Знакомство с Unity

Занятие 2

Оглавление

ВВЕДЕНИЕ

Данный файл, несмотря на его размер, направлен не на то, чтобы напугать Вас и оттолкнуть от возможности знакомства с Unity.

Цель занятия: знакомство с основной терминологией, связанной с программированием в Unity.

Только путем создания своего собственного проекта практически с нуля Вы сможете освоить как программирование, так и «нажатие кнопочек», которые предлагает Вам Unity при первом знакомстве.

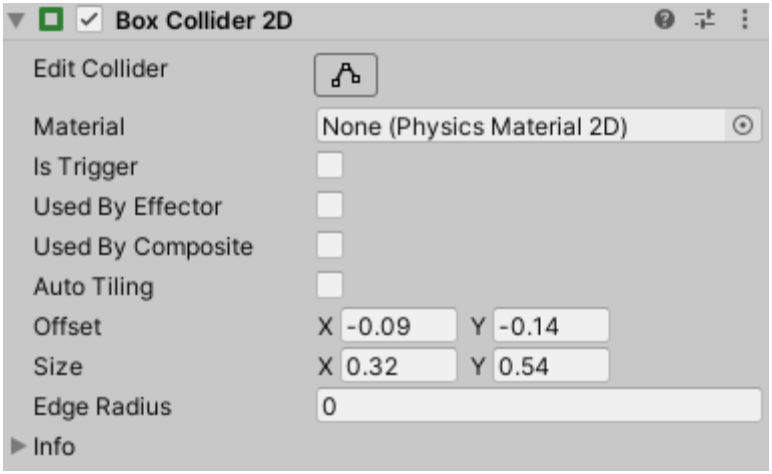
В ОГЛАВЛЕНИЕ

КОМПОНЕНТЫ 2D ОБЪЕКТОВ

Рассмотрим некоторые компоненты и события 2D-объектов и их отличия от компонентов и событий, использующихся для 3D-объектов.

BoxCollider2D

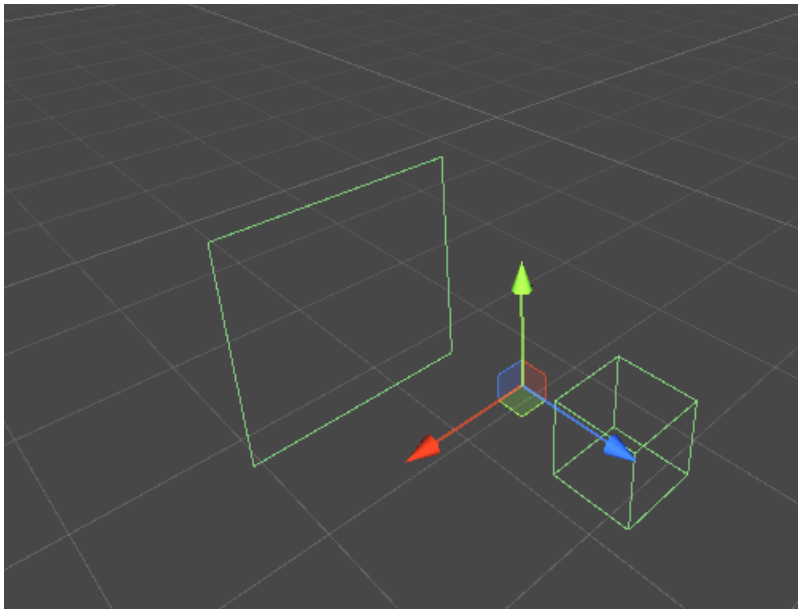
BoxCollider2D – это прямоугольный коллайдер, используемый при работе с 2D физикой.



2D коллайдер/триггер обрабатывает события (сталкивание, пересечение и т.д.) независимо от того, где он располагается по координате Z. Все свойства коллайдера точно такие же, как и у BoxCollider, за исключением одного. Переключатель Used By Effector отвечает за то, используется ли данный коллайдер прикрепленным компонентом – Effector 2D (данный компонент бывает разного типа: точка, область, платформа и т.д.).

Дадим краткие характеристики для каждого из пунктов компоненты:

- Material – физический материал, который определяет свойства столкновений, такие как трение и отскок.
- Is Trigger – является ли коллайдер триггером?
- Used by Effector – используется ли данный коллайдер прикрепленным компонентом – Effector 2D.
- Used by Composite – установите этот флажок, если вы хотите, чтобы этот коллайдер использовался прикрепленный Composite Collider 2D.
- Auto Tiling – установите этот флажок, если компонент Sprite Renderer установлен в режим Tile. Это позволит автоматически, обновлять форму коллайдера, которая будет корректироваться по форме спрайта.
- Offset – локальное смещение геометрии коллайдера.
- Size – ширина и высота прямоугольника в единицах локального пространства.
- Edge Radius – смягчает грани, делая коллайдер более круглым, по умолчанию значение равно 0.
- Info – общая информация о состоянии коллайдера.

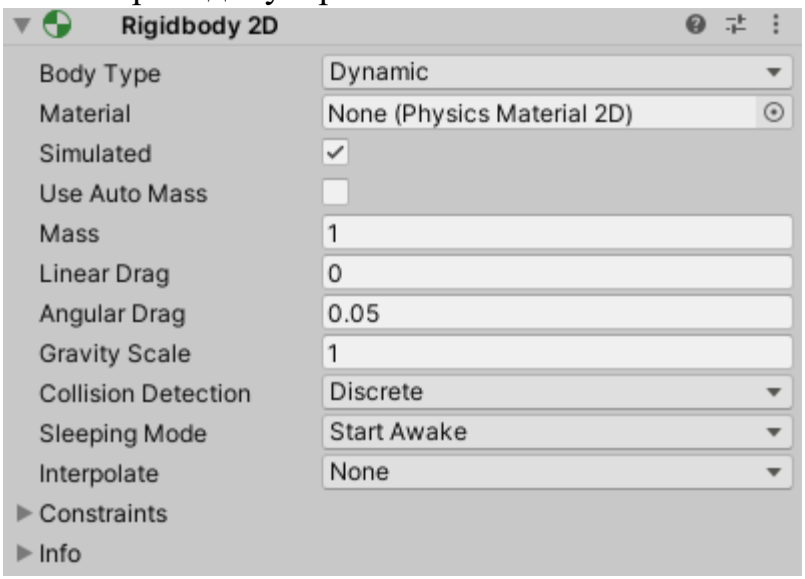


Пустые GameObject с компонентами BoxCollider и BoxCollider2D в 3D режиме сцены

В ОГЛАВЛЕНИЕ

RigidBody2D

Rigidbody2D – компонент помещает объект под контроль физического движка. Само по себе это означает, что на спрайт будет воздействовать гравитация и им можно будет управлять из скриптов с помощью сил. Добавив соответствующий компонент коллайдера, спрайт также будет реагировать на столкновения с другими спрайтами. В целом компонент похож на Rigidbody, но объект может двигаться только в плоскость XY, а вращаться только перпендикулярно этой плоскости.



Дадим краткие характеристики для каждого из пунктов компоненты:

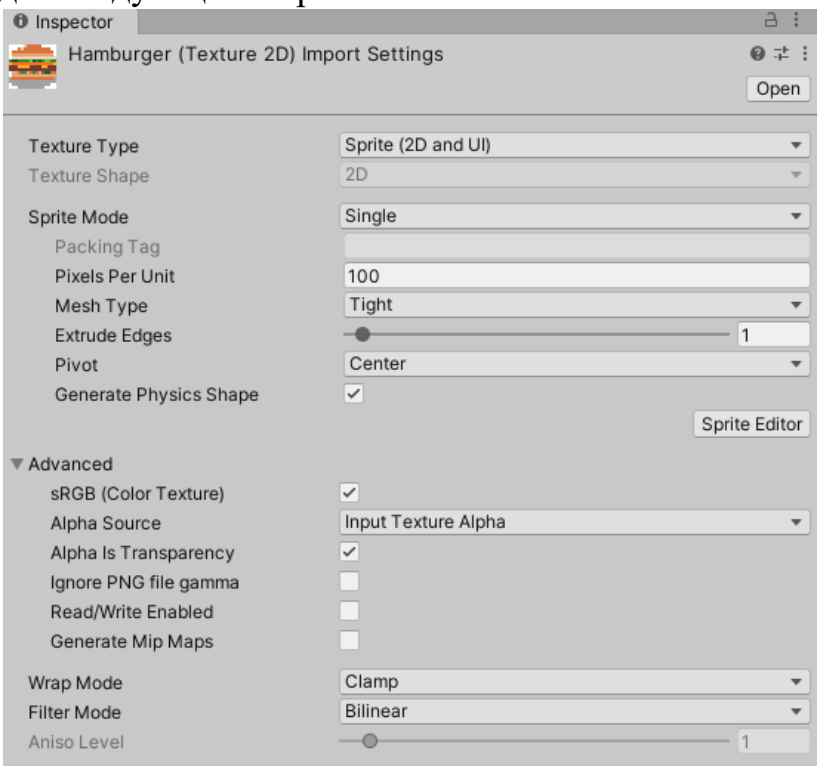
- Body Type – тип физического поведения (Dynamic, Kinematic, Static).
- Simulated – указывает, должно ли физическое тело вести себя как твердое тело или нет.
- Use Auto Mass – рассчитать массу автоматически, исходя из коллайдера;
- Mass – масса объекта Rigidbody;
- Linear Drag, Angular Drag – коэффициенты линейного и углового сопротивления;
- Gravity Scale – как гравитация воздействует на объект;
- Collision Detection – метод, используемый физическим движком для проверки столкновения двух объектов.
- Sleeping Mode – режим, когда объект выключается из физической симуляции (в целях оптимизации). Never Sleep – всегда активен (необходимо стараться избегать данного режима), Start Awake – активен при запуске (режим включен по умолчанию), Start Asleep – неактивен при запуске, но может быть разбужен (столкновением или через скрипт).
- Interpolate – физическая интерполяция, используемая между обновлениями

В ОГЛАВЛЕНИЕ

СПРАЙТЫ

НАСТРОЙКИ СПРАЙТА

Информация по импорту 2D ресурсов может быть найдена здесь.
Первым делом необходимо импортировать изображение в проект. Сделать это можно, переместив файл с изображением в окно Assets, находящееся на вкладке Project внизу. Также можно нажать ПКМ → Import new asset → Выбрать файл на компьютере.
После импорта изображения в проект в 2D-режиме разработки его настройки по умолчанию выглядят следующим образом:



Дадим краткие характеристики для этих настроек:

- В верхней части слева находится иконка – превью спрайта (также превью находится в нижней части инспектора), правее отображается название файла, кнопка Open, которая открывает изображение в программе просмотра изображений, установленной в системе по умолчанию.
- Texture Type – устанавливает базовые параметры в зависимости от предназначения текстуры (изображения).
- Sprite Mode – устанавливает, в каком режиме спрайт будет извлекаться из изображения, как один спрайт (Single) или как несколько раздельных спрайтов (Multiple). Второй режим может использоваться для импорта спрайтов как кадров анимации, а также как несколько спрайтов, относящихся к одному объекту. Третий режим Polygon позволяет настраивать форму, размер и pivot point для вашего спрайта.
- Packing Tag – имя атласа, в который данное изображение должно быть запаковано.
- Pixels Per Unit – количество пикселей в изображении спрайта, которые соответствуют одному юниту (внутренняя единица измерения в Unity).

- Mesh Type – определяет тип меша, который создаётся для спрайта, по умолчанию это Tight.
    - Full Rect – создает квадрат по размеру спрайта,
    - Tight – создает сетку на основе альфа-канала в пикселях, сгенерированная сетка повторяет форму спрайта.
- Extrude Edges – определяет область вокруг спрайта, которую нужно оставить в сгенерированной сетке.
- Pivot – точка спрайта, которая является центром локальной системы координат.

Advanced

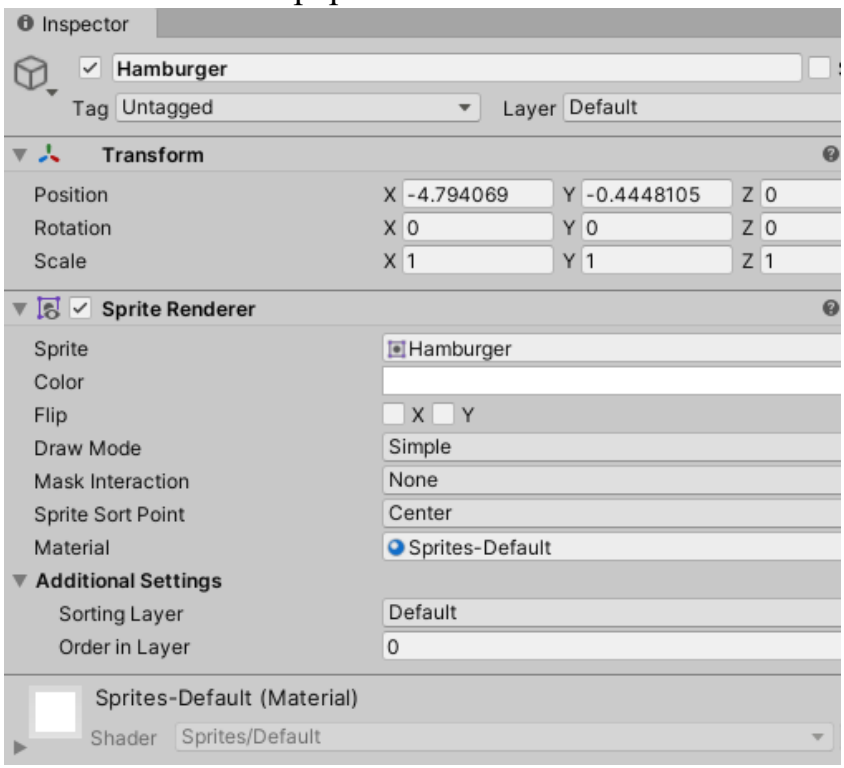
- sRGB (Color Texture) – указывает, хранится ли данная текстура в гамма пространстве. Это значение должно быть у всех цветных текстур, не имеющих HDR. Если вам нужно обратиться к точным значениям текстуры в шейдере (например, smoothness или metalness, то снимите флажок.)
- Alpha Source – указатель, как генерируется альфа-канал с текстуры.
    - None – не генерировать альфу, даже если текстура её имеет;
    - Input Texture Alpha – использовать альфа-канал текстуры;
    - From Grayscale – сгенерировать альфу из средних тонов.
- Alpha is Transparency – если у спрайта есть прозрачный альфа-канал, стоит включить Alpha is Transparency, чтобы увеличить радиус фильтрации, чтобы избежать артефактов на краях текстуры.
- Read/Write Enabled – этот параметр разрешает доступ к данным текстуры из функций скриптов (таких как: Texture2D.SetPixels, Texture2D.GetPixels и другие функции Texture2D). Но помните, что будет сделана копия данных Texture, удваивая объем памяти, необходимый для Texture Asset, поэтому используйте это свойство, если это действительно необходимо. Это справедливо только для несжатых и сжатых текстур DXT; другие типы сжатых текстур не будут работать. По умолчанию это свойство отключено.
- Generate Mip Maps – генерировать или нет mipmap-ы изображения (результат можно посмотреть в превью в нижней части инспектора, используя ползунок в правом верхнем углу). Mipmap – последовательность изображений, каждое из которых имеет более низкое разрешение, т.е. уменьшенные версии текстуры, которые используются, когда на экране текстура представлена в очень маленьком размере
    - Border Mip Maps – включите, чтобы мип-мапы плавно становились черно-белыми по прогрессу мип уровней. Это используется для Detail карт. Самый левый мип, это самый первый уровень мипа на переходе в серые цвета. Самый правый определяет мип уровень, где текстура будет полностью выцветшей.
    - Mip Map Filtering – для оптимизации качества изображения доступно 2 способа фильтрации мип-мапов:
        - Box – самый простой способ это затухание мипмапов – уровни мипов становятся глаже и глаже по мере уменьшения размера.

