Optimization-based computation of locomotion trajectories for Crowd Patches

Abstract

For many years now, simulating crowds in virtual environments have become more and more important. We want to see crowds in movies, TV shows, video-games, etc, and we want them to look good and natural. An important part of crowd simulation is the way the people (or animals or any other moving agent inhabiting the environment) walk from one place to another. This paper concentrates in improving the crowd patches approach suggested by Yersin et al. [Yersin et al. 2009]. This method is based on the construction of animation blocks (called patches) stuck together to create a bigger and richer animation. The main purpose of this paper is to improve the way trajectories are created using an optimization-based method. Our main contributions include a better way to match the end points of trajectories using the Gale-Shapley algorithm [Gale and Shapley 1962] , a better method for collision avoidance that allows a more natural look to trajectories, as well as a proposal for smoothing the final trajectories using cubic splines. We show in the results several examples of patches and how they were improved by our method. We also discuss limitations and future work in the final sections.

CR Categories:

Keywords: keywords go here

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...

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 great performances [Pettré et al. 2006; Treuille et al. 2006], high densities [Narain et al. 2009] or controllable crowds [Guy et al. 2009]. There has 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, ...

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

Simulation-based techniques are ideal for creating an ambient crowd for large environments. 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, it is 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 please refer to [Yersin et al. 2009] for more details.

Nevertheless, using the crowd patches approach, it is important to work with a limited set of patterns 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 spatiotemporal 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).

Compared to previous techniques to compute internal trajectories, 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 *good* 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 in 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

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 trajectories initial generation, collision handling and smoothing out (Sections 3.3–3.5).

3.1 Definitions

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

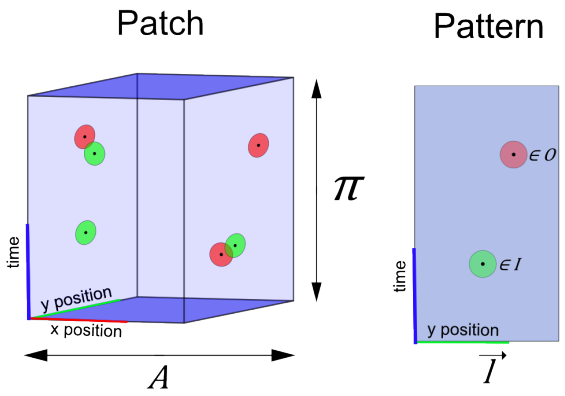


Figure 2: ***Patches and Patterns*** *A* patch *is defined by the geometrical area A where a set of dynamic and static objects (D and S respectively) can move over a period of time π.* Patterns *define boundary conditions for the patches and act as portals connecting neighboring patches.*

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.

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:

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

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 spacetime, 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 $\tau : [0, \pi] \rightarrow A$, 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. $\tau(0) = \tau(\pi)$.

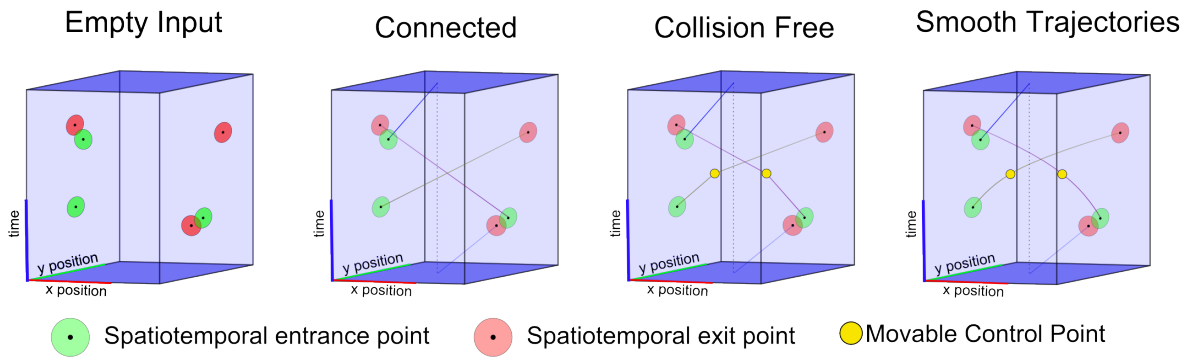


Figure 1: ***Overview*** *Input and output points in a patch's patterns are initially connected and subsequently modified using the proposed optimization approach and smoothed out to be collision free.*

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 $[0, t_{final}]$.

