Lecture 6
Animation:Bring It to Life

98-127: Game Creation (S20)

Written by N Carter Williams

Instructors:
Adrian Biagioli (abiagiol@andrew.cmu.edu)
Carter Williams (ncwillia@andrew.cmu.edu)
Woody McCoy (mwmccoy@andrew.cmu.edu)
Sebastian Yang (yukaiy@andrew.cmu.edu)

1 Objectives

By the end of this lesson you will be able to:

- Understand the principles of animation
- Create animations using *Key Frames* and *Curves*
- Control the properties of scripts using the animation window
- Create a simple state machine using *Mecanim*
- Use unity animations, particle systems, trails, and post processing effects to add feedback and juice to games

2 The 12 Principles of Animation

In 1981, a couple smart animator people wrote a book about animation that is generally considered to be pretty good. That book listed 12 principles of animation based off of the philosophy of Disney's animators in the 1930's. Not all of these principles are very relevant to animation in games, but it's wise nonetheless to review them all and consider how our games can utilize the ideas to feel more alive.

1. Squash and Stretch

Squash and stretch maybe the most relevant principle for game animation. The idea is that objects should stretch or squash when they move or collide. Even if the sprite represents a rigid object, like a space ship, we expect it to have a little bit of give as it zips around or crashes into something. We may also expect abstract objects to shrink or grow like buttons that jiggle when pressed.

Be careful to keep the volume of an object constant as you stretch them in any particular dimension though. In general, you can't go wrong applying a little bit of squash and stretch to anything in your game especially if the aesthetic is on the cartoony side.

2. Anticipation

Anticipation is also really important for games. Anticipation is when we foreshadow actions with a visual cue. This may mean a windup animation from a boss attack or a knee bend before a jump. Sometimes anticipation will give an action more power, but sometimes it can just make an action feel less sudden. In games, anticipation is about conserving *perceivable consequence*, which is the player's ability to understand how their actions will affect the world and how the world will affect their character. If an enemy attacks the player without warning, then frustration will ensue.

Add power to physical actions and telegraph dangerous events by visually anticipating them.

3. Staging

In games, staging is more about level design. Staging is the arrangement of characters and environment to convey themes and create a mood. You can guide the player's attention by placing objects in such a way that they can always see the next objective. If the designer is in control of the camera, they can use it to make certain objects look bigger by viewing them from a low angle. A platformer may inform the player to go right by placing the camera to the right of the character.

Think about how the world is layed out and how you can use the camera to convey information to the player.

4. Straight ahead action and pose to pose

This principle is specifically about how traditional animators drew their frames. You can either draw each frame in order, or you can draw all of the extremes then interpolate between them. We'll be talking about *Vector Animation* where the animator creates *Key Frames* and informs the computer about how it wants to interpolate between them with *Curves*.

5. Follow through and overlapping action

This principle is generally about how objects have momentum. They can't stop or start immediately. If a character starts moving, parts of them will linger for a moment. Similarly when the stop, part of them should swing forward. This is super important for video game characters, as movement is often the primary action that players can take. This principle is also lumped with the fact that cycles in your animations shouldn't share their peaks and troughs. It's more natural for different motions to begin and end at different times. Lastly, this principle also describes idle animations, which are necessary for bring your game to life. Nothing makes a game feel unpolished like a character that doesn't move when the controller is down.

Always take momentum into consideration in your animations to make them feel ”weighty”, but make sure that your controls don't feel too ”floaty” by having the character respond to input quickly even if it defies the laws of physics. And always, always add idle animations to anything that's alive (and even things that aren't). A simple way to do this is to combine this with squash and stretch to make a character breath in and out. Take a cue from Rayman and just make all your vegetation bounce up and down and nobody will care how silly that would be in real life.

Watch `https://youtu.be/rAd6MlU5yiY?t=1854` and see how the mushrooms are just playing around. It's beautiful. The mushrooms in this game just dance around without a care.

6. Slow in and Slow out

Similar to follow through, the concept here is that objects do not suddenly begin moving at full speed or stop without slowing down. In addition, you want actions to be very quick in the middle. The typical example is that a punch will feel ”punchy” if has a lot of frames coming in, very few at the peak, and then many frames coming out. This makes it feel like the punch is moving very fast at when it lands, but it has a lot of mass because it took so long to get there. We'll talk move about this when we look at *Curves*.

