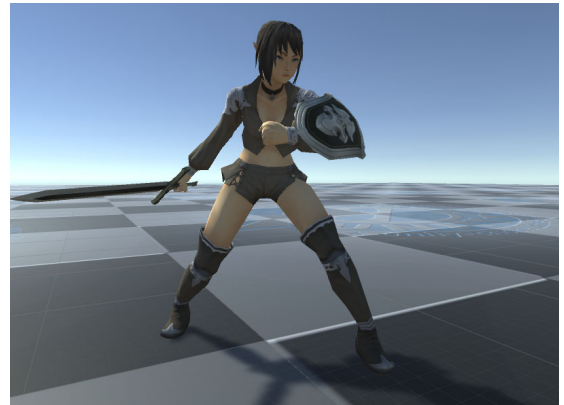


BattleSystem (バトルシステム: 3Dキャラの肉弾戦の仕組み)

3Dキャラクターを扱ったアクションゲームに用いられるバトルシステムについて、基本的な仕組みを演習します。移動や戦闘時における各種設定を中心に紹介しています。

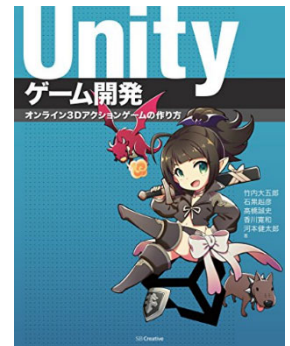
今回制作するコンテンツは、自キャラを操作して敵と肉弾戦を行う仕組みのみで、ゲームの要素はありません。

自身が移動し攻撃する仕組みを実現したら、敵キャラクターを設置して待機・移動・ダウン・攻撃などを実現させます。また、自身のダメージもヘルスバーなどで管理し、最後にダウンも実現させて、肉弾戦を Unity でどのように表現するか？の演習となります。



この教材はもともと、独習形式でゲームエンジンの理解を進めていく良書「Unity ゲーム開発:SB クリエイティブ社刊」について、バージョンや使用概念が古い為に扱われなくなっていくことが非常に残念であり、その一部であるバトル構造の構築部分だけでも、現行バージョンの Unity で簡素に判り易く説明が出来ないか？との思いでほんの一部分を再構成したものです。

バージョンの関係で販売扱いは無くなったようですが、学校の図書として残っているの、一度手に取り、3Dアクションゲーム構築の追体験をお勧めします。通信対戦部分は非常に古くて参考にならないかも知れませんが。。

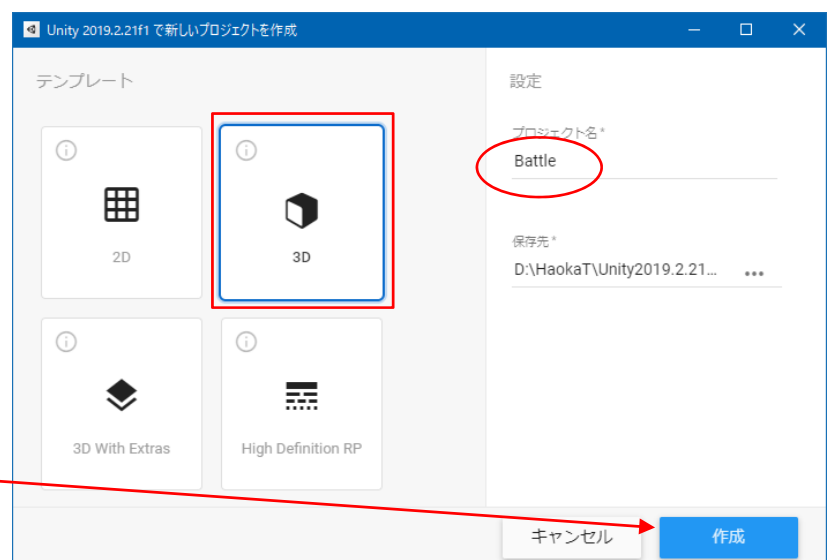


シーンの準備

[STEP1] プロジェクトの作成

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

ボタン作成を押下します。



- Game 画面のサイズを **Standalone (1024x768)** とします。



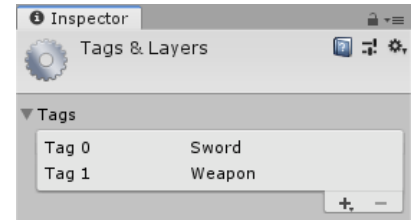
【STEP2】 タグの準備

- ヒエラルキー欄の MainCamera を選択し、インスペクタの Tag から **Add Tag...** を選択します。
- タグのプラスボタン(+)を押下して、次の2つのタグを追加します。

Sword (ソード:剣)

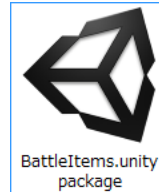
Weapon (ウェポン:武器)

こんな感じになります。

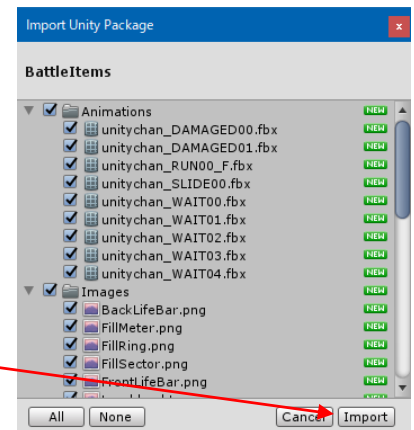


【STEP3】 素材の読み込み

- 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 **BattleItems.unitypackage** を指定します。

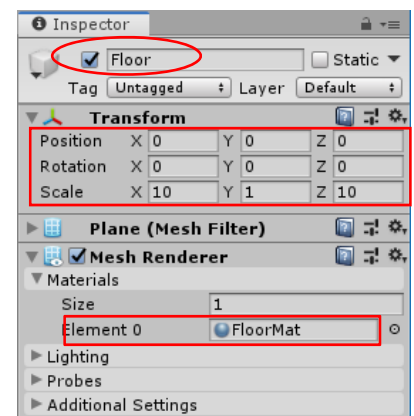
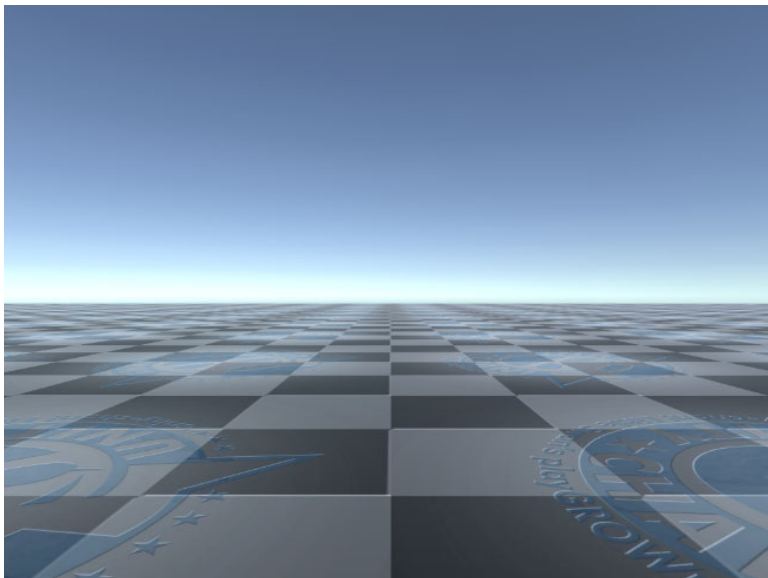


- 内容物が表示されたら、Import を押下します。
旧データなので、少量の警告が出ますが、スルーしても大丈夫です。



【STEP4】 シーンの設定

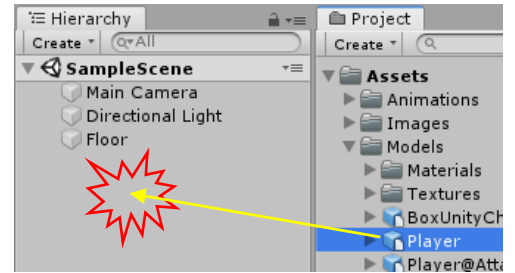
- ヒエラルキー欄の Create から 3D Object > **Plane** を選択し、名称を **Floor** とします。
インスペクタでパラメータを設定します。



プレイヤーの運営

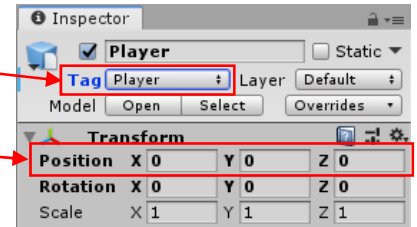
【STEP5】プレイヤーの設置

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



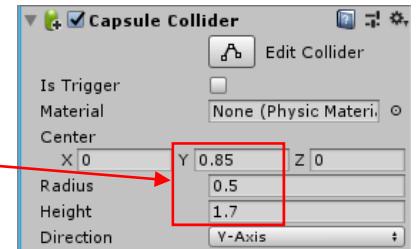
- インスペクタでタグを **Player** に設定します。

念の為、**原点(0,0,0)**にいるか？を確認します。



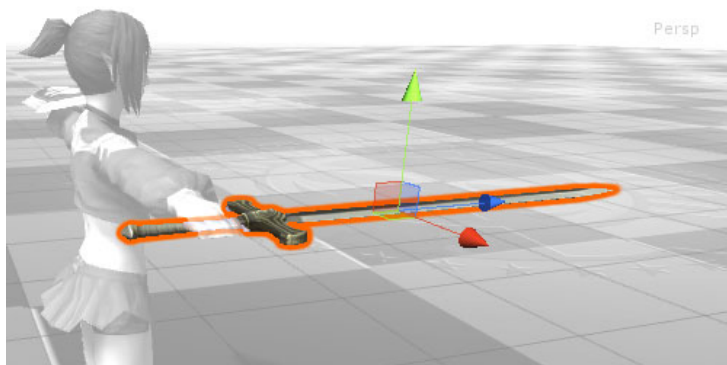
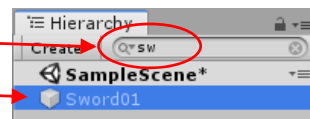
- インспекタの Add Component を押下して、Physics > **Capsule Collider** を選択します。

インспекタでパラメータを設定します。

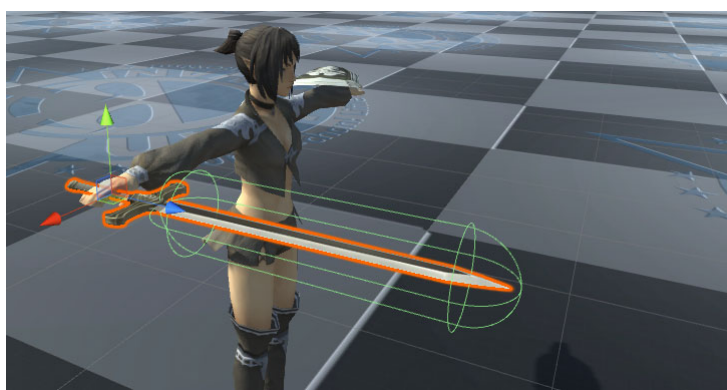
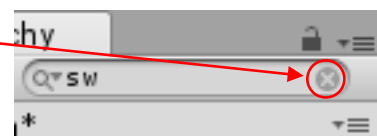


- ヒエラルキー欄の検索欄で **sw** と入力し、検索します。

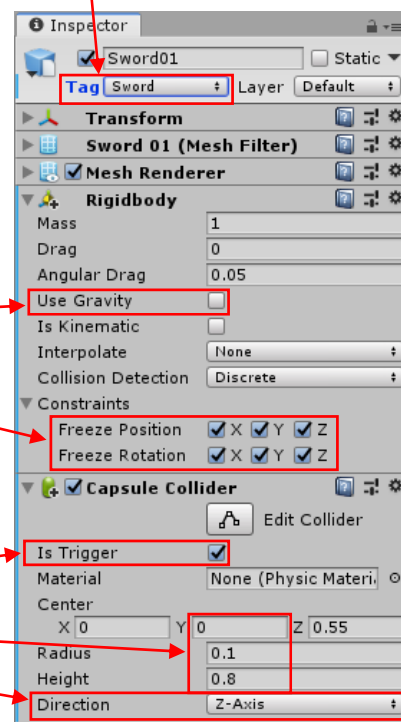
Sword01 のみが残るので、これを選択します。



選択が終われば、このモードは用済みなので、**クローズ(X)**を押下し、検索表示を解除しておきます。



タグを **Sword** に設定します。



- インспекタの Add Component を押下して、Physics > **Rigidbody** を選択します。

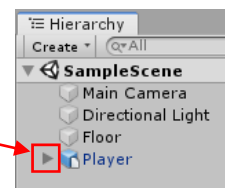
インспекタでパラメータを設定します。

- インспекタの Add Component を押下して、Physics > **Capsule Collider** を選択します。

インспекタでパラメータを整えます。

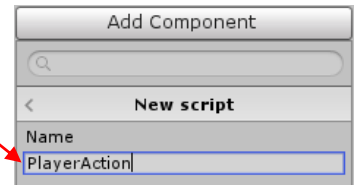
Z-Axis なので注意のこと！

- Player の構造が広がったままですので、▼マークを押下して閉じておきます。



【STEP6】プレイヤーの移動

- ヒエラルキー欄の Player を選択し、インスペクタの Add Component から最下段の **New script** を選択します。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;
using UnityEngine.UI; //後ほどUI要素を扱う際に必要です。

public class PlayerAction : MonoBehaviour {

    public GameObject PatSmoke;
    public GameObject PatStrong;
    public GameObject PatBubble;
    public GameObject PatHeal;
    public bool CanAttack = false;
    public bool isDead = false;

