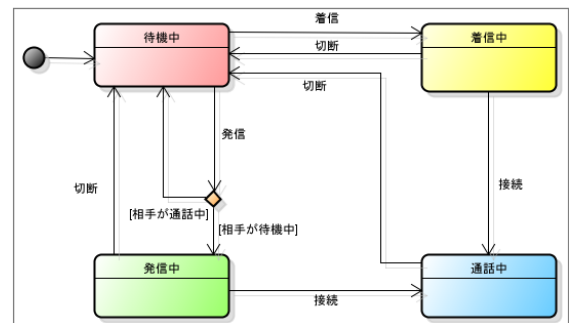


StateMachine (ステートマシン)

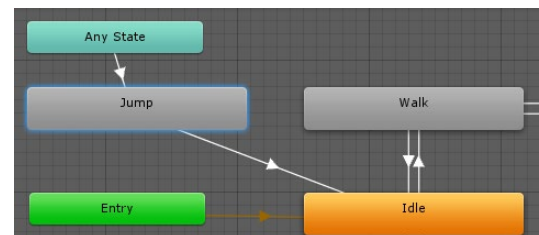
ステートマシン図の例

Unity を代表としたソフト群「ゲームエンジン」のほとんどにはステートマシンという概念があります。

情報処理用語「状態遷移機構(ステートマシン)」が語源なので考え方はよく似ているのですが、使用する現状としてはかなり楽しく取り組めるものとなっています。注: 情報処理用語「流れ図(フローチャート)」とは全く異なります。



我々がゲームエンジンで扱うステートマシンは、操作するキャラクターの動き(状態)を個々のノード(拠点)に据え、それらを状態遷移線で接続し、プレイヤーからの入力条件によって動きを遷移させていく考え方となります。



Unity には以前からキャラクターのアニメーション機能がありましたが、バージョン4になった時に **Mecanim(メカニム)** という呼称でステートマシンを導入し、ゲーム制作作業が飛躍的に効率化されました。

それまでは3DCG技術者に頼っていたキャラクターの動作設定を、ある程度までプログラマーやプランナー側で扱えるようになったのです。フリーのMMDモデルデータや、動かない3Dキャラクターモデルでも、ボーン(骨)とウェイトスキニングの設定さえあれば、生き生きとゲーム内で動かせるゲーム作りが可能になり、その動きもプログラム側から細かくチューニング出来るようになったのです。画期的！

同時期に Unity は普及促進の為に「ユニティちゃん」というバーチャルアイドルを登場させ、ライセンスさえ守れば無料で利用できる扱いになっています。様々な場面で見かけることがあるかも知れません。

今回はゲームなどのコンテンツ制作ではありませんが、Unity でのステートマシン(メカニム)の扱い方を理解する為の演習となります。

Unity 公式データだけあって都合な部分も多く、説明にはユニティちゃんを使いますが、古いバージョンの Unity にも対応するべく、少し古いプログラム記述や設定が警告となって表示されます。(スルーして大丈夫ですヨ。)

使用許可ライセンスに従い、組み立てプリントや制作コンテンツの中でロゴを明示しています。



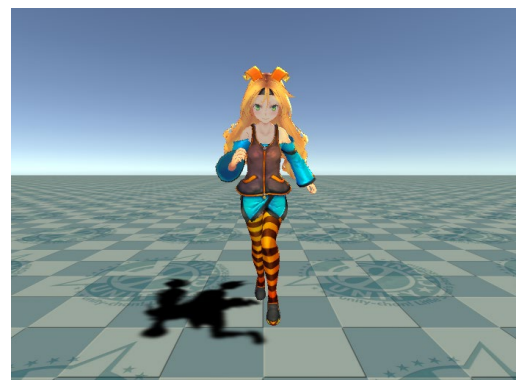
また、Unity では、以下の用語が全て同じものを指すこととなります。

ステートマシン : 業界標準の用語

メカニム : Unity が普及促進を意図した表現

アニメーター・コントローラー : コンポーネントを作成する時の呼び名

アニメーター : コンポーネント名、クラス名、設定パネルとして呼ぶ時

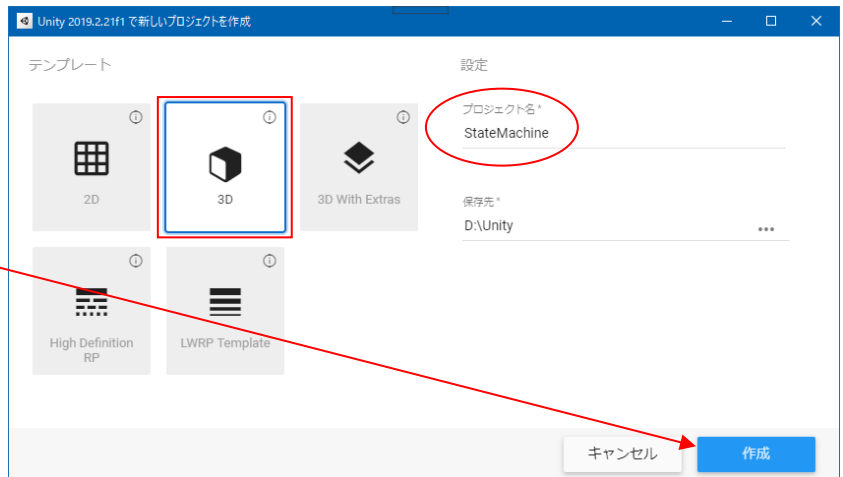


シーンの構築

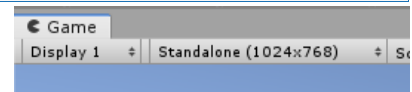
【STEP1】 プロジェクトの作成

- Unity を起動して、新しいプロジェクト **StateMachine** を 3D モードで準備します。

ボタン作成を押下します。



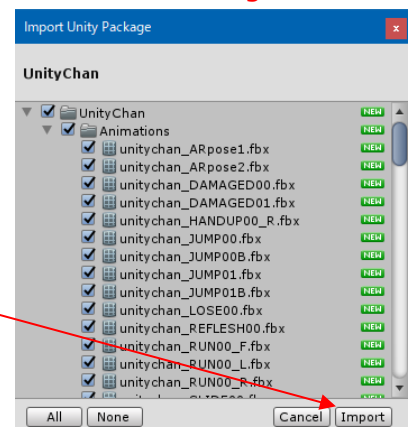
- パネル Game の画面サイズを **Standalone(1024x768)** に設定します。



【STEP2】 素材の読み込み

- 配布された素材を読み込みます。メニューAssets から Import Package > **Custom Package** と進み、今回のフォルダ内にある **UnityChan.unitypackage** を指定します。

内容物が表示されたら、**Import** を押下します。

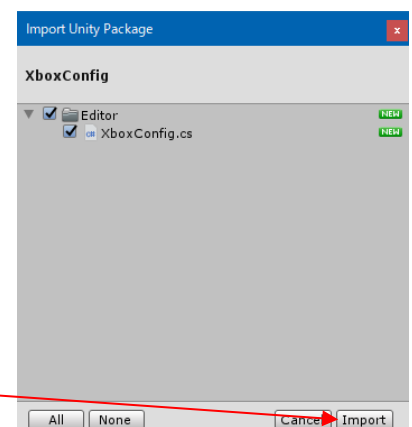


- 同様に、メニューAssets から Import Package > **Custom Package** と進み、**XboxConfig.unitypackage** をインポートします。

