Centro de Investigación en Matemáticas

CIMAT

Tésis

para obtener el grado de

Maestría en Computación

presentada y defendida públicamente el Miércoles 17 de diciembre del 2014

Optimization-based Computation of Locomotion Trajectories for Crowd Patches

José Guillermo Rangel Ramírez

Jurado

Dr. Julien Pettré (INRIA-Rennes), Lector Especial y Co-Director de la Tesis
Dr. Rafael Eric Murrieta Cid (CIMAT), Presidente
Dr. Jean Bernard Hayet (CIMAT), Secretario
Dra. Claudia Elvira Esteves Jaramillo (UG), Vocal y Co-Director de la Tesis

Abstract

Populating large virtual worlds has become an important subject of study in Computer Science, due to its vast number of uses in the entertaining industry. There are techniques to animate these virtual environments that produce results with great quality, but they require high computational power that usually can not be provided by a standard computer. The Crowd Patches technique, first proposed by Yersin et al. ([Yersin et al. 2009]), finds a balance between performance and quality, by creating large environments composed of smaller low cost regions of animation connected together. The trajectories being followed by the virtual characters in each of those smaller regions are crucial to the realism of the scene. For this reason, the purpose of this thesis is to increment the quality of the generated trajectories. Specifically, an optimization-based approach to generate smooth collision-free trajectories for Crowd Patches is proposed.

The proposed approach is divided into 3 main steps: (1) Connecting the endpoints of the trajectories, (2) handling collisions between them or fixed obstacles, and finally (3) applying a smoothing process. An extension to the approach in order to achieve some group walking behavior is also given. We show as results several examples of generated patches and how they are improved compared to patches generated with the previous techniques. Finally, we discuss some of the limitations of our method as well as some proposed future directions that research on Crowd Patches can take.

Dedico esta tesis a mis padres,
que desde siempre han estado apoyándome.

Acknowledgements

I would like to thank all the people who helped me make this work possible.

First of all, I would like to thank my advisors Dr. Claudia Esteves and Dr. Julien Pettré for accepting me as their student. Their knowledge was a compass that guided me when I needed it most. Thanks for all their patience, counsel, encouragement and for all their precious time invested in me. Their example of hard work and passion for the Computer Science will be forever engraved in my mind.

I'd like to extend my sincere gratitude to the people who helped adapting this work into a paper of the same name submitted to the ACM SIGGRAPH Conference on Motion in Games 2014, Dr. Panayiotis Charalambous, for his excellent advise and help with the writing, and student Devin Lange, for his fresh insight into the topic and beautiful figures. Their help was key for the paper's success.

Thanks to all my professors from CIMAT, all of them wise, and always willing to impart their knowledge as weapons to succeed in life. Specially, thanks to Dr. Jean-Bernard Hayet and Dr. Rafael Murrieta for agreeing into forming part of the jury and whose corrections and suggestions improved in many ways the work presented today.

I would also like to thank the "Consejo Nacional de Ciencia y Tecnología" (CONACyT) for providing me with a scholarship during my stay at CIMAT. Thanks also to the French Institute for Research in Computer Science and Automation (INRIA) for the extra resources they supplied during my stay in Rennes, France. My deepest thanks to the people at INRIA who helped me with the initial set up of the software used in this work and whom I had the pleasure to collaborate along my stay, Kevin Jordao and Tristan Le Bouffant.

To all the people who share with me their treasured friendship, I want to say thanks for all the necessary distractions they contributed. Unfortunately (but fortunately) too many to mention all, I will just mention the two of them who walked along the same path in life these past two years, Ángel Martínez and Ulises Tirado. Thanks for the healthy rivalry that inspired me to always give my best.

Lastly, I would like to thank my family for always believing in me, and the heart of it, my parents, Leticia and Martín, for their constant caring and support. They provided the best environment possible where I could write this work in the best state of mind.

Contents

1 Introduction

Over the years, animating environments with virtual people has become an important matter. Movies, video games, even TV programs rely more and more on realistically simulated or populated virtual worlds. But virtual simulations of people do not only have an entertaining aspect attached to them, they can also be used to simulate the evacuation of buildings or any other enclosed space. They can also be applied to increase the level of realism to existing objects: Imagine an online map where actual people could be seen walking, and where the number of people could depend on the time of the day and the day of the week.

The increment in computational power and rendering techniques has allowed video games to constantly display larger and livelier virtual environments. The recent game of Grand Theft Auto [Rockstar-Games 2013] is an example of this, taking place in Los Santos, a completely virtual city. Still, Los Santos remains relatively sparsely populated with virtual people. This phenomenon is due to the large computational cost needed to simulate *ambient crowds*.

To address this issue, the *Crowd Patches* technique has been recently introduced by Yersin et al. [Yersin et al. 2009]. This thesis is a direct continuation of that work. It is also for this reason that Yersin et al.'s paper is one of the main references of this thesis, and the majority of the material, including definitions and notations are originated from there.

This thesis generated a paper presented in the ACM SIGGRAPH Conference on Motion in Games 2014 [Ramirez et al. 2014]. Consequently, much of the material, such as tables or figures, occurs again here. It is important to say, this thesis includes longer explanations to the methods used as well as new techniques developed after the paper was written.

The Crowd Patches approach relies on creating small pieces of pre-computed periodic animation and then, respecting some boundary conditions, pasting them together to form a larger animation (Figure 1.1). By being periodic in time, this grants the ability of being played endlessly in time.

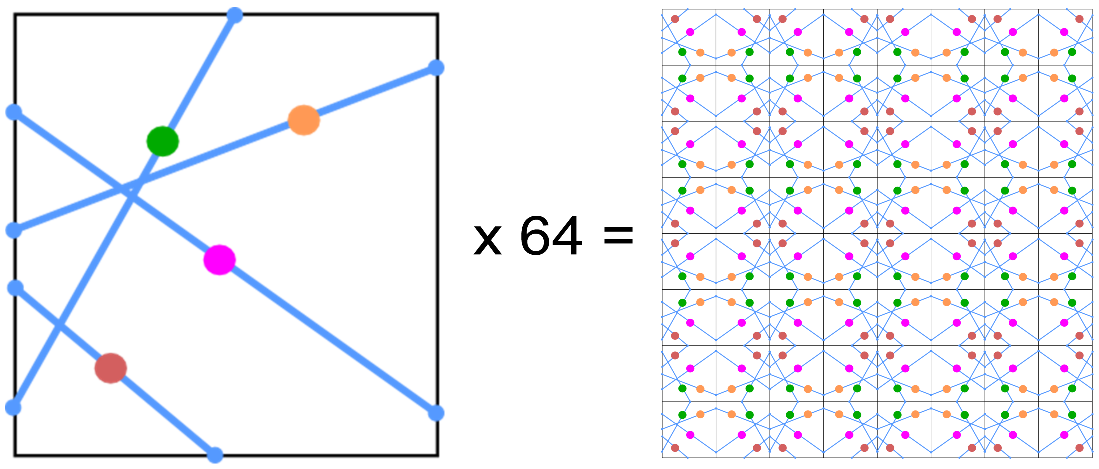


Figure 1.1: Crowd Patches Technique. To the left, a simple periodic piece of animation. The blue lines represent the trajectories being followed by the moving agents, in this case portrayed by points. To the right, a larger animation, conformed by 64 of the smaller pieces, original or mirrored, concatenated together.

The motivations for using the Crowd Patches approach are mainly three:

1. We want to be able to populate environments of any dimension. We want the domain of our approach to range from small areas, like public gardens or market squares, to very big ones, like villages, or entire cities, and potentially more.
2. Decrease computational resources needed to generate crowds relative to the whole cost of generating the virtual environment.
3. Increase the speed at which crowds are generated. A future goal is to be able to generate patches on the go, allowing to populate environments in real-time.

The continuity of the animation generated using crowd patches depends fundamentally in respecting boundary conditions between patches. These conditions allow characters to move between the patches and compose large crowds in a flowing motion.

This imposes a problem. Trajectories inside patches need at the same time to meet several conditions:

1. **Periodicity and continuity:** This will be elaborated later on in this work, but it suffices to say for now, that these conditions are met when trajectories pass over certain spatio-temporal control points (points in the patches with space and time coordinates) at the frontiers on the patch.
2. **Be free from collisions:** A collision happens when two agents walking along different trajectories get too close to one another, leading to complex interactions between them (usually crashing into each other).

3. Look natural: Last, but equally important condition, trajectories must look the most natural possible. Very convoluted, oscillating or leading to unrealistically fast displacements trajectories should be avoided. There will always be special cases when one of those trajectories will be needed, for example, outside a crowded bar, trajectories going in zig zag motion could be more real that more straight ones, but for the purpose of this work, trajectories will try to stick to going towards their goal (in this case, passing trough some boundary control points) with the least amount of detours possible; in other words, minimizing the number of collision avoidance maneuvers as much as possible.

To summarize, the main purpose of this thesis is to propose a new optimization-based method to compute internal trajectories for crowd patches meeting these conditions. The method receives as input an empty patch with the set of boundary control points, and the output generated is the set of trajectories connecting those points, free from collisions and smoothed out.

This thesis will be divided in the following chapters:

- **Previous Work.** In this chapter, several other techniques available in the literature to handle crowd simulation are discussed.
- **Definitions.** Formal definitions of a patch and its components are described in this chapter for a complete understanding of the methodology.
- **Methodology.** The main method for creating collision-free trajectories is described. Formal definitions of a patch and its components will be given, and then the main method is explained step by step. Some modifications to the algorithm are also explained, in order to add extra desirable features like walking in groups behavior.
- **Results and Comparison with Other Approaches.** The results obtained are reported in this section, as well as some of the advantages and disadvantages over other methods.
- **Future Work and Conclusions.** Final remarks, contributions and limitations of this approach are discussed in this final chapter.

2 Previous Work

Usually, virtual environments are populated based on simulation approaches [Thalmann and Raupp Muse 2013]. In simulation approaches, an ambient crowd is generated from a large set of moving characters, mainly walking ones; the individual characters follow some rules to determine their movement.

Reynolds presents a way of handling flocks of birds (which can be generalized for schools of fishes and herds) [Reynolds 1987]. In his work, birds in flocks are controlled by 3 main behaviors:

1. Collision Avoidance: avoid collisions with nearby flockmates.
2. Velocity Matching: attempt to match velocity with nearby flockmates.
3. Flock Centering: attempt to stay close to nearby flockmates.

Steering behaviors for individual characters were also introduced by Reynolds [Reynolds 1999]. Steering is described as a change in the current velocity of an agent in order to achieve a specific behavior. Some of the steering behaviors described are the following: (1) **Seek** gradually steers the current velocity until it is aligned perfectly towards the goal. (2) **Flee** is similar to **seek** but the velocity is gradually changed to point away from the goal.(3) **Pursuit** also works similar to **seek**, except that the target is another moving character.

Recent efforts in crowd simulation have enabled dealing with improving computational performance.

For example, Pettré et al. presented an approach for handling scalable simulations and rendering [Pettré et al. 2006]; using different **levels of simulations**, they allow to distribute the available computation time spatially and temporally according to the spectator's point of view: its quality is enhanced where attention is focused and progressively decreases toward invisible zones. In their work, they also described a preprocessing technique used to generate navigation graphs.

Treuille et al. [Treuille et al. 2006] present a real-time crowd model based on continuum dynamics, where a dynamic potential field simultaneously integrates global navigation with moving obstacles such as other people, without the need for explicit collision avoidance.

Other efforts in crowd simulations are spent dealing with high densities. Narain et al. [Narain et al. 2009] augment the standard representation of a crowd as a set of agents with a continuous representation, which characterizes the crowd as a continuum fluid described by density and flow velocity.

Guy et al. [Guy et al. 2009] address the problem of real-time collision avoidance in multi-agent systems that use distributed behavior models. They use a discrete optimization method to efficiently compute the motion of each agent, and the resulting algorithm can be parallelized, increasing the performance.

There has also been a lot of research to develop velocity-based approaches. Paris et al. [Paris et al. 2007] describe a method for solving interactions between pedestrians and avoiding collisions, where each agent perceives surrounding agents and extrapolates their trajectory in order to react to potential collisions.

The **Reciprocal Velocity Obstacles** technique, introduced by van den Berg et al. [van den Berg et al. 2007], takes into account the reactive behavior of other agents by implicitly assuming that the other agents make a similar collision-avoidance reasoning. This method displays much smoother and realistic locomotion trajectories, especially thanks to the anticipatory adaptation to avoid collisions between characters.

Simulation-based techniques seem ideal for producing an ambient crowd for large environments but several problems are recurrent with such approaches:

1. Crowd simulation is computationally demanding and crowd size is severely limited for interactive applications on light computers.
2. Simulation is based on simplistic behaviors (e. g., walking, avoiding collisions, etc.) and therefore it is difficult to generate diverse and rich crowds based on classical approaches.
3. Crowd simulation is prone to animation artifacts or deadlock situations and it is thus impossible to guarantee animation quality.

Example-based approaches attempt to solve the limitations on animation quality. The main idea of these approaches is to indirectly define the crowd behavior rules from existing crowd data (such as trajectories from real people):

- Lerner et al. [Lerner et al. 2007] , describe a method to create realistic crowds based on real data. First, they construct a database, which takes place during pre-processing: the input video is manually tracked generating a set of trajectories. Then, at run-time, the virtual trajectories of the agents are synthesized individually by encoding their surroundings and searching the database for a similar example.

- In the work done by Ju et al. [Ju et al. 2010], new crowd animation is generated from blending existing crowd data. Thus, this allows the creation of a more diverse crowd behavior.
- Charalambous and Chrisanthou [Charalambous and Chrysanthou 2014] propose a data-driven method to generate believable steering behaviors; for this, a perception-action graph is constructed from the input data. This graph identifies situations or interactions between characters and exploits possible transitions between them. At run time, agents can efficiently simulate similar behavior to the source data.

Data-driven trajectories are typically of good quality, because they reproduce real recorded ones. However, such approaches raise other difficulties: Their variety depends directly on the variety of the example database content; it is difficult to guarantee that this database can cover all the required content, and it is also hard to control behaviors and interactions displayed by characters if the database content is not carefully selected. Data-driven approaches are also amongst the most time computationally demanding; even more so than traditional simulation based techniques. Some researches, such as Boatright et al. [Boatright et al. 2014] seek to find a middle ground between example and simulation-based methods aiming for both the better quality provided by data-driven methods and the fast speed associated with simulation methods.

One alternative to solve both performance and quality issues, are methods that interconnect pre-computed patches of animation to create larger ambient animation.

Crowd Patches [Yersin et al. 2009] are a type of 3D animated texture elements, which record the trajectories of several moving characters. Trajectories are periodic in time so that the crowd motion can be played endlessly. Trajectories' boundary conditions at the geometrical limits of patches are spatio-temporally controlled to allow connecting together two different patches with characters moving from one patch to another. Therefore, a crowd animated from a set of patches have seamless motion and patches' limits can't easily be detected. The boundary conditions are all registered into patterns, which function similar to gates for patches, with a set of spacetime input/output points.

Some techniques expanding and improving the Crowd Patches approach already exist. Kim et al. [Kim et al. 2012] propose a method to generalize the motion of crowd patches into motion patches, where individuals can perform any kind of action inside the patch, and not only walking. To stitch two patches together, not only the time and space position of the entry and exit points must match, but also the character's position, direction and fullbody pose must be within user-specified thresholds. Jordao et al. [Jordao et al. 2014] introduce a method to interactively design populated environments by using intuitive deformation gestures to drive both the spatial coverage and the temporal sequencing of a crowd motion.

Nevertheless, when using the Crowd Patches approach, a set of patterns should be used to be able to connect various patches together. As a result, it is important to construct a patch by starting from a set of patterns, and then deducing internal trajectories for that patch from the set of boundary conditions defined by the patterns. For this reason, we need to compute trajectories for characters that pass through a given set of spatio-temporal control points; i.e., characters should reach specific points in space at specific moments in time. This problem is difficult since generally speaking, steering techniques for characters consider 2D spatial goals, but do not consider the exact time a character must take to reach its waipoint. Therefore, dedicated techniques are required.

Yersin et al. suggest using an adapted Social Forces [Helbing et al. 2005] technique to compute internal trajectories. The key idea is to construct the trajectories by modeling the movement of particles (the characters) walking from one end of the trajectory to other; these particles are repulsed by other particles (in order to avoid collisions) and attracted towards the goal; the level of attraction towards the goal increases as the particle is closer to the time it should reach it. One problem with these approach is the limited density level, as well as the level of quality of trajectories that suffer from the usual drawbacks of the Social Forces approach such as lack of anticipation, which results into natural looking local avoidance maneuvers.

Compared to previous techniques we suggest formulating the problem of computing internal trajectories as an optimization problem. First, we suggest optimizing the way spatio-temporal input and output points are connected. Especially , since boundary control points are defined in space and time, we connect them aiming for *comfortable* walking speeds (i.e. close to the average human walking speed). Indeed, characters moving too slow or too fast are visually evident artifacts. Second, after having connected the initial boundary points with linear trajectories, we deform them to remove any collisions by employing an iterative approach. This approach aims at minimizing the changes to the initial trajectories. We demonstrate improvements in the quality of results as compared to the original work by Yersin et al. (Chapter 5).

3 Definitions

3.1 Patch

Understanding patches is one of the most fundamental aspects of this thesis, and for that reason, the focus of this chapter is the notion of a Patch and all of its components.

Simply put, patches are animation-building blocks. Each patch contains by itself a certain number of moving characters that repeat their movement periodically. Each patch by itself is already a small crowd simulation, but when several patches are put together, it's when their full potential is unlocked, creating a bigger and richer animation.

More formally, based on the description given in Yersin et al. [Yersin et al. 2009], a **patch** is a set $\{\mathbf{A}, \pi, \mathbf{D}, \mathbf{S}\}$ where:

- $\mathbf{A} \subset \mathbb{R}^2$ is a geometrical convex area. All the examples detailed in this work will use rectangles or squares, but all the techniques used can be generalized for any convex polygon.
- π is the duration (or period) of time of the animation. In the coordinates of the patch, the end and beginning of the time period are considered the same, so 0 is equivalent to π.
- $\mathbf{D}$ and $\mathbf{S}$ are sets indicating the dynamic and static objects, respectively. Static objects are the ones that remain completely inside the patch for the entire duration of the animation; these objects represent standing people or unmovable obstacles, like statues or fountains. Dynamic objects represent the agents traversing the patch in some direction, entering at some point in time and exiting at another. These agents follow paths that are connected between patches, and these paths must respect certain continuity conditions which will be fully described in the next section. These sets can be empty and in that case the patch is

called an empty patch.

Patches can be visualized as three dimensional objects (Figure 3.1), but at the moment of the animation, only a time slice will be shown each instant.

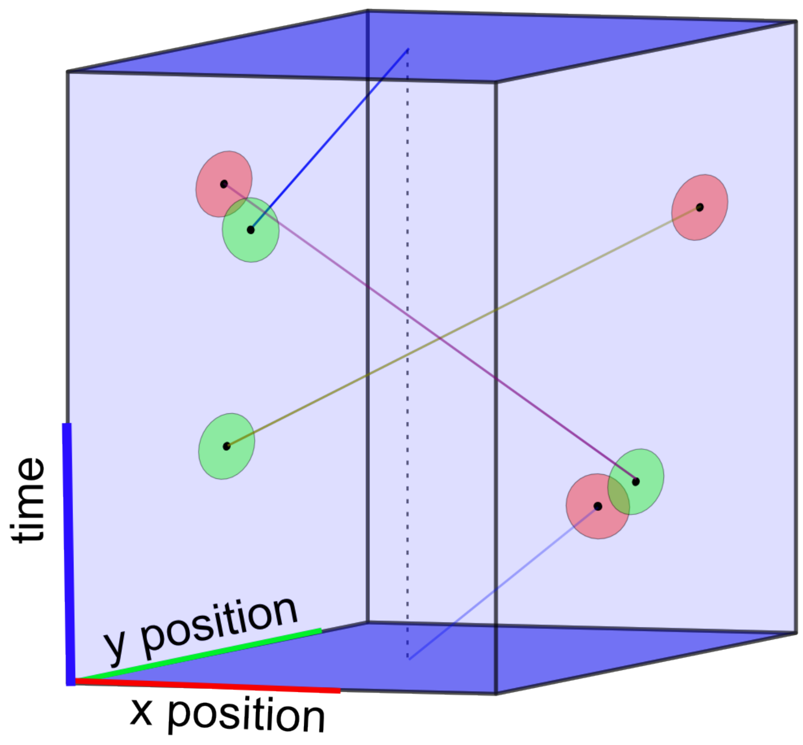


Figure 3.1: Simple Patch. A simple visualization of a patch with 3 dynamic objects, connected by simple linear paths. Notice how one of the paths ends in the top of the patch and enters again in the same spatial position at the bottom. Green circles represent entry points and red points represent exit points.

3.2 Paths and Trajectories

Dynamic objects are divided further into 2 categories: Endogenous and exogenous agents. Endogenous agents remain inside the patch at all times. Exogenous agents can travel between patches and because of this, it is possible that they only appear a fraction of the period inside the patch. They enter the patch at position $\mathbf{p_{initial}}$ and time $t_{initial}$ and they exit at $\mathbf{p_{final}}$ and t_{final}. To describe the movement of the agents between those extremes, paths will be used. In the remainder of this work, we will call a **Path** a sequence of trajectories.

A **trajectory** is a continuous function $\tau(t)$ that goes from $[t_1, t_2] \in \mathbb{R}$ (time) into $\mathbf{A} \in \mathbb{R}^2$ (2d space). This function represents the position of an agent at each given time t.

$$\tau : [t_1, t_2] \rightarrow \mathbf{A}, \quad 0 \leq t_1 < t_2 \leq \pi \tag{3.1}$$

While any continuous function can work effectively as a trajectory, in this work, piecewise linear functions will be used for their simplicity when calculating intersections and minimal distances.

Each trajectory will be then represented as a list of control points connected by segments.

A **control point** will be defined as a point in space and time $\mathbf{cp} = \{\mathbf{p}_{cp}, t_{cp}\}$. There will be two types of control points, movable and fixed. Fixed control points are defined by $\mathbf{p_{initial}}, t_{initial}$ and $\mathbf{p_{final}}, t_{final}$ stated above. These points are called also **boundary control points** because they mark the two extremes of a path. Boundary control points can not be moved, added or deleted since that would alter the continuity between patches. All the other points will be movable controls points or **middle control points**. Middle control points can be moved, added or deleted, as long as their coordinates remain inside the boundaries of the patch.

A **segment** is the straight line connecting two control points. It is assumed that a segment always goes from past to future, since traveling backwards in time is forbidden.