        - Kaiser – алгоритм резкости Кайзера работает на мип-мапах по мере их уменьшения размера. Если на расстоянии текстуры слишком размытые, то попробуйте эту опцию.
        - Fade Out Mipmaps – включите, чтобы мип-мапы плавно становились черно-белыми по прогрессу мип уровней. Это используется для Detail карт. Самый левый мип, это самый первый уровень мипа на переходе в серые цвета. Самый правый определяет мип уровень, где текстура будет полностью выцветшей.
- Wrap Mode – отвечает за поведение текстуры при тайлинге:
    - Repeat – текстура повторят (тайлит) саму себя;
    - Clamp – края текстуры растягиваются.
- Filter Mode – отвечает за то, как текстура фильтруется, если её растянуло при 3D-трансформациях:
    - Point (no filter) – вблизи текстура становится более пиксельной;
    - Bilinear – вблизи текстура становится более размытой;
    - Trilinear – как и Bilinear, но текстура также размывается и на разных мип уровнях.
- Aniso Level – увеличивает качество текстуры, при обзоре под большим углом. Хорошо подходит для пола и текстур земли.
- Настройки разрешения и сжатия для всех платформ или для каждой по отдельности.
- Кнопки Revert и Apply для сохранения или отмены сделанных изменений.

В ОГЛАВЛЕНИЕ

КОМПОНЕНТ SPRITE RENDERER

Подробную информацию о Sprite Renderer можно найти здесь.
Данный компонент позволяет отображать спрайты и используется как в 2D, так и в 3D-сценах.
Чтобы посмотреть содержимое данного компонента, необходимо перенести недавно добавленный ассет Hamburger на сцену. Справа откроется окно инспектора, слева мы увидим, что он добавился в окно иерархии.



Компонент имеет следующие свойства:

- Sprite – спрайт;
- Color – цвет;
- Flip – отображение спрайта по оси X или Y;
- Draw Mode – выберите параметр из списка Draw Mode, чтобы определить, как масштабируется спрайт при изменении его параметров.
    - Simple – это значение по умолчанию. Изображение изменяется во всех направлениях, когда его размеры меняются.
    - Sliced – применяйте это, если хотите нарезать спрайт на 9 частей, чтобы эти части можно было растягивать. В этом режиме углы остаются одинаковыми, верхняя и нижняя часть тянутся в собственную высоту, а центр тянется в ширину и высоту.
        - Size – используйте это значение, чтобы изменить горизонтальный и вертикальный размер спрайта. Если вы хотите, чтобы slice работала корректно, вам нужно настроить это значение. Компонент Transform применяется только по умолчанию.
    - Tiled – применяйте это если хотите нарезать спрайт на 9 частей, чтобы эти части можно было повторять. В этом режиме углы остаются одинаковыми,

верхняя и нижняя часть тянутся в собственную высоту, а центр тянется в ширину и высоту.

- Tile Mode – если у вас установлено значение Tile, используйте свойство Tile Mode, чтобы управлять тем, как разделы повторяются при изменении размеров спрайта.
  - Continuous – режим тайла по умолчанию установлен на Continuous. Когда изменяется размер спрайта, повторяющиеся секции повторяются равномерно в спрайте.
  - Adaptive – если режим Tile Mode установлен на Adaptive, повторяющиеся разделы повторяются только тогда, когда размеры Sprite достигают значения растяжения.
    - Stretch Value – с помощью ползунка Stretch Value установите значение между 0 и 1. Обратите внимание: единица представляет изображение, уменьшенное в два раза до его первоначальных размеров, поэтому, если значение Stretch Value установлено на единицу, секция повторяется, когда изображение растягивается в два раза до его первоначального размера

- Mask Interaction – установите, как средство визуализации спрайтов ведет себя при взаимодействии с маской спрайта.
  - None – средство визуализации спрайтов не взаимодействует ни с какими масками спрайтов в сцене. Это опция по умолчанию.
  - Visible Inside Mask – спрайт виден там, где на него накладывается маска спрайта, но не за ее пределами.
  - Visible Outside Mask – спрайт виден снаружи маски спрайта, но не внутри нее. Маска спрайта скрывает части спрайта, которые она накладывает.
- Sprite Sort Point – выберите между центром спрайта или его точкой вращения при расчете расстояния между спрайтом и камерой.
- Material – материал, стандартные шейдеры, которые используют для материала спрайта – Sprites/Default (не взаимодействует с освещением на сцене), Sprites/Diffuse (взаимодействует с освещением на сцене);
- Sorting Layer – слой сортировки (новый слой можно создать в меню Edit – Project Settings – Layers);
- Order in Layer – порядок спрайта в слое сортировки (спрайты с большим значением отрисовываются поверх спрайтов с меньшим).

В ОГЛАВЛЕНИЕ

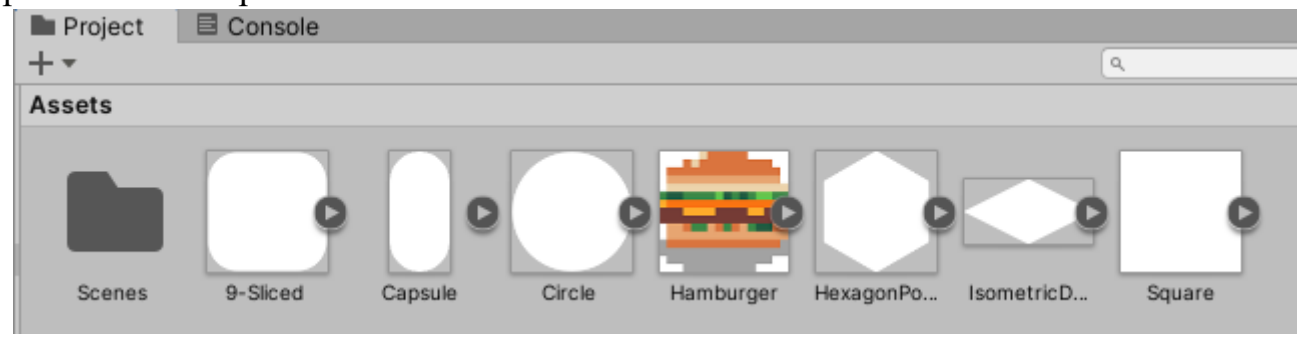
SPRITE CREATOR

Руководство по Sprite Creator расположено на этой странице.

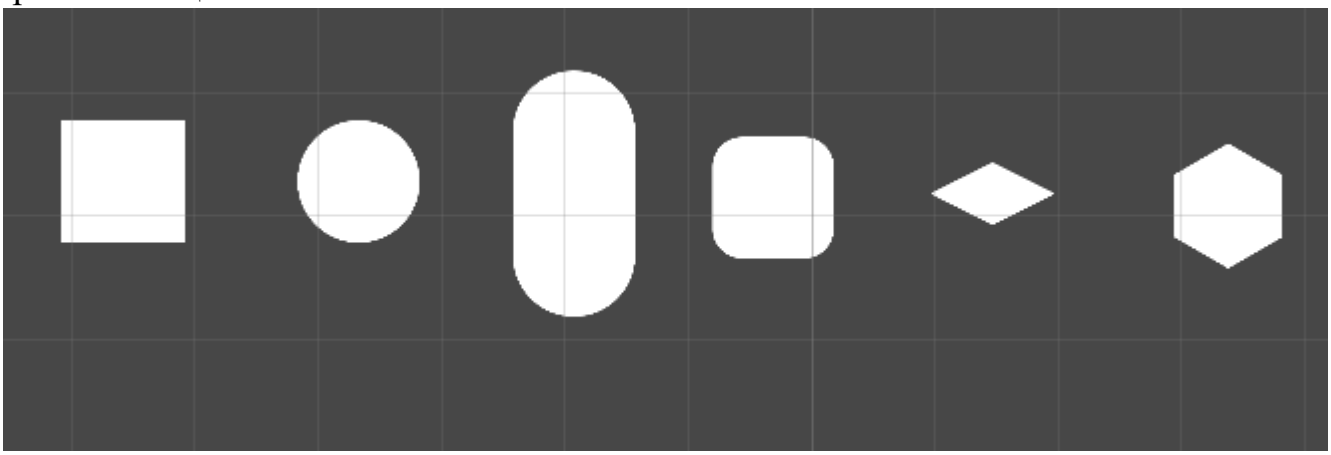
С помощью этого инструмента можно создавать временные заменители спрайтов. Их можно использовать в проекте в процессе разработки, а затем, когда будет создана необходимая графика, заменить. Для создания таких спрайтов необходимо выбрать меню Assets – Create – 2D – Sprites, далее один из доступных типов спрайтов: Square, Circle, Capsule, Isometric Diamond, Hexagon Pointed-Top, 9-Sliced.

Через меню Assets – Create создаются не объекты, а ассеты, поэтому после создания спрайта в папке проекта появляется новый ассет, а на сцене ничего не изменяется. Далее необходимо добавить созданный спрайт на сцену.

Спрайты в окне проекта:



Спрайты на сцене:



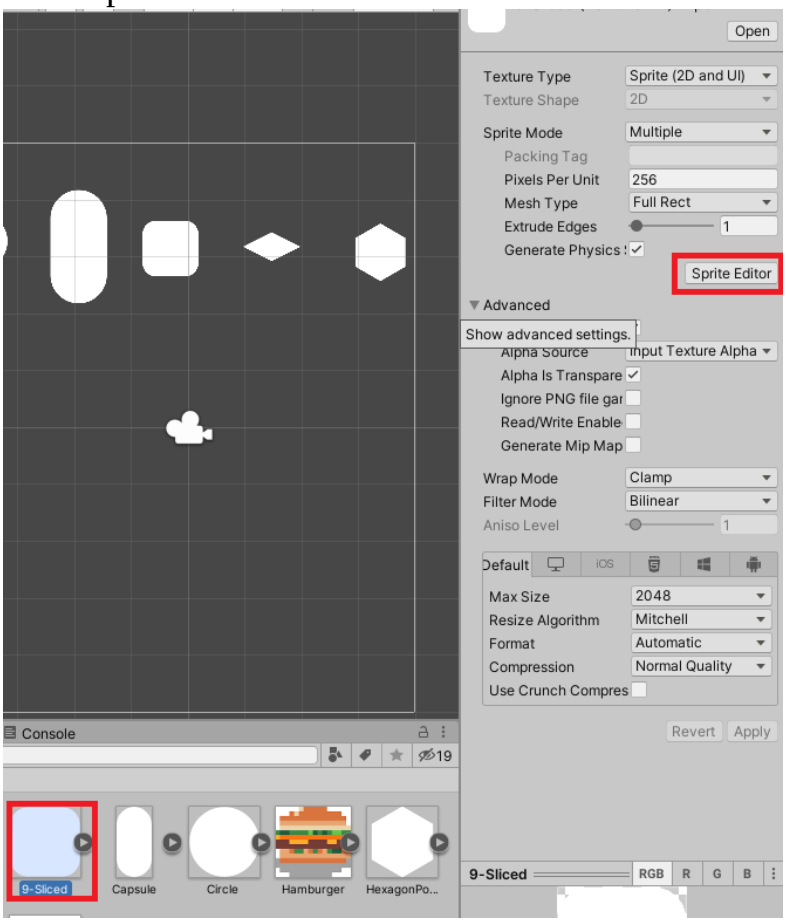
Sprite Creator создает белые контурные PNG текстуры размером 4x4.

Чтобы изменить форму спрайта и некоторые его параметры можно воспользоваться Sprite Editor.

