Optimization-based Computation of Locomotion Trajectories for Crowd Patches

Jose Guillermo Rangel Ramirez[1]* Devin Lange[2]† Panayiotis Charalambous[3]
Marc Christie[3] Claudia Esteves[1] Julien Pettré[3]

[1] Centro de Investigacion en Matematicas, A.C. (CIMAT), Mexico [2] University of Minnesota, USA [3] Inria-Rennes, France

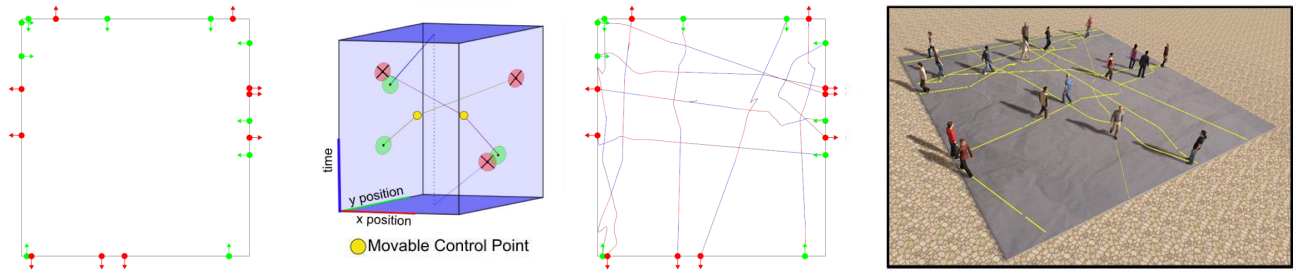


Figure 1: *A patch is a time-periodic clip of crowd motion. Given a set of spatio-temporal input and output control points, an optimization based approach is used to generate crowd motion that can be used to generate high quality crowd patches.*

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 move from one place to another. This paper concentrates on improving the *crowd patches* approach proposed by Yersin et al. [Yersin et al. 2009] that aims on efficiently 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 run-time cost. Specifically, an optimization based approach to generate smooth collision free trajectories for crowd patches is proposed. The contributions of this work to the crowd patches framework 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 an improved algorithm 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 the proposed method, some limitations and directions for future improvements.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

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

*e-mail:josguil@cimat.mx
†e-mail:ange604@umn.edu

1 Introduction

Video Games are constantly displaying larger and livelier virtual environments due to increased computational power and advanced behavior and rendering techniques. For example, the recent Grand Theft Auto (GTA) game [Rockstar-Games 2013] 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 simulate *ambient crowds* into such large environments. To address this issue, the *Crowd Patches* technique has been recently introduced by Yersin et al. [Yersin et al. 2009].

Crowd patches are precomputed elements (patches) of crowd animations that are time-periodic so that they can be endlessly played in time. To do so, the boundary conditions of precomputed animations are accurately controlled to enable combining patches in space and time so that characters can move 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 a new optimization-based method to compute internal trajectories for patches is proposed. This 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. Trajectories are then iteratively optimized to handle collisions. This optimization procedure aims in generating trajectories that are as close as possible to the initial trajectories minimizing the number of collision avoidance maneuvers as much as possible.

Concluding, the main contribution of this work is an optimization-based algorithm to compute high quality navigation trajectories for

individual crowd patches under constraints expressed as sets of spatio-temporal boundary control points .

The remainder of this paper is organized as follows: Section 2 presents a short overview on related work, Section 3 details the proposed technique for trajectory generation, in Section 4 some results are presented, together with their performance and quality analysis followed by 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 [Reynolds 1987; Reynolds 1999; 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 smoother and realistic locomotion trajectories, especially thanks to anticipatory adaptation to avoid collisions between characters.

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 behavior rules from existing crowd data (such as trajectories from real people) [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 time computationally demanding; even more so than traditional simulation based techniques. Some researchers, 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 at speeds comparable to simulation methods.

An alternative to solve both performance and quality issues, are methods that interconnect pre-computed patches of animations [Yersin et al. 2009; Kim et al. 2012; Jordao et al. 2014] to generate larger ambient crowd animations. Crowd patches [Yersin et al. 2009] more specifically are a kind 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. Thus, a crowd animated from a set of patches have seamless motion and patches' limits cannot easily be detected. The boundary conditions are all registered into *patterns*, which are sort of gates for patches with a set of spacetime input/output points. 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 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 waypoint. 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 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 the Social Forces approach such as lack of anticipation, which results into unnatural looking local avoidance maneuvers (Figure 7).

