CE318/CE818 High-Level Games Development

Persistence & Animations – Lecture 7

Dr Aikaterini (Katerina) Bourazeri

27^{th} November 2023

Introduction

- Persistence: Loading and Saving Data
- Finite State Machines, Animations and Animators Transitions
- Blending Animations
- Animation Scripting

Between Scenes – Player Preferences

The easiest (and **wrong!**) way to save and load data between scenes is using `PlayerPrefs`. Essentially, *PlayerPrefs* is a hash map that allows to store and retrieve values at any time, and it is not destroyed or deleted while closing a scene.

Unity stores 'PlayerPrefs' data differently based on which operating system the application runs on (please check Unity Documentation).

`void` `Save()`**:** Writes all modified preferences to disk (not recommended to call during gameplay for performance reasons). By default Unity writes preferences to disk on Application Quit.

PlayerPrefs examples

```
using UnityEngine;
using System.Collections;

public class LoadAdditive : MonoBehaviour {

    public void LoadAddOnClick(int levelIdx)
    {
        PlayerPrefs.SetInt("Health", 100);
        SceneManager.LoadScene(levelIdx, LoadSceneMode.Additive);
    }

        void OnLevelLoaded(int levelIdx)
        {
            player.health = PlayerPrefs.GetInt("Health");
        }
}
```

Why Shouldn't We Use PlayerPrefs?

However, *PlayerPrefs* is **not** a secure way to save information between scenes:

- It's plain text!
- *PlayerPrefs* should be used to save... *player preferences*! (Not important data for the proper game).

Don't Destroy On Load

Better, you can save data in an object, and use `DontDestroyOnLoad` to avoid destroying an object from scene to scene:

```
public static void DontDestroyOnLoad(Object target);
```

DontDestroyOnLoad makes the object target not be destroyed automatically when loading a new scene. When loading a new level all objects in the scene are destroyed, then the objects in the new level are loaded. In order to preserve an object during level loading call *DontDestroyOnLoad* on it. If the object is a component or game object then its entire transform hierarchy will not be destroyed either.

Keeping Objects Between Scenes

We can create the following script and attach it as a component to any object that we want to preserve:

```
public  class DontDestroy : MonoBehaviour
{
  void Awake()
    {
      DontDestroyOnLoad (gameObject);
    }
}
```

Note that `gameObject` or `transform.gameObject` is a reference to the game object that contains this component.

Singletons

For keeping data, we can have a game object with the following script attached:

```
public  class GameControl : MonoBehaviour
//Singleton. Accessible from ANY script!
{
  public static GameControl control;
  public int  health;

  void Awake()
     {
     if (control == null){
       DontDestroyOnLoad (gameObject);
       control this;
     }
    else if (control != this){
        Destroy(gameObject);
     }
   }
}
```

Each scene has a game object with this script attached. The object is persistent between scenes and can be referenced from any one of them.

Persistence: Saving to a File

How can you save data between executions of your game? A nice way to do it is through *Serialization*. *Serialization* is "the process of converting an object into a stream of bytes in order to store the **object** or transmit it to memory, a database, or a *file*".

In Unity, you indicate that an object is serializable simply by including `Serializable` before the class definition:

```
[Serializable]
class PlayerData
{
  public float health;
}
```

Saving & Loading State

Therefore, you can create a class that contains all information about the game state that you want to save to a file. For instance, you can create this class in the same *GameControl* script we saw in the previous slide. This way:

- It will be accessible from any script in the scene.
- We can add Load() and Save() functions for loading and saving from everywhere.

Persistence: Saving to a File

```
using UnityEngine;
using System.Collections;
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
 [Serializable]
class PlayerData
{
  public int health;
}

public class GameControl : MonoBehaviour
{
   public static GameControl control;
   public int health;

   void Awake();
   {
     //Awake code from singletons here
   }

   public void Save() { ... }
   public void Load() { ... }
}
```

Saving

```
public void Save()
{
    string filename = Application.persistentDataPath + "/playInfo.dat";
    BinaryFormatter bf = new BinaryFormatter();
    FileStream file = File.Open(filename, FileMode.OpenOrCreate);

    PlayerData pd = new PlayerData();
    pd.health = health;

    bf.Serialize(file, pd);
    file.Close();
}
```

Loading

```
public void Load()
{
   string filename = Application.persistentDataPath + "/playInfo.dat";
   if (File.Exists (filename))
   {
   BinaryFormatter bf = new BinaryFormatter();
   FileStream file = File.Open(filename, FileMode.Open);
   PlayerData pd = (PlayerData) bf.Deserialize(file);
   file.Close();

   pd.health = health;
   }
}
```

Revenge of the Finite State Machines

- Finite state machines turn up all over the place.
- Why would *animation* be any different?

Finite State Machines

Finite state machines (FSMs) is a mathematical model conceived as an abstract machine that can be in one of a finite number of **states**. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a **transition**.

FSMs are characterised by:

- A set of **states** linked to one another via **transitions**.
- Each state usually corresponds to an action or behaviour.
- As long as the agent is in that state, it will carry out that particular action.
- A state moves to another state when an event (either internal or external) **triggers** the transition.

Finite State Machines

FSMs can be implemented in many different ways (hence no standards). The implementation can either be flexible or hard-coded:

* **Flexible**: uses classes and interfaces for its components.
* **Hard-coded**: all logic is part of the code itself.