В ОГЛАВЛЕНИЕ

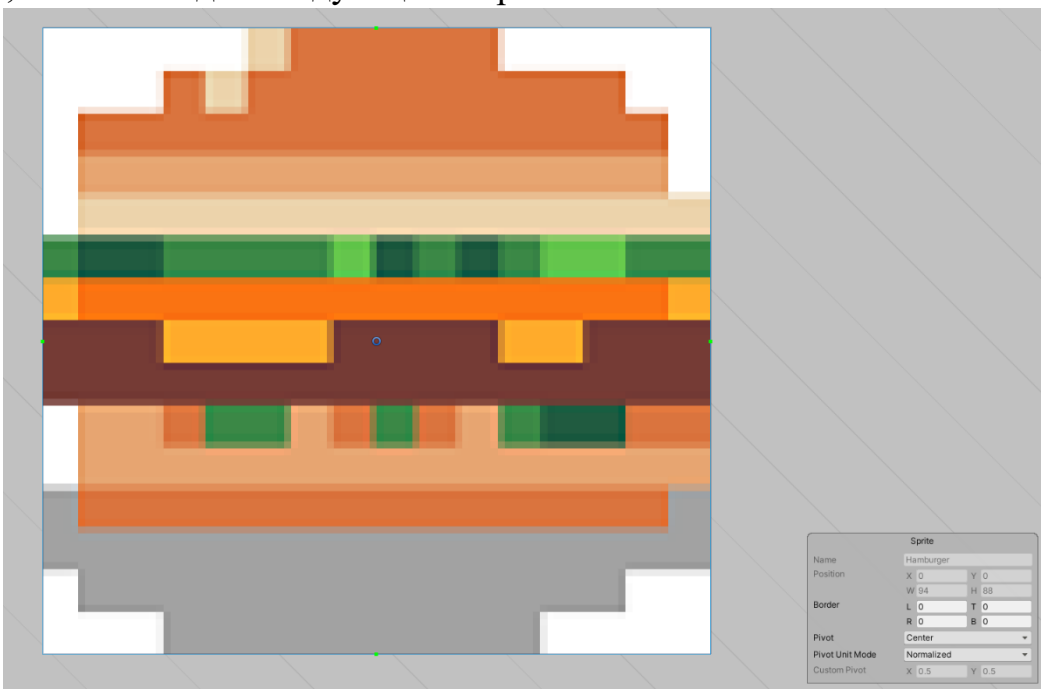
SPRITE EDITOR

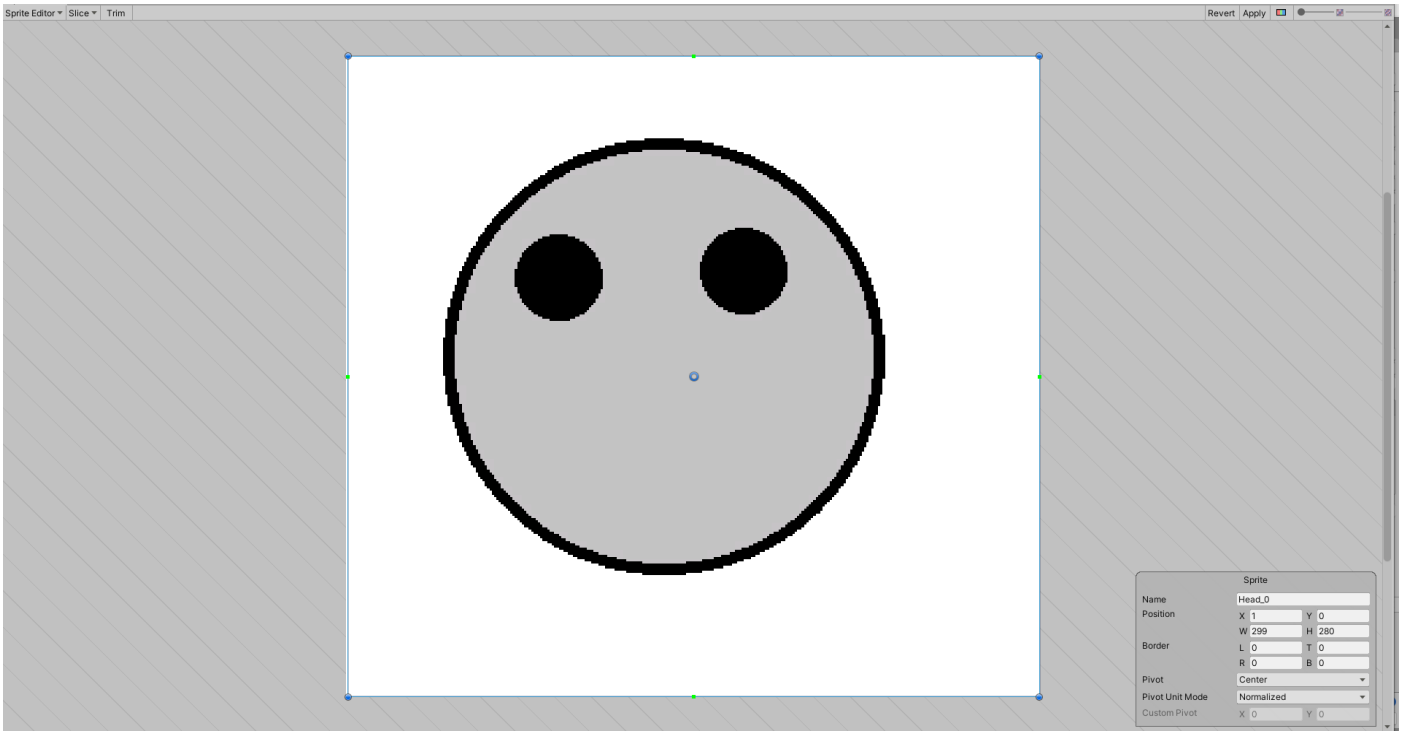
Чтобы открыть Sprite Editor щелкните по ассету в окне Project внизу. В окне инспектора справа появится кнопка Sprite Editor.



Чаще всего спрайт – это одно изображение, но иногда спрайт содержит несколько элементов, которые каким-либо образом связаны друг с другом и объединены в одно изображение. Для работы с такими спрайтами в Unity есть специальный редактор.

Если для спрайта установлен режиме Sprite Mode – Single, часть функций редактора недоступна, и он выглядит следующим образом:





Меню в правом верхнем углу:

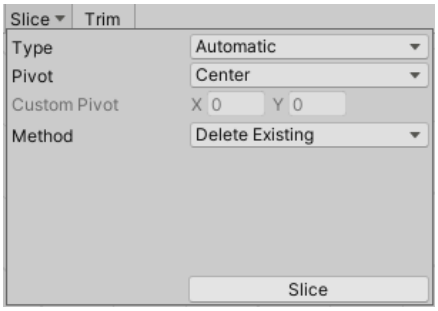
- Revert – отменить изменения;
- Apply – сохранить изменения;
- Переключатель в виде цветных полосок – переключение между цветным и альфа отображением изображения;
- Ползунок для изменения масштаба отображения (приближения);
- Ползунок переключения mipmaps (если они были созданы в настройках спрайта).

Меню в правом нижнем углу:

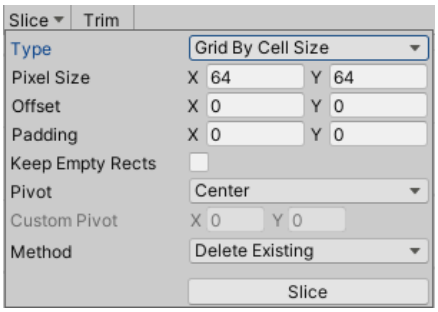
- Name – название спрайта (редактируется только для каждого отдельного спрайта, когда включен режим Multiple);
- Position – позиция спрайта: по X, по Y, ширина, высота (редактируется только для каждого отдельного спрайта, когда включен режим Multiple). Может редактироваться прямо на изображении (отображается небольшими синими кругами);
- Border – границы спрайта: слева, справа, сверху, снизу (используется для изменения размера спрайтов в UI-элементах). Может редактироваться прямо на изображении (отображается небольшими зелёными квадратами);
- Pivot – точка спрайта, которая является центром локальной системы координат. Может быть установлен вручную прямо на изображении (отображается иконкой), тогда автоматически включится режим Custom;
- Custom pivot – редактируется, если Pivot задан как Custom.

Меню в левом верхнем углу (доступно только, когда включен режим Multiple) используется для нарезки спрайтов:

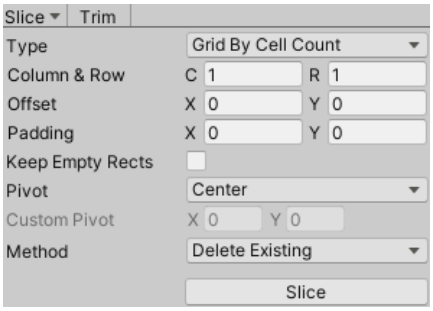
- Trim – подравнивание выделенного спрайта;
- Slice – нарезка спрайта:



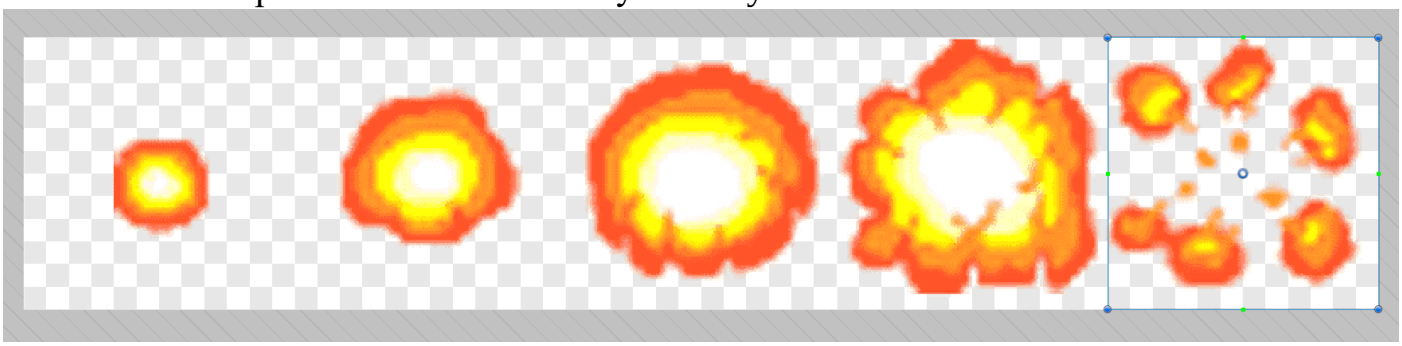
Автоматическая нарезка спрайтов. Для каждого спрайта устанавливается Pivot. Выпадающий список Method позволяет: Delete Existing (удалить существующие спрайты и нарезать новые), Smart (сохранить или отредактировать существующие спрайты и нарезать новые), Safe (сохранить существующие спрайты без изменения и нарезать новые).



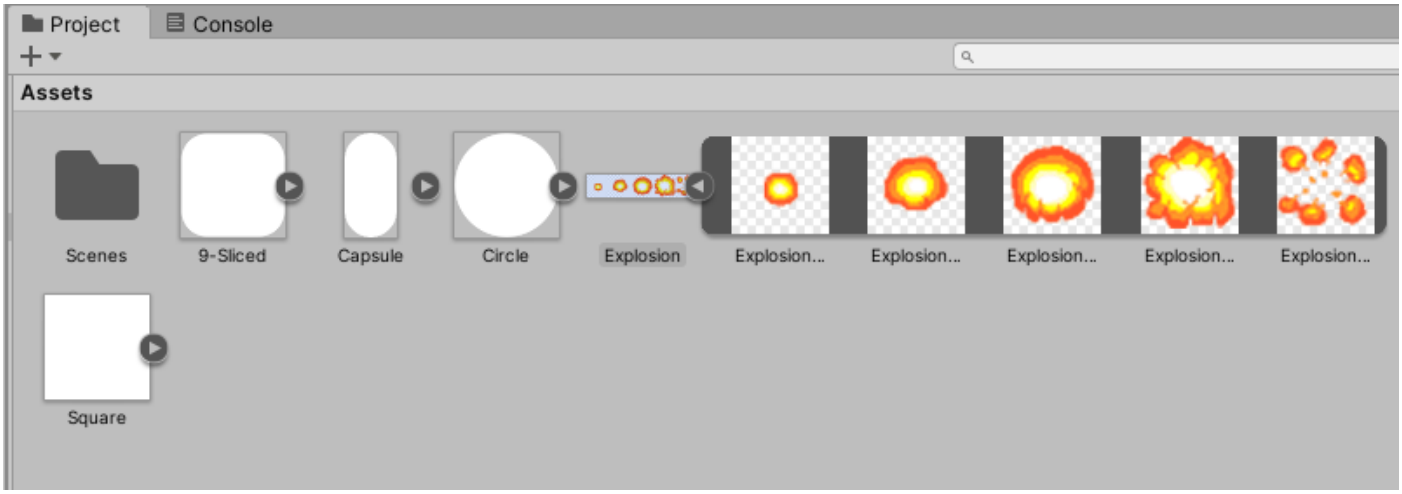
Нарезка спрайтов в виде решётки с заданным размером (Pixel Size) ячеек. Также задается смещение относительно верхнего левого угла (Offset), откуда начинается решётка и расстояние между ячейками решетки (Padding). Для каждого спрайта устанавливается Pivot.



Нарезка спрайтов в виде решетки с заданным количество столбцов (Column) и строк (Row). Размер ячеек определяется автоматически, исходя из размеров спрайта. Остальные настройки аналогичны типу Grid By Cell Size.



После того как спрайт нарезан и все изменения сохранены, при его открытии в папке проекта он будет отображаться следующим образом:



На сцене мы можем использовать каждый из этих спрайтов по отдельности. Часто такие последовательности спрайтов используются в спрайтовой анимации.

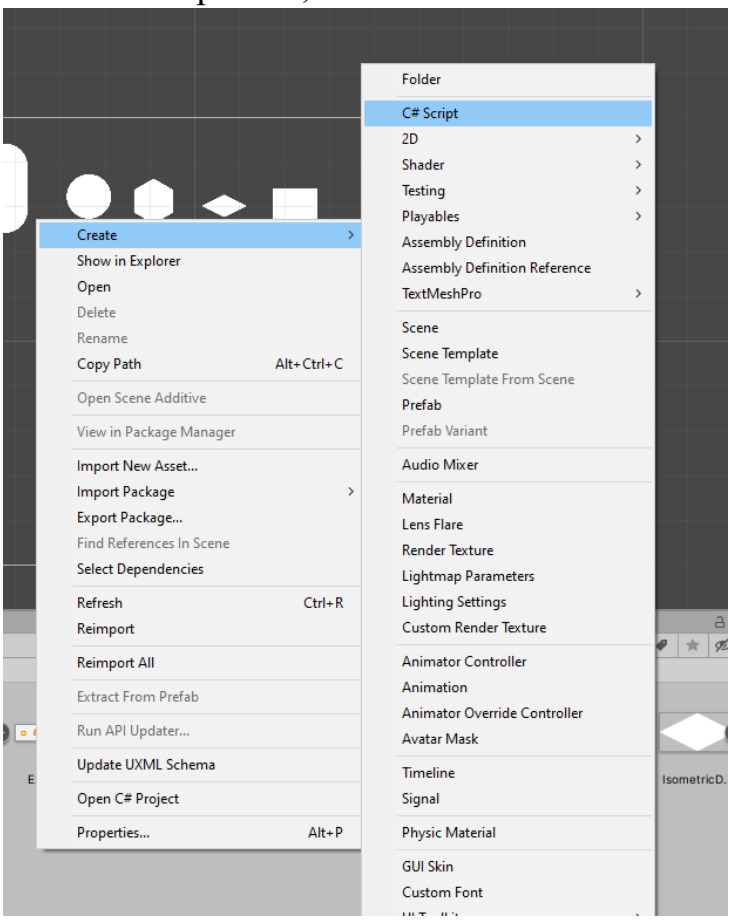
В ОГЛАВЛЕНИЕ

СКРИПТЫ

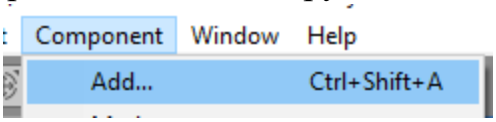
ОСНОВНЫЕ МЕТОДЫ КЛАССА MONOBEHAVIOUR

MonoBehaviour – класс, на базе которого (наследуются) по умолчанию создаются все скрипты в Unity. Скрипты создаются через:

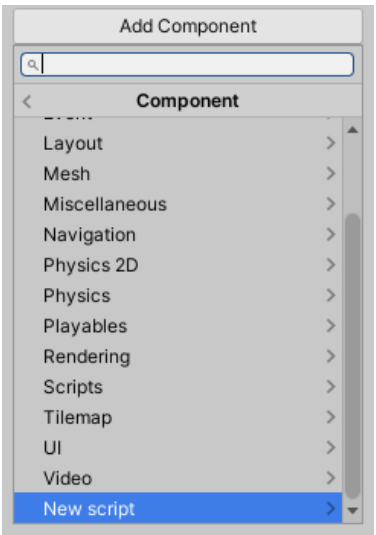
- контекстное меню в окне проекта;



