Camera Planning in Virtual Environments Using the Corridor Map Method

Roland Geraerts*
Institute of Information and Computing Sciences, Utrecht University
3508 TA Utrecht, the Netherlands. E-mail: roland@cs.uu.nl

Abstract. Planning high-quality camera motions is a challenging problem for applications dealing with interactive virtual environments. This challenge is caused by conflicting requirements. On the one hand we need *good motions*, formed by trajectories that are collision-free and keep the character that is being followed in clear view. On the other hand, we need *frame coherence*, i.e. the view must change smoothly such that the viewer does not get disoriented. Since camera motions dynamically evolve, good motions may require the camera to jump, leading to a broken frame coherence. Hence, a careful trade-off must be made. In addition to this challenge, interactive applications require real-time computations, preventing an exhaustive search for 'the best' solution.
We propose a new method for planning camera motions which tackles this trade-off in real-time. The method can be used for planning camera motions of NPC's and first-person characters. Experiments show that high-quality camera motions are obtained for both scenarios in real-time.

1 Introduction

In interactive virtual environment applications, such as games, training systems and architectural applications, a virtual camera needs to be steered. In contrast to first-person games, in which the camera is steered by the user, we focus at applications which require automatic planning of camera motions. In such applications, manual camera control would require too much mental energy or would be too difficult for inexperienced users [1, 2].

A camera motion consists of two paths: a camera path, which describes the camera positions in the environment over time, and an aim path, which describes the corresponding positions the camera looks at. We aim at creating good camera motions which feature a high *spatial awareness* of the viewer and coherent motions preventing the viewer from getting *motion sick*.

The viewer's spatial awareness, i.e. the knowledge of the viewer's location and orientation within the environment, can be maintained if the following constraints are satisfied. The camera and aim positions must not collide with obstacles in the environment. In addition, the character should always be in view.

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

Finally, the camera should not be too close to the character or obstacles (otherwise a large part of the view would be blocked), nor should it be too far (because the camera needs to stay behind the character to maintain the understanding of its orientation). This can be accomplished by keeping some minimal amount of clearance from the camera to the obstacles in the environment. Also, the camera should be placed behind the character at a certain preferred distance.

A viewer can get motion sick when its view changes too abruptly, and, hence, the positions on the paths should change smoothly and the angular velocity of the camera view should be minimal. Also, motion-sickness can be reduced by anticipating the direction in which the view is going. Hence, keeping frame coherence is of major importance.

Automated camera planning is a difficult task because a careful trade-off must be made between satisfying the constraints and frame coherence [3]. That is, favoring the constraints (e.g. keeping the character always in view) can lead to a broken frame coherence. Likewise, favoring frame coherence can lead to situations where the character blocks the user's view.

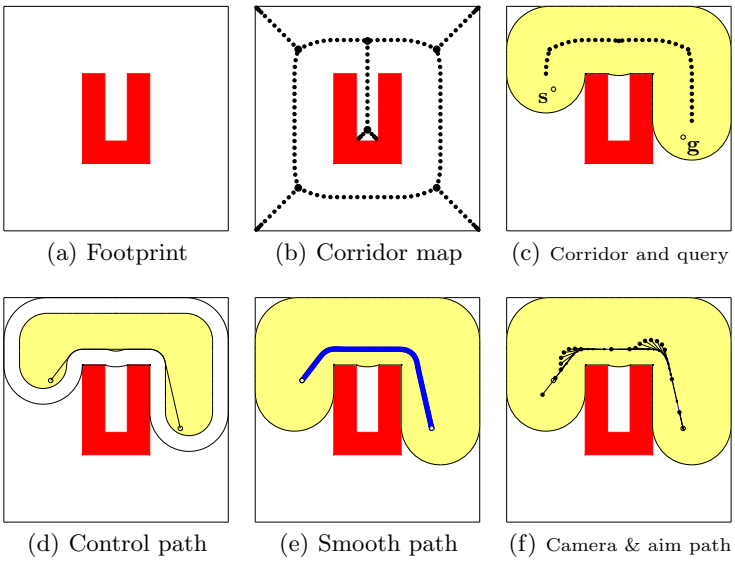
In related work, a distinction can be made between methods that deal with camera planning for *known* input paths [2–4] versus *unknown* paths [1, 3, 5]. Applications that deal with known input paths are games and architectural applications. In a game, the known path could be a character's path which is replayed or an NPC's (Non-Player Character) path which is being followed. In an architectural application, the path could be a walk-through in a virtual environment. A typical application in which the character's positions are created dynamically (so the path is unknown in advance) is a third-person game.

