

---

# Сценарии

## Введение

Много было сказано об инструментах, которые позволяют пользователям создавать видеоигры без программирования. Многие независимые разработчики мечтали создавать игры, не умея программировать. Эта потребность существует уже давно, даже внутри компаний, где геймдизайнеры хотят иметь больший контроль над ходом игры.

Многие продукты поставляются с обещаниями среды без программирования, но результат часто оказывается неполным, слишком сложным или неэффективным по сравнению с традиционным кодом. В результате программирование останется надолго. На самом деле общее направление в игровых движках заключалось в добавлении инструментов, которые пытаются уменьшить количество кода, необходимого для написания конкретных задач, чтобы ускорить разработку.

В этом смысле Годо принял несколько полезных дизайнерских решений для достижения этой цели. Первая и самая важная — это система сцен. Цель этого сначала не очевидна, но потом хорошо работает. То есть, чтобы освободить программистов от ответственности за архитектуру кода.

При разработке игр с использованием системы сцен весь проект разбивается на *дополняющие друг друга* сцены (а не на отдельные). Сцены дополняют друг друга, а не являются отдельными. Позже будет много примеров по этому поводу, но очень важно помнить об этом.

Для тех, кто имеет хороший опыт программирования, это означает, что шаблон проектирования отличается от MVC. Godot обещает эффективность за счет отказа от привычек MVC, которые заменяются сценами *в качестве дополнительного шаблона*.

Godot также использует шаблон [расширения](#) для сценариев, что означает, что сценарии расширяются из всех доступных классов движка.

## GDScript

[GDScript](#) — это язык сценариев с динамической типизацией, подходящий для использования в Godot. Он был разработан со следующими целями:

- Во-первых, и самое главное, сделать его простым, знакомым и максимально легким для изучения.

- Делаем код читабельным и безопасным для ошибок. Синтаксис в основном заимствован из Python.

Программистам обычно требуется несколько дней, чтобы изучить его, и в течение двух недель они чувствуют себя комфортно.

Как и в большинстве языков с динамической типизацией, более высокая производительность (код легче изучать, быстрее писать, нет компиляции и т. д.) уравнивается потерей производительности. Но наиболее важный код написан на C++ уже в движке (векторные операции, физика, математика, индексирование и т. д.), что обеспечивает более чем достаточную производительность для большинства типов игр.

В любом случае, если требуется более высокая производительность, критические разделы можно переписать на C++ и сделать прозрачными для сценария. Это позволяет заменить класс GDScript классом C++ без изменения остальной части игры.

## Сценарий сцены

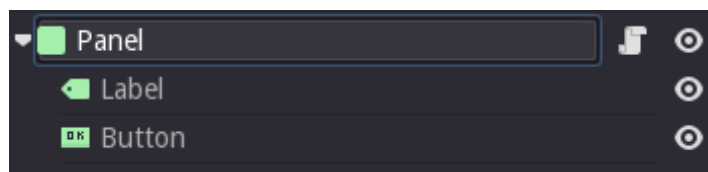
Прежде чем продолжить, обязательно прочитайте справку по [GDScript](#). Это простой язык, а справочная информация короткая, для ознакомления с концепциями потребуется не более нескольких минут.

## Настройка сцены

Этот урок начнется с написания сценария простой сцены с графическим интерфейсом. Используйте диалоговое окно добавления узла, чтобы создать следующую иерархию со следующими узлами:

- Панель
  - Этикетка
  - Кнопка

В дереве сцены это должно выглядеть так:



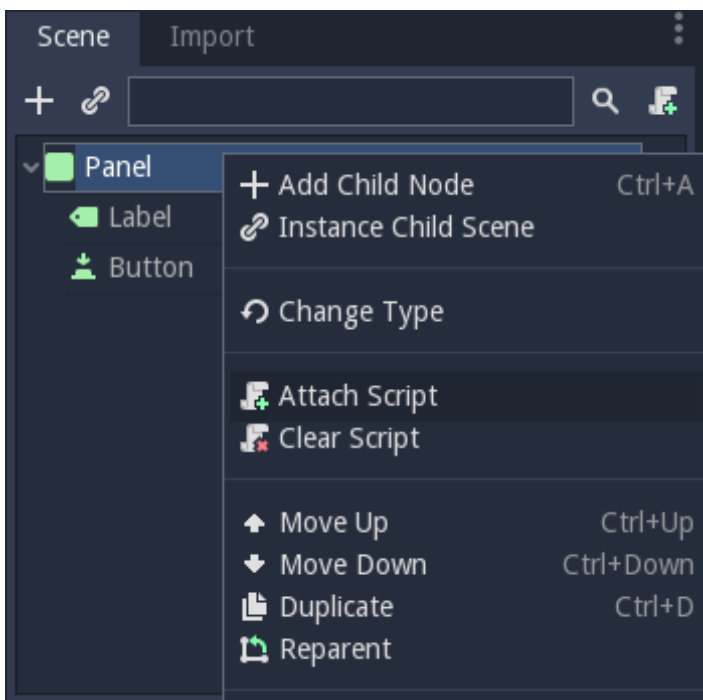
Используйте 2D-редактор, чтобы расположить и изменить размер кнопки и метки, чтобы они выглядели, как на изображении ниже. Вы можете установить текст в панели Inspector.



