Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 67 (2007) 91–104

Science of Computer Programming

www.elsevier.com/locate/scico

High quality navigation in computer games

D. Nieuwenhuisen*, A. Kamphuis, M.H. Overmars

Center for Geometry, Imaging and Virtual Environments, Institute of Information and Computing Sciences, Utrecht University, Padualaan 14, 3584 CH Utrecht, The Netherlands

Received 1 October 2005; received in revised form 15 March 2006; accepted 26 June 2006
Available online 6 April 2007

Abstract

Navigation plays an important role in many modern computer games. Currently the motion of entities is often planned using a combination of scripting, grid-search methods, local reactive methods and flocking. In this paper we describe a novel approach, based on a technique originating from robotics, that computes a roadmap of smooth, collision-free navigation paths. Because the vast amount of computation time is spent in the pre-processing phase, navigation during the execution of an application is almost instantaneous. The created roadmap can be queried to obtain high quality paths. Furthermore, the applications of the roadmap are not limited to navigating an entity. Therefore, besides navigation for an entity, two other applications are presented; one for planning the motion of groups of entities and one for creating smooth camera movements through an environment. All applications are based on the same underlying techniques.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Navigation; Roadmap; Camera movement; Groups; Computer games

1. Introduction

Many games genres, specifically Real Time Strategy (RTS) and (tactical) First/Third Person Shooters (FPS/TPS), contain large numbers of entities. These entities can for example be world inhabitants, non-playing characters (NPCs), and vehicles. These games heavily depend on the correct and believable motion of entities. To create those motions, the games must plan the paths of entities between locations in the game world. Currently this is typically achieved using a combination of scripting, A^*-based grid search, local reactive methods and flocking.

In scripting, the designer explicitly described the paths that can/must be followed by the entities. This is normally part of level design. Scripting is a time-consuming process for the designer. In addition, it can lead to repetitive behavior that is easily observed by the user. Scripting gets increasingly complicated when many entities move in the same space and when spaces become larger (as is the case in modern gaming).

Grid-based methods divide the world in a grid of cells and plan motion using an A^*-based search on the free cells (see e.g. [1,2]). If the game world is small and grid-like, this technique is useful. However, when the world becomes complicated and large and many entities must move around, grid-based methods take a large amount of computation time. Pruning the search reduces the time but might lead to wrong paths. Also, motions created by grid

* Corresponding author.
E-mail addresses: dennis@cs.uu.nl (D. Nieuwenhuisen), arnok@cs.uu.nl (A. Kamphuis), markov@cs.uu.nl (M.H. Overmars).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.06.008

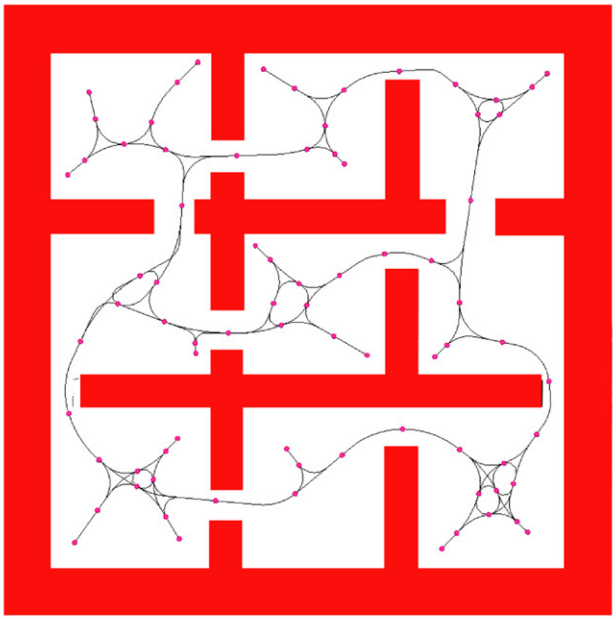


Fig. 1. An example of a smooth roadmap computed by our technique. The roadmap was created by our implementation.

search tend to look unnatural. In order produce smooth paths using grid searches, the paths need to be smoothed in the post-processing phase resulting in expensive queries.

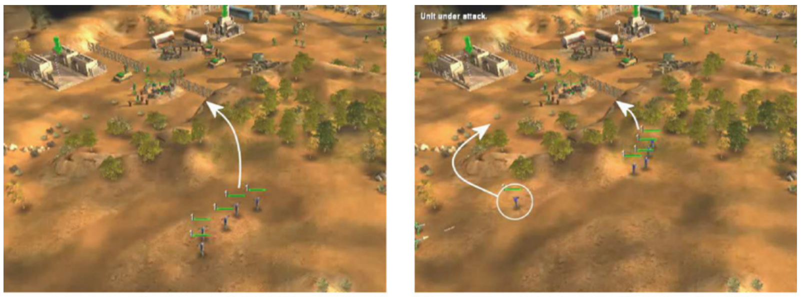
Reactive methods adapt a previously computed motion to obstacles found near the path that were not taken into account during initial planning; for example other entities or small, movable objects (see e.g. [3–6]). Even though theoretically reactive methods can be used to compute full paths, this normally leads to deadlock situations in which the entities no longer know where to move (e.g. they get stuck in a corner of the room). Another problem with reactive methods is that it is often difficult to adapt the internal animation of the character to the motions produced.

Flocking originates from the animation community. Reynolds first introduced the approach in [7]. Based on the natural phenomena seen in birds and sheep, the entities in flocking look to their direct neighbors and, based on some simple rules, try to do the same as them. This results in very natural movement of the entities, but guarantees nothing on the paths. For example, it is not guaranteed that the entities will ever reach their goal. Flocking is often used in conjunction with other method to overcome this problem, typically by guiding one or more entities using other motion planning techniques, see e.g. [8].

In addition to the above issues, in some applications the game world is not completely known at design time because it contains for example a door or a moving car (see e.g. [27,28]). Also there is a tendency to allow games to automatically generate game worlds rendering existing navigation methods less efficient. Therefore there is a need for algorithms that automatically (i.e. without the need of a human designer) allow for high quality navigation using only negligible computation time.

During the last three decades, many path planning approaches have been developed in the field of robotics, that may be applicable to path planning in games. One popular path planning technique in robotics is the Probabilistic Roadmap Method (PRM). It has been studied by many authors, see e.g. [9–12,24]. In a pre-processing phase this method builds a roadmap of possible motions of the robot through the environment. When a particular path planning query must be solved, a path is retrieved from this roadmap using a simple and fast graph search. The PRM approach is suited for very complicated environments. Unfortunately, the roadmap produced by the method can be rather wild, leading to low quality paths consisting of straight line segments, that require a lot of time-consuming smoothing during game play.

