Оглавление

[Блок.1. Концепция 2d игр 2](#_Toc117858670)

[1.1. Настройка окружения. Сцена. GameObject. 2](#_Toc117858671)

[1.1.1. Настройка окружения. 2](#_Toc117858672)

[1.1.2. Настройка Unity3d 4](#_Toc117858673)

[1.1.3. Настройка проекта. 5](#_Toc117858674)

[1.1.4. Редактируем сцену. GameObjcet. 6](#_Toc117858675)

[1.2. Скрипт управление персонажем с помощью Rigidbody2d 8](#_Toc117858676)

[1.2.1. Подготовка персонажа 8](#_Toc117858677)

[1.2.2. Первые строчки кода 11](#_Toc117858678)

[1.2.3. Учим персонажа прыгать и ходить. 14](#_Toc117858679)

[1.2.4. Исправляем неточности. 16](#_Toc117858680)

[1.3. Скрипт управления движением. Компонент Transform. 19](#_Toc117858681)

[1.3.1. Компонент Transform. 19](#_Toc117858682)

[1.3.2. Математика векторов. 21](#_Toc117858683)

[1.3.3. Передвигающаяся платформа. 25](#_Toc117858684)

[1.4. Столкновения, коллизии и триггеры. Компоненты Collider 2D. 28](#_Toc117858685)

[1.4.1. Базовый компонент «Collider 2D». 28](#_Toc117858686)

[1.4.2. Событийные методы столкновения. 30](#_Toc117858687)

[1.4.3. Триггеры. 31](#_Toc117858688)

[1.4.4. Создаём простой телепорт. 32](#_Toc117858689)

[1.4.5. Создаём простой батут. 40](#_Toc117858690)

[1.4.6. Собираем вещи на карте. 43](#_Toc117858691)

[1.4.7. Активаторы и действия. 46](#_Toc117858692)

[Приложение 50](#_Toc117858693)

[Интересные решения 50](#_Toc117858694)

[Эффект тряски (Shake). Глава (1.4.7). Камера. Математика. 50](#_Toc117858695)

[Эффект падающего снега. Глава (нет). Система частиц. 51](#_Toc117858696)

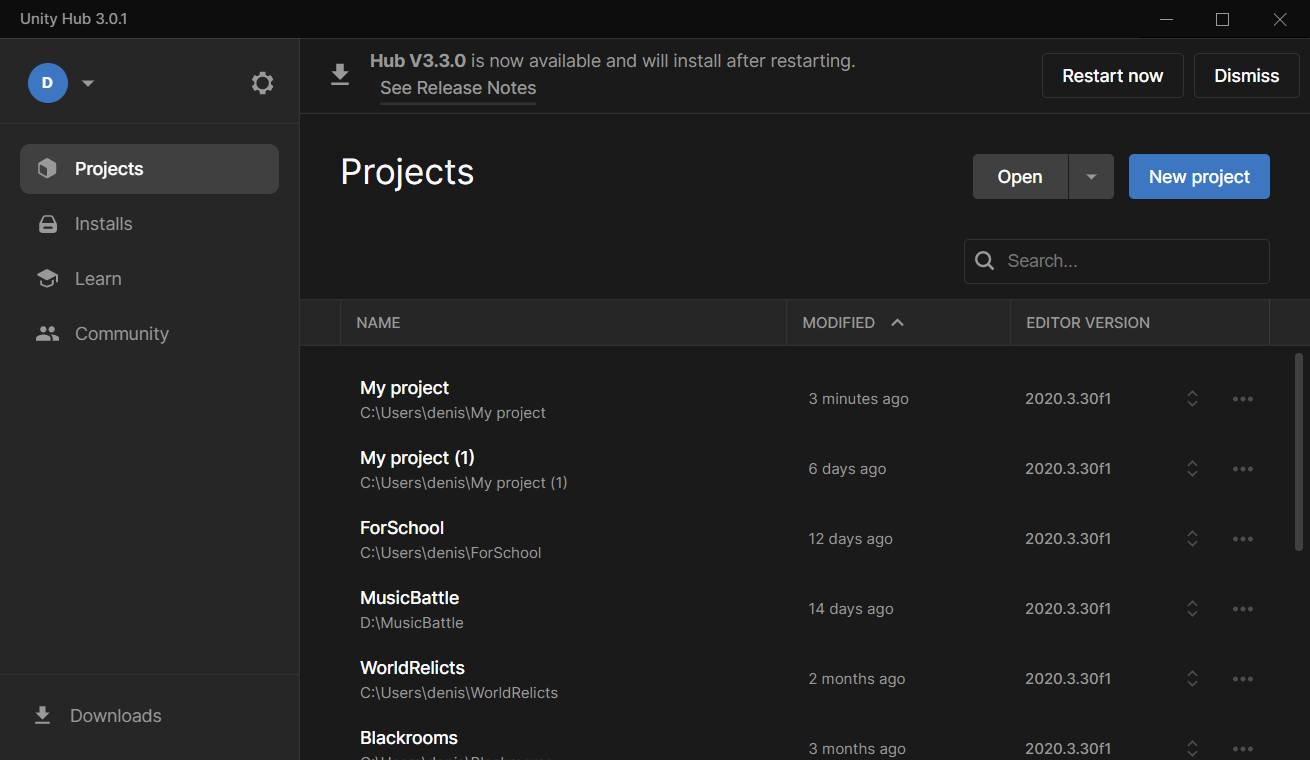
[Словарь 54](#_Toc117858697)

# Блок.1. Концепция 2d игр

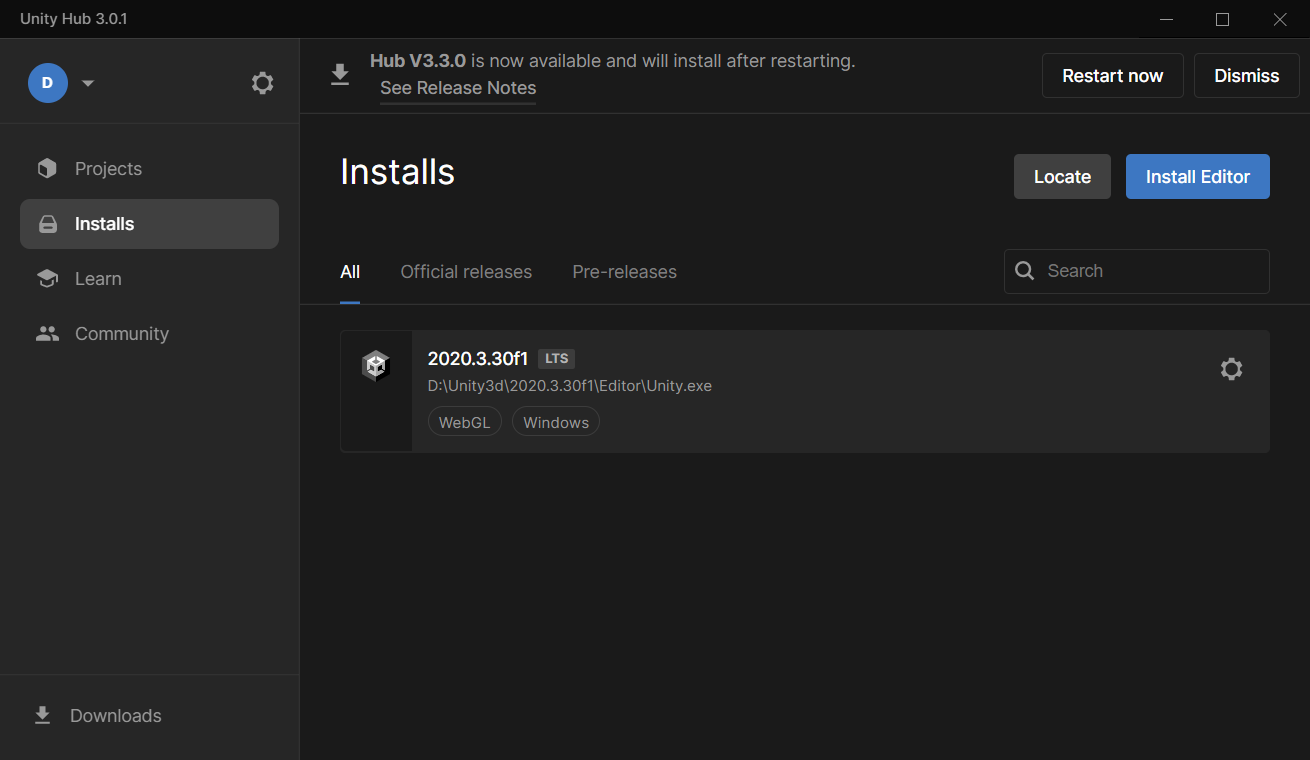
## **Настройка окружения. Сцена. GameObject.**

### Настройка окружения.

Откройте Unity Hub.

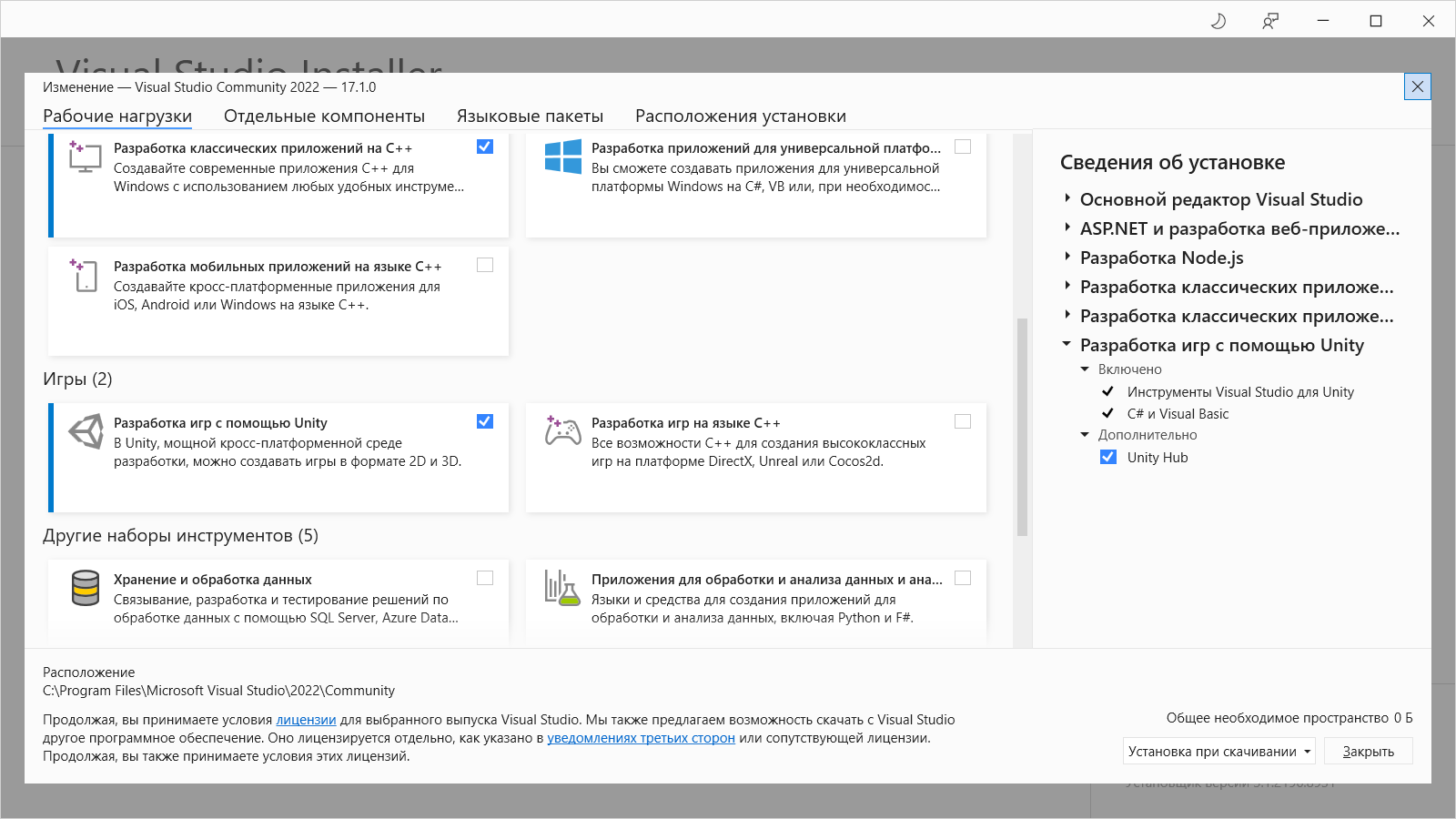


Это окно управления проектами. Здесь вы можете создавать, удалять и настраивать свои проекты. Вкладка «Installs» отвечает за установку самого Unity.



