

# **Лабораторная работа. Создание 2D-игры Flappy Bird.**

Данный урок создан на основе следующего видео-урока: [How to make Flappy Bird in Unity \(Complete Tutorial\)](#)

**Цель работы:** изучить процесс создания 2D-игры в Unity и освоить основные элементы разработки игровой логики, физики и пользовательского интерфейса.

**Задачи:**

## **1. Настройка окружения:**

- Запустить Unity Hub и создать новый 2D-проект.
- Настроить основные параметры сцены и объектов.

## **2. Создание игровых объектов:**

- Создать и настроить игровые объекты: "Bird", "Ground", "Pipe" с соответствующими физическими параметрами и компонентами.
- Создать границы игрового поля (пределы экрана) и задать фон сцены.

## **3. Разработка игровой логики:**

- Написать скрипт для управления птицей, включая полет и другие действия.
- Настроить поведение труб, включая их движение по экрану и реакцию на столкновения с птицей.
- Создать и настроить скрипты для подсчета очков и отображения их на экране.

## **4. Улучшение игровых механик:**

- Добавить возможность изменения скорости движения труб.
- Реализовать постепенное увеличение сложности игры, увеличение скорости движения труб с течением времени.

## **5. Интерфейс пользователя и звуковое сопровождение:**

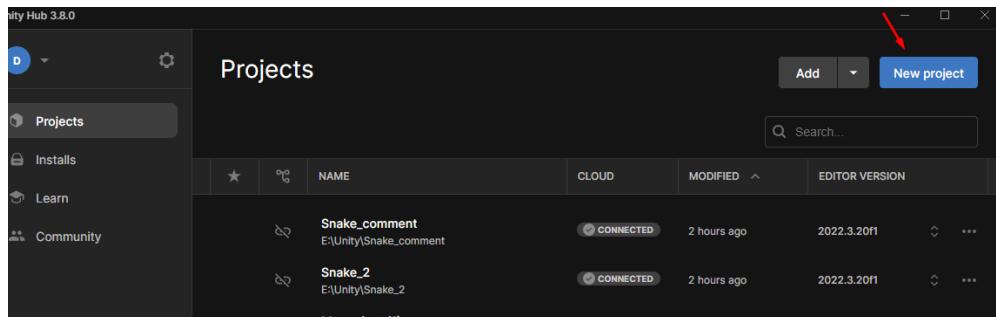
- Создать и настроить элементы интерфейса для отображения счета.
- Добавить звуковое сопровождение, включая звуки прыжков и звуки столкновений.

## **6. Финальные настройки и сборка проекта:**

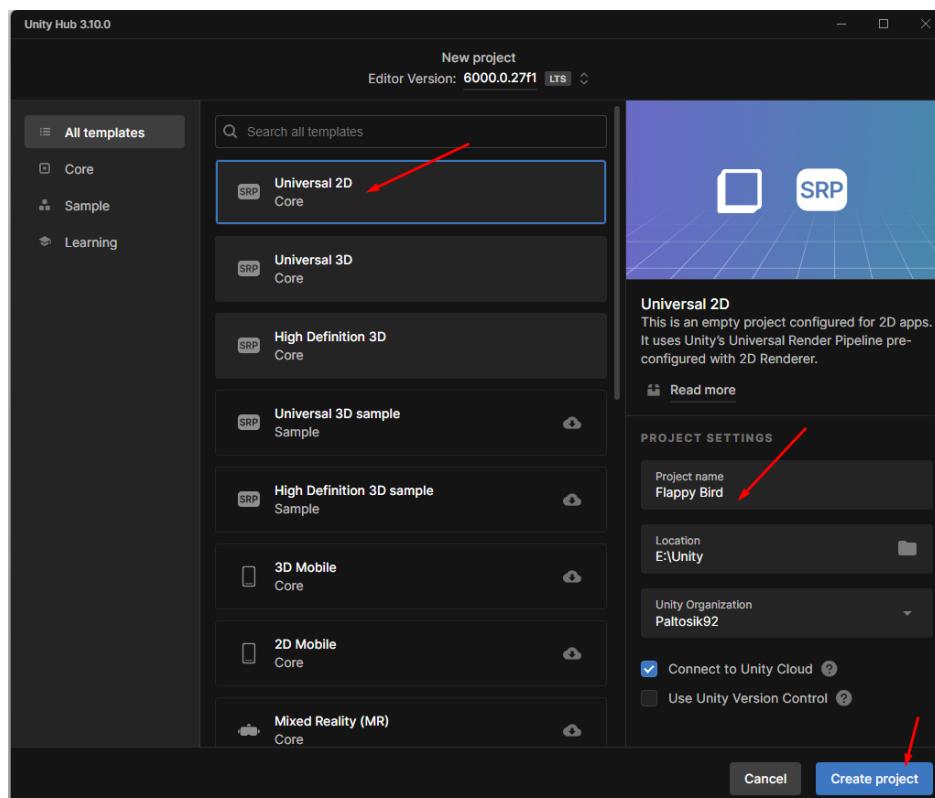
- Провести рефакторинг кода для улучшения читаемости и производительности.
- Скомпилировать и сохранить финальную версию игры.

## 1. Запускаем Unity Hub.

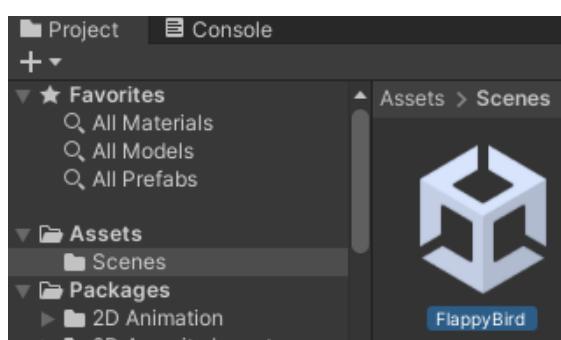
Создаём новый проект – **New project**:



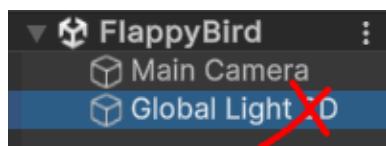
Выбираем шаблон **Universal 2D**, вводим **название** проекта, выбираем **место расположения** и нажимаем **Create project**:



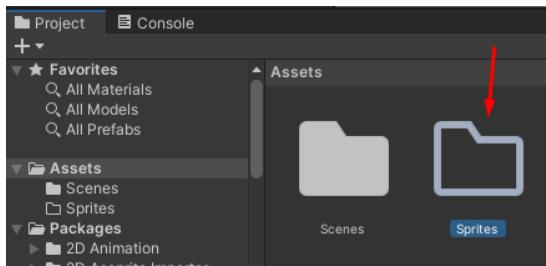
2. В папке **Scenes** переименовываем сцену, дав ей имя игры **FlappyBird**:



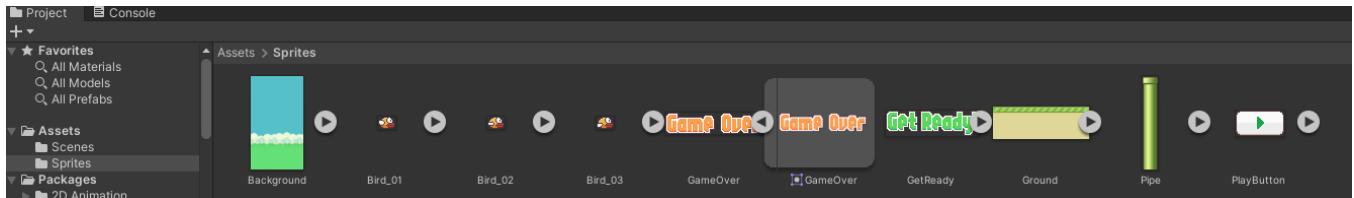
Удаляем источник света **Directional Light**, так как в 2D-игре он не нужен:



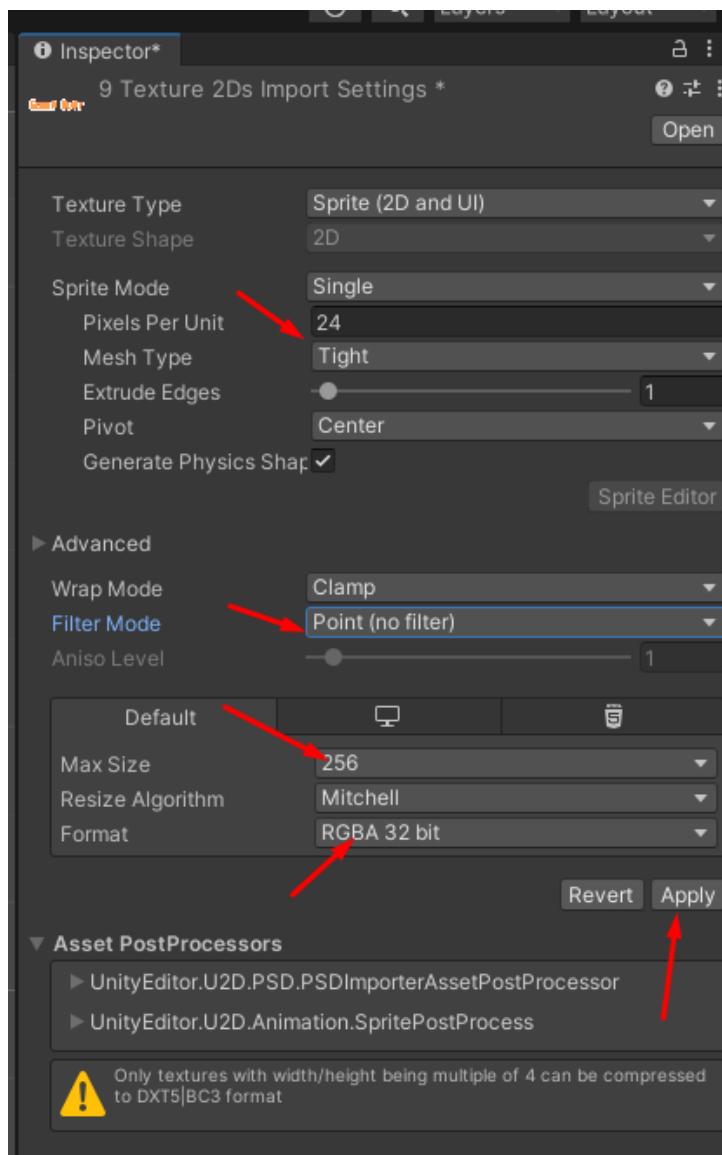
### 3. Создаём папку Sprites:



Перемещаем в неё файлы, прикрепленные к лабораторной работе:



Выделяем все спрайты (**Ctrl+A**) и изменяем их параметры:



- **Pixel Per Unit** на **24**. Пиксели на единицу — это количество пикселей ширины и высоты в изображении спрайта на единичном расстоянии в мировом пространстве. Исходные спрайты были спроектированы **24 пикселя**, на единицу измерения, поэтому мы выбираем данное значение. Установка **PPE** на **24** помогает сохранить пиксельный стиль и предотвратить размытие пикселей при масштабировании;

- режим фильтрации с билинейного меняем на точечный (**Point**), чтобы сохранить пиксельную графику без размытия;

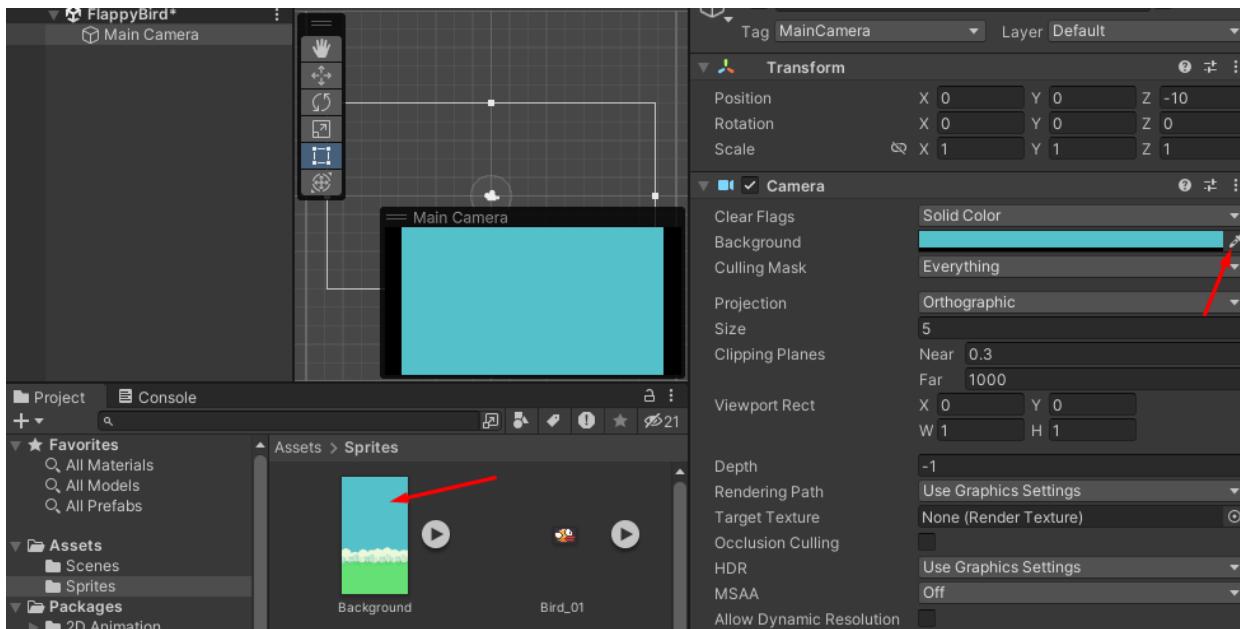
- формат меняем на **RGBA**, чтобы поддерживалась прозрачность;

- максимальный размер выбираем **256**, это позволяет оптимизировать производительность, так как текстуры меньшего размера быстрее рендерятся и занимают меньше памяти.

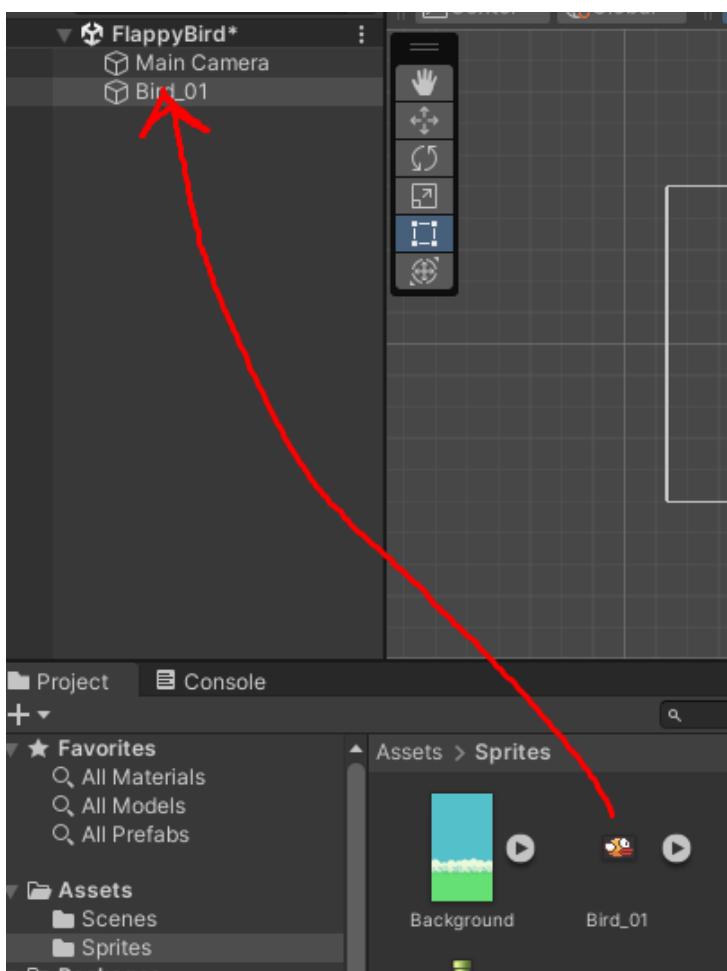
После изменения параметров нажимаем **Apply**.

4. Меняем цвет фона у камеры, чтобы он соответствовал цвету фона спрайтов.

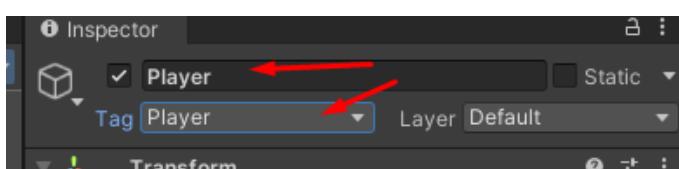
Для этого используем инструмент **пипетка**:



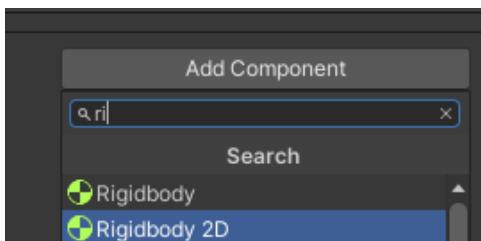
5. Перетаскиваем Bird\_01 в Иерархию (Hierarchy):



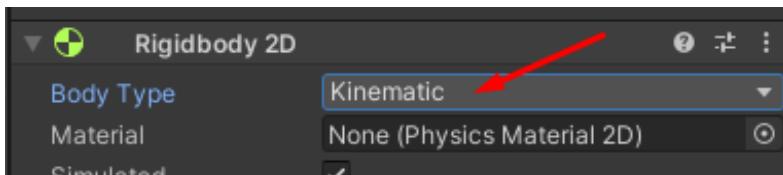
Переименовываем объект в **Player** и присваиваем ему тег **Player**:



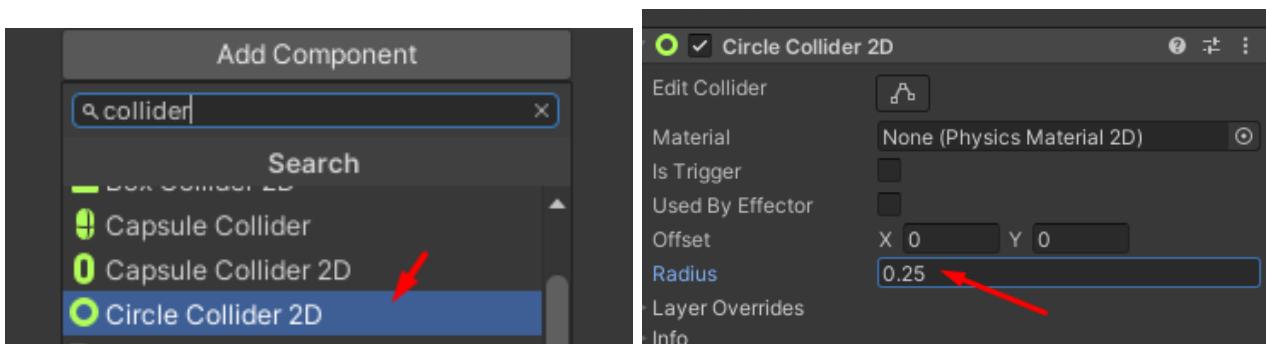
Добавляем компонент **Rigidbody 2D**:



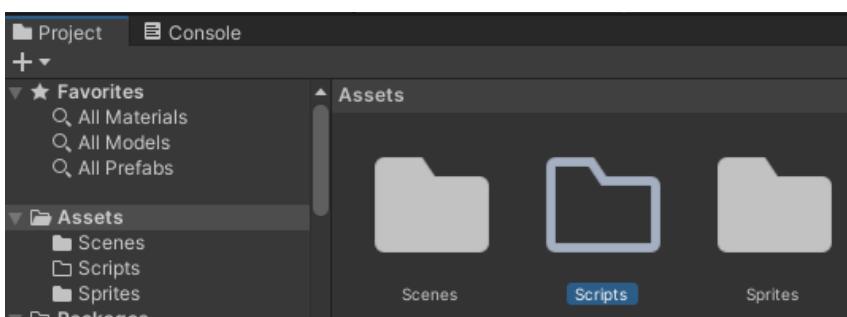
Тип меняем на **Kinematic** (в нашем случае будет простое управление и нам не нужно учитывать воздействие физики):



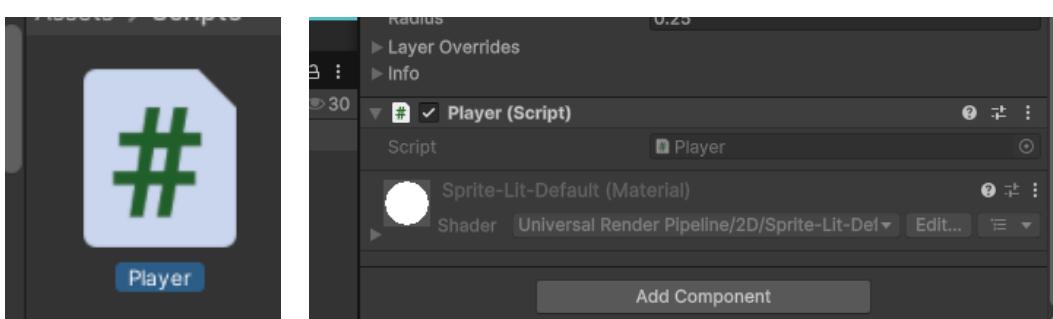
Добавляем компонент **Circle Collider 2D** и изменяем его радиус на **0.25** (подгоняем размер коллайдера под размер спрайта):



6. Теперь напишем скрипт для нашего игрока. Создаём папку **Scripts**:



Внутри создаём скрипт **Player** и переносим его на объект **Player**:



Открываем скрипт **Player** и пишем код для управления персонажем.

