

Лабораторная работа. Создание 2D-игры Asteroids.

Данный урок создан на основе следующего видео-урока: [How to make Asteroids 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: "Player", "Asteroid", "Bullet", "Spawner".
- Реализовать три сцены:
 - Главное меню
 - Игровая сцена
 - Сцена конца игры (с отображением итогового счета).
- Настроить систему частиц для эффектов взрывов.

3. Разработка игровой логики:

- Реализовать управление игроком с возможностью передвижения и стрельбы (Player).
- Настроить спавн астероидов в случайных точках с разными размерами и направлениями движения (AsteroidSpawner).
- Реализовать уничтожение астероидов при попадании пули и добавление очков (GameManager).
- Внедрить механику жизней – у игрока 3 жизни, при их исчерпании игра заканчивается.

4. Улучшение игровых механик:

- Добавить динамическое изменение сложности, увеличивая частоту появления астероидов со временем.
- Реализовать эффект взрывов при уничтожении астероидов.
- Добавить перезапуск игры при нажатии на пробел на экране конца игры (EndGameManager).

5. Интерфейс пользователя и звуковое сопровождение:

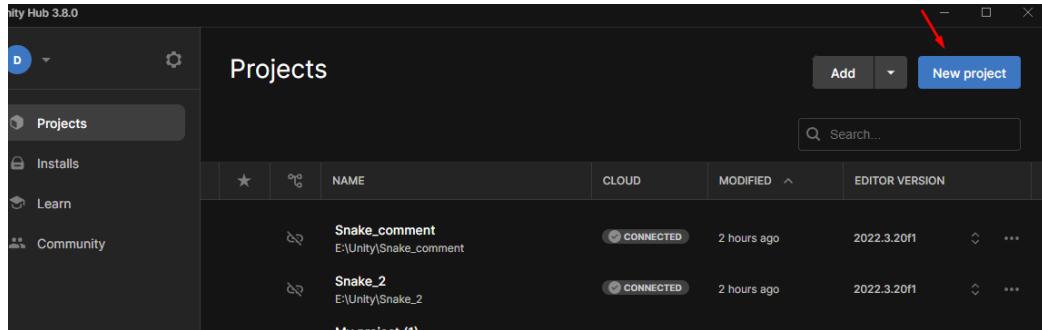
- Отобразить количество жизней и очки на экране (GameManager).
- Добавить звуковые эффекты для выстрелов, столкновений и уничтожения астероидов (ExplosionSoundPlayer, AsteroidSoundPlayer).

6. Финальные настройки и сборка проекта:

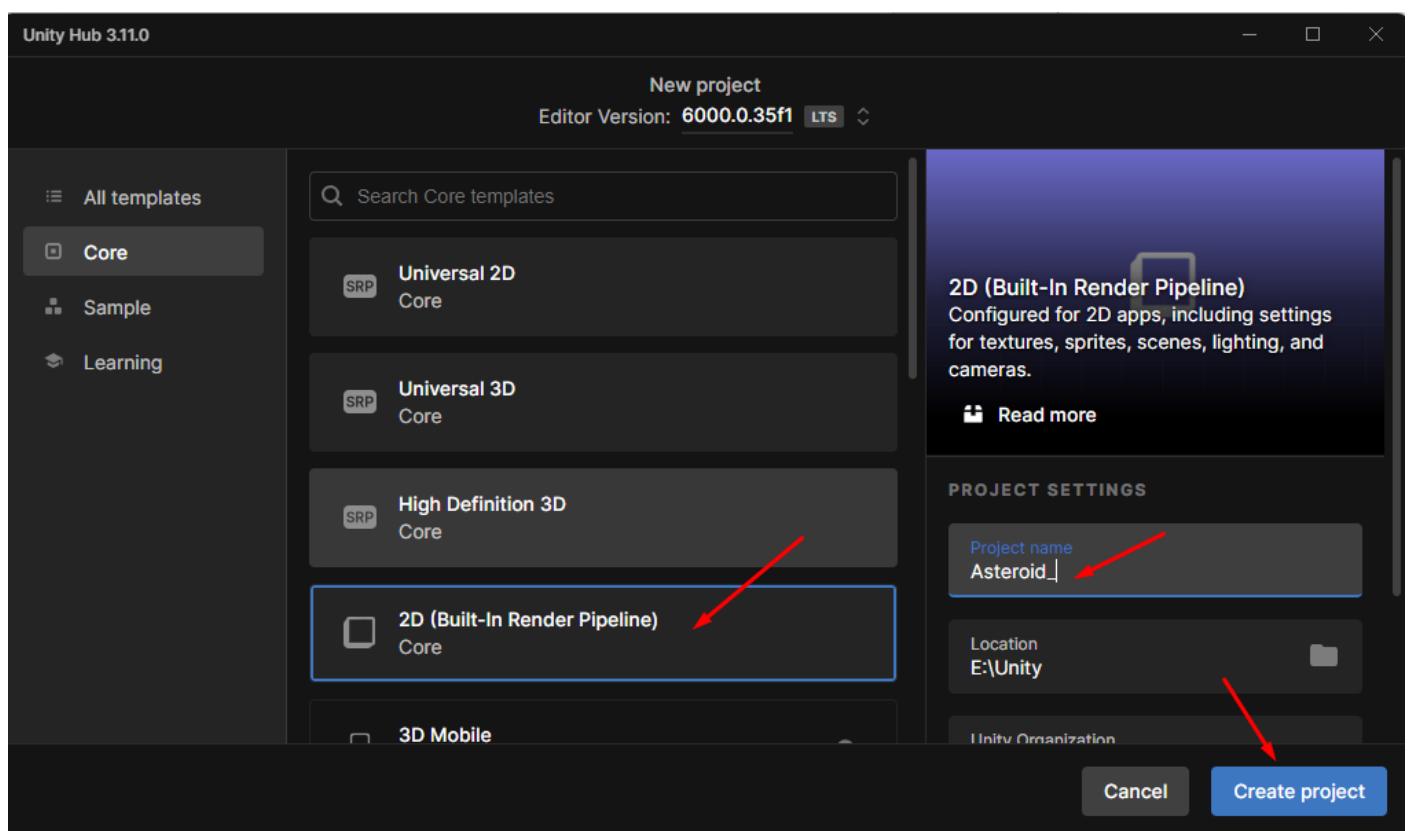
- Провести рефакторинг кода для улучшения читаемости и производительности.
- Скомпилировать и сохранить финальную версию игры.

1. Запускаем Unity Hub.

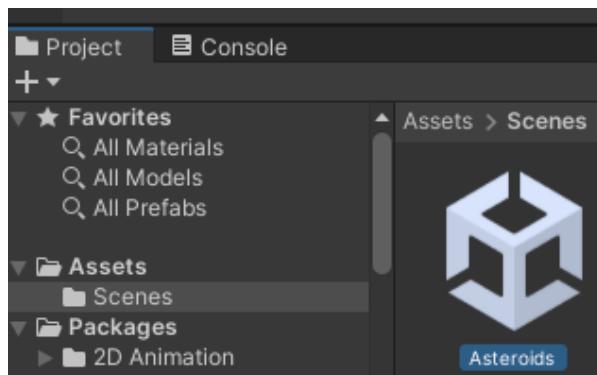
Создаём новый проект – **New project**:



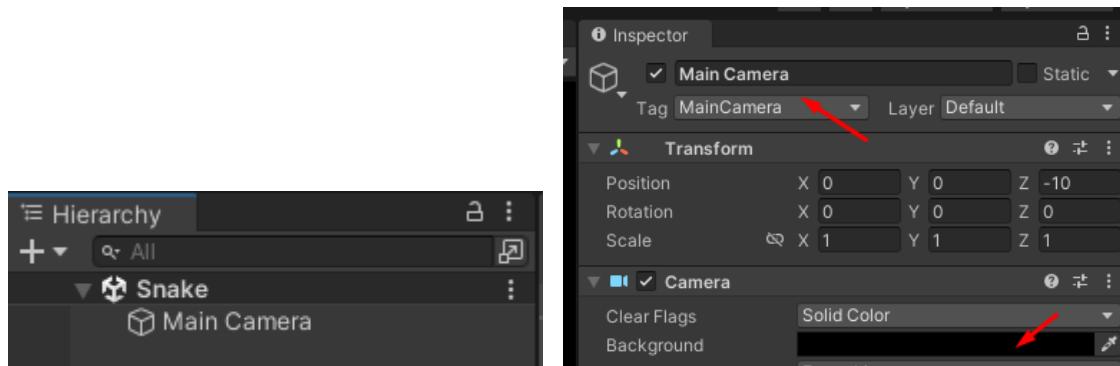
Выбираем **2D(Built-In Render Pipeline)**, если он у вас отсутствует, то скачайте (его плюс, что размер меньше на 400 Мб, и для нашего проекта его вполне хватит). Вводим название проекта, выбираем место расположения, и нажимаем **Create project**.



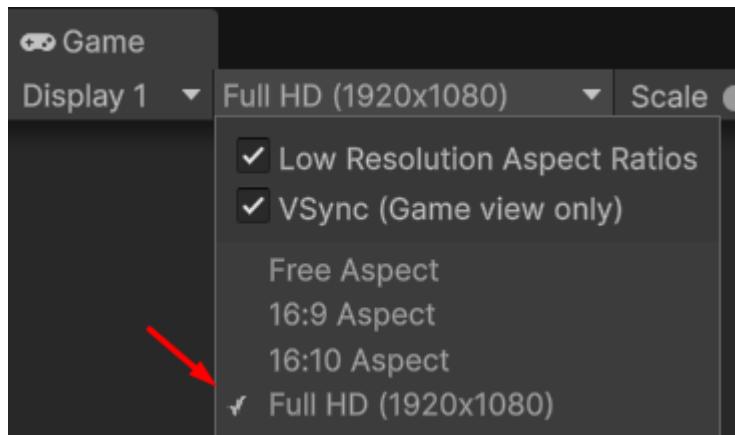
2. В папке **Scenes** меняем название сцены, на имя игры **Asteroids**:



Меняем цвета объекта **Main Camera** (нажимаем на него в **Hierarchy**, и в **Inspector** появляются свойства объекта) на **чёрный**:

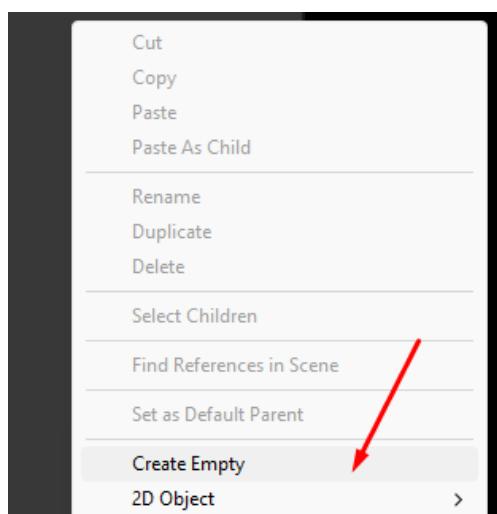


Поменяем на вкладке **Game** отношение сторон на **1920x1080**:

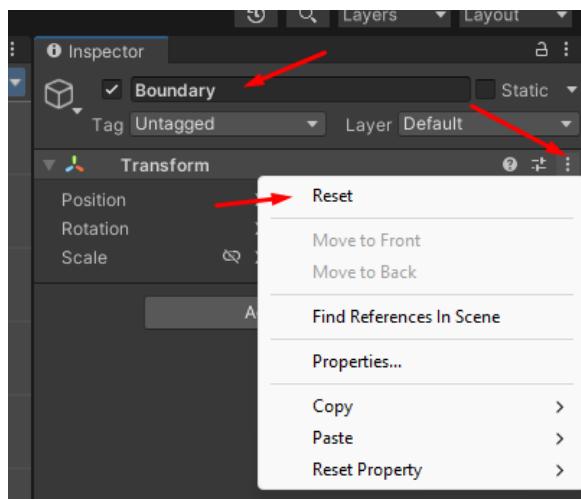


3. Теперь создадим стены, по бокам нашего экрана, чтобы игрок не мог выходить за их пределы.

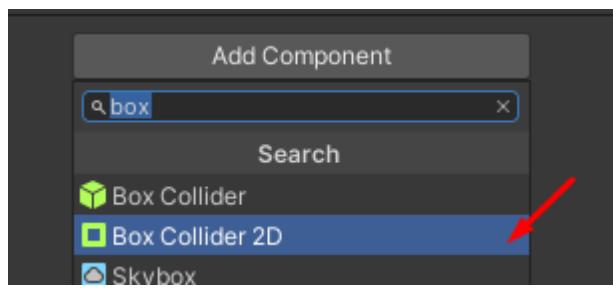
Создаём новый объект – щёлкаем **правой кнопкой мыши** в **иерархии** - **Creaty Empty**:



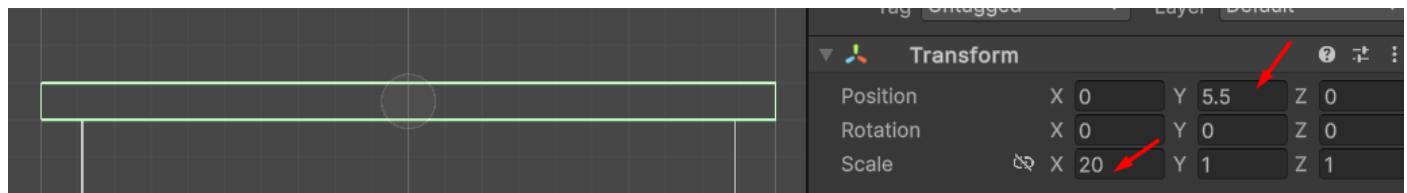
Называем его **Boundary** и сбрасываем трансформацию:



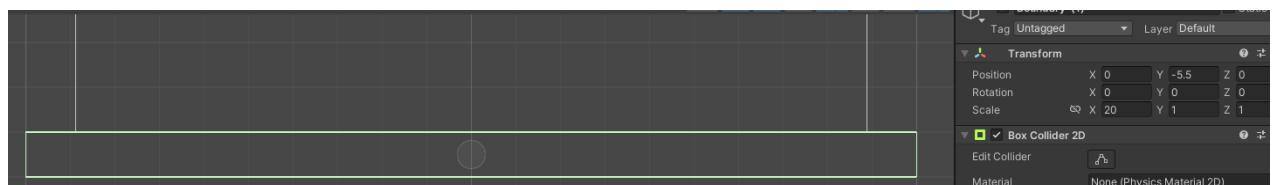
Добавляем **Box Collider 2D**:



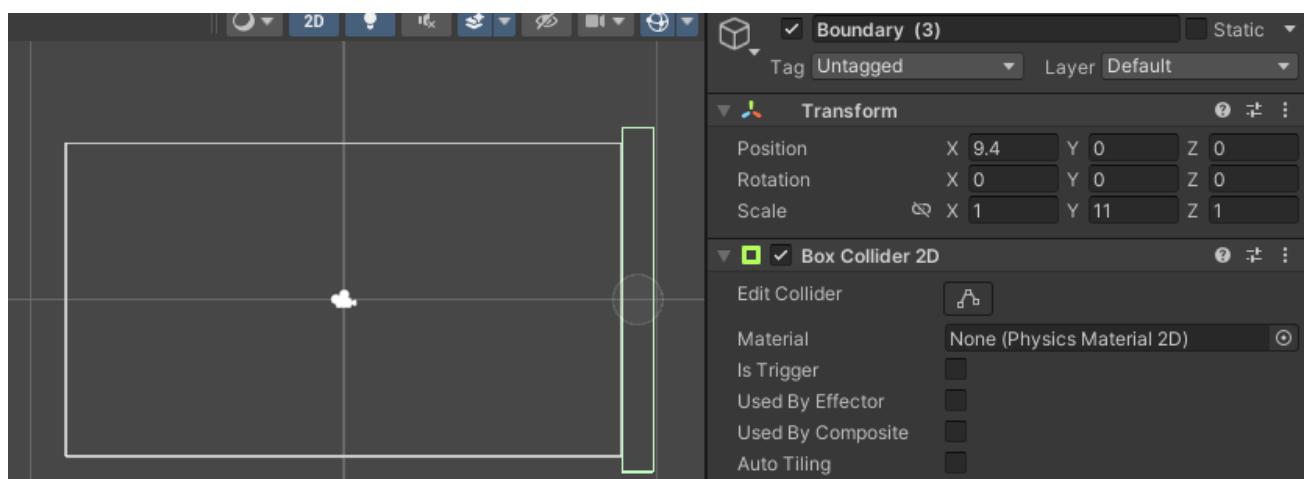
Настраиваем **размер и позицию** по оси **Y**, чтобы он находился **выше** камеры:



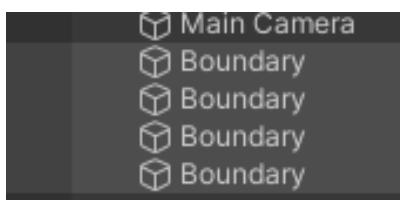
Создаём дубликат и опускаем его ниже по оси **Y**:



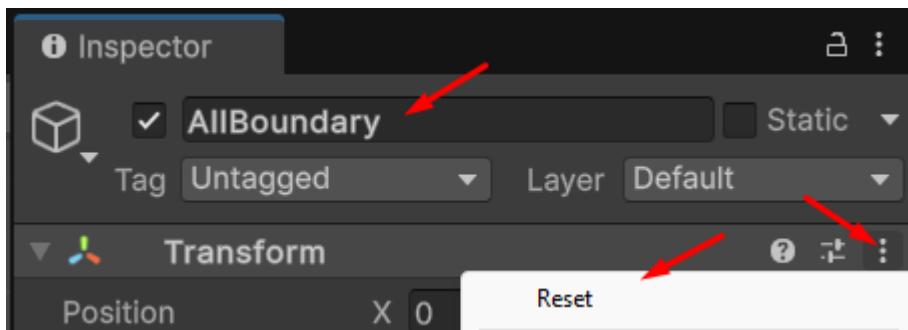
Аналогично создаём дубликаты и настраиваем левую и правую стену:



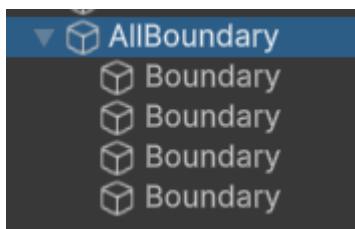
Выделите все 4 объекта и переименуйте их на **Boundary**:



Создайте пустой объект, назовите его **AllBoundary** и сбросьте для него трансформацию:



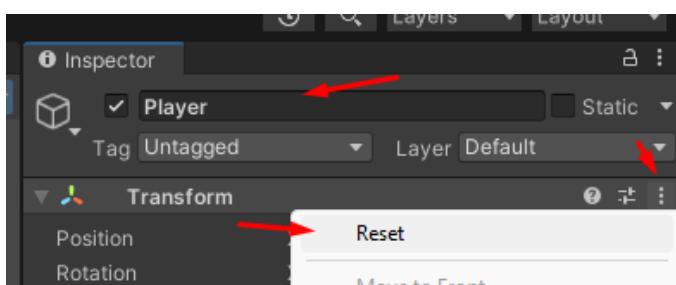
Перенесите внутрь него все стены:



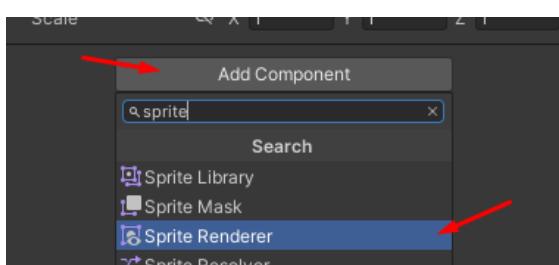
4. Создаём новый объект – щёлкаем правой кнопкой мыши в иерархии - **Creaty Empty**:



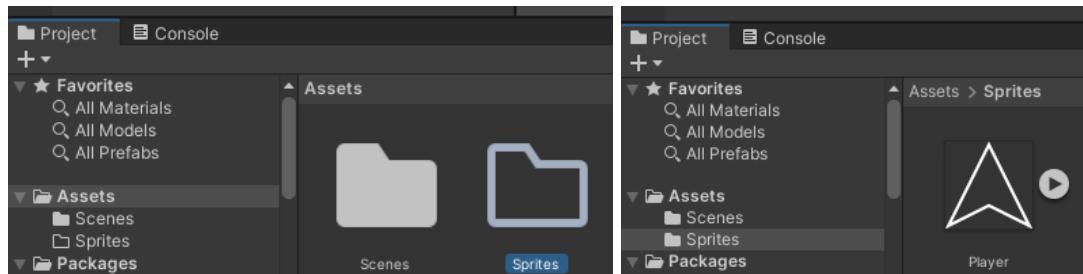
Называем его **Player** и сбрасываем трансформацию:



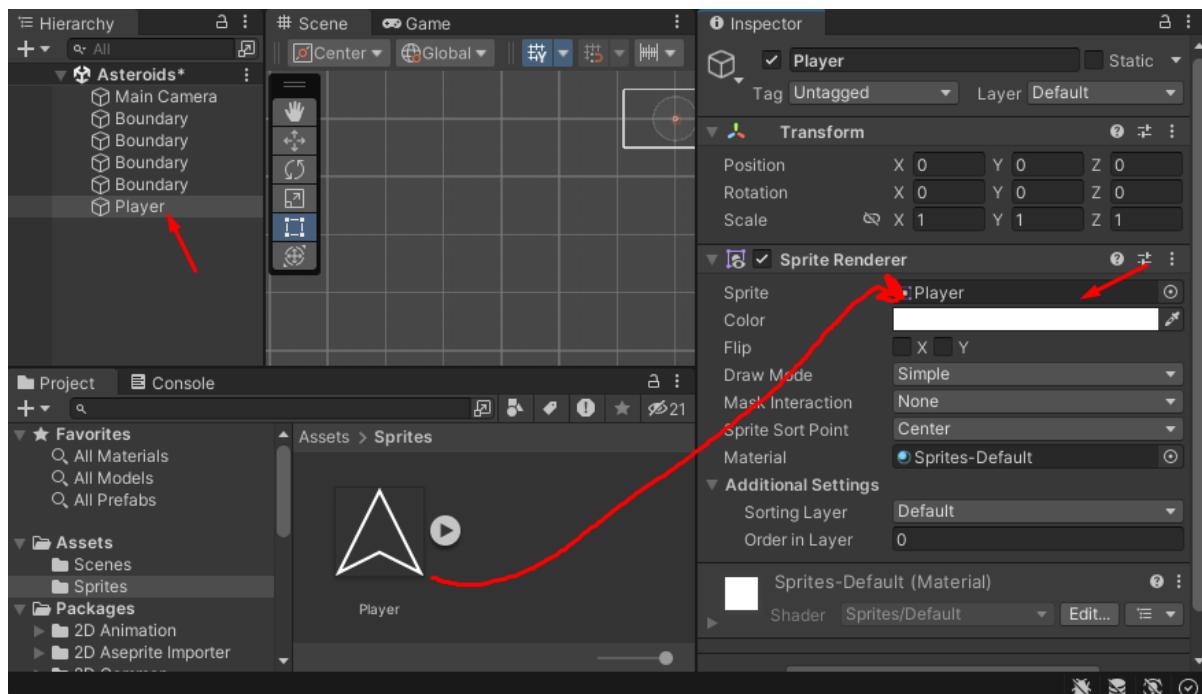
Добавляем ему компонент **Sprite Renderer**:



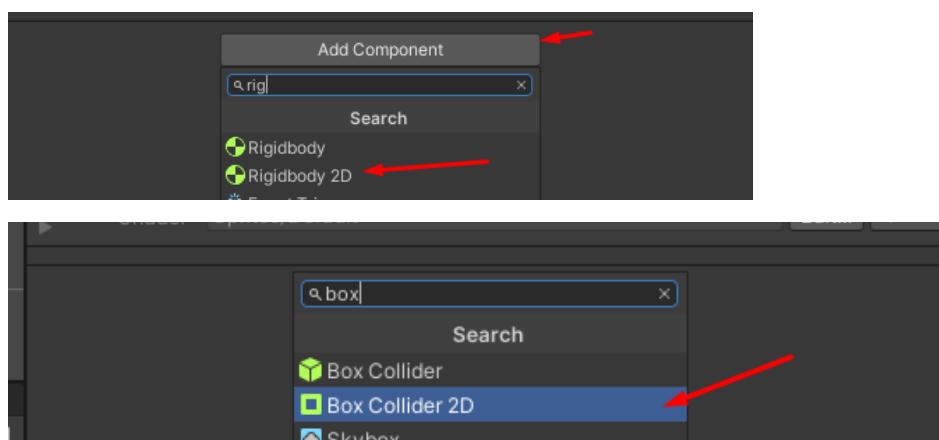
Далее в наших ассетах создадим папку **Sprites** и подгрузим спрайт **Player**:



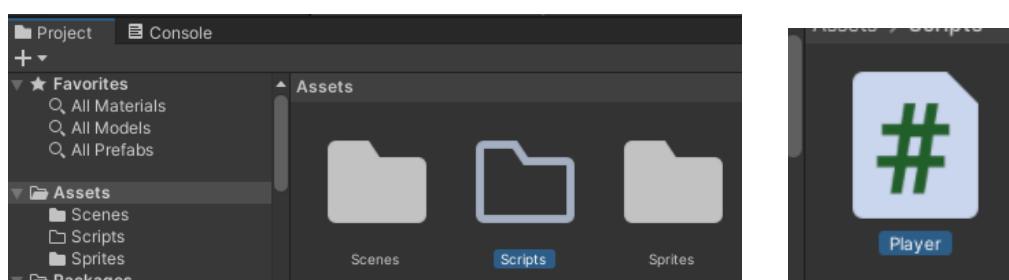
Добавим его на наш объект:



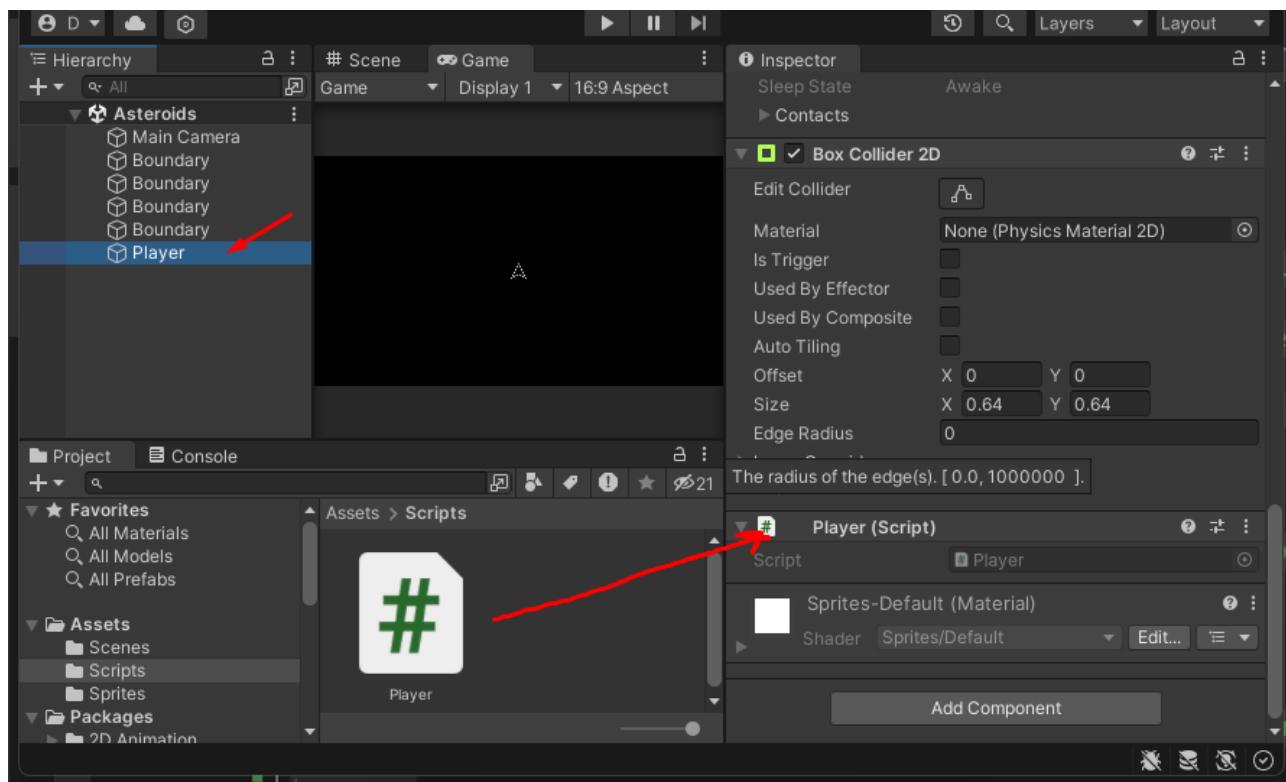
Добавляем компонент **Rigidbody 2D** и **Box Collider 2D**:



5. Теперь напишем скрипт для нашего игрока. Создаём папку **Scripts** и в ней создаём скрипт **Player**:



Переносим скрипт Player на игрока:



Открываем скрипт Player. Пропишем следующий код для управления движением и поворотом игрока в зависимости от нажатых клавиш.

