

Лабораторная работа. Создание 2D-игры Dino Game

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

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

Задачи:

1. Настройка окружения:

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

2. Создание игровых объектов:

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

3. Разработка игровой логики:

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

4. Улучшение игровых механик:

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

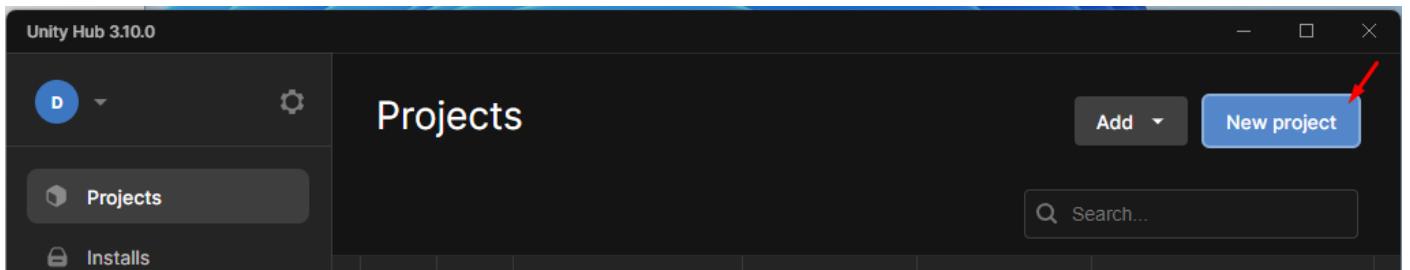
5. Интерфейс пользователя и звуковое сопровождение:

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

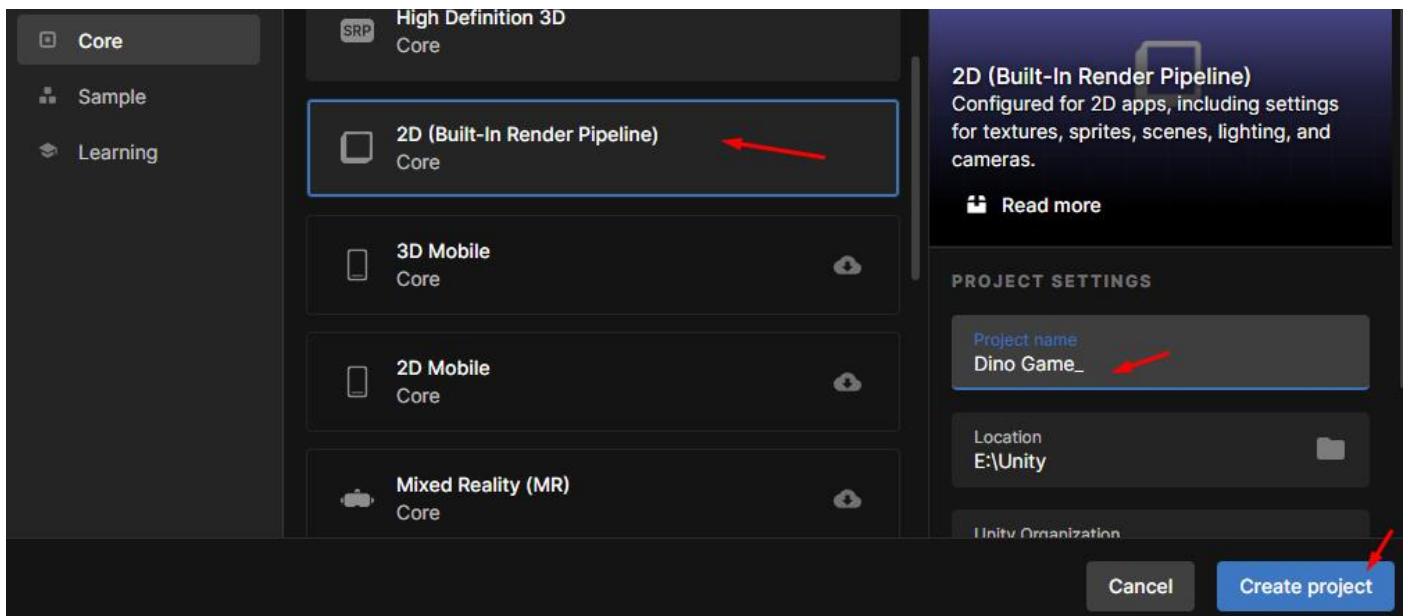
6. Финальные настройки и сборка проекта:

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

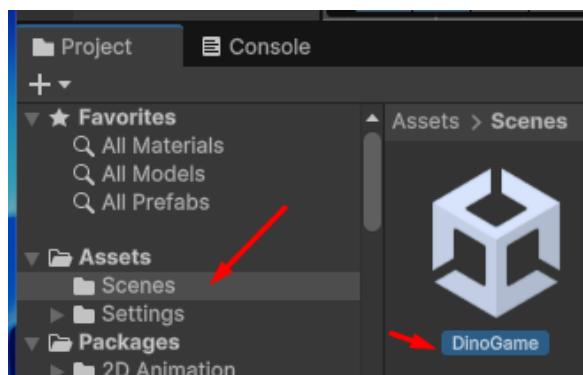
1. Запускаем Unity Hub.



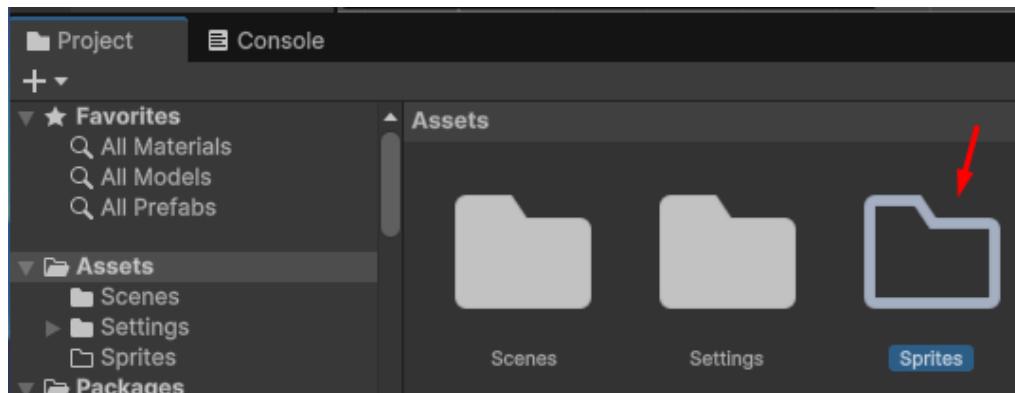
Выбираем **2D (Built-In Render Pipeline)**, вводим название проекта - **Dino Game**, выбираем место расположения, и нажимаем **Create project**:



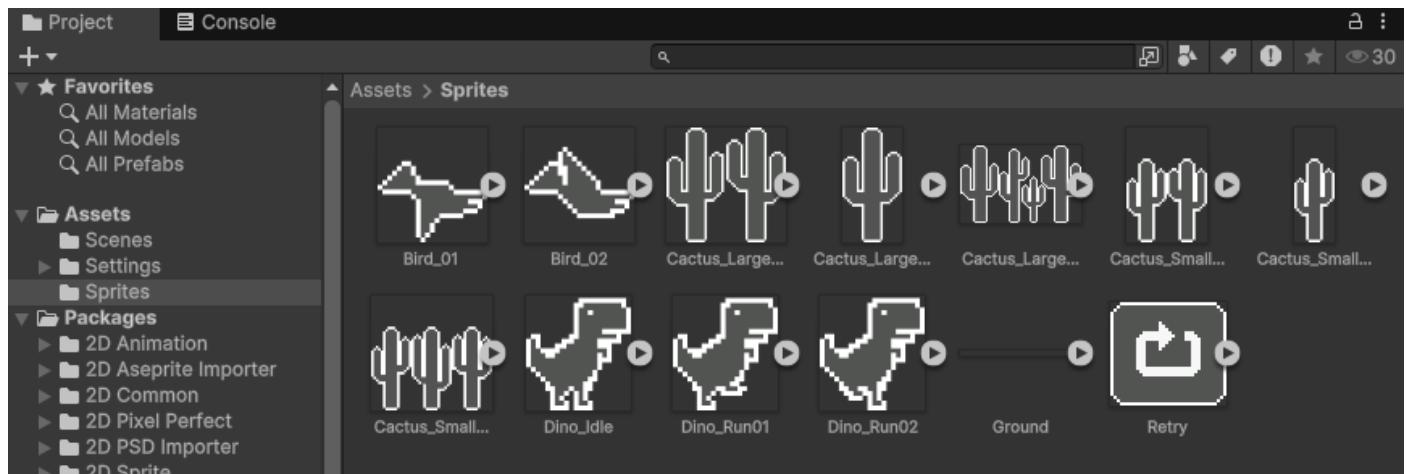
В папке **Scenes** меняем название сцены на **DinoGame**:



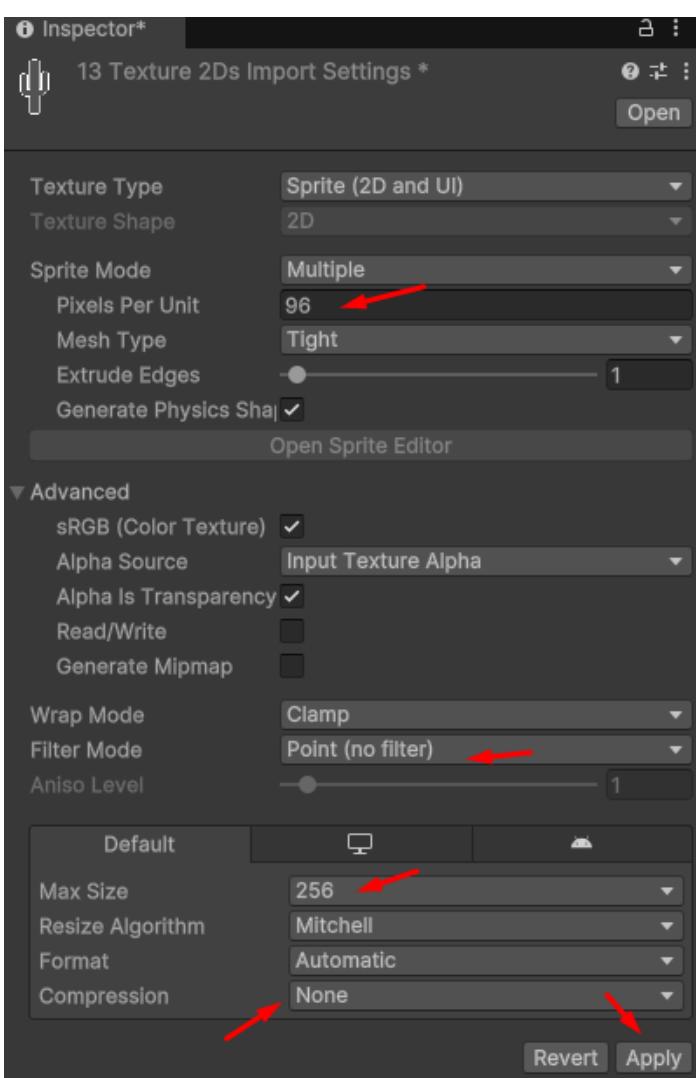
2. В папке **Assets** создаём папку **Sprites**:



Переносим ассеты из архива Assets.DinoGame в папку Sprites:

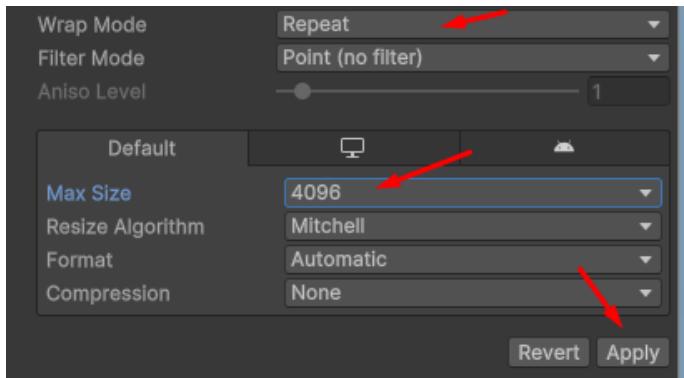


Выберите все спрайты, сделайте следующие изменения и нажмите **Apply**:

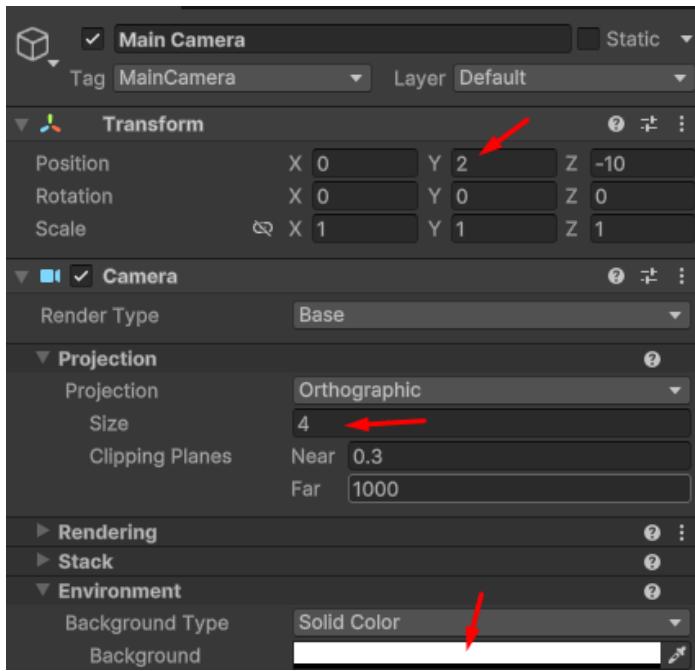


- **Pixel Per Unit** на **96**. Установка PPE на 96 помогает сохранить пиксельный стиль и предотвратить размытие пикселей при масштабировании.
- **Режим фильтрации** с Билинейного меняем на Точечный (**Point**), чтобы избежать размытия спрайтов при их масштабировании.
- **Компрессию** отключаем. Наши изображения занимают мало места, поэтому нет необходимости их сжимать.
- **Максимальный размер** выбираем **256**. Установка максимального размера текстуры на 256 позволяет оптимизировать производительность, так как текстуры меньшего размера быстрее рендерятся и занимают меньше памяти.

