The Full Spectrum Warrior Camera System

John Giors

1 INTRODUCTION

1.1 Motivation

Camera systems are among the most important systems in any game. After all, the camera is the window through which the player interacts with the simulated world. Unfortunately, the camera system is often neglected during the development process, as it is easy to take for granted. When making Full Spectrum Warrior (FSW), special attention was given to camera system issues, resulting in unique solutions to several problems.

1.2 Road map

The following discusses specifics of the camera system as developed for FSW. A brief functional overview of the camera system will be followed by a description of the high-level architecture. Following that, the details of the motion system will be examined. The next section covers FSW's unique "autolook" feature. Then, the bane of every camera system—collision avoidance—will raise its ugly head. Finally, debugging, tuning, and miscellaneous issues will be covered. To wrap it all up, the limitations of the FSW camera system will be discussed, along with some general recommendations that apply to most projects.

2 FSW CAMERA SYSTEM FUNCTIONAL OVERVIEW

The primary camera system is under the control of the user, manipulated with the gamepad's right thumbstick.

In FSW, the user controls two teams of four soldiers each. One of the soldiers on each team is the team leader. Pressing the Y button switches teams. The camera performs a flyby to the other team's leader, assuming the same facing as the soldier. During the flyby, the camera is not under user control. When the camera arrives at the team leader, camera control returns to the user.

There is a brief lockout period at the beginning of a flyby during which inputs from the gamepad are not accepted. However, after the initial lockout period, the user may make inputs. In this situation, when an input is detected, the camera cuts immediately to the destination and the user regains control of the system.

The user may also select different soldiers on a team using the d-pad. The camera automatically attaches to the character, performing a mini-flyby. This allows the user to get different vantages on the current situation.

FSW also has in-game cinematics, which take over camera control. In addition to use of the normal in-game view system, the cinematic camera may move between preset positions.

3 HIGH-LEVEL ARCHITECTURE

3.1 Individual cameras

Although only one camera can be viewed at any particular time, the camera system is actually comprised of several cameras, each of which may operate simultaneously and independently. Using multiple, independent cameras is useful because different viewing situations have different requirement. Attempting to make one general purpose camera that does everything is likely to lead to an overly-complex system.

Additionally, using multiple cameras allows blending camera positions and orientations during transitional cases. This technique is not currently used in FSW, though cinematic transitions use a filtering algorithm that behaves very much like a blend.

In FSW, each camera is derived from an abstract base class, which provides an interface for common functions, such as obtaining the camera world matrix or zoom factor. The "main camera" implements the primary user-controlled view and flyby systems. There are several other cameras for cinematics, in-game events, and the playback system.

3.2 Camera manager

With several cameras in simultaneous operation, it can be difficult to coordinate their activities and their interactions with other game components. FSW has a "camera manager" object which tracks the cameras. All interactions with other game components take place in part through the camera manager.

4 MOTION SYSTEM

4.1 Coordinate system and orientation

Instead of using FSW's actual coordinate system (it is a bit unorthodox), the following coordinate system will be used for the remainder of this paper.

The view coordinate system is a typical right-handed system. Relative to the screen: the positive x-axis points to the right, the positive y-axis points up, and the positive z-axis points into the screen (away from the viewer).

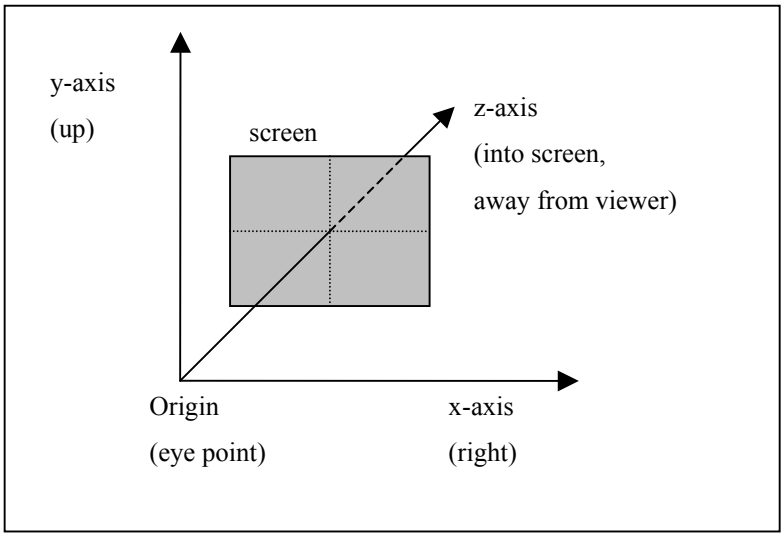

Figure 1: View coordinate system.

The world coordinate system is similar to the view coordinate system: the positive x-axis points east, the positive y-axis points up, and the positive z-axis points north.

Since the camera is always assumed to remain horizontal, the orientation can be specified with only two angles, pan and tilt. Pan rotates around the y-axis, while tilt rotates around the x-axis. Note that, when calculating camera matrices, tilt is applied first, then pan.