The hierarchy to remember is:

$$\textbf{control points} \subset \textbf{trajectories} \subset \textbf{paths} \tag{3.2}$$

The sequence of trajectories in a path have to follow a continuity rule, which is, if τ_i and τ_{i+1} are trajectories next to each other in a path, then:

$$\tau_i(t_\pi) = \tau_{i+1}(t_0), \ \ \forall i < n. \tag{3.3}$$

where t_π represents the time of the last control point at trajectory τ_i and t_0 is the time at the first control point of trajectory τ_{i+1}. These times have to be a match, that means, either they are exactly equal or, π and 0. Constructing path with sequences of more than trajectory usually leads to the second case, as will be seen in later chapters.

3.2.1 Examples

Let $\mathbf{cp_0} = \{(-8, 0), 2.5\}$ and $\mathbf{cp_1} = \{(8, 0), 7.5\}$ be two control points defining the extremes of a path. Then, several simple examples of different paths can be constructed (in a patch having an area of 16x16 and a period of 10).

1. A simple path going from $\mathbf{cp_0}$ to $\mathbf{cp_1}$ consisting of only 1 trajectory (Figure 3.2a):

   $Path = \{\tau_0\}$
   $\tau_0 = \{\{(-8, 0), 2.5\}, \{(8, 0), 7.5\}\}$

   In this case, both of the points are boundary control points and can't be moved.

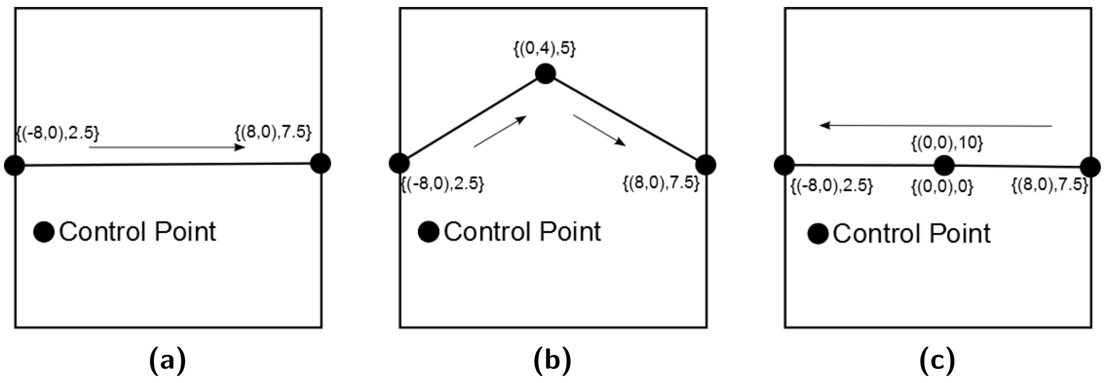


Figure 3.2: Simple Paths. (a) A simple path composed of one simple trajectory. (b) One path composed of one trajectory with 3 control points. (c) One path composed of two trajectories, each with two control points (the middle points are overlapping).

2. A slighter different path consisting of adding a new middle point to the trajectory in the previous example (Figure 3.2b):

   $Path = \{\tau_0\}$
   $\tau_0 = \{\{(-8,0),2.5\},\{(0,4),5\},\{(8,0),7.5\}\}$

3. A more interesting path is the one going from $\mathbf{cp_1}$ to $\mathbf{cp_0}$, with two trajectories (Figure 3.2c):

   $Path = \{\tau_0,\tau_1\}$
   $\tau_0 = \{\{(8,0),7.5\},\{(0,0),10.0\}\}$
   $\tau_1 = \{\{(0,0),0\},\{(-8,0),2.5\}\}.$

3.3 Patterns

As it was said before, patches need to have some sort of continuity between the trajectories traversing them. To enforce this continuity, *patterns* are used. Remember, a patch can be visualized as a spatio-temporal right prism (its shape will depend on the polygonal area of its base); then, each lateral side of that prism is a **pattern**.

More specifically, a **pattern** is a rectangle whose base is one of the edges of the polygonal area $\mathbf{A}$ (we call $\mathbf{l} \in \mathbb{R}^2$ this two-dimensional segment) and whose height is equal to the time of the period π (Figure 3.3).

Additionally, each pattern also include the sets $\mathbf{I}$ and $\mathbf{O}$ of Input and Output boundary control points defined by all exogenous agents entering and exiting through them. $\mathbf{I}$ contains the boundary control points where exogenous agents begin their trajectories, called *Entry Points*. Similarly, $\mathbf{O}$ contains the control points indicating the position at which the exogenous agents leave the patch, called *Exit Points*.

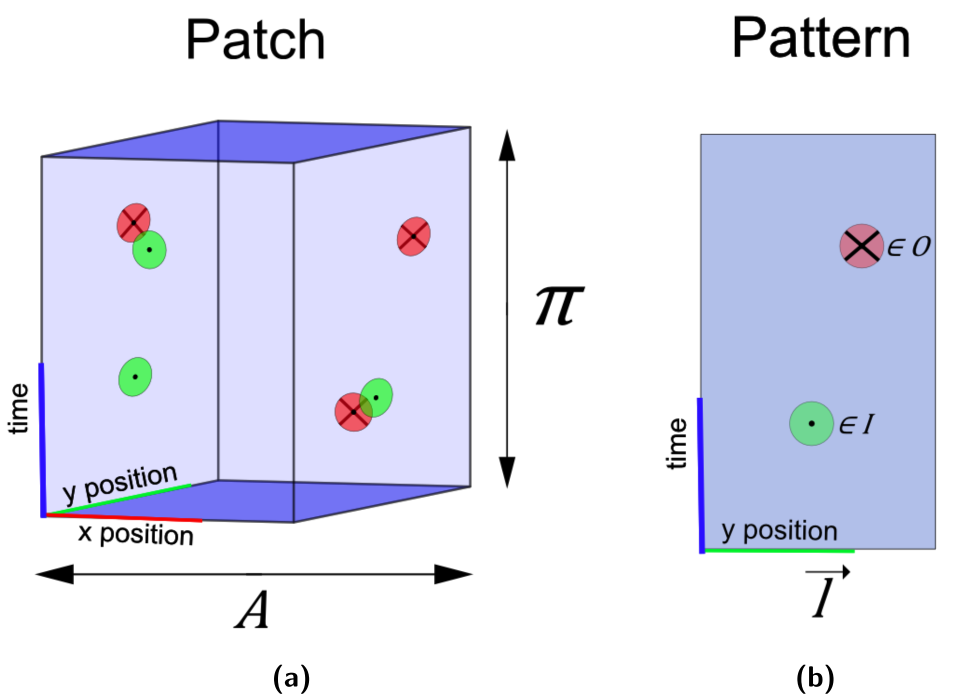


Figure 3.3: Patch and Pattern. a) A Patch is defined by the geometrical area A where the set of dynamic and static objects move over the time period π. b) In this case, one of the four patterns is represented as a rectangle, with a view from outside the patch (orientation of patterns will become important later on). Patterns define boundary conditions that assure continuity between patches.

Formally, a pattern $\mathbf{P}$ is:

$$\mathbf{P} = \{\mathbf{l}, \pi, \mathbf{I}, \mathbf{O}\} \tag{3.4}$$

Using this definition, we can now establish the necessary and sufficient conditions to say that two patterns $\mathbf{P_1}$ and $\mathbf{P_2}$ are well connected:

Let be

$$\mathbf{P_1} = \{\mathbf{l_1}, \pi_1, \mathbf{I_1}, \mathbf{O_1}\} \tag{3.5}$$

$$\mathbf{P_2} = \{\mathbf{l_2}, \pi_2, \mathbf{I_2}, \mathbf{O_2}\} \tag{3.6}$$

then, $\mathbf{P_1}$ and $\mathbf{P_2}$ can be connected if and only if:

- $\mathbf{l_1} = \mathbf{l_2}$
- $\pi_1 = \pi_2$
- $\mathbf{I_1} = \mathbf{O_2}$
- $\mathbf{O_1} = \mathbf{I_2}$

If any two patterns meet these conditions, they are **mirror patterns** of each other. Agents traversing these patterns will do it with continuity C^0 thanks to these conditions.

The number of patterns in a patch is determined directly by the number of the edges of **A**. Let's assume **A** has n sides, then n patterns are defined $\mathbf{P_1}, \mathbf{P_2}, ..., \mathbf{P_n}$. Since the number of entry points are the same as the number of exit points (each exogenous agent determines an entry and an exit point) then we can rewrite that property in terms of the patterns:

$$\sum_{i \in [1:\mathrm{n}]} |I_i| = \sum_{i \in [1:\mathrm{n}]} |O_i| \tag{3.7}$$

The equation 3.7 is called the *parity condition* of a patch.

Until now, we have seen how from all the components of a patch, the patterns are fully defined. For the purpose of this work, patches will be created from the information of the patterns. Specifically, the main interest of this work is how to construct free collision paths for the exogenous agents.

4
Methodology

This chapter will focus on the methodology concerning the construction of patches and more specifically, the paths of the dynamic agents that populate them. The area A, the period π and the desired number of static and dynamic objects will be given as input to the function that creates the patches. In some cases, the entry and exit points of the dynamic agents will be also given (when they come from an adjacent patch); when not given, the necessary entry and exit points will be created. Once the entry and exit points are defined, the trajectories between them will also be constructed. In the next section, an overview of the proposed methodology is presented. The rest of the chapter details each stage of our algorithm.

4.1 Strategy Overview

Figure 4.1 depicts our overall strategy for constructing a patch given the area A, the period π, the desired number of static and dynamic agents and the entry and exit points. Our algorithm identifies three main stages:

1. **Matching**: Selecting which exit control points correspond to each entry point is the first step of this technique. Certain pairings of points are better than others, depending on their associated speed and form. To solve this problem, the Stable Marriage Algorithm [Gale and Shapley 1962] is used to assure a good quality for the pairings. Because of the importance of creating well spaced entry and exit points when they are not determined by neighboring patterns, the Poisson Sampling Technique is described. Finally, an explanation of how the initial straight line paths are created, including the modifications done to solve initial violations to the trajectories constraints in space-time is presented. The output of

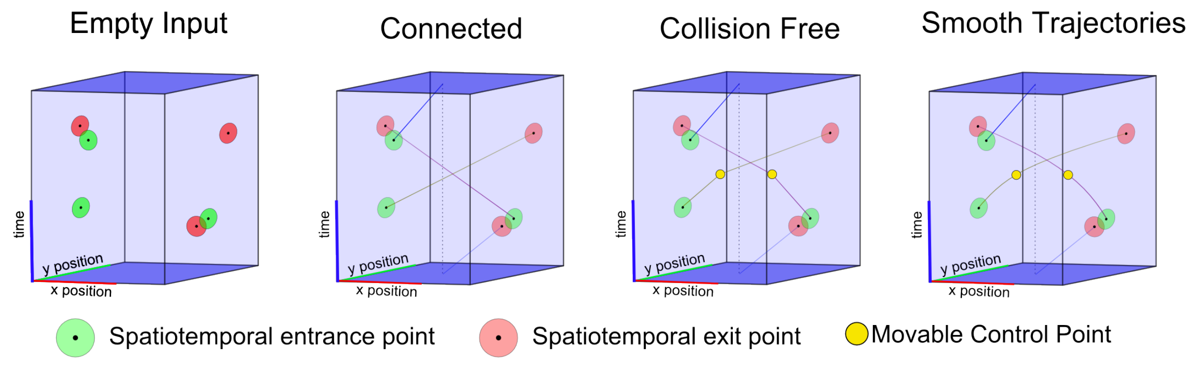


Figure 4.1: Overview. Initial boundary control points are connected, then their trajectories are modified twice, first by the collision avoiding step and finally by the smoothing step.

this function will be the first batch of linear paths.

2. **Creation of Collision Free Trajectories**: This step will be explained in three sections, given its complexity.

   (a) **Collision avoidance:** Starting from simple trajectories, new control points will be added and modified until they don't produce any more collisions. This will be done in an iterative process. Boundary control points of the patch are hard constraints since they must not be moved.

   (b) **Obstacle handling**: Endogenous agents, or obstacles, are harder to avoid, since their position is usually a constraint that can't be changed. In this section, an approach for handling obstacles will be explained using obstacle grouping.

   (c) **Group Handling**: An extension of the algorithm will be explained in this section, so more than one people can be seen walking together.

3. **Smoothing Process**: An initial approach for smoothing the paths found in previous steps is explored in this section. Cubic splines will be used with the extra precaution that original paths can't be modified much or they may start colliding again.

4.2 Matching

The initial matching of the boundary control points is one of the contributions of this thesis. In previous works, points were randomly matched, which in some cases produced undesirable results, such as trajectories created between them staying on the sides of the patch. As a quick introduction, first let's establish what is exactly the matching problem that we want to solve.

Given two sets, one of entry **I** and one of exit points **O** (both with the same cardinality) we want to match their elements in pairs optimizing certain aspects (Figure 4.2):

1. The path between the pair has its endpoints lying in different patterns. Otherwise, the path stays on one side of the pattern and that is an undesirable behavior, since, judging

by observation, trajectories that pass through or near the center of a patch look better and are more interesting.

2. The speed at which the agent walks over the path stays close to some believable speed. Usually, walking human speed is close to 1.33 m/s [Whittle 2003].

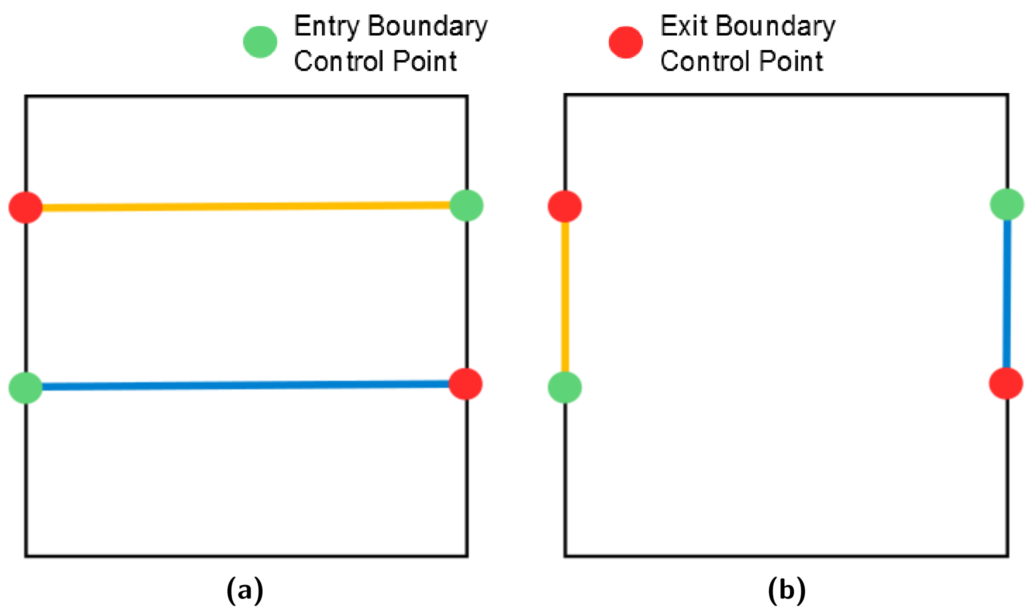


Figure 4.2: Matching Examples Two examples of how matching can be done with two pairs of entry and entry points. Entry points are in green and exit points are in red. (a) Desired matching. The paths cross the patch. (b) Non-desired matching. The paths stay to the side of the patches, and leave empty the area in the middle. Notice these are the only two possibilities for a matching. The matching that makes an x is forbidden since it would match points of the same color.

4.2.1 Creation of Boundary Control Points

Before diving straight into the matching algorithm, let's make an initial remark on how the initial group of entry and exit points is obtained. They may have been determined by neighboring patches, but when they are not given they have to be created automatically.

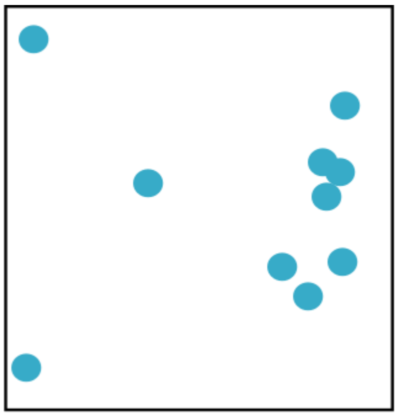
The points can be created at random, but that leads sometimes to bad initial conditions. Some entry and exit points can end up very close to each other. As it will be seen later, collisions happening at the border of the patch are the most difficult to avoid, since boundary control points can't be modified.

Therefore, one better way to create them is by using the Poisson Disk Sampling. Using the **Poisson Disk Sampling**, the samples are randomly placed with the restriction that no two samples are closer than a certain distance [Cook 1986].

One simple way to implement this is the **Dart Throwing Algorithm** described in the same work: A look-up table is created by generating random sample locations and discarding any location that is closer than a certain distance to any of the locations already chosen. Locations are generated until the sampling region is full, and a region is considered full if, after a certain number of trials, it can't find a new suitable location apart from all the other existing ones.

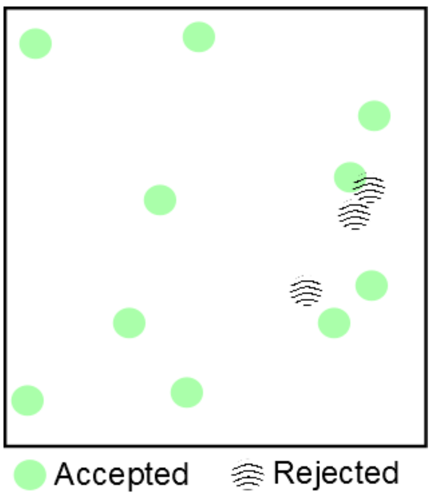
For the purpose of this work, we don't need to sample until the region is full, only until we have encounter the required number of entry and exit points. This is a very straightforward technique, though computationally expensive, but since the range of the number of control points needed is short (usually no more than 40 pairs of entry and exit points are needed), it can be implemented without further optimizations. Now, there is still a problem, and that is when the algorithm halts before finding the required number of points. That can happen if the area of the patterns is too small, or when the desired distance between two points is too big, for example. In those cases, we have no other choice than to select the remaining points at random.

Next, some examples of sampled points created in a square using both methods. Random Method (Figure 4.3) and Poisson Sampling (Figure 4.4) are shown.



X	Y
250	83
280	238
229	107
27	292
262	192
276	184
120	175
21	25
265	164
278	111

Figure 4.3: Random Sampling. Ten points are selected at random. Notice how some of them overlap. Those points create very difficult or even impossible collisions to avoid.



X	Y	Accepted
250	83	YES
280	238	YES
229	107	NO
27	292	YES
262	192	YES
276	184	NO
120	175	YES
21	25	YES
265	164	NO
278	111	YES
149	297	YES
140	31	YES
97	83	YES

Figure 4.4: Poisson Sampling. Some of the points are now rejected for being too close to previously existing ones. A better set of spaced points is obtained.

A last note for this section is on how to measure the distance between two boundary points and how to calculate a good threshold for the distance they need to be apart.

A slight problem arises when we consider that the points we want to obtain from the sample have two space coordinates and one time coordinate. Usual euclidean distance may not work since the scales can vary a lot depending on the units used. For that, a rescaling may be needed. Also, because time is periodic, when calculating distance between two points $\{\mathbf{p_1}, t_1\}$ and $\{\mathbf{p_2}, t_2\}$ with $t_1 < t_2$, it has to be done in two ways:

1. $d(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\})$
2. $d(\{\mathbf{p_1}, t_1 + \pi\}, \{\mathbf{p_2}, t_2\})$

The minimum between those two is taken as the closest distance (Figure 4.5).

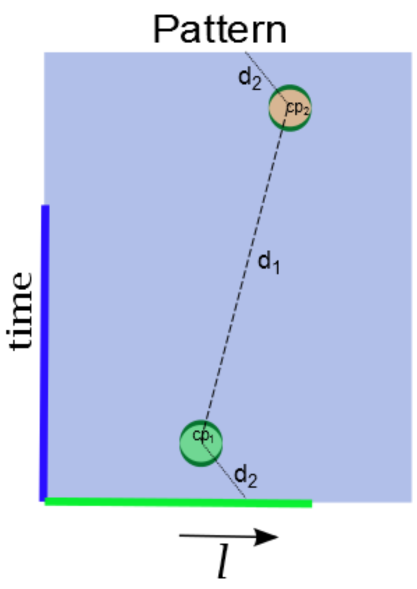


Figure 4.5: Distance between boundary points. When distance is measured between boundary points, the minimum between d_1 (measuring directly) and d_2 (measuring using periodicity) is considered.

The next problem is to define the distance function. We want the points to be a certain time and a certain distance apart. Different thresholds of minimum distance were used in the dart throwing algorithm depending on the number of boundary points required and the area of the patterns, but usually a minimal separation of 1.5 meters or seconds is required. One way to define the distance can be (when both units are meters and seconds):

$$d(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) = max(\|\mathbf{p_1} - \mathbf{p_2}\|, |t_1 - t_2|) \tag{4.1}$$

Another way to define the distance can be the euclidean definition:

$$d(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) = \sqrt{(\|\mathbf{p_1} - \mathbf{p_2}\|^2 + |t_1 - t_2|^2)} \tag{4.2}$$

Equation 4.1 defines rectangular neighborhoods, and Equation 4.2 defines circular neighborhoods, which are slightly smaller and therefore less restrictive, giving more space to the Poisson Sampling Method to successfully find all the points. For this reason, in this work Euclidean distance was preferred. One thing to have in mind when using any of the distances is that if a rescale is made in the units, an inverse rescale will be needed before calculating the distance.

4.2.2 Stable Matching Algorithm

The matching problem we want to solve is the next one:

Given two sets I and O of entry and exit points, we want to find a bijective function between them, but not any bijective function; we want to pair them in an optimal way, trying to maximize the quality of each pair.

First, a way to measure the quality between pairings is going to be defined. A pair of boundary control points will define paths and, as previously said, we want those paths to cross the patch and not only stay on the sides. Then, among all candidate matchings belonging to different patterns, we prefer the ones that allow a walking speed close to $1.33m/s$ (referred to in this work as u_{cft}), the mean of walking speed of humans in unconstrained environments. Assuming that $(\mathbf{p}_1, t_1)$ and $(\mathbf{p}_2, t_2)$ are the position and time of the entry and exit points respectively, speed is defined as $u = s/\Delta t$ where $s = |\mathbf{p}_2 - \mathbf{p}_1|$ and $\Delta t = t_2 - t_1$ when $t_2 > t_1$, otherwise $\Delta t = \pi + t_2 - t_1$. [1]

The quality of the path between two points will be defined as:

$$qu(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) = u_{match}(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) + p(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) \tag{4.3}$$

where

$$u_{match}(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\}) = arctan(|u_{cft} - u(\{\mathbf{p_1}, t_1\}, \{\mathbf{p_2}, t_2\})|) \in [0, \pi/2) \tag{4.4}$$

and p is a penalization factor. Its value depends on where the two points lie relative to each other; for points on opposing patterns there is no penalty, for neighboring patterns it is 2 and for points on the same pattern it is 4 (Figure 4.6).