- меню Component на верхней панели инструментов

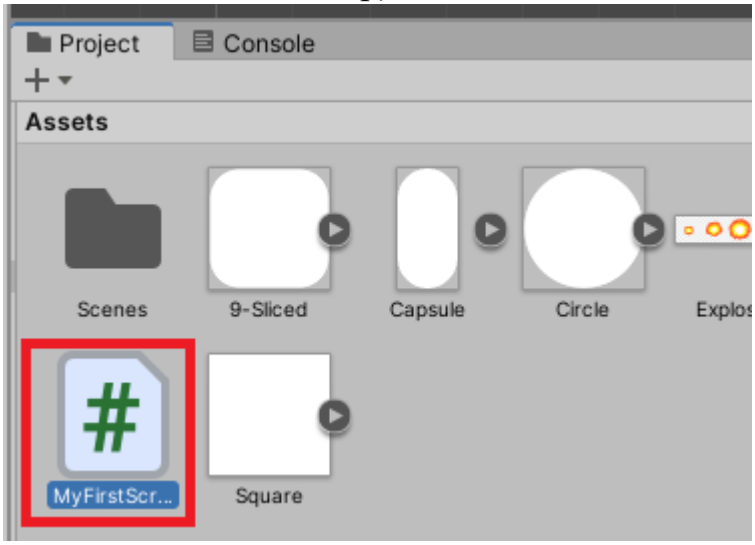


- при помощи кнопки Add Component в окне инспектора (когда выделен какой-либо объект)

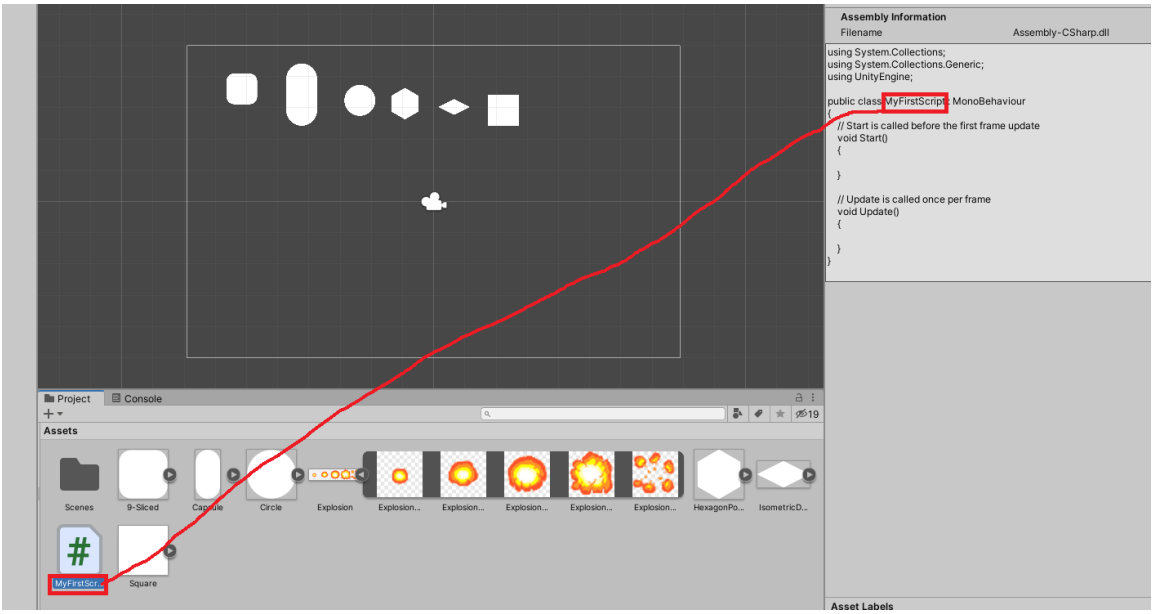


Скрипт работает только тогда, когда он является компонентом объекта на сцене, причём некоторые функции (методы) скрипта могут срабатывать даже тогда, когда сам скрипт выключен (неактивен).

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 или MonoDevelop). 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 (метода) – Start() и Update(), но существуют и другие функций, которые могут быть использованы как унаследованные от MonoBehaviour. Также можно создавать свои функции.



По умолчанию скрипт выглядит следующим образом:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyFirstScript : MonoBehaviour{
 // Start is called before the first frame update
 void Start(){

 }
 // Update is called once per frame
 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.

```
void Awake() {

}
```

START()

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

```
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.).

```
void FixedUpdate(){

}
```

UPDATE()

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

```
void Update(){

}
```

Для того, 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 для того, чтобы упорядочить выполнение действий в скрипте. Например, если камера следует за объектом, движение которого реализовано в Update() функции.

```
void LateUpdate(){

}
```

В ОГЛАВЛЕНИЕ

РАБОТА С ОБЪЕКТАМИ И КОМПОНЕНТАМИ

ПОЛУЧЕНИЕ ДОСТУПА

Доступ к игровому объекту на сцене:

- ключевое слово gameobject (если скрипт является компонентом необходимого игрового объекта);

```
void Start(){
gameObject.SetActive(false); // Данный объект становится неактивным
}
```

- ссылка на другой игровой объект;

```
public GameObject anotherObject;
void Start(){
anotherObject.SetActive(false); // Переданный в скрипт объект становится неактивным
}
```

- через transform (как к объекту родителю или ребенку);

```
void Start(){
transform.parent.gameObject.SetActive(false); // Родительский объект данного объекта становится неактивным (при этом вся иерархия объектов также становится неактивной)
}
void Start(){
transform.GetChild(0).gameObject.SetActive(false); // Нулевой ребенок данного объекта становится неактивным
}
```

- поиск игрового объекта по имени (операция, требующая больших ресурсов, лучше стараться её избегать);

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

- поиск игрового объекта по тегу;

```
void Start(){
/*
Первый найденный объект с тэгом "Player" становится неактивным,
также существует метод поиска всех объектов с заданным тегом "FindGameObjectsWithTag"
*/
GameObject.FindGameObjectWithTag("Player").SetActive(false);
}
```

После получения доступа к объекту, можно получить доступ к компоненту:

- через метод GetComponent<>

```
void Start(){
gameObject.GetComponent<BoxCollider>().enabled = false; // Выключение коллайдера для данного объекта
}
void Start(){
Destroy(gameObject.GetComponent<AudioSource>()); // Уничтожение компонента AudioSource для данного объекта
}
```

ВЕКТОРЫ

При разработке игр часто приходится использовать такой математический объект, как вектор – направленный отрезок прямой. В Unity векторы используются для позиции объекта, направления движения и различных расчетов.

Определение длины вектора:

```
public Vector2 vector2D;
public Vector3 vector3D;
void Start(){
print(vector2D.magnitude);
print(vector3D.magnitude);
}
```

Расстояние между двумя объектами (позиция каждого объекта – это вектор). Если необходимо определить только отношение расстояний, то лучше использовать sqrMagnitude:

```
public Vector3 p1, p2;
private float distance;
void Start(){
distance = Vector3.Distance(p1, p2);
print(distance);
distance = (p1 - p2).magnitude;
print(distance);
distance = Mathf.Sqrt((p1 - p2).sqrMagnitude);
print(distance);
}
```

Скалярное произведение – скаляр (число) для векторов A и B равное (Ax*Bx)+(Ay*By)+(Az*Bz). Если скалярное произведение равно нулю, то данные вектора перпендикулярны друг другу.

```
public Vector3 p1, p2;
void Start(){
print(Vector3.Dot(p1, p2));
}
```

Векторное произведение – вектор, перпендикулярный двум исходным. Компоненты результирующего вектора C определяются следующим образом Cx = Ay*Bz – Az*By, Cy = Az*Bx – Ax*Bz, Cz = Ax*By – Ay*Bx.

```
public Vector3 p1, p2;
void Start(){
print(Vector3.Cross(p1, p2));
}
```

В ОГЛАВЛЕНИЕ

ПЕРЕМЕЩЕНИЕ ОБЪЕКТОВ

Так как мы фактически работаем в трехмерном пространстве, наши gameobject'ы имеют transform.position, который является структурой vector3. Рассмотрим варианты изменения позиции объекта (не связанные с физикой):

- иерархия объектов. При перемещении родительского объекта все объекты, для которых он является родителем, будут изменять свою позицию соответственно.
- изменяем позицию через компонент Transform (простое, но далеко не оптимальное решение, так как операция new выполняется каждый кадр):

```
int speed = 1;
void Update()
{
transform.position += new Vector3(Time.deltaTime * speed, 0, 0);
}
```

- метод Translate. В параметрах указывается вектор направления движения или координаты по отдельности, а последним параметром (необязательный) – относительного чего происходит движение (локальная система координат, глобальная система координат, другой объект – например, камера). По умолчанию задана локальная система координат.

```
void Update()
{
// Движение объекта вперед со скоростью 1 единица в секунду
transform.Translate(Vector3.forward*Time.deltaTime);
}
void Update()
{
// Движение объекта вперед в глобальной системе координат
transform.Translate(Vector3.forward*Time.deltaTime, Space.World);
}
void Update()
{
// Движение объекта вдоль оси Z
transform.Translate(0, 0, Time.deltaTime);
}
void Update()
{
// Движение объекта вперед относительно камеры
transform.Translate(Vector3.forward*Time.deltaTime, Camera.main.transform);
}
```

- метод MoveTowards. Движение объекта по прямой до указанной позиции с заданной скоростью

```
public Transform target;
float speed = 1;
void Update()
{
transform.position = Vector3.MoveTowards(transform.position, target.position,
speed*Time.deltaTime);
}
```

- Vector3.Lerp() – линейная интерполяция между двумя векторами, метод, не относящийся напрямую к движению объекта, но часто использующийся для этой цели.

В ОГЛАВЛЕНИЕ

QUATERNION И ПОВОРОТ ОБЪЕКТОВ

В Unity для представления поворота объекта используются кватернионы, они сложны для простого интуитивного понимания, но достаточно просты в использовании. Для создания 2D-игры кватернионы будут встречаться реже чем в 3D, но всё таки они тоже играют важную роль. Большую часть необходимых задач, связанных с поворотом объекта, решают следующие методы:

- Quaternion.Identity – значение, которое соответствует отсутствию поворота

```
void Start()
{
   transform.rotation = Quaternion.identity;
}
```

- Quaternion.LookRotation – метод для поворота объекта в сторону позиции заданного объекта (цели).

```
public Transform target;
   void Update()
   {
      Vector3 relativePos = target.position - transform.position;
      Quaternion rotation = Quaternion.LookRotation(relativePos);
      transform.rotation = rotation;
   }
```

- Quaternion.Angle – метод, возвращающий значение угла между двумя объектам.

```
public Transform target;
void Update()
{
   float angle = Quaternion.Angle(transform.rotation, target.rotation);
   print(angle);
}
```

- Quaternion.Euler – метод, принимающий в качестве параметров вектор или три компонента вектора (x, y, z), и возвращающий значение в виде кватерниона.

```
void Start()
   {
      transform.rotation = Quaternion.Euler(0, 30, 0);
   }
```

- Quaternion.Slerp – сферическая интерполяция между двумя значениями, т.е. возвращаемое значение изменяется с заданной скоростью от одного значения (первый параметр) до другого (второй параметр).

```
public Transform from;
public Transform to;
void Update()
{
   transform.rotation = Quaternion.Slerp(from.rotation, to.rotation, Time.time);
}
```

- Quaternion.FromToRotation – создаёт поворот от вектора, заданного первым параметром, к вектору, заданного вторым.

```
void Start()
  {
     transform.rotation = Quaternion.FromToRotation(Vector3.up, Vector3.right);
  }
```

- Метод Rotate. Вращение объекта. В параметрах указывается вектор, вокруг которого происходит вращение или его координаты по отдельности, второй параметр (необязательный) – скорость поворота, последний параметр

(необязательный) – в какой системе координат происходит поворот. По умолчанию задана локальная система координат.

```
void Update()
  {
     transform.Rotate(Vector3.up * Time.deltaTime);                    // Вращение объекта со скоростью 1 градус в секунду вокруг вектора, направленного вверх для объекта, в локальной системе координат

  }

void Update()
  {
       transform.Rotate(Vector3.up * Time.deltaTime , Space.World); // Вращение объекта вокруг вектора, направленного вверх от объекта, в глобальной системе координат
  }
void Update()
    {
     transform.Rotate(Vector3.up, Time.deltaTime);                     // Скорость поворота можно задать отдельным параметром

  }

void Update()
  {
     transform.Rotate(Time.deltaTime * 10, 0, 0);                      // Координаты вектора заданы по отдельности
  }
```

- Метод RotateTowards – возвращает вектор, направленный в сторону объекта «цели». Первый параметр – текущее направление вектора, второй – необходимое направление, третий – скорость поворота, четвертый – максимальная длина вектора. Для наглядности используется метод Debug.DrawRay(), который отрисовывает результирующий вектор.

```
public Transform target;
public float speed = 1;
  void Update()
  {
     Vector3 targetDir = target.position - transform.position;

       Vector3 newDir = Vector3.RotateTowards(transform.forward, targetDir, speed * Time.deltaTime, 0.0F);
     Debug.DrawRay(transform.position, newDir, Color.red);
     transform.rotation = Quaternion.LookRotation(newDir);
  }
```

- Метод RotateAround – может изменять как позицию, так и поворот объекта. Первый параметр – точка относительно которой происходит поворот, второй – ось вращения, третий – скорость.

```
public Transform target;
void Update()
{
   transform.RotateAround(target.position, Vector3.right, 10 * Time.deltaTime);
}
```

- Метод LookAt – поворот объекта в направлении «цели», заданной в качестве параметра. Также есть скрытый параметр – это вектор, указывающий направление оси вверх для текущего объекта, по умолчанию он соответствует глобальной оси Y. Данный метод часто используется для камеры.

```
public Transform target;
void Update()
{
   transform.LookAt(target);
}
```

В ОГЛАВЛЕНИЕ

СОЗДАНИЕ И УНИЧТОЖЕНИЕ ОБЪЕКТОВ

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

```
Instantiate(prefab, Vector3.zero, Quaternion.identity);
```

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

```
void Start () {

        Destroy(gameObject, 3);

        }
```

В ОГЛАВЛЕНИЕ

ТРИГГЕРЫ И КОЛЛИЗИИ

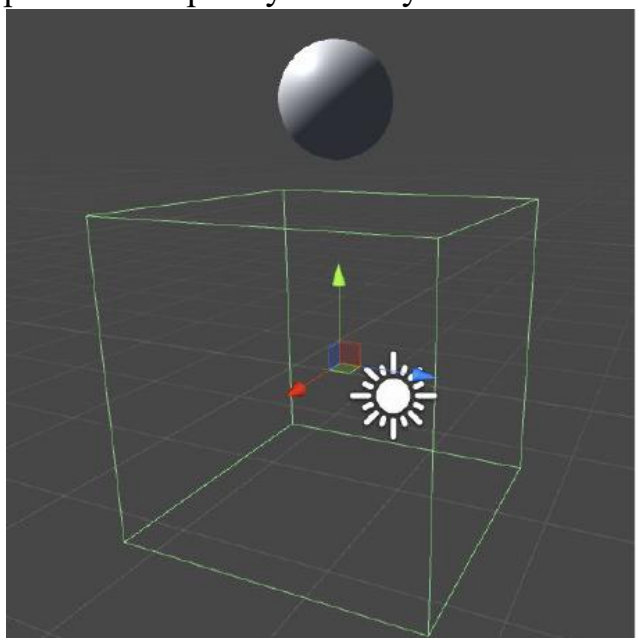
При взаимодействии объектов между собой в зависимости от объектов, их компонентов и характера взаимодействия, возможно регистрировать и обрабатывать различные события.

СОБЫТИЯ, СВЯЗАННЫЕ С ТРИГГЕРАМИ

Данные события происходят тогда, когда к объекту прикреплен коллайдер, который является триггером. Также хотя бы у одного из объектов должен быть компонент Rigidbody.

- OnTriggerEnter() – происходит один раз в момент, когда объект касается объекта с триггером.
- OnTriggerStay() – происходит каждый кадр, пока объект находится внутри триггера другого объекта.
- OnTriggerExit() – происходит один раз в момент, когда объект выходит из триггера другого объекта.

В примере ниже на сцене находятся два объекта: первый – пустой объект с компонентом BoxCollider, который является триггером; второй – сфера с добавленным компонентом Rigidbody. Скрипт прикреплен к первому объекту.