    void Start () {

    }

    void Update () {
        // 入力操作を取得する
        float h = Input.GetAxis("Horizontal"); //左右移動を取得する
        float v = Input.GetAxis("Vertical"); //前後移動を取得する
        Vector3 dir = new Vector3(h, 0, v);
        // 入力方向へ徐々に向いていく(回る)
        if (dir.sqrMagnitude > 0.01f) {
            Vector3 LookPos = Vector3.Slerp(transform.forward, dir, 8.0f * Time.deltaTime);
            transform.LookAt(transform.position + LookPos);
        }
        // 入力方向へ移動する
        transform.position += dir.normalized * 4.0f * Time.deltaTime;
    }
}
```

このキャラクターへの方向指示ベクトル dir について、認定する大きさ 0.1 以上の時に、回転が与えられます。

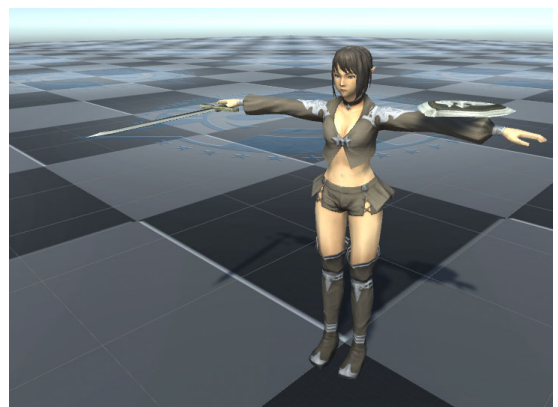
その判定式 `dir.magnitude > 0.1f` について、右辺左辺共に正の値なので、2乗しても不等号関係は変わりません。ならば2乗して `dir.sqrMagnitude > 0.01f` にすれば、①平方根の計算が不要で速い！②小数の最後が欠落せず正確！となります。

- プレイボタンを押下します。



方向キーでプレイヤーが移動することを確認します。

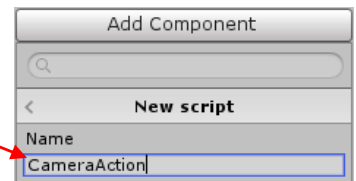
(Xbox コントローラーの場合、左ジョイスティックの操作で、Horizontal と Vertical の信号を送って来るので、特別な記述や設定をしなくても、対応が来ています。)



【STEP7】フェイス撮影の実現

フォロー撮影も可能ですが、ここではフェイス撮影での実現例とします。カメラがプレイヤーを追いかけるように改造します。

- ヒエラルキー欄の **MainCamera** を選択し、インスペクタの Add Component をから **New script** を選択します。**CameraAction** と命名します。



- スクリプト **CameraAction** を次のように編集します。

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


public class CameraAction : MonoBehaviour {

    GameObject Player;
    Vector3 CamPos = new Vector3(0.0f, 3.0f, -2.5f);
    Vector3 OffSet = new Vector3(0.0f, 1.5f, 0.0f);

    void Start () {
        Player = GameObject.FindGameObjectWithTag("Player");
    }

    void LateUpdate() {
        transform.position = Player.transform.position + CamPos;
        transform.LookAt(Player.transform.position + OffSet);
    }

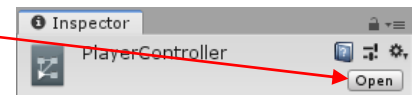
}
```

- プレイボタンを押下します。 プレイヤー移動にカメラが追従することを確認します。

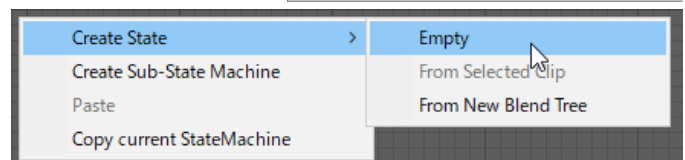


【STEP8】ステートマシンの基本設定

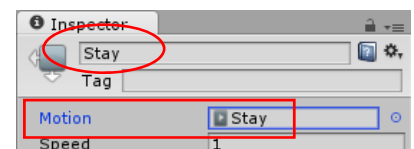
- プロジェクト欄の Create から **Animator Controller**(アニメーターコントローラー)を作成します。名称を **PlayerController** とします。**Open** を押下して編集状態にします。



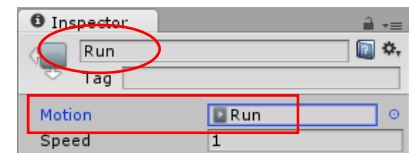
- 背景の方眼紙を右クリックし、Create State > Empty と進み、空(から)の状態を作成します。



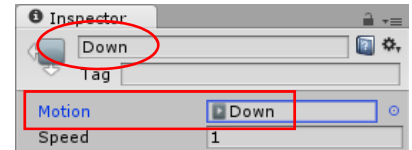
- オレンジ色の New State(新しい状態)を選択し、インスペクタで名称を **Stay** とし、Motion を **Stay** にします。



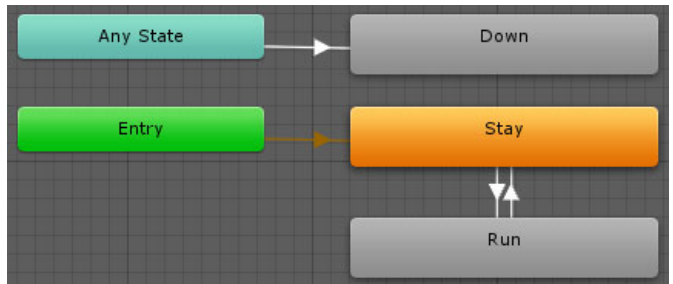
- 同様に新しい状態を作り、名称を **Run**、Motion も **Run** にします。



- 同様に新しい状態を作り、名称を **Down**、Motion も **Down** にします。



- Stay を右クリックして **Make Transition** を選び、白い矢印がマウスについてくるようになったら、そのまま **Run** をクリックして接続します。
- その逆向きの白い矢印も作成します。
- AnyState から Down に向けて白い矢印を作成します。



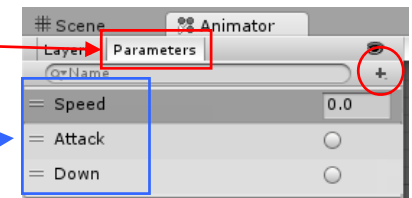
こんな感じになります。

- パラメータタブを選択します。
 プラスボタン(+)を押下して以下のデータ型でパラメータを3個定義します。

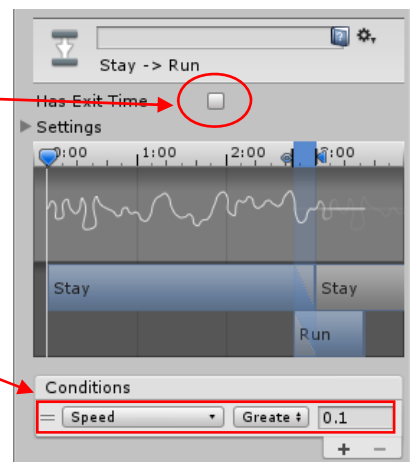
Speed (Float 型)

Attack (Trigger 型)

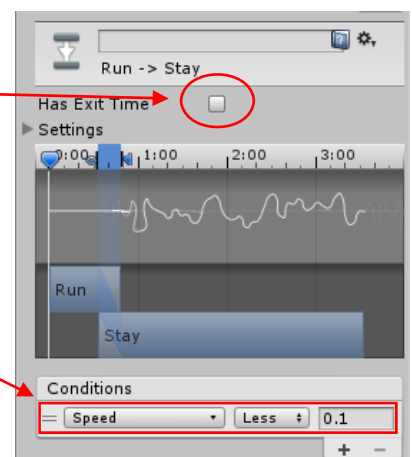
Down (Trigger 型)



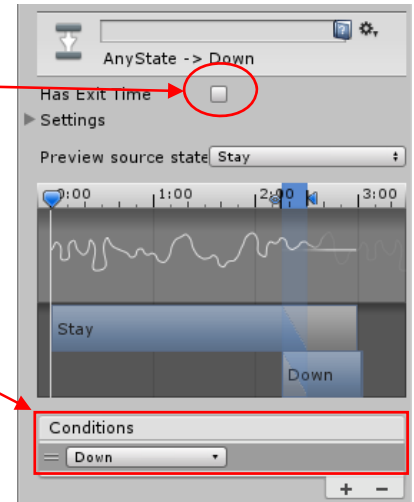
- 白い矢印 **Stay -> Run** を選択し、遷移条件 (Conditions) として、**HasExitTime** をオフにして、**Speed が 0.1 を超えたら (Grater)** とします。



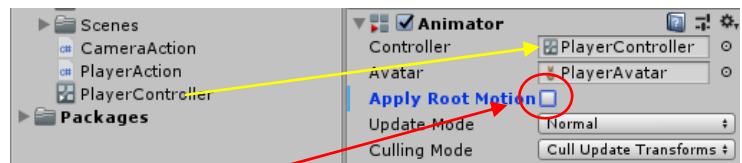
- 白い矢印 **Run -> Stay** を選択し、遷移条件 (Conditions) として、**HasExitTime** をオフにして、**Speed が 0.1 を下回ったら (Less)** とします。



- 白い矢印 **AnyState** -> **Down** を選択し、遷移条件 (Conditions) として、**HasExitTime** をオフにして、**Down** が宣せられたら、とします。



- ヒエラルキー欄の **Player** を選択し、Animator コンポーネントの Controller に **PlayerController** をドラッグ & ドロップします。



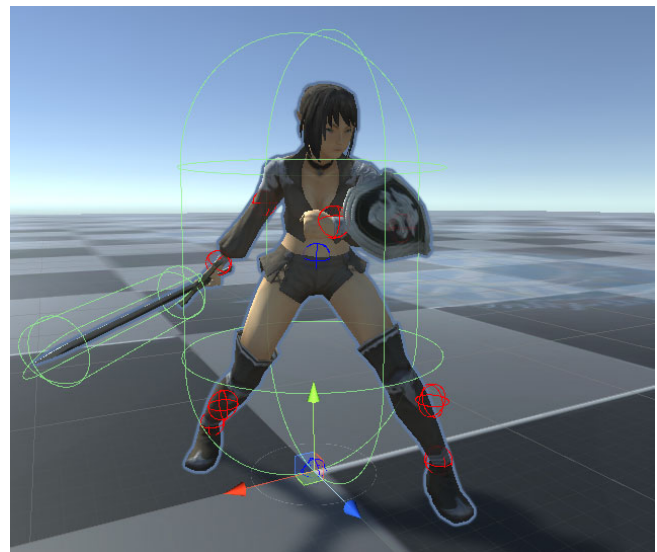
- 項目 **Apply Root Motion** をオフにします。

各種のモーションが終わった時に、キャラクターの位置がずれていることがあり、連続して同じ動き (ジャンプやキック) を行った際に、そのズレを加えて (Apply) いくと、少しずつ移動してしまう現象を防ぐ措置です。

- プレイボタンを押下します。
プレイヤーが戦闘待機のモーションとなることを確認します。



このまま移動すると、奇妙な現象になってしまうので、移動情報をメカニムに伝わるようにします。




【STEP9】 移動とステートマシンの連動

- スクリプト **PlayerAction** を編集します。注意！ **CameraAction** ではない！

```
// 前略
Animator MyAnim; // 自身のアニメーター

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

void Update() {
    // 入力操作を取得する
    float h = Input.GetAxis( "Horizontal" ); //左右移動を取得する
    float v = Input.GetAxis( "Vertical" ); //前後移動を取得する
    Vector3 dir = new Vector3( h, 0, v );
    // 入力方向へ徐々に向いていく(回る)
    if (dir.sqrMagnitude > 0.01f) {
        Vector3 forward = Vector3.Slerp (transform.forward, dir, 8.0f * Time.deltaTime);
    }
    // 入力方向へ移動する
    transform.position += dir.normalized * 4.0f * Time.deltaTime;
    MyAnim.SetFloat("Speed", dir.magnitude);
}
}
```

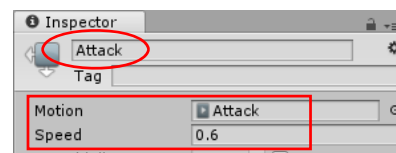
- プレイボタンを押下します。
プレイヤーが移動時に走行モーションになることを確認します。



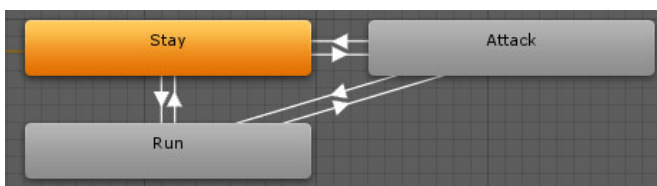
攻撃モーションとの共存

【STEP10】 攻撃モーションの実現

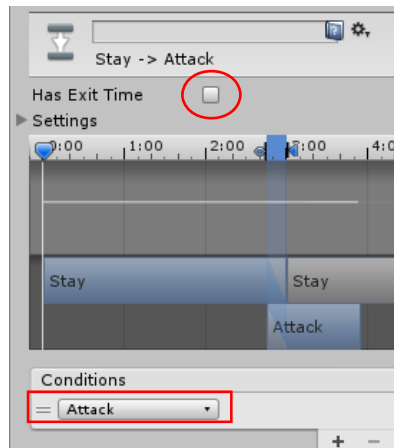
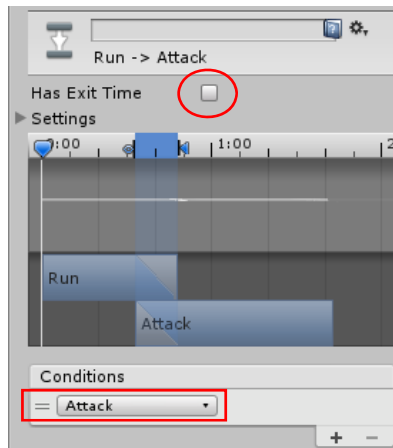
- プロジェクト欄の **PlayerController** を編集状態にします。
- 新しい状態を作成し、名称を **Attack** とします。Motion にも **Attack** を設定します。
- 少し攻撃行動が速いモーションなので、再生速度を **0.6 倍** にしてゆっくりした動きにします。



