Enhancing Corridor Maps for Real-Time Path Planning in Virtual Environments

Roland Geraerts and Mark H. Overmars*
Institute of Information and Computing Sciences, Utrecht University
3508 TA Utrecht, the Netherlands, roland@cs.uu.nl

Abstract

A central problem in interactive virtual environments is planning high-quality paths for characters avoiding obstacles in the environment. Current applications require a path planner that is fast (to ensure real-time interaction with the environment) and flexible (to avoid local hazards). In addition, paths need to be natural, i.e. smooth and short.

To satisfy these requirements, we need an adequate representation of the free space stored in a convenient data structure, a fast mechanism for querying this data structure, and an algorithm that constructs natural paths for the characters.

We improve an existing data structure, the *Corridor Map*, which represents the free space by a graph whose edges correspond to collision-free corridors. We show that this structure, together with a kd-tree, can be used for fast querying, resulting in a corridor that guides the global path of the character. Its local motions are controlled by force functions, providing the desired flexibility. Experiments show that the improvements lead to a method which can steer a crowd of $\pm$10,000 characters in real-time.

1 Introduction

One of the main challenges in virtual environments is path planning for characters. These characters have to traverse from a start to a goal position without colliding with obstacles and other characters. In the past twenty years, many algorithms have been devised to tackle the path planning problem [10, 11]. These algorithms were mainly developed in the field of robotics, aiming at creating a path for one or a few robots having many degrees of freedom. Usually, much CPU time was available for computing a *nice path*, which often meant a short path having some clearance to the obstacles, because a bad path could be expensive to traverse, and could damage the robot or environment.

While these algorithms were successfully applied in fields such as mobile robots, manipulation planning and human robot planning [10], current virtual environment applications, e.g. games, pose many new challenges to the algorithms. That is, *natural* paths for many characters traversing in the ever growing environments need to be planned simultaneously and in real-time. Hence, only a (fraction of a) millisecond per second CPU time may be spent per character for computing the natural path (i.e. a path that is smooth, short, keeps some clearance to obstacles, avoids other characters, etcetera).

In short, current environments require a fast flexible planner generating natural paths. A candidate for the flexible planner is a Potential Field method [14], because it can be used to evade characters and to create smooth paths. Due to the method's local behavior, it will not always find a path.

Roadmap based methods, such as Visibility graphs [10], Probabilistic Roadmap Methods [11], and an A* method operating on a graph-like grid [16], can usually ensure that a path can be found if one exists. However, they lack flexibility because they output a fixed path (extracted from a one-dimensional graph). In addition, the paths are unnatural. While some optimization algorithms exist, they are too slow to be applied in real-time [6].

Recently, the Corridor Map Method (CMM) has been proposed, which satisfies the requirements mentioned above [5]. The CMM directs the global motions of a character traversing a corridor. Such a corridor is extracted from the *corridor map* which is a graph with clearance information. Local motions are controlled by potential fields inside a corridor, providing the desired flexibility.

In this paper, we improve the corridor map, which considerably decreases the extraction times of the paths. We expect that our ideas can also be applied to similar path planning methods such as [1, 8, 12, 15]. We show that with these improvements, the CMM can be used to generate natural paths for thousands of characters in real-time.

Our paper is organized as follows. Section 2 reviews the CMM. We then focus on creating high-quality corridor maps and corresponding data structures in Section 3 and 4. Next, Section 5 shows how the CMM can be used for simulating a large crowd of characters. In Section 6 we conduct experiments to test the improvements and conclude in Section 7 that the method scales well for environments with many characters, each having an independent goal, without compromising the real-time behavior.

*This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie). In addition, part of this research has been funded by the Dutch BSIK/BRICKS project.

2 The Corridor Map Method

The CMM uses an enhanced graph (i.e. the corridor map), storing a system of collision-free corridors. Such a corridor consists of a local backbone path and a set of balls centered around this path. In [5], a corridor $\mathcal{B} = (B[t], R[t])$ is defined as a sequence of balls with radii $R[t]$ whose center points lie along its backbone path $B[t]$. The parameter t is an index ranging between 0 and 1, and the backbone path is defined as a list of coordinates.

To plan a path for a character, which is modeled by a ball with radius r, we need to extract an appropriate corridor from the graph. This corridor, enclosing the future path of the character, is formed by concatenating corridors extracted from the map.