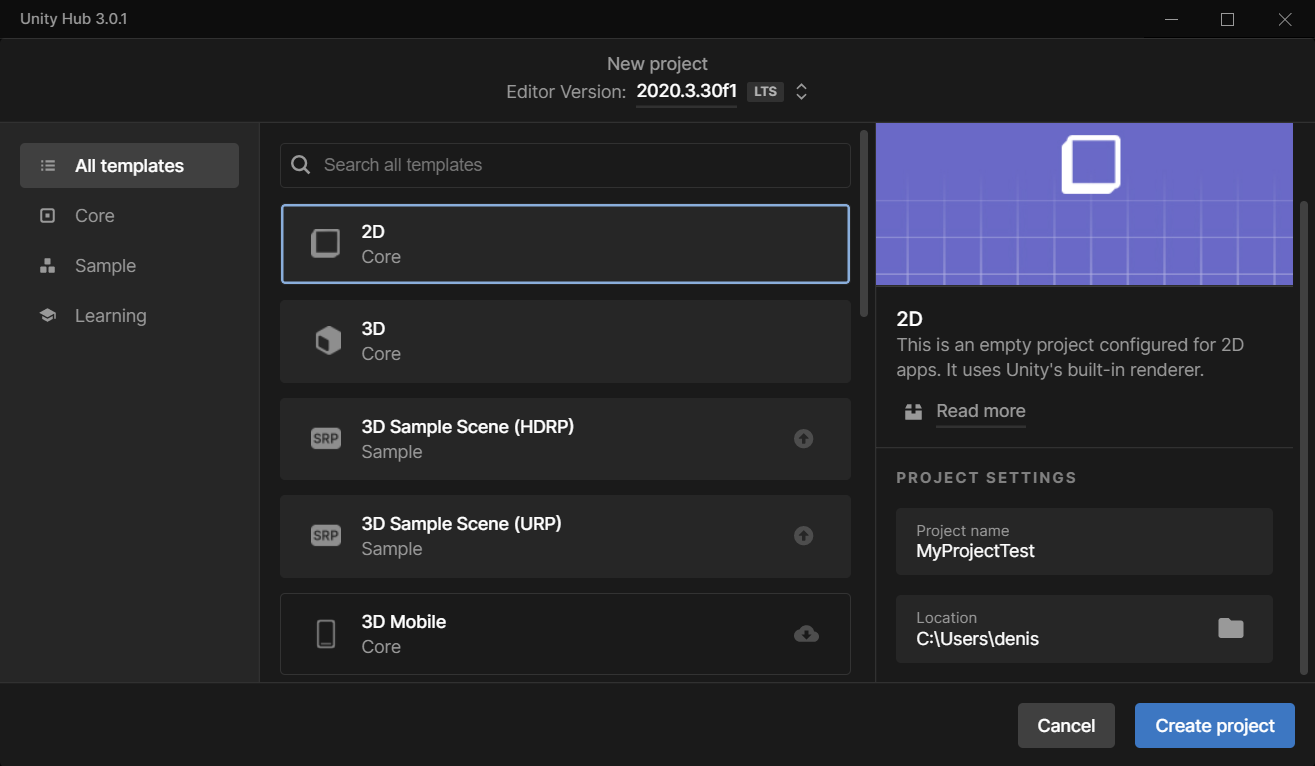
Менеджер версий. «2020.3.30f1» - Версия Unity.

Перейдём к настройке Visual Studio 2019 (2022). Если у вас его нет, установите его или найдите аналоги. Откройте Visual Studio Installer. «Изменить». Установите компоненты необходимые для Unity.



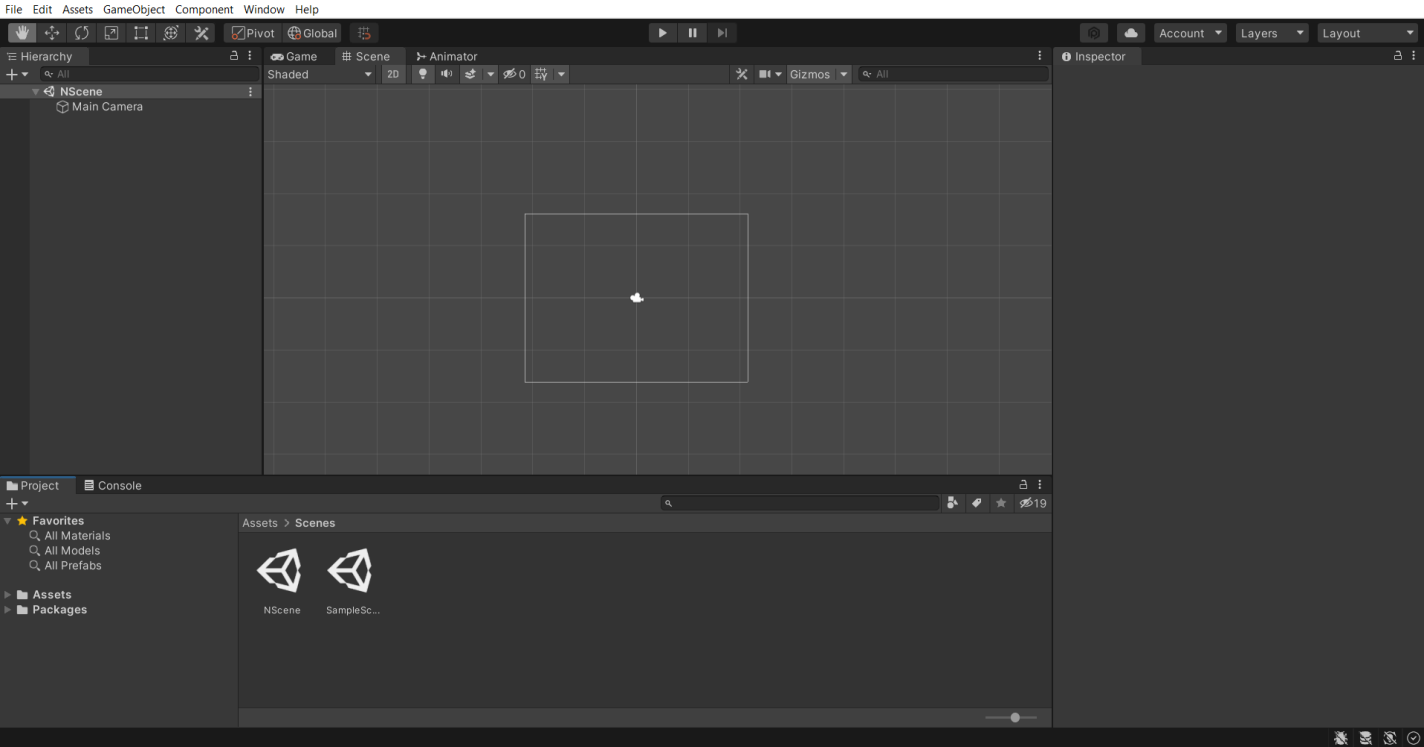
Выберете нужные компоненты и нажмите скачать и установить.

После установки, вернёмся во вкладку «Projects» и создадим проект. Выберите 2d проект, назовите его и нажмите создать.

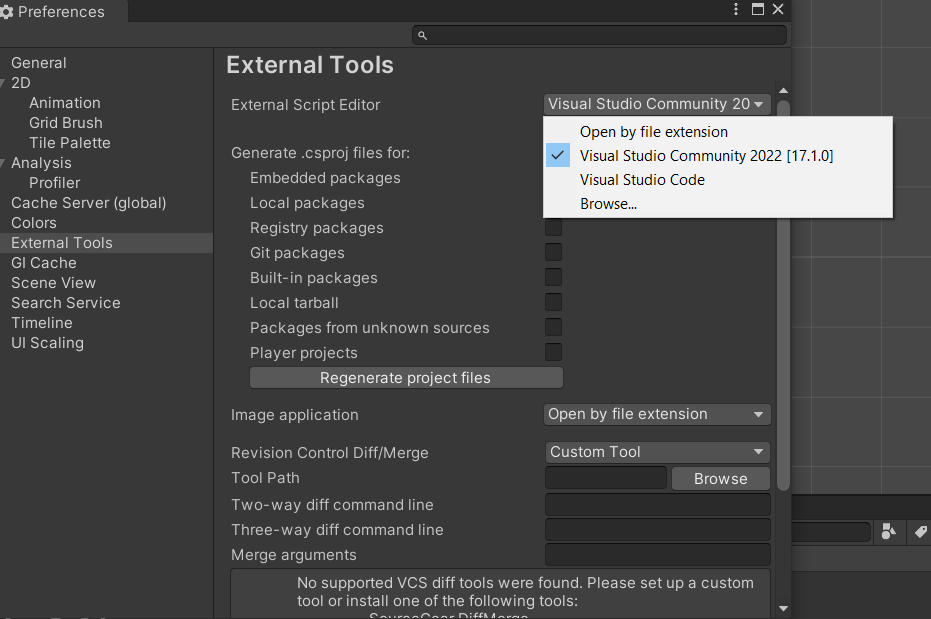


### Настройка Unity3d

После создания проекта, откроется окно. Данное окно является редактором Unity.

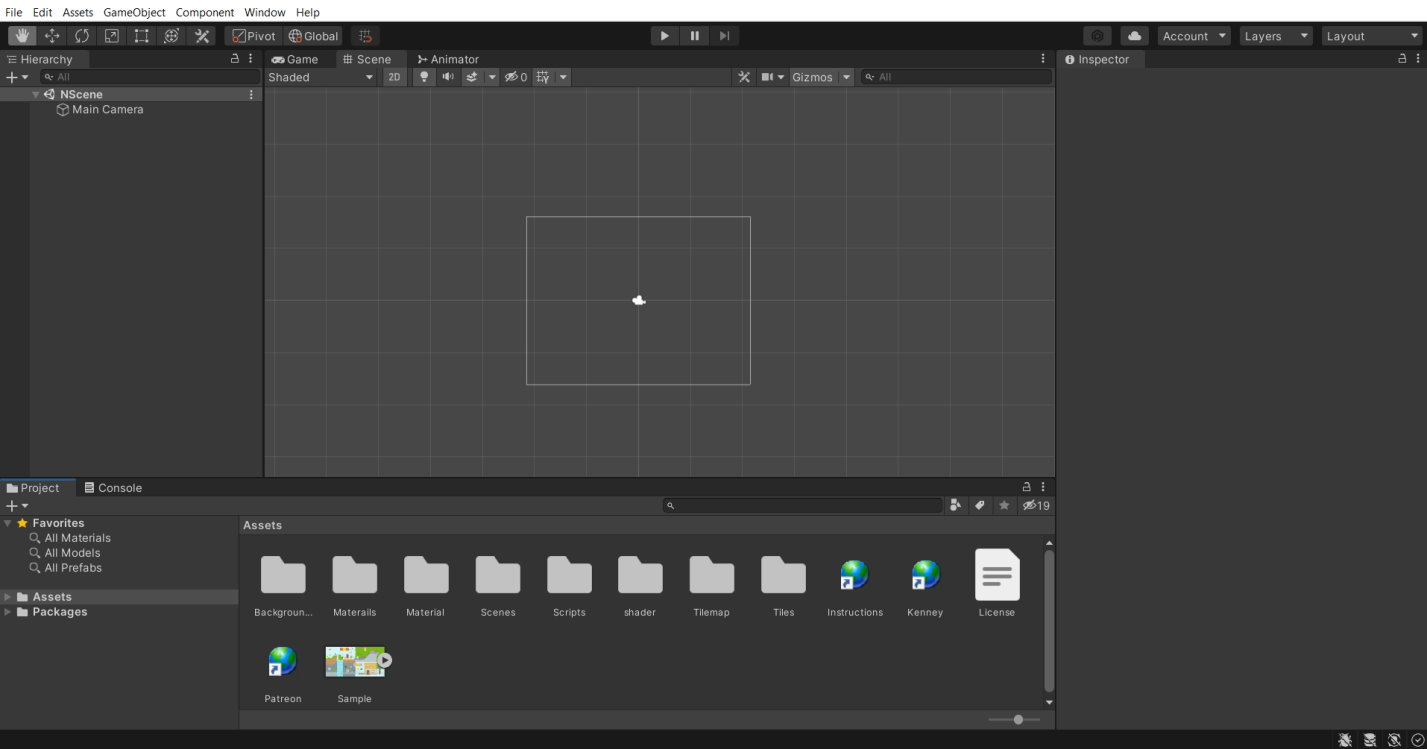


Здесь мы можем уже начать создавать игры. Но для начала немного настроем Unity для будущей работы с кодом. Перейдём по «Edit» -> «Preference» -> «External Tools» -> «External Script Editor». Выберем свой внешний инструмент для работы с кодом в Unity.



