**Создание первой игровой сцены**

Создание первой игры из компонентов, спрайтов и скриптов.

**Оглавление**

[Настройка сцены](#_vleot626mk56)

[Основные методы класса MonoBehaviour](#_2jxsxqh)

[Awake()](#_9i9pxsb2tzdk)

[Start()](#_z337ya)

[FixedUpdate()](#_3j2qqm3)

[Update()](#_1y810tw)

[LateUpdate()](#_4i7ojhp)

[Создание и уничтожение объектов](#_bu0v2tbzhyva)

[Скриптинг, работа с объектами и компонентами](#_uwqvqvnkqewq)

[Получение доступа](#_4d34og8)

[Понятие FPS](#_8phc0o7hgrg8)

[Триггеры и коллизии](#_lnxbz9)

[События, связанные с триггерами](#_35nkun2)

[События, связанные с коллизиями](#_1ksv4uv)

[Input](#_4sfashnwrn5h)

[Input Manager](#_e7kixb5j8fo8)

[Tags & Layers](#_r29hbwmkgre)

[Специальные папки и структура папок проекта](#_8ts5p1ji64bl)

[Практика](#_6s49am9nisne)

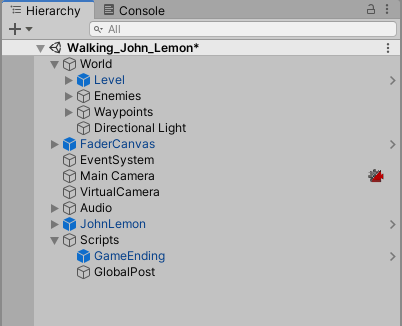
[Практическое задание](#_1ci93xb)

[Дополнительные материалы](#_3whwml4)

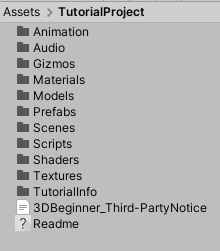
[Используемая литература](#_2bn6wsx)

# Настройка сцены

При работе в редакторе стоит уделить внимание структурированности не только файловой системе, но и иерархии на сцене, все объекты стоит строго рассортировать по их подгруппам для удобной работы с проектом в дальнейшем.



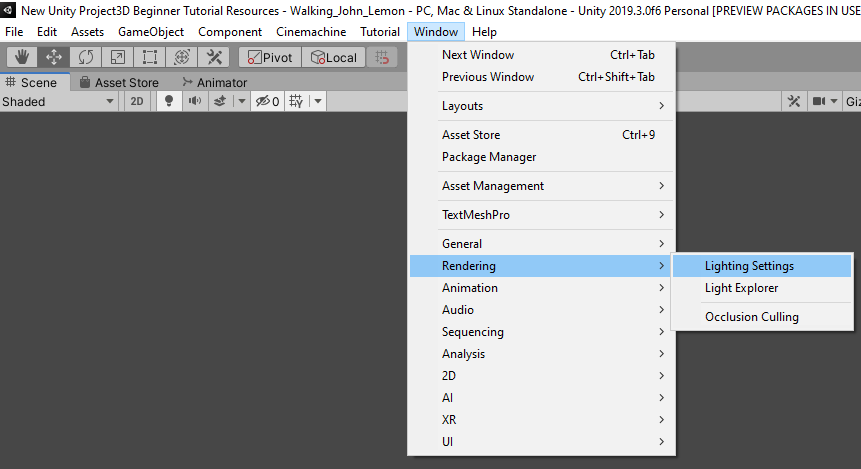
Файловая система проекта тоже должна быть строго сгруппирована.

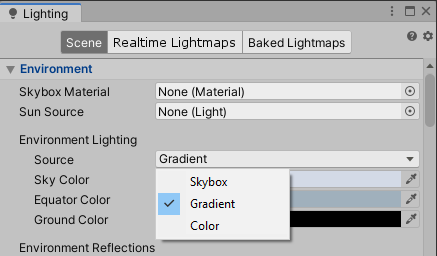


Для скачанных ассетов стоит выделить свою отдельную папку, чтобы не перегружать файловую систему проекта. Также не стоит хранить в проекте избыточное количество ассетов, чтобы не усложнять сборку проекта и запуск редактора.

