47

Tips and Tricks for a Robust Third-Person Camera System

Eric Martel

47.1 Introduction

Working on the camera system for a third-person game is technically challenging, yet critically important to the success of the title. Not only are your teammates relying on you to show off their amazing artwork and level design, but a major part of the player's experience is defined by the way he perceives the environment, which is mostly driven by the camera. At the same time, game development is highly dynamic. Everything from the game design to the art assets themselves can and will change during production. A robust camera system must be easily modifiable and adaptive. It should typically keep the player's character on screen and do its best to make the player aware of dangers in the environment. In this section we explore the tools and techniques that helped us build the camera system for one of the most successful action adventure games.

47.2 Understanding What's Going On

It is often extremely difficult to understand what's going on with the camera, particularly since it is our primary point of view into the game. As a result, effective debug and visualization tools are critical to a good camera system. These tools are going to enable you to

investigate problematic situations, tune the system's performance, and make your game's visual artistry shine.

47.2.1 Debug Display

The first feature to implement is a debug display, which includes both 2D and 3D features. The 2D should simply write text to the screen that exposes information such as the active camera set and their state, the selected camera's position, orientation, and FOV (field of view). You can extend the list of rendered information as you see fit with internal data to the system like environmental triggers used, the blending state of the cameras, and so forth. All of these should update in real time, so you can see when and how the information changes.

The 3D debug display draws its information graphically to the screen using a combination of color codes and primitives to easily understand a situation. For example, you could use a pyramid to display the position and the orientation of each camera: blue when the camera is active, black when it is not. When using the debug camera, these will allow you to easily understand what is going on with your gameplay cameras. We will discuss later in this article the different ways you can test the environment surrounding the camera and the player. For each of these tests you should choose different primitive and color combinations to understand the scene at a glance. Collisions are another example of displayable information. Collision geometries are often different from the meshes visible on screen. These two representations for each object can get out of synch—for example, if an artist updates the look of an object but forgets to change the collision geometry to match. Since the collision geometry is not typically displayed, it is often quite difficult to recognize situations where this is the problem—all you see is that the camera is behaving badly. If you display a red cylinder for every collision test that fails, it is much easier to spot situations where these are out of place, allowing you to quickly debug the problem and assign the issue to whoever is best fit to fix it.

You should also consider the lifetime of each piece of information in the debug display. While some text could remain permanently on screen (e.g., the position and orientation of the camera), it is often useful to provide hotkeys for turning on just the information that you need. Additionally, 3D displays tend to clobber the view after a while. Usually, the best options are to either give each debug display object a lifetime—so, for example, the red collision cylinders might stay on screen for 5 seconds—or use some sort of "Etch a Sketch" list of display items that only gets cleared when you send a command to do so. The latter is probably better when trying to find a solution to a problem, as you don't have to think quickly before your display disappears. Implementing both doesn't take much time, and the method can be selected from a command, hotkey, or cheat menu option.

47.2.2 Debug Camera

It is often useful to have the ability to "step outside" of the camera system when debugging, and to watch the behavior of the gameplay camera from an external point of view. In order to do this, you can implement a separate debug camera that is under explicit control of the user. When we implemented our debug camera, we placed the controls for our debug camera on a second controller so as to keep the player and camera movement on the main controller as consistent as possible. We used the left stick to control forward, backward, and strafe left and right movement of the camera, and the right stick to change the orientation. We also used the triggers to pan up and down, the start button to

enable/disable the debug camera, and the back button to reset the camera to the gameplay camera's parameters.

We quickly found many other uses for the debug camera. For example, animators would use it to see their animations from a different perspective, level artists would simply go inspect the mesh they wanted without actually having to play the game, and marketing guys would create in-game trailers with one person playing the game and another controlling the camera. The marketing usage required some changes to the system, such as adding controls to roll the camera, modify the field of view, and control its movement speed, but in retrospect it was well worth the time invested.

47.3 Managing Multiple Cameras

If your gameplay changes during the game, you might want to have specific cameras that are built optimally for each situation. For example, when running, the player doesn't care about the same things as when he's using cover; therefore, it's the camera's responsibility to adjust automatically to reflect these changed priorities and provide an optimal view to guide him.

47.3.1 View Manager

The view manager is responsible for providing the renderer with the position, orientation, and field of view to use when drawing the scene. If you decide to implement a debug camera, this would also be a good place to switch to and from it. Finally, this manager is responsible for updating all the active cameras and keeping track of the transitions between the cameras.

