

Лабораторная работа. Brick Breaker

Данный урок создан на основе следующего видео-урока: [How to make Brick Breaker 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. Создание глобальной сцены Global для работы с общей логикой

- **GameManager** осуществляет логику основной игры.
- **AudioManager** управляет музыкой в нашей игре.

2. Создание и настройка игровых сцен

В проекте реализованы следующие сцены:

- **Global** – глобальная сцена, управляющая логикой игры.
- **MainMenu** – главное меню с кнопками старта игры.
- **Level1, Level2** – игровые уровни с усложняющимся геймплеем.
- **EndGameScene** – сцена завершения игры с отображением итогового счета и возможностью перезапуска.

3. Создание игровых объектов и управление ими

В проекте настроены следующие игровые объекты:

- **Player (Paddle)** – управляемая платформа, отскакивающая мяч.
- **Ball** – мяч, разрушающий блоки.
- **Bricks** – кирпичи разных цветов с разными значениями очков.
- **Trail** – система частиц для мяча, оставляющая визуальный след.

4. Разработка игровой логики (GameManager)

Основной контроль за игрой выполняет **GameManager** (скрипт GameManager.cs).

Функционал:

- Управление очками и жизнями.
- Отслеживание разрушенных кирпичей, проверка условий победы/поражения.
- Переход между уровнями, а при завершении – загрузка EndGameScene.
- Увеличение скорости мяча с ростом очков.

Также реализована логика конца игры (EndGameManager.cs), обнуляющая очки и жизни при перезапуске.

5. Улучшение механики игры

Добавлены механики для усложнения игры:

- **Увеличение скорости мяча** с ростом счета (Ball.cs).
- **Обновленный Trail-эффект** для мяча, очищающийся при сбросе уровня (TrailEffect.cs).
- **Обработаны столкновения с границами** (WallSound.cs) для аудиоэффектов.

6. Работа с анимациями и эффектами

- **Trail Renderer** используется для отображения следа от мяча, очищается при его сбросе.
- **Animator** применен для анимации изображений.
- Реализована система частиц для сцены окончания игры.

7. Пользовательский интерфейс и аудио

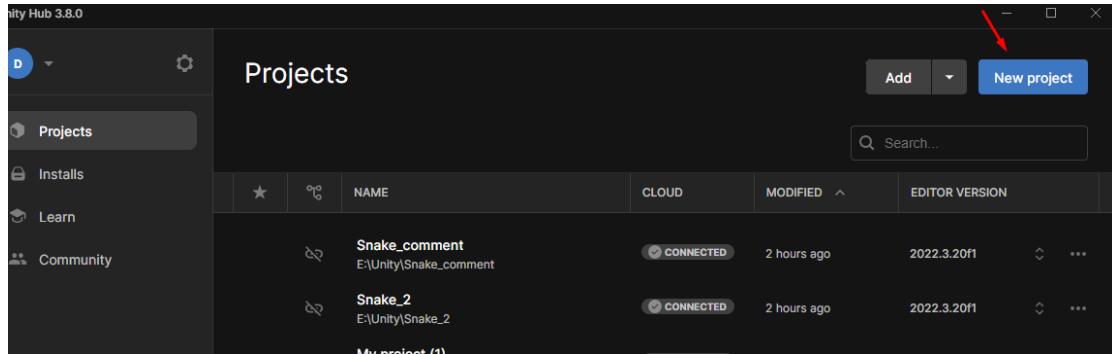
- **Интерфейс отображает жизни и счет** (GameManager.cs).
- **Добавлено звуковое сопровождение** (AudioManager.cs), включая звуки столкновений и разрушений.
- **Реализовано управление с клавиатуры**, включая Пробел для перезапуска уровня (EndGameManager.cs).

8. Финальные настройки

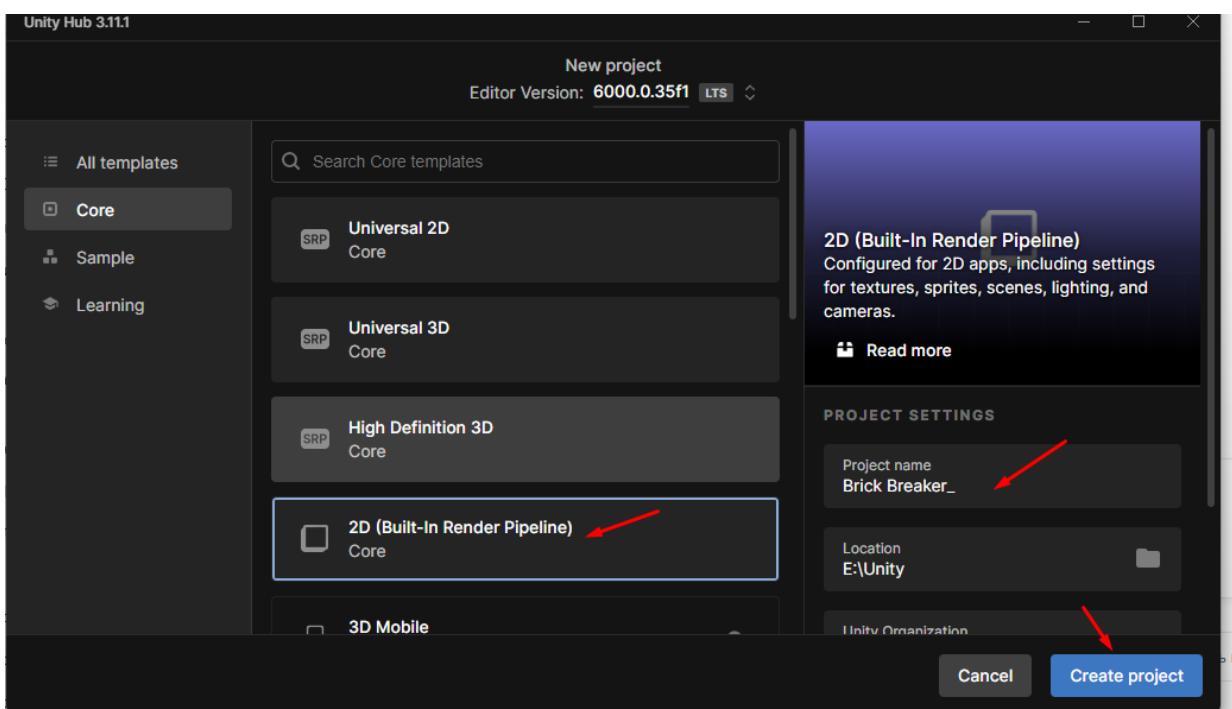
- **Рефакторинг кода**, исправление багов.
- **Оптимизация работы с глобальной сценой**, чтобы GameManager не создавался повторно.
- **Подготовка сборки проекта для Windows** (Build Settings).

1. Запускаем Unity Hub.

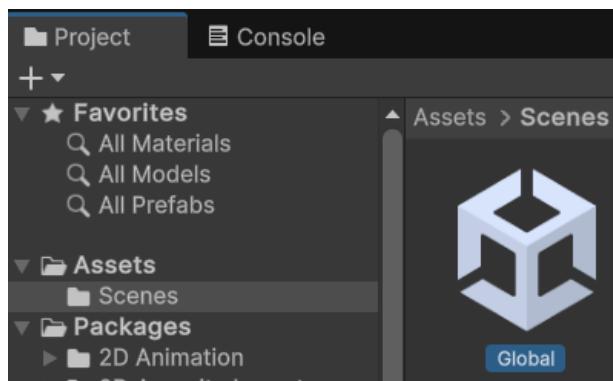
Создаём новый проект – **New project**:



Выбираем **2D(Built-In Render Pipeline)**, вводим название проекта - **Brick Breaker**, выбираем место расположения, и нажимаем **Create project**:



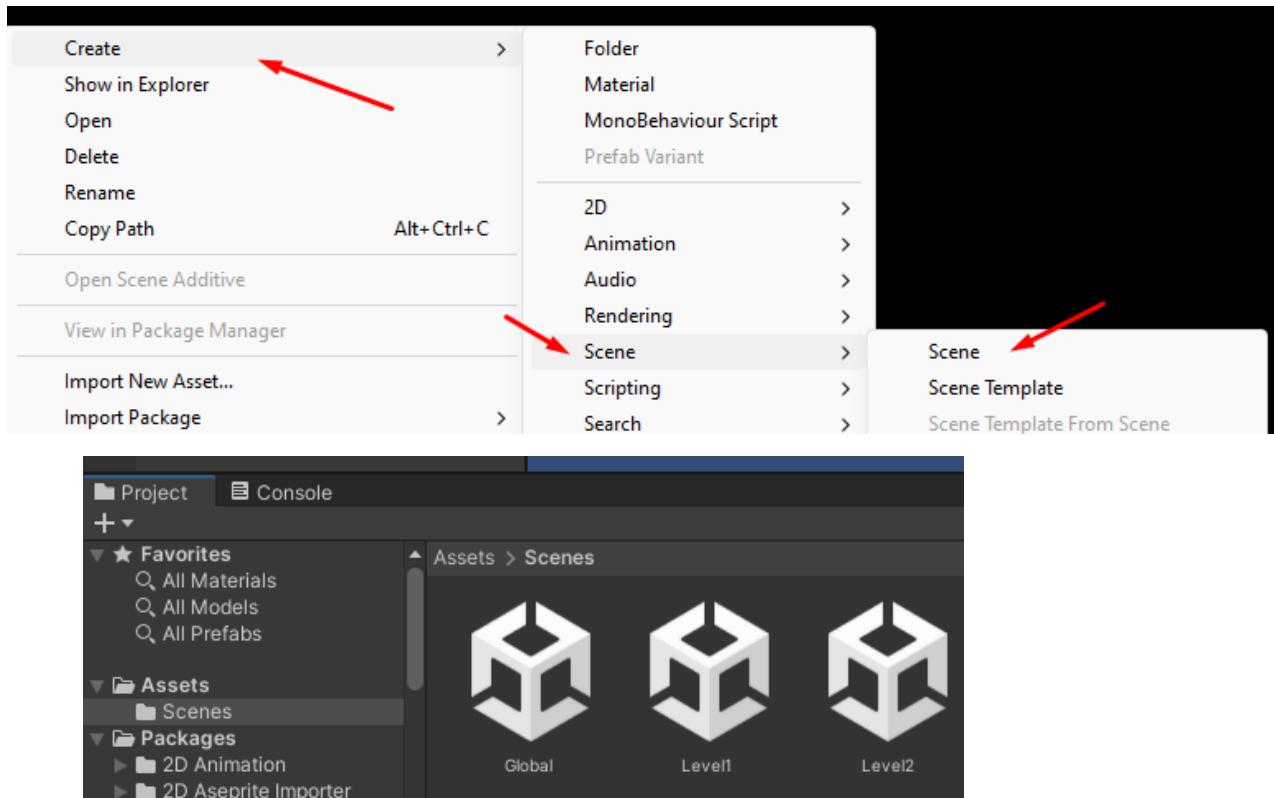
В папке **Scenes** меняем название сцены на **Global**:



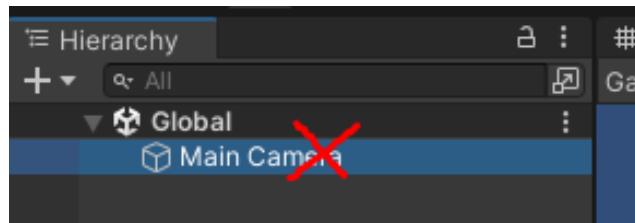
Она будет отвечать за:

- Переходы между уровнями;**
- Сохранение очков, жизней и текущего уровня** при смене сцен.

Создадим сцены для двух уровней – **Level1**, **Level2**:

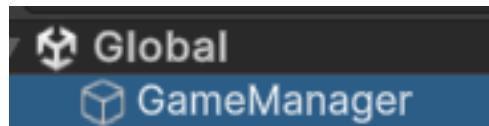


Удалим из сцены **Global** нашу камеру:

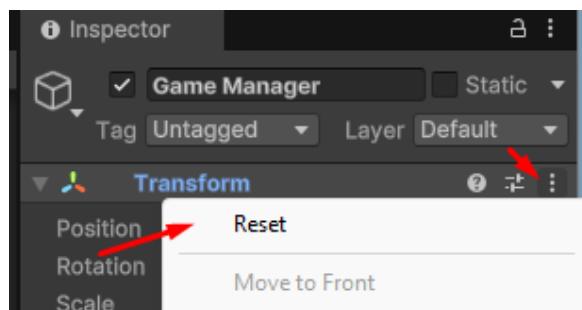


Камера удаляется, потому что каждая игровая сцена (уровень) уже содержит свою камеру. Если оставить камеру в глобальной сцене, то при переходе между уровнями может возникнуть дублирование камер. Это может привести к неожиданным визуальным эффектам или конфликтам.

Добавим новый объект, и назовём его **GameManager**:

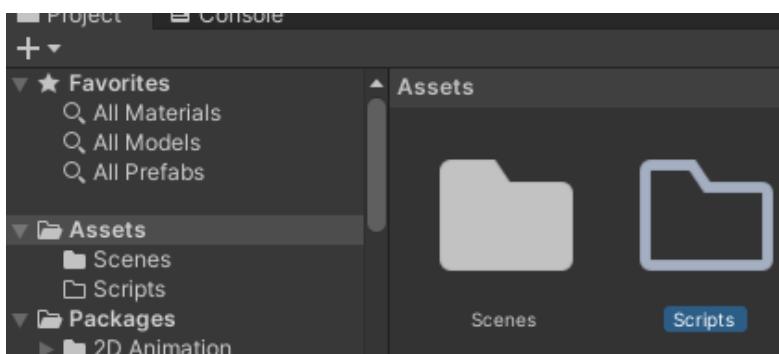


Также сбросим для него трансформацию:

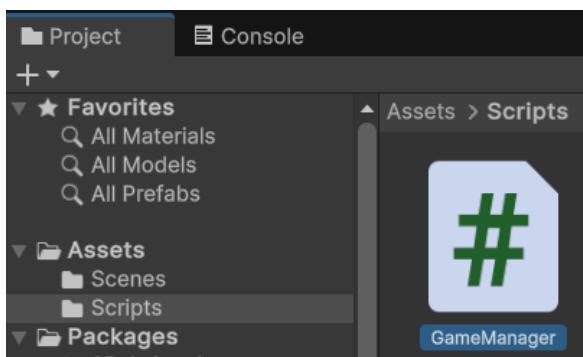


2. Создадим скрипт для переключения между уровнями. Сперва создаём папку

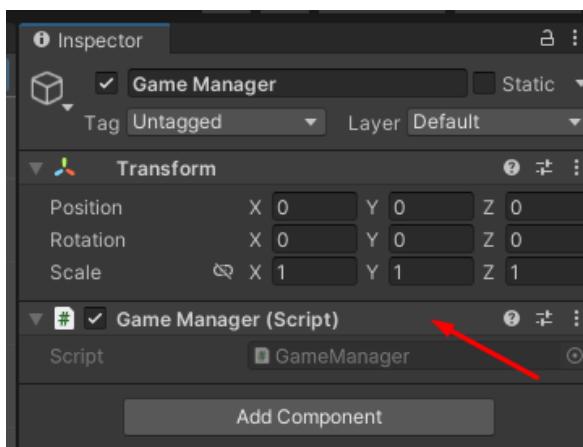
Scripts:



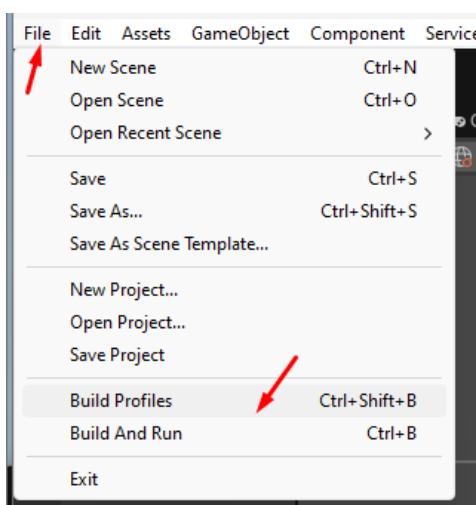
Далее создадим в ней скрипт **GameManager**:



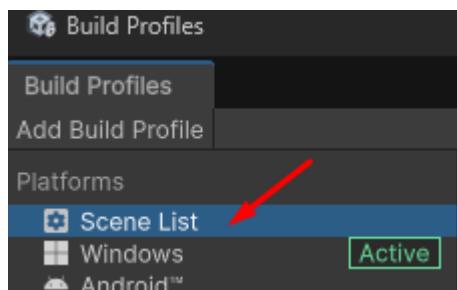
Перетащим скрипт на наш объект **Game Manager**:



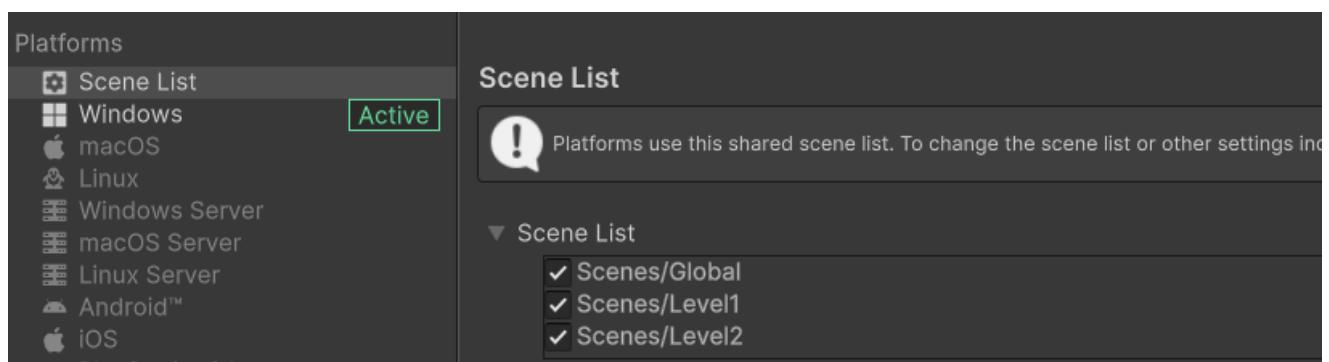
Добавим наши уровни во вкладке **Build Profiles**:



Переходим во вкладку **Scene List**:



Переносим наши уровни в лист соблюдая правильный порядок:



3. Открываем наш скрипт **GameManager**. В этом скрипте мы реализуем менеджер игры, который управляет ключевыми параметрами игры: **уровнем, жизнями и очками**. Он также отвечает за загрузку уровней и запуск новой игры.

Напишем следующий код.

➤ В самом начале скрипта подключим пространство имён **SceneManagement**:

```
using UnityEngine.SceneManagement;
```

➤ Затем внутри скрипта объявляем переменные:

```
[Header("Игровые параметры")]
[SerializeField] private int currentLevel = 1; // Текущий
уровень
[SerializeField] private int score = 0; // Очки игрока
[SerializeField] private int lives = 3; // Количество жизней
```

○ **private int currentLevel = 1;** — переменная, которая **хранит номер текущего уровня**. Мы инициализируем её значением 1, потому что игра начинается с первого уровня.

○ **private int score = 0;** — переменная для хранения **очков игрока**. Они обнуляются в начале игры.

○ **private int lives = 3;** — переменная для **жизней игрока**. По умолчанию у игрока 3 жизни.

➤ Создадим метод **Awake()**:

```
private void Awake()
{
    DontDestroyOnLoad(gameObject); // Предотвращает уничтожение
объекта при смене сцены
}
```

- `DontDestroyOnLoad(gameObject);` — делает так, чтобы объект **GameManager** не уничтожался при загрузке нового уровня.
- Это нужно, чтобы **GameManager** оставался активным на всех сценах и сохранял данные (очки, жизни, текущий уровень).

➤ Создадим метод **LoadLevel()**:

```
private void LoadLevel(int levelNumber)
{
    currentLevel = levelNumber; // Устанавливаем текущий
уровень
    SceneManager.LoadScene("Level" + levelNumber); // Загружаем
сцену соответствующего уровня
}
```

Важно! Названия сцен в Unity должны быть **Level1**, **Level2** и т. д., иначе метод не найдёт их.

➤ Создадим метод **StartNewGame()**:

```
private void StartNewGame()
{
    score = 0; // Сброс очков
    lives = 3; // Сброс жизней
    LoadLevel(1); // Загрузка первого уровня
}
```

➤ В методе **Start()** вызовем его:

```
private void Start()
{
    StartNewGame(); // Запуск новой игры
}
```

Итоговый код:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    [Header("Игровые параметры")]
    [SerializeField] private int currentLevel = 1;
    [SerializeField] private int score = 0;
    [SerializeField] private int lives = 3;

    private void Awake()
    {
        DontDestroyOnLoad(gameObject);
    }
}
```

```

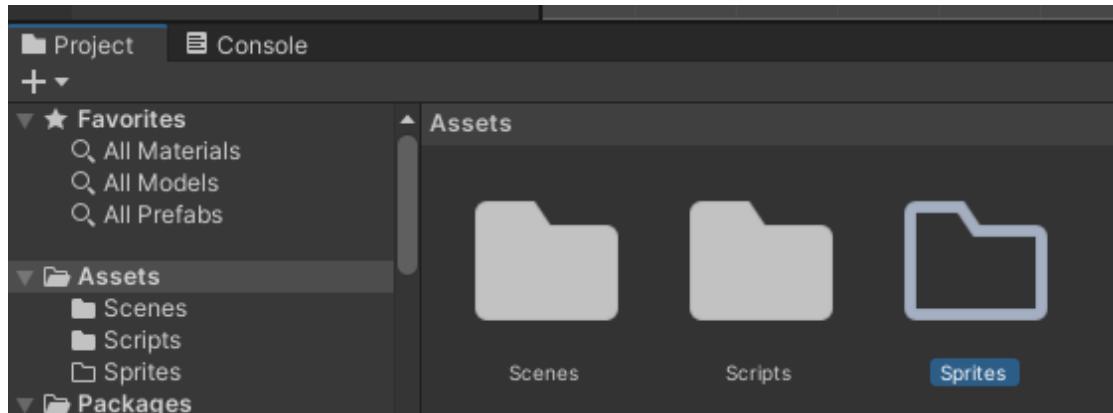
private void Start()
{
    StartNewGame();
}

