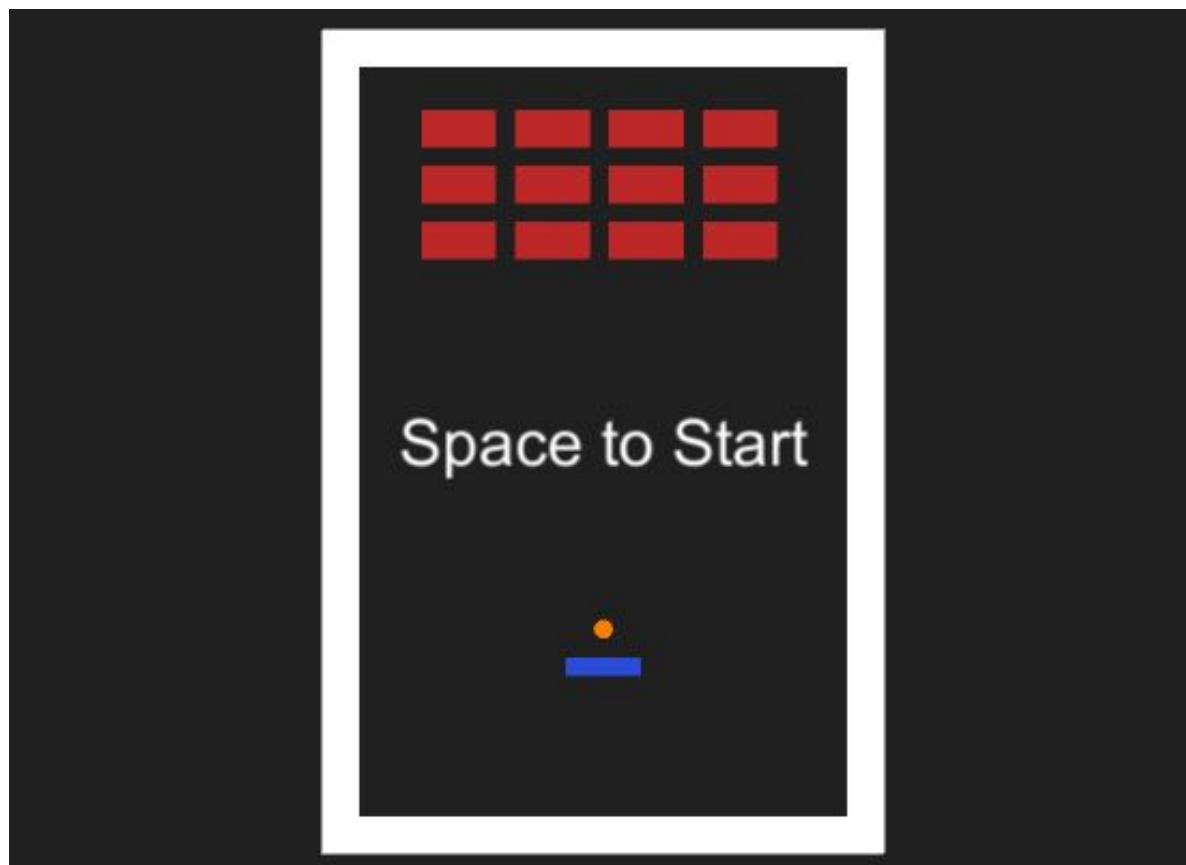


# ブロック崩し2D or 3D

---

# 作るゲーム

---



# 作成スタート

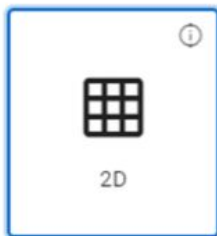
---

# プロジェクトのテンプレートは2Dを選択

Unity 2019.3.8f1 で新しいプロジェクトを作成

— □ ×

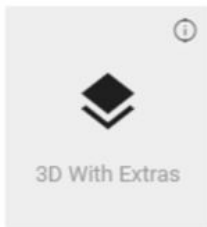
テンプレート



2D



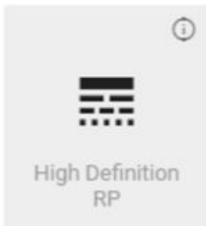
3D



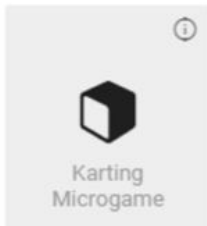
3D With Extras



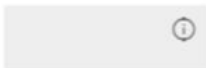
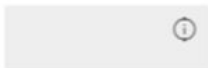
FPS Microgame



High Definition  
RP



Karting  
Microgame



設定

プロジェクト名 \*

Tutorial

保存先 \*



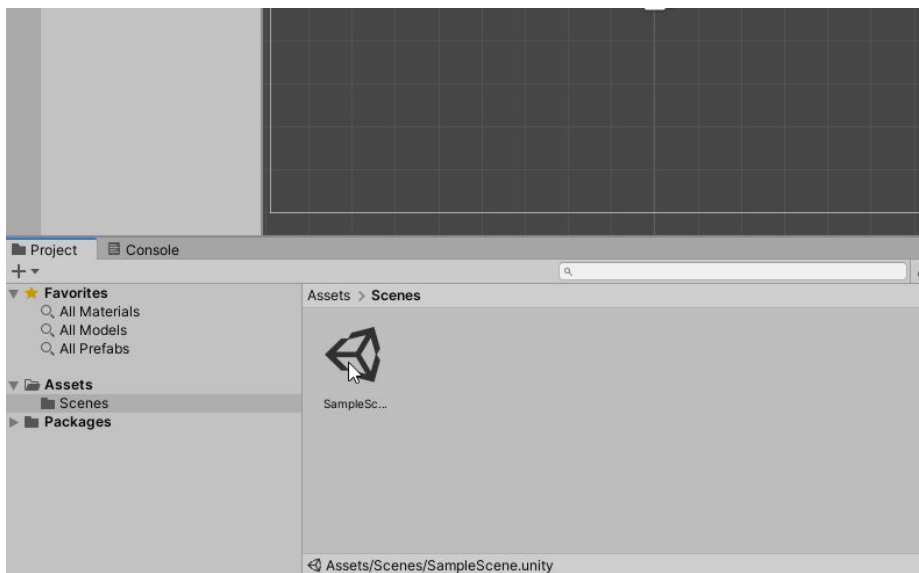
キャンセル

作成

# シーン名「Play」

プロジェクトを作成するとSample Sceneというシーンが存在するのですが、この名前だと何のシーンなのか判断できないので、プレイ画面という意味を込めてPlayにシーン名を変更します。

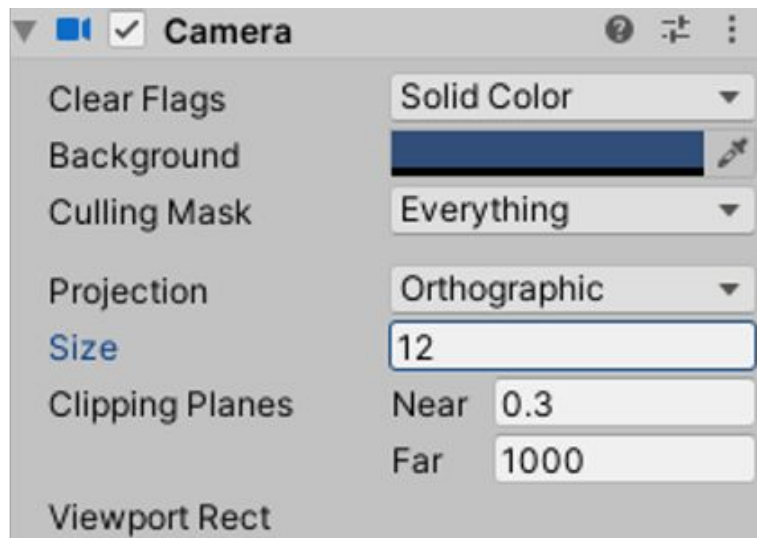
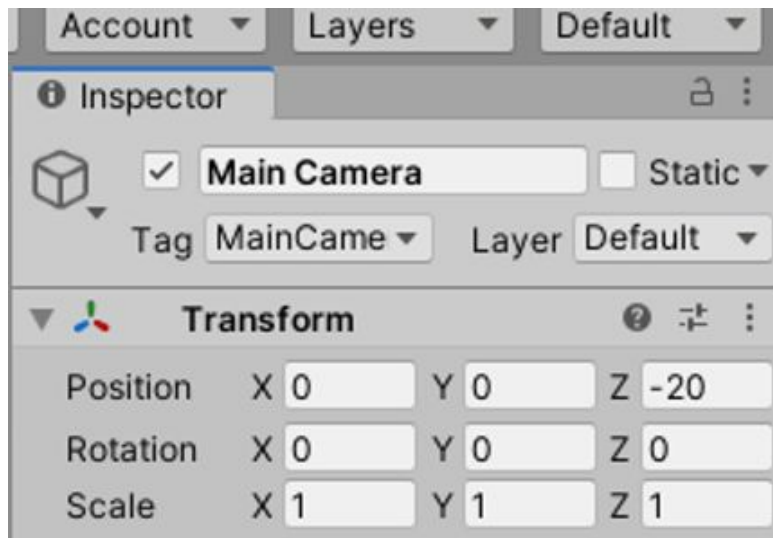
途中でダイアログが出ますが、Reloadで問題ないです。



# カメラの設定を行う

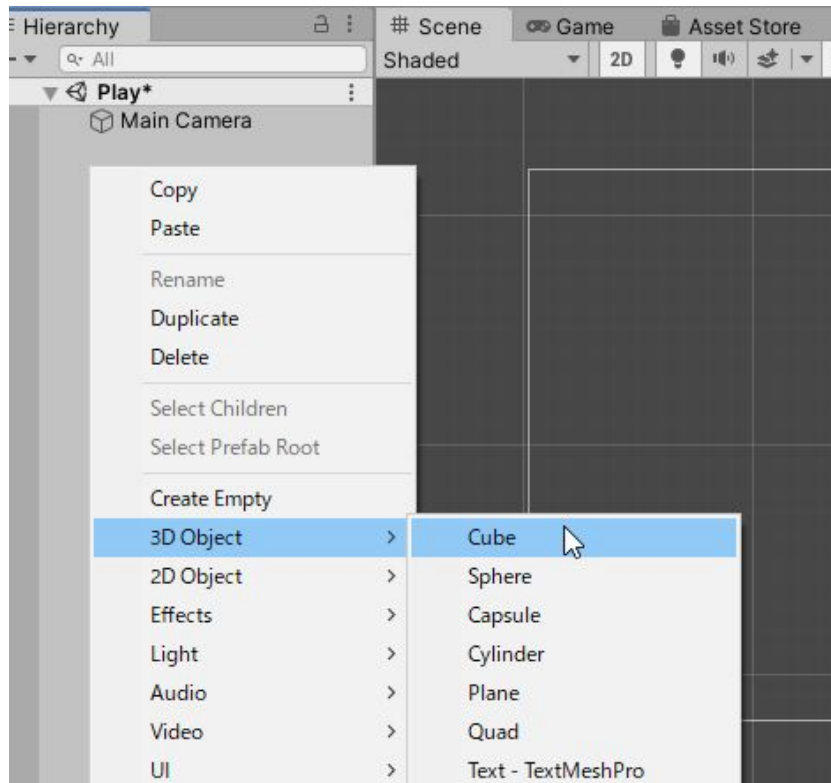
初期状態だとカメラの表示範囲が狭いので、調整します。TransformコンポーネントのPositionの値を(0, 0, -20)に設定してカメラを位置を変更します。そして、表示範囲を広くするために、CameraコンポーネントのSizeの値を12にします。

背景の色はCameraコンポーネントのBackgroundで変更できます。お好みで。



# 壁を作成する

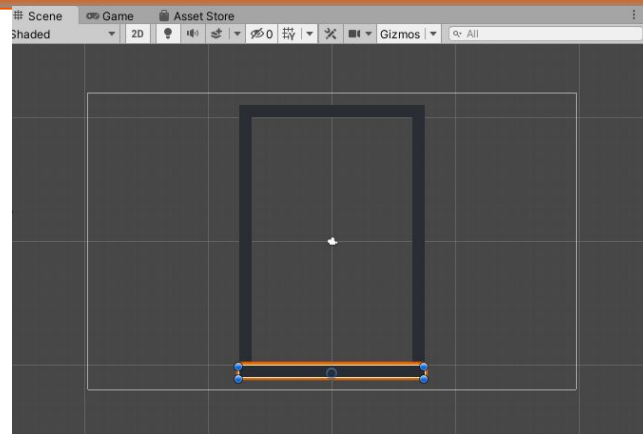
まず、壁を作成します。Hierarchyビューで右クリックをしてCubeを作成します。





Cubeを4つ作成して大きさと位置を変えて上下左右の壁を作りましょう。次の値を用いました。

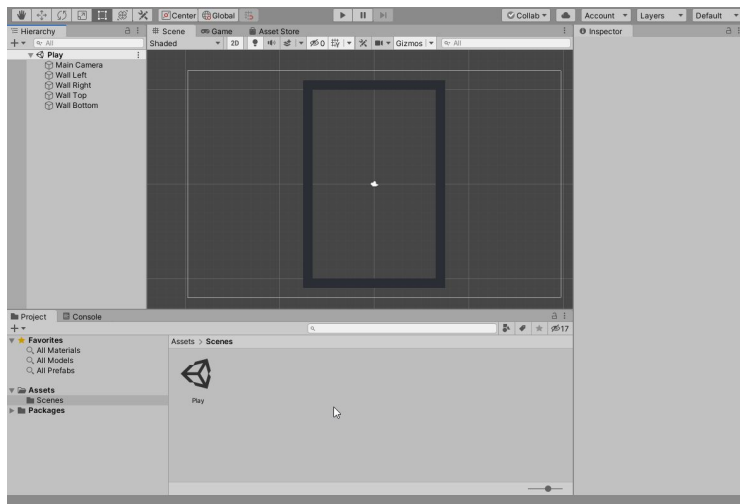
- 上下の壁のScaleは(15, 1, 1)
- 左右の壁のScaleは(1, 20, 1)
- 位置は以下の表の通り



名前	x	y	z
Wall Left	-7	0	0
Wall Right	7	0	0
Wall Top	0	10.5	0
Wall Bottom	0	-10.5	0