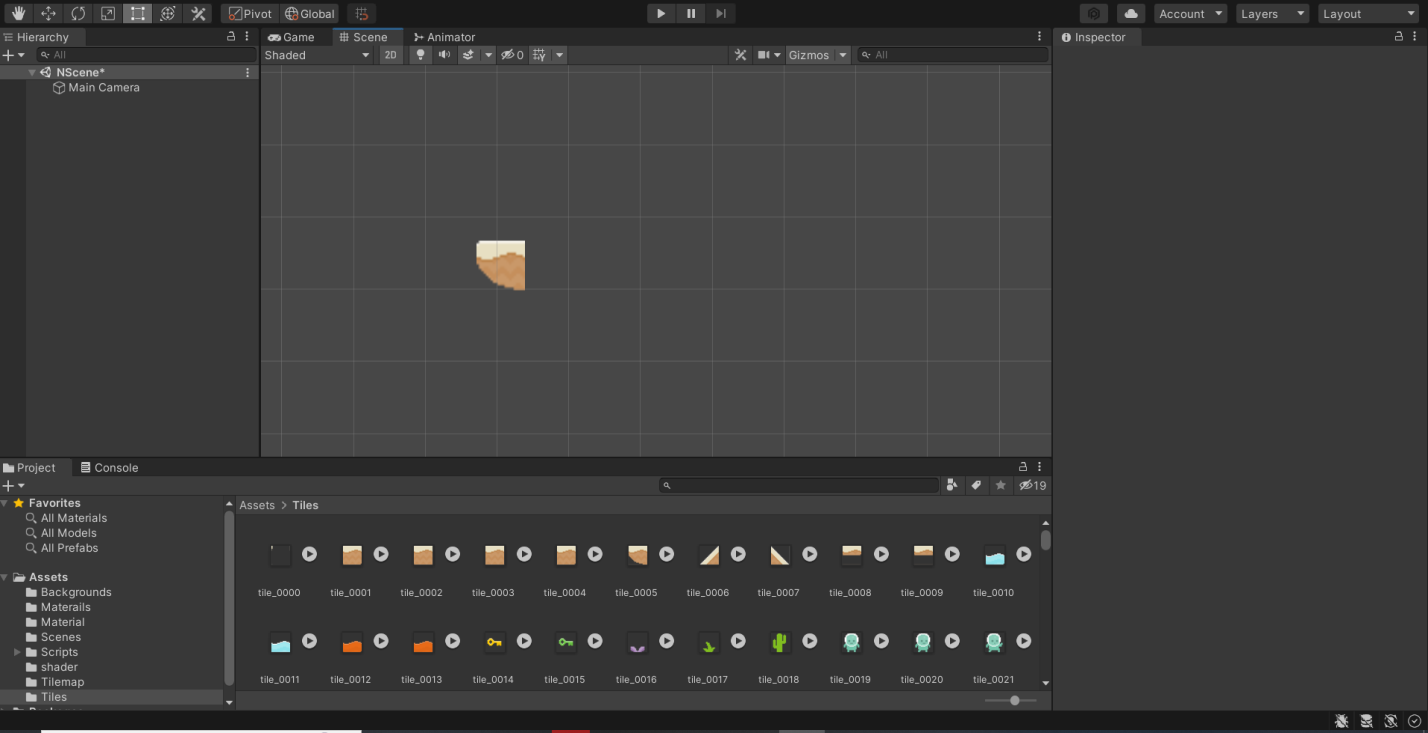
### Настройка проекта.

Для начала скачаем интернет пакет с набором ресурсов. Ссылка: «<https://www.kenney.nl/assets/platformer-art-pixel-redux>». Скачиваем ресурс. Распаковываем. Копируем все содержимое в папку Assets. Для этого достаточно выделить скачанные файлы и перетащить их в окно проекта. Получается результат.

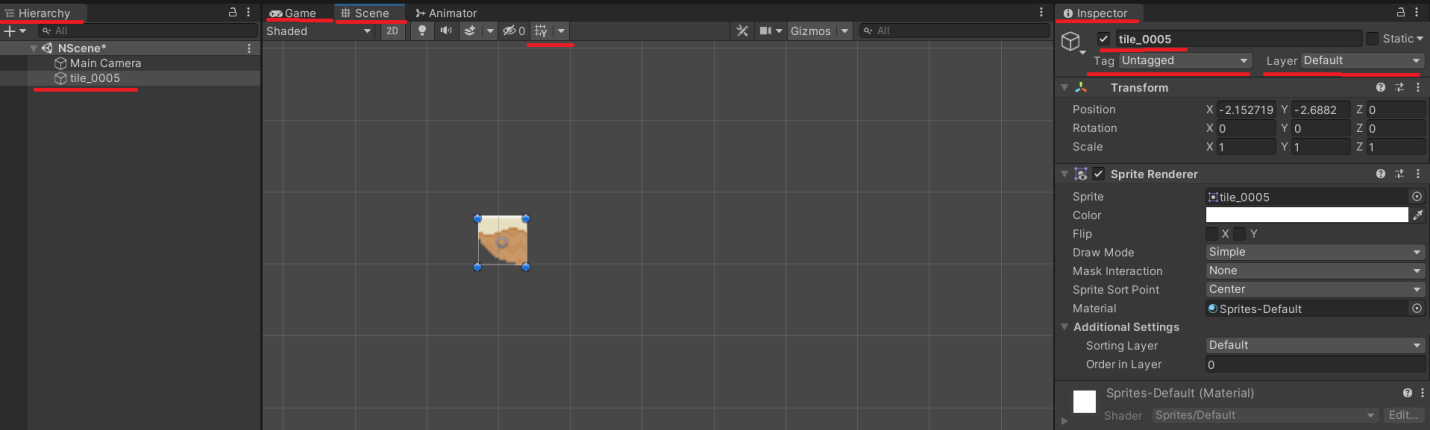


### Редактируем сцену. GameObjcet.

Перетащим из папки «Tiles» любую из текстур на сцену редактора Unity.



Результат, это создание нового объекта на сцене. Мы создали объект «GameObject». Рассмотрим данный компонент.



Рассмотрим все выделенные объекты.

|  |  |
| --- | --- |
| Hierarchy | Иерархия – показывает все объекты на текущей сцене, с учётом их положения относительно друг друга. |
| Title\_0005 | Название нашего нового объекта. |
| Game | Режим игры |
| Scene | Режим камеры |
|  | Отображение сетки |
| Inspector | Инспектор – для отображения всех свойств компонентов объекта. |
| Tag | Тэг объекта. Используется в коде. |
| Layer | Слой, в котором находится объект. Используется для настройки камеры. |
| AddComponent | Добавляет компоненты на объект. |

Добавим на наш объект несколько компонентов. Нажмите AddComponent. Добавим Polygon Collider 2D на наш объект.

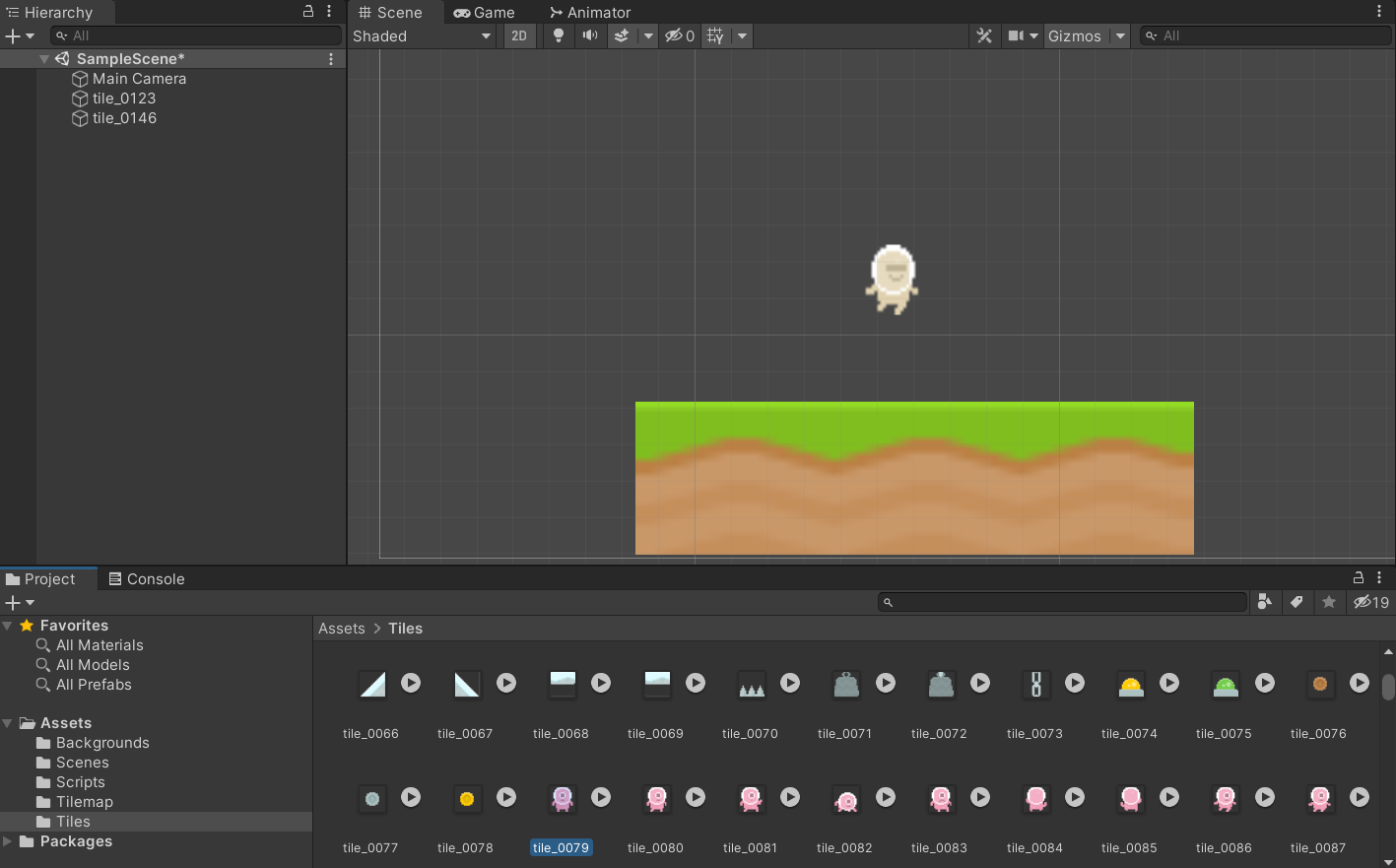


|  |
| --- |
| Задание.1  Добавьте ещё несколько объектов на сцену. |

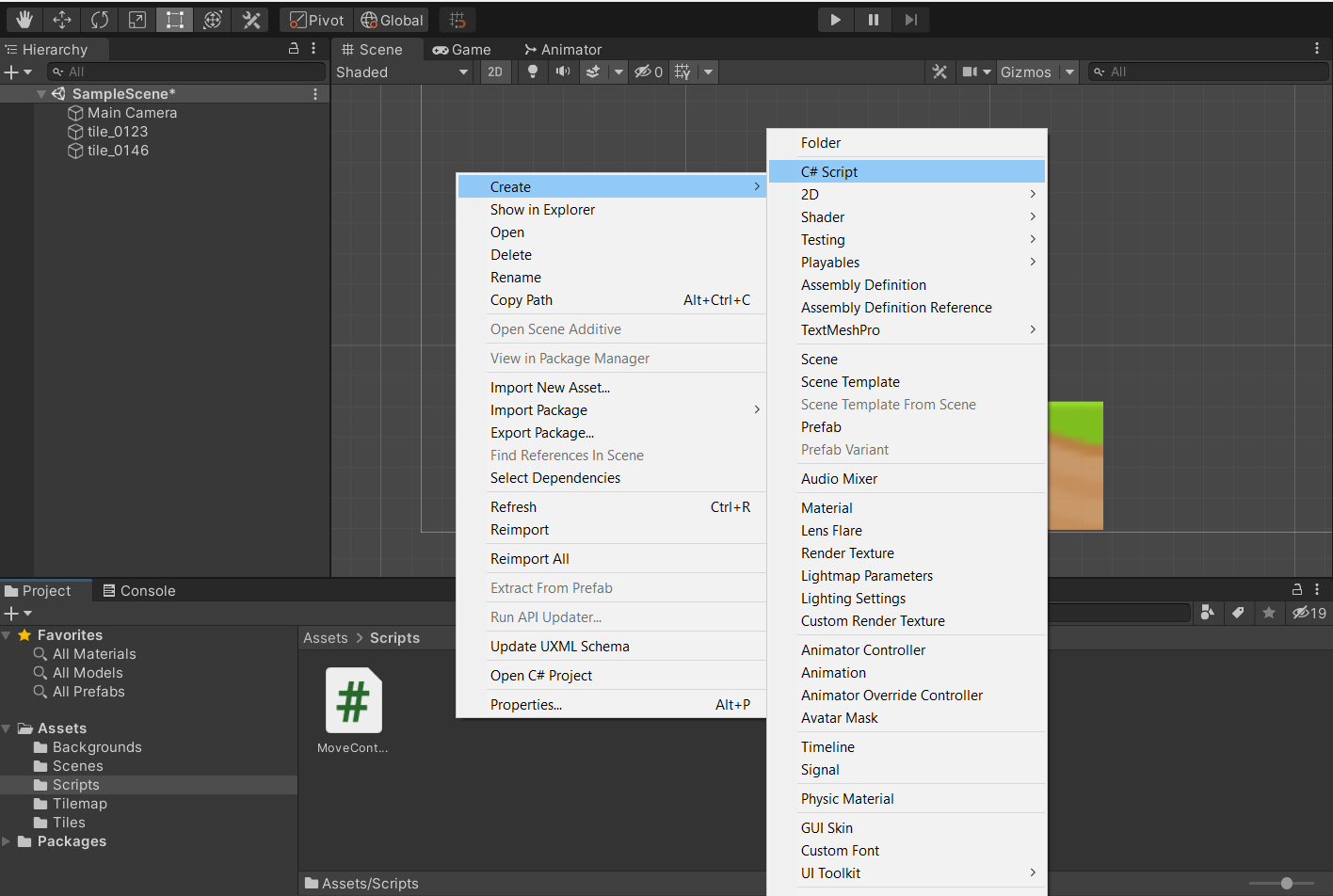
## **Скрипт управление персонажем** с помощью Rigidbody2d

