基本テキスト
1章

Godotエンジンの基礎知識

ver.1.0.0

1-1　プロジェクトと画面構成
1-2　画面と基本操作
1-3　シーンとノード
1-4　その他の要素
1-5　オブジェクト指向とツリー

この章ではGodotエンジンの基本要素やエディタの基礎について説明しています。特にノードとシーンについては最も重要な要素となります。
書かれていることを記憶する必要はありませんが、ざっと目を通しておくと実際にゲーム制作をする際に理解しやすくなります。

バージョン4.2.1をベースに作成し、4.3.0にも対応しています。

1-1 プロジェクトと画面構成

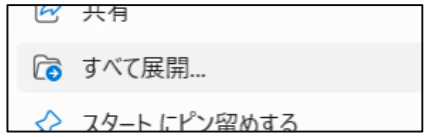
- ダウンロードと準備
- プロジェクトの開始
- プロジェクトのレンダラー設定
- プロジェクトのインポート
- プロジェクトのエクスポート
- プロジェクトのエクスポート（１）Windows
- プロジェクトのエクスポート（２）Web
- ファイル管理

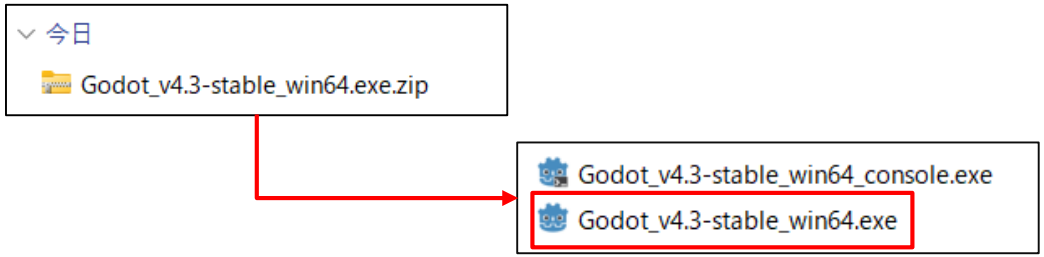
ダウンロードと準備

まずは、Godotエンジンをダウンロードします。
https://godotengine.org/

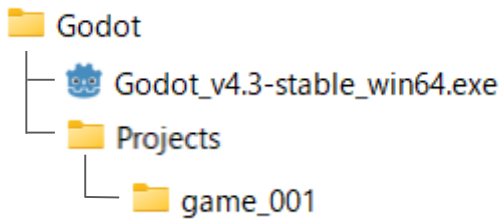
アクセスしているPCによって、そのPCに合ったものをダウンロードできるようになっています。

ダウンロードしたファイルは圧縮ファイルになっているため、まずはフォルダを解凍（展開）します。

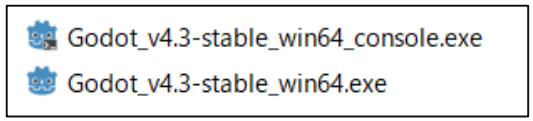




この場合だと、Godot_v4.3-stable_win64.exeがGodotの実行ファイル（アプリケーション）です。
PC内のどこか自分が分かりやすい場所へ移動しておくようにします。例えば下図のような構成にすると良いでしょう。



ダブルクリックで実行するとGodotが立ち上がります。

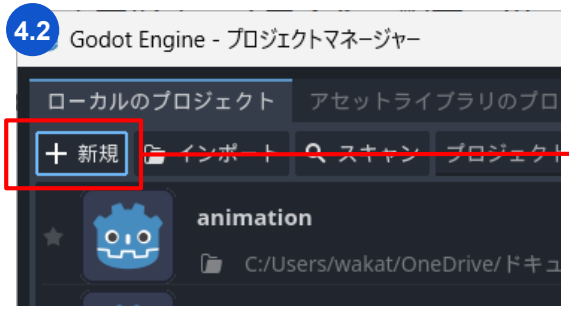


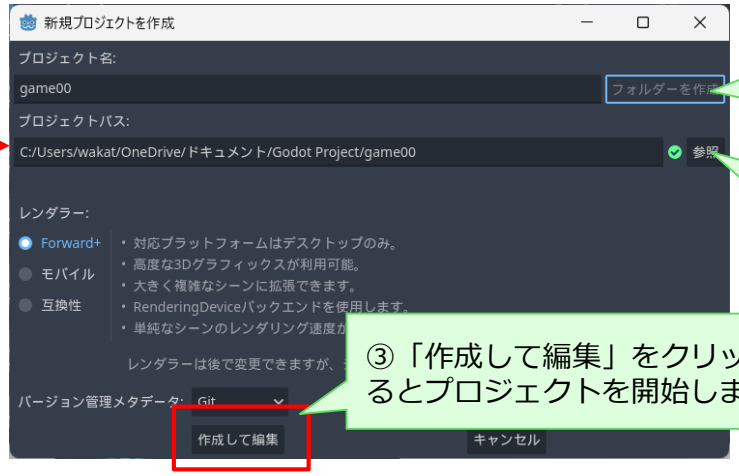
consoleと書かれたものは、黒いコンソールウインドウも一緒に開くバージョンで、ログ確認したいときなどに便利なものですが、通常はconsoleでは無い方を使います。

プロジェクトの開始

Godotアプリを起動させたら、プロジェクトマネージャー画面から新規プロジェクトを開始します。
プロジェクト１つでゲーム１つとなります。ゲームをつくるごとにプロジェクトを作成します。

4.2



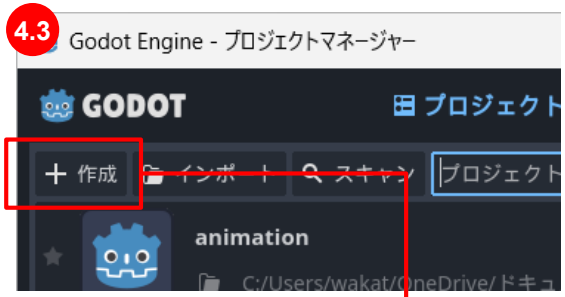


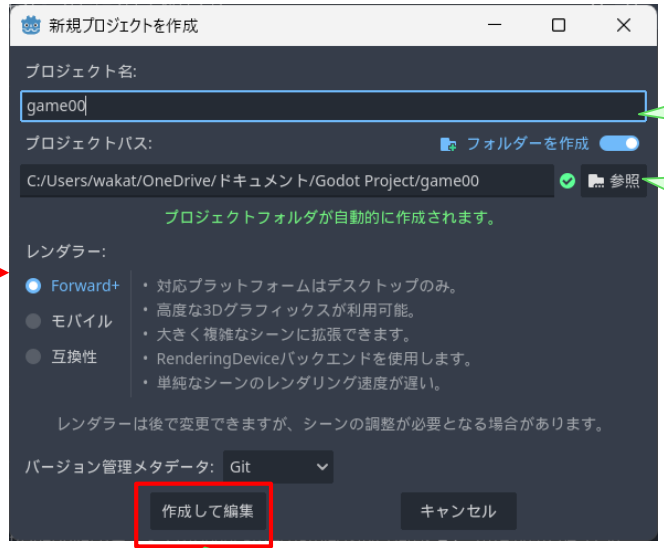
②プロジェクト名を決めて「フォルダーの作成」をクリックします。

①プロジェクトを保存するパス（PC内のフォルダ）を選びます。

③「作成して編集」をクリックするとプロジェクトを開始します。

4.3





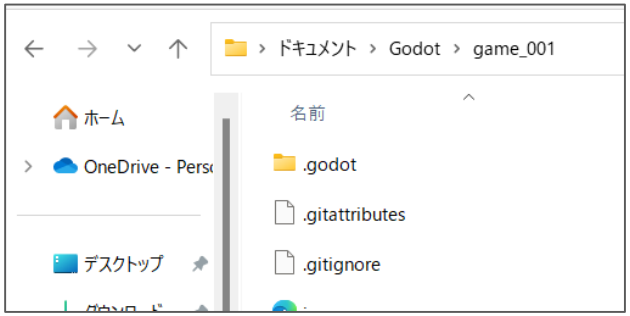
①プロジェクト名を決める。

②プロジェクトを保存するパス（PC内のフォルダ）を選びます。

③「作成して編集」をクリックするとプロジェクトを開始します。

バージョン4.3からは、デフォルトで自動的にプロジェクト名のフォルダを作成するようになったため、分かりやすくなりました。

プロジェクトを作成すると、フォルダが作成されてプロジェクトファイルが生成されます。



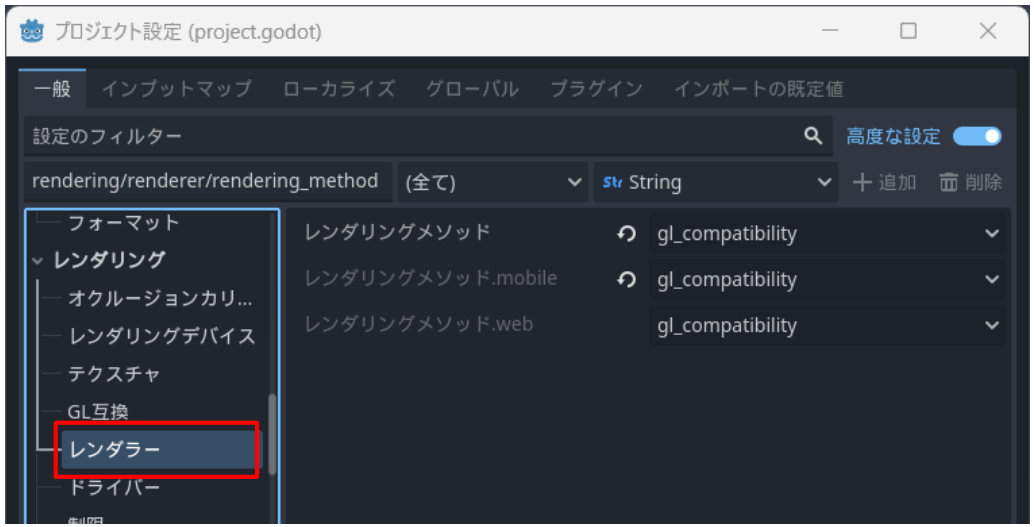
プロジェクトのレンダラー設定

プロジェクトを開始するときにレンダラーを選択できます。

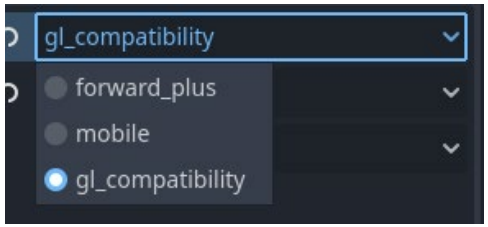
特に支障がなければデフォルトの [Forward+] のままでも良いですが、2Dゲームを制作する程度であれば、3つ目の選択肢にある [互換性] でも良いです。

特に使用しているPCが低スペックであるなら、互換性を選択しておくと良いでしょう。

レンダラーの設定はプロジェクト設定メニューからでも変更可能です。



プロジェクト設定では選択肢は日本語になっていません。
（バージョン4.3の場合）



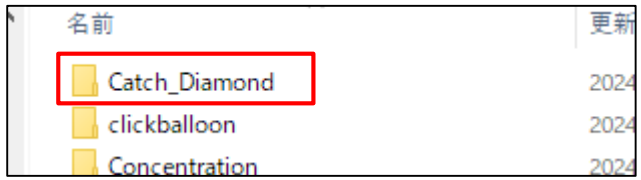
また、制作時にGodotが固まってしまうなど不具合が出る場合は、レンダラー設定を [互換性（gl_compatibility）] に変更すると解消する場合もあります。

おおまかには、下記を参考にしてください。

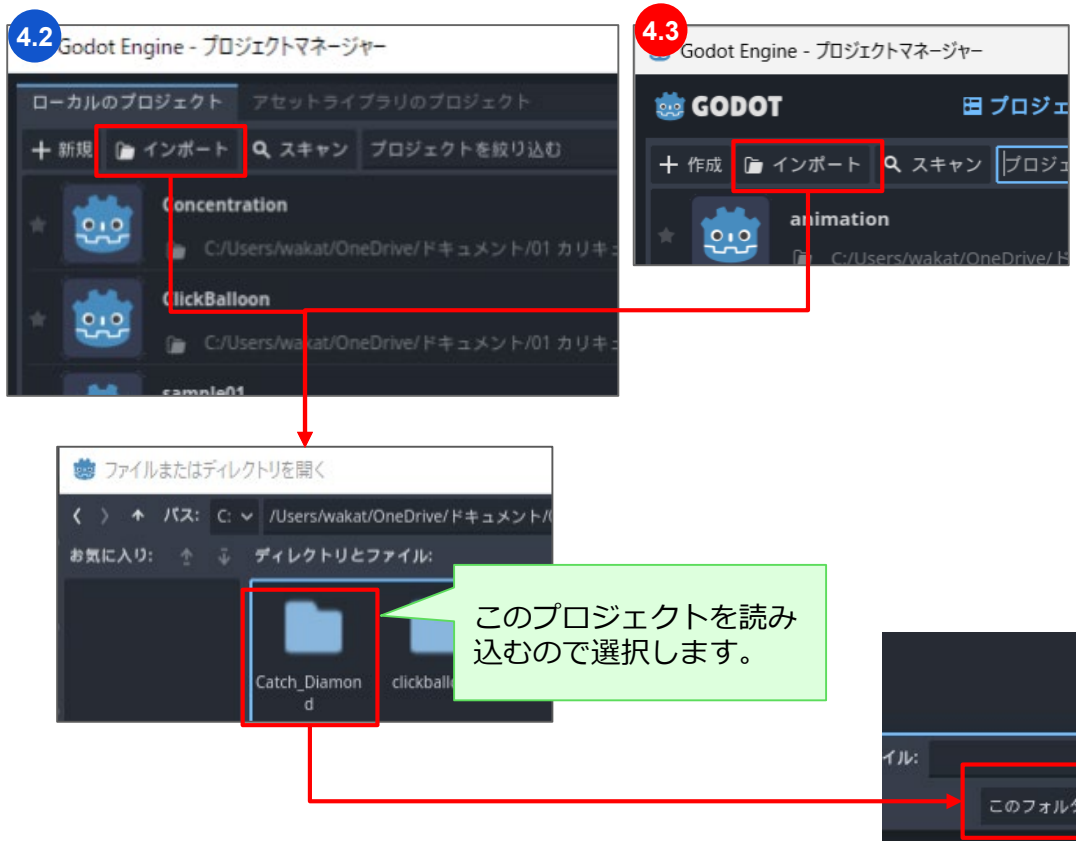
・Forward+: 高品質な3Dゲーム向け（高スペックPC / 最新コンソール）
・Forward Mobile: モバイル向け（低消費電力 / 高速）
・Compatibility: 旧型PCやWeb向け（軽量 / 互換性重視）

プロジェクトのインポート

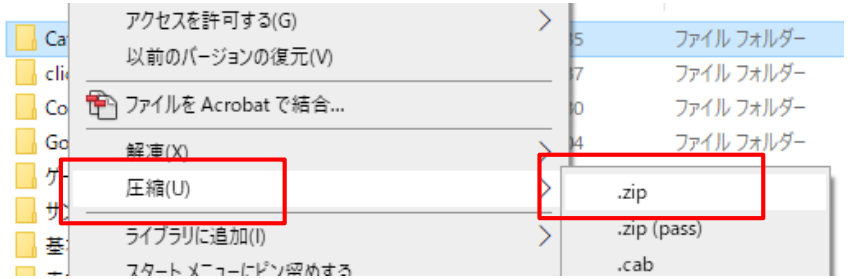
他の人が作ったプロジェクトを読み込むには、プロジェクトマネージャーからインポートを行います。
まずは、プロジェクトのフォルダごとPCの任意の場所へ移動しておきます。



Godotを立ち上げて、プロジェクトマネージャーの画面にし、インポートボタンをクリックします。

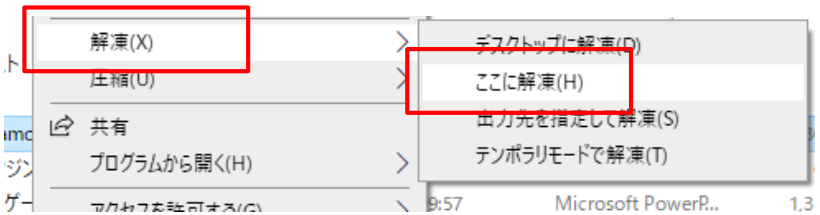


Godotのプロジェクトはフォルダ単位で管理できるため扱いやすいのですが、フォルダ内に含まれるファイル数が多いため、他の人と共有する場合は、プロジェクトフォルダをzipファイルなどに圧縮して受け渡しをするといいでしょう。



対象のフォルダ上で右クリックして、圧縮メニューを選び、保存する場所を選んで圧縮ファイルを作成します。

圧縮ファイルを受け取った側は、圧縮と同じような手順で、解凍をすればフォルダを復元できます。

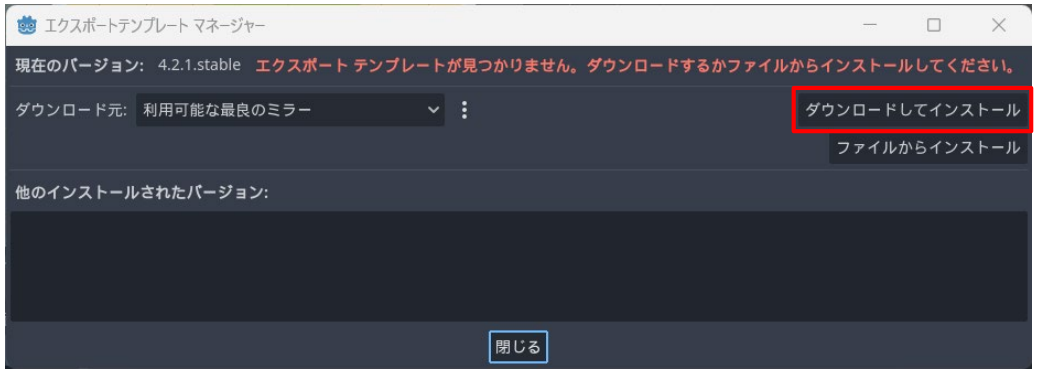


プロジェクトのエクスポート

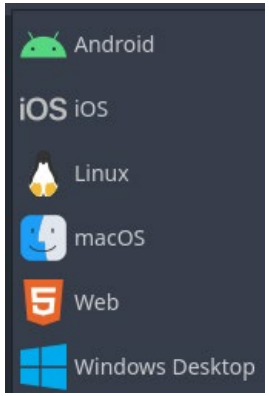
作成したプロジェクトは単独で動作する形式に書き出すことができます。PCはもちろんですが、スマートフォンやタブレット用の形式にすることもできます。

まずはエクスポート用のテンプレートを用意します。

[エディター] > [エクスポートテンプレートの管理...]



ダウンロードしてインストールを行います。ダウンロードが完了したらウィンドウを閉じます。その後、次ページ以降の説明のように、ターゲットプラットフォームごとのエクスポートメニューを選択します。



標準ではこのようなプラットフォームへのエクスポートに対応しています。

AndroidやiOSなど、スマートフォン、タブレット向けに書き出してデバイスに転送するには、やや複雑な手順が必要になるため、ここでは解説していません。

Godotでつくったゲームを、Nintendo Switchなどの家庭用ゲーム機で遊ぶことはできるのでしょうか？

1つの方法として、W4 Gamesが提供する「W4コンソール」というサービスがあります。
https://www.w4games.com/w4consoles

もちろん有料のサービスにはなりますが、家庭用ゲーム機向けに移植することができるようになります。

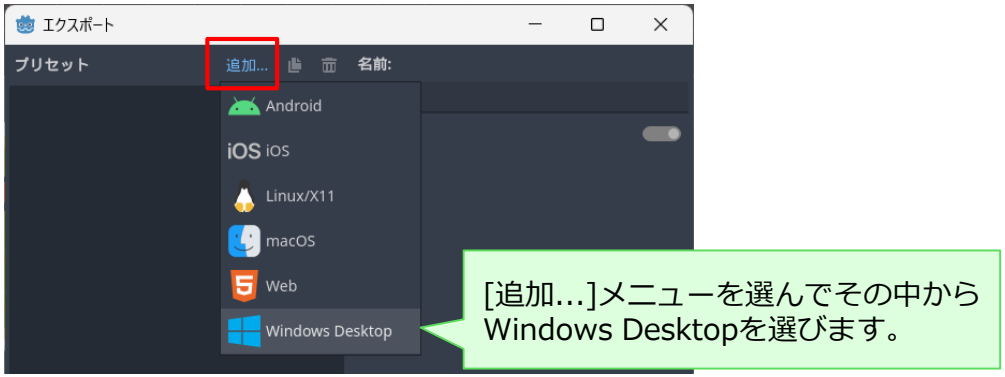
※「Nintendo Switch」は任天堂の商標です。

プロジェクトのエクスポート（１）Windows

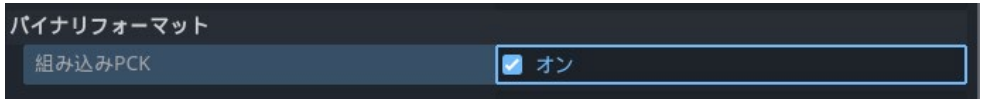
■Windows

Windows PCで動作する実行ファイル（exe）形式で出力します。

[プロジェクト] > [エクスポート...]