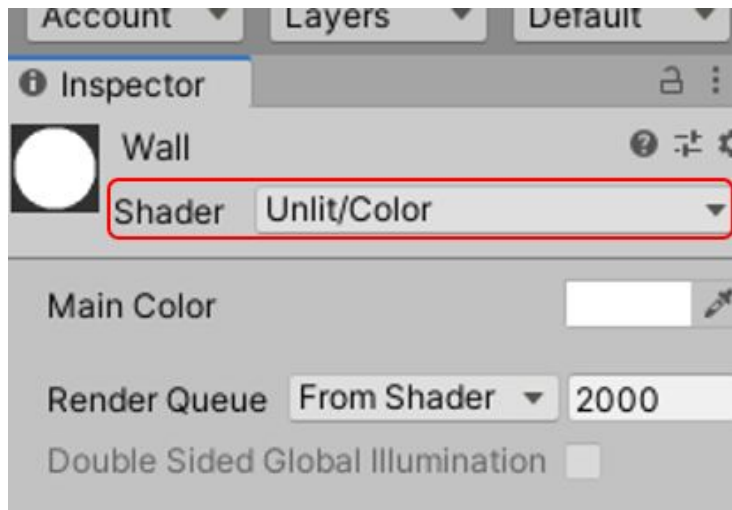
現段階だと、壁は暗く表示されています。これはシーン上にライトが存在しないためです。ライトを作成してもいいのですが、今回はライトなしで単色に表示するマテリアルを作成します。

これからいろいろなファイルを作っていくため、整理しやすいようにMaterialsフォルダを作り、その中に「WallMaterial」を作成しました。今後マテリアルを作成する場合はこのフォルダの中に作るといいでしょう。



マテリアルのShaderの箇所をUnlitのColorに変更してください。Unlitは「アンライト」という意味であり、つまり、ライトに関係なくゲームオブジェクトを描画する設定です。これで好きな色で表示できるようになります。

作成したマテリアルを4つの壁に適用すれば、壁は完成です。(マテリアルの適用はマテリアルをSceneビューかHierarchyビューにあるゲームオブジェクトにドラッグ&ドロップすれば可能です。)



# 2Dゲームで3D Objectを使用する理由

---

今回作成するのは2Dゲームですが、2D Objectを使用する場合は画像ファイルを用意する必要があります。(今回でいえば、円の画像を用意しなければなりません。)

楽をするために今回は3D ObjectのCubeとSphereを四角と円として使用することにします。カメラのProjectionの設定が**Orthographic(平行投影)**になっていれば、遠いもの/近いものを同じ大きさで表示するので2Dに見せることができます。

# ボールを作成する

---

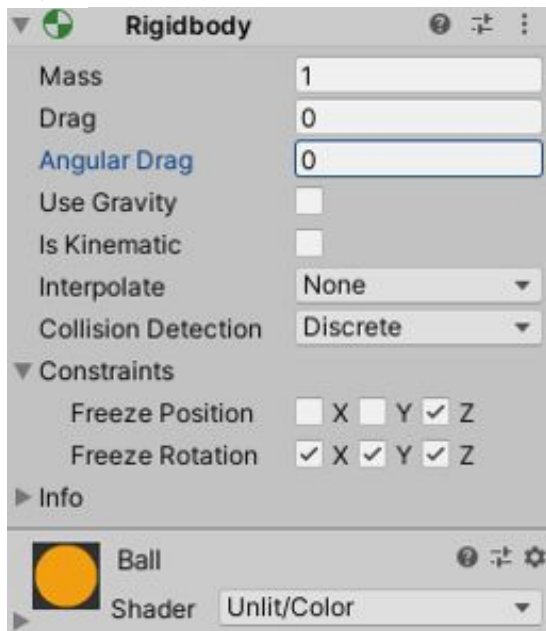
# 位置、大きさ、マテリアルの設定

---

続いて、ボールを作成します。3D ObjectのSphereを作成し、名前「**Ball**」にします。位置はとりあえず(0, 0, 0)で良いでしょう。Scaleは(1, 1, 1)だとすこし大きいので、(0.5, 0.5, 0.5)にします。また、壁と同様にマテリアルを作成し、好きな色に設定しましょう。

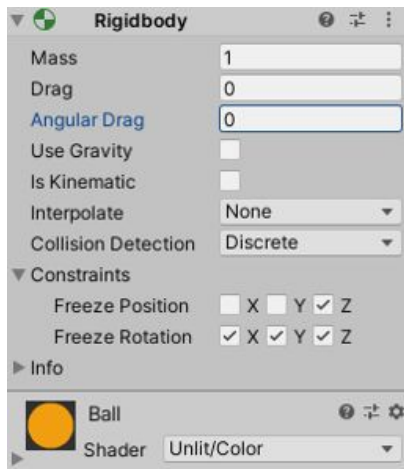
# Rigidbodyの設定

ボールは壁やブロック、プレイヤーとの衝突によって跳ね返る必要があります。このような挙動は物理演算によって実現するのが簡単です。そこで、BallにRigidbodyコンポーネントをアタッチします。(Inspectorビューの下部にあるAdd ComponentからPhysics→Rigidbodyを選択する。)



Rigidbodyでは質量(Mass)や重力を使用するかどうか(Use Gravity)、空気抵抗(Drag, Angular Drag)、制約条件(Constraints)などが設定できます。ここでの設定は以下の通りです。

- 空気抵抗は無しにするので、DragとAngular Dragは0にする。
- 重力は使用しないので、Use Gravityはoffにする。
- 位置はz方向に動いて欲しくないので、Constraintsのpositionのzをonにする。
- 回転して欲しくないので、Constraintsのrotationの全てをonにする。

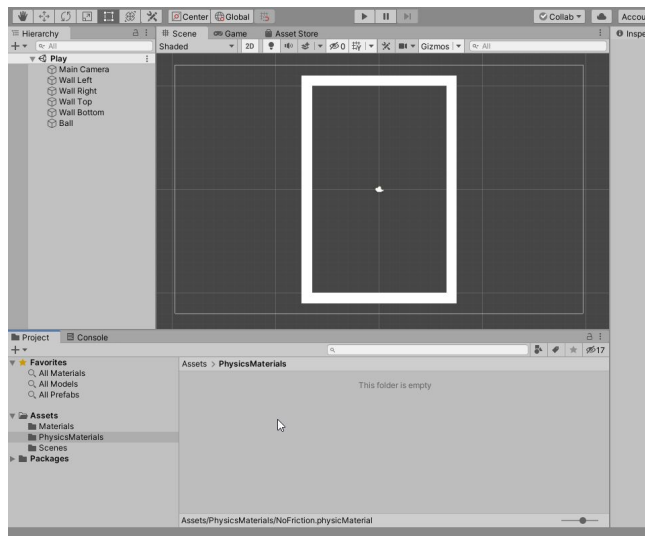




# Physics Materialの設定

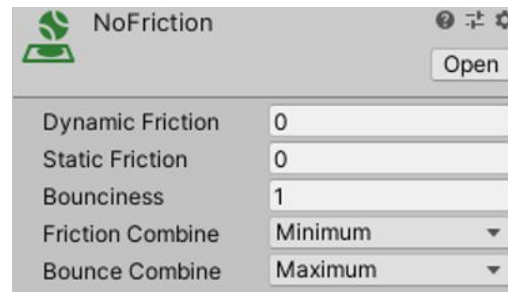
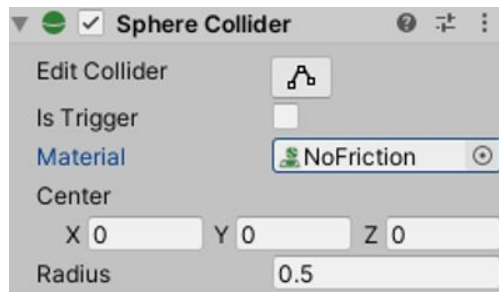
物理演算の設定はもう一つあり、Physics Materialで行います。これは、他の物体と衝突したときにどのくらい摩擦が生じるか(摩擦係数)、どのくらいの強さで跳ね返るのか(反発係数)を指定するものです。

Projectビューで右クリックしてCreate->Physic Materialを選択して新規作成します。(画像では分類のため、PhysicsMaterialsフォルダを作成し、その中に作成しています。)



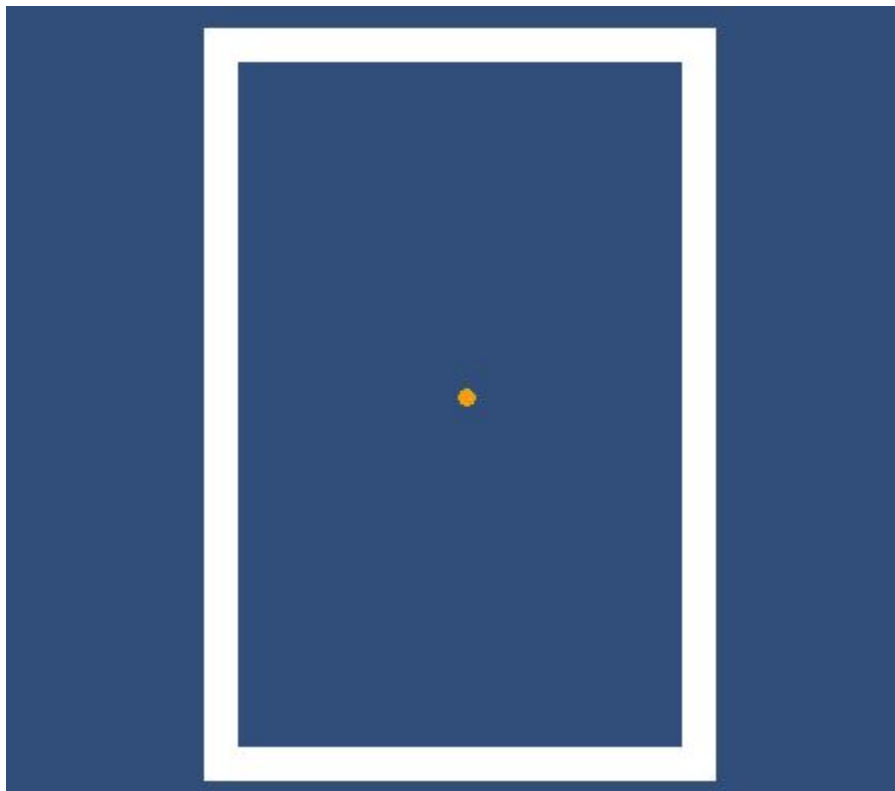
作成したPhysics Materialの名前は「NoFriction」としました。摩擦をなくすために**Dynamic/Static Frictionを0**にし、減速して跳ね返ることがないように**Bouncinessを1**にします。Friction/Bounce Combineは2つの物体がぶつかったときに摩擦/反発係数をどう計算するかを設定する項目です。Minimum/Maximumは2つのゲームオブジェクトに設定された2つのPhysics Materialのうち小さい/大きい値を採用する設定です。したがって、**Friction CombineをMinimumにすれば、常にFrictionが0で計算されるようになります。同様の理由からBounce CombineはMaximumを選び、常にBouncinessを1として計算するようにします。**

作成したNoFrictionはBallの衝突判定処理を担っている**Sphere ColliderのMaterialに設定**してください。



# スクリプトの作成

Ballスクリプトを作成し、ボールを動かしてみましょう。以下のように作成し、Ballゲームオブジェクトにアタッチしてください



# Ball.cs

```
using UnityEngine;
class Ball : MonoBehaviour
{
    [SerializeField] float speed = 5f; // ボールの移動の速さを指定する変数
    Rigidbody _myRigidbody;
    void Start()
    {
        // Rigidbodyにアクセスして変数に保持しておく
        _myRigidbody = GetComponent<Rigidbody>();
        // 右斜め45度に進む
        _myRigidbody.velocity = new Vector3(speed, speed, 0f);
    }
}
```

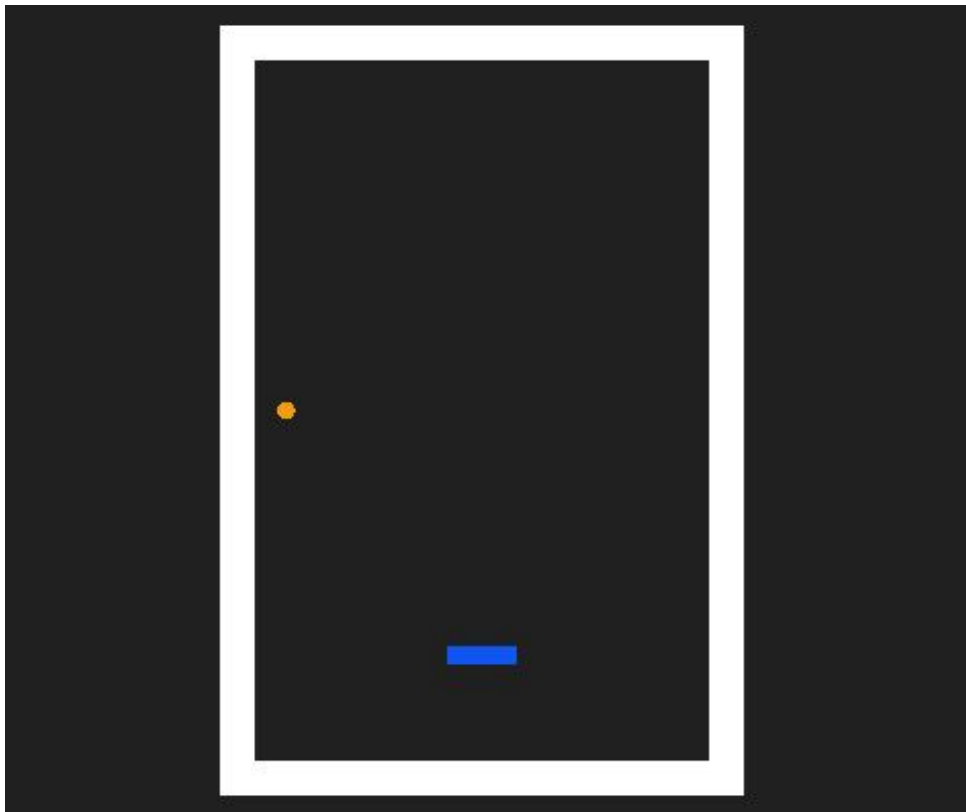
StartメソッドでRigidbodyコンポーネントにアクセスして速度velocityを設定することでボールを動かします。x成分、y成分を同じにしているため、斜め45度で進むことになります。

移動の速さはInspectorビュー上で調整できるようにしています。この値を変えてプレイしてみてもちよいといい値を探しましょう。



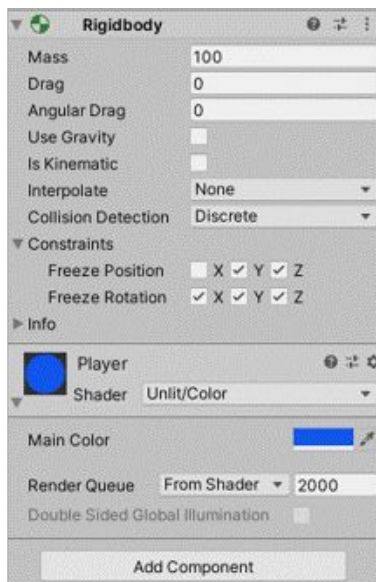
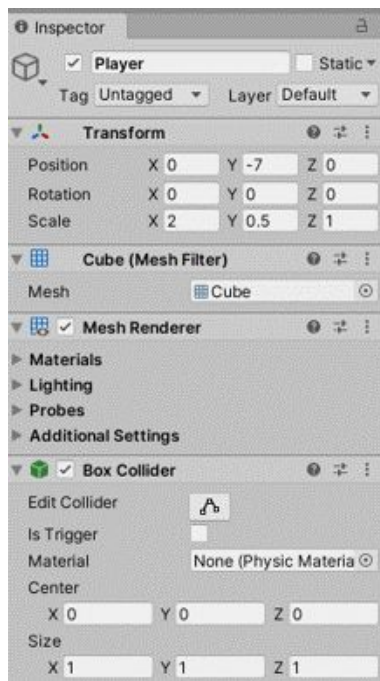
# プレイヤーを作成する

