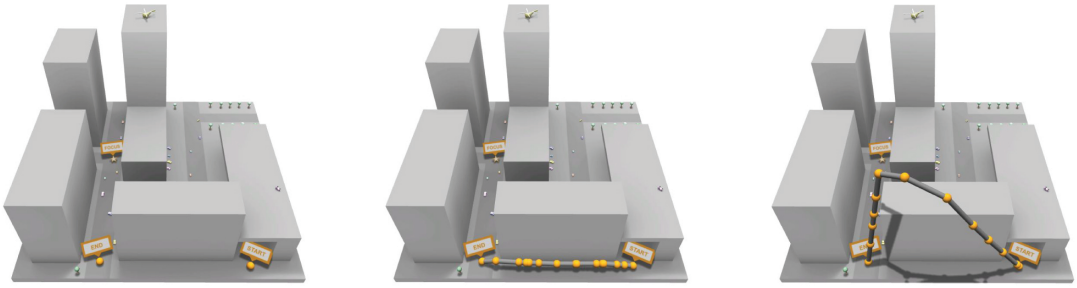
Visibility Transition Planning For Real-Time Camera Control



Thomas Oskam

Master Thesis
February 2008

Supervisors:
Prof. Dr. Markus Gross
Dr. Robert W. Sumner

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
Computer Graphics Laboratory ETH Zurich

Abstract

This thesis presents an algorithm for visibility transition planning that can compute large, occlusion free camera paths in real time through complex environments. The algorithm incorporates the visibility of a focus point into the optimally criteria, so that the chosen path strives to keep the focus target in view. The efficiency of the algorithm comes from a visibility roadmap data structure that allows the precomputation of a coarse representation of all collision-free paths through an environment, together with an estimate of the pair-wise visibility between all portions of the environment. The runtime system executes a path planning algorithm using the precomputed roadmap values to find a coarse path that is optimal in terms of visibility up to the resolution of the roadmap. Next, a more exact visibility estimate is determined by computing a sequence of occlusion maps along the coarse path. The same path-planning algorithm is executed on these occlusion maps to ensure optimal visibility on a fine scale. An iterative smoothing algorithm, together with a physically-based camera model, ensures that the path followed by the camera is smooth in both space and time. Finally, a dynamic camera controller is shown that encorporates the visibility transition planning into a real-time re-routing system that is able to follow a fast paced game character in a complex environment.

Contents

List of Figures

1

Introduction

In any virtual environment, computer game, or other interactive application, the chosen camera viewpoint and its motion are crucial for a positive user experience. Successful navigation through content strongly depends on appropriate camera movement that seems natural to the user. In the same way that good camera work can enhance, the user experience bad camera work can destroy it. When considering automatic camera control, four critical criteria include:

1. **Real-time.** Static or precomputed viewpoints are unsuitable for interactive environments since the user's actions are not known a priori. Instead, the camera must adapt in real-time to the movements of the user.
2. **Collision-free.** As the camera moves through a scene, it must not collide with or pass through objects in the environment as this lends an unrealistic feel to the camera movement and detracts from the user's sense of immersion.
3. **Smooth.** Teleporting the viewpoint from one place to another may disorient the user since the continuity of the view is broken. Consequentially, camera movement should be smooth in both space and time.
4. **Visible.** Ultimately, the camera's goal is to look at something. Thus, visibility is of utmost importance: the player or other focus target must be kept in view and unobstructed.

1.1. Local Camera Control

Classical models control the camera using algorithms that are inherently local in nature. Small adjustments in position and orientation are made based on objects in the camera's immediate vicinity. Such local camera control is effective in some situations, such as a camera mounted at a

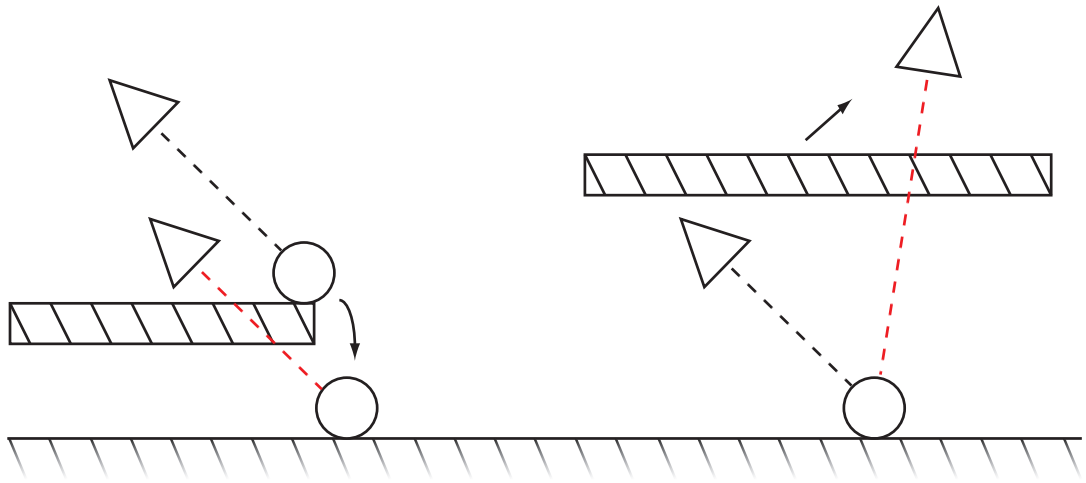


Figure 1.1.: This figure shows two typically difficult situations for local camera control. On the left, the camera follows a character that jumps down a ledge. On the right, the camera viewpoint is changed from a third-person view to a top-down view.

certain angle and small distance behind a virtual character. Moreover, these approaches usually integrate a high controllability of certain properties. Depending on the camera constraints, the user is able change the view direction or angle of the camera to his or her needs.
However, the local control of the camera's configuration is limited. Especially in applications where the camera is required to follow an avatar. If the character dashes quickly behind corners, the locality assumptions can be broken and the camera may be forced to pass through an object or teleport to an unobstructed view.

Figure 1.1 shows two situations in which local methods can fail. The main problem is the limited information that is used per frame to find an alternative camera position when the view is obstructed. If the camera is required to follow a fast paced character the local information can change very quickly. Local methods need to break one or more of the four defined properties in order to reattain a clear view to the character.

More notably, local camera models are, intrinsically, not able to perform large scale transitions between viewpoints, such as transitioning from a third-person view to a top-down view. Current systems rely on simple heuristics for such transitions, and lack global visibility information.
Several sophisticated local methods have been developed that can very robustly guide the camera through a virtual environment. The locality of the solution, however, can be exploited to construct cases where these methods will break either the property of smoothness or the avoidance of penetration.

1.2. Global Camera Control

This research complements local camera models and alleviates the above described problems by focusing on large-scale camera transitions. Collision-free movement through an arbitrary

three-dimensional environment elevates the problem to the category of path planning. However, existing path planning algorithms do not incorporate any notion of visibility into the computed paths. Thus, we define the new problem of visibility transition planning in which a smooth, collision-free transition between arbitrary start and end points is computed, while maximizing the visibility of a third focus point. Such a transition may deviate significantly from the shortest path in order to ensure that the focus point is maximally visible. A typical application might be the switch from a third-person view of an avatar in a virtual city environment to an overhead view while ensuring a clear view of the avatar throughout the camera motion. Maintaining unbroken focus on the avatar would help the user better understand the position and context of the avatar within the greater scope of the city.
Visibility computations in arbitrary three-dimensional environments are notoriously complicated and time-consuming. The difficulty of this problem is exacerbated by several factors. The global nature of the transition planning problem means that a potentially huge number of visibility evaluations must be considered. Neither the start point, nor the end point, nor the focus point are know in advance, making pre-computation a non-trivial task. Finally, the algorithm has strict, real-time requirements, with only milliseconds available for all computations.

We present an algorithm for visibility transition planning that achieves near-optimal results in real-time for arbitrary environments. This achievement rests on several key insights. We develop a visibility roadmap data structure that allows the pre-computation of a coarse representation of all collision-free paths through an environment, together with an estimate of the pair-wise visibility between all portions of the environment. Once the start, end, and focus point have been specified, our runtime system executes a path planning algorithm using the precomputed roadmap values to find a coarse path that is optimal in terms of visibility up to the resolution of the roadmap. Next, exact visibility is determined by computing a sequence of GPU-assisted occlusion maps along the coarse path.
The same path-planning code is executed with these occlusion maps to ensure optimal visibility on a fine scale. An iterative smoothing algorithm together with a physically-based camera model ensures that the path followed by the camera is smooth in both space and time. All run-time computation is output sensitive, so that the required time depends on the final path length.

1.3. Overview

Chapter 2 discusses previous and related work. An implementation approach for a physical camera module is discussed in chapter 3. This module is designed to handle constraints for different camera styles interchangeably and is the basis of our dynamic camera control. The pre-computations for our global approach are discussed in chapter 4. This part explains how a three dimensional roadmap can be created that includes visibility information that can be queried at runtime for an arbitrary focus point. Chapter 5 discusses visibility transition planning on the pre-computed roadmap. A notion of an optimal visibility path is given and we show how classical search algorithms can be modified in order to deliver paths that satisfy this condition. The path post-processing is discussed in depth in chapter 6. Since the visibility path planning delivers an initial guess of the optimal transition several optimizations can be applied to increase both the optimality and smoothness of the path. Finally, chapter 7 shows the dy-

namic implementation of the camera controller that builds up on the physical camera module introduced in chapter 3. We show how the visibility transition planning can be moved to a second CPU core in order to dynamically update a camera path at runtime. Results of this research are presented in chapter 8. The implementation of visibility transition planning is presented in an urban environment. We demonstrate that more intuitive navigation is possible using our approach. Results of the dynamic camera controller are presented in an environment that contains structures that are typically difficult to solve for local methods. The dynamic controller is able to solve problematic situations more gracefully than classical methods. Chapter 9 concludes this thesis with a discussion of the limitations of our approach, possible solutions, and areas of future work.

2

Related Work

2.1. Camera Models and Control

Classical camera models for computer games or other interactive applications are inherently local in nature. Often, simple heuristics implement a third-person camera that follows an avatar, making adjustments to position and orientation based on objects in the local vicinity in order to resolve occlusions. A common camera model used in computer games relies on a ray-cast from the camera position to the focus target. Occlusions are resolved by teleporting the camera to the ray intersection closest to the focus point, leading to a noticeable jump in view.
Halper, Halbing, and Strothotte [HHS01] present a more sophisticated local camera model that resolves occlusions more gracefully using occlusion maps and predictive camera planning. They emphasize the fact that camera work in games is quite different than movies, since a game camera must react in realtime, and scenes can not be reshot. They present a camera engine that consists of a director module and a predictive camera planner. They also develop a constraint solver that is able to adapt to or neglect some constraints, because in some situations, the computed position may not be desired. They use occlusion maps with a resolution of 32x32 pixels in order to find an alternative position for the camera. This leads to more consistent camera motion than one based on simple heuristics like the ray-cast method. Marchand and Courty [MC02] develop an image-based camera controller that uses visual servoing to resolve visibility tasks and occlusion constraints. It consists of positioning a camera according to the information perceived in the image. To be able to react automatically to modifications of the environment, they also consider the introduction of constraints into the control.

More high-level approaches to camera control focus on virtual cinematography, in which cinematographic knowledge is incorporated into the choice of camera position and scene composition. Bares and Lester [BGL98] present a constraint-based camera planner for shot composition

that models different cinematographic styles. He, Cohen, and Salesin [wHCS96] encode filmmaking heuristics into a hierarchical finite state machine that controls camera selection and placement. Hornung, Lakemeyer, and Trogemann [HLT03] develop a camera agent for interactive narrative applications. Tomlinson, Blumberg, and Nain [TBN00] develop a behavior-based cinematography system that controls both camera and lighting in order to convey a sense of emotion, while Kennedy and Mercer [KM02] focus on conveying theme and mood.
Halper and Masuch [HM] introduces methods for the automatic extraction and evaluation of action scenes in computer games. They present selection strategies that allow the automatic generation of summaries after the game which are shown as a sequence of images. Their system can also provide timing information to a camera for live specatator-mode viewing of the action.
Kahn and colleagues [KKS+05] present a new interaction techniquue, called HoverCam, for navigating around 3D objects at close proximity. When a user is closely inspecting an object the camera motions needed to move across its surface can become complex. To keep the camera a small distance above the surface, their camera module intelligently integrates tumbling, panning, and zooming controls into a single operation. Their main focus is to provide a camera with intuitive controls so that the user is not distracted from his or her work.

Drucker and Zeltzer [DZ94] present a framework for camera control in virtual environments. They create a system for handling a number of cameras to enable different cinematic shots in a scene, which builds upon their previous work [Dru, DGZ92]. They describe a system for specifying behaviors in terms of up to 7 degrees of freedom. The essential part of this work is the path planning that mainly was drawn from work in robotics. The system is implemented as camera control within a virtual museum and handles mainly the movement of the camera through different rooms. Their approach focusses on creating camera paths for off-line animations.

2.2. Path Planning

The problem of visibility transition planning is closely related to motion planning in which optimal paths are found through continuous space [LaV]. Our work is motivated by the interactive navigation system of Salomon and colleagues [SGLM03]. The problem this work solves is to guide an avatar hover from a given start position to a given destination. Legal paths are constrained by geometrical properties so that paths only may lie on geometry whose normal vector does not deviate more than a certain threshold from the up vector. Additionally, a legal path is defined by the property that the avatar can travel along it without colliding with its surroundings. Salomon and colleagues propose a solution using a precomputed road map, which is used to compute paths at runtime. The roadmap is computed by randomly sampling avatar positions in the environment and computing neighboring positions by testing a virtual walk for collisions. A pruning step after the sampling computations prunes away redundant nodes to simplify the road map. At runtime, both start and destination positions are added to the roadmap, and a best-first search algorithm [DP85] finds a legal path along which the avatar can walk to reach the goal position.

Niederberger and colleagues [NRG04] develop an algorithm based on a modified A* search that yields robustly yields short paths on a triangulated height field. Their approach constructs

in a pre-computation step a roadmap that connects triangles on which an avatar can walk. The nodes of the roadmap are represented by triangular areas. This data structure is used to find legal avatar paths around objects at runtime. For shortest path computations, the well-known A* algorithm [DP85] is widely used. For our visibility planning we also make use of the A* search, but introduce a penalty term for the visibility of the target. Previous work in this area focuses on two-dimensional problems or height fields, but nevertheless provides a good basis for our three-dimensional approach. A popular algorithm for 2D path planning in dynamic environments is the radar path planner by Roth and colleagues [RWHK97]. It computes a path based on a wavefront propagated from the target point through the scene towards the source location.

2.3. Visibility

A key aspect of our camera system is the focus on optimal visibility during large camera transitions. Visibility problems are fundamental to graphics, robotics, vision, computational geometry, and other areas. As shown by surveys on the topic ([COCS00, Bit02]), many visibility algorithms strive to identify which objects, lights, or other portions of a scene are visible from a given vantage.

Zhang and co-workers [ZMHH97] introduce hierarchical occlusion maps for visibility culling on complex models with high depth complexity. The culling algorithm uses an object-space bounding volume hierarchy and a hierarchy of image-space occlusion maps. Occlusion maps represent the aggregate of projections of the occluders onto the image plane. For each frame, the algorithm selects a small set of objects from the model as occluders and renders them to form an initial occlusion map, from which a hierarchy of occlusion maps is built. The occlusion maps are used to cull away a portion of the model not visible from the current viewpoint. For models with high depth complexity, the algorithm achieves a remarkable speed up by culling a great portion of occluded or invisible polygons. Although the algorithm was designed to cull occluded parts of complex models it can be transformed to be of use for visibility tests. The occlusion maps can be used as an indicator to decide if an object is occluded from a certain point of view.

Durand, Drettakis, Thollot and Puech [DDTP00] presente a visibility preprocessing method that computes potentially visible geometry for volumetric viewing cells. They introduce novel extended projection operators, which permit efficient and conservative occlusion culling with respect to all viewpoints within a cell. The method uses the extended projection of occluders onto a set of projection planes to create extended occlusion maps. An important advantage of the approach is that extended projections can be re-projected onto a series of projection planes, and accumulate occlusion information from multiple blockers. This approach allows the creation of occlusion maps for hard-to-treat scenes such as leaves of trees in a forest and thus works for almost arbitrarily complex environments.

Wonka, Wimmer and Sillion [WWS01] present an online occlusion culling system which computes visibility in parallel to the rendering pipeline. They show how to use point visibility algorithms to quickly calculate a tight potentially visible set which is valid for several frames by shrinking the occluders used in visibility calculations by an adequate amount. These visibility

calculations can be performed on a visibility server, possibly a distinct computer communicating with the display host over a local network. The resulting system combines the advantages of online visibility processing and region-based visibility calculations, allowing asynchronous processing of visibility and display perations.

3

Virtual Camera

A virtual camera in three dimensional space can be seen as an autonomous agent that moves in a constrained pattern. Depending on the application, these constraints may directly affect the camera or constrain its position and orientation with respect to an object or moving character. The low-level behavior of the camera is of special importance. If the basic movement of the camera is unnatural, a user can get confused. Such unpleasing behavior is introduced by the following points:

1. **High-frequency movement.** A camera that jumps or oscillates is, in most cases, undesired. Instantly teleporting to a different location can confuse the user. Also a camera that seems to change its velocity instantly can give an artificial impression. Such patterns are not possible with real cameras and therefore seem unrealistic in virtual environments as well.

2. **Penetration of geometry.** Real cameras cannot move through walls. In a virtual world, however, they can. Even if the movement is smooth, this lends an unrealistic feel and can confuse the user. Therefore, special care needs to be taken to avoid the camera colliding with or passing through objects in a scene.

The first point, high-frequency movement, is introduced by discontinuous constraints on the camera position. This movement can be either a sharp kink in the camera path or even a large gap. There basically are two possibilities to resolve this problem. On one hand, the constraints that guide the camera need to be frame coherent. Figure 3.1 shows three examples of simple constraints. While this approach provides full control of the camera it is very difficult to achieve in practice if several hard and soft constraints need to be observed. On the other hand, the camera itself can, on top of the constraint layer, filter out discontinuities using a physical model. This approach is easier to achieve since it is completely independent of the underlying constraints. Another advantage is that the camera always moves in the same fashion regardless

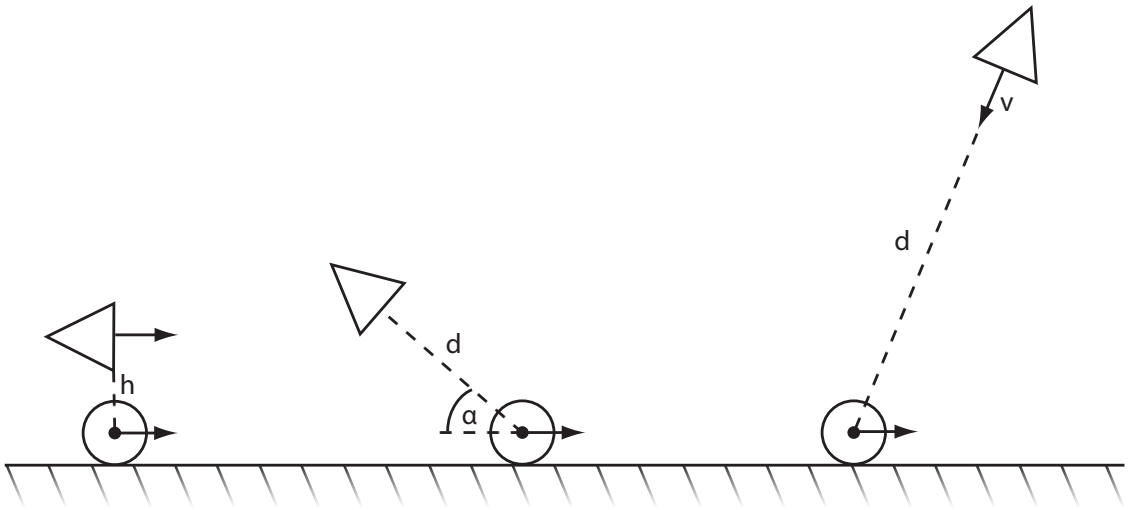


Figure 3.1.: This figure shows three examples of simple camera constraints. On the left, a first-person camera is shown. It is constrained to the player by an offset height h. The example in the middle shows a third-person view. The camera depends on a distance d and an angle α with respect to the player's position and orientation. A top-down view is shown on the right. It is constrained by a view direction v and a distance d from the player.

of different sets of constraints. This enforces the camera's authenticity.
The second point, avoiding penetrations and collisions, is a more difficult problem. A simple approach would be to use a bounding structure around the camera that is used to test for collisions. If the camera touches its surrounding geometry, a penalty force along the normal of the intersected triangles could be applied in order to resolve the situation. This approach would avoid penetrations of the geometry by the camera. Additionally, however, the underlying constraints might be violated. This can lead to the camera getting stuck at a wall when the penalty force and the constraints direct it into two opposite directions. Therefore, the avoidance of penetrating through or colliding with objects needs to be handled on the constraint level. Since the complete avoidance of collisions and penetrations is, in fact, a problem that cannot be solved entirely by local constraints, a global approach is inevitable. The camera constraints can, however, be chosen carefully so that the camera avoids its surrounding objects as good as possible.

