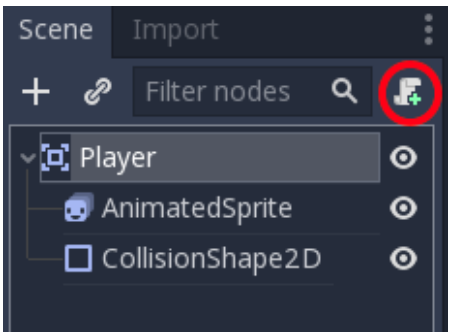
Кодирование плеера

В этом уроке мы добавим передвижение игрока, анимацию и настроим всё для определения столкновений.

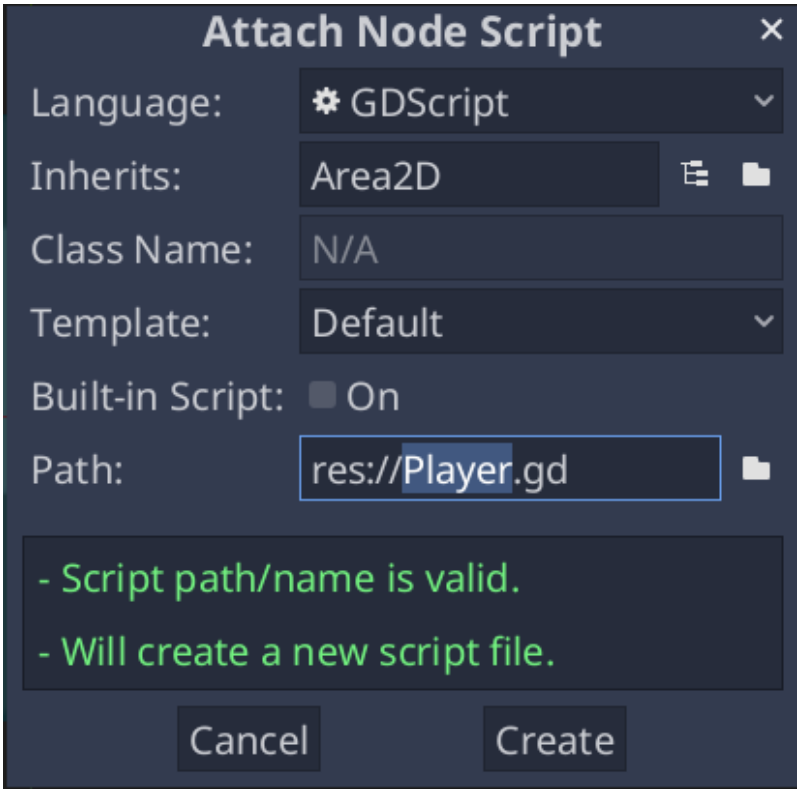
Для этого нам нужно добавить некоторые функции, которые мы не можем получить от встроенного узла, поэтому мы добавим скрипт. Щелкните `Player` узел и нажмите кнопку «Прикрепить скрипт»:



В окне настроек скрипта вы можете оставить все настройки по умолчанию. Просто нажмите "Создать":

Примечание

Если вы создаете скрипт на C# или другом языке, то перед созданием выберете этот язык в выпадающем меню *Язык*.



Примечание

Если вы впервые столкнулись с GDScript, пожалуйста, прочтите Языки скрипта перед продолжением.

Начните с объявления переменных - членов, которые понадобятся этому объекту:

GDScript C# C++

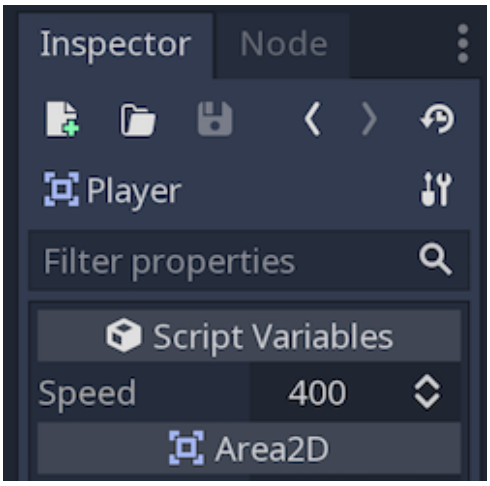
```
extends Area2D

export var speed = 400 # How fast the player will move (pixels/sec).
var screen_size # Size of the game window.
```

Использование ключевого слова `export` у первой переменной `speed` позволяет устанавливать ее значение в Инспекторе. Это может быть полезно, если вы хотите изменять значения точно так же как и встроенные свойства узла. Щелкните на узел `Player` и вы увидите, что свойство появилось в разделе "Script Variables" в Инспекторе. Помните, что если изменить значение здесь, то оно перезапишет значение, установленное в скрипте.

