基本テキスト

2章

Godotエンジンの基本操作

ver.1.0.0

この章では実際にゲーム制作をしていく際に、特によく行う基本操作についてソースコードも合わせて説明しています。操作に迷ったら見返すようにしましょう。

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。

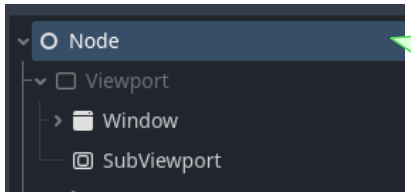
2-1 ノードとシーンの操作

- ノードの作成
- ノードの操作
- ノード間のアクセス（１）
- ノード間のアクセス（２）
- ノード間のアクセス（３）
- ノード間のアクセス（４）
- シーンの作成
- シーンをインスタンス化する【エディタ画面】（１）
- シーンをインスタンス化する【エディタ画面】（２）
- シーンをインスタンス化する【Script】

ノードの作成

はじめにゲームのベースとなるシーンが必要ですので、そのルートノードをつくります。2Dゲームの場合は、2Dシーンノードを選択し「Node2D」を作成するのが一般的です。

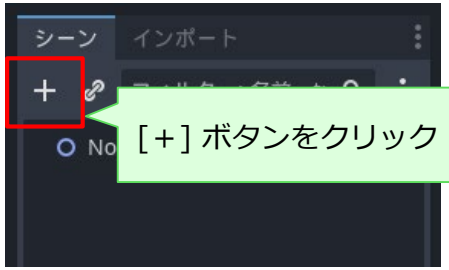
ルートには画面表示に関することが不要なら、その他のノードから最も上のレベルの「Node」にしても良いです。



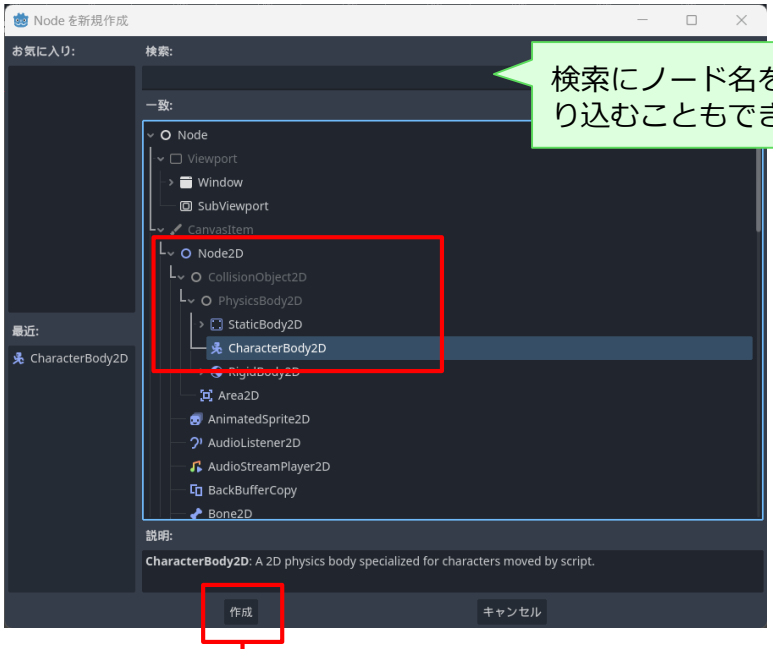
Nodeにしておくと、見た目でもルートであることが分かりやすくなります。

これは、ゲーム全体をコントロールしたり、この後に作成していく、各シーンの入れ物のような役割を果たすメインシーンになります。
そのため「Main」「Game」「World」などの名前にすることがよくあります。

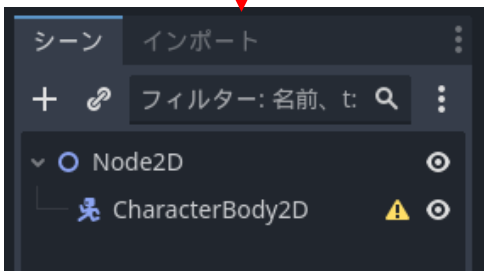
さらにノードを追加してみます。



開いたウインドウの中で、ここでは試しにNode2Dを展開していき「CharacterBody2D」を選択し、[作成] します。



検索にノード名を入力して絞り込むこともできます。



このようにして、必要なノードを追加して、シーンを構成していきます。

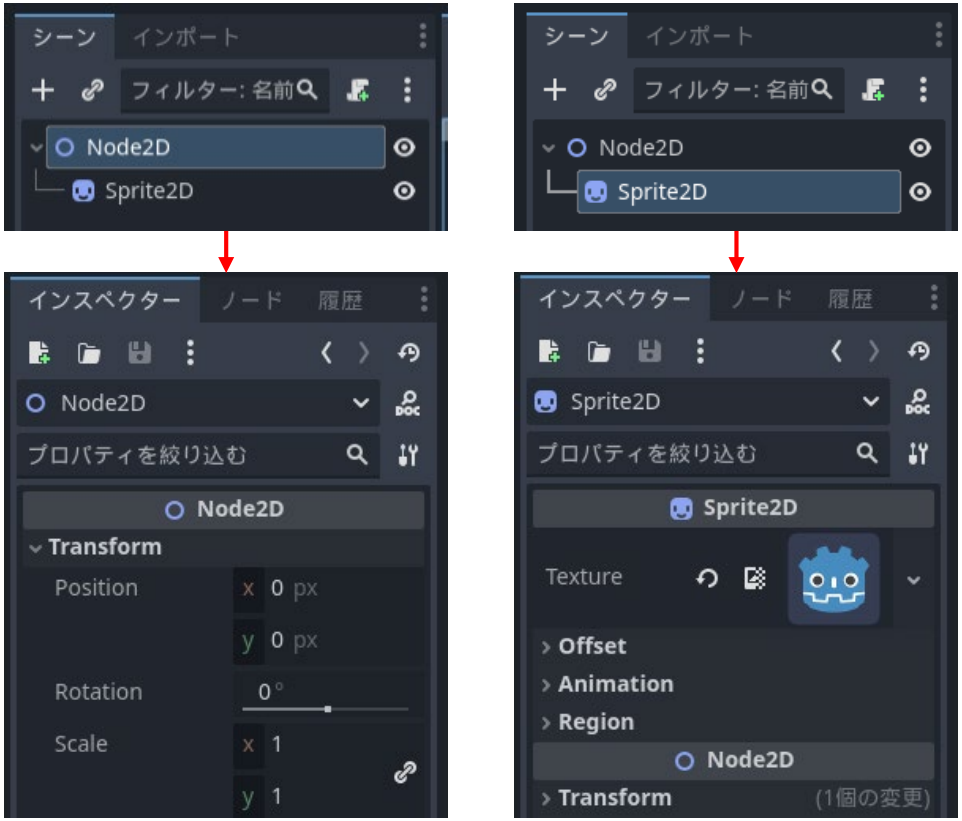
ノードは後からツリーの順番を変えられますし、不要なものは削除できます。
良く使うノードはある程度決まっていますし、慣れてくると自分がやりたいことを実現するためのノード選びもできるようになります。

ノードの操作

ノードにはいろいろな種類がありますが、ノードに対して様々な設定を行ったり、スクリプトをアタッチしたりしながらゲームをつくっていきます。

■プロパティの変更

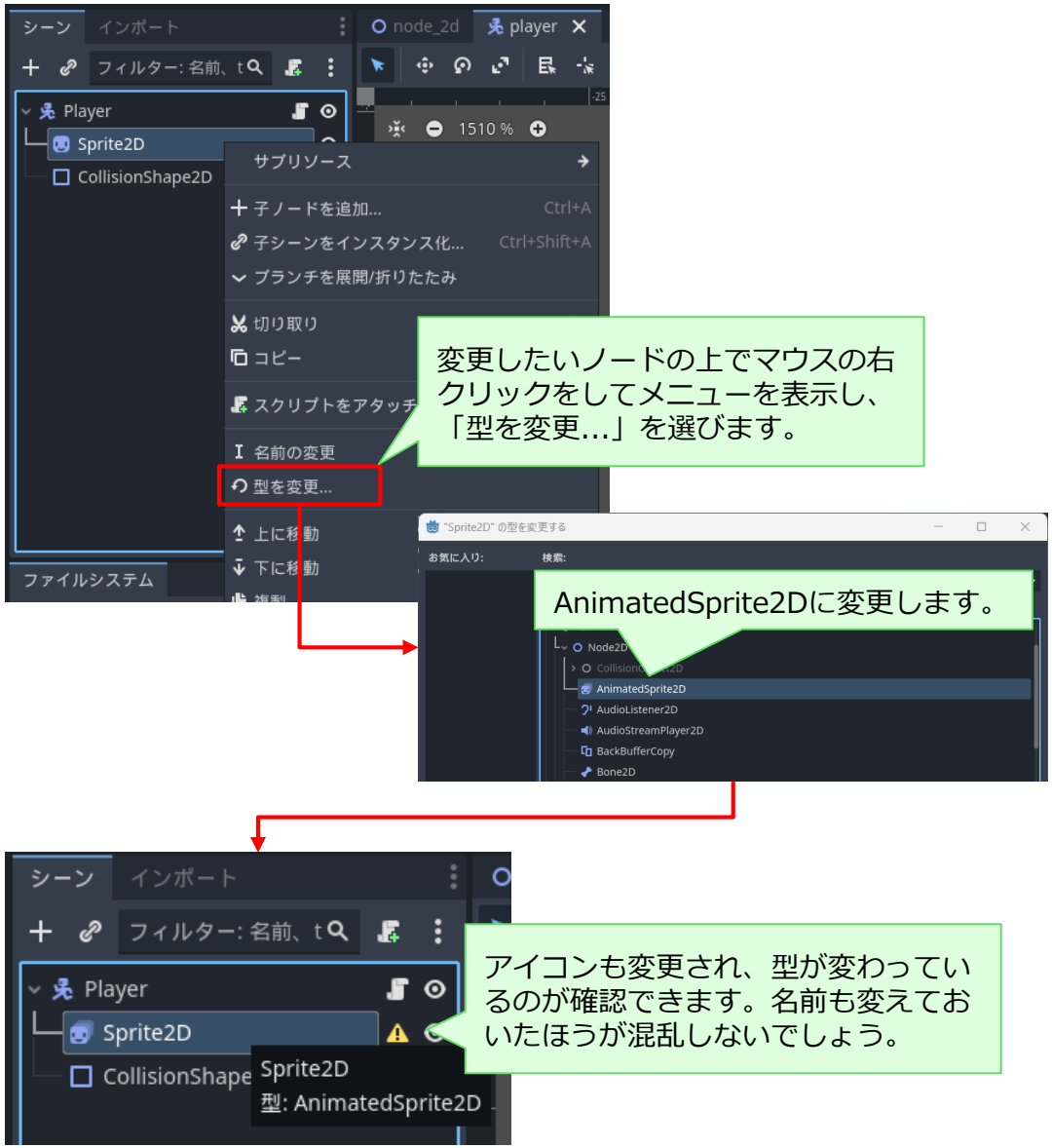
ノードのプロパティ（パラメータ）の変更は、インスペクターを用いて設定をします。スクリプト内で実行時に設定することもできます。