### Подготовка персонажа

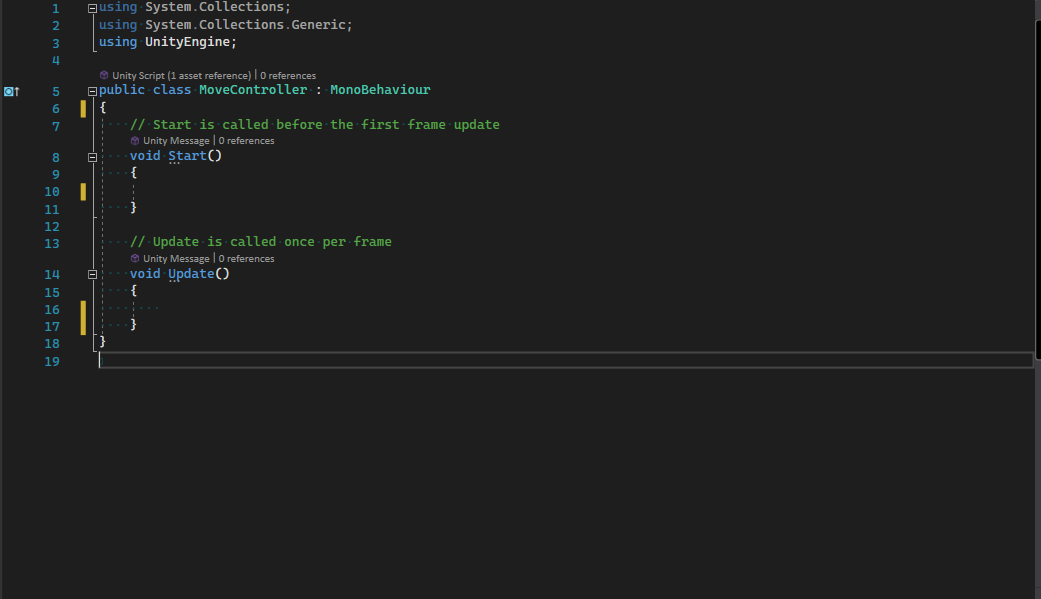
После того, как мы на прошедшем уроке создали простую сцену, можно приступать к созданию персонажа. Первым делом добавим его. Перетащим понравившегося нам персонажа на сцену.



Приступим к созданию нашей первой микропрограммы. Для начала в папке Assets создадим папку Scripts. Внутри неё создадим нашу первую микропрограмму и назовём её “MoveController” (контроллер движения).



Откроем наш скрипт с помощью среды программирования «Visual Studio» (для настройки Unity3d, смотрите в раздел дополнительные настройки). Откроется окно.



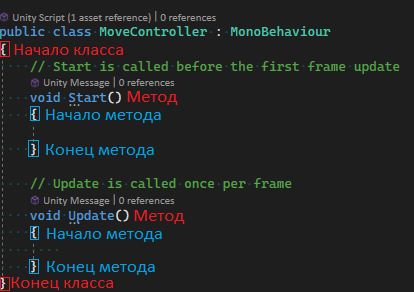
В данном окне, как правило, пишутся все микропрограммы. Логику работы каждого блока мы рассмотри позже. Сейчас же напишем простую микропрограмму, отвечающую за прыжок нашего игрока.



Давайте немного рассмотрим строение класса. Наш класс называется «MoveController». Первое слово «public» - это модификатор доступа, что означает открытый класс. Второе слово «class». С помощью данного слова компьютер понимает, что он работает с классом, а не с чем-то другим. Третие слово «MonoBehaviour» - это класс с внутренними инструментами Unity3d. Когда мы пишем конструкцию class1 : class2, мы передаём параметры из класса 2 в класс 1.

|  |
| --- |
| Задание.1 |
| Удалите слово «MonoBehaviour» и символ «:» и посмотрите, что будет. После верните обратно и сделайте выводы. |

Границы класса определяются с помощью скобочек «{» и «}».



### Первые строчки кода

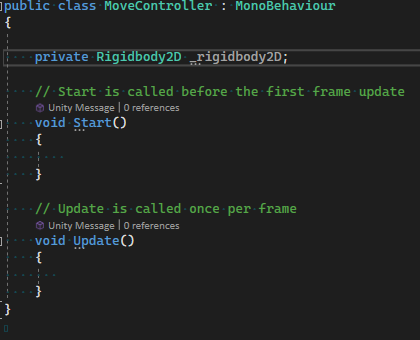
Понимая принципы строения класса, начнём программировать. Мы в рамках данного урока, будем использовать компонент «Rigidbody2D». Он отвечает за физику объекта в пространстве. Для того, что начать работу с ним, давайте объявим его в коде, в качестве поля. Для этого, достаточно написать следующую строку внутри нашего класса, не трогая методы.

|  |
| --- |
| private Rigidbody2D \_rigidbody2D; |

Первое слово «private» - это модификатор доступа. Данное слово говорит о том, будет ли видно наше поле в других классах. Второе слово «Rigidbody2D» - это нужный нам компонент, который мы хотим в дальнейшем использовать. Третье слово «\_rigidbody2D» - это имя нашего поля (вы можете задавать его произвольно).

|  |
| --- |
| Задание.2. |
| Замените «private» у вашего поля на «public» и оцените разницу в редакторе Unity3d. Измените имя вашего поля. Что изменилось? Замените второе слово на любое другое из списка, что предоставляет вам Visual Studia. |

В результате всех действий получается код:



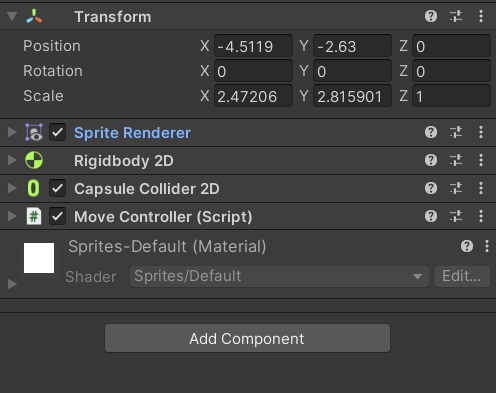
Как видно на картинке, мы создали поле «\_rigidbody2D» внутри класса. Методы остались не тронутыми. По правилам программирования поля должны располагаться выше всех методов и функций.

|  |
| --- |
| Пояснение  Мы создали пустое поле «\_rigidbody2D», что по-другому можно обозвать «ящиком». Почему ящик? Потому что мало создать поле (его определить), надо его ещё и заполнить. Тут всё просто. Давайте вспомним простой стакан. У стакана два состояния: в нём что-то есть, и в нём ничего нет. В нашем случае стакан не простой и всегда при создании является пустым. В нём может лежать только то, что вы указали вторым словом после модификатора доступа. |

Заполним поле объектом, что должен быть определён на нашем персонаже. Для этого внутри метода *void Start()* (между символами «{» и «}») напишите код.

|  |
| --- |
| \_rigidbody2D = GetComponent<Rigidbody2D>(); |

«GetComponent<…>» получает любой компонент, что вы укажете между символами <…> на персонаже, на котором висит скрипт. Для этого вернитесь в Unity3d и добавьте персонажу три компонента: «Rigidbody2D», «CapsuleCollider2d» (или композитный колидер) и добавьте наш «MoveController» (если его нет в списке, то вернитесь в код и сохраните его). Результат приведён на картинке ниже.



Мы записали «Rigidbody2D» в наше поле в коде. Теперь оно не является пустым, и мы можем продолжать работу над персонажем.

|  |
| --- |
| Задание.3. |
| Найдите способ определить - пустое ли поле?  Подсказка: поменяйте модификатор доступа поля c private на public |

Подготовка к написанию кода для движения совершена. Приступим к созданию нашей полноценной микропрограммы.

### Учим персонажа прыгать и ходить.

Внутри метода void Update() напишите следующую конструкцию.

|  |
| --- |
| if(Input.GetButtonDown("Jump"))  {  \_rigidbody2D.AddForce(Vector2.up \* 4f, ForceMode2D.Impulse);  } |

Input – статичный класс для работы с периферийными устройствами. GetButtonDown("Jump") – функция, получающая событие нажатия кнопки прыжка. \_rigidbody2D – имя поля, что мы задавали ранее и заполняли.

AddForce(Vector2.up \* 4f, ForceMode2D.Impulse) – метод, который выполняет движение. Vector2.up - куда направлено двжиение, 4f - скорость, ForceMode2D.Impulse - характеристика движения.

Сохраним проект и перейдём в Unity3d. Запустите игру и посмотрите результат. Наш персонаж научился прыгать! Правда он будет это делать бесконечно, при каждом нажатии на кнопку прыжка.

|  |
| --- |
| Задание.4. |
| Поиграйте с аргументами в функциях и методах. Оцените результат? (измените скорость, характеристику, вектор направления и прочее) |

Теперь научим нашего персонажа ходить. Напишите следующий код внутри метода void Update() ниже предыдущей конструкции.

|  |
| --- |
| var delta = Input.GetAxis("Horizontal");  if(delta != 0)  {  \_rigidbody2D.AddForce(new Vector2(delta, 0) \* 2, ForceMode2D.Impulse);  } |

Рассмотрим подробнее, что он делает. Мы снова используем класс Input для получения функции события нажатия клавиш «лево» и «право» с помощью функции GetAxis("Horizontal"). Данная функция возвращает направление в виде числа «1» (вправо по x) или «-1» (влево по x). Передадим это число в условие. Если число равняется нулю, то пользователь ничего не зажимал. Иначе мы передаём это число на новый вектор, который будет совершать действие по оси X. Так же его умножим на скорость движения «2». Режим прикладываемой силы, укажем «Impulse».

Сохраните проект и посмотрите результат в Unity3d. Наш персонаж начал ходить!

### Исправляем неточности.

Теперь самое сложное. Давайте сделаем так, что бы персонаж ходил и прыгал правильно. Для этого будем использовать свойства векторов. Модифицируем сегмент кода.

|  |
| --- |
| Старый код |
| var delta = Input.GetAxis("Horizontal");  if(delta != 0)  {  \_rigidbody2D.AddForce(new Vector2(delta, 0) \* 2, ForceMode2D.Impulse);  } |
| Новый код |
| var delta = Input.GetAxisRaw("Horizontal");  if(delta != 0 && \_rigidbody2D.velocity.x < 5 && \_rigidbody2D.velocity.x > -5)  {  \_rigidbody2D.AddForce(new Vector2(delta, 0) \* 0.05f, ForceMode2D.Impulse);  }  else  {  \_rigidbody2D.velocity = new Vector2(\_rigidbody2D.velocity.x / 1.01f, \_rigidbody2D.velocity.y);  } |

Мы задали ограничение по ускорению «velocity». Оно не может быть больше 5, и быть меньше -5. Поменяли скорость с «4» на «0.05f». Поменяли характеристику движения. Переопределим ускорение, что бы оно затухало по оси «x». Для этого создаём новый вектор и передаём туда модифицированные старые компоненты. Число «1.01f» - это число затухания скорости или более научно будем называть коэффициентом трения. Чем оно выше, тем быстрее персонаж остановится. Чем ближе к 1, тем медленнее.

|  |
| --- |
| Пояснение  Внимание! Если число «1.01» будет строго меньше 1, скорость будет бесконечно возрастать! Это следует из свойств деления. |

Ограничим прыжок. Для этого будем использовать новое поле. Объявите поле «\_isAir» и модифицируете код.

|  |
| --- |
| if(Input.GetButton("Jump") && !\_isAir)  {  \_rigidbody2D.AddForce(Vector2.up \* 2f, ForceMode2D.Impulse);  \_isAir = true;  } |

Если прыжок был совершён, мы меняем состояние нашего поля на «true», тем самым блокируя его дальнейший вызов, пока наш персонаж находится в воздухе.