```
void OnTriggerEnter()
{
   print("Enter");
}
void OnTriggerStay()
{
   print("Stay");
}
void OnTriggerExit()
{
   print("Exit");
}
```

Также из событий можно получить информацию об объекте, с которым произошло взаимодействие.

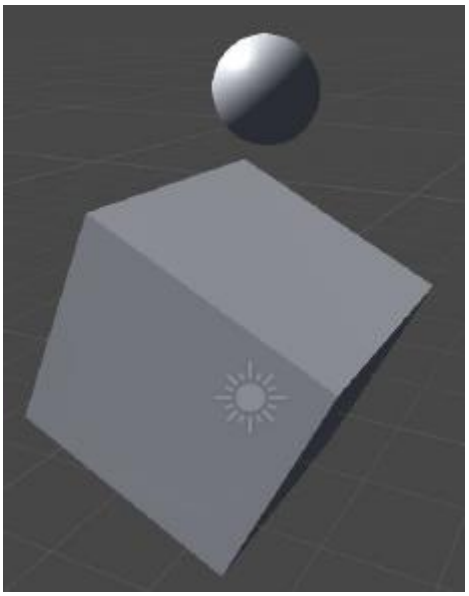
```
void OnTriggerEnter(Collider coll)
{
   print("Enter "+coll.name);
}
```

СОБЫТИЯ, СВЯЗАННЫЕ С КОЛЛИЗИЯМИ

Данные события происходят тогда, когда оба объекта имеют коллайдеры. Также хотя бы у одного из объектов должен быть компонент Rigidbody.

- OnCollisionEnter() – происходит один раз, в момент касания объектов;
- OnCollisionStay() – происходит каждый кадр, пока объекты касаются друг друга;
- OnCollisionExit() – происходит один раз в момент, когда касание объектов заканчивается.

В примере ниже на сцене находятся два объекта: первый – куб со скриптом и остальными компонентами по умолчанию; второй – сфера с добавленным компонентом Rigidbody.



```
void OnCollisionEnter(Collision collision)
{
    print("Collision enter");
}
void OnCollisionStay(Collision collision)
{
    print("Collision stay");
}
void OnCollisionExit(Collision collision)
{
    print("Collision exit");
}
```

Доступ к информации об объекте, с которым произошло взаимодействие, осуществляется аналогично.

В ОГЛАВЛЕНИЕ

INPUT

Вы можете считывать ввод с клавиатуры прямым способом через условие, есть несколько способов:

- Input.GetKey – напрямую считывает с клавиатуры вводимые данные.
- Input.GetKey – возвращает true пока нажата указанная клавиша.
- Input.GetKeyDown – единожды возвращает true, если указанная клавиша была нажата.
- Input.GetKeyUp – единожды возвращает true, если указанная клавиша была отпущена.

```
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");
```

Для того 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 – 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 определённой мыши.

- Input.GetMouseButton – Возвращает true пока нажата указанная кнопка мыши.
- Input.GetMouseButtonDown – Возвращает true когда отпущена указанная кнопка мыши.
- Input.GetMouseButtonUp – Единожды возвращает true когда нажата указанная кнопка мыши.

```
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 находится во вкладке Edit > Project Settings > Input.
Подробнее про применение Input Manager в Unity можно почитать здесь.

В ОГЛАВЛЕНИЕ

TAGS & LAYERS

Теги используются для обозначения каких-либо особых игровых объектов, например, главная камера, игрок и т.д. Тэг для объекта выбирается в выпадающем списке ниже имени объекта в окне инспектора. Существует набор тэгов (они не отображаются в меню Tags & Layers) по умолчанию, которые можно использовать.
После добавления нового тэга в меню необходимо выделить объект и назначить ему тэг.
В скриптах тэги используются следующим образом

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

В Unity два вида слоев: Layers и Sorting Layers. Первый используются для группировки объектов определённых категорий, например, окружающая среда, декорации, интерфейс, противники и т.д. Второй используется 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 изображения в слое, когда в одном слое несколько изображений).

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.

В ОГЛАВЛЕНИЕ

КЛАССЫ MATHF, RANDOM, INVOKE

MATHF

Класс предоставляет набор математический функций. Примеры наиболее часто используемых из них:

- Перевод градусов в радианы.

```
public float deg = 30.0F;
void Start()
{
    float rad = deg * Mathf.Deg2Rad;
    print(deg + " градусов = " + rad + " радиан");
}
```

- Значение числа ПИ.

```
void Start()
{
    print("Число ПИ = " + Mathf.PI);
}
```

- Возвращение модуля числа.

```
void Start()
{
    print("Модуль числа -10 = " + Mathf.Abs(-10));
}
```

- Возвращение косинуса угла, заданного в радианах.

```
void Start()
{
    print("Косинус угла 60 градусов = " + Mathf.Cos(Mathf.Deg2Rad*60));
}
```

- Ограничение возможного значения числа

```
void Update()
{
    transform.position = new Vector3(Mathf.Clamp(Time.time, 1.0F, 5.0F), 0, 0);
}
```

- Возведение в степень.

```
void Start()
{
    print(Mathf.Pow(5, 2));
}
```

- Линейная интерполяция, функция возвращает промежуточное значение между двух величин на основании значения третьего параметра.

```
void Start()
{
    print(Mathf.Lerp(10f, 30f, 0.5f));
}
```

- Возвращение максимального среди заданного набора чисел.

```
void Start()
{
    print(Mathf.Max(1, 4, 6, 2, 0, 9, 5));
}
```

RANDOM

Класс для генерации случайных чисел.

- Возвращает случайное число между двумя заданными (границы включаются).

```
void Start()
{
   print(Random.Range(4.6f, 12.3f));
}
```

- Возвращает случайное число от 0.0 до 1.0.

```
void Start()
{
   print(Random.value);
}
```

- Возвращает случайное число внутри единичной сферы.

```
void Start()
{
   print(Random.insideUnitSphere);
}
```

INVOKE, INVOKEREPEATING, CANCELINVOKE

Invoke – функция, которая позволяет вызвать другую функцию через заданное время (один раз).
InvokeRepeating – функция, которая позволяет вызывать другую функцию через заданный промежуток времени (периодически).
CancelInvoke – функция для остановки Invoke и InvokeRepeating.

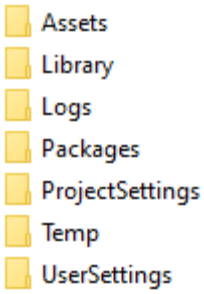
```
  void OnGUI()
  {
     if (GUILayout.Button("Start Invoke"))
     {
        Invoke("MyInvokeFunction", 1f);
     }
     if (GUILayout.Button("Start InvokeRepeating"))
     {
        InvokeRepeating("MyInvokeRepeatingFunction", 1f, 2f);
     }
     if (GUILayout.Button("Cancel Invoke"))
     {
            CancelInvoke(); // Останавливает все запущенные через Invoke функции, также можно
указать конкретную функцию, которую необходимо остановить
     }
  }
  private void MyInvokeFunction()
  {
     print("Invoke");
  }
  private void MyInvokeRepeatingFunction()
  {
     print("RepeatingInvoke "+Time.time);
  }
```

Не вошло в данный файл: корутины.

В ОГЛАВЛЕНИЕ

СТРУКТУРА ПРОЕКТА

При создании нового проекта Unity создаёт достаточно много папок и файлов, но основные из них – это Assets и ProjectSettings, остальные генерируются на основе этих двух.



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 предполагают, что необходимый для них ресурс находится в папке Assets и не требуют явного указания расположения.
- Editor – все скрипты, размещённые в этой папке (или подпапке), расцениваются как скрипты редактора, т.е. эти скрипты предназначены для расширения функционала редактора и недоступны во время выполнения проекта.
- Editor Default Resources – папка, которая должна быть размещена в корневой папке (Assets) и содержащиеся в ней ассеты, использует функция EditorGUIUtility.Load, которая загружает файлы по мере надобности.
- Gizmos – дополнительные изображения для иконки-маркера объектов на сцене размещаются в этой папке.
- Plugins – различные расширения (модули, dll) для Unity располагаются в этой папке.
- 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 Unity.
- Pro Standard Assets – стандартные ассеты Unity для версии Unity Pro.
- WebPlayerTemplates – папка для хранения шаблонов, по которому будет оформлена страница для размещения web-плеера при сборки версии игры под web-плеер.

В ОГЛАВЛЕНИЕ

ПРИЛОЖЕНИЕ 1. ИГРА 2D ROGUELIKE (РОГАЛИК)

ОБ ИГРЕ

Roguelike («rogue-подобные» (игры), сленг «рогалик») – жанр компьютерных игр. Характерными особенностями классического roguelike являются генерируемые случайным образом уровни, пошаговость и необратимость смерти персонажа – в случае его гибели игрок не может загрузить игру и должен начать её заново. Многие roguelike выполнены в декорациях эпического фэнтези под сильным влиянием настольных ролевых игр наподобие Dungeons & Dragons.

Основные условия игры:

- Игра должна быть пошаговой, нажатие одной клавиши на устройстве ввода должно соответствовать одному действию и одному ходу.
- Игровые уровни должны генерироваться случайным образом, будучи уникальными для каждого прохождения.
- Игра должна содержать «перманентную смерть», не позволяя игроку продолжить прохождение после гибели персонажа.
- Игра должна предоставлять игроку не какой-то единый линейный путь, а свободу со множеством вариантов прохождения.
- Игрок должен самостоятельно исследовать найденные предметы и открывать их свойства.

В предлагаемой реализации игры каждый уровень игры будет иметь новую карту с новым случайным расположением еды, врагов и стен. С увеличением сложности количество врагов будет логарифмически увеличиваться.

Игрок начинает игру с количеством очков, равным 100. Монстры будут отнимать 10 очков у игрока при столкновении, еда и питье будут прибавлять 10 или 20 очков. Игрок имеет возможность ломать стены, затрачивая на это определенное количество ходов.

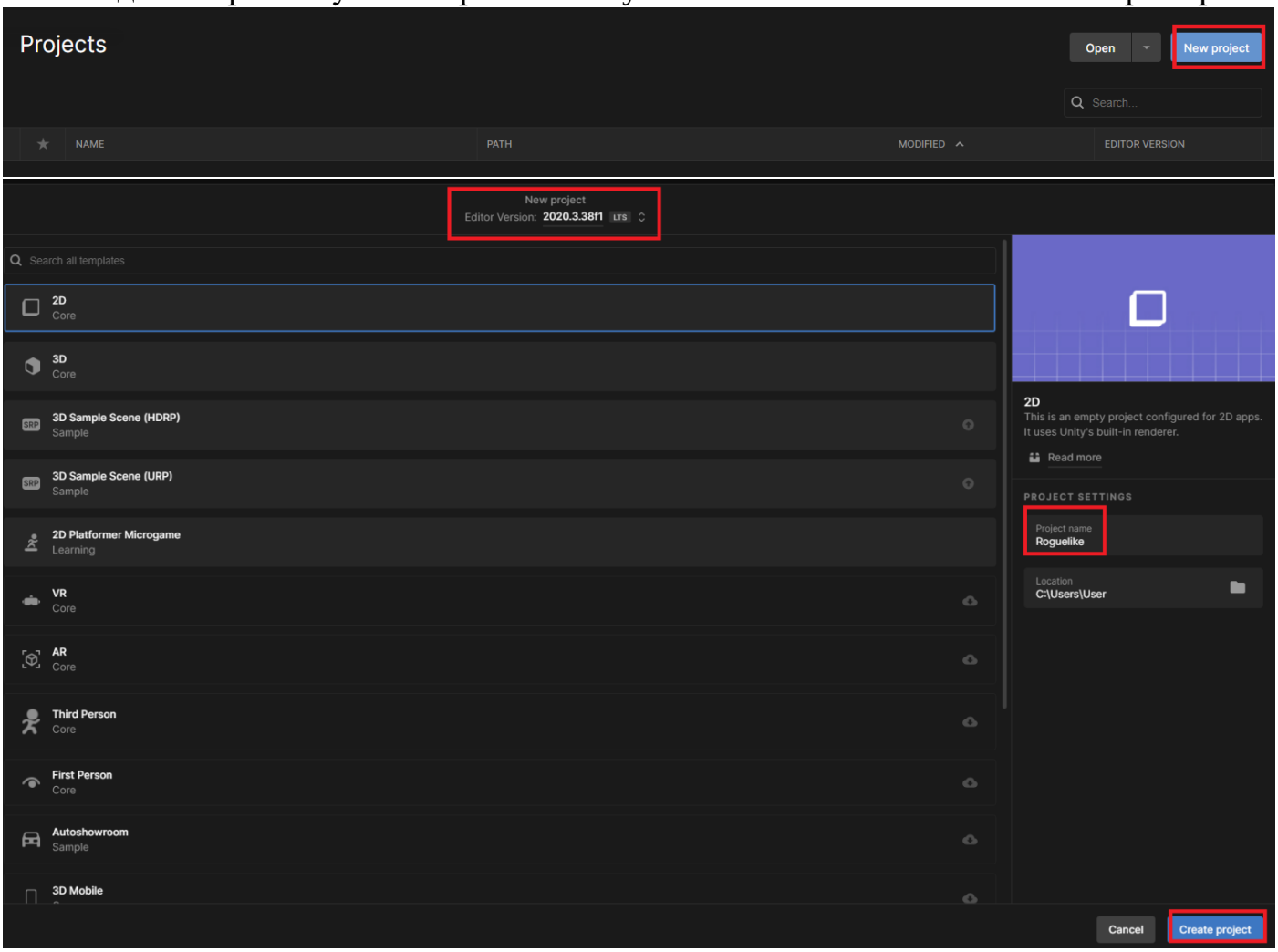
Количество очков игрока будет сохраняться при переходе от уровня к уровню, тем самым заставляя его собирать еду и питье на карте.
Монстры стараются двигаться в сторону игрока, совершая один шаг за несколько шагов игрока. Каждый шаг игрока отнимает у него 1 очко.
Количество уровней бесконечно. Игра будет окончена, когда число очков игрока станет меньше 0. Чтобы перейти на следующий уровень игроку необходимо дойти до квадрата с надписью Exit. Ходить можно только по четырем направлениям, по диагонали движение запрещено.