シーンドックで選んだノードによって、インスペクタードックの内容が変わります。間違って異なるノードを選んで変更してしまわないように注意が必要です。

■ノードの種類変更

ノードを作成した後から、ノードの種類を変更したい場合に、作り直さなくても変えることができます。

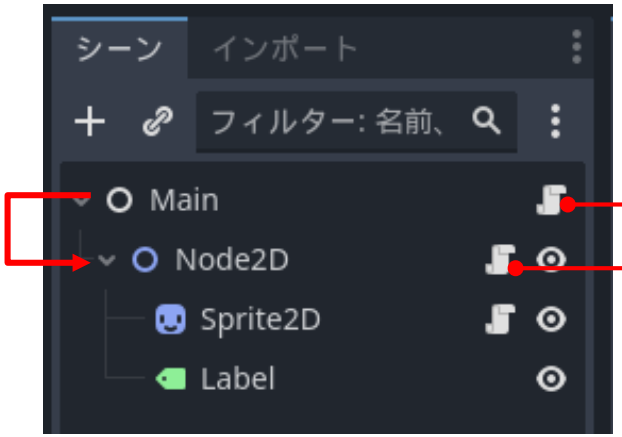


ノード間のアクセス（１）

ノード間でそれぞれの変数や関数に直接アクセスすることができます。ツリー構造をたどるようにアクセスします。

■子ノードへのアクセス

ノードツリー内で1つ下の階層（子ノード）のノードのメソッドや変数に
アクセスするには、get_node()関数を利用します。



MainからNode2Dへのアクセス

```gdscript
extends Node

func _ready():
    print(get_node("Node2D").var_test)
    get_node("Node2D").test()
```

```gdscript
extends Node2D

var var_test = "Node2Dの変数"

func test():
    print("Node2Dの関数を呼び出し")
```

実行結果

```
Node2Dの変数
Node2Dの関数を呼び出し
```

Mainのスクリプトから、子ノードである
Node2Dの「var_test変数」と
「testメソッド」にアクセスしています。

get_node(“ノード名”)

ノード名を指定して子ノードへアクセスします。
また代わりに $ を使用して短縮表記も行えます。

```gdscript
func _ready():
    print($Node2D.var_test)
    $Node2D.test()
```

■子ノードをまとめて取得

get_children()関数を使うと、子ノードをまとめて取得する
ことができるため、子ノード全てに対して同じ処理をする場
合などに便利です。

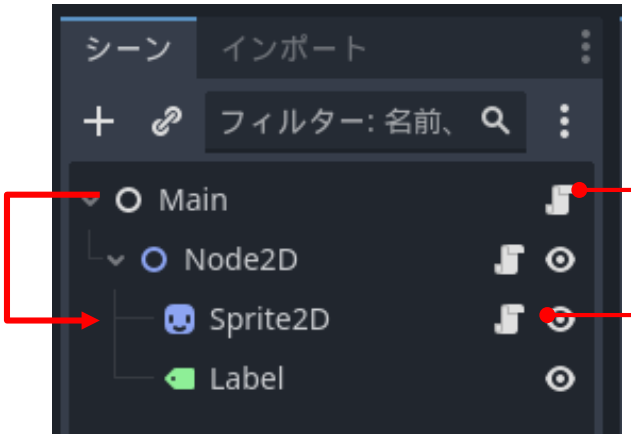
```gdscript
var children = get_children()

for child in children:
    print(child.name)
```

children（チルドレン）は英語で
「子ども」の意味です。

ノード間のアクセス（２）

■孫ノードへのアクセス

ノードツリーの階層で、２つ以上下の階層は「孫」にあたるため、例えば２階層下の
ノードにアクセスするには、子ノードの子ノードのようにアクセスします。



MainからSprite2Dへのアクセス

```gdscript
extends Node

func _ready():
    print(get_node("Node2D").get_node("Sprite2D").var_test)
```

get_node()を2回使うことで、2階層下のノードにアクセスします。

```gdscript
print(get_node("Node2D/Sprite2D").var_test)
```

/（半角スラッシュ）を使って、ファイルパスのようにアクセスもできます。
こちらの方が直感的で分かりやすいですね。

```gdscript
extends Sprite2D

var var_test = "Sprite2Dの変数"
```

実行結果

```
Sprite2Dの変数
```

■ドラッグして入力 ! 要注意

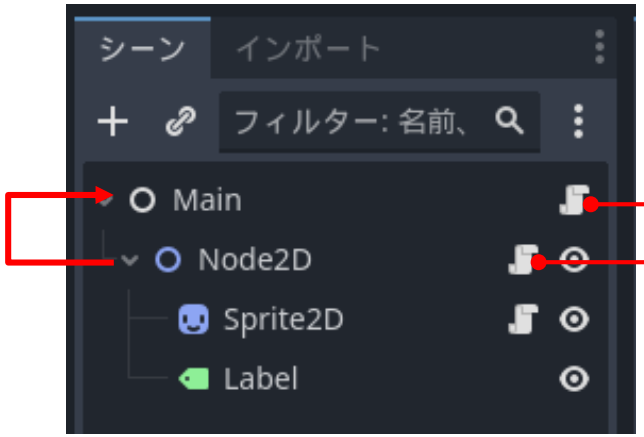
ノードツリーから直接スクリプトへドラッグすることで、
$を使用した短縮表記で入力されます。


```gdscript
func _ready():

    $Node2D/Sprite2D
```

ノード間のアクセス（３）

■親ノードへのアクセス

ノードツリー内で上の階層のノードのメソッドや変数にアクセスするには、
get_parent()関数を利用します。



Node2DからMainへのアクセス

```gdscript
extends Node

var var_test = "Mainの変数"

func test():
    print("Mainの関数を呼び出し")
```

```gdscript
extends Node2D

func _ready():
    print(get_parent().var_test)
    get_parent().test()
```

実行結果

```
Mainの変数
Mainの関数を呼び出し
```

Node2Dのスクリプトから、親ノードであるMainの「var_test変数」と「testメソッド」にアクセスしています。

get_parent()

親ノードへアクセスします。また代わりに、../を使用して短縮表記も行えます。parent（ペアレント）は英語で「親」の意味です。

複数の階層をたどる場合は、get_node()と同じく、
get_parent().get_parent()のように使います。

```gdscript
func _ready():
    get_parent().get_parent().test()
```

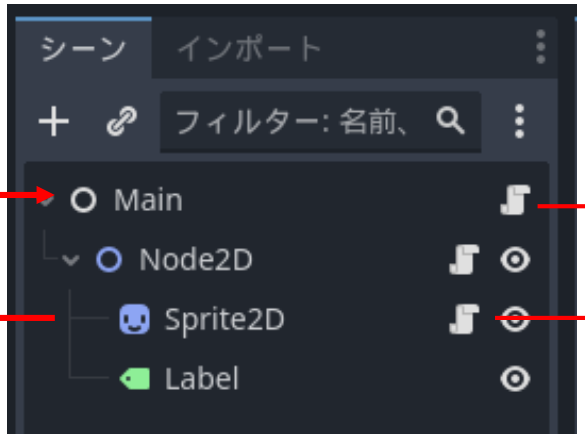
get_node()とパス”../”を使う表記でもアクセスできます。

```gdscript
func _ready():
    get_node("../").test()
    $"../".test()
```

ノード間のアクセス（４）

■ルートノードへのアクセス

ノードを親子関係を使ってアクセスする以外にも、一番上の階層であるルートノードへアクセスしてから、対象のノードへアクセスする方法もあります。



Sprite2DからMainへのアクセス

```gdscript
extends Node

func test():
    print("Mainの関数を呼び出し")
```

実行結果

```
Mainの関数を呼び出し
Mainの関数を呼び出し
Mainの関数を呼び出し
```

```gdscript
func _ready():
    get_tree().get_root().get_node("Main").test()
    get_tree().root.get_node("Main").test()
    get_node("/root/Main").test()
```

