Animation Blending - 動畫混合

官方參考資料：
http://unity3d.com/support/documentation/Manual/Character-Animation.html
範例下載：
http://unity3d.com/support/resources/example-projects/character-animation
在 Unity 裡面，可以使用內建的 Animation API 將兩種動畫(ex.走路+射擊)，直接用 Script 去指定動作和骨架來合成出動作，這個方法的好處在於省下 3D 模型調動作的時間：這個方法我們稱之為 Animation Blending

觀念：
Unity 的 3D 動作其實就像是在 PhotoShop 裡面的圖層一樣，圖層數字越大表示優先性越高，當兩個動作相衝突時，數字大的動作圖層會壓過數字小的動作圖層。例如，如果射擊動作大於走路動作的層級。在走路時同再進行人物射擊動作的話，你會看到你的人物做出了射擊動作，但人物的腳沒有做出走路動作，因為你的圖層被整個蓋過去了。因此要想要同時進走路動作和射擊動作的話，可以使用 Animation Blending 將源有的射擊動作與走路動作混合出新動作。簡單來說，Unity 是將射擊動作複製為一個只包含上半身骨架的新動作，空出下半身的骨架，讓下半身骨架可以去配合其他動作(ex.走路)，就可以做出同時攻擊和同時走路的動作了。
為了做到上述幾點，我們主要會用到 Animation API 中的兩個函式：
animation.AddClip( )與 animation.AddMixingTransform( )
大家可以打開專案案中 Robot Simple Mixing 資料夾下的 Mixing - Cross fade 場景來，參考 Robot 角色身上的 MixingCrossfade.js 程式
首先我們使用 animation.AddClip 來複製出一個新的動作 Clip：

```
                                    ↓用來複製的動作        ↓這個新動作的名稱
animation.AddClip(animation["shoot"].clip, "shootUpperBody");
```

接著使用 animation.AddMixingTransform 來指定這個動作使用到的骨架，語法如下

```
                                                    ↓指定使用的骨架階層
animation["AtkUpperBody_LC"].AddMixingTransform(transform.Find("Robot_Center/Robot_Pelvis/Robot_Spine1"));
```

這樣一來 Unity 就會新增一個只有包含射擊動作上半身的新動作 shootUpperBody，接下來使用 Animation.CrossFadeQueued( )播放這個新動作：

```
animation.CrossFadeQueued("shootUpperBody", 0.3, QueueMode.PlayNow);
```

為何不是使用 CrossFade( ) 呢，那是因為 CrossFadeQueued( )是將要播放的 AtkUpperBody_LC 動作重複做為下一個要接續的動作，因此在連點按鈕的時候，可以快速的接續為連續射擊動作。
注意事項：
使用 BlandingAnimation 這個方法，你必須要注意 3D 模型的骨架階層。