While the corridor guides the global motions of the character, its local motions are led by an *attraction point*, $\alpha(x)$, moving on the backbone path of the corridor toward the goal. The attraction point is defined such that making a step toward this point leads the character, located at position x, toward the goal. As stated in [5], the attraction point attracts the character with force $\mathbf{F}_a$. Let d be the Euclidean distance between the character's position x and the attraction point $\alpha(x)$. Then $\mathbf{F}_a(x) = f\frac{\alpha(x)-x}{||\alpha(x)-x||}$, where $f = \frac{1}{R[t]-r-d} - \frac{1}{R[t]-r}$.

Additional behavior can be incorporated by adding extra forces to $\mathbf{F}_a$, resulting in a force $\mathbf{F}$. The final path is obtained by iteratively integrating $\mathbf{F}$ over time while updating the velocity, position and attraction point of the character.

In the following two sections, we will show how to create high-quality corridor maps.

3 The Corridor Map

An important impact on the quality of the paths is the quality of the corridor map. We will describe a new approach that creates maps satisfying the following five criteria. First, if a path exists in the free space then a corridor must exist in the map, which leads the character to its goal position. Second, the map includes all cycles that are present in the environment. These cycles provide short global paths and alternative routes which allow for variation in the characters' routes. Third, corridors extracted from the map have a maximum clearance. Such a corridor provides maximum local flexibility. Fourth, the map is small with respect to the number of vertices and edges. Such a map facilitates low query times. Fifth, the map facilitates efficient extraction of paths. Creating paths efficiently is crucial because we focus on interactive environments.

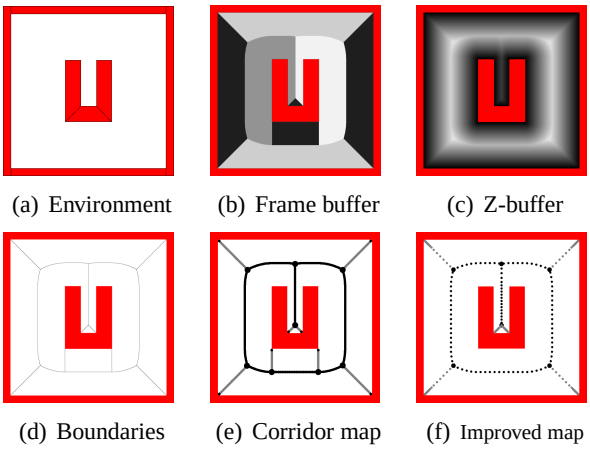
A good base for a data structure satisfying these criteria is the *Generalized Voronoi Diagram* (GVD). A GVD is a decomposition of the free space into regions such that all points p in a region $\mathcal{R}(p)$ are closer to a particular obstacle than to any other obstacle in the environment. Such a region is called a *Voronoi region*. We are particularly interested in the boundaries of the Voronoi regions because they have some useful properties. First, the boundaries form a connected component if the free space in which the character moves is also connected [4]. Second, the union of maximum balls placed on these boundaries results in a complete coverage of the free space. Consequently, a GVD satisfies the first two criteria. Third, the points on the boundaries have a maximum local clearance to the obstacles, which satisfies the third criterion.

We will show in Section 3.1 how the (boundaries of the) GVD can be computed efficiently. These boundaries form the skeleton of the corridor map which is stored in a graph. Each vertex in the graph corresponds to a crossing of boundaries. An edge connects two vertices and traces the boundaries between two Voronoi regions. Such an edge forms the backbone path of a corridor. Because a vertex is placed only at a crossing of boundaries, the corridor map will have a small number of vertices. In practice, their number of incident edges is also small. Hence, the fourth criterion is also satisfied. We will meet the fifth criterion by appropriately sampling the boundaries in Section 3.2 and adding data to the corridor map in Section 3.3.

3.1 Computing the GVD

Approaches for computing the GVD can be divided into two classes: *exact* and *approximate*. An exact approach provides an algebraic description of the GVD. Such an approach poses many challenges in terms of robustness and CPU speed due to the algebraic complexity and high precision calculations that are required [13]. Consequently, the GVD is often approximated in practice.

A promising approach for approximating the GVD is proposed by Hoff *et al.* [8]. The authors describe a brute-force technique that exploits the fast computation of a 2D/3D GVD using graphics hardware. This technique computes a 3D distance mesh, consisting of polygons, for each geometric obstacle present in the environment. Each of the meshes is rendered on the graphics card in a differ-



(a) Environment (b) Frame buffer (c) Z-buffer

(d) Boundaries (e) Corridor map (f) Improved map

Fig. 1. Construction of the corridor map.

ent color. A parallel projection of the upper envelope of the arrangement of these meshes gives the GVD. The diagram can be retrieved from the graphics card's frame buffer and the clearance values (i.e. distance values) can be found in its Z-buffer.

