Overview

The Adventure Camera & Rig is a multi-behavior camera built specifically for quality 3rd Person Action/Adventure games. Use it as a basis for your custom camera system or out-of-the-box to kick start your own adventure!

While the 'ReadMe' document is meant to give you a quick intro, this document is meant to help you understand the code and give you a basis for customizing the camera (if needed).

In the end, the code and tips in this document could be used to create any type of camera.

Concepts

Disconnecting the Camera

As much as possible, we want to disconnect the camera from the avatar that it's following. However, since it is following the avatar, we can't do that completely.

The reason we want to disconnect the camera is so that the camera doesn't have any of the rocking or jittery motion that is sometimes found in the avatar animation and root motion.

> Imagine a camera is attached to a dune-buggy with a solid steel bar. You can image that every bump the buggy hits is going to be immediately transferred over to the camera. The recording is going to be super jittery.

This jittering comes in two forms. First is from the raw positioning of the camera and second is from the rotation of the camera.

There's a couple of things we can do mute the jittering:

1. Use a "lerp" to dull the camera movement and rotation. This works good most of the time, but you can still get into a situation where the camera sways and throbs back and forth.

2. Ensure the physics engine is iterating over the simulation. Do this by assigning a Rigid Body to the avatar and setting the "Interpolate" property to "Interpolate".

3. If the camera controls the view (while in First Person mode), ensure that the avatar rotations are applied within the frame. This means forcing the avatar to rotate in the camera's LateUpdate() function. While this isn't ideal, it will keep the avatar rotation and camera rotations in synch.

4. Don't "look at" the avatar directly. Instead of using the "LookAt" function to rotate the camera, calculate the rotation based on the yaw and pitch the player sets as they are rotating the view. This works great since the camera won't pick up any animation data.

5. Disconnect the camera from any "solid steel bar". Instead, use something like a physics based spring camera. See below.

Spring Camera & Smoothing

When building a camera that moves, you'll want to create some sort of smoothing function. In a lot of cases, a "lerp" or Unity's "SmoothDamp" function will work great.

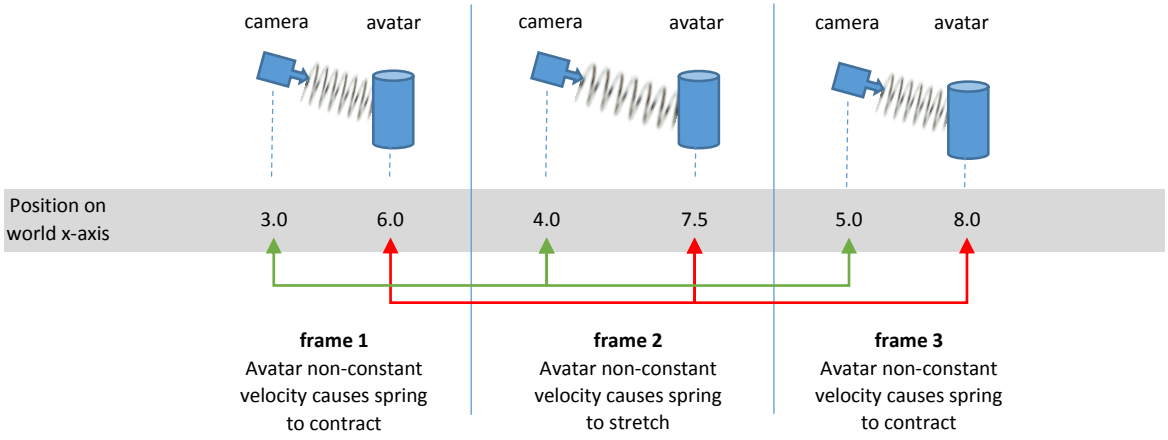
However, with a camera that is attached to an avatar that can move at random speeds, you need something more. This is especially true when using root-motion that hasn't been smoothed.

A physics based 'spring' helps to minimize this because the camera's velocity isn't tied directly to the avatar. Instead, it tries to reach the desired position on its own "near constant" velocity. Most variations in the avatar velocity are absorbed by the spring.

