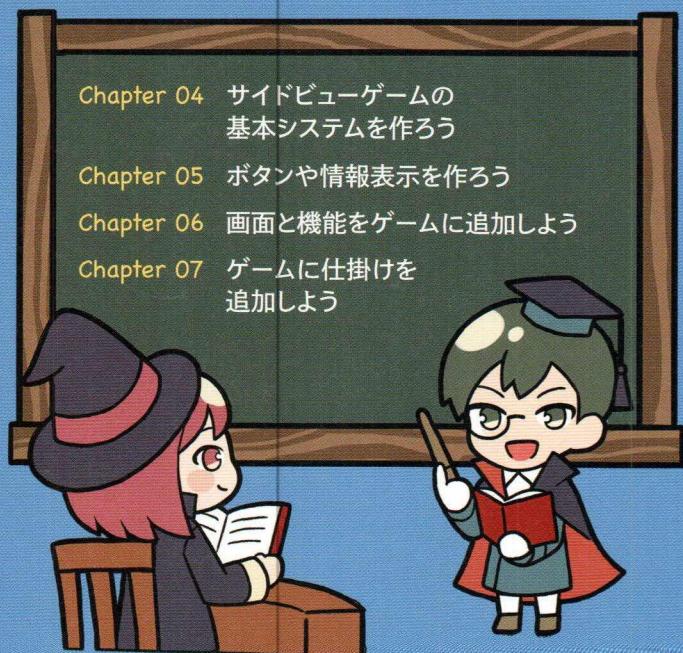


# 第2部

## サイドビューゲームを作ろう

第2部では第1部で作ったプロジェクトをもとにして、サイドビューゲームの作り方を解説します。第2部前半でサイドビューゲームに必要なシステムを作り、後半にゲームを楽しくする仕掛けを使っていきます。

- Chapter 04 サイドビューゲームの基本システムを作ろう
- Chapter 05 ボタンや情報表示を作ろう
- Chapter 06 画面と機能をゲームに追加しよう
- Chapter 07 ゲームに仕掛けを追加しよう





## サイドビューゲームの基本システムを作ろう

ここからは、「プレイヤーを表示し、キーボード操作で移動させ、ゴールに到達する」というサイドビューゲームシステムを順を追って作っていきます。そのために、第1部で作ったプロジェクトを更新していきます。

簡単に、ここから作っていくゲームの要素をまとめておきましょう。



- ルール：右端のゴールにたどり着くとゴール
- 敵と障害：穴に落ちるとゲームオーバー
- 干渉と変化：ジャンプで穴を飛び越える
- 報酬：穴を乗り越えてゴールにたどり着く



### 4.1 サイドビューゲームってどんなゲーム？

**サイドビュー**はゲームの世界を真横からのアングルで見たゲームシステムのことです。真横から見たアングルなので、移動は「右」と「左」の2方向になります。また縦方向は高さを表すため「ジャンプする」というアクションが可能ですね。移動が横だけなので判断がシンプルにでき、比較的アクションゲームに向いているシステムです。

最初に、サイドビューゲームとしてこの章でどのようなものを作っていくのか確認しておきましょう。プレイヤーキャラを左右に動かし、画面左から右にあるゴールを目指すサイドビューの「ラン&ジャンプ」ゲームです。またゴールすると、画面にゴール表示のUIを表示します。

## 使うゲームオブジェクトとスクリプトを考えてみよう

サイドビューゲームを作るために、以下のような機能を持ったゲームオブジェクトとそれに関するスクリプトを書いていきます。

### ◆ プレイヤーキャラ



プレイヤーが操作するゲームキャラクターです。左右への移動とジャンプができるようにします。そして移動中やジャンプなどのとき、キャラクターにアニメーションを付けてみましょう。

### ◆ 地面とブロック



プレイヤーが乗ってその上を移動することができる地面とジャンプで飛び乗れる足場のブロックです。前章では地面を作成しましたが、ここではその他の形のブロックを作りましょう。

### ◆ ゴールとゲームオーバー



ゲームステージの右端にゴールとなるゲームオブジェクトを設置して、そこに触れるとステージクリアとなる仕組みを作りましょう。さらに、地面下に落下することでゲームオーバーになる仕組みもあります。これらができればゲームらしくなりますね。

## ◆ ステータス表示とリストアート

# GAME OVER



RESTART

ゲームの開始、ゲームオーバー、ゲームクリアのときに画像を表示しましょう。またゲームオーバーになったときにゲームをリストアートさせて、最初からプレイできるようにしていきます。



## 4.2

## まずはサンプルゲームを実行してみよう

### プロジェクトを Unity Hub に追加しよう

以下のURLが、サイドビューゲーム「JEWELRY HUNTER」のサンプルプロジェクトです。ダウンロードして、圧縮ファイルを解凍しましょう。

- <https://www.shoeisha.co.jp/book/download/3600/read>



ダウンロードしたら、サンプルプロジェクトをUnityで開いてみましょう。新規作成ではなく、「すでにあるプロジェクトをUnityで開く」には、Unity Hubのプロジェクトタブにある「リストに追加」ボタンをクリックしてください。

プロジェクトフォルダー「JewelryHunter」を選択して、右下の「開く」ボタンをクリックします。これで、Unity Hubのプロジェクトリストに追加されます。



## サンプルゲームを確認しよう

プロジェクトを開いたら、まず、この章で作る横スクロールゲームのサンプルを確認してみましょう。Scenes フォルダーにある「Title」というシーンを開いて、ツールバーの実行ボタンでゲームを実行します。

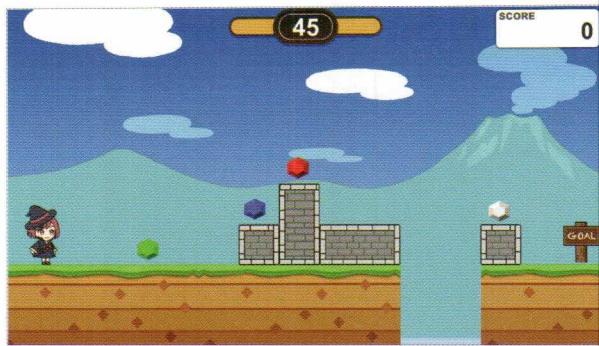


追加できたら、以降はこのリストをクリックすることでプロジェクトが開きます。リスト追加後にプロジェクトフォルダーを移動してしまうと開けなくなってしまいます。注意しましょう。

実行を開始すると、まずゲームのタイトル画面が表示されます。

ここで、「START」ボタンを押すと、ゲームが開始されます。「JEWELRY HUNTER」は宝石を集めてスコアを競うサイドビューのジャンプアクションゲームです。

