalise $\Delta_{SquaredDist}$ without computing a square root, we use the following approximation:

$$\frac{\Delta_{Dist^2}}{v_{agent}} \approx \Delta_{Dist} \tag{4.4}$$

where v_{agent} is the linear speed of the agent.

The three marks obtained are then combined into a mean and different directions can be compared. The direction that has the best mark becomes the **advised direction**.

4.3.2 Dynamic obstacles

For an agent, dynamic obstacles are the other agents. As seen in section 3.3.2, the ray tracing method used to perform human collision avoidance is different according to the local density. The two methods are presented in Figure 4.5 for normal density and in Figure 4.6 for high density and are explained in the following. As one tile of the map used for agent-to-agent collision detection is approximately $1m^2$ wide, the critical density value is set at $1,5m^2/agent$ according to the work of Feurtey [6]. In any case, a ray is traced from the agent position in the direction of its speed. If the ray does not intersect any occupied tile after 10 tiles crossed, we consider that there is no collision to avoid. Otherwise, the ray is intercepted and the distance between its interception and its origin is filed in one of the distance ranges: *Close*, *Near* and *Far*. The agent's reaction is not the same according to this range.

Distance	Behaviour of the other	Type of collision	Reaction
Close	Waiting	Any	If possible, avoid the other by turning $\pm\pi/4$, $\pm\pi/2$ Else, slow down or wait
	Walking	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/4$, $\pm\pi/2$ Else, slow down or wait
		$\uparrow\uparrow$	Take the same linear speed as the other
		$\perp$	Wait
Near	Waiting	$\uparrow\uparrow$	Slow down
		else	If possible, avoid the other by turning $\pm\pi/4$ Else, slow down or wait
	Walking	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/4$ Else, slow down or wait
		$\uparrow\uparrow$	Take the mean of both linear speeds as a speed
		$\perp$	Slow down or wait
Far	Waiting	$\uparrow\uparrow$	Slow down
		else	If possible, avoid the other by turning $\pm\pi/8$ Else, slow down or wait
	Walking	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/8$ Else, slow down or wait
		$\uparrow\uparrow$	
		$\perp$	Slow down

Figure 4.5: Free-flow collision avoidance: This table summaries the algorithm of agent-to-agent collision avoidance in case of a free-flow traffic. The symbol $\uparrow\downarrow$ means that this is a front collision; the symbol $\uparrow\uparrow$ means that the considered agent follows the other; finally, $\perp$ means that the other agent crosses perpendicularly the way of the agent.

Distance	Behaviour of the other	Type of collision	Reaction
Close	Waiting	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/4$, $\pm\pi/2$ Else, slow down or wait
		$\uparrow\uparrow$ or $\perp$	Wait
	Walking	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/4$, $\pm\pi/2$ Else, slow down or wait
		$\uparrow\uparrow$	Take the same linear speed as the other
		$\perp$	Wait
Near	Waiting	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/8$, $\pm\pi/4$ Else, slow down or wait
		$\uparrow\uparrow$ or $\perp$	Slow down sharply
	Walking	$\uparrow\downarrow$	If possible, avoid the other by turning $\pm\pi/8$, $\pm\pi/4$ Else, slow down or wait
		$\uparrow\uparrow$	Take the same speed
		$\perp$	Slow down or wait

Figure 4.6: Congested traffic collision avoidance: This table summaries the algorithm of agent-to-agent collision avoidance in case of a congested traffic. See on Figure 4.5 for the meaning of symbols.

There are two main differences between the **congested and normal traffics**: first, the *Far* range does not have any significance unless the traffic is normal. Second, agents turn more sharply when the traffic is congested.

On point not illustrated in the figures could cause two agents to get stuck for ever. In case of a perpendicular collision, it is not enough to wait until the other agent has changed tile. With the discretisation of space, it may be that two agents standing on two nearby tiles have perpendicular speed vectors and still both want to move on the tile of the other (see on Figure 4.7). With the two presented algorithms, one would wait for the other to move and so would the other do. To solve this problem, the waiting state is followed by a small rotation of the speed. Thus, the situation will not lead to a deadlock. Another point that needs to be developed is that for some perpendicular collisions where the agent does not wait, it always tries to avoid the other on its backside.

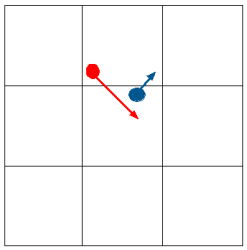

Figure 4.7: Case of positions and speed where both agents want to move on the tile of the other.