➤ Добавляем переменные в начале скрипта:

```
[Header("Speed")]
// скорость движения вперёд
[SerializeField] private float thrustSpeed = 3f;
// скорость поворота
[SerializeField] private float turnSpeed = 0.1f;

// флаг для отслеживания нажатия кнопки движения
private bool _thrusting;
// сохраняем направление скорости
private float _turnDirection;
// ссылка на компонент Rigidbody2D
private Rigidbody2D _rigidbody;
```

➤ Создаём метод Awake(), в котором находим Rigidbody2D:

```
private void Awake()
{
    // находит компонент Rigidbody2D на объекте и сохраняет в
    // переменную _rigidbody
    _rigidbody = GetComponent<Rigidbody2D>();
```

➤ Реализуем обработку ввода HandleInput():

```

private void HandleInput()
{
    //Если нажата W или ↑ → включаем _thrusting
    _thrusting = Input.GetKey(KeyCode.W) ||
Input.GetKey(KeyCode.UpArrow);
    //Если нажата A или ← → 1.0f (поворот влево).
    if (Input.GetKey(KeyCode.A) ||
Input.GetKey(KeyCode.LeftArrow))
    {
        _turnDirection = 1.0f;
    }
    //Если нажата D или → → -1.0f (поворот вправо).
    else if (Input.GetKey(KeyCode.D) ||
Input.GetKey(KeyCode.RightArrow))
    {
        _turnDirection = -1.0f;
    }
    //Если клавиши не нажаты → _turnDirection = 0.0f.
    else
        _turnDirection = 0.0f;
}

```

➤ Добавляем обработку ввода Update():

```

private void Update()
{
    HandleInput();
}

```

➤ Реализуем движение ApplyMovement():

```

private void ApplyMovement()
{
    if (_thrusting)//Если игрок ускоряется → AddForce() толкает
его вперёд
        _rigidbody.AddForce(transform.up * thrustSpeed);
    if (_turnDirection != 0.0f)//Если игрок поворачивает →
AddTorque() вращает его
        _rigidbody.AddTorque(_turnDirection * turnSpeed);
}

```

➤ Добавляем физику движения в FixedUpdate():

```

private void FixedUpdate()
{
    ApplyMovement();
}

```

ИТОГОВЫЙ КОД:

```
[Header("Speed")]
[SerializeField] private float thrustSpeed = 3f;
[SerializeField] private float turnSpeed = 0.1f;
private bool _thrusting;
private float _turnDirection;
private Rigidbody2D _rigidbody;

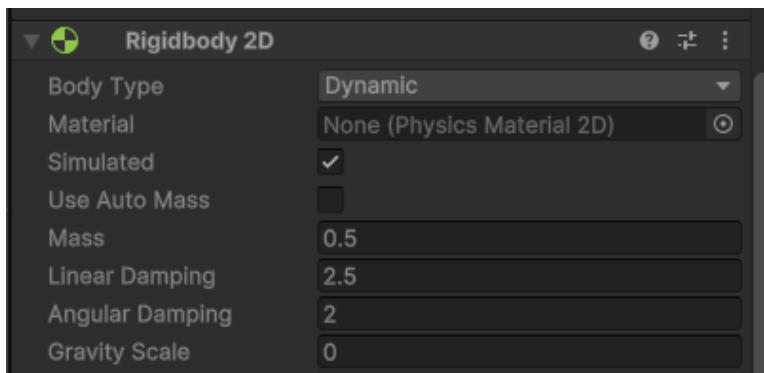
private void Awake()
{
    _rigidbody = GetComponent<Rigidbody2D>();
}

private void Update()
{
    HandleInput();
}
private void FixedUpdate()
{
    ApplyMovement();
}

private void HandleInput()
{
    _thrusting = Input.GetKey(KeyCode.W) ||
Input.GetKey(KeyCode.UpArrow);
    if (Input.GetKey(KeyCode.A) ||
Input.GetKey(KeyCode.LeftArrow))
        _turnDirection = 1.0f;
    else if (Input.GetKey(KeyCode.D) ||
Input.GetKey(KeyCode.RightArrow))
        _turnDirection = -1.0f;
    else
        _turnDirection = 0.0f;
}

private void ApplyMovement()
{
    if (_thrusting)
        _rigidbody.AddForce(transform.up * thrustSpeed);
    if (_turnDirection != 0.0f)
        _rigidbody.AddTorque(_turnDirection * turnSpeed);
}
```

Для объекта **Player** поменяем наши параметры в **Rigidbody 2D**:



Mass – 0,5 (определяет массу объекта. Объект с меньшей массой легче перемещать и ускорять);

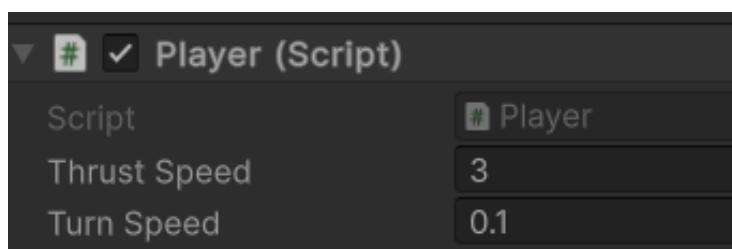
Linear Damping – 2.5 (управляет скоростью уменьшения линейного движения объекта со временем);

Angular Damping – 2 (управляет

скоростью уменьшения углового движения (вращения) объекта со временем. Значение 1 создаст заметное сопротивление вращению, что сделает вращение объекта более плавным и быстро затухающим);

Gravity Scale – 0.

А также значение скорости и скорости вращения:



Thrust Speed – 3 (Значение 3 выбрано для создания сбалансированного ускорения, позволяющего игроку контролировать движение без чрезмерной скорости);

Turn Speed – 0.3 (Значение 0.3

выбрано для плавного и контролируемого вращения.).

2 способ передвижения

Если вы хотите сделать управление персонажем более отзывчивым и резким, то можете поменять наши параметры на следующие:

Mass – 0,5;

Linear Damping – 10;

Angular Damping – 10;

Thrust Speed – 20;

Turn Speed – 0.5.

3 способ передвижения.

И последний вариант, для ещё более отзывчивого управления:

Mass – 1;

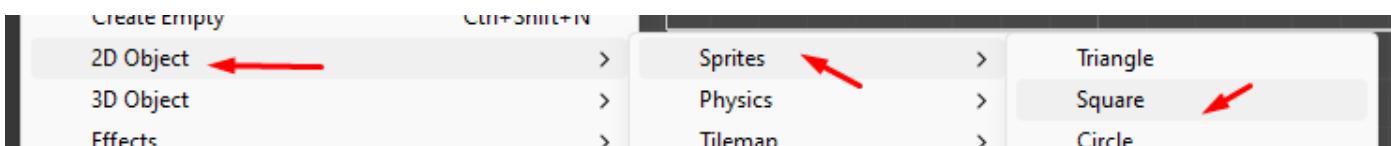
Linear Damping – 100;

Angular Damping – 100;

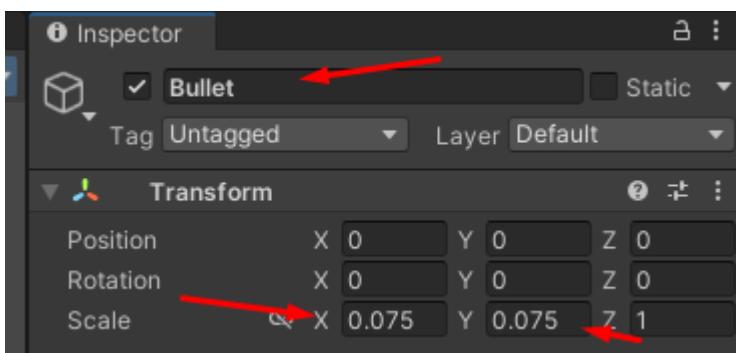
Thrust Speed – 350;

Turn Speed – 15.

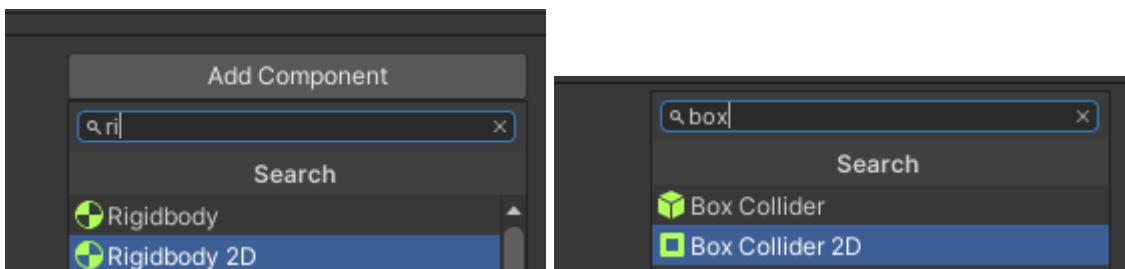
6. Займёмся стрельбой. Создаём новый 2D Object – Sprites - Square:



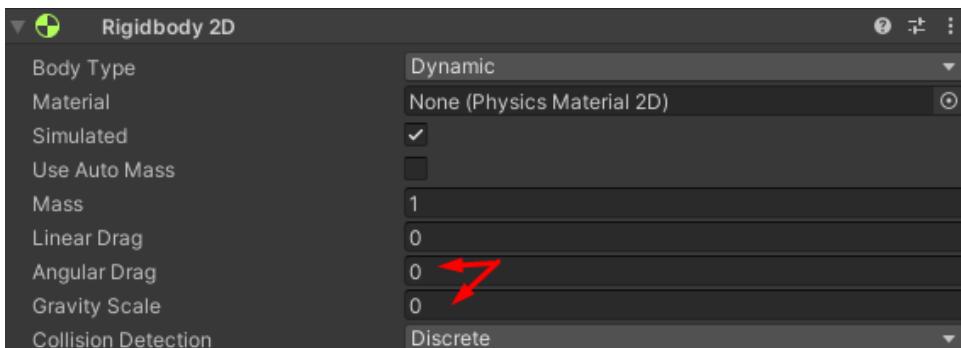
Называем его **Bullet**, сбрасываем трансформацию и уменьшаем размер до **0.075** по **x** и **y**:



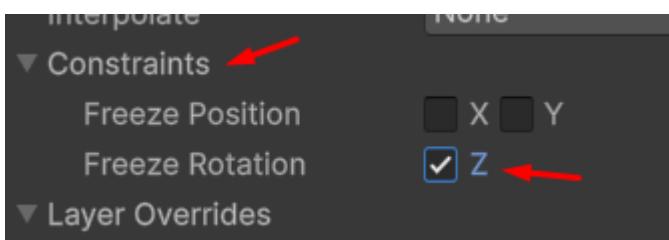
Добавляем **Rigidbody 2D** и **Box Collider 2D**:



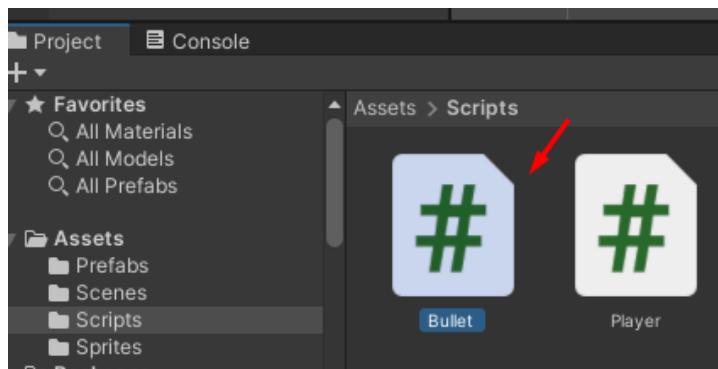
Нам не нужны такие параметры как **гравитация** и **Angular Damping** (отвечает за сопротивление), поэтому убираем их в **0**:



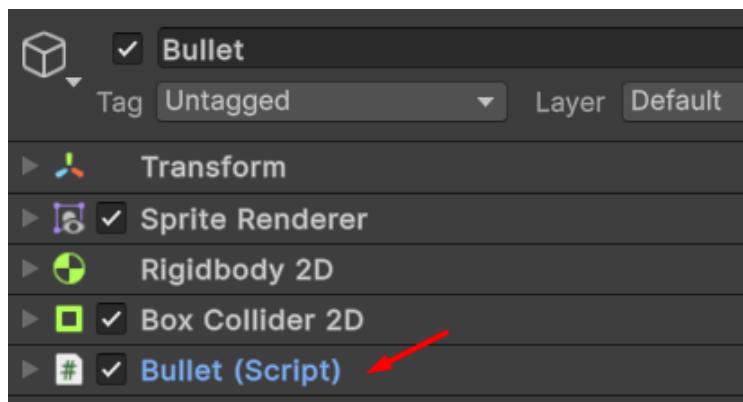
Также уберём с вами вращение по оси **Z**, «заморозим» её, для этого раскройте **Contains – Freeze Rotation**:



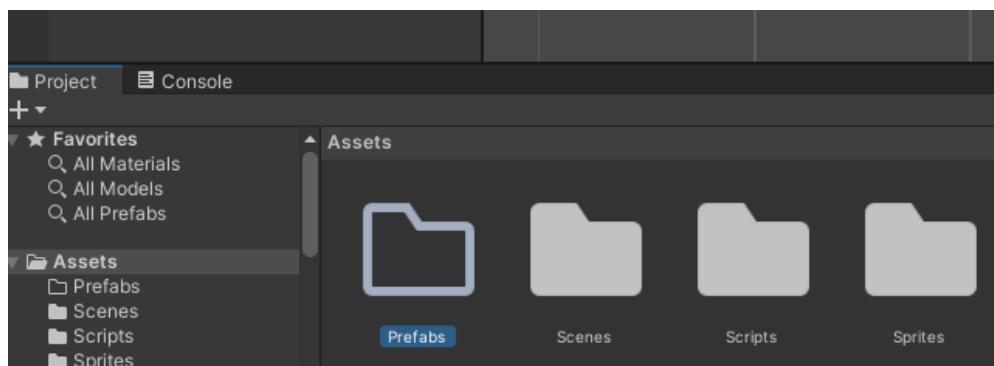
Создаём новый скрипт **Bullet**:



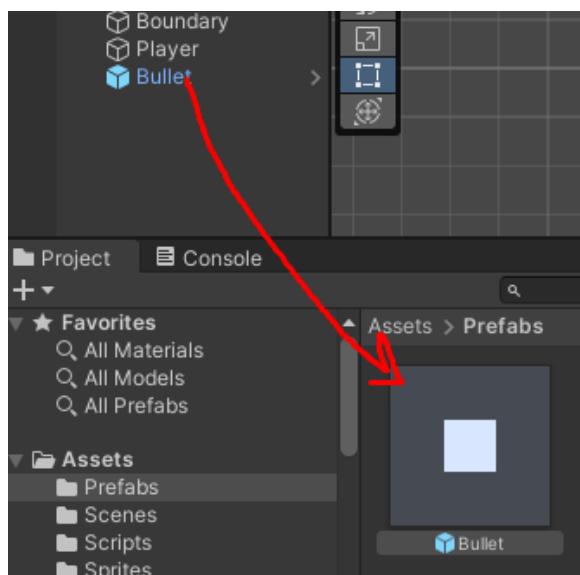
Добавим его на нашу пулю:



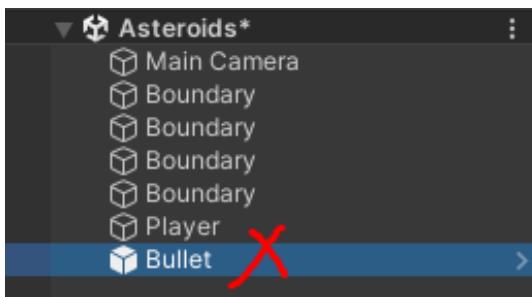
Далее нам нужно превратить пулю в префаб. Для этого создаём папку **Prefs**:



Перетаскиваем наш объект пулю в папку:



Со сцены удаляем объект **Bullet**:



7. Открываем скрипт **Bullet**. И пропишем скрипт, который позволяет создавать пулю, которая будет лететь в указанном направлении и исчезнет через некоторое время, предотвращая переполнение сцены объектами.

➤ Добавляем переменные в начале скрипта:

```
[Header("Настройки пули")]
// сила, с которой выстреливает пуля
[SerializeField] private float shootForce = 500f;
//время, через которое пуля будет уничтожена
[SerializeField] private float lifeTime = 2f;
// Ссылка на компонент Rigidbody2D, отвечающий за физику
//объекта
private Rigidbody2D _rigidbody;
```

➤ Создаём метод **Awake()**, в котором находим **Rigidbody2D**:

```
private void Awake()
{
    // находит компонент Rigidbody2D на объекте и сохраняем в
    //переменную _rigidbody
    _rigidbody = GetComponent<Rigidbody2D>();
}
```

➤ Реализуем метод для стрельбы **Shoot()**:

```
public void Shoot(Vector2 direction) // метод принимает
направление direction
{
    // добавляем силу к пуле, заставляя её лететь
    _rigidbody.AddForce(direction * shootForce);
    // запускаем таймер, через который пуля будет
    //уничтожена
    Destroy(gameObject, lifeTime);
}
```

Итоговый код:

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    [Header("Настройки пули")]
    [SerializeField] private float shootForce = 500f;
    [SerializeField] private float lifeTime = 2f;
    private Rigidbody2D _rigidbody;

    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
    }

    public void Shoot(Vector2 direction)
    {
        _rigidbody.AddForce(direction * shootForce);
        Destroy(gameObject, lifeTime);
    }
}
```

Переходим в скрипт игрока **Player** и немного обновим наш код.

Мы должны получать ссылку на нашу пушку, чтобы создавать её из игрока.

Поэтому в скрипте объявим публичную переменную, для будущего префаба:

```
public class Player : MonoBehaviour
{
    [Header("Speed")]
    [SerializeField] private float thrustSpeed = 3f;
    [SerializeField] private float turnSpeed = 0.1f;

    private bool _thrusting;
    private float _turnDirection;
    private Rigidbody2D _rigidbody;

    public Bullet bulletPrefab; // ссылка на префаб пули
```

Добавим в конце новый метод **ShootBullet()**:

```
private void ShootBullet() // метод для стрельбы
{
    Bullet bullet = Instantiate(bulletPrefab,
transform.position, transform.rotation); // создаём пушку
(объект, место, вращение)
    bullet.Shoot(transform.up); // запускаем пушку вперёд
}
```

- **Создание пули:**
 - Используется метод **Instantiate**, чтобы создать новую пулю на основе префаба **bulletPrefab**.
 - Пуля создается в той же позиции и с тем же направлением, что и объект игрока.
- **Запуск пули:**
 - После создания, метод **ShootBullet** пули вызывается для её запуска.
 - В метод **ShootBullet** передается направление **transform.up**, которое соответствует направлению, куда "смотрит" объект игрока.

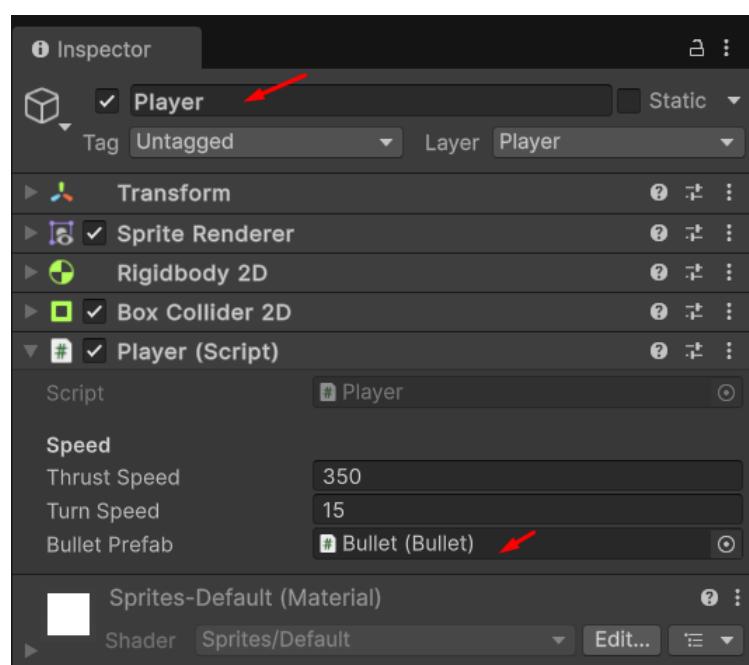
В методе **HandleInput()** вызовем наш метод **ShootBullet()**:

```
private void HandleInput()
{
    _thrusting = Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow);
    if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
        _turnDirection = 1.0f;
    else if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))
        _turnDirection = -1.0f;
    else
        _turnDirection = 0.0f;

    if (Input.GetKeyDown(KeyCode.Space) || Input.GetMouseButtonDown(0)) // добавляем стрельбу на нажатие пробела или ЛКМ
        ShootBullet(); // вызываем метод стрельбы
}
```

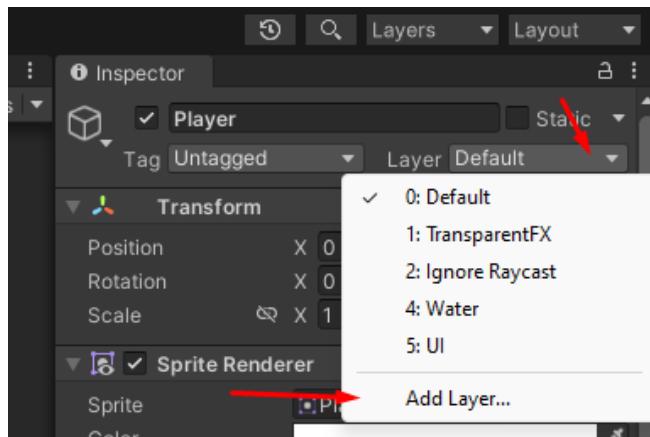
- **Создание экземпляра пули:** Каждый раз, когда игрок нажимает пробел или левую кнопку мыши (**Input.GetKeyDown(KeyCode.Space)** или **Input.GetMouseButtonDown(0)** в методе **Update**), метод **ShootBullet** вызывается.
- **Направление пули:** Пуля направляется вперёд относительно текущего направления игрока (**transform.up**), что позволяет игроку стрелять в том направлении, в котором он движется или поворачивается.

Добавляем в скрипт **Player** у объекта **Player** наш префаб:

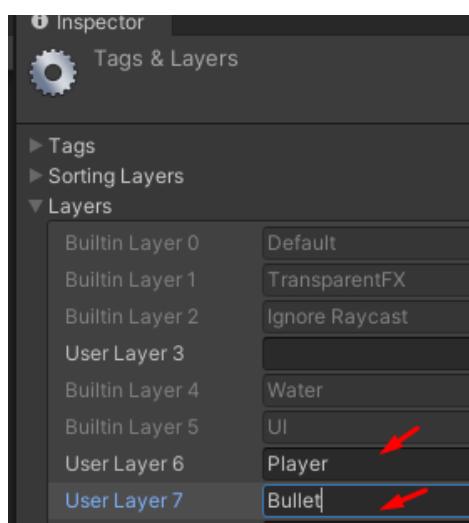


Если мы запустим игру и попробуем пострелять, то заметим, что игрок стал вращаться при стрельбе. Это связано с тем, что пули сталкиваются с игроком. Давайте изменим их поведение.

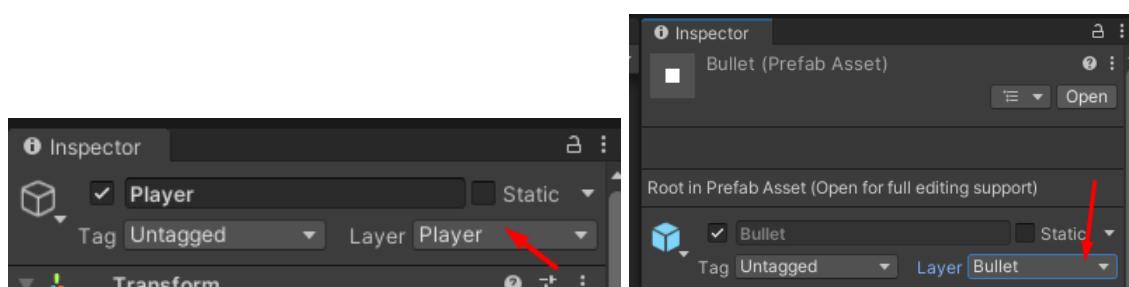
Выбираем любой объект, например **Player**, и во вкладке слоёв **Layer** нажмём добавить слой **Add Layer**:



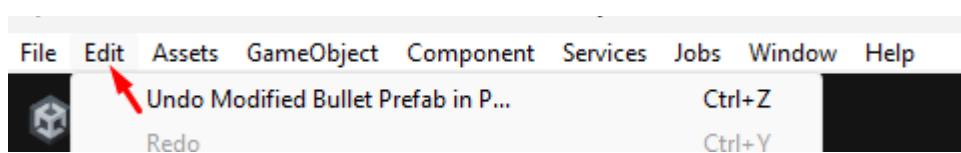
Создадим два новых слоя **Player** и **Bullet**:

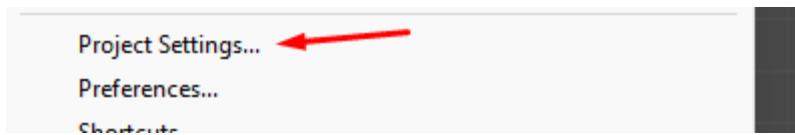


И затем выберем для нашего игрока слой **Player** и для префаба пуль **Bullet**:

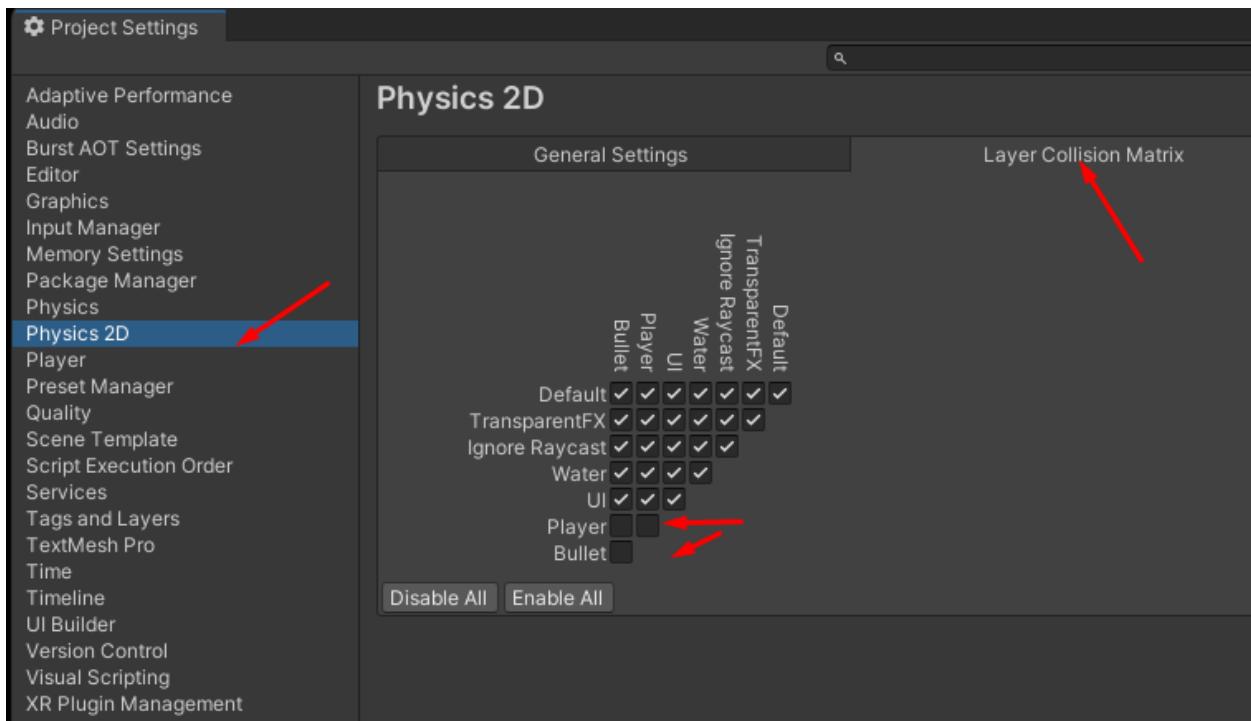


Затем перейдём в настройки **Edit – Project Settings**:





Переходим во вкладку **Physics 2D – Layer Collision Matrix** и снимаем галочки пересечения игрока и пуль:



Layer Collision Matrix в Unity позволяет настроить, какие объекты могут сталкиваться друг с другом на основе их принадлежности к различным слоям. Это полезно для оптимизации физики и улучшения производительности игры.

8. Сразу проведём небольшой рефакторинг. Во-первых, мы хотим добавить проверку на **null**. Если в инспекторе не установлен префаб пули, метод **ShootBullet()** вызовет ошибку. **Решение** – перед созданием пули проверять, установлен ли префаб.