パスで直接rootからたどるようなアクセスもできます

get_tree()
シーンツリー（SceneTree）にアクセスします。

get_tree().get_root()
シーンツリーのメソッドで、ルートノードを参照します。

get_tree().root
シーンツリーのプロパティで、ルートノードを参照します。

他にも、find_node()関数を使って、目的のノードを探す方法もあります。

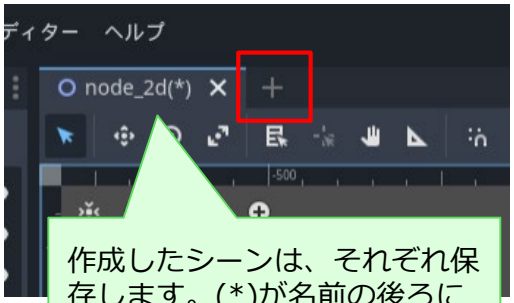
シーンの作成

シーンはノードからできています。新規プロジェクトを開始すると空のシーンが1つありますので、まずはルートとなるノードを作成します。

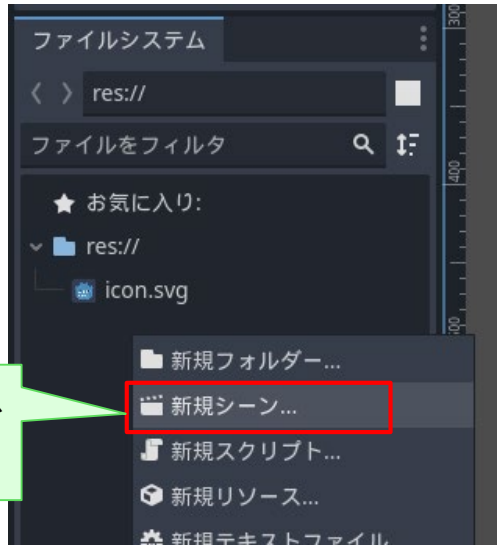
2Dゲームの場合は、2Dシーンを選んでNode2Dを作成し、メインシーンとすることが一般的です。

■シーンの新規作成

追加でシーンを作成する場合は、下図のようにシーン選択タブの「＋」ボタンをクリックして作成します。

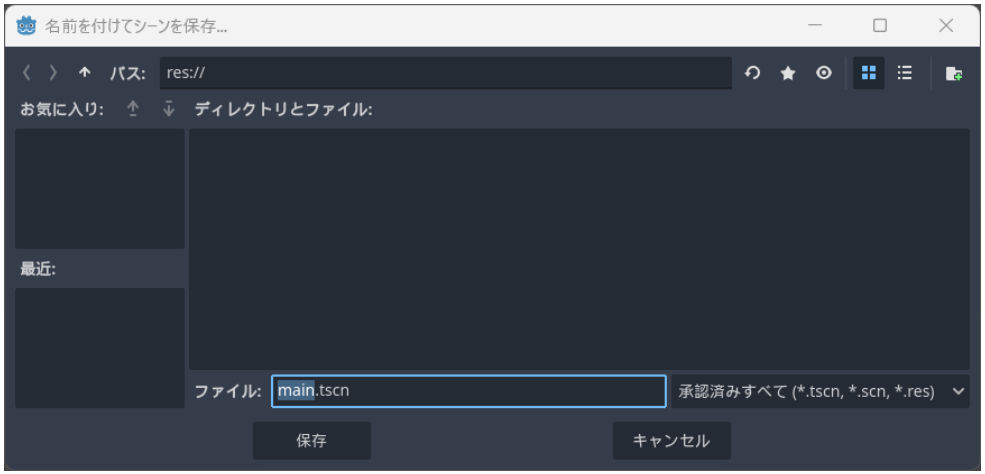


作成したシーンは、それぞれ保存します。(*)が名前の後ろについているシーンは、保存されていないものです。



ファイルシステムドック上で、右クリックメニューから作成することもできます。

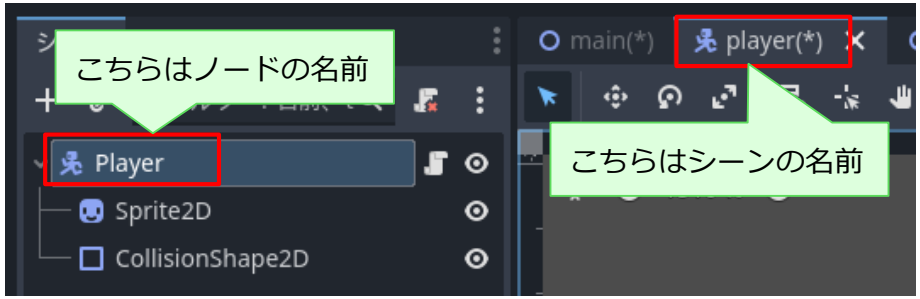
シーンはファイルとして保存します。ファイル拡張子は基本的に「.tscn」を使います。