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 waypoints 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. 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 the changes to the initial trajectories. We demonstrate improvements in the quality of results as compared to the original work by Yersin et al. (Section 4).

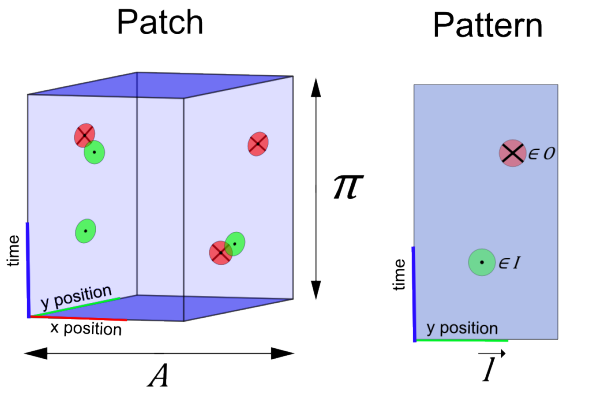


Figure 2: ***Patches and Patterns*** *A* patch *is defined by the geometrical area A where a set of dynamic and static objects (D and S respectively) can move over a period of time π.* Patterns *define boundary conditions for the patches and act as portals connecting neighboring patches.*

3 Methodology

In this section we present our methodology for generating spatio-temporally constrained trajectories for crowd patches. We start by giving some definitions and notations (Section 3.1), followed by an overview of the method (Section 3.2) and the three main steps of the algorithm; control point matching, collision handling and trajectory

smoothing (Sections 3.3–3.5).

3.1 Definitions

Some definitions and notations regarding Crowd Patches are presented required for the proper understanding of the proposed algorithms. Please refer to Figure 2 for a visual representation of the definitions and to [Yersin et al. 2009] for a more detailed description of the concepts.

A **patch** is a set $\{\mathbf{A}, \pi, \mathbf{D}, \mathbf{S}\}$ where $\mathbf{A} \subset \mathbb{R}^2$ is a geometrical area with a convex polygonal shape, π the period of time of the animation and $\mathbf{D}$ and $\mathbf{S}$ are the sets of dynamic and static objects, respectively. These last two sets may be empty in the case of an empty patch. **Static objects** are simple obstacles whose geometry is fully contained inside the patch, whereas **dynamic objects** are animated; i.e., they are moving in time according to a set of trajectories T.

3.1.1 Trajectories

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

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

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 $\mathbf{A}$ ($\mathbf{p}_{cp} \in \mathbf{A}$) 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 $\mathbf{A}$ for the total period π of the patch. In order to achieve periodicity for the animation, they are associated with a trajectory $\tau : [0, \pi] \rightarrow A$, 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 $\mathbf{A}$. 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 $\mathbf{A}$ otherwise they cannot be exogenous agents.
2. If the sequence is composed by more than one trajectories, the ending position of one trajectory should be the same as the starting position of the next one to ensure continuity[1]: $\tau_i(t_\pi) = \tau_{i+1}(t_0), \quad \forall i < n.$

[1] Please refer to [Yersin et al. 2009] for more details.

Note that the second condition 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 $[0, t_{final}]$.

3.1.2 Patterns

A patch can be considered as a spatio-temporal right prism depending on the type of polygon used for its area $\mathbf{A}$ (cube in the case of a square 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; called the *Entry Points*. Conversely the output control points 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 $\mathbf{l}$ 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 $\mathbf{P}^{(1)} = \{\mathbf{l}^{(1)}, \pi^{(1)}, \mathbf{I}^{(1)}, \mathbf{O}^{(1)}\}$ and $\mathbf{P}^{(2)} = \{\mathbf{l}^{(2)}, \pi^{(2)}, \mathbf{I}^{(2)}, \mathbf{O}^{(2)}\}$, then, in order to satisfy C^0 continuity the following must apply:

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

Under these conditions, $\mathbf{P}^{(1)}$ is the mirror pattern of $\mathbf{P}^{(2)}$ and vice versa. When these patches are animated, agents will be moving from one patch to its adjacent ones. If the area $\mathbf{A}$ of a patch is a square then four patterns are defined, one per side. Importantly, 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* of the patch:

$$\sum_{i \in \text{patch}} |I^{(i)}| = \sum_{i \in \text{patch}} |O^{(i)}|$$

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 patches given a set of patterns (one for each side of the patch). 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):

1. Match the elements in the Input and Output sets contained within a patch – 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

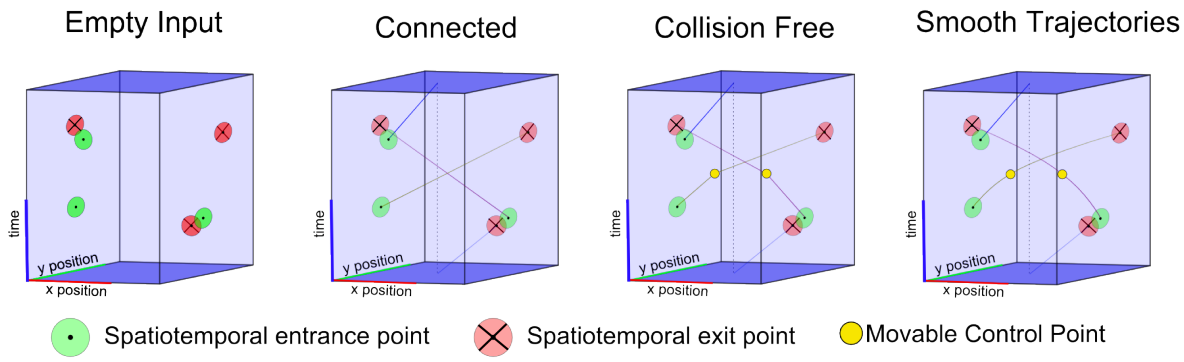


Figure 3: ***Overview*** *Input and output points in a patch's patterns are initially connected and subsequently modified using the proposed optimization approach and smoothed out to be collision free.*

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; successive bending is applied 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 follows in the remainder of this section.

3.3 Connecting Boundary Control Points

The first step to the proposed algorithm is matching all 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. 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, an order of preference between matching patterns is defined; matching points between opposing patterns are preferred, followed by neighboring ones and finally with points that lie on the same pattern. 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 u_{cft} is selected.

To solve this matching problem, the *Gale-Shapley algorithm* [Gale and Shapley 1962] (see Algorithm 1), commonly referred to as the algorithm to solve the *stable marriage problem* is employed. 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*. Algorithm 1 demonstrates the Gale-Shapley algorithm in relation to two equal lists of entry and exit points who are being matched for pairing.

In order to apply Algorithm 1, preference values for all pairs of

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

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

entry and exit points should be defined. To do so, all entry and exit points keep a *proposal list* $\mathbf{L}_s$ indicating the order of preference for their matching (Table 1). The following approach is employed to rank each possible match-up:

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 $\Delta t = \pi + t_2 - t_1$. [2]

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 closeness. $p = \{0, 2, 4\}$ 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.[3]

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

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

[3] This can be generalized to any prism-like patch

Table 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.*

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

It should be emphasized here, that all control points (both entry and exit) keep their own proposal lists. 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 remain 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.

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 obstacles or 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 *must* 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 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 $min(M) > \alpha$, where α represents a minimum allowed distance value. Given typical circular agents of radius r, $\alpha = 2r$ (therefore $min(M) > \alpha \Leftrightarrow$ patch is collision free). To do so, new control points are added during iterations (i.e., trajectories are split into segments) that are moved under some constraints until the trajectories are collision free (see Figure 4 for an example).

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 τ_i and τ_j that are at distance d ;
  Apply repulsion forces to $\mathbf{cp}_i$ and $\mathbf{cp}_j$;
  Update τ_i and τ_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 $\mathbf{F}_j$:

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

$\Delta\hat{\mathbf{p}}_{i,j}$ 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[4], $R(\phi)$ is a small random noise rotation matrix to help prevent infinite loops ($\phi : -0.5 \leq \phi \leq 0.5\ rad$), and finally w_i is a weight 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 $\mathbf{F}_i$, point $\mathbf{p}_i$ is displaced using 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 within a small time interval is performed; if successful, then that point is moved to $\mathbf{p}_i^{new}$, otherwise a new control point is added at $\mathbf{p}_i^{new}$. Finally, columns i, j and rows i, 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 has fast convergence rate and produces collision free trajectories (Section 4). However there are still situations where it converges slowly or even gets stuck in an infinite loop; and therefore a maximum number of iterations is set.

