Synchronizing navigation algorithms for crowd simulation via topological strategies

Wouter van Toll, Julien Pettré

Computers & Graphics (2020)

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Synchronizing navigation algorithms for crowd simulation via topological strategies

Wouter van Toll[a,*], Julien Pettré[a]

[a] *Univ Rennes, Inria, CNRS, IRISA, France*

ARTICLE INFO

Article history:
Received 31 January 2020
Received in final form 31 March 2020
Accepted 13 April 2020
Available online 20 April 2020

Keywords: Navigation, Crowd simulation, Intelligent agents
2010 MSC: 68T42

ABSTRACT

We present a novel topology-driven method for enhancing the navigation behavior of agents in virtual environments and crowds. In agent-based crowd simulations, each agent combines multiple navigation algorithms for path planning, collision avoidance, and more. This may lead to undesired motion whenever the algorithms disagree on how an agent should pass an obstacle or another agent.

In this paper, we argue that all navigation algorithms yield a *strategy*: a set of decisions to pass obstacles and agents along the left or right. We show how to extract such a strategy from a (global) path and from a (local) velocity. Next, we propose a general way for an agent to resolve conflicts between the strategies of its algorithms. For example, an agent may re-plan its global path when collision avoidance suggests a detour. As such, we bridge conceptual gaps between algorithms, and we synchronize their results in a fundamentally new way. Experiments with an example implementation show that our strategy concept can improve the behavior of agents while preserving real-time performance. It can be applied to many agent-based simulations, regardless of their specific navigation algorithms. The concept is also suitable for explicitly sending agents in particular directions, e.g. to simulate signage.

DOI: https://doi.org/10.1016/j.cag.2020.04.003.

1. Introduction

Simulating the motion of human crowds is a research topic with many real-world and entertainment applications [1, 2]. Many crowd-simulation techniques are *agent-based*: they model each member of the crowd as an intelligent agent with its own properties and goals. Within this paradigm, it is common to split an agent's navigation task into *global* path planning (finding an overall route to the goal) and *local* behavior (following this route while avoiding short-term collisions) [3]. Several *mid-term* planning algorithms can be added as well [4, 5, 6]. As such, agents combine navigation algorithms for different purposes, each solving their own 'level' of the navigation problem.

This multi-level navigation approach often works well, but it can have problems in scenarios with many agents or obstacles. The algorithms of different levels may give conflicting commands, for example when an agent's global path runs through a passage that turns out to be blocked by other agents (as in Figure 1(b)). The agent is then likely to get stuck, unless the global and local algorithms actively collaborate to resolve the issue.

Although solutions (e.g. planning a detour or waiting) may seem obvious to the human eye, it is difficult to detect automatically when navigation algorithms are in conflict. We argue that this is partly due to a design choice in local algorithms: in each simulation frame, their outcome is simply a velocity that optimizes local criteria, and not an *explicit decision* to pass agents or obstacles on a certain side.

*Corresponding author.
e-mail: wouter.van-toll@inria.fr (Wouter van Toll)

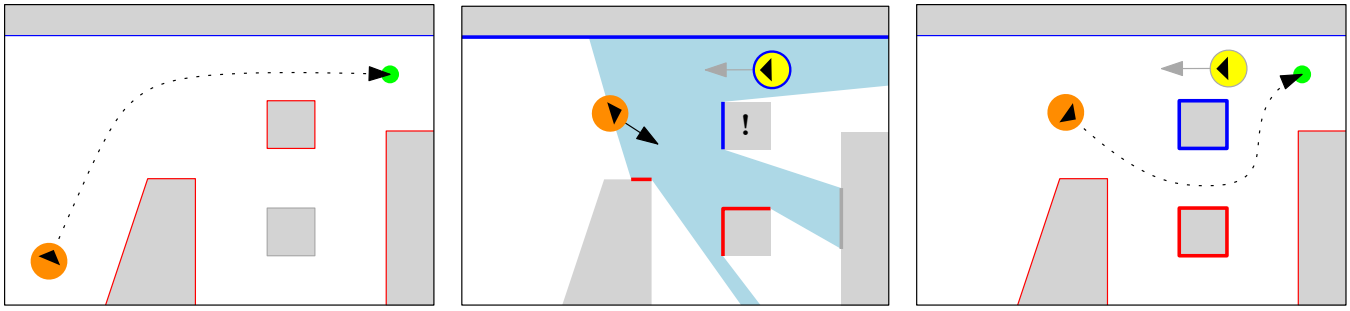


Fig. 1. We define a *navigation strategy* as a set of decisions to pass obstacles and agents via the left (red) or right (blue). Left image: A global path yields a long-term strategy for obstacles. Middle image: A local velocity yields a short-term strategy for nearby agents and obstacles. For the obstacle marked with '!', the global and local strategies are in conflict. Right image: Our method detects such conflicts and allows the agent to plan a new global path that complies with its local strategy.

1.1. Goals and contributions

We present a novel topology-inspired approach to improve agent navigation in crowds. We propose the concept of a *navigation strategy*: a set of decisions to move around certain obstacles or agents via the left or right. Such a strategy can be computed for various navigation results, such as a global path and a local velocity. With this unified representation, we can systematically detect and resolve conflicts between the algorithms used by an agent. This can yield more responsive agents and more efficient crowd motion. Figure 1 shows an example: we can let an agent plan a (global) detour when (local) collision avoidance suggests this. This kind of synchronization between algorithms was previously not possible because paths and velocities did not have a common meaning.

Planning a new global path (i.e. *re-planning*) is a relatively costly operation that one would prefer to avoid unless it is necessary. A conflict between strategies is an intuitive trigger for re-planning. By detecting these conflicts, we contribute to an efficient simulation where agents re-plan at the right time.

Our main contributions are the following:

- We formulate an agent's navigation behavior as a topological *strategy* amidst obstacles and agents (Section 3.2).
- We show how to obtain such a strategy from a global path (Section 4.1) and from a local velocity (Section 4.2).
- We present a simulation framework in which path planning, path following, and collision avoidance collaborate to resolve conflicts between their strategies (Section 5).
- We generalize this concept to an abstract way for agents to resolve conflicts between the strategies of arbitrary navigation algorithms (Section 6).
- Via experiments using an example implementation (Section 7), we show that this conflict resolution can improve the crowd's behavior in several scenarios, without sacrificing real-time performance.
- We also show (in Section 7.4) that our concepts can be used to easily guide agents in specific directions, e.g. to simulate the effect of signage in buildings.

We emphasize that we do *not* propose a new algorithm for mid-term planning or collision avoidance. Instead, we offer a fundamentally new way to harmonize existing agent-navigation algorithms, by assigning a uniform meaning to the choices made at each level. This concept can be applied to many different navigation algorithms and simulation frameworks.

1.2. Extension of previous work

This paper is an extension of a conference publication [7] in which we used the strategy concept to resolve conflicts between path following and collision avoidance. In this extended work, Section 6 generalizes this idea to conflict resolution between arbitrary navigation algorithms in a more abstract framework. To demonstrate this, we add an optional mid-term planning algorithm to our implementation (see Appendix A), and we give agents the option to resolve conflicts at this level as well. In Section 7, we add this variant of the simulation to our experiments. We also show more scenarios, and we provide a more thorough analysis of the system's parameters and performance.

2. Related work

In an agent-based crowd simulation, each member of the crowd plans a path to its own goal. While traversing their paths, agents use local algorithms to adjust their motion when needed. This *multi-level* approach is the current state of the art; many such systems have been developed in recent years [8, 3, 9, 10, 11].

2.1. Global and local navigation

Global navigation concerns the computation of an overall path through a 2D or 3D environment with static obstacles. To facilitate this, a *navigation mesh* subdivides the environment into regions [12]. Many types of navigation meshes exist that can be computed automatically [13, 14, 15]. Given a navigation mesh, agents plan a path on the dual graph of the mesh regions, using graph search algorithms such as A* [16]. This results in a sequence of regions for the agent to move through. Within this sequence, agents can compute a specific geometric curve to follow. We will use the term *(global) path* for this curve.

Local navigation concerns the agent's motion during the simulation loop. In each frame of the loop, agents first choose a point on their global path to move to. This *path following* induces a preferred velocity $\mathbf{v}_{\text{pref}}$ per agent [17, 18]. Then, agents apply *collision avoidance* to find a velocity $\mathbf{v}_{\text{new}}$ that is (ideally) close to $\mathbf{v}_{\text{pref}}$ while avoiding nearby obstacles *and* agents. Collision avoidance is a popular research topic, and increasingly intelligent solutions have been proposed [19, 20, 21, 22]. Local navigation can also entail other tasks, such as letting agents move together in small groups [23, 24].

The term 'crowd simulation' is sometimes used for collision avoidance alone. However, path following and global planning are equally important, especially in environments with many obstacles. Small obstacles could be handled purely locally, but this only works up to a certain point that is hard to define. Therefore, we propose a solution where local and global planning (and potentially other algorithms) can coordinate.

2.2. Extensions and alternatives

In some scenarios (e.g. in the case of high-density crowds), a 'regular' combination of global and local navigation may not be able to bring each agent to its goal. To improve the behavior in such cases, there have been several attempts to add new 'levels' of navigation in-between. We will refer to this idea collectively as *mid-term* navigation because it conceptually lies between path planning and collision avoidance. One category of mid-term algorithms adapts the agent's *preferred velocity* based on local information, such as the crowd density [5] or crowd flow [6]. Another category refines the agent's *path* to include short-term interactions with other agents [4].

The work by Kapadia et al. [11] also defines several levels of navigation (down to the level of skeletal animation), and it uses the results of higher levels to narrow down the search at lower levels. This is essentially a more detailed version of the navigation hierarchy outlined earlier. Our work focuses specifically on handling *conflicts between levels*.

The behavior of agents can also be improved by adding local information to global planning, such as information about the crowd density [25, 26, 27]. This makes an existing navigation algorithm more intelligent instead of adding a *new* algorithm.

In general, while additional (or more informed) algorithms can yield improved behavior, the navigation algorithms are still isolated processes that can produce conflicting strategies. This paper focuses on resolving this issue.

Another option is to remove the global-local distinction altogether by adding the agents as 'obstacles' to the navigation graph [28, 29]. This is ideal for dense and (nearly) stationary crowds, where an agent should actively look for gaps between other agents. For other use cases, a combination of global and local planning generally works better. The 'agents as obstacles' concept could still be integrated into a traditional framework, as a mid-term algorithm that plans manoeuvres around agents.

Alternatively to agent-based methods, *flow-based* simulations model the crowd as a whole that moves along a flow field [30, 31]. These methods do not suffer from conflicts as much, but they cannot efficiently model crowds in which all agents have individual properties and goals. In this paper, we therefore focus on agent-based simulations.

2.3. Topology-driven navigation

All agent navigation methods, at every level of detail, have one common trait: they compute how an agent should move around objects (which can be agents or obstacles). In this paper, we propose the concept of a *navigation strategy* to formalize exactly this. It allows us to compare the navigation plans of different levels and to resolve conflicts between them.

Our idea of enriching navigation with topology is not new. Robotics researchers have added topological constraints to global [32] and local planning [33], and they have formulated multi-robot coordination in terms of topological decisions [34]. Our work uses similar ideas, but focuses on the synchronization *between* levels of planning, and on the integration into real-time crowd simulations.

3. Definitions

As in most crowd-simulation research, we approximate *agents* by disks. Let $AG = \{A_j\}_{j=0}^{m-1}$ be the set of m agents being simulated. All agents A_j are moving towards their own goal positions $\mathbf{g}_j$, using any combination of navigation algorithms.

We will gradually introduce new symbols throughout this paper. For convenience, a summary of the most important notation is given in Table 4.