47.3.2 Competing Cameras

In addition to a list of available cameras, the camera system can store activation rules to prioritize and pick the most relevant of them. This approach will allow you to add more depth to your gameplay, or even to expose control to your designers. For example, you might give level designers the ability to add triggers in the environment that enable or disable specific cameras, such as cameras introducing the level to the player or a boss fight camera.

47.3.3 Transitions

When a camera change event occurs, there needs to be a transition from the previous camera to the new one. Cutting from one view to the other can feel abrupt, but can be necessary when the orientation between the two views is more than 90 degrees. It can also make sense in some level designs to have camera cuts when the view comes from different discrete point of views, such as another character's point of view or security cameras. Otherwise, it is usually better to blend between the views by linearly interpolating the parameters such as position, orientation, and field of view. More complex interpolation methods can be used if you have specific needs.

It is worth putting some thought into the logic which will drive your transitions. You can begin by giving each camera default transition in and transition out parameters, which specify the transition type and blend time. We settled on using the minimum value of the in and out parameters pair for blend time as our gameplay indicated that when a camera

required to blend in or out quickly, this should have precedence over the other camera's slower reaction time. You will probably also want to have the ability to provide specific transition parameters for specific situations or between specific pairs of cameras. For example, you might want to use a very fast blend when switching from the walking camera to the running camera, even though these cameras would normally transition more slowly.

When activating a camera, it's often useful to pass in the state of the previous camera, so that it can update its internal state to match as closely as possible its predecessor. For example, if you change from a standard camera to one farther away with a sharper field of view when a trigger is held, you will want to set the position and orientation of the trigger camera so that the center of view matches that of the previous camera. You could keep them in synch at all times, but it is simpler to just match them up when the transition begins.

47.4 Input Transform

When controlling a character, the player's input will usually be relative to the in-game character he's controlling or to the view. In this article we'll only discuss the latter. Transforming the input into the camera's reference is straightforward—the vector representing the input gets transformed using the view's rotation, but maintaining that transform as the view changes can be quite a bit more challenging. There are two cases: continuous changes over time (such as when the camera is moving or rotating around the player, or when we're blending between cameras) and discrete changes (such as a camera cut).

Our experience has been that when continuously changing the view reference, the player is typically able to adapt and gradually adjusts the input to the new view. In other words, when the player is given visual feedback that is not exactly what he wants, he will automatically move the stick slightly, often without even noticing, to adjust the view appropriately.

Discrete changes are harder to handle. The camera cut entirely changes the reference the player has to the virtual world. As a result, transforming the input using the view after the cut will often result in incorrect input for a short time while the player familiarizes himself with the new view. One solution is to retain the previous view's matrix, and to use this matrix for the transformation as long as the input is held. As soon as the stick is released, the system can switch to using the matrix for the new view. This leaves the player in a comfortable situation where his input will not be drastically changed after a single frame. It also helps avoiding the "ping-pong" effect, when you have two triggers with cameras facing each other. For example, imagine that the player crosses the threshold between two cameras, which face in opposite directions, by pressing up on the stick. If we switched immediately to the matrix of the new camera, and the player did not let go of the stick as soon as the camera changed, then the direction that his character is moving would reverse, causing him to pop right back over the threshold, and back to the first camera. In this way, he could find himself bouncing between the two cameras very rapidly unless he approached the threshold slowly and carefully, with just delicate taps on the stick.

47.5 Configuration

Not specific to cameras, this tip is simply that anything that might change during the production of the game should be an exposed setting. You don't want to waste your time recompiling just to test new configurations.

47.5.1 Data Driven Approach

From the list of default cameras to their settings, everything should be controllable through data. Whether you use .ini files, XML files, objects in the editor, or even an exposed scripting language such as Lua or Python, you need to provide the ability for anyone working with the cameras (including you) to be able to change them efficiently without requiring a new build. Using a factory design pattern, it should be relatively easy to have every part of the configuration settings constructed from a file or object.

47.5.2 Activation Rules

Since you might want to have multiple cameras of the same type with different configurations active at different times, it makes sense to expose the rules of activation in the data as well. Depending on the variety of gameplay you have and the flexibility you want to give to your system, you can create small objects that poll game states internally, like "player is riding a horse at full speed." You can also create a more complex system that exposes the logic to the person configuring the cameras, where a particular camera is activated when the target character "is on a horse" and "is at full speed." That's what we did, and by allowing first level logic operators, it was really easy to test complex combinations of game states. It allowed us to save a lot of time when iterating on the various cameras in the game.