Main advantages and drawbacks of FSMs:

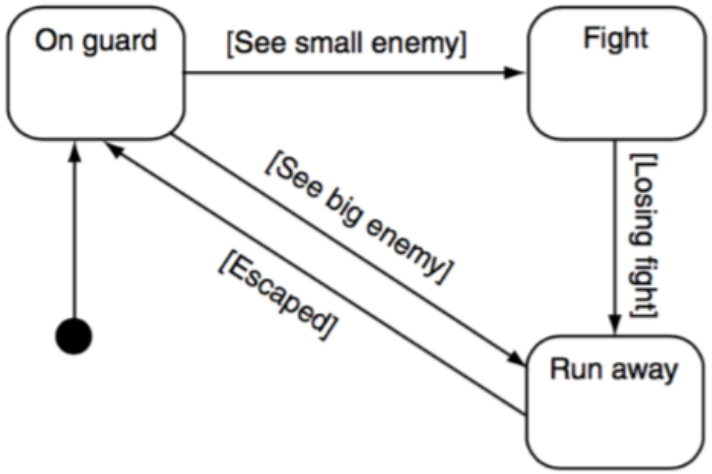
* Simple to implement.
* Easy to visualize.

But,

* Scale poorly to handle complex logic.
* Aren't designed to deal with concurrency.

Example of FSM

Example: Behaviour of a soldier protected by some cover. If there is an enemy on sight, it shoots at them during 5 seconds and goes back to the cover position. The soldier could get out of ammo and low on health, getting more ammo (that could include travelling to a location) and healing respectively.



We will use FSMs for two different aspects in this module: Unity's animation system (this lecture) and Artificial Intelligence (Lecture 8).

Unity's Animation System

Why use the Unity Animation system over a dedicated tool?

Unity's Animation System

- It is built into the tool we are already using.
- It can access other parts of the game (e.g., scripts).
- It means you don't have to learn another tool.
- (It is free(ish)?)

Mecanim

Mecanim is the Unity Animation System, that allows to setup animation on humanoid characters, transit between animations, apply animations from one character model onto another, etc.

Workflow in Mecanim can be split into three major stages:

1. **Asset preparation and import**: This is done by artists or animators, with 3^{rd} party tools, such as 3Ds Max or Maya. This step is independent of Mecanim features.
2. **Character setup**: Either humanoid (biped models) or generic (creatures, animated props, four-legged animals, etc.).
3. **Bringing characters to life**: Setting up animation clips, state machines, blend trees and scripting. **An animation clip** contains data that can be used for animated characters or simple animations. It is a simple "unit" piece of motion, such as (one specific instance of) "Idle", "Walk" or "Run".

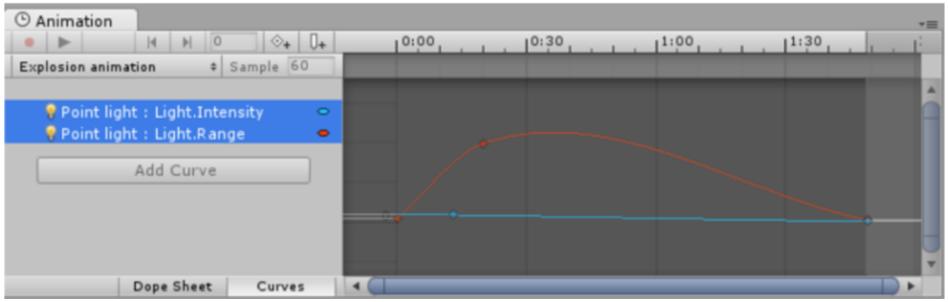
Key Terms

You can find an animation glossary defining the many terms used in Mecanim here:

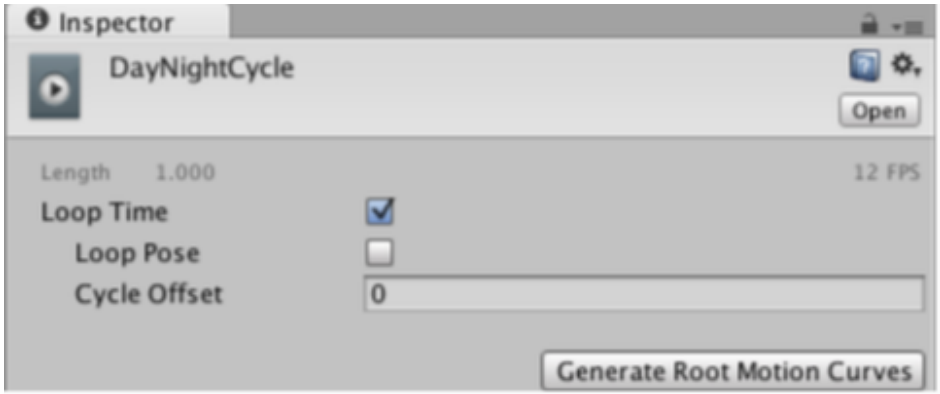
`http://docs.unity3d.com/Manual/AnimationGlossary.html`

Animation Clips

Animation Clips can be created from the Project View or from the Animation Window:

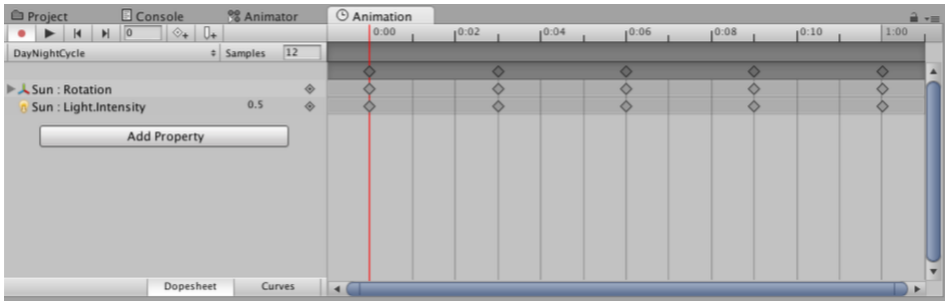


Animation Clips are always part of an Animator (that we will see later). This is how an animation clip looks like when it is created:



The Animation View

It shows the timeline and keyframes of the animation for the currently selected game object.



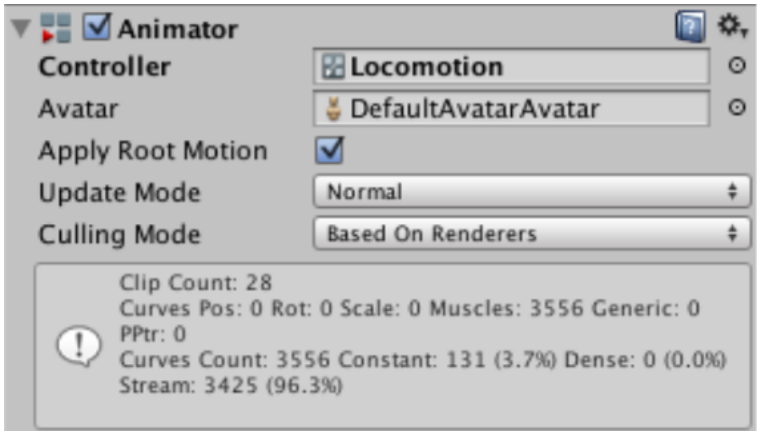
* On the left, this window shows the list of animated properties.
* The value of each property at each point in time is shown next.
* By moving the red line, you can see these values.
* If the recording button is pressed (top left corner), introducing new values will create new keyframes at the time where the red bar is placed.
* The timeline view has two modes, "Dope Sheet" and "Curves".

More information in
`http://docs.unity3d.com/Manual/AnimationEditorGuide.html`.

The Animation Setup

The Animator Component

The **Animator Component** allows to animate the properties of game objects in unity (change some of their values over time).



Controller: Reference to an Animator Controller asset.

Avatar: Asset created by Unity when a 3D humanoid is imported. It binds the model to be animated to the animator.

The Animator Component

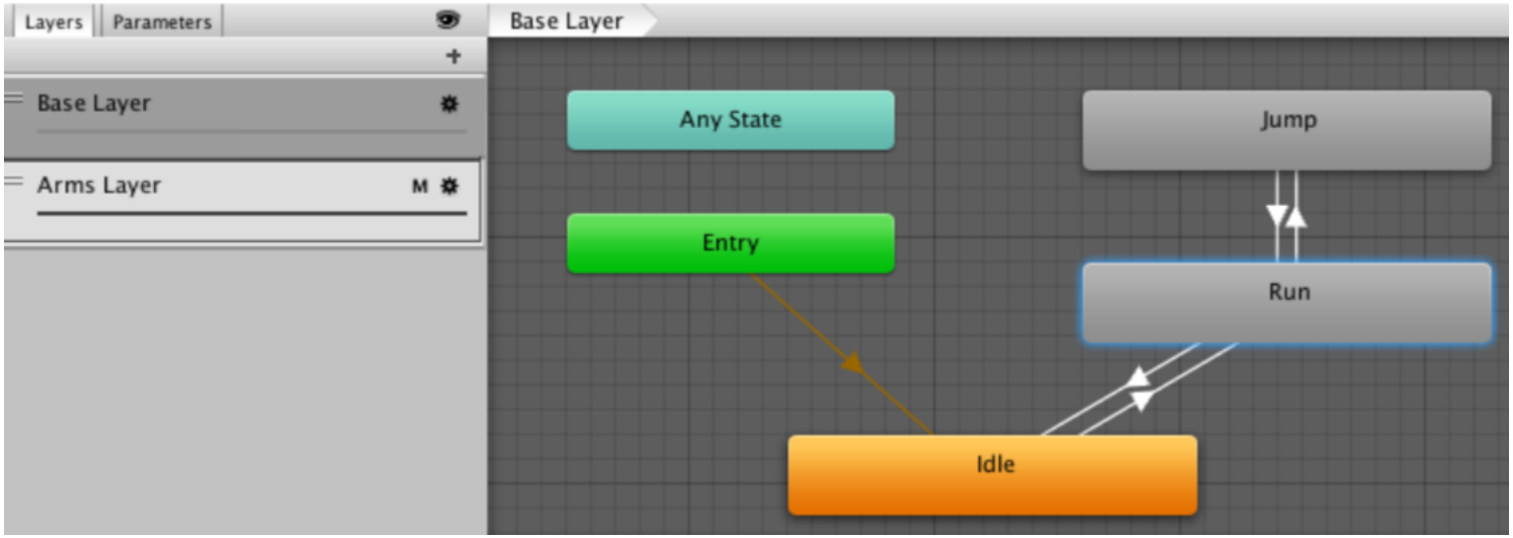
Apply Root Motion: Determines whether the animation can affect the Transform of the animated object, or if this should be controlled from script. This should be *on* if the animation moves the model, and *off* if the animation happens on the spot.