7. Arc

Straight lines are pretty uncommon in nature. Projectile objects generally travel in a parabola as gravity affects their trajectory downwards. Humans find arcs natural in general, so animators use them for all kinds of motions. In games, we will often have objects arranged in grids for simplicity, but curved lines and arcing motions will always feel more natural and fluid. Slow in and slow out is an example of an arc that's not in space, but velocity.

8. Secondary Action

Similar to overlapping action, secondary action is about including the extra motions that happen around the primary motion. The player shouldn't stop breathing when they start walking. Secondary action is the extra polish you can put in to make your game seem like a real place, but it often comes at the cost of doing a lot of work that isn't going to be immediately noticeable.

9. Timing

Traditional animators control the timing of actions by the amount of *Key Frames* spent in that action. They use this to make slow actions feel slow and quick ones quick. You should try to make your actions feel true to life as a baseline, but exaggerate actions that benefit from exaggeration. Timing is important for more than just a sense of physics. Timing can be used for comedic effects or to extend a feeling of suspense.

10. Exaggeration

Life, when presented accurately, will not seem as exciting in your animation as it would in reality. To account for this, animators exaggeration simple things to increase the effect of every action. The extent to which you exaggerate will depend upon the tone of the game and the importance of the action in question. You have the power to make everything bigger and more extreme, but be careful not to distract from information relevant to gameplay.

11. Solid Drawing

It doesn't quite matter how well something is animated if it isn't attractive to look at. This one is kinda hard to do if, like me, you don't have a background in visual art. What you can do is scope to your ability. If your team doesn't have a 2D artist, you can still make your game look good by using geometric shapes or simple drawings. If you can't draw hands, don't give your character hands. Plenty of my favorite games have been made with simple, good looking art.

12. Appeal

Similar to solid drawing, your characters and world should be appealing by design. A classic example of this in games is to add googly eyes to inanimate objects to make them feel cute. Appeal doesn't just mean likable, but also visually interesting or eye-catching. Super Meatboy isn't necessarily ”appealing”, but his character has ”appeal” in this sense. When designing a character or even an object think about references that you find captivating and try to understand why you like them and how you can copy that design.

3 Unity's Animation System

Adding an animation to an object in Unity is a little bit complicated because the system does two jobs. Firstly, it creates animation clip assets that store the information necessary to create the desired motion. Secondly, it controls which animation clip has control over a given object using a *State Machine* called *Mecanim*.

To get started we'll walk through the steps required to make an animation in Unity for a specific object. We'll attach an animation to a simple square sprite.

1. Click Create ≫ Animation Controller. I'm naming mine *controller*.
2. Go to your GameObject and click Add Component and select **Animator**. Don't be fooled by that **Animation** component. That's the old system and it's not supported anymore.
3. Attach your **Animation Controller** to the **Animator** by dragging the former into the **Controller** property on the **Animator**.
4. Now that our GameObject is equipped with an **Animator**, we double click on the **Animation Controller** or select Windows ≫ Animation ≫ Animator. This where your **State Graph** will go when we start adding **Animation Clips** to this Animation Controller. We'll come back to it when there's more to do.
5. To begin Animating select Window ≫ Animation ≫ Animation. Just like the **Inspector** you can lock this window by clicking the lock symbol in the top right of the pane.
6. To create an **Animation Clip**, click on the button that says **Create**. Let's call this *clip01*.
7. Now we're ready to begin animating!

4 Key Frames and Curves

Animation Clips are files that describe the values that the **Properties** (like the position or rotation of a transform) have over a period of time. From a mathematical prospective, clips are functions that take in a time and the name of property and give you back the (serialized) value of that property. The way that we can store this function without specifying the value of every property at every point in time is to only store the values of properties specific **Key Frames** and then *interpolating* the values between **Key Frames**.

There are two ways to specify which properties a clip specifies. You can either add the property manually with the Add Property button, or you can click on the red button in the top left part of the **Animation Window** and *Unity* will add all of the properties that you change while in record mode. All serialized properties on components (including Transform) of the game object with the **Animator** component and all serialized properties on children of this game object are eligible to be animated.[1] When select an object in the *Hierarchy Window*, the animator switches to the **Animator** component on that GameObject unless that GameObject doesn't have an **Animator** component and it is a child of the previously selected GameObject. This feature can be incredibly frustrating when you want to inspect different GameObjects while you are animating a specific one. To avoid this, I lock the **Animation Window** by clicking on the lock in the top right of the window.