It is also worth noting that some cinematic transitions interpolate the camera orientation using quaternions. However, quaternion interpolation is the exception rather than the rule in the FSW camera system.

4.2 Using "target points"

To control the motion of a camera, it is often useful to move the camera using target points. A target point is the final position where the camera should arrive. The camera controller filters the camera position to move the camera to the target point with a smooth motion. Using target points helps separate the problem of choosing good viewing positions from the problem of providing smooth and predictable motion.

4.3 The basic proportional controller (PC)

The PC is common in many camera systems. If you have worked on a camera system, you have probably used a PC, though it might have been called something else--terminology for this particular construct is not very consistent.

The typical implementation is to set the velocity vector, V, equal to a proportion, C, of the difference between the current position, P, and a target position, Pt:

$$V = C(P_t - P)$$

One advantage of the PC is its simplicity. In addition, the PC has a nice *arrival characteristic*. As the current point, P, approaches the target, Pt, the velocity asymptotically approaches zero, following an exponential decay. This gives the PC a "graceful" feeling when it is approaching its destination.

4.4 Drawbacks to the basic PC

Although the PC is simple and has a nice arrival characteristic, it also has significant disadvantages.

4.4.1 The "exit" characteristic can be abrupt.

When the target point, Pt, changes suddenly (as happens when the target is changed from one game object to another), the PC will change velocity instantaneously. This sudden change in velocity makes the camera system feel unnatural.

4.4.2 The current position, P, lags behind moving targets.

When a target is moving, the PC lags behind by the following proportion, which can be found by rearranging the previous equation:

$$Lag = \left|P_t - P\right| = \left|V\right| / C$$

This lag behind moving points leads to tuning issues. When the PC is tuned for a nice "arrival" characteristic, the lag behind moving objects will often be very large. When the PC is then tuned to reduce lag, the arrival characteristic will often be too fast (and the exit characteristic can grow to excessive levels).

4.5 The modified proportional controller (MPC)

To solve the problems encountered with the PC, the FSW camera system uses a *modified proportional controller* ("MPC" for short). The MPC addresses the two drawbacks of the basic PC in the following manner.

4.5.1 The exit characteristic is controlled.

The exit characteristic is controlled by limiting the acceleration of the current position, P. This is a straightforward calculation.

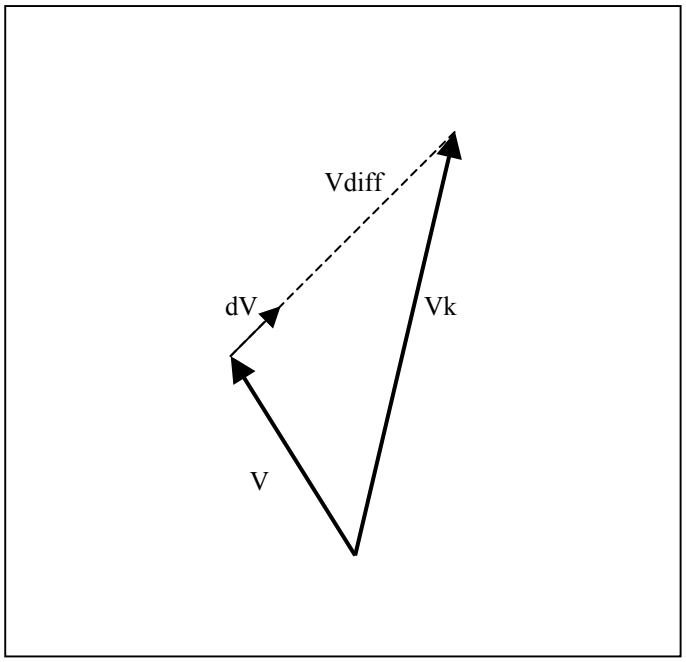

Figure 2: Limiting acceleration. V is the current velocity and Vk is the desired velocity. Vdiff is the current difference and dV is the limited acceleration (delta velocity) for one frame.

Given a current velocity vector, V, and a desired velocity vector, Vk, calculate the difference:

$$V_{diff} = V_k - V$$

Assume that Alim is the acceleration limit (a scalar, not a vector). The maximum change in velocity for one frame is:

$$dV_{\max} = A_{\lim} \bullet dt$$

Now, compare dVmax with Vdiff:

$$if\left(\left|V_{diff}\right| < dV_{\max}\right)$$
$$dV = V_{diff}$$
$$else$$
$$dV = dV_{\max} V_{diff} \Big/ \left|V_{diff}\right|$$

Now, use dV to update the velocity.

$$V' = V + dV$$

4.5.2 Lag is compensated.

Lag is corrected by adding the target point's velocity, Vt, into the desired velocity:

$$V_k = C(P_t - P) + V_t$$

This desired velocity, Vk, is then used to determine the acceleration, which is limited as mentioned above.

4.6 Angular motion

As mentioned previously, camera orientation is parameterized by a pan and a tilt value. The two parameters are updated independently using a one-dimensional version of the MPC described above.