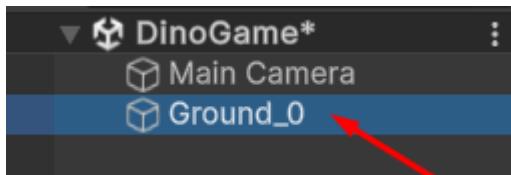
У спрайта **Ground** нужно выбрать режим **Repeat** (чтобы он повторялся), максимальный размер **4096** (т.к. его размеры больше 2000), и нажимаем **Apply**:



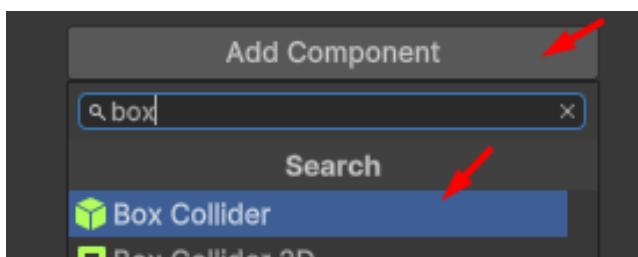
3. Меняем цвет фона у камеры на **белый** и поднимаем позицию по **Y на 2** (чтобы в дальнейшем наши объекты оставались на оси **Y** по **0**, что упростит нашу работу), размер устанавливаем на **4** (чтобы сделать объекты более приближенными к нам):



4. Перетаскиваем спрайт **Ground** в иерархию и меняем его название на **Ground**:

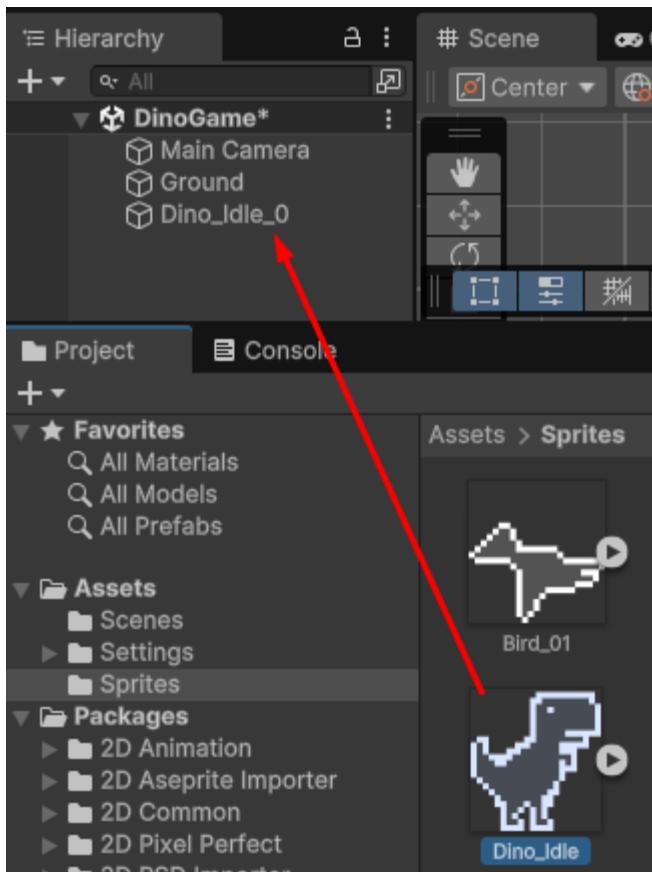


Для **Ground** (земли) добавим **обычный Box Collider**:

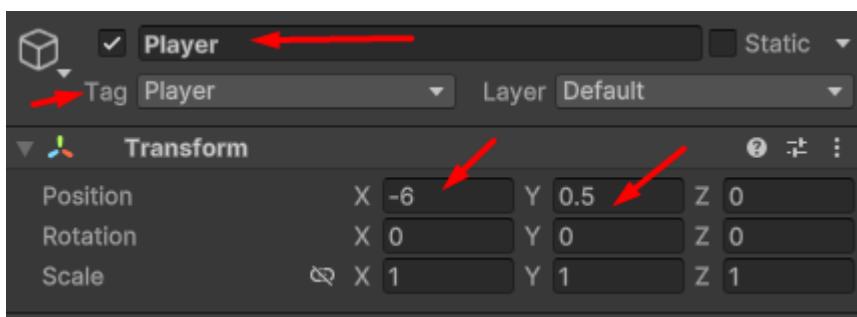


Мы выбираем обычный Box Collider, который работает с 3D объектами в связи с тем, что для нашего персонажа будем использовать другой компонент для управления, который работает в 3D.

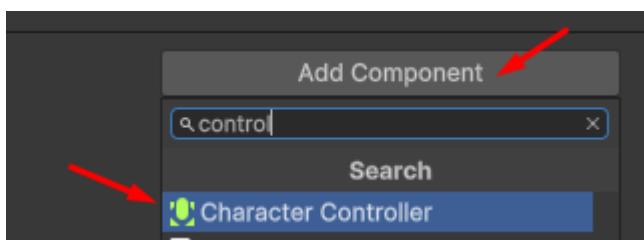
Перетаскиваем спрайт **Dino_Idle** в иерархию:



Меняем его название на **Player**. Добавляем для него тег **Player** и смещаем позицию по **X на -6** (для того, чтобы расположить его в левую часть), по **Y на 0.5** (чтобы немного приподнять):



Добавляем ему компонент **Character Controller**:



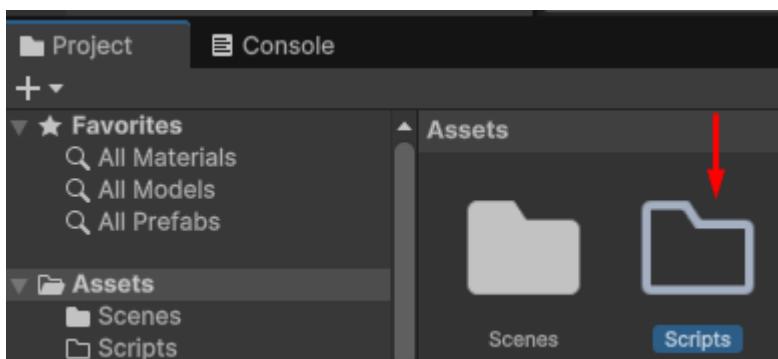
Character Controller предоставляет готовый набор функций для управления перемещением персонажа. Он упрощает создание базовой физики и управления движением, что идеально для платформеров и подобных игр. Он обеспечивает точный контроль над перемещением персонажа. Можно управлять каждым аспектом движения, включая прыжки, падения и столкновения, без необходимости учитывать физические силы, такие как масса или трение.

Поскольку **Character Controller** не использует физический движок Unity, он обеспечивает мгновенные реакции на пользовательский ввод. Это означает, что персонаж будет двигаться точно так, как мы этого захотим, без задержек или инерции.

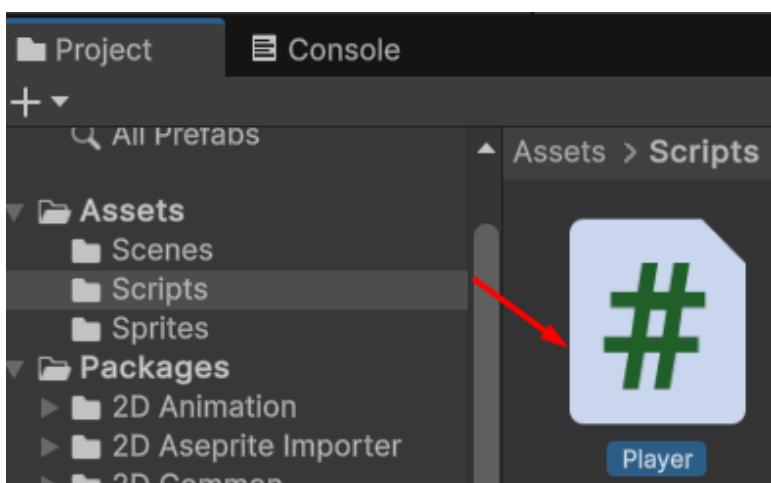
Для земли (**Ground**) нужно будет наш **Box Collider** по оси **Y** опустить его под ноги динозавра (в будущем, возможно, придётся ещё немного подредактировать и опустить его ниже):



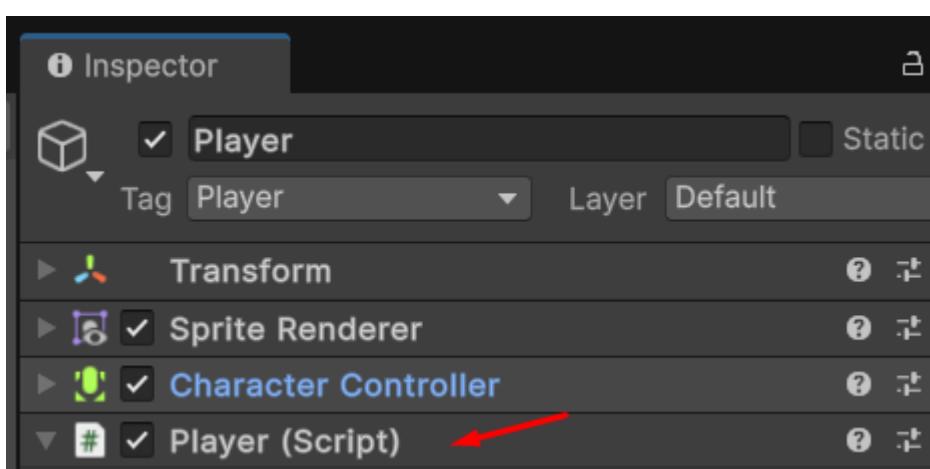
5. Создаём папку Scripts:



В папке **Scripts** создаём скрипт **Player**:



Добавляем его к нашему игроку:



6. Открываем скрипт Player.

➤ Добавляем переменные

```
[Header("Movement Settings")]
//сила гравитации, увеличенная в 2 раза (19.62, вместо 9.81).
[SerializeField] private float gravity = 19.62f;
//сила прыжка
[SerializeField] private float jumpForce = 7f;
//количество возможных прыжков (двойной прыжок)
[SerializeField] private int maxJumps = 2;
//множитель для удержания прыжка
[SerializeField] private float jumpHoldMultiplier = 0.5f;

private CharacterController _characterController; // компонент
для управления передвижением
private Vector3 _direction; // направление движения
private int _jumpCount; // счетчик прыжков
private bool _isJumping; // отслеживание состояния прыжка
private float _jumpTime; // время удержания прыжка
```

➤ Инициализация в Awake(). Получаем компонент CharacterController:

```
private void Awake()
{
    _characterController = GetComponent<CharacterController>();
}
```

➤ Реализация гравитации (ApplyGravity())

```
private void ApplyGravity()
{
    if (_characterController.isGrounded)
    {
        _direction = Vector3.down;
        _jumpCount = 0;
    }
    else
    {
        _direction += Vector3.down * (gravity *
Time.deltaTime);
    }
}
```

Метод следит за тем, чтобы персонаж падал вниз:

- Если динозавр стоит на земле – сбрасываем движение вниз.
- Если в воздухе – добавляем силу гравитации вниз.

➤ Обработка прыжков (HandleJumpInput())

```

private void HandleJumpInput()
{
    if (Input.GetKeyDown(KeyCode.Space) && _jumpCount <
maxJumps)
    {
        _direction = Vector3.up * jumpForce;
        _isJumping = true;
        _jumpTime = 0f;
        _jumpCount++;
    }

    if (Input.GetKey(KeyCode.Space) && _isJumping)
    {
        _direction += Vector3.up * (jumpForce *
jumpHoldMultiplier * Time.deltaTime);
        _jumpTime += Time.deltaTime;
    }

    if (Input.GetKeyUp(KeyCode.Space))
    {
        _isJumping = false;
    }
}

```

Этот метод управляет прыжками по нажатию Space:

1. Первое нажатие Space

- Если прыжков меньше максимума (*maxJumps*), поднимаем персонажа (*jumpForce*).
- Включаем флаг *_isJumping*.
- Обнуляем *_jumpTime*.
- Увеличиваем *_jumpCount*.

2. Удержание Space

- Если *_isJumping* активно, то продолжаем подъем с ослабленной силой (*jumpHoldMultiplier*).

3. Отпускание Space

- Выключаем *_isJumping*.

➤ Основной игровой цикл (*Update()*)

```

private void Update()
{
    ApplyGravity();
    HandleJumpInput();
    _characterController.Move(_direction * Time.deltaTime);
}

```

Каждый кадр выполняет:

1. Применение гравитации;
2. Обработка прыжков;
3. Передвижение персонажа;

➤ Сброс состояния (ResetPlayerState())

```
private void ResetPlayerState()
{
    _direction = Vector3.zero;
    _jumpCount = 0;
    _isJumping = false;
    _jumpTime = 0f;
}
```

Этот метод обнуляет переменные, перед новым запуском.

➤ Сброс состояния в OnEnable()

```
private void OnEnable()
{
    ResetPlayerState();
}
```

Метод OnEnable() вызывается при активации объекта. Мы вызываем метод ResetPlayerState(), который сбрасывает переменные движения

Итоговый код:

```
using UnityEngine;
public class Player : MonoBehaviour
{
    [Header("Movement Settings")]
    [SerializeField] private float gravity = 19.62f;
    [SerializeField] private float jumpForce = 7f;
    [SerializeField] private int maxJumps = 2;
    [SerializeField] private float jumpHoldMultiplier = 0.5f;

    private CharacterController _characterController;
    private Vector3 _direction;
    private int _jumpCount;
    private bool _isJumping;
    private float _jumpTime;

    private void Awake()
    {
        _characterController =
GetComponent<CharacterController>();
    }
}
```

```
private void OnEnable()
{
    ResetPlayerState();
}

private void Update()
{
    ApplyGravity();
    HandleJumpInput();
    _characterController.Move(_direction * Time.deltaTime);
}

private void ApplyGravity()
{
    if (_characterController.isGrounded)
    {
        _direction = Vector3.down;
        _jumpCount = 0;
    }
    else
    {
        _direction += Vector3.down * (gravity *
Time.deltaTime);
    }
}

private void HandleJumpInput()
{
    if (Input.GetKeyDown(KeyCode.Space) && _jumpCount <
maxJumps)
    {
        _direction = Vector3.up * jumpForce;
        _isJumping = true;
        _jumpTime = 0f;
        _jumpCount++;
    }

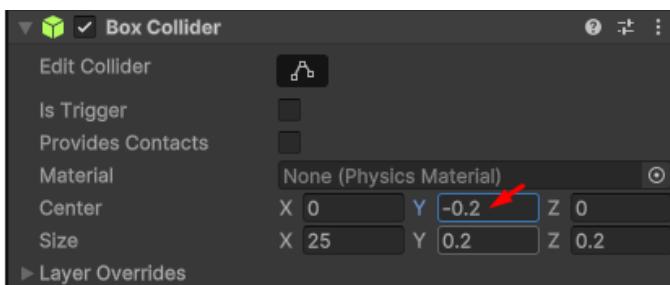
    if (Input.GetKey(KeyCode.Space) && _isJumping)
    {
        _direction += Vector3.up * (jumpForce *
jumpHoldMultiplier * Time.deltaTime);
        _jumpTime += Time.deltaTime;
    }
}
```

```

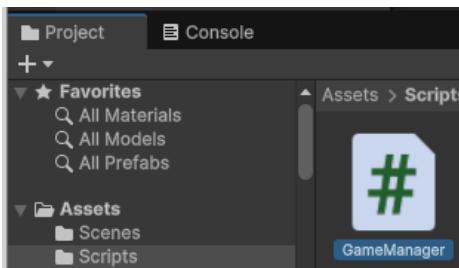
        if (Input.GetKeyUp(KeyCode.Space))
    {
        _isJumping = false;
    }
}
private void ResetPlayerState()
{
    _direction = Vector3.zero;
    _jumpCount = 0;
    _isJumping = false;
    _jumpTime = 0f;
}
}

```

Немного уменьшим у объекта **Ground** координату **Y** на **-0.2**, чтобы происходило лучшее соприкосновение динозавра с землёй в момент окончания прыжка:



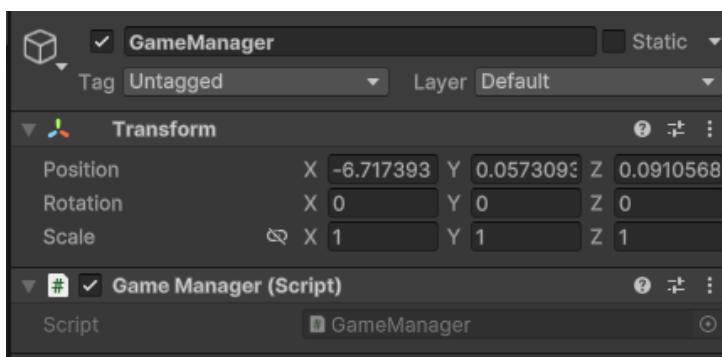
8. Теперь создадим скрипт **GameManager**, в нём мы реализуем прокрутку наших игровых объектов, которые со временем будут ускоряться.