3.1. Basic Camera Controller

Camera constraints can either directly affect the camera configuration or indirectly constrain it. The only difference is that indirect constraints depend on external parameters like the position and orientation of a character. Wooten [Woo] presents a model that is able to simulate human behavior based on different sets of constraints. The setup in his work is very similar to controlling a virtual camera. Figure 3.3 shows our implementation of a basic camera controller inspired by Wooten's human control system. The idea is to have different control routines that

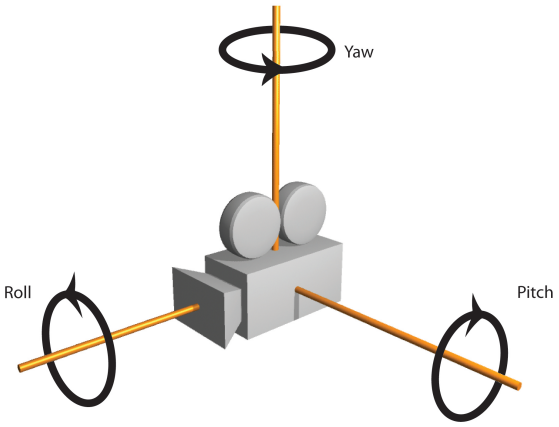


Figure 3.2.: The camera orientation can be described by the values yaw, pitch, and roll. They describe three elemental rotations about each one of the principal axis of the camera.

depend on input parameters like the current camera configuration, character position, and orientation. The different control routines each implement a different set of constraints that define a desired camera style. They also provide an interface to query their desired camera configuration based on the input. The basic camera controller can use these control routines interchangeably. Depending on the current camera mode, the controller uses the appropriate control routine to get a desired camera position and orientation. Using this desired configuration, the camera's current configuration can be driven towards the constrained configuration by numerical integration. This approach allows the implementation of different camera styles in one module without changing the camera behavior.

3.2. Physical Camera Model

A camera in three dimensional space can be uniquely defined by a vector of six degrees of freedom (DoF). The orientation of the camera can be expressed by the three DoFs yaw, pitch and roll. The yaw-pitch-roll system is a hierarchical formulation of angles that allows all orientations in space to be split up into three rotations around the local object axis (See figure 3.2). This description of an oriented object in space allows intuitive treatment of the three angles as these parameters directly correlate with the local coordinate system of the camera object. The other three DoFs are given by the camera's position in space.

$$\boldsymbol{c} = (x, y, z, \alpha, \beta, \gamma)^T \tag{3.1}$$

The goal is, to move the camera in a realistic way. This can be achieved by regarding the camera object as a rigid body. By physically simulating the camera in the six-dimensional space of its DoFs, a natural movement can be achieved. To properly describe the camerera's physical behaviour, additionally, a defined mass m and friction f for each independent DoFs are required.

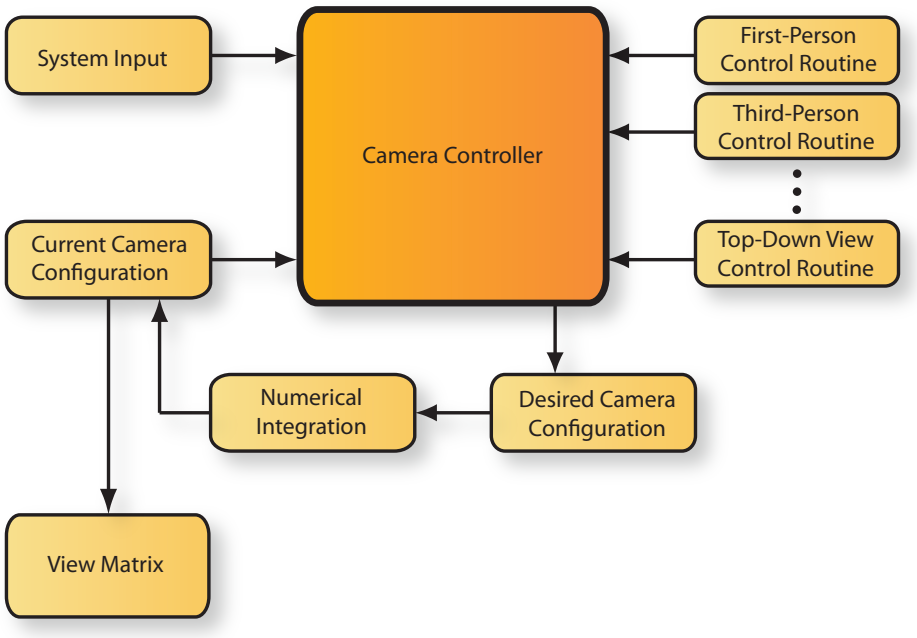


Figure 3.3.: This diagram shows our implementation of a physical camera module. The basic camera controller holds a list of control routines that each implement a different set of constraints for the camera. The controller queries a desired camera configuration from the active control routine. Using numerical integration, the current camera configuration can be driven physically towards the desired configuration.

These parameters allow the camera to be viewed as a free-flying object. In order to move the object, forces are to be applied to the individual DoFs. As shown in figure 3.3, the camera controller delivers a desired configuration $\boldsymbol{c_d}$ for each frame. By physically maneuvering the current camera configuration $\boldsymbol{c_c}$ towards $\boldsymbol{c_d}$, its movement is automatically dampened, removing high-frequency motions.

3.3. Camera Simulation Loop

The low-level camera control system can be interpreted as a simulation cycle which can easily be integrated into the main game loop. In the camera simulation loop, the camera has a configuration $\boldsymbol{c_c} = (x_c, y_c, z_c, \alpha_c, \beta_c, \gamma_c)^T$ that defines the view transformation matrix for the current frame. The current camera state and the system input (like user controls, avatar position, etc.) are passed to the active control routine which computes the desired camera configuration $\boldsymbol{c_d} = (x_d, y_d, z_d, \alpha_d, \beta_d, \gamma_d)^T$.

By planting forces on and numerically integrating $\boldsymbol{c_c}$, the camera is driven in a physically correct way towards its constrained configuration. A simple but effective way to compute forces between $\boldsymbol{c_c}$ and $\boldsymbol{c_d}$ is to use forward dynamics as described by Brogan and his colleagues [BMH98]. Basically, a spring system can be imagined between the individual entries of $\boldsymbol{c_c}$ and $\boldsymbol{c_d}$.

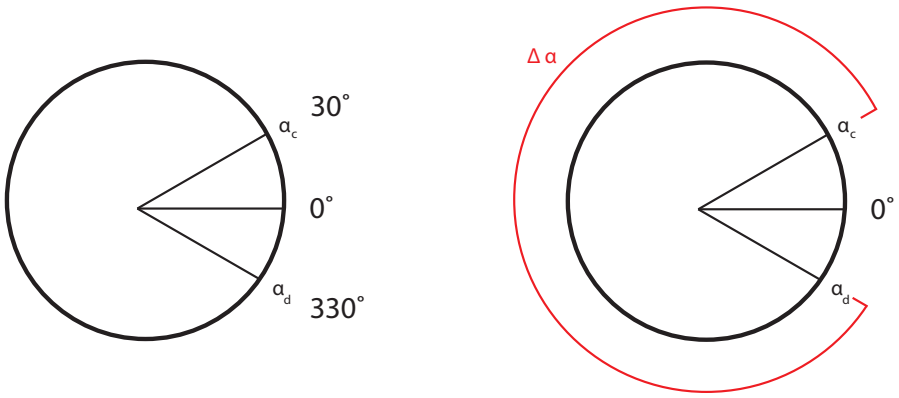


Figure 3.4.: If the difference between angles is computed straight forward there is always a configuration that results in a wrong distance. In order to solve this problem a case differentiation is required.

$$a = \frac{k^s \Delta e}{m} \tag{3.2}$$

Equation 3.2 shows the general spring equation. The acceleration a depends on the spring constant k_s and the expansion lenght Δe. The parameter Δe corresponds to the difference between the individual DoFs of $\boldsymbol{c_c}$ and $\boldsymbol{c_d}$. The difference Δx, Δy, and Δz of the positional parameters x_c, y_c, and z_c and x_d, y_d, and z_d, respectively, are computed by a simple subtraction as shown in the equations 3.3, 3.4, and 3.5.

$$\Delta x = x_d - x_c \tag{3.3}$$
$$\Delta y = y_d - y_c \tag{3.4}$$
$$\Delta z = z_d - z_c \tag{3.5}$$

The differences $\Delta\alpha$, $\Delta\beta$, and $\Delta\gamma$ between the orientational entries of the current and desired configurations require separate treatment. Since the angular values are defined in a periodic space, simple subtraction may lead to undesired camera turns. If, for example, the current yaw $\alpha_c = 30$ and the desired yaw $\alpha_d = 330$, a subtraction of α_c from α_d would lead to a turn around the longer side. Figure 3.4 illustrates this example. In order to get a distance value that defines a turn around the shorter segment, a case differentiation is required. The follwing equation covers all cases and leads to the correct difference angle $\Delta\alpha$. The computations of the difference values $\Delta\beta$ and $\Delta\gamma$ are analogous.

$$\Delta\alpha = \begin{cases} \alpha_d - \alpha_c - 360 & \text{if } \alpha_d - \alpha_c > 180 \\ \alpha_d - \alpha_c + 360 & \text{if } \alpha_d - \alpha_c < -180 \\ \alpha_d - \alpha_c & \text{otherwise} \end{cases}$$

The individual difference parameters computed using the corresponding equations above lead to the difference configuration $\Delta \boldsymbol{c} = (\Delta x, \Delta y, \Delta z, \Delta \alpha, \Delta \beta, \Delta \gamma)^T$. This vector can now be inserted into equation 3.2 in order to compute the acceleration vector $\boldsymbol{a}$.

$$\boldsymbol{a} = K^{s/m} \Delta \boldsymbol{c} \tag{3.6}$$

In the above equation, the spring constant k^s is also replaced by the spring matrix $K^{s/m}$. This parameter is the diagonal matrix of the individual spring constants for the six DoFs of the camera configuration. Also, the mass m of the camera is incorporated into this matrix.

$$K^{s/m} = \begin{bmatrix} \frac{k_x}{m} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{k_y}{m} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{k_z}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{k_\alpha}{m} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{k_\beta}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{k_\gamma}{m} \end{bmatrix} \tag{3.7}$$

When using the acceleration $\boldsymbol{a}$ computed with the equation 3.6, the camera is never slowed down once it gets in motion. This lack will eventually lead to undesired oscillations of the camera around $\boldsymbol{c_d}$. To avoid oscillations, a friction component f must be added for each DoF. The spring matrix $K^{s/m}$ in equation 3.6 only contains constant values. Therefore, this matrix can easily be extendet with a friction component for each DoF and leads to the camera parameter matrix P.

$$P = \begin{bmatrix} \frac{k_x(1-f_x)}{m} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{k_y(1-f_y)}{m} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{k_z(1-f_z)}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{k_\alpha(1-f_\alpha)}{m} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{k_\beta(1-f_\beta)}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{k_\gamma(1-f_\gamma)}{m} \end{bmatrix} \tag{3.8}$$

To incorporate P, equation 3.6 is rewritten to compute the dampened acceleration vector $\hat{\boldsymbol{a}}$:

$$\hat{\boldsymbol{a}} = P \Delta \boldsymbol{c}. \tag{3.9}$$

3.4. Numerical Integration

Once the appropriate acceleration vector $\hat{\boldsymbol{a}}$ is known, the last step of the camera simulation loop can be processed: numerical integration. For this purpose, a stable and fast integration method is preferred. The explicit leapfrog scheme provides excellent stability while remaining simple and fast.
The new camera configuration $\boldsymbol{c_c}(t+dt)$ can now be integrated from the old camera configuration $\boldsymbol{c_c}(t)$ using the previously calculated acceleration vector $\hat{\boldsymbol{a}}$ and the time step dt:

$$\boldsymbol{v}\left(t+\frac{dt}{2}\right) = \boldsymbol{v}\left(t-\frac{dt}{2}\right)+dt\hat{\boldsymbol{a}}(t) \tag{3.10}$$

$$\boldsymbol{c_c}(t+dt) = \boldsymbol{c_c}(t)+dt\boldsymbol{v}\left(t+\frac{dt}{2}\right). \tag{3.11}$$

The following boundary conditions are used to start the simulation:

$$\boldsymbol{v}\left(0-\frac{dt}{2}\right) = (0,0,0,0,0,0)-\frac{dt}{2}\hat{\boldsymbol{a}}(0) \tag{3.12}$$

$$\boldsymbol{c_c}(0) = (x_0,y_0,z_0,\alpha_0,\beta_0,\gamma_0). \tag{3.13}$$

4

Construction of the Roadmap

We develop a camera model that guarantees smooth, collision-free movement by phrasing the problem as a global search for an appropriate camera path between arbitrary start and end points in space. To be able to compute such global transitions in an efficient way at runtime requires discretization of the free space in the environment. As the environment can be both large and complex, this discretization requires heavy computations. These data structures can be created in a pre-computation step. This permits a fast on-line algorithm that can create camera transitions in real-time.
Furthermore, since it is the most fundamental task of the camera to look at something, visibility information is very important. A camera transition therefore should not only be a path from a start to an end position. Visibility of a focus point should be integrated as well.

This chapter discusses the creation of a roadmap data structure that captures free space in the environment using convex bounding volumes. The computation of the convex objects is explained in detail in the first part. To incorporate visibility into the data structure, a second pre-computation phase is discussed.

4.1. Roadmap Creation

Representing free space is a similar problem to representing obstacles. Niederberger and colleagues [NRG04] develop a method to triangulate parts of a height field that can be traveled by an avatar. They propose building a graph structure based on these triangles. The nodes of the graph represent the area of the corresponding polygon. This idea can be extended to the third dimension. By discretizing parts of the continuous space using convex objects, a good approximation can be achieved. These objects then represent a part of the free space and are assigned to

nodes in the graph. There exist a variety of different approaches to discretize three dimensional space. The most common solutions use bounding boxes. Structures like BSP-Trees or Oct-trees have proven to be very useful for capturing objects and simplifying collision detection. Since our task is to discretize free space by a graph a hierarchical subdivision is not required.
The right choice of a convex volume is very important at this point. Since the post-processing of visibility transitions will directly operate on these objects, the simplicity of operations plays an eminent roll. The simpler these constructions are the faster the runtime algorithm will be. Although the whole visibility planning is possible for any kind of convex hull we choose bounding spheres as basic objects. Spheres are, on one hand, very simple objects and allow for fast intersection tests. On the other hand, spheres have a homogeneous expansion in all directions. This property allows an easy and accurate analysis of the discretization error.

In a first pre-computation step the graph is built that acts as a roadmap that the camera is able to travel through. As explained above, spheres have several advantages over other convex objects. Their simplicity allows building a roadmap graph whose nodes overlap. This permits a very dense representation of free space and provides excellent adaptivity in narrow regions.

Neighboring spheres are defined as two spheres that overlap. The overlap region of these spheres, bounded by an overlap circle, represents the space that is allowed to travel from a point in the first sphere to a point in the second sphere. By restricting spheres not to contain any centers of other spheres an overly dense sampling automatically is avoided. Furthermore, this property allows certain assumptions and simplifies case differentiation during the path post-processing.

4.1.1. Spheres Sampling

Computing a collection of spheres that covers free space as good as possible while being optimized for overlap regions is a non-trivial task. Especially if the number of spheres needs to be as small as possible in order to keep the size of the resulting roadmap graph small. Since the operations on the roadmap will be output sensitive, its complexity directly correlates to the computational cost of the visibility transition planning explained in chapter 5.

The following algorithm explains our approach to create a collection of spheres that adaptively and uniformly sample free space. First, a large number of possible candidate spheres is created. Each one of these spheres is positioned such that it does not contain any geometry of the environment. Then, in a second step, the number of spheres is reduced by computing a score function for all spheres. The sphere with the highest score is selected sequentially until the free space is fully sampled. Figure 4.1 illustrates the individual steps of the algorithm.

```
0. Initial environment.
1. Embed the environment into a three-dimensional grid
   with resolution delta_x that resolves all important
   features.
2. Mark all cells that intersect geometry occupied.
```

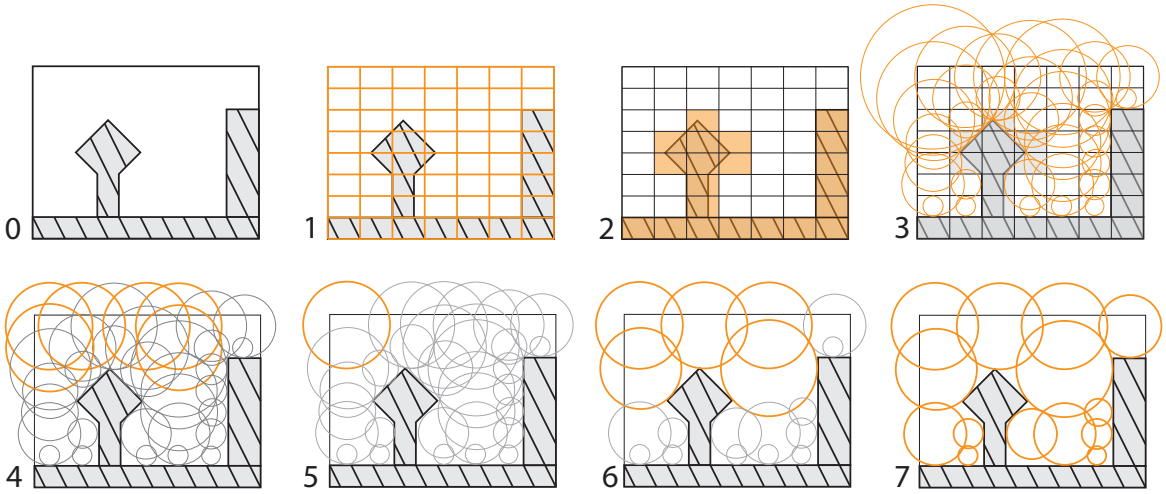


Figure 4.1.: This figure illustrates the individual steps of our sphere sampling algorithm. Based on a grid, a large number of spheres are initially generated. Using a score function, the number of spheres is vastly reduced to an adaptively samplted set.

```
3. For all free cells, compute the largest possible
   sphere s = (center, radius) that fits into the
   environment without penetrating the geometry.
4. For each cell that is not occupied:
   If s.radius < s_min then mark cell occupied.
   If s.radius > s_max then set s.radius = s_max.
5. Initialize an empty list of spheres L. Find the cell
   with the largest sphere s and insert s in L.
   Mark all cells whose sphere contains s.center or whose
   center is contained in s occupied.
6. Find in all cells that are not occupied the sphere s
   that has the highest score S(s, L). Insert s in L and
   mark all cells whose sphere contains s.center or whose
   center is contained in s occupied.
7. Repeat step 6 until the maximum number of spheres is
   reached or all cells are marked occupied.
```

First, the environment is embedded into a three-dimensional grid. Each cell center represents a set of discretized positions inside the level in which a sphere center can lie. The density of the grid only influences the positions where sphere centers can be set. If the cell spacing is too coarse for narrow hallways, then no spheres will be generated there. On the other hand, if an ovelray is too dense, the computation time increases dramatically. To avoid a to dense grid the number of cells for each dimension can be made dependent on the scale of the environment bounding box. Virtual maps are often much wider than high, and therefore a reduction of cells in the y-direction results in a significant speed-up without compromising the accuracy of the sampling. If the grid is restricted by the environment's bounding box, space at the top of an

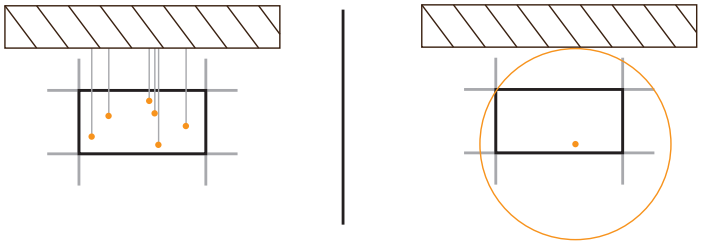


Figure 4.2.: This figure illustrates the sphere center multi-sampling. A series of random positions inside the cell boundaries are generated. By fitting a sphere into the free space for each of these random centers, the best sphere can be found.

outdoor level can be cut off. In such cases, a safety buffer can be introduced that extends the grid in the y-direction by a margin.