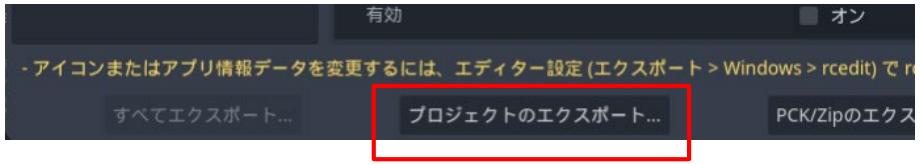


設定画面が表示されます。ここでは必要最低限の設定のみ行います。

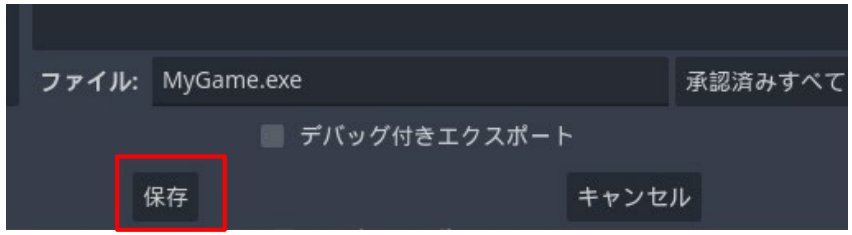


PCKファイル（Packed Resource File）は、プロジェクトのリソース（画像、サウンド、スクリプトなど）をパッケージングするためのファイルです。プロジェクト内のすべてのリソースを1つのPCKファイルにまとめ、それをエクスポートされた実行ファイル（.exe）と一緒に使用します。

その他の設定は必須ではないので、プロジェクトのエクスポートを行います。



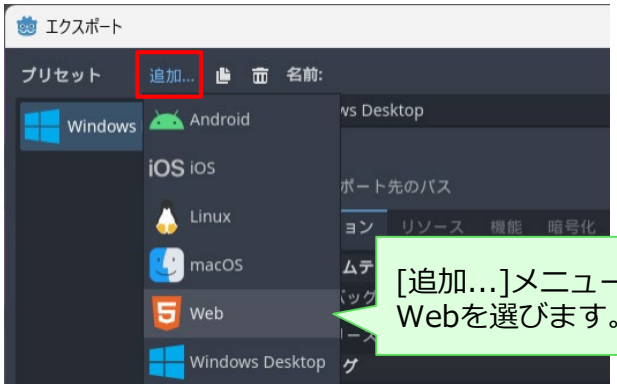
保存したい場所を選んだらファイル名を決めて保存します。



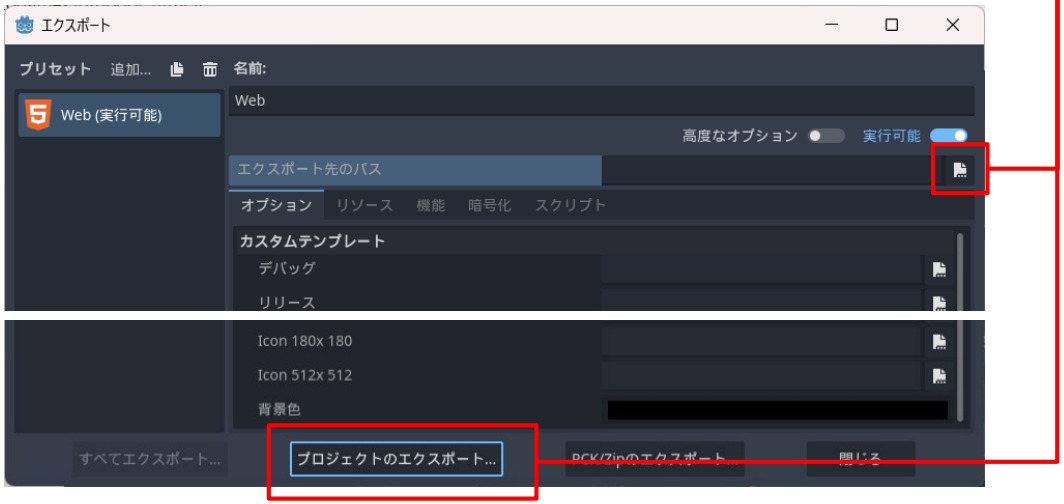
プロジェクトのエクスポート（２）Web

■Web

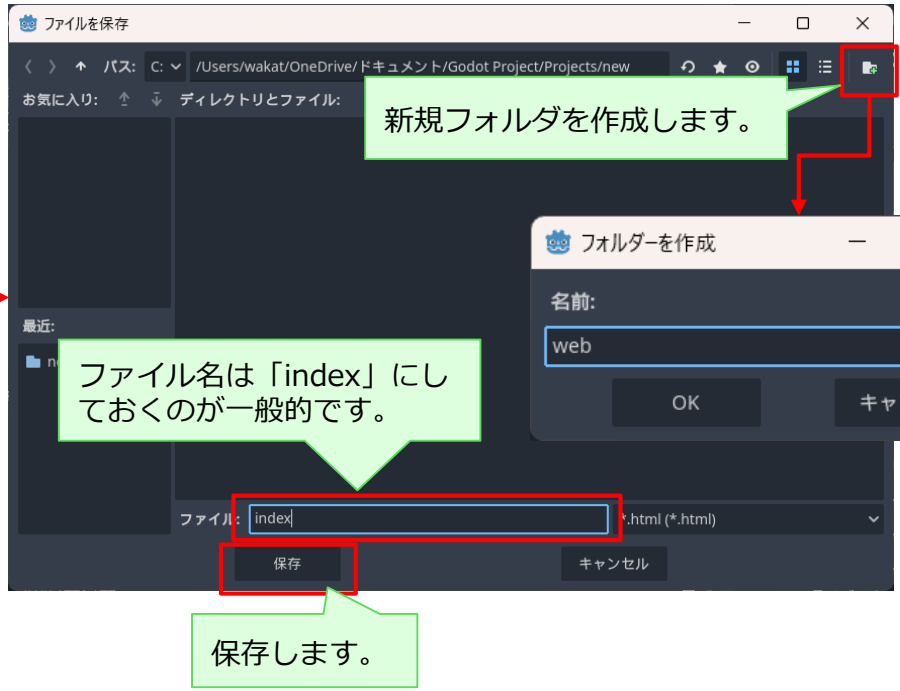
[プロジェクト] > [エクスポート...]



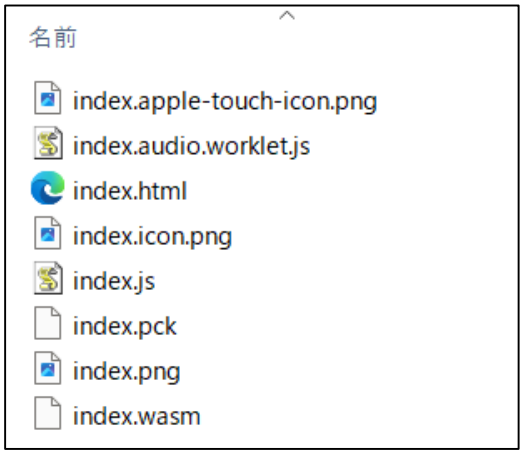
設定画面が表示されます。基本的にはデフォルトの設定のままでも構いません。



Web用にエクスポートする場合は、複数のファイルが生成されるためフォルダをつくっておきます。フォルダはあらかじめ作成しておいても良いです。



ファイル一式が指定したフォルダに生成されます。

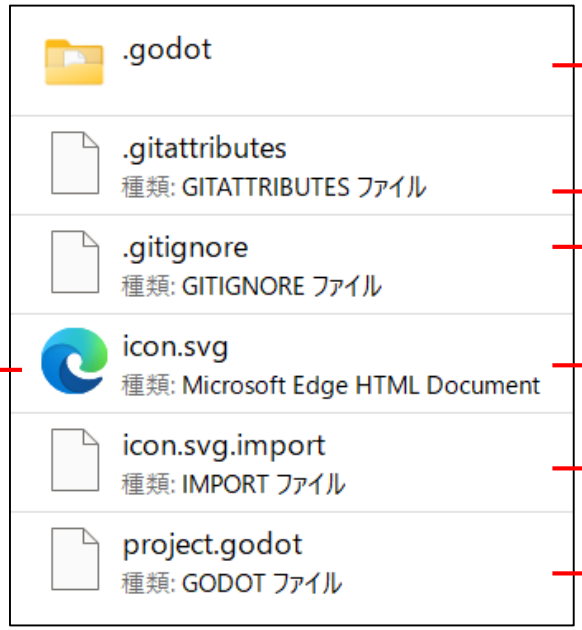


webブラウザで開いてもこのままでは動作せず、webサーバーにアップロードしてアクセスする必要があります。

ファイル管理

Godotでゲーム制作をしていくと、様々なファイルを管理していく必要が出てきます。
まずは、プロジェクト作成時に生成されるものをチェックしてみましょう。

icon.svgのみが、ファイルシステムに表示されます。



特にゲーム内では読み込まれないファイルが入っている

プロジェクト作成時に、バージョン管理としてGit（ギット）を選択した場合に生成される、Gitバージョン管理システムで使用される設定ファイル。

Godotのアイコン画像

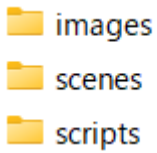
画像、音、フォントなどの読み込み素材ごとに生成される、ファイル読み込みに関する情報が含まれたファイル

プロジェクトの設定ファイル

ゲーム制作を進めていくと、主にこのようなファイルが生成されていきます。

.tscn	シーンファイル
.gd	GDScriptファイル
.import	素材ファイルの読み込み設定ファイル

ゲームの規模が小さくファイルの数が少ない間は、特に整理をしなくても良いですが、数が多くなるとフォルダ分けして管理することが必要です。



例えば、このようにフォルダをつくって
・画像素材：imagesフォルダ
・シーン：scenesフォルダ
・スクリプト：scriptsフォルダ
のように、仕分けて保存しておくとアクセスしやすくなります。

フォルダ名を日本語（例えば「画像」など）にしても動作はしますが、文字化けやエラーのリスクがあるため、英語表記に慣れるようにしておきましょう。

1-2 画面と基本操作

- 画面構成（１）
- 画面構成（２）
- 画面サイズ
- ツールメニューとワークスペース
- プロジェクトの実行
- エラー処理
- ヘルプ
- 右クリックメニュー、ショートカット

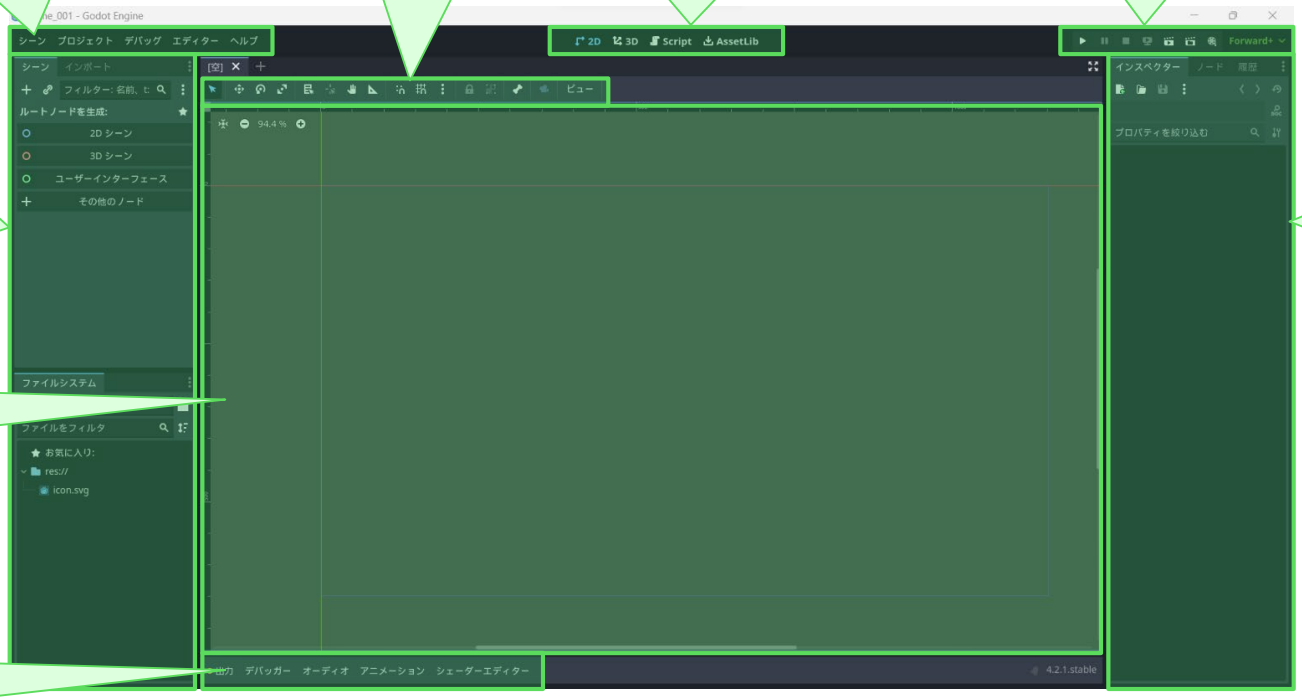
画面構成（１）

◆メインメニュー
プロジェクト全体の操作

◆ツールバー
画面内の操作
マウスモードの切り替え

◆ワークスペース
ビューポートの切り替え

◆テストプレイボタン
プロジェクトの実行



◆ドック

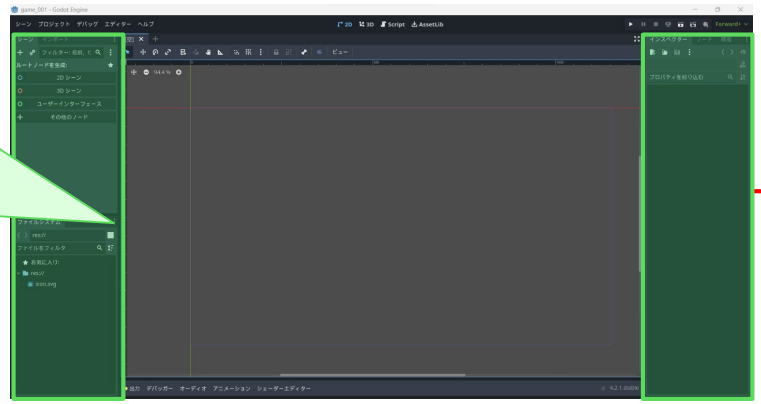
◆ドック

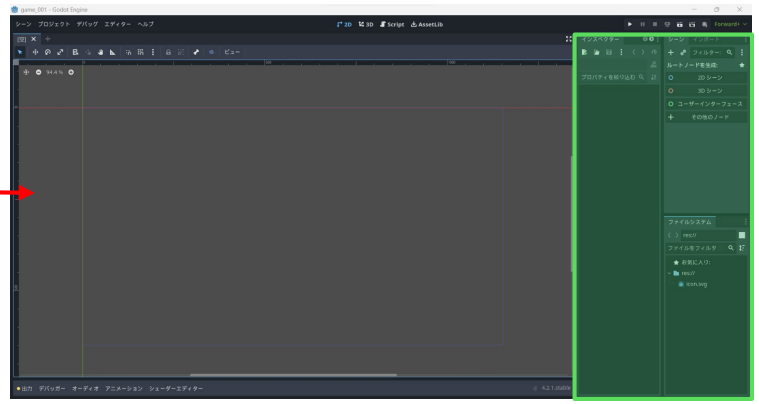
◆ビューポート
各オブジェクトの
レイアウト

◆ボトムパネル
機能に合わせて
パネルを変更

ドックのレイアウトは左右好きな場所に変更して、
保存しておけます。

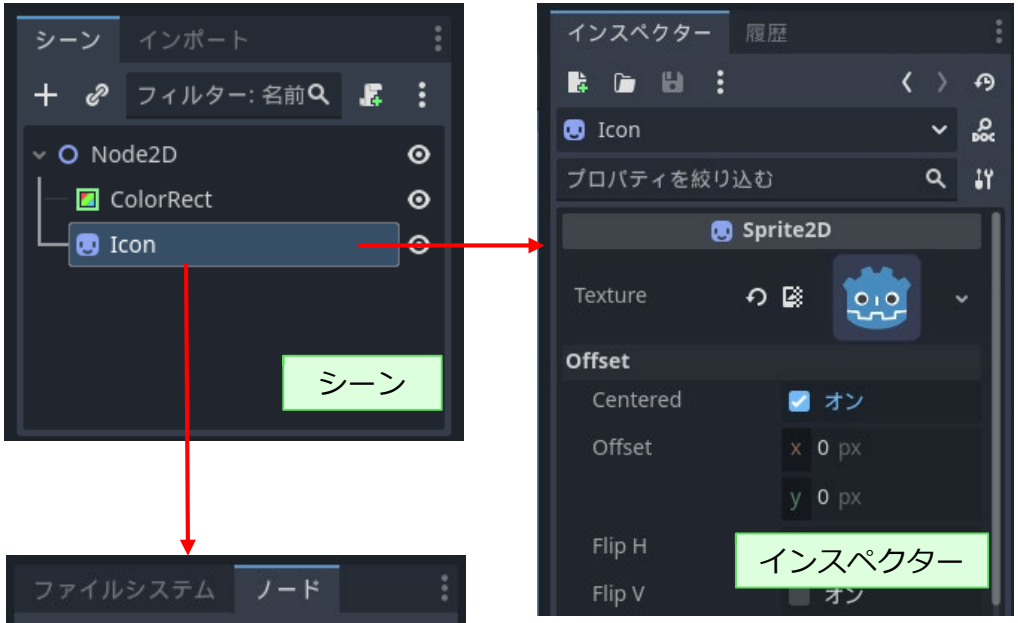
[エディター] > [エディターレイアウト]
> [レイアウトを保存]

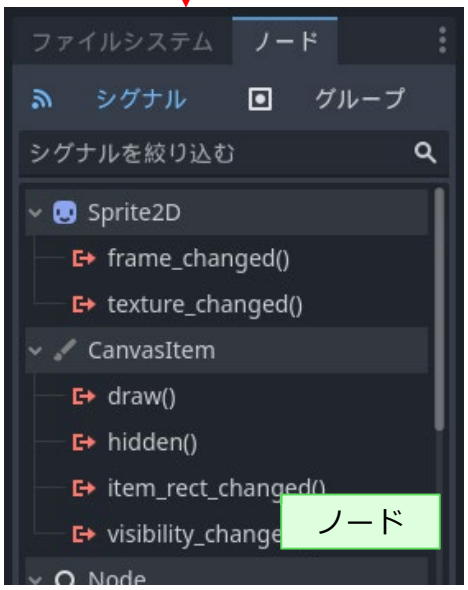




画面構成（2）

【シーン】には、そのシーンに含まれるノードやシーンが表示されます。【インスペクター】では、選択したノードのパラメータを設定することができます。





【ノード】は選択されているノードの「シグナル」や「グループ」を管理します。

よく使うのは【シーン】【インスペクター】【ノード】ドックです。

【ファイルシステム】は、プロジェクトで使用するアセット（素材）を一覧で管理します。選択したアセットは【インポート】で設定したり、再インポートを行えます。

【履歴】には操作した履歴が管理されています。ここから操作を巻き戻すこともできます。

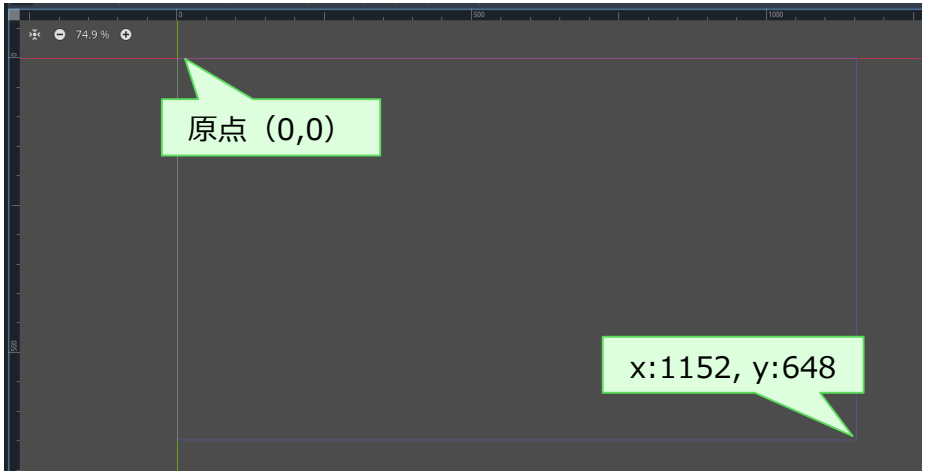
画面サイズ

例えばScratchだと画面サイズは、横480×縦360ピクセルと決まっています。自由度がないようですが、固定されているからこそ制約のある範囲で工夫できることもあります。
Godotの場合はゲーム画面サイズを自由に設定可能です。

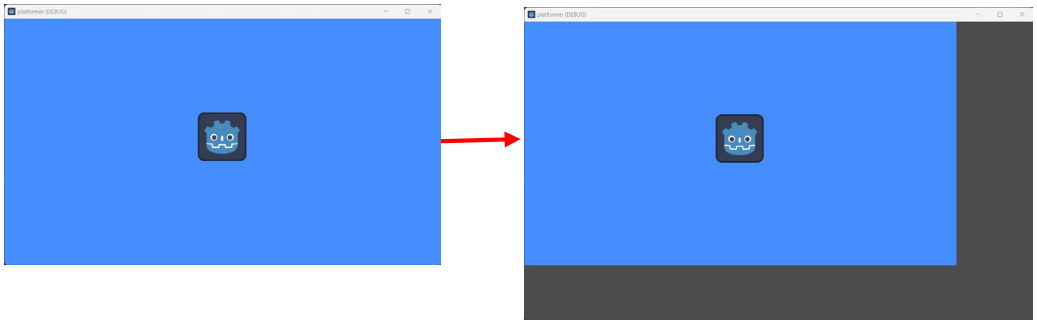
[プロジェクト] > [プロジェクト設定...]