```
Ссылка: 1
private void ShootBullet()
{
    if (bulletPrefab == null) // проверка на null
    {
        Debug.LogWarning("Не добавлен префаб пули!"); // выводим предупреждение в консоли
        return;
    }
    Bullet bullet = Instantiate(bulletPrefab, transform.position, transform.rotation);
    bullet.Shoot(transform.up);
}
```

Теперь, если префаб не установлен, игра не вылетит.

Во-вторых, если мы начнём стрелять, нажимая быстро сразу ЛКМ и Space, то сейчас игрок может **спамить выстрелами бесконечно быстро**. **Решение** – добавить **переменную shootCooldown**, которая задаёт задержку между выстрелами (создаём новые переменные):

```
[Header("Shoot")]
// Время между выстрелами
[SerializeField] private float fireCooldown = 0.2f;
// Когда последний раз стреляли
private float lastFireTime;
```

И добавит в методе **HandleInput()** проверку:

```
// Проверяем, прошло ли достаточно времени после последнего выстрела
if (Time.time - lastFireTime >= fireCooldown)
{
    if (Input.GetKeyDown(KeyCode.Space) || Input.GetMouseButtonUp(0))
    {
        ShootBullet();
        lastFireTime = Time.time; // Обновляем время последнего выстрела
    }
}
```

Итоговый код с правками для скрипта Player:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    [Header("Speed")]
    [SerializeField] private float thrustSpeed = 3f;
    [SerializeField] private float turnSpeed = 0.1f;

    private bool _thrusting;
    private float _turnDirection;
    private Rigidbody2D _rigidbody;
    public Bullet bulletPrefab;

    [Header("Shoot")]
    [SerializeField] private float fireCooldown = 0.2f;
    private float lastFireTime;

    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
    }
    private void Update()
    {
        HandleInput();
    }
    private void FixedUpdate()
    {
        ApplyMovement();
    }
}
```

```
private void HandleInput()
{
    _thrusting = Input.GetKey(KeyCode.W) ||
Input.GetKey(KeyCode.UpArrow);
    if (Input.GetKey(KeyCode.A) ||
Input.GetKey(KeyCode.LeftArrow))
        _turnDirection = 1.0f;
    else if (Input.GetKey(KeyCode.D) ||
Input.GetKey(KeyCode.RightArrow))
        _turnDirection = -1.0f;
    else
        _turnDirection = 0.0f;

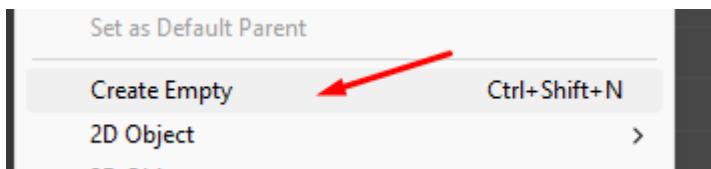
    if (Time.time - lastFireTime >= fireCooldown)
    {
        if (Input.GetKeyDown(KeyCode.Space) ||
Input.GetMouseButton(0))
        {
            ShootBullet();
            lastFireTime = Time.time;
        }
    }
}

private void ApplyMovement()
{
    if (_thrusting)
        _rigidbody.AddForce(transform.up * thrustSpeed);
    if (_turnDirection != 0.0f)
        _rigidbody.AddTorque(_turnDirection * turnSpeed);
}
private void ShootBullet()
{
    if (bulletPrefab == null)
    {
        Debug.LogWarning("Не добавлен префаб пули!");
        return;
    }
    Bullet bullet = Instantiate(bulletPrefab,
transform.position, transform.rotation);
    bullet.Shoot(transform.up);
}
```

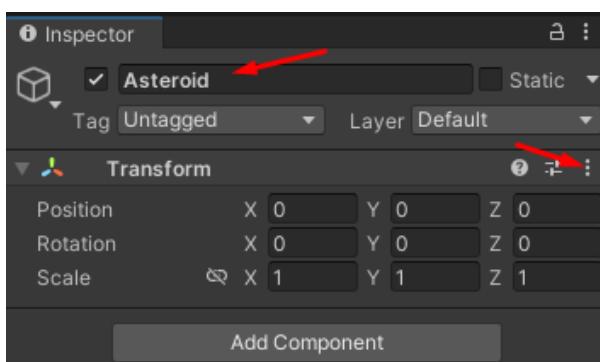
9. Теперь сделаем так, чтобы пули уничтожались при столкновении с объектами, в том числе и со стенами. В скрипте **Bullet** допишем в самый конец:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject); // Уничтожаем пушку при столкновении
}
```

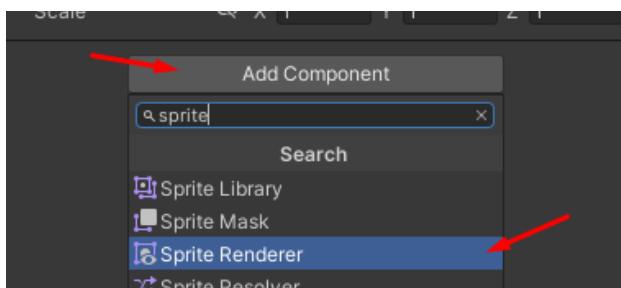
10. Перейдём к созданию **астероидов**. В иерархии создаём новый объект – **Creaty Empty**:



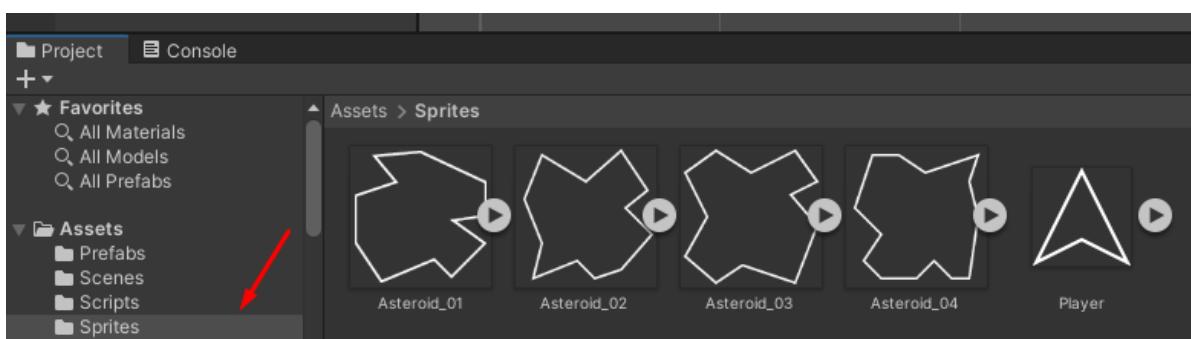
Называем его **Asteroid** и сбрасываем трансформацию:



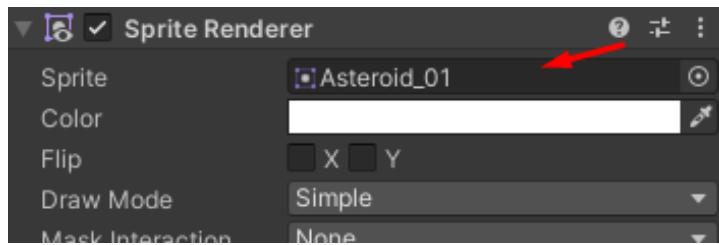
Добавляем ему компонент **Sprite Renderer**:



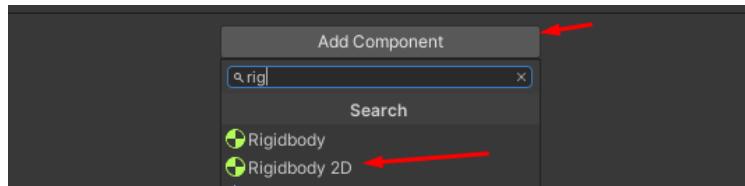
Далее в наших **ассетах** в папку **Sprites** подгрузим **4 спрайта** астероидов из **архива**:



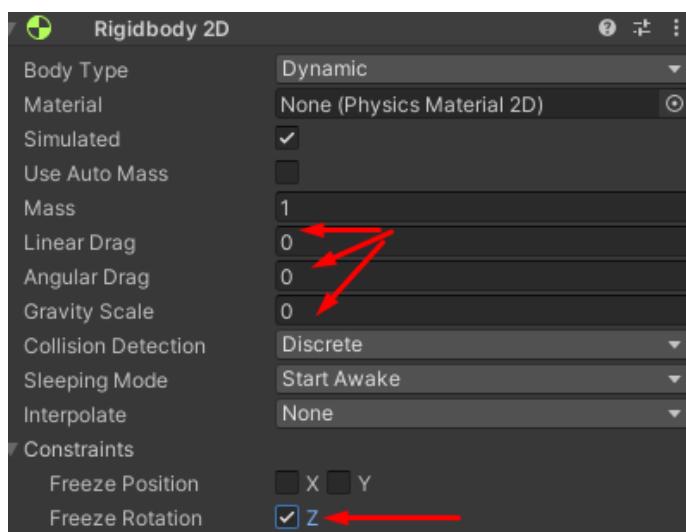
Добавим самый первый астероид в **Sprite** (далее мы зададим рандомный выбор):



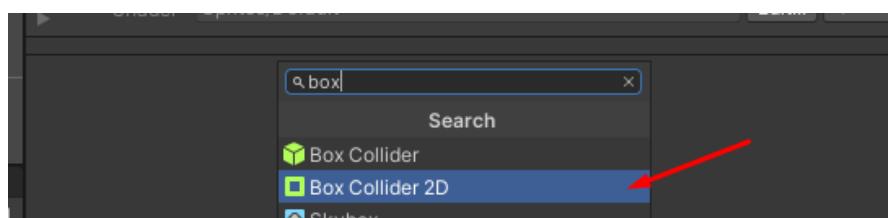
Добавляем компонент **Rigidbody 2D**:



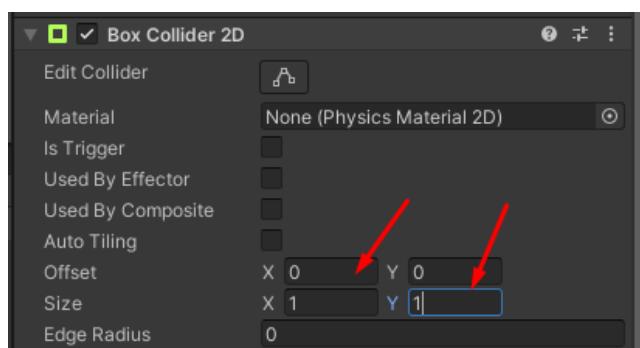
Нам не нужны такие параметры как **гравитация** и **Angular Damping** (отвечает за сопротивление), поэтому убираем их в **0**, также уберём с вами вращение по оси **Z**, «заморозим» её, для этого раскройте **Contains – Freeze Rotation**:



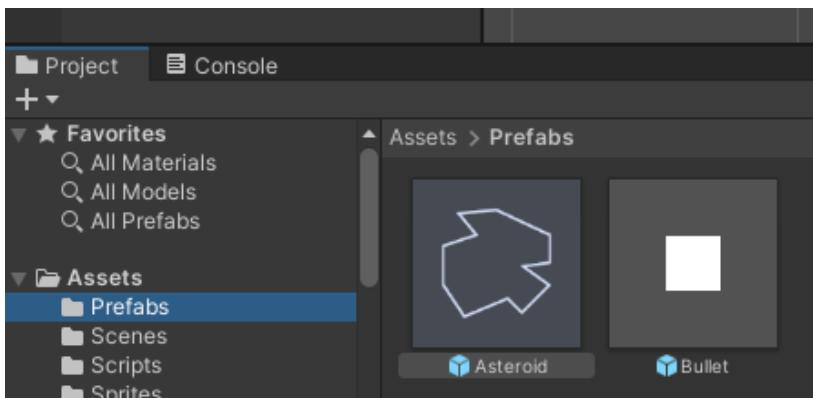
Добавляем компонент **Box Collider 2D**



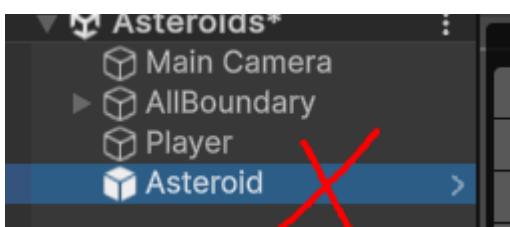
Размер поменяем на **1** по **X** и **1** по **Y**:



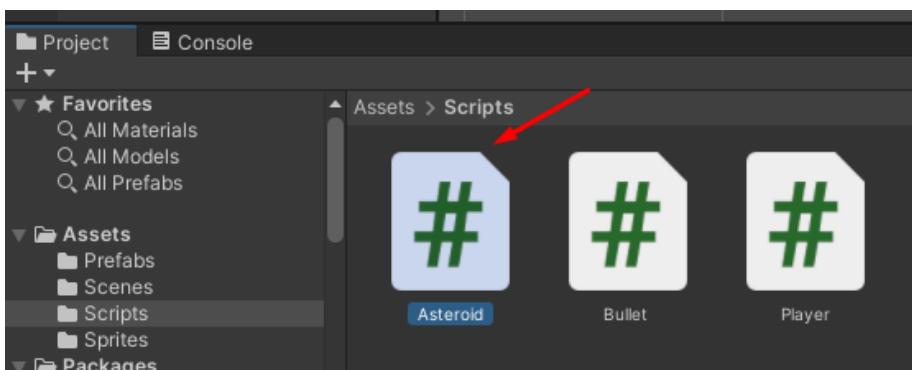
Добавляем наш астероид в папку с префабами:



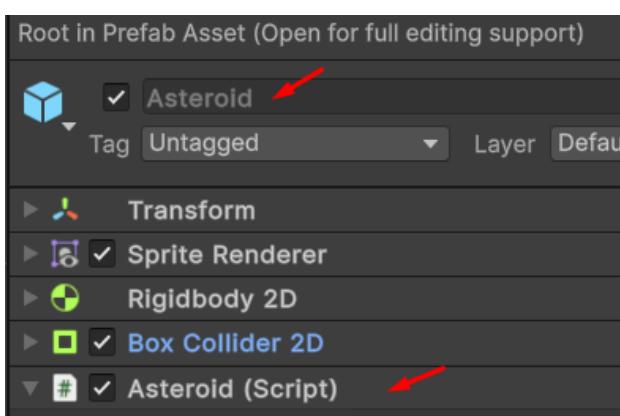
И удаляем его из объектов:



10. Создаём скрипт Asteroid:



Добавляем его на наш префаб:



Открываем скрипт **Asteroid**. Нам нужно создать рандомное появление астероидов.

- 1. В начале кода мы объявляем **переменные**, разделяя их по смыслу с помощью [**Header**], чтобы в инспекторе Unity они были организованы:

```
[Header("Настройки астероида")]
[SerializeField] private Sprite[] sprites; // Массив спрайтов
для астероида
public float size = 1f; // Размер астероида (по умолчанию 1)
public float minSize = 0.35f; // Минимальный размер астероида
public float maxSize = 1.65f; // Максимальный размер астероида

[Header("Движение астероида")]
[SerializeField] private float movementSpeed = 50f; // Скорость
движения астероида
[SerializeField] private float maxLifetime = 20f; //
Максимальное время жизни астероида

private SpriteRenderer _spriteRenderer; // Ссылка на компонент
SpriteRenderer
private Rigidbody2D _rigidbody; // Ссылка на компонент
Rigidbody2D
```

➤ 2. Метод Awake()

Далее нам нужно получить ссылки на компоненты **SpriteRenderer** и **Rigidbody2D** при создании объекта. Это важно, чтобы мы могли управлять внешним видом и физикой астероида. Поэтому пропишем метод **Awake()**:

```
private void Awake()
{
    _spriteRenderer = GetComponent<SpriteRenderer>();
    _rigidbody = GetComponent<Rigidbody2D>();
}
```

➤ 3. Метод Start()

1. Выбираем случайный спрайт

- Если в массиве есть текстуры, берём случайную.
- Если массив пуст – выводим предупреждение в консоли.

2. Устанавливаем случайное вращение

- *Random.value * 360f* – это случайный угол от 0 до 360 градусов.

3. Изменяем размер

- *Масштаб* объекта меняется в соответствии с *size*.

4. Настраиваем массу

- *Масса* равна *размеру (size)*, чтобы маленькие астероиды имели меньшую массу, а большие — большую.

```

private void Start()
{
    // Проверяем, есть ли спрайты в массиве, чтобы избежать
    ошибки
    if (sprites != null && sprites.Length > 0)
        _spriteRenderer.sprite = sprites[Random.Range(0,
    sprites.Length)];
    else
        Debug.LogWarning("Массив спрайтов для астероида
пуст!");
}

transform.eulerAngles = new Vector3(0f, 0f,
Random.value * 360f);
transform.localScale = Vector3.one * size;
_rigidbody.mass = size;
}

```

➤ 4. Метод SetTrajectory()

```

public void SetTrajectory(Vector2 direction)
{
    _rigidbody.AddForce(direction * movementSpeed);
    Destroy(gameObject, maxLifetime);
}

```

📌 Что делает этот метод?

- Получает вектор направления *direction*, в котором астероид должен двигаться.
- Использует *Rigidbody2D.AddForce()*, чтобы придать астероиду импульс в этом направлении.
- Через *maxLifetime* секунд астероид автоматически уничтожается (*Destroy(gameObject, maxLifetime);*).

◆ Разбор кода:

1. *direction * movementSpeed* – вычисляет силу, с которой нужно толкнуть астероид.
2. *Destroy(gameObject, maxLifetime);* – уничтожает астероид через *maxLifetime* секунд, чтобы не перегружать память.

Итоговый код:

```
using UnityEngine;

public class Asteroid : MonoBehaviour
{
    [Header("Настройки астероида")]
    [SerializeField] private Sprite[] sprites;
    public float size = 1f;
    public float minSize = 0.35f;
    public float maxSize = 1.65f;

    [Header("Движение астероида")]
    [SerializeField] private float movementSpeed = 50f;
    [SerializeField] private float maxLifetime = 20f;

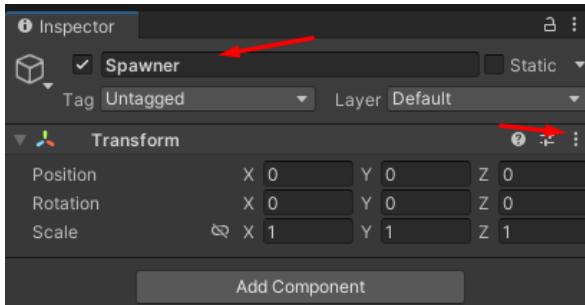
    private SpriteRenderer _spriteRenderer;
    private Rigidbody2D _rigidbody;

    private void Awake()
    {
        _spriteRenderer = GetComponent<SpriteRenderer>();
        _rigidbody = GetComponent<Rigidbody2D>();
    }

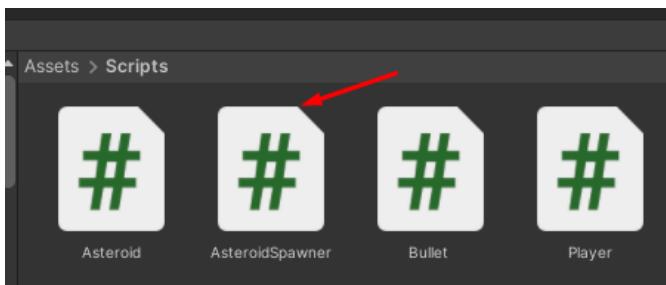
    private void Start()
    {
        if (sprites != null && sprites.Length > 0)
            _spriteRenderer.sprite = sprites[Random.Range(0,
sprites.Length)];
        else
            Debug.LogWarning("Массив спрайтов для астероида
пуст!");
        transform.eulerAngles = new Vector3(0f, 0f, Random.value * 360f);
        transform.localScale = Vector3.one * size;
        _rigidbody.mass = size;
    }

    public void SetTrajectory(Vector2 direction)
    {
        _rigidbody.AddForce(direction * movementSpeed);
        Destroy(gameObject, maxLifetime);
    }
}
```

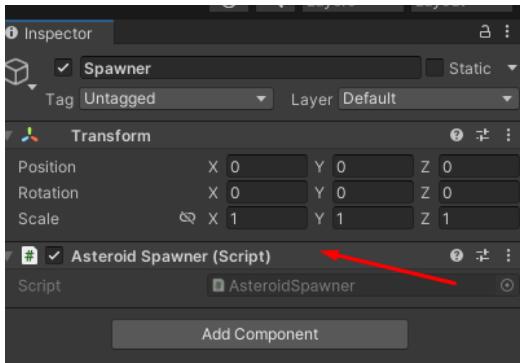
11. Теперь создадим спаун астероидов. Создаём объект Spawner и сбрасываем трансформацию:



Создаём новый скрипт AsteroidSpawner:



Добавляем его к нашему объекту Spawner:



Перейдём к написанию скрипта для AsteroidSpawner.

➤ Объявляем переменные:

```
[Header("Настройки спауна")]
[SerializeField] private Asteroid asteroidPrefab; // Префаб астероида
[SerializeField] private float spawnRate = 1f; // Частота появления астероидов
[SerializeField] private float spawnDistance = 12f; // Расстояние от центра спауна
[SerializeField] private int amountPerSpawn = 1; // Количество астероидов за один спаун

[Header("Настройки траектории")]
[SerializeField] private float trajectoryVariance = 15f; // Вариация направления движения
```

- Напишем метод **Spawn()**, который будет создавать астероиды и направлять их внутрь экрана:

```
private void Spawn()
{
    for (int i = 0; i < amountPerSpawn; i++)
    {
        Vector3 spawnDirection = Random.insideUnitCircle.normalized * spawnDistance;
        Vector3 spawnPoint = transform.position + spawnDirection;

        float variance = Random.Range(-trajectoryVariance, trajectoryVariance);
        Quaternion rotation = Quaternion.AngleAxis(variance, Vector3.forward);

        Asteroid asteroid = Instantiate(asteroidPrefab, spawnPoint, rotation);
        asteroid.size = Random.Range(asteroid.minSize, asteroid.maxSize);
        asteroid.SetTrajectory(rotation * -spawnDirection);
    }
}
```

Разберём его подробно:

◆ 1. Цикл создания астероидов

```
for (int i = 0; i < amountPerSpawn; i++)
```

Цикл *for* создаёт нужное количество астероидов (*amountPerSpawn*).

◆ 2. Определяем точку спауна

```
Vector3 spawnDirection = Random.insideUnitCircle.normalized * spawnDistance;
Vector3 spawnPoint = transform.position + spawnDirection;
```

- *Random.insideUnitCircle* – создаёт случайную *двумерную* точку внутри круга *радиусом 1*.
- *normalized* – нормализуем вектор, чтобы он имел *длину 1* (направление).
- *spawnDistance* – умножаем на *расстояние* (*spawnDistance*), чтобы объект появился на определённой дистанции.
- *spawnPoint = transform.position + spawnDirection* – точка спауна будет на этом расстоянии от центра (*transform.position*).

◆ 3. Добавляем отклонение направления

```
float variance = Random.Range(-trajectoryVariance, trajectoryVariance);
Quaternion rotation = Quaternion.AngleAxis(variance, Vector3.forward);
```

- *variance* – случайное значение от *-trajectoryVariance* до *+trajectoryVariance*.

- `Quaternion.AngleAxis(variance, Vector3.forward)` – создаём вращение (квaternion) на variance градусов вокруг оси Z.

- ◆ 4. Создаём астероид

```
Asteroid asteroid = Instantiate(asteroidPrefab, spawnPoint, rotation);
```

- `Instantiate(asteroidPrefab, spawnPoint, rotation)` – создаёт копию префаба в указанной позиции и с нужным вращением.

- `Asteroid asteroid` – получаем ссылку на созданный объект, чтобы настроить его параметры.

- ◆ 5. Устанавливаем размер астероида

```
asteroid.size = Random.Range(asteroid.minSize, asteroid.maxSize);
```

- ◆ 6. Задаём траекторию движения

```
asteroid.SetTrajectory(rotation * -spawnDirection);
```

- `rotation * -spawnDirection` –
 - Берём `spawnDirection`, но инвертируем (`-spawnDirection`), чтобы астероид двигался к центру.
 - Умножаем на `rotation`, чтобы учесть отклонение `variance`.
- `SetTrajectory()` – передаёт окончательный вектор движения астероиду.

➤ Напишем метод `Start()`

```
private void Start()
{
    if (asteroidPrefab == null)
    {
        Debug.LogWarning("Префаб астероида не назначен в
инспекторе!");
        return;
    }
    InvokeRepeating(nameof(Spawn), spawnRate, spawnRate);
}
```

 Что происходит?

1. Проверка префаба

- Если `asteroidPrefab` не назначен в `Inspector`, выводим предупреждение и прекращаем выполнение (`return`).

2. Запуск постоянного спауна

- `InvokeRepeating(nameof(Spawn), spawnRate, spawnRate);`
- Метод `Spawn()` вызывается каждые `spawnRate` секунд.
- Используем `nameof(Spawn)`, чтобы избежать ошибок при переименовании метода.

Итоговый код:

```
using UnityEngine;
public class AsteroidSpawner : MonoBehaviour
{
    [Header("Настройки спауна")]
    [SerializeField] private Asteroid asteroidPrefab;
    [SerializeField] private float spawnRate = 1f;
    [SerializeField] private float spawnDistance = 12f;
    [SerializeField] private int amountPerSpawn = 1;

    [Header("Настройки траектории")]
    [SerializeField] private float trajectoryVariance = 15f;

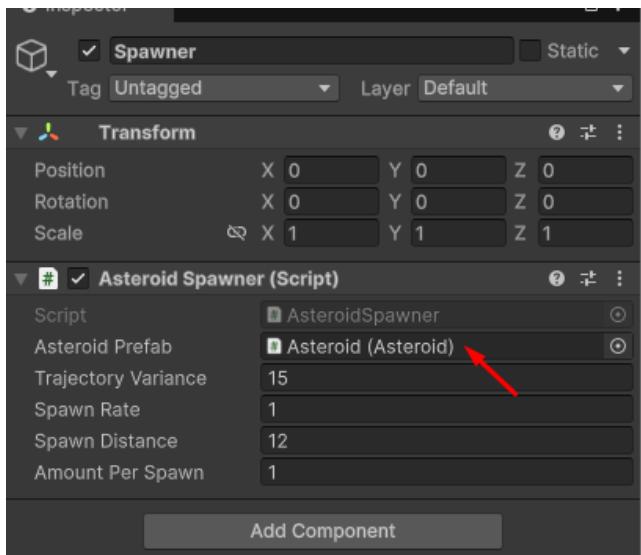
    private void Start()
    {
        if (asteroidPrefab == null)
        {
            Debug.LogWarning("Префаб астероида не назначен в
инспекторе!");
            return;
        }
        InvokeRepeating(nameof(Spawn), spawnRate, spawnRate);
    }

    private void Spawn()
    {
        for (int i = 0; i < amountPerSpawn; i++)
        {
            Vector3 spawnDirection =
Random.insideUnitCircle.normalized * spawnDistance;
            Vector3 spawnPoint = transform.position +
spawnDirection;
            float variance = Random.Range(-trajectoryVariance,
trajectoryVariance);
            Quaternion rotation = Quaternion.AngleAxis(variance,
Vector3.forward);

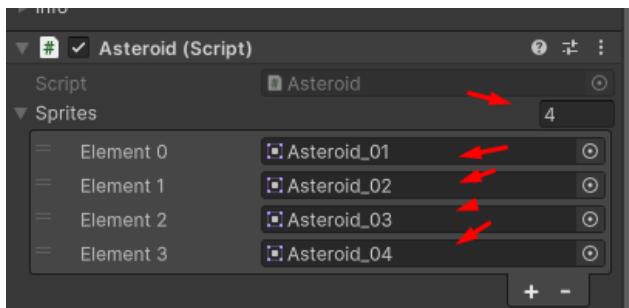
            Asteroid asteroid = Instantiate(asteroidPrefab,
spawnPoint, rotation);

            asteroid.size = Random.Range(asteroid.minSize,
asteroid.maxSize);
            asteroid.SetTrajectory(rotation * -spawnDirection);
        }
    }
}
```

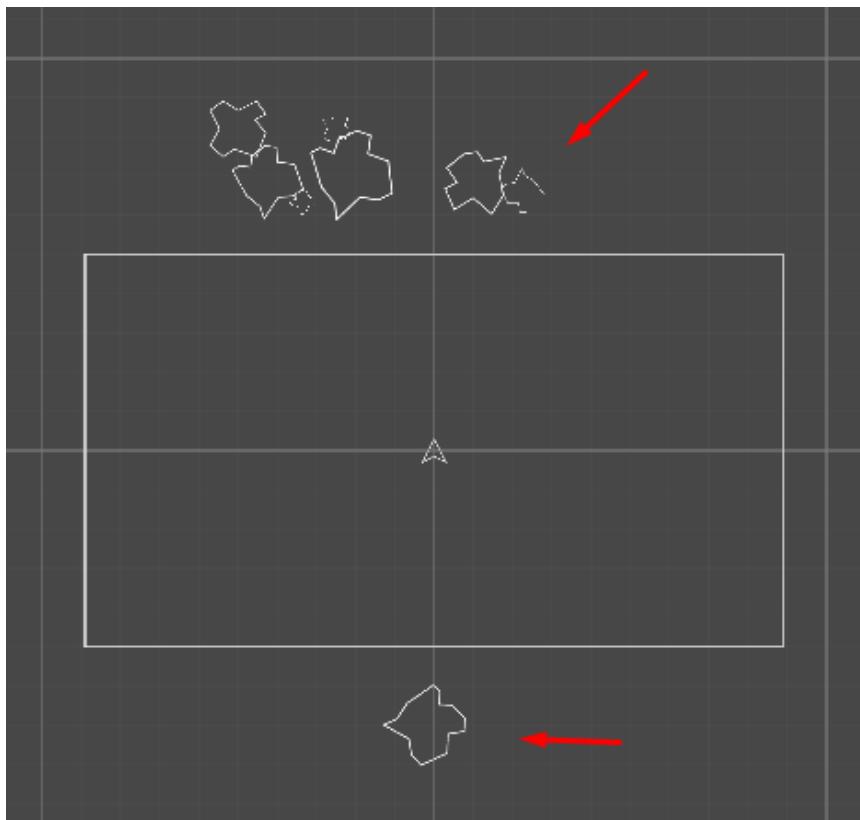
Для объекта **Spawner** добавим префаб нашего астероида:



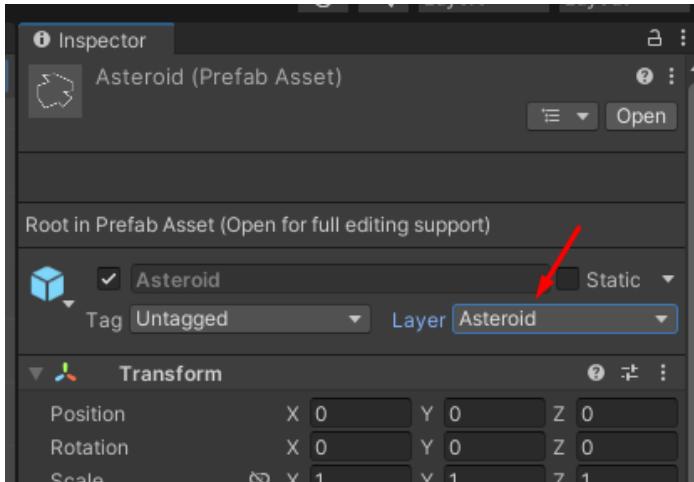
Добавим в наш скрипт у префаба астероида **4 элемента**, и выберем все спрайты:



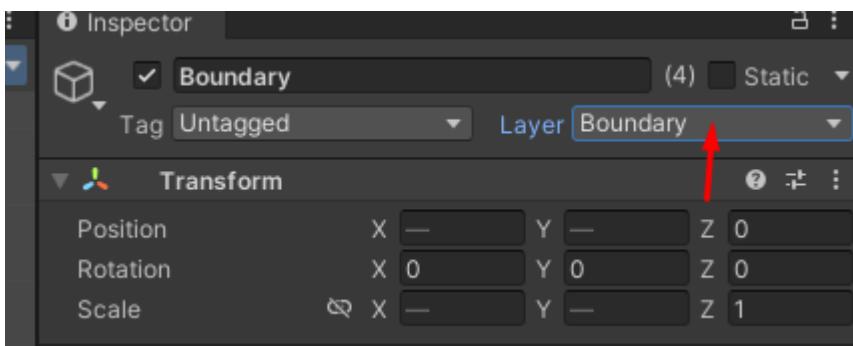
Если мы запустим сцену, то увидим, что астероиды летят и не могу залететь к нам внутрь сцены:



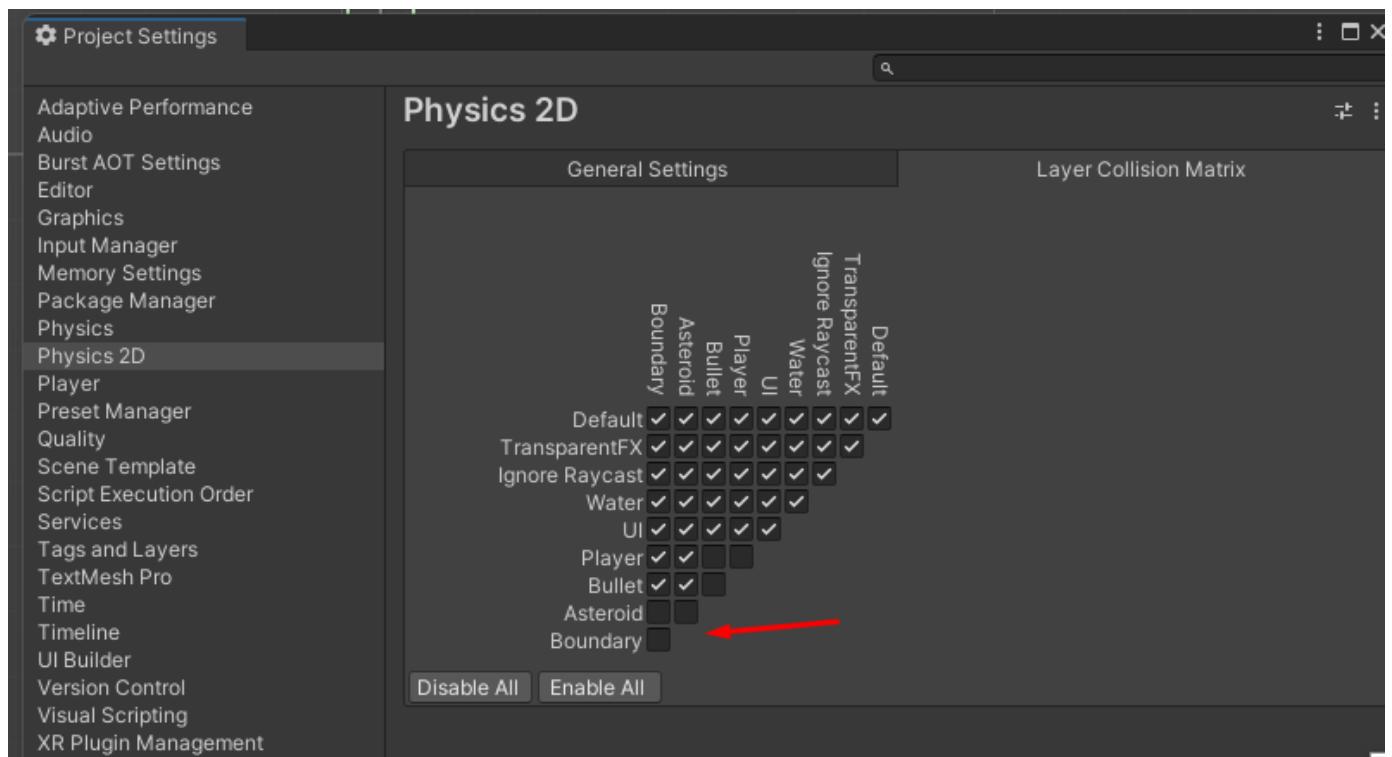
Чтобы решить эту проблему создадим новый слой **Asteroid** и добавим его к нашему префабу астероида:



Для объектов **Boundary** создаём новый слой **Boundary**:



Затем перейдём в настройки **Edit – Project Settings – Physics 2D – Layer Collision Matrix**. И настроим видимость слоёв:



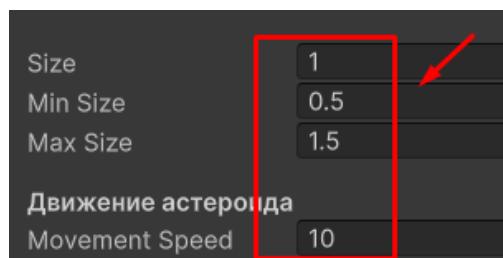
Также в скрипте **Asteroid** можем удалить инструкцию уничтожения объекта за определенное время (удаляем одну строчку кода):