Теперь добавим событийный метод void OnCollisionStay2D внутри нашего класса (не внутри методов!).

|  |
| --- |
| private void OnCollisionStay2D(Collision2D collision)  {  \_isAir = false;  } |

Данный метод вызывается каждый раз, когда произошло столкновение с любой поверхностью. При столкновении, меняем поле на состояние ложь. Сохраните код и проверьте работу.

Если оставить всё как есть, то наш персонаж при столкновении с любым объектом, что имеет компонент collider2d, будет сбрасываться поле прыжка. Это значит, что мы получили частичные возможности человека паука – прыгать по стенам.

Для того, что бы избавиться от данной проблемы, измените блок кода на следующий сегмент. Оцените и опишите результат.

|  |
| --- |
| private void OnCollisionStay2D(Collision2D collision)  {  if(collision.contacts[0].normal == Vector2.up)  {  \_isAir = false;  }  }  private void OnCollisionExit2D(Collision2D collision)  {  \_isAir = true;  } |

На этом урок по написанию базовых механик подошёл к концу. Дальше будут представлены задания для самостоятельного изучения.

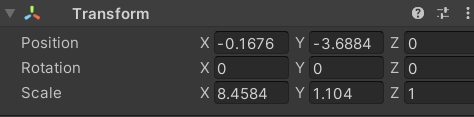


|  |
| --- |
| Задание. 5. |
| Для каждого числа в коде, создайте своё поле с открытым доступом. |
| Придумайте способ, убрать подёргивания для горизонтального движения при столкновении с преградой (Можно использовать аналог с прыжком). |

## Скрипт управления движением. Компонент Transform.

### Компонент Transform.

Компонент Transform отвечает за положение объектов в пространстве. Положение задаётся с помощью трех векторов:



* + - Position – позиция объекта в пространстве.
    - Rotation – поворот объекта в пространстве.
    - Scale – размер объекта в пространстве.

Разработчик может самостоятельно менять эти параметры, тем самым персонаж на сцене будет перемещаться в координаты, что вы указали. Менять размеры и вертеться.

Как мы поняли из прошлого урока, мы можем получать эти свойства в коде и управлять ими. К примеру, ограничивать движение игрока или другого объекта. Или поменять написать управление движениями персонажа через данный компонент, комбинируя физику с компонентом Rigidbody2d. Можно задать строгое уравнение или можно добиться много разных интересных механик для вашей игры. Перечислим самый простые механики с примерами реализацией в коде:

|  |  |
| --- | --- |
| Объект | Код в C# |
| Телепорт | transform.position = new Vector(0,0,0) |
| Движение | transform.position += new Vector(1,0,0) \* Time.deltaTime; |

Как видно из таблицы, для телепортации, достаточно поставить знак равенства у свойства позиции объекта на нужную координату. Для создания передвижной платформы, достаточно складывать нужное нам направление с позицией объекта при помощи операции «+=».

В дальнейшем, для работы с компонентами Transform нам будет необходимо узнать больше о векторной математике. Рассмотрим несколько математических операций, которые используются обычно совместно с расчётом движения.

### Математика векторов.

Один из важнейших блоков для программирования движения является математика. Если в уроке «1.2» компонент Rigidbody2d сам выполнял вычисления интерполяции и передвижения, то при использовании Transform мы должны позаботиться лично обо всех действиях. Другими словами, Rigidbody2d делал всё за нас.

Рассмотрим самые частые функции, применяемые над векторами:

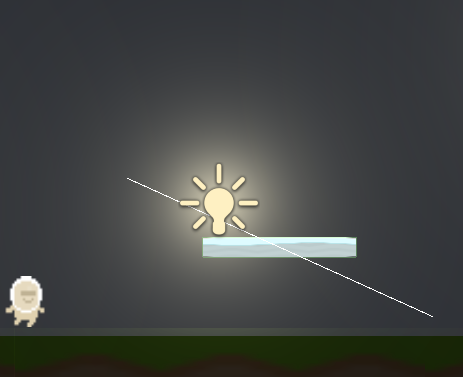
|  |  |
| --- | --- |
| Функция | Что делает |
| Vector2.Angle(v1,v2) | Считает угол между двумя векторами. |
| Vector2.Lerp(v1,v2,1/distance) | Вычисляет линейную интерполяцию. |
| Vector2.Slerp(v1,v2,1/distance) | Вычисляет сферическую интерполяцию. |
| Vector2.Distance(v1,v2) | Вычисляет расстояние между двумя точками |

Угол между векторами – это число, выражаемое в градусах, что является математическим углом между двумя прямыми.

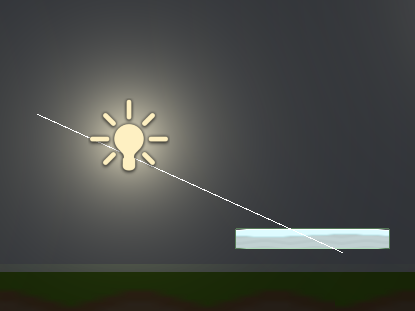


Угол обычно рассчитывается, что бы учитывать параметры скольжения, затухания движения по наклонностям. Как видим, компонент Rigidbody2d автоматически рассчитывает направление по наклонности.

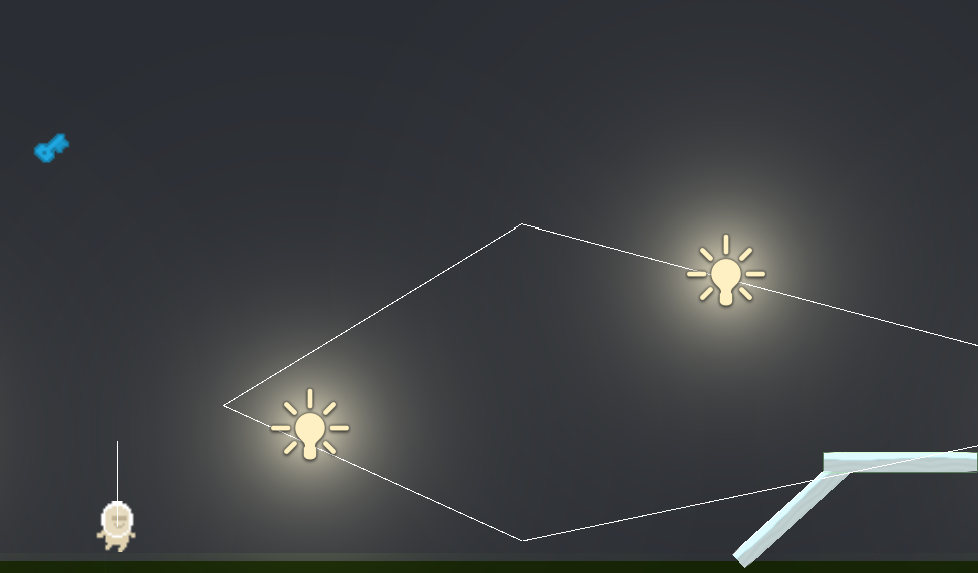
Линейная интерполяция – это операция поиска точки между двумя точками на определённой фиксированной прямой. Пример интерполяции Vector2.Lerp(v1,v2,0,5f).



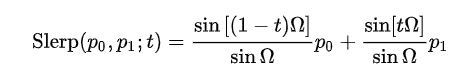
Как видно на картинке, интерполяция с коэффициентом ½ делит линию пополам. Пример интерполяции Vector2.Lerp(v1,v2, 0,1f).



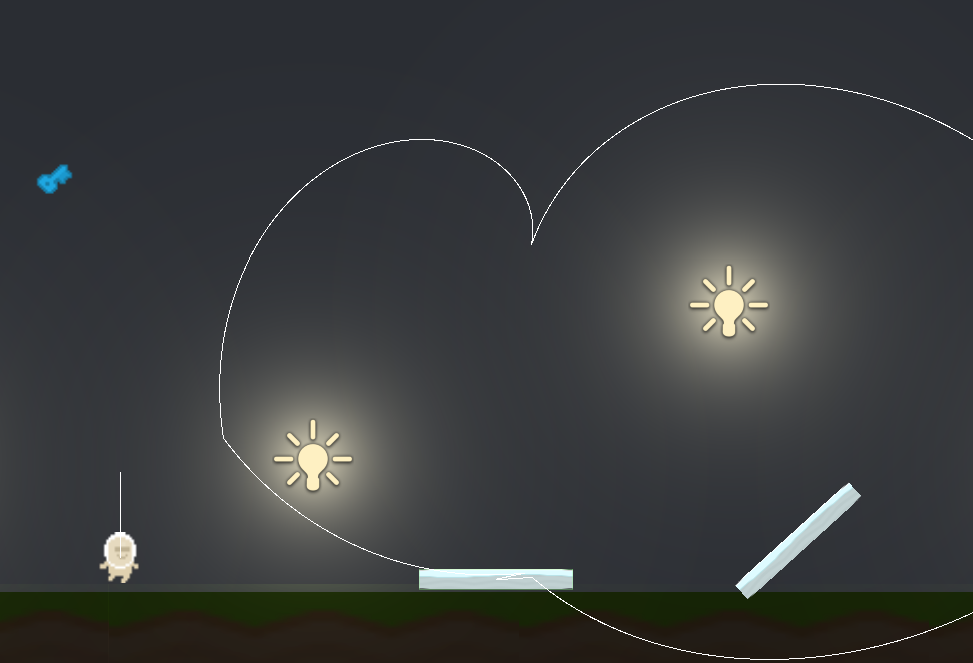
Линия была разделена на 10 частей. Наш объект передвинулся на 1/10 часть от общего расстояния. Тогда полноценная формула будет выглядеть так. Пример интерполяции Vector2.Lerp(v1, v2, 1/distance), где distance – это расстояние от начала пути до его окончания.



Сферическая интерполяция – это интерполяция, имеющая следующую формулу:



Для простоты понимания отобразим её на экране. Как видно, движение стало иметь сферический вид передвижения.



Другими словами интерполяция – это предполагаемое вычисление, поиск точек, которые располагаются между двумя опорными, которое строиться по определённой формуле.

Рассмотрев базовую математику, мы можем перейти к созданию нашей первой передвигающейся платформе.

### Передвигающаяся платформа.

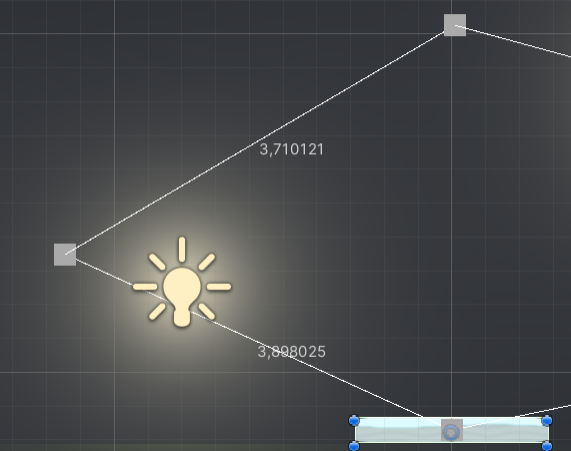
Перейдём в редактор Unity3d и создадим простую 2d платформу. Для данной платформы создадим специальный скрипт, который будет заниматься его передвижением в пространстве. Назовите его, как считаете нужным и перейдите в редактор Visual Studio.

Создадим два поля внутри нашего класса. Первое открытое поле с типом «Transform» мы назовём «Target». Второе поле приватное с типом целого числа назовём «\_pointer».



Для начала разработаем алгоритм, по которому мы будем вычислять движение нашего объекта.

Из прошлой главы, мы узнали, что для метода поиска линейной интерполяции нам нужно получить начальную точку, конечную точку и дистанцию между ними. Конечные точки у нас хранятся в поле «Target». Указатель «\_pointer» указывает на точку из поля «Target». Текущая точка хранится внутри системного свойства «transform.position».