デフォルトでは、1152x648ピクセルです。

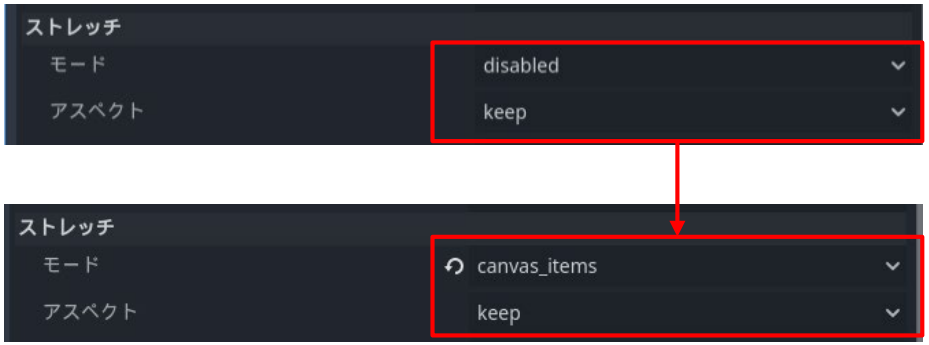
青い線の四角が設定された画面サイズの範囲です。

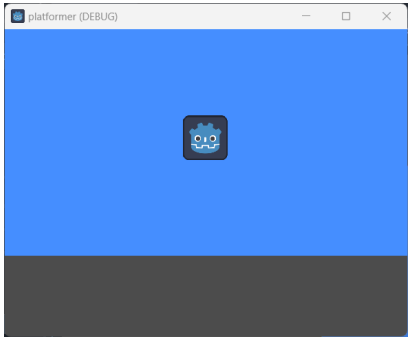


ゲームが実行されているウインドウサイズを変更した場合、デフォルト設定ではゲーム内容自体は拡大縮小されません。

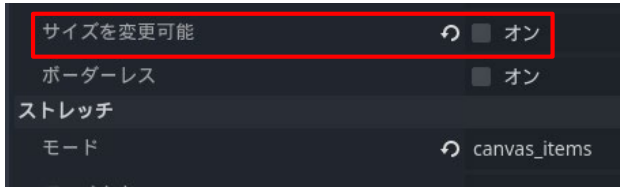


プロジェクト設定のストレッチの設定を変更すれば、ゲーム内容もウインドウサイズに追随するようになります。





サイズ変更をオフにしておくとウインドウサイズは変えられなくなります。



レイアウトワークスペースで使えるツールメニューです。表中の「オブジェクト」とは、ノードやシーンのことです。画面上にオブジェクトが増えてくると目的のものを操作するのが難しくなるため、ツールを使い分けると操作しやすくなります。

	選択	画面上のオブジェクトを選択する
	移動	選択したオブジェクトを移動操作に限定する
	回転	選択したオブジェクトを回転する
	スケール	選択したオブジェクトを拡大縮小する Shiftキーを押しながらだと縦横比率を保つ
	リスト	クリックした位置の選択可能なノードリストを表示し、選択することができる
	ピボット変更	回転の中心位置を変更する
	パンモード	スペースキーを押しているときと同じく、画面を上下左右に水平移動させる
	定規	画面上の長さや角度を計測する
	スナップ	オブジェクトの移動時に吸着させる
	グリッド	オブジェクトの移動時にグリッドに吸着させる
	スナップモード	スナップモードを変更する
	ロック	選択したオブジェクトを動かせなくするためにロックする
	グループ化	オブジェクトをグループ化・解除する

■ワークスペースの切り替え

ゲーム画面の制作をするときは。ワークスペースを2D（3D）に切り替えて操作をします。

スクリプトを書く場合は、Scriptに切り替えます。

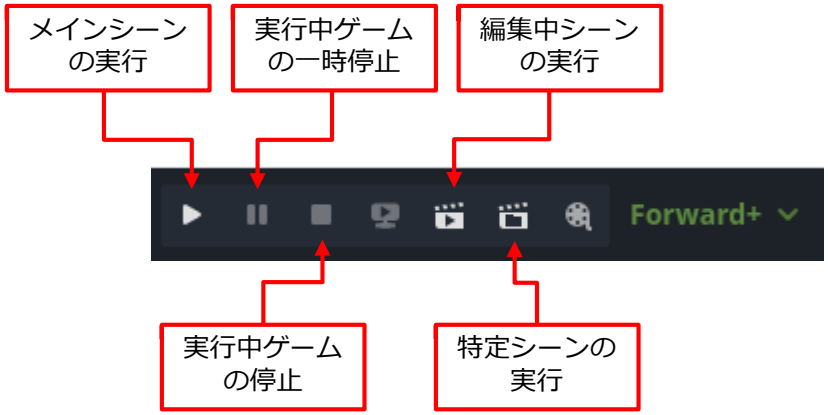
要注意

ここで注意しなければならないのは、スクリプトを選ぶ場合は、シーンドックでノードを選ぶのではなく、スクリプトのリストから、編集したいスクリプトを選ぶと言うことです。

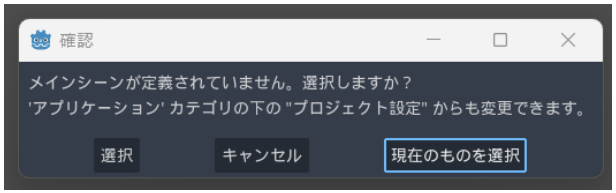
慣れないうちはとても間違えやすいため、特にスクリプトを編集する際には、よく確認をしましょう。

プロジェクトの実行

ゲーム制作中には何度もテストプレイをしながらプログラミングをして完成度を高めていきます。プロジェクトの実行は画面右上のテストプレイボタンで行います。



最終的にプロジェクトとして実行されるメインシーンは、初めて実行する場合に選択することができます。

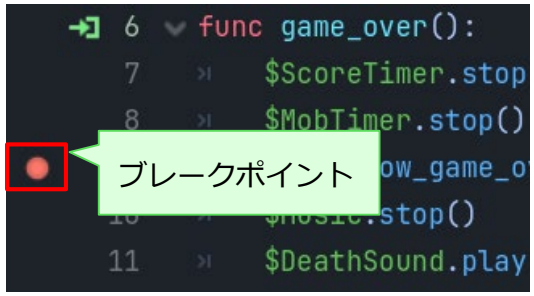


メインシーンは途中で変更することもできます。

[プロジェクト] > [プロジェクト設定...]

■ブレークポイント

コードエディタの行番号左端をクリックすると、赤丸印をつけられます。プロジェクトを実行するとこの部分で一時停止させることができ、ゲームの状態を確認できます。



ブレークポイントで停止すると、ボトムパネル右端にあるメニューが使えるようになります。ステップイン、ステップオーバーは、１行ずつコードを実行していきます。

ステップイン：１行ずつ実行する中で、関数の呼び出しがあると、関数の中も１行ずつ実行していきます。

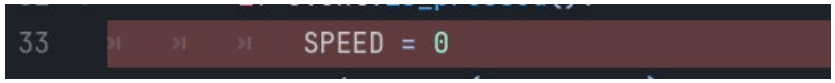
ステップオーバー：関数の呼び出しがあった場合、関数の中には入らずに、関数自体を１行として次に進みます。

ゲームが複雑になってきたら、ブレークポイントを積極的に使ってみよう。

エラー処理

プログラミングにおいては、エラーは必ず発生するものです。
Godotでは、主にコード作成時に発生するエラーと、ゲーム実行時に発生するエラーがあります。
エラーが発生すると、コード下部やボトムパネルのコンソールにエラーメッセージが表示されます。

例えば、このようにコード上でエラーが発生している場合は、該当する場所が赤色ぽく表示されます。

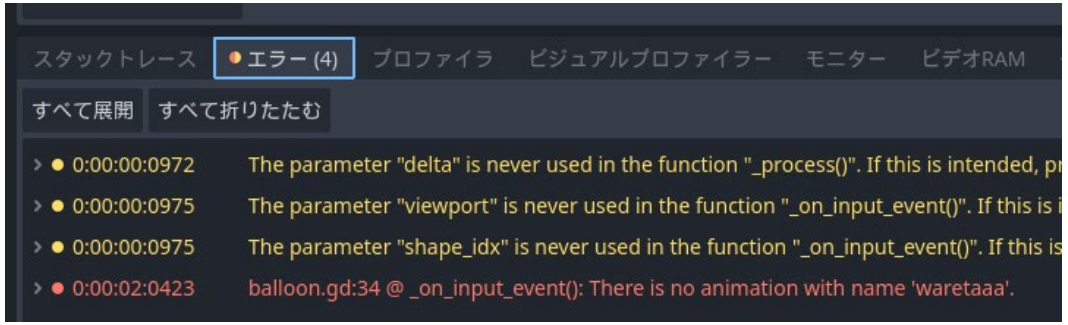


この例では、定数をコード中で変更しようとした為、エラーが発生しています。コード下部にもエラーメッセージが表示されています（英語で表示されます）。

ここをクリックすれば、下に選択可能なテキストも表示されるので、コピペして翻訳するなども可能です。

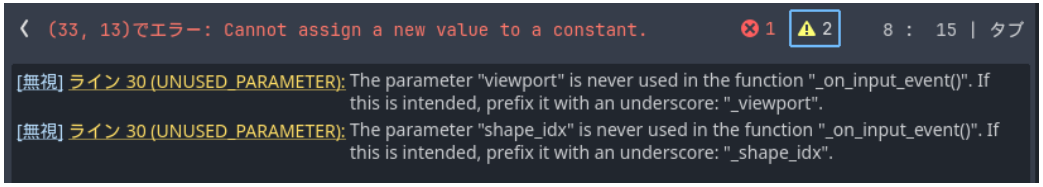
エディター設定でパネルの表示設定ができます。

実行中に発生したエラーは、コンソールのエラータブ内にメッセージが表示されます。



この例では、34行目で存在しない「waretaaa」という名前のアニメーションを指定しているために発生したエラーです。

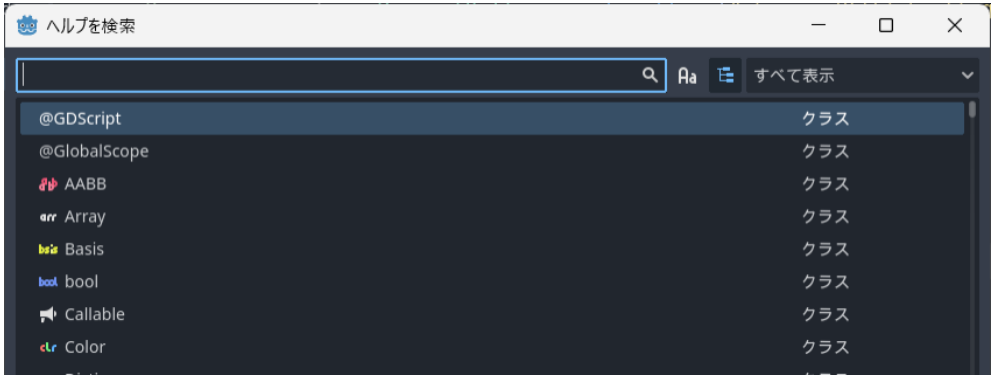
ちなみに黄色文字の方は「警告」で、エラーではないですが、改善しておいたほうが良いような事柄になっています。



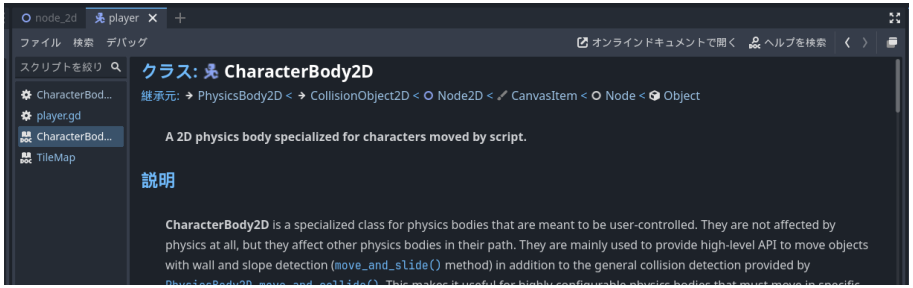
ここでは、引数に使用されているviewportやshape_idxは、関数の中で使われていないため、_を付けて、_viewportや_shape_idxにするよう促されています。

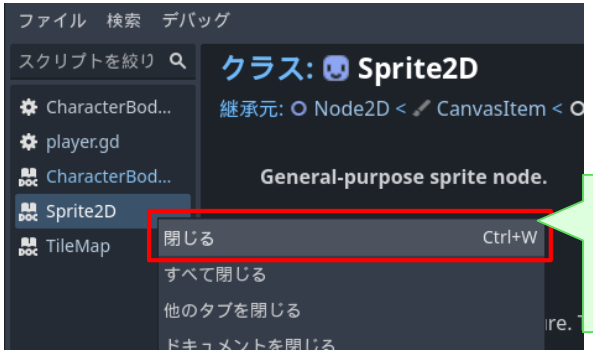
ネット上にも公式のヘルプサイトはありますが、Godotのエディタ内でもヘルプを参照することができます。

[ヘルプ] > [ヘルプを検索...]



この画面から調べたいクラスを選ぶと、ヘルプを見ることができます。

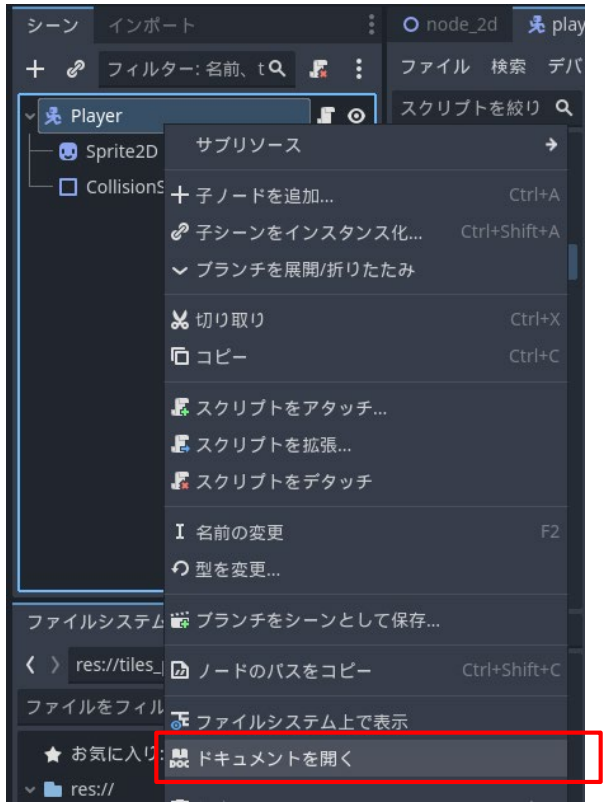




閉じる場合はリストから右クリックして閉じるメニューを選びます。

また、Script画面の時に出る、ヘルプを検索をクリックしても同様の動作をします。

シーンドック内のノードの上で右クリックし、[ドキュメントを開く]メニューを選ぶと、直接そのノードのヘルプが開きます。



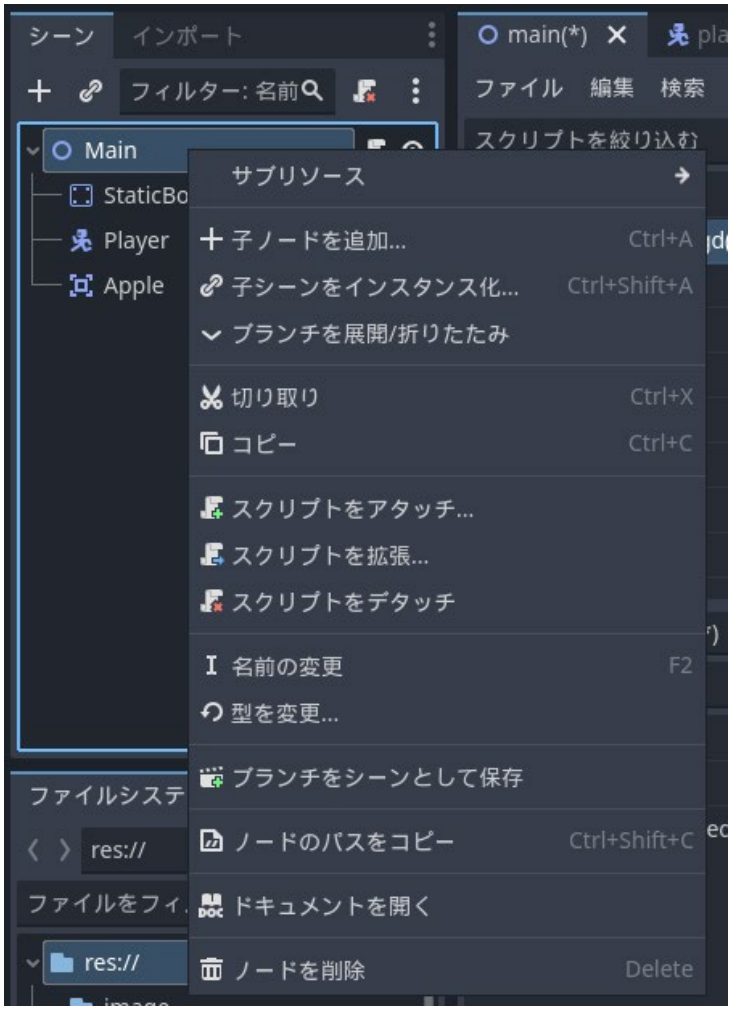
右クリックメニュー、ショートカット

アプリケーションに慣れてきたら、ショートカット操作を少しずつでも使うようにしていくと、より効率よく開発を進めることができます。

ショートカットは他にもありますが、まずはこのあたりから慣れていこう。

ctrl + a	ノードの新規追加
ctrl + n	シーンの新規追加
ctrl + k	コードの選択範囲をコメントアウト、解除
ctrl + s	保存
ctrl + z	１つ元の操作に戻る
ctrl + shift + z (ctrl + y)	１つ次の操作にやり直し
F5	プロジェクト実行
F6	現在のシーンを実行
Alt＋マウスホイール	一定間隔でのビューポートの拡大縮小
Ctrl＋マウスホイール	コードの拡大縮小
Space＋マウス	ビューポートの移動

この例は、ノード上でマウスを右クリックした際に表示されるメニューです。このように右クリックで簡単にアクセスできるメニューもありますので、いろんな場面で試しながら覚えていくとよいでしょう。



1-3 シーンとノード

- シーンとノード（1）
- シーンとノード（2）
- ノード概要
- 2Dノード
- Controlノード
- 2Dゲームでよく使うノード
- 衝突検出ノード
- RigidBody2D
- CharacterBody2D（1）
- CharacterBody2D（2）
- StaticBody2D、AnimatableBody2D、Area2D
- コリジョン設定