In this paper we will describe a novel path planning approach, building on the PRM method, that can be effectively applied to games. The approach also constructs a roadmap of possible motions but guarantees that the paths are short, have enough clearance from the obstacles, and are C^1 continuous, leading to natural looking motions. See Fig. 1 for an example of a roadmap computed using our approach. After constructing the roadmap, which can be done in the pre-processing phase, paths can be retrieved almost instantaneously, and do not require any post-processing.



(a) A group of five characters should attack the site pointed to by the arrow. (b) The group inappropriately splits up, loosing some of its troops.

Fig. 2. One of the problems with the current techniques for motion planning for multiple units is that the group splits up to reach the goal. This scene was taken from Command and Conquer: Generals from EA Games.

Besides the standard application, in which the roadmap is used for planning the paths for individual entities, we describe two additional applications. First, we consider the motion of groups of entities. This problem is often solved using a combination of grid-based planning and flocking [7,13]. Unfortunately, this can lead to unwanted behavior where the group of entities splits up (see Fig. 2 for an example). We will use the smooth paths computed by the new planning approach as a backbone path and then use a social potential field approach to guide the flock through a corridor around this path (extending our earlier work in [14]). This results in a natural motion in which the group is guaranteed to stay together.

In many games, moving the camera through an environment is one of the most fundamental operations. Currently the camera is often under direct control of the user or under indirect control of the user. Direct camera control is difficult, it easily leads to motion sickness due to redundant motions, and is often not required. Building on our earlier work in [15] we describe a method in which the user only specifies positions of interest and the camera automatically moves to such positions using a smooth, collision-free motion. For computing the camera path we will use the new planning approach described. This is then combined with techniques to control the view direction and the speed of the camera to obtain a camera motion that is pleasant to watch.

2. Roadmap generation

In this section we will describe how, in a pre-processing phase, a roadmap of possible motions for the entities can be computed. This roadmap can be stored together with the scene and is used to create convincing motions for an entity in very little time. Because the vast amount of work is done in pre-processing, the creation of a motion through an environment does not take longer than a few milliseconds.

A roadmap is normally represented as a graph in which the nodes correspond to placements of the entity and the edges represent collision-free paths between these placements. A standard technique for automatic roadmap creation is the probabilistic roadmap method (PRM).

Unfortunately, the PRM method leads to low quality roadmaps that can take long detours. This is due to the random nature of the PRM method. Techniques exist to improve paths, but these have to be initiated after the query and thus consume valuable computing time during the execution of the application. Here we present a variant of the PRM method that creates short, smooth and high quality roadmaps in the pre-processing phase. These roadmaps can then in real time be used to solve path planning queries almost instantaneous using a simple shortest path graph search algorithm (for example A^* search).

In the pre-processing phase we create the roadmap graph, consisting of vertices (V) and edges (E). For the placement of the entity we only take its position (x, y) into account since these are the only parameters that are important for planning the path. Later, the other parameters (such as orientation) can be added depending on the application. The edges of the graph will represent straight line and circular paths between the vertices. Only vertices and edges that are collision free are allowed in the graph.

In order to be able to use the graph for as many different queries as possible, we need a good coverage of the space. Many improvements for the PRM method have been proposed in order to achieve this (see e.g. [16–22]), but all are based on the same underlying concept and lead to roadmaps consisting of straight line segments only that result in low path quality.

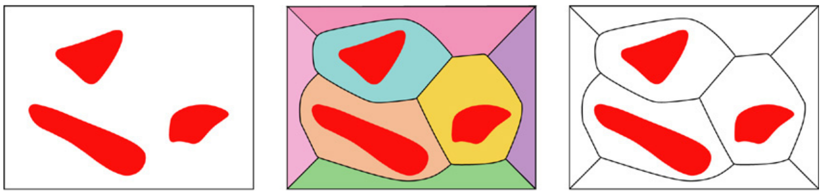

(a) An scene with 3 obstacles. (b) The Voronoi regions of the scene. (c) The boundaries of the Voronoi regions together form a nice road map that could be used by an entity for navigation.

Fig. 3. An example of a Voronoi diagram. The boundaries of the workspace are treated as obstacles.

For a roadmap that is used to navigate entities in games, we can formulate the following criteria:

- The paths generated by the roadmap should always keep some minimum amount of clearance from the obstacles in the scene.
- The paths should be smooth i.e. it should be C^1 continuous.
- Paths should be short, not taking any significant detours.
- A path needs to be created very fast (not delaying the motion). To achieve this, as much work as possible should be done during pre-processing.

2.1. *Creating samples on the Voronoi diagram*

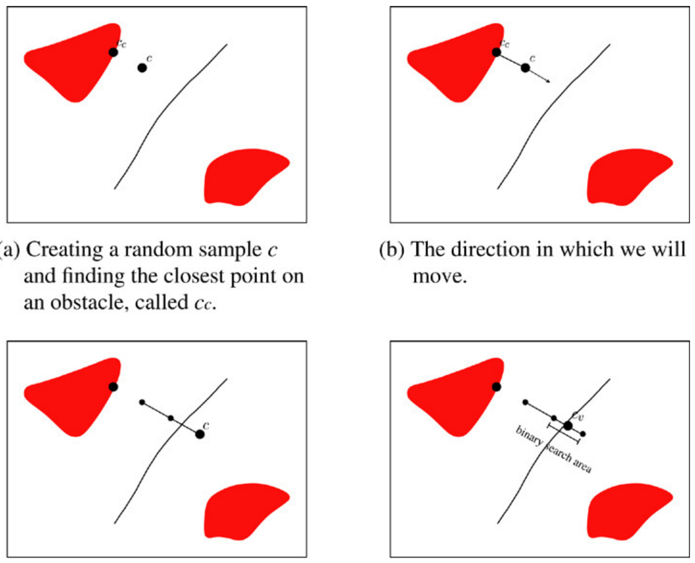
Given the criteria of the previous section, an ideal roadmap for our purpose would be the roadmap formed by the boundaries of the regions of the *Voronoi diagram*. The Voronoi diagram of a scene defines for every obstacle a set of points in the free space that are closer to this obstacle than to any other obstacle in the scene; together these points form the Voronoi diagram. The sets of points form the Voronoi regions (see Fig. 3(a), (b) for an example).

As can be seen from Fig. 3 the boundaries of the Voronoi regions would make a nice roadmap that complies to the criteria of the previous section. If an entity could use this roadmap for navigation, it would maintain a maximum clearance with the obstacles, resulting in a natural looking path for an entity. Also the pats are smooth and a shortest path could be extracted by using, for example, A^*. Unfortunately, computing the Voronoi diagram in an exact manner is infeasible in realistic environments where obstacles can have arbitrary shapes.