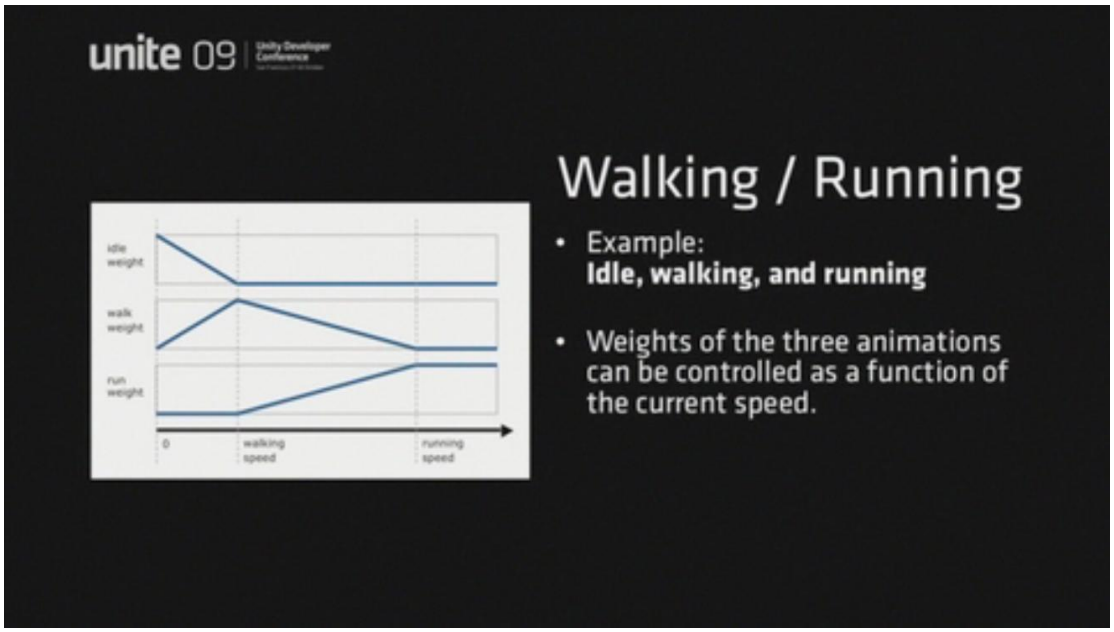
Additive Blending - 附加動畫混合

官方參考資料：
http://unity3d.com/support/documentation/Manual/Character-Animation.html
範例下載：
http://unity3d.com/support/resources/example-projects/character-animation

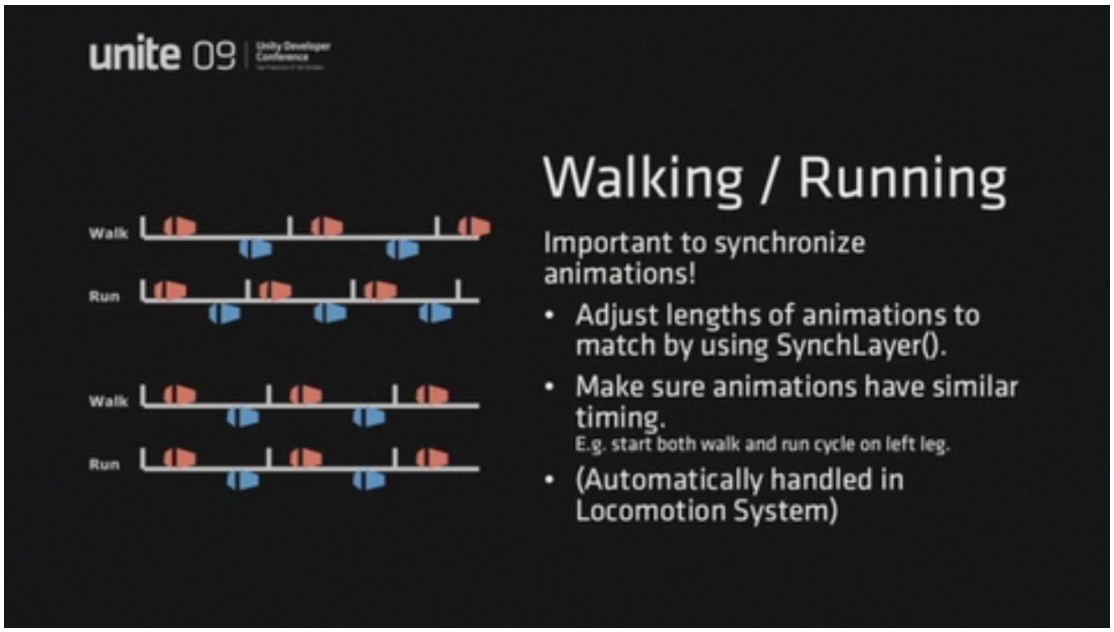
Additive Blending 跟 Animation Blending 一樣都可以用來降低動畫的數量，例如你已經有一個「走路」與一個「跑步」動畫，如果想在走路或跑步時增加傾斜的動作，你可能需要美術人員再製作「左傾走路」、「右傾走路」、「左傾跑步」與「右傾跑步」動畫，因此動畫的數量會增加到六個。為了降低美術製作負擔並降低遊戲檔案大小，可以只製作「左傾」與「右傾」的動作再後使用 Additive Blending 技術來混合出「左傾走路」、「右傾走路」、「左傾跑步」與「右傾跑步」動畫。如此一來動畫數量就會下降到只需要「左傾」、「右傾」、「走路」、「跑步」四個。

首先在製作 Additive Blending 之前，讓我們再釐清幾個動畫製作概念

1. 動作的切換是一種權重概念 (介於 0~1 之間)，可以透過程式來加以控制，例如下圖最上方線段表示 idle 動畫權重的推移，接收到走路的命令後，隨著時間推移 idle 的權重逐漸減少為 0，下方第二條線表示 walk 動畫權重的推移，隨著時間慢慢上升為 1，這時候 walk 動作就完全取代掉 idle 動作了。接著當玩家接收到跑步命令時，跑步權重也是慢慢上升取代掉走路，可以看到由於走路的動畫時間較長，故切換時間也長



