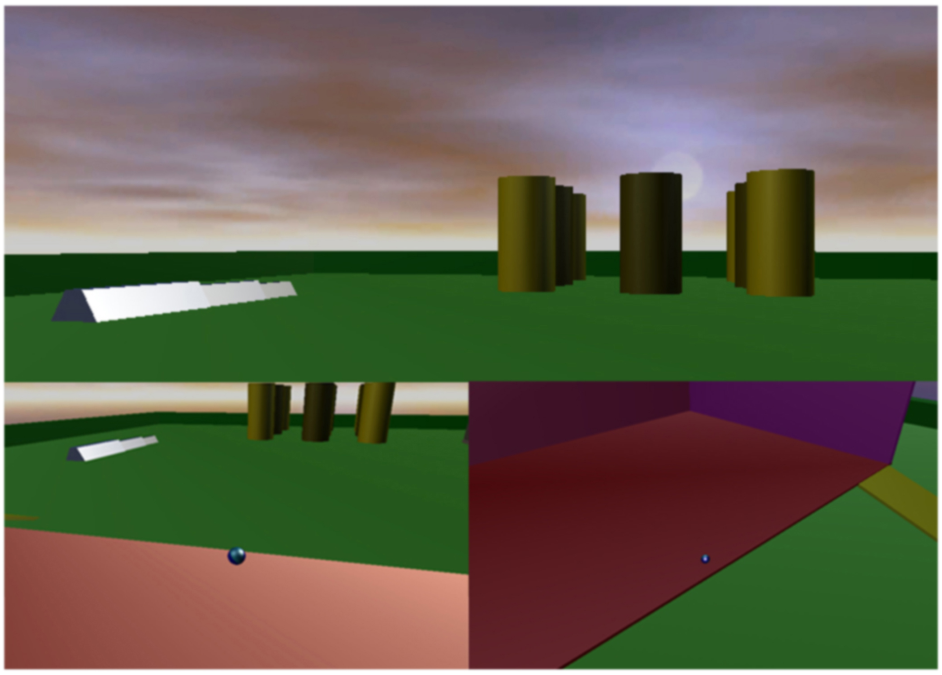
Reusable Game Camera for 3D Environments

Thomas Oskam

July 17, 2007



Abstract

The variety of computer game genres has grown very fast over the last decade. Environments have become bigger and more complex. Also the intricacy of game play has increased along with the geometry in a game scene. One problem, which comes along with this, is the control of the virtual camera. Different situations arise where a simple approach may fail when several properties of the camera, like low-frequency movement or penetration avoidance, need to be achieved. Also different types of games require different kinds of camera movements and controls to enhance the game play. Three types of game cameras are wide spread and cover most of the genres: First person camera, third person camera, and isometric camera.

This work presents a method to encapsulate a camera module that is capable of performing the three main camera types simultaneously and still maintains a clean and easy to use interface. A controller is shown which can be integrated into the game loop, consisting of different camera control routines by which a physical camera model is guided through the environment. A variant of the camera simulation loop is also introduced, which provides an interface for ray casting and bounding volume tests without the camera knowing about the surrounding geometry. This extended controller is exploited to make the camera automatically avoid line of sight blocker and collisions depending on the active camera control routine.

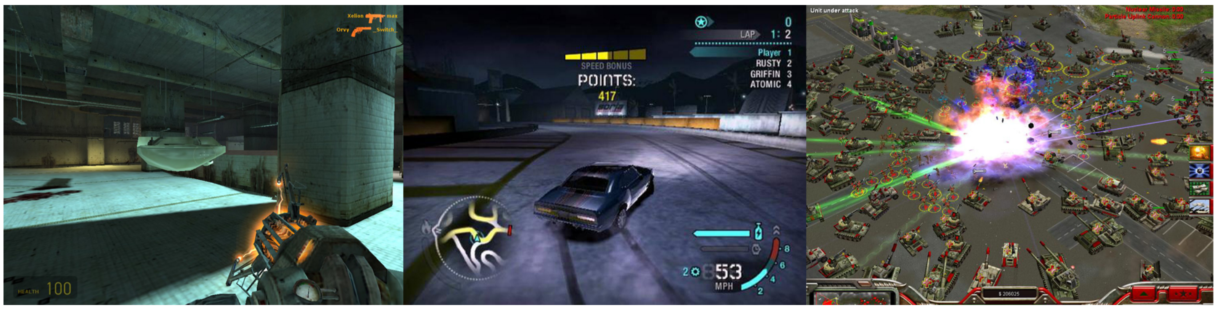
Contents

List of Figures

1 Introduction

Over the last years, computer and console game development has consistently increased. The technical aspects became more complex as well as the content itself. These days, professional games are created by a large number of people coming from different backgrounds like programming, design, or music. This implies, on one hand, that a game engine consists of many different modules that abstract to a high-level encapsulation of several processes. Most low-level applications, which used to be the focus in visual computing, are almost completely hidden. The know-how about algorithms for special tasks also grew through the steep increase in game production, and a variety of good solutions for some of these tasks emerged. On the other hand, the demand for better quality and for generic behavior of these modules has grown along with the abstraction. An underlying engine is often developed independently of the game and then improved and reused for sequels or other games. This has lead to an almost complete separation of the technical aspects of a game, e.g. game play, physics, and sound processing, from the content like graphics, music, and design.

One of the most important features of a game is a good camera. As the complexity of game scenes and the variety of game genres has grown, different camera styles have emerged as well. However the programming interface to create a camera is, in most development kits or libraries, quite limited. There is usually support for the creation of the view- and projection matrix, but this alone is not sufficient to have a reusable camera, as no generic functionality is supplied. In the different 3D game genres, that evolved over the last decade, three types of cameras are wide spread and cover most of the games. The so called first person camera is often used in shooter and action games. The main property of this camera is to simulate a head mounted display. It moves with the game character through the environment as if the player would walk through it himself. The second style is a third person view where the camera has to follow the game character. The camera behaves, like the name implies, as a third person in the environment filming the player. This style is found in most games from racing to role-playing and has a wide



First Person Camera Third Person Camera Isometric Camera

Figure 1.1.: Examples of games using different camera styles. The picture on the left shows a scene from Half Life 2 where a first person camera is used. In the middle you an image from Need For Speed Carbon using a third person camera is shown. The picture on the right shows the isometric view from Command And Conquer Generals.

use. A third widespread view style is the isometric or top-down camera. It is a fixed camera, which observes the game progress from a neutral distance. The isometric view is mostly used in games where the player is in control of a large number of units or has to keep track of a huge area. This style is often found in strategy games or construction games. Figure 1.1 shows examples of the three camera styles.

Besides the different movement patterns, the camera is in general a part of the game that has to remain unnoticed. If the camera is working well, the user will not be directly aware of its presence. If it behaves unnaturally, however, this will instantly annoy the user. Such undesired behavior is mostly introduced by high frequency movement that is not physically possible in the real world, such as jumping instantly from one position to a new one. Another problem that introduces undesired camera behavior is the penetration of geometry. Especially in 3D games, where the environment may be arbitrarily complex, these side effects can occur very often if not handled explicitly. Similar to penetrations, occlusion of the camera by objects will be annoying for the user. It is thus crucial to avoid this effect in order to create a well behaving camera. One of the most challenging aspects here is to create a computationally fast camera which is, at the same time, flexible enough to handle dynamically changing environments.

This thesis shows a way to encapsulate a reusable game camera for 3D environments in a clear interface, which can handle the three main camera styles, and guarantees certain properties to avoid unnatural behavior. The camera itself is modeled as a physical entity where forces are applied to guide it towards a desired position. An adapted version of the controller introduced in [Woo98] is used to compute a desired configuration, towards which the camera object is driven using forward dynamics.
Furthermore, an analysis of different situations that can cause conflicts with the geometry of the environment is discussed, and a solution is presented, which resolves most of them, while maintaining the interface small and the computation time low.

2

Previous Work

There has been a lot of research in the area of camera control in virtual environments. However, not only the encapsulation of the camera itself is important, but also techniques that handle occlusions and object avoidance. A well behaving camera in free space can rapid lead to several problems when having to deal with obstacles. Advanced methods of both visibility and path planning are of interest to resolve such problems. These fields have also been studied well in the past and there exist several approaches that can be interesting for camera control as well. Therefore the following subsections discuss related work from the fields of camera control, visibility, and path planning in more detail.

2.1. Camera Modules

Drucker and Zeltzer [DZ94] presented a framework for camera movements in virtual environments. They created a system for handling a number of cameras to enable different cinematic shots in a scene, which built up on their previous work [Dru] and [DGZ92]. They described a system for specifying behaviors for virtual cameras in terms of task level goals and constraints. Each camera contains a controller and a set of constraints that need to be met. The constraints describe the camera behavior in terms of up to 7 degrees of freedom (3 for position, 3 for orientation and field of view). The essential part of this work is the path planning that mainly was drawn from robotics. In a first step, a path graph is created off-line in a known environment by performing visibility tests. A first path then is computed using the known A* algorithm to traverse the path graph. In a second step, a room to room planning algorithm is used to create paths that avoid obstacles and lead to objects that are of interest. The system was implemented as camera control within a virtual museum and handles mainly the movement of the camera through different rooms. The main drawback of this system for our purpose is, that it is designed to create constraint based paths through an environment by using sophisticated optimizations. These optimizations are too complex to be applicable in real time and therefore have to be precomputed.