We will present a new method for planning good camera motions in real-time. We induce collision-free motions by using the Corridor Map Method, which is a global path planning technique [6, 7]. The character is kept in clear view by anticipating on the future trajectory. Next, frame coherence is maintained by modeling the motions as forces applied to the camera and its view.

The paper is organized as follows. In Section 2 we will review the Corridor Map Method which is the framework we use for the new method. Its key ingredient, i.e. creating smooth collision-free paths, is discussed in Section 3. We will then introduce our new method in Section 4 which creates smooth camera motions, composed of a smooth camera and aim path, for a known character path. We will adjust the method in Section 5 such that we can deal with paths whose positions are unknown in advance. Next, we will conduct experiments in Section 6 to test the quality and performance of the method and conclude in Section 7 that high-quality camera motions are computed in real-time.

2 The Corridor Map Method

The *Corridor Map Method* (CMM) has been designed for path planning in real-time interactive environments and games [8]. The strength of the CMM is that it combines a fast global planner with a powerful local planner, providing real-time performance and the flexibility to handle a broad diversity of path planning



(a) Footprint (b) Corridor map (c) Corridor and query

(d) Control path (e) Smooth path (f) Camera & aim path

Fig. 1. The construction phase (a–b) and query phase (c–f) of the CMM.

problems. The CMM consists of an off-line construction phase and a on-line query phase (see Fig. 1). In the construction phase, we build a *Corridor Map* which is a data structure that represents the free space, i.e. the 2D walkable space, in a virtual environment. The free space is defined as the space that is not occupied by the *footprint* of the environment. This footprint is made up of a collection of geometric primitives, such as points, lines, polygons, and disks, all lying in the ground plane. The underlying graph of the map is the Generalized Voronoi Diagram [7]. The edges of this graph are sampled. We assign to each sampled point on an edge the radius of the largest empty disk. Each point/radius combination forms a maximum clearance disk. Then, a sequence of disks forms a *Corridor*, and a system of corridors is referred to as the *Corridor Map*.

In the query phase, we use the Corridor Map to compute the shortest corridor which encloses the future path of the character. We connect the start and goal positions of the character to the map and retrieve the shortest path in the map connecting these positions. We refer to this path as the *backbone path* of the corridor. The corresponding sequence of disks forms the corridor which guides the global motions of the character (or camera). An example of such a corridor, backbone path (small disks), and query is displayed in Fig. 1(c).

The local motions of a character are created by following an *attraction point* which moves along a *control path* toward the goal. In [6], we used the backbone path as control path. If a character is attracted by this path, its final path will have much clearance because the backbone path is composed of maximum clearance points. Nevertheless, the final path might be longer than is necessary. We are interested in a short control path that has some amount of minimum

clearance to the environment. Such a path is created by shrinking the corridor with the preferred amount of clearance. The control path is then the shortest path inside the shrunk corridor [9]. We refer the reader to Fig. 1(d) for an example of such a path. It might be tempting to use the control path as final path for the character. However, this path is not smooth. In addition, traversing the fixed control path would decrease the flexibility of the method. Because the character is able to deviate from the control path, the character's motions can become smooth as is shown in Fig. 1(e).

In the next section we will discuss the technique for creating a smooth path for the character. This technique will form the basis for creating smooth camera motions such as visualized in Fig. 1(f). The two open disks denote the character's start and goal positions. The black closed disks denote the camera positions and the arrows denote the aim positions.

3 Creating a smooth path

Given a start position $\mathbf{s}$ and goal position $\mathbf{g}$, we extract a corridor from the corridor map which encloses the future path connecting the start to the goal. Next we extract a control path which is represented by a continuous sequence of points. This control path is then used to create the smooth path.