2. 另一個重點是動畫長度不相等時，可以透過 SynchLayer( )命令來同步，但記得先確定兩個動畫有相似的節奏（例如下圖 walk 與 run 都是從左腳先開始），下圖可以看到 run 與 walk 動畫同步後已經具有相同的長度。



3. 當角色腳觸地時，要能夠以固定的速度來移動（由程式調整適當位移量）避免滑步現象

注意以上幾點，就可以開始使用 Additive Blending 了。

為了使用 Additive Blending，我們需要能夠完全掌握動畫的控制權，我們可以取得動畫的 AnimationState 來進行控制。AnimationState 允許製作者修改任何播放動畫的速度(Speed)、權重(weight)、時間(Time)和動畫圖層(Layers) 。這個方式在實做上需要搭配程序化的方式來驅動動畫的混合，因此在程式腳本的設計上會比在 Animation Blending 教學中單純使用 AddMixingTransform( )與 CrossFadeQueued ( )的用法要來得複雜，在這份 Additive Blending 的範例中你不會看到 CrossFade( )這個語法出現，但相對的他可以控制的部分也更多，可以讓動畫的混合變得更多樣化且自然。

一般來說可以先建立用來儲存 AnimationState 的變數，語法如下

```
private var run : AnimationState;
run = animation["run"];
```

例如上述的方法就是先宣告出一個名為"run"的 AnimationState 變數，再將我們的動畫"run"丟進去。這樣的話以後要存取 AnimationState 的各種屬性就會方便許多。接下來如果想讓 run 的速度變成 0.5 的話我只要設定 run.speed= 0.5 即可，而不需要寫成 animation[run].speed=0.5，這樣在程式的可讀性上會變得容易且清楚許多。

再來是"weight"，也就是所謂的"權重"，這個值會介於 0~1 之間。但事實上他是以百分比來算，也就是從 0%~100%。權重的高低主要會影響動畫的混合比例。例如下面的寫法就是讓跑步的動畫權重在一秒鐘後達到 1，而走路動畫在一秒鐘後達到 0，我們便會看到由跑步漸漸切換成走路動作的一個效果。

```
Function Update(){
    run.weight  -= Time.deltaTime;
    walk.weight += Time.deltaTime;
}
```

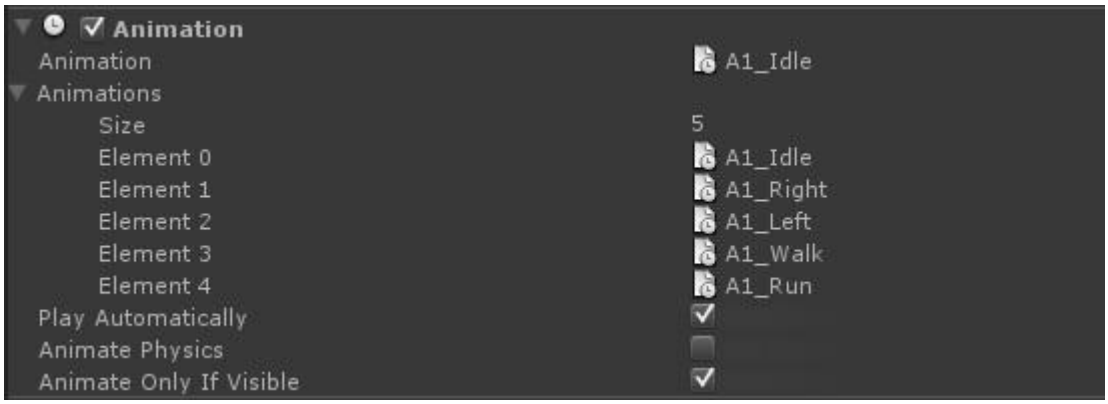
前面我們提到動畫長度不相等時，可以透過 SynchLayer( )命令來同步，這個 API 可以用來讓同一層 Layer 上的動畫進行同步，下面的語法表示讓 Layer(0)這一層的所有動作同步。

```
                        ↓此數字表示要同步的動畫圖層（Layer）
    animation.SyncLayer(0);
```