Halper et al. [HHS01] developed a game camera engine using several specifications that were made as extension to cinematic camera work. They emphasized the fact that the handling of game cameras is quite different from movies, as camera work in games has to react in realtime, and has no way to re-shoot scenes. The presented engine consists of a director module and a predictive camera planner. The director module has libraries of several templates for different moods and shots, which consist of hard and soft constraints. A template selector chooses the appropriate sets. They also developed a constraint solver that is able to adapt or neglect some constraints, because in some situations, the computed positions may not be desired. Occlusions are resolved by a separate shadow rendering for each point. Through reduction of the shadow map down to 32x32 pixel, the method is fairly fast and resolves different geometrical situations. The second part, the camera planner, makes assumptions about future configurations of the camera using the acceleration parameter of past frames. These future positions are also processed using the constraint solver, to predict occlusions ahead of time.

Hornung, Lakemeyer and Trogemann's work [HLT03] on an autonomous real time camera agent mainly targeted to incorporate different cinematic rules for shooting a scene into a module. The method integrates a camera together with a shot library into an autonomous agent, which is integrated in the scene. The main target of this work was to create a real time system that automatically shoots storytelling within a game. The system chooses the camera work based on calls from virtual actors when they get active.
Bares et al. [BGL98] developed a similar system that is able to capture movie shots of scenes with different actors using a constraint solver.
As these methods were designed to automatically show storytelling in a fashion, we are used to from movies or television, it is, however, not suited for interactive gaming and aims more on storytelling elements in between.

Pickering [Pic] presents an alternative method of describing camera behavior. Instead of expressing camera behavior and settings with constraints, he uses a declarative approach, which is based on the experience of photographers describing their scenes. He showed that an image description language can be created, that is rich enough to capture recognizable styles of photography, and can be translated into an optimization problem. Through the generation of spatial constraints using space optimizing structures such as Oct- or BSP trees, the optimal camera motion can be computed.

The basic idea, that mostly influenced the work in this thesis, is the use of constraints to define the camera behavior. This was introduced by the work of Drucker and Zeltzer, and used in most work on camera control. Also the idea to directly incorporate different camera behavior pattern, as introduced in the work of Halper et al., is a core property of the camera module presented in this thesis.

2.2. Visibility

When creating a camera that follows a game character in an unknown 3D space, visibility also plays an important role. Through defined movement, depending on constraints, different situations can arise, where the camera-player line of sight gets occluded. To resolve such situations, different approaches to check for visibility may be viable. Although none of these methods were used in this thesis, they give some ideas to improve the camera control in future work.

Zhang et al. [ZMHH97] introduced 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 map 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 indicator to decide if an object is occluded from a certain point of view.

Durand, Drettakis, Thollot and Puech [DDTP00] presented a visibility preprocessing method which computes potentially visible geometry for volumetric viewing cells. They introduced novel extended projection operators, which permits efficient and conservative occlusion culling with respect to all viewpoints within this cell. The method uses 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 arbitrary complex environments.

Wonka, Wimmer and Sillion [WWS01] presented an online occlusion culling system which computes visibility in parallel to the rendering pipeline. They showed 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 essentially combines the advantages of online visibility processing and region-based visibility calculations, allowing asynchronous processing of visibility and display perations.

The method that is used in this work to perform visibility tests is a simple ray cast. As it may not be as strong in severals situations as more sophisticated visibility tests, it has the advantage of simplicity. Chapter 4 discusses in depth different situations that can lead to occlusion and proposes a ray cast approach, that guarantees the camera never to stay in an occluded state.

2.3. Path Planning

Occlusions can indeed be detected using one or more visibility tests. To some degree, these approaches also can resolve occlusions as well. However, to include a certain degree of intelligence into a camera module, it is indispensable to have some kind of look-ahead mechanism. Several situations can arise where the camera, despite visibility checks and collision response, can get stuck or occluded. In such cases, a path planning approach may be a good point to start. Years of research, mainly in the field of robotics, resulted in a variety of path planning algorithms with different spatial constraints. The approaches discussed in this sub section is an assortment of different approaches that may be interesting to be used for advanced camera control.

Roth, Walker, Hilmann and Klar [RWHK97] presented a path planning algorithm, the radar path planner, which is capable of performing in unknown, partially known, or changing environments. Their main goal was to find a solution for situations where previously planned paths, through unknown or changing environment, have to be re-planned. They presented the radar path planner, which is based on a simple idea: A starting point and a target point are given in the workspace. In regular intervals, the target point sends out a wave front traversing with constant speed. The wave front cannot move through obstacles. The start point itself has several neighbor guards that can detect the wavefront. The guard, that is first reached by the wavefront, must be part of an optimal path. This guard then becomes the new start point and the procedure is repeated until the target point is reached.

Sugihara and Smith [SS97] addressed adaptive motion planning which can modify the existing path whenever an environmental change occurs. They proposed the use of genetic algorithms to compute optimal paths from a start point to a destination point. A way to efficiently encode a monotone path on a grid into a binary string of constant length was shown. Observations indicated that a simulation with a population of about 30 agents, paths can be generated that are nearly optimal. Through the fixed-length of the binary path representation, the performance of the approach is extremely good. The adaptive nature of genetic algorithms together with the speed of the simulation allow for dynamic path computations in a realtime system where the environment may constantly change.

Nissoux, Simeon and Laumond [NSL99] introduced visibility roadmaps as a variant of probabilistic roadmaps ([KcL94] and [OS95]) to compute roads of configurations for robotics from a start configuration to an end configuration. They exploit visibility from different guard positions between two configurations to get collision free guidance of several degrees of freedom through obstacles.

Simhon and Dudek [SD03] presented a method for robot path planning based on learning motion patterns. A motion pattern is defined as the path that results from applying a set of probabilistic constraints to a ŞrawŤ input path. For example, a user can sketch an approximate path for a robot without considering issues such as bounded radius of curvature. The system would then elaborate it to include such a constraint. In this approach, the constraints that generate a path are learned by capturing the statistical properties of a set of training examples using supervised learning. This learned distribution is then used to synthesize a preferred path from an arbitrary input path by choosing some mixture of the training set biases, that produce the maximum likelihood estimate.

Helbing and Strothotte [HS] proposed a camera path planing algorithm that is designed as a trade off between an optimal path and computational complexity. The goal is to guide the camera from its current position to a configuration from which a specified object is captured. The first part of the algorithm searches for a position that displays the chosen object in a desired way. This is the case, when the projection onto the image plane occludes approximately half of it. A problem that arises at this point is to find a camera configuration that is not occluded by other objects. In the second part, they described a path planning algorithm that guides the camera from its current configuration to the target configuration. The algorithm starts with the straight line between current and target position. When this line is intersected by an obstacle, a backtracking mechanism is used. By subdividing the connection line at the intersection points a path is found that leads to the target position without penetrating geometry. In a post processing step the high frequency corners of the path are smoothed out using Hermite curves. Through the recursive structure of this method the runtime can not be guaranteed to be fast but, in average case, it allows real time interaction. In distressed cases, however, this can lead to long computations and result in inordinate complex paths.

3 Reusable Camera Module

This chapter discusses the general setting and lists several key properties for a reusable game camera, which need to be integrated. The second part shows a camera controller, that can handle different camera movement styles as well as maintaining certain properties.

3.1. General Problem

Current 3D games mostly make use of one ore more of the following camera styles; First person camera (FPC), third person camera (TPC) and isometric camera (IC). These styles each follow different dynamics and constraints. For example a TPC has to maintain a constant distance to the player and a constant angle to the ground. A FPC, however, can simply be set at the player's position, and has to focus on a point of interest. The IC is quite similar to the TPC as it follows the player. But, in contrast, the view direction and camera orientation remain constant and are independent of the player orientation. One key property of a reusable game camera therefore needs to be an internal structure that is capable of handling these different constraint sets, and can be extended by new styles as well.

Regardless, some basic properties are desired for all three camera styles. One of the most importatnt is smooth movement. High-frequency motion, which is not possible in the real world, will attract undesired attention. These movement patterns, like instantly changing the camera position, hence have to be filtered.

Another property that needs to be taken into account is the programming interface of the camera module. As the goal is to create a reusable camera, which is applicable in a wide variety of games, the interface must be small and enclosed. Is should be as easy as possible to integrate the camera module into a game engine, but without hiding key functionalities behind the interface.

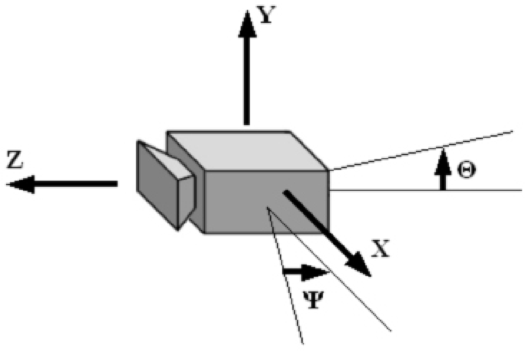


Figure 3.1.: The camera with its 5 DoF: X, Y and Z for the position in space. Ψ for the camera pitch and θ for the camera yaw. The up vector is assumed constant and thus the third orientational DoF drops out.

Furthermore, the performance is an important factor as well. Games are one of the most demanding applications when it comes to fast algorithms. Physics, artificial intelligence, and especially graphics consume a lot of CPU and GPU time. Therefore, the techniques used in the camera module must be as computationally cheap as possible in order to keep the CPU free for the game and physics update logic.

3.2. Physical Camera Model

Typical game camera work consists of 5 degrees of freedom (DoF); Three DoF for the position of the camera in space and two DoF for the orientation. The up-vector of the camera is considered constant. Figure 3.1 shows the DoF used in this approach in detail.
To create a realistic and smooth motion, the camera can be modeled as a physical object with mass and friction. By applying forces along the positional DoF, and torques to the orientation DoF respectively, the camera is moved similar to a real camera. With this approach, undesired high frequency motion is automatically filtered.