```
ССЫЛКА: Asteroid.cs
public void SetTrajectory(Vector2 direction)
{
    _rigidbody.AddForce(direction * movementSpeed);
    Destroy(gameObject, maxLifetime);
}
```

Также тут же можем удалить время жизни астероида:

```
[Header("Движение астероида")]
[SerializeField] private float movementSpeed = 50f;
[SerializeField] private float maxLifetime = 20f;
```

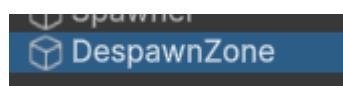
Самое время поэкспериментировать с параметрами у **префаба астероида**, чтобы изменить скорость и размер (настройте их под себя, или воспользуйтесь предлагаемыми мной параметрами):



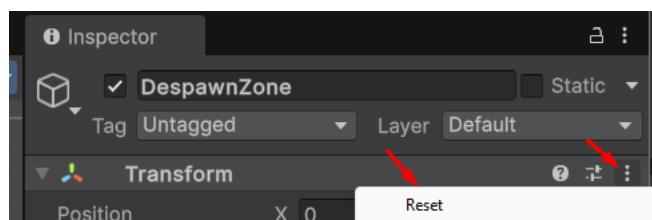
Рефакторинг

Если мы посмотрим в сцене, то увидим, что астероиды существуют достаточно долго и вылетая за пределы экрана могут ещё длительное время куда-то лететь. Это не очень хорошо, потому что игру нужно оптимизировать сразу. Да, мы могли бы поэкспериментировать со временем жизни астероида, но гораздо лучшее решение – создать новую область по размеру чуть больше нашей камеры, и затем уничтожать астероид через X секунд после вылета за её границы.

Поэтому давайте создадим пустой объект и назовём его **DespawnZone**:



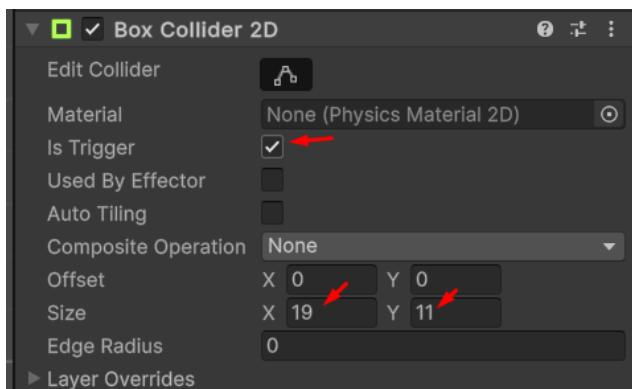
На забудем сбросить трансформацию:



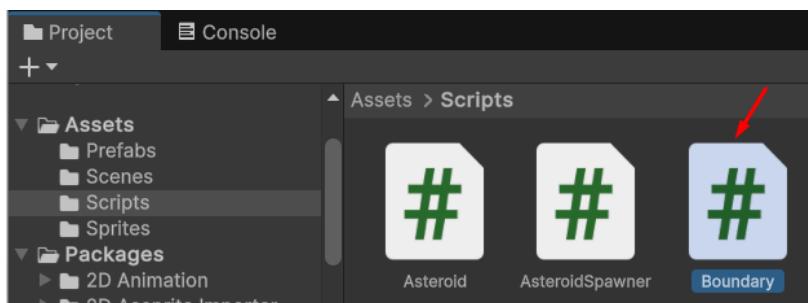
Перенесём внутрь **объекта с границами**:



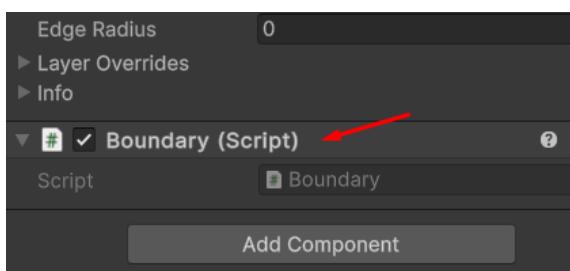
Добавим компонент **Box Collider 2D** и настроим размеры, выходящие за пределы экрана (в моём случае **x = 19, y = 11**) и на забудем поставить галочку **Is Trigger** для регистрации столкновений:



Далее создадим скрипт **Boundary**:



И прикрепим его к нашему объекту **DespawnZone**:



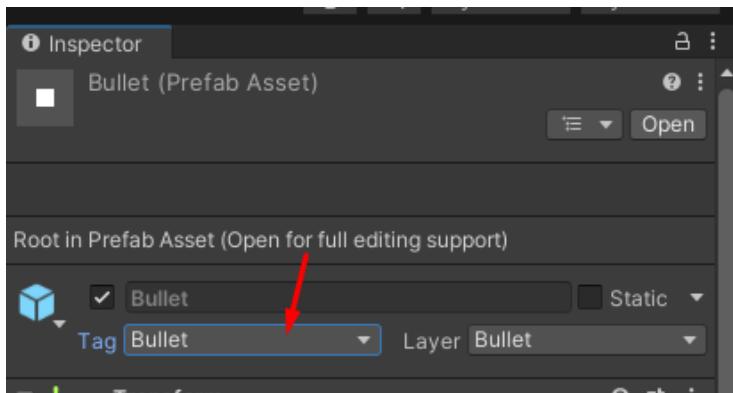
Открываем скрипт и пропишем код:

```
private void OnTriggerExit2D(Collider2D other)
{
    Destroy(other.gameObject); // Уничтожаем объект, если он
    // ВЫХОДИТ за границу
}
```

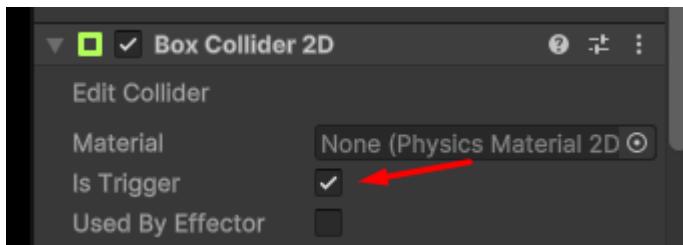
12. Сделаем возможность уничтожать астероиды.

1 способ.

Для начала добавим для префаба **Bullet** – тег **Bullet**:



Далее у нашего префаба **Bullet**, поставим галочку **Is Trigger**:



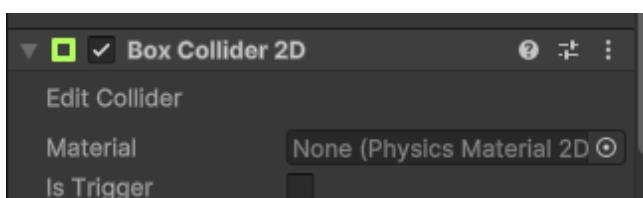
Наша пуля будет регистрировать столкновения с астероидами, но не будет оказывать физического воздействия на них (не будет сталкиваться или отталкиваться).

Затем в методе **Asteroid** добавим новый метод:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Bullet"))
    {
        Destroy(other.gameObject); // уничтожаем пулю
        Destroy(gameObject); // уничтожаем астероид
    }
}
```

2 способ, учитывающий физику объектов.

Убираем у нашего префаба **Bullet** галочку **Is Trigger**:



Изменяем код:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Bullet"))
    {
        Destroy(gameObject);
    }
}
```

Что лучше использовать?

OnCollisionEnter2D

1. Физические столкновения:

- Этот метод используется для обработки столкновений, когда два объекта физически сталкиваются друг с другом.
- Оба объекта должны иметь компоненты Collider2D, и хотя бы один из них должен иметь компонент Rigidbody2D.

2. Физическое воздействие:

- OnCollisionEnter2D применяется, когда важно учитывать физические свойства столкновения, такие как масса, импульс и результат столкновения (например, отскакивание).

OnTriggerEnter2D

1. Обработка триггеров:

- Этот метод используется для обработки пересечений коллайдеров, когда хотя бы один из них настроен как Trigger.
- Объекты не будут физически взаимодействовать (не будут сталкиваться или отталкиваться), но будут вызывать события триггера.

2. Нет физического воздействия:

- OnTriggerEnter2D применяется, когда важно только зафиксировать факт пересечения коллайдеров, а не физическое столкновение.

13. Добавим возможность разбивать большие астероиды, на два более мелких.

Добавляем внутрь скрипта **Asteroid** новый метод **CreateSplit**:

```
private Asteroid CreateSplit()
{
    Vector2 position = transform.position;
    position += Random.insideUnitCircle * 0.5f;
    Asteroid half = Instantiate(this, position,
        transform.rotation);
    half.size = size * 0.5f;
    half.SetTrajectory(Random.insideUnitCircle.normalized);
    return half;
}
```



Шаги выполнения метода:

1. Создаём точку спауна для нового астероида

```
Vector2 position = transform.position;
position += Random.insideUnitCircle * 0.5f;
```

- Берём **текущую позицию** астероида (transform.position).
- Random.insideUnitCircle * 0.5f – **добавляем небольшое случайное смещение** (до 0.5 единицы) в пределах круга.
- Это делается, чтобы новые астероиды не появлялись точно в одной точке (избегаем наложения).

2. Создаём новый астероид

```
Asteroid half = Instantiate(this, position, transform.rotation);
```

- Instantiate(this, position, transform.rotation);
 - Создаёт **копию текущего астероида** (this) в новой позиции (position).
 - **Берёт его текущее вращение (transform.rotation)**, чтобы спавн происходил без резких изменений угла.
- Asteroid half – сохраняем ссылку на новый объект, чтобы дальше менять его параметры.

3. Уменьшаем размер нового астероида

```
half.size = size * 0.5f;
```

- Новый астероид будет **в 2 раза меньше** родительского (size * 0.5f).

4. Задаём случайную траекторию движения

```
half.SetTrajectory(Random.insideUnitCircle.normalized);
```

- Random.insideUnitCircle.normalized – создаёт **случайное направление движения**.
- .normalized – нормализует вектор, чтобы он имел **длину 1** (чистое направление).
- SetTrajectory() – отправляет новый астероид в случайном направлении.

5. Возвращаем созданный астероид

```
return half;
```

- Метод возвращает ссылку на новый астероид, если нам вдруг понадобится работать с ним в будущем.

В метод **OnCollisionEnter2D** добавим условие, что если текущий размер астероида больше или равен половине минимального размера (**minSize**), то вызывается метод **CreateSplit()** дважды (для создания двух новых астероидов):

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Bullet"))
    {
        if ((size * 0.5f) >= minSize)
        {
            CreateSplit();
            CreateSplit();
        }
        Destroy(gameObject);
    }
}
```

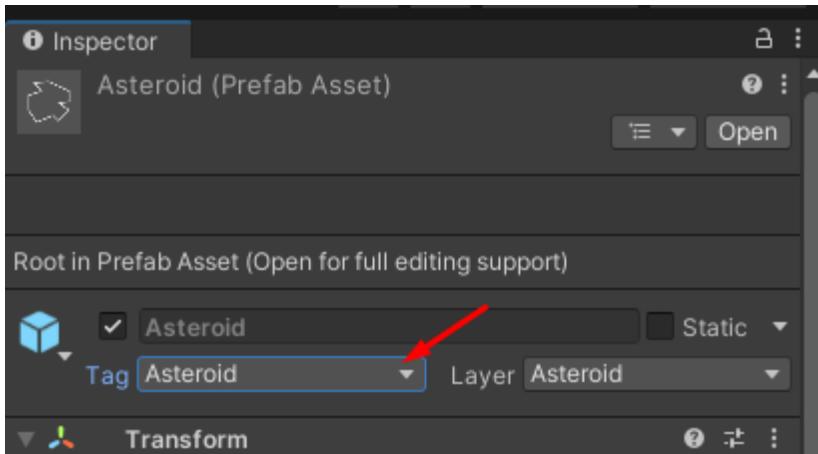
Если мы запустим и посмотрим игру, то обратим внимание, что астероиды разбиваются довольно медленно. Давайте это поправим. В скрипте **Asteroid** в методе **CreateSplit()** нашу траекторию умножим на скорость нашего астероида:

```

Ссылок: 2
private Asteroid CreateSplit()
{
    Vector2 position = transform.position;
    position += Random.insideUnitCircle * 0.5f;
    Asteroid half = Instantiate(this, position, transform.rotation);
    half.size = size * 0.5f;
    half.SetTrajectory(Random.insideUnitCircle.normalized * movementSpeed);
    return half;
}

```

14. Добавим смерть и респавн игрока. Вначале добавим новый тег для астероида:



В скрипте **Player** допишем в конце новый метод:

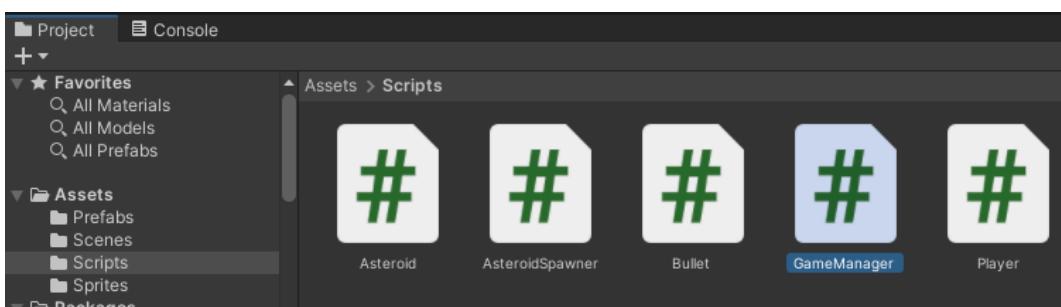
```

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Asteroid"))
    {
        _rigidbody.linearVelocity= Vector3.zero; // Обнуляем
        скорость при столкновении с астероидом
        _rigidbody.angularVelocity = 0f; // Обнуляем угловую
        скорость

        //Invoke() метод, который вызывает обработчик смерти игрока
    }
}

```

Чтобы дописать метод **Invoke()** нам нужно создать новый скрипт **GameManager**:



Открываем скрипт **GameManager**. Пропишем код.

➤ Объявляем переменные:

```
[Header("Настройки игрока")]
[SerializeField] private Player player; // Ссылка на игрока

[Header("Игровые параметры")]
[SerializeField] private float respawnTime = 3f; // Время
возрождения игрока
[SerializeField] private int lives = 3; // Количество жизней
```

➤ Создадим метод **GameOver()**, его пока оставим пустым:

```
private void GameOver()
{
    // Позже добавим логику завершения игры
    Debug.Log("Game Over!");
}
```

➤ Пропишем метод **Respawn()**:

```
private void Respawn()
{
    if (player == null)
    {
        Debug.LogError("GameManager: Игрок не найден при
респауне!");
        return;
    }
    player.transform.position = Vector3.zero; // Перемещаем игрока
    в центр
    player.gameObject.SetActive(true); // Активируем объект игрока
}
```

➤ Пропишем метод **PlayerDied()**:

```
public void PlayerDied()
{
    if (player == null)
    {
        Debug.LogError("GameManager: Игрок не назначен!");
        return;
    }
    lives--; // Уменьшаем количество жизней
    if (lives <= 0)
    {
        GameOver(); // Если жизней больше нет, завершаем игру
    }
    else
    {
        Invoke(nameof(Respawn), respawnTime); // Если жизни
остались, вызываем респаун через время
    }
}
```

Итоговый код:

```
using UnityEngine;
public class GameManager : MonoBehaviour
{
    [Header("Настройки игрока")]
    [SerializeField] private Player player;

    [Header("Игровые параметры")]
    [SerializeField] private float respawnTime = 3f;
    [SerializeField] private int lives = 3;

    public void PlayerDied()
    {
        if (player == null)
        {
            Debug.LogError("GameManager: Игрок не назначен!");
            return;
        }
        lives--;
        if (lives <= 0)
        {
            GameOver();
        }
        else
        {
            Invoke(nameof(Respawn), respawnTime);
        }
    }

    private void Respawn()
    {
        if (player == null)
        {
            Debug.LogError("GameManager: Игрок не найден при респауне!");
            return;
        }
        player.transform.position = Vector3.zero;
        player.gameObject.SetActive(true);
    }

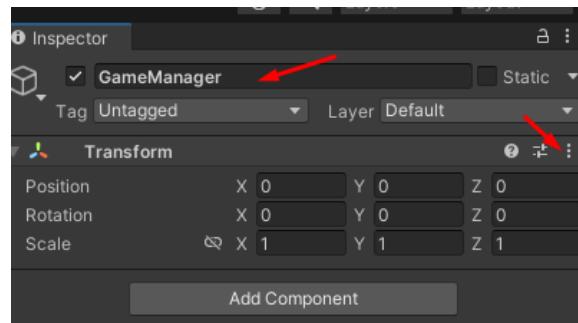
    private void GameOver()
    {
        Debug.Log("Game Over!");
    }
}
```

Допишем метод Player:

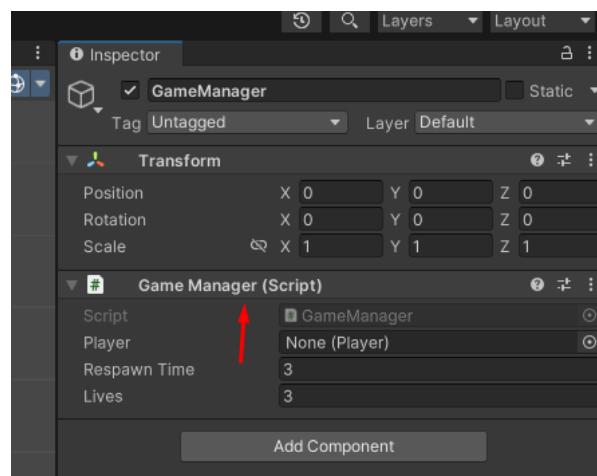
```
+ Свободные слоты в скрипте
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Asteroid"))
    {
        _rigidbody.linearVelocity = Vector3.zero; // Обнуляем
            скорость при столкновении с астероидом
        _rigidbody.angularVelocity = 0f; // Обнуляем угловую скорость

        gameObject.SetActive(false); // отключаем наш игровой объект
        FindFirstObjectOfType<GameManager>().PlayerDied(); // запускаем событие, происходящее при смерти игрока
    }
}
```

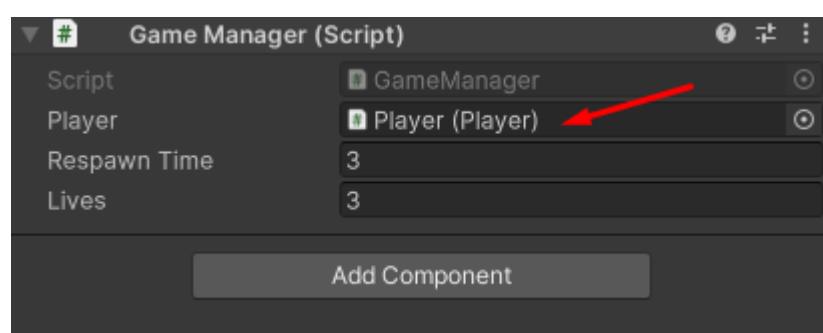
Создаём новый объект **GameManager** и сбрасываем у него трансформацию:



Добавим к нему скрипт:

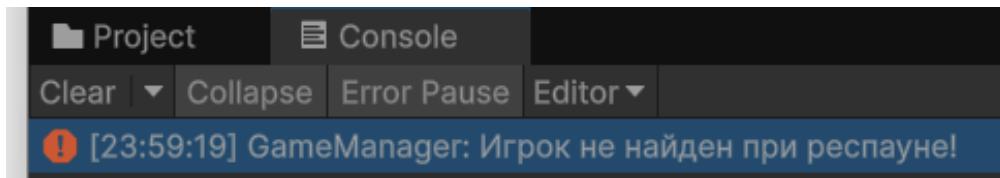


Перенесём игрока в поле **Player (объект)**:

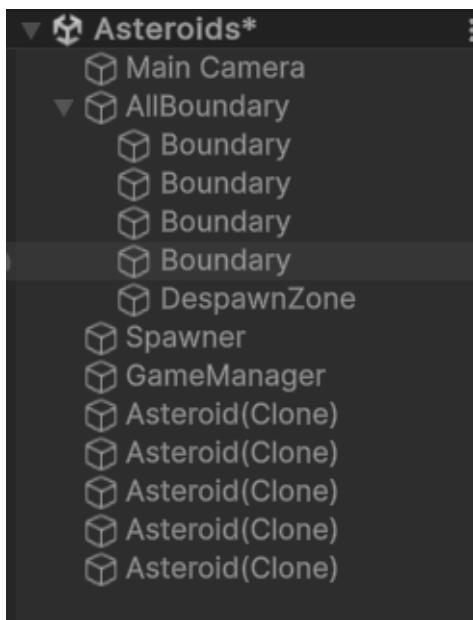


Рефакторинг с уничтожением игрока и ошибкой.

При уничтожении от астероида наш игрок не появляется, зато мы видим в консоли ошибку:



И если внимательно посмотри в иерархии, то видим что игрок пропал как объект:

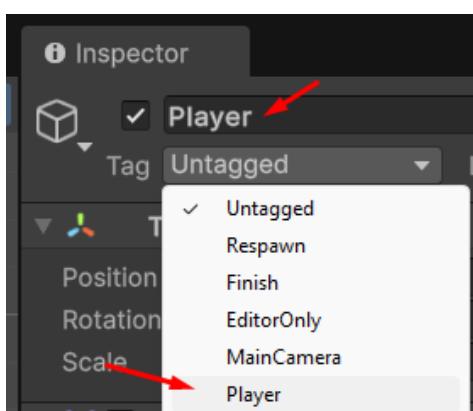


А он должен становиться **не активным** (серым)!

Всё дело в том, что игрок **уничтожается из-за зоны DespawnZone**, потому что она удаляет **все объекты, покидающие игровую зону**, включая игрока.

🔧 Исправление проблемы с DespawnZone.

Нам нужно исключить игрока из зоны уничтожения. Для этого вначале добавляем игроку **тег Player**:



Затем изменим скрипт **Boundary**:

```

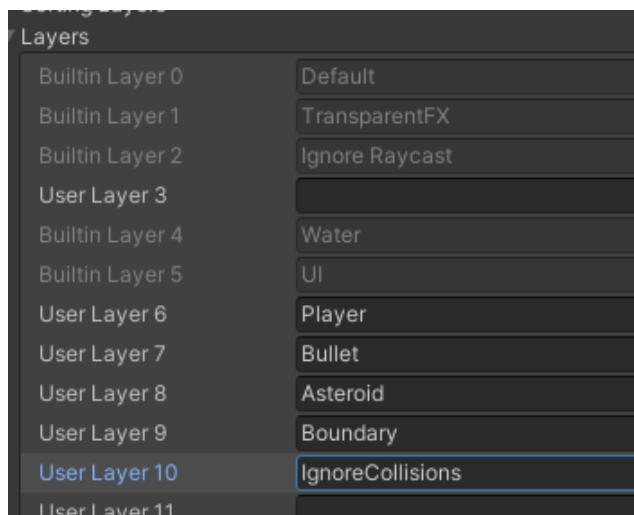
public class Boundary : MonoBehaviour
{
    // Сообщение Unity | Ссылка на ресурсы | Ссылка в
    private void OnTriggerEnter2D(Collider2D other)
    {
        // Проверяем, является ли объект игроком
        if (other.CompareTag("Player"))
        {
            Debug.Log("Игрок покинул DespawnZone, но не уничтожаем его.");
            return; // Прерываем метод, не уничтожаем игрока
        }

        // Если объект не игрок – уничтожаем
        Destroy(other.gameObject);
    }
}

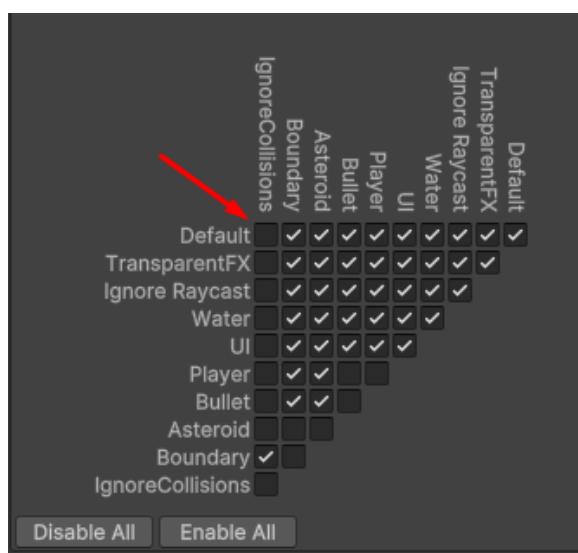
```

15. Сделаем возможность при **возрождении** игрока, чтобы он не сталкивался с астероидами некоторое время.

Создаём новый слой **IgnoreCollisions**



Затем **Edit – Project Settings - Physics 2D**. И настроим видимость слоёв:



Для границ оставим галочку, чтобы игрок не смог выходить за их пределы.