The algorithm computes a path Π which is defined as a continuous sequence of two-dimensional positions. Each position is annotated with a time stamp. We require that the character, modeled by a disk with radius r, is inside the corridor for any position along the path. More formally, we define a path as follows:

Definition 1 (Path Π)**.** *A path Π inside a corridor $\mathcal{C}$ is a continuous map $\Pi \in [0, t_{max}] \rightarrow \mathbb{R}^2$ such that $\forall t \in [0, t_{max}] : \Pi[t] \in \mathcal{C}$.*

By $\Pi[t]$ we denote the position at time t. For example, the start and goal position are equal to $\mathbf{s} = \Pi[0]$ and $\mathbf{g} = \Pi[t_{max}]$, respectively. Of course, the value for t_{max} will be known after the computation of the path.

The algorithm iteratively computes positions and time stamps that make up the path. The algorithm finishes when the Euclidean distance between the last computed position and the goal is smaller than a small threshold $\epsilon = 0.01$.

The character is initially placed at position $\mathbf{x} = \mathbf{s}$ and is set into motion and steered by receiving steering and boundary forces [8]. The steering force $\mathbf{F}_s(\mathbf{x})$ guides the character toward the goal and is defined by the difference between the character's attraction point $\alpha(\mathbf{x})$ and the character's current position $\mathbf{x}$. For computing the attraction point, we first need to find the closest point on the backbone path. Given its associated disk, we compute $\alpha(\mathbf{x})$ as the furthest intersection point between this disk and the control path. The line segment connecting $\mathbf{x}$ with $\alpha(\mathbf{x})$ is free of collisions, ensuring a collision-free path.

The boundary force $\mathbf{F}_b(\mathbf{x})$ pushes the character away from the corridor's boundary, and, hence, it keeps the character inside the corridor. If the character's clearance to the boundary is larger than its preferred safe distance then this force is $\mathbf{0}$. (We set the safe distance to the character's radius r.) Otherwise, the force

is directed away from the boundary and its magnitude goes to infinity when the character touches the boundary.

The combined force is defined as $\mathbf{F}(\mathbf{x}) = \mathbf{F}_s(\mathbf{x}) + \mathbf{F}_b(\mathbf{x})$. When this force is applied to the character, it starts accelerating. This phenomenon is summed up by Newton's Second Law, i.e. $\mathbf{F} = M\mathbf{a}$, where M is the mass of the character and $\mathbf{a}$ is its acceleration. Without loss of generality, we assume that $M = 1$. Hence, the force can be expressed as $\mathbf{F}(\mathbf{x}) = \frac{d^2\mathbf{x}}{dt^2}\ m/s^2$. Combining the two expressions for $\mathbf{F}(\mathbf{x})$ gives us $\frac{d^2\mathbf{x}}{dt^2} = \mathbf{F}_s(\mathbf{x}) + \mathbf{F}_b(\mathbf{x})$, which is an equation providing the positions for the character. Because this equation cannot be solved analytically, we have to revert to a numerical approximation, such as Euler integration [10], to compute the positions of the character's path.

In Euler's integration scheme, the position $\mathbf{x}$ is updated by moving it during Δt time with velocity $\mathbf{V}$. The velocity is updated similarly. If its magnitude becomes larger then the maximum velocity v_{max}, then $\mathbf{V}$ is scaled such that its magnitude becomes equal to v_{max}. Consequently, the character's speed is limited to the maximum velocity. The new position is added to the path and is annotated with the current time stamp. Finally, the time is increased with the step size Δt and the loop continues.

The algorithm produces a smooth path, i.e. the path is C^1-continuous when $\Delta t \rightarrow 0$, because the positions are the result of a function that was doubly integrated [6]. We use such a smooth path as input for creating camera motions.

4 Following a known path

In this section we will present an algorithm which computes a *camera motion* for a path whose positions are known (or can be computed) in advance. Applications such as games (replay a path, follow an NPC) and architectural applications (create a virtual walk-through inside a city) can benefit from the approach.

A camera motion consists of two paths which comply to Definition 1, i.e. a camera path Π_{cam}, which describes the camera positions in the environment over time, and an aim path Π_{aim}, which describes the corresponding positions the camera looks at. The algorithm consists of the following steps.

The first step is to ensure that we have the right input. The next step is to find a proper initial camera placement if this is not given. Then, the camera and aim paths are computed. Next, additional constraints are incorporated into the algorithm. Finally, we handle the case in which a character stops moving, e.g. when the character reaches its destination.