We use this approach to create the GVD's for the environments because it is fast and gives a good approximation of the GVD.

Our environments are composed of geometric primitives such as triangles and polygons. For each of these primitives, a distance mesh is computed and rendered in a different color. Because these primitives are projected onto an array of pixels, we need to specify its dimensions. The *resolution* (i.e. the number of pixels in each dimension) determines the accuracy of the GVD as well as the distances.

As an example, consider the environment depicted in Fig. 1(a). We decomposed the obstacles into seven convex parts. (We need convex obstacles as Hoff's approach does not discover Voronoi boundaries running into concavities.) These parts contribute to the Voronoi regions, depicted in Fig. 1(b) as uniformly colored regions in the frame buffer. Then, the projection of the corresponding distance meshes fills the Z-buffer with distance values. In Fig. 1(c), light gray colors correspond to a large distance while dark gray colors correspond to a small distance to the obstacles. After computing the GVD, we proceed to trace its boundaries. In our example, Fig. 1(d) shows the boundaries corresponding to the frame buffer from Fig. 1(b).

3.1.1 Tracing the boundaries of the GVD

Our goal is to create an annotated graph $G = (V, E)$. Each vertex $\nu \in V$ in this graph stores the coordinates corresponding to the position in the frame buffer F at which three or four adjacent pixels have a different color, and stores a corresponding clearance value which is retrieved from the Z-buffer. An edge $\epsilon \in E$ connects two vertices $\nu', \nu'' \in V$ and stores the corridor (and its length). Its backbone path runs from ν' to ν'' along the *boundaries* of two differently colored regions.

Definition 1 (Boundary)**.** *Let p_1 and p_2 be two adjacent integer coordinates of pixels in the frame buffer or Z-buffer. The space between these two pixels is a boundary if the corresponding pixels in the frame buffer (F) have a different color and their clearance values in the Z-buffer (D) are larger than zero, i.e $F_{p_1} \neq F_{p_2} \wedge D_{p_1} > 0 \wedge D_{p_2} > 0$.*

A vertex with coordinates p_1 can have at most four outgoing edges (i.e. edges heading left, up, right, down). Let p_2 be the coordinates formed by traveling one pixel in each of these directions, respectively. We consider all directions for which p_2 lies on a boundary. For each direction, we trace the boundaries in the frame buffer until we hit the location of a vertex (or obstacle). Because we are dealing with a discrete list of coordinates, it is more convenient to use the concept of a *discrete corridor* instead of the continuous corridor defined in [5].

Definition 2 (Discrete Corridor)**.** *A discrete corridor $\mathcal{B} = (B_i, R_i)$ is defined as a sequence of balls with radii R_i whose center points B_i lie along its backbone path B. For the index i holds that $1 \leq i \leq n$, where n denotes the number of balls.*

Fig. 1(e) shows an example of a graph that was extracted by the algorithm. Its vertices and local backbone paths are displayed as small discs and curved lines, respectively. The two short vertical edges do not belong to the Voronoi boundaries. These artifacts are removed in the next section to improve the running times in the query phase.

3.1.2 Pruning the graph

Hoff's approach may introduce edges which are not part of the GVD, increasing the size of the graph. This occurs when neighboring pixels near two adjacent convex parts have a different color while their boundary is not part of the actual GVD. We remove these redundant edges and corresponding vertices, as follows. We consider all vertices $\nu \in V$ in the graph with out degree one and determine the position (p) of ν in the frame buffer. Next, we consider the colors of the pixels neighboring p, which correspond to unique obstacles. By using simple geometric calculations, we can compute the two closest

points from p to these obstacles. If they are equal, then vertex ν and its adjacent edge is removed.

3.2 Resampling the boundaries of the GVD

Since we are dealing with a set of discrete corridors, we have to decide how coarsely such a corridor needs to be sampled. There is a trade-off between accuracy and efficiency of the algorithm for different sampling densities of balls centered on the backbone path. That is, many samples result in a higher coverage of the free space but they make the extraction of the final path slower. In contrast, few samples lead to a lower coverage of the free space but the reduction of data leads to a faster algorithm.