Select your GameObject and add the **Transform Position** property to clip01 by selecting Add Property > Transform > Position. You can now see the initial and terminal **Key Frames** from zero seconds to one minute. The values at those key frames is the current position of your GameObject. You can edit this by double clicking in the **Animation Window** under the 30 second mark. Now you have a new **Key Frame** that specifies that your object will be where it currently is around the 30 second mark. You can change this position by either writing a new value directly into the table at the left or selecting the red, circular, record button that's above that table and moving the GameObject in the **Scene Window** or **inspector**. The vertical white line in the **Animation Window** represents the time at which the new value will be written. If there isn't a **Key Frame** at that time for that property, Unity will create one with the new value[2]. Be careful with the record button, because it will record every eligible property that you change while it is on. Click on it again to turn it off when you are done.

Now you can see the animation by clicking the play button to the right of the record button. You can move more slowly through your animation by scrubbing your mouse along the bar above the **Key Frames**.

You can see how the object interpolates between the initial **Key Frame**, the one you created, and the terminal **Key Frame**. Instead of just jumping between them it smoothly. Unity, like many other animation and design software, uses a *Bezier Curve* to find the middle values between **Key Frames** that feel smooth. Between any two points, there are four input parameters: The initial and terminal values, and two values in between that the curve will go toward on its journey. This is easier to understand visually, so Unity uses a visual editor to configure it. Click on the **Curves** tab in the bottom left. You may have to do some scrolling to find our change, but there should be three color-coded curves corresponding to the three dimensions of the position property. The dots on the curve represent **Key Frames**. If you click on a dot, you can two

[1]You will need to be careful about where you place the **Animator** because you need it to be high enough in the hierarchy to control all of the components that it must control, and moving an afterward is technically possible but very tedious. It's not uncommon to have a dummy game object that houses you **Animator** which is a parent of the rest of the object.

[2]This is also true for properties that haven't been added.

bars. The tips of these bars correspond to the parametes to the Bezier Curve. The right one is the first intermediate value between this **Key Frame** and the next and the left tip is the second intermediate value between this **Key Frame** and the previous one. By default this bars are force to share a slope, to keep the curve differentiable. The angle of the bar determines exactly the angle of the curve at the **Key Frame** from the side that the bar is on. Right click on the dot and select Broken to be able to split the bars up and get a really sharp curve.

Just like before you can click in the area to add new **Key Frames** and then you can drag them around and change the values. Let's remove the changes we made to position and add the *Rotation* property. With 2D images we only really use the z rotation, so focus on just the light blue curve. You can do this by clicking on the blue curve on the table at the left. Get a feel for changing this with curves. Make sure your changes are small enough because the default window zoom will make big changes seem small. One last thing to mention with rotations that will be more important with 3D is that interpolating Euler (xyz) values isn't a great idea and Unity can actually change the representation of your rotations under the hood to make them work nicer. You can change this by right clicking on the rotation row of the table and selecting interpolation > Euler Angles (Quaternion).

An other common component to edit is the *Sprite Renderer*. Last weeks animations used the *Sprite* property to change between frames of the sprite sheet. I like to use the *Animator* to have objects grow into the scene or shrink out. You can also use *Animation Events* to cue in music or sound effects.

Let's back up and talk about the **Animator** again.

5 Mecanim

Mecanim is Unity's way of telling which *Animation Clip* is controlling a property at any given time. This allows us to have a run cycle that controls the same legs that the walk cycle controls. Check out the **Animator Window** that we openned earlier. You can see that there's a new orange box labelled "clip01". That box represents a state that the controlled object can be in. While the object is in that state, the animation clip that is in the *Motion* property of that state will have control over *ALL* of the animated properties [3]. The organeness of the box indicates that this state is initial state. That means that the object will enter this state when it enters the game. Let's add another animation clip like clip01; I'll call mine clip02. To make a new clip, click on clip01 > Create New Clip in the top left of the **Animation Window**. You can use **Mecanim** to transistion from clip01 to clip02 during gameplay by adding a *transition* between them in the **Animator Window**. Right click on clip01 and select Make Transition, then click on clip02. To edit when this transition takes effect, click on the arrow between clip01 and clip02.

By default, *Has Exit Time* willbe set true. This means that the transistion will happen automatically, and clip02 will begin at a specified point during clip01's playtime. This is often not the desired behaviour and can be turned off. The important property to examine is the *Conditions* menu. This is where you will add predicates on parameters to determine when a transisition should happen. Click on Parameters > + > Trigger in the top left of the window. You can name it "trigger". Then make a float parameter and name it "speed". Now you can go to the *Conditions* menu and hit the plus icon. Then select "trigger" from the

[3]If one of your states controls a value, all other states will also control that value regardless of whether or not their own clips right to that value.

dropdown menu. This means that we will go from clip01 to clip02 when the trigger is fired. This can be from a script that makes a call to *SetTrigger("trigger")* or from you hitting it in this window. Arragne your windows so that you can see the Scene and Animator window simultaneously and hit play (then click back on the scene window to avoid the game window). Now (if you have the animated game object selected) you can see that your it is in state clip01. You can make it transition by clicking on the radio button next to the "trigger" parameter. Unity will show you the transition in real-time in the Animator window.