4.7 More about target points

Although the MPC helps solve many of the issues involved in making a camera system operate in a nice, smooth manner, it does not solve every problem, and it can still exhibit problems with certain types of inputs. In particular, when working on the motion system, the following problem arose.

4.7.1 The problem with moving target points

When a moving target point is specified, it is expected that the camera will eventually track exactly onto the moving point, i.e., the camera may have to fly to catch up, but it should eventually be dead-on.

That is a reasonable expectation, however, since the camera has limited acceleration (to keep it moving smoothly), it becomes obvious that, when a target point stops abruptly, it is impossible for the camera to do anything but overshoot (since its acceleration is limited and it cannot “stop on a dime”).

This means there are only two possible choices: either the motion system can lag behind the target to give it a “buffer space” for handling sudden decelerations, or the motion system can track exactly onto target points, but it will overshoot when the target point decelerates rapidly.

4.7.2 Potential solutions

There are rather many potential ways that the MPC could be changed to handle this problem. One option is to ensure that the camera lags behind moving target points. Another is to try to make some sort of "smart" algorithm that overshoots, but then does not attempt to return to the target point.

4.7.3 The FSW solution

There is a deceptively simple alternative to modifying the MPC: avoid the issue by making target points behave reasonably whenever possible. The MPC algorithm remains unchanged, but its inputs are improved. Examples from FSW are:

Target points generated from user controls are acceleration-limited. When the user pans or tilts the camera, the thumbstick inputs do not directly affect the velocity. Instead, the user controls the acceleration, and a resistance value limits the maximum velocity.

Soldier positions are filtered with a PC prior to target-point generation. The MPC is not used in this situation, since the target points will eventually feed into the MPC, anyway. In addition to keeping camera motion smooth, this creates a "lag behind" feature that the FSW designers specifically requested.

5 THE FSW "AUTOLOOK" FEATURE

5.1 What is autolook?

Autolook is a unique feature of the FSW camera system that automatically assists looking around corners.

In most third-person games, the character is centered on the screen or is viewed at a constant offset (e.g., over the right shoulder). However, because of the nature of realistic urban combat, this does not work well for FSW. Soldiers tend to spend a lot of time near walls and the corners of buildings. Consequently, using a view centered on the soldier will often result in over 50% of the FOV obstructed by wall. Autolook solves this problem by determining which part of the view is unobstructed.

5.2 How it works

Horizontal rays (y value is constant) are cast from the camera center towards the left and right of the camera centerline. When rays intersect the environment, the distance to the intersection is multiplied by a weighting factor (which is identical for all rays) and added to (or subtracted from) the angular offset. Lines on the left side of the camera centerline add to the offset, causing the camera to pan right. Lines on the right side are subtracted from the offset, causing the camera to pan left.

Autolook only affects the orientation of the camera. Position is not modified, as changing the position in this case caused disturbing motion. After the autolook angle is calculated, the angle is filtered, preventing jarring motions.

The weighting factor for autolook depends upon the typical viewing distance. This factor is a tune parameter in FSW and was adjusted until an acceptable result was obtained.