The number of samples is related to the chosen distance between adjacent balls. Setting it to zero would lead to an infinite number of balls. Since this is not practicable, we set a *lower bound* on the distance, d_{min}. We set d_{min} to the width of a pixel.

To ensure that a path can be found inside the corridor, we have to find an *upper bound* on the distance between two adjacent balls, $\mathcal{B}_i = (B_i, R_i)$ and $\mathcal{B}_{i+1} = (B_{i+1}, R_{i+1})$. This upper bound is dependent on their Euclidean distance, $d(B_i, B_{i+1})$, the radius of the second ball, R_{i+1}, and the radius of the character, r, which is going to traverse the corridor. This dependence can be described by the following relation: $d(B_i, B_{i+1}) + r \leq R_{i+1}$.

A discrete corridor is r-valid if this relation holds for each two adjacent balls in the Corridor:

Definition 3 (r-Valid Discrete Corridor)**.** *A discrete corridor $\mathcal{B} = (B_i, R_i)$ is r-valid if $\forall i$: $d(B_i, B_{i+1}) + r \leq R_{i+1}$.*

Theorem 1 (Existence of a path)**.** *If a Discrete Corridor is r-valid, then a path exists within the corridor for a character with maximum radius r.*

This theorem is proved in our extended paper. The idea is that a character can always be pulled toward the attraction point, which eventually causes the character to reach the goal.

For an original corridor $\mathcal{B}$ holds that $\forall i$: $d(B_i, B_{i+1}) = d_{min}$ because it consists of samples placed on adjacent pixel crossings. For efficiency reasons, we want a coarser sampling of the corridor such that $d(B_i, B_{i+1}) = d_{pref}$ (if possible), where d_{pref} denotes the preferred distance.

We resample each corridor by removing each ball which is too close to its neighbors and has enough overlap to be traversed by the character with maximum radius r_{max}. That is, we remove all balls with coordinates B_i for which $d(B_{i-1}, B_{i+1}) \leq \min(d_{pref}, R_{i+1} - r_{max})$.

We refer the reader to Fig. 1(f) for an example of a resampled corridor map.

3.3 Adding data to the corridor map

We add information to elements of the corridor map to speed up the query phase. First, we identify the connected components for each vertex in the graph. As a result, we can quickly check whether two vertices (to which the start and goal are connected) can be connected by a path in the graph. Second, for all corridors, we compute the maximum radius a character can have for traversing the corridor to speed up finding the shortest backbone path in the map. By rearranging the expression $d(B_i, B_{i+1}) + r \leq R_{i+1}$, it becomes clear that the maximum value for r is determined by the minimum value of $R_{i+1} - d(B_i, B_{i+1})$ of all adjacent balls $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$. We store this value, cl_{max}, in the data structure representing the corridor.

4 Querying

In the online query phase, we have to perform the following four steps to find a path for a character. First, find the closest ball $\mathcal{B}_s = (B_s, R_s) \in \epsilon_s \in E$ enclosing the character placed at its *start* position s. In addition, find the closest ball $\mathcal{B}_g = (B_g, R_g) \in \epsilon_g \in E$ enclosing the character placed at its *goal* position g. Note that edge ϵ_s (ϵ_g) is the edge in the graph containing ball $\mathcal{B}_s$ ($\mathcal{B}_g$). Second, connect the start and goal to the graph and find the shortest backbone path between B_s and g. Third, compute the corridor with a backbone path connecting B_s to g. Fourth, compute the path. We will now provide the details of these steps.

4.1 Finding the closest ball

We need to find the closest ball in the corridor map (i.e. the ball whose center is closest to the character) enclosing the character placed at its start and goal position, respectively. A brute-force approach for finding them is to check each ball in each corridor while keeping track of the closest ball, leading to a query time of $O(n)$, where n is the number of balls in the map. Since this may be too expensive for large maps, we build a search structure that is based on a kd-tree [2]. Such a tree takes $O(n \log n)$ time to build and uses $O(n)$ memory. We recursively subdivide the space into smaller spaces using

alternating axis-aligned lines, separating two non-overlapping sets of balls. These balls do not intersect with the line. Balls that do intersect are stored in an interval tree [2] together with each node in the kd-tree. Such an interval corresponds to the segment of the axis-aligned line whose end points intersect with the bounding box of the ball.