47.5.3 Priorities

Since the activation rules do nothing to prevent more than one camera being active at the same time, we need to find a way to select which one to use. There really is no reason to have something complex here, so having a simple number defining the priority for each camera should be good enough for most projects. To simplify the task of adding new cameras it is often better to bundle the priorities into ranges, giving a hint to new camera creators what sort of priority they should assign to their asset.

47.5.4 Object References

Adding an object wrapper can be a good way to enable you to specify a variety of different objects in data. You might find that you need to support a variety of different types of objects as reference points for your camera, including dynamic objects such as the player or another character, or even pieces of equipment that the player is carrying (such as a weapon), as well as fixed objects (such as security cameras) that can be placed directly in the level. You should provide a mechanism for specifying these objects in data, so with a unique name or by building a tool that lets you specify them from a drop-down. In-game, the cameras will then be able to look up the positions and orientation of their reference objects, and set their own position and orientation accordingly.

47.6 Camera Behaviors

In this section we will discuss various camera behaviors which are common in third-person games. Some of these behaviors are very simple but are really useful to prototype gameplay. By exposing these behaviors in data, we can provide the game designers and level designers with the tools that they need to configure the cameras on their own. This allows

us to spend our time developing new features and debugging existing code, and also maximizes their control over the player's experience in the game.

47.6.1 Fixed Camera and Tracking Camera

The simplest camera possible is a fixed one. It's defined by a position and an orientation. Even though it's not doing much, it can still be used for menus or to present a challenge or objective to the player. By adding a target object to the camera, you can easily turn it into a tracking camera; its position remains the same but it reorients to frame the target. For example, a tracking camera with the player as a target might be used to simulate a security camera that has caught the player as he enters a restricted area.

47.6.2 Orbit Camera

Many action adventure games allow the default camera to orbit around the main character, with the player controlling the camera's azimuthal and polar angles using the right stick on the controller. The simplest way to represent this is to use spherical coordinates [Stone 04] relative to the target (i.e., the player), since the camera should always lie on the surrounding sphere.

In order to facilitate navigation, any time that the player is not actively rotating the camera and the character is in motion, it is better to blend the azimuthal angle back to 0 (behind the character) and the polar angle to some default value. This allows the camera to smoothly reset to a position that will allow the player to see where they are going without requiring him to micromanage that control if, for example, he was examining the ceiling or floor when he came under attack and decided to flee.

47.6.3 Over-the-Shoulder Camera

Most third-person shooters will use a camera that is directly behind the main character's shoulder and that always follows the character's facing. This type of camera, even though it's in third-person, mostly reacts like a first-person camera that had been moved behind the character by a couple of meters.

The complexity in this kind of camera is not related to the camera itself, but how you handle the targeting and firing of projectiles. If you're using the center of the screen as the aiming target, you will see that depending on the distance between the character and the aimed position, the direction taken by the projectiles will differ. The closer the target position is, the more the position offset of the character will increase the angle difference between the camera's facing and the character's aiming, as seen in Figure 47.1. For very close targets, it will most likely look pretty stupid. A simple solution is to add a layer of inverse kinematics to dynamically adapt the aiming direction of your character depending on which object is targeted at any moment.

47.6.4 First-Person Camera

Even though your game is in third-person, it is always useful to allow the player to go into first-person, as some players prefer this view to analyze their surroundings. This camera is quite simple to implement; its position can be fixed relative to the character, or attached to the head bone (or a specifically animated bone). The latter approach can provide a more realistic camera movement, including breathing and head bobbing as the character moves.

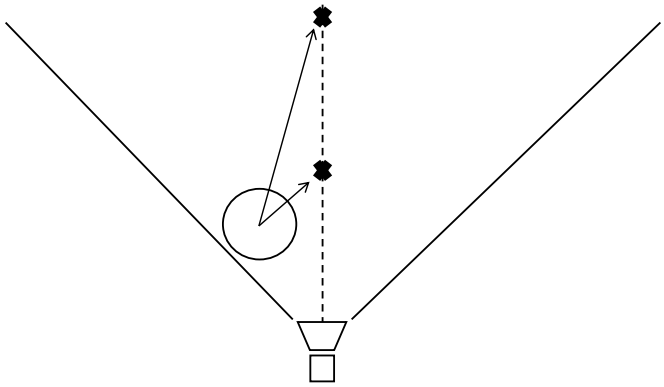

Figure 47.1

Top view showing aiming differences based on distance from the camera.