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 side 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, this will be observed as moving agents 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 should be generated that interconnect all of them. This process has three main steps (Figure 1):

1. Match the elements in the Input and Output sets contained within a patch. In this step, Entry to 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; these are bended iteratively until they are collision free. Points lying at the borders, i.e. Entry and Exit points, are hard restraints and can never be moved.

3. Smooth out trajectories (if needed). Lastly splines are used to minimize the hard turns making sure that the smoothed out trajectories stay as close as possible to the original ones from step 2 so that no new collisions are introduced.

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

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].

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 that 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*.

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 order of preference for matching (Figure 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 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]

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

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 (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 is 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 connect with 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; similarly to approach followed by [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.

To conclude, 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 result in collisions, since no special care was taken to handle that. To do so, an iterative approach has been proposed.

3.4 Removing Collisions

Most of the times, the set of linear trajectories generated by Algorithm 1 have collisions and therefore care should be taken to remove them since collisions rarely occur in real life human crowds. An algorithm that manipulates the linear trajectories by moving control

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 number.*

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

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

points is proposed. 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 simple; given a matrix M that stores the current minimum distances in-between all trajectories iterate modifying trajectories (and therefore 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 in each iteration (i.e., trajectories are split into segments) that are moved under some constraints until the trajectories are collision free.

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 two 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.

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

We find that in many 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.

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 did 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:

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

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 all the minimum distances between all the segments of two trajectories, finding the minimum distance is trivial (consult Table 2 for an example.

3.5 Smoothing

Smoothing: We have obtained some collision free trajectories, but they may have a jerky trajectory, so we need a final the step to smooth them. We need to take special care of how we do this, we don want the trajectories to change in such a way that they make

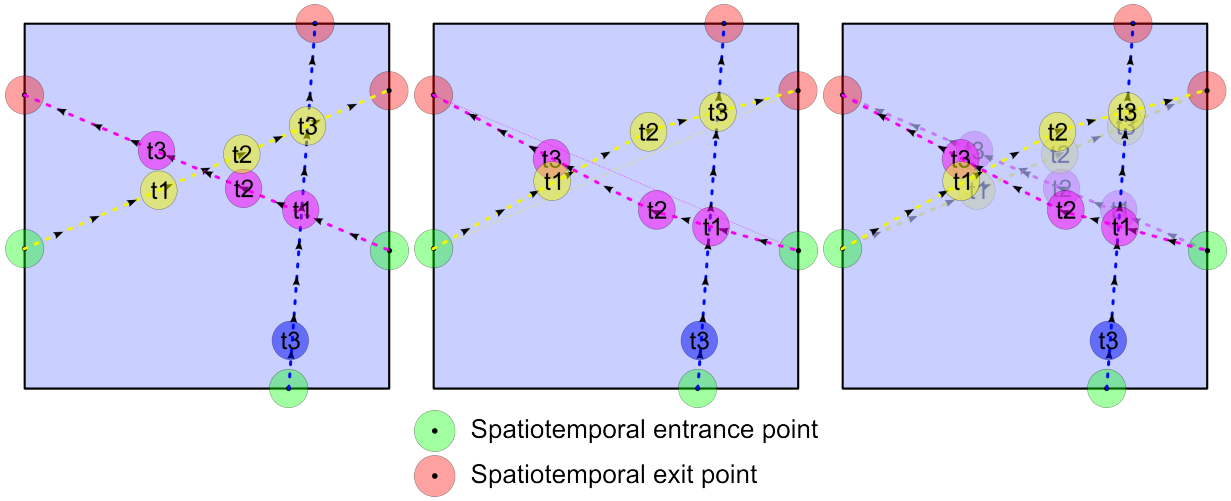


Figure 3: ***Collision removal***

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

new collisions. A simple approach is to use cubic splines. Cubic splines connects every pair of points in a trajectory with a polynomial function of degree 3:

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

In total, we will need to find the coefficients a, b, c, and d for every segment we have in our trajectory. Those coefficients are determined by continuity C^1 restrictions:

For every spline associated with a trajectory between control point $\mathbf{cp}_1 = (\mathbf{p}_1, t_1)$ and $\mathbf{cp}_2 = (\mathbf{p}_2, t_2)$, the spline must pass through those same points, that is, $S(t_1) = (\mathbf{p}_1)$ and $S(t_2) = (\mathbf{p}_2)$.

For every two contiguous splines S_1 and S_2, for the common point they share p, the velocity must be equivalent, that is, $S_1'(\mathbf{p}) = S_2'\mathbf{p})$.

This restrictions can be accommodated in a such way that they form a system of linear equations that can be solved using Cholesky decomposition.