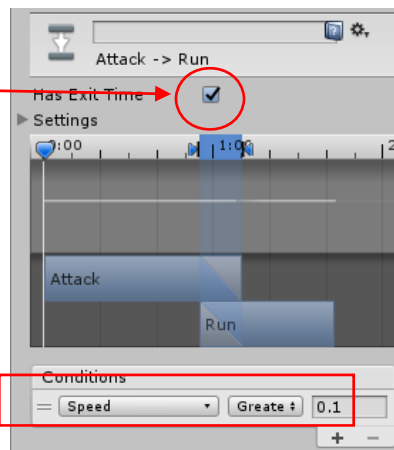
- **Stay** と **Attack** を行き来するトランジション(白い矢印)を2つ作成します。
- **Run** と **Attack** を行き来するトランジション(白い矢印)を2つ作成します。



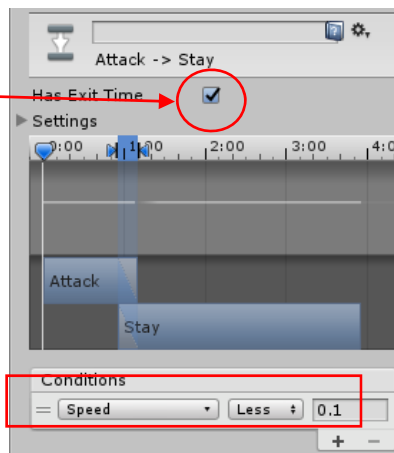
- 両方から Attack へ向かう条件は同じとなり、Has Exit Time をオフにして、Conditions に Attack を指定します。



- Attack -> Run の遷移条件として、HasExitTime がオン、Speed が 0.1 を超えていたら (Greater) とします。



- Attack -> Stay の遷移条件として、HasExitTime がオン、Speed が 0.1 を下回っていたら (Less) とします。



- スクリプト PlayerAction を次のように編集します。

```
// 前略

// 入力方向へ移動する
transform.position += dir.normalized * 4.0f * Time.deltaTime;
MyAnim.SetFloat("Speed", dir.magnitude);

if (Input.GetButtonDown( "Fire1" ) ) {
    MyAnim.SetTrigger("Attack"); // 攻撃モーションの発動
}
}
```

Input.GetMouseButtonDown(0) では LMB で検出できますが、Xbox コントローラーに対応できていません。Input.GetButtonDown("Fire1") と記述することで、LMB と Xbox コントローラーのAボタンの両方に対応できます。

- プレイボタンを押下します。



LMB(Xbox コントローラーのAボタン)を押下すると、プレイヤーが攻撃モーションになることを確認します。

おや？移動したまま(方向キーを押下したまま)LMBクリックすると、足を大地に踏ん張って攻撃しているのに、地面を滑って移動してしまうという現象が発生してしまいます。

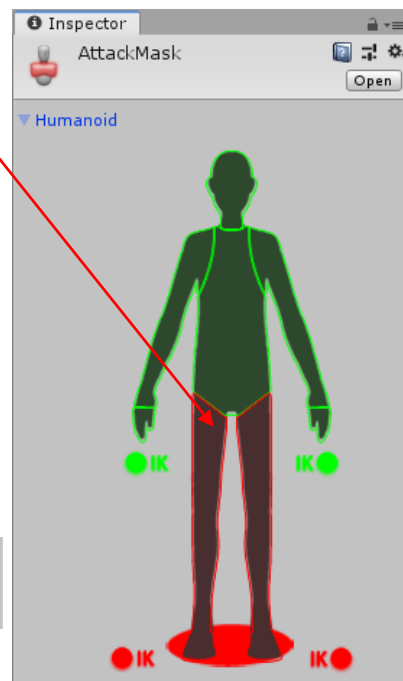


【STEP11】 上半身のみ攻撃モーションにする

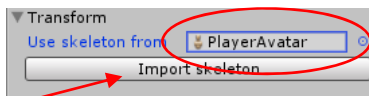
- プロジェクト欄の Create から **AvatarMask**(アバターマスク)を作成し、**AttackMask** と命名します。

- インспекタの ▼**Humanoid** をクリックして開き、図の**部位(5か所)**をクリックして赤い状態にします。

Attack モーションの影響(足の踏ん張り)が及ばないようにするマスク(隠し)を定義します。



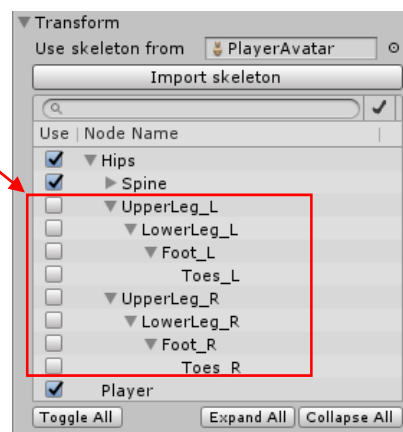
- 次に ▼**Transform** をクリックして開き、項目 Use skeleton from の丸印を押下して、**PlayerAvatar** を選択します。



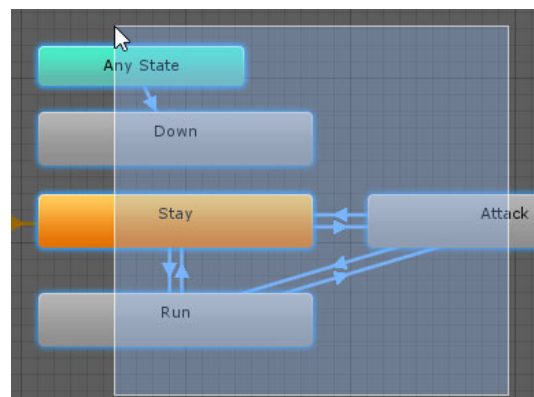
- Import skeleton を押下します。

- マスクした(赤い)部位に合致するスケルトン(骨構造)を**チェック・オフ**にします。

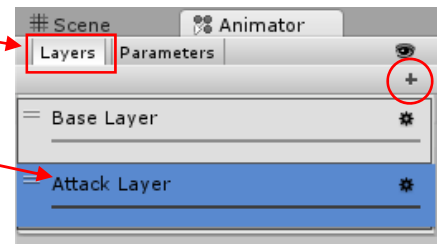
これでアバターマスクのデータが完成しました。



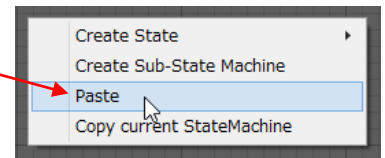
- プロジェクト欄の **PlayerController** を編集状態にします。
状態4つと **AnyState** をマウス・ドラッグで選択して**コピー**(Ctrl + C)します。



- アニメーターコントローラーPlayerAniCon の **Layers** タブを選択し、プラスボタン(+)から新規レイヤーを作成したら、名称を **AttackLayer** とします。

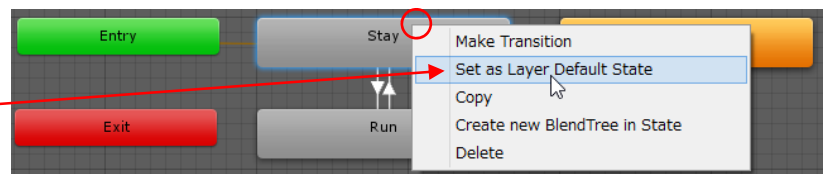


- 新しいレイヤー**Attack Layer** に来ているはずですが。
- 背景の方眼を右クリックし、先ほどコピーしていたもの(状態4つ)を**貼り付け(Paste)**します。

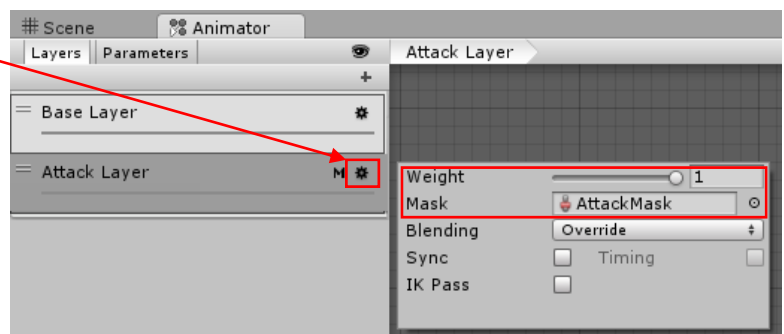


- この時にオレンジ色の箱が変更されてしまいます。

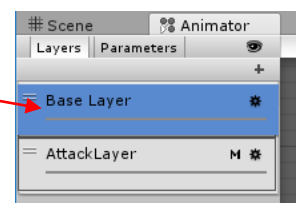
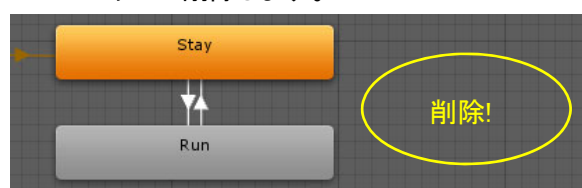
Stay を右クリックして **Set as Layer Default State** を選択してオレンジ色に戻しておきます。



- レイヤーAttackLayer の**歯車アイコン**をクリックします。
項目 Weight に **1** を設定します。
項目 Mask の右端の丸印を押下し、**AttackMask** を設定します。



- **元の Base Layer** をクリックして編集状態にします。
状態 Attack を選択し、Delete キーで削除します。



- プレイボタンを押下します。
走行中にLMB(Xbox コントローラーのAボタン)を押下して、走りながら攻撃モーションになることを確認します。



プレイヤーの状態をエフェクトにする

ゲームでは、ユーザーの操作を数値化したり視覚化したりして、その行いが価値や意義のある物に錯覚させています。視覚効果(エフェクト)の多くはパーティクル(粒子)と呼ばれる仕組みを用い、水や炎、魔法に見せかけます。ここでは Unity のパーティクルシステム(通称シュリケン・パーティクル)を用いたエフェクト生成を4つ取り上げます。

①走行時の砂煙 ②パワーアップ状態 ③毒に冒された状態 ④体力回復

今回は自作するものの、爆発や剣のきらめきなど、秀逸なエフェクトがアセットストアに数多くありますので、ダウンロードして改造しながら学ぶのも名案でしょう。

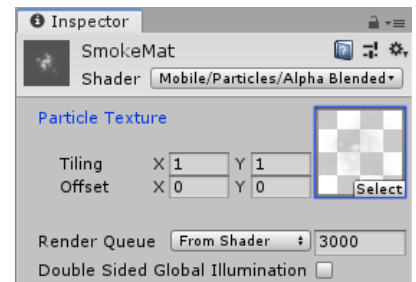
【STEP12】 走行時の砂煙エフェクト

- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **SmokeMat** と命名します。

➤ Shader : Mobile > Particles > Alpha Blended

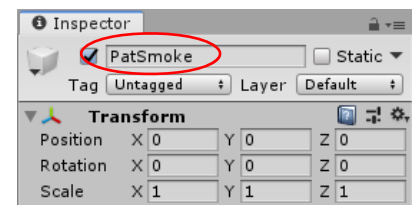
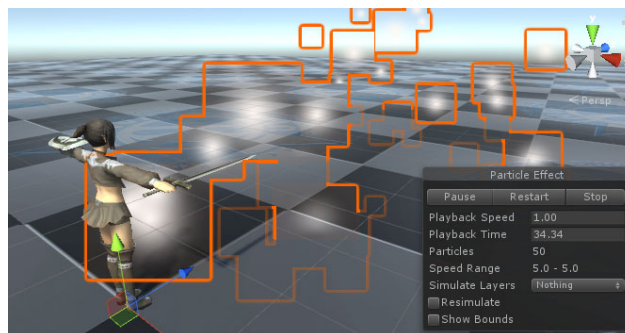
➤ Texture : tex_smoke

【学習項目】 パーティクルなどのエフェクトには、できるだけ描画が軽いものを選定すべきで、クオリティに問題が無ければモバイル環境で無くても、シェーダーにはモバイル向けに書かれたシェーダー定義を用います。

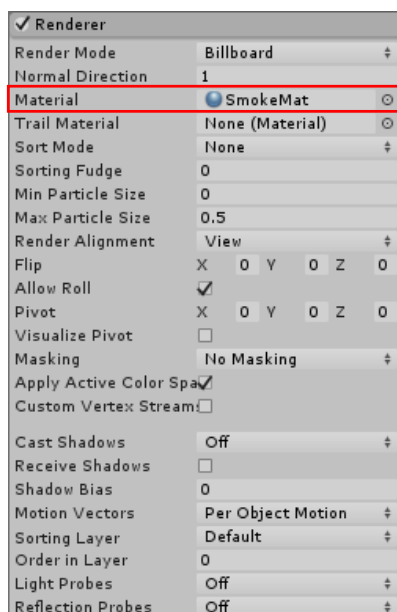


- ヒエラルキー欄の **Player** を右クリックし、Effects > **ParticleSystem** を選択します。

- 名称を **PatSmoke** と命名します。インスペクタでパラメータを設定します。

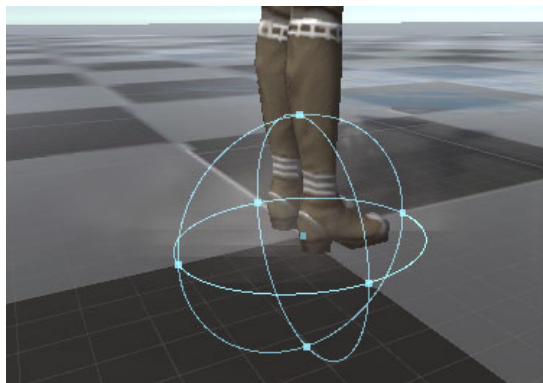


- パーティクルのパラメータを設定します。真っ先に **Renderer** を設定するのが通説です。

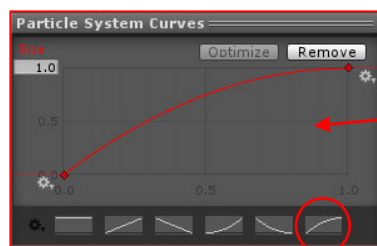
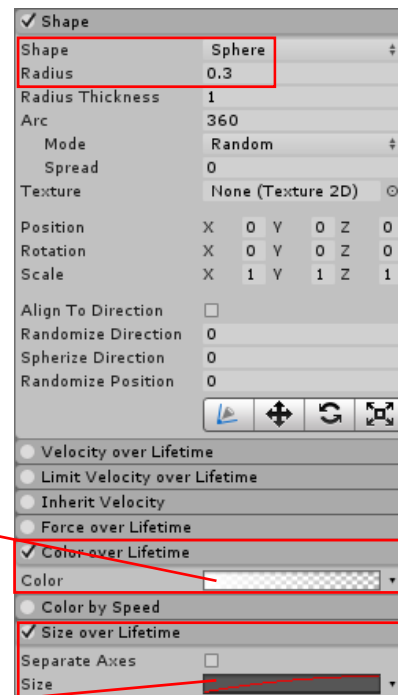
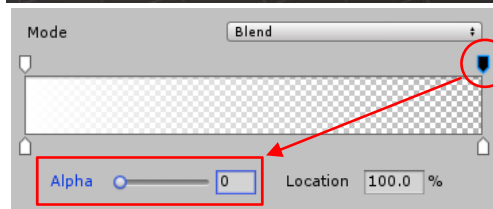