4.3.3 Results when considering by both static and dynamic obstacles

In the algorithm, we first run the method that gives the agent an advised direction according to the static entities (section 4.3.1), and then this advised direction is modified using the algorithms of section 4.3.2.

We assessed the performance of the algorithm in two different ways. First, we measured the **flow of pedestrians** in a particularly busy street of the model. Figure 4.8 and Figure 4.9 present the relations between the flow of pedestrians and the number of pedestrians (proportional to the density). We can see that, without agent-to-agent collision detection (Figure 4.8), the flow is proportional to the density, which is quite in opposition with Feurtey's observations [6]. But, in this case, nothing prevents the agents from walking on the same tile. The second graph, however, is in agreement

with Feurtey's observations: it shows that, when the agent-to-agent collision detection is performed, the flow reaches a maximum for a certain value of the density. This gives evidence that the way we have implemented the crowd behaviour reflects realistic motion.

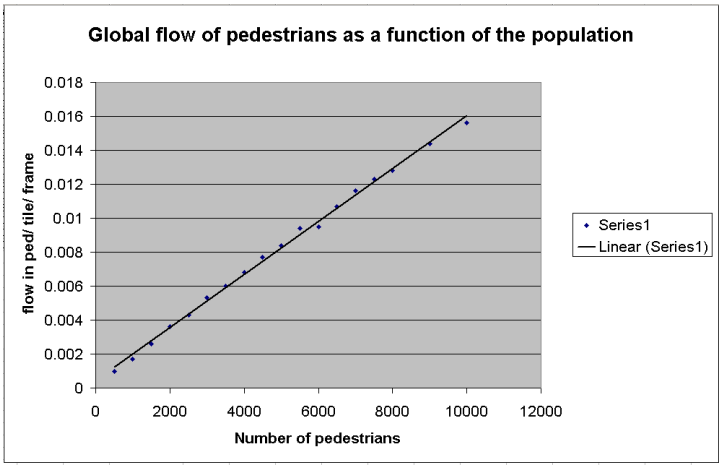

Figure 4.8: The flow was measured for every agent while no agent-to-agent collision detection was performed.

The second way to assess the efficiency of the simulation is to observe **agent trajectories**. On Figure 4.10, we can compare the results given by the algorithm when the agent-to-agent collision avoidance is performed or not. In this particular case, the trajectory does not seem to be too much affected: the other agents seem to be avoided in a very smooth way.

4.3.4 Further behaviours

The three previous sections essentially deal with the modification of the direction of an agent. But the **linear speed changes** are also a big issue concerning the realistic appearance of the behaviour. Thus, for instance, a turn is always followed by a decrease of the linear speed. The sharper the turn is, the more the agent slows down.

However, so far, no kind of behaviour lasts more than one frame. This would be a way to give the simulation more coherence. For example, when an agent has to stop because of a perpendicular collision, this would be better if the stop could last more than one frame. To that extent, the variable **Behaviour** is a new member of the class Human.

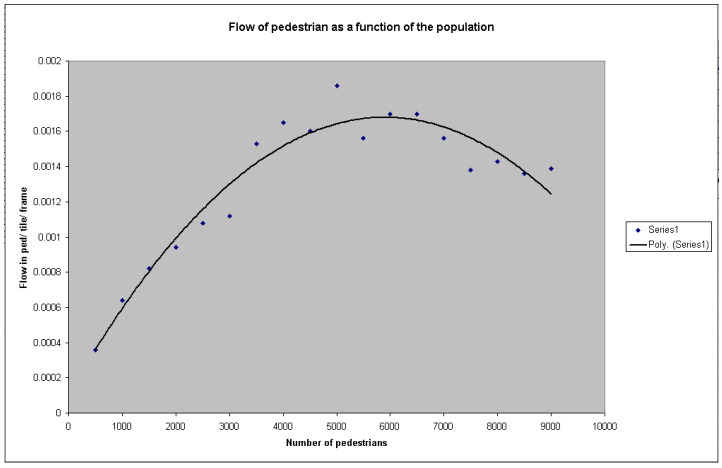

Figure 4.9: The measures have been done in a busy street and every pedestrian involved was performing agent-to-agent collision detection.

