Optimization-based computation of locomotion trajectories for Crowd Patches

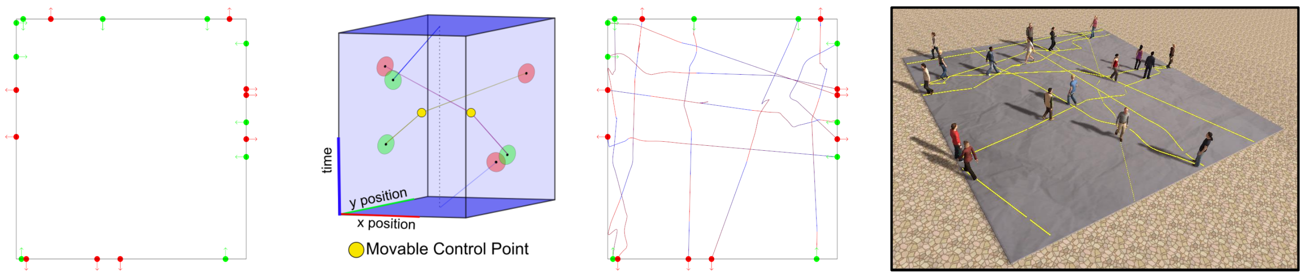


Figure 1: *A patch is a time-periodic clip of crowd motion. Given a set of spatio-temporal input and output control points, an optimization based approach is used to generate crowd motion that can be used to generate high quality crowd patches.*

Abstract

Over the past few years, simulating crowds in virtual environments has become an important tool to give life to virtual scenes; be it movies, games, training applications, etc. An important part of crowd simulation is the way that people (or any other living entities such as animals) walk from one place to another. This paper concentrates in improving the *crowd patches* approach proposed by Yersin et al. [Yersin et al. 2009] that aims in animating ambient crowds in a scene. This method is based on the construction of animation blocks (called patches) concatenated together under some constraints to create larger and richer animations with limited cost. An optimization based approach to generating smooth collision free trajectories for crowd patches is proposed in this work. The contributions of this work to crowd patches are threefold; firstly a method to match the end points of trajectories based on the Gale-Shapley algorithm [Gale and Shapley 1962] is proposed that takes into account preferred velocities and space coverage, secondly a better method for collision avoidance is proposed that gives natural appearance to trajectories and finally a cubic spline approach is used to smooth out generated trajectories. We demonstrate several examples of patches and how they were improved by our method, some limitations and directions for future improvements.

CR Categories:

Keywords: crowd simulation; crowd patches; virtual humans; optimization

1 Introduction

Video Games are constantly displaying larger and livelier virtual environments due to increased computational power and advanced rendering techniques. For example, the recent Grand Theft Auto (GTA) game [Rockstar-Games 2004] takes place in Los Santos and its surroundings, a completely virtual city. In spite of the impressive quality and liveliness of the scene, Los Santos still remains relatively sparsely populated with virtual people. The reason for this phenomenon is the large computational cost required to get an *ambient crowd* in such large environments. To address this issue, the technique of *Crowd Patches* has been recently introduced by Yersin et al. [Yersin et al. 2009].

Crowd patches are precomputed elements (patches) of crowd animations. Patches are time-periodic so that they can be endlessly played in time. The boundary conditions of precomputed animations are accurately controlled to enable combining patches in space (i.e., characters can move from in-between patches) and compose large ambient crowds. This technique eases the process of designing performance efficient ambient crowds.

One problem with this technique however, is the computation of internal animation trajectories for patches that satisfy both, time-periodicity and boundary conditions amongst patches. Satisfying both of these constraints is difficult, since it is equivalent to computing collision-free trajectories that exactly pass through spatio-temporal waypoints (i.e., at some exact position in time) whilst at the same time solving possibly complex interactions between agents (collision-avoidance). In addition to that, trajectories should look as natural as possible.

In this paper we propose a new optimization-based method to compute these internal trajectories. Our method starts by initially assigning linear space-time trajectories which are easy to compute and satisfy both, periodicity and boundary conditions, but at the same time might introduce collisions between characters. Then, iteratively, we optimize the trajectories to handle collisions. We try to keep the generated trajectories as close as possible to the initial linear trajectories, to minimize the magnitude of collision-avoidance maneuvers. How do we define this? Do we measure it? Could we present this as the principle of **least effort**; i.e. the agents do as little as possible to achieve their goals without doing any complex movement... Yes, I think we can present it as a least effort approach. Even though we cannot guarantee the optimal solution, it tends towards it as we are moving the closest points in the trajectories, aren't we?

To conclude, the main contribution of this work is an optimization-based algorithm to compute high quality animation trajectories ($2D$ global navigation trajectories) for individual crowd patches under constraints (expressed as a set of spatio-temporal boundary control points).

The remainder of this paper is organized as follows: Section 2 proposes a short overview on the state of the art. Section 3 details our technique to compute these internal trajectories. Then, in Section 4 some results, together with their performance and quality analysis are shown before a brief discussion and concluding remarks (Sections 5 and 6 respectively).

2 State of the Art

Most often, virtual environments are populated based on crowd simulation approaches [Thalmann and Raupp Muse 2013]. An ambient crowd is generated from a large set of moving characters, mainly walking ones. Recent efforts in crowd simulation have enabled dealing with improving computational performance [Pettré et al. 2006; Treuille et al. 2006], dealing with high densities [Narain et al. 2009] or controllable crowds [Guy et al. 2009]. There has also been a lot of effort to develop velocity-based approaches [Paris et al. 2007; van den Berg et al. 2007] which display much more smooth and realistic locomotion trajectories, especially thanks to anticipatory adaptation to avoid collisions between characters. Nevertheless, ... Is this an idea that we want to continue or a forgotten word?

Most of the related work we present here is agent based, even though flow based approaches are relevant also.

Simulation-based techniques seem ideal for creating an ambient crowd for large environments but several problems are recurrent with such approaches: a) crowd simulation is computationally demanding, crowd size is severely limited for interactive applications on light computers; b) simulation is based on simplistic behaviours (e.g., walking, avoiding collisions, etc.) and therefore it is difficult to generate diverse and rich crowds based on classical approaches; c) 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 key idea of this approaches is to indirectly define the crowd rules from existing crowd data (such as real people trajectories) [Lerner et al. 2007; Ju et al. 2010; Charalambous and Chrysanthou 2014]. Locally, trajectories are typically of good quality, because they reproduce real recorded ones. However, such approaches raise other difficulties: it is difficult to guarantee that the example database will cover all the required content and it can also be difficult to control behaviors and interactions displayed by characters if the database content is not carefully selected. Finally, those approaches are most of the times computationally demanding; even more so than traditional simulation based techniques.