Take for example a camera that is always meant to be 3-meters from the avatar. Then, when the avatar moves at a non-constant velocity (which is pretty standard for motion captured root-motion data), the camera will move at a non-constant velocity.

When your camera moves at a non-constant velocity, the world looks like its bouncing or throbbing. This is because the camera is slowing down and speeding up each frame.

With a spring based approach, the spring absorbs most of the non-constant movement of the avatar and the camera's physics cause it to move at a smoother velocity towards the target spot.



Avatar Stances

Camera designers use lots of different types of cameras to create the experience you get in AAA quality games. While the player isn't really aware of this, the camera is changing to meet the situation.

Tomb Raider's Lead Camera Designer talked about this at the 2013 GDC. You can see his presentation here: http://gdcvault.com/play/1018141/Creating-an-Emotionally-Engaging-Camera

The Adventure Camera & Rig follows this methodology by creating 3 'stances' and 3 different types of cameras.

A stance is simply a mode that the player is in. It can be a subconscious mode or one explicitly set by the developer. The 3 stances are: Exploration Stance, Melee Stance, and Targeting Stance.

Each of these stances causes the camera to react in a somewhat different, but natural way.

Exploration Stance

In this stance, the player is mostly concerned with moving around the environment. As with most modern action/adventure games, the player is able to run forward, to the left, to the right, and even towards the camera.

This works because the camera view stays somewhat constant as the player moves; grounding them in the scene. In this mode, the camera works as a '**3rd person follow**' camera.

One of the key things to note is that the player actually rotates around the camera. While the camera will follow the player, if the player starts to turn left or right, they actually run in a circle around the camera.

This really helps with maneuvering the avatar through obstacles when using a gamepad.

The exploration stance is where you'd expect the player to do jumping and climbing (which are not implemented in this project).

If the player uses a mouse and keyboard, the camera changes to a traditional '**3rd person trailing**' camera. This is an 'always behind' view where the camera rotates around the avatar. It works because the player is always moving forward in relation to the camera view.

Melee Stance

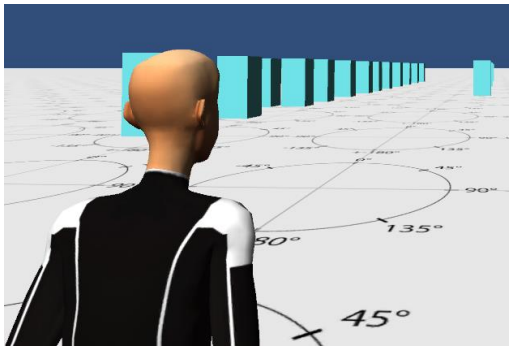
The melee stance is used to keep the player always facing forward.

This is important if you're making a hand-to-hand combat game where avatars are up close and personal. In this mode, you typically don't want the avatar to turn around and run towards the camera. Pulling back on the gamepad usually means you are simply trying to back up some.

So instead of pivoting the avatar, this mode always keeps the player facing forward with the camera. While still able to run forward, the avatar can strafe left and right or move backwards.

This stance uses the same camera types as the Exploration Stance. In fact, when moving forward the player won't notice a difference.

Targeting Stance



The targeting stance moves the camera from behind the avatar to the right. This stance is used with ranged combat, casting spells, or needing to target something in the environment.

You typically see this in games like Tomb Raider when she pulls out her bow or pistol.