Update Mode: Determines whether the animation should be executed in time with the physics engine (*Animate Physics*), with an unscaled time (*Unscaled Time*) or normally.

Culling Mode: To indicate if the animations should be played while they are not rendered. *Always Animate* will always play the animation even if the model is hidden from the player, while *Based on Renderers* only animates the character if it is being rendered (saving performance). Note that with either mode, if *Apply Root Motion* is on, the changes in the Transform will be applied anyway.

The Animator Controller

Animator Controllers contain one or more state machines that determine which animations are currently being played and blends between animations.



The Animator Controller Window contains animation layers, animation parameters and the state machine.

Animation States

Animation States are the building blocks of the state machine. They contain the animation (or blend tree) to be played when the state machine is in that state.

The animation states are controlled by parameters that get their values from script. Changes in these parameters are used to trigger transitions to different states. When a transition happens, the character will be left in a new state whose animation sequence will then take over.

There are three special states:

* **Entry** (in green): Illustrates the entry point of the state machine.
* **Default State** (in orange): the state that the machine will be in when it is first activated.
* **Any State** (in blue): a special state, always present, used to represent transitions that must happen from any state.

Animation States

If you click on an animation state, you can see its properties in the Inspector panel.

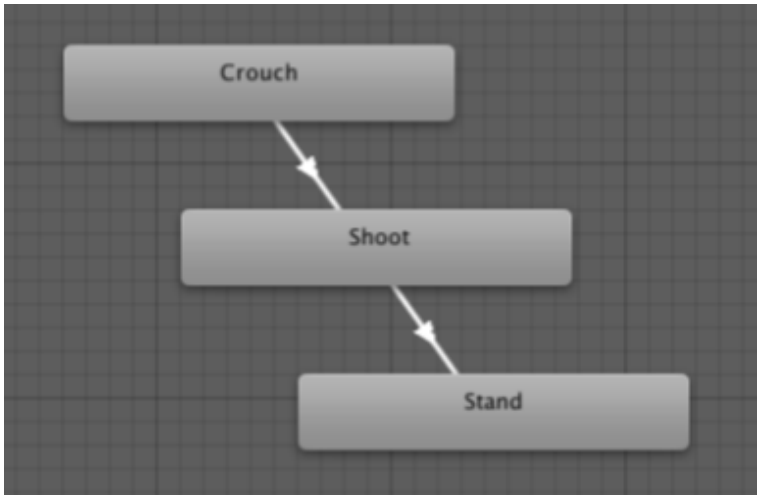
* **Name** and **Tag** can be specified for this particular state.
* **Speed**: speed at which the animation is played. 1 is normal speed.
* **Motion**: the animation clip or blend tree for the state.
* **Foot IK**: Foot Inverse Kinematics, to eliminate, if checked, any foot slipping in the animation.
* **Mirror**: Flips the animation left to right.
* Each animation state contains information about its **transitions**.

Sub-State Machines

It is common for a character to have complex actions that consist of a number of stages. Rather than handle the entire action with a single state, it makes sense to identify the separate stages and use a separate state for each.

Although this is useful for control purposes, the downside is that the state machine will become large and unwieldy as more of these complex actions are added. It is possible to collapse a group of states into a single named item in the state machine diagram. These collapsed groups of states are called **Sub-state machines**.

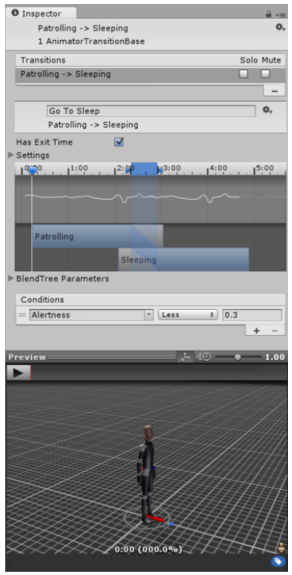
Sub-State Machines



A sub-state machine is just a way of visually collapsing a group of states in the editor, so when you make a transition to a sub-state machine, you have to choose which of its states you want to connect to. The Up state represents the "outside world", the state machine that encloses the sub-state machine in the view.

Animation Transitions

Animation Transitions define what happens when you switch from one Animation State to another. There can be only one transition active at any given time in a state machine.



For each transition, there are two check buttons:

* Solo: If checked, it disables all other animations from the originating state, except of those marked as *Solo* as well.
* Mute: Effectively disables the animation.

Animation Transitions

There are three main parts in the Animation Transition panel:

* **If Has Exit Time** is marked, the transition will occur when the animation of the previous state ends. Exit Time and other settings can be used to tune this transition.
* **Conditions** decide when the transition is triggered, allowing to base this in parameter values (those controlled via script).
* A **Preview** window shows how the transition looks like in the model.

Animation Parameters

Animation Parameters are variables that are defined within the animation system and can also be accessed and assigned values from scripts.

Parameters can be of 4 different types:

* Integer, Float, Bool
* Trigger: a boolean parameter that is reset by the controller when consumed by a transition.