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:

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

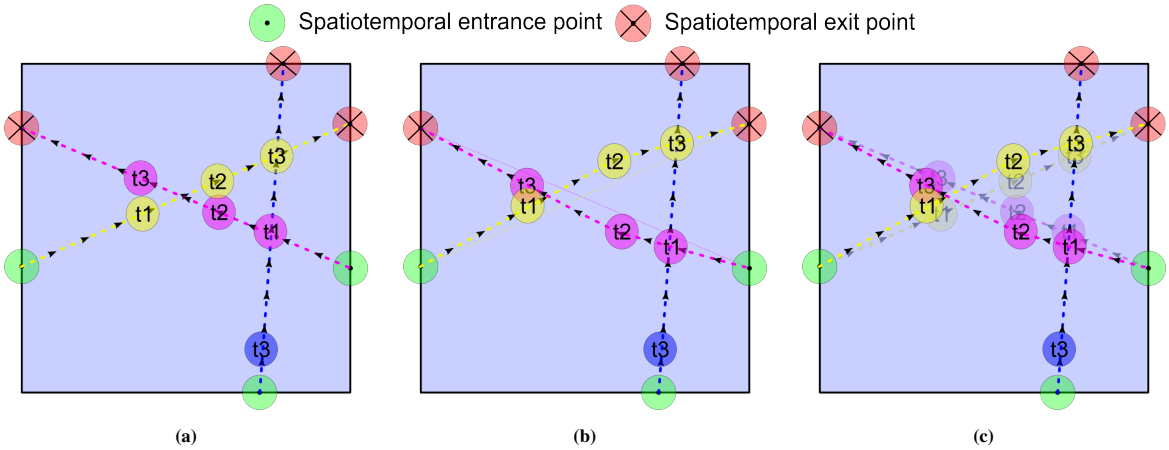


Figure 4: ***Collision Handling*** *(a) The "worst" collision is currently identified to be the one between the yellow and pink trajectories at timestep t_2 (b) The two trajectories are deformed to handle the collision (c) Overlayed difference between the two trajectories.*

- $M(i,i) = 0$
- $M(i,j) = M(j,i)$, 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. at the point in time where they are closest 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:

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

To find the minimum distance, we set the derivative $d'(t) = 0$ and solve for t to get the time of closest approach:

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

If $0 \leq t_c \leq t_e - t_s$, then by 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. By having the minimum distances between all the segments of the two trajectories, it is trivial to find the minimum distance (see Table 2 for an example).

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

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). For each one of the trajectory's segments, the spline's coefficients 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 $S(t_{i+1}) = \mathbf{p}_{i+1}$.
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.,

$$\frac{\partial}{\partial t} S_{j-1}(t_j) = \frac{\partial}{\partial t} S_j(t_j) \tag{11}$$

These restrictions can be accommodated in such a 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

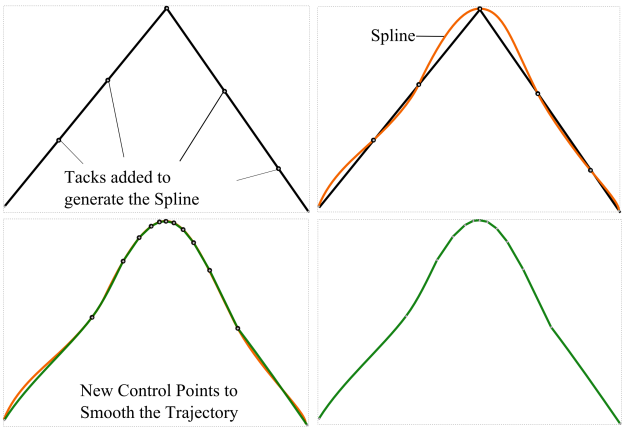
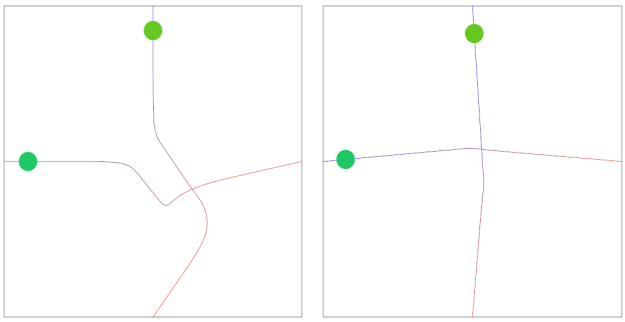