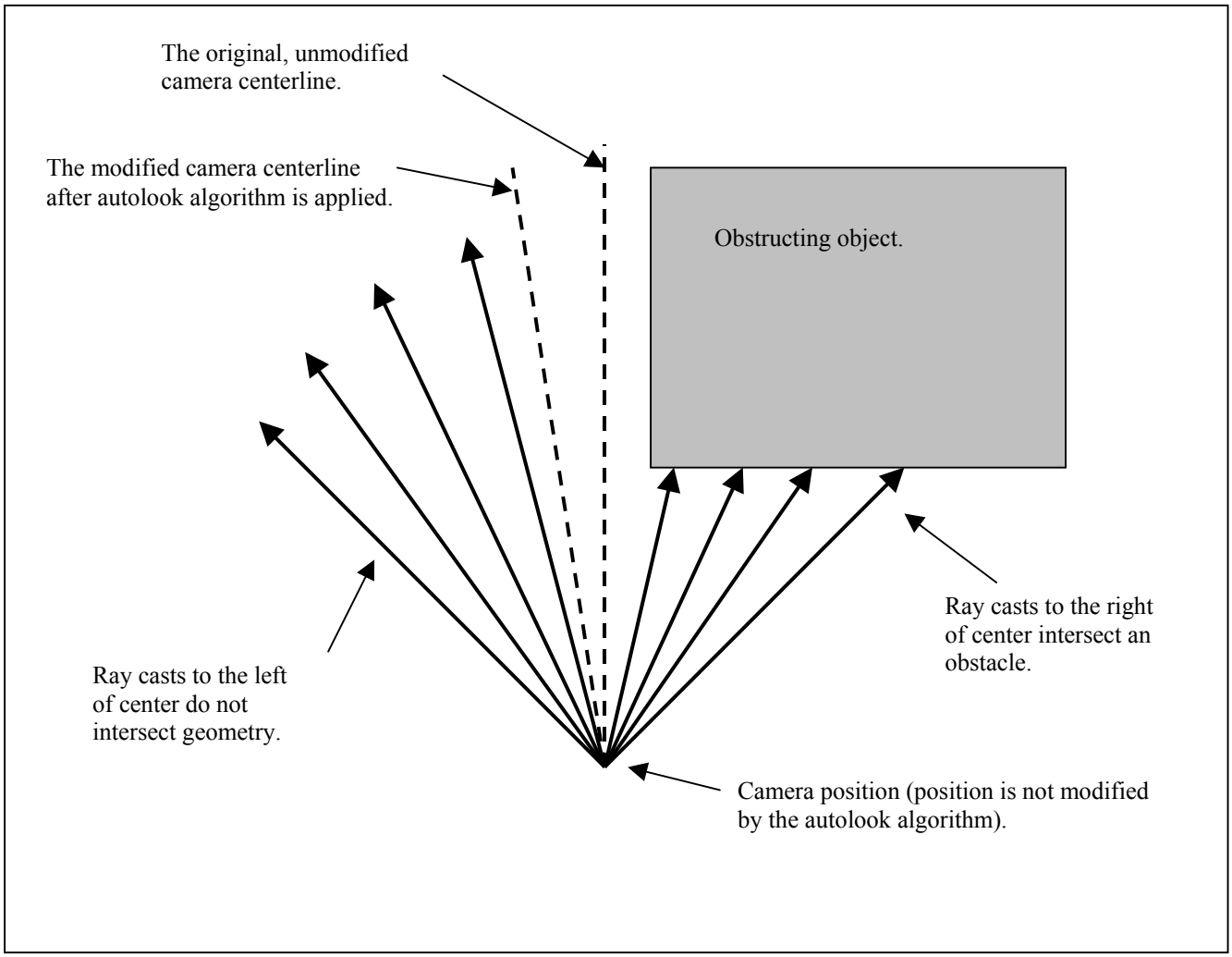

Figure 3: Autolook raycasts (top view).

Image 1: Autolook, before and after comparison. The image on the left has no autolook. Roughly 50% of the screen is occupied by a wall. The image on the right uses autolook. The wall occupies much less screen space, bringing more of the playable area into view. Notice how the soldier is framed to the left instead of center.

5.3 Is autolook appropriate for other types of games?

During playtesting, autolook has been a well-received feature of FSW. But does this mean that it would be useful for other games?

It isn't possible to give a definitive answer, but, since autolook changes the view angle outside the direct control of the user, it is reasonable to assume that any game which requires quick and precise screen-space aiming would not be well suited to this technique.

However, FSW uses a world-space reticle, which removes many of the issues involved with screen-space aiming. Since the reticle moves through world space, the precise position of the camera is not critical. It is only necessary that the camera keep the reticle in view. It remains to be seen if this technique is responsive enough for quick-paced action games, but the technique should work well for slower-paced games.

6 COLLISION AVOIDANCE AND CUTS

The FSW camera system uses various techniques to avoid collisions and to prevent obstruction of the view.

6.1 Normal view collisions

The camera attempts to avoid collisions with the environment by casting several rays from the lookat point towards the general vicinity of the camera. Each ray is cast to points that have the same altitude (y value) as the camera. This creates an arc of an inverted cone of rays over the lookat object (see Figure 4 and Figure 5).