Для начала получим точку, в координаты которой мы будем двигать нашу платформу. Поскольку мы храним наши точки в массиве, будем доставать каждую поочередно с помощь целочисленного указателя «\_pointer».

|  |
| --- |
| var targetPosition = Target[\_pointer]; |

Найдём расстояние между нашей платформой и полученной точкой.

|  |
| --- |
| var distance = Vector2.Distance(transform.position, targetPosition); |

Применим формулу линейной интерполяции.

|  |
| --- |
| transform.position = Vector3.Lerp(transform.position, targetPosition, 1/distance \* Time.fixedDeltaTime); |

В конце, добавим проверку, когда платформа достигает необходимой координаты и дополнительную проверку для сброса указателя до начальных значений.

|  |
| --- |
| if(distance <= 0.1f)  {  \_pointer++;  if(\_pointer >= Target.Length)  {  \_pointer = 0;  }  } |

Сохраним код и проверим его в Unity3d. Результат:



Мы можем добавлять точки или убирать. Платформа будет последовательно двигаться из одной точки в другую.

|  |  |
| --- | --- |
| Дополнительно  Вы теперь можете настраивать последовательность точек. Но, как видите, это будет не очень удобным процессом. Предлагаю добавить метод в ваш код. Он значительно упростит процесс настройки точек передвижения вашей платформы. Можете изучить, как это работает на досуге (требуется директива «using UnityEditor;»).   |  | | --- | | private void OnDrawGizmosSelected()  {  var oldPositin = transform.position;  foreach(var item in Target)  {  Gizmos.DrawLine(oldPositin, item);  var pos = Vector2.Lerp(oldPositin, item, 0.5f);  var dist = Vector2.Distance(oldPositin, item);  oldPositin = item;  Gizmos.DrawCube(item, Vector2.one / 5);  Handles.Label(pos, dist.ToString());  }  } | |

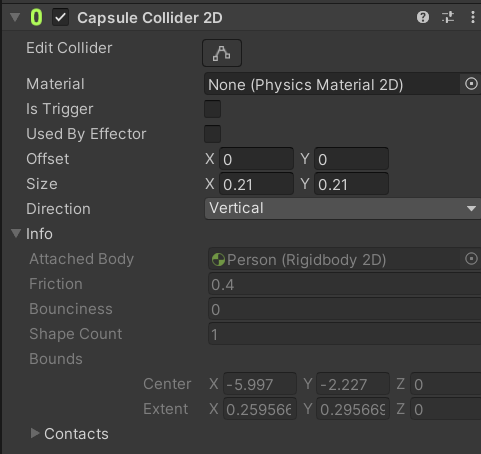
## Столкновения, коллизии и триггеры. Компоненты Collider 2D.

### Базовый компонент «Collider 2D».

Collider – это группа компонентов, которая определяют физическую форму GameObject для физических столкновений. Коллайдер не обязательно должен быть видимый и иметь ту же форму, что и сам объект GameObject. Грубая аппроксимация (примерная структура коллидера) часто более эффективна и неразличима в игровом процессе.



Рассмотрим некоторые свойства коллайдера.



У всех коллайдеров есть «Материал». Он отвечает за физику столкновения вашего персонажа с ним. В качестве характеристик, там выступают «Трение» и «Отскок».

Второе свойство всех коллайдеров, это флаг «Триггера». Он переключает режим работы столкновений в отслеживаемые зоны. Попадая в неё, физический объект может проходить сквозь текстуры, при этом мы можем накладывать разные взаимодействия или совершать какие-либо действия.

Третье свойство флаг использования «Эффекторов». Один из примеров использования «Эффектора», является область принудительного движения, когда игровой объект, словно с помощью конвейера переносят по пространству.

«Offset» - настройка смещения коллайдера.

«Size» - размер коллайдера.

«Auto Tilling» - флаг автоматической корректировки при изменении размеров или структуры спрайта.

Остальные свойства используются для тонкой настройки.

Стоит отметить, что только объекты Rigidbody2d на безкодовом уровне могут взаимодействовать с физикой collider2d. В остальном случае, столкновения работать не будут.

|  |  |  |
| --- | --- | --- |
| Collider | (нет взаимодействия) | Collider |
| Collider | (есть взаимодействие) | Rigidbody |
| Rigidbody | (есть взаимодействие) | Rigidbody |

### Событийные методы столкновения.

Мы ранее уже использовали событийные методы столкновения, а сейчас рассмотри их более детально. Событийные методы - это методы, которые выполняются после совершения какого-либо события. В данном случае, событием является столкновение.

Столкновение можно разделить на три типа действия.

|  |  |
| --- | --- |
| Описание | Метод |
| Когда игрок первый раз касается поверхности | private void OnCollisionEnter2D(Collision2D collision)  {  //Ваш код  } |
| Когда игрок находится на поверхности | private void OnCollisionStay2D(Collision2D collision)  {  //Ваш код  } |
| Когда игрок покинул поверхность (отодвинулся) | private void OnCollisionExit2D(Collision2D collision)  {  //Ваш код  } |

Обрабатывая подобные события, мы можем тщательно настраивать дальнейшее взаимодействие игрока с плоскостью: создать трамплин, метод активации и прочее.



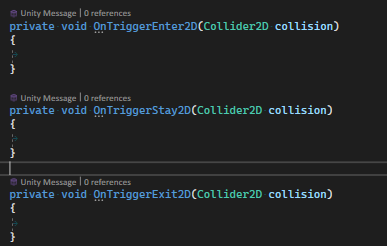
### Триггеры.

Триггеры – это зона, которая что-то делает. Примером триггера, может являться область, которая активирует движение; или область, которая телепортирует игрока и многое другое.

Триггеры, как и области, так же делятся на три типа взаимодействия:

|  |  |
| --- | --- |
| Описание | Метод |
| Когда игрок первый раз попадает в область | private void OnTriggerEnter2D(Collision2D collision)  {  //Ваш код  } |
| Когда игрок находится в области | private void OnTriggerStay2D(Collider2D collision)  {  //Ваш код  } |
| Когда игрок покинул область | private void OnTriggerExit2D(Collider2D collision)  {  //Ваш код  } |

Обрабатывая подобные события, мы можем тщательно настраивать дальнейшее взаимодействие игрока с областью: создать телепорты, метод активации, область респавна, область нанесения урона и многое другое.

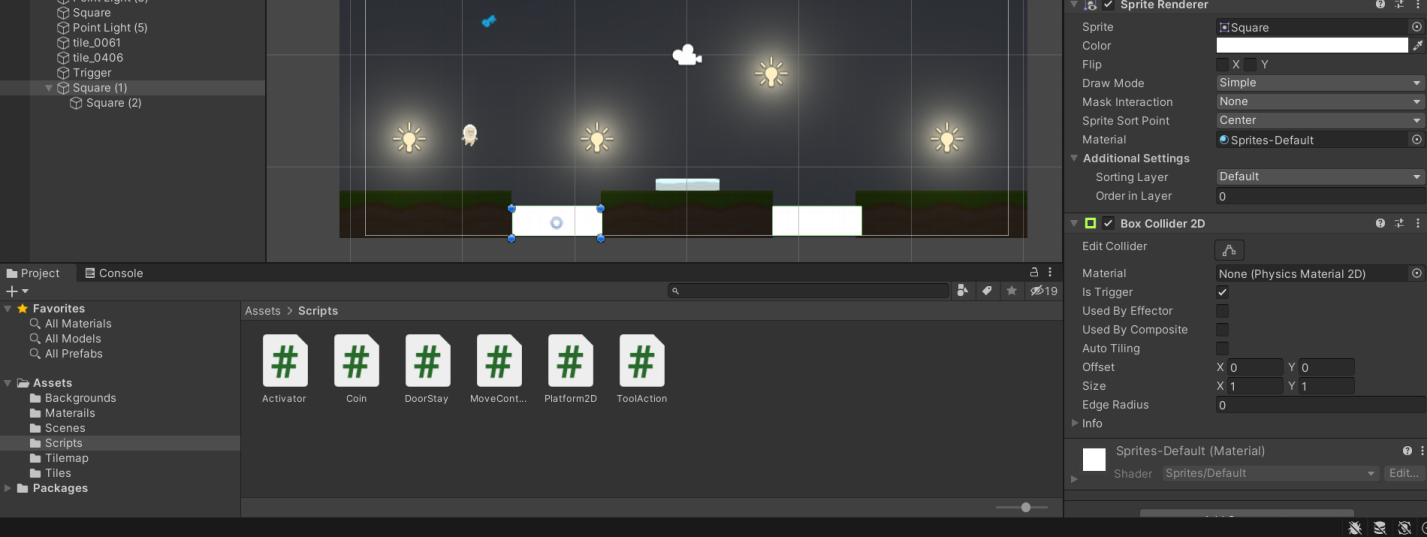


### Создаём простой телепорт.

#### Подготовка

Создадим простой 2d спрайт на сцене. Добавим на него компонент BoxCollider2D (или любой другой). В компоненте BoxCollider2D установим включенным флаг «Trigger».

Расставим на сцене объекты так, что бы было куда падать.



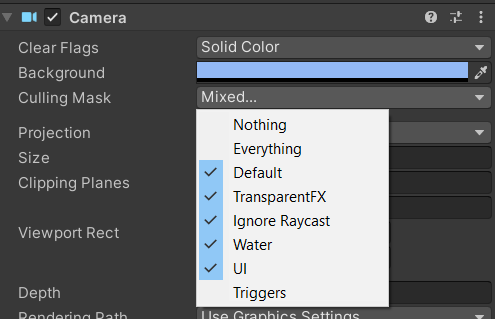
Теперь можно поступить двумя путями. Первый путь - это удалить текстуру с созданных блоков. Второй путь рассмотрим более подробно.

#### Настройка игровых слоёв

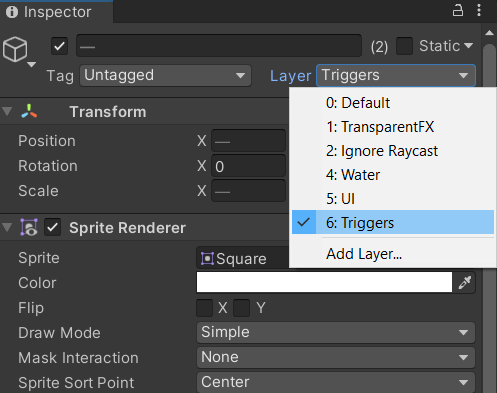
Самый правильный способ это настроить камеру так, что бы в редакторе мы видели блоки, а в игре нет. Так будет удобнее при работе со сценой. Давайте настроем камеру. Рекомендую перейти в настройки камеры и прямо оттуда создать новый слой (Add Layer).

|  |  |
| --- | --- |
|  |  |

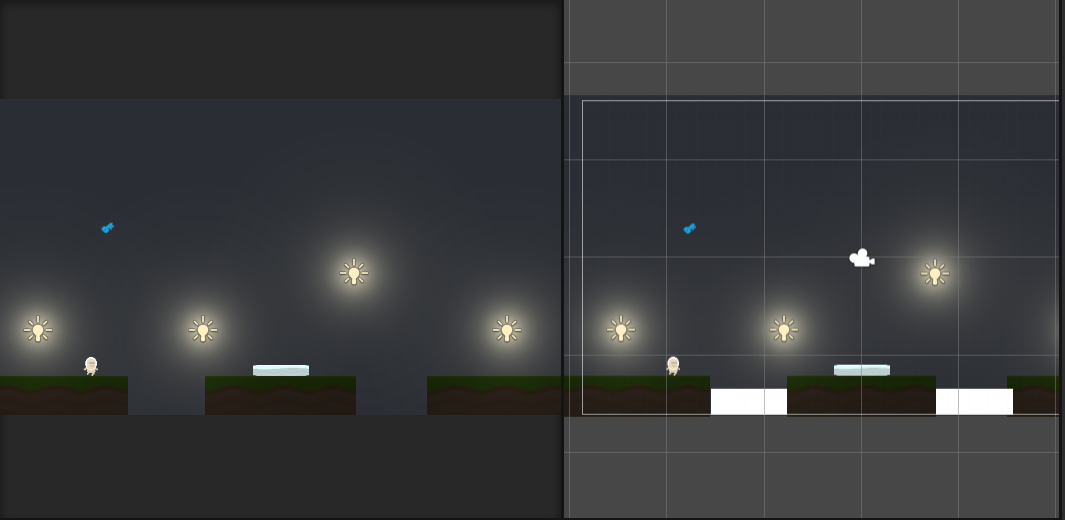
После создания слоя, вернитесь в настройку камеры и откройте список «Culling Mask».



Снимите галочку с объекта «Triggers». После вернитесь к недавно созданным спрайтам, и установите «Layer», как показано на картинке.



Если мы перейдём сейчас в сцену, то получим следующий результат



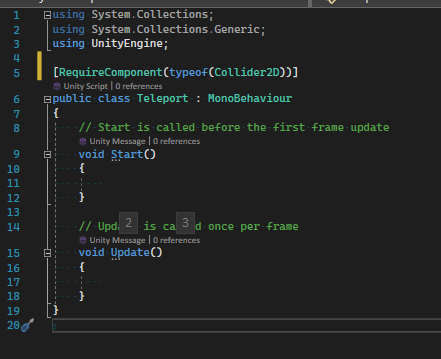
Как видно на картинке, в редакторе блоки есть. В режиме игры, блоков нет. Теперь давайте пометим наши блоки, распознавательной текстурой, что бы все триггеры выделялись от остальных объектов.