private void StartNewGame()
{
    score = 0;
    lives = 3;
    LoadLevel(1);
}

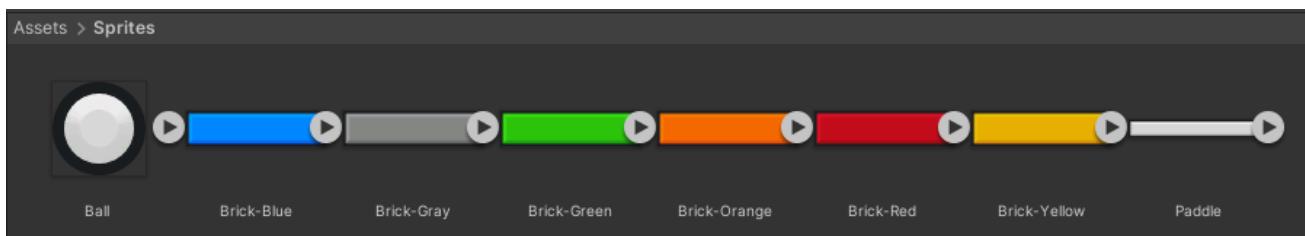
private void LoadLevel(int levelNumber)
{
    currentLevel = levelNumber;
    SceneManager.LoadScene("Level" + levelNumber);
}
}

```

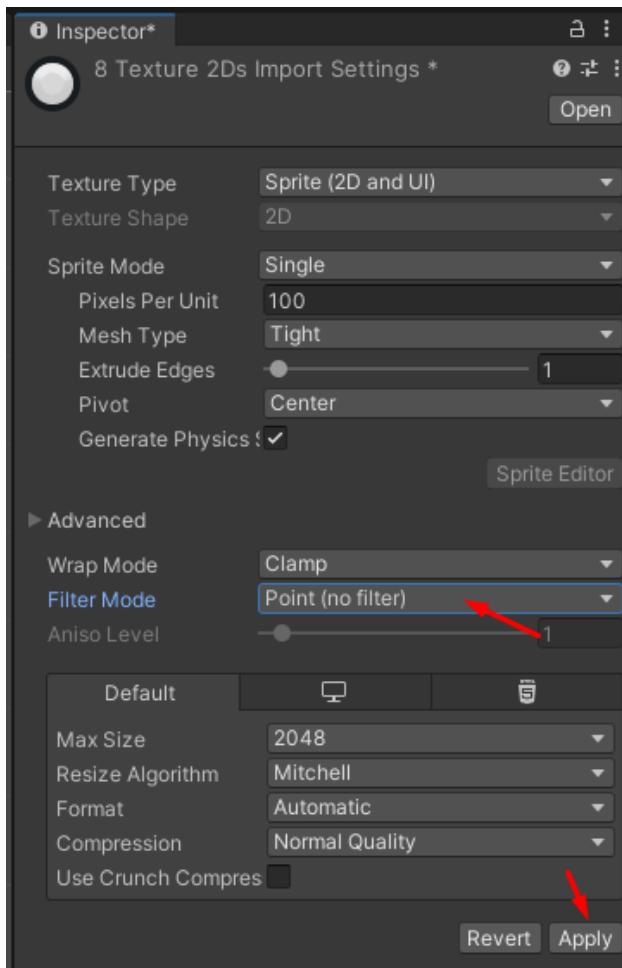
4. Перейдём к наполнению 1 уровня. Создаём папку **Sprites**:



В неё переносим **ассеты** из папки с лабораторной:

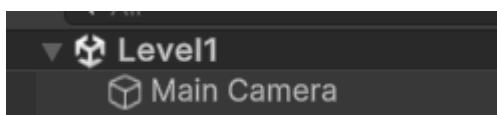


Выделите все спрайты и поменяйте **Filter Mode** на **Point** и нажмите **Apply**:

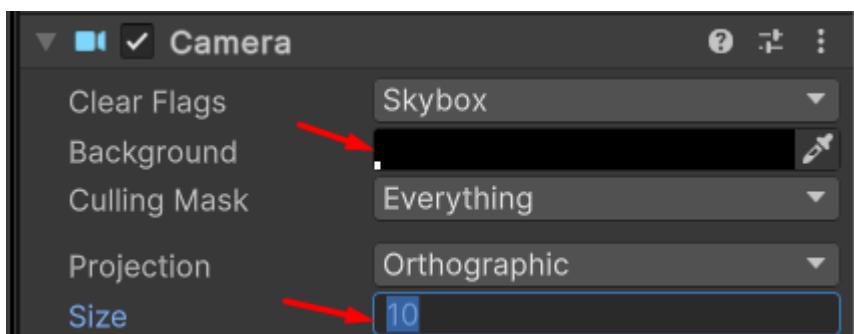


Не забудьте **сохранить** проект!

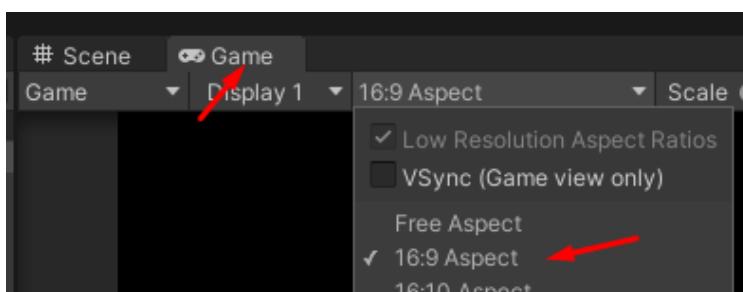
5. Затем переходим в **Level1**:



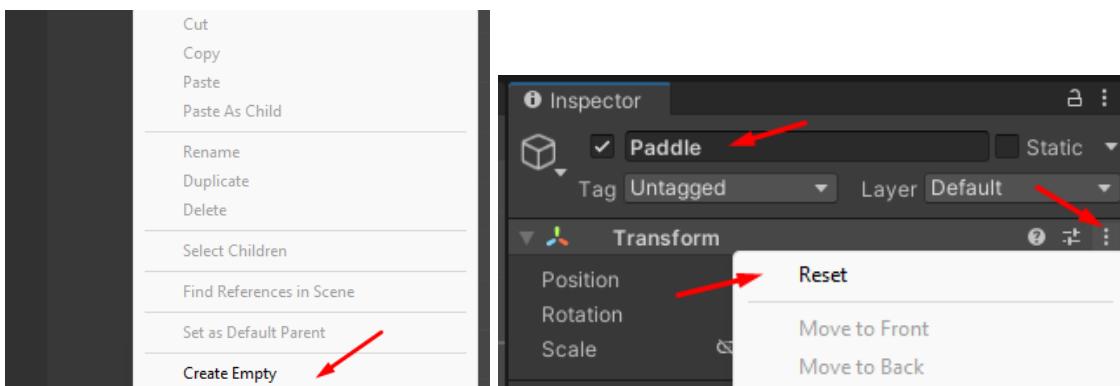
Меняем цвет у камеры на **чёрный** и увеличим размер до **10**:



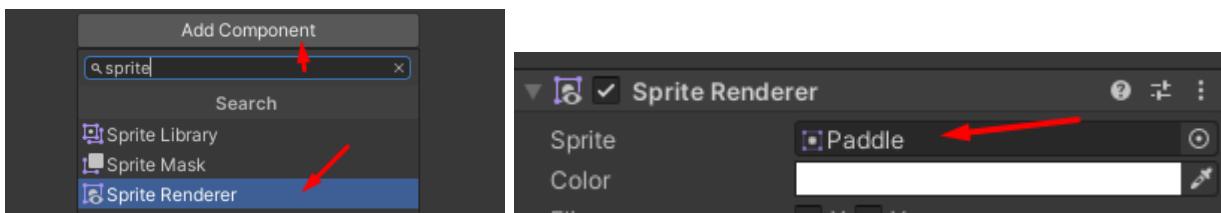
Также поменяем во вкладке **Game - Aspect** игры на **16:9**:



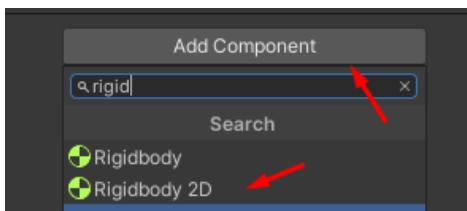
6. Создадим новый объект, назовём его **Paddle**, сбросим трансформацию:



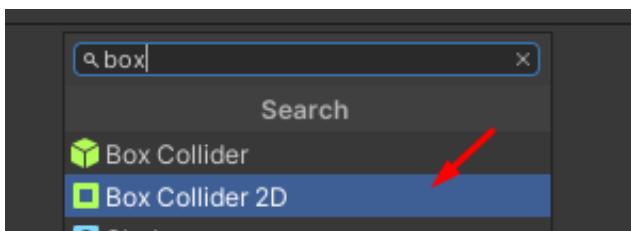
Добавим компоненты – **Sprite Render** и перенесём из спрайтов наш **Paddle**:



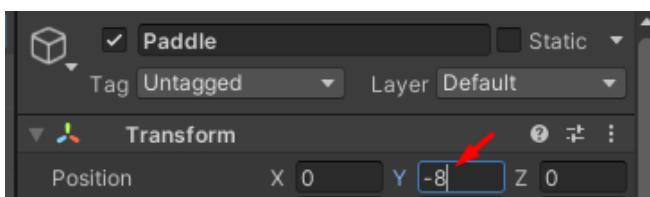
Далее добавим компонент - **Rigidbody 2D**:



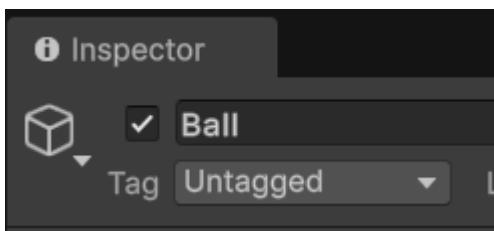
Следующий компонент - **Box Collider 2D**:



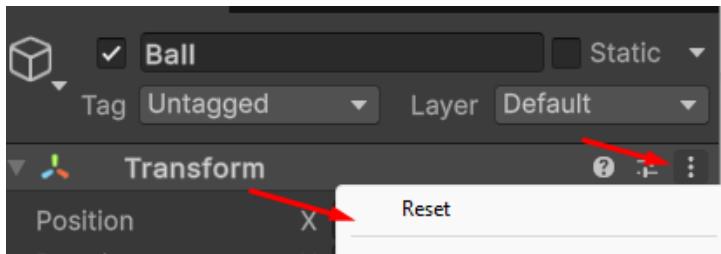
Немного опустим наш **Paddle** (-8 по y):



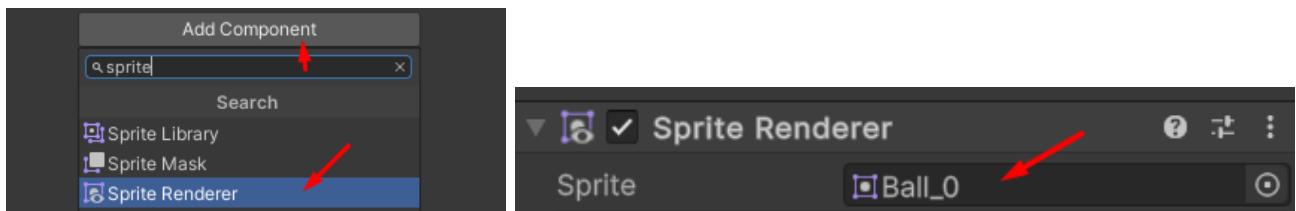
7. Аналогично создаём новый объект, называем его **Ball**:



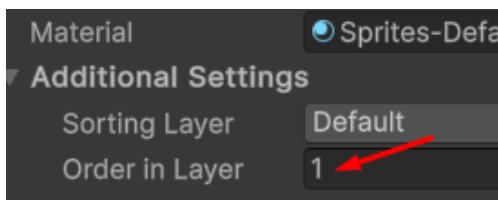
Сбрасываем трансформацию:



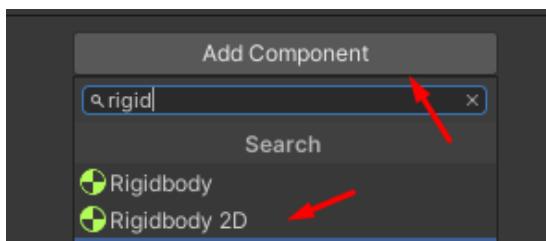
Добавим компоненты – **Sprite Render** и перенесём из спрайтов наш **Ball**:



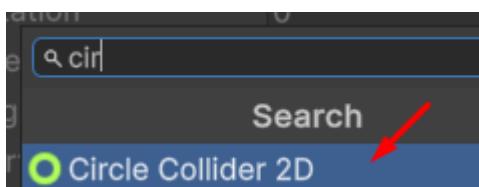
Поставьте **Order in Layer – 1**, чтобы мячик отображался поверх других объектов:



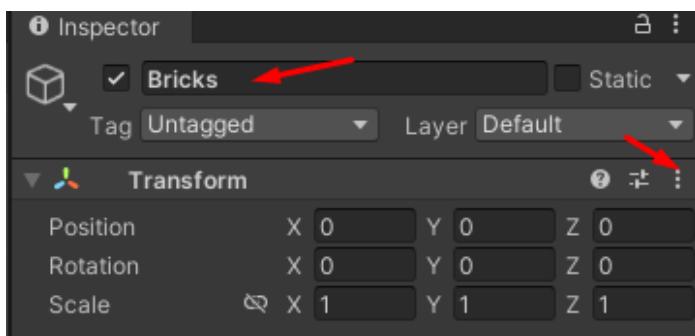
Далее добавим компонент - **Rigidbody 2D**:



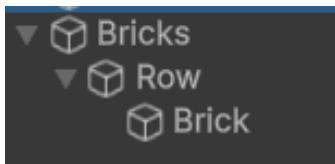
Следующий компонент - **Circle Collider 2D**:



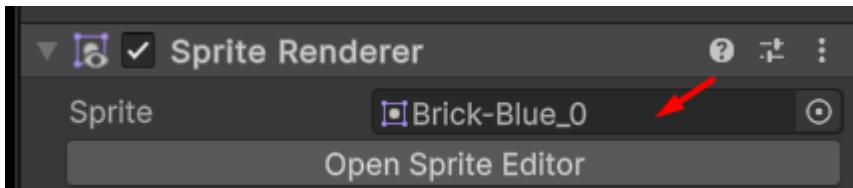
8. Создаём новый объект и называем его **Bricks** и сбрасываем для него трансформацию:



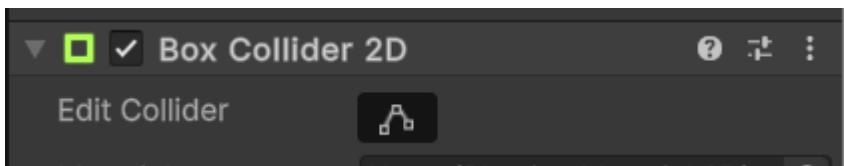
Затем создаём внутри нашего объекта **Row** и внутри него **Brick**:



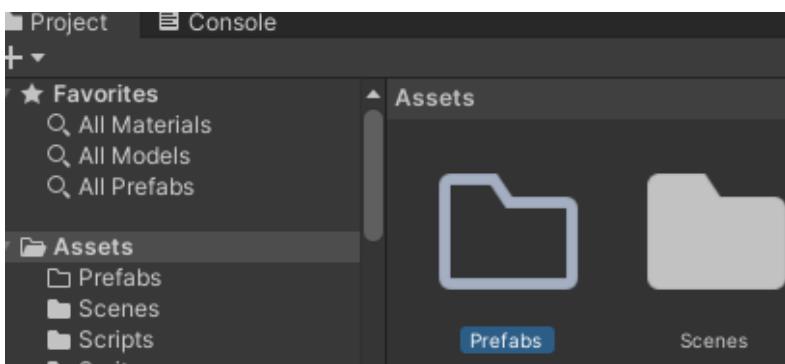
Для **Brick** добавляем **Sprite Renderer** (помещаем наш любой спрайт кирпича, к примеру **Brick-Blue** в **Sprite**)



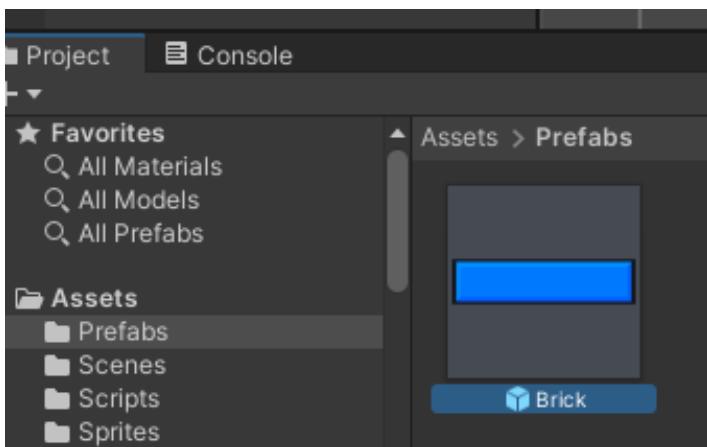
Добавляем **Box Collider 2D**:



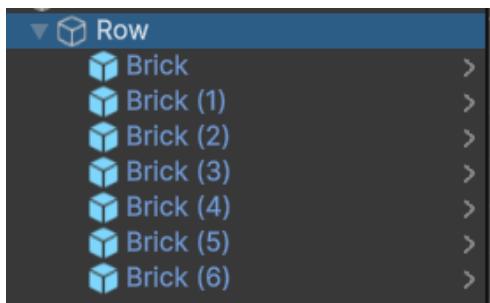
9. Создаём папку **Prefabs**:



Переносим **Brick** в нашу папку с префабами:



Дальше ваша задача создать несколько копий префаба и составить первую линию, можете расставить кол-во и расстояние между ними как вам нравится:

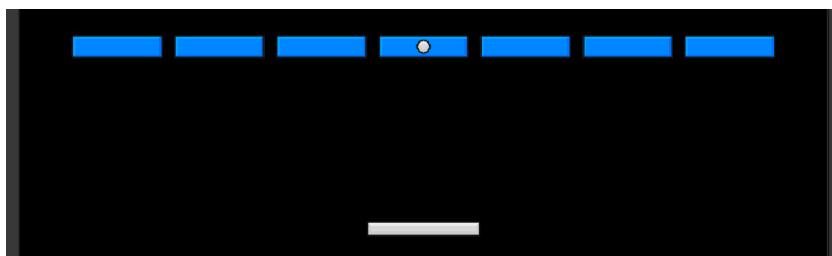


Для примера я поставил первый кирпичик по у позицию в **-13.5**:

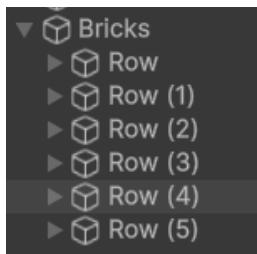


И затем двигался по **-4.5** по оси у для каждого кирпичика (т.е. получились значения **-9** для второго, **-4.5** для третьего, **0** для четвертого, **4.5** для пятого, **9** для шестого, **13.5** для седьмого).

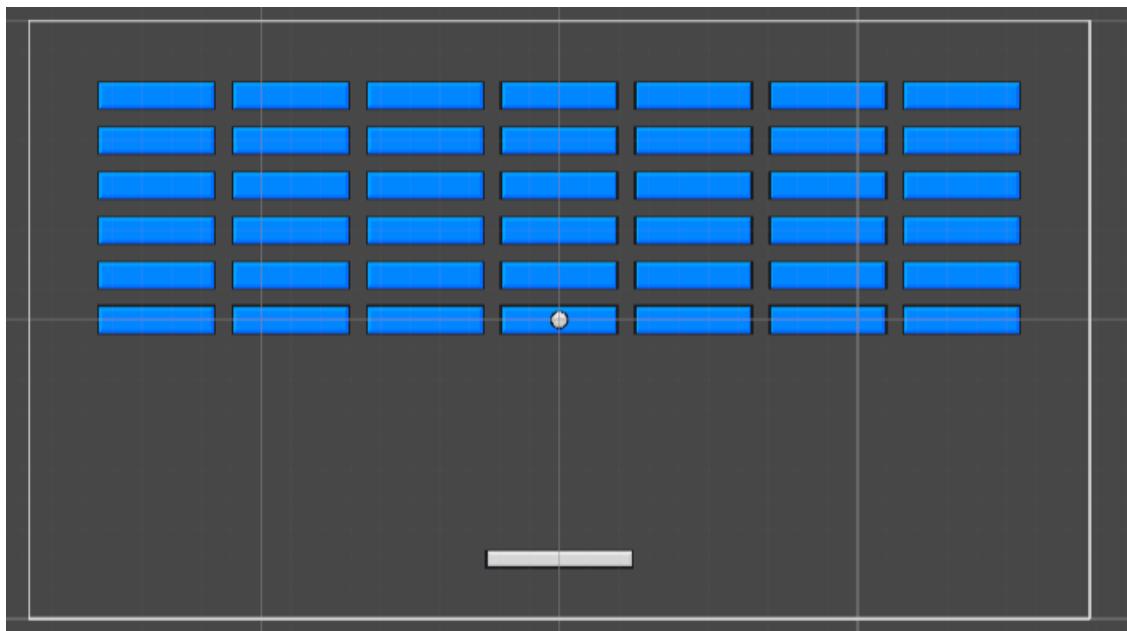
Итог:



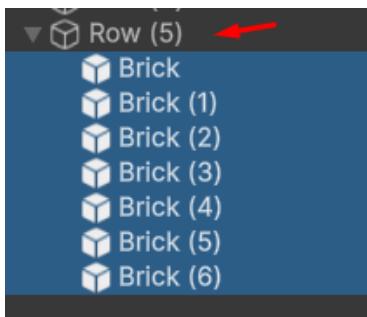
Далее можете клонировать **Row** и расставить по сцене ваши **Brick**.



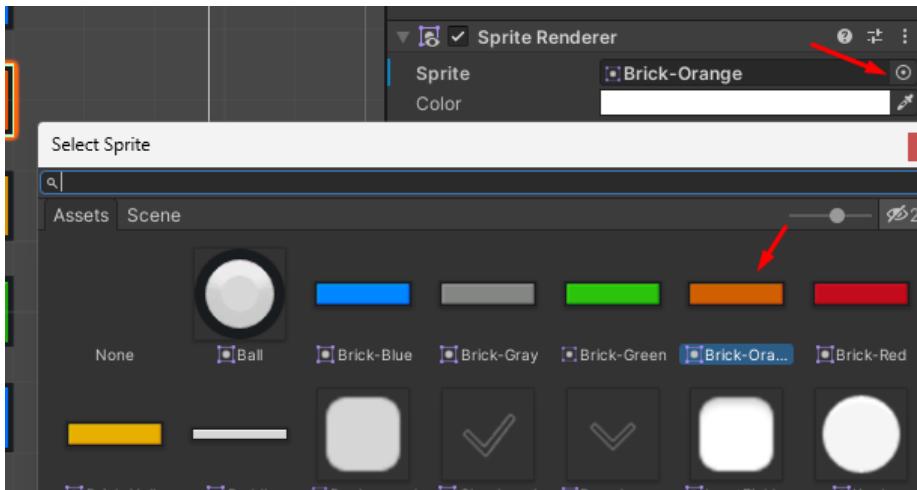
Итог:



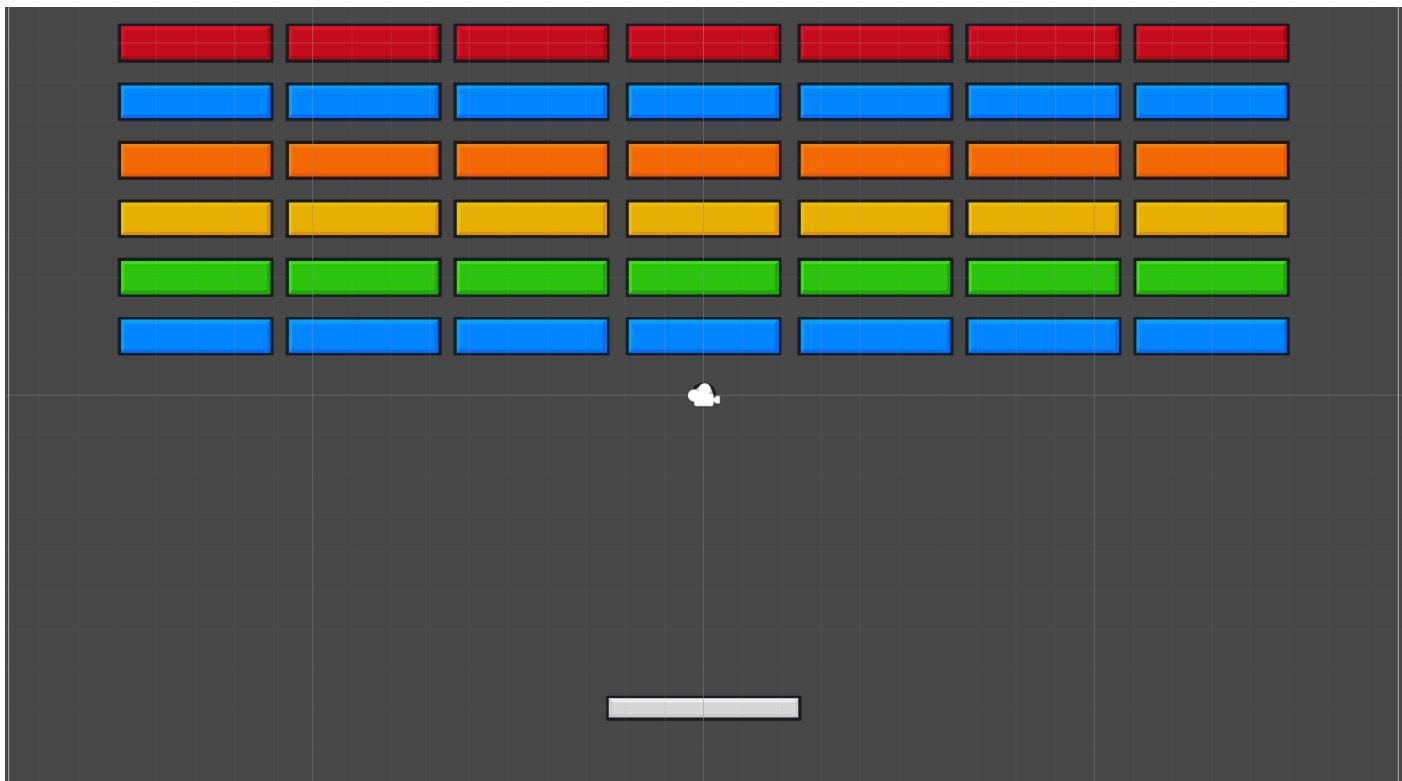
Далее вам нужно поменять цвет для разных рядов, для этого выделяете нужные:



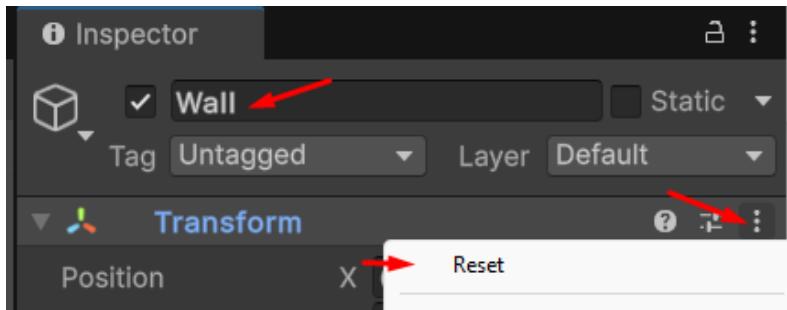
И затем меняете на нужный **Sprite**:



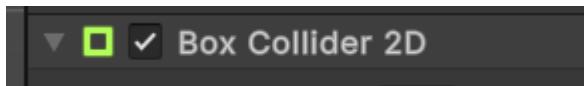
Пример того, что должно получиться:



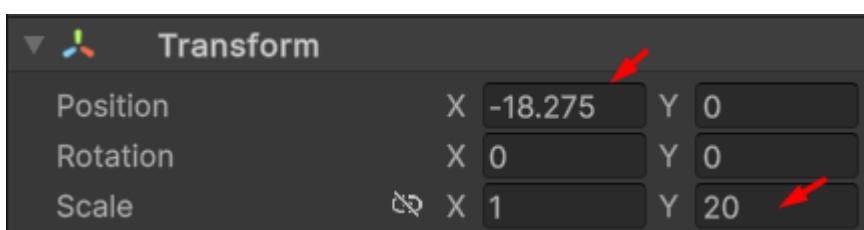
10. Займёмся созданием стен. Добавляем пустой объект и называем его **Wall** и сбрасываем трансформацию:



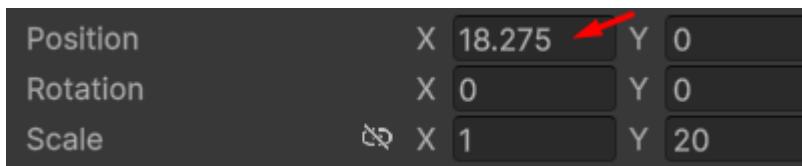
К ней добавляем **Box Collider 2D**:



Первая стена пусть у нас будет левой. Поэтому сдвинем её позицию по оси **X** на **-18.275** и размер по **Y** увеличим на **20**:



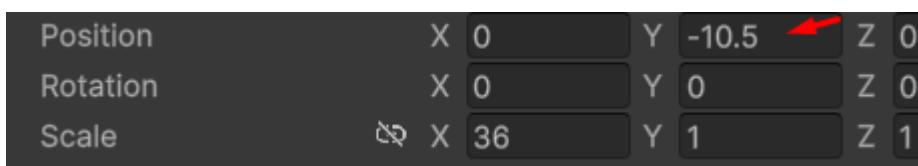
Копируем её и меняем позицию по **X**:



Также копируем и настроим для **верхней** стены:



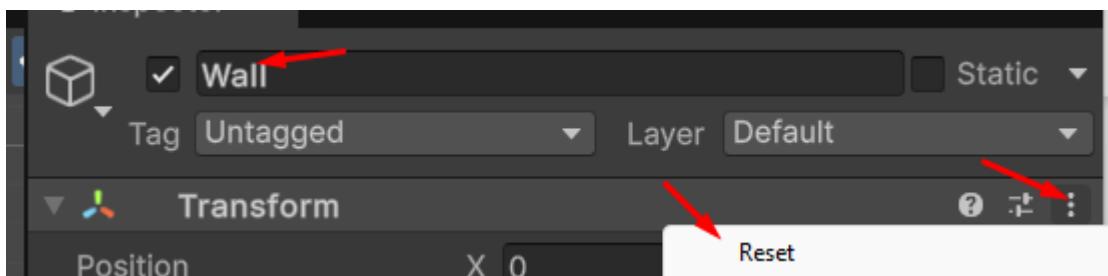
Копируем и настраиваем **нижнюю** стену:



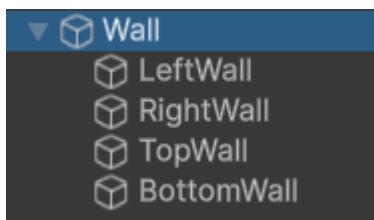
Переименовываем их на **LeftWall**, **RightWall**, **TopWall**, **BottomWall**:



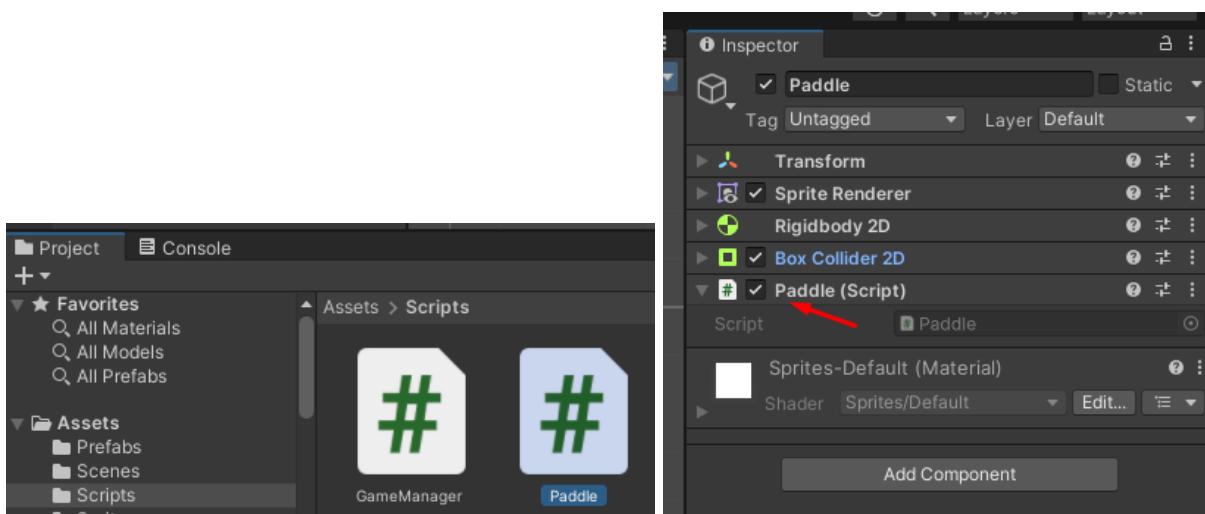
Создадим пустой объект, назовём его **Wall** и сбросим трансформацию:



Перенесём туда все наши стены:



11. Создадим новый скрипт **Paddle** и переместим его на наш **Paddle**:



Открываем его и пишем.

➤ Объявляем переменные:

```
[Header("Параметры ракетки")]
[SerializeField] private float speed = 30f; // Скорость движения

[Header("Компоненты")]
private Rigidbody2D _rigidbody; // Физическое тело ракетки
private Vector2 _direction; // Направление движения
```

➤ Получаем ссылку на компонент **Rigidbody2D** в методе **Awake()**:

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

➤ Напишем метод для управления клавишами **HandleInput()**:

```

private void HandleInput()
{
    if (Input.GetKey(KeyCode.A) ||
Input.GetKey(KeyCode.LeftArrow))
        _direction = Vector2.left;

    else if (Input.GetKey(KeyCode.D) ||
Input.GetKey(KeyCode.RightArrow))
        _direction = Vector2.right;

    else
        _direction = Vector2.zero;
}

```

➤ Вызовем его в **Update()**:

```

private void Update()
{
    HandleInput(); // Обрабатываем нажатие клавиш
}

```

➤ Реализуем движения ракеткой в методе **FixedUpdate()**

Напомню, что есть 3 простых способа реализовать движение:

1 способ. Самый простой без физического движка:

```
_rigidbody.linearVelocity = _direction * speed;
```

2 способ. С использованием физического движка:

```
_rigidbody.AddForce(_direction * speed);
```

3 способ. Использующий интерполяцию для большей плавности:

```
_rigidbody.linearVelocity = Vector2.Lerp(_rigidbody.linearVelocity,
_direction * speed, Time.fixedDeltaTime * 10f);
```

И использую 3 способ:

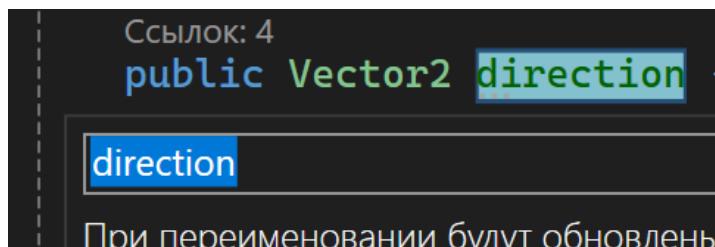
```

private void FixedUpdate()
{
    // Устанавливаем скорость
    _rigidbody.linearVelocity =
Vector2.Lerp(_rigidbody.linearVelocity, _direction * speed,
Time.fixedDeltaTime * 10f);
}

```

Ну и можно ввести новое понятие про **геттеры и сеттеры (properties)** — это мощный инструмент, и их использование может сделать код более **безопасным и удобным**.

Нам надо с вами изменить переменные `_rigidbody` и `_direction` (чтобы убрать нижнее подчёркивание, напомню, что для изменения по всему кода нужно нажать **Ctrl+R+R** и ввести новое имя):



```
public new Rigidbody2D rigidbody { get; private set; }  
// Геттер позволяет только читать  
public Vector2 direction { get; private set; }  
// Геттер позволяет только читать
```

Геттер (`get`) позволяет читать переменную, а сеттер (`set`) позволяет изменять её.

Если `set private`, то менять значение можно только внутри класса, а другие скрипты могут только читать.

- ◆ Что изменилось с геттерами и сеттерами?

Переменные `rigidbody` и `direction` теперь свойства (properties).

`get; private set;` означает:

- ◆ Другие скрипты могут читать `rigidbody` и `direction`.
- ◆ Но менять их может только сам `Paddle`.

Больше нет `_rigidbody` и `_direction`, потому что теперь они не поля, а свойства.

Итоговый код:

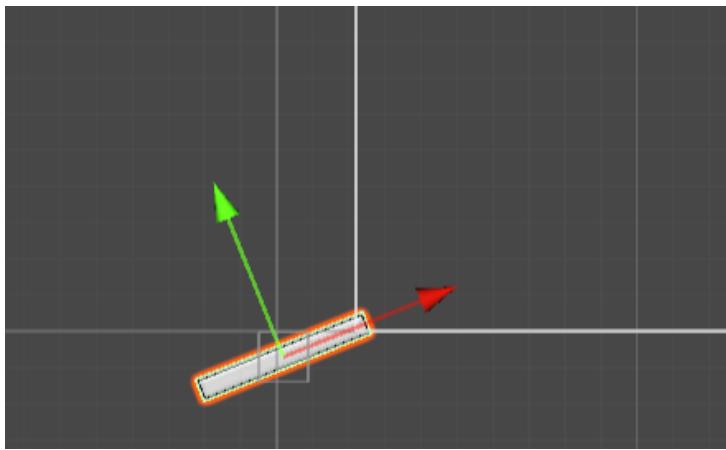
```
using UnityEngine;  
  
public class Paddle : MonoBehaviour  
{  
    [Header("Параметры ракетки")]  
    [SerializeField] private float speed = 30f;  
  
    [Header("Компоненты")]  
    public new Rigidbody2D rigidbody { get; private set; }  
    public Vector2 direction { get; private set; }  
  
    private void Awake()  
    {  
        rigidbody = GetComponent<Rigidbody2D>();  
    }
```

```
private void Update()
{
    HandleInput();
}

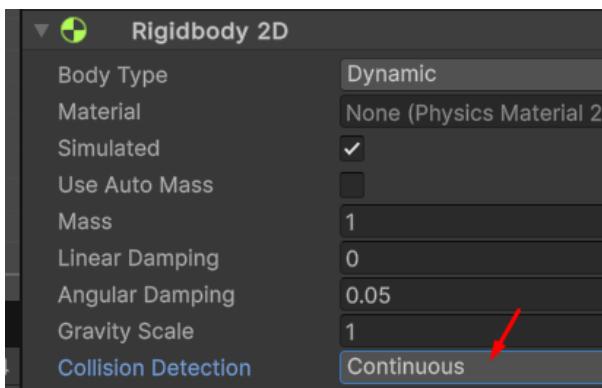
private void FixedUpdate()
{
    rigidbody.linearVelocity =
Vector2.Lerp(rigidbody.linearVelocity, direction * speed,
Time.fixedDeltaTime * 10f);
}

private void HandleInput()
{
    if (Input.GetKey(KeyCode.A) ||
Input.GetKey(KeyCode.LeftArrow))
    {
        direction = Vector2.left;
    }
    else if (Input.GetKey(KeyCode.D) ||
Input.GetKey(KeyCode.RightArrow))
    {
        direction = Vector2.right;
    }
    else
    {
        direction = Vector2.zero;
    }
}
```

Если мы запустим игру, выставим слишком большую **скорость** и будем двигать нашу ракету **влево** или **вправо**, то можем поймать такой баг - **вылетающей за пределы сцены ракетки**:



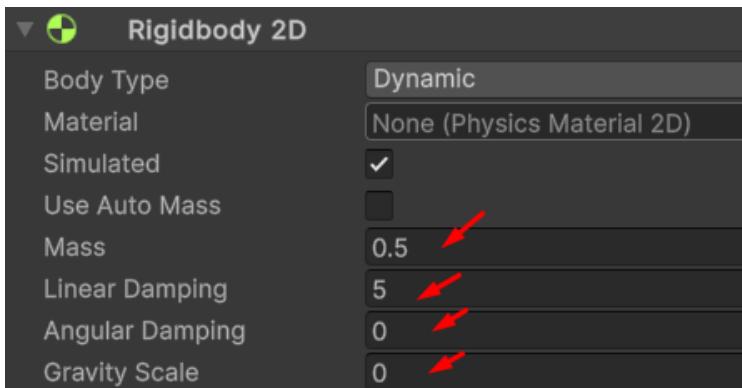
Меняем режим **Continuous** для коллизий:



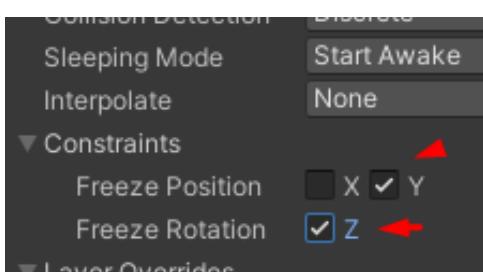
По умолчанию используется режим **Discrete**, который не всегда точен для быстрых объектов.

Изменение его на **Continuous** позволит Unity учитывать движение мяча между кадрами и предотвращать пропуск коллизий.

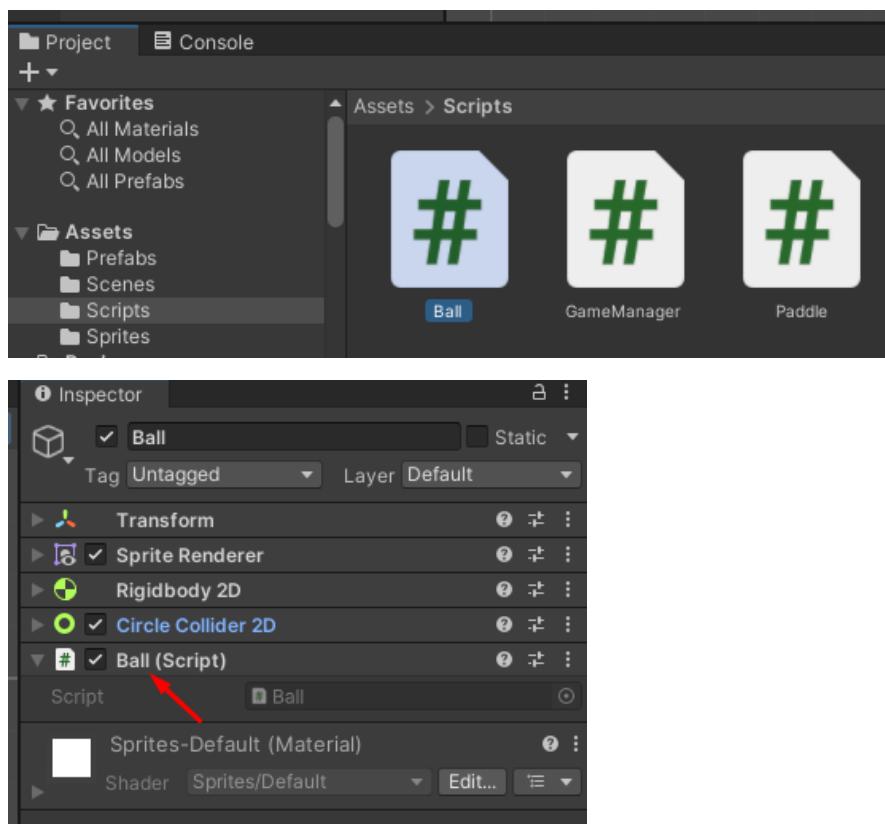
В **Rigidbody 2D** измените для нашего **Paddle** массу, гравитацию и углы:



Также в **Rigidbody 2D** заморозьте позицию по **Y** и вращение по **Z**:



12. Займёмся мячом. Создаём новый скрипт **Ball** и переносим его на наш **Ball**:



Открываем его и пишем.

➤ **Объявляем переменные:**