3.1. Environment and navigation mesh

We assume that the simulation takes place in a two-dimensional *environment* $\mathcal{E}$ with polygonal obstacles and a polygonal outer boundary, for which we will use the term 'obstacle' as well. Let $OBS = \{O_i\}_{i=0}^{n-1}$ be the set of n obstacles in $\mathcal{E}$, and let N be the total number of vertices for all obstacles. We assume that each obstacle O_i is a simple polygon, that the inner obstacles do not overlap, and that the outer boundary contains all inner obstacles. (If necessary, such an obstacle representation can be obtained from a navigation mesh.)

To represent the environment for navigation purposes, we require a *navigation mesh* $\mathcal{M}$ that subdivides the obstacle-free space of $\mathcal{E}$ into non-overlapping polygonal regions $REG = \{R_i\}_{i=0}^{r-1}$. For any region R_i, all vertices should be on the boundary of an obstacle. Consequently, each boundary segment of R_i either lies on an obstacle boundary or is shared with another region R_j. We will use the term *passage* for a boundary segment shared by two regions. Each passage is a line segment that starts and ends at an obstacle. Figure 2 shows two possible navigation meshes for an example environment. Many types of navigation meshes exist, including ones where the combined complexity of all regions is $O(N)$.

3.2. Decisions and strategies

We now formalize the concept of a *navigation strategy*: a set of decisions to pass obstacles and agents on a certain side. Later, we will use this definition to assign a common topological meaning to the results of global and local planning.

During the simulation, any agent A_j that has not yet reached its goal should navigate around obstacles and other agents, to which we will collectively refer as *objects*. A_j can decide to pass any object either via the left or via the right. Formally, let

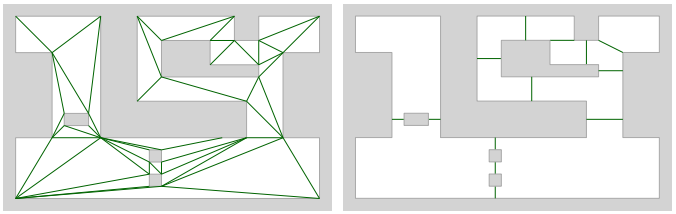


Fig. 2. An environment with two possible navigation meshes. Passages are shown in green. Left: triangulation. Right: local minima of the medial axis, used in this paper.

$D(A_j, B) \in \{\texttt{L}, \texttt{R}, \texttt{X}\}$ be a *decision* of A_j with respect to an object B:

- $D(A_j, B) = \texttt{L}$ if A_j intends to pass B via the *left* (thus keeping B on their right-hand side);
- $D(A_j, B) = \texttt{R}$ if A_j intends to pass B via the *right* (thus keeping B on their left-hand side);
- $D(A_j, B) = \texttt{X}$ if A_j has not (yet) made a decision for B.

In the figures of this paper, we will use the color red to denote the decision L, blue to denote the decision R, and gray to denote X. Note that we can define decisions for both static and dynamic objects. This difference does not matter conceptually, but it may inpact how a decision is *computed*, as we will show in Section 4.

Next, a *navigation strategy* $S_{(\cdot)}$ for A_j is a set of decisions $D_{(\cdot)}$ with respect to all objects:

$$S_{(\cdot)}(A_j) = \{D_{(\cdot)}(A_j, B)\}_{B \in OBS \cup AG - \{A_j\}}$$

where $(\cdot)$ is an arbitrary subscript to denote the correspondence between S and D. Throughout this paper, we will use different subscripts depending on the context, and we will omit the subscript when it is not relevant. Also, we will typically omit the argument A_j because A_j is always the agent in question.

In terms of coloring, a strategy is an assignment of the colors red and blue to the objects between which the agent wants to pass, while leaving any irrelevant (or undecided) objects in gray. Decisions of X do not have to be stored explicitly in a strategy; for any object that is not mentioned, we will implicitly assume a decision of X.

Figure 3 shows an example of a strategy. Here, note that the decision regarding a neighboring agent A_k also depends on the *velocity* of A_k. For instance, the agent A_4 is blue because it will have moved to the left before A_0 passes it. We will explain this further in Section 4.2.

In a typical crowd simulation, each agent A_j tries to follow a global path. This path has a corresponding *global* strategy S_G that usually only concerns static obstacles. Thus, it does not yet describe how to avoid agents, i.e. $D_G(A_j, A_k) = \texttt{X}$ for all other agents A_k. Over time, local navigation algorithms should determine how to move around agents. This results in one or more *local* strategies (one for each algorithm) that change over time. These local strategies also impose their own decisions for nearby *obstacles*, and these decisions may be in conflict with the global strategy S_G.

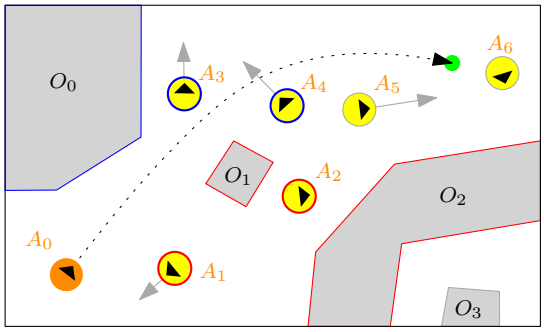


Fig. 3. A navigation strategy. The agent A_0 will move to its goal (green) by passing obstacles and other agents via the right (outlined in blue) or via the left (outlined in red). Undecided or irrelevant objects are outlined in gray.

3.3. Comparing strategies

To solve navigation problems, it is useful to define consistency between strategies. First, we say that two *decisions* $D_1(A_j, B)$ and $D_2(A_j, B)$ for a given object B are *inconsistent* if one decision is L and the other is R. Otherwise (i.e. if they are equal or if either decision is X), the decisions are *consistent*. We will use $=_c$ to denote consistency and $\neq_c$ for inconsistency.

Now, let S_1 and S_2 be two full *strategies*. For any object B where $D_1(A_j, B) \neq_c D_2(A_j, B)$, we say that S_1 and S_2 have a *conflict*. The strategies are *consistent* if and only if they have no conflicts, i.e. $S_1 =_c S_2$ iff $D_1(A_j, B) =_c D_2(A_j, B)$ for all B. Otherwise, the strategies are *inconsistent* ($S_1 \neq_c S_2$).

The problem from Figure 1 occurs when the strategies of collision avoidance and path following have an obstacle-related conflict. Section 5 focuses on resolving conflicts of this type.

3.4. Summary

To summarize, given an agent A_j:

- $D_{(\cdot)}(A_j, B) \in \{\texttt{L}, \texttt{R}, \texttt{X}\}$ is a *navigation decision* for A_j with respect to an *object* B (an obstacle or an agent).
- $S_{(\cdot)}(A_j)$ is a *navigation strategy* for A_j, which is a set of decisions $D_{(\cdot)}$ with respect to all objects.
- Two strategies S_1 and S_2 are inconsistent ($S_1 \neq_c S_2$) iff they have a *conflict*, meaning that there is an obstacle B for which $D_1(A_j, B) = \texttt{L}$ and $D_2(A_j, B) = \texttt{R}$ (or vice versa).
- In a crowd simulation, each navigation algorithm of A_j yields its own strategy. The purpose of this paper is to detect and resolve conflicts between these strategies.

4. Obtaining navigation strategies

This section explains how to obtain navigation strategies from a *path* and from a *velocity*. Section 5 will integrate this into a simulation framework that handles conflicts between an agent's navigation algorithms.

4.1. Converting a path to a strategy

A global planning algorithm computes a *path* from a start position $\mathbf{s}$ to a goal position $\mathbf{g}$. This path is a curve $\pi : [0, 1] \to \mathbb{R}^2$ through the environment, where $\pi(0) = \mathbf{s}$ and $\pi(1) = \mathbf{g}$. The curve does not intersect any obstacles in OBS, but it may intersect the agents in AG because agents are (usually) not yet considered in this phase.

We now describe how to obtain a navigation strategy S from a path π. In other words, we show how π leads to a red/blue coloring of obstacles. Figure 4 shows an example of a path and its strategy. Note that the path keeps blue obstacles to its left and red obstacles to its right, while some obstacles remain gray.

4.1.1. Overview

To convert π to a strategy S, the first step is to find the sequence of *passages* that π traverses in the navigation mesh $\mathcal{M}$. In this paper, we assume that $\mathcal{M}$ was already used to compute π in the first place. The sequence of passages for π can then easily be constructed during the path-planning algorithm itself. (If π is an arbitrary curve that was not computed using $\mathcal{M}$, one could compute the passages that π intersects, and remove any duplicate entries caused by backtracking or loops in the path.)

Let $\mathcal{P} = \{p_i\}_{i=0}^{k-1}$ be the sequence of k passages that π traverses. The example path in Figure 4 visits nine passages. Recall that each passage p_i is a line segment with an obstacle at its left and right endpoint; let us denote these obstacles by $O_{i,l}$ and $O_{i,r}$ respectively. By traversing p_i, the agent will pass $O_{i,l}$ via the right (blue) and $O_{i,r}$ via the left (red). To compute all relevant navigation decisions D, we check the traversed passages one by one, and we perform the following steps for each p_i:

1. Try to set $D(A_j, O_{i,l})$ to `R`; that is, try to give $O_{i,l}$ the color blue. (We will explain below what 'trying to set' means.)
2. Try to set $D(A_j, O_{i,r})$ to `L`; that is, try to make $O_{i,r}$ red.
3. If p_i is not the last passage of $\mathcal{P}$, let R_i be the region to which p_i leads. In the clockwise ordering of passages bounding R_i, visit all passages between p_i and p_{i+1}. The light-blue passage in Figure 4 is an example of this case. For each such passage p', get its left obstacle O'_l and try to set $D(A_j, O'_l)$ to `R`.
4. Symmetrically, visit all passages in the counter-clockwise ordering between p_i and p_{i+1}. Examples are the pink passages in Figure 4. For each such passage p', get its right obstacle O'_r and try to set $D(A_j, O'_r)$ to `L`.

This results in a strategy S, implicitly augmented with $D(A_j, B) =$ `X` for any undecided object B. If the boundaries of regions can be traversed via pointers (next/prev/twin), we can compute S in $O(N')$ time, where N' is the total complexity of the regions that π visits.

4.1.2. Handling ambiguities

We use the term 'try to set' instead of 'set' because an obstacle O might occur as both a left and right obstacle at different points along the path. The outer boundary of Figure 4 is an example of this. In such cases, we want to set $D(A_j, O)$ to `X`, as we cannot use either `L` or `R` along the whole path. So, in the discussion above, 'try to set $D(A_j, O)$ to `R`' is defined as follows:

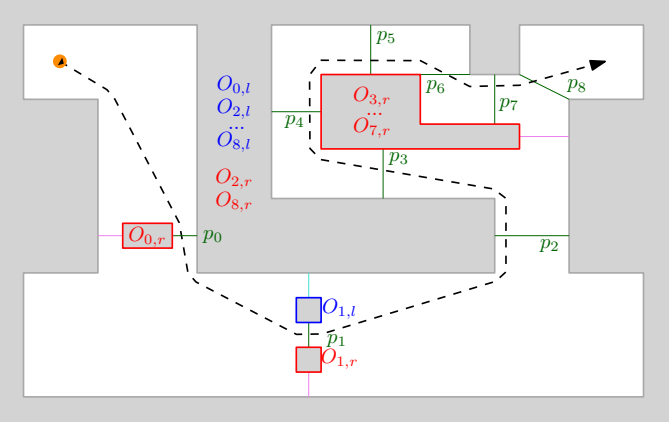


Fig. 4. A global path (the dashed curve) and its navigation strategy, obtained from the relevant passages (in green) and their neighboring passages (in light blue and pink).