特に指定がなければシーン名は、全て小文字で単語を_でつなぐ、snake_case（スネークケース）で命名するよう推奨されています。

要注意

シーン選択タブに表示されている名前は、ファイルとして保存したシーン名であることに注意しましょう。

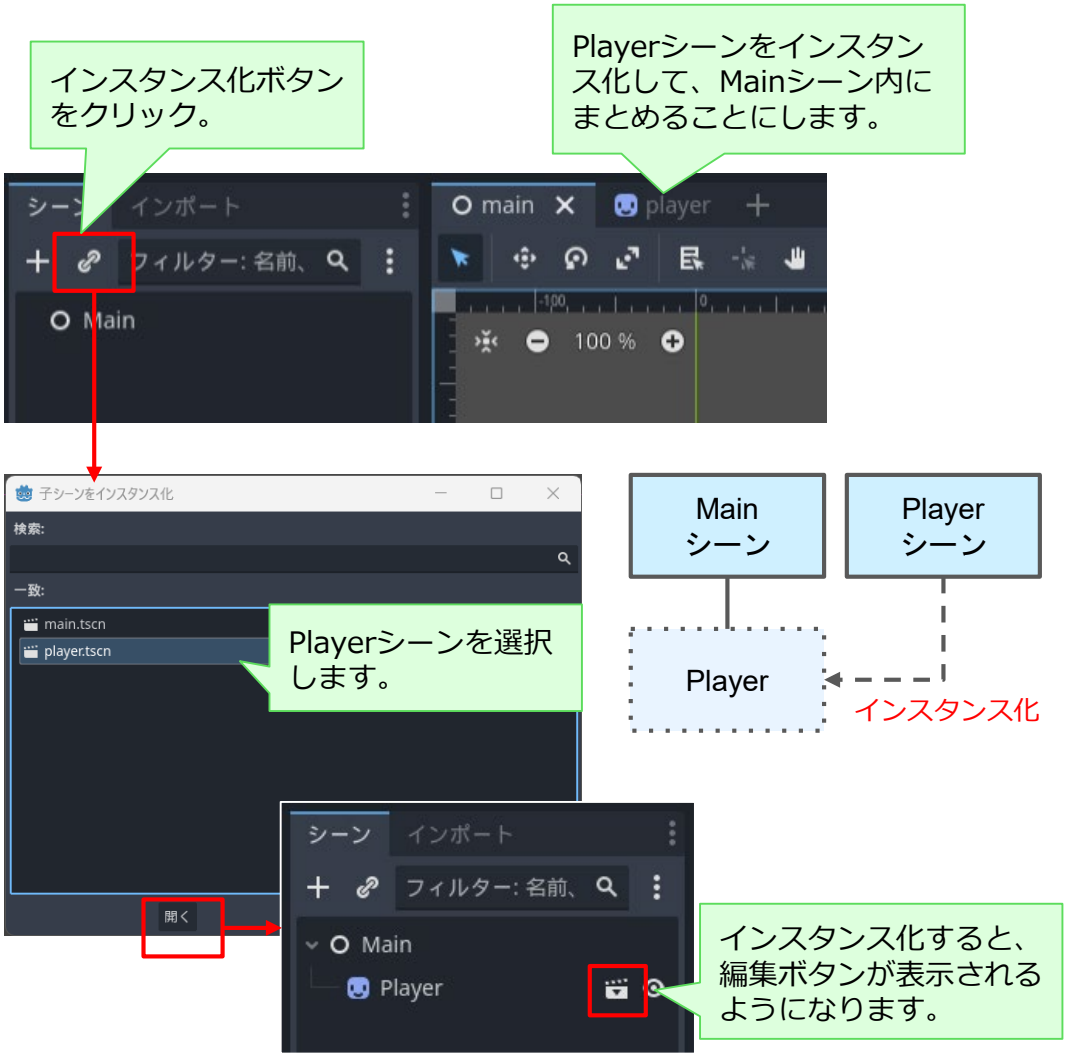


ただ、シーン内のルートノードは、何のシーンなのか分かりやすいような名前に変えておきましょう。

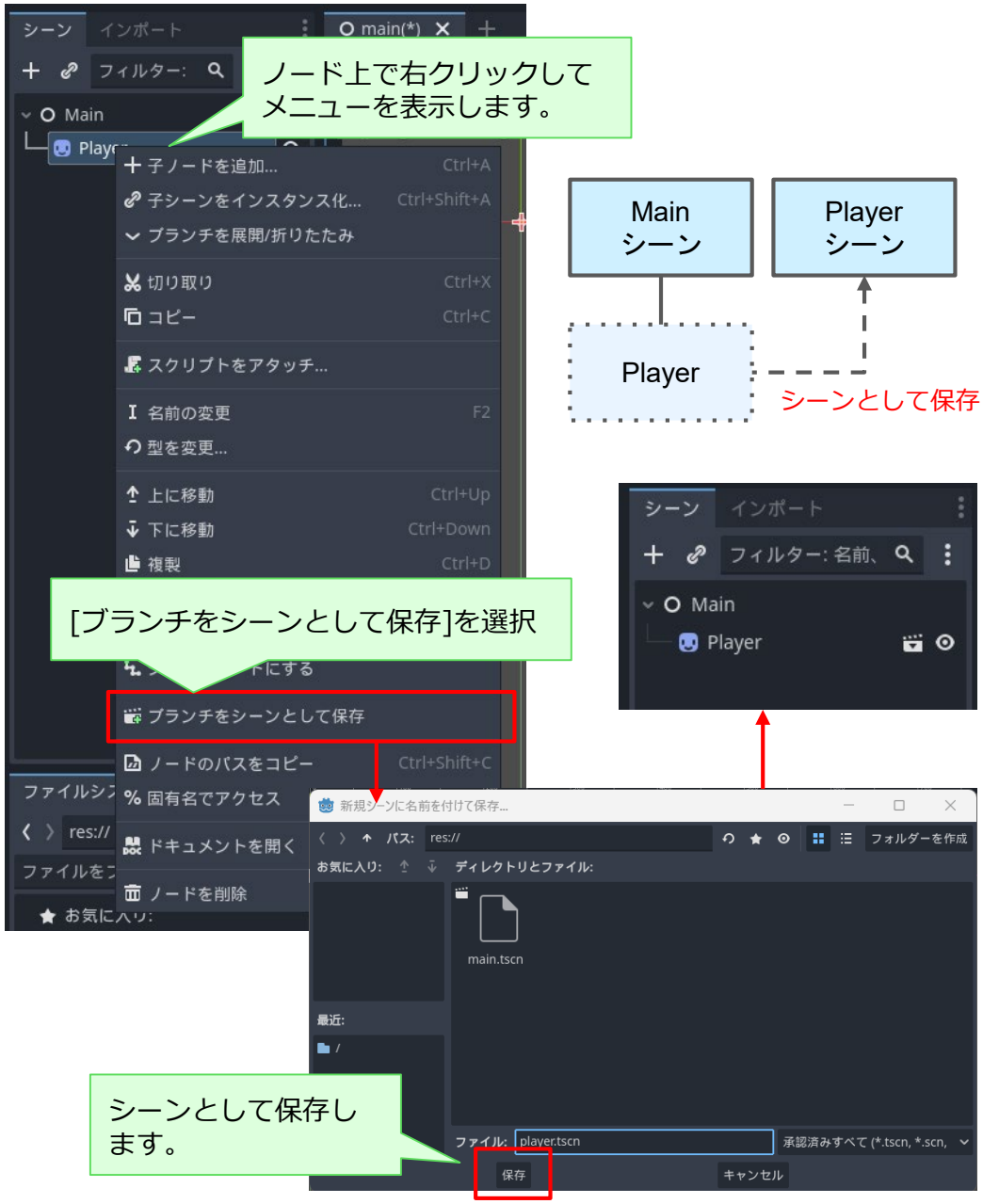
シーンをインスタンス化する【エディタ画面】（１）

ゲーム内で使用するオブジェクトをシーン別に制作したあとに、メインシーン内にインスタンス化してまとめます。

プレイヤーが操作するキャラクターや、ステージの地形、あらかじめ決まった位置にある宝箱など、ゲーム制作中に配置しておいたほうが見た目的にも直感的に分かりやすくなります。



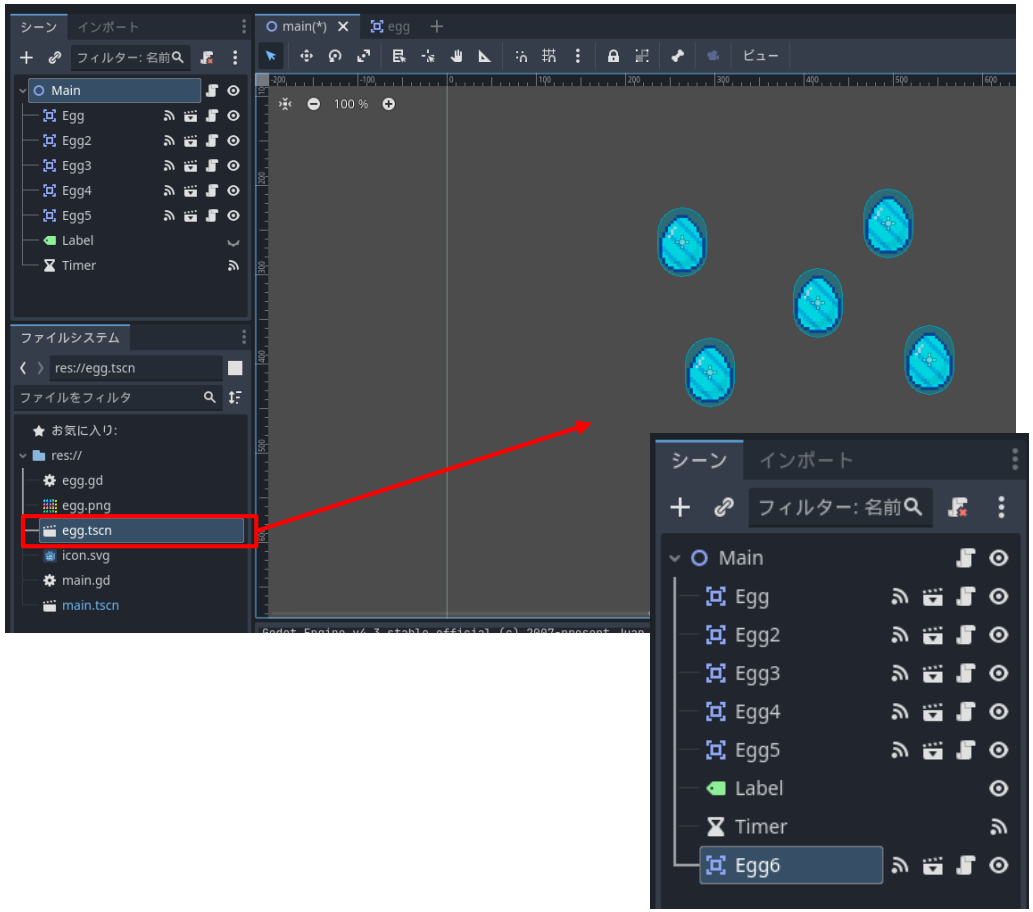
別の方法として、すでに制作済みのノードを別途シーンとして保存しなおし、インスタンス化されたシーンにします。



シーンをインスタンス化する【エディタ画面】（２）

シーンファイルを直接ドラッグ＆ドロップしてインスタンス化する方法もあります。

ビューポートに直接ドラッグ＆ドロップしても、同じようにインスタンス化されます。



インスタンスの名前は、シーンのルートノードの名前がベースになります。

シーンをインスタンス化する【Script】

ゲーム実行時にスクリプトからシーンをインスタンス化して画面に登場させます。
特定のタイミングで登場する敵キャラや、キャラクターから発射される弾丸などは例として分かりやすいでしょう。

あらかじめ preload（プリロード）関数で、インスタンス化するシーンを読み込んでおきます。ここではappleシーンを例にします。

```
var apple_scene = preload("res://apple.tscn")
```

instantiate() 関数でインスタンス化します。

```
var apple_instance = apple_scene.instantiate()
```

インスタンス化したものは、add_child()関数を使って、実際にシーン内の子ノードとして、シーンツリーに加えます。

```
add_child(apple_instance)
```

この例ではマウスでクリックした位置にリンゴ（appleシーン）を生成します。

```
var apple_scene = preload("res://apple.tscn")

func _process(delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
        var mouse_position = get_viewport().get_mouse_position()
        var apple_instance = apple_scene.instantiate()
        apple_instance.position = mouse_position
        add_child(apple_instance)
```

■インスタンスの削除

インスタンスをコード実行中に削除するには、インスタンスに対してqueue_free()メソッドを実行します。

```
apple_instance.queue_free()
```

queue（キュー）は「待ち行列」のような意味で使われています。queue_free（キュー・フリー）関数では対象のインスタンスは、いったん削除予定リストに登録され、適切なタイミングで削除処理をされます。