The second step marks all cells that reside inside or intersect with geometry as occupied. These cells represent possible sphere positions that are not relevant for the sphere sampling. By removing these cells at the very beginning of the sampling a significant computational overhead is already removed.

In step three, for every free cell a maximal sphere is computed whose center is within the cell boundaries. In order to fully use the space given by the cell a, multi-sampling of the sphere center can increase the overall sphere size. First, a number of potential sphere centers are uniformly sampled inside the given cell. Second, for each sampled position, the largest possible sphere that does not intersect geometry is computed. The overall largest sphere is then stored for this cell. Figure 4.2 illustrates an example of this multi-sampling approach.

Step four applies a pre-selection based on two threshold parameter, s_{min} and s_{max}. All unoccupied cells that contain a sphere with radius smaller than s_{min} are marked occupied. These spheres are regarded as too small and only increase the size of the data structures while contributing very little to the discretization. Therefore, they can already be removed at this stage. All spheres whose radius is greater than s_{max} are clamped to have a radius of size s_{max}. The discretization error of the roadmap is directly correlated with the size of the underlying spheres. The limitation of the sphere radius therefore also bounds the discretization error.

The fifth step of the algorithm initializes the actual sphere selection process. After creating an empty list L of spheres, a seed sphere s is required. This sphere is choosen as the largest one available and will most likely be one of the spheres that were clamped down to a radius of s_{max}. A seed sphere is needed because the scoring function $S(L, s')$ used for the sphere selections in step six requires L to be non-empty.
After s is selected and inserted into L, a series of cells must be removed. All cells that contain a sphere whose center is inside s or whose sphere contains $s.center$ are marked occupied. The following statement is true for all spheres s' that need to be removed:

$$Dist(s.center, s'.center) < Max(s.radius, s'.radius). \tag{4.1}$$

This test guarantees that, at the end of sphere creation, no sphere contains the center of another

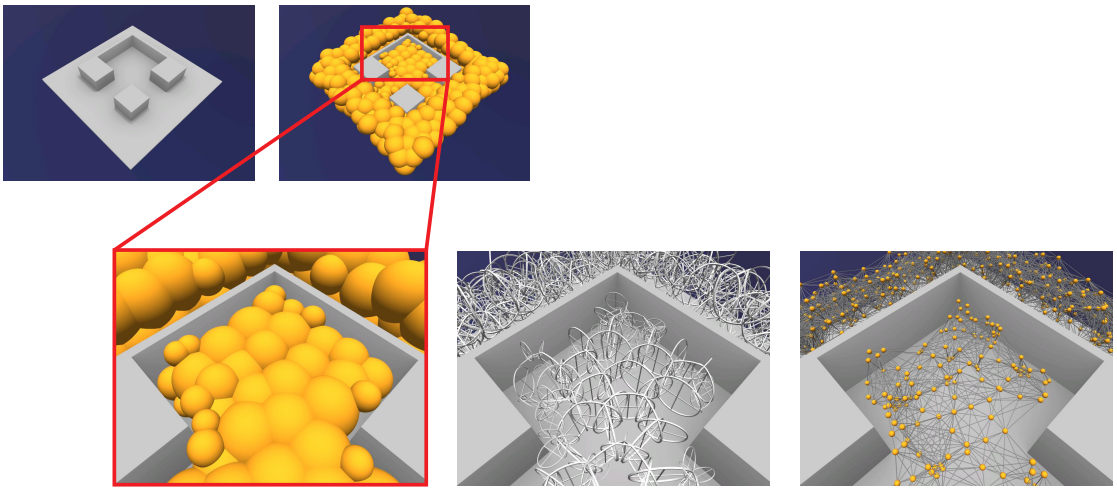


Figure 4.3.: This is a visualization of the individual steps of the roadmap construction from the sampled spheres. The image on the left shows the sampled spheres. All intersection circles of the spheres are shown in the image in the middle. On the right side, the resulting roadmap nodes and connections are shown.

sphere.

Step six is the actual score-based reduction of the available spheres. Given the list of all already selected spheres L, a score $S(L, s')$ for each sphere s' of all free cells is computed. The score function used in the implementation of this thesis optimizes for overlap areas with all spheres in L:

$$S(L, s') = \sum_{t \in L} OverlapArea(s', t). \tag{4.2}$$

The sphere s with the highest score is selected and inserted into L. Then, equivalent to the process in the previous step, all cells whose sphere returns true for equation 4.1 are marked as occupied.
By selecting new spheres based on the largest overall overlap region with all previous selected ones, the spheres are implicitly optimized for size. Larger spheres will potentially have larger overlap areas and therefore be selected more likely than small ones.
Appendix A.1 presents all steps required to compute the overlap area of two spheres.

Step six of the algorithm is now repeated until either all cells are marked occupied or a maximum number of spheres is reached. If, however, the algorithm is aborted before all cells are occupied, no guarantee can be made about the distribution of the spheres. A large region of free space may only be connected to the rest by a small window. Because the algorithm starts with a single sphere it needs to "grow" into the other areas.

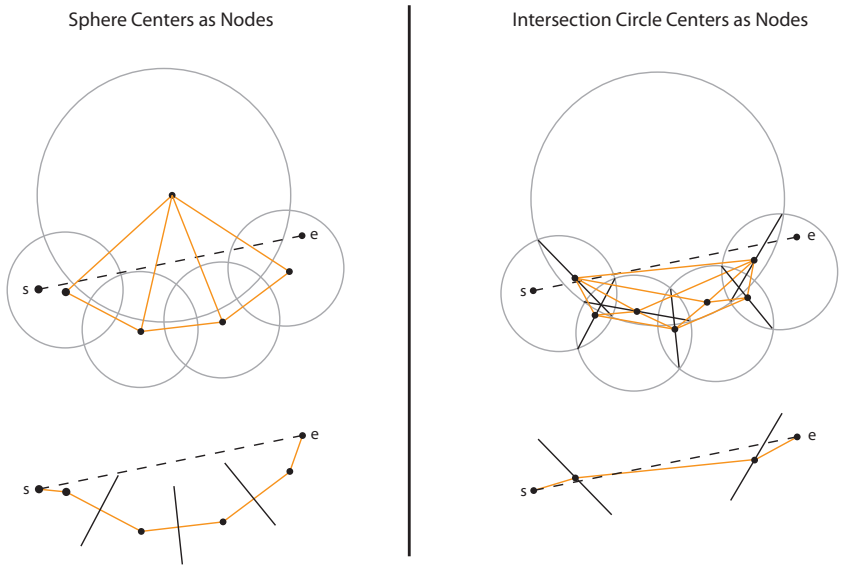


Figure 4.4.: (Left) The discretization of the roadmap nodes to a point in the center of the sphere introduces an indirection error when the path is required to pass through the sphere. As shown in this example, the path around the large sphere would be shorter than a path that includes an edge to the large sphere's center. (Right) By defining the roadmap nodes in the center of the intersection circles, a higher connectivity and a smaller discretization error is given.

4.1.2. Construction of the Roadmap using Spheres

Given the list of spheres L that optimally sample free space, the roadmap can be constructed. The task is to find positions inside the spheres for the graph nodes and a neighboring relation that defines the graph edges. A straightforward method would be to use the sphere centers as nodes and the intersection of spheres as a neighbor relation. This approach, however, introduces a potential error. When approximating flat structures (like a plane), large spheres can touch the surface. As a direct consequence of the adaptive sphere sampling, the room between the sphere and the plane can be sampled by an array of smaller spheres. Distances through the large sphere are penalized with an indirection to the sphere center. This effect leads to a high approximation error in such cases. Figure 4.4 illustrates this problem.

In order to avoid this inaccuracy the roadmap can be build differently. An alternative method to select roadmap nodes is to use the centers of the intersection circles between the spheres. A neighbor relation between two nodes exists when both nodes lie inside the same sphere. This approach to create a graph leads to a much higher connectivity of the roadmap.

4.2. Pre-computation of Visibility

Because we want to compute camera transitions between two positions that also consider visibility to a third position as a key issue, it is important to incorporate this property into the

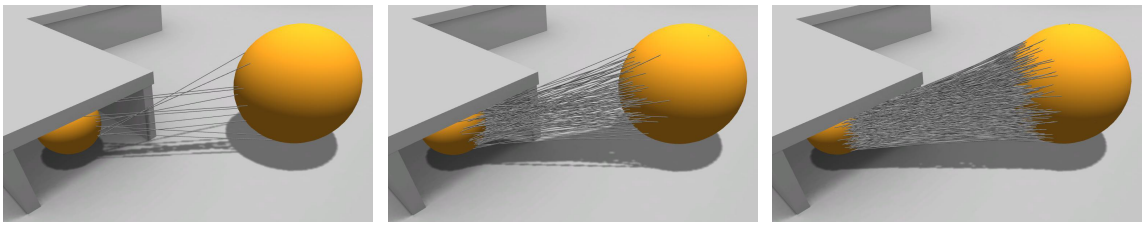


Figure 4.5.: To precompute the visibility probability between two random points in the source and target sphere, a Monte Carlo approach can be used. The approximative visibility probability is computed by dividing all intersection free rays by the total number of rays.

roadmap data structure. Optimizing a path for visibility to a focus point can have a large impact on the computed path. Chapter 5 explains the concept of visibility transition planning in depth and introduces a notion of a visibility optimal path.

A focus point may lie anywhere in the environment and its position is not known in advance. To create a data structure that allows fast queries for visibility at runtime, a pre-computation between all pairs of spheres in the roadmap is performed.

Visibility between two spheres is computed as a probability using a Monte Carlo approach. For each pair of spheres s_i and s_j, n rays are generated that start at a random position inside the first sphere and end at a random position inside the second sphere. The following pseudo code implements a uniform distributed random sampling of points inside a given sphere s with radius r and center c.

```
x = UniformRandom(0,1);
y = UniformRandom(0,1);
z = UniformRandom(0,1);

while (sqrt(x^2 + y^2 + z^2) > 1) {
    x = UniformRandom(0,1);
    y = UniformRandom(0,1);
    z = UniformRandom(0,1);
}

Vector3 v = new Vector3(x,y,z);
Vector3 randomPosInSphere = c + v * r;
```

The number of intersection-free rays between s_i and s_j is denoted by k_{ij}. The visibility probability $p(s_i, s_j)$ is computed by scaling the number of intersection free rays k_{ij} by the total number of rays n. Figure 4.5 shows an example of the visibility computation between two spheres for different sizes of n.

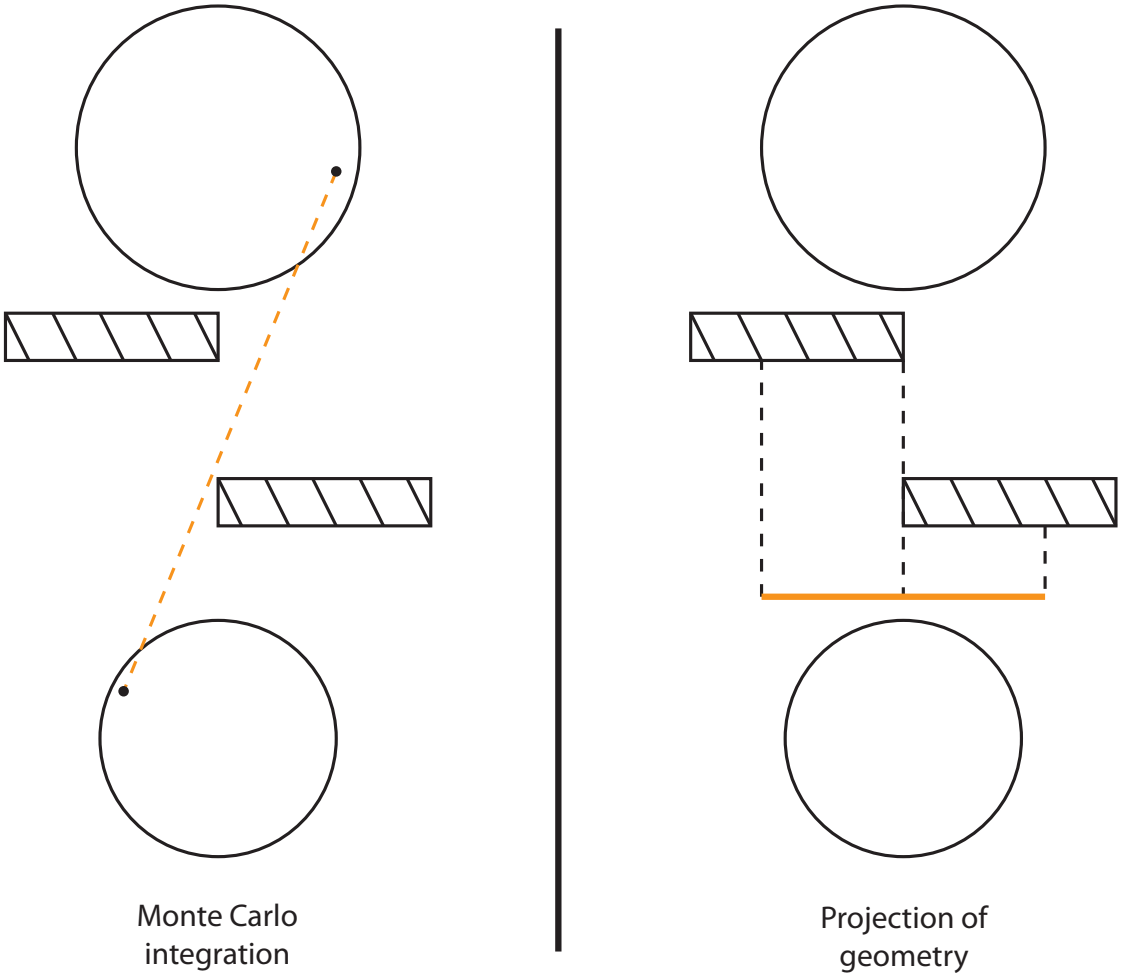


Figure 4.6.: This example illustrates the problem of using a projection approach to compute visibility between two spheres. Because of the nature of the projection, possible view lines are neglected. A Monte Carlo integration, in contrast, is able to capture nonparallel viewlines.

$$p(s_i, s_j) = p_{ij} = \frac{k_{ij}}{n} \tag{4.3}$$

An alternative to this Monte Carlo scheme would be based on projections. However, we have chosen a Monte Carlo method because of higher accuracy of the resulting probability values. The focus position f and the position t for which p_{ij} needs to be known may lie anywhere inside the two spheres s_i and s_j. This setup can introduce a nonparallel visibility line. As demonstrated in figure 4.6, a projection map only uses parallel lines and therefore neglects possible view lines.

4.2.1. Spatial Complexity of Visibility Data

Both the spatial and computational complexity of the visibility pre-computation is $O(nt^2)$, with t being the number of spheres of the roadmap and n being the number of sample rays. The memory usage can become large if, for each sphere s_i, the probability p_{ij} for all other spheres s_j is stored. In order to have fast query times at runtime, the list of visibility probabilities of a sphere to all other spheres can be stored in a hash table. This reduces the runtime acces to $O(1)$. A reduction in memory overhead can be achieved if probabilities $p_{ij} = 0$ are not stored. With increasing complexity of the environment, and therefore increased number of view-blocking objects, the space to store the probabilities is automatically reduced. By introducing a "fog" component to the visibility computation, the spatial complexity can be reduced to $O(knt)$ where k is a constant value. Given a maximal distance d_{vis} at which objects are visible, all probabilities p_{ij} can be set equal to zero when the distance of all points within the respective spheres s_i and s_j is larger than d_{vis}. If the following condition is true, all points in s_i are further away than d_{vis} from all points in s_j:

$$Distance(s_i.center, s_j.center) - (s_i.radius + s_j.radius) > d_{vis}. \tag{4.4}$$

5

Visibility Transition Planning

The previous chapter explained the construction of the three-dimensional roadmap that includes visibility. The information is given in two data structures. First, a roadmap contains a discretization of all paths in free space. This information is given in the form of a graph with Euclidean distances between the individual nodes. The second data structure is the list of all spheres that sample free space. Each of these spheres has a list of visibility probabilities of all other spheres. The nodes of the roadmap are defined on intersection circles between two spheres. Therefore, each node is associated with two spheres. Furthermore, edges of the roadmap are defined between nodes that lie in the same sphere.

This chapter addresses path planning on these data structures. A notion of optimal visibility transition is given that employs the pre-computed visibility information. This notion is used to modify classical shortest path search algorithms in order to determine paths optimized for visibility.

5.1. Graph with Visibility Region

The pre-computations described in the previous chapter deliver an undirected graph $G(V, E)$ with $|V|$ nodes and $O(|V|)$ connections between the nodes. Each node is associated with two spheres that represent free space in the environment. The distance between two neighboring nodes in the graph represents the true distance in Euclidean space. All graph edges $e \in E$ lie inside a sphere s_i. Therefore, for all e between two neighboring nodes, a list of visibility probabilities p_{ij} to all other spheres s_j in space is given.

Given a focus point $\boldsymbol{f}$, the graph G can be extended in run-time with the visibility information of all edges e by selecting the visibility probability p_{if} of the sphere s_f in which the focus

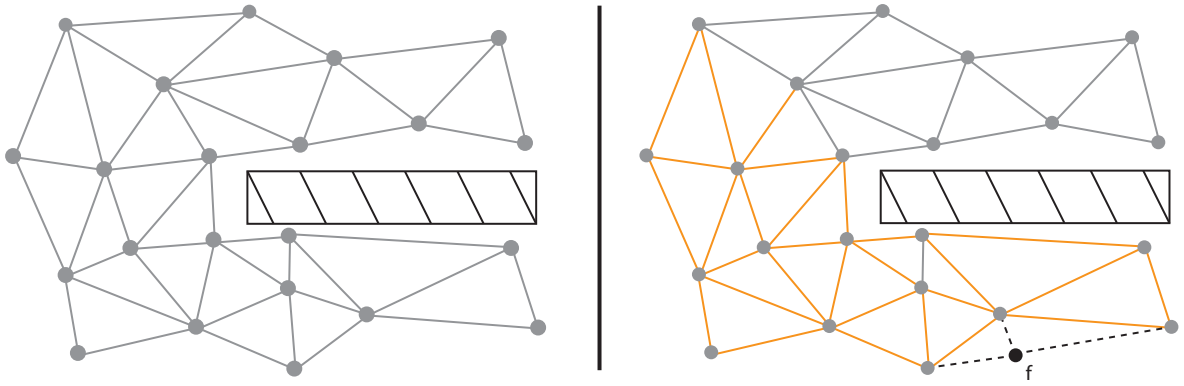


Figure 5.1.: (Left) The roadmap graph G. (Right) When a focus point f is inserted in the graph, the corresponding visibility probabilities of the edges in the G form a visibility region of $G_{f,v}$.

point lies. This application of the visibility probability creates a new graph $G_f(V, E)$ with an embedded subgraph $G_{f,v}(V_v, E_v)$ of nodes and connections with visibility to the focus point f. This subgraph is called the visibility region of f.
The visibility is a property of an edge e and denotes the probability that a random point on e is visible from f. This assumption results in a graph in three-dimensional space that has Euclidean distances between the nodes and an approximative visibility probability p_{if} defined on the edges. Figure 5.1 illustrates the graph and the visibility region.