3.2.1. Dynamic Simulation

The camera control system can be interpreted as a dynamic simulation, which can easily be integrated into the main game loop. Figure 3.2 shows a scheme of the underlying state machine. In the simulation, the camera has a configuration that defines the view transformation for the current time step. The camera state and the system input (player controls) are passed to a controller, which computes the forces and torques that need to be applied to the different DoF in order to move the camera towards a desired configuration. The desired configuration in turn depends on the camera mode. In a further step, the forces are planted to the physical camera and, through numerical integration, the new camera configuration is calculated.

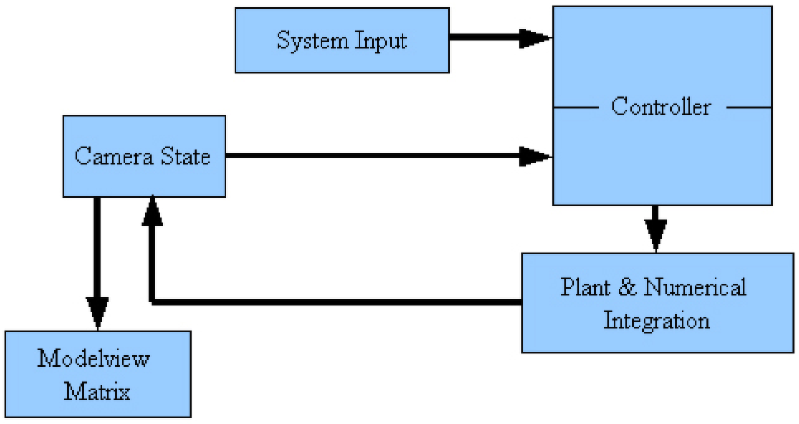


Figure 3.2.: Scheme of the dynamic camera simulation.

3.3. Camera controller

The camera is guided by a controller, which calculates the necessary forces for the different DoF. The controller itself consists of two layers (see Figure 3.3). The first layer contains different control routines, one for each of the three main camera styles discussed above. The active control routine takes player input to compute the desired camera position and orientation. This desired configuration is then, together with the current camera configuration, given to the second layer; The servo unit. This unit calculates forces that need to be applied to the current configuration, using forward dynamics, to move the camera towards the desired configuration. The calculated forces are planted and, through numerical integration, the new camera configuration is calculated. This leads to a flexible behavior as the active camera mode can be specified by the user at any point in the game.

At the beginning of the next simulation loop, the previously computed configuration is the current camera configuration in this time frame. Even if the control routine has changed, the camera will not move with high frequency because the applied forces just change towards the new desired configuration. This approach always drives the camera smoothly to its desired configuration, and the high frequencies are filtered out automatically.

The camera controller is designed similar to controller introduced in [Woo98], where it is used to calculate different human movement pattern that also may change over time. This allows for further extensions by new camera control routines, which can implement a set of constraints for a new camera style.

3.4. Control Routine

The different camera behaviors are realized through control routines, which are the integral part of the camera controller. In the beginning of a simulation loop one control routine, specified by the user, is active. The active routine delivers a desired configuration for the camera, which is the position and orientation of the camera depending on the position and orientation of the player

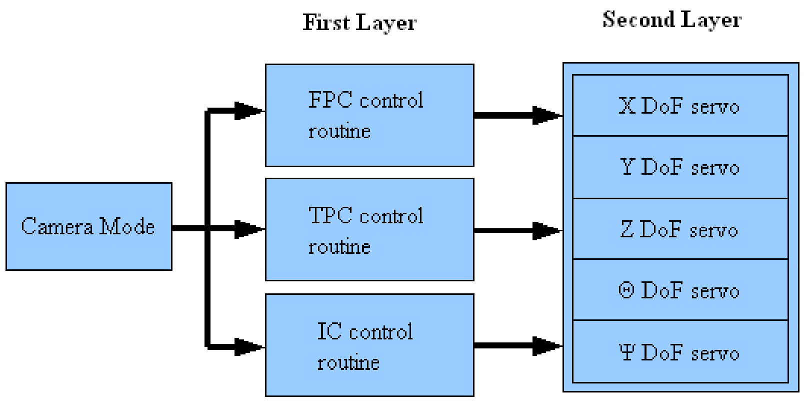


Figure 3.3.: This figure shows a scheme of the camera controller. The first level consists of three control routines, which define the movement for each of the three main camera styles. The second layer computes forces and torques for the five DoF. The camera mode is selected by the user and may change at any time.

it is connected to. In the following subsections the computation of the desired configuration for each of the three camera styles is described in detail.

3.4.1. First Person Camera

The constraints for the first person camera are pretty straightforward. The camera itself depends directly on the player position. Therefore, the desired value of the camera position $P = (X, Y, Z)$ is equal to the player position $P_{player} = (p_x, p_y, p_z)$. The camera orientation only depends on a virtual point, at which the user is looking. This point lies in the line of the player orientation $O_{player} = (o_x, o_y, o_z)$. The 5 camera parameter for the desired configuration depending on P_{player} and O_{player} can thus be expressed as follows:

$$P = P_{player} = (p_x, p_y, p_z) \tag{3.1}$$

$$\Psi = \cos(\langle O^{xz}_{player}, (1, 0, 0)\rangle) \tag{3.2}$$

$$\theta = \tan(\frac{\sqrt{o_x^2 * o_z^2}}{o_y}) \tag{3.3}$$

The projection of the player position onto the x-z plane O^{xz}_{player} is defined as follows:

$$O^{xz}_{player} = \frac{(o_x, 0, o_z)}{\sqrt{o_x^2 * o_z^2}} \tag{3.4}$$

If the camera needs to have an offset $P_{off} = (x_{off}, y_{off}, z_{off})$ of the player position P_{player}, the equation 3.1 can be extended by adding P_{off}:

$$P = P_{player} + P_{off} = (p_x + x_{off}, p_y + y_{off}, p_z + z_{off}) \tag{3.5}$$

3.4.2. Third Person Camera

In contrast to a first person view, the third person camera movement depends only indirectly on the player position. It is desired that the camera maintains a constant distance d_{tpc} to the player, as well as a constant angle α_{tpc} to the top down projection of the player orientation. The 5 DoF can be computed as follows:

$$\begin{aligned} P &= P_{player} + D = (p_x + d_x, p_y + d_y, p_z + d_z) && (3.6)\\ \Psi &= \cos(\langle O^{xz}, (1,0,0)\rangle) && (3.7)\\ \theta &= -\alpha_{tpc} && (3.8)\end{aligned}$$

The vector $D = (d_x, d_y, d_z)$ is the offset vector from the player to the camera. It can be computed by rotating $D' = (d'_x, d'_y, d'_z)$ around the up vector $(0,1,0)$ by Ψ degrees:

$$\begin{aligned} D' &= \frac{(-1, \tan(\alpha_{tpc}), 0) * d_{tpc}}{\sqrt{1 + (\tan(\alpha_{tpc}))^2}} && (3.9)\\ D &= \begin{bmatrix} \cos(\Psi) & 0 & -\sin(\Psi) \\ 0 & 1 & 0 \\ \sin(\Psi) & 0 & \cos(\Psi) \end{bmatrix} * \begin{pmatrix} d'_x \\ d'_y \\ d'_z \end{pmatrix} && (3.10)\end{aligned}$$

3.4.3. Isometric Camera

The Isometric camera is quite similar to the third person camera, as it has to maintain a constant distance d_{ic} to the player. The angle of the camera, however, should be constant too. Therefore the camera orientation is independent of the player orientation. This relation can be expressed through a look at vector $L_{ic} = (l_x, l_y, l_z)$, which defines the direction in which the camera is looking at the player. The camera configuration can thus be calculated as follows:

$$\begin{aligned} P &= P_{player} + U = (p_x + u_x, p_y + u_y, p_z + u_z) && (3.11)\\ \Psi &= \cos(\langle U^{xz}, (1,0,0)\rangle) && (3.12)\\ \theta &= acos(\langle U^{xz}, L_{ic}\rangle) && (3.13)\end{aligned}$$

using the offset vector U from player to camera and the vector U^{xz}, the projection of U onto the x-z plane:

$$U = (u_x, u_y, u_z) = L_{ic} * -d_{ic} \tag{3.14}$$

$$U^{xz} = \frac{(u_x, 0, u_z)}{\sqrt{u_x^2 * u_z^2}} \tag{3.15}$$

3.5. Computation of Forces and Numerical Integration

In the first layer of the controller, the control routines compute the desired configuration for the camera. The second layer computes forces that move the camera from its current position and orientation towards the desired configuration. The algorithm to perform this task should be computationally simple as performance is crucial. Forward dynamics provide a method for calculating appropriate forces in an acceptable computational complexity. Also the property, that this algorithm only uses information from the current point in time, fits well into the simulation loop.

3.5.1. Forward Dynamics for Force Computation

The task of computing appropriate forces is performed by the servo unit. This is independent of the previous steps, and thus allows a generic processing of the input. To compute the forces, two parameter are needed; the current camera configuration and the previously computed desired camera configuration.

Given the current camera position $P_{cur} = (X_{cur}, Y_{cur}, Z_{cur})$, pitch ψ_{cur} and yaw θ_{cur} as well as the desired configuration parameter $P_{des} = (X_{des}, Y_{des}, Z_{des})$, ψ_{des} and θ_{des} the positional force F_{pos} can be computed by

$$F_{pos} = k_{pos} * (P_{des} - P_{cur}) \tag{3.16}$$

the yaw force F_θ by

$$F_\theta = \begin{cases} k_\theta * (\theta_{des} - \theta_{cur}) & \text{if } \theta_{des} - \theta_{cur} \leq 180 \\ k_\theta * (\theta_{des} - \theta_{cur}) - 360 & \text{otherwise} \end{cases} \tag{3.17}$$

The calculation of the pitch force F_Ψ is not as straight forward as the yaw force F_θ. In contrast to the yaw angle, the pitch angle is not limited to values between [-90;90]. This can cause problems because in situations, where the difference between Ψ_{cur} and Ψ_{des} exceeds 180 degrees, the direction in which the forces should be applied can not be determined anymore. To resolve this problem, a third parameter D_Ψ is needed, which stores the current pitch direction of the player. $D_\Psi < 0$ indicates a current pitch rotation in negative direction and $D_\Psi \geq 0$ a rotation in positive direction.