これは、前回の授業で、さんざん苦勞して構成した手続きを、プログラムで一気に行う Unity の改造指示です。エディタ・スクリプトと言い、アセットストアでは、ゲーム本体やゲーム素材を差し置いて、ダウンロード数や儲かる額が抜き出ている状況です。

内容物が表示されたら、**Import** を押下します。

改造が済んでいるか？を、インプットマネージャーで確認します。

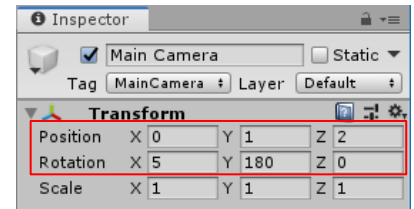


- 素材ではありませんが、ここで Xbox ゲームコントローラー for Windows を USB 接続します。必ず接続完了メッセージが出ることを確認します。

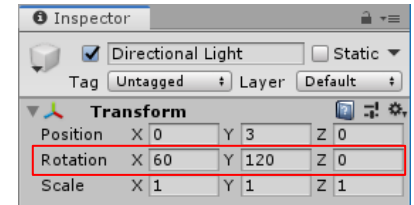
Joystick connected ("Controller (XBOX One For Windows)").

【STEP3】 シーンの設定

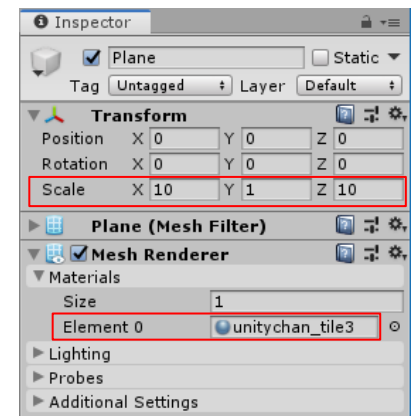
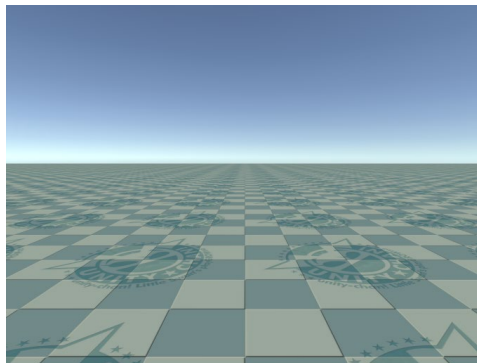
- ヒエラルキー欄の **MainCamera**(メインカメラ)を選択し、インスペクタでパラメータを設定します。



- ヒエラルキー欄の **Directional Light**(ディレクショナルライト)を選択し、インスペクタでパラメータを設定します。



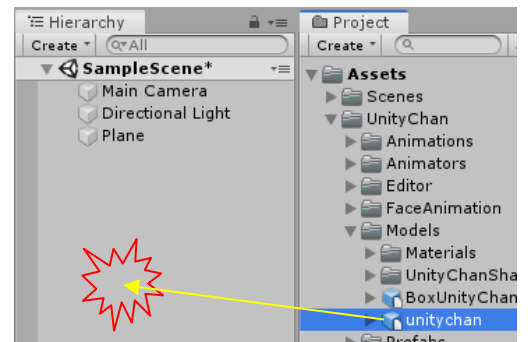
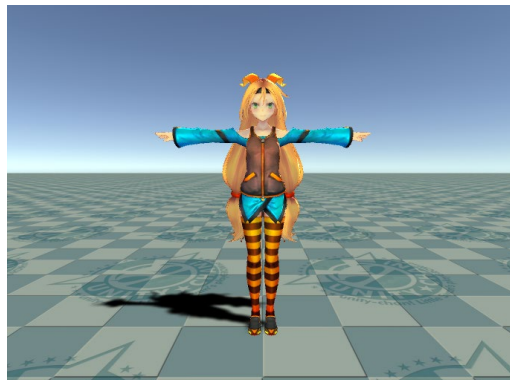
- ヒエラルキー欄の Create から 3D Object > **Plane**を選択し、インスペクタでパラメータを設定します。



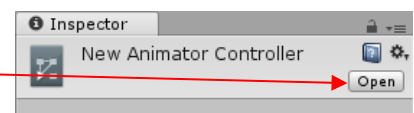
ステートマシンの設定

【STEP4】 アニメーターコントローラーの作成

- プロジェクト欄のフォルダ UnityChan > Models > **unitychan** をヒエラルキー欄にドラッグ & ドロップします。

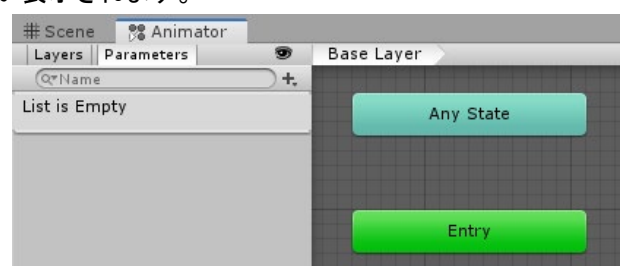


- プロジェクト欄の Create から **Animator Controller**(アニメーターコントローラー)を選択します。名称は New Animator Controller のままでいいでしょう。
- インスペクタに表示される **Open** ボタンを押下します。

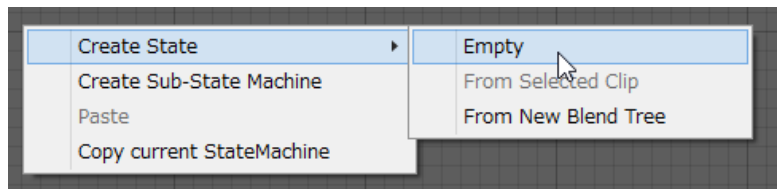


アニメーターコントローラーを編集するパネル **Animator** が表示されます。

- マウスの中央ボタンで画面全体をドラッグする練習
- マウスのスクロールホイールで拡大縮小をする練習

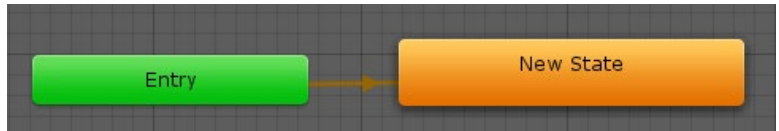


- 背景のグレーの方眼紙を右クリックし、Create State > **Empty** と進み、空っぽの「状態」を作ります。

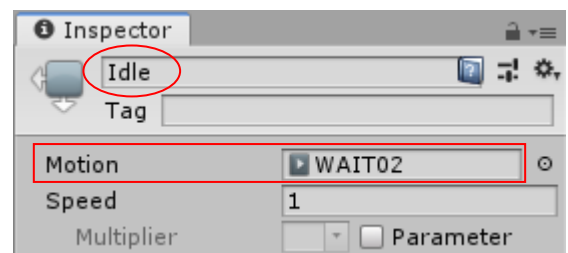


新しい **状態(New State)** が作成されています。

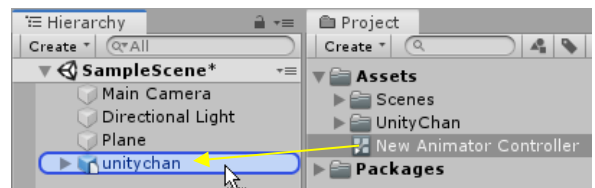
自動的に Entry(開始)から動きの遷移(トランジション: Transition)が伸びて接続されます。開始したら、この状態を再生することになります。