5.2. Concept of Optimal Visibility Transitions

In general, there can be many paths between two nodes in the graph. Of these many paths, we are interested in the particular shortest path. However, the notion of shortest path depends heavily on the model of graph edge cost selected.
The euclidean distance is the simplest form of edge cost in this example. Because the graph nodes are indeed placed correctly in Euclidean space, the cost has a tight connection to standard shortest path planning. The second property of the given graph is the visibility probability $p_{i,f}$ of an edge i in sphere s_i to a given focus node $\boldsymbol{f}$. This additional information can be coupled with the Euclidean distance to create edge costs that change when the visibility to f changes. The way in which Euclidean distance and visibility are combined needs to be chosen carefully, however, because search algorithms have strict assumptions in order to guarantee optimal results.

The most important question is how to define an optimal path with respect to visibility. This decision influences the way Euclidean distance and visibility probabilities are coupled. The concept of a shortest path minimizes the whole path between a start and end position by distance. This, however, is not satisfactory in a general case if visibility is a key issue. The shortest path may lead to a transition that does not reach visibility until the very end. Figure 5.2 shows an example of such a case.

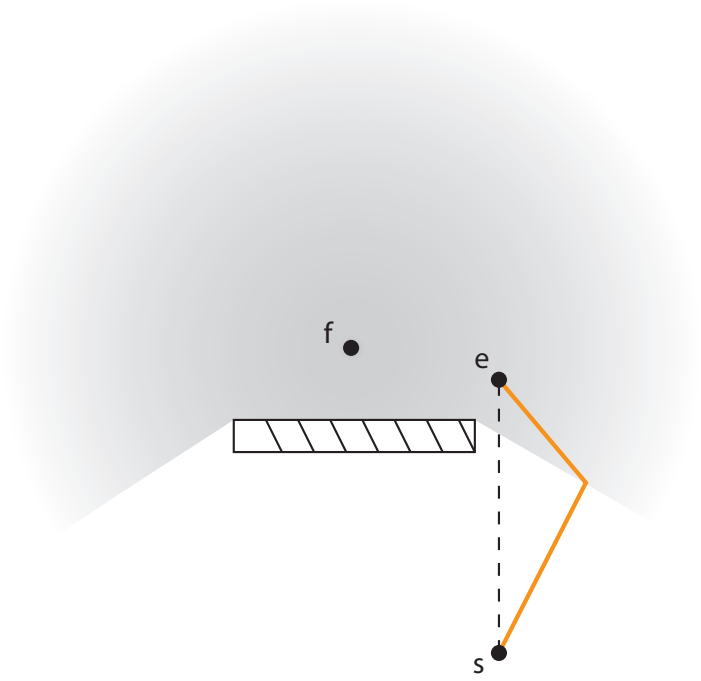


Figure 5.2.: The example configuration in this figure shows that a shortest path (dashed line) between a start position s and an end position e can be unsatisfactory in terms of visibility. The view to the focus point f is blocked until the very end. A path that "seeks" visibility (orange line) is more desirable.

A satisfactory camera transition that optimizes for visibility to a focus point, in contrast, should actively seek the visibility region. As soon as this region is reached the shortest path to the end point is chosen. This approach leads to a new notion of an optimal path.

Definition: An optimal visibility transition is the path between a start and end point that minimizes the distance for all parts of the path that lie outside the visibility region with higher priority than the positions inside the visibility region.

Several algorithmic approaches exist to find an optimal path. A shortest path algorithm, however, has several advantages over other kinds of algorithms in this problem setting. First, shortest path algorithms are guaranteed to deliver the shortest path given an underlying model. With a proper definition of edge costs, the search algorithm guarantees that there exists no path that has lower cost. Randomized algorithms, for example, cannot make this guarantee. Second, shortest path algorithms are well studied. There exist several approaches that are highly optimized in terms of computational speed. Fast computation is also a main concern as our approach should be interactive, and therefore as fast as possible.

Shortest path algorithms, however, require certain properties to be met by the underlying graph. Most importantly, shortest path algorithms require the edge costs to be greater than zero. There exist algorithms (cf. Bellman-Ford Algorithm [CG98]) that can handle single negative edges, but they cannot find a path when negative cycles exist in the graph. Because the graph represented by our roadmap is undirected, a negative weighted edge fro n_i to n_j also introduces a negative edge cost from n_j to n_i. This property automatically introduces a negative cycle. Therefore, our model needs to guarantee that edge weights are greater than zero. The following edge cost C between the two neighboring map nodes n_i and n_j holds this requirement:

$$C(n_i, n_j) = v(n_i, n_j) d_{eucl}(n_i, n_j) + \alpha(1 - v(n_i, n_j)) d_{eucl}(n_i, n_j), \tag{5.1}$$

where $v(n_i, n_j)$ denotes the visibility of the edge between n_i and n_j. If the shared sphere of n_i and n_j is s_k, then $v(n_i, n_j) = p_{k,f}$. The edge cost C is composed of two parts. The first part is the Euclidean distance between n_i and n_j multiplied by the edge visibility probability $v(n_i, n_j)$. The second part is the euclidean distance between the two nodes multiplied by the inverse visibility probability $(1 - v(n_i, n_j))$, weighted by a factor α.
The two parts of the edge cost C represent parts of the path inside the visibility region of f and outside of the visibility region, respectively. The first part can be considered as the part of the edge that lies inside $G_{f,v}$. The second part, weighted with α, can be considered as the part of the edge that lies outside $G_{f,v}$. If $v(n_i, n_j) = 1$, the edge would be fully inside the visibility region. For this case the edge cost C would simplify to the euclidean distance between n_i and n_j. If, in contrast, $v(n_i, n_j) = 0$ then the edge would be completely outside of the visibility region.

This redefinition of distance cost is now the overall cost to travel from n_i to n_j with α as a regulation parameter between visible and invisible parts. Using this cost function a path can be reformulated as being optimal with respect to visibility by adding a penalty for parts outside the visibility region. If $\alpha < 1$ the parts of a path that lie outside the visibility region are minimized with a much higher priority than the parts inside visibility. Since the edge cost C is greater or equal to zero, it can be used by a shortest path search algorithm.

5.3. Speedup Using General Best First Search

The subclass of generalized best first search algorithms [DP85] is prooven to be able to prune away large parts of the graph when finding a path between two nodes. For a path $\widetilde{P}$ that is only partially travelled, a heuristic $h(\widetilde{P})$ is used to estimate the total cost if it would be travelled to the end. Using this $h(\widetilde{P})$, the estimated best path can be traversed first until another path has a lower estimated cost. The estimated best path is defined as the traversed sub-path $\widetilde{P}$ with the lowest score $s(\widetilde{P})$.

$$s(\widetilde{P}) = F(C(\widetilde{P}), h(\widetilde{P})) \tag{5.2}$$

A well known subclass of generalized best first search strategies is A*. This algorithm was successfully used in several real-time applications like in the work of Niederberger, Radovic, and Gross [NRG04] and Salomon, Garber, Lin and Manocha [SGLM03]. The A* variant applies the following score function $F(\widetilde{P})$:

$$F(\widetilde{P}) = C(\widetilde{P}) + h(\widetilde{P}) \tag{5.3}$$

To be able to apply the A* search algorithm for the task of visibility path planning several properties need to be fulfilled in addition to a positive cost function C. As stated in the work of Dechter and his colleagues [DP85], the estimated heuristic needs to be optimistic. This property is needed in order to stop the search as soon as the destination node in G_f is reached. This early abort of the search prunes away large parts of the graph while still delivering the optimal solution. Especially for short distances in a large graph, a significant speed-up can be achieved. If $h^*(\widetilde{P})$ is the true distance of the rest of $\widetilde{P}$, the estimated heuristic $h(\widetilde{P})$ needs to fulfill the follwing condition:

$$h(\widetilde{P}) \leq h^*(\widetilde{P}) \tag{5.4}$$

To get an accurate heuristic estimation of $\widetilde{P} = (p_0), \ldots, (p_k)$ the function $h(\widetilde{P})$ can be formulated in a similar way to the cost function C in equation 5.1, where $\boldsymbol{e}$ is the destination node of the path in G_f.

$$h(\widetilde{P}) = v(\boldsymbol{p_k}) d_{eucl}(\boldsymbol{p_k}, \boldsymbol{e}) + \alpha(1 - v(\boldsymbol{p_k})) d_{eucl}(\boldsymbol{p_k}, \boldsymbol{e}) \tag{5.5}$$

The heuristic function h depends on the end node $\boldsymbol{p_k}$ of the partially travelled path $\widetilde{P}$. Similar to C, a weighted sum of parts outside and parts inside the visibility region is calculated. The function $v(\boldsymbol{p_k})$ is the visibility probability of the node p_k to the given focus node f. If $\alpha >> 1$ this function delivers a $h(\widetilde{P}) \leq h^*(\widetilde{P})$ when the start point of the search is inside the visibility region. In this case, $v(\boldsymbol{p_k})$ is monotonically decreasing. Therefore $1 - v(\boldsymbol{p_k})$ is monotonically increasing.
Because $\alpha >> 1$, the distances inside the visisility region contribute much less to the total distance, and therefore a high visibility renders the estimated costs of the rest smaller than any edge cost outside the visibility. If the search would start outside the visibility region with

$v(\boldsymbol{p_k}) = 0$, an estimate of $h(\widetilde{P}) > h^*(\widetilde{P})$ is possible. In practice, however, this estimation error only delivers a slightly sub-optimal path when the destination position is exactly on the border. In cases where the end point of the path lies fully outside G_f, v or in the second row of nodes inside G_f, v, the search delivers the correct path.

Another property that needs to be fulfilled by the score function $F(C(\widetilde{P}), h(\widetilde{P}))$ is, according to Dechter [DP85], that it is order preserving. Since both the cost function C and the heuristic h have been reformulated for visibility transition planning, following implication must hold for F:

$$F(\widetilde{P_1}) \geq F(\widetilde{P_2}) \Rightarrow F(\widetilde{P_1}\widetilde{P_3}) \geq F(\widetilde{P_2}\widetilde{P_3}) \tag{5.6}$$

For any given pair of partial paths $\widetilde{P_1}$, $\widetilde{P_2}$ from a start node $\boldsymbol{s}$ to an extension node $\boldsymbol{p_k}$, and for any common extension $\widetilde{P_3}$, the implication demands: if the score of $\widetilde{P_1}$ is greater or equal to the score of $\widetilde{P_2}$, than it also is for the extension by $\widetilde{P_3}$. The score function of A* is defined as in equation 5.3. Equation 5.6 can now be rewritten as follows:

$$C_1 + h_1 \geq C_2 + h_2 \Rightarrow C_1 + C_3 + h_3 \geq C_2 + C_3 + h_3. \tag{5.7}$$

If we take the right part of the equation and formulate it using equation 5.1 where $\boldsymbol{p3}$ is the end node of the common extension P_3; we get:

$$\begin{aligned} &C_1 + \alpha d_{eucl}(\boldsymbol{p_k}, \boldsymbol{p3})(1 - v(\boldsymbol{p_k}, \boldsymbol{p3})) + d_{eucl}(\boldsymbol{p_k}, \boldsymbol{p3})v(\boldsymbol{p_k}, \boldsymbol{p3}) + h3 \quad \geq \\ &C_2 + \alpha d_{eucl}(\boldsymbol{p_k}, \boldsymbol{p3})(1 - v(\boldsymbol{p_k}, \boldsymbol{p3})) + d_{eucl}(\boldsymbol{p_k}, \boldsymbol{p3})v(\boldsymbol{p_k}, \boldsymbol{p3}) + h3 \end{aligned} \tag{5.8}$$

Since the two sub-paths $\widetilde{P_1}$ and $\widetilde{P_2}$ share the same end node $\boldsymbol{p_k}$, all parts except the distances C_1 and C_2 result in the same terms. If we take the left part of equation 5.6 we can say that $C_1 \geq C_2$. This inequation is true because the heuristic h is the same for both sub-paths since it only depends on the last node $\boldsymbol{p_k}$. Therefore, equation 5.8 is also true and equation 5.6 is satisfied for the cost function C defined in 5.1 and the heuristic h defined in 5.5.

All functions required for an A* search have now been modified to incorporate visibility. They can be applied in an arbitrary implementation of A* in order to find visibility optimal transitions. The algorithm implemented in the scope of this thesis was taken from Dechters article [DP85] and is described in the following pseudo-code:

```
1. Put the start node s on a list called OPEN of
   unexpanded nodes.
2. If OPEN is empty, exit with failure; no solution
   exists.
3. Remove from OPEN a node n at which F is minimum
   (break ties arbitrarily, but in favor of a goal
   node), and place it on a list called CLOSED to be
   used for expanded nodes.
```

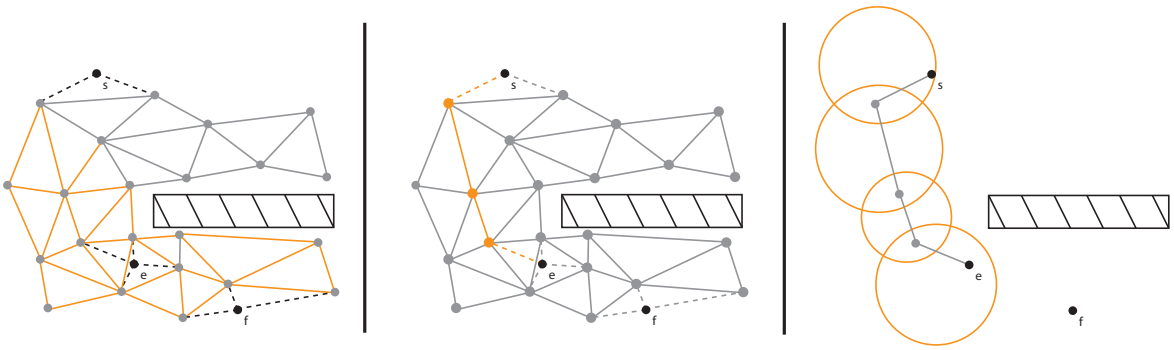


Figure 5.3.: The tree steps of the visibility search are shown. First, the start point $\boldsymbol{s}$ and end point $\boldsymbol{e}$ are inserted in G_f (shown on the left). Then the search is applied to find the visiblity optimal path between $\boldsymbol{s}$ and $\boldsymbol{e}$ (shown in the middle). The resulting path can then be associated with a series of spheres that define a region of space that contains the visibility optimal path (shown on the right).

```
4. If n is a goal node, exit successfully with the
   solution obtained by tracing back the path along
   the pointers from n to s (pointers are assigned in
   Steps 5 and 6).
5. Expand node n, generating all its successors with
   pointers back to n.
6. For every successor n' of n:
   a. Calculate F(n').
   b. If n' was neither in OPEN nor in CLOSED, then
      add it to OPEN. Assign the newly computed F(n')
      to node n'.
   c. If n' already resided in OPEN or CLOSED, compare
      the newly computed F(n') with that previously
      assigned to n'. If the new value is lower,
      substitute it for the old (n' now points back to
      n instead of to its predecessor). If the matching
      node n' resided in CLOSED, move it back to OPEN.
7. Go to Step 2.
```

By inserting a given start position $\boldsymbol{s}$ and a given end position $\boldsymbol{e}$ into G_f and applying the search algorithm with the score function F defined in equation 5.3, a series of nodes is found that describes a visibility optimal transition to a given focus point $\boldsymbol{f}$. These nodes are associated with spheres in space. The final result of the path search therefore is a series of spheres that contain the optimal transition between $\boldsymbol{s}$ and $\boldsymbol{e}$. Figure 5.3 illustrates the individual steps of the path search.

6

Path Post-processing

The visibility transition search described in chapter 5 results in a series of overlapping spheres that isolates a portion of space in which the camera may move without colliding. The resulting spheres are arranged in such a way that they contain the optimal visibility transition. An initial guess of the camera transition through these spheres from a defined start position s inside the first sphere to an end position e inside the last sphere can be computed by connecting the centers of the overlapping circles. This path, however, is most likely to be sub-optimal because the spheres can be large in comparison to the length of the path. On one hand, the connection of the circle centers can introduce large indirections. On the other hand, the spheres offer a lot of room to adjust the path. Especially, spheres that have partial visibility to the focus point potentially contain cluttered regions of occlusion and visibility.
This chapter discusses in detail a post-processing on the initial guess of the path. An iterative smoothing algorithm is presented that optimizes the path for two criteria. First, in regions of full or no visibility the shortest path is created. Second, in spheres with partial visibility occlusion maps projected from the focus point are used to refine the path. These maps contain border positions which enter or exit the visibility region. Using a prioritized optimization the path converges towards the optimal visibility transition as defined in chapter 5.

6.1. Initial Guess of the Path

The series of overlapping spheres that defines the limited space for the camera transitions can be simplified to a pipe within which the optimal visibility path with respect to the graph must lie. The overlapping space of two consecutive spheres is bound by their intersection circle (Appendix A.1 explains the construction of the intersection circle between two spheres in detail.) Therefore, a path from one sphere to its successor needs to pass through this circle. This setting

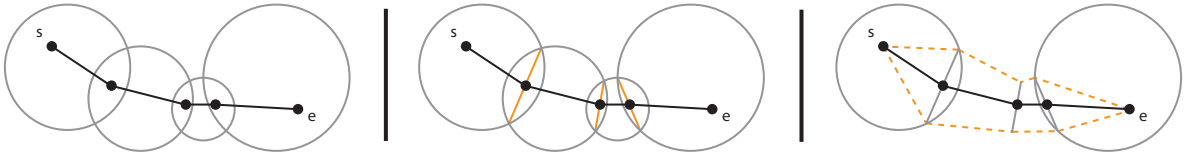


Figure 6.1.: The path with its corresponding spheres, shown on the left, is the result of visibility transition planning. The intersection circles of overlapping spheres (center) define a border pipe in which the path can be altered without exiting the spheres (right).

is illustrated in figure 6.1.
A path that is linearly interpolated between two portal circles, however, cuts away parts of valid positions inside the sphere. This space reduction is not a problem if this sphere has no or full visibility. In this case the partial optimal visibility path is equal to the shortest path. In spheres with partial visibility, however, the linear connection between the two bordering circles could miss visible portions of the sphere. This case is handled by refining the positions inside partially visible spheres using occlusion maps.

The initial guess of a visibility transition P between the start point $\boldsymbol{s}$ and the end point $\boldsymbol{e}$ can be expressed as a list of positions.

$$P = \boldsymbol{s}, \boldsymbol{p_1}, \ldots, \boldsymbol{p_{n-1}}, \boldsymbol{e} \tag{6.1}$$

The path is represented by the positions of P and the linear interpolations between those points. Except for the start and the end positions, all initial path positions $\boldsymbol{p_i}$ have a dedicated circle $C_i = (\boldsymbol{p_i^c}, \boldsymbol{n_i^c}, r_i^c)$ in three dimensional space with center $\boldsymbol{p_i^c}$, normal $\boldsymbol{n_i^c}$ and radius r_i^c. The path positions $\boldsymbol{p_i}$ can move inside their circle C_i. The initial guess of the path P^{init} can therefore be expressed as in equation 6.2. Figure 6.1 shows such an initial guess of the path.

$$P^{init} = \boldsymbol{s}, \boldsymbol{p_1^c}, \ldots, \boldsymbol{p_{n-1}^c}, \boldsymbol{e} \tag{6.2}$$

6.2. Iterative Optimization on Circles

We are given the initial guess of the path P^{init} with start position $\boldsymbol{s}$ and end position $\boldsymbol{e}$. Using a local smoothing for each position $\boldsymbol{p_i}$ on C_i, the path positions $\boldsymbol{p_i}$ can be moved towards the optimal visibility path. Since the path is required to pass through the circles and visibility is expressed in the spheres in between, the smoothing on intersection circles does not need to be optimized for visibility. The refinement of a point $\boldsymbol{p_i}$ on C_i can therefore be expressed as a function of the position itself, its predecessor path position $\boldsymbol{p_p} = \boldsymbol{p_{i-1}}$, and its successor node $\boldsymbol{p_s} = \boldsymbol{p_{i+1}}$. If the circle lies on a sphere with partial visibility, either $\boldsymbol{p_p}$ or $\boldsymbol{p_s}$ or both will be special border nodes inside the sphere. Refinement inside partially visibile spheres is covered by section 6.3.

