МБОУ «Лицей имени Н. Г. Булакина», г. Абакан

ПРОЕКТ

по информатике

«Создание игр на движке Godot»

Выполнил: ученик 8Д класса Гавровский Андрей

Руководитель:
Журавлева Е. А.,
учитель информатики

Абакан, 2023

Содержание

Введение 3
Основные жанры в гейм-индустрии 4
Что такое Godot? 5
Возможности программы Godot 6
  2Д или 3Д 6
  Gdscript 6
  Шейдеры 6
  Узлы (Nodes) 6
  Аниматор 6
Создаём собственную игру 7
  Прототип 7
  Игра «RobotsOnEarth» 26
Анкетирование учащихся МБОУ «Лицей имени Н.Г. Булакина» 28
Выводы 30

Введение

Умение программировать так же важно, как умение читать. Это требования рынка и нового мира. Об этом рассказали ученые из НИУ ВШЭ в своем докладе «Универсальные компетентности и новая грамотность». Программирование развивает вычислительное мышление, которое учит формулировать проблему, искать решение и анализировать его. Программирование — часть вычислительного мышления и самый эффективный способ его освоить.

Вычислительное мышление помогает детям развивать навыки решения задач, креативное мышление, умение учиться и навыки совместной работы.

В наше время высокоразвитых технологий, всвязи с тем, что компьютер стал общедоступным и необходимым элементом повседневной жизни, трудно назвать человека, который никогда бы не играл в компьютерные игры. Многие люди, играя, наверняка задумывались о том, чтобы создать свою игру. И сейчас самое время, когда это может сделать каждый.

Разработка игр — это наиболее актуальный способ заинтересовать детей и привлечь их в сферу информационных технологий и программирования.

Гипотеза: я предполагаю, что любой школьник может создать свою игру, зная лишь азы программирования.

Объектом моего исследования является программирование. **Предметом** — среда программирования Godot.

Цель проекта: создать свою собственную полноценную игру, интересную сверстникам, на игровом движке Godot.

Задачи проекта:

- провести анкетирование среди школьников 6 – 7 классов и выяснить степень их интереса к программированию;
- расширить общий кругозор школьников, развить их познавательную активность, познакомив их с игровой платформой Godot;
- изучить основные жанры в гейм-индустрии;
- рассмотреть игровой движок Godot, изучить его достоинства и недостатки;
- создать собственную игру;
- вдохновить сверстников на создание своих игр.

Основные жанры в гейм-индустрии

Перед тем как приступить к созданию своей игры, нам нужно определится с её сюжетом. Для этого, в первую очередь, нужно узнать, какие есть игровые жанры.

Экшен — это жанр компьютерных игр, где игра требует хороших игровых навыков, таких как быстрая реакция, умение быстро продумывать последующие действия. К этому жанру, в основном, относятся шутеры, файтинги и платформеры. К этому жанру можно отнести такие игры, как FarCry 3, Mortal Kombat и игры Марио.

Платформеры — да, они буквально только что упоминались, так уж работают жанры в играх, они часто между собой пересекаются. Платформеры — это один из популярнейших у инди-разработчиков жанр. Из названия ясно, что нужно прыгать с одной платформы на другую. Платформеры требуют от игрока умения правильно координировать и выполнять свои движения. К таким играм относятся Марио, Соник, Крэш Бандикут и многие другие.

Аркады - жанр игр, характеризующийся коротким по времени, но интенсивным игровым процессом. Иными словами, это игры, предназначенные для игровых автоматов. К таким играм можно отнести большинство мобильных игр — убивалок времени такие игры как Subway Surfers и Angry Birds.

Головоломки — это жанр игр, где для прохождения уровней от игрока будет требоваться умение логически мыслить. Таких игр довольно много: от простых и маленьких игр головоломок, таких как Zuma или Tetris, до больших игр с сюжетом, таких как одна из моих самых любимых игр - Portal 2.

Симуляторы — это жанр игр, имитирующих процессы из реальной жизни, к примеру вождение машины, строение городов, гонки и так далее. Но иногда бывают симуляторы, созданные для веселья, абсурдные, не пытающиеся добиться реалистичного геймплея игры. Например, тот же Симулятор Козла. К симуляторам относятся такие игры как The Sims, Cities: Skylines, Two Point Hospital, Microsoft Flight Simulator.

Песочницы — это такой жанр игр, где для игрока открывается полная свобода действия и нет никакой конечной цели, которую нужно достичь. Можно делать интересные механизмы в Scrap Mechanic, веселиться по полной в Garry’s Mod и построить всё что придёт на ум в Minecraft.

Квесты — жанр игр, которые представляют интерактивную историю с главным героем, управляемый игроком. Здесь главным аспектом является сюжет и исследование мира, чаще всего путём решения головоломок для открытия следующей местности. К квестам можно отнести Point’n’Click игры (это, к слову, ещё один жанр), к примеру NeverHood. Или же к квестам можно отнести любую игру от Telltale Games, которые представляют собой эпизодическое интерактивное кино, где нередко предстоит совершать трудные выборы.

Стратегии — жанр игр, где для достижения какой-либо цели игроку нужно неоднократно применять своё стратегическое мышление. Таких игр очень много, и особо в них я не играл, поэтому примеры привести не могу, но просто чтобы стало яснее, по сути, шахматы (хоть это и не видео игра) можно отнести к пошаговым стратегиям.

Ну что? У вас уже есть идеи для игр? У меня тоже, так что приступим к следующему пункту.

Что такое Godot?

В IT-индустрии самыми популярными движками для создания игр являются Unity и Unreal Engine. Именно на них создаётся большинство современных игр. Но им есть хорошая альтернатива — Godot. Godot обрёл свою популярность относительно недавно, поэтому о нём ещё знает довольно мало людей, но поверьте, скоро этот движок будет наравне с популярными движками. Godot отличается своей простотой, удобством и скоростью работы.

Движок обладает своим собственным языком программирования под названием gdscript. Он сильно напоминает собой язык Python, с чем и связана его простота. В Godot довольно трудный и непривычный интерфейс, в котором трудно разобраться поначалу. Но уверяю, стоит вам разобраться во всём, вы поймёте как эта программа легка и удобна.

Тем временем как Unity и Unreal требуют большое количество ресурсов для создания игр на них, и далеко не все компьютеры смогут их потянуть, то Godot, как и игры сделанные на нём, потянет почти у каждого, и это является одним из его самых главных плюсов — быстрота. За всё моё время использования, Godot всегда запускался за считанные секунды, и ни разу не зависал, что нельзя сказать про Unreal и Unity.

С остальными функциями движка мы ознакомимся, когда приступим к созданию своей собственной игры.

Достоинства платформы:

- является абсолютно бесплатной программой с открытым исходным кодом.
- отличен как в 2Д так и в 3Д играх.
- программа занимает не больше 100 Мб и не требует мощного компьютера.
- простой и удобный интерфейс и собственный язык, чем-то напоминающий Python.

К недостаткам можно отнести малое количество материалов на русском языке.

Возможности программы Godot

Прежде чем мы начнём делать нашу игру, давайте узнаем, на что, вообще, способен Godot. Очевидно, что Godot не так хорошо продвинут, как другие игровые движки. Но он может выполнить почти всё, что мы можем пожелать для своей игры.

2Д или 3Д

Godot способен работать как в 2Д, так и в 3Д. Godot может быть хуже в 3Д, чем другие движки в плане графики и оптимизации, но это не значит, что он плох. Если хотите сделать 3Д игру в Годоте, то всё в ваших руках! В 2Д же Godot способен на всё что сможет тот же Unity.

Gdscript

Как я уже и говорил, Godot имеет собственный язык программирования, который прост и лёгок для изучения. Так же Godot поддерживает более трудные языки - C++ и C#, но рекомендуется использовать именно gdscript, так как если понадобится помощь с игрой, то все гайды, скорее всего, будут только на языке gdscript. Многие жалуются на то, что gdscript в разы медленнее C++ и C# , но это не так! Он действительно медленнее, но не так сильно, как говорят. Вы вряд ли вообще сможете заметить разницу в скорости.

Шейдеры

Благодаря шейдерам в Godot'е можно создавать самые разные красивые вещи: от обычной обводки объекта, до реалистичной воды с отражениями. Для того, чтобы делать собственные шейдеры, нужно выучить их собственный язык. Для некоторых новичков это может быть далеко не просто, поэтому есть сайты в интернете, где можно скачать уже готовые примеры шейдеров.

Узлы (Nodes)

Узлы в Godot'е представляют собой всё, что мы используем в нашей сцене, то бишь, это обычный предмет, игрок, а может быть это какой-нибудь текст, а может это встроенный аниматор, с помощью которого персонажи могут менять свои позы. Система узлов - это то, что отличает этот движок от других. Её необычность - это причина, из-за которой новичкам трудно даётся это приложение. Разобраться в узлах поначалу может показаться очень трудным, но не стоит паниковать. Вы быстро поймёте, как ими пользоваться, и поймёте какие они на самом деле удобные и простые в использовании.

Аниматор

Когда я говорил об узлах, то уже упоминал аниматор. Аниматор - это один из вспомогательных узлов, который должен облегчать разработку игры, для того, чтобы разработчику не нужно было писать большой объем кода. Аниматор, как мне кажется, одно из главных достоинств Godot'а. Этот вспомогательный узел способен на многое. Один лишь вспомогательный узел, который сможет анимировать как простую ходьбу игрока, так и полноценную катсцену с большим количеством движущихся объектов, с возможностью вставлять звук в анимации и отдельные функции из скриптов. То есть, если я пропишу в скрипте функцию, которая, к примеру, положит в руку моего персонажа какой-нибудь предмет, то я смогу в любой момент вызвать эту функцию в аниматоре. Каждый узел аниматора имеет собственный набор анимаций и, поэтому, если мы захотим, мы можем даже включить какую-нибудь анимацию в аниматоре из ДРУГОГО аниматора! Сколько возможностей и функций в одном лишь узле.

Создаём собственную игру

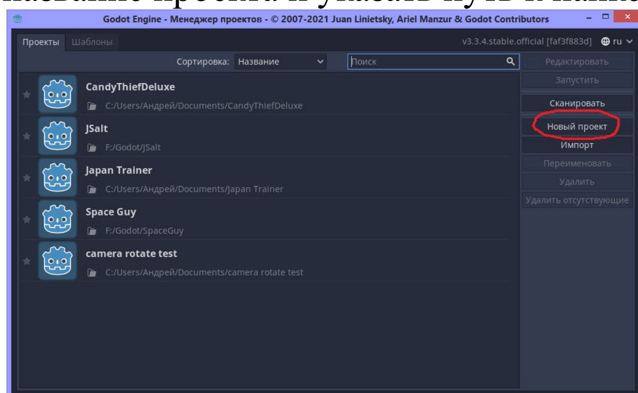
Создавая свою самую первую игру, нужно учитывать свои способности и умения. Начинающий разработчик часто не может сразу продумать этапы создания большого проекта, поэтому желательно начинать с самых маленьких и простых игр и, постепенно, делать более масштабные и продвинутые игры.

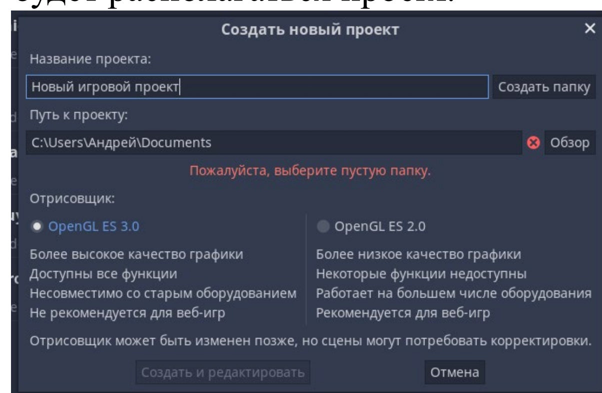
Первым проектом может быть что угодно: какой-нибудь простенький кликер или что-то другое. Я сделал достаточно простую игру. В ней игрок играет за коробку, двигающуюся влево или вправо, которая должна ловить падающие конфеты и уворачиваться от бомб.

Прототип

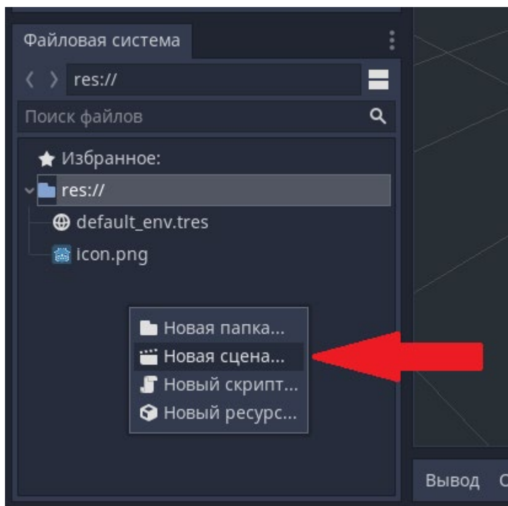
Я замечал, что многие, когда начинают делать игру, первым делом создают её главное меню, а не саму игру. Но с этого не следует начинать. Причина в том, что, если вы будете первым делом работать над меню, то есть большая вероятность, что вы просто потратите время впустую. Потому что, когда дойдёт очередь делать геймплей, вы можете понять, что ваша идея для игры не так хороша, как казалась вначале или же просто вы ещё не готовы создавать игры подобной сложности. Поэтому трудиться над меню в начале разработки не стоит, так же как создавать ресурсы для игры (по типу моделей, спрайтов, текстур и так далее). Перед этим нужно создать просто рабочий прототип игры, чтобы понять, насколько хорош у неё геймплей и следует ли вам её продолжать, дабы не тратить много времени впустую.

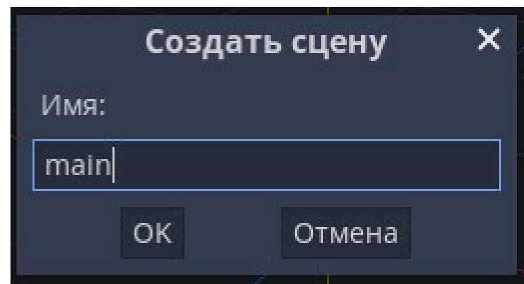
Давайте, наконец-то, приступим к созданию нашей игры. При входе в программу мы видим пустой список ваших проектов (или не пустой, а с демонстрационными играми, смотря откуда вы скачали Godot). Справа жмём на кнопку «Новый проект», после чего появляется меню, где нас просят ввести название проекта и указать путь к папке где будет располагаться проект.



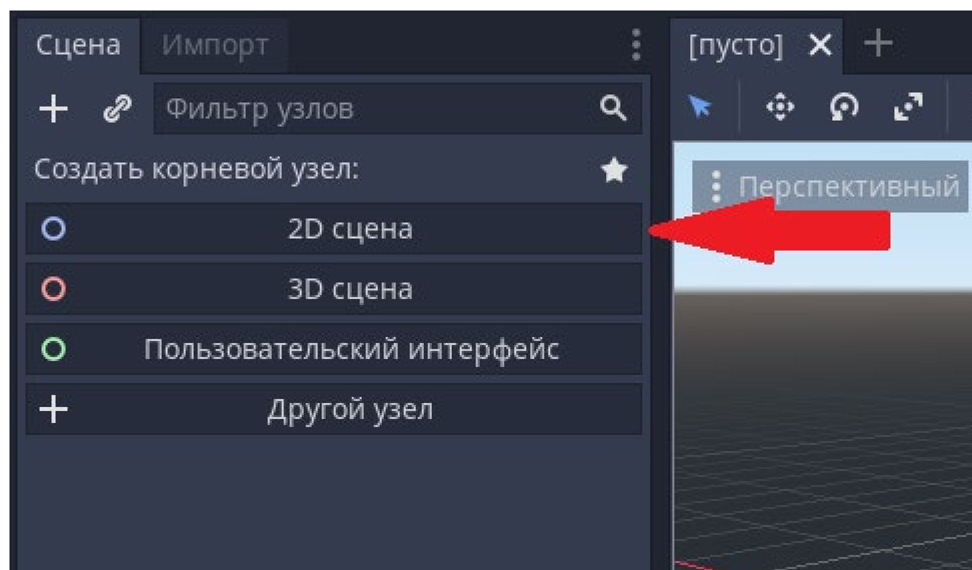


После того как мы всё указали, нажимаем на кнопку «Создать и редактировать». После чего появляется наш пустой проект. Первым делом создадим сцену, в которой всё будет происходить. Смотрим влево вниз, это наш проводник, он показывает все файлы проекта, жмём там по пустому месту правой кнопкой мыши, и выбираем создать новую сцену. Назовём её как хотим, к примеру «main», нажимаем ОК.



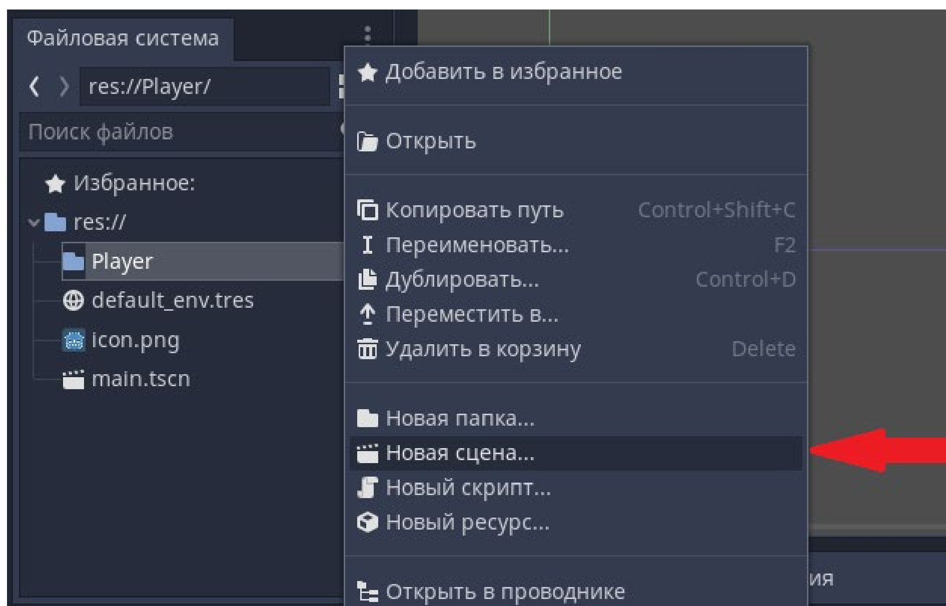


Теперь посмотрим влево и вверх. Тут мы сразу видим те самые узлы, про которые было сказано вначале. Это самые основные, корневые узлы, но они могут быть и другими. Выбираем узел 2Д, так как мы делаем 2Д игру.

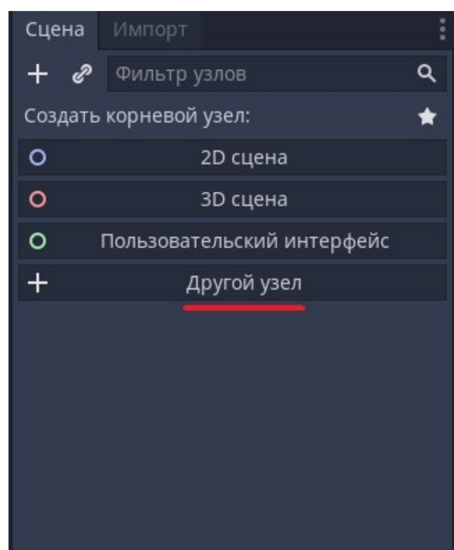


Теперь мы можем сразу сохранить сцену, нажав Ctrl + S, после чего сцена сразу появится в проводнике. Это будет основная сцена, где мы и будем ловить конфеты.

Пока что оставим эту сцену пустой, и создадим новую. Но для этого мы в проводнике создадим новую папку, назовём её Player. В этой папке создадим сцену и назовём её так же. Это сделано для удобства, чтобы все файлы связанные с игроком класть в эту папку и не путаться в файлах.



В этот раз основные узлы нам не нужны, поэтому жмём на «Другой узел».



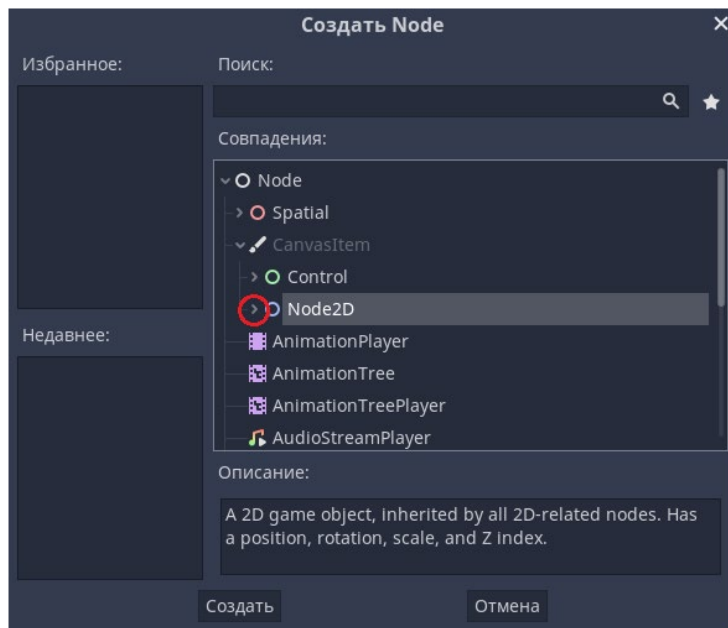
Нам нужен такой узел, с помощью которого можно будет осуществлять движение коробки. Для этого существует объект «физические тела». Их есть 3 варианта: KinematicBody, RigidBody и StaticBody.

KinematicBody используется для движения предмета, не взаимодействуя с физическими телами и явлениями. Гравитации у этого предмета не будет, если её нужно добавить, то её делают сами с помощью скриптов. Это самый удобный и часто используемый узел для движения игрока. RigidBody так же можно использовать для игрока, но чаще его используют для предметов, с которыми можно взаимодействовать, которые можно уронить и так далее.

StaticBody используется в качестве препятствия, стен, пола, то есть физический статический объект, который сам по себе не должен двигаться. С ним могут взаимодействовать как KinematicBody, так и RigidBody.

Для нашей коробки мы будем использовать KinematicBody, так как нам не нужно, чтобы на коробку влияли физические явления, такие как гравитация. Нужно, чтобы она могла двигаться только в две стороны.

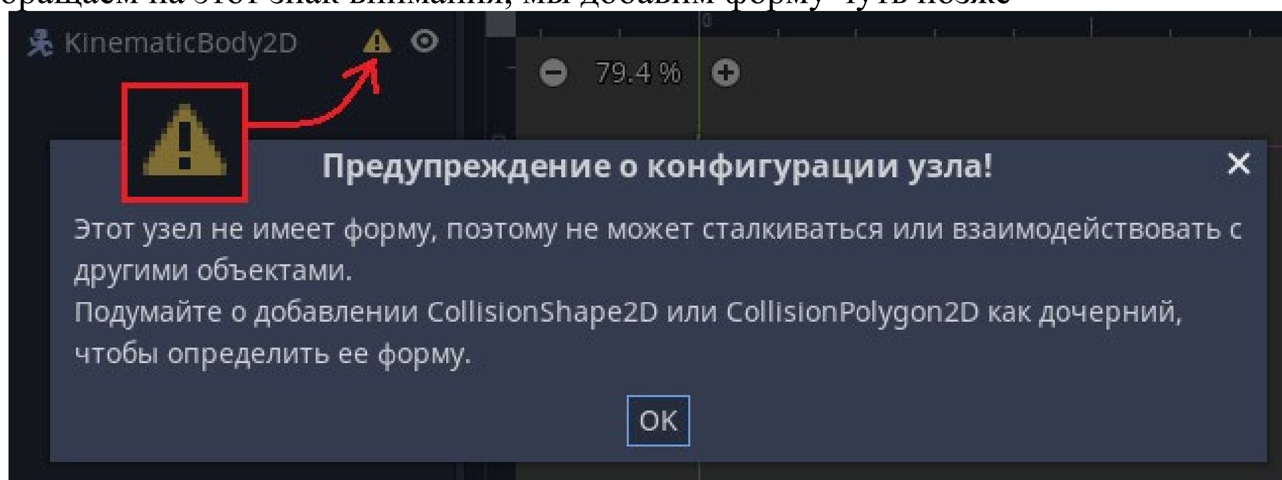
Физические тела могут быть созданы в 2D или в 3D. Так как у нас 2D игра, мы ищем 2D физические тела. Для этого жмём на стрелочку в Node2D



Открывается очень большой список разных узлов, в котором очень трудно искать нужные вещи, особенно, когда не знаешь их назначение. Но ничего, в скором времени мы запомним все, что нам нужно.

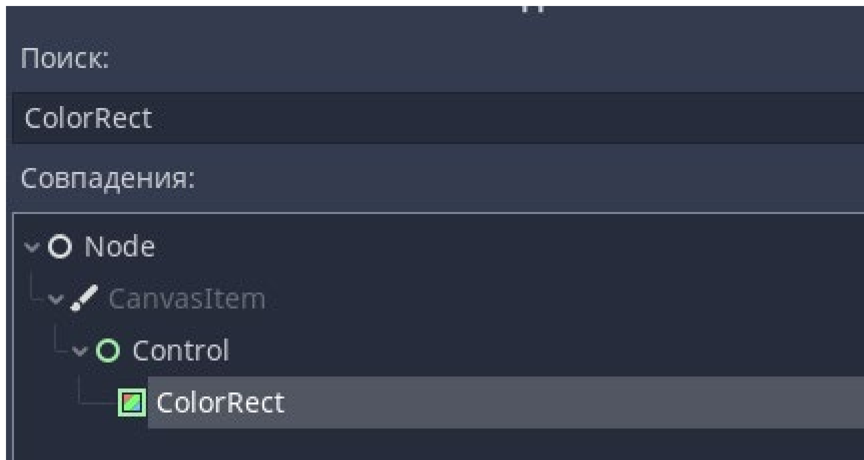
Итак, в открывшемся списке, почти в самом верху, мы видим CollisionObject2D, раскрываем его так же нажав на стрелочку, в новом списке открываем ещё и PhysicsBody2D, и там выбираем уже наш KinematicBody2D, кликнув на него два раза.

Справа от нашего узла появляется восклицательный знак. Он означает, что нет CollisionShape, узла который определяет форму тела, если узла нет, значит и формы тела нет, а значит KinematicBody не сможет касаться других предметов. Пока что не обращаем на этот знак внимания, мы добавим форму чуть позже

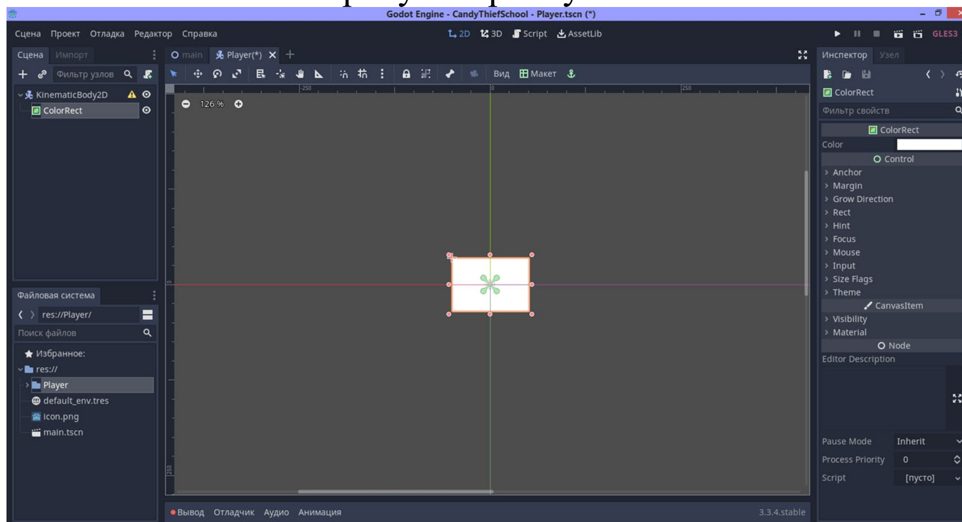


Добавим спрайт самой коробки, но, так как у нас его нет, мы можем по-быстрому его сделать, либо же на время обойтись без спрайта и сделать просто прямоугольник. Так мы и поступим.

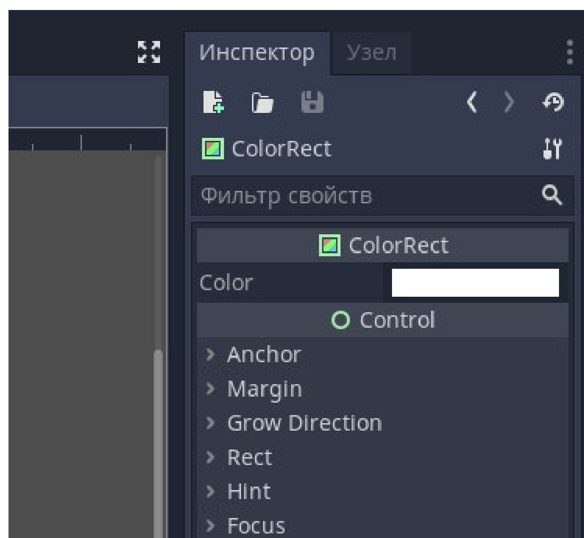
Для этого под KinematicBody2D добавляем ещё один узел. Жмём на него правой кнопкой и сверху выбираем «Добавить дочерний узел». Ищем прямоугольник, но, чтобы как в прошлый раз не раскрывать кучу списков, мы просто введём в поиск ColorRect. Нажмём два раза на подсвеченный узел, и он добавляется в нашу сцену.

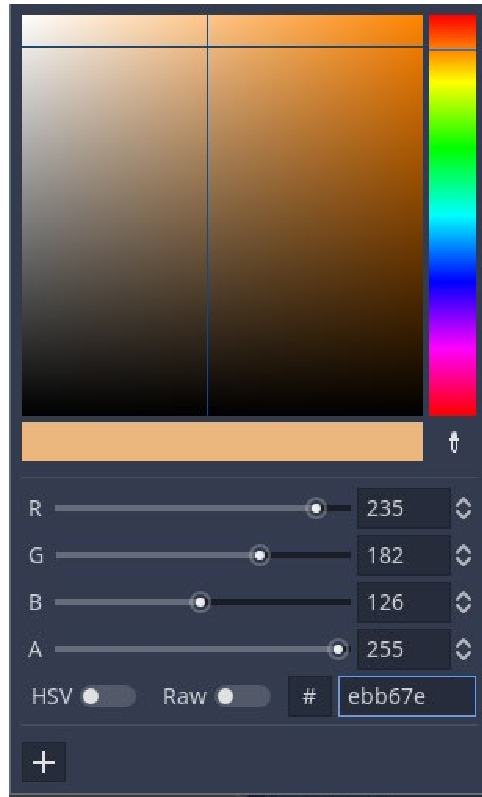


После этого располагаем наш прямоугольник по центру и меняем его размер так, чтобы был немного похож на широкую коробку.

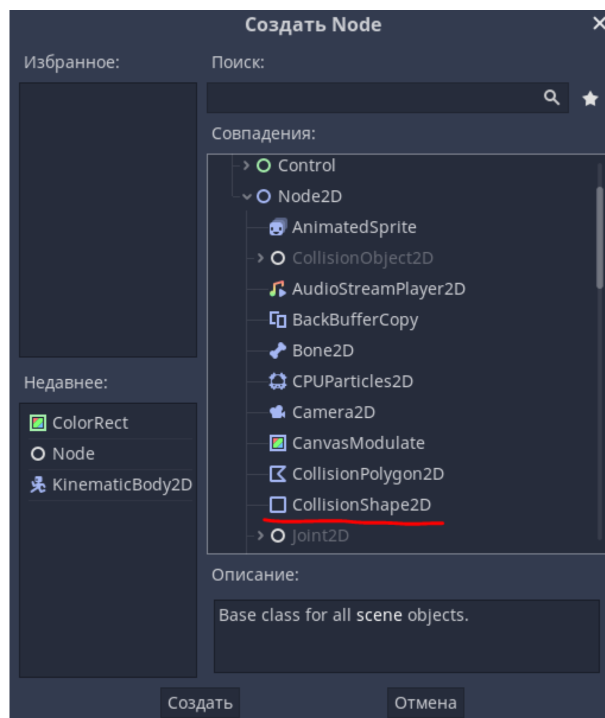


Теперь выберем прямоугольник и в окне справа поменяем его цвет на цвет как у коробок.

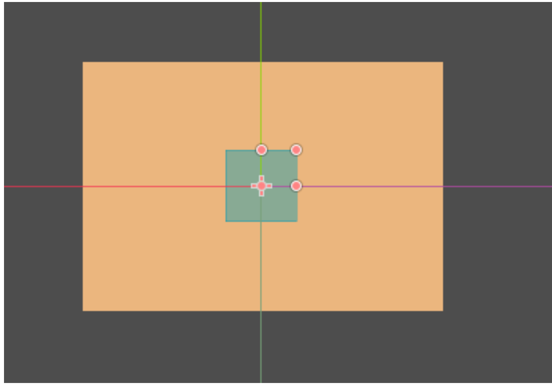


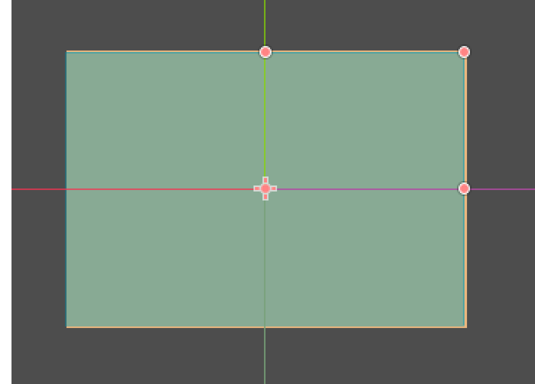


И вот теперь вставим узел CollisionShape, чтобы задать форму. Жмём по KinematicBody2D правой кнопкой, жмём добавить дочерний узел. Раскрываем Node2D, ищем CollisionShape2D, кликаем по нему два раза.



Выделяем CollisionShape2D, в инспекторе жмём на Shape и выбираем там RectangleShape2D. На нашей коробке появляется квадрат, это и есть форма нашей коробки, её не будет видно в игре, по сути форма это то что мы «чувствуем», а сам спрайт коробки это то что мы видим, в играх то что мы чувствуем и то, что мы видим может быть разных размеров. Растягиваем форму так же, как и коробку в длину и ширину.





По сути, наша коробка готова, осталось только сделать ей управление. Для удобства переименуем наш KinematicBody2D в Player. Жмём по его названию 2 раза, стираем надпись и пишем вместо неё Player. Теперь жмём по Player правой кнопкой и выбираем «Прикрепить скрипт».

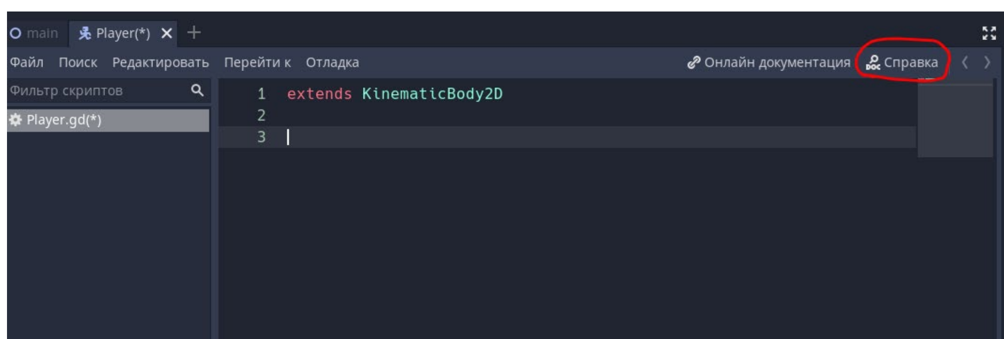
Название скрипта менять необязательно, жмём создать. Всё что там есть стираем кроме самой первой строчки.

```
1  extends KinematicBody2D
2
3
4  # Declare member variables here. Examples:
5  # var a = 2
6  # var b = "text"
7
8
9  # Called when the node enters the scene tree for the first time.
10 func _ready():
11 	pass # Replace with function body.
12
13
14 # Called every frame. 'delta' is the elapsed time since the previ
15 #func _process(delta):
16 #	pass
17
```

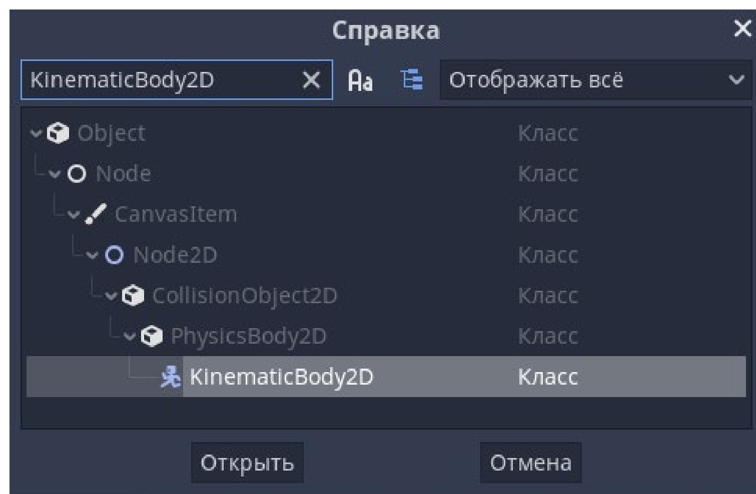
```
Перейти к   Отладка
1  extends KinematicBody2D
2
3
```

Самая интересная часть! Что делать? Ничего непонятно. В первое время у вас такое будет часто, но для таких незнающих, как мы есть специальная оффлайн справка в приложении (к сожалению, она полностью на английском) и официальная онлайн документация с гайдами.

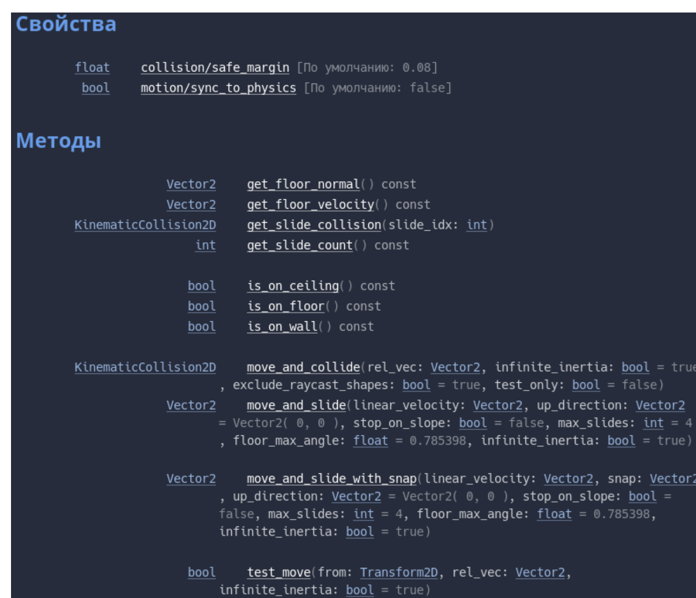
Сверху в правом углу откроем справку.



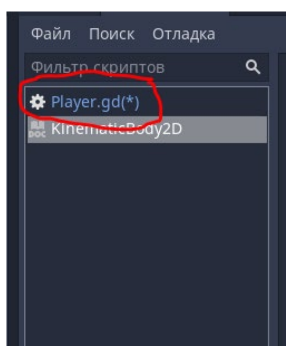
Так как мы осуществляем движение нашей коробки благодаря узлу KinematicBody2D, то в справке должно быть написано, как это сделать. Поэтому в открывшемся окне в поиск мы пишем KinematicBody2D и открываем его.



Немного пролистав ниже, мы видим все методы и свойства узла.



При нажатии на любой из методов или свойств, страничка пролистается вниз к его описанию. Нажмём на move_and_collide, страничка пролистывается к его описанию. В описании указано, что благодаря методу, объект будет двигаться, и если столкнётся с препятствием, то остановится. Это то, что нам нужно! Хоть в нашей игре и нет препятствий. Правее переключимся обратно к нашему скрипту.



Отступим пару строчек и напишем func _physics_process(delta):.

Слово func означает функцию или метод. Есть уже готовые функции, а можно указывать свои функции в скрипте, которые потом можно вызывать.

Функция _physics_process означает, что всё что указано под этой функцией

будет выполняться всё время, независимо от fps игры, то есть все действия будут выполняться с одинаковой скоростью. Есть функция просто _process. Все действия под функцией будут выполняться каждый кадр, то есть скорость выполнения действий будет зависеть от fps, и если меньше fps, то и скорость меньше, если больше, то и скорость больше.

Поскольку нам нужно чтоб коробка двигалась с одинаковой скоростью, независимо от fps, выбираем функцию _physics_process.

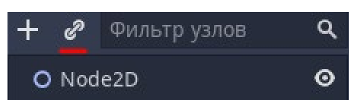
Теперь мы жмём на Enter, курсор переходит на следующую строчку, но теперь он смещён вправо, потому что всё, что мы сейчас пишем, относится к функции. Если мы переведём курсор обратно влево, то эта и следующие строчки уже не будут к ней относиться.

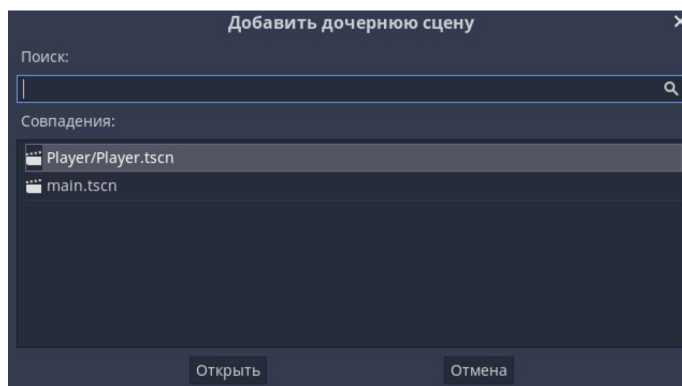
Теперь пишем тот метод движения. Метод - это тоже функция, но в этот раз мы пишем без func, так как мы её не объявляем, как в прошлый раз, а вызываем. Всё, у чего есть скобки на конце - это функция. Пишем move_and_collide(). В скобках нам надо указать определённые переменные. В справке их несколько, но обязательно указать нужно только одну — направление движения. Направление движение обозначается Vector2. Если мы в 2D или Vector3, если в 3D. Пока что просто напишем Vector2.RIGHT.

```
1  extends KinematicBody2D
2
3  func _physics_process(delta):
4      move_and_collide(Vector2.RIGHT)
5
```

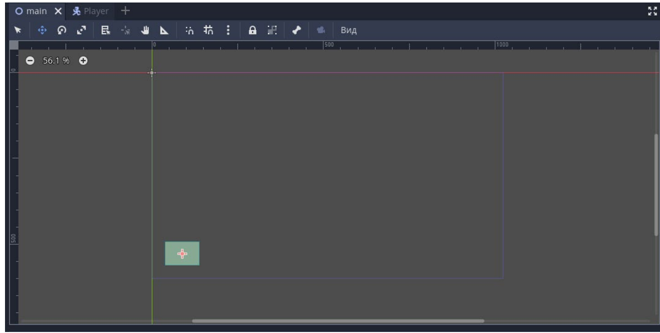
Теперь перейдём в нашу сцену main. Сверху перейдем с режима script в режим 2D.

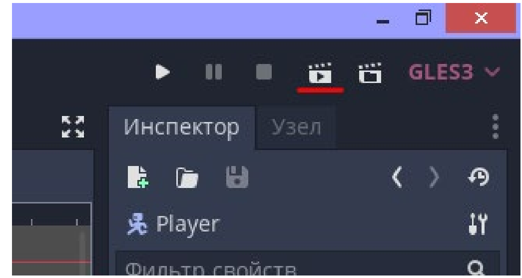
Выделим Node2D и нажмём чуть выше на значок «Добавить дочернюю сцену». В открывшемся окне нажимаем на сцену нашего игрока 2 раза, и он добавляется под Node2D.





Расположим нашего игрока внизу слева игрового окна, и запустим сцену нажав на специальный значок справа сверху или нажав на F6.





Запустим и увидим, как коробка просто медленно двигается вправо, без нашего участия.

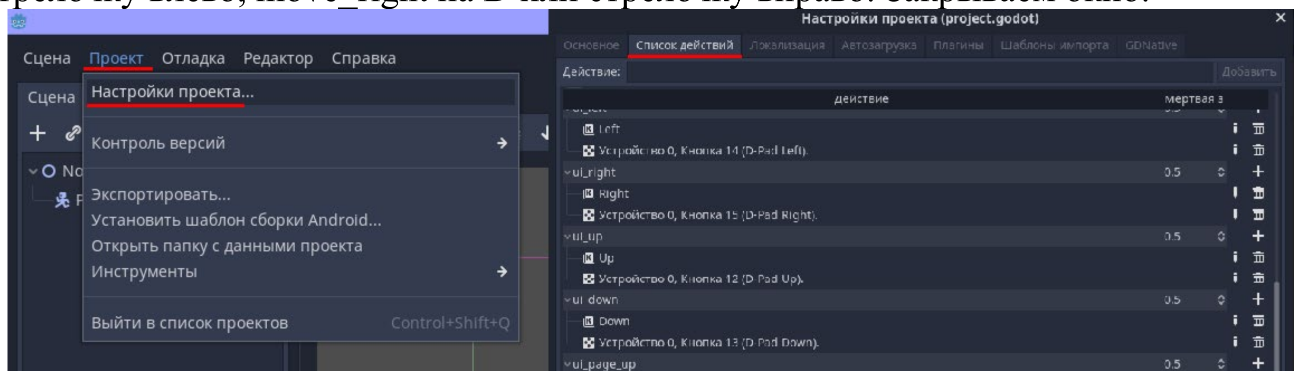
Что нам нужно теперь сделать? Увеличить скорость, и возможность двигаться в разные стороны. Для начала увеличим нашу скорость. Вернёмся в скрипт и перед функцией объявим переменную скорости, speed. Пишем var speed = 100. То есть переменная speed равна 100.

Теперь в нашем методе в скобках умножаем Vector2.RIGHT * speed * delta. Умножили на дельту, чтобы при движении скорость не была слишком большой.

```
1  extends KinematicBody2D
2
3  var speed = 100
4
5  func _physics_process(delta):
6      move_and_collide(Vector2.RIGHT * speed * delta)
7
```

Снова запускаем сцену и видим, что коробка движется быстрее, но всё равно медленно. Приравняем переменную speed к 500, и посмотрим, что получится. Запускаем, теперь скорость слишком высокая. Поставим speed равный 400, и теперь скорость самое то. Если мы сделаем скорость отрицательной, то коробка будет двигаться в противоположную сторону, но мы оставим просто 400.

Теперь надо сделать возможность двигаться в разных направлениях, для этого нам нужно настроить управление. Слева сверху жмём на проект, выбираем «Настройки проекта». В открывшемся окне жмём «Список действий». Добавляем два действия: move_left и move_right. Настраиваем move_left на английскую A или стрелочку влево, move_right на D или стрелочку вправо. Закрываем окно.



Теперь возвращаемся к скрипту. Над функцией объявим новые переменные, var direction = Vector2(), var Velocity = Vector2(). Мы не назначили какое-то определённое число, а указали что эти переменные являются направлением движения, и поскольку

в скобках мы ничего не указали, то по умолчанию направление равно 0, 0 (X и Y).

Теперь в функции мы пишем:

direction.x = Input.get_action_strength("move_right") — Input.get_action_strength("move_left")

Функция get_action_strength() берёт силу нажатия действия. Например, если мы зажмём какую-то кнопку, то её сила нажатия будет равна 1, если отпустим, то сила будет равна 0. Таким образом, мы меняем направление движения по оси X. Если мы нажмём кнопку вправо, то значение будет 1 - 0 = 1, если значение положительное, то двигаться будем вправо. Если мы зажмём кнопку влево, то получится 0 - 1 = -1, значение отрицательное, значит движение будет влево. Если мы зажмём обе кнопки, то получится 1 - 1 = 0, значит коробка будет стоять на месте, никуда не двигаться.

Ниже пишем Velocity = direction * speed * delta. Velocity это скорость движения коробки в настоящий момент, в ней мы умножаем директорию на скорость, таким образом коробка будет двигаться в определённом управлении с указанной скоростью. После этого умножаем на дельту.

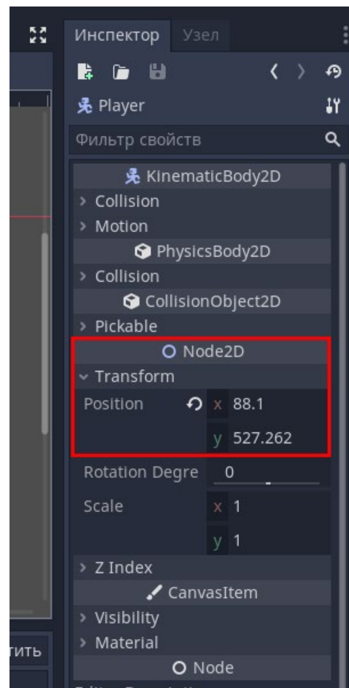
Теперь наконец-то то, что было под скобками в move_and_collide, мы просто заменяем на Velocity. Таким образом у нас получается вот такой вот скрипт:

```
1  extends KinematicBody2D
2
3  var speed = 400
4
5
6  var direction = Vector2()
7  var Velocity = Vector2()
8
9  func _physics_process(delta):
10
11     direction.x = (Input.get_action_strength("move_right")  -
12         Input.get_action_strength("move_left"))
13
14     Velocity = direction * speed * delta
15
16     move_and_collide(Velocity)
17
```

Запустим наш проект, и как видим, теперь коробка может двигаться, в два направления по нажатию кнопок влево или вправо. Но коробка может зайти за границу экрана, давайте это исправим.

Мы поступим так: мы бы могли просто сделать невидимые стены по краям экрана, но это бы выглядело глупо, поэтому лучше сделать это с помощью скрипта. Мы будем проверять, когда позиция коробки будет заходить за края, и если она заходит, то возвращаем коробку на поле.

Если мы выделим коробку и посмотрим в инспектор, ток под вкладкой transform мы можем увидеть её позицию, поворот, размер и так далее.



Если наведём курсор на надпись Position, то нам покажут название этого параметра в скрипте. Его название такое же — position, в принципе логично. Значит теперь мы знаем, как получить позицию объекта в скрипте.

Переходим в наш скрипт. Под методом move_and_collide мы пишем:

```
if position.x < 0:
    position.x = 0
elif position.x > OS.window_size.x:
    position.x = OS.window_size.x
```

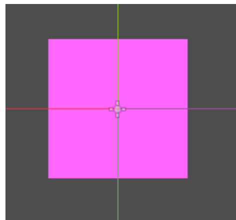
Это можно перевести как: если позиция меньше нуля, то приравниваем позицию коробки обратно к нулю, или если позиция больше ширины экрана, то приравниваем позицию к ширине экрана. По умолчанию, если в вашей игре нет передвигающейся камеры, то 0 всегда будет левым краем окна, значит если позиция меньше нуля, то мы возвращаем коробку обратно на ноль. OS.window_size - это размер окна, про то как его узнать, я посмотрел так же в справке, введя в поисковике window и пролистав немного вниз. OS.window_size.x - это размер экрана по горизонтали, то есть его ширина, а OS.window_size.y - это размер экрана во вертикали, то есть высота. Значит, если позиция коробки больше, чем ширина экрана, то возвращаем коробку на позицию равную ширине экрана.

If - это условие. У условия так же, как и у функции есть своё собственное тело, строки в его теле всегда отступают вправо. Так же забыл сказать, что у условий, так же, как и в объявлении функций в конце есть двоеточие, его необходимо ставить чтобы тело им присваивалось, если не поставить, то редактор будет выдавать ошибку.

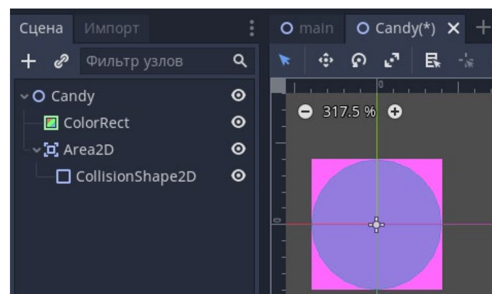
Теперь, если мы запустим игру, то коробка не может зайти за край окна (точнее может, но только на половину, но мне это нравится поэтому я так и оставлю, но хотите по-другому, то это легко сделать).

Наконец-то мы сделали движение коробки! Теперь мы сделаем сами конфетки, которые надо собирать. В проводнике так же создадим папку Candy и в ней новую сцену под тем же названием Candy. Так как наша конфета тоже двигается, корневой

узел мы бы могли сделать так же KinematicBody2D, но поскольку конфета просто падает вниз, мы можем просто сделать движение меняя ей позицию по Y, так что корневым узлом будет просто Node2D – это основной узел для 2D объектов. Теперь так же вставляем ColorRect, и сделаем его примерно таким:



Под наш корневой узел мы добавим Area2D. Это узел, который, выполняет функцию зоны, он может сообщить, когда в неё попадают предметы и так далее. В нашем случае Area2D сообщает, когда в зону попадает игрок. Зона будет размером с саму конфету. Под Area2D добавляем CollisionShape2D, чтобы придать зоне определённую форму, в этот раз в shape мы укажем не квадрат, а CircleShape2D, так как в будущем моя конфетка будет круглой, увеличим её до определённого размера, и в итоге получается как-то так.

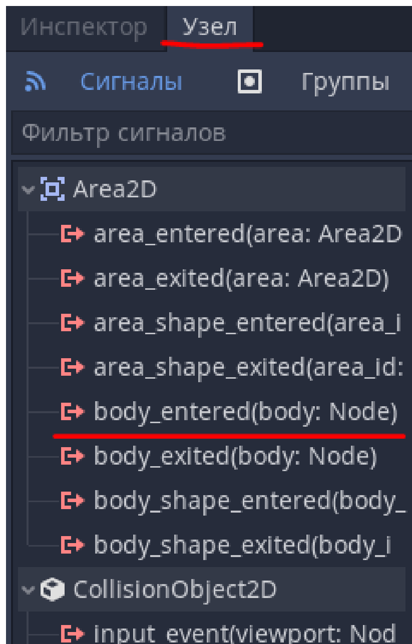


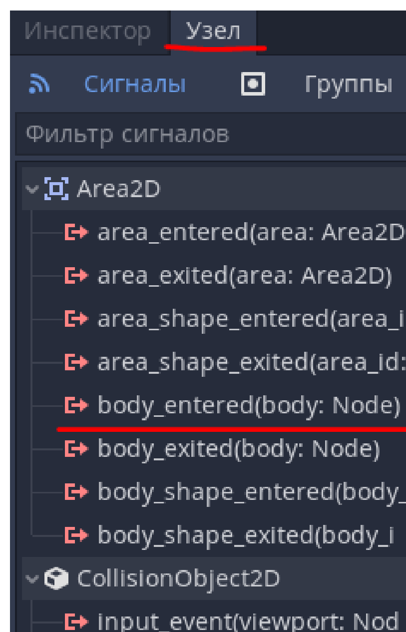
Сохраним сцену нажав на Ctrl + S. Теперь перейдём в main и поставим в ней конфету таким же образом, как и до этого коробку. Расположим её примерно по центру сверху. Перейдём обратно в сцену Candy. Нажмём на наш Node2D правой кнопкой и выберем «Прикрепить скрипт». Так же ничего не меняя в появившемся окне, жмём ОК.

В скрипте сразу объявляем переменную скорости var speed = 250. Теперь так же объявляем нашу функцию func _physics_process(delta):. Под ней мы напишем position.y += speed * delta. Знак += означает что к позиции мы прибавляем значение скорости умноженной на дельта. Получается как-то так:

```
1  extends Node2D
2
3  var speed = 250
4
5  func _physics_process(delta):
6      position.y += speed * delta
7
```

Теперь если мы запустим сцену main, то конфета, поставленная нами, упадёт. Но она просто упадёт мимо коробки, мы не можем её собрать.





Перейдём в сцену Candy, выделим Area2D, и справа поменяем с инспектора на вкладку «узел» и выбираем сигнал body_entered. Сигналы - это способ сообщить другому узлу информацию, что в area зашёл какой-то другой объект и так далее. У каждого узла есть собственные узлы. Так же, кроме как встроенных, в скрипте можно создать и свои сигналы.

Выбираем body_entered, выбираем туда куда наш сигнал будет отправляться, это обязательно должен быть узел со скриптом, выбираем наш корневой узел Node2D.

Теперь в самом низу у нас появилась функция «func _on_Area2D_body_entered(body):». Под этой функцией пишем, что если название тела - «Игрок», то просто уничтожаем нашу конфету командой queue_free().

```
9  func _on_Area2D_body_entered(body):
10     if body.name == "Player":
11         queue_free()
```

Теперь, когда мы подбираем конфету, она будет исчезать. Дальше нам нужно сделать, чтобы конфеты всё время появлялись сверху в разных местах и падали вниз. Сейчас будет не очень понятный процесс, поэтому объяснять в деталях сильно я его не буду, со временем вы сами разберётесь как всё работает.

Перейдём в нашу сцену main, и к её корневому узлу Node2D прикрепим скрипт. Пишем

var candypath = preload("res://Candy/Candy.tscn")

Мы загружаем сцену нашей конфеты, чтобы скрипт в дальнейшем мог её спавнить, то есть ставить на сцену. В скобках мы пишем путь к сцене в проводнике в кавычках. Чтобы получить путь, мы в проводнике жмём по сцене конфеты правой кнопкой и жмём «Копировать путь». После этого вставляем его в скобках и кавычках.

Дальше перейдём обратно к нашей основной сцене, и создадим под корневым узлом узел Timer, для удобства переименуем его в SpawnTimer. В инспекторе его настройках мы поставим галочку под Autostart, чтоб таймер работал с самого начала. Теперь так же переключим с инспектора в узел и присоединим сигнал timeout() к нашему корневому узлу. В скрипте под появившейся функцией пишем randomize(),

чтобы каждый раз когда вызывалась функция её итоги были бы рандомные (случайные), так как нам нужно каждый раз спавнить конфету в разных местах. Дальше пишем

var spawn = candypath

В будущем у нас будут спавниться и конфеты и бомбы, и поэтому здесь будет выбираться между ними, но так как мы пока что не сделали бомбы, будут просто конфеты. Дальше мы пишем

var spawninst = spawn.instance()

Здесь мы превращаем выбранный путь бомбы или конфеты в настоящий предмет, который пока что в игре нам нигде не виден. Вот теперь уже наш объект мы добавляем на сцену, пишем

add_child(spawninst)

Дальше мы просто поменяем позицию нашего объекта:

spawninst.position.y = -20

spawninst.position.x = rand_range(0, OS.window_size.x)

Позицию по Y мы ставим -20, чтоб объекты спавнились чуть выше верхнего края. Позицию по X мы ставим так же от 0 — левого края окна, до OS.window_size.x — правого края. Теперь нам остаётся просто перезапустить таймер.

rand_range() функция выбирающая число между двумя разными числами.

$SpawnTimer.start(rand_range(0.2,1.5))

Чтобы обратиться к определённому узлу, мы всегда пишем $ и путь к этому узлу. Начинаем таймер командой start(). В скобках выбираем время от 0.2 секунд до полторы секунды, чтобы каждый раз объекты спавнились через разное время. Пока что у нас получается такой вот скрипт

```
1  extends Node2D
2
3  var candypath = preload("res://Candy/Candy.tscn")
4
5  func _on_SpawnTimer_timeout():
6  	randomize()
7  	var spawn = candypath
8  	var spawninst = spawn.instance()
9  	add_child(spawninst)
10 	spawninst.position.y = -20
11 	spawninst.position.x = rand_range(0, OS.window_size.x)
12 	$SpawnTimer.start(rand_range(0.2,1.5))
13
```

Теперь, если мы запустим сцену, то конфеты будут падать с потолка, и мы можем их подбирать.

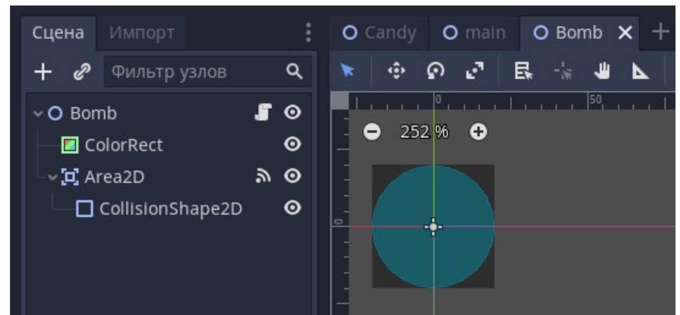
Теперь давайте добавим бомбы. Но перед тем, как мы сделаем бомбы мы сделаем жизни игрока. У игрока будут тратиться жизни при прикосновении с бомбами, и когда жизней станет 0, то мы проигрываем. Перейдём в скрипт игрока. В начале объявим новую переменную:

var lives = 3

Это значит, что в начале жизней будет всего три, три подбора бомбы и игра заканчивается. Теперь создадим уже саму бомбу. Создаём папку Bomb и в ней сцену с тем же названием. Повторяем тот же процесс создания, как и у конфеты, но цвет сделаем более черный.

Теперь прикрепим скрипт к корневому узлу. Так же, как и у конфет сразу объявим скорость и сделаем падение бомбы.

```
1  extends Node2D
2
3  var speed = 200
4
5  func _physics_process(delta):
6      position.y += speed * delta
7
```



Теперь мы так же присоединим сигнал Area2D под названием body_entered. Под сигналом пишем то же самое, но перед queue_free() пишем:
«body.lives -= 1». То есть просто вычитаем одну жизнь.

```
1  extends Node2D
2
3  var speed = 200
4
5  func _physics_process(delta):
6      position.y += speed * delta
7
8
9  func _on_Area2D_body_entered(body):
10     if body.name == "Player":
11         body.lives -= 1
12         queue_free()
```

Перейдём обратно к скрипту игрока. Внизу под _physics_process мы пишем
if lives <= 0:
queue_free()
То есть, если жизней меньше или равны нулю, то уничтожаем коробку, позже сделаем чтоб появлялось меню гейм овера. Получается как-то так:

```
9  func _physics_process(delta):
10
11     direction.x = (Input.get_action_strength("move_right")  -
12         Input.get_action_strength("move_left"))
13
14     Velocity = direction * speed * delta
15
16     move_and_collide(Velocity)
17
18     if position.x < 0:
19         position.x = 0
20     elif position.x > OS.window_size.x:
21         position.x = OS.window_size.x
22
23     if lives <= 0:
24         queue_free()
```

Теперь чтобы проверит работает ли всё, мы сделаем чтоб бомбы могли спавниться с конфетами. Переходим в скрипт нашей основной сцены. После var candypath мы напишем var bombpath = preload(“”) и таким же образом, как и в прошлый раз находим путь бомбы.

Теперь, чтобы в переменной spawn выбиралось между конфетой или бомбой, надо создать массив, я назову его things. Выше метода мы объявляем переменную var things = [candypath, bombpath].

Дальше мы меняем значение var spawn на things[randi()% things.size()]. То есть,

выбираем любое значение из нашего массива.

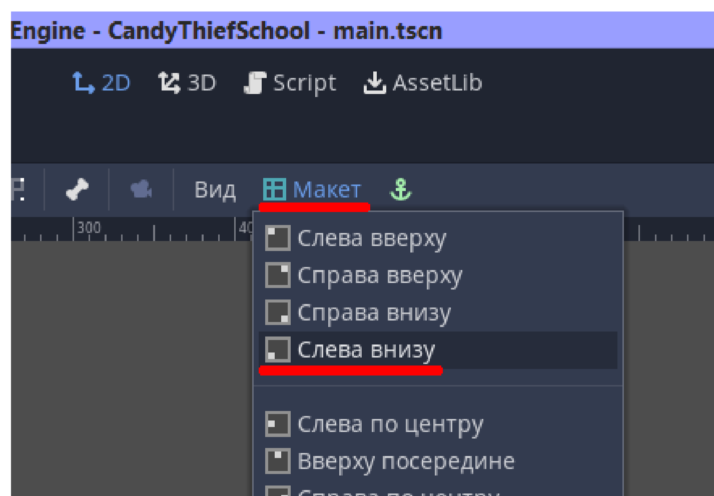
Получается как-то так:

```
extends Node2D

var candypath = preload("res://Candy/Candy.tscn")
var bombpath = preload("res://Bomb/Bomb.tscn")

var things = [candypath, bombpath]

func _on_SpawnTimer_timeout():
	randomize()
	var spawn = things[randi()% things.size()]
	var spawninst = spawn.instance()
	add_child(spawninst)
	spawninst.position.y = -20
	spawninst.position.x = rand_range(0, OS.window_size.x)
	$SpawnTimer.start(rand_range(0.2,1.5))
```

Теперь, если мы запустим сцену, то сверху будут спавниться как конфеты, так и бомбы, и если мы коснёмся бомб 3 раза, то коробка исчезнет.

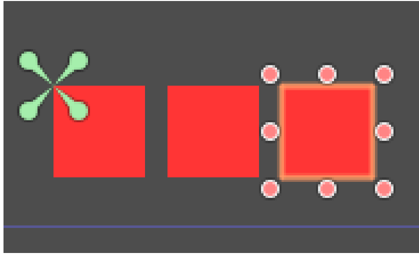
Давайте сделаем счетчик жизней коробки. Можно простым способом — просто текстом показывать количество жизней, а можно сделать сердечки, которые будут исчезать если потратятся жизни.

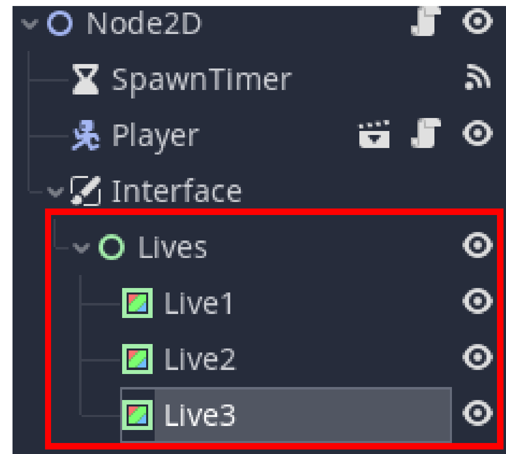
Давайте добавим в нашей основной сцене новый узел — CanvasLayer. Внутри этого узла, как правило делают интерфейс игры, так как он определяет, где какой край окна, и сам может перетаскивать определённые компоненты интерфейса при изменении размера окна. CanvasLayer я переименую в Interface. Создаём в нём узел Control, который, по сути, выполняет те же функции что и Node2D, но в интерфейсе. Переименуем его в Lives. Выделим этот узел и сверху нажмём на кнопку «Макет» и выберем «Слева внизу», ну или там, где вы бы хотели, чтобы были ваши жизни.



Переместим их немного вверх и вправо, чтоб жизни не находились прямо в краю. Теперь в этот узел добавляем сами спрайты жизней, но опять же так как у меня их нет, и мне лень их делать, на время просто поставим ColorRect'ы. Делаем 3 квадрата, так как у нас всего 3 жизни и расставляем диагонально на равном расстоянии друг от друга.

Переименуем эти квадратики в Live1, Live2, Live3.





К узлу Lives мы прикрепим новый скрипт. В скрипте пишем: export var PlayerPath : NodePath — это мы объявляем переменную, путь к узлу игрока. Export означает что эту переменную можно поменять в скрипте, а можно в инспекторе, так мы и поступим позже. Мы написали: NodePath, чтобы указать какого типа переменная, чтобы скрипт знал, что нам нужен именно путь к узлу.

onready var Player = get_node(PlayerPath) — объявляем саму переменную игрока, с помощью которой мы сможем обратиться к переменной жизней в скрипте игрока. Функция get_node() находит узел по пути, который мы указали Onready. Мы пишем, так как в обычных переменных нельзя использовать такие функции как get_node и так далее.

Дальше мы объявляем func _process(delta) – повторять действия каждый кадр. Под функцией пишем такой вот скрипт:

```
6  func _process(delta):
7  	if Player.lives == 3:
8  		$Live1.visible = true
9  		$Live2.visible = true
10 		$Live3.visible = true
11 	elif Player.lives == 2:
12 		$Live1.visible = true
13 		$Live2.visible = true
14 		$Live3.visible = false
15 	elif Player.lives == 1:
16 		$Live1.visible = true
17 		$Live2.visible = false
18 		$Live3.visible = false
19 	else:
20 		$Live1.visible = false
21 		$Live2.visible = false
22 		$Live3.visible = false
```

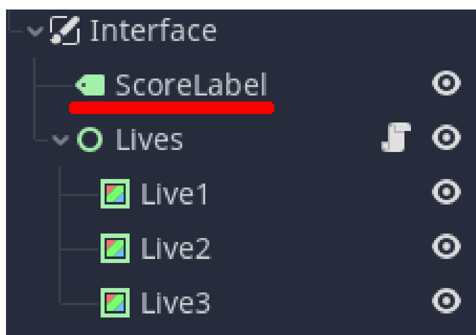
То есть, мы делаем, что, если у игрока 3 жизни, то все 3 жизни видимы. Если у игрока 2 жизни, то только 2 жизни видны и так далее до нуля. Можно было сделать это более быстрым способом, но и так всё работает, главное, что всё просто. Теперь остаётся только указать путь к игроку в инспекторе.

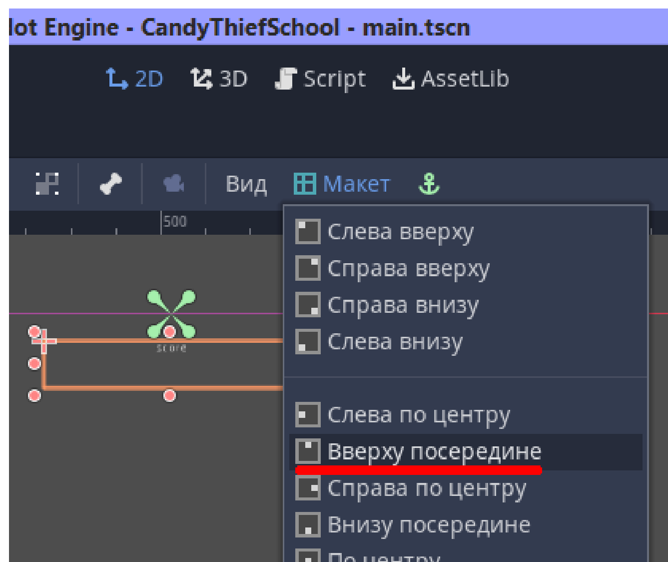
Если мы запустим, то всё будет работать, жизни тратятся когда собираем бомбу, но если потратить все жизни, то выдаст ошибку, так как скрипт жизней будет пытаться найти игрока, а игрок умер и уничтожился, и поэтому его теперь нет на сцене. Мы исправим это позже, пока что не обращаем внимания.

Ну что ж, наша игра уже очень сильно преобразовалась.

Давайте теперь добавим счётчик очков, показывающий, собственно, сколько

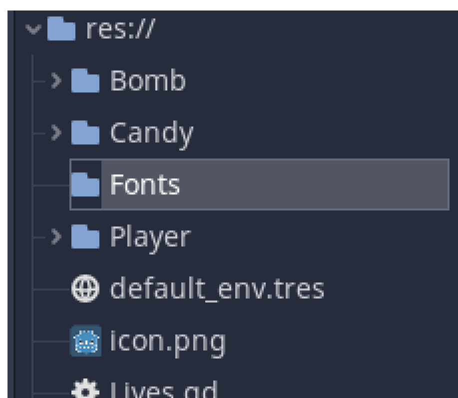
очков мы набрали. Добавим под Interface узел Label. Это узел показывающий текст. Назовём его Score. Немного увеличим его и нажав на кнопку «Макет» выберем его положение «Сверху посередине».





Теперь в инспекторе, в параметре text, напишем score, и если мы посмотрим на то, как он выглядит в окне, то увидим, что он очень маленький.

Чтобы увеличить размер текста нам нужно импортировать в проект шрифт. Поэтому для начала давайте в проводнике создадим папку под названием fonts, туда мы будем кидать наши шрифты.



Шрифты можно скачать с интернета, а можно взять встроенные шрифты, установленные у вас в операционной системе. На время сворачиваем программу, если вы на Windows, открываем проводник, и переходим по пути C:\Windows\Fonts. Здесь у нас находятся все шрифты, которые установлены на ваш компьютер. Мне очень нравится шрифт Impact, вводим его название в поиск, потом открываем обратно Godot и переносим с папки со шрифтами Impact в папку Fonts.

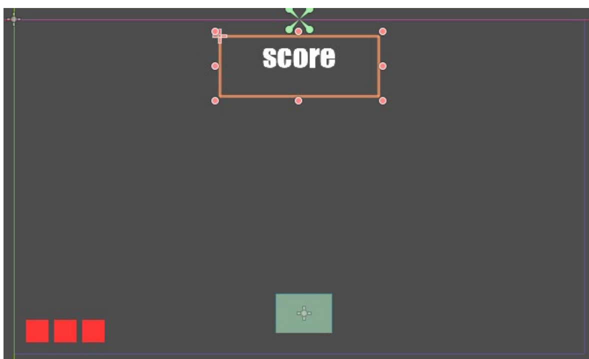
Как только шрифт переместится, снова выделяем Label, и в инспекторе, пролистав чуть ниже, под вкладкой Custom Fonts выбираем «Новый DynamicFont». Потом жмём по нему, выбираем вкладку Font, и в Font Data добавляем наш шрифт Impact. Шрифт поменялся. Теперь так же в Custom Fonts выбираем вкладку Settings и меняем Size пока не станет нужного размера, я поставил размер 56. Сам счёт мы будем так же хранить в игроке. Поэтому в его скрипте до функции мы объявим новую переменную var score = 0. Теперь сделаем чтобы при подборе конфет счёт увеличивался. Переходим в скрипт конфет, и пишем это:

```
9  func _on_Area2D_body_entered(body):
10     if body.name == "Player":
11         body.score += 100
12         queue_free()
13
```

Теперь при подборе конфет к счёту будет прибавляться 100 очков. Переходим обратно в скрипт ScoreLabel и пишем:

```
6  func _process(delta):
7      text = str(Player.score)
```

Теперь текст будет показывать наш счёт. Метод str() в нашем случае форматирует число в обычный текст. Наш прототип готов! Остаётся только добавить спрайты, сделать главное меню, меню проигрыша и отполировать игру до конца!



После того как я окончательно отполировал игру. Мне захотелось чего-то ещё. Мне захотелось сделать такую игру, чтобы полностью подтвердить свою точку зрения о том, что любой школьник может создать собственную и интересную игру.

Игра «RobotsOnEarth»

Если моя прошлая игра с коробкой «Candy Thief», была похожа на очень многие другие игры, то своей новой игрой я хотел сделать что-то оригинальное, то чего не делал ещё никто, и добьюсь я этого наличием сюжета и диалогов.

Моя задумка заключается в том, что это 3Д квест в котором Роботы-инопланетяне прибыли на Землю и хотят её поработить. Они превращают людей в роботов-убийц, чтобы с помощью них уничтожить человечество. Роботы наметили и на тебя свой глаз. Выберись из их тюрьмы прежде, чем они превратят тебя в робота-убийцу.

Первым делом я примерно продумал дизайн уровня, и то, как с него надо сбегать, что находить, что делать, что нажимать, в какой последовательности и так далее.

После этого я сразу начал делать прототип. За движение игрока я взял уже готовый скрипт с интернета, так как на написание собственного у меня ушло бы много времени. Я сделал небольшую комнату и поэкспериментировал над графикой.

Потом я начал делать 3Д модели в программе Blender, и это была моя самая первая попытка в моделировании. Вместо многих моделек по типу столов и тумбочек по началу я просто использовал разноцветные кубы. Модели делал для важных объектов, таких как кнопочки, отвёртки, ключа и так далее.

После того, как примерно за 3 дня я сделал готовый прототип, я начал

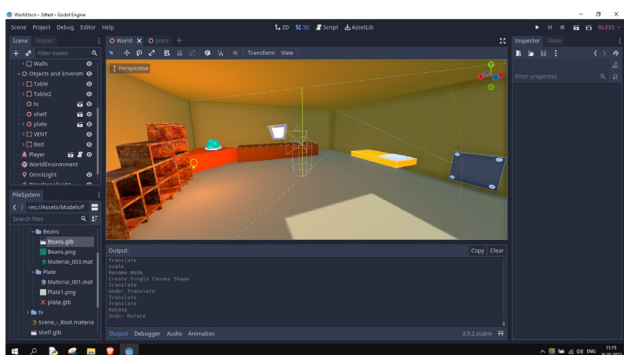
полноценно дополнять комнаты 3Д моделями и текстурами. Самой большой моей проблемой было не само моделирование предметов, а их UV-развёртка (то как текстура развёртывается на объект). Мне очень трудно это давалось, и я до сих пор не уверен насколько правильно и эффективно я их делал.

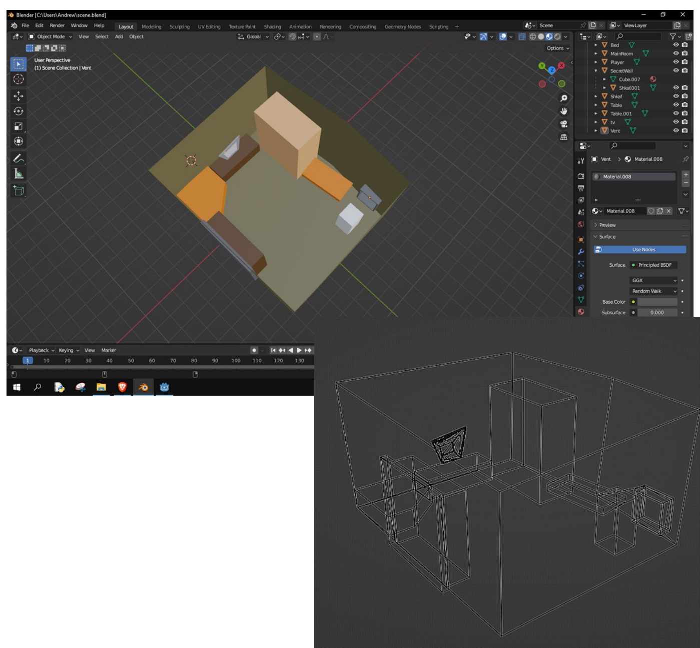
Через ещё примерно 3 дня игра очень сильно преобразилась. И тогда пришла пора делать кат сцены и озвучку. Кат сцены — это сцены, которые происходят между геймплеем, чтобы лучше понимать сюжет игры. Озвучку я делал в программе Audacity, она позволила мне очистить шум своего микрофона, и сделать эффект голоса робота.

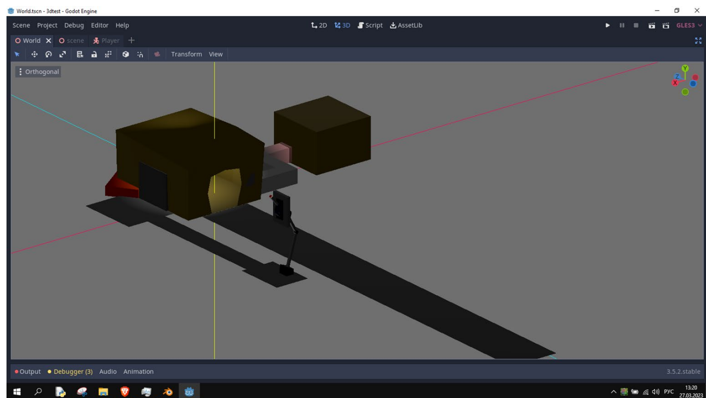
Скриптить сами катсцены оказалось в разы проще чем я предполагал, большую роль в них играл узел Аниматора, который позволяет вставлять всякие эффекты и звуки одновременно, поэтому количество скрипта было минимальным в моих катсценах.

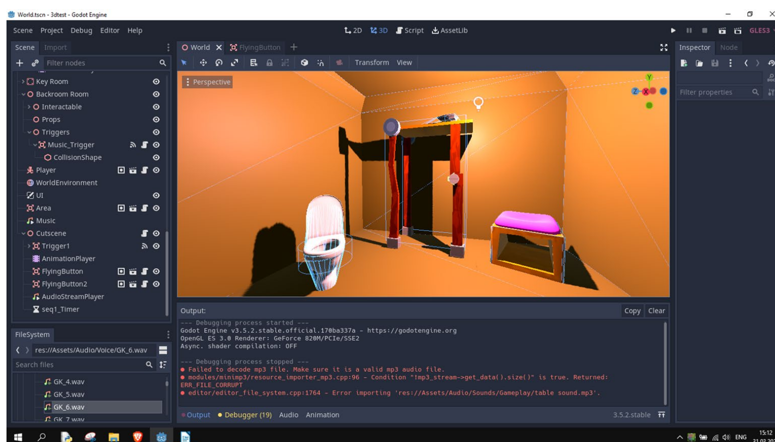
Ну и после этого я начал добавлять звуковые эффекты для нажатия кнопок и разных взаимодействий предметов.

К сожалению, игра не готова, да и это просто моя демонстрация того, что можно добиться за 2 недели. Она предназначена для того, чтобы её можно было пройти за 5 минут. Возможно, в будущем из этого проекта я сделаю полноценную игру, так как в моих глазах, у этой игры есть потенциал.









Анкетирование учащихся МБОУ «Лицей имени Н.Г. Булакина»

В ходе работы над проектом мы провели анкетирование среди учеников 6 – 7 классов с целью выявить степень заинтересованности в изучении программирования. Всего было опрошено 68 школьников. Опрашиваемым было предложено всего два вопроса:

1. Хотите ли вы изучать программирование:
   a. Да
   b. Нет
   c. Я уже умею программировать.
2. Что бы вы хотели создать?
   a. Системное приложение
   b. Игру
   c. Прикладную программу
   d. Другое_______. Укажите, что именно.

В результате выяснили, что 19% детей считают, что уже умеют программировать, а большинство ребят (58%) хотело бы научиться программировать. Детям интереснее всего изучать программирование путём создания собственных простых видеоигр (65%). Системные программы хотят создавать 10% опрошенных, прикладные программы – 19% опрошенных, другое (социальную сеть, чат-боты) создали бы 6% опрошенных школьников.

Поэтому мы сделали вывод о том, что разработка игр — это лучший способ заинтересовать детей и привлечь их в сферу информационных технологий и программирования.

Диаграмма 1.

Диаграмма 2.

Выводы

Работая над проектом, я убедился в том, что можно научиться создавать компьютерные игры. Моя гипотеза подтвердилась. При этом не обязательно быть экспертом в программировании. Исходя из изложенного материала, я пришёл к выводу, что действительно любой школьник может научиться этому. При большом желании можно добиться результатов.

Проведенное мной анкетирование школьников 6-х и 7-х классов показало, что ребята хотят изучать программирование. 58% опрошенных ответили положительно на вопрос «Хотите ли вы изучать программирование?» 19% школьников посчитали, что они умеют программировать, у остальных (23%) интереса к программированию пока нет. При этом среди желающих научиться программировать наибольший интерес вызывают игры. 65% детей хотят создавать свои видеоигры. Такой интерес понятен, ведь создание своей игры – очень увлекательный процесс! Можно придумать свой оригинальный сюжет, который еще никто не использовал.

Процесс разработки видео игр увлекательный и интересный, развивает способность лучше запоминать и обрабатывать информацию, учит находить нестандартные решения сложных задач, развивает интеллект и умственные способности.

Из своего опыта, могу сказать, что лучше начинать процесс разработки видеоигр с программ попроще, которые имеют русскую версию.

Таким образом, потратив немного времени, у меня получилось создать 2 интересные игры, при этом я профессионально программированием не владею. В дальнейшем я хотел бы создать развивающую игру, которую можно было бы использовать в школе на кружке по программированию. Так же я планирую усовершенствовать свои навыки и продолжить свою работу в более сложных программах.

Список источников

1. https://godotengine.org/
2. https://gdquest.github.io/learn-gdscript/
3. https://tmoeg.itch.io/learn-godots-gdscript-from-zero-russian-language
4. https://www.gdquest.com/tutorial/godot/
5. https://dzen.ru/godotengine/