ゲームを開始すると、「GAME START」という画像が1秒間、画面中央に表示されます。



ゲーム中は、画面中央上に「カウントダウンタイマー」が、右上に「スコア表示」が表示されます。カウントダウンタイマーは60秒からカウントダウンしていきます。



ステージに配置された宝石を取ると、それがスコアになって右上の数字が増えていきます。



画面右端のゴールにたどり着くとステージクリアです。  
[NEXT] ボタンを押すことで次のステージに進めます。



それでは、このサンプルと同じゲームシステムのサイドビューゲームを作っていきましょう。第1部の続きから進めていきます。ダウンロードしたサンプルプロジェクトを開いている場合は一度閉じて、第1部で途中まで作成したプロジェクトを開いてください。

Tips

### 完成データのダウンロード

この章で作成するプロジェクトの完成データは、以下のアドレスからダウンロードできます。

- <https://www.shoeisha.co.jp/book/download/3604/read>



## 4.3

# ゲームステージを作ろう

それでは、いよいよ実際にサイドビューのゲーム画面を作っていきます。

Chapter 3までの作業で、以下のような状態になっているはずです。もし、以下のようになっていなければChapter 2と3に戻って確認してみてください。

## Chapter 3までを振り返ろう

### ◆ シーン

「Stage1」という名前でゲーム画面のシーンがAssets/Scenes フォルダーに保存されています。

### ◆ 背景

背景として「back」という画像がシーンの中央に配置されています。

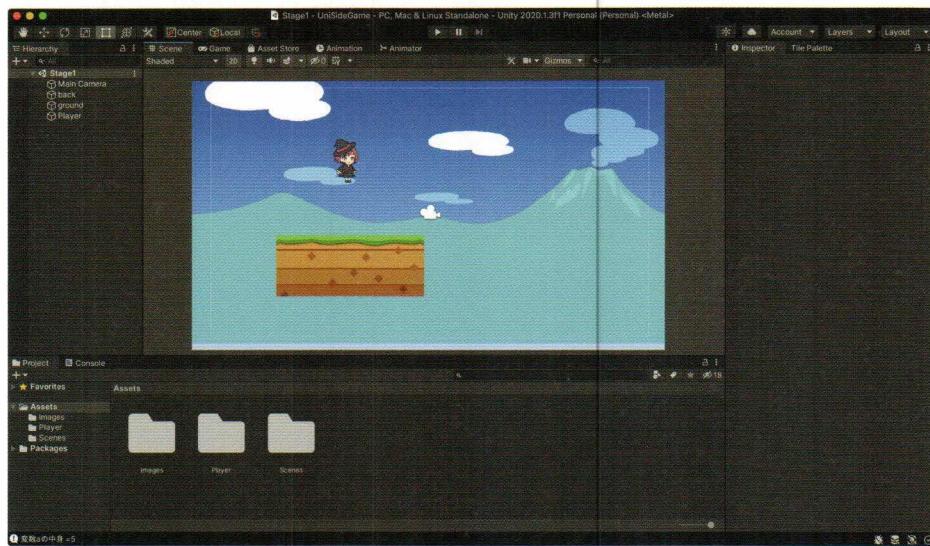
### ◆ 地面

地面として「ground」という画像がシーンに配置されています。また、Sprite Rendererコンポーネントの「Order in Layer」が「2」に設定されています。Box Collider 2Dコンポーネントがアタッチされていることも確認しておきましょう。

### ◆ プレイヤーキャラクター

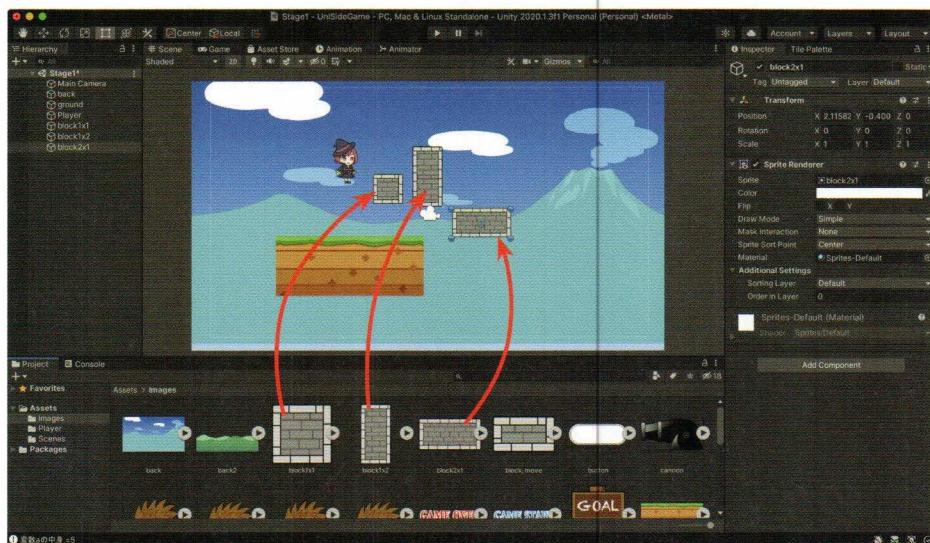
キャラクター画像として「player\_stop」がシーンに配置されています。Rigidbody 2Dコンポーネントがアタッチされ、「Freeze Rotation Z」がチェックされていることを確認してください。

さらに、Sprite Rendererコンポーネントの「Order in Layer」が「3」に設定されていて、Capsule Collider 2Dコンポーネントがアタッチされていることと、Player Controller (Script)がアタッチされていることも確認しておきましょう。



## 地面ブロックを作ろう

地面はできているので、次はジャンプで飛び乗れる「足場」の地面ブロックを作りましょう。



地面ブロック用の画像は形が違う3種類を用意しております。3種類をドラッグ & ドロップで、シーンビューに配置してください。今のところはシーンビューのどこでもかまいません。

作り方は地面とまったく一緒です。それぞれ以下の設定とコンポーネントのアタッチを行ってください。

## ◆ Sprite Renderer コンポーネント

「Order in Layer」を「2」に設定します。

 「表示の優先順位を知ろう」 [33 ページ](#)

## ◆当たり

Box Collider 2D コンポーネントをアタッチします。

 「ゲームオブジェクトに当たりを付けよう」 [53 ページ](#)