#### Телепорт

Пусть у нас будет два класса. Первый будет отвечать за телепорт. Второй будет отвечать за место телепортации. (Можно поступить проще, просто укажите точку телепортации). Создадим новый код в Unity для триггера телепортации. Назовите его соответствующе.

Давайте потребуем от объекта, обязательное наличие компонента Collider2D с помощью строчки «[RequireComponent(typeof(Collider2D))]».



Определим поля класса.

|  |
| --- |
| private Rigidbody2D[] \_listenerPlayer;// Все объекты с физикой  [SerializeField] //Атрибут сериализации поля, что бы он был виден в инспекторе Unity  private Vector2 \_target; //Точка телепорта |

Зададим инициализацию поля «listenerPlayer» в методе void Start().

|  |
| --- |
| void Start()  {  //Получаем все объекты на сцене, что имеют Rigidbody2D.  \_listenerPlayer = FindObjectsOfType<Rigidbody2D>();  } |

Используем наш событийный метод, который будет срабатывать, если объект первый раз попал в зону.

|  |
| --- |
| private void OnTriggerEnter2D(Collider2D collision)  {  //Перебираем всех игроков  foreach(var player in \_listenerPlayer)  {  //Проверям есть ли полученный объект в \_listenerPlayer  if(player.transform == collision.transform)  {  //Меняем координаты  player.transform.position = \_target;  }  }  } |

Проверьте результат в Unity. Не забудьте перенести код на ваш объект телепортации и настроить его.

Создадим второй код в Unity и назовём его «TeleportDestination». Перейдём вовнутрь и отредактируем код.

|  |
| --- |
| public class TeleportDestination : MonoBehaviour  {  private void Start()  {  //Автоматически меняем его на слой Triggers  gameObject.layer = LayerMask.NameToLayer("Triggers");  }  } |

Так будет проще настраивать телепортацию в точку назначения. Изменим поле в коде класса «Teleport». Заменим «Vector2» на «TeleportDestination». Поправим код телепорта. Будем брать позицию одного объекта, и передавать другому.

|  |
| --- |
| player.transform.position = \_target.transform.position; |

Так же добавим дополнительную обрисовку связи.

|  |
| --- |
| private void OnDrawGizmos()  {  //Проверяем не пустой ли объект  if(\_target != null)  {  //Выбираем зеленый цвет  Gizmos.color = Color.green;  //Рисуем линию от нашей области до точки телепортации.  Gizmos.DrawLine(transform.position, \_target.transform.position);  }  else  {  //Создаём прозрачный красный цвет  Gizmos.color = new Color(1,0,0,0.1f);  //Рисуем круг  Gizmos.DrawSphere(transform.position, transform.lossyScale.x / 2);  }  } |

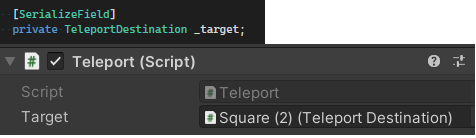
Сохраните оба класса и перейдите в Unity. Если телепорт не определён, то мы увидим, как рядом с нашими областями рисуются красные прозрачные круги.



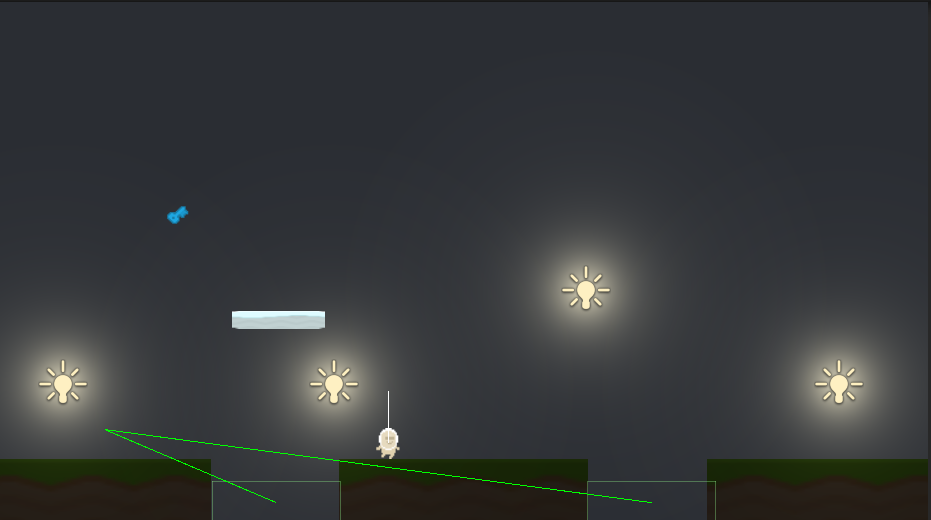
Если телепорт определён, то мы увидим линии связи.



Что бы определить точку телепорт нужно задать, куда вы будете телепортироваться. Поле в коде «\_target» равно свойству компонента «Target». Можете его переназвать или создать ещё и посмотрите результат.



Если вы ещё не осознали ранее, то все публичные поля или поля с атрибутом серилизации будут видны в инспекторе юнити. Запустим игру и посмотрим результат. Если вам мешают линии, отключите «Gizmos».



### Создаём простой батут.

Для создания простого батута, будем использовать событийный метод столкновения первого прикосновения к поверхности. Но перед этим создадим способ инициализации посложнее, через рефлексию c#.

Создадим новый класс и назовём его «ToolAction». Данный класс будет играть роль действий в игре. В нём мы будет реализовать простые механики и воздействия на объекты.

Зададим поле.

|  |
| --- |
| private Rigidbody2D \_rigidbody2D |

Напишем его инициализацию, через рефлексию. Внимание, не злоупотреблять и использовать только в тестовых целях. Рефлексия очень дорогая операция и может серьёзно сказаться на производительности и качестве вашей игры. Зато, все компоненты заполняться автоматически.

|  |
| --- |
| private void Start()  {  //Получаем все возможные поля у данного класса.  var fields = this.GetType().GetFields(System.Reflection.BindingFlags.NonPublic |  System.Reflection.BindingFlags.Instance | System.Reflection.BindingFlags.Public);  //Ищем для этих полей подходящий для них компонент.  foreach (var field in fields)  {  //Устанавливаем в поле значение полученного компонента  field.SetValue(this, GetComponent(field.FieldType));  }  } |

Теперь в качестве инструментов определим, пока один метод.

|  |
| --- |
| public void TouchForce(Vector2 vector)  {  Rigidbody2D.AddForce(vector, ForceMode2D.Impulse);  } |

Данный метод будет толкать персонажа в сторону направления вектора. Будем его вызывать из другого класса, который мы скоро создадим.

Сохраним код. Перейдём в Unity. Добавим созданный класс на нашего персонажа. После, создадим ещё один класс «PushArea». Перейдём в него. Определим поля.

|  |
| --- |
| private ToolAction[] \_listenerPlayer; //Прослушываемые игроки  [SerializeField]  private Vector2 \_direct; //Направление толчка |

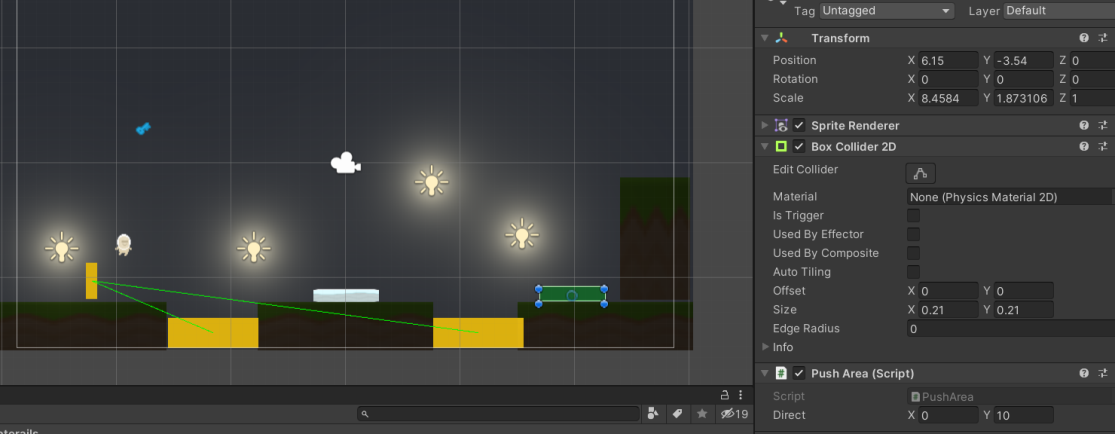
Напишем инициализацию.

|  |
| --- |
| void Start()  {  //Получаем все объекты на сцене, что имеют ToolAction.  \_listenerPlayer = FindObjectsOfType<ToolAction>();  } |

Напишем обработку столкновения.

|  |
| --- |
| private void OnCollisionEnter2D(Collision2D collision)  {  //Перебираем всех игроков  foreach(var player in \_listenerPlayer)  {  //Проверям есть ли полученный объект есть в \_listenerPlayer  if(player.transform == collision.transform)  {  //Вызываем метод TocuhForce у нашего класса ToolAction.  player.TouchForce(\_direct);  }  }  } |

В первую очередь мы создали инструмент – «толкатель». Во вторую, сам активатор, из которого мы вызываем наш толкатель. Сохраните код и перейдите в Unity. Создайте платформу и добавьте на неё компонент вашего батута. Укажите направление. Проверьте игру.



Результат.



### Собираем вещи на карте.

Создадим новый спрайт нашей будущей собираемой вещи. В моём случае, выбор пал на ключик:



Добавим ему Collider2d. Создадим новый скрипт. Назовём его «ItemPickup». Перейдём в код. Удалим все методы. После определим лишь один метод.

|  |
| --- |
| private void OnTriggerEnter2D(Collider2D collision)  {  Destroy(this.gameObject, 0.1f);  } |

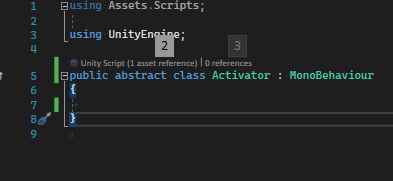
Данный метод будет удалять нашу вещь со сцену, симулируя его подбор. В будущем, будем учитывать этот процесс, а пока просто его удалим. Сохраните код. Перейдите в Unity. Поставьте вашему компоненту Collider2D флаг включенного триггера. Попробуйте забрать его.

|  |
| --- |
|  |
|  |
|  |

|  |
| --- |
| Задание.1  Напишите свой событие, которое будет что-то делать, после подбора вещи.  Задание.2  Придумайте своё применение триггеру.  Задание.3  Придумайте своё применение коллизии. |

### Активаторы и действия.

Разделим все наши написанные классы на активаторы и действия. Активатор – это класс, который будет что-то активировать. Действие – это класс, который будет что-то делать. Для этого создадим новый класс. Назовём его «Activator». После сразу создадим класс «Action». Перейдём в код «Activator». Внимание, сделаем эти классы абстрактными.



Добавим ему новое виртуальное свойство «Action».

|  |
| --- |
| public abstract class Activator : MonoBehaviour  {  public virtual Action Action { get; set; }  } |

Перейдём во второй класс «Action». Сделаем его тоже абстрактным. Объявим свойство «IsActive»

|  |
| --- |
| public abstract class Action : MonoBehaviour  {  public virtual bool IsActive { get; set; }  } |

Теперь, везде где мы хотим использовать логику: Активатор -> Действие, мы будем делать следующий алгоритм. Рассмотрим на примере нашей движущейся платформы.

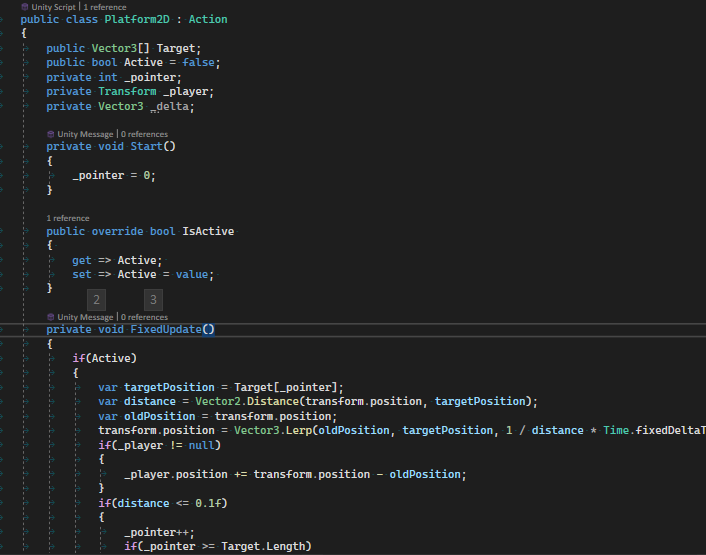
#### Действие (Action)

Переопределим наследство класса.