We exploit the fact that the position $\boldsymbol{p_i}$ lies on the two-dimensional circle C_i in three-dimensional

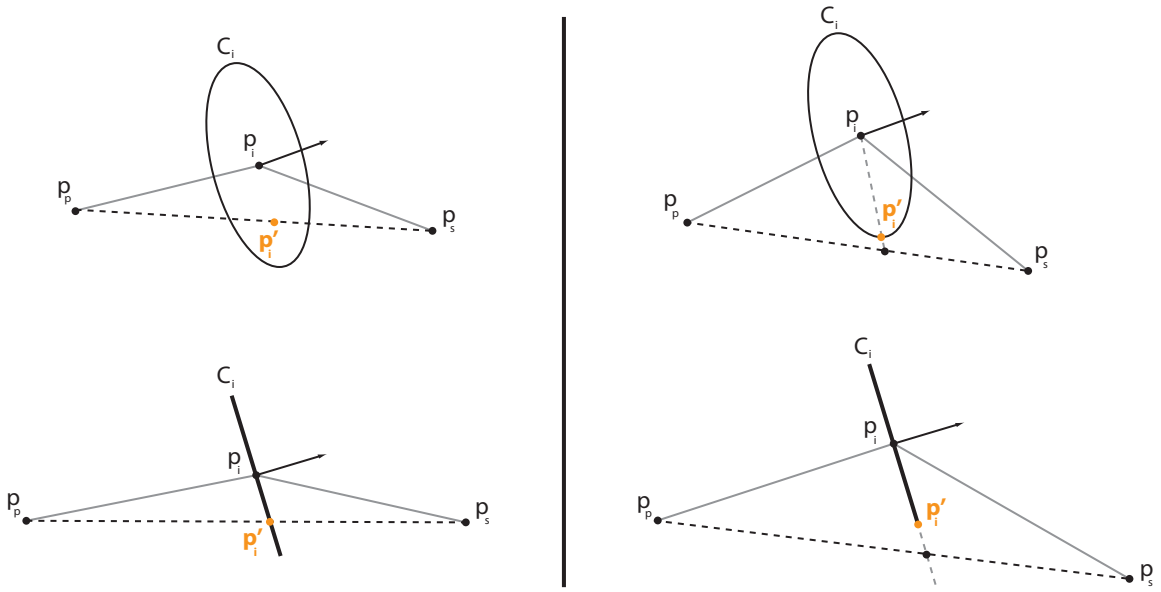


Figure 6.2.: This figure shows the default case of the local smoothing on an intersection circle C_i. The position $\boldsymbol{p_i}$ on C_i is replaced by the intersection of the ray between the previous and successor nodes $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ with the plane of C_i. The drawings on the right show the case when the intersection point lies outside C_i. In this case the new position $\boldsymbol{p'_i}$ is clamped with C_i

space. The goal is to minimize the distance between $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$. Since the shortest distance in Euclidean space is always a line, a new position $\boldsymbol{p'_i}$ can be computed by intersecting the ray defined between $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ with C_i. If the intersection of the ray with the plane of C_i lies outside the circle, it simply is clamped to the border of the circle in order to result in a valid position. Figure 6.2 illustrates these two default cases.

By iteratively applying this local smoothing successively for all circle nodes $\boldsymbol{p_i}$ of P, the path is smoothed towards the lowest curvature and converges to an optimum.
The simplicity of the approach using a ray between $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ to optimize $\boldsymbol{p_i}$ is the biggest advantage. The fact that it can be applied locally by only using the neighboring nodes in P allows simple multi-pass smoothing where $\boldsymbol{p_i}$ only needs to know the positions of its predecessor and successor nodes. It does not matter whether these neighboring nodes are positions on an intersection circle or special boundary nodes introduced by a partially visible sphere.

The local smoothing of $\boldsymbol{p_i}$, however, can have two special configurations in which this approach falls into a local optimum that is significantly worse than the global optimum. The following subsections explain these problems in detail and show how they can be solved.

6.2.1. Special Case: Overlapping Circles

The case of two overlapping circles needs to be handled specifically. Figure 6.3 illustrates the problem. If two consecutive portal circles C_i and C_{i+1} overlap one another, they share a line of

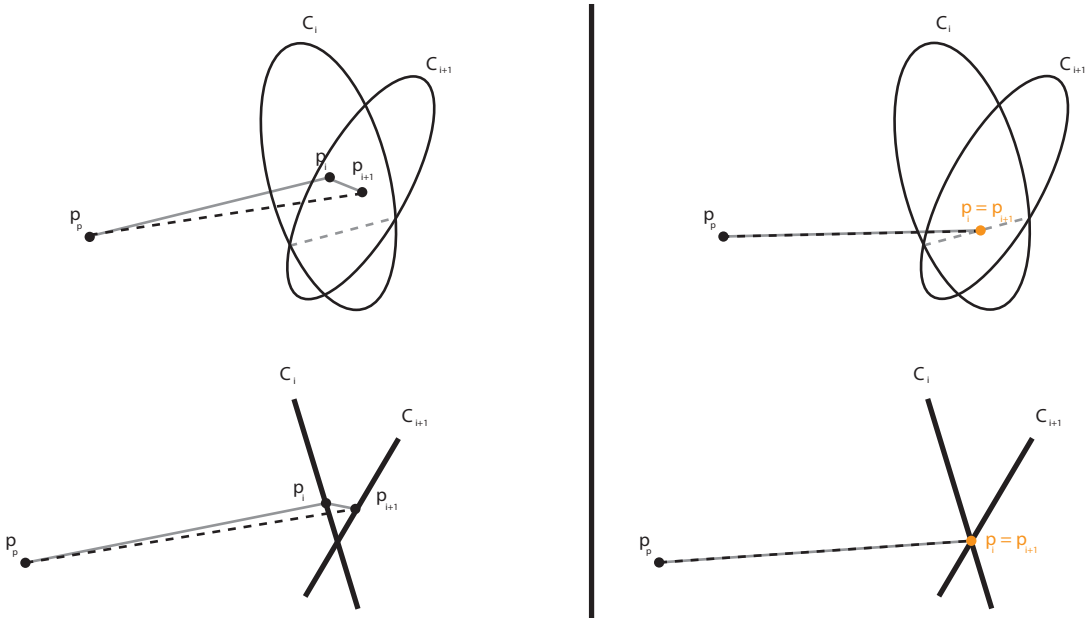


Figure 6.3.: This figure illustrates the problem that arises when two intersection circles C_i and C_{i+1} overlap. The initial setup is shown on the left. By iterativels applying the local smoothing the two positions $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ fall together. The smoothing stops in a local optimum.

points where both circles have valid positions. The case that $C_i = C_{i+1}$ can not occur since the spheres are selected such that they do not contain any other sphere's center. Due to the locality of the iterative smoothing, the position $\boldsymbol{p_i}$ on C_i and the position $\boldsymbol{p_{i+1}}$ on C_{i+1} may collapse to a single point. In this case the local smoothing explained above stops in a local optimum for both $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$. The ray defined by the predecessor and sucessor of both $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ intersects the circles at the collapsed point.

The problem is that the path converges to a local optimum since the circle portals C_i and C_{i+1} have a fixed order. An intuitively more optimal solution to this case implicitely assumes that C_i and C_{i+1} can change their order.
The special case of overlapping circles can be handled using a combined smoothing. When the distance between $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ falls below a threshold ϵ, this special case arises. In this situation, the predecessor path position $\boldsymbol{p_p}$ of $\boldsymbol{p_i}$ and the sucessor path position $\boldsymbol{p_s}$ of $\boldsymbol{p_{i+1}}$ need to be taken into account. These neighboring positions always exist since the path P starts with the fixed position $\boldsymbol{s}$ and ends with the fixed position $\boldsymbol{e}$.
Using $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$, the positions $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ on the two overlapping circles C_i and C_{i+1} can be smoothed similar to the default case. This combined smoothing is illustrated in figure 6.4.
The ray defined by $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ is intersected with and clamped to the two overlapping circles C_i and C_{i+1}. The resulting positions $\boldsymbol{p'_i}$ and $\boldsymbol{p'_{i+1}}$ are both positions between the neighboring path nodes that minimize the path length. The order of $\boldsymbol{p'_i}$ and $\boldsymbol{p'_{i+1}}$, however, may have changed. This requires an additional re-ordering of the two circles C_i and C_{i+1}. If $|p_p - p_{i+1}| < |p_{pred} - p_i|$ then their order has changed. In this case, the path nodes and their

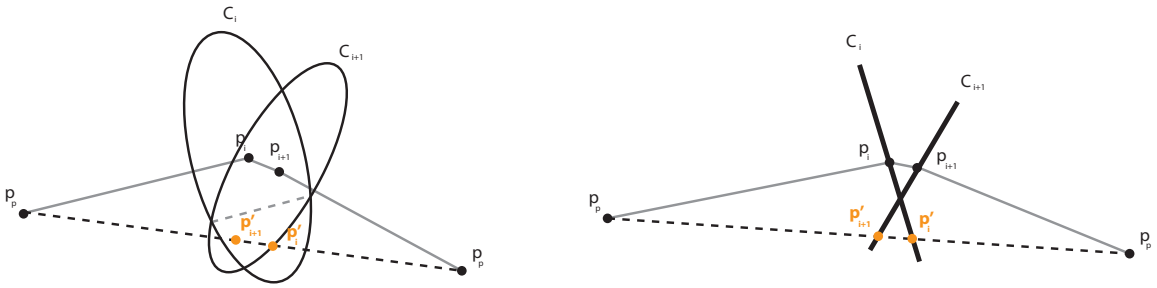


Figure 6.4.: In order to correctly smooth the positions $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ on two overlapping circles C_i and C_{i+1}, a combined approach is needed.

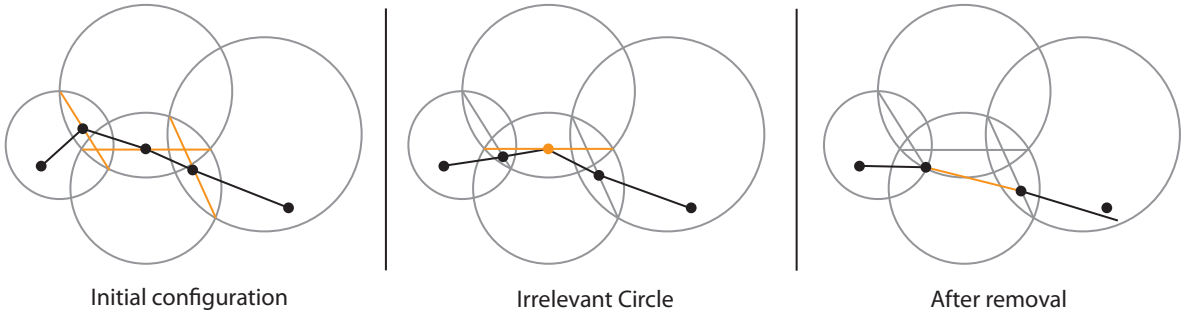


Figure 6.5.: The image on the left shows a configuration that can occur during visibility transition planning. The iterative smoothing can change the situation such that a circle becomes irrelevant. This case is shown in the middle. The image on the right shows the path after the removal of the irrelevant circle portal.

respective circles need to be switched in order to restore a well defined path.

6.2.2. Special Case: Irrelevant Circles

Another special configuration of a portal circle C_i that needs to be handeled separately is when its neighboring path nodes $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ lie on the same side of the circle plane. This case can occur because roadmap nodes are defined on the sphere's intersection circles. This case is particularly interesting because the local smoothing is computed as the intersection of the line between $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ with the plane C_i lies in. In this configuration, however, the line has no defined intersection point $\boldsymbol{p'_i}$ on C_i anymore. Figure 6.5 shows an example of this problem.

The main problem is the requirement of passing through all corresponding circles of the nodes in P. Circles with a configuration such that the neighbor nodes lie on the same side of the plane, however, do not contribute positions contained in the optimal path. Therefore, this case is handled separately in order to guarantee an optimal path within the bounding spheres.

It can be shown that if $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ lie on the same side of the plane of C_i, they are both situated in the same sphere. Since positions are only inside a sphere if they enter through a portal circle,

it is clear that both C_p and C_s are in the same sphere. This property is guaranteed because of the definition that no sphere contains the center of another sphere. Therefore, all positions on C_i do not contribute to the restriction in space. Moreover, if the path is required to visit p_i on C_i, it is drawn away from the optimum. Since both $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ are in the same sphere s_i its associated circle C_i can simply be removed from the path.

6.3. Optimization in Partially Visible Spheres

The path through the circle centers is valid as it stays inside the collision free area defined by the spheres. Furthermore, by the local definition of the iterative smoothing, the path nodes p_i do not have to worry about where their neighbors are defined. The smoothing converges towards a position on the dedicated circle C_i that minimizes the path between their predecessor and successor. The path smoothing so far, however, is not satisfactory regarding visibility. In contrast, it is optimized for overall shortest distance. Even though the list of spheres is the result of path planning with visibility criteria, spheres of this path can be large and therefore potentially contain positions that allow an early path to visibility. Since we are given the visibility probability p_{if} for each sphere s_i of the path, all spheres that may contain such positions can be identified. If p_{if} is either 0 or 1, all positions inside s_i have equal visibility to the focus point. In these spheres, the shortest path created by the local smoothing on the portal circles is already optimal. However, if p_{if} is inside the range $(0; 1)$, positions of different visibility are distributed inside s_i.

In order to find a path that optimizes for the notion of visibility given in chapter 4, these spheres are important. They contain somewhere the position at which the path first enters or exits the visibility region of $\boldsymbol{f}$. The only information available at this point, however, is the visibility probability p_{if}. No assumptions can be made about where the point lies that first enters or exits visibility. Refined information about the visibility between all points inside the partially visible sphere $\hat{s}_i$ and the focus point $\boldsymbol{f}$ is required.

This three dimensional problem can be reduced to a two dimensional one. By projecting the geometry between $\boldsymbol{f}$ and $\hat{s}_i$ onto an occlusion map, an approximative layout of the visibility region inside the sphere can be computed. In this approach $\boldsymbol{f}$ is the center of projection. The projection plane is a square area that encloses s_i seen from f. Chapter A.2 explaines this projection in more detail. Figure 6.6 illustrates this projection schematically.

The position and orientation of the projection map is clearly defined in space. Therefore, each pixel $\boldsymbol{t^{2D}}$ of the map corresponds to a position $\boldsymbol{t^{3D}}$ in three-dimensional space. Furthermore, each pixel of the occlusion map is interpreted as a node in a new graph. The edges between the nodes are defined between neighboring pixels. This allows a similar visibility search on the map pixel to the one described in chapter 5. Figure 6.7 illustrates this small-scale search.
The only difference is that visibility is given as a boolean value on the nodes instead of being defined on edges. The visibility search on the occlusion map can be expressed with only small modifications to the large scale search on the roadmap. The following equation describes the distance cost for an expansion of a pixel $\boldsymbol{t^{2D}}$ to its neighbor $\boldsymbol{u^{2D}}$:

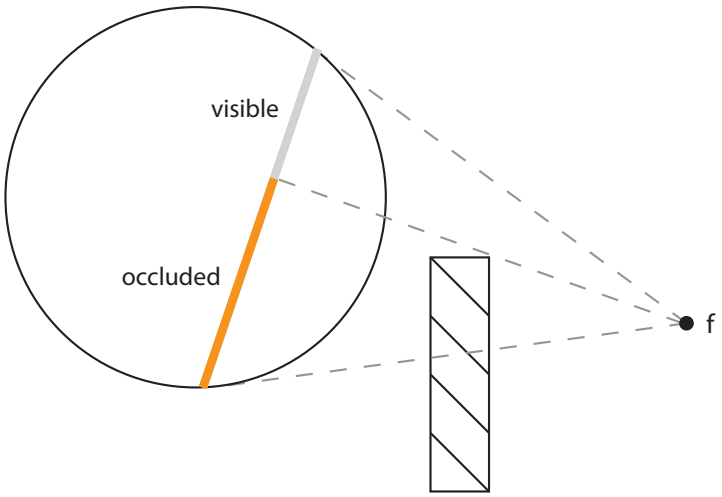


Figure 6.6.: By applying a projection of the geometry between the focus point $\boldsymbol{f}$ and the partially visible sphere s_i, a refinement of the visibility regions inside the sphere can be achieved.

$$D(\boldsymbol{t^{2D}}, \boldsymbol{u^{2D}}) = d_{eucl}(\boldsymbol{t^{2D}}, \boldsymbol{u^{2D}}) + \beta(1 - v(\boldsymbol{u^{2D}})). \tag{6.3}$$

The distance cost $D(\boldsymbol{t^{2D}}, \boldsymbol{u^{2D}})$ is the combination of the two-dimensional distance $d(\boldsymbol{t^{2D}}, \boldsymbol{u^{2D}})$ and a penalty β. The penalty is added if the new pixel $\boldsymbol{u^{2D}}$ is occluded and therefore $v(\boldsymbol{u^{2D}})$ equals 1. The magnitude of β determines how long a series of visible pixels needs to be in order to have a larger distance than one occluded pixels. Similar to the penalty for edges outside visibility in chapter 5, β should be chosen such that one occluded pixel costs more than the longest possible path of visible pixel. Since the map has a defined resolution, an appropriate value for β can be computed. In our implementation, an β of two times the resolution has proven to work well.

This small-scale visibility search optimizes positions inside $\hat{s}_i$ such that the distance outside visibility is minimized. Since $\hat{s}_i$ is between the two path nodes $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$, the start point $\boldsymbol{s^{2D}}$ and end point $\boldsymbol{e^{2D}}$ on the occlusion map can be calculated by projecting the neighboring path positions onto the map. By connecting $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ with the focus point $\boldsymbol{f}$, a ray is created that intersects the projection map at $\boldsymbol{s^{2D}}$ and $\boldsymbol{e^{2D}}$, respectively. This procedure is illustrated in figure 6.7.

6.3.1. Border Nodes

The main goal is to minimize the parts of the path lie outside of visibility. Since the visibility search on the projection map already optimizes on the projection plane to minimize occluded regions, the problem simplifies to individual border points on the map path. The positions of interest are points on the projection map that either enter or leave visibility. Since the map search heavily penalizes occluded regions, the parts of the path that lie in an occluded area always form a straight line. This property allows the simplification for the path smoothing to

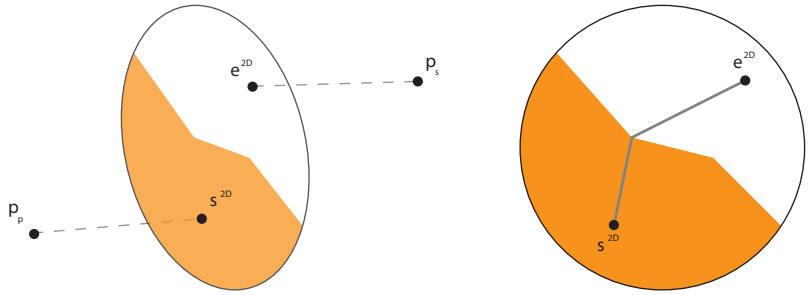


Figure 6.7.: This figure illustrates the setup of the small-scale search on an occlusion map. The given neighbor nodes $\boldsymbol{p_p}$ and $\boldsymbol{p_s}$ in the path are projected onto the occlusion map to find the path start and end positions $\boldsymbol{s^{2D}}$ and $\boldsymbol{e^{2D}}$. Then a search on the map is performed to find the optimal path with respect to visibility between $\boldsymbol{s^{2D}}$ and $\boldsymbol{e^{2D}}$.

only optimize for these border points.
The special treatment of these border points on the occlusion map is applied when their 3D positions are reconstructed. Each of these border points lies somewhere on the line defined by the 3D position of the map pixel and the focus point $\boldsymbol{f}$. Furthermore, valid positions on this line are restricted by the sphere onto which the occlusion map was projected. In order to properly reconstruct the corresponding path nodes from the border nodes, five cases need to be distinguished. These cases are illustrated in figure 6.8.

The goal is to insert 3D reconstructions of these border positions into the path. With a proper reconstruction of these points, the path inside the sphere is automatically optimized for visibility. This is because the border nodes in the path are actively optimized for visibility. The nodes on the portal circles are then indirectly optimized for visibility as well, since they use neighboring border points in the local smoothing.
The different cases of the border points arise depending on the visibility of the projected start $\boldsymbol{s^{2D}}$ and end $\boldsymbol{e^{2D}}$. The three-dimensional reconstructions $\boldsymbol{b_i^{3D}}$ of the border points $\boldsymbol{b_i^{2D}}$ on the map need to be chosen such that the distance outside visibility is minimal. Border points where the path enters visibility thus need to be reconstructed as the nearest point on their view line to the predecessor $\boldsymbol{p_p}$. Border points where the path exits visibility need to be optimized the other way round. They are reconstructed such that they minimize the distance to $\boldsymbol{p_s}$. These reconstructions are illustrated in figure 6.8.
Case 5 in figure 6.8 follows the same idea as the previous ones. The situation, however, is different. If a path on the occlusion map is completely invisible no border point exists. Still, there might be points on the map that are not occluded. This case arises when the number of pixels the path needs to follow between $\boldsymbol{s^{2D}}$ and $\boldsymbol{e^{2D}}$ is smaller than the number of pixels that are needed to reach visibility. In order to still find a valid border point a search is started that finds the nearest pixel $\boldsymbol{b^{2D}}$ to $\boldsymbol{s^{2D}}$ that is not occluded. This pixel can now be treated the same way as case 1 in figure 6.8. This additional case assures that the visibility region not will be missed.