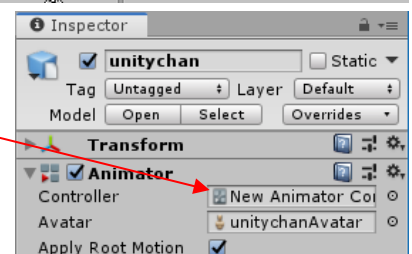
- この名称を変更します。
このオレンジ色の箱を選択しているまま、インスペクタで名称を **Idle(アイドル)** とします。
- 動きを Motion(モーション)で指定します。右横の○ボタンを押下し、選択リストの中から **WAIT02** をダブルクリックして指定します。



- 一度、ここで動作確認をします。
プロジェクト欄の **New Animator Controller** をヒエラルキー欄の **unitychan** にドラッグ & ドロップします。



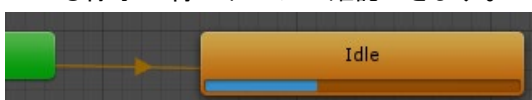
ヒエラルキー欄の **unitychan** を選択すると、インスペクタで **New Animator Controller** を所有していることが見えます。



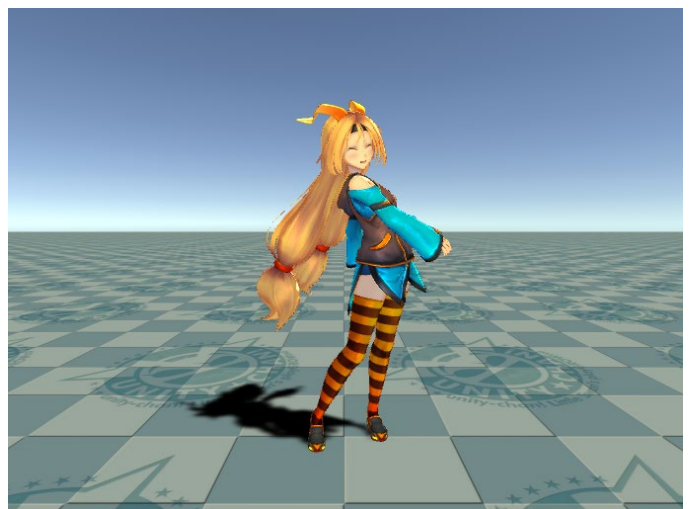
- プレイボタンを押下します。
ユニティちゃんが大げさな呼吸待機を行うことを確認します。



動作中にヒエラルキー欄の **unitychan** を選択すると、Animatorウィンドウで現在のモーションが再生されている様子が青いゲージで確認できます。



約 7.5 秒程度のモーションがループ再生されていることが判ります。

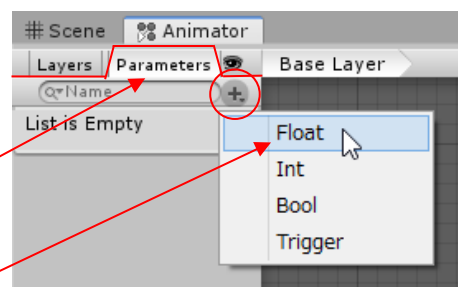


【STEP5】 歩くモーション

- Animator ウィンドウには2つのタブがあります。

- Layers(レイヤ)
- Parameters(パラメータ)

必ず！パラメータのタブであることを確認します。

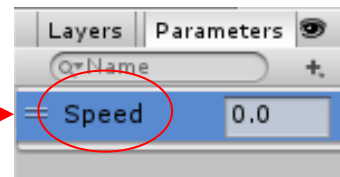


- プラス(+)を押下して、Float(フロート)型(浮動小数型)のパラメータを追加します。

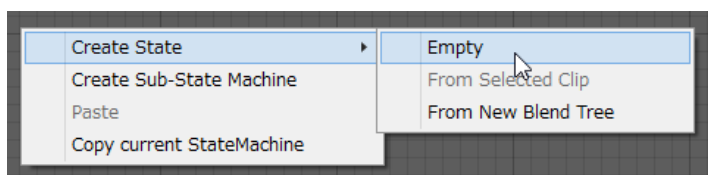
名称は Speed(スピード)とします。

後ほど、プログラムとの整合性が必要になるので、スペルミスが許されません。

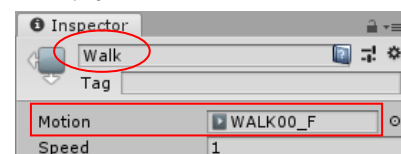
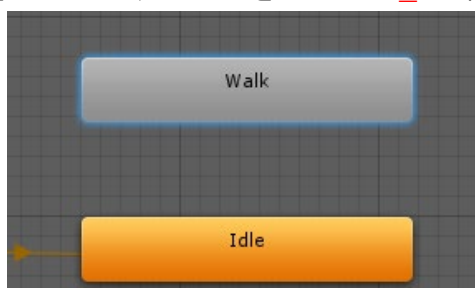
また、1文字目は大文字で始めることです。



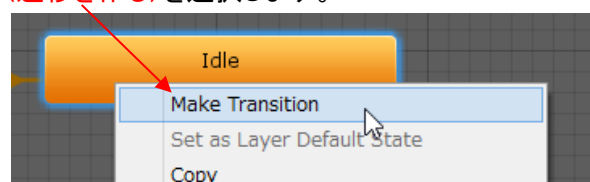
- Idle と同様にして新しい「状態」を作成します。
開始(Entry)直後に再生されるメインのモーションのみがオレンジ色となり、この後に追加する他のモーションは全てグレーで表示されます。



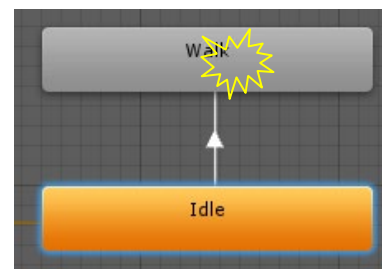
- インスペクタで状態(箱)の名称を Walk とし、Motion を WALK00_F に設定します。



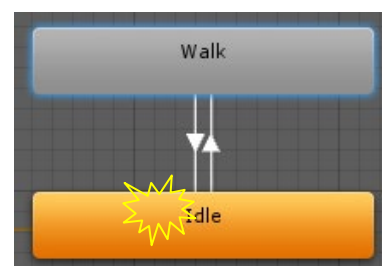
- 状態 Idle(オレンジの箱)を右クリックし、Make Transition(遷移を作る)を選択します。