В ОГЛАВЛЕНИЕ

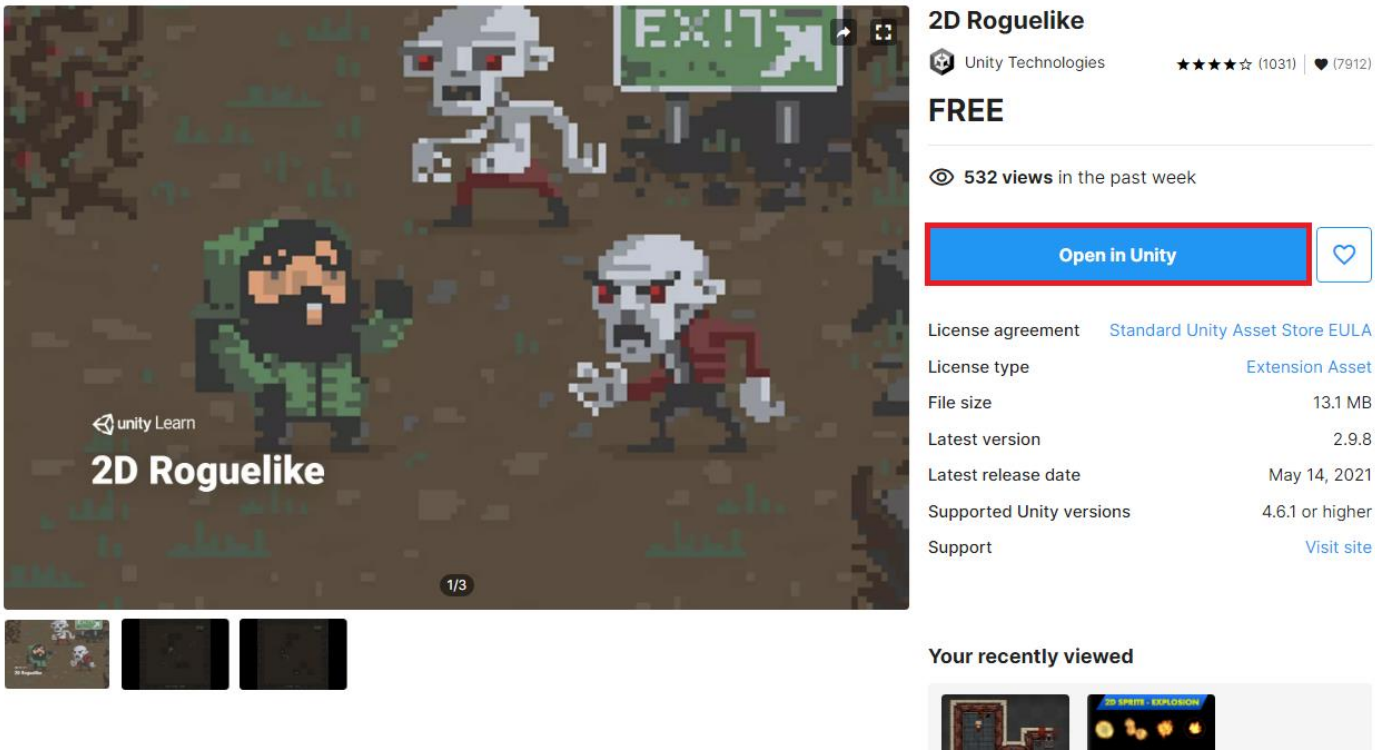
ПОДГОТОВКА К РАБОТЕ

Ваша задача – создать игру по туториалу, размещенному на сайте Unity.
Основная сложность – отличие версии Unity Editor в туториале от вашей.
Ассет для создания игры расположен на этой странице. Необходимо импортировать его в ваш проект (подсказки есть в туториале).

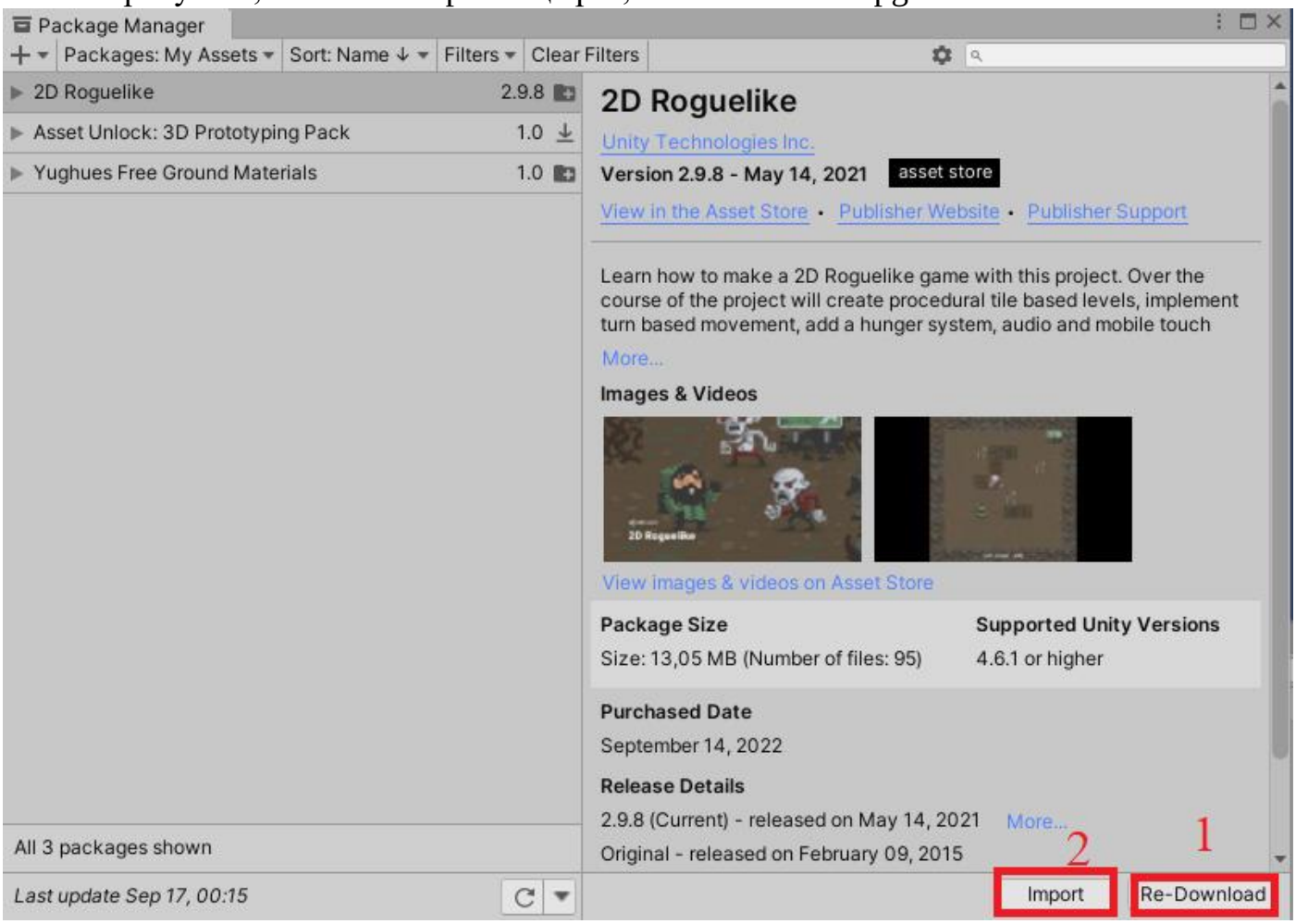
1. Первым делом необходимо залогиниться на сайте Unity. Если вы до сих пор не зарегистрировались, сделайте это сейчас.
2. Зайдите в Unity Hub на вашем компьютере и создайте пустой 2D проект. После создания проекта у вас откроется Unity Editor. Этот может занять некоторое время.



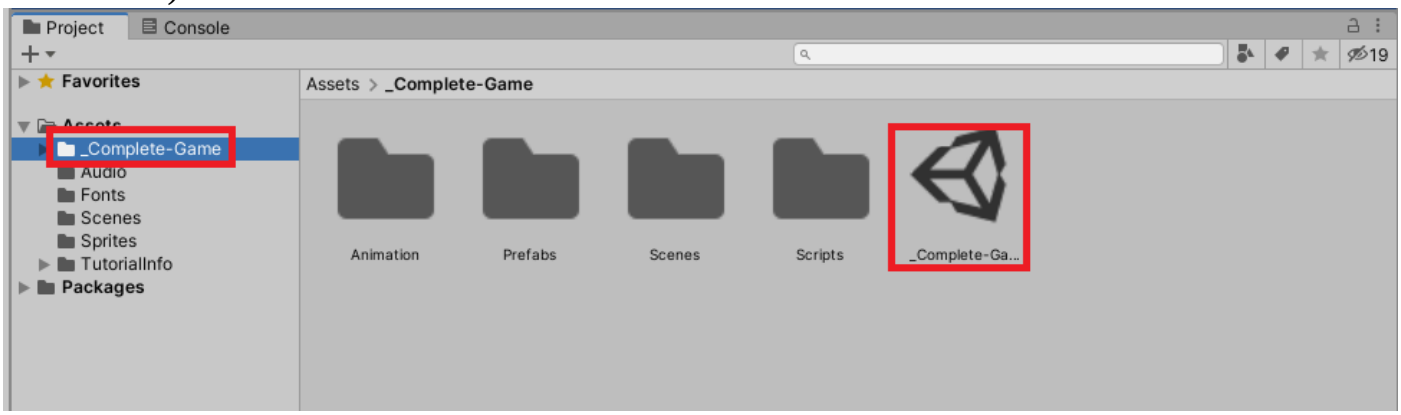
3. Открываем страницу с ассетом.
4. При необходимости нажимаем **Get asset** (или что-то подобное), а затем **Open in Unity**.



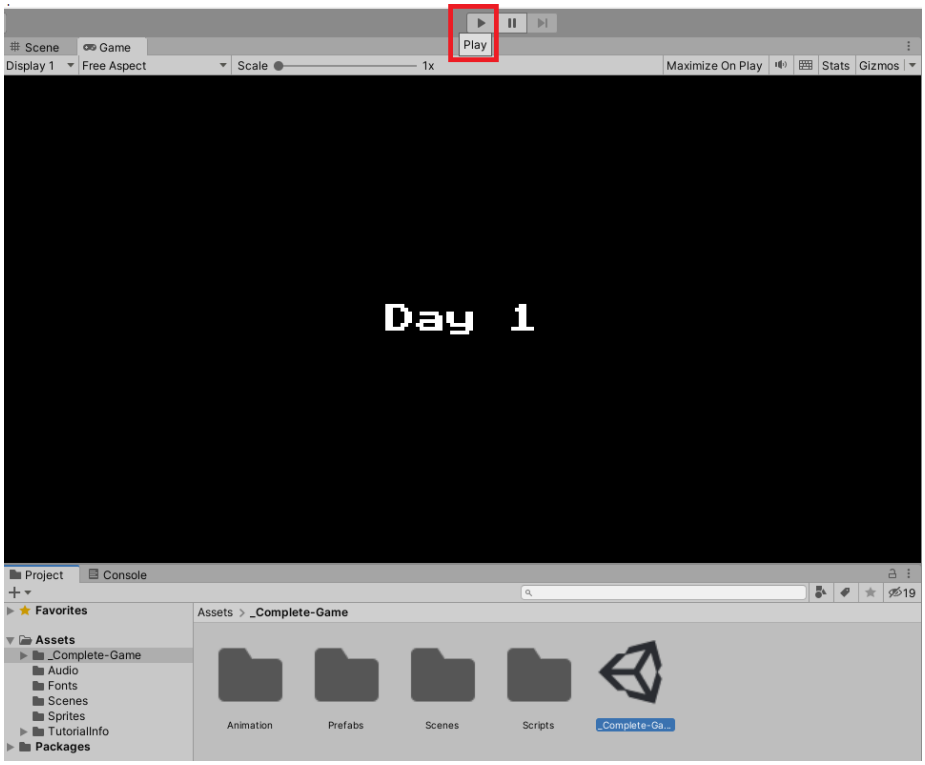
5. Возможно, браузер попросит вас открыть приложение Unity Editor, согласитесь. В окне Unity Editor появится Package Manager. Вам необходимо будет нажать на кнопку Download в правом нижнем углу, а затем на кнопку Import. Если потребуется, нажмите Import еще раз, а затем Install/Upgrade.



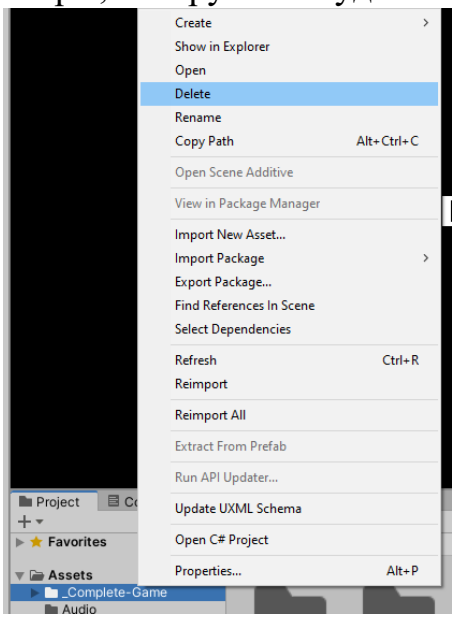
6. В окне Project внизу появится много папок с материалами и даже готовый вариант игры, который у вас должен получиться в ходе работы (хранится в папке Complete-Game).