Предупреждение

Если вы используете C#, вам нужно (пере)собирать сборки проекта всякий раз, когда вы хотите увидеть новые экспортируемые переменные или сигналы. Эта сборка может быть запущена вручную путем нажатия на слово "Mono" в нижней части окна редактора, чтобы открыть Mono Panel, а затем на кнопку "Build Project".



Функция `_ready ()` вызывается, когда узел входит в дерево сцены, что является хорошим моментом для определения размера игрового окна:

GDScript C# C++

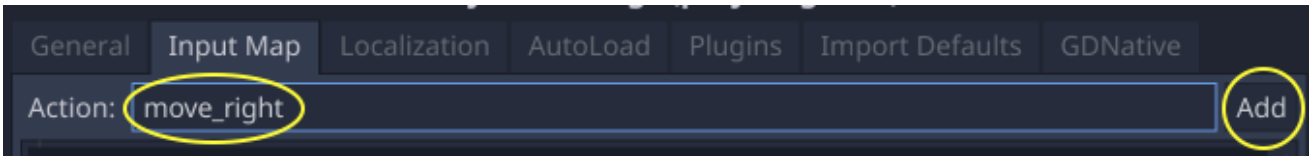
```
func _ready():
    screen_size = get_viewport_rect().size
```

Теперь мы можем использовать функцию `_process()` для определения того, что игрок будет делать. `_process()` вызывается каждый кадр, поэтому мы будем использовать ее для обновления состояния тех элементов нашей игры, которые будут часто изменяться. Для игрока, сделаем следующее:

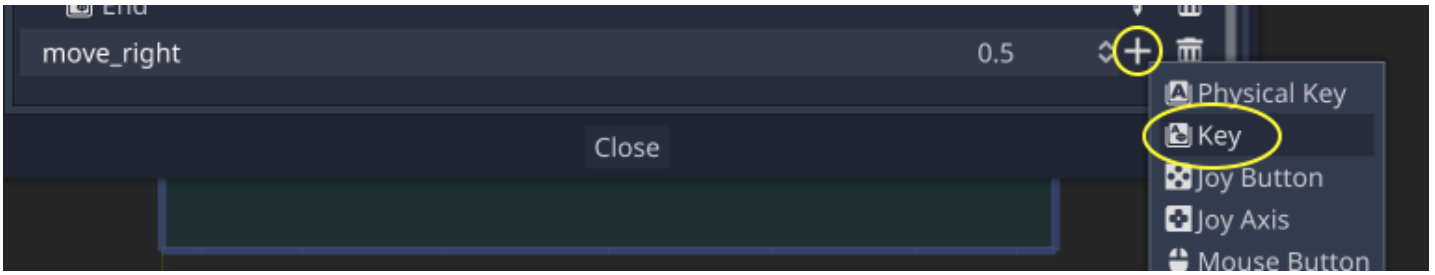
- Проверка ввода.
- Перемещение в заданном направлении.
- Воспроизвести соответствующую анимацию.

Во-первых, нам нужно проверить ввод — игрок нажимает клавишу? Для этой игры у нас есть 4 входа направления для проверки. Действия ввода определяются в настройках проекта в разделе «Карта ввода». Здесь вы можете определить пользовательские события и назначить им различные клавиши, события мыши или другие входные данные. В этой игре мы сопоставим клавиши со стрелками с четырьмя направлениями.

Нажмите на *Проект -> Настройки проекта*, чтобы открыть окно настроек проекта, и нажмите вверху на вкладку *Input Map*. Введите "move_right" в верхней панели и нажмите на кнопку "Добавить", чтобы добавить действие `move_right`.



Нам необходимо присвоить клавишу этому действию. Нажмите справа на иконку "+", затем нажмите на параметр "Клавиша" в выпадающем меню. В диалоговом окне Вас попросят ввести желаемую клавишу. Зажмите правую стрелку на Вашей клавиатуре и нажмите "Ок".



Повторите эти шаги, чтобы добавить ещё три маппинга:

1. `move_left` соответствует стрелке влево.
2. `move_up` соответствует стрелке вверх.
3. А `move_down` соответствует стрелке вниз.

Вкладка Список действий должна выглядеть так:

Нажмите кнопку "Закрыть" чтобы закрыть настройки проекта.

Примечание

Мы присвоили только одну клавишу каждому действию, но Вы можете присвоить тем же самым действиям несколько клавиш, кнопки джойстика или мыши.

Вы можете определить, нажата ли клавиша с помощью функции `Input.is_action_pressed()`, которая возвращает `true`, если клавиша нажата, или `false`, если нет.