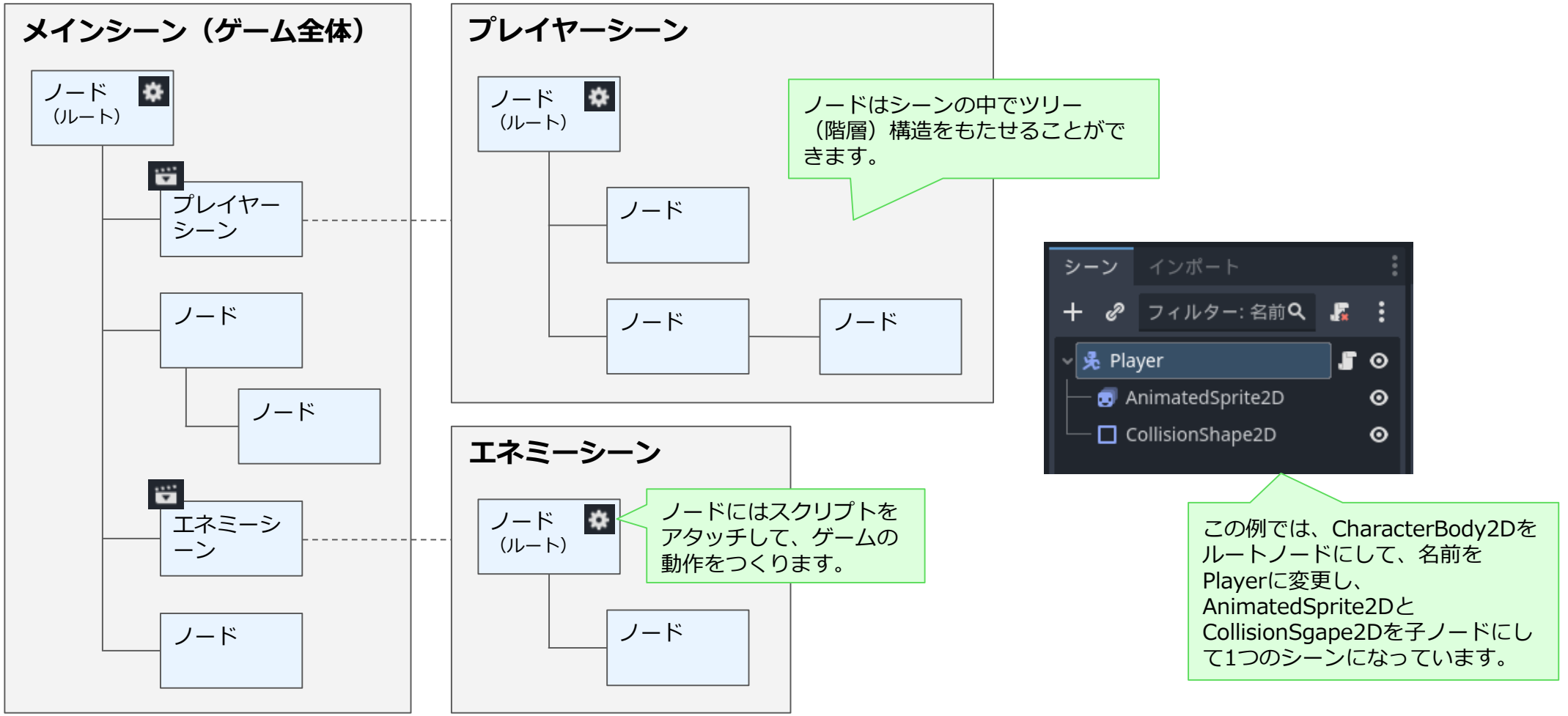
シーンとノード（1）

Godotで最も基本となる要素は「シーン」と「ノード」です。例えば、プレイヤー、敵キャラ、マップ、メニュー画面などをそれぞれシーンとして作成します。
シーンはシーンごとに個別のファイルで保存します。

「ノード」はシーンの中の実際の部品にあたります。様々な機能をもったノードを組み合わせて一つのシーンを作成していきます。

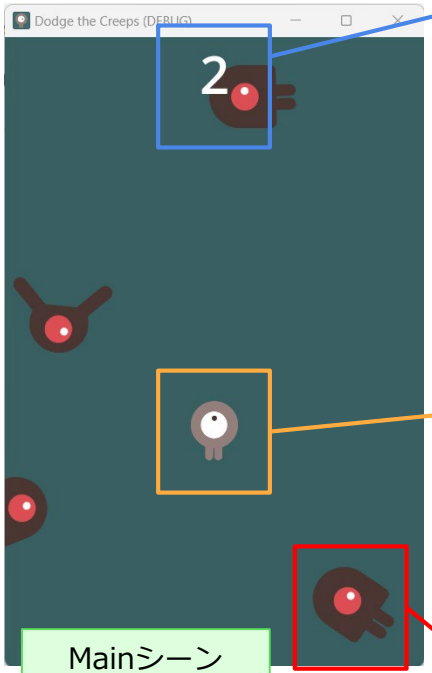
ノードには「スクリプト」をアタッチ（関連付け）して、ゲーム中の動作をつくっていきます。スクリプトの基本については基本テキスト第3章で解説しています。

ゲームをつくっているうちに慣れていきますので、あまり難しく考える必要はありません。

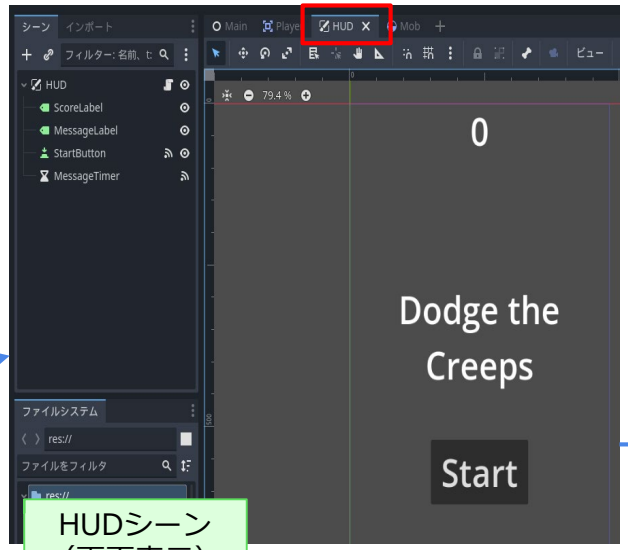


シーンとノード（2）

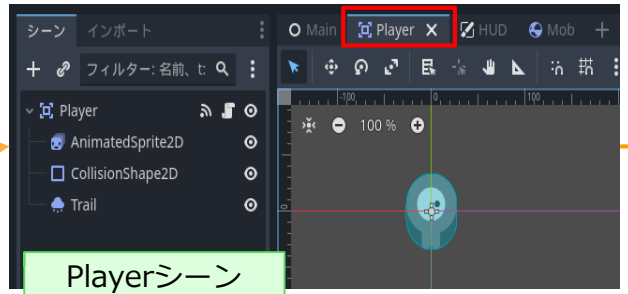
公式のチュートリアルゲームを例にして、シーンとノードの関係を見ましょう。



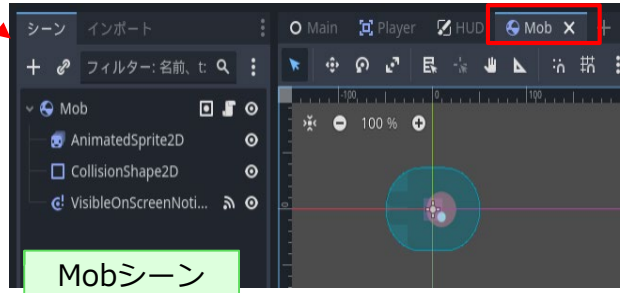
Mainシーン
（ゲーム全体）



HUDシーン
（画面表示）

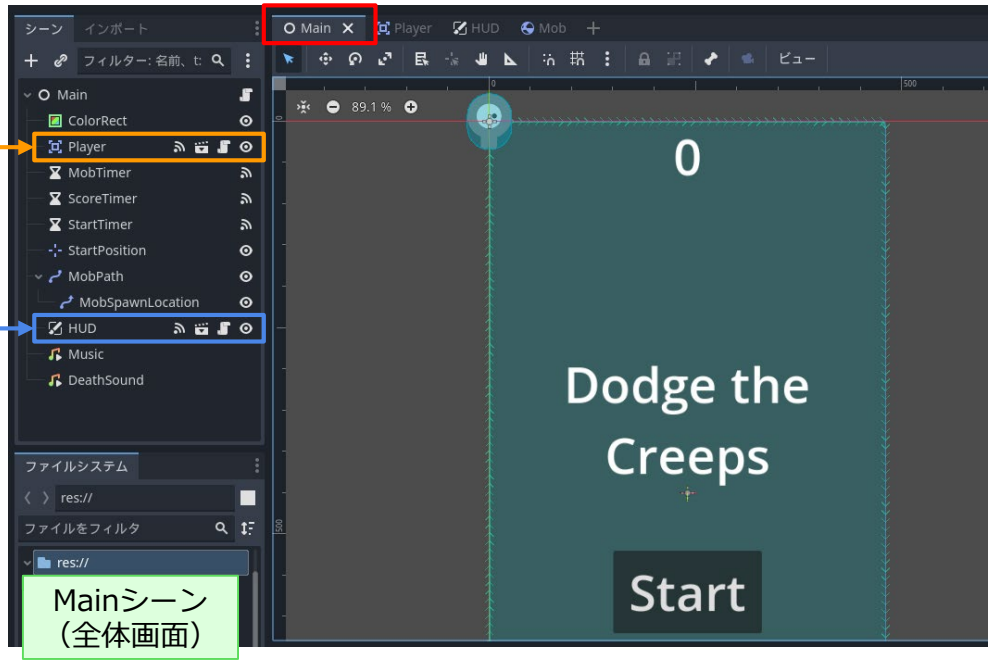


Playerシーン
（操作キャラ）



Mobシーン
（敵キャラ）

このゲームは4つのシーンで構成されており、各シーンがそれぞれいくつかのノードで作られているのが分かります。



Mainシーン
（全体画面）

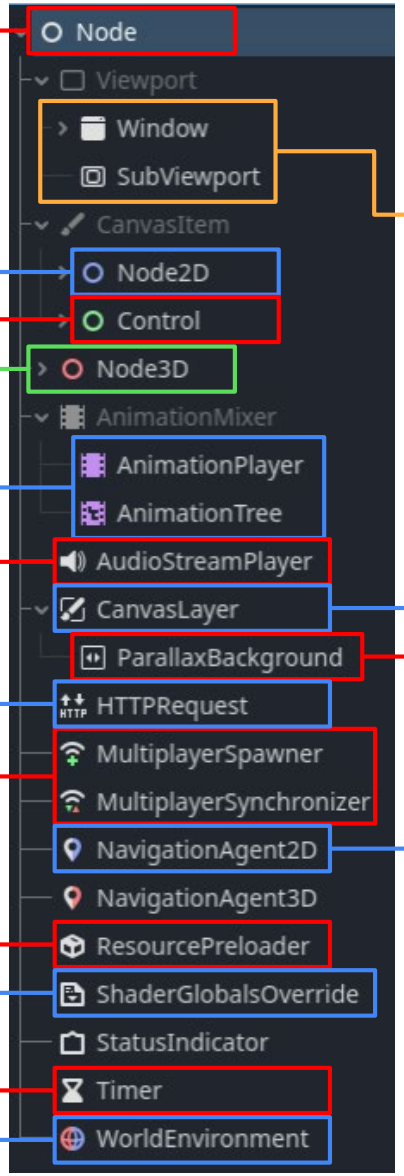
PlayerシーンとHUDシーンはMainシーン内に配置（インスタンス化）されています。敵キャラはゲームプレイ中にMainシーンのスクリプトの命令でランダムに生成（インスタンス化）されます。
そのためMainシーンのツリー内にはありません。
※インスタンス化については44ページを参照

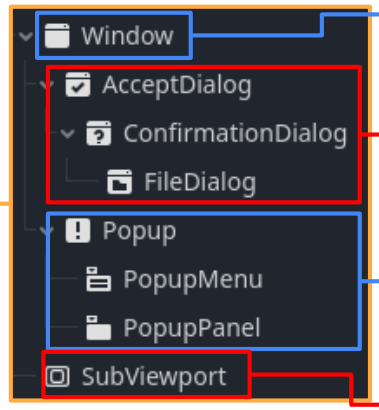
HUDとは（Head-Up Display：ヘッドアップディスプレイ）の略でゲーム内の情報を画面上に表示させるもののことを表しています。

Mobとは、元々は集団や群れを表す言葉ですが、ゲーム内では敵キャラや中立キャラなどの意味で使用されます。

ノード概要

ノードはGodotでゲームをつくるための部品となるものです。
つくりたいものに合わせて適切なノードを選べるようになるのが理想ですが、種類がとても多いため、まずはよく使うものに慣れよう。





すべてのノードのベースとなるノード。

2Dゲームの要素をつくるためのノードカテゴリー。最も良く使うノードが含まれる。

ノードをレイアウトしたり、文字情報などを表示するノードカテゴリー。

3Dゲーム制作に使うノード。

ノードのアニメーションに使う。

音の再生に使う。

Webサーバーと通信する。

オンラインマルチプレイヤーゲームに使う。

リソースを事前に読み込む。

シェーダー変数を上書きする。シェーダーとは画面上の描画をコントロールするしくみ。

タイマー処理に使う。ゲーム中の時間処理にとてもよく使う。

画面全体に視覚効果を加えることができる。

サブウインドウを表示する。

ダイアログウインドウを表示する。
メッセージや確認画面などに使う。

ポップアップウインドウを表示する。
ツールチップなど操作を妨げない補助的な表示に使う。

サブビューポートを表示する。例えばミニマップなどメイン画面と異なるものを描画する。

画面の描画順をまとめてコントロールする。

背景の表示を制御する。（バージョン4.3ではParallax2Dノードを使います）

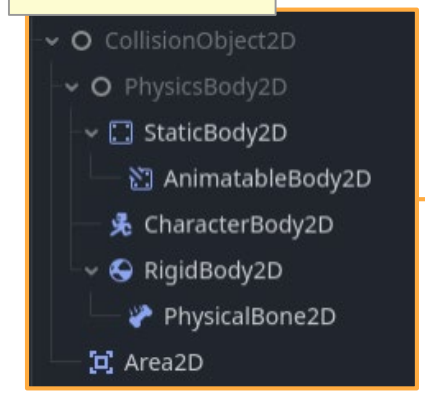
経路探索や移動に使う。例えば敵キャラクターがプレイヤーを追いかけるような場合に利用する。

Godotバージョン4.3のノード一覧です。
Godotのバージョンによってノードの種類や、アイコンの見た目が異なる場合があります。

2Dノード

2Dノードは2Dゲーム内に登場する様々なオブジェクトをつくるノードです。

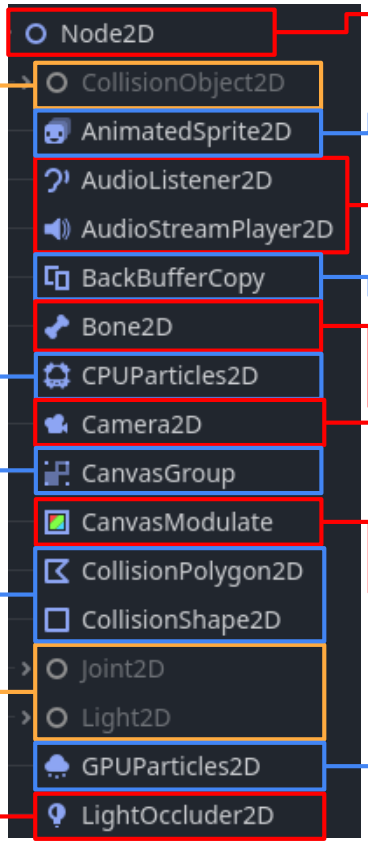
※詳細は26ページ



CPUでパーティクル処理を行う。

子ノードの描画処理をまとめて行う。

当たり判定に使うコリジョンオブジェクトを作成する。



2Dオブジェクトのベースとなるノード。

アニメーションするキャラクター。

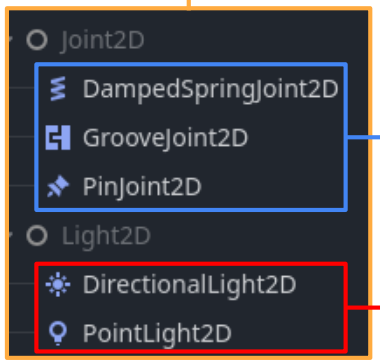
音の再生に使う。

画面の一部領域をバッファにコピーしシェーダーで扱う。

オブジェクトを骨格でつないでアニメーションさせる際に使用。

カメラの制御を行う。

シーン全体の描画処理をまとめて行う。



光源からの光をさえぎるオブジェクト。

2つのBodyオブジェクトを関連付けて特定の動作をさせる。

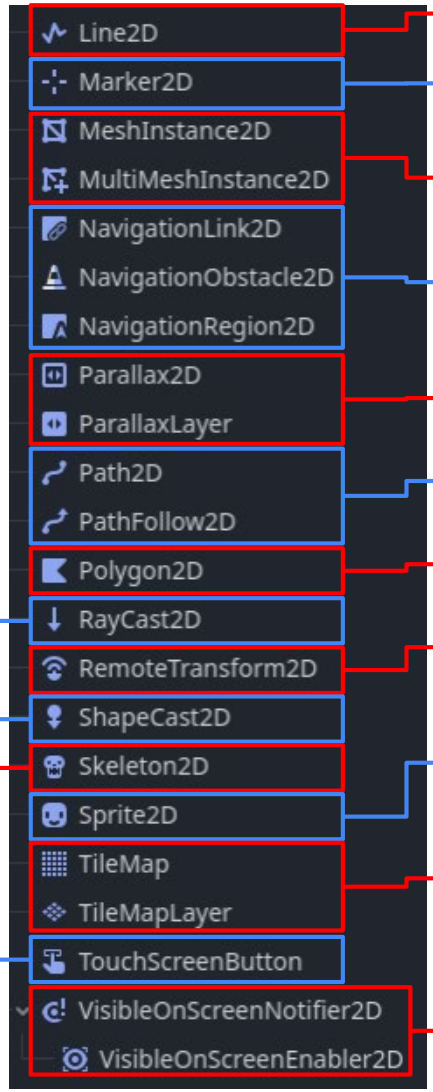
2Dシーン内に光源を設置してライティング効果を表現する。

特定方向に光線を出して、衝突検知に利用する。

四角や丸などの形状で、衝突検知に利用する。

GPUでパーティクル処理を行う。

モバイルデバイスなどのタッチスクリーン入力に対応する。



線を描画する。

シーン内で特定の位置を指定する。

メッシュオブジェクトを利用する。

経路探索やオブジェクトの移動経路に使う。

背景の表示を制御する。

オブジェクトの移動経路になる軌道（パス）を作成する。

多角形を描画する。

別のノードの位置、回転、サイズなどを利用する。

画像を指定でき、キャラクターやアイテムなど様々なところで使える。

タイル状の画像を使ってマップや地形などを作成する。

オブジェクトが画面内に表示されているかどうかを判定する。

Controlノード

Control（コントロール）ノードは、オブジェクトのレイアウトやインターフェースなどの画面表示、入力に関するノードが含まれています。

2Dゲームでよく使うノード

ノードの種類は多数ありますが、常に全てを使うわけではなく、特にゲーム内のキャラクターなどはよく使うノードが決まっています。

Node

Node：（ノード）シンプルに入れ物やフォルダのような使い方としてのシーンでよければ、このノードを使うことがあります。

Node2D

Node2D：（ノード2D）2Dゲームのための基本ノードで、位置、サイズ、回転など基本的なパラメータを設定できます。

Sprite2D

Sprite2D：（スプライト2D）画像を設定して表示させることができ、ゲーム内の様々なところで活用できます。

AnimatedSprite2D

AnimatedSprite2D：（アニメイテッド・スプライト2D）複数の画像を使ってアニメーションを作ることができ、歩行、ジャンプなど異なるアニメーションを設定して切り替えて使うことができます。

CollisionShape2D

CollisionShape2D：（コリジョン・シェイプ2D）物体に衝突領域を設定することで、他の物体との衝突検出ができるようになります。

RigidBody2D

RigidBody2D：（リジッド・ボディ2D）物理シミュレーションを実装しており、衝突、重力、摩擦などの物理効果を使えます。

CharacterBody2D

CharacterBody2D：（キャラクター・ボディ2D）壁などと衝突判定ができるので、ユーザーが操作するキャラクターに最適。

StaticBody2D

StaticBody2D：（スタティック・ボディ2D）壁や床など動かない物体、障害物などに使われる。

AnimatableBody2D

AnimatbleBody2D：（アニメイタブル・ボディ2D）移動する物体。物理シミュレーションの影響は受けないが、プログラムから動かすことができるもの。

Area2D

Area2D：（エリア2D）特定のエリアを定義することで、衝突判定のある物体が入った時、出た時（重なり）を検出できます。
そのため、ゲーム内でなにかキャラクターに対してアクションをおこすエリアなどに使えます。

