Generalized Microscropic Crowd Simulation using Costs in Velocity Space

Wouter van Toll, Fabien Grzeskowiak, Axel López, Javad Amirian, Florian Berton, Julien Bruneau, Beatriz Cabrero Daniel, Alberto Jovane, Julien Pettré

Generalized Microscopic Crowd Simulation using Costs in Velocity Space

Wouter van Toll
wouter.van-toll@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Fabien Grzeskowiak
fabien.grzeskowiak@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Axel López
axel.lopez-gandia@irisa.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Javad Amirian
javad.amirian@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Florian Berton
florian.berton@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Julien Bruneau
julien.bruneau@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Beatriz Cabrero Daniel
beatriz.cabrero@upf.edu
Universitat Pompeu Fabra
Barcelona, Spain

Alberto Jovane
alberto.jovane@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

Julien Pettré
julien.pettre@inria.fr
Univ Rennes, Inria, CNRS, IRISA
Rennes, France

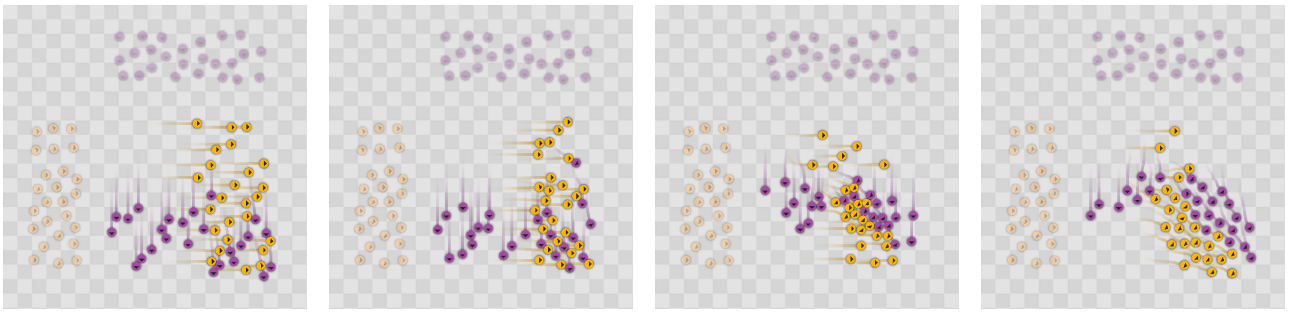


Figure 1: A scenario where 25 agents (in orange) move to the right and 25 other agents (in purple) move down. This figure shows the crowd after 10 seconds, simulated using various algorithms in the UMANS framework. From left to right: RVO [van den Berg et al. 2008], ORCA [van den Berg et al. 2011], PLEdestrians [Guy et al. 2010], and PowerLaw [Karamouzas et al. 2014].

ABSTRACT

To simulate the low-level ('microscopic') behavior of human crowds, a local navigation algorithm computes how a single person ('agent') should move based on its surroundings. Many algorithms for this purpose have been proposed, each using different principles and implementation details that are difficult to compare.

This paper presents a novel framework that describes local agent navigation generically as optimizing a cost function in a velocity space. We show that many state-of-the-art algorithms can be translated to this framework, by combining a particular cost function with a particular optimization method. As such, we can reproduce many types of local algorithms using a single general principle.

Our implementation of this framework, named *UMANS (Unified Microscopic Agent Navigation Simulator)*, is freely available online. This software enables easy experimentation with different algorithms and parameters. We expect that our work will help understand the true differences between navigation methods, enable honest comparisons between them, simplify the development of new local algorithms, make techniques available to other communities, and stimulate further research on crowd simulation.

CCS CONCEPTS

• **Computing methodologies** → **Motion path planning**; **Intelligent agents**; **Real-time simulation**.

KEYWORDS

crowd simulation, navigation, collision avoidance, intelligent agents

ACM Reference Format:
Wouter van Toll, Fabien Grzeskowiak, Axel López, Javad Amirian, Florian Berton, Julien Bruneau, Beatriz Cabrero Daniel, Alberto Jovane, and Julien Pettré. 2020. Generalized Microscopic Crowd Simulation using Costs in Velocity Space. In *Symposium on Interactive 3D Graphics and Games (I3D '20), May 5–7, 2020, San Francisco, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3384382.3384532

I3D '20, May 5–7, 2020, San Francisco, CA, USA

1 INTRODUCTION

The real-time simulation of human crowds is an important field of research with both real-world and entertainment applications. It is a highly hybrid research area combining aspects of computer animation, artificial intelligence, robotics, psychology, and more. Algorithms for crowd simulation can broadly be categorized as either *flow-based* (modelling the entire crowd as a fluid-like substance) or *agent-based* (modelling each member of the crowd as an individual agent). Within the agent-based paradigm, it is common to split the problem into *global* navigation (finding an overall path around the environment's static obstacles) and *local* navigation (letting an agent follow this path while avoiding dynamic objects).

Local navigation has been a popular research topic in particular: many algorithms have been proposed over the past decades. These are also referred to as *microscopic* simulation methods because they model the low-level behavior of individuals. Generally, at a given moment in time, a local navigation algorithm should compute a velocity or acceleration that lets an agent follow its path while respecting local rules, such as collision avoidance with other agents.

1.1 Motivation

New algorithms for microscopic crowd simulation appear regularly, each with their own apparent (dis)advantages and based on different principles. Also, parameter settings and implementation details can have a large impact on the overall behavior of a crowd. This renders it near-impossible to compare algorithms on a conceptual level, or to make any general claims about the quality of a method.

For these reasons, we observe a clear need for a framework that uses a single principle to replicate the essence of as many algorithms as possible. Such a framework would be useful for obtaining a deeper understanding of the differences between methods, for allowing honest comparisons, and for steering further research.

1.2 Goals and Contributions

This paper has two main purposes. First, we show that many algorithms for local navigation can be reformulated as methods that *optimize a cost function in a velocity space.* After the appropriate transformations, the algorithms can differ in terms of:

(1) the cost function C that they use,
(2) the (optimization) method that they apply to C to compute an acceleration vector for the agent.

Thus, we obtain a framework that can reproduce many local navigation algorithms by making certain choices for both items.

Second, we present an implementation named *UMANS (Unified Microscopic Agent Navigation Simulator)*, which uses these principles to integrate various local navigation methods into one system. We show that UMANS can indeed reproduce many state-of-the-art microscopic algorithms. The software allows for easy comparison of algorithms and parameter settings in user-specified scenarios. UMANS is intended as a shared platform, and its source code is available online: https://project.inria.fr/crowdscience/download/.

Compared to other crowd-simulation frameworks [Curtis et al. 2016; Kielar et al. 2016; Singh et al. 2009a; van Toll et al. 2015], we focus only on local navigation. Therefore, our work could be seen as one component of a larger system. In exchange, we aim to truly understand the differences between microscopic algorithms, by expressing these algorithms in similar terms and unifying as many other settings as possible. We expect that this will greatly aid future research.

2 RELATED WORK

Crowd simulation has been an active research topic during the past two decades. As mentioned, algorithms for crowd simulation can be subdivided into two categories: agent-based and flow-based.