---



# 既存コンポーネントの設定

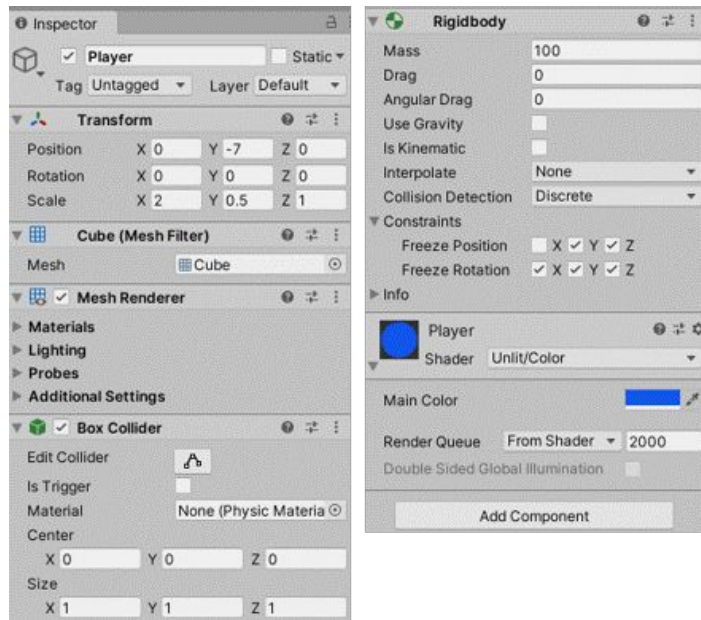
Cubeを作成し、名前をPlayerにします。Transformコンポーネントの値を変えて位置と大きさを調整してください。今回は**位置(0, -7, 0)**、**大きさ(2, 0.5, 1)**にしました。Playerのマテリアルを作成し、色を変更しましょう。



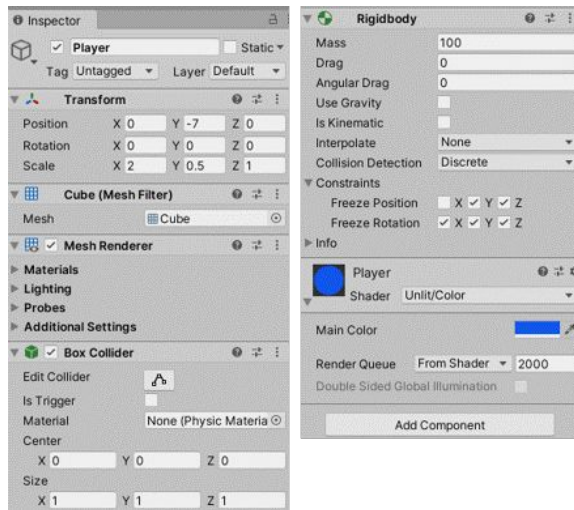


# 既存コンポーネントの設定

プレイヤーもボールや壁にぶつかるべきなので、移動を物理演算を用いて実装します。つまり、「Rigidbody」の設定を行います。Rigidbodyコンポーネントをアタッチして、Ballと同様に**空気抵抗を0、重力なし**にしてください。Constraintsは**位置xのみ許容**するようにして、**回転しない**ようにしましょう。

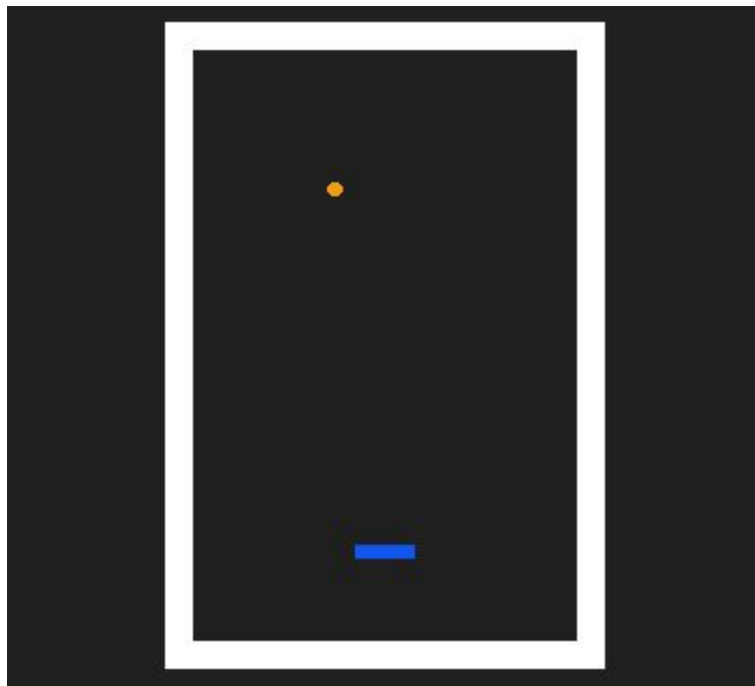


このままだとボールと衝突した際に、ボールは跳ね返らないと思います。これは、プレイヤーの質量がボールと同じであり、ボールに加わる力が小さいためです。(軽いものにぶつかる時と重いものにぶつかる時を考えてください。同じ速度でぶつかっても重いものにぶつかったときのほうが衝撃が強くなることが想像できるはずです。)ボールをきちんと跳ね返らせるために、Playerの質量を大きくします。**PlayerにアタッチしているRigidbodyのMassの値を大きくしてください。100ぐらいで大丈夫です。**



# スクリプトの作成

Playerスクリプトを作成し、左右に動かせるようにします。



# Player.cs

```
using UnityEngine;
class Player : MonoBehaviour
{
    [SerializeField] float speed = 10f; // プレイヤーの移動の速さ
    Rigidbody _myRigidbody;
    void Start()
    {
        _myRigidbody = GetComponent<Rigidbody>(); // Rigidbodyにアクセスして変数に保持
    }
    void Update()
    {
        // 左右のキー入力により速度を変更する
        _myRigidbody.velocity = new Vector3(Input.GetAxis("Horizontal") * speed, 0f, 0f);
    }
}
```

# Playerの移動

---

移動はRigidbodyにアクセスし、速度velocityの値を変更することで行います。

Input.GetAxis(“Horizontal”)は右入力の時1に、左入力の時-1に、入力なしのとき0になるのでこれを利用します。速度のx成分にこの値を使えば、右入力/左入力の時に速度のx成分が正/負になるため、左右に動くようになります。

Input.GetAxis 自体は (-1から1の範囲)

また、速さの調整ができるように変数speedを用意しています。

<https://docs.unity3d.com/ja/2021.1/Manual/class-InputManager.html>

# Rigidbodyをつける？付けない？

今回の例では、壁にはRigidbodyをつけていませんが、ボールはちゃんと壁に衝突しています。そのため、プレイヤーもRigidbodyをつけずにTransformコンポーネントの位置を変えて動かせばいいと思う人もいるかもしれません。

Rigidbodyをつけるか付けないかの基準は「動くか/動かないか」で決めるとよいです。つまり、壁は動かないのでRigidbodyをつけず、プレイヤーは動くのでRigidbodyをつけています。

これはUnityの物理演算の仕様なのですが、動く物体にRigidbodyをつけないと処理が重くなったり、正しく処理できなくなる不具合が生じます。また、物理演算する物体を動かす際に、transformで位置を変えることも推奨されていません。プレイヤーをRigidbodyのvelocityで動かしているのはそのためです。

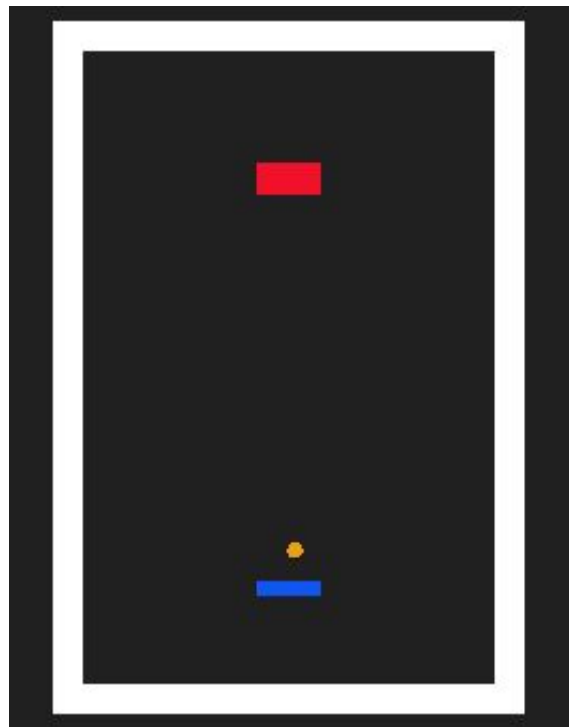
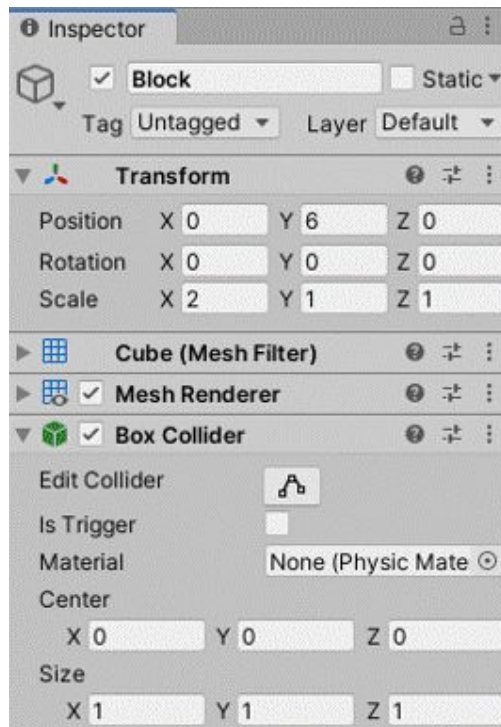
よくわからないなら、「物理演算させるものは全てRigidbodyを使い、Rigidbodyで動かす」と覚えておきましょう。今回の場合なら、壁にRigidbodyをつけても構いません。

# ブロック作成

---

# 既存のコンポーネントの設定

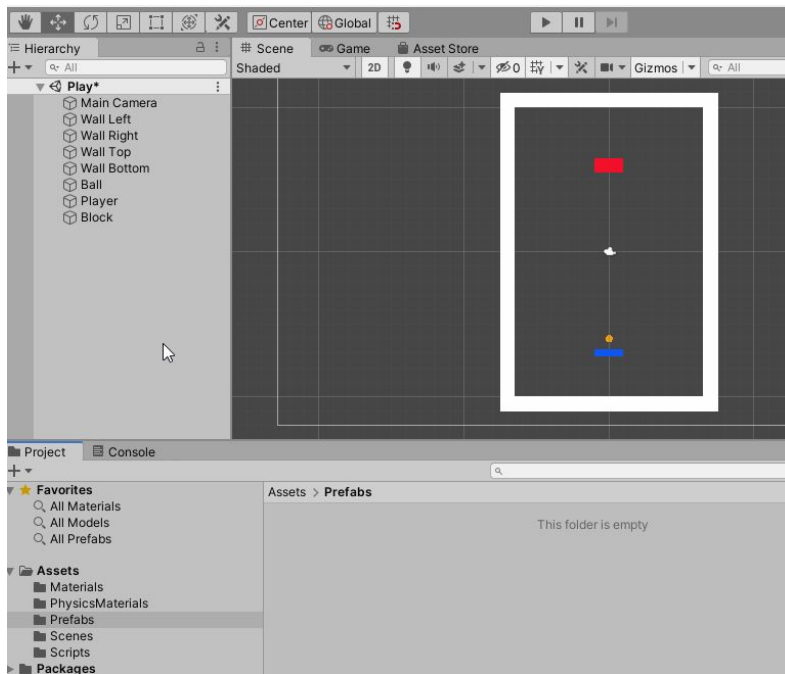
ブロックを作成します。Cubeを作成し、名前をBlockに変更します。大きさと位置を調整して色を変えましょう。大きさを(2, 1, 1)に、色を赤色にしてみました。





# Prefabにする

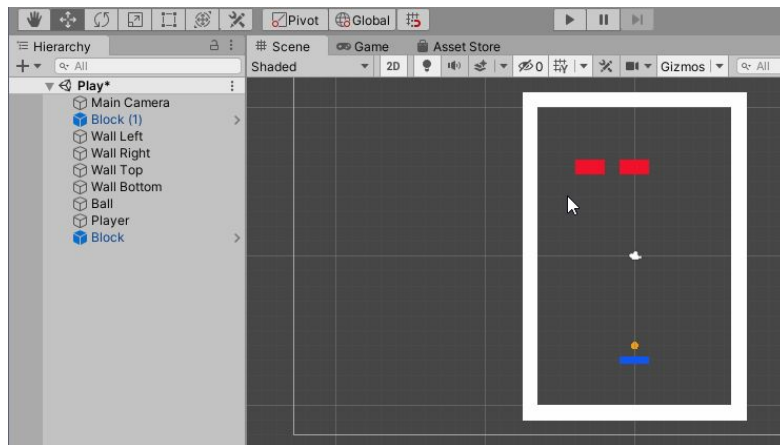
ブロックはステージに複数配置します。同じようなゲームオブジェクトを複数使用する場合はプレハブ機能が便利です。プレハブにしたいゲームオブジェクトをHierarchyビューからProjectビューにドラッグ&ドロップすれば作成できます。Blockプレハブを作成しましょう。



プレハブからゲームオブジェクトを作成する(インスタンス化といいます)にはプレハブをSceneビューやHierarchyビューにドラッグ&ドロップします。試しに、BlockプレハブからBlockのゲームオブジェクトを作成してみてください。