Старайтесь не допускать на своих игровых сценах чересчур чёрного или тёмного окружения, старайтесь держать баланс на уровне приемлемой видимости, для этого, вы можете настраивать цвет и свет окружения с помощью **Environment Lighting** в настройках **Lighting Settings**. Режим **Gradient** достаточно оптимален для максимального контроля избытка тёмных тонов.

Окно **Lighting Settings** можно найти по адресу Window / Rendering / Lighting Settings







# Основные методы класса MonoBehaviour

MonoBehaviour — класс, на базе которого по умолчанию создаются (наследуются) все скрипты в Unity. Скрипты добавляются через контекстное меню в окне проекта, через меню **Component** на верхней панели инструментов, при помощи кнопки **Add Component** в окне инспектора (когда выделен какой-либо объект). Скрипт работает только тогда, когда он является компонентом объекта на сцене, причем некоторые функции (методы) скрипта могут срабатывать даже тогда, когда сам скрипт выключен (неактивен).

Для редактирования скрипта после его создания необходимо дважды нажать на него ЛКМ в окне инспектора или в окне проекта. Скрипт откроется в той среде разработки, которая указана в настройках **Editor — Preferences — External Tools — External Script Editor** (обычно это **Visual Studio**). Там же можно сменить среду разработки.

Имя класса в скрипте должно совпадать с именем файла (это происходит автоматически при создании файла). По умолчанию в скрипте создаются две функции (два метода) — **Start()** и **Update()**. Но существуют и другие, которые могут быть использованы как унаследованные от **MonoBehaviour**. Также можно создавать свои функции.

По умолчанию скрипт выглядит так:

|  |
| --- |
| using UnityEngine; using System.Collections;  public class MyFirstScript : MonoBehaviour  {  private void Start ()  {   }  private void Update ()  {   } } |

## Awake()

Функция, которая вызывается один раз при запуске игры до функции **Start()**. Вызывается в случайном порядке среди объектов на сцене, для которых она реализована. Функция срабатывает даже тогда, когда скрипт неактивен.

|  |
| --- |
| private void Awake()  { } |

## Start()

Функция, которая также вызывается один раз, но в тот момент, когда скрипт становится активным. Вызывается перед функцией **FixedUpdate()**, **Update()** и т. д.

|  |
| --- |
| private void Start ()  { } |

## FixedUpdate()

Функция, которая выполняется каждый фиксированный отрезок времени. Можно настроить в **Edit — Project Settings — Time — Fixed Timestep**. Используется для обработки физики (приложения силы, импульса к объекту и т. д.)

|  |
| --- |
| private void FixedUpdate()  { } |

## Update()

Функция, которая выполняется каждый кадр. Используется для обработки игровой логики.

|  |
| --- |
| private void Update()  { } |

Чтобы реализовать выполнение какого-либо действия с определенной частотой, — например, раз в секунду — используют **Time.deltaTime \* 1**.

**deltaTime** — время (в секундах), прошедшее с последнего кадра. Это значение позволяет сделать игру независимой от скорости смены кадров.

## LateUpdate()

Функция, которая выполняется каждый кадр. Вызывается после того, как все Update()-функции будут вызваны. Используется, чтобы упорядочить выполнение действий в скрипте. Например, если камера следует за объектом, движение которого реализовано в Update()-функции.

|  |
| --- |
| private void LateUpdate()  { } |

## Создание и уничтожение объектов

Для создания объекта на сцене (из префаба или как копии другого объекта) используется функция **Instantiate()**. В качестве параметров передаются префаб или объект со сцены, позиция, в которой он создается, и поворот.

Первым параметром указываем из наших ассетов префаб, который хотим создать, задаем точку и поворот префаба в пространстве.

|  |
| --- |
| Instantiate(prefab, Vector3.zero, Quaternion.identity); |

|  |
| --- |
| Instantiate(prefab, parent.transform); |

При необходимости указываем, нужно ли при создании сохранять мировые системы координат или следует создать объект относительно родительского.

|  |
| --- |
| Instantiate(prefab, parent.transform, true); |

Для уничтожения объекта или компонента используется функция **Destroy()**. Также в качестве параметра можно задать время, через которое произойдет уничтожение.

|  |
| --- |
| private void Start ()  {  Destroy(gameObject, 3); } |

# Скриптинг, работа с объектами и компонентами

## Получение доступа

Доступ к игровому объекту на сцене:

* ключевое слово **gameobject** (если скрипт является компонентом необходимого игрового объекта):

|  |
| --- |
| private void Start ()  {  gameObject.SetActive(false); *// Данный объект становится неактивным* } |

* ссылка на другой игровой объект:

|  |
| --- |
| [SerializeField] private GameObject \_anotherObject;  private void Start ()  {  \_anotherObject.SetActive(false); *// Переданный в скрипт объект становится неактивным* } |

* через **transform** (к объекту-родителю);

|  |
| --- |
| private void Start ()  {  transform.parent.gameObject.SetActive(false); *// Родительский объект данного объекта становится неактивным (как и вся иерархия объектов)* } |

* через **transform** (к объекту-наследнику); Данный способ обращения является ненадежным, так как объекты в иерархии могут менять своё местоположения при непредсказуемых обстоятельствах (например кто-то в компании решит переставить объекты местами)

|  |
| --- |
| private void Start ()  {  transform.GetChild(0).gameObject.SetActive(false); *// Нулевой ребенок данного объекта становится неактивным*  *}* |

* поиск игрового объекта по имени (это операция, требующая больших ресурсов, — лучше ее избегать):

|  |
| --- |
| private void Start ()  {  GameObject.Find("GameObject").SetActive(false); *// Первый найденный объект с именем GameObject становится неактивным* } |

* поиск игрового объекта по тегу:

|  |
| --- |
| private void Start ()  {  GameObject.FindGameObjectWithTag("Player").SetActive(false); *// Первый найденный объект с тегом Player становится неактивным. Также существует метод поиска всех объектов с заданным тегом FindGameObjectsWithTag*  } |

После объекта можно получить доступ и к компоненту:

* через метод **GetComponent<>**:

|  |
| --- |
| private void Start ()  {  gameObject.GetComponent<BoxCollider>().enabled = false; *// Выключение коллайдера для данного объекта* } |

|  |
| --- |
| private void Start ()  {  Destroy(gameObject.GetComponent<AudioSource>()); *// Уничтожение компонента AudioSource для данного объекта* } |

|  |
| --- |
| private void Start ()  {  GameObject.Find("GameObject").SetActive(false); *// Первый найденный объект с именем GameObject становится неактивным* } |

# 

# Понятие FPS

**FPS** — аббревиатура от **Frames Per Second**, что в переводе означает «количество кадров в секунду». Видео — это процесс смены кадров. Каждый кадр — статическое изображение. Количество кадров в секунду — это скорость смены изображений.

В компьютерной игре используется визуализация на основе рендеринга. То есть каждая картинка кадра визуализируется за счет программно-аппаратных алгоритмов на основе действий игрока.

Человеческий глаз способен воспринимать последовательность картинок как видео, когда сменяется не менее 25–30 кадров в секунду.

Чем качественнее каждая картинка в кадре, тем больше ресурсов компьютера нужно для ее рендеринга. Чем больше логических операций в коде — тем более значительные ресурсы процессора потребуются, чтобы выполнить кадр. Соответственно, понадобится больше времени на подготовку следующего кадра. Таким образом, показатель FPS будет уменьшаться, что приведет к нестабильной работе программы.

Чем ниже показатель FPS, тем меньше кадров для логики. В этих кадрах и происходит игровая логика, поэтому задача программиста — оптимизировать работу каждого кадра.



## Input

Вы можете считывать ввод с клавиатуры прямым способом через условие. Есть несколько вариантов:

* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.GetKey** — напрямую считывает с клавиатуры вводимые данные;
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.GetKey** — возвращает **true**, пока нажата указанная клавиша;
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.GetKeyDown** — единожды возвращает **true**, если указанная клавиша была нажата;
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.GetKeyUp** — единожды возвращает **true**, если указанная клавиша была отпущена.

|  |
| --- |
| private void Update()  {  if (Input.GetKey(KeyCode.RightArrow))  print("Right Arrow pressed");   if (Input.GetKey(KeyCode.D))  print("D pressed");   if (Input.GetKeyDown(KeyCode.RightArrow))  print("Right Arrow was pressed");   if (Input.GetKey(KeyCode.D))  print("D pressed");  } |