Также создадим пустой объект и назовём его **GameManager**:



Перенесите на него наш скрипт:



Открываем наш скрипт и пишем код:

➤ Объявляем Singleton

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

- Создает **единственный** экземпляр **GameManager**, доступный из любого места в коде.
- **static** позволяет обращаться к **GameManager.Instance**.
- **private set;** запрещает изменять **Instance** извне.

➤ Объявляем переменные для настройки игры

```
[Header("Game Settings")]
[SerializeField] private float initialGameSpeed = 5f;
[SerializeField] private float gameSpeedIncrease = 0.1f;
```

- **initialGameSpeed** — начальная скорость игры (5 единиц).
- **gameSpeedIncrease** — насколько скорость будет увеличиваться каждую секунду (0.1).

➤ Объявляем переменную текущей скорости игры

```
public float GameSpeed { get; private set; }
```

- Хранит **текущую** скорость игры.
- **private set;** запрещает изменять ее вне **GameManager**.

➤ Настраиваем Singleton в Awake()

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

- Если **Instance еще нет**, назначаем текущий **GameManager**.
- Если **Instance уже существует**, уничтожаем этот объект (**Destroy(gameObject);**).
- Гарантирует, что всегда будет **только один** **GameManager**.

➤ Запускаем новую игры в Start()

```
private void Start()
{
    NewGame();
}
```

- Когда игра начинается, вызывает **NewGame()**, устанавливая **стартовые параметры**.

➤ Увеличиваем скорость игры в Update()

```
private void Update()
{
    GameSpeed += gameSpeedIncrease * Time.deltaTime;
}
```

- Каждую **секунду** увеличивает **GameSpeed**.

- **Time.deltaTime** делает изменение плавным и независимым от **FPS**.
 - Чем дольше идет игра, тем быстрее двигаются объекты.
- **Очищаем Instance при уничтожении объекта**

```
private void OnDestroy()
{
    if (Instance == this)
        Instance = null;
}
```

- При удалении **GameManager** сбрасывает **Instance** в **null**.
- Полезно, если **GameManager** **создается заново** при рестарте сцены.

➤ **Запуск новой игры (NewGame)**

```
public void NewGame()
{
    GameSpeed = initialGameSpeed;
}
```

- Сбрасывает скорость игры к начальному значению (**initialGameSpeed**).
- Вызывается в **Start()**, а также при рестарте игры.

➤ **Остановка игры (GameOver)**

```
public void GameOver()
{
    GameSpeed = 0f;
}
```

- Останавливает движение в игре, **обнуляя скорость**.
- Вызывается, когда игрок проиграл.

Итоговый код:

```
using UnityEngine;
public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }
    [Header("Game Settings")]
    [SerializeField] private float initialGameSpeed = 5f;
    [SerializeField] private float gameSpeedIncrease = 0.1f;
    public float GameSpeed { get; private set; }
    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(gameObject);
    }
}
```

```

private void Start()
{
    NewGame();
}

private void Update()
{
    GameSpeed += gameSpeedIncrease * Time.deltaTime;
}

private void OnDestroy()
{
    if (Instance == this)
        Instance = null;
}

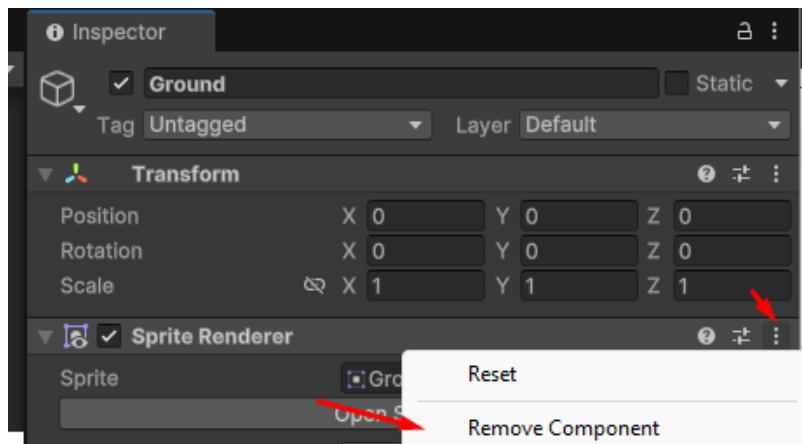
public void NewGame()
{
    GameSpeed = initialGameSpeed;
}

public void GameOver()
{
    GameSpeed = 0f;
}
}

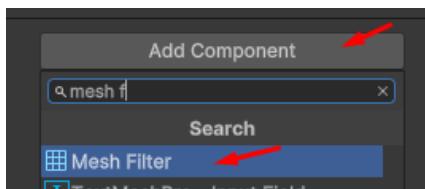
```

9. Теперь мы хотим реализовать передвижения заднего фона, меняя фрагменты нашей текстуры.

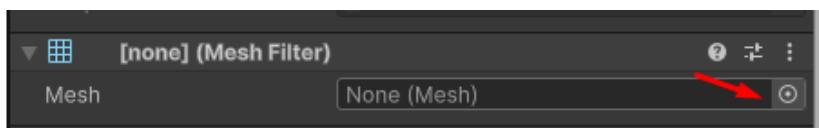
Удаляем для **Ground** компонент **Sprite-Renderer**:



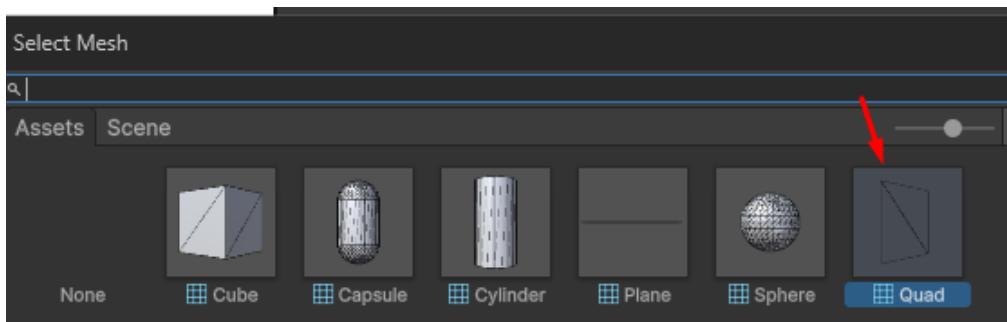
Добавляем новый компонент **Mesh Filter** (будет определять визуализируемую сетку):



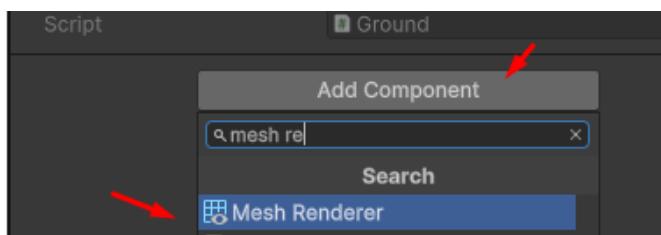
Нажимаем на выбор меша:



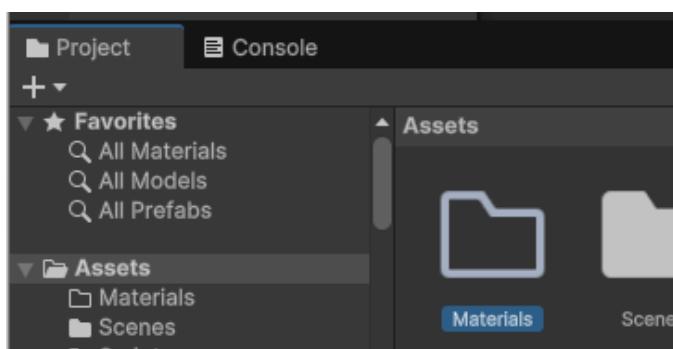
И выберем **Quad** (четырехугольник):



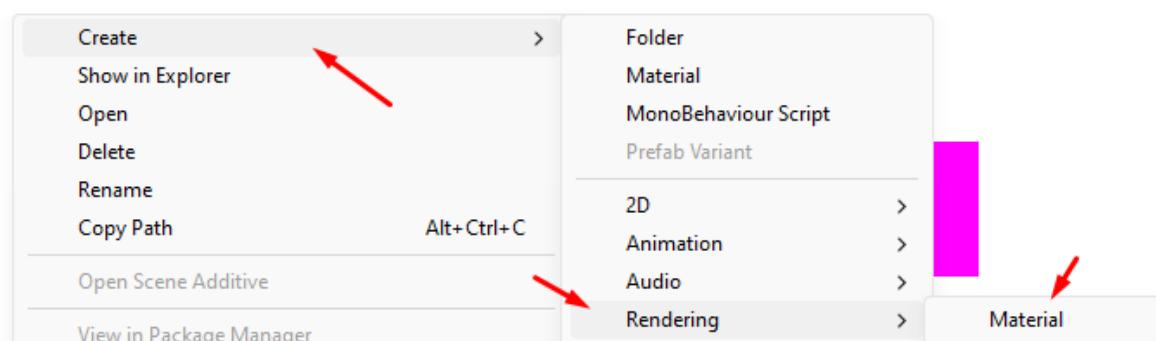
Добавляем новый компонент **Mesh Renderer**:



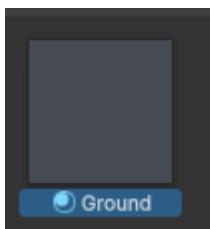
10. Теперь в ассетах создадим папку **Materials**:



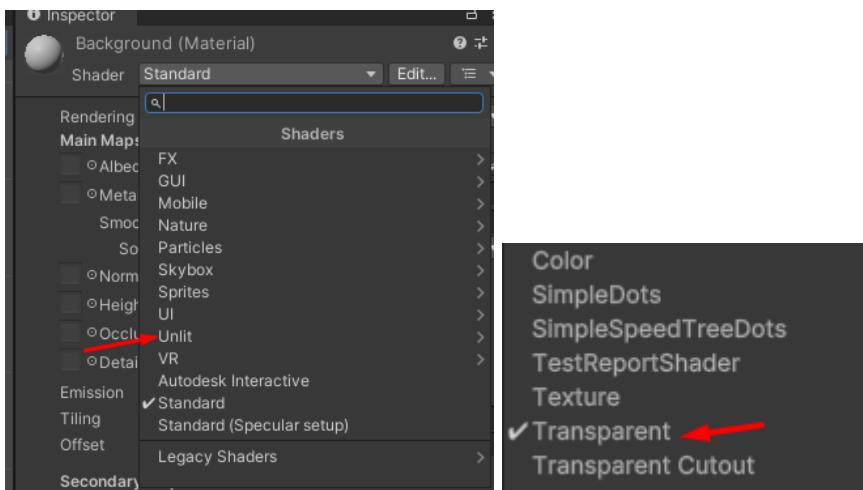
В ней создадим новый материал **Create – Rendering – Material**:



Называем его **Ground**:

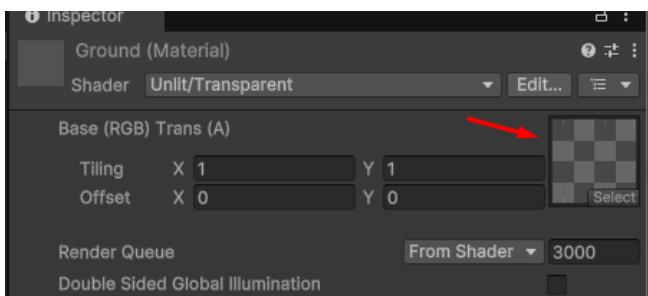


Поменяем у него шейдер на **Unlit – Transparent**:

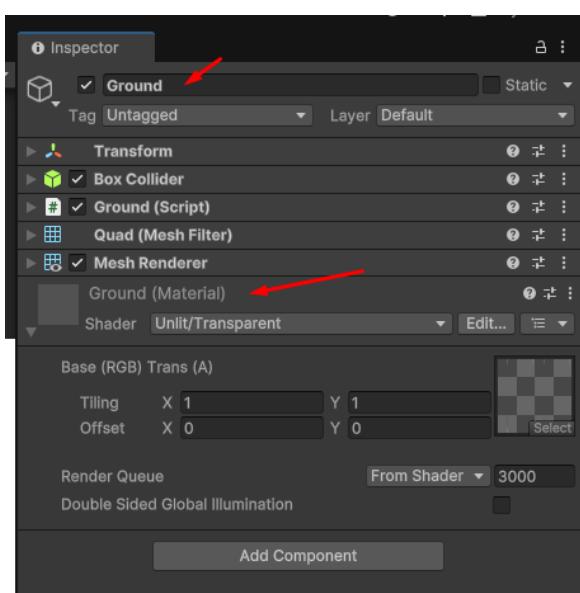


Unit – Texture — это шейдер, который поддерживает прозрачность. Так как, наша текстура прозрачна, то мы выбрали данный шейдер.