The direction in which D_Ψ is changing is determined by the player orientation change. Therefore D_Ψ can be computed at the beginning of the camera module update by evaluating the difference between the current and the old player orientation.

Using the parameter D_Ψ as indicator for the direction, the pitch force can be computed as follows:
if $D_\Psi \geq 0$

$$F_\Psi = \begin{cases} ((\Psi_{des} + 360) - \Psi_{cur}) & \text{if } \Psi_{des} - \Psi_{cur} \leq 180 \\ (\Psi_{des} - \Psi_{cur}) & \text{otherwise} \end{cases} \tag{3.18}$$

else if $D_\Psi < 0$

$$F_\Psi = \begin{cases} ((\Psi des - 360) - \Psi_{cur}) & \text{if } \Psi_{des} - \Psi_{cur} \leq 180 \\ (\Psi_{des} - \Psi_{cur}) & \text{otherwise} \end{cases} \tag{3.19}$$

The pitch force still needs two further steps; First, a normalization to get values between (-180; 180), and second, a scaling with the factor k_Ψ:

$$F_\psi := \begin{cases} F_\Psi - 360 & \text{if } F_\Psi > 180 \\ F_\Psi + 360 & \text{if } F_\Psi < -180 \end{cases} \tag{3.20}$$

$$F_\psi := F_\Psi * k_\Psi \tag{3.21}$$

The parameter k_{pos}, k_Ψ and k_θ are linear factors that are applied to the forces. These can be used to fine-tune the reaction speed of the camera with respect to the player position and orientation. The higher these factors are, the bigger are the computed forces. These parameters can also be connected to one or more global parameters of the camera module. This allows the adjustment of the camera's stiffness by the game loop.

3.5.2. Numerical integration

Now that the forces F_{pos}, F_Ψ and F_θ are known, the last step of the camera simulation loop can be processed; The numerical integration. For this purpose, a stable and fast integration method is needed. It has been shown, that leap frog integration is a good method, which provides excellent stability. The new camera configuration can now be integrated from the old camera configuration using the previously calculated fores by the following equations:

$$F'_{pos} = F_{pos} * (1 - k_{fric}) \tag{3.22}$$
$$v_{pos,i+\frac{1}{2}} = v_{pos,i-\frac{1}{2}} + F'_{pos} * dt \tag{3.23}$$
$$P_{cur,i+1} = P_{cur,i} + v_{pos,i+\frac{1}{2}} * dt \tag{3.24}$$

$$F'_{\Psi} = F_{\Psi} * (1 - k_{fric}) \tag{3.25}$$
$$v_{\Psi,i+\frac{1}{2}} = v_{\Psi,i-\frac{1}{2}} + F'_{\Psi} * dt \tag{3.26}$$
$$\Psi_{cur,i+1} = \Psi_{cur,i} + v_{\Psi,i+\frac{1}{2}} * dt \tag{3.27}$$

$$F'_{\theta} = F_{\theta} * (1 - k_{fric}) \tag{3.28}$$
$$v_{\theta,i+\frac{1}{2}} = v_{\theta,i-\frac{1}{2}} + F'_{\theta} * dt \tag{3.29}$$
$$\theta_{cur,i+1} = \theta_{cur,i} + v_{\theta,i+\frac{1}{2}} * dt \tag{3.30}$$

The parameter k_{fric} used in the equations 3.22, 3.25 and 3.28 is the linear friction coefficient of the velocity. Note that k_{fric} should be linearly dependent on the factors k_{pos} in 3.16, k_{Ψ} in 3.21 and k_{θ} in 3.17 from the force computation step in order to keep the simulation stable. The bigger the linear factors are, the higher the velocities get. This causes the camera to oscillate around its desired position if the friction is not adjusted accordingly. The parameter k_{fric} can also be different for positional and angular velocities, but observations show that it works well with just one coefficient for all three forces.

4

Camera Module and Surrounding Geometry

Forces are now applied to the camera object during the simulation loop. This drives the camera automatically towards the desired position. The behavior is determined by the controller that defines the position and orientation depending on the current active control routine. Through the physical representation of the camera, the task of filtering out high frequency movement is guaranteed. The design of the simulation loop also assures that the paths, that need to be traveled by the camera when switching between different camera styles, are smooth. Through the use of the leap frog method for numerical integration, the stability of the system as well as the low performance costs are met as well.

However, one of the most important tasks is not considered yet in the current model; Avoiding the surrounding geometry. The camera is not aware of any geometry at the moment, and will always stolidly move towards the desired configuration. This behavior can cause it to penetrate objects or get occluded.

4.1. Problems using the Camera Controller

The fact that the scene geometry is not taken into account introduces several situations that are highly undesired. Figure 4.1 shows different settings where the camera module will penetrate the geometry, or the line of sight between camera and player gets occluded.

In 4.1 a), the player, represented by the circle, jumps down a brink. The camera follows in third person view as the angle between the horizontal plane and the look direction is by definition constant. This causes the camera to be occluded, and it can get even worse, if the brink is high

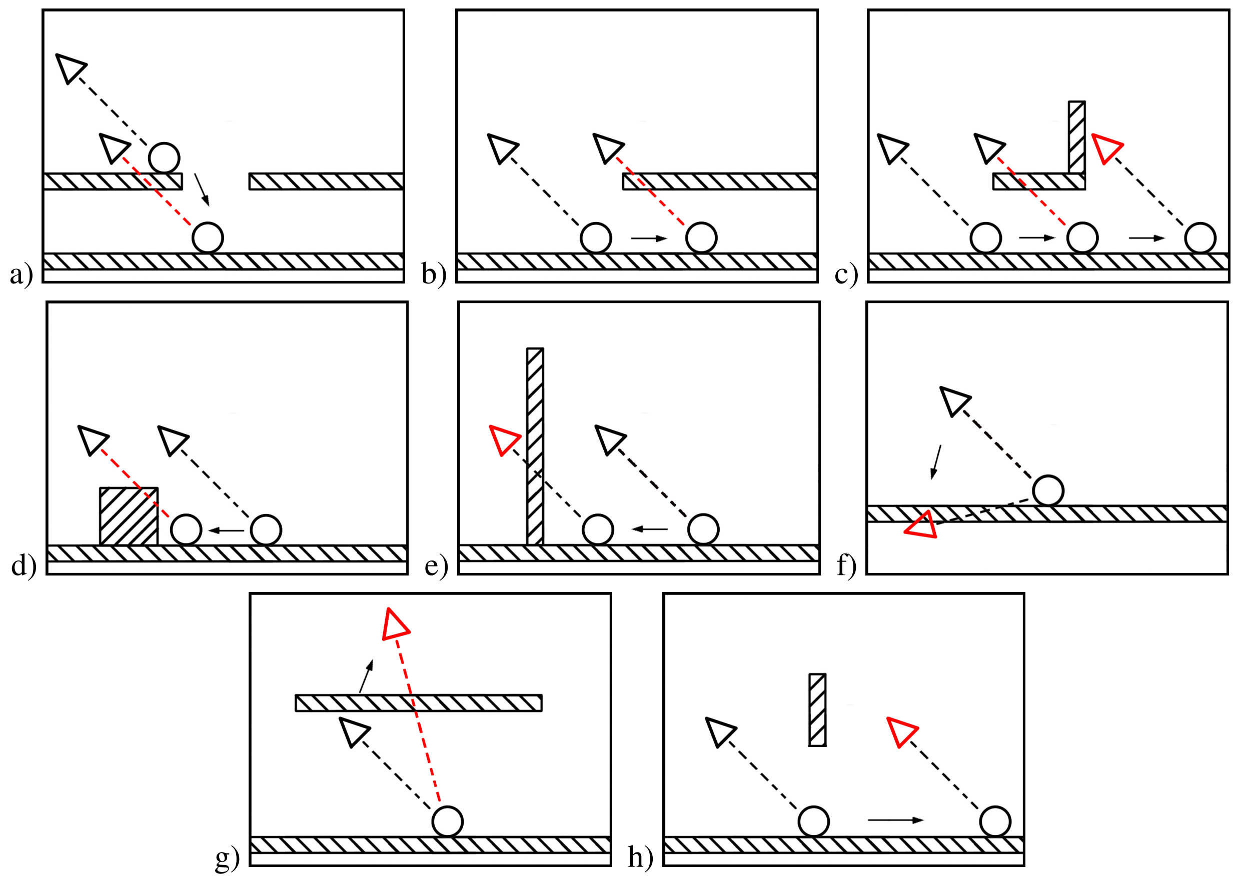


Figure 4.1.: Situations where a simple camera model fails.

enough that the camera will penetrate the geometry.

The image 4.1 b) demonstrates a similar problem. The camera occlusion is caused by the player moving underneath an object. Again, the line of sight is blocked because the camera does not know that there is geometry in the way. The figure 4.1 c) shows the case, which can arise in extension to b). While in the first configuration, the line of sight occlusion may be resolved when the player exits the occlusion area, the camera will still get forced through the corner at the end of the object.