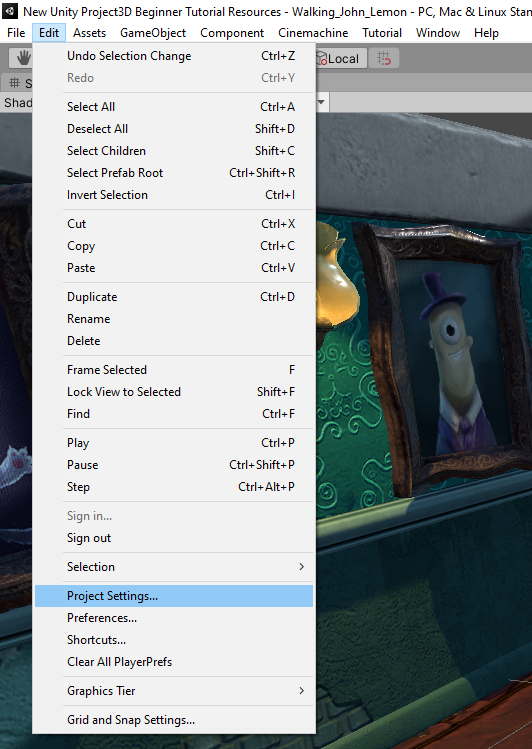
Чтобы увидеть полный список команд, используя Visual Studio, впишите **Input.GetKey(KeyCode** и нажмите F12, чтобы перейти к описанию **KeyCode**.

Для считывания ввода с мыши используйте [**GetMouseButton**](https://docs.unity3d.com/ru/current/ScriptReference/Input.GetMouseButton.html) — он работает почти так же, как и **GetKey**, только в скобках вы указываете индекс кнопки мыши типа **int**, где «0» — это ЛКМ (левая кнопка мыши), «1» — ПКМ и «2» — СКМ. Далее — в зависимости от конфигурации определенной мыши:

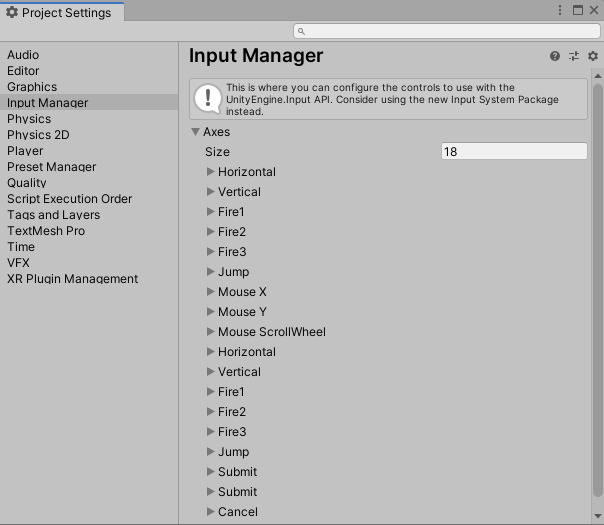
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.**[**GetMouseButton**](https://docs.unity3d.com/ru/current/ScriptReference/Input.GetMouseButton.html) — возвращает **true**, пока нажата указанная кнопка мыши;
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.**[**GetMouseButton**](https://docs.unity3d.com/ru/current/ScriptReference/Input.GetMouseButton.html)**Down** — возвращает **true**, когда отпущена указанная кнопка мыши;
* [**Input**](https://docs.unity3d.com/ru/current/ScriptReference/Input.html)**.**[**GetMouseButton**](https://docs.unity3d.com/ru/current/ScriptReference/Input.GetMouseButton.html)**Up** — единожды возвращает **true**, когда нажата указанная кнопка мыши.

|  |
| --- |
| private void Update()  {  if (Input.GetMouseButton(0))  print("LMB pressed");   if (Input.GetMouseButtonUp(1))  print("MMB was released");   if (Input.GetMouseButtonDown(2))  print("RMB was clicked");  } |

## Input Manager

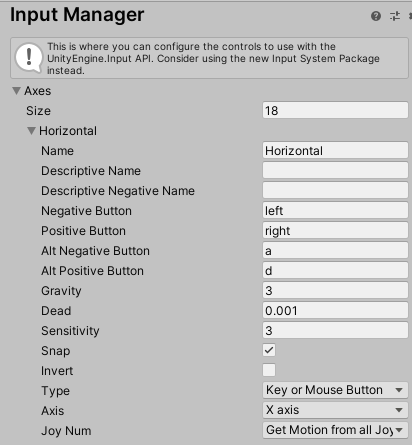
Input Manager находится во вкладке **Edit > Project Settings** 

Во вкладке **Input Manager.**

****

**Input Manager** позволяет задать оси ввода и кнопки в рамках игрового проекта, а не одной сцены. Это удобно: во-первых, с точки зрения обращения к осям и кнопкам, а во-вторых — для настройки управления игроком.