47.6.5 Camera on Rails

In the movies, they will often mount the camera on a cart that travels on rails in order to create extremely smooth movement. We can take a similar approach procedurally, even using splines to describe the curve of the "rails." This can provide very smooth performance when panning or zooming, and is particularly useful for cut scenes or for the camera in side-scrollers. The usual approach is to define the track that the camera will follow (whether this is a spline or a series of curves and line segments), and then provide a method which takes an input between 0 and 1 and maps that to the corresponding position (and, if appropriate, orientation) along the track. You can then move the camera along the track simply by interpolating the input value smoothly from 0 to 1. An easy way to obtain such a value is to add a box surrounding your play area and calculate the relative position of your target inside that box. Using a single axis, you can divide the relative position on that axis with its length to obtain the mapping you are looking for, giving you continuous values between 0 and 1 when the target moves, for example, from left to right in the level.

47.7 Advanced Features

This section will focus on a collection of features that can be added to your base classes, which can be toggled on or off through your camera settings in order to make your cameras more artistically pleasing. In our implementation, most of these were added late in the project at minimal cost but adding them sooner would allow the artists and level designers to make more widespread use of the advanced features.

47.7.1 Spline for the Vertical Axis

When dealing with high- or low-angle shots on an orbit camera, it is possible that your art director will ask you to tweak the distance of the camera depending on the angle. Usually, a closer shot for a low-angle and moving the camera away for a high-angle shot will give better results. When we introduced the orbit camera, we said the distance should be constant, and we talked about a polar angle. Let's get rid of these concepts and imagine a spline that can rotate around the target on the vertical axis. This curve defines both the

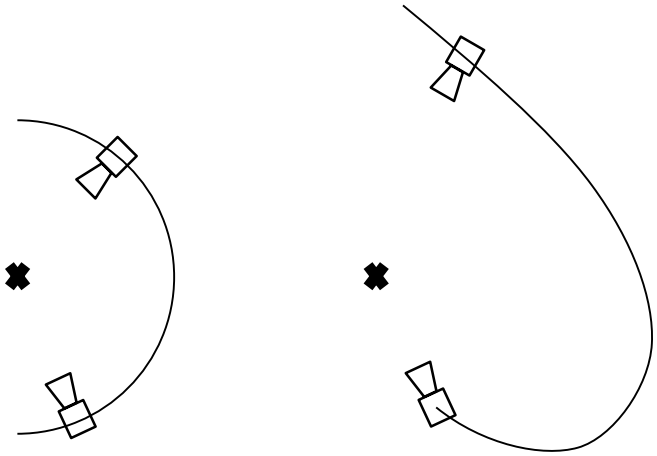

Figure 47.2

Comparison of a standard orbit camera to a camera using a spline as its vertical axis.

distance and the angle the camera should have. The right stick controls remain the same for the horizontal movement, but for the vertical axis we want to control a value between 0 and 1 that maps the position of the camera along the spline (Figure 47.2).

47.7.2 Using Bones

Sometimes, specific actions require some very specific camera movement. For example, you might want the camera to go near the ground when the player performs a duck-and-roll action. Instead of having to program this behavior, you could allow the animators to take control of the position and heading of the camera by animating camera-specific bones. Other settings such as the field of view or the depth of field can be exposed for more cinematographic effects.

Another way to improve the system with bones is to expose a bone property in our object reference system. This way, whenever we're using the object reference to get a position or orientation, it could instead fetch the values from the object's skeleton, if available; otherwise, return a recognizable and valid value that will be helpful in case you need to debug it. You never want to feed the renderer invalid positions or orientations as it will break the view matrix, resulting in undefined behavior.

47.7.3 Sweet Spot

A "sweet spot" is an area on the screen where the target (typically the player character) should be. When enabled, the sweet spot system will only allow the camera to rotate on its own when the target exits this area. This technique helps to stabilize the camera when the target makes subtle movements. The size of the zone can be adjusted to absorb bigger movements, such as crouching or jumping, or focused down to make the camera feel more dynamic. When the player was actively rotating the camera, we would simply recenter the view on its target and disable the sweet spot.