在 Animation Blending 中我們提到 Layer 是用來表示動畫所在的圖層，當兩個動畫播放衝突的時候，圖層高的動畫會壓過圖層低的動畫而播放出來，不過相同圖層內的動畫呢？那就要看權重高低來決定了。事實上你可以很簡單的把每一層都個別看成一個組別。例如下面的寫法代表 idle 和 shoot 在同一層，而 run 和 walk 在同一層。

```
    animation["idle"].layer = 1;
    animation["shoot"].layer = 1;
    animation["run"].layer = 2;
    animation["walk"].layer = 2
```

假設我們的動作表如下圖：



你可以看到我們用到了 Idle、Walk、Run 等三個常見的動作。另外表上面的左傾（Left）和右傾（Right）動作則是表示讓人物左右彎腰的動作，如下圖：

（↑Left　　↑Idle　　↑Right）

在製作彎腰動作時其實只是在 3D 動畫軟體中調整人物腰部的一根骨架的旋轉而已，腰部骨架旋轉即會帶動上半身骨架左右偏移，其餘骨架部分皆沒有調整到動作。因此 Left 與 Right 這兩個動畫的目的是用來與走路與跑步動畫混合，做出轉彎時出現的側傾動作。要達到這點需要先將 Left 與 Right 動畫的混合模式(AnimationBlendMode)設定為 Additive，且左傾與右傾動畫不需要循環播放（Loop），因此也要將兩個動畫的 WrapMode 設定為 ClampForever（播放到底後維持在最後一個影格）語法如下：

```
Left.blendMode = AnimationBlendMode.Additive;
Right.blendMode = AnimationBlendMode.Additive;
Left.wrapMode = WrapMode.ClampForever;
Right.wrapMode = WrapMode.ClampForever;
```