Вначале займёмся **полями класса** (*переменными*):

```
[Header("Настройки физики")]
// Гравитация, влияющая на игрока (ускорение вниз)
[SerializeField] private float gravity = -9.8f;
// Сила прыжка (начальная скорость вверх при прыжке)
[SerializeField] private float strength = 4f;

[Header("Настройки анимации")]
// Массив спрайтов для анимации персонажа
[SerializeField] private Sprite[] sprites;

// Текущее направление движения игрока
private Vector3 _direction;
// Компонент для отрисовки спрайтов на объекте
private SpriteRenderer _spriteRenderer;
// Индекс текущего спрайта из массива
private int spriteIndex;
```

Затем при инициализации скрипта, мы хотим получить ссылку на компонент **SpriteRenderer**, поэтому пропишем метод **Awake**:

```
// Метод Awake вызывается до Start и используется для
инициализации компонентов
private void Awake()
{
    // Получаем компонент SpriteRenderer, который отвечает
    // за отображение спрайта
    _spriteRenderer = GetComponent<SpriteRenderer>();
}
```

После мы хотим прописать методы, отвечающие за движения игрока:

```
// Метод для выполнения прыжка
private void Jump()
{
    // Устанавливаем направление движения вверх с заданной
    // силой прыжка
    _direction = Vector3.up * strength;
}
```

◆ *Как это влияет на поведение игрока?*

- Игрок сразу получает импульс вверх (вектор **\_direction** теперь указывает вверх).
- После этого гравитация начнет его снова тянуть вниз.
- Если быстро нажимать пробел, можно удерживать игрока в воздухе дольше.

```
// Метод для применения гравитации
private void ApplyGravity()
{
    // Гравитация изменяет вертикальную составляющую
    // направления со временем
    _direction.y += gravity * Time.deltaTime;
}
```

◆ *Как это влияет на поведение игрока?*

- Если игрок не нажимает пробел, гравитация начинает *уменьшать* у в `_direction`, что *замедляет подъём* и затем вызывает *падение вниз*.
- Чем дольше игрок находится в воздухе без прыжка, тем быстрее он падает.

```
// Метод для перемещения игрока
private void MovePlayer()
{
    // Изменяем позицию игрока на основе текущего
    // направления и прошедшего времени
    transform.position += _direction * Time.deltaTime;
}
```

◆ *Как это влияет на поведение игрока?*

- Игрок движется вверх (`Jump()` добавляет силу).
- Затем он начинает *замедляться*, потому что `gravity` уменьшает у в `_direction`.
- В конечном итоге игрок *начинает падать*.
- Этот процесс *повторяется* каждый раз, когда игрок нажимает пробел.

Затем вызовем их в методе **Update**:

```
// Метод Update вызывается каждый кадр
private void Update()
{
    // Проверяем, был ли нажат пробел или левая кнопка мыши
    // для прыжка
    if (Input.GetKeyDown(KeyCode.Space) ||
    Input.GetMouseButton(0))
    {
        Jump();
    }
    // Применяем гравитацию к направлению движения
    ApplyGravity();
    // Обновляем позицию игрока согласно текущему
    // направлению
    MovePlayer();
}
```

Далее реализуем метод для смены анимации:

```
// Метод для анимации смены спрайтов
private void AnimateSprite()
{
    // Переходим к следующему спрайту
    spriteIndex++;
    // Если индекс выходит за пределы массива, сбрасываем
    // его в начало
    if (spriteIndex >= sprites.Length)
    {
        spriteIndex = 0;
    }
    // Обновляем спрайт, который отображается на объекте
    _spriteRenderer.sprite = sprites[spriteIndex];
}
```

◆ *Как это влияет на поведение игрока?*

- Игрок *непрерывно меняет спрайты*, создавая эффект анимации машущих крыльев.
- Когда *spriteIndex* достигает конца массива, он *сбрасывается в начало*, создавая *зацикленный цикл анимации*.
- *Метод не вызывается вручную в Update(), а работает благодаря InvokeRepeating()*.

И вызовем его в методе **Start**:

```
// Метод Start вызывается при запуске скрипта
private void Start()
{
    // Запускаем метод AnimateSprite() повторно каждые 0.15
    // секунд для анимации
    InvokeRepeating(nameof(AnimateSprite), 0.15f, 0.15f);
}
```

◆ *Как работает InvokeRepeating()?*

- *InvokeRepeating(nameof(Метод), задержка\_перед\_первым\_вызовом,*  
*интервал\_между\_вызовами);*
- *nameof(AnimateSprite) – передаём название метода как строку (без ошибок в названии).*
- *0.15f (первый аргумент) – через 0.15 секунд после старта сцены начнётся анимация.*
- *0.15f (второй аргумент) – затем метод AnimateSprite() будет повторяться каждые 0.15 секунд.*

## 7. Итоговый код для Player:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    [Header("Настройки физики")]
    [SerializeField] private float gravity = -9.8f;
    [SerializeField] private float strength = 4f;

    [Header("Настройки анимации")]
    [SerializeField] private Sprite[] sprites;

    private Vector3 _direction;
    private SpriteRenderer _spriteRenderer;
    private int spriteIndex;

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

    private void Start()
    {
        InvokeRepeating(nameof(AnimateSprite), 0.15f, 0.15f);
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space) ||
        Input.GetMouseButtonUp(0))
        {
            Jump();
        }
        ApplyGravity();
        MovePlayer();
    }

    private void Jump()
    {
        _direction = Vector3.up * strength;
    }

    private void ApplyGravity()
    {
        _direction.y += gravity * Time.deltaTime;
    }
}
```

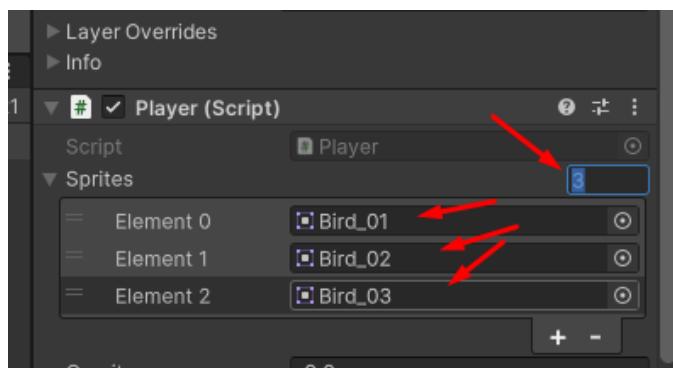
```

private void MovePlayer()
{
    transform.position += _direction * Time.deltaTime;
}