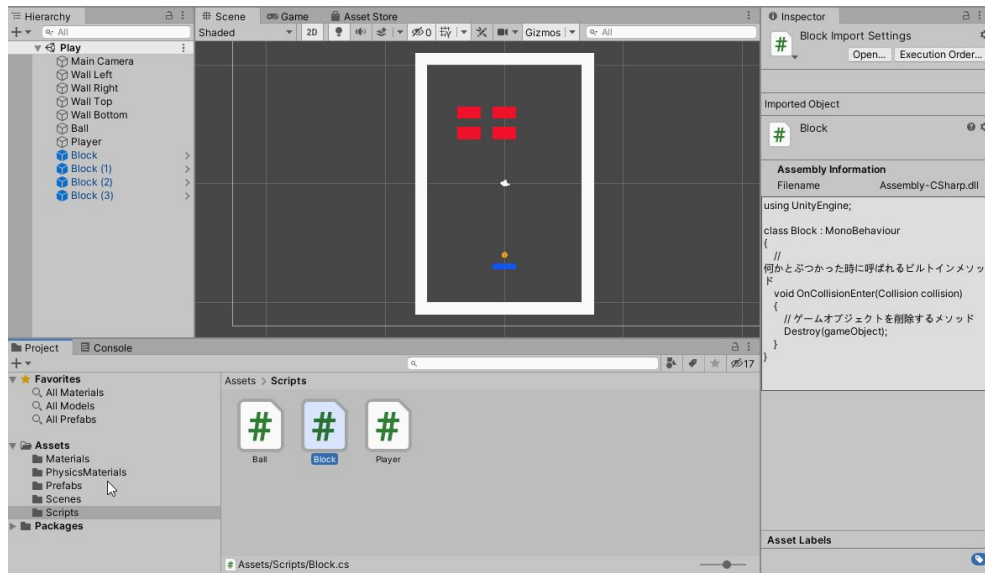
Blockゲームオブジェクトを複数作成して並べてみましょう。実はコピーを作るだけならプレハブ機能を使わなくてもCtrl+dのショートカットで実現できます。

プレハブ機能の真価が発揮されるのはBlockゲームオブジェクトに変更を加えたい場合です。次のスクリプト作成でその効果を利用してみましょう。



# スクリプトの作成

Blockスクリプトを作成し、ボールと衝突したときに消えるようにします。スクリプトを作成し、**Blockプレハブ**にアタッチしてください。プレハブにアタッチすることで、複製したBlockゲームオブジェクト全てに、その変更が反映されます。この機能のおかげでBlockの変更が容易になり、ゲームオブジェクト1つ1つにスクリプトをアタッチする必要がなくなります。



# Block.cs

---

```
using UnityEngine;
class Block : MonoBehaviour
{
    // 何かとぶつかった時に呼ばれるビルトインメソッド
    void OnCollisionEnter(Collision collision)
    {
        // ゲームオブジェクトを削除するメソッド
        Destroy( gameObject );
    }
}
```

# OnCollisionEnter

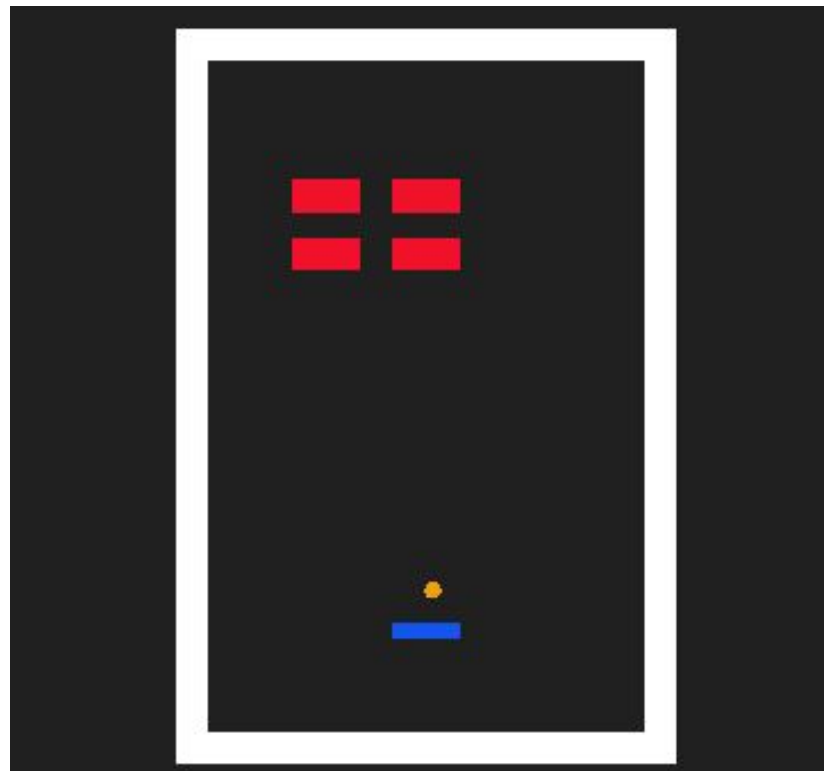
---

衝突時の処理はUnityが用意しているビルトインメソッドの1つであるOnCollisionEnterメソッドを使用することで実現できます。引数には衝突した相手の情報が入るCollision型の変数が必要です。

誰と衝突したかを判定する場合にはこの引数を使用しますが、今回は簡単のために、相手に関係なくBlockを消すことにするのでここでは使っていません。

ゲームオブジェクトを削除するメソッドとしてDestroyメソッドが用意されていますので、それを使用します。引数に削除するゲームオブジェクトを指定します。自分自身のゲームオブジェクトは小文字で始まるgameObjectによって取得できるので、それを指定すればOKです。

実行すると衝突時にBlockが消えていることがわかります。



# 衝突時に呼ばれるビルドインメソッド

---

OnCollisionEnterメソッドのほかに衝突が終わったとき(離れる瞬間)に呼ばれるOnCollisionExitメソッドや触れている間にずっと呼ばれるOnCollisionStayメソッドがあります。

これらのメソッドはBox ColliderやSphere ColliderなどのColliderコンポーネントがタッチされていないゲームオブジェクトに対しては呼ばれない他、Colliderが2Dのものだったり、IsTriggerの設定がoffになっている場合にも呼ばれないので注意しましょう。

※Colliderが2Dの場合(Box Collider 2Dなど)には代わりにOnCollisionEnter2Dなど2Dがついたメソッドが呼ばれます。

---

ここまでで、ボールとプレイヤー、ブロックの実装をしました。

次はゲームクリア/ゲームオーバー処理を実装します。



2

—

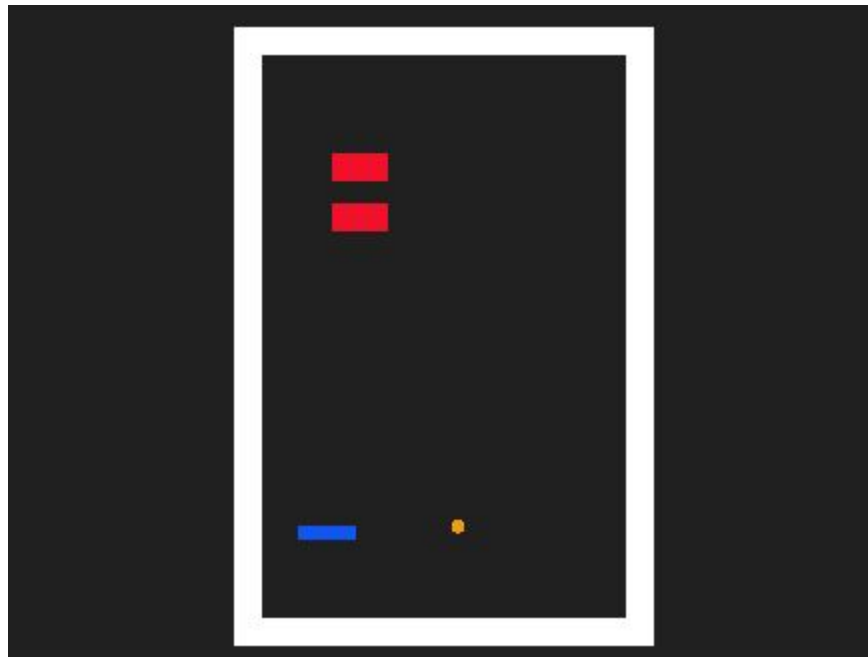
# ゲームオーバー

---

# ボールを消す

ブロック崩しはボールが下に行ってしまったらミスとします。そこで、下の壁に当たったときにボールを削除することにしましょう。

スクリプトを作成し、下の壁にアタッチしてください。これで、下の壁に当たったときにボールが消えるようにします。



# GameOver.cs

---

```
using UnityEngine;
class GameOver : MonoBehaviour
{
    // 衝突時に呼ばれる
    void OnCollisionEnter(Collision collision)
    {
        // 当たったゲームオブジェクトを削除する
        Destroy(collision.gameObject);
    }
}
```

# 衝突処理

---

衝突処理は、Blockの処理と同様に衝突時に呼ばれるOnCollisionEnterメソッドを使用します。

削除する処理も同様にDestroyメソッドを使えば問題ありません。ただし、ブロックの処理では削除するものがBlock自身だったので、自分自身のゲームオブジェクトを表すgameObjectを指定していましたが、今回は衝突した相手(この場合はボール)を削除するため、相手のゲームオブジェクトを指定しなければなりません。

相手の情報はOnCollisionEnterメソッドの引数であるCollision型の変数collisionに含まれており、**collision.gameObject**とすれば、**衝突した相手のゲームオブジェクトが得られます**。

※OnCollisitonEnterメソッドの引数の変数名がcollisionでない場合(括弧で括られた部分)は、必要に応じてcollisionを変数名で読み替えてください。

# 自分自身へのアクセスと衝突相手へのアクセス

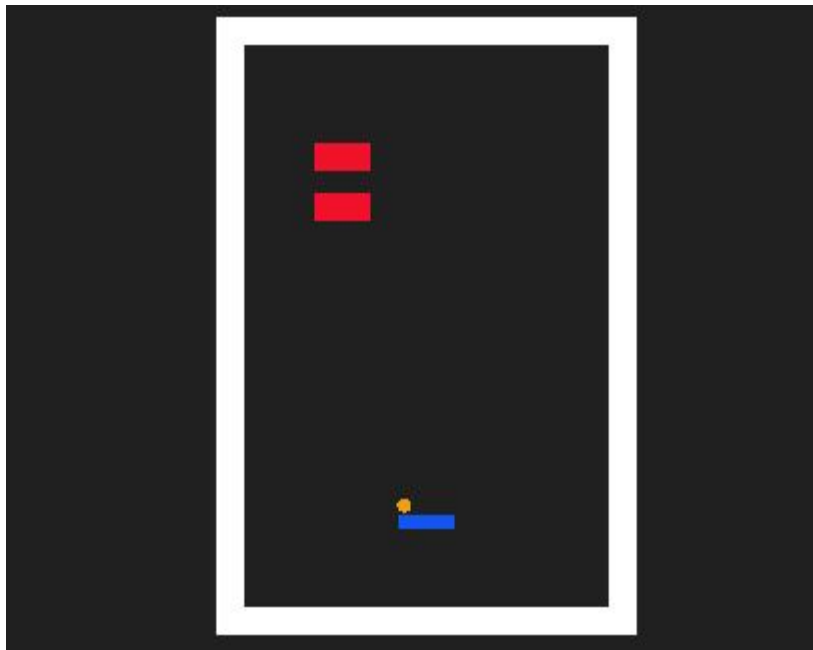
アクセス対象	自分自身	衝突相手
ゲームオブジェクト	<code>gameObject</code>	<code>collision.gameObject</code>
コンポーネント T	<code>GetComponent&lt;T&gt;()</code>	<code>collision.GetComponent&lt;T&gt;()</code>

# ゲームクリア

---

# ゲームをストップする

すべてのブロックを壊すことができれば、ゲームクリアです。ゲームクリアしたらどうなるのかという点でいえば、「次のステージに進む」、「リザルト画面を表示する」といったいくつかの案が考えられますが、ここでは簡単に「ゲームをストップする」処理を行うことにします。





# 親子関係

---

すべてのブロックが破壊されたことを判定する方法はいくつかありますが、今回は親子関係を利用して実装してみます。

親子関係とはゲームオブジェクト同士の関係を表すものです。具体的には以下の特徴があります。

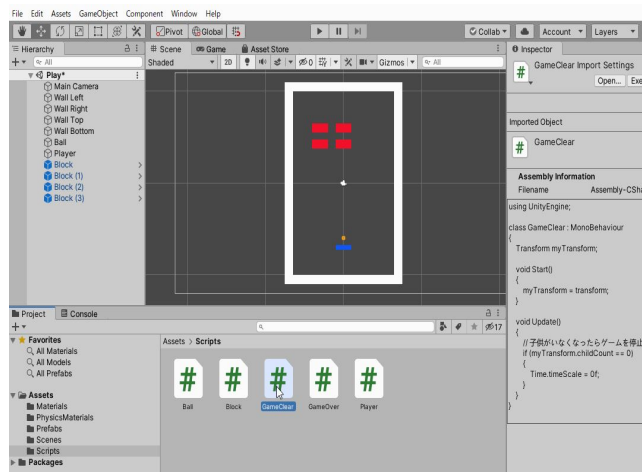
- 親のゲームオブジェクトが動くと、子供のゲームオブジェクトも一緒に動く
- 子供のゲームオブジェクトが動いても、親のゲームオブジェクトは動かない

この性質から、親のゲームオブジェクトの移動に対して一緒に動いて欲しい場合によく使用されます。例えば、プレイヤーキャラクターが身に着ける装備はプレイヤーキャラクターの子供にしておけばいいわけです。

今回の使用目的はそうではなく、ゲームオブジェクトの整理や管理のために親子関係を用います。フォルダのように階層分けすると言えば、イメージが伝わるでしょうか。

親子関係を作成するためには、Hierarchyビュー上で、子供にしたいゲームオブジェクトを親にしたいゲームオブジェクトにドラッグ&ドロップします。

今回はブロックを管理する親としてBlocksゲームオブジェクトを作成します。管理に使用するだけで、画面上に表示する必要がないので、Hierarchyビューで右クリック、**Create Empty**を選択して空のゲームオブジェクトを作成してください。そして、作成した空のゲームオブジェクトにBlockゲームオブジェクトをドラッグしてBlockが子供になるようにします。親となるゲームオブジェクトに次のスクリプトをアタッチしてください。



# GameClear.cs

```
using UnityEngine;
class GameClear : MonoBehaviour
{
    Transform _myTransform;
    void Start() {
        _myTransform = transform; // Transformコンポーネントを保持しておく
    }
    void Update() {
        // 子供がいなくなったらゲームを停止する
        if (_myTransform.childCount == 0){
            Time.timeScale = 0f;
        }
    }
}
```

# childCount

---

子供のゲームオブジェクトの数を調べるには、Transformコンポーネントの **childCount** を使用します。Transformコンポーネントに何度もアクセスするため、Startメソッドで変数に保持(キャッシュ)しておきます。

「ブロックが全て無くなった」=「子供のゲームオブジェクトの数が0になった」ですので、Updateメソッド内で子供のゲームオブジェクトの数をチェックして0の時にクリアの処理を行います。

# Time.timeScale

---

今回はクリア時に「ゲームを停止する」ということなので、ゲーム内時間を設定するTime.timeScaleの値を変更することにします。Time.timeScaleは動画の再生速度を考えるとわかりやすいと思います。つまり、1の場合には通常と同じ速さでゲームが進み、1より大きい場合には早送りのようになります。2にすれば2倍速に、1より小さくすればスロー再生ということですね。

**クリア時にはTime.timeScaleを0にしてゲームを停止します。**これで、ブロックを全て壊した際に、ゲームがストップするはずです。

# Time.timeScaleの影響範囲

---

timeScaleにより影響を受けるのは主に物理演算処理やアニメーション機能を使用した場合です。そのためtimeScaleを0にした場合、Rigidbodyを用いて物理演算で動いている場合にはそのゲームオブジェクトは停止します。

一方で、停止しないものもあり、例えば、UpdateメソッドはtimeScaleが0の場合でも毎フレーム（通常1秒に60回）実行されます。

timeScaleを使用する際には、何が影響を受けるのか/受けないのかという点に注意してください。

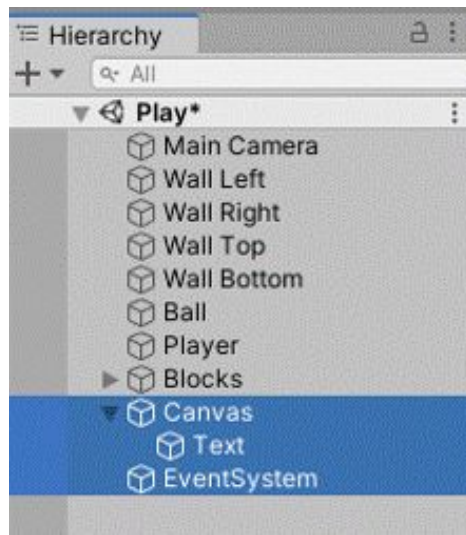
# テキストの作成

---

# Canvasの設定

処理自体は書けましたが、このままではゲームオーバー/ゲームクリアしたことがプレイヤーに伝わりません。そこで、「Game Over」、「Game Clear」と画面に表示するようにします。

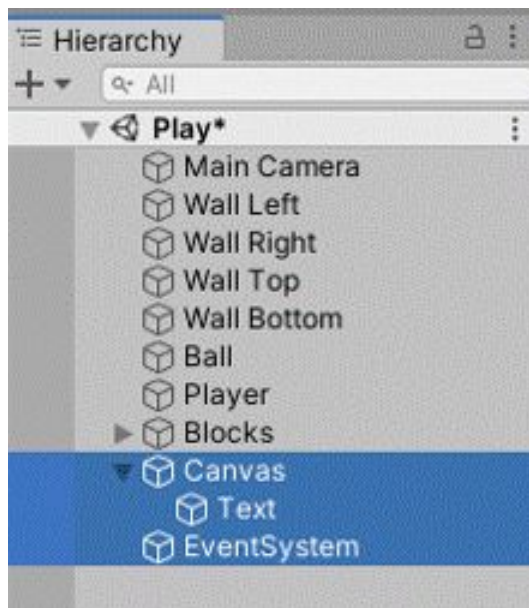
画面に文字を表示したり、プレイヤーが押せるボタンといったUI要素を使用する場合はCanvasを作成します。CanvasはHierarchyビューのメニューでUI->Canvasから作成できますが、他のUI要素を作成した場合にも自動で作成されます。





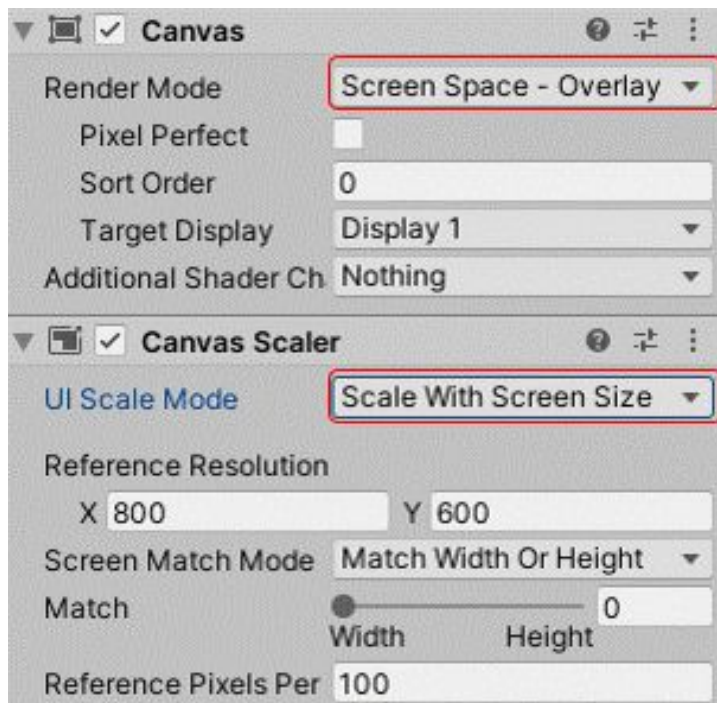
# UI→Text

**UI→Textを選択して作成してください。**以下の画像のように、Canvasの子としてTextゲームオブジェクトが作成されます。また、同時にUIのイベント処理を行うEventSystemが作成されますが、今回はUIのイベント処理は使用しないため、ここでは説明しません。（今回のゲームでは、EventSystemを消しても問題なく動くはずです。）

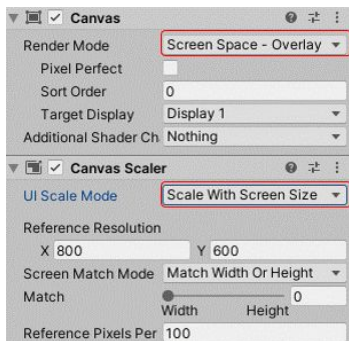


# RenderModel・UI ScalseModel

作成されたCanvasの設定を行います。項目がいくつかありますが、最初に気にするのはRender ModeとUI Scale Modeの2つです。



# RenderModel



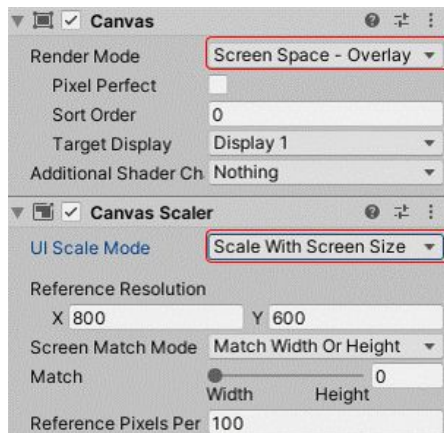
CanvasコンポーネントのRender ModeはUIをどのように表示するかのオプションです。今回は、文字を常に表示するようにしたいので、**Screen Space - Overlay**を選択します。

Render Mode	機能
Screen Space - Overlay	常に画面手前に描画する。カメラがなくても表示される。
Screen Space - Camera	指定したカメラの前に描画する。カメラが無いと表示されない。指定した距離よりも近くにあるゲームオブジェクトはUIより手前に表示される。
World Space	シーン内の平面オブジェクトであるかのように描画する。ゲーム内スクリーンや看板などに使用する。

# UI ScalseMode

Canvas ScalerのUI Scale ModeはUIの大きさをどう制御するかのオプションです。デフォルトではConstant Pixel Sizeになっていると思いますが、これだとウィンドウ画面の大きさによって大きさが変わってしまうので、**Screen With Screen Sizeにしましょう**。画面解像度を固定するようなゲーム(ドット絵をつかうようなゲーム)以外にはこの設定にしておくことをお勧めします。

※画面比率の変化まで考慮する場合にはScreen Match ModeをExpandにすると良いと思います。

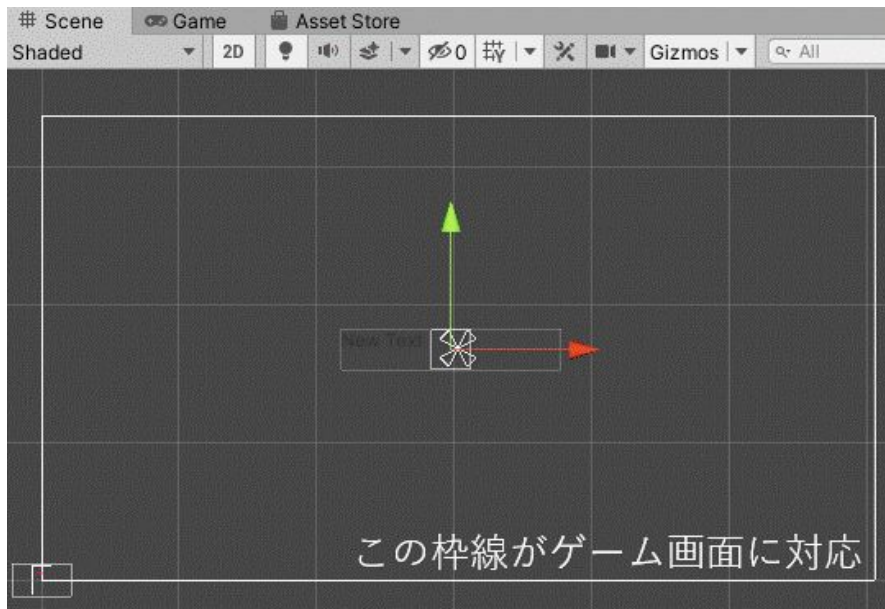


# Text設定

---

# Text設定

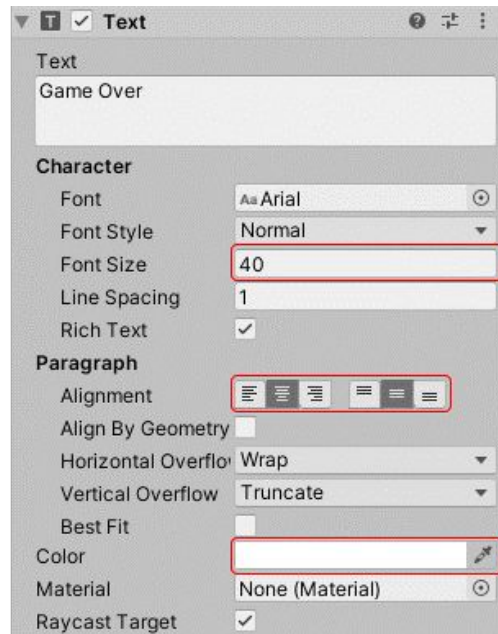
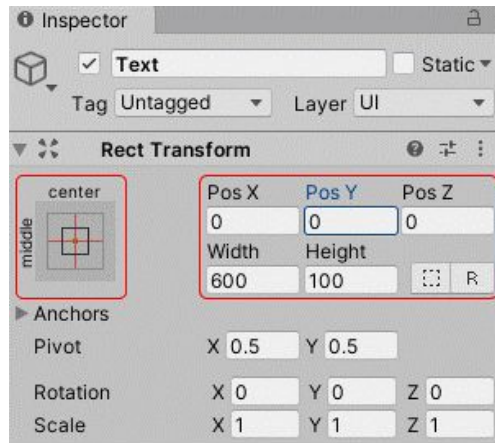
UI要素は通常のゲームオブジェクトとは異なる座標計算をするので注意が必要です。Textをダブルクリックしてみると、BlockやPlayerが置いてある位置とは別のところにあることがわかります。Canvasを表す枠線がゲーム画面に対応するので、画面中央に文字が来るように位置を調整しましょう。



# Text設定

下の画像のように設定しました。文字の色は背景に埋もれないように白色にしています。また、画面中央に表示するという事なので、中央揃えにしています。

フォントサイズがRect TransformのWidthとHeightの値に比べて大きい場合には文字が表示されないので注意しましょう。



# スクリプトからTextの内容を変更する

---



# 最初は何も表示されないようにしておく

最初は何も表示されないように、Textの文字を消すことにします。スクリプトを書かずにTextコンポーネントのTextの項目を無しにしてもいいですが、あまりお勧めしません。なぜなら、表示する場所や大きさを変更する場合に、

1. Textに文字を入力してSceneビューで文字を確認できるようにする。
2. 位置や大きさを変更する。
3. Textの文字を消す。

というように手順が増えるからです。ゲームスタート時に自動でTextの文字を消す処理を入れておけば、この手間を省くことができます。

次のスクリプトを作成してTextにアタッチしてください。

# ResetText.cs

```
using UnityEngine;
using UnityEngine.UI; // Textコンポーネントを使用する場合に必要
class ResetText : MonoBehaviour
{
    void Start()
    {
        // アクセスは1回きりなので、フィールド変数を用意していない
        Text _myText = GetComponent<Text>();
        _myText.text = ""; // textに空の文字列を設定する
    }
}
```

TextコンポーネントはUnityEngine.UI名前空間に存在するコンポーネントですので、使用する場合は、usingを使って宣言する必要があります。文字を設定するにはTextコンポーネントのtextが使用できます。空の文字列を代入することで何も表示しないようにします。

# ResetText.cs(TextMeshPro)

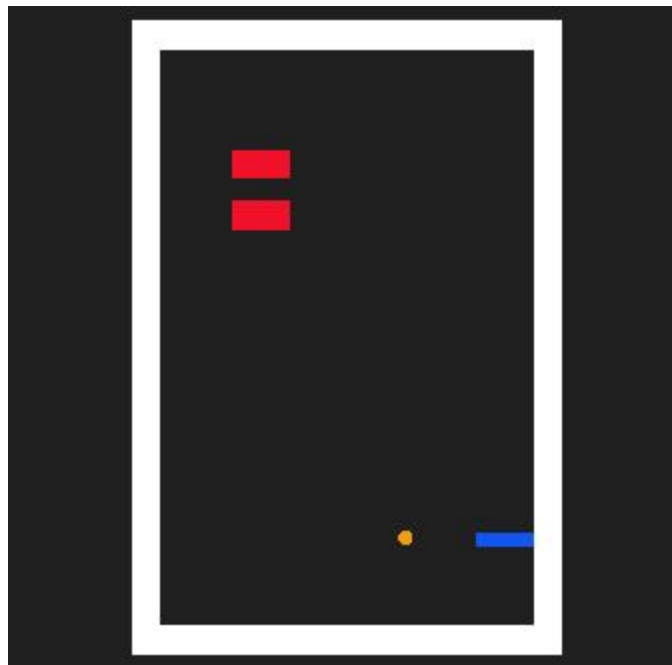
```
using UnityEngine;
using TMPro;
class ResetText : MonoBehaviour
{
    void Start()
    {
        // アクセスは1回きりなので、フィールド変数を用意していない
        TextMeshProUGUI _myText = GetComponent<TextMeshProUGUI>();
        _myText.text = ""; // textに空の文字列を設定する
    }
}
```

# ゲームオーバー

---

---

ゲームオーバー時には「Game Over」に変更する  
GameOverスクリプトを次のように変更してください。



# GameOver.cs

---

```
using UnityEngine;
using UnityEngine.UI; // 追加
class GameOver : MonoBehaviour
{
    // publicにしてInspectorから設定できるようにする
    [SerializeField] Text gameOverMessage;

    void OnCollisionEnter(Collision collision)
    {
        // Game Overと表示する
        gameOverMessage.text = "Game Over";
        Destroy(collision.gameObject);
    }
}
```