Наконец, сохраните сцену, подходящим названием может быть «sayhello.scn».

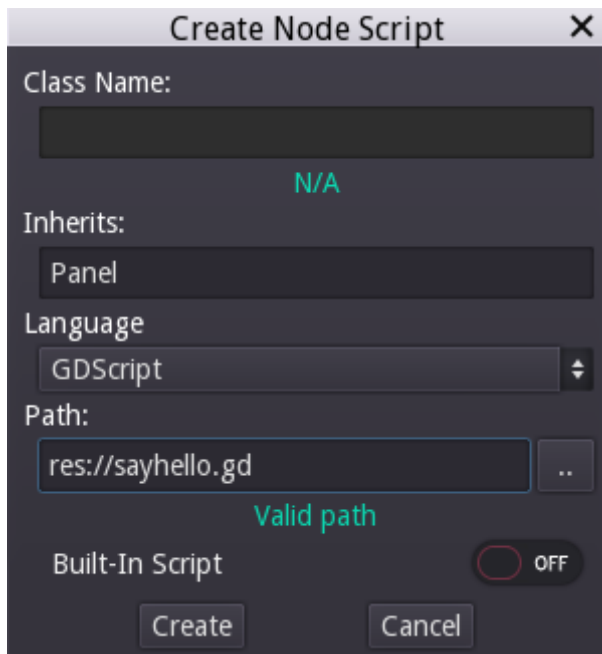
## Добавление скрипта

Щелкните правой кнопкой мыши узел панели, затем выберите «Добавить скрипт» в контекстном меню:

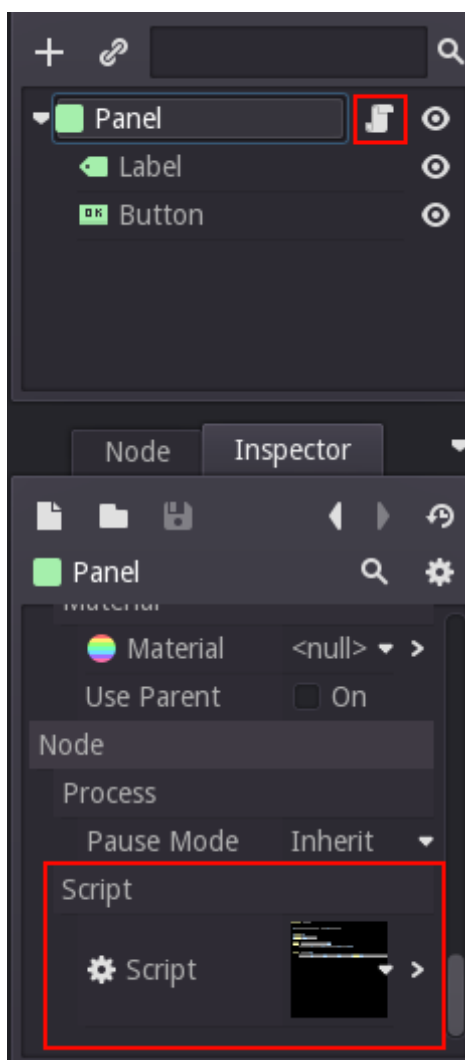


Появится диалоговое окно создания скрипта. В этом диалоговом окне можно выбрать язык, имя класса и т. д. GDScript не использует имена классов в файлах сценариев, поэтому это поле недоступно для редактирования. Скрипт должен наследоваться от «Panel» (поскольку он предназначен для расширения узла типа «Panel», он заполняется автоматически).