## ゲームオブジェクトをグループ分けする仕組み（レイヤー）

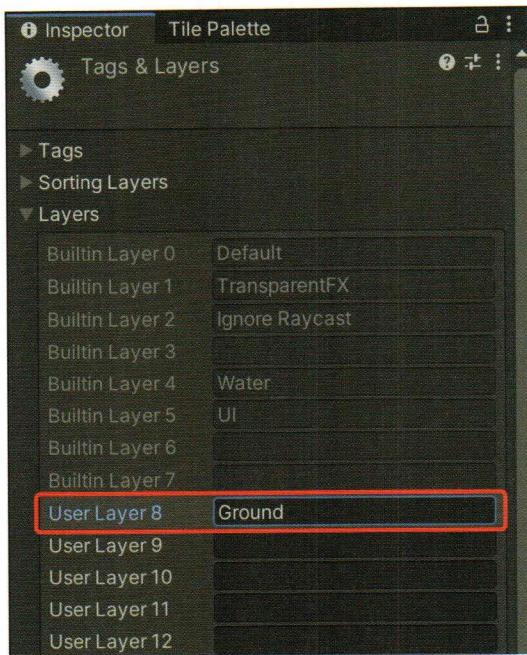
Unityには、ゲームオブジェクトを区別するための仕組みがいくつかあります。ここでは地面とブロックをグループ分けして「プレイヤーが乗れる足場」として区別できるようにしましょう。

ところでなぜ区別が必要かというと、プレイヤーがジャンプするときに「地面の上にいるからジャンプできる」という条件を付けたいからです。

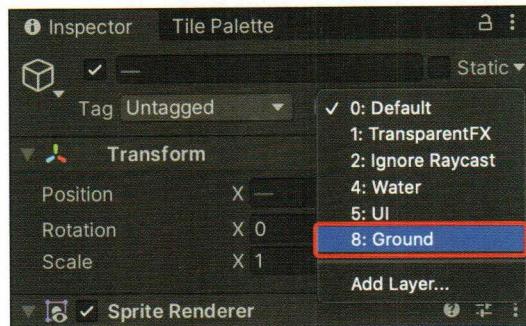
ここでは、区別を付けるために**レイヤー**（Layer）という「ゲームオブジェクトをグループ化して扱う」ための仕組みを使います。地面とブロックを「Ground」という名前でグループ化し、プレイヤーキャラクターがGroundレイヤーに接触しているときにだけジャンプができるようにしていきます。



まず、地面（またはブロック）を選択して、インスペクタービューの右上にある「Layers」というプルダウンメニューを見てみましょう。すると、すでに用意されているいくつかのLayerが見えます。これからここに「Ground」という新しいレイヤーを追加します。**[Add Layer…]** を選択してください。



するとインスペクタービューの表示が切り替わり、現在登録されているLayerのリストが表示されます。Layer 0からLayer 7まではすでにUnityにより使用されているので、Layer 8以降に「Ground」と入力してください。



もう一度ゲームオブジェクトを選択し、追加されたGroundをプルダウンメニューから選択します。

ここまでできたら、地面とすべてのブロックを同じようにGroundレイヤーに設定してください。

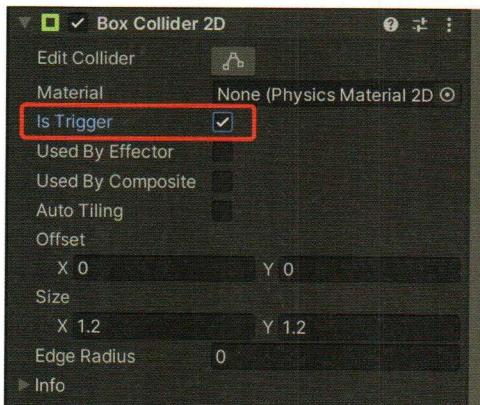
## ゴールを配置しよう

次はゴールの配置です。ゴール用の画像をシーンビューにドラッグ＆ドロップして配置しましょう。



Sprite Renderer の「Order in Layer」を「2」、つまり地面と同じ優先順位にしておきます。プレイヤーキャラクターは「Order in Layer」を「3」にしたので、地面とゴールは常にプレイヤーキャラクターの下（背面）に表示されることになります。

## ゴールの判定を設定しよう



ゴールに接触したことを判定するためには、Box Collider 2Dを付けて「Is Trigger」にチェックを付けておきましょう。

これまで、コライダーは「ゲームオブジェクトどうしを物理的に接触させる」ために付けていましたが、「Is Trigger」にチェックを付けることで物理的な当たりが発生しなくなり、すり抜けるようになります。しかし「当たった」というイベントはスクリプトで受け取ることができます。これを使ってゴールの判定を行います。

実際のゴール判定処理はこのあと、スクリプトで作っていきます。

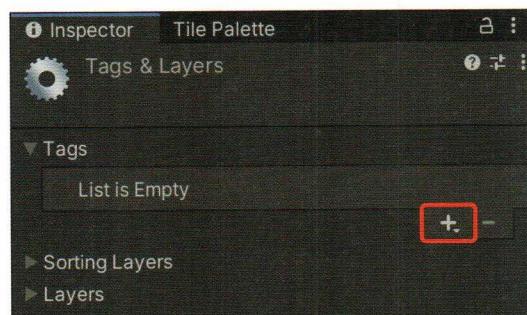
## ◆ ゲームオブジェクトを区別する仕組み（タグ）

ゴールに接触したということを知るために、**タグ**(Tag) という機能を使います。先ほど使ったLayerは「ゲームオブジェクトをグループ化する」というものでしたが、タグは「ゲームオブジェクトに文字を付けて区別する」仕組みです。

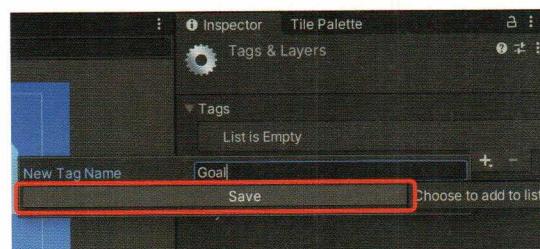
まずはシーンビューの「goal」を選択してインスペクタービューを見ると、「Tag」というプルダウンメニューがあります。プルダウンメニューを選択してみると、「Respawn」や「Finish」など、いくつかのタグがすでにあることがわかりますね。



今回はこれらのタグは使わず、新しくタグを追加します。タグは、一番下の [Add Tag…] を選択することで自由に追加することができます。



インスペクタービューの表示が切り替わり、一番上に「Tags」というメニューが表示されています。その下にある [+] ボタンをクリックすることでタグの追加ビューが開きます。