値を2つ入力する欄では、選択肢 Random Between Two Constants (2値間での乱数)を選びます。
また、英語の意味を翻訳しながら値を設定すると、設定の意味が判ってきます。





最初からチェックが付いていないプロパティは、チェックをオンにすると、設定が可能になります。



- このままだと走っていても砂煙が出てしまうので、プログラムから制御します。
スクリプト **PlayerAction** を次のように修正します。

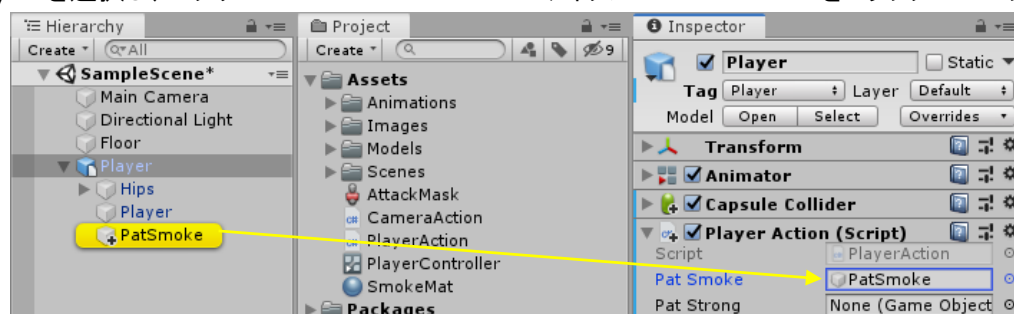
```
// 前略
ParticleSystem.MainModule SmokeMain; //砂煙の本体

void Start() {
    MyAnim = GetComponent<Animator>(); // 自身のアニメーターを取得
    SmokeMain = PatSmoke.GetComponent<ParticleSystem>().main;
}

void Update() {
    // 入力操作を取得する
    float h = Input.GetAxis( "Horizontal" ); //左右移動を取得する
    float v = Input.GetAxis( "Vertical" ); //前後移動を取得する
    Vector3 dir = new Vector3( h, 0, v );

    // 移動方向への量に応じて砂ぼこりを制御する。
    SmokeMain.startSize = dir.sqrMagnitude * 1.5f;
// 後略
```

- ヒエラルキー欄の Player を選択し、スクリプトの PatSmoke にパーティクルの **PatSmoke** をドラッグ&ドロップします。



- プレイボタンを押下します。



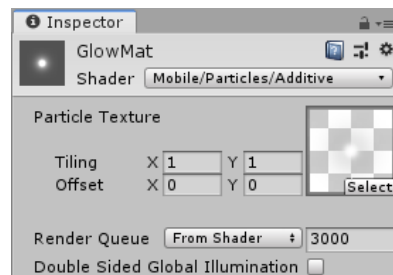
移動している時に砂煙のスモークが表現されるようになります。



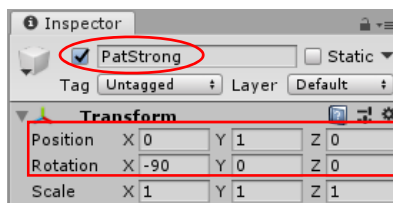
【STEP13】 パワーアップ時のエフェクト例

- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **GlowMat** と命名します。

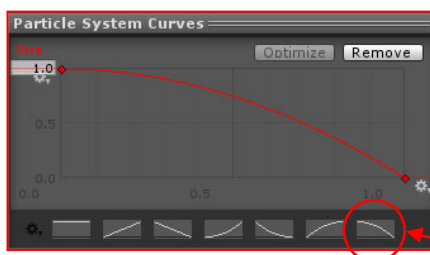
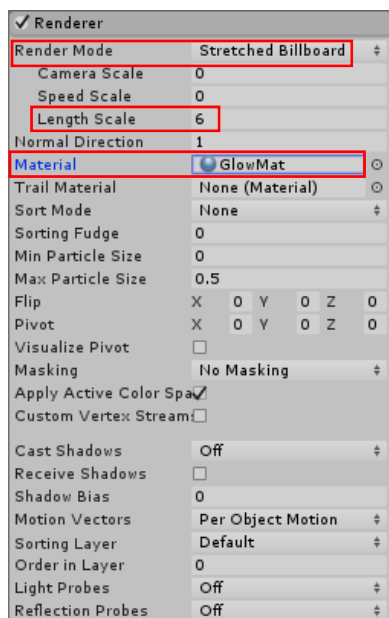
- Shader : Mobile > Particles > Additive
- Texture : tex_glow



- ヒエラルキー欄の **Player** を右クリックし、Effects > **ParticleSystem** を選択します。名称を **PatStrong** と命名します。インスペクタでパラメータを設定します。



- パーティクルのパラメータを設定します。



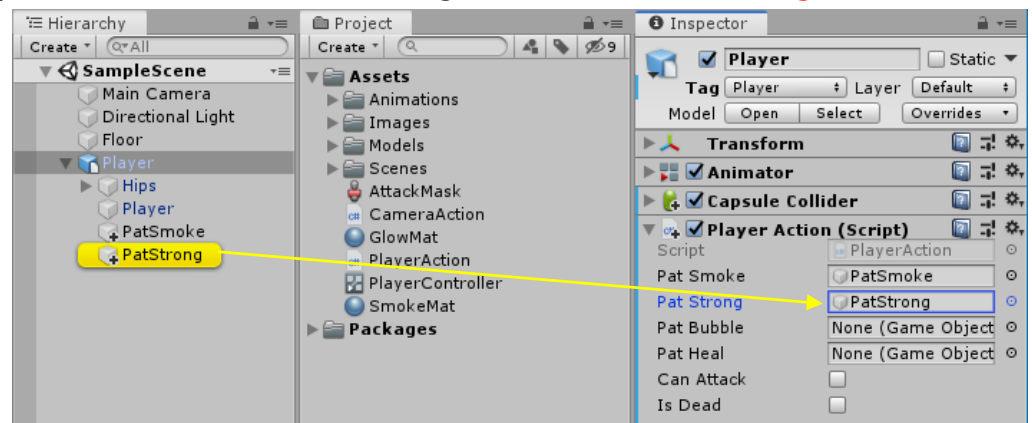
- キーボードの PageUp を押下している間だけこのエフェクトが発生するようにします。
スクリプト **PlayerAction** を編集します。

```
// 前略
void Start () {
    MyAnim = GetComponent<Animator>(); // 自身のアニメーターを取得
    SmokeMain = PatSmoke.GetComponent<ParticleSystem>().main;
    PatStrong.SetActive(false);
}

void Update () {
    // PageUpキーでパワーアップを表現する
    PatStrong.SetActive(Input.GetKey(KeyCode.PageUp));
}

// 後略
```

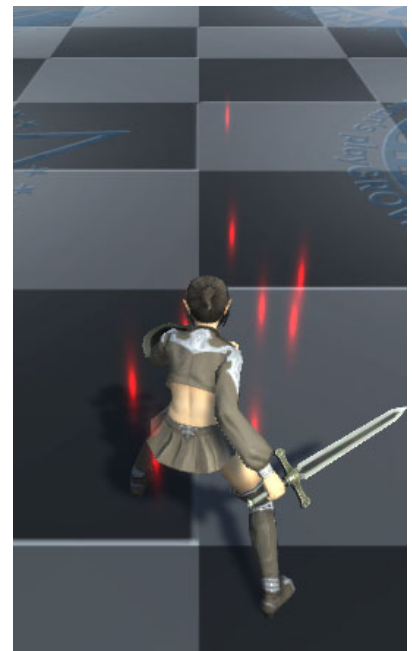
- ヒエラルキー欄の Player を選択し、スクリプトの PatStrong にパーティクルの **PatStrong** をドラッグ&ドロップします。



- プレイボタンを押下します。



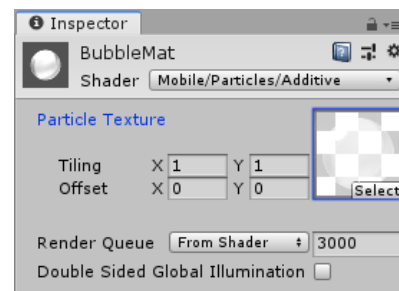
キーボードの PageUp を押下している間だけパワーアップ状態のエフェクトが出るようになりました。



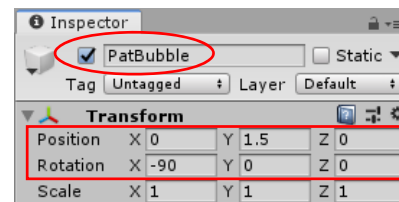
【STEP14】 毒に冒された時のエフェクト例（反復練習です）

- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **BubbleMat** と命名します。

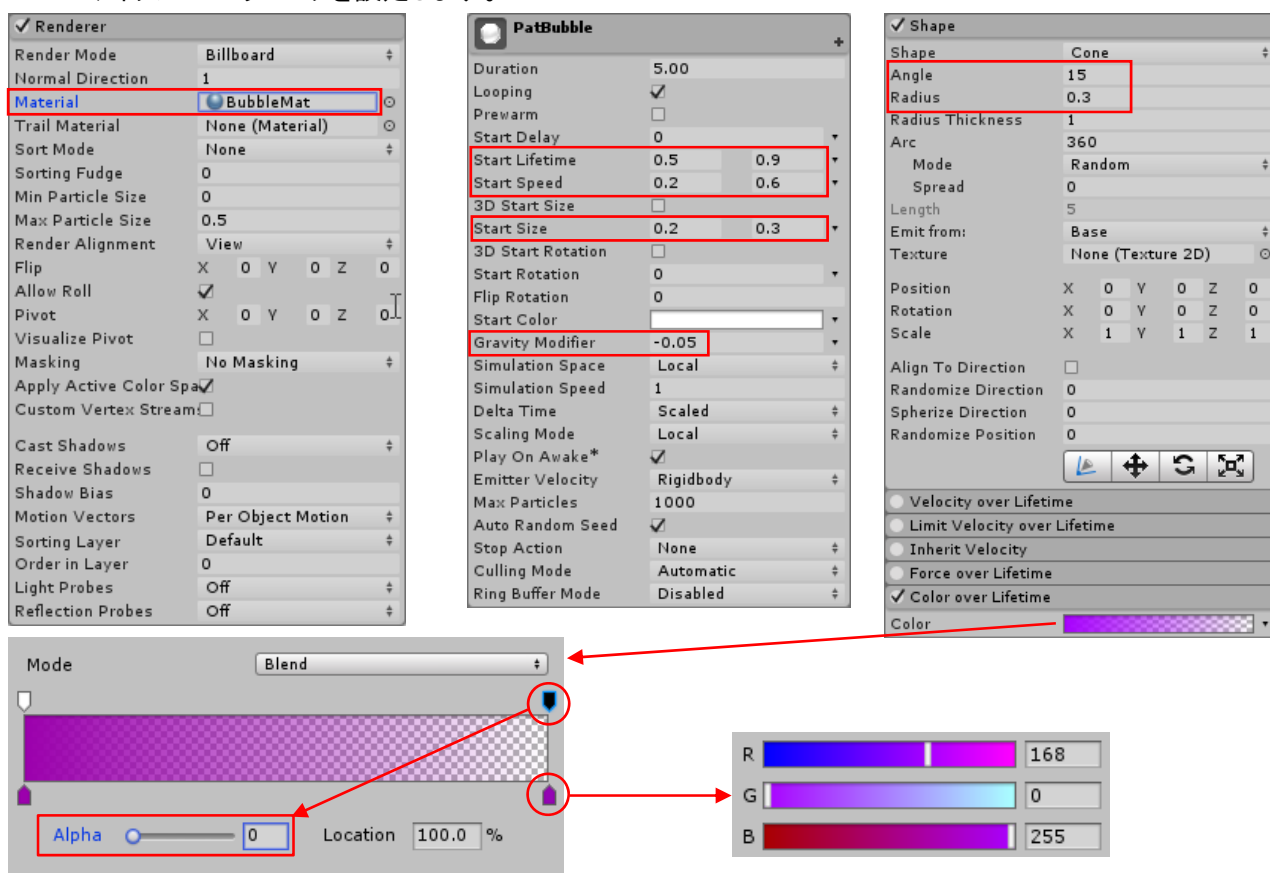
- Shader :Mobile > Particles > Additive
- Texture :tex_bubble



- ヒエラルキー欄の **Player** を右クリックし、Effects > **ParticleSystem** を選択します。
- 名称を **PatBubble** と命名します。インスペクタでパラメータを設定します。