В скрипте **GameManager** нужно добавить новый метод, который будет устанавливать стандартный слой игрока:

```
private void TurnOnCollisions()
{
    player.gameObject.layer = LayerMask.NameToLayer("Player");
// добавляем ссылку на слой игрока
}
```

Затем добавим в методе **Respawn** ссылку на наш слой и вызовем его через 3 секунды:

```
private void Respawn()
{
    if (player == null)
    {
        Debug.LogError("GameManager: Игрок не найден при респауне!");
        return;
    }

    player.transform.position = Vector3.zero;
    player.gameObject.layer = LayerMask.NameToLayer("IgnoreCollisions"); // добавляем ссылку на наш новый слой
    player.gameObject.SetActive(true);
    Invoke(nameof(TurnOnCollisions), 3f); // через 3 секунды вызываем слой игрока
}
```

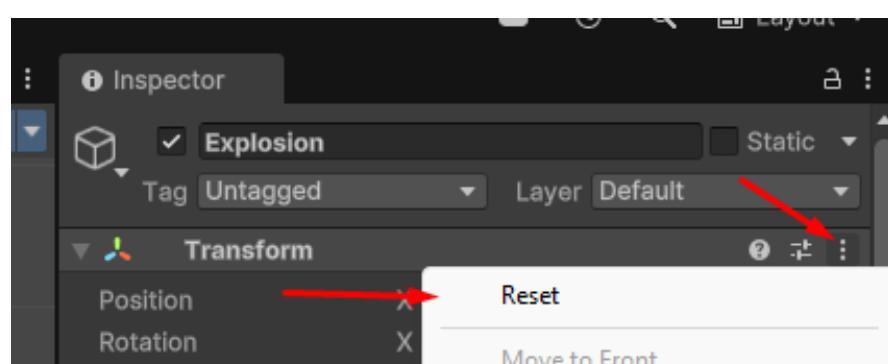
16. Добавим простой эффект частиц, который будет воспроизводится при столкновении пули с астероидами, и при смерти игрока. Создаём в **Hierarchy** систему частиц (**Effects – Particle System**)



Назовём её **Explosion**:



Сбросим трансформацию:



Duration (продолжительность) - **1**;

Looping (зацикливание) уберём;

Start Lifetime (продолжительность жизни) – **0.5**;

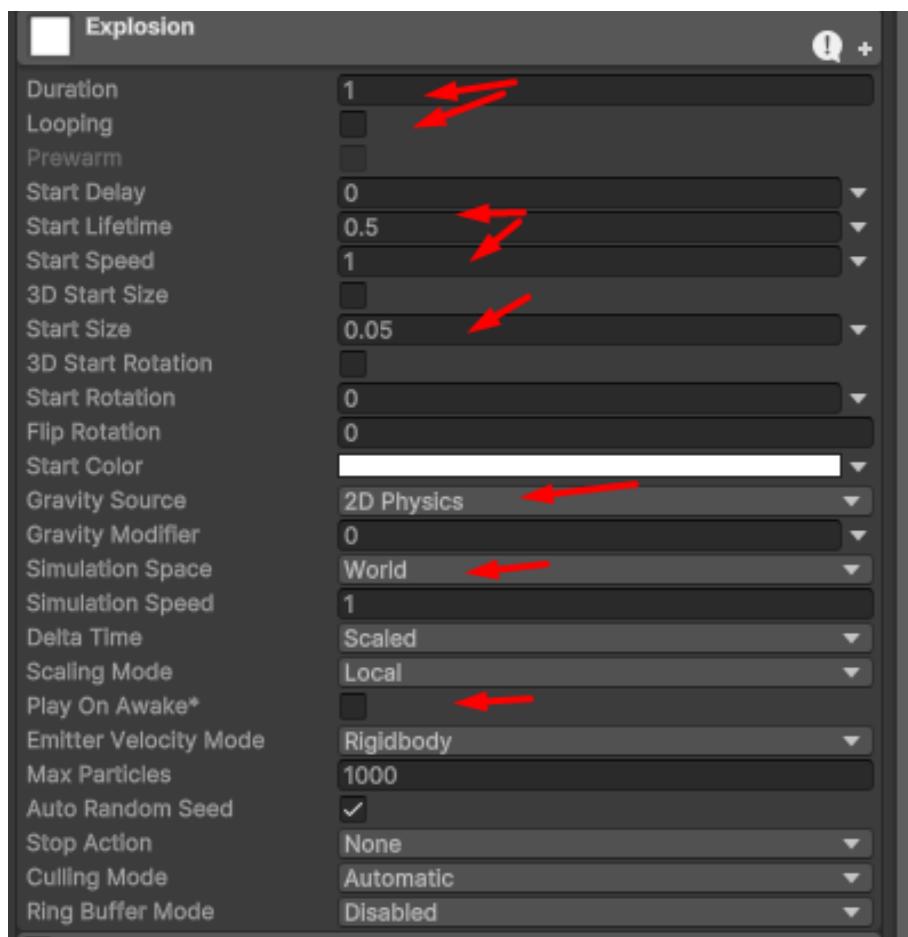
Start Speed (начальная скорость) – **1**;

Start Size (начальный размер) – **0.05**;

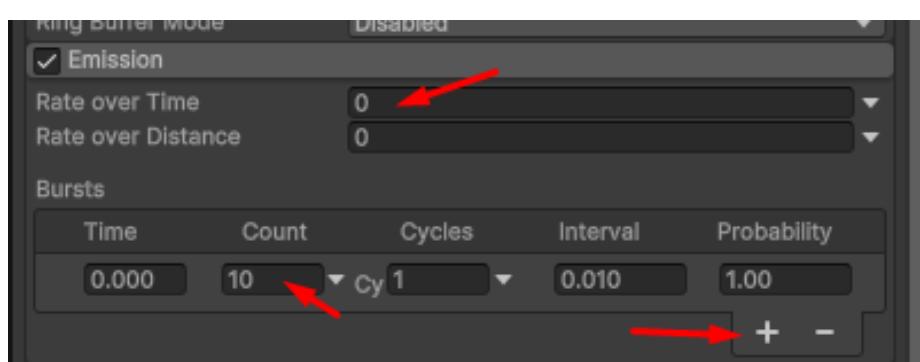
Gravity Source (источник гравитации) – **2D Physics**;

Simulation Space (моделирование пространства) – **World**;

Play On Awake – выключаем;



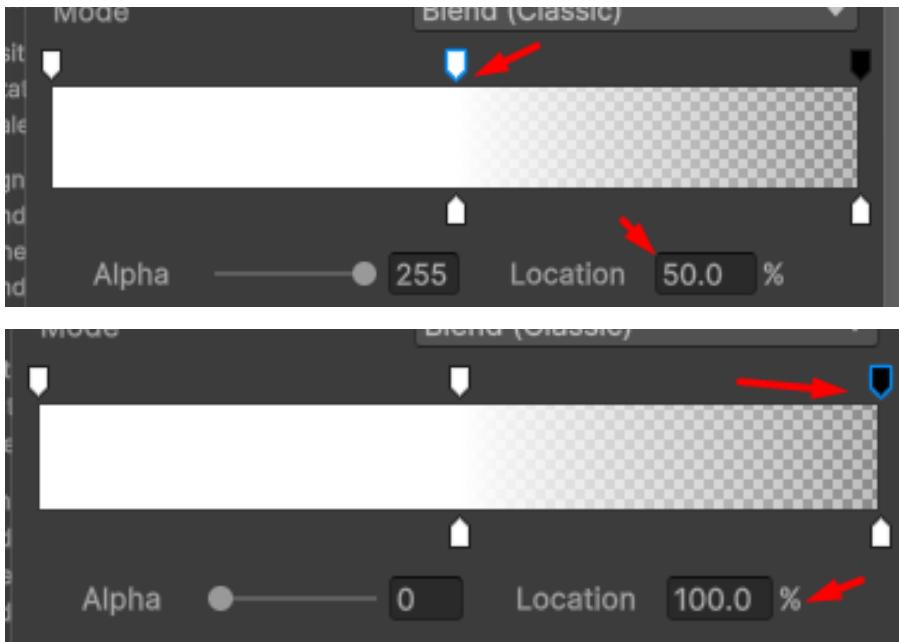
Раскроем меню **Emission** и установим **Rate over Time** на **0**, и добавим новый список, где установим количество наших частиц **10**:



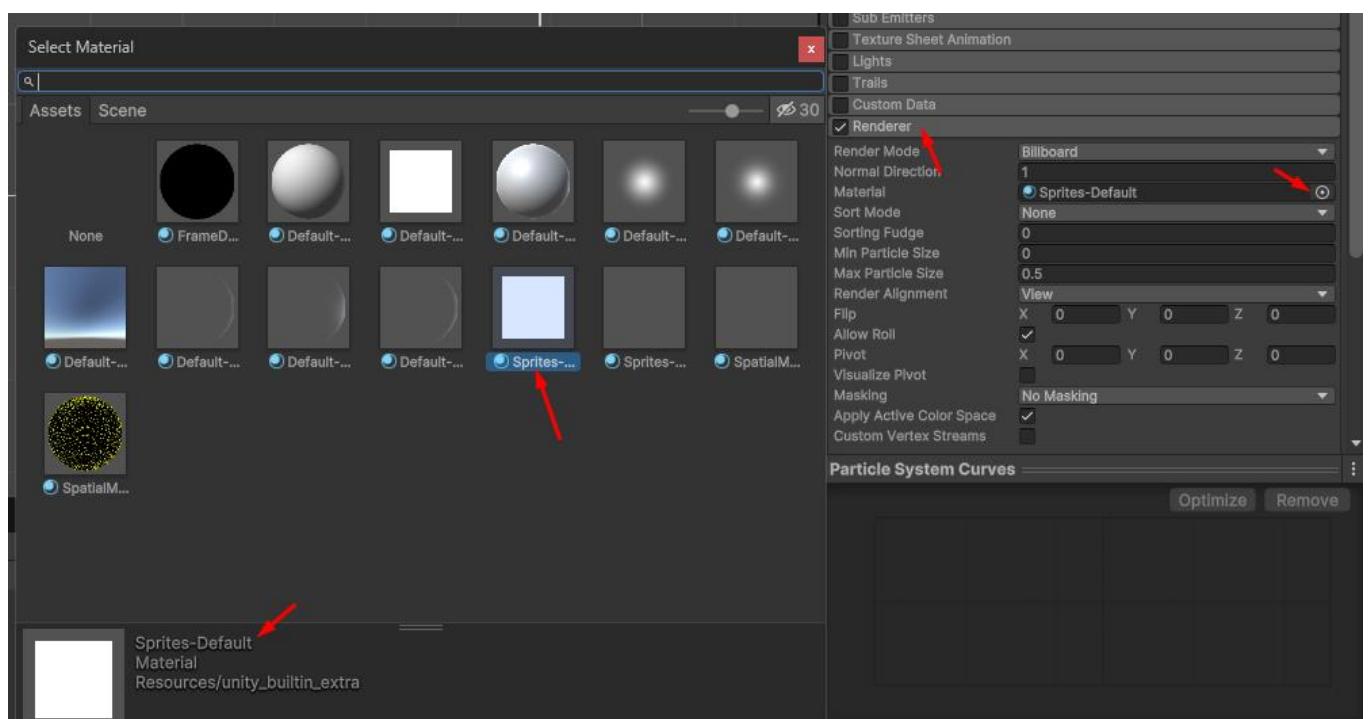
В меню **Shape** поменяю форму на **Circle** и установим минимальный радиус **0.0001**:



Поставим галочку в меню **Color over Lifetime** и настроим его так, чтобы со временем он затухал. Открываем **Color** ставим точку по середине (**Location** на 50%). Затем в конце ((**Location** на 100%)) и меняем **Alpha** на 0:



В **Render** выберите **Sprites-Default**:



Переходим в скрипт **GameManager**. Добавим **переменную** для наших частиц и заодно обновим **название** нашего атрибута:

```
[Header("Объекты")]
[SerializeField] private Player player;
[SerializeField] private ParticleSystem explosion;
```

Далее в этом же скрипте напишем метод для вызова системы частиц **AsteroidDestroyed()**:

```
public void AsteroidDestroyed(Asteroid asteroid)
{
    explosion.transform.position =
asteroid.transform.position; // устанавливаем позицию взрыва на позицию астероида
    explosion.Play(); // воспроизводим взрыв
}
```

Также добавим в начале метода **PlayerDied()** взрыв для игрока:

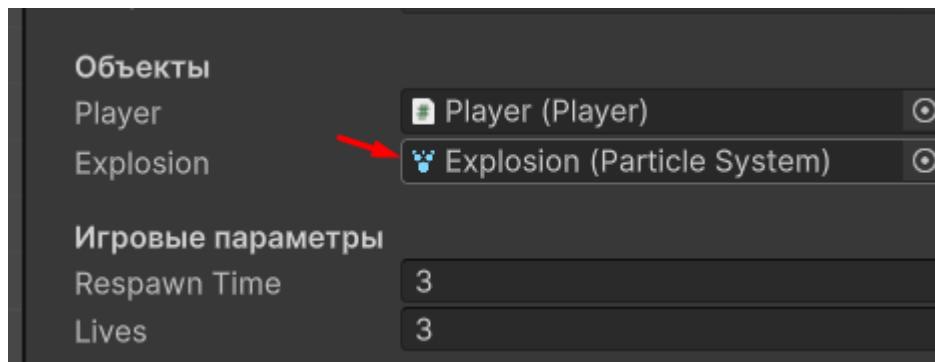
```
Ссылок: 1
public void PlayerDied()
{
    if (player == null)
    {
        Debug.LogError("GameManager: Игрок не назначен!");
        return;
    }
    // устанавливаем позицию взрыва на позицию игрока
    explosion.transform.position = player.transform.position;
    // воспроизводим взрыв
    explosion.Play();

    lives--;
}
```

В скрипте **Asteroid** обновим метод **OnCollisionEnter2D**:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Bullet"))
    {
        if ((size * 0.5f) >= minSize)
        {
            CreateSplit();
            CreateSplit();
        }
        FindFirstObjectOfType<GameManager>().AsteroidDestroyed(this); // создаём взрыв
        Destroy(gameObject);
    }
}
```

Для объекта **GameManager** нужно добавить систему частиц в поле **Explosion**:



Рефакторинг системы частиц.

При реализации нашего метода с системой частиц есть недостатки:

1. Изменение позиции существующего объекта explosion:

- мы изменяем позицию объекта **explosion**, чтобы он находился на месте астероида.
- это может привести к проблемам, если объект **explosion** используется где-то еще в сцене. Например, если взрыв уже воспроизводится в другом месте, изменение его позиции прервёт текущий взрыв и переместит его на новую позицию.

2. Отсутствие уничтожения объекта:

- объект **explosion** не уничтожается после завершения анимации. Это может привести к накоплению объектов в сцене, что негативно скажется на производительности.

3. Проблемы с многократным использованием:

- Если метод вызывается несколько раз подряд (например, при уничтожении нескольких астероидов), объект **explosion** будет "перепрыгивать" между позициями, и только последний вызов будет корректно отображать взрыв.

Поэтому гораздо лучше использовать создание нового объекта для каждого взрыва, автоматически уничтожать его и изолировать эффект. Перепишем метод:

```
public void AsteroidDestroyed(Asteroid asteroid)
{
    ParticleSystem explosionInstance =
    Instantiate(explosion, asteroid.transform.position,
    Quaternion.identity);
    explosionInstance.Play();
    Destroy(explosionInstance.gameObject,
    explosionInstance.main.duration);
}
```

Время задержки равно длительности анимации взрыва (`explosionInstance.main.duration`), что позволяет системе частиц завершить воспроизведение перед уничтожением.

Тоже самое пропишем в методе **PlayerDied()**:

```

    public void PlayerDied()
    {
        if (player == null)
        {
            Debug.LogError("GameManager: Игрок не назначен!");
            return;
        }

        ParticleSystem explosionInstance = Instantiate(explosion, player.transform.position,
            Quaternion.identity);
        explosionInstance.Play();
        Destroy(explosionInstance.gameObject, explosionInstance.main.duration);

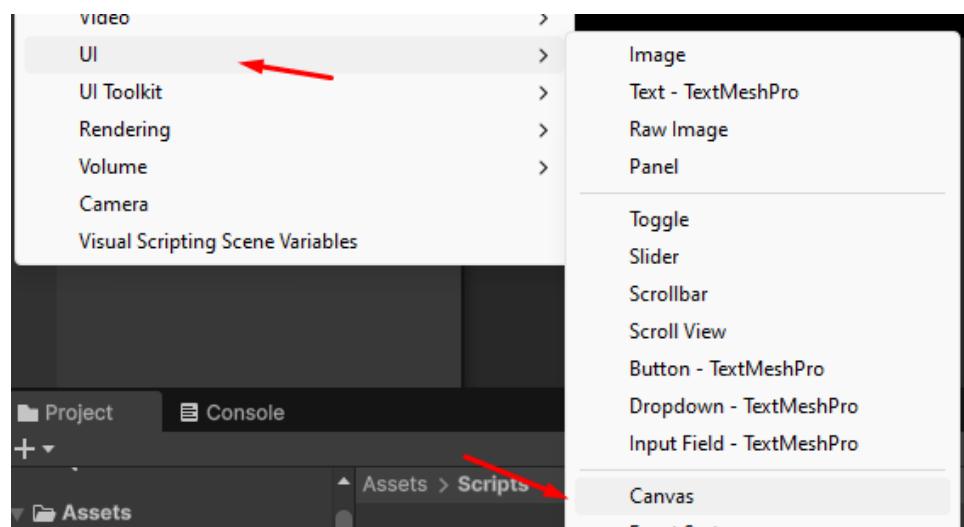
        lives--;
    }

```

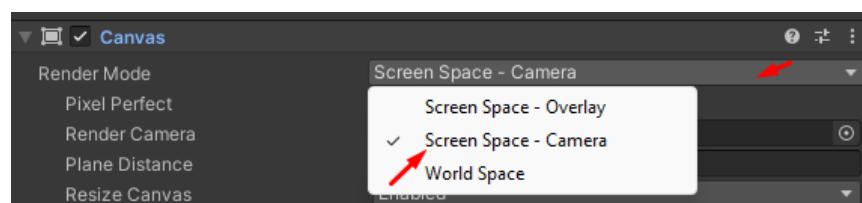
Обратите внимание что вызываем на месте **игрока**.

17. Сделаем на нашей сцене отображение **очков** и показ **жизней** игрока.

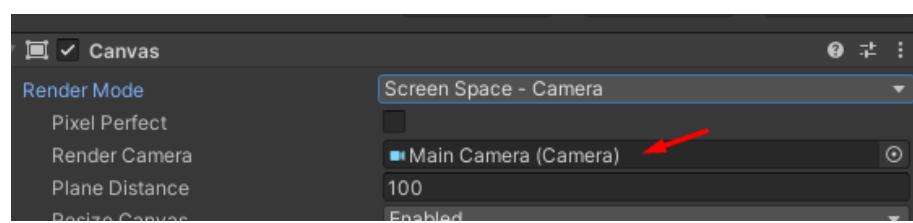
Щёлкаем правой кнопкой мыши в **иерархии** и добавляем **UI - Canvas**:



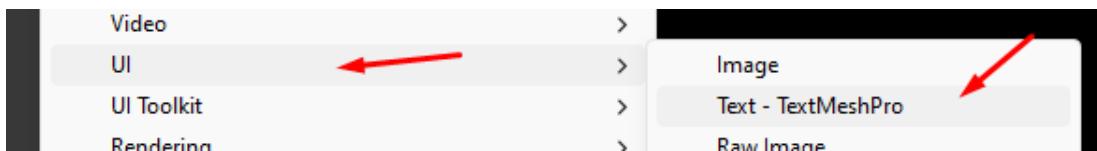
В **Inspector** выбираем в модели рендера **Screen Space-Camera**:



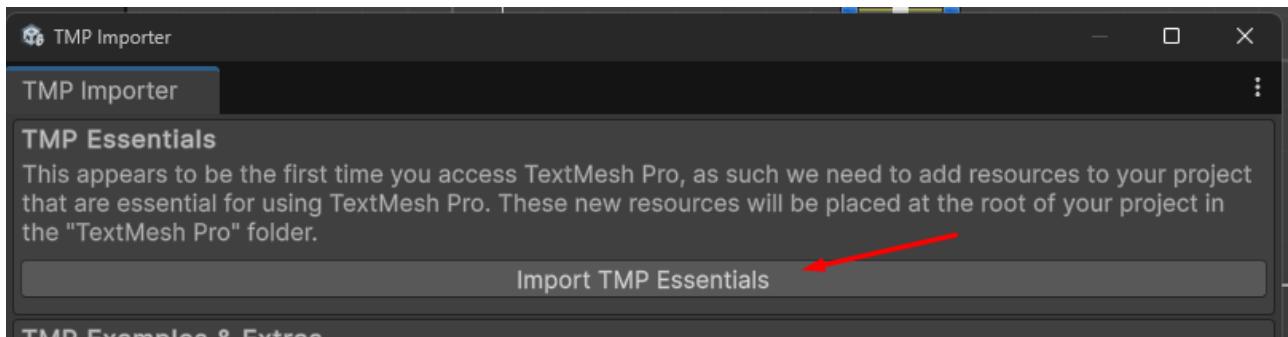
Переносим **Main Camera** в рендер:



Затем щёлкаем на **Canvas** и выбираем **UI – Text-TextMeshPro**:



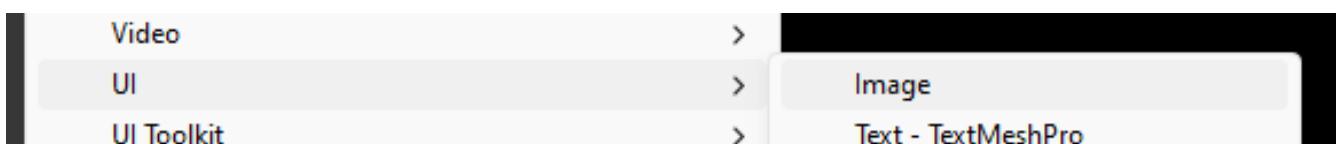
Соглашаемся на импорт и после закрываем окно:



Создаём два текста:

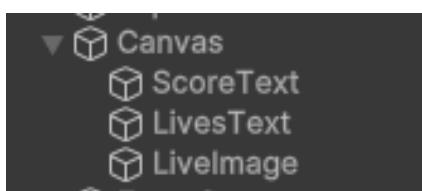
- **ScoreText**
- **LivesText**

Также добавляем **UI – Image**:

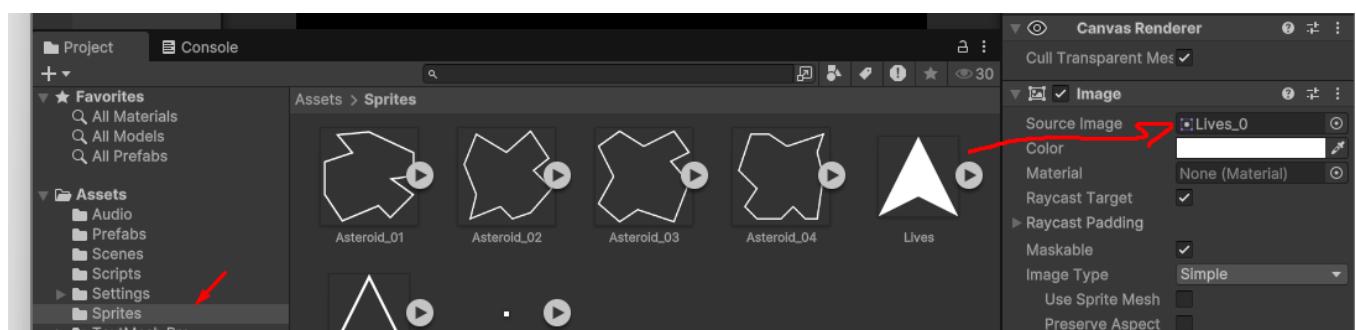


Называем его – **LiveImage**.

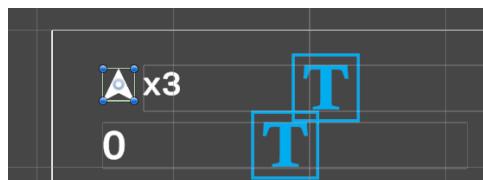
Итог:



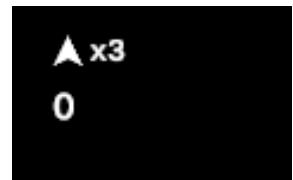
Далее в наш объект **LiveImage** в поле **Source Image** подгружаем спрайт **Lives**:



Переносим текст в левый угол, настраиваем его размеры на ваше усмотрение. В самом выводе напишем для **ScoreText** – **0**, **LivesText** – **x3**. По итогу у нас должно получиться следующее:



Окно Scene

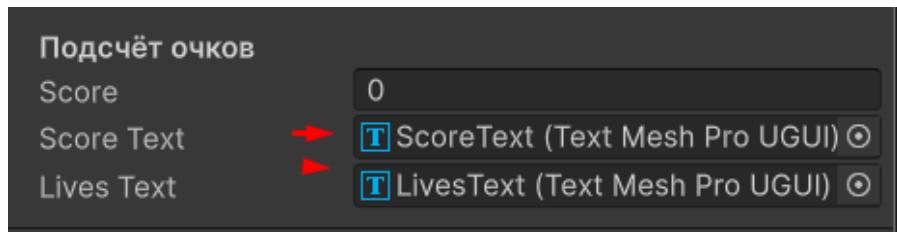


Окно Game

Перейдём к написанию кода. В скрипте **GameManager** объявим переменную для счёта и поля для отображения текста количества очков и жизней на экране:

```
[Header("Подсчёт очков")]
[SerializeField] private int score = 0;
[SerializeField] private TextMeshProUGUI scoreText;
[SerializeField] private TextMeshProUGUI livesText;
```

Перенесите для объекта **GameManager** в поля **scoreText** и **livesText** ваши текстовые объекты из **Canvas**:



Для подсчёта очков будем увеличивать их в зависимости от размера уничтоженного астероида и вызовем отображение очков. Поэтому обновим метод **AsteroidDestroyed()**:

```
public void AsteroidDestroyed(Asteroid asteroid)
{
    ParticleSystem explosionInstance = Instantiate(explosion,
asteroid.transform.position, Quaternion.identity);
    explosionInstance.Play();
    Destroy(explosionInstance.gameObject,
explosionInstance.main.duration);

    // подсчёт очков
    if (asteroid.size < 0.7f)
        score += 100;
    else if (asteroid.size < 1.4f)
        score += 50;
    else
        score += 25;

    SetScore(score); // Обновляем UI очков сразу после
изменения
}
```

Теперь мы хотим с вами создать два метода, которые будут отображать наши жизни и количество набранных очков— **SetLives()** и **SetScore()**:

```
private void SetScore(int score) // метод для подсчёта очков
{
    scoreText.text = score.ToString();
}
private void SetLives(int lives) // метод для отображения жизней
{
    livesText.text = $"x{lives}";
}
```

Обновим метод **PlayerDied** чтобы у нас отображались жизни используя метод

SetLives:

```
public void PlayerDied()
{
    if (player == null)
    {
        Debug.LogError("GameManager: Игрок не назначен!");
        return;
    }

    ParticleSystem explosionInstance = Instantiate(explosion, player.transform.position,
        Quaternion.identity);
    explosionInstance.Play();
    Destroy(explosionInstance.gameObject, explosionInstance.main.duration);

    SetLives(lives - 1); // отображаем жизни через метод
```

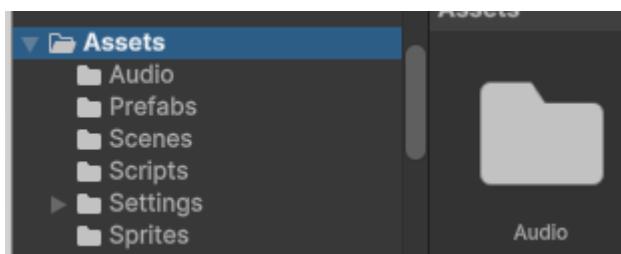


Осталось обновить метод **GameOver**. В нём мы будем обнулять переменные для очков, устанавливать количество жизней на **3**, и вызывать метод **Respawn**:

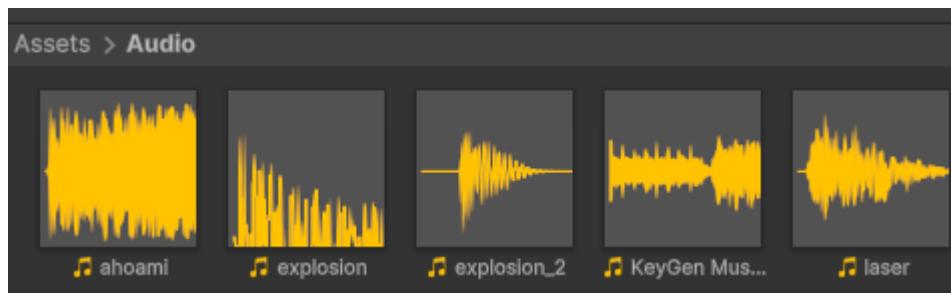
```
private void GameOver() // обнуляем данные
{
    lives = 3;
    score = 0;
    SetLives(lives); // Обновляем UI
    SetScore(score); // Обновляем UI
    Invoke(nameof(Respawn), respawnTime);
}
```

18. Добавим звуки в нашу игру. Можете воспользоваться подготовленными мной в архиве **Audio.zip**

Создаём папку **Audio**:

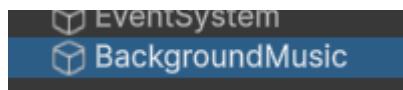


Переносим в неё звуки:

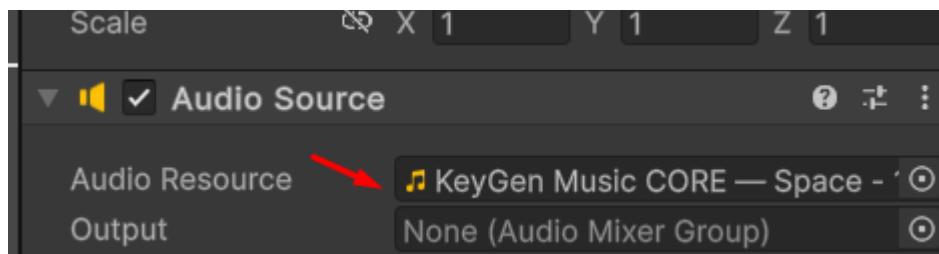


Для фоновой музыки и звуков стрельбы всё реализуется довольно просто, для звука взрыва и уничтожения корабля нам придётся с вами создать дополнительные префабы и скрипты.

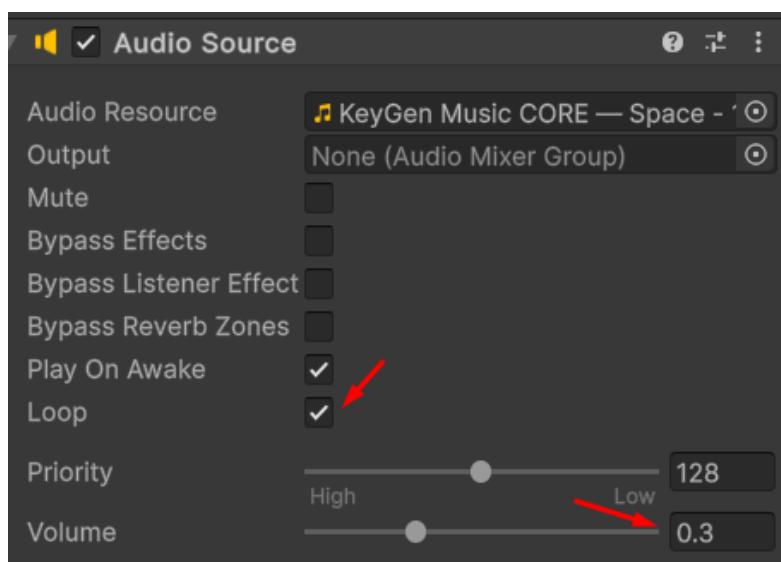
Начнём с фоновой музыки. Создайте объект **BackgroundMusic**:



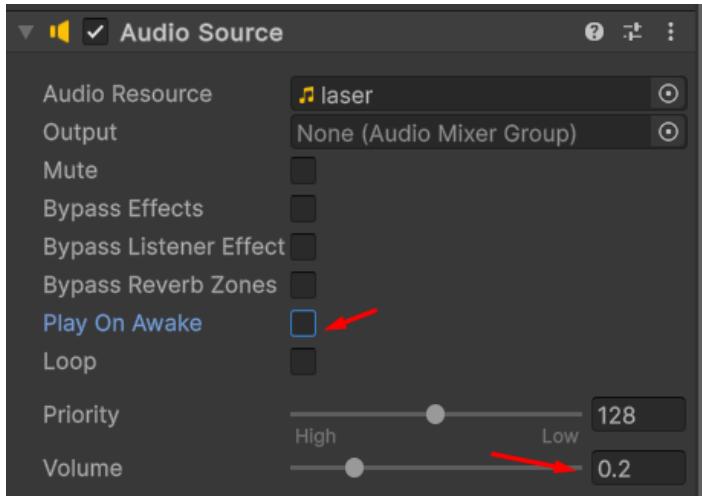
Перетащите на него звук **KeyGen Music CORE — Space - 16bit**:



Можете настроить громкость, и не забудьте поставить галочку на **Loop**, чтобы зациклить воспроизведение музыки:



Для префаба **Bullet** добавьте звук **laser**, уберите галочку в **Play On Awake** (нам не нужно воспроизводить звук при инициализации) и уменьшите громкость:



Далее перейдём в скрипт **Bullet** и обновим его:

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    [Header("Настройки пули")]
    [SerializeField] private float shootForce = 500f;
    [SerializeField] private float lifeTime = 2f;

    private Rigidbody2D _rigidbody;
    private AudioSource _audioSource; // Ссылка на
    // Получаем компонент AudioSource
    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
        _audioSource = GetComponent<
```