For a character with radius r and position q, we can compute the closest ball $\mathcal{B}_q = (B_q, R_q)$ in $O(k + \log^2 n)$ time, where k is the number of reported results. (This is proved in the extended paper). For each reported ball, we check whether it encloses the character. A character is enclosed by ball $\mathcal{B}_q$ if $d(q, B_q) < R_q - r$. From the enclosed balls, we compute the one closest to the character.

4.2 Finding the shortest backbone path

We described a data structure for finding balls $\mathcal{B}_s = (B_s, R_s) \in \epsilon_s$ and $\mathcal{B}_g = (B_g, R_g) \in \epsilon_g$, enclosing the start position s and goal position g, respectively. Because we have stored a connected component number for each vertex, we can now determine in constant time whether there cannot be a path between s and g. Hence, we can stop searching for the shortest backbone path if the numbers differ.

Let now s' and g' be the two vertices with coordinates B_s and B_g, respectively. To find a shortest path in the graph between s' and g, we split edges ϵ_s and ϵ_g such that they include s' and g'. (Note, to create a shorter backbone path, that the backbone path starts at s' and not at position s. This is valid because both s and s' are enclosed by ball $\mathcal{B}_s$.)

The shortest backbone path is a path connecting s' to g with the minimum distance. The distance between two adjacent vertices can be retrieved from the map in $O(1)$ time because it has been computed in the construction phase. If a character with radius r cannot traverse the corridor corresponding to the local path between the two vertices, i.e. $r > cl_{min}$, the distance is set to ∞. Hence, the edge incident to these vertices will not be part of the shortest path.

The shortest path can be found by e.g. Dijkstra's algorithm or the A* algorithm. While their theoretical running times are equal, i.e. $O(|V| \log |V| + |E|)$ time [16], the latter is often faster in practice because A* cuts down on the subgraph that must be explored by using a heuristic function guiding the search toward the goal. A function that often works well is the Euclidean distance between a vertex and the goal because virtual environments usually contain many cycles. Consequently, the Euclidean distance is a good estimator for the graph distance.

4.3 Computing the corridor

The shortest backbone path, connecting s' to g, is a list of vertices from the graph. Now let $\epsilon_1 \dots \epsilon_n \subseteq E$ be the sequence of edges connecting s' to g. The backbone path $B[t]$ is then given by $\Pi_{\epsilon_1} \oplus \dots \oplus \Pi_{\epsilon_n}$, where operator $\oplus$ concatenates the local paths Π_{ϵ_i}.

Recall that a corridor is composed of a backbone path and radii of the balls placed on this path. All radii can be retrieved from the local paths corresponding to the edges, except for the last edge, ϵ_n, connecting g' to g. Now let p be a position on ϵ_n. Then its radius is given by $R_g - d(B_g, p)$, where $d(B_g, p)$ is the Euclidean distance between center of the goal ball and the position on the edge. The distance between the samples on edge ϵ_n are chosen such that ϵ_n is r-valid.

4.4 Computing the path

While the global path of the character is determined by the corridor, its local path is determined by forces applied to the character. We define force $\mathbf{F}$ as the sum of $\mathbf{F}_a$ and other forces, $\mathbf{F}_o$, i.e. $\mathbf{F}(x) = \mathbf{F}_a(x) + \mathbf{F}_o(x)$. The force $\mathbf{F}$ causes the character (positioned at x) to accelerate, pulling it toward the goal. This phenomenon is summed up by Newton's Second Law, i.e. $\mathbf{F} = Ma$, where M is the mass of the character and a is its *acceleration*. Without loss of generality, we assume that $M = 1$. Hence, the force can be expressed as $\mathbf{F}(x) = \frac{d^2x}{dt^2}\ m/s^2$. Combining the two expressions for $\mathbf{F}(x)$ gives us $\frac{d^2x}{dt^2} = \mathbf{F}_a(x) + \mathbf{F}_o(x)$, which is an equation providing the positions for the character. Because this equation cannot be solved analytically, we have to revert to a numerical approximation.

Many approximation methods are available such as Euler and Verlet integration [3]. There is a trade-off between the precision of the method and its efficiency. While basic methods (e.g. Euler) are more efficient but less inaccurate, more advanced methods (e.g. Verlet) are slower but more accurate. We need Verlet's precision because many forces are going to be applied to a character (see Section 5).

In this integration scheme, the character's velocity is unbounded which may lead to inaccurate calculations and unnatural paths (because the path gets more curved for higher speeds). Hence, we set a maximum on the velocity (referred to as v_{max}).