Parameters in Scripts

Parameters can be assigned values from a script using functions in the:

Animator class: SetTrigger, SetFloat, SetInt and SetBool.

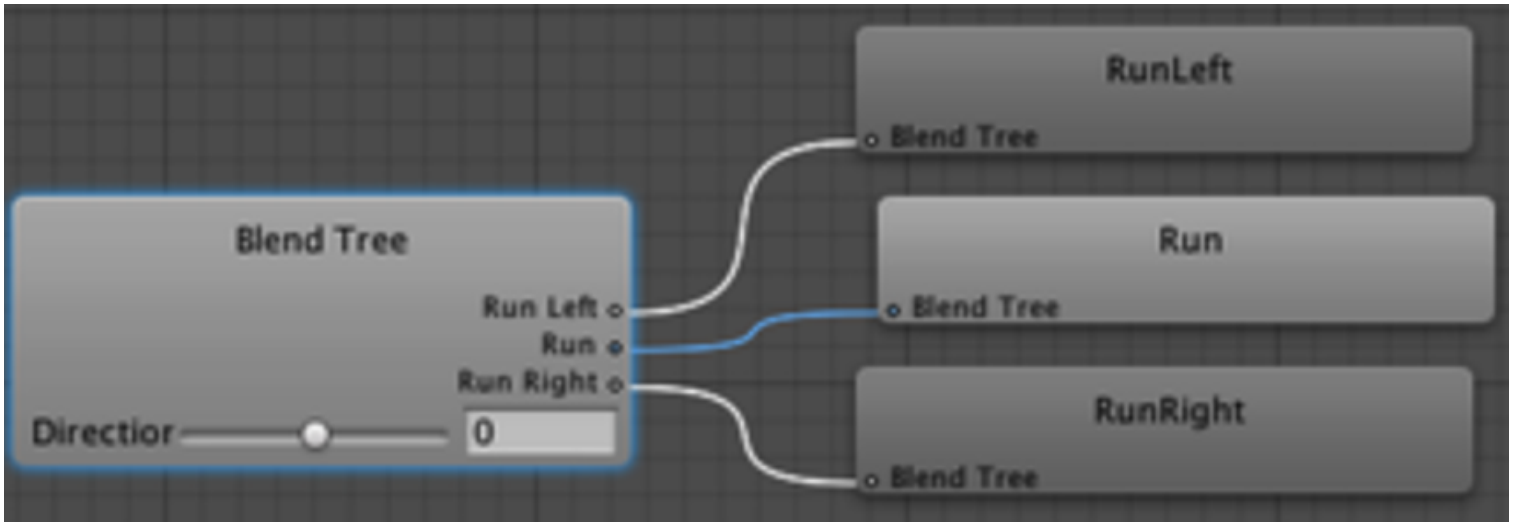
```
Animator animator = GetComponent<Animator>();
If (Input.GetButton("Fire1"))
   animator.SetBool("Jump", true);

float h = Input.GetAxis("Horizontal");
float v = Input.GetAxis("Vertical");
animator.SetFloat("Speed", h*h+v*v);
```

Blend Trees

Blend trees are used to blend between two or more similar motions. Typical examples are blending of walking and running animations according to the character's speed, or a character leaning to the left or right as he turns during a run. Note the difference between transitions and blend trees:

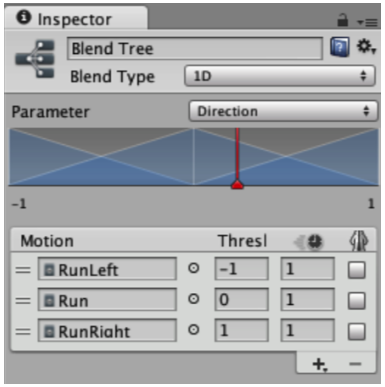
* *Transitions* are used for transitioning smoothly from one Animation State to another over a given amount of time. A transition from one motion to a completely different motion is usually fine if the transition is quick.
* *Blend Trees* are used for allowing multiple animations to be blended smoothly by incorporating parts of them all to varying degrees. The amount that each of the motions contributes to the final effect is controlled using a blending parameter.

Example

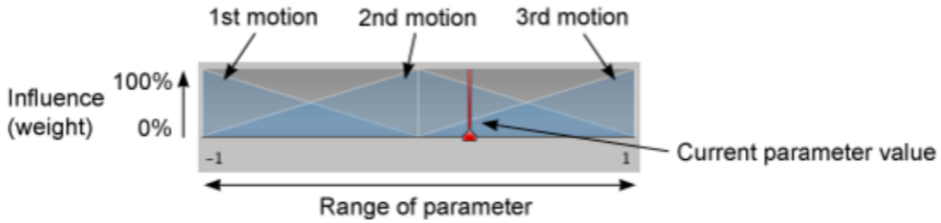

RunLeft
Blend Tree
Blend Tree
Run Left
Run
Run Right
Directior
0
Run
Blend Tree
RunRight
Blend Tree

Blend Trees – 1D

1–Dimensional Blend Tree blends between motions using **one** parameter. In this example, the blend tree blends between a Run and a RunRight / RunLeft animations using a Direction parameter.