Using this function, we can now rank every possible candidate for a point's match in order of quality (Table 4.1), where low numbers indicate a bigger quality. Remember, there are two sets, I and O, each with a quantity of n points. Therefore, in total, $2n$ rankings are created. A stable match between those sets is such that these two conditions don't happen at the same time:

1. An element $i \in I$ prefers some element $o \in O$ over its current partner, and
2. o also prefers i over its own partner.

[1]More details on why this this last assumption is made will be presented later during the creation of the initial set of trajectories.

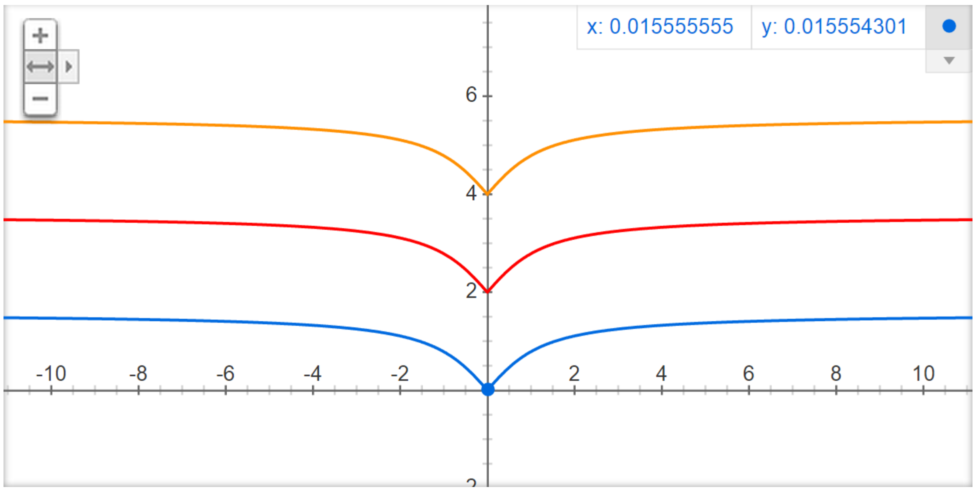


Figure 4.6: Weights Graph Three main levels of weights are distinguished. The top graph represents the possible weights between points in the same pattern. The middle graph represents the possible weights between points in neighboring patches. The bottom graph represents the possible weights between points in opposite patterns. The x-axis represents the difference between the comfort speed and the current one used to travel in a straight line between the control points.

Entry Point: 1	
Exit Point	Preference
b	**0.34**
c	1.3
a	2.3
e	2.4
d	4.5
f	4.6

Table 4.1: Proposal List. Each entry point keeps a list of preference scores for all possible exit points. Smaller values indicate higher preference, with exit point b in this case being the most preferred one. Exit Points f and d lie in the same pattern as Entry Point 1, so they receive a larger score.

The problem of finding a stable matching between two sets is frequently stated as the Stable Marriage Problem:

In Bachelor City, there live n men and n women. All of them have made a list ranking the members of the opposite sex with a unique number between 1 and n in order of preference. Your task is to marry them such that there are no two people of opposite sex who would rather prefer being married to each other than the husband/wife they currently have. If you can succeed with this task, all the marriages are "stable".

To solve this matching problem, the *Gale-Shapley Algorithm* [Gale and Shapley 1962] is employed. Using our notation we can state the *Gale-Shapley Algorithm* as in Algorithm 4.1.

A point to make here is, even though the solution that can be found by the Stable Matching Algorithm is a stable one, it usually happens that the last pairings are from individuals who are close to the bottom of each others' ranking list. This will produce that sometimes points from the same pattern will still be matched. This is not a problem that can be solved, since

Algorithm 4.1: Gale-Shapley Stable Marriage Algorithm from [Gusfield and Irving 1989].

```
Initialize all $i \in \mathbf{I}$ and $o \in \mathbf{O}$ to free ;
while $\exists$ free entry point $i$ who still has an exit point $o$ to propose to do
    $o \leftarrow i's$ highest ranked exit point to whom it has not yet proposed ;
    if $o$ is free then
        $(i, o)$ become paired ;
    else
        some pair $(i^*, o)$ already exists ;
        if $o$ prefers $i$ to $i^*$ then
            $(i, o)$ become paired ;
            $i^*$ becomes free ;
        else
            $(i^*, o)$ remain paired;
        end
    end
end
```

sometimes it is impossible to please everyone. Imagine a situation of a rectangular point with 4 entry points in the left pattern and 4 exit points, each of them in a different pattern. One of the entry points will have to be matched with the exit point in the same pattern (Figure 4.7).

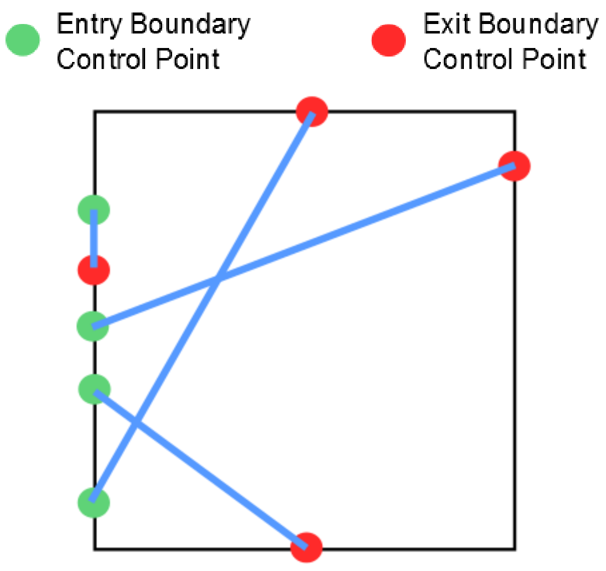


Figure 4.7: An example of an unavoidable situation when one of the matches has to be done between points belonging to the same pattern.

After the pairing, initial paths are created between the points, each of them consisting of only of one trajectory with those two points as its only control points. Before going to the next step for collision removal, some adjustments have to be done to those initial trajectories to alleviate some of the problems introduced by the matching algorithm (as not credible speed or paths not traversing the patch). Also, some end points may not meet the condition of $t_1 < t_2$

and may effectively try to go backwards in time. Another correction is introduced for those paths.

4.2.3 Initial Straight Line Trajectories

This section will detail three important modifications made to the initial trajectories.

1. Going-backwards-in-time correction:
   When the endpoints of a path don't meet the condition $t_1 < t_2$ one single trajectory is not enough to represent the path. The path is broken in two trajectories, one going from t_1 to π and the other one going from 0 to t_2.
2. Same Pattern Correction:
   Sometimes the path connects two points in the same pattern, which, asides from not looking very good, augments the possibilities for collisions near the border, which are very hard to avoid. For that reason, an additional control point in the middle of the trajectory is added, pulling the trajectory towards the center of the patch.
3. Speed Correction:
   Occasionally, the only problem is the speed at which an agent has to travel a path. If the time is not enough, the virtual characters animated in that patch will look unrealistically fast. For those cases, a trick similar to the one used for going backwards in time will be employed. There could also be the case of points traveling very slowly towards their goal. Those cases are not as critical, since it could be the path of an old person, or of someone carrying something heavy. Regardless, when those trajectories are not desired, stopping points can be introduced in the trajectories. People will walk at normal speed until a certain point, will stop, and then resume their walk.

One path can need one or more of these corrections and it is recommended to do them in the order suggested above. Speed can vary after corrections 1 and 2 and for that reason it is left until the end. Correction 1 is also easier if it is made before adding any extra points or creating extra trajectories.

4.2.3.1 Going-backwards-in-time Correction

Trajectories of the form $\{(\mathbf{p_1}, t_1), (\mathbf{p_2}, t_2)\}$ with $t_1 > t_2$ are ill-formed since they try to go backwards in time. These kind of trajectories will be split into two,

$$\tau_1 = \{(\mathbf{p_1}, t_1), (\mathbf{p_{aux}}, \pi)\}$$

and

$$\tau_2 = \{(\mathbf{p_{aux}}, 0), (\mathbf{p_2}, t_2)\}$$

where $\mathbf{p_{aux}}$ is the position at which the point would be at time π if the point travels from $\mathbf{p_1}$ to $\mathbf{p_2}$ at constant velocity, starting at time t_1 and ending at time $t_2 + \pi$ (Figure 4.8).

$$\mathbf{p_{aux}} = \mathbf{p_1} + \frac{\mathbf{p_2} - \mathbf{p_1}}{t_2 + \pi - t_1} * (\pi - t_1) \tag{4.5}$$

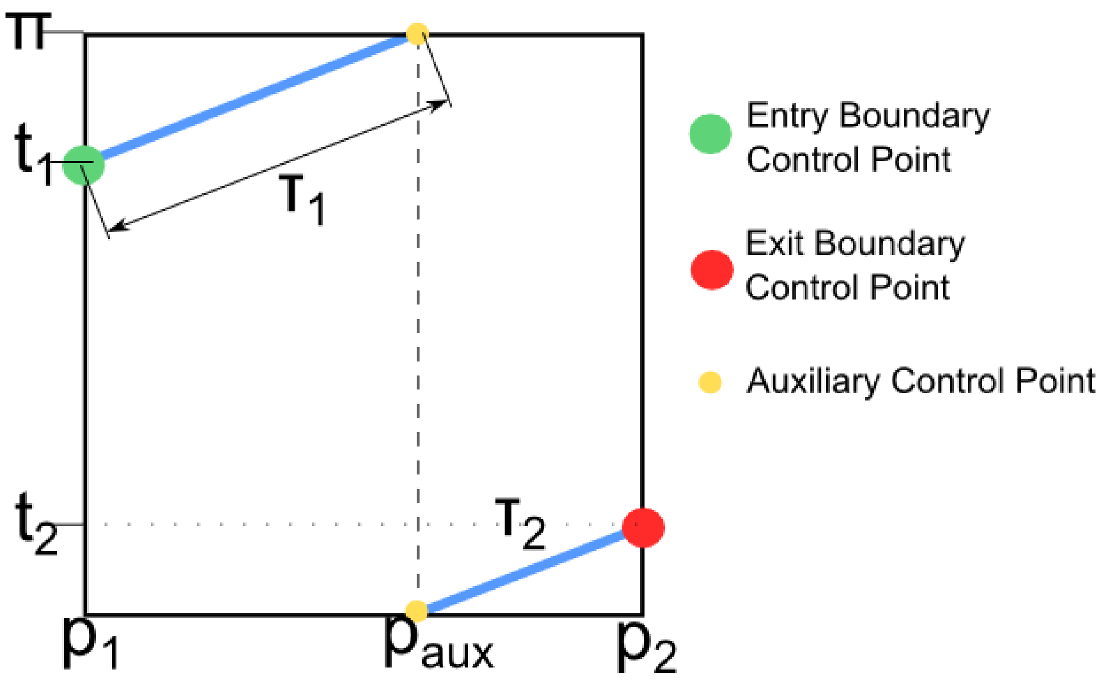


Figure 4.8: Trajectory Splitting Instead of connecting directly the boundary points, auxiliary control points are used and two trajectories are created in the path. Notice how this is NOT a top view of the patch nor one of its patterns.

4.2.3.2 Same Pattern Correction

Another auxiliary point will be used to modify trajectories whose endpoints are in the same pattern. Depending on the form of the patch this point can be created in several ways. For the purpose of this work, squares were used as the base of the patch. Auxiliary points were created by calculating the middle point in the trajectory and then by translating it by a vector perpendicular to the trajectory pointing towards the patch with a magnitude 1/4 of the size of the square.

$$\mathbf{P_{middle}} = \frac{\mathbf{P_1} + \mathbf{P_2}}{2} + v$$

where $\|v\| = \frac{l}{4}$ and is perpendicular to $\mathbf{P_1} - \mathbf{P_2}$ (Figure 4.9).

4.2.3.3 Speed Correction

When the speed between trajectories is unrealistic and above a certain threshold called maximal velocity ($1.8m/s$ in this work), a correction almost identical to the one made in subsection 4.2.3.1 is done. The difference is, the number of trajectories created can be more than two. This can happen due to the distance between the endpoints being too big or the period being too

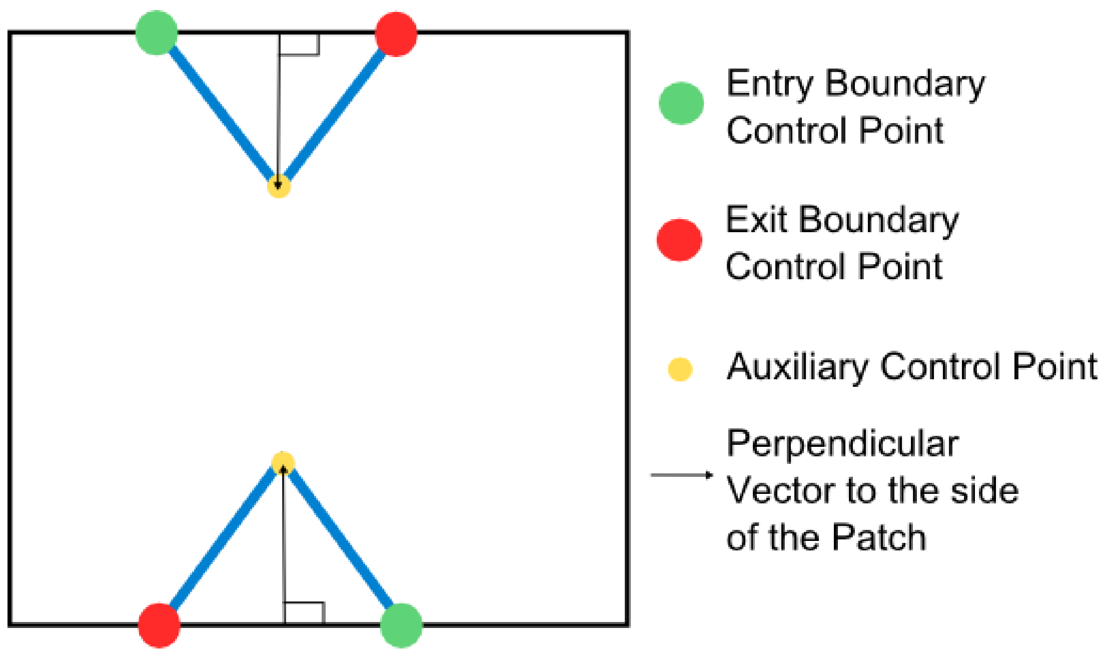

Entry Boundary Control Point
Exit Boundary Control Point
Auxiliary Control Point
Perpendicular Vector to the side of the Patch

Figure 4.9: Top view of a patch, where two middle points are created for two trajectories. After adding the new control point, both of the trajectories separate from the border, evading possible collisions near the edges and taking advantage of the space in the middle.

short. The number of extra trajectories depends directly on the number of times k the period has to be added to inequality 4.6 to become true. Left side represents the speed at which the agents will be traveling in each of the trajectories created.

$$\frac{\mathbf{p_2} - \mathbf{p_1}}{t_2 + k\pi - t_1} \leq max\ speed \tag{4.6}$$

Each auxiliary position $\mathbf{q_i}$ for $i \in [1 : k]$ will be created using 4.7.

These auxiliary control points can be visualized as the intersecting points between the line connecting the entry point to some exit point in a future period and the lines marking the periods. Then, the intersected points are projected into the current period (Figure 4.10).

$$\mathbf{q_i} = \mathbf{p_1} + \frac{\mathbf{p_2} - \mathbf{p_1}}{t_2 + i\pi - t_1} * (i\pi - t_1) \tag{4.7}$$

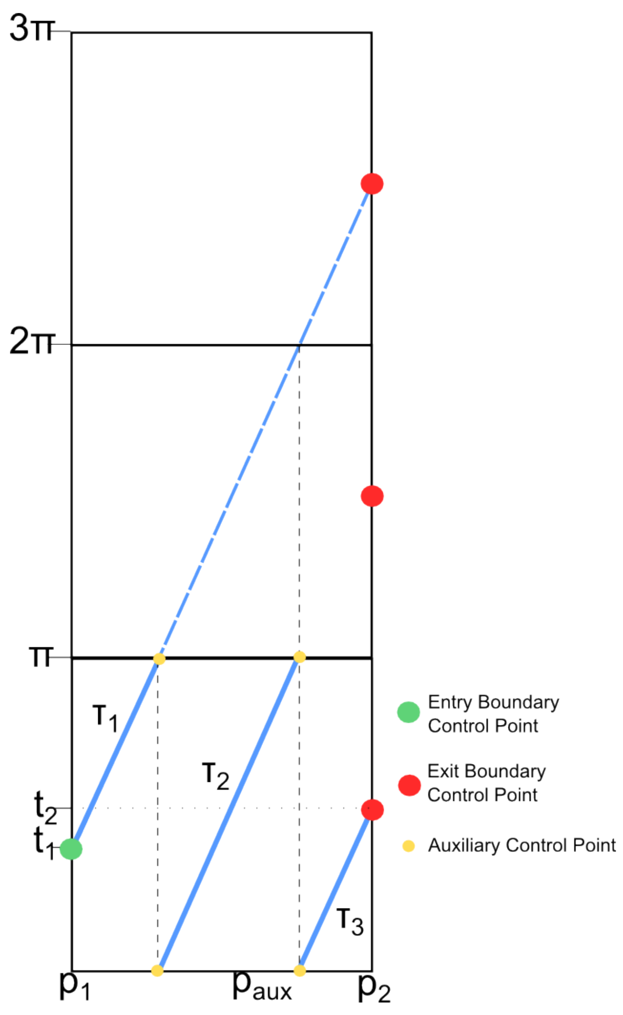


Figure 4.10: Speed Correction In this example, the period had to be added twice in 4.6 for the inequality to be true. That creates two additional trajectories, forming a grand total of 3 trajectories. In the animation, between times t_1 and t_2 three agents will be seeing at the same time in the path.

4.3 Collision Avoidance

After having created the first set of trajectories, the next step is to avoid collisions between them. A collision happens when two trajectories are closer than a certain threshold. This threshold α is determined by the radii r_1 and r_2 of the agents that will be passing through the trajectories $\alpha = r_1 + r_2$. When the minimum distance between all the pairs of trajectories is bigger than α the trajectories are collision-free. It is important to avoid collisions, otherwise the people will act as ghosts, passing through each other, which looks unrealistic.

To solve this problem, an algorithm is proposed that edits the trajectories by adding, moving and deleting control points (but not the boundary control points, since they are important for continuity between patches). The main idea of this algorithm is described in the following steps:

1. Detecting the points where collisions are happening, storing the minimal distances between trajectories.

2. Creating new control points (or using already created ones that are near the collision).
3. Applying repulsion forces to those points.

These modifications may create new collisions, so the process is repeated until no more collisions are detected (or a maximum number of iterations is reached).

4.3.1 Distance Matrix Creation

The creation of a matrix storing the distance between all the pairs of trajectories (including the ones belonging to the same path) will be of vital importance for the iterative algorithm handling the collisions.

The **minimum distance** between two trajectories is defined as their minimum spatio-temporal distance; i.e. at the point in time where they are closest to each other. Recall that trajectories can consist of two or more control points defining linear segments. For this reason, the minimum has to be found in-between all the trajectory's segments.

When calculating the minimum distance between two segments $\mathbf{s_1} = \{(\mathbf{p_1}, t_1), (\mathbf{p_2}, t_2)\}$ and $\mathbf{s_2} = \{(\mathbf{p_3}, t_3), (\mathbf{p_4}, t_4)\}$, two cases can happen:

1. The segments don't intersect in time. This happens when $t_2 < t_3$ or $t_1 > t_4$, which means one of the segments starts after the other is already finished. For those cases, the distance is ∞.
2. The segments overlap in time. For those cases, we construct the time segment t_{start}, t_{end} with $t_{start} = max(t_1, t_3)$ and $t_{end} = min(t_2, t_4)$. Then, the minimum distance is searched in that period of time.

To know exactly which is the minimum distance, we have first to represent the position of two point traveling along those two segments as functions:

$$f_1(t) = \mathbf{q_1} + t * \mathbf{v_1} \tag{4.8}$$

$$f_2(t) = \mathbf{q_2} + t * \mathbf{v_2} \tag{4.9}$$

where $\mathbf{q_1}$ and $\mathbf{q_2}$ are the two segments' position at time t_{start}, $\mathbf{v_1}$ and $\mathbf{v_2}$ are the velocities associated with those segments, $\mathbf{v_1} = \frac{\mathbf{p_2}-\mathbf{p_1}}{t_2-t_1}$ and $\mathbf{v_2} = \frac{\mathbf{p_4}-\mathbf{p_3}}{t_4-t_3}$, and $t \in [0, t_{end} - t_{start}]$.

Then, the distance relative to the time will be:

$$d(t) = \|f_1(t) - f_2(t)\| = \|\mathbf{q_1} - \mathbf{q_2} + t(\mathbf{v_1} - \mathbf{v_2})\| \tag{4.10}$$

If we rename $w = \mathbf{q_1} - \mathbf{q_2}$ and $dv = \mathbf{v_1} - \mathbf{v_2}$ the Equation 4.10 becomes:

$$d(t) = \|w + t * dv\| \tag{4.11}$$

The problem now is to find the minimum of that function, which is equivalent to finding the minimum of

$$d^2(t) = \|w + t * dv\|^2 \tag{4.12}$$

After some calculations, the first and second derivatives are found:

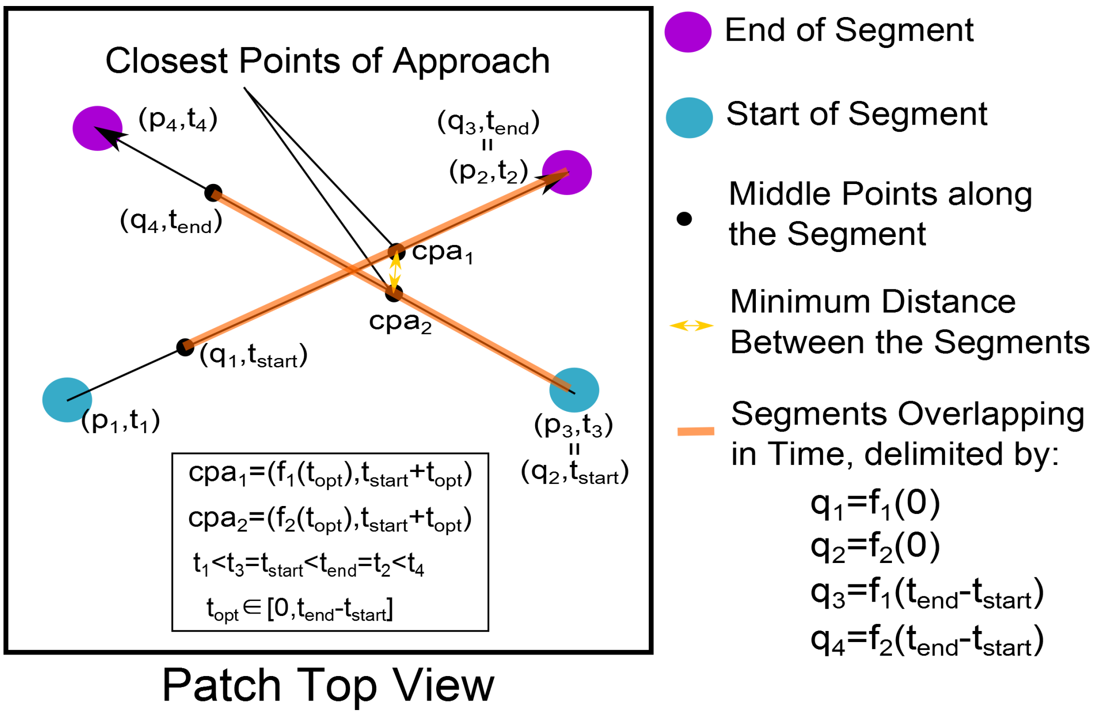
$$(d^2(t))' = 2(w \cdot dv + t\,\|dv\|^2) \tag{4.13}$$

$$(d^2(t))'' = 2\,\|dv\|^2 \tag{4.14}$$

Let's assume $(d^2(t))'' > 0$. If $(d^2(t))'' = 0$ then the segments are parallel and $d(t)$ is constant; for those cases the distance between segments is $\|w\|$ at all times. Then the minimum is found from finding t when Equation 4.13 is equal to 0.

$$2(w \cdot dv + t\,\|dv\|^2) = 0 \Longrightarrow t_{opt} = \frac{-w \cdot dv}{\|dv\|^2} \tag{4.15}$$

With t_{opt}, the closest distance is obtained from Equation 4.10 and the closest points of approach for both segments are obtained from Equations 4.8 and 4.9. Usually, the closest points of approach are not in the spatially projected intersection of the segments (Figure 4.11).