Image 4.1 d) shows the situation, where the player is moving backwards to an object. Similar to the case in 4.1 a), the line of sight between player and camera gets occluded. In 4.1 e), again, the player is moving backwards. The object, however, is big enough to not get between the camera and the player. In this setting, the camera is forced to penetrate the geometry. Also changing the camera angle introduces problems. This is shown in 4.1 f), where the view angle may be reduced such that the camera moves below the floor.

Another problem, which is a direct consequence of the different camera styles, is shown in 4.1 g). If the camera mode is changed from one style to another, the new desired position may lie on the other side of a solid object. For example switching from third person view to isometric view, where the distance from player to camera typically will increase, the camera will be forced through the object. The same is true when changing from first person view to third person, or vice versa. The camera has no means to detect these situations and will penetrate the geometry nonetheless. In third person view it will also remain in an undesired occluded state.

A further problem is introduced in situations where the camera is at the same height as an object. In these cases, schematically shown in 4.1 d), the line of sight may not be intersected, but the camera penetrates the geometry nonetheless.

4.2. Extended Camera Controller

The next step to improve the camera behavior is to incorporate methods that permit the camera to react to the environment. The straightforward approach would be to have a reference to the scene geometry or to the surrounding parts. This, however, leads to problems keeping the interface clean. On one hand, if the camera would have a reference to the data structure, in which the geometry is stored, the representation must be known to some degree. This limits the programmer in his choice of the data structure. On the other hand, if the camera would have its own representation of the geometry, there would be an overhead at the initialization as the game has to give the geometry to the camera. Also the whole data would possibly needed to be copied, which is an undesired memory overhead.

In 3D games today it can be assumed that the game itself has some spatial representation of the geometry, that allows it to test for collisions with known primitives, like bounding spheres or bounding boxes, itself. This can be exploited by shifting the camera loop in such a way that the collision tests are coherent with the beginning and the end of the camera's update routine. This shift is shown by the two diagrams in Figure 4.2. The diagram on the left side shows the current loop of the camera, with the extension of collision detection and response. After the computation of the desired configuration for the camera, it should be tested if the position is

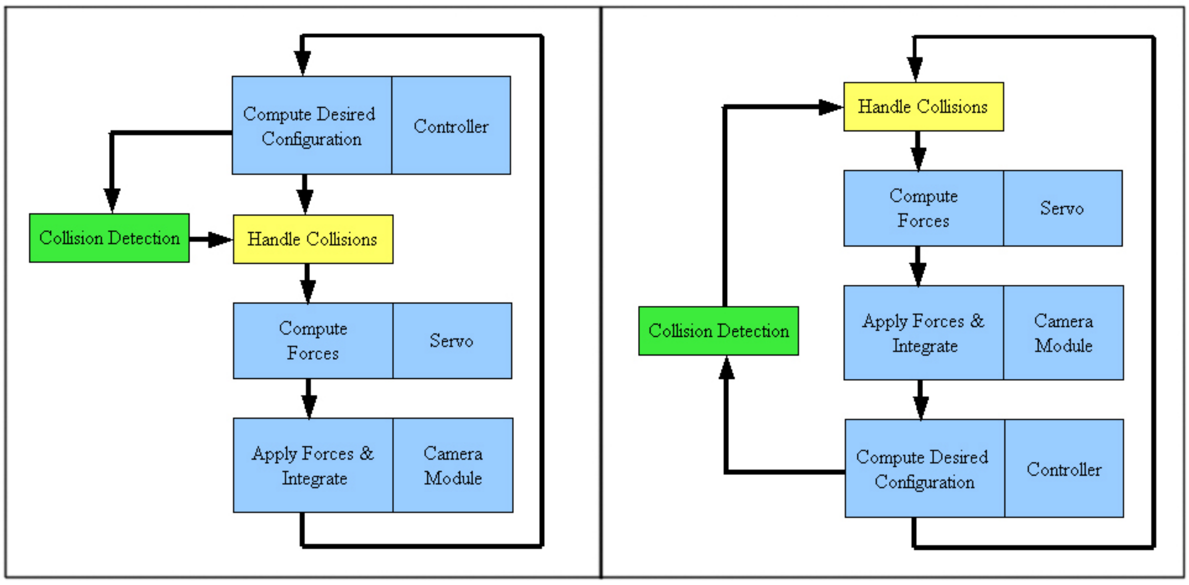


Figure 4.2.: These diagrams show the extended camera simulation loop. On the left side the collision handling is added to the original loop. The interface needs input about the collision detection from the game main loop. The scheme on the right shows the same loop, but shifted up a step. This shift allows the game loop to calculate collisions and intersections between two camera update calls.

valid. If the position is not valid, the camera can fix it. The diagram on the right shows the same loop, but shifted up one step. This leads to the situation that the tests, which need to be done by the game main loop, are at the end of the camera update. This gives the possibility to add structures (like bounding volumes or rays) to the interface, which can be tested against collisions or intersections by the game main loop. Through another function of the interface, the results of such tests can be reported back to the camera module and, during the next update loop, it can adapt the desired configuration accordingly.

4.3. Improvements using a Raycast

Through the shift of the update loop, it is now possible for the camera to define structures that can be tested by the game main loop. One possibility to improve some of the above discussed situations is a ray cast.

Depending on the current active control routine, the line of sight may be important. In third person view it is highly undesired if the camera gets occluded. To improve situations where this may happen, a ray, starting at the player position P_{player} and ending at the desired camera position P_{des}, is held up-to-date by the third person control routine. The game main loop can request this ray after a camera update, and test for the first intersection of this ray against the environment.

The ray is represented as a set of parameter $Ray = \{P_{ray}, D_{ray}, t_{max}\}$, where P_{ray} is the start point of the ray, D_{Ray} is the direction vector of the ray, and t_{max} is the maximal valid parameter

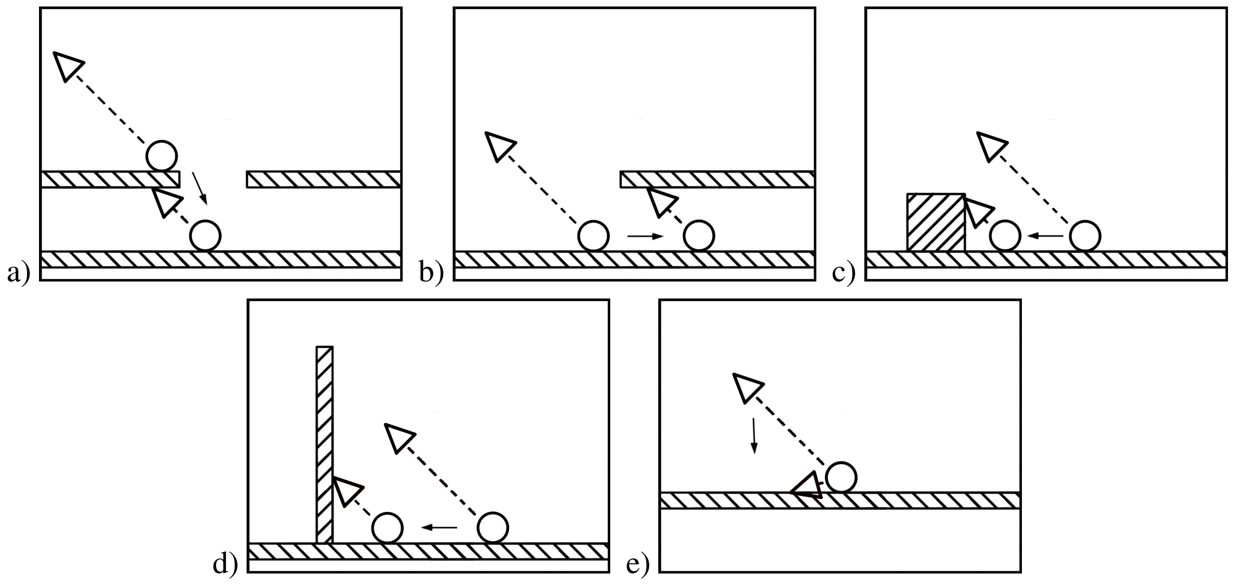


Figure 4.3.: Situations in which a raycast resolves an occluding state of the camera.

on the ray. The third person camera control routine holds the ray up-to-date by computing the parameter at the end of the camera update routine, when the new desired camera configuration is known:

$$P_{ray} = P_{player} \tag{4.1}$$
$$D_{ray} = \frac{P_{des} - P_{player}}{||P_{des} - P_{player}||} \tag{4.2}$$
$$t_{max} = ||P_{des} - P_{player}|| \tag{4.3}$$

The game loop can now check for a ray intersection with the environment, and, if found, report the intersection parameter t_{isect} back to the camera module. During the first step of the next camera update call, the control routine reacts to a reported intersection by changing the desired camera position P_{des}, which was computed at the end of the previous loop, as follows:

$$P_{des} = P_{ray} + (D_{ray} * t_{isect}) \tag{4.4}$$

The desired camera position is set to the reported intersection on the ray. This is the first valid point in the line from player to camera where the camera is not in an occluded state. Through the computation of forces by the servo unit, the camera is automatically driven towards this position without high frequency movement. Situations that are resolved using the ray cast are shown in Figure 4.3.

In 4.3 a), when the player jumps down a ledge, the occlusion gets resolved by the camera moving towards the point where the player-camera ray hits the geometry. This, however, leads to the camera moving through the geometry. The same is true for 4.3 b) and 4.3 c), where the player moves underneath an object, or backwards against an object respectively. The situations shown in 4.3 d), where the player moves backwards against a wall, and 4.3 e), where the camera angle is changed, are fully resolved by this method.

