Лабиринт

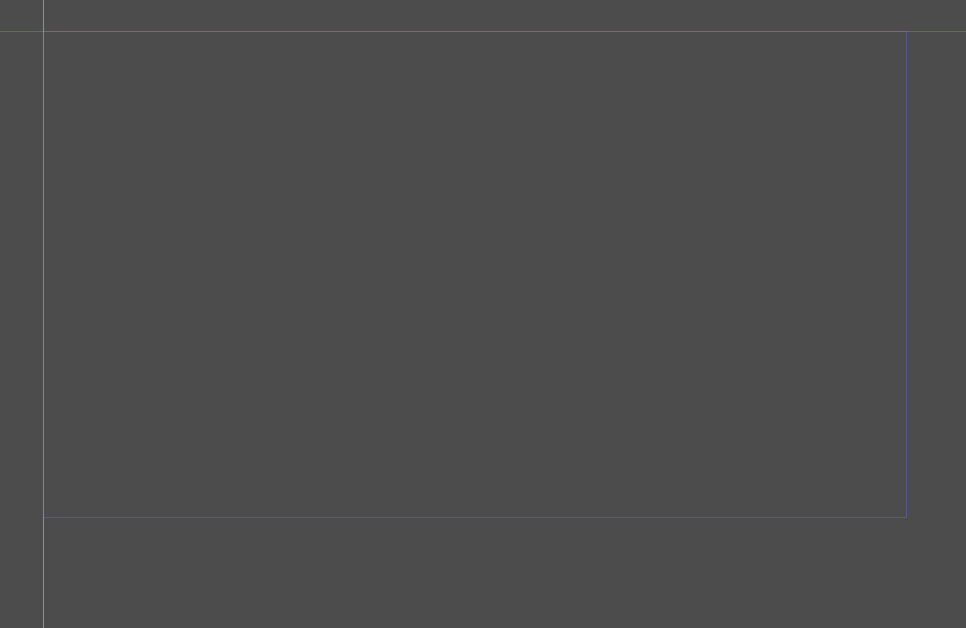
Первый проект ребят, направленный на создание движения персонажа и изучение и тренировки работы с узлами.

Т.к. компьютер будет общий, пусть создадут в папке Student свои папки и все проекты будут содержать там.

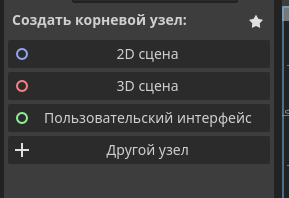
# 1 урок

## Создание уровня и лабиринта

На этом проекте мы еще не работаем с камерой и поэтому игра будет ограничена в пределах игрового экрана, который видно в движке



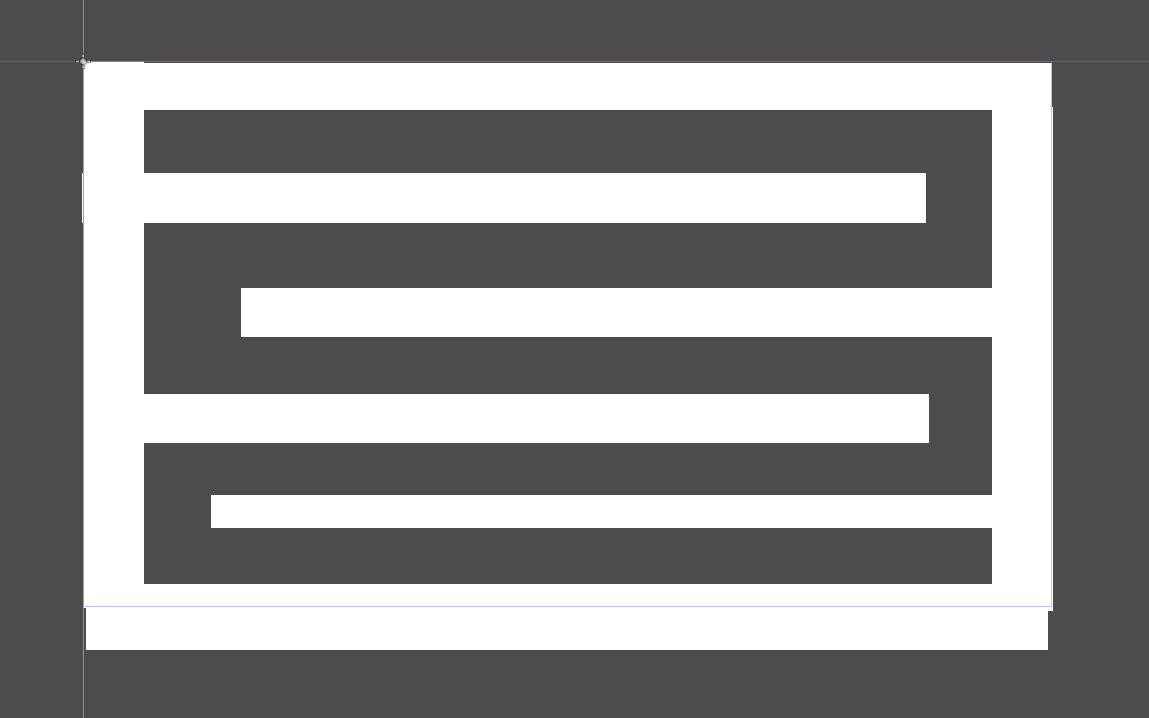
В качестве корневого узла мы выбираем 2D ноду

