Лабораторная работа. Создание 2D-игры Pong.

Данный урок создан на основе следующего видео-урока: How to make Pong 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 объект "Ball", "Player Paddle", "Computer Paddle" с соответствующими физическими параметрами и компонентами.
- Создать границы игрового поля (стены) и линию, разделяющую поле по центру.

3. Разработка игровой логики:

- Написать скрипты для управления игроком и компьютером, используя наследование от общего класса Paddle.
- Настроить поведение мяча, включая начальное направление движения и реакцию на столкновения.
- Создать и настроить скрипты для подсчета и отображения очков.

4. Улучшение игровых механик:

- Добавить возможность изменения скорости мяча при столкновениях.
- Реализовать методику увеличения скорости мяча со временем для повышения сложности игры.

5. Интерфейс пользователя и звуковое сопровождение:

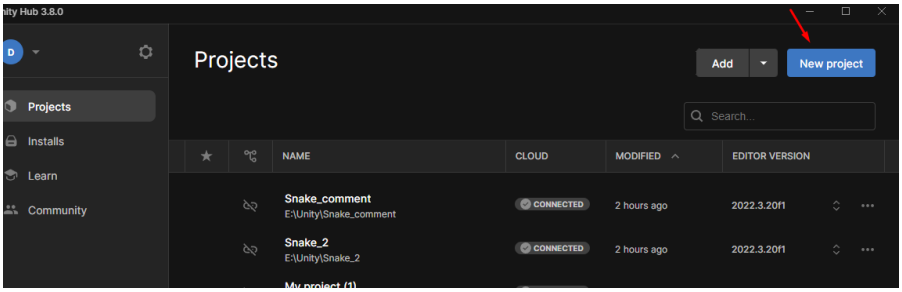
- Создать и настроить элементы интерфейса для отображения счета.
- Добавить звуковое сопровождение, включая фоновые звуки и звуки столкновений.

6. Финальные настройки и сборка проекта:

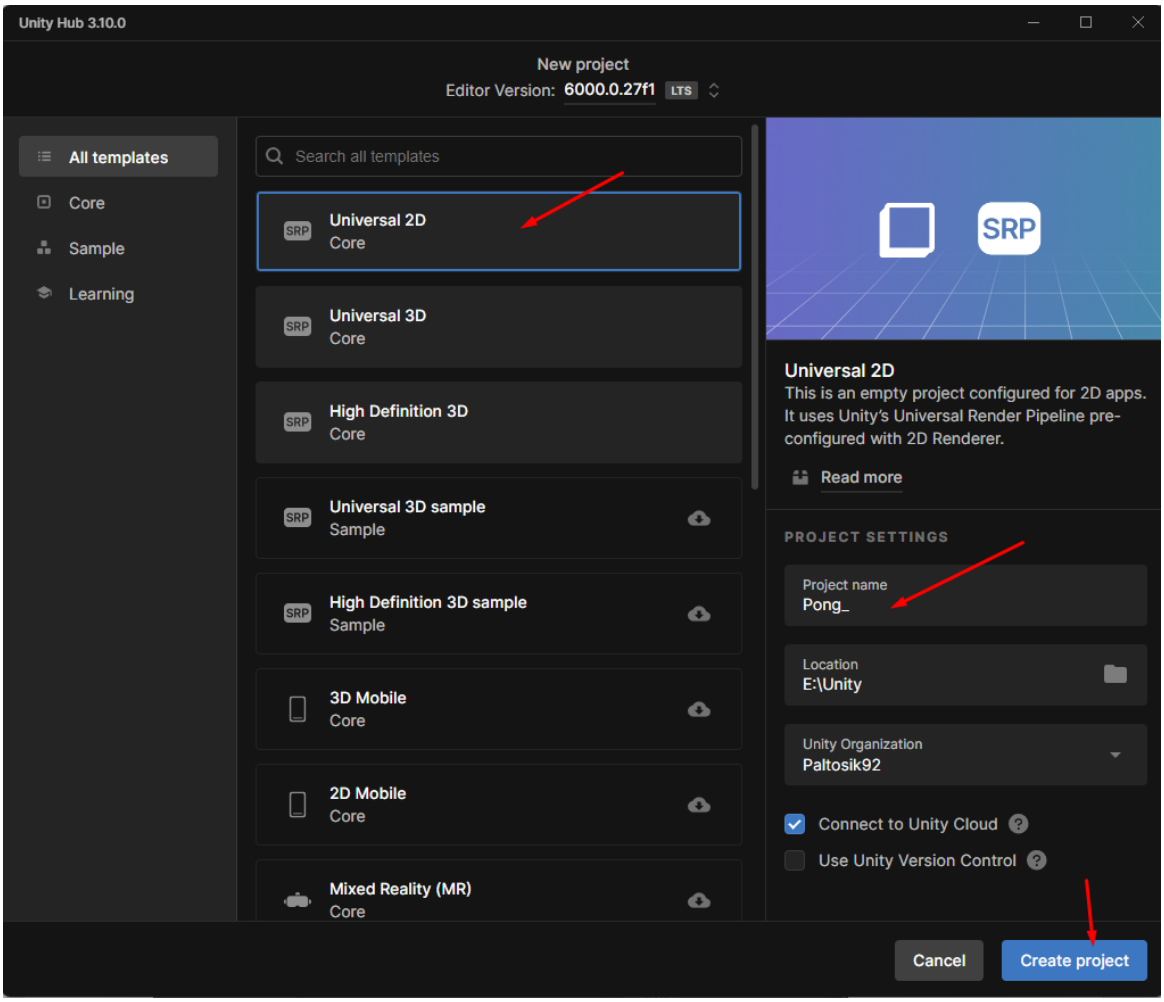
- Провести рефакторинг кода для улучшения читаемости и производительности.
- Скомпилировать и сохранить финальную версию игры.

1. Запускаем **Unity Hub**.

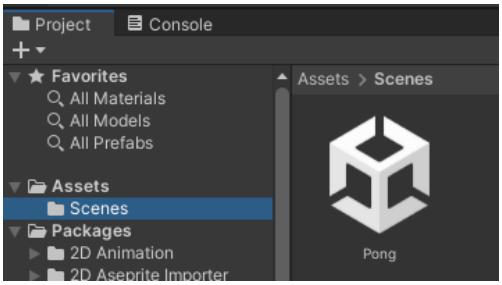
2. Создаём новый проект – **New project**:



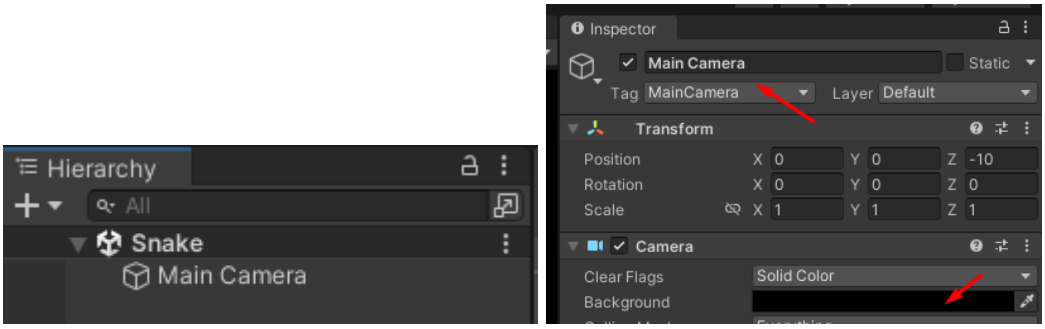
3. Выбираем **2D**. Вводим название проекта, выбираем место расположения, и нажимаем **Create project**.



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

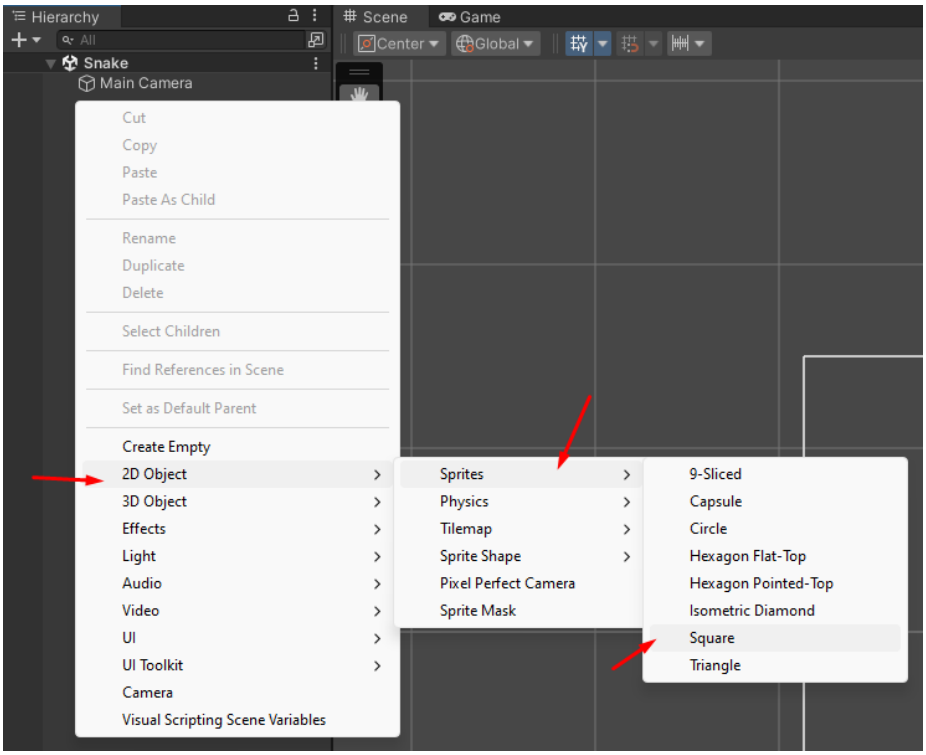


Меняем цвета объекта **Main Camera** (нажимаем на него в **Hierarchy**, и в **Inspector** появляются свойства объекта) на **чёрный**:

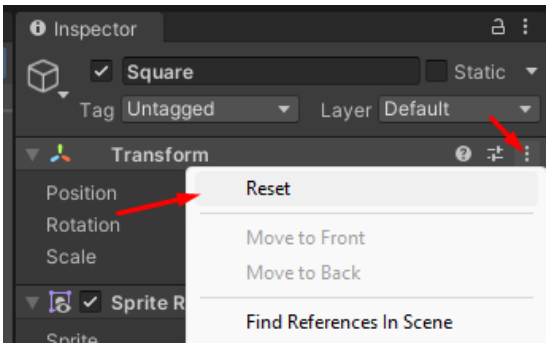


5. Создадим наш игровой объект мяч – **Ball.**

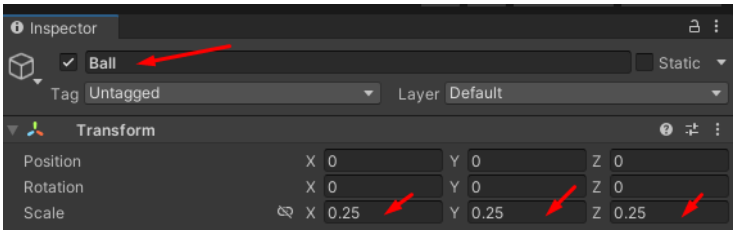
Щёлкаем правой кнопкой мыши в **Hierarchy → 2D Object → Sprites → Square:**



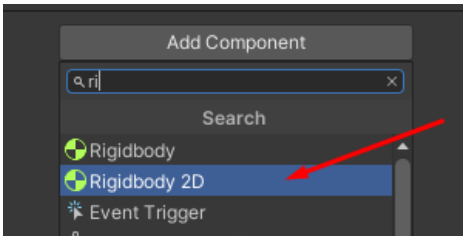
В **Inspector** сбрасываем для нашего объекта трансформацию (не забудьте, что объект должен быть выделен):



Изменим размер нашего мяча – параметр **Scale** и заодно сразу переименуем его на **Ball**:

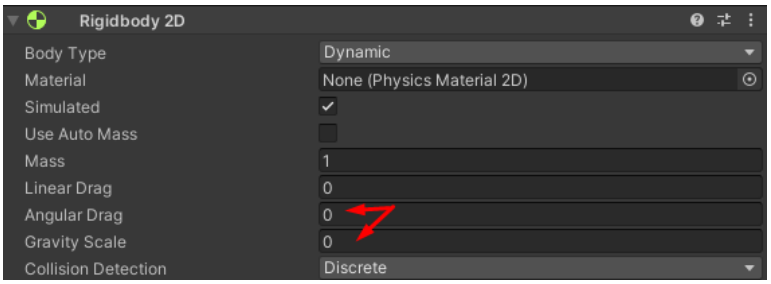


Добавляем компонент **Rigidbody 2D**:

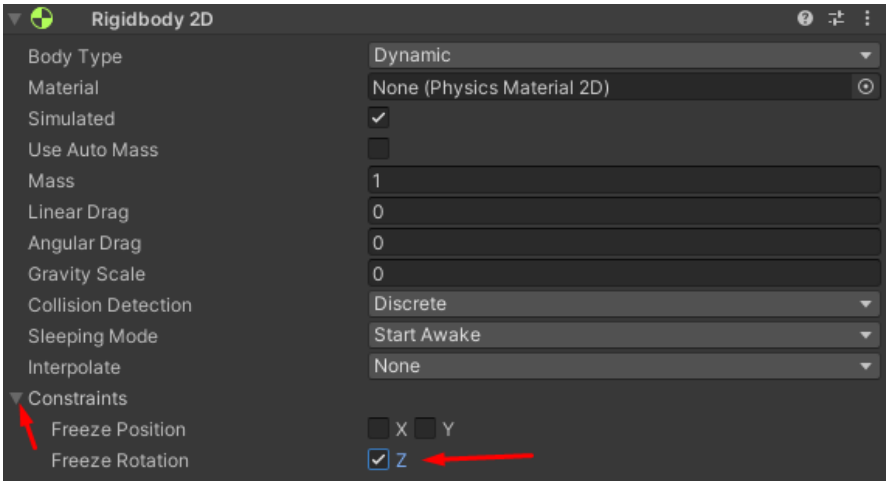


Rigidbody 2D – Добавление компонента (класса) Rigidbody2D к спрайту передает его под контроль физического движка. Само по себе это означает, что на спрайт будет воздействовать сила тяжести, и им можно управлять из скриптов с помощью сил. При добавлении соответствующего компонента collider спрайт также будет реагировать на столкновения с другими спрайтами.

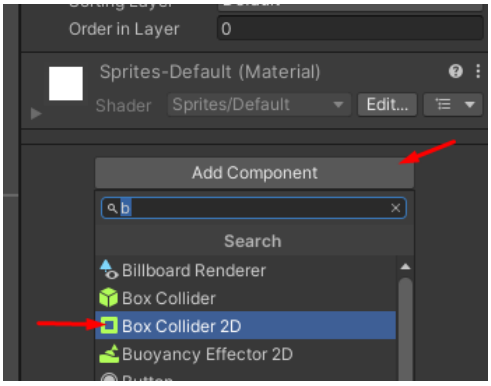
В нашей игры нам не нужны такие параметры как **гравитация** и **Angular Drag** (отвечает за сопротивление), поэтому убираем их в **0**:



Т.к. у нас **2-D** игра уберём с вами вращение по оси **Z,** «заморозим» её, для этого раскройте **Contains – Freeze Rotation:**

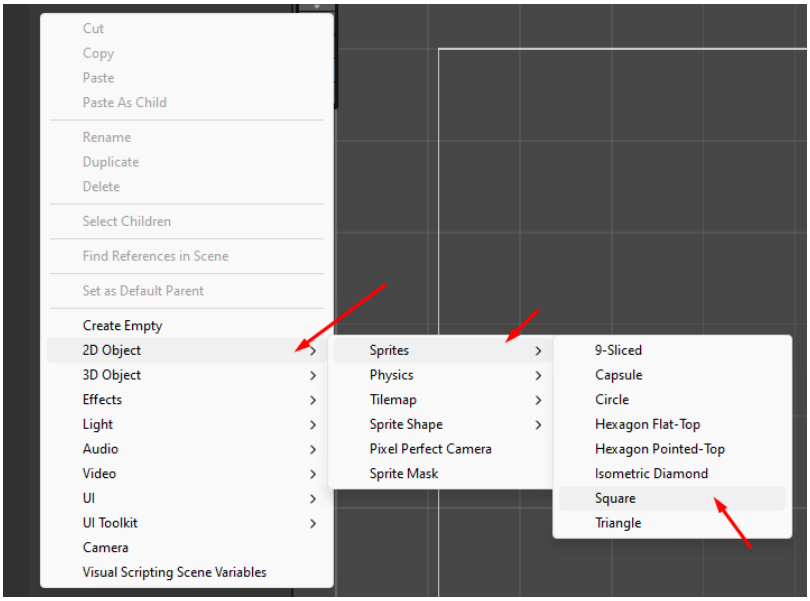


Добавим компонент **Box Collider 2D**:

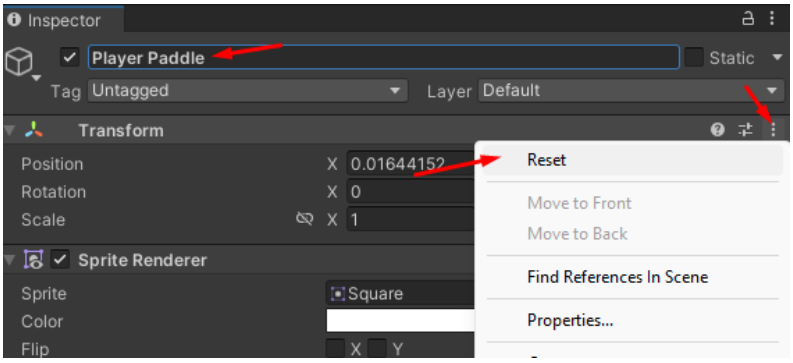


Box Collider 2D – Коллайдер для 2D-физики, представляющий собой прямоугольник, выровненный по оси, влияет на столкновение объектов.