Initialization As input we have a known path which may be traversed by a character or may be a path observed by the camera. We call this the character path Π_{char}. In an architectural application, such a path may not be given, and, hence, we need to create a path from a given start to a given goal. We construct this path (and corresponding corridor) by the approach from the previous section. By constructing a path that has enough clearance, we can create camera motions that will not run too close to the obstacles. Also, having enough clearance will decrease the need for sharp turns and prevents views that are

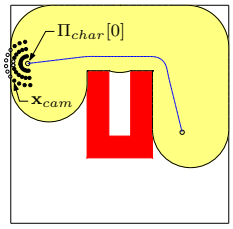


Fig. 2. Computing a collision-free initial placement of the camera. The camera can be placed on the black disks but not on the white disks.

blocked by the obstacles. Since the path is also short and smooth, the final camera motions will be efficient and smooth too. If path Π_{char} is given in advance, we smooth it first for the same reasons mentioned above. Next, we compute a corridor $\mathcal{C}$ containing this path. Details on this procedure can be found in [8].

Besides path Π_{char} and corridor $\mathcal{C}$, we have the following input. The parameter d_{cam} denotes the preferred distance from the camera to the character. Next, the camera looks at a future position of the character. The corresponding look ahead time is denoted by t_{la}. Finally, we are given the maximum velocity v_{max} of the camera and aim. This value should be at least at large as the character's maximum velocity to assure that the camera can keep the character in view.

Before we create the camera path, we need to set the initial camera and aim positions. The camera initially looks at the character's start position. Since the camera stands still, its aim and camera velocity is set to $\mathbf{0}$.

Placement of the camera If the camera position $\mathbf{x}_{cam}$ is not provided in advance, we need to choose a proper camera placement (see Fig. 2). The camera, modeled as a disk with radius r, does not have to be placed inside the corridor. We try to place the camera behind the character at the preferred distance d_{cam}. By 'behind' we mean a position on the line segment which starts at $\Pi_{char}[0]$ and is extended in the direction $\mathbf{v} = \Pi_{char}[0] - \Pi_{char}[\Delta t]$, where Δt is a small time step. The initial preferred placement is then given by $\mathbf{x}_{cam} = \Pi_{char}[0] + d_{cam} * \mathbf{v}/||\mathbf{v}||$. If this camera placement or the line through $\mathbf{x}_{cam}$ and $\Pi_{char}[0]$ is not inside the free space, we need to try other placements. We first consider a series of placements on the half circle with center $\Pi_{char}[0]$, radius d_{cam}, behind the line that is perpendicular to direction $\mathbf{v}$. If all placements collide, we decrease the radius (e.g. by $d_{cam}/3$) and test a new series of placements. We continue this procedure until we find a collision-free placement $\mathbf{x}_{cam}$. After finding $\mathbf{x}_{cam}$, we test whether $\mathbf{x}_{cam}$ is inside the corridor $\mathcal{C}$. If it is outside the corridor, we extract a new one enclosing the path from $\mathbf{x}_{cam}$ to $\Pi_{char}[0]$ and add it to $\mathcal{C}$.

Creating the camera and aim paths Like in Section 3, we compute a smooth camera and aim path by iteratively updating their positions and time stamps until the last position of the character's path is being viewed by the camera. For the aim as well as the camera path, we define corresponding attraction points.

For the aim's attraction point $\alpha(\mathbf{x}_{aim})$ we take a future position on the character's path determined by the current time stamp t plus the time lapse t_{la}.

Since the camera starts looking at a location in which the character will be in t_{la} time, the viewer gets a hint about the possible changes in direction, leading to a smaller chance of getting motion sick. The force, which will be applied to the aim position $\mathbf{x}_{aim}$ is computed like in Section 3. This force is also composed of a steering force (i.e. $\alpha(\mathbf{x}_{aim}) - \mathbf{x}_{aim}$) and boundary force. Next, we integrate the force function, compute the new aim position, and add it to the aim path.