Therefore we propose an algorithm that does not compute the Voronoi diagram exactly, but rather results in a roadmap that approximates it. In practice there is no difference between navigation using an exact computation of the Voronoi diagram or a good approximation of it. Our algorithm is based on a new variant of the PRM method. It works as follows. In every iteration of the algorithm we randomly pick a sample (placement) of the entity, called c. Then we check whether c is collision free for the entity. If it is not collision free, another random sample is picked. If it is collision free, we will move it to a position close to a Voronoi region boundary. For this, first the closest obstacle point from c is calculated. This point is called c_c, an example is shown in Fig. 4(a). Next, c is moved in the opposite direction of the line between c and c_c. This direction is shown in Fig. 4(b). The step size of moving c is equal to the original distance between c and c_c. We continue moving c until the closest obstacle to c changes, as shown in Fig. 4(c). This change indicates that we have passed a Voronoi region boundary.

Because we have used a relatively large step size, we may have moved too far and thus have move too far from the Voronoi region boundary (as is the case in Fig. 4(c)). Therefore we continue by using binary search between the last position before we crossed the boundary and c, with precision ϵ until c has its two closest obstacles at the same distance. This sample, called c_v, is at most a distance ϵ away from the Voronoi diagram (Fig. 4(d)). Now, we add c_v to the list of vertices V in the roadmap graph.

This procedure is repeated until a sufficient amount of samples have been created that adequately represent the Voronoi boundaries. After every addition of a sample as a vertex to the graph, we determine its neighbor vertices. Searching for neighbor vertices can efficiently be implemented using a Kd-tree [29]. The set of neighbor vertices of vertex v, called N_v is defined as all vertices V that are closer to v than some chosen maximum neighbor distance. For each vertex v_n in N_v we test whether the straight line connection between v and v_n is collision free. If this is the case,



(a) Creating a random sample c and finding the closest point on an obstacle, called c_c.

(b) The direction in which we will move.

(c) The first time c' is closer to another obstacle.

(d) Using binary search to find a point close to theVoronoi region boundary.

Fig. 4. Retracting samples to the Voronoi diagram. The Voronoi region boundary is shown as the thin line.

we add the connection (v, v_n) as an edge to the set of edges E of the graph. If two vertices are already connected in the graph (via other vertices), then we only add the new edge if the path between v and v_n is shortened considerably by at least some constant K (for more details see [23]).

2.2. Retracting edges

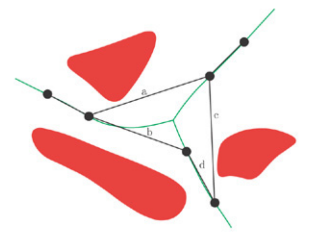
We have retracted the vertices of the graph to the Voronoi diagram (within a certain boundary ϵ) but the connections between them are usually not on the Voronoi diagram and can get very close to obstacles (Fig. 5). Thus, entities using the graph as a guide for their motions also get very close to obstacle boundaries, which contradicts our criteria.

In order to solve this problem, edges are retracted to the Voronoi diagram as well until every part of the edge is at least some pre-specified distance away from the obstacles. We achieve this by proceeding in the following manner: if (a part of) an edge is too close to an obstacle, this edge is split in two equal length parts and the middle point is retracted to the Voronoi diagram using the same procedure as described in the previous section. This procedure is recursively repeated for the two new edges until every edge has enough clearance with the obstacles. An example of this procedure is shown in Fig. 5. In some cases (when the edge passes through a very narrow corridor), the clearance threshold will never be reached and the edge will be split an infinite number of times. In order to prevent this, we stop retracting edges if their length is shorter than some predefined value.

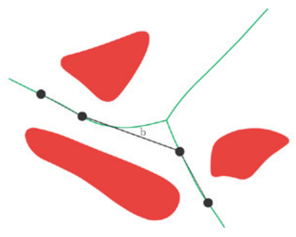
Retracting edges to the Voronoi diagram may result in some edges overlapping each other which could lead to erratic motions of the entity. For example in Fig. 5(a) if edge c is retracted, it will overlap with edge d. This is shown in Fig. 5(e). Fortunately, detecting overlapping edges is easy. For every pair of edges e_i and e_j, we check how far their endpoints are away from the other edge. If this distance is smaller than some predefined distance, then we try to project the vertices of e_i on e_j and vice versa. If at least one of these projections is successful, we call the two edges overlapping and we can join them. We can distinguish four different types of overlapping edges. In Fig. 6 these are shown together with the situation after removing the overlap. The type of overlap in Fig. 5(e) for example is equal to that of Fig. 6(c).

2.3. Circular blends

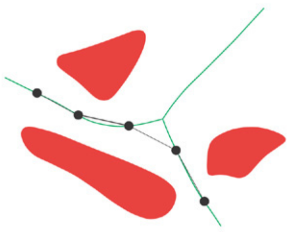
After retracting the vertices and the edges, our graph approximates the Voronoi region boundaries. However, this approximation consists of small straight line segments. If an entity follows such a path it will have C^1 discontinuities



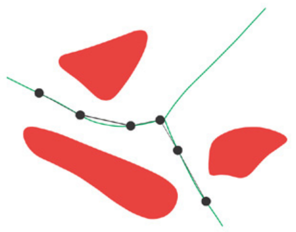
(a) Although the vertices are on the boundaries of the Voronoi regions, the edges can get very close to obstacles.



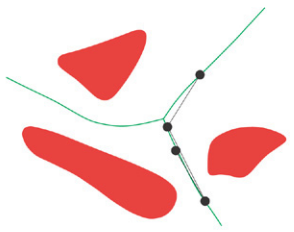
(b) Edge b will be retracted to the Voronoi diagram.



(c) We retract the center edge b to the Voronoi diagram, creating two new edges.



(d) One of the new edges is still not close enough to the Voronoi region boundary. The process is repeated for this edges. Now all (new) edges have enough clearance with the obstacles.



(e) If edge c is retracted, the resulting edges overlap with edge d.

Fig. 5. Retracting an edge to the Voronoi diagram.