To solve both performance as well as quality issues, crowd patches were introduced by Yersin et al. [Yersin et al. 2009]. The key idea is to generate an ambient moving crowd from a set of interconnected patches. Each patch is a kind of $3D$ animated texture element, which records 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 controlled to be able to connect together two different patches with characters moving from one patch to another. Thus, a crowd animated from a set of patches have a seamless motion and patches' limits cannot be easily detected. The boundary conditions are all registered into *patterns*, which are sort of gates for patches with a set of spacetime input/output points. For a more detailed expanation on the crowd patches approach please refer to [Yersin et al. 2009].

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

Yersin et al. suggest using an adapted Social Forces technique to compute internal trajectories [Helbing et al. 2005]. The key idea is to connect input/output points together with linear trajectories and model characters as particles attracted by a goal moving along one of these linear trajectories, combined with repulsion forces to avoid collision between them and static obstacles. One problem with this approach is limited density level, as well as the level of quality of trajectories that suffer from the usual drawbacks of Helbing's generated trajectories, i.e., lack of anticipation, which results into non natural local avoidance maneuvers (see Figure XXXXXXXX).

Figure 10?

Compared to previous techniques we suggest formulating the problem of computing internal trajectories as an optimization problem. First, we suggest optimizing the way input and output points are connected. Especially, since waypoints are defined in space and time, we connect them trying to have some *comfortable* walking speed (i.e., close to the average human walking speed). Indeed, characters moving too slow or too fast are visually evident artifacts. Secondly, after having connected waypoints with linear trajectories, we deform them to remove any collisions by employing an iterative approach. This approach aims at minimizing as much as possible the changes to the initial trajectories. We show improvements in the quality of results as compared to the original work by Yersin et al (see Figure XXXXXXXX - same as previous paragraph, could keep only one of the two references, this one).

3 Methodology

It is a bit weird to have Figure 3 before Figure 2 in the paper isn't it? Maybe we can put Figure 2 after, as they are referenced in the paper? Like this ?

In this section we present the proposed methodology for generating trajectories for patches. We first give some definitions and notations (Section 3.1), followed by an overview of the method (Section 3.2), and the description of our method to generate initial trajectories, handle collisions and smoothing out the resulting trajectories (Sections 3.3–3.5).

3.1 Definitions

Before presenting our approach on trajectory generation, we present some definitions and notations regarding Crowd Patches. Please refer to Figure 2 for a visual representation of the definitions and to [Yersin et al. 2009] for a more detailed description on these concepts.

A **patch** is a set $\{A, \pi, D, S\}$ where A is geometrical area with a convex polygonal shape, π the period of time of the animation and D and S are the sets of dynamic and static objects, respectively. These last two sets may be empty in case of an empty patch.

Static objects are simple obstacles whose geometry is fully contained inside the patch.

Dynamic objects are animated; i.e., they are moving in time according to a set of trajectories T.

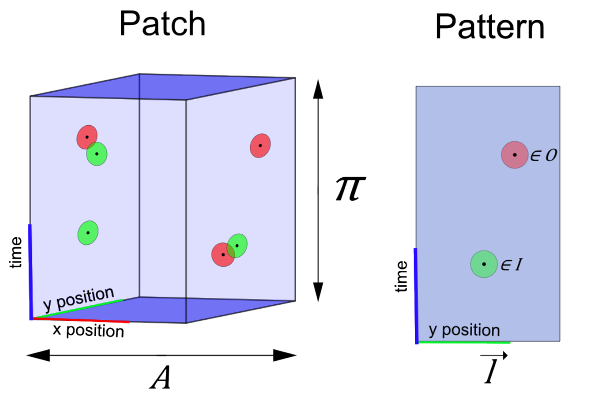


Figure 2: ***Patches and Patterns*** *A* patch *is defined by the geometrical area A where a set of dynamic and static objects (D and S respectively) can move over a period of time π.* Patterns *define boundary conditions for the patches and act as portals connecting neighboring patches.*

3.1.1 Trajectories

We define a **trajectory** inside a patch as a function $\tau(t)$ going from time to position, more specifically from a subset of $[0, \pi]$ to A:

$$\tau : [t_1, t_2] \to A, \quad 0 \leq t_1 < t_2 \leq \pi \tag{1}$$

We represent a trajectory as a list of control points connected by segments:

- A **control point** is a point in space and time $\mathbf{cp} = \{\mathbf{p}_{cp}, t_{cp}\}$. All control points in a trajectory can either be boundary or movable ones. Boundary control points serve as entry and exit points to the patch and cannot be moved, added or deleted. Movable control points on the other hand can be moved, added, or removed from the trajectory as long as they do not violate the constraints of the patch; i.e., their positions must lie inside area A ($\mathbf{p}_{cp} \in \mathbf{A}$) and their time t_{cp} must be between t_1 and t_2.
- A **segment** is a straight line connecting two control points in a specific order. Since these are unidirectional lines in space-time, it is important to remember that they are not allowed to go backwards in time.

There are two categories of dynamic objects: endogenous and exogenous agents. **Endogenous agents** remain inside A for the total period of time π. In order to achieve periodicity for the animation, they are associated with a trajectory $\tau : [0, \pi] \to A$, such that it respects the periodicity condition: the position at the start and at the end of the animation must be the same, i.e. $\tau(0) = \tau(\pi)$.

Exogenous agents on the other hand go outside A. They enter the patch at time $t_{initial}$ and position $\mathbf{p}_{initial}$, and they exit at time t_{final} and position $\mathbf{p}_{final}$. For each agent we associate a sequence of $n \geq 1$ trajectories $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Sequences may have only one trajectory, but some agents require additional trajectories in order to satisfy speed and time constraints. The following conditions must be respected in each sequence of trajectories associated with an exogenous agent:

1. $\mathbf{p}_{initial}$ and $\mathbf{p}_{final}$ must be points on the borders of A otherwise they cannot be exogenous agents.
2. If the sequence is composed by more than one trajectory, for each two contiguous trajectories, the following must be true to ensure continuity: $\tau_i(\pi) = \tau_{i+1}(0)$.