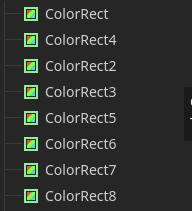


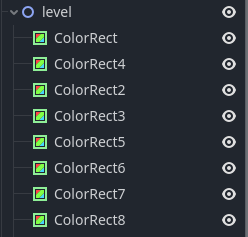
Стены будем делать из ColorRect

**ColorRect** - просто квадрат, который можно закрасить

Строим лабиринт из ColorRect’ов.

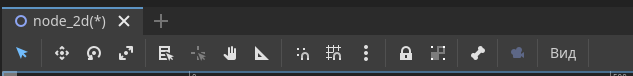






Лучше всего однотипные узлы сгруппировать в node\_2d и переименовать его в говорящее название типа level или walls.

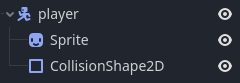
Добавляем столько, сколько нужно. Инструменты для изменения размеров ColorRect представлены вверху сцены



1. Курсор. Самый основной инструмент
2. Режим перемещения.
3. Поворот вокруг оси
4. Расширение объекта

## Создаем игрока

Создаем основные узлы для персонажа, под контролем игрока.



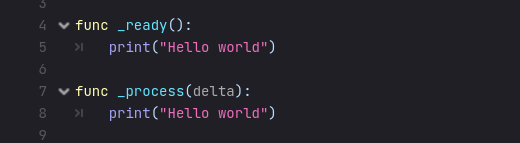
Узел **CharacterBody2D** в Godot используется для создания персонажей, которые могут перемещаться по игровому полю. Он обеспечивает управление движением персонажа, а также обнаружение столкновений с другими объектами.

Узел **CollisionShape2D** используется для создания коллизий объектов в игре. Он позволяет определить форму и размер объекта, что позволяет обнаруживать столкновения с другими объектами и реагировать на них.

Узел **Sprite** содержит в себе картинку.

## Переходим к работе со скриптом

Создаем скрипт у **Node2D**. Про создание скрипта у игрока пока ничего не говорим, а если кто-то спросит, то сказать, что мы сперва напишем управление из корневого узла и после этого можем порассуждать на эту тему (почему стоит писать скрипт для управления все же в игроке, а не в корневом узле уровня).



Пусть попробуют вызвать сообщение в консоль через метод **ready**, а потом через **process**.

Будет необходимо разъяснить этот момент: разницу между двумя методами: ready и process.

Метод `**ready**` и метод `**process**` являются двумя основными методами в **Godot Engine**, используемыми для определения логики поведения объектов в игре.