Closest Points of Approach
(p_4,t_4)
(q_3,t_{end})
=
(p_2,t_2)
(q_4,t_{end})
cpa_1
cpa_2
(q_1,t_{start})
(p_1,t_1)
(p_3,t_3)
=
(q_2,t_{start})
$cpa_1=(f_1(t_{opt}),t_{start}+t_{opt})$
$cpa_2=(f_2(t_{opt}),t_{start}+t_{opt})$
$t_1<t_3=t_{start}<t_{end}=t_2<t_4$
$t_{opt}\in[0,t_{end}-t_{start}]$
Patch Top View
End of Segment
Start of Segment
Middle Points along the Segment
Minimum Distance Between the Segments
Segments Overlapping in Time, delimited by:
$q_1=f_1(0)$
$q_2=f_2(0)$
$q_3=f_1(t_{end}-t_{start})$
$q_4=f_2(t_{end}-t_{start})$

Figure 4.11: Closest Points of Approach. The two closest points of approach between two segments are found, as well as the minimum distance between them.

It can be the case that the time t_{opt} found is not between $[0, t_{end} - t_{start}]$. Those are the cases when the segments end before the points can get to the closest distance. When that happens, a comparison between the distances at the endpoints ($d(t_{start})$ and $d(t_{end})$) is made. Whichever is the minimum will be the minimum distance (Figure 4.12).

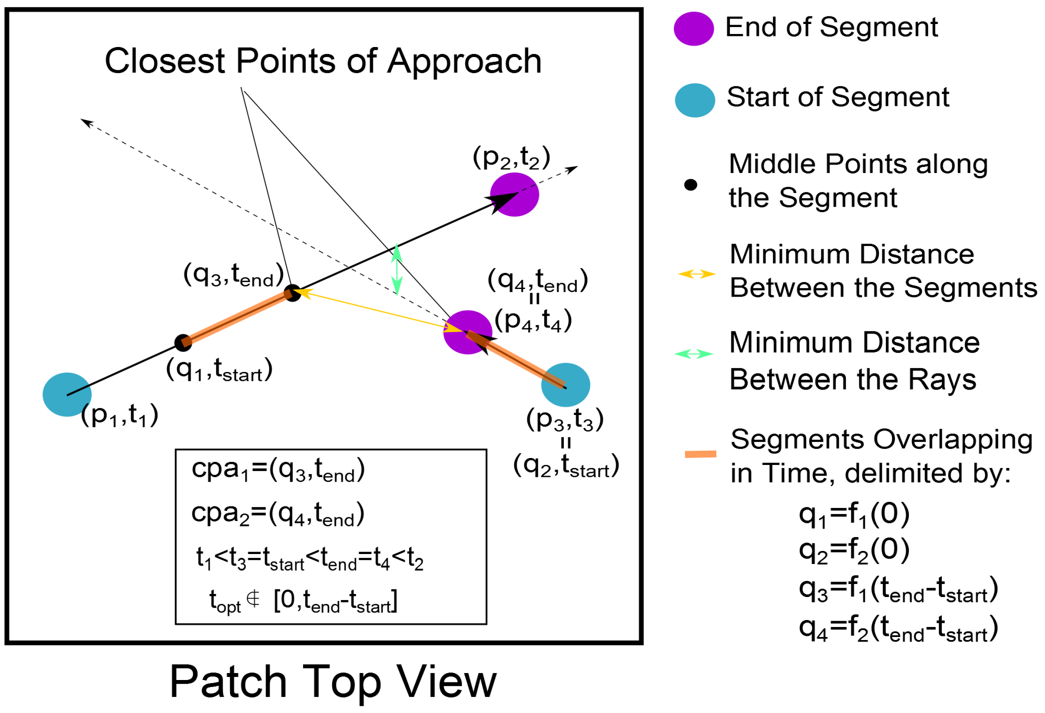


Figure 4.12: In this example, the time at which the rays defined by the segment are closest is not part of both segments. Instead, the distance between q_1 and q_2 (the points where segments start overlapping in time) is measured. We also measure the distance at the moment the trajectories end overlapping p_4 and $f_1(t_{end} - t_{start})$. The minimum between those two is stored as the minimum distance for those segments.

For every two trajectories, all pairs of their segments are compared but only the minimal of the minimal distances is stored in the distance matrix $M \in \mathbb{R}^{n\times n}$. The reason behind this will become clearer in the next section, but the idea behind this is that the trajectory is only modified once per iteration when it is in collision with another trajectory. Instead of infinity for trajectories that don't overlap in time, a big number (at least bigger than the threshold of collision) is stored. This work uses 1000. The space-time positions at which these minimum distances happen (called closest points of approach) are also stored in a different matrix C as well, where each entry in C is storing two values, $C(i,j) = \{\mathbf{cpa_{ij}}, \mathbf{cpa_{ji}}\}$ where $\mathbf{cpa_{ij}}$ is the closest point in trajectory τ_i to τ_j.

Both matrices have the next properties:

1. $M(i,i) = 0$ and $C(i,i)$ is null.
2. $M(i,j) = M(j,i)$, i.e., the matrix is symmetric.
3. $C(i,j)$ and $C(j,i)$ are permutations of each other, i.e. if $C(i,j) = \{\mathbf{cpa}, \mathbf{cpa}'\}$ then $C(j,i) = \{\mathbf{cpa}', \mathbf{cpa}\}$
4. $M(i,j) = \infty$, $\forall(\tau_i, \tau_j)$ that are never present at the same time. For those cases, $C(i,j)$ is also null.

Algorithm 4.2: The control points generation algorithm

```
Compute minimum distance matrix $M$ and matrix of closest points of approach $C$ ;
Create a boolean matrix $ID$ of size $n \times 1$ and set it to 0 ;
while there exists at least one entry in $M$ below the threshold do
    for every pair of indices $i$ and $j$ for which $M(i, j)$ has a value $d$ smaller than the
    threshold do
        Find in matrix $C$ control points $\mathbf{cp}_i$ and $\mathbf{cp}_j$ that are at such distance $d$, given by
        $C(i, j)$ ;
        Apply Repulsion Forces to $\mathbf{cp}_i$ and $\mathbf{cp}_j$;
        Add or modify control points in trajectories $\tau_i$ and $\tau_j$ to match $\mathbf{cp}_i$ and $\mathbf{cp}_j$;
        Set $ID[i] = 1$ and $ID[j] = 1$;
    end
    for every $k$ for which $ID[k] = 1$ do
        Apply Tension Forces to $\tau_k$ and all trajectories belonging to the same path.
    end
    Update $M$ and $C$;
    Set $ID$ to matrix 0;
end
```

Using those properties, some calculations can be avoided to reduce computation time.

4.3.2 Iterative Algorithm

Now that the process of creating a matrix storing the minimum distances between trajectories has been described, the algorithm for avoiding collisions will be presented (Algorithm 4.2).

Basically, what this algorithm does is simple. First it finds all the positions that are in collision between the trajectories, apply repulsion forces to them, and create a new control point (or modify an existing one which is close) in the trajectory at that position, pulling the trajectory to a non-colliding configuration (Figure 4.13).

After all repulsion forces are applied, tension forces are applied along the trajectory. Tension forces , contrary to repulsion forces, try to move the point back to its original place, but they are applied all over the path, not just to a single point. Depending on the magnitude of these forces, a parameter of elasticity is associated.

Calculating both repulsion forces and tension forces needs more description, so the next sections will elaborate on that.

An extra parameter for the maximum number of iterations is encouraged to be set as one of the stopping conditions of the algorithm, since for dense cases scenarios it will be very hard to find a collision free solution.

4.3.2.1 Repulsion Forces

Repulsion forces are very straightforward. When we have two points that are closer than a certain threshold, the most natural idea is to pull them apart until their distance is bigger than the threshold. While there are many ways to do this, we will do it maintaining several hard

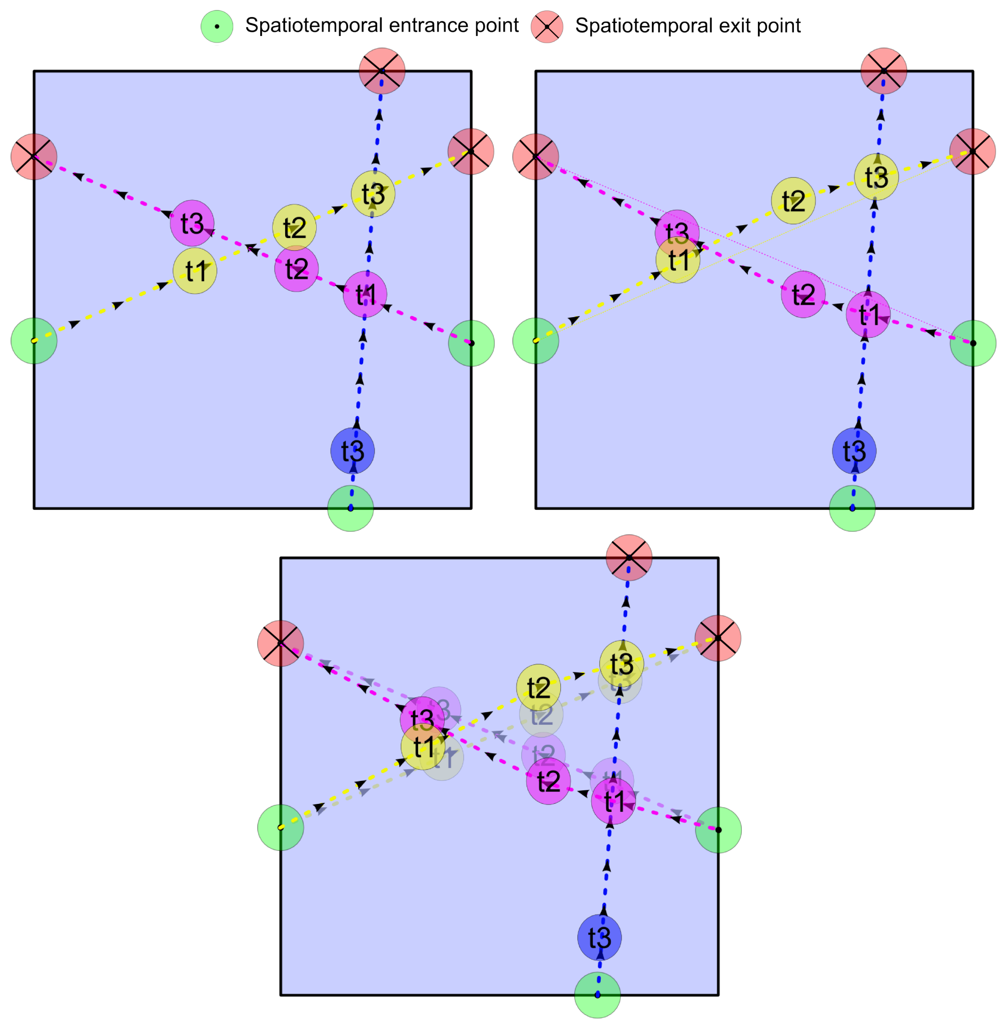


Figure 4.13: Collision Handling (left)A collision is currently identified between the yellow and pink trajectories at timestep t_2 (right) The two trajectories are deformed to handle the collision (bottom) Overlayed difference between the two trajectories.

constraints.

1. Boundary control points can't be moved, so if a collision involves one of them, the other point has full responsibility for moving. For this reason, if a collision happens between two boundary control points, such a collision can't be avoided. This is the importance of generating the points using the Poisson Sampling Method.
2. Points should not go over the boundaries of the patch. This is another reason for which collisions near the border are hard to avoid. When one of the points involved in the collision is near the border and the force is trying to pull it outside the path, it receives the same treatment as the boundary control points, and doesn't move, giving full responsibility to the other point for moving. When both points are trying to be pulled out of the patch (as near the corners) two options are presented: letting the collision happen, or, allowing one of the points to go out. For purposes of easily detecting these cases, in this work most of

the time, the second option was chosen.

Another aspect to have in mind is the speed of the points. We don't want to move much the points that are already moving too fast or we take the risk of make them move even faster. So the weight for the moving responsibility falls more on the slower point. Last thing to consider is that points will be moved just in spatial position, not in time.

For control points $\mathbf{p_i}$ and $\mathbf{p_j}$ the displacement vectors resulting from repelling forces will be:

$$\mathbf{F}_i = R(\phi) * \Delta\hat{\mathbf{p}}_{i,j} * \alpha * w_i \tag{4.16}$$

$$\mathbf{F}_j = R(\phi) * \Delta\hat{\mathbf{p}}_{j,i} * \alpha * w_j \tag{4.17}$$

$\Delta\hat{\mathbf{p}}_{i,j}$ is the normalized vector connecting the two points $(\Delta\mathbf{p}_{i,j} = \mathbf{p}_i - \mathbf{p}_j))$, $\alpha = r_i + r_j$ is the sum of the two agents' radii and defines a threshold value for minimum distance[2], $R(\phi)$ is a small random noise rotation matrix to help prevent infinite loops $(\phi : -0.5 \leq \phi \leq 0.5\ rad)$, since even when a control point returns to a previous position, this small random rotation allows the displacement vector to be slightly different than before. Finally w_i is a weight to reduce speed artifacts and prevent agents from leaving the boundaries of the patch:

$$w_i = \begin{cases} u_j/(u_i + u_j) & \text{if point stays in patch} \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

u_i and u_j are the speeds of trajectories τ_i and τ_j.

The new positions of the points will be defined as: $\mathbf{p_{inew}} = \mathbf{p_i} + F_i$ and $\mathbf{p_{jnew}} = \mathbf{p_j} + F_j$.

Up to now, $\mathbf{p_{inew}}$ and $\mathbf{p_{jnew}}$ are virtual or auxiliary points, since they are not part of the actual trajectories. Next step is adding them.

To add $\mathbf{p_{inew}}$, we look in trajectory τ_i the two contiguous control points $\mathbf{p_a} = \{x_a, y_a, t_a\}$ and $\mathbf{p_b} = \{x_b, y_b, t_b\}$ such that $t_a \leq t_{inew} \leq t_b$. After identifying the segment in which p_{inew} lies, a new point is constructed in the middle. For computational reasons, it is very expensive to add new points each time, since more control points involve more time for calculating and updating the distance matrix. So, we define a threshold of time tt, and if $|t_a - t_{new}| < tt$ then instead of adding the new point we modify the existing one: $x_a = x_{new}$, $y_a = y_{new}$. Notice how t_a is left the same. There are some aspects to consider:

1. If distances from t_{new} to t_a and t_b are both below the threshold we choose the one closer to t_{new} to be modified.
2. When $\mathbf{p_a}$ (or $\mathbf{p_b}$) is a boundary control point of the path, modifying it is not a possibility. So for those cases a new point is always added.
3. When $\mathbf{p_a}$ is the first control point in the trajectory and it is modified, the last point in the previous trajectory in the path is also modified. Remember that a path is formed

[2]In our implementation $r_1 = r_2 = r$, making the threshold constant.

by a sequence of trajectories and they must have spatial continuity in their endpoints (Figure 4.14). This condition is symmetric for $\mathbf{p_b}$

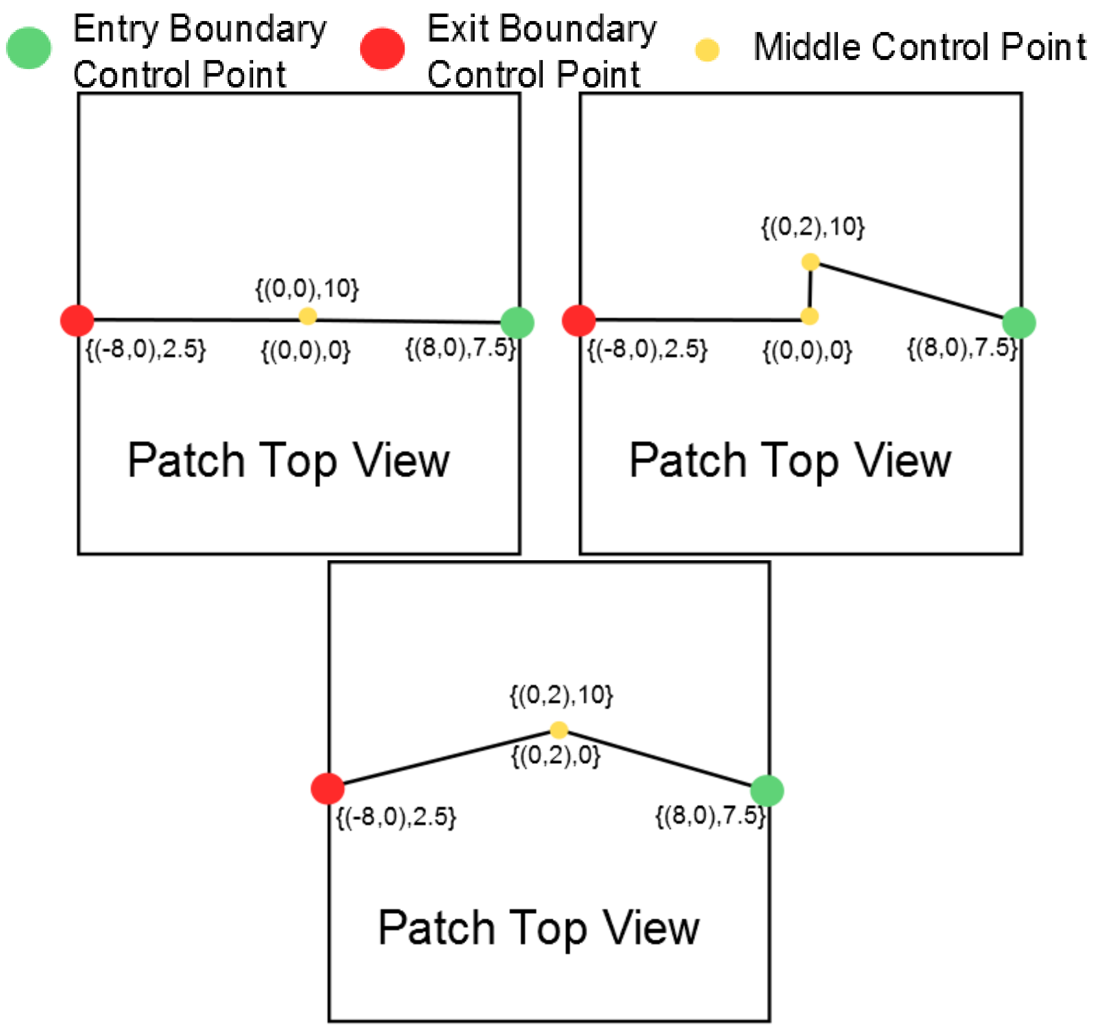


Figure 4.14: **Moving Control Points** (top left) Original path $Path = \{\tau_1, \tau_2\}$, $\tau_1 = \{\{(8,0), 7.5\}, \{(0,0), 10\}\}$, $\tau_2 = \{\{(0,0), 0\}, \{(-8,0), 2.5\}\}$. Notice how connecting endpoints in trajectories have the same spatial coordinates. (top right) If only one of the control points in the path is moved, discontinuities can be created. Following this path, in the animation the result is a point jumping instantly from one position to another each time the time reaches the period. (bottom) Both of the connecting endpoints are moved to the same position to avoid discontinuities.

4.3.2.2 Tension Forces

Intuitively, tension forces will make our paths similar to elastic bands. Imagine the path is originally a non deformed elastic band. After pulling the band in some points, it deforms, and some counteracting forces (tension forces) start trying to get the band to its original shape. If we let go the pulled points the band eventually returns to its original shape. The time it lasts to do so depends on the elasticity of the band.

In this work, tension forces acting over the path are presented as displacement vectors acting over all the middle control points of such path. The displacement vector applied to each control point depends directly on its neighboring control points and the speed of agents passing through it.

The neighboring control points of another point are the two points next to it in the path even if they don't belong to the same trajectory. When one of the points is the ending point of the trajectory, the neighbor point is the **second** one on the next trajectory (not the first, since the first is practically a copy with their time position translated by the period) and for purposes of calculations, we add a period to the time coordinate. In Figure (4.14) the neighbors of $(0, 2, 10)$ are $(8, 0, 7.5)$ and $(-8, 0, 12.5)$. We act similarly for the points at the start of trajectories, the neighbors of (0,2,0) are $(8, 0, -2.5)$ and $(-8, 0, 2.5)$

Each neighboring point adds a pulling force towards them. Then, both forces are added and then the resulting force is scaled so it doesn't overshoot over its original path or go outside the boundaries (Figure 4.15).

Tension vectors are defined as follow (relative to control point (x, y, t) and neighbor point (x_{nb}, y_{nb}, t_{nb}))

$$T = \beta * \frac{V}{\|V\|} \tag{4.19}$$

$$V = (x_{nb} - x, y_{nb} - y) \tag{4.20}$$

$$\beta = \begin{cases} (u - u_{cft})^3 & \text{if } u > u_{cft} \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

$$u = \frac{\|V\|}{|t_{nb} - t_1|} \tag{4.22}$$

β is a scalar that measures how far from the comfort speed is the actual speed between the two points.

The displacement vector associated with these tension forces will only move the point spatially. Displacing the point in the time dimension would require further refinements that is left for future work.