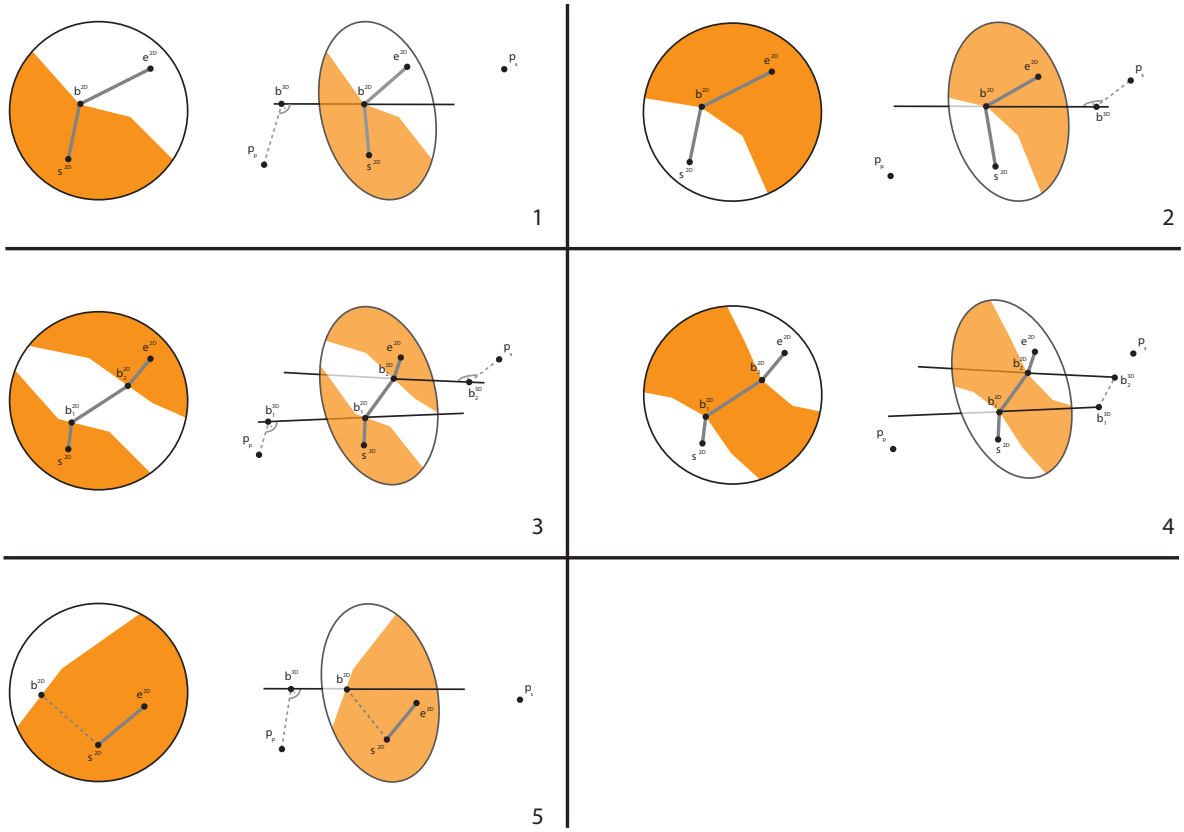


Figure 6.8.: This figure shows the different cases of border points on an occlusion map that need to be handled. On the left side, the situation on the map is shown. On the right side, the correct reconstruction of the border points' 3D position is shown.

6.3.2. Iterative Smoothing with Projection Maps

The small scale visibility search on projection maps in partially visible spheres introduces a refinement to the path smoothing. By inserting all border points $\boldsymbol{b_i^{3D}}$ into the path node list, the iterative smoothing results in a path optimized for visibility. Since the local smoothing on circle nodes was defined to only use the predecessor and successor, it still optimizes correctly if these neighboring points are introduced border points.

The iterative smoothing for the border nodes can be applied on partially visible spheres in a similar way. The smoothing algorithm for each border node only depends on the positions of the node's predecessor and successor nodes in the path. Since these neighbor nodes most likely change with each iteration, a new search on the projection map is required. Therefore, also the reconstruction of the 3D border nodes $\boldsymbol{b_i^{3D}}$ along the map path also needs to be calculated for each iteration. These newly constructed $\boldsymbol{b_i^{3D}}$, in turn, provide updated positions that are used for the smoothing of the neighboring points. The result is that the path converges towards the optimal visibility transition.

7

Dynamic Camera Control

Chapters 4, 5, and 6 have shown an approach to discretize the free space of a scene and how to create a visibility aware transition between two arbitrary points. The path search and transitions smoothing provide a global solution for camera work. Whenever local methods like ray casts or occlusion maps come to their limits, a global solution can be found and used as a fall-back option.
The main application of a global camera transition is the switch between camera modes. This switch can be a long transition like changing from first-person view to a top-down camera. Also, several similar local camera modes like a collection of alternative over-the-shoulder views can be connected using the visibility transition.
The problem with a camera constrained to a character, however, is that the avatar is dynamically and unpredictably moving. This problem also contains the constant and possibly sudden change of the local surroundings. At this point, local camera control reach their limits and eventually break some of the defined properties of a virtual camera: smoothness, continuity, or collision and penetration avoidance. Also, a simple static computation of a transition path may lead to unpleasing results when its desired configuration is changing while the camera is traveling the path. In order to create a robust global camera controller, dynamic update of camera transitions is inevitable. This chapter explains an extension to the camera controller introduced in chapter 3. This extension only affects the control module itself and therefore preserves the capability of handling different camera modes interchangeably.

7.1. Dynamic Controller

The goal of the controller is it to deliver a desired camera configuration at each program update. Usually this configuration is the position and orientation of the camera constrained by the cur-

rently active control routine. When another mode is requested, however, the controller needs to find an adequate visibility transition between the two configurations. This approach introduces two main problems. First, since the computation of visibility transitions is output sensitive the calculations may require tens of milliseconds. A naive implementation may block the main program. This overhead introduces undesired lags. Second, since the requested camera mode may be directly constrained to a user controlled object, the camera position most likely changes. The camera, on the other hand, should not surpass a certain speed. This leads to the problem that a path can become suboptimal while it is travelled.

To solve these problems, a multi threaded approach can be used. By sourcing the computations of the visibility path out to a second CPU core, the main program can be freed of large blocks of computations. This parallel approach solves the problem of temporal overhead and resolves undesired lags. Dynamic re-computation of the visibility path while it is traveled solves the problem of changing end- and focus points. While a previously computed path is traveled, a new one can be computed that has up-to-date information about its destination position and the point to focus on.

The visibility planning contains two computationally heavy parts. The first part is the planning on the pre-computed road map for an initial guess of the path. The costs for the planning are output sensitive. The second part is the path smoothing. If the path needs to be smoothed using several projection maps in partially visible spheres, the overhead can also become quite large.

In order to source out both the path computation and smoothing to a parallel thread special measures need to be taken. The problem is the creation of the projection maps at the beginning of the smoothing. Since these maps are computed on the GPU, an asynchronous processing may lead to collisions with the hardware. This requires an extra synchronization step. In order to handle this problem, a state machine implementation of the controller has proven useful. A state machine provides excellent functionality for thread synchronization by communication through a thread-save collection of parameters. Additionally, state machine is easily extensible with new states without changing the old ones.

The state machine shown in figure 7.1 implements a dynamic controller that handles the different parallelizations and synchronizations. It automatically addresses change requests and guides the camera along a dynamically updated transition.

The state machine consists of three main parts. The first part is the *Ready-State* and its designated synchronization states. While the camera controller resides in this first part, it returns the desired configuration of the current active control routine. When a request for a new camera mode arrives, a new thread is started that computes the visibility transition between the current position and the requested mode's position. The controller continues to return the configuration of the old mode until both the transition computation and post-processing computations are finished. The controller then changes to the second part: the initialization of path traveling.
The initialization of the traveling has two assignments. First, the actual initialization of the path traveling includes the creation of data structures and the computation of parameters that are needed by the *Travel-State*. Second, this part also initializes a new transition search for the conditions that have changed while the first transition was computed. By estimating the time that is required to compute this new transition, a position on the available path can be calculated from which the new transition should take over. This position is used as the start point for the new transition search.

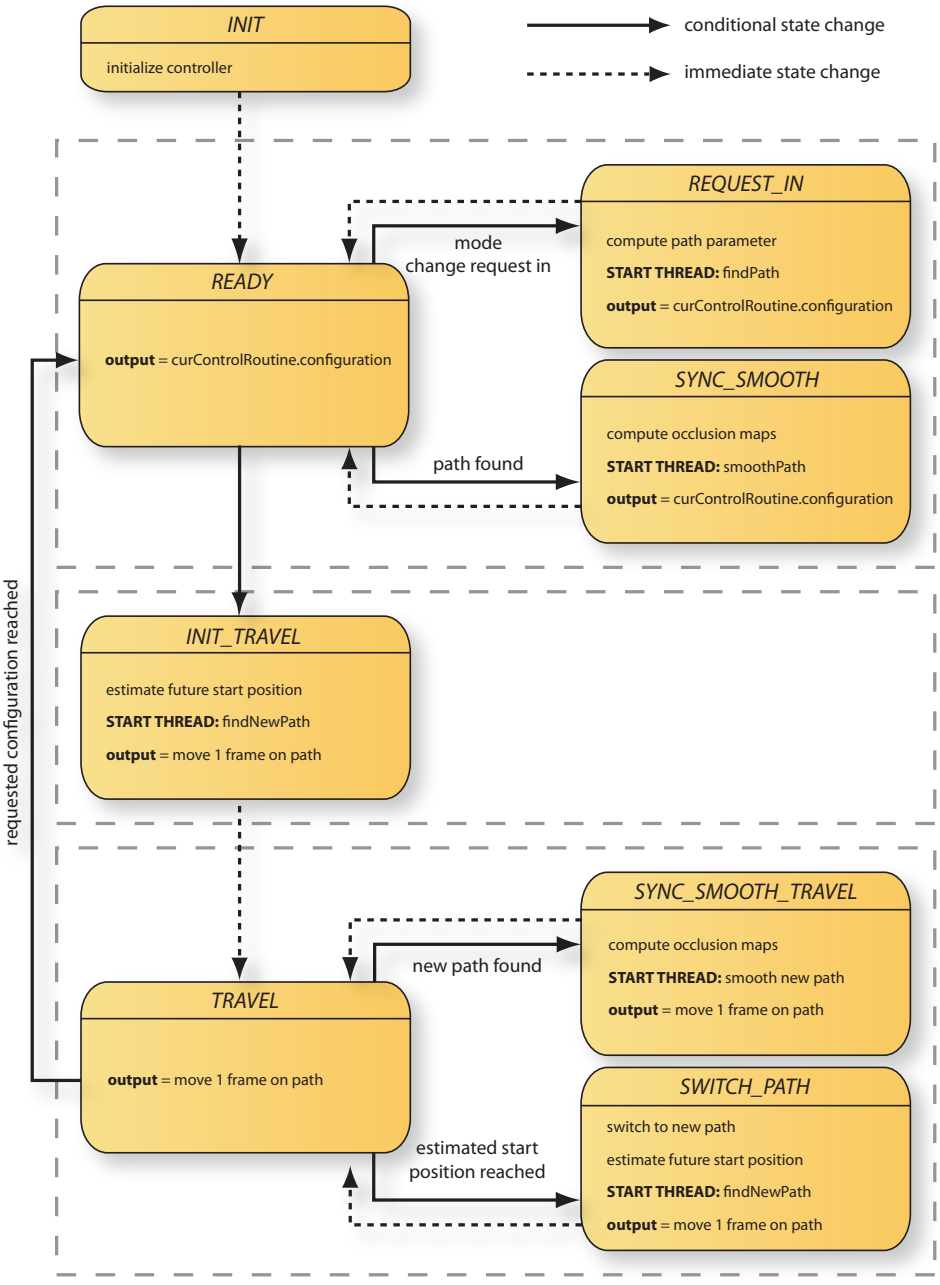


Figure 7.1.: This figure shows the state machine of the dynamic camera controller. It is an extension of the simple controller that was shown in the context of the physical camera module in chapter 3.

The last part of the controller state machine is the actual traveling of a camera transition and the dynamic update of the path. The layout of the third part is very similar to the first part. The difference is in the returned desired configuration. In contrast to the first part the controller moves a point on the path and returns this position and an extra check is made in the *Travel-State*. When the previously computed estimated future position on the currently traveled transition is reached, the controller changes to the *Switch-State* where the current transition is switched with the updated one. Then also, similar to the initialization state, a future position on the new transition is computed to start another thread to compute a new transition. The controller remains in the third part until the destination is reached and switches back to the *Ready-State*.

7.1.1. States In Detail

This section explains the individual states of the controller state machine in detail. The behavior of each state is described as well as the conditional switches to other states according to figure 7.1.

Init-State. At the construction of the controller class its current state is set to the initialization state. This state is only active at the beginning of the state machine update. It initializes all parameters that are needed by the dynamic controller. This initialization guarantees the correct setup of the state machine after the first update call.

The *Init-State* immediately changes to the *Ready-State*.

Ready-State. The *Ready-State* stores the desired configuration of the current active control routine in a global property of the controller. This property allows the state machine to store different output configurations by maintaining the same interface for the camera to access the desired configuration. The *Ready-State* is also the only state in which external requests for mode changes are allowed. If, however, a transition computation or smoothing thread is running, requests are blocked. This restriction is needed to guarantee the robustness of the state machine.

If a request for a new camera mode arrives, the *Ready-State* stores the desired camera configuration for the currently active control routine to the output fields and then changes to the *Request-In-State*.

Request-In-State. If this state is active a mode-change request together with the desired camera mode is pending. The duty of this state is to collect the information needed to begin a visibility transition search. These parameters are start-, end-, and focus-point of the transition. The start point is the current camera configuration. The end- and focus-points are computed by the control routine of the requested camera mode. After gathering the necessary information, the path search on the roadmap is started in a new thread. Additionally, a time stamp is stored which holds the time when the transition planning thread has started. This time stamp will be used in the *Travel-Init-State* to estimate where the new transition path shoud begin.
Finally, the *Request-In-State* needs to define an output configuration. Since the transition computations have just started, the desired configuration of the current camera mode are stored.

The *Request-In-State* then immediately changes back to the *Ready-State* which will hold the output configuration up-to-date while the visibility transition planning thread is running in parallel.

Sync-Smooth-State. The visibility transition planning thread started in the *Request-In-State* actively changes the state machine to the *Sync-Smooth-State* when the search is finished. This state is needed to synchronize the computations of the projection maps for the path smoothing with the main program. The synchronization state first computes the required occlusion maps on the GPU and starts a new thread that performs the post-processing on the previously computed camera transition. At the end, it stores the desired configuration of the current camera mode to the global output fields.

The *Sync-Smooth-State* state immediately changes back to the ready state, which will keep the output configuration up-to-date while the path smoothing is performed in parallel.

Travel-Init-State. When the transition post-processing thread is finished, it actively changes the current controller state to the *Travel-Init-State*. This state prepares for traveling on the computed and smoothed path. Additionally, this state initializes the dynamic re-planning of the camera path.
First, the path controller is initialized with the computed camera path. In order to be able to start a new path search, a future position needs to be estimated. Using the time stamp that was stored when the path search was started, a future position on the transition can be guessed at which the new path search should be finished. Given the duration Δt of all computations for the camera transition and the travel velocity v, the future position on the path can be estimated as follows:

$$\boldsymbol{p} = P(v\Delta t(1+\delta)). \tag{7.1}$$

In order to avoid lags introduced by waiting for the results of the threaded computations, a safety buffer δ is used. Using this future position, a new transition planning thread can be started. At this point, the time stamp that stores the search start time is updated to the current time.

The initialization state for traveling also needs to store an output configuration. This configuration is computed by moving one frame on the path and storing the according configuration.

The *Travel-Init-State* then immediately changes to the *Travel-State* state.

Travel-State. This state is very similar to the *Ready-State*. It is a state that organizes the path traveling while parallel threads compute a new path. The job of the *Travel-State* is, for each call, to move one frame on the path and store the corresponding configuration in the output fields.

The *Travel-State* has two possible exits. First, if the end position on the path is reached the traveling has finished. In this case, the state machine is changed to the *Ready-State*. Second, if the estimated start position of the newly computed transition path is reached, a change to the *Switch-Path-State* can be attempted. At this point, however, the new camera transition may not

yet be ready. This mainly depends on the parameter δ used in equation 7.1. To handle this time difference, the *Travel-State* must wait at the estimated start position until the new path is available before changing to the *Switch-Path-State*.

Sync-Smooth-Travel-State. The purpose of this state is to synchronize the occlusion map creation of the parallel computed path with the main thread. It has the equivalent functionality as the *Sync-Smooth-State*. The motivation to use a different state for a similar task lies in the difference of the output. Since this state is activated when a path is traveled, it has a different output. In contrast to the *Sync-Smooth-State*, the output depends on the path.
When the parallel visibility transition planning thread started in the *Init-Travel-State* has finished, the state machine is changed to the *Sync-Smooth-Travel-State*. Its first task is to compute the occlusion maps in order to initialize the path post-processing. After starting the transition smoothing in a separate thread, the output configuration is computed by moving one frame on the current path and storing the corresponding configuration.

The *Sync-Smooth-Travel-State* then immediately changes back to the *Travel-State*.

Switch-Path-State. This state is activated if the travel state has reached the estimated new start position of the new path and the parallel computations of the new path have finished. The *Switch-Path-State* re-initializes the path controller with the new computed camera transition. It also estimates a new start position using the same approach as the *Init-Travel-State*. (See equation 7.1.) In order to initialize a new visibility planning, the time stamp for the new search is updated to the current time. Then, the same thread function is started as in the *Travel-Init-State* which performs a visibility search on the roadmap. The output fields are updated by moving one frame on the new path and storing the corresponding configuration to the output fields.

The *Switch-Path-State* then immediately changes back to the *Travel-State*.

7.1.2. Dynamic Controller For Local Control

The dynamically updated camera controller in figure 7.1 supports an arbitrary number of local camera modes. These modes are active while the state machine resides in the *Ready-State*. External requests are handled using a dynamically updated visibility transition that guides the camera to the desired position of the requested camera mode.
Local methods, however, have several limitations because of their inherent lack of global information. One method to resolve situations in which local methods fail would be to detect them and use the dynamic transition planning in order to reach an appropriate position. The difficulty in this approach is that for everly local method, every undesired situation must be detected. In some cases, this may be trivial. In other cases, however, this detection can explode in complexity.
A far easier and more robust approach is to avoid the problem of detecting bad situations. If a camera mode is viewed as the destination configuration of a dynamically updated camera path, discontinuities will automatically be resolved. Also, situations where the camera would be

forced through objects are avoided. By removing the constraint in the *Travel-State* that changes the state machine to the *Ready-State* as soon as the destination point is reached, the dynamic visibility planning algorithm stays active all the time. This approach always provides a valid global transition available that guides the camera towards the desired configuration. The camera control routines can be used, unmodified, as in a normal camera mode. The visibility planning automatically decides which path is optimal with respect to the focus point to move from the current camera configuration to the one defined by the control routine. Therefore, an arbitrary local method can be used as a guide to define a broad camera configuration which then is approximated using transitions that satisfy the demands of smoothness, continuity and visibility awareness.