- If $D(A_j, O)$ has not yet been set before, set it to `R`.
- If $D(A_j, O)$ was already set to `L`, set it to `X`.
- If $D(A_j, O)$ was already set to `R` or `X`, then it stays that way.

The definition for the decision `L` is symmetric.

4.2. Converting a velocity to a strategy

In each simulation frame, every agent A_j performs local algorithms for e.g. path following and collision avoidance. Each algorithm computes a *velocity* that is used in a certain way. To detect conflicts between algorithms, we need to convert their velocities to strategies.

We now explain how to obtain a navigation strategy S for an arbitrary velocity $\mathbf{v}$. That is, we show how to compute a red/blue coloring of nearby objects assuming that agent A_j uses the velocity $\mathbf{v}$ for a certain amount of time. We focus on non-zero velocities. If $\|\mathbf{v}\| = 0$, the agent A_j does not really 'navigate', and we can (implicitly) use $D(A_j, B) =$ `X` for all B.

In the following discussion, $\overline{\mathbf{ab}}$ denotes the line segment between two points $\mathbf{a}$ and $\mathbf{b}$, and $\angle(\mathbf{u}, \mathbf{v})$ denotes the angle (in radians) between two vectors $\mathbf{u}$ and $\mathbf{v}$. Let $\mathbf{p}$ be the current position of agent A_j, and let τ be the *time window*, a parameter that we will set in Section 5. In τ seconds, the velocity $\mathbf{v}$ will send A_j to position $\mathbf{p}' = \mathbf{p} + \mathbf{v} \cdot \tau$.

4.2.1. Neighboring objects

The short-term use of a velocity only affects objects that are nearby. Similarly to collision-avoidance algorithms, we consider nearby *agents* and nearby *obstacle segments*. Thus, we treat the boundary segments of obstacles as separate entities. Appendix A specifies how we find these neighboring objects in our implementation. For each neighboring object, we determine a navigation decision. The final strategy S consists of all these decisions, (again) implicitly augmented with `X` for unhandled objects.

4.2.2. Decision for a neighboring agent

Figure 5(a) shows how the segment $\overline{\mathbf{pp'}}$ divides the vicinity of A_j into four areas: back (a_B), front (a_F), left (a_L), and right (a_R). For a neighboring agent A_k with position $\mathbf{p}_k$ and velocity $\mathbf{v}_k$, let $\mathbf{p}'_k = \mathbf{p}_k + \mathbf{v}_k \cdot \tau$ be the predicted position of A_k after τ seconds. We define the decision $D(A_j, A_k)$ as follows:

- If $\mathbf{p}_k$ lies in a_B, then A_k starts behind A_j, so A_j does not actively pass it (anymore). Thus, $D(A_j, A_k) = \texttt{X}$.
- If $\mathbf{p}'_k$ lies in a_F, then A_k ends up in front of A_j, so A_j does not pass it (yet). Thus, $D(A_j, A_k) = \texttt{X}$.
- Otherwise, if $\overline{\mathbf{pp'}}$ and $\overline{\mathbf{p}_k\mathbf{p}'_k}$ intersect at some point $\mathbf{x}$, let t be the time after which A_j will reach $\mathbf{x}$. The decision depends on the position of A_k at that time, i.e. on $\mathbf{p}''_k = \mathbf{p}_k + \mathbf{v}_k \cdot t$:
  - If $\mathbf{p}''_k$ lies exactly on $\overline{\mathbf{pp'}}$, then $D(A_j, A_k) = \texttt{X}$.
  - If $\mathbf{p}''_k$ lies in a_L, then $D(A_j, A_k) = \texttt{R}$.
  - If $\mathbf{p}''_k$ lies in a_R, then $D(A_j, A_k) = \texttt{L}$.
- Otherwise, $\overline{\mathbf{p}_k\mathbf{p}'_k}$ lies fully or partly in either a_L or a_R.
  - If $\overline{\mathbf{p}_k\mathbf{p}'_k}$ overlaps with a_L, then $D(A_j, A_k) = \texttt{R}$.
  - If $\overline{\mathbf{p}_k\mathbf{p}'_k}$ overlaps with a_R, then $D(A_j, A_k) = \texttt{L}$.

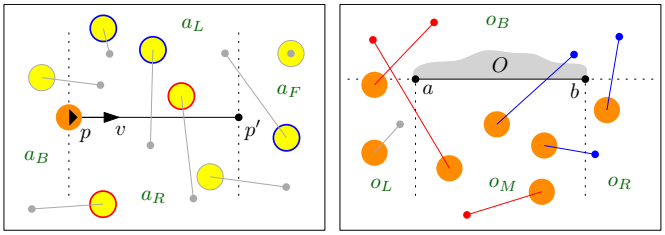
Our definition ignores whether A_j and A_k are on collision course. Even if $\mathbf{v}$ would lead to a collision, it is useful to know which navigation decision A_j is most likely to have in mind.

4.2.3. Decision for a neighboring obstacle segment

For obstacle segments, different segments of the same obstacle O may give conflicting decisions. When this happens, we would like to set $D(A_j, O)$ to X, as we cannot decide uniformly for the entire obstacle. Therefore, just like in Section 4.1, we treat each obstacle segment either by ignoring it or by *trying* to set the decision for the corresponding object O, where 'trying to set' is defined as before.

For a segment $\overline{\mathbf{ab}}$ belonging to an obstacle O, assume (w.l.o.g.) that $\mathbf{a}$ and $\mathbf{b}$ appear on O's boundary in counterclockwise order. $\overline{\mathbf{ab}}$ divides the space into four areas, as shown in Figure 5(b): back (o_B), left (o_L), middle (o_M), and right (o_R). We treat $\overline{\mathbf{ab}}$ as follows:

- If $\mathbf{p}$ lies in o_B, ignore this segment: it is currently not visible to the agent A_j.
- If $\overline{\mathbf{pp'}}$ stays entirely inside o_L or o_R, ignore this segment: A_j does not make an active decision yet.
- If $\overline{\mathbf{pp'}}$ and $\overline{\mathbf{ab}}$ intersect, A_j will collide with $\overline{\mathbf{ab}}$. The decision depends on the angle $\theta = \angle(\mathbf{v}, \mathbf{b} - \mathbf{a})$:
  - If $\theta < \pi/2$, try to set $D(A_j, O)$ to R.
  - If $\theta > \pi/2$, try to set $D(A_j, O)$ to L.
  - If $\theta = \pi/2$, ignore this segment.



(a) Agent-Agent (b) Agent-Obstacle

Fig. 5. A velocity v and its navigation strategy, with respect to (a) other agents, and (b) an obstacle segment.

- Otherwise, if $\overline{\mathbf{pp'}}$ moves into o_B, let $\mathbf{x}$ be the first point where this happens.
  - If $\mathbf{x}$ is closer to $\mathbf{a}$ than to $\mathbf{b}$, try to set $D(A_j, O)$ to L.
  - Otherwise, try to set $D(A_j, O)$ to R.
- Otherwise, A_j moves along the segment without fully passing it. The decision then depends on θ as described earlier.

5. A basic strategy-driven simulation framework

This section shows how to incorporate the strategy concept into a crowd simulation. For ease of comprehension, we start with the basic simulation framework from our previous publication [7], and we will generalize the idea in Section 6.

The implementation that we use for our experiments is similar to the framework described here, except that it includes one additional navigation algorithm. For simplicity, we will ignore this extra algorithm in this section.

5.1. Navigation algorithms

In this first version of the framework, each agent A_j has three navigation tasks:

- Path planning (performed once): Compute a global path π to the agent's goal position $\mathbf{g}_j$.
- Path following (performed in each frame): First compute a *reference point* $\mathbf{p}_{\text{ref}}$ that denotes the agent's progress along π. Then compute an *attraction point* $\mathbf{p}_{\text{att}}$ on π, and a preferred velocity $\mathbf{v}_{\text{pref}}$ that sends A_j to $\mathbf{p}_{\text{att}}$ at its preferred speed s_{pref}.
- Collision avoidance (performed in each frame): Compute a velocity $\mathbf{v}_{\text{new}}$ close to $\mathbf{v}_{\text{pref}}$ that avoids nearby collisions.

This framework is not restricted to specific algorithms per task. Our implementation uses the data structures and algorithms listed in Appendix A.

It is common to let an agent re-plan its path whenever the path-following algorithm cannot compute an attraction point $\mathbf{p}_{\text{att}}$, i.e. when it no longer knows how to follow the current path [18]. We will do this in our implementation as well.

5.2. Strategic level

In the standard framework with three tasks, agents always accept the choices made by each algorithm. We propose to add a fourth task –the *strategic level*– in which the agent analyzes the strategies of each task and responds to conflicts. For now, we focus on *obstacle*-related conflicts between path following and collision avoidance. In Section 6, we will present a generalized framework that supports other types of conflicts.

The strategic level first converts the agent's preferred velocity $\mathbf{v}_{\text{pref}}$ to a strategy S_F, and the collision-avoidance velocity $\mathbf{v}_{\text{new}}$ to a strategy S_C, using the method from Section 4.2. We consider only obstacles for S_F (because agents are not yet relevant at the path-following level), and we consider both obstacles and agents for S_C. For the time window of these conversions, let τ_{att} be $\frac{\|\mathbf{p}_{\text{att}}-\mathbf{p}\|}{s_{\text{pref}}}$, i.e. the time required to reach the attraction point at the preferred speed. Let $TTC(\mathbf{v}'')$ be the time to the first collision with obstacles when using a given velocity $\mathbf{v}''$. We use a time window $\tau = \min(\tau_{\text{att}}, TTC(\mathbf{v}_{\text{pref}}))$.

Next, we check if $S_C \neq_c S_F$. When this happens, collision avoidance and path following disagree on how to pass one or more obstacles. If the agent uses $\mathbf{v}_{\text{new}}$ for τ seconds, it will move around these obstacles differently than prescribed by $\mathbf{v}_{\text{pref}}$. We compute two ways to resolve the conflict: an *alternative path* π' that satisfies S_C (instead of S_F), and an *alternative velocity* $\mathbf{v}'_{\text{new}}$ that satisfies S_F (instead of S_C). Section 5.3 will describe how to compute π' and $\mathbf{v}'_{\text{new}}$.

When π' and $\mathbf{v}'_{\text{new}}$ have been computed, the agent can choose one of the two, or it can decide to ignore the conflict and use the velocity $\mathbf{v}_{\text{new}}$ that it already had in mind. There are many ways to let an agent choose between these options. We suggest a simple model with two parameters: a maximum detour factor W and a Boolean flag *Strict*. The agent decides as follows:

1. Use π' if it exists and if it is $\leq W$ times longer than the remainder of π. We define this remainder as the line segment from $\mathbf{p}$ to $\mathbf{p}_{\text{att}}$, plus the subpath of π from $\mathbf{p}_{\text{att}}$ to the goal.
2. Otherwise, if *Strict* = `True`, use $\mathbf{v}'_{\text{new}}$.
3. Otherwise, use $\mathbf{v}_{\text{new}}$ (and thus ignore the conflict).

As mentioned in Section 1, global (re-)planning is a relatively costly operation. By planning an alternative path π' only in case of a conflict, we avoid re-planning when it is likely that the current path can still be followed. Still, it can be useful to limit the frequency at which the agent re-plans. This will reduce computation times even further, and it will prevent the agent from becoming too indecisive. Therefore, we introduce the *re-planning time* T_R: an agent only looks for an alternative path π' when its last path update was at least T_R seconds ago. In any frame where this is not the case, we consider π' to be non-existent.