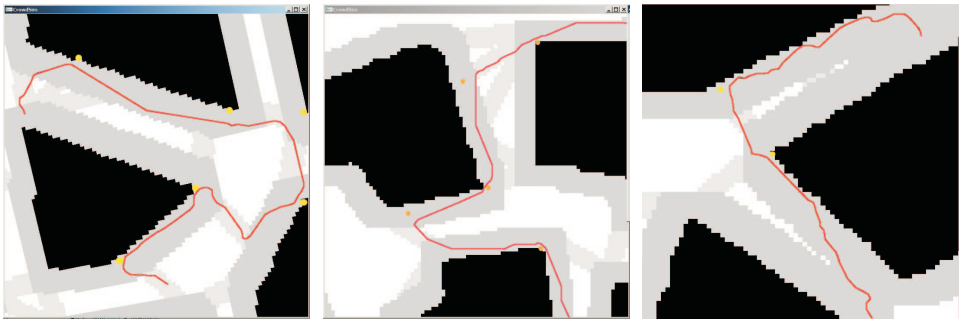

Figure 4.10: On (a) and (b), we can see the trajectories followed by an agent without being seen by the camera (that is without agent-to-agent collision detection performed) during simulation involving respectively 10,000 and 1,000 agents. On (c), an agent has been tracked (the agent-to agent collision detection was activated) in a busy street; the simulation was running with 5,000 agents. We can see detours due to collision avoidance with other agents.

The first **lasting behaviour** (different from NORMAL) is therefore WAIT. There are several available values for WAIT, each one corresponding to a waiting time. When an agent's behaviour is set to WAIT, it just decreases its WAIT variable without moving. When the variable WAIT is 0, the *Behaviour* is back to NORMAL. The agent is able to continue its route with the same direction as before stopping. Though, to avoid some infinitely repeated WAIT because of external obstacles, the WAIT *behaviour* is often performed in parallel with a small rotation of the direction of the agent.

Since agents may slow down, they have to be able to accelerate. ACCELERATE is a *Behaviour* that is switched on only if there is nobody to intercept the ray traced during the collision avoidance method. Then the indexed value of the linear speed can be increased by 1. But there are only three ACCELERATE states, so that the acceleration may remain realistic. As soon as somebody appears ahead of the agent or as soon as it turns, the *Behaviour* is back to NORMAL.

Despite all these efforts that have, without any doubt, improved the reality of the simulation, there is still a problem linked with the space discretisation. Although the discretisation is really the most important feature of that method, it has some edge-effects: two agents may be on two different tiles and, though, really close the one from the other. This means that in spite of all our work, the user still sometimes has the impression that a collision has occurred (see on Figure 4.11).

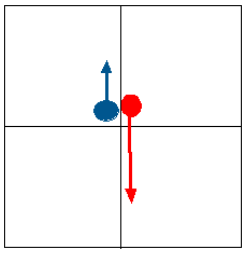

Figure 4.11: Edge effect due to the discretisation: the two agents are undoubtedly too close the one from the other whereas none of our rules has been broken.

4.4 Improving the algorithm efficiency

The computer on which the program was run is a PC with a 2GHz Pentium 4 processor and 512MB of RAM. The graphic card that deals with the 3D

polygon display, is a NVIDIA GeForce Ti 4600.

However, we still had to find methods to ameliorate the simulation rapidity. The first one is a **multi-level direction search** used in the assessment of static obstacles that aims at giving an advised direction (section 4.3.1). It is not really useful to compute a mark for each one of the 16 available directions. We prefer doing several passes according to the mark obtained by the former direction (see in algorithm 4.4)

Algorithm 4.4 Multi-level direction searching

```
trialDir ← formerDir
If ( Mark(trialDir) < c1 ) then
    compare the marks of trialDir ±π/8
    If ( both marks are less than c2 ) then
        compare the marks of trialDir ±π/4
        If ( both marks are less than c3 ) then
            WAIT
        Else
            take the best mark direction
        End if
    Else
        take the best mark direction
    End if
Else
    take this direction
End if
```

The three constants c_1, c_2 and c_3 have to be well chosen to optimised the performance of the algorithm. Our choice of the constant enables around 50% of the agents to go straight, that is to take their former direction as a new direction, and thus to save computation time.

The second main improvement is the **multi-resolution collision detection** already mentioned in Musse et al.'s work [9]. It relies on the fact that, when the viewpoint is far enough from an agent, the user cannot see if an agent-to-agent collision occurs or not. Thus if an agent is farther enough from the viewpoint or outside the range of view of the camera, it is said to be not visible (as in a frustum culling method). The agent-to-agent collision detection presented in 4.3.2 is thus performed only if the agent is visible.