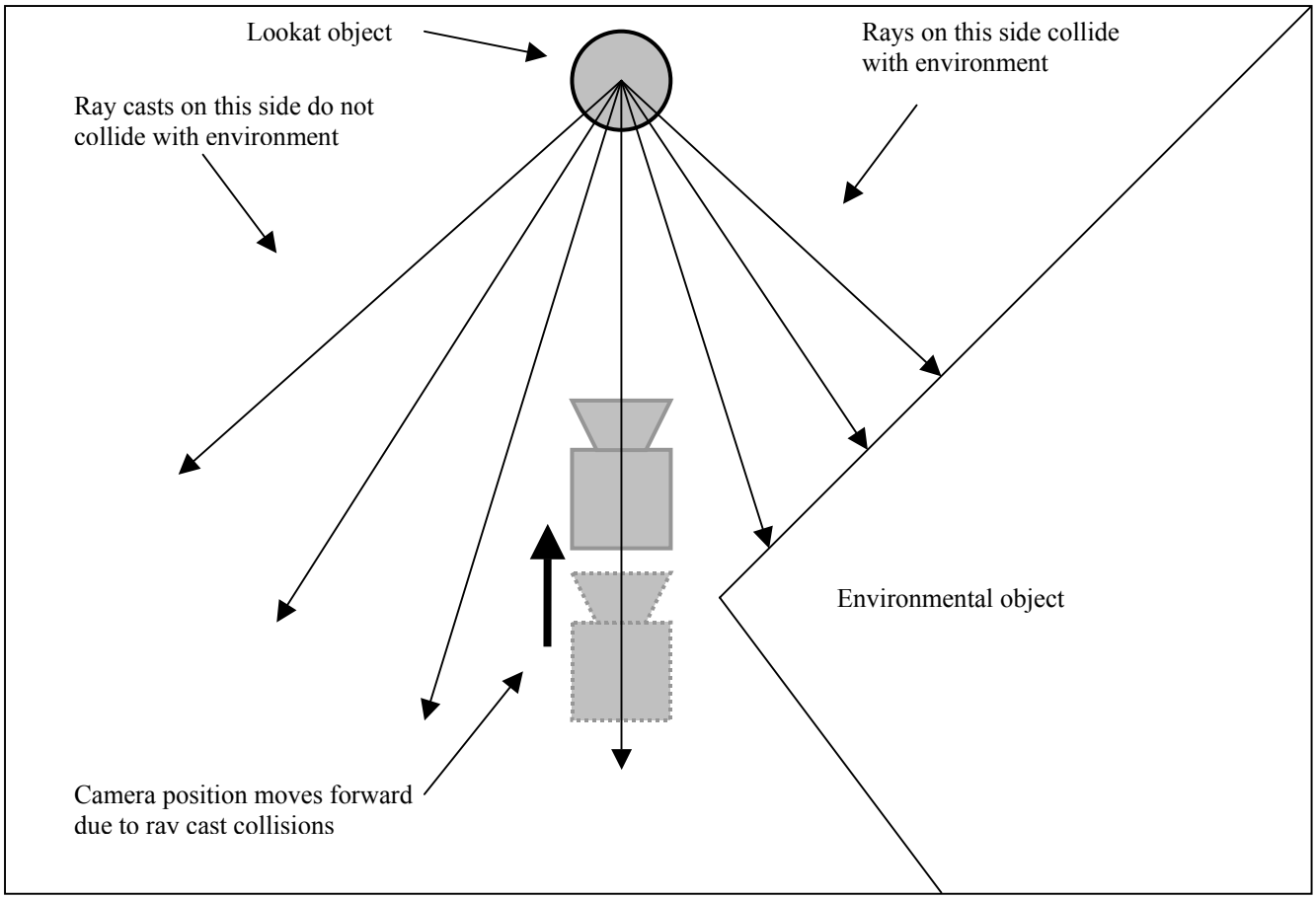

Figure 4: Collision raycasts (top view).

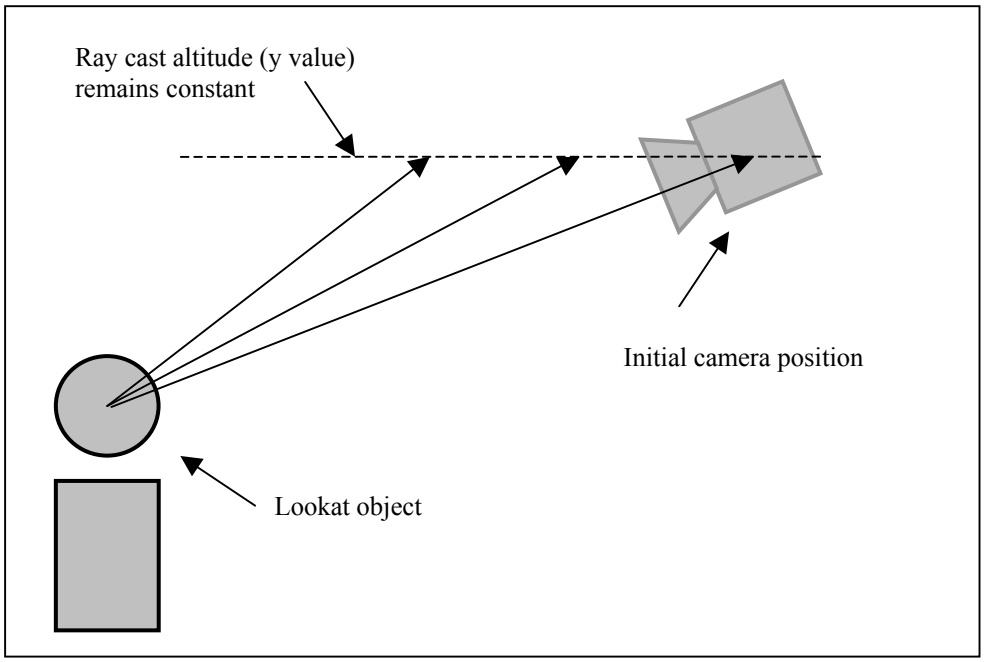

Figure 5: Collision raycasts (side view), showing how an inverted cone of rays is created.

Each ray has a weight that determines how strongly it affects the view radius. The rays nearest the camera affect the view radius 100%. The view radius will never exceed the length of these rays. Rays further from the center have progressively smaller effect on the view radius. The farthest rays have a very weak effect.

The calculated view radius is used to generate the target point for the motion system, which smoothly interpolates the position, preventing jarring motion.

6.2 Flyby collision avoidance

Another algorithm helps the camera avoid walls when it is performing a "flyby" between distant points. This algorithm also casts rays from the lookat point, a single ray to the left and a single ray to the right (see Figure 6). Note that the ray casts may not be horizontal, as they originate at the height (y-position) of the lookat point and end at the height of the camera.

When a ray intersects a wall, it applies a small sideways acceleration to the camera, nudging it away from the wall. The size of the side acceleration is proportional to the distance from the lookat point to the intersection.