After getting the Tension Forces T_1 and T_2 from both neighboring points, the displacement vector T_{dis} is calculated:

$$T_{dis} = \gamma * \frac{(T_1 + T_2)}{\|(T_1 + T_2)\|} \tag{4.23}$$

$$\gamma = d * (1 - e^{-\|(T_1 + T_2)\|}) \tag{4.24}$$

Where d is the distance from the control point to the ideal segment (the line connecting directly the two neighboring points) going over the ray defined by $T_1 + T_2$. What gamma represents, is a transformation of the magnitude of the vector $T_1 + T_2$ from the space $[0, \infty)$ to the space $[0, d)$. This gamma function was chosen for its simplicity, but other functions transforming the spaces may also work. This way, T_{dis} is never going to overshoot. The updated control point (x, y, t) will now be $(x + \lambda * T_{dis_x}, y + \lambda * T_{dis_y}, t)$.

λ is the parameter representing elasticity. It is a number between 0 and 1 and depending on it, the movement coming from tension adjustments will be big or small.

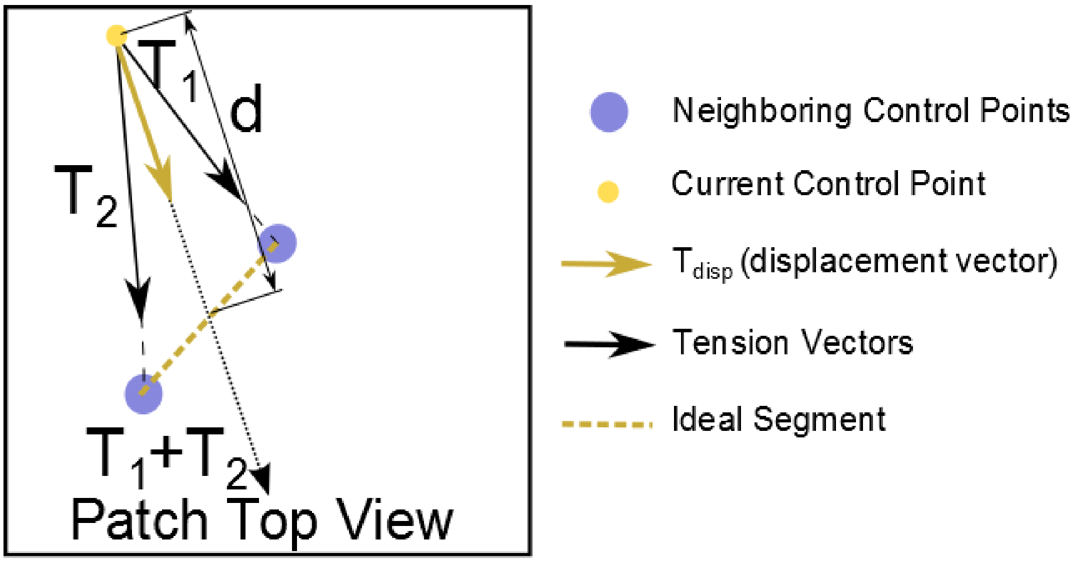


Figure 4.15: Tension Forces Tension forces are calculated from the direction of the current control point towards its neighboring control points in the path. The magnitude of these forces depend on how fast a point passing through that segment differs from the comfort speed. Both forces are added and are rescaled so they never go out from the ideal segment.

A point to notice when applying tension forces is that there are two main iterations. One calculating all the displacement vectors for the control points and storing them, and another adding the calculated vectors to the control points. It is done this way because we cannot modify the control points until we have found all the displacement vectors that are depending on them.

In this version of the iterative algorithm, tension forces are applied once in each iteration of the main Algorithm 4.2 and only for the paths that were modified in that iteration.

Tension has the added property that it propagates over time, but sometimes the algorithm ends before the tension can propagate nicely. Proposing a different end condition for Algorithm 4.2 that allows a nice tension propagation is left for future work.

4.3.2.3 Update Function

Last in this section, some notes regarding the update function from Algorithm (4.2). Updating the matrix M is computationally the most expensive part of the algorithm, since it is comparing pairs of segments between all the modified trajectories, and the quantity of those segments augment in each iteration.

A first version of this algorithm, the one who appeared in [Ramirez et al. 2014] only changed one pair of points in each iteration, the one having the minimal distance, before updating. Further examination and tests revealed that it is faster to change all pairs that are having a distance below the threshold of collision at the same time, and then update the matrix.

When updating M it is only necessary to change the rows and columns of the trajectories that were modified. This will allow for the last iterations (the ones involving just one or two collisions) to be much faster.

4.4 Obstacle Handling

This section describes how the avoidance of obstacles can be added to the previously described system.

Obstacles are static agents: Trees, benches, statues, or even people standing still. The main characteristic is that they dont move through the duration of the period of the patch. Thus, they are totally defined by two characteristics, their position and radius.

The first idea that comes naturally when handling obstacles is to treat them as any other trajectory, with the restriction that they are unmovable trajectories. This means, any other trajectory colliding with it has full responsibility to avoid it. In Equations 4.16 and 4.17, when the weight w for the displacement vectors are calculated, a special condition is added: if one of the trajectories belongs to an obstacle, its weight is set to 0, since the obstacle can not move by itself.

This simple idea has the setback of augmenting the possibilities of creating loops (Figure 4.16).

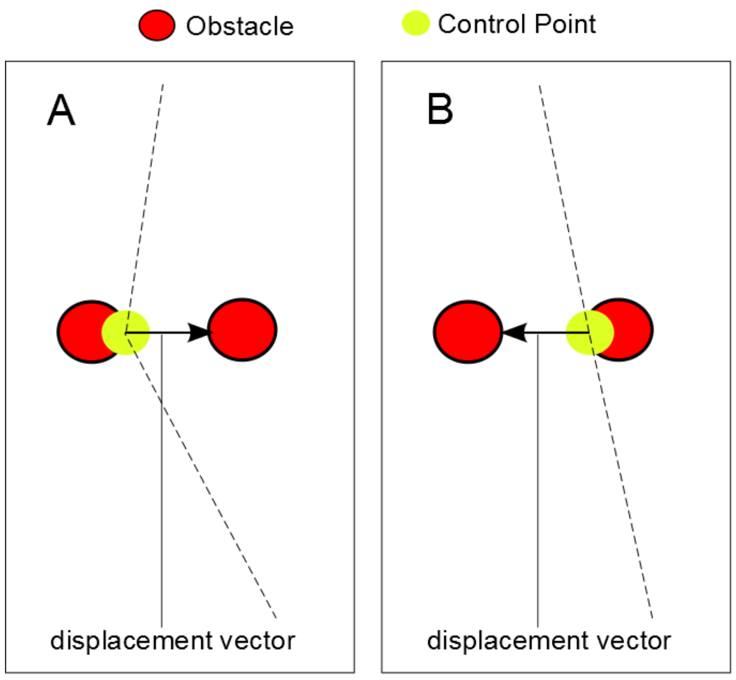


Figure 4.16: The control point in A moves to avoid collision, but it ends in collision with another obstacle in B. Next iteration moves back the control point to its original position, creating a loop.

To avoid this behavior, obstacles that are closer are grouped together.

4.4.1 Grouping Obstacles

One option to alleviate the problem of looping is to group obstacles together, then, consider the geometric center of the group of obstacles as the reference point the control point wants to get away. In the example mentioned before, this causes the control point to effectively avoid all obstacles (Figure 4.17).

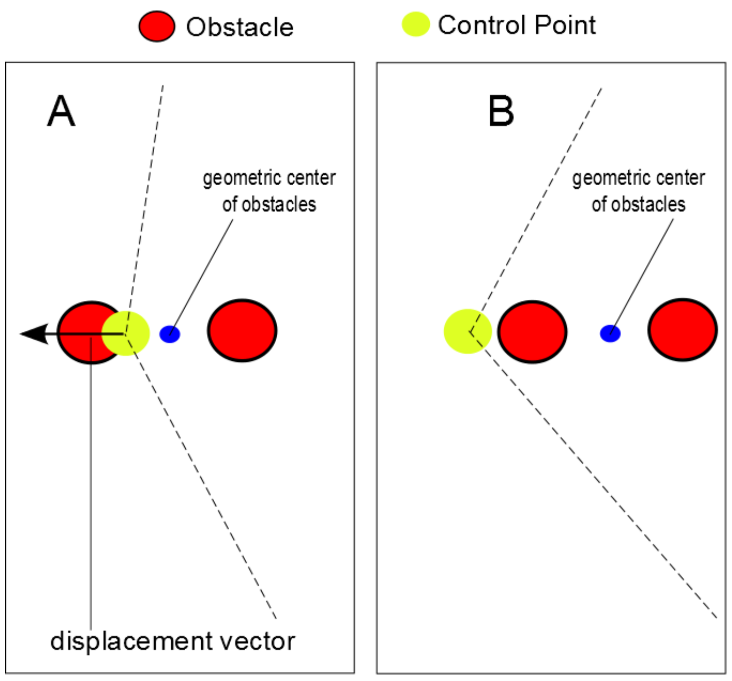


Figure 4.17: Control point now moves away from the geometric center of the two obstacles and avoids the collision.

The geometric center $\mathbf{c}$ for a group of n circles is calculated in the following way:

$$\mathbf{c} = \frac{\sum \mathbf{o_i}}{n} \tag{4.25}$$

where $\mathbf{o_i}$ is the center of circle i. This is also the equivalent of finding the center of mass of n particles with mass 1.

The following problem now arises, how to make the grouping (also called clustering) for the obstacles. Strict partitioning clustering, where each obstacles belongs to exactly one group may not work in some cases, since a lot of unnecessary information can be added in the process. Overlapping clustering doesn't have that problem, that is why it is preferred in this particular circumstance (Figure 4.18).

An overlapping cluster is defined by an obstacle, and all the other obstacles closer than a certain threshold, including itself. The threshold to decide if an obstacle is contained in the cluster of another obstacle is at least the sum of both obstacles radii and the diameter of an

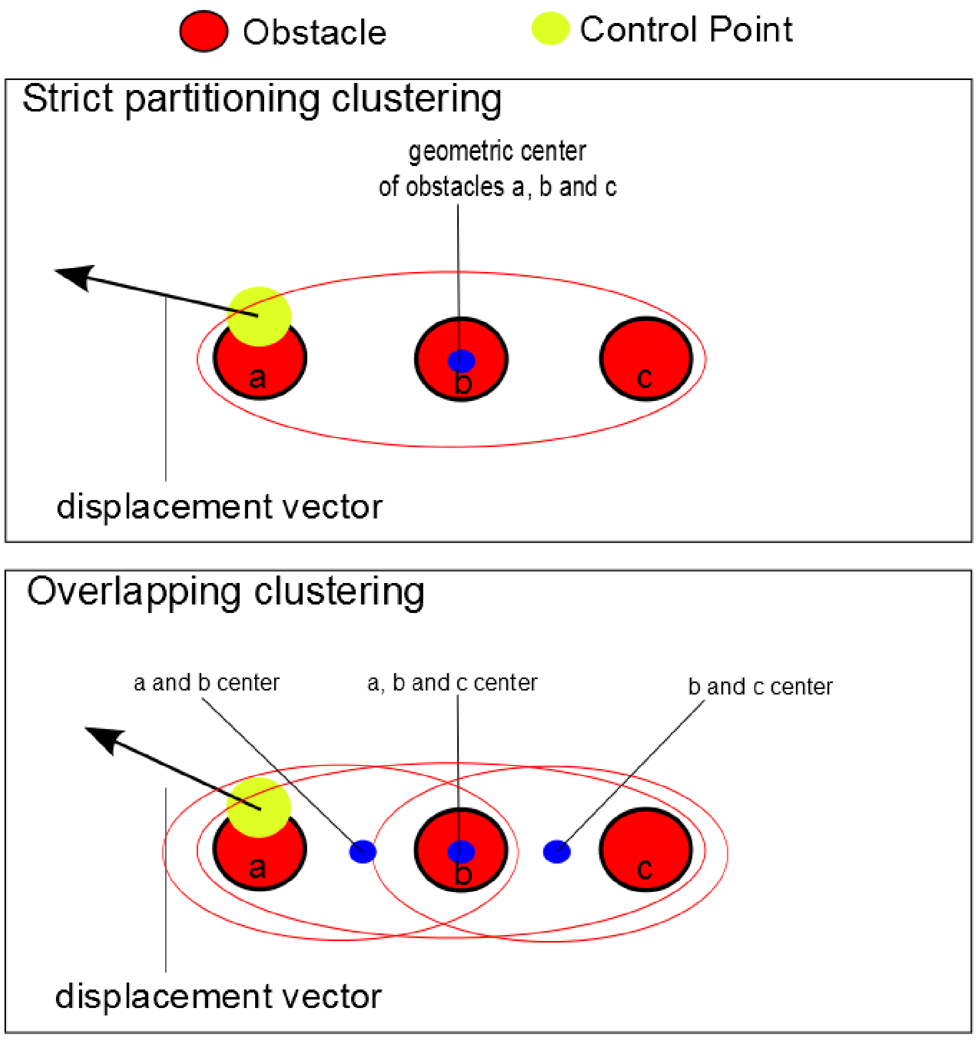


Figure 4.18: With strict partition clustering, obstacle c is taken into consideration for moving the control point, even though it is far away enough to not matter. With overlapping clustering, only the geometric center of obstacles a and b are considered, since they are in the cluster defined by obstacle a, who is the one colliding with the control point. Notice the slight change in the angle of the displacement vector.

agent passing through. But since a person usually doesn't choose to pass through two very close objects, this threshold can be relaxed by adding an additional diameter.

For the purposes of the iterative algorithm, after the clusters associated with each obstacle are created, the geometric center is also calculated and stored. This is the main data that will be used when pushing a control point away from collision with obstacles. It must be noted that this method for avoiding obstacles works better with sparse scenarios. Dense cases must be processed differently, this is discussed in the future work section.

4.5 Group Handling

In crowd simulation, two or more people walking together add greatly more realism.

This section explains how group behavior can be added to this approach. The main idea to achieve this will be adding extra trajectories to the sides of the original ones and applying the collision avoidance algorithm over them again. Notice how the adding the new trajectories doesn't produce any displacement to the original ones, this is because trajectories are now sorted by levels of priority, older trajectories having more priority (or level 1), and newer ones, less priority (level 2) . This way, level 1 trajectories can still be simulated by themselves without worrying about them avoiding ghost trajectories(Figure 4.19).

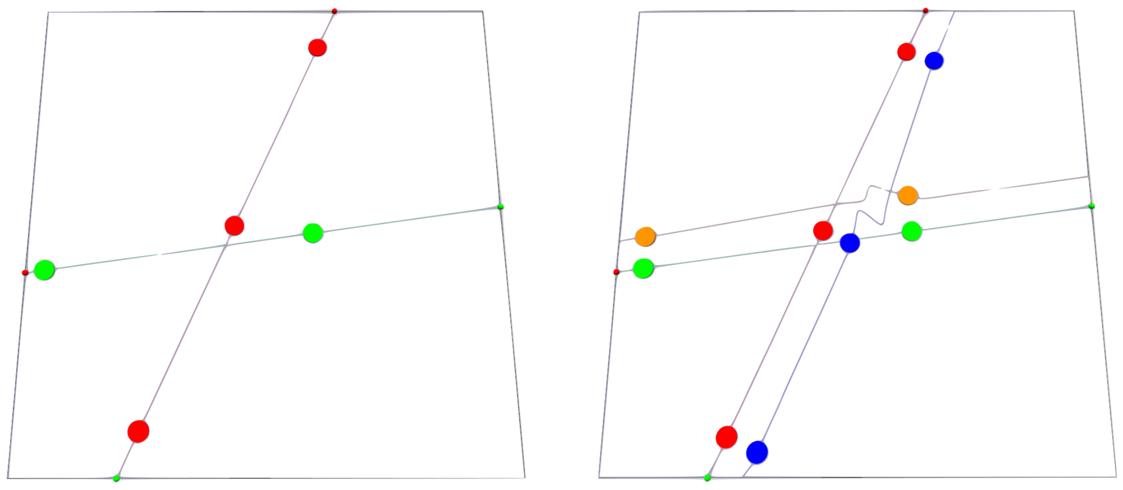


Figure 4.19: Walking by pairs (Left) Level 1 trajectories. (Right) Level 2 trajectories are added to the sides. Level 1 trajectories don't change.

Separating trajectories by levels will allow later to choose some of them to be turned off and have more diversity on the types of patches that can be made. Of course, the function one step above who is in charge of pasting patches together will be in charge of deciding which entry and exit points are chosen to have their level 2 or higher level trajectories turned on.

4.5.1 Plug Definition

The first step is to create the plugs at each entry and exit control point. A plug defines the group behavior of the agents following the trajectories. It could be two people walking side by side, or one agent walking behind another.

Formally defined, a plug is the set $\{\mathbf{cp}, \mathbf{V}\}$ where $\mathbf{cp}$ is an entry or exit control point and $\mathbf{V} = \mathbf{v_1}, ..., \mathbf{v_p}$ is a list of vectors indicating where the companion entry and exit points will be created in the pattern.

Entrance (or exit) point with priority 2 will be $\mathbf{cp}+\mathbf{v_1}$ (Figure 4.20).

A plug can have any number of vectors, but there can only be two types of plugs in total, the plug for entry points and the plug for exit points. Plug for exit points is actually constructed from a reflection of the entry plug centered around the original control point (Figure 4.21).

This will allow for agents that enter to the right (or left) of another agent, to also exit the

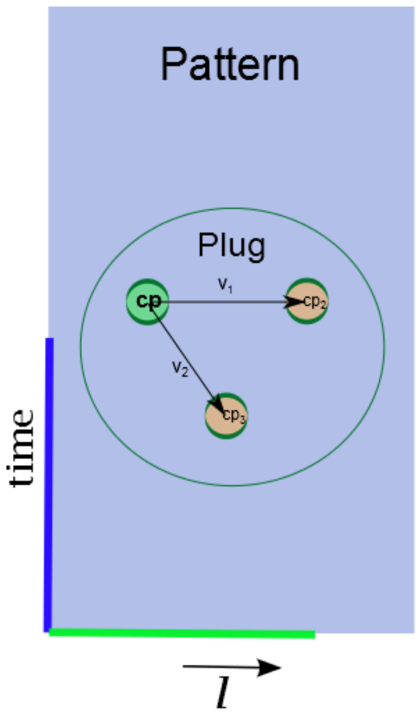


Figure 4.20: Plug. A 3 point plug is created by adding 2 vectors to the original control point. The priority of the companion points is defined by the order of the vectors creating them. The second point defined by $\mathbf{v_2}$ will be used to create a trajectory with priority 2 and the point defined by $\mathbf{v_3}$ will be used for a level 3 trajectory.

patch by the same side of such agent, which reduces the number of crossings and therefore, the number of possible collisions. A bit of convention needs to be added here so right and left are always consistent for all patterns:

Vectors delimiting the area of a patch have to be given (or redefined if not) in a counter clockwise direction. So, figures like (Figure 4.20) and (Figure 4.21) become exterior views of the patch. Notice that with this added condition, corresponding vectors of touching patches are mirrored. This ensures that level 2 and bigger level entry points match with level 2 and bigger level exit points in the two corresponding patterns.

Some other restrictions have to be added when using plugs. A plug cannot place an additional point outside the pattern, so if one of the left or right boundaries are reached, the plug approach can't be used. To avoid this, when creating random entry and exit points, it must be ensured that they are far enough from the boundaries, giving enough space for the plug to be created later. Also, if the Poisson sampling method is going to be used to create space between control points, a much bigger radius has to be given. Even though two original control points are well spaced, the control points created by the plugs can be placed very close to each other, creating collisions very hard to avoid (Figure 4.22). For the same reason, vectors defined by the plug must space well each new companion control point from all others in the same plug.

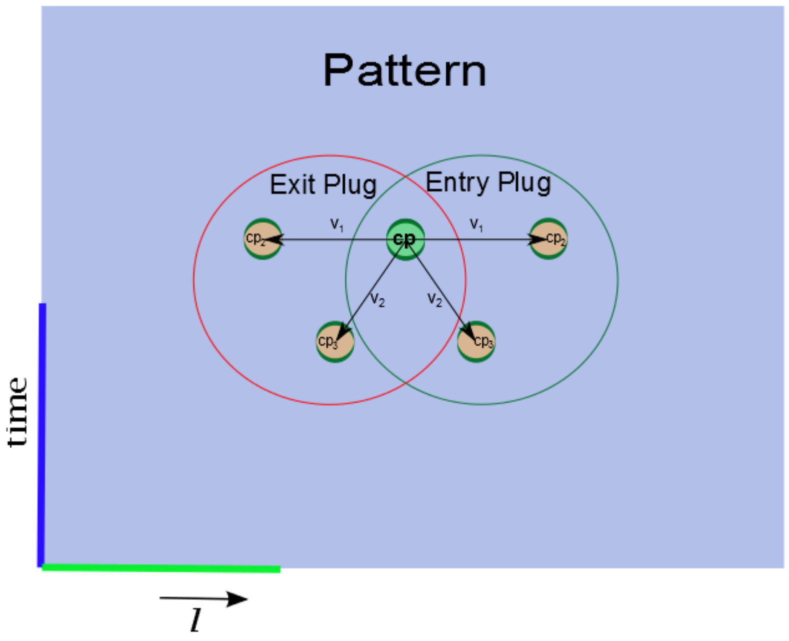


Figure 4.21: Entry and Exit Plugs. Depending on which kind of control point we have, the corresponding plug is created. Notice that if **cp** is an entry point, it is giving us its back, and $\mathbf{cp_2}$ from entry plug is placed to its right. If **cp** is an exit point, then it is facing us, and exit plug places $\mathbf{cp_2}$ also to its right, even though it appears to be in the left side in this perspective.

Notice that having a bigger radius for the Poisson sampling, limits considerably the number of original control points that can be placed in a good position.

One last thing to have in mind when using the plug approach is the temporal restriction. When a control point is near the upper or lower boundaries, the vector of the plug may try to create a companion that surpasses the temporal restriction. This can be fixed by subtracting, or adding the period of the patch to the coordinate trespassing the boundaries (Figure 4.23). More details on this can be found on the next subsection.

4.5.2 Middle Plugs

The previous section detailed how the plugs for entry and exit points are created. After the set of original control points are paired and the initial linear paths are constructed between them, the companion entry and exit points created by the plugs are paired implicitly. If A and B are a pair, then companions of A are matched with companions for B according to their level.

However, initial paths between companion points can't be created in a straightforward way using the same algorithm that the original control points use. There are two main reasons: First, companion paths may be of different length than the original ones, which may cause different speed corrections for them (Figure 4.24). Second, the going over the edges correction may pull the path towards a direct collision (Figure 4.25).