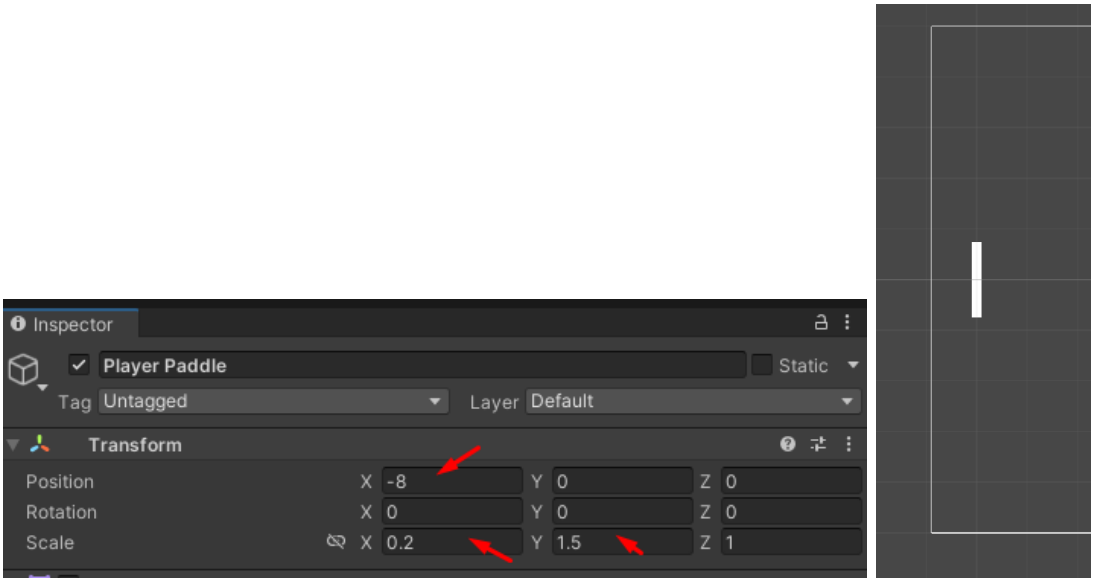
6. Далее создадим наши ракетки. В **Hierarchy → 2D Object → Sprites → Square:**



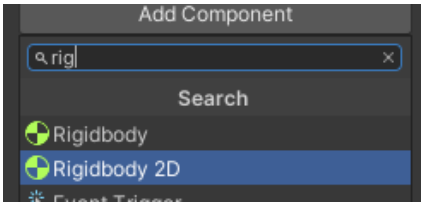
Меняем название на **Player Paddle** и сбрасываем трансформацию:



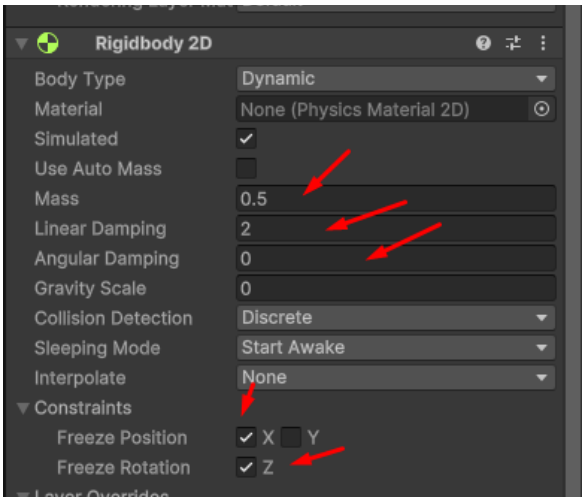
Меняем позицию по **X** и изменяем размер, чтобы наша ракетка встала в левом краю:



Добавляем ей **Rigidbody 2D:**



Выставим следующие значения – массу (**Mass**) уменьшим до 0.5, линейное сопротивление (**Linear Damping**) выставим на 2, угол сопротивления (**Angular Damping**) на 0, и заморозим позицию по **X**, вращение по **Z**:



Масса твердого тела.

Масса указывается в произвольных единицах, но основные физические принципы массы применяются. Классическое уравнение Ньютона `force = mass x acceleration` показывает, что чем больше масса объекта, тем больше силы требуется для его ускорения до заданной скорости. Масса также влияет на импульс, который важен во время столкновений; объект с большой массой будет меньше смещен при столкновении, чем объект с меньшей массой.

Линейное затухание линейной скорости (Linear Damping)

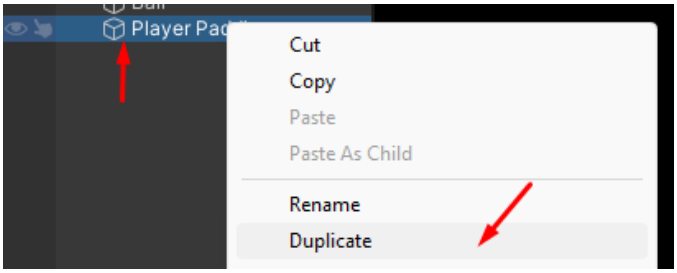
Демпфирование может использоваться для уменьшения величины Rigidbody2D.linearVelocity (линейной скорости) Rigidbody2D с течением времени.

Ноль указывает на то, что демпфирование не должно использоваться, тогда как более высокие значения увеличивают демпфирование, эффективно замедляя линейное движение быстрее. В отличие от контактного трения, линейное демпфирование применяется всегда.

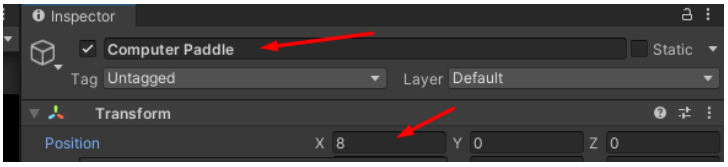
Коэффициент углового торможения (Angular Damping)

"Торможение" - тенденция объекта, которая позволяет его замедлить из-за трения с воздухом или водой, окружающих объект. "Угловое торможение" относится к вращательному движению и устанавливается отдельно от «линейного торможения», которое воздействует на позицию движения. Более высокое значение углового торможения приведет к тому, что вращаемый объект остановится более быстро после столкновения или крутящего момента.

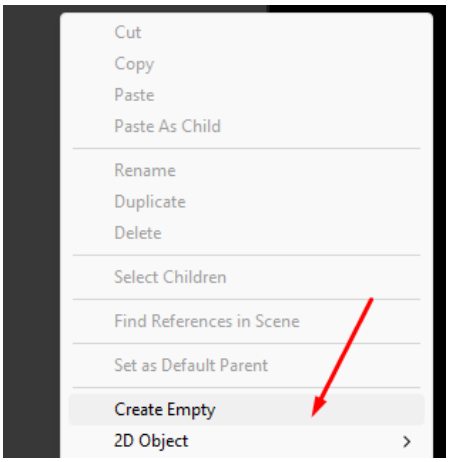
7. Добавим ракету для нашего противника. Создадим дубликат уже созданной, для этого нажмите правой кнопкой мыши по **Player Paddle** – **Duplicate** (или горячие клавиши **Ctrl+D**):



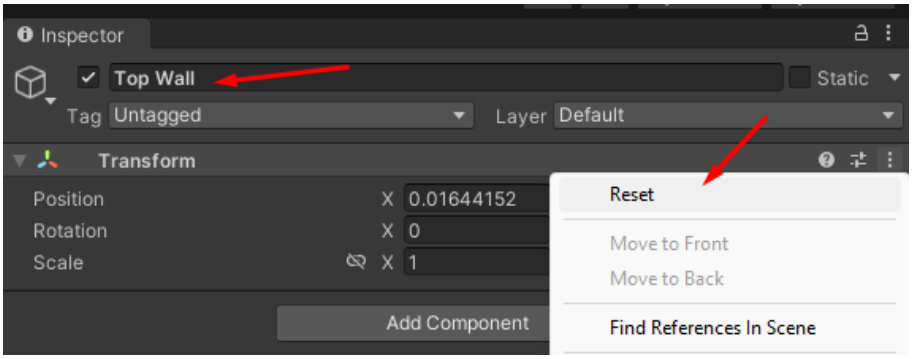
Меняем название на **Computer Paddle** и меняем позицию по **X** на противоположную:



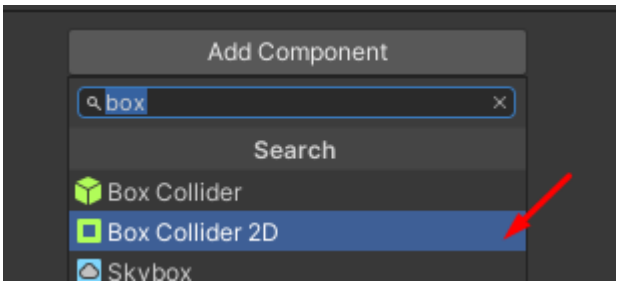
8. Создаём новый объект – **Creaty Empty**:



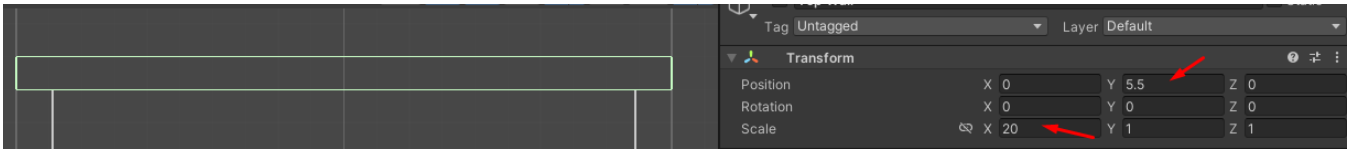
Называем его **Top Wall** и сбрасываем трансформацию:



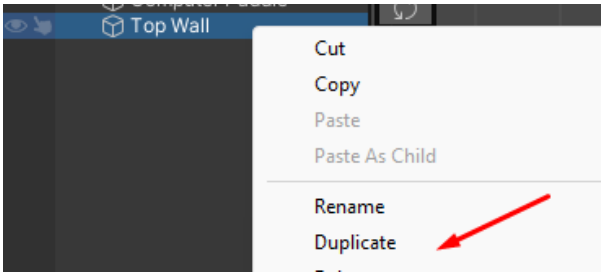
Добавляем **Box Collider 2D:**

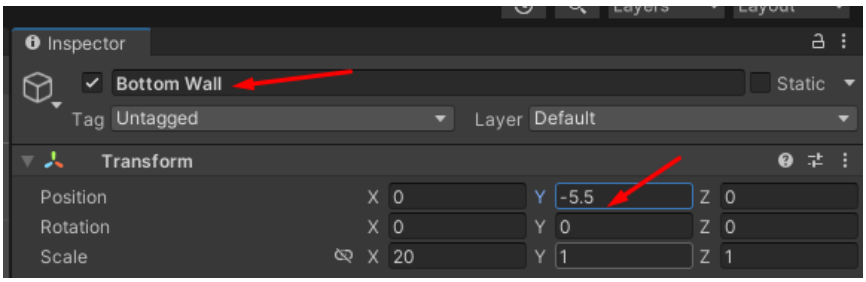


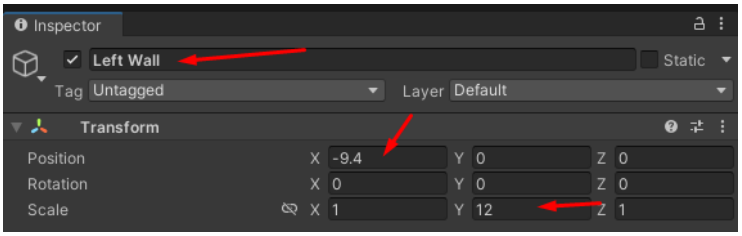
Настраиваем размер и позицию по Y, чтобы он занимал всё место выше камеры:

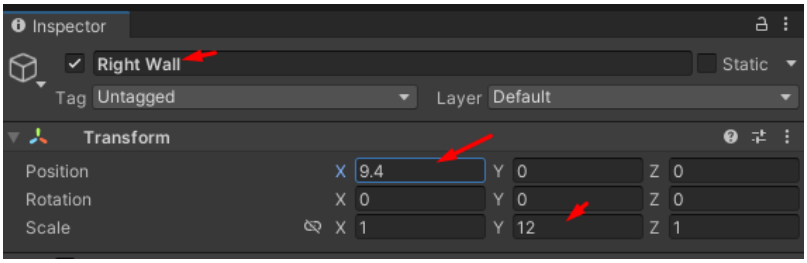


9. Далее создаём его **дубликаты** и делаем **верхнюю**, **левую** и **правую** стены:

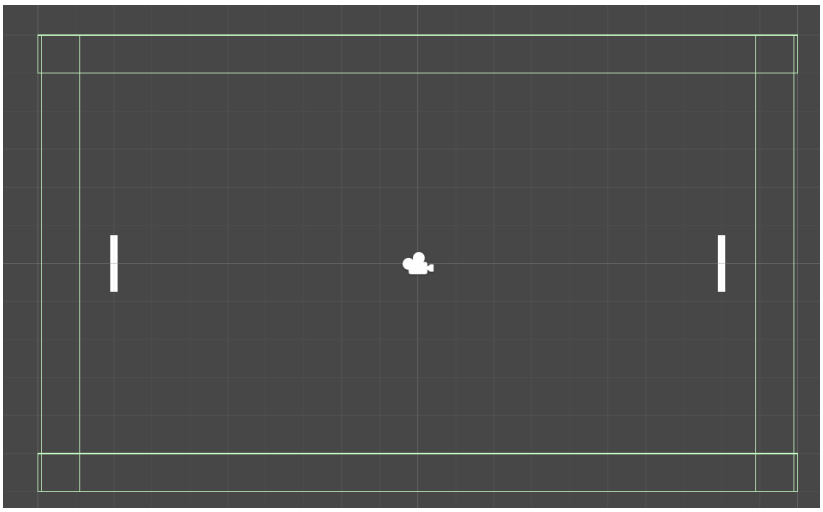




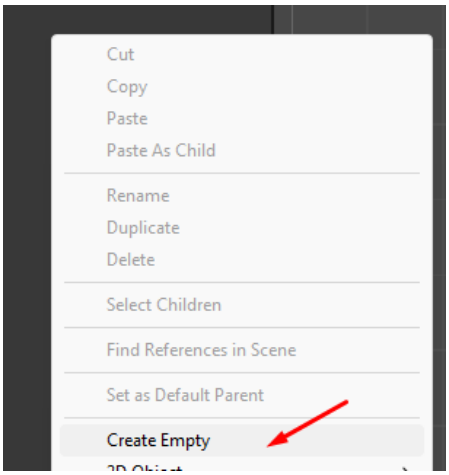




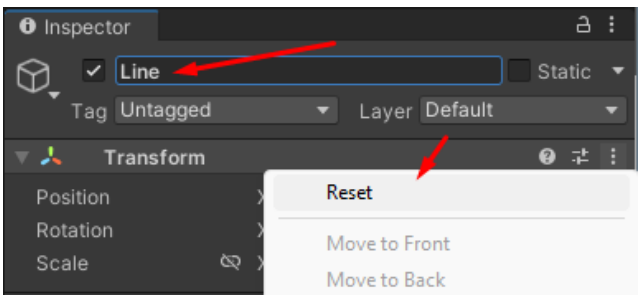
В итоге получится следующее:



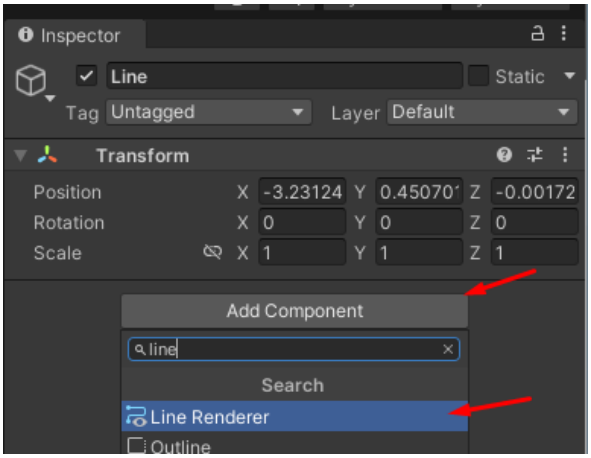
10. Создадим линию, разделяющую поле по центру - **Creaty Empty**:



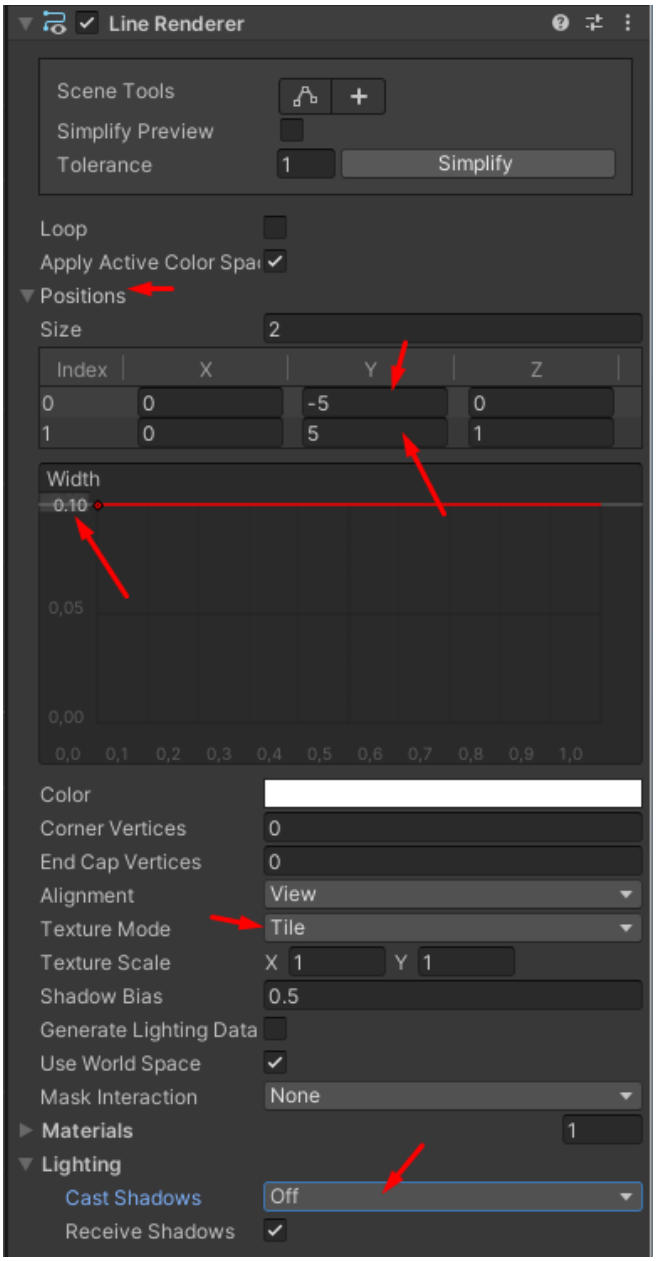
Переименовываем его на **Line** и делаем **Reset**:



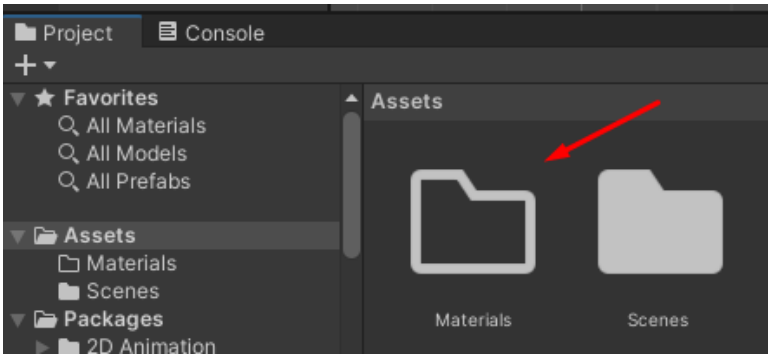
Добавляем ему компонент **Line Renderer:**



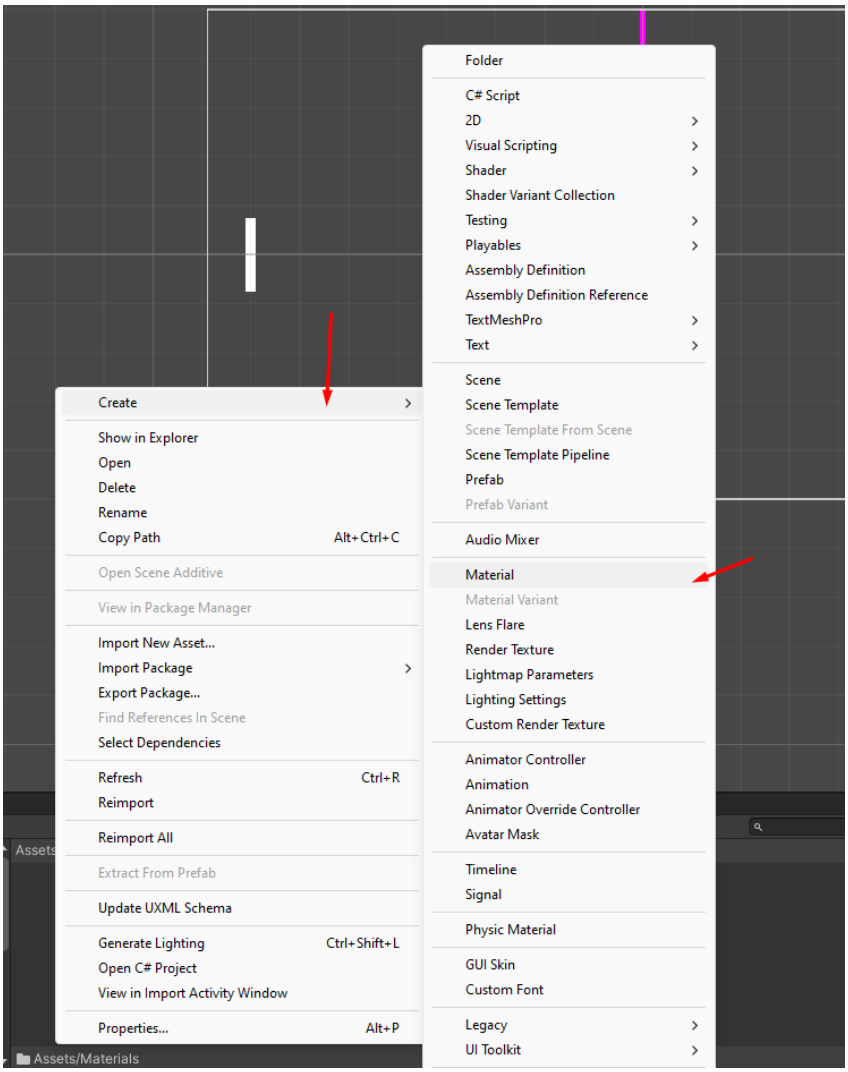
Раскрываем пункт **Position** и меняем Y на **-5** по 0 индексу, и на **5** по 1 индексу. Ширину поменяем на **0.1**. **Texture Mode** меняем на **Tile**. **Cast Shadows** на **Off**.



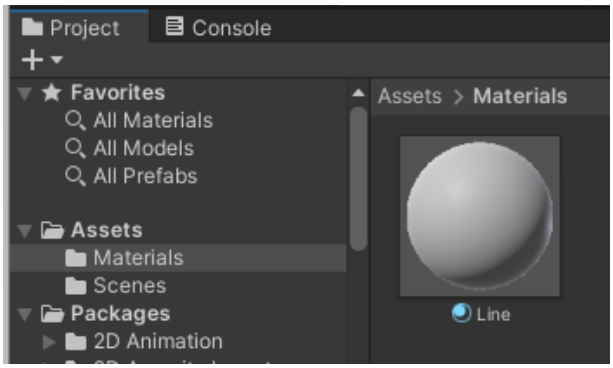
Создадим в нашей папке с ассетами новую папку – **Materials:**



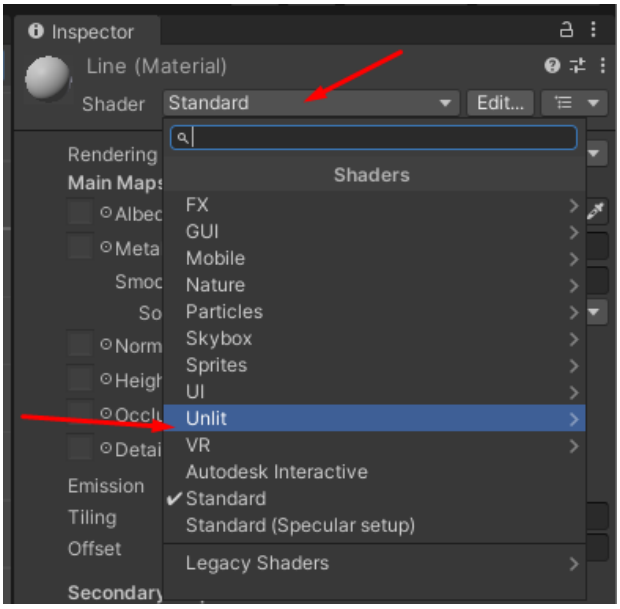
Нажимаем ПКМ - **Create – Material:**

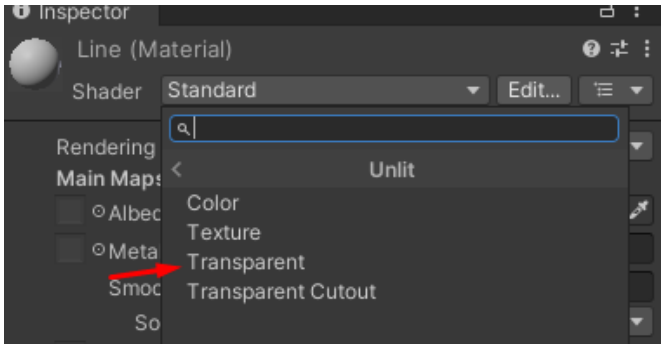


Называем его **Line**:



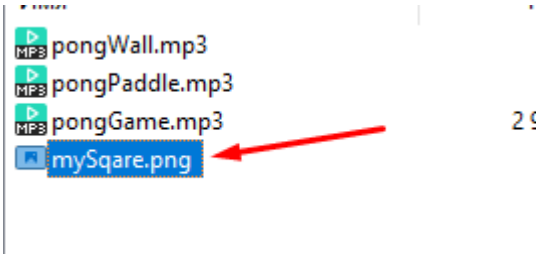
В **Inspector** поменяем шейдер на **Unit-Transparent**:



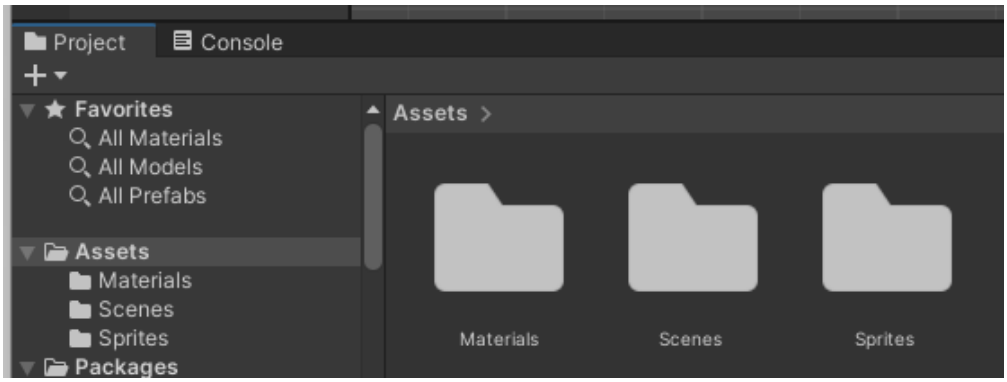


Если интересно узнать, как создать свою текстуру то можете почитать в конце лабораторной работы в **дополнительно.**

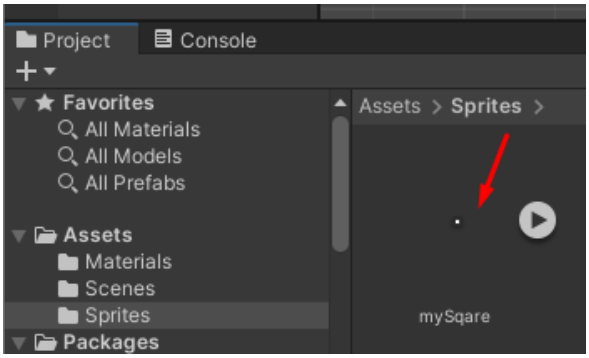
Мы же используем готовый **png** файл из папки с архивом файл **mySqare:**



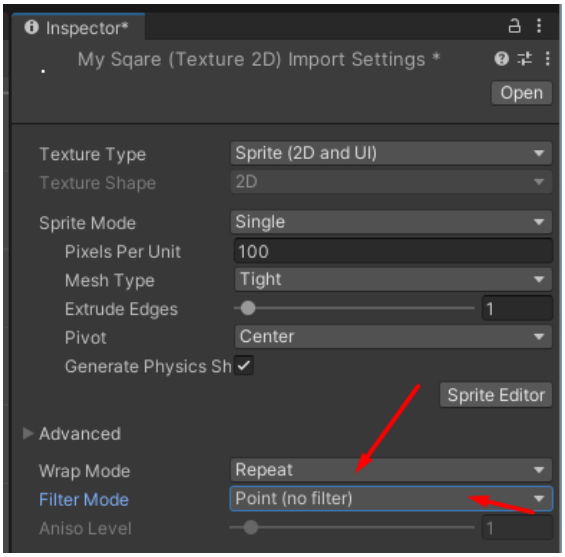
В папке с ассетами создайте папку **Sprites**:



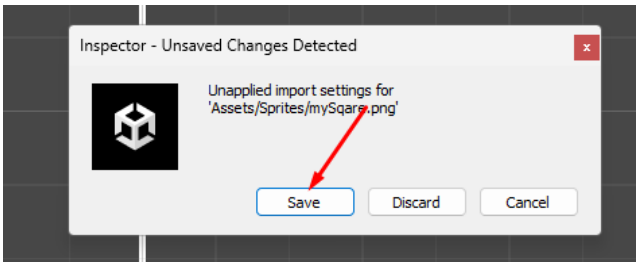
Перенесите в неё созданное вами изображение **mySqare:**



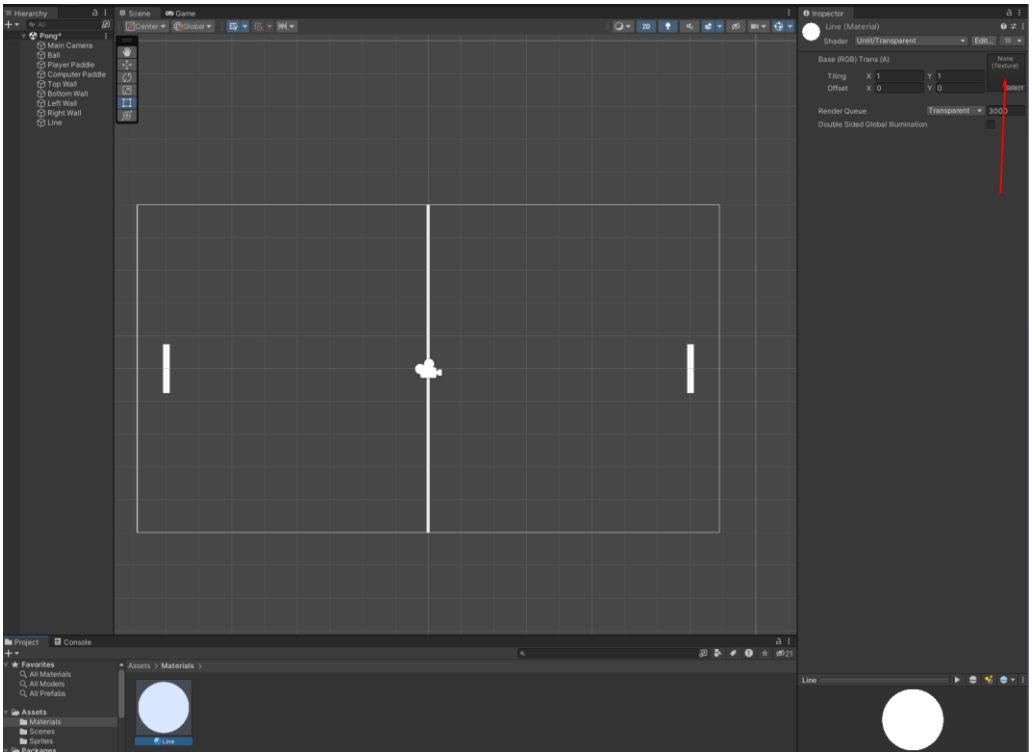
Нажмите на него, и в **Inspector** поменяйте **Wrap Mode** на **Repat,** чтобы он повторялся и **Filter Mode** выберите **Point,** чтобы не было замыливание текста**:**