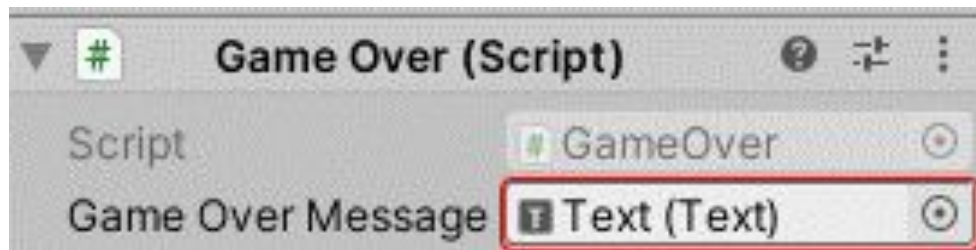
# GameOver.cs(TextMeshPro)

```
using UnityEngine;
using TMPro; // 追加
class GameOver : MonoBehaviour
{
    // publicにしてInspectorから設定できるようにする
    [SerializeField] TextMeshProUGUI gameOverMessage;

    void OnCollisionEnter(Collision collision)
    {
        // Game Overと表示する
        gameOverMessage.text = "Game Over";
        Destroy(collision.gameObject);
    }
}
```

GameOverスクリプトがアタッチされているゲームオブジェクトはTextゲームオブジェクトとは異なるため、今までのようにアクセスすることができません。異なるゲームオブジェクトにアタッチされているコンポーネントにアクセスする最も簡単な方法はpublic変数などを用意してInspector上で設定することです。ただし、設定し忘れるとエラーが起こるので注意してください。

設定はHierarchyビューまたはSceneビューで下の壁を選択して、InspectorビューのGame Over Messageとなっている箇所にTextゲームオブジェクトをドラッグ&ドロップすることで可能です。他にも右側にある丸のマークをクリックして選択してもよいです。**Textゲームオブジェクト**を指定しましょう。

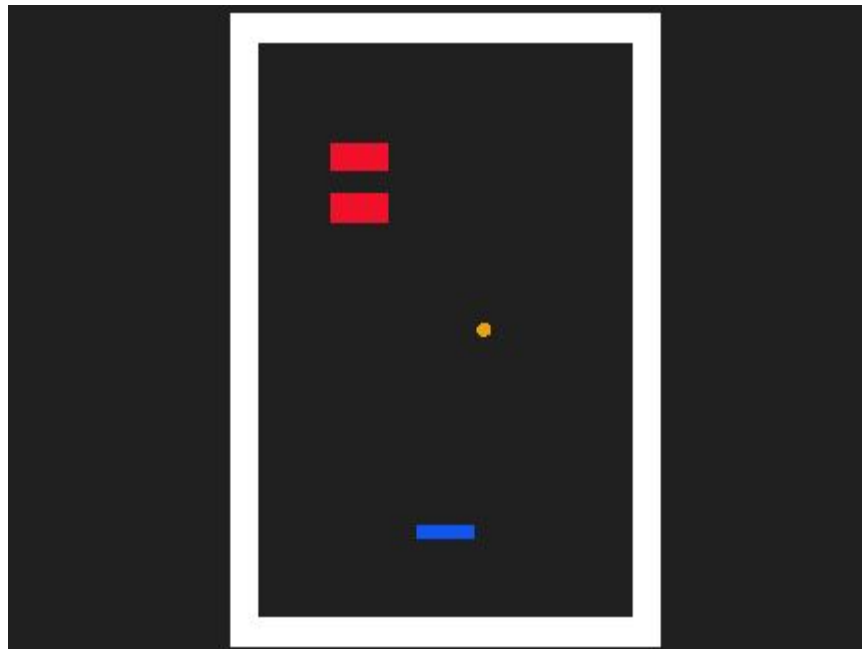




ゲームクリア時には「Game Clear」に変更する

ゲームクリア時も同様にTextゲームオブジェクトのTextコンポーネントにアクセスして文字列を設定しましょう。

GameClearスクリプトを以下のように変更してください。



# GameClear.cs

```
using UnityEngine;
using UnityEngine.UI; // 追加
class GameClear : MonoBehaviour
{
    [SerializeField] Text _gameClearMessage; // 追加
    Transform _myTransform;
    void Start() { _myTransform = transform; }
    void Update() {
        if ( _myTransform.childCount == 0)
        {
            _gameClearMessage.text = "Game Clear\n<size=25>~press Enter  
to continue~</size>"; // 追加
            Time.timeScale = 0f;
        }
    }
}
```

# GameClear.cs(TextMeshPro)

```
using UnityEngine;
using TMPro; // 追加
class GameClear : MonoBehaviour
{
    [SerializeField] TextMeshProUGUI _gameClearMessage; // 追加
    Transform _myTransform;
    void Start() { _myTransform = transform; }
    void Update() {
        if ( _myTransform.childCount == 0)
        {
            _gameClearMessage.text = "Game Clear\n<size=25>~press Enter  
to continue~</size>"; // 追加
            Time.timeScale = 0f;
        }
    }
}
```

---

こちらでもGameOverスクリプト同様、Inspector上でTextゲームオブジェクトを設定することを忘れないようにしましょう。

以上で、ゲームクリア/ゲームオーバー処理の最低限の実装が完了です。

今回はゲームクリア/ゲームオーバー処理を実装しました。これで、最低限ゲームの形になったと思います。ただし、現状だと1度しかプレイできない、スタートと同時にボールが動いてしまう、などの不便な点があります。次はそこを改善していきます。

3

—

# ゲームオーバー後

---

ゲームオーバーになった後にボタンを押すと再度プレイができるようにします。



# GameOver.cs

```
using UnityEngine.SceneManagement; // 追加
class GameOver : MonoBehaviour
{
    [SerializeField] Text gameOverMessage;
    bool isGameOver = false; // ゲームオーバーしたかどうかを判断するための変数
    void Update(){
        // ゲームオーバーになっている、かつ、Submitボタンを押したら実行する
        if(isGameOver && Input.GetButtonDown("Submit")){
            SceneManager.LoadScene("Play"); // Playシーンをロードする
        }
    }
    void OnCollisionEnter(Collision collision){
        gameOverMessage.text = "Game Over\n<size=25>~Press Enter~</size>";
        Destroy(collision.gameObject);
        isGameOver = true; // isGameOverをtrueにする(フラグを立てる)
    }
}
```

---

ゲームオーバーになっていない状態でボタンを押しても、何も反応しないようにするために、**ゲームオーバーになっているか/いないかを記憶しておくための変数 isGameOverを用意しました。**

**ゲームスタート時にはisGameOverはfalseにし、ゲームオーバーになったらtrueにします。**ゲームオーバーの判定処理を行っているOnCollisionEnterメソッド内に isGameOverをtrueにする処理を追加します。これで、isGameOverの値を確認すれば、ゲームオーバーになっているかどうか分かるようになりました。



---

あとは毎フレーム呼ばれるUpdateメソッド内でゲームオーバー時にボタンを押すと処理が行われるようにif文を書けばOKです。入力処理にはInput.GetButtonDownメソッドを用いています。ここでは、Submitボタンを指定しました。デフォルトではスペースキーなどが割り当てられています。(InputMangaerで確認できます。)

---

リトライ機能の実装ですが、おそらく最も簡単であろう「シーンをロードし直す」方法を採用します。シーンのロードにはSceneManagerクラスのLoadSceneメソッドを使用します。

```
SceneManager.LoadScene(ロードするシーン);
```

シーンPlayをロードするには以下のようになります。

```
SceneManager.LoadScene("Play");
```

SceneManagerクラスは名前空間UnityEngine.SceneManagement内にありますので、名前空間UnityEngine.SceneManagementをusingで指定しましょう。

# フラグ

---

変数isGameOverのように、ゲームの状態を管理する際に、「した/していない」を判定するための変数として「true/false」のどちらかの値を取るbool型の変数が良く使われます。

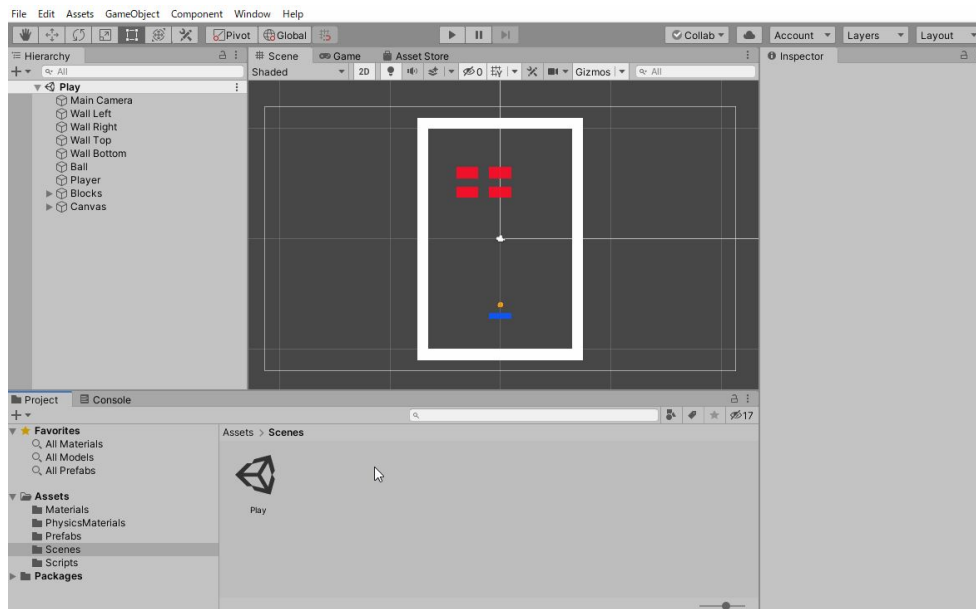
このような処理はフラグとも呼ばれ、falseだった値をtrueにすることをフラグを立てると言ったりします。

フラグは「条件を満たす」という意味合いのプログラミング用語でしたが、今では伏線や予測といった意味合いで広く用いられていますね。「フラグ回収」や「フラグが立った」という単語はどこかで見たことがあるのではないのでしょうか。

# ビルドマネージャーにシーンを登録する

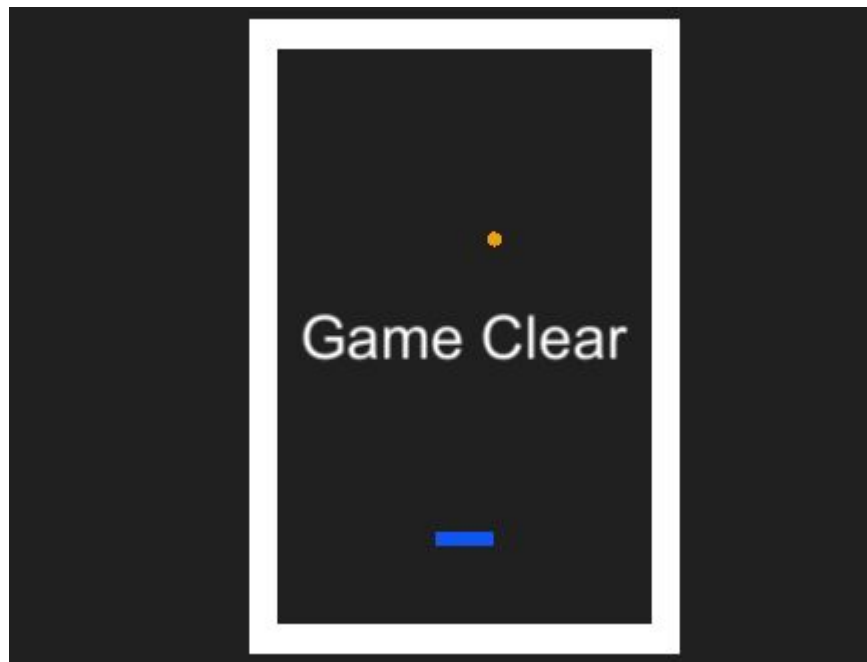
Build Settingsに登録していないシーンをロードしようとするエラーが起こるので注意が必要です。

メニューのFile->Build Settingsを開いて、PlayシーンをScenes In Buildの箇所にドラッグ&ドロップして登録を行いましょう。



# ゲームクリア後

ゲームクリアした際にもボタン入力でゲームをもう一度遊べるようにしましょう。  
GameClearスクリプトを以下のように変更してください。



# GameClear.cs

---

```
using UnityEngine;
using UnityEngine.UI;
// 追加
using UnityEngine.SceneManagement;

class GameClear : MonoBehaviour
{
    public Text gameClearMessage;
    Transform myTransform;
    // ゲームクリアしたかどうかを管理するフラグを用意
    bool isGameClear = false;
    void Start()
    {
        myTransform = transform;
    }
}
```

```
void Update()
{
    if (myTransform.childCount == 0)
    {
        gameClearMessage.text = "Game Clear";
        Time.timeScale = 0f;
        // ゲームクリアのフラグを立てる
        isGameClear = true;
    }
}
```

```
// ゲームクリアしている、かつ、ボタン入力でシーンを再ロード
if(isGameClear && Input.GetButtonDown("Submit"))
{
    // timeScaleを1に戻しておく
    Time.timeScale = 1f;
    // シーンのロード
    SceneManager.LoadScene("Play");
}
}
```

---

基本的な処理はGameOverスクリプトの変更と同じです。ゲームクリアしたかどうかをbool型の変数isGameClearを用いて管理し、Submitボタン入力でPlayのシーンをロードするように書いています。

注意点はTimeScaleだけです。ゲームクリア時にはTimeScaleの値を0にしてゲームを停止しているので、**再びゲームをスタートするためにはTimeScaleの値を1に戻す必要があります。**

今回は、ゲーム終了後にも再度ゲームを遊べるようにリトライ機能を実装しました。これで、繰り返し遊べるだけでなく、テストプレイ時にも毎回再生ボタンを押さずに済みます。次回はボールの挙動を改善します。



4

—

---

今回はボールの挙動を改善します。止まったり速くなりすぎないようにするために、スピードに制限を書けます。また、プレイヤーとの衝突時に跳ね返る方向をプレイヤーとの衝突位置によって変えるようにします。

---

現在、ボールの動きは物理演算に完全に頼っており、プレイヤーとのぶつかり方によっては速度が上がったり、あるいは、止まってしまいそうぐらい遅くなってしまうことがあります。物理学的には正しくても、ゲームとして遊べなければ意味がありません。そのため、ボールの動きはある程度、制御できるようにしたほうがよいでしょう。

