Motion Planning for Camera Movements in Virtual Environments[*]

Dennis Nieuwenhuisen   Mark H. Overmars
Institute of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. Email: `[dennis,markov]@cs.uu.nl`.

January 30, 2003

Abstract

When users navigate through a virtual environment they often directly control the camera. Such direct control is difficult for inexperienced users and results in rather ugly camera motions that easily lead to motion sickness. In this paper we describe a new technique for automatic generation of camera motion using motion planning techniques from robotics. In this approach the user simply specifies a required goal position (and orientation) using e.g. a map, and the system automatically computes a smooth camera motion from the current position and orientation to the required position (and orientation). As preprocessing the approach uses the probabilistic roadmap method to compute a roadmap through the environment. When a camera motion is required a path is obtained from the roadmap which is then improved by various smoothing techniques to satisfy camera constraints. The method has successfully been integrated in a system for walkthroughs in architectural designs and urban planning.

1 Introduction

In many virtual environment applications, like games, architectural walkthroughs, urban planning systems, CAD model inspection systems, and training systems, the user must navigate through the environment to inspect it and perform certain tasks. Navigation means: steering a virtual camera through the environment. In most of today's systems this is done using a joystick or arrow keys on the keyboard in combination with a mouse. Such direct control has a number of disadvantages. It is difficult for inexperiences users, it results in rather ugly motions that easily lead to motion sickness, and it requires a lot of attention while the user should preferably concentrate on more higher-level tasks at hand.

In this paper we describe a new technique for navigation through a virtual environment. In this approach the user simply specifies a required goal position (and orientation) using e.g. a map, and the system automatically computes a smooth motion from the current position and orientation to the required position (and orientation), avoiding collisions with obstacles in the scene. This motion result in a smooth path for the camera that is much more pleasant to watch and enables the user to concentrate on other tasks.

[*]This work has been (partially) 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).

We base ourselves on motion planning techniques from robotics, in particular the probabilistic roadmap method (PRM)[2, 5, 15, 16, 24, 26] that has been successfully applied in many motion planning problems, ranging from robot arms to the folding of proteins. This technique is combined with theory for cinematography to obtain smooth camera motions.

Globally speaking the method works as follows: As preprocessing, using the PRM approach, a roadmap through the environment is computed. This roadmap forms a crude representation of possible motions. When a particular camera motion is required, a path is obtained from the roadmap which is then improved by various smoothing techniques to satisfy camera constraints. A continuous speed function is computed to guide the motion along the path. The viewing direction and up vector are taken into account to achieve the desired result.

The method is fast (after little preprocessing camera motions are computed instantaneously) and experiments show that the resulting motions are indeed pleasant to watch. The method has successfully been integrated in a system for walkthroughs in architectural designs and urban planning.

Previous Work

In the past decade a lot of research has been done on supporting camera motion in virtual environments. In this section we briefly review some of the work. We can distinguish three different types of results.

A number of authors have studies techniques to support the motion by the user. For example, the user is kept on a constraint subspace[12], the “guide manifold”, which can be seen as a virtual side walk, or the speed of motion is automatically adapted to rapidly cover distance[21]. These methods do not assist the user in the actual navigation to a particular goal.

Another collection of papers studies the computation of effective fixed camera positions to assist the user in performing certain (manipulation) tasks, the so-called “shot systems”[4, 8, 11]. To achieve this a collection of constraints is often derived which is then solved. These systems do not plan obstacle-avoiding motions.

Most previous work on planning camera motions is directed toward systems in which the camera must follow an object (like in third-person games). A distinction can be made here in systems where the motion of the object is known beforehand[20] and systems where this motion is not known[3, 7, 10]. Similar problems have been studied in robotics where a robot with a camera must track certain targets[9, 19].

Our problem setting is rather different from the ones studied before. It requires motion planning rather than reactive behavior. Also, in a first-person view smoothness of the motion is much more critical than in third-person views (where the camera motion does not even have to be continuous).

Paper Structure

In the next section we will more precisely define the notion of camera motion, the requirements, and the problem we want to solve. In Section 3 we describe the preprocessing we do on the scene to create a roadmap of possible motions. In Section 4 we describe how we use the roadmap to compute an initial path for the camera. This path though is not smooth. In Section 5 we propose a number of techniques to improve the quality of the path such that it can be used for the camera. Besides the path of the camera also the viewing direction and the up vector play an important role. In Section 6 we show how these can be incorporated in the motion. Finally in Section 7 we describe the CAVE system that we built which incorporates the automated camera motion approach in an application for architectural walkthroughs.

2 Camera Movement

In this section we will define what we mean with a good camera motion. A camera can be described with a position, an orientation, and a zoom-factor. (There are some other parameters, like the type of projection used, shear, etc. but for natural camera motions we obviously want a perspective projection and no shear.) Also, because we use the camera in a first-person view, we will not allow zooming in and out. The configuration (position and orientation) of a camera can be described in a few different ways. Throughout this paper we will use three parameters to describe the position of the camera, another three to describe the point the camera looks at (which represents the viewing direction) and one parameter that describes the rotation of the camera around the axis spanned by the two positions, also called the twist or roll. This makes a total of seven parameters to describe a camera configuration, although one is redundant (the viewing direction can also be specified by two parameters).

The camera can move in four different ways (often combined), as depicted in Figure 1: the camera can translate, that is, move to a new position, it can rotate horizontally, it can rotate vertically, and it can roll around its main axis.

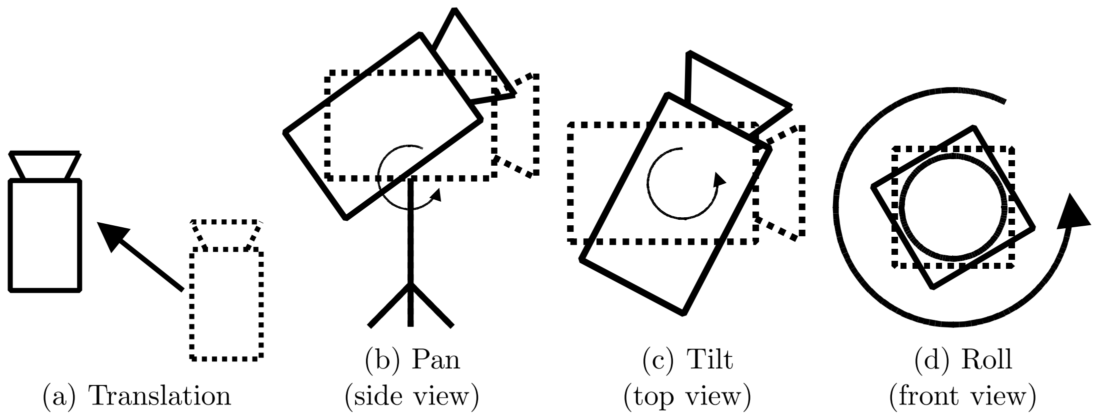

Figure 1: Possible camera motions.

Using literature from cinematography[22, 30] some general guidelines on the best motions can be distilled.

- The camera should preferable not pass too close to obstacles. Otherwise too big a part of the screen will be filled with the obstacle and this would give the viewer a feeling of (almost) a collision. So we need some clearance for the camera.
- The horizon of the camera should be kept as straight as possible to prevent a “drunk” feeling by the viewer. This means that we should preferably not roll the camera.
- When the camera makes a sharp turn, its speed should be lowered. Otherwise objects will move too fast through the view.

- To prevent long dull shots, the speed of the camera should always be as high as possible, bounded by some speed that depends on the application (e.g., are we walking or flying through an environment).
- The viewer should get cues about where the camera is going. In particular, the viewer should be able to anticipate a rotation. As we will see, this can be achieved by letting the camera not look in the direction of motion but into the turns, that is, toward the position were it will be some short time in the future.

The goal in this paper is to compute camera motions that satisfy these guidelines. More precisely, we can formulate the problem as follows: Given an environment consisting of a set of obstacles O, a start configuration s, and a goal configuration g, compute a short, natural looking camera motion from s to g avoiding collisions with all $o \in O$.

3 Creating a Roadmap

To enable immediate response to camera motion requests we need to preprocess the scene. The result of this preprocessing will be a roadmap consisting of collision-free camera motions. Only the position of the camera is stored in the roadmap. The viewing direction will be computed when the actual camera path is required.

It is way too time and storage consuming to compute all possible camera motions. Hence, we will only compute a coarse roadmap. This roadmap will consist of straight-line camera motions. When the actual camera path is required, the roadmap will be locally extended and a smooth path will be computed, using the roadmap as a guideline.

To compute the roadmap we will use the so-called *Probabilistic Roadmap Method (PRM)* that has been developed in robotics [2, 5, 15, 16, 24, 26]. It constructs a roadmap of possible motions in a probabilistic way. Free camera positions (or robot configurations in general) are created randomly and form the nodes of the roadmap graph. For nearby nodes we check whether the straight-line connection between them is collision-free and, if so, add a corresponding edge to the graph. It can be shown that, once the graph gets dense enough, all nodes will get connected. See Figure 2 for an example of a roadmap for a simple scene of a house.

A big advantage of PRM is that its complexity tends to be dependent on the difficulty of the path, and much less on the global complexity of the scene. In the past few years the method has been successfully applied in many motion planning problems dealing with robot arms[17], car-like robots[27, 29], multiple robots[28], manipulation tasks[25] and even flexible objects[13, 18].

Let us describe the application of PRM to planning camera motions in more detail. To guarantee that the camera keeps a minimal clearance to the objects in the scene, we don't consider the camera as a point but as a sphere $S_\delta(c)$ of radius δ around camera position c. Here δ is the minimal clearance required. The roadmap will take the form of a graph $G = (V, E)$. The nodes in V correspond to collision-free positions for the sphere $S_\delta(c)$ while each edge $cc' \in E$ corresponds to a collision-free motion between positions c and c'. This can be tested by considering the cylinder $C_\delta(c, c')$ between positions c and c' of radius δ. If this cylinder does not intersect an obstacle we have a collision-free motion.

The algorithm picks random camera positions, adds them to the roadmap when they are collision-free and next tries to find collision-free edges. The construction algorithm looks as

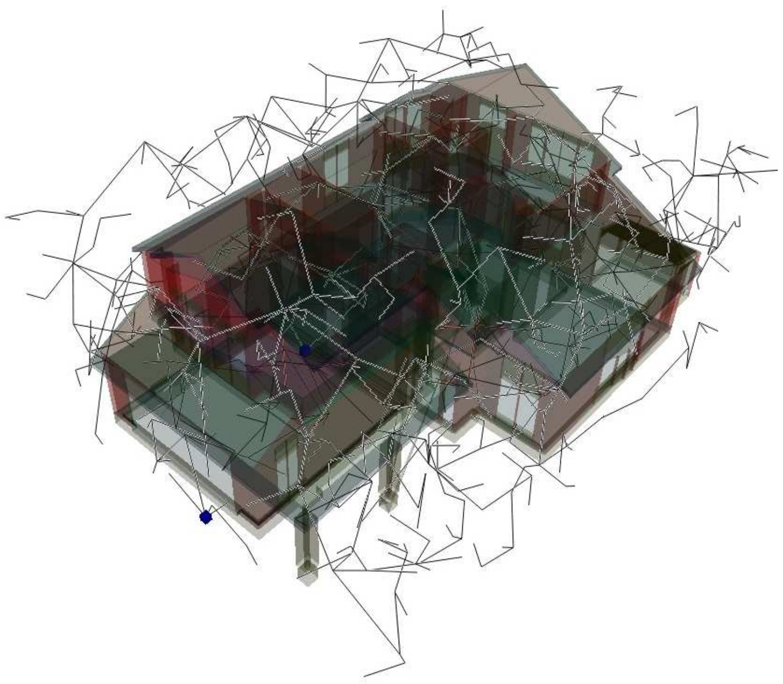

Figure 2: An example of a roadmap for a house.

follows:

Algorithm 1 ConstructRoadmap

```
Let: $V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
 1: loop
 2:   $c \leftarrow$ a (random) position
 3:   if $S_\delta(c)$ is collision-free then
 4:     $V \leftarrow V \cup \{c\}$
 5:     $N_c \leftarrow$ a set of neighbor nodes chosen from $V$
 6:     for all $c' \in N_c$ do
 7:       if $C_\delta(c, c')$ is collision-free then
 8:         add the edge $cc'$ to $E$
```

There are two important choices to be made here. The first is how to select the positions. If we allow arbitrary camera motion through the scene we can pick random positions. If we require the camera to stay on a fixed height we pick random configurations on that height. If particular positions are important or should preferably be avoided we can also take that into account here.

The second choice is the set N_c of neighbor nodes for which we try connections. If we use a large set the algorithm will run slow because of the many collision checks, but the roadmap will become very dense, leading to good motions. If we use a smaller set the algorithm will run faster and the number of roadmap edges will be smaller, leading to potentially longer camera paths. In the robotics community much research has been done in the correct choice of N_c and on other techniques to improve the quality of the roadmap[1, 6, 14, 23, 31]. For the application in camera motions a simple approach choosing a fixed number of relatively nearby nodes in different directions around c suffices.

Once enough nodes are added (one way to test this is to check whether the number of connected components in the roadmap graph stops changing) the roadmap can be used to solve motion queries. This will be described next.

4 Finding a Path

The algorithm described in Section 3 results in a graph that covers the free space of the environment. This roadmap is computed in a preprocessing phase (or stored with the environment). It is rather small (a few hundred nodes in the scenes we used in our experiments). We will use this roadmap to compute a path when the user wants to move.

As indicated above, a camera configuration consists of a position and an orientation. We will first only plan the motion for the position of the camera. Later we will incorporate the orientation. Our first step is to compute an initial path. To this end we need to add the start and goal position of the camera to the roadmap and compute a good path in the graph, taking into account the guidelines from Section 2.

4.1 Adding the Start and Goal

Adding the start position s and goal position g to the roadmap is done in exactly the same way as the random positions were added. We create two nodes for them and try to connect these nodes to a set of N_s and N_g of neighbor nodes. If such connections are successful and s and g end up in the same connected component of the graph, then a path exists.

If s or g cannot be connected to the graph we can expand the graph by adding a number of additional random positions in the vicinity of s and g. This is in particular important if s or g lies in a very confined region in the environment. This local expansion can be done very fast (normally just a few nodes suffice).

If s and g can be connected to the graph but are not in the same connected component, there either does not exist a path, which can be reported, or the initial roadmap was not dense enough. We can either spend some more time on the roadmap construction or use techniques to expand the roadmap at critical places[17]. Based on properties of the PRM technique the following theorem can be proven:

Theorem 4.1 *Given enough preprocessing time the method will always find a path between s and g if such a path exists.*

Stated differently, the method is probabilistically complete[26].

4.2 Computing the Shortest Path

In order to find the shortest path between s and g in the roadmap graph, Dijkstra's shortest path algorithm can be applied. This would, however, lead to a path that is short in length, but does not take any of the guidelines for a natural camera motion into account. It will also not necessarily lead to the fastest path because the camera has to slow down when approaching a sharp turn and after the turn it has to accelerate again. Wider turns are preferred over short sharp turns as long as the path length does not increase too much. It can be beneficial to make a small detour, thus avoiding sharp turns. The overall camera speed will be higher, because the camera has to slow down less, see Figure 3.

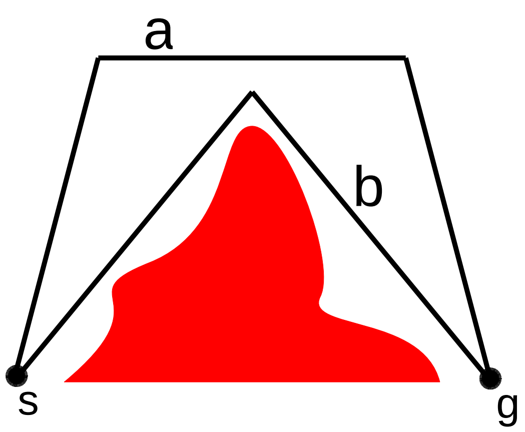

Figure 3: Path b is shorter than path a but a is preferred because the overall speed is higher.

To find a balance between a good and a short path, we use a penalty function. As we will see in Section 5.1 we will replace turns in the path by circle arcs. As the speed with which such an arc can be traversed depends on the radius we need to take the angle between successive edges of the path into account when computing the cost of a path. The penalty value for an edge e when arriving from edge e' is $p(e, e')$. The length of edge e, $l(e)$ also needs to be taken into account. This results in a total distance measure $d(e)$ for e arriving from e' of $p(e, e') + l(e)$.

To compute the shortest path according to this distance measure we need to use an edge-based version of Dijkstra's shortest path algorithm rather than the usual node-based version. For each edge in the graph we compute the shortest path from the start position, until we reach the best edge leading to the goal position. This is an easy modification. Let $\deg(c)$ denote the degree of node c in the roadmap and let $\mathrm{DEG} = \sum_{c \in V} \deg(c)$. Then the algorithm will take time $O(\mathrm{DEG} \log \mathrm{DEG})$. Fortunately, we connect each node c with at most $|N_c|$ other nodes, the size of the neighbor set for c and we pick only a constant number of neighbors. So the total number of edges is linear in the size of V. We can also avoid that too many edges connect to the same node. As a result $\mathrm{DEG} = O(|V|)$. This leads to the following theorem:

Theorem 4.2 *A short camera path in the roadmap graph $G = (V, E)$ between s and g can be computed in time $O(|V| \log |V|)$.*

One can improve the running time in practice by using a modification of the A* algorithm. But as computing the shortest path is not a major cost factor in the algorithm we do not work out the details.

5 Improving the Path

We have now found a path between the start and goal configurations that avoids obstacles. However, the path consists of straight line segments and, hence, is only C^0 continuous. For a smooth camera motion the path must be at least C^1 continuous. In this section we will describe how to make the path C^1 continuous using circular blends and how to compute a continuous speed function for traversing the path.

5.1 Adding Circular Blends

The camera path produced so far consists of edges that correspond to straight-line motions that meet at nodes. These nodes introduce first-order discontinuities in the motion. To remove these first-order discontinuities we use circular blends between the edges that meet at a node. (We could also use parabolic blends or clothoids to get a higher degree of continuity but these are more difficult to compute and lead to a higher maximum curvature which in turn limits the maximal speed with which the corner can be traversed.)

Let e and e' be two consecutive edges in the final path that meet at a node v. Let p_m be the midpoint of e and let p'_m be the midpoint of e' (see Figure 4). We replace part of the path between p_m and p'_m by a circle arc that lies in the plane spanned by the two edges. This arc will have its center on the bisecting line of e and e', will touch e and e' and have either p_m or p'_m on its boundary, depending on which one lies closer to the node v. Because the arc touches the two edges it will remove the first-order discontinuity at node v. Since the arc will only remove at most half of the edges e and e' we can repeat this for each node on the path. The resulting path will be C^1 continuous.

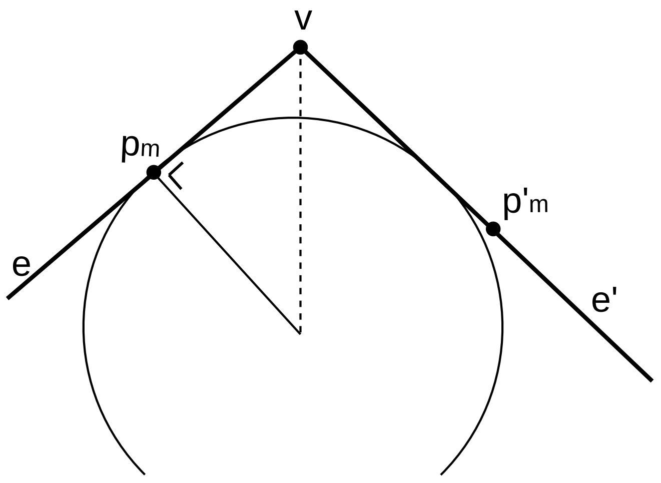

Figure 4: Replacing a node by a circle arc.

There is a problem though. Replacing a part of the edges with an arc might introduce collisions between the camera sphere S_δ and the obstacles when the camera moves along the circle arc. If this is the case we use a smaller arc instead by reducing the distance between v and the touching points on the edges (see Figure 5). It is easy to see that, assuming the initial path does not touch any obstacles, there exists a radius for the circle arc for which the motion along the arc is collision free. We use a binary search on the radius to find the largest possible radius for which the motion is collision free. Note that computing collision checks between the circular blends and the obstacles is the most time-consuming part of the algorithm (after the preprocessing). Most collision checkers do not allow such checks directly.

Hence, for the purpose of collision checking, we approximate the arc by a number of short line segments and check these instead. Because our initial path had a clearence of δ it is easy to make sure that a collision of the arc also results in a collision for one of the segments. So the resulting arc is indeed collision free. See Figure 6 for an example of a path with the circular blends.

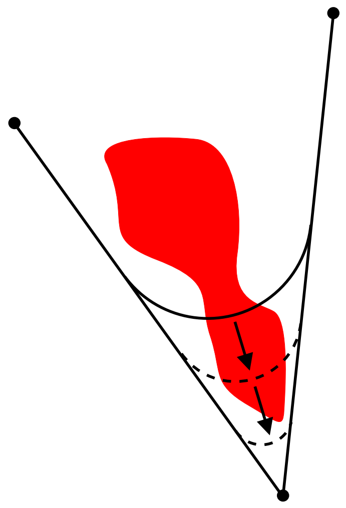
Figure 5: Moving a circle arc to avoid an obstacle.

Theorem 5.1 *Let π be a collision-free path that consists of straight-line motions and that does not touch any of the obstacles. There exists a collision-free path π' consisting of straight-line motions and circle arcs that is first-order continuous.*

5.2 Speed Diagram

Smoothness of the path itself is not enough for a smooth camera motion. Also the speed along the path should change in a continuous way. To this end we need to compute a speed diagram that indicates the speed $s(c)$ at each location c along the path. The maximum speed $s_{\max}(c)$ allowed must depend on the curvature of the path. Preferably we would like to bound the speed with which the viewing direction changes. But, as we will see below, this viewing direction again depends on the speed. So instead we experimentally determined an appropriate function $s_{\max}(r)$ that depends of the radius r of the circular arc. For the straight parts of the path there will also be a maximum speed, depending on the type of camera motion we want. Using this information we can create a maximum speed diagram. Because our path consists of straight line segments and circle arcs, the maximal speed diagram will be a step function (see Figure 7).

To turn this maximum speed function $s_{\max}(c)$ into a continuous speed function $s(c)$ we need to take the maximum allowed acceleration and deceleration into account. We will briefly indicate how this process can be done in time linear in the number of pieces of the path, using a greedy algorithm. $s_{\max}(c)$ consists of a number of steps, each with a constant speed. The i'th step of $s_{\max}(c)$ is denoted with step$_i$. We look at successive steps of $s_{\max}(c)$, starting with

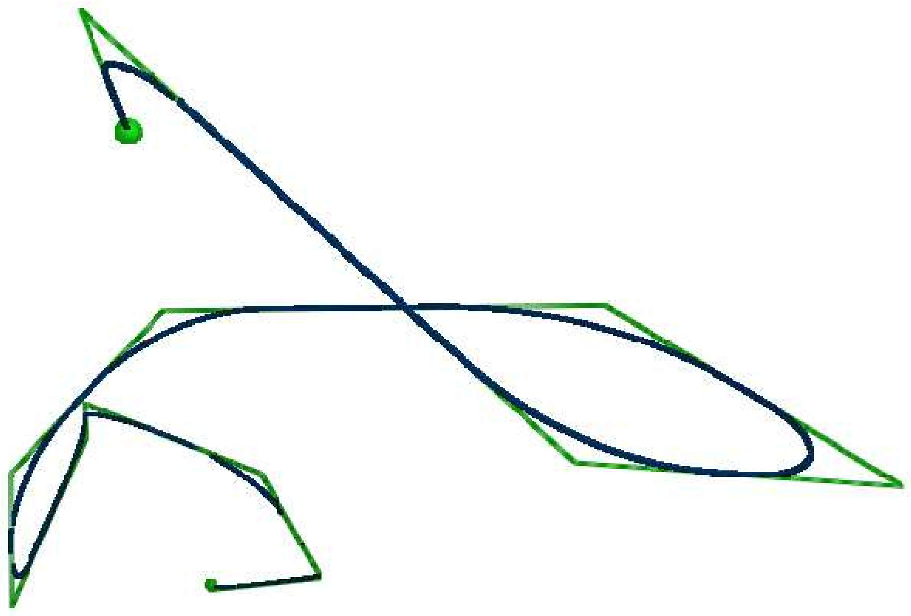

Figure 6: An 3-dimensional example of a path with circular blends. The obstacles were removed for clarity.

the first segment. step_i has a begin position b_i and an end position e_i. We can distinguish the following two cases:

1. If the maximum speed of step_i is higher than the current speed in $s(b_i)$, then we add an acceleration curve that starts at b_i and ends at the position where the maximum speed is reached or at e_i otherwise. This curve is a root function, depending on the maximal acceleration.

2. If the maximum speed of step_i is lower than the current speed in $s(b_i)$, we need to find a position p where to start the deceleration in $s(p)$ that makes sure that the speed reached at b_i is equal to the maximum speed of step_i. For this, we follow a deceleration curve backwards from b_i, removing pieces of $s(c)$ until the curve intersects $s(c)$ at a position p (see Figure 8). We can charge the time this takes to the pieces of $s(c)$ that we remove. Hence the total time required for adding deceleration curves is linear.

Theorem 5.2 *A continuous speed function for the camera path can be computed in time linear in the number of segments and arcs on the path.*

5.3 Shortening the Path

The method described so far does result in smooth camera paths. There is a problem though. Because the roadmap graph is rather coarse the shortest path in the graph might not really be short. The default approach used with the PRM method is to improve the length of the path by making shortcuts. Repeatedly, two random configurations c and c' on the path are

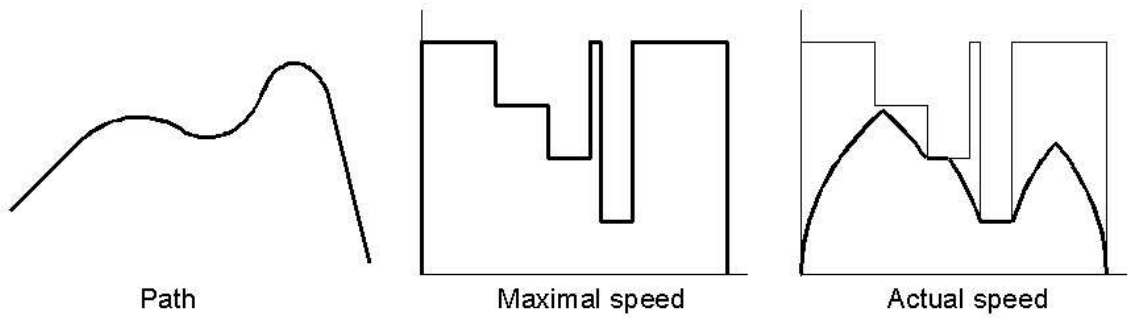

Figure 7: A smoothed path, the maximal speed diagram, and the speed diagram.

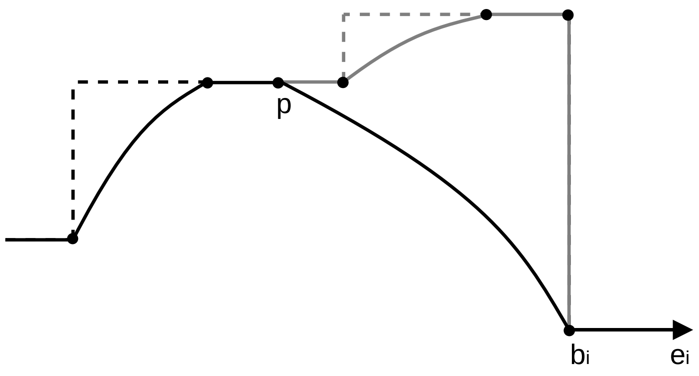

Figure 8: Finding the place to start the deceleration.

chosen and we verify whether the cylinder $C_\delta(c, c')$ between them is collision-free. If so, we replace the path between c and c' by a straight line. We cannot directly apply this approach because a shorter path might actually not be faster. Instead we need to evaluate whether the new path is indeed faster. To this end we compute the total time by adding circle blends and computing a speed diagram as indicated above. (To speed up the processing we do not collision test the circle arcs but experiments show that this is not crucial in this stage. We still obtain a reasonable approximation of the traversal time.) Only when the new total time is lower than the previous total time we replace the piece of the path.

The resulting path is shorter and faster but usually consists of many short line segments. This has as a disadvantage that the speed needs to change often. Paths with less, larger edges and arcs look nicer. So we add another step that removes nodes on the path that lie near to each other, checking whether no sharp turns are introduced. The result of the two improvements is demonstrated in Figure 9. At the left you see the path without shortening. In the middle figure the path is improved by making shortcuts. And in the right figure unnecessary nodes are removed.

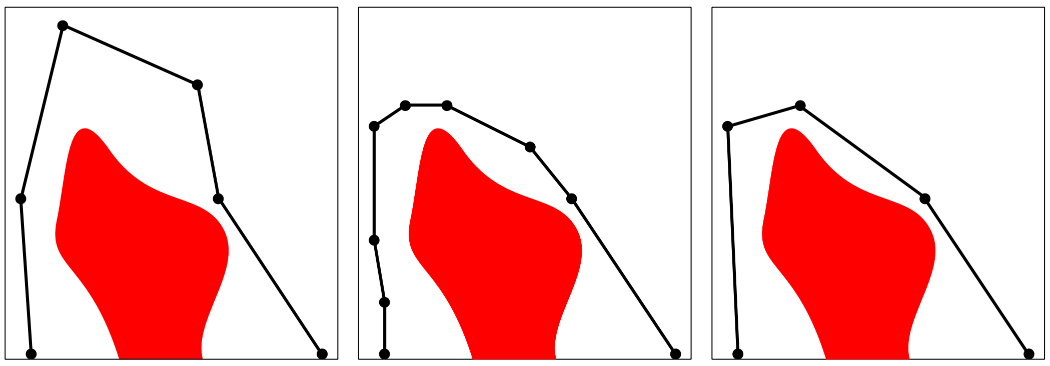

Figure 9: Optimizing the path using shortening.

6 Other Camera Parameters

In the previous sections we concentrated on creating the motion of the camera. But the viewing direction and twist have been ignored so far. In this section we will include these in the motion.

6.1 Smoothing the Viewing Direction

For a smooth camera motion, intuitively one might think that the viewing direction should be equal to the direction of motion. This though is not true. As indicated in Section 2 it is important to give the viewer cues about where the motion is going. This can be achieved by looking at the position on the path where the camera will be in a short time (about 1 second seems a good amount). See Figure 10. Note that, as we fix the time we look ahead, the distance we look ahead changes depending on the speed. This is exactly what we want to achieve as in sharp turns we want to look at a nearer point than in wide turns.

This has a second important effect. If we look along the direction of motion and we reach a circle arc, there is a discontinuity in the speed with which the viewing direction changes. Stated differently the viewing direction is only C^0 continuous. For a smooth motion the viewing direction must be C^1 continuous. As we will show, looking at a position ahead will achieve this.

Let us formalize this approach. Let $W(t)$ denote the position of the camera in the world at time t. At time t we will look at position $W(t + t_d)$ where t_d is the time we look ahead (as stated above a good value for t_d is 1 second). The viewing direction $D(t)$ is along the line from $W(t)$ to $W(t + t_d)$ (see Figure 11). We want to proof that $D(t)$ is C^1 continuous.

The path is represented by a function $P(i) : [0..l] \rightarrow \mathbb{R}^3$, where l is the length of the path. The speed function is represented by $V(i) : [0..l] \rightarrow \mathbb{R}$. (In term of the previous section, $V(i) = s(P(i))$.) We first need a function $U(t)$, which, given a time t, returns a position i. (We assume that the motion starts at time 0.) Then $W(t) = P(U(t))$.

It is easy to see that the time to reach a certain position i on the path is

$$T(i) = \int_0^i \frac{1}{V(x)} dx \tag{1}$$

This function is defined since $V(i)$ is always positive, except for the start and end positions

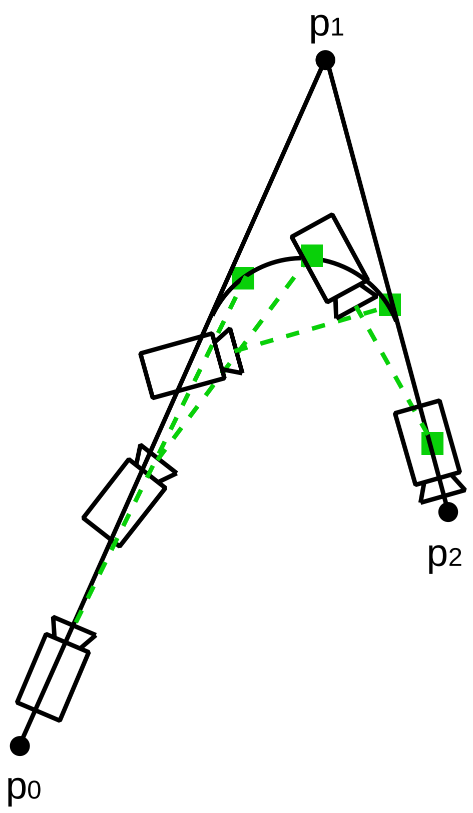

Figure 10: Looking into the turn. The dotted line is the viewing direction. At the end of the turn, the camera has nearly completed its tilt.

which we need to treat separately anyway. The function $U(t)$ we are interested in is the inverse of $T(i)$:

$$U(t) = T^{-1}(t) \tag{2}$$

From the previous section we know that $P(i)$ is C^1 continuous (see Section 5.1) and that $V(i)$ is C^0 continuous (see Section 5.2). It follows that $1/V(i)$ is also C^0 continuous because $V(i)$ is positive. Because the derivative of $T(i)$ is $1/V(i)$, $T(i)$ is C^1 continuous. $T(i)$ is a one-to-one function. So for every i there is exactly one value for $T(i)$ and vice versa. As a result we can indeed take the inverse of $T(i)$. Taking the inverse of a one-to-one function does not change the continuity. So $U(t)$ is also C^1 continuous.

Remember that $W(t) = P(U(t))$. Since both $P(i)$ and $U(t)$ are C^1 continuous, $W(t)$ is also C^1 continuous. The viewing direction $D(t)$ equals:

$$D(t) = \frac{W(t+t_d) - W(t)}{|W(t+t_d) - W(t)|} \tag{3}$$

The function of the difference of two C^1 continuous functions divided by a positive number is also C^1 continuous. So $D(t)$ is a C^1 continuous function as we wanted to proof.

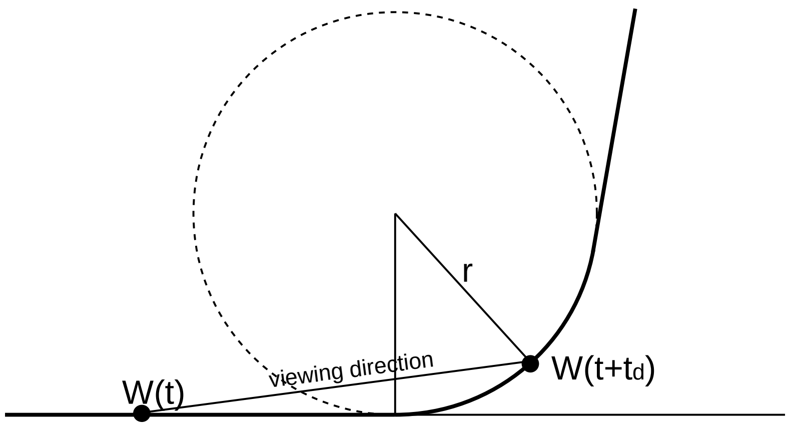

Figure 11: The viewing direction

Theorem 6.1 *When moving along the camera path and looking at a position on the path t_d ahead, the viewing direction is a C^1 continuous function.*

Experiments with a number of persons show that the above method is very effective. Setting d_t very close to 0, that is, looking in the direction of motion quickly caused motion sickness and the motion was unpleasant to look at. Setting d_t to about 1 second removed the motion sickness and gave the best result.

6.2 Up Vector

We have now smoothed the camera motion and the viewing direction. This leaves us with one more parameter to deal with: the roll of the camera around its longitudinal axis. This roll is often indicated by the up vector. The preferred direction of the up vector is parallel to the y-axis, since then the horizon of the camera is equal to the horizon in the normal world (see Section 2). When the up vector changes too much, the user feels the effect of being in a roller coaster.

A straightforward solution would be to always let the up vector point in the direction of the y-axis. But this can lead to problems. When the camera makes a looping-like motion, the up vector normally points down halfway the motion. If we force it to point upwards, it rotates the camera 180° about the viewing direction in a very short period of time. This sudden rotation is called a twist and should be avoided.

We have investigated a number of solutions to deal with the twist. The best solution was to try to avoid situations where a twist can occur. There is a high chance of twists when the plane spanned by two consecutive edges in the path has a small angle with the y-axis. We cannot forbid such pairs of edges altogether because they might constitute the only possible path. But we can discourage them by including this angle in the penalty function for finding shortest paths in Section 4.2. Experiments showed that an exponential penalty function works best, so that a path with a twist is only selected if there is really no alternative. If a twist still occurs we can adapt the speed of motion to make it less dramatic. In our experiments though, always another path without twists was returned by the algorithm.

6.3 Start and Goal

Up to now we assumed that the camera starts with a viewing direction along the first edge of the path and end with a direction along the final edge. This is of course not the case. In particular at the start of the path the camera is most likely pointing in a different direction. At the goal it depends whether the user also specifies the final orientation or only the final position. In the latter case no addition processing is required there.

Let us consider the start situation. (The goal can be treated in a similar way.) The camera is looking in some direction d and the first edge of the path has direction d'. We assume that there is no twist but this can also be taken into account. After trying a number of different approaches we found that the most natural solution is to start with a rotation from d to d' before starting to move the camera. The rotation is performed using Euler angles, because this more naturally corresponds to our own way of turning (we move our head up or down and rotate around our vertical axis). We compute a speed function for this rotational part of the motion. To avoid stopping at orientation d' before starting to move with the camera (which is unnatural) we blend the last part of the rotation with the start of the camera path.

7 The CAVE System

The approach to generating camera movements as described above has been implemented in the form of a C++ library called CAVE (CAmeras in Virtual Environments). The library has a very simple API, allowing for easy integration in different virtual environment systems. CAVE does not require a description of the scene in any format. It only requires a pointer to the collision checker of the VE system in which it is used. It uses this collision checking to determine whether camera paths are collision free.

CAVE first of all contains routines to create, store, load, and extend roadmaps, using the PRM approach described in Section 3. All the VE system has to do is to provide an appropriate collision checking function, a bounding box of the scene, and a minimal clearance. CAVE can create both two-dimensional roadmaps for moving through the virtual environment at a fixed height, and three-dimensional roadmaps.

Secondly there are routines to compute a path between a given start position and orientation and a required goal position (and orientation, if required). Some parameters can be set, like the maximal speed and acceleration. The routines return the time it takes to reach the goal and next the VE system can query the path for the position and orientation at different moments in time. The VE system can then use this information to visualize the motion. This approach leaves the final control of the camera to the VE system, making it easier to e.g. incorporate user input.

The CAVE library is available in the form of a DLL from the CAVE website:

`http://www.give.nl/cave/`

Here you can also find further information about the API and about licensing the library, plus a demonstration program.

To test the effectiveness of the approach we incorporated the library into some architectural walkthrough applications by the company LetsLook (`http://www.letslook.com/`). The walkthroughs mainly consisted of the scene with some dynamic objects plus a simple mechanism for walking through the environment using a combination of cursor keys and the

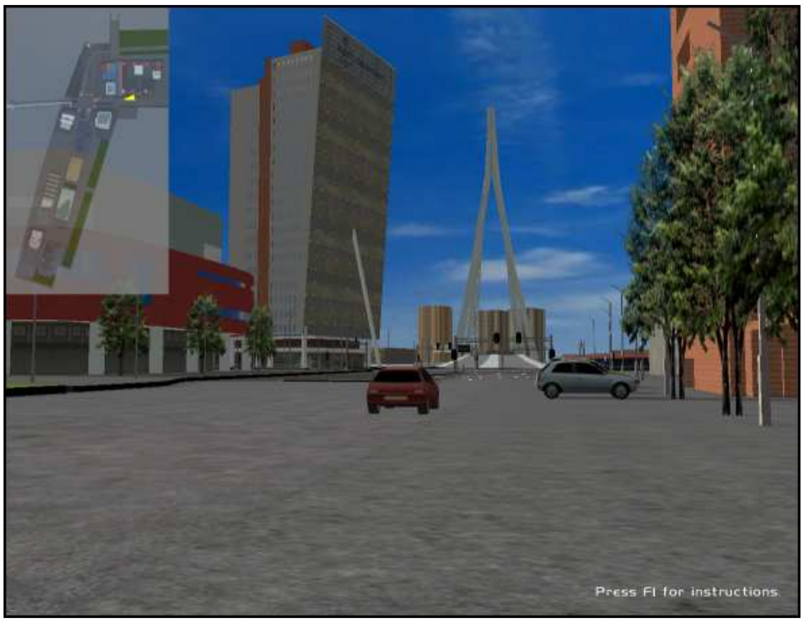

Figure 12: The interface for the walkthrough of part of Rotterdam using the CAVE system for planning camera movements. (Scene created by LetsLook.)

mouse. The applications have been created within the development environment Quest3D (`http://www.quest3d.com/`). Quest3D uses a graphical representation for creating VE applications. Incorporating the CAVE library in Quest3D was just a few hours of work. Next we adapted the walkthroughs by adding a clickable map and some additional position feedback. In Figure 12 you find an image of the interface. At the top left there is the clickable map with a triangle indicating the current position and orientation of the camera. The user can click on any location in the map to indicate that the camera should move there.

Preprocessing the scene of part of Rotterdam in Figure 12 to create an adequate roadmap using PRM took about 5 seconds (on a Pentium 4, 2.4 GHz). After this preprocessing smooth motions can be computed between any pair of positions in less than half a second. This feels like immediate response. Experiments show that the technique works very robust. Even when the user places the camera almost against a wall, the method successfully finds a motion. Different users confirmed that the resulting motions were pleasant to look at. A video of a typical camera motion in the scene is included.

The walkthrough of Rotterdam uses a two-dimensional roadmap at a fixed height. To test our approach in a full three-dimensional environment we also incorporated CAVE in a model of a building (Figure 13), again provided by LetsLook. Using the same setup, building a roadmap took about 8 seconds. Motions can again be created in less than half a second. This scene is a good environment to test our solution to deal with the up vector when going from a low to a high position in the scene. Again, we showed the resulting motions to a number of people that confirmed the quality of the results. A video of a typical camera motion in the scene is included.

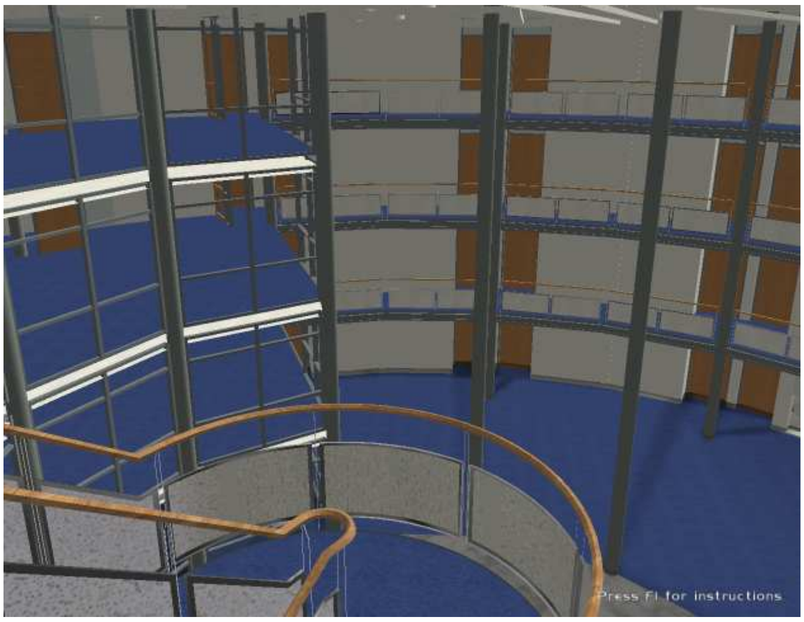

Figure 13: A 3D implementation of CAVE. (Scene created by LetsLook.)

8 Conclusions

In this paper we have describe a new approach to automatically planning camera motions in first-person views of virtual environments. The technique is based on a novel application of the Probabilistic Roadmap Planning approach originally developed in robotics. Combining the approach with theory for cinematography and applying various smoothing techniques we obtained a generic system that can easily be integrated in various virtual environment applications. The method is fast and versatile. Experiments with architectural walkthroughs verified the quality of the resulting motions.

The method described allows for free-flying camera motions, either on a constraint surface or in space. We are currently investigating extensions of the approach in which the camera must follow a "human" path, that is, stay at a particular height above the ground, incorporating the possibility to climb up stairs and ladders. We are also investigating the application of a similar approach in third-person view applications were the motion of the object to follow is (partially) known in advance. Here a key additional complication is that the view of the object should not be obstructed by obstacles in the scene.

References

[1] N. Amato, O. Bayazit, L. Dale, C. Jones, D. Vallejo, OBPRM: An obstacle-based PRM for 3D workspaces, in: P.K. Agarwal, L.E. Kavraki, M.T. Mason (eds.), *Robotics: The algorithmic perspective*, A.K. Peters, Natick, 1998, pp. 155–168.

[2] N. Amato, Y. Wu, A randomized roadmap method for path and manipulation planning, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1996, pp. 113–120.

[3] W.H. Bares, J. Grgoire, J. Lester, Realtime constraint-based cinematography for complex interactive 3D worlds, *IAAI-98: Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence,* 1998, pp. 1101–1106.

[4] W.H. Bares, S. Thainimit, S. McDermott, A model for constraint-based camera planning, *Smart Graphics. Papers from the 2000 AAAI Spring Symposium,* AAAI Press, 2000, pp. 84–91.

[5] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, P. Raghavan, A random sampling scheme for path planning, *Int. Journal of Robotics Research* **16** (1997), pp. 759–774.

[6] V. Boor, M.H. Overmars, A.F. van der Stappen, The Gaussian sampling strategy for probabilistic roadmap planners, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1999, pp. 1018–1023.

[7] N. Courty, E. Marchand, Computer animation: a new application for image-based visual servoing, *Proc. IEEE Int. Conf. on Robotics and Automation,* 2001, pp. 223–228.

[8] S.M. Drucker, D. Zeltzer, CamDroid: A system for implementing intelligent camera control, In: P. Hanrahan and J.Winget (Eds), *SIGGRAPH Symposium on Interactive 3D Graphics,* 1995, pp. 139–144.

[9] H.H. Gonzalez-Banos, C.Y. Lee, J.C. Latombe, Real-time combinatorial tracking of a target moving unpredictably among obstacles, *Proc. IEEE Int. Conf. on Robotics and Automation,* 2002.

[10] N. Halper, R. Helbing, T.Strothotte, Computer games: A camera engine for computer games, *Computer Graphics Forum* **20** (2001).

[11] N. Halper, P. Olivier, CAMPLAN: A camera planning agent, *Smart Graphics. Papers from the 2000 AAAI Spring Symposium,* AAAI Press, 2000, pp. 92–100.

[12] A. Hanson, E. Wernert, Constrained 3D Navigation with 2D Controllers, *IEEE Visualisation*, 1997, pp. 175–182.

[13] C. Holleman, L. Kavraki, J. Warren, Planning paths for a flexible surface patch, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1998, pp. 21–26.

[14] D. Hsu, L. Kavraki, J.C. Latombe, R. Motwani, S. Sorkin, On finding narrow passages with probabilistic roadmap planners, in: P.K. Agarwal, L.E. Kavraki, M.T. Mason (eds.), *Robotics: The algorithmic perspective*, A.K. Peters, Natick, 1998, pp. 141–154.

[15] L. Kavraki, *Random networks in configuration space for fast path planning,* PhD thesis, Stanford University, 1995.

[16] L. Kavraki, J.C. Latombe, Randomized preprocessing of configuration space for fast path planning, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1994, pp. 2138–2145.

[17] L. Kavraki, P. Švestka, J-C. Latombe, M.H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, *IEEE Trans. on Robotics and Automation* **12** (1996), pp. 566–580.

[18] F. Lamiraux, L.E. Kavraki, Planning paths for elastic objects under manipulation constraints, *Int. Journal of Robotics Research* **20** (2001), pp. 188–208.

[19] S.M. LaValle, H.H. Gonzalez-Banos, C. Becker, J.-C. Latombe, Motion strategies for maintaining visibility of a moving target, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1997.

[20] T.-Y. Li, T.-H. Yu, Planning object tracking motions, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1999.

[21] J. Mackinlay, S. Card, G. Robertson, Rapid controlled movement through a virtual 3d workspace, *Computer Graphics (SIGGRAPH '90 Proceedings),* 1990, pp. 171-176.

[22] G. Millerson, *TV Camera Operation,* Focal Press, 1973.

[23] C. Nissoux, T. Siméon, J.-P. Laumond, Visibility based probabilistic roadmaps, *Proc. IEEE Int. Conf. on Intelligent Robots and Systems,* 1999, pp. 1316–1321.

[24] M.H. Overmars, *A random approach to motion planning,* Technical Report RUU-CS-92-32, Dept. Comput. Sci., Utrecht Univ., Utrecht, the Netherlands, 1992.

[25] T. Simeon, J. Cortes, A. Sahbani, J.P. Laumond, A manipulation planner for pick and place operations under continuous grasps and placements, *Proc. IEEE Int. Conf. on Robotics and Automation,* 2002.

[26] P. Švestka, *Robot motion planning using probabilistic roadmaps,* PhD thesis, Utrecht Univ. 1997.

[27] P. Švestka, M.H. Overmars, Motion planning for car-like robots, a probabilistic learning approach, *Int. Journal of Robotics Research* **16** (1997), pp. 119–143.

[28] P. Švestka, M.H. Overmars, Coordinated path planning for multiple robots, *Robotics and Autonomous Systems* **23** (1998), pp. 125–152.

[29] S. Sekhavat, P. Švestka, J.-P. Laumond, M.H. Overmars, Multilevel path planning for nonholonomic robots using semiholonomic subsystems, *Int. Journal of Robotics Research* **17** (1998), pp. 840–857.

[30] M. Wayne, *Theorising Video Practise,* 1997.

[31] S.A. Wilmarth, N.M. Amato, P.F. Stiller, MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space, *Proc. IEEE Int. Conf. on Robotics and Automation,* 1999, pp. 1024–1031.