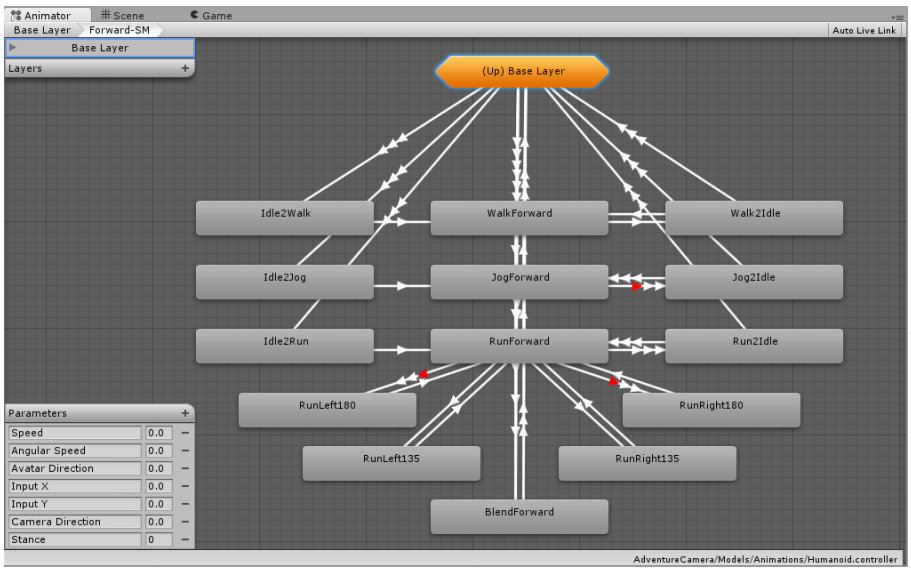
In this mode, the player's movement is slowed and, like the Melee Stance, the player is always facing forward with the camera.

Movement in this mode is much closer to a **1st person** shooter.

Working Together

The full experience is brought together when these three stances work together. As the developer, you can decide when (and if) the player will transition from one stance to another.

Animator Blending



One of the big challenges I had was determining how to organize the Animator.

At first, I had a simple 'Movement' blend tree that blended a walk, jog, and run. However, what I found was that you can't have transitions using that approach.

For example, if I wanted a 'run-to-idle' animation transition it would never fire since the blend tree would always take the 'run' down to a 'walk' before running the 'run-to-idle' animation.

In spending days playing with this, I realized the best approach was to limit movement to 3 speeds: walk, jog, and run. You see this in the center of the layer. (Note that the avatar can transition between these speeds smoothly).

Using this approach, the player is able to go directly from an idle into a run (using a nice transition animation) and then from a run into an idle (again using a nice transition animation).

You can review the included Animator tree for how animations transition between each other. All data sent to the Animator is done in the AdventureController.cs file.

Root Motion

Root motion is a great tool for helping get precise movement tied to the animations. The animation itself controls the position of the player. For example, in a walk cycle a real human doesn't have a constant speed. They speed up during a stride and slow down as their foot impacts the floor. Root motion can capture these subtleties.

However, one issue is that root motion is tied to the FixedUpdate function. When being processed, the position of the character is actually determined at the fixed time-step. That means if you want perfect camera synchronization, it also needs to be moved on the fixed time-step. However, we really want to wait to move the camera until the LastUpdate function.

So, one thing we can do is take control of the root motion. By using the OnAnimatorMove function, we can extract out the root motion. This function is called right after FixedUpdate on the fixed time-step. However, we want to use the information in the traditional Update function (which has a variable time-step).

You'll see this in the code walk-through below.

Enumerations

In the code, you'll see several files prefixed with the term 'Enum'. These files represent enumerations that are used throughout the code base.

I prefer to use integers for enumerations instead of an actual 'Enum' structure. While it is not type-safe, it simplifies conversions and serialization.

Code-Walkthrough

I'm a fanatic about well documented code. As such, reviewing the code itself should be pretty easy. The files listed below represent the complete logic for this camera system.

The purpose of this section of the document is to help give context to the code itself.

ControllerState.cs

In order to watch for trends, I track the controller state frame-to-frame. This way we can see if the player is slowing the avatar down, speeding them up, etc.

This struct contains no logic, but holds basic data such as speed, acceleration, input settings, etc.

The AdventureController holds three instances of this struct:

mState – Current data that represents a completed state.

mPrevState – Data from the last completed state (or frame)

mTempState - Data for the current frame that is being gathered and manipulated. Not until the manipulation is done does it become the 'current' state.

AdventureController.cs

The controller is what the player uses to control the avatar. Here we have logic that controls what animations to play, how to interpret user input, and how to rotate the avatar.

The properties are all pretty self-explanatory.