Rather than computing the visibility of every agent, we use an approximated visibility map, that is a low resolution array (such as the density map) that states whether the corresponding area of the city is visible or not from the viewpoint. This map is updated for every movement of the camera in a constant time seeing its small size. Thus, the agent-to-agent collision avoidance is actually scarcely performed because, in case the user is tracking one agent, only some hundreds of agents are likely to be visible (even if there are 10,000 agents).

A way to judge the efficiency of our algorithm is to measure the number of frame per second (fps). Ideally, it should be of 24 fps for an optimal fluidity of the animation. Figure 4.12 shows that the fps decreases with the number of pedestrians. The animation speed is greater than 24 frames per second up to 6,000 simulated agents.

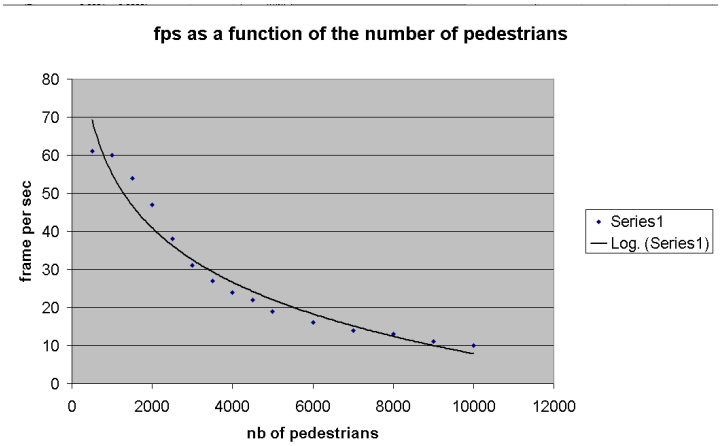

Figure 4.12: The number of frame per second decreases less and less with the number of pedestrians. Up to 6,000 agents, the animation remains quite fluid.

4.5 Integration of the pedestrian behaviour in the previous city model

Our work is integrated in the previous 3D city model presented in section 2.1. Thanks to the rendering optimisations of this previous model, the simulation is fluid up to 7,000 pedestrians. Human textures, shadows and 3D buildings obviously make the pedestrians movement more realistic (see on Figure 4.13).

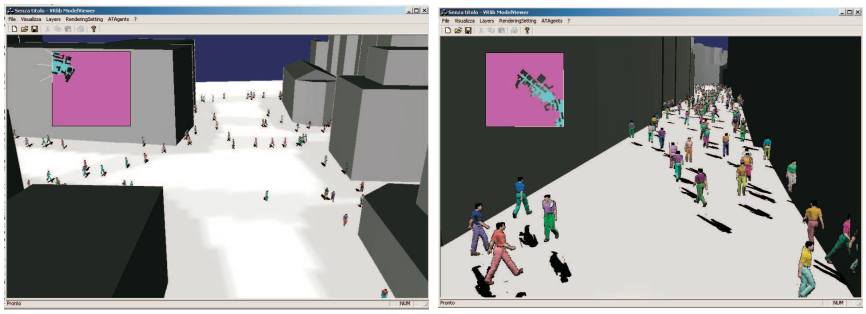

Figure 4.13: Example of simulation of our work integrated in the previous 3D city model presented in section 2.1. The simulation runs with 7,000 pedestrians.

Though, this integration makes us aware of the difficulty of carrying out such a big project. The different libraries built from the different contributions are not always full-compatible. It also emphasises the necessity of coding in a really understandable way so that anyone may be able to correct an other's code.

Chapter 5

Future work and conclusion

In this report, we improved an agent-based method to simulate hight density crowds behaviours in virtual city environments. The main asset of this project is that the simulation is really realistic concerning global behaviours such as streaming. This was performed by using an array storing a local direction, updated by the agents, that affects the speed of the nearby agents. In addition, the method used to set and attribute goals to the agents is quite effective too insofar as the pedestrian density is far from being uniform, as in space syntax modeling. This method also enables the automatic definition of pavements and pedestrian crossings that constitute the structure of pedestrian areas. Thanks to the consecutive processes of static and dynamic obstacles, individual behaviours have been well improved too, seeing the straight and smooth trajectories of the pedestrians.