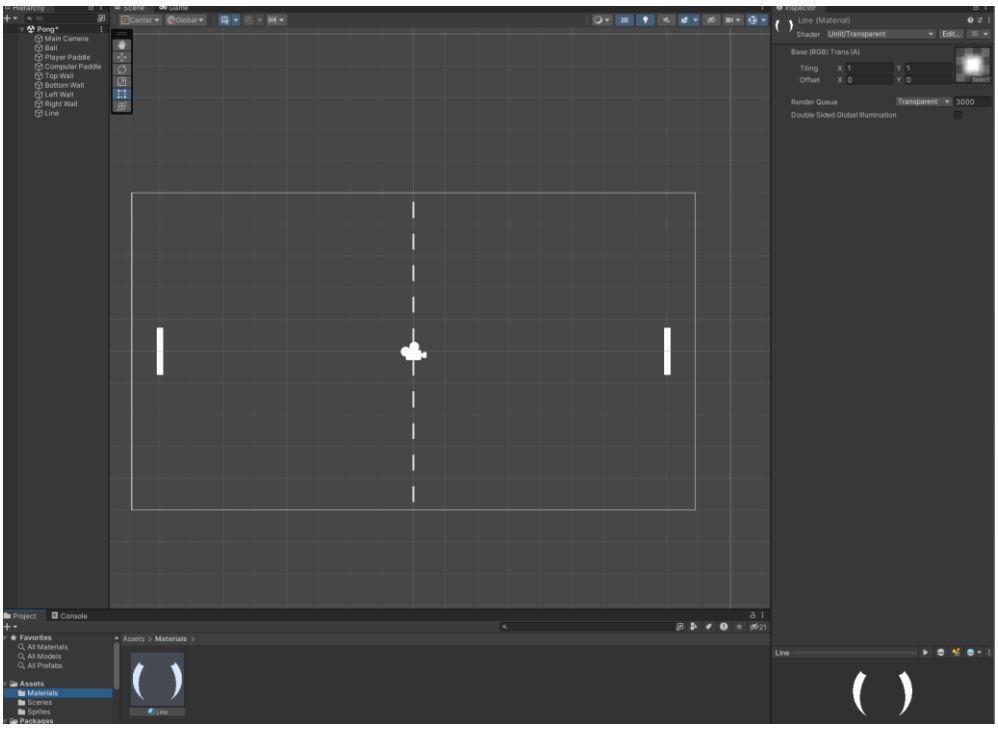


Щёлкните в любом месте, и **Unity** предложит вам сохранить, нажмите **Save**:

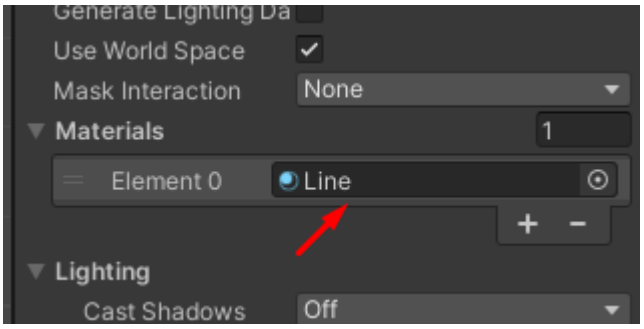


Далее вам нужно будет перенести спрайт **mySqare** в поле текстуры вашего материала:

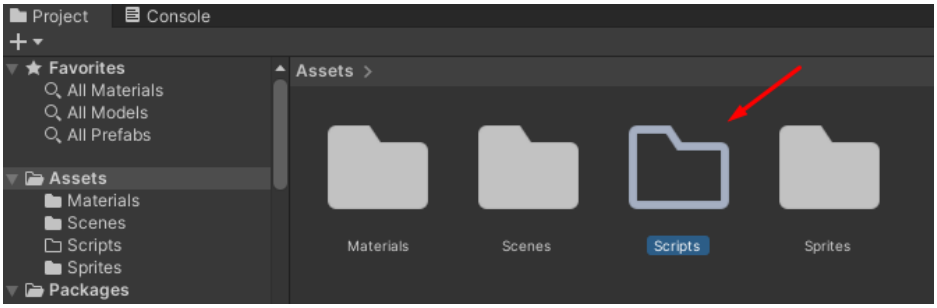




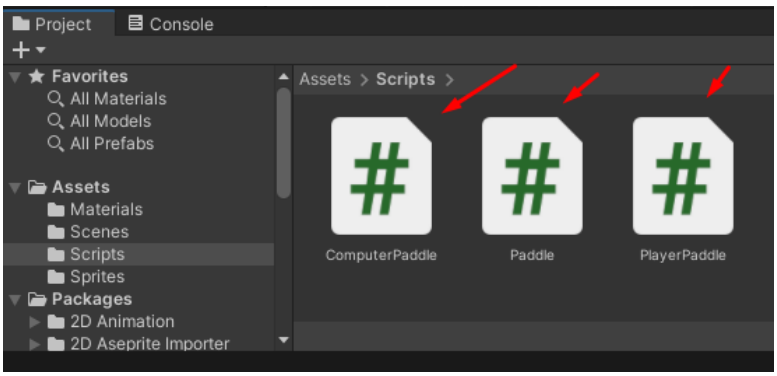
Переносим к нашему **Line** материал:



11. Создаём папку **Scripts**:



В ней создадим три скрипта **Paddle, PlayerPaddle** и **ComputerPaddle**



Открываем три скрипта и удалим ненужные строчки кода:

```
PlayerPaddle.cs | ComputerPaddle.cs | Paddle.cs
Assembly-CSharp | Paddle
1  using UnityEngine;
2
   Скрипт Unity | Ссылок: 0
3  public class Paddle : MonoBehaviour
4  {
5
6  }
```

```
PlayerPaddle.cs | ComputerPaddle.cs | Paddle.cs
Assembly-CSharp | PlayerPaddle
1  using UnityEngine;
2
   Скрипт Unity | Ссылок: 0
3  public class PlayerPaddle : MonoBehaviour
4  {
5
6  }
```

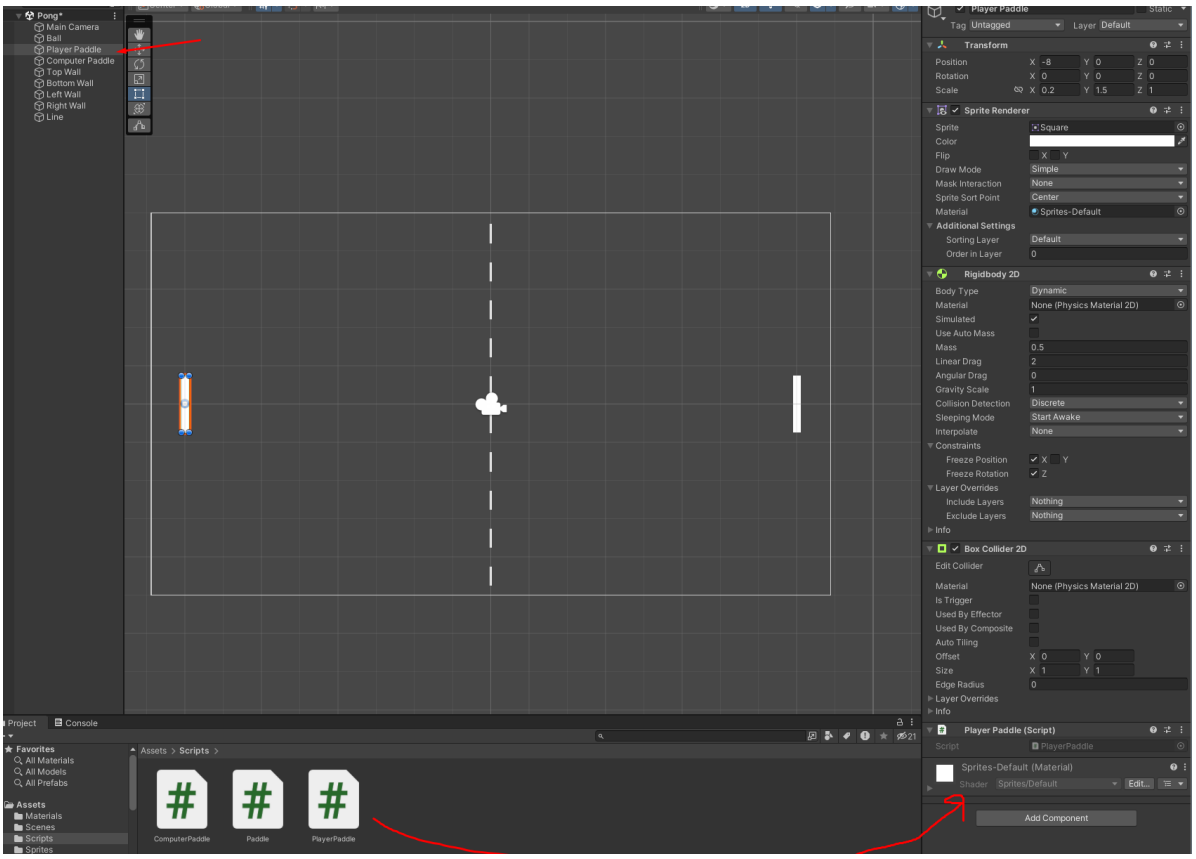
```
PlayerPaddle.cs | ComputerPaddle.cs | Paddle.cs
Assembly-CSharp | ComputerPaddle
1  using UnityEngine;
2
   Скрипт Unity | Ссылок: 0
3  public class ComputerPaddle : MonoBehaviour
4  {
5
6  }
```

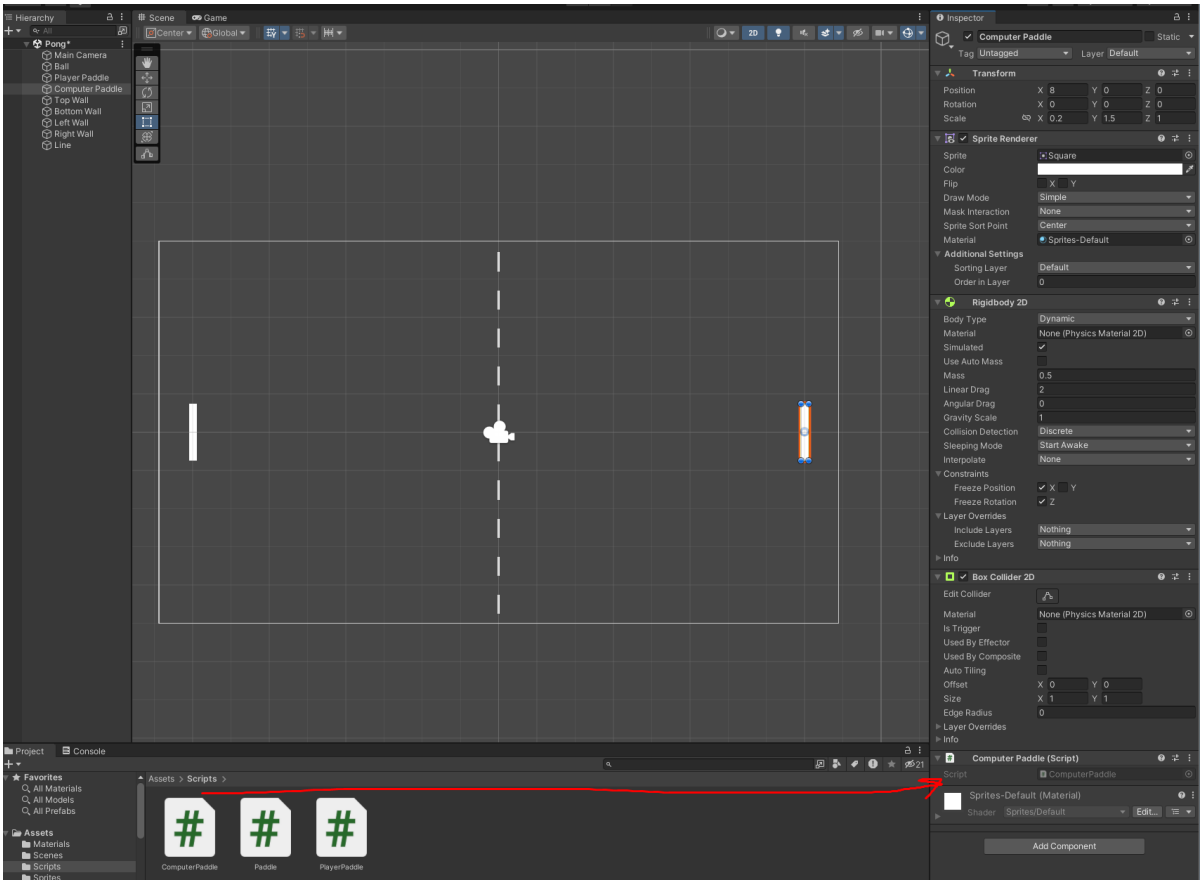
Наши скрипты игрока и компьютера будут наследовать от класса **Paddle**, пропишем это:

```
PlayerPaddle.cs | ComputerPaddle.cs | Paddle.cs
Assembly-CSharp | PlayerPaddle
1  using UnityEngine;
2
   Скрипт Unity | Ссылок: 0
3  public class PlayerPaddle : Paddle
4  {
5
6  }
```

```
PlayerPaddle.cs | ComputerPaddle.cs | Paddle.cs
Assembly-CSharp | ComputerPaddle
1  using UnityEngine;
2
   Скрипт Unity | Ссылок: 0
3  public class ComputerPaddle : Paddle
4  {
5
6  }
```

В Unity добавляем скрипт для игрока и компьютера на их игровые объекты:





12. Для **Paddle** пишем следующий код:

```
using UnityEngine;

public class Paddle : MonoBehaviour
{
    public float speed = 10.0f; // объявление публичной
переменной (поля) типа float с именем “speed”. Переменная
имеет начальное значение 10.0.

    protected Rigidbody2D _rigidbody; // объявление
защищенной (protected) переменной (поля) типа Rigidbody2D с
именем “_rigidbody”. Эта переменная будет хранить ссылку на
компонент Rigidbody2D.

    private void Awake() // метод “Awake”, который
вызывается при активации объекта.
    {
        _rigidbody = GetComponent<Rigidbody2D>(); // Внутри
метода _rigidbody получает ссылку на компонент Rigidbody2D,
прикрепленный к этому объекту.
    }
}
```

13. Для **PlayerPaddle** пишем следующий код:

```
using UnityEngine;

public class PlayerPaddle : Paddle // объявление класса с
именем “PlayerPaddle”, который наследует функциональность
от класса “Paddle”.
{
    private Vector2 _direction; // объявление приватной
переменной (поля) типа Vector2 с именем “_direction”. Эта
переменная будет хранить направление движения.
    private void Update() // метод “Update”, который
вызывается каждый кадр.
    {

        // внутри метода проверяется, какие клавиши (W,
UpArrow, S, DownArrow) нажаты, и устанавливается
соответствующее направление в переменную “_direction”.
```

```
        if (Input.GetKey(KeyCode.W) ||
Input.GetKey(KeyCode.UpArrow))
            _direction = Vector2.up;
        else if (Input.GetKey(KeyCode.S) ||
Input.GetKey(KeyCode.DownArrow))
            _direction = Vector2.down;
        else
            _direction = Vector2.zero;
    }

    private void FixedUpdate()
    {
    //linearVelocity - Вектор линейной скорости твердого
тела. Он представляет собой скорость изменения положения
твердого тела.
    //Умножаем направление на скорость и устанавливаем это
значение как скорость для Rigidbody2D.
    _rigidbody.linearVelocity = _direction * this.speed;
    }
}
```

***В старых версиях Unity используйте _rigidbody.velocity**

В данном случае в методе **FixedUpdate** мы осуществляем прямое задание скорости объекта.

Особенности:

- Полный контроль над скоростью объекта.
- Не зависит от физических сил, таких как трение или гравитация.

Плюсы:

- Простота реализации.
- Полный контроль над движением.

Минусы:

- Физика Unity может игнорироваться, что делает поведение менее реалистичным.
- Может требовать дополнительной настройки для плавного изменения скорости.

Другие реализации **FixedUpdate:**

2 способ:

```
  private void FixedUpdate()
    {
    if (_direction.sqrMagnitude != 0)
            _rigidbody.AddForce(_direction * this.speed);
    }
}
```

В данном случае мы используем метод **AddForce,** который добавляет силу к объекту непосредственно через физический движок Unity.

Особенности:

- Подходит для симуляции реалистичной физики.

- Учитывает массу, трение, сопротивление воздуха и другие физические силы.
- Подходит для объектов с инерцией.

Плюсы:

- Реалистичное поведение в физической среде.

Минусы:

- Требует дополнительных проверок, чтобы не прикладывать силу к объекту, когда она не нужна.

Дополнительно мы используем проверку, чтобы убедиться, что вектор **_direction** не равен нулю. Вектор направления **_direction** принимает значение **Vector2.zero** (нулевой вектор) в случае, если ни одна из клавиш для управления не нажата. Если бы мы не проверяли **_direction**, то могли бы добавить силу к объекту, даже если игрок ничего не нажимает, что могло бы привести к ошибочному поведению.

Что такое sqrMagnitude?

Vector2.sqrMagnitude — это квадрат длины (магнитуды) вектора. Он рассчитывается как сумма квадратов компонентов вектора:

sqrMagnitude=x^2+y^2

Зачем использовать **sqrMagnitude** вместо **magnitude**?

magnitude — это длина вектора, которая вычисляется как корень из квадрата длины:

magnitude=x^2+y^2

Однако вычисление квадратного корня — операция относительно дорогая с точки зрения производительности, особенно если она выполняется многократно в цикле. Если вам не требуется точная длина вектора, а нужно только проверить, ненулевой ли вектор, использование **sqrMagnitude** позволяет избежать вычисления квадратного корня, что делает код более производительным.