Trending

One exception may be the properties with 'Trend' in their names and 'mMecanimUpdateDelay'. What I learned is that I don't always want to send state data to the Animator immediately. Sometimes I want to wait to see what the trend will be and then send the data.

A good example of this is when the player lets off the game stick. The value of the stick won't go from 1 to 0 immediately. It may take several frames. In this case, the avatar would go from running to jogging to walking and then stop. What I want is for them to stop immediately.

So, I watch for a trend and update the speed data when I feel the trend is complete.

As a note, Unity does support setting Animator data over time with the SetFloat() function. However, I didn't want to do this all the time and it wasn't predictable.

Animator States

Lines 195-235 are used to store Animator state IDs. These represent the nodes and transitions in our Mecanim Animator tree. Using these, we can tell what animation we're currently in and make decisions on what to do next.

Start()

Load up the animator states we defined earlier.

FixedUpdate()

This is no longer used. Instead, root motion values are taken from the fixed time-step and used in the Update function. Movement is applied during the update.

Update()

As expected, this is the work-horse of the code.

Lines 369-399: Determine the stance (Exploration, Melee, or Targeting) that we should be in.

Line 402: Convert player input (i.e. key presses) into avatar movement (i.e. rotation).

Line 409-440: For the Targeting Stance:

> First, ensure that the camera is using our 1st person mode. Then, determine what direction we think the player is wanting to head. This could be based on the current Animator state, but also could be based on the angle we see the player wanting to move.
>
> The major thing here is that we look at the viewing direction of the player and ensure the avatar is facing forward along with the camera.

Line 442-576: For the Melee Stance:

> There's a lot of conditions here, but what we're really doing is determining what direction we think the player is wanting to head. In doing that, we determine if we need to rotate the avatar.
>
> Remember (Lines 485+), we don't want the player to be able to run towards the camera like they do in Exploration mode. So, we apply some limits to the rotation.

Line 587-615: Look for trends so we can hold onto the data to let the speed trend finish.

Line 624-625: Use the root motion we stored (see below) to determine the actual movement and rotation.

Line 629-654: Telling the Animator:

> At this point, we know the direction the player is heading and basic information like speed. However, before we send it to the Animator, we may want to tweak it.
>
> Here we'll look to see if we're starting a new trend (explained earlier). If so, we'll delay some of the data.
>
> After swapping the ControllerState data, we then send the data off to the Animator.

Line 657-661: Debugging

These are examples of using the Debug Logger to write data to the screen. In these cases, we're writing text to specific 'lines' on the screen.

ApplyMovement()

Before we handle root motion, we need to deal with gravity. Since we're applying gravity ourselves, we need to determine the gravitational velocity and then accumulate that velocity while the avatar is falling.

Here we are going to use the root motion we store in the OnAnimatorMove function to move the avatar. However, we need to 'decode' it first.

ApplyRotation()

This function looks long, but it's really pretty simple. First, we decode the root motion rotation and apply it. Then, based on the stance we determine how much to rotate the avatar.

OnAnimatorMove()

This is the function that captures the root motion. We want to put it in world coordinates so we can manipulate it if needed. Then, in the ApplyMovement function we'll decode it to be used in the time slice.

Note that if you use this function, root motion will no longer be automatically applied by the engine. You'll have to do it in ApplyMovement.

IsIn...()

The "IsIn" functions like "IsInForwardState" are used to determine if the Animator is currently playing a specific node or transition. This way, we know what the avatar is doing and can control rotations better (see line 304).

FaceCameraForward()

When in First Person mode, we let the camera drive the direction the player faces. This way we can rotate the avatar to the forward direction and it won't take control of the view.

This means after we determine how the camera is facing, we need to update the avatar's rotation. If we don't, the rendering of the avatar will be out of synch with the camera and you'll get a jittery avatar.

StickToWorldSpace()

This is a function to convert movement data to avatar data; meaning we transform the gamepad or keyboard data into things like speed, rotation, etc. This is where we fill the ControllerState structs.

AdventureRig.cs

As the AdventureController controls the avatar, the AdventureRig controls the camera and the view.

Think of the default 'Main Camera' that Unity creates as the lens. The Adventure Rig is the dolly or rig that holds the lens. We use it to control the position and rotation of the camera.