GDScript C# C++

```
func _process(delta):
    var velocity = Vector2.ZERO # The player's movement vector.
    if Input.is_action_pressed("move_right"):
        velocity.x += 1
    if Input.is_action_pressed("move_left"):
        velocity.x -= 1
    if Input.is_action_pressed("move_down"):
        velocity.y += 1
    if Input.is_action_pressed("move_up"):
        velocity.y -= 1

    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite.play()
    else:
        $AnimatedSprite.stop()
```

Начнём с того, что установим значение `velocity` в `(0, 0)` - по умолчанию игрок двигаться не должен. Затем, мы проверяем каждый ввод и добавляем/вычитаем значение из `velocity`, чтобы получить общее направление. Например, если вы одновременно удерживаете `right` и `down`, полученный вектор `velocity` будет `(1, 1)`. В этом случае, поскольку мы добавляем горизонтальное и вертикальное движение, игрок будет двигаться *быстрее*, чем если бы он перемещался только по горизонтали.

Можно избежать этого, если мы *нормализуем* скорость, что означает, что мы устанавливаем ее *длину* на `1` и умножаем на желаемую скорость. Это означает отсутствие более быстрого диагонального движения.

Совет

Если вы никогда раньше не использовали векторную математику или нуждаетесь в повторении, вы можете увидеть объяснение использования вектора в Godot по ссылке

Векторная математика. Ее полезно знать, но она не понадобится для остальной части этого урока.

Мы также проверяем, движется ли игрок, чтобы мы могли вызвать `play()` или `stop()` на AnimatedSprite.

Совет

`$` является сокращением для `get_node()`. Таким образом, в приведенном выше коде `$AnimatedSprite.play()` это то же самое, что и `get_node("AnimatedSprite").play()`.

В GDScript `$` возвращает узел по относительному пути от текущего узла или возвращает, `null` если узел не найден. Поскольку AnimatedSprite является дочерним элементом текущего узла, мы можем использовать `$AnimatedSprite`.

Теперь, когда у нас есть направление движения, мы можем обновить позицию игрока. Мы также можем использовать `clamp()`, чтобы он не покинул экран. *Clamping* означает ограничение движения диапазоном. Добавьте следующее в конец функции `_process` (убедитесь, что он не имеет отступа под *else*):

GDScript C# C++

```
position += velocity * delta
position.x = clamp(position.x, 0, screen_size.x)
position.y = clamp(position.y, 0, screen_size.y)
```

Совет

Параметр "delta" в функции "_process()" означает *длительность кадра*, то есть время, потраченное на завершение обработки предыдущего кадра. Благодаря использованию этого значения скорость движения будет постоянной даже при изменении частоты кадров.

Нажмите "Запустить сцену" (`F6`, `Cmd + R` on macOS) и удостоверьтесь, что вы можете перемещать игрока по экрану во всех направлениях.

Предупреждение

Если вы получаете ошибку в панели "Отладчик", которая говорит

`Попытка вызова функции 'play' в основании 'null instance' на нулевом экземпляре`

это, скорее всего, означает, что вы ввели название узла AnimatedSprite неверно. Имена узлов чувствительны к регистру, а `$NodeName` должен совпадать с именем, которое вы видите в дереве сцены.

Выбор анимации

Теперь, когда игрок может двигаться, нам нужно изменять анимацию AnimatedSprite в зависимости от направления движения. У нас есть анимация "walk", которая показывает игрока, идущего направо. Эту анимацию следует перевернуть горизонтально, используя свойство `flip_h` для движения влево. У нас также есть анимация "up", которую нужно перевернуть вертикально с помощью `flip_v` для движения вниз. Поместим этот код в конец функции `_process ()`:

GDScript C# C++

```
if velocity.x != 0:
    $AnimatedSprite.animation = "walk"
    $AnimatedSprite.flip_v = false
    # See the note below about boolean assignment.
    $AnimatedSprite.flip_h = velocity.x < 0
elif velocity.y != 0:
    $AnimatedSprite.animation = "up"
    $AnimatedSprite.flip_v = velocity.y > 0
```

Примечание

Логические присваивания в коде выше являются общим сокращением для программистов. Поскольку мы проводим проверку сравнения (логическую, булеву), а также *присваиваем* булево значение, мы можем делать и то, и другое одновременно. Рассмотрим этот код в сравнении с однострочным логическим присваиванием выше:

GDScript C#

```
if velocity.x < 0:
    $AnimatedSprite.flip_h = true
else:
    $AnimatedSprite.flip_h = false
```

Воспроизведите сцену еще раз и проверьте правильность анимации в каждом из направлений.

Совет

Общей ошибкой является неправильное именование анимаций. Имена анимаций в панели SpriteFrames должны совпадать с именами анимаций в вашем коде. Если вы назвали анимацию "Walk", вы должны также использовать заглавную букву "W" в коде.