The final path Π can be obtained by iteratively applying the scheme while updating the character's position and corresponding attraction point until the character is near the goal (i.e. their distance is within some small threshold δ).

5 Simulating a crowd

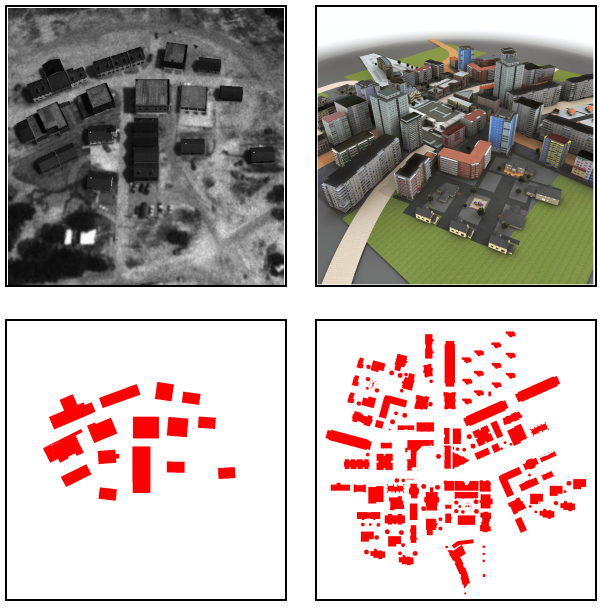
We want to show that the CMM scales well to environments with many independent characters by using the method for simulating a large crowd.

Real-time crowd simulation has gained much attention recently [7, 12, 15]. It requires the modeling of group behavior, pedestrian dynamics, path planning and graphical rendering [15]. We focus on the path planning part.

In our simulation, each character will be assigned a random start and goal position which together fix a corridor. When a character has reached its goal, a new goal will be chosen, *ad infinitum*. Because we are dealing with more than one character, we have to choose an appropriate force $\mathbf{F}_c$ such that characters evade each other naturally. We use Helbing and Molnár's social force model for collisions avoidance because their simulations have shown that it exhibits realistic crowd behavior [7]. The model describes three functions, representing the acceleration toward the desired velocity of motion, the behavior of characters keeping a certain distance to other characters and borders, and attractive effects among characters. Force $\mathbf{F}_a$ captures the effect of their acceleration force and the force keeping characters away from borders. As collision avoidance force, $\mathbf{F}_c$, we use the force described in equation (3) of their paper. Unlike [7], we do not model attractive effects among characters.

In each iteration of the simulation, we need to find the set of neighbors for each character. Since this operation is carried out many times, we have to revert to an efficient data structure for answering nearest neighbors queries. We maintain a 2D grid storing the locations of all characters (with radius r). Each cell in the grid stores a sorted set of character id's. Preliminary experiments have shown that the cell size c can be chosen fairly small, e.g. $c = 3 + 2r$ meter, to obtain realistic collision avoidance. By including the term $2r$, we only have to check 3 by 3 cells for carrying out a nearest neighbors query. Also, an update corresponding to a changed position of a character is efficient because this can be achieved in $O(\log k)$ time where k is the number of characters in the appropriate cell.

The simulation consists of some user defined number of iterations. For each character, the task per iteration includes computing the forces $\mathbf{F}_a$ and $\mathbf{F}_o = \mathbf{F}_c$, integrating the final force $\mathbf{F}$, adding the new position of the character to its path, and possibly choosing a new goal (and corresponding corridor).



(a) McKenna (b) City

Fig. 2. The two test environments. The first row shows the 3D models and the second row the corresponding footprints, used for constructing the corridor maps.

6 Experiments

We created corridor maps for two environments and tested whether they could be used for fast querying. Then, we checked whether the CMM could produce high-quality paths for a large crowd in real-time.

6.1 Experimental setup

We implemented two applications in Visual C++ under Windows XP. The first one was used for generating the corridor maps and was built on top of the HAVOC library [8] which provides functions for computing the GVD. The second one was used for simulating the crowd and was integrated in our CMM framework [5].

All the experiments were run on a PC with a NVIDIA GeForce 8800 GTX graphics card and an Intel Core2 Quad CPU (2.4 GHz) with 4 GB memory. Our application used one core. We conducted experiments with the two environments depicted in Fig. 2. They have the following properties:

McKenna This environment is a model of the McKenna MOUT facility [17], which is used to train soldiers for operations in urban environments. The model measures 200x200 meter and consists of 566 polygons. Its footprint consists of only 46 triangles.