```
[Header("Компоненты")]
public new Rigidbody2D rigidbody { get; private set; }
// публичное свойства rigidbody типа Rigidbody2D
[Header("Параметры мяча")]
[SerializeField] private float speed = 500f;
// скорость движения мяча
```

➤ Получаем ссылку на компонент **Rigidbody2D** в методе **Awake()**:

```
private void Awake()
{
    rigidbody = GetComponent<Rigidbody2D>();
```

➤ Напишем метод **SetRandomTrajectory()** для генерации случайного направления:

```
private void SetRandomTrajectory()
{
    Vector2 force = new Vector2(Random.Range(-1f, 1f),
    Random.value < 0.5f ? -1f : 1f); // Может лететь вверх или вниз
    rigidbody.linearVelocity = Vector2.zero; // Обнуляем
    предыдущую скорость
    rigidbody.AddForce(force.normalized * speed);
}
```

◆ Генерируем случайную траекторию:

- **Random.Range(-1f, 1f)** — случайное значение **по оси X** (мяч может лететь влево или вправо).
 - **Random.value < 0.5f ? -1f : 1f** — случайное направление **по оси Y**:
 - **50% шанс полететь вверх (1f)**
 - **50% шанс полететь вниз (-1f)**
 - ◆ **Обнуляем скорость перед запуском:**
 - Это **предотвращает накопление скорости**, если мяч уже двигался.
 - ◆ **Применяем силу к мячу:**
 - **.normalized** делает вектор единичной длины, чтобы скорость была одинаковой.
 - **AddForce()** добавляет силу, и мяч начинает двигаться.
- Вызываем его в методе **Start()**:

```
private void Start()
{
    Invoke(nameof(SetRandomTrajectory), 1f);
}
```

- **Invoke()** вызывает метод **с задержкой**.
- **SetRandomTrajectory()** запустится **через 1 секунду** после старта.
- Это нужно, чтобы у игрока было время подготовиться перед началом игры.

Итоговый код:

```
using UnityEngine;

public class Ball : MonoBehaviour
{
    [Header("Компоненты")]
    public new Rigidbody2D rigidbody { get; private set; }
    [Header("Параметры мяча")]
    [SerializeField] private float speed = 500f;
    private void Awake()
    {
        rigidbody = GetComponent<Rigidbody2D>();
    }

    private void Start()
    {
        Invoke(nameof(SetRandomTrajectory), 1f);
    }

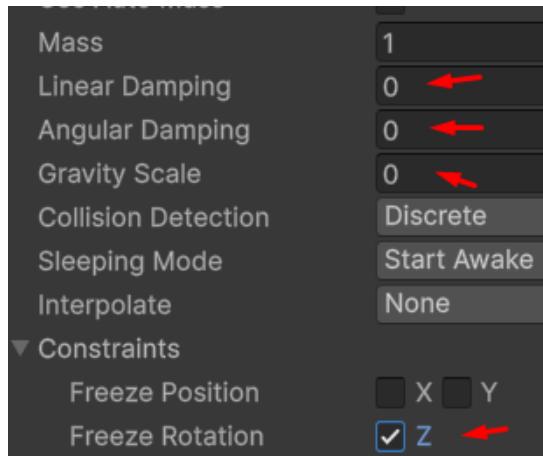
    private void SetRandomTrajectory()
    {
```

```

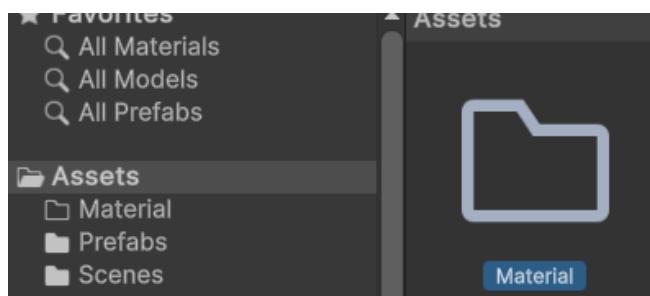
        Vector2 force = new Vector2(Random.Range(-1f, 1f),
Random.value < 0.5f ? -1f : 1f);
        rigidbody.linearVelocity = Vector2.zero;
        rigidbody.AddForce(force.normalized * speed);
    }
}

```

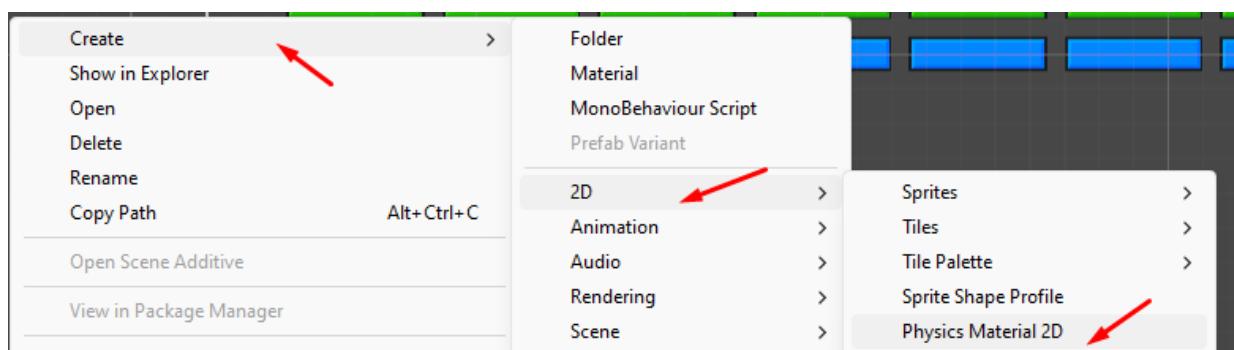
В Rigidbody 2D измените для Ball гравитацию и углы и заморозьте вращение по Z:



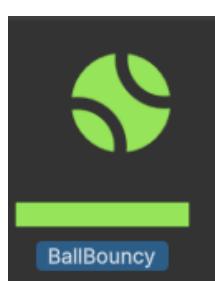
Создаём новую папку Material:



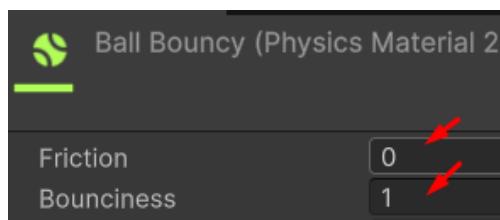
В ней создаём Create – 2D – Physics Material 2D:



Называем его BallBouncy:

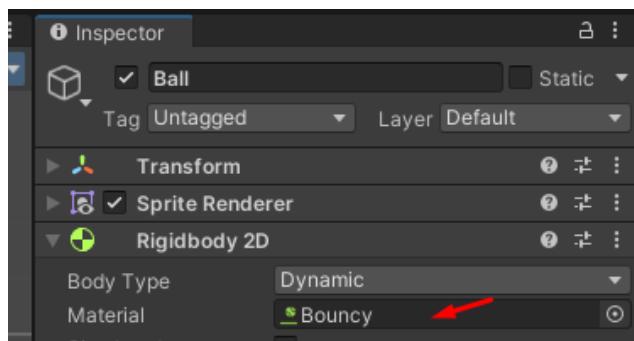


Меняем Friction на 0 и Bounciness на 1:

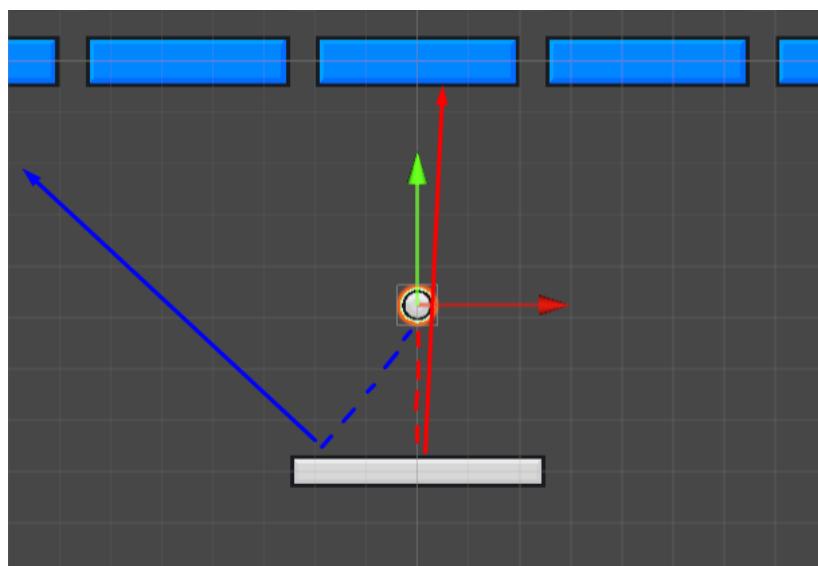


- **Friction = 0:** Объект будет скользить без трения, то есть не будет замедляться при контакте с другими объектами.
- **Bounciness = 1:** Объект будет идеально упругим, то есть полностью сохранит энергию при столкновении и отскочит без потери скорости.

Затем перенесём в наш объект **Ball** созданный материал:



13. Разнообразим нашу игру и сделаем возможность для мяча отскакивать в разные стороны, в зависимости от места удара на нашей ракетке:



Открываем скрипт **Paddle**.

➤ Добавим в самое начало новую переменную угла наклона нашего мячика:

```
[Header("Параметры ракетки")]
[SerializeField] private float speed = 30f;
[SerializeField] private float maxBounceAngle = 75f; // максимальный угол отклонения мяча после удара о ракетку
```

- Затем создадим метод **BallBounce()**, который будет отвечать за изменение угла отскока мяча при столкновении с ракеткой:

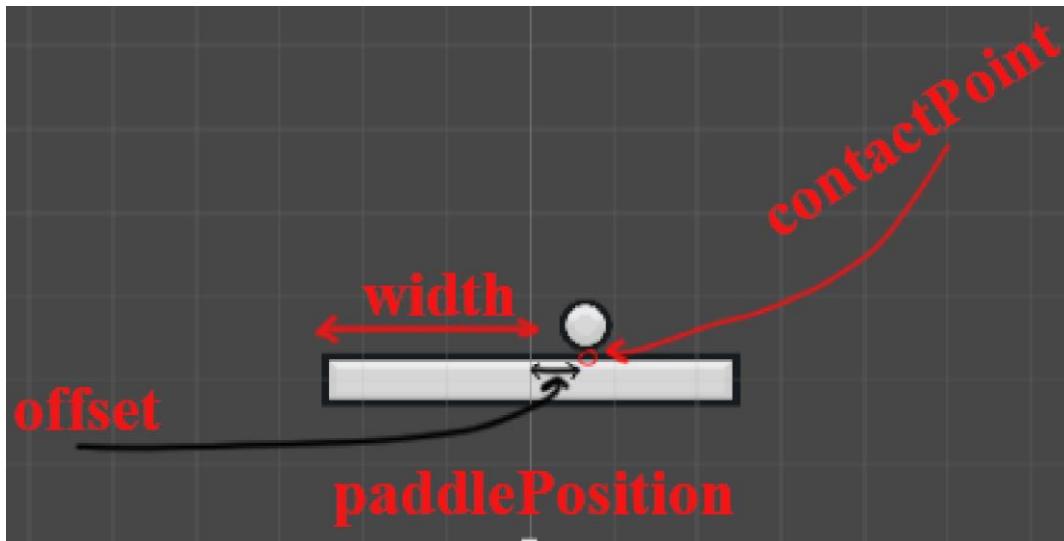
```
private void BallBounce(Ball ball, Collision2D collision)
{
}
```

◆ 1. Получение позиций и расчет смещения.

Внутри метода объявим переменные:

```
private void BallBounce(Ball ball, Collision2D collision)
{
    // координаты центра ракетки
    Vector3 paddlePosition = transform.position;
    // координаты точки, где мяч коснулся ракетки
    Vector2 contactPoint = collision.GetContact(0).point;
    // разница между центром ракетки и точкой контакта
    float offset = paddlePosition.x - contactPoint.x;
    // половина ширины ракетки (используется для нормализации
    // смещения)
    float width = collision.otherCollider.bounds.size.x / 2;
}
```

Схематичное изображение наших переменных:



◆ 2. Определение текущего угла движения мяча

Добавляем в метод:

```
float currentAngle = Vector2.SignedAngle(Vector2.up,
ball.rigidbody.linearVelocity);
```

❖ Что делает эта строка?

- Вычисляет угол между вектором "вверх" (0,1) и направлением движения мяча.
- Если мяч летит **вправо**, угол будет **положительным**.

- Если мяч летит **влево**, угол будет **отрицательным**.

◆ 3. Вычисление нового угла отскока

Добавляем в метод:

```
float bounceAngle = (offset / width) * maxBounceAngle;
```

📌 **Как вычисляется угол отскока?**

- **offset / width** приводит смещение к диапазону [-1, 1] (нормализация).
- Затем мы умножаем это значение на **maxBounceAngle**, который определяет максимальный угол отклонения (**75 градусов**).
- Если удар по **краю ракетки** → максимальный угол.
- Если удар **по центру** → угол остается прежним (прямой отскок).

◆ 4. Ограничение угла в допустимых пределах

Добавляем в метод:

```
float newAngle = Mathf.Clamp(currentAngle + bounceAngle, -maxBounceAngle, maxBounceAngle);
```

📌 **Что делает Mathf.Clamp()?**

- **currentAngle + bounceAngle** – прибавляем рассчитанное отклонение к текущему углу.
- **Mathf.Clamp()** гарантирует, что угол останется **в пределах [-maxBounceAngle, maxBounceAngle]**, чтобы мяч не отскакивал под слишком острым углом.

◆ 5. Применение нового угла к мячу

Добавляем в метод:

```
Quaternion rotation = Quaternion.AngleAxis(newAngle, Vector3.forward);
```

📌 **Что здесь происходит?**

- Мы создаем **вращение** (Quaternion) вокруг оси Z на **newAngle** градусов.
- Это задает направление движения мяча **с учетом нового угла отскока**.

◆ 6. Обновление скорости мяча

Добавляем в метод:

```
ball.rigidbody.linearVelocity = rotation * Vector2.up * ball.rigidbody.linearVelocity.magnitude;
```

📌 **Что здесь происходит?**

- **Vector2.up** – вектор вверх (0,1).
- **rotation * Vector2.up** – поворачиваем вектор вверх на **newAngle**, получая новое направление.
- **ball.rigidbody.linearVelocity.magnitude** – сохраняем изначальную скорость мяча.

👉 **В итоге мяч продолжает двигаться с той же скоростью, но под новым углом!**

Итоговый код для метода BallBounce():

```
private void BallBounce(Ball ball, Collision2D collision)
{
    Vector3 paddlePosition = transform.position;
    Vector2 contactPoint = collision.GetContact(0).point;
    float offset = paddlePosition.x - contactPoint.x;
    float width = collision.otherCollider.bounds.size.x / 2;

    float currentAngle = Vector2.SignedAngle(Vector2.up,
ball.rigidbody.linearVelocity);
    float bounceAngle = (offset / width) * maxBounceAngle;
    float newAngle = Mathf.Clamp(currentAngle + bounceAngle, -
maxBounceAngle, maxBounceAngle);

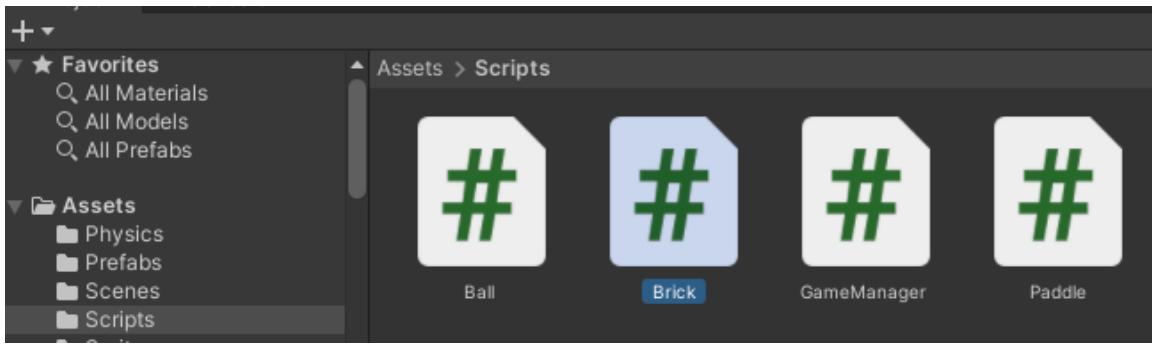
    Quaternion rotation = Quaternion.AngleAxis(newAngle,
Vector3.forward);
    ball.rigidbody.linearVelocity = rotation * Vector2.up * 
ball.rigidbody.linearVelocity.magnitude;
}
```

И теперь нам нужно сделать проверку столкновений ракетки с мячом. Сделаем это методе **OnCollisionEnter2D()**:

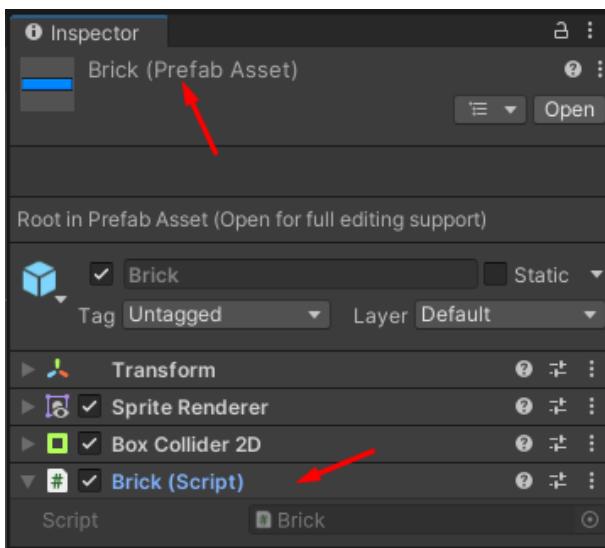
```
private void OnCollisionEnter2D(Collision2D collision)
{
    // получаем ссылку на колизию компонента мяча, с
    // которым будет происходить столкновение
    Ball ball = collision.gameObject.GetComponent<Ball>();

    if (ball != null) // проверяем, что столкнувшийся
    // объект – это мяч
    {
        BallBounce(ball, collision); // если это мяч,
        // вызываем метод BallBounce
    }
}
```

14. Настало время заняться реализацией уничтожения кирпичей, при ударе по ним мяча. При этом мы хотим, чтобы разноцветные кирпичи уничтожались от разного количества ударов. Создаём новый скрипт **Brick**:



Перемещаем его на наш префаб:



Открываем его.

➤ Объявляем переменные:

```
[Header("Компоненты")]
//отвечает за отображение спрайта кирпича.
public SpriteRenderer spriteRenderer { get; private set; }

[Header("Состояния кирпича")]
// Список спрайтов для разных состояний
[SerializeField] private Sprite[] states; кирпича
// Флаг неразрушимости
[SerializeField] private bool unbreakable;
// Текущее здоровье кирпича

public int Health { get; private set; }
```

➤ Получаем ссылку на компонент SpriteRenderer в методе Awake():

```
private void Awake()
{
    spriteRenderer = GetComponent<SpriteRenderer>(); // Получаем
    // компонент SpriteRenderer
}
```

➤ Устанавливаем стартовое здоровье в методе Start():

```

private void Start()
{
    if (!unbreakable) // Если кирпич неразрушающий, у него нет
здоровья
    {
        Health = states.Length; // Задаем здоровье по количеству
состояний (спрайтов)
        spriteRenderer.sprite = states[Health - 1]; //
Устанавливаем соответствующий спрайт
    }
}

```

➤ Напишем метод **Hit()**, который реализует логику повреждения кирпича:

```

public void Hit() // Метод вызывается при ударе мяча
{
    if (unbreakable) return; // Если кирпич неразрушимый, выходим

    Health--; // Уменьшаем здоровье

    if (Health <= 0)
        gameObject.SetActive(false); // Деактивируем объект
    else
        spriteRenderer.sprite = states[Health - 1]; // Обновляем
спрайт
}

```

◆ Метод вызывается при попадании мяча в кирпич.

- ✓ Если **unbreakable == true**, кирпич не разрушается (прерываем выполнение).
- ✓ Уменьшаем **Health** на 1.
- ✓ Если **Health <= 0**, выключаем объект (**SetActive(false)**).
- ✓ Иначе обновляем спрайт, чтобы отобразить новый спрайт.

➤ Создаём метод **OnCollisionEnter2D** — столкновение с мячом:

```

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Ball")) // Проверяем, что
столкновение с мячом
    {
        Hit();
    }
}

```

Итоговый код:

```
using UnityEngine;

public class Brick : MonoBehaviour
{
    [Header("Компоненты")]
    public SpriteRenderer spriteRenderer { get; private set; }

    [Header("Состояния кирпича")]
    [SerializeField] private Sprite[] states;
    [SerializeField] private bool unbreakable;

    public int Health { get; private set; }

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

    private void Start()
    {
        if (!unbreakable)
        {
            Health = states.Length;
            spriteRenderer.sprite = states[Health - 1];
        }
    }

    public void Hit()
    {
        if (unbreakable) return;

        Health--;

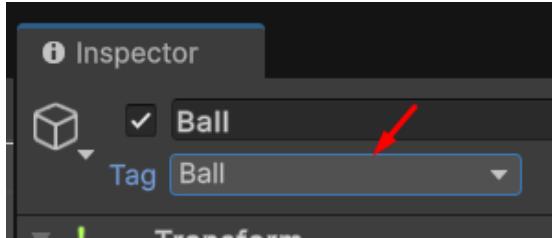
        if (Health <= 0)
        {
            gameObject.SetActive(false);
        }
        else
        {
            spriteRenderer.sprite = states[Health - 1];
        }
    }
}
```