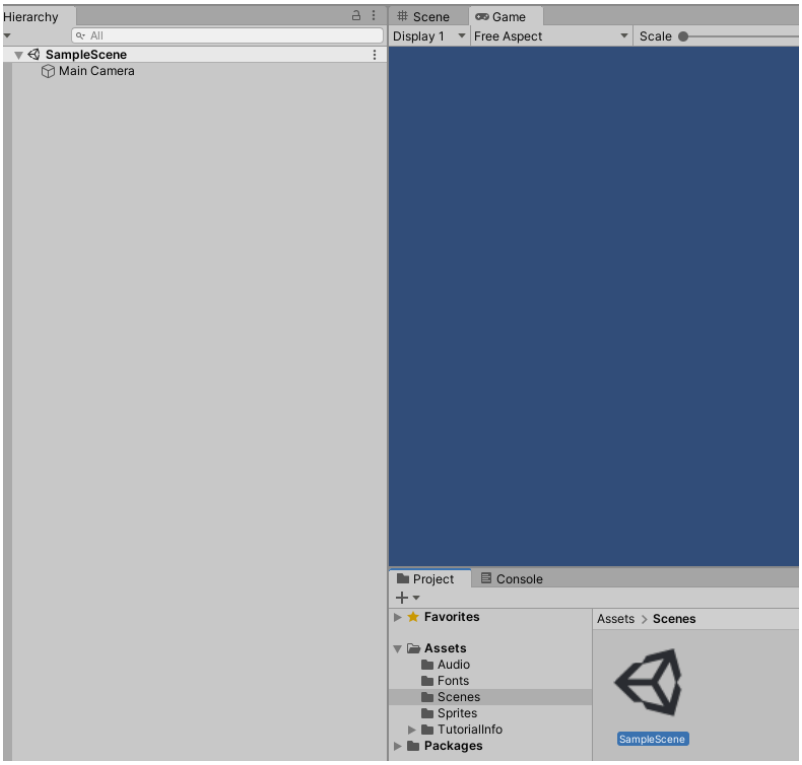
7. Чтобы ознакомиться с ожидаемым результатом нажмите по папке Complete-Game, а затем щелкните 2 раза ЛКМ по файлу, отмеченному на скриншоте выше. У вас появится сцена, можно нажать на треугольник для запуска игры. Остановка игры – тоже треугольник.



8. После остановки игры обязательно удалите папку Complete-Game. Она может повлиять на компиляцию игры, которую вы будете создавать в дальнейшем.



9. Чтобы сбросить «остатки» сцены с образцом игры можете зайти в папку Scenes и щелкнуть 2 раза ЛКМ по лежащей там SampleScene. При необходимости нажмите Don’t Save.

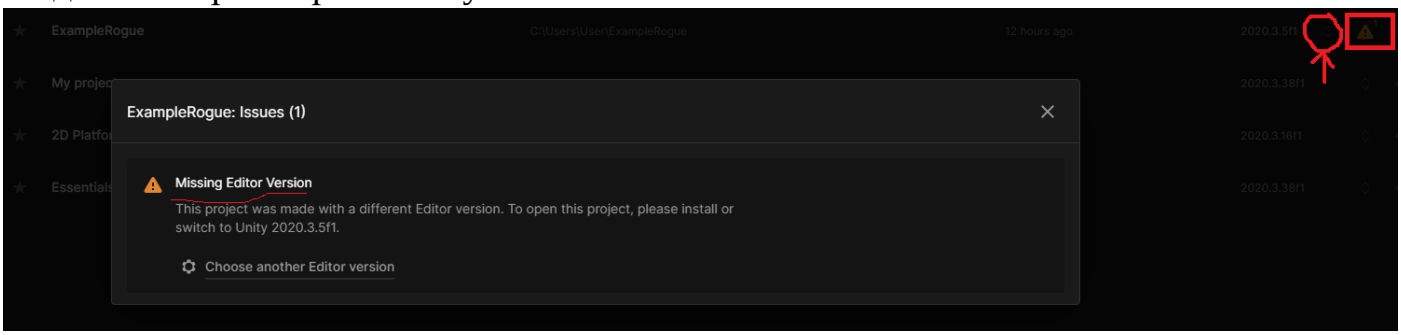


Можно приступать к знакомству с видеоматериалами из туториала. Всего в этом туториале 14 видеозаписей. Для создания полноценной игры достаточно ознакомиться с 13/14.

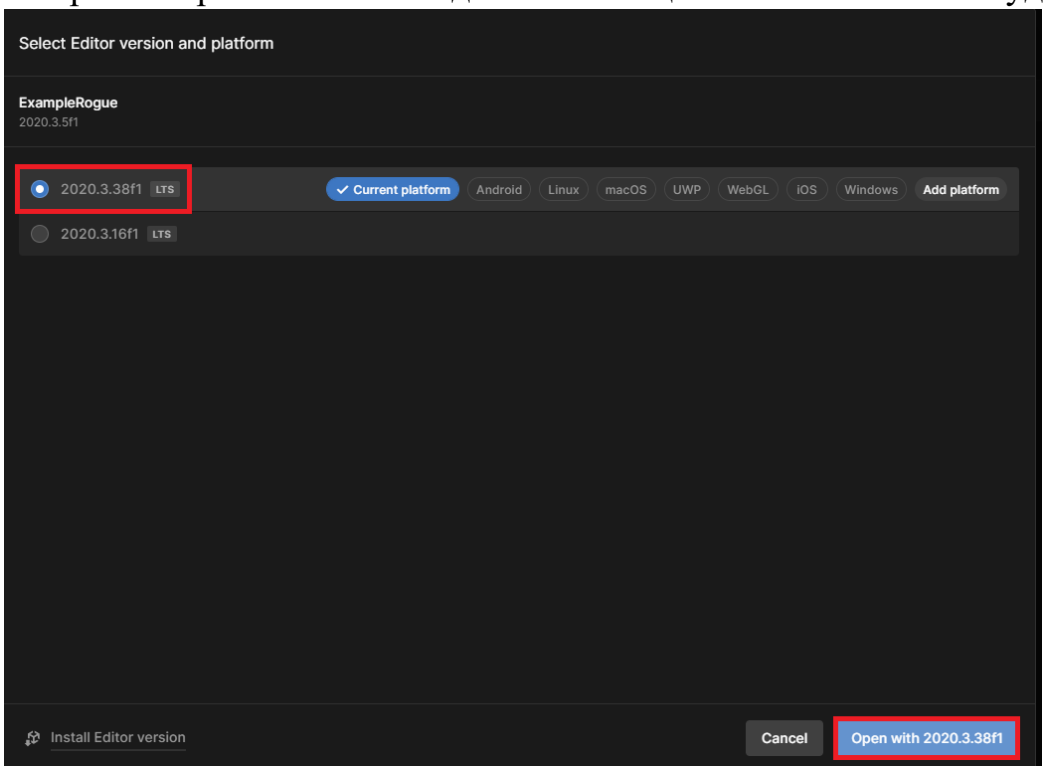
В ОГЛАВЛЕНИЕ

ПРЕДОСТЕРЕЖЕНИЯ

После закрытия игры в Unity Editor в списке проектов из Unity Hub напротив данного проекта вы скорее всего увидите восклицательный знак, сообщающий о том, что игра создана в старой версии Unity Editor.



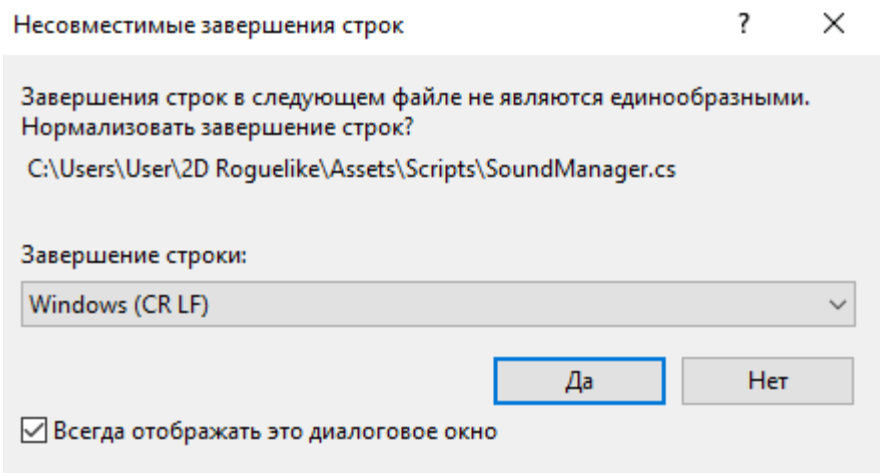
Не обращайте внимания на эту надпись, просто выберите вашу рабочую версию для повторного открытия проекта. Больше данное сообщение появляться не будет.



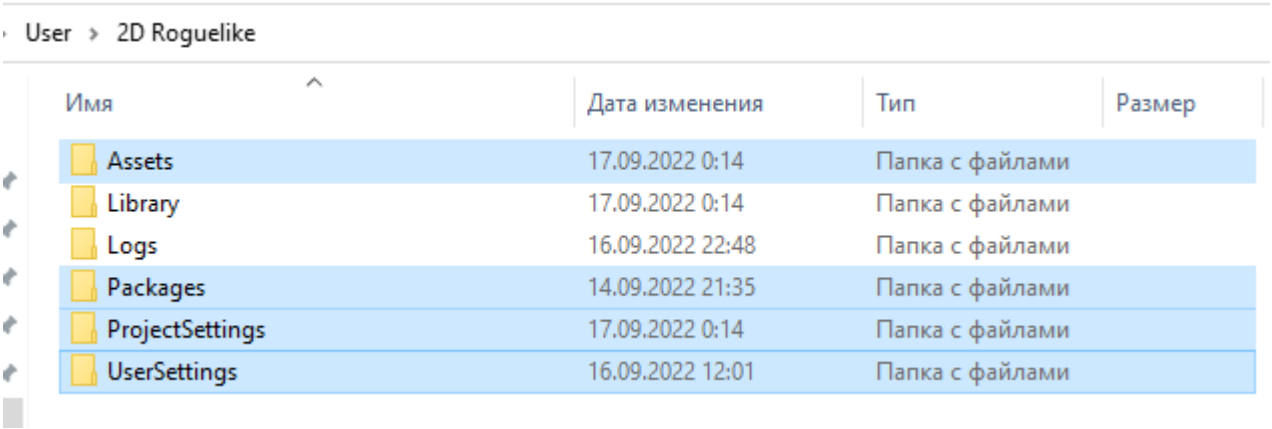
Поскольку версия из туториала отличается от вашей, у вас могут возникнуть (и обязательно возникнут) затруднения.

Основные трудности и особенности:

1. Вместо среды программирования MonoDevelop у вас будет Visual Studio.
2. Не рекомендуется писать комментарии в коде на русском языке.
3. Не копируйте код из сторонних источников. Дело в том, что у автора Mac, а это значит, что там другие окончания строк, о чем вам обязательно сообщит VS при копировании (проблема нормализации строк).



4. Много изменений в коде из-за более новой версии. Подсказки для контроля даны ниже.
5. Список действительно нужных папок, которые необходимо оставить в самом конце работы над проектом, представлен на скриншоте ниже (выделены синим).



В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: BOARD MANAGER

```
using UnityEngine;
using System;
//Allows us to use Lists.
using System.Collections.Generic;
//Tells Random to use the Unity Engine random number generator.
using Random = UnityEngine.Random;

public class BoardManager : MonoBehaviour{
    // Using Serializable allows us to embed a class with sub properties in the
inspector.
    [Serializable]
    public class Count
    {
        public int minimum;             //Minimum value for our Count class.
        public int maximum;             //Maximum value for our Count class.

        //Assignment constructor.
        public Count(int min, int max)
        {
            minimum = min;
            maximum = max;
        }
    }

    public int columns = 8;             //Number of columns in our game board.
    public int rows = 8;                //Number of rows in our game board.
    //Lower and upper limit for our random number of walls per level.
    public Count wallCount = new Count(5, 9);
    //Lower and upper limit for our random number of food items per level.
    public Count foodCount = new Count(1, 5);
    public GameObject exit;             //Prefab to spawn for exit.
    public GameObject[] floorTiles;     //Array of floor prefabs.
    public GameObject[] wallTiles;      //Array of wall prefabs.
    public GameObject[] foodTiles;      //Array of food prefabs.
    public GameObject[] enemyTiles;     //Array of enemy prefabs.
    public GameObject[] outerWallTiles; //Array of outer tile prefabs.

    //A variable to store a reference to the transform of our Board object.
    private Transform boardHolder;
    //A list of possible locations to place tiles.
    private List<Vector3> gridPositions = new List<Vector3>();

    //Clears our list gridPositions and prepares it to generate a new board.
    void InitialiseList()
    {
        //Clear our list gridPositions.
        gridPositions.Clear();

        //Loop through x axis (columns).
        for (int x = 1; x < columns - 1; x++)
        {
            //Within each column, loop through y axis (rows).
```

```
            for (int y = 1; y < rows - 1; y++)
            {
                //At each index add a new Vector3 to our list with the x and y
coordinates of that position.
                gridPositions.Add(new Vector3(x, y, 0f));
            }
        }
    }


    //Sets up the outer walls and floor (background) of the game board.
    void BoardSetup()
    {
        //Instantiate Board and set boardHolder to its transform.
        boardHolder = new GameObject("Board").transform;

        //Loop along x axis, starting from -1 (to fill corner) with floor or
outerwall edge tiles.
        for (int x = -1; x < columns + 1; x++)
        {
            //Loop along y axis, starting from -1 to place floor or outerwall
tiles.
            for (int y = -1; y < rows + 1; y++)
            {
                //Choose a random tile from our array of floor tile prefabs and
prepare to instantiate it.
                GameObject toInstantiate = floorTiles[Random.Range(0,
floorTiles.Length)];

                //Check if we current position is at board edge, if so choose a
random outer wall prefab from our array of outer wall tiles.
                if (x == -1 || x == columns || y == -1 || y == rows)
                    toInstantiate = outerWallTiles[Random.Range(0,
outerWallTiles.Length)];
                //Instantiate the GameObject instance using the prefab chosen
for toInstantiate at the Vector3 corresponding to current grid position in
loop, cast it to GameObject.
                GameObject instance = Instantiate(toInstantiate, new Vector3(x,
y, 0f), Quaternion.identity) as GameObject;

                //Set the parent of our newly instantiated object instance to
boardHolder, this is just organizational to avoid cluttering hierarchy.
                instance.transform.SetParent(boardHolder);
            }
        }
    }

    //RandomPosition returns a random position from our list gridPositions.
    Vector3 RandomPosition()
    {
        //Declare an integer randomIndex, set it's value to a random number
between 0 and the count of items in our List gridPositions.
        int randomIndex = Random.Range(0, gridPositions.Count);
```

```
        //Declare a variable of type Vector3 called randomPosition, set it's
value to the entry at randomIndex from our List gridPositions.
        Vector3 randomPosition = gridPositions[randomIndex];

        //Remove the entry at randomIndex from the list so that it can't be re-
used.
        gridPositions.RemoveAt(randomIndex);

        //Return the randomly selected Vector3 position.
        return randomPosition;
    }


    //LayoutObjectAtRandom accepts an array of game objects to choose from
along with a minimum and maximum range for the number of objects to create.
    void LayoutObjectAtRandom(GameObject[] tileArray, int minimum, int maximum)
    {
        //Choose a random number of objects to instantiate within the minimum
and maximum limits
        int objectCount = Random.Range(minimum, maximum + 1);

        //Instantiate objects until the randomly chosen limit objectCount is
reached
        for (int i = 0; i < objectCount; i++)
        {
            //Choose a position for randomPosition by getting a random position
from our list of available Vector3s stored in gridPosition
            Vector3 randomPosition = RandomPosition();

            //Choose a random tile from tileArray and assign it to tileChoice
            GameObject tileChoice = tileArray[Random.Range(0,
tileArray.Length)];

            //Instantiate tileChoice at the position returned by RandomPosition
with no change in rotation
            Instantiate(tileChoice, randomPosition, Quaternion.identity);
        }
    }

    //SetupScene initializes our level and calls the previous functions to lay
out the game board
    public void SetupScene(int level)
    {
        //Creates the outer walls and floor.
        BoardSetup();

        //Reset our list of gridpositions.
        InitialiseList();

        //Instantiate a random number of wall tiles based on minimum and
maximum, at randomized positions.
        LayoutObjectAtRandom(wallTiles, wallCount.minimum, wallCount.maximum);
```

```
        //Instantiate a random number of food tiles based on minimum and
maximum, at randomized positions.
        LayoutObjectAtRandom(foodTiles, foodCount.minimum, foodCount.maximum);

        //Determine number of enemies based on current level number, based on a
logarithmic progression
        int enemyCount = (int)Mathf.Log(level, 2f);

        //Instantiate a random number of enemies based on minimum and maximum,
at randomized positions.
        LayoutObjectAtRandom(enemyTiles, enemyCount, enemyCount);

        //Instantiate the exit tile in the upper right hand corner of our game
board
        Instantiate(exit, new Vector3(columns - 1, rows - 1, 0f),
Quaternion.identity);
    }
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: ENEMY

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Enemy : MovingObject
{
    public int playerDamage;

    private Animator animator;
    private Transform target;
    private bool skipMove;
    public AudioClip enemyAttack1; // added at step 13
    public AudioClip enemyAttack2; // added at step 13

    protected override void Start()
    {
        GameManager.instance.AddEnemyToList(this); // added at step 11
        animator = GetComponent<Animator>();
        target = GameObject.FindGameObjectWithTag("Player").transform;
        base.Start();
    }

    protected override void AttemptMove <T> (int xDir, int yDir)
    {
        if (skipMove)
        {
            skipMove = false;
            return;
        }

        base.AttemptMove<T>(xDir, yDir);
        skipMove = true;
    }

    public void MoveEnemy()
    {
        int xDir = 0;
        int yDir = 0;

        if (Mathf.Abs(target.position.x - transform.position.x) <
float.Epsilon)
            yDir = target.position.y > transform.position.y ? 1 : -1;
        else
            xDir = target.position.x > transform.position.x ? 1 : -1;
        AttemptMove<Player>(xDir, yDir);
    }

    protected override void OnCantMove<T> (T component)
    {
        Player hitPlayer = component as Player;

        hitPlayer.LoseFood(playerDamage); // change place
```

```
        animator.SetTrigger("enemyAttack"); // added at step 11

        SoundManager.instance.RandomizeSfx(enemyAttack1, enemyAttack2); //
added at step 13
    }
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: GAME MANAGER