We can not apply directly this method using our current control points. That results in trajectories that vary too much from the original, thus they may end up creating new collisions. To make the spline more similar to the original trajectory, we add new virtual control points sampled uniformly in the linear segments of the trajectories. We call these tacks. These tacks help fix the splines, because they will now pass through the control points and the tacks. Note that the more tacks we put, the closer the spline will fit to the original path.

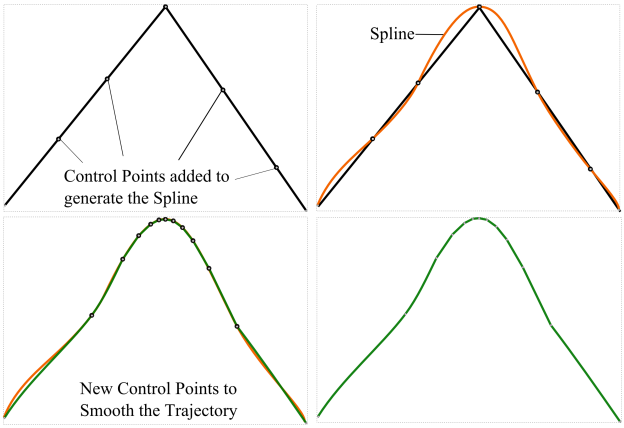


Figure 4: ***Smoothing Proccess***

After that, we compute the coefficients to create the splines and resample along the spline to get new control points. The new trajectory will still be composed of line segments, but since they are smaller, and following closely the path of a cubic function, to the naked eye they will look like curves. The finer the sample, the more it will look like a smooth path.

For optimization purposes, the second sampling is not taken uniformly. The trajectories are relatively straight in the middle of the the segments and have higher curvature near control points. For that reason, we take more samples around inflexion points (old control points) than far from them. These samples become the new control points that define the trajectory.

There may be cases, even with a healthy number of tacks, 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 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 follow either of them when walking, so other methods can be tried to improve this initial approach.

4 Results

We found that our algorithm created trajectories that were free of collisions and produced smooth motion. In Figure 5 we have a diagram showing the patch in various steps of our algorithm. Next we show the final curved trajectories with various sized input. Lastly we show some performance tests.

the images are not lining up how I want them too. Devin, I took the liberty of modifying the figures below to show you how its typically done. Note the label for referencing the figures in the text. I don't really know how to control the position at which the figures appear, or where do we want them exactly.

The performance was measured on a computer with an Intel Xeon quad core 2.8 GHz processor and 8GB of RAM. We found trajectories for patches with randomly generated entry and exit points. The patches all had base square of size 16 by 16. The period was 10 seconds. There were 20 samples for each number of agents between 1 and 50 for a total of 1000 samples. 994 of these were collision free. The other 6 were stopped after they reached 2000 iterations. We recorded the number of iterations required for collision free trajectories as well as the computation time in seconds.

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 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.

6 Conclusions

We present a novel technique for computing trajectories for use in crowd patches. We can take an empty patch and generate trajectories within that can represent human motion. Our technique produces smooth collision free motion. This algorithm is slow to converge, especially with larger groups. Furthermore while our approach produces collision free trajectories it is lacking in other aspects of human motion. For one, agents can reach unrealistic speeds. Additionally agents can have abnormal motion that is not true to human behavior. In the future we plan to implement a more global approach that looks at more than just the closest agent and incorporates speed into the calculations. We hope that this will provide faster convergence in high density situations as will as fixing motion artifacts.

- reminder of contributions - comment on results and limitations - paths for future work.

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*.

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.

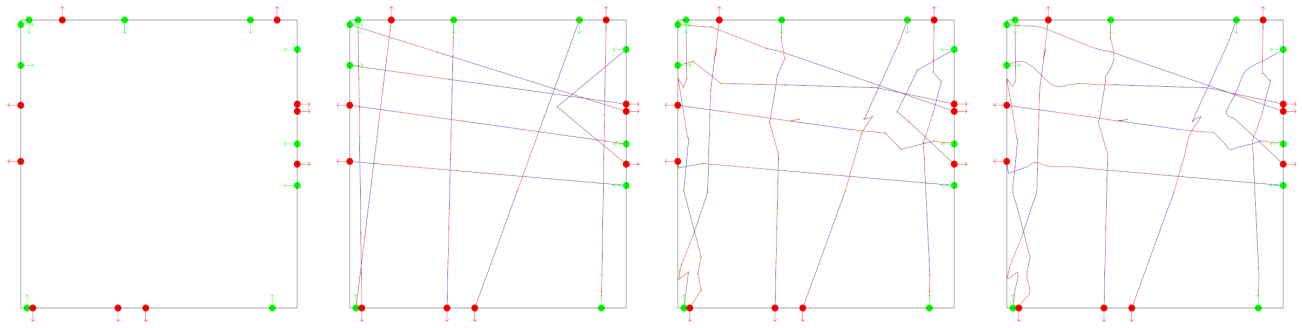
GUY, S. J., CHHUGANI, J., KIM, C., SATISH, N., LIN, M., MANOCHA, D., AND DUBEY, P. 2009. Clearpath: Highly parallel collision avoidance for multi-agent simulation. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation*, ACM, SCA '09, 177–187.