- マウスに白い矢印がくっついてくるので、状態 Walk をクリックして接続します。



- 同様にして、Walk から Idle への逆向きの接続も行います。

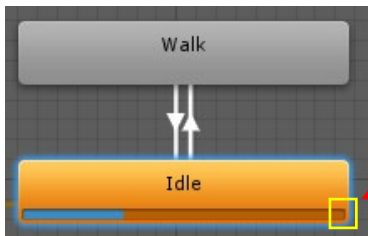


- 白い矢印 **Idle -> Walk** をクリックします。
青くなります。

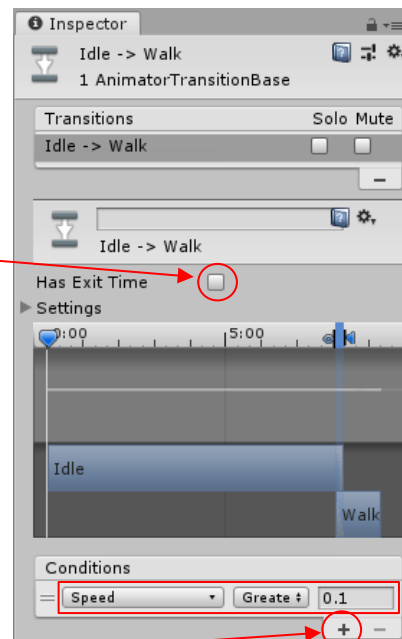


- チェックボックス **HasExitTime** をオフにします。

アニメ終了時間(ExitTime)を持っていたのでは、アイドル(呼吸待機)の動作終了を待ってから次のモーションへ行くので、とてもゲームになりません。

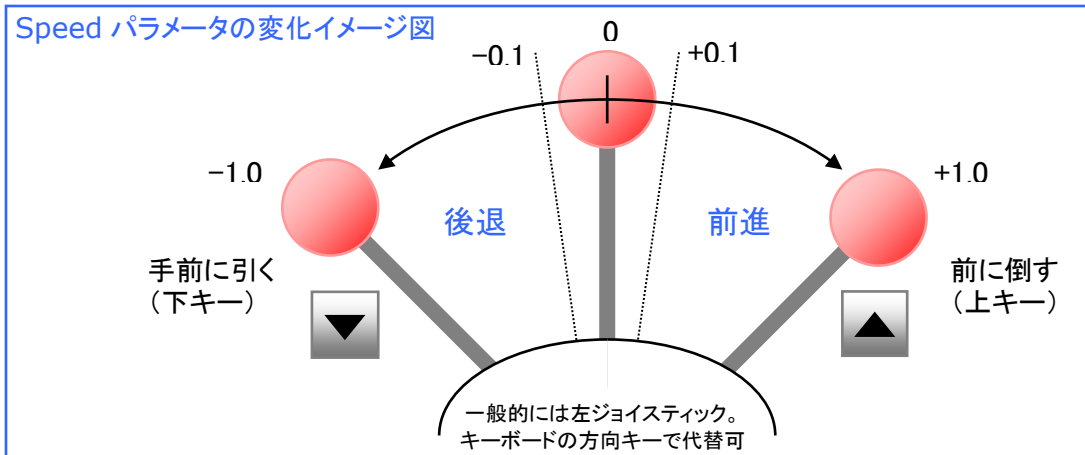


ここに ExitTime がある。待たない。

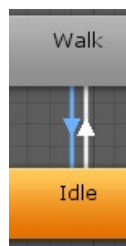


- Conditions(コンディション:条件)のプラスボタン(+)を押下します。

Speed が 0.1 を超えたら(Greater)の条件を設定します。



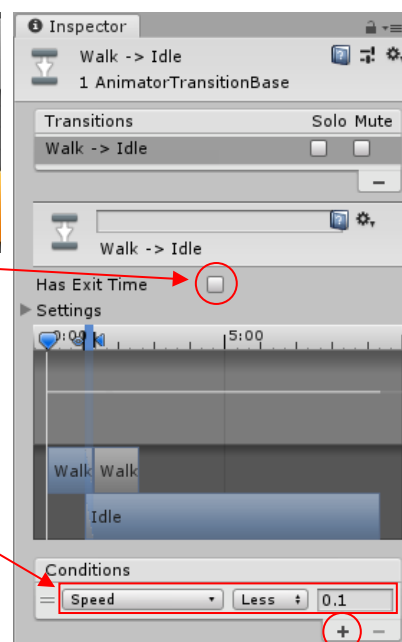
- 逆向きの白い矢印 **Walk -> Idle** をクリックします。



- チェックボックス **HasExitTime** をオフにします。

- Conditions(コンディション:条件)のプラスボタン(+)を押下します。

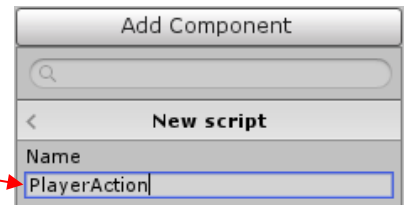
Speed が 0.1 を下回ったら(Less)の条件を設定します。



設定は出来たのですが、入力機器(キーボード)からの情報を(パラメータ Speed を介して)このステートマシンに流し込むプログラムが準備できていないので作成します。

【STEP6】 入力機器の情報を取得する

- ヒエラルキー欄のunitychanを選択し、インスペクタのAdd Component からNew scriptを選択します。
スクリプトの名称を **PlayerAction** と命名します。



- スクリプト **PlayerAction** を次のように編集します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

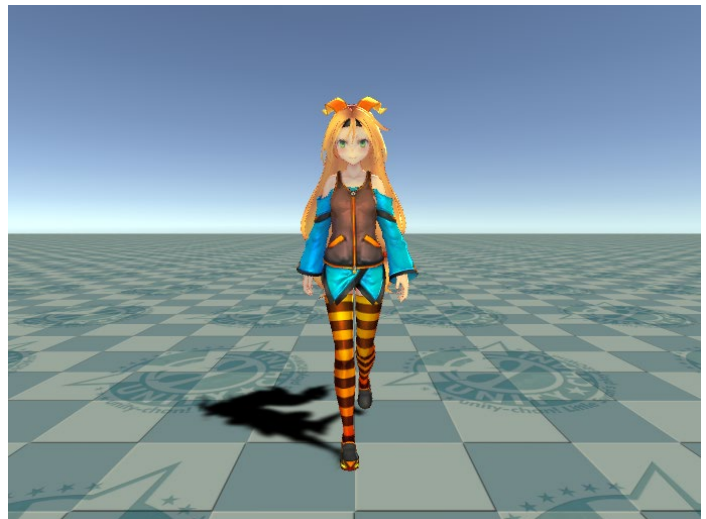
public class PlayerAction : MonoBehaviour {