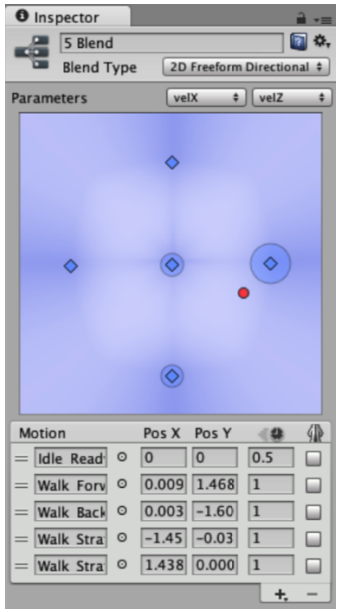
The list of motions contains the different animations to blend. Each animation/motion has an animation *clip*, a *threshold* (value of the parameter used to start/end the blending), a *speed* at which the animation is played and an option to *mirror* the animation.



Blend Trees – 2D

2–Dimensional Blend Tree blends between motions using **two** parameters.

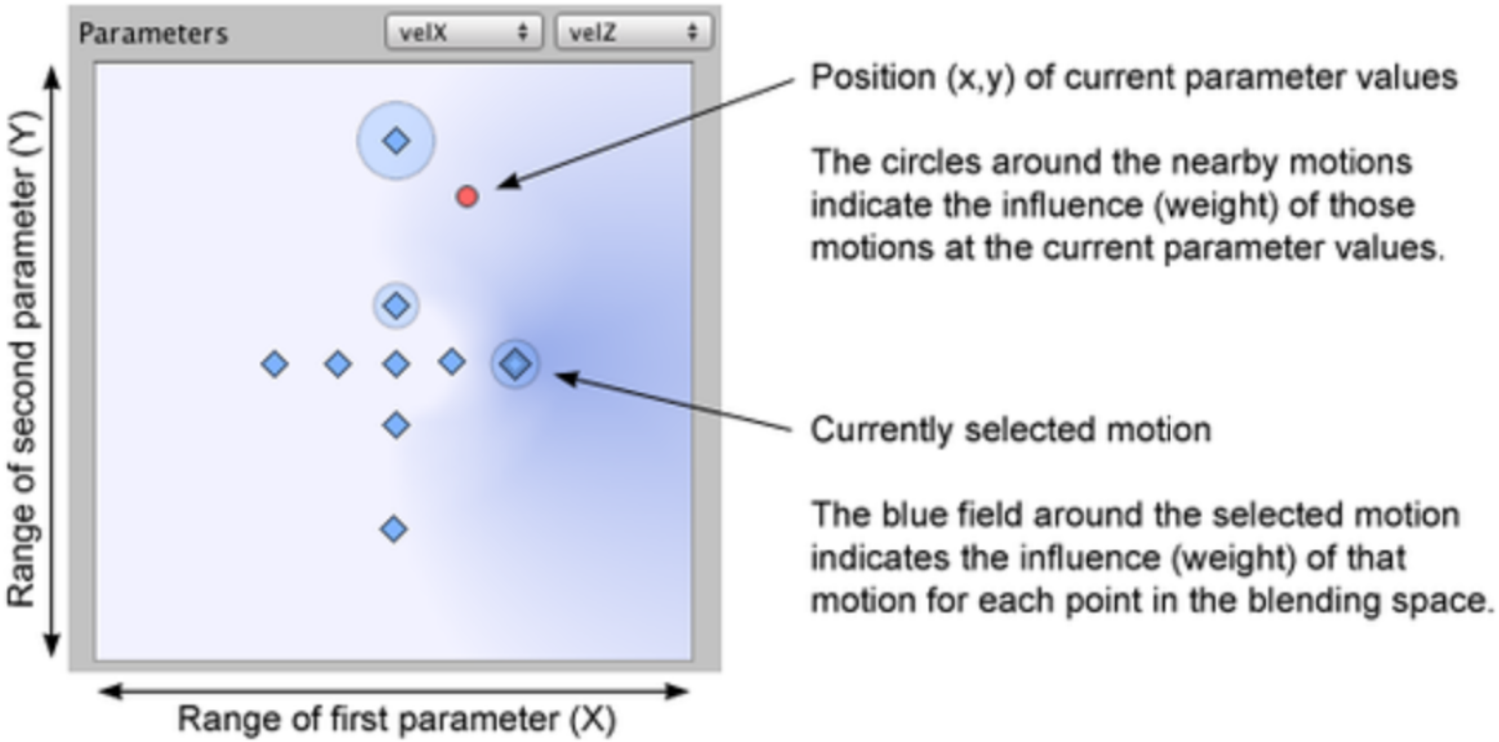
In this example, the blend tree blends between five animations: *Idle*, *Walk Forward*, *Walk Backwards*, *Walk Strafe Left* and *Walk Strafe Right*.



It uses a *VelocityX* and *VelocityZ* parameters to blend between these animations, shown in the list of motions. The parameters for these motions are the same as in the 1D case.

Blend Trees – 2D

The 2D Blending Diagram shows each motion depicted as a blue dot. The red dot indicates the values of the two parameters.



Humanoid Avatars

Avatars are definitions systems that tell the animation system how to animate the transforms of a model. You can configure the avatar by clicking on the model's mesh (in the prefab), and going to the Rig tab. If you click on *Configure Avatar*, you can configure the avatar for your humanoid. This allows to:

* **Mapping**: map between bones and transforms.
* **Muscles**: range of motion of the bones of the avatar.