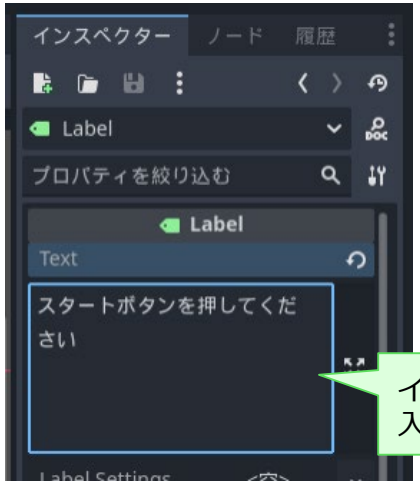
2-2 画面描画

- 文字を表示する
- 線を引く
- 四角を描く
- 表示/非表示
- コンテナ（１）
- コンテナ（２）

文字を表示する

ゲーム画面上に文字を表示するには、Label（ラベル）ノードを使います。

Label



ポイントをドラッグしてサイズを変更できます

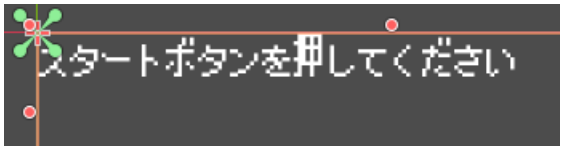
インスペクターでテキストを入力

テキストの文字サイズやフォントを変更できます。

フォントファイルを読み込んでおきます

Fontのところへフォントファイルをドラッグします

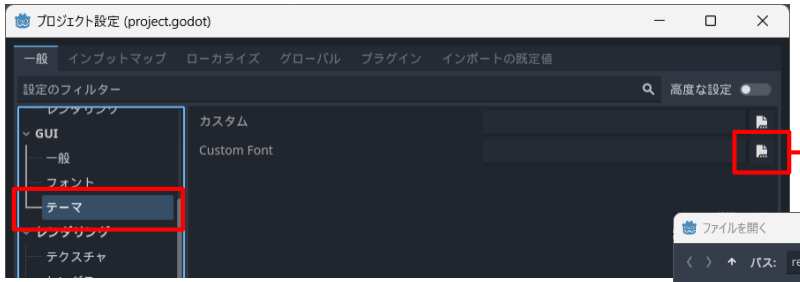
Labelノードにフォントが設定されました

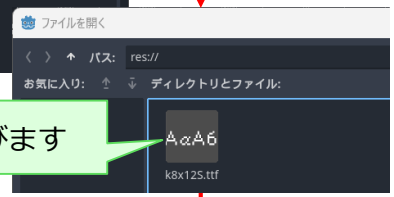


フォントサイズも設定できます

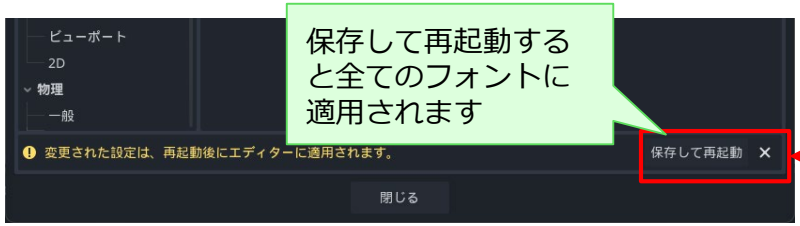
Labelノード単体でフォントを変更する場合はこの方法でできますが、ゲーム全体でフォントを設定する場合は、プロジェクト設定で行います。

[プロジェクト] > [プロジェクト設定...]





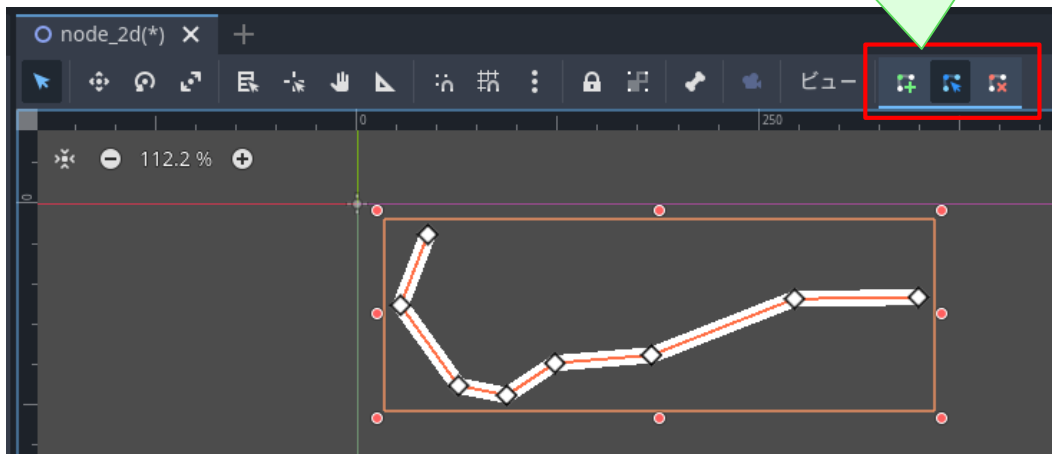
Fontを選びます



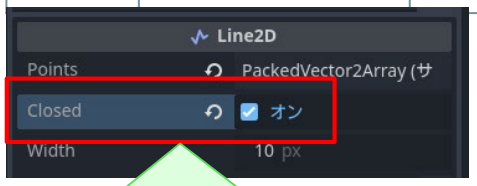
保存して再起動すると全てのフォントに適用されます

線を引く

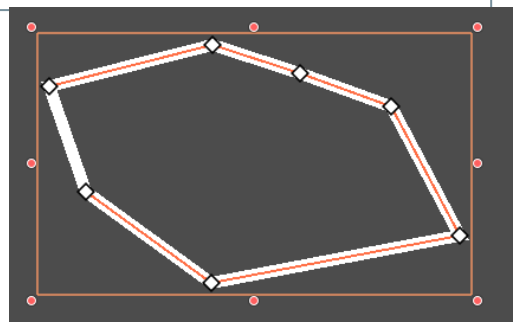
ゲーム画面上に線を引く場合は、Line2D（ライン2D）ノードを使います。単なる画面上の線として使うことも、スクリプトからポイントを制御して、アニメーション効果をつけることもできます。