Использование **sqrMagnitude** в данном случае — это оптимизированный способ убедиться, что **_direction** не равен нулю, перед тем как применить силу к объекту. Это особенно важно в реальном времени, где каждая оптимизация кода может улучшить общую производительность игры.

3 способ:

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

1. **Vector2.Lerp** — это линейная интерполяция между двумя векторами. Она плавно изменяет текущую скорость **_rigidbody.velocity** по направлению к целевой скорости **_direction** * this.speed.

2. **Time.fixedDeltaTime** используется для учета времени между вызовами **FixedUpdate**, чтобы обеспечить плавное и предсказуемое изменение скорости.

3. Умножение на коэффициент **10f** регулирует скорость интерполяции. Чем выше коэффициент, тем быстрее **velocity** приближается к целевому значению.

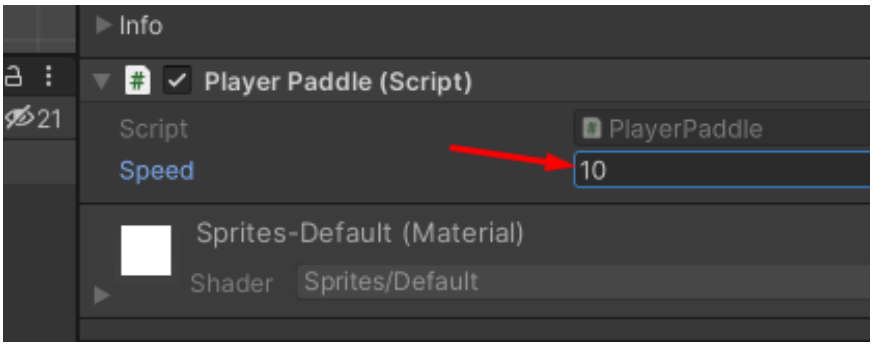
Плюсы:

- Плавное движение.
- Контролируемое затухание.
- Стабильность.

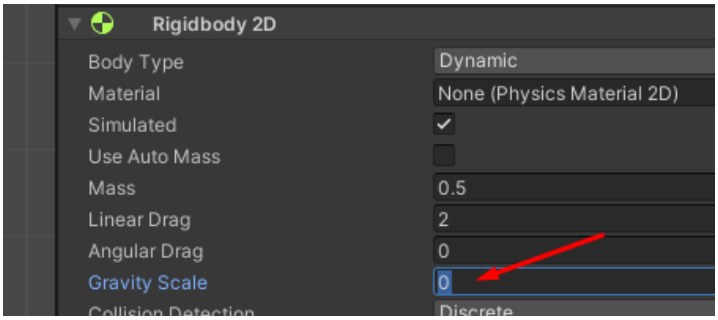
Минусы:

- Меньший контроль над физикой.
- Потеря реакции на мгновенные действия.
- Дополнительная настройка.

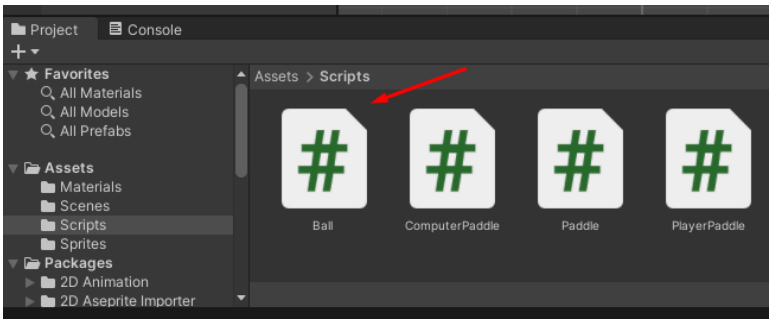
14. Для **PlayerPaddle** ставим значение переменной **Speed**, например на **10**:



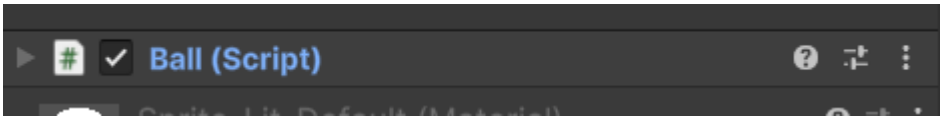
Давайте уберём гравитацию, чтобы каретка не падала вниз:



15. Создаём новый скрипт **Ball**:



И добавляем его к **мячу**:



Открываем наш скрипт.

И напишем следующий код (в версии **Unity 6**, если у вас начнёт ругаться на класс **Random** удалите включаемое в начале пространство имён - **using Unity.Mathematics;**).

```
using UnityEngine;

public class Ball : MonoBehaviour
{

    public float speed = 200.0f; // Это публичное поле, которое
определяет скорость мяча.

    private Rigidbody2D _rigidbody; // Приватное поле для хранения
ссылки на компонент Rigidbody2D.
    private void Awake() // Вызывается при инициализации объекта.
    {
        _rigidbody = GetComponent<Rigidbody2D>(); // Получает
ссылку на Rigidbody2D компонент, чтобы мы могли с ним
взаимодействовать.
    }

    private void Start() // Запускает метод AddStartingForce().
    {
        AddStartingForce();
    }

    public void AddStartingForce() // Генерирует случайное
направление движения мяча.
    {
        float x; // Если Random.value меньше 0.5, устанавливает x в
-1, иначе в 1.
        if (Random.value < 0.5f)
            x = -1.0f;
        else
            x = 1.0f;

        float y; // Генерирует случайное значение y в диапазоне (-
1, -0.5) или (0.5, 1).
        if (Random.value < 0.5f)
            y = Random.Range(-1.0f, -0.5f);
        else
            y = Random.Range(0.5f, 1.0f);

        Vector2 _direction = new Vector2(x, y); // Создает вектор
_direction с этими значениями.
        _rigidbody.AddForce(_direction * speed); // Добавляет силу
к Rigidbody2D, чтобы мяч начал двигаться.
    }
}
```

Зачем брались такие диапазоны:

-1.0f и 1.0f для x: Это значение определяет направление движения объекта по горизонтали (влево или вправо). Использование четких значений -1.0 и 1.0 обеспечивает, что объект будет двигаться полностью влево или вправо.

Random.value возвращает случайное число в диапазоне [0, 1) (включительно 0, но не включая 1). Условие **if (Random.value < 0.5f)** делит этот диапазон пополам:

Если **Random.value** меньше **0.5**, выбирается **x = -1.0f**.

Если **Random.value** больше или равен **0.5**, выбирается **x = 1.0f.**

Таким образом, шанс того, что мяч начнёт движение влево или вправо, составляет **50%**.

Если бы в условии вместо **0.5** было использовано **0**, то выражение **Random.value < 0** никогда бы не выполнялось, поскольку минимальное значение **Random.value** равно **0**. В результате **x** всегда был бы равен **1.0f**, и мяч двигался бы только вправо.

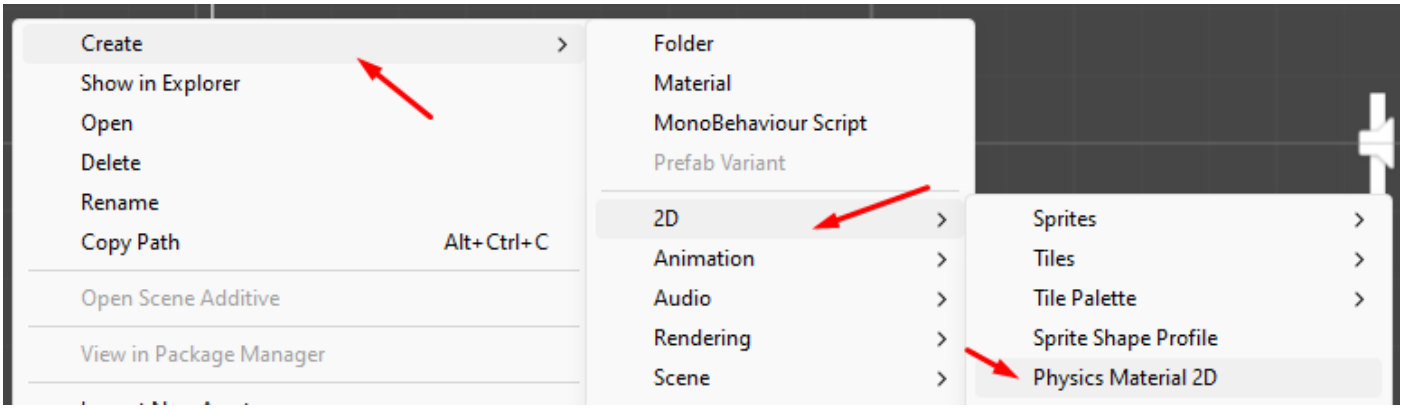
Именно поэтому используется значение **0.5** — оно гарантирует равную вероятность для выбора направления по оси **X**.

Диапазоны для y (-1.0f до -0.5f и 0.5f до 1.0f): Эти значения используются, чтобы объект двигался вверх или вниз, но с некоторым разбросом скорости. Это создает небольшой элемент случайности и делает движение менее предсказуемым. Использование диапазона от 0.5 до 1.0 или от -0.5 до -1.0 исключает очень медленное вертикальное движение, что делает игру более динамичной.

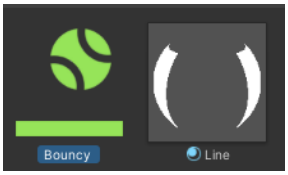
Мы можем сократить наш блок с **if-else**, через тернарный оператор:

```
        float x = Random.value < 0.5f ? -1.0f : 1.0f;
        float y = Random.value < 0.5f ? Random.Range(-1.0f,
-0.5f) : Random.Range(0.5f, 1.0f);
```

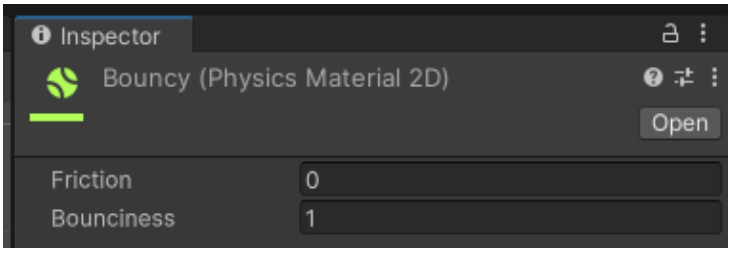
16. Создаём в папке **Materials** –**Physic Material 2D**:



Назовём его **Bouncy**:



В **Inspector** меняем параметры **Friction = 0**, **Bounciness = 1**:



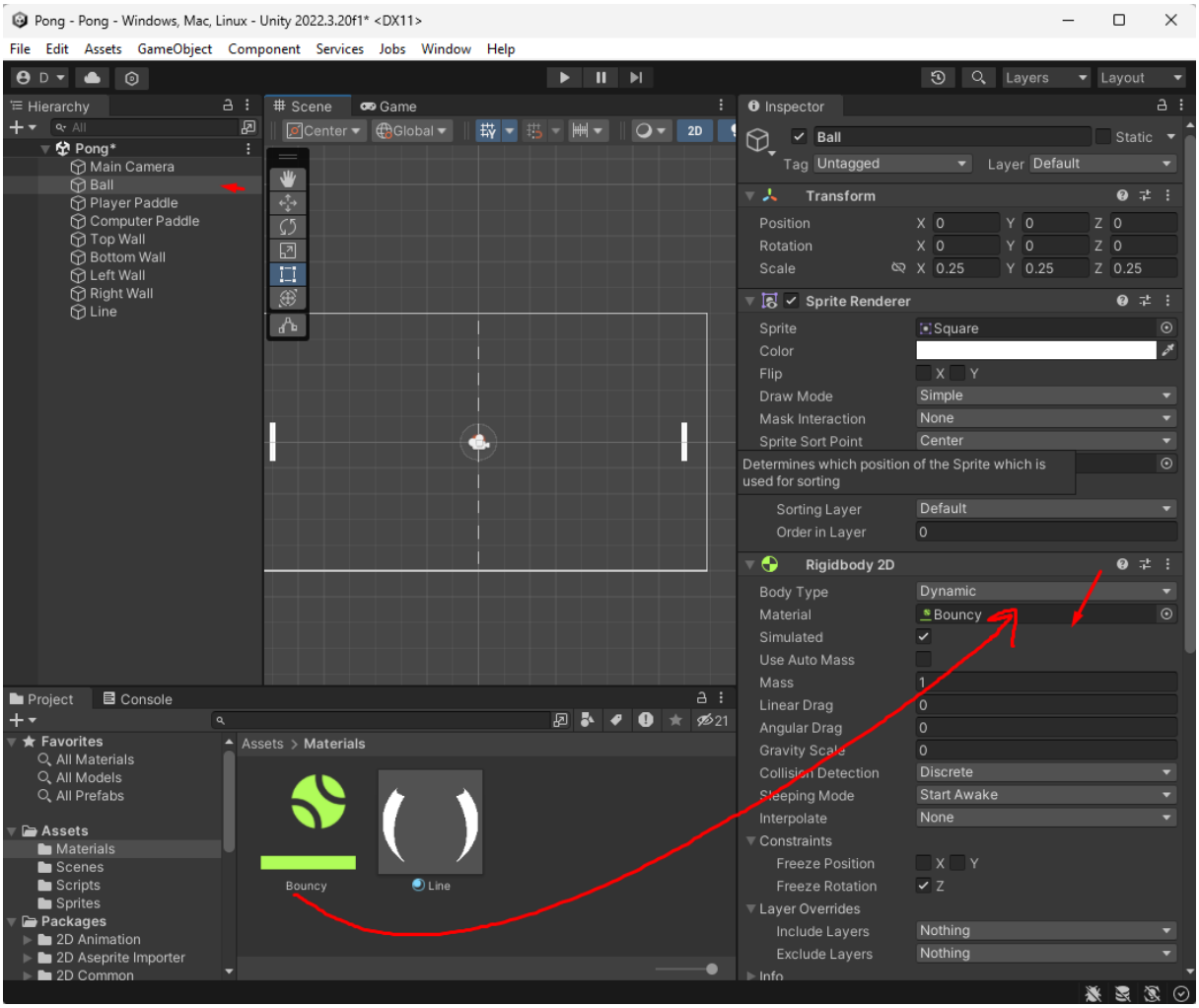
Friction - коэффициент трения.

Трение используется для управления тем, как реакция на столкновение снижает скорость. Значение, равное нулю, указывает на полное отсутствие трения, тогда как более высокие значения приводят к увеличению трения.

Bounciness - коэффициент восстановления.

Восстановление (прыгучесть) используется для управления тем, насколько «эластичным» является ответ на столкновение. Значение ноль указывает на полное отсутствие отскока, а значение единица указывает на идеальную эластичность (приблизительно).