Animator Controller Layers

Animator Layers are used for managing complex state machines, in order to animate specific parts of the body. For instance, you could have a base layer that is taking care of the locomotion of the avatar, and a second layer that takes care of shooting with a weapon. Each layer may be playing a different animation at the same time: both are blended depending on how the layers are configured.

By default, all controllers have a *Base Layer* where you can lay your states and transitions. You can change the name of your layer in the layer panel, located at the top left corner of the Animator Window.

You can add a new layer by pressing the + on the widget.

Animator Controller Layers

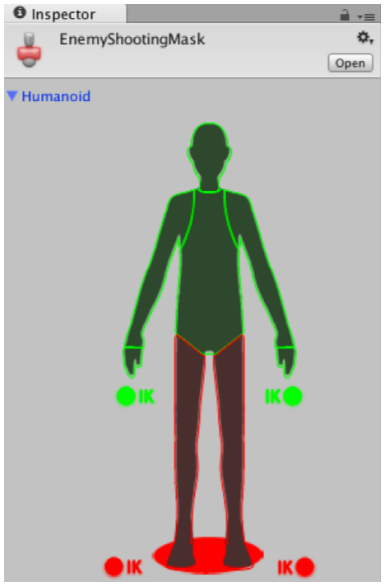
It is possible to specify several parameters for each layer:

- **Name**: name of the layer.
- **Weight**: how much the layer affects the final animation. 0 means that the layer will not affect the animation, while a value of 1 will take full priority.
- **Mask**: Avatar Mask used to isolate body parts for animation.
- **Blending**: two different types:
  - *Override*: the animation of this layer overrides the one from the lower layer.
  - *Additive*: the animation is added to the animation of the lower layer, weighted by the *weight* parameter.

Avatar/Body Masks

Specific body parts can be selectively enabled or disabled in an animation using a so-called **Body Mask**. The body parts included are: *Head*, *Left Arm*, *Right Arm*, *Left Hand*, *Right Hand*, *Left Leg*, *Right Leg* and *Root* ("shadow" under the feet).