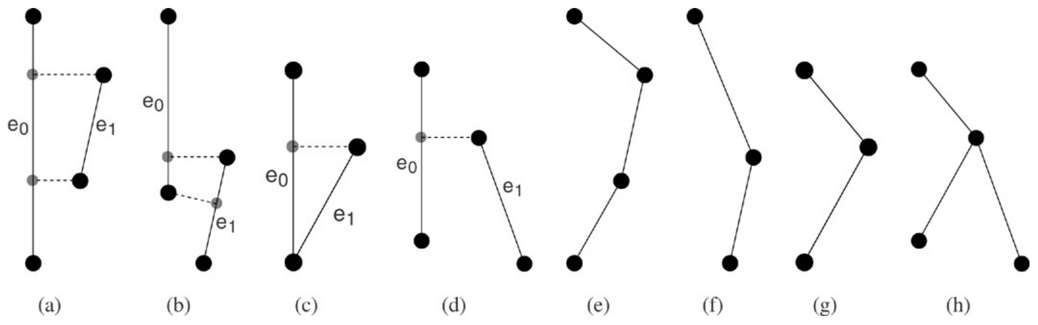


Fig. 6. Merging two overlapping edges e_0 and e_1. Four different cases can be distinguished (a..d). The results after merging are shown in (e..h).

in its motion at the vertices that cause sudden directional changes. In order to solve this problem we will replace parts of the straight line edges by circular blends.

The degree of a vertex is defined as the number of edges that is connected to this vertex. If a vertex has degree 1, it is an endpoint of a path segment, and no circular blend needs to be added. If a vertex has degree 2, the addition of the circular blend is straightforward. We find the centers of the two edges, and use these to create a circle arc that touches both edges. Now, we use this to replace a part of the path (see Fig. 7(a)). If the degree of a vertex v is higher than 2, we find the centers of all incoming edges. We now add a blend for every pair of these centers (see Fig. 7(b) for an example of a vertex with degree 3).

In the previous section, we retracted the edges of the graph to the Voronoi until they had at least some predefined clearance. Adding circular blends may decrease this clearance. Since we do not want the clearance to be lower than some predefined value, we check the minimum clearance of each circular blend. If it is too low, then we replace the blend by another blend that has a smaller radius. We repeat this until the blend has enough clearance. This procedure is shown in Fig. 7(c).

Fig. 1 shows an example of a typical roadmap graph created with this method. Computing this roadmap took about 1 s on a Pentium IV 2.4 GHz.

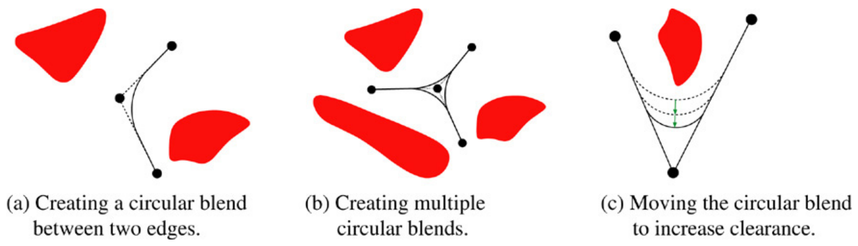

(a) Creating a circular blend between two edges.
(b) Creating multiple circular blends.
(c) Moving the circular blend to increase clearance.

Fig. 7. Creating the circular blends.

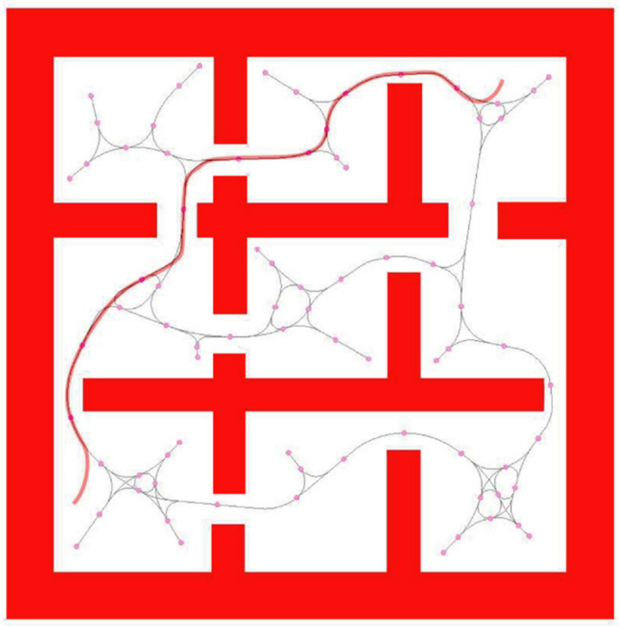


Fig. 8. An example of query to create a path for an entity. The thick line shows the resulting path that complies to all our criteria.

2.4. Answering path queries

Recall that the described algorithm is run in the pre-processing phase. Therefore, a graph can be stored with a scene and loaded when necessary. Whenever an entity has to move to a new location, it can search the graph in real time to plan its route through the environment. Because of the properties of the roadmap, paths will be short, have enough clearance from the obstacles, and are smooth. To obtain a path we first need to connect the current position of the entity to the graph. This can easily be done by finding the closest vertex in the graph and connecting the current placement to this vertex using circular blends. We proceed in the same way for the goal placement of the entity. Now we can use a shortest path algorithm to find a path between the current and goal positions. See Fig. 8 for an example of such a path.

Computing paths is extremely fast. Even for a large roadmap graph consisting of 1000 vertices and 3000 edges, the calculation of the shortest path takes less than 10 ms on a Pentium IV 2.4 GHz.

3. Applications

The roadmap generated by the method described in the previous section can be used to plan the paths for single entities. The resulting paths can be generated very fast and the quality of the paths is high (no large detours, no sharp turns). However, this is not the only possible use of the high quality roadmap. In this section we give two applications where the high quality roadmap is used. The first application is path planning for groups. In this application we use the path for a single entity queried from the roadmap as a basis for the paths of a group. The second application is the planning of camera motions. Again, the path from the high quality roadmap is used for the motion of the camera.

3.1. *Path planning for groups*

Virtual worlds are often populated with a large number of moving entities. The entities should often behave as a coherent group rather than as individuals. For example, when one needs to simulate the behavior of whole army divisions. Current techniques solve the problem of path finding on the entity level, i.e. they plan the motion of individual entities, using techniques like flocking to keep the entities together. However, in cluttered environments this often leads to loss of coherence. There is no guarantee that the entities will stay together, albeit that 'staying together' is not well defined. Even though the entities all have a similar goal, they try to reach this goal without real coherence. This results in groups splitting up and taking different paths to the goal, for example as in Fig. 2.