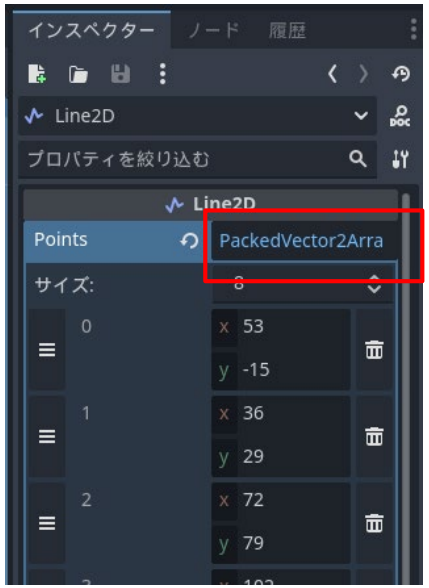


	ポイント追加	ポイントを描画する
	ポイント選択	ポイントを選択、移動する
	ポイント削除	ポイントを削除する



Closedにチェックを入れるとラインを閉じることができます

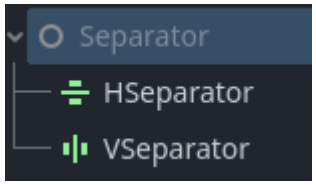


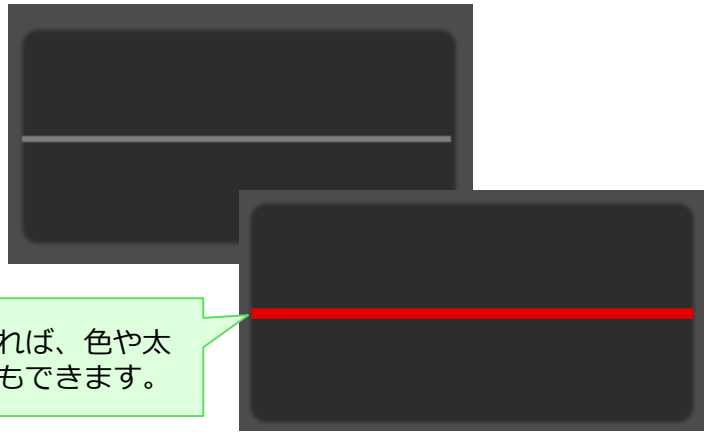


各ポイントはPackedVector2Arrayという配列データ形式で格納されていて、インスペクターで直接座標を指定することもできます。

他にも線の幅や色なども設定できます。

区切り線としてのセパレーターノードを使っても、簡単なラインをつくることができます。





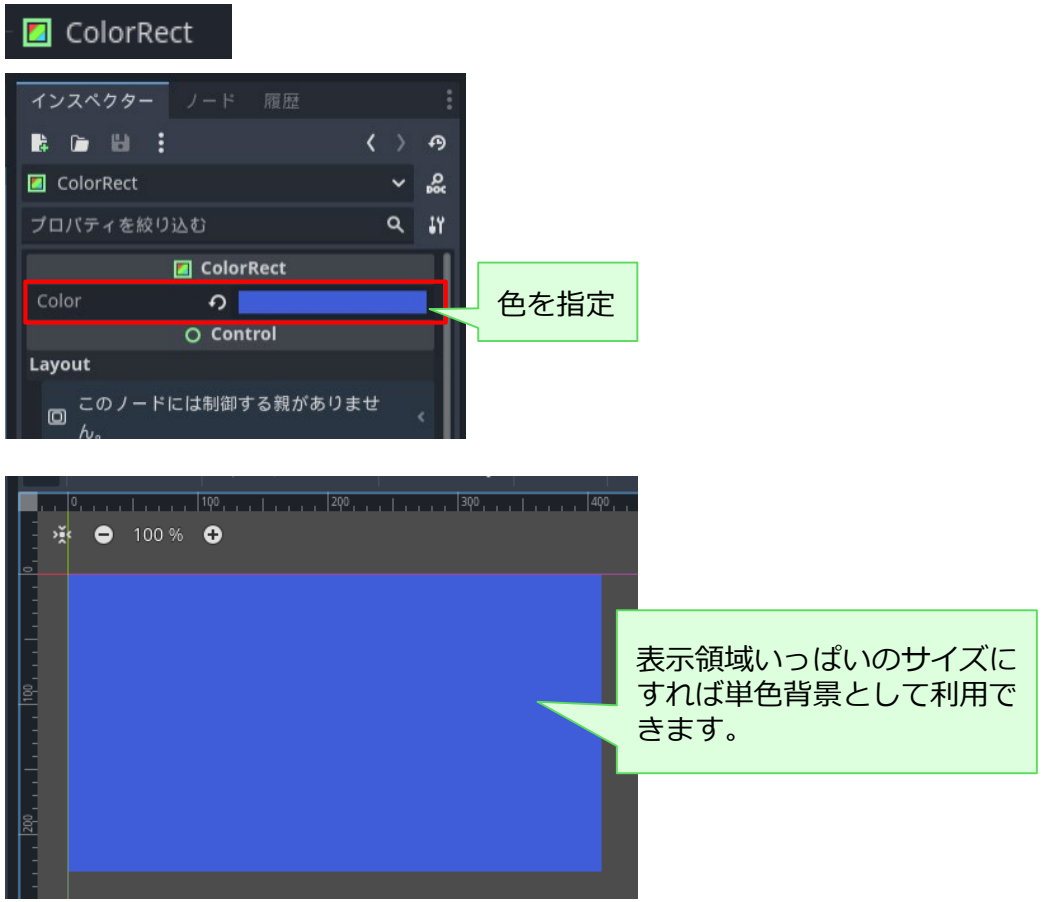
テーマを変更すれば、色や太さを変えることもできます。

四角を描く

シンプルな四角形は開発中にキャラクターの代わりに使ったり、背景のように画面表示の一部と使うこともできます。

■単色四角形

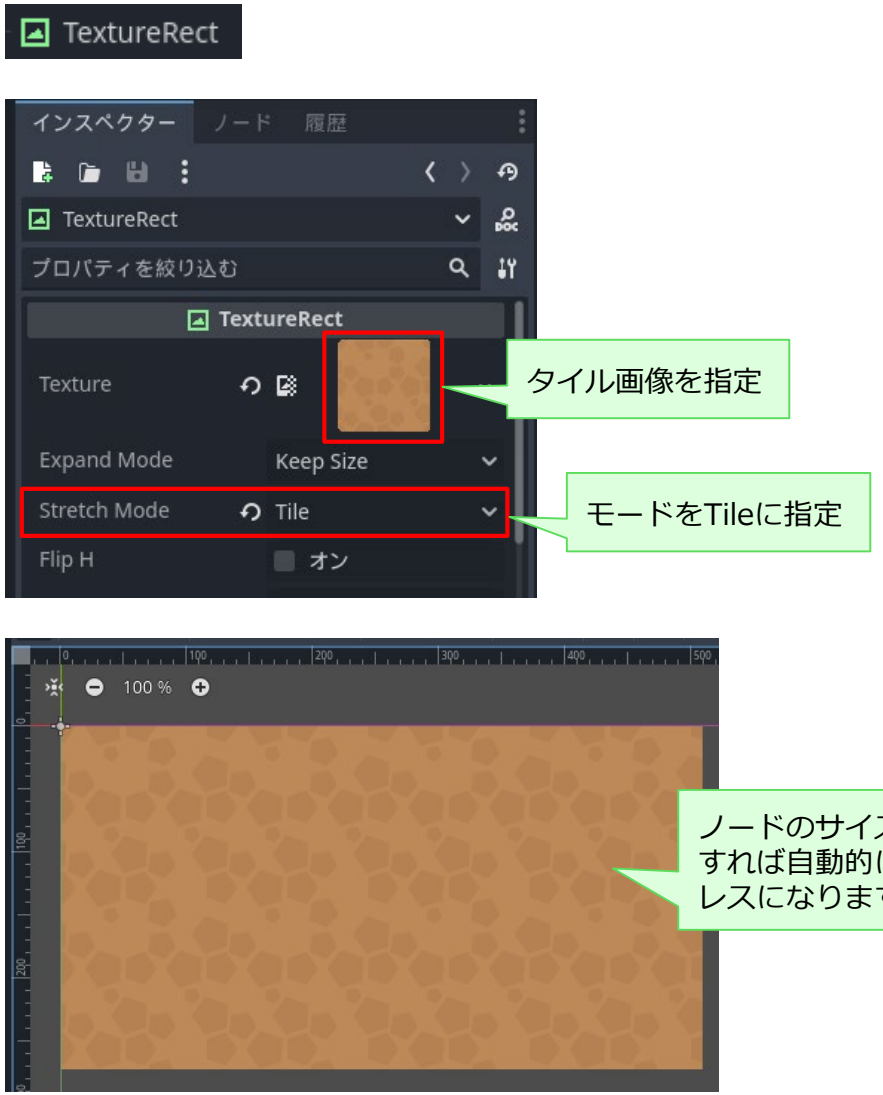
ColorRect（カラー・レクト）ノードを配置すれば、そのまま四角形が描かれるため、サイズや色を変更して使います。



Rectとは四角形を意味する、Rectangle（レクタングル）の略です

■タイル画像を使う

図のようなタイル画像を使って四角形を塗りつぶす場合は、TextureRect（テクスチャ・レクト）ノードを使います。



表示/非表示

画面上に表示されているものを隠したり、表示させたりできます。ゲーム中ではとても良く使われる処理です。

■hide(), show()メソッド

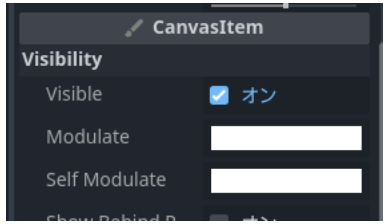
hide()とshow()メソッドは、画面表示されるノード全てで使えるメソッドです。hide()は画面上から隠し、show()は画面上に表示させます。


```
func _ready():
    $Label.hide()

func gameover():
    $Label.text = "GAMEOVER"
    $Label.show()
```

この例では、ゲームが始まるとLabelを画面から隠し、gameover()関数が呼び出されると、テキスト内容を「GAMEOVER」にして、表示を行います。

隠れているノードは画面から見えないだけで、プログラム内から削除されていませんので、アクセスすることができます。

hide(),show()メソッド、visibleプロパティは、Node2DやControlの親クラスである、CanvasItemクラスに定義されているため、画面表示されるノード全てで使うことができます。



■visibleプロパティ

visible（ヴィジブル）も、画面表示されるノード全てで使えるプロパティです。trueで表示、falseで非表示になります。

```
func _ready():
    $Label.visible = false

func gameover():
    $Label.text = "GAMEOVER"
    $Label.visible = true
```

さっきのメソッドを使ったものと動作は同じです。

どちらを使ってもよいですが。メソッドを使った方は処理命令として表示/非表示が分かりやすくなっています。
visibleを使うと、よりプログラムでコントロールしているイメージになります。

このコードは、表示/非表示を入れ替えることができます。

```
$Label.visible = !$Label.visible
```

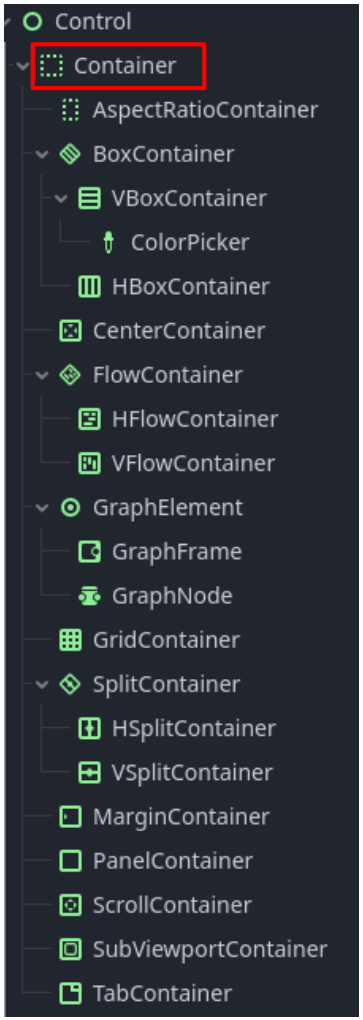
is_visible()メソッドは、表示状態をtrue/falseで返します。

```
var check = $Label.is_visible()
```

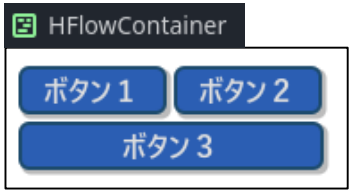
コンテナ（1）

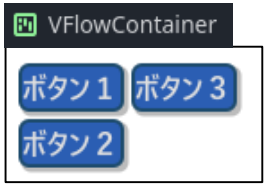
ノードやシーンを画面に配置する際にコンテナを使うと、子要素を整列させることができます。

右図のように、コンテナノードの子要素として、配置したいオブジェクトを構成します。
いくつか使用例を紹介します。

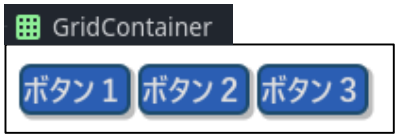


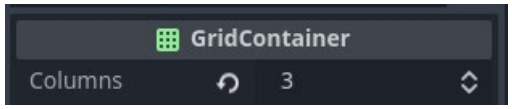
縦横に整列させるコンテナ。
Hは水平（Horizontal：ホリゾンタル）
Vは垂直（Vertical：バーティカル）



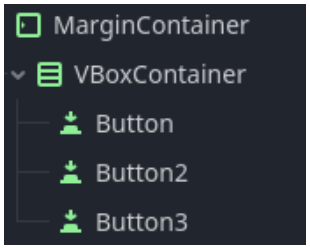


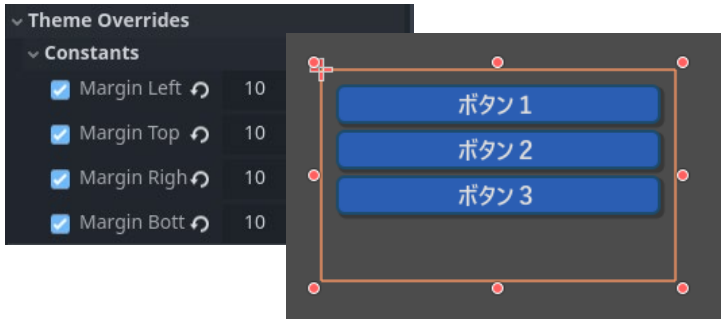
GridContainerは縦列の数を指定して、マス目状に並べることができます。