Note that the second condition implicitly implies that in sequences with multiple trajectories, each middle trajectory must be fully defined in the period of time $[0, \pi]$, while τ_1 must be defined in $[t_{initial}, \pi]$ and τ_n must be defined in $[0, t_{final}]$.

I am a bit confused with the parameter π. Is this the whole-animation time or is it just the parameter for each section of a trajectory (and also the complete trajectory)? I think the notation is clearer like in the next section with $\pi^{(i)}$ for the ending parameter of a subsection and just π for the ending parameter of the overall trajectory. What do you think?

3.1.2 Patterns

A patch can be considered as a spatio-temporal right prism depending on the type of polygon used as its area (cube in the case of a squared area patch). A **pattern** can be defined as one lateral face of the prism (Figure 2). Specifically, it is a rectangle whose base is one of the edges of the polygonal area (we define $\mathbf{l} \in \mathbb{R}^2$ as this two dimensional vector), and its height is equal to the period π. In addition to these, patterns also include the sets $\mathbf{I}$ and $\mathbf{O}$ of Input and Output boundary control points respectively. The input set contains the boundary control points where exogenous agents begin their trajectories; we call these *Entry Points*. Conversely the elements of output are called *Exit Points*; they establish the position in time and space that the exogenous agents finish their paths. Formally defined, a pattern $\mathbf{P}^{(i)}$ is:

$$\mathbf{P}^{(i)} = \{\mathbf{l}^{(i)}, \pi^{(i)}, \mathbf{I}^{(i)}, \mathbf{O}^{(i)}\} \tag{2}$$

To populate virtual environments, patches are concatenated together. Thus, continuity between trajectories should be enforced for exogenous agents passing through two contiguous patches. This means that two adjacent patches must have a similar pattern on the side they share; i.e., the vector l and period π must be the same and the input and output sets must be exchanged. More formally, having two patterns $\mathbf{P}_1$ and $\mathbf{P}_2$ where $P^{(1)} = \{\mathbf{l}^{(1)}, \pi^{(1)}, \mathbf{I}^{(1)}, \mathbf{O}^{(1)}\}$ and $P^{(2)} = \{\mathbf{l}^{(2)}, \pi^{(2)}, \mathbf{I}^{(2)}, \mathbf{O}^{(2)}\}$, then, in order to satisfy C^0 continuity the following must apply:

$$l^{(1)} = l^{(2)}, \pi^{(1)} = \pi^{(2)}, \mathbf{I}^{(1)} = \mathbf{O}^{(2)}, \mathbf{I}^{(2)} = \mathbf{O}^{(1)} \tag{3}$$

Under these conditions, P_1 is the mirror pattern of P_2 and vice versa. When these patches are animated, agents will be going from one patch to adjacent ones. If the area $\mathbf{A}$ of a patch is a square then 4 patterns are defined, one per side. Patterns defined by a patch have the property that the sum of the cardinality of all the inputs is the same as the sum of the cardinality of all outputs; we call this the parity condition: $\sum(|inputs|) = \sum(|outputs|)$.

A patch defines a set of patterns, and conversely, a set of patterns that satisfies the parity condition (i.e. all patterns in the set have same period and whose vectors define a convex polygonal area) can be used to create a patch indirectly.

3.2 Overview

The objective of the proposed work is to generate a patch given a set of patterns. This implies that given a set of constraints such as input and output spatio-temporal control points, a set of believable collision-free trajectories that interconnect all of them should be generated. This process has three main steps (Figure 3):

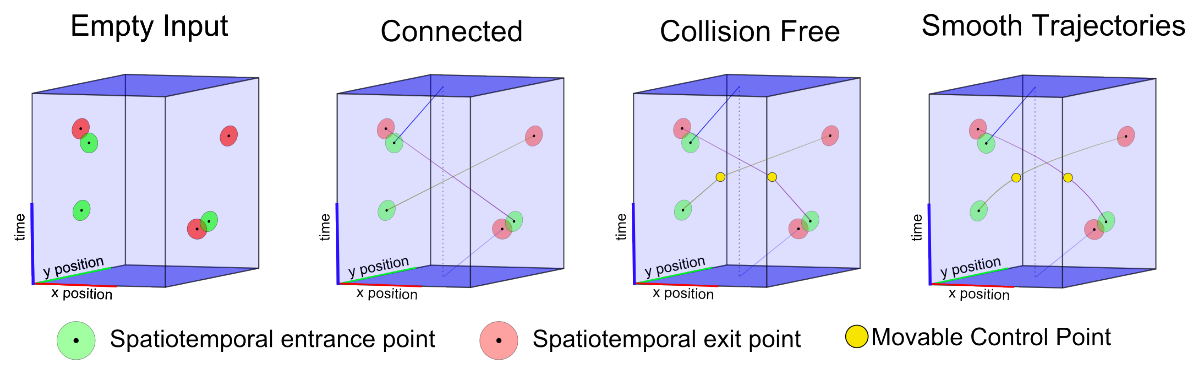


Figure 3: ***Overview.*** *Input and output points in a patch's patterns are initially connected and subsequently modified using the proposed optimization approach and smoothed out to be collision free.*

1. Match the elements in the Input and Output sets contained within a patch. – In this step, entry and exit points are connected based on a score function that tries to keep agents close to their preferred speed while at the same time avoiding connections to similar patterns, thus reducing unwanted u-turns. The input for this step is a set of patterns and the output is a set of piecewise linear trajectories connecting the entry and exit points.

2. Create collision free trajectories for these pairings.– Starting from simple line trajectories; bend them by iterative subdivisions until they are collision free. Points lying at the borders, i.e. entry and exit points, are hard constraints and can never be moved.

3. Smooth out trajectories (if needed).– Splines are used as a final step to minimize the hard turns ensuring that the smoothed-out trajectories stay as close as possible to the original ones from Step 2 to avoid introducing new collisions.

A more detailed look on all three steps is given over the remainder of this section.

3.3 Connecting Boundary Control Points

The first step to the proposed algorithm is to match all the entry and exit points in an optimal way. To do this, a measure of the match's quality has to be defined. Intuitively, there are some matches that are better than others; e.g., judging by observation, trajectories passing near the center of the patch look better than the ones staying close to the borders forcing in essence the characters to cover more space. Some other aspects can also be considered, such as how close the speed needed by an agent to travel from an entry to an exit point is compared to typical walking comfort speeds of humans. A comfort speed of $u_{cft} = 1.33\ m/s$, which is the normal walking speed of humans in an unconstrained environment is used in this work [Whittle 2003].