19. Теперь для добавления звука уничтожения корабля вам нужно создать скрипт **ExplosionSoundPlayer** и прописать в нём:

```
using UnityEngine;

public class ExplosionSoundPlayer : MonoBehaviour
{
    private AudioSource _audioSource;

    private void Awake()
    {
        _audioSource = GetComponent<AudioSource>();
    }

    public void PlaySound(Vector3 position)
    {
        transform.position = position;
        _audioSource.Play();
        Destroy(gameObject, _audioSource.clip.length); // Уничтожаем объект после воспроизведения звука
    }
}
```

Затем в скрипте **Player** добавляем переменную:

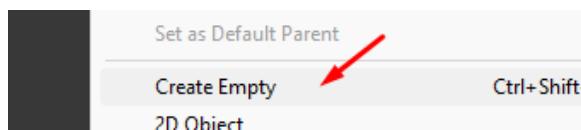
```
public Bullet bulletPrefab;
public ExplosionSoundPlayer explosionSoundPlayer; // префаб для звука
```

И обновляем метод **OnCollisionEnter2D**:

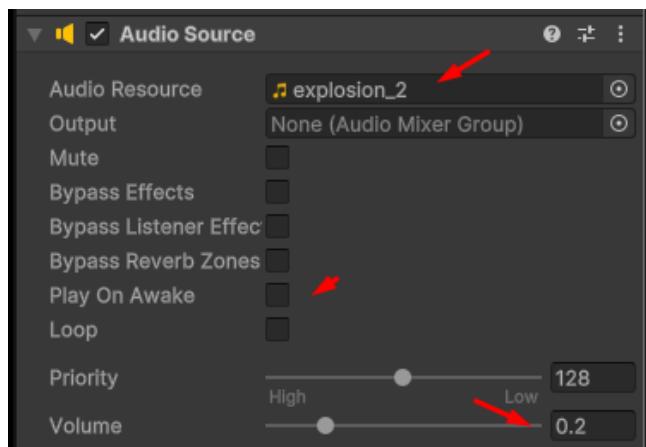
```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Asteroid"))
    {
        ExplosionSoundPlayer explosionSound =
Instantiate(explosionSoundPlayer); // создаём объект звука взрыва
        explosionSound.PlaySound(transform.position); // воспроизводим взрыв
        _rigidbody.linearVelocity = Vector3.zero;
        _rigidbody.angularVelocity = 0f;

        gameObject.SetActive(false);
        FindFirstObjectByType<GameManager>().PlayerDied();
    }
}
```

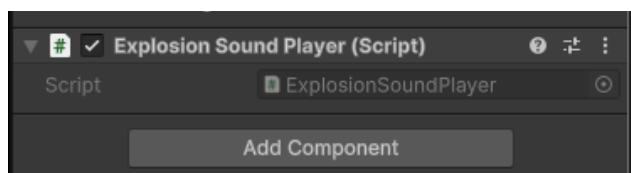
Затем создаём новый объект и называем его **ExplosionSoundPlayer**



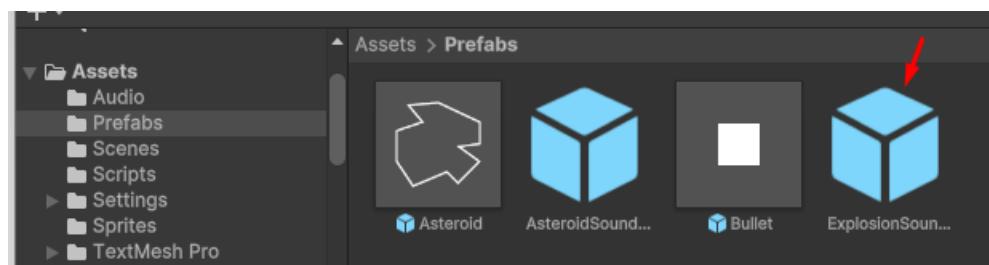
Добавляем к нему **Audio Source**, выбираем и настраиваем громкость звука и уберите галочку в **Play On Awake** (нам не нужно воспроизводить звук при инициализации):



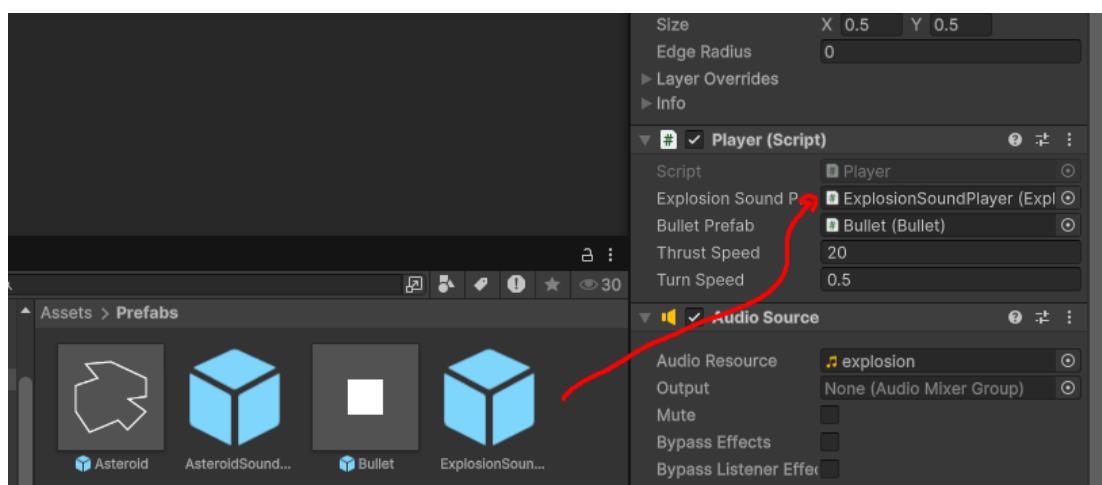
Также добавьте к нему наш скрипт **ExplosionSoundPlayer**:



После перенесите ваш объект в папку с **префабами**, и удалите его из **иерархии**:



Осталось перенести наш префаб в скрипт **Player** в поле **ExplosionSoundPlayer**:



20. Аналогично нам нужно сделать для звуков уничтожения астероидов. Создаём скрипт **AsteroidSoundPlayer**, пропишем в нём:

```
using UnityEngine;

public class AsteroidSoundPlayer : MonoBehaviour
{
    private AudioSource _audioSource;

    private void Awake()
    {
        _audioSource = GetComponent<AudioSource>();
    }

    public void PlaySound(Vector3 position)
    {
        transform.position = position;
        _audioSource.Play();
        Destroy(gameObject, _audioSource.clip.length); // Уничтожаем объект после воспроизведения звука
    }
}
```

Затем в скрипте **Asteroid** добавляем в **переменную**:

```
public AsteroidSoundPlayer explosionSoundPrefab; // префаб для воспроизведения звука взрыва
```