5.3. Computing an alternative path and velocity

The strategic level computes $\mathbf{v}'_{\text{new}}$ by performing collision avoidance constrained by S_F, and π' by performing path planning constrained by S_C. We now explain generally how to compute a path or a velocity under the topological constraints of another strategy. Let the *constraint strategy* S_T be a strategy that contains all constraints of our interest.

To plan a *global path* whose strategy $S_{G'}$ is consistent with S_T, we plan a path on the navigation mesh $\mathcal{M}$ in the usual way, but with the additional rule that we cannot traverse any passages that would lead to a conflict between $S_{G'}$ and S_T. For any passage p_i that we encounter during the search, traversing p_i would imply $D_{G'}(A_j, O_{i,l}) = \mathtt{R}$ and $D_{G'}(A_j, O_{i,r}) = \mathtt{L}$. Thus, if $D_T(A_j, O_{i,l}) = \mathtt{L}$ or $D_T(A_j, O_{i,r}) = \mathtt{R}$, we do not search further in this direction, as traversing p_i would imply $S_{G'} \neq_c S_T$. (Technically, $D_{G'}(A_j, O_{i,l})$ could be cancelled out to `X` elsewhere on the path, but in this case, we assume that it is undesirable to pass an obstacle on the wrong side at least once.)

In our previous publication [7], we added more constraints to global re-planning to prevent the agent from backtracking. This turned out to be unnecessary for all our scenarios, so we will now omit it from our discussion.

To perform *collision avoidance* such that the resulting strategy $S_{C'}$ is consistent with S_T, we should disallow the selection of any velocity that would lead to conflicts. If the collision-avoidance algorithm uses sampling (i.e. choosing the best velocity out of several candidates), then S_T is easy to incorporate. For each candidate velocity $\mathbf{v}''$, we compute the navigation strategy $S_{C''}$ and we discard $\mathbf{v}''$ if $S_{C''} \neq_c S_T$. We expect that a similar adaptation can be made for other collision-avoidance algorithms, such as those based on velocity obstacles [20].

6. Generalization of the framework

We now generalize Section 5 to an abstract framework in which agents use an arbitrary number of navigation algorithms.

6.1. Hierarchy of algorithms

Generally, an agent in a crowd simulation has a sequence of l navigation algorithms $\{L_i\}_0^{l-1}$ at its disposal. These algorithms are ordered hierarchically: each L_i operates at its own 'level of detail'. Algorithms with a higher index are typically performed more frequently than those with a lower index. As an example, our full implementation (described in Appendix A) uses path planning for L_0, path following for L_1, a velocity-adjusting mid-term algorithm for L_2, and collision avoidance for L_3. Path planning is performed once; the other algorithms are performed in each frame of the simulation loop. If a manoeuvre-based mid-term planning algorithm were to be added to this framework, it would lie between path planning and path following, and it would be performed about once per second [4].

For convenience, we will refer to the output of an algorithm L_i as a *plan* P_i. A plan can (for example) be a path, a velocity, or a sequence of manoeuvres around agents. In our implementation, P_0 is a path π, P_1 is a preferred velocity $\mathbf{v}_{\text{pref}}$, P_2 is an adjusted preferred velocity $\mathbf{v}_{\text{adj}}$, and P_3 is a new velocity $\mathbf{v}_{\text{new}}$ that the agent ends up using.

6.2. Generalized strategic level

The strategic level from Section 5.2 can now be generalized as follows. Instead of having a single strategic level, agents may check for conflicts at every level of detail. Each time an agent has executed an algorithm L_i to compute a plan P_i, it performs the following steps:

1. Convert the newly computed plan P_i to a strategy S_i.
2. If desired (and if $i > 0$), check for conflicts with the 'parent' strategy S_{i-1} imposed by algorithm L_{i-1}. (The designer of the simulation should decide at which levels this conflict detection is desired. We will discuss several options in Section 6.2.2.) If there are no conflicts, continue the simulation as usual. Otherwise, do the following:
   2a. Let $k(i) \in \{0 \dots i-1\}$ be a pre-defined higher level in the hierarchy. If $P_{k(i)}$ is allowed to change at the current moment in time, try to compute an alternative plan $P'_{k(i)}$ constrained by S_i. (This is a generalization of the alternative path π' from Section 5.2.)
   2b. Compute an alternative plan P'_i constrained by S_{i-1}. (This is a generalization of the alternative velocity $\mathbf{v}'_{\text{new}}$ from Section 5.2.)
   2c. According to certain selection criteria, choose between using $P'_{k(i)}$, using P'_i, or ignoring the conflict.

6.2.1. Requirements

For step 1, it is important that any plan P_i can be converted to a strategy S_i. Section 4 already explained the conversion process for paths and velocities. The only type of output that we have not yet considered is a sequence of manoeuvres computed by a mid-term algorithm [4, 29]. However, in these cases, the translation to a strategy is trivial: after all, these algorithms explicitly compute how to move around obstacles and agents.

For step 2, one must also define how to compute a plan P_i constrained by an arbitrary strategy. Section 5.3 explained the details for path planning and collision avoidance. For path following, we can disallow attraction points that would yield a conflict with the constraint strategy. For manoeuvre-based mid-term navigation, it is easy to disallow specific manoeuvres around obstacles or agents.

6.2.2. Suggestions for details

In step 2, one must define for each level $i > 0$ if conflict detection is desired. Our implementation allows conflict detection at the velocity-adjustment and collision-avoidance levels. To encode this, we use two Boolean flags $Check_A$ and $Check_C$.

In step 2a, one must define $k(i)$ for each $i > 0$. That is, for each navigation algorithm, one must define at which 'higher' navigation algorithm the agent should try to find an alternative plan. In our implementation, we use $k(1) = k(2) = k(3) = 0$, so the agent always considers re-planning its global path.

Also in step 2a, one can limit the frequency at which a plan is allowed to change. One option is to introduce parameters $T_{R,i}$ that denote the minimum required time between two changes in P_i. This is a generalization of T_R from Section 5.2. In our implementation, we only need a single parameter T_R for global planning, because step 2a always considers the global path.

For step 2c, many choice models are possible again. In line with Section 5.2, one option is to accept $P'_{k(i)}$ based on a measure of attractiveness, and to (otherwise) accept P'_i based on a Boolean flag. In our implementation, we accept π' based on a maximum detour length W. We use two Boolean flags $Strict_A$ and $Strict_C$ to set the agent's strictness at the velocity-adjustment and collision-avoidance levels.

The conversion of a velocity to a strategy requires a *time window* parameter. In our implementation, we always use the time window suggested by path following, as explained in Section 5.2. However, if manoeuvre-based mid-term planning is included, the time window for collision avoidance should most likely be reduced.

Whenever an agent switches to an alternative plan, this may cause new conflicts with other algorithms in the hierarchy. In preliminary experiments, we have tried to resolve these new conflicts recursively. We have observed that this can cause significant computational overhead without noticeably improving the agent's behavior. Instead, it is easier to wait until the next simulation frame to see if the conflicts remain.

7. Experiments and results

We have implemented our strategy-driven simulation framework in C++. All relevant settings and implementation details can be found in Appendix A. We perform all experiments on a Windows 10 device with an Intel Core i7-7920HQ CPU. In our experiments, we will consider the following variants of the simulation framework:

- *CA-Only*: agents perform no path planning or path following, and their $\mathbf{v}_{\text{pref}}$ always points straight to the goal.
- *Standard*: agents perform all navigation levels *except* velocity adjustment. They compute strategies, but they never detect or resolve conflicts: $Check_A = Check_C =$ `False`.
- *Standard+2.0, Standard+0.5*: variants of *Standard* with $Check_C =$ `True`, so agents respond to strategic conflicts between $\mathbf{v}_{\text{new}}$ and $\mathbf{v}_{\text{adj}}$. Other settings are $W = \infty$, $Strict_C =$ `False`, and either $T_R = 2.0$ or $T_R = 0.5$s.
- *Standard+0.1+Strict*: a variant of the above with $T_R = 0.1$s and $Strict_C =$ `True`. That is, an agent may re-plan its global path in each frame, and it follows its path strictly.
- *Streams*: agents perform all navigation levels *including* velocity adjustment. They compute strategies, but they never detect or resolve conflicts: $Check_A = Check_C =$ `False`.
- *Streams+2.0, Streams+0.5*: variants of *Streams* with $Check_A =$ `True`, so agents respond to strategic conflicts between $\mathbf{v}_{\text{adj}}$ and $\mathbf{v}_{\text{pref}}$. Other settings are $W = \infty$, $Strict_A =$ `False`, and either $T_R = 2.0$s or $T_R = 0.5$s.
- *Streams+0.1+Strict*: a variant of the above with $T_R = 0.1$s and $Strict_A =$ `True`.
- *Complete+Strict*: a variant of *Streams+0.1+Strict* that also has $Check_C =$ `True` and $Strict_C =$ `True`. That is, agents respond to conflicts at two levels.

In total, we test 10 simulation variants, as opposed to 4 variants in our original publication [7]. We will refer to the strategy-enhanced simulations collectively as *Standard+* and *Streams+*.

The main purpose of our experiments is to show how strategies and conflict resolution can improve the behavior of agents

compared to a traditional simulation. We will show this via several scenarios, and we will discuss the results per scenario. Section 8 will discuss the advantages and limitations of our method in general. To see all scenarios in motion, we encourage the reader to watch this paper's *supplementary video*.

7.1. Single-agent demonstration

To demonstrate the advantages of conflict resolution, we first show an agent navigating through an environment with obstacles and other agents. Figure 6(a) shows our example scenario. The red agent needs to move from the top left to the top right, and its initial path (computed by global path planning) runs through several problematic areas. Some passages are blocked by stationary agents (in purple and green), and the lower part contains agents flowing to the left (in blue) and to the right (in orange). These 'other' agents use the *CA-Only* simulation variant.

Figure 6(b) shows the trajectories for the red agent when using different simulation variants. When the agent uses *CA-Only* (shown in pink), it gets trapped behind the first wall on the straight line to the goal. When it uses *Standard* (shown in brown), it gets stuck at the opening blocked by purple agents.

The red trajectory is the result for *Standard+2.0*. With these settings, the global plan changes at most every two seconds, the agent accepts detours of any length, and it is not forced to follow its path locally. This configuration allows the agent to reach its goal. Figure 6(c) shows where the agent changes its global plan. In cases 1 and 4, the agent chooses a detour because collision avoidance recognizes that an area is blocked. In cases 2 and 3, the agent switches between different openings and eventually chooses the lowest one, to join the orange agents who are moving in the same direction.

Note that this is merely meant as a demonstration of how the strategy-based simulation *could* be used. Many variations are possible. For example, if we used a small value of W, the agent would consider the last detour to be too large, and it would (deliberately) keep trying to use the passage with the green agents. This may be the desired behavior in certain scenarios.

7.2. Crowd flows with small obstacles

To show what happens when many agents use a certain simulation variant, we simulate unidirectional and bidirectional crowd flows in a U-turn corridor that contains several small obstacles (*UTurn*, 40×24 m). We compute start and goal positions via uniform random sampling at the corridor's far ends. We insert 3 agents per second, and we remove an agent when it lies within 0.5m of its goal. To make the crowd more diverse, we give each agent a (uniformly sampled) random preferred speed s_{pref} between 1.2 and 1.4m/s.

In every simulation variant, we measure the total number of agents (*NrAgents*) that are added after $t = 50$s and that reach their goal before $t = 300$s. For each such agent, we measure the distance travelled, the total travel time, the average speed, and the time spent walking more slowly than 0.5m/s. Of these metrics, we report the average and standard deviation over all agents (*Dist*, *Time*, *AvgSpeed*, *SlowTime*). These are indicators of how efficiently the crowd moves. We exclude the first 50s to prevent distorting the results with a quiet 'warm-up' period.