In script you could call *SetFloat("speed",mySpeed)* in update to let the Animator know about the speed of the character. Then you can add a parameter to the transition from walk cycle to run cycle to check if the "speed" parameter is above a threshold. But you'll want to add another transition back to the walk cycle[4]. One last note is that clips will loop by default, so you should find the clip in the project window and uncheck the *Loop* property.

There's a lot of cool stuff you can do with *Mecanim* other than animation transitions, but you can explore that later.

6 Particle Systems

You can spend days playing around with particles, but I'll just show you how to make a simple one. I want to make the effect that should play when a ball hits a paddle in Pong. Click GameObject > Effects > Particle System. Now the particles will play in front of us as long as we have the new GameObject selected. You can see in the inspector the Particle System component. Its one complicated component, but like many things in this course, we will ingore most of it. The first parameter we'll change is to turn off *Looping* and decrease *Duration* to about .4 seconds. We can change the *Start Lifetime* to .6 seconds. A lot of properties say "Start" on them. That's because we could vary these properties over time with some fancy maths. We won't do that here though. We'll decrease the *Start Size* to .4.Then I'll change the *Start Color* to FFFA70[5]. Next we'll uncheck *Play On Awake*. Next we'll change the *Emission* tab. Set *Rate over Time* to zero, and hit the plus sign under Bursts three times. Play around with the the number and timing of the bursts until you like spread of your particle. Next let's skip to *Shape*. Cone is a good shape for this system, but you should explore the others. I'll set the *Radius* to .3 and the *Angle* to 15. I might mess around with the size over life time curves to make it look neat. I'll definitely add a trail and set it's *Lifetime* to .2. Unfortunately the generated trail will be debug magenta. We can fix this by going down to *Renderer* and setting *Trail Material* to default line. I'll also set the *Material* of the particle to default line because it's a nice basic blank material for particles and lines. Now the particle looks pretty cool, but it could definitely benefit from more fiddling. Make sure to keep an eye on the clock as you mess with particles because you can just keep fiddling with them all day.

[4]The threshold should be a bit lower so that you don't toggle back and forth too quickly when the speed is near the threshold.

[5]Hearthstone's front end is made in Unity and the particles on the Shadow Priest's hero power have their simulation space set to local. This means that when you hit the hero power the particles rotate with the button instead of staying where they are in the world.

7 Trails

The trails on the particles are pretty neat, Unity game us the ability to add them to normal GameObjects. To add a trail behind an object:

1. Click Add Component > Trail Renderer
2. Set Material > Element 0 to default line.
3. Reduce the time property to around .1. Reduce the maximum width by clicking the box labeled width in the top left and typing .25.

Trails are a special case of the *Line Renderer*. We won't cover that here, but lines are super handy and I'd recommend using them. Just make sure that you extend the size of your vertex array before setting it and you pay attention to whether you are using world space or local space.

8 Post Processing[6]

To enable post processing effects open the package manager with Window > Package Manager. From there find and install the post processing stack.

There are several parts to this stack.

1. Post Processing Layers: Tells a camera to apply the post processing effects to specific layers
   (a) Add this to your camera.
2. Post Processing Profile: This contains a set of effects and their parameters.
   (a) Create this by selecting Create > Post Processing Profile
3. Post Processing Volume: This applies a profile to the world.
   (a) To get this to apply to all camera regardless of their location, select *Is Global*.
   (b) You can use these to blend smoothly between effects by placing multiple in the world with different locations and it will lerp automatically or by changing the weight of the volumes via script over time.

To add bloom, add a post processing layer to the camera and select *Everything* as the layer. Then add a post processing volume and check *Is Global*. Then hit New for the profile field. Then hit Add Effect > Unity > Bloom and then check *Intensity* and turn it up to 20.

[6]This section is on the second version of the post processing stack. This stuff is all subject to change will new material but the effects will be similar.

9 Assignment

Find the package for this week on the course website. It's a bare bones version of brick breaker. You should use the tools learned here to make bring it to life. Add several new features that make the more juicy.