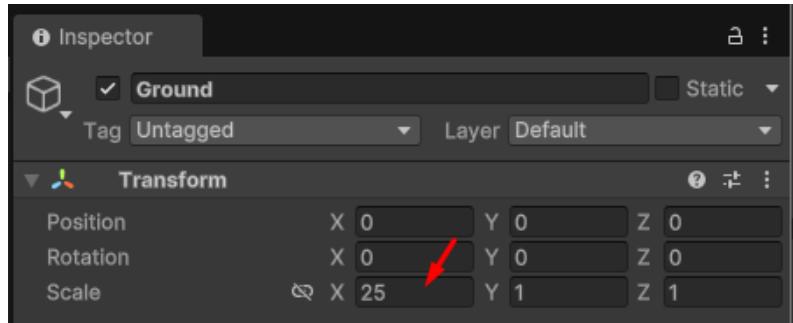
Перенесём на текстуру нашу землю **Ground** из папки **Sprites**:



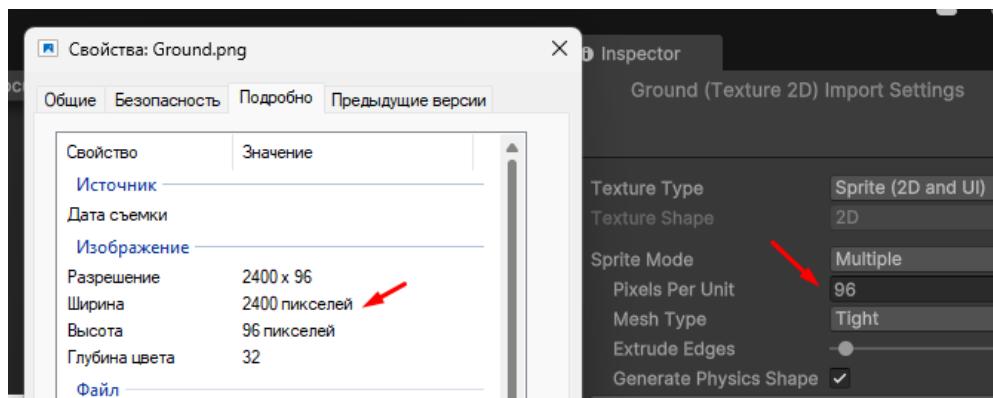
И после перенесём наш материал **Ground** на объект **Ground**:



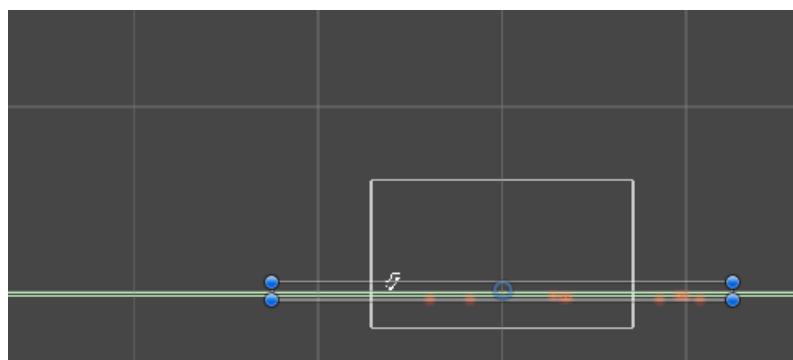
Теперь для нашей земли (объект **Ground**), мы хотим установить размер по **X** на **25:**



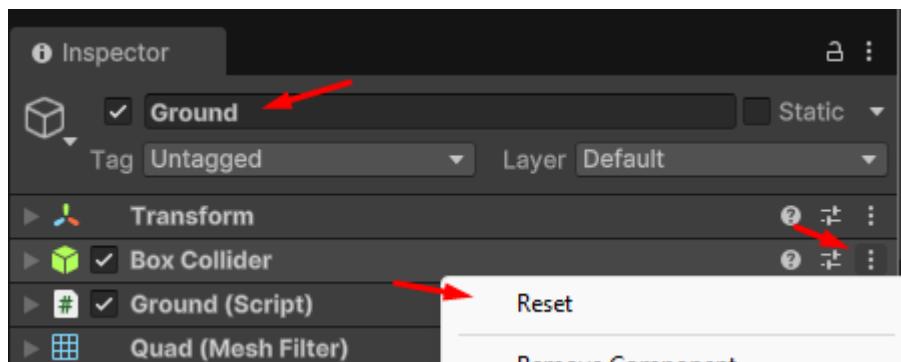
Почему 25? Дело в том что ширина нашей земли, если мы посмотрим свойства объекта = 2400 пикселей. В Unity для объектов мы ставили PPU на 96. $2400 / 96 = 25$. Вот откуда мы получили данное число.



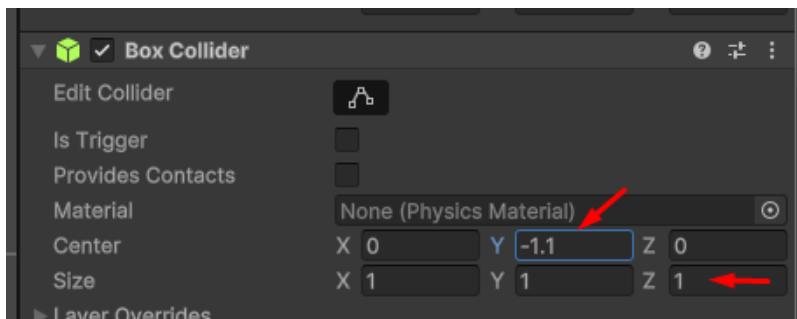
Если мы посмотрим, то увидим, что **Box Collider** у нашего **Ground** стал слишком большим:



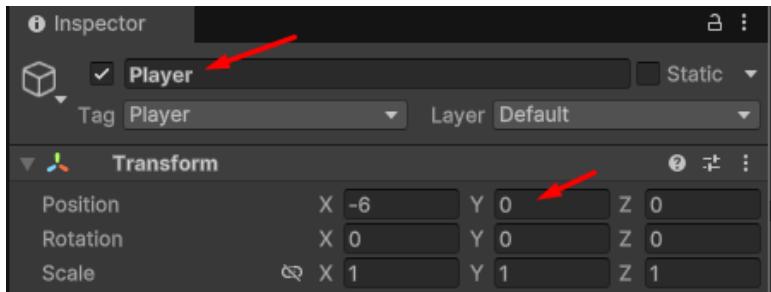
Поэтому давайте обновим его. Нажмём у **Box Collider - Reset:**



Теперь поменяйте значение по **Y** в центре на **-1.1** и **Z** размера на **1** у **Box Collider**:

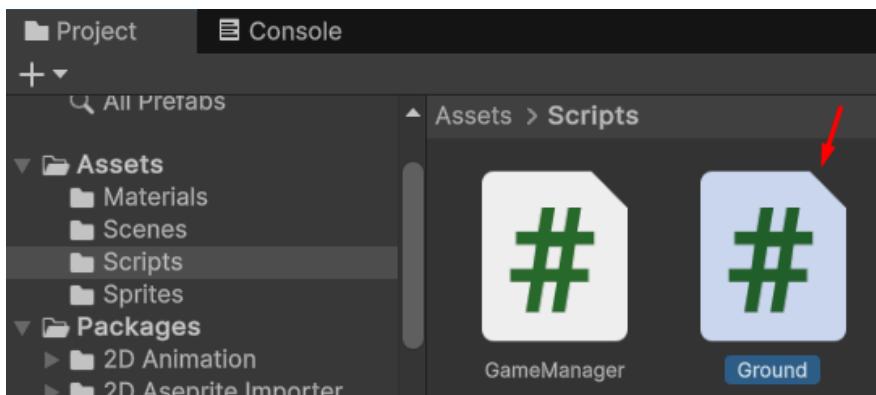


Также можно приподнять игрока **Player** по оси **Y** на **0**:



11. Переходим к написанию скрипта **Ground**. Мы хотим создать эффект бесконечного движения текстуры земли в 2D-игре. Для этого мы будем использовать скорость игры, управляемую **GameManager**, и масштаб объекта для корректировки скорости движения текстуры. В методе **Update** будем обновлять смещение текстуры каждый кадр, создавая иллюзию движения земли.

Создаём скрипт **Ground** в папке с ассетами:



Открываем скрипт и пишем.

➤ Объявляем переменную для работы с материалом

```
private MeshRenderer _meshRenderer;
```

- MeshRenderer отвечает за **отрисовку** объекта и доступ к его материалу.
- Нам нужен материал (material), чтобы **изменять текстуру**.

➤ Инициализируем MeshRenderer в Awake()

```
private void Awake()
{
    _meshRenderer = GetComponent<MeshRenderer>();
}
```

- При старте игры **находим компонент MeshRenderer** у объекта Ground.

- Без этого не сможем менять текстуру земли.

➤ Двигаем текстуру в Update()

```
private void Update()
{
    var speed = GameManager.Instance.GameSpeed /
transform.localScale.x;
    _meshRenderer.material.mainTextureOffset += Vector2.right *
(speed * Time.deltaTime);
}
```

- Получаем **скорость движения земли** var speed:
 - Берем GameSpeed из GameManager.
 - Делим на localScale.x, чтобы учитывать **размер объекта**.
 - Чем шире объект, тем **медленнее** движется текстура.
- **Двигаем текстуру вправо**.
 - mainTextureOffset **смещает** текстуру.
 - Vector2.right ((1,0)) **двигает по оси X**.
 - Time.deltaTime делает движение **плавным и независимым от FPS**.

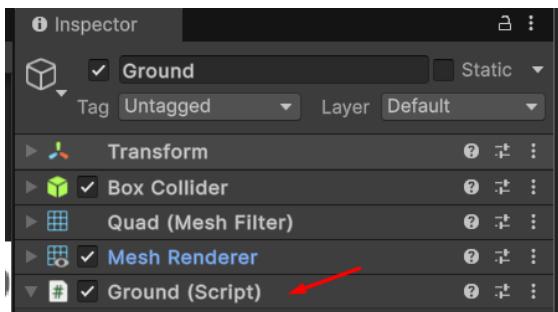
В итоге текстура земли **двигается**, создавая **эффект движения** без перемещения объекта!

Итоговый код:

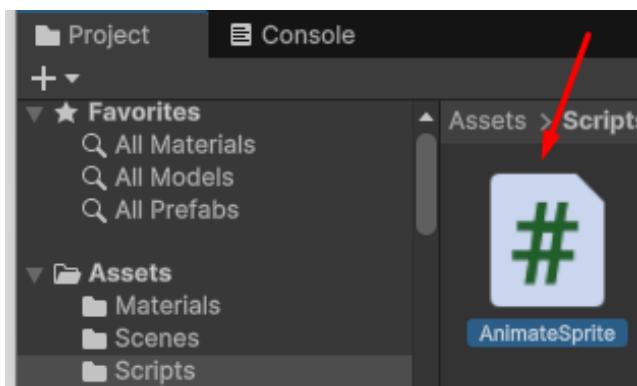
```
using UnityEngine;

public class Ground : MonoBehaviour
{
    private MeshRenderer _meshRenderer;
    private void Awake()
    {
        _meshRenderer = GetComponent<MeshRenderer>();
    }
    private void Update()
    {
        var speed = GameManager.Instance.GameSpeed /
transform.localScale.x;
        _meshRenderer.material.mainTextureOffset +=
Vector2.right * (speed * Time.deltaTime);
    }
}
```

Переносим скрипт **Ground** на наш объект **Ground**:



12. Перейдём к анимации динозавра. Создадим скрипт **AnimateSprite**:



Открываем его и пишем код:

➤ **Создаём переменные**

```
[SerializeField] private Sprite[] sprites;
private SpriteRenderer _spriteRenderer;
private int _frame;
```

- `sprites` — массив спрайтов для анимации.
- `_spriteRenderer` — компонент для отображения текущего кадра.
- `_frame` — номер текущего кадра (индекс в массиве `sprites`).

➤ **Находим SpriteRenderer в Awake()**

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

- При старте находит компонент `SpriteRenderer` на объекте.
- Без этого не сможем менять спрайты.

➤ **Запускаем анимацию при активации объекта**

```
private void OnEnable()
{
    Invoke(nameof(Animate), 0f);
}
```

- `Invoke(nameof(Animate), 0f)` немедленно запускает анимацию.
- Когда объект активируется, анимация сразу стартует.

➤ **Останавливаем анимацию при деактивации объекта**

```
private void OnDisable()
{
    CancelInvoke();
}
```

- `CancelInvoke()` останавливает все запущенные `Invoke`.
- Когда объект отключается, анимация прекращается. Это делается, чтобы анимация не продолжала работать в фоне, когда объект скрыт или уничтожен.

➤ Основной метод `Animate()`

```
private void Animate()
{
    _frame++; // увеличиваем номер текущего кадра
    if (_frame >= sprites.Length) // если кадр вышел за границы
        массива
        _frame = 0; // сбрасываем на 0

    if (_frame >= 0 && _frame < sprites.Length) // запускаем
        анимацию сначала, когда кадры закончились
        _spriteRenderer.sprite = sprites[_frame]; // Меняем
        текущий спрайт в SpriteRenderer. Проверяем, что индекс
        корректный. Назначаем спрайт из массива sprites

    Invoke(nameof(Animate), 1f /
GameManager.Instance.GameSpeed);
    // Запускаем следующий кадр через динамический
    интервал. Скорость анимации зависит от
    GameManager.Instance.GameSpeed. Чем выше GameSpeed, тем быстрее
    меняются кадры
}
```

Итоговый код:

```
using UnityEngine;
public class AnimateSprite : MonoBehaviour
{
    [SerializeField] private Sprite[] sprites;
    private SpriteRenderer _spriteRenderer;
    private int _frame;

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

```

private void OnEnable()
{
    Invoke(nameof(Animate), 0f);
}

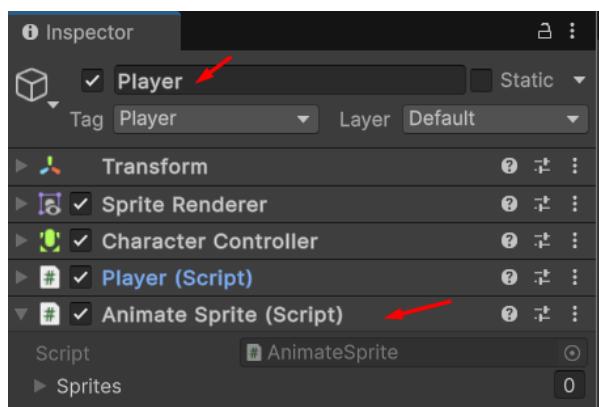
private void OnDisable()
{
    CancelInvoke();
}
private void Animate()
{
    _frame++;
    if (_frame >= sprites.Length)
    {
        _frame = 0;
    }

    if (_frame >= 0 && _frame < sprites.Length)
    {
        _spriteRenderer.sprite = sprites[_frame];
    }