7.2.1. Results for unidirectional flows

We first discuss the results for the unidirectional scenario. Figure 7 shows screenshots for a number of simulation variants; Table 1 provides quantitative results.

The *CA-Only* simulation variant (Figure 7(a)) clearly fails in this corridor because global planning is required. In the *Standard* variant (Figure 7(b)), all agents follow roughly the same path, which causes some of them to slow down around the first obstacle. This is reflected by high averages and standard deviations for *Time* and *SlowTime*. Also, parts of the corridor are under-utilized because they are not chosen by global planning. Agents only switch to an alternative path when their path-following algorithm fails. This usually occurs quite late, and it is a result of 'luck' rather than a deliberate change of strategy.

By contrast, in the *Standard+* variants (such as in Figure 7(c)), agents can decide to move around obstacles differently when their collision-avoidance algorithm suggests this. This causes the agents to make better use of alternative paths and spread out over the corridor. Consequently, the agents walk faster on average, and they spend less time walking slowly. In all versions of *Standard+*, the *Time*, *AvgSpeed*, and *SlowTime* metrics are significantly different from those of *Standard* ($p < 0.001$ using a two-tailed Student's t-test).

This experiment shows that it can be useful to let agents take global detours based on their local velocities. Furthermore, we observe only minor differences *among* the *Standard+* variants, both quantitative and visually. This suggests that frequent re-planning is not strictly necessary, which is good news for the simulation's computation time.

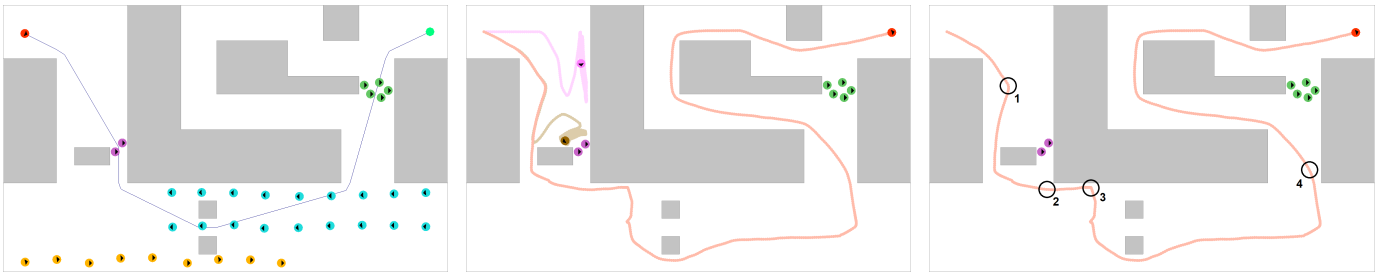
The addition of a velocity-adjustment algorithm (in *Streams* and *Streams+*) decreases the quality of the simulation. This algorithm averages out velocities of nearby agents that are moving in the same direction. In our case, this also happens around small obstacles, which makes agents less decisive about how to pass these obstacles. This explains why the results for the *Streams* simulation variant are worse than those for *Standard*.

However, the *Streams+* variants improve upon *Streams* significantly again. In these simulations, agents recognize when velocity adjustment tries to re-route them around an obstacle, and they translate this to a new global path. This shows that our strategy concept also works on the velocity-adjustment level, even though the velocity-adjustment algorithm itself is not ideal for this scenario. (Among all simulations with velocity adjustment, the *Complete+Strict* variant is the only one that performs significantly better than *Standard*.)

7.2.2. Results for bidirectional flows

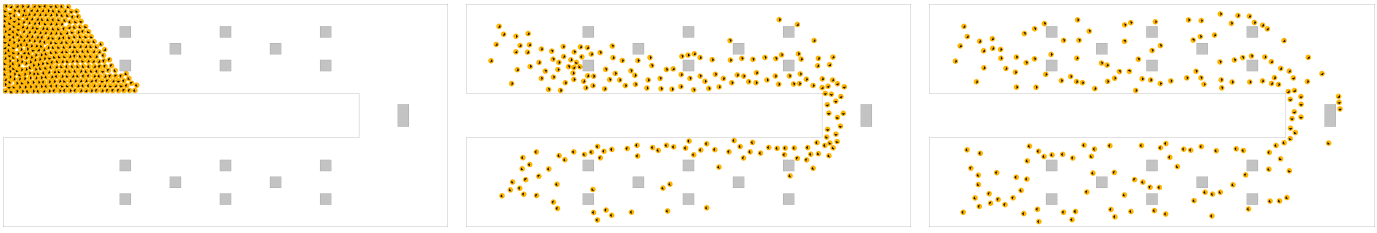
For the bidirectional version of this experiment, screenshots and metrics can be found in Figure 8 and Table 2, respectively.

In the *Standard* simulation variant (Figure 8(a)), a congestion occurs in the middle of the corridor where all agents attempt to use the same area. As shown in Table 2, only a few agents reach the goal, and even those move rather slowly. In the *Standard+* variants (such as in Figure 8(a)), this deadlock is avoided because agents automatically take alternative paths. Again, we do not observe large differences among the *Standard+* simulations. Table 2 reports the best values for the *Standard+0.1+Strict* vari-



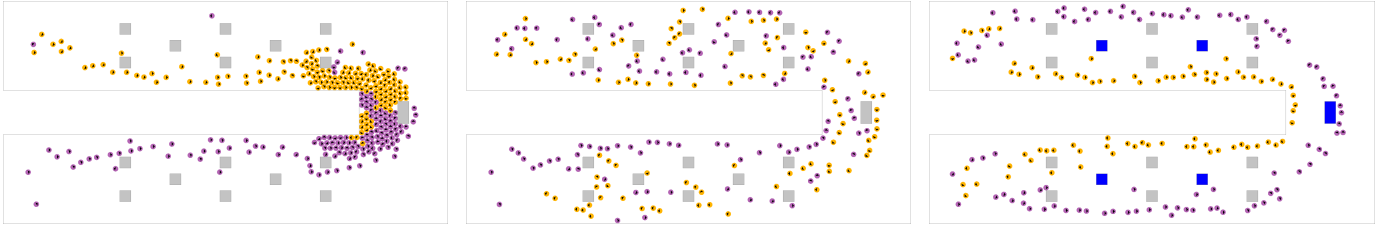
(a) Overview (b) Trajectories (c) Re-planning actions for *Standard+2.0*

Fig. 6. Single-agent demonstration. (a) An agent is instructed to move from the top left to the top right. The curve is its initial global path. (b) The agent's trajectory using *CA-Only* (pink), *Standard* (brown), and *Standard*+2.0 (red). (c) The *Standard*+2.0 agent changes its global path on several occasions, indicated by the black circles.



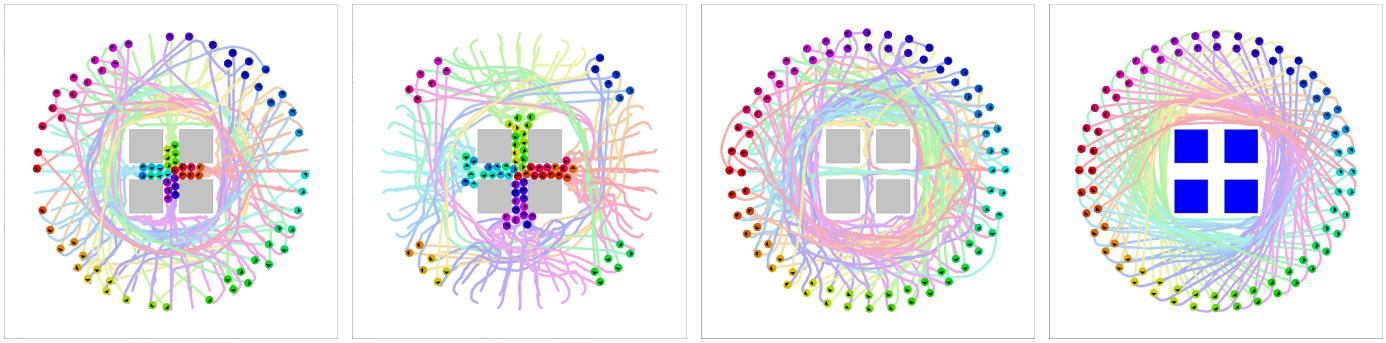
(a) *CA-Only* (b) *Standard* (c) *Standard*+2.0

Fig. 7. Screenshots of 1-directional crowd flows in the *UTurn* environment, with different simulation variants. Agents are instructed to move from the top-left to the bottom-left. The screenshots were taken after 120 simulated seconds.



(a) *Standard* (b) *Standard*+2.0 (c) *Standard* with environmental guidance

Fig. 8. Screenshots of 2-directional crowd flows in the *UTurn* environment, with different simulation variants. The orange agents move from the top-left to the bottom-left; the purple agents move in the opposite direction. The screenshots were taken after 120 simulated seconds.



(a) *CA-Only* (b) *Standard* (c) *Standard+0.5* (d) *Standard* with env. guidance

Fig. 9. Screenshots of the *Circle* scenario with different simulation variants. Trajectories are shown in different colors per agent, and the screenshots were taken when the last non-blocked agent reached its goal.

ant, but the standard deviations among agents are generally high. (Significance tests between *Standard+* and *Standard* are not very meaningful in this case, due to the congestion in the *Standard* variant.)

Just like in the unidirectional case, the results for *Streams* and *Streams+* are not as good. These simulation variants all cause deadlocks at a certain moment. However, in every version of *Streams+*, these deadlock occurs substantially later than in the regular *Streams* variant. This is reflected by higher values for the *NrAgents* metric in Table 2. Thus, the use of strategies does improve the simulation again. On the other hand, it does not fully compensate for the poor performance of velocity adjustment in this environment.

7.2.3. A note on density-based planning

Density-based global path planning [26] is another way to distribute agents over different global paths. Such a method uses density information to change the attractiveness of areas in the navigation mesh. However, this typically does not recognize small congestions in large areas, and it does not consider the *directions* in which agents move. By contrast, our method uses the result of collision avoidance to trigger re-planning. This is purely based on the agent's local observations. We expect that our method could be *combined* with density-based path planning to further improve the crowd's behavior.

7.3. Circle scenario with obstacles

As another example, we consider the scenario in Figure 9 in which 80 agents need to move to the opposite end of a circle with an inner diameter of 15m, while avoiding four 2×2m obstacles in the middle. A circle scenario *without* obstacles is commonly used for showcasing a collision-avoidance method. The addition of obstacles makes the scenario more complicated, and it enables the use of global planning and strategies.

When using the *CA-Only* simulation variant (Figure 9(a)), all agents greedily move towards their goal without explicitly navigating around the obstacles. As a result, some of the agents get stuck in the middle. In the *Standard* simulation (Figure 9(b)), the problem is worse: more agents now explicitly choose to go through the narrow passage, as this corresponds to the shortest path in the underlying navigation graph. The *Streams* simulation yields similar results.

In all simulations enhanced with conflict resolution (i.e. *Standard+*, *Streams+*, and *Complete+Strict*), every agent reaches its goal. An example is shown in Figure 9(c). The first agents to arrive at the obstacles will try to pass through the middle. This causes a congestion, but a smaller one than before. The later agents realize on time that collision avoidance attempts to send them around the obstacles. This information is translated to a global detour. Furthermore, the agents in the middle can now explicitly decide to *go back* as soon as collision avoidance suggests this. Eventually, this allows the congestion to get resolved. (Just as in the *UTurn* scenarios, we observed no meaningful differences among the *Standard+* or *Streams+* variants.)