For a square patch, we define an order of preference between matching; first we prefer to match points between opposing patterns, followed by neighboring ones and finally with points on the pattern itself. For any of these cases, if there exist multiple possible matching options on the same pattern, the point whose associated trajectory is closest to the defined comfort speed is selected.

To solve this matching problem, we employ the *Gale-Shapley algorithm* [Gale and Shapley 1962] (see Algorithm 1), commonly referred to as the algorithm to solve the *stable marriage problem*. This algorithm assures that at the end, if we have Alice engaged to Bob and Carol engaged to Dave, it is not possible for Alice to prefer Dave and Dave to prefer Alice – this is called a *stable match*.

```
Initialize all $m \in M$ and $w \in W$ to *free* ;
**while** $\exists$ *free man $m$ who still has a woman $w$ to propose to* **do**
    $w \leftarrow m's$ highest ranked woman to whom he has not yet proposed ;
    **if** *$w$ is free* **then**
        $(m, w)$ become engaged ;
    **else**
        some pair $(m', w)$ already exists ;
        **if** *$w$ prefers $m$ to $m'$* **then**
            $(m, w)$ become engaged ;
            $m'$ becomes *free* ;
        **else**
            $(m', w)$ remain engaged;
        **end**
    **end**
**end**
```

Algorithm 1: Gale-Shapley Stable Marriage Algorithm from [Gusfield and Irving 1989].

Algorithm 1 demonstrates the Gale-Shapley algorithm in relation to two equal lists of men and women who are being matched for marriage. However the algorithm generalizes to any matchable objects, which in our case are entry and exit points.

In order to apply Algorithm 1, preference values for all combinations of entry and exit points should be defined. To do so, each entry point keeps a *proposal list* $\mathbf{L}_s$ indicating the order of preference for its matching (Table 1). The following approach is employed to rank each possible matching:

1. Find the speed it would take to travel from an entry point to all exit points. 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]

2. Next, each pair of points is assigned a preference value:

$$pr_{score} = u_{match} + p \tag{4}$$

where $u_{match} = arctan(|u_{cft} - u|) \in [0, \pi/2)$ indicates closeness to desired speed with 0 indicating maximum close-

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

ness. $p = \{0, 2, 4\}$ defines a *penalty* value that depends on where the two points lie relative to each other; for points on opposing patterns there is no penalty, for neighboring patches it is 2 and for points on the same pattern it is 4 (this can be generalized to any prism-like patch).

3. Sort $\mathbf{L}_s$ in ascending order; the first entry indicates the most desired exit point.

It should be emphasized here, that each entry point keeps its own proposal list. After each entry point has been assigned a proposal list, Algorithm 1 is used to define matches between the entry and exit points; every two points that remained engaged at the end of the algorithm become a pair.

The final step is creating the initial batch of trajectories. Firstly, the paired points are connected via straight lines; if a line tries to connect two points backwards in time (that is if $t_2 < t_1$), the initial trajectory is split into two parts – from t_1 to π and from 0 to t_2 as in [Yersin et al. 2009]. The positions of these new control points are in the same straight line, taken in such a way that the speed is the same in both segments. The same approach is used if the trajectory enforces unrealistically high speed values ($u \gg u_{cft}$).

Further adjustments to the initial trajectories are done for some special cases. For agents traveling only over an edge, a control point is added near the center of the patch. For agents moving slowly, a control point with the same position but on a different time is added, resulting in agents that stop suddenly (as if pausing to look around) but later on continuing their journey at a better speed.

This step results in linear trajectories that are optimized for speed and coverage of space using an objective function (Equation 4). These trajectories though can be colliding with each other, since no special care has been taken up to this point to handle that. To address this issue, the iterative technique, described in the next paragraphs has been proposed.

3.4 Removing Collisions

The set of linear trajectories generated by Algorithm 1 will most likely have collisions with the environment or with other trajectories. As collisions rarely take place in real-life human crowds, a strategy to remove them from the initial trajectories should be defined. For this, we propose an algorithm that manipulates the linear trajectories by moving control points. Since patches are concatenated together to create larger crowds, care should be taken during trajectory modification so that the spatio-temporal boundary control points (i.e., entry and exit ones) are not modified; other control points can be added and manipulated.

Algorithm for collision handling An iterative algorithm for handling the collisions is proposed (Algorithm 2). The main idea is the following: given a matrix M that stores the current minimum distances in-between all trajectories, the algorithm iterates modifying the trajectories (and therefore its closest values M) until $min(M) > \theta$, where θ represents a minimum allowed distance value. Given typical circular agents of radius r, $\theta = 2r$ (therefore iff $min(M) > \theta \Leftrightarrow$ patch is collision free). To do so, new control points are added at each iteration (i.e., trajectories are split into segments) that are moved under some constraints until the trajectories are collision free.

```
Compute minimum distance matrix $M$;
while there exists at least one entry in $M$ below the threshold do
    Find indices $i$ and $j$ for which $M(i,j)$ has the smallest value $d$;
    Create temporary control points $\mathbf{cp}_i$ and $\mathbf{cp}_j$ in $\tau_i$ and $\tau_j$ that are at distance $d$ ;
    Apply repulsion forces to $\mathbf{cp}_i$ and $\mathbf{cp}_j$;
    Update $\tau_i$ and $\tau_j$;
    Update $M$ ;
end
```
Algorithm 2: The control points generation algorithm

First, the minimum distance matrix is calculated (Section 3.4.1). As long as collisions exist, trajectories τ_i and τ_j having the minimum value are found – that minimum value corresponds to a moment in time and two points: $\mathbf{p}_i$ and $\mathbf{p}_j$. These two points are moved to handle the collision using correction forces $\mathbf{F}_i$ an $\mathbf{F}_j$:

$$\mathbf{F}_i = R(\phi) * \Delta\hat{\mathbf{p}}_{i,j} * \theta * w_i \tag{5}$$

$\Delta\hat{\mathbf{p}}_{i,j}$ is the normalized distance vector between the two points ($\Delta\mathbf{p}_{i,j} = \mathbf{p}_i - \mathbf{p}_j$)), $\theta = r_j + r_j$ 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 ($\phi : -0.5 \leq \phi \leq 0.5\ rad$), an finally w_i and w_j are weights to reduce speed artifacts and prevent agents from leaving the bounds of the patch:

$$w_i = \begin{cases} u_j/(u_i + u_j) & \text{if point stays in patch} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

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

Having the correction force, point $\mathbf{p}_i$ is moved according to the following equation:

$$\mathbf{p}_i^{new} = \mathbf{p}_i + \mathbf{F}_i \tag{7}$$

Once $\mathbf{p}_i^{new}$ is found, a check to find if there is an existing control point withing a small time interval is performed; if successful, then the existing control point is moved to $\mathbf{p}_i^{new}$, otherwise a new control point is added at the same position. Finally, column i and row j of the distance matrix M are updated with new distances. Calculations for force $\mathbf{F}_j$ and point $\mathbf{p}_j$ are symmetric.