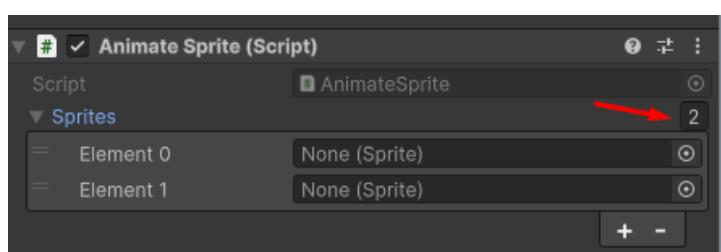
    Invoke(nameof(Animate), 1f /
GameManager.Instance.GameSpeed);
}
}

```

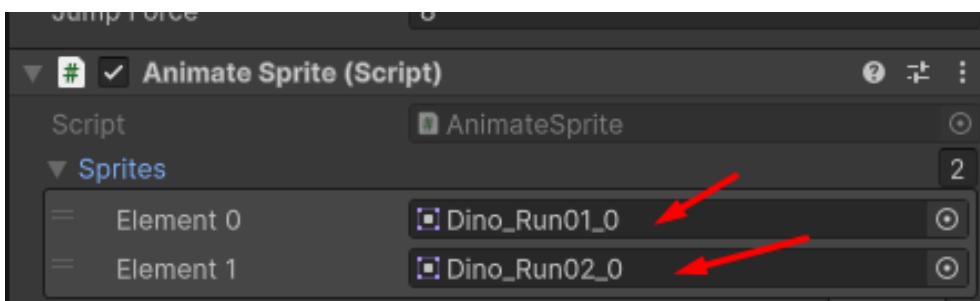
Далее переносим наш скрипт на объект игрока Player:



Указываем количество спрайтов на 2:

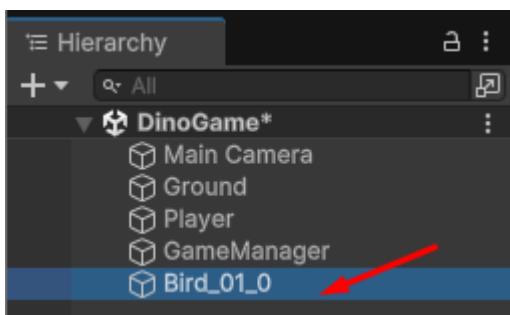


Затем из папки **Sprites** переносим на 1 и 2 элемент анимации для бега:



Теперь при запуске проекта, вы можете убедиться, что динозавр бежит.

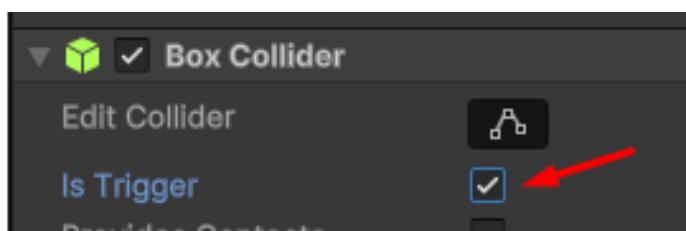
13. Перейдём к созданию других объектов с помощью префабов. Перенесём спрайт с нашей птицей в иерархию:



Поменяем её название на **Bird**

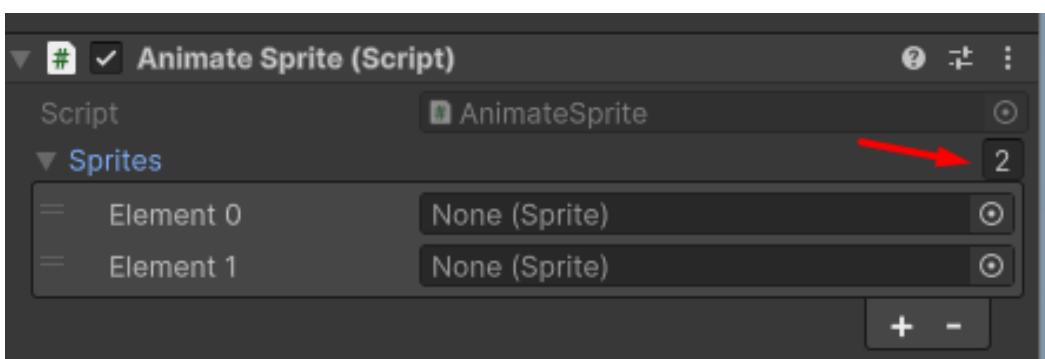


Добавим ей **Box Collider** и для обнаружения столкновений ставим галочку **Is Trigger**:

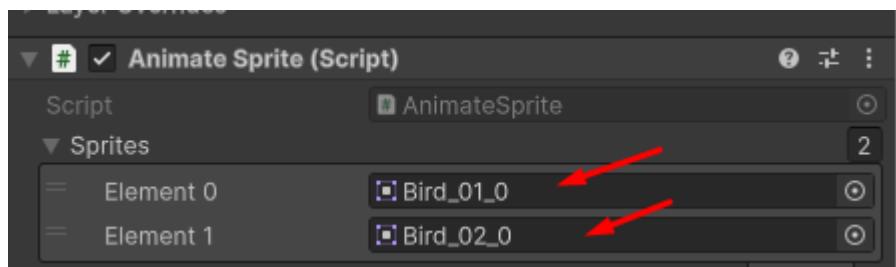


Далее переносим скрипт **AnimateSprite** на объект игрока **Bird**.

Указываем количество спрайтов на 2:

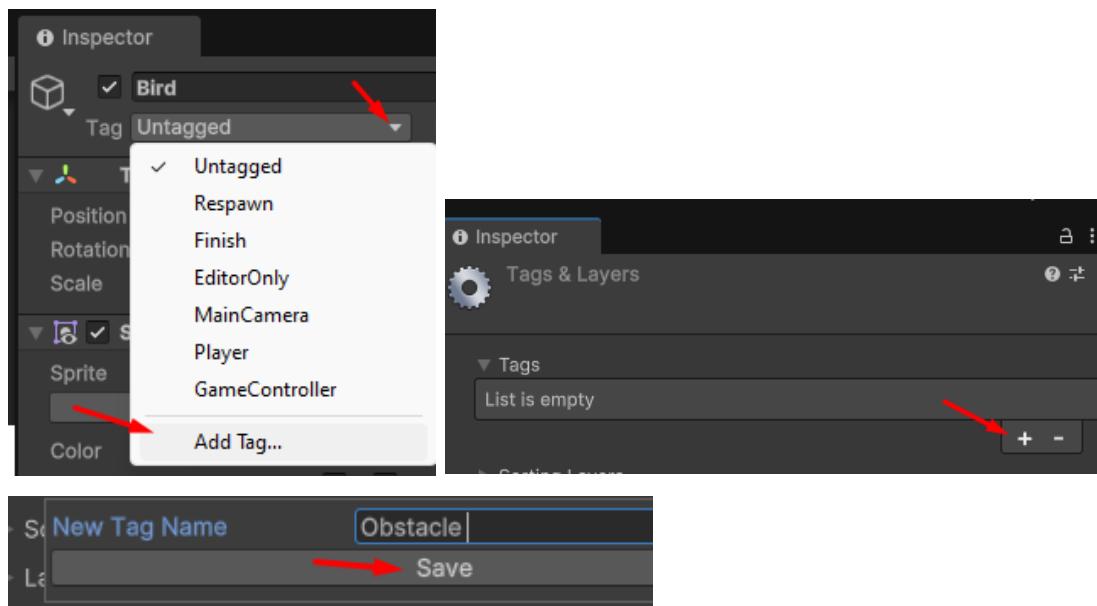


Затем из папки **Sprites** переносим на 1 и 2 элемент анимации для полёта:

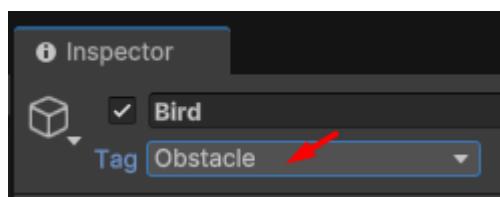


Теперь при запуске проекта, вы можете убедиться, что птица летит.

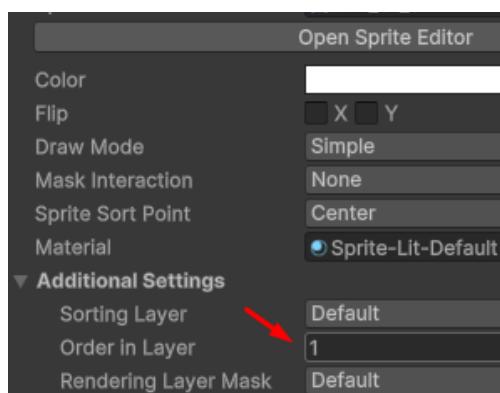
Добавим новый тег **Obstacle**:



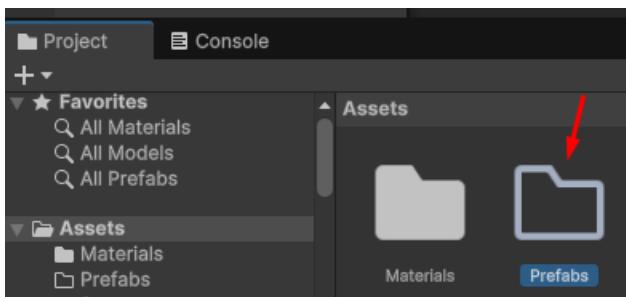
Назначим его птице:



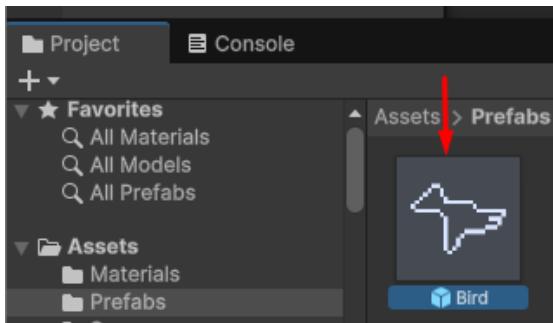
Для **Bird** (птицы) и **Player** (игрока) установите порядок слоя на 1:



Создаём папку **Prefabs**:



Перетаскиваем туда нашу птицу:



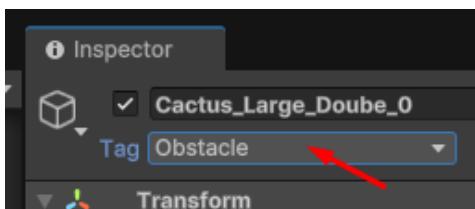
И удаляем птицу из иерархии:



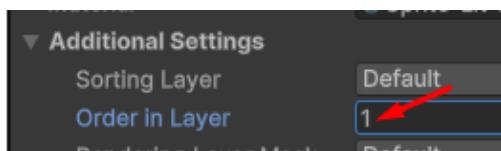
Теперь тоже самое проделываем для кактуса **Cactus_Large_Single**. Переносим его в иерархию:



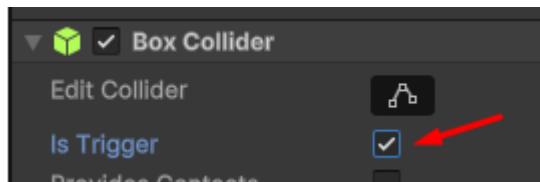
Выбираем тег **Obstacle**:



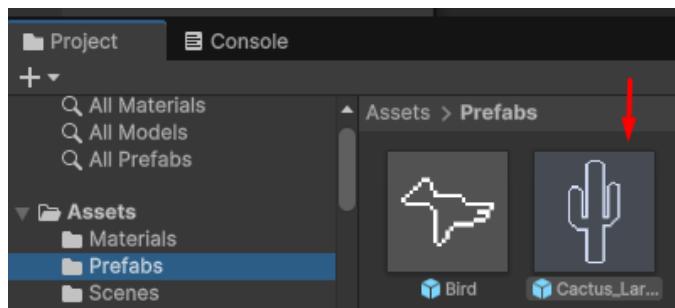
Меняем **Order in Layer** на 1:



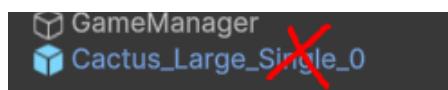
Добавим **Box Collider** и для обнаружения столкновений ставим галочку **Is Trigger**:



Переносим в папку **Prefabs**:



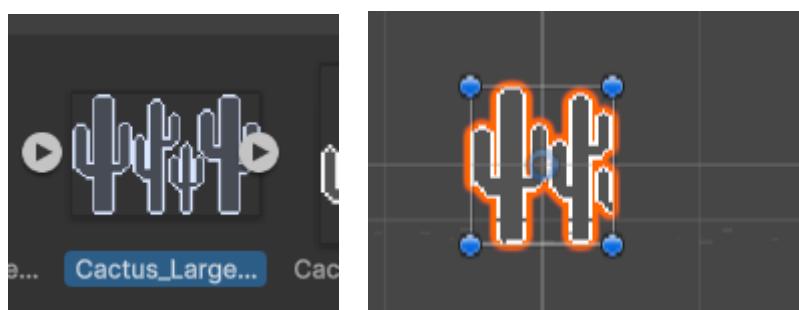
Удаляем из иерархии:



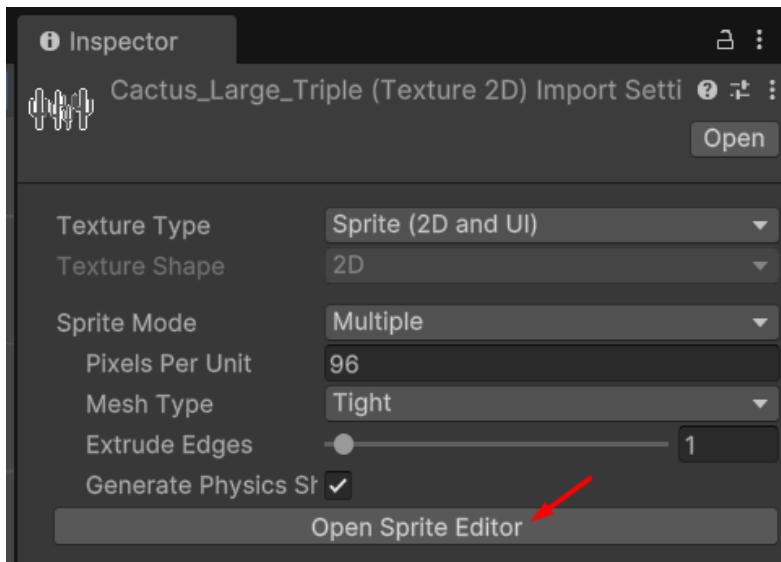
Повторяем для всех остальных кактусов те же самые действия.

- Переносим в иерархию;
- Выбираем тег **Obstacle**;
- Меняем **Order in Layer** на **1**;
- Добавляем **Box Collider** и ставим галочку **Is Trigger**;
- Переносим в папку **Prefabs**;
- Удаляем из иерархии.

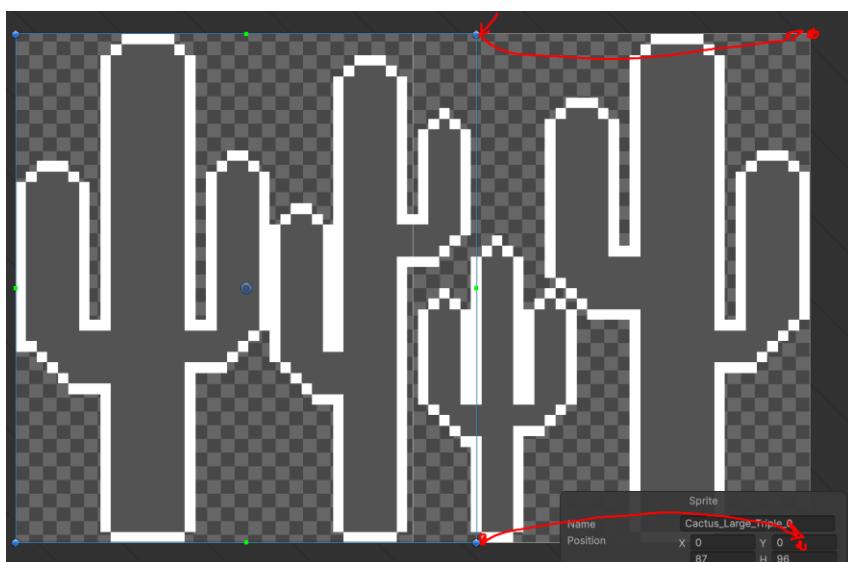
Скорее всего вы можете столкнуться с такой проблемой, что при добавлении некоторых кактусов они выставляются не полностью на сцене, как например на следующих скриншотах у нас отображается часть объекта:



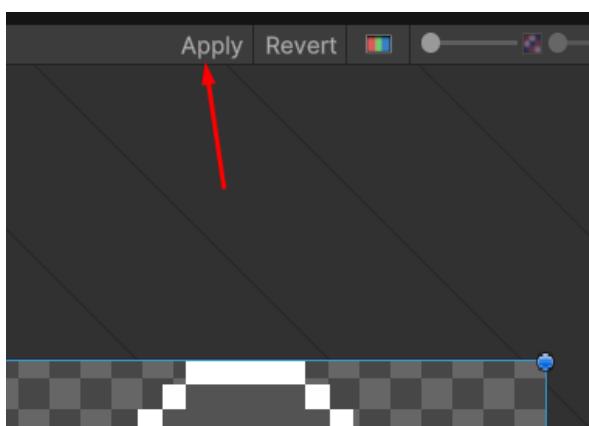
Чтобы решить эту проблему выбираем нужный спрайт, с которым у нас проблема, и нажимаем на **Open Sprite Editor**:



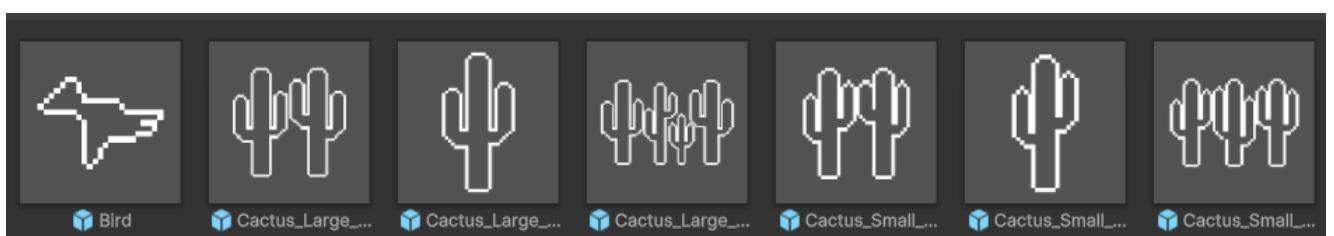
Далее нам нужно перенести границы до конца в угол (передвинуть синие точки):



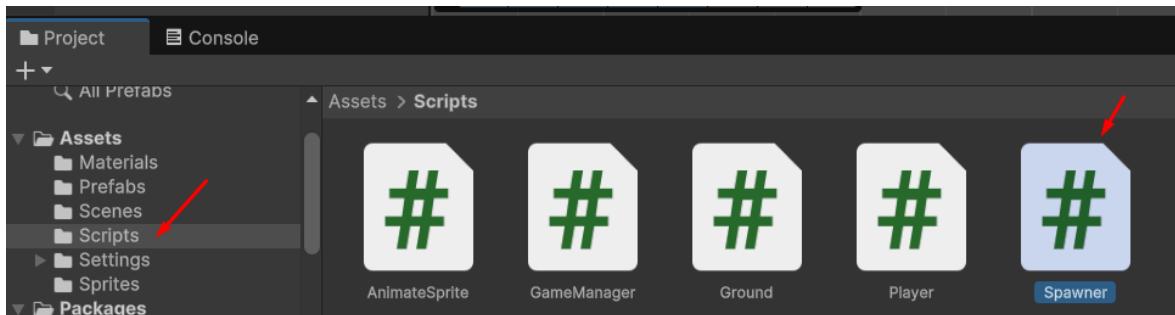
И нажать **Apply**:



Итак, у нас должно получиться **6** префабов кактусов и одна птица:



14. Создадим скрипт для спавна объекта. Назовём его Spawner:



➤ Открываем его и пропишем в начале структуру для объектов и объявим переменные:

```
[System.Serializable]
public struct SpawnObject
{
    public GameObject prefab;
    [Range(0f, 1f)]
    public float spawnChance;
}
[SerializeField] private SpawnObject[] objects;
[SerializeField] private float minSpawnRate = 1f;
[SerializeField] private float maxSpawnRate = 2f;
```

- **SpawnObject** — это **структурка** (тип данных) для хранения **объектов, которые будут появляться**.
- **public GameObject prefab** — ссылка на **префаб** (объект, который создаётся).
- **[Range(0f, 1f)] public float spawnChance** — **шанс появления** (от 0 до 1).
- **objects** — массив объектов, которые могут появляться.
- **minSpawnRate** и **maxSpawnRate** — минимальный и максимальный интервал между спавнами.

Что такое struct?

- ◆ В C# структура (**struct**) — это **облегчённая версия класса**, которая используется для хранения **связанных данных**.

Отличие от class:

- Структуры хранятся в **стеке**, а классы — в **куче (heap)**.
- Структуры **копируются по значению**, а классы — по **ссылке**.
- В Unity **struct** полезны для **небольших наборов данных**, которые не изменяются **после создания**.

Что делает структура SpawnObject?

1. Объединяет два свойства:

- **GameObject prefab** — какой объект спавнить.
- **float spawnChance** — шанс появления объекта (**от 0 до 1**).

2. Используется в массиве objects

```
[SerializeField] private SpawnObject[] objects
```

- Это массив структур, где **каждый элемент** содержит **префаб и шанс спавна**.
- Удобно, потому что мы можем **гибко настраивать** разные объекты в инспекторе.

Зачем нужен [System.Serializable]?

Unity не отображает структуры в Inspector по умолчанию.

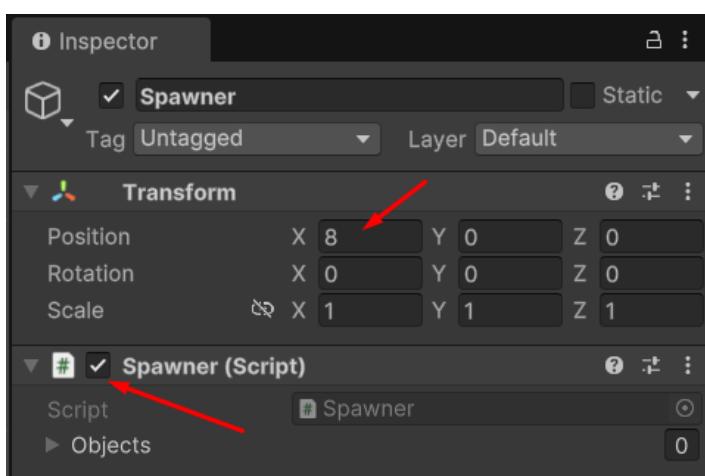
[System.Serializable] разрешает показывать структуру в инспекторе.

Без этого атрибута SpawnObject[] objects не будет видно в инспекторе!

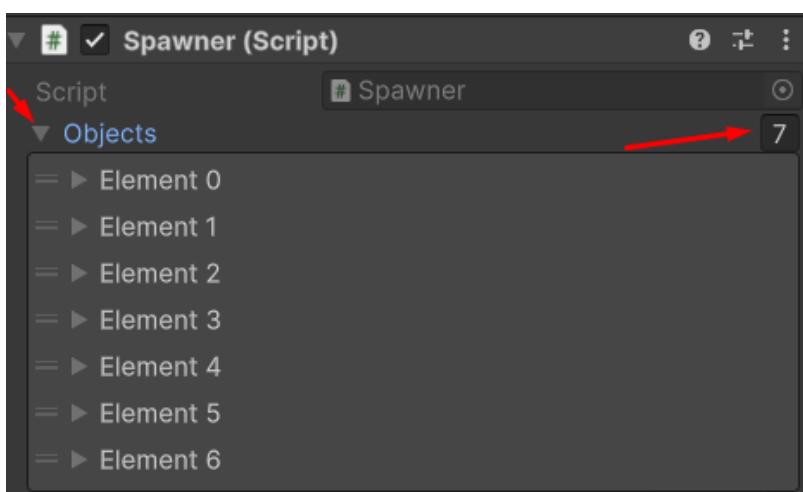
Далее в иерархии создаём пустой объект и называем его **Spawner**:



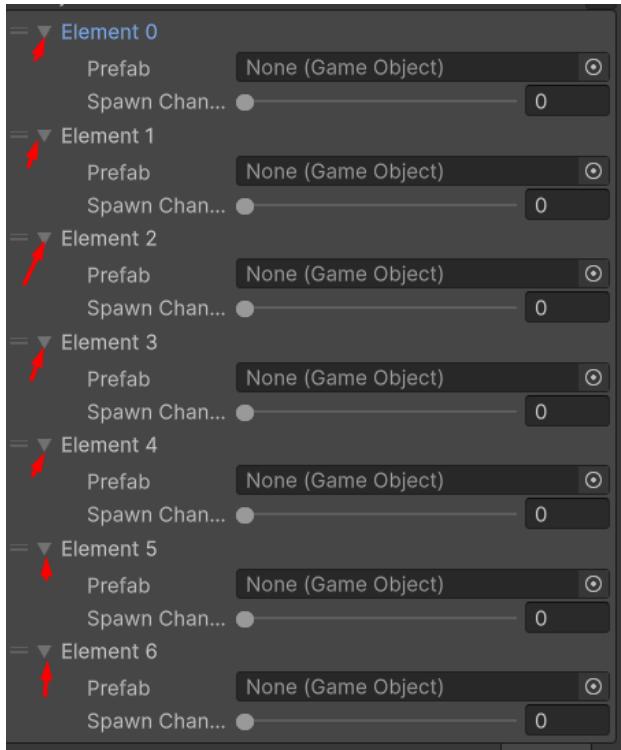
Перенесём на него наш скрипт **Spawner** и вынесем его по позиции **X** за пределы нашей сцены (например, на **8** по **x**):



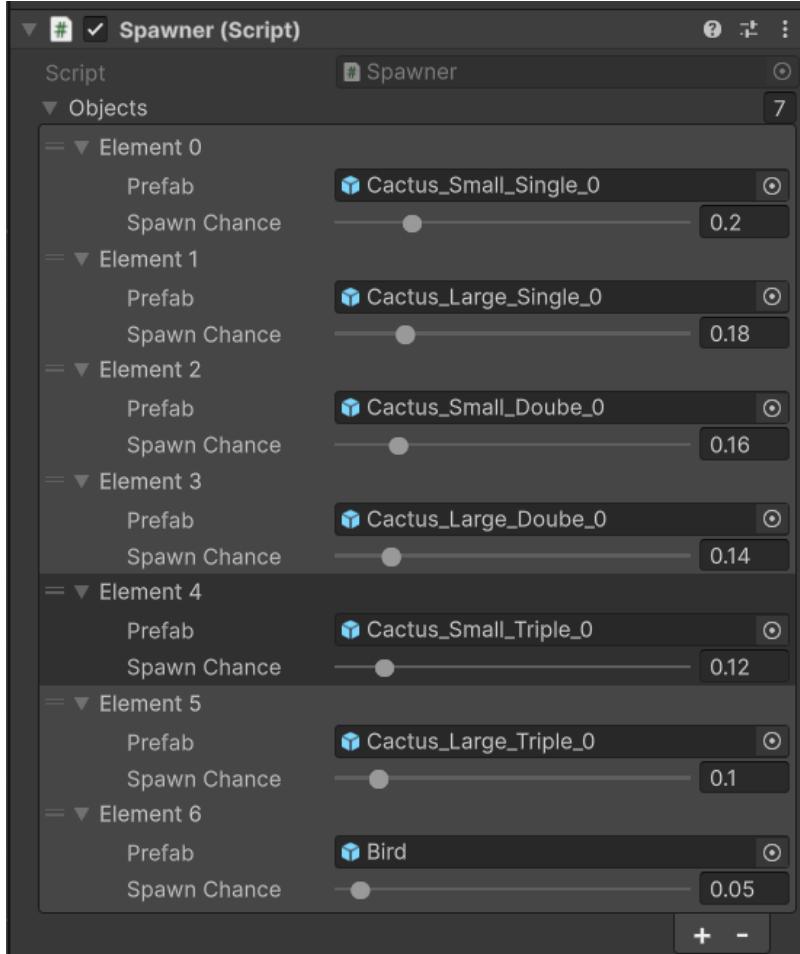
У нас с вами **6 кактусов** и **1 птица**, следовательно у нас с вами **7 объектов**, которые мы хотим добавить. Поэтому пишем **7** в количество объектов и раскрываем их на стрелочку:



На стрелочки раскроем элементы:

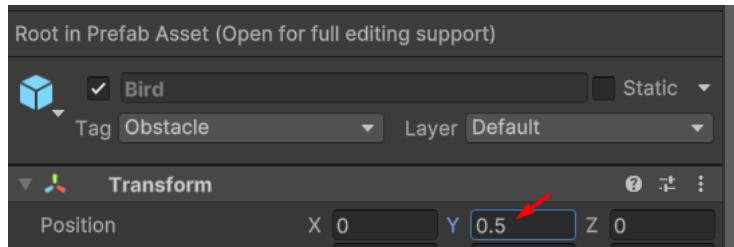


Теперь нам нужно перенести наши префабы и настроить их шанс появления (я выставил шанс появления в следующем порядке: **одиночные кактусы -> двойные кактусы -> тройные кактусы -> 4 кактуса -> птица**. Но вы можете выставить свои значения):



Обратите внимание, если будете выставлять свои значения, то их сумма должна быть близка к **1** (в моём примере **0.95**, что составляет около **95%, 5%** - шанс того, что ничего не появится).

Давайте для префаба нашей птицы немного приподнимем координаты по **Y**:



Продолжим писать скрипт

➤ Создаём метод **Spawn()**:

```
private void Spawn()
{
    var spawnChance = Random.value;

    foreach (var obj in objects)
    {
        if (spawnChance < obj.spawnChance)
        {
            var obstacle = Instantiate(obj.prefab);
            obstacle.transform.position += transform.position;
            break;
        }
        spawnChance -= obj.spawnChance;
    }
    Invoke(nameof(Spawn), Random.Range(minSpawnRate,
maxSpawnRate));
}
```

Теперь подробнее разберём его:

1. Генерируем случайное число от **0** до **1** (определяется какой объект появится):

```
var spawnChance = Random.value;
```

2. Перебираем массив **objects** (проверяем каждый объект в списке):

```
foreach (var obj in objects)
```

3. Проверяем шанс появления (если выпал шанс, создаём объект):

```
if (spawnChance < obj.spawnChance)
```

4. Создаём объект (**Instantiate**):

```
var obstacle = Instantiate(obj.prefab);
obstacle.transform.position += transform.position;
```

- Создаём копию **prefab**.

- Переносим объект в позицию спавнера.

5. Если объект создан — прекращаем цикл, чтобы не создавать несколько объектов сразу:

```
break;
```

6. Если объект не создался, уменьшаем spawnChance (убираем шанс текущего объекта и проверяем следующий объект):

```
spawnChance -= obj.spawnChance;
```

7. Запускаем Spawn() снова (новый объект появится через случайный промежуток времени):

```
Invoke(nameof(Spawn), Random.Range(minSpawnRate,  
maxSpawnRate));
```

➤ Останавливаем спавн при отключении объекта:

```
private void OnDisable()  
{  
    CancelInvoke();  
}
```

- CancelInvoke(); **останавливает все вызовы Invoke**.
- Если объект **отключён**, новые объекты **не создаются**.

🛠 **Зачем?**

- **Оптимизация** — не тратим ресурсы, когда объект неактивен.

➤ Запускаем спавн при активации объекта:

```
private void OnEnable()  
{  
    Invoke(nameof(Spawn), Random.Range(minSpawnRate,  
maxSpawnRate));  
}
```

- Когда объект **включается**, вызываем Spawn() через случайное время.
- Random.Range(minSpawnRate, maxSpawnRate) — случайное время **до первого появления объекта**.

Итоговый код:

```
using UnityEngine;  
  
public class Spawner : MonoBehaviour  
{  
    [System.Serializable]  
    public struct SpawnObject  
    {  
        public GameObject prefab;  
        [Range(0f, 1f)]  
        public float spawnChance;  
    }  
  
    [SerializeField] private SpawnObject[] objects;  
    [SerializeField] private float minSpawnRate = 1f;  
    [SerializeField] private float maxSpawnRate = 2f;
```

```

private void OnEnable()
{
    Invoke(nameof(Spawn), Random.Range(minSpawnRate,
maxSpawnRate));
}