Figure 5: ***Smoothing Process*** *(top row) Linear segments are sampled to generate tacks that are used to create cubic splines. (bottom row) These splines are then sampled depending on their curvature to generate a new set of linear segments.*



(a) *[Yersin et al. 2009]* **(b)** *Iterative*

Figure 7: ***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.*

control points that define a new set of *linear* segments; the higher the curvature the finer the sampling (bottom left of Figure 5).

There may be cases that even using a large number of tacks the splines will differ significantly from the original ones. Threshold value α as was defined in Section 3.4 is used to restrict on the maximum allowed displacement of splines. If a new control point surpasses this threshold, it is simply discarded. There may be extreme cases (for example, due to bad sampling of the tacks at the beginning), where the spline has extreme curves that are very different from the initial trajectory. In these extreme cases, most of the new control points would then not be added and the smoothed trajectory would end up being very similar to the original one.

Splines are a relative cheap approach with nice smoothing of trajectories, but other techniques are currently being considered so that the generated trajectories are more close to real-life ones.

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 in time around the borders 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 the increased density.

Additionally, some of the experiments having more than a total of 30 pairs of entry-exit control points (0.6% of the experiments) had very slow convergence rates 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 this was mainly due to the placement of the initial boundary control points; if these points are placed too close together then it may be impossible to resolve collisions while staying within the patch and maintaining realistic speed. This can also be due to the fact that the proposed approach resolves local minima at each step using a greedy approach.

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; they handle collisions late resulting in unnatural looking trajectories even for simple scenarios such as the one in Figure 7 whereas the proposed approach emulates better anticipatory behaviour.

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 human comfort speeds and are also visually pleasing. Figure 8 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 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 9) 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 large numbers of trajectories (Figure 10). Increasing the number of control points increases density and the time required for calculating these trajectories but recall that crowd patches can be *precomputed* and therefore during simulation time large numbers of moving agents can be animated with small cost (compared to real-time simulations for example). Additionally, the proposed method aims in covering the entire patch's space over a period of time; we emphasize here that density control is difficult using other real-time simulation techniques.

5 Discussion

Convergence: The convergence of this algorithm, depends mostly on the number of agents required in the virtual scene; large numbers of agents lead to long convergence time as well as the possibility that the algorithm will not converge and the patch generation will fail. Convergence failure can occur if at least two boundary control

