Way Portals: Efficient Multi-Agent Navigation with Line-Segment Goals

Sean Curtis* Jamie Snape† Dinesh Manocha‡
University of North Carolina at Chapel Hill
http://gamma.cs.unc.edu/WayPortals

Abstract

It is a common artifact of multi-agent motion planning for groups of agents, following similar paths, to converge to a line. This occurs because the agents' immediate goals, a.k.a. way points, are frequently a shared point in space. Contention for the point goal causes agents to line up and generally interferes with agent motion. By extending the definition of an immediate point goal to a line segment, which we call a "way portal", we enable the agents to better utilize the space available to them in responding to dynamic constraints. We present a novel multi-agent navigation algorithm to efficiently compute the trajectories of autonomous agents using these way portals. We have incorporated the concept into a velocity obstacle-based local navigation model and present a new segment optimization algorithm that efficiently computes a new agent velocity with respect to the way portal. We show how way portal data is extracted from current global navigation data structures, such as navigation meshes. The algorithm is relatively simple to implement and has a small run-time cost (approximately 3 μs per agent.) We highlight its performance in different game-like scenarios and observe improved agent behavior and better utilization of free space.

CR Categories: I.6.8 [Simulation and Modeling]: Types of Simulation—Gaming; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Intelligent agents; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—Heuristic methods;

Keywords: local collision avoidance, navigation, non-player characters

1 Introduction

Autonomous agents have become ubiquitous in modern games; non-player characters (NPCs) populate virtual worlds. These NPCs may be passive or antagonistic to the player character. But they have something in common: they all move towards some goal and use navigation techniques to compute collision-free paths.

The most common navigation technique in games is based on combination of global and local navigation techniques [Reynolds 1987; Helbing and Molnar 1995; Van den Berg et al. 2008b; Karamouzas and Overmars 2010]. The global portion is typically handled offline. The free space of the virtual world is computed and stored in a global navigation data structure (e.g. roadmaps, navigation meshes, guidance fields, etc.) When an agent moves from one region of the world to another, these data structures provide an efficient basis for computing a global path.

This global path acts as input to local navigation methods. Local methods are appropriate for driving the agent towards an immediate goal, provided that the obstacles between the agent and that goal are small enough to be resolved with purely local information. At every step in the simulation, an immediate goal for the agent is selected from the path and the local navigation attempts to drive the agent towards that goal while avoiding collisions with other agents and dynamic obstacles.

The immediate goals are typically represented by single points, known as "way points". At any given time in the simulation, an agent has a point goal which it strives to reach. The use of a point goal is computationally efficient but it leads to undesirable artifacts in the simulation.

When agents follow similar paths, they often end up sharing way points. By definition, only one agent can "occupy" a point goal at any time, so the agents are vying for an unsharable resource. This contention gives rise to several undesirable motion artifacts. The only way to resolve the contention is to serialize access, which causes the agents to form a line. In most cases, the contention is artificial. The point goals are coarse approximations of wide portals. Every portion of that portal could be considered equally valid toward reaching a final, global goal.

We seek to reduce, if not eliminate, the source of this contention by eliminating the point approximation of a region. We increase the dimension of the intermediate goal, a way point, from a point goal to a line segment goal, creating a "way portal". The way portal communicates the full size of the portal to which the agent is moving. With greater information, the local navigation algorithm can select collision-free velocities while still making progress through the portal; the agent has more flexibility in responding to dynamic obstacles.

Main Contributions. We present a novel reformulation of local navigation in which an agent seeks to reach, not a single intermediate goal, but one of a continuous set, defined by a line segment. This concept is general and should apply to many types of local navigation algorithms. We show, specifically, how this model can be implemented in a velocity obstacle-based navigation system. To effect this integration, we present a new segment-based optimization algorithm to compute an "optimal" collision-free velocity for each agent. Our formulation includes analysis of error and how to select a single velocity when there is a space of equally feasible velocities. Finally, we show how way portals can be easily extracted from a common global navigation data structure: a navigation mesh. The algorithm is straightforward and is very computationally efficient. We show that the average computaion time, per agent, is approximately 3 μs. Furthermore, the navigation system is sufficiently stable to admit large time steps (as large as 0.2 s.) The small cost and large stability make it ideal for games; navigation work can be done in less than a millisecond every few frames. Finally, we demonstrate how performing navigation with local line segment goals improves the behavior of agents in a complex, dynamic environment.

The rest of this paper is organized as follows. In Section 2 we

*seanc@cs.unc.edu
†snape@cs.unc.edu
‡dm@cs.unc.edu

present related work. We discuss the form line segment goals take in velocity obstacle-based local navigation in Section 3. In Section 4 we provide details and analysis of using line segment goals in practice. We show how line segment goals easily integrate with navigation meshes in Section 5. And finally, we give experimental results and discussion of the impact of line segment goals in Sections 6 and 7.

2 Related Work

The prevalent approach to global navigation in games has been based on roadmaps [Latombe 1991]. In roadmap-based methods, game agents are constrained to the edges of a graph between intermediate goal nodes (way points). Increasingly, navigation meshes [Snook 2000; Kallmann 2010; Van Toll et al. 2011] and similar methods [Pettré et al. 2005; Geraerts et al. 2008] have begun to supplant roadmaps in games. Navigation meshes are a decomposition of the freespace of game world into a mesh consisting of convex polygons. The connectivity of the graph is stored as a graph, similar to roadmaps. Navigation meshes have advantages over roadmaps in that all edges of a polygon are implicitly connected to each other, i.e. because of the convexity there is a straight-line path from any point in the polygon to any boundary. In addition, a single navigation mesh can encode clearance for arbitrarily sized agents. Finding a global path with a navigation mesh consists of searching the connectivity graph for the shortest path between two polygons. The cost of a graph edge between two polygons depends on the length of the shared edge of those two polygons. If the edge is not large enough to accommodate the agent, the cost is infinite. The clearance of the freespace is intrinsically part of the navigation mesh. For these reasons, we focus on navigation meshes over roadmaps.

Algorithms are available to generate navigation meshes, including triangulation algorithms [Hertel and Mehlhorn 1985], space-filling volumes [Tozour 2003], and Recast [Mononen 2009], allow for automatic navigation mesh generation. Frequently, however, roadmaps and navigation meshes are manually defined during level design. Roadmap-based planners have also been adapted to accommodate dynamic environments by reusing previously computed information [Jaillet and Simeon 2004; Kallman and Mataric 2004; Ferguson et al. 2006; Zucker et al. 2007] or integrating dynamic obstacle movement directly into the planner [Hsu et al. 2002].

Alternatives to changing a precomputed roadmap or navigation mesh include potential field planners [Khatib 1986], which use gradient descent to move toward a goal at a sink, inevitable collision states [Petti and Fraichard 2005], which adds a time parameter to allow for dynamic obstacles, and elastic or deformable roadmaps [Sud et al. 2007; Yang and Brock 2007].

Velocity-based methods, such as the reciprocal velocity obstacle [Van den Berg et al. 2008a] and its extensions [Van den Berg et al. 2011; Snape et al. 2011] have exhibited improvements in terms of local collision avoidance and in the behavior of game agents, as well as improved computational performance. However, all of these formulations assume point-based goals positions.

3 Local Navigation and Line Segment Goals

Local navigation methods compute velocities for agents based on strictly local data: local goals and obstacles. There are many methods for performing local navigation (cellular automata, social forces, and velocity obstacles, etc.) In order to incorporate line segment goals into a local navigation algorithm, three issues must be addressed. First, how does the point goal manifests itself in the model? Second, how would that manifestation change if the underlying goal changed from a point to a line segment? And, third, at run-time, determine what portion of the line segment goal is accessible to the agent due to occlusion by dynamic obstacles.

While we believe it is possible to extend many forms of local navigation to include line segment goals, in this paper we focus on one particular type of local navigation model: velocity obstacles. We've selected velocity obstacles for several reasons. First, there has been considerable recent interest in using velocity obstacles and efficient implementations are available as part of a publicly available library. Second, many games (e.g. Warhammer 40,000: Space Marine) are using these techniques for NPCs. Finally, velocity obstacle formulations inherently model space-time occlusion in its planning; i.e. it explicitly encodes the occlusion state over a period of time. Force-based local navigation has also been used in games to avoid collisions. However, the repulsive forces typically have extreme responses in close proximity. This leads to potentially stiff systems which require small simulation time steps. As shown in Section 6, the velocity obstacle approach provides very consistent and stable simulation results even for quite large time steps.

3.1 Velocity Obstacles

Velocity obstacles originated in robotics [Fiorini and Shiller 1998] and correspond to a geometric construct in velocity space. It represents the space of all velocities an agent can take which will lead to a collision with another entity at some point in time (the other entity is assumed to have a constant velocity.) The velocity obstacle is a cone in velocity space. If the computation is modified to limit the obstacle to collisions within a specified time, τ, then the cone becomes a truncated cone (Fig. 1(a).) The cone encodes goal occlusion in a time-dependent manner. The moving obstacle will occlude a particular region for a finite period of time and velocities which cause the agent to arrive at the location sufficiently before or after the obstacle's arrival would be collision free. Thus, a direction that appears occluded at one moment of time may not actually limit the agent's ability to move in that direction because it will no longer be occluded by the time the agent arrives. When velocity obstacles are applied in a multi-agent context, each agent computes a new velocity respecting multiple velocity obstacles–one for each obstacle it seeks to avoid.

Velocity obstacles are used in local navigation by providing a preferred velocity. The velocities outside the union of the velocity obstacles are *feasible* velocities. The planning agent selects a feasible velocity which minimizes some cost function (where the preferred velocity has the globally minimum value.) Unsurprisingly, the preferred velocity is derived from the point goal. The velocity's direction is the direction from the agent towards the goal (Fig. 1(a).) The magnitude of the velocity is some pre-determined preferred speed. For velocity obstacles to be extended to use line segment goals, the preferred velocity must represent the space of velocities which would reach the line segment goal.

3.2 Goal Space to Velocity Space

A point goal implies a single preferred velocity. A line segment goal produces an arc of preferred velocities. Given a line segment goal defined by the end points ($\mathbf{p}_0$, $\mathbf{p}_1$), the representation of that goal in velocity space is a circular arc with radius equal to the preferred speed, v_{pref}, spanning the angle subtended by the goal from the agent's perspective. Specifically, we represent it with two vectors: $\mathbf{v}_0$ and $\mathbf{v}_1$, where

$$\mathbf{v}_i = v_{pref} \frac{\mathbf{p_i} - \mathbf{p_A}}{\|\mathbf{p_i} - \mathbf{p_A}\|}, i \in \{0, 1\} \quad (1)$$

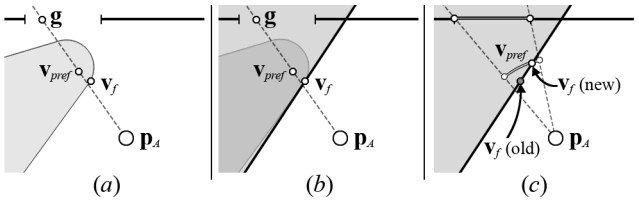


Figure 1: *Computation of optimal feasible velocity using ORCA-constraints using both a point goal and line segment goal. (a) The basic feasible velocity computation using a truncated velocity obstacle with a point goal. (b) The truncated velocity obstacle is bound by an ORCAlinear constraint with a point goal. (c) The ORCAconstraint is used to optimize with respect to a velocity arc. Compared to the feasible velocity from (a) and (b), the new feasible velocity maintains preferred speed by changing direction towards the unoccluded region of the portal.*

for an agent with position $\mathbf{p}_A$. The line segment goal subtends the angle $\theta_G = \cos^{-1}(< \mathbf{v}_0, \mathbf{v}_1 > / v^2_{pref})$. Local navigation would then consist of finding a velocity outside the union of velocity obstacles "near" the velocity arc.

Finding this "near" velocity is problematic. Computing the feasible regions of an arbitrary set of cones is computationally expensive and algorithmically complex. The free space could easily consist of numerous disjoint regions. Secondly, the circular arc isn't convex which means for many cost functions there may not be a unique minimum for selecting the best alternative velocity when the arc lies entirely within velocity obstacles. Finally, in the event that portions of the arc are in the feasible regions, a mechanism must exists for selecting one velocity from the valid set. To address the issues of computational efficiency and optimization, we approximate the arc with its corresponding chord—a velocity segment.

4 Velocity Segment

Van den Berg et al. [2011] proposed an alteration to the canonical representation of velocity obstacles: ORCA. They bound the velocity obstacle with a half plane (Fig. 1(b).) The problem of finding an alternate, feasible velocity becomes an optimization problem with linear constraints. The set of linear constraints define a convex region of feasible velocities and, using a Euclidian-distance metric, the "optimal" feasible velocity is the point on the convex region nearest the preferred velocity (Fig. 1.) Furthermore, the authors recommend a randomized algorithm which can compute this optimal velocity in O(n) time for n constraints.

The velocity *arc* is non-convex and, therefore, cannot be used with this algorithm. By approximating the velocity arc with its corresponding chord (as a *velocity segment*), we can adapt the optimization framework from ORCAto find the optimal velocity with respect to a space of velocities, defined by a line segment. By doing so, we benefit from the computational efficiency and theoretical collision-free guarantees of ORCA, while extending the local navigation to exploit multiple, equally valid immediate trajectories (Fig. 1(c).)

We define the velocity segment as $\mathbf{V} \in \mathbb{R}^6 = [\mathbf{v}_0, t_0, \mathbf{v}_1, t_1]^T$. Where $\mathbf{v}_0$ and $\mathbf{v}_1$ are the end points of the segment as defined in Equation 1 and t_0 and t_1 are corresponding parameter values for the end points (see Section 4.2.)

In this section, we describe the algorithm to optimize with respect to the velocity segment, show how to select a single velocity from a set of feasible velocities, analyze the approximation error, and present additional navigation strategies to exploit the line segment goal formulation.

4.1 Segment-based Optimization

We use a variation of the randomized linear programming algorithm presented in [De Berg et al. 2008]. The algorithm is an iterative algorithm. The algorithm works by maintaining a "best" optimal solution. Each constraint line is processed one at a time.

- If the best optimal solution is still feasible (i.e. both both end points lie in the feasible region) with respect to the new line, no action is taken.
- If the optimal solution is infeasible (i.e. none of the end points lie in the feasible region), in any way, with respect to the new line, we know that the new optimal solution is uniquely defined by the new line.
- If the previous optimal solution is partially feasible (i.e. only one end point lies in the feasible region) with respect to the new constraint, then the resulting optimal solution is clipped by the half plane.

In the case where the best optimal solution is infeasible (even partially) the new optimal solution must lie on the boundary of the new half plane.

The set of constraints we use differs from that used in ORCA. We include two new constraints to the system: half planes aligned with the extents of the velocity arc. This guarantees that any feasible velocity determined leads in the direction of the goal.

Algorithm 1 describes the full iterative process. The description relies on four functions: `notFeasible`, `clipBoundary`, `clipVelocity` and `nearestOnBoundary`.

The function `notFeasible` takes as arguments a velocity space (which can be either a point or a segment) and a single linear constraint and determines if any portion of the velocity space lies in the infeasible region of the linear constraint.

The function `clipBoundary` takes as arguments a set of linear constraints and a line and returns the portion of the line which is in the feasible space of all the linear constraints. The result could be a line, a ray, a segment, a point or the empty set.

When the optimal solution is a velocity segment and the new constraint intersects the segment, then the function `clipVelocity` is called. As the name suggests, the velocity segment is clipped to span the space from the feasible end point to the point of intersection between segment and half-plane boundary.

The function `nearestOnBoundary` computes the region on the clipped boundary closest to a the optimization velocity space. The closest region is typically a point, but if the boundary and optimization segment are parallel, the closest region could, in turn, be a segment. It's also possible for the optimal solution to be a single point but by adding another constraint, the subsequent optimal solution is again a line segment.

The algorithm finds the optimal space of velocities with respect to the preferred velocity and the set of linear constraints. This space may be a single velocity or a velocity segment. In the case of a velocity segment, a single velocity must be selected from the space.

4.2 Velocity Bias

Velocity bias determines which specific velocity, from a continuous feasible velocity segment, the agent will take. The name arises from the idea that even when there is a space of equally reasonable

Algorithm 1: *Compute the best velocity with repsect to a preferred velocity space and a set of half planes represented by lines.*

input : A set of Lines, $lines$, representing half planes, and a VelocitySpace, $velOpt$, the space of preferred velocities.
output: A VelocitySpace, possibly empty, representing the best feasible velocities.

```
prevLines = {}
best ← velOpt
while |lines| > 0 do
    line ← getRandomLine( lines )
    lines = lines − {line}
    if notFeasible( line, best ) then
        boundary = clipBoundary( line, prevLines )
        if boundary ≠ ∅ then
            if best ⋂ boundary = ∅ then
                best ← nearestOnBoundary( boundary,
                velOpt )
            else
                best ← clipVelocity( boundary, best )
        else
            best ← ∅
            break
        prevLines = prevLines ∪ {line}
return best
```

velocities available, the agent is *biased* towards one in particular. We've defined a bias function, $\beta(\mathbf{V})$, such that for any single subsection of the segment, the agent will have the most bias towards a single, unambiguous velocity (e.g. it has a single extremum on the domain of the segment.) Furthermore, the bias value is used for global navigation. By setting the bias function towards the global path, the agent will follow the global path when there are no dynamic obstacles.

Parameter bias We have defined the velocity segment with parameter values for the end points. When the preferred velocity segment is initialized, t_0 and t_1 are assigned the values zero and one, respectively. During optimization, when the solution is truncated or projected, the parameters from the corresponding points on the preferred velocity are stored with the best solution. The bias velocity is computed in the following manner:

$$\begin{aligned} \beta(\mathbf{V}) &= (1-\alpha)\mathbf{v}_0 + \alpha\mathbf{v}_1 \\ \alpha &= \left(\left[\min_{t\in[t_0,t_1]} |t-\zeta|\right] - t_0\right)/(t_1 - t_0), \end{aligned} \tag{2}$$

where $\zeta \in [0,1]$ is the bias value. When ζ is 0.5, the agent prefers to walk through the middle of the portal. As ζ deviates from 0.5, the agent is biased towards one end of the portal or the other. This formulation satisfies the requirements of being well behaved. The function has a single global minimum at ζ. Furthermore, if ζ is set to be the parameter value of where the global path crosses the portal, the agent will be biased to follow the global path.

4.3 Segment-Arc Error

A velocity arc and its corresponding velocity segment span the same space of velocity *directions*. In the arc, all vectors have uniform magnitude. In the segment, the magnitude varies, greatest at the ends and smallest at the center. By approximating the arc with a segment, we introduce speed error. The maximum amount of speed error, ε, found in the center of the arc spanning θ° at a speed of v_{pref}, is equal to:

$$\varepsilon = v_{pref}(1 - \cos(\theta/2)) \tag{3}$$

For a fixed speed, as the angle increases, the maximum error increases. And, similarly, for a fixed angle, as the speed increases the maximum error likewise increases. By optimizing with respect to the segment, we may end up selecting a velocity vector whose speed is significantly different from the preferred speed. At the limit, where the arc spans 180°, the maximum error is exactly equal to the preferred speed because the segment cuts through the center of the circle; the zero velocity is considered to be a preferred velocity. This case is quite common; the arc angle goes to 180° as the distance to the portal goes to zero. The effect of this error is that, as the agent gets closer to an uncongested portal, it gradually slows to a stop.

We address this by bounding the error. We can bound the error in two ways: relative bound and absolute bound. The relative bound limits the speed error to some fraction of the preferred speed: $\varepsilon_R(\mathbf{V}) \leq \sigma v_{pref}$. The absolute bound limits the speed error to a fixed amount, regardless of speed: $\varepsilon_A(\mathbf{V}) = \sigma$. The two bounding strategies have different effects.

For relative error, the approximation error is guaranteed to be within the allowable error when the following expression is satisfied:

$$\cos(\theta) \geq 2(1-\sigma)^2 - 1. \tag{4}$$

The relative bound defines a *constant* limit on the arc size. Regardless of what the preferred speed may be, the maximum arc angle is constant. For a fixed value of σ, the maximum arc angle allowed is $\theta_{\max} = \cos^{-1}(2(1-\sigma)^2 - 1)$.

Similarly, for absolute error, the following condition guarantees allowable error:

$$\cos(\theta) \geq 2\left(1 - \frac{\sigma}{v_{pref}}\right)^2 - 1 \tag{5}$$

The constraint on the arc angle defined by the absolute error bound depends on the preferred speed. At high speeds, the arc must be narrower to satisfy the error constraint, but the viable angle increases with lower preferred speeds. When the preferred speed falls below σ, the arc is allowed to span the full 180°. For a fixed value for σ, the maximum angle allowable velocity arc is $\theta_{\max} = \cos^{-1}(2(1-\sigma/v_{pref})^2 - 1)$.

4.4 Arc Contraction

If the arc exceeds the angle allowed by the error bound, the arc must be contracted to an acceptable angle. However, there are an infinite number of arcs of the reduced angle which are subsets of the full arc. We require a strategy to select a particular arc.

Our contraction strategy uses a *contraction* vector. Given a contraction vector, we reduce the span of the arc, proportionately, towards the contraction vector. We compute how much the arc needs to contract, ϕ. Given the contraction vector $N = (N_x, N_y)$, we then compute an angle of rotation, R_0 and R_1, for the vectors that bound the velocity space, $\mathbf{v}_0$ and $\mathbf{v}_1$. respectively.

$$R_0 = \begin{cases} 0^\circ & \text{if } x_0N_y - y_0N_x > 0 \\ \phi - \theta & \text{if } x_1N_y - y_1N_x < 0 \\ -\frac{\phi\theta_0}{\theta} & \text{otherwise} \end{cases} \tag{6}$$

$$R_1 = \begin{cases} \theta - \phi & \text{if } x_0N_y - y_0N_x < 0 \\ 0^\circ & \text{if } x_1N_y - y_1N_x > 0 \\ \frac{\phi\theta_1}{\theta} & \text{otherwise} \end{cases} \tag{7}$$

We define the contraction vector based on the bias function. Using the parameter bias function, $\mathbf{N} = (1-\zeta)\mathbf{v}_0 + \zeta\mathbf{v}_1$.

4.5 Arc Expansion

As the distance to the portal increases, it subtends an increasingly smaller angle. It is tempting to define a threshhold, below which the portal no longer defines an arc, but collapses down to a point. This would increase computational efficiency; optimizing velocity with respect to a point is cheaper than for a segment. However, it is contrary to the underlying motivation of using a line segment goal.

We use a line segment goal so that, as the agent responds to dynamic obstacles, the local navigation doesn't artifically increase contention by approximating a goal region as a goal point. Collapsing a small arc into a point returns the agent to the overly constrained goal model.

When an agent is a great distance from its goal, perturbations in the agent's trajectory towards the goal have little significance. For example, for an agent 100 m away from its goal, taking a 1 m detour perpendicular to the direction toward the goal and then walking directly towards the goal only increases the total distance travelled by 1%. If this 1 m detour helps the agent avoid a congested region, the actual time to the goal may improve. To exploit this, we expand the arc when the agent is far from the goal.

We model this in the following manner. An agent at distance d from a goal can walk a distance, Δ, in a direction α degrees from the goal direction. After moving that distance, the agent would return to walking straight toward the goal over a new distance d'. To limit the magnitude of the detour, we define a maximum deviation factor, $\gamma > 1$ and require $\Delta + d' \leq \gamma d$. The angle to which we want to expand the velocity arc is $\theta_E = 2\alpha, \theta_E \in [0, \theta_{\max}]$.

For a given detour amount (Δ) and deviation factor (γ), the expansion arc reaches its maximum angle ($\theta_{\max}$) at:

$$d_{\max} = \frac{2\Delta[\cos(\theta_{\max}/2) - \gamma]}{1 - \gamma^2}, \tag{8}$$

and reaches zero at distance Δ. So, we define the expansion angle as:

$$\theta_E = \begin{cases} 0 & \text{if } d \leq \Delta \\ 2\cos^{-1}\left(\frac{2\Delta\gamma + d - \gamma^2 d}{2\Delta}\right) & \text{if } \Delta < d \leq d_{\max} \\ \theta_{max} & \text{if } d > d_{\max} \end{cases} \tag{9}$$

Finally, we compute the actual angle for the velocity arc as follows:

$$\theta = \min(\theta_{\max}, \max(\theta_E, \theta_G)). \tag{10}$$

If the goal arc requires expansion, we expand in the same manner as we contracted. However, for expansion, the expansion normal is the middle of the goal and *not* the bias direction. This provides a symmetric increase in the appearance of the goal.

5 Global Navigation

Efficiently performing local navigation with line segment goals is useful only if the global navigation algorithm can produce a path in which this goal information is encoded. In this section, we describe the path used with line segment goals and how it is computed from global navigation data structures such as navigation meshes.

5.1 Waypoints and Way Portals

An agent point-goal path can be generalized as the function $\Pi : \mathbb{A} \rightarrow \mathbb{R}^2$, where $\mathbb{A}$ is the space of agent configuration state. Essentially, it examines the agent's current state and produces an immediate goal point in $\mathbb{R}^2$. These immediate goals are often called "way points". To produce line segment goals, we simply need to map to a higher dimension: $\Pi_1 : \mathbb{A} \rightarrow \mathbb{R}^4$. The new path maps the current agent state into a way *portal*, $\pi = [\mathbf{p}, \mathbf{d}]^T, \mathbf{p}, \mathbf{d} \in \mathbb{R}^2$. The way portal serves as the line segment goal and spans the space $\mathbf{p} + t\mathbf{d}$, where $t \in [0, 1]$.

In practice, computing a smooth, high-order function for the path is challenging. Instead, Π is usually approximated by a set of discrete way points. The way points are located at critical points in the static environment, such as in doorways or portals. The discrete point-goal path is $[\mathbf{s}, \mathbf{p_0}, \mathbf{p_1}, ..., \mathbf{p_N}, \mathbf{g}]$, where, $\mathbf{s}$ and $\mathbf{g}$ are the start and goal positions, respectively. In the equivalent discrete way-portal path, we replace the intermediate way points with way portals: $[\mathbf{s}, \pi_0, \pi_1, ..., \pi_N, \mathbf{g}]$.

The way-portal path does not uniquely define a single path as the point-goal path does. The way-portal path defines a space of paths. When a single agent traverses the environment, one single path must be selected from the space. For a way portal path with N portals, a single path is uniquely described by $\mathbf{t} \in \mathbb{R}^{N+1}$. Each t_i corresponds with a position along the length of each portal. Traditionally, the desired path is the shortest path. To find the shortest path, Π_1^*, we would find the vector $\mathbf{t}^*$ that minimizes the total path length subject to the constraint, $t_i \in [0, 1]$.

$$\begin{aligned} t^* \quad = \quad & \min_{t \in \mathbb{R}^{N+1}} (\|\mathbf{s} - \mathbf{p}_0 + t_0\mathbf{d}_0\| + \\ & \sum_{i=0}^{N-1} \|\mathbf{p}_i + t_i\mathbf{d}_i - \mathbf{p}_{i+1} + t_{i+1}\mathbf{d}_{i+1}\| + \\ & \|\mathbf{p}_N + t_N\mathbf{d}_N - \mathbf{g}\|) \end{aligned} \tag{11}$$

The values in $\mathbf{t}$ would serve as the values for ζ in the bias function. As the agent heads to portal, π_i, the bias function would be set to $\zeta = t_i$. However, this is a tightly-coupled, non-linear system. Finding the optimal shortest path is infeasible for real-time applications. So, we use a simple heuristic.

5.2 Path Planning

We want to compute a value for the bias parameter, ζ_i for crossing portal π_i such that the path the agent follows approximates the shortest path through the portals. We apply a simple, O(1) heuristic for defining this value. We compute a line between the agent's current position and the center of portal $i + k$. We then intersect the line with portal p_i. The value of ζ_i is such that $\mathbf{p}_i + \zeta_i\mathbf{d}_i$ is the intersection point. If the line does not intersect the portal, then ζ_i is 0 or 1, depending on which end of the portal is closest to the line.

The efficacy of this approach depends on the choice for k. The best choice for k would be a dynamic selection strategy. One should choose $k > i$ such that π_k is the first portal that is not visible from the agent's position $\vec{p}_A$. This is the "right" choice because the direction of the unseen portal communicates the most information about turning corners. However, this requires frequent visibility queries. A system-wide constant value for k would make the heuristic very cheap. But the right choice depends on the scene in which it is applied. In practice, we find two is a good value. Two portals ahead is sufficient to optimally turn right-angled corners in the environment.

As the agent progresses along the path, each time it passes one way portal and selects the next to serve as its immediate goal, it performs this heuristic to determine where the agent should cross the new way portal.

Approach vector In the absence of dynamic obstacles, the agent will walk towards the bias point. The presence of dynamic obstacles can cause the agent to deviate from its path. In principle, this shouldn't make a difference because as the agent nears the portal,

the arc subtended by the portal converts to 180°. However, because of error bounds, the angle is limited to a much smaller arc (e.g. $\sim 60^\circ$ for a relative error bound of 0.1.) We introduce the concept of approach vector. When the agent first computes ζ_i, it also computes the initial approach vector to the bias point, $\mathbf{p}_i + \zeta_i \mathbf{d}_i$. While making its way to the portal, if the agent's approach vector to the bias point exceeds some threshold, the heuristic is applied again and a new value for ζ_i is computed.

5.3 Navigation Mesh

There are many ways to decompose the simulation domain to facilitate global path planning: roadmaps, guidance fields, navigation meshes, etc. Any of these can be modified to produce way portal paths instead of way point paths, but navigations meshes are the most natural fit. Navigation meshes are also ideally suited to simulate agents with different sizes and motion constraints.

A navigation mesh decomposes the free space into a mesh of convex polygons. The edges of each polygon admits one of two interpretations. Either an edge is an exterior edge, in which case it represents a wall of the environment, or it is an interior edge and serves as a portal between the two adjacent polygons. Because the polygons in the mesh are convex, an agent inside any given polygon can take a straight-line path towards any interior edge without encountering static obstacles. These interior edges naturally encode the portal data required to form a way portal path. And, conversely, the exterior edges serve as obstacles in terms of local collision avoidance computation using velocity obstacles.

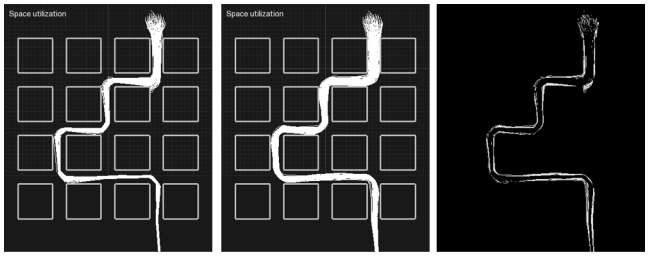
The connectivity of the mesh implies a graph where each polgyon is a node and an edge exists between two nodes if they share an edge. The way portal path is generated in the following fashion:

1. Compute the start node by determining which polygon the agent is inside.
2. Compute the goal node by determining which polygon the goal position lies inside.
3. Perform an A* algorithm on the graph between the start and goal nodes.
   (a) If the edge length is shorter than agent diameter, $2r_A$, edge cost is infinite.
4. For each interior edge, $e = (\mathbf{q}_0, \mathbf{q}_1)$, $\|e\| = \|\mathbf{q}_1 - \mathbf{q}_0\|$, $\hat{\mathbf{e}} = \frac{\mathbf{q}_1 - \mathbf{q}_0}{\|e\|}$, traversed in the path computed by A*, create the way portal, $\pi_i = (\mathbf{q}_0 + r_A\hat{\mathbf{e}}, \hat{\mathbf{e}}(\|e\| - 2r_A))$.

6 Results

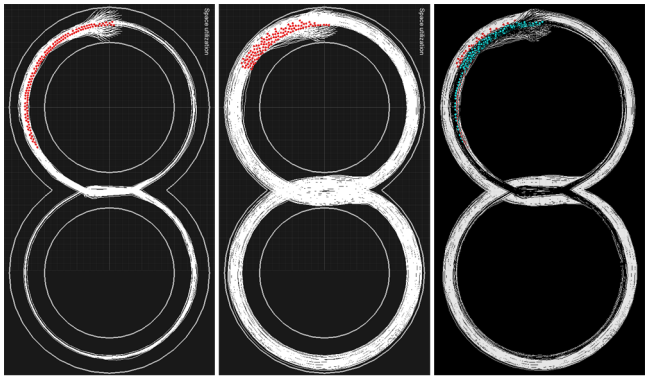
In this section, we demonstrate the beneftis of our model by applying various metrics to some representative benchmarks. We will show the following benefits with the indicated metrics. First, the algorithm is very efficient; planning for hundreds of agents simultaneously takes very little time. In each scenario we will show the average computation time for the local navigation algorithm, comparing it to the cost of the corresponding point-goal algorithm. Second, the simulation results are consistent and stable even with large time steps. To illustrate this point, we've run simulations on the scenarios with widely varying time steps. In each case, we'll show that the time on the path is equivalent and that the rate that collisions occur remains the same, even as the time step grows. Finally, line segment goals reduce the artifcial contention between agents which cause them to converge to a line. We show graphs of the environments which show the increased space utilization.

The example scenarios are as follows (see video for full detail):



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

Figure 2: *The space used by agents performing traversing the 16 block scenario. The white regions indicate the aggregate paths of the agents. (a) The paths using point goals. (b) The paths using line segment goals. (c) The regions used due to line segment goals over point goals.*



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

Figure 3: *The space used by agents performing traversing the infinity loop scenario. The interpretation of the figures is the same as that in Fig. 2.*

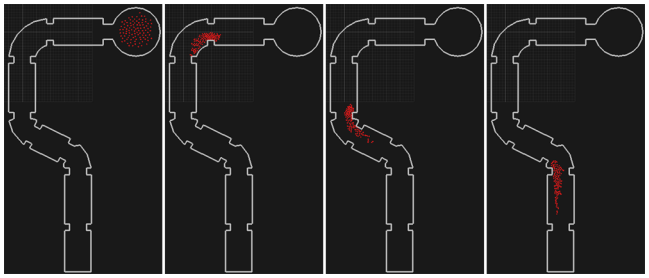


Figure 4: *Four snapshots of the simulation of 97 agents passing through a dungeon scene. The wall's protrusions are handled well by the agents using way portals.*

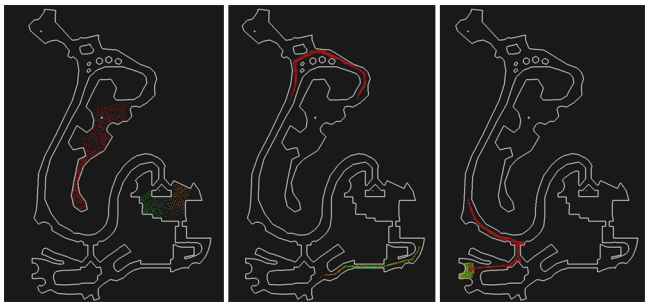


Figure 5: *Three moments from the castle scenario. 713 agents move through a fortress towards the central courtyard.*

Table 1: *Cost of line segment goal local navigation. Increasing from a point goal to a line segment goal is a small cost, less than 1 μs per agent.*

Scenario	Point Goal	Line Segment Goal	
	Average cost	Average Cost	Per-agent average
16 Blocks	0.24 ms	0.28 ms	3.3 μs
Infinity	0.26 ms	0.36 ms	3.0 μs
Dungeon	0.28 ms	0.32 ms	3.3 μs
Castle	1.46 ms	1.6 ms	2.2 μs

1. **16 Blocks**. 85 agents travel a serpentine path through 16 blocks laid out in a uniform array (see Fig. 2.) The agents in this scenario must take frequent 90-degree turns. This type of motion makes maintaining cohesion difficult.

2. **Infinity loop**. 119 agents traverse a loop in the shape of the infinity symbol (see Fig. 3.) The group of agents represents a group of agents traversing a long path without significant turns.

3. **Dungeon**. 97 agents pass through a tunnel-like dungeon (inspired by [Mononen 2009].) The tunnel has many local minima–protusions from the walls act to obstruct agents (see Fig. 4.)

4. **Castle**. 713 agents walk paths towards a fortress courtyard (see Fig. 5.) This illustrates large numbers of agents traversing an authentic game environment.

Computational efficiency. The linear-constraint optimization model and algorithm prove to be very efficient. For each of the four scenarios, we have computed the average computation time, per simulation step, to compute preferred velocity, optimal velocity and to update the agent's state. Table 1 highlights these results. The computaiton is done in parallel using a simple OpenMP loop. A larger number of agents better exploits the parallelism, exhibiting a lower *per-agent* average, as shown by the Castle scenario. While optimizing with respect to the velocity segment incurs a greater cost than for a single velocity, the per-frame cost is still quite small.

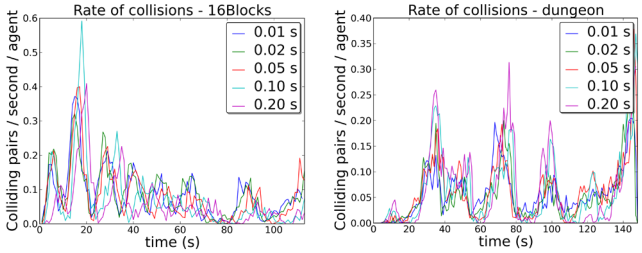
Stability To illustrate the stability of the simulation we compute the global time required for the last agent to reach the region enclosing the final goal position. This measures the consistency of the simulation across different time steps. The last agent's progress is constrained by every previous agent. A consistent final time indicates that the simulation produces the same simulation result across varying time steps. Table 2 shows this consistency. The video allows for a more compelling, qualitative examination of the impact of different time steps.

Another measure of stability is the number of collisions that occur in the simulation. An unstable simulation will lead to more collisions; the underlying assumptions to determine a collision-free velocity are no longer valid in unstable conditions. Fig. 6 shows the number of collisions (normalized for number of frames and number of agents) across multiple time steps. To save space, data from only two scenarios is shown. The data shows that the larger time steps produce collisions at a lower rate. In Fig. 6(a) the spikes in collisions occur as a large portion of the agents negotiate a turn. Similarly, in Fig. 6(b), those spikes map to the moments when the mass of agents pass a protrusion from the wall.

Space utilization. Changing the point goal to a line segment goal, we reduce artificial contention. Agents no longer converge to a line because there is a space of equally viable velocities which take the agent towards their goal. Figures 2 and 3 illustrate the importance of the line goal segment. Figure (a) shows the accumulated paths

Table 2: *The simulation time required for the last agent to reach the final goal region for varying simulation time steps.*

Scenario	Simulation Time Step				
	0.01 s	**0.02 s**	**0.05 s**	**0.1 s**	**0.2 s**
16 Blocks	104.6 s	105.4 s	107.7 s	110.3 s	114.8 s
Infinity	210.7 s	211.4 s	212.3 s	214.0 s	215.6 s
Dungeon	138.0 s	137.8 s	138.7 s	140.3 s	139.8 s



(a) (b)

Figure 6: *The rate of collisions for two scenarios over a range of simulation time steps (color in original). The number of collisions is normalized across time step by multiplying by the time step size. (a) 16 Block scenario. (b) Dungeon.*

of the agents using point goals. Similarly, (b) shows the paths of the agents using line segment goals. Finally, (c) shows the the difference in these two regions. The white regions in (c) are those regions the agents take using line segment goals that they did not take using point goals. The 16 block demo is quite constrained, so the difference is smaller, but the agents use the full corridor space. In the infinity loop, the agents take up a much larger swath of the space.

Planning. During our experiments we came across a noteworthy phenomenon. When the intent is to have a group of agents traversing a space as a unit, the bias values should remain 0.5. The group of agents can be thought of as a meta entity with a larger radius. This larger radius needs greater clearance and a value of 0.5 gives maximum clearance. When each individual member of the group plans a shortest path as if they were alone, the center of mass of the group seeks to hug the walls. This causes the group to be spread out, as if material were being worn off the meta entity through friction with the wall. This phenomenon affects both point-goal and line segment-goal local navigation (although the point-goal suffers more.)

7 Conclusions and Future Work

We've described a model for improving multi-agent navigation. Using point goals in a local navigation algorithm causes needless contention between agents vying for the same infinitisemally small goal point. The result of this contention is the well-known pheonomenon in which agents converge towards a line. This behavior is unrealistic and destroys the illusion that the agents form any kind of cohesive group.

By performing local navigation with line segment goals, the navigation algorithm has more flexibility in choosing appropriate collision-free velocities. The line segment goals better approximate the true intermediate goals in game maps. Seldom do agents truly need to stand at a single point. Instead, an agent passes through regions on its way to a final goal position. Any point on this intermediate region is basically equivalent for reaching the final goal. Navigation meshes perfectly encode this information. The interior

edges translate perfectly into way portals and this information can be fed directly to local navigation. Finally, the stability and efficiency of this approach would allow a game to perform planning at very low frequency (5-10 Hz) at less that a millisecond for even hundreds of agents. Our preliminary results are promising; groups of agents maintain far better cohesion and use the space more effectively.

Our approach has limitations. Bounding the error introduced by approximating an arc with a line segment limits the span of velocities available to the agent. This, in turn, limits the full impact that line segment goals can have in this formulation. Furthermore, there are artifacts which arise from the underlying local navigation model (the group of agents arrays itself to the outside of curves due to inertia in the ORCAformulation. See Fig. 3.)

This approach is general, in the future we look forward to applying it to other local navigation models (e.g. social forces) as well as other global navigation data structures. Applying it to roadmaps is trivial as a roadmap can, in many ways, be considered the dual of the navigation mesh. These techniques also apply to structures like guidance fields and, in fact, applying the concept of non-point goals may make the calculation of guidance fields simpler.

Acknowledgements

This research is supported in part by ARO Contract W911NF-04-1-0088, NSF awards 0917040, 0904990, 100057 and 1117127, and Intel.

References

DE BERG, M., CHEONG, O., VAN KREVELD, M., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer, Heidelberg.

FERGUSON, D., KALRA, N., AND STENTZ, A. 2006. Replanning with RRTs. In *Proc. IEEE Int. Conf. Robot. Autom.*, 1243–1248.

FIORINI, P., AND SHILLER, Z. 1998. Motion planning in dynamic environments using velocity obstacles. *Int. J. Robot. Res. 17*, 760–772.

GERAERTS, R., KAMPHUIS, A., KARAMOUZAS, I., AND OVERMARS, M. 2008. Using the corridor map method for path planning for a large number of characters. In *Motion in Games*. Springer, Heidelberg, 11–22.

HELBING, D., AND MOLNAR, P. 1995. Social force model for pedestrian dynamics. *Phys. Rev. E 51*, 4282–4286.

HERTEL, S., AND MEHLHORN, K. 1985. Fast triangulation of the plane with respect to simple polygons. *Inform. Control 64*, 52–76.

HSU, D., KINDEL, R., LATOMBE, J.-C., AND ROCK, S. 2002. Randomized kinodynamic motion planning with moving obstacles. *Int. J. Robot. Res. 21*, 233–255.

JAILLET, L., AND SIMEON, T. 2004. A PRM-based motion planning for dynamically changing environments. In *Proc. IEEE RSJ Int. Conf. Intell. Robot. Syst.*, vol. 2, 1606–1611.

KALLMAN, M., AND MATARIC, M. 2004. Motion planning using dynamic roadmaps. In *Proc. IEEE Int. Conf. Robot. Autom.*, vol. 5, 4399–4404.

KALLMANN, M. 2010. Shortest paths with arbitrary clearance from navigation meshes. In *Proc. ACM SIGGRAPH Eurographics Symp. Comput. Animat.*, 159–168.

KARAMOUZAS, I., AND OVERMARS, M. 2010. Simulating the local behaviour of small pedestrian groups. In *Proc. ACM Symp. Virtual Real. Softw. Tech.*, 183–190.

KHATIB, O. 1986. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Robot. Res. 5*, 90–98.

LATOMBE, J.-C. 1991. *Robot Motion Planning*. Springer, Heidelberg.

MONONEN, M., 2009. Recast: navigation-mesh construction toolset for games. `http://code.google.com/p/recastnavigation/`.

PETTI, S., AND FRAICHARD, T. 2005. Safe motion planning in dynamic environments. In *Proc. IEEE RSJ Int. Conf. Intell. Robot. Syst.*, 2210–2215.

PETTRÉ, J., LAUMOND, J.-P., AND THALMANN, D. 2005. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *Proc. Int. Workshop Crowd Simul.*

REYNOLDS, C. 1987. Flocks, herds and schools: a distributed behavioral model. In *Proc. ACM Int. Conf. Comput. Graph. Interact. Tech.*, 25–34.

SNAPE, J., VAN DEN BERG, J., GUY, S., AND MANOCHA, D. 2011. The hybrid reciprocal velocity obstacle. *IEEE T. Robot. 27*, 696–706.

SNOOK, G. 2000. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*. Charles River, Hingham, Mass., ch. 3, 288–304.

SUD, A., GAYLE, R., ANDERSEN, E., GUY, S., LIN, M., AND MANOCHA, D. 2007. Real-time navigation of independent agents using adaptive roadmaps. In *Proc. ACM Symp. Virtual Real. Softw. Tech.*, 99–106.

TOZOUR, P. 2003. Search space representations. In *AI Game Programming Wisdom 2*. Charles River, Hingham, Mass., ch. 2, 85–102.

VAN DEN BERG, J., LIN, M., AND MANOCHA, D. 2008. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proc. IEEE Int. Conf. Robot. Autom.*, 1928–1935.

VAN DEN BERG, J., PATIL, S., SEWALL, J., MANOCHA, D., AND LIN, M. 2008. Interactive navigation of multiple agents in crowded environments. In *Proc. Symp. Interact. 3D Graph. Game.*, 139–147.

VAN DEN BERG, J., GUY, S., LIN, M., AND MANOCHA, D. 2011. Reciprocal n-body collision avoidance. In *Robotics Research: The 14th International Symposium ISRR*. Springer, Heidelberg, 3–19.

VAN TOLL, W., COOK, IV, A., AND GERAERTS, R. 2011. Navigation meshes for realistic multi-layered environments. In *Proc. IEEE RSJ Int. Conf. Intell. Robot. Syst.*, 3526–3532.

YANG, Y., AND BROCK, O. 2007. Elastic roadmaps: globally task-consistent motion for autonomous mobile manipulation. In *Proc. Robot. Sci. Syst.*, 279–286.

ZUCKER, M., KUFFNER, J., AND BRANICKY, M. 2007. Multipartite RRTs for rapid replanning in dynamic environments. In *Proc. IEEE Int. Conf. Robot. Autom.*, 1603–1609.