7.2. Travelling On Visibility Transition Paths

After visibility search and post-processing, a visibility transition path is given as a list of vectors in three dimensional space:

$$Path = \{\boldsymbol{p_0}, \boldsymbol{p_1}, \ldots, \boldsymbol{p_{n-1}}\}. \tag{7.2}$$

In order to be able to travel on this path with an arbitrary speed v, the positions between the guard nodes $\boldsymbol{p_i}$ need to be interpolated. Since these guard nodes allow no assumptions about their distance, special care needs to be taken with the linear interpolation. In one step, several nodes may be passed in order to travel the required distance $v\delta t$. The following block of pseudo-code performs a correct interpolation between the nodes regardless the size of the distance $v\Delta t$.

```
Vector3 moveOnPath(float distance, Vector3[] Path) {
    int curIndex = 0;

    Vector3 current = Path[curIndex];
    Vector3 next = Path[curIndex + 1];

    float nodeDistance = Distance(current, next);
    float travelDist = distance;

    while (travelDist > nodeDistance) {
        if (curIndex + 1 == Path.Length -1) {
            // path end reached
            return Path[curIndex + 1];
        }

        travelDist -= nodeDistance;
        curIndex += 1;

        current = Path[curIndex];
```

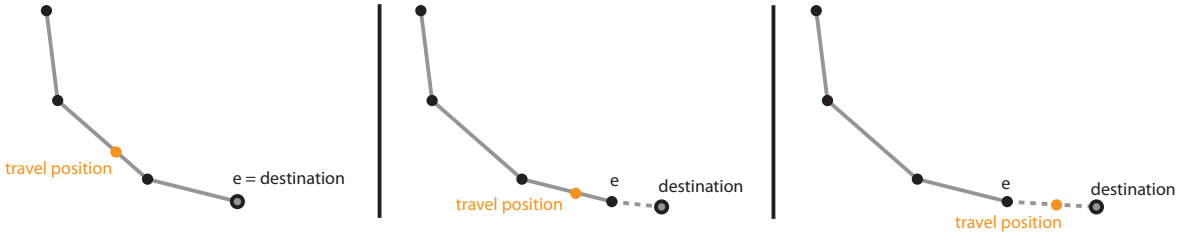


Figure 7.2.: In order to decrease the need for path re-computations for small distances, a path can be extended by one node. This last node is kept up-to-date with respect to the destination position. This additional node allows the path to be traveled to the end when the destination only has changed its position by a small amount.

```
            next = Path[curIndex + 1];

            nodeDistance = Distance(current, next);
        }
        // travelling ends between current and next node
        return lin_interpol(current, next, travelDist)
}
```

The function takes the the path node list and the distance to travel as parameters and returns the position on the path at the desired distance. The purpose of the while-loop is to handle the case that an arbitrary number of nodes may be passed while traveling. Each time a node is passed, the focus is shifted by one node and the distance between those nodes is subtracted from the total distance that needs to be traveled. At the beginning of each loop, a test determines if the path has traveled to the end. If so, the end node of the path is returned.
When the while loop ends, the focus window of the loop has stopped at the two nodes where the end point of the traveling lies in between. Here, the final position on the path can simply be computed by a linear interpolation between the two node points.

Traveling a path in a dynamic environment leads to several problems. Large-scale changes of end- or focus point are adequately handled by the dynamic approach, when the end point is not moving. One pitfall, however, is introduced by small changes of the end point during the travelling. The static path that is available during the time the dynamic update computes a new one becomes very short when the end point is almost reached. It is in such cases possible that the end of the path is reached before the new path is available. The destination position of the path, however, may have changed as well, introducing a gap between the end point of the available path and the destination position. Figure 7.2 shows an example of this case.
This problem could be resolved by the controller waiting for the new computed transition at the end position of the currently available path. Since the distance to the destination point is likely to be very small, the situation can be resolved more aesthetically by introducing an extra segment at the end of the path that is in each frame updated to connect to destination. This trick is performed by adding an extra node to the path list:

$$Path' = \{Path, \boldsymbol{p_{dest}}\} = \{\boldsymbol{p_0}, \boldsymbol{p_1}, \ldots, \boldsymbol{p_{n-1}}, \boldsymbol{p_{dest}}\}\,. \tag{7.3}$$

At each update step that uses a transition path, the first thing that needs to be done is change the additional path node $\boldsymbol{p_{dest}}$ to be equal to the moving destination point. All other computations can continue in the same way as before. This small trick introduces an extra linear interpolation of a static path with the actual destination. Using $\boldsymbol{p_{dest}}$, a path always ends at its destination. In cases where the path is almost traveled until its end with a small change of the destination point the additional node removes the need to wait for an updated version of the path.

7.3. Camera Orientation During Travelling

The problem of orienting the camera during the transition traveling phase is mainly an artistic problem. The transitions are travelled in static form, wile a new one is computed. Therefore, they also assume a static focus. With an adequate update rate, this focus only stays for a limited number of frames at the same position. This duration, however, is already too long to orient the camera to. Every glitch or kink in the orientation will be noticed by the user. Therefore, the orientation of the camera needs to be computed dynamically while traveling on the path rather than use the given focus of the currently travelled path.

The easiest way to orient a camera between the start and end of a transition is to keep a single point in focus. This point can be static in cases where the object in question is not moving. The point may also be moving as well. This is the case if the camera changes between two third-person views on the character or is resolving dynamically problems of local methods.
Each time the controller stores the output configuration while traveling, the vector between the current camera configuration and the focus point can be used to compute the appropriate orientation. This approach will lead to a smooth camera work. Additionally, the camera keeps the avatar in the center of the image plane even when it may be temporally occluded. The user therefore already gets an impression of the point on which the camera is focusing.

Using the approach of computing the orientation depending on several parameters after the actual position of the camera on the visibility transition is known allows complete freedom on how to orient the camera. Since the optimality criteria introduced in chapter 5 only restricts the positioning of the camera, several high-level goals can be defined for the orientation. Since, for each point on the path, information about the visibility of the focus point is available, sophisticated camera work may be implemented that changes the camera orientation based on these parameters.

8

Results

This chapter presents the results of the visibility transition planning in interactive environments. The first part discusses the implementation of the planning algorithm and the dynamic camera controller. This part mainly focuses on the technical parts of the implementation as well as the computational costs for the underlying hardware. Then, two example applications of the approach are shown. First, it is shown that the visibility planning algorithm can be used for navigation in urban environments. Especially, the differences between visibility path and shortest path transitions are discussed. Second, the capabilities of the dynamic camera controller are shown. In this example the camera automatically follows a fast paced character that moves in a typical game environment.

8.1. Implementation

The presented roadmap creation and visibility transition planning was implemented using C#. For the visualizations, the Microsoft XNA 1.0 Refresh Framework was used.
The pre-computations use only one CPU core. Since these computations do not need to be performed on-line, no effort was made to speed up the calculations. A BSP-tree was used for the ray cast intersections in the visibility pre-computations. This alone achieves a significantly faster computation time for the second pre-computation phase. Figure 8.1 shows three example environments on which the pre-computations were performed using an Intel Core 2 CPU at 2.4 GHz and 2 GB ram.
The simple level was computed using an underlying grid of resolution 20x10x20 cells. The entire creation of the roadmap for this level was done in less than two minutes and resulted in 544 spheres and a graph with 2204 nodes. The visibility computations were performed without a distance threshold and with 20 sample rays for each sphere. The time required to compute the

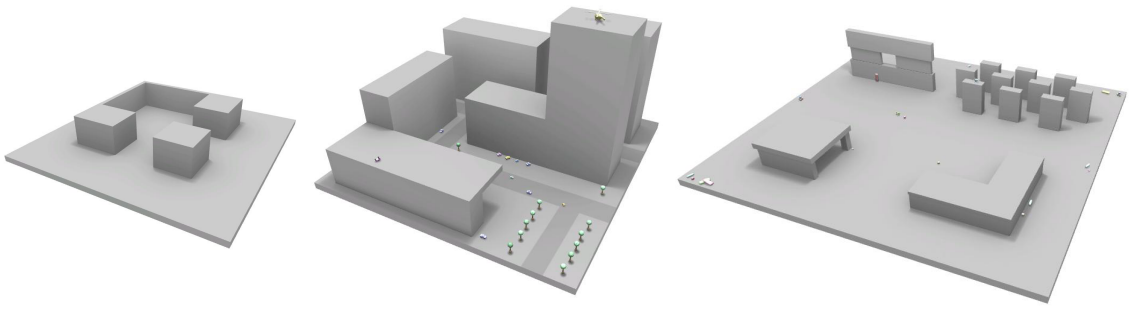


Figure 8.1.: These are screen shots from three demo levels on which the algorithm was tested. The level on the left is a simple setup that allows to examine differences between shortest path and visibility path. In the middle an example of an urban level is shown. The image on the right shows a map that contains several obstacles that create typically difficult situations for a camera that follows a game character.

visibility probabilities for the 504 spheres was also about two minutes.
The pre-computations on the urban level required a denser sampling, especially in the y-direction. The grid was of resolution 30x40x30 cells and was one third higher than the level's bounding box. The resulting sampling of the free space resulted in 3245 spheres and a roadmap with 13920 nodes. The reason for the dense sampling of the spheres is that this level required the roadmap to reach high above the buildings. The upper boundary for the radii of the spheres was set to be three times the average cell size. The creation of the roadmap required about 55 minutes. The visibility computations were done in less than 200 minutes.
These results already show that the pre-computation time mostly depends on the density of the grid. This observation makes sense, since the grid resolution results in a runtime of $O(n^3)$. The visibility pre-computations depend on the number of spheres and the number of sample rays used to approximate the visibility probability. In our tests, a sample rate of 20 rays per sphere pair has proven to be quite accurate, while still keeping the computation time relatively low.
The third level, the game environment, allowed a reduced grid resolution in y direction. A grid resolution of 30x20x30 cells to capture of all features. This level was sampled with 2418 spheres and the resulting roadmap contained 10300 nodes. The following table summarizes the technical data for the three example maps:

Level	Simple	Urban	Gamecity
Grid	30x15x30	40x60x40	40x20x40
Spheres	544	3245	2418
Map Nodes	2204	13920	10300
Map Edges	23157	223346	132405
Roadmap pre-c. [min.]	2.1	55	15
Visibility pre-c. [min.]	1.4	191	58
Memory [MB]	circa 4	circa 20	circa 18

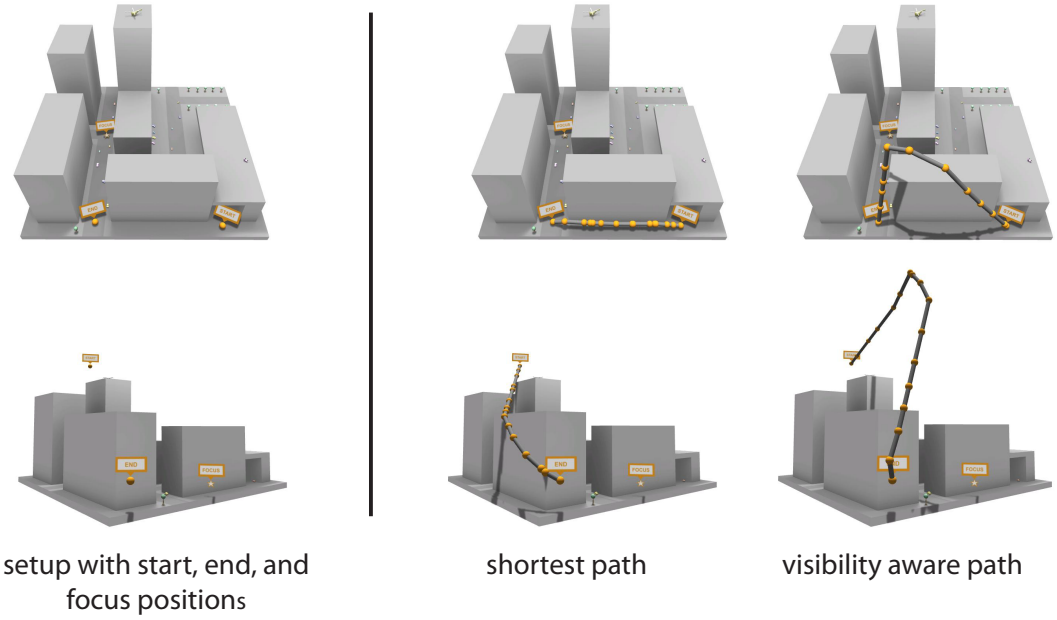


Figure 8.2.: Two examples of the difference between a shortest path and a visibility aware path are shown. On the left, the initial positions of start, end, and focus points are shown. The image on the right side show the paths that are generated using shortest path planning and visibility transition planning respectively.

8.2. Navigation in Urban Environments

As a first example application of the visibility transition planning, an interactive urban environment is simulated. The main characteristics of such application areas is that the environment consists of huge objects like buildings and skyscrapers that can block the view of specific objects of interest. The navigation in such environments becomes an interesting problem if a specific location needs to be found by a user in this environment. By simply showing this location the user has little information about where this place is located with respect to his or her own position.
Visibility transitions can be used to move the camera from the user's local view towards a shot that shows the object of interest. With the optimization for visibility, these transitions seek, by definition, the shortest path to the visibility region. In several situations this path deviates from the true shortest path. Figure 8.2 shows two examples where this is the case.

Visibility transitions can be used in urban environments to connect defined shots of objects to a connected camera work. Since the paths are computed in real-time, any two shots can be connected. The main advantage of a global visibility aware solution is that the optimization of transitions for visibility produces more intuitive camera movement that helps the user understand his or her context. Figure 8.3 shows a sequence of images from the camera's view. The top row was created using a visibility transition. In the bottom row the camera follows the shortest path. This example illustrates that camera transitions optimized for visibility also can be used to show specific objects or locations in their larger surroundings. The camera following the shortest path early looses its focus object while the camera following the visibility transition

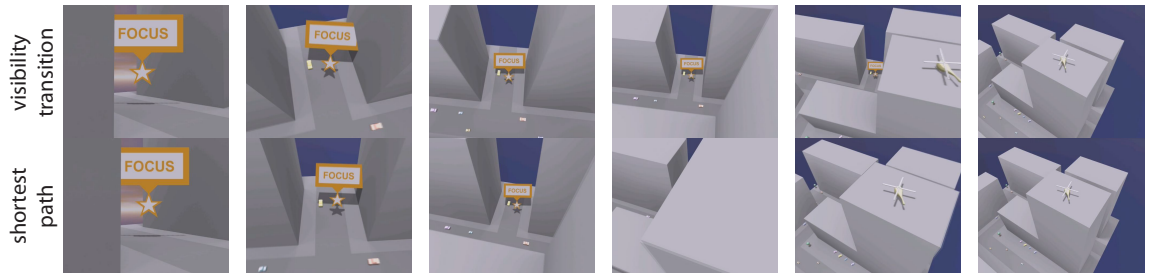


Figure 8.3.: This sequence of images show the difference between visibility transition and shortest path from the view of the camera. The focus point is visible significantly longer for the visibility transition shown in the upper row.

keeps the focus point visible significantly longer.

The computation time of visibility transitions is output sensitive. They heavily depend on the distance and graph complexity between the start and end positions. The search algorithm implemented for this thesis is a modified version of the A* search algorithm described in Dechters work [DP85]. Except for some fifo-lists no special data structures were used to optimize the search. Therefore, there is room to further improve the performance. Nonetheless, average large scale transitions, like the examples shown in figure 8.2 and 8.3, require only between 80 and 250 milliseconds on an Intel Core 2 at 2.4 GHz. These timings allows high interactivity. Other interactive applications could use this method to connect shots. An example is the work of Bares and his colleagues [BGL98] where cinematographic rules are applied in real-time to position the camera based on constraints. Visibility transition planning can be used to connect several of these shots in fluid sequence.

8.3. Dynamic Camera Control

In this setting, a third-person camera typically has many problems. Highly dynamic environments like computer games provide another interesting area in which our camera controller can be applied. As discussed in chapter 1, several problems arise when a camera is guided with local information. Especially if the character quickly moves behind corners, local camera models reach their limits.

A widely used local camera controller in computer games is based on a ray cast between the avatar and the camera. As soon as the view is obstructed, the camera is teleported to the first point on the ray that provides an occlusion free view. Such unnatural camera movement is avoided using the dynamic camera controller described in chapter 7. The controller dynamically updates a visibility transition between the current camera configuration and the desired camera configuration, as determined by the ray cast. This dynamic re-routing of the camera avoids the need for teleporting. The camera is automatically guided around corners to an unobstructed view. This dynamic adaption solves problematic situations more gracefully and preserves the natural feel of the camera. Figure 8.4 shows a comparison of the classical ray cast method and the dynamic visibility planning.

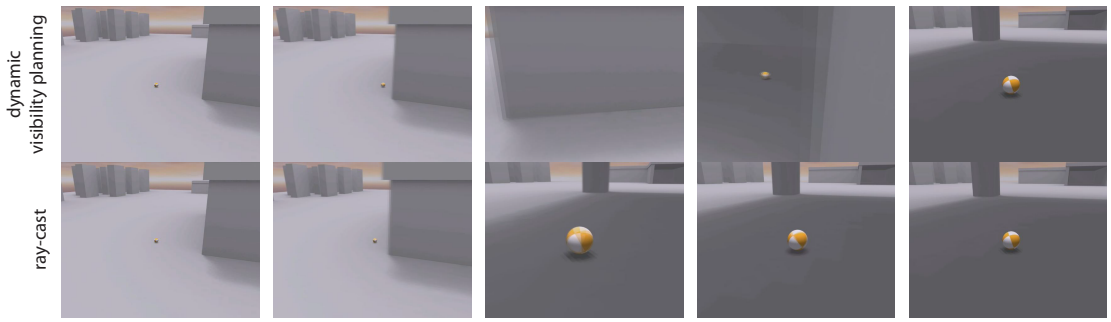


Figure 8.4.: This sequence shows a typical problem of local camera control: A player quickly moving behind a corner. While dynamic visibility planning guides the camera around the corner (shown in the top row) the ray cast method teleports the camera to an unobstructed view (shown in the bottom row).

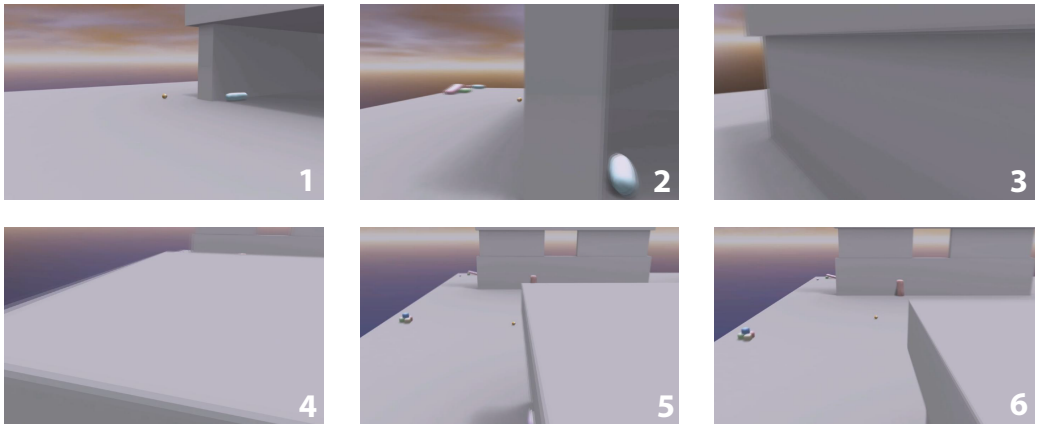


Figure 8.5.: This sequence shows how the dynamic camera controller automatically adapts to a situation and moves such that the camera catches up with the character.

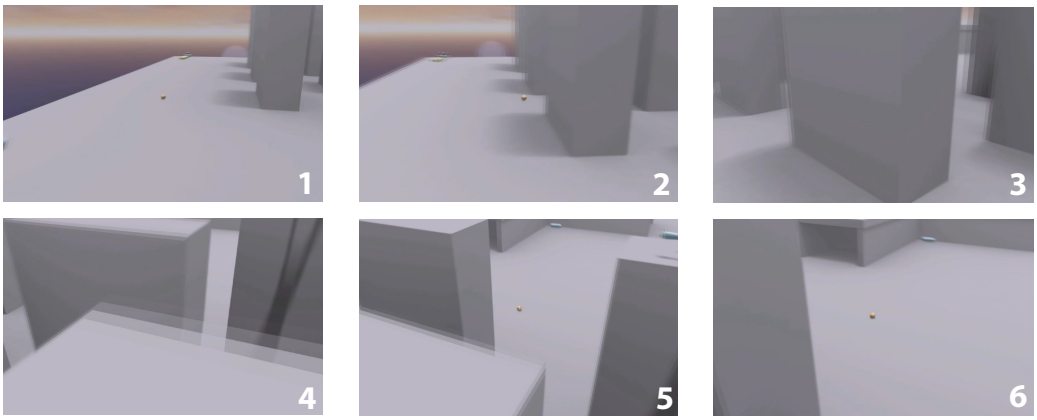


Figure 8.6.: In this example the player vanishes behind the columns. The dynamic camera controller adapts to this situation and moves up in order to spot the player between two columns in the back row.