The attraction point for the camera $\alpha(\mathbf{x}_{cam})$ is computed differently. The camera is attracted to its optimal position, which is located behind the character on the line which runs through the character and has the same direction as the current aim direction $\mathbf{v}$, where $\mathbf{v} = \Pi_{aim}[\Delta t] - \Pi_{aim}[t - \Delta t]$. Hence, $\alpha(\mathbf{x}_{cam}) = \Pi_{char}[t] - d_{cam} * \mathbf{v}/||\mathbf{v}||$. If $t = 0$, then the aim position $\Pi_{aim}[t - \Delta t]$ does not exist; we then use $\alpha(\mathbf{x}_{cam}) = \mathbf{x}_{cam}$ instead. We need to ensure that the camera can 'see' its attraction point, otherwise the camera could get stuck behind an obstacle. This can be checked by determining whether the line segment through $\mathbf{x}_{cam}$ and $\alpha(\mathbf{x}_{cam})$ is inside the corridor. If they cannot see each other, we iteratively decrease the preferred camera distance d_{cam} and update $\alpha(\mathbf{x}_{cam})$ correspondingly until the line segment is inside $\mathcal{C}$. The camera's steering and border forces are then computed like the aim's forces. These forces are applied to the camera, resulting in a new camera position which is added to the path.

Satisfying additional constraints We have now described a procedure for creating camera motions which satisfy almost all criteria mentioned in the introduction. That is, the camera and aim paths are collision-free because they are kept inside the corridor by the boundary force. Moreover, this force assures a minimum clearance to the obstacles so that the view is blocked less by the obstacles. Next, the camera is placed as much as possible at a certain preferred distance behind the character to keep a good view. This criterion, together with the fact that the two paths change smoothly prevent motion-sickness.

Two constraints however may not have been satisfied yet. First, when the character is inside a narrow passage, it may not be in the camera's view. If the camera cannot see its corresponding aim point, we anticipate on the blocked view by temporarily increasing the camera's maximum velocity and decreasing the preferred distance d_{cam} to the camera. We increase the velocity v_{max} in each iteration until they see each other. Hence, the camera's speed will change smoothly. Besides, we iteratively decrease d_{cam}. When the character is in the camera's view again, we decrease v_{max} and increase d_{cam} until they have reached their initial values. As a result, the camera speeds up before the character would get out of view and the character stays (longer) in view. Nevertheless, increasing the camera's velocity may increase the camera's *angular velocity*, which may cause the view to change too fast. The angular velocity v_{rot} is defined by the angle between the current camera direction and the previous one. If v_{rot} becomes larger than some predefined maximum angular velocity v_{rot_max}, we decrease the camera's maximum velocity, like we described in the previous paragraph, and increase the velocity again when $v_{rot} \leq v_{rot\_max}$. Note that these two constraints contradict each other. Depending on the application, we can favor one particular constraint. We will discuss this trade-off further in Section 6.

Reaching the destination A nice property of modeling motion by forces is that the camera and aim smoothly accelerate until a they reach a certain maximum velocity. When the character (suddenly) stops moving, they both can have a high velocity, and, hence, they continue moving for a short while, which may result in oscillating motions near their steady attraction points. We handle this problem by appropriately slowing down the camera and aim during a short amount of time t_s (e.g. $t_s = 1$ second). We set the aim's attraction point to the character's current location (if it has not reached this location yet). The camera's attraction point is left unchanged. Next, during t_s time, we lower the maximum velocity like we did in the previous section. Such a scheme assures that the camera (and aim) smoothly decelerate within t_s seconds.

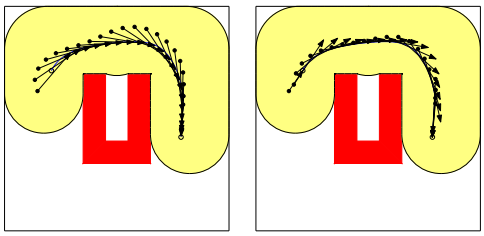
5 Following an unknown path

In third person games, a camera needs to follow a character which is controlled by the player. Consequently, the character's path is being created dynamically. We will adjust our approach such that the camera motions are also computed on the fly. This means that we cannot refer anymore to future character positions. This happens twice in our approach. First, the algorithm that finds the initial camera position assumes that the character's initial direction is known. Since we think this is a reasonable assumption, we require that this direction is given as input to the algorithm. Second, we need to adjust the computation of the aim's attraction point $\alpha(\mathbf{x}_{aim})$. We place $\alpha(\mathbf{x}_{aim})$ on its *estimated* future position. Starting from the character's current location $\Pi_{char}[t]$, we move in the aim's direction $\mathbf{v} = \Pi_{char}[t] - \Pi_{char}[t - \Delta t]$ with displacement $t_{la} * v_{max}$. If $t = 0$, we set $\alpha(\mathbf{x}_{aim})$ to the character's start position.