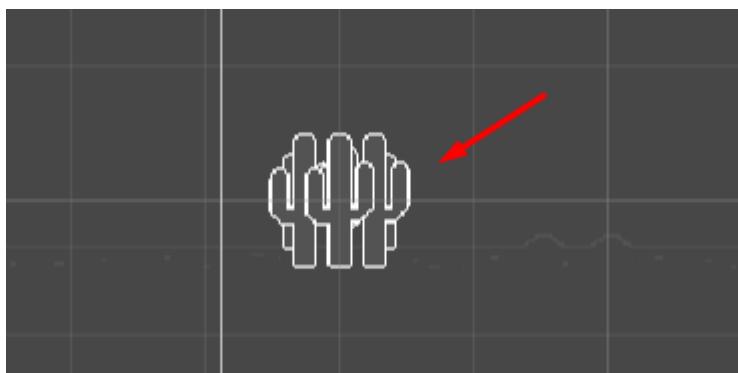
private void OnDisable()
{
    CancelInvoke();
}

private void Spawn()
{
    var spawnChance = Random.value;

    foreach (var obj in objects)
    {
        if (spawnChance < obj.spawnChance)
        {
            var obstacle = Instantiate(obj.prefab);
            obstacle.transform.position +=
transform.position;
            break;
        }
        spawnChance -= obj.spawnChance;
    }
    Invoke(nameof(Spawn), Random.Range(minSpawnRate,
maxSpawnRate));
}

```

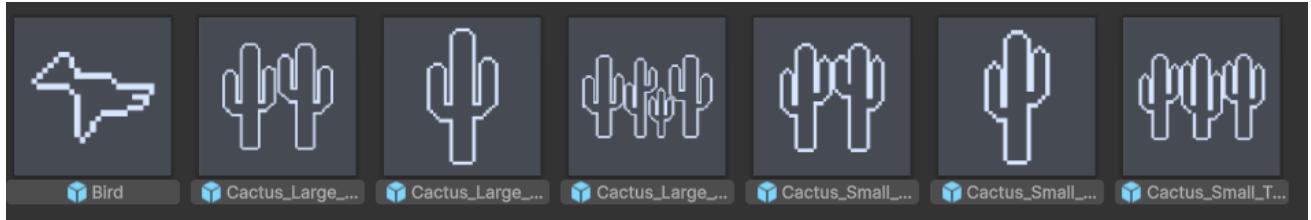
Теперь если мы запустим сцену, то увидим, что наши объекты появляются за пределами сцены:



15. Создадим скрипт в папке **Scripts** для перемещения объектов, назовём его **MoveEnemy**:



Перетащите скрипт на все наши префабы:



Откроем и напишем скрипт:

➤ **Определяем переменную _leftEdge**

```
private float _leftEdge;
```

- `_leftEdge` — левая граница экрана, за которую объект не должен выходить.
- Когда объект пересекает границу, он удаляется.

➤ **Вычисляем _leftEdge в Start()**

```
private void Start()
{
    if (Camera.main != null)
    {
        _leftEdge =
Camera.main.ScreenToWorldPoint(Vector3.zero).x - 2f;
    }
}
```

- `Vector3.zero` — точка (0,0) на экране (левый нижний угол).
- `ScreenToWorldPoint()` переводит координаты из экрана в мир.
- `.x` — левая граница камеры в мировых координатах.
- Смещаем границу ещё левее на 2 единицы, чтобы объект удалялся заранее.

🛠 Зачем проверка `if (Camera.main != null)?`

- Избегаем ошибок в случае, если в сцене нет камеры.

➤ **Двигаем объект в Update() и удаляем если он вышел за экран:**

```
private void Update()
{
    transform.position += Vector3.left *
(GameManager.Instance.GameSpeed * Time.deltaTime);
```

```

    if (transform.position.x < _leftEdge)
    {
        Destroy(gameObject);
    }
}

```

*Vector3.left * (GameManager.Instance.GameSpeed * Time.deltaTime)*

- Двигает объект **влево** со скоростью GameSpeed.
- **Time.deltaTime** делает движение **плавным и зависимым от FPS**.
- Если координата x объекта **меньше _leftEdge**, объект **удаляется**.
- **Destroy(gameObject);** освобождает память и оптимизирует игру.

Итоговый код:

```

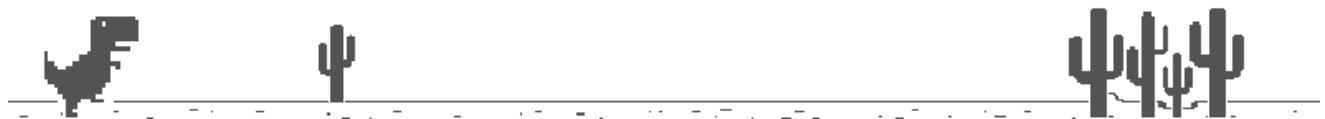
using UnityEngine;

public class MoveEnemy : MonoBehaviour
{
    private float _leftEdge;
    private void Start()
    {
        if (Camera.main != null)
        {
            _leftEdge =
Camera.main.ScreenToWorldPoint(Vector3.zero).x - 2f;
        }
    }
    private void Update()
    {
        transform.position += Vector3.left *
(GameManager.Instance.GameSpeed * Time.deltaTime);

        if (transform.position.x < _leftEdge)
        {
            Destroy(gameObject);
        }
    }
}

```

Запускаем и проверяем что объекты идут в нашей игре:



16. Добавим на сцену UI.

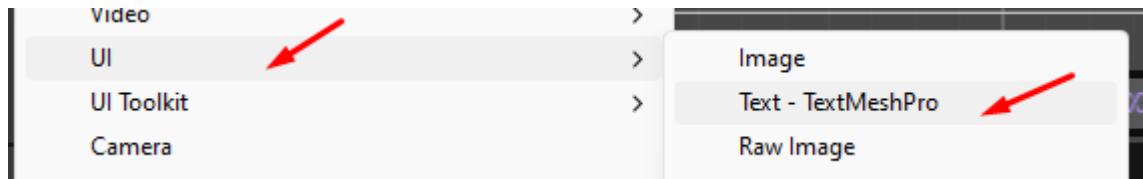
Мы хотим добавить на сцену – **надпись окончания игры**, **кнопку для перезапуска**, **два типа очков** (первые отображают текущий счёт, вторые максимальный набранный в прошлый раз):

00000 00000

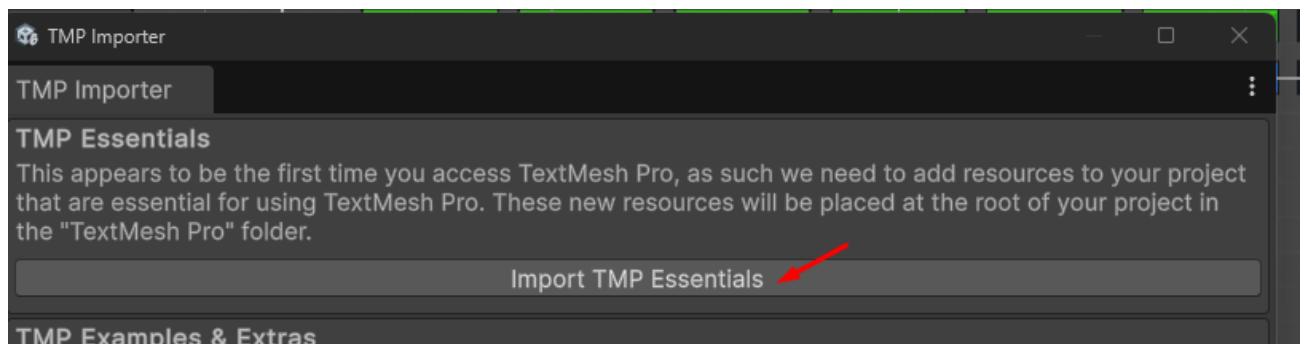
Game Over



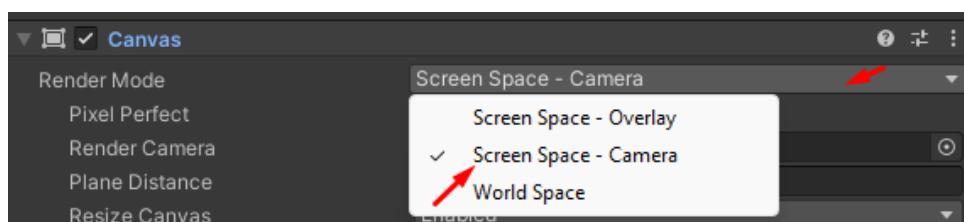
Щёлкаем правой кнопкой мыши в иерархии и добавляем **UI** – **Text** – **TextMeshPro**:



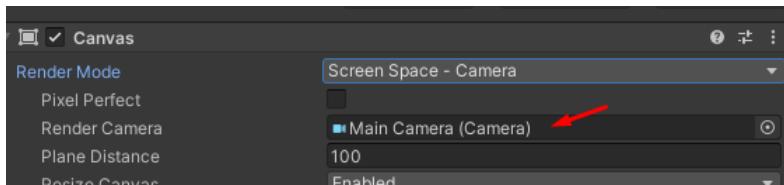
Соглашаемся на импорт шрифтов и после закрываем окно:



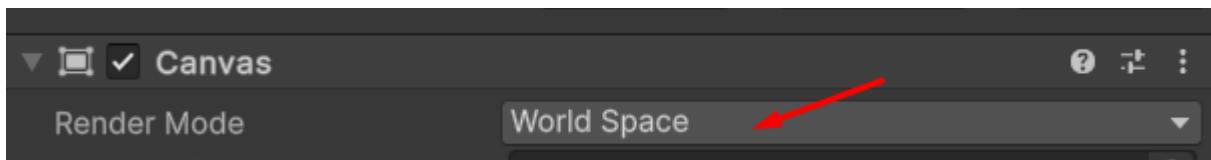
В **Inspector** у **Canvas** выбираем в модели рендера **Screen Space - Camera**:



Переносим **Main Camera** в рендер:



Затем меняем **Canvas** на **World Space**:



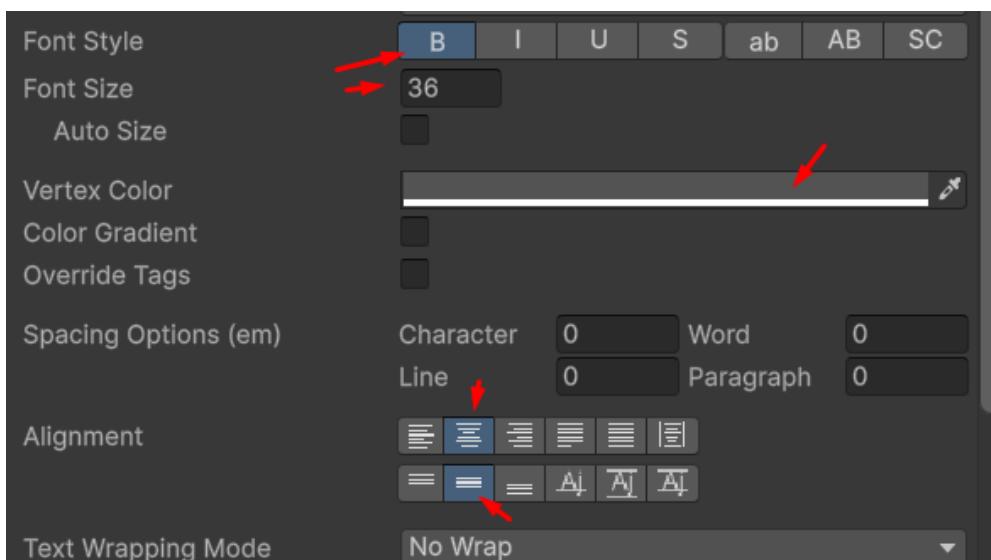
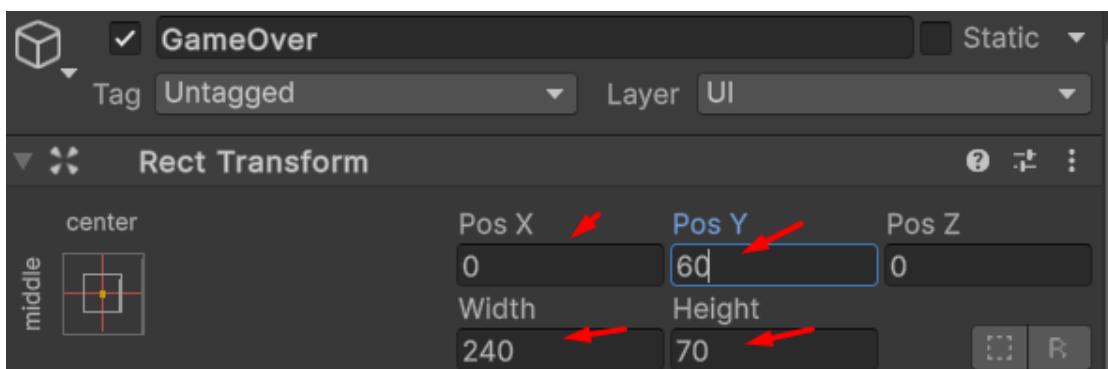
Подгоняем размер текста, под нашу игровую область. Меняем размер шрифта, цвет и текст на **Game Over**:

Game Over

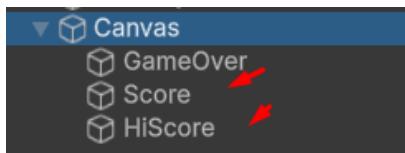
Меняем название объекта на **GameOver**:



Можете воспользоваться моими настройками или сделать на своё усмотрение:



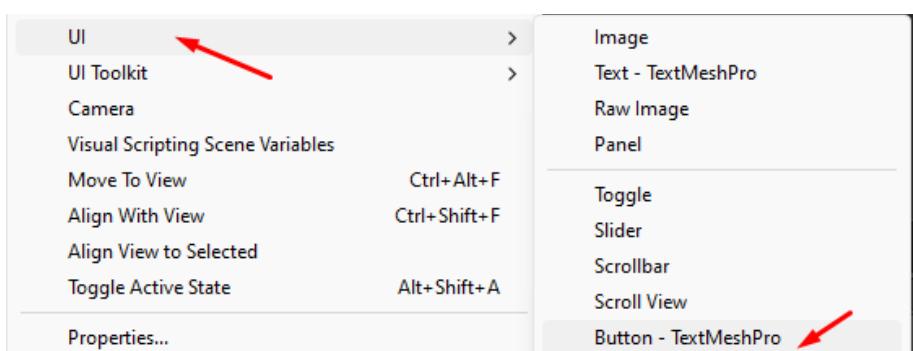
Сразу же создадим ещё два текста **Text – TextMeshPro** и назовём их **Score** и **HiScore**:



Подгоняем размер текста, под нашу игровую область. Меняем размер шрифта, цвет и текст на **00000**:



Создаём кнопку, щёлкаем по **Canvas** и выбираем **UI – Button TextMeshPro**:



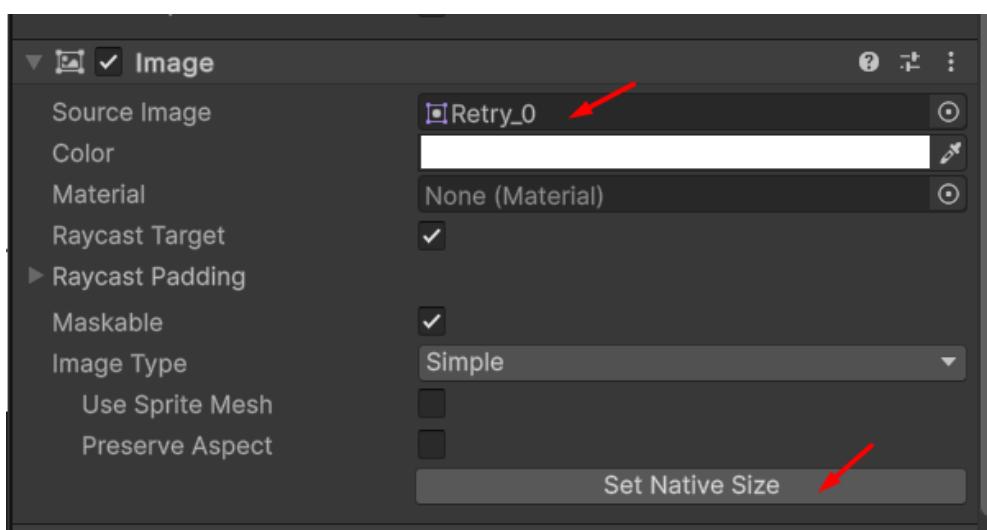
Меняем название на **ButtonRetry**:



Удаляем внутри кнопки текст:



Переносим в нашей кнопке в **Source Image** из папки **Sprites** наш спрайт кнопки **Retry** и нажмём **Set Native Size**:



17. Теперь нам нужно внести правки в скрипт **GameManager** - обработку столкновений, отображение счёта, сохранение рекорда и управление UI.

Открываем его и пишем.

➤ Добавляем новые пространства имён