```

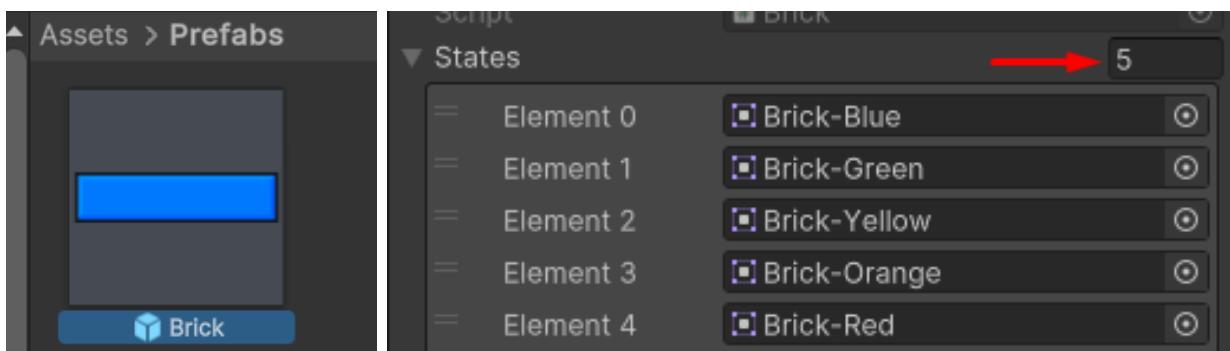
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Ball"))
    {
        Hit();
    }
}

```

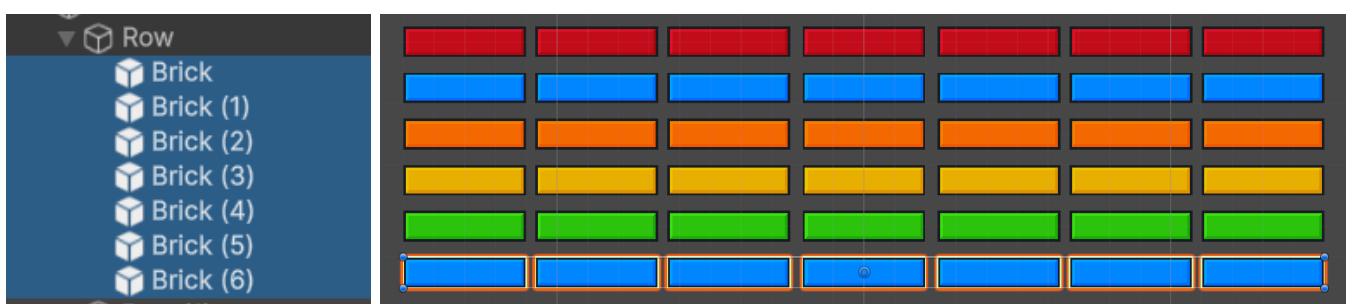
Добавляем для мячика тег – Ball:



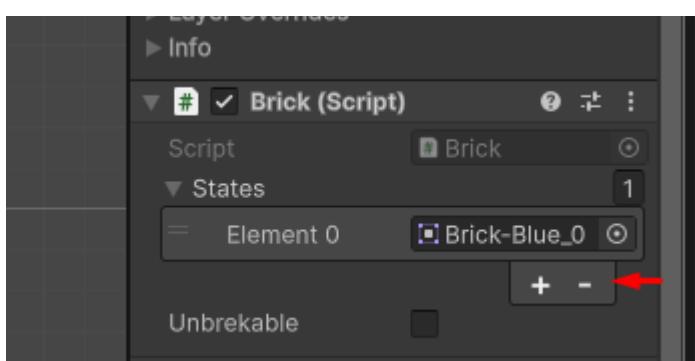
15. Далее у нашего префаба Brick меняем размер у States на 5 и ставим элементы из спрайтов по цветам Blue, Green, Yellow, Orange, Red:



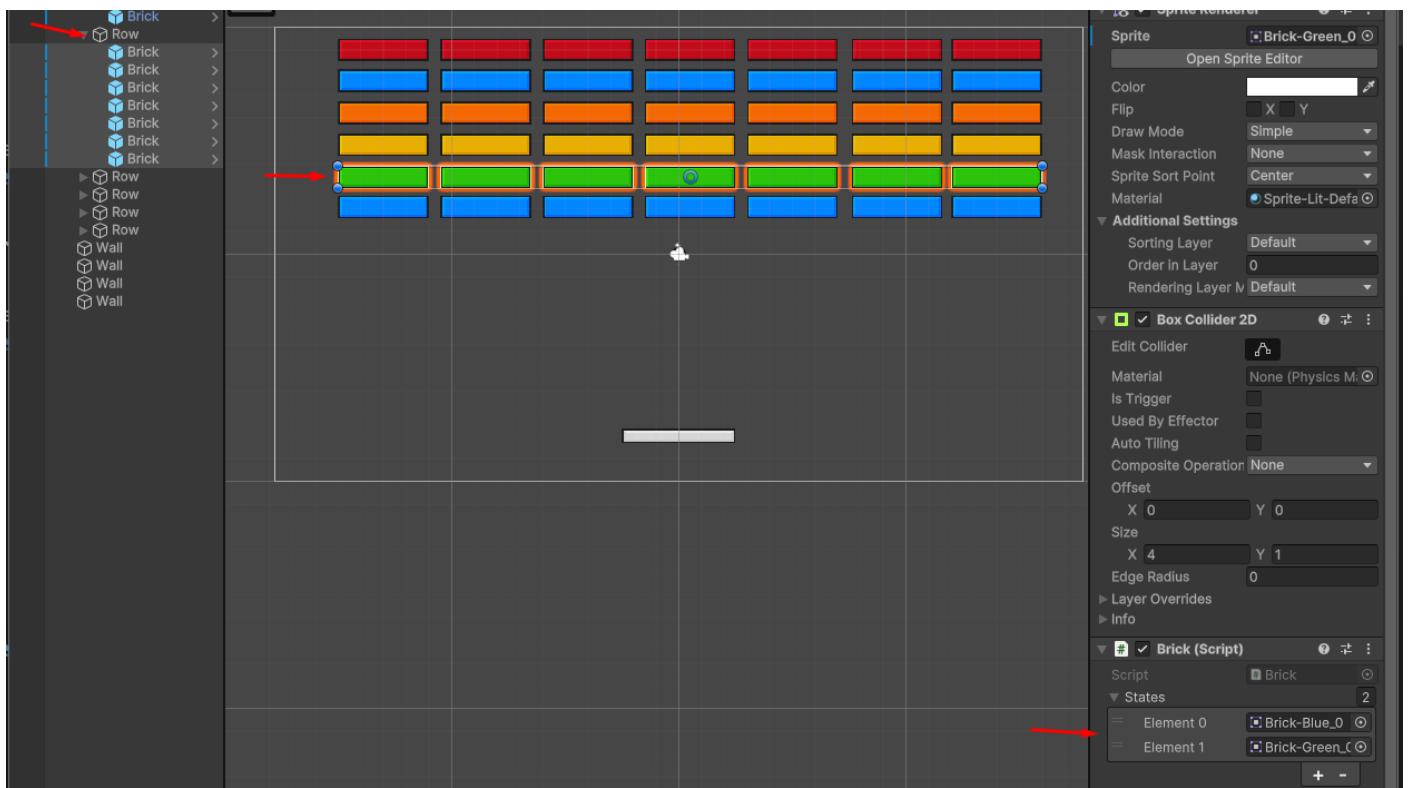
Далее выделяем в иерархии наши синие префабы:



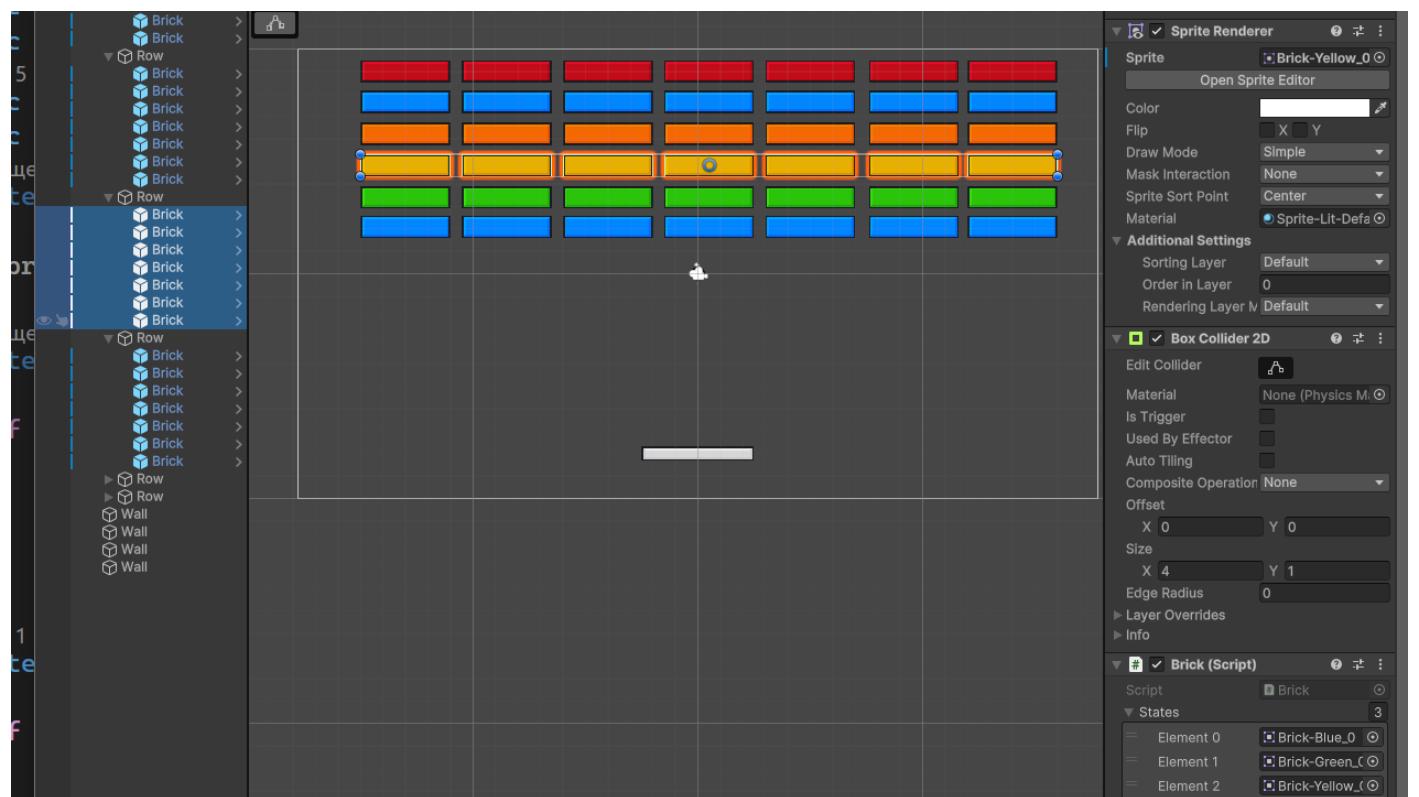
И оставляем для них один элемент (для удаления нажмите на -):



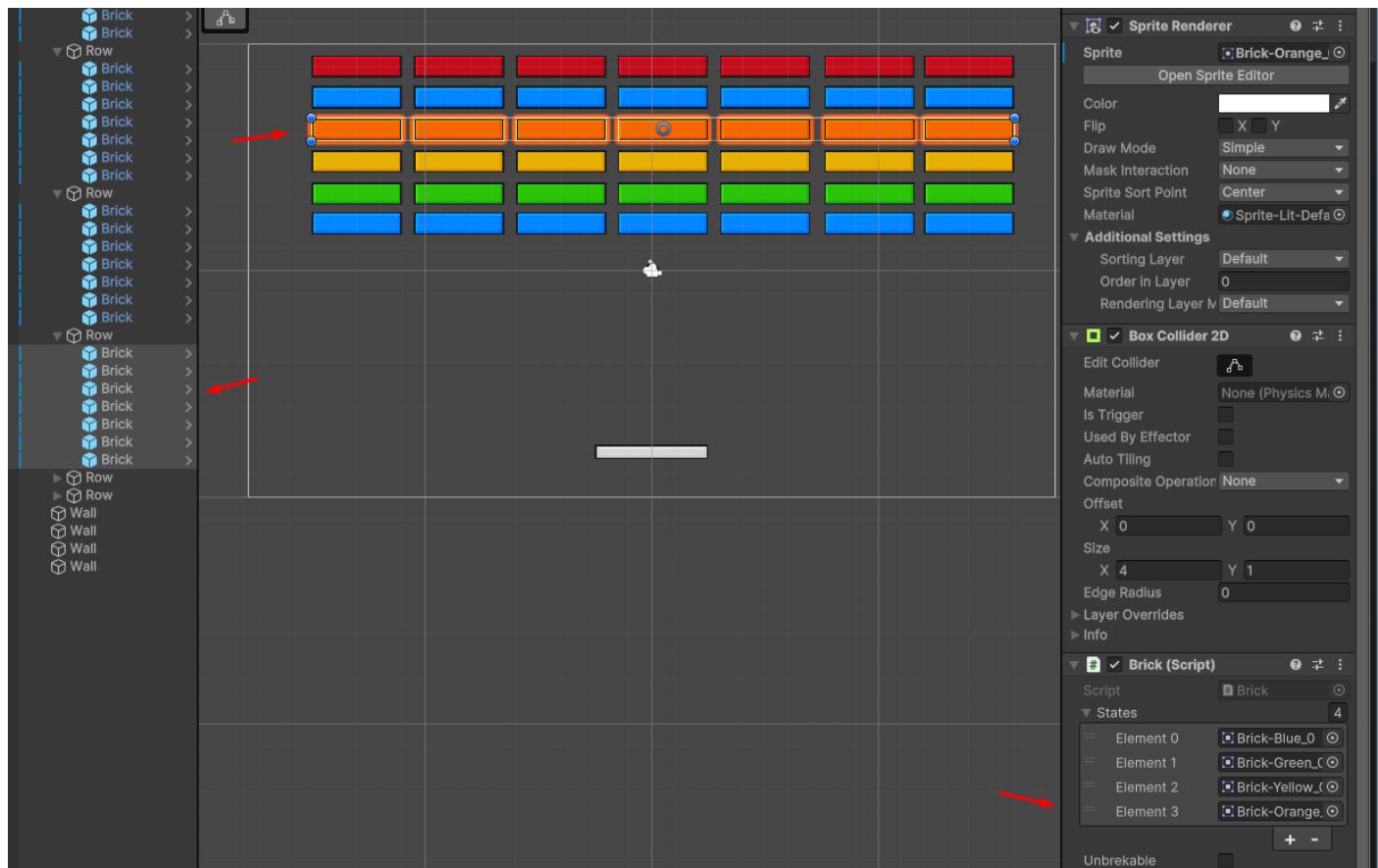
Зелёные – два элемента:



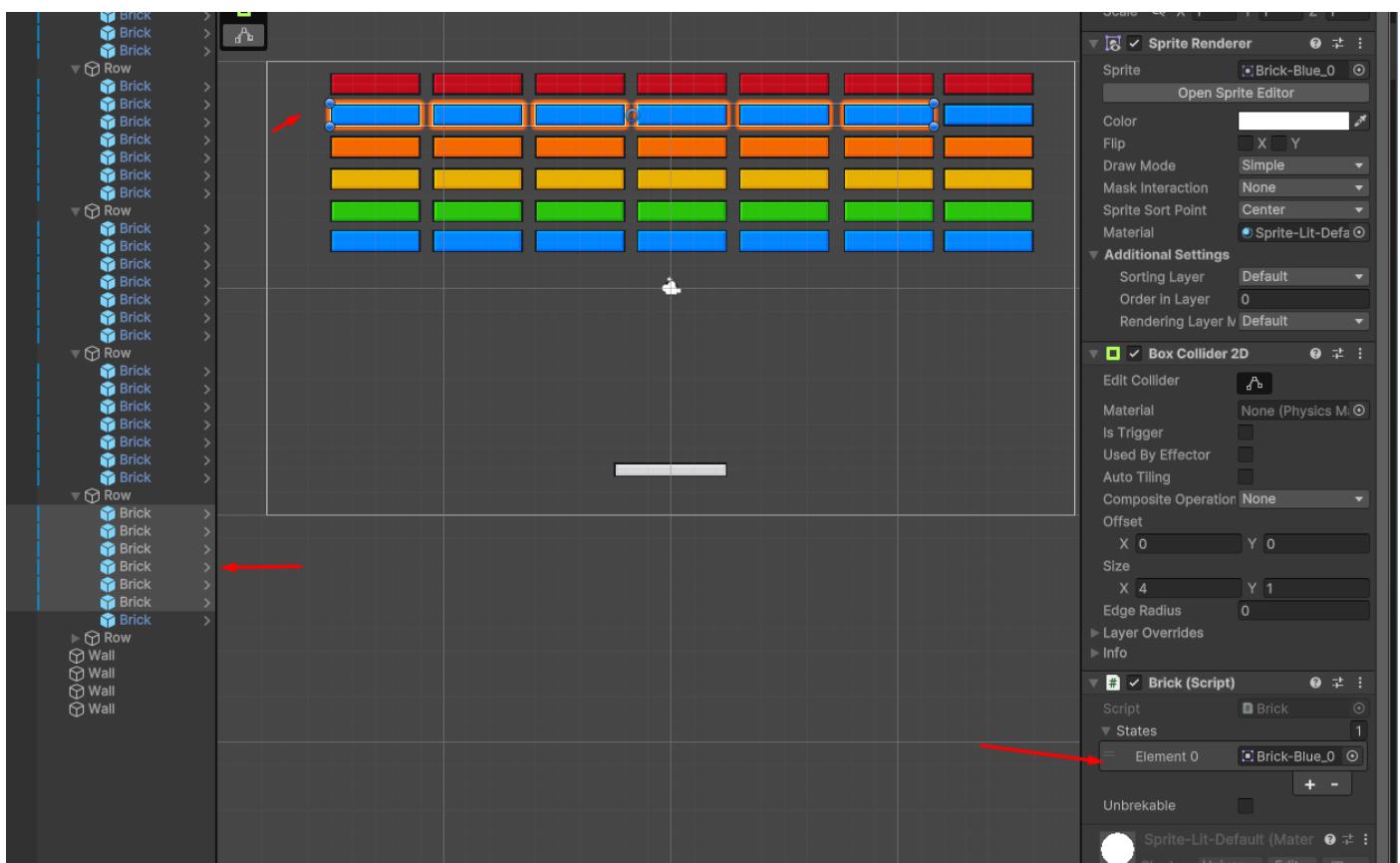
Жёлтые – три элемента:



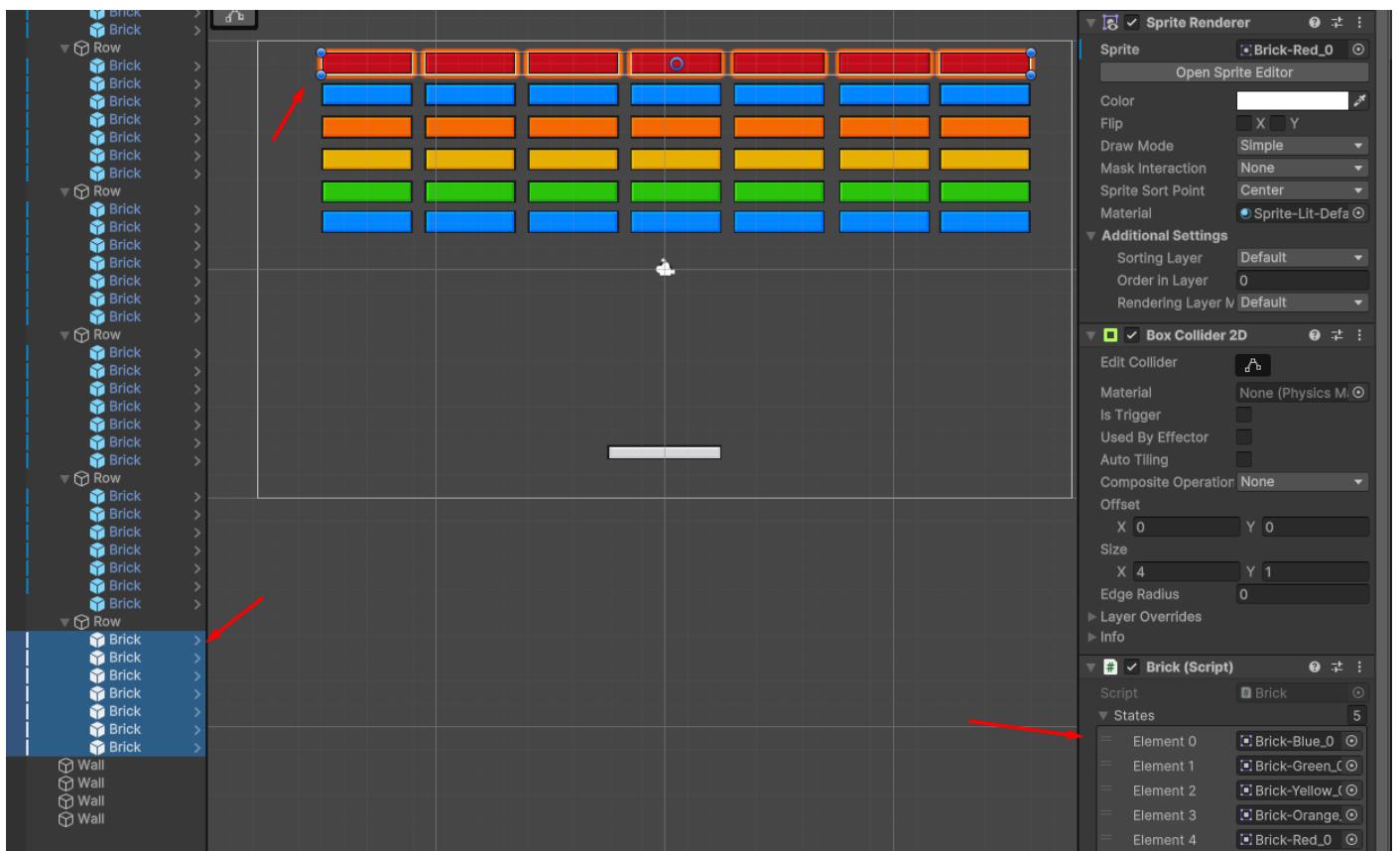
Оранжевые – четыре элемента:



В моём случае ещё раз синие, значит **один** элемент:



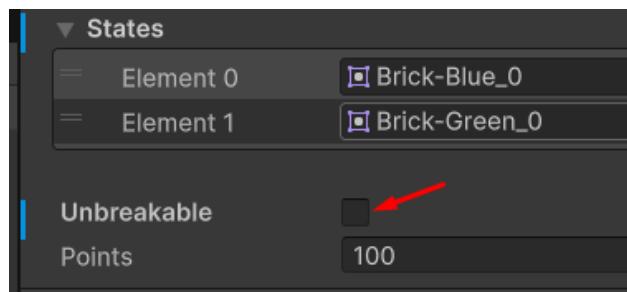
У красных оставляем все элементы:



Запускаем и начинаем тестировать игру.

Если у вас вдруг перестали разрушаться кирпичи, проверьте что не стоит галочка

в Unbreakable:



Натыкаемся на пару багов – замедление мяча, и иногда он застревает в горизонтальном или вертикальном движении.

🛠 Баг 1: Со временем мяч замедляется

🔍 Причина

При каждом столкновении мяч теряет часть скорости из-за физических расчетов движка Unity.

Это происходит из-за:

- ✓ Трения (Friction) между объектами.
- ✓ Демпфирования (Linear Drag) у Rigidbody2D.

✓ Потери импульса при AddForce().

🛠 Решение: Установить минимальную скорость

Переходим в скрипт **Ball**. Добавляем новую переменную минимальной скорости - **minSpeed** и для текущей скорости – **currentSpeed**, также подкорректируем значения:

```
[Header("Параметры мяча")]
[SerializeField] private float speed = 17f; "20"
[SerializeField] private float minSpeed = 16f; "18"
private float currentSpeed;
```

В этом же скрипте добавим метод **FixedUpdate()**:

```
private void FixedUpdate()
{
    float currentSpeed = rigidbody.linearVelocity.magnitude;
    // Ограничеваем скорость
    if (currentSpeed < minSpeed)
    {
        rigidbody.linearVelocity = rigidbody.linearVelocity.normalized *
minSpeed;
    }
}
```

🛠 **Баг 2: Мяч застревает в горизонтальном или вертикальном движении**

🔍 Причина

При вычислении нового угла **newAngle**, иногда он становится **слишком близким к 90° или 0°** , что приводит к **почти горизонтальному или вертикальному движению**.

Если угол = 0° , мяч двигается **строго влево или вправо**.

Если угол = $\pm 90^\circ$, мяч летит **строго вверх или вниз**, что ломает геймплей.

🛠 Решение: Ограничить углы отскока

Добавим **безопасный диапазон углов** (например, $\pm 10^\circ$ от 0° и $\pm 80^\circ$ от 90°).

Открываем скрипт **Paddle** и внесём правки в метод **BallBounce()**:

```

private void BallBounce(Ball ball, Collision2D collision)
{
    Vector3 paddlePosition = transform.position;
    Vector2 contactPoint = collision.GetContact(index: 0).point;
    float offset = paddlePosition.x - contactPoint.x;
    float width = collision.otherCollider.bounds.size.x / 2;

    float currentAngle = Vector2.SignedAngle(Vector2.up, ball.rigidbody.linearVelocity);
    float bounceAngle = (offset / width) * maxBounceAngle;
    float newAngle = Mathf.Clamp(currentAngle + bounceAngle, min: -maxBounceAngle,
        maxBounceAngle);

    // Добавляем проверку на вертикальные и горизонтальные углы
    if (Mathf.Abs(newAngle) < 10f) newAngle = 10f * Mathf.Sign(newAngle);
    if (Mathf.Abs(newAngle) > 80f) newAngle = 80f * Mathf.Sign(newAngle);

    Quaternion rotation = Quaternion.AngleAxis(newAngle, Vector3.forward);
    ball.rigidbody.linearVelocity *= rotation * Vector2.up * ball.rigidbody
        .linearVelocity.magnitude;
}

```

16. Добавим подсчёт очков в нашу игру.

В скрипте **Bricks** добавим переменную, которая будет отслеживать очки, и чтобы каждый кирпич давал очки:

```
public int points = 100;
```

В скрипте **GameManager** добавим метод:

```
public void Hit(Brick brick) // объявляем новый метод,
в который передаём кирпич, по которому был нанесён удар
{



}
```

В скрипте **Bricks** в метод **Hit** добавим:

```
public void Hit() // Метод вызывается при ударе мяча
{
    FindAnyObjectByType<GameManager>().Hit(brick: this);
```

Этот метод ищет в сцене объект типа **GameManager** и возвращает ссылку на него. **GameManager**. После того как объект **GameManager** найден, вызывается его метод **Hit**. **this** передается как аргумент в метод **Hit**. **this** в данном контексте указывает на текущий экземпляр класса, в котором выполняется этот код.

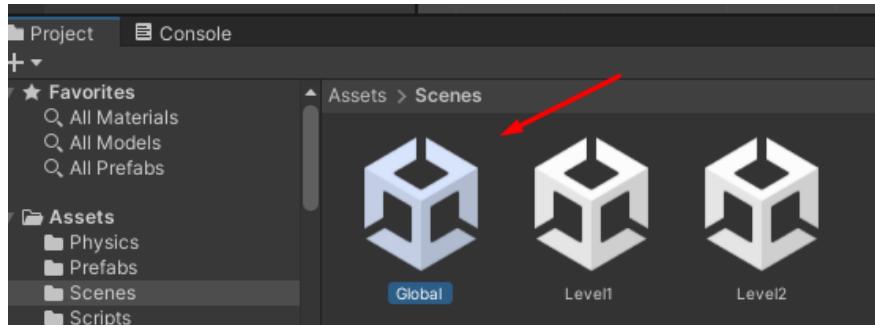
В скрипте **GameManager** допишем метод **Hit** (будет увеличивать очки при ударе об кирпич):

```

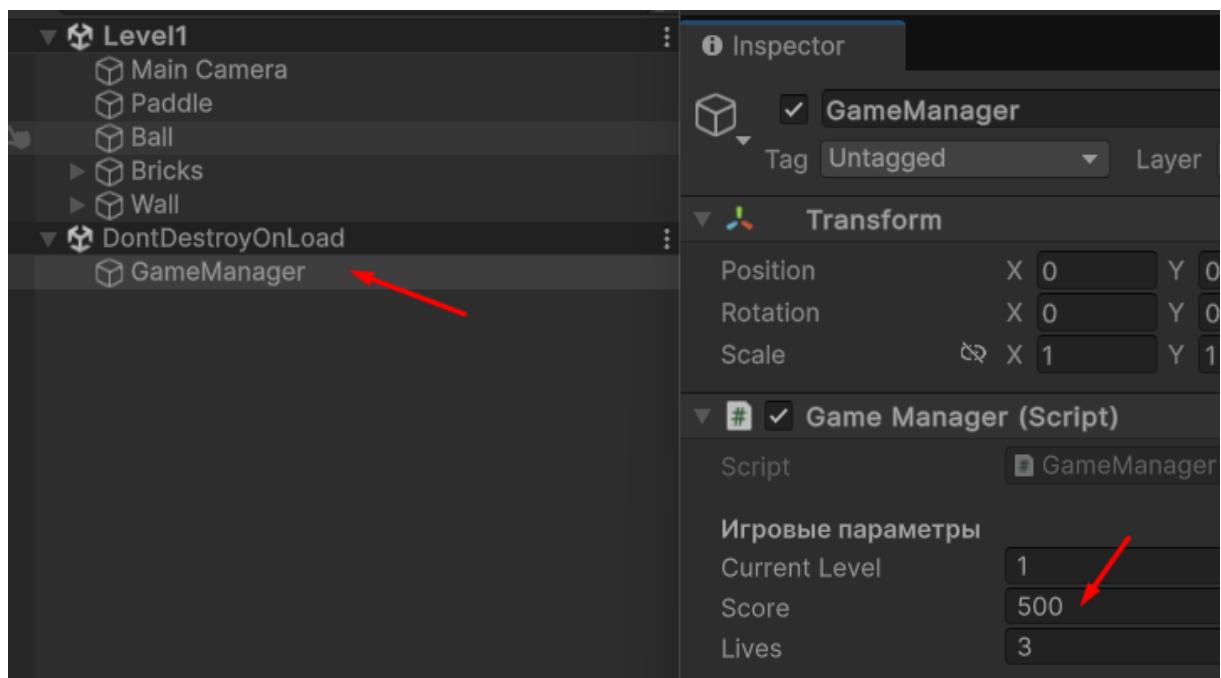
public void Hit(Brick brick)
{
    score += brick.points;
}

```

Чтобы у нас не возникало ошибки, нужно игру для просмотра запускать через сцену **Global**:



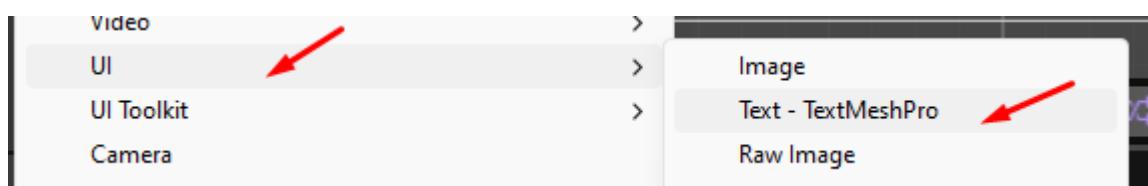
Теперь вы можете посмотреть, что **очки** отображаются в **GameManager**:



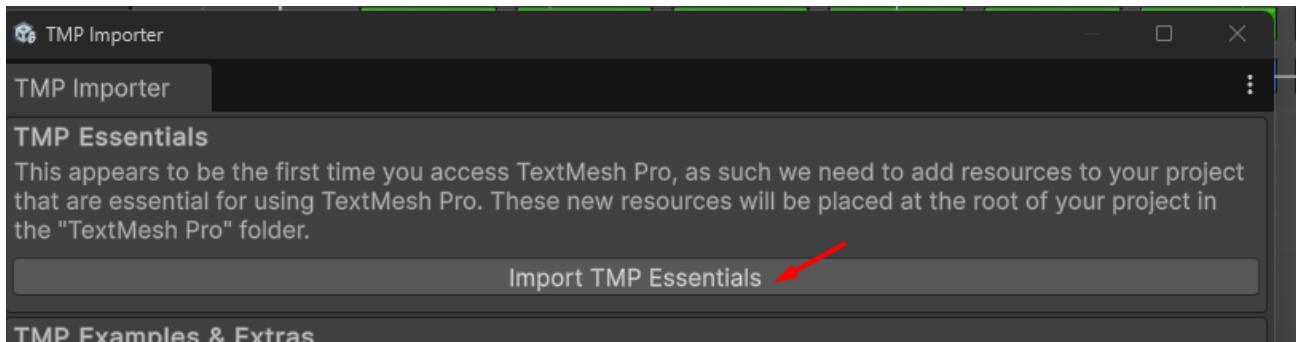
Теперь добавим отображение очков на экран.

Щёлкаем правой кнопкой мыши в иерархии и добавляем **UI - Text** -

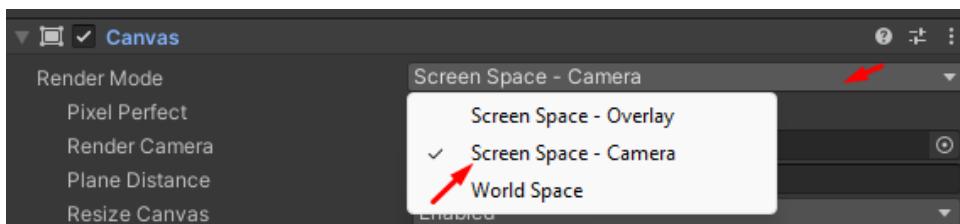
TextMeshPro:



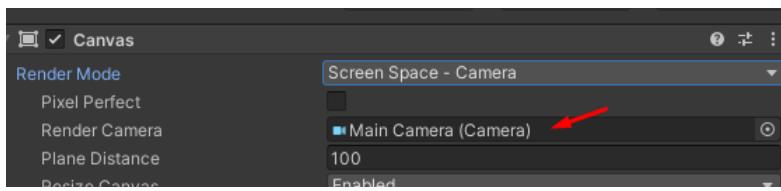
Соглашаемся на импорт шрифтов и после закрываем окно:



В Inspector у Canvas выбираем в модели рендера **World Space**:



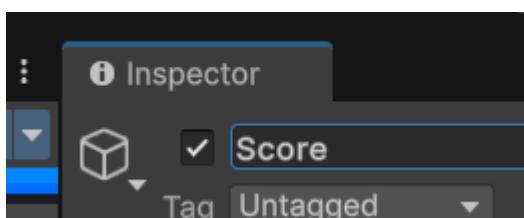
Переносим Main Camera в рендер:



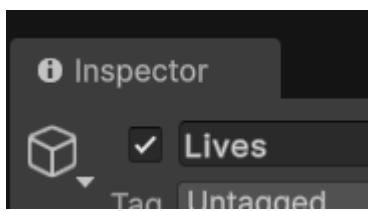
Подгоняем размер текста, под нашу игровую область. Меняем размер шрифта, цвет и текст на **Score**:



Меняем название на **Score**:



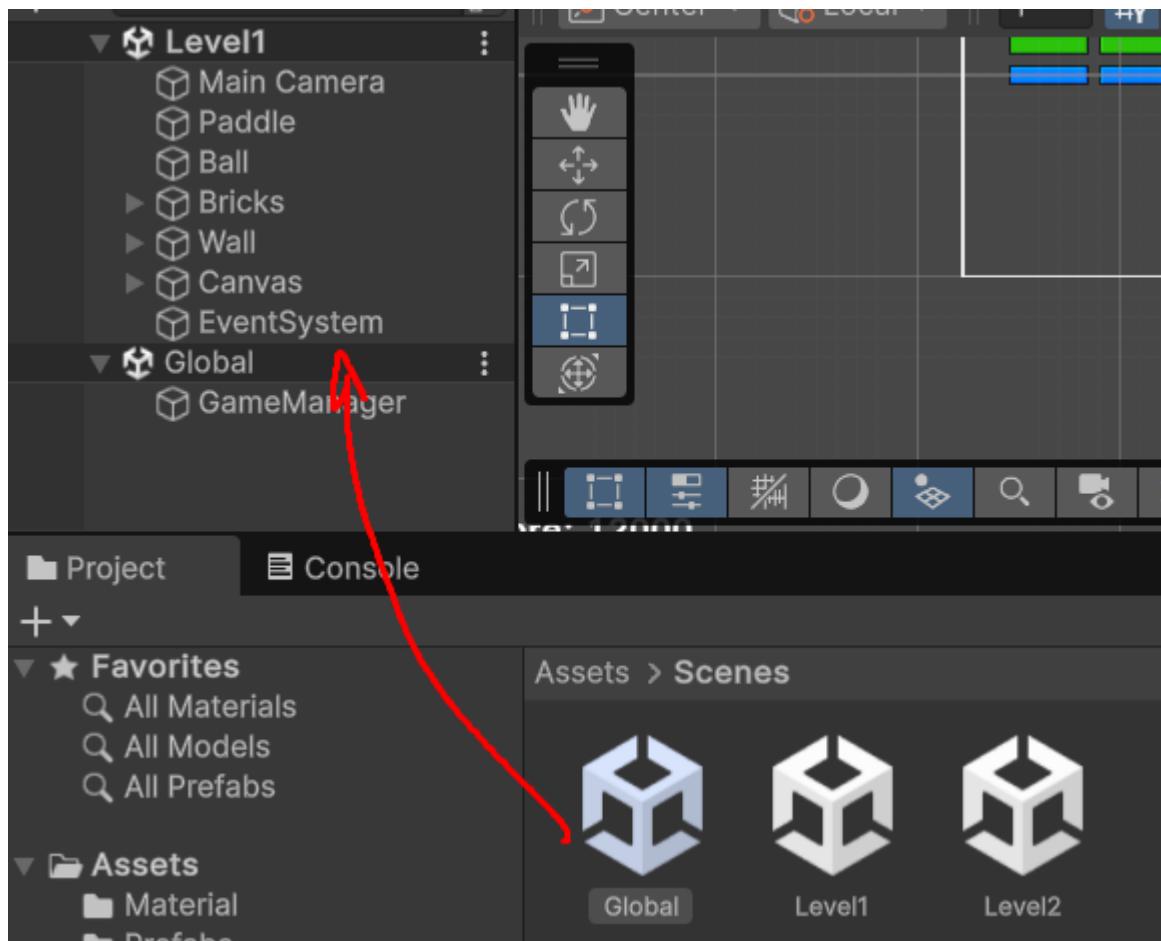
Сразу же создадим второй Text – **TextMeshPro** и назовём его **Lives**:



Подгоняем размер текста, под нашу игровую область. Меняем размер шрифта, цвет и текст на **Lives: 3**:

Lives: 3

Или, если у вас возникнет ситуация, что вы хотите протестировать уровень дальше, тогда откройте нужный вам уровень и перетащите **Global** внутрь него:



И вот мы хотим с вами по привычке объявить публичное поле и добавить на него наш текст. Но есть маленькая проблема, **Canvas** создаётся в сцене уровня, а **GameManager** живёт в глобальной сцене, поэтому просто перетащить ссылку на текст в инспекторе нельзя. Но есть **три хороших решения** этой проблемы:

Способ	Плюсы	Минусы
1. Находить <code>Text</code> при загрузке уровня	<ul style="list-style-type: none">✓ Работает без изменений UI.✓ Подходит для динамических сцен.	<ul style="list-style-type: none">✗ Небольшая задержка поиска объекта.✗ <code>FindObjectOfType</code> медленный при больших сценах.
2. Canvas в глобальной сцене (лучший вариант)	<ul style="list-style-type: none">✓ Работает стабильно.✓ UI не пропадает между уровнями.✓ Не нужно искать <code>Text</code>.	<ul style="list-style-type: none">✗ Нужно переносить <code>Canvas</code> в <code>Global</code>-сцену.
3. Использовать <code>PlayerPrefs</code>	<ul style="list-style-type: none">✓ Очки сохраняются между сессиями.	<ul style="list-style-type: none">✗ Не обновляет текст в реальном времени.

Мы воспользуемся первым способом.

Переходим к скрипту **GameManager**, и добавим изменения в скрипт:

```
using TMPro; // подключаем пространство имён
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    [Header("Игровые параметры")]
    [SerializeField] private int currentLevel = 1;
    [SerializeField] private int score = 0;
    [SerializeField] private int lives = 3;

    private TMP_Text scoreText; // Переменная для текста очков

    private void Awake()
    {
        DontDestroyOnLoad(gameObject);
        SceneManager.sceneLoaded += OnSceneLoaded; // Подписываемся на событие загрузки сцены
    }

    private void Start()
    {
        StartNewGame();
    }

    private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        FindScoreText(); // Ищем текст на новой сцене
        UpdateScoreText(); // Обновляем отображение очков
    }

    private void FindScoreText()
    {
        GameObject scoreObject = GameObject.Find("Score"); // Ищем объект по имени
        if (scoreObject != null)
        {
            scoreText = scoreObject.GetComponent<TMP_Text>(); // Получаем компонент TMP_Text
        }
    }
```

```

private void UpdateScoreText()
{
    if (scoreText != null)
        scoreText.text = $"Score: {score}";
}

private void StartNewGame()
{
    score = 0;
    lives = 3;
    LoadLevel(1);
}

private void LoadLevel(int levelNumber)
{
    currentLevel = levelNumber;
    SceneManager.LoadScene("Level" + levelNumber);
}

public void Hit(Brick brick)
{
    score += brick.points;
    UpdateScoreText(); // После изменения очков обновляем
текст
}
}

```

📌 Как это работает?

- ✓ **FindScoreText()** ищет текст по имени в сцене **каждый раз при загрузке нового уровня**.
- ✓ **UpdateScoreUI()** обновляет текст очков.
- ✓ **OnSceneLoaded()** вызывает **FindScoreText()** при смене уровня.
- ✓ Используется **TMP_Text**, чтобы работало с разными типами текста.

20. Давайте сделаем игру интереснее и добавим увеличение скорости мяча при достижении определённого количества очков.

В скрипте **GameManager** мы хотим добавить:

- Коэффициент увеличения скорости:

```

public class GameManager : MonoBehaviour
{
    [Header("Игровые параметры")]
    [SerializeField] private int currentLevel = 1; // Unchanged
    [SerializeField] private int score = 0; // Unchanged
    [SerializeField] private int lives = 3; // Unchanged
    [SerializeField] private float speedMultiplier = 1.0f; // Множитель скорости мяча

```

- Метод для обновления скорости мяча (напишем в конце скрипта):

```

private void UpdateBallSpeed()
{
    speedMultiplier = 1.0f + ((score / 500f) * 0.01f); // Каждые 500 очков увеличиваются скорость с корректировкой
    Ball ball = FindObjectOfType<Ball>();

    if (ball != null)
    {
        ball.SetSpeedMultiplier(speedMultiplier); // Передаем
        множитель в мяч
    }
}

```

Теперь нам необходимо вызвать его в методе Hit():