We have observed that good estimations are obtained when a smoothed version was used of the character's path instead of the original path. Each position (with time stamp t) is smoothed by computing an averaged position over a small time window t_w (e.g. $t_w = 0.25$ seconds).

Finally, the algorithm makes use of a corridor which is unknown in advance. We dynamically update the corridor while the character moves. That is, if the character moves out of the corridor (which initially is computed by the sequence of disks that connect the initial camera position with the initial character position), we add the new disk that encloses the character's new position.

We refer the reader to Fig. 3 which visualizes the effect of having knowledge (or not) about the character's future positions. To compare these effects, we used the same input character path. If the character's positions are *known* in advance (left picture), then the camera's motions anticipate well to changes of the character's future positions. That is, the character's positions (disks) are located behind the character as is required and its aim (arrows) is targeted at the character's near future positions. If the character's positions are *unknown* in advance, its future positions are estimated. Consequently, the camera motions react later on a change in the character's direction then the case in which the path is known in advance.



(a) Known path (b) Unknown path

Fig. 3. Camera motions induced by knowing (or not) the character's path in advance.

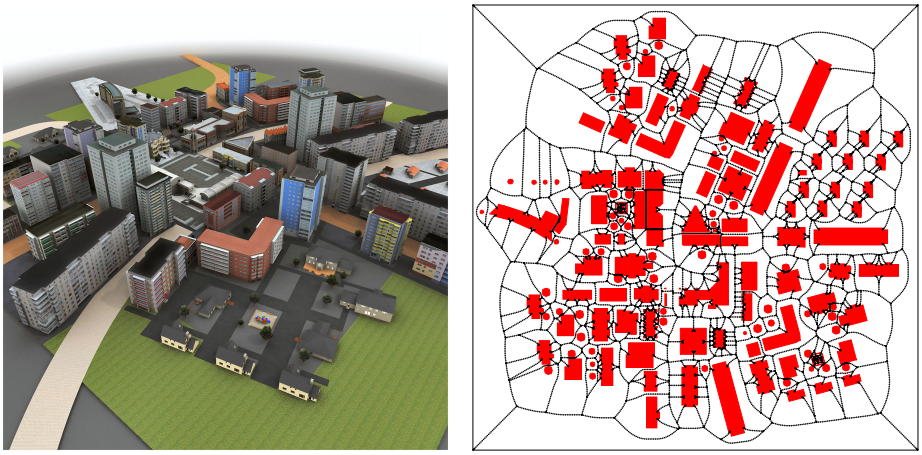
6 Experiments

We have tested our method inside a large virtual city. We experimentally verified whether the method can create high-quality camera motions in real-time. We implemented two applications in Visual C++ under Windows XP. The first one generated the Corridor Map of the city [7]. The second one was our CMM framework [6] in which we integrated camera planning. All the experiments were run on a PC with a NVIDIA GeForce 8800 GTX graphics card and an Intel Core2 Quad CPU (2.4 GHz) with 4 GB memory. Our application used one core.

We conducted experiments with the city environment depicted in Fig. 4. The city measured 500x500m. Its footprint (2,122 triangles) formed the input primitives for the Corridor Map. Creating the map took 0.64s, resulting in 1,434 vertices, 1,606 edges and 25,863 samples (i.e. the number of disks in all corridors). The characters' radius was set to $r = 0.25$ and the time step to $\Delta t = 0.1s$.

We performed two experiments. In the first one, we extracted 1,000 uniform random collision-free queries, yielding 1,000 character paths. Each path was respectively treated as a path that was known or unknown in advance. We recorded the average running times for both cases. To view these times in the right perspective, we defined the CPU-load, which is the spent CPU time / averaged traversed time $*$ 100%. We set the character's maximum velocity to $2m/s$ which corresponds to a fast walking speed [11]. We considered a CPU-load of 1% to be real-time. In the second experiment, we selected one representative query. Its start was located in the lower part while its goal part was located in the upper part of the city. Then, a corridor, enclosing the query, was extracted from the Corridor Map [7]. The first half of the corridor corresponded to places with much clearance and little curvature while the second half corresponded to places with relatively little clearance and much curvature. Clearly, creating good camera motions will be more difficult in the second half.

We considered two cases. In the first case, we created camera motions for a virtual walk-through (so the character's path is known in advance). Its control path was constructed by extracting the shortest minimum clearance path inside this corridor [9]. By using much clearance (i.e. $3m$), a good overview of the environment was to be expected. We set the camera distance d_{cam} to $5m$, the time lapse t_{la} to $2s$ and maximum velocity v_{max} to $2m/s$. These settings favored



(a) City environment (b) Footprint and Corridor Map

Fig. 4. The test environment, its footprints and corridor map.

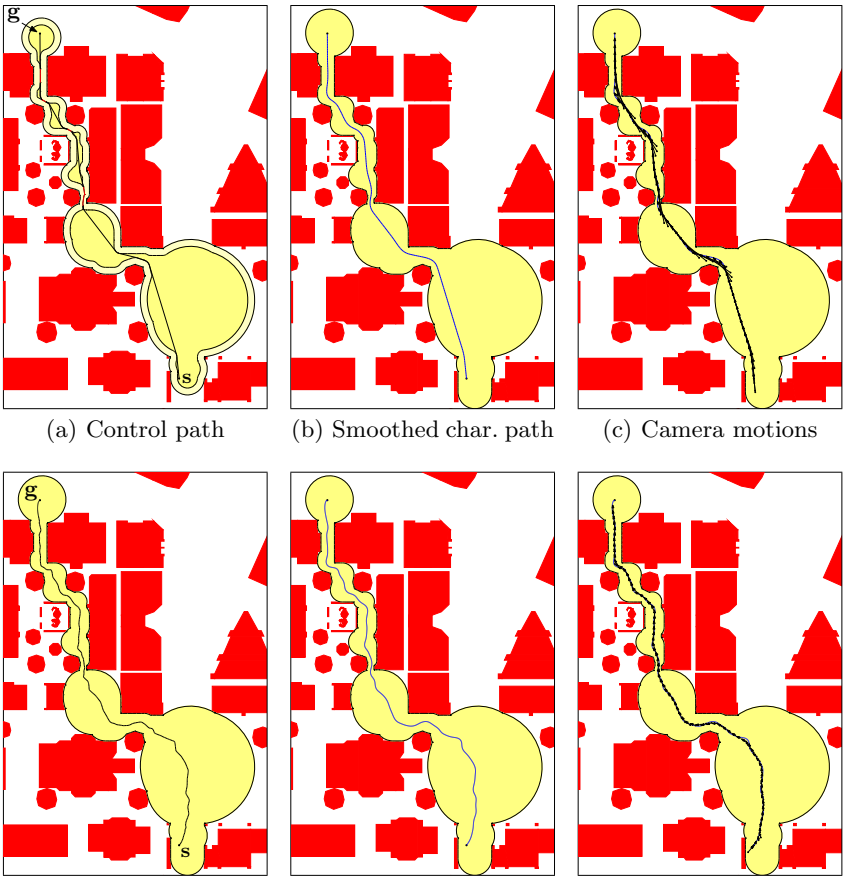
looking ahead and anticipated well to future directional changes. In addition, we used a low maximum angular velocity (i.e. $v_{rot\_max} = 2m/s$) at the expense of higher velocities at straight parts of the path. These settings favor a slow change of the user's rotating view.

In the second case, we created camera motions for following a character inside a third-person game (so the character's path is unknown in advance). We simulated a user-controlled path by creating a control path with many local changes [8]. In addition, the path sometimes ran close the obstacles, leading to an increased chance for a bad view. We used a shorter camera distance (i.e. $3m$) to keep a good third-person view and used a smaller time lapse (i.e. $0.5s$) to minimize the errors induced by wrongly estimating the future directions. We used $v_{rot\_max} = 5m/s$ to favor fast reactions on the character's motions.

Both control paths were smoothed. The algorithm created an aim and camera path, annotated with time stamps. The two paths were composed of x- and y-coordinates. A z-coordinate was added (i.e. $z = 1.8m$) to place the camera above the ground. We will now discuss the results.

For the known character path, the 1,000 queries were computed in $27.3ms$ on average. The averaged traversed time was $105.2s$. Hence, the CPU-load was 0.026%. For the unknown character path, the queries were computed in $29.6ms$ and the traversed time was $105.2s$, resulting in a CPU-load of 0.028%. Consequently, the running times can be considered as real-time.

The top row of Fig. 5 shows the results for the virtual walk-through. We used the smoothed control path to create the camera motions. The camera (disks) and aim path (arrows), as well as the view (i.e. the line connecting the camera with its aim) was collision-free as none of these glyphs intersected with the obstacles. The camera motions were smooth, i.e. there were no sudden changes in positions or orientations, even not at the locations with little clearance and high curvature. Since the latter is hard to see, we created a movie which can be viewed at



(a) Control path (b) Smoothed char. path (c) Camera motions

(d) Control path (e) Smoothed char. path (f) Camera motions

Fig. 5. Creating camera motions. A virtual walk-through is constructed for a *known* path (first row). A character's *unknown* path is followed (second row).

`http://people.cs.uu.nl/roland/motion_planning/camera`. The movie confirms that the motions were smooth. The maximum angle constraint reduced the camera's speed wherever required, leading to coherent motions. After a sharp corner, the camera's velocity was temporarily increased to make up for the delay. Finally, by anticipating on the character's positions, it was not lost out of sight.

The bottom row Fig. 5 shows the results for a dynamic third-person camera. The control path was rather bumpy. In each iteration, a small part of this path was smoothed. The final smoothed path is displayed in Fig. 5(e). By using smoothed character positions, relatively smooth camera motions were created as can be seen in Fig. 5(f). The figure also shows that no collisions occurred. We refer the reader to the corresponding movie. Is makes clear that the motions are indeed smooth. However, the view moved faster than in the case in which the character's path was known in advance, because the character's future positions

were being estimated while constructing the camera motions. However, these estimations and a small camera distance still made sure that the character was centered in the view. Consequently, the viewer would not get disoriented.

7 Conclusions

We introduced a new method for planning *good* camera motions in real-time. Good motions are composed of camera and aim positions which are collision-free and keep the character in clear view. Camera planning is a challenging problem because a sequence of good camera motions may not be continuous while continuity of the user's view (i.e. *frame coherence*) is of major importance. By smoothing the character's positions, looking ahead, slowing down when the view's angle gets too large, and by modeling the camera motions by forces, we have obtained good camera motions while keeping frame coherence. These motions were generated for both paths that were known as well as paths that were unknown in advance. Real-time performance was achieved by using the Corridor Map for answering collision and visibility queries.

We tested our approach on a static flat environment. We will extend the method such that a clear view is maintained as much as possible when dynamic changes occur, e.g. when another character blocks the user's view. Then, we would also like to handle 2.5D environments which include terrains and bridges.

References

1. Li, T.Y., Cheng, C.C.: Real-time camera planning for navigation in virtual environments. In: Smart Graphics. Volume 5166 of Lecture Notes in Computer Science., Springer (2008) 118–129
2. Nieuwenhuisen, D., Overmars, M.: Motion planning for camera movements. In: IEEE International Conference on Robotics and Automation. (2004) 3870–3876
3. Halper, N., Helbing, R., Strothotte, T.: A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. Eurographics **20** (2001) 174–183
4. Li, T.Y., Yu, T.H.: Planning tracking motions for an intelligent virtual camera. In: IEEE International Conference on Robotics and Automation. (1999) 1353–1358
5. Bourne, O., Sattar, A.: Evolving behaviours for a real-time autonomous camera. In: Australasian conference on Interactive entertainment. (2005) 27–33
6. Geraerts, R., Overmars, M.: The corridor map method: A general framework for real-time high-quality path planning. Computer Animation and Virtual Worlds **18** (2007) 107–119
7. Geraerts, R., Overmars, M.: Enhancing corridor maps for real-time path planning in virtual environments. In: Computer Animation and Social Agents. (2008) 64–71
8. Overmars, M., Karamouzas, I., Geraerts, R.: Flexible path planning using corridor maps. In: Algorithms – ESA. Volume 5193 of LNCS., Springer (2008) 1–12
9. Geraerts, R.: Planning short paths with clearance using explicit corridors. In: IEEE International Conference on Robotics and Automation (submitted). (2010)
10. Butcher, J.: Numerical Methods for Ordinary Differential Equations. Wiley (2003)
11. Knoblauch, R., Pietrucha, M., Nitzburg, M.: Field studies of pedestrian walking speed and start-up time. Transportation Research Record (1996) 27–38