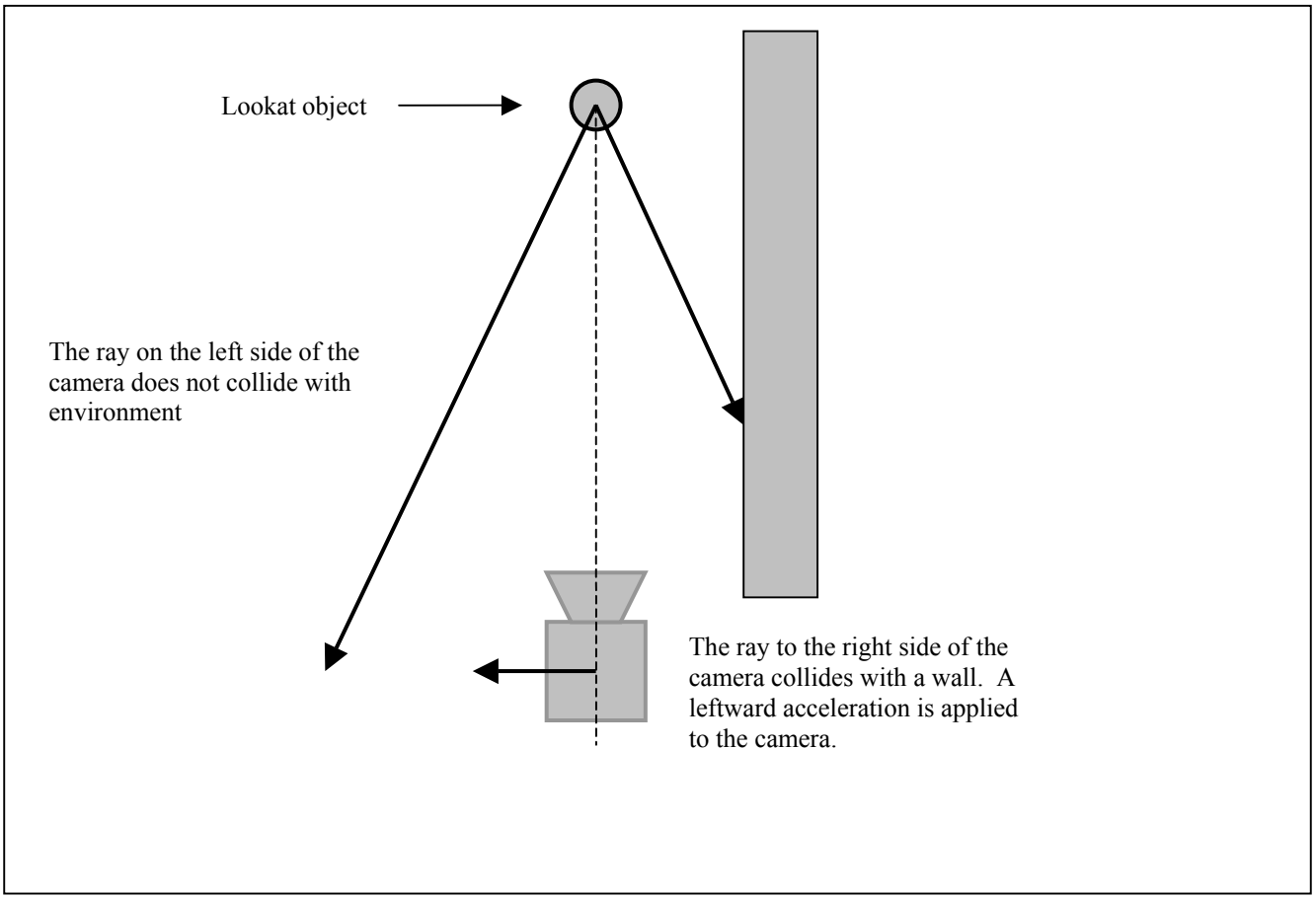

Figure 6: Flyby collision avoidance (top view).

6.3 Lookat avoidance

A "lookat avoidance" algorithm prevents the camera from passing too close to the lookat point, which would otherwise result in rapid and disorienting camera pan. This algorithm analyzes the distance from the lookat point and the approach velocity.

If the camera is in danger of passing too close to the lookat point, its target velocity is modified by reducing its magnitude and moving the vector away from the line between the camera and the lookat point.

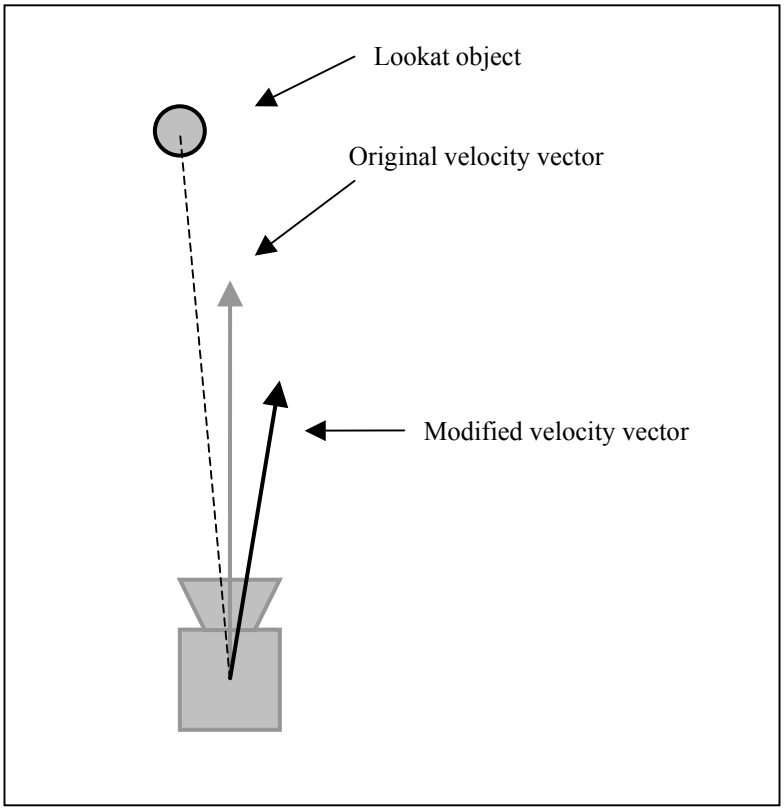

Figure 7: Lookat avoidance (top view). The original target velocity vector is pointing almost directly at the lookat object. The target velocity is modified by reducing its magnitude and moving it farther from the line between the camera and the lookat point.

Although lookat avoidance is an important consideration for many camera systems, its solution is game dependent. Rather than delving into a detailed discussion of the calculations used by the FSW algorithm (which is beyond the scope of this paper), the following summarizes some important things to consider when working on this problem.

First, determine if lookat avoidance is necessary. In general, lookat avoidance is most problematic during flybys. If your system cuts between viewing positions, then lookat avoidance may be a moot point. Even when your system has flybys, lookat avoidance may not be an issue if the camera always flies straight towards objects, since it does not need to pass to the opposite side of viewed objects.

When the camera motion controller has lag, it is possible that moving objects may catch up to the camera in certain circumstances, requiring a different type of lookat avoidance. This problem can usually be avoided in games like FSW, since the camera is attached to human characters whose maximum speed is a quick run. However, this can become a serious issue in games with fast moving vehicles. A potential solution for this particular problem is to apply a repulsive force from the lookat point to the camera. The force grows in inverse proportion to the distance between the camera and the lookat.

If your system does need lookat avoidance, consider what general approach you want to use. The FSW system modifies the camera velocity, but there are other solutions. Your camera can maintain its orientation, fly past the viewed object, then turn to face it, preventing rapid-pan problems. Or, you might be able to move the camera's target position so that the camera no longer needs to pass beyond the viewed object.

Whatever your solution, it is important to take the environment into consideration, since lookat avoidance changes the camera position which may unintentionally cause the camera to intercept environmental geometry.

6.4 Camera cuts

In cases when the destination is on the other side of a wall or obstacle, the camera cuts past the obstruction. When this situation arises, the camera animates as it would for a flyby. It pivots in place (if necessary) to face towards the new lookat point. Then, it begins to fly forward until it is close to an intervening obstacle (usually a wall). The camera cuts to an unobstructed point and continues its flyby.

The camera also performs a cut instead of a flyby whenever the destination is beyond a certain distance. This prevents overly lengthy flybys which would interrupt gameplay.

6.5 Characters and tagged objects

In FSW, the camera does not collide with characters. Any character that obstructs the view becomes translucent. Additionally, environmental objects can be tagged so that the camera does not collide with them. This is generally used for thin objects, such as telephone poles.

7 DEBUGGING AND TUNING AIDS

FSW has over 100 parameters that affect the camera system. Many of these parameters require tuning by a designer or programmer. With such a large number of parameters, it can be difficult to predict how a change in a value will affect the camera system. Indeed, it can even be difficult to comprehend exactly what a value represents and what it is intended to change.

In order to facilitate the tuning process, FSW has debug displays that graphically display aspects of the camera system. Most of the displays indicate ray casts through the environment. The rays used for collision probing are displayed. Rays that intersect the environment are displayed in a different color from those that do not. The raycasts for the autolook system are also displayed.

An additional debug feature that is very useful is the ability to view the scene from any camera. This allows examining the dynamics of a camera by viewing it from another perspective. This is especially useful for examining raycasts, since raycasts often originate near or behind the eye point, where they are difficult or impossible to see from the normal perspective. It can also be useful to view the camera's motion from an alternate perspective, which can elucidate the source of motion glitches.

The camera system timescale can also be altered to allow closer examination of glitches. Sometimes a slow-motion examination will reveal details that are not visible at normal speed.

8 OTHER CONSIDERATIONS

8.1 Gameplay record and playback

FSW can record and playback game sessions. This is done by recording the input stream and a minimal amount of other critical information (such as timestamps). To re-create the original gameplay the game is started and the original input stream is fed back into the system. In order for this type of playback system to function properly, the gameplay components must be repeatable, behaving identically during a replay as during the original recorded session.

However, the user can view from different angles in replay, which means that the camera will not be at the same position as it was during the original session. This could disrupt playback, since the camera affects the operation of UI components (e.g., the movement cursor operates relative to the main camera). This problem is solved by adding special *replay cameras* that function only during replays (they are inactive during normal gameplay). The original gameplay cameras repeat their same behaviors during replay even when the user is not viewing them.

8.2 Field of view (FOV)

As with many features of a game camera system, the field of view (FOV) affects both aesthetics and usability. Note that, in the following, FOV is measured horizontally (the vertical FOV is determined by the aspect ratio of the screen).

In FSW, the typical gameplay FOV is 80 degrees wide. This is wider than in many games (it seems that 60 degrees is the "unwritten standard"). However, the wider FOV provides a better view of the actions and situations in the game, which is helpful in a game like FSW where the user must evaluate and plan strategies involving a large portion of the play area while viewing from a close third-person perspective. An even wider FOV might have been useful, but after 80 degrees, the distorted perspective is too unnerving for most players.

FSW changes the FOV in two situations: (1) The player can zoom the camera to get a closer view of enemies. The result is a reduced FOV, (2) FOV is changed during cinematics, again to provide a zoom effect.

9 LIMITATIONS OF THE SYSTEM AND POTENTIAL IMPROVEMENTS

Okay, so now you've seen the best parts of the FSW camera system. Unfortunately, like all things, it is not perfect. Here are some of the limitations of the FSW camera system and what has been done about them (if anything).

The biggest limitation of the FSW camera system is that the main camera can temporarily clip through walls in certain situations. These problems are always temporary--the camera never gets "hung up" on geometry. The situation is fairly rare, and usually occurs only during a flyby. It would be nice to fix for aesthetics, but there is the risk of introducing worse problems in the process. In the end, we have decided to live with the issue.

A second limitation of the system is that it does not provide a good playing perspective in tight spaces. This is not a major concern for FSW, since tight spaces have been avoided by design for a multitude of reasons unrelated to the camera system.

Although the main camera satisfied the requirements of the gameplay camera system, it is also very complex. Part of its complexity is due to serving more than one purpose in the game. It handles the user-controlled pan/tilt operation, flyby functionality, plus a portion of the cinematics system. Splitting the main camera along those functional lines to create multiple cameras would simplify code and, more importantly, would simplify the tuning process.

10 RECOMMENDATIONS

When aesthetics and usability are in conflict, always choose usability. The camera system is too important to fundamental gameplay to allow a cool effect to take precedence over the user's ability to understand and interact with the game. Once the system is stable and functions acceptably for users, then you can experiment with aesthetics. Of course, you can pretty much go wild during cut scenes, so it would be wise to concentrate aesthetic concerns there.

Keep motions smooth, e.g., use the MPC described above.

Don't pan the camera quickly except under user control, since it can disorient the user.

Keep user-control separated from programmatic control as much as possible. When the camera is under user control, give the user as much freedom of motion as possible. Restricting motion feels very unnatural and can make players feel that something has gone wrong.

Use a layered design and multiple cameras.

Provide visual feedback, tuning, and debugging aids.

Always look for alternative solutions to problems. Often, issues with the camera can be resolved by means outside of the camera system itself, such as using translucency to prevent camera obstruction rather than colliding with characters. Sometimes, these alternate solutions are preferred by design, since they are straightforward and provide more information for players (e.g., players will see a translucent character whereas a collision system may move the camera in a way that the user does not understand, since he cannot see the character causing the collision).

11 FUTURE DIRECTIONS

In general, it would be nice to see more published papers on the topic of real-time camera systems, especially for games.

Integrating traditional cinematic camera ideas (from the film industry) into real time systems. This is an interesting topic, since games often have different requirements from traditional film. However, there is over a century of filmmaking

experience that we can tap into. At the same time, this must be approached with caution--implementing cinematic camera system without proper concern for gameplay requirements could be devastating for gameplay.

Volumetric examination of the environment instead of linear ray-hits. This might improve the operation of various systems, such as collision avoidance and autolook.

Examine more techniques for providing feedback during the tuning process.

12 REFERENCES

Bobick, Nick, "Rotating Objects Using Quaternions", Game Developer Magazine, April 1998

Drucker, Steven M., Zeltzer, David, “Intelligent Camera Control in a Virtual Environment”, Proceedings of Graphics Interface ‘94

Hawkins, Brian, "Creating an Event-Driven Cinematic Camera" (two parts), Game Developer Magazine, October/November 2002

He, Li-wei, Cohen, Michael F., and Salesin, David H., "The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing", SIGGRAPH 96 Proceedings, pp 217-224.

Rabin, Steve, "Classic Super Mario 64 Third-Person Control and Animation", Game Programming Gems II, Charles River Media, 2001

Treglia II, Dante, "Camera Control Techniques", Game Programming Gems, Charles River Media, 2000