以上大概是用 Additive Animation 之前所需具備的專業知識，接下來大家可以打開官方專案範例中的 Soldier Blend Advanced 資料夾內的 WalkRunIdleBlend 場景來瞭解 Additive Blending 的實際作法。現將此範例使用的 ComplexWalkBlend.js 腳本程式說明如下：

```
//用來表示走路的最大速率
var walkSpeed = 2.3;
//用來表示跑步的最大速率
var runSpeed = 3.6;
//用來判斷是否要回到 idle 動作
var idleThreshold = 0.3;
//用來調整動作切換過程的平順度（數值越高動畫切換花費的時間越多）
var speedSmoothing = 8.0;
```

```
//建立用來儲存四個動畫檔的變數
private var run : AnimationState;
private var walk : AnimationState;
private var leanLeft : AnimationState;
private var leanRight : AnimationState;

//儲存玩家目前位移速度的變數，預設值為 0.0
private var currentSpeed = 0.0;
//smoothedSpeed 會慢慢往玩家目前位移的速度靠近，最後會相等
private var smoothedSpeed = 0.0;
//目前玩家傾斜度的值，預設值為 0.0
private var currentLean = 0.0;
//用來表示傾斜動畫要播到哪一個影格
private var smoothedLean = 0.0;

//接收 SampleMoveScript.js 傳過來的參數，並更新 curentSpeed 的值
function SetCurrentSpeed ( newSpeed : float)
{
	currentSpeed = newSpeed;
}
//接收 SampleMoveScript.js 傳過來的參數，並更新 currentLean
function SetCurrentLean ( newSpeed : float)
{
	currentLean = newSpeed;
}

function Start ()
{
	//確認沒有任何動作被播放
	animation.Stop();
	animation.wrapMode = WrapMode.Loop;

	//將各個動作丟到該丟的變數裡面
	walk = animation["walk"];
	run = animation["runSlow"];
	leanLeft = animation["leanLeft"];
	leanRight = animation["leanRight"];
```

```
        //如果有傾斜動作存在，就將傾斜動作設定為 Additive Animation
        // blendMode：設定混合的模式，這邊要用 Additive，也就是以加入的模式來混合。
        // wrapMode：設定動作播放狀態，這邊使用 ClampForever，代表動作會播到最後一格
        //然後一直播最後一格的動作。
        // layer：因為這傾斜動畫播放的優先權最高，所把他們兩個動作的 layer 放在最上層。
        // weight：先設定播放權重為最大。
        // enabled：設定動畫可被播放。

        if (leanLeft)
        {
            leanLeft.blendMode = AnimationBlendMode.Additive;
            leanRight.blendMode = AnimationBlendMode.Additive;
            leanLeft.wrapMode = WrapMode.ClampForever;
            leanRight.wrapMode = WrapMode.ClampForever;
            leanLeft.layer = 100;
            leanRight.layer = 100;
            leanLeft.weight = 1;
            leanRight.weight = 1;
            leanLeft.enabled = true;
            leanRight.enabled = true;
        }

        //設定 walk 與 run 動畫可被播放，並且進行動作同步。
        walk.enabled = true;
        run.enabled = true;
        animation.SyncLayer(0);

        //設定 idle 動作圖層為最低，簡單來說就是沒有播放其他任何動作時才會播放 idle 動作
        animation["idle"].layer = -1;
        animation["idle"].enabled = true;
        animation["idle"].weight = 1;
}

function Update () {
        // smoothedSpeed 會慢慢往 currentSpeed(玩家目前位移的速度)靠近，最後會相等
        smoothedSpeed = Mathf.Lerp(smoothedSpeed, currentSpeed, Time.deltaTime *
speedSmoothing);
```

```
        //smoothedSpeed 沒有超過 2.4(walkSpeed)之前，runWeight 的值都是 0，等於
        //3.6(runSpeed)時 runWeight 值為 1，依照 Sample Move Script 中 speed 的預設值可
        //知 smoothedSpeed 最大為 3 或 4.5
        var runWeight = Mathf.InverseLerp(walkSpeed, runSpeed,
Mathf.Abs(smoothedSpeed));

        // runWeight 在 smoothedSpeed 沒有超過 2.4 時都為 0，也就是說在位移速度 2.4 以下
        //（走路狀態）時不需要附加傾斜動作
        var targetLean = currentLean * runWeight;
        smoothedLean = Mathf.Lerp(smoothedLean, targetLean, Time.deltaTime *
speedSmoothing);

        //這邊在設定 run 和 walk 動畫播放的速率，會配合玩家位移速度，位移速度越大播放越快。
        run.speed = smoothedSpeed / runSpeed;
        walk.speed = smoothedSpeed / walkSpeed;

        //當玩家位移速度低於 idleThreshold 時，漸漸將 run 或 walk 的權重釋放出來（這個時候
        //會開始播放 idle）
        if (Mathf.Abs(smoothedSpeed) < idleThreshold)
        {
                run.weight    -= Time.deltaTime * 5.0;
                walk.weight -= Time.deltaTime * 5.0;
        }
        else
        {

                var totalWeight = run.weight + walk.weight;
                totalWeight += Time.deltaTime * 12;
                //totalWeight 的值會介於 0~1 之間
                totalWeight = Mathf.Clamp01(totalWeight);
                //用 runWeight 來設定 run 與 walk 的權重，乘上 totalWeight 可以讓數值變化更細
                //膩些，可降低 run 與 walk 動畫切換時跳動的感覺
                run.weight = runWeight * totalWeight;
                walk.weight = (1.0 - runWeight) * totalWeight;
        }

        //下面代表有側傾動作存在的話才繼續進行，不然就不需要效能來執行
        if (leanLeft)
        {
                //如果 smoothedLean 大於 0，表示要往右傾
```

```
            if (smoothedLean > 0)
            {
                //設定左傾動畫停在動畫第 0 秒，也就是不播的意思
                leanLeft.normalizedTime = 0;
                //設定右傾動畫播放到 smoothedLean 數值表示的時間點（nomalizedTime
                //是動畫的正規化時間長度，介於 0~1 之間，0 表示動畫一開始，1 表示動畫
                //的結尾），由於 smoothedLean 的值也是位於 0~1 之間，故可用來表示動畫
                //播放的時間點，因此這邊會開始播放右傾的動畫
                leanRight.normalizedTime = smoothedLean;
            }
            else
            {
                //設定左傾動畫播放的時間點（smoothedLean 在此時是負值，故要乘-1 轉為
                //正值）
                leanLeft.normalizedTime = -smoothedLean;
                //設定右傾動畫停在一開始的時間
                leanRight.normalizedTime = 0;
            }
        }
}
```

由上可知，Additive Blending 並非如同之前 Animation Blending 範例中透過 CrossFade 等自動化的方式來運作動畫，反而要透過程式撰寫來自行設定權重與播放時間點，可以說具有完全的主控權，搭配程序化的設計模式可以設計出相當細膩的動作系統，且可大幅節省動畫製作數量。