The ray cast that resolves several situations which mainly arise for the third person camera mode, is not restricted to one mode. Because the ray is updated and the desired position is fixed by the control routine, the approach can also be used in other camera modes. For isometric view, it may be reasonable to maintain a minimal offset distance d_{offset} in viewing direction from the geometry. This can be achieved by a ray, that starts at the desired camera position P_{des}, and reaches out the minimal offset in viewing direction:

$$P_{ray} = P_{des} \tag{4.5}$$

$$D_{ray} = L_{ic} \tag{4.6}$$

$$t_{max} = d_{offset} \tag{4.7}$$

A reported intersection of the ray with the geometry gives the parameter t_{isect}, which is the point from which the desired camera position P_{des} should maintain the distance d_{offset} in direction $-L_{ic}$. The isometric control routine can fix the desired camera configuration by recomputing P_{des} as follows:

$$P_{des} := P_{des} - (L_{ic} * (t_{max} - t_{isect})) \tag{4.8}$$

4.4. Collision Response using the Extended Camera Controller

The extended camera controller has now, through the shifted loop, the possibility to add primitives to the interface, that can be used to compute information in between two camera update calls. This is not limited to a ray cast, but also can be extended with a bounding primitive to report collisions to the camera. For this purpose a bounding sphere with a position P_{bound} and a constant radius r_{bound} is an appropriate representation of the camera. The only parameter, that needs to be updated by the camera loop, is the position of the sphere, which is the same as the current camera position P_{cur}:

$$r_{bound} = const \tag{4.9}$$

$$P_{bound} = P_{cur} \tag{4.10}$$

Collision response can be done in several ways. The easiest solution, which also leads to good results, is the use of penalty forces proportional to the penetration depth. Therefore the game main loop needs to report collisions of the camera bounding sphere by delivering the

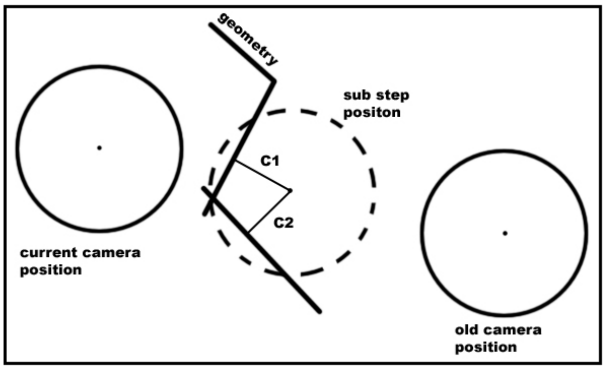


Figure 4.4.: The parameter, that are needed to resolve a collision of the camera bounding sphere with primitives are shown. The average collision point C_{avg} is the normalized sum of all vectors connecting the sphere origin with the collision points of each primitive. The camera sub step position is the position of the camera bounding sphere, at which the first intersection was detected.

average collision point C_{avg}. This point can be computed as average of all collision points $C_1, C_2, \ldots, C_n$ of the bounding sphere with each primitive (see Figure 4.4).

$$C_{avg} = \frac{1}{n} * \sum_{k=1}^{n} C_k \tag{4.11}$$

To resolve a collision reported by the interface, a penalty force $F_{penalty}$ is computed relative to the penetration depth and in inverse direction of the penetration direction. To calculate $F_{penalty}$, the position of the camera is needed at which the collision was found. To allow the game loop to perform sub stepping, a second parameter indicating the sub stepping collision position $P_{avg,sub}$ is introduced. This simply is the position at which the center of the bounding sphere was assumed by the game loop, when a collision was detected (See also Figure 4.4). The penetration direction D_{pen} as well as the penalty force $F_{penalty}$ can be computed using the parameter C_{avg} and $P_{avg,sub}$:

$$D_{pen} = \frac{C_{avg} - P_{avg,sub}}{||C_{avg} - P_{avg,sub}||} \tag{4.12}$$

$$F_{penalty} = -D_{pen} * \frac{(r_{bound} - |C_{avg} - P_{avg,sub}||) * k_{penalty}}{r_{bound}} \tag{4.13}$$

The penalty force $F_{penalty}$ can now be added to the positional force F_{pos} in equation 3.22 to resolve the collision during the integration step. The parameter $k_{penalty}$ is, again, a scaling coefficient for fine tuning the penalty force w.r.t. the other forces.

Including collision response into the camera simulation loop also affects different situations, which were undesired before. Figure 4.5 shows different situations that change when collision response is active. In 4.5 a), where the player moves backwards against a wall, the camera will

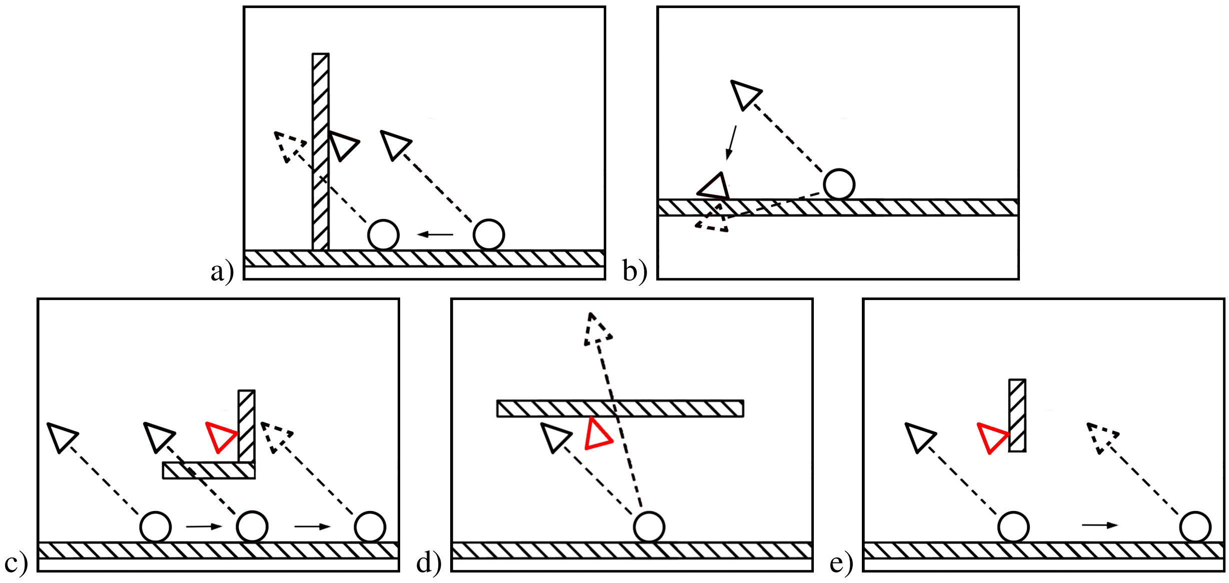


Figure 4.5.: Situations that change using collision response.

be forced to the first non-occluding position in penetration direction. The same effect keeps the camera from penetrating the ground in 4.5 b), where the player changes the angle of the camera-player line of sight. The setting in 4.5 c), however, gets negatively affected by the collision detection. As the player moves underneath the geometry, the camera will follow, and eventually get stuck at the corner. This situation will not occur if the raycast is used as well.

The situation in 4.5 d) is slightly different. Here the player changes the camera mode. The camera should move from the third person view to the isometric view. This however will get the camera stuck, if there is an object between the two positions. Through the forward dynamics of the camera movement, it will move in a straight line, and therefore not avoid the object.

Figure 4.5 e) shows another problem. If an object is at the same hight as the camera, the view will not get occluded. The camera bounding sphere, however, will collide with the object. This can be a problem as the penalty force is proportional to the penetration direction. It can happen that the camera is forced back in the same direction as the player moves. This causes the camera to get stuck at the object, even if a slight movement down would resolve the situation.

4.5. Summary and Remaining Issues

With the extended and shifted camera controller, it is now possible to integrate response structures into the camera controller without the a reference to the scene geometry. Through the request of a primitive, like a ray or a bounding volume, the game main loop can test for collisions and intersections with its own data structures, and report them back to the camera.

In the previous sections two methods have been shown to get the camera to avoid penetrations and occlusions. The ray cast resolves a lot of situations where the camera was forced through objects as well as situations where the player-camera line got occluded. This mechanic is very robust as the camera will never stay in an occluded state. It however does not guarantee pen-

etration free movement. Especially in situations like 4.1 a), where a ledge gets between the camera and the player, the camera is forced to move through the object to the non-occluding state.
Collision detection and response seems to be a good solution to avoid penetrations. Through a penalty force, which is applied in the inverse direction of the penetration direction, the camera will not move through an object. This mechanic however also adds different problems. The camera can get stuck (see eg 4.5 c)), which is highly undesirable. Also, due to the mechanics of penalty forces, the camera can start to oscillate, if it is forced into a corner.

If collision response and ray cast are both active another problem arises. Take the situation in 4.1 a) as an example; The player jumps down a ledge and, for a short period of time, the camera gets occluded. The ray cast mechanics will recompute the desired camera position to the point just underneath the geometry, and the camera starts to move towards it. At the moment where the camera starts to penetrate the ledge, it bumps off the object because of the collision response. This motion is also undesired as it looks very uncontrolled.

To summarize the use of the two solutions to avoid penetrations, collisions, and occlusions; It is currently the best tradeoff, to only use ray casts. This mechanic guarantees that all occlusions are resolved, although the camera sometimes will move through an object. It depends on the game level whether the collision response can improve the overall camera behavior or not.

5 Conclusion and Outlook

In this thesis a method to create a reusable game camera was shown. The camera itself is physically simulated with 5 degrees of freedom; Three degrees for the position, one for pitch and one for yaw. A camera controller was introduced that is able to perform different camera styles through the use of different control routines. This mechanism also permits for further extensions with new styles without changing the overall architecture. With an extension to the controller it is possible to have the camera react to its surroundings without explicit knowledge about the geometry. Two possible exploitations of this extended controller were shown to improve the camera behavior. First, a ray cast for intersection testing of the line of sight was shown with which the camera can resolve all occlusions. In a similar way collision detection and response was integrated into the system to prevent the camera from penetrating geometry.