```
using TMPro; // Добавляем TextMeshPro для работы с текстом
using UnityEngine.UI; // Добавляем поддержку UI-кнопок (Button)
```

➤ Добавляем переменные для UI (позволят отображать игровой счёт, рекорд и кнопку рестарта):

```
[Header("UI Elements")]
[SerializeField] private TextMeshProUGUI gameOverText; // Текст "Game Over"
[SerializeField] private TextMeshProUGUI scoreText; // Отображение текущего счёта
[SerializeField] private TextMeshProUGUI hiScoreText; // Отображение рекорда
[SerializeField] private Button retryButton; // Кнопка "Заново"
```

➤ Добавляем ссылки на Player и Spawner:

```
private Player _player; // Ссылка на игрока
private Spawner _spawner; // Ссылка на спавнер объектов
```

➤ Находим игрока и спавнер при старте

```
private void Start()
{
    _player = FindAnyObjectByType<Player>(); // Находим объект с компонентом Player
    _spawner = FindAnyObjectByType<Spawner>(); // Находим объект с компонентом Spawner
    NewGame(); // Запускаем новую игру
}
```

FindAnyObjectByType<Player>() и FindAnyObjectByType<Spawner>() позволяют автоматически находить нужные объекты без привязки в инспекторе.

➤ Добавляем подсчёт очков в методе Update():

```
private void Update()
{
    GameSpeed += gameSpeedIncrease * Time.deltaTime; // Увеличиваем скорость игры со временем
    _score += GameSpeed * Time.deltaTime; // Добавляем очки в зависимости от скорости
    scoreText.text = Mathf.FloorToInt(_score).ToString("D5");
    // Обновляем текст счёта
}
```

- Очки (_score) растут в зависимости от скорости игры (GameSpeed).
 - Счёт обновляется каждую секунду и отображается на экране.
 - Используем "D5", чтобы число отображалось в формате "00001", "00010" и т. д.
- Напишем метод для сохранения и обновления рекорда UpdateHiScore():

```
private void UpdateHiScore()
{
    var hiScore = PlayerPrefs.GetFloat("hiScore", 0); // Получаем сохранённый рекорд
    if (_score > hiScore) // Если текущий счёт больше рекорда
    {
        hiScore = _score; // Обновляем рекорд
        PlayerPrefs.SetFloat("hiScore", hiScore); // Сохраняем рекорд
    }
    hiScoreText.text =
    Mathf.FloorToInt(hiScore).ToString("D5"); // Обновляем отображение рекорда
}
```

- Берём рекорд из PlayerPrefs (локальное хранилище Unity).
- Если игрок побил рекорд, сохраняем его.
- Обновляем hiScoreText на экране.

➤ Создаём метод для логики управления игрой SetGameState():

```
private void SetGameState(bool isActive)
{
    enabled = isActive; // Включаем или выключаем GameManager
    _player.gameObject.SetActive(isActive); // Включаем или выключаем игрока
    _spawner.gameObject.SetActive(isActive); // Включаем или выключаем спавнер
    gameOverText.gameObject.SetActive(!isActive); // Показываем "Game Over" при проигрыше
    retryButton.gameObject.SetActive(!isActive); // Показываем кнопку "Заново" при проигрыше
}
```

- При проигрыше (isActive = false) выключаем управление, спавнер и игрока.
- Включаем текст "Game Over" и кнопку "Заново".
- При старте (isActive = true) всё снова включается.

➤ Вызовем созданные методы в GameOver():

```

public void GameOver()
{
    GameSpeed = 0f;
    SetGameState(false);
    UpdateHiScore();
}

```

➤ Внесём правки в методе NewGame():

```

public void NewGame()
{
    foreach (var obstacle in
FindObjectsOfType<MoveEnemy>(FindObjectsInactive.Exclude,
FindObjectsSortMode.None)) // ищем все объекты на сцене
    {
        Destroy(obstacle.gameObject); // уничтожаем объекты
    }

    _score = 0f; // обнуляем очки
    GameSpeed = initialGameSpeed;
    SetGameState(true); // сохраняем рекорд
    UpdateHiScore(); // запускаем метод обновления очков
}

```

Итоговый код:

```

using TMPro;
using UnityEngine.UI;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }
    [Header("Game Settings")]
    [SerializeField] private float initialGameSpeed = 5f;
    [SerializeField] private float gameSpeedIncrease = 0.1f;

    [Header("UI Elements")]
    [SerializeField] private TextMeshProUGUI gameOverText;
    [SerializeField] private TextMeshProUGUI scoreText;
    [SerializeField] private TextMeshProUGUI hiScoreText;
    [SerializeField] private Button retryButton;

    private Player _player;
    private Spawner _spawner;
    private float _score;
    public float GameSpeed { get; private set; }
}

```

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

private void Start()
{
    _player = FindAnyObjectOfType<Player>();
    _spawner = FindAnyObjectOfType<Spawner>();
    NewGame();
}

private void Update()
{
    GameSpeed += gameSpeedIncrease * Time.deltaTime;
    _score += GameSpeed * Time.deltaTime;
    scoreText.text =
Mathf.FloorToInt(_score).ToString("D5"));
}

private void OnDestroy()
{
    if (Instance == this)
        Instance = null;
}

public void NewGame()
{
    foreach (var obstacle in
FindObjectsOfType<MoveEnemy>(FindObjectsInactive.Exclude,
FindObjectsSortMode.None))
    {
        Destroy(obstacle.gameObject);
    }

    _score = 0f;
    GameSpeed = initialGameSpeed;
    SetGameState(true);
    UpdateHiScore();
}
```

```

public void GameOver()
{
    GameSpeed = 0f;
    SetGameState(false);
    UpdateHiScore();
}

private void SetGameState(bool isActive)
{
    enabled = isActive;
    _player.gameObject.SetActive(isActive);
    _spawner.gameObject.SetActive(isActive);
    gameOverText.gameObject.SetActive(!isActive);
    retryButton.gameObject.SetActive(!isActive);
}

private void UpdateHiScore()
{
    var hiScore = PlayerPrefs.GetFloat("hiScore", 0);
    if (_score > hiScore)
    {
        hiScore = _score;
        PlayerPrefs.SetFloat("hiScore", hiScore);
    }
    hiScoreText.text =
Mathf.FloorToInt(hiScore).ToString("D5");
}
}

```

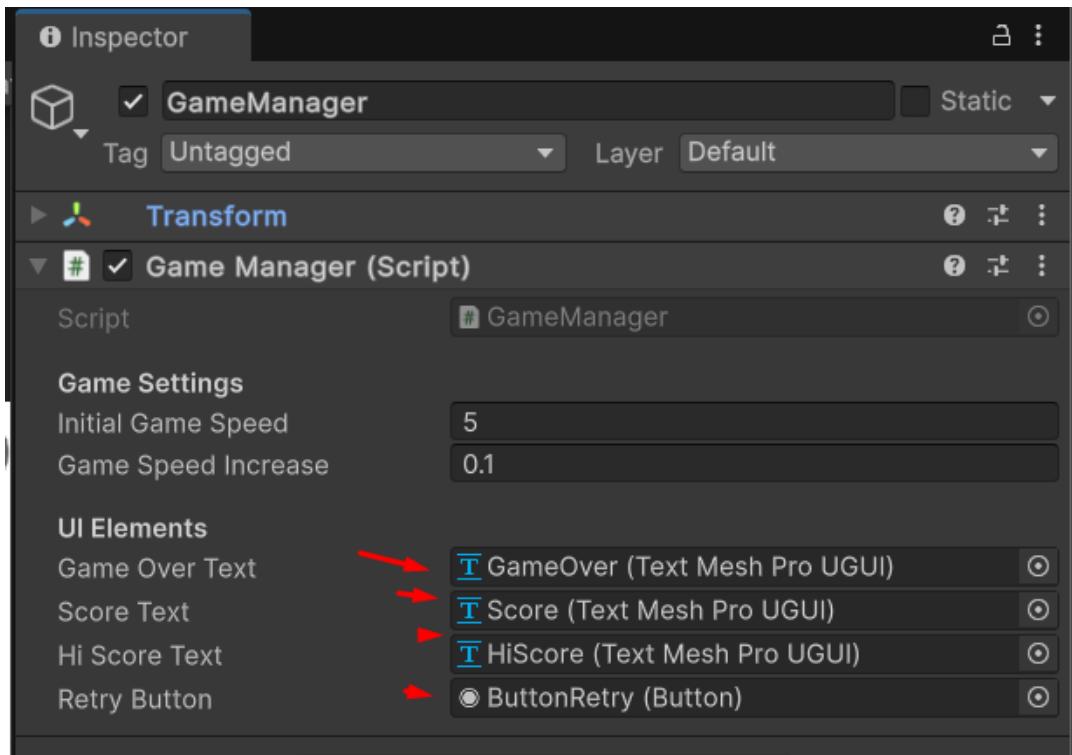
18. Открываем скрипт **Player** и напишем метод **OnTriggerEnter()** для столкновений с объектами (обратите внимание что мы вызываем **обычный метод**, а не **2D**):

```

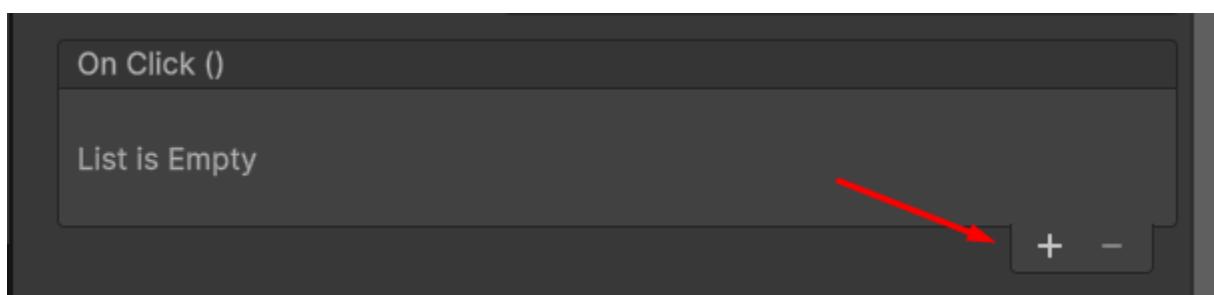
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Obstacle"))
    {
        GameManager.Instance.GameOver();
    }
}

```

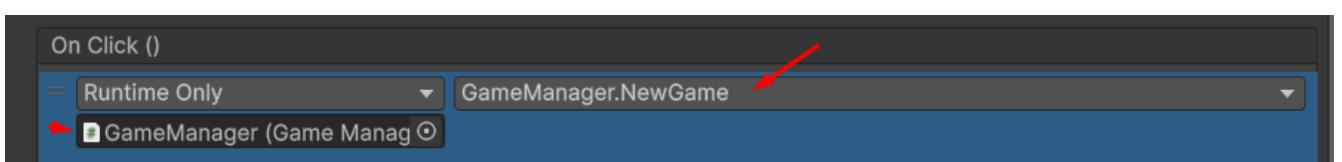
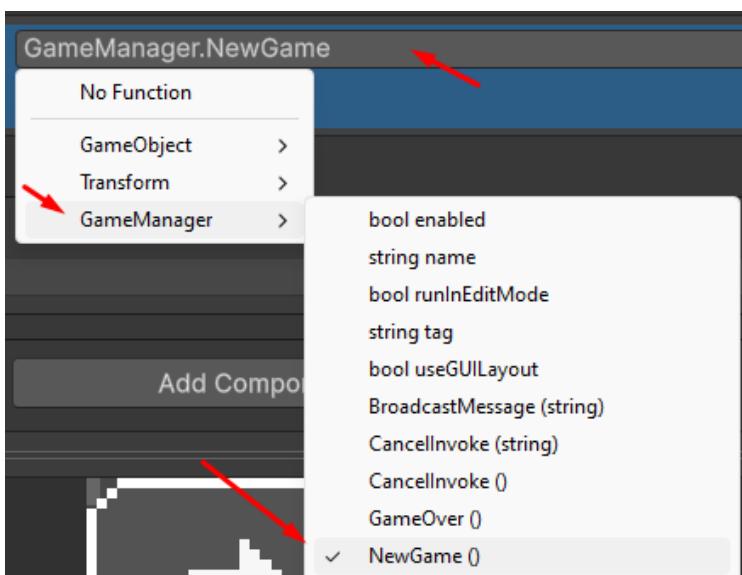
19. Сохраняем скрипты и возвращаемся в юнити. В объект **GameManager** нам нужно перенести в соответствующие поля тексты и кнопку:



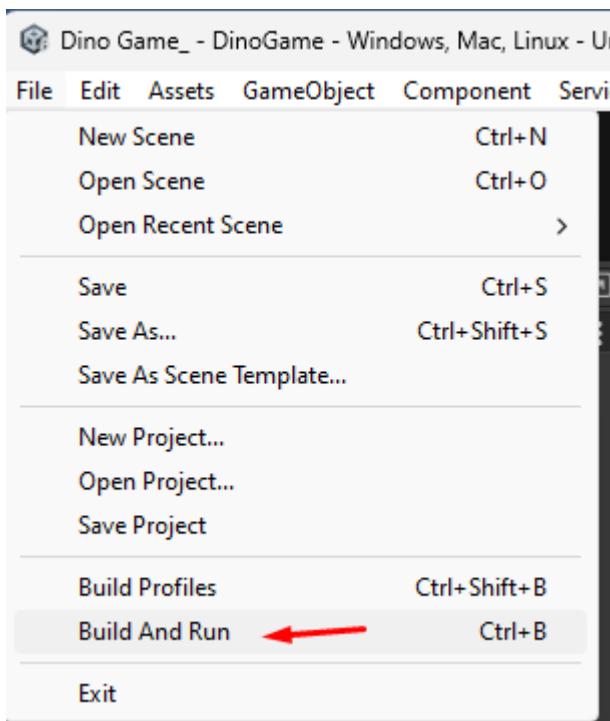
Для нашей кнопки **ButtonRetry** нужно добавить перезапуск игры, щёлкаем на ПЛЮСИК:



Переносим наш объект **GameManager** и у него выбираем функцию **New Game**:



20. . 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.