- パーティクルのパラメータを設定します。

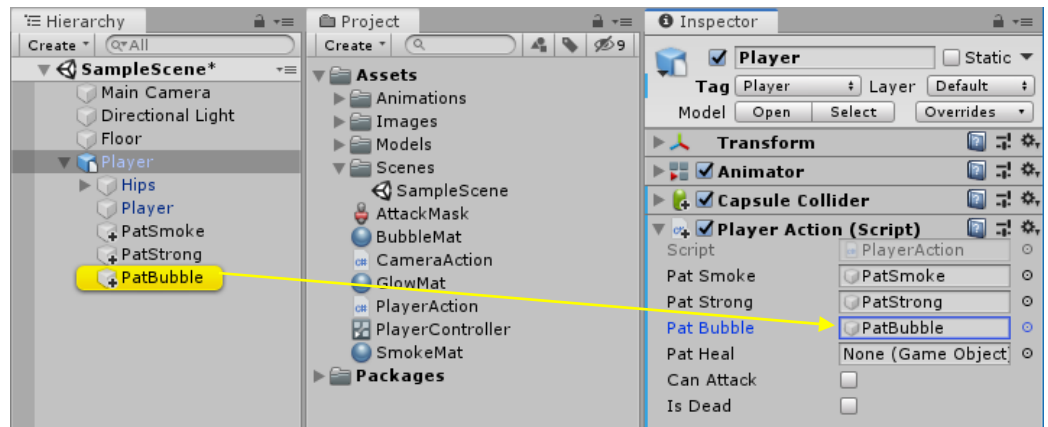


- キーボードの PageDown を押下している間だけこのエフェクトが発生するようにします。PlayerAction を編集します。

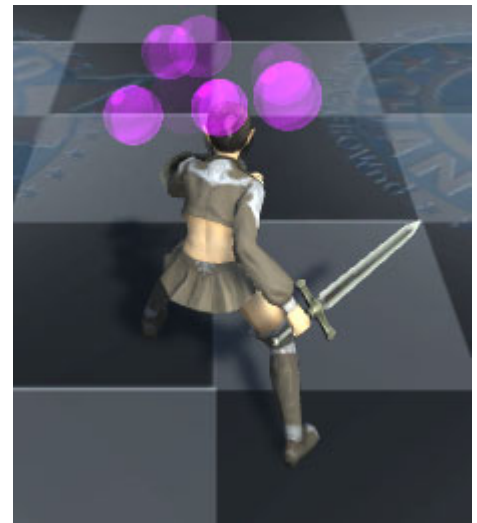
```
// 前略
void Start () {
    MyAnim = GetComponent<Animator>(); // 自身のアニメーターを取得
    SmokeMain = PatSmoke.GetComponent<ParticleSystem>().main;
    PatStrong.SetActive(false);
    PatBubble.SetActive(false);
}

void Update () {
    // PageDownキーで毒に冒された表現にする
    PatBubble.SetActive(Input.GetKey(KeyCode.PageDown));
}
// 後略
```

- ヒエラルキー欄の Player を選択し、スクリプトの PatBubble にパーティクルの PatBubble をドラッグ&ドロップします

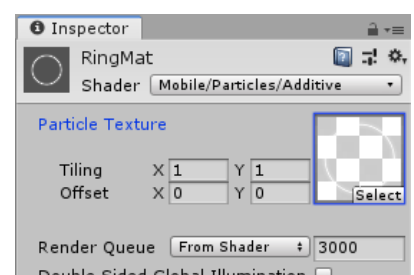
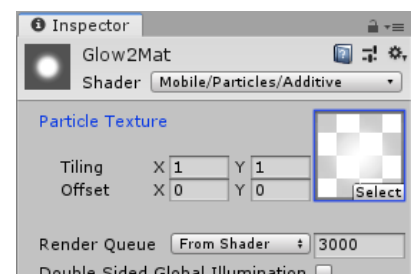
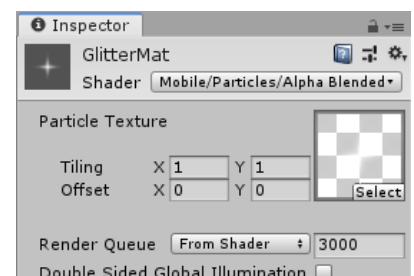


- プレイボタンを押下します。
キーボードの PageDown を押下している間だけ、毒に冒された状態のエフェクトが出るようになりました。

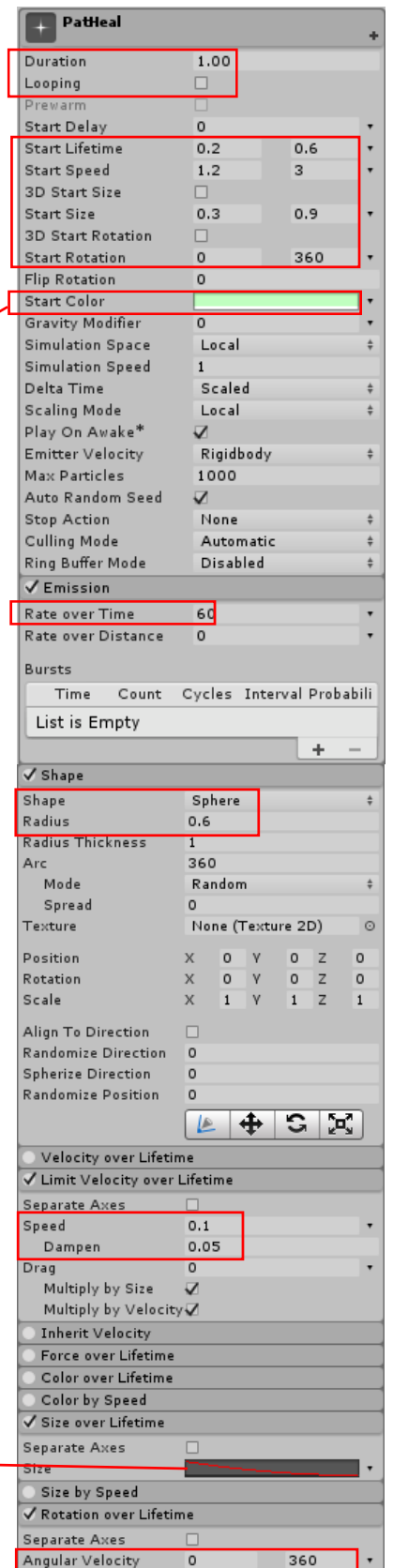
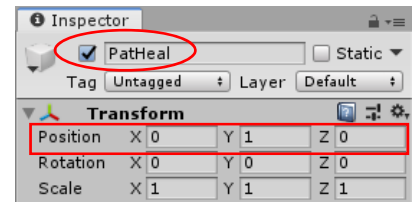
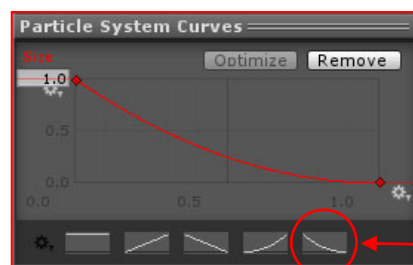
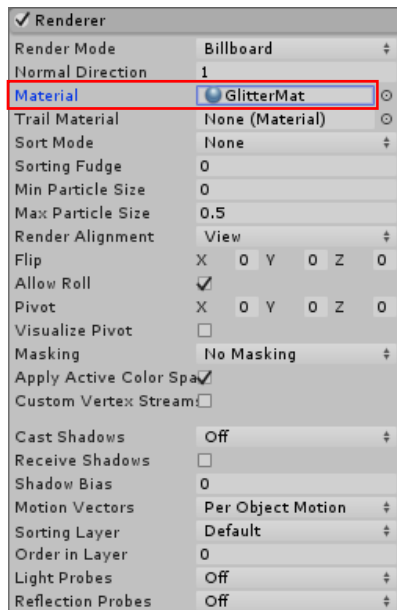


【STEP15】回復時のエフェクト例（3つのパーティクルを一気に動かします。）

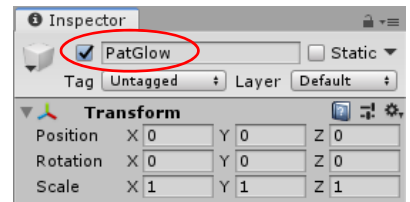
- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **GlitterMat** と命名します。
 - Shader :Mobile > Particles > Alpha Blended
 - Texture :tex_glitter
- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **Glow2Mat** と命名します。
 - Shader :Mobile > Particles > Additive
 - Texture :tex_glow2
- プロジェクト欄の Create から Material(マテリアル)を作成し、名称を **RingMat** と命名します。
 - Shader :Mobile > Particles > Additive
 - Texture :tex_ring



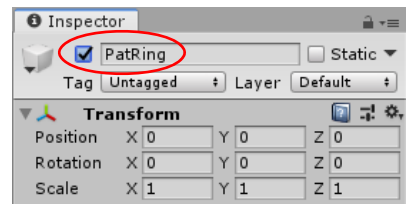
- ヒエラルキー欄の **Player** を右クリックし、Effects > **ParticleSystem** を選択します。名称を **PatHeal** と命名します。インスペクタでパラメータを設定します。
- パーティクルのパラメータを設定します。



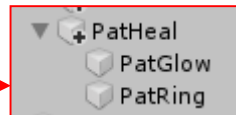
- ヒエラルキー欄の **PatHeal** を右クリックし、Effects > **ParticleSystem** を選択します。名称を **PatGlow** と命名します。



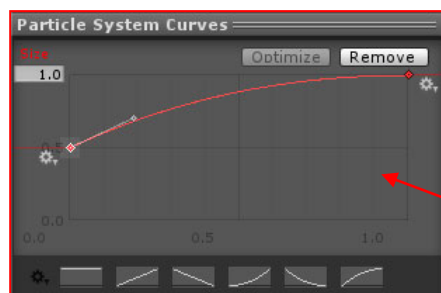
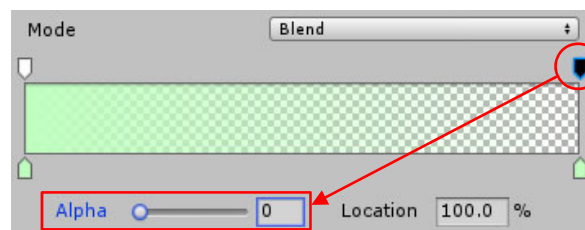
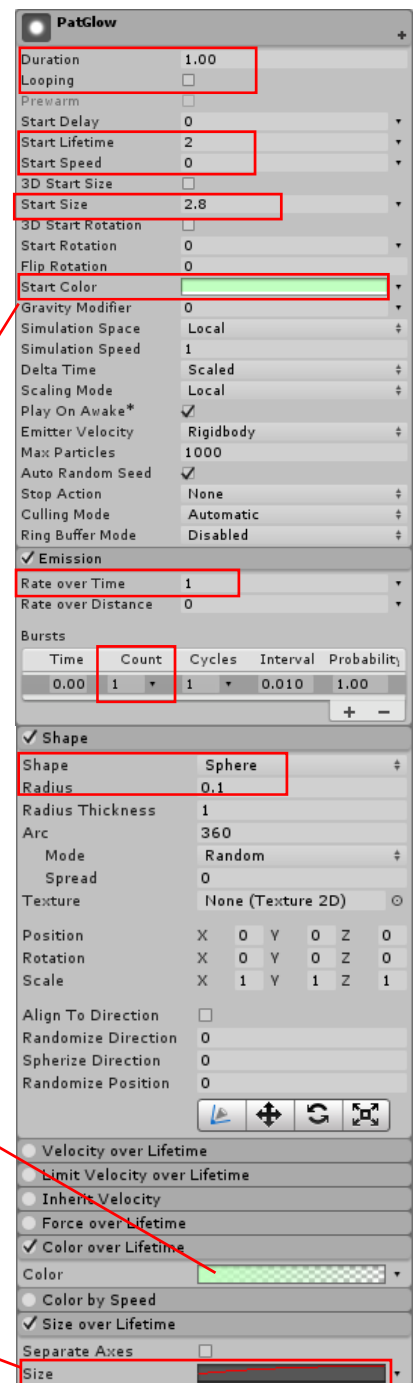
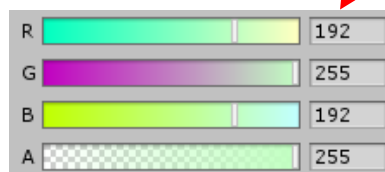
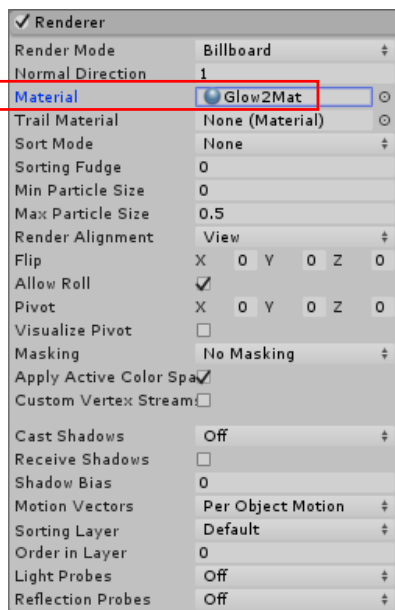
- もう一度同じ操作をし、名称を **PatRing** と命名します。