```

public void Hit(Brick brick)
{
    score += brick.points;
    UpdateScoreText();
    UpdateBallSpeed(); // вызываем метод увеличения скорости
}

```

Затем переходим в скрипт **Ball** и добавим метод **SetSpeedMultiplier()**, который изменяет скорость в зависимости от множителя:

```

public void SetSpeedMultiplier(float multiplier)
{
    currentSpeed = speed * multiplier;
    rigidbody.linearVelocity =
    rigidbody.linearVelocity.normalized * currentSpeed; // Пересчитываем скорость
}

```

◆ Как работает этот код?

- В **GameManager**

- **speedMultiplier** увеличивается каждые **500 очков**
- При каждом попадании по кирпичу вызывается **UpdateBallSpeed()**, который передаёт множитель в **Ball**

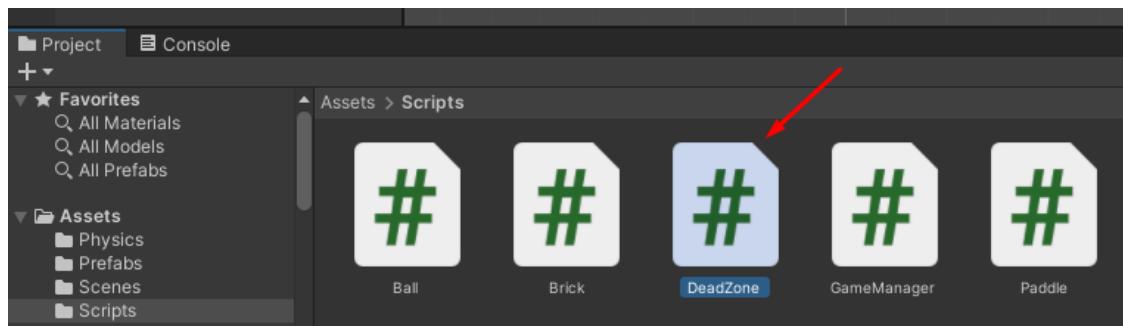
- В Ball

- SetSpeedMultiplier() увеличивает currentSpeed
- Скорость мяча пересчитывается и обновляется

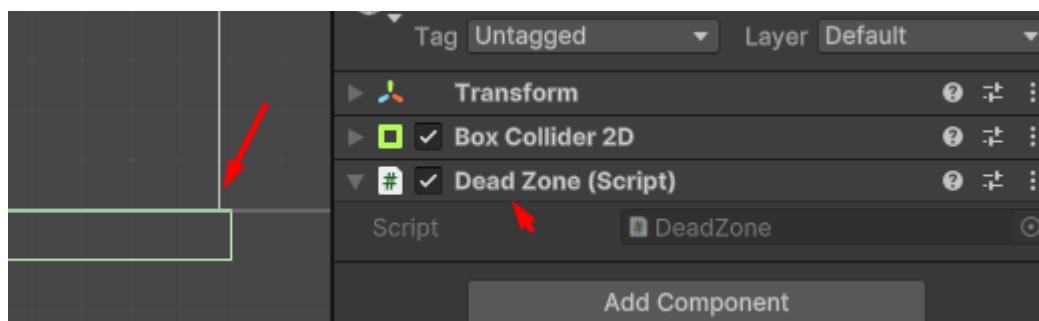
Запускаем и начинаем тестировать игру.

21. Теперь займёмся жизнями игрока.

Создадим новый скрипт **DeadZone**:



Добавим его на нижнюю стену в нашем 1 уровне:



❖ Переходим в скрипт **Ball** и добавим в конце скрипта новый метод для сброса позиции мяча:

```
public void ResetBall() // метод для сброса позиции мяча
{
    transform.position = new Vector2 (0, -2); // устанавливаем позицию в начальное положение
    rigidbody.linearVelocity = Vector2.zero; // сбрасываем скорость
    Invoke(nameof(SetRandomTrajectory), 1f); // делаем задержку в 1 секунду и запускаем движение мяча
}
```

- ❖ Переходим в скрипт **Paddle** и добавим в конце скрипта новый метод для сброса позиции ракетки:

```
public void ResetPaddle() // метод для сброса позиции
ракетки
{
    transform.position = new Vector2(0f,
transform.position.y); // перемещаем ракетку в начальную
позицию по оси X
    rigidbody.linearVelocity = Vector2.zero; // сбрасываем
линейную скорость
}
```

- ❖ Переходим в скрипт **GameManager**.

➤ начнём с добавления **переменных**:

```
[Header("Текстовые объекты")]
private TMP_Text scoreText;
private TMP_Text livesText; // UI для отображения жизней

[Header("Игровые объекты")]
public Ball ball { get; private set; } // ссылка на мяч
public Paddle paddle { get; private set; } // ссылка на ракетку
```

➤ Далее добавим метод **FindLivesText()** для поиска текста **Lives**, по аналогии с тем, что мы делали со **score**:

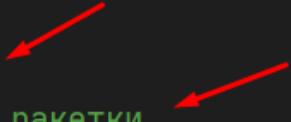
```
private void FindLivesText() // поиск текстового объекта для
жизней
{
    GameObject livesObject = GameObject.Find("Lives"); // ищем
объект по имени
    if (livesObject != null) // проверка на null
    {
        livesText = livesObject.GetComponent<TMP_Text>(); // Получаем компонент TMP_Text
    }
}
```

➤ Далее добавим метод **FindGameObjects()** для поиска игровых объектов мяча и ракетки на сцене:

```
private void FindGameObjects() // поиск мяча и ракетки на сцене
{
    ball = FindAnyObjectByType<Ball>(); // Ищем мяч
    paddle = FindAnyObjectByType<Paddle>(); // Ищем ракетку
}
```

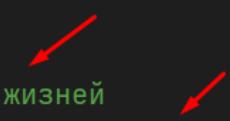
➤ Вызовем их методе **OnSceneLoaded()**:

```
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    FindScoreText();
    UpdateScoreText();
    FindLivesText(); // обновляем UI жизней
    FindGameObjects(); // инициализация мяча и ракетки
}
```



➤ Вызовем отображение жизней на экране в методе **UpdateScoreText()**:

```
private void UpdateScoreText()
{
    if (scoreText != null)
        scoreText.text = $"Score: {score}";
    if (livesText != null) // дополнительная проверка для жизней
        livesText.text = $"Lives: {lives}"; // отображаем кол-во жизней
}
```



➤ Создадим метод **Miss()**, который будет вызываться, когда мяч ударится о пол. Он будет отнимать жизни и выводить их на экран:

```
public void Miss() // обработка пропуска мяча
{
    lives--; // отнимаем жизни
    UpdateScoreText(); // сразу обновляем текст
    if (lives > 0) // проверяем если жизней больше 0
        ResetLevel(); // вызываем сброс уровня
    else
        StartNewGame(); // перезапуск игры
}
```

➤ Создадим метод **ResetLevel()**, который будет сбрасывать позицию и скорость мяча и ракетки:

```
private void ResetLevel() // сброс позиции мяча и ракетки
{
    ball.ResetBall(); // сбрасываем позицию и скорость мяча
    paddle.ResetPaddle(); // сбрасываем позицию и скорость
    ракетки
}
```

У нас было очень много правок в скрипте **GameManager**, поэтому полная версия:

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

public class GameManager : MonoBehaviour
{
    [Header("Игровые параметры")]
    [SerializeField] private int currentLevel = 1;
    [SerializeField] private int score = 0;
    [SerializeField] private int lives = 3;
    [SerializeField] private float speedMultiplier = 1.0f;

    [Header("Текстовые объекты")]
    private TMP_Text scoreText;
    private TMP_Text livesText;

    [Header("Игровые объекты")]
    public Ball ball { get; private set; }
    public Paddle paddle { get; private set; }

    private void Awake()
    {
        DontDestroyOnLoad(gameObject);
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    private void Start()
    {
        StartNewGame();
    }

    private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        FindScoreText();
        UpdateScoreText();
        FindLivesText();
        FindGameObjects();
    }

    private void LoadLevel(int levelNumber)
    {
        currentLevel = levelNumber;
        SceneManager.LoadScene("Level" + levelNumber);
    }
}
```

```
private void FindGameObjects()
{
    ball = FindAnyObjectOfType<Ball>();
    paddle = FindAnyObjectOfType<Paddle>();
}
private void FindScoreText()
{
    GameObject scoreObject = GameObject.Find("Score");
    if (scoreObject != null)
    {
        scoreText = scoreObject.GetComponent<TMP_Text>();
    }
}
private void FindLivesText()
{
    GameObject livesObject = GameObject.Find("Lives");
    if (livesObject != null)
    {
        livesText = livesObject.GetComponent<TMP_Text>();
    }
}
private void UpdateScoreText()
{
    if (scoreText != null)
        scoreText.text = $"Score: {score}";
    if (livesText != null)
        livesText.text = $"Lives: {lives}";
}

private void StartNewGame()
{
    score = 0;
    lives = 3;
    LoadLevel(1);
}

public void Hit(Brick brick)
{
    score += brick.points;
    UpdateScoreText();
    UpdateBallSpeed();
}
```

```

private void UpdateBallSpeed()
{
    speedMultiplier = 1f + ((score / 500f) * 0.01f);
    Ball ball = FindAnyObjectByType<Ball>();
    if (ball != null)
        ball.SetSpeedMultiplier(speedMultiplier);
}

public void Miss()
{
    lives--;
    UpdateScoreText();
    if (lives > 0)
        ResetLevel();
    else
        StartNewGame();
}
private void ResetLevel()
{
    ball.ResetBall();
    paddle.ResetPaddle();
}
}

```

❖ Осталось прописать наш скрипт **DeadZone** для обработки столкновений мяча с определенной зоной, которая определяет потерю жизни или окончание уровня:

```

using UnityEngine;

public class DeadZone : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.name == "Ball")
        {
            FindAnyObjectByType<GameManager>().Miss();
        }
    }
}

```

22. Теперь нам нужно создать метод, который будет проверять очистили ли мы игровое поле при ударе мяча о кирпич, чтобы потом загрузить **новый уровень**.

В начале нам нужно с вами будет получать ссылку на переменную **unbreakable** из скрипта **Brick**. Но мы её делали с **[SerializeField]**, и как мы узнали сегодня использовать **public** для переменных — не лучшая практика, так как это нарушает инкапсуляцию и делает код менее

контролируемым. В вашем случае можно сделать переменную **unbreakable** доступной для чтения из других скриптов, но ограничить запись только внутри текущего класса. Для этого можно использовать свойство с модификатором доступа { **get**; **private set**; }.

Поэтому в скрипте **Brick** меняем переменную **unbreakable** на свойство:

```
public bool unbreakable { get; private set; }
```

Переходим в скрипт **GameManager**.

➤ Добавим переменную ссылку на наш массив кирпичей:

```
public Brick[] bricks { get; private set; }
```

➤ Затем в методе **FindGameObjects()** будем искать все кирпичи, включая скрытые:

```
private void FindGameObjects()
{
    ball = FindAnyObjectByType<Ball>();
    paddle = FindAnyObjectByType<Paddle>();
    → bricks = FindObjectsOfType<Brick>(FindObjectsInactive.Include,
        FindObjectsSortMode.None); // ищем кирпичи
}
```

- **FindObjectsInactive.Include**: Включает в поиск неактивные объекты.
- **FindObjectsSortMode.None**: Отключает сортировку результатов, что делает метод быстрее.

➤ Создадим метод **Cleared()**, он будет проверять, есть ли разрушаемые кирпичи:

```
private bool Cleared()
{
    foreach (Brick brick in bricks)
    {
        if (brick.gameObject.activeInHierarchy &&
!brick.unbreakable)
            return false;
    }
    return true;
}
```

- Проходит по всем кирпичам в *bricks*.
- Если есть разрушаемый кирпич, который активен, возвращаем *false* (уровень не очищен).
- Если все кирпичи разрушены, возвращаем *true* (можно загружать новый уровень).

➤ Создадим метод **Cleared()**, он будет проверять, очищен ли уровень:

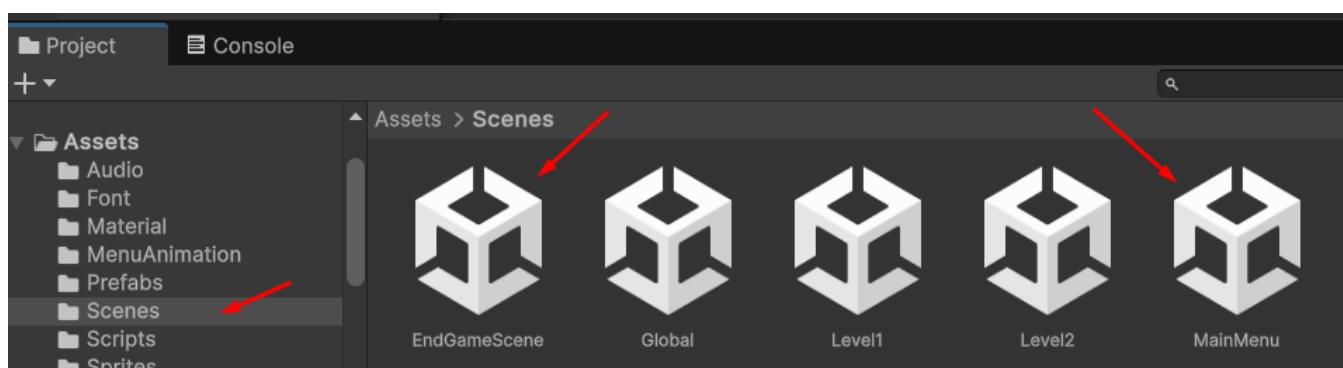
```
private void CheckLevelClear()
{
    if (Cleared())
        LoadLevel(currentLevel + 1);
}
```

Добавить в метод Hit() – Обработка удара по кирпичу:

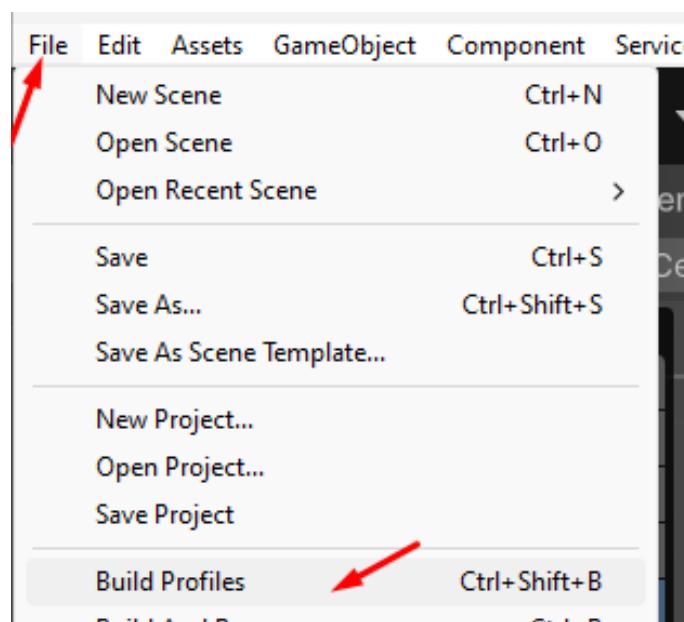
```
public void Hit(Brick brick)
{
    score += brick.points;
    UpdateScoreText();
    UpdateBallSpeed();

    // Отложенная проверка через 0.1 секунды
    Invoke(nameof(CheckLevelClear), 0.1f);
}
```

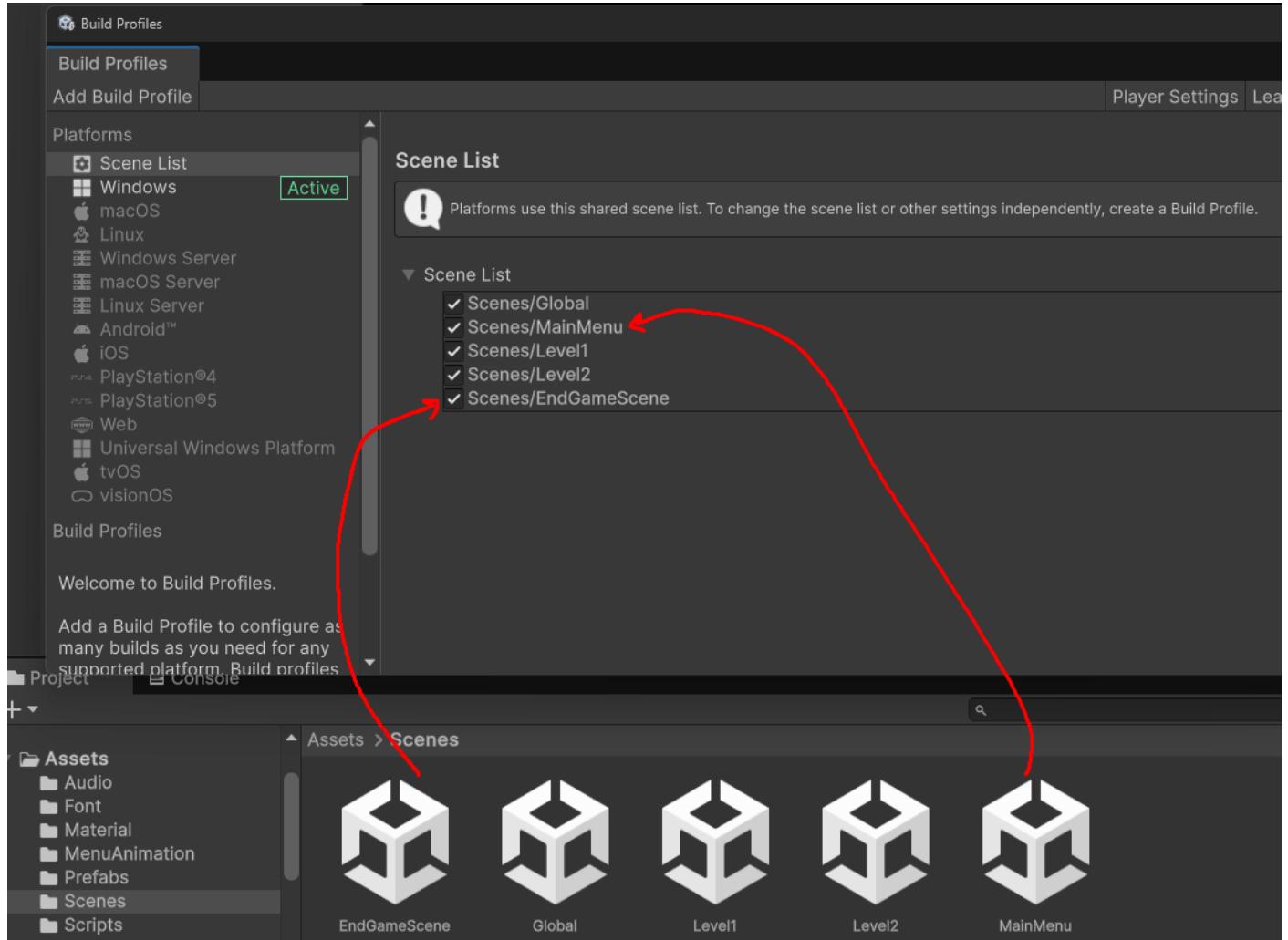
23. Добавим и оформим начальную и конечную сцену в папке **Scenes**. Назовём их **EndGameScene** и **MainMenu**:



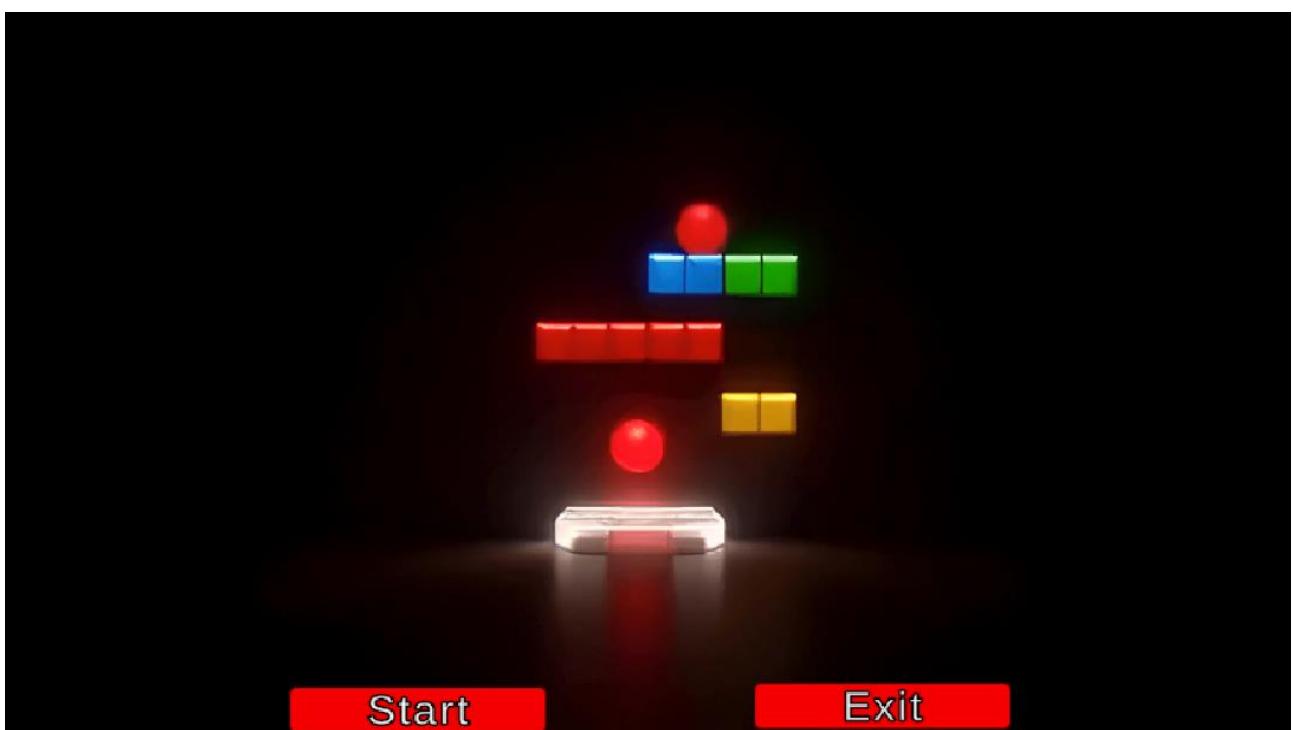
Затем перейдём в **File – Build Profiles**:



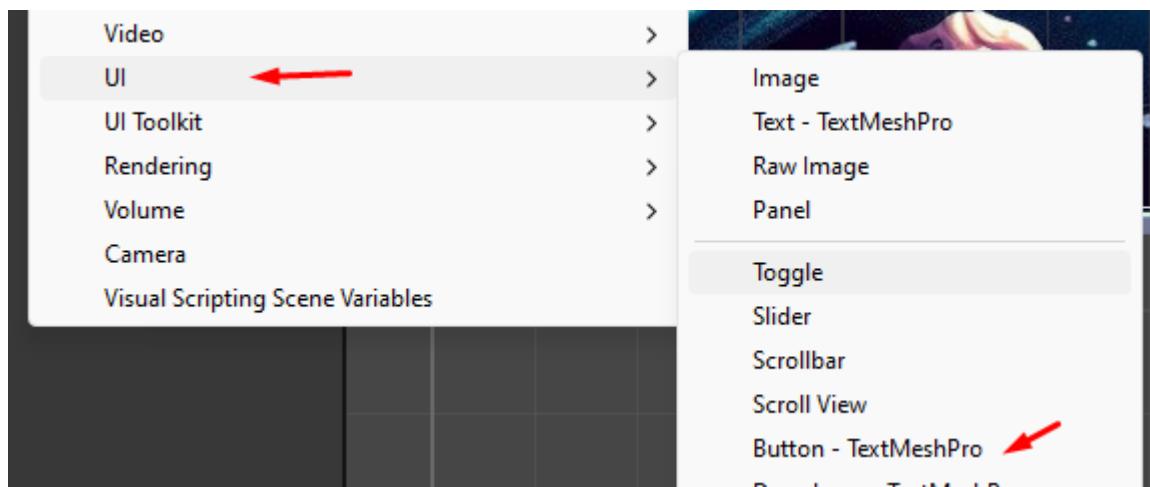
И в **Scene List** перенесём наши **две** новые сцены, соблюдая правильную последовательность наших сцен:



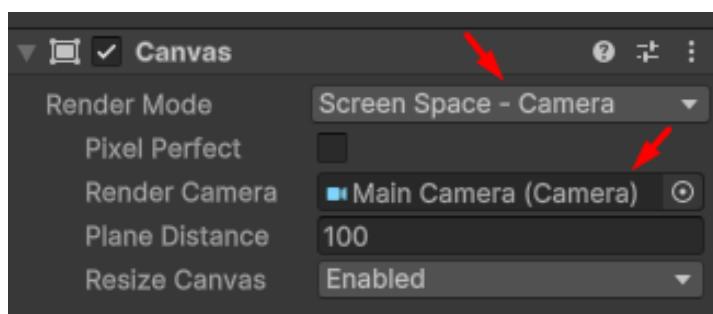
24. Переходим к начальной сцене **MainMenu**. В ней мы хотим сделать **анимацию из наших картинок и две кнопки** – выхода из игры и начала игры:



Создаём UI-Button-TextMeshPro:



Для нашего **Canvas** меняем отображение на **World Space**, и добавляем нашу камеру:

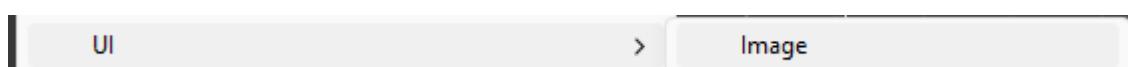


Также создаём вторую кнопку, и называем их **ButtonStart** и **ButtonExit** соответственно. Оформляем их на свой вкус.

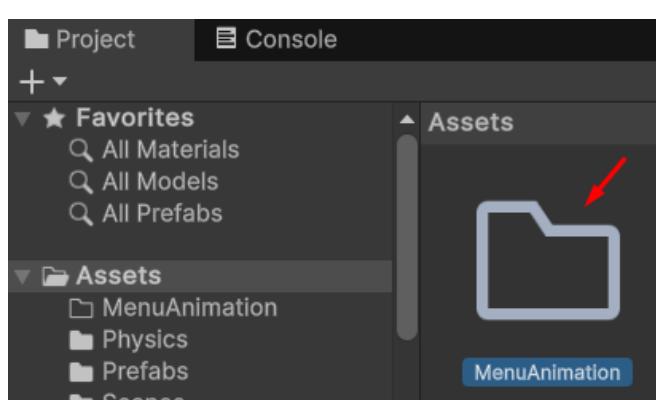
Теперь добавим изображение. Щёлкаем правой кнопкой мыши по **Canvas**



И выбираем **UI-Image**:



Создаём в папке с ассетами новую папку **MenuAnimation**:

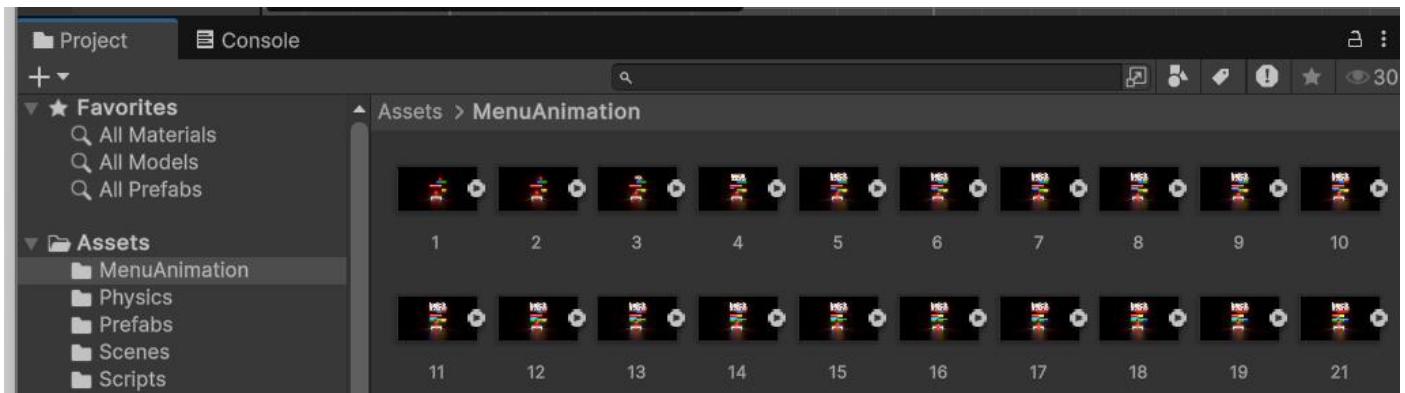


Переносим в неё из архива

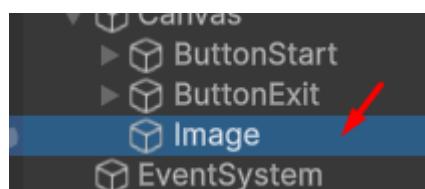
Assets. Brick Breaker. Menu.rar

наши

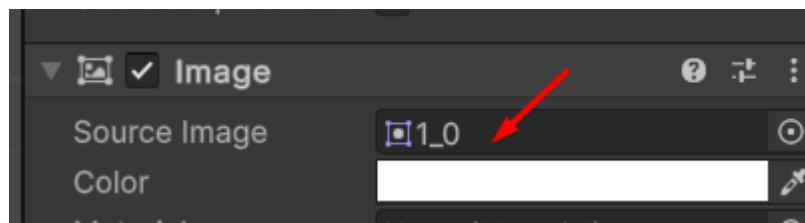
изображения:



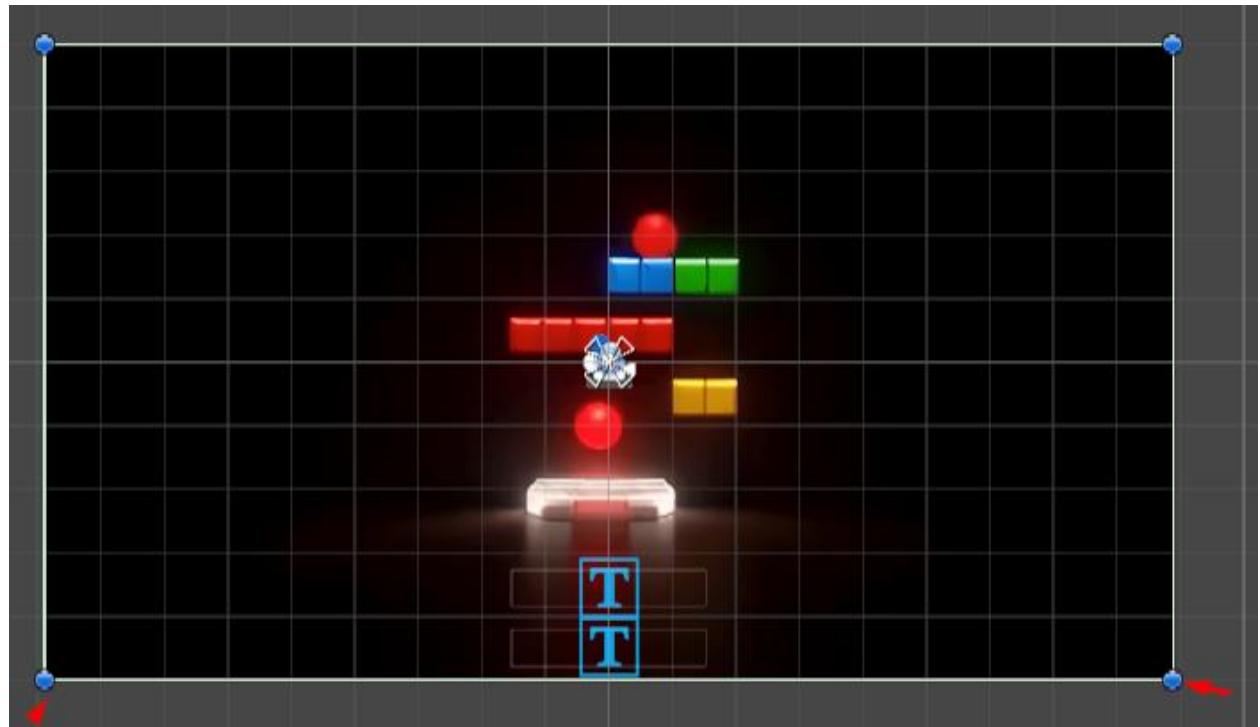
Для нашего объекта **Image**:



Перенесём в поле **Source Image** наше первое изображение:



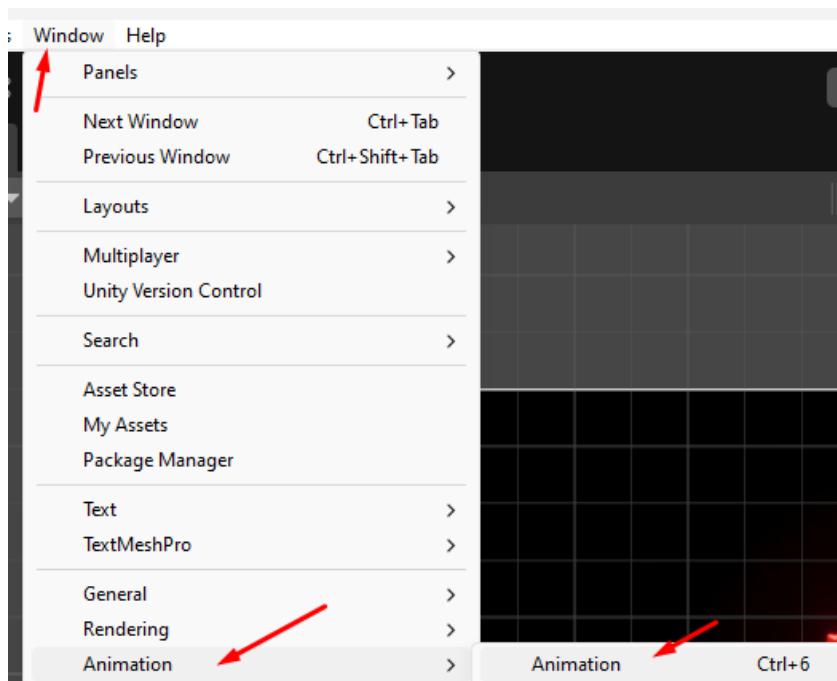
Растянем его в пределах нашей камеры:



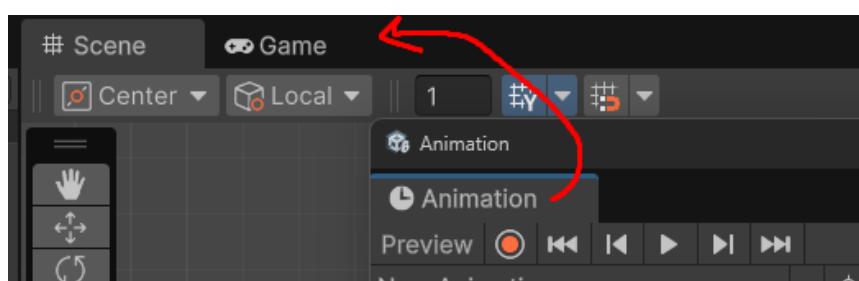
Обратите внимание что ваши кнопки стали перекрыты. Чтобы сделать их видимыми, нужно поднять вашу картинку в иерархии выше кнопок:



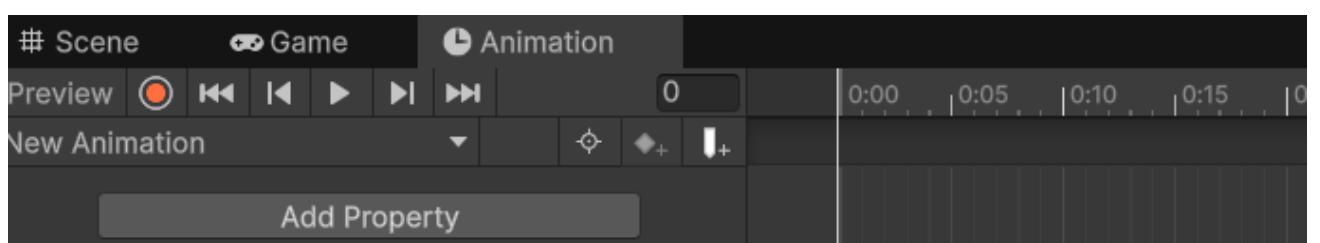
Теперь откроем окно для работы с анимацией – Windows - Animation – Animation:



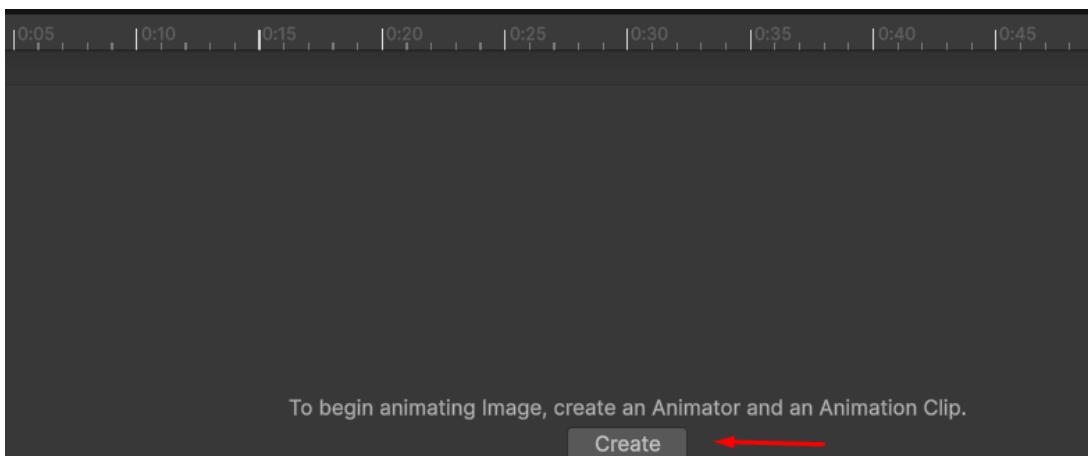
Для удобства работы закрепите ваше окно к любой из панели, например где сцена и игра:



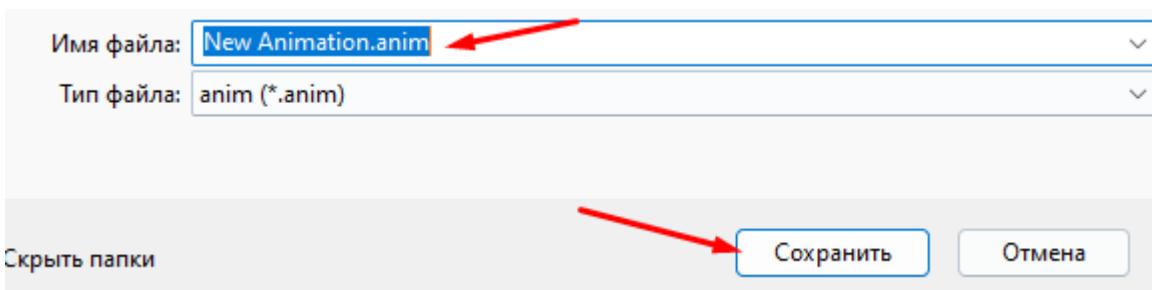
Стало:



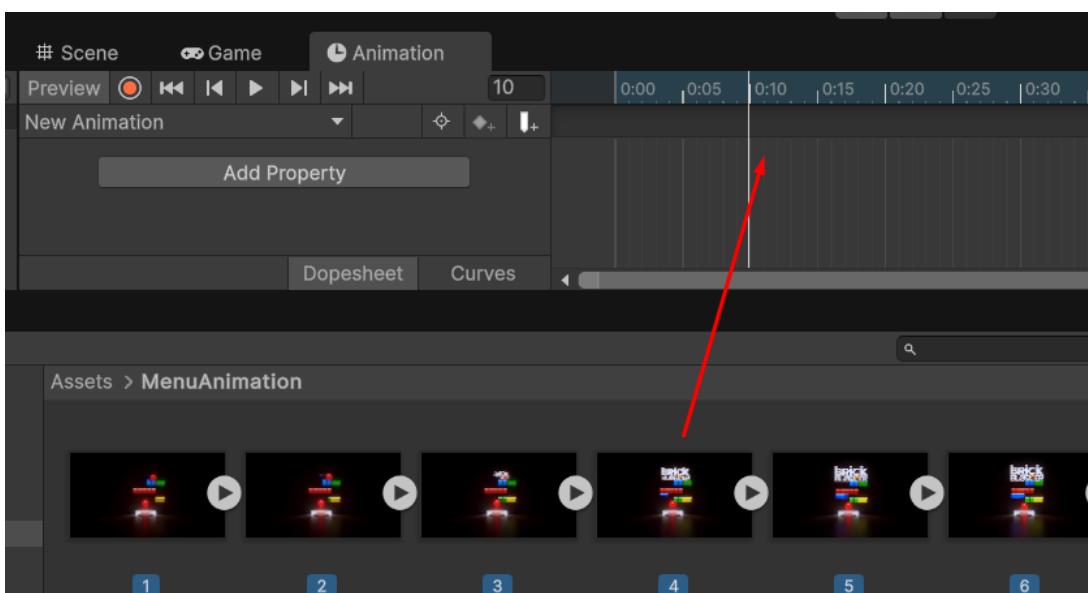
Выделяем наш объект **Image**, нажимаем на **Create** для создания новой анимации в текущем окне:



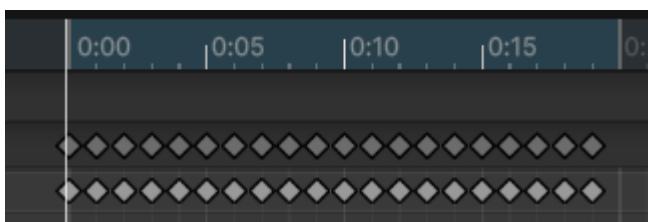
Вам предложит ввести имя и место сохранения (нас устраивает всё по умолчанию, поэтому нажимаем **Сохранить**):



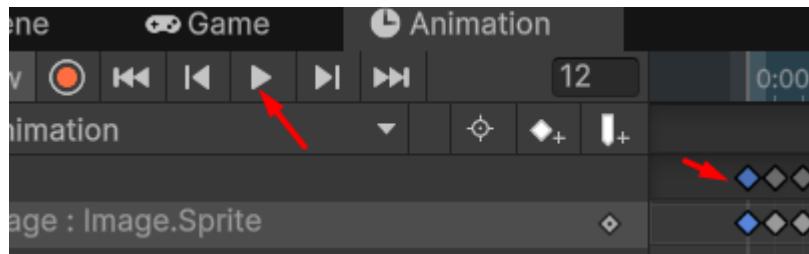
Теперь выбираем все наши изображения и переносим на таймлайн:



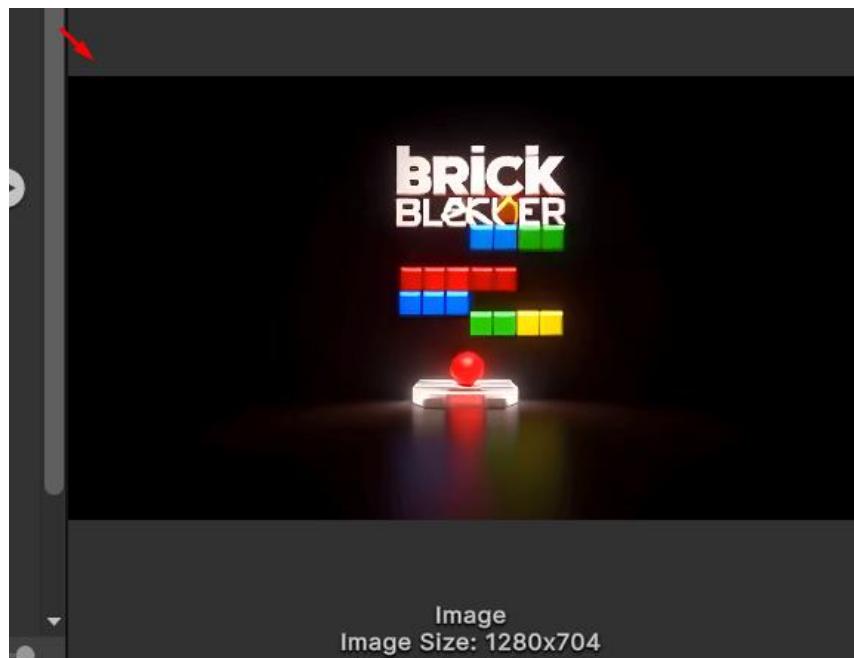
Обратите внимание теперь у нас появились все кадры:



И выбрав первый картд, мы можем запустить и посмотреть, как выглядит анимация:



Тут анимация в инспекторе воспроизводится:

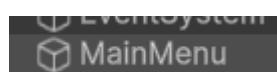


В нашем случае она выглядит слишком быстрой, поэтому выделяем наши все ключевые кадры и тянем за уголок, чтобы увеличить промежуток между кадрами:

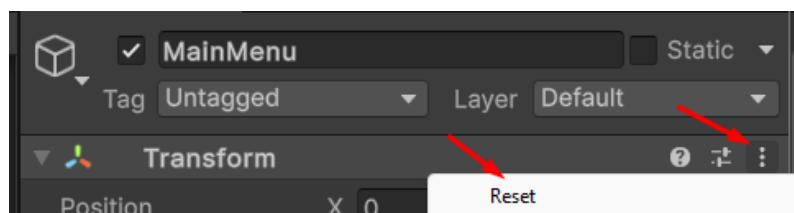


Можно растянуть до **40-60** секунд, на ваш выбор.

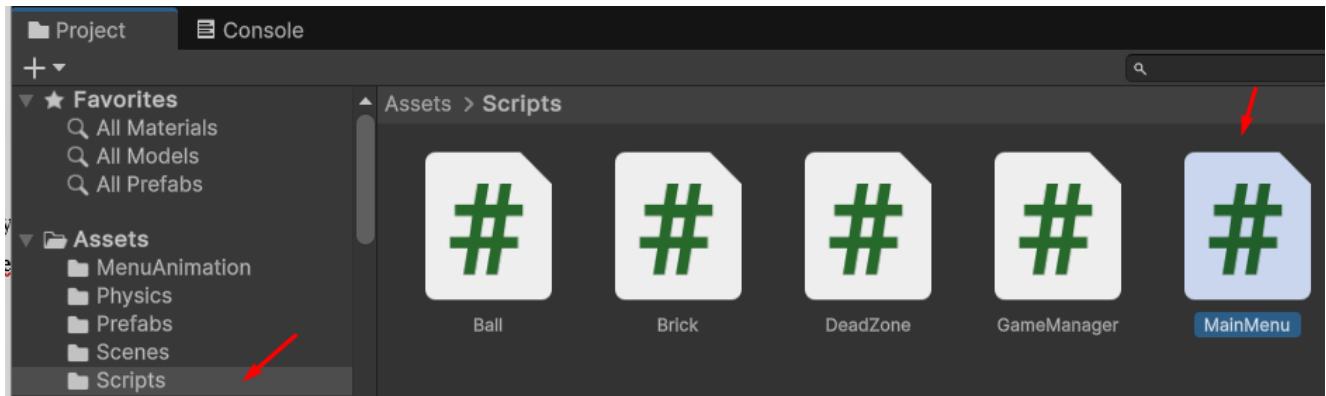
25. Теперь создадим объект **MainMenu**



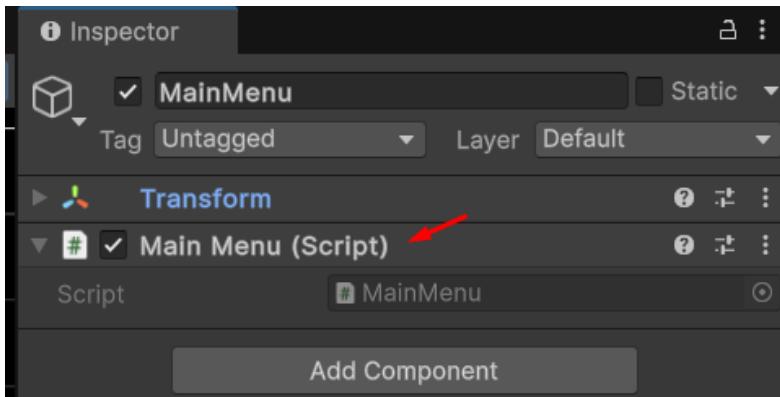
Сбрасываем для него трансформацию:



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



И прикрепляем его к нашему объекту **MainMenu**:



Открываем его и пишем код:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
    public void StartGame()
    {
        SceneManager.LoadScene("Level1"); // Переход на новую
сцену
    }

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

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

Директивы препроцессора — это специальные команды, которые выполняются ещё до компиляции кода. Они позволяют изменять поведение программы в зависимости от условий.

1. **#if UNITY_EDITOR** — проверяет, запущен ли код в редакторе Unity

2. Если игра запущена в редакторе, выполняется:

UnityEditor.EditorApplication.isPlaying = false;

Это просто останавливает выполнение сцены в редакторе.

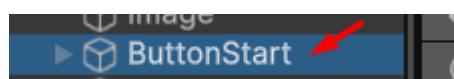
3. **#else** — если игра запущена уже как готовое приложение (например, на ПК, телефоне и т. д.), выполняется:

Application.Quit;

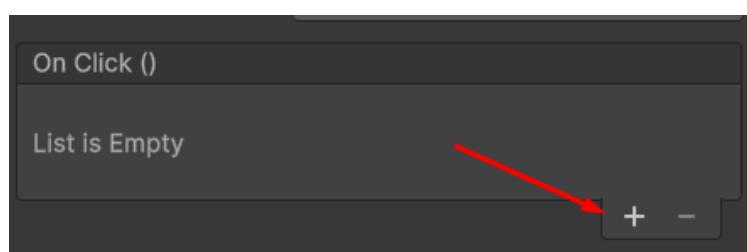
Этот метод закрывает игру.

4. **#endif** завершает блок условия.

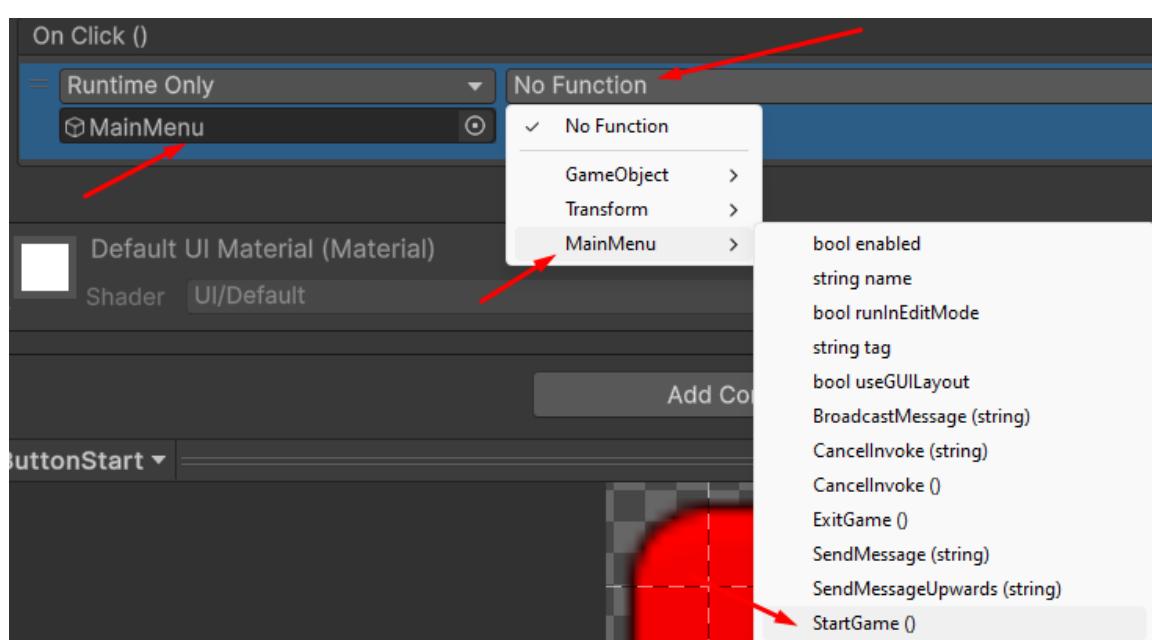
Выбираем нашу кнопку **ButtonStart**:



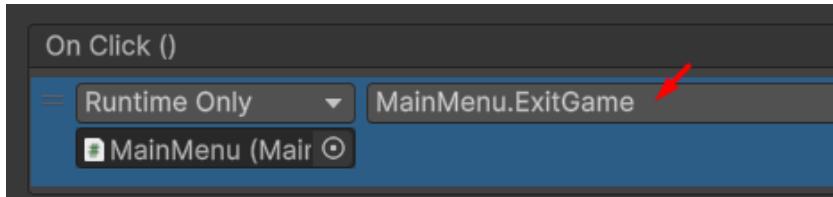
Добавляем для неё новый клик нажав на +:



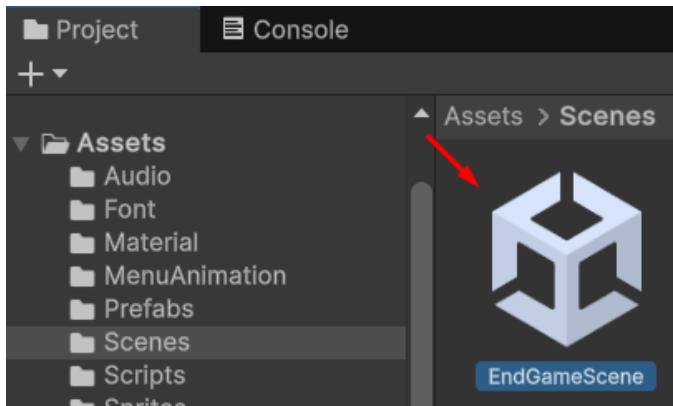
Переносим объект **MainMenu**, и выбираем функцию **MainMenu – StartGame()**:



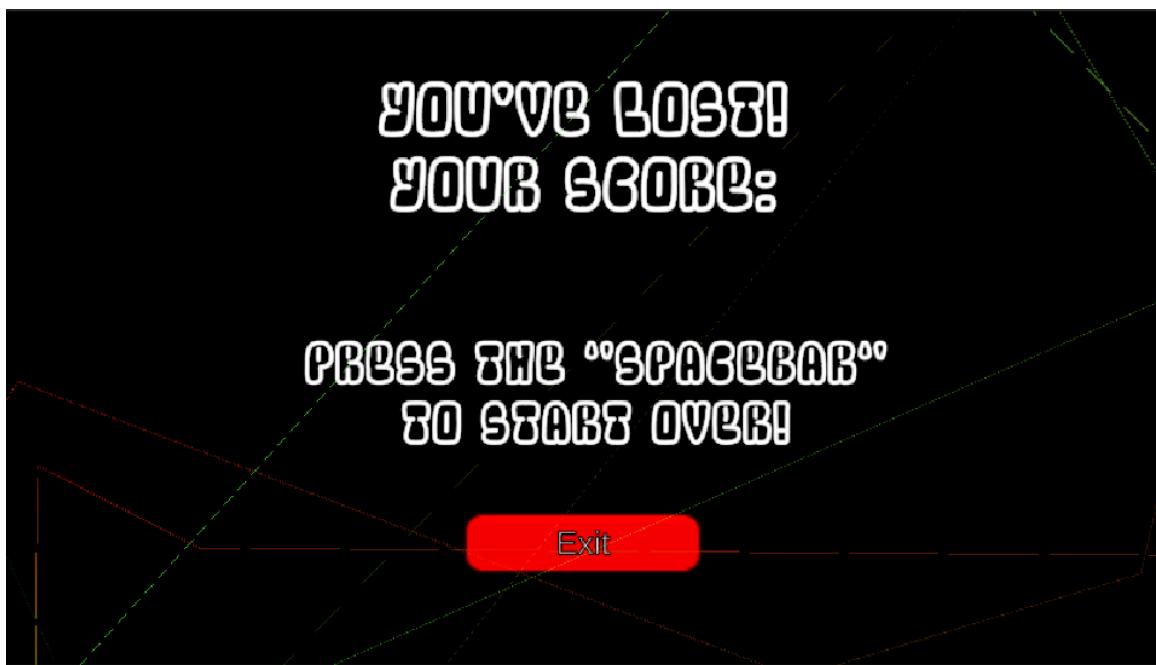
Тоже самое делаем для кнопки **ButtonExit**, но выбираем функцию **ExitGame()**:



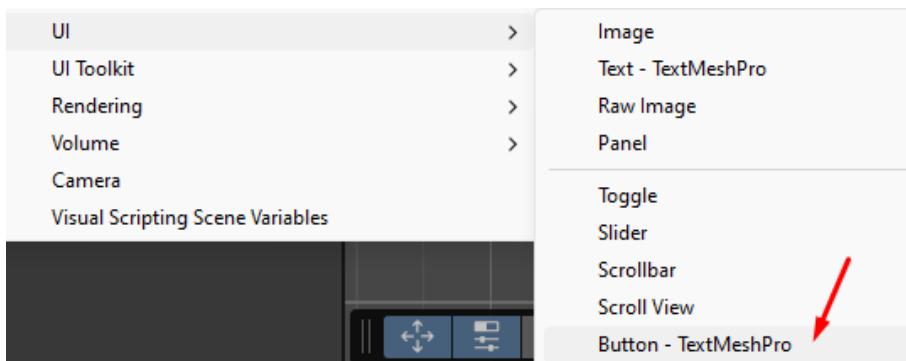
25. Теперь перейдём к финальной сцене **EndGameScene**:



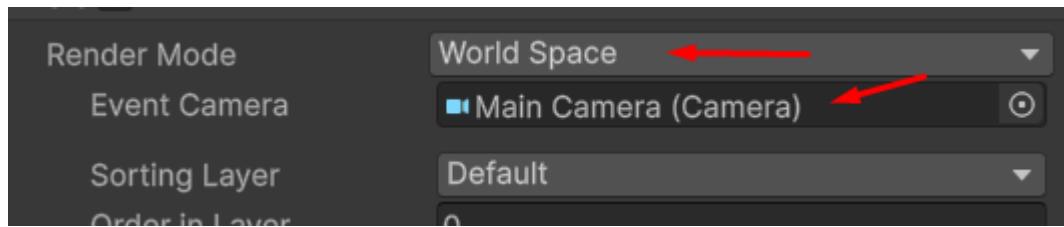
Мы хотим на ней выводить наш **счёт**, сделать возможность вернуться в игру по нажатию **кнопки пробел**, добавить **кнопку выхода из игры**, а также добавить **систему частиц**:



Создаём **UI-Button-TextMeshPro**:



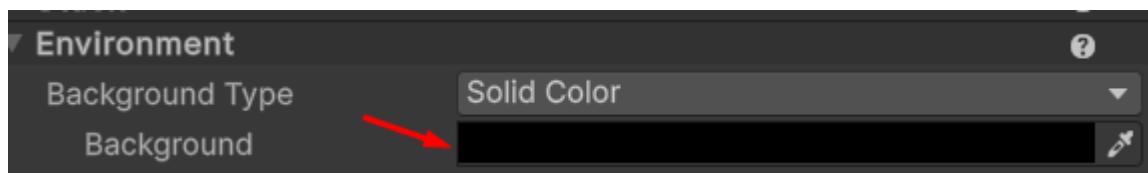
Для нашего **Canvas** меняем отображение на **World Space**, и добавляем нашу камеру:



Называем кнопку **ButtonExit**. Оформляем её на свой вкус и переносим в центр:



Меняем цвет **камеры** на **черный**:



Внутри **Canvas** создаём **TextMeshPro**, и называем – **FinalScoreText**:

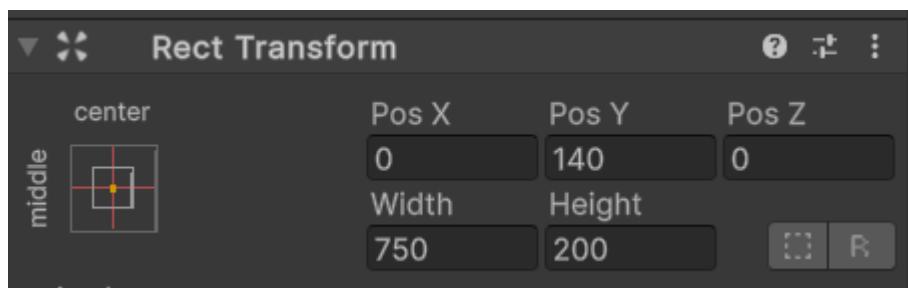


В тексте пишем:

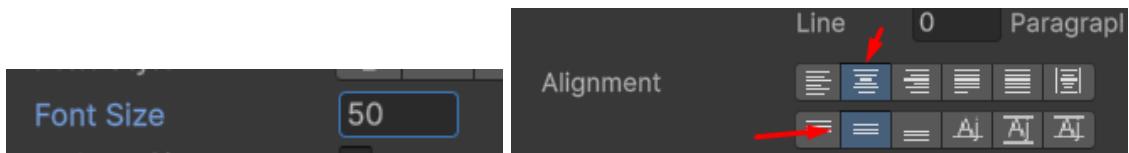
You've lost!

Your score:

И расположим его по середине экрана:



Выравнивание можно сделать по **центру** и размер шрифта установить на **50**:



26. 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:

Bubblez Graffiti Regular 400

УОУЩВ ҃ОСЗ! УОУВ ҃СОВВ:

Нажимаем скачать:

Шрифты Онлайн > Шрифты > Bubblez Graffiti

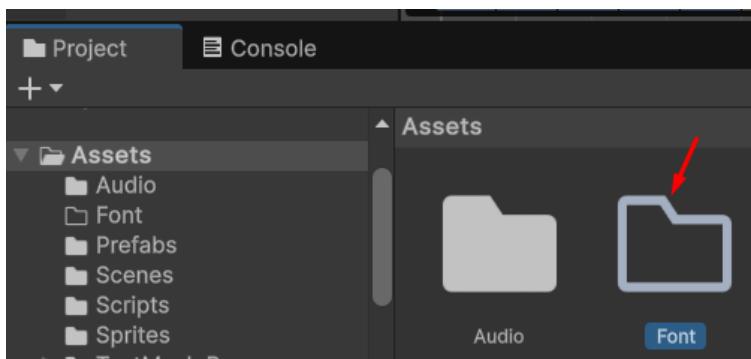
Bubblez Graffiti

Шрифт Bubblez Graffiti. Гарнитура содержит 2 файла и поддерживает 31 язык. Лицензия SIL OFL 1.1. Можно использовать в коммерческой и не коммерческой деятельности. Разработка Bubblez Graffiti велась GGBotNet.

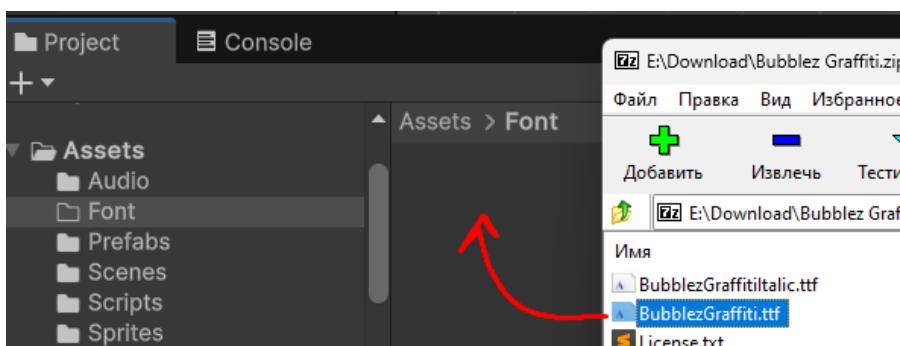
You've lost! Your score: 90px

Скачать Bubblez Graffiti

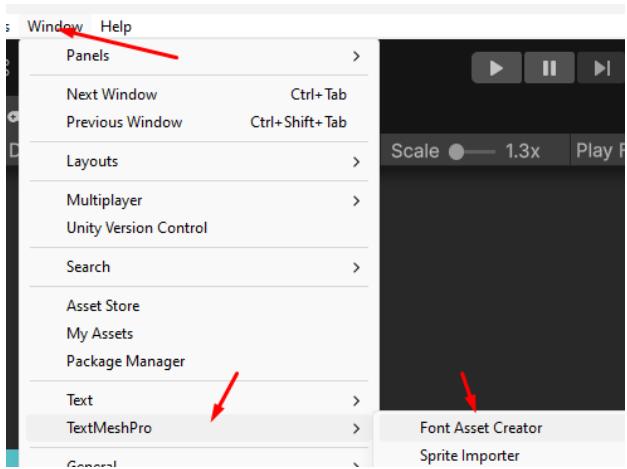
Затем создадим папку **Font**:



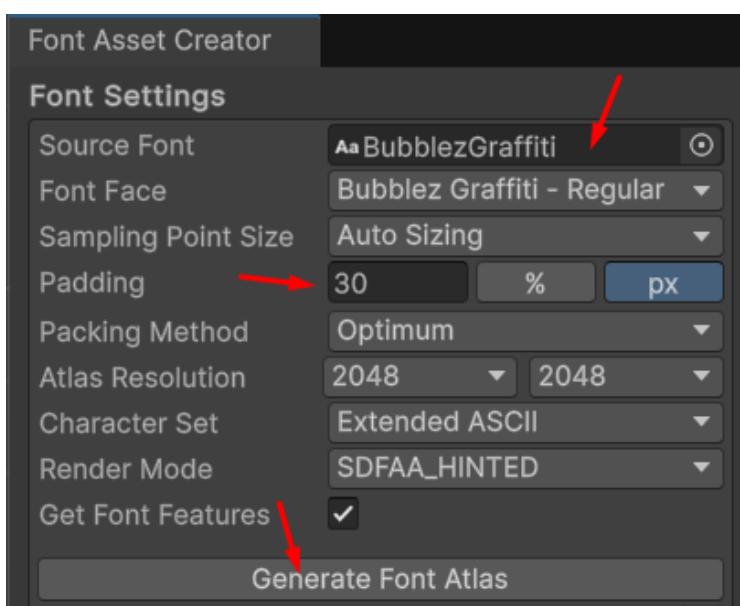
И перенесём в неё шрифт из архива (в качестве примера я возьму обычный):



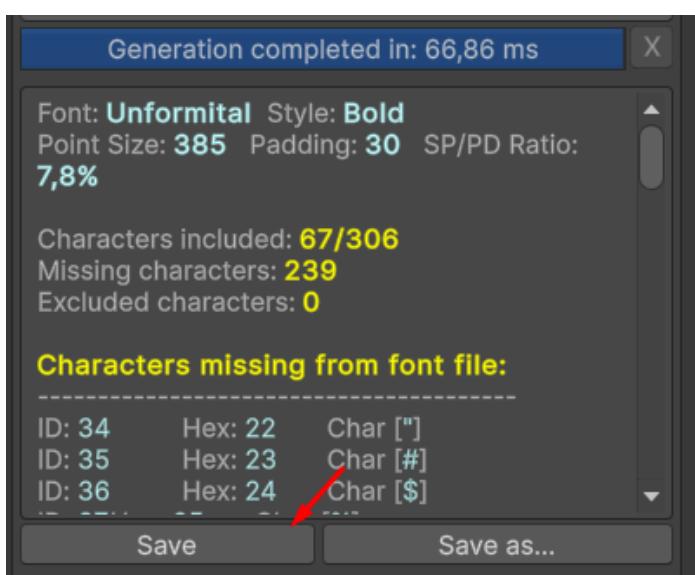
Затем нажимаем на вкладку **Window – TextMeshPro – Font Assets Creator**:



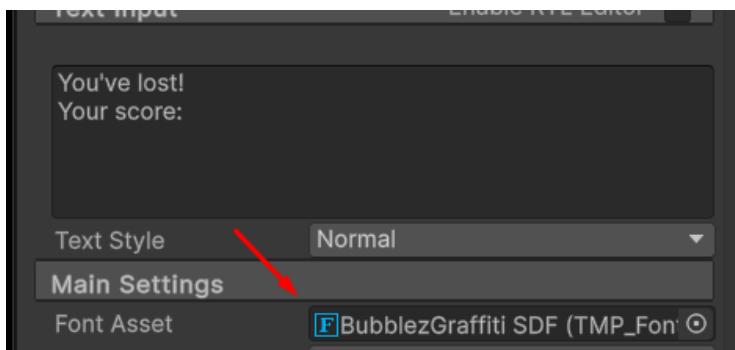
Дальше выберите **шрифт**, поставьте **Padding** на **30**, **Atlas Resolution** на **2048x2048**, и нажимаем **Generate Font Atlas**:



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**:



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

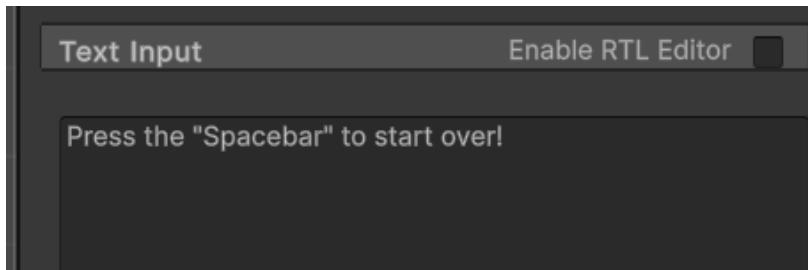
Создаём новый UI-Text-TextMeshPro:



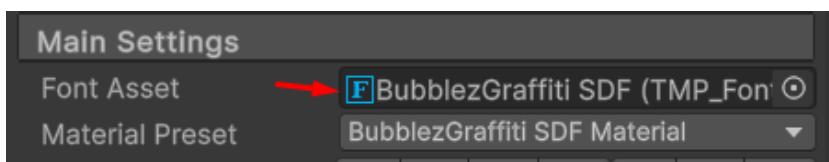
Называем его **PlayAgain**.

В тексте напишите:

Press the "Spacebar" to start over!

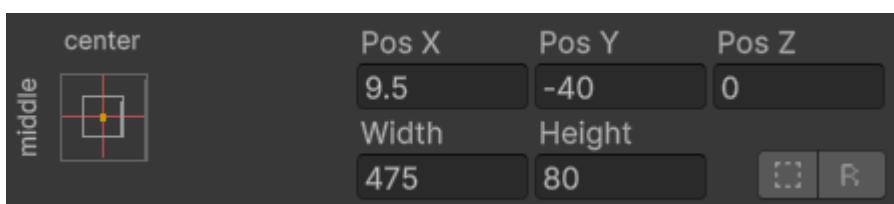


Ассет шрифта назначаем в инспекторе на Font Asset:



Оформление можете использовать как у меня, или придумать своё.

Позиции:



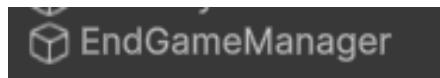
Размер шрифта 40:



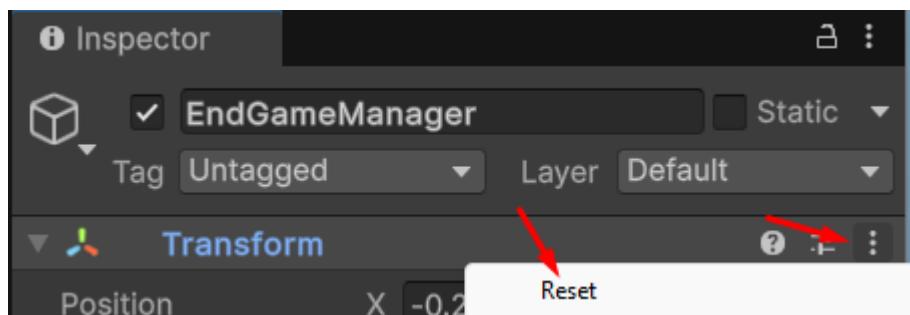
Выравнивание по центру:



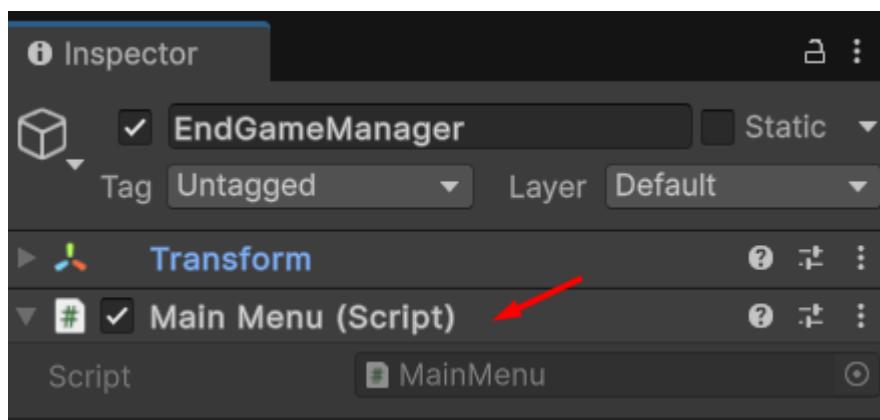
27. Создаём пустой объект и называем его **EndGameManager**:



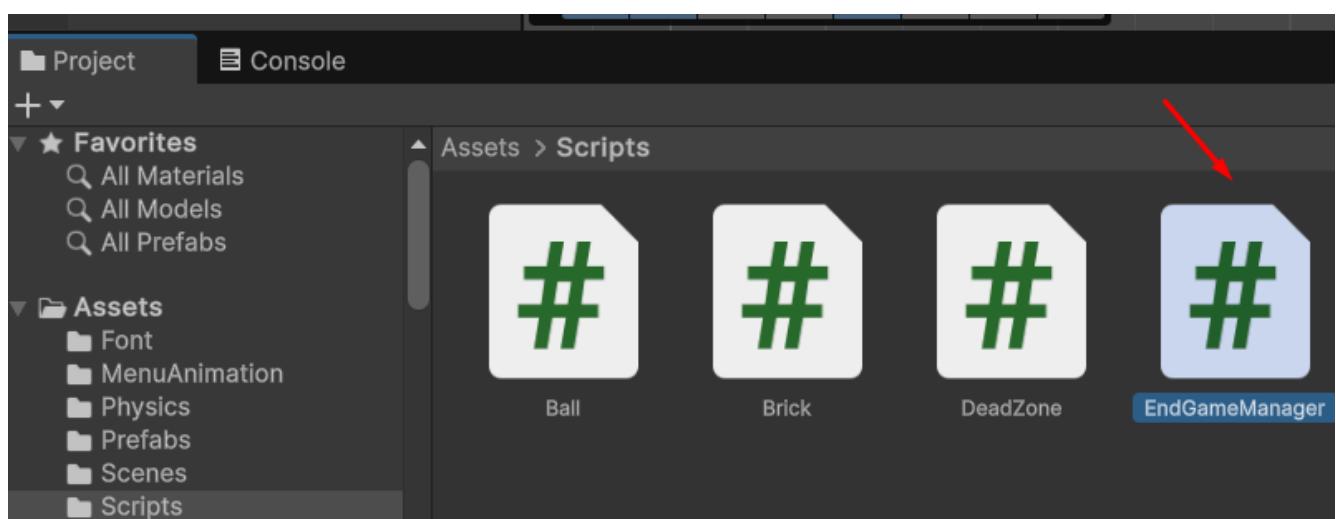
Сбрасываем для него трансформацию:



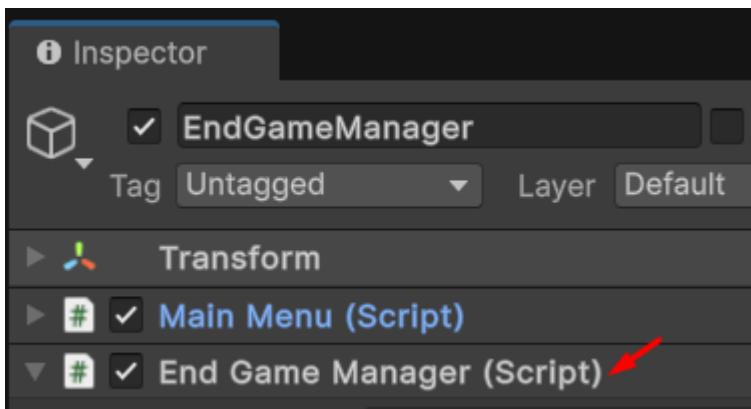
Добавляем для него скрипт **MainMenu**:



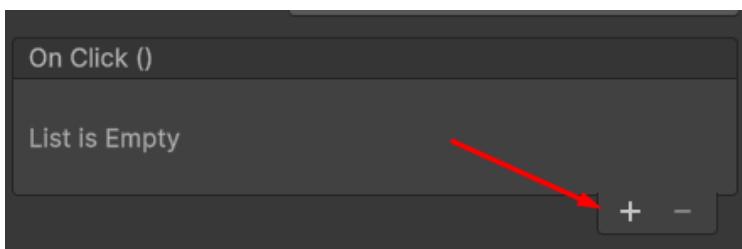
Создаём в папке **Scripts** новый скрипт **EndGameManager**:



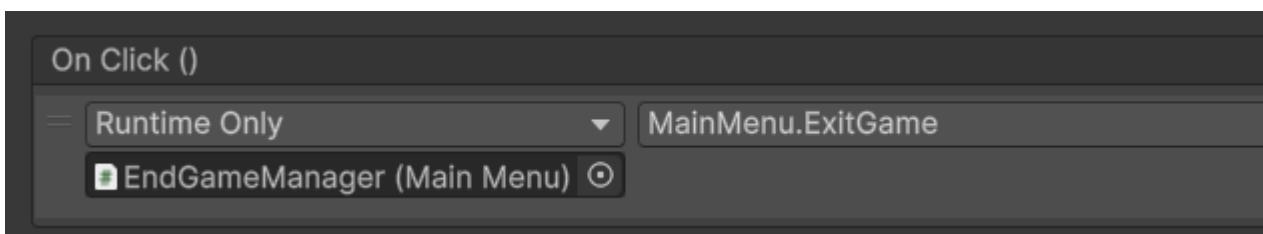
Переносим его на наш объект **EndGameManager**:



Выбираем нашу кнопку **ButtonExit**. Добавляем для неё новый клик нажав на +:



Переносим объект **EndGameManager**, и выбираем функцию **MainMenu – ExitGame()**:

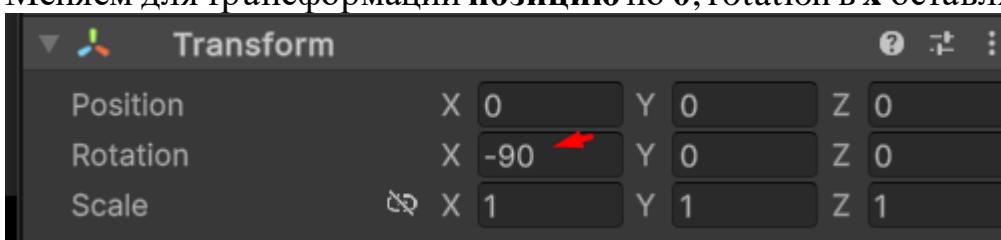


28. Добавим на сцену **систему частиц**.

Щёлкаем правой кнопкой мыши, выбираем **Effects – Particle System**:



Меняем для трансформации позицию по 0, rotation в x оставляем -90:



Duration (продолжительность) меняем на 1:



Start Speed выставляем в 0:



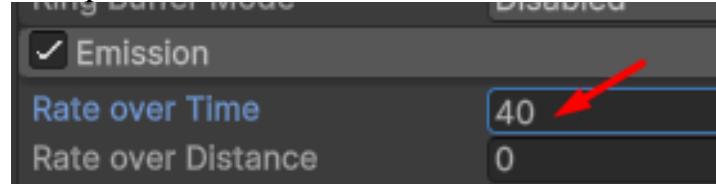
Start Size выставляем на **0.005**:



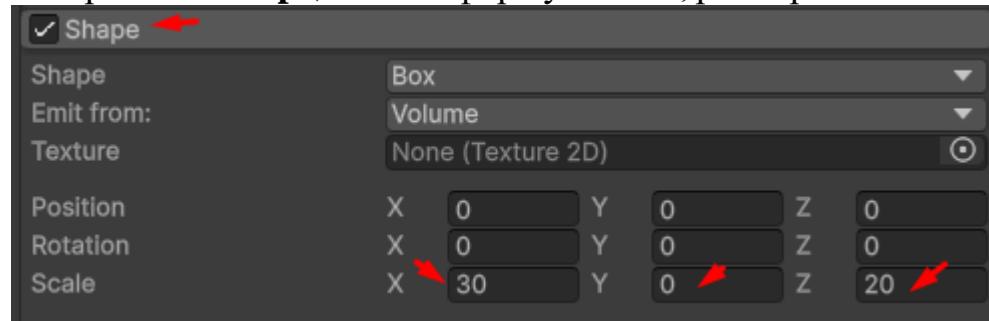
Gravity Source меняем на **2D Physics**:



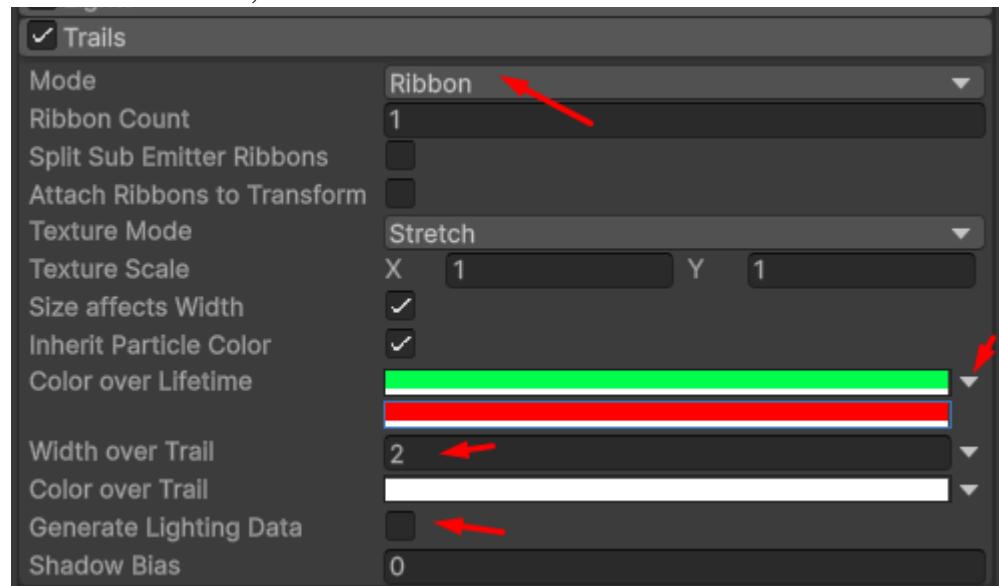
Раскрываем **Emission** и меняем **Rate over Time** на **40**:



Раскрываем **Shape**, меняем форму на **Box**, размеры выставим **30** по x и **20** по y:



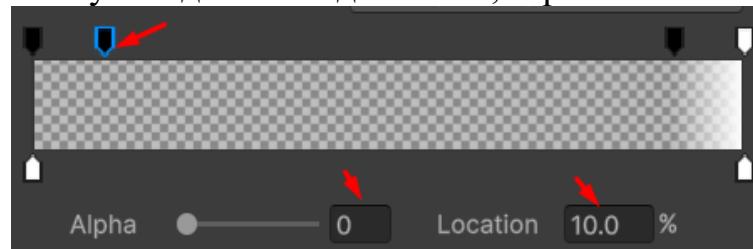
Раскрываем **Trails** и меняем **Mode** на **Ribbon**, выставляем цвета на ваш выбор, **Width over Trail = 2**, **Shadow Bias = 0**:



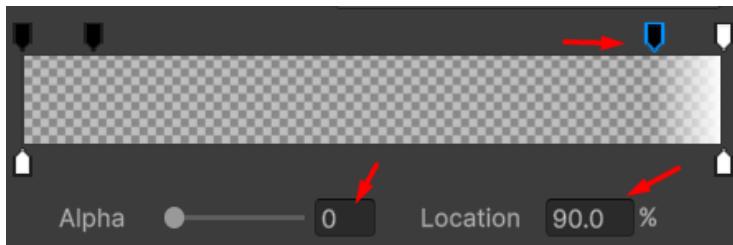
Ставим галочку у **Color over Lifetime** и нажимаем на **полосу**:



Нам нужно добавить две точки, первая на **10%** и выставить ей значение **Alpha** в **0**:



У второй точки локацию ставим на **90%** и значение **Alpha** выставляем также в **0**:

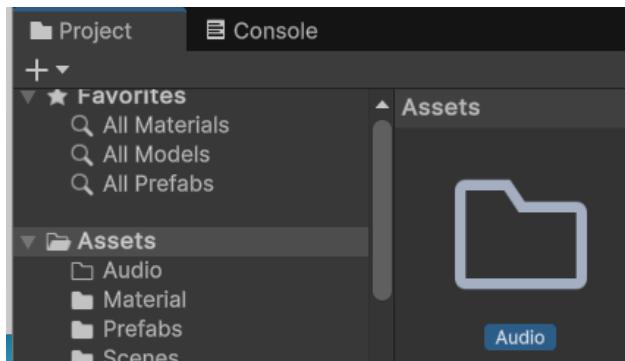


29. Добавим фоновый звуки для меню, разных уровней и меню окончания игры, а также звуки столкновения мяча с ракеткой, стенами и кирпичами. Лучше использовать **централизованную систему управления звуками — AudioManager**.

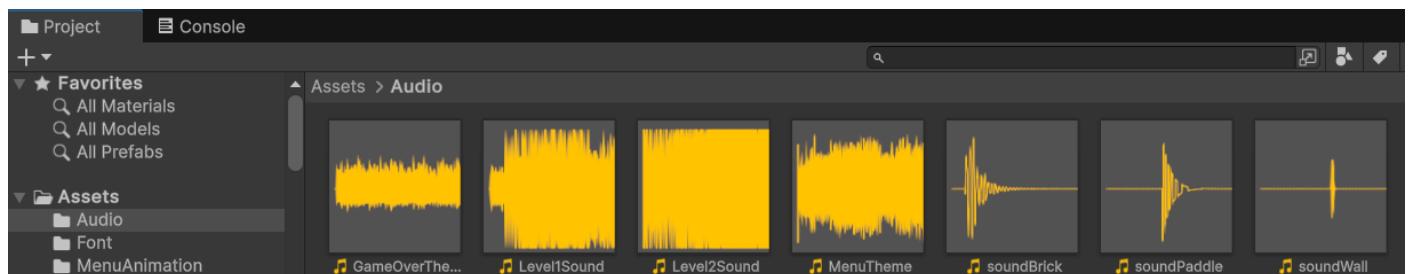
◆ Почему AudioManager — лучший вариант?

- Удобное управление звуками в одном месте.
- Избегаем дублирования **AudioSource** на каждом объекте.
- Легко добавлять новые звуки.
- Можно управлять громкостью и менять музыку на разных уровнях.

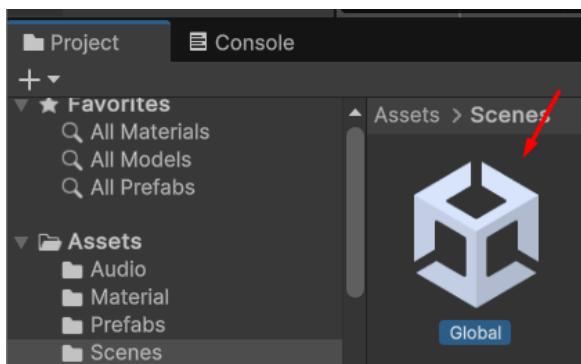
Создаём в ассетах папку **Audio**:



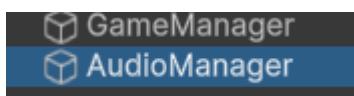
И переносим в неё звуки из архива с музыкой:



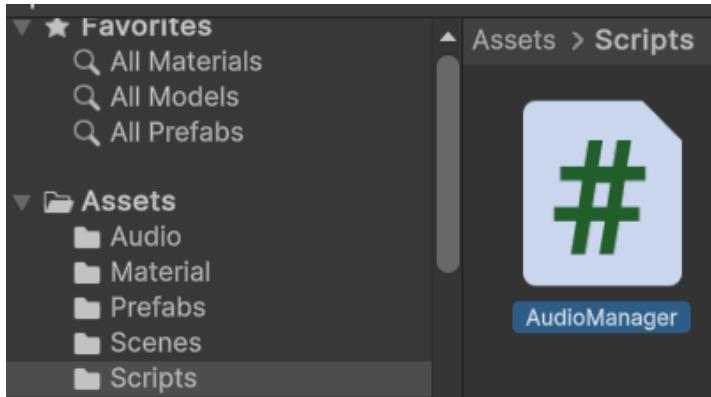
Переходим в нашу глобальную сцену **Global**:



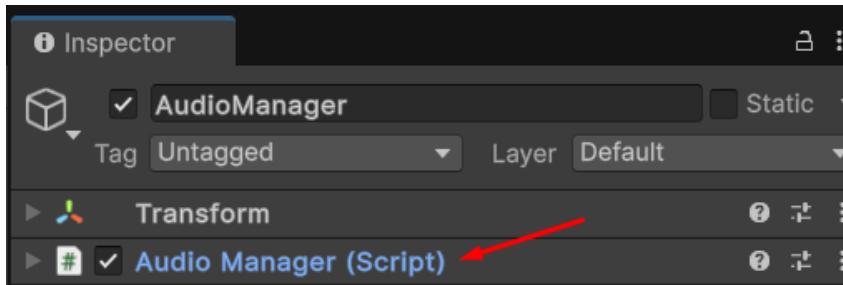
Создаём пустой объект в Hierarchy, называем его **AudioManager**:



Создаём новый скрипт **AudioManager** в папке **Scripts**:



Добавляем на наш **объект AudioManager** скрипт **AudioManager**:



Открываем его и пишем:

➤ 1: Подключение необходимых пространств имен:

```
using UnityEngine.SceneManagement; // позволяет отслеживать  
смену сцен и менять музыку.
```

➤ 2: Создание Singleton

```
public static AudioManager Instance { get; private set; }
```

Instance – статическая переменная, которая содержит ссылку на единственный экземпляр

AudioManager.

get; private set; – позволяет читать экземпляр из других скриптов, но запрещает изменять его.

Справочная информация.

Singleton (Одиночка) – это **шаблон проектирования**, который гарантирует, что **существует только один экземпляр класса**, и предоставляет **глобальную точку доступа** к нему.

В **Unity** этот шаблон часто используется для **глобальных менеджеров**, таких как

☒ **GameManager** – управление логикой игры.

☒ **AudioManager** – управление музыкой и звуками.

UIManager – управление интерфейсом.

SaveManager – управление сохранениями.

Как работает Singleton?

- 1) При первом запуске создаётся экземпляр класса.
- 2) При последующих попытках создать новый экземпляр – возвращается уже существующий объект.
- 3) Объект сохраняется между сценами (если используется DontDestroyOnLoad).

Простой пример Singleton в Unity (его не нужно писать, это лишь пример):

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    //Статическая переменная Instance содержит единственный
    //экземпляр класса.
    public static GameManager Instance { get; private set; }

    private void Awake() // Проверяем, есть ли уже Instance
    {
        if (Instance == null) // Если Instance == null
        {
            Instance = this; // Создаём его
            DontDestroyOnLoad(gameObject); // Сохраняем объект при
            //смене сцен
        }
        else // Если экземпляр уже существует
            Destroy(gameObject); // Удаляем дубликат
    }
}
```

➤ 3: Объявление переменных для звука

```
[Header("Компоненты")]
// отвечает за фоновую музыку
private AudioSource musicSource;
// воспроизводит звуковые эффекты
private AudioSource sfxSource;

[Header("Настройки громкости")]
// Громкость музыки
[Range(0f, 1f)] public float musicVolume = 0.1f;
// Громкость эффектов
[Range(0f, 1f)] public float sfxVolume = 0.3f;
```

➤ 4: Добавление аудиофайлов

```
[Header("Музыка")]
public AudioClip mainMenuMusic;
public AudioClip backgroundMusicLevel1;
public AudioClip backgroundMusicLevel2;
public AudioClip gameOverMusic;
```

AudioClip – это ссылки на аудиофайлы, которые мы будем воспроизводить как музыку.

```
[Header("Звуковые эффекты")]
public AudioClip hitBrickSound; // Удар о кирпич
public AudioClip hitPaddleSound; // Удар о ракетку
public AudioClip hitWallSound; // Удар о стену
```

Эти переменные хранят звуки столкновений с объектами.

➤ 5: Настройка Singleton и создание источников звука

```
private void Awake()
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else
    {
        Destroy(gameObject);
        return;
    }
    // Создаём два источника звука
    musicSource = gameObject.AddComponent< AudioSource >();
    sfxSource = gameObject.AddComponent< AudioSource >();
    // Настройки источников
    musicSource.loop = true;
    musicSource.playOnAwake = false;
    musicSource.volume = musicVolume;

    sfxSource.playOnAwake = false;
    sfxSource.volume = sfxVolume;
}
```

- Проверяем, существует ли уже экземпляр **AudioManager**. Если да – уничтожаем дубликат.
- **DontDestroyOnLoad(gameObject);** – сохраняем объект при смене сцен.

- Создаём два **AudioSource**:
 - musicSource – для фона (зацикленный).
 - sfxSource – для эффектов (короткие звуки).

➤ 6: Воспроизведение звуковых эффектов

```
public void PlaySound(AudioClip clip)
{
    if (clip != null)
    {
        sfxSource.PlayOneShot(clip, sfxVolume);
    }
}
```

*Метод **PlayOneShot** проигрывает разовый звуковой эффект без прерывания текущего звука.*

➤ 7: Воспроизведение музыки в зависимости от уровня

```
public void PlayMusic(int level)
{
    AudioClip newMusic = null;
    switch (level)
    {
        case 1: // Главное меню
            newMusic = mainMenuMusic;
            break;
        case 2: // Уровень 1
            newMusic = backgroundMusicLevel1;
            break;
        case 3: // Уровень 2
            newMusic = backgroundMusicLevel2;
            break;
        case 4: // Конец игры
            newMusic = gameOverMusic;
            break;
    }
    // Если уже играет нужная музыка, не перезапускаем её
    if (musicSource.clip == newMusic && musicSource.isPlaying)
    {
        return;
    }
    musicSource.clip = newMusic;
    musicSource.volume = musicVolume;
    musicSource.Play();
}
```

- Используем **switch-case** для выбора музыки в зависимости от уровня. Почему case начинается с 1? Потому что в нашем листе сцен меню имеет номер 1:

✓ Scenes/Global	0
✓ Scenes/MainMenu	1
✓ Scenes/Level1	2
✓ Scenes/Level2	3
✓ Scenes/EndGameScene	4

- Если уже играет нужная музыка, **не перезапускаем её** (чтобы избежать повторного запуска).

➤ 8: Остановка музыки

```
public void StopMusic()
{
    musicSource.Stop();
}
```

➤ 9: Отслеживание смены сцен и смена музыки

```
private void OnEnable()
{
    SceneManager.sceneLoaded += OnSceneLoaded;
}

private void OnDisable()
{
    SceneManager.sceneLoaded -= OnSceneLoaded;
}
```

- Подписываемся на **событие загрузки сцены** при включении объекта.
- Отписываемся от события при выключении, чтобы избежать утечек памяти.

➤ 10: Обработка загрузки новой сцены

```
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    if (Instance != null)
    {
        int level = SceneManager.GetActiveScene().buildIndex;
        Instance.PlayMusic(level);
    }
}
```

- Получаем индекс текущей сцены.
- Автоматически **переключаем музыку** при смене уровня.

Итоговый код:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class AudioManager : MonoBehaviour
{
    public static AudioManager Instance { get; private set; }

    [Header("Компоненты")]
    private AudioSource musicSource;
    private AudioSource sfxSource;

    [Header("Настройки громкости")]
    [Range(0f, 1f)] public float musicVolume = 0.1f
    [Range(0f, 1f)] public float sfxVolume = 0.3f;

    [Header("Музыка")]
    public AudioClip mainMenuMusic;
    public AudioClip backgroundMusicLevel1;
    public AudioClip backgroundMusicLevel2;
    public AudioClip gameOverMusic;

    [Header("Звуковые эффекты")]
    public AudioClip hitBrickSound;
    public AudioClip hitPaddleSound;
    public AudioClip hitWallSound;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }

        // Создаём два источника звука
        musicSource = gameObject.AddComponent<AudioSource>();
        sfxSource = gameObject.AddComponent<AudioSource>();
```

```
// Настройки источников
musicSource.loop = true;
musicSource.playOnAwake = false;
musicSource.volume = musicVolume;

sfxSource.playOnAwake = false;
sfxSource.volume = sfxVolume;
}

public void PlaySound(AudioClip clip)
{
    if (clip != null)
    {
        sfxSource.PlayOneShot(clip, sfxVolume);
    }
}

public void PlayMusic(int level)
{
    AudioClip newMusic = null;

    switch (level)
    {
        case 1: // Главное меню
            newMusic = mainMenuMusic;
            break;
        case 2: // Уровень 1
            newMusic = backgroundMusicLevel1;
            break;
        case 3: // Уровень 2
            newMusic = backgroundMusicLevel2;
            break;
        case 4: // Конец игры
            newMusic = gameOverMusic;
            break;
    }

    // Если уже играет нужная музыка, не перезапускаем её
    if (musicSource.clip == newMusic &&
musicSource.isPlaying)
    {
        return;
    }
}
```

```

        musicSource.clip = newMusic;
        musicSource.volume = musicVolume;
        musicSource.Play();
    }

    public void StopMusic()
    {
        musicSource.Stop();
    }

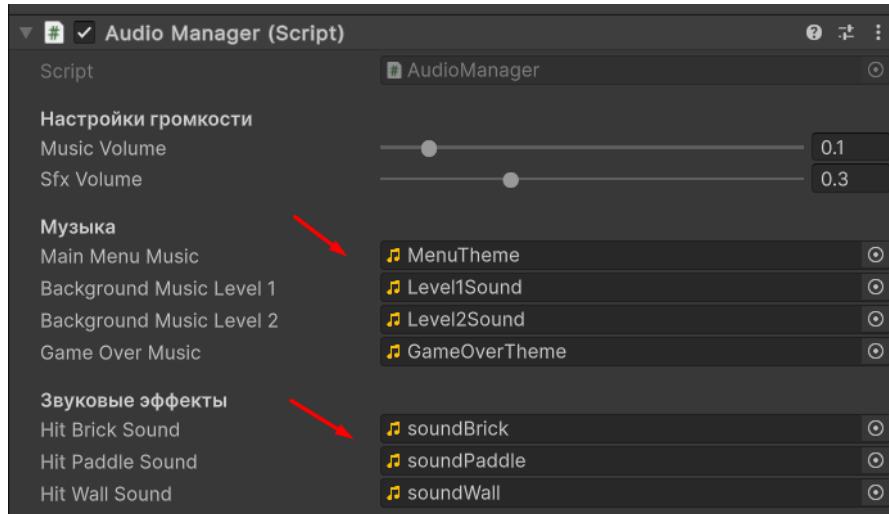
    private void OnEnable()
    {
        SceneManager.sceneLoaded += OnSceneLoaded;
    }

    private void OnDisable()
    {
        SceneManager.sceneLoaded -= OnSceneLoaded;
    }

    private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        if (Instance != null)
        {
            int level =
SceneManager.GetActiveScene().buildIndex;
            Instance.PlayMusic(level);
        }
    }
}

```

В проекте и добавляем соответствующие звуковые эффекты и музыку в поля:



30. Теперь добавим соответствующие звуки удара мячика для скриптов.