We will describe a technique in which groups are guaranteed to stay together. More details can be found in [14]. We are given a game world in which a group of entities must move from a given start to a given goal position. The entities must avoid collisions with the environment and with each other, and should stay together as one group. Just like when planning for a single entity, the entities in the group are modeled/approximated as discs (or cylinders) and are assumed to move on a plane or terrain. Later, the resulting paths for the cylinders can be used to animate avatars, e.g. sprites or motion captured human-like avatars.

The method consists of three on-line (in game) phases. We assume that off-line the high quality roadmap is generated and can be accessed during the game. These phases are:

(1) A so-called *backbone path* for a single entity is computed, i.e. a query is performed on the high quality roadmap to find a path from the start position of the group to a goal position of the group. This path defines the homotopy class used by all entities. Two paths P_0 and P_1 are said to be homotopic if P_0 can be continuously deformed into P_1 without intersecting any obstacle.
(2) From this backbone path a *corridor* is constructed in which all entities must stay. This corridor consists of a set of circles around the backbone path such that no obstacles are inside these circles.
(3) The movement of the entities is generated using force fields with attraction points on the backbone path. By limiting the distance between the attraction points for the different entities, coherence of the group is guaranteed.

These phases are outlined in more detail in the following subsections.

3.1.1. *Finding the backbone path*

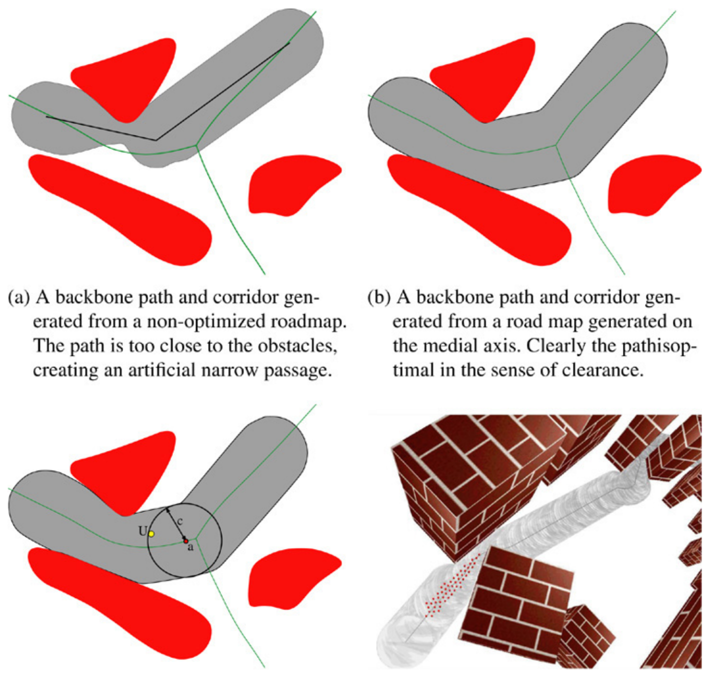
The first phase of the approach consists of finding the backbone path. Since every entity should be able to traverse the path, the clearance on the path should be bounded by a some minimum value, namely the radius of the enclosing circle/cylinder of the largest entity. The backbone path can thus be defined as follows: A backbone path is a path in the 2D or 2.5D workspace (or subspace of the 3D world), where the clearance at every point on the path is at least the radius of the enclosing circle/cylinder.

Although a minimum clearance of the radius of the enclosing circle is sufficient to find a path, we prefer a larger clearance, since a larger clearance leads to behavior that is more coherent. Also we prefer the paths to be smooth and short. Hence, the paths in the high quality roadmap created with the method described above are very well suited for this application. Since the path is already smooth, no real-time smoothing is required. This makes finding the backbone path very fast and well suited to be used in games.

3.1.2. *Construction of the corridor*

From the backbone path, a corridor is created. As described above, a corridor is a collection or set of circles/cylinders around the backbone path such that no obstacles are inside these circles/cylinders. To be able to construct the corridor we need to know the clearance around the path. On every point on the path, the clearance is defined as the radius of the largest circle around this point that does not intersect with the environment. In large open spaces this might lead to very wide corridors. This is often unwanted. Therefore, we upper bound the clearance by the maximum group width, i.e. the clearance can never exceed the maximum group width. The union of all the upper-bounded clearance circles forms a corridor around the backbone path.

Fig. 9(a) shows a backbone path that is too close to the obstacles, resulting in artificial narrow passages in the corridor. In contrast, the backbone path in Fig. 9(b) was generated with the high quality roadmap approach described above. The backbone path lies in the middle between the obstacles, i.e. it lies far from obstacles. The resulting corridor is much more natural without any artificial narrow passages. The group will be able to traverse this corridor much easier than the previous corridor.



(a) A backbone path and corridor generated from a non-optimized roadmap. The path is too close to the obstacles, creating an artificial narrow passage.

(b) A backbone path and corridor generated from a road map generated on the medial axis. Clearly the pathisoptimal in the sense of clearance.

(c) The attraction point a of an entity U when point a has a clearance c.

(d) A real corridor.

Fig. 9. The corridor.

3.1.3. Generating the motion inside the corridor

Once the corridor is created, we need to use it to generate the motion of the individual entities in the group. The approach used is an artificial force field technique. Forces are defined that act on the entities and influence their movement using basic physics laws, like Newtons laws of motion, i.e. for every entity i in the group holds:

$$\vec{F_i} = m_i \vec{a_i}; \qquad \frac{\partial \vec{v_i}}{\partial t} = \vec{a_i}; \qquad \frac{\partial \vec{x_i}}{\partial t} = \vec{v_i}$$

where $\vec{F_i}$ is the total force exerted on the entity, m_i its mass, $\vec{a_i}$ its acceleration, $\vec{v_i}$ its velocity, and $\vec{x_i}$ its position. For the sake of simplicity we will assume in this paper that the mass of an entity is equal to 1.

So given the force on an entity we can calculate the path the entity is going to follow by numerically integrating the accelerations and velocities. This can be done using any known time integration scheme, like Euler (Forward), RungeKutta or Verlet integration. Due to the construction of the forces (see below) it is recommended to use a integration scheme that is quite stable, such as RungeKutta4.

There are several forces acting on the entities. First, the entities should not collide with each other. This means that they need to repulse each other, i.e. a force is needed in the direction from one entity to another. This force should repulsive very little when entities are far apart, but repulse strongly when close together. For this the following force on entity i is defined:

$$\vec{F_{i,rep}} = \sum_{j,i \neq j} \frac{c_{rep}(\vec{x_i} - \vec{x_j})}{\|\vec{x_i} - \vec{x_j}\|^2}$$

where c_{rep} is an arbitrary constant specifying the relative strength of the force, $\vec{x_i}$ and $\vec{x_j}$ are the positions of entities i and j, respectively.