Рассмотрим, как по умолчанию выглядит **Input Manager**.



**Первая ось Horizontal и ее настройки:**

● **Name** — имя, использующееся для обращения к оси через скрипт и в настройках управления игрока (окно, которое по умолчанию появляется перед запуском игры, — далее окно настроек игры);

● **Descriptive Name** — подробное описание **Positive Button** для окна настроек игры;

● **Descriptive Negative Name** — подробное описание **Negative Button** для окна настроек игры;

● **Negative Button** — кнопка, которая отправляет отрицательное значение оси;

● **Positive Button** — кнопка, которая отправляет положительное значение оси;

● **Alt Negative Button** — альтернативная кнопка (отрицательное значение оси);

● **Alt Positive button** — альтернативная кнопка (положительное значение оси);

● **Gravity** — скорость, с которой значение возвращается в исходное состояние. Используется только для типа **Key or Mouse button**;

● **Dead** — любое положительное или отрицательное значение оси, которое меньше данного значения, будет рассматриваться как нулевое. Используется для стиков джойстика, когда их небольшое отклонение не учитывается;

● **Sensitivity** — чувствительность, время отклика на нажатие;

● **Snap** — если включена, то значение оси будет сразу же сброшено в исходное состояние при нажатии противоположной кнопки. Используется только для типа **Key or Mouse button**;

● **Invert** — инверсия направления оси;

● **Type — Key or Mouse button** (любые кнопки), **Mouse movement** (движение мыши и ее колесика), **Joystick Axis** (джойстики);

● **Axis** — ось, значение которой берется с устройства;

● **Joy Num** — какой джойстик использовать. По умолчанию значения берутся со всех джойстиков. Используется только для осей, не для кнопок.

Для удаления или добавления новой оси необходимо нажать ПКМ на имя любой уже существующей оси и выбрать соответствующий пункт меню: **Duplicate Array Element** (дублировать ось), **Delete Array Element** (удалить ось). Также можно изменить количество осей напротив поля **Size**. Удаляться или дублироваться в таком случае будет последняя ось.

Для использования заданных имен в скрипте применяется **GetAxis()** и **GetButton()**:

* **Input.GetButtonDown («Horizontal»)** — возвращает **true** в первый кадр после нажатия;
* **Input.GetButton («Horizontal»)** — возвращает **true**, пока нажата кнопка;
* **Input.GetButtonUp («Horizontal»)** — возвращает **true** в первый кадр после отпускания;
* **Input.GetAxis («Horizontal»)** — возвращает значение оси от –1 до 1.

|  |
| --- |
| float h = Input.GetAxis(«Horizontal»);  print("Value h = " + h) *// Выведется сообщение о текущем значении h*   If (Input.GetButton(«Horizontal»)  print(«Horizontal Button pressed»); |

## Tags & Layers

Теги используются для обозначения особых игровых объектов (например, главная камера, игрок и т. д.). Тег для объекта выбирается в выпадающем списке ниже имени объекта в окне инспектора. Существует набор тегов по умолчанию, которые можно использовать (они не отображаются в меню Tags & Layers).

После добавления нового тега в меню необходимо выделить объект и назначить ему тег. В скриптах теги используются следующим образом:

|  |
| --- |
| *// Получение тега объекта*  print(gameObject.tag);  *// Поиск объекта с тегом Player*  GameObject player = GameObject .FindGameObjectWithTag(«Player»)  *// Поиск объектов с тегом Player*  GameObject[] players = GameObject .FindGameObjectsWithTag(«Player»)  *// Проверка, имеет ли коллайдер объекта заданный тег*  coll.CompareTag(«Player») |

В Unity два вида слоев: **Layers** и **Sorting Layers**. Первый используется для группировки объектов определенных категорий (окружающая среда, декорации, интерфейс, противники и т. д.). Второй — для спрайтов (2D-изображений), чтобы установить порядок их отрисовки при одинаковом расположении по оси Z.