City We modeled a city which is much larger (i.e. 500x500 meter) than the previous environment. The geometry is composed of 220K triangles. Its footprint consists of 2122 triangles. There

are many alternative routes, and large ample spaces as well as narrow passages.

The characters' radius was 25cm ($r = 0.25$).

6.1.1 Setup for the corridor maps

We set the minimum distance between sample points to 12.5 cm, i.e. $d_{min} = \frac{1}{8}$, so the resolution of the maps were 1600x1600 and 4000x4000 pixels for the McKenna and City environments, respectively. We set the preferred distance between samples as well as r_{max} to 1, i.e. $d_{pref} = r_{max} = 1$.

We carried out the following two experiments. First, we measured the number of vertices, edges and samples of the corridor maps in three cases, studying the impact on the maps of resampling and pruning. We also recorded the construction times. Second, we measured the performance of the maps by recording the costs of A* vs. Dijkstra for 10,000 random queries, and the costs of extracting 10,000 corridors corresponding to 10,000 (valid) queries.

6.1.2 Setup for the crowd simulation

We steered the characters by computing force $\mathbf{F} = \mathbf{F}_a + \mathbf{F}_c$. We integrated $\mathbf{F}$ with Verlet integration with step size $\Delta t = 0.1$ and set the maximum speed, v_{max}, to 1.2 m/s [9]. The cell size of the grid was set to $c = 3 + 2r = 3.5$ meter.

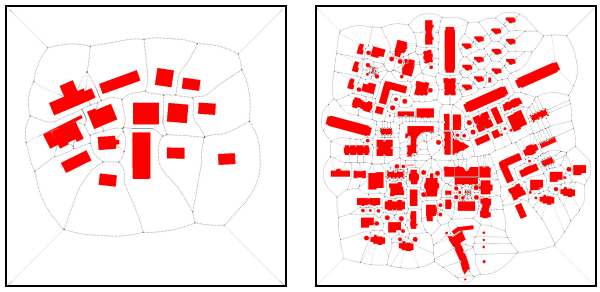
In the experiments, we used the resampled and pruned maps and measured the CPU-load for a varying number of characters. We defined the CPU-load as the total CPU time / averaged traversed time $*$ 100%. When the CPU-load was at most 100%, we considered the performance as real-time.

6.2 Experimental results

6.2.1 Results for the corridor maps

For each environment, we created three corridor maps. The first one was neither resampled nor pruned, the second one was only pruned, and the third one was both resampled as well as pruned. Table 1 shows the results and Fig. 3 displays the corridor maps corresponding to the latter case.

The results make clear that pruning reduces the number of vertices and edges of the underlaying graph by at least 15%. Also, redundant edges were successfully removed. When the edges were pruned, the numbers of samples were reduced by almost an order of magnitude, leading to faster running times in the query phase. The construction times were small (i.e. < 1 s). Approximately 85% was spent on computing the GVD and 15% was spent on resampling and pruning of the map.



(a) McKenna (b) City

Fig. 3. The two corridor maps (resampled and pruned).

Table 1. Results for the corridor maps.

McKenna	**no resampl. no pruning**	**no resampl. pruning**	**resampl. pruning**
Nr vertices	72	56	56
Nr edges	77	70	70
Nr samples	17,102	16,885	1,927
Running time (s)	0.05	0.05	0.05
City			
Nr vertices	1,814	1,434	1,434
Nr edges	1,986	1,606	1,606
Nr samples	177,626	166,662	25,863
Running time (s)	0.64	0.72	0.64

In the second experiment, we extracted 10,000 corridors from each environment. The average extraction time for one corridor (excluding the times for finding the shortest path and locating the query) was 0.17 ms and 0.87 ms for the McKenna and City environments, respectively. Table 2 summarizes how much the extraction times were relatively increased for the two different choices of finding the shortest path and locating the start and goal.

The results confirmed that A* is faster than Dijkstra and that a kd-tree performed better than linear search in our environments. The results suggest that the differences become larger when a corridor map increases in size. In the crowd simulation, we used the optimal choices, i.e. the pruned and resampled corridor maps with A* and the kd-tree. These choices led to fast extraction times of the corridors, i.e. averaged 0.19 ms and 1.19 ms per corridor for the McKenna and City environments, respectively.