private void AnimateSprite()
{
    spriteIndex++;
    if (spriteIndex >= sprites.Length)
    {
        spriteIndex = 0;
    }
    _spriteRenderer.sprite = sprites[spriteIndex];
}

```

В **Inspector** у **Player** нажимаем +, добавляем 3 спрайта из папки **Sprites**:

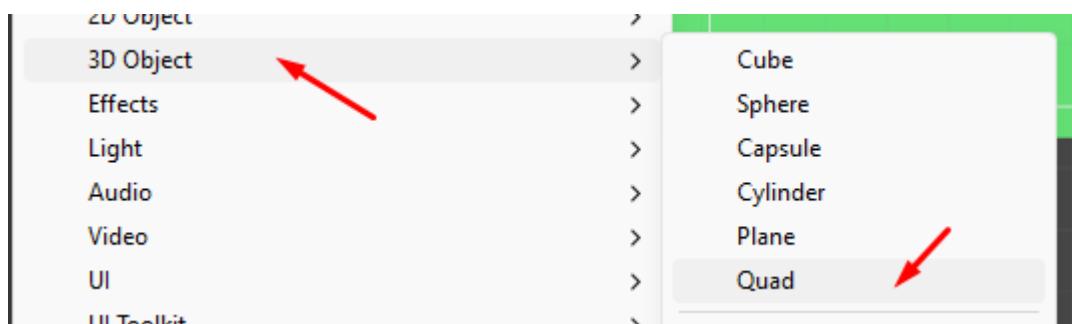


## 8. Создание фона с параллакс-эффектом

**Parallax 2D** — это техника в компьютерной графике, при которой фоновые изображения двигаются медленнее, чем изображения переднего плана, создавая иллюзию глубины в 2D-сцене.

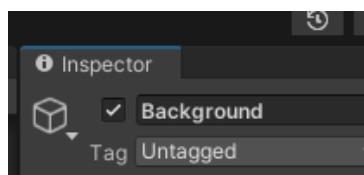
### Хороший пример в 2D.

Для начала создадим новый объект. В **иерархии** щёлкаем **правой кнопкой мыши** - **3D Object – Quad** (так как он удобен для отображения текстур):

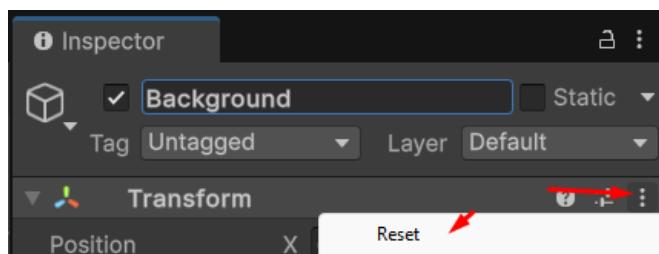


**Quad** — это простой плоский прямоугольник с двумя треугольниками, используемый для отображения текстур. В отличие от других 3D объектов, таких как кубы или сферы, Quad потребляет меньше ресурсов и проще в использовании для отображения фона и других 2D-элементов в 3D пространстве. Это особенно удобно для создания фона с параллакс-эффектом.

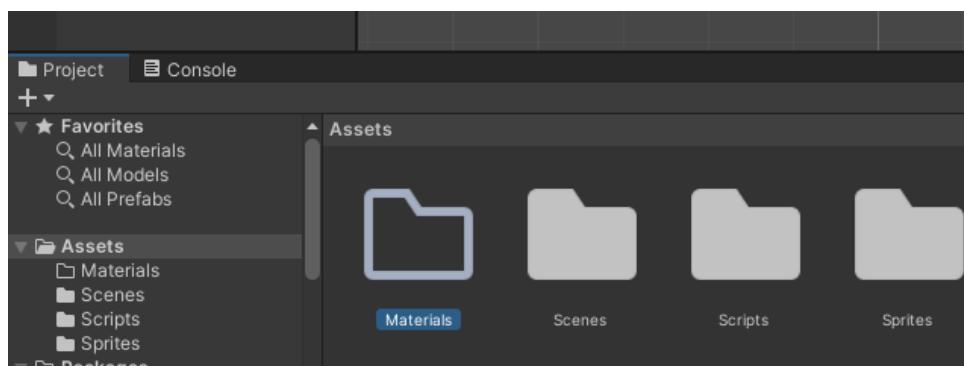
Переименовываем его в **Background**:



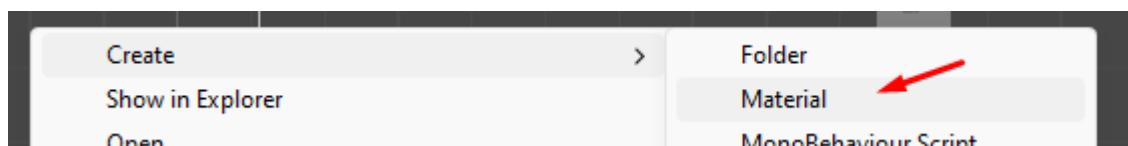
Сбрасываем трансформацию (**Reset Transform**):



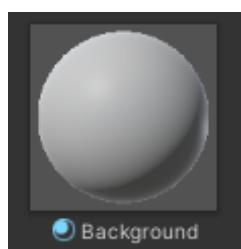
9. Создаём новую папку **Materials**:



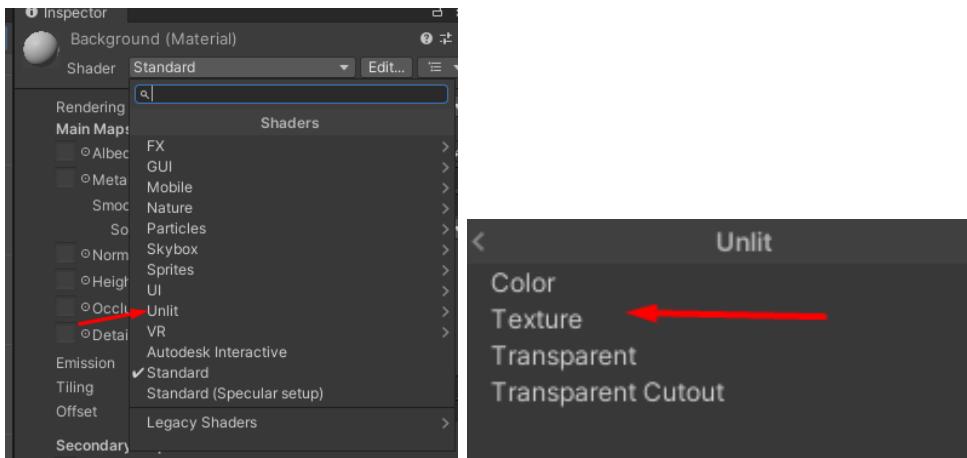
В ней щёлкаем правой кнопкой мыши - **Create – Materials**:



Называем его **Background**:

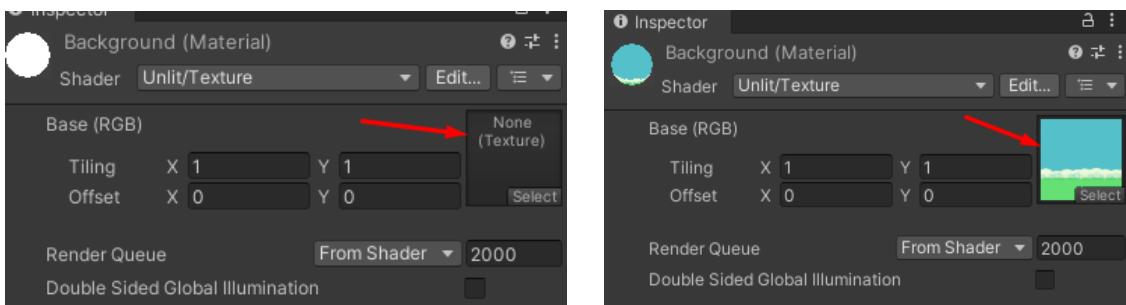


Меняем у него шейдер на **Unlit – Texture** (используется для отображения текстуры без освещения):

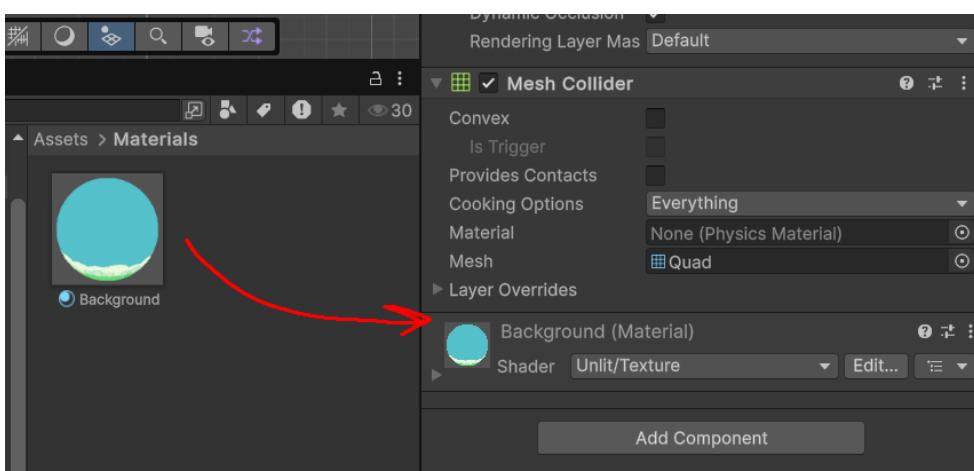


**Unlit – Texture** — это шейдер, который отображает текстуру без всяких дополнительных эффектов. Это просто рендеринг текстуры на объекте. Использование этого шейдера для фона обеспечивает правильное отображение вашего спрайта на фоне, без необходимости учитывать освещение или другие 3D эффекты.

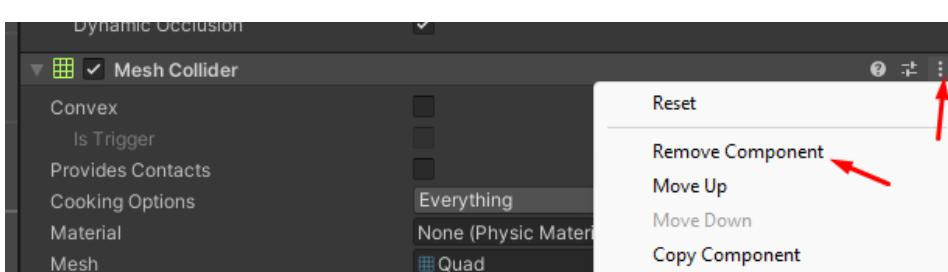
Повесим на текстуру наш фон из спрайтов:



Перетаскиваем на наш **объект Background** созданный материал:

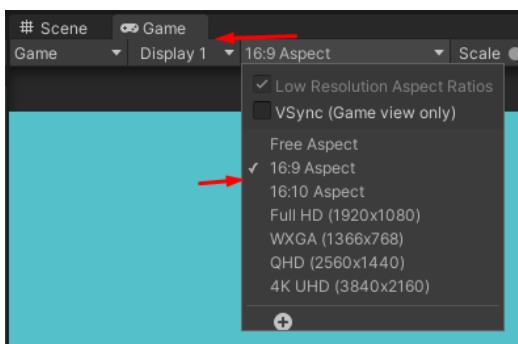


Удаляем **Mesh Collider**, так как он используется для 3D-коллизий, а нам он не нужен:

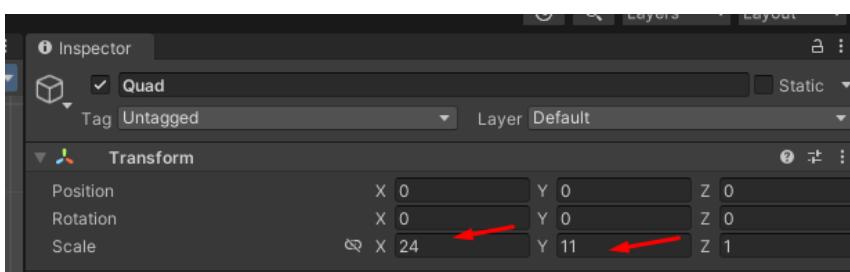


**Mesh Collider** используется для взаимодействия объекта с физической системой Unity. В случае с фоном, который не предназначен для столкновений и не участвует в физике сцены, **Mesh Collider** не нужен. Удаление **Mesh Collider** также помогает оптимизировать производительность.

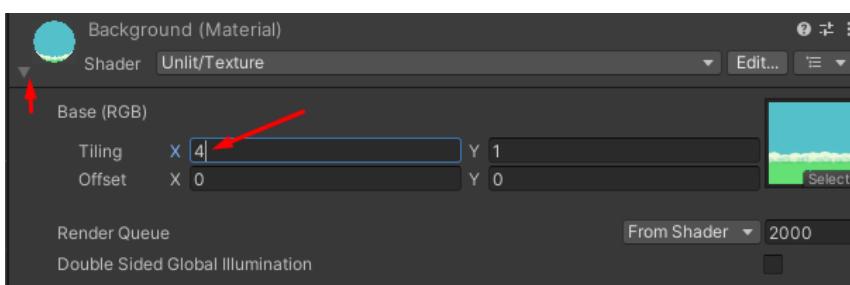
В окне **Game** устанавливаем соотношение сторон **16:9**:



Увеличиваем размер **Background** до 24 по X и 11 по Y:

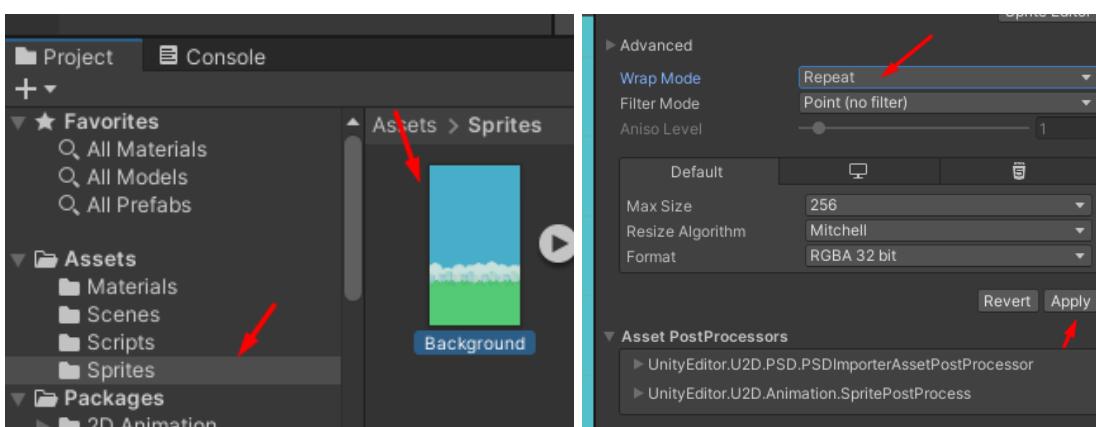


Тут же разворачиваем меню у **Background** и ставим количество **Tiling = 4**:



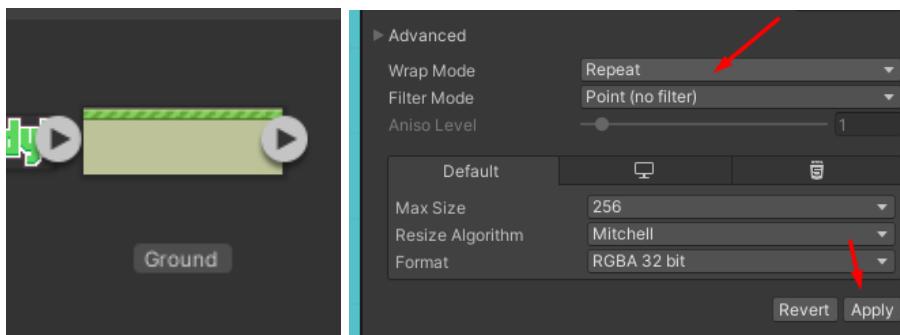
**Tiling** — это параметр, который задает, сколько раз текстура повторяется по X и Y осям на объекте. В вашем случае, **Tiling = 4** означает, что текстура будет повторяться 4 раза по горизонтали, создавая бесшовный фон для нашего параллакса. Это позволяет создать иллюзию непрерывного движения.

В **Inspector** у **Background** меняем **Wrap Mode** на **Repeat** и нажимаем **Apply**:

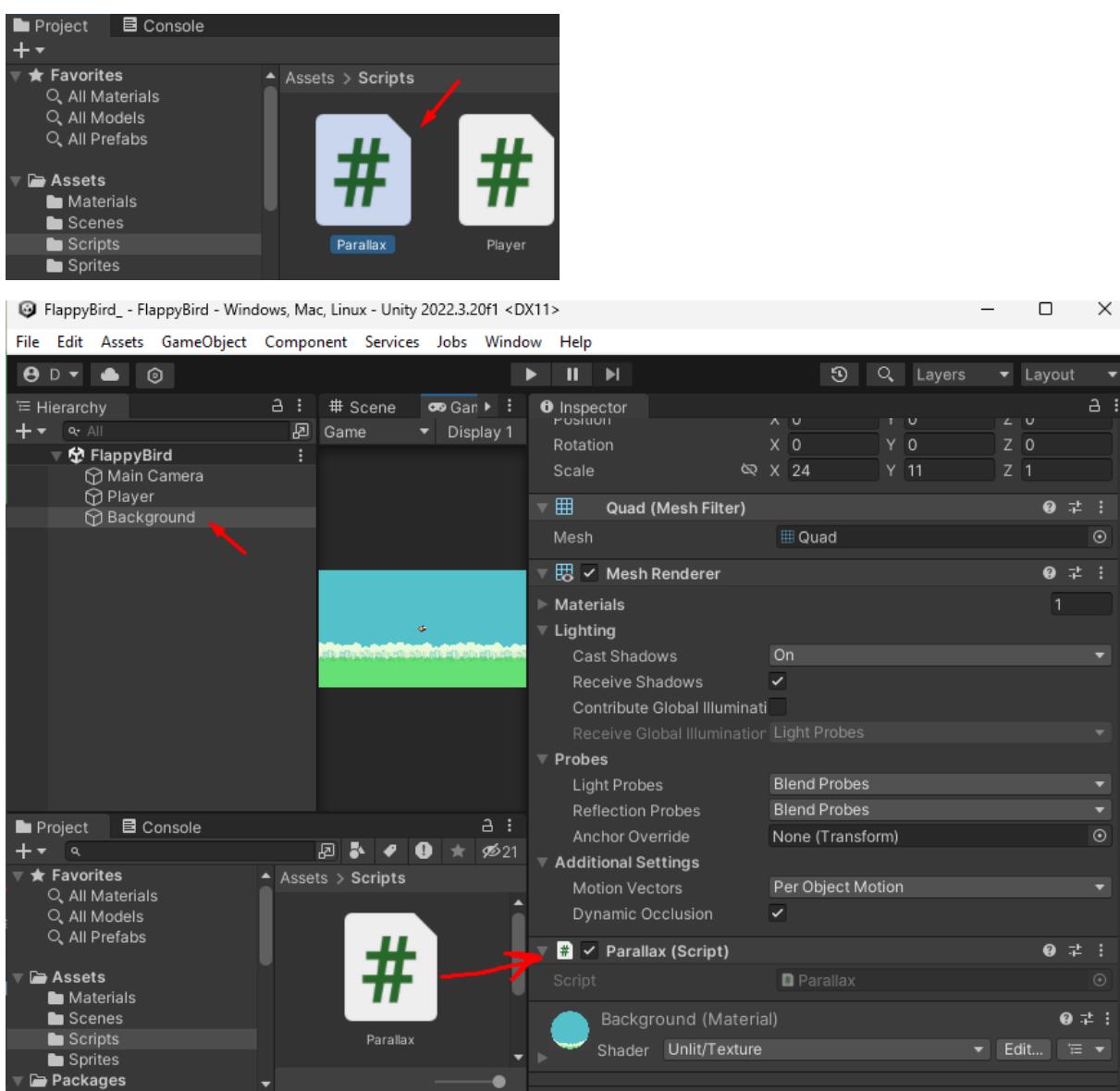


**Wrap Mode** определяет, как текстура будет отображаться, если она выходит за пределы стандартных координат текстуры (0,0) до (1,1). В режиме **Repeat**, текстура будет **повторяться**, если координаты выходят за эти пределы, что важно для создания бесшовного фона.

Повторяем этот шаг для **Ground**:



10. Создаём новый скрипт **Parallax** и добавляем его на **Background**:



Открываем скрипт и пишем код, отвечающий за плавное движение фона:

```
using UnityEngine;
public class Parallax : MonoBehaviour
{
    // Компонент, отвечающий за отрисовку текстуры на объекте
    // (Quad).
    private MeshRenderer meshRenderer;
    // Скорость движения текстуры (можно настроить в инспекторе).
    public float animationSpeed = 1f;

    private void Awake()
    {
        // Получаем ссылку на компонент MeshRenderer, чтобы
        // работать с материалом фона.
        meshRenderer = GetComponent<MeshRenderer>();
    }