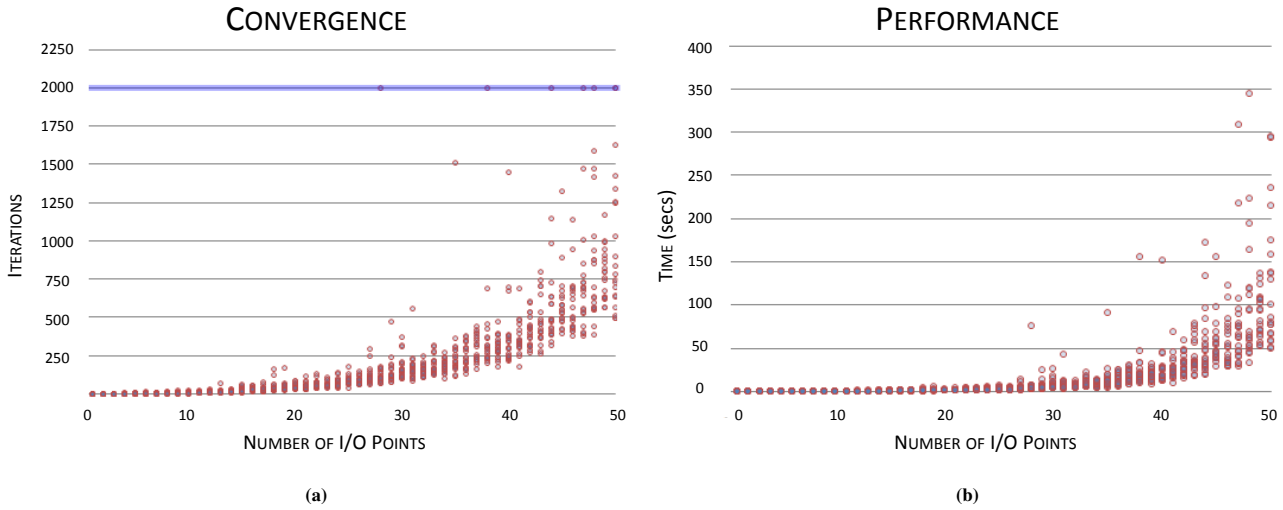
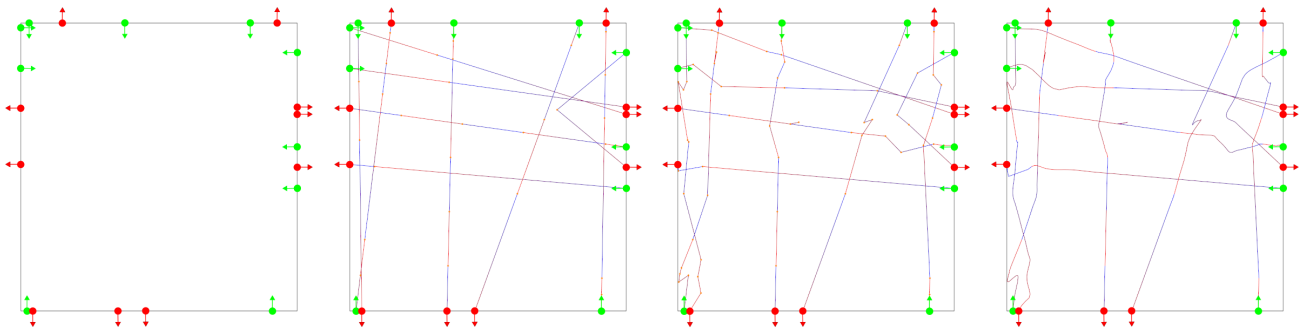


Figure 6: ***Performance*** *A dot on each graph represents a simulated patch. (a) The number of iterations for the proposed algorithm to converge is exponential to the number of initial control points – some of the experiments failed to converge. (b) Time is directly correlated to the number of iterations.*



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

Figure 8: ***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.*

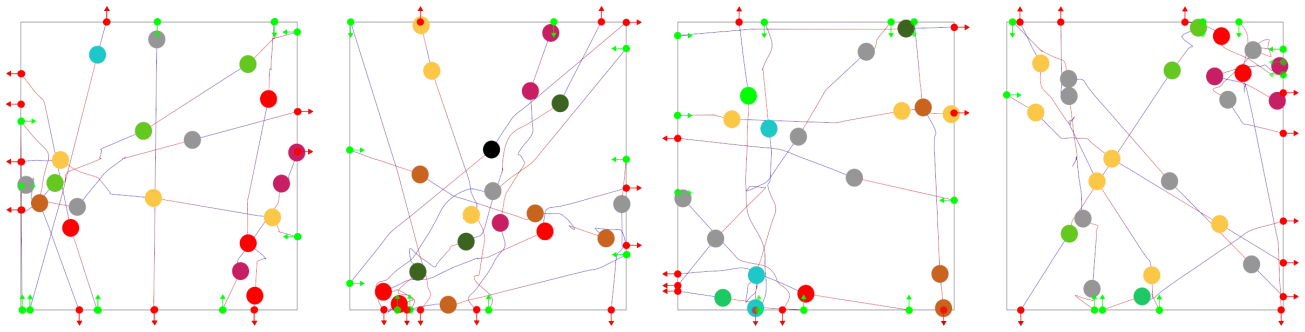
points are closer to the minimum allowed threshold α (Section 3.4) at the same exact moment in time; this is by definition a convergence failure since *boundary control points cannot be moved*. If on the other hand, the boundary control points differ slightly in time, the algorithm will converge but with a high probability of generating agents with unrealistic speeds.