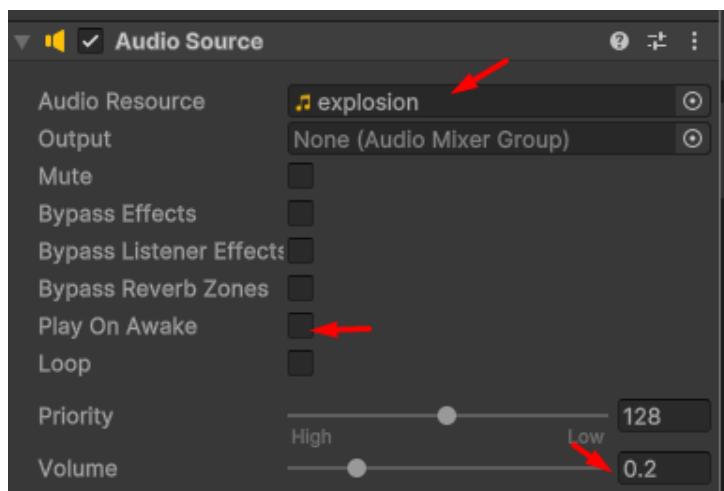
И обновляем метод **OnCollisionEnter2D**:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Bullet"))
    {
        AsteroidSoundPlayer explosionSound =
Instantiate(explosionSoundPrefab); // создаём объект звука взрыва
        explosionSound.PlaySound(transform.position); // воспроизводим взрыв
        if ((size * 0.5f) >= minSize)
        {
            CreateSplit();
            CreateSplit();
        }
        FindFirstObjectOfType<GameManager>().AsteroidDestroyed(this);
        Destroy(gameObject);
    }
}
```

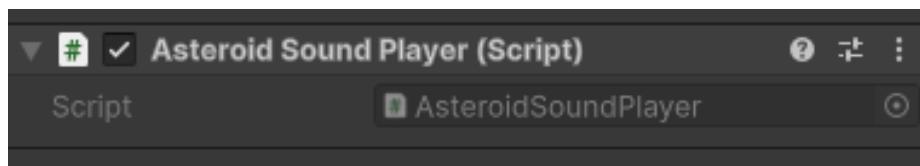
Затем создаём новый объект и называем его **AsteroidSoundPlayer**



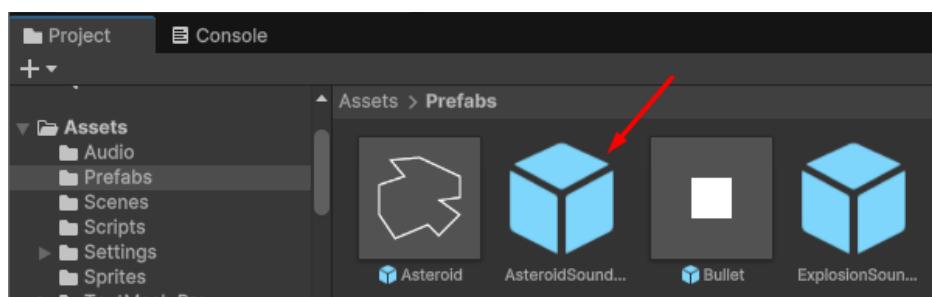
Добавляем к нему **Audio Source**, выбираем и настраиваем громкость звука и уберите галочку в **Play On Awake** (нам не нужно воспроизводить звук при инициализации):



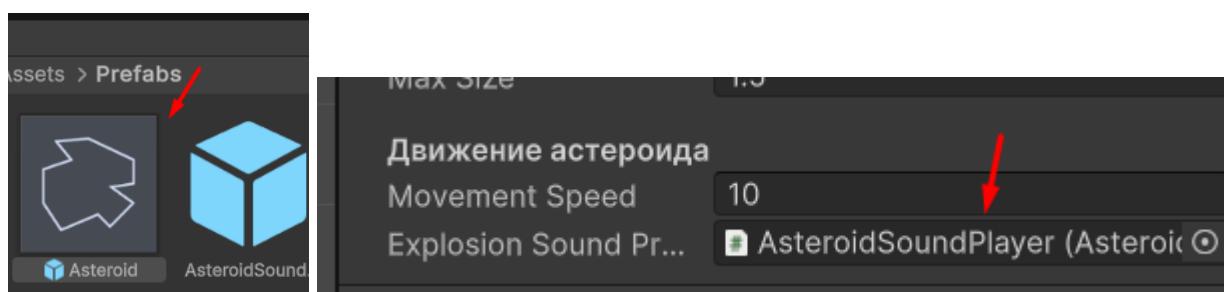
Также добавьте к нему наш скрипт **AsteroidSoundPlayer**:



После перенесите ваш объект в папку с **префабами**, и удалите его из **иерархии**:



Осталось перенести наш префаб в **префаб Asteroid** в поле **ExplosionSoundPrefab**:

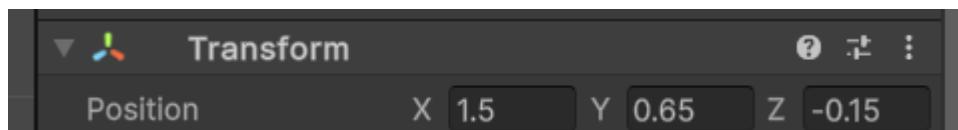


21. Реализуем эффекты частиц на экране. ПКМ – Effects – Particle System:

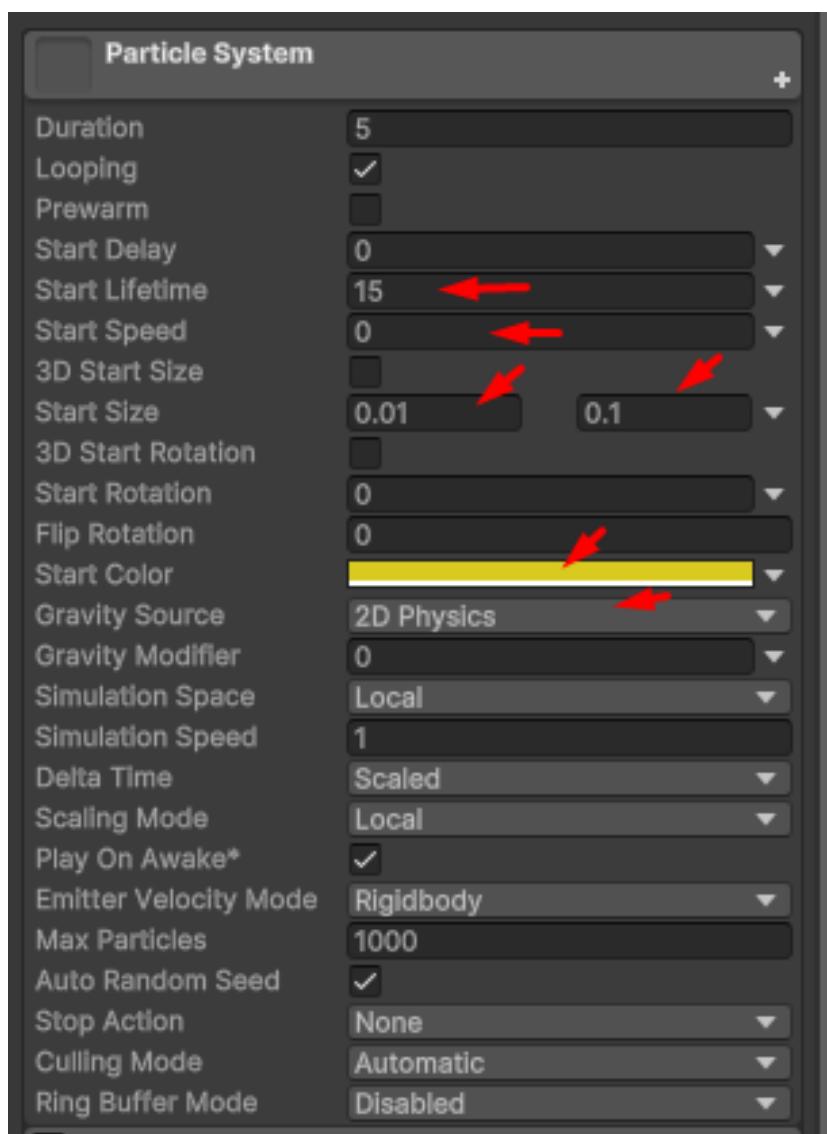


Можете использовать мои настройки, или поэкспериментировать со своими:

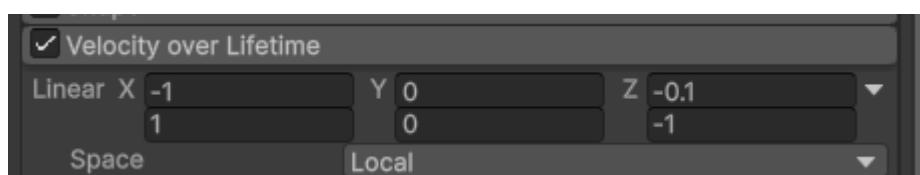
Position:



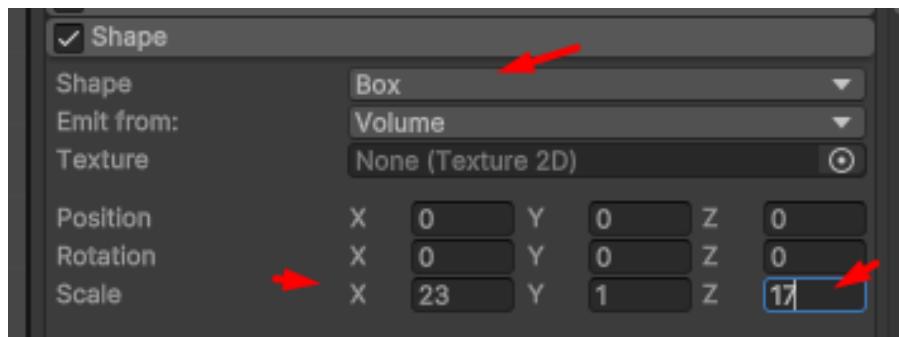
Основные параметры:



Velocity over Lifetime:



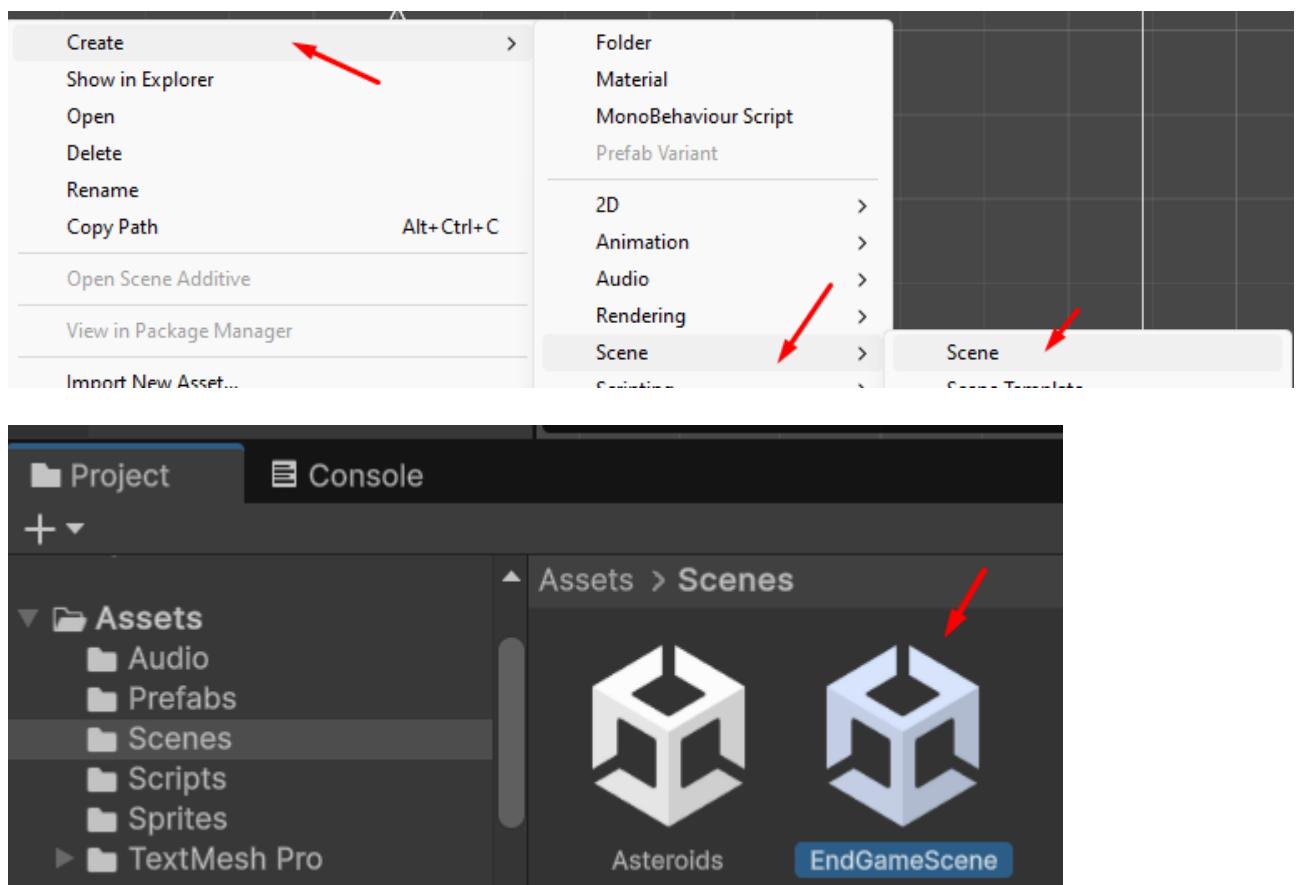
Shape:



22. Если мы хотим выключить отображение курсора при начале игры, то в скрипте **GameManager** добавим новый метод **Start()**:

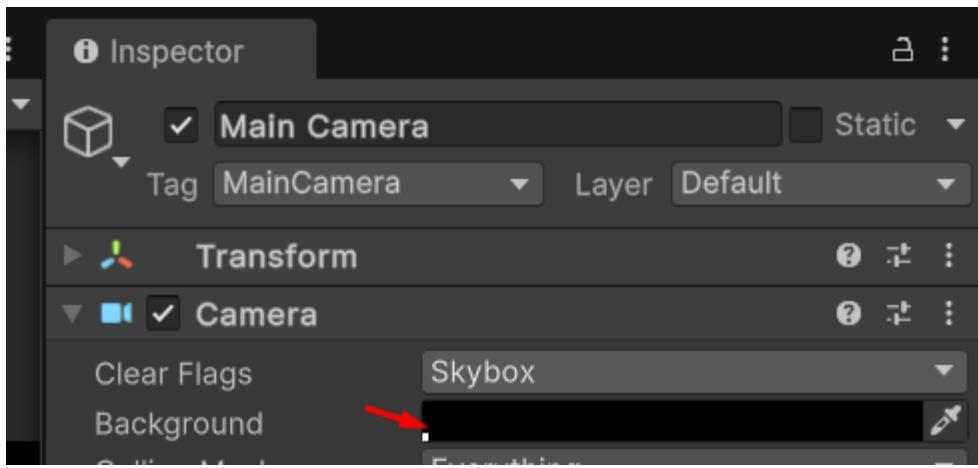
```
private void Start()
{
    Cursor.visible = false; // отключаем курсор
}
```

23. Добавим новую финальную сцену с названием **EndGameScene** в папке **Scenes**:

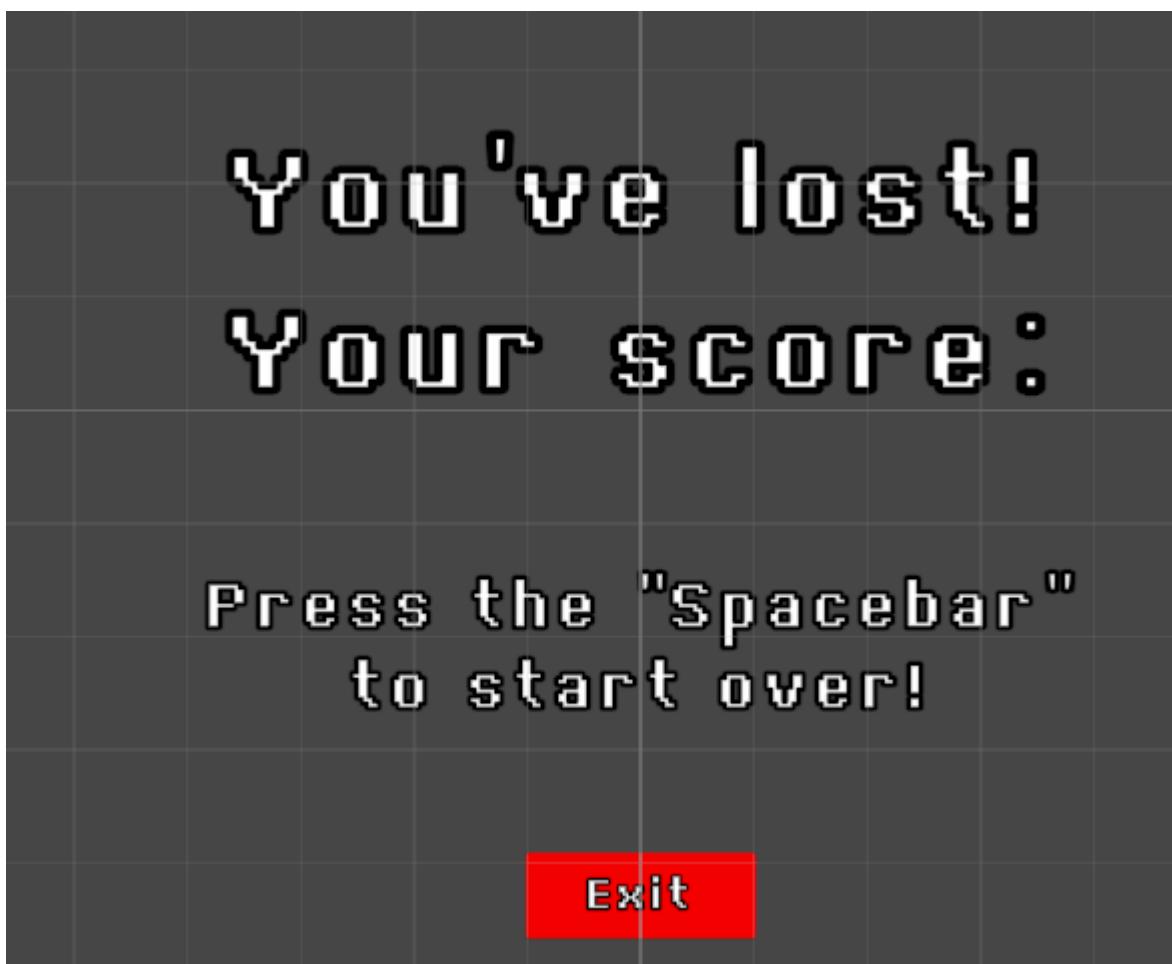


Переходим на неё.

Меняем цвет камеры на чёрный:



В ней мы хотим реализовать **показ набранных нами очков, возможность продолжать игру по нажатии клавиши**, а также **кнопку выхода из игры**:



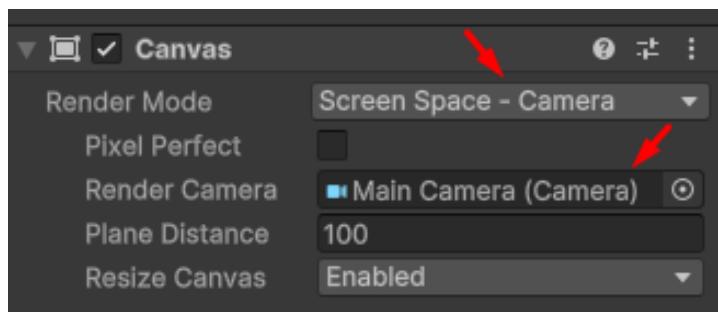
Окна сцены окончания игры

Создаём UI-Text-TextMeshPro:



Называем его **FinalScore**.

Для нашего Canvas меняем отображение на **Screen Space – Camera**, и добавляем нашу камеру:

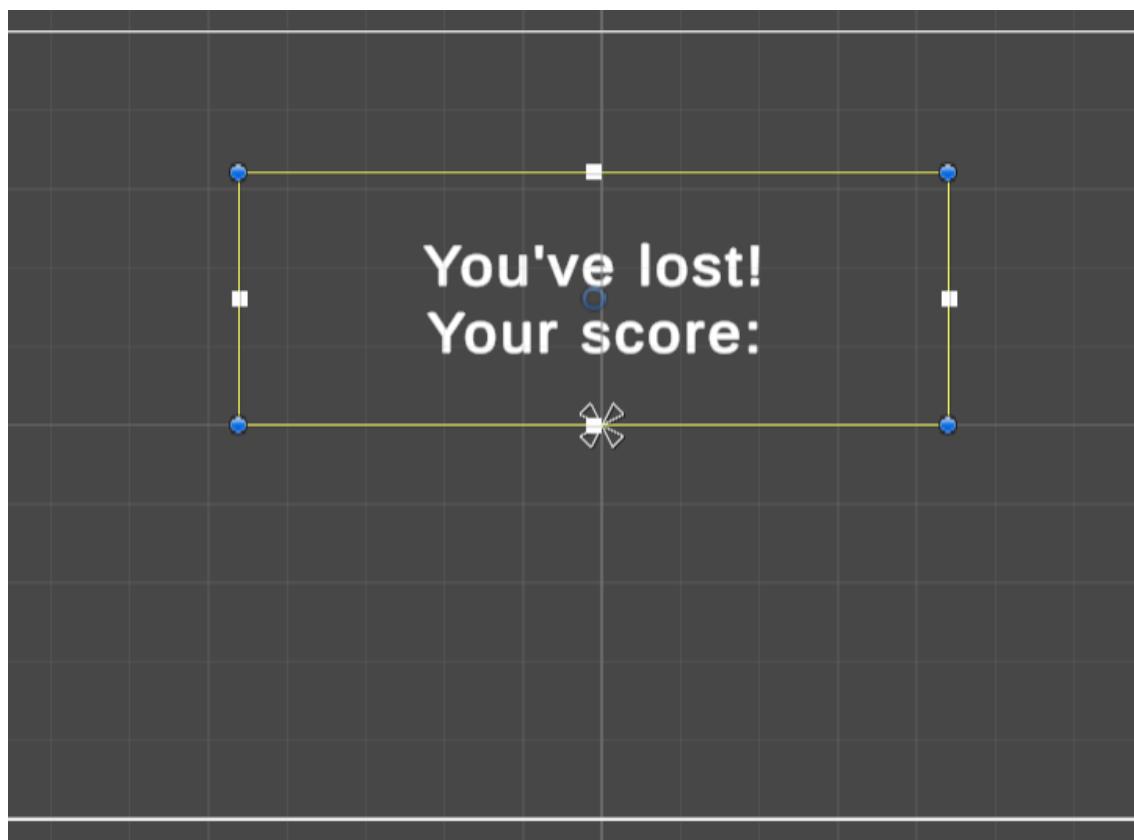


В тексте пишем:

You've lost!

Your score:

И расположим его по середине экрана:



24. Давайте загрузим и установим шрифт с интернета.

Для примера перейдём на сайт:

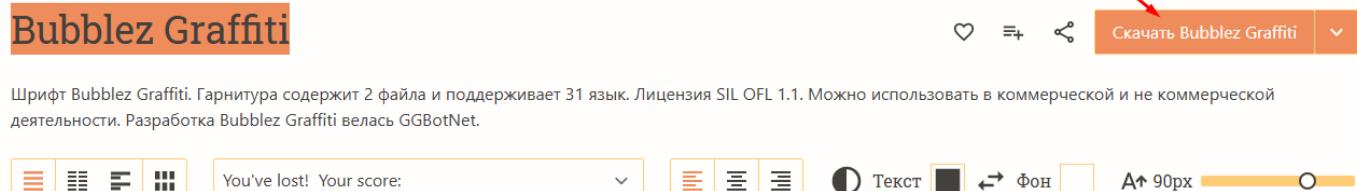
<https://fonts-online.ru/categories/pixel-fonts>

Я выберу шрифт **Bubblez Graffiti**, вы можете использовать любой другой:

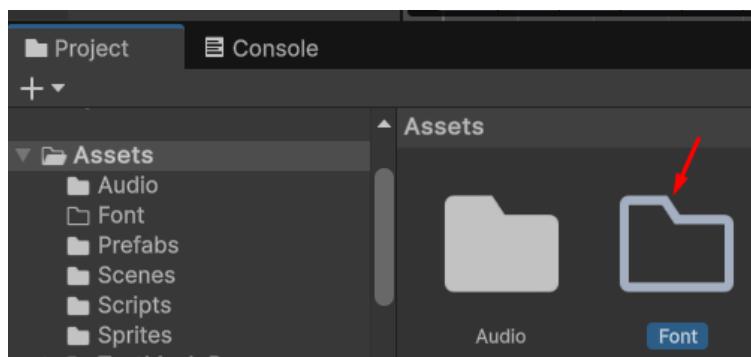
УДОБНЫЕ ВОСЬМЬ ШРИФТВ:)

Нажимаем скачать:

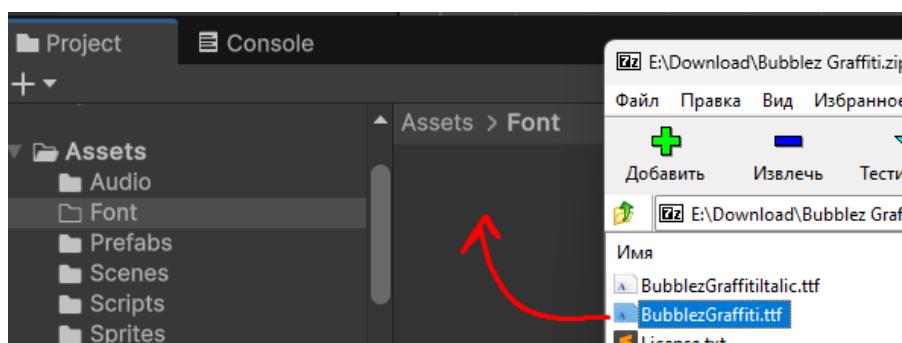
Шрифты Онлайн > Шрифты > Bubblez Graffiti



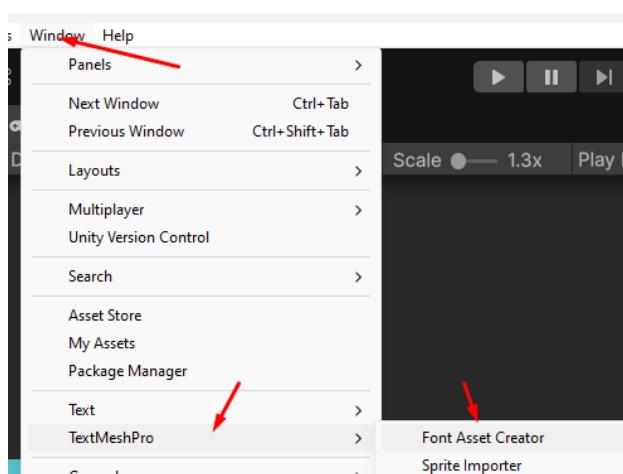
Затем создадим папку Font:



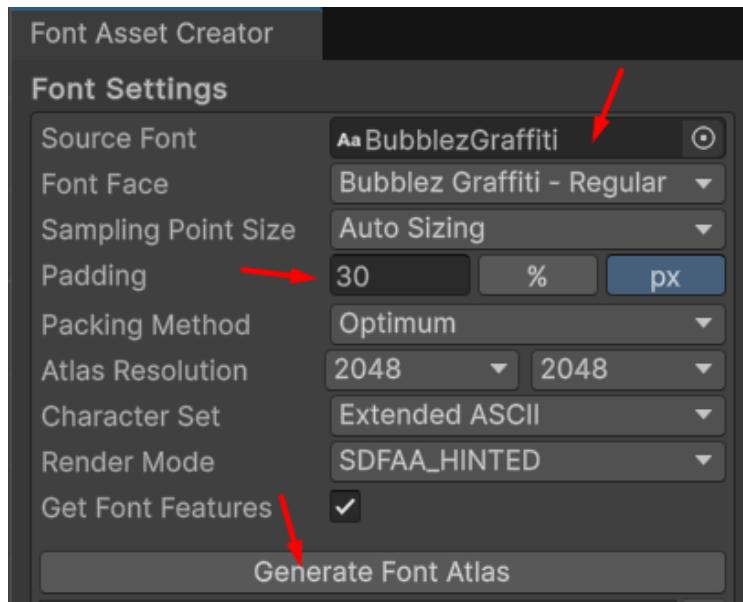
И перенесём в неё шрифт из архива (в качестве примера я возьму обычный):



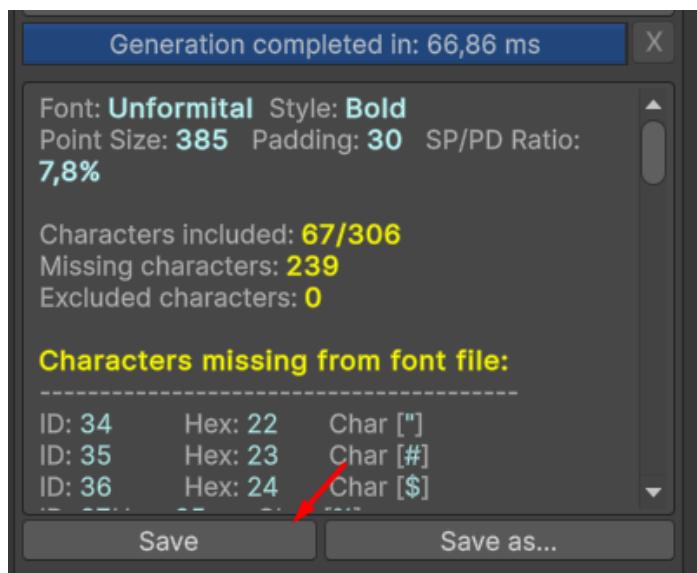
Затем нажимаем на вкладку Window – TextMeshPro – Font Assets Creator:



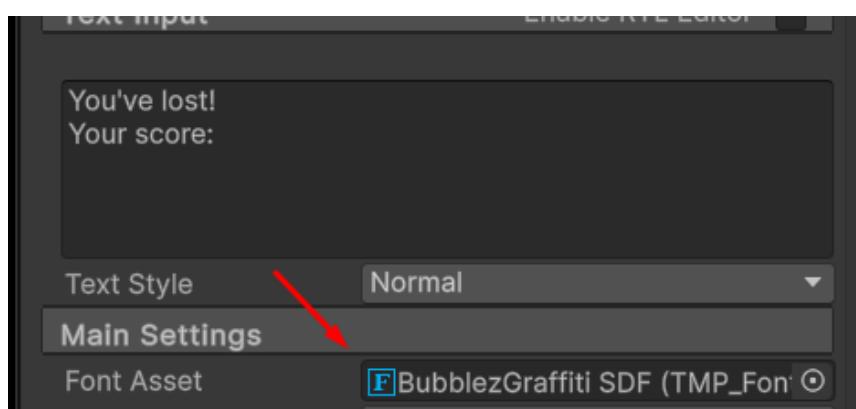
Дальше выберите **шрифт**, поставьте **Padding** на **30**, **Atlas Resolution** на **2048x2048**, и нажимаем **Generate Font Atlas**:



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

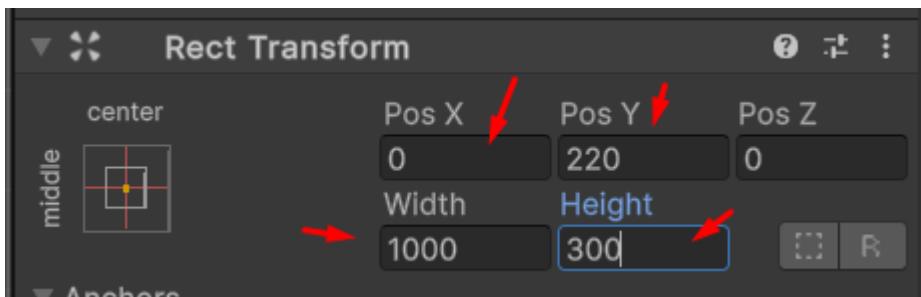


После наш **ассет шрифта** мы можем назначить в **инспекторе** на **Font Asset**:



Оформление можете использовать как у меня, или придумать своё.

Позиции:



Размер шрифта 120:

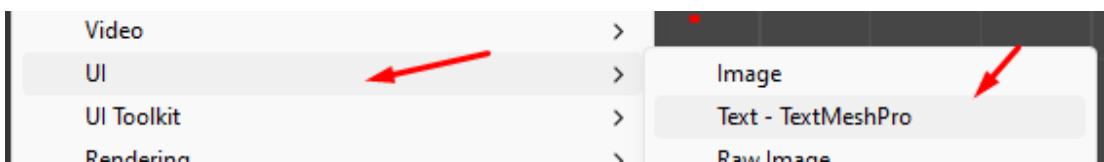


Выравнивание по центру:



Обратите внимание, нельзя ставить полужирный шрифт, иначе у вас слетит отображение шрифта!

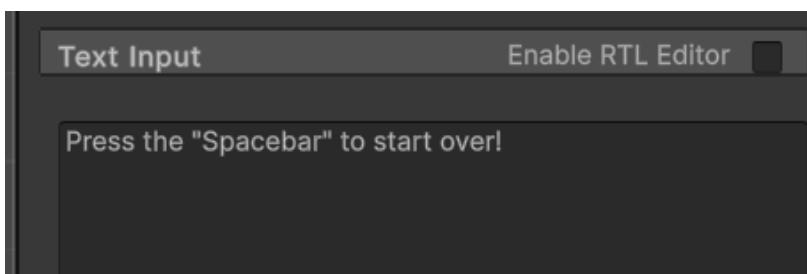
Создаём новый **UI-Text-TextMeshPro**:



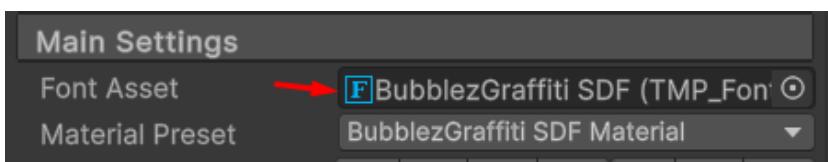
Называем его **PlayAgain**.

В тексте напишите:

Press the "Spacebar" to start over!

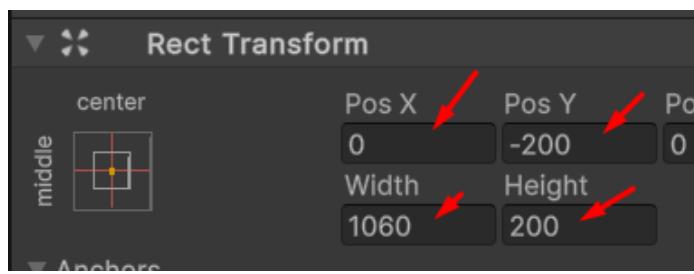


Ассет шрифта назначаем в инспекторе на Font Asset:



Оформление можете использовать как у меня, или придумать своё.

Позиции:



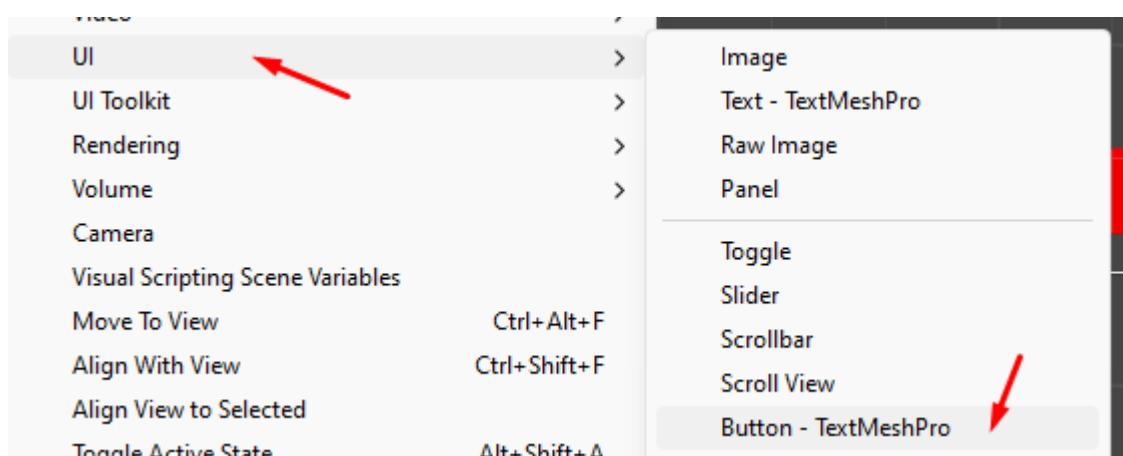
Размер шрифта 80:



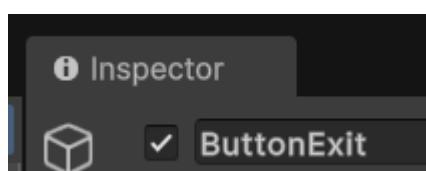
Выравнивание по центру:



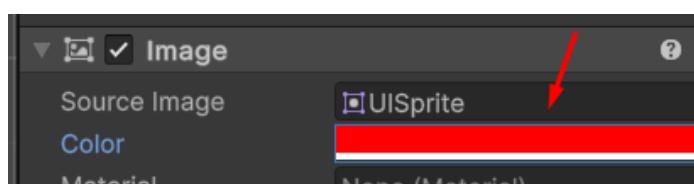
Теперь осталось сделать кнопку. Внутри **Canvas** щёлкаем правой кнопкой мыши и выбираем **UI – Button - TextMeshPro**:



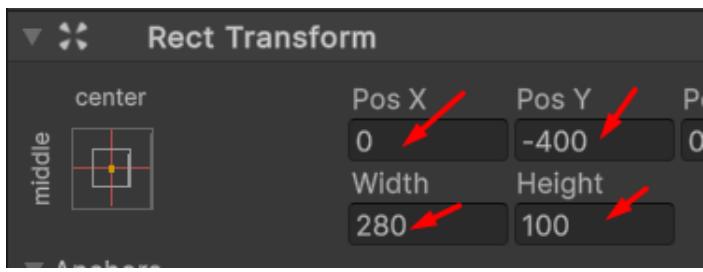
Называем её **ButtonExit**.



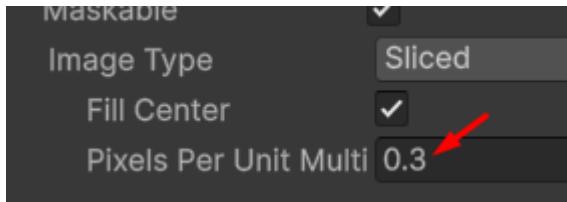
Меняем цвет на **красный**:



Настраиваем позицию и размеры:



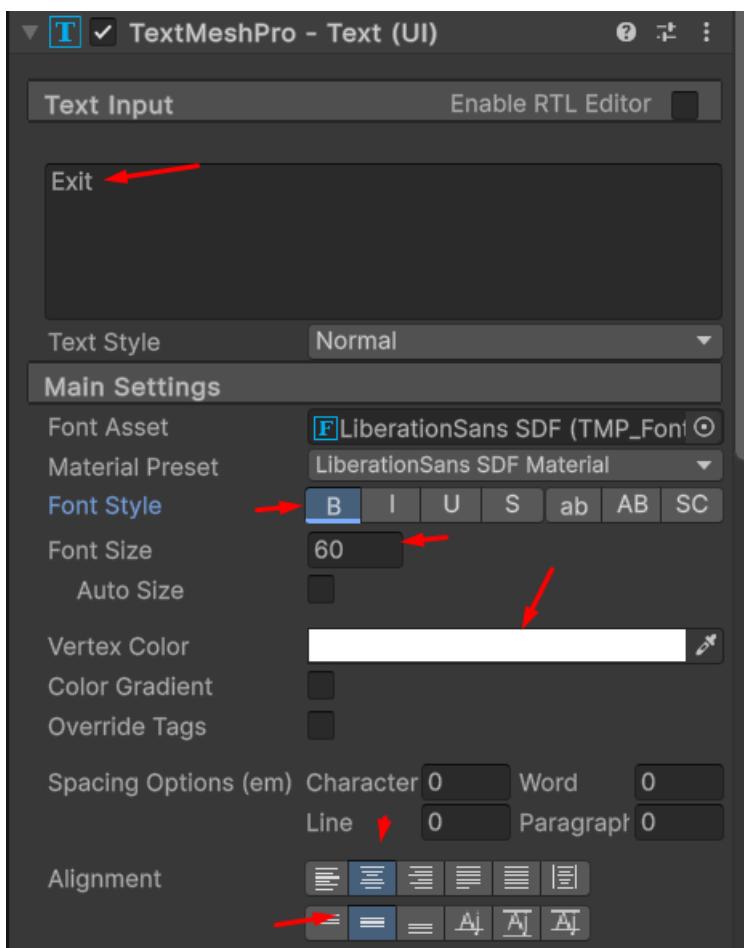
Делаем кнопку округленной:



Теперь редактируем сам текст внутри кнопки:



Меняем текст на **Exit**, делаем его **полужирным**, цвет выбираем **белый**, **выравнивание по центру**:



25. Добавим на сцену **систему частиц**.

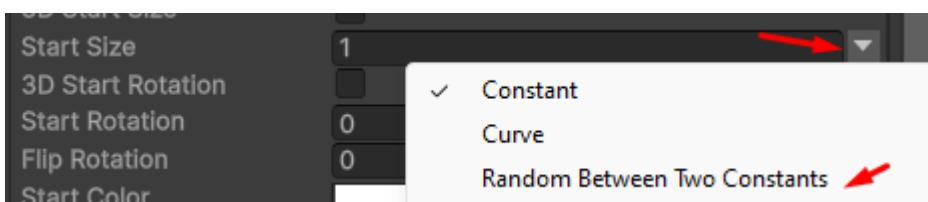
Щёлкаем правой кнопкой мыши, выбираем Effects – Particle System:



Меняем Start LifeTime на 15:



Начальный размер сделаем вариацию из двух чисел. Для этого нам нужно нажать на стрелочку у Start Size и выбрать Random Between Two Constants:



Выставляем два значения 0.001 и 0.1:



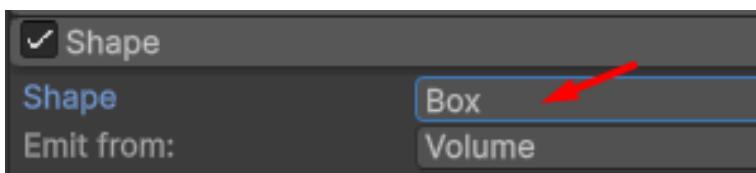
Меняем цвет на красный:



Gravity Source меняем на 2D Physics:



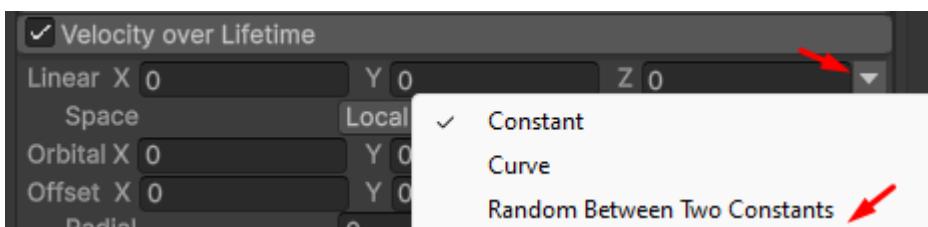
Shape поменяем на Box:



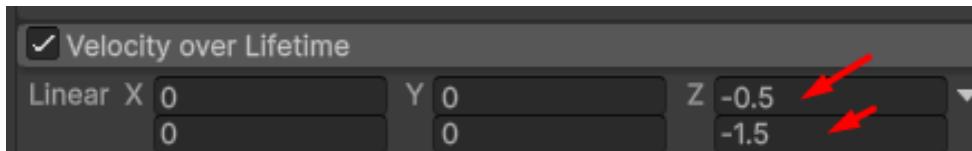
Размер поставим по x – 24, по z – 20:



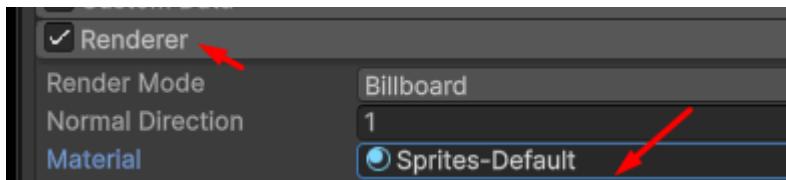
Ставим галочку у Velocity over Lifetime и на стрелку сделаем выбор двух значений:



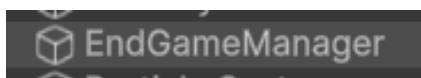
Меняем у Z значения на -0.5 и -1.5:



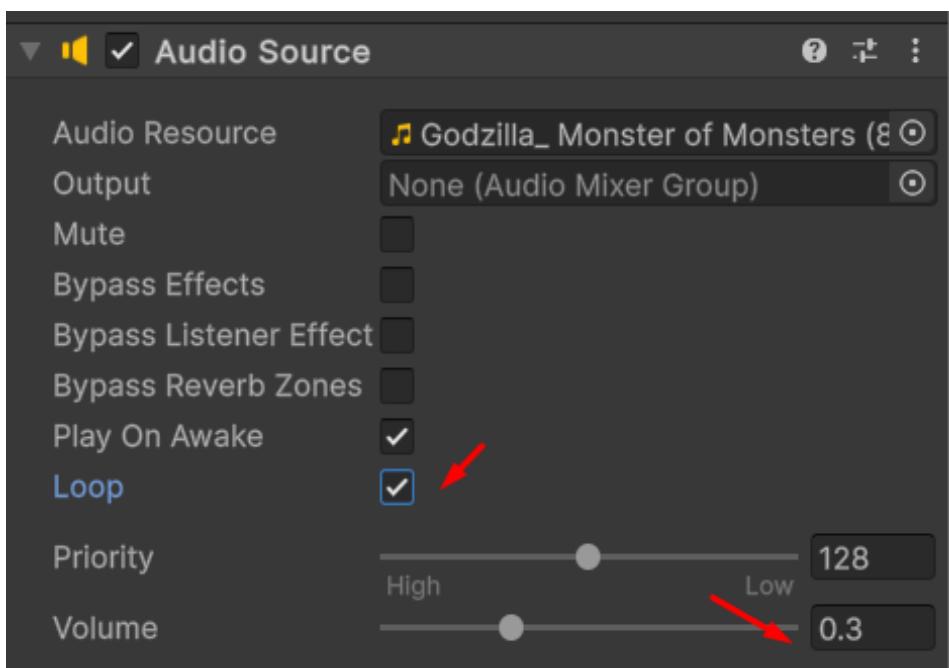
В Renderer поменяем Material на Sprites-Default:



26. Теперь мы хотим добавить на нашу сцену музыку и назначить скрипты. Для этого создаём новый пустой объект и называем его **EndGameManager**:



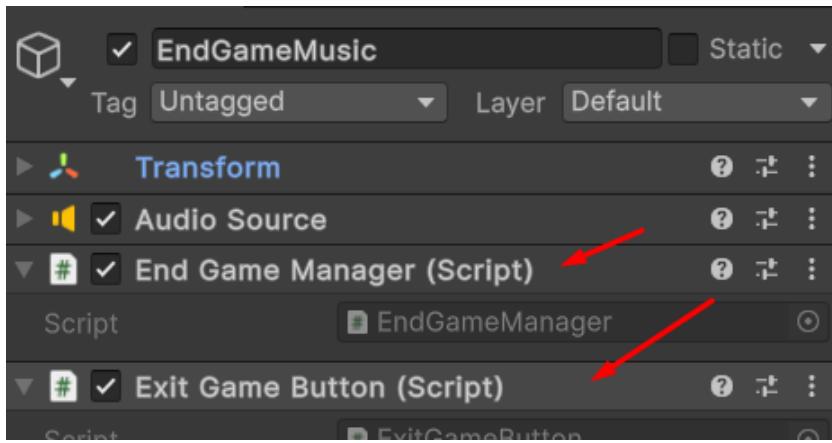
Добавляем на него из папки **Audio** звук - **Godzilla_ Monster of Monsters (8 bit_dendy)**—**Game Over**, настраиваем громкость и включаем зацикливание:



Затем нам нужно создать в папке со скриптами два новых – **EndGameManager** и **ExitGameManager**:



Прикрепляем их на наш объект **EndGameManager**:



Открываем скрипт **ExitGameButton**. Его логика заключается в том, что, когда игрок нажимает на кнопку "Exit" в нашей игре, то мы выходим из игры. Также сделаем возможность закрывать игру в самой сцене используя **препроцессоры**.

Пишем код:

```
using UnityEngine;

public class ExitGameButton : MonoBehaviour
{
    public void ExitGame()
    {
        // Для завершения игры в редакторе Unity:
#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
#else
        // Для завершения игры на устройстве:
        Application.Quit();
#endif
    }
}
```

Директивы препроцессора #:

1. В данном скрипте используются директивы препроцессора `#if` и `#else` для проверки условий компиляции. Это необходимо, чтобы различать поведение программы в зависимости от среды выполнения (редактор Unity или целевое устройство).

Объяснение директив:

2. **#if UNITY_EDITOR:**
 - Эта директива проверяет, выполняется ли код в редакторе **Unity**. Если это так, выполняется следующий блок кода, вплоть до `#else`.
3. **UnityEditor.EditorApplication.isPlaying = false;**
 - Эта строка завершает выполнение игры в редакторе **Unity**. Установка значения **false** останавливает режим игры.

4. #else:

- Если код не выполняется в редакторе **Unity**, то выполняется следующий блок кода, вплоть до **#endif**.

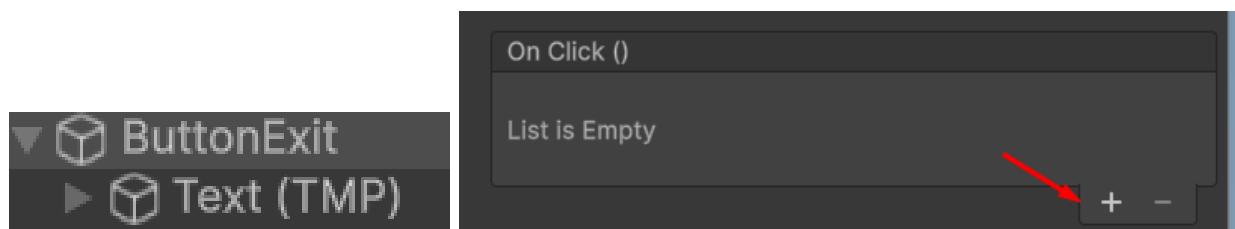
5. Application.Quit();

- Эта строка завершает выполнение игры на целевом устройстве. Метод **Application.Quit()** закрывает приложение.

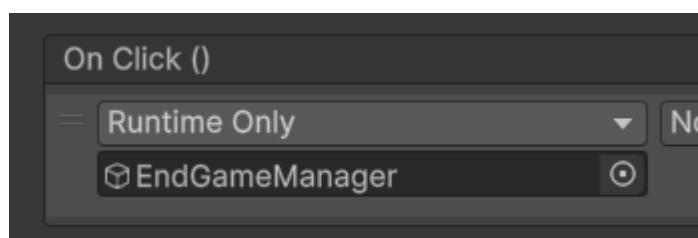
6. #endif:

- Конец блока директив препроцессора. Эта директива завершает условную компиляцию.

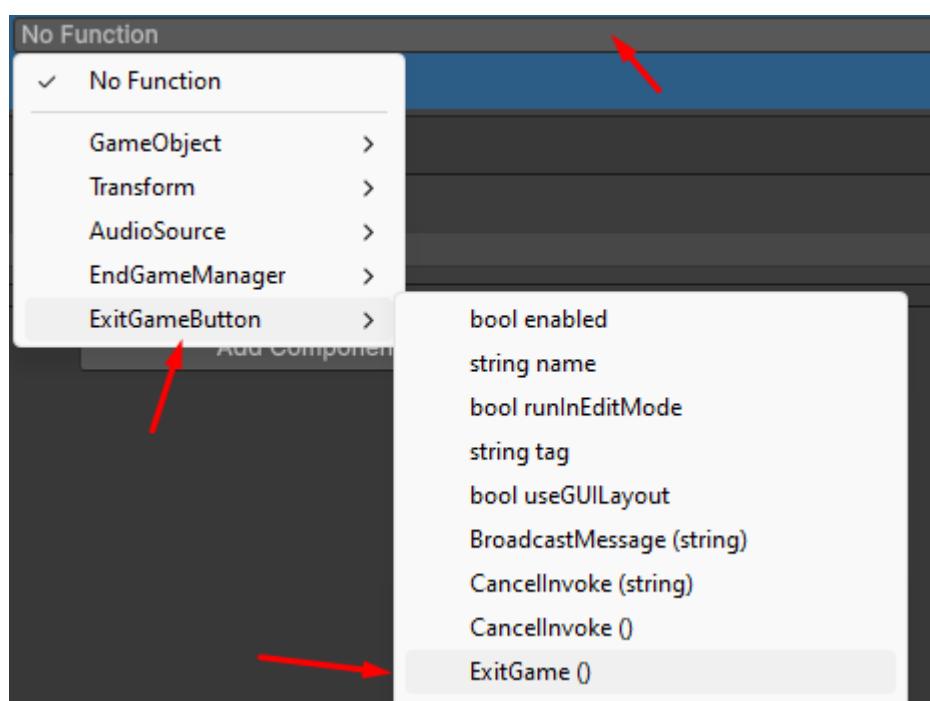
Добавляем в нашу **кнопку** новое событие в **On Click()** нажав на +:



После нам нужно перенести наш **объект EndGameManager** в пустое поле:



Далее щёлкаем на поле **Function**, и выбираем **ExitGameButton – ExitGame()**:



Запускаем и проверяем что при нажатии на **кнопку Exit** игра закрывается.

27. Теперь откроем и напишем код для скрипта **EndGameManager**:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

public class EndGameManager : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI finalScoreText; // Текст для отображения финальных очков

    private void Start()
    {
        Cursor.visible = true; // Показываем курсор, так как игра окончена
        Cursor.lockState = CursorLockMode.None; // Разблокируем курсор

        // Получаем финальные очки из PlayerPrefs
        int finalScore = PlayerPrefs.GetInt("FinalScore", 0);
        finalScoreText.text = $"You've lost! \nYour score: {finalScore}"; // Отображаем очки
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            RestartGame(); // Перезапускаем игру
        }
    }

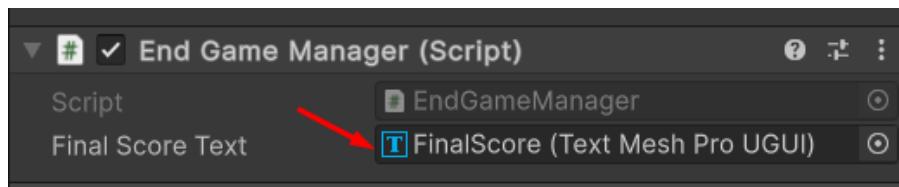
    private void RestartGame()
    {
        SceneManager.LoadScene("Asteroids"); // Загрузка основной сцены
    }
}
```

Мы дополнительно подключили библиотеку **UnityEngine.SceneManagement**, которая импортирует библиотеку для управления сценами в Unity.