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.

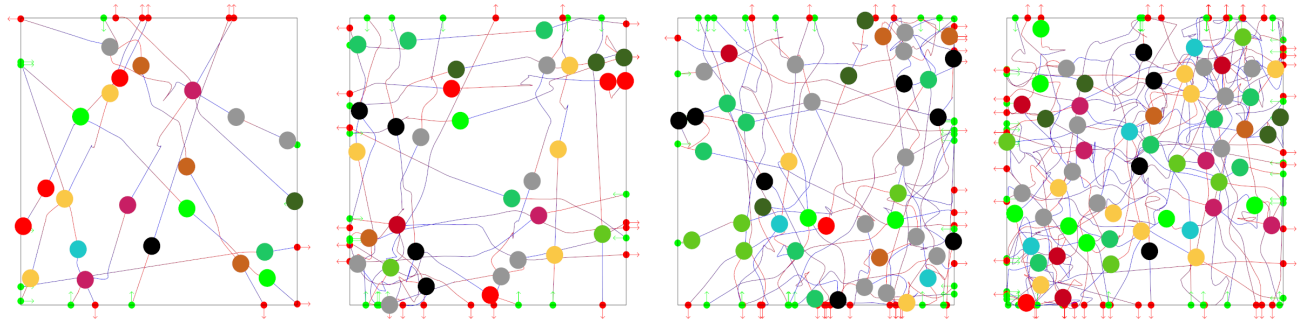
LERNER, A., CHRYSANTHOU, Y., AND LISCHINSKI, D. 2007. Crowds by example. *Computer Graphics Forum 26*, 3 (September), 655–664.

NARAIN, R., GOLAS, A., CURTIS, S., AND LIN, M. C. 2009. Aggregate dynamics for dense crowd simulation. In *Proc. of*



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

Figure 5: *Various steps of our algorithm.*



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

Figure 6: *Various input to our algorithm.*

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.

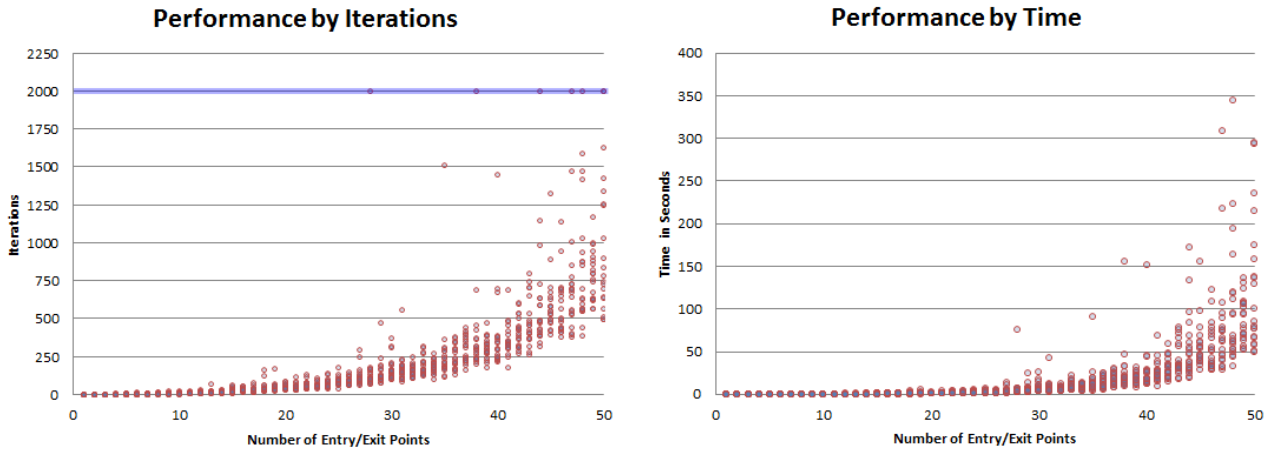
THALMANN, D., AND RAUPP MUSE, S. 2013. *Crowd Simulation*, 2nd ed. Springer.