Добавим между полями и методами наше виртуальное свойство.

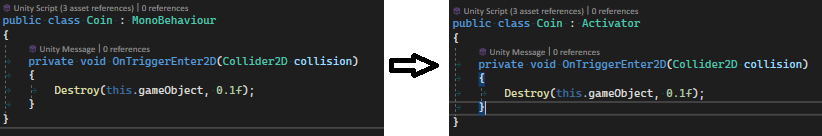
|  |
| --- |
| public override bool IsActive  {  get => Active;  set => Active = value;  } |



Готово.

#### Активатор (Activator)

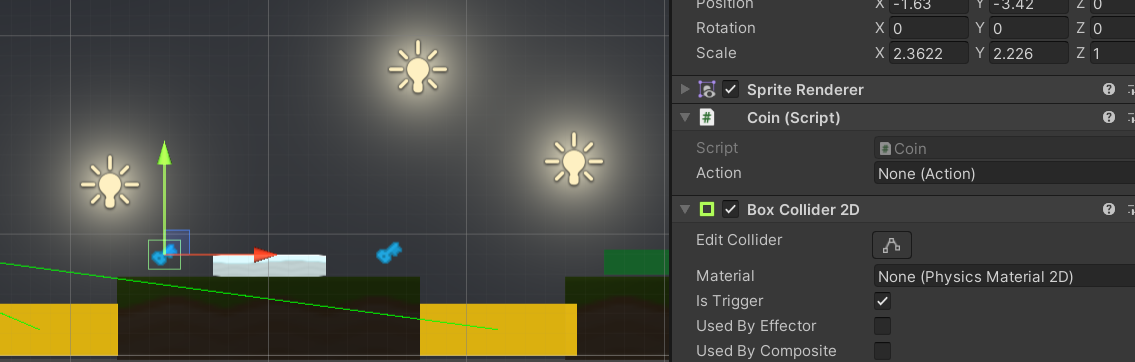
Теперь выберем активатор. Пусть это будет наш «ключик». Переопределим наследство, под наш абстрактный класс активаторов.

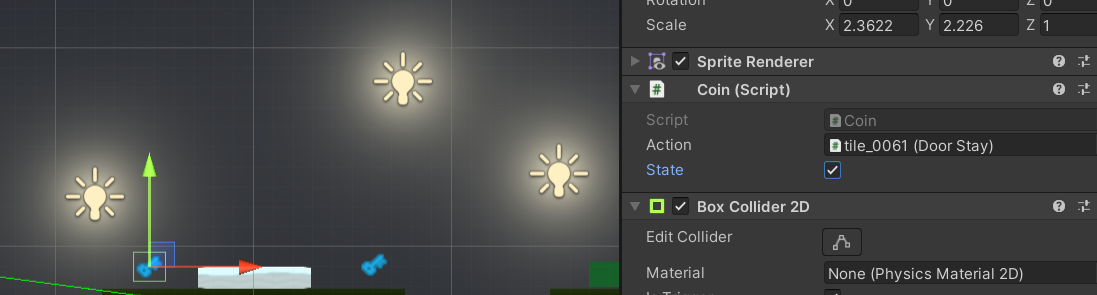


Реализуем абстрактное свойство нашего активатора. Создадим поле с действием и поле с состоянием (True/False). Переназначим поле свойству. Добавим активацию нашего действия в активаторе, при подборе вещи.

|  |
| --- |
| [SerializeField]  private Action \_action; //Поле действия  [SerializeField]  private bool \_state;  //Свойство действия  public override Action Action  {  get => \_action;  set => \_action = value;  }//Переназначение на \_action  private void OnTriggerEnter2D(Collider2D collision)  {  //Если действие не пустое  if(Action != null)  {  //Передаём наще сотсояние дейстивию.  Action.IsActive = \_state;  }  Destroy(this.gameObject, 0.1f);  } |

Как результат, в инспекторе Unity добавилось свойство выбора действия. Выбираем наше действие и проверяем результат.

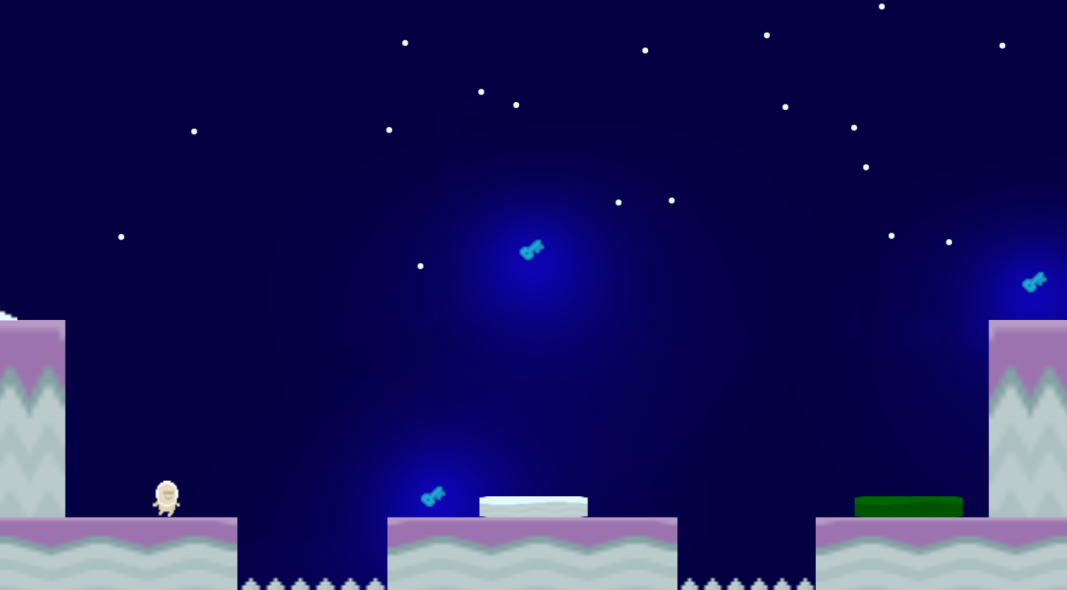




Теперь, когда мы подбираем ключ, платформа запускается и начинает движение.

|  |
| --- |
| Задание.4  Придумайте своё действие (переведите ToolAction на Action).  Задание.5  Придумайте свой активатор. |

В следующих уроках…



# Приложение

## Интересные решения

В данной рубрике представлены интересные решения, найденные в интернете или же разработанные автором курса. Для каждого решения будут прописаны необходимые главы, для понимания, как это использовать.

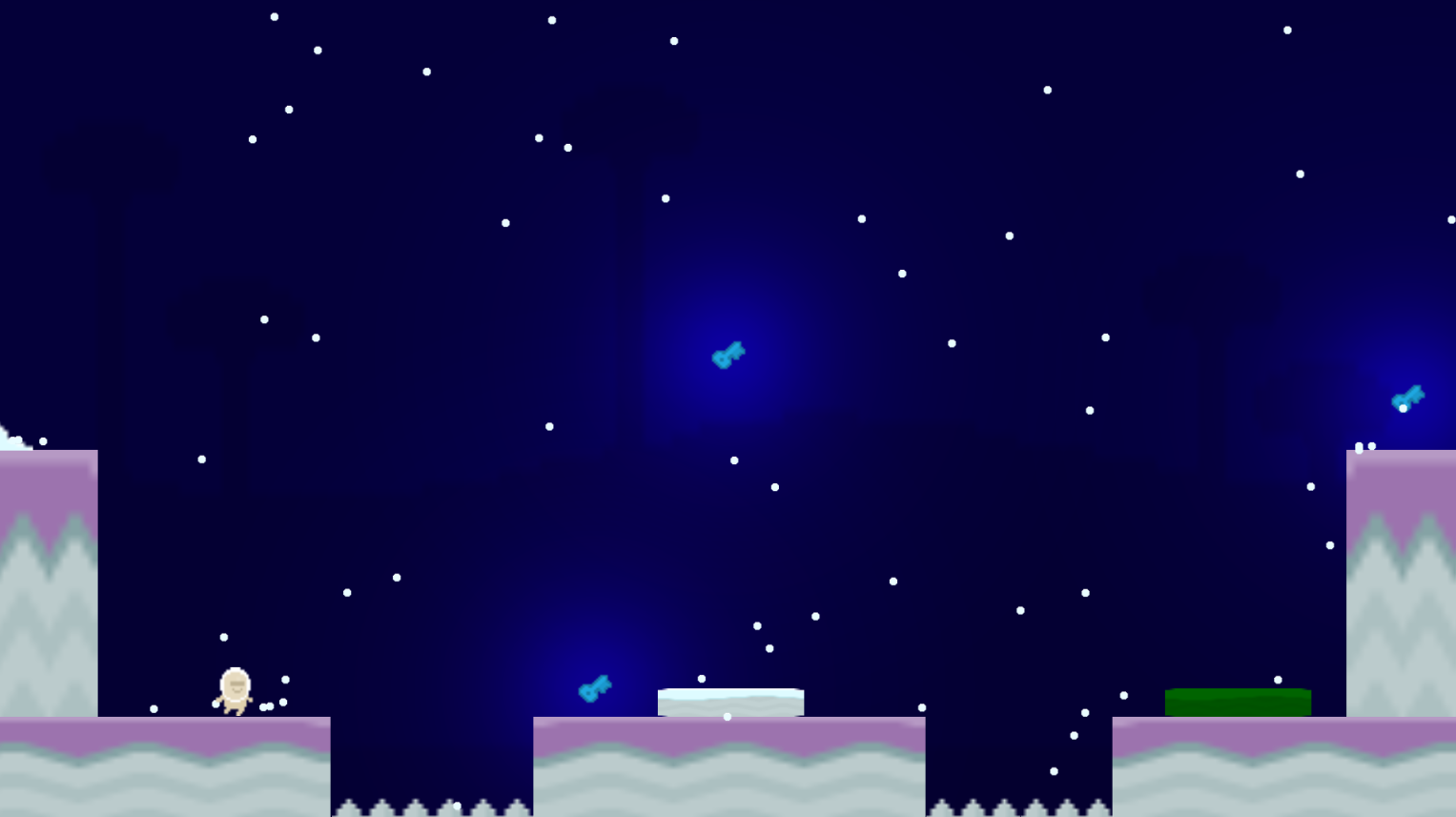
### Эффект тряски (Shake). [Глава (1.4.7).](#_Активаторы_и_действия.) Камера. Математика.

Используем знания главы (1.4.7) и труды человека, представленные по ссылке: «<https://geekstand.top/development/dobavljaem-jeffekt-trjaski-kamery-v-unity/>».

Аккумулируем наши труды с трудами автора и получаем интересный эффект тряски камеры для схемы: «Активатор» -> «Действие». Переопределяем класс нашим абстрактным методом. Реализуем методы класса «Action», где будем вызывать ShakeAction().

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class Shake : Action  {  public float \_duration = .8f;  private Transform \_cameraTransform;  private Vector3 \_originalPosition;  public override bool IsActive  {  get => base.IsActive;  set  {  ShakeAction(); //Вызываем тряску  base.IsActive = false;  }  }  // Start is called before the first frame update  void Start()  {  \_cameraTransform = Camera.main.transform;  \_originalPosition = \_cameraTransform.position;  }  public void ShakeAction()  {  \_originalPosition = \_cameraTransform.position;  StartCoroutine(\_Shake());  }  IEnumerator \_Shake()  {  float x;  float y;  float timeLeft = Time.time;  while((timeLeft + \_duration) > Time.time)  {  x = Random.Range(-0.3f, 0.3f);  y = Random.Range(-0.3f, 0.3f);  \_cameraTransform.position = new Vector3(x, y, \_originalPosition.z); yield return new WaitForSeconds(0.025f);  }  \_cameraTransform.position = \_originalPosition;  }  } |

### Эффект падающего снега. Глава (нет). Система частиц.



Для создания подобного эффекта достаточно использовать стандартную систему частиц в Unity. Рассмотрим параметры данного компонента.

|  |  |
| --- | --- |
|  | * Установим начальную скорость частиц. * Поменяем начальный размер. * Увеличим время жизни. * Переопределим «Emmiter Velocity» |
|  | * Меняем тип распределения на Box. * Устанавливаем размер Box. |
|  | * Добавим шум для не стандартного движения частиц. * Position Amount меняем на график. |
|  | * Меняем тип с Planes на World. * Mode – 2D. * Качество коллизий – Low (Static Collider), чтобы не мешать нашему персонажу (альтернатива Collides With – настройка слоёв физики) |
|  | * (Можно не трогать) * Mode – Sprites * Выбираем подходящую картинку |

## Словарь