    Animator myAnim; //自身のアニメーター

    void Start () {
        //自身のアニメーターを取得
        myAnim = GetComponent<Animator>();
    }

    void Update () {
        float zengo;
        zengo = Input.GetAxis("Vertical");
        myAnim.SetFloat("Speed",zengo);
    }
}
```

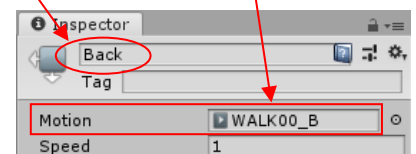
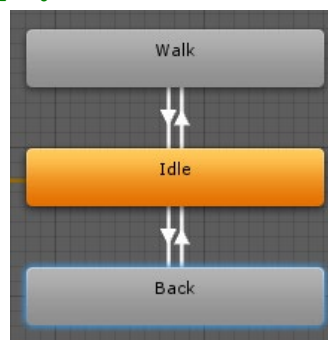
- プレイボタンを押下します。
左ジョイスティックを奥へ倒している間、ユニティちゃんが歩くモーションになることを確認します。



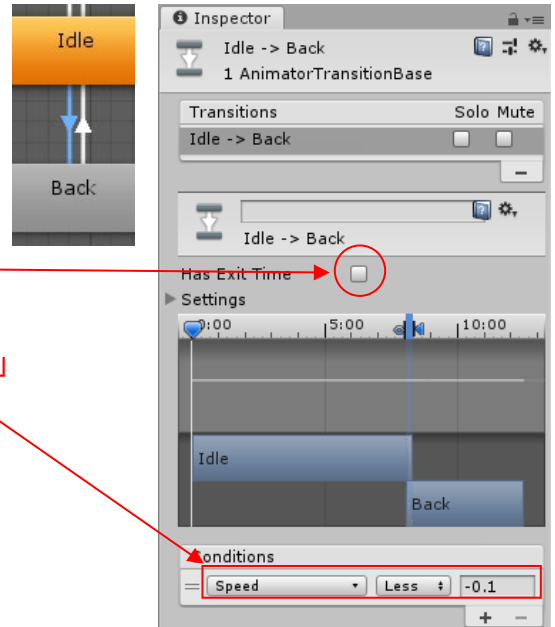
【STEP7】 後ろへ歩かせる

- 【命題】 アニメーターコントローラーに新しい状態(箱)を作って名称を **Back** とし、Motion を **WALK00_B** に指定します。
また、白い矢印 2 本で図のように接続して下さい。

左ジョイスティックを手前に引いて後進させるので、Idle の箱の下に作った方が直感的でしょう。



- 白い矢印 **Idle -> Back** をクリックします。

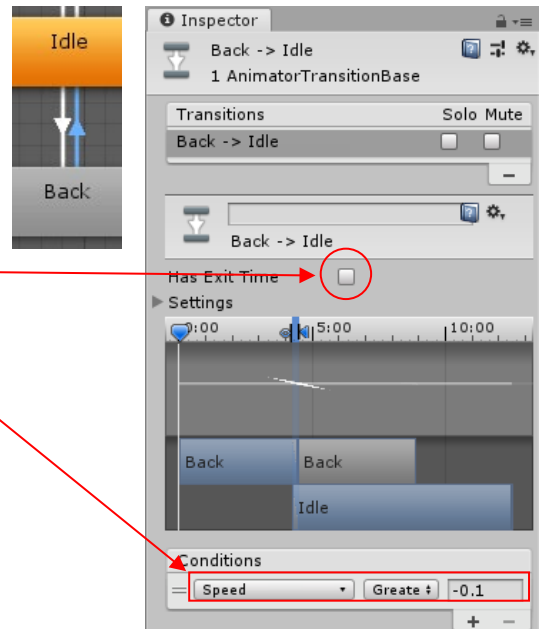


チェックボックス **HasExitTime** をオフにします。

条件を追加して組み立てます。「Speed が-0.1 を下回ったら...」

マイナスですよ！ マイナス！

- 逆向きの白い矢印 **Back -> Idle** をクリックします。



チェックボックス **HasExitTime** をオフにします。

条件を追加して組み立てます。「Speed が-0.1 を上回ったら...」

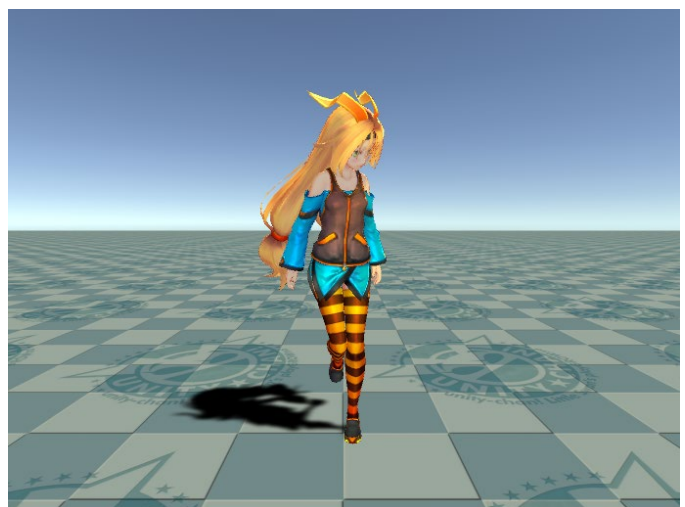
マイナスですよ！ マイナス！

- プレイボタンを押下します。



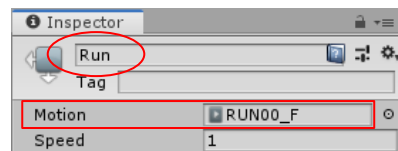
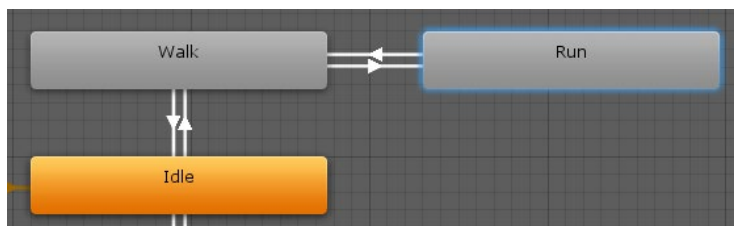
左ジョイスティックを手前に引いている間、ユニティちゃんが後退するモーションになることを確認します。

一応、キーボードの上キー／下キーや、Wキー／Sキーで代替機能が果たせているか？も確認します。



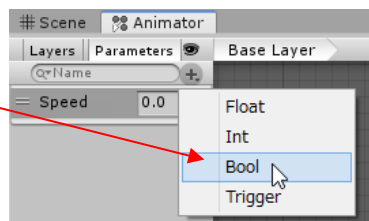
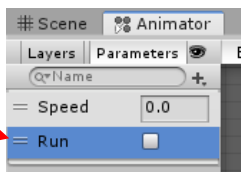
【STEP8】 走るモーション

- 【命題】 アニメーターコントローラーに新しい状態(箱)を作って名称を **Run** とし、Motion を **RUN00_F** に指定します。また、白い矢印 2 本で図のように接続して下さい。



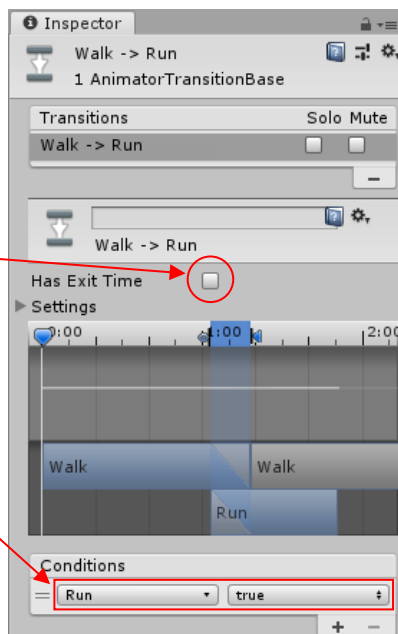
走る行為は **Walk** から派生して再生されるのが妥当です。

- Animator ウィンドウのパラメータータブで、**真偽型 (Bool 型)** のパラメータを追加します。



- 名称を **Run** とします。

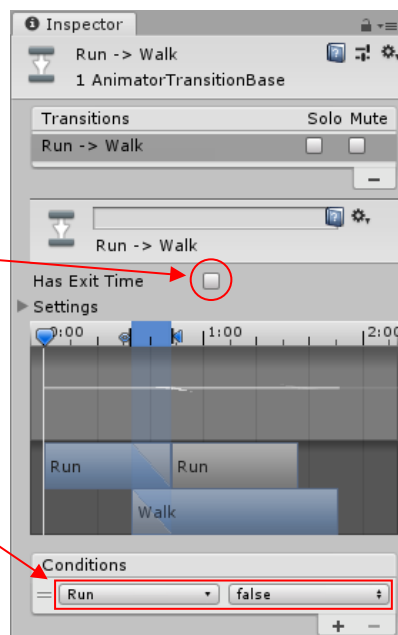
- 白い矢印 **Walk -> Run** をクリックします。



チェックボックス **HasExitTime** を**オフ**にします。

条件をプラス(+)で追加して組み立てます。「**Run が true**」

- 白い矢印 **Run -> Walk** をクリックします。