Метод `**ready**` вызывается один раз во время инициализации объекта и используется для настройки начальных параметров и компонентов объекта. В этом методе вы можете инициализировать переменные, создавать дочерние объекты, устанавливать коллайдеры и т.д. Он выполняется до начала игрового цикла и предоставляет возможность подготовить объект к работе.

Метод `**process**` вызывается на каждом кадре отрисовки и используется для обновления состояния объекта. В этом методе вы можете добавить логику перемещения, анимаций, проверки столкновений и обработки ввода пользователя. Логика, размещенная в методе `**process**`, будет выполнена на каждом кадре, поэтому он обеспечивает непрерывное обновление состояния объекта.

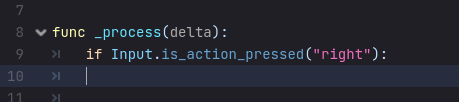
Разница между `**ready**` и `**process**` состоит в том, что `**ready**` вызывается только один раз при инициализации объекта, а `**process**` вызывается на каждом кадре отрисовки. `**ready**` используется для настройки начальных параметров, а `**process**` - для обновления объекта во время игры.

*Оба метода являются важными для разработки игр в Godot Engine и предоставляют гибкую архитектуру для определения логики объектов и контроля их поведения.*

Задаем логичный вопрос ученикам: Если мы хотим добавить управление игроку, то что нам для этого нужно? Что сделать чтобы персонаж начал двигаться?

Верно. Добавить управление. Поэтому добавим его через *Проект -> Настройки проекта -> Список действий.* В открывшемся окне находим поле *Добавить новое действие* , прописываем его имя: right. Затем добавляем кнопку управления.

Переходим в скрипт и можем порассуждать всегда ли игрок нажимает на кнопки? (наводим на мысль, что нам нужно условие нажатия кнопки, ведь игрок может нажать на кнопку, а может не нажать!)



Оформляем в условие и разберем компоненты этой строки

Вот компоненты этой строки:

- **if** : это условный оператор, который позволяет выполнять определенный блок кода только при выполнении определенного условия.

- `***Input***`: Это встроенный класс в **Godot Engine**, который предоставляет функционал для работы с вводом от пользователя. Класс `***Input***` содержит набор методов для проверки состояния клавиш, кнопок, мыши и тачскрина.

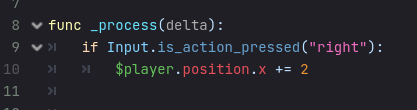
* Классы в программировании - это шаблоны или модели, которые определяют состояние и поведение объектов. Они представляют собой абстрактные типы данных, которые могут содержать свойства (переменные) и методы (функции), которые могут быть использованы для создания экземпляров объектов. Классы могут быть наследованы другими классами, что позволяет создавать более сложные структуры и иерархии объектов. Они являются одним из основных инструментов объектно-ориентированного программирования.
* Методы и функции это два разных термина, но они могут выполнять схожие задачи.
* Функция - это блок кода, который может принимать аргументы, обрабатывать их и возвращать результат. Функции могут быть вызваны из любой части программы.
* Метод - это функция, которая определена внутри класса и может быть вызвана только у объектов этого класса. Методы могут иметь доступ к свойствам объекта и изменять их состояние.

Таким образом, методы являются частным случаем функций, определенных внутри класса.

- `***is\_action\_pressed***`: Это метод класса `***Input***`, который используется для проверки состояния нажатой кнопки или клавиши. Он принимает строковый аргумент - название "действия", и возвращает `***true***`, если данное действие (кнопка или клавиша) в данный момент нажата, и `***false***`, если она не нажата. В данном случае, "***right***" - это имя действия, которое, связано с кнопкой или клавишей, отвечающей за перемещение вправо.

Таким образом, строка `***Input*.*is\_action\_pressed***("***right***")` выполняет проверку состояния нажатой кнопки или клавиши с именем "***right***". Если данное действие активно (кнопка или клавиша нажата), то метод вернет `***true***`, иначе вернет `***false***`. Это может быть использовано для управления перемещением или выполнения других действий в игре на основе состояния кнопок или клавиш.

Можно показать сперва другой способ через ***is\_action\_pressed*** и пусть посмотрят разницу.

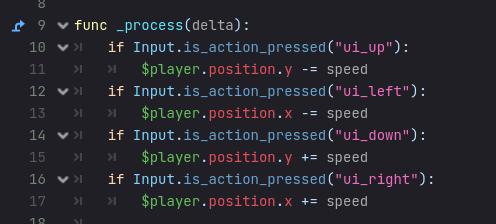


Так как избрано движение вправо (или нет, если они решили сделать движение в любую другую сторону), то выбираем соответствующий вектор (в примере это ось Х и будет координата по этой оси увеличиваться => двигаться вправо).

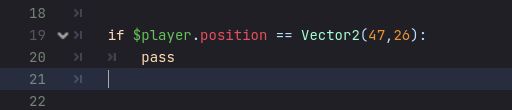
Движение во все другие стороны оставшиеся пусть пробуют сами, если прошло достаточно много времени (от 5 минут), то помогаем.

Можно заметить, что мы часто используем число 2, то тогда его можно поместить в отдельную переменную для нашего удобства. Названия переменных должны отражать содержание или смысл переменной. Поэтому без всяких var aboba = 2 и т.д.





Пока стены мы делать не будем, но сделаем финиш. Тут обсуждаем с ними, как это можно сделать. Скорее всего, я почти уверен, они скажут, что можно сделать через какую-то координату (могут записать ее в переменную или просто сказать, а могут просто молчать). Пускаем их по ложному следу. Предлагаем сделать конкретную точку с координатами

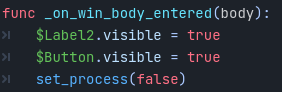


2 урок

1. В ареа2д добавляем узел body\_entered() и делаем обновление сцены



1. Делаем менюшку победы. Для этого добавляем еще 1 ареа2д и добавляем ему коллизию + переименовываем. Добавляем label и кнопку. И пишем следующий код. Добавляем узел body\_entered()



* добавляем кнопке функционал на обновление сцены



1. Добавляем таймер. Для этого добавляем Label и в параметре Text пишем Time:

В коде создаем переменную типа флоат.

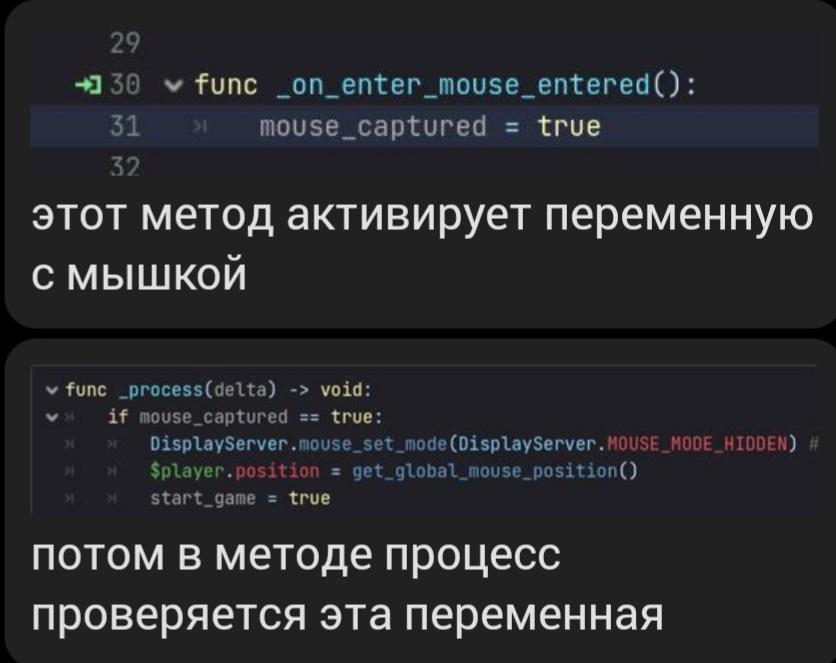


И пишем следующий код. Дельта это время с прошедшего кадра.



1. Можно дать попробовать самим, если не получится показать как





Ссылка на скрпит:

<https://gist.github.com/mykweenn/77992373e8eccb0b494c30e49c4e09f4>