Добавим наш материал **Bouncy** в **Ball – Rigidbody 2D:**



17. Теперь пропишем скрипт для **ComputerPaddle:**

```
using UnityEngine;

public class ComputerPaddle : Paddle
{
    public Rigidbody2D ball; // Это публичное поле, которое
хранит ссылку на компонент Rigidbody2D - мяч.

    private void FixedUpdate() // Вызывается на каждом
фрейме с фиксированным временем.
    {
        if (ball.linearVelocity.x > 0.0f) // Проверяет
скорость мяча, если мяч движется вправо
        {
            if (ball.position.y > transform.position.y)
                _rigidbody.AddForce(Vector2.up * speed); //
Если позиция мяча по вертикали (ball.position.y) выше
позиции платформы (transform.position.y), то добавляет силу
вверх (Vector2.up * speed)
            else if (ball.position.y <
transform.position.y)
                _rigidbody.AddForce(Vector2.down * speed);
// Иначе, если позиция мяча ниже позиции платформы,
добавляет силу вниз (Vector2.down * speed)
        }
        else // Иначе (если мяч движется влево)
        {
            if (transform.position.y > 0.0f)
                _rigidbody.AddForce(Vector2.down * speed);
// Если позиция платформы выше 0, добавляет силу вниз
            else if (transform.position.y < 0.0f)
                _rigidbody.AddForce(Vector2.up * speed); //
Иначе, если позиция платформы ниже 0, добавляет силу вверх.
        }
    }
}
```

Мы можем немного улучшить код:

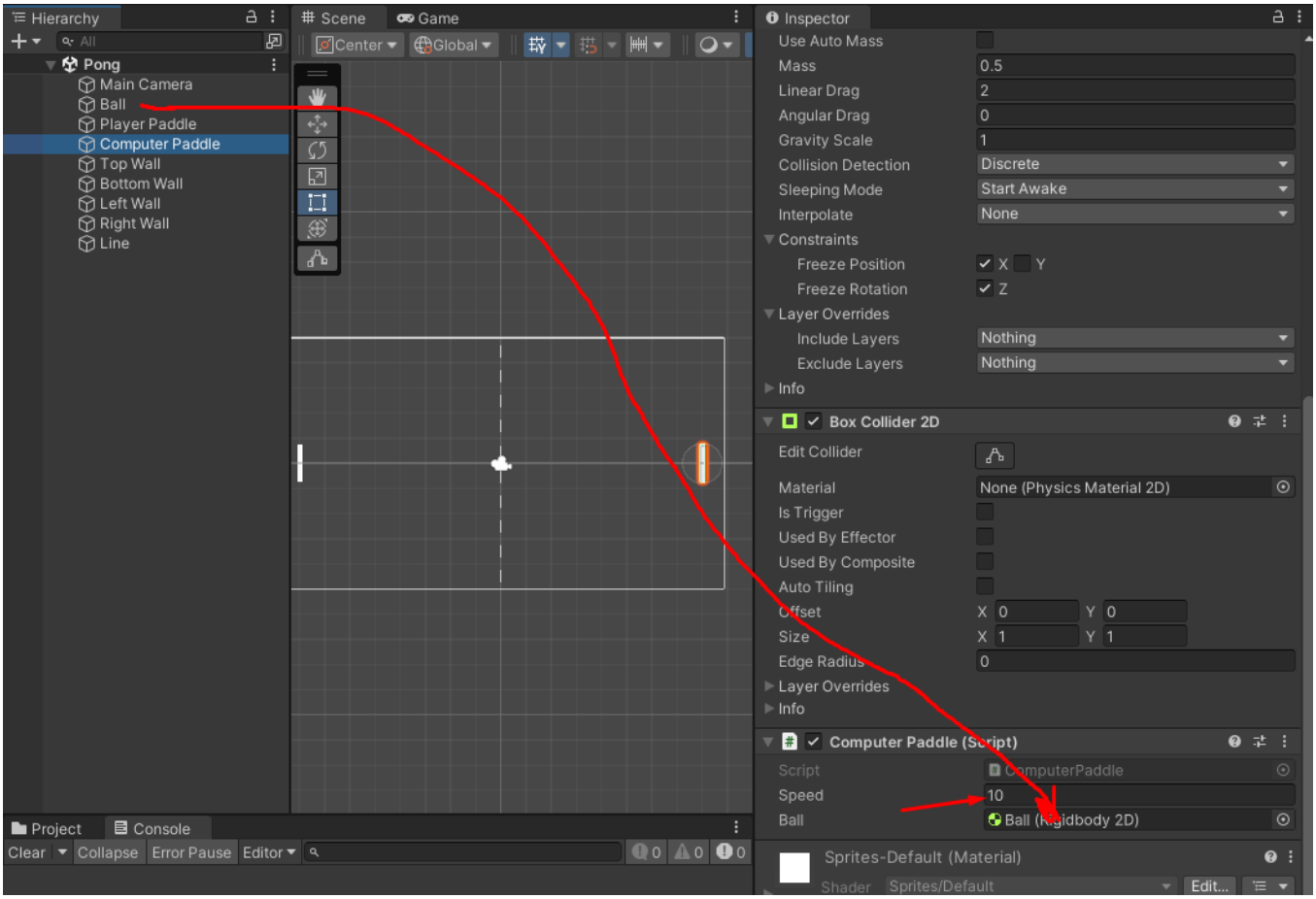
- создадим отдельный метод **MovePaddle**(). Это улучшает читаемость и позволяет избежать дублирования кода.

- упростим логику в **FixedUpdate,** таким образом, чтобы минимизировать повторение кода и сделать его более читаемым.

```
public class ComputerPaddle : Paddle
{
    public Rigidbody2D ball;

    Сообщение Unity | Ссылок: 0
    private void FixedUpdate()
    {
        if(ball.linearVelocity.x > 0f)
        {
            if (ball.position.y > transform.position.y)
                MovePaddle(Vector2.up);
            else if (ball.position.y < transform.position.y)
                MovePaddle(Vector2.down);
        }
        else
        {
            if(transform.position.y > 0f)
                MovePaddle(Vector2.down);
            else if (transform.position.y < 0f)
                MovePaddle(Vector2.up);
        }
    }
    // Метод для перемещения ракетки
    Ссылок: 4
    private void MovePaddle(Vector2 direction)
    {
        _rigidbody.AddForce(direction * speed);
    }
}
```

Добавим в поле **Ball** наш объект **Ball** и установим значение переменной на **10**.

Если мяч будет двигаться иногда по прямой, то можно поменять ему угол отскока или наклона, эта информация есть в дополнительных материалах в конце лабораторной.



18. Создадим скрипт **BoundcySurface**, который будет увеличивать скорость мяча со временем.

Добавляем в конец скрипта **Ball** новый метод:

```
public void AddForce(Vector2 force) // Этот метод добавляет силу к компоненту Rigidbody2D, который привязан к объекту (в данном случае, к мячу). Он принимает вектор force в качестве аргумента, который определяет направление и величину силы.
{
    _rigidbody.AddForce(force); //Внутри метода вызывается _rigidbody.AddForce(force), что применяет указанную силу к объекту.
}
```

Теперь пропишем новый скрипт **BouncySurface:**

```
using UnityEngine;

public class BouncySurface : MonoBehaviour
{
    public float bounceStrength; // Это публичное поле, которое
определяет силу отскока.

    private void OnCollisionEnter2D(Collision2D collision) //
Вызывается, когда объект сталкивается с другим объектом.
Параметр collision содержит информацию о столкновении.
    {
        Ball ball = collision.gameObject.GetComponent<Ball>();

        if (ball != null) // Мы проверяем, является ли объект,
с которым столкнулся мяч, экземпляром класса Ball.
        {
            Vector2 normal = collision.GetContact(0).normal; //
Если это так, получаем нормаль (вектор перпендикуляра) к
поверхности столкновения с помощью
collision.GetContact(0).normal.
            ball.AddForce(-normal * bounceStrength); //
Применяем силу отскока к мячу, умножая нормаль на
bounceStrength и меняя знак (чтобы отскок был в противоположном
направлении).
        }
    }
}
```

Пример визуализации нормали:

Представьте, что мяч ударяет по стене. Нормаль столкновения — это вектор, который указывает перпендикулярно от стены в точке соприкосновения. Если стена вертикальная и находится справа от мяча, нормаль будет вектором, указывающим влево. Применяя силу в направлении, противоположном нормали, мы заставляем мяч отскочить обратно.

Подробнее про наш скрипт:

1. Получение компонента Ball:

```
Ball ball = collision.gameObject.GetComponent<Ball>();
```

когда происходит столкновение, метод **OnCollisionEnter2D** вызывается с параметром **collision**, который содержит информацию о столкновении. Мы пытаемся получить компонент **Ball** из объекта, с которым произошло столкновение. Если объект действительно является мячом, мы продолжаем обработку.

2. Проверка на null:

```
if (ball != null)
```

эта проверка гарантирует, что мы работаем с мячом, а не с каким-то другим объектом, у которого может не быть компонента **Ball**.

3. Получение нормали столкновения:

```
Vector2 normal = collision.GetContact(0).normal;
```

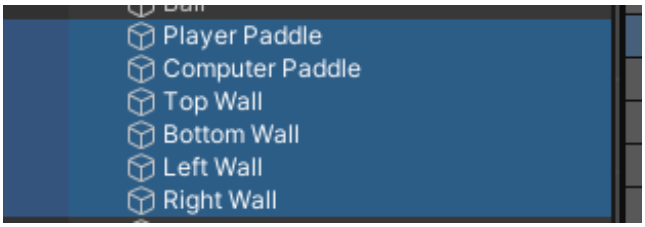
здесь мы получаем нормаль столкновения. Нормаль — это вектор, который перпендикулярен поверхности в точке столкновения. Вектор нормали указывает направление от поверхности, что позволяет определить, в каком направлении мяч должен отскочить. Например, если мяч сталкивается с вертикальной стеной, нормаль будет горизонтальной (влево или вправо в зависимости от стороны столкновения).

4. Применение силы для отскока:

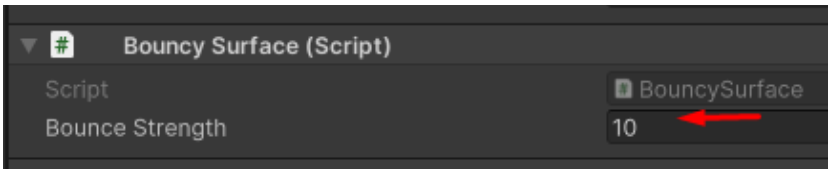
```
ball.AddForce(-normal * bouncyStrength);
```

мы применяем силу к мячу в направлении, противоположном нормали, умножая ее на **bouncyStrength**. Это создает эффект отскока, делая мяч отскакивающим от поверхности. Величина силы отскока определяется значением **bouncyStrength**.

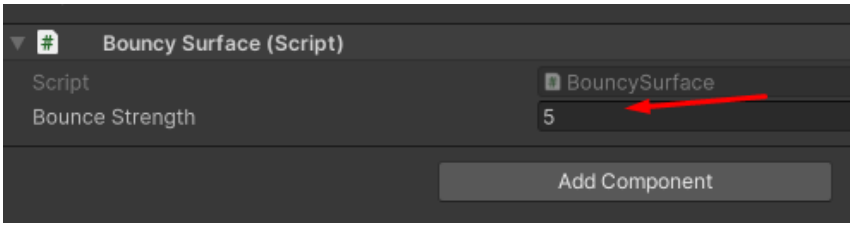
Затем добавим наш скрипт на игровые объекты:



И подредактируем силу. Для **Player Paddle** и **Computer Paddle** поставим значение, например **10**:



Для 4 стен - **Top Wall, Bottom Wall, Left Wall, Right Wall** = **5**:



В скрипте **Ball** допишем метод:

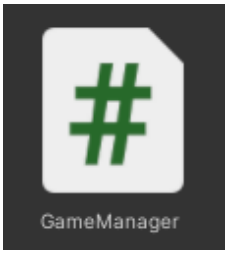
```
 public void ResetPosition() // возвращение мяча в начальную позицию
 {
     _rigidbody.position = Vector3.zero; // Устанавливает позицию мяча (_rigidbody.position) в начальную точку (в данном случае, Vector3.zero).
     _rigidbody.linearVelocity= Vector3.zero; // Обнуляет скорость мяча (_rigidbody.velocity) (останавливает его движение).
```

Также в **Ball** изменением метод **Start:**

```
Сообщение Unity | Ссылок: 0
private void Start()
{
    ResetPosition();
    AddStartingForce();
}
```

19. Теперь реализуем подсчёт очков.

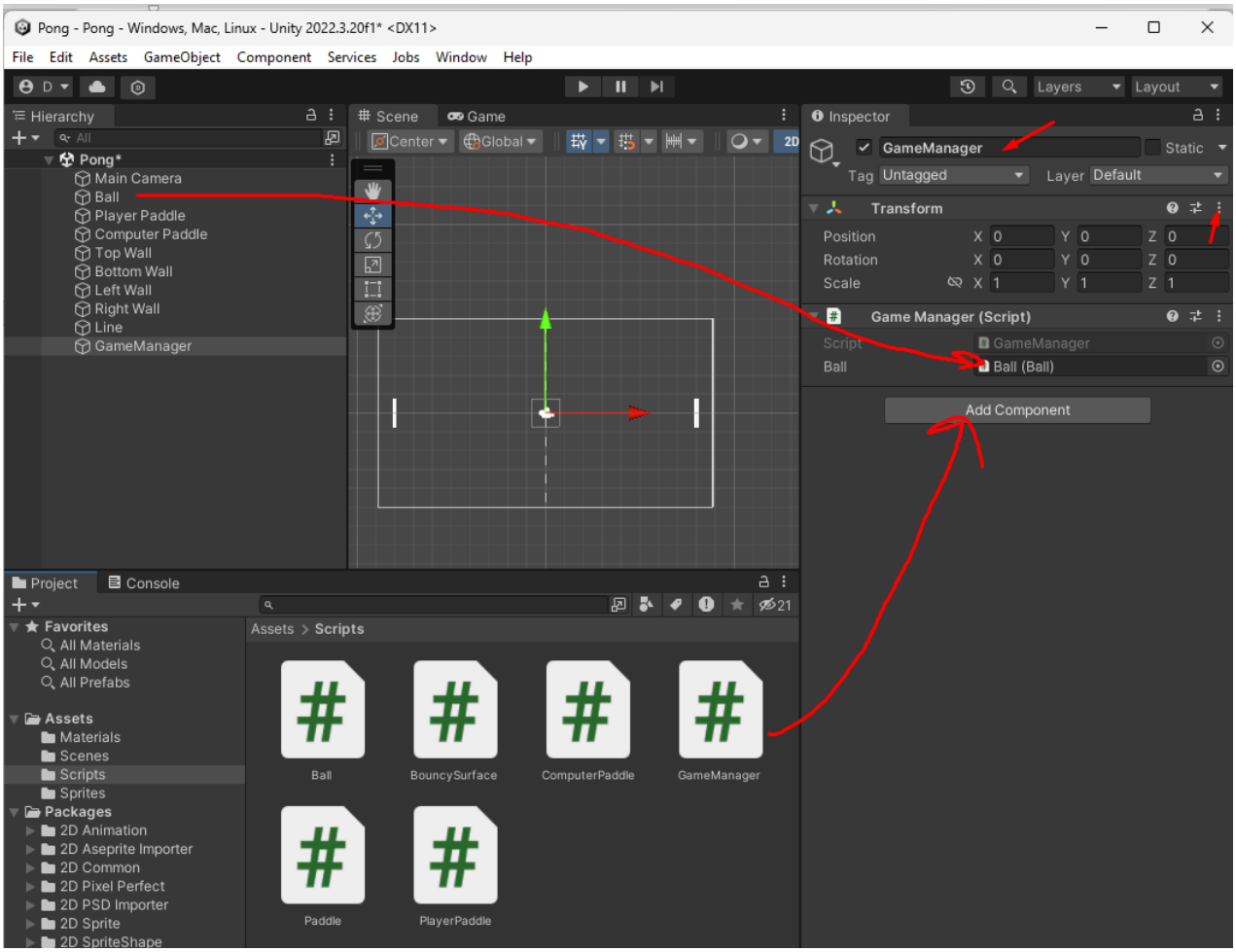
Создаём новый скрипт – **GameManager**:



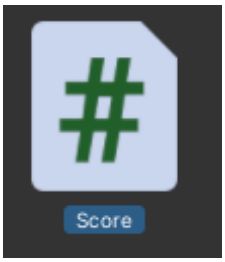
Теперь пропишем наш скрипт для **GameManager:**

```
using UnityEngine;
public class GameManager : MonoBehaviour
{
    public Ball ball; // Это публичное поле, которое хранит
ссылку на объект мяча.

    private int _playerScore; // Счет игрока.
    private int _computerScore; // Счет компьютера.

    public void PlayerScores() // Вызывается, когда игрок
забивает гол.
    {
        _playerScore++; // Увеличивает счет игрока на 1.

        ball.ResetPosition(); // Затем вызывает метод
ResetPosition() у мяча.
    }

    public void ComputerScores() // Вызывается, когда компьютер
забивает гол.
    {
        _computerScore++; // Увеличивает счет компьютера на 1.
        ball.ResetPosition(); // Затем вызывает метод
ResetPosition() у мяча.
    }
}
```

20. Создайте новый объект **GameManager**, добавьте на него скрипт, в поле **Ball** добавьте объект **Ball**, и выполните **Reset** трансформации:



21. Создаём новый скрипт **Score**:



Пишем для него код:

```
using UnityEngine;
using UnityEngine.EventSystems; // подключаем систему событий класса

public class Score : MonoBehaviour
{
    public EventTrigger.TriggerEvent scoreTrigger; // Это публичное поле, которое хранит событие, связанное со счетом.
```

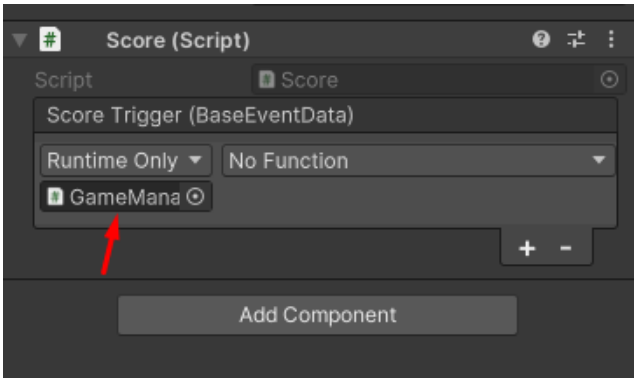
```
    private void OnCollisionEnter2D(Collision2D collision) // Вызывается, когда объект сталкивается с другим объектом.
    {
        Ball ball = collision.gameObject.GetComponent<Ball>();

        if (ball != null) // Мы проверяем, является ли объект, с которым столкнулся мяч, экземпляром класса Ball.
        {
            BaseEventData eventData = new BaseEventData(EventSystem.current); // Если это так создаем новое событие BaseEventData, которое содержит основную информацию о событии. EventSystem.current: текущая система событий в Unity.
            scoreTrigger.Invoke(eventData); // Вызываем событие scoreTrigger.Invoke(eventData).
        }
    }
}
```

Что такое событие (Event)?