These three points (streaming, respect of the city structure and smooth trajectories) lie really at the root of a realistic appearance of the pedestrian crowd movement. Though, the main drawback of the simulation, which is in the same time what brings its performance, is the discretised space: whereas it enables a fast and efficient collision detection and a trade-off concerning memory costs, the user may see some agents too close from other ones (still on two different tiles).

We have proposed a fast method to simulate the group behaviour which would enable great time savings. Unfortunately, its early stage of development seems to prevent it from being really realistic. We clearly need a group method that, for instance, would render in a smooth way two persons walking together.

A great number of other interesting features could be implemented. Among them, agents sources and wells would enable the modeling of build-

ings way-in and out (such as a shop entrance). As we managed to build pavements and pedestrian crossing automatically, it is also quite attractive to integrate moving cars with their self behaviours. Finally, a multi-resolution platform could be developed, in which the level of detail would vary with the distance from the view point. For example, agents could begin to simulate a dialog or shake hands when the camera is close enough. Some interactions with the user could even be implemented.

This kind of project could have a big influence on the curing method of certain phobia. Indeed, many virtual reality laboratories are equipped with a CAVE (Cave Automatic Virtual Environment). In a CAVE, the user stands into a cube, on 5 faces of which the 3D environment is displayed. The stereo-vision glasses and the head tracker worn by the user contribute to give him a deep sensation of reality. Besides, the reality of the simulation is so good that, even if the user knows that this is only virtual, his first physical reactions (vertigo's in case of a cliff displayed; stress in case he stands in front of an assembly looking at him) are the same as if it was real. This means that such a tool may be used to therapeutic ends. For example, the development of populated virtual cities is very interesting according to agoraphobia specialists: a person suffering from agoraphobia could be cured by practicing crowd contacts into a virtual environment. By changing some parameters, such as the colour of the light, the density of agents or the kind of environment (shopping centre or street), and thanks to the patient's reactions, it would be easier to find the exact phobia the patient suffers from.

Bibliography

[1] Ruth Aylett, Marc Cavazza: Intelligent Virtual Environments- A state-of-the-art report, 2001.

[2] Franco Tecchia, Yiorgos Chrysanthou: Real-time visualisation of densily populated urban Environments: a simple and fast algorithm for collision detection, *Eurographics UK*, April 2000.

[3] Franco Tecchia, Céline Loscos, Ruth Conroy, Yiorgos Chrysanthou: Agent behavior simulator (ABS): a platform for urban behavior development , *Proc. Game Technology* (GTEC 2001),CD-ROM, 2001.

[4] Franco Tecchia, Céline Loscos, Yiorgos Chrysanthou: Image-based crowd rendering, *IEEE computer graphics and applications*, March/April 2002, p. 36-43.

[5] Koji Ashida, Seung-Joo Lee, Jan M. Allbeck, Harold Sun, Norman I. Balder, Dimitris Metaxas, Pedestrians: creating agent behaviors through statistical analysis of observation data, 2001.

[6] Franck Feurtey, Takashi Chikayama, Simulating the collision avoidance of pedestrians, February 2000.

[7] Bin Jiang, Christophe Claramunt, Björn Klarqvist, An integration of space syntax into GIS for modelling urban spaces, *JAG*, Volume 2, Issue 3/4, 2000.

[8] Soraia R. Musse, Christian Babski, Tolga Capin and Daniel Thalmann, Crowd Modelling in Collaborative Virtual Environments, *ACM VRST'98*, Taiwan,1998.

[9] S.R. Musse, D. Thalmann, a model of human crowd behavior: group inter-relationship and collision detection analysis, *Proc Workshop of*

Computer Animation and Simulation of Eurographics'97, Sept 1997, Budapest, Hungary.

[10] C.W. Reynolds, Steering behaviours for autonomous characters, *Game Developers Conference*, Miller Freeman Game Group,1999.

[11] Bouvier, Cohen, simulation of human flow with particles systems, 1994.

[12] http://www.equator.ac.uk/

[13] F. Tecchia and Y. Chrysanthou: Real-Time Rendering of Densely Populated Urban Environments, *Eurographics Rendering Workshop 2000*, p. 83-88, Springer Computer Science, 2000, Rendering Techniques 2000.

[14] Céline Loscos, Franco Tecchia, Yiorgos Chrysanthou: Real Time Shadows for Animated Crowds in Virtual Cities, *ACM Symposium on Virtual Reality Software & Technology 2001 (VRST01)*, Banff, Alberta, Canada, November 2001.