チェックボックス **HasExitTime** を**オフ**にします。

条件を追加して組み立てます。「**Run が false**」

- スクリプト **PlayerAction** を次のように編集します。

普段は TriggerR の値はゼロなので、毎フレームで false を伝えています。右トリガーを押下して、その引き代の値が 0.7 を越えたら、true を伝えます。

つまり、true と false のどちらかが、常にパラメータ Run に送られ続けます。

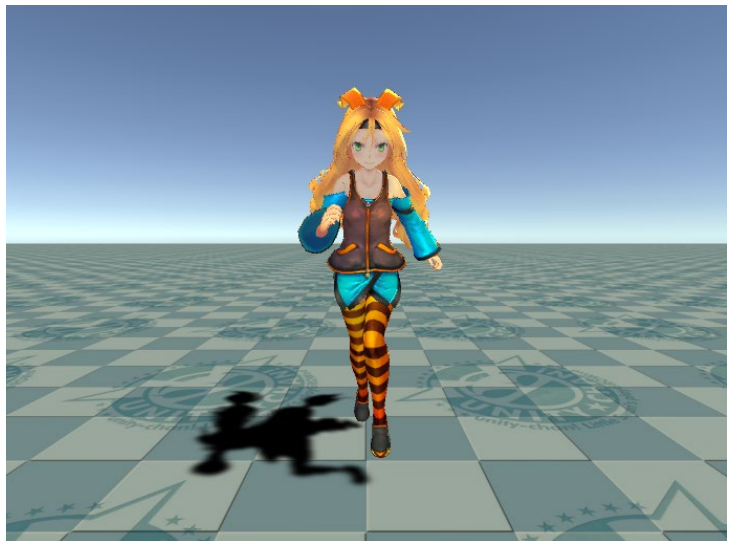
```
//～前略～  
void Update() {  
    float zengo;  
    zengo = Input.GetAxis( "Vertical" );  
    myAnim.SetFloat( "Speed", zengo );  
    if (Input.GetAxis( "TriggerR" ) < 0.7f) {  
        myAnim.SetBool( "Run", false );  
    } else {  
        myAnim.SetBool( "Run", true );  
    }  
}
```

- プレイボタンを押下します。



ジョイスティックを倒して歩いている状態になったら、右トリガーを引きます。
その間だけ、走るモーションへ遷移することを確認します。

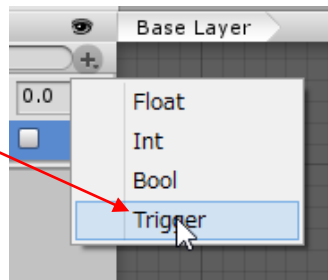
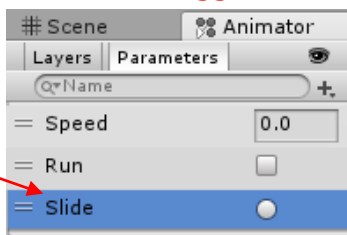
キーボードの右Altキーでも代替できます。



【STEP9】 ループしない状態の設置

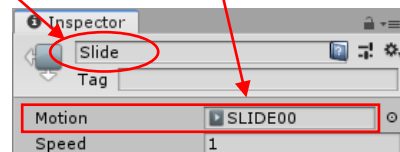
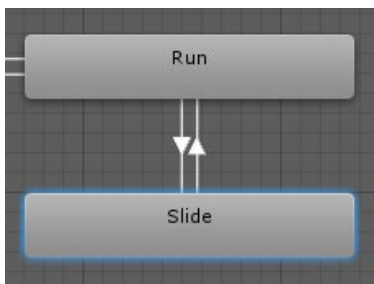
走る、歩く、待機するなどのモーションは全て**ループ・モーション**でしたが、滑りこんだりジャンプしたり、キックやパンチなどは**一発屋なモーション**です。滑り込むモーションを行う為のパラメータを作成します。

- Animator ウィンドウのパラメータータブで、**Trigger(引き金)型**のパラメータを追加します。
- 名称を **Slide** とします。



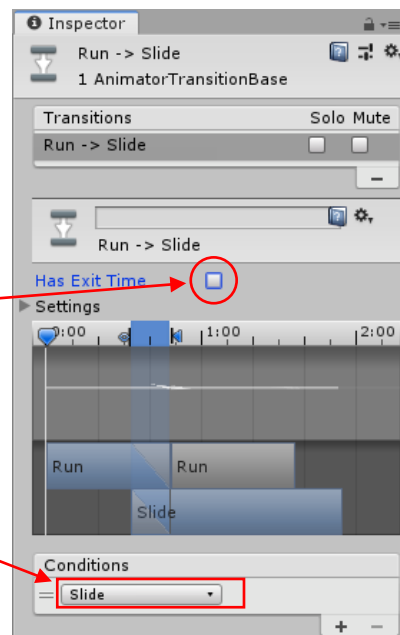
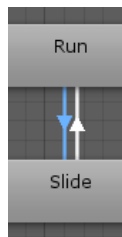
- 【命題】 アニメーターコントローラーに新しい状態(箱)を作って名称を **Slide** とし、Motion を **SLIDE00** に指定します。

また、白い矢印 2 本で図のように接続して下さい。



走る行為から発生するのが妥当です。

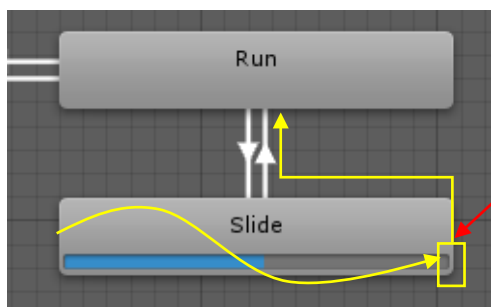
- 白い矢印 **Run -> Slide** をクリックします。



- チェックボックス **HasExitTime** を**オフ**にします。
- 条件を追加して組み立てます。**Slide が宣せられたら**。

- 白い矢印 **Slide -> Run** は何も設定しません。

元々ExitTime がオンなので、アニメーションが再生されたらこの矢印をたどるようになっているからです。

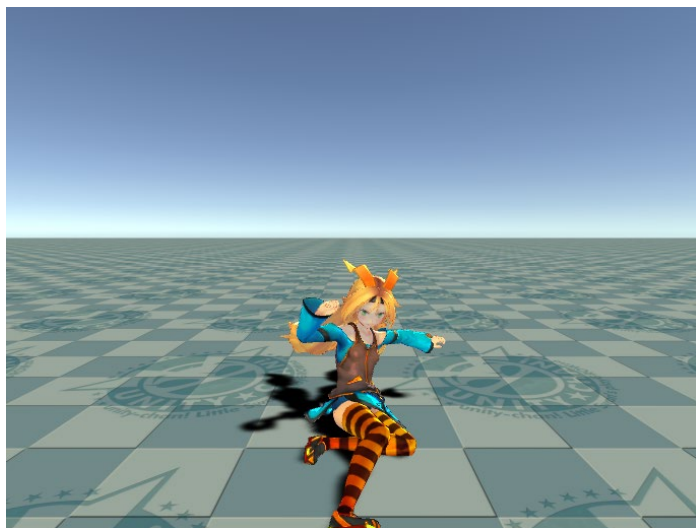


この ExitTime を持っているので、条件は「ここに来たら」となる。

- 【命題】 走っている状態であり、かつ、左バンパー押下を検出したら、パラメータ Slide を1回だけ発動します。次のような命令になります。スクリプト PlayerAction の妥当な位置に挿入してください。