そこで、**ボールの動く速さに最大値と最小値を設定**します。次のようにBallスクリプトを変更してください。

# Ball.cs

```
class Ball : MonoBehaviour
```

```
{
```

```
    [SerializeField] float speed = 5f;
```

```
    [SerializeField] float minSpeed = 5f; // 速さの最小値を指定する変数を追加
```

```
    [SerializeField] float maxSpeed = 10f; // 速さの最大値を指定する変数を追加
```

```
    Rigidbody _myRigidbody;
```

```
    void Start() { _myRigidbody = GetComponent<Rigidbody>();
```

```
        _myRigidbody.velocity = new Vector3(speed, speed, 0f); }
```

```
    void Update() { // 毎フレーム速度をチェックする
```

```
        Vector3 velocity = myRigidbody.velocity; // 現在の速度を取得
```

```
        // 速さを計算
```

```
        float clampedSpeed = Mathf.Clamp(velocity.magnitude, minSpeed, maxSpeed);
```

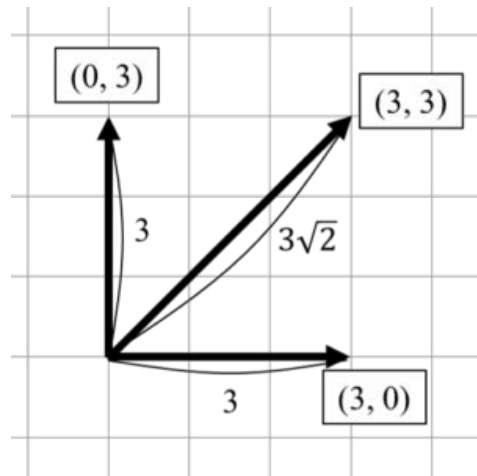
```
        myRigidbody.velocity = velocity.normalized * clampedSpeed; // 速度を変更
```

```
    }
```

```
}
```

このスクリプトを理解するには、少しばかりベクトルの知識が必要になります。ややこしくならないように、2次元で話をします。

ベクトルには方向と大きさという概念があります。例えば、ベクトルの値が $(3, 0)$ の場合、ベクトルの方向は $x$ の正方向で、大きさは3となります。同様に、ベクトル $(0, 3)$ は方向が $y$ の正方向、大きさが3となります。

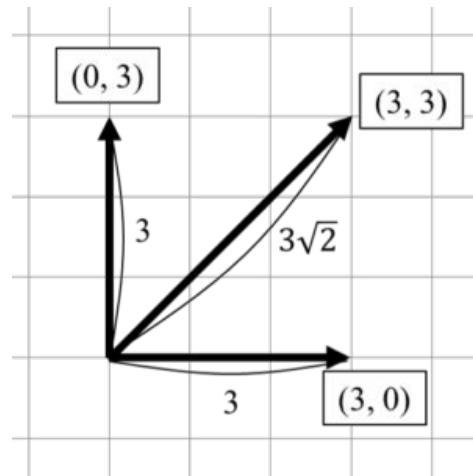


では、ベクトルが(3, 3)の場合はどうなるでしょうか。方向は斜め45度になり、大きさは三平方の定理から $3\sqrt{2}$ になります。

ベクトルの大きさはmagnitudeで計算できます。例えば、ベクトルvelocityの大きさを取得するには以下のように書くことができます。

```
velocity.magnitude
```

ベクトル	方向	大きさ
(3, 0)	(1, 0)	3
(0, 3)	(0, 1)	3
(3, 3)	$(1/\sqrt{2}, 1/\sqrt{2})$	$3\sqrt{2}$



---

一方、ベクトルの方向を表す場合には大きさ1のベクトルが良く用いられます。ベクトル(3, 0)の場合には方向は(1, 0)が該当します。

(3, 3)の場合、大きさが $3\sqrt{2}$ なので、x成分とy成分をそれぞれ $3\sqrt{2}$ で割り算した $(1/\sqrt{2}, 1/\sqrt{2})$ が長さ1のベクトルとなります。

大きさ1のベクトルが使われるのは、計算が楽になるからです。方向を変えずに大きさだけを変更したい場合には、大きさ1のベクトルに掛け算してやればいいだけです。方向(1, 0)、大きさ5のベクトルを計算したかったら、(1, 0)にそのまま5をかけて(5, 0)が得られます。

方向をそのままにして大きさを1にしたベクトルを取得するにはnormalizedが利用できます。例えば、ベクトルvelocityの方向を表す大きさ1のベクトルは以下のように書くことができます。

`velocity.normalized`

# Ball.cs

---

```
void Update()
{
    // 現在の速度を取得
    Vector3 velocity = myRigidbody.velocity;
    // 速さを計算
    float clampedSpeed = Mathf.Clamp(velocity.magnitude, minSpeed, maxSpeed);
    // 速度を変更
    myRigidbody.velocity = velocity.normalized * clampedSpeed;
}
```



# Mathf.Clamp( 値 , 最小値 , 最大値 )

まず、Rigidbodyコンポーネントにアクセスして現在の速度を取得し、変数velocityに代入します。

次に、速さに制限を書けるためにMathf.Clampメソッドを使用して制限後の速さを計算して変数clampedSpeedに代入しています。Mathf.Clampメソッドは値をある範囲内に強制的に収めるメソッドです。

// Mathf.Clamp( 値 , 最小値 , 最大値 )のように用いる

// 5は1を超えているので1になる

```
Mathf.Clamp(5, 0, 1);
```

// 5は6より小さいので6になる

```
Mathf.Clamp(5, 6, 10);
```

# Mathf.Clamp( 値 , 最小値 , 最大値 )

---

したがって、1つめの引数に現在のボールの速さであるvelocity.magnitudeを、2つめの引数に最低速度のminSpeedを、3つめの引数に最高速度のmaxSpeedを指定すれば、制限をかけた後の速度を計算できます。

変数minSpeed, maxSpeedに関しては、**インスペクタービュー上で値を変更できるように宣言**しています。

最後に、**Rigidbodyコンポーネントの速度velocityに大きさを変えたベクトルを設定**してあげれば完了です。normalizedを用いて方向を計算し、それに速さを掛け算すれば、方向を変えずに速さだけを変更できます。

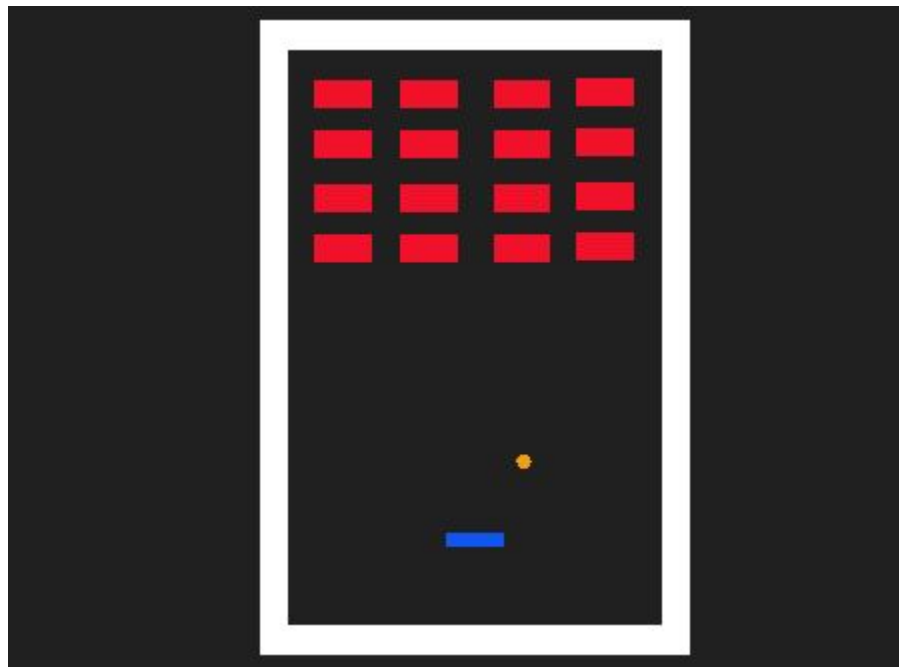
# ボールの挙動を変える

---

# ボールの挙動を変える

ボールが当たった場所によって、ボールの跳ね返る方向が変わるようにします。具体的には、プレイヤーの右側に当てればボールは右に、左側に当てればボールは左に跳ねるように変更します。これにより、ボールを当てる位置が重要になり、プレイにテクニックが必要になります。物理的には正しくないですが、ゲームとして面白くなるならこのような挙動もありです。

Ballスクリプトを以下のように変更してください。



# Ball.cs

```
using UnityEngine;
class Ball : MonoBehaviour
{
    public float speed = 5f;
    public float minSpeed = 5f;
    public float maxSpeed = 10f;
    Rigidbody _myRigidbody;
    Transform _myTransform; //Transformコンポーネントを保持しておくための変数を追加

    void Start()
    {
        _myRigidbody = GetComponent<Rigidbody>();
        _myRigidbody.velocity = new Vector3(speed, speed, 0f);
        _myTransform = transform; // Transformコンポーネントを取得して保持しておく
    }
}
```

# Ball.cs

```
void Update()
{
    Vector3 velocity = myRigidbody.velocity;
    float clampedSpeed = Mathf.Clamp(velocity.magnitude, minSpeed, maxSpeed);
    myRigidbody.velocity = velocity.normalized * clampedSpeed;
}

void OnCollisionEnter(Collision collision) // 衝突したときに呼ばれる
{
    if (collision.gameObject.CompareTag("Player")) // プレイヤーに当たると跳ね返る方向を変える
    {
        Vector3 playerPos = collision.transform.position; // プレイヤーの位置を取得
        Vector3 ballPos = myTransform.position; // ボールの位置を取得
        Vector3 direction = (ballPos - playerPos).normalized; // プレイヤーから見たボールの方向計算
        float speed = myRigidbody.velocity.magnitude; // 現在の速さを取得
        myRigidbody.velocity = direction * speed; // 速度を変更
    }
}
```

---

跳ね返る方向を計算するために、ボールの現在位置を取得する必要があるの  
で、Transform型の変数を用意して、Startメソッドで自身のTransformコンポーネン  
トを取得しています。

続いて、衝突した際の跳ね返る方向を制御するために、衝突時に呼ばれる  
OnCollisionEnterメソッドを追加します。OnCollisionEnterメソッドの引数は衝突した  
相手の情報が含まれるCollision型のデータが渡されます。Blockスクリプトではこ  
の変数は使用していませんでしたが、今回はぶつかった相手を区別するために使用し  
ます。つまり、**壁やブロックに当たったときには普通に跳ね返り、プレイヤーに当  
たったときのみ特殊な跳ね返りを起こすようにします。**

衝突した相手のゲームオブジェクトには、Collision型の変数collisionにgameObject  
をつけることでアクセスできます。

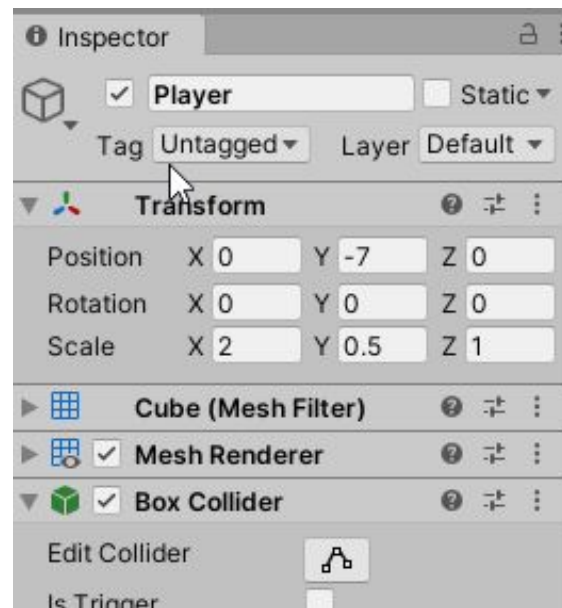
## // 衝突した相手のゲームオブジェクト

`collision.gameObject`

ぶつかった相手の種類を識別する方法の1つとしてタグを利用する方法を紹介します。ゲームオブジェクトにはタグと呼ばれる文字列をそれぞれに設定できるので、プレイヤーのゲームオブジェクトにPlayerというタグを設定して、プレイヤーとそれ以外を区別することにします。

タグの設定はインスペクタービューで行います。Playerゲームオブジェクトを選択してTagの部分をクリックし、Playerを選択して**TagをPlayerに変更しましょう**。

(Playerはよく使われるタグなので、デフォルトで用意されていますが、「Add Tag…」から自分で好きなタグを作成することもできます。)





# タグを用いた判定(CompareTag)

```
// CompareTag(“タグ”)のように使う  
// 衝突した相手のゲームオブジェクトのタグがPlayerだったらtrueになる  
collision.gameObject.CompareTag("Player");
```

if文と組み合わせることで、特定の相手とぶつかった時の処理を行うことができます。

```
void OnCollisionEnter(Collision collision) // 衝突したときに呼ばれる  
{  
    if (collision.gameObject.CompareTag("Player"))  
    {  
        // タグがPlayerであるゲームオブジェクトの場合のみ実行される処理をここに書く  
    }  
}
```

# タグの判定方法

タグを判定する方法には、CompareTagメソッドを使用する以外にもタグを直接比較する方法もあります。今回のケースだと、if文の条件を以下のように書いても問題なく動作します。

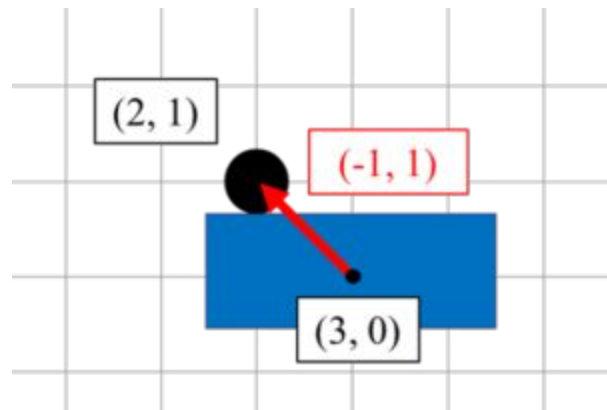
```
if (collision.gameObject.tag == "Player"){ // Playerとの衝突処理 }
```

CompareTagメソッドを用いる方法との違いは計算速度です。CompareTagメソッドのほうが高速ですので、CompareTagメソッドを使用することを推奨されています。

ボールがプレイヤーの右側に当たったら右に、左側に当たったら左に跳ね返るようにするには、跳ね返る方向を計算する必要があります。

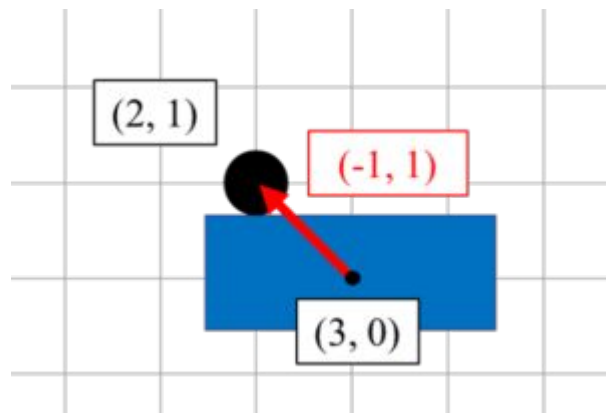
まず、計算に必要なプレイヤーとボールの位置をTransformコンポーネントにアクセスして取得します。

```
// プレイヤーの位置を取得  
Vector3 playerPos = collision.transform.position;  
// ボールの位置を取得  
Vector3 ballPos = myTransform.position;
```