Note that even with our enhancements, agents do not *anticipate* the congestion before it can occur, and there is no active *collaboration* among agents. This is a logical outcome because our implemented simulation framework simply does not contain algorithms for such purposes. However, this paper shows that the use of strategies can already make agents more *responsive* and improve the crowd's behavior in complicated situations. We expect that additional navigation algorithms for specific purposes can improve the results even further.

7.4. Environmental guidance

A useful 'bonus' application of topological strategies is that we can now *guide* agents around obstacles in specific ways. As a simple example, we can give obstacles an optional label indicating a specific navigation decision for agents. Whenever an agent (re-)plans its global path, we include these decisions in the constraint strategy. The real-world analogy of this is to place directional signs all around particular objects or walls.

This *environmental guidance* can be used to regulate the crowd flow in challenging scenarios. Figures 8(c) and 9(d) show two examples: respectively, the bidirectional *UTurn* scenario and the *Circle* scenario. Here, we instruct agents to pass the blue-colored obstacles on the right during global path planning. The result is a well-organized crowd flow in which the agents avoid difficult situations.

For completeness, Table 3 provides quantitative results for the bidirectional *UTurn* simulations with guidance. In this case, the *Standard* simulation performed best (although the differences with *Standard+* were small). This suggests that when the environment already tries to guide agents, extra features such as conflict resolution and velocity adjustment might be redundant. Finding the right balance between high-level guidance and agent autonomy is challenging and application-specific.

Navigation strategies provide an intuitive way to enforce decisions in the crowd. As said, this type of environmental guidance is comparable to placing signage in a real-world environment to improve pedestrian flows. This analogy with signage can be taken further. We will mention several options in Section 9 as suggestions for future work.

7.5. Performance analysis

Finally, we discuss the computation times of our system in the *UTurn* simulations. Our discussion will focus on the unidirectional scenario; we have observed similar results for the bidirectional simulations (both with and without guidance).

As described in Appendix A.1, each simulation frame is subdivided into 12 *substeps* that correspond to a particular task. In our previous publication, we reported the average running time (over all frames) of each substep. However, these results are highly influenced by the number of agents in the environment. In this extended paper, we would like to obtain more insight in the computational cost *per agent*. We therefore do the following: for each frame of 0.1s between $t = 50$s and $t = 300$s, we measure the running time of each substep and divide it by the number of agents in the environment during that frame. Per frame, this gives us 12 values indicating the average time spent *per substep per agent*. Finally, we compute the averages of these values over all frames. Figure 10 plots our results.

Table 1. Quantitative results of the 1-directional *UTurn* simulations from Section 7.2.1. Standard deviations are shown in square brackets. A gray name indicates that a deadlock occurred in the simulation. An asterisk indicates that a metric of a strategy-driven simulation is significantly different ($p < 0.001$, two-tailed Student's t-test) from its non-strategic counterpart. (For *Complete+Strict*, we compare to the *Standard* simulation.)

Simulation variant	*NrAgents*	*Dist*	*Time*	*AvgSpeed*	*SlowTime*
CA-Only	0	-	-	-	-
Standard	516	72.29 [3.43]	73.19 [7.98]	1.00 [0.11]	3.73 [5.44]
Standard+2.0	568	72.64 [4.92]	* 60.02 [3.54]	* 1.21 [0.06]	* 0.10 [0.26]
Standard+0.5	570	72.82 [5.13]	* 60.20 [3.49]	* 1.21 [0.07]	* 0.11 [0.30]
Standard+0.1+Strict	568	72.84 [5.06]	* 60.06 [3.61]	* 1.21 [0.06]	* 0.14 [0.36]
Streams	463	73.59 [4.74]	89.82 [15.35]	0.84 [0.14]	17.52 [15.58]
Streams+2.0	514	72.82 [4.89]	* 78.68 [7.18]	* 0.94 [0.12]	* 4.85 [5.20]
Streams+0.5	505	72.81 [4.90]	* 78.32 [6.70]	* 0.94 [0.12]	* 4.23 [4.68]
Streams+0.1+Strict	510	72.76 [4.88]	* 78.63 [7.06]	* 0.94 [0.12]	* 4.37 [4.97]
Complete+Strict	556	73.28 [4.86]	* 65.33 [2.65]	* 1.12 [0.08]	* 0.28 [0.63]

Table 2. Quantitative results of the 2-directional *UTurn* simulations from Section 7.2.2. Standard deviations are shown in square brackets. A gray name indicates that a deadlock occurred in the corresponding simulation.

Simulation variant	*NrAgents*	*Dist*	*Time*	*AvgSpeed*	*SlowTime*
CA-Only	0	-	-	-	-
Standard	37	85.13 [5.02]	81.81 [22.48]	1.09 [0.18]	13.30 [20.35]
Standard+2.0	532	82.84 [8.76]	72.22 [9.00]	1.15 [0.06]	1.00 [1.16]
Standard+0.5	535	81.62 [8.71]	70.81 [8.57]	1.16 [0.06]	0.83 [1.29]
Standard+0.1+Strict	542	80.79 [8.16]	69.78 [7.54]	1.16 [0.06]	0.89 [1.74]
Streams	26	83.89 [6.87]	107.18 [20.99]	0.80 [0.10]	19.67 [20.29]
Streams+2.0	165	84.02 [3.38]	98.81 [15.72]	0.86 [0.10]	16.12 [14.52]
Streams+0.5	175	82.98 [4.03]	98.03 [9.82]	0.85 [0.07]	12.56 [9.07]
Streams+0.1+Strict	119	83.30 [3.76]	95.94 [12.52]	0.88 [0.08]	12.07 [11.00]
Complete+Strict	415	82.03 [7.83]	100.52 [13.24]	0.82 [0.08]	12.98 [8.28]

Table 3. Quantitative results of the 2-directional *UTurn* simulations with guidance, as described in Section 7.4. Standard deviations are shown in square brackets. A gray name indicates that a deadlock occurred in the corresponding simulation.

Simulation variant	*NrAgents*	*Dist*	*Time*	*AvgSpeed*	*SlowTime*
CA-Only	0	-	-	-	-
Standard	569	73.58 [6.74]	59.69 [5.80]	1.23 [0.04]	0.23 [0.49]
Standard+2.0	563	76.03 [7.33]	62.03 [6.57]	1.23 [0.05]	0.18 [0.34]
Standard+0.5	568	75.82 [7.28]	61.71 [6.45]	1.23 [0.05]	0.20 [0.88]
Standard+0.1+Strict	561	75.89 [8.83]	61.88 [7.76]	1.23 [0.05]	0.17 [0.47]
Streams	520	74.45 [6.40]	74.24 [4.62]	1.00 [0.08]	2.88 [3.25]
Streams+2.0	523	74.38 [6.11]	74.39 [4.12]	1.00 [0.08]	2.77 [3.42]
Streams+0.5	525	74.35 [6.24]	73.60 [4.10]	1.01 [0.08]	2.70 [3.20]
Streams+0.1+Strict	529	74.03 [6.06]	72.83 [3.84]	1.02 [0.08]	2.51 [3.19]
Complete+Strict	508	75.53 [7.40]	76.68 [6.46]	0.99 [0.06]	2.86 [3.04]

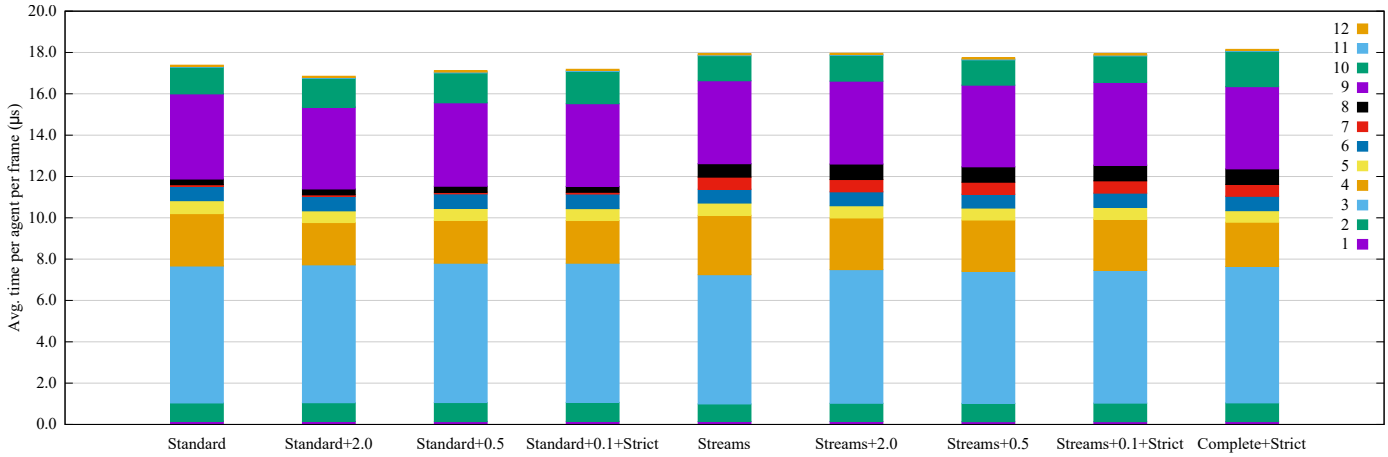


Fig. 10. Stack plot of the average performance of all simulation substeps, per agent per frame, in the 1-directional *UTurn* simulations. The substeps are stacked from bottom to top. The numbered labels correspond to the substep numbers described in Appendix A.1.

7.5.1. Performance of substeps

In all simulations, computing visibility polygons (substep 3) is the most computationally expensive substep, followed by collision avoidance (substep 9) and computing neighboring objects (substep 4). These three substeps make up 70% of the overall computation time in the *Complete+Strict* simulation, and an even larger percentage in the other variants.

All strategy-related tasks are performed in substeps 6, 8, and 10. Substep 10 includes computing a strategy S_C from the agent's new velocity $\mathbf{v}_{\text{new}}$. It is the fourth most expensive substep in all simulation variants, even in *Standard* and *Streams* where the result S_C is not used. As can be seen in Figure 10, the cost of this substep increases only slightly in the variants that *do* use the strategy S_C. This means that conflict detection and global re-planning have only a very small impact on the overall running time. By letting agents re-plan *only* in case of a conflict, we can keep the simulation computationally efficient. Smaller values of the re-planning time T_R imply more potential re-planning, but the influence of this parameter on the overall performance is minimal.

7.5.2. Analysis of real-time performance

Overall, our simulations roughly take 3.0 to 4.5 milliseconds per frame, and they contain between 150 and 200 agents at the same time. Thus, the simulations easily run in real-time.

For a more detailed analysis, we consider the average *total* computation time per agent per frame. This is the sum of all substep times discussed earlier, and it corresponds to the total height of a stack in Figure 10. Among the simulation variants, this value varies between 16.85μs (for *Standard+2.0*) and 18.15μs (for *Complete+Strict*). It is logical that these simulation variants are the two extremes: *Complete+Strict* performs the most tasks per agent, and *Standard+2.0* is the simplest simulation variant that yields a smooth crowd flow without overcrowded areas.

Given that each frame simulates 0.1 seconds (i.e. 100,000μs), extrapolation suggests that our program can simulate between $5,500$ and $6,000$ agents in real-time on the machine we have used. Note that most substeps can trivially be parallelized over agents; for our experiments, we have used 4 parallel threads.

Naturally, this extrapolation is optimistic. For example, the kd-tree construction time does not scale linearly with the number of agents. Also, many substeps depend on the number of objects around each agent, which in turn depends on the crowd density and on the environment's geometry. In particular, visibility queries (substep 3) will take more time in environments with many detailed obstacles, and nearest-neighbor queries (substep 4) will be more costly in case of bottlenecks and congestions.