ここに記載したノードの用途はあくまで一例ですので、必ずこのようにしなければならないわけではありません。

衝突検出ノード

特にアクションゲーム内の物体同士の衝突や、キャラクターがジャンプしたとき、落下するときなど、物理的な現象をシミュレートする機能がゲームエンジンには搭載されています。

RigidBody2D

RigidBodyを使うと、物体に物理シミュレーションを行うことができます。プログラムで動きをつけるのではなく、パラメータを設定することで物体が物理シミュレーションをした結果を反映します。

CharacterBody2D

CharacterBodyを使うと、キャラクターを地面にそって歩かせたり、壁との衝突などを判定できます。

StaticBody2D

StaticBodyは基本的には動かない物体に使うノードですが、他の物体との衝突検出を行うことができます。他の物体がぶつかっても動かされることはないですが、ぶつかった物体の反応を設定できます。

AnimatableBody2D

AnimatableBody2Dも他の物体との衝突検出ができますが、例えば動く床などアニメーションさせる場合などに効果的なノードです。

Area2D

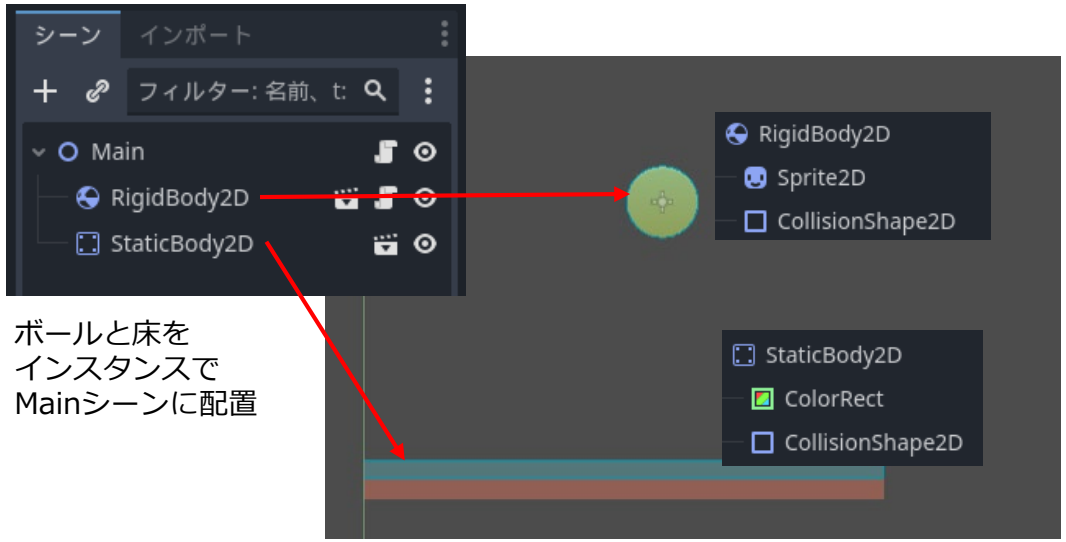
Areaも他の物体との衝突判定に使えますが、物理的な衝突ではなく、エリア内への出入りを検出します。これにより特定エリアへの出入りや、アイテムとの重なりなどゲーム内のイベント検出に使えます。

これらのノードは子ノードに、CollisionShape2Dノードで衝突範囲を作成することで、衝突する形状を決めます。

RigidBody2Dは、物理エンジンによって制御される動きをシミュレーションすることができます。

例えばボールが落下して床で跳ね返るような動きを、プログラミングだけで実現しようとすると、ボールの落下、衝突、跳ね返りをそれぞれつくる必要があります。

しかし、物理エンジンを使えばパラメータの設定だけでそれらを実現することができます。



ボールと床を
インスタンスで
Mainシーンに配置

ボールと床それぞれには、画像用のノードと、当たり判定のためのCollisionShape2Dノードがあります。

Rigidbodyとは「剛体」という意味です。力を加えても変形しない物体のことで、Unityにも同じ名前を持つコンポーネントがあり、同じく物理特性を制御するために使われます。

ボールである、RigidBody2Dには、このようなプロパティを設定できます。

Mass：質量

Friction：摩擦係数

Bounce：弾力性

これだけの設定で何もプログラムを書かなくても、ゲームがスタートするとボールは落下して、床で弾む動作を繰り返します。

さらに、開始時に初速を与える１行を追加すれば、横方向への力が加わり、ボールは床で跳ね返ったあと、右方向に落ちていきます。

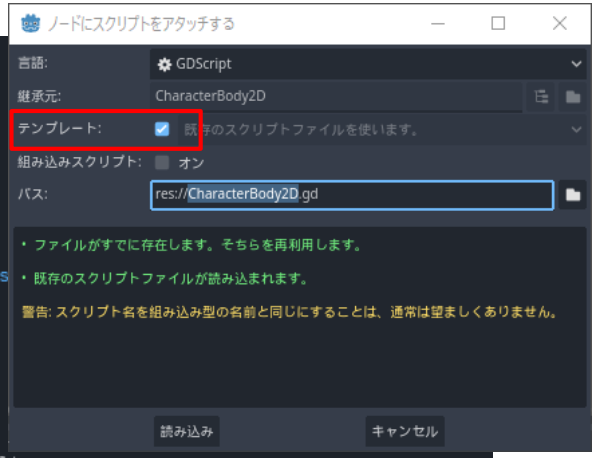
```
1   extends RigidBody2D
2
3   func _ready():
4       linear_velocity = Vector2(100, -100)
```

x=100, y=-100なので
右上方向への初速を与えます。

CharacterBody2D（1）

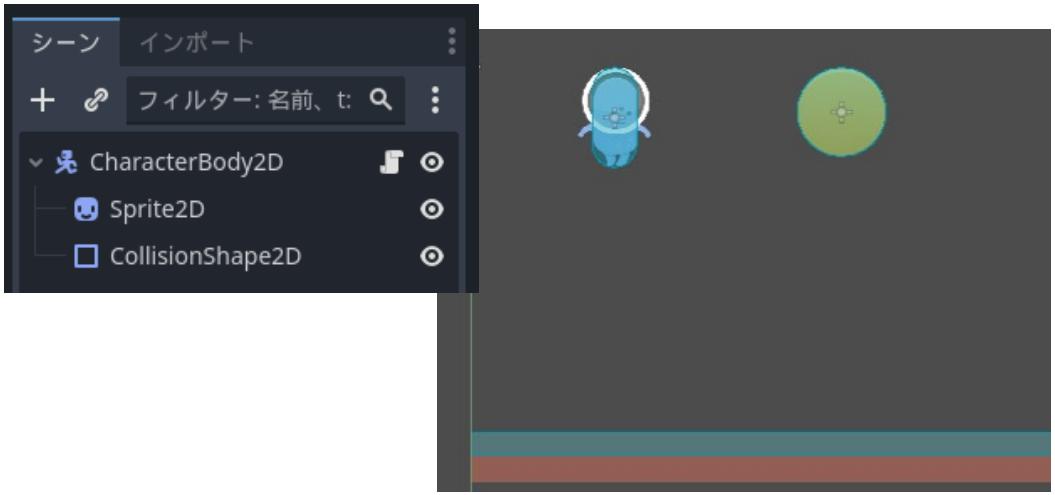
CharacterBody2Dは、RigidBody2Dと同じく物理的な挙動を制御できるノードです。RigidBody2Dが一般的な物理シミュレーションを行うノードに対し、CharacterBody2Dはその名の通りキャラクター特有の動きである「移動・ジャンプ・衝突」などを簡単に実装できるようになっています。

CharacterBody2Dノードにスクリプトをテンプレート有で作成すると、このようにあらかじめたくさんコードが書かれた状態でつくられます。

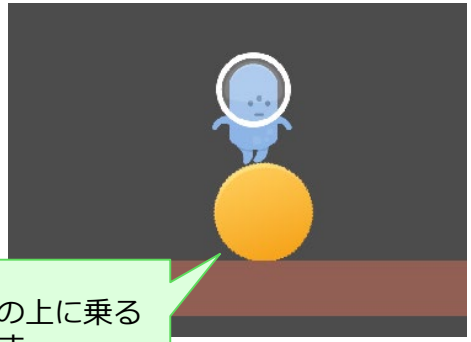



```gdscript
extends CharacterBody2D


const SPEED = 300.0
const JUMP_VELOCITY = -400.0

# Get the gravity from the project
var gravity = ProjectSettings.get_s


func _physics_process(delta):
    # Add the gravity.
    if not is_on_floor():
        velocity.y += gravity * delta

    # Handle jump.
    if Input.is_action_just_pressed("ui_accept") and is_on_floor():
        velocity.y = JUMP_VELOCITY

    # Get the input direction and handle the movement/deceleration.
    # As good practice, you should replace UI actions with custom gameplay acti
    var direction = Input.get_axis("ui_left", "ui_right")
    if direction:
        velocity.x = direction * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0, SPEED)

    move_and_slide()
```

例として、キャラクターを作成しました。ノード自体は、キャラクター画像のためのSprite2Dと衝突判定のためのCollisionShape2Dがあるだけです。



テンプレートのスクリプトだけで、矢印キーでの左右移動と、スペースキーでのジャンプ動作ができ、床やボールとの衝突判定もできます。



ジャンプしてボールの上に乗るようなこともできます。

もちろん細かなパラメータ調整をすれば、移動スピードやジャンプ力なども調整できます。
特にアクション系ゲームのプレイヤーキャラクターには、CharacterBody2Dが最適です。

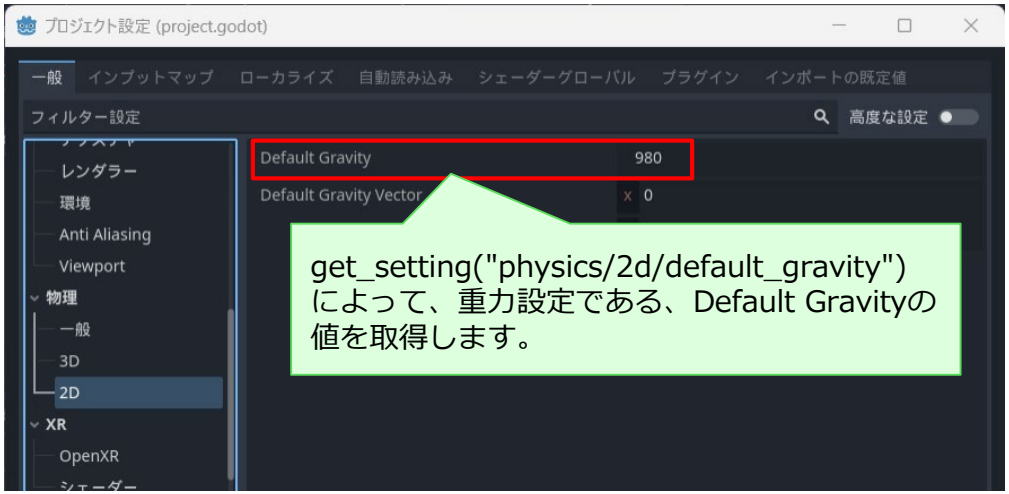
CharacterBody2D（2）

さきほどのテンプレートコードについて簡単に解説します。

```
8    var gravity = ProjectSettings.get_setting("physics/2d/default_gravity")
```

8行目のProjectSettingsはプロジェクト設定を管理するためのクラスです。プロジェクト設定はメニューからアクセスします。

[プロジェクト] > [プロジェクト設定...]

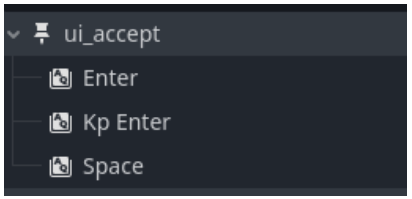


13行目の、**is_on_floor()** はキャラクターが地面に設置しているかどうかを判定する関数です。

```
13      if not is_on_floor():
14          velocity.y += gravity * delta
```

設置していればTrueが返りますが、notで否定していますので、設置していなければ・・・の条件分岐となります。
つまり、キャラクターが空中にいれば、重力設定を使ってキャラクターを落下させます。

17行目はデフォルトアクションでui_acceptに割り当てられたキーを押し、なおかつキャラクターが床に接しているときの条件分岐です。




```
17      if Input.is_action_just_pressed("ui_accept") and is_on_floor():
18          velocity.y = JUMP_VELOCITY
```

床に接していれば、18行目でvelocityプロパティの上方向へ速度を与えることでジャンプ動作を行います。

Input.get_axis() メソッドで左右の移動方向（-1,0,1のいずれか）を取得し、移動もしくは、減速の動作になるような制御を行っています。

```
22      var direction = Input.get_axis("ui_left", "ui_right")
23      if direction:
24          velocity.x = direction * SPEED
25      else:
26          velocity.x = move_toward(velocity.x, 0, SPEED)
```

最後の **move_and_slide()** 関数は、ここまでで得られたvelocityプロパティのy方向、x方向の値を使ってキャラクターを移動させる処理です。

```
28      move_and_slide()
```

このようにCharacterBody2Dは、ゲームキャラクターを動かすための処理がたくさん用意されている、ゲームエンジンならではのノードになっています。

■StaticBody2D

StaticBody2Dは、動かない物体を表現するために使われます。RigidBody2DやCharacterBody2Dの例で使用していた「床」が最もよく使われるパターンです。

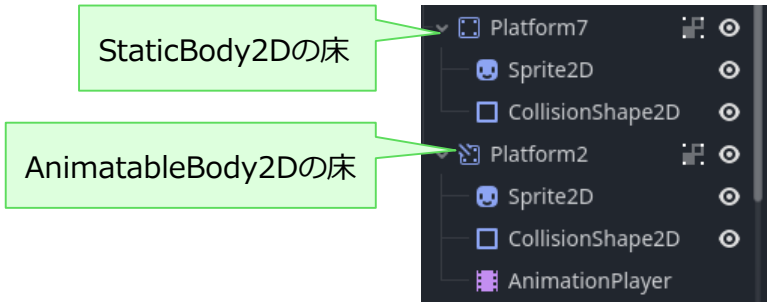
重力の影響を受けないため、空中に浮いている足場や、壁、障害物に最適です。

「動かない」というのは、重力や衝突の影響を受けないのであって、コードの中で座標を指定すれば、もちろん移動させることはできます。

■AnimatableBody2D

よく使われる例としては、プラットフォーマーゲームの動く床などがあげられます。StaticBody2Dの性質を持ちながら、AnimationPlayerと組み合わせて、位置やサイズ、回転などをアニメーションさせることができます。

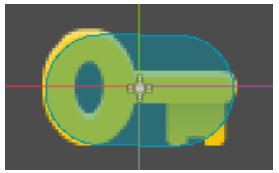
これにより、プレイヤーキャラクターが床として認識しながら、一定の場所を往復移動する床として動作します。



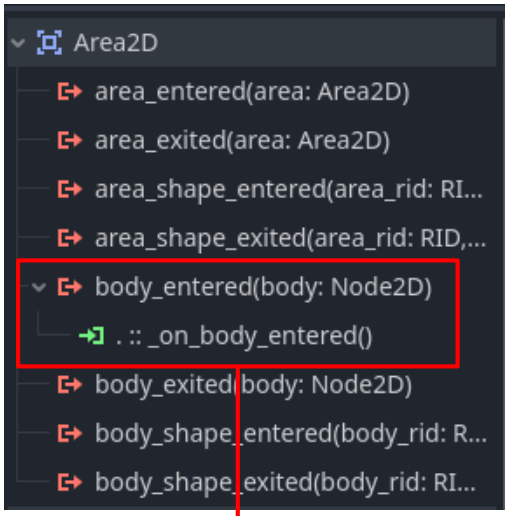
■Area2D

Area2Dは衝突の検出を行いますが、RigidBody2DやCharacterBody2Dのように、物理的な衝突（反発するなど）ではなく、領域内にオブジェクトが存在するかどうかの検出を行います。

例えばこのようなArea2Dノードを使った「カギのアイテム」があったとします。

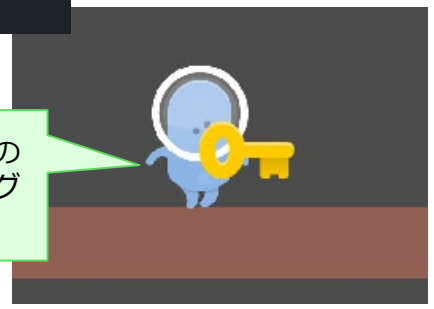


Area2Dにはオブジェクトが領域を出入りするシグナルがあるので、これを利用して、キャラクターが重なったときに、カギをとることができるようなアクションをつくれます。



シグナルが発生すると、対応するメソッドが呼び出されます。

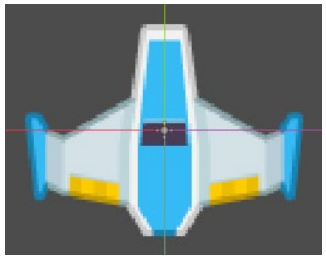
```
14 func _on_body_entered(body):
15     print("カギをとった！")
```



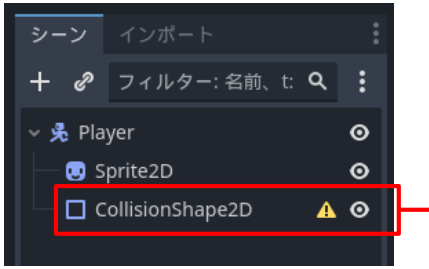
このようにキャラクターとカギのアイテムが重なったときに、シグナルが発生します。

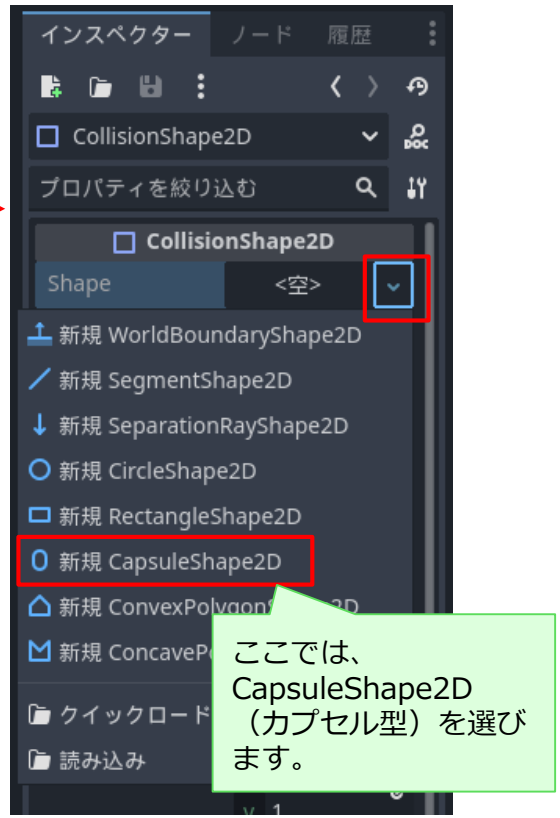
コリジョン設定

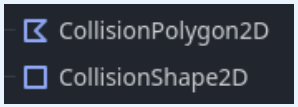
衝突検出のためには、キャラクターに衝突範囲を設定します。
例としてこのような、戦闘機のシーンを考えます。



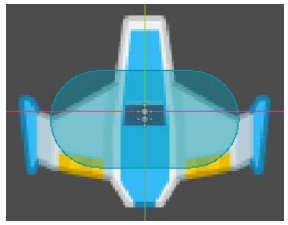
当たり判定の領域は、CollisionShape2D（コリジョン・シェイプ2D）ノードを追加して設定します。

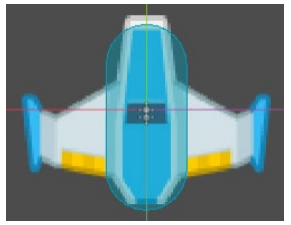


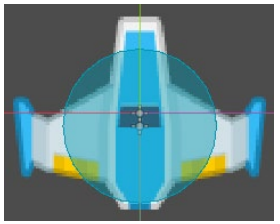




他にもCollisionPolygon2Dがありますが、形状を複雑にすると、当たり判定の処理が重くなります。
同様に、基本シェイプでも円（Circle）や矩形（くけい・Rectangle）などシンプルな形の方が、計算処理は軽くなります。

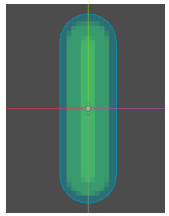




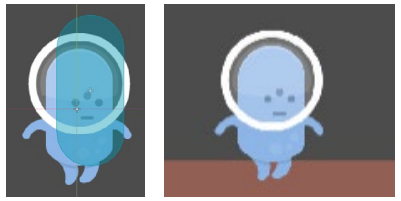


どのような形状と、範囲にするかでゲームの難易度も変わってきます。いわゆる弾幕系シューティングのようなジャンルの場合は、当たり判定の範囲は極力小さくしないと、すぐに弾にあたって、気持ちよくプレイできません。