Также мы используем **PlayerPrefs** - встроенный класс в Unity, используемый для хранения и извлечения данных между сессиями игры. Он часто применяется для

сохранения настроек, результатов игры и других данных, которые нужно сохранять между запусками игры.

Далее нам нужно перенести наш текст в **поле** у объекта **EndGameManager** в скрипте **EndGameManager**:



Для скрипта **GameManager** внесём правки. Добавляем новый метод **EndGame()**:

```
private void EndGame()
{
    // Сохранение очков в PlayerPrefs, для отображения в
    // конце игры
    PlayerPrefs.SetInt("FinalScore", score);
    // Переход на сцену конца игры
    SceneManager.LoadScene("EndGameScene");
}
```

И в методе **PlayerDied()** вызовем его при **0** жизнях:

```
lives--;
SetLives(lives); // отображаем жизни через

if (lives <= 0)
{
    EndGame(); // Код, выделенный красным квадратом
}
else
{
```

Также теперь мы можем удалить метод **GameOver()**:

```
Ссылок: 0
private void GameOver() // обнуляем данные
{
    lives = 3;
    score = 0;
    SetLives(lives); // Обновляем UI
    SetScore(score); // Обновляем UI
    Invoke(nameof(Respawn), respawnTime);
}

Ссылок: 1
```

28. Добавим эффект мерцания с помощью **Coroutine**, изменяя прозрачность спрайта игрока.

В скрипте **GameManager** подключаем пространство имён **System.Collections**:

```
using System.Collections;
```

В нём же создадим новый метод **BlinkPlayer()**:

```
private IEnumerator BlinkPlayer()
{
    // Получаем компонент SpriteRenderer для изменения цвета
    // спрайта игрока
    SpriteRenderer spriteRenderer =
    player.GetComponent<SpriteRenderer>();
    // Если у игрока нет SpriteRenderer, завершаем корутину
    if (spriteRenderer == null)
        yield break;

    float blinkDuration = 3f; // Длительность неуязвимости
    // (секунды)
    float blinkInterval = 0.2f; // Интервал мерцания (секунды)
    float elapsedTime = 0f; // Отслеживание прошедшего времени
    bool isVisible = true; // Флаг, указывающий, должен ли
    // игрок быть видимым

    // Пока прошло времени меньше, чем продолжительность
    // неуязвимости
    while (elapsedTime < blinkDuration)
    {
        isVisible = !isVisible; // Переключаем состояние
        // (видимый / невидимый)
        // Меняем прозрачность: если isVisible – 1f (полностью
        // видимый), иначе 0.3f (полупрозрачный)
        spriteRenderer.color = new Color(1f, 1f, 1f, isVisible
        ? 1f : 0.3f);
        elapsedTime += blinkInterval; // Увеличиваем прошедшее
        // время
        yield return new WaitForSeconds(blinkInterval); // Ждем
        // заданный интервал перед следующим изменением
    }
    // После завершения неуязвимости возвращаем игроку
    // нормальный цвет
    spriteRenderer.color = Color.white;
}
```

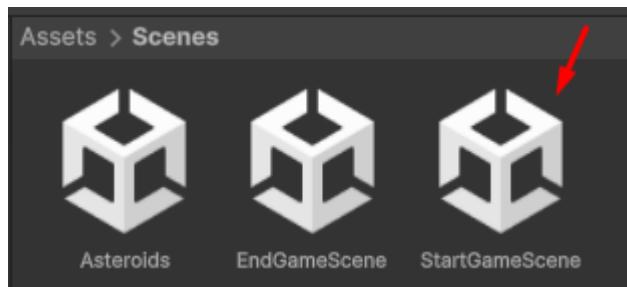
В методе Respawn() запустим мерцание:

```
player.gameObject.SetActive(true);  
StartCoroutine(BlinkPlayer()); // Запускаем мерцание  
  
Invoke(nameof(TurnOnCollisions), 3f);
```

Как он работает в игре?

- При респауне запускается **корутина**.
- Игрок **мерцает** ($1\text{f} \rightarrow 0.3\text{f} \rightarrow 1\text{f}$) в течение 3 секунд.
- После завершения он **становится полностью видимым** и уязвимым.

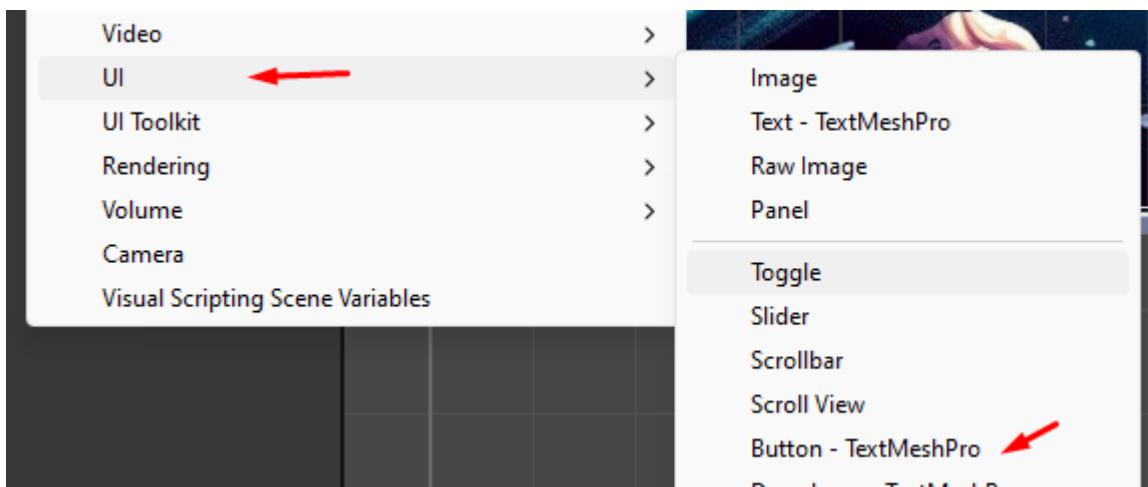
29. Теперь создадим начальную сцену в папке **Scenes**, и назовём её **StartGameScene**:



В ней мы хотим поставить **начальную картинку** и **две кнопки** – выхода из игры, и кнопку начала игры:

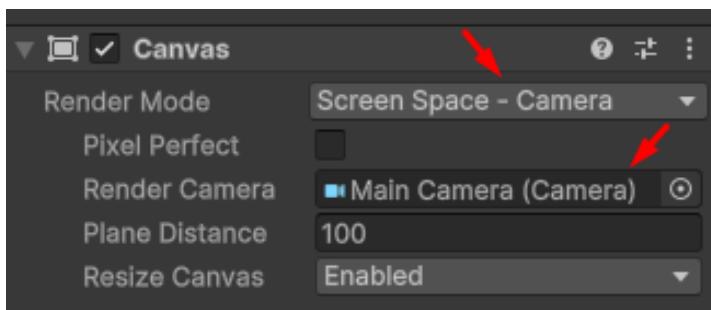


Создаём UI-Button-TextMeshPro:

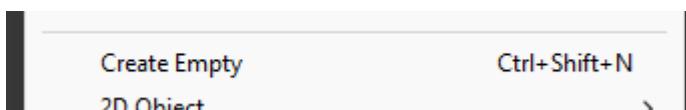


Также создаём вторую кнопку, и называем их **ButtonStart** и **ButtonExit** соответственно. Оформляем их на свой вкус.

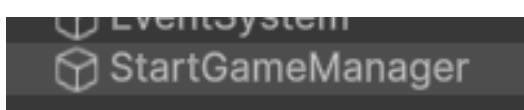
Для нашего Canvas меняем отображение на **Screen Space – Camera**, и добавляем нашу камеру:



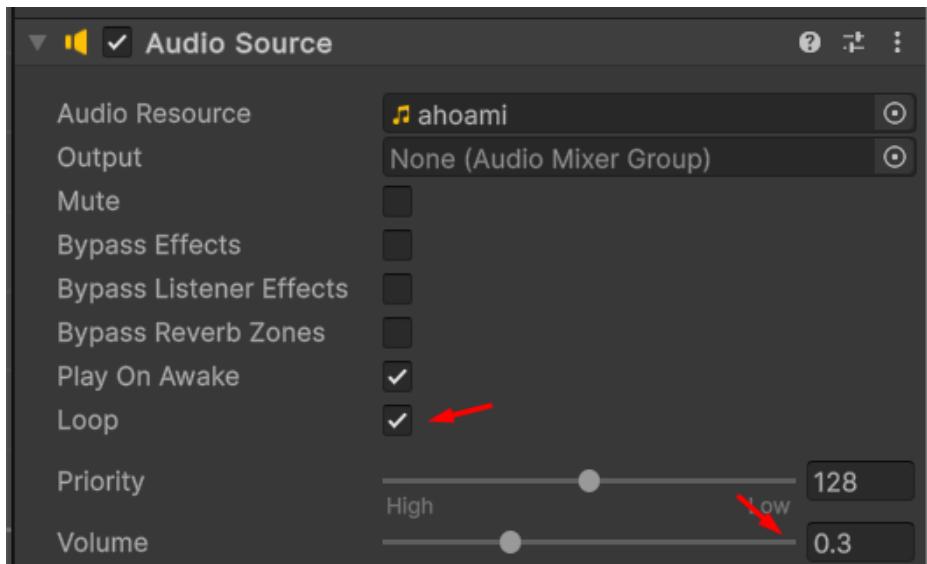
Теперь мы хотим добавить на нашу сцену **музыку, картинку** и назначить **скрипты**. Для этого создаём новый пустой объект:



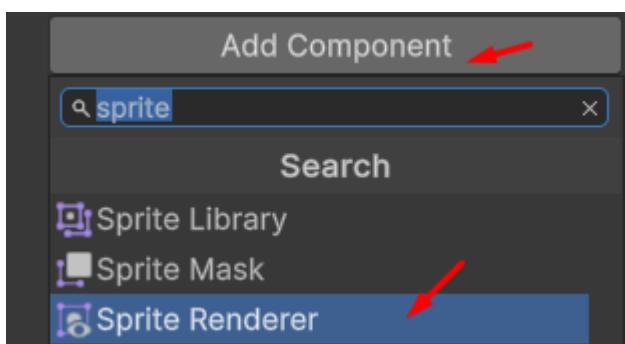
Называем его **StartGameManager**:



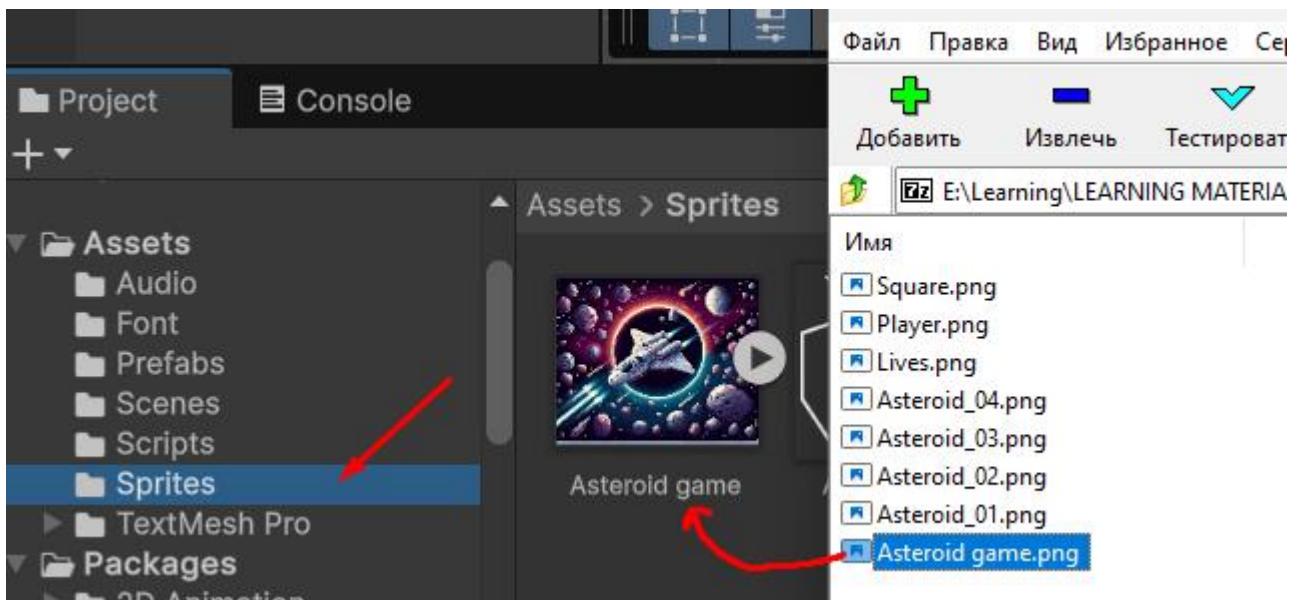
Переносим на него музыку из папки **Audio – ahoami**, включаем **зацикливание**, и изменяем **громкость**:



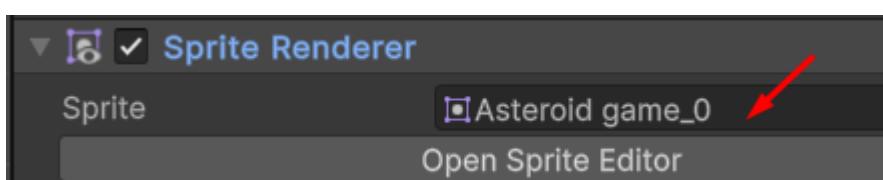
Добавляем компонент **Sprite Renderer**:



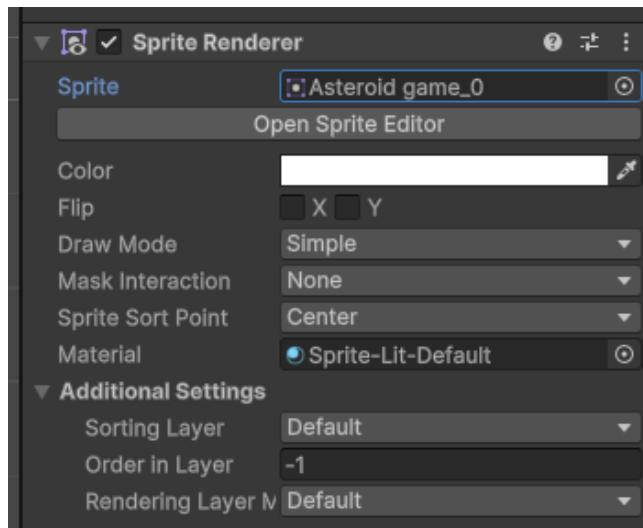
В папку **Sprites** подгружаем из архива фоновую картинку – **Asteroid game**:



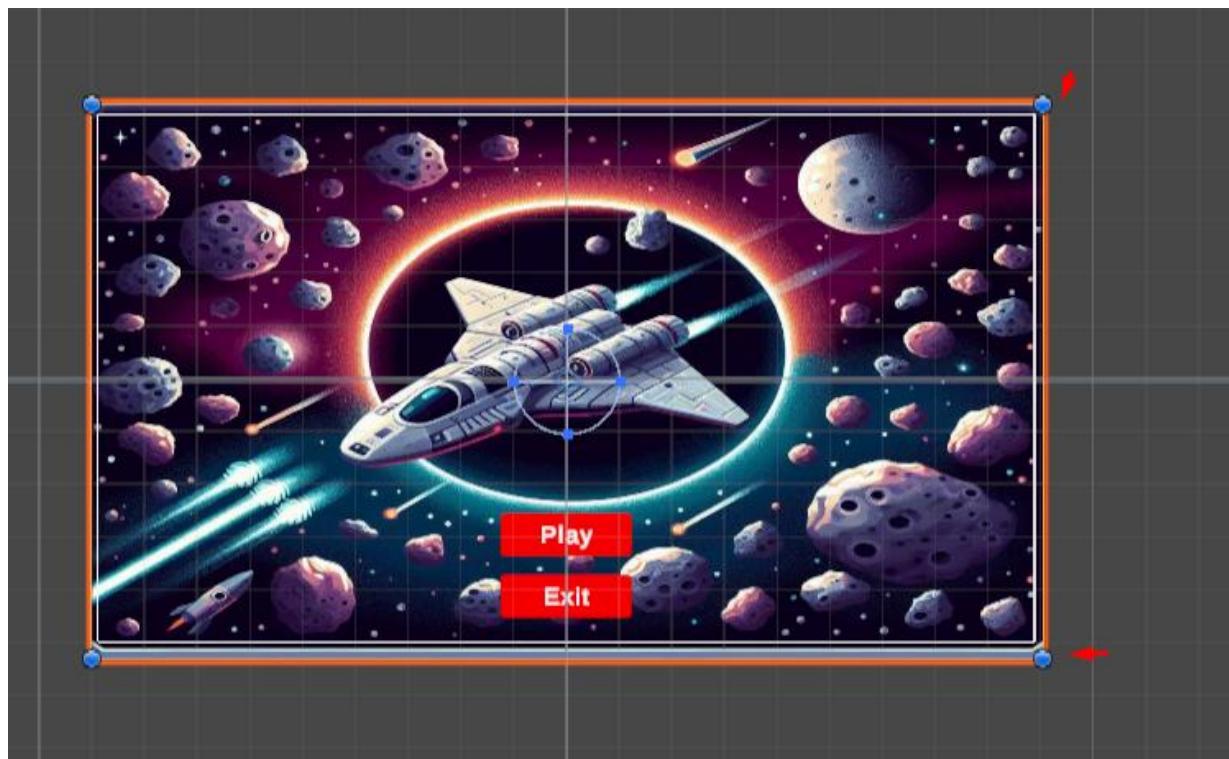
Переносим наше изображение в источник **Sprite**:



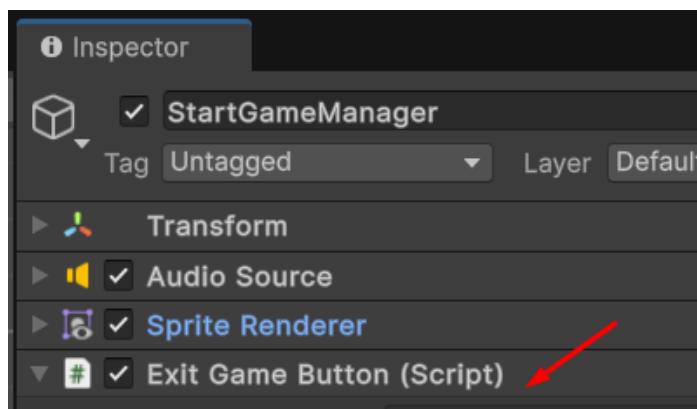
Для того, чтобы оно было позади других элементов в пункте **Order in Layer** поставьте **-1**:



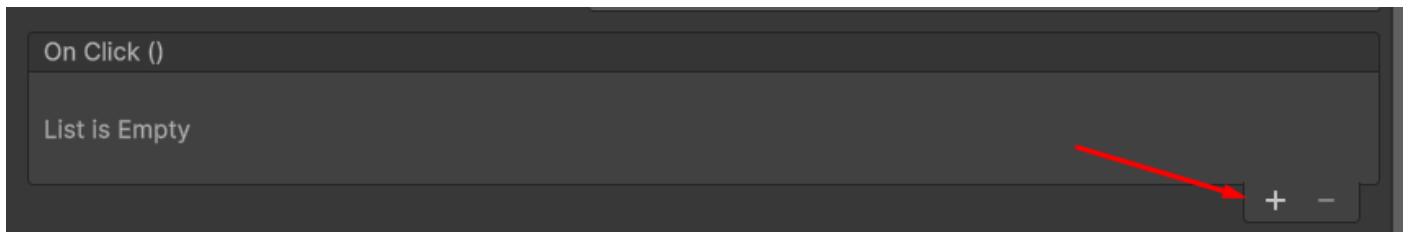
Растяните ваше изображение, чтобы оно заполнило всю сцену:



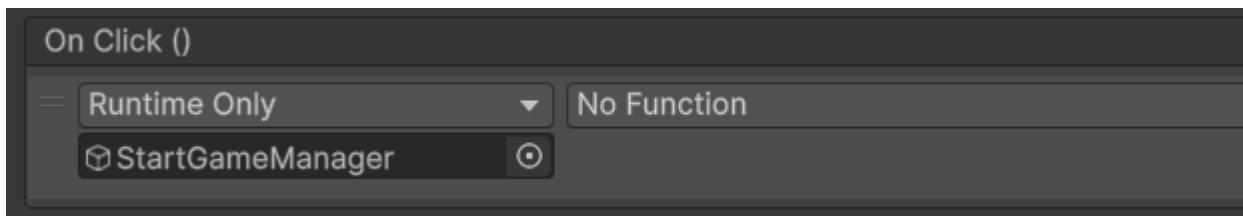
30. Скрипт **ExitGameButton** у нас уже есть. Поэтому по аналогии сделаем что и раньше. Добавляем его на объект **StartGameManager**:



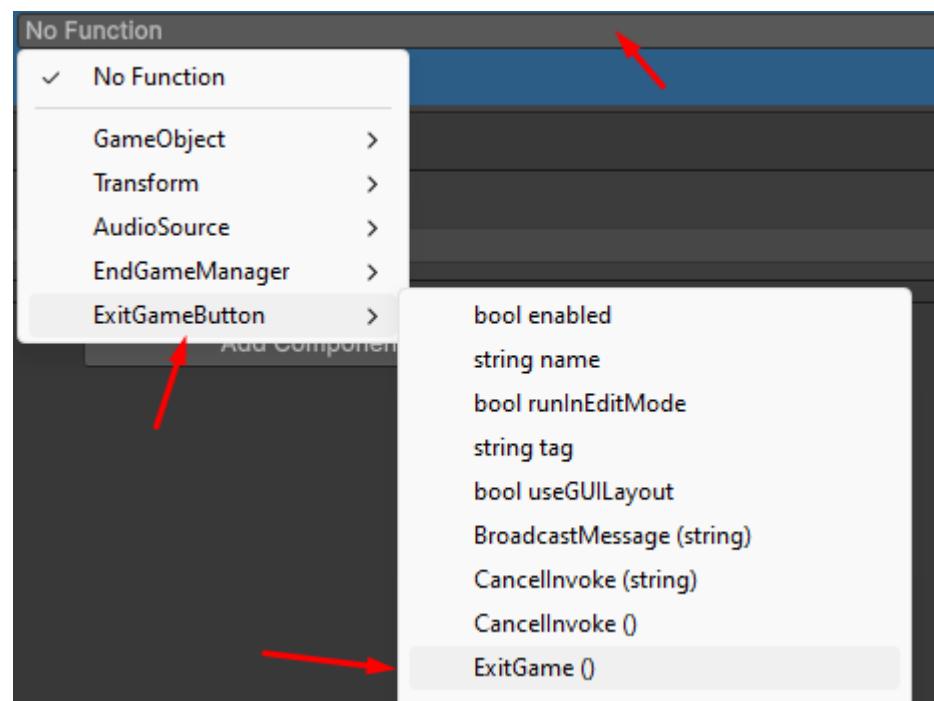
Далее нам нужно в нашу кнопку **ButtonExit** добавить новое событие в **On Click()** нажав на +:



После нам нужно перенести наш **StartGameManager** в пустое поле:



Далее щёлкаем на поле **Function**, и выбираем **ExitGameButton – ExitGame()**:



В папке **Scripts** создаём новый скрипт **StartGameButton**:

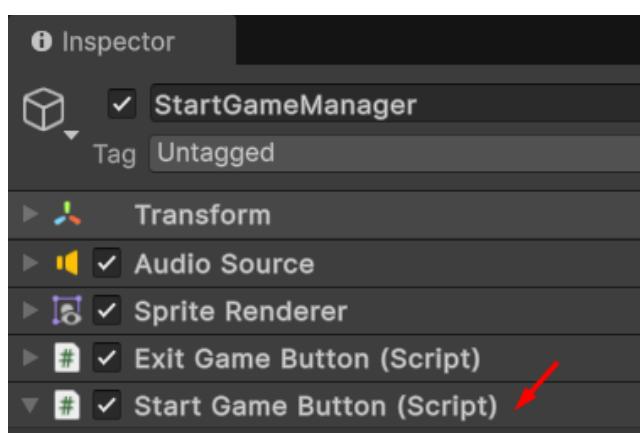


Открываем его и пишем код:

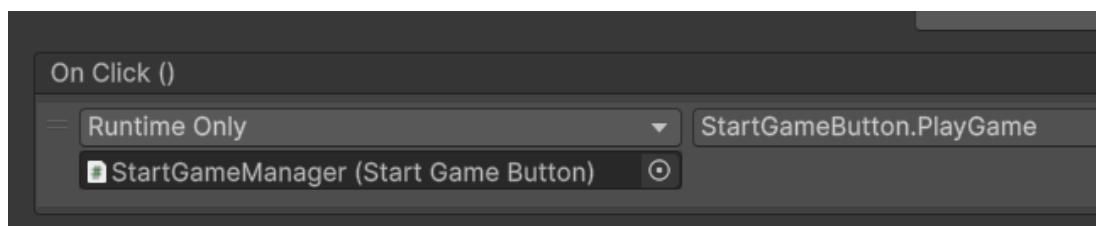
```
using UnityEngine;
using UnityEngine.SceneManagement;

public class StartGameButton : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene("Astroids"); // Загрузка
        основной сцены
    }
}
```

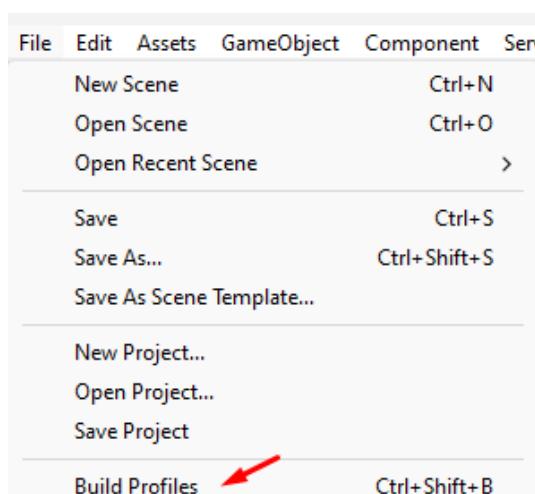
Добавляем скрипт на наш объект **StartGameManager**:



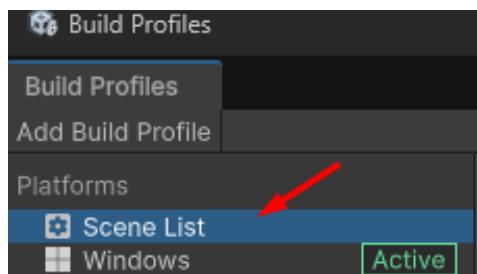
По аналогии добавляем на нашу кнопку **ButtonStart**:



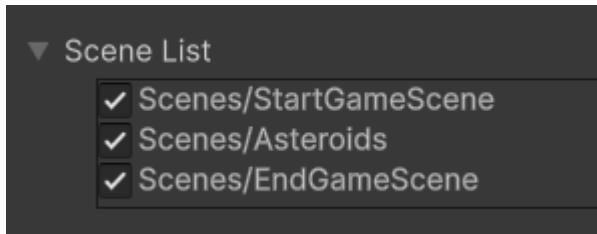
31. Теперь нам нужно подключить все наши сцены. Переходим в **File- Build Profiles**:



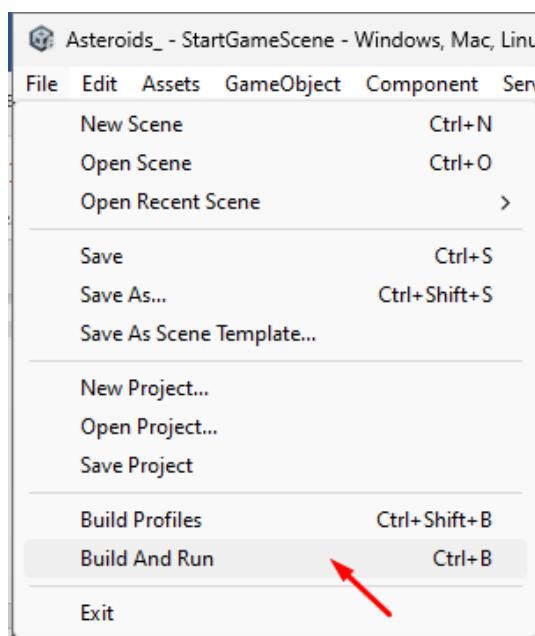
Откройте Scene List:



Перетащите все сцены и настройте их правильный порядок:



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



Выбираете любую папку куда хотите сохранить игру, или создаёте новую папку.

После можете запустить игру через .exe.