Furthermore, two control points near the corners may have a repulsion force that pushes one of them outside the patch's boundaries. For these cases, two approaches can be taken; either ignore these two points and move on or let one of the two points leave the boundaries of the patch. In the current implementation the latter was employed due to its simplicity.

This leads to one important conclusion; given bad input (i.e., patterns and their boundary control points) the algorithm will behave badly resulting into bad trajectories. Therefore, we are currently looking into approaches of generating patches that will guarantee well behaved trajectories (defining well behaved trajectories is also an interesting problem).

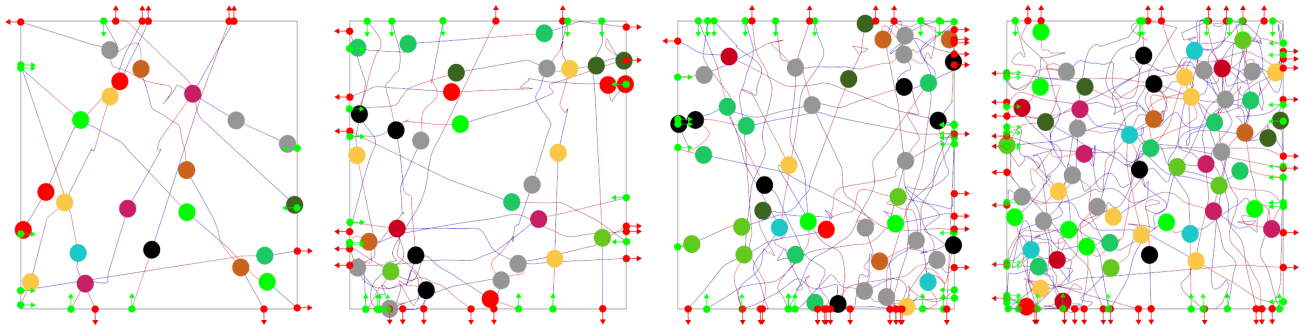
Temporal modifications of control points: In this paper, mostly spatial corrections to the trajectories of the agents are employed. For some scenarios, moving control points in space might not be the best approach, since this could result in either agents moving with unnatural speed and acceleration or into trajectories leaving the area of the patch (as described in the previous paragraphs). Collisions can be avoided if temporal displacement is applied to control points without any spatial modification which in essence accelerate (or decelerate) agents. Obviously, this could resolve spatial issues such as out of boundary cases but care should be taken so that agents' velocities remain realistic.

Obstacles: A patch can have static objects; i.e., obstacles. Having a small number of obstacles is equivalent to defining a set of special agents that are not allowed to move. Obviously, placement of these obstacles plays a significant role on the trajectory generation since boundary control points could be "trapped" or if a trajectory passes closely between two static obstacles it would end up in a position where the collision avoidance algorithm would not converge since a control point could end up oscillating between the obstacles. One possible solution to that is grouping obstacles together as bigger ones with a cost on the free area that can be used by other trajectories. Approaches to handle dense obstacles are currently being considered as alternatives to this approach and on the design of crowd patches. Obstacle handling is still under development and not implemented in the current version of the algorithm.



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

Figure 9: ***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 10: ***Density Adaptability*** *The proposed method can generate collision free trajectories for many situations; from sparse to very dense.*

Evaluating the quality of motion: One of the largest issues in crowd simulation research is evaluating the quality of generated behaviour. Currently, this is done by visual inspection which is a very biased approach. Several methods have been proposed to evaluate the quality of crowd motion in a quantitative approach that we plan on investigating 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 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 can then 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 that are generated 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 a globally optimal 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 from real-world crowds can also be used during optimization so that the generated trajectories provide more real life like behaviours. Evaluating the quality of the generated trajectories is one of the most important issues we are investigating. 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

This work is funded by the French National Research Agency ANR, project CHROME number ANR-12-CORD-0013. The authors would like to thank Tristan Le-Bouffant and Julian Joseph for their help on preparing the demonstrations.

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*.

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.

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*, ACM, SIGGRAPH Asia '10, 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*, 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.

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/, September.

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*, 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 *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '09, 207–214.