2.1 Agent-Based Simulation

In an *agent-based* crowd simulation, each person in the crowd is modelled as an intelligent agent with its own properties and goals. Each agent tries to reach its own goal position while avoiding collisions with obstacles and other agents. To model this, it is common to distinguish between *global* and *local* navigation. First, an agent computes a global path around static obstacles, typically using a *navigation mesh* that describes the environment for navigation purposes [van Toll et al. 2016]. Then, during the simulation loop, the agent repeatedly makes a local adjustment to step in a certain direction. This local behavior can be based on many criteria: an agent may want to move towards it goal, avoid collisions with other agents, stay close to a specific agent or a group, and so on.

By far, most crowd-simulation literature concerns local navigation (and collision avoidance in particular). Overall, these 'microscopic' algorithms can be based on forces [Helbing and Molnár 1995; Karamouzas et al. 2014; Reynolds 1999], velocity selection [Moussaïd et al. 2011; Paris et al. 2007; Pettré et al. 2009; van den Berg et al. 2011, 2008], or vision [Dutra et al. 2017; López et al. 2019; Ondřej et al. 2010]. In this paper, we reproduce these different types of algorithms using one common principle. Section 4 will discuss many of these algorithms and their translation to our framework.

To bridge the conceptual gap between global and local navigation, one can resolve conflicts between these two [van Toll and Pettré 2019] or add a form of 'mid-term planning' in-between [Bruneau and Pettré 2017; Golas et al. 2013; Kapadia et al. 2009]. The different 'levels' of navigation form a crowd-simulation framework of which several implementations have been proposed [Curtis et al. 2016; Kielar et al. 2016; Pelechano et al. 2007; van Toll et al. 2015]. In this paper, we will focus purely on local navigation. Our goal is to capture many types of local algorithms in a single system.

2.2 Other Simulation Methods

In contrast to agent-based simulations, *flow-based* methods model the crowd as one entity [Narain et al. 2009; Patil et al. 2010; Treuille et al. 2006; Weiss et al. 2017]. They can successfully model large and dense crowds where the behavior of individual people is less relevant, but they are not as suitable for lower-density crowds.

Data-driven simulation methods base the crowd motion directly on (observed) trajectories [Lee et al. 2007; Lerner et al. 2007], or they use pre-computed *patches* with periodically repeating crowd motion [Yersin et al. 2009]. Motivated by the complexity of human behavior, methods based on machine learning are currently surging [Amirian et al. 2019; Pellegrini et al. 2012; Zhong et al. 2016].

As mentioned before, our work focuses on local navigation within the agent-based paradigm. The other aspects of (or methodologies for) crowd simulation are considered to be out of scope.

2.3 Evaluation and Shared Platforms

A growing research topic lies in measuring the 'realism' of a simulation, by measuring the similarity between two fragments of (real or simulated) crowd motion. This requires a way to summarize the motion of a crowd, for example by detecting patterns among trajectories [Wang et al. 2016] or by computing fundamental diagrams [Zhang 2012]. However, since each real-world scenario is different, it is difficult to draw general conclusions from such measurements.

The behavior of a simulation is strongly influenced by parameter settings. With this in mind, it is possible to calibrate the parameters of a microscopic algorithm to match a given set of trajectories [Wolinski et al. 2014]. With such techniques, many algorithms can get reasonably close to an input scenario, but this does not yield very valuable insights into the overall quality of an algorithm.

Several implementations of full crowd-simulation frameworks exist [Curtis et al. 2016; Kielar et al. 2016; Pelechano et al. 2007; van Toll et al. 2015]. Because they all have different implementation details, it is difficult to say which framework works best for a particular application, even if we focus on local navigation only.

A project close to our work is *SteerSuite* [Singh et al. 2009a], an open framework specifically meant for local navigation. However, SteerSuite (as well as the other frameworks) treats local navigation as a black box. With UMANS, we deliberately go one step 'deeper' by translating all local algorithms to a single domain (i.e. the optimization of cost functions). This should help researchers understand the differences between algorithms at a deeper conceptual level.

3 SYSTEM OVERVIEW

This section describes our system for microscopic simulation. It first defines the simulation loop, followed by a general framework for local navigation using cost functions and optimization methods. Section 4 will translate many algorithms to this framework.

As is usual in this field, we approximate each *agent* A_j by a disk with radius r_j. Agents may be added or removed over time. Let m be the largest number of agents being simulated simultaneously.

To keep our discussion orderly, we will assume that the *environment* $\mathcal{E}$ is an unbounded 2D plane without obstacles. All concepts can be extended to handle polygonal obstacles and environment boundaries, without any fundamental changes.

3.1 Simulation Loop

We use a simulation loop with frames of a fixed length Δt. The simulation runs in real time if each frame takes at most Δt real-world seconds to compute. A frame consists of the following steps:

(1) Create a spatial hash containing all agent positions, to facilitate nearest-neighbor computations.
(2) For each agent A_j, find all agents within r_N meters of A_j, where r_N is a parameter. These are the neighboring agents that A_j will consider during local navigation (in step 4).
(3) For each agent A_j, compute a preferred velocity $\mathbf{v}_{\text{pref}}$. In our implementation, this is the velocity that would move A_j directly to its goal $\mathbf{g}$ at a preferred speed s_{pref}. This could be extended to a velocity that follows a global path, but in this paper, we deliberately focus on the local aspect only.
(4) For each agent A_j, perform a local navigation algorithm of choice. This induces an acceleration vector $\mathbf{a}$, prescribing a change in velocity. Note: this step is the focus of this paper, and we will discuss it further in Sections 3.2 and 4.
(5) For each agent A_j, compute a contact force vector $\mathbf{F}_c$ due to any collisions that are currently happening.
(6) For each agent A_j, update the velocity $\mathbf{v}$ and position $\mathbf{p}$ via forward Euler integration:

$$\mathbf{v} := \mathit{clamp}(\mathbf{v} + \mathit{clamp}(\mathbf{a}, a_{\max}) \cdot \Delta t, s_{\max}) + \mathbf{F}_c/m \cdot \Delta t,$$

$$\mathbf{p} := \mathbf{p} + \mathbf{v} \cdot \Delta t,$$

where m is the mass of the agent, $s_{\max}$ and $a_{\max}$ are its maximum speed and acceleration, and $\mathit{clamp}(\mathbf{x}, L)$ clamps a vector $\mathbf{x}$ to a maximum length L. (Technically, other integration schemes could be used as well. We choose forward Euler as it is the most common choice in this community.)

Step 4 is the focus of our work. Each agent can use its own local navigation algorithm in this step. Our main contribution is that we replicate many types of local algorithms in a single framework, which we will present next.

3.2 Local Navigation Framework

We now describe our framework for local navigation, which fills in step 4 of the simulation loop. The two main components of this framework are *cost functions* and *optimization methods*.

3.2.1 Cost Function Overview. For an agent A_j at a given moment in the simulation, the purpose of a local navigation algorithm is to compute how A_j should update its velocity for the upcoming simulation step. The idea is that the agent should stay close to its *preferred* velocity while respecting local rules, such as collision avoidance with nearby obstacles and other agents.

For this purpose, let the *velocity space* $\mathcal{V} \subseteq \mathbb{R}^2$ be the set of all possible velocities that agents can have. The two axes of $\mathcal{V}$ are the x- and y-coordinates of velocities. Thus, $\mathcal{V}$ can be thought of a disk around $(0, 0)$ with a radius of the maximum speed $s_{\max}$.

Next, let a *(velocity) cost function* $C : \mathcal{V} \to \mathbb{R}$ be a function that assigns to each possible velocity $\mathbf{v}' \in \mathcal{V}$ a scalar cost $C(\mathbf{v}')$. This cost can be thought of as the 'attractiveness' of choosing $\mathbf{v}'$ as the agent's next velocity, where a lower cost is better.

3.2.2 Common Components of a Cost Function. The cost of a velocity $\mathbf{v}'$ for an agent A_j can be based on various kinds of information, including the agent's *current* position $\mathbf{p}$ and velocity $\mathbf{v}$, its *preferred* velocity $\mathbf{v}_{\text{pref}}$ and goal position $\mathbf{g}$, and the positions and velocities of *other agents* or *obstacles* around A_j. This information induces several metrics that are often used in a cost function C:

Distance *dist* (A_j, A_k) The current Euclidean distance (in meters) between agents A_j and A_k.

Time to collision *ttc* $(\mathbf{v}', A_j, A_k)$ The time (in seconds) after which agent A_j will collide with another agent A_k, assuming that A_j uses $\mathbf{v}'$ and A_k uses its current (observed) velocity.

Distance to collision *dc* $(\mathbf{v}', A_j, A_k)$ The distance to the point where A_j and A_k will collide under the same assumptions, i.e. $\mathit{ttc}(\mathbf{v}', A_j, A_k) \cdot ||\mathbf{v}'||$.

Time to closest approach *ttca* $(\mathbf{v}', A_j, A_k)$ The time at which the distance between A_j and A_k will be smallest. This is equal to *ttc* if the agents are expected to collide.

Distance of closest approach $dca(\mathbf{v}', A_j, A_k)$ The predicted smallest distance between A_j and A_k, i.e. their distance after $ttca(\mathbf{v}', A_j, A_k)$ seconds. This is zero if the agents will collide.

The exact equations for these metrics can be found throughout literature, and our software provides implementations for them. Each metric can also be defined for a neighboring (static) *obstacle* instead of a neighboring agent. Also, for each concept, we can define a version that takes the minimum time or distance among *all* neighboring agents and obstacles. We will denote these using capital letters: $TTC(\mathbf{v}', A_j)$, $DC(\mathbf{v}', A_j)$, $TTCA(\mathbf{v}', A_j)$, and $DCA(\mathbf{v}', A_j)$.

It is worth noting that C is not necessarily smooth, and its *gradient* ∇C may not be well-defined. For example, if C uses *TTC* or *DC*, the costs of two nearly-similar velocities $\mathbf{v}'$ and $\mathbf{v}''$ can be very different if one velocity causes a collision while the other avoids it. (By contrast, *TTCA* and *DCA do* change smoothly, which is why they are preferred by algorithms that rely on the gradient of C.)

3.2.3 Optimization Methods. Using the velocity cost function C, there are several ways to choose the agent's next action. Throughout literature, we identify two main options:

Gradient step Starting at the agent's current velocity $\mathbf{v}$, move in the opposite direction of the gradient ∇C. The result is an acceleration vector $\mathbf{a} = -\nabla C(\mathbf{v})$. Naturally, this approach requires $\nabla C(\mathbf{v})$ to be well-defined and computable.

Global optimization Find a velocity $\mathbf{v}^* \in \mathcal{V}$ with minimal cost. The search may be restricted to a subset of $\mathcal{V}' \subseteq \mathcal{V}$; this can be emulated by giving all velocities outside $\mathcal{V}'$ an infinite cost. Finally, convert the resulting velocity $\mathbf{v}^*$ to an acceleration as follows: $\mathbf{a} = (\mathbf{v}^* - \mathbf{v})/\max(\tau, \Delta t)$. Here, τ is a *relaxation time* (in seconds) limiting the agent's change in velocity. Some methods allow agents to immediately use $\mathbf{v}^*$; this can be achieved by using $\tau = 0$.

Depending on the cost function, global optimization might not have an analytical solution. Some implementations therefore approximate the optimal velocity by *sampling* multiple 'candidate' velocities (from the relevant subset $\mathcal{V}'$) and choosing the one with the lowest cost.

3.2.4 Summary. In short, our local navigation framework expresses a navigation method as a combination of the following choices:

(1) a specific *cost function* C that assigns a cost to any velocity (possibly using elements from Section 3.2.2),
(2) a specific *optimization method* over C (i.e. gradient step or global optimization, possibly approximated by sampling).

The next section will show how to replicate many commonly-used navigation algorithms by making particular choices.

4 TRANSLATING EXISTING ALGORITHMS

This section shows how to express many state-of-the-art local navigation algorithms in terms of a cost function C and a local optimization method. Thus, we translate these algorithms to our framework of costs in velocity space, even if they were originally based on different principles. Table 1 gives an overview of all considered algorithms and their translations. We will now discuss these translations in more detail, treating one 'category' of methods at a time.

4.1 Forces

Local navigation methods based on *forces* [Helbing and Molnár 1995; Karamouzas et al. 2009, 2014; Reynolds 1999] treat agents as particles that exert forces to repel each other. Furthermore, each agent receives an attractive force towards its individual goal. As such, at any moment in the simulation, each agent A_j experiences a number of forces $\{\mathbf{F}_1, \ldots, \mathbf{F}_k\}$ that sum up to a vector $\mathbf{F}_{\text{total}}$. This yields an acceleration $\mathbf{a} = \mathbf{F}_{\text{total}}/m$, where m is the agent's mass.

In our framework, this can be emulated by performing gradient step on a cost function that has $\nabla C(\mathbf{v}) = -\mathbf{F}_{\text{total}}/m$. In theory, all other details of C are irrelevant. However, to enable experimentation with other optimization methods, it is useful to define a function C that makes sense for other velocities as well. We propose the following translation. Let $\mathbf{v}^* = \mathbf{v} + \mathbf{F}_{\text{total}}/m \cdot \Delta t$ be the velocity that the agent would reach by applying $\mathbf{F}_{\text{total}}$ in the current time step. For any velocity $\mathbf{v}' \in \mathcal{V}$, we define the cost function as:

$$C(\mathbf{v}') = \frac{1}{2\Delta t}\left\|\mathbf{v}' - \mathbf{v}^*\right\|^2, \quad \nabla C(\mathbf{v}') = \frac{1}{\Delta t}(\mathbf{v}' - \mathbf{v}^*)$$

That is, the cost depends on the distance to $\mathbf{v}^*$, and the gradient points away from $\mathbf{v}^*$, with a magnitude such that the condition $\nabla C(\mathbf{v}) = -\mathbf{F}_{\text{total}}/m$ is met. Figure 2 shows a visual interpretation.

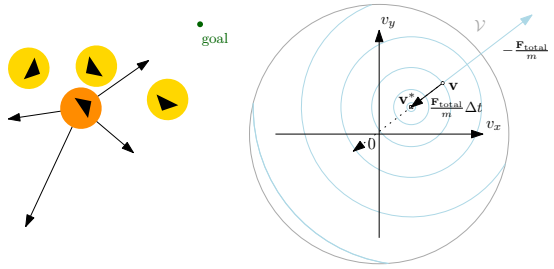


Figure 2: Translating a force-based navigation method to our domain. Left: An agent experiences forces from other agents and from the goal. Right: The cost $C(\mathbf{v}')$ depends on the distance between $\mathbf{v}'$ and the velocity $\mathbf{v}^*$ suggested by the forces.

4.2 Velocity Sampling

4.2.1 Overview. Navigation algorithms based on *velocity sampling* [Karamouzas and Overmars 2010; Moussaïd et al. 2011] explicitly define a cost function C and approximate its minimum. They typically test a fixed number of angles and speeds, limited to a 'viewing cone' in front of the agent. To produce smooth motion, these algorithms either use a relaxation time $\tau > 0$, or they include the difference to $\mathbf{v}$ in their cost function. Figure 3 shows an example of an algorithm that samples regularly in a 180-degree cone.

4.2.2 Specific Considerations. Karamouzas and Overmars [2010] sample regularly in a cone shape, but the range of allowed speeds and angles depends on $TTC(\mathbf{v}_{\text{pref}})$. Table 1 summarizes this range as $range(TTC(\mathbf{v}_{\text{pref}}))$; its definition can be found in the original paper.

The algorithm of Moussaid et al. [2011] first finds an optimal *angle* (via sampling), assuming that the agent will use its preferred speed. This yields a velocity $\mathbf{v}^*$ with $\left\|\mathbf{v}^*\right\| = s_{\text{pref}}$. It then scales down $\mathbf{v}^*$ if a collision is nearby. We can mimic this two-step behavior in one function C as provided in Table 1. It works as follows:

- $K(\mathbf{d})$ is the cost of a direction $\mathbf{d}$ of unit length. It is the cost of the 'full' velocity $\mathbf{d} \cdot s_{\text{pref}}$ as defined in the original paper.
- $S(\mathbf{d})$ is the optimal speed in direction $\mathbf{d}$ as defined in the paper. Its calculation is based on $DC(\mathbf{d} \cdot s_{\text{pref}}, A_j)$.
- The final cost $C(\mathbf{v}')$ uses $K(\frac{\mathbf{v}'}{\|\mathbf{v}'\|})$, i.e. the 'Moussaid cost' of the directional component of $\mathbf{v}'$, but we scale this value so that the minimum (among all velocities in this direction) lies at speed $S(\frac{\mathbf{v}'}{\|\mathbf{v}'\|})$. This way, the global minimum of C matches the velocity that would be chosen originally.

4.2.3 Other Algorithms. For two early collision-avoidance methods [Paris et al. 2007; Pettré et al. 2009], several details are unclear, even after correspondence with their respective authors. Overall, these methods divide $\mathcal{V}$ into angular regions and choose a velocity $\mathbf{v}^*$ inside the most 'attractive' region. In theory, this idea can be replicated using velocity sampling. For the algorithm by Paris et al. [2007], we have already written a working translation that captures the paper's essence. However, due to the uncertainty of crucial details, and due to the absence of the original code and its results, our implementation cannot be fairly judged. For now, we therefore omit these two methods from our discussion.

4.3 Velocity Obstacles

4.3.1 Overview. Algorithms based on *velocity obstacles* [Guy et al. 2010; van den Berg et al. 2011, 2008] define a set $\mathcal{V}_{\text{obs}}$ of velocities for A_j that will lead to a collision in the near future. The purpose of these algorithms is to find the best velocity $\mathbf{v}^* \in \mathcal{V} \setminus \mathcal{V}_{\text{obs}}$. Here, 'best' usually means 'closest to $\mathbf{v}_{\text{pref}}$', but other criteria may be used as well, such as the expected energy consumption caused by a detour [Guy et al. 2010]. The result $\mathbf{v}^*$ can often be found analytically, i.e. without sampling. Figure 4 shows a typical example. In our framework, velocity obstacles can be included in the cost function C by assigning an infinite cost to velocities inside $\mathcal{V}_{\text{obs}}$.

4.3.2 Specific Considerations. Some methods are based on 'reciprocity': the assumption that two agents A_j and A_k evenly share the effort for avoiding a collision. The *Reciprocal Velocity Obstacles (RVO)* algorithm [van den Berg et al. 2008] captures this by basing all time-to-collision computations on $2\mathbf{v}' - \mathbf{v}$ instead of $\mathbf{v}'$. The *Optical Reciprocal Collision Avoidance (ORCA)* method [van den Berg et al. 2011] handles it differently. For each neighboring agent A_k, it defines a half-plane containing all 'optimal' avoidance velocities for A_j, again assuming an equal distribution of effort. It then looks for the velocity closest to $\mathbf{v}_{\text{pref}}$ that lies inside all half-planes. The solution can be found via linear programming.

The first implementation of RVO is actually sampling-based, to approximate a solution that was (at the time) considered too hard to find analytically. Although the RVO paper describes this sampling as "evenly distributed", its source code uses uniform *random* sampling.

The ORCA algorithm also considers the option that the entire velocity space is forbidden, which may happen in dense crowds. In this case, ORCA looks for the 'least bad option' according to another cost function. In our framework, we can implement this 'fallback' behavior by specifying a second cost function C_2 that will be used whenever C 'fails', i.e. when searching over C does not yield any velocity with finite cost. Although it may be possible to merge C and C_2 into one function with the same minimum, the closed-form solution of ORCA relies on this two-step approach.

The *PLEdestrians* method [Guy et al. 2010] uses an easier definition for $\mathcal{V}_{\text{obs}}$, combined with a more involved cost function. The authors show that, for this combination, the optimal velocity $\mathbf{v}^*$ can be found by solving specifically derived equations.

In our code base, programmers can implement global optimization separately per cost function. (For example, the solution for ORCA is publicly available online.) To keep our framework generic, this implementation is *optional*. Whenever an exact solution is not provided, the program can always fall back to sampling. This allows users to easily try a new cost function without having to derive and implement a closed-form solution themselves.

4.4 Gradients and Rendering

4.4.1 Overview. Some navigation algorithms inherently update the velocity $\mathbf{v}$ according to the gradient of a cost function C [Dutra et al. 2017; López et al. 2019; Ondřej et al. 2010]. These algorithms use rendering techniques to simulate the agent's field of vision or optical flow, and then they combine costs per pixel. As such, the cost function C is too complex to evaluate many times. In exchange, the *gradient* ∇C has an analytical form, and the agent can move using the 'gradient step' optimization method.

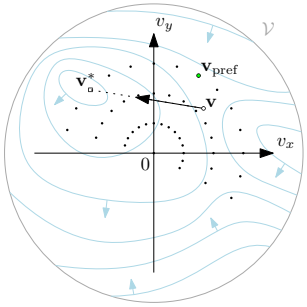


Figure 3: Translating a typical sampling-based navigation method to our domain. Values and gradient of the cost function are visualized in light blue.

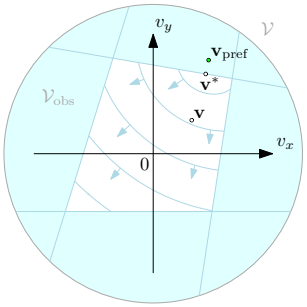


Figure 4: Translating a velocity-obstacle-based navigation method to our domain. The method defines a forbidden space $\mathcal{V}_{\text{obs}}$ (the light blue area) and a cost function in its complement. The costs in $\mathcal{V}_{\text{obs}}$ can be thought of as infinite. The optimal velocity $\mathbf{v}^*$ can often be computed analytically.

Table 1: Overview of local navigation algorithms and their translation to our framework. In the cost functions, we have omitted any irrelevant constants, we have renamed weight parameters to w_a, w_b, w_c, w_d, and time/distance thresholds to $t_{\min}$, $t_{\max}$, $d_{\max}$. For clarity, these weights and thresholds are shown in blue. Furthermore, n is the number of neighbors that an agent considers, $\angle$ denotes the angle (in radians) between two vectors, and $\hat{\mathbf{x}}$ denotes the normalized version of a vector $\mathbf{x}$.

Algorithm	Cost function C (+ fallback function C_2)	Optimization method
[Helbing and Molnár 1995] (SF), [Karamouzas et al. 2009], [Karamouzas et al. 2014] (PowerLaw), etc.	$C(\mathbf{v}') = 1/(2\Delta t) \cdot \|\mathbf{v}' - \mathbf{v}^*\|^2$, where $\mathbf{v}^* = \mathbf{v} + \mathbf{F}_{\text{total}}/m \cdot \Delta t$, and $\mathbf{F}_{\text{total}}$ differs per algorithm	Gradient step (∇C is explicit)
[van den Berg et al. 2008] (RVO)	$C(\mathbf{v}') = w_a / TTC(2\mathbf{v}' - \mathbf{v}, A_j) + \|\mathbf{v}' - \mathbf{v}_{\text{pref}}\|$	'Global optimization': random sampling in $\{\mathbf{v}' \mid \|\mathbf{v}' - \mathbf{v}\|/\Delta t \le a_{\max}\}$, $\tau = 0$
[Guy et al. 2010] (PLEdestrians)	$C(\mathbf{v}') = \begin{cases} \infty, \text{ if } TTC(\mathbf{v}', A_j) < t_{\min} \\ t_{\max}(w_a + w_b \|\mathbf{v}'\|^2) + 2\|\mathbf{g} - \mathbf{p} - t_{\max} \cdot \mathbf{v}'\| \sqrt{w_a w_b}, \text{ otherwise} \end{cases}$	Global optimization, $\tau = 0$
[Karamouzas and Overmars 2010]	$C(\mathbf{v}') = w_a \cdot (1 - \cos \angle(\mathbf{v}', \mathbf{v}_{\text{pref}})) + w_b \cdot \|\|\mathbf{v}'\| - \|\mathbf{v}\|\| / s_{\max} + w_c \cdot \|\mathbf{v}' - \mathbf{v}_{\text{pref}}\| / s_{\max} + w_d \cdot TTC(\mathbf{v}', A_j)/t_{\max}$	'Global optimization': regular sampling in $range(TTC(\mathbf{v}_{\text{pref}}))$, $\tau = 0$
[Moussaïd et al. 2011]	$C(\mathbf{v}') = K(\hat{\mathbf{v}}') \cdot \left(1 + \left(\frac{S(\hat{\mathbf{v}}') - \|\mathbf{v}'\|}{S(\mathbf{v}')}\right)^2\right)$, $K(\mathbf{d}) = 1 + d_{\max}{}^2 + DC^*(\mathbf{d})^2 - 2 d_{\max} \cdot DC^*(\mathbf{d}) \cdot \cos \angle(\mathbf{d}, \mathbf{v}_{\text{pref}})$, $S(\mathbf{d}) = \min(s_{\text{pref}}, DC^*(\mathbf{d})/\tau)$, $DC^*(\mathbf{d}) = \min(d_{\max}, DC(\mathbf{d} \cdot s_{\text{pref}}, A_j))$	'Global optimization': regular sampling in $\{\mathbf{v}' \mid \angle(\mathbf{v}', \mathbf{v}_{\text{pref}}) > \theta_{\max} \wedge \|\mathbf{v}'\| \le s_{\text{pref}}\}$, $\tau = 0.5$
[van den Berg et al. 2011] (ORCA)	$C(\mathbf{v}') = \begin{cases} \infty, \text{ if } \min_{k=0}^{n-1}(\mathbf{v}' - (\mathbf{v} + \frac{1}{2}\mathbf{u}_{jk})) \bullet \mathbf{n}_{jk} < 0, \\ \|\mathbf{v}' - \mathbf{v}_{\text{pref}}\|, \text{ otherwise} \end{cases}$ $C_2(\mathbf{v}') = \max_{k=0}^{n-1}(\mathbf{v}' - (\mathbf{v} + \frac{1}{2}\mathbf{u}_{jk})) \bullet \mathbf{n}_{jk}$, where $\mathbf{u}_{jk}$ and $\mathbf{n}_{jk}$ define the ORCA$^{t_{\max}}$ half-plane for A_j and A_k	Global optimization, $\tau = 0$
[Dutra et al. 2017]	$C(\mathbf{v}') \approx \sum_{k=0}^{n-1}\left(W_k \cdot e^{f_1(\mathbf{v}', A_k)}\right) \cdot \sum_{k=0}^{n-1} W_k + \left(1 - \frac{1}{2}(e^{f_2(\mathbf{v}')} + e^{f_3(\mathbf{v}')})\right)$, $W_k = \frac{1}{dist(A_j, A_k)^2}$, $f_1(\mathbf{v}', A_k) = -\frac{1}{2}\left(\left(ttca(\mathbf{v}', A_j, A_k)/w_a\right)^2 + \left(dca(\mathbf{v}', A_j, A_k)/w_b\right)^2\right)$, $f_2(\mathbf{v}') = -\frac{1}{2}\left(\angle(\mathbf{v}', \mathbf{v}_{\text{pref}})/w_c\right)^2$, $f_3(\mathbf{v}') = -\frac{1}{2}\left(\left(\|\mathbf{v}'\| - \|\mathbf{v}_{\text{pref}}\|\right)/w_d\right)^2$	Gradient step (∇C is explicit)

4.4.2 Coordinate Conversion. For the control of an agent, these methods usually consider a polar coordinate system whose axes are the absolute *speed* s and the *angle* θ. By contrast, our framework uses a Euclidean velocity space. The difference between these coordinate systems is shown in Figure 5. This does not affect C itself, but it does change the analytical form of ∇C. In polar coordinates, the gradient of C has the following form:

$$\nabla_{s,\theta} C = \left(\frac{\partial C}{\partial \theta}, \frac{\partial C}{\partial s}\right).$$

We can first transform this to a Euclidean coordinate frame *relative* to the velocity $\mathbf{v}'$ at which the gradient is requested:

$$\nabla_{\mathbf{v}'} C = \left(s \sin \frac{\partial C}{\partial \theta}, s\left(1 - \cos \frac{\partial C}{\partial \theta}\right) + \frac{\partial C}{\partial s}\right).$$

Next, we can transform it to global coordinates through a matrix:

$$\nabla C(\mathbf{v}') = \frac{1}{\|\mathbf{v}'\|} \begin{bmatrix} v'_y & v'_x \\ -v'_x & v'_y \end{bmatrix} \nabla_{\mathbf{v}'} C(\mathbf{v}').$$

where v'_x and v'_y are the global x- and y-coordinates of $\mathbf{v}'$. For any gradient defined in speed-angle space, this conversion gives a closed form for ∇C in Euclidean velocity space.