The second force needed for every entity is the so-called attraction force to the backbone path. This force will ensure that the entities will move forward along the backbone path and stay inside the corridors. For this, we need to have an attraction point on the backbone path for every entity in the group. This attraction point is chosen as the maximum advanced point p along the backbone such that the entity is still completely inside the circle centered at that point p with radius equal to the clearance at p (see Fig. 9(c)). This means that the center of the entity will always maintain a distance from the edge of the circle, namely the radius of the entity. By applying a force on an entity pointing toward this attraction point makes the entity move forward and stay inside the corridor (the force is 0 in the center of the circle and approaches infinity toward the edge of the circle). The attraction point for an entity i is denoted by $\vec{att_i}$. The clearance at the attraction points is denoted by $c(\vec{att_i})$. Please refer to Fig. 10 for a graphical representation of these quantities. Then, the force can be defined as:

$$\vec{F_{i,att}} = \frac{c_{att}(\vec{att_i} - \vec{x_i})}{c(\vec{att_i}) - \|\vec{att_i} - \vec{x_i}\|}$$

where c_{att} is an arbitrary constant specifying the relative strength of the force.

3.1.4. *Keeping coherence in the group*

In order to keep the group coherent, the dispersion should be upper bounded. Due to the manner in which the corridor is constructed, the lateral dispersion (dispersion perpendicular to the backbone path) is automatically upper bounded by the group width. However, the longitudinal dispersion (in the direction of the backbone path) is not yet bounded in this approach. To correct this, the distance along the path from the least advanced attraction point to the most advanced attraction point should be limited. Fig. 11 depicts how this should be done. Normally, for entity j the attraction point would be $\vec{att'_j}$. However, since the group area is bounded, the actual attraction point should not exceed $\vec{maxatt}$. By not allowing the attraction points to be passed this maximum the entities are pulled back (slowed down) toward the rest of the group as soon as they are too far ahead. This results in the entities in front waiting for the entities at the back.

3.1.5. *Results*

The behavior of the group can be controlled by adjusting the coherence parameters, *lateral dispersion* and *longitudinal dispersion*. Fig. 12(a)–(c) show a group of 50 entities moving through an environment. In these pictures the lateral and longitudinal dispersion is varied, resulting in a longer, more stretched group 12(a) or more compact group 12(c).

Fig. 12(d) shows the same group moving through the environment from the left lower corner to the right upper corner. The most advanced entities, i.e. the entities that passed the narrow passage earliest, wait for the last entity to pass the passage.

To test the usability in real-time games, we constructed a demo application written in C++. In this demo application, the units are allowed to move at a maximum speed of 15 per second (can off course be any arbitrary value). Remember that the measure of distance is the unit radius. From experiments with this demo application we can conclude that, in order to produce paths reliably (that is without collisions), we are allowed to move 3 per iteration. Although the physics engine of a typical computer game runs at 50–60 Hz, the positions and velocities of the units in our demo application need to be updated at 5 iterations per second. Every iteration (pre-)calculates about 0.2 s of movement. The resulting processor usages for different group sizes are given in the following table:

group size	**processor usage**
20	0.5%
40	0.8%
60	1.2%
80	1.8%
100	2.4%

For example, it takes 0.02 s to calculate a second of movement for a group of 100 units, i.e. only 2% of processor time.

3.2. *Planning camera motions*

Every virtual world has a camera through which the user views the world. Usually this camera moves depending on user input and place of action. A camera motion directs the camera from one position to another while controlling camera speed and view direction. There are numerous situations in which an automatic camera motion could be

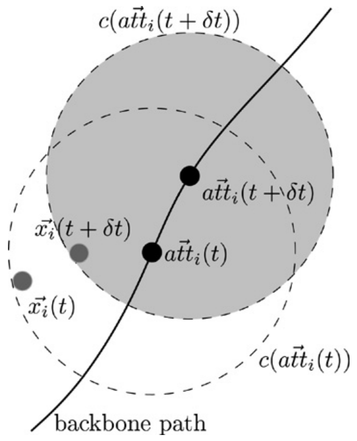


Fig. 10. The attraction point ($\vec{att}_i$) for an entity i is defined as the furthest advanced point on the backbone path that still contains the entity. This figure shows to attraction points for two subsequent positions of the entity, namely at times t and $t + \delta t$.

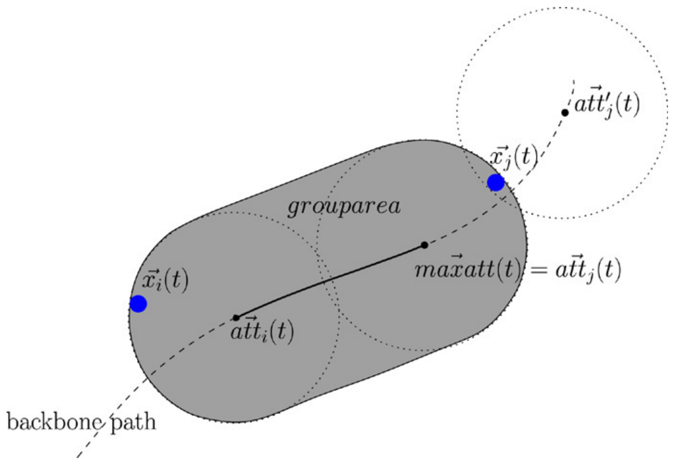


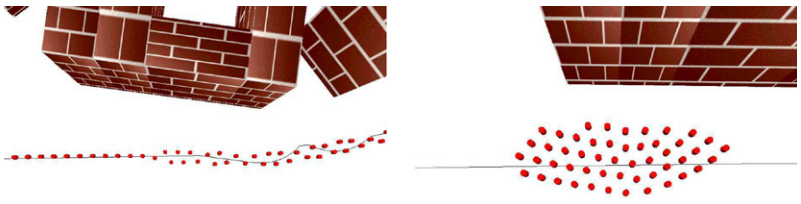
Fig. 11. The group area upper bounds the possible attraction points for the entities.

of great use. Think for example about point-and-click adventure games or third person games in which an entity is followed by the camera. Also situations in which an environment is presented to a user (for example to point out interesting spots) need automatic camera motions. Automatic camera control provides for much more high-level interaction with a user. If a virtual world is presented to a user, then, by allowing the user to click on interesting locations, the user can enjoy the virtual worlds without the problems of getting lost or queasy.