The properties are mostly all pretty self-explanatory and documented in the code.

Update()

Here we're handling the player rotating the view around the avatar. We don't actually change the camera yet, but we are grabbing data points such as yaw and pitch.

Line 268: When in 1st person mode, the camera controls the direction the avatar is going to face.

Grab the yaw and pitch. Respect the boundaries (if there are any). As we get closer to the boundaries, we'll use the "DragStrength" to dull the motion.

Line 311: Free viewing is when we allow the player to rotate the camera around the avatar. This is typically always on, but we can set some limits.

Determine the yaw. If there are limits the drag will have us slow the camera down as we approach them. This is much nicer than a hard stop.

Determine the pitch. If there are limits the drag will have us slow the camera down as we approach them. This is much nicer than a hard stop.

LateUpdate()

We typically wait for LateUpdate() to actually move the camera because we want to wait for the avatar to finish it's movements. Once that happens, we can use the avatar to help determine the camera's next position.

However, sometimes we want the rig to wait for the controller to finish its LateUpdate(). Unfortunately, there's no guarantee what order the LateUpdate() calls will be made. So we have the ability to stop the camera rig from using it's LateUpdate() and wait for it to be updated directly. This is done with the PostControllerLateUpdate() function.

PostControllerLateUpdate()

Line 377: The mAnchorPosition is the point at which we're going to (typically) rotate the camera round. This is the head or eye of the avatar.

Line 380: For modularity we move the rotation and movement functions into their own functions. This helps when inheriting from the camera.

Line 385: In the case of the first person camera, we want to control the rotation of the avatar based on the camera movement. So, the last thing we do is rotate the avatar.

ApplyMovement()

Line 400: When in first-person mode, we use the camera rotation as the heading for the avatar.

Line 417: We use a smooth 'zoom' to transition into first-person mode. The percentage is based on a timer we have to determining how quickly to zoom. That value is then turned into a curve (instead of being linear). This creates a nice slide into the zoom.

Line 434: Now that we have the zoom percentage, we can determine how much to move the camera forward and to the side. This creates the over-the-shoulder first-person view.

The rotation of the camera is determined by a point forward of the camera. In this case, we're choosing 8-units + some offset to the side.

Line 437: If we're in the third-person camera mode and currently moving the view, we want the view to help drive the avatar.

First, we calculate the relative position on the rig.

Then, we pull out of the zoom (assuming we're coming out of targeting/aiming). This is the reverse of what we did in the first-person logic above.

Lastly, we calculate the world position of the camera based on the 'anchor position' we figured out earlier in line 375. In this way, the camera is always directly behind the avatar based on the offset values.

Line 479: If we're not in first-person and not rotating the camera, we use a different approach to figuring out the position of the camera. In the previous section, we always moved it behind the avatar by the pre-set offset. This is how older third-person cameras worked.

However, in modern games the avatar actually rotates around the camera when running to the left or right. It makes moving through the environment much smoother (when using a gamepad).

So, the logic here supports this type of movement. See the detailed comments in code for how this works.

Line 515: Here's where the spring camera kicks into action. Now that we know where the camera wants to move towards, we don't simply place it. Instead, we use the physics properties of the camera rig to set its own velocity.

By doing this, we semi-disconnect the camera's velocity from the avatar's velocity.

If you have an incredibly frantic avatar, the spring approach won't totally compensate for it. However, in most circumstances it will.

Line 539: Here we check for collisions. If there is an object in between the avatar and the position of the camera, we'll hard-move the camera.

ApplyRotation()

Using information gathered earlier, we determine how much to rotate the camera. In some cases, we look at the character, but it's better to use an independent pitch value so we don't catch the character when he bumps up.

TransitionToMode()

Starts the transition from the third-person camera to the first-person and back.

HandleCollision()

Function used to see if there is an object between the avatar's current position and the camera's position. If there is, we send back a safe position before the collision.

Support

If you have any comments, questions, or issues, please don't hesitate to email me at support@ootii.com. I'll help any way I can.

Thanks!

Tim