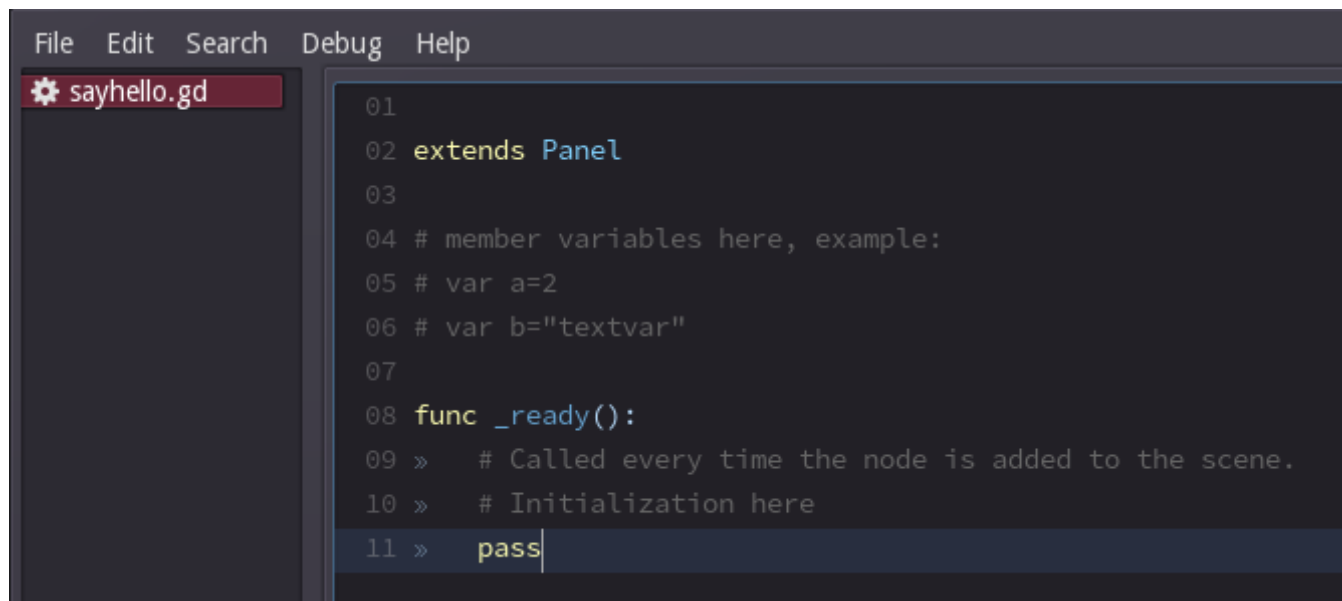
Введите путь к скрипту и затем выберите «Создать»:



Как только это будет сделано, скрипт будет создан и добавлен к узлу. Вы можете увидеть это как в виде дополнительной иконки в узле, так и в свойстве скрипта:



Чтобы отредактировать сценарий, выберите любую из выделенных кнопок. Это приведет вас к редактору скриптов, где по умолчанию будет включен существующий шаблон:

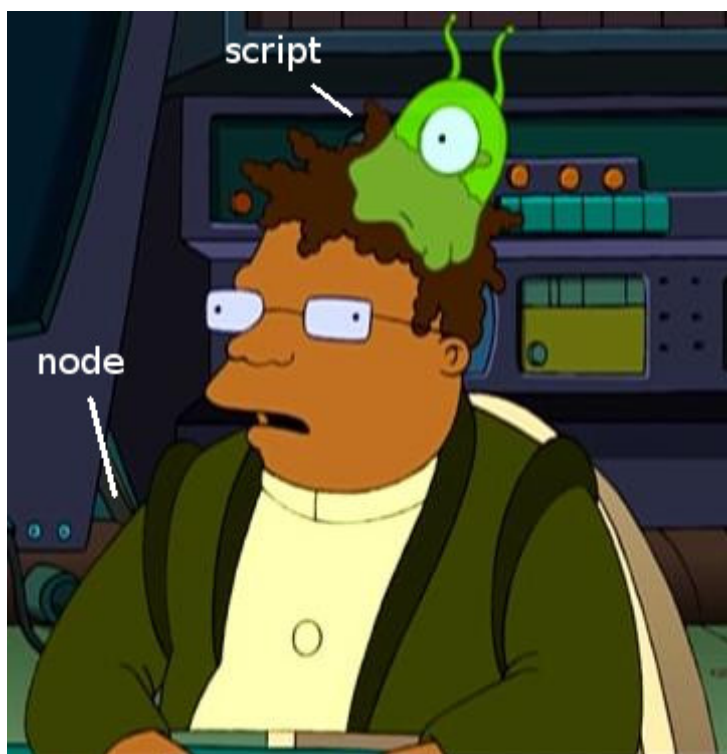


```
01
02 extends Panel
03
04 # member variables here, example:
05 # var a=2
06 # var b="textvar"
07
08 func _ready():
09 > # Called every time the node is added to the scene.
10 > # Initialization here
11 > pass
```

Там не так много. Функция «\_ready()» вызывается, когда узел (и все его дочерние элементы) вошли в активную сцену. (Помните, это не конструктор, это конструктор «\_init()»).