6.2.2 Results for the crowd simulation

Fig. 4 shows that $\pm$10,000 characters can be simulated in real-time. For up to 7,000 characters, the simulation in the McKenna environment cost less time due to its small map. After this threshold, McKenna became rather crowded compared to the

Table 2. Average relative increase of time for extracting a corridor: A* vs Dijkstra and kd-tree vs linear search.

	McKenna		City	
	A*	**Dijkstra**	**A***	**Dijkstra**
Increase of time	9.8%	21.0%	11.5%	138.2%
	kd-tree	**linear**	**kd-tree**	**linear**
Increase of time	3.1%	27.7%	1.0%	59.7%

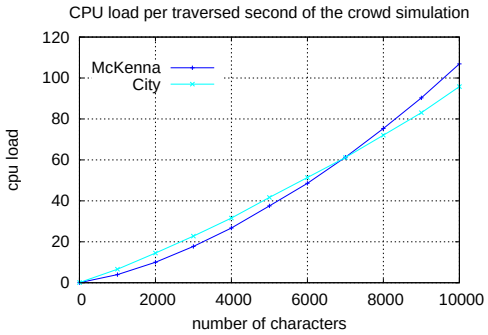


Fig. 4. The relation between the number of characters and the CPU-load per second traversed time.

City, decreasing its performance. We conclude that the CMM can be used for real-time path planning with many characters.

7 Conclusion

The Corridor Map Method (CMM) is a framework for real-time path planning in interactive virtual environments. The strength of the CMM is that it combines a fast global planner with a powerful local planner, providing real-time performance and the flexibility to model natural behavior of characters.

In this paper, we focused on creating efficient corridor maps and appropriate data structures which are used for efficient path planning. As proof of concept, we showed that the CMM can be used for simulating a large crowd of characters, each having an independent goal, without compromising the real-time behavior.

In future work, we will extend the CMM such that the method can also deal with three-dimensional problems (e.g. problems involving a non-flat terrain). In addition, we will study how the method can be used for creating camera paths, alternative paths and tactic paths.

References

[1] J. Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin, "Interactive navigation of individual agents in crowded environments," in *Symposium on Interactive 3D Graphics and Games*, 2008, pp. 139–147.

[2] M. Berg, M. Kreveld, M. Overmars, and M. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2000.

[3] J. Butcher, *Numerical Methods for Ordinary Differential Equations*. Wiley, 2003.

[4] H. Choset and J. Burdick, "Sensor-based exploration: The hierarchical generalized Voronoi graph," *International Journal of Robotics Research*, vol. 19, pp. 96–125, 2000.

[5] R. Geraerts and M. Overmars, "The corridor map method: A general framework for real-time high-quality path planning," *Computer Animation and Virtual Worlds*, vol. 18, pp. 107–119, 2007.

[6] ——, "Creating high-quality paths for motion planning," *International Journal of Robotics Research*, vol. 26, pp. 845–863, 2007.

[7] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Physical Review*, vol. 51, pp. 4282–4287, 1995.

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

[9] R. Knoblauch, M. Pietrucha, and M. Nitzburg, "Field studies of pedestrian walking speed and start-up time," *Transportation Research Record*, pp. 27–38, 1996.

[10] J.-C. Latombe, *Robot Motion Planning*. Kluwer, 1991.

[11] S. LaValle, *Planning Algorithms*. http://planning.cs.uiuc.edu, 2006.

[12] F. Morini, B. Yersina, J. Maïm, and D. Thalmann, "Real-time scalable motion planning for crowds," in *International Conference on Cyberworlds*, 2007, pp. 144–151.

[13] A. Okabe, B. Boots, K. sugihara, and S. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed. John Wiley, 2000.

[14] E. Rimon and D. Koditschek, "Exact robot navigation using artificial potential fields," *IEEE Transactions on Robotics and Automation*, vol. 8, pp. 501–518, 1992.

[15] A. Sud, R. Gayle, E. Andersen, S. Guy, M. Lin, and D. Manocha, "Real-time navigation of independent agents using adaptive roadmaps," in *ACM symposium on Virtual reality software and technology*, 2007, pp. 99–106.

[16] K. Trovato, "A* planning in discrete configuration spaces of autonomous systems," Ph.D. dissertation, Universiteit van Amsterdam, 1996.

[17] US Army Training and Doctrine Command, "Fort benning," Internet: http://www.globalsecurity.org/military/facility/fort-benning.htm, 2006.