We have found that in most situations this algorithm converges quickly and produces collision free trajectories (Section 4). However there are still situations where it converges slowly or even gets stuck in an infinite loop; for these a maximum number of iterations is set.

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

Table 1: ***Proposal List*** *Each entry point keeps a list of preference scores for all possible exit points. Lower values indicate bigger 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 higher score.*

Entry Point: 1	
Exit Point	Preference
B	**0.34**
C	1.3
A	2.3
E	2.4
D	4.5
F	4.6

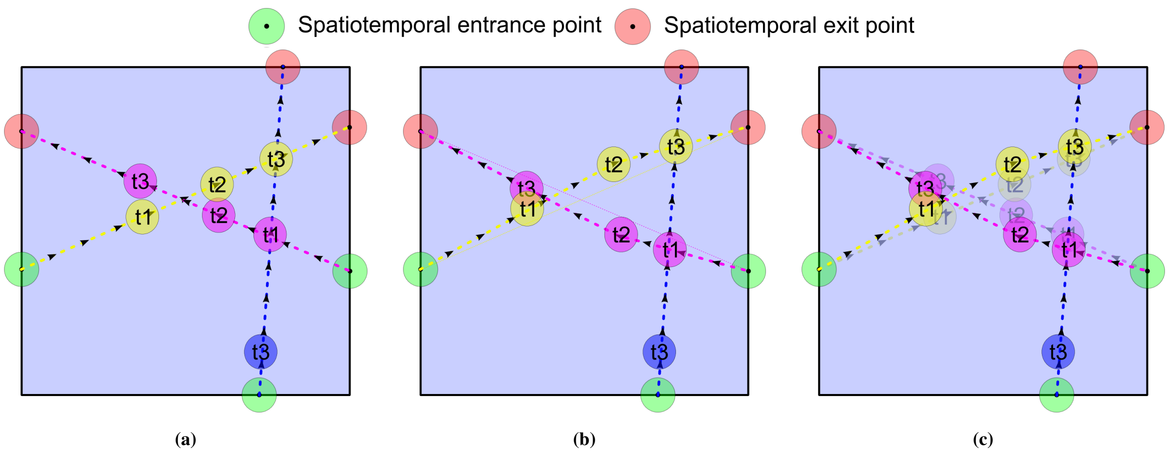


Figure 4: ***Collision Handling*** *(a) The "worst" collision is currently identified to be the one between the yellow and pink trajectories at timestep t_2 (b) The two trajectories are deformed to handle the collision (c) Overlayed difference between the two trajectories.*

3.4.1 Distance Matrix Calculation

Distance Matrix The first step of the collision handling algorithm is generating a distance matrix $M \in \mathbb{R}^{nxn}$ between all the n trajectories; i.e., the value at $M(i, j)$ represents the minimum distance between trajectories τ_i and τ_j (Table 2).

The following properties apply for all $i, j \in [1, n]$ and can be employed to reduce computation time:

- $M(i, i) = 0$
- $M(i, j) = M(j, i)$, i.e., the matrix is symmetric
- $M(i, j) = \infty$, $\forall(\tau_i, \tau_j)$ that are never present at the same time

Minimum Distance The minimum distance between two trajectories is defined as their minimum spatio-temporal distance; i.e. the point in time where they are closer to each other. Recall that a trajectory can consist of one or more segments separated by control points (Section 3.1) and therefore the minimum has to be found in-between all the trajectory's segments.

Given any two linear trajectory segments $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$, their minimum distance can be found with an analytic approach. First, their common time interval is found; i.e., the period of time that the trajectories coexist in the patch. If the two segments do not have any common time interval, then their distance is set to infinity (practically a large value). If there exists a common interval $[t_s, t_e]$, then we set $\mathbf{p}_s^{(1)}$ and $\mathbf{p}_s^{(2)}$ as the two segments position at time t_s. Additionally, agents moving on these two segments have a velocity of $\mathbf{v}^{(1)}$ and $\mathbf{v}^{(2)}$ respectively. So, for any point in time $t : 0 \leq t \leq t_e - t_s$, their distance is:

$$d(t) = ||(\mathbf{p}_s^{(1)} + \mathbf{v}^{(1)} * t) - (\mathbf{p}_s^{(2)} + \mathbf{v}^{(2)} * t)|| \tag{8}$$

Setting $\mathbf{w} = \mathbf{p}_s^{(1)} - \mathbf{p}_s^{(2)}$ and $\mathbf{\Delta v} = \mathbf{v}^{(1)} - \mathbf{v}^{(2)}$ Equation 8 becomes:

$$d(t) = ||\mathbf{w} + \mathbf{\Delta v} * t|| \tag{9}$$

Solving for t after setting the derivative $d'(t) = 0$ we get the time of minimum distance t_c:

Table 2: ***Distance Matrix.*** *Minimum distances between 4 trajectories ($\tau_0 - \tau_3$) are updated whilst modifying the trajectories. At this iteration of the algorithm, the minimum distance was between trajectories τ_1 and τ_2 and therefore they will be modified for the next step.*

	τ_0	τ_1	τ_2	τ_3
τ_0	0	∞	8.31	2.10
τ_1	∞	0	**0.14**	7.60
τ_2	8.31	**0.14**	0	∞
τ_3	2.10	7.60	∞	0

$$t_c = (-\mathbf{w} \cdot \mathbf{dv})/||\mathbf{dv}||^2 \tag{10}$$

If $0 \leq t_c \leq t_e - t_s$, then setting $t = t_c$ in Equation 8 the minimum distance between segments $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$ is found. If t_c is outside the bounds of the segment we check the endpoints of the line segment for collision. Having the minimum distances between all the segments of two trajectories, finding the minimum distance is trivial (see Table 2 for an example).