- **Событие (Event)** — это механизм, который позволяет объекту уведомлять другие объекты о том, что что-то произошло. Это особенно полезно для создания гибких и масштабируемых систем взаимодействия.
- В данном контексте, событие **scoreTrigger** будет использоваться для уведомления системы о том, что мяч столкнулся с объектом, что, скорее всего, означает добавление очков.

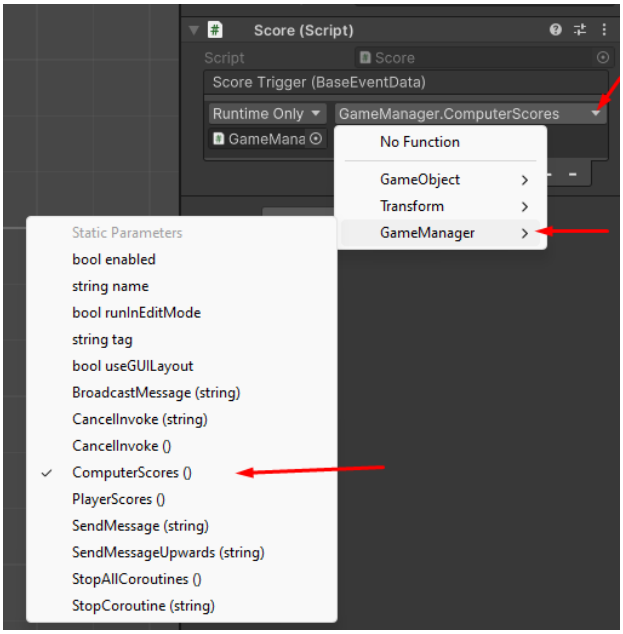
Добавляем его на правую и левую стену:

Для **двух** стен добавляем в объект – **GameManager**:

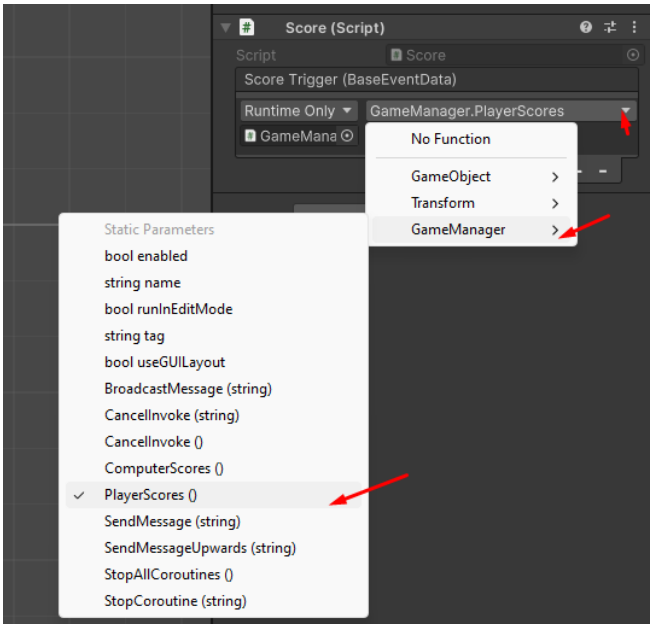


Для левой стены выбираем в пункте **Function – GameManager – ComputerScore()**

Для правой стены выбираем в пункте **Function – GameManager – PlayerScore()**



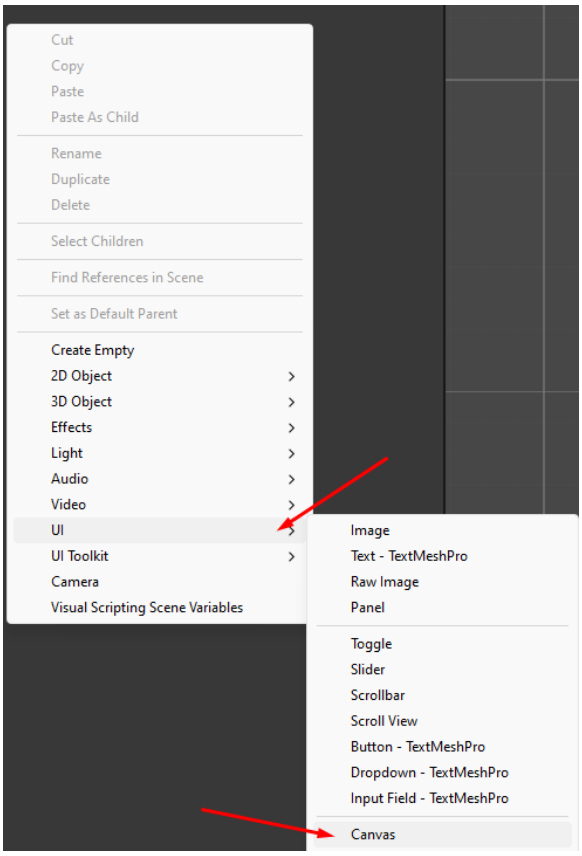
Левая стена



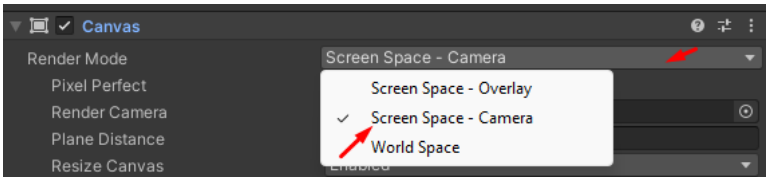
Правая стена

22. Теперь добавим отображение очков в нашу игру.

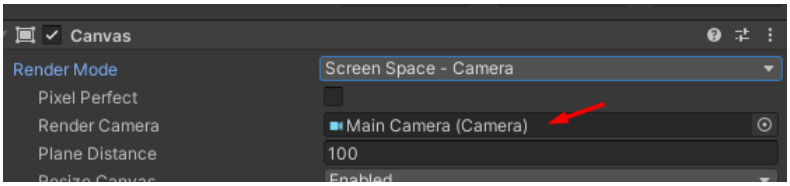
Щёлкаем правой кнопкой мыши в иерархии и добавляем **UI - Canvas**:



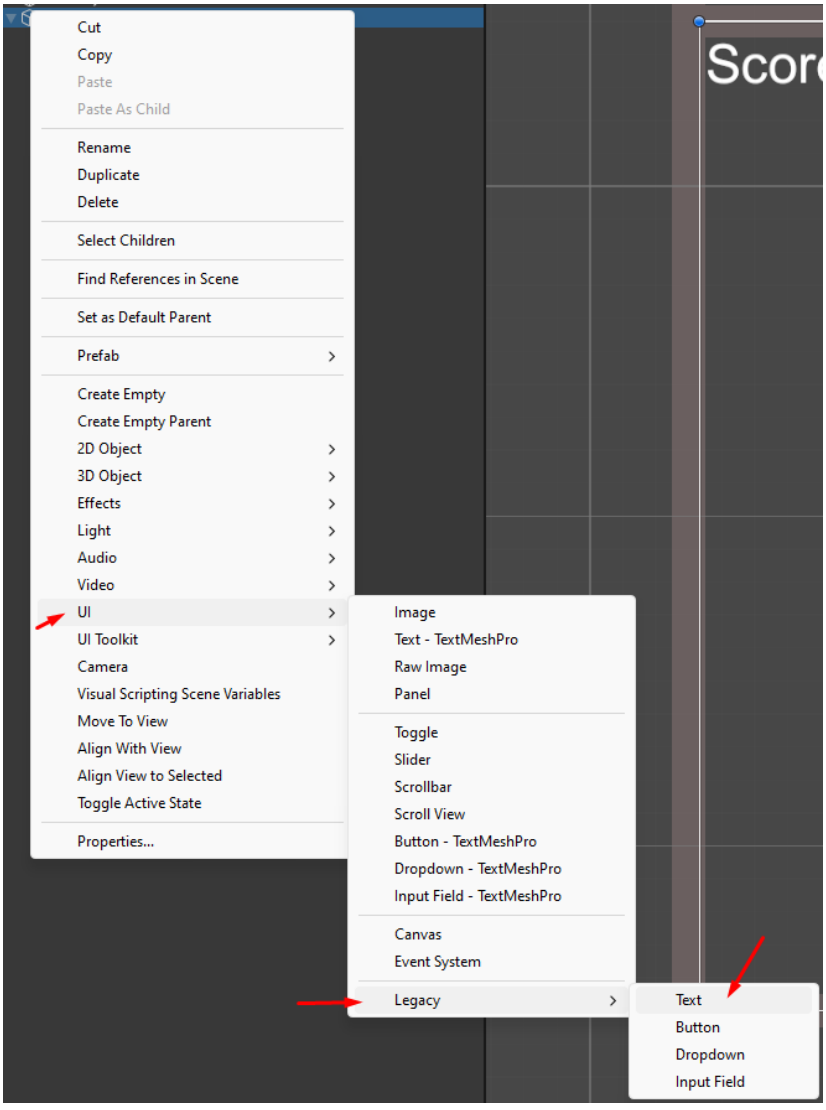
В **Inspector** выбираем в модели рендера **Screen Space-Camera:**



Переносим **Main Camera** в рендер:



Затем щёлкаем на **Canvas** и выбираем **Legacy-Text:**



Далее меняем наши параметры (приблизительно как на скриншоте ниже):

- название меняем на **PlayerScore**;

- выравниваем позицию к верхнему левому экрану;

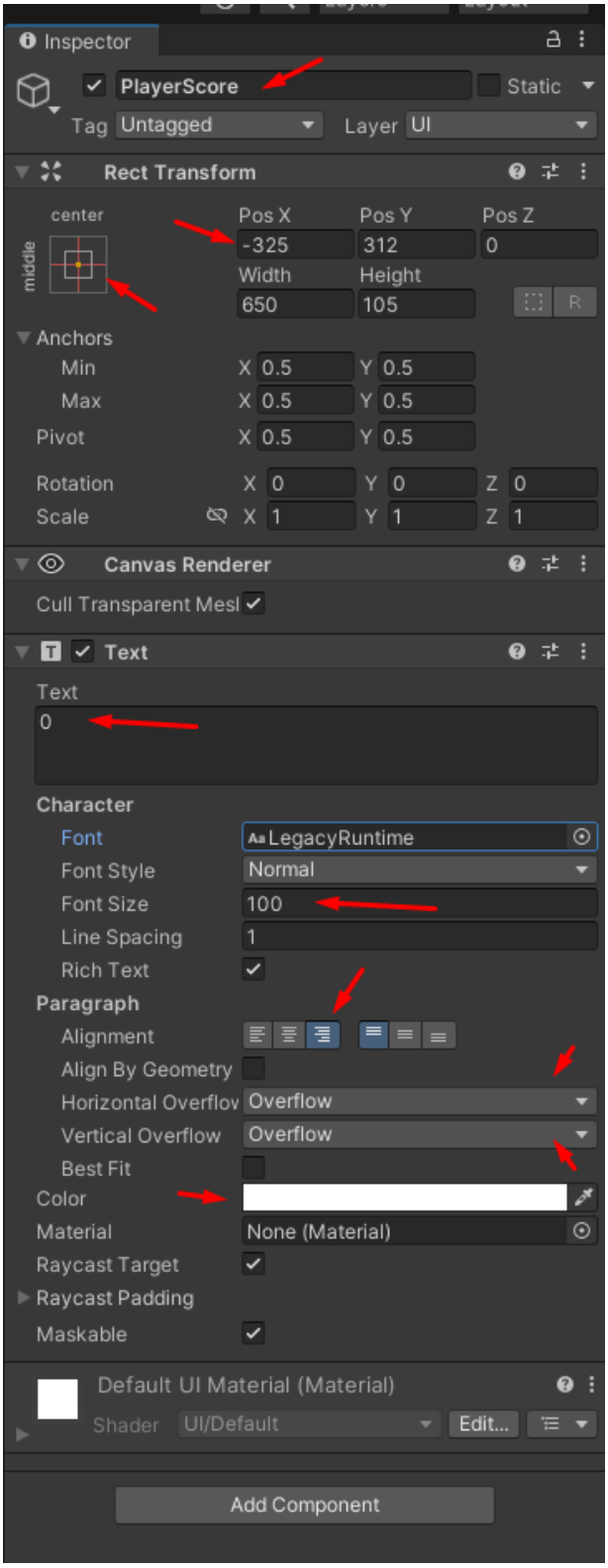
- выравнивание по правому краю;

- **Horizontal Overflow** и **Vertical Overflow** меняем на **Overflow**;

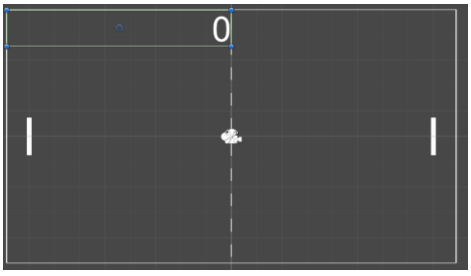
- Цвет меняем на белый;

- В тексте пишем 0;

- Размер шрифта приблизительно на 100:

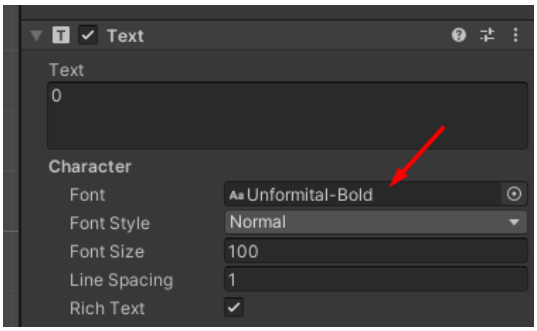


В итоге у нас будет выглядеть следующим образом:

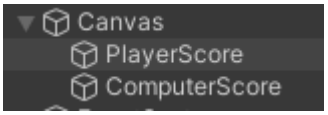


----------Необязательный шаг----------

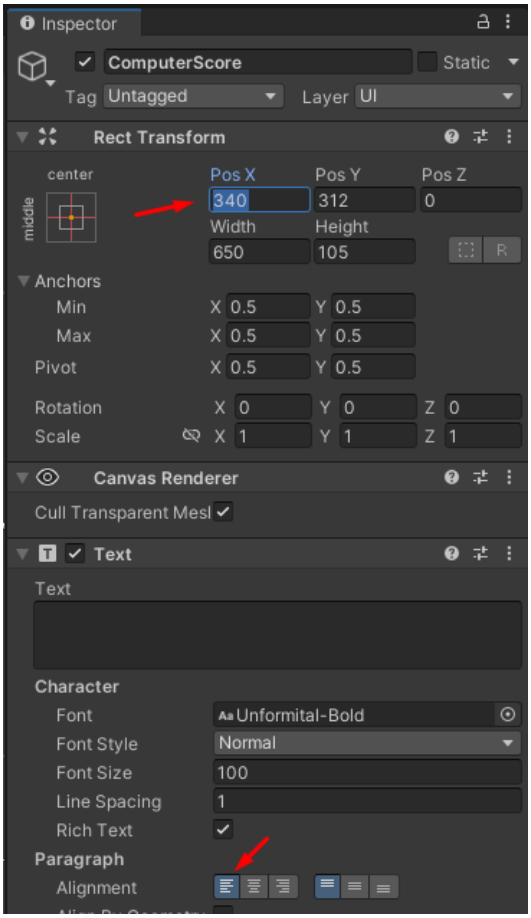
Для примера, чтобы лучше передать стиль пиксельных игр, я скачал с интернета пиксельный шрифт, создал папку Fonts и загрузил его туда, после подкрепил на мой объект. НО вы можете не повторять данную процедуру:

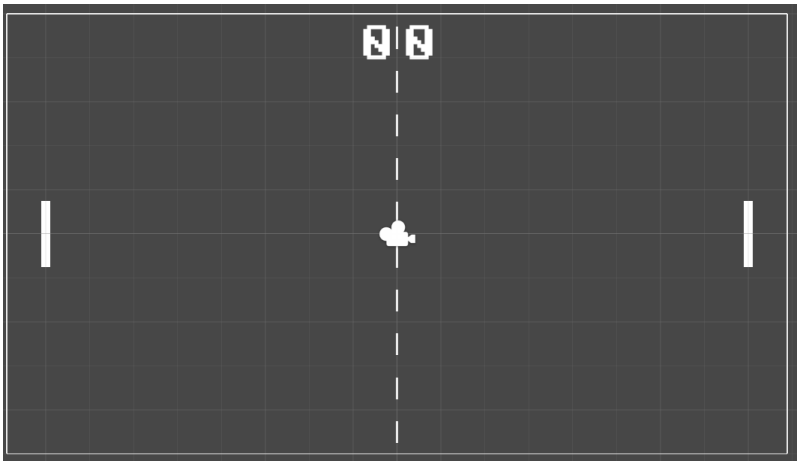


23. Скопируем наш **PlayerScore** и на его основе создадим **ComputerScore.**



Самое главное смените позицию **X** и поменяйте выравнивание **по левому краю**:





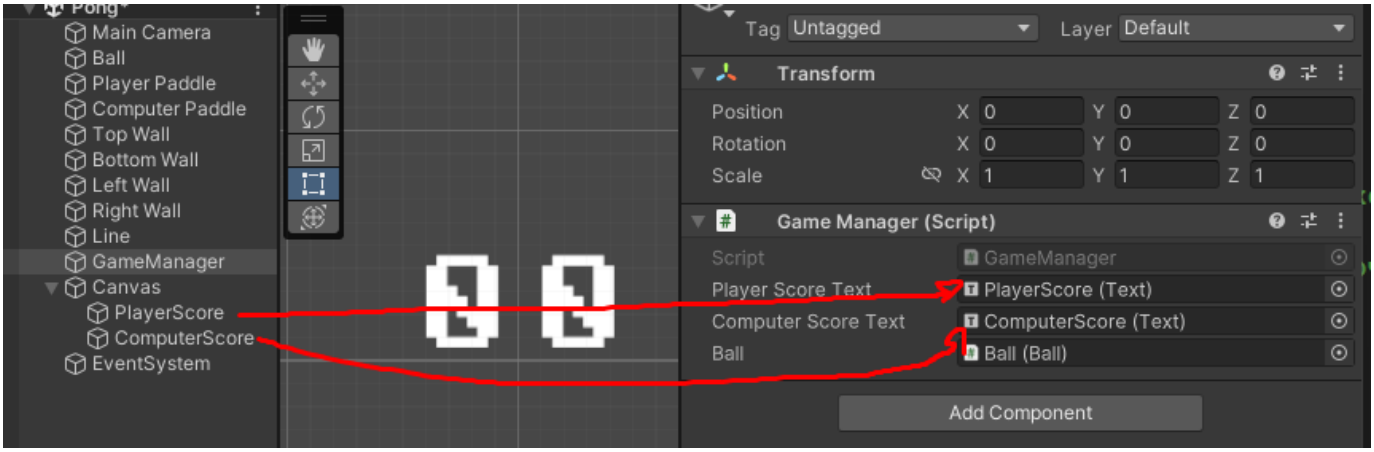
24. Далее открываем скрипт **GameManage**r и внесём новые данные:

```
using UnityEngine;
using UnityEngine.UI; // подключаем пространство имён для UI

Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 0
public class GameManager : MonoBehaviour
{
    public Text playerScoreText; // Создаём переменную подсчёта очков для игрока

    public Text computerScoreText; // Создаём переменную подсчёта очков для
      компьютера

    public Ball ball;
```

Затем вносим изменения в методы **PlayerScore** и **ComputerScore**:

```
public void PlayerScores()
{
    _playerScore++;
    playerScoreText.text = _playerScore.ToString();
    ball.AddStartingForce();
    ball.ResetPosotion();
}

Ссылок: 0
public void ComputerScores()
{
    _computerScore++;
    computerScoreText.text = _computerScore.ToString();
    ball.AddStartingForce();
    ball.ResetPosotion();
}
```

В **GameManager** добавим в переменные наши игровые объекты:



Рефакторинг

25. Скрипт **Paddle:**

```
using UnityEngine;

Скрипт Unity | Ссылок: 4
public class Paddle : MonoBehaviour
{
    public float speed = 10.0f;

    protected Rigidbody2D _rigidbody;

    Сообщение Unity | Ссылок: 0
    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
    }
    Ссылок: 0
    public void ResetPosition()
    {
        _rigidbody.position = new Vector2(_rigidbody.position.x, 0.0f);
        _rigidbody.linearVelocity = Vector2.zero;
    }
}
```

Скрипт **GameManager**:

```
Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 0
public class GameManager : MonoBehaviour
{
    public Paddle playerPaddle; // объявляем новую наследуемую переменную для
      игрока

    public Paddle computerPadlle; // объявляем новую наследуемую переменную для
      компьютера

    public Text playerScoreText;
```

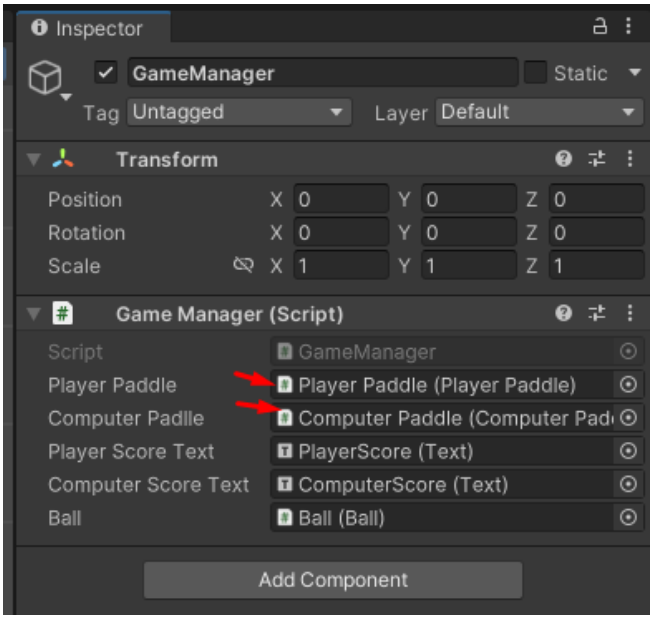
Обновим методы для очков игрока, компьютера, сброса и подсчёта очков:

```
public void PlayerScores()
{
    _playerScore++;
    playerScoreText.text = _playerScore.ToString();
    ResetRound();
}

Ссылок: 0
public void ComputerScores()
{
    _computerScore++;
    computerScoreText.text = _computerScore.ToString();
    ResetRound();
}
Ссылок: 2
public void ResetRound()
{
    playerPaddle.ResetPosition();
    computerPaddle.ResetPosition();
    ball.ResetPosition();
    ball.AddStartingForce();
}
```

Добавим метод **Start**(), который инициализирует начальные значения счета. Это полезно для сброса состояния игры при старте:

```
private void Start()
{
    _playerScore = 0;
    _computerScore = 0;
}
```

Вносим правки в **GameManager** (добавляем объекты):



26. Давайте добавим возможности выходить из игры по нажатию клавиши **ESC**. Для этого допишем в скрипте **PlayerPaddle**, методе **Update**:

```
private void Update()
{
    if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
        _direction = Vector2.up;
    else if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.Escape))
        Application.Quit();
    else
        _direction = Vector2.zero;
}
```

27. В **PlayerPaddle** вынесем управление входом игрока в отдельный метод для улучшения читабельности:

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

Ссылок: 1
private void HandleInput()
{
    if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
        _direction = Vector2.up;
    else if (Input.GetKey(KeyCode.S) || Input.GetKey
      (KeyCode.DownArrow))
        _direction = Vector2.down;
    else
        _direction = Vector2.zero;
}
```

28. В **ComputerPaddle** изменим логику на более читабельную:

```
private void FixedUpdate()
{
    if (ball.linearVelocity.x > 0.0f)
    {
        MovePaddle(ball.position.y > transform.position.y ?
Vector2.up : Vector2.down);
    }
    else
    {
        MovePaddle(transform.position.y > 0.0f ? Vector2.down :
Vector2.up);
    }
}
```

29. В **Score** улучшим читабельность добавив метод для проверки и вызова события:

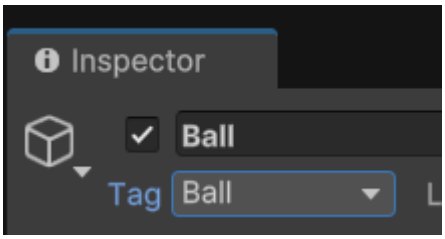
```
private void OnCollisionEnter2D(Collision2D collision)
{
    Ball ball = collision.gameObject.GetComponent<Ball>();

    if( ball != null)
    {
        TriggerScoreEvent();
    }
}
Ссылок: 1
private void TriggerScoreEvent()
{
    BaseEventData eventData = new BaseEventData(EventSystem.current);
    scoreTrigger.Invoke(eventData);
}
```

30. Аналогично сделаем в скрипте **BoundcySurface:**

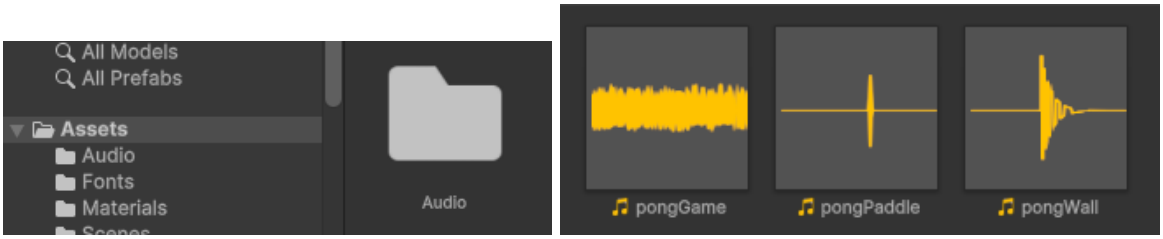
```
private void OnCollisionEnter2D(Collision2D collision)
{
    Ball ball = collision.gameObject.GetComponent<Ball>();

    if(ball != null)
    {
        BounceBall(collision, ball);
    }
}
Ссылок: 1
private void BounceBall(Collision2D collision, Ball ball)
{
    Vector2 normal = collision.GetContact(0).normal;
    ball.AddForce(-normal * bouncyStrength);
}
```

31. Добавим фоновый звук, и звук столкновения с ракеткой и с левой и правой стеной. Вначале скачиваем нужные нам звуки.

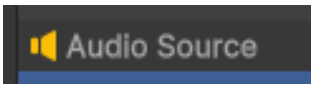
Сперва добавляем для мяча тег **Ball**:



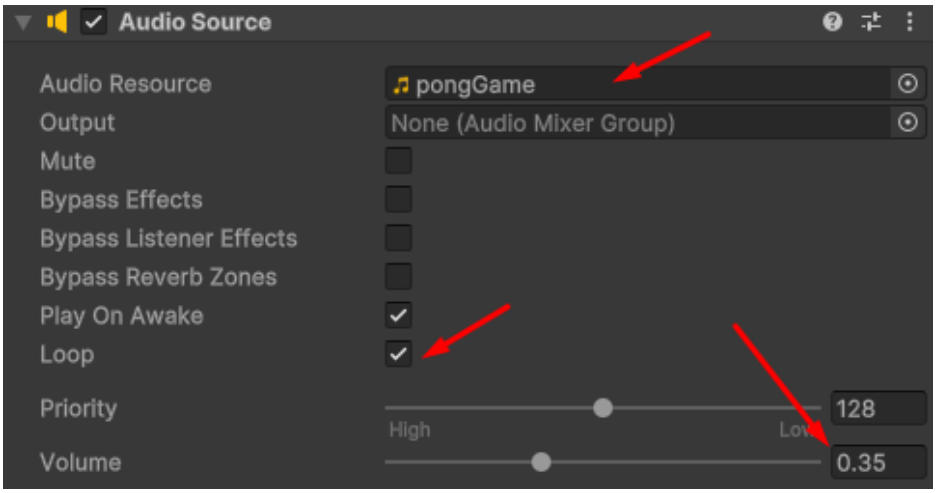
Затем создаём в ассетах папку **Audio**, переносим, заранее переименованные звуки внутрь папки:



Добавляем для камеры компонент **Audio Source:**

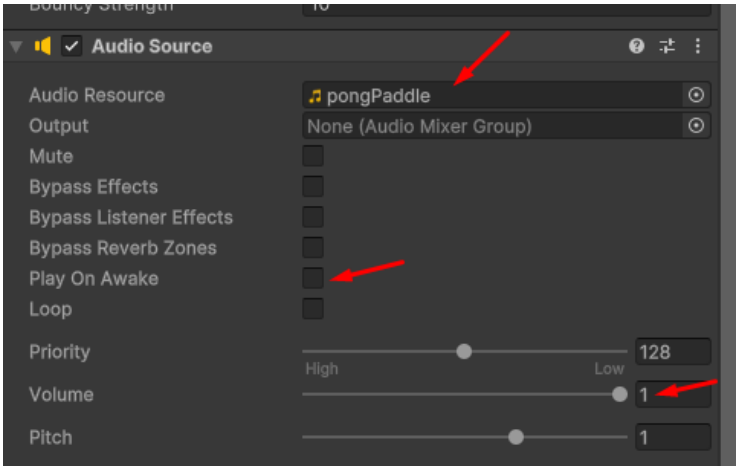


Перетащите звук в источник, можете настроить громкость, и не забудьте поставить галочку на **Loop**, чтобы зациклить воспроизведение музыки:

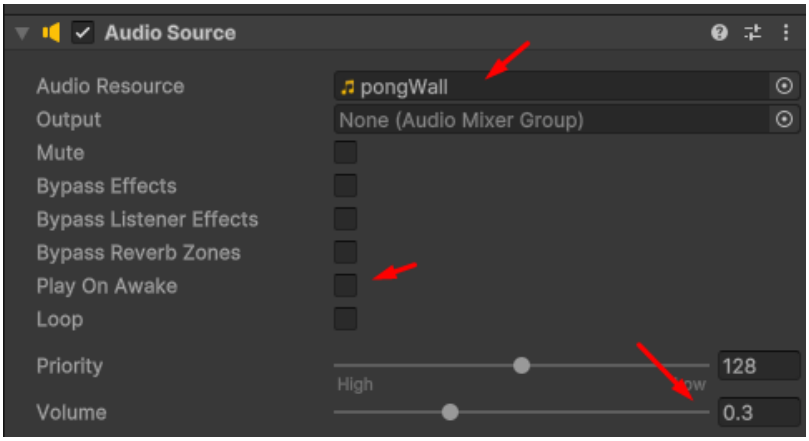


Далее добавляем компонент **Audio Source** для объектов **PlayerPaddle, ComputePaddle, TopWall, BottomWall, LeftWall, RightWall**

Для **PlayerPaddle** и **ComputePaddle** добавляем в источник наш звук, убираем галочку с пункта **Play on Awake**, чтобы звук не воспроизводился при старте игры. Также можете подкорректировать громкость (**Volume**):



Аналогично делаем для **стен**, выбрав другой источник звука:



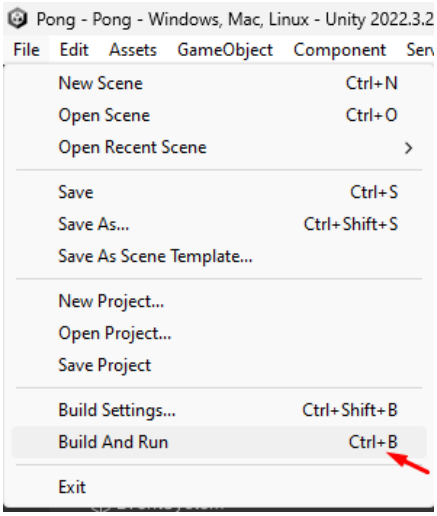
Теперь обновим скрипт для **Paddle:**

```
public class Paddle : MonoBehaviour
{
    public float speed = 10.0f;

    protected Rigidbody2D _rigidbody;

    private AudioSource _audioSource; // Добавляем AudioSource

    Сообщение Unity | Ссылок: 0
    private void Awake()
    {
        _rigidbody = GetComponent<Rigidbody2D>();
        _audioSource = GetComponent<AudioSource>(); // Инициализируем
          AudioSource
    }
    Ссылок: 2
    public void ResetPosition()
    {
        _rigidbody.position = new Vector2(_rigidbody.position.x, 0.0f);
        _rigidbody.linearVelocity = Vector2.zero;
    }
    //Добавляем метод для воспроизведения музыки, при столкновении мяча с
      ракеткой
    Сообщение Unity | Ссылок: 0
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Ball"))
        {
            _audioSource.Play(); // Воспроизводим звук при столкновении с
              мячом } }
        }
    }
}
```

И скрипт для **BoundcySurface:**

```
public class BoundcySurface : MonoBehaviour
{
    public float bouncyStrength;
    private AudioSource _audioSource; // Добавляем AudioSource

    Сообщение Unity | Ссылок: 0
    private void Awake()
    {
        _audioSource = GetComponent<AudioSource>(); // Инициализируем
          AudioSource
    }

    Сообщение Unity | Ссылок: 0
    private void OnCollisionEnter2D(Collision2D collision)
    {
        Ball ball = collision.gameObject.GetComponent<Ball>();

        if (ball != null)
        {
            BounceBall(collision, ball);
            _audioSource.Play(); // Воспроизводим звук при столкновении с
              мячом
        }
    }
    Ссылок: 1
    private void BounceBall(Collision2D collision, Ball ball)
    {
        Vector2 normal = collision.GetContact(0).normal;
        ball.AddForce(-normal * bouncyStrength);
    }
}
```

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

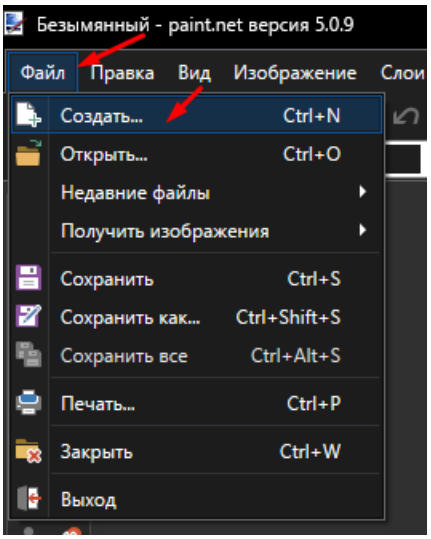


Выбираете любую папку куда хотите сