Towards Believable Crowds: A Generic Multi-Level Framework for Agent Navigation

Wouter van Toll*, Norman Jaklin† and Roland Geraerts‡
Utrecht University
Department of Information and Computing Sciences
Princetonplein 5, 3584 CC Utrecht
E-mail: *W.G.vanToll@uu.nl, †N.S.Jaklin@uu.nl, ‡R.J.Geraerts@uu.nl

***Abstract*—Path planning and crowd simulation are important computational tasks in computer games and applications of high social relevance, such as crowd management and safety training. Virtual characters (agents) need to autonomously find a path from their current position to a designated goal position. This is usually solved by running the A* algorithm on a grid or a navigation mesh. However, in many modern applications, strictly traversing the resulting path is not sufficient. Agents need to be able to deviate from these paths, e.g. to avoid each other or react to dynamic changes in the environment. Multiple *levels* of planning are necessary to efficiently simulate realistic behavior, and the underlying data structures and algorithms should support those levels. Many existing crowd simulation frameworks do not have this flexibility.**

In this paper, we propose a five-level hierarchy for agent navigation in virtual environments. The five levels are high-level planning, global route planning, route following, local movement, and animation. The three center levels concern *geometric* planning and require a *navigation mesh* that represents the navigable space of the environment. We describe an efficient and flexible navigation mesh for 2D and multi-layered 3D environments. We also present our crowd simulation software that uses this mesh; we outline its architecture and show that the framework is easily extendible. Finally, we show that our software can simulate large autonomous crowds in real-time.

I. Introduction

Path planning and crowd simulation are fundamental AI problems in computer games, and they are becoming increasingly important for serious applications such as crowd management, evacuation studies, and safety training. In these applications, virtual characters (or *agents*) need to autonomously find and traverse paths through the environment. Agents should act in a realistic manner: their trajectories must be short and smooth, there should not be any collisions between agents, and the agents are typically expected to mimic human behavior. Furthermore, the simulation should be efficient even if the crowd is very large or dense. In short, agents need to perform multiple tasks that reach beyond a simple path planning algorithm. As such, a crowd simulation system requires multiple *levels* of planning.

In this paper, we propose a generic five-level hierarchy for solving agent navigation problems, and we present our algorithms and implementations of the three center levels. A preliminary version of the hierarchy has been outlined in previous work [1]. We improve upon this work by outlining gaps in existing frameworks, by emphasizing the need for a surface-based representation (a *navigation mesh*) of the environment, by describing the architecture of our generic crowd simulation software, and by showing the software's functionalities and efficiency in more detail.

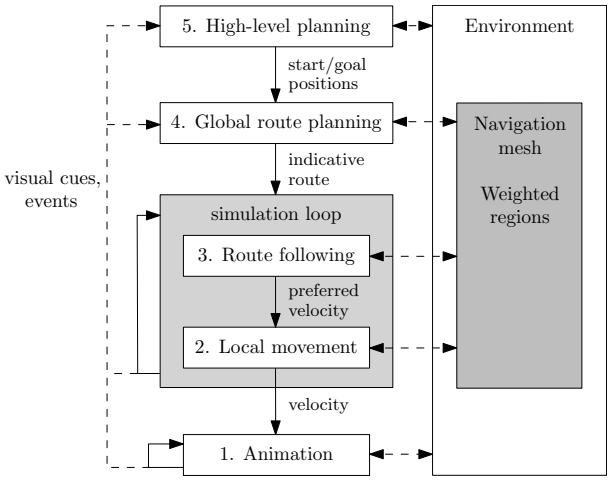


Fig. 1. A five-level hierarchy for path planning and crowd simulation systems. The geometric levels, which can be solved using our algorithms and software, require an efficient *navigation mesh* of the environment.

Figure 1 shows an overview of the hierarchy. Using a single agent as an example, the levels can be summarized as follows:

- *High-level planning* uses AI techniques to translate a semantic action (e.g. 'go home') to one or more geometric queries ('find a path from position s to position g').
- *Global planning* computes an *indicative route*, i.e. a path from s to g that should be roughly followed.
- The three lower levels update the agent in every step of the simulation loop. *Path following* lets the agent choose a *preferred velocity* such that it follows the indicative route. Note that a velocity is a 2D vector that encodes both *speed* and *direction*.
- *Local movement* computes a velocity that deals with local hazards, e.g. to prevent collisions with other agents, while remaining close to the preferred velocity. The simulation then applies this velocity through time integration.
- Finally, *animation* handles the visual movement of the agent, down to its 3D skeleton representation.

This paper focuses on the *geometric* aspects of planning,

Fig. 2. Example of a crowd simulation in a university-like environment. This simulation was generated using our software in combination with Unity3D.

i.e. the tasks imposed by levels 4, 3, and 2. To solve these tasks efficiently, we argue that a navigation mesh is necessary.

II. Related Frameworks

Path planning in the context of game AI has been studied extensively [2], and all aspects of crowd simulation are active research topics [3]. We will refer to related work on specific aspects in Section III.

Several frameworks exist for crowd simulation and analysis in serious applications, such as training and evacuation studies [4]–[9]. One of these uses our ECM software as a black box for geometric planning [7]; thus, our software is already being used in the simulation industry. Most other simulaton frameworks require more manual work, as they do not automatically compute an efficient navigation data structure. In the entertainment industry, the Unity3D game engine has recently adopted the Recast and Detour systems for automatic navigation meshes and agent simulation [10], whereas Golaem Crowd [11] has shifted its focus to high-quality plug-and-play crowds for entertainment applications. Massive [5] is used for crowd generation in many movies and games.

In the research community, SteerSuite [12] is used for simulating and evaluating local movement. ADAPT [13] is a platform for developing agent behavior with an emphasis on animation. SimPed and NOMAD are models for passenger flows, based on real-world observations [14], [15]. The work closest to our own is Menge [16], a system in development that uses a multi-level hierarchy similar to ours [1].

Our work differs in that it presents a more generic solution for the *geometric* aspects of path planning and crowd simulation. First, our navigation mesh has many advantages, such as a small memory footprint, fast query times, independence of agent sizes, and support for dynamic environments. Second, we treat route following as a separate level for better flexibility. Third, we include specialized algorithms for route planning and route following in weighted regions.

III. A Multi-level Planning Hierarchy

As mentioned, a crowd simulation system comprises more than just path planning, which highlights the need for multiple *levels* of planning. In this section, we further describe the five levels of planning proposed in Figure 1. We also list the most important related work in each level.

A. High-Level Planning

At the top of the hierarchy, high-level planning (level 5) translates the desired *semantic* behavior of a agent to *geometric* path planning problems. First, an agent's abstract task such as 'take the train to work' can be converted to a list of more concrete tasks, e.g. 'go to the train station, buy a train ticket, go to the correct platform, enter the train', and so on. Based on this plan, the agent should compute an ordered list of goal positions [17].

High-level planning is a research topic of its own, involving techniques such as *STRIPS* [18] and *Hierarchical Task Networks* [19]. Cognitive decision-making models have also been applied to crowd simulation [17], [20]. In the remainder of this paper, we focus on geometric planning, and we deliberately treat high-level planning as a black box. Hence, our geometric software framework can be plugged into any simulation system that produces specific start and goal positions for an agent.

B. Global Route Planning

Next, global route planning (level 4) uses the agent's current goal position to compute a geometric route through the environment. We refer to the result as an *indicative route*, since it is a preliminary indication of how the agent should move. Having an indicative route that is followed roughly (instead of a path that is followed exactly) yields greater flexibility in the lower levels of the hierarchy. An indicative route can be any curve through the walkable space. In practice, it is often a piecewise linear curve given by a sequence of bending points.

The A* search algorithm [21] is a valuable method for solving global planning problems: it computes an optimal (e.g. shortest) route between two arbitrary query points in any graph structure. To use this algorithm for crowd simulation, a simplified representation of the environment is required. Section IV will show why a navigation mesh is most suitable.

It is common to look for a *short* route through the environment, or a route that stays on the left or right side of the walkable space while keeping some distance to obstacles [22]. Routes can also be computed based on other criteria. For instance, one could map *crowd density* information onto the graph to let agents prefer routes that are less congested, which automatically spreads a crowd over multiple routes [23]. Alternatively, *visibility* information can be used to plan stealthy routes along which an agent is not seen by others [24]. Linear programming techniques can simulate global coordination between groups of agents [25]. The environment can also contain *weighted regions* for which an agent has personal preferences. For example, a pedestrian might prefer to walk on the sidewalk while avoiding roads, puddles or muddy terrain. Planning optimal global routes in such environments is computationally difficult, but provably good approximating techniques exist [26]. See Figure 3 for another example.

C. Route Following

Route following (level 3) ensures that the agent follows the indicative route π_{ind} smoothly. The goal of this level is to

compute a *preferred velocity* v_{pref} for the agent in each time step of the simulation.

Many researchers and software systems do not treat route following as a separate level. However, we believe that route following is crucial for the following main reasons:

- The indicative route is generally not smooth, and it is often computationally expensive to smoothen it beforehand.
- Modern collision avoidance algorithms (as described in the next subsection) require a preferred velocity as input. Unless the agent can walk towards its goal in a straight line, we need an algorithm that chooses a desired walking direction at any point in time.
- Due to the presence of other agents, an agent is often not located exactly on π_{ind}. A route following algorithm can define how the agent should gradually move back onto the desired route.
- The virtual environment may contain *weighted regions* that are less or more attractive to traverse. A route following algorithm can take this into account, e.g. to let an agent cut corners based on its personal preferences.

Two recent algorithms for route following are based on *attraction points*: in each simulation step, they choose a point p_{att} on the indicative route towards which the agent wants to move. The preferred velocity is then computed as the vector that takes the agent to p_{att} at its preferred walking speed. This last step is analogous to one of the 'steering behaviors' described by Reynolds [27], who was also among the first to acknowledge route following as a separate process.

The first algorithm, the Indicative Route Method (IRM) [28], defines p_{att} as the farthest point along π_{ind} that lies inside the largest obstacle-free disk containing the agent's position. When more free space is available, p_{att} lies farther along the route and the amount of smoothing increases.

The successor of IRM, called Modified and Indicative Routes and Navigation (MIRAN) [29], defines a set of *candidate* attraction points along π_{ind} and chooses p_{att} as the best candidate according to the agent's personal region preferences. In other words, the amount of smoothing and route shortening depends on the local terrain costs for that particular agent. A user-controlled parameter determines how far along the route the candidate attraction points are allowed to lie, which controls how closely the agent will follow the route. Figure 3 shows an example of this method.

D. Local Movement

At the local movement level (2), the agent might temporarily deviate from its route to resolve local collisions with other agents. This is generally referred to as *collision avoidance*.

In early collision avoidance algorithms, agents exerted attractive and repulsive *forces*, and physical laws of motion yielded new velocities for each agent [27], [30]. A disadvantage of these models is that they are inherently reactive, instead of letting the agents actively choose how to move based on the movement of other agents.

Hence, more recent algorithms are based on *velocity selection* [31]–[33]. These algorithms let an agent pick the best speed and direction from a sampled range of options (i.e. candidate velocities), based on a cost function. The cost of a candidate velocity is based on the difference to the agent's preferred velocity, and on the predicted collisions with other agents based on their current movement. In the end, the agent chooses a velocity v_{new} that avoids collisions while being similar to v_{pref}. When all agents have computed a new velocity, their positions are updated using time integration [28] and the next simulation step begins.

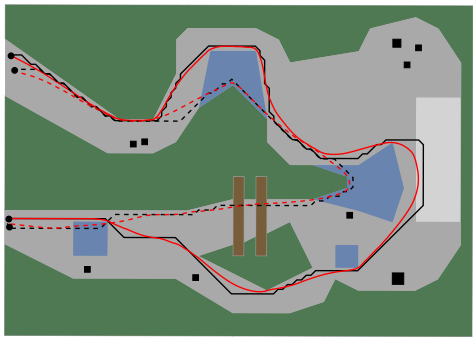


Fig. 3. Path planning and following in a forest (green) with tree obstacles (black), puddles (blue), fallen trees (brown), and a spot with a panoramic view (light gray). For two agents (adult and child), we compute an indicative route (solid and dashed black, respectively) on a grid that uses personalized region weights. The smoothed paths (solid and dashed red) are computed using the MIRAN method.

Collisions are usually only predicted for a small number of neighbors within the agent's field of view. Finding these neighboring agents is a computationally expensive step of the simulation loop. A data structure such as a kd-tree is suitable for answering nearest-neighbor queries [34]. Since the distribution of agents in the environment is constantly changing, this query structure is typically rebuilt in each simulation step. Our own software does not use a kd-tree, but a grid with square cells of 10×10 meters; agents only need to consider the agents in their surrounding grid cells.

E. Animation

Finally, the animation level (1) produces visual output by animating and translating the agent's 3D model in the environment [35]. This is relatively simple if the 3D motion clips are available. Producing smooth and physically correct animations without requiring such data is an active research topic that is outside the scope of this paper.

Note that the animation and the simulation usually have different framerates. Crowd simulations often use a fixed timestep of $0.1s$ (i.e. 10 frames per second) [28], whereas smooth animation requires a much higher framerate. We therefore assume that the animation level uses a separate loop. Whenever a *simulation* step finishes and all agent positions have been updated (in the simulation model only), the *animation* layer is notified and visually brings the agents to their new positions.

F. Communication between layers

It is important to note that the planning process of our hierarchy is not purely serial. Events in the lower levels may

cause an agent to reconsider its global plans. For instance, when an agent has reached its goal position, it returns to the global planning or high-level planning level to determine its next action. Another example is *re-planning*: if a part of the environment turns out to be too crowded, or a section of the indicative route is unexpectedly blocked by a dynamic obstacle, an agent may choose to reconsider its route and take a detour.

IV. Environment Representation

To facilitate the geometric steps of the planning hierarchy, we need a convenient representation of the virtual environment. The environment's original 3D geometry is not suitable: performing queries for thousands of agents in 3D would be too expensive. Instead, we require a simplified representation for navigation tasks, which makes use of our assumption that agents are constrained to walkable surfaces. We will argue that a *navigation mesh* is the best choice for efficient real-time solutions in levels 4, 3, and 2 of the hierarchy.

In the remainder of this paper, we assume that the *walkable environment* is a set of connected 2D layers; each layer is a planar surface with polygonal obstacles. From this point on, we will also simply refer to this as 'the environment'.

A. Related work

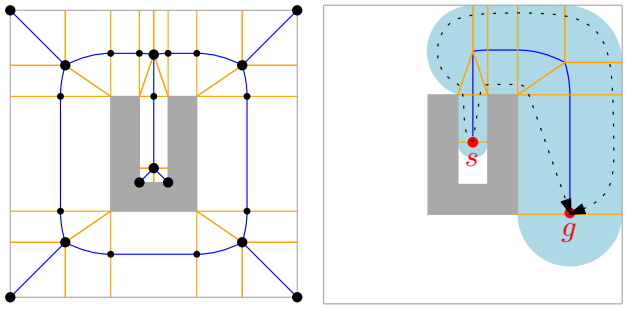
Representations of a walkable environment exist in three categories: traditional graphs, grids, and navigation meshes.

Traditional graphs represent the environment by a set of vertices and edges. They are a popular choice for high-dimensional motion planning problems in robotics [36], [37]: these problems are difficult to represent in an exact manner, and a working solution is often preferred over a natural-looking one. Graphs are less suitable for crowds of agents on walkable surfaces: agents would need to either follow the edges exactly (which leads to collisions between agents), or perform expensive geometric tests to check how they can deviate from an edge.

Grids subdivide the environment into regular cells such as squares. They are intuitive and easy to implement [38]. However, grids have resolution problems: a coarse grid (with few cells) does not capture the environment's details, whereas a fine grid (with many cells) quickly becomes too costly to store and query.

By contrast, navigation *meshes* efficiently subdivide the walkable space into polygonal regions. A global path in a navigation mesh can be found by performing A* search on the *dual graph*, which consists of one vertex per region and one edge for each pair of adjacent regions. The result is a sequence of regions to move through, such that agents can use a strip of free space to locally adjust their movements during the simulation steps. In other words, navigation meshes also support the route following and local movement levels of our suggested hierarchy.

Many navigation meshes exist for 2D environments [22], [39], [40], and for *multi-layered* 3D environments in which multiple 2D layers are connected [10], [41]–[43]. In general, navigation meshes represent the enviroment more adequately, they are more scalable to large environments, and they are flexible enough to support complex and crowded situations.



(a) Explicit Corridor Map (b) Indicative routes

Fig. 4. A simple environment with obstacles (shown in gray). (a) The ECM is the medial axis (blue) annotated with closest-obstacle information (orange). This subdivides the walkable space into polygonal regions. (b) A path along the medial axis induces a corridor (light blue) due to the ECM's annotations. Within the corridor, we can define any indicative route from s to g; two examples are shown in black.

B. The Explicit Corridor Map

We have developed a navigation mesh called the *Explicit Corridor Map* (ECM) [22], [42]. An example is shown in Figure 4a. The ECM is based on the *medial axis*, which is the set of all points that have at least two distinct equidistant closest obstacle points in the environment. It is tightly related to the *generalized Voronoi diagram* with obstacle polygons being the corresponding Voronoi sites [34]. The medial axis is a graph in which all edges run through the middle of the walkable space. Each vertex of this graph has at least three different nearest obstacle points, or it lies at a non-convex obstacle corner. Each edge of the medial axis consists of a sequence of line segments and parabolic arcs, depending on the type of obstacles to the left and right. For a 2D environment with n obstacle vertices, the medial axis has $O(n)$ complexity and can be constructed in $O(n \log n)$ time [22].

The ECM is an *annotated medial axis*: it stores the left and right closest obstacle points for each edge section. This partitions the environment into polygonal walkable regions.

The ECM has many features that make it well-suited for our framework and for crowd simulation in general:

- It is a sparse graph with only $O(n)$ vertices and edges. Hence, it requires little storage, and global paths can be extracted efficiently using e.g. A* search.
- It can be constructed in $O(n \log n)$ time, so it scales well to larger environments.
- It represents the exact geometry of the walkable environment. This resolves the issues that are inherent to approximated representations such as grids.
- Its regions are non-overlapping, which makes many queries and algorithms easier.
- For any point in the free space, the ECM cell containing it can be found in $O(\log n)$ time, after which the nearest obstacle can be found in $O(1)$ time. This makes collision

checking with static obstacles very efficient. In one simulation step, an agent either stays in the same cell or moves to an adjacent one, so point location queries are rarely needed. Hence, on average, collision checking with obstacles takes constant time per agent.
- It enables path planning for agents of any size, using only a single data structure. Because the ECM stores clearance information, A* can determine in real-time whether an agent is small enough to traverse an edge. Most other navigation meshes artificially inflate the obstacles and work well for only one agent size.
- It can produce a variety of indicative routes, e.g. routes that stay on the left and right side of the walkable space, or short paths with a preferred amount of clearance [22]. Figure 4b illustrates this.
- It can be efficiently updated in response to insertions and deletions of obstacles [44], such that it allows crowd simulation in dynamic environments.
- It is well-defined for multi-layered 3D environments [42] that consist of multiple connected 2D layers. Any 2D algorithm that uses the ordering of ECM edges (e.g. dynamic updates, visibility queries, or computing short paths with clearance) will work trivially in these environments.

In short, the ECM is a generic basis for efficient path planning and crowd simulation.

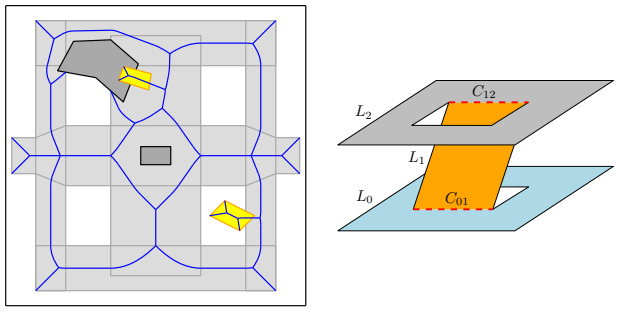
V. Crowd Simulation Software

In this section, we describe our crowd simulation software framework based on the ECM navigation mesh. Our software can be applied to the *geometric* planning problems induced by a *semantic* high-level planner. The framework is written in C++ using Visual Studio 2013, but its code is platform-independent and compiles successfully on Linux as well.

A. Input and Output

The environment is assumed to be given in a simple XML file format that describes the geometry of each 2D layer. Per layer, the geometry can consist of *walkable areas* (polygons on which agents can walk), *obstacles* (non-walkable polygons that overrule walkable areas), and *openings* (walkable polygons that overrule obstacles). This combination of elements makes the environment easy to define. An example is shown in Figure 5a. Layers can also contain *connections*; a connection is a line segment that connect the walkable space of two adjacent layers. Figure 5b shows a multi-layered environment with connections. Finally, each layer can contain *weighted regions*: polygons with a certain type (e.g. 'grass' or 'road') to which agents can associate a personal weight. These weights are stored in separate *agent profiles*: XML blocks that describe all agent properties required for a simulation, such as the radius, the preferred walking speed, and the algorithms to use for global planning, path following, and collision avoidance.

A computed ECM is saved as an XML file that describes all vertices, edges, and closest-obstacle annotations. Running a simulation in our framework requires a navigation mesh, a set of agent profiles, and the environment. All results (the environment, the ECM, or the state of the simulation at any point in time) can also be exported to a vector file for Ipe [45].



(a) 2D (b) Multi-layered

Fig. 5. Constructing an environment. (a) Users can define the walkable space of a layer in terms of walkable areas (light gray), obstacles (dark gray), and openings (yellow). The medial axis of the combined walkable space is shown in blue. (b) Multi-layered environments consist of 2D layers L_i and connections C_{ij}. The layers are drawn in different colors for clarity.

B. Computing Navigation Meshes

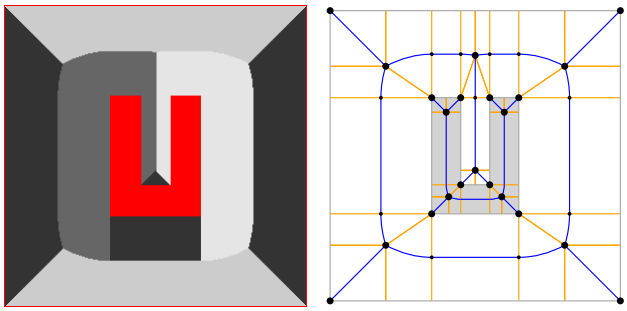
Our software can compute the ECM of a walkable environment. For a 2D environment, we first convert the geometry to a set of disjoint non-walkable polygons, by applying Boolean operations using the OpenGL tessellator [46]. Despite being part of the OpenGL library, this tessellator does not require any discretized rendering.

The resulting polygons are sent to an *ECM generator* of choice. Three such generators have been implemented:

- The first implementation renders an approximated Voronoi diagram on the GPU [47], which is converted to an ECM on the CPU [22]. It requires the user to set the rendering resolution, e.g. at 20 pixels per meter. See Figure 6a for an illustration.
- The second implementation uses Vroni [48], a library that is widely used in geometry-related research. It computes a topologically correct Voronoi diagram for a set of line segments (i.e. the contours of the walkable space). From the result, we extract the medial axis, add the ECM's closest-obstacle annotations, and filter out graph components that lie inside the obstacle space. Figure 6b shows an example.
- The third implementation works similarly to the second, but it uses the Voronoi diagram functionalities of the open-source Boost library [49].

Ideally, Boost and Vroni are preferred, because they do not depend on a resolution parameter. However, since these exact methods respond to every imprecision in the input, their resulting ECMs may contain unwanted details. Additional filtering steps (e.g. filling small openings, merging line segments that are close together, and resolving intersections between segments) are needed to achieve robustness for arbitrary environments.

For *multi-layered 3D* environments, we use an iterative algorithm that first computes the 2D ECM for each layer



(a) GPU-based (b) Exact

Fig. 6. Two ways to generate the ECM. (a) Using rendering techniques, we can approximate the Voronoi diagram on the GPU frame buffer. (b) External libraries such as Boost and Vroni can compute the Voronoi diagram, which we convert to an ECM. Graph elements inside obstacles are filtered out in a post-processing step.

and then stitches these ECMs together to obtain a continuous multi-layered navigation mesh [42]. Since this navigation mesh locally has the same properties as in a single layer, all 2D algorithms (e.g. visibility queries, global path planning, and dynamic updates) trivially work in the multi-layered ECM as well. The construction algorithm consists only of a sequence of 2D steps, so any of the generators described above can be used.

The ECM generator can either return the navigation mesh as an XML file or keep it in memory, so one could immediately start a simulation instead of loading the file again.

C. Algorithms of the Planning Hierarchy

For global planning, we have implemented A* to compute shortest paths on the medial axis. The resulting path can be converted to various types of indicative routes: short paths with a preferred amount of clearance, or paths with side preference (e.g. for staying on the left and right side of the free space). We have also included a re-planning algorithm that recomputes the medial axis path efficiently after an obstacle has been inserted or deleted.

When weighted regions are present, agents can perform A* search on a weighted grid instead. At the time of writing, an improved method for weighted regions is being integrated into the framework [26].

For path following, our framework includes both IRM [28] and MIRAN [29]. At the local movement level, we have implemented two recent velocity-based collision avoidance algorithms by Moussaïd et al. [33] and Karamouzas et al. [31]. We have also included the popular ORCA collision-avoidance library [32] in our framework.

D. Architecture

We will now highlight a number of details concerning the architecture of our software. This discussion focuses on aspects that make the framework modular and efficient.

Modularity. For each of the three geometric levels in the hierarchy, any algorithm can be plugged in as long as it implements the required abstract methods. For example, all route following implementations should compute the preferred velocity for a given agent, but the programmer can decide the internal details. We use the *factory* design pattern so that new implementations can easily be added. Users can assign any combination of algorithms to an agent (e.g. short global paths with clearance, MIRAN for path following, and RVO for collision avoidance) using a settings file. A similar architecture is used for the ECM generators described in Section V-B, so that users can easily switch between implementations.

Sequence of simulation loops. Instead of performing all computations at once for each agent, we subdivide a simulation step into multiple loops. We first compute the preferred velocity of each agent; next, we compute the actual velocity of each agent; finally, we update the agents' positions. This ensures that the order in which the agents are stored does not matter: the first agent in the ordering uses the exact same information as the last agent, and the result is deterministic.

Multi-threading. Each of the loops in a simulation step can easily be parallellized, because a single agent only needs to *read* properties of the environment or neighboring agents. Hence, the calculations for different agents are completely independent. We use basic OpenMP instructions [50] to automatically divide multiple agents over multiple threads, without having to lock parts of the code to prevent conflicts and deadlocks between threads.

API / Library. We have also built the framework as a Windows library (DLL) with a number of basic API functions, e.g. for loading an environment, computing the ECM, adding an agent, et cetera. The API function that performs a single simulation step fills an array of wrapper objects (C structs) that contain the new positions and orientations of each agent, and it returns a pointer to this array. If an external program is linked to our DLL and defines the exact same wrapper object, both programs can share the array. Using this technique, we have linked our DLL to the Unity3D game engine to display moving crowds in 3D. The Pedestrian Dynamics crowd analysis software [7] uses our framework in a similar fashion. This software was used for various crowd simulations in preparation for real-life events, such as crowded sports stadiums and King's Day in Amsterdam.

VI. Experiments

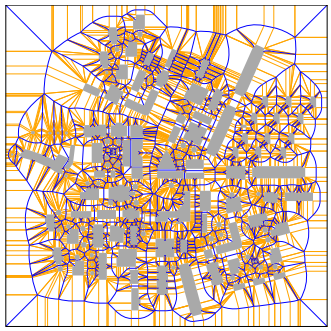
In this section, we demonstrate the capabilities of our software using two large environments: *City*, a 2D footprint of a virtual city, and *Station*, a multi-layered model of a train station in The Netherlands. Both environments and their ECMs are shown in Figure 7. Details of the environments can be found in Figure 8.

All experiments were performed on a Windows 7 PC with a 3.20 GHz Intel i7-3930K CPU, an NVIDIA GeForce GTX 680 GPU, and 16 GB of RAM. In general, only one CPU core was used, except in the final experiment which shows the benefit of multi-threading.

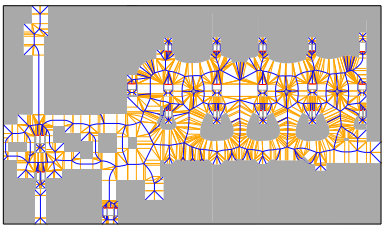
A. Generating the ECM

We have computed the ECM for both environments using all three implementations. For the GPU-based method, we used

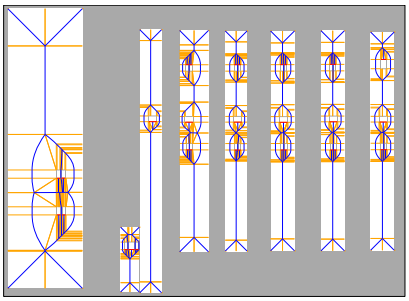
(a) *City* (3D)



(b) *City* (2D)



(c) *Station* (terminal layer)



(d) *Station* (platform layer)

Fig. 7. The two environments used in our experiments. The medial axis is shown in blue; closest-point annotations are shown in orange. For the *Station* environment, only the two main layers are shown. Layer connections are displayed in red.

a resolution of 4000×4000 pixels. All computations were repeated 10 times. The results are shown in the last three columns of Figure 8. This table shows that our Vroni-based implementation is the fastest: on average, it computes the ECM of *City* in 84 ms, and the ECM of *Station* in 368 ms. Hence, even for large and multi-layered environments, the ECM is generated well within a second, which allows our framework to be integrated into a modelling tool with interactive feedback.

B. Dynamic Updates

Next, we have dynamically inserted 100 obstacles into both environments at random free positions. After an insertion, the ECM is updated along with its point-location data structure, such that agents can use the updated navigation mesh in future simulation steps. For simplicity, we only added square obstacles measuring 2×2 meters, although the insertion algorithm supports any convex polygon that does not intersect existing geometry [44]. Figure 10a shows the obstacles and the resulting ECM for *City*.

The insertion times (for updating both the ECM and the point-location structure) are shown in Table 9. On average, a insertion took 3.70 ms in *City* and 1.25 ms in *Station*. The running times in *City* are higher because this environment is more complex, meaning that a dynamic update affects more ECM cells on average. These results indicate that the ECM can efficiently model dynamically changing environments, e.g. with bridges that may collapse, parked vehicles that temporarily block roads, et cetera. More experiments can be found in a previous publication [44].

C. Computing Visibility Polygons

For any point p in the walkable space, our software can also compute the 2D *visibility polygon* $V(p)$, i.e. the area that is visible from p, assuming all surfaces are flat. In multi-layered environments, the polygon may lie on multiple layers, so this is an approximation of 3D visibility. The visibility polygon can be used to model what agents can visually perceive, e.g. to let them respond to an event in the environment when they see it. This polygon is computed by traversing the ECM cells in an ordered manner; hence, the running time depends on the number of cells that $V(p)$ intersects.

We computed the visibility polygon for $1,000$ random query points in both environments. This algorithm takes 0.15 ms in *City* and 0.10 ms in *Station* on average. Thus, the ECM framework can easily answer visibility queries for many agents in real-time. Figure 10b shows a number of visibility polygons in the *City* environment.

D. Computing Indicative Routes

Next, we have computed global indicative routes for $1,000$ pairs of random start and goal positions per environment. To compute such a route, we first perform A* search on the ECM, which yields a shortest path along the medial axis. We then extract an indicative route through the corridor around this path, as explained in Figure 4b.

In Section IV-B, we stated that there are many options for computing an indicative route through a corridor. Since these options have comparable complexity, showing only one option is sufficient for giving a general indication of global planning times. For this experiment, we have used the shortest route [22] with a preferred clearance of 0.5 m.

On average, global planning takes 1.17 ms in *City* ($\sigma = 0.70$) and 0.85 ms in *Station* ($\sigma = 0.47$). The computation time for a single indicative route is roughly proportional to the number of vertices on its ECM route. This explains the high standard deviations, since random query points yield paths of different lengths. Examples of indicative routes in *City* are shown in Figure 10c.

E. Crowd Simulation

To show the efficiency of our crowd simulation software, we have generated increasingly large crowds of agents in our environments. Figure 10d shows a crowd in the *City* environment.

We measured the running time of each simulation step as long as all agents were still traversing a path, i.e. up to and including the step in which the first agent reached its goal. All agents received random start and goal positions, with a minimum 2D Euclidean distance of 50 meters between start and goal to ensure that the agents did not reach their goal too quickly. Note that we excluded the time required for computing the indicative routes; this aspect was already covered in the previous experiment.

We ran all simulations using fixed timesteps of 0.1 seconds, which is common in crowd simulation software [28]. Thus, whenever these steps take at most 100 ms to compute, our simulation runs in real-time. In line with real-life measurements [51], we used an agent radius of 0.24 meters and a preferred walking speed of 1.4 m/s. For path following, we used MIRAN [29] with a sampling distance of 1 m and a shortcut parameter of 5 m. We used the vision-based collision algorithm by Moussaïd et al. [33] because it yields the best results in many scenarios. However, since collision avoidance is inherently the most expensive phase (because it requires agents to look for other agents), we have run this experiment both with and without collision avoidance.

We have also performed all simulations with and without *multi-threading*. As described in Section V-D, a step of the simulation loop consists of multiple subtasks: computing the preferred velocities of all agents, computing the actual velocities to avoid collisions, and updating all positions using these velocities. We have used OpenMP to divide the workload of each subtask over 8 threads. Note that the three loops are still executed in sequence, to maintain consistency among agents.

We choose to report the results of *City* only, since the results for *Station* are comparable and the *City* environment is physically larger, i.e. it supports larger crowds.

Without Collision Avoidance. When collision avoidance is disabled, the running time of a simulation step scales linearly with the number of agents, as indicated by the red line in Figure 11a. For example, with $100,000$ agents, a step took 82 ms on average. With *one million* agents, it is worth noting that the framework used $2.3GB$ of memory in this scenario, which is a small memory footprint considering the size of the crowd. A crowd of this size cannot be simulated in real-time yet, but multi-threading techniques and hardware improvements could make this possible in the near future.

Using 8 parallel threads greatly improves the running times, as indicated by the green line in Figure 11a. The simulation does not become 8 times as fast, because not the entire simulation step can be parallellized, e.g. the three subtasks need to be performed sequentially. Still, we achieve a speed-up factor of 3 to 4 for large crowds. For instance, simulating $200,000$ agents now takes 50 ms per step on average. Future work will show how the number of threads influences the improvement ratio.

With Collision Avoidance. Figure 11b shows the running times for the same scenarios *with* collision avoidance. Note that this figure intentionally has a different scale on both axes. We could not model the largest crowds because the environment became too full; simulating up to a million agents would only make sense in a much larger environment.

A multi-threaded simulation with $10,000$ agents requires 63 ms per step on average. The computation time now appears to scale quadratically with the number of agents. This is most likely due to the grid used for nearest-neighbor queries, which has cells of a fixed size. If large clusters of agents appear in one cell, agents need to evaluate many potential neighbors. A data structure such as a kd-tree could overcome this problem, although such a structure would need to be rebuilt in each simulation step. We leave a comparison of these data structures for future work.

In conclusion, the ECM framework can simulate tens of thousands of agents in real-time, including collision avoidance. Of course, many simulations will include more steps than the ones we measured, such as global (re-)planning and high-level planning. Also, if real-time visualization is desired (such as in a computer game or an interactive simulation), less time is available for the simulation itself. The details depend on the application at hand.

VII. Conclusion and Future Work

In this paper, we have suggested a five-level hierarchy for solving agent navigation problems in games and simulations. By treating each level separately, we subdivide a complex problem into more manageable subproblems.

We have presented our Explicit Corridor Map (ECM) framework as an implementation of the three center levels, which comprise the *geometric* aspects of navigation: global planning, route following, and local movement. The ECM is a *navigation mesh* that has many advantages over classical graph or grid representations. For instance, it enables fast and flexible global path planning due to an underlying sparse graph, it supports agents of all widths using only one data structure, and it allows fast collision checking with obstacles due to its cell decomposition. Experiments show that our software allows many operations at interactive rates, and that it can simulate large crowds in complex 2D and multi-layered environments in real-time.

Our framework is general enough to support various future extensions and improvements. One possible extension is to take agents of different heights into account. For instance, big vehicles may not fit through small tunnels that regular agents can use. Furthermore, future versions of the framework will allow the simulation of small groups of agents [52] and high-density crowds with improved coordination.

We believe that our flexible and extensible framework provides a comprehensive set of techniques for researchers, game developers, and crowd simulation engineers.

Acknowledgments

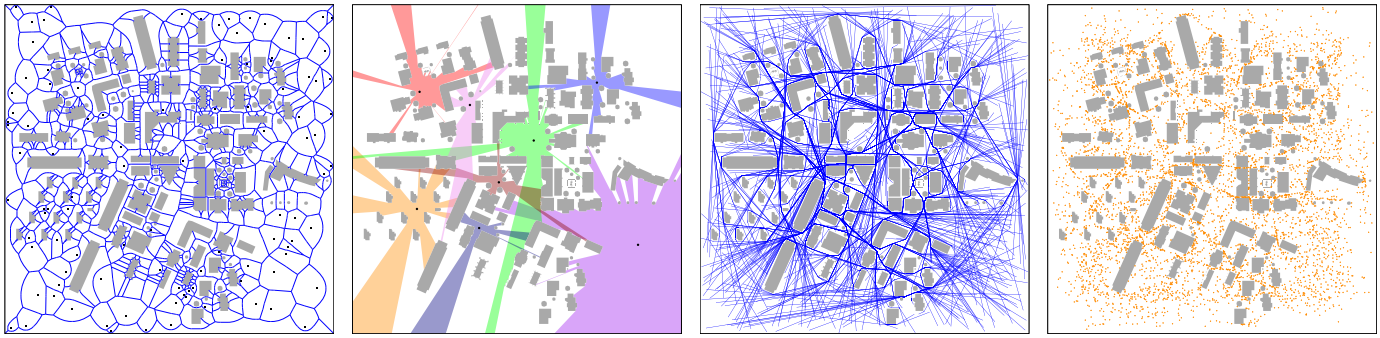
This research has been supported by the COMMIT project (http://www.commit-nl.nl/).

Environment	Geometry			ECM			ECM construction time (ms)		
	#Obstacles	#Vertices	Size (m)	#Vertices	#Edges	#Annotations	Vroni	Boost	GPU
City	184	2098	500×500	1444	1623	6310	84 [1.3]	141 [1.7]	554 [4.6]
Station	568	1800	153×111	660	768	2804	368 [4.4]	381 [5.2]	1266 [8.1]

Fig. 8. The experimental environments and their ECMs. The *Geometry* columns show the number of obstacles, their combined number of vertices, and the width and height of the environment (in meters). The *ECM* columns show the complexity of the ECM: the number of vertices, edges, and points with closest-obstacle information. The last three columns show the construction time for the ECM using all three implementations. Running times were averaged over 10 runs; the standard deviations are shown between square brackets.

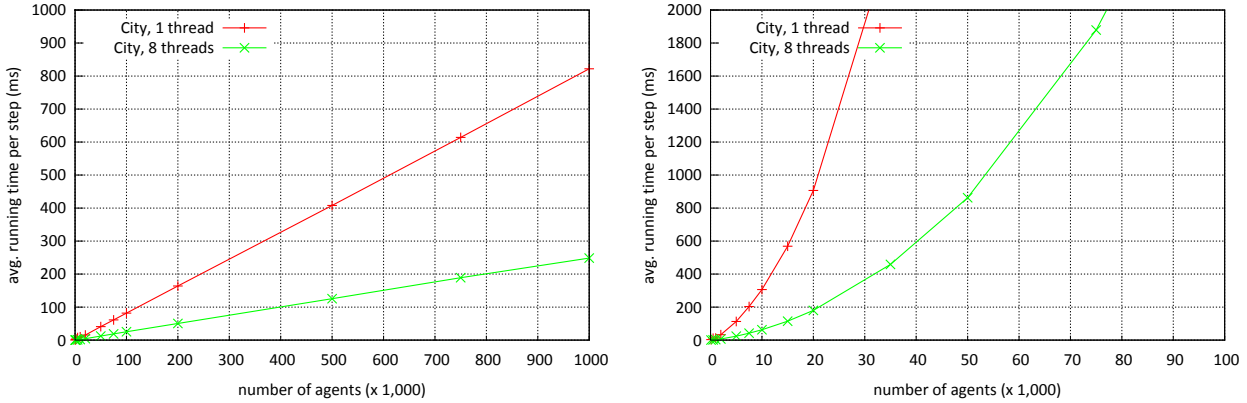
Environment	Dynamic insertions (ms)	Visibility (ms)	Indicative routes (ms)
City	3.70 [0.38]	0.15 [0.06]	1.17 [0.70]
Station	1.25 [0.21]	0.10 [0.07]	0.85 [0.47]

Fig. 9. Results of three experiments. Standard deviations are shown in square brackets. The *Dynamic insertions* column displays the time to dynamically insert a square obstacle into the ECM, averaged over 100 obstacles. The *Visibility* column displays the time to compute a visibility polygon, averaged over $1,000$ random positions. The *Indicative routes* column denotes the time to compute a short indicative route with clearance, averaged over $1,000$ pairs of random start and goal positions.



(a) Dynamic updates (b) Visibility (c) Indicative routes (d) Crowd

Fig. 10. Experiments in the *City* environment. (a) We have dynamically inserted 100 square obstacles (shown in black) at random positions. The updated medial axis is shown in blue. (b) Visibility polygons. Query points are shown in black; their visibility polygons are shown in different colors. (c) Examples of 500 indicative routes (shown in blue) between random start and goal positions, computed by performing A* on the ECM and computing a short route within the resulting corridor. (d) A crowd of $5,000$ agents, shown as orange disks. Agents have been enlarged for illustrative purposes.



(a) Without collision avoidance (b) With collision avoidance

Fig. 11. Running times of crowd simulations in the *City* environment. The horizontal axis shows the number of agents ($\times 1,000$); the vertical axis shows the average running time of a simulation step, which models 100 milliseconds of simulation time. (a) Without collision avoidance, the running time is proportional to the number of agents. Multi-threading improves the results by a factor of 3 to 4. (b) With collision avoidance, the running time increases at a higher rate. Multi-threading allows us to simulate over $10,000$ agents in real-time.

REFERENCES

[1] N. Jaklin, W. van Toll, and R. Geraerts, "Way to go - a framework for multi-level planning in games," in *Proceedings of the 3rd International Planning in Games Workshop*, 2013, pp. 11–14.

[2] S. Rabin, *AI Game Programming Wisdom*. Charles River Media, 2002.

[3] D. Thalmann and S. Musse, *Crowd Simulation*, 2nd ed. Springer, 2013.

[4] "Legion," http://www.legion.com/, 2015.

[5] "Massive," http://www.massivesoftware.com/, 2015.

[6] Oasys Software, "MassMotion," http://www.oasys-software.com/, 2015.

[7] Incontrol Simulation Solutions, "Pedestrian Dynamics," http://www.pedestrian-dynamics.com/, 2015.

[8] "SimWalk," http://www.simwalk.com/, 2015.

[9] PTV Group, "VisWalk," http://vision-traffic.ptvgroup.com/, 2015.

[10] M. Mononen, "Recast Navigation," https://github.com/memononen/recastnavigation/, 2014.

[11] "Golaem Crowd," http://www.golaem.com/, 2015.

[12] S. Singh, M. Kapadia, P. Faloutsos, and G. Reinman, "An open framework for developing, evaluating, and sharing steering algorithms," in *Proceedings of the 2nd International Workshop on Motion in Games*, 2009, pp. 158–169.

[13] A. Shoulson, N. Marshak, M. Kapadia, and N. Badler, "Adapt: The agent development and prototyping testbed," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2013, pp. 9–18.

[14] W. Daamen, "SimPed: a pedestrian simulation tool for large pedestrian areas," in *Proceedings of the EuroSIW Conference*, 2002.

[15] S. Hoogendoorn, "Microscopic simulation of pedestrian flows," in *Proceedings of the 82nd Annual Meeting at the Transportation Research Board*, 2003.

[16] S. Curtis, A. Best, and D. Manocha, "Menge: A modular framework for simulating crowd movement," University of North Carolina at Chapel Hill, Tech. Rep., 2014.

[17] W. Shao and D. Terzopoulos, "Autonomous pedestrians," *Graphical Models*, vol. 69, no. 5–6, pp. 246–274, 2007.

[18] R. Fikes and N. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[19] J. Kelly, A. Botea, and S. Koenig, "Offline planning with Hierarchical Task Networks in video games," in *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008, pp. 60–65.

[20] S. Paris and S. Donikian, "Activity-driven populace: a cognitive approach to crowd simulation," *IEEE Computer Graphics and Applications*, vol. 29, pp. 34–43, 2009.

[21] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[22] R. Geraerts, "Planning short paths with clearance using Explicit Corridors," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2010, pp. 1997–2004.

[23] W. van Toll, A. Cook IV, and R. Geraerts, "Real-time density-based crowd simulation," *Computer Animation and Virtual Worlds*, vol. 23, no. 1, pp. 59–69, 2012.

[24] R. Geraerts and E. Schager, "Stealth-based path planning using corridor maps," in *Proceedings of the International Conference on Computer Animation and Social Agents*, 2010.

[25] I. Karamouzas, R. Geraerts, and A. van der Stappen, "Spacetime group motion planning," in *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*, 2012, pp. 227–243.

[26] N. Jaklin, M. Tibboel, and R. Geraerts, "Computing high-quality paths in weighted regions," in *Proceedings of the 7th International Conference on Motion in Games*, 2014, pp. 77–86.

[27] C. Reynolds, "Steering behaviors for autonomous characters," in *Proceedings of the Game Developers Conference*, 1999, pp. 763–782.

[28] I. Karamouzas, R. Geraerts, and M. Overmars, "Indicative routes for path planning and crowd simulation," in *Proceedings of the 4th International Conference on Foundations of Digital Games*, 2009, pp. 113–120.

[29] N. Jaklin, A. Cook IV, and R. Geraerts, "Real-time path planning in heterogeneous environments," *Computer Animation and Virtual Worlds*, vol. 24, no. 3, pp. 285–295, 2013.

[30] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, pp. 4282–4286, 1995.

[31] I. Karamouzas and M. Overmars, "A velocity-based approach for simulating human collision avoidance," in *Proceedings of the 10th International Conference on Intelligent Virtual Agents*, 2010, pp. 180–186.

[32] J. van den Berg, S. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Proceedings of the 14th International Symposium on Robotics Research*, 2011, pp. 3–19.

[33] M. Moussaïd, D. Helbing, and G. Theraulaz, "How simple rules determine pedestrian behavior and crowd disasters," in *Proceedings of the National Academy of Science*, vol. 108, 2011, pp. 6884–6888.

[34] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer, 2008.

[35] B. van Basten, J. Egges, and R. Geraerts, "Combining path planners and motion graphs," *Computer Animation and Virtual Worlds*, vol. 22, pp. 59–78, 2011.

[36] L. Kavraki, P. Švestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[37] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[38] W. Lee and R. Lawrence, "Fast grid-based path finding for video games," in *Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7884, pp. 100–111.

[39] R. Wein, J. van den Berg, and D. Halperin, "The Visibility-Voronoi complex and its applications," in *Proceedings of the 21st Annual ACM Symposium on Computational Geometry*, 2005, pp. 63–72.

[40] M. Kallmann, "Navigation queries from triangular meshes," in *Proceedings of the 3rd International Conference on Motion in Games*, 2010, pp. 230–241.

[41] J. Pettré, J. Laumond, and D. Thalmann, "A navigation graph for real-time crowd animation on multilayered and uneven terrain," in *Proceedings of the First International Workshop on Crowd Simulation*, 2005, pp. 81–89.

[42] W. van Toll, A. Cook IV, and R. Geraerts, "Navigation meshes for realistic multi-layered environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 3526–3532.

[43] R. Oliva and N. Pelechano, "NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments," *Computers & Graphics*, vol. 37, no. 5, pp. 403–412, 2013.

[44] W. van Toll, A. Cook IV, and R. Geraerts, "A navigation mesh for dynamic environments," *Computer Animation and Virtual Worlds*, vol. 23, no. 6, pp. 535–546, 2012.

[45] O. Cheong, "The Ipe extensible drawing editor," http://ipe7.sourceforge.net/, 2014.

[46] "OpenGL," https://www.opengl.org/, 2015.

[47] K. Hoff, III, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Fast computation of generalized Voronoi diagrams using graphics hardware," *International Conference on Computer Graphics and Interactive Techniques*, pp. 277–286, 1999.

[48] M. Held, "VRONI and ArcVRONI: Software for and applications of Voronoi diagrams in science and engineering," in *Proceedings of the 8th International Symposium on Voronoi Diagrams in Science and Engineering*, 2011, pp. 3–12.

[49] "The Boost C++ library," http://www.boost.org/, 2015.

[50] "OpenMP," http://openmp.org/, 2015.

[51] U. Weidmann, "*Transporttechnik der Fussgänger - Transporttechnische Eigenschaften des Fussgängerverkehrs*," ETH Zürich, Institut für Verkehrsplanung, Transporttechnik, Strassen- und Eisenbahnbau, Literature Research 90, 1993, in German.

[52] I. Karamouzas and M. Overmars, "Simulating and evaluating the local behavior of small pedestrian groups," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 394–406, 2012.