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.

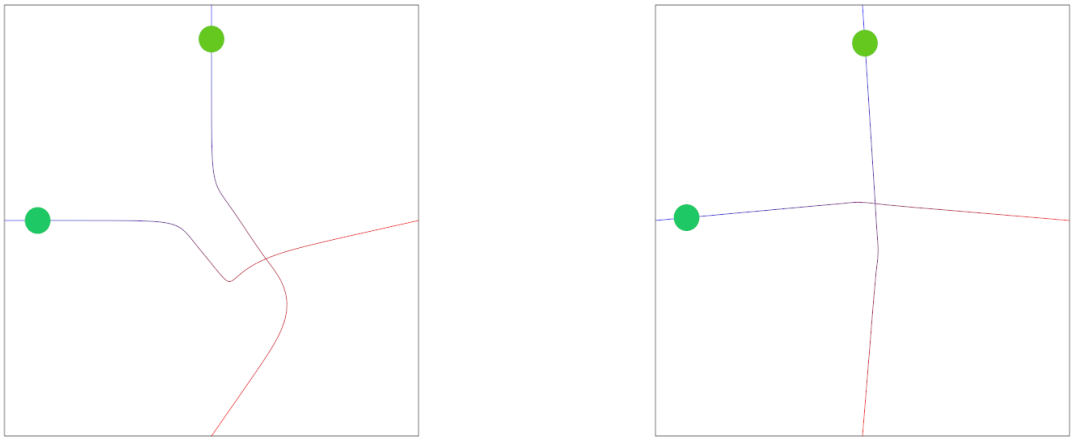
YERSIN, B., MAÏM, J., PETTRÉ, J., AND THALMANN, D. 2009. Crowd patches: Populating large-scale virtual environments for real-time applications. In *Proc. of the 2009 Symp. on Interactive 3D Graphics and Games*, ACM, I3D '09, 207–214.



(a) **(b)**

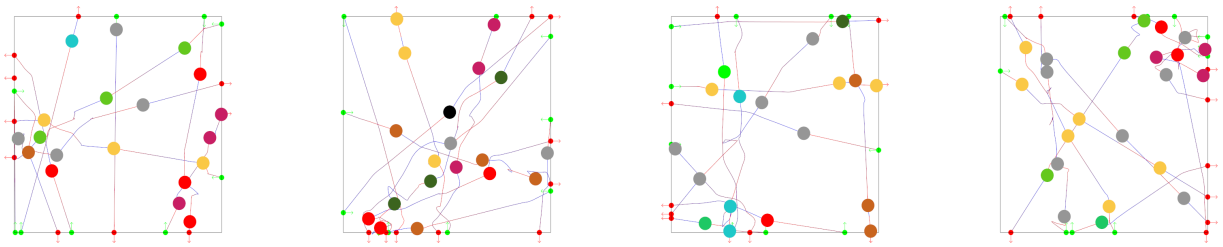
Figure 7: *A dot on each graph represents a simulated patch. The patch had a specified number of entrance and exit points whose locations were randomly generated. Our algorithm was run until either the patch was collision free, or it reached 2000 iterations.* ***(a)*** *The the number of iterations was recorded in relation to number of Entry/Exit Points. The blue line at 2000 iterations shows where the simulation was stopped (6 out of 1000 patches were stopped here, and thus were not collision free).* ***(b)*** *The time to converge was measured in seconds. is this explanation more clear?*

//each experiment needed to converge whereas the blue horizontal line indicates the maximum number of allowed iterations (6 *out of* 1000 *inputs What do we mean by inputs? experiments or the total number of trajectories. Does this graph show how much time it took for a whole patch or for the trajectories with collisions? didn't converge after* 2000 *iterations) .* ***(b)*** *The time to converge was measured in seconds. should I remove the titles on the images of the graphs, and instead put their title in the caption? No, you should leave them.. If you can edit the images, please remove the borders. I think it will look nicer. Also, this graphs confuse me a little bit. We should discuss them with Guillermo - I want to get a feeling of what exactly are we showing here*



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

Figure 8: *A comparison between both methods, shows the improvement in terms of anticipating a collision. Figure (a) shows little anticipation, it modifies its trajectory until it is about to collide, whereas (b) starts modifying its trajectory gradually from the beginning.*



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

Figure 9: *Several examples showing the final trajectories for 10 Entry/Exit Points*