Several goals of the problem description in chapter 3 are met. High-frequency movement is removed from the camera motion through the physical simulation. The filtering is also done when switching between different control routines as the camera always is driven by forward dynamics towards its desired configuration. The desired degree of flexibility of the camera is reached because the control routines allow the encoding of different camera behavior patterns. It can also easily be extended as only new control routines defining the new behavior need to be added. The overall computational performance of the camera module is quite low as well. Both forward dynamics for force computations and the leap frog method for numerical integration are computationally cheap methods that perform very well for their tasks. Through the extension of the camera controller, occlusions and penetrations of the camera can also be handled. Not only is the main purpose of a small and easy interface met, but, as a consequence, no information about the surrounding scene needs to be stored.

Observations show that the camera performs best when only the ray cast is active. With it, the camera is guaranteed to never stay in an occluded state. It is, however, possible that the camera moves through corners of objects to get to the non-occluding position. Collision response turned out to be a mixed blessing. On one hand, it efficiently avoids penetration of objects. On the other hand, it introduces new problems. If collision response is active it becomes possible for the camera to get stuck in certain situations. This effect is highly undesired and therefore the best tradeoff at the moment is to drop collision response and allow the camera to penetrate geometry in order to avoid an occluded state.

For future work, the camera has several issues that may be resolved using advanced visibility and path planning techniques as well as machine learning algorithms. In its current state, the camera only has knowledge about the current frame. There is no kind of look-ahead mechanism integrated, which would be necessary to allow the camera to react in advance of problematic situations. A possible solution may be to have a number of frames where the camera has knowledge about its position and possibly parts of the surroundings. This could be used to predict future configurations of the camera and test them for occlusions and collisions.
As the ray cast visibility test is the simplest approach, several more sophisticated methods may lead to better results. By using a single ray cast the camera is either occluded or not. No information about positions that resolve occlusions can be computed besides the first non-occluding point on the ray itself. Techniques similar to shadow maps could give some more information about an occlusion as they also take the neighborhood into consideration.
The camera movement is also very straightforward at the moment. Through the force computations by forward dynamics, the camera stolidly moves towards a desired position. This method is appropriate because if the camera only has knowledge about the current time frame. However, if the system is extended to a series of frames, this approach may become to simple. Several path planning algorithms, that compute forces depending on more factors than current and desired position could improve the behavior.
Another way to resolve problematic situations may be to incorporate machine learning techniques, where the camera has a mechanism to learn its behavior from off line computed solutions of different situations. This approach would need, on one hand, a deeper geometrical analysis of the different problems. On the other hand, a well designed machine learning approach, like a support vector machine or a neural network, is needed that can learn from the results of the analysis. This also implies the need for some kind of sensor system, which can gather information from the surroundings of the camera.

A

Reusable Game Camera for XNA

As part of this thesis I implemented the reusable game camera in C#. I choose to implement it for the Microsoft XNA game development framework. This appendix describes the use of the camera module in an XNA game project. The camera module can be used for both a Windows game and an XBox 360 game.

A.1. Adding the camera to a new project

This section shows you step-by-step what needs to be done to include the camera into your XNA game project. After this section the camera will automatically follow an assigned player.

A.1.1. Include source files

First you need to download the source files. The current version can be found on the thesis CD. You need to unpack the .zip file and copy the .cs files into your XNA project folder.

To include the classes, go to the solution explorer of your Visual Studio project and right-click onto the folder where you want the files. Go to Add > ExistingItem and select the source files you just unpacked (See Figure A.1).

A.1.2. Create camera object and connect it to a player

Now that you have the camera class files in your project, you can create the camera. First, you need to add the camera name space to your game class:

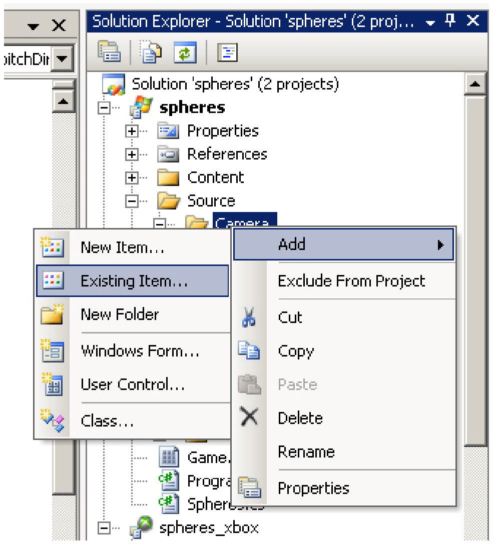


Figure A.1.: Include camera source in an XNA project using the Solution Explorer.

```
using GameCamera;
```

Now you can declare the camera object as well as create and initialize it:

```
private GameCamera.GameCamera camera;

public YourConstructor() {
    ...

    camera = new GameCamera.GameCamera();

    ...
}


public void InitializeYourClass() {
    ...

    // initialize camera with aspect ratio and opening angle
    camera.initialize(800.0f / 600.0f, 50.0f);

    ...
}
```

To get the camera working, you need to connect it to a player by calling the camera update routine with the parameter *actual game time*, *player position*, *player orientation* and *game pad state*.

```
public void YourUpdate(GameTime gameTime) {
    ...

    // get game pad state of pad 1
    GamePadState gamePadState = GamePad.GetState(0);

    // update camera
    camera.update(gameTime, playerPosition,
                  playerOrientation, gamePadState);

    ...
}
```

The camera now follows the player position and orients itself to the player orientation. The camera's default mode is third person view. For this mode there are some parameter that define the behavior of the camera. These can be changed at every point in your program code:

```
camera.TPCAngle = 40.0f;     // the angle between the player-camera
                             // ray and the xz-plane

camera.TPCDistance = 25.0f; // the distance of the camera along
                             // the player camera ray
```

The only thing, that is still missing, is to get the view and projection matrix, and draw your scene from the cameras point of view:

```
public void YourDrawFunc() {

    // get the view matrix
    Matrix viewMatrix = camera.viewMatrix;

    // get the projection matrix
    Matrix projectionMatrix = camera.projectionMatrix;


    ... // draw your geometry using the matrices

}
```

A.2. Change camera modes and mode settings

In the previous section, you learned to integrate the camera into your project. The camera has different camera modes, which each define a specific behavior. The modes themselves have each different settings that can be adjusted to your personal needs.

A.2.1. Change camera mode

The camera is initialized by default in third person view. There are two more common game camera modes available; Namely first person view and isometric view. The following code snippet shows how to change the camera mode.

```
// set camera to first person view
camera.mode = GameCamera.CamMode.firstPerson;

// set camera to third person view
camera.mode = GameCamera.CamMode.thirdPerson;

// set camera to isometric view
camera.mode = GameCamera.CamMode.isometric;
```

You can change the camera mode at any point of your program code. The camera simulation, which is completely physical, guarantees seamless fades between the different modes. If you want to iterate through the different styles, the static class CamMode provides a field with the number of modes:

```
for (int i=0; i < GameCamera.CamMode.numModes; i++ {
    ...
}
```

A.2.2. First person mode setting

The first person camera view sets the camera to the player position + offset. The orientation of the camera is entirely determined by the player's orientation, which is given as parameter to the camera update function.

The offset mentioned above is a $Vector3$ that defines the position of the camera relative to the player. The default value of this vector is (0.0f, 1.5f, 0.0f). It can be changed by the following piece of code:

```
camera.FPCcamOffset = new Vector3(1.0f, 5.0f, 1.0f);
```

A.2.3. Third person mode settings

As mentioned above, the third person camera view is the default mode of the camera module. In this view the camera follows the player position, trying to keep two parameter constant; The distance to the player and the angle between the xz-plane and the player-camera line. The camera orients itself to the player orientation a way that it looks from behind on the player.

The following properties of the camera are used to change the distance and the angle of this camera mode:

```
camera.TPCAngle = 40.0f;    // the angle between the player-camera
                            // ray and the xz-plane

camera.TPCDistance = 25.0f; // the distance of the camera along
                            // the player camera ray
```

A.2.4. Isometric mode settings

Last, but not least, there is the isometric camera. The camera in this mode also tries to keep a defined distance from the player. In contrast to the third person view, however, the orientation of the camera does not change when the player orientation changes. Therefore the second parameter is a $Vector3$ that defines the look direction of the camera onto the player.

These parameter can be accessed through the following camera properties:

```
camera.ICDistance = 60.0f; // set isometric camera distance

// set view direction
camera.ICLookDir = new Vector3(-1.0f, -1.0f, -1.0f);
```

As you see in the example above, the look direction can be set with a non-normalized vector. This is done automatically by the camera class.

A.2.5. Using mode settings to influence the camera

At this point, you should be able to change the camera modes to your own needs, and also change the settings of each mode as you feel like it. With a bit creativity you can easily influence the camera behaviour to fit your needs; Zooming for example. In third person view is done just by changing the parameter TPCDistance.