**Layers** выбирается в выпадающем списке ниже имени объекта в окне инспектора. Существует набор слоев по умолчанию (максимальное количество слоев ограничено).

Доступ к слоям через скрипт:

|  |
| --- |
| print(gameObject.layer); *// Вывод номера слоя, нумеруются с 0* gameObject.layer = 1; *// Слой задается по номеру* |

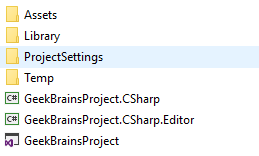
Sorting Layers устанавливается в компоненте (также может быть установлен порядок изображений, когда в одном слое их несколько).

Доступ к слоям сортировки через скрипт:

|  |
| --- |
| sprite.sortingLayerID = 0; *// Задать или получить номер слоя сортировки*  sprite.sortingLayerName = «Background»; *// Задать или получить имя слоя сортировки* sprite.sortingOrder = 1; *// Задать или получить порядок спрайта в слое сортировки* |

# Специальные папки и структура папок проекта

В каждом новом проекта Unity создает много папок и файлов, но основные из них — это **Assets** и **ProjectSettings**. Остальные генерируются на их основе.



Рассмотрим некоторые специальные папки проекта:

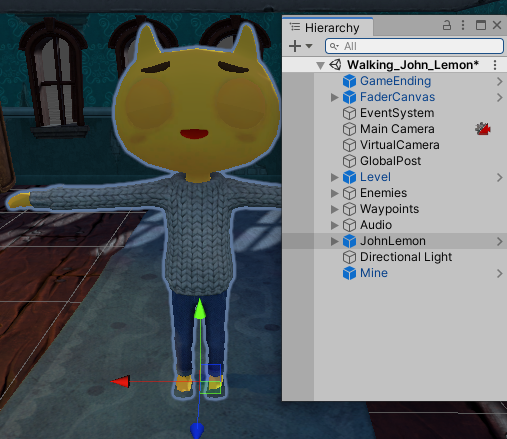
* **Assets** — главная папка, в которой содержатся все ассеты, которые могут быть использованы проектом. Содержимое окна проекта соответствует содержимому папки **Assets**. Большинство функций предполагает, что необходимый для них ресурс находится в папке **Assets** и не требует явного указания расположения;
* **Editor** — все скрипты, размещенные в этой папке (или подпапке), расцениваются как скрипты редактора. То есть они предназначены для расширения функциональности редактора и недоступны во время выполнения проекта;
* **Editor Default Resources** — папка, которая должна быть размещена в корневой папке (Assets). Содержащиеся в ней ассеты использует функция **EditorGUIUtility.Load**, которая загружает файлы по мере надобности;
* **Gizmos** — в этой папке размещаются дополнительные изображения для иконки-маркера объектов на сцене;
* **Plugins** — хранит различные расширения (модули, dll) для Unity;
* **Resources** — в данной папке размещаются ассеты, экземпляры которых создаются в проекте и используются во время игры. Может быть несколько папок Resources, и располагаться они могут где угодно;
* **Standard Assets** — стандартные ассеты Unity для обычной версии движка;

Также важно учитывать порядок компиляции скриптов. Подробнее об этом — в дополнительных материалах.

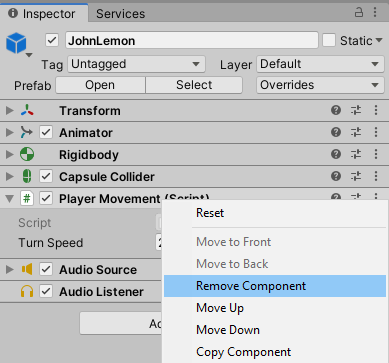
# Практика

Перед началом практики необходимо скачать **3D Beginner: Complete Project**,с которым мы будем работать.