In short, the exact performance will be different in each scenario, and it is difficult to make general claims about performance based on our current experiments. For this paper, the key observation is that the strategy-related substeps (6, 8, and 10) do not have a large impact on the overall running times. In other words, the concepts presented in this paper can be added to an existing simulation without harming its real-time performance.

8. Discussion

Our method offers a way to coordinate between the different levels of agent navigation. Still, the overall 'intelligence' of agents depends largely on the quality of the navigation algorithms themselves. There are several types of algorithms that we have not yet considered in our experiments, such as manoeuvre-based mid-term planning [4], detailed navigation in high-density crowds [29], and density-based adaptation of the preferred velocity [5]. We could also consider adding algorithms for coordination *among* agents [34, 35], either as a replacement for global planning or as an additional level of navigation. We expect that all these algorithms will benefit from conflict detection, but future work will have to demonstrate this.

The crowd's behavior also depends on the parameter settings within each algorithm. It can even be desirable to vary the settings per agent for the sake of variety. Crowd simulations are highly complex systems where small changes can have large and unpredictable consequences. Thus, it is difficult to draw general conclusions from any experiment. Our results at least indicate that topological strategies *can* improve a simulation without having to change any other simulation settings. However, as is usual in this research area, each scenario may require careful visual inspection and parameter tuning.

On a related note, we can now detect strategic conflicts and find possible solutions, but choosing the right solution at the right time is difficult and highly scenario-dependent. Our suggested choice model is only one of many possibilities. Even within this model, each situation may impose its own optimal parameter settings.

Also, it remains the case that each navigation algorithm uses different information to determine its plan. As a result, agents could end up switching back and forth between strategies, if one algorithm keeps 'forgetting' the information that was used by another algorithm. Preventing such problems requires more advanced rules and memory models for the individual agents.

We have not yet considered scenarios where agents can choose between *multiple goals*, such as an evacuation with multiple ways to leave a building. This can cause situations where an agent may not only update its path, but also change its goal. This is easily compatible with our method if we extend global path planning to use multiple possible goals. An agent will then automatically choose another goal when nearby obstacle constraints make it more attractive.

Finally, a largely open research question is how to measure the *realism* of a crowd simulation. Our method can solve navigation problems that are easy to see in a top-down view, and it results in trajectories that are objectively more efficient. However, in complex scenarios with obstacles, it is unclear how real humans behave, how to translate this to simulation settings, and how to properly compare a simulation to reality. Overall, the research area is not yet ready to make any general claims about how 'human-like' a simulation is.

9. Conclusions and future work

We have presented a method for improving the navigation behavior of autonomous agents in virtual environments. We have

defined a *navigation strategy* as a set of decisions to pass obstacles and agents on certain sides. Such a strategy can be obtained from the result of global navigation (path planning) *and* of local navigation (path following and collision avoidance). With this uniform definition, we can detect and resolve conflicts between the strategies produced by different navigation algorithms. For example, when path following and collision avoidance disagree on how to pass an obstacle, an agent can attempt to re-plan its global path, using its local strategy as guidance.

This concept can easily be integrated into an existing crowd-simulation framework, by letting agents repeatedly compare the strategies produced by their algorithms. We have presented an abstract scheme for this, which can be applied to many algorithms and frameworks. Our experiments with an example implementation show that strategies and conflict resolution can improve the behavior of agents in real time. The strategy concept is also suitable for explicitly sending agents in certain directions, e.g. to simulate the effect of directional signs.

As mentioned earlier, our simulation framework can be enhanced with many more navigation algorithms. Eventually, we envision a simulation in which agents can use different navigation algorithms for different occasions. The generic concept of a strategy makes it easier to detect when two algorithms behave inconsistently, or when certain algorithms are not sufficient for solving a problem. This may lead to a system that can automatically choose the best algorithms for a given situation.

To plan an alternative path, we currently run the A* search from scratch. We can improve efficiency by re-using parts of the old path, or perhaps by using a *hierarchical* approach in which the agent starts with a coarse path and refines it during the simulation. The agent will then (automatically) also refine its *strategy* over time.

Our experiment with environmental guidance (Section 7.4) suggests another promising area for future work: simulating how people respond to signage in their environment. For instance, we could replace the 'obstacle labels' of Section 7.4 by 'sign' objects at specific positions, and then update an agent's constraint strategy only when it sees the corresponding sign. This could then be used to study the effectiveness of having signs at particular positions. Of course, there has been ample previous work on crowd simulation with signage, e.g. [36, 37]. Our method could help make such simulations more efficient and less dependent on specific navigation algorithms.

Finally, we consider this work to be a first step towards looking at agent-based crowd simulation in a fundamentally different way. By modelling agent navigation in terms of topological decisions, we can bridge conceptual gaps between algorithms. On the long term, we hope that this insight will lead to a fully hybrid simulation technique, in which all aspects of agent navigation are merged into one process.

References

[1] Pelechano, N, Allbeck, JM, Kapadia, M, Badler, NI. Simulating Heterogeneous Crowds with Interactive Behaviors. CRC Press; 2016.

[2] Thalmann, D, Musse, SR. Crowd Simulation. 2 ed.; Springer; 2013.

[3] van Toll, W, Jaklin, N, Geraerts, R. Towards believable crowds: A generic multi-level framework for agent navigation. In: ASCI.OPEN. 2015,.

[4] Bruneau, J, Pettré, J. EACS: Effective Avoidance Combination Strategy. Comput Graph Forum 2017;36(8):108–122.

[5] Best, A, Narang, S, Curtis, S, Manocha, D. DenseSense: Interactive crowd simulation using density-dependent filters. In: Proc. 13th ACM SIGGRAPH / Eurographics Symp. Computer Animation. 2014, p. 97–102.

[6] van Goethem, A, Jaklin, N, Cook IV, AF, Geraerts, R. On streams and incentives: A synthesis of individual and collective crowd motion. In: Proc. 28th Conf. Computer Animation and Social Agents. 2015,.

[7] van Toll, W, Pettré, J. Connecting global and local agent navigation via topology. In: Proc. 12th ACM SIGGRAPH Int. Conf. Motion in Games. 2019, p. 33:1–33:10.

[8] Pelechano, N, Allbeck, JM, Badler, NI. Controlling individual agents in high-density crowd simulation. In: Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation. 2007, p. 99–108.

[9] Kielar, PM, Biedermann, DH, Borrmann, A. MomenTUMv2: A modular, extensible, and generic agent-based pedestrian behavior simulation framework. Tech. Rep. TUM-I1643; Technische Universität München, Institut für Informatik; 2016.

[10] Curtis, S, Best, A, Manocha, D. Menge: A modular framework for simulating crowd movement. Collective Dynamics 2016;1(A1):1–40.

[11] Kapadia, M, Beacco, A, Garcia, F, Reddy, V, Pelechano, N, Badler, NI. Multi-domain real-time planning in dynamic environments. In: Proc. 12th ACM SIGGRAPH/Eurographics Symp. Computer Animation. 2013, p. 115–124.

[12] van Toll, W, Triesscheijn, R, Kallmann, M, Oliva, R, Pelechano, N, Pettré, J, et al. A comparative study of navigation meshes. In: Proc. 9th ACM SIGGRAPH Int. Conf. Motion in Games. 2016, p. 91–100.

[13] Kallmann, M. Dynamic and robust Local Clearance Triangulations. ACM Trans Graph 2014;33(5).

[14] Geraerts, R. Planning short paths with clearance using Explicit Corridors. In: Proc. IEEE Int. Conf. Robotics and Automation. 2010, p. 1997–2004.

[15] Oliva, R, Pelechano, N. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. Computers & Graphics 2013;37(5):403–412.

[16] Hart, PE, Nilsson, NJ, Raphael, B. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Systems Science and Cybernetics 1968;4(2):100–107.

[17] Karamouzas, I, Geraerts, R, Overmars, MH. Indicative routes for path planning and crowd simulation. In: Proc. 4th Int. Conf. Foundations of Digital Games. 2009, p. 113–120.

[18] Jaklin, NS, Cook IV, AF, Geraerts, R. Real-time path planning in heterogeneous environments. Computer Animation and Virtual Worlds 2013;24(3):285–295.

[19] Helbing, D, Molnár, P. Social force model for pedestrian dynamics. Physical Review E 1995;51(5):4282–4286.

[20] van den Berg, JP, Guy, SJ, Lin, MC, Manocha, D. Reciprocal n-body collision avoidance. In: Proc. 14th Int. Symp. Robotics Research. 2011, p. 3–19.

[21] Wolinski, D, Lin, MC, Pettré, J. WarpDriver: Context-aware probabilistic motion prediction for crowd simulation. ACM Trans Graph 2016;35(6):164:1–164:11.

[22] Dutra, TB, Marques, R, Cavalcante-Neto, JB, Vidal, CA, Pettré, J. Gradient-based steering for vision-based crowd simulation algorithms. Comput Graph Forum 2017;36(2):337–348.

[23] Kremyzas, A, Jaklin, NS, Geraerts, R. Towards social behavior in virtual-agent navigation. Science China - Information Sciences 2016;59(11):112102.

[24] Ren, Z, Charalambous, P, Bruneau, J, Peng, Q, Pettré, J. Group modeling: a unified velocity-based approach. Comput Graph Forum 2017;36:45–56.

[25] Pettré, J, Grillon, H, Thalmann, D. Crowds of moving objects: Navigation planning and simulation. In: Proc. IEEE Int. Conf. Robotics and Automation. 2007, p. 3062–3067.

[26] van Toll, WG, Cook IV, AF, Geraerts, R. Real-time density-based crowd simulation. Computer Animation and Virtual Worlds 2012;23(1):59–69.

[27] Höcker, M, Berkhahn, V, Kneidl, A, Borrmann, A, Klein, W. Graph-based approaches for simulating pedestrian dynamics in building models. In: eWork and eBusiness in Architecture, Engineering and Construction. 2010, p. 389–394.

[28] Sud, A, Gayle, R, Andersen, E, Guy, S, Lin, MC, Manocha, D. Real-time navigation of independent agents using adaptive roadmaps. In: Proc.

ACM Symp. Virtual Reality Software and Technology. 2007, p. 99–106.
[29] Stüvel, SA, Magnenat-Thalmann, N, Thalmann, D, van der Stappen, AF. Torso crowds. IEEE Trans Vis Comput Graphics 2016;23(7):1823–1837.
[30] Treuille, A, Cooper, S, Popović, Z. Continuum crowds. ACM Trans Graph 2006;25:1160–1168.
[31] Patil, S, van den Berg, JP, Curtis, S, Lin, MC, Manocha, D. Directing crowd simulations using navigation fields. IEEE Trans Vis Comput Graphics 2010;17:244–254.
[32] Bhattacharya, S, Likhachev, M, Kumar, V. Topological constraints in search-based robot path planning. Autonomous Robots 2012;33(3):273–290.
[33] Knepper, RA, Srinivasa, SS, Mason, MT. Toward a deeper understanding of motion alternatives via an equivalence relation on local paths. Int Journal of Robotics Research 2011;31(2):167–186.
[34] Mavrogiannis, CI, Knepper, RA. Multi-agent path topology in support of socially competent navigation planning. Int Journal of Robotics Research 2019;38(2–3):338–356.
[35] Karamouzas, I, Geraerts, R, van der Stappen, AF. Spacetime group motion planning. In: Proc. Workshop on the Algorithmic Foundations of Robotics. 2012, p. 227–243.
[36] Chu, ML, Parigi, P, Latombe, JC, Law, KH. Simulating effects of signage, groups, and crowds on emergent evacuation patterns. AI & Society 2015;30:493–507.
[37] Zhang, Z, Jia, L, Qin, Y. Optimal number and location planning of evacuation signage in public space. Safety Science 2017;91:132–147.
[38] Boost, . The Boost C++ library. `http://www.boost.org/`; 2020.
[39] Hershberger, J, Snoeyink, J. Computing minimum length paths of a given homotopy class. Comput Geom Theory Appl 1994;4(2):63–97.

Appendix A. Implementation details

In our implementation of the crowd-simulation framework, we model agents as disks with a radius of 0.25m, a mass of 80kg, and a preferred speed s_{pref} of 1.2m/s, except in the *UTurn* experiments where we deliberately vary s_{pref} among agents.

Appendix A.1. Simulation loop

Our simulation loop uses fixed timesteps (*frames*) of $\Delta t = 0.1$s. Per frame, the program performs the following *substeps*:

1. Build a kd-tree of the agents' positions.
2. For each agent, do a point-location query in the navigation mesh (described in Appendix A.2).
3. For each agent A_j, compute a visibility polygon V_j by traversing the navigation mesh.
4. For each agent A_j, compute neighboring objects. The neighboring *agents* are those that currently collide with A_j, plus the 10 nearest agents that are inside V_j *and* in a 180° cone centered at A_j's viewing direction $\mathbf{d}$. The neighboring *obstacles* are all (partial) obstacle segments within 5m of A_j that bound the visibility polygon.
5. For each agent, perform path following (Appendix A.3) to compute a preferred velocity $\mathbf{v}_{\text{pref}}$.
6. For each agent, convert $\mathbf{v}_{\text{pref}}$ to a strategy S_F. (We do not check for conflicts with the global strategy S_G, as our path-following algorithm never takes detours around obstacles.)
7. For each agent, perform velocity adjustment (Appendix A.4) to compute an adjusted preferred velocity $\mathbf{v}_{\text{adj}}$.
8. For each agent, convert $\mathbf{v}_{\text{adj}}$ to a strategy S_A. If $Check_A$ = `True`, check for conflicts with S_F. If $S_A \neq_c S_F$:
   - If the global path π was last changed $\geq T_R$ seconds ago, compute a path π' constrained by S_A. If π' is below a maximum detour length W, use it.
   - Otherwise, if $Strict_A$ = `True`, take an alternative $\mathbf{v}'_{\text{adj}} = \mathbf{v}_{\text{pref}}$, i.e. ignore velocity adjustment.
9. For each agent, perform collision avoidance (Appendix A.5) to compute a new velocity $\mathbf{v}_{\text{new}}$.
10. For each agent, convert $\mathbf{v}_{\text{new}}$ to a strategy S_C. If $Check_C$ = `True`, check for conflicts with S_A. If $S_C \neq_c S_A$:
    - If the global path π was last changed $\geq T_R$ seconds ago, compute a path π' constrained by S_C. If π' is below a maximum detour length W, use it.
    - Otherwise, if $Strict_C$ = `True`, use an alternative velocity $\mathbf{v}'_{\text{new}}$ constrained by S_A.
11. For each agent, compute a contact force vector $\mathbf{F}$ for any collisions that are currently happening [19].
12. For each agent, update the velocity and position as follows:

$$\mathbf{a} := \mathbf{F}/m + (\mathbf{v}_{\text{new}} - \mathbf{v})/t_r, \quad \mathbf{v} := \mathbf{v} + \mathbf{a} \cdot \Delta t, \quad \mathbf{p} := \mathbf{p} + \mathbf{v} \cdot \Delta t$$

    where $t_r = 0.25$s is a *relaxation time*. The viewing direction $\mathbf{d}$ is updated similarly to $\mathbf{v}$, but without the factor $\mathbf{F}/m$, so that agents do not rotate when they are pushed aside.

Note that the strategic level is represented by substeps 6, 8, and 10. Without these substeps, we would obtain a regular crowd-simulation framework with four navigation algorithms.

The 12 substeps were chosen in this particular way to maximize the simulation's efficiency. All substeps except the first contain an independent process per agent, and their work can be distributed over parallel threads. We use 4 threads for our performance experiments in Section 7.5.

Appendix A.2. Path planning

As our navigation mesh, we use the Explicit Corridor Map (ECM) [14]: the medial axis (MA) of the environment annotated with nearest-obstacle information. We construct it using the Voronoi library of Boost [38]. To obtain a set of passages from the ECM, we compute its *local minima*: the points on the MA where the distance to obstacles is locally smallest. For each local minimum p, we define a passage as the line segment between the two nearest obstacle points of p. There is at most one local minimum per MA edge, so there are $O(N)$ passages in total, and they subdivide the environment into $O(N)$ regions.

To plan a *path* using the ECM, we first use A* search [16] to find a shortest path on the medial axis. We then apply the funnel algorithm [39] to compute a shortest path (that keeps a distance of 0.5m to obstacles) in the same homotopy class. The sequence of *passages* traversed by this path (and thus the global strategy S_G) can be obtained trivially during the A* search.

Appendix A.3. Path following

In the *CA-Only* simulation variant, we compute a preferred velocity $\mathbf{v}_{\text{pref}}$ that sends the agent in a straight line to its goal $\mathbf{g}_j$ at its preferred speed s_{pref}. In all other simulation variants, we use the following path-following algorithm:

Table 4. Overview of the most important notation used in this paper.

Category	Symbol	Meaning
Agent properties	A_j	Agent with index j in the simulation
	$\mathbf{p}_j$, $\mathbf{p}$	Position of A_j in the environment
	$\mathbf{v}_j$, $\mathbf{v}$	Current velocity of A_j (in m/s)
Strategies	$D_{(.)}(A_j, B)$	Navigation decision of A_j with respect to an object B. Its value is `L`, `R`, or `X`.
	$S_{(.)}(A_j)$, $S_{(.)}$	Navigation strategy of A_j, containing decisions with respect to all objects
	$S_1 =_c S_2$	Statement that two strategies S_1 and S_2 are consistent (i.e. they have no conflicts)
	$S_1 \neq_c S_2$	Statement that two strategies S_1 and S_2 are inconsistent (i.e. they have at least one conflict)
	τ	Time window (in seconds) used for converting a velocity to a strategy
Implementation: General	π	Path of A_j, computed by a path-planning algorithm
	$\mathbf{v}_{\text{pref}}$	Preferred velocity of A_j, computed by a path-following algorithm
	$\mathbf{v}_{\text{adj}}$	Adjusted preferred velocity of A_j, computed by a velocity-adjustment algorithm
	$\mathbf{v}_{\text{new}}$	New velocity of A_j, computed by a collision-avoidance algorithm
Implementation: Strategies	S_G	Global navigation strategy, computed from the path π
	S_F	Navigation strategy for path following, computed from $\mathbf{v}_{\text{pref}}$
	S_A	Navigation strategy for velocity adjustment, computed from $\mathbf{v}_{\text{adj}}$
	S_C	Navigation strategy for collision avoidance, computed from $\mathbf{v}_{\text{new}}$
	π'	Alternative path computed in case of a conflict
	$\mathbf{v}'_{\text{adj}}$, $\mathbf{v}'_{\text{new}}$	Alternative velocities computed in case of a conflict
Implementation: Parameters	T_R	Re-planning time: the minimum time (in seconds) between two updates of π
	$Check_A$, $Check_C$	Whether to check for conflicts at the level of S_A and S_C, respectively
	$Strict_A$, $Strict_C$	Whether to always reject the alternative velocity $\mathbf{v}'_{\text{adj}}$ and $\mathbf{v}'_{\text{new}}$, respectively
Abstract framework	$L_0, \ldots L_{l-1}$	Sequence of navigation algorithms used by A_j
	P_i	Plan (e.g. a path or a velocity) computed by algorithm L_i
	S_i	Navigation strategy computed from plan P_i
	$P'_{k(i)}$	Alternative higher-level plan ($k(i) < i$), computed when $S_i \neq_c S_{i-1}$
	P'_i	Alternative current-level plan, computed when $S_i \neq_c S_{i-1}$

Let $\mathbf{p}^-_{\text{ref}}$ and $\mathbf{p}^-_{\text{att}}$ be an agent's last used reference point and attraction point on its path π. First, we compute the new reference point $\mathbf{p}_{\text{ref}}$ as the point on π between $\mathbf{p}^-_{\text{ref}}$ and $\mathbf{p}^-_{\text{att}}$ that is closest to the agent [18]. Then, we compute the new attraction point $\mathbf{p}_{\text{att}}$ as the first point on π from the following options:

- the last point of π (i.e. the agent's goal $\mathbf{g}_j$);
- the point that lies 5m ahead of $\mathbf{p}_{\text{ref}}$ along π;
- the first point on π that the agent cannot see, starting at $\mathbf{p}_{\text{ref}}$.

If $\mathbf{p}_{\text{att}} = \mathbf{p}_{\text{ref}}$, the agent does not know how to follow its path. It then immediately re-plans its global path and computes $\mathbf{p}_{\text{att}}$ again. Finally, the preferred velocity $\mathbf{v}_{\text{pref}}$ is computed as the vector $\mathbf{p}_{\text{att}} - \mathbf{p}$ scaled to magnitude s_{pref}.

Appendix A.4. Velocity adjustment

In the *CA-Only*, *Standard*, and *Standard+* simulation variants, we do not adjust the preferred velocity; that is, we use $\mathbf{v}_{\text{adj}} = \mathbf{v}_{\text{pref}}$. In the other simulation variants, we use the *Streams* algorithm [6]. This algorithm computes $\mathbf{v}_{\text{adj}}$ by interpolating between $\mathbf{v}_{\text{pref}}$ and a 'perceived stream velocity' $\mathbf{v}_{\text{str}}$, which is a weighted average of the velocities of neighboring agents. Agents moving in the opposite direction are ignored in this calculation. The interpolation between $\mathbf{v}_{\text{pref}}$ and $\mathbf{v}_{\text{str}}$ is based on an *incentive* between 0 and 1, which depends on the local crowd density and on $\angle(\mathbf{v}_{\text{pref}}, \mathbf{v}_{\text{str}})$. Compared to the original *Streams* publication [6], we ignore the 'time spent' factor that increases an agent's incentive over time, and we set the agent's 'internal motivation' to zero.

Appendix A.5. Collision avoidance

We use a custom collision-avoidance routine that yielded better behavior than existing methods. To compute a new velocity $\mathbf{v}_{\text{new}}$ for an agent A_j, we sample 15 candidate angles in a 180° range around $\mathbf{v}_{\text{adj}}$, and 2 candidate speeds (s_{pref} and $0.5 \cdot s_{\text{pref}}$). Each combination leads to a candidate velocity.

Let $\mathbf{v}$ be the agent's last used velocity. For a candidate velocity $\mathbf{v}''$, let $\delta(\mathbf{v}'')$ be the distance to the first collision with nearby objects if A_j would use $\mathbf{v}''$, clamped to a maximum distance $d_{\max} = 5\text{m}$. We choose the optimal velocity $\mathbf{v}_{\text{new}}$ as:

$$\operatorname*{argmin}_{\mathbf{v}''} \Big(d_{\max} - \delta(\mathbf{v}'') + \angle(\mathbf{v}'', \mathbf{v}_{\text{adj}}) + \angle(\mathbf{v}'', \mathbf{v}) + \frac{\|\mathbf{v}''\| - s_{\text{pref}}}{s_{\text{pref}}} \Big).$$

Supplementary material

Supplementary material associated with this article can be found, in the online version, at `https://doi.org/10.1016/j.cag.2020.04.003`.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Wouter van Toll: Conceptualization, Methodology, Investigation, Software, Validation, Writing - original draft, Writing - review & editing, Visualization. Julien Pettré: Conceptualization, Methodology, Validation, Writing - review & editing, Supervision, Project administration, Funding acquisition.