3.5 Smoothing

Handling collisions by control points manipulation results in trajectories with sharp changes in direction. Cubic spline interpolation is proposed to smooth out the generated trajectories without introducing any new collisions; i.e., the smoothed out trajectory is as close as possible to the original sharp one (Figure 5). Cubic splines interconnect pairs of points in a trajectory with a polynomial function of the 3^{rd} degree:

$$S(x) = a + bx + cx^2 + dx^3 \tag{11}$$

For each one of the trajectory's segments, coefficients a, b, c and d need to be found under C^1 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., $S(t_i) = \mathbf{p}_i$ and $S(t_{t+1}) = \mathbf{p}_{i+1}$.

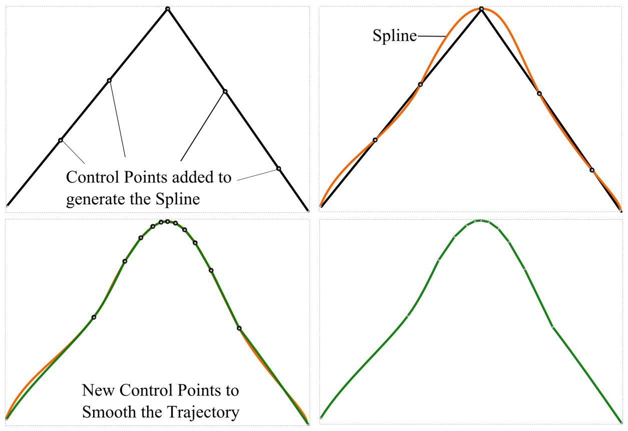

Spline
Control Points added to generate the Spline
New Control Points to Smooth the Trajectory

Figure 5: ***Smoothing Process*** *(top row) Linear segments are sampled to generate new control points that are use to fit cubic splines. (bottom row) These splines are then sampled depending on their curvature to generate a new set of linear segments.*

2. The speed at the control point $\mathbf{p_j}$ that connects two consecutive splines S_{j-1} and S_j must be equal; i.e.,

$$\frac{\partial}{\partial t} S_{j-1}(t_j) = \frac{\partial}{\partial t} S_j(t_j) \tag{12}$$

These restrictions can be accommodated in such way that they form a system of linear equations that can be solved using Cholesky decomposition. This method cannot directly be applied to the the current control points, since typically these points are few and the resulting splines are very different to the originally estimated linear trajectories and therefore new collisions are introduced. To handle this, the trajectory is uniformly sampled to generate *tacks*, i.e., new *virtual control points*. These tacks enforce the splines to be flatter and closer to the original linear trajectories (Figure 5).

Splines are now calculated based on the tacks. Instead of storing the splines, these are again sampled based on curvature to get new control points that define a new set of *linear* segments; the higher the curvature the finer the sampling (bottom left of Figure 5).

Below needs cleaning up

There may be cases, even with a healthy number of tacks,

What is *"healthy"* in this context?

that the splines still vary too much from the original trajectory. We define a threshold based on the same threshold of collision to know how far a way a point can be moved. For these cases where the new control point surpasses the threshold, we simply don't add it to the set of new control points. There may be extreme cases (for example, due to too a bad sampling of the tacks at the beginning), where the spline has extreme curves that are very different from our initial trajectory. In those extreme cases most of the new control points would not be added and the new smooth trajectory would end up being very similar to the original one.

Having splines for trajectories is better than having simple straight lines, but we know humans don't follow either of them when walking, so other methods can be tried to improve this initial approach.

Is there a reference to support the above statement?

4 Results

The proposed trajectory generation algorithm was integrated into our own crowd patches platform in C++. Each of the experiments described in the following paragraphs were for patches of size $A = 16m * 16m$ and period $\pi = 10sec$.

Performance All the performance measurements presented in this section were run on a single thread of an Intel Xeon quad core 2.8 GHz processor having 8GB of RAM (Figure 6. Each experiment consisted of placing equal numbers of entry/exit points in random positions around the border of the patches. For each possible number of entry/exit points (ranging from $1 - 50$) the experiment was run multiple times (20) resulting in 1000 different patches. For each one of the experiments, time and number of iterations required for convergence were measured. The results indicate a direct correlation of the number of iterations to the time required for convergence. More importantly, increasing the number of control points decreases performance exponentially due to increased density.

Additionally, some of the experiments having more than a total of 60 control points (0.6% of the experiments) had very slow convergence and were forced to stop at 2000 iterations resulting in some collisions in the patch. Examining the patches that failed to converge we observed that these was mainly due to the placement of the initial control points; if control points are clustered on a subset of the patch's sides matching between them is more difficult and non-optimal. To account for this (assuming that the whole crowd patch generation is completely automatic) a better control point generation algorithm must be implemented.

Example visualization The proposed method manages to create trajectories that are both free of collisions and with smooth motion; i.e., agents following these trajectories have speeds near regular humans comfort speeds and are also visually pleasing. Figure 7 shows the trajectories generated by the proposed method for a patch consisting of 10 input and 10 output points and the intermediate steps; first input and output points are matched and connected with linear trajectories using Algorithm 1, then any collisions are handled using Algorithm 2 and finally the generated trajectories are smoothed out.

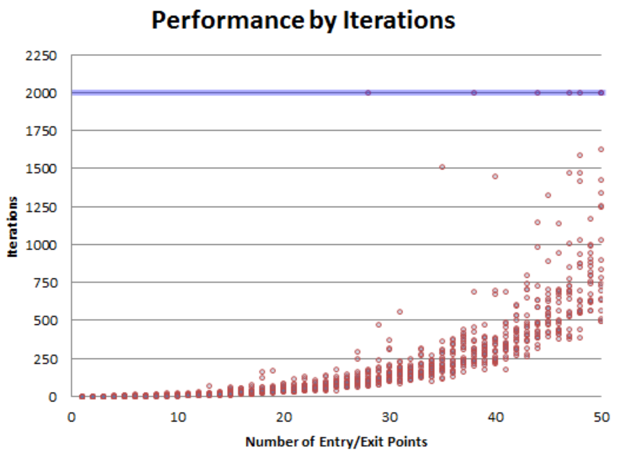
Adaptability to control point placement This method is able to adjust to the placement of the control points (Figure 8) aiming at the same time for good space coverage over the period of the patch.