Another use we made of this system is to smoothly frame the enemies around the player during fights, displaying as many as possible while ensuring that the closest enemies (and

the player's character, of course!) were always visible. We did this by sorting the closest *n* enemies by distance and then readjust the camera's heading on each of them, making the smallest change necessary to place each one in the sweet spot. The last heading adjustment was always for the player. As a result, the player would always be on screen, and we would get as many of the enemies as possible while giving preference to the closest ones. This also allowed enemies to join and leave without affecting the view too much.

The area used to define the sweet spot could be anything, but we used a rectangle. This was sufficient to get good-looking performance, and was easy to work with because we can simply convert the screen space rectangle into angles, which can then be used with two simple dot products to determine if the target is in or out of the sweet spot.

47.8 Collision System

The complexity of most camera systems lies in the way the camera interacts with the environment—which is to say, the way that we handle situations where the camera collides with the level's geometry or when objects get in between the camera and the player character, causing occlusion. Games that place the cameras on rails may not need to deal with collisions at all, since the level designer is responsible for creating camera paths that aren't obstructed by the geometry. In many cases over-the-shoulder cameras can simply move closer to the player as needed, but collisions can present a number of challenges for orbit cameras. The remainder of this section will focus on handling those problems.

47.8.1 Collision Flags

We tried to base our collision checks on the collision system that was already in the engine for the physics and AI needs. We found that in some cases the camera's needs were unique, however. As a result, we added collision and occlusion flags that could be used by the modelers to annotate objects appropriately. This made it possible to have the camera ignore small posts, for example, while considering foliage as occluding the view but not blocking camera movement.

47.8.2 Camera Size

Mathematically speaking, your camera is a one-dimensional point; it has no size. Without providing a size, however, graphical issues can arise if you use a very small near clip and the camera is too close to the world geometry. What you might see are large polygons taking up a good portion of the screen. To remedy this problem, one option is to increase the near clip, which should cull the geometry that's too close to the camera. Another approach is to ensure that every movement the camera makes is padded with a virtual size, so that it always keeps a minimum distance from collisions, which could be achieved by moving around a primitive in your physics system. On our project we used the latter but, in retrospect, considering the usage of our camera, it would have been far simpler just to adjust the near clip.

47.8.3 Collision Reaction

When handling collision reactions, we have two cases to worry about: the player driving the camera into a collision and the camera colliding with the environment while autonomously following the player. We describe the approaches that we took for each

case below, but you may need to experiment in order to find the solution that best suits your game.

When controlled by the player, our orbit camera would simply reduce its distance to the target in order to avoid colliding with obstacles. Our goal was to keep the player's input intact by keeping the changes in azimuth and position along the spline, but when transforming the camera's position, we would take the point of impact and the camera size to compute a new distance to the target. As the player would walk away from the collision, the camera would therefore keep the same point of view and start following at the original distance once the collision is cleared.

We handled collisions during navigation a bit differently. Since our camera tried to stay behind the player as best as possible, our most common source of collision was when the player was backing towards a wall. The method used was to smoothly modify the azimuth angle of the camera in the direction of the collision's normal vector if it would allow the camera to be closer to its desired distance. This way, we avoided oscillation in corners while moving the camera horizontally along the walls until we were clear of collision.

47.8.4 Occlusion Reaction

For the occlusion reaction, we have the same two cases as for collision, except that when handling user input we decided not to do anything; if the player wanted to occlude his character, it was his choice and we allowed it.

In our case, we only used static geometry for occlusion; therefore, if an occlusion occurred, it had to be because the player moved in a way to insert geometry between his character and the camera's position. We used what seemed to be the simplest way to solve this problem, by compensating to the player's movement by moving the camera in the opposite direction. In the case of vertical movement, we simply moved the camera along the spline. Sometimes, the simplest solutions give the best results, and I believe this is a good example!

47.9 Conclusion

When writing your own camera system, give yourself the proper tools to facilitate your job. Visualization is a very important aspect to understand the interaction between the camera, its targets, and the environment. Make the system as configurable as possible and encourage your designers and animators to play with it. The more they work with it on their own, the more time you have to develop features and debug code, and the more they will feel like they have ownership of the player's experience. When building the actual camera behaviors, try to think what you'd want as a player and make other people test them out before submitting your code. Remember that the camera is a central part of the game, everybody is affected by it, and as such it is critical to get as many people to try it and give you feedback as possible. Always think of the player first and build your cameras accordingly, because the cameras are a vital part of the gameplay!

References

[Stone 04] J. Stone. "Third-person camera navigation." In *Game Programming Gems 4*, edited by Andrew Kirmse. Boston, MA: Charles River Media, 2004, pp. 303–314.