```
if (Input.GetButtonDown( "BumperL" )) {
    myAnim.SetTrigger( "Slide" );
}
```

- プレイボタンを押下します。
走る状態を作ってから左トリガーを引きます。
ユニティちゃんが滑りこむモーションが再生されます。



キーボードのLキーでも代替機能になります。

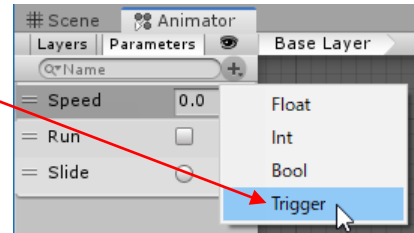
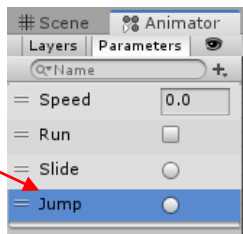
【STEP10】 どの状態からでも遷移可能なモーション

ジャンプの動作は、走る、歩く、待機のどの状態からでも(例外はあると思いますが)実行が可能です。

しかし増え続ける全ての状態からジャンプへの「白い矢印」を作るのは複雑になってしまいます。そこで **AnyState** (どの状態でも)という概念が準備されています。

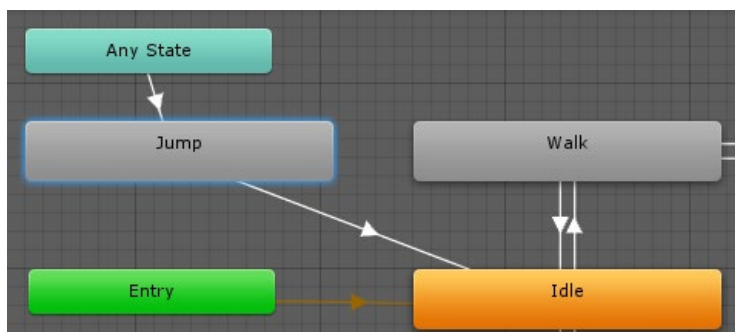
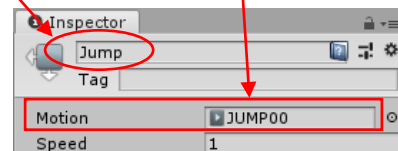


- Animator ウィンドウのパラメータータブで、**Trigger(引き金)型**のパラメータを追加します。
- 名称を **Jump** とします。

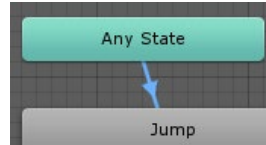


- 【命題】 アニメーターコントローラーに新しい状態(箱)を作って名称を **Jump** とし、Motion を **JUMP00** に指定します。

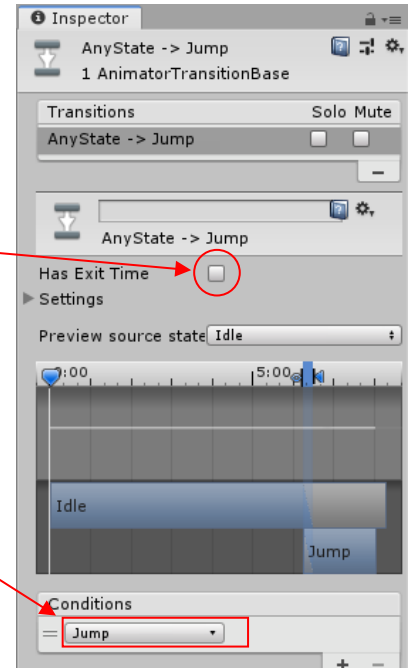
また、状態遷移の白い矢印は AnyState から発生して Jump へ、ジャンプが終わったら一旦、待機モーション Idle へ戻る設定とします。



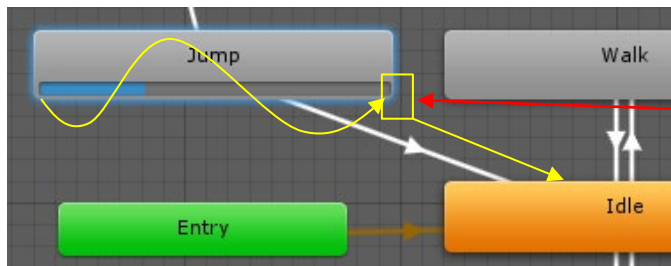
- 白い矢印 **AnyState -> Jump** をクリックします。



チェックボックス **HasExitTime** を**オフ**にします。
 条件を追加して組み立てます。**Jump** が宣せられたら。




- 白い矢印 **Jump -> Idle** には何も設定しません。ジャンプが終わったら自動的に Idle へ進ませます。

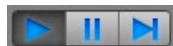


ここに ExitTime があって、到着したら次へ遷移します。

- 【命題】 走っていろいろが何をしようが、ボタン「BtnA」が押下されたことを検出し、トリガー「Jump」をステートマシンに伝えてジャンプを行います。スクリプト PlayerAction をそのように編集してください。

- プレイボタンを押下します。
 どの状態からでもボタンA  でジャンプの動作が再生されます。

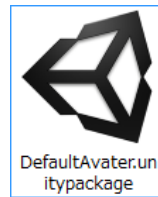
マウスの左ボタンでも代替機能になります。



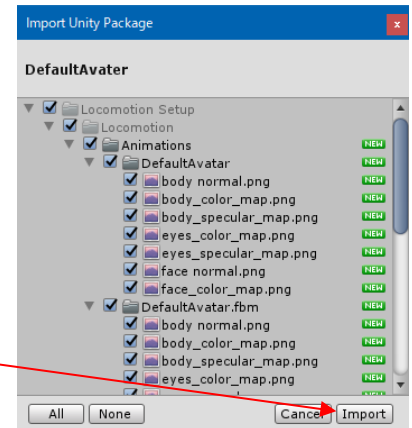
ステートマシンの互換性

【STEP11】異なるキャラクターへの反映

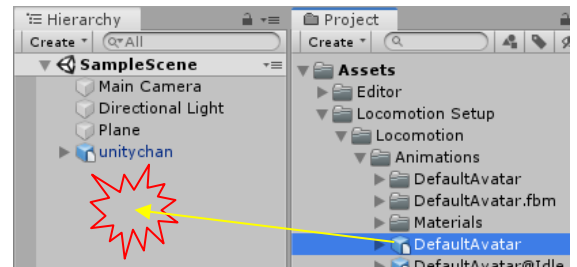
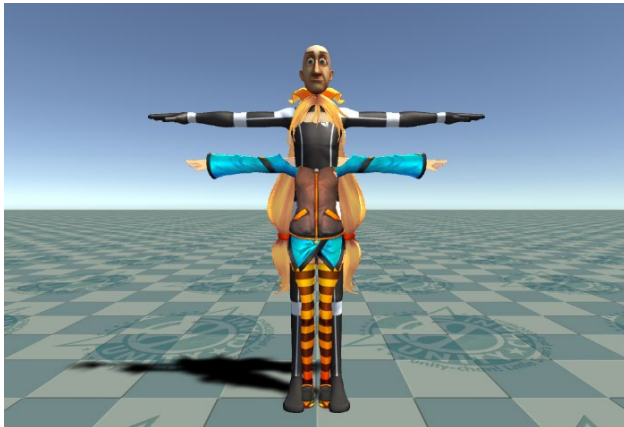
- メニューAssets から Import Package > Custom Package と進み、今回のフォルダ内にある **DefaultAvater.unitypackage** を指定します。



内容物が表示されたら **Import** を押下します。

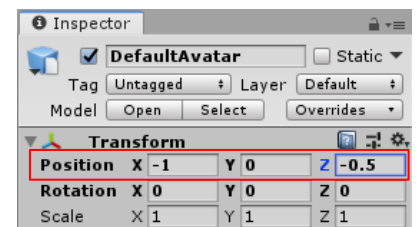


- プロジェクト欄のフォルダ LocomotionSetup(ロコモーションセットアップ) > Locomotion(ロコモーション) > Animations > **DefaultAvater(デフォルトアバター)**をヒエラルキー欄にドラッグ & ドロップします。

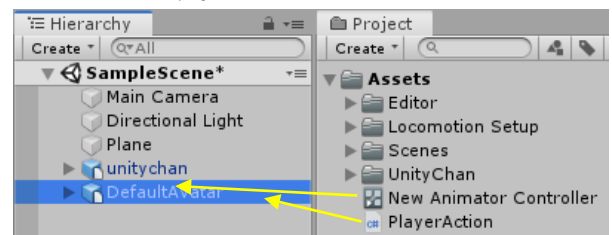


ちょっと、大変なことになっています。

インスペクタで**パラメータ**を設定します。



- ヒエラルキー欄の DefaultAvatar に、今まで作ってきたアニメーターコントローラー **New Animator Controller** とプログラム **PlayerAction** を両方ともドラッグ & ドロップします。