Adaptability to density The proposed method can adapt to different numbers of initial control points; i.e., it manages to generate smooth collision free trajectories for increasing larger numbers of trajectories (Figure 9). Increasing the number of control points increases density and the time required for calculating these trajectories but recall that crowd patches are *precomputed* and therefore during simulation time huge numbers of moving agents can be simulated at limited cost. Again, trajectories aim to cover the patch's space.

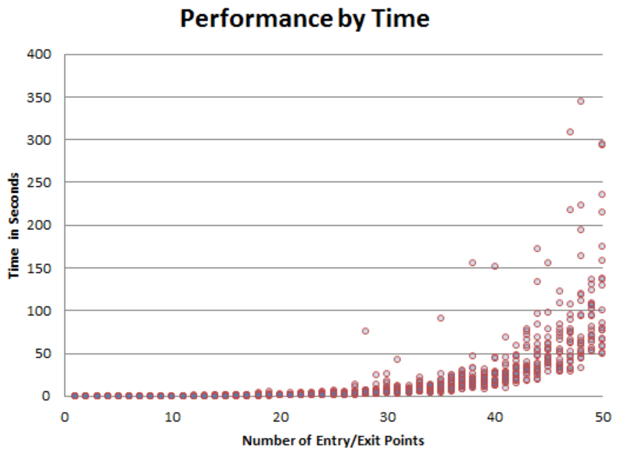
Collision Anticipation In the original work [Yersin et al. 2009] Helbing's social forces [Helbing et al. 2005] approach was used to handle collisions. One problem with that approach is the lack of anticipation by agents; agents handle collisions late resulting in unnatural looking trajectories even for simple scenarios such as the one in Figure 10 whereas the proposed approach emulates better anticipatory behaviour.

5 Discussion

Convergence The convergence of this algorithm, depends mostly on the how many agents well want in the scene. As we can see, the

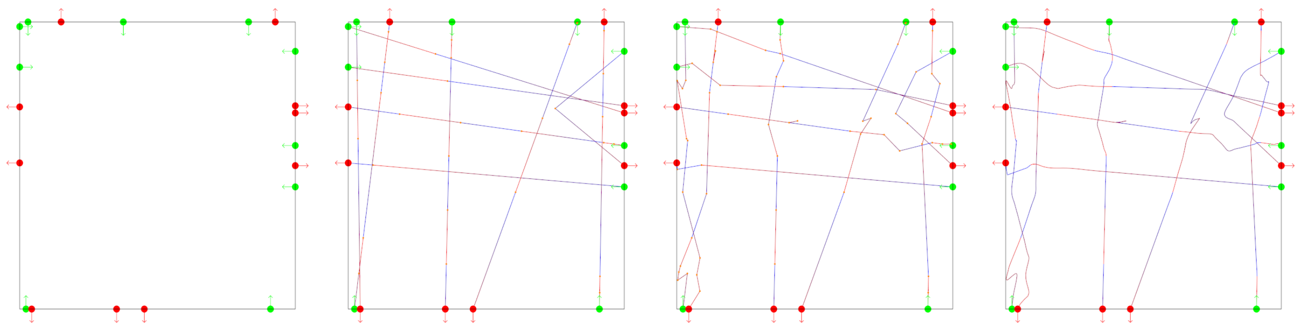


(a)



(b)

Figure 6: ***Performance*** *A dot on each graph represents a simulated patch. (a) The number of iterations for the proposed algorithm to converge is exponential to the number of initial control points – some of the experiments failed to converge. (b) Time is directly correlated to the number of iterations.*



(a) *Input* **(b)** *Connected* **(c)** *Collision Free* **(d)** *Smoothed Out*

Figure 7: ***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.*

bigger that number is, the slower it takes to create a patch and the most dangerous it becomes not reaching a solution.

When we have two boundary control points very near each other in space-time coordinates, the harder it will be for the trajectories to converge, since the only way to find a collision free trajectory would be moving those fixed boundary control points. For that, a better technique may be implemented, so when we just want a random initial batch of boundary control points, they are generated in such a way there is enough space between one another.

Spatial moving of waypoints only In this paper, we use mostly spatial corrections to the trajectories of the agents. For some scenarios, moving the point in space coordinates may not be the best way to go, it could provoke the control point to try to move outside of the patch boundaries or make a huge change in the agents speed.

Some collisions may be avoided just by changing the time in one of the two movable control points of a segment, thus making the agent move faster or slower without modifying the spatial trajectory. This would solve the problem of going out of the boundaries, but we have to be careful on how we change the time if we dont want to produce an unrealistic change in velocity. We also have to be careful that trajectories can never go backwards in time.

Obstacles We mentioned earlier a patch may have static agents, called obstacles. With this technique, how can obstacles be added? For a small number of obstacles, we can consider obstacles just like any other agent, and incorporate them into the algorithm with the extra condition that they can never be moved. The only problem, is that obstacles must be well placed. When two obstacles are close enough that the form a tunnel for one initial trajectory, the movable control point will start looping going from one obstacle to another, unable to find a collision free trajectory. A possible solution for this, may be grouping the obstacles that are close to one another and consider them as a bigger obstacle. This will quickly start occupying the area of the patch, so in general, the obstacles will need to be few.

Quality of motion We believe that by visual inspection the quality of motion in our approach is better than the method used by Yersin et al. [Yersin et al. 2009]. There are several methods that can be used to quantify quality of motion in crowds. These include (=====================). We plan to use one or more of these methods to assess the quality of our motion in the future.