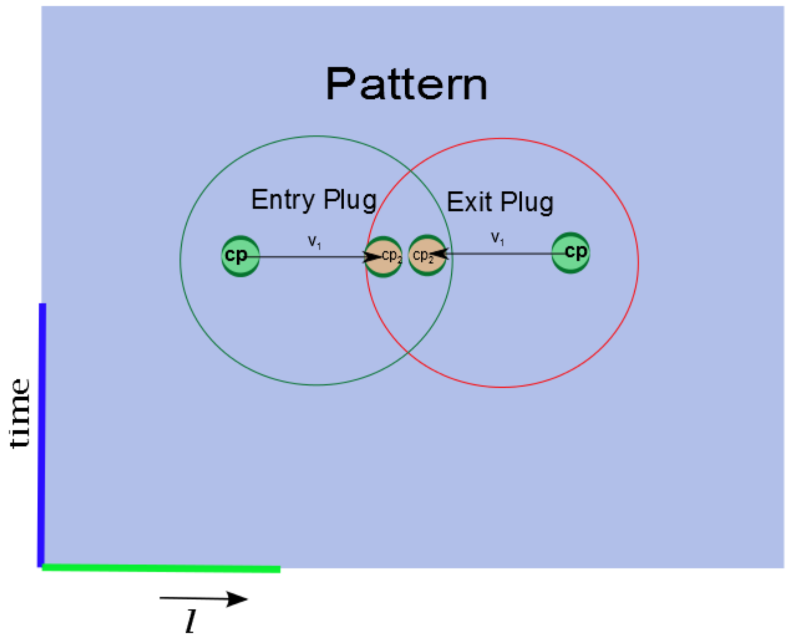


Figure 4.22: Bad Initial Placement. Even though at the beginning the original control points were far enough from each other, the corresponding companion points created by the plugs are very close to each other, augmenting the possibilities of a collision.

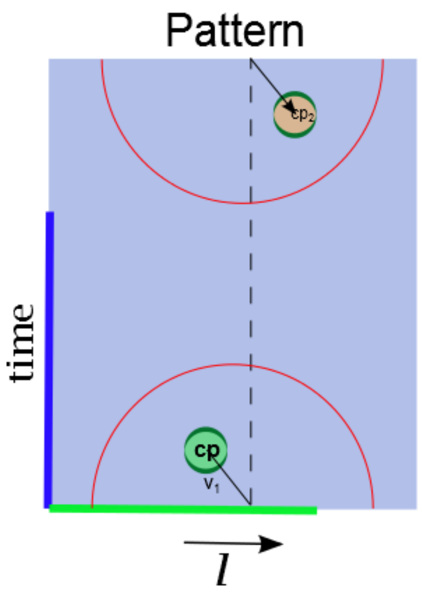


Figure 4.23: Bad Initial Placement. Plug is split into two parts, but because of the periodicity properties of a patch, this won't matter for the simulation.

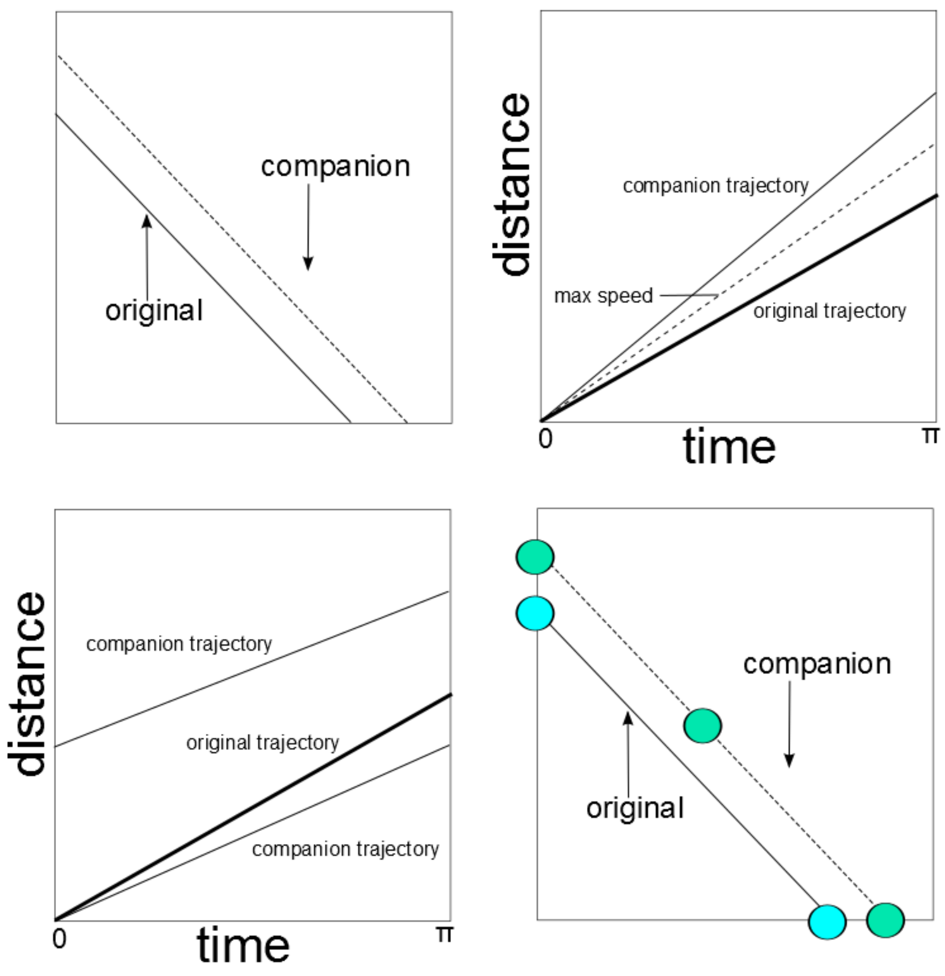


Figure 4.24: (Top left) Companion path travels a further distance than the original. (Top right) Companion path is over the max speed limit. (Bottom left) Companion path is split into two so it stays behind the maximum speed. (Bottom right) Agents at time=0, notice how the companion patch now doesn't follow as closely the original path because of the splitting.

To avoid these problems, middle plugs are used to adjust the companion paths to the original trajectories. Middle plugs are used in each middle control point of the path, because they point to each break or modification that was done due to corrections. Middle plugs are defined in the same way as before, with the only change being the system of coordinates. The axis that was before the vector limiting the patch is changed to a vector pointing to the right of the agent. To get this new axis, first we get the vector pointing directly behind the agent. This vector is easily constructed by subtracting the current point of the path from the next control point in the path, then a 90 degree rotation is applied (Figure 4.26). Linear paths can now be created by connecting the middle companion control points in a sequence of linear trajectories.

Some of the companion trajectories will be shifted in time, producing endpoints that go backwards in time, for example:

Assume the original trajectory starts at 0 and ends at 5s. The plug has a vector creating

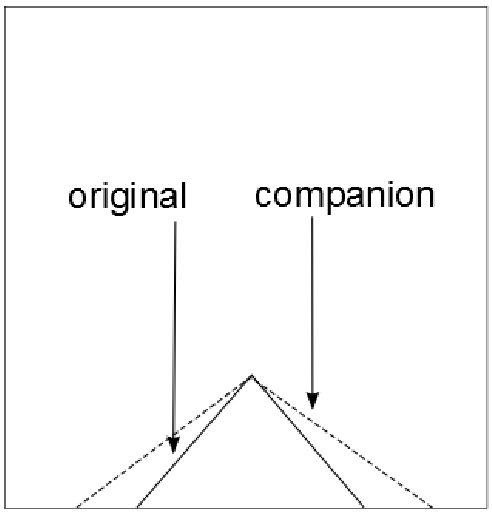


Figure 4.25: The companion path gets pulled exactly to the same spot than the original, creating a collision.

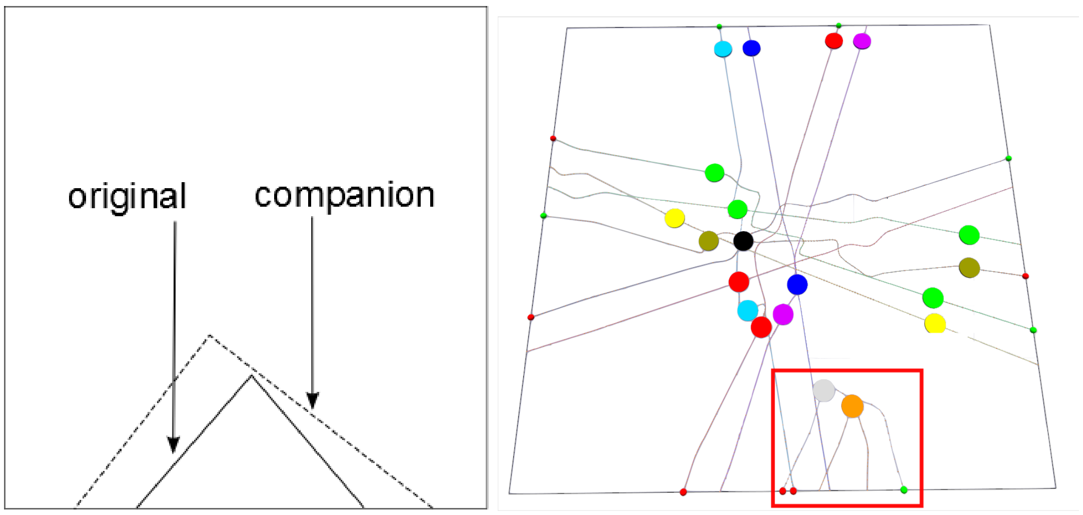


Figure 4.26: (Left) The companion path is adjusted with a middle plug created at the middle control point. (Right) Section of a patch using this modification after the collision and smoothing algorithms are applied.

the companion trajectory 1 second behind, so it lasts from -1s to 4s. Since this is not possible, the period (let's say it's 10s) is added, now the companion trajectory goes from 9s to 4s. As described in a previous section, a "going backwards in time" correction can be applied, splitting the companion trajectory into two, one going from 9s to 10s and a second one going from 0s to 4s. For this reason, companion paths will sometimes have a bigger number of trajectories, but each of them will still have the same speed as if it wasn't shifted in time, so no further modifications are necessary.

4.5.3 Collision Avoidance Algorithm Adjustment

This last subsection will discuss a few details about how to modify the avoidance algorithm to handle collision of different priority trajectories.

Much like obstacles in previous section, a trajectory with a bigger priority will not move when in collision with another one with a lower priority. All the avoidance will have to be done by the lower priority trajectory. When trajectories with same priority are in collision the algorithm will behave as normal. One thing to have in mind is that an extra parameter has to be added indicating which level of priority a trajectory has. This parameter has to be carried over at the end of the algorithm so later, at the moment of connecting patches, it can be known what priority level has each trajectory.

The idea behind this is that connected trajectories between patches will define circuits, and then those circuits can be chosen to have 2 or more lanes (depending on the number of extra levels of trajectories). This has not yet been implemented but it is planned for future work.

4.6 Smoothing Algorithm

Adding and modifying the control points in the trajectories may end up in sharp changes in direction. For this reason, a smoothing process is applied to the output trajectories of the avoid collision algorithm. The smoothing we are looking for has the following properties:

1. The curve has continuity C^2 . That means first and second derivatives are the same between two pairs of connecting curves.
2. The curve passes through all the control points of the trajectories.
3. The trajectories belonging to the same path are smoothed together.

The first thing to notice is that the structure of our trajectories can't represent curves, only linear segments, so in order to get close to a curve representation a finer re-sampling will be taken from the generated curves at the end.

The second thing to notice is that curves generated by the smoothing process can vary too much, which has two problems (Figure 4.27):

1. Collisions happening again with other trajectories.
2. New trajectories may vary too much from their original form.

To solve these problems, a threshold is created, indicating the maximum possible distance a new control point can be created afar from the original trajectories. If a point coming from the re-sampling is at a distance bigger than the threshold that point is not added to the trajectory.

To improve closeness to original shape of the trajectory, additional virtual control points called tacks are added before creating the spline in the trajectory. Tacks are put uniformly between every two control points in the trajectories (Figure 4.28). The bigger the number of the tacks added, the more restricted the spline will be.

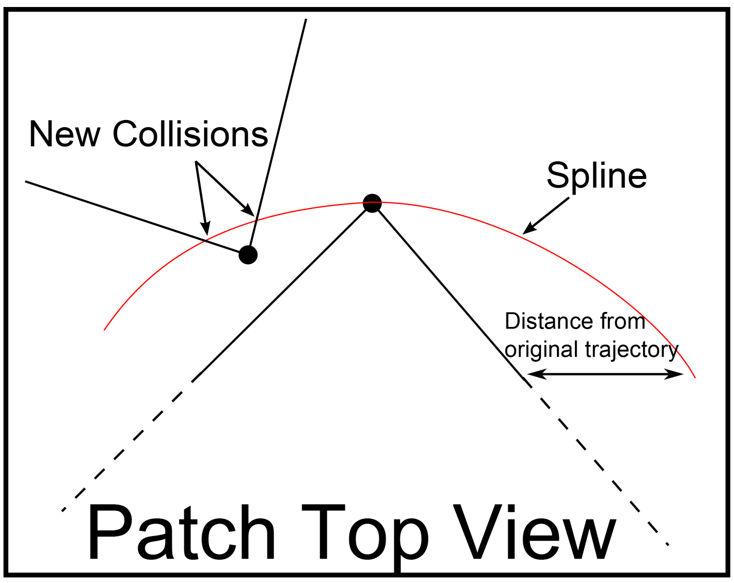


Figure 4.27: Smoothing Possible Problems. When a spline is generated only from the control points in a trajectory, it can lead to new collisions or to a curve whose new form is very different from the original trajectory.

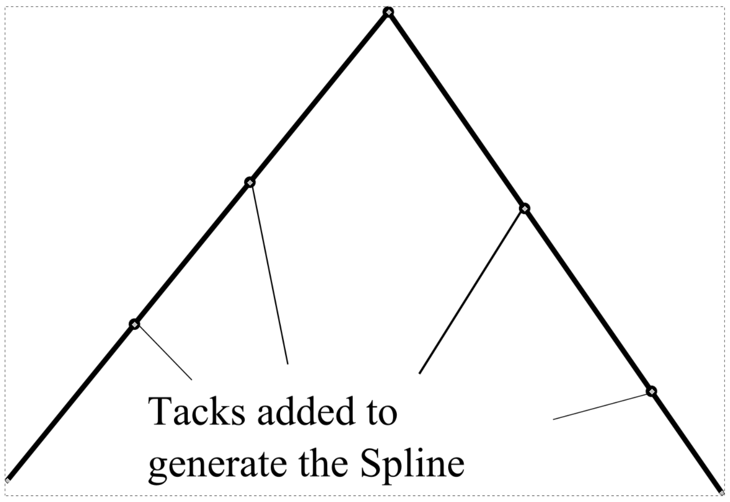


Figure 4.28: Tacks. Tacks are added to the trajectory before the creation of the spline, as a mean to preserve the original shape of the trajectory.

After adding the tacks, the next step is to merge all the trajectories belonging to the same path into a single trajectory. To do this, the control points in the first trajectory in the path are added, and next, for all other trajectories, the first control points are ignored (so there are not two repeated points) and all the others are added with the time coordinate changed accordingly, by adding multiples of the period π.

Merging Example: Path $P = \{\tau_1, \tau_2, \tau_3\}$ with $\tau_1 = \{(-8, 0, 5), (0, 0, 10)\}$, $\tau_2 = \{(0, 0, 0), (3, 3, 10)\}$, $\tau_3 = \{(3, 3, 0), (8, 5, 7)\}$ and $\pi = 10$, is transformed into: $P = \{\tau_1\}$ with $\tau_1 = \{(-8, 0, 5), (0, 0, 10), (3, 3, 20), (8, 5, 27)\}$.

Now, a cubic spline S that passes through all the $n+1$ points $\{(\mathbf{p}_1, t_1), ..., (\mathbf{p}_{n+1}, t_{n+1})\}$ in that merged trajectory is created. S is formed by a sequence of n cubic equations $S_i(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3$ For each one of the trajectory's segments, the spline's coefficients need to be found under C^2 continuity restrictions:

1. Every spline associated with a trajectory between two consecutive control points $\mathbf{cp}_i = (\mathbf{p}_i, t_i)$ and $\mathbf{cp}_{i+1} = (\mathbf{p}_{i+1}, t_{i+1})$ must pass through those same points; i.e., $S_i(t_i) = \mathbf{p}_i$ and $S_i(t_{i+1}) = \mathbf{p}_{i+1}$.
2. The speed and acceleration at the control point $\mathbf{cp}_i = (\mathbf{p}_i, t_i)$ that connects two consecutive cubic equations S_{i-1} and S_i must be equal; i.e.,

$$\frac{\partial}{\partial t} S_{i-1}(t_i) = \frac{\partial}{\partial t} S_i(t_i) \tag{4.26}$$

$$\frac{\partial^2}{\partial t^2} S_{i-1}(t_i) = \frac{\partial^2}{\partial t^2} S_i(t_i) \tag{4.27}$$

These restrictions can be arranged in the form of a system of linear equations that can be solved using the Cholesky Decomposition Method [Nocedal and Wright 1999].

After finding the coefficients for the Spline associated with each path, a re-sampling is done. In order to not add many unnecessary new control points, these re-samplings are more dense near the inflection points of the trajectories (Figure 4.29). If one of the new re-sampled points is farther than a certain distance α from its original position, the point will not be added.

Finally, after getting all the new control points in the new trajectory, it is split again in order to maintain time constraints. For example, trajectory $\tau = \{(-8, 0, 2), (0, 0, 10), (0, 8, 13)\}$ is split into $\tau_1 = \{(-8, 0, 2), (0, 0, 10)\}$, $\tau_2 = \{(0, 0, 0), (0, 8, 3)\}$.

There are a few drawbacks to using splines for smoothing trajectories. Depending on the number of tacks, and the threshold of maximum displacement α from the original trajectory, it may happen that many of the points found by re-sampling into the spline are not added into the new trajectory, and the end result is not noticeably different than the original trajectory. Also, splines may not be the best curves for representing human trajectories, in the future, some other methods may be explored to improve the smoothing method.

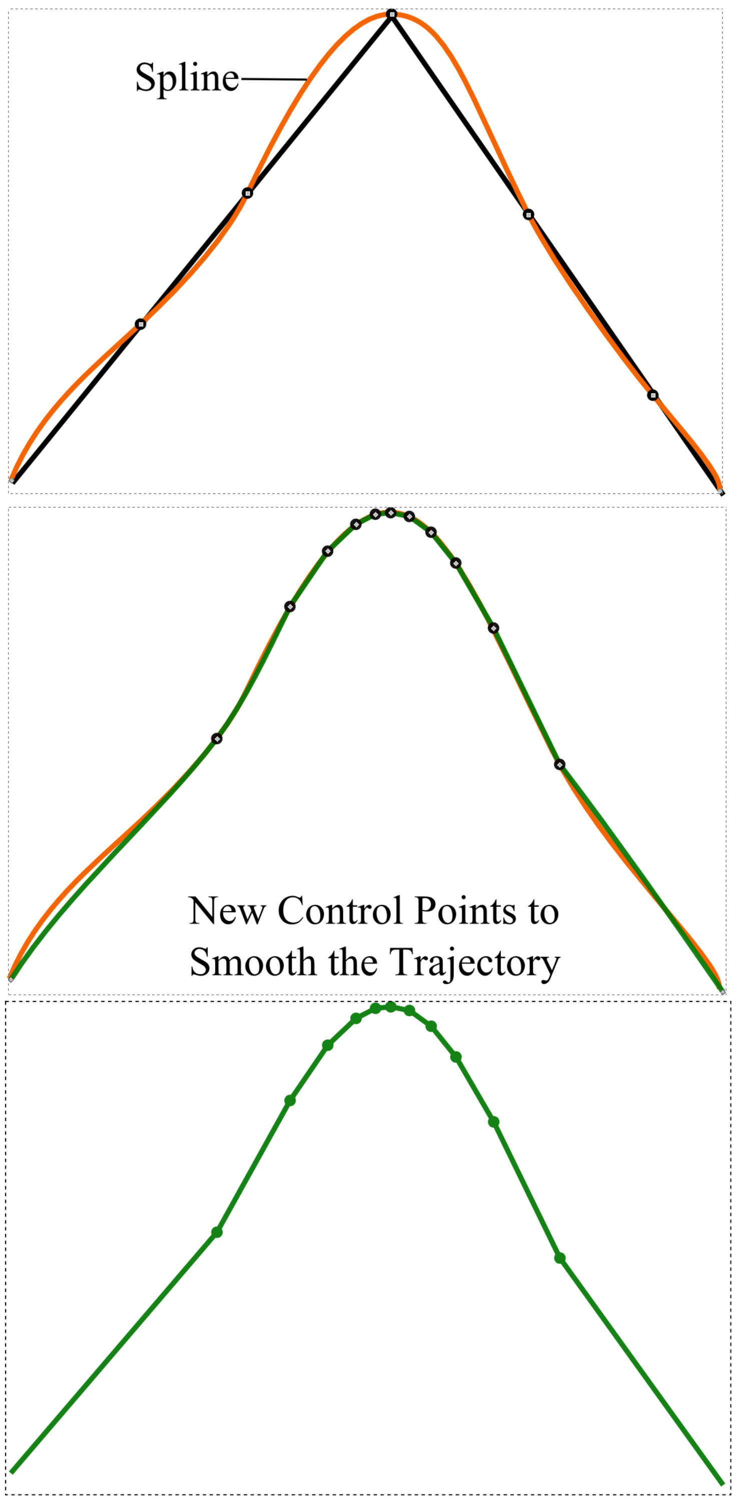


Figure 4.29: Smoothed Trajectories. (Top) Spline is generated from the continuity restrictions. (Middle) A new re-sampling is done, more dense near the points of inflection. (Bottom) Final trajectory is still piecewise linear, but with a bigger resemblance to a curve.

5 Results and Comparison with Other Approaches

This chapter will present the results for the techniques developed, as well as comparisons with other techniques.