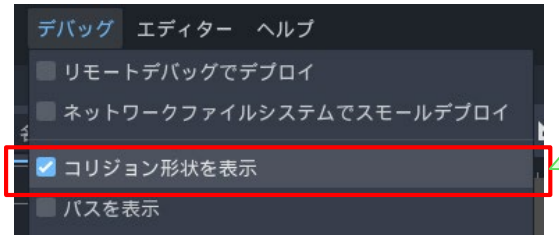
collision（コリジョン）は、そのまま衝突という意味です。ゲーム制作をする上では、必ずこの概念は必要となります。



逆にプレイヤーが発射する弾丸は、少し大き目のコリジョン設定をしておけば、より敵にヒットしやすくなり、爽快感を生み出すことができるかもしれません。



コリジョン設定がずれていると、この例のように地面にめりこんでしまいます。



デバッグメニューの「コリジョン形状を表示」をONにしておくとゲーム実行中にもコリジョンを表示して確認できます。

1-4 その他の要素

- フレームグループ
- シグナル（１）
- シグナル（２）
- カメラ
- サウンド
- リソース
- アドオン

フレーム

ゲームや動画などで「フレームレート」という言葉を聞いたことがあるかもしれません。30FPSや60FPSといった表現で使われます。
FPSはFrames Per Secondの略で、1秒間に何フレーム画面が描き変わるかの意味です。60FPSの場合は1秒間に60回、画面が描き変わって（画面が更新されて）います。

パラパラ漫画が例としては分かりやすいでしょう。パラパラ漫画の1枚1枚が「静止画」として描かれた**フレーム**にあたります。

60FPSの場合は、1秒間に60枚のページが必要となります。
（大変ですね...）

ちなみに一般的なテレビアニメーションは、24FPSでつくられています。

テンプレートスクリプトにも含まれる、_process()メソッドは1フレームごとに呼び出されます。

```
func _process(delta: float) -> void:
```

60FPSの場合は、1秒間に60回呼び出されることになります。
()内のdelta（デルタ）は、前回呼び出されてからの経過時間が設定されます。

60FPSだとdeltaの値は、基本的に、1/60 = 0.0166...ということになります。（多少のズレはあります）

Godotはデフォルトで60FPSで動作しますが、PCの性能によっても左右されます。
実行中のフレームレートはこのメソッドで調べられます。

```
func _process(delta: float) -> void:
    print(Engine.get_frames_per_second())
```

最大フレームレートは、プロジェクト設定の「高度な設定」内に設定項目があります。

もしくは、このように直接指定しても同様に設定できます。

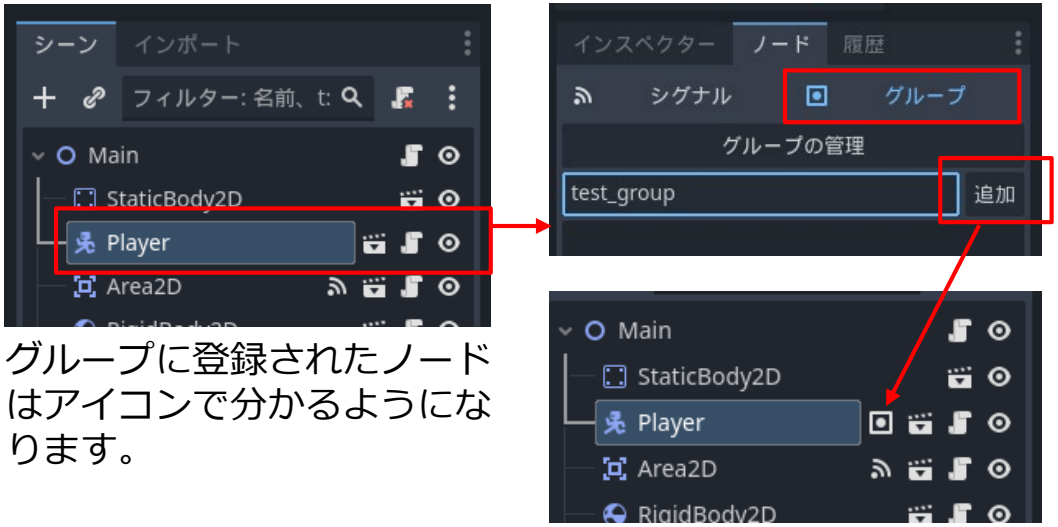
```
func _ready() -> void:
    Engine.max_fps = 30
```

同じFPSでも、First Person Shooterの場合もあります。これは操作キャラクター本人の目線で戦うシューティングゲームのことを指します。

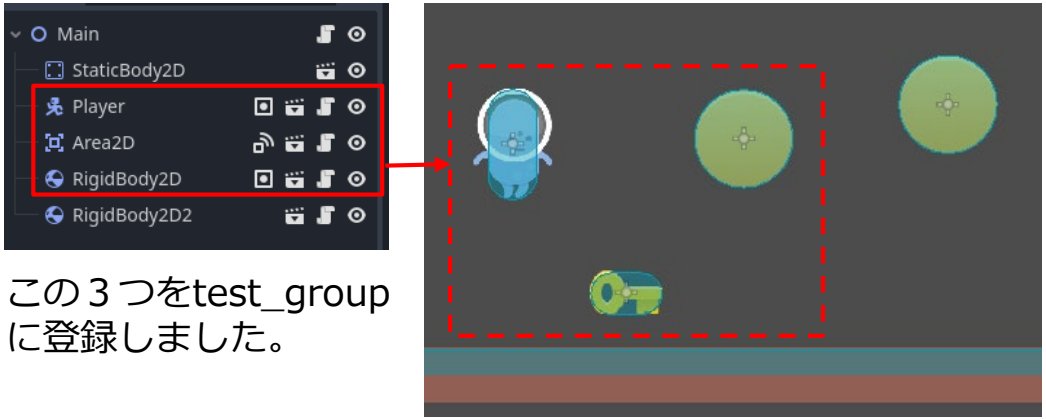
グループ機能を使うと、種類の異なるノード（インスタンス化されたシーンなど）に、まとめて同じ処理を行うことができるようになります。

■エディタで設定

グループに登録したいノードを選び、ノードドックのグループタブに切り替え、グループ名を入力して追加します。



グループに登録されたノードはアイコンで分かるようになります。



この３つをtest_groupに登録しました。

例えばこのように、test_groupに登録されたノード全てに同じ処理を行えるため、非常に効率よくノードを処理することができるようになります。

```
for char in get_tree().get_nodes_in_group("test_group"):
    char.queue_free()
```

■コードで設定

例えば実行中にインスタンス化された敵キャラクターシーンなどは、コードの中でグループ化することになります。
add_to_group(“グループ名”) 関数を使って登録します。

```
func _ready():
    add_to_group("test_group_2")
```

これでさきほどと同じように、グループに対して一斉に処理を行えるようになります。
また、このようにグループに登録されているかをチェックすることもできます。

```
func _on_body_entered(body):
    if body.is_in_group("test_group"):
        print("test_groupに入ってます")
```

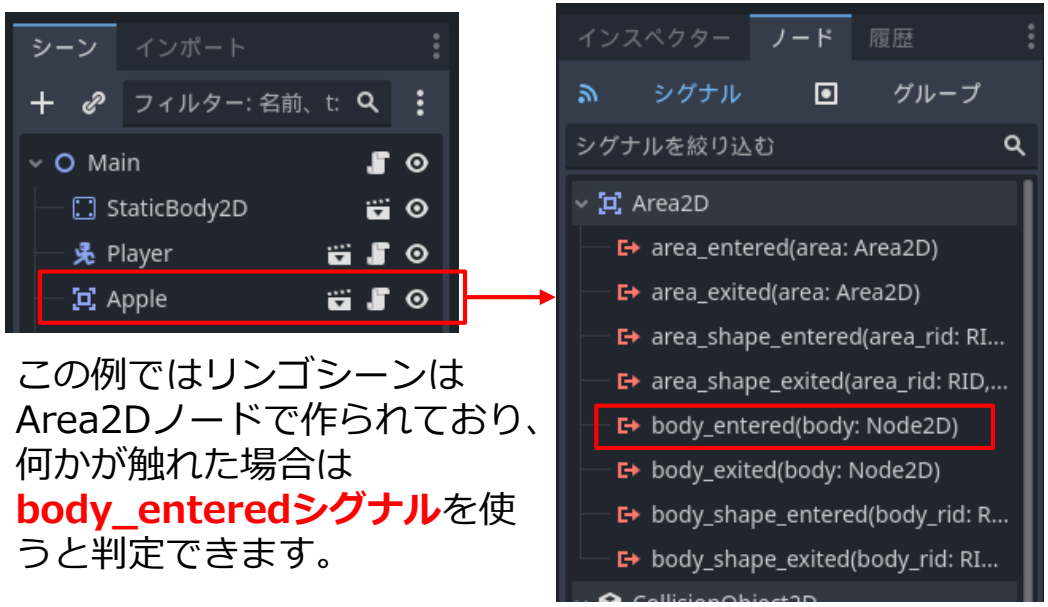
シグナルは各オブジェクトになんらか変化があった（イベントが発生した）際に、特定の信号が送られるしくみです。

ノードにはそのノードに応じたシグナルがいくつか用意されているので、使いたいシグナルを選んで、そのシグナルを受信するためのメソッドを接続します。

リンゴシーンに、プレイヤーシーンが接触した。

接触シグナル発生

あらかじめシグナルに接続されたメソッドを呼び出して処理を行う。



この例ではリンゴシーンはArea2Dノードで作られており、何かが触れた場合は**body_enteredシグナル**を使うと判定できます。

■エディタでの設定

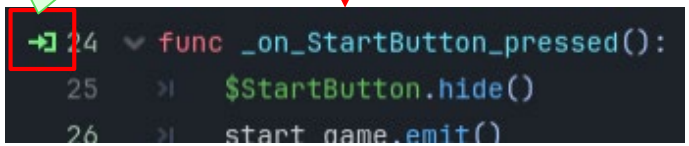
エディタ画面でシグナルの設定をします。対象のノードやシーンを選んで、シグナル一覧から設定したいシグナルを選び、どのスクリプトにメソッドを設置するのか決めます。

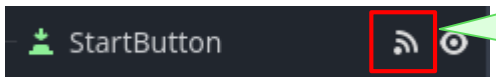
Startボタンが押されると[**pressed()**]シグナルが発生し、[**on_StartButton_pressed()**]メソッドが呼び出されます。

シグナル

ソースコードでは、シグナルによって呼び出されるメソッドであることが分かります。

メソッド呼び出し



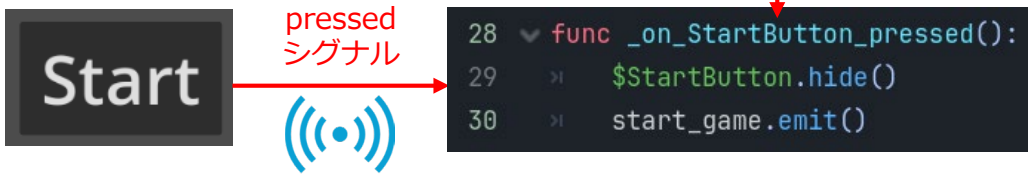


ノード一覧にも、シグナルを使っていることが分かるようなアイコンがつきます。

■コードでの設定

簡単にシグナルを利用するだけならさきほどのエディタで設定する方法で良いですが、プログラム内で管理しておくほうが分かりやすい場合もあります。
connect()関数を使えば、ソースコード内でシグナルの接続設定ができます。

```
5  func _ready():
6      $StartButton.pressed.connect(self._on_StartButton_pressed)
```



この方法を使えば、コードの実行中にインスタンス化されるシーンにも、シグナルの設定ができるようになります。

また同じシーン内だけではなく、異なるシーンのメソッドにもシグナルを送ることができ、一つのシグナルを多数の対象に送ることもできます。

■カスタムシグナル

例えば、ゲームプレイヤーのライフが０になってゲームオーバーになるときを考えましょう。

ゲームの処理としては
・プレイヤーの見た目を変える
・操作できなくする
・ゲームオーバーの表示を出す
・コンティニューボタンを出す
などが必要となりそうです。

しかし「gameoverになった」といったシグナルは用意されていませんので、カスタムシグナルを使います。

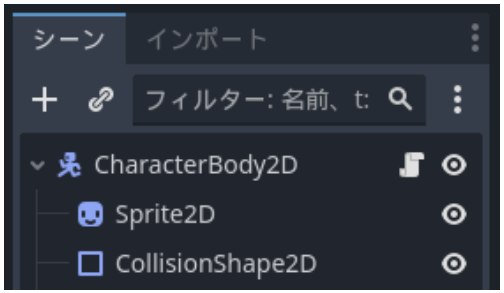

```
9      gameover.emit()
```

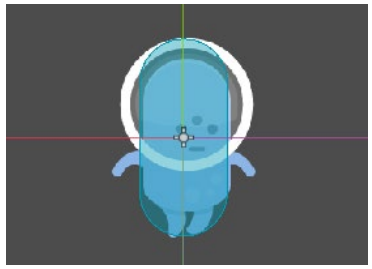
カスタムシグナルを発信する場合は、emit()（エミット）関数を使います。emitは英語で「発する」という意味があります。

よくある横スクロール型アクションゲーム（プラットフォーマー）や、見下ろし型ゲームなど、プレイヤーが画面の中央付近に固定されていて、ステージの要素が動くタイプのゲームがあります。

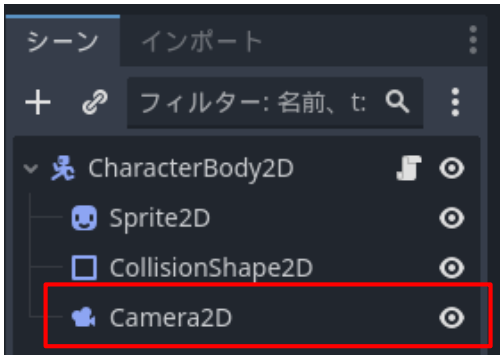
このようなゲームではステージを動かしているのではなく、プレイヤーに追随するカメラを通して画面を映しているため、ステージが動いているように見えています。
GodotではCamera2Dを使います。

このようなプレイヤーキャラクターシーンに追随するカメラを設定します。

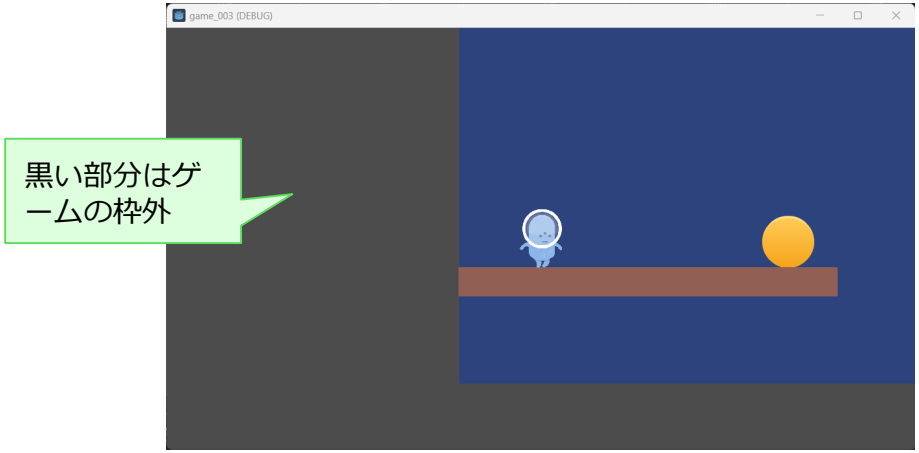




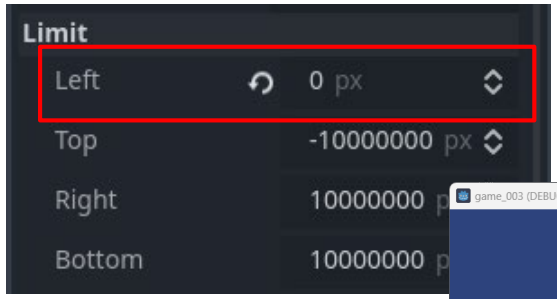
最も簡単な方法は、そのままこのシーンにCamera2Dノードを追加するだけです。

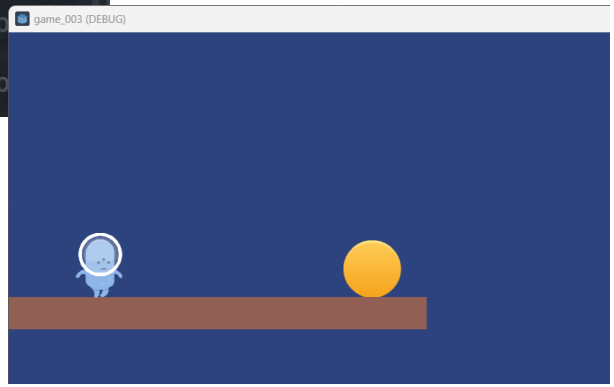


これだけで何もコードを書かなくても、カメラがプレイヤーを中心に動くようになります。



ただこのままでは、キャラクターが端の方にいると、上の画面のようにゲームの枠外も画面に表示されてしまいます。
Limitを設定すれば、それ以上先は表示されなくなります。

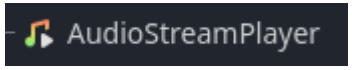




その他にも様々なパラメーターを設定することで、カメラの動作を制御することができます。

ゲームにおいてはプレイ中に流れるBGMや、アクションが起こった際に聞こえる効果音など、音の要素は重要なポイントとなります。

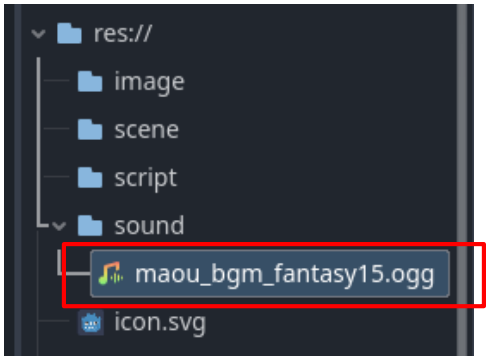
最もシンプルに音を再生するには、AudioStreamPlayerノードを使用します。

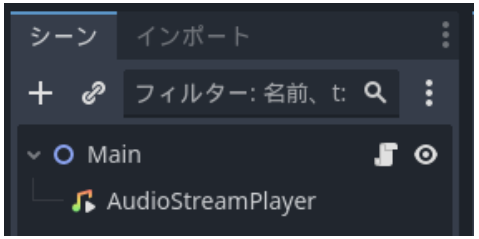


まずは音素材ファイルをプロジェクトに保存しておきます。例として魔王魂®から、素材をダウンロードします。ファイル形式は「ogg」を選びます。

魔王魂®　https://maou.audio/

OGGファイルは圧縮率が高く音質も良いので、ゲーム制作や配信用途に利用されている、音声ファイル形式です。





インスペクターで素材のファイルを割り当てておきます。

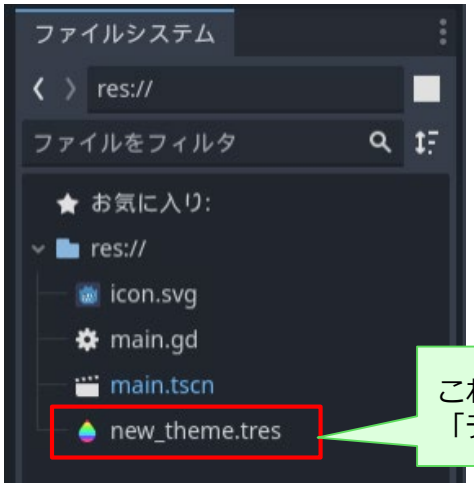
Playingをチェックすれば、再生することができます。

このサンプルでは左右の矢印キーで再生と停止、上下の矢印キーで音量のアップダウンができます。

```
@onready var bgm = $AudioStreamPlayer

func _process(delta):
    if Input.is_action_pressed("ui_right"):
        bgm.play()
    if Input.is_action_pressed("ui_left"):
        bgm.stop()
    if Input.is_action_pressed("ui_up"):
        bgm.volume_db += 0.1
    if Input.is_action_pressed("ui_down"):
        bgm.volume_db -= 0.1
```

リソース

リソース（Resource）はゲーム内で使用する共有可能なデータのことです。具体的には、シーンファイル（.tscn）やスクリプトファイル（.gd）もそうですし、画像や音声、テーマなどもリソースです。