Since the dynamic camera controller computes visibility-aware paths to circumnavigate obstacles, it automatically adapts to situations where it looses sight of the character. Because a visibility path minimizes the distance to the visibility region with high priority, it is able to take short cuts in order to catch up with the character. This behavior automatically leads to a more intuitive and realistic camera movement. Two examples of this effect are shown in Figures 8.5 and 8.6. These results are demonstrated via live captures in a video that accompanies this thesis.

9

Conclusion and Future Work

We have presented an algorithm that enables sophisticated camera control by phrasing the problem as a global search on a precomputed visibility roadmap together with local runtime refinement. The balance between pre-computation and runtime calculation allows our system to generate collision-free paths that maximize a notion of visibility while maintaining real-time performance. Because it is global in nature, our method can generate large-scale camera transitions between distant points in a scene, even in complicated geometric situations. It permits an adroit third-person camera model that avoids collisions and resolves occlusions in a natural fashion, regardless of how quickly the player moves or how complicated the visibility situation becomes. The computation time is output sensitive. Large camera transitions require tens of milliseconds, while shorter ones used in our third-person camera are computed in under one millisecond.

Since our method relies on the precomputed visibility roadmap for real-time performance, our system does not support dynamically changing scenes. The roadmap itself could reladively easy be adjusted to changes in the environment by canceling out affected nodes or connections. Adapting the visibility data, however, is trickier. Each sphere contains a list of visibility probabilities for all fully or partially visible spheres. Simply recomputing these values is too slow to be done in runtime. A possible solution to this problem could be to calculate some form of dependency relationships between spheres, so that visibility relationships can be updated more efficiently.

Currently, the transition planning only supports one focus point. This limitation is not a problem for the dynamic camera controller. The camera always follows the character and one focus point suffices to create adequate camera transitions. For navigation in urban environment, however, one focus point can be too limiting. Objects or locations of interest can be quite large. The approximation of these structures by only one point can result in camera transitions that only keep a portion of the structure in view. The design of the visibility roadmap, however, allows

approaches to solve this problem. Since visibility regions can be queried with linear cost, the search algorithm could be modified in order to optimize for several focus points. In each sphere with partial visibility to one or more of the given focus points, a series of occlusion maps can be computed. The path refinement computations could then superpose the given maps and optimize for several focus points at once.

Our system currently computes a fairly dense sampling of the ambient space using the roadmap spheres. Accurately sampling small regions of the environment may require tiny spheres. Such a sampling leads to more accurate visibility estimations and better camera paths. Especially in large spaces limiting the maximum radius of the spheres results in lessened discretization errors. For large environments, however, the number of roadmap nodes and edges can become very large. While this increase is not so much of a problem for the runtime search, it requires a lot of memory. We plan to explore level-of-detail for the visibility roadmap, where "highways" are traveled for large distances, and "local roads" are used for more fine-scale movement.
Another problem of the sphere sampling arises during the path post-processing. Although our sphere generation algorithm uses a score based on overlap areas it can happen that a path is selected that contains small portal circles. Such small circles decrease the adaptivity of the path during the iterative smoothing. This can result in undesired kinks in the final camera transition. While the physical camera model will smooth these kinks out, the camera movement may still look a little odd.
The problem of sampling free space with spheres is non-trivial. The cost function used in our approach may be sub-optimal in some cases. Future work includes exploring different cost functions to select spheres from the pre-computed ones on the grid. Also, a completely different approach may lead to an improved sampling. Randomized and genetic algorithms have proven to be very strong for similar problems.

Another source of improvement lies in the dynamic re-computation of the visibility transitions. The dynamic camera controller currently computes an entirely new path based on changed end- and focus-points. While this works well in applications where the camera needs to dynamically follow a character, it could be optimized on the path search level. The assumption could be made that a path only partially changes when the end- and focus-point change. Using such updated paths the iterative smoothing could only be applied on parts that have changed. This approach could decrease the overall CPU time of the transition planning and refinement.

Even though the dynamic camera controller provides impressive camera movement in situations where local methods typically fail, it still has some drawbacks. The controllability of the camera is limited. Widely used methods like a local ray cast provide excellent control of view direction. In many situations this controllability is highly demanded. The dynamic camera presented in this thesis is able to adopt view directions but not without delay. This delay is mainly because it is bound to travel on a path. If the view direction is required to change fast the camera needs some time to travel to the new location.
The dynamic camera controller in its presented form is very reactive. If the focus character is lost, the camera moves such that it catches up with the player. This behavior causes the character to vanish for a short period of time. In the future, the camera controller could be extended to learn the players movement style. Since the controller is designed as a parallelized state machine, it could easily be extended with new states that incorporate some form of intelligence. Depending on the position of the character, structures in its neighborhood, and the user's input, the camera could learn to react to problematic situations in advance. It might be possible to cre-

ate a camera controller that is able to learn a users play style at runtime and use this information to avoid difficult situations rather than to react when it is too late.

In this work, we have focused on incorporating visibility into the optimality criteria for transition planning, since achieving a non-occluded view of an object is the most fundamental goal of camera control. In future research, we wish to include higher-level goals such as cinematographic rules that influence perceptual aspects of the computed transitions to convey different styles or moods. One particularly exciting avenue of research is to generalize from artist-created camera paths during transition planning. This could be applied so that greater control over the character of the transitions is given to the user.

A

Geometric Constructions

The camera transition planning method presented in this paper optimizes paths for visibility based on an underlying spatial representation of free space. There are several parts in the approach that require algebraic computations to construct primitives in 3D. This appendix discusses such constructions in more depth and shows the mathematical background that is needed to identify these primitives.

A.1. Intersection circle of two overlapping spheres

We are given two spheres $S_1 = (c_1, r_1)$ and $S_2 = (c_2, r_2)$ with centers c_1, c_2 and radii r_1, r_2.

$$d = |c_2 - c_1| \tag{A.1}$$

If the distance d between the sphere centers is smaller than the sum of the two radii r_1 and r_2, the intersection of the spheres is a circle. A circle $C = (p_c, n_c, r_c)$ in 3D is defined by a position p_c, a normal vector n_c, and a radius r_c. These parameters can be computed using the relations shown in figure A.1. The intersection circle C is orthogonal to the line between c_1 and c_2. Therefore, the normal n_c of the circle plane is the normalized direction vector between these two points.

$$n_c = \frac{c_2 - c_1}{d} \tag{A.2}$$

The circle position p_c lies on the line defined by c_1 and c_2. To find the exact position on this line the following system of 3 equations and 3 unknown is solved:

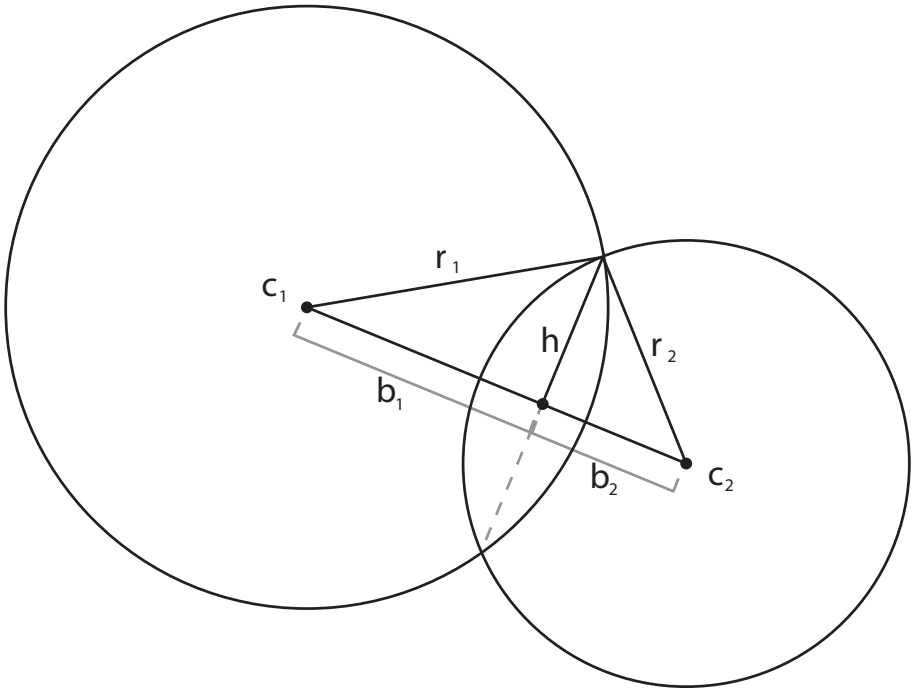


Figure A.1.: Geometric construction of two overlapping spheres.

$$b_1^2 + h^2 = r_1^2 \tag{A.3}$$

$$b_2^2 + h^2 = r_2^2 \tag{A.4}$$

$$b_1 + b_2 = d. \tag{A.5}$$

Equations A.3 and A.4 are the triangular relations between the bases b_1, b_2 and the radii r_1, r_2 of the spheres with the radius r_c of the intersection circle. Equation A.5 is the relationship between the bases and the distance d.

Subtracting equation A.3 from equation A.4 results in the following equation:

$$r_1^2 - b_1^2 = r_2^2 - b_2^2 \tag{A.6}$$

The parameter b_2 can now be substituted by $d - b_1$ according to equation A.5. This substitution leads to the following equation:

$$r_1^2 - b_1^2 = r_2^2 - (d - b_1)^2. \tag{A.7}$$

This equation can now be solved for b_1:

$$b_1 = \frac{r_1^2 - r_2^2 + d^2}{2d}. \tag{A.8}$$

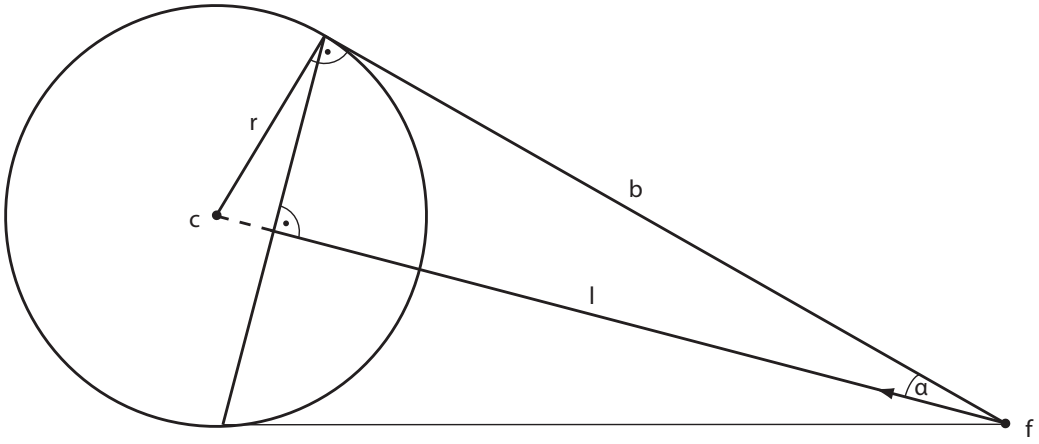


Figure A.2.: This figure shows the setup of a view frustum that lies on a sphere.

The base b_1 can be interpreted as the parameter on the ray defined by c_1 and n_c where the circle plane is intersected. Therefore, the circle center p_c can be constructed by adding the normal n_c scaled by b_1 to c_1.

$$p_c = c_1 + b_1 n_c \tag{A.9}$$

The last parameter to uniquely identify the intersection circle is its radius r_c. This parameter can be computed by exploiting the relation between b_1, r_1 and r_c.

$$h = \sqrt{r_1^2 - b_1^2} \tag{A.10}$$

A.2. Point-Sphere projection

To create a projection of the geometry between a focus point f and a sphere S to a plane, the homogenity of the sphere can be explointed. A sphere can, from all directions, be fit into a box. Therefore, there exists always a square projection plane that intersects the entire sphere. Figure A.2 shows how a perspectie viewing volume around the sphere can be defined such that the focal point coincides with f and the space between S and f is captured.

To uniquely define a perspectie projection in space, the projection center p, projection direction v, aspect ratio a, opening angle α, near distance d_{near} and far distance d_{far} are needed (figure A.3).

The projection center and the aspect ratio are defined by the setup:

$$p = f \tag{A.11}$$

$$a = 1. \tag{A.12}$$

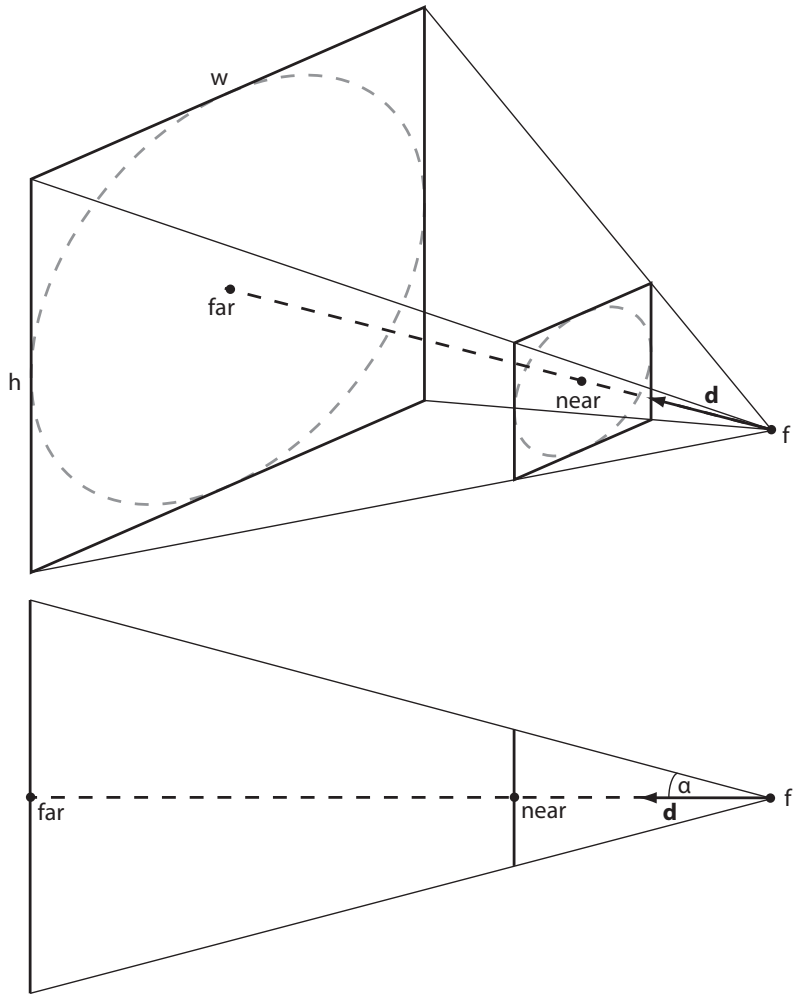


Figure A.3.: This figure illustrates the parameters that are needed to define a view frustum.

The projection direction v can be computed using the sphere center c:

$$v = \frac{c - p}{|c - p|}. \tag{A.13}$$

To specify the opening angle, the trigonometric relation between the sphere radius r and the distance between the projection point p and the sphere center c can be exploited:

$$d = |c - p| \tag{A.14}$$

$$\alpha = \sin^{-1}\left(\frac{r}{d}\right). \tag{A.15}$$

The near plane distance d_{near} of the projection should be zero. This value, however, is not possible since the perspectie projection requires d_{near} to be greater than zero. Additionally, a very small value results in numerical inaccuracies in both the projection and depth values of the projection. For our implementation values between 1 and 0.1 has shown to work well. We use $d_{near} = 0.5$.

The far distance d_{far} can be calculated using the trigonometric relation between α and the base distance b. The base shares an edge in an orthogonal triangle with r and d between c and p. Therefore, b can be computed by applying the Pythagorean theorem:

$$b = \sqrt{(|c - p|)^2 - r^2} \tag{A.16}$$

$$d_{far} = b \cos(\alpha). \tag{A.17}$$

Bibliography

[BGL98] William H. Bares, Joël P. Grégoire, and James C. Lester. Realtime constraint-based cinematography for complex interactive 3d worlds. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 1101–1106, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[Bit02] J. Bittner. *Hierarchical Techniques for Visibility Computations*. PhD thesis, 2002.

[BMH98] David C. Brogan, Ronald A. Metoyer, and Jessica K. Hodgins. Dynamically simulated characters in virtual environments. *IEEE Comput. Graph. Appl.*, 18(5):58–69, 1998.

[CG98] Dirceu Cavendish and Mario Gerla. Internet qos routing using the bellman-ford algorithm. In *HPN*, pages 627–646, 1998.

[COCS00] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications, 2000.

[DDTP00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 239–248. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[DGZ92] Steven M. Drucker, Tinsley A. Galyean, and David Zeltzer. CINEMA: A system for procedural camera movements. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics, Special Issue of Computer Graphics, Vol. 26*, pages 67–70, 1992.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the

optimality of A*. *J. ACM*, 32(3):505–536, 1985.

[Dru] S.M Drucker. Intelligent camera control in graphical environments, PhD Thesis, 1994, Massachusetts Institute of Technology, Cambridge, ma.

[DZ94] Steven M. Drucker and David Zeltzer. Intelligent camera control in a virtual environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff, Alberta, Canada, 1994.

[HHS01] Nicolas Halper, Ralf Helbing, and Thomas Strothotte. A camera engine for computer games: Managing the trade-off between constraint satisfaction and frame coherence. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 174–183. Blackwell Publishing, 2001.

[HLT03] Alexander Hornung, Gerhard Lakemeyer, and Georg Trogemann. An autonomous real-time camera agent for interactive narratives and games. In *IVA 2003 - Fourth International Working Conference on Intelligent Virtual Agents*, 2003.

[HM] Nick Halper and Maic Masuch. Action summary for computer games.

[KKS+05] Azam Khan, Ben Komalo, Jos Stam, George Fitzmaurice, and Gordon Kurtenbach. Hovercam: interactive 3d navigation for proximal object inspection. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 73–80, New York, NY, USA, 2005. ACM.

[KM02] Kevin Kennedy and Robert E. Mercer. Planning animation cinematography and shot structure to communicate theme and mood. In *SMARTGRAPH '02: Proceedings of the 2nd international symposium on Smart graphics*, pages 1–8, New York, NY, USA, 2002. ACM.

[LaV] S. M. LaValle. *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/). To be published in 2006.

[MC02] E. Marchand and N. Courty. Controlling a camera in a virtual environment. *The Visual Computer Journal*, 18(1):1–19, 2002.

[NRG04] Christoph Niederberger, Dejan Radovic, and Markus Gross. Generic path planning for real-time applications. In *CGI '04: Proceedings of the Computer Graphics International (CGI'04)*, pages 299–306, Washington, DC, USA, 2004. IEEE Computer Society.

[RWHK97] Ulrich Roth, Marc Walker, Arne Hilmann, and Heinrich Klar. Dynamic path planning with spiking neural networks. In *IWANN*, pages 1355–1363, 1997.

[SGLM03] Brian Salomon, Maxim Garber, Ming C. Lin, and Dinesh Manocha. Interactive navigation in complex environments using path planning. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 41–50, New York, NY, USA, 2003. ACM.

[TBN00] Bill Tomlinson, Bruce Blumberg, and Delphine Nain. Expressive autonomous cinematography for interactive virtual environments. In Carles Sierra, Maria Gini, and Jeffrey S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 317–324, Barcelona, Catalonia, Spain, 2000.

ACM Press.

[wHCS96] Li wei He, Michael F. Cohen, and David H. Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. *Computer Graphics*, 30 Annual Conference Series: 217–224, 1996.

[Woo] Wayne Wooten. Transitions between dynamically simulated motions: Leaping, tumbling, landing, and balancing.

[WWS01] Peter Wonka, Michael Wimmer, and François X. Sillion. Instant visibility. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 411–421. Blackwell Publishing, 2001.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics*, 31 Annual Conference Series: 77–88, 1997.