## Роль сценария

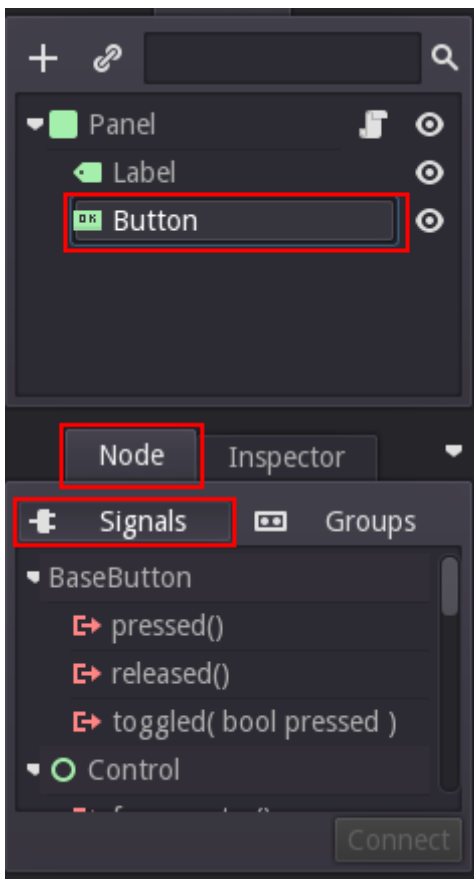
Сценарий добавляет поведение узлу. Он используется для управления функциями узла, а также другими узлами (дочерними, родительскими, братьями и сестрами и т. д.). Локальной областью действия сценария является узел (как и при обычном наследовании), а виртуальные функции узла захватываются сценарием.



## Обработка сигнала

Сигналы используются в основном в узлах с графическим интерфейсом (хотя они есть и в других узлах). Сигналы «излучаются», когда происходит какое-то определенное действие, и могут быть связаны с любой функцией любого экземпляра скрипта. На этом этапе «нажатый» сигнал кнопки будет подключен к пользовательской функции.

В редакторе есть интерфейс для подключения сигналов к вашим скриптам. Вы можете получить к нему доступ, выбрав узел в дереве сцены, а затем выбрав вкладку «Узел». Убедитесь, что вы выбрали «Сигналы».



В любом случае, в этот момент понятно, что нас интересует «нажатый» сигнал. Вместо использования визуального интерфейса мы выберем код соединения.

Для этого существует функция, которая, вероятно, чаще всего используется программистами Godot, а именно [Node.get\\_node\(\)](#) . Эта функция использует пути для выборки узлов в текущем дереве или в любом месте сцены относительно узла, содержащего скрипт.

Чтобы получить кнопку, необходимо использовать следующее:

```
get_node ( "Кнопка" )
```

Далее будет добавлен обратный вызов, который изменит текст метки при нажатии кнопки:

```
func _on_button_pressed ():  
    get_node ( "Ярлык" ) . set_text ( "ПРИВЕТ!" )
```

Наконец, сигнал «нажата» кнопка будет связан с этим обратным вызовом в `_ready()` с помощью [Object.connect\(\)](#) .

```
func _ready ():  
    get_node ( "Кнопка" ) . подключить ( «нажато» , «я» , «_на_кнопке_нажато» )
```

Окончательный скрипт должен выглядеть так:

```
расширяет панель

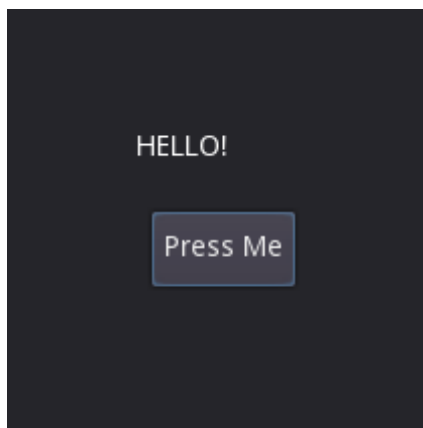
# здесь переменные-члены, например:

# var a=2
# var b="textvar"

func _on_button_pressed():
    get_node( "Ярлык" ). set_text( "ПРИВЕТ!" )

func _ready():
    get_node( "Кнопка" ). подключить ( «нажато» , «я» , «_на_кнопке_нажато» )
```

Запуск сцены должен иметь ожидаемый результат при нажатии на кнопку:



**Примечание.** Поскольку в этом руководстве часто встречается недоразумение, давайте еще раз уточним, что `get_node(path)` работает, возвращая *непосредственных* дочерних элементов узла, управляемого сценарием (в данном случае *Panel*), поэтому *Button* должен быть дочерним элементом *Panel* для приведенный выше код работает. Чтобы дать этому пояснению больше контекста, если бы *Button* был дочерним элементом *Label*, код для его получения был бы таким:

```
# не в этом случае
# а на всякий случай
get_node( "Ярлык/Кнопка" )
```

Также помните, что на узлы ссылаются по имени, а не по типу.