ファイルシステムドックに表示されている「res://」フォルダ内にあるものがリソースです。

※resは「resource」の略です

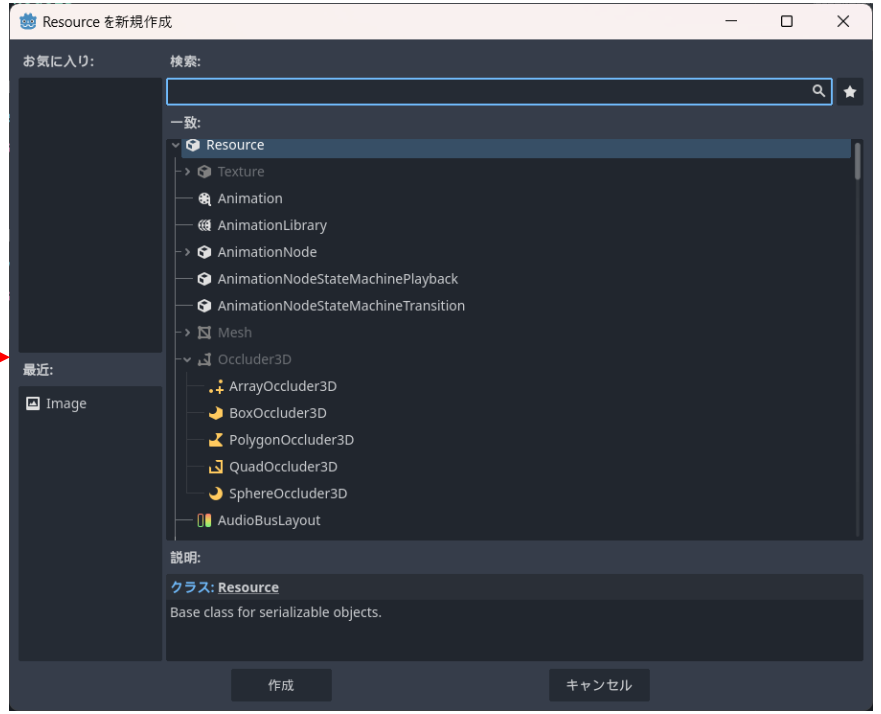
これは、ボタンの見た目を設定した「テーマ」のリソースファイルです。

「.tres」拡張子のファイルはリソースファイルで、実際の中身はこのようなテキストファイルになっています。

```
[gd resource type="Theme" load_steps=2 format=3 uid="uid://d1un1g00ubcbt"]

[sub_resource type="StyleBoxFlat" id="StyleBoxFlat_3tqt3"]
bg_color = Color(0.228525, 0.227999, 0.625401, 1)

[resource]
Button/styles/normal = SubResource("StyleBoxFlat_3tqt3")
```

リソースファイルは個別に作成することもできます。

インスペクタードックにある、新規リソース作成ボタンをクリックします。



ここに一覧表示されるものは、Godot標準のリソースです。数が多くありますが特に覚える必要はありません。

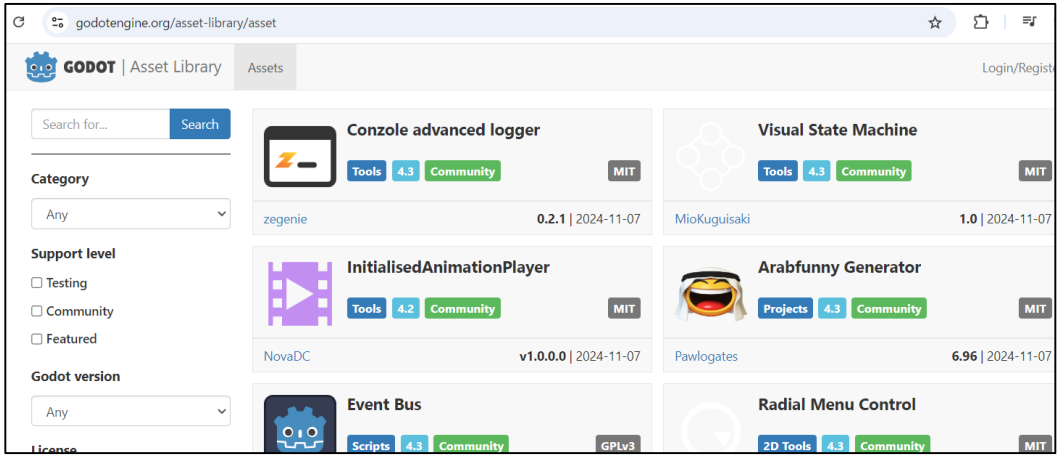
他にも「カスタムリソース」をつくってデータコンテナとして活用できますが、やや複雑ですし、Godotでの開発のはじめのうちは使うことはありませんので、「リソース」という名前が出てきたら、素材やなんらかのデータのことを指しているという認識程度で大丈夫です。

アドオン

アドオン（Addon）は、Godotを拡張するプラグインシステムです。エディタ機能の拡張や、カスタムノード、ゲーム機能の追加など、様々なアドオンがあります。

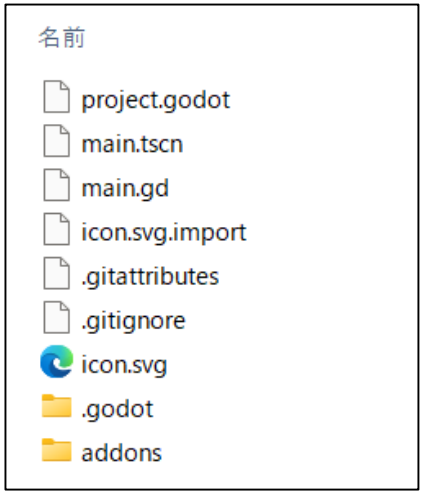
アドオンはたくさんの開発者により作成されて提供されています。もちろん自分で開発することもできます。

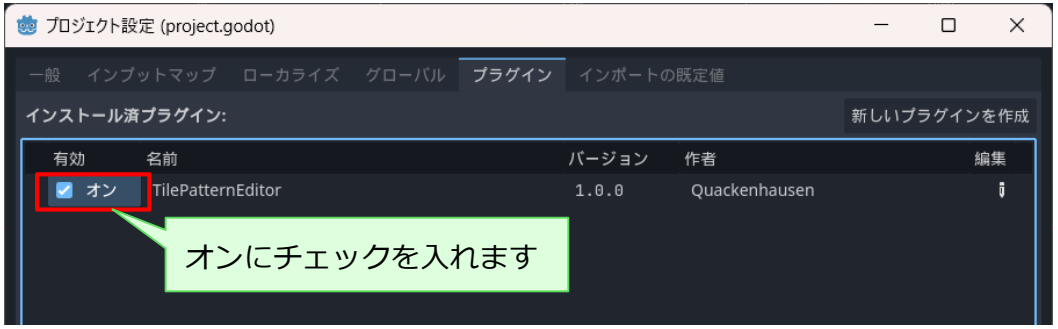
エディタ内や公式Webサイトで入手でき、「ツール」「スクリプト」などのカテゴリ別に分かれています。



基本的にはダウンロードしたzipファイルを解凍し、プロジェクトフォルダ内に、addonsフォルダとして格納します。

その後、プロジェクト設定のプラグインの中で有効化します。
※無効化する場合はチェックを外します。





アドオンの使い方はそれぞれ個別に調べる必要があります。

アドオンそれぞれに「利用ライセンス」があり、たいていは自由に使えるものですが、一部のものは再配布の決まりがあったりします。

YouTubeを検索すると、便利なアドオンを紹介する動画もたくさんあります。Godotでのゲーム開発に慣れてきたら、アドオンを使ってより効率的に制作が進められるようにしても良いですね。

1-5 オブジェクト指向とツリー

- オブジェクト指向プログラミング（1）
- オブジェクト指向プログラミング（2）
- オブジェクト指向プログラミング（3）
- オブジェクト指向のメリット・デメリット
- オートロード（シングルトン）
- ツリー構造（1）
- ツリー構造（2）

オブジェクト指向プログラミング（1）

Godotを学ぶ上でも、プログラミング学習としても知っておいた方が良いものとして「オブジェクト指向プログラミング」があげられます。

どのようなプログラミング言語でも「保守性」「拡張性」を保つ必要があります。膨大なソースコードを管理するためにはなんらかの手段を用いて、プログラミングを行わなければなりません。その手段の一つがオブジェクト指向プログラミングです。

■基本概念

- カプセル化
- 継承
- ポリモーフィズム
- オブジェクト
- クラス
- インスタンス

プログラミングで扱う要素は基本的には大きく「データ」と「処理」に分けられます。

◆データ
・レベル
・攻撃力
・スピード

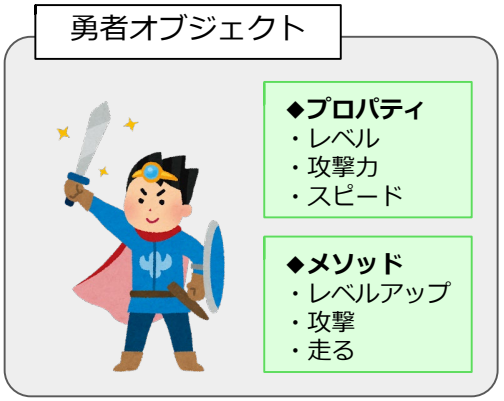
◆処理
・レベルアップ
・攻撃
・走る

◆データ
・国語の点数
・算数の点数
・社会の点数

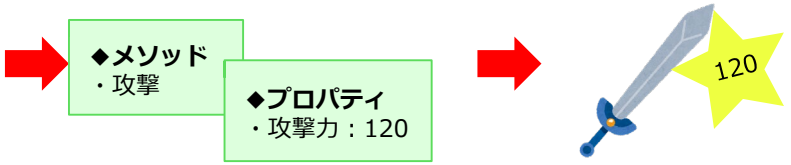
◆処理
・平均点を計算
・最高点を計算
・グラフを表示

この「データ」「処理」をひとまとめにしたものが「**オブジェクト**」です。

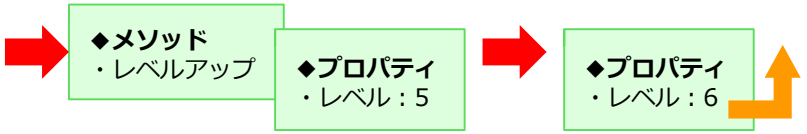
プログラミング用語で表すとデータは「プロパティ」、処理は「メソッド」とよびます。（※プログラミング環境によって呼び方は変わります）



勇者オブジェクトに対して攻撃メソッドを呼び出せば、勇者は自身のプロパティである「攻撃力」に応じた攻撃をくりだします。外部から攻撃力の値を指定する必要はありません。



同じようにレベルアップのメソッドを呼び出せば、自動的に現在のレベルに１を足したレベルになります。



このようにオブジェクト自身が密接に結びついたプロパティとメソッドを持っているため、外部の影響を受けずに単体で管理することができます。
このような概念を「**カプセル化**」とよびます。オブジェクトというカプセルにまとめてしまうイメージです。

オブジェクト指向プログラミング（２）

ゲーム内に登場するものをオブジェクトとして定義すると、例えば・・・

見た目など少し設定が異なるキャラクター

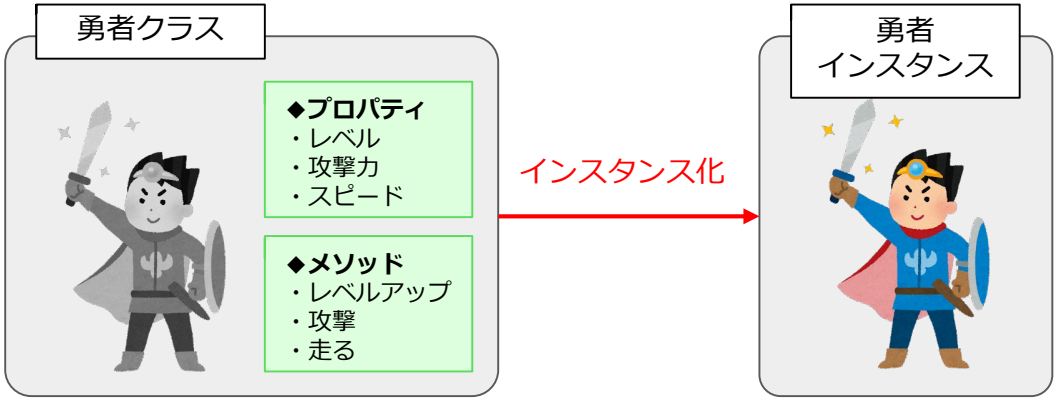
同じ敵キャラがたくさん登場する

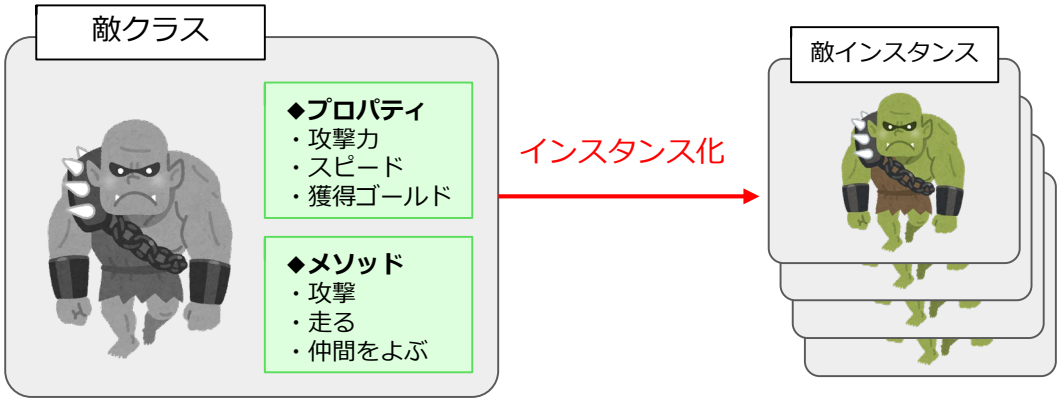
同じオブジェクトを大量にコピーしたり、個別に少しだけプロパティを変えたりする必要があります。
例えば、敵キャラやオブジェクトをたくさんコピーしてから敵の強さを変えたりなど、オブジェクトの仕様が変わると、コピーした全てに同じ変更を行うことになります。

そこで、オブジェクトの設計図を用意しておき、設計図から都度オブジェクトを作り、必要であればオブジェクトごとにプロパティを変更し、仕様が変更される場合は設計図を変更するようなしくみにします。

これを「**クラス**」「**インスタンス**」とよびます。

設計図にあたるのが「クラス」で、クラスから生成したものが「インスタンス」です。また、生成することを「インスタンス化」とよびます。
インスタンス化することではじめて、プログラムの中で使用することができるようになります。



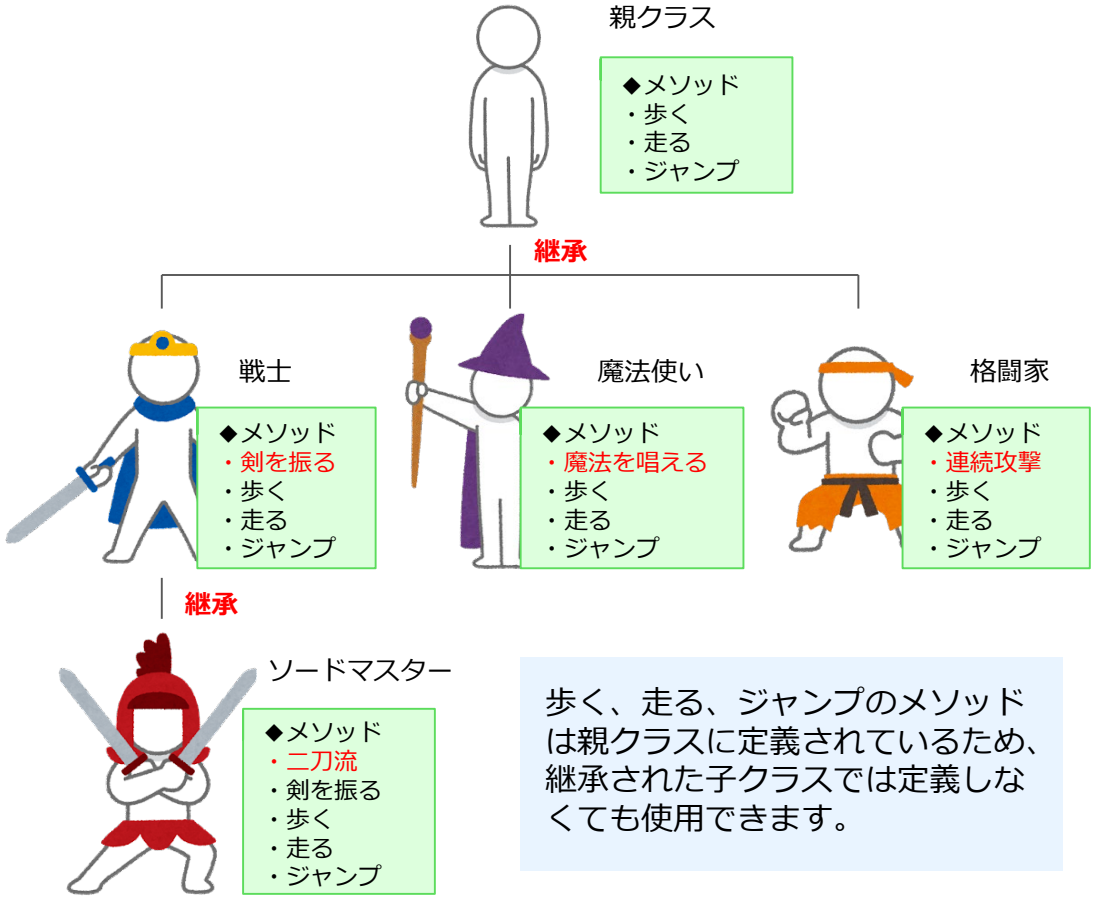


Godotでは、各ノードや、いくつかのノードで構成されるシーンが、クラスにあたります。

オブジェクト指向プログラミング（3）

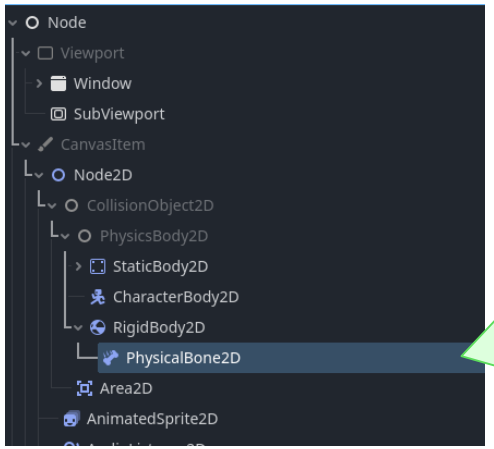
ツリー構造に似ていますが、「**継承**」は特にオブジェクト指向プログラミングで用いられている概念です。

親クラス（基礎となるもの）は基本となるアクション（メソッド）を持っています。キャラクターのバリエーションが増えるたびに、また同じ基本アクションをそれぞれに定義するのは無駄が多くなってしまいます。
継承を用いると、親クラスの特徴である基本アクションはそのまま子に受け継がれ、さらに新しい特徴を追加することができます。



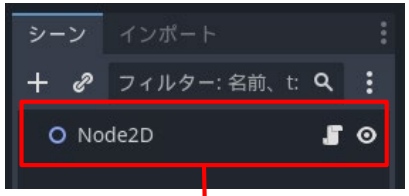
歩く、走る、ジャンプのメソッドは親クラスに定義されているため、継承された子クラスでは定義しなくても使用できます。

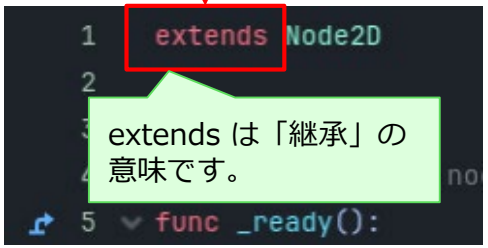
ノードそのものも、この継承の概念が使われています。
ノード作成時に表示されているツリー状の関係が、そのまま継承されている関係性を示しています。



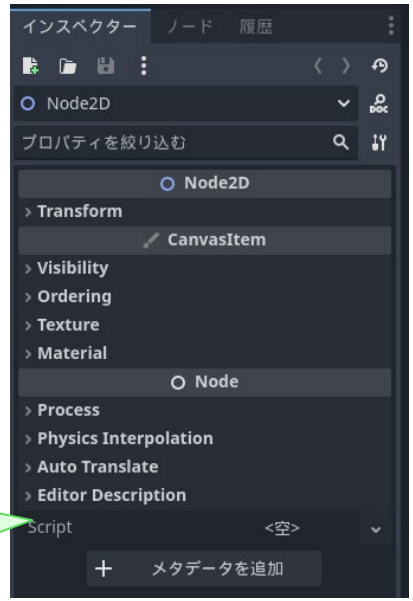
PhysicalBone2Dノードは
親ノードである
・RigidBody2D
・Node2D
・Node
のプロパティやメソッドを受け継いでいます。

新規スクリプトを作成した際には、そのノードを継承する形でスクリプトが作成されます。また、インスペクターでは親ノードから継承されたプロパティを扱えます。





extends は「継承」の意味です。



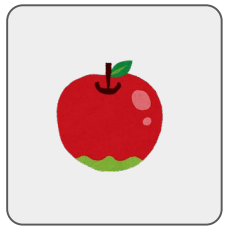
Node2Dは、CanvasItemとNodeを継承していることが分かります。

オブジェクト指向のメリット・デメリット

オブジェクト指向のメリットとしては、ゲーム内に登場するあらゆるものをオブジェクトとしてカプセル化することで、プロジェクト全体をメンテナンスしやすくできることです。