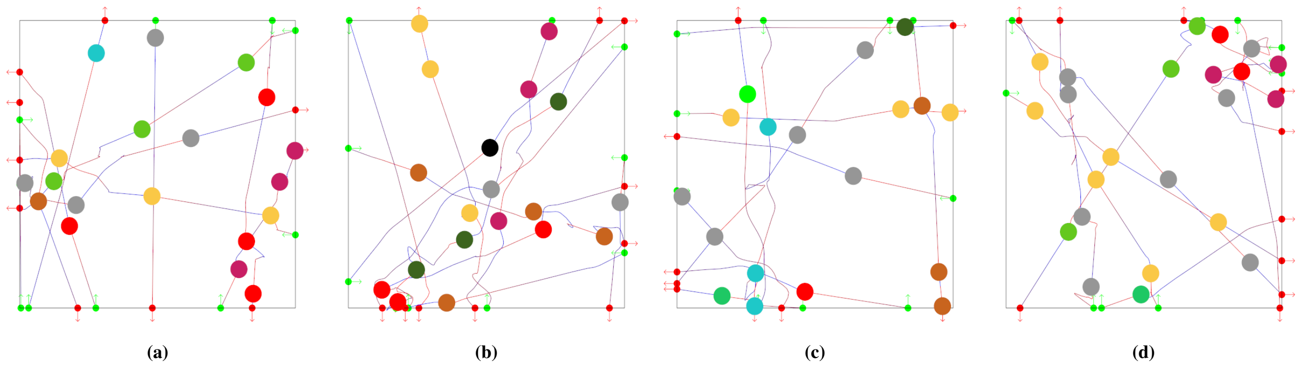

(a) **(b)** **(c)** **(d)**

Figure 8: ***Variance*** *We demonstrate different results for the same number of initial spatio-temporal control points but different placement. All the examples here consist of* 10 *entry and* 10 *exit control points.*

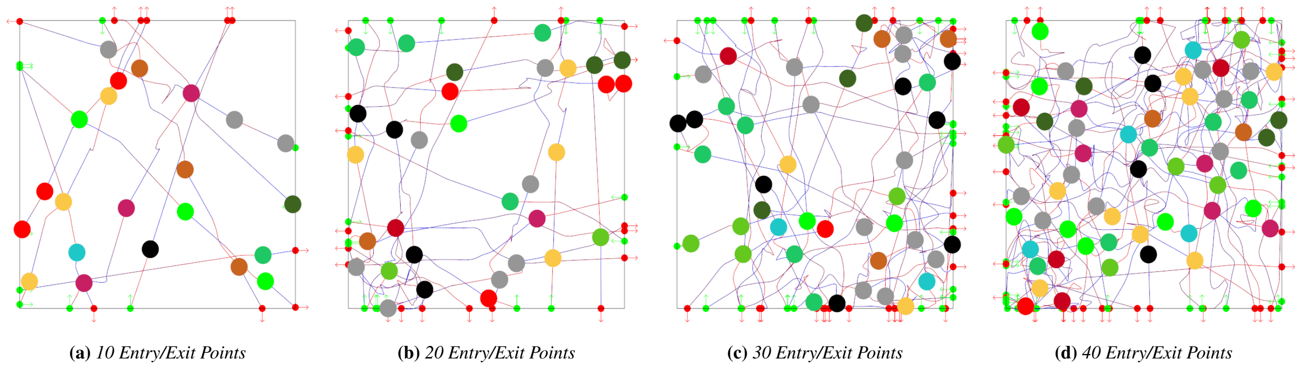

(a) *10 Entry/Exit Points* **(b)** *20 Entry/Exit Points* **(c)** *30 Entry/Exit Points* **(d)** *40 Entry/Exit Points*

Figure 9: ***Density Adaptability*** *The proposed method can generate collision free trajectories for many situations; from sparse to very dense.*

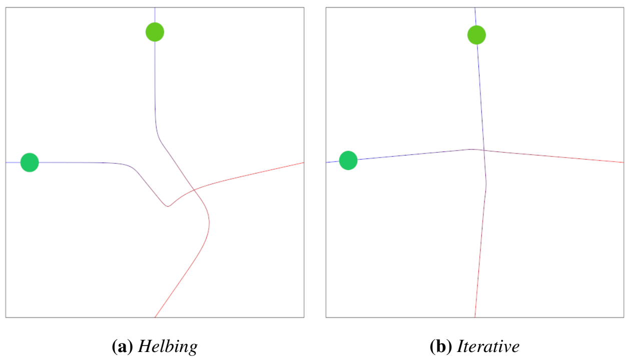

(a) *Helbing* **(b)** *Iterative*

Figure 10: ***Collision Anticipation*** *(a) Using the approach described in [Yersin et al. 2009] trajectories lack anticipation as they suddenly curve when collisions are near whereas (b) the proposed method generates trajectories with higher anticipation resulting in gradually changing direction.*

6 Conclusions

A novel optimization based technique for generating trajectories for crowd patches has been presented; given an empty patch and a set of spatio-temporal control points on the edges of the patch that define entry and exit points for characters, a set of smooth collision free trajectories is generated. Patches then can be combined to efficiently represent an ambient crowd since most of the calculations are done at pre-processing; no collision handling is performed at run-time. Therefore, even though generating patches for dense patches using the proposed technique is expensive, run-time performance is not affected.

Even though the algorithm produces collision free trajectories, there are some limitations. Some times trajectories are generated that enforce unrealistic speeds (much different than the comfort speed of humans) or abnormal looking behaviour can emerge such as sudden turns. Additionally, the proposed algorithm is a greedy one since at each step local optimization is performed (i.e., we look at the agent with higher collision score) and therefore in the end a globally optimized solution might not be achieved. As future work, we plan on expanding this method using a global optimization approach that takes into account various plans of action and takes into account speed. Data-driven from real-world crowds can also used during optimization so that the generated trajectories provide more real life like behaviours. Finally, collision points are moved in space; an approach that moves trajectories in space and time could potentially solve unrealistic looking behaviours, remove motion artifacts and converge faster.

Acknowledgements

Who do we love so much we want to worship them?

References

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

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*, ACM, SCA '09, 177–187.

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.

JU, E., CHOI, M. G., PARK, M., LEE, J., LEE, K. H., AND TAKAHASI, S. 2010. Morphable crowds. In *Proc. of ACM SIGGRAPH Asia*, ACM, SIGGRAPH Asia '10, 140:1–140:10.

LERNER, A., CHRYSANTHOU, Y., AND LISCHINSKI, D. 2007. Crowds by example. *Computer Graphics Forum 26*, 3 (September), 655–664.

NARAIN, R., GOLAS, A., CURTIS, S., AND LIN, M. C. 2009. Aggregate dynamics for dense crowd simulation. In *Proc. of ACM SIGGRAPH Asia*, ACM, SIGGRAPH Asia '09, 122:1 – 122:8.

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.

ROCKSTAR-GAMES, 2004. Grand theft auto: San andreas. http://www.rockstargames.com/grandtheftauto/, October.

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*, ACM, SIGGRAPH '06, 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*, IEEE, ICRA '07, 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 *Proc. of the 2009 Symp. on Interactive 3D Graphics and Games*, ACM, I3D '09, 207–214.