Если вы уверены, что движение работает правильно, добавьте эту строку в `_ready()`, чтобы игрок был скрыт при запуске игры:

GDScript C# C++

```
hide()
```

Подготовка к столкновениям

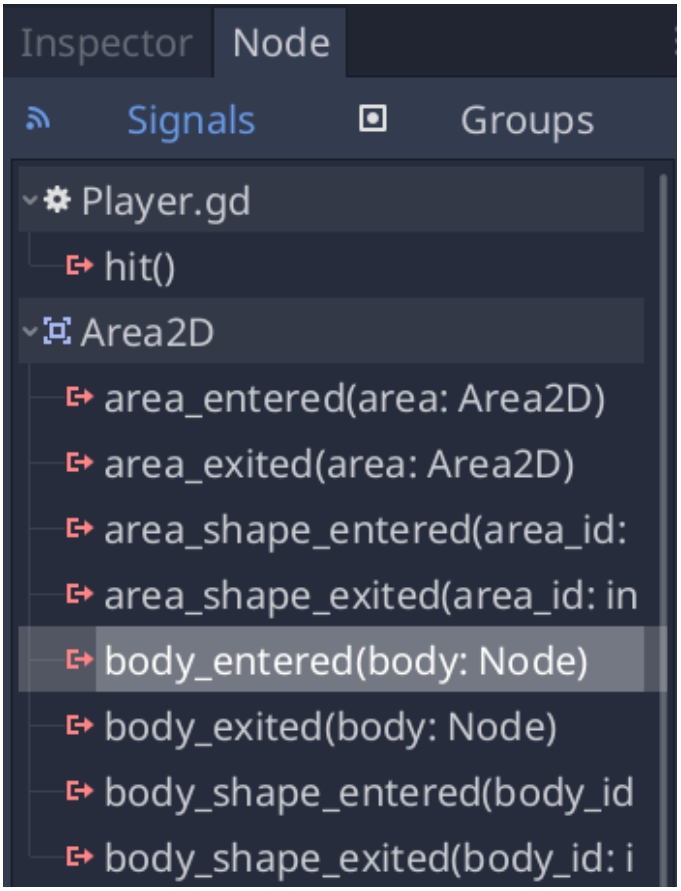
Мы хотим, чтобы `Player` обнаруживал столкновение с врагом, но мы еще не сделали никаких врагов! Это нормально, потому что мы будем использовать такой функционал Godot, как *сигнал*.

Добавьте следующее в верх скрипта после `extends Area2d`:

GDScript C# C++

```
signal hit
```

Это определяет пользовательский сигнал под названием "hit" ("удар"), который наш игрок будет излучать (отправлять), когда он сталкивается с противником. Мы будем использовать `Area2D` для обнаружения столкновения. Выберите узел `Player` ("Игрок") и щелкните по вкладке "Узел" (Node) рядом с вкладкой "Инспектор" (Inspector), чтобы просмотреть список сигналов, которые игрок может посылать:



Обратите внимание, что наш пользовательский сигнал "hit" там также есть! Поскольку наши противники будут узлами `RigidBody2D`, нам нужен сигнал `body_entered(body: Node)`. Он будет отправляться при контакте тела (body) с игроком. Нажмите "Присоединить..." - появится окно "Подключить сигнал к методу". Нам не нужно изменять какие-либо из этих настроек, поэтому еще раз нажмите "Присоединить". Godot автоматически создаст функцию в скрипте вашего игрока.

```
42 func _on_Player_body_entered(body):
43     pass # Replace with function body.
```

Обратите внимание на зеленый значок, указывающий на то, что сигнал подключен к этой функции. Добавьте этот код в функцию:

GDScript C# C++

```gdscript
func _on_Player_body_entered(body):
    hide() # Player disappears after being hit.
    emit_signal("hit")
    # Must be deferred as we can't change physics properties on a physics callback.
    $CollisionShape2D.set_deferred("disabled", true)
```

Каждый раз, когда противник ударяет игрока, будет посылаться сигнал. Нам нужно отключить столкновение игрока, чтобы не вызывать сигнал `hit` более одного раза.

Примечание

Отключение формы области столкновения может привести к ошибке, если это происходит во время обработки движком столкновений. Использование `set_deferred()` говорит Godot ждать отключения этой формы, пока это не будет безопасно.

Последняя деталь - добавить функцию, которую мы можем вызвать для перезагрузки игрока при запуске новой игры.

GDScript C# C++

```
func start(pos):
    position = pos
    show()
    $CollisionShape2D.disabled = false
```

Теперь, когда игрок работает, мы займёмся врагом в следующем уроке.