5.1 Results

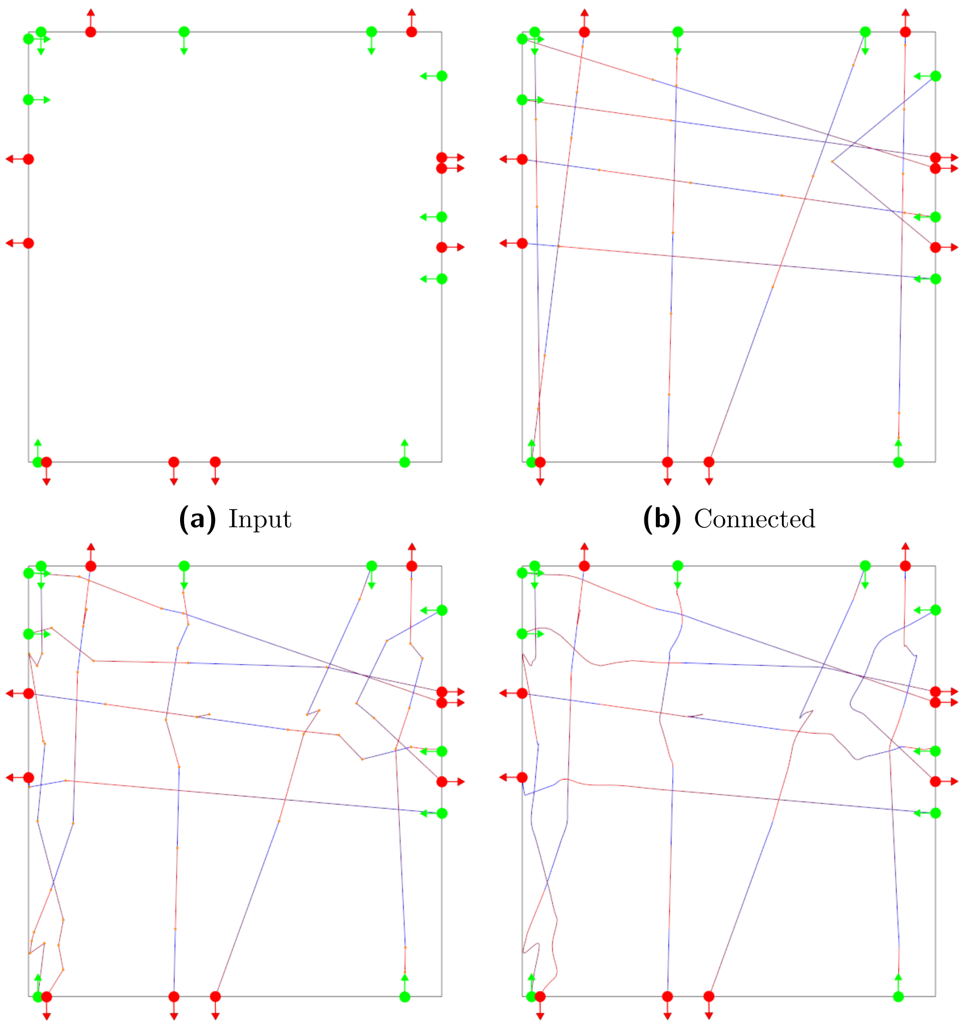
The proposed algorithm was integrated into a crowd patches platform in C++. The next experiments were done in a patch of size $A = 16m * 16m$ and period $\pi = 10s$.

Overall Methodology: The first result will be an overall look of the output of the algorithm (Figure 5.1).

Random vs Stable Matching: A comparison between the Random Matching and the Stable Matching (Figure 5.2).

3D Visualization of Trajectories: Some trajectories displayed in the 3d representation of a patch. Trajectories will never be represented as 3d strings in the final animation, but this type of visualization helps to a better understanding of the structure of a patch, as well as some of the steps of the algorithm (Figure 5.3).

Density Adaptability: Different examples of a patch with different number of entry and exit points (Figure 5.4). Density depends on different aspects, the number of entry and exit points, the size of the patch, time of the animation, and even on the maximum allowed speed. This last dependency is because trajectories are split into two or more trajectories as an initial correction (see Figure 4.10 in previous chapter). Usually, no more than 20 points are necessary on a patch. Remember, the purpose of a patch is not to recreate the whole animation, but just a small part of it.


(a) Input

(b) Connected

(c) Collision Free

(d) Smoothed Out

Figure 5.1: Method example. The proposed method starts from (a) a set of control points in the boundaries of an empty patch, (b) interconnects them in an optimal way, (c) resolves any temporal collisions and finally (d) smooths the trajectories.

Obstacles Handling: Obstacles are positioned at random at the middle of the patch. Trajectories avoid them successfully. Because obstacles never move, in some cases, it is still hard to avoid collisions, so dense case scenarios should be avoided (Figure 5.5).

Group Behavior: By creating companion paths with different priority it is possible to simulate groups of two or more people walking together. For this example, a slightly bigger patch, $A = 24m * 24m$ was used. This is because paths need more space in order to accommodate the companion trajectories (Figure 5.6). Sometimes, companion paths stray away from the original trajectories in order to avoid some of the collisions, and the cohesiveness of the group is lost for some moments. Further studies have to be done in order to say if this is a behavior expected from real people or not.

Performance: Different number of patches were created requiring different number of entry

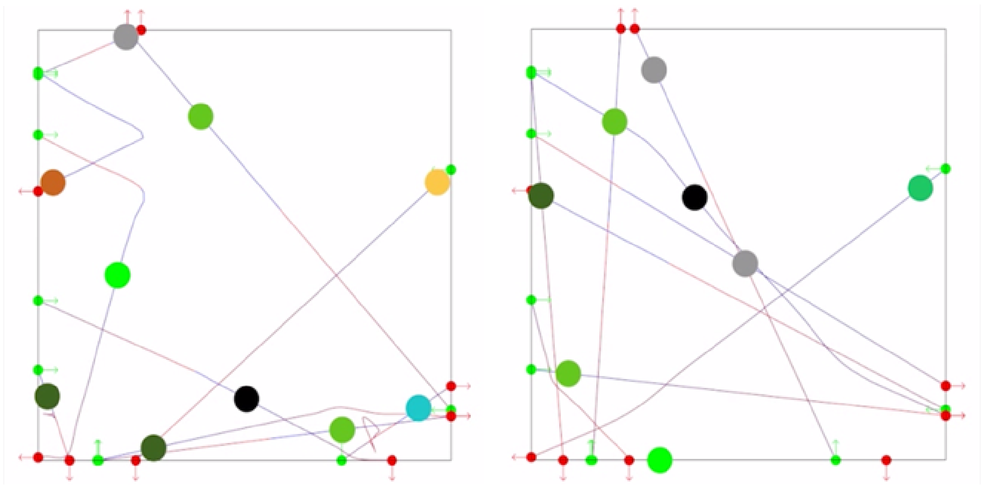
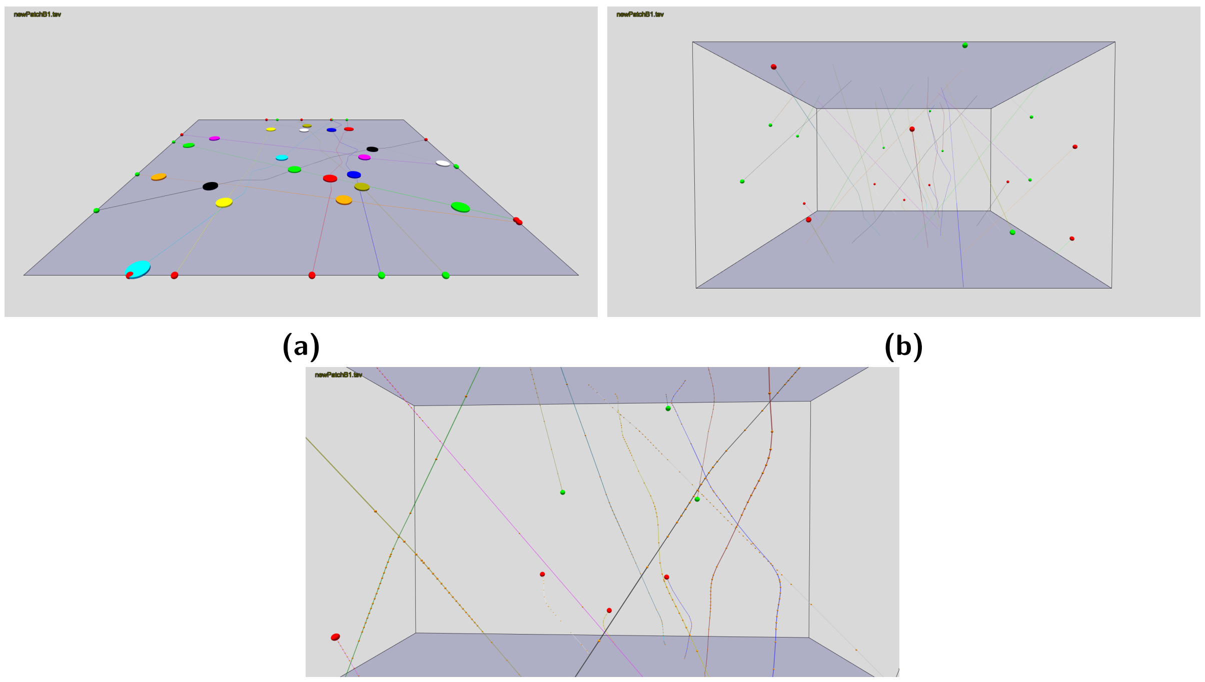

(a) Random Matching **(b)** Stable Matching

Figure 5.2: Random vs Stable. In this example, the random matching connects some endpoints belonging to the same pattern, whereas the Stable Matching finds paths whose endpoints are in different patterns.

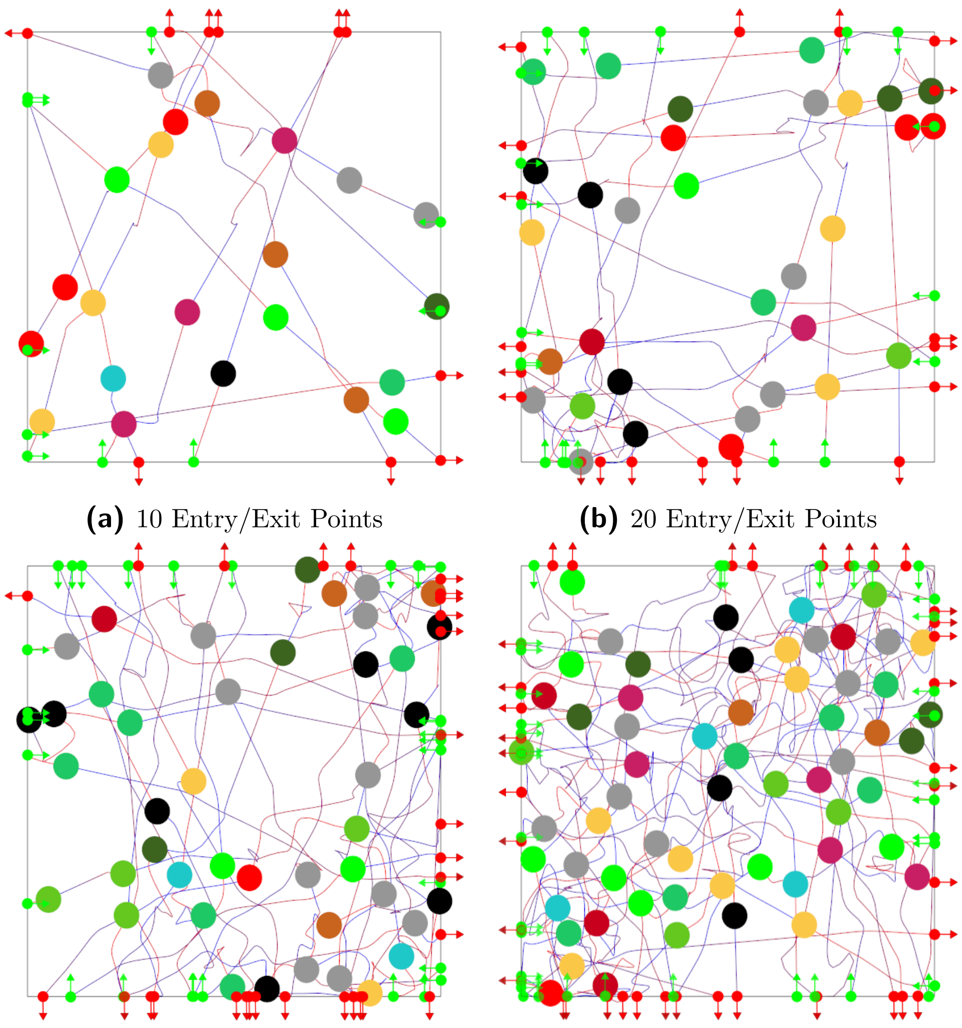


(a) **(b)** **(c)**

Figure 5.3: 3d Visualization. (a) Flat view of a patch from a tilted perspective. (b) Trajectories represented in their 3d representation. Notice how trajectories that go outside the patch in the ceiling enter again at the floor at the same position. (c) Close-up of the trajectories. In orange, the control points are shown. Notice how the density of the orange points along the trajectory changes depending on its curvature. This is because of the smoothing step.

and exit points, from 1 to 29 pairs. Period was $20s$ and the the patch had an area of

$$A = 12m * 12m$$



(a) 10 Entry/Exit Points

(b) 20 Entry/Exit Points

(c) 30 Entry/Exit Points

(d) 40 Entry/Exit Points

Figure 5.4: Density Adaptability. The proposed method can generate collision free trajectories for many situations; from sparse to very dense.

(Figure 5.7). The tests were done in a laptop with a quad core processor 2.4GHz, and 12GB of RAM, with the program using around 15% of the total processor utilization. A maximum number of iterations 300 was used. No obstacles or group behavior was used. For the cases where a solution is not reached, there exist two possible solutions:

1. Ignore tensions forces acting on the trajectories. This will decrease the quality of speed in some trajectories.
2. Increment the maximum speed of the points. This will make less initial splits into initial trajectories, making the patch less dense.

Animation: Some examples of the final results. First, an example of a patch with rendered humans(Figure 5.8). Second, an example with multiple patches together (Figure 5.9).

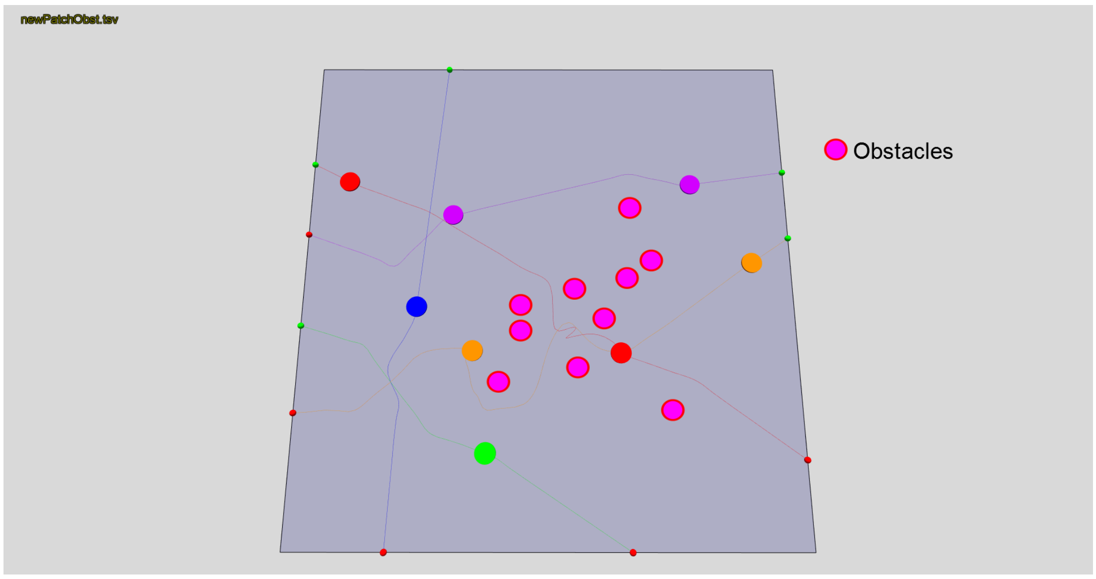


Figure 5.5: Obstacle Handling. 5 Pairs of Entry/Exit Points and 10 obstacles were used as the input of the patch. A bigger number of obstacles increment the patch density, and special care has to be taken in order to not totally block the way between two patterns.

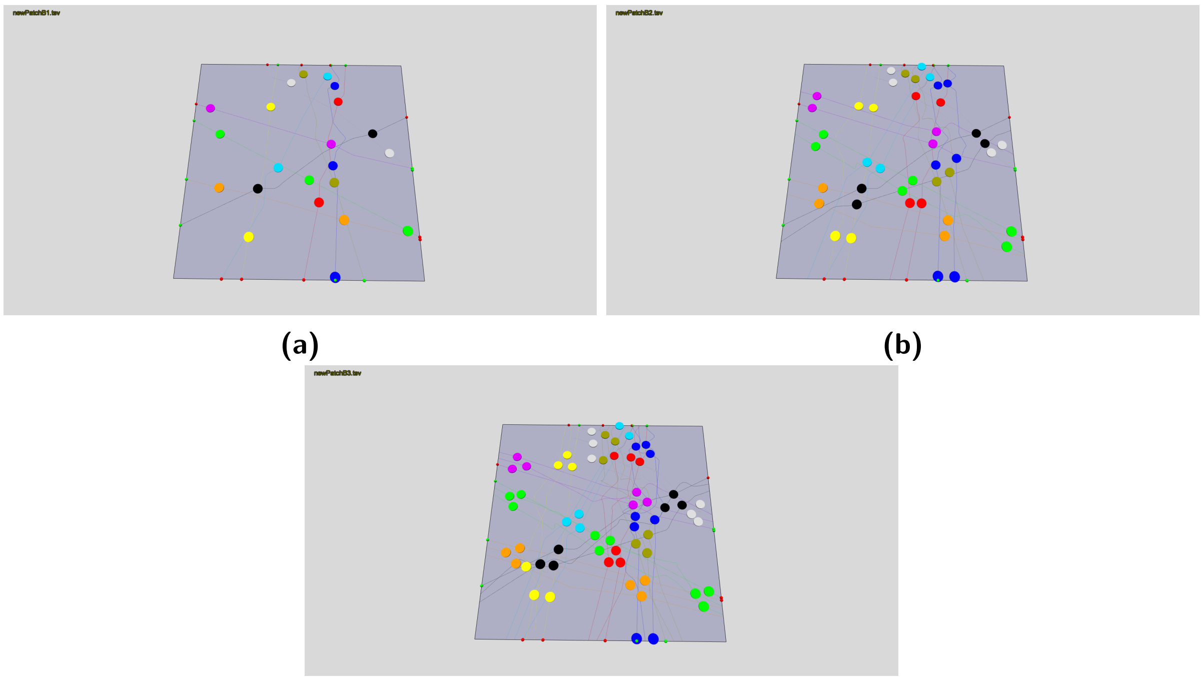
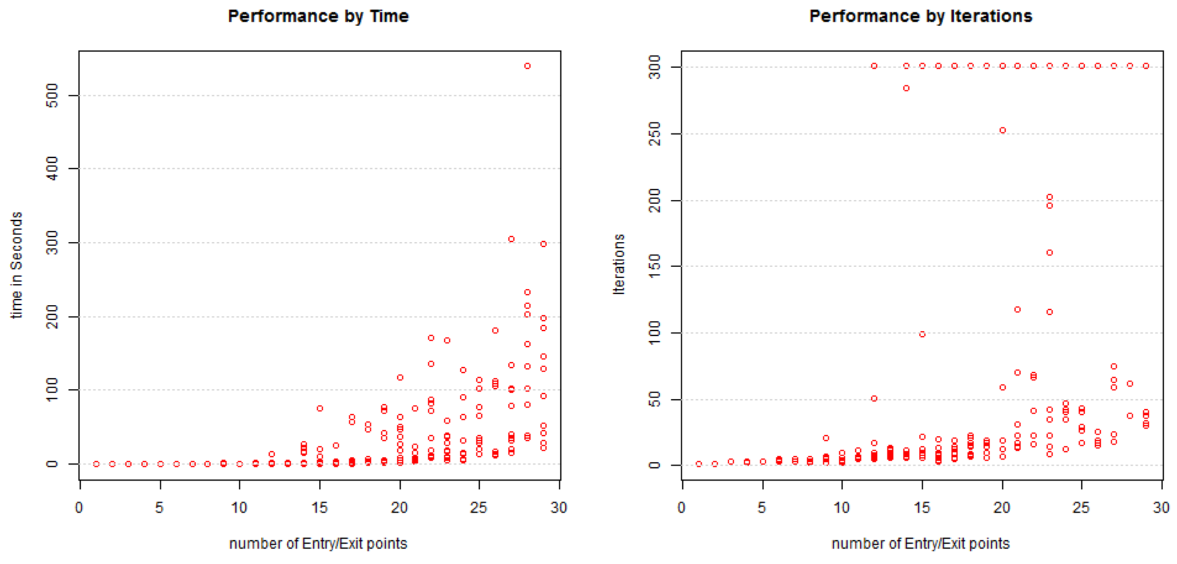


Figure 5.6: Group Behavior. Companion paths are, for this example, colored the same way as the original ones. Different levels of priority are separated and shown.Notice how the addition of each new level doesn't modify the shape of the previous ones. (a) Only the main paths are drawn. (b) Priority 2 paths are shown. (c) Priority 3 paths are shown. These paths are the ones that have the bigger modifications, since they take full responsibility for avoiding collision between other levels of priority.



(a) **(b)**

Figure 5.7: Performance. Each dot in the graph represents a different patch. 10 cases were simulated for each different number of pair of Entry/Exit points (a) Time is directly correlated to the number of iterations. (b) The number of iterations increases exponentially as the number of initial boundary control points augment. Some of the experiments still fail to converge.

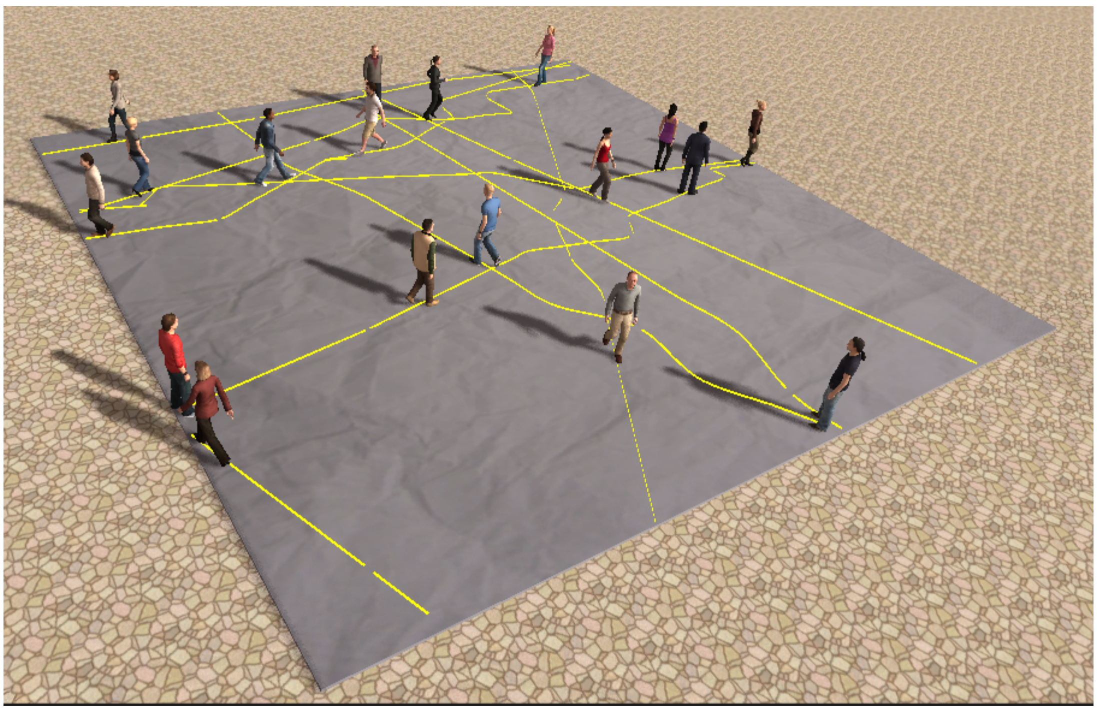


Figure 5.8: Patches for Virtual Humans. An example with actual humans walking through the patch generated.