The following code snippet shows how to implement different controls for the camera modes. The code just needs to be called during your update routine.

```
// process input
switch (mMode) {
    case (CamMode.firstPerson):
        break;
    case (CamMode.thirdPerson):
        // adjust angle using the thumb stick
        mTPCAngle -= gamePadState.ThumbSticks.Right.Y * 1.3f;
        if (mTPCAngle > 85.0f) mTPCAngle = 85.0f;
        else if (mTPCAngle < -85.0f) mTPCAngle = -85.0f;

        // adjust distance (zoom)
        if (gamePadState.DPad.Down == ButtonState.Pressed)
        {
            mTPCDistance += (gameTime.ElapsedGameTime.Milliseconds
                                                    / 1000.0f) * 25;
        }
        else if (gamePadState.DPad.Up == ButtonState.Pressed)
        {
            mTPCDistance -= (gameTime.ElapsedGameTime.Milliseconds
                                                    / 1000.0f) * 25;
        }

        break;
    case (CamMode.isometric):
        // adjust distance (zoom)
        if (gamePadState.DPad.Down == ButtonState.Pressed)
        {
            mICDistance += (gameTime.ElapsedGameTime.Milliseconds
                                                    / 1000.0f) * 25;
        }
        else if (gamePadState.DPad.Up == ButtonState.Pressed)
        {
            mICDistance -= (gameTime.ElapsedGameTime.Milliseconds
                                                    / 1000.0f) * 25;
        }
        break;
}
```

A.3. Avoid line of sight blocker and collisions

If you followed the previous sections, you should have a working camera in your game that acts as you want it to. The scene geometry, however, is not taken into account yet. This can lead to some ugly situations. For example, if an object gets between the camera and the player, the view is blocked. Furthermore, as collisions are also neglected, the camera may penetrate the geometry as well (See Figures 4.3 and 4.5).

This section explains two functions of the camera interface, that can be used to get the camera aware of such situations and react to them.

A.3.1. Avoid line of sight blocker

Probably the most undesired effect of the camera, trying to stay at a defined postition relative to the player, is when it gets occluded. This may happen if the player moves backwards towards a wall or moves behind an object. For these situations, the camera provides a ray to be collided with the scene geometry, and a function to set the ray intersection parameter. If this function is called before the next camera update(), the camera automatically will resolve the occlusion.

The first step is to request the collision ray from the camera. This ray is always kept up to date by the camera's update() routine and represents the ray the camera wants to be checked against collisions. The ray may represent different lines depending on the camera mode. This however does not have to concern you, as the camera will know the meaning of the intersection parameter you report.

```
// get the ray that needs to be tested
GameCamera.IRay ray = camera.intersectionRay;
```

The camera module now assumes that the ray is collided with the scene, and that the nearest collision to the ray's start point is reported. Using the following function, you can report the intersection parameter t on the ray back to the camera. Note that a parameter smaller than zero is interpreted as no collision.

```
// test ray for collisions ->
// t = first collision on ray

...

if ((t > 0) && (t < ray.Max_t)) {
    // set parameter t if collision found
    camera.setRayCastParameter(t);
}

...

// update camera
camera.update(...)
```

By the call of the function $setRayCastParameter$, the camera will be aware of an intersection, and handle it in its next update() call automatically.

The reaction to this call depends on the camera mode. In first person view, the camera will do nothing (as the requested ray has no meaning for this mode as well). In third person mode, however, the ray represents the line between player and camera. A detected ray collision will force the camera to the collision point. This means, the camera moves to a new state, where its view onto the player is not occluded anymore. In isometric view the camera uses the ray cast to maintain a constant distance above the ground in look direction. If the player moves under an object, the camera will zoom out to give a better overview over the scene.

A beautiful and very fast algorithm to compute ray-triangle collisions can be found in [PH04]. The implementation of this routine in C# is shown below:

```
private float intersectRayTriangle(GameCamera.IRay r,
                                         Vector3 p1,
                                         Vector3 p2,
                                         Vector3 p3) {

    // computing barycentric coordinates of the triangle
    Vector3 e1 = p2 - p1;
    Vector3 e2 = p3 - p1;
    Vector3 s1 = Vector3.Cross(r.Direction, e2);
    float divisor = Vector3.Dot(s1, e1);
    if (divisor == 0)
        return -1;

    float invDivisor = 1.0f/ divisor;

    // compute first baricentric coordinate b1
    Vector3 d = r.Position - p1;
    float b1 = Vector3.Dot(d, s1) * invDivisor;
    if ((b1 < 0) || (b1 > 1))
        return -1;

    // compute second baricentric coordinate b2
    Vector3 s2 = Vector3.Cross(d, e1);
    float b2 = Vector3.Dot(r.Direction, s2) * invDivisor;
    if ((b2 < 0) || (b1 + b2 > 1))
        return -1;

    // compute parameter t of the intersection
    float t = Vector3.Dot(e2, s2) * invDivisor;
    if ((t < 0) || (t > r.Max_t))
        return -1;

    // return intersection parameter t
    return t;
}
```

A.3.2. Avoid collisions

With the ray cast, a lot of problems with the geometry are solved. However depending on your scene geometry, there might still be some undesired behavior of the camera. The ray cast only resolves line of sight problems and does not handle collisions with the environment. This can lead to situations where the camera penetrates the geometry. Note, however, that in some cases collision detection and response also can have a negative effect; For example, when a corner gets in between the player and the camera, it may get stuck.

To get the camera react on collisions, a similar functionality exists in the interface as with the ray request and collision report. First you need the bounding sphere of the camera, which can

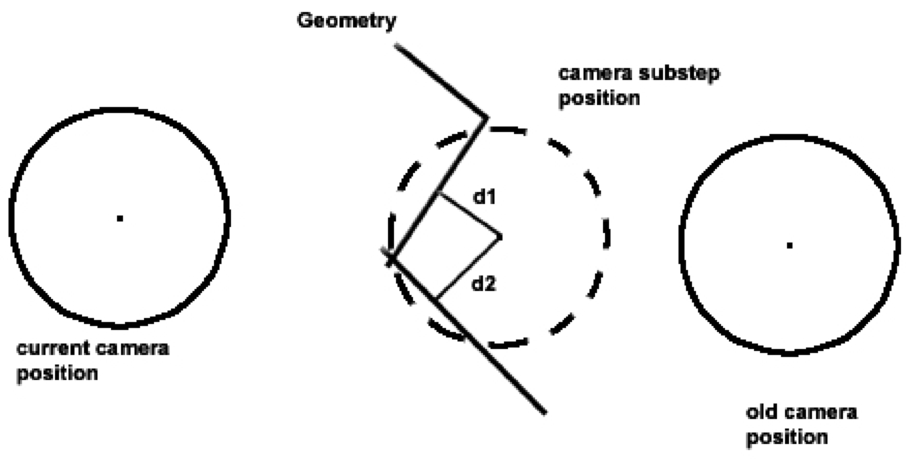


Figure A.2.: This figure shows the parameter needed by the camera module to resolve collisions.

be requested as follows:

```
// get the bounding sphere
BoundingSphere sphere = camera.boundingSphere;
```

In the same way as with the ray cast, the camera depends on reports of the collision by the game main loop. In this case, two parameters are needed; the average collision point and the camera sub stepping position. To report a collision, you can do the following:

```
// test bounding sphere for collisions

...

if (collision) {
    // set parameter if collision found
    camera.setCollisionParameter(avgCollisionPoint,
                                 camSubstepPosition);
}

...

// update camera
camera.update(...)
```

Again, the next call of the camera's update() method will handle the collision.

The parameter *average collision point* and *camera substep position* are schematically shown in Figure A.2.

The camera sub step position is the first position of the camera between the old and current position of the camera, where you detect one or more colliding faces. In most cases this position is be equivalent to the current camera position. However, if the velocity of the camera is too large, meaning the camera entirely moves through the geometry in one update step, this gives the possibility for sub stepping. The average collision point is easy to compute once all colliding faces are found. For each of these faces you have to find the nearest point to the camera sub step position. The vectors from the camera sub step position to all of these points just need to be averaged.

A.3.3. Issues

The performance of the camera resolving occlusions and collisions depends a lot on the geometry you are using in your scene. In some situations problems can arise, when using both line of sight test and collision test at the same time.

If the camera gets behind an object, it is forced back to the first position on the ray, where it is not blocked. Due to the delay introduced by the physical simulation the camera will attempt to move through the object. At this point the collision detection will stop it from getting to the desired position. The camera will gets stuck behind the object until forces get big enough to push the camera through the geometry in one update step. This can be avoided by only using the ray cast.

Another issue can arise if the camera is forced into a sharp corner (smaller than 90 degree). This situation can, in some cases, lead to an oscillation of the camera. Again, this problem can be resolved if collisions are not reported. If collision response is disabled, however, the camera may sometimes penetrate objects. In the current version of the generic game camera, this is the best tradeoff. You may test different combinations of ray casts and collision tests in your game scene to see what works best.

Bibliography

[BGL98] William H. Bares, Joel P. Gregoire, and James C. Lester. Realtime constraint-based cinematography for complex interactive 3d worlds. In *AAAI/IAAI*, pages 1101–1106, 1998.

[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.

[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.

[HS] Ralf Helbing and Thomas Strothotte. Quick camera path planning for interactive 3d environments.

[KcL94] Lydia Kavraki and Jean claude Latombe. Randomized preprocessing of configuration space for fast path planning, 1994.

[NSL99] C. Nissoux, T. Simeon, and J. Laumond. Visibility based probabilistic roadmaps, 1999.

[OS95] Overmars and Svestka. A probabilistic learning approach to motion planning. In Goldberg, Halperin, Latombe, and Wilson, editors, *Algorithmic Foundations of Robotics, The 1994 Workshop on the Algorithmic Foundations of Robotics, A. K. Peters*, 1995.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufman Publ.Inc., 2004.

[Pic] Jonathan H Pickering. Intelligent camera planning for computer graphics.

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

[SD03] S. Simhon and G. Dudek. Path planning using learned constraints and preferences, 2003.

[SS97] K. Sugihara and J. Smith. Genetic algorithms for adaptive motion planning of an autonomous mobile robot, 1997.

[Woo98] Wayne L. Wooten. Simulation of leaping, tumbling, landing and balancing humans. 1998.

[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.