```
using UnityEngine;
using UnityEngine.SceneManagement; // edited from guide pdf (2)
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;


public class GameManager : MonoBehaviour
{


    public float levelStartDelay = 2f; // added at step 12
    public float turnDelay = .1f;
    public int playerFoodPoints = 100;
    public static GameManager instance = null;
    [HideInInspector] public bool playersTurn = true;


    private Text levelText; // added at step 12
    private GameObject levelImage; // added at step 12
    public BoardManager boardScript;
    private int level = 1; // added at step 12
    private List<Enemy> enemies;
    private bool enemiesMoving;
    private bool doingSetup = true; // added at step 12

    // private bool firstLevel = true; // don't use this

    void Awake()
    {
        if (instance == null)
            instance = this;
        else if (instance != this)
            Destroy(gameObject);

        DontDestroyOnLoad(gameObject);
        enemies = new List<Enemy>();
        boardScript = GetComponent<BoardManager>();
        InitGame();
    }

    // added from forum
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
    static public void CallbackInitialization()
    {
        SceneManager.sceneLoaded += OnSceneLoaded;
    }
    static private void OnSceneLoaded(Scene arg0, LoadSceneMode arg1)
    {
        instance.level++;
        instance.InitGame();
    }
```

```
    void InitGame()
    {
        doingSetup = true; // added at step 12

        levelImage = GameObject.Find("LevelImage"); // added at step 12
        levelText = GameObject.Find("LevelText").GetComponent<Text>(); // added
at step 12
        levelText.text = "Day " + level; // added at step 12
        levelImage.SetActive(true); // added at step 12
        Invoke("HideLevelImage", levelStartDelay); // added at step 12

        enemies.Clear();
        boardScript.SetupScene(level);
    }

    // added at step 12
    void HideLevelImage() // private or not, corrected
    {
        levelImage.SetActive(false);
        doingSetup = false;
    }

    void Update()
    {
        if (playersTurn || enemiesMoving || doingSetup) // edited at step 12
            return;

        StartCoroutine(MoveEnemies());
    }

    public void AddEnemyToList(Enemy script)
    {
        enemies.Add(script);
    }

    public void GameOver()
    {
        levelText.text = "After " + level + " days, you starved."; // added at
step 12
        levelImage.SetActive(true); // added at step 12
        enabled = false;
    }

    IEnumerator MoveEnemies()
    {
        enemiesMoving = true;
        yield return new WaitForSeconds(turnDelay);
        if (enemies.Count == 0)
        {
            yield return new WaitForSeconds(turnDelay);
        }
```

```
        for (int i = 0; i < enemies.Count; i++)
        {
            enemies[i].MoveEnemy();
            yield return new WaitForSeconds(enemies[i].moveTime);
        }

        playersTurn = true;
        enemiesMoving = false;
    }

    // don't use this
    /*void OnLevelWasLoaded(int index)
    {
        level++;
        InitGame();
    }*/ // edited pdf 2

    /*void OnLevelFinishedLoading(Scene scene, LoadSceneMode mode)
    {
        if (firstLevel) // forum
        {
            firstLevel = false;
            return;
        }
        level++;
        InitGame();
    } // edited pdf 2

    void OnEnable()
    {
        SceneManager.sceneLoaded += OnLevelFinishedLoading;
    } // edited pdf 2

    void OnDisable()
    {
        SceneManager.sceneLoaded -= OnLevelFinishedLoading;
    } // edited pdf 2*/
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: LOADER

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Loader : MonoBehaviour
{
    public GameObject gameManager;
    void Awake()
    {
        if (GameManager.instance == null)
            Instantiate(gameManager);
    }
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: MOVINGOBJECT

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public abstract class MovingObject : MonoBehaviour
{
    public float moveTime = 0.1f;
    public LayerMask blockingLayer;

    private BoxCollider2D boxCollider;
    private Rigidbody2D rb2D;
    private float inverseMoveTime;
    private bool isMoving; // edit from pdf (6)

    // Start is called before the first frame update
    protected virtual void Start()
    {
        boxCollider = GetComponent<BoxCollider2D>();
        rb2D = GetComponent<Rigidbody2D>();
        inverseMoveTime = 1f / moveTime;
    }

    protected bool Move(int xDir, int yDir, out RaycastHit2D hit)
    {
        Vector2 start = transform.position;
        Vector2 end = start + new Vector2(xDir, yDir);
        boxCollider.enabled = false;
        hit = Physics2D.Linecast(start, end, blockingLayer);
        boxCollider.enabled = true;

        // if (hit.transform == null) // edit from pdf (6)
        if (hit.transform == null && !isMoving) // edit from pdf (6)
        {
            StartCoroutine(SmoothMovement(end));
            return true;
        }
        return false;
    }

    protected IEnumerator SmoothMovement(Vector3 end)
    {
        isMoving = true; // edit from pdf (6)
        float sqrRemainingDistance = (transform.position - end).sqrMagnitude;

        while (sqrRemainingDistance > float.Epsilon)
        {
            Vector3 newPosition = Vector3.MoveTowards(rb2D.position, end,
inverseMoveTime * Time.deltaTime);
            rb2D.MovePosition(newPosition);
            sqrRemainingDistance = (transform.position - end).sqrMagnitude;
            yield return null;
        }
```

```
        rb2D.MovePosition(end); // edit from pdf (6)
        isMoving = false; // edit from pdf (6)
    }

    protected virtual void AttemptMove <T> (int xDir, int yDir)
        where T: Component
    {
        RaycastHit2D hit;
        bool canMove = Move(xDir, yDir, out hit);

        if (hit.transform == null)
            return;
        T hitComponent = hit.transform.GetComponent<T>();

        if (!canMove && hitComponent != null)
            OnCantMove(hitComponent);
    }

    protected abstract void OnCantMove<T>(T component)
        where T : Component;
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: PLAYER

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI; // added at step 12
using UnityEngine.SceneManagement; // edit from pdf (9)

public class Player : MovingObject
{
    public float restartLevelDelay = 1f;
    public int pointsPerFood = 10;
    public int pointsPerSoda = 20;
    public int wallDamage = 1;
    public Text foodText; // added at step 12
    public AudioClip moveSound1; // added at step 13
    public AudioClip moveSound2; // added at step 13
    public AudioClip eatSound1; // added at step 13
    public AudioClip eatSound2; // added at step 13
    public AudioClip drinkSound1; // added at step 13
    public AudioClip drinkSound2; // added at step 13
    public AudioClip gameOverSound; // added at step 13


    private Animator animator;
    private int food;

    // private bool isMoving; // edit from pdf (9), don't use this

    // Start is called before the first frame update
    protected override void Start()
    {
        animator = GetComponent<Animator>();

        food = GameManager.instance.playerFoodPoints;

        foodText.text = "Food: " + food; // added at step 12

        base.Start();
    }

    private void OnDisable()
    {
        GameManager.instance.playerFoodPoints = food;
    }

    // Update is called once per frame
    void Update()
    {
        if (!GameManager.instance.playersTurn) return;

        int horizontal = 0;
        int vertical = 0;
```

```
        horizontal = (int) (Input.GetAxisRaw("Horizontal"));
        vertical = (int) (Input.GetAxisRaw("Vertical"));

        if (horizontal != 0)
            vertical = 0;

        if (horizontal != 0 || vertical != 0)
            AttemptMove<Wall>(horizontal, vertical);
    }

    protected override void AttemptMove <T> (int xDir, int yDir)
    {
        food--;
        foodText.text = "Food: " + food; // added at step 12

        base.AttemptMove<T>(xDir, yDir);

        RaycastHit2D hit;

        Move(xDir, yDir, out hit); // added at step 13, corrected on forum at
step 13
        if (hit.transform == null)
        {
            SoundManager.instance.RandomizeSfx(moveSound1, moveSound2);
        } // added at step 13, corrected on forum at step 13

        CheckIfGameOver();

        GameManager.instance.playersTurn = false;
    }

    protected override void OnCantMove<T>(T component)
    {
        Wall hitWall = component as Wall;
        hitWall.DamageWall(wallDamage);
        animator.SetTrigger("playerChop");
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Exit")
        {
            Invoke("Restart", restartLevelDelay);
            enabled = false;
        }
        else if (other.tag == "Food")
        {
            food += pointsPerFood;
            foodText.text = "+" + pointsPerFood + " Food: " + food; // added at
step 12
            SoundManager.instance.RandomizeSfx(eatSound1, eatSound2); // added
at step 13
            other.gameObject.SetActive(false);
```

```
        }
        else if (other.tag == "Soda")
        {
            food += pointsPerSoda;
            foodText.text = "+" + pointsPerSoda + " Food: " + food; // added at
step 12
            SoundManager.instance.RandomizeSfx(drinkSound1, drinkSound2); //
added at step 13
            other.gameObject.SetActive(false);
        }
    }

    private void Restart()
    {
        // Application.LoadLevel(Application.LoadedLevel); // edit from pdf (9)
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex,
LoadSceneMode.Single); // edit from pdf (9) + take from completed version
    }

    public void LoseFood (int loss)
    {
        animator.SetTrigger("playerHit");
        food -= loss;
        foodText.text = "-" + loss + " Food: " + food;
        CheckIfGameOver();
    }

    private void CheckIfGameOver()
    {
        if (food <= 0)
        {
            SoundManager.instance.PlaySingle(gameOverSound); // added at step
13
            SoundManager.instance.musicSource.Stop(); // added at step 13
            GameManager.instance.GameOver();
        }

    }
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: SOUNDMANAGER

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SoundManager : MonoBehaviour
{
    public AudioSource efxSource;
    public AudioSource musicSource;
    public static SoundManager instance = null;

    public float lowPitchRange = 0.95f;
    public float highPitchRange = 1.05f;

    void Awake()
    {
        if (instance == null)
            instance = this;
        else if (instance != this)
            Destroy(gameObject);

        DontDestroyOnLoad(gameObject);
    }

    public void PlaySingle(AudioClip clip)
    {
        efxSource.clip = clip;
        efxSource.Play();
    }

    public void RandomizeSfx(params AudioClip [] clips)
    {
        int randomIndex = Random.Range(0, clips.Length);
        float randomPitch = Random.Range(lowPitchRange, highPitchRange);

        efxSource.pitch = randomPitch;
        efxSource.clip = clips[randomIndex];
        efxSource.Play();
    }
}
```

В ОГЛАВЛЕНИЕ

ИТОГОВЫЙ СКРИПТ ДЛЯ КОНТРОЛЯ: WALL

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Wall : MonoBehaviour
{
    public Sprite dmgSprite;
    public int hp = 3; // or 4, wall hp, really doesn't matter
    public AudioClip chopSound1; // added at step 13
    public AudioClip chopSound2; // added at step 13


    private SpriteRenderer spriteRenderer;

    void Awake()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    public void DamageWall(int loss)
    {
        SoundManager.instance.RandomizeSfx(chopSound1, chopSound2); // added at
step 13
        spriteRenderer.sprite = dmgSprite;
        hp -= loss;
        if (hp <= 0)
            gameObject.SetActive(false);
    }
}
```

В ОГЛАВЛЕНИЕ

ПРИЛОЖЕНИЕ 2: ИГРА APPLE PICKER

Игра **Apple Picker** является аналогом известной вам игры **Ну, погоди!**.

Цель игры: сбор «падающих» сверху-вниз объектов для накапливания очков.

Увеличение сложности игры обычно заключается в изменении скорости падения объектов, а также появлении объектов, которые необходимо избегать.

Вариант игры Apple Picker, предложенный на стр. 496–534 в книге [4], является трехмерным. Падающие объекты – яблоки, которые выбрасывает яблоня. Ловить объекты необходимо в объемную корзину, управляемую пользователем.

Электронный вариант издания книги можно найти в интернете самостоятельно или на странице курса в Moodle.

В ОГЛАВЛЕНИЕ

ДОМАШНЕЕ ЗАДАНИЕ

1. Ознакомиться с информацией из данного файла.
2. Реализовать 2D игру из Приложения 1 (есть подсказки) или 3D игру из Приложения 2 (нет подсказок).

В ОГЛАВЛЕНИЕ

СПИСОК ЛИТЕРАТУРЫ

1. Официальный сайт Unity.
2. Документация по Unity 2020.3 (на русском (2020.2) и английском).
3. Туториал к 2D Rogue Like
4. Бонд Д. Г. Unity и C#. Геймдев от идеи до реализации. 2-е изд. – СПб.: Питер, 2019. – 928 с.: ил. – (Серия «Для профессионалов»).

В ОГЛАВЛЕНИЕ