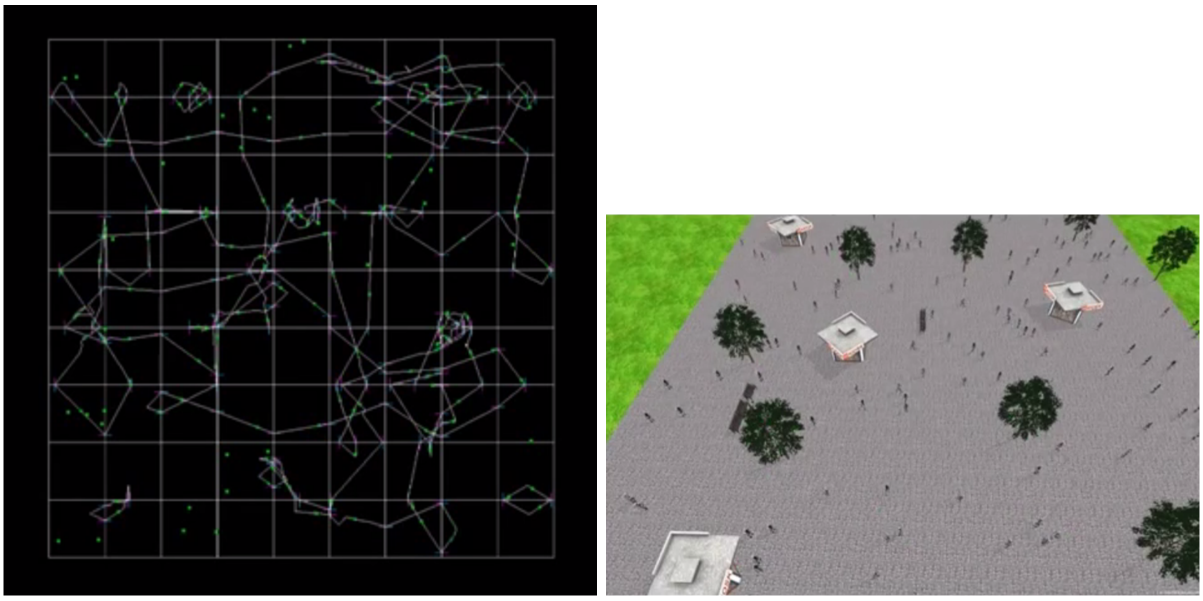

(a) (b)

Figure 5.9: Multiple Patches. An example of the technique for generating multiple patches. (a) Output of the algorithm. This took just a couple of seconds to be created. Each patch has only a few trajectories. (b) Final result. No boundary lines are drawn between patches, so the animation looks continuous.

5.2 Comparison with other Approaches

Particle-Based Approach

In Yersin et al. [Yersin et al. 2009] they used a particle based approach for their collision avoidance algorithm.

1. Agents track an attraction point moving along their initial trajectory $\tau(\mathrm{t})$.
2. Objects are repulsed by all other objects in their vicinity.
3. In order to avoid dead-lock situations, attractive and repulsive forces are balanced; the farther an object is from its attraction point and the closer the time t is to the initial time t_0, the more paramount attraction forces are, as compared to repulsive ones.

One of the problems with algorithm is that anticipatory behavior can't be simulated. With this technique, this behavior can be shown (Figure 5.10).

Reciprocal Velocity Obstacles Approach

Reciprocal Velocity Obstacles [van den Berg et al. 2007] is a technique for solving collisions between trajectories. In each time step, velocities for all agents are calculated. Simply put, velocities for each agent are chosen from a set of velocities that won't make the objects collide in the future. The main problem with this approach was getting the agents to the endpoints at the proper time. If the patch is dense, then some points will teleport to the final goal when their reach time gets closer.

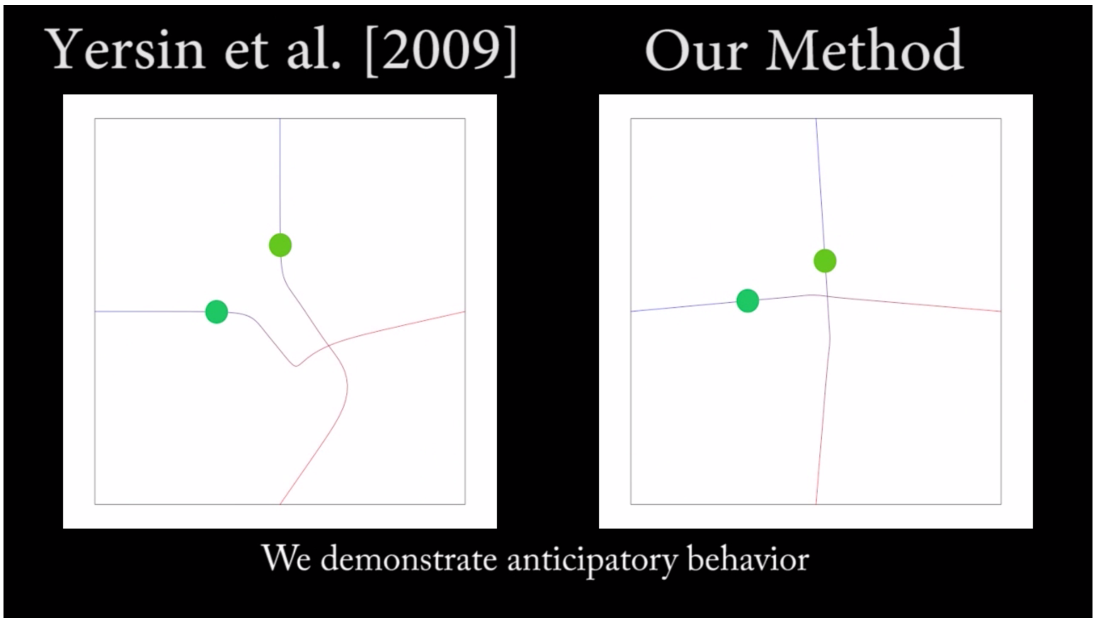


Figure 5.10: A comparison between the two methods is shown. With the particle based approach, the agent modifies its path just as it is about to collide. With this approach, the particles starts avoiding sooner the obstacle.

6
Future Work and Conclusions

This final chapter will present a summary of the work presented, as well as the limitations of this approach and how future work may handle them.

Scenarios The properties of the periodicity of crowd patches make them very adaptable to most scenarios, but not all of them. Cases where the camera, or the view point, remains static for a time longer than the period, makes obvious that the animation is repeating over time, so this static camera scenarios should be avoided. Filling, or emptying places scenarios can't be represented with this approach. Each person entering a patch, must always leave it a some time in the period of the animation.

Matching: The matching algorithm makes pairs of entry and exit points according to preferred speed and being in different patterns. Still, this doesn't take into consideration crossing between trajectories. In Figure 6.1 we can see two different cases of the matching of two pairs. One of them with a crossing between the paths. An initial matching that avoids this type of crossings can make the collision avoidance function work faster.

Since the crossings depend on the other pairs of points, this priority can't be added to the Stable Matching Algorithm. An initial guess to how this can be worked around would be, from crossing paths, exchanging their end points. This way, the crossing between those paths is eliminated. This may end up creating more crossings with other paths, but mathematically, using the fact that the projected length of the paths always decreases, it can be demonstrated that repeating this step can lead to a free crossing configuration. The problem is, most of the paths may end up with endpoints in the same pattern. The way to balance crossings and paths with ending points in different patterns is still an open question.

Convergence: Finding a collision free configuration for the paths depends heavily on the

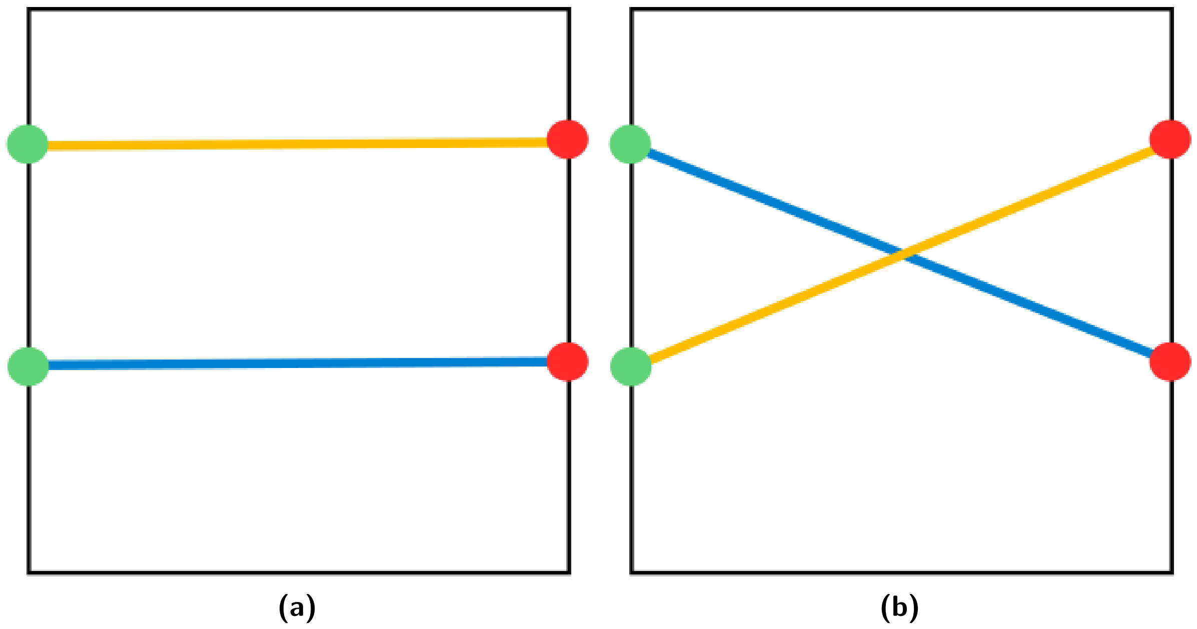


Figure 6.1: Crossings (a) No crossing between the trajectories. (b) A crossing between the trajectories. Choosing an initial matching that avoids crossing makes easier the work for the collision avoidance algorithm.

density of the patch. Sometimes, the number of entry and exit points may seem small, but if the period is too short, or the path to small, the algorithm may end up not finding a nice solution, or needing a bigger number of iterations than the maximum stated. How to select the maximum number of iterations depending on each particular patch and case is also an open question.

Obstacles: Obstacles can be handled just in a small number. Obstacle grouping depends on a certain distance threshold that decides if an obstacle must belong to the same group as another obstacle. This threshold is hard to decide, since obstacles having multiple trajectories passing in between them need a bigger threshold and need to be grouped (Figure 6.2). Using the same threshold for all the obstacles may lead to looping scenarios where trajectories start doing oscillations. Trying to decide the threshold depending on the number of trajectories passing in the middle of obstacles is equally hard, since the trajectories are prone to move. An even better way to approach obstacles is left for future work.

Temporal Modification of Control Points: In this thesis, mostly spatial corrections are done to the trajectories. For moving a control point in time, there is the restriction that it can't have a time bigger than the next control point or smaller than the previous one. For this reason, when moving points in time, more than one point may end up moving. Of course, this leads to some complications and speed issues. Nevertheless, there are some cases when spatial moving may not lead to the best scenario possible. In real life, people wait for other people to pass in front of them, instead of trying to go around (like it is the case in Figure 5.10). The problem of how to integrate direct temporal movements to the control points is still being investigated.

Global Calculation of Forces: When calculating repulsion forces acting over the control points, only the two points entering in a collision are taken into account. It may be better if

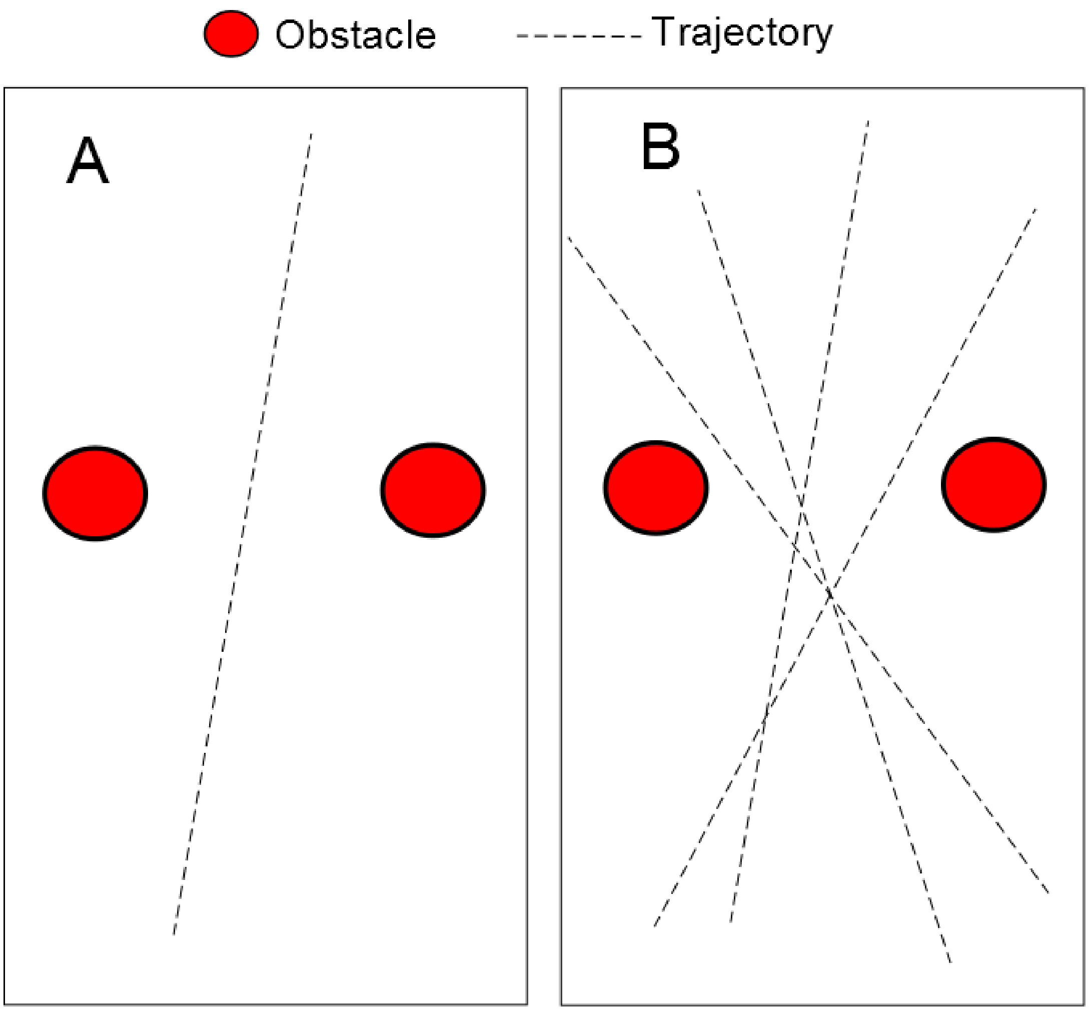


Figure 6.2: Obstacle Threshold Obstacles in cases A and B are at the same distance. In case A, they should not be considered as group, since there is enough space for the trajectory passing between them. In case B, the obstacles should be group together, since the number of trajectories passing in the middle is too many and the space is not enough.

other trajectories and obstacles near of them could also influence these forces. This way, moving points into new collisions would be harder. The main problem is how to do this, since calculating the virtual positions at which the other trajectories are at the moment of collision increments heavily the computation time.

Evaluating the Quality of Motion: One of the most important questions in crowd simulation research is: How good is the simulation? The evaluation of the quality of the generated crowd patches in this work is currently done by visual inspection, which depends heavily on the personal opinion of the observant. This is, of course, a very biased approach. Several methods have been proposed to evaluate the quality of crowd motion in a quantitative approach that will be investigated as future work.

For example Singh et al. [Singh et al. 2009] propose to evaluate quality based on the capability of a crowd simulation algorithm of accomplishing simple scenarios. Lerner et al.

[Lerner et al. 2010] and Guy et al. [Guy et al. 2012] take alternative approaches where the quality of a crowd simulator is compared to real-world data.

6.1 Conclusions

This thesis presents a novel optimization based technique for creating trajectories in crowd patches. The input of this technique is an empty patch and a set of boundary control points (or, if not given, the desired number of them). The output of the algorithm is the set of smoothed, collision free trajectories representing the paths the agents in the animation will follow.

Trajectories are first created from the pairing of the boundary control points, which is done using the Stable Matching Algorithm for optimal pairings. Then initial trajectories are modified in pairs, by displacing and creating control points; the displacement is affected not only by the colliding trajectory (or obstacle), but also by the tension force acting on the segment of the trajectory. A quick modification in the algorithm is given to simulate walking in groups behavior, and finally, an easy initial approach for smoothing trajectories is shown.

Patches can then be combined to form bigger animations, allowing an efficient representation of crowds. None of the calculations are performed at run-time, the creation of trajectories, including the collision handling and smoothing process are done pre-processing. Therefore, the final animation run-time is not affected, not even for dense case scenarios.

Some limitations still exist. The matching process doesn't take into account crosses between trajectories, incrementing the risk of collisions. Obstacle handling can be done with only a small number of obstacles, before increasing the risk of looping. Unrealistic speed is partially solved by tension forces, since they propagate the changes done in one control point to the rest of the trajectory, but how to balance the number of times the tension forces propagates is still an open problem. Propagating too fast leads to new collisions, whereas propagating too slow may lead to imperceptible changes. While handling dense scenarios is not the main purpose of this approach, they are still a limitation. The smoothing function is still a basic approach, and more complicated, human motion related approaches can be investigated in the future. Finally, evaluating the quality of the motion generated is one of the most important issues being investigated.

References

Boatright, C. D., Kapadia, M., Shapira, J. M., and Badler, N. I. 2014. Generating a multiplicity of policies for agent steering in crowd simulation. *Computer Animation and Virtual Worlds*.

Charalambous, P. and Chrysanthou, Y. 2014. The PAG Crowd: A Graph Based Approach for Efficient Data-Driven Crowd Simulation. *Computer Graphics Forum*.

Cook, R. L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Graph. 5,* 1 (Jan.), 51–72.

Gale, D. and Shapley, L. S. 1962. College admissions and the stability of marriage. *American Mathematical Monthly*, 9–15.

Gusfield, D. and Irving, R. W. 1989. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, Cambridge, MA, USA.

Guy, S. J., Chhugani, J., Kim, C., Satish, N., Lin, M., Manocha, D., and Dubey, P. 2009. Clearpath: Highly parallel collision avoidance for multi-agent simulation. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation.* SCA '09. ACM, 177–187.

Guy, S. J., van den Berg, J., Liu, W., Lau, R., Lin, M. C., and Manocha, D. 2012. A statistical similarity measure for aggregate crowd dynamics. *ACM Trans. Graph. 31,* 6 (Nov.), 190:1–190:11.

Helbing, D., Buzna, L., Johansson, A., and Werner, T. 2005. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science 39,* 1 (February), 1–24.

Jordao, K., Pettré, J., Christie, M., and Cani, M.-P. 2014. Crowd sculpting: A space-time sculpting method for populating virtual environments. In *Computer Graphics Forum.* Vol. 33. Wiley Online Library, 351–360.

Ju, E., Choi, M. G., Park, M., Lee, J., Lee, K. H., and Takahasi, S. 2010. Morphable crowds. In *Proc. of ACM SIGGRAPH Asia.* SIGGRAPH Asia '10. ACM, 140:1–140:10.

Kim, M., Hwang, Y., Hyun, K., and Lee, J. 2012. Tiling motion patches. In *Proceedings of the 11th ACM SIGGRAPH/Eurographics conference on Computer Animation.* Eurographics Association, 117–126.

Lerner, A., Chrysanthou, Y., and Lischinski, D. 2007. Crowds by example. *Computer Graphics Forum 26,* 3 (September), 655–664.

Lerner, A., Chrysanthou, Y., Shamir, A., and Cohen-Or, D. 2010. Context-dependent crowd evaluation. *Computer Graphics Forum 29,* 7, 2197–2206.

Narain, R., Golas, A., Curtis, S., and Lin, M. C. 2009. Aggregate dynamics for dense crowd simulation. In *Proc. of ACM SIGGRAPH Asia*. SIGGRAPH Asia '09. ACM, 122:1 – 122:8.

Nocedal, J. and Wright, S. 1999. *Numerical Optimization*. Springer Series in Operations Research. Springer.

Paris, S., Pettré, J., and Donikian, S. 2007. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Computer Graphics Forum 26,* 3 (September), 665–674.

Pettré, J., Ciechomski, P., Mam, J., Yersin, B., Laumond, J.-P., and Thalmann, D. 2006. Real-time navigating crowds: Scalable simulation and rendering. *Computer Animation adn Virtual Worlds 17,* 3–4, 445–455.

Ramirez, J. G. R., Lange, D., Charalambous, P., Esteves, C., and Pettré, J. 2014. Optimization-based computation of locomotion trajectories for crowd patches. In *Proceedings of the Seventh International Conference on Motion in Games*. MIG '14. ACM, New York, NY, USA, 7–16.

Reynolds, C. W. 1987. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics 21,* 4, 24–34.

Reynolds, C. W. 1999. Steering behaviors for autonomous characters. In *Game Developers Conference*. 763–782.

Rockstar-Games. 2013. Grand theft auto v. http://www.rockstargames.com/grandtheftauto/.

Singh, S., Kapadia, M., Faloutsos, P., and Reinman, G. 2009. Steerbench: a benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds 20,* 5-6, 533–548.

Thalmann, D. and Raupp Muse, S. 2013. *Crowd Simulation*, 2nd ed. Springer.

Treuille, A., Cooper, S., and Popović, Z. 2006. Continuum crowds. In *Proc. of ACM SIGGRAPH 2006*. SIGGRAPH '06. ACM, 1160–1168.

van den Berg, J., Lin, M., and Manocha, D. 2007. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*. ICRA '07. IEEE, 1928–1935.

Whittle, M. W. 2003. *Gait analysis: an introduction*. Butterworth-Heinemann.

Yersin, B., Maïm, J., Pettré, J., and Thalmann, D. 2009. Crowd patches: Populating large-scale virtual environments for real-time applications. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. ACM, New York, NY, USA, 207–214.