追加したいタグ名を入力して、[Save] ボタンをクリックすれば保存されます。ここではゴール用のタグとして「Goal」を追加しました。



最後に、作ったGoalタグをゴール用のゲームオブジェクトに設定しましょう。



## 4.4 ゲームオブジェクトを再利用しよう

これで、地面となるゲームオブジェクトができました。

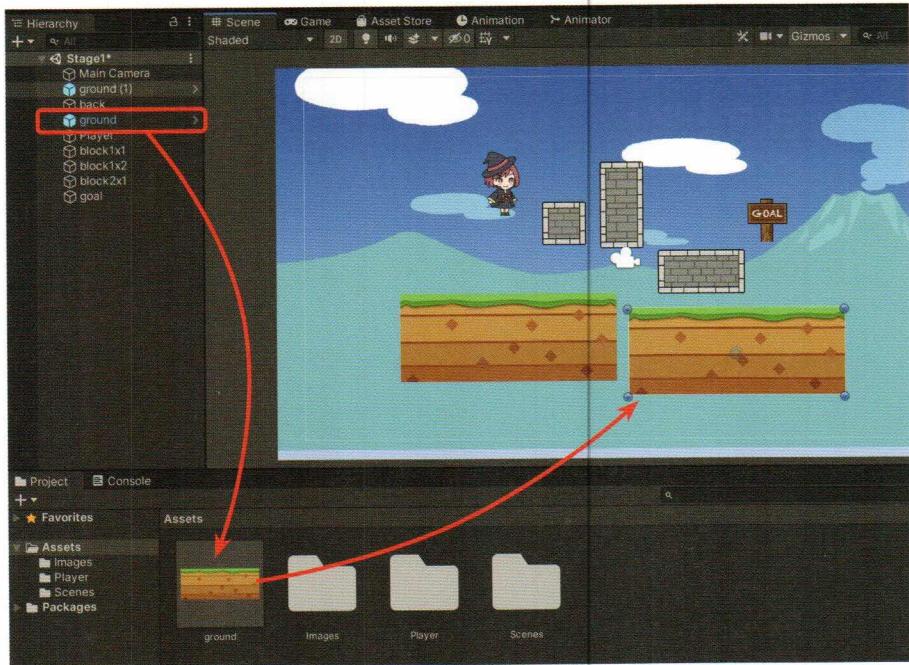
ここからはこの地面をたくさん作って並べ、ゲームの「地形」を作っていきます。しかし、1つ地面を作るたびに、いちいち「画像を配置して、コライダーを付けて……」としていてはとても手間がかかりますね。

Unityには、プレハブ(Prefab)という「ゲームオブジェクトを複製する」ための便利な仕組みがあるので、それを使うことにしましょう。

### プレハブの作成

プレハブを作るのは簡単です。プレハブ化したいゲームオブジェクトをヒエラルキービューで選択して、プロジェクトビューにドラッグ&ドロップするだけです。

すると、プロジェクトビューに、周囲が濃いグレーのアイコンができるはずです。これがプレハブです。同時にヒエラルキービューのゲームオブジェクトは青いアイコンに変化しましたね。プレハブから作られたゲームオブジェクトのアイコンは、ヒエラルキービュー上で青くなります。



ここまでできたら、地面とすべてのブロック、ゴールを同様にプレハブ化しておいてください。

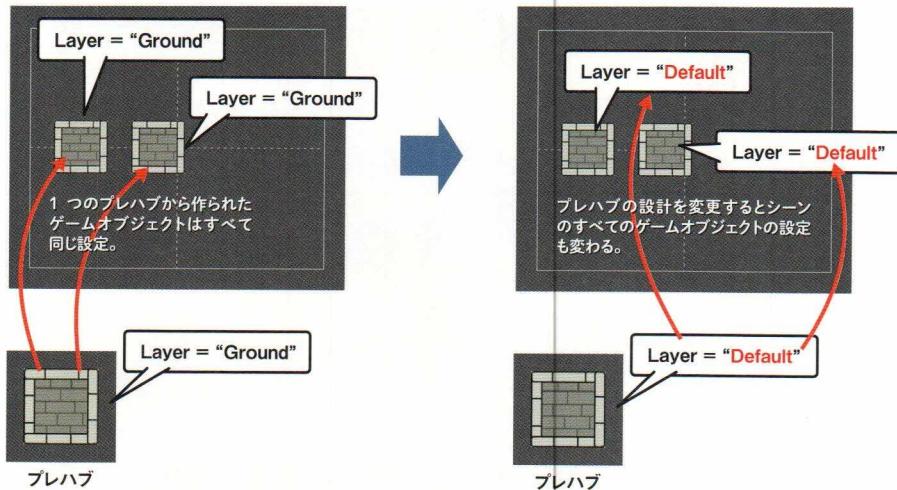
それでは、プロジェクトビューにできたプレハブアイコンをシーンビューにドラッグ＆ドロップして配置してみましょう。配置されたゲームオブジェクトのインスペクタービューを見ると、すでにBox Collider 2Dが付いた状態になっていますね。今後はプロジェクトビューにあるプレハブアイコンを配置することで、簡単に同じゲームオブジェクトが作れるというわけです。

### ◆ プレハブと「コピーしたゲームオブジェクト」の違い

ところで、プレハブと「コピーしたゲームオブジェクト」は何が違うのでしょうか？

例として、ブロックを数十個シーンに配置したあとで、画像や当たりの範囲、サイズなど、コンポーネントを変更したくなった場合を考えてみましょう。

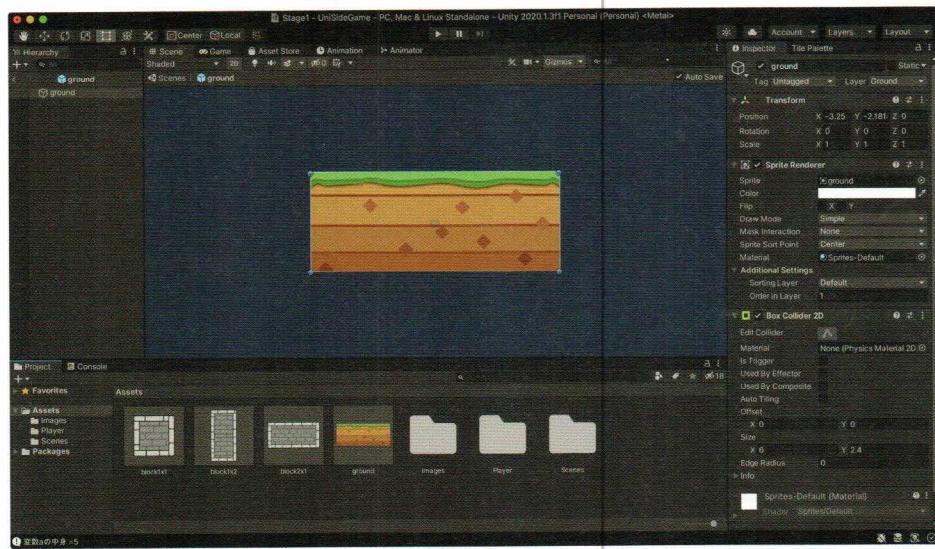
コピーしたゲームオブジェクトが10個あった場合は、10個それぞれを同じように変更しなければいけません。しかしこれがプレハブなら、プロジェクトビューにあるデータを1個変更するだけでシーンに配置されているゲームオブジェクトをすべて同じように変更することができます。



ただし、シーン上に配置されたゲームオブジェクトのコンポーネントを変更した場合、元のPrefabを変更してもそのコンポーネントの値は変わりません。つまり、Prefabの初期値はすべての配置オブジェクトに反映されますが、個別に変更したものは変化しないので、全変更したり、個別にカスタムが簡単にできるというわけです。

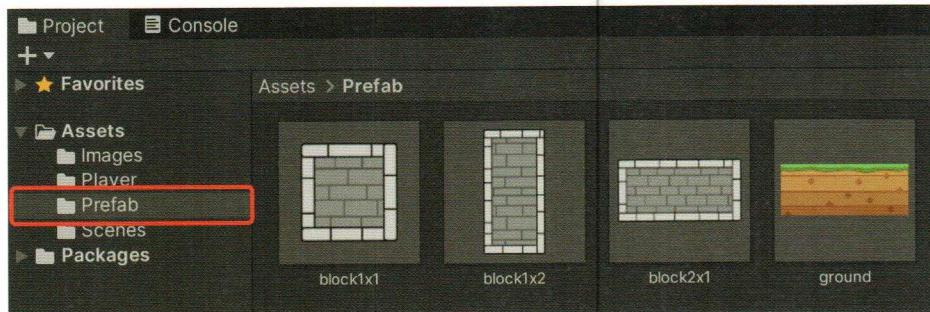
## Prefabを編集しよう

ヒエラルキーからドラッグ＆ドロップして作ったPrefabを編集するには、プロジェクトビューのPrefabアイコンをダブルクリックしてください。またはヒエラルキーで表示されているゲームオブジェクトの右端にある [>] ボタンをクリックします。すると、シーンビューがPrefabの編集画面に変わります。

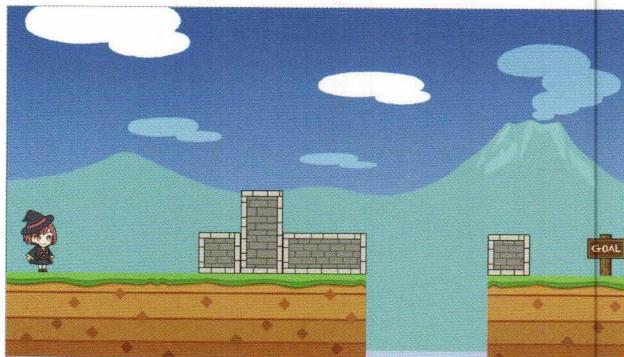


インスペクタービューにはそのプレハブのコンポーネントが表示されており、自由に編集することができます。編集が終わったら、ヒエラルキービューの左上にある戻る ([<]) ボタンをクリックすることで、元の画面に戻ります。

プレハブを整理するために、Prefab フォルダーを作って、その中に作成したプレハブデータを入れておくようにしましょう。今後、作成したプレハブはこのフォルダーの中に入れるようにします。



## プレハブを配置して地面を作ろう



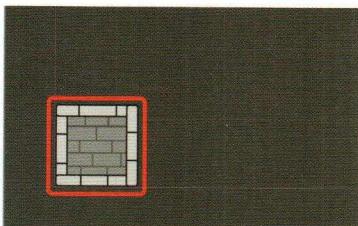
それでは、地面とブロックのプレハブを使って地面の配置を作ります。

次の図のように、ゲーム画面の左端から右端まで平らな地面を作りましょう。さらに真ん中あたりに1カ所、段差からの地面のない穴を作っておきましょう。この穴がプレイヤーへの障害になります。

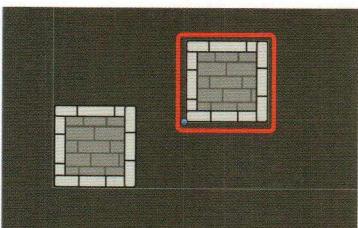
それから、キャラクターは画面の左端に配置しておきましょう。

### ◆ ゲームオブジェクトをそろえて並べる

ところで、今回作った地面のように、「ゲームオブジェクトをきっちりそろえて並べたい」とき便利な機能があるので紹介しておきましょう。

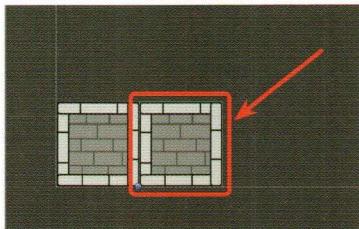


まず、ゲームオブジェクトを1つ左端下に置いてください。



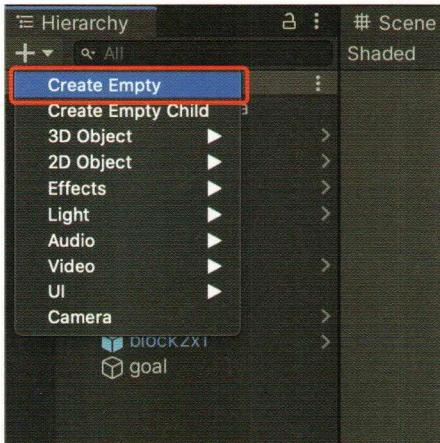
置けたら、その右側にもう1つゲームオブジェクトを置きましょう。位置は適当でかまいません。そして新しく置いたゲームオブジェクトを選択状態にして、マウスカーソルをそのブロックの上に持ってきます。

マウスボタンをクリックせずに、キーボードの [V] キーを押してみましょう。その状態でマウスカーソルを動かすと、カーソルの動きと合わせて、選択されているゲームオブジェクトの四隅と中央に青いポイントが1カ所だけ表示されます。

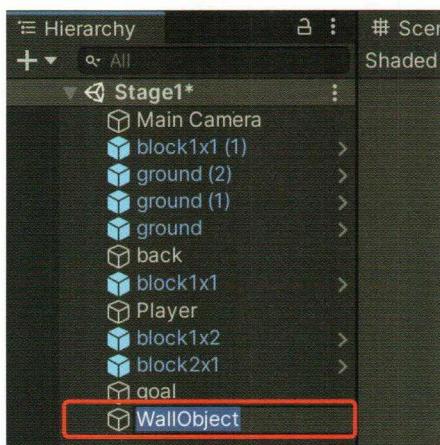


その状態でゲームオブジェクトをつかんで動かしてみてください。青いポイントが近くのゲームオブジェクトの角に吸い付くように移動しましたね。この方法でゲームオブジェクトを配置すれば、キレイに並べて配置することができます。