- プレイボタンを押下します。

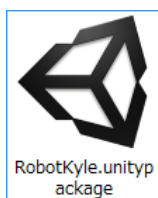


もう一体のキャラクターがユニティちゃんと同じアニメーション動作をすることを確認します。
キー操作も同様です。



Humanoid(ヒューマノイド: 人体)型の骨格構造を持っていることを条件に、モーションコントローラーを共有することができます。物理的な形状モデルと、走る・歩くなどの動きのデータは、互いに束縛されてはならず、自由に使い回せる事を意味しています。

- 【命題】 配布してある RobotKyle(ロボットカイル)をステージに登場させ、3体で同じモーションを行うように設定してください。



MMDモデルデータをたくさんダウンロードしてきて、全員揃って同じモーションをさせることも可能です。

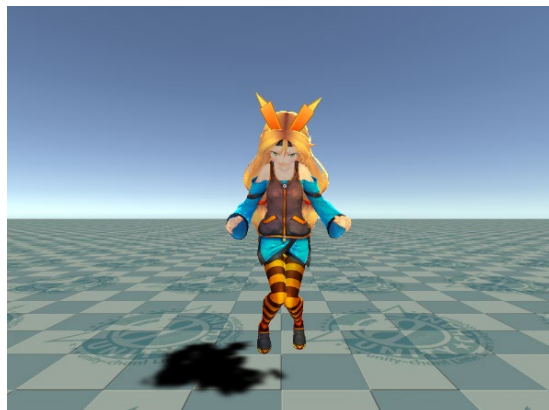


これで完成です。

【STEP12】 ジャンプが終わるまでジャンプを却下するには

現状では、どんなモーション状況下であっても、ジャンプモーションを行います。

すると、ジャンプ中であってもジャンプが可能になっています。これの対策案は色々あるのですが、Unity 特有のステートマシンの機能を用い、これを却下する方法を紹介します。



- スクリプト **PlayerAction** を次のように編集します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerAction : MonoBehaviour {

    static public bool CanJump;
    Animator myAnim; //自身のアニメーター

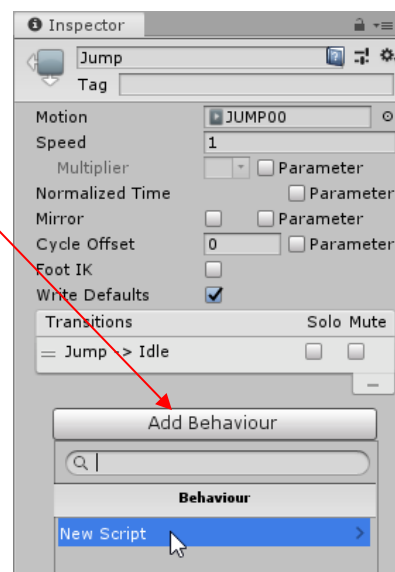
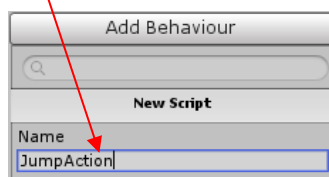
    void Start() {
        CanJump = true;
        //自身のアニメーターを取得
        myAnim = GetComponent<Animator>();
    }

    void Update() {
        if (Input.GetButtonDown( "BtnA" ) && CanJump) {
            myAnim.SetTrigger( "Jump" );
        }
    }

    //～後略～
}
```

- アニメーター設定画面に戻り、状態 **Jump** を選択します。

インスペクタの Add Behaviour(アッド・ビヘイビア:ふるまいを追加)を押下して **New script** を選択し、名称を **JumpAction** とします。



- スクリプト **JumpAction** を次のように編集します。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class JumpAction : StateMachineBehaviour {

    // OnStateEnter is called when a transition starts and the state machine ...
    override public void OnStateEnter(Animator animator,
        AnimatorStateInfo stateInfo, int layerIndex) {
        PlayerAction.CanJump = false;
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter ...
    //override public void OnStateUpdate(Animator animator, AnimatorStateInfo ...
    //{
    //
    //
    //}

    // OnStateExit is called when a transition ends and the state machine ...
    override public void OnStateExit(Animator animator,
        AnimatorStateInfo stateInfo, int layerIndex) {
        PlayerAction.CanJump = true;
    }

    //～後略～
}
```

以上