4.4.3 Rendering Approximation. The use of rendering limits the usefulness of these algorithms for real-time crowd simulation. Our implementation replaces the aspect of rendering by a computationally cheaper alternative. This makes the code more portable, and it allows to experiment with other optimization methods (such as sampling) in the future. Our approximation computes a weighted average of the costs for each neighboring agent A_k (instead of for each pixel), where the weight for A_k is $1/dist(A_k, A_j)^2$. Although this is not a perfect replication of how many pixels A_k occupies (e.g. it does not consider occlusion among neighbors), it does simulate the effect of treating nearby agents as more important. Finally, we empirically found that scaling the overall cost (and therefore the gradient) by a factor $S = 20$ yields reasonably responsive agents. We will use this in our demonstration as well.

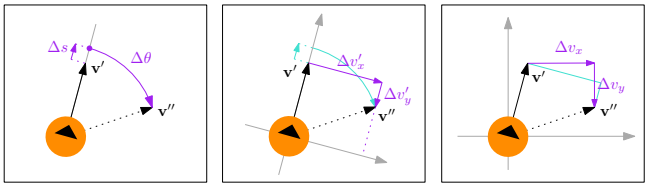


Figure 5: Three coordinate systems for expressing differences between velocities: speed and angle relative to $\mathbf{v}'$ (left), Euclidean space relative to $\mathbf{v}'$ (middle), and global Euclidean space (right). This affects the definition of gradients.

5 IMPLEMENTATION: UMANS

We have implemented the framework from Section 3 with the translated algorithms from Section 4. Our code includes all cost functions from Table 1 and two force models: *social forces* [Helbing and Molnár 1995] and *PowerLaw* [Karamouzas et al. 2014]. Thus, we will demonstrate 8 different navigation algorithms.

The program has been written in platform-independent C++11. For this paper, we have compiled it in Visual Studio 2017 on Windows. The software, named UMANS, is available online[1] so that it can serve as a shared platform for microscopic crowd simulation.

As in other crowd-simulation frameworks, all steps of the simulation loop from Section 3.1 describe an independent process that can be executed per agent (except the first step that builds a spatial hash of all agents). Therefore, to decrease the simulation's computation time, the work inside these steps can be performed for multiple agents on parallel threads. Previous work (e.g. [Curtis et al. 2016; van Toll et al. 2015]) has shown that this can easily yield real-time simulations of crowds with thousands of agents.

5.1 Cost Functions

As described, we express a local navigation algorithm as a combination of a cost function and an optimization method. In our codebase, each cost function is a subclass of the abstract class `CostFunction`, and it can compute three things: the cost $C(\mathbf{v}')$ for a velocity $\mathbf{v}'$, the gradient $\nabla C(\mathbf{v}')$ for a velocity $\mathbf{v}'$, and the velocity $\mathbf{v}^*$ with minimal cost. The implementations of the gradient and the global minimum are *optional*, as some cost functions have no closed-form gradient or no analytical way to find a global minimum. If no implementation is given, the program will automatically resort to sampling.

5.2 Optimization and Policies

For the optimization methods, gradient step and global optimization simply call the appropriate method of `CostFunction`. We have also implemented random and regular sampling in disks or cones inside $\mathcal{V}$. This covers all types of sampling that occur in Table 1.

A `Policy` is a combination of a cost function (possibly including specific parameters) and a velocity selection method (where the 'sampling' option requires several settings). Via XML files, users can specify these policies and assign them to agents, which can have individual start positions, goal positions, and radii.

This set-up allows to easily test 'variants' of an algorithm by changing the cost function, its parameters, or the optimization method. Programmers can easily add new components as well. In the future, we also intend to add tools allowing 'non-programmers' to define new cost functions without having to write a full class.

5.3 Ports and Interpretations of Algorithms

For most navigation algorithms, the original source code is either unavailable or not freely usable. In these cases, the code in UMANS is *our own interpretation* of the algorithm, combined with the translation process of Section 4. One exception is ORCA [van den Berg et al. 2011]: its source code is available online with a compatible license. Therefore, UMANS uses parts of this original code, including the linear-programming solution for finding an optimal velocity.

[1] https://project.inria.fr/crowdscience/download/

6 DEMONSTRATION OF RESULTS

This section demonstrates our implementation in several small crowd-simulation scenarios. The purpose of this is to show that the software can indeed reproduce many local navigation algorithms.

Note that we deliberately do not yet make comparisons *between* algorithms. A comparative study of microscopic algorithms would involve many more details and considerations that would violate the space constraints for this paper. We intend to address this in a separate publication. Our current contribution is the framework itself, which is an important first step towards a honest comparison.

6.1 Scenarios and Settings

Our scenarios and settings are available along with the UMANS software. Example scenarios include agents moving to opposite ends of a circle, a single agent crossing a stream of 10 other agents at various angles, and two groups of 25 agents moving in opposite directions. In all scenarios except the circles, the agent's goals are chosen to lie far away, so that agents keep moving in the same direction after having avoided collisions.

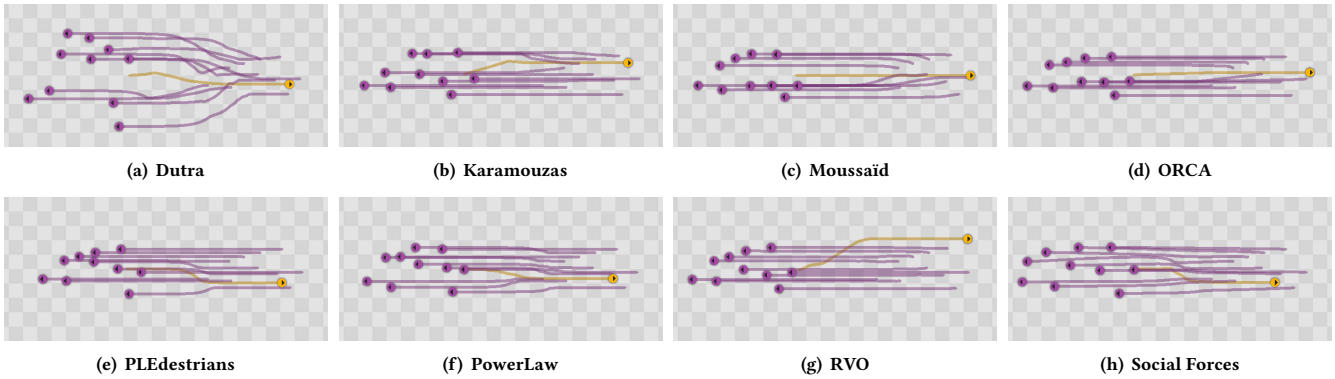
We use simulation settings that are commonly used in this research area. The time step Δt is set to 0.1 s. Agents have a radius r_j of 0.3 m, a preferred speed s_{pref} of 1.3 m/s, a maximum speed s_{max} of 1.6 m/s, a maximum acceleration a_{max} of 5 m/s^2, a mass m of 1 kg, and a nearest-neighbor distance r_N of 100 m (meaning that all agents are considered as neighbors). For the parameters of specific cost functions, we always use the default values suggested by their respective authors. Of course, many more configurations are possible. However, our current purpose is to show that our cost-based framework can reproduce many different algorithms, and not to judge how these algorithms compare to one another.