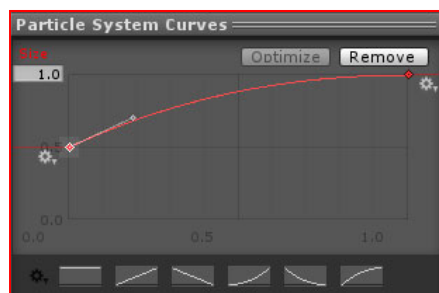
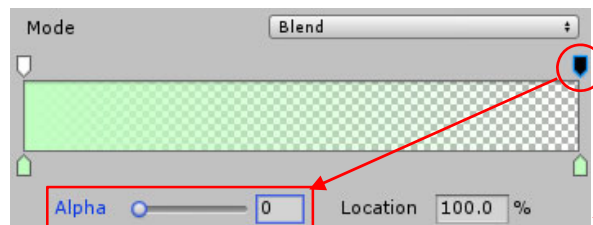
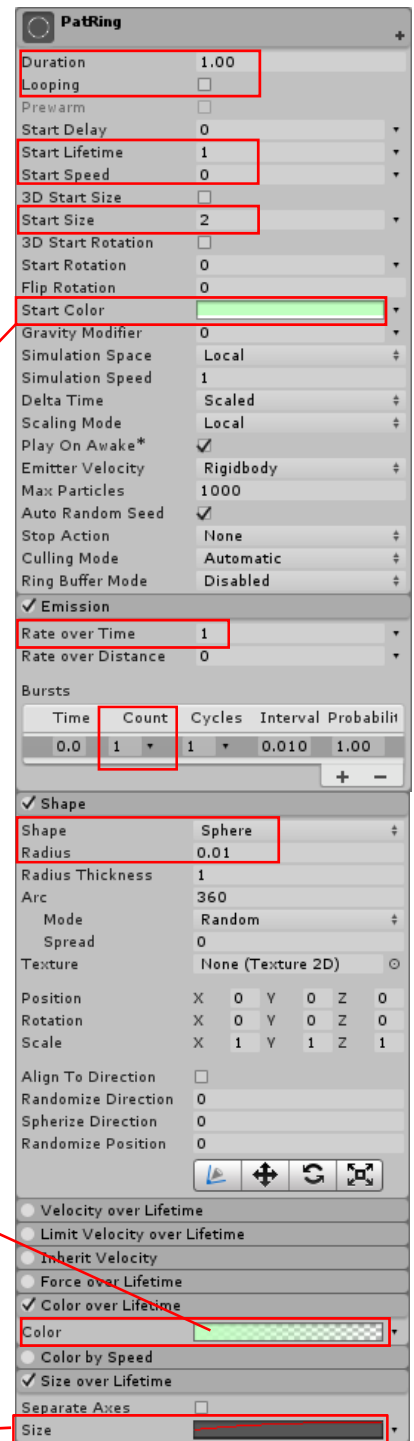
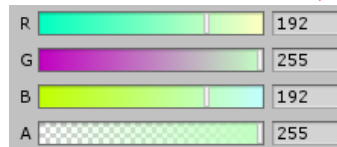
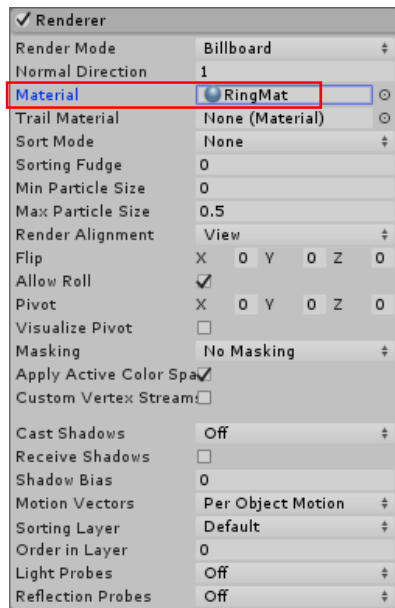
3つのパーティクルが図のような階層構造になっていることを確認します。



- **PatGlow** を選択し、インスペクタでパラメータを設定します。



- **PatRing** を選択し、インスペクタでパラメータを設定します。



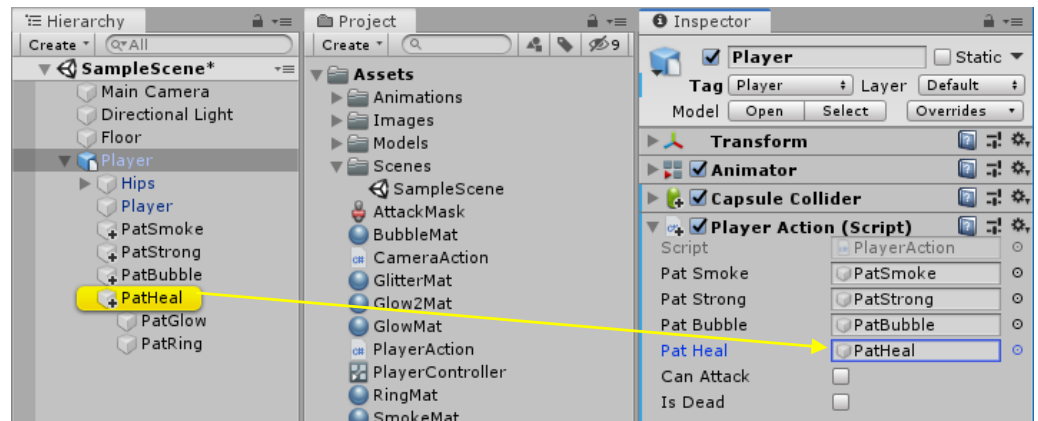
- スペースキー押下でこのエフェクトが発生するようにします。

先ほどまでのパターンと異なり、ループしないパーティクルです。プレハブとしてインスタンス生成、その後削除すればいいのですが、プレイヤーが内蔵しているので、アクティブ／非アクティブを繰り返す運用となります。スクリプト **PlayerAction** を以下のように編集します。

```
// 前略
void Start () {
    MyAnim = GetComponent<Animator>(); // 自身のアニメーターを取得
    SmokeMain = PatSmoke.GetComponent<ParticleSystem>().main;
    PatStrong.SetActive(false);
    PatBubble.SetActive(false);
    PatHeal.SetActive(false);
}

void Update () {
    // スペースキー押下で回復のエフェクトが発生する。
    if (Input.GetKeyDown(KeyCode.Space)) {
        PatHeal.SetActive(false);
        PatHeal.SetActive(true);
    }
}
// 後略
```

- ヒエラルキー欄の Player を選択し、スクリプトの **PatHeal** にパーティクルの **PatHeal** をドラッグ & ドロップします。



- プレイボタンを押下します。



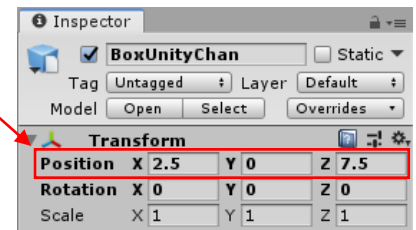
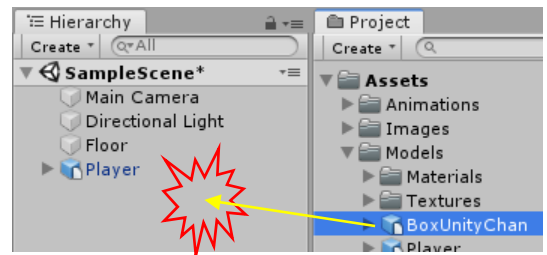
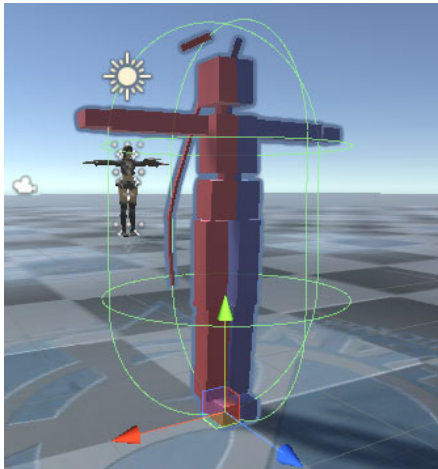
スペースキーを押下した時に、回復エフェクトが出るようになりました。



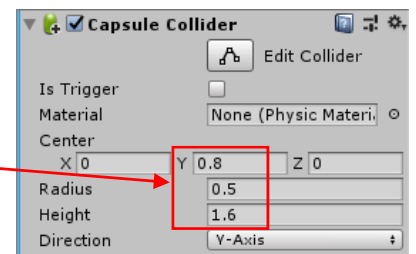
敵の運営

【STEP16】 敵の設置

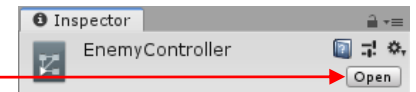
- プロジェクト欄の Models > **BoxUnityChan** をヒエラルキー欄にドラッグ&ドロップします。
インスペクタでパラメータを設定します。



- インスペクタ欄の Add Component から Physics > **CapsuleCollider** を取り付けます。
インスペクタでパラメータを設定します。

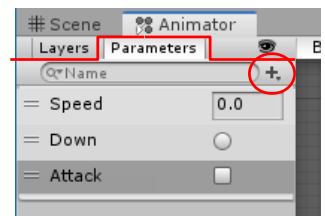


- プロジェクト欄の Create から Animator Controller (アニメーターコントローラー) を選択します。名称を **EnemyController** とします。
Open オープンボタンを押下して編集状態にします。

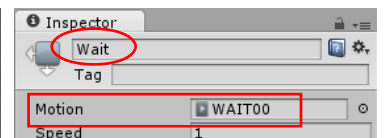
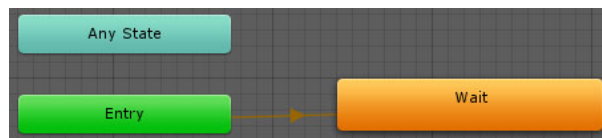


- パラメータタブを選択し、**プラスボタン(+)**を押下して、以下のパラメータを作成します。

Speed (Float 型)
Down (Trigger 型)
Attack (Bool 型)

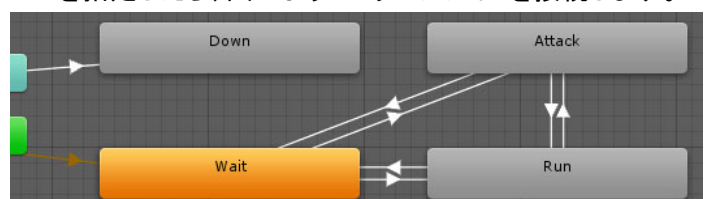


- 新しい状態を作成し、名称を **Wait** とし、Motion を **WAIT00** とします。



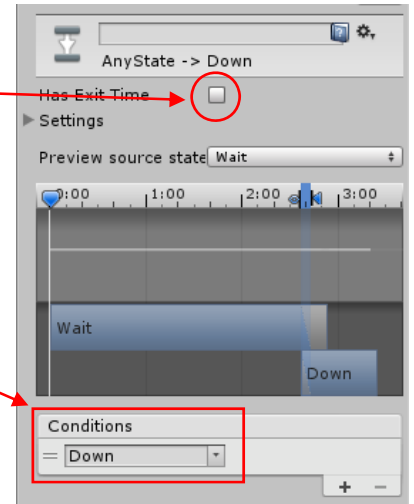
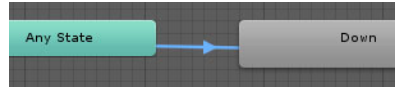
- 同様に以下の名称の状態を作成し、それぞれ Motion を指定したら、図のようにトランジションを接続します。

状態 **Run** モーション **RUN00_F**
状態 **Attack** モーション **WAIT04**
状態 **Down** モーション **Down**



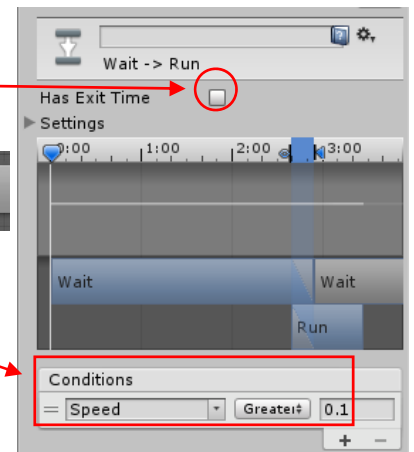
● AnyState -> Down の条件を設定します。

- Has Exit Time をオフ
- Down トリガー



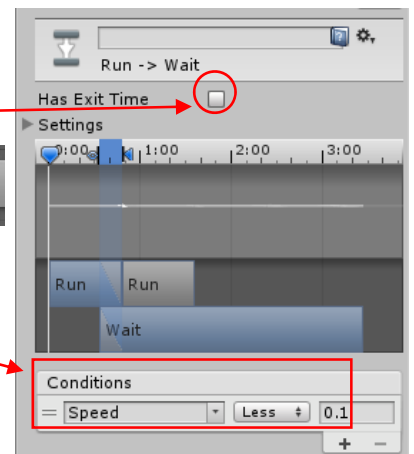
● Wait -> Run の条件を設定します。

- Has Exit Time をオフ
- Speed が 0.1 を超えたら (Greater)



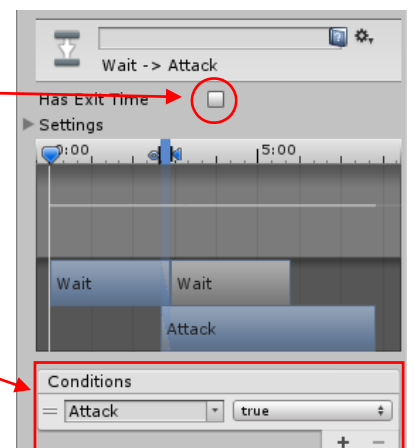
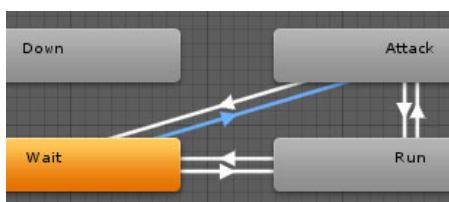
● Run -> Wait の条件を設定します

- Has Exit Time をオフ
- Speed が 0.1 を下回ったら (Less)