今回のケースでは、ボールの跳ね返り方向は、プレイヤーから見たボールの方向にしています。プレイヤーから見たボールの方向はベクトルの引き算により求められます。

例えば、プレイヤーが(3, 0)にいて、ボールが(2, 1)にいるときには、プレイヤーから見てボールは左斜め上におり、ボールの位置からプレイヤーの位置を引き算して得られるベクトル $(2, 1) - (3, 0) = (-1, 1)$ の方向に相当します。



---

つまり、ボールの座標からプレイヤーの座標を引き算したうえでnormalizedを使えば、大きさが1の方向を表すベクトルが計算できます。

// プレイヤーから見たボールの方向を計算

```
Vector3 direction = (ballPos - playerPos).normalized;
```

方向が得られたので、あとは速さを掛け算してやれば、衝突後のボールの速度が求められます。速さは、ボールの速さをそのまま使用すればいいでしょう。

// 現在の速さを取得

```
float speed = myRigidbody.velocity.magnitude;
```

最後に、Rigidbodyコンポーネントのvelocityに求めた速度を代入すれば完了です。

// 速度を変更

```
myRigidbody.velocity = direction * speed;
```

---

今回は、ボールの速さに制限をかけて、跳ね返る方向を工夫しました。次はビルドを行い実行ファイル化することで、Unityなしでもゲームがプレイできるようにします。

# BGM追加

---

# Player.cs

```
AudioSource m_MyAudioSource;
void Start()
{
    // Rigidbodyにアクセスして変数に保持
    myRigidbody = GetComponent<Rigidbody>();
    m_MyAudioSource = GetComponent<AudioSource>();
}
void OnCollisionEnter(Collision collisionInfo)
{
    if (collisionInfo.gameObject.CompareTag("Ball"))
    {
        m_MyAudioSource.Play();
    }
}
```



5

—

ビルド

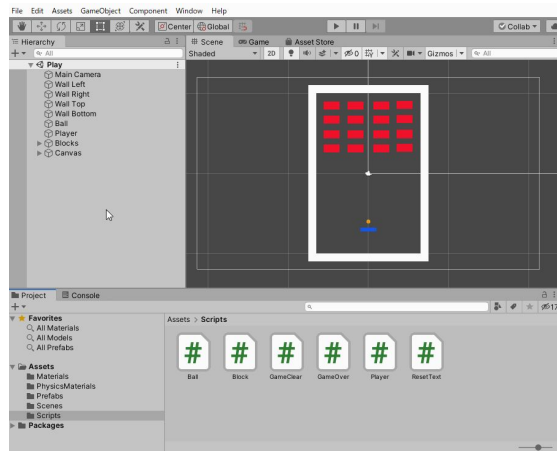
—

---

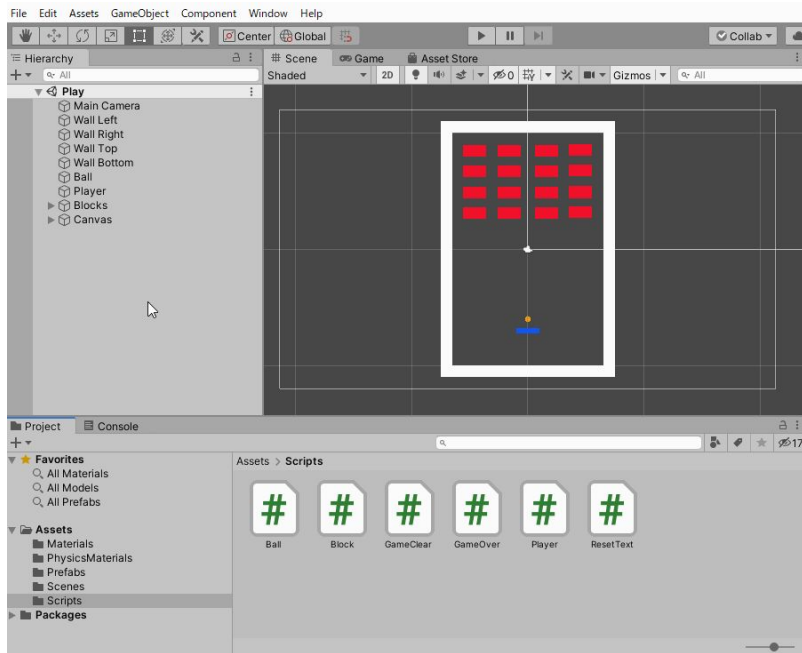
今回はゲームをビルドします。今まではUnity上でしかゲームをプレイすることができませんでした。ゲームをビルドすることでパソコンやWebブラウザでゲームを遊ぶことができるようになります。この記事では、PC向けのビルドとWebブラウザ向けのビルドを行います。

UnityではWindowsやMacなどのパソコン用ゲームのほかにも、Android, iOSといったモバイルゲーム、Webブラウザで実行できるWebGL形式のゲームなど様々な機器・環境で動作するゲームを作成することができます。

このような動作環境のことをプラットフォームと呼びます。**プラットフォームの変更はBuild Settingsにて行います。**File->Build SettingsからBuild Settingsを開き、対象となるプラットフォームを選択してSwitch Platformを押せばプラットフォームを変更できます。



選択できるプラットフォームの種類は、Unityインストール時にビルドサポートを導入したかによって変わります。目的のプラットフォームがない場合には、Unity Hubから追加のモジュールをインストールしましょう。



PC

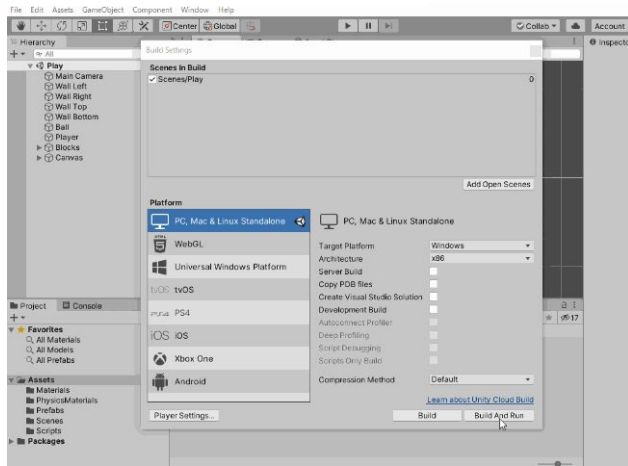
—

# とりあえずビルドしてみる

プラットフォームを変更したら、BuildまたはBuild And Runをクリックして保存先フォルダを指定すれば、ビルドを開始します。

Build And Runの場合はビルド後に自動で実行されるようになります。最終チェックしたい場合はBuild And Runを選択しましょう。

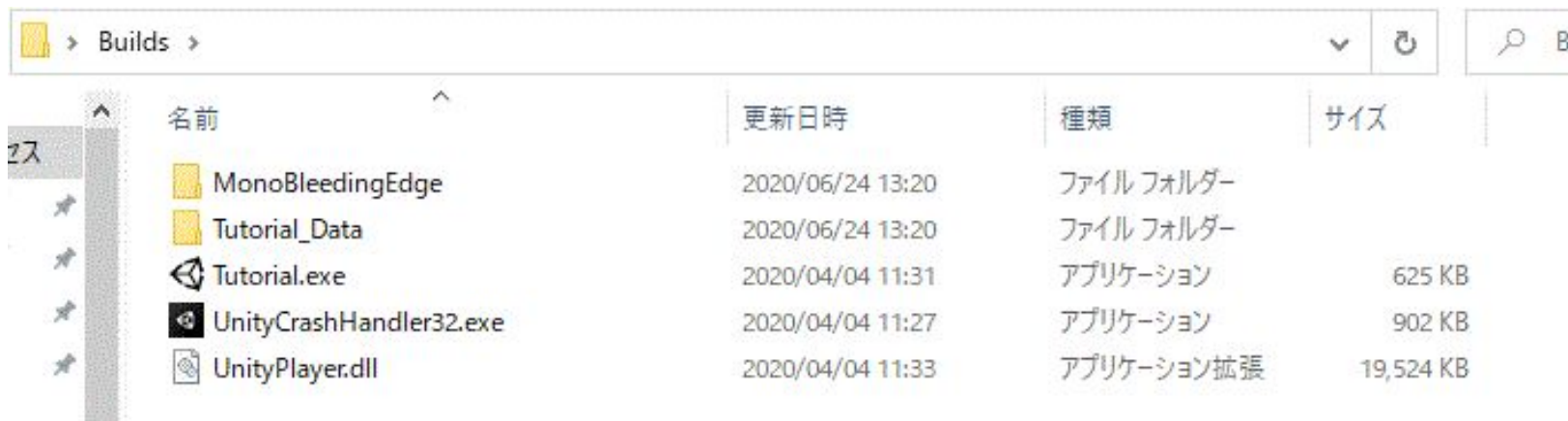
デフォルトだと全画面表示で表示されると思います。今回のゲームではゲームを終了するボタンを作成していないため、やめる場合には強制終了しましょう。WindowsならAlt+F4で閉じたり、タスクマネージャーからプロセスを終了させます。



# ビルド後

ビルド後は実行ファイルがフォルダー内に作成されます。下の画像はWindows向けにビルドした場合の例です。実行ファイル名はデフォルトではプロジェクト名になります。今回、Tutorialという名前のプロジェクトを作成したので、Tutorial.exeになっています。これを実行すれば、ゲームが起動するはずです。

他の人に遊んでほしい場合はビルドしたフォルダーをまるまる渡しましょう。実行ファイルのみでは動かないので注意してください。

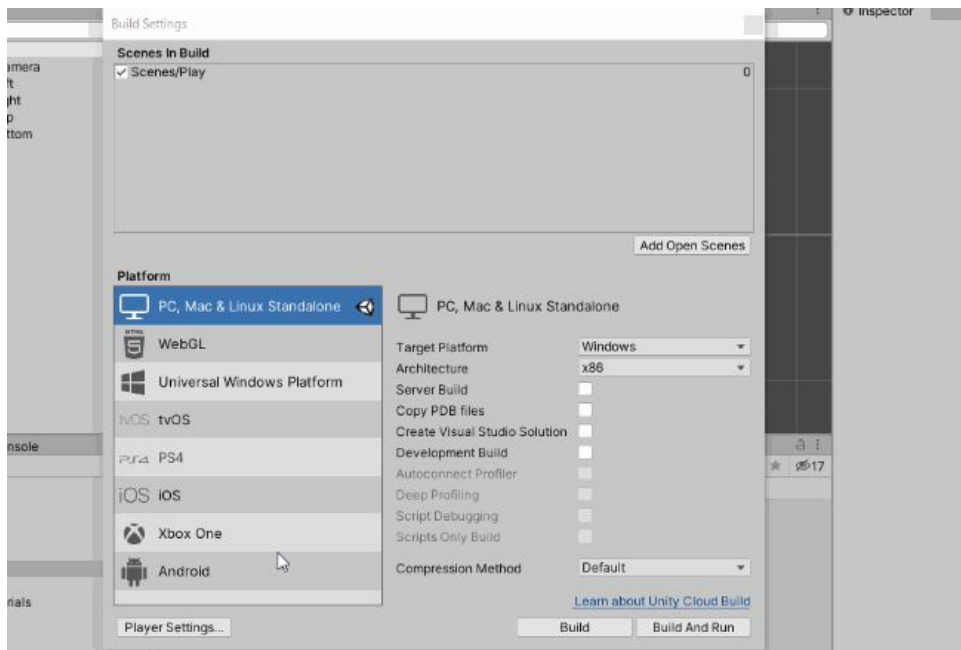


名前	更新日時	種類	サイズ
MonoBleedingEdge	2020/06/24 13:20	ファイル フォルダー	
Tutorial_Data	2020/06/24 13:20	ファイル フォルダー	
Tutorial.exe	2020/04/04 11:31	アプリケーション	625 KB
UnityCrashHandler32.exe	2020/04/04 11:27	アプリケーション	902 KB
UnityPlayer.dll	2020/04/04 11:33	アプリケーション拡張	19,524 KB



# プロダクトの基本設定

ビルドの設定はPlayer Settingsにより行います。Build Settingsの左下にあるPlayer Settings...を押したり、左上のメニューからEdit->Project Settings...を選択することでアクセスできます。



Player Settingsの上部にある項目では、会社名や製品名、アイコンの設定などができます。Player Settingsの上部にある項目では、会社名や製品名、アイコンの設定などができます。ここではこれ以上のことには触れませんが、Player Settingsでは画面比率や描画、コンパイルの設定など複数の項目があるので、必要に応じて設定を変更できます。

**Player**

Company Name: DefaultCompany

Product Name: Tutorial

Version: 0.1

Default Icon: None (Texture 2D) [Select]

Default Cursor: None (Texture 2D) [Select]

Cursor Hotspot: X 0 Y 0

項目	説明
Company Name	会社名を記入します。個人で制作する場合にも制作者の名前などに変えるべきです。
Product Name	製品名(ゲーム名)を記入します。これが実行ファイルの名前になります。
Version	バージョンを設定します。完成品は1.0から始まり、小さなアップデート(バグの修正など)には1.1などのように小数部分を増やし、大きなアップデート(機能追加など)には2.0などのように整数部分を増やすことが多いです。
Default Icon	ゲームのアイコンを設定します。何も設定しないとUnityのマークになります。
Default Cursor	マウスカーソルの画像を変更できます。

# WebGL




---

# ビルド

WebGLでビルドする場合もPC向けにビルドする場合とほぼ同様の手順でビルドできます。Build SettingsでプラットフォームをWebGLに変更してBuildまたはBuild And Runを実行しましょう。WebGL形式はWindows/Mac向けにビルドするよりも多くの時間が掛かります。エラーが出ないことを祈りながら待ちましょう。

Build And Runの場合には、ビルド後に既定のWebブラウザが開いてゲームが開始されます。Buildの場合には、ファイルのみが生成されます。index.htmlの記載内容を参考にwebサーバにアップロードすれば、ブラウザゲームとして公開可能です。

※環境によってはindex.htmlをGoogle Chromeで開いてもゲームを遊ぶことはできません。Webサーバにアップロードするか、ブラウザを変えるように言われたりします。Microsoft Edgeで開くとゲームプレイができるなどがあるそうです

名前	更新日時	種類	サイズ
 Build	2020/06/25 21:33	ファイル フォルダー	
 TemplateData	2020/06/25 21:33	ファイル フォルダー	
 index.html	2020/06/25 21:33	Chrome HTML Do...	1 KB

# 画面解像度の設定

ゲーム製品の設定も同様にPlayer Settingsで行います。WebGL版の場合には解像度を固定値で指定してあげる必要があります。Resolution and Presentationの項目でDefault Canvas Width/Heightを指定しましょう。この値によって画面比率も変わるので注意しましょう。