    private void Update()
    {
        // Создается новый вектор с смещением по горизонтали и
        // нулевым смещением по вертикали. Этот вектор добавляется к текущему
        // смещению текстуры(mainTextureOffset) материала. Таким образом,
        // текстура будет смещаться горизонтально с заданной скоростью.
        meshRenderer.material.mainTextureOffset += new
        Vector2(animationSpeed * Time.deltaTime, 0f);
    }
}
```

◆ *Переменные и их назначение*

- *meshRenderer* – компонент, отвечающий за отрисовку текстуры на объекте (*Quad*).
- *animationSpeed* – скорость движения текстуры (можно настроить в инспекторе).

◆ *Метод Update()*

- *mainTextureOffset* – это *смещение текстуры* внутри материала.
- Мы *изменяем его по оси X*, двигая текстуру *влево* (+*animationSpeed \* Time.deltaTime*).
- *Time.deltaTime* делает движение *плавным и независимым от FPS*.

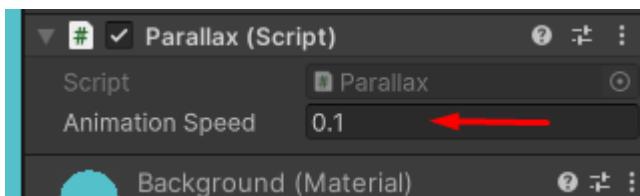
◆ *Как это создаёт эффект параллакса?*

- Мы *не двигаем сам объект, а только его текстуру*.
- Если скорость *animationSpeed* мала, фон движется *медленно* → создаётся *глубина сцены*.
- Можно сделать *разные скорости* для объектов (например, дальний фон медленнее, земля быстрее).

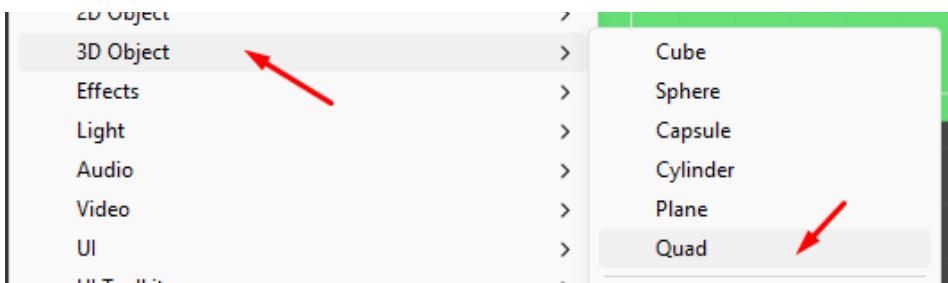
Мы используем *MeshRenderer*, потому что *SpriteRenderer* не поддерживает *mainTextureOffset*.

Это эффективный и оптимизированный способ прокрутки фона без затрат на физику.

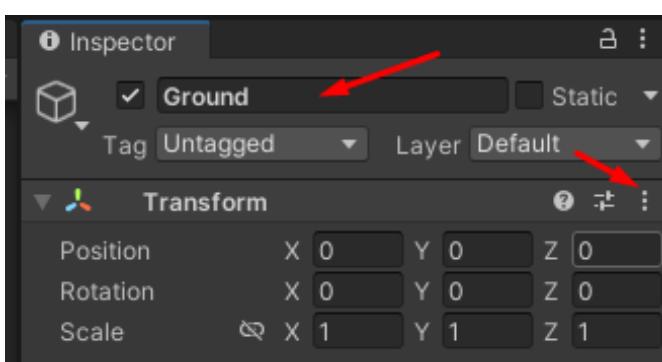
Т.к. это задний фон, то он должен двигаться медленно, поэтому поменяем скорость анимации на **0.1** (но вы можете выставить своё значение):



11. Теперь сделаем тоже самое для земли. Создадим новый объект **3D Object – Quad**:



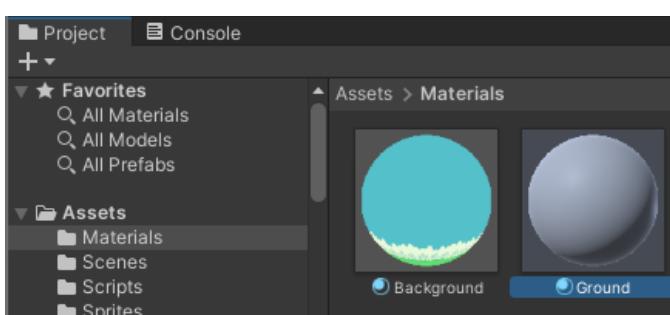
Сбрасываем трансформацию и переименовываем в **Ground**:



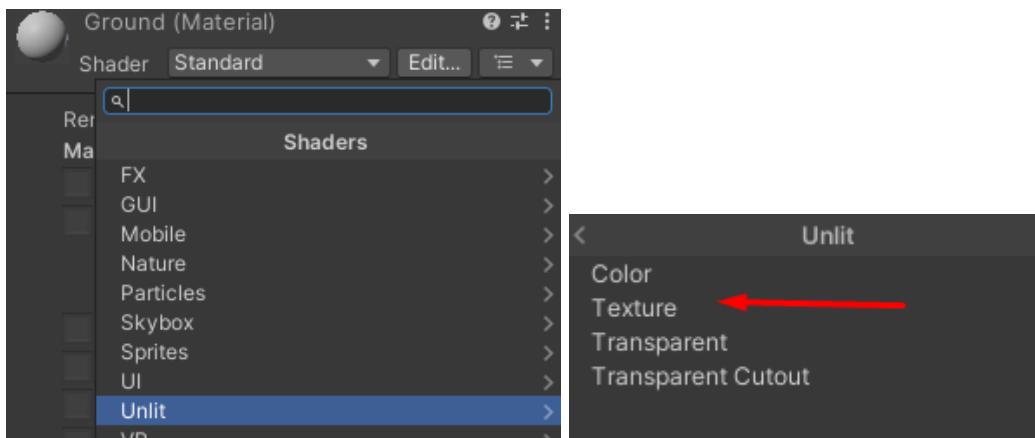
В папке **Materials** создаём новый материал - **Create – Materials**:



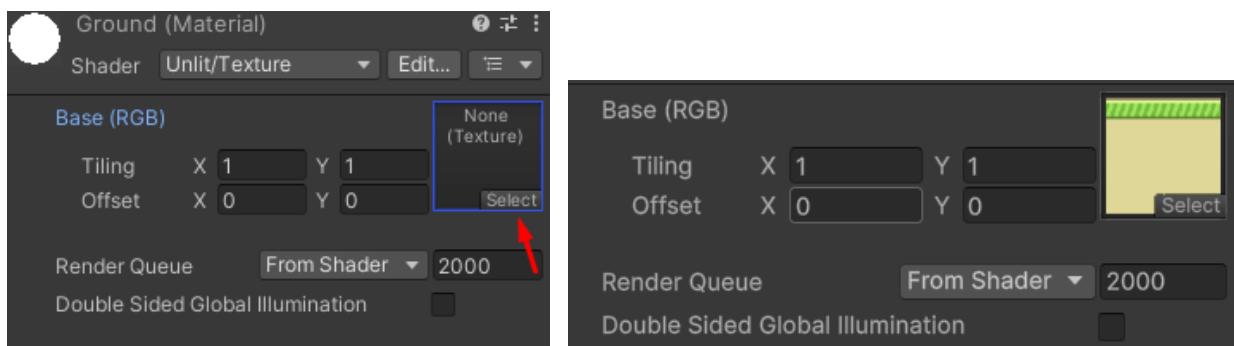
Называем **Ground**:



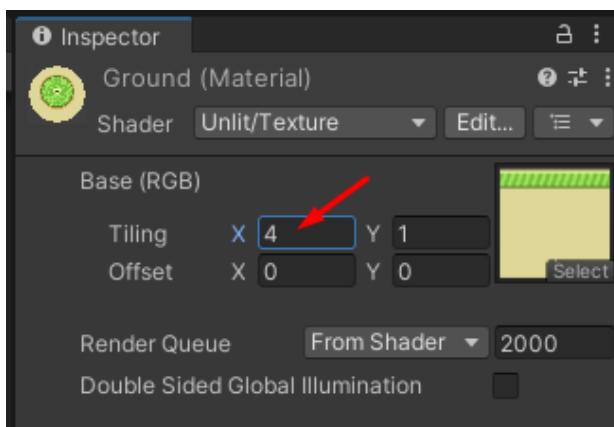
Поменяем у него шейдер **Unlit – Texture**:



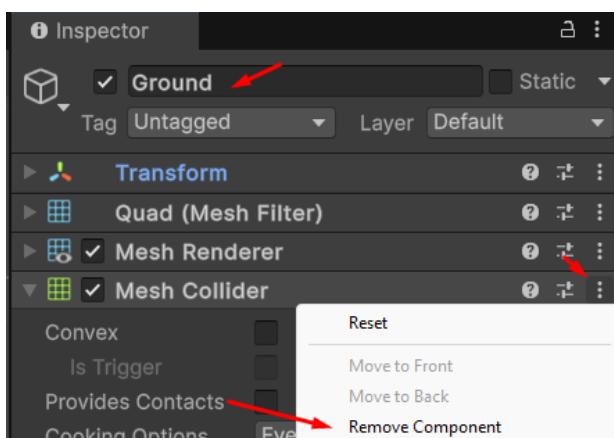
Повесим на текстуру наш **Ground** из спрайтов:



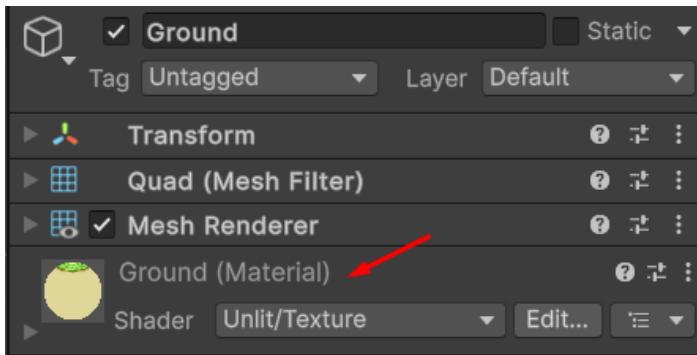
Ставим 4 Tiling:



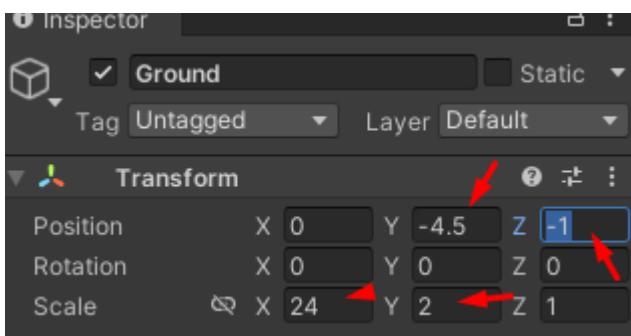
Удаляем у объекта **Ground** компонент **Mesh Collider**, так как он не нужен:



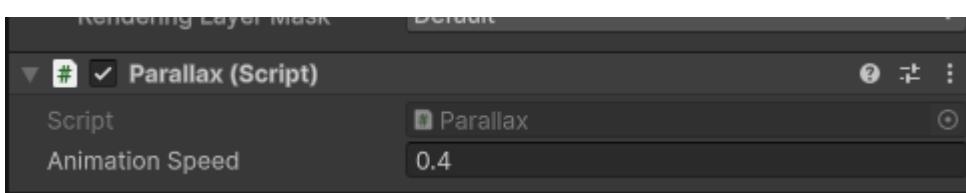
Добавим материал к объекту **Ground**:



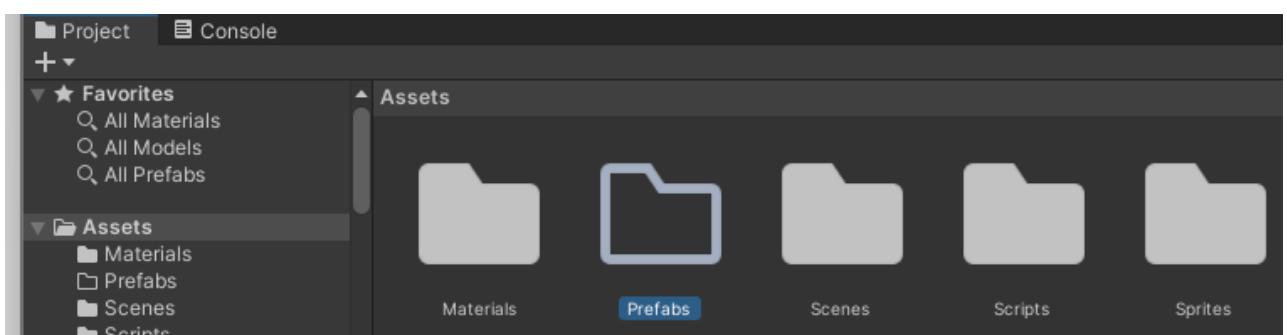
Устанавливаем размеры **24** по X, **2** по Y, позицию **Y = -4.5, Z = -1**:



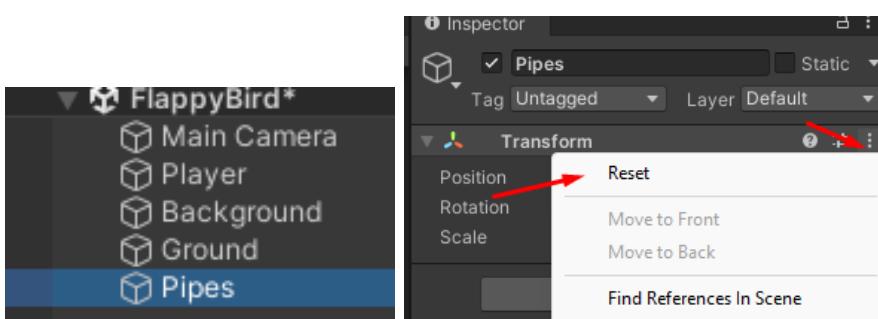
Добавляем на **Ground** скрипт **Parallax** и устанавливаем скорость (например, **0.4**):



12. Перейдём к созданию труб. Создаём папку **Prefabs**:



Создаём новый объект (**Create Empty**), называем его **Pipes** и сбрасываем трансформацию:



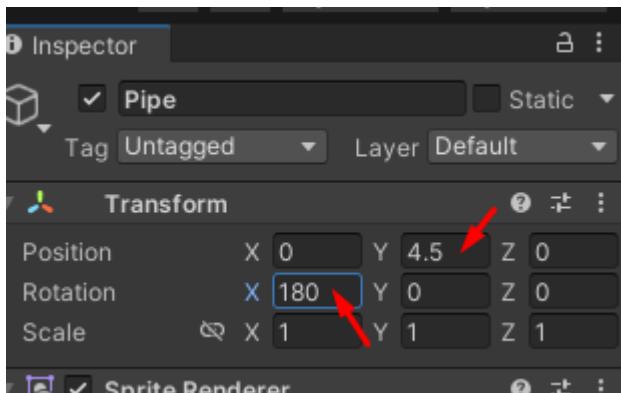
Переносим в него из папки **Sprites** нашу трубу **Pipe**:



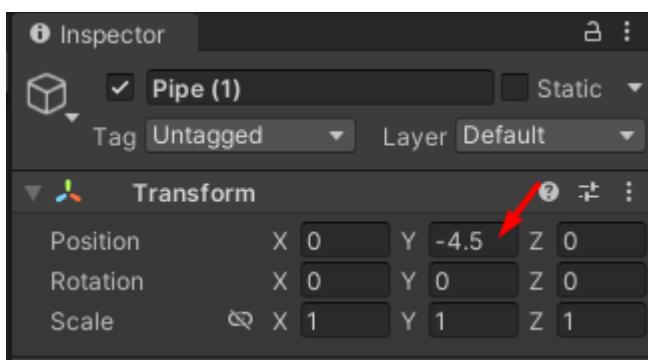
Создадим дубликат нашей трубы (**Ctrl+D**):



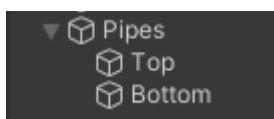
Первую трубу развернём на **180 по x**, и поднимем на **4.5 по y**:



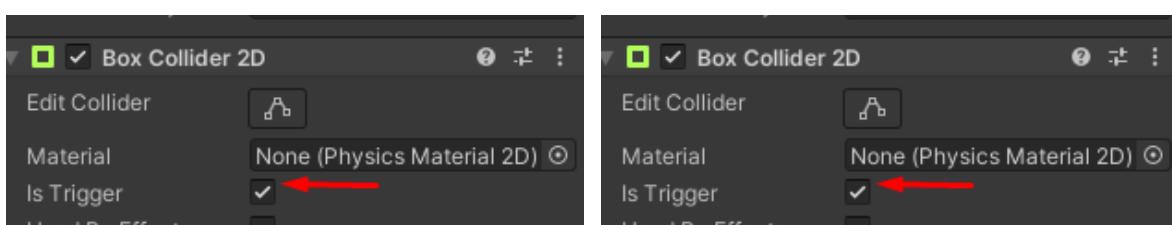
Вторую трубу опустим на **-4.5 по y**:



Поменяем названия наших труб на **Top** и **Bottom**:

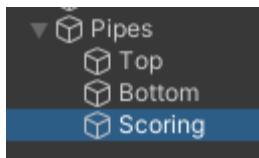


Добавляем **Box Collider 2D** для двух труб и ставим галочку **Is Trigger**:

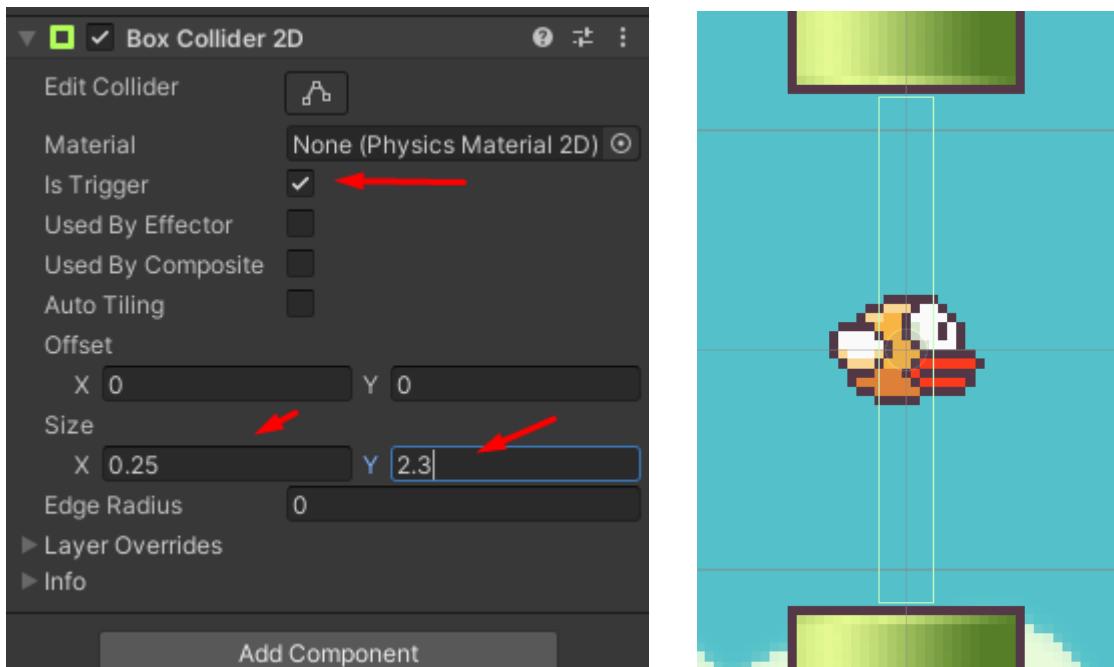


13. Создадим между трубами объект подсчёта очков.

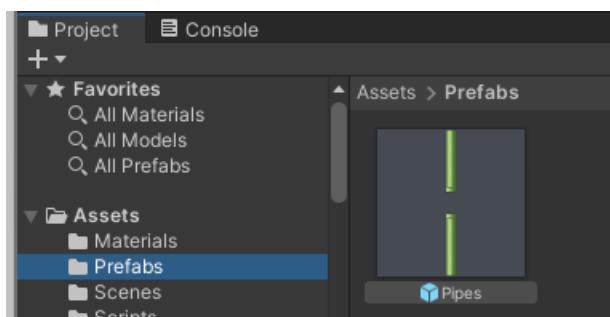
Внутри **Pipes** создаём новый пустой объект и называем его **Scoring**:



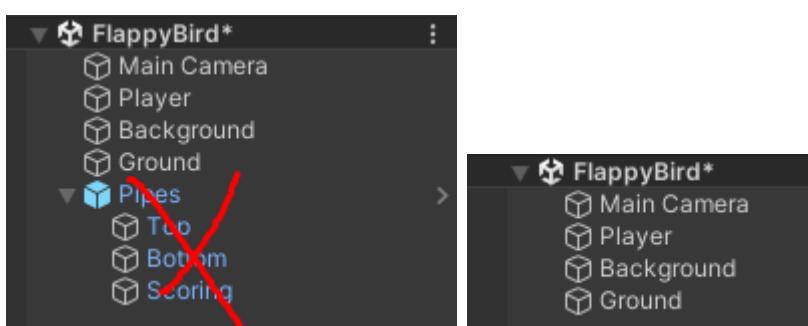
Добавляем ему **Box Collider 2D**, ставим галочку **Is Trigger**, и меняем размер по **x** = 0.25, по **y** = 2.3:



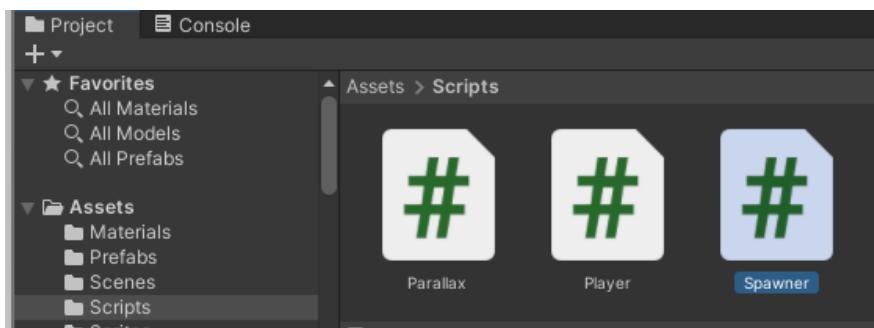
Перетаскиваем наш объект в папку **Prefabs**:



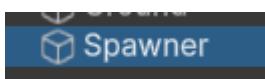
Удалим объект из нашей иерархии:



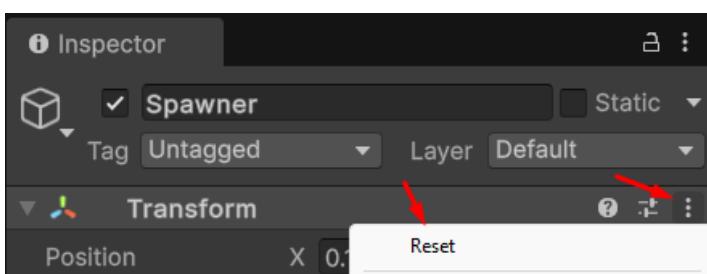
14. В папке **Scripts** создадим скрипт для появления наших труб **Spawner**:



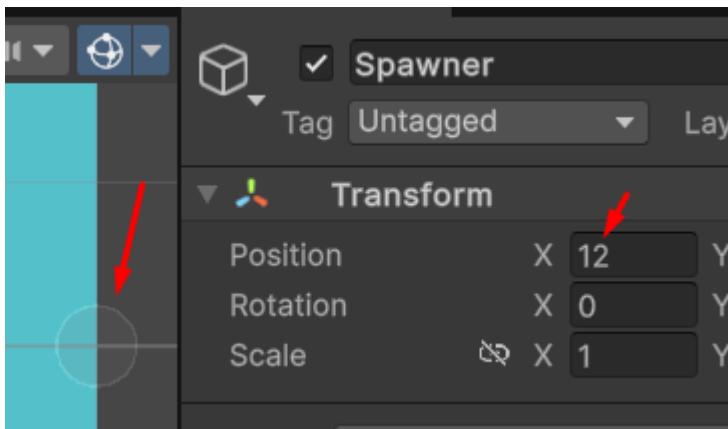
Создадим новый объект и назовём его **Spawner**:



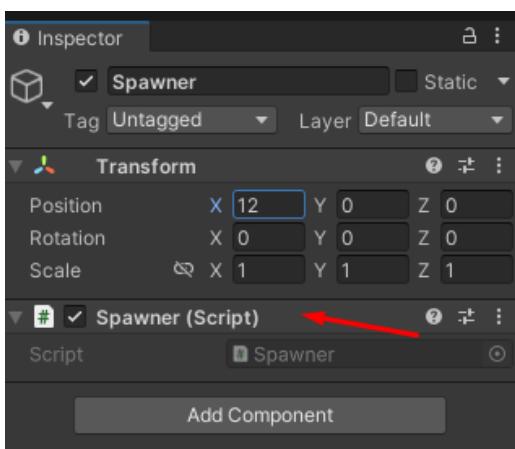
Сбросим для него трансформацию:



Позицию по **X** поменяем на **12**, чтобы он был за пределами игрового поля:



Добавим на него наш скрипт **Spawner**:



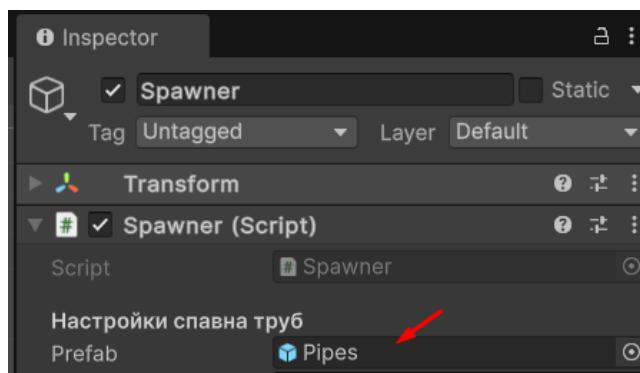
Пропишем сам скрипт Spawner:

```
using UnityEngine;

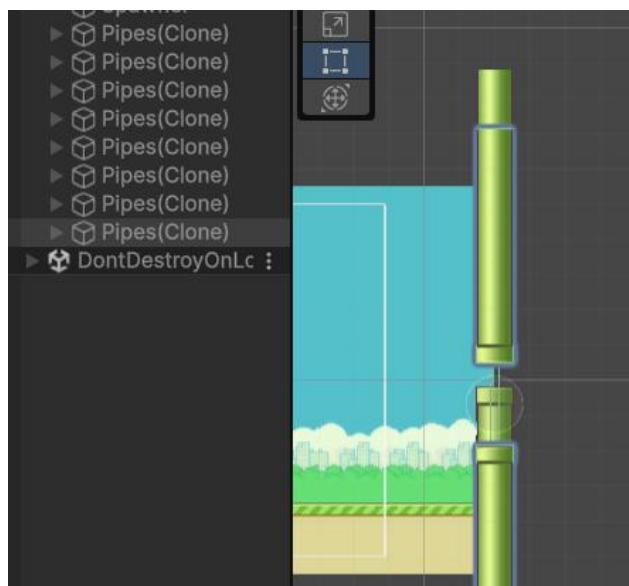
public class Spawner : MonoBehaviour
{
    [Header("Настройки спавна трубы")]
    // Префаб трубы
    [SerializeField] private GameObject prefab;
    // Частота появления труб
    [SerializeField] private float spawnRate = 1f;
    // Минимальная высота появления трубы
    [SerializeField] private float minHeight = -1f;
    // Максимальная высота появления трубы
    [SerializeField] private float maxHeight = 1f;

    private void OnEnable() // Этот метод вызывается, когда
    // объект становится активным
    {
        // Запускаем спавн с заданной частотой
        InvokeRepeating(nameof(Spawn), spawnRate,
        spawnRate);
    }
    private void OnDisable() // Этот метод вызывается,
    // когда объект становится неактивным
    {
        // Останавливаем спавн при отключении объекта
        CancelInvoke(nameof(Spawn));
    }
    private void Spawn()
    {
        Vector3 spawnPosition = transform.position +
        Vector3.up * Random.Range(minHeight, maxHeight);
        Instantiate(prefab, spawnPosition,
        Quaternion.identity); // Создаём трубу в случайной позиции
    }
}
```

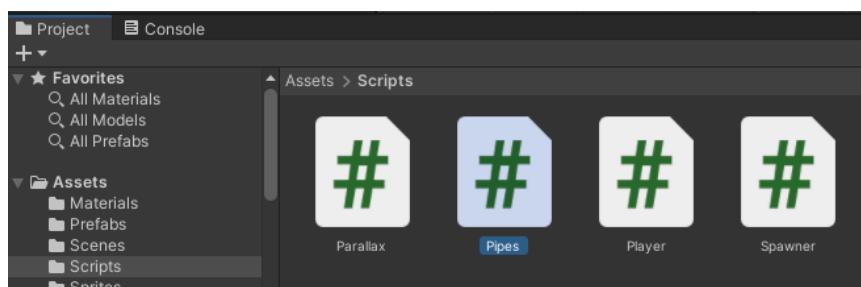
Перенесём наш префаб **Pipes** на объект **Spawner**:



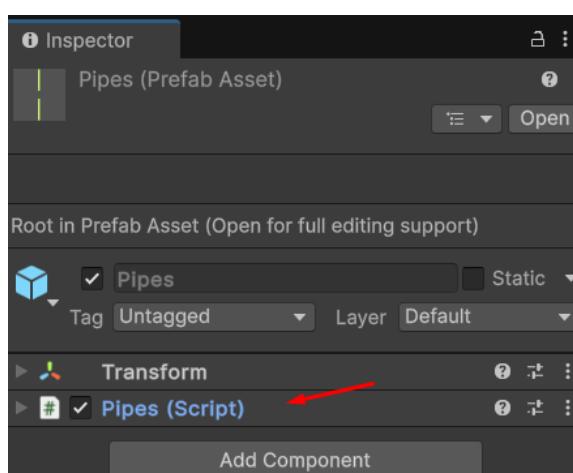
Запустив сцену, мы увидим, что объекты создаются:



15. Теперь нужно прописать движение труб. Создаём новый скрипт **Pipes**:



Добавляем его в наш префаб **Pipes**:



Открываем и пишем скрипт Pipes:

```
using UnityEngine;

public class Pipes : MonoBehaviour
{
    [Header("Настройки движения")]
    [SerializeField] private float speed = 5f; // Скорость
движения труб влево
    private float leftEdge; // Граница, за которую трубы
удаляются

    private void Start()
    {
        leftEdge =
Camera.main.ScreenToWorldPoint(Vector3.zero).x - 1f;
    }

    private void Update()
    {
        MovePipes();
        CheckDestroyCondition();
    }

    // Двигает трубы влево
    private void MovePipes()
    {
        transform.position += Vector3.left * speed *
Time.deltaTime;
    }