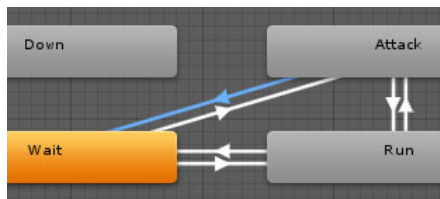
● Wait -> Attack の条件を設定します。

- Has Exit Time をオフ
- Attack が true だったら

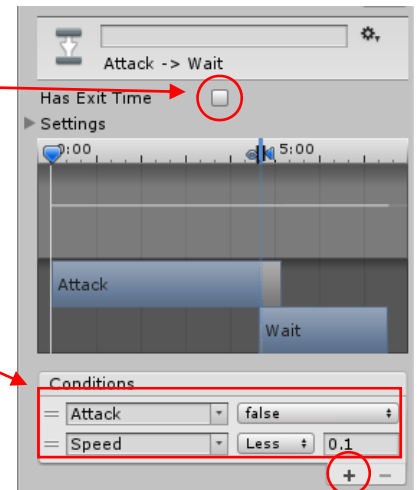


● Attack -> Wait の条件を設定します。

- Has Exit Time をオフ
- Attack が false だったら
- Speed が 0.1 を下回ったら

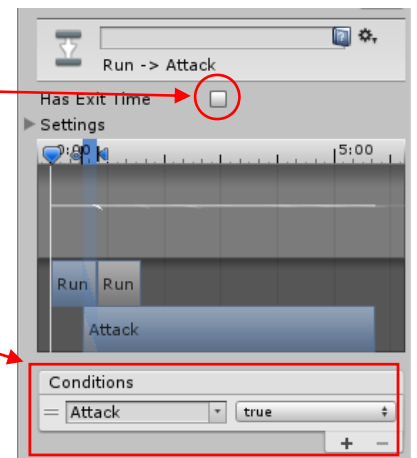


条件は2つあるので、プラスボタン(+)を押下して行を増やします。



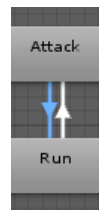
● Run -> Attack の条件を設定します。

- Has Exit Time をオフ
- Attack が true だったら

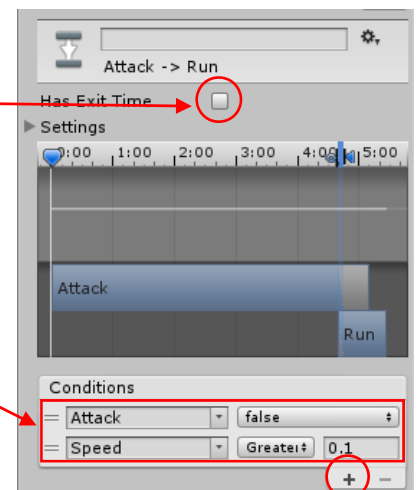


● Attack -> Run の条件を設定します。

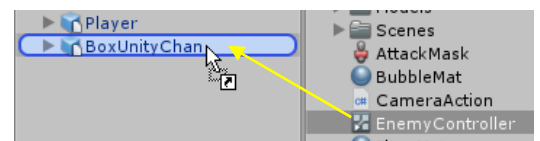
- Has Exit Time をオフ
- Attack が false だったら
- Speed が 0.1 を上回ったら



条件は2つあるので、プラスボタン(+)を押下して行を増やします。



● EnemyController をヒエラルキー欄の BoxUnityChan にドラッグ&ドロップします。



● プレイボタンを押下します。

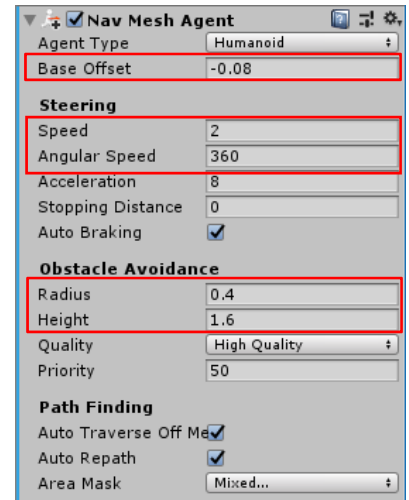
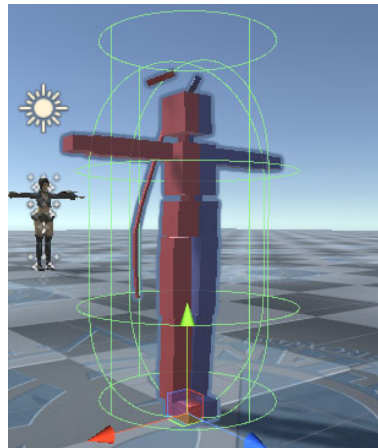


BoxUnityChan が呼吸待機することを確認します。

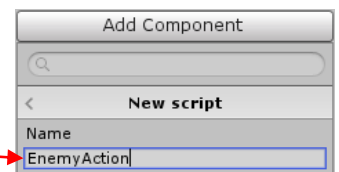
【STEP17】 ナビメッシュによる敵の運営

- ヒエラルキー欄の BoxUnityChan を選択し、インスペクタの Add Component から Navigation > Nav Mesh Agent を追加します。

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 BoxUnityChan を選択し、インスペクタの Add Component から最下段の New script を選択します。
- スクリプトの名称を EnemyAction と命名します。
- スクリプト EnemyAction を次のように編集します。



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI; //ナビメッシュの運用に必要

public class EnemyAction : MonoBehaviour {

    NavMeshAgent MyNavi; //自身のナビメッシュ
    Animator MyAnim; //自身のアニメーター
    GameObject Player; //プレイヤー
    PlayerAction PA; //プレイヤーアクション
    bool isDead = false; //自身の死亡フラグ
    public float deathTime = 3.0f; //死亡後に消えるまでの時間
    public float D; //プレイヤーとの距離

    void Start() {
        MyNavi = GetComponent<NavMeshAgent>();
        MyAnim = GetComponent<Animator>();
        //プレイヤーを見つけてPlayerActionを取得する
        Player = GameObject.FindGameObjectWithTag( "Player" );
        if (Player != null) {
            PA = Player.GetComponent<PlayerAction>();
        }
    }

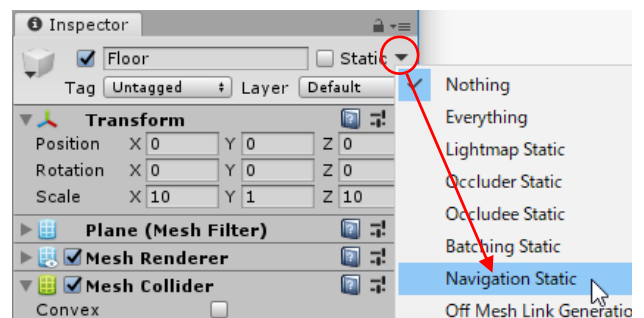
    void OnTriggerEnter(Collider other) {
        //死んでなくて、刺さったオブジェクトがタグSwordであれば
        if (other.gameObject.tag == "Sword" && !isDead) {
            isDead = true; //死亡判定
            MyAnim.SetTrigger( "Down" ); //死亡モーション
            MyNavi.enabled = false; // ナビメッシュ切る
            MyAnim.SetFloat( "Speed", 0 ); //移動はしない
            Destroy( gameObject, deathTime ); //deathTime後に撤去
        }
    }

    //続きます
```

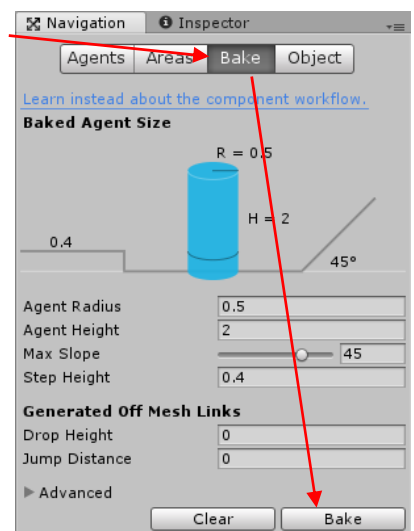

//続きです

```
void Update() {
    if (isDead || !Player) {
        return; //プレイヤーがいないか、自身が死んでたら何もしない
    }
    //プレイヤーが死んでいる
    if (PA.isDead) {
        MyNavi.enabled = false; // ナビメッシュ切る
        MyAnim.SetFloat( "Speed", 0 ); //移動はしない
        MyAnim.SetBool( "Attack", false ); //攻撃は停止
    } else {
        //プレイヤーとの距離を求める
        D = Vector3.Distance( transform.position, Player.transform.position );
        // プレイヤーとの距離が1以下になった、立ち止まって攻撃開始
        if (D <= 1) {
            MyNavi.enabled = false; //ナビ停止
            MyAnim.SetFloat( "Speed", 0 ); //移動はしない
            MyAnim.SetBool( "Attack", true ); //攻撃開始
        }
        // プレイヤーとの距離が5以下になった、追いかける
        else if (D <= 5) {
            MyNavi.enabled = true; //追いかける
            MyNavi.destination = Player.transform.position; //ターゲットを指示
            MyAnim.SetFloat( "Speed", MyNavi.velocity.magnitude ); //走行モーション
            MyAnim.SetBool( "Attack", false ); //攻撃は停止
        }
        // プレイヤーと距離が5以上なら呼吸待機
        else if (D > 5) {
            MyNavi.enabled = false; //ナビ停止
            MyAnim.SetFloat( "Speed", 0 ); //移動はしない
            MyAnim.SetBool( "Attack", false ); //攻撃は停止
        }
    }
}
```

- ヒエラルキー欄の **Floor** を選択し、インスペクタの **Static▼**を押下し、**Navigation Static**を選択します。



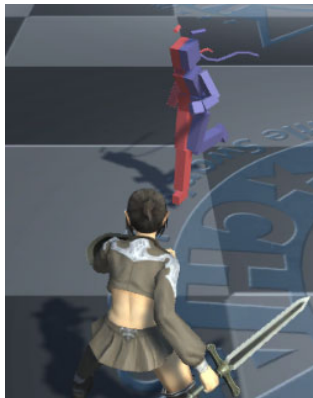
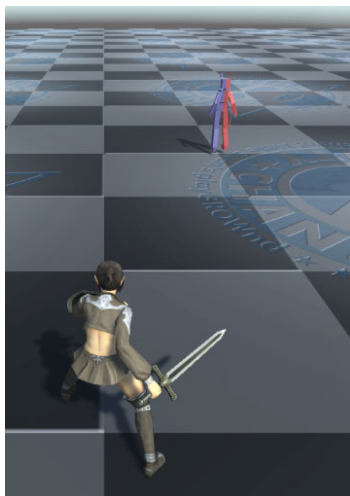
- メニューWindow から AI > **Navigation** を選択し、**Bake** タブから **Bake** を押下します。



- プレイボタンを押下します。



- ① プレイヤーと敵との距離が十分にあるので、敵は呼吸待機をすることを確認します。
- ② プレイヤーを操作して敵に近づき、距離が縮まったら敵はプレイヤーに向かって走ってくることを確認します。
- ③ 敵が至近距離に位置したら、近付く行動を止めて攻撃モーションを行うことを確認します。



- ④ 敵が攻撃中にプレイヤーが逃げたら、攻撃を止めてまた追いかけてくることを確認します。



- ⑤ プレイヤーが敵から逃げ切ったら、敵はまた呼吸待機のモーションに戻ることも確認します。
- ⑥ マウスクリックで敵を斬り付けると、敵は死亡モーションを行い、一定時間が経過したらシーンから撤去されることを確認します。

何らかの音や獲得ポイント、日本で許される程度の血しぶきなど、パーティクルで生成可能な視覚効果があった方がプレイヤーには判りやすいでしょう。



【STEP18】 攻撃可能期間の設定

既に気付いていると思われますが、攻撃行動しなくても、持っている剣が少しでも敵に触れたら、敵はダウンしてしまいます。これは改善すべき状況です。

剣を高く掲げ、振り下ろし行動に入った瞬間に攻撃可能期間とし、振り下ろした最大値で攻撃期間終了とする考え方とします。

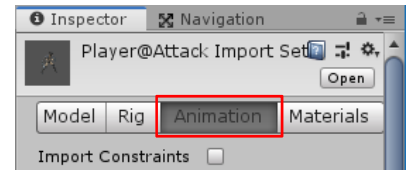
アニメーションの時間軸にイベントを発生させて攻撃可能の真偽値をオン・オフさせます。



- スクリプト **PlayerAction** に次の処理を追加します。

```
void AttackStart() {
    CanAttack = true;
}
void AttackFinish() {
    CanAttack = false;
}
```

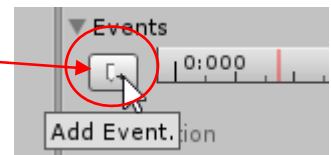
- プロジェクト欄の Models > **Player@Attack** を選択し、インスペクタを **Animations タブ** にします。
下部の ▼ **Events** 欄を開いておきます。



- **タイムスライダ**を動かして **0.20 秒**の位置にします。

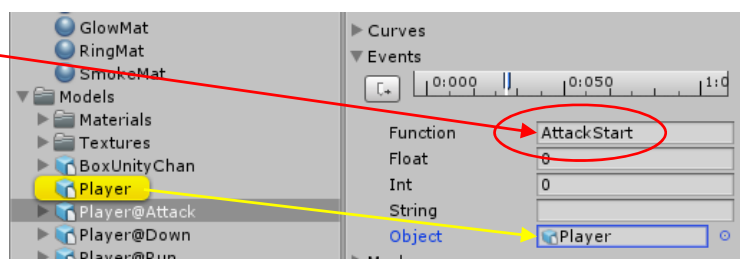


- Events 欄のボタン **AddEvent**を押下します。



- 処理関数の名称を **AttackStart** とします。

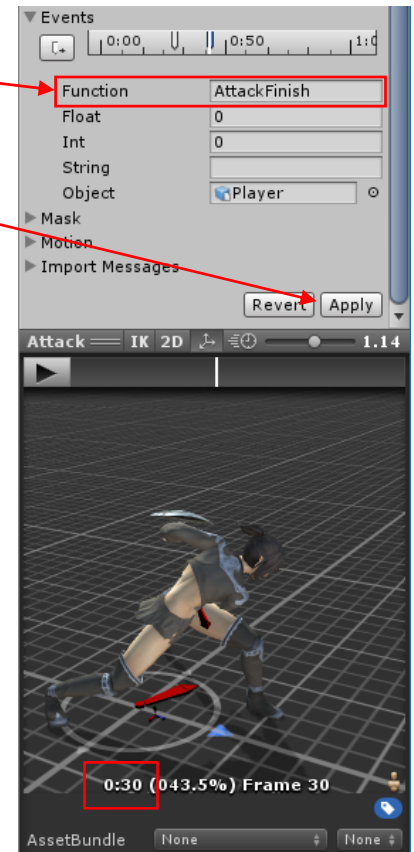
伝達するオブジェクトを Models > **Player** に設定します。



アニメーションのタイムラインが、処理のトリガーを発信するようになります。CanAttack が true になります。

- 同様にして 0.30 秒の位置にイベント **AttackFinish** を作成します。

- 設定が終われば **Apply** ボタンを必ず押下します。



- スクリプト **EnemyAction** を次のように修正します。