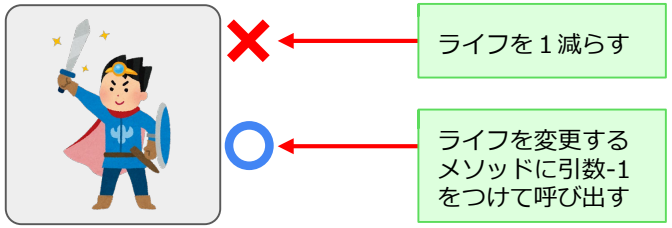
特にゲーム制作においては、個々のオブジェクト（クラス）が再利用しやすくなることは大きなメリットです。



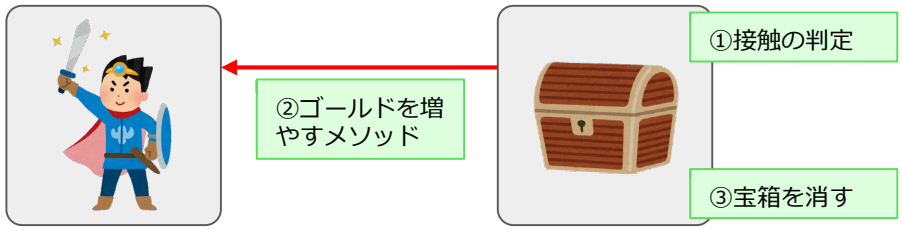



Playerである勇者はゲーム内に一人しかいないが、宝箱やモンスター、アイテムは何度も登場し、なおかつ様々なバリエーションも存在するので、オブジェクト指向設計が有効になる場面です。

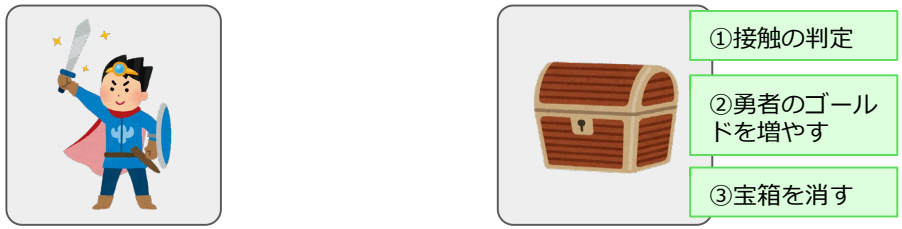
オブジェクト指向プログラミングをする上で、重要なことにオブジェクト個々の内部の状態を「**隠ぺい化**」することがあげられます。
例えば勇者のレベル、ライフポイント、攻撃力などのプロパティが外部から自由に変更できてしまうと、不必要なエラーや誤用の元になってしまいます。そのため、オブジェクト同士はメソッドを介してやりとりを行います。



ただ、この勇者と宝箱のような単純な場合でも、厳密なオブジェクト思考設計にすることによって、より複雑になってしまうこともあります。



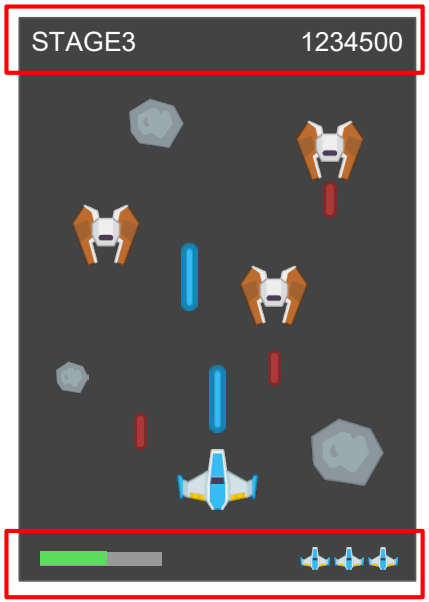
ゲームの規模や内容によっては、このように宝箱側で直接勇者のプロパティを変更してしまうような、シンプルな設計にすることも考えられます。



特に個人でつくる小規模なゲームであれば、再利用性などを考えるよりも、まずは短期間で簡単に仕上げられることを優先することも大事です。

オートロード（シングルトン）

ここまでの説明のように、Godotではシーン単位でゲーム内に登場するオブジェクトをつくって、それぞれでゲーム内のデータや挙動を管理します。



しかし、例えばシューティングゲームの得点や、自機の残りライフなどは、ゲーム内に存在するのは１つだけですし、シグナルなどを使わずにゲームのどこからでもデータにアクセスできれば、簡単に管理できて分かりやすい気がします。

特にゲーム全体の管理や進行に関する部分は、ゲーム画面に表示する個々のオブジェクトとは役割も異なります。

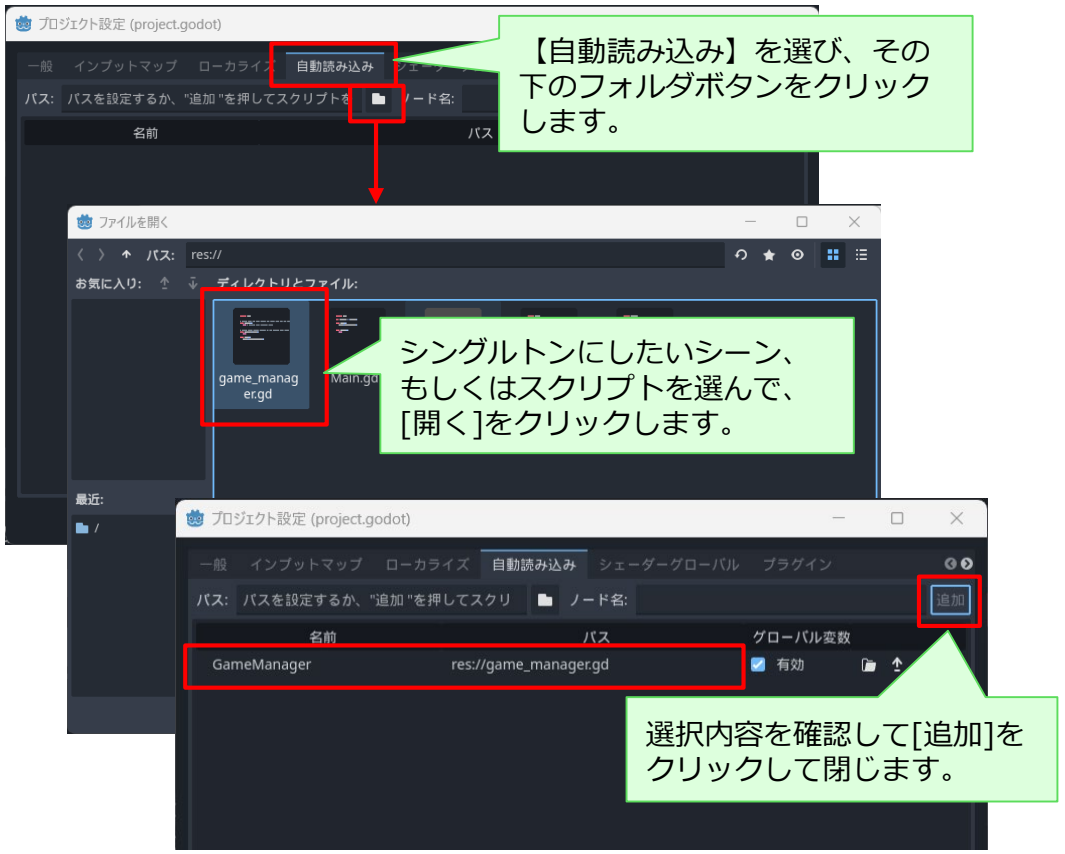
プログラミングの手法としてデザインパターンとよばれるものがありますが、そのうちの一つに「シングルトン(singleton)」があります。これは、一度だけインスタンス化でき、グローバルにアクセスできるようなクラスのことです。

> グローバルとは、プログラムのどこからでもアクセスできるようなものを指します。

Godotでも「**オートロード**」機能を使って、シングルトンを実装し、どこからでもアクセス可能なインスタンスを作ることができます。

あらかじめシングルトンにするシーン、もしくはスクリプトを作成しておきます。ここでは「game_manager.gd」を作っておきます。

[プロジェクト] > [プロジェクト設定...]



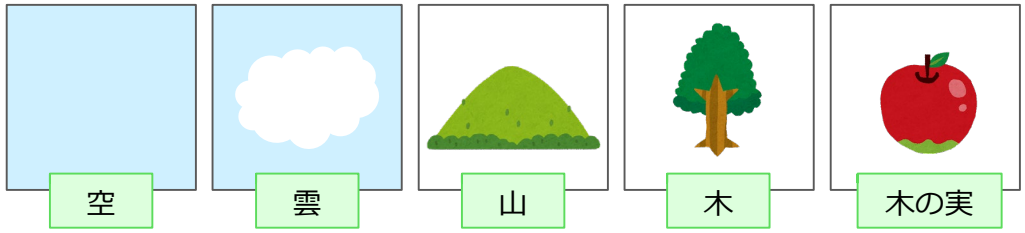

```gdscript
1   extends Node
2
3   var score = 10000
```

```gdscript
GameManager.score += 500
```

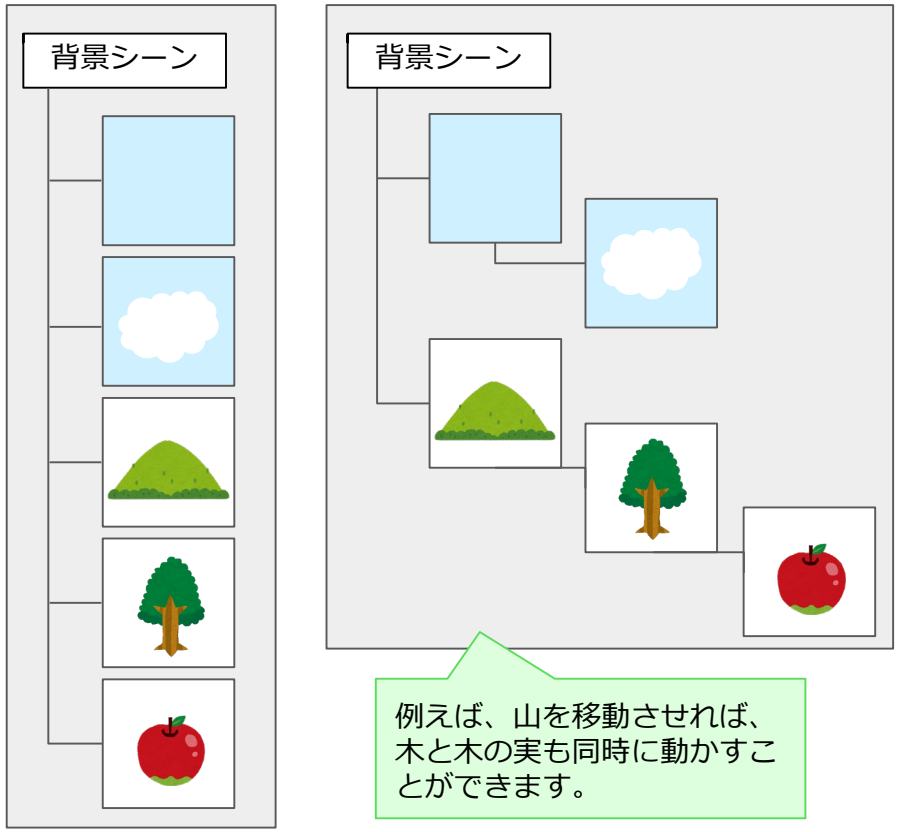
これで、どのシーンからでも直接GameManagerの変数へアクセスすることができます。

ツリー構造（1）

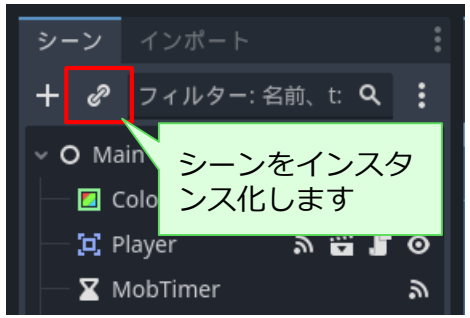
ここまでの説明でも少し出てきた「ツリー構造」について解説します。例えばあるゲーム内にこのような要素があり、背景シーンを作るとします。

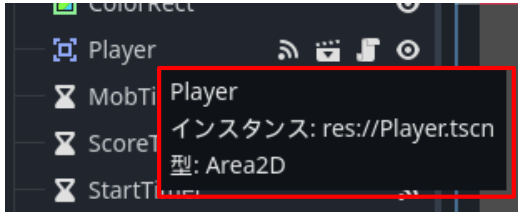


全てが並列なものと、ツリー構造のものを比べると、ツリー構造の方が管理しやすいのが分かります。

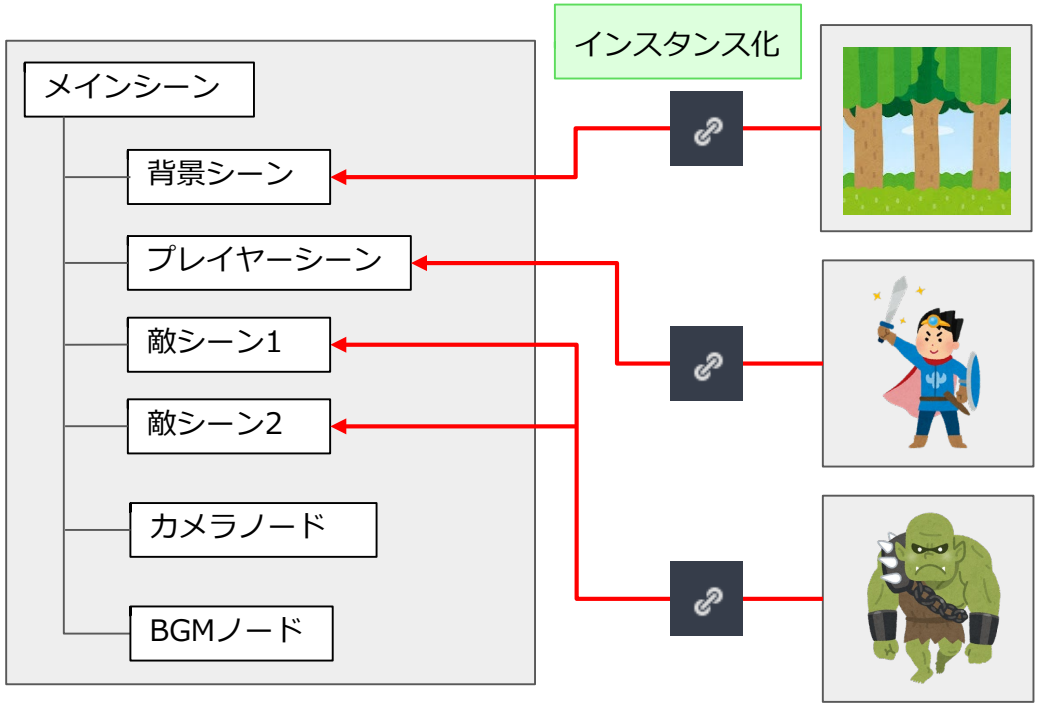


シーンもノードもどちらもツリー構造で管理します。シーンは別のシーンへ「インスタンス化」してツリーに入れることができます。





Playerにマウスオーバーすれば、インスタンス化されていると分かります。

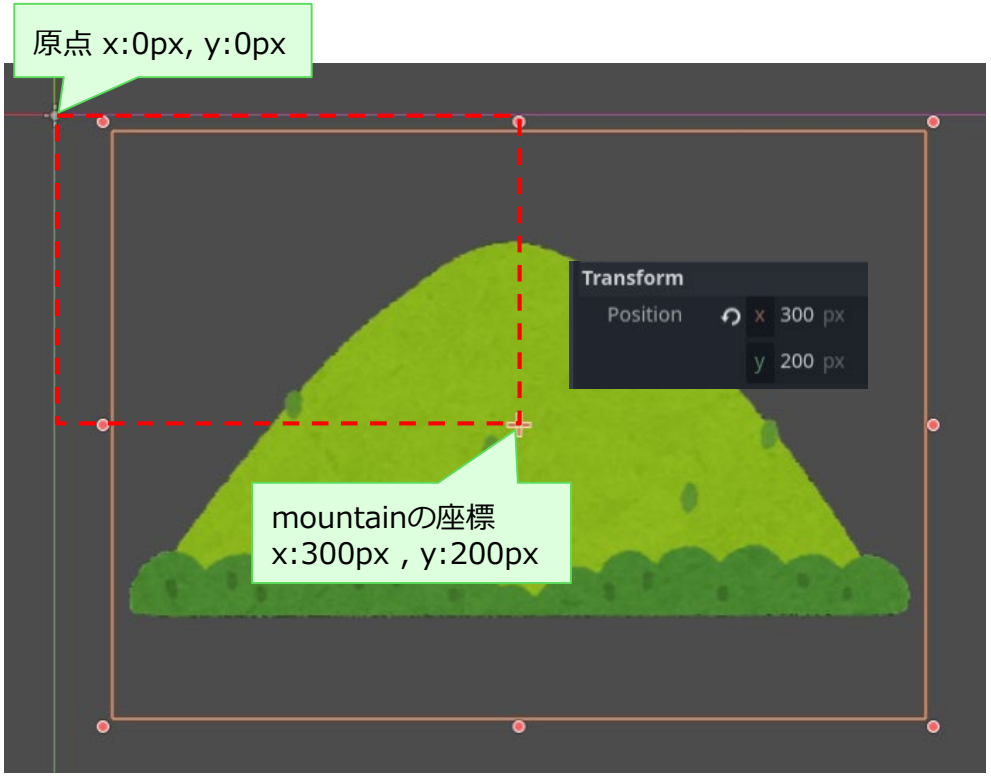


ゲーム内の要素をシーンごとに作成し、インスタンス化することで、要素を個別に管理したり、シーンを複製することもできます。

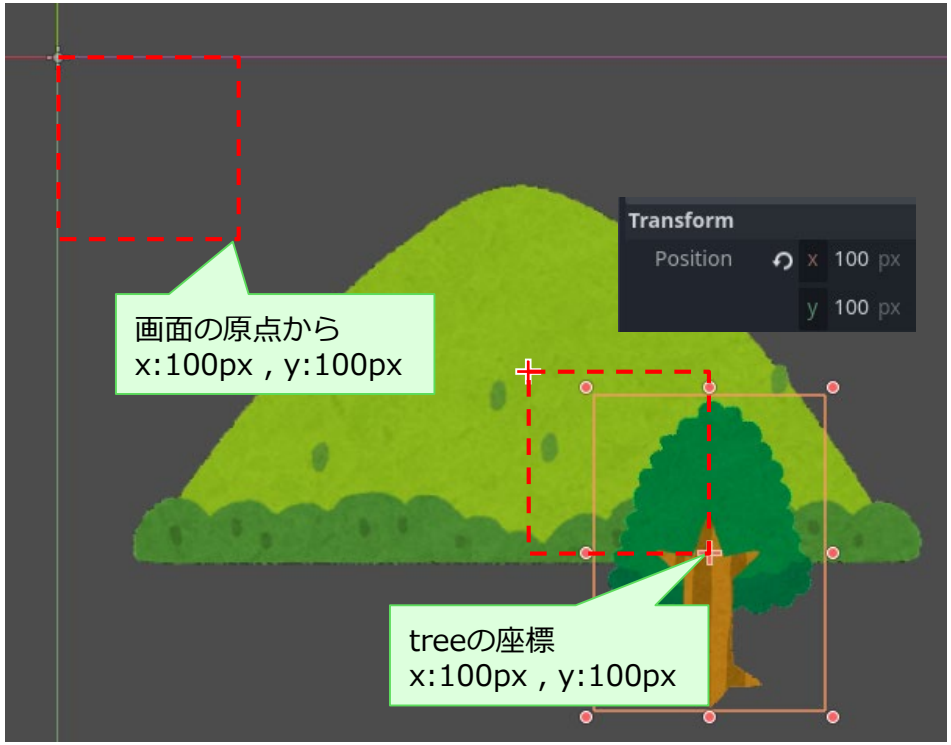
ツリー構造（2）

ツリー構造は画面の中の配置にも影響を与えます。
ノードが親子関係にあるとき、子ノードは親ノードの中心点を原点として配置されます。

また親ノードを移動させれば、子ノードも一緒に移動されます。



mountainノードを、x:300,y:200 の座標に指定した場合は、画面（ルート）の原点を基準に配置されます。



treeノードを、x:100,y:100 の座標に指定した場合は、画面の減点を基準とするのではなく、親要素であるmountainノードの原点（中心）を基準に配置されます。