The roadmaps from the previous section are very suited to steer a camera. Using these roadmaps, the camera is guaranteed to keep a certain amount of clearance from the obstacles and the circular arcs make sure that the camera motion is gentle. The roadmap alone however is not enough to create a smooth camera motion. Camera theory [25, 26] shows that we need to take care of two more variables. First, the speed of the camera should be adapted according to the curvature of the path. Otherwise, objects will move too fast through the view, similar to the effect of a fast-forwarding movie. Second, the user should get cues about where the camera is going. In particular the viewer should be able to anticipate a camera rotation. We will resolve these issues in the next two sections.

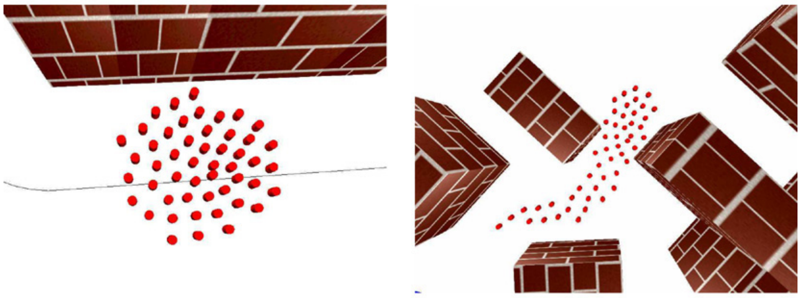
3.2.1. Adapting the camera speed

Smoothness of the camera path alone is not enough for a smooth camera motion. The speed of the camera along the path should be adapted according to the curvature of the path. Also there should be a maximum acceleration and deceleration for the camera in order to prevent too abrupt speed changes. Controlling the combination of these variables results in natural looking camera motions.



(a) The group with high longitudinal dispersion and very low lateral dispersion.

(b) The group with medium longitudinal dispersion and medium lateral dispersion.



(c) The group with low longitudinal dispersion and larger lateral dispersion.

(d) The group moves through a narrow passage, making the first wait for the last ones (movement is from the lower-left to the upper-right).

Fig. 12. A group of 50 entities moving in a virtual world. These paths and behaviors are created with the same approach, only by varying the parameters.

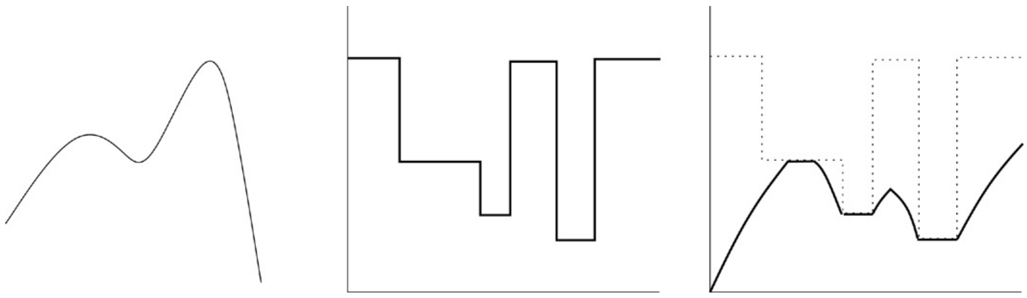


Fig. 13. A speed diagram. The left image shows the path, the middle image shows the maximal speed allowed at each position. The right image shows the actual speed, taking acceleration and deceleration bounds into account.

Since our path consists of straight lines and circle arcs, we can adapt the speed of our camera by making use of the radius of the arcs. The smaller the radius, the lower the camera speed. When the camera leaves an arc with a small radius (and thus has a low speed), we accelerate until we have reached the maximum speed of the current arc or straight line. If, on the other hand, the next arc requires a lower speed than the current camera speed, we must start decelerating on time, such that when we reach the next arc, our speed is sufficiently low. A speed diagram can be computed efficiently that satisfies both the constraints on the maximal speed for each arc and the bounds on acceleration and deceleration. See Fig. 13 for an example of such a speed diagram for a simple path.

3.2.2. *Smoothing the viewing direction*

Intuitively one might think that the viewing direction should be equal to the direction of the camera motion. As stated before however, camera theory states that the user should be given cues about where the camera is heading or

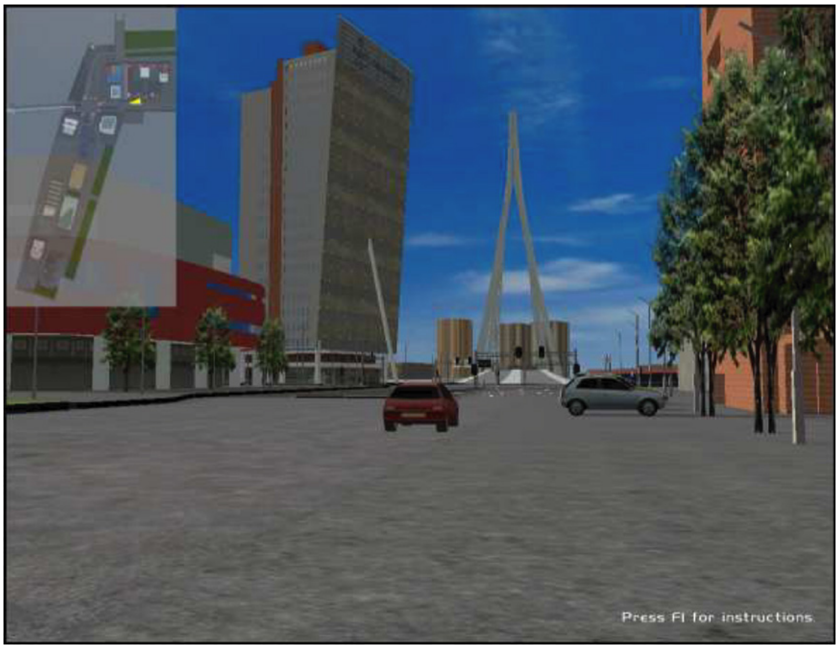


Fig. 14. An implementation of the techniques. The user can click on a location in the map at the left top and the program creates a smooth camera motion to that location.

else the user may experience discomfort. We can achieve this by always looking at the position the camera will be in a short time. Experiments show that about 1 s is the right amount. Note that, as we fix the time we look ahead, the distance we look ahead changes depending on the speed of the camera. This is exactly what we want to achieve as in sharp turns we want to look at a nearer point than in wide turns. Looking ahead has another important effect. If we would look in the direction of motion and the camera reaches a circular arc, then it suddenly starts rotating at the start of the arc. Stated more formally, the rotation of the camera is only C^0 continuous. It can be proved that looking ahead solves this issue by making the camera rotation C^1 continuous.