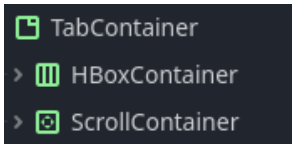


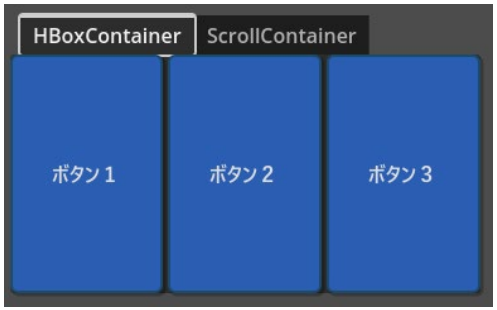
MarginContainerはコンテナ内の上下左右に隙間を設定できます。要素間の間隔をコントロールするのに役立ちます。

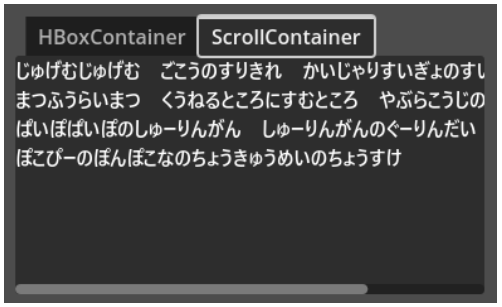




TabContainerは、子要素をタブで切り替えて表示できるようになります。情報量が多い場合や、カテゴリーごとに表示内容を切り替えたい場合に有効です。



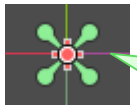




コンテナの子要素はサイズを自由に設定することはできません。

コンテナ（2）

コンテナだけではなく、Controlノード全般の配置を行う際に「アンカー」について知っておくと良いでしょう。アンカーはレイアウトをする際の基準点のようなものです。



Control系ノードを配置したときに表示される緑色のマーカーがアンカーです。

画面上部のアンカーメニューから、アンカーの位置を選択できます。

アンカーの位置を設定すると、親ノードの範囲に合わせて、ノードの配置が変わるのが分かります。

Rect全面を選択すると、親要素のサイズいっぱいにまでノードが拡大されます。

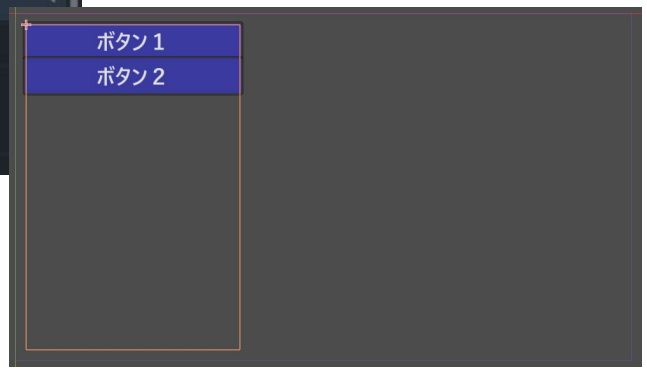
アンカーでレイアウトをする利点は、ゲーム画面サイズの変化に応じて、どこを基準に要素を再レイアウトするのかを決めておけることです。
例えば中央にアンカー設定しておくと、画面サイズが変わっても常に中央に表示されるようになります。

Control系ノードのレイアウトの挙動はやや複雑で、ノードの組み合わせやサイズ、親子関係によって思ったように表示できない場合もあります。いろいろ試してみるのが良いでしょう。

コンテナ内のノードはアンカー設定できず、サイズも自動的に決まってしまいますが、ノードの最小値を設定することで、ある程度サイズをコントロール可能です。

このようなノード構成の際、何も設定しなければ最小限のサイズでVBoxContainerとボタンのサイズが決まります。

ボタンノードの、Custom Minimum Sizeを設定すると、この場合は幅400pxが確保されるため、ボタンの幅をコントロールすることができます。



2-3 入力と移動

- 入力（1）
- 入力（2）
- 入力（3）
- 入力（4）
- キャラクターの移動【positionプロパティ】
- キャラクターの移動【Vector2】

入力（1）

ゲーム制作においては、プレイするユーザーからの入力の処理は必須の要素です。いくつか入力処理の方法があります。

■ _input(event)関数

_input(event)関数を使うと、ユーザーからの様々な入力を取得し、引数の**event**でどのタイプの入力イベントが発生したのか判別できます。

```
func _input(event):
    print(event.as_text())
```

実行結果

```
Mouse motion at position ((461, 323)) with velocity ((0, 0.908083))
Mouse motion at position ((461, 322)) with velocity ((0, -15.24716))
Mouse Wheel Down
Mouse Wheel Down
A
A
Space
```

キーボードやマウス操作するたびに、_input()関数が反応しているのが分かります。

イベントの種類ごとに処理を分けるためには、まずeventがどのInputEventなのかをフィルタリングします。

```
func _input(event):
    if event is InputEventKey:
```

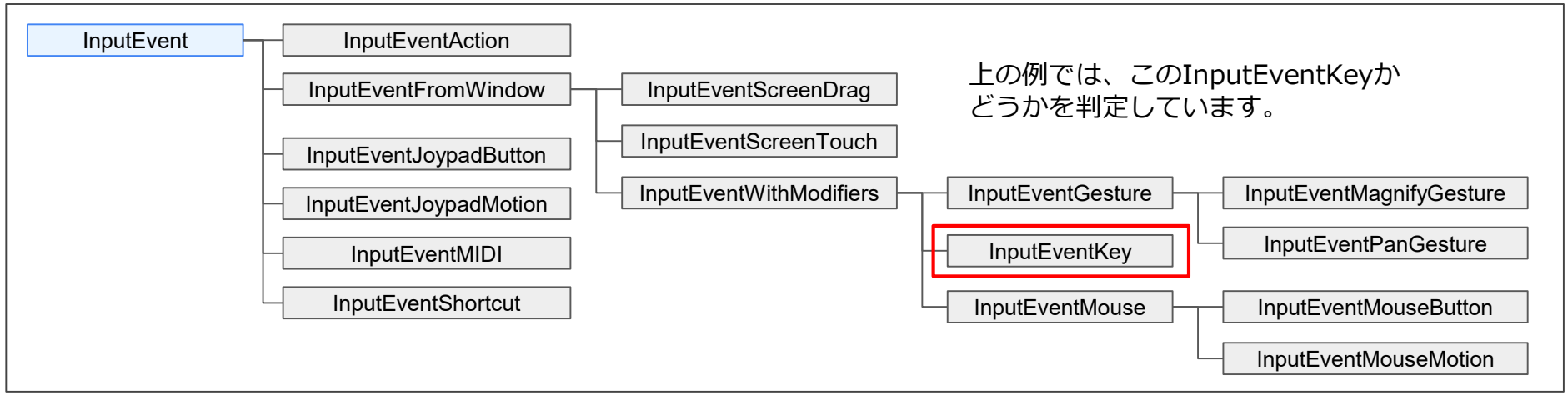
InputEventKey（キーボード入力）かどうかをチェックします。

その後、どのキーが押されたのかを判定します。

```
func _input(event):
    if event is InputEventKey:
        if event.pressed and event.keycode == KEY_SPACE:
            print("スペースキーが押されました")
```

_input()関数は、連続的なキー入力ではなく、一度ずつの入力判定をする場合に使うのが良いでしょう。

引数のevent自体はInputEventオブジェクトで、下図のような種類があり、順に継承していく関係になっています。



上の例では、このInputEventKeyかどうかを判定しています。

■Inputクラス

どのシーンやノードからでもアクセスできる、Inputクラス（シングルトン）を利用して入力処理を行えます。

```
func _process(delta):
    if Input.is_mouse_button_pressed(MOUSE_BUTTON_LEFT):
        print("マウスの左ボタンが押されています")

    if Input.is_key_pressed(KEY_A):
        print("Aキーが押されています")

    if Input.is_joy_button_pressed(0, JOY_BUTTON_A):
        print("Aボタンが押されています")
```

例のように_process()関数の中で定義することで、毎フレームごとに入力をチェックするような使い方に適しています。

マウスボタン入力（抜粋）

左ボタン	MOUSE_BUTTON_LEFT
右ボタン	MOUSE_BUTTON_RIGHT
中央ボタン	MOUSE_BUTTON_MIDDLE
ホイール上スクロール	MOUSE_BUTTON_WHEEL_UP
ホイール下スクロール	MOUSE_BUTTON_WHEEL_DOWN

キー入力（抜粋）

スペース	KEY_SPACE
エンター	KEY_ENTER
A	KEY_A
Z	KEY_Z
←	KEY_LEFT
→	KEY_RIGHT
↑	KEY_UP
↓	KEY_DOWN

TAB	KEY_TAB
SHIFT	KEY_SHIFT
CTRL	KEY_CTRL
ALT	KEY_ALT
0	KEY_0
9	KEY_9
PageUp	KEY_PAGEUP
PageDown	KEY_PAGEDOWN

ジョイパッド入力（抜粋）

SONY	Xbox	Nintendo	
X	A	B	JOY_BUTTON_A
○	B	A	JOY_BUTTON_B
□	X	Y	JOY_BUTTON_X
△	Y	X	JOY_BUTTON_Y
Select	Back	−	JOY_BUTTON_BACK
Options	Menu	＋	JOY_BUTTON_START
L1	LB		JOY_BUTTON_LEFT_SHOULDER
R1	RB		JOY_BUTTON_RIGHT_SHOULDER
十字上			JOY_BUTTON_DPAD_UP
十字下			JOY_BUTTON_DPAD_DOWN
十字左			JOY_BUTTON_DPAD_LEFT
十字右			JOY_BUTTON_DPAD_RIGHT

ジョイパッドのスティック入力は、水平（X軸）垂直（Y軸）の傾け具合を、それぞれ-1～1の数値で取得します。
無操作状態は0が取得されます。

```
var left_stick_x = Input.get_joy_axis(0, JOY_AXIS_LEFT_X)
var left_stick_y = Input.get_joy_axis(0, JOY_AXIS_LEFT_Y)
```