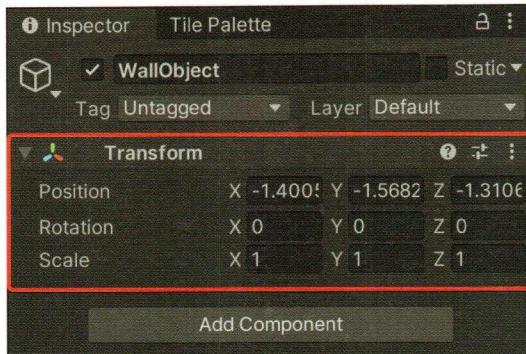
## 見えない当たりを作ろう



画面の両端からプレイヤーキャラクターが落ちないように、透明な壁、つまりコライダーを作成しておきましょう。まずはコライダーを作ります。そして、ヒエラルキー視覚左上の[+]ボタンから「Create Empty」を選択します。



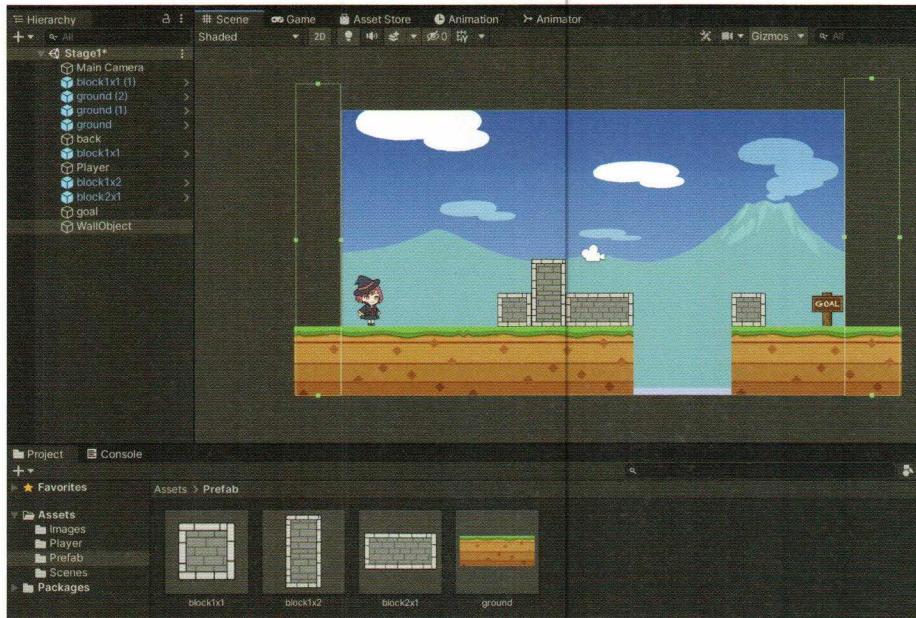
ゲームオブジェクトがシーンビューに追加されます。ここでは名前を「WallObject」としています。壁の当たりを設置するためのゲームオブジェクトなので、わかりやすければ名前は何でもかまいません。



このゲームオブジェクトのインスペクタービューを見てください。位置や回転、スケールを設定するTransformコンポーネントだけがアタッチされた状態のゲームオブジェクトになっています。

このように、「Transformだけがアタッチされた空のゲームオブジェクト」は今後いろいろな場面で利用します。作り方をぜひ覚えておきましょう。

このゲームオブジェクトBox Collider 2Dを2つアタッチしましょう。アタッチしたら、それぞれこののような形に調整してください。

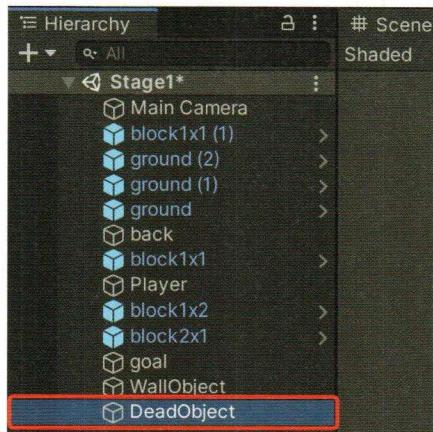


## ゲームを実行しよう

ここまでできたら、ゲームを実行して動作を確認してみましょう。キャラクターは画面の両端の先に進めず、落ちなくなっているはずです。

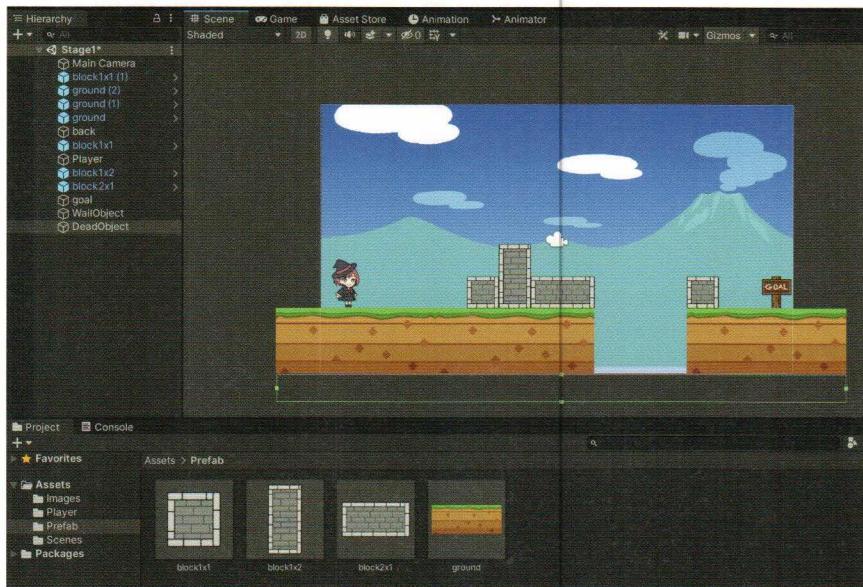
## ゲームオーバーの当たりを作ろう

次は、ゲームオーバーにする仕組みを作りていきましょう。現在は穴に落ちたら復帰できずにゲームを進められなくなります。「Dead」というタグが付いたゲームオブジェクトを作り、それに接触したらゲームオーバーになるようにしていきます。



先ほどと同じように、[Create] メニューから [Create Empty] を選択して空のゲームオブジェクトを作り、名前を「DeadObject」に変更しておきましょう。

DeadObject に Box Collider 2D をアタッチし、このような形に調整します。

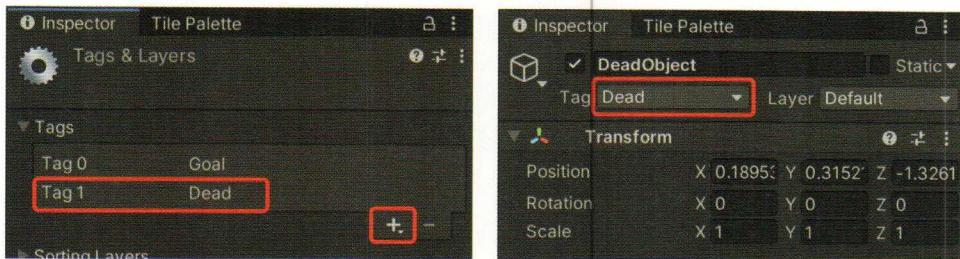


ゴール同様にイベント当たりにするので、Box Collider 2D コンポーネントの「In Trigger」にチェックを付けておいてください。

## ◆ ゲームオーバー用のタグ設定

それでは、「Dead」タグを追加しましょう。[+] ボタンを押して「Dead」タグを追加し、DeadObjectにDeadタグを設定します。

参照 「ゲームオブジェクトを区別する仕組み（タグ）」 [99 ページ](#)



ここではひとまず判定までの仕組みを作りました。ゴール後、ゲームオーバー時のゲームの流れはこのあと作っていきます。



## 4.5

## プレイヤーキャラクターを作ろう

以上でゲームステージができました。次はプレイヤーキャラクターを作り込んでいきましょう。第1部までの作業で、パソコンの左右矢印キーで移動できる状態になっていますね。

### 「Player」タグを設定しよう



まずは、プレイヤーキャラクターをシーン上で区別できるように「Player」タグを設定しておきましょう。「Player」タグは最初から用意されているので、ヒエラルキービューでPlayerゲームオブジェクトを選択して、「Player」タグを設定しておきましょう。

## ジャンプできるようにしよう

次にプレイヤーキャラクターがジャンプできるようにしていきましょう。ここでは、パソコンの [スペース] キーを押すことでジャンプできるようになります。

それでは、プレイヤーキャラクターをジャンプさせるスクリプトを PlayerController に追記しましょう。以下がスクリプトの内容です。変更する箇所をハイライトしています。

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

public class PlayerController : MonoBehaviour
{
    Rigidbody2D rbody; // Rigidbody2D 型の変数
    float axisH = 0.0f; // 入力
    public float speed = 3.0f; // 移動速度

    public float jump = 9.0f; // ジャンプ力
    public LayerMask groundLayer; // 着地できるレイヤー
    bool goJump = false; // ジャンプ開始フラグ
    bool onGround = false; // 地面に立っているフラグ

    // Start is called before the first frame update
    void Start()
    {
        // Rigidbody2D を取ってくる
        rbody = this.GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        // 水平方向の入力をチェックする
        axisH = Input.GetAxisRaw("Horizontal");
        // 向きの調整
        if (axisH > 0.0f)
        {
            // 右移動
            Debug.Log("右移動");
            transform.localScale = new Vector2(1, 1);
        }
        else if (axisH < 0.0f)
        {
            // 左移動
            Debug.Log("左移動");
            transform.localScale = new Vector2(-1, 1); // 左右反転させる
        }
    }
}
```

```

// キャラクターをジャンプさせる
if (Input.GetButtonDown("Jump"))
{
    Jump(); // ジャンプ
}

void FixedUpdate()
{
    // 地上判定
    onGround = Physics2D.Linecast(transform.position,
        transform.position - (transform.up * 0.1f),
        groundLayer);

    if (onGround || axisH != 0)
    {
        // 地面の上 or 速度が 0 ではない
        // 速度を更新する
        rbody.velocity = new Vector2(speed * axisH, rbody.velocity.y);
    }

    if (onGround && goJump)
    {
        // 地面の上でジャンプキーが押された
        // ジャンプさせる
        Debug.Log(" ジャンプ！ ");
        Vector2 jumpPw = new Vector2(0, jump); // ジャンプさせるベクトルを作る
        rbody.AddForce(jumpPw, ForceMode2D.Impulse); // 瞬間的な力を加える
        goJump = false; // ジャンプフラグを下ろす
    }
}

// ジャンプ
public void Jump()
{
    goJump = true; // ジャンプフラグを立てる
    Debug.Log(" ジャンプボタン押し！ ");
}

```

それではこのスクリプトの中身を見ていきましょう。

### ◆ 変数

まず、変数が4つ追加されています。`float`型の`jump`変数はジャンプ力を設定する変数です。あとでジャンプ処理を作るときに使います。なお、Unityエディターからこの変数を変更できるように`float`の前に`public`を付けています。

`groundLayer`は先ほど地面用に追加したGroundレイヤーを設定しておくための変数です。レイヤーは`LayerMask`という型で表されます。これもあとからUnityエディターで設定できるように`public`を付けています。

参照 「ゲームオブジェクトをグループ分けする仕組み（レイヤー）」 [96 ページ](#)

`goJump` はジャンプキー（[スペース] キー）が押されたことを保存しておくフラグです。

`onGround` はキャラクターが地面に接触していることを示すフラグです。`goJump` と `onGround` が `true` の場合にジャンプが可能になります。

### ◆ Update メソッド

`Update` メソッドにジャンプの条件を追記しています。ここでは「条件が成り立たない」こと（不成立）がありませんので、`else { ... }` は必要ありません。このように `else` は必要なれば省略できます。

それでは、`if` 文の条件を見てみましょう。

```
// キャラクターをジャンプさせる
if (Input.GetButtonDown("Jump"))
```

ここには `Input` クラスの `GetButtonDown` メソッドが入っています。`Input` クラスは入力系を扱うクラスでしたね。`GetButtonDown` は指定したキーが押されたかどうかを `bool` 型で返すメソッドです。

また、引数には文字列で "Jump" を渡しています。Unity 標準ではキーボードのスペースキーがジャンプボタンに割り当てられており、ここでは「スペースキーが押されたら、`Jump` メソッドを呼んで、その中で `goJump` 変数に `true` を入れる」という処理になっています。

参照 「いろいろな装置からの入力をサポートする `Input` クラスと `Input Manager`」

[76 ページ](#)



Tips

### Input クラスのいろいろな入力メソッド

`Input` クラスにはこの他にもいろいろな入力を扱うメソッドがあります。それらをいくつか紹介しましょう。

#### ● `GetKeyDown` / `GetKey` / `GetKeyUp`

```
bool down = Input.GetKeyDown(KeyCode.Space);
bool press = Input.GetKey(KeyCode.Space);
bool up = Input.GetKeyUp(KeyCode.Space);
```

GetKey 系のメソッドは3つあります。それぞれKeyCode型の引数を取り、引数で指定したキーボードのキーが押されたとき／押されっぱなし／離されたときを検知します。今回は [スペース] キーが押されれば、true が返されます。

#### ● GetMouseButtonDown / GetMouseButton / GetMouseButtonUp

```
bool down = Input.GetMouseButtonDown(0);  
bool press = Input.GetMouseButton(0);  
bool up = Input.GetMouseButtonUp(0);
```

GetMouseButton 系のメソッドは3つあります。先ほどのGetKey 系のメソッドと同様に、マウスのボタンが押されたとき／押されっぱなし／離されたときを検知します。引数には0／1／2の数値が入り、それぞれ左クリック／右クリック／中央（スクロールホイール）クリックを意味します。またGetMouseButton 系のメソッドはスマートフォンのタッチパネルにも対応しています。

#### ● GetButtonDown / GetButton / GetButtonUp

```
bool down = Input.GetButtonDown("Jump");  
bool press = Input.GetButton("Jump");  
bool up = Input.GetButtonUp("Jump");
```

GetButton 系のメソッドは3つあります。このメソッドはいろいろな入力機器のボタンが押されたとき／押されっぱなし／離されたときを検知します。引数には文字列が入りますが、それはInput Managerで設定してあるものです。

具体的なボタンは、Input Managerで「"Jump"」の場合はキーボード上で [スペース] キーとなる」というように指定します。そして "Jump" のように同じ名前の設定を複数作っておくことで、それら複数のボタンをゲーム内で「ジャンプのためのボタン」として割り当てられるようになります。

参照 → 「いろいろな装置からの入力をサポートする Input クラスと Input Manager」  
76 ページ

#### ● GetAxis / GetAxisRaw

```
float axisH = Input.GetAxis("Horizontal");  
float axisV = Input.GetAxisRaw("Vertical");
```

GetAxis 系メソッドは2つあります。このメソッドはいろいろな入力機器の仮想的な軸入力を取ってくれます。パソコンなら矢印キー、ゲームコントローラーならアナログス

ティックです。引数には文字列が入りますが、それはInput Managerで設定してあるものです。

`GetAxis`と`GetAxisRaw`の違いは、戻り値に補完がかかるかどうかです。`GetAxisRaw`メソッドは-1、0、1の3つの値だけが返りますが、`GetAxis`メソッドは-1～0～1の連続した値が返されます。これにより、ゲームコントローラーのアナログスティックで操作した場合、アナログスティックの傾きによって速度が変化するように操作できます。キーボードの場合は一定時間で徐々に値が変化します。

### ◆ Jump メソッド

`Jump`メソッドは`goJump`フラグに`true`をセットするだけのメソッドです。また`public`を付けて外部からも呼べるようにしています。これは後ほどタッチスクリーンによる操作に対応させるためのものです。

### ◆ FixedUpdate メソッド

`FixedUpdate`メソッドでは、まず`Physics2D`コンポーネント（クラスもあります）の`Linecast`メソッドを使い、`groundLayer`変数に設定したレイヤーに接触しているかを判断しています。



#### レイヤーとの接触を感知する Linecast メソッド

`Linecast`メソッドは、指定した2点をつなぐラインにゲームオブジェクトが接触しているかを調べ、`bool`型で`true`か`false`を返すメソッドです。1番目の引数が始点、2番目の引数が終点、3番目の引数が対象となるレイヤーの指定です。

ここでは、始点がゲームオブジェクト（プレイヤーキャラクター）の現在位置となり、終点は

```
transform.position - (transform.up * 0.1f)
```

という指定になっています。`transform.up`というのはベクトルとして、(x=0, y=1, z=0)を指定したものです。つまり「Y軸に対して0.1下がった地点」ということになります。



キャラクターのピボット（基準点）は足元にしているので、ラインはこのようにキャラクターの足元から真下に0.1伸びていることになります。このラインが「3番目の引数に指定したレイヤー」のゲームオブジェクトに触れることで`onGround`が`true`になり、触れていなければ`false`になります。

また、その後`if`文で速度を更新する条件を設定しています。これはどんな働きをしているのでしょうか。

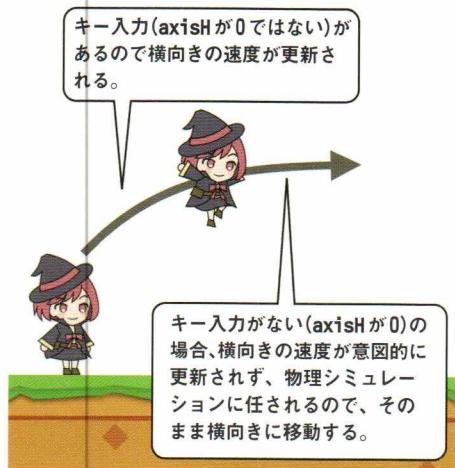
ジャンプ中に左右キーを押すと空中で左右に移動させることができます。空中にいるときに左右キーを離すとその位置から真下に落下してしまいます。というのも、左右移動のキー入力値が0になったために横向きの速度も0になってしまいます。

これを防ぐために、入力による速度更新に「地面上にいる、または入力が0ではない」という条件を付けているというわけです。

#### 速度更新に条件をつけない場合



#### 速度更新に条件を付ける場合



この条件付けにより、空中でキー入力がない場合、入力による速度の更新（未入力により0になる）は行われなくなり、真下に落下することはなくなります。

`if`文では同時に複数の判断を行うことができます。「`|`」(縦棒) が2つ並んだ演算子は「この左右にある条件のどちらかが`true`なら、全体が`true`になる」という意味です。主な論理演算子には以下のようなものがあります。

- `&& : &` (アンド) を2つつなげます。条件すべてが`true`であれば、`true`になります
- `|| : |` (縦棒) を2つつなげます。条件のいずれかが`true`であれば、`true`になります

ここではジャンプさせる条件として、`onGround`と`goJump`という2つの変数をチェックして、両方が`true`ならジャンプ処理を行っています。

ジャンプさせるためには、`Rigidbody`クラスの`AddForce`メソッドを呼んでいます。`AddForce`メソッドは`Rigidbody 2D`がアタッチされているゲームオブジェクトに力を加えるためのメソッドです。力を加えられたゲームオブジェクトは物理法則に従って動くことになります。その際、力はベクトルで表されます。

この場合は「上方向に`9.0f` (`jump`変数の値) の力を加える」、つまり真上にジャンプするということになります。さらに「どのような力を加えるのか」を、`AddForce`メソッドの2番目の引数で指定しています。`ForceMode2D.Impulse`というのは「瞬間的な力を加える」という指定です。

## レイヤーを設定しよう



ここまでできたら、Unityエディターに戻ってパラメーターの設定をしましょう。`groundLayer`変数を`public`で定義したので、`PlayerController(Script)`で「`Ground Layer`」というプルダウンメニューが

表示されています。ここから、「`Ground`」を選択してください。