Переходим в скрипт **Paddle** и внесём правки в метод **OnCollisionEnter2D**:

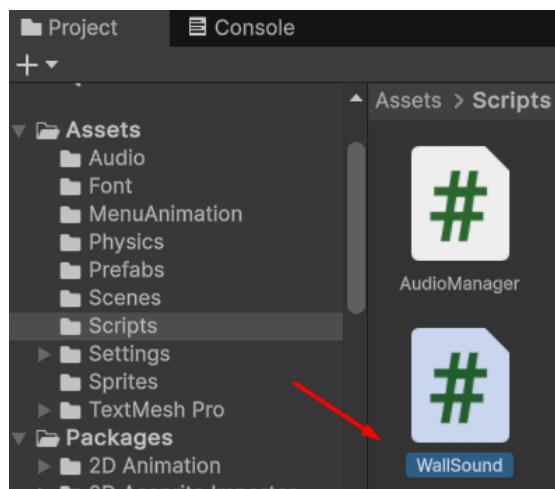
```
private void OnCollisionEnter2D(Collision2D collision)
{
    Ball ball = collision.gameObject.GetComponent<Ball>();

    if (ball != null)
    {
        BallBounce(ball, collision);
    }
    if (AudioManager.Instance != null) // проверка на null
    {
        AudioManager.Instance.PlaySound(AudioManager.Instance.hitPaddleSound); // проигрываем звук удара
    }
}
```

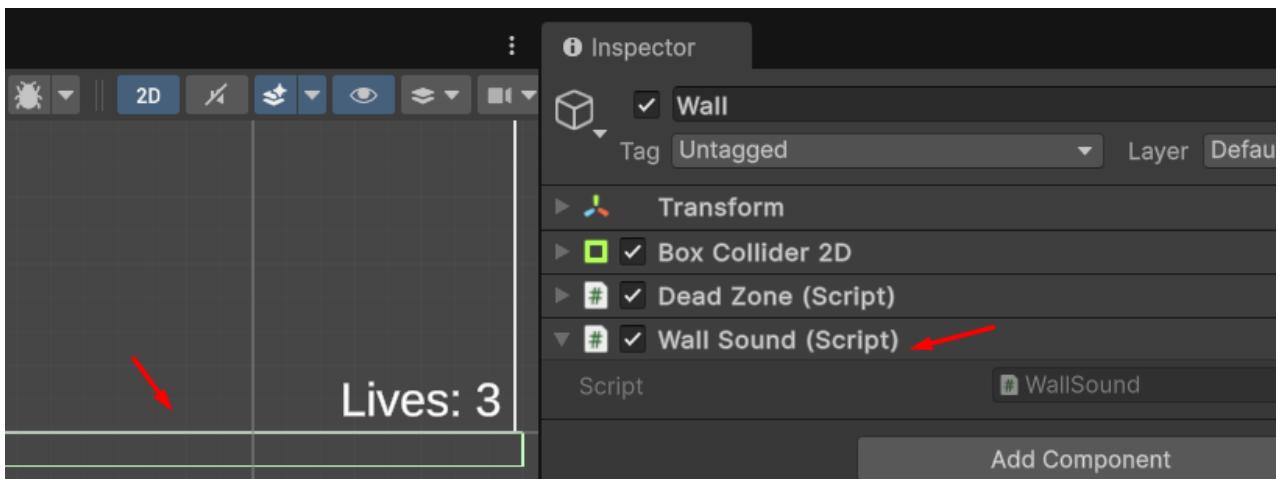
Тоже самое делаем для наших кирпичей в скрипте **Brick**, в методе **OnCollisionEnter2D** внесём правки:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Ball"))
    {
        Hit();
        if (AudioManager.Instance != null) // проверка на null
        {
            AudioManager.Instance.PlaySound(AudioManager.Instance.hitBrickSound); // вызываем звук удара
        }
    }
}
```

Для стен мы создадим новый скрипт **WallSound** в папке **Scripts**:



И прикрепляем скрипт на наш объект – нижнюю стену Wall в Level1:



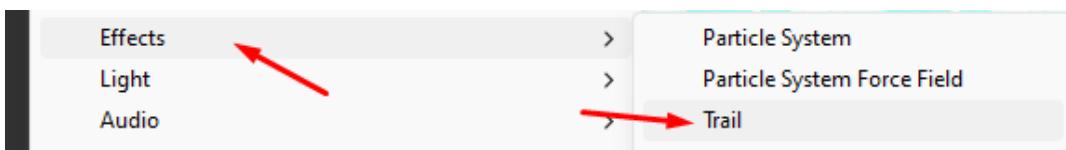
Открываем его и пишем:

```
using UnityEngine;

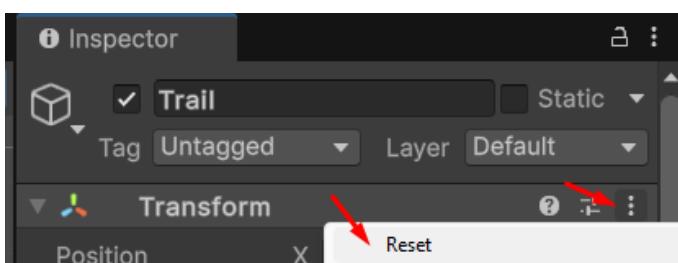
public class WallSound : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Ball"))
        {
            if (AudioManager.Instance != null)
            {
                AudioManager.Instance.PlaySound(AudioManager.Instance.
hitWallSound);
            }
        }
    }
}
```

31. Теперь создадим след от хвоста для нашего мячика.

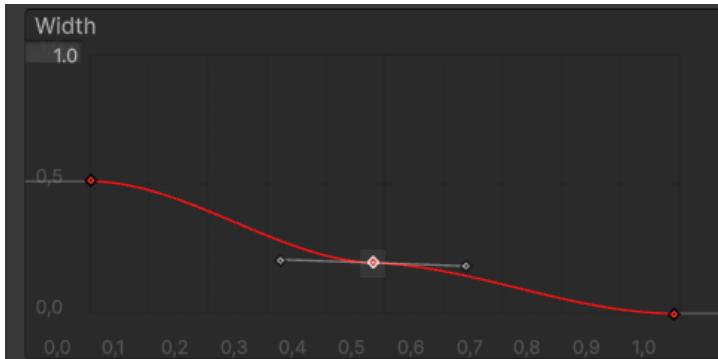
В иерархии щёлкаем правой кнопкой мыши и выбираем Effects – Trail:



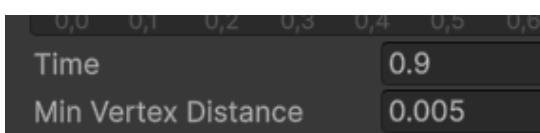
Сбрасываем позицию:



Далее настроим ширину нашего хвоста на затухание с размера **0.5** следующим образом:



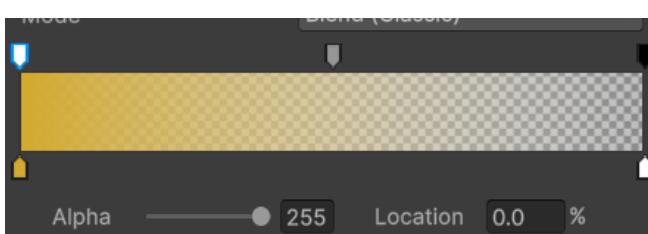
Time выставим на **0.9**, **Min Vertex Distance** на **0.005**:



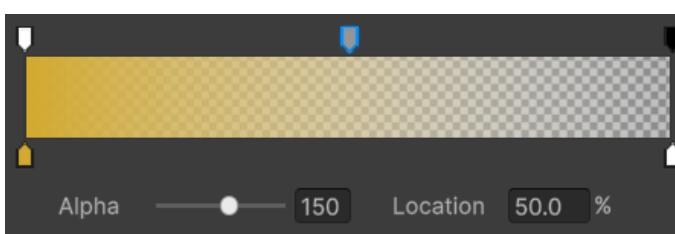
Color настроим следующим образом:



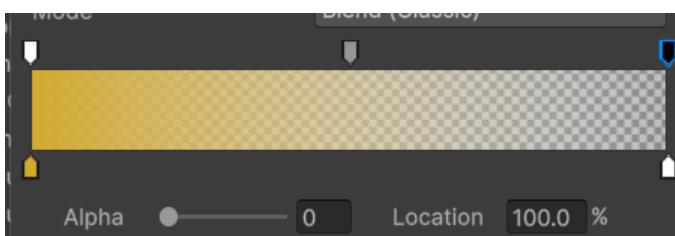
1 точка Alpha – **255**, Location **0%**:



2 точка Alpha – **150**, Location **50%**:



3 точка Alpha – **0**, Location **100%**:



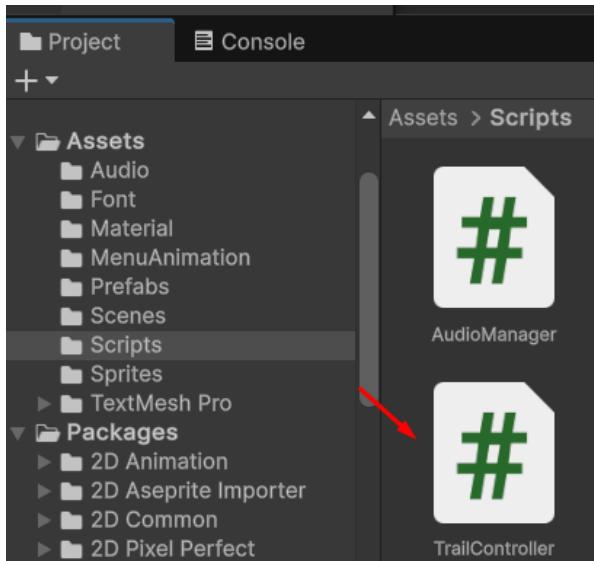
Texture Mode – **Tile**, **Texture Scale** – **4**:



Переносим наш **Trail** внутрь объекта **Ball**:



Создаём скрипт **TrailController**



Его логика будет заключаться в следующем:

- ✓ Отслеживает движение объекта.
- ✓ Включает TrailRenderer, если объект двигается.
- ✓ Выключает TrailRenderer, если объект остановился.
- ✓ Очищает след при необходимости.

Открываем его.

➤ **Объявляем переменные**

```
// ссылка на компонент TrailRenderer, который рисует след
public TrailRenderer trailRenderer;
// минимальное расстояние, которое объект должен пройти, чтобы
// след активировался.
public float movementThreshold = 0.1f;
// хранит последнюю позицию объекта, чтобы отслеживать его
// движение
private Vector3 lastPosition;
```

➤ В методе **Start()** запоминаем начальную позицию объекта:

```
private void Start()
{
    lastPosition = transform.position;
}
```

➤ Напишем метод **Update()**, который будет **включать** или **отключать** след:

```

private void Update()
{
    // Включаем или выключаем Trail Renderer в зависимости от
    // движения
    if (Vector3.Distance(transform.position, lastPosition) >
movementThreshold)
    {
        if (!trailRenderer.emitting)
        {
            trailRenderer.emitting = true; // Включить след
        }
    }
    else
    {
        if (trailRenderer.emitting)
        {
            trailRenderer.emitting = false; // Выключить след
        }
    }
    lastPosition = transform.position; // Обновить последнюю
    // позицию
}

```

➤ **Что делает этот код?**

1. **Определяет, двигался ли объект:**

- *Vector3.Distance(transform.position, lastPosition)* вычисляет расстояние между текущей позицией и предыдущей.
- *Если оно больше movementThreshold*, значит объект движется.

2. **Если объект движется, включаем TrailRenderer.**

3. **Если объект остановился, выключаем TrailRenderer:**

4. **Обновляем lastPosition, чтобы в следующем кадре снова проверить движение.**

А. Напишем метод **ClearTrail()** для очистки следа:

```

public void ClearTrail()// сброс после reset
{
    if (trailRenderer != null)
    {
        trailRenderer.Clear();
    }
}

```

➤ **Для чего нужен этот метод?**

- Полностью очищает след (*trailRenderer.Clear()*).
- Используется при **сбросе уровня** или после респауна мяча.

ИТОГОВЫЙ КОД:

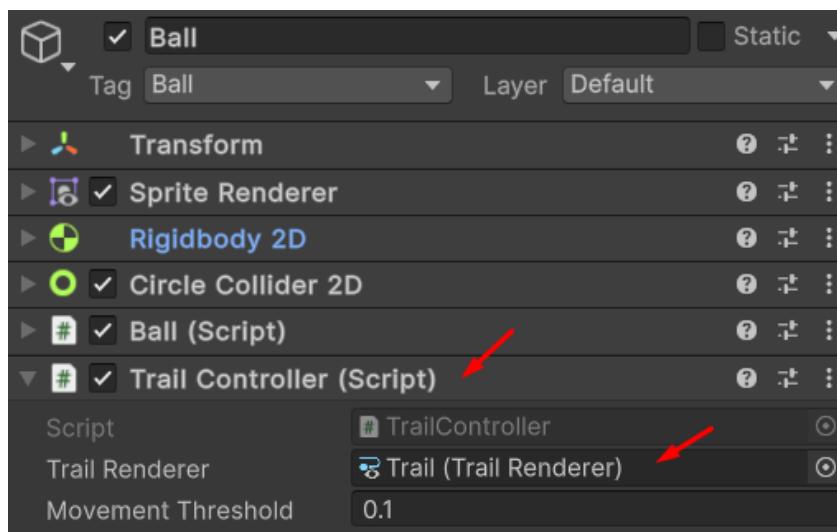
```
using UnityEngine;
public class TrailController : MonoBehaviour
{
    public TrailRenderer trailRenderer;
    public float movementThreshold = 0.1f;
    private Vector3 lastPosition;

    private void Start()
    {
        lastPosition = transform.position;
    }

    private void Update()
    {
        if (Vector3.Distance(transform.position, lastPosition) > movementThreshold)
        {
            if (!trailRenderer.emitting)
            {
                trailRenderer.emitting = true;
            }
        }
        else
        {
            if (trailRenderer.emitting)
            {
                trailRenderer.emitting = false;
            }
        }
        lastPosition = transform.position;
    }

    public void ClearTrail()
    {
        if (trailRenderer != null)
        {
            trailRenderer.Clear();
        }
    }
}
```

Прикрепляем скрипт к объекту **Ball** и переносим в поле мяча наш эффект:



32. Теперь перейдём к правкам для скрипта **GameManager**

1) Добавляем работу с **TrailRenderer** (эффект частиц для мяча)

- Добавляем дополнительное поле:

```
public TrailRenderer trail { get; private set; } // ссылка на  
частицы
```

- В методе **FindGameObjects()** добавляем её поиск:

```
private void FindGameObjects()  
{  
    ball = FindAnyObjectByType<Ball>();  
    paddle = FindAnyObjectByType<Paddle>();  
    bricks = FindObjectsOfType<Brick>(FindObjectsInactive.Include,  
FindObjectsSortMode.None);  
    trail = FindAnyObjectByType<TrailRenderer>(); // ищем след от  
мяча  
}
```

- В методе **ResetLevel()** очищаем след:

```
private void ResetLevel()  
{  
    ball.ResetBall();  
    paddle.ResetPaddle();  
    trail.Clear(); // Очищаем след мяча при перезапуске уровня  
}
```

2) Добавляем работу с глобальной сценой (**Global**):

```
private void Start()  
{  
    if (SceneManager.GetActiveScene().name == "Global")  
    {  
        SceneManager.LoadScene("MainMenu");  
    }  
}
```

Что изменилось:

Теперь в методе `Start()` добавлена проверка, если текущая сцена — `"Global"`, то загружается `"MainMenu"`.

Зачем это нужно:

Сцена `"Global"` теперь выполняет роль сцены-менеджера, которая не участвует в игровом процессе. Это полезно, если `"GameManager"` создаётся в этой сцене и затем остаётся активным на протяжении всей игры.

3) Улучшаем систему перехода между уровнями в методе `CheckLevelClear()`:

```
private void CheckLevelClear()
{
    if (Cleared())
    {
        int nextLevel = currentLevel + 1;

        if (Application.CanStreamedLevelBeLoaded("Level" +
nextLevel))
        {
            LoadLevel(nextLevel);
        }
        else
        {
            SaveFinalScore();
            SceneManager.LoadScene("EndGameScene");
        }
    }
}
```

Что изменилось:

Теперь перед загрузкой следующего уровня проверяется, существует ли он (`Application.CanStreamedLevelBeLoaded()`). Если уровней больше нет, игра переходит на сцену окончания (`EndGameScene`).

Зачем это нужно:

Это предотвращает ошибки, если, например, игрок завершил последний уровень, и игра не знает, что делать дальше.

4) Добавляем сохранение итогового счёта (`SaveFinalScore()`)

```
private void SaveFinalScore()
{
    PlayerPrefs.SetInt("FinalScore", score);
    PlayerPrefs.Save();
}
```

Что изменилось:

Теперь очки сохраняются перед выходом в сцену конца игры (**EndGameScene**) с помощью **PlayerPrefs**.

Зачем это нужно:

Чтобы игрок видел свой итоговый результат на экране окончания игры.

5) Добавляем отображение финального счёта (**DisplayFinalScore()**)

```
private void DisplayFinalScore()
{
    GameObject finalScoreObject = GameObject.Find("FinalScoreText");
    if (finalScoreObject != null)
    {
        TMP_Text finalScoreText = finalScoreObject.GetComponent<TMP_Text>();
        finalScoreText.text = "Final Score: " +
PlayerPrefs.GetInt("FinalScore", 0);
    }
}
```

Что изменилось:

Теперь при загрузке сцены "**EndGameScene**" выводится сообщение с итоговым счётом игрока.

Зачем это нужно:

Игрок может увидеть свои достижения после завершения игры.

6) Добавляем возможность перезапуска игры по нажатию **Space**

```
private void Update()
{
    if (SceneManager.GetActiveScene().name == "EndGame" &&
Input.GetKeyDown(KeyCode.Space))
    {
        ResetGame();
    }
}
```

Что изменилось:

В сцене "**EndGameScene**", если игрок нажимает *Space*, игра сбрасывает состояние и загружает "**Level1**".

Зачем это нужно:

Упрощает повторное прохождение игры без необходимости вручную возвращаться в главное меню.

7) Обновляем методы **ResetGame()** и **ResetGameState()**, которые будут сбрасывать все игровые параметры, удалять сохранённые очки и перезапускать первый уровень:

```

private void ResetGame()
{
    ResetGameState();
    PlayerPrefs.DeleteKey("FinalScore");
    SceneManager.LoadScene("Level1");
}

public void ResetGameState()
{
    score = 0;
    lives = 3;
    speedMultiplier = 1.0f;
    currentLevel = 1;
}

```

8) Обновляем метод **Miss()**:

```

public void Miss()
{
    lives--;
    UpdateScoreText();
    if (lives > 0)
    {
        ResetLevel();
    }
    else
    {
        SaveFinalScore();
        SceneManager.LoadScene("EndGameScene");
    }
}

```

Итоговый код для GameManager:

```

using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{

    [Header("Игровые параметры")]
    [SerializeField] private int currentLevel = 1;
    [SerializeField] private int score = 0;
    [SerializeField] private int lives = 3;
    [SerializeField] private float speedMultiplier = 1.0f;

    [Header("Текстовые объекты")]
    private TMP_Text scoreText;
    private TMP_Text livesText;
}

```

```
[Header("Игровые объекты")]
public Ball ball { get; private set; }
public Paddle paddle { get; private set; }
public Brick[] bricks { get; private set; }
public TrailRenderer trail { get; private set; }
private void Awake()
{
    DontDestroyOnLoad(gameObject);
    SceneManager.sceneLoaded += OnSceneLoaded;
}
private void Start()
{
    if (SceneManager.GetActiveScene().name == "Global")
    {
        SceneManager.LoadScene("MainMenu");
    }
}
private void Update()
{
    if (SceneManager.GetActiveScene().name == "EndGame" &&
Input.GetKeyDown(KeyCode.Space))
    {
        ResetGame();
    }
}
private void FindGameObjects()
{
    ball = FindAnyObjectOfType<Ball>();
    paddle = FindAnyObjectOfType<Paddle>();
    bricks = FindObjectsOfType<Brick>(FindObjectsInactive.Include,
FindObjectsSortMode.None);
    trail = FindAnyObjectOfType<TrailRenderer>();
}
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    FindScoreText();
    UpdateScoreText();
    FindLivesText();
    FindGameObjects();
    if (scene.name == "EndGame")
    {
        DisplayFinalScore();
    }
}
private void LoadLevel(int levelNumber)
{
    currentLevel = levelNumber;
    SceneManager.LoadScene("Level" + levelNumber);

    if (AudioManager.Instance != null)
    {
        AudioManager.Instance.PlayMusic(levelNumber);
    }
}
```

```

private void FindScoreText()
{
    GameObject scoreObject = GameObject.Find("Score");
    if (scoreObject != null)
        scoreText = scoreObject.GetComponent<TMP_Text>();
}
private void FindLivesText()
{
    GameObject livesObject = GameObject.Find("Lives");
    if (livesObject != null)
    {
        livesText = livesObject.GetComponent<TMP_Text>();
    }
}
private void UpdateScoreText()
{
    if (scoreText != null)
        scoreText.text = $"Score: {score}";
    if (livesText != null)
        livesText.text = $"Lives: {lives}";
}
public void Hit(Brick brick)
{
    score += brick.points;
    UpdateScoreText();
    UpdateBallSpeed();
    Invoke(nameof(CheckLevelClear), 0.1f);
}
private void CheckLevelClear()
{
    if (Cleared())
    {
        int nextLevel = currentLevel + 1;
        if (Application.CanStreamedLevelBeLoaded("Level" + nextLevel))
            LoadLevel(nextLevel);
        else
        {
            SaveFinalScore();
            SceneManager.LoadScene("EndGameScene");
        }
    }
}
private bool Cleared()
{
    foreach (Brick brick in bricks)
    {
        if (brick.gameObject.activeInHierarchy && !brick.unbreakable)
        {
            return false;
        }
    }
    return true;
}

```

```
private void UpdateBallSpeed()
{
    speedMultiplier = 1f + ((score / 500f) * 0.01f);
    Ball ball = FindAnyObjectByType<Ball>();
    if (ball != null)
        ball.SetSpeedMultiplier(speedMultiplier);
}
private void SaveFinalScore()
{
    PlayerPrefs.SetInt("FinalScore", score);
    PlayerPrefs.Save();
}

private void DisplayFinalScore()
{
    GameObject finalScoreObject = GameObject.Find("FinalScoreText");
    if (finalScoreObject != null)
    {
        TMP_Text finalScoreText =
finalScoreObject.GetComponent<TMP_Text>();
        finalScoreText.text = "Final Score: " +
PlayerPrefs.GetInt("FinalScore", 0);
    }
}

public void Miss()
{
    lives--;
    UpdateScoreText();
    if (lives > 0)
    {
        ResetLevel();
    }
    else
    {
        SaveFinalScore();
        SceneManager.LoadScene("EndGameScene");
    }
}

private void ResetLevel()
{
    ball.ResetBall();
    paddle.ResetPaddle();
    trail.Clear();
}

private void ResetGame()
{
    ResetGameState();
    PlayerPrefs.DeleteKey("FinalScore");
    SceneManager.LoadScene("Level1");
}
```

```

public void ResetGameState()
{
    score = 0;
    lives = 3;
    speedMultiplier = 1.0f;
    currentLevel = 1;
}
}

```

33. Переходим к скрипту **EndGameManager**. Открываем его и пишем:

```

using UnityEngine;
using TMPro;
using UnityEngine.SceneManagement;

public class EndGameManager : MonoBehaviour
{
    public TMP_Text scoreText; // Сюда передаём текстовое поле для
очков

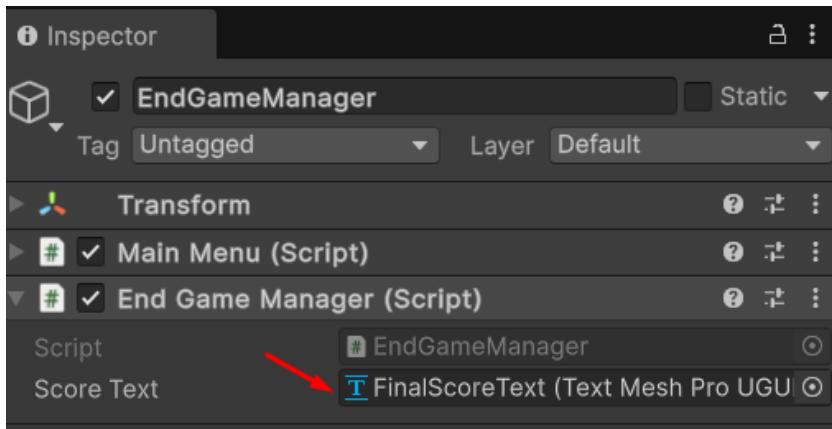
    private void Start()
    {
        int finalScore = PlayerPrefs.GetInt("FinalScore", 0); // Получаем очки
        scoreText.text = "You've lost! \nYour score: " +
finalScore; // Отображаем очки
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            PlayerPrefs.DeleteKey("FinalScore"); // Очищаем счёт

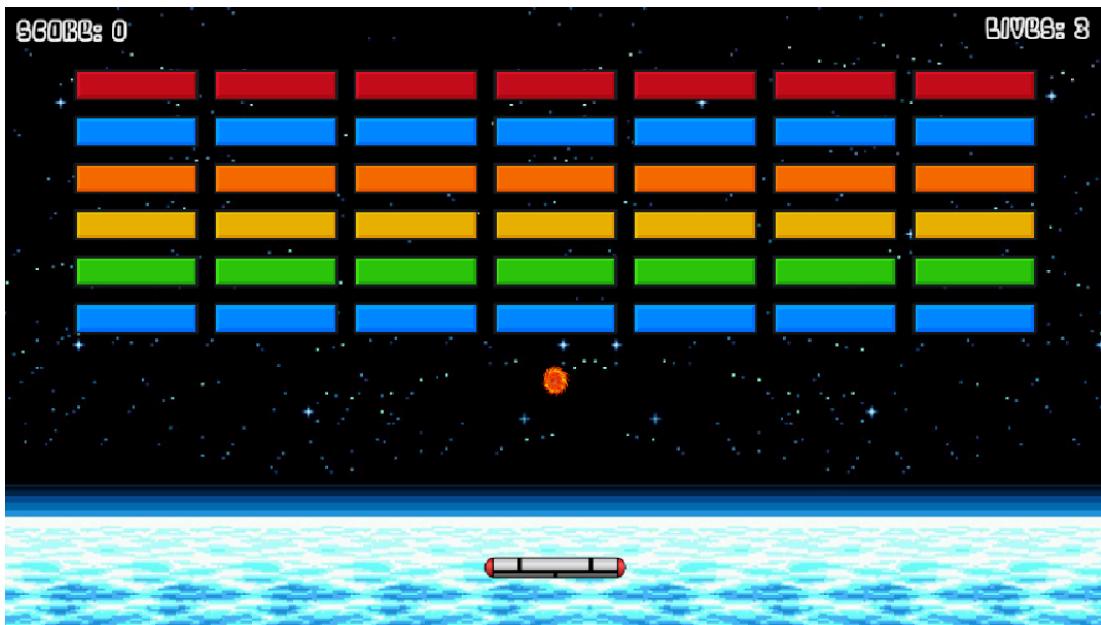
            // Получаем ссылку на GameManager и сбрасываем его
            состояния
            GameManager gameManager =
FindFirstObjectByType<GameManager>();
            if (gameManager != null)
            {
                gameManager.ResetGameState();
            }
            SceneManager.LoadScene("Level1"); // Перезапуск игры
        }
    }
}

```

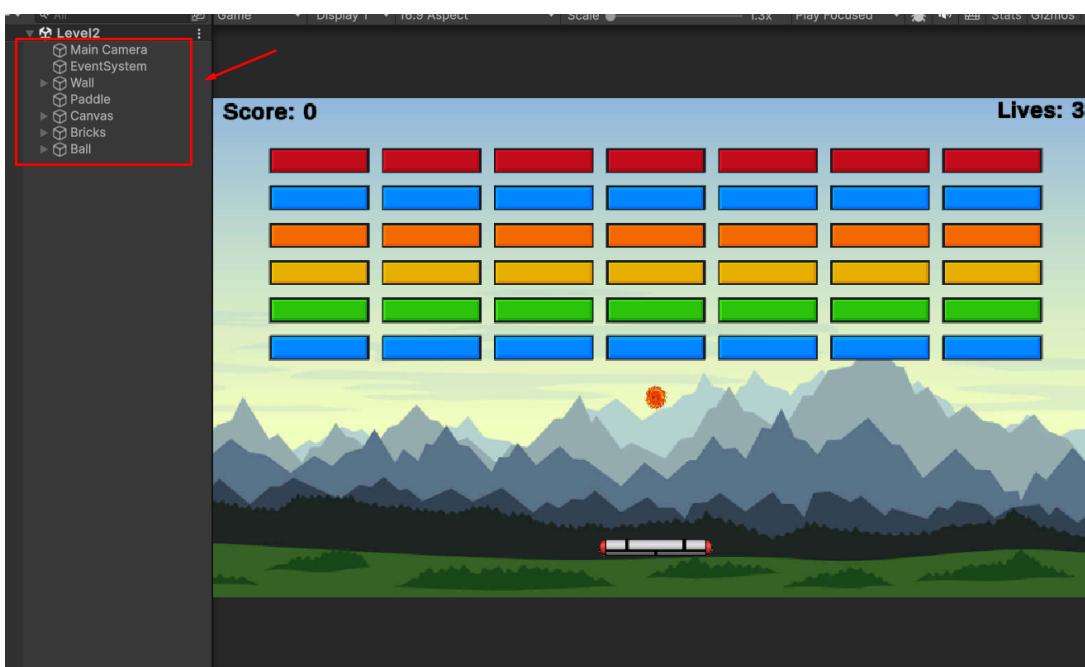
Открываем сцену **EndGameScene** и в объект **EndGameScene** переносим наш текст:



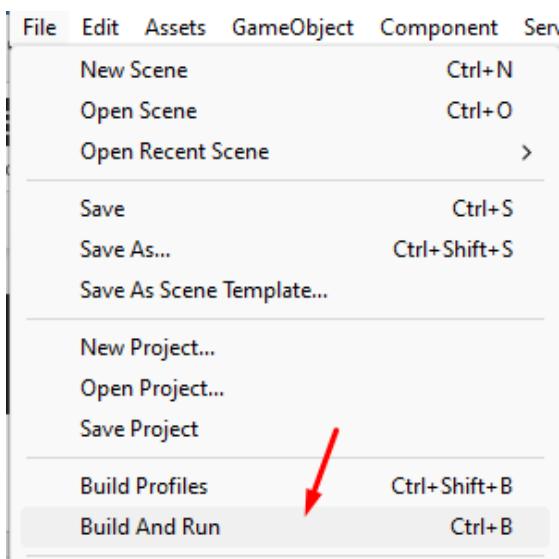
34. Теперь вы можете настроить внешний вид для вашего **уровня 1**. Вот вам пример реализации:



Также перейдите на **уровень 2**, и скопируйте на него объекты с первого уровня, и расставьте их в другом порядке:



35. 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.