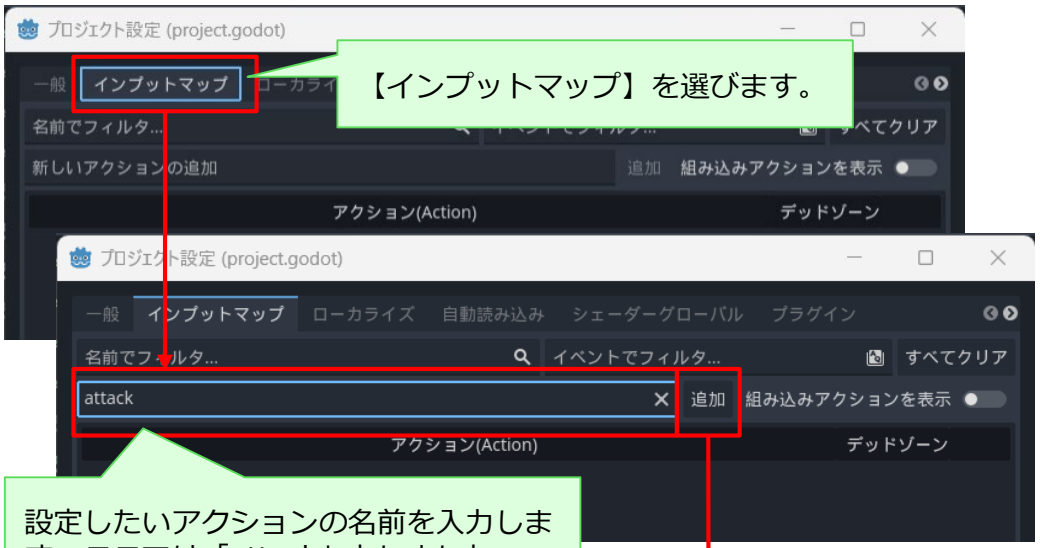
左スティック水平	JOY_AXIS_LEFT_X
左スティック垂直	JOY_AXIS_LEFT_Y
右スティック水平	JOY_AXIS_RIGHT_X
右スティック垂直	JOY_AXIS_RIGHT_Y

例えば水平と垂直の数値の組み合わせによって、キャラクターを斜め移動させたり、傾け度合いで移動速度や攻撃の強さを変えるなど応用できます。

■インプットマップ

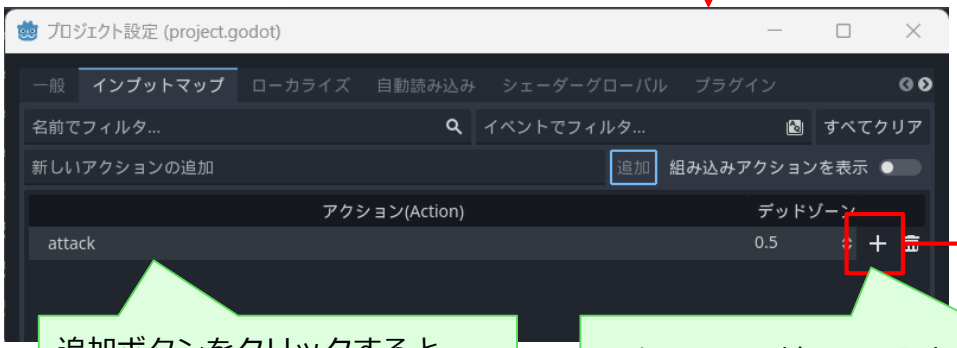
インプットマップ機能を使えばゲームの特定のアクションに対して入力を割り当てることができます。

[プロジェクト] > [プロジェクト設定...]



【インプットマップ】を選びます。

設定したいアクションの名前を入力します。ここでは「attack」としました。



追加ボタンをクリックすると、アクションがリストに追加されます。

アクションに対して、入力イベントを割り当てます。

ここでは、Spaceキーとジョイパッドのボタンを割り当てました。

Inputクラス、_input()関数のいずれかで、設定したアクションを指定して入力を受け取ることができます。

```
func _process(delta):
    if Input.is_action_pressed("attack"):
        print("攻撃ボタンが押されています")
```

```
func _input(event):
    if event.is_action_pressed("attack"):
        print("攻撃ボタンが押されました")
```

それぞれの動作の違いを確認してみよう。

入力（4）

さきほどはインプットマップに独自にアクションを設定してキー入力を割り当てましたが、あらかじめデフォルトのアクションとして設定されているものがあります。

[プロジェクト] > [プロジェクト設定...]

プロジェクト設定で、組み込みアクションの表示を行うと、デフォルトのものが確認できます。

使い方はカスタムアクションのときと同じです。

```
if Input.is_action_pressed("ui_right"):
    print("右矢印キー")
```

```
if event.is_action_pressed("ui_right"):
    print("右矢印キー")
```

■その他の入力

Input.get_vector() は、キーボードやゲームコントローラーの入力をVector2データにとして受け取れるため、キャラクターの移動に便利に使えます。

```
var direction = Input.get_vector("ui_left", "ui_right", "ui_up", "ui_down")
print(direction)
```

実行結果

```
(-0.040981, -0.99916)
(-0.012694, -0.999919)
(0.014647, -0.999893)
(0.014647, -0.999893)
(0.04293, -0.999078)
```

単純にゲームコントローラーのアナログ入力の強さを取得するだけなら、Input.get_action_strength() も使えます。

```
var strength = Input.get_action_strength("ui_right")
print(strength)
```

実行結果

```
0.71098971366882
0.71098971366882
0.68950474262238
0.66997289657593
```

キャラクターの移動【positionプロパティ】

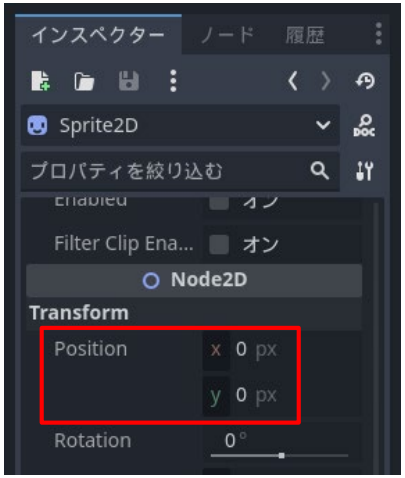
ユーザーが操作するキャラクターや、ゲーム内に登場する敵、弾丸など、画面内を移動させるには、その物体やノードによって移動処理を使い分けます。

■座標で動かす

Scratchでゲーム制作をしたことがあればイメージしやすい方法です。 x 方向、 y 方向の増減を指定することで、画面の中を移動させます。

```
func _process(delta):
    if Input.is_action_pressed("ui_right"):
        position.x += 10
    if Input.is_action_pressed("ui_left"):
        position.x -= 10
```

position（ポジション）は、Node2Dノードのプロパティですので、Node2Dを継承するノードの全てで使え、画面内の位置を指定します。



このコードでは、position.xとx座標成分のみ指定しており、左右の矢印キーが押されていると、1フレームに10px移動します。

1秒間に60フレーム（60FPS）ですので、この場合は最大1秒間に600px移動します。

もう少し高度な処理をする場合、移動量を固定値にするのではなく、一定の値にdeltaを掛けた値にします。

```
func _process(delta):
    if Input.is_action_pressed("ui_right"):
        position.x += 100 * delta
    if Input.is_action_pressed("ui_left"):
        position.x -= 100 * delta
```

delta（デルタ）は前回のフレームから現在のフレームまでの経過時間で、1秒間に60フレームの場合だと、delta = 1/60となり、上記の例だと毎フレーム、100/60ピクセル移動します。そこから、60フレーム＝1秒間には、100px移動します。

このようにdeltaを掛けるようにしておくと、フレームレートが変わったとしても、1秒間に移動する距離を一定にすることができます。

さらに発展させるなら、100のところを変数にしておいて、ゲームの調整をしやすくするといいでしょう。

```
var speed = 100

func _process(delta):
    if Input.is_action_pressed("ui_right"):
        position.x += speed * delta
    if Input.is_action_pressed("ui_left"):
        position.x -= speed * delta
```

■Vector2を使う

positionプロパティを直接増減させるのではなく、ｘとｙの両方の値を同時に扱える、Vector2（ベクター2）を利用します。
移動する値を指定するのではなく、方向だけを1、もしくは-1で指定します。

```
var speed = 100

func _process(delta):
    var movement = Vector2.ZERO
    if Input.is_action_pressed("ui_right"):
        movement.x = 1
    if Input.is_action_pressed("ui_left"):
        movement.x = -1
    position += movement * speed * delta
```

方向の値だけが指定されたVector2にspeedとdeltaを掛けることで、移動量を算出しています。

上記の例ではｘ軸成分だけですが、ｙ軸成分の処理も加えれば、上下左右斜めにもキー入力で移動させることができます。

Vector2.ZEROは、値を(0,0)でリセット処理しています。

ｘとｙの両方の成分を加えた処理のサンプルコードです。

```
var speed = 100

func _process(delta):
    var movement = Vector2.ZERO
    if Input.is_action_pressed("ui_right"):
        movement.x = 1
    if Input.is_action_pressed("ui_left"):
        movement.x = -1
    if Input.is_action_pressed("ui_up"):
        movement.y -= 1
    if Input.is_action_pressed("ui_down"):
        movement.y += 1
    position += movement.normalized() * speed * delta
```

normalized()メソッドを使っているのは。斜め移動のときに上下左右に比べて移動が速くならないようにするためです。

基本テキスト3章も参考にしてください。

基本テキスト 3-1 Vector2