3.2.3. Results

We implemented our approach in a walk-through system for virtual worlds called Quest3D. This is a DirectX-based 3D development package. The navigation techniques were added as a module using Visual C++. In Fig. 14 a screenshot of this application is shown. Rather than letting the user steer the camera directly, we display a map in the top left corner. By clicking on the map the user indicates the position he wants to move to. An automatic smooth camera motion is then calculated such that the user can focus on the environment instead of the camera control. Again, because the vast amount of calculation is done in the pre-processing phase, the processor time required for camera control during the running of the application is minimal. Experiments indicate that paths generated by the methods described in this section are pleasant to watch.

4. Conclusions

In this paper we have described a novel technique for automatic construction of high quality roadmaps in games that can be used for computer controlled entities to quickly navigate. We have also shown how this technique can be used to plan the motion for groups of entities and to navigate a camera. Due to the nature of our roadmap, the approach also works in large environments, where grid-based method typically fail (requiring too much storage and processing time).

We presented our method as a two-dimensional approach in which entities move on a ground plane. It is easy to extend it to e.g. terrains and even motions in buildings in which the roadmap would automatically follow the corridors and stairs.

Roadmap construction is best seen as being part of the construction of the game world. It is easy to incorporate special requirements from the level designer. For example, the designer can add fake obstacles to force the path to e.g. stay on the sidewalks of a road. Also the designer can easily manipulate the roadmap graph by manually adding, changing, or removing nodes. Moreover, weights can be added to the graph to e.g. indicate preferred routes.

Since the creation of the roadmap on which the navigation is based is done completely automatically, our methods are also suited for applications in which the game world is not completely known at design time.

Acknowledgements

The authors would like to thank Marlies Mooijekind for her contribution. This research was supported by the Dutch Organization for Scientific Research (N.W.O.). This research was also supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2001-39250 (MOVIE - Motion Planning in Virtual Environments). Part of this research has been funded by the Dutch BSIK/BRICKS project.

References

[1] M. DeLoura (Ed.), Game Programming Gems 1, Charles River Media, 2000.
[2] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 1994.
[3] F. Lamiraux, O.L.D. Bonnafous, Reactive path deformation for nonholonomic mobile robots, in: IEEE Transactions on Robotics, 2004 (in press).
[4] W. Stout, Smart moves: Intelligent path-finding, Game Developer.
[5] S. Baert, Motion planning using potential fields, gamedev.net. URL: http://www.gamedev.net.
[6] M. Pinter, Toward more realistic pathfinding, gamasutra.com. URL: http://www.gamasutra.com.
[7] C. Reynolds, Flocks, herds, and schools: A distributed behavioral model, Computer Graphics 21 (4) (1987) 25–34.
[8] O. Bayazit, J.-M. Lien, N. Amato, Better flocking behaviors using rule-based roadmaps, in: Algorithmic Foundations of Robotics V, Springer Tracts in Advanced Robotics 7, Springer-Verlag, Berlin, Heidelberg, 2004, pp. 95–111.
[9] L. Kavraki, P. Švestka, J.-C. Latombe, M. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Transactions on Robotics and Automation 12 (1996) 556–580.
[10] P. Švestka, M. Overmars, Coordinated path planning for multiple robots, Robotics and Autonomous Systems 23 (1998) 125–152.
[11] L. Kavraki, J.-C. Latombe, Randomized preprocessing of configuration space for fast path planning, in: Proc. IEEE Int. Conf. on Robotics and Automation, IEEE Press, San Diego, CA, 1994, pp. 2138–2139.
[12] C. Holleman, L. Kavraki, A framework for using the workspace medial axis in prm planners, in: Proc. IEEE Int. Conf. on Robotics and Automation, vol. 2, 2000, pp. 1408–1413.
[13] C. Reynolds, Steering behaviors for autonomous characters, in: Game Developers Conference, 1999.
[14] A. Kamphuis, M.H. Overmars, Finding paths for coherent groups using clearance, in: Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2004 (in press).
[15] D. Nieuwenhuisen, M. Overmars, Motion planning for camera movements, in: Proc. IEEE Int. Conf. on Robotics and Automation, IEEE Press, San Diego, CA, 2004, pp. 3870–3876.
[16] R. Bohlin, L. Kavraki, Path planning using lazy prm, in: Proc. IEEE Int. Conf. on Robotics and Automation, 2000, pp. 521–528.
[17] V. Boor, M. Overmars, A. vander Stappen, The gaussian sampling strategy for probabilistic roadmap planners, in: Proc. IEEE Int. Conf. on Robotics and Automation, 1999, pp. 1018–1023.
[18] C. Nissoux, T. Siméon, J.-P. Laumond, Visibility based probabilistic roadmaps, in: Proc. IEEE Int. Conf. on Intelligent Robots and Systems, 1999, pp. 1316–1321.
[19] S. Wilmarth, N. Amato, P. Stiller, Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space, in: Proc. IEEE Int. Conf. on Robotics and Automation, 1999, pp. 1024–1031.
[20] D. Hsu, T. Jiang, J. Reif, Z. Sun, The bridge test for sampling narrow passages with probabilistic roadmap planners, in: Proc. IEEE Int. Conf. on Robotics and Automation, 2003.
[21] M. Branicky, S. Lavalle, K. Olson, L. Yang, Quasi randomized path planning, in: Proc. IEEE Int. Conf. on Robotics and Automation, 2001.
[22] P. Isto, Constructing probabilistic roadmaps with powerful local planning and path optimization, in: IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2002, pp. 2323–2328.
[23] D. Nieuwenhuisen, M. Overmars, Useful cycles in probabilistic roadmap graphs, in: Proc. IEEE Int. Conf. on Robotics and Automation, IEEE Press, San Diego, CA, 2004, pp. 446–452.
[24] R. Geraerts, M. Overmars, A comparative study of probabilistic roadmap planners, in: Algorithmic Foundations of Robotics V, Springer Tracts in Advanced Robotics 7, Springer-Verlag, Berlin, Heidelberg, 2004, pp. 43–57.
[25] G. Millerson, TV Camera Operation, Focal Press, London, 1973.
[26] M. Wayne, Theorising Video Practice, Lawrence and Wishart, London, 1997.
[27] J.P. van den Berg, D. Nieuwenhuisen, L. Jaillet, M.H. Overmars, Creating robust roadmaps in changing environments, Tech. Rep. UU-CS-2004-069, Institute of Information and Computing Sciences, Utrecht University, The Netherlands, 2004.
[28] J.P. van den Berg, M.H. Overmars, Roadmap-based motion planning in dynamic environments, in: Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2004, pp. 1598–1605.
[29] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (1975) 509–517.