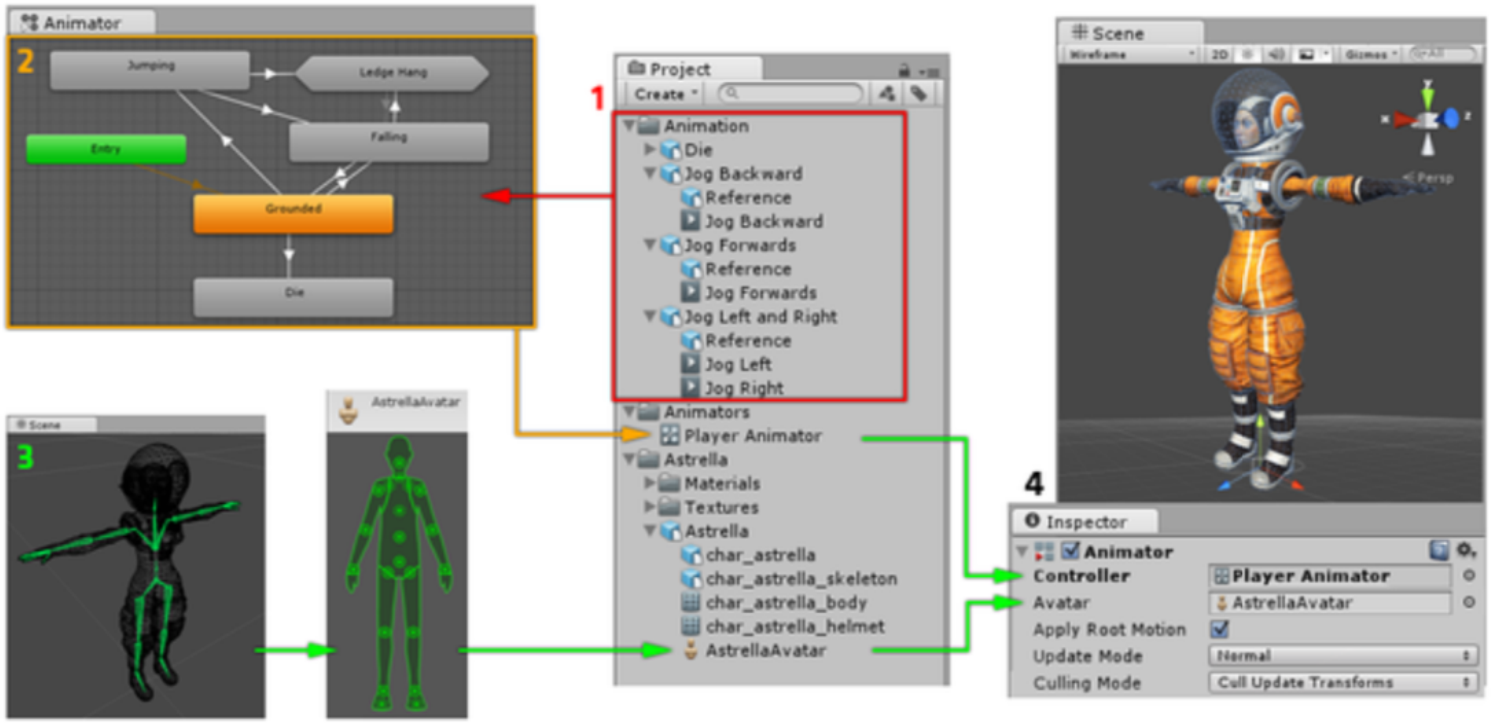
Avatar/Body Masks



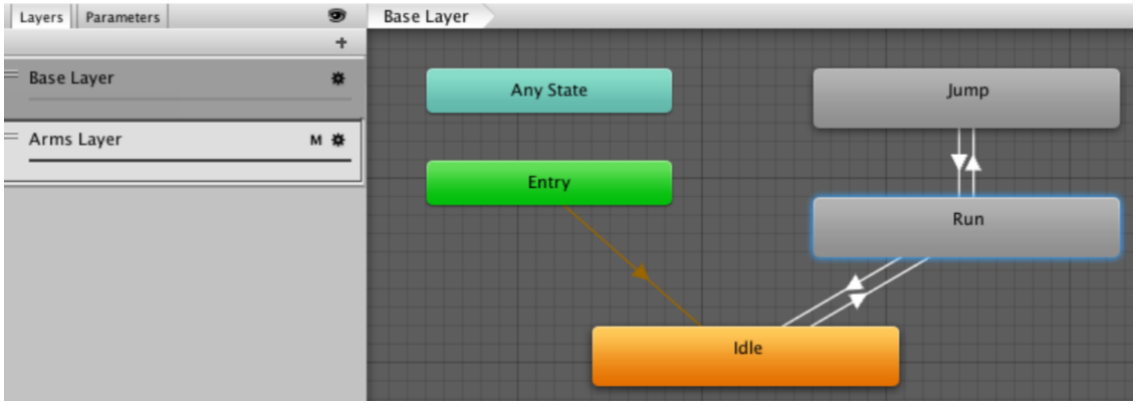
You can click the avatar part to toggle inclusion (green) or exclusion (red). For instance, if only the left arm of an overriding layer is green, the bones and transforms will be modified by the animation of this layer. The rest of the body's transform will be determined by the animation in the base layer.

You can double click in empty space surrounding the avatar to toggle all body parts.

The Animator Controller



Animator Scripting Example



The transitions are set up so:

* Idle $\longrightarrow$ Run: Float Speed > 0.1
* Run $\longrightarrow$ Idle: Float Speed < 0.1
* Run $\longrightarrow$ Jump: Trigger Jump is **true**
* Jump $\longrightarrow$ Run: Exit Time at 0.90

In this example we have three states: Idle, Run, Jump ... and four parameters: Float Speed, Float Direction, Trigger Jump and Trigger Hi.

Animator Scripting Example

```
//It's more efficient changing parameters through their hash value.
int runStateHash = Animator.StringToHash("BaseLayer.Run");
int jumpHash = Animator.StringToHash("Jump");
int speedHash = Animator.StringToHash("Speed");
int dirHash = Animator.StringToHash("Direction");
int hiHash = Animator.StringToHash("Hi");
Animator anim;

void Start ()
{ //Reference to the Animator Component.
  anim = GetComponent<Animator>();
}

void Update () {
AnimatorStateInfo stateInfo = anim.GetCurrentAnimatorStateInfo(0);

 if (stateInfo.IsName("BaseLayer.Run"))
    if (Input.GetButton("Fire1"))
        anim.SetBool(jumpHash, true);
  else
        anim.SetBool(jumpHash, false);

  if (Input.GetButtonDown("Fire2") && anim.layerCount>=2)
        anim.SetBool(hiHash, !animator.GetBool(hiHash));
```

Animator Scripting Example

```
float h = Input.GetAxis("Horizontal");
float v = Input.GetAxis("Vertical");

anim.SetFloat(speedHash, h*h+v*v);
anim.SetFloat(dirHash, h, DirectionDampTime, Time.deltaTime);
```

The Animator Class

Animator.StringToHash: Generates a parameter id from a string, used for optimised setters and getters on parameters.

GetBool, **GetFloat**, and **GetInteger**: Get values of the parameters.

SetBool, **SetFloat**, **SetInteger** and **SetTrigger**: Set values of the parameters.

ResetTrigger: Resets a trigger parameter.

SetLayerWeight: Sets the layer's current weight.

GetCurrentAnimatorStateInfo: Gets the current State information on a specified AnimatorController layer.

GetAnimatorTransitionInfo: Gets the Transition information on a specified AnimatorController layer.

The AnimatorStateInfo & AnimatorTransitionInfo Structs

AnimatorStateInfo: Information about the current or next state.

* **length**: Current duration of the state.
* **loop**: Is the state looping.
* **(int) nameHash**: Name of the State.
* **normalizedTime**: Normalised time of the State.
* **tagHash**: The Tag of the State.

AnimatorTransitionInfo: Information about the current transition

* **anyState**: Returns true if the transition is from an AnyState node.
* **fullPathHash**: The unique name of the Transition ("FullPath.CurState $\longrightarrow$ FullPath.NextState").
* **nameHash**: The simplified name of the Transition ("CurState $\longrightarrow$ NextState").
* **normalizedTime**: Normalised time of the Transition.
* **userNameHash**: The user-specified name of the Transition.

An Aside

An Aside

- Just because the tools are there, it doesn't make the animations look good.
- Speaking as a designer here ...
- Have a look at the 12 basic principles of Animation `https://en.wikipedia.org/wiki/Twelve_basic_principles_of_animation`

Quiz!

https://forms.office.com/r/qsJ3XcJyH3