```
//～前略～

void OnTriggerEnter(Collider other) {
    //死んでなくて、刺さったオブジェクトがタグSwordで、プレイヤーがアタック期間中であれば
    if (other.gameObject.tag == "Sword" && !isDead && PA.CanAttack && !PA.isDead ) {
        isDead = true; //死亡判定
        MyAnim.SetTrigger( "Down" ); //死亡モーション
        MyNavi.enabled = false; // ナビメッシュ切る
        MyAnim.SetFloat( "Speed", 0 ); //移動はしない
        Destroy( gameObject, deathTime ); //deathTime後に撤去
    }
}

//～後略～
```

- プレイボタンを押下します。



剣を振り下ろしている間だけ攻撃が有効になっていることを確認します。

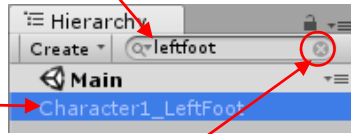


【STEP19】 プレイヤーの死亡を実現する

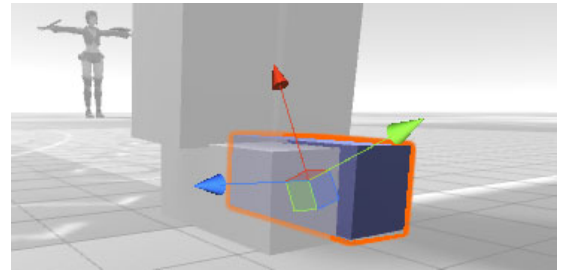
敵は武器を持っていないので、キック・モーションを利用して足首に攻撃判定を持たせます。

- ヒエラルキー欄の検索欄に **leftFoot** と入力します。

対象オブジェクトが検出されるので、これを選択します。

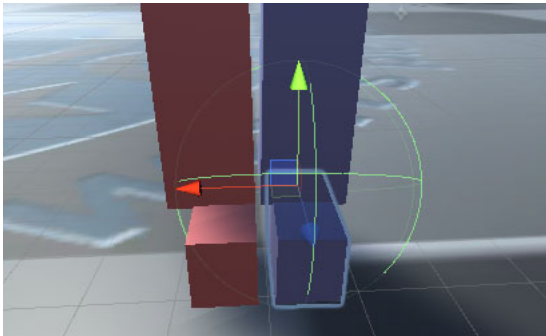


クローズ(X)を押下して選択は解除しておきます。

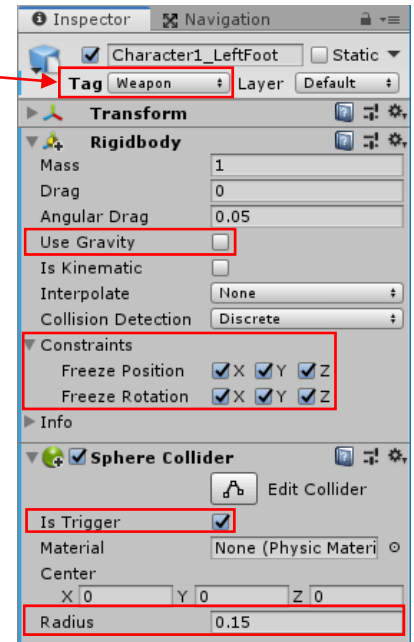


- タグを **Weapon** にします。

- インスペクタの Add Component から Physics > **Rigidbody** を取り付けます。インスペクタでパラメータを設定します。



同様に Add Component から Physics > **Sphere Collider** を取り付けます。インスペクタでパラメータを設定します。




- スクリプト **PlayerAction** を修正します。

```
//～前略～

void OnTriggerEnter(Collider other) {
    //死んでなくて、刺さったオブジェクトがタグWeaponであれば
    if (other.gameObject.tag == "Weapon" && !isDead) {
        isDead = true; //死亡判定
        MyAnim.SetTrigger( "Down" ); //ダウンモーション発動
    }
}

void Update() {
    if (isDead) {
        return; //自身が死んでたら何もしない
    }
}

//～後略～
```


- プレイボタンを押下します。
敵がキックをしてプレイヤーが倒れることを確認します。

プレイヤーが倒れる際に剣が敵に当たる場合でも、プレイヤーの死亡が成立しているので敵は無反応となります。

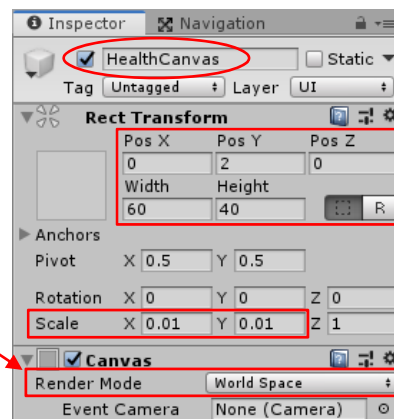
敵は攻撃距離にあっても、相手が死亡しているので、キック行動は行いません。



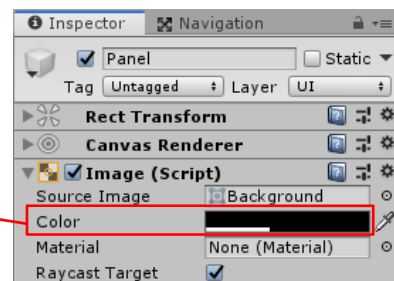
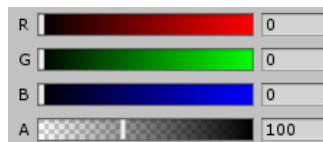
【STEP20】 プレイヤーのヘルスを管理する

互いに、たった一度の武器ヒットでダウンするのも問題でしょう。ヘルス(もしくはライフ)が減って、ゼロになった時点で死亡する仕組みを実装します。ヘルスは棒グラフで表現されるのが一般的です。

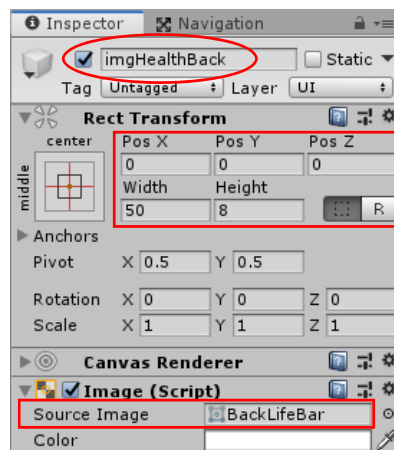
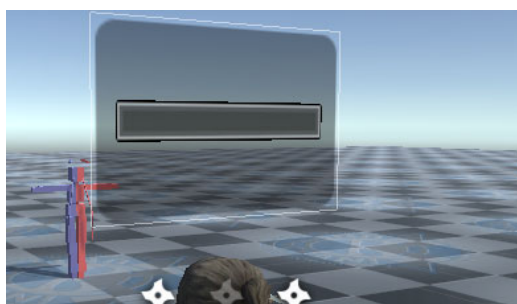
- ヒエラルキー欄の Player を右クリックし、UI > **Panel** を選択します。
- Player 配下に作成された **Canvas** を選択し、インスペクタでパラメータを設定します。
真っ先に RenderMode を **WorldSpace** にします。



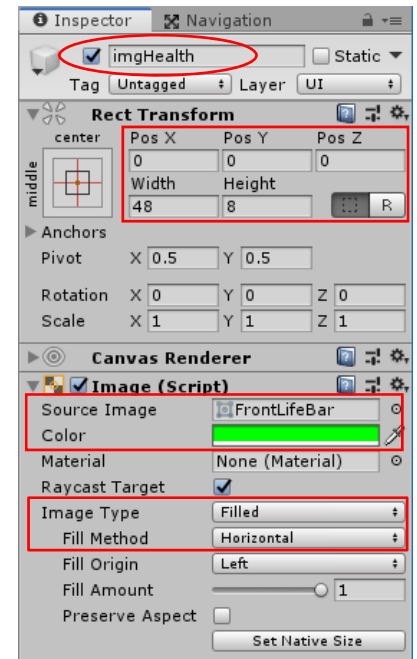
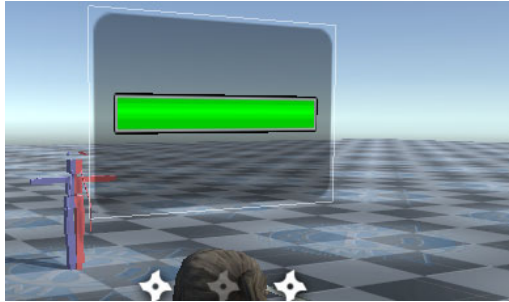
- ヒエラルキー欄の HealthCanvas > **Panel** を選択し、インスペクタでパラメータを設定します。



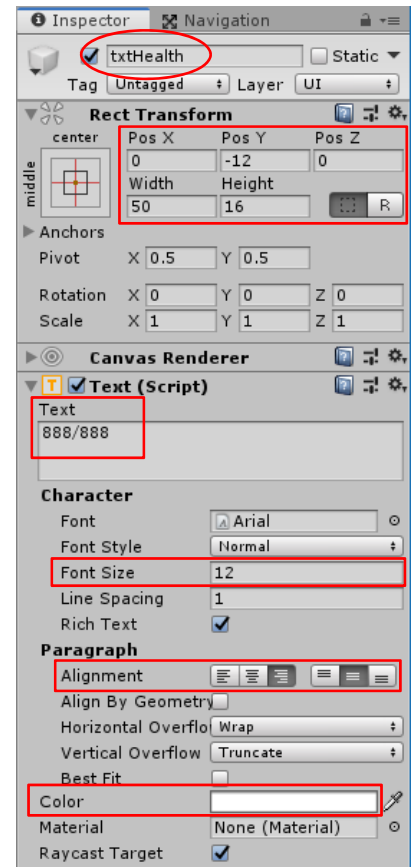
- ヒエラルキー欄の HealthCanvas を右クリックし、UI > **Image** を選択します。名称を **imgHealthBack** とします。
インスペクタでパラメータを設定します。



- この imgHealthBack を複製 (Ctrl + D) し、名称を **imgHealth** とします。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 HealthCanvas を右クリックし、UI > **Text** を選択します。名称を **txtHealth** とします。インスペクタでパラメータを設定します。



- スクリプト **PlayerAction** を次のように修正します。

```
//～前略～

Transform myCanvas; //自身のCanvas
Image imgHealth; //ヘルスバー
Text txtHealth; //ヘルス文字
public int MaxHealth = 100; //ヘルスの最大値
int Health; //自身のヘルス値

//続きます
```

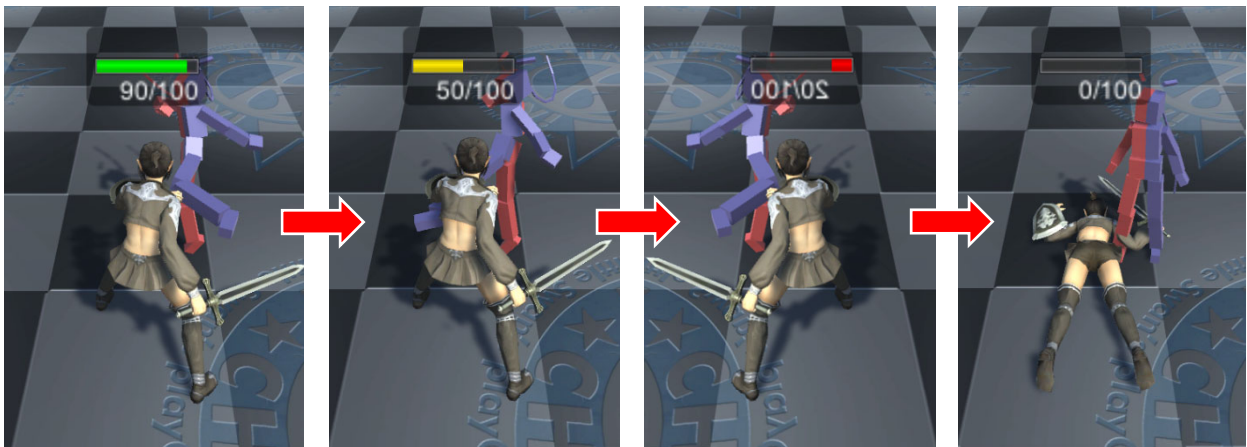
//続きです

```
void Start() {  
    //自身のヘルス表示を取得  
    myCanvas = transform.Find( "HealthCanvas" );  
    imgHealth = myCanvas.transform.Find( "imgHealth" ).GetComponent<Image>();  
    txtHealth = myCanvas.transform.Find( "txtHealth" ).GetComponent<Text>();  
    Health = MaxHealth; //ヘルスを最大にする  
    MyAnim = GetComponent<Animator>(); // 自身のアニメーターを取得  
    SmokeMain = PatSmoke.GetComponent<ParticleSystem>().main;  
    PatStrong.SetActive(false);  
    PatBubble.SetActive( false );  
    PatHeal.SetActive( false );  
}  
  
void OnTriggerEnter(Collider other) {  
    //死んでなくて、刺さったオブジェクトがタグWeaponであれば  
    if (other.gameObject.tag == "Weapon" && !isDead) {  
        Health -= 10;  
        if (Health <= 0) {  
            isDead = true; //死亡判定  
            MyAnim.SetTrigger( "Down" ); //ダウンモーション発動  
        }  
    }  
}  
  
void LateUpdate() {  
    //ヘルスバーを増減して色を決定  
    imgHealth.fillAmount = Health / ( float ) MaxHealth;  
    if (imgHealth.fillAmount > 0.5f) {  
        imgHealth.color = Color.green;  
    } else if (imgHealth.fillAmount > 0.2f) {  
        imgHealth.color = Color.yellow;  
    } else {  
        imgHealth.color = Color.red;  
    }  
    //ヘルス値を表示  
    txtHealth.text = Health.ToString( "f0" ) + "/" + MaxHealth.ToString( "f0" );  
    //常にキャンバスをカメラに向ける  
    myCanvas.forward = Camera.main.transform.forward;  
}  
  
//～後略～
```

- プレイボタンを押下します。



攻撃される都度、ライフが減る様子がゲージに表示され、ライフがゼロに至るとダウンのアクションになります。キャラクターの名前や職業アイコン、所持アイテム、残り弾数などを表示するUIとしても扱えます。



以上