6.2 Results and Discussion

For all scenarios, the *supplementary document* of this paper contains screenshots of the 8 navigation algorithms side by side. Our *supplementary video* shows the results in motion. For illustrative purposes, we have included one example in this main paper as well. Figure 6 shows the *1to10-180Degrees* scenario, where an agent crosses a group of 10 agents going into the opposite direction.

As mentioned, most navigation methods have been translated literally to our framework. In theory, there should only be minimal differences between our implementation and an original version by the method's authors. As an example, Figure 7 shows how our version of the ORCA [van den Berg et al. 2011] and PowerLaw [Karamouzas et al. 2014] algorithms compares to the code by their authors that is available online. The scenario features one agent crossing a group of 10 agents at a 90-degree angle. This figure shows that the agents behave practically similarly in both implementations. For the other navigation algorithms, we were unable to obtain the original source code, and we cannot make such a comparison.

In general, if there *are* any differences in behavior between our implementation and an original version, then these are due to the general simulation parameters mentioned earlier, or due to implementation choices in our simulation loop. We strongly believe that differences at this level of detail are not worth judging. After all, crowd simulations are highly complex systems where a small detail can have large consequences. Therefore, we suggest to use



(a) Dutra (b) Karamouzas (c) Moussaïd (d) ORCA

(e) PLEdestrians (f) PowerLaw (g) RVO (h) Social Forces

Figure 6: Screenshots of the *1to10-180Degrees* scenario after simulation 10 seconds. All background squares measure 1 m^2.

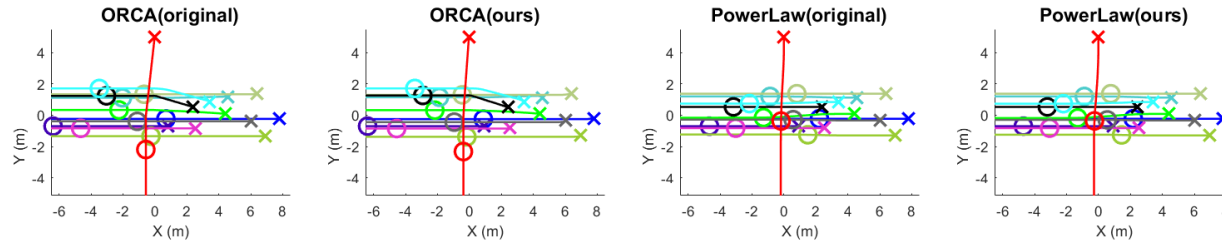


Figure 7: Comparison between UMANS and the original code of ORCA and PowerLaw, using the *1to10-90Degrees* scenario. Each color represents one agent's trajectory, where 'X' is the agent's starting position and 'O' is its position after 5.5 seconds.

UMANS as a reference for future comparisons between algorithms. This way, as many simulation details as possible can be unified, and the comparison can truly focus on the navigation algorithms themselves (i.e. cost functions and optimization methods).

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a general framework for local navigation (e.g. collision avoidance) in agent-based crowd simulation. Many local navigation algorithms exist, and they are strongly influenced by parameter settings and implementation details. We have shown that all these algorithms can be translated to a common principle: optimizing a cost function in velocity space. To our knowledge, *all* collision-avoidance methods can be translated to this domain, as they all fit in one of the categories from Section 4.

Next to providing the translations themselves, we have also presented an implementation (UMANS) that uses this principle to reproduce many state-of-the-art navigation algorithms. UMANS allows users to mix cost functions and optimization methods arbitrarily. By unifying as many other simulation details as possible, our system makes navigation algorithms easier to compare, and it allows for a deeper understanding of their differences. We intend for UMANS to be a shared platform to be used by researchers and end users alike, and we believe that it will be a valuable tool for future research.

With this framework in place, a logical next step is to critically *compare* algorithms, which includes analyzing the influence of parameter settings. Such a comparison was previously not possible (in an objective or meaningful way) due to the many dispersed implementations. The comparison itself could be based on quantitative metrics per agent [Singh et al. 2009b] or on higher-level fundamental diagrams [Zhang 2012], but new theories for measuring a simulation's quality need to be developed as well. We hope that our UMANS framework will facilitate this development.

UMANS is currently purely a (microscopic) simulation engine. To obtain more insights in the differences between navigation algorithms, we intend to develop more tools, e.g. for the visualization of cost functions, for changing simulation settings at run-time, and for easily defining new cost functions in a user-friendly manner.

Another direction for future work is to experiment with other optimization methods, such as gradient descent or simulated annealing. This may be computationally cheaper than sampling while still yielding a velocity whose cost is close to the global minimum. This allows to create more efficient versions of existing microscopic algorithms, and thus to simulate even larger crowds in real-time.

Finally, we believe that the concept of cost-function optimization can also be applied to *other* levels of navigation, such as mid-term and global planning. The main difference between these levels is the time window over which they plan the agent's motion. Even flow-based crowd simulation, considered to be a different paradigm, is eventually based on costs and gradients (but it operates in position space instead of velocity space). On the long term, this idea may lead to a *holistic* approach to crowd simulation in which the distinction between levels and paradigms disappears.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779942 (CROWDBOT) and No 856879 (PRESENT). This work is partly funded by the ANR OPMoPS project (ANR-16-SEBM-0004).

REFERENCES

Javad Amirian, Wouter van Toll, Jean-Bernard Hayet, and Julien Pettré. 2019. Data-driven crowd simulation with generative adversarial networks. In *Proc. Int. Conf. Computer Animation and Social Agents*. 7–10.

Julien Bruneau and Julien Pettré. 2017. EACS: Effective Avoidance Combination Strategy. *Computer Graphics Forum* 36, 8 (2017), 108–122.

Sean Curtis, Andrew Best, and Dinesh Manocha. 2016. Menge: A modular framework for simulating crowd movement. *Collective Dynamics* 1, A1 (2016), 1–40.

Teofilo B. Dutra, Ricardo Marques, Joaquim B. Cavalcante-Neto, Creto A. Vidal, and Julien Pettré. 2017. Gradient-based steering for vision-based crowd simulation algorithms. *Comput. Graph. Forum* 36, 2 (2017), 337–348.

Abhinav Golas, Rahul Narain, and Ming C. Lin. 2013. Hybrid long-range collision avoidance for crowd simulation. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D '13)*. 29–36.

Stephen J. Guy, Jatin Chhugani, Sean Curtis, Pradeep Dubey, Ming Lin, and Dinesh Manocha. 2010. PLEdestrians: A least-effort approach to crowd simulation. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*. 119–128.

Dirk Helbing and Péter Molnár. 1995. Social force model for pedestrian dynamics. *Phys. Rev. E* 51, 5 (1995), 4282–4286.

Mubbasir Kapadia, Shawn Singh, William Hewlett, and Petros Faloutsos. 2009. Egocentric affordance fields in pedestrian steering. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games*. 215–223.

Ioannis Karamouzas, Peter Heil, Pascal van Beek, and Mark H. Overmars. 2009. A predictive collision avoidance model for pedestrian simulation.. In *Proc. Int. Workshop on Motion in Games*. 41–52.

Ioannis Karamouzas and Mark H. Overmars. 2010. A velocity-based approach for simulating human collision avoidance. In *Proc. Int. Conf. Intelligent Virtual Agents*. 180–186.

Ioannis Karamouzas, Brian Skinner, and Stephen J. Guy. 2014. Universal power law governing pedestrian interactions. *Phys. Rev. Lett.* 113 (2014), 238701:1–5. Issue 23.

Peter M. Kielar, Daniel H. Biedermann, and André Borrmann. 2016. *MomenTUMv2: A modular, extensible, and generic agent-based pedestrian behavior simulation framework*. Technical Report TUM-I1643. Technische Universität München, Institut für Informatik.

Kang H. Lee, Myung G. Choi, Qyoun Hong, and Jehee Lee. 2007. Group behavior from video: A data-driven approach to crowd simulation. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*. 109–118.

Alon Lerner, Yiorgos Chrysanthou, and Dani Lischinski. 2007. Crowds by example. *Comput. Graph. Forum* 26, 3 (2007), 655–664.

Axel López, François Chaumette, Eric Marchand, and Julien Pettré. 2019. Character navigation in dynamic environments based on optical flow. *Comput. Graph. Forum* 38, 2 (2019), 181–192.

Mehdi Moussaïd, Dirk Helbing, and Guy Theraulaz. 2011. How simple rules determine pedestrian behavior and crowd disasters. In *Proc. National Academy of Science*, Vol. 108. 6884–6888. Issue 17.

Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C Lin. 2009. Aggregate dynamics for dense crowd simulation. *ACM Trans. Graph.* 28, 5 (2009), 122.

Jan Ondřej, Julien Pettré, Anne-Hélène Olivier, and Stéphane Donikian. 2010. A synthetic-vision based steering approach for crowd simulation. *ACM Trans. Graph.* 29, 4 (2010), 123:1–123:9.

Sébastien Paris, Julien Pettré, and Stéphane Donikian. 2007. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Comput. Graph. Forum* 26 (2007), 665–674. Issue 3.

Sachin Patil, Jur P. van den Berg, Sean Curtis, Ming C. Lin, and Dinesh Manocha. 2010. Directing crowd simulations using navigation fields. *IEEE Trans. Vis. Comput. Graphics* 17 (2010), 244–254. Issue 2.

Nuria Pelechano, Jan M. Allbeck, and Norman I. Badler. 2007. Controlling individual agents in high-density crowd simulation. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*. 99–108.

Stefano Pellegrini, Juergen Gall, Leonid Sigal, and Luc van Gool. 2012. Destination flow for crowd simulation. In *Proc. European Conf. Computer Vision*. 162–171.

Julien Pettré, Jan Ondřej, Anne-Hélène Olivier, Armel Cretual, and Stéphane Donikian. 2009. Experiment-based modeling, simulation and validation of interactions between virtual walkers. In *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation*. 189–198.

Craig Reynolds. 1999. Steering behaviors for autonomous characters. In *Proc. Game Developers Conf.* 763–782.

Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. 2009a. An open framework for developing, evaluating, and sharing steering algorithms. In *Proc. Int. Workshop on Motion in Games*. 158–169.

Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. 2009b. SteerBench: A benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds* 20 (2009), 533–548. Issue 5–6.

Adrien Treuille, Seth Cooper, and Zoran Popović. 2006. Continuum crowds. *ACM Trans. Graph.* 25, 3 (2006), 1160–1168.

Jur P. van den Berg, Stephen J. Guy, Ming C. Lin, and Dinesh Manocha. 2011. Reciprocal n-body collision avoidance. In *Proc. Int. Symp. Robotics Research*. 3–19.

Jur P. van den Berg, Ming C. Lin, and Dinesh Manocha. 2008. Reciprocal Velocity Obstacles for real-time multi-agent navigation. In *Proc. IEEE Int. Conf. Robotics and Automation*. 1928–1935.

Wouter van Toll, Norman Jaklin, and Roland Geraerts. 2015. Towards believable crowds: A generic multi-level framework for agent navigation. In *ASCI.OPEN*.

Wouter van Toll and Julien Pettré. 2019. Connecting global and local agent navigation via topology. In *Proc. ACM SIGGRAPH Int. Conf. Motion, Interaction and Games*. 33:1–33:10.

Wouter van Toll, Roy Triesscheijn, Roland Geraerts, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, and Julien Pettré. 2016. A comparative study of navigation meshes. In *Proc. ACM SIGGRAPH Int. Conf. Motion in Games*. 91–100.

He Wang, Jan Ondřej, and Carol O'Sullivan. 2016. Path patterns: Analyzing and comparing real and simulated crowds. In *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games*. 49–57.

Tomer Weiss, Chenfanfu Jiang, Alan Litteneker, and Demetri Terzopoulos. 2017. Position-based multi-agent dynamics for real-time crowd simulation. In *Proc. ACM SIGGRAPH Int. Conf. Motion in Games*. 10:1–10:8.

David Wolinski, Stephen J. Guy, Anne-Hélène Olivier, Ming C. Lin, Dinesh Manocha, and Julien Pettré. 2014. Parameter estimation and comparative evaluation of crowd simulations. *Comput. Graph. Forum* 33, 2 (2014), 303–312.

Barbara Yersin, Jonathan Maïm, Julien Pettré, and Daniel Thalmann. 2009. Crowd Patches: Populating large-scale virtual environments for real-time applications. In *Proc. Symp. Interactive 3D Graphics and Games*. 207–214.

Jun Zhang. 2012. *Pedestrian fundamental diagrams: Comparative analysis of experiments in different geometries*. Ph.D. Dissertation. Forschungszentrum Jülich.

Jinghui Zhong, Wentong Cai, Linbo Luo, and Mingbi Zhao. 2016. Learning behavior patterns from video for agent-based crowd modeling and simulation. *Autonomous Agents and Multi-Agent Systems* 30, 5 (2016), 990–1019.