Найдём Джона Лемона на сцене



Удалим скрипт перемещения с Джона Лемона



Напишем свой скрипт передвижения на основе **transform.Translate**, не использующей физику.

|  |
| --- |
| [SerializeField] private float \_speed; *// Скорость движения, а в дальнейшем ускорение* [SerializeField] private Vector3 \_direction; *// Направление движения*  private void Update() {  \_direction.x = Input.GetAxis(*"Horizontal"*);  \_direction.z = Input.GetAxis(*"Vertical"*);  }  private void FixedUpdate() {  var speed = \_direction \* \_speed \* Time.deltaTime;  transform.Translate(speed);  } |

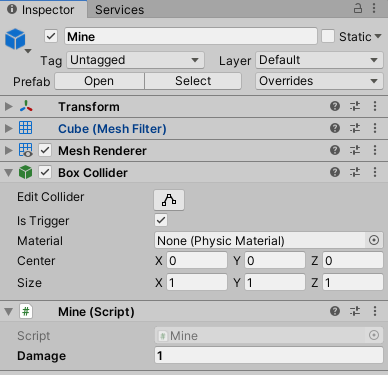
Создадим класс, который будет создавать мину, которая будет убивать противника.

|  |
| --- |
| [SerializeField] private GameObject \_mine; *// Наша мина*  [SerializeField] private Transform \_mineSpawnPlace; *// точка, где создается мина*  private void Update() {  *// Если нажата кнопка*    if (*Input*.GetButtonDown(«Fire1»))  {    *Instantiate(*\_mine*,* \_mineSpawnPlace*.position,* \_mineSpawnPlace*.rotation);*   *// Создаем \_mine в точке \_mineSpawnPlace*  }  } |

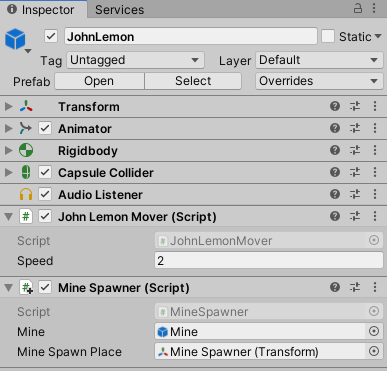
**Mine**— скрипт мины, которая наносит урон и убивает.

|  |
| --- |
| [SerializeField] private int \_damage;  private void OnTriggerEnter(Collider other)  {  if (other.gameObject.CompareTag(*"Enemy"*))  {  var enemy = other.GetComponent<MyEnemy>();  enemy.Hurt(\_damage);  Destroy(gameObject);  }  } |

Далее надо создать префаб мины, чтобы создавать её на сцене



После чего “повесим” наши скрипты в виде компонентов на Джона Лемона. В итоге должно получиться так:



Возьмём уже существующего призрака и дадим ему будущий скрипт **MyEnemy**.





А вот содержимое нашего противника:

|  |
| --- |
| public class MyEnemy : MonoBehaviour {  [SerializeField] private int \_health;  public void Hurt(int damage)  {  print(«Ouch: » + damage);  \_health -= damage;;  if (\_health <= 0)  {  Die();  }  }  private void Die  {  Destroy(gameObject);  } } |

Выполните самостоятельно практическое задание — реализуйте отнимание жизней и смерть противника.

# Практическое задание

1. Собрать свой уровень (по своему концепту).
2. Написать свой скрипт контроллера (передвижение).
3. Реализовать спаунер противников.
4. \* Реализовать стрельбу или “бросок предмета”.
5. \* Реализовать своего передвигающегося противника с полем зрения.
6. \* Добавить загадки и открывающиеся/исчезающие двери.
7. \* Добавить звуки для объектов (например двери) и взрывов.
8. \* Сделать прыжок.
9. \*\* Сделайте счётчик убитых, как только Джон победит всех, он выиграл.

Задание под \* — необязательное. Это дополнительная задача для тех, кто считает, способен на большее.

# Дополнительные материалы

1. Asset Workflow: [https://docs.unity3d.com/Manual/AssetWorkflow.html.](https://docs.unity3d.com/Manual/AssetWorkflow.html)
2. John Lemon's Haunted Jaunt: 3D Beginner: [John Lemon's Haunted Jaunt: 3D Beginner - Unity Learn](https://learn.unity.com/project/john-lemon-s-haunted-jaunt-3d-beginner)
3. Класс Time [Scripting API: Time](https://docs.unity3d.com/ScriptReference/Time.html)

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальный сайт Unity: [https://unity3d.com/ru.](https://unity3d.com/ru)
2. Метод Translate <https://docs.unity3d.com/ScriptReference/Transform.Translate.html>
3. Input Manager <http://docs.unity3d.com/Manual/class-InputManager.html>