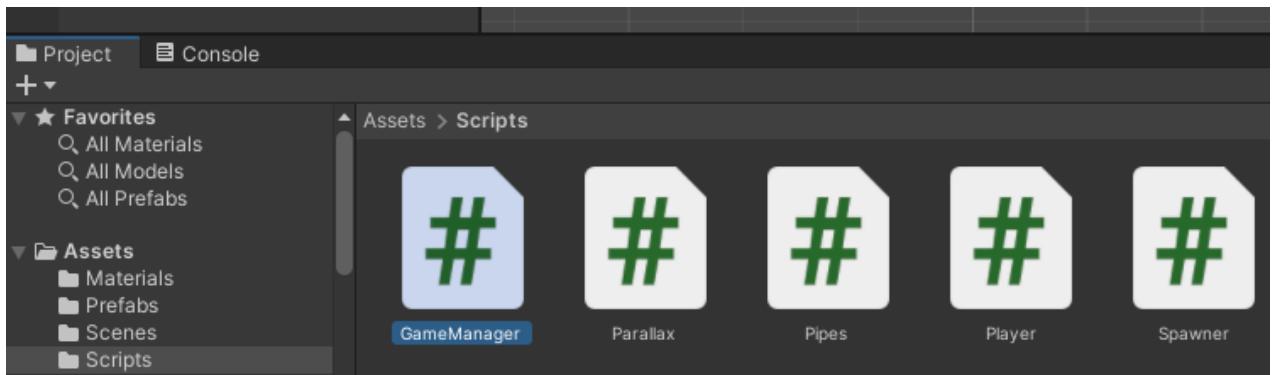
    // Уничтожает трубы, если они вышли за пределы экрана
    private void CheckDestroyCondition()
    {
        if (transform.position.x < leftEdge)
        {
            Destroy(gameObject);
        }
    }
}
```

- ◆ **Метод Start**
- Мы вычисляем **крайний левый предел экрана** (*leftEdge*).
- *Camera.main.ScreenToWorldPoint(Vector3.zero).x* → переводит координату **левой границы экрана** в мировые координаты.
- *-1f* → добавляем **запас в -1 единицу**, чтобы трубы удалялись **чуть раньше**, избегая задержек.

📌 **Зачем это нужно?**

*Когда труба уходит за экран, она становится невидимой.* Чтобы не нагружать игру, мы её **удаляем**.

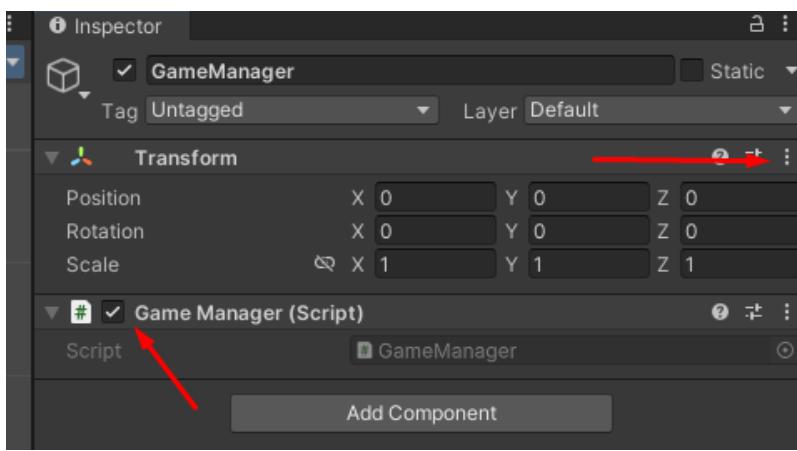
## 16. Создадим скрипт для общего управления нашей игрой **GameManager**:



Также создадим новый объект **GameManager**:

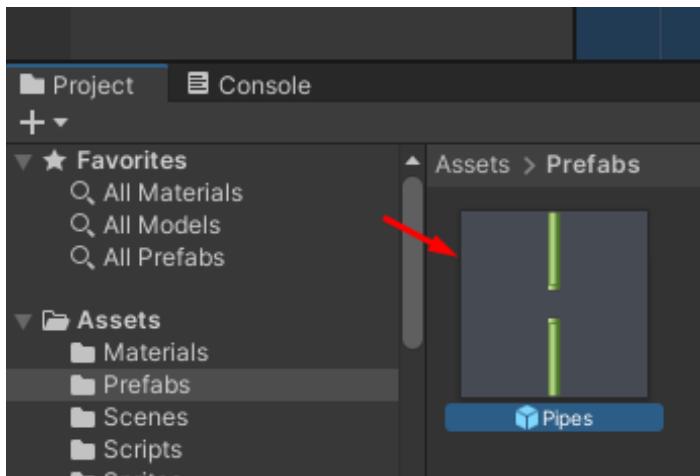


Сбросим для него трансформацию и добавим в него скрипт **GameManager**:

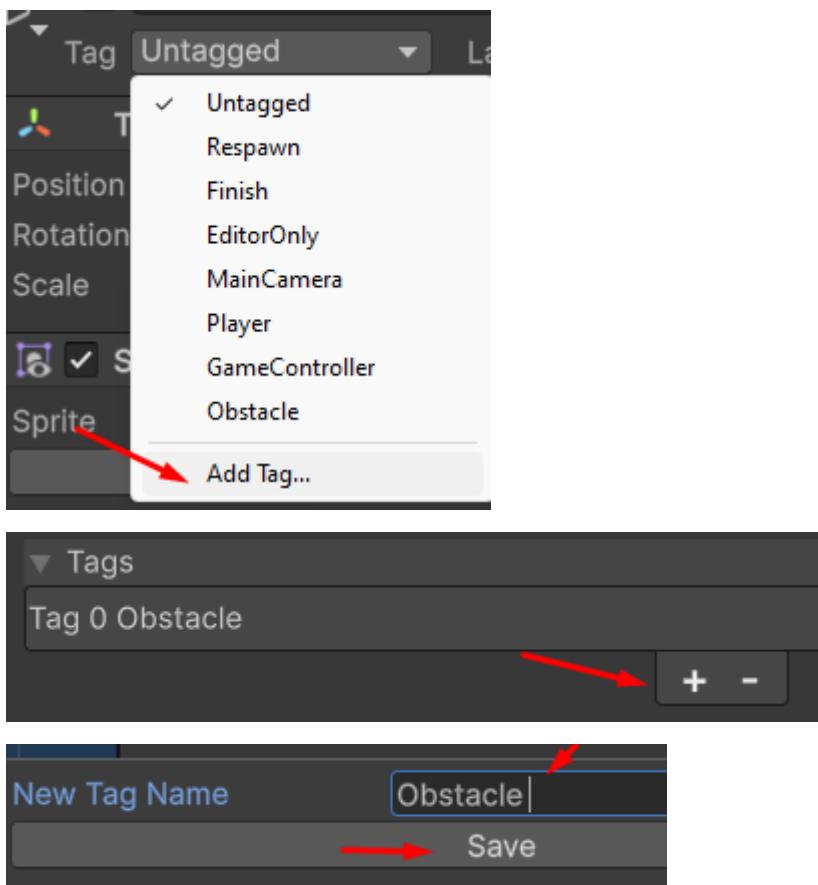


17. Нам нужна знать когда **птичка** прошла через трубу и когда ударяется в неё или землю, чтобы завершить игру.

Для этого переходим в наш префаб **Pipes**:



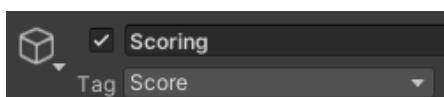
И добавим теги, **Obstacle** для **Top** и **Bottom** труб:



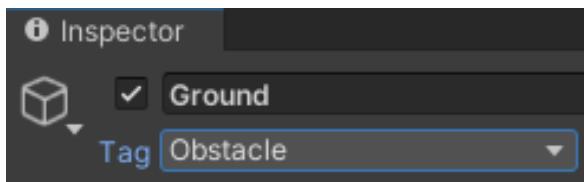
**Итог:**



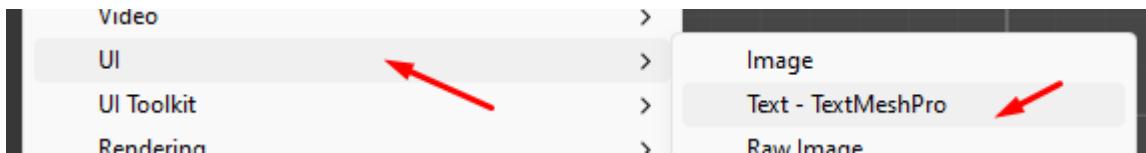
И тег **Score** для **Scoring**:



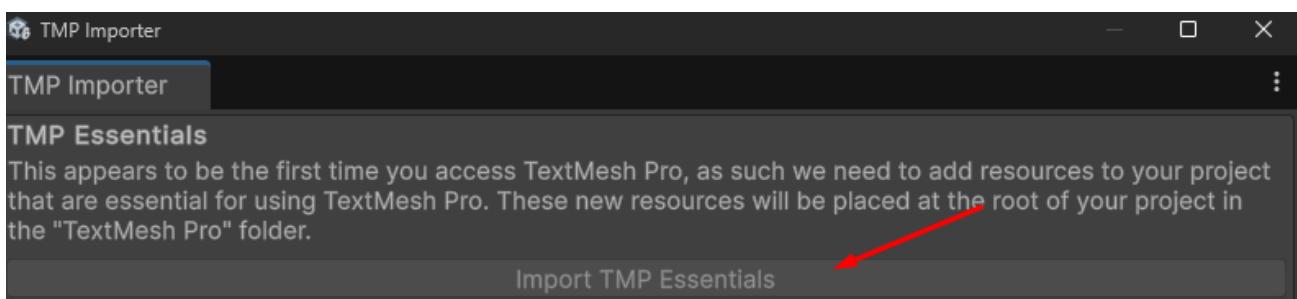
Также добавим тег **Obstacle** для **Ground**:



18. Добавим UI и начнём с отображения очков. Жмём правой кнопкой мыши, выбираем **UI – TextMeshPro**:

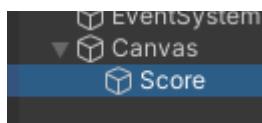


Соглашаемся на импорт:

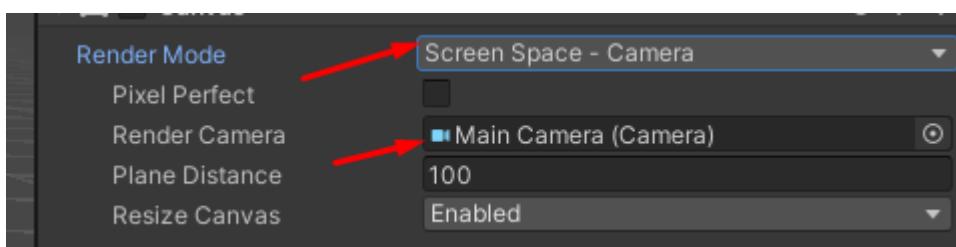


После можем закрыть это окно.

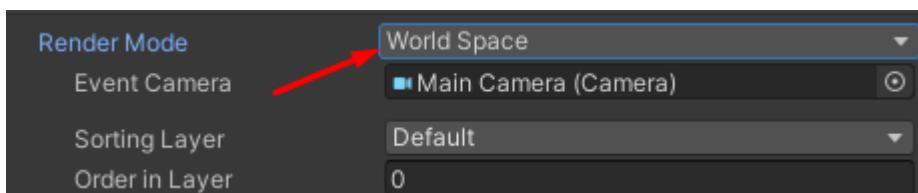
Называем текст – **Score**:



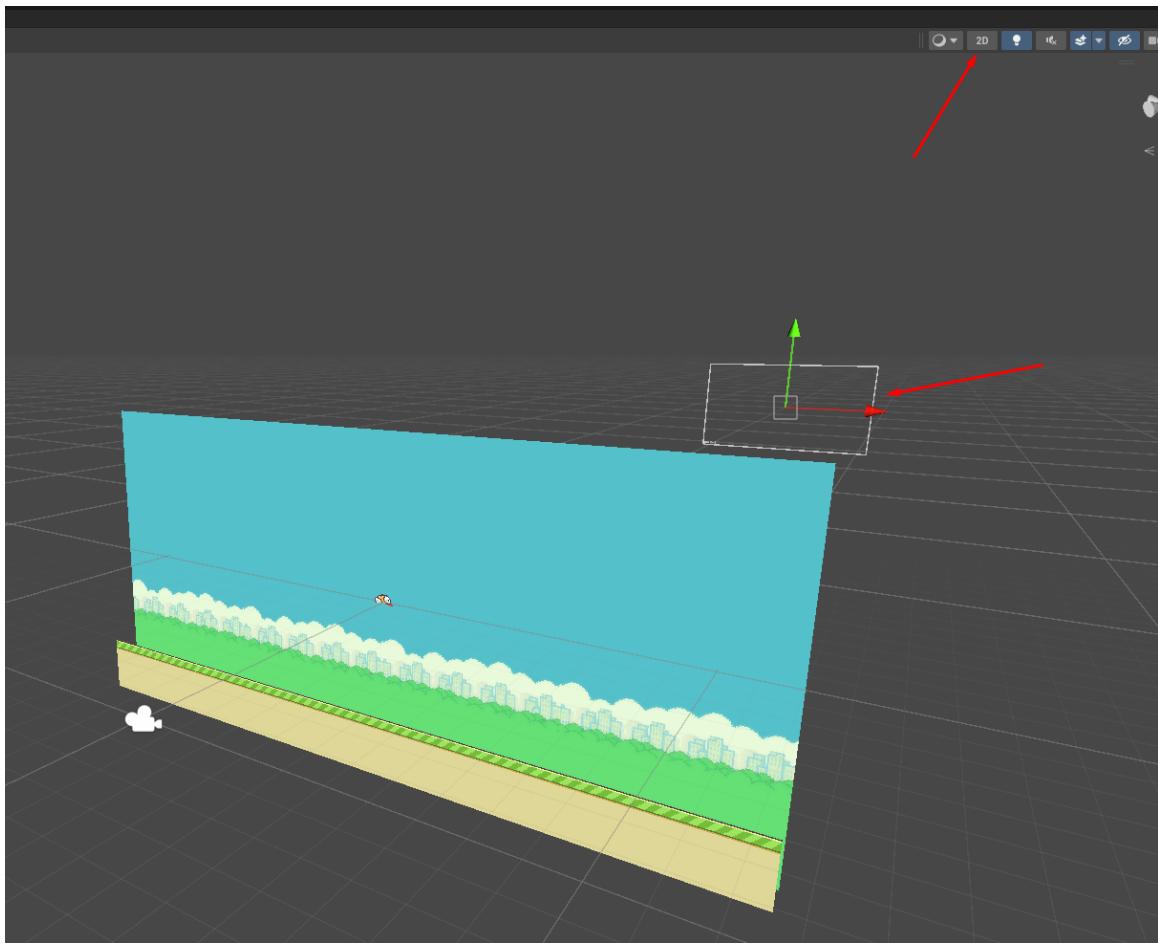
Выбираем **Canvas**. В Render Mode выбираем режим **Screen Space** и ставим нашу камеру:



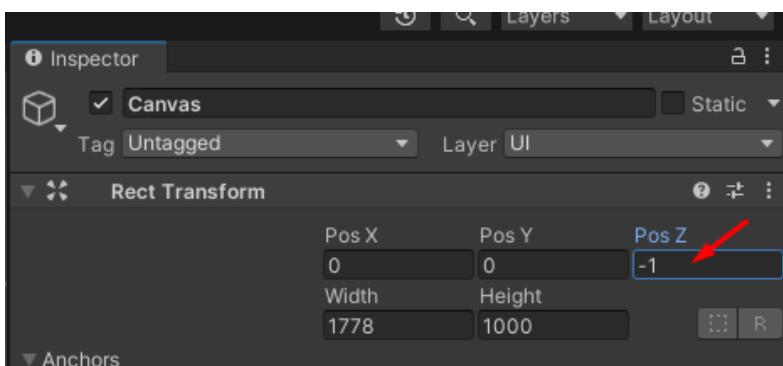
Затем меняем режим на **World Space**:



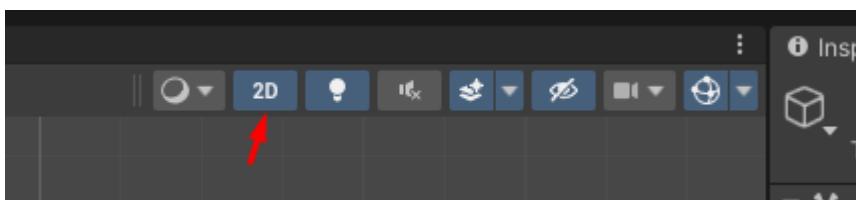
И текст у нас по-прежнему не отобразился. На самом деле он есть, но спрятан за областью наших других слоёв, чтобы в этом убедиться, уберите галочку с 2D и повращайте камерой на сцене:



Чтобы его поставить на передний слой, для этого мы должны изменить ось Z, меняем у нашего **Canvas** ось Z на -1:



Не забудьте переключиться снова в **2D** режим:



19. Давайте загрузим и установим пиксельный шрифт с интернета.

Для примера перейдём на сайт:

<https://fonts-online.ru/categories/pixel-fonts>

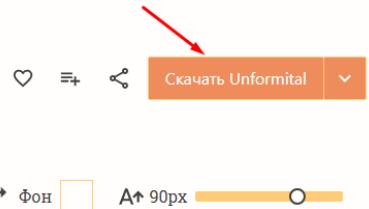
Я выберу шрифт **Unformital**, вы можете использовать любой другой:

# 0123456789

Нажимаем скачать:

Шрифты Онлайн > Шрифты > Unformital

Unformital

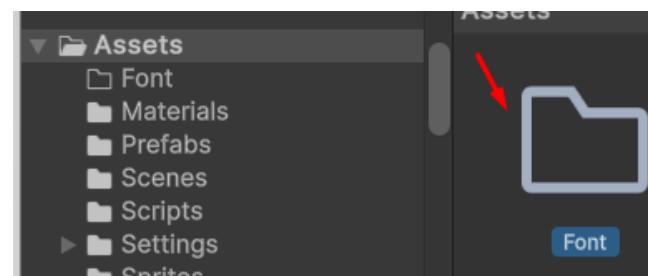


Шрифт Unformital. Гарнитура содержит 6 файлов и поддерживает 12 языков.

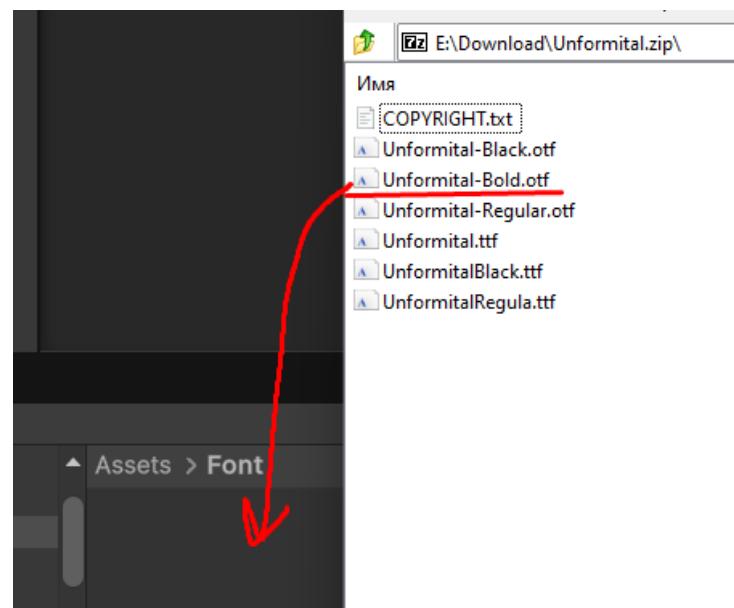


Unformital Regular 400

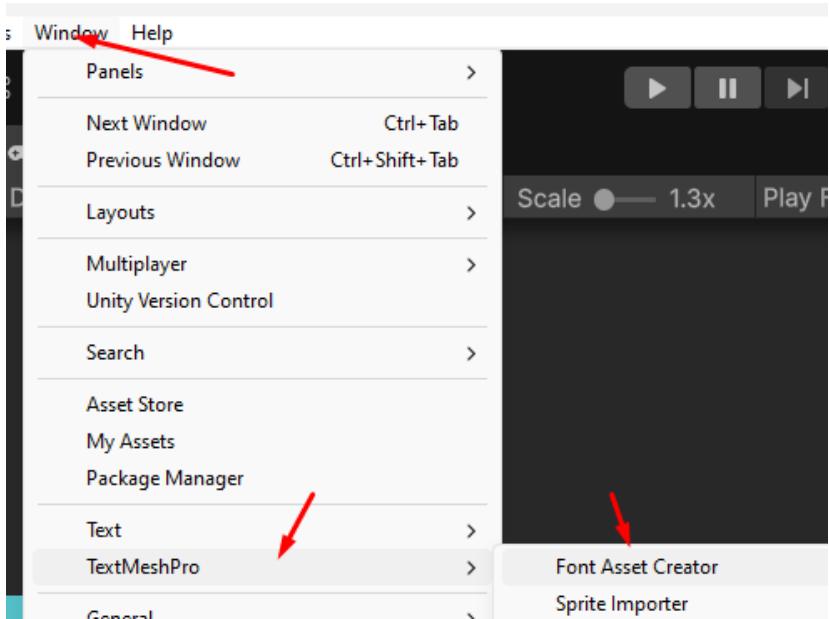
Затем создадим папку Font:



И перенесём в неё шрифт из архива (в качестве примера я возьму один – **Bold**):

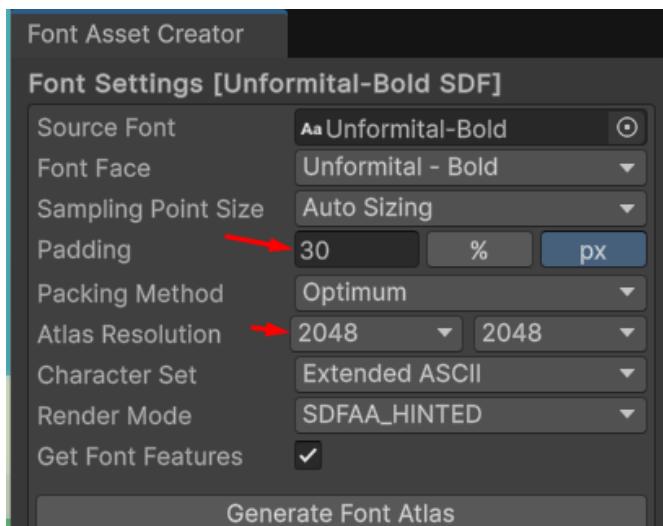


Затем нажимаем на вкладку Window – TextMeshPro – Font Assets Creator:

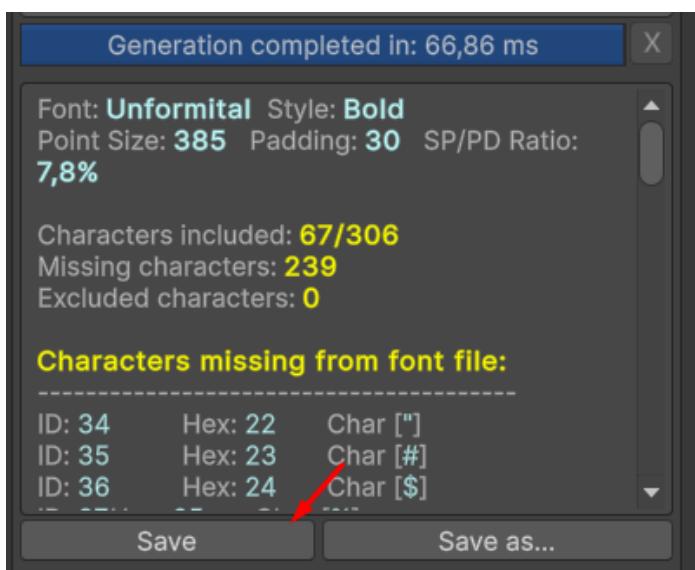


Дальше поставьте **Padding** на **30**, **Atlas Resolution** на **2048x2048**, и нажимаем

**Generate Font Atlas:**

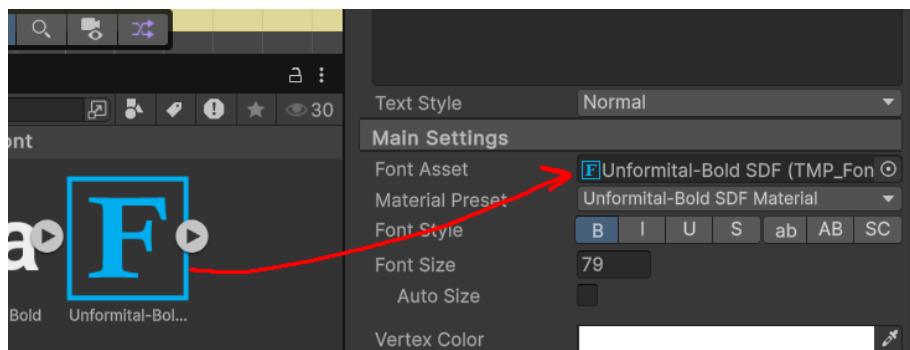


После сохраним сгенерированный шрифт в папку **Font**:



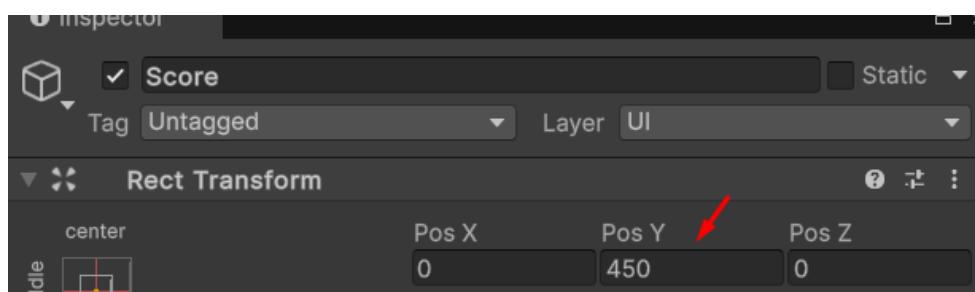
После в этом же окне нажимаем **Save** и сохраним шрифт в нашу папку **Font**:

После наш **ассет шрифта** мы можем назначить в **инспекторе** на нужный нам текст:

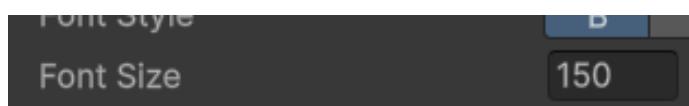


Оформление можете использовать как у меня, или придумать своё:

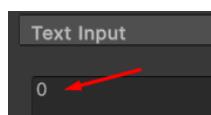
- позиции по **x = 0, y = 350**:



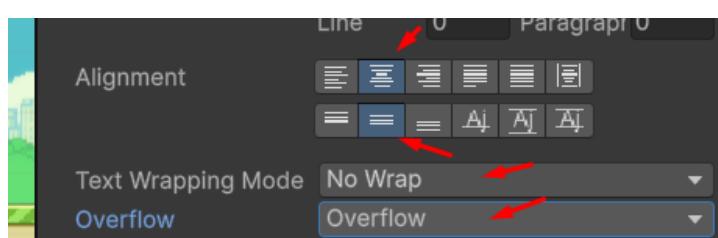
Чем крупнее шрифт, тем лучше выглядит она на экране (также это всё зависит от настроек при генерации шрифта), я установил размер на **150**:



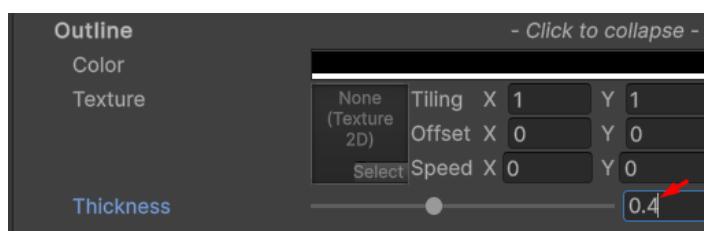
- в тексте пишем **0**:



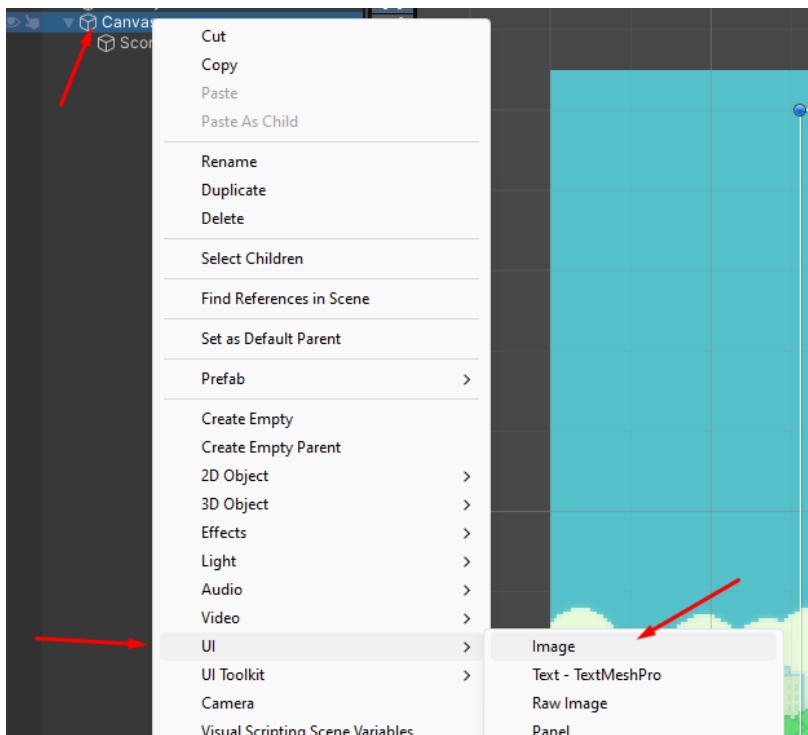
- выравнивание по центру, **Overflow** стоит в **Overflow**, **y Wrapping** в положении **No Wrap** (в старых версиях Unity – **Disabled**):



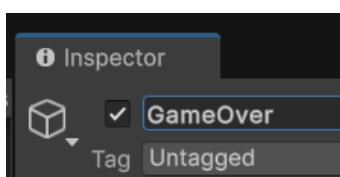
- добавляем небольшую обводку в **Outline – 0.4**:



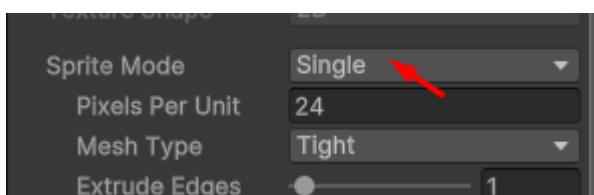
## 20. Добавим спрайт в Canvas - Image:



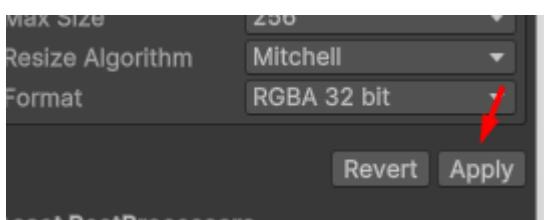
Переименовываем в **GameOver**:



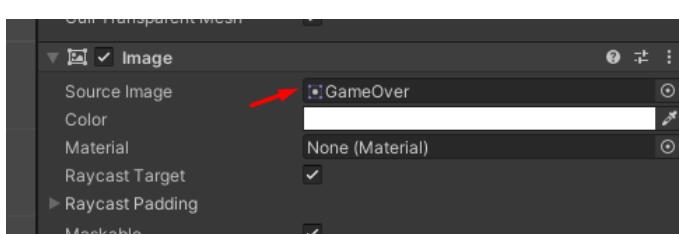
Выберем в папке **Sprites** надпись **GameOver** и поменяем **Sprite Mode** на **Single**:



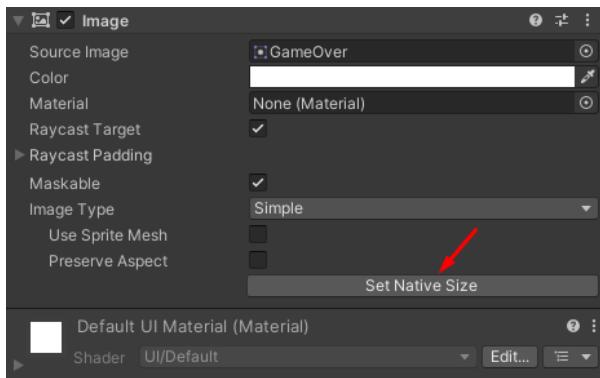
И не забудем нажать **Apply**:



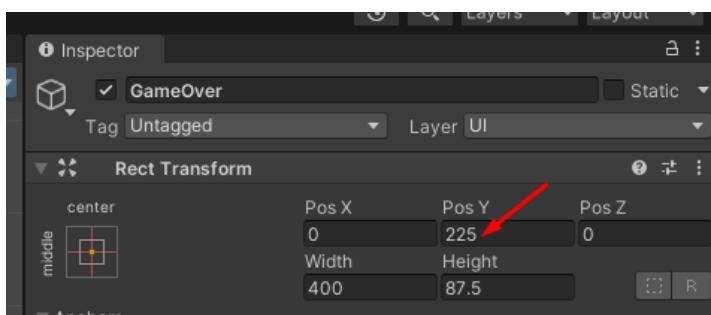
Переносим спрайт **GameOver** в **Source Image**:



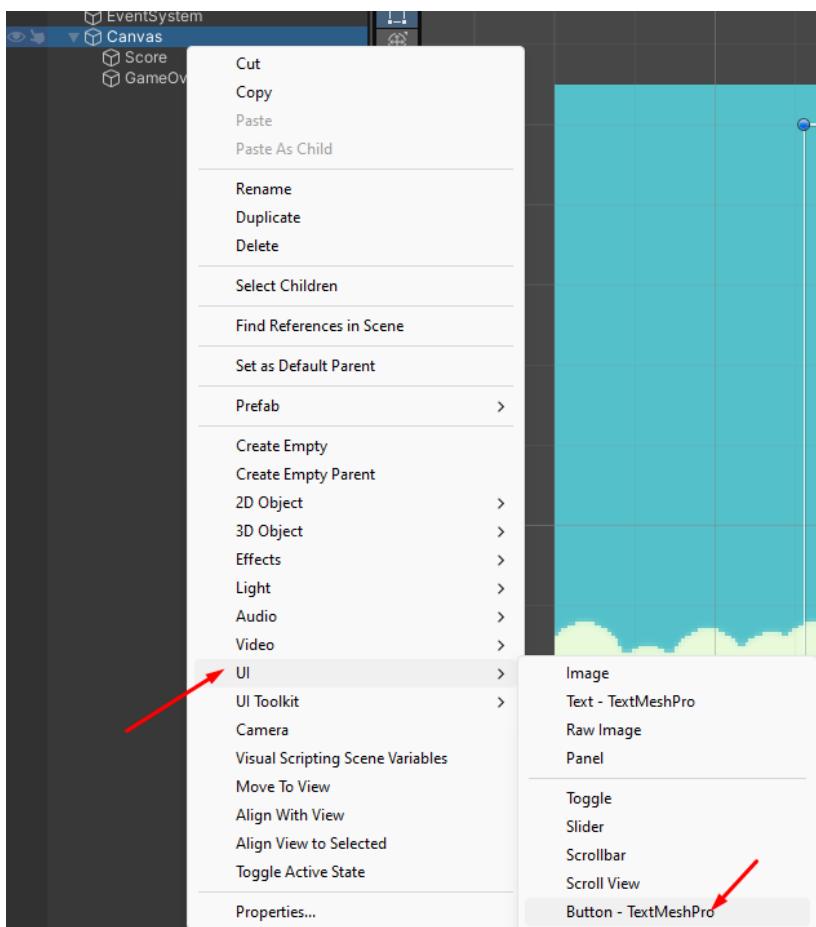
Нажимаем **Set Native Size**, чтобы растянуть наше изображение:



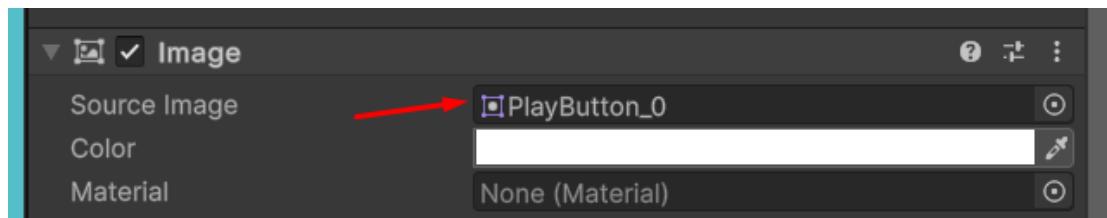
Положение поставим повыше на 225 по y:



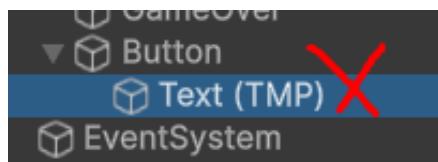
21. Добавим кнопку начала игры. Создаём в Canvas – Button TextMeshPro:



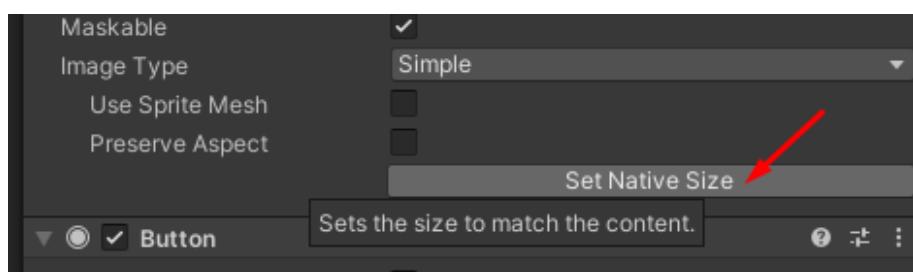
Добавляем на неё спрайт кнопки из папки **Sprites**:



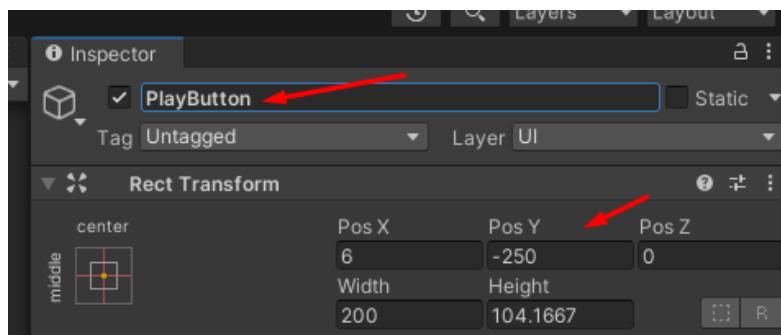
Нам не нужен текст, поэтому мы можем раскрыть нашу кнопку и удалить его:



Нажимаем **Set Native Size**, чтобы растянуть наше изображение:



Называем её **PlayButton** и опускаем немножко пониже (-250 по y):



22. Теперь отредактируем скрипт **GameManager**.

Его суть будет заключаться в следующем:

### A. При запуске игры

- Устанавливаем FPS.
  - Останавливаем игру и ждём нажатия кнопки.
- ### B. Когда игрок нажимает "Play"
- Очищаем старые данные.
  - Запускаем игру.
  - Удаляем старые трубы.
- ### C. В процессе игры
- Увеличиваем счёт при прохождении препятствий.
- ### D. Когда игрок проигрывает GameOver()

➤ Отображаем экран "Game Over".

➤ Останавливаем игру.

1) В начале подключим пространство имён **TMPro** для работы с текстом:

```
using TMPro;
using UnityEngine;
```

2) Затем мы создаём пространства имён (**переменные**), которые понадобятся для управления игрой:

```
[Header("Игровые объекты")]
[SerializeField] private Player player; // Ссылка на игрока
чтобы включать/выключать его управление
[SerializeField] private TextMeshProUGUI scoreText; // UI-
элемент для отображения текущего счёта
[SerializeField] private GameObject playButton; // Кнопка
начала игры
[SerializeField] private GameObject gameOver; // Экран "Game
Over"

[Header("Игровые настройки")]
[SerializeField] private int targetFrameRate = 60; //
Ограничение частоты кадров
private int score; // переменная для хранения текущего счёта
игрока
```

3) Далее пропишем метод **Awake**:

```
private void Awake()
{
    Application.targetFrameRate = targetFrameRate; //
Устанавливаем желаемую частоту кадров
    Pause(); // Останавливаем игру при запуске
}
```

Когда сцена загружается, вызывается **Awake()**, где:

Устанавливается ограничение FPS:

**Application.targetFrameRate = targetFrameRate;**

*это свойство в Unity, которое позволяет задать целевую частоту кадров (FPS) для приложения. Когда вы устанавливаете значение **Application.targetFrameRate**, Unity пытается поддерживать указанную частоту кадров, насколько это возможно, в зависимости от возможностей устройства и сложности сцены.*

Игра ставится на паузу

**■** Это нужно, чтобы игра **не начиналась сразу**, а ждала нажатия кнопки "**Play**".

4) Затем напишем метод **Pause()**:

```
private void Pause()
{
    Time.timeScale = 0f; // Останавливаем игровое время
    player.enabled = false; // Отключаем управление игроком
}
```

Когда игра на паузе:

- Время останавливается (**Time.timeScale**= 0f).

*Метод **Time.timeScale** в Unity управляет масштабом времени в игре. Это свойство влияет на то, как быстро или медленно протекает игровое время. Установивая значение **Time.timeScale**, вы можете ускорять, замедлять или полностью останавливать время в игре.*

*Что делает **Time.timeScale**:*

- **Time.timeScale = 1f;** — время течет с нормальной скоростью (реальное время).
- **Time.timeScale = 0.5f;** — время замедляется в два раза.
- **Time.timeScale = 0f;** — время полностью останавливается (игра "замораживается").
- **Time.timeScale = 2f;** — время ускоряется в два раза.

- Отключается управление игроком.

6) Метод **ResetGame()** (сброс параметров перед началом игры):

```
private void ResetGame()
{
    score = 0; // Обнуляем счёт
    scoreText.text = score.ToString(); // Обновляем UI
    playButton.SetActive(false); // Скрываем кнопку "Play"
    gameOver.SetActive(false); // Скрываем экран "Game
Over"
}
```

При каждом новом старте:

- Обнуляется счёт.
- Скрываются элементы UI ("Play" и "Game Over").

7) Метод **Resume()** (сброс параметров перед началом игры):

```
private void Resume()
{
    Time.timeScale = 1f; // Возобновляем игровое время
    player.enabled = true; // Включаем управление игроком
}
```

При каждом новом старте:

- Обнуляется счёт.
- Скрываются элементы UI ("Play" и "Game Over").

8) Метод **ClearPipes()** (удаление старых труб):

```
private void ClearPipes()
{
    foreach (Pipes pipe in FindObjectsOfType<Pipes>()) // Находим все трубы в сцене
    {
        Destroy(pipe.gameObject); // Удаляем каждую трубу
    }
}
```

Перед началом новой игры нужно убрать **все старые трубы**, которые остались с прошлого запуска:

- FindObjectsOfType<Pipes>()** находит все трубы в сцене.

В **Unity 6** лучше использовать (более быстрый и оптимизированный, позволяет искать и сортировать объекты):

```
FindObjectsOfType<Pipes>(FindObjectsInactive.Exclude,
FindObjectsSortMode.None)
```

◆ *Параметры:*

- *FindObjectsInactive.Exclude* – ищет **только активные** объекты (как старый метод).
- *FindObjectsSortMode.None* – не сортирует найденные объекты (ускоряет поиск).

- Destroy(pipe.gameObject)** удаляет их.

9) После вызовем наши методы в методе **Play()** (начало игры):

```
public void Play()
{
    ResetGame(); // Сбрасываем игровые параметры
    Resume(); // Запускаем игру
    ClearPipes(); // Удаляем все трубы перед стартом
}
```

## 10) Метод GameOver() (завершение игры):

```
public void GameOver()
{
    gameOver.SetActive(true); // Отображаем экран "Game
Over"
    playButton.SetActive(true); // Показываем кнопку "Play"
    Pause(); // Останавливаем игру
}
```

Вызывается, когда игрок сталкивается с препятствием:

- Показывается экран "Game Over".
- Отображается кнопка "Play".
- Игра ставится на паузу.

## 11) Метод IncreaseScore() (увеличение очков):

```
public void IncreaseScore()
{
    score++; // Увеличиваем счёт на 1
    scoreText.text = score.ToString(); // Обновляем UI-
отображение счёта
}
```

Этот метод вызывается, когда игрок проходит сквозь препятствие:

- Увеличивает значение **score**.
- Обновляет отображение счёта в UI.

## Итоговый скрипт:

```
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [Header("Игровые объекты")]
    [SerializeField] private Player player;
    [SerializeField] private TextMeshProUGUI scoreText;
    [SerializeField] private GameObject playButton;
    [SerializeField] private GameObject gameOver;

    [Header("Игровые настройки")]
    [SerializeField] private int targetFrameRate = 60;

    private int score;

    private void Awake()
    {
        Application.targetFrameRate = targetFrameRate;
        Pause();
    }

    public void Play()
    {
        ResetGame();
        Resume();
        ClearPipes();
    }

    public void GameOver()
    {
        gameOver.SetActive(true);
        playButton.SetActive(true);
        Pause();
    }

    public void IncreaseScore()
    {
        score++;
        scoreText.text = score.ToString();
    }
}
```

```

private void Pause()
{
    Time.timeScale = 0f;
    player.enabled = false;
}

private void Resume()
{
    Time.timeScale = 1f;
    player.enabled = true;
}

private void ResetGame()
{
    score = 0;
    scoreText.text = score.ToString();
    playButton.SetActive(false);
    gameOver.SetActive(false);
}

private void ClearPipes()
{
    foreach (Pipes pipe in FindObjectsOfType<Pipes>())
    {
        Destroy(pipe.gameObject);
    }
}
}

```

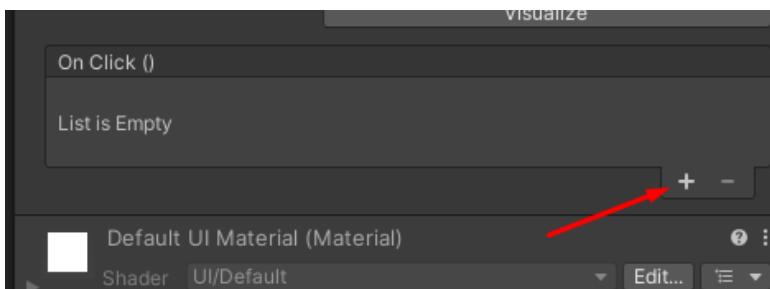
23. Также допишем в скрипт **Player** в самый конец новый метод:

```

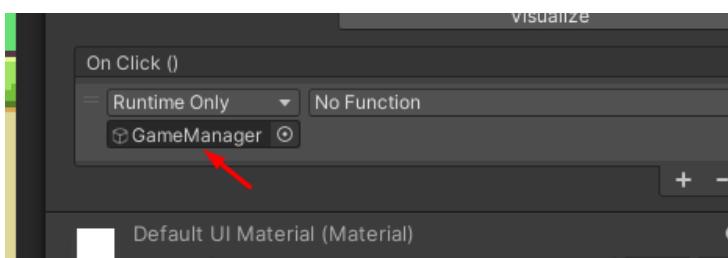
private void OnTriggerEnter2D(Collider2D other)
{
    // Если тег коллайдера - "Obstacle", эта строка
    // вызывает метод с именем GameOver() у экземпляра класса
    GameManager
    if (other.CompareTag("Obstacle"))
        FindFirstObjectByType<GameManager>().GameOver();
    // Если тег коллайдера - "Scoring", эта строка вызывает
    // метод с именем IncreaseScore() у экземпляра класса
    GameManager
    else if (other.CompareTag("Score"))
        FindFirstObjectByType<GameManager>().IncreaseScore(
    );
}

```

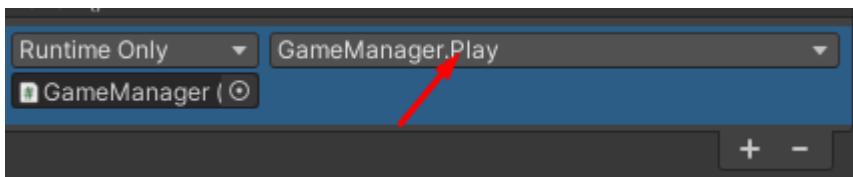
24. Выбираем нашу кнопку **PlayButton** и добавляем для неё новый список:



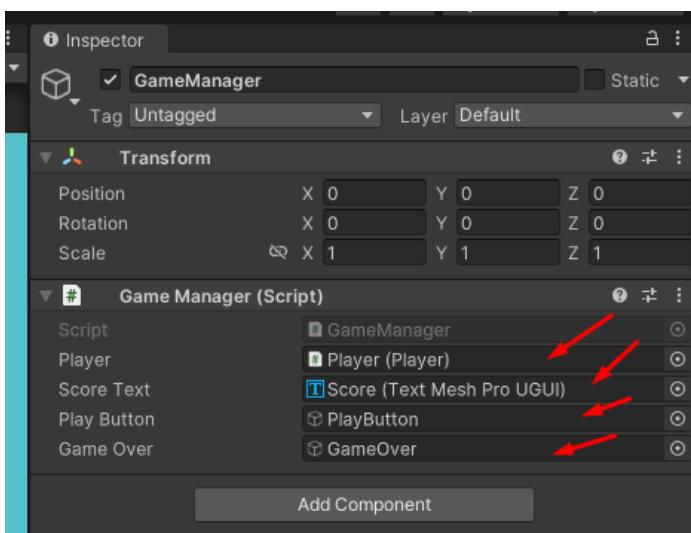
Добавляем на неё **GameManager**. Обратите внимание, что добавлять нужно **объект!!!**



Выбираем функцию **GameManager.Play**:

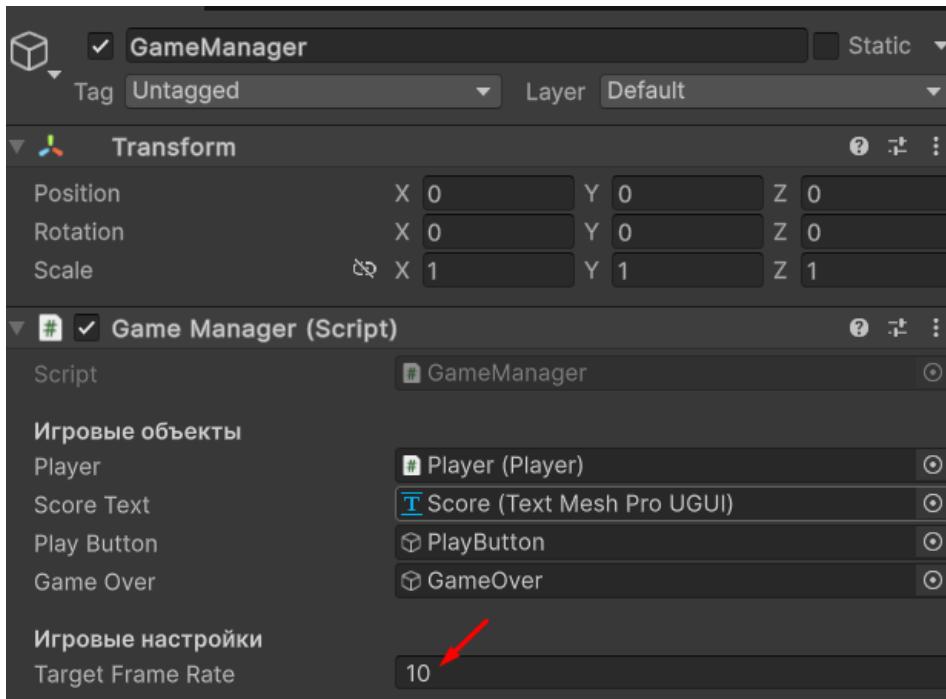


Также добавляем наши объекты в объекте **GameManager**:

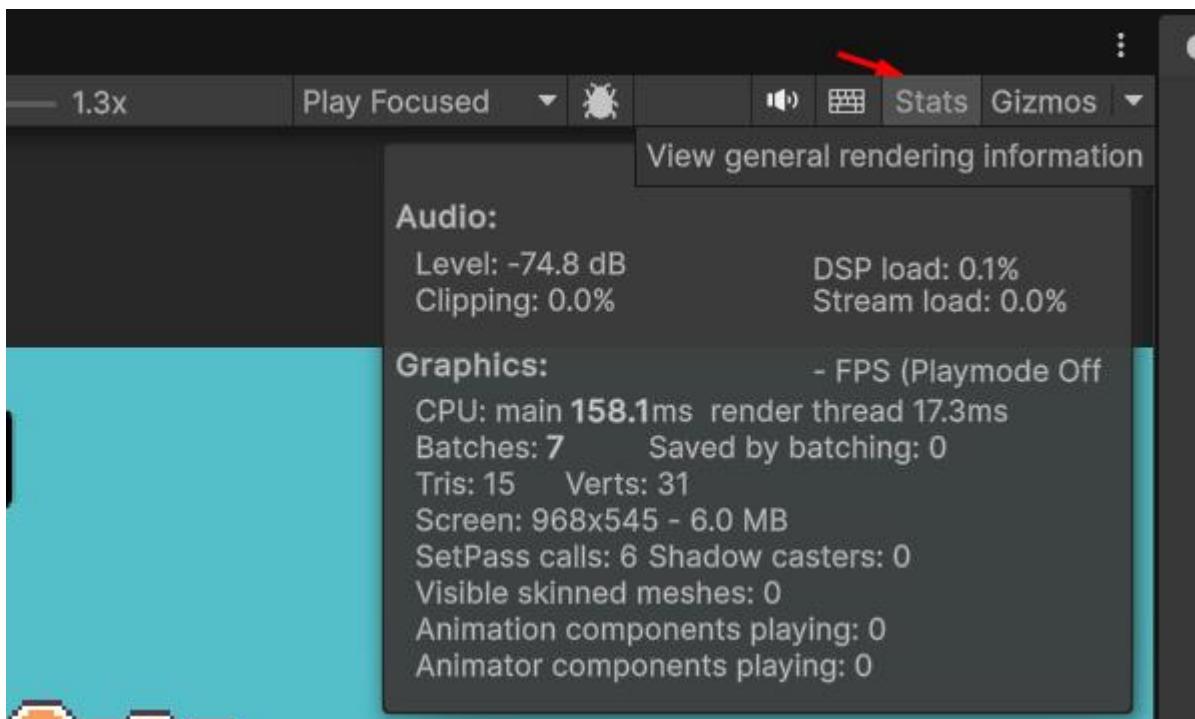


25. Запускаем и тестируем игру.

Попробуйте выставлять разное количество **Frame Rate** в параметрах у объекта **GameManager**. Например, гляньте как игра будет себя вести при **10** кадрах:



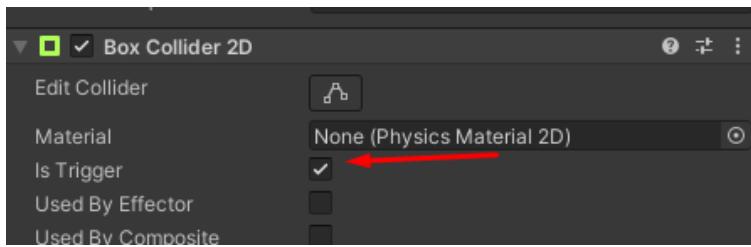
Также нажав на **Stats** у вас, будут выводиться параметры **FPS**, текущей загрузки процессора и другие параметры:



## 26. Фиксим баги.

### 1 Проблема. Игрок проваливается за землю, и игра не останавливается.

Дело в том, что мы забыли добавить коллайдер для нашей земли. Для решения этой проблемы добавим объекту **Ground - Box Collider 2D** и поставим галочку **Is Trigger**:



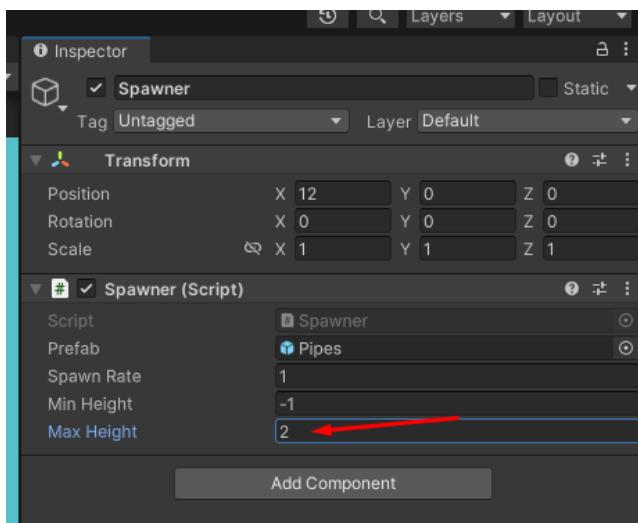
## 2 Проблема, при продолжении игры игрок спавнится в том же месте, где умер.

Чтобы решить эту проблему, нам нужно обнулять позицию нашего игрока.

Для этого переходим в скрипт **Player** и допишем метод:

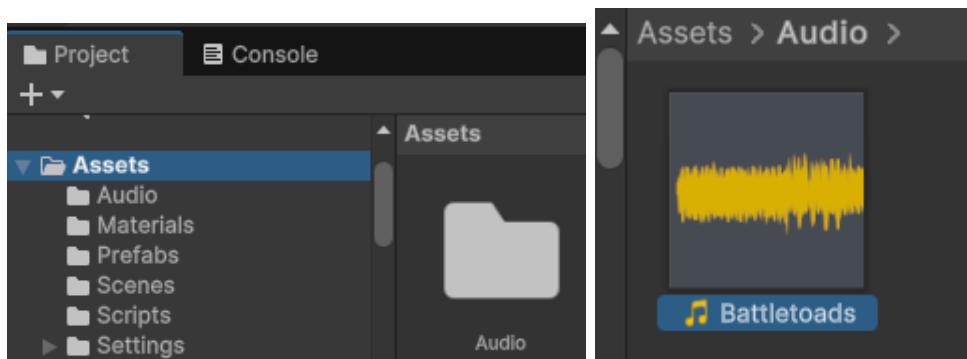
```
public void OnEnable() // Этот метод вызывается, когда
объект становится включенным и активным.
{
    transform.position = Vector3.zero; // Сбрасываем
позицию игрока в центр
    _direction = Vector3.zero; // Обнуляем движение
}
```

27. Для усложнения игры можно изменить для объекта **Spawner** макс. высоту:

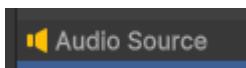


28. Добавим фоновую музыку.

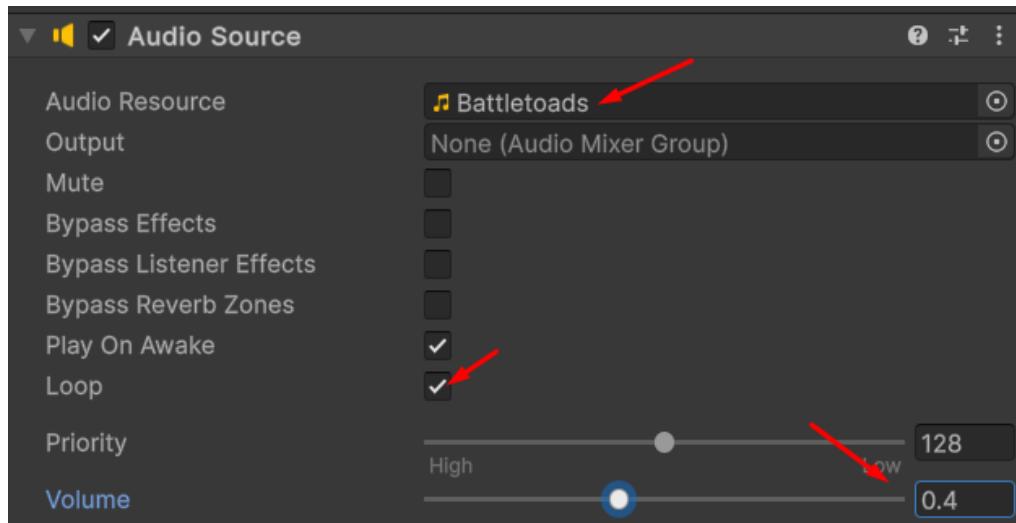
Создаём в ассетах папку **Audio**, переносим из ассетов музыку внутрь папки:



Добавляем для камеры компонент **Audio Source**:



Перетащите звук в источник, можете настроить громкость, и не забудьте поставить галочку на **Loop**, чтобы зациклить воспроизведение музыки:



Теперь сделаем так, чтобы музыка играла заново только при нажатии на кнопку **Play**. Для этого возвращаемся в скрипт **GameManager**.

Добавим **переменную** для нашей музыки:

```
[Header("Игровые настройки")]
[SerializeField] private int targetFrameRate = 60;
private int score;
private AudioSource _audioSource; // Переменная для музыки
```

Затем в методе **Awake** получим ссылку на неё (т.к. мы прикрепили её на камеру, то указываем это):

```
private void Awake()
{
    Application.targetFrameRate = targetFrameRate;
    Pause();
    _audioSource = Camera.main.GetComponent<
```

Добавим методы **PlayMusic** и **StopMusic** для вызова и остановки музыки:

```
private void PlayMusic() // метод для воспроизведения
музыки
{
    if (_audioSource != null)
        _audioSource.Play();
}
```

```
private void StopMusic() // метод для остановки музыки
{
    if (_audioSource != null && _audioSource.isPlaying)
        _audioSource.Stop();
}
```

Зачем проверять `_audioSource` на `null`?

- Проверка на `null` в методах `PlayMusic()` и `StopMusic()` нужна для предотвращения `NullReferenceException` и обеспечения устойчивости кода.
- `_audioSource` может быть `null`, если компонент не был присвоен, отсутствует на объекте или объект был уничтожен.

И теперь в методе `Awake` вызовем метод `StopMusic()`, чтобы музыка не воспроизводилась:

```
private void Awake()
{
    Application.targetFrameRate = targetFrameRate;
    Pause();
    _audioSource = Camera.main.GetComponent< AudioSource >(); // находим музыку
    StopMusic(); // отключаем музыку в начале игры ←
```

Добавляем остановку музыки в методах `GameOver()`:

```
public void GameOver()
{
    gameOver.SetActive(true);
    playButton.SetActive(true);
    Pause();
    StopMusic(); // останавливаем музыку ←
```

В методе `Pause()`:

```
private void Pause()
{
    Time.timeScale = 0f;
    player.enabled = false;
    StopMusic(); // останавливаем музыку ←
```

В методе `Play` сделаем запуск музыки:

```
public void Play()
{
    ResetGame();
    Resume();
    ClearPipes();
    PlayMusic(); // запускаем музыку ←
```

29. Чтобы сделать игру интереснее, давайте добавим увеличение скорости при наборе очков. Допишем два метода в скрипте **Pipes**:

```
public static void IncreaseSpeed(float increment) // Этот
метод увеличивает скорость всех труб на указанное значение
increment. Используя этот метод, мы можем постепенно
увеличивать сложность игры, повышая скорость труб при
увеличении очков игрока.
{
    speed += increment;
}
public static void SetSpeed(float newSpeed) // Этот метод
устанавливает новую скорость для всех труб. Он используется
для сброса скорости труб до исходного значения при
перезапуске уровня.
{
    speed = newSpeed;
}
```

Переменную скорости изменяем на **static**:

```
public class Pipes : MonoBehaviour
{
    [Header("Настройки движения")]
    [SerializeField] private static float speed = 5f;
    private float leftEdge;
```

Зачем изменяем на **static**? Сделав переменную **speed** статической, мы обеспечиваем, что она будет общей для всех экземпляров класса **Pipes**. Это позволяет нам управлять скоростью всех труб одновременно, что важно для увеличения скорости при увеличении очков. Если бы **speed** не был статическим, у каждой трубы была бы своя собственная скорость, и изменение скорости одной трубы не влияло бы на другие.

Дополним скрипт **GameManager**:

➤ Добавим переменные:

```
[Header("Настройка труб")]
// Увеличение скорости труб
[SerializeField] private float pipeSpeedIncrement = 0.5f;
// Начальная скорость труб
[SerializeField] private float defaultPipeSpeed = 5f;
```

➤ В методе **IncreaseScore** добавим увеличение скорости труб:

```
public void IncreaseScore()
{
    score++;
    scoreText.text = score.ToString();

    // Увеличиваем скорость труб
    Pipes.IncreaseSpeed(pipeSpeedIncrement);
}
```

- Создадим новый метод для сброса скорости труб **PipeReset()**:

```
private void PipeReset()
{
    Pipes.SetSpeed(defaultPipeSpeed); // сбрасываем скорость
    труб
}
```

- В методе **Play()** вызовем его:

```
Ссылка: 0
public void Play()
{
    ResetGame();
    Resume();
    ClearPipes();
    → PipeReset(); // сбрасываем скорость труб
    PlayMusic();
}
```

### 30. Рефакторинг кода.

Наш код неплох, но мы можем немного улучшить его:

- Оптимизировать работу с музыкой** (избежать повторного **StopMusic()**)
- Сделать названия методов более точными**
- Объединить логику Pause и StopMusic()**
- Объединить PipeReset() с ClearPipes() для удобства**

**Вносим правки**

- 1) **Методы Pause()** и **Resume()** переименовываем в **StopGame()** и **ResumeGame()**

И заодно вызовем музыку в методе **ResumeGame**. Для изменения имени по всему коду можно воспользоваться сочетанием клавиш **Ctrl + R + R**:

```

private void Pause()
{
    Time.timeScale = 0f;
    player.enabled = false;
    StopMusic();
}

Ссылок: 1
private void Resume()
{
    Time.timeScale = 1f;
    player.enabled = true;
}

```

**Было**

```

Ссылок: 2
private void StopGame()
{
    Time.timeScale = 0f;
    player.enabled = false;
    StopMusic();
}

Ссылок: 1
private void ResumeGame()
{
    Time.timeScale = 1f;
    player.enabled = true;
    PlayMusic(); ←
}

```

**Стало**

**2) Объединим ClearPipes() и PipeReset() в один метод ResetPipes():**

```

private void ClearPipes()
{
    foreach (Pipes pipe in FindObjectsByType<Pipes>
        (FindObjectsInactive.Exclude, FindObjectsSortMode.None))
    {
        Destroy(pipe.gameObject);
    }
}

Ссылок: 1
private void PipeReset()
{
    Pipes.SetSpeed(defaultPipeSpeed);
}

```

**Было**

```

private void ResetPipes()
{
    foreach (Pipes pipe in FindObjectsByType<Pipes>
        (FindObjectsInactive.Exclude, FindObjectsSortMode.None))
    {
        Destroy(pipe.gameObject);
    }
    Pipes.SetSpeed(defaultPipeSpeed);
}

```

**Стало**

Вызовем его в методе ResetGame():

```

private void ResetGame()
{
    score = 0;
    scoreText.text = score.ToString();
    playButton.SetActive(false);
    gameOver.SetActive(false);
    ResetPipes(); // вызываем метод
}

```

3) В методе **Play** вызовем только два метода:

```
public void Play()
{
    ResetGame();
    ResumeGame();
    ClearPipes();
    PipeReset();
    PlayMusic();
}
```

Было

```
public void Play()
{
    ResetGame();
    ResumeGame();
}
```

Стало

4) В методе **GameOver()** убираем лишний вызов музыки:

```
public void GameOver()
{
    gameOver.SetActive(true);
    playButton.SetActive(true);
    StopGame();
    StopMusic();
}
```

5) В методе **PlayMusic()** добавим дополнительную проверку, чтобы исключить повторный запуск музыки, если она уже играет:

```
private void PlayMusic()
{
    if (_audioSource != null && !_audioSource.isPlaying)
        _audioSource.Play();
```

6) В методе **Awake()** оставим вызов только **StopGame()**, чтобы сразу останавливалась и игра и музыка:

```
private void Awake()
{
    Application.targetFrameRate = targetFrameRate;
    _audioSource = Camera.main.GetComponent< AudioSource>();
    StopGame(); ←
}
```

## ИТОГОВЫЙ код:

```
using TMPro;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [Header("Игровые объекты")]
    [SerializeField] private Player player;
    [SerializeField] private TextMeshProUGUI scoreText;
    [SerializeField] private GameObject playButton;
    [SerializeField] private GameObject gameOver;

    [Header("Игровые настройки")]
    [SerializeField] private int targetFrameRate = 60;

    [Header("Настройка труб")]
    [SerializeField] private float pipeSpeedIncrement = 0.5f;
    [SerializeField] private float defaultPipeSpeed = 5f;

    private int score;
    private AudioSource _audioSource;

    private void Awake()
    {
        Application.targetFrameRate = targetFrameRate;
        _audioSource = Camera.main.GetComponent<AudioSource>();
        StopGame();
    }

    private void PlayMusic()
    {
        if (_audioSource != null && !_audioSource.isPlaying)
            _audioSource.Play();
    }

    private void StopMusic()
    {
        if (_audioSource != null && _audioSource.isPlaying)
            _audioSource.Stop();
    }

    public void Play()
    {
        ResetGame();
        ResumeGame();
    }
}
```

```
public void GameOver()
{
    gameOver.SetActive(true);
    playButton.SetActive(true);
    StopGame();
}

public void IncreaseScore()
{
    score++;
    scoreText.text = score.ToString();
    Pipes.IncreaseSpeed(pipeSpeedIncrement);
}

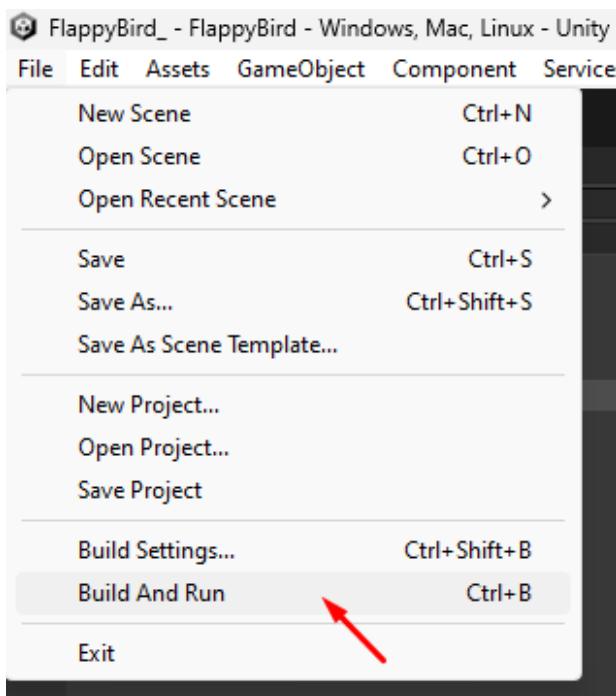
private void StopGame()
{
    Time.timeScale = 0f;
    player.enabled = false;
    StopMusic();
}

private void ResumeGame()
{
    Time.timeScale = 1f;
    player.enabled = true;
    PlayMusic();
}

private void ResetGame()
{
    score = 0;
    scoreText.text = score.ToString();
    playButton.SetActive(false);
    gameOver.SetActive(false);
    ResetPipes();
}

private void ResetPipes()
{
    foreach (Pipes pipe in FindObjectsOfType<Pipes>(FindObjectsInactive.Exclude,
FindObjectsSortMode.None))
    {
        Destroy(pipe.gameObject);
    }
    Pipes.SetSpeed(defaultPipeSpeed);
}
```

31. Осталось только скомпилировать нашу игру. Переходим в **File – Build And Run:**



Выбираете любую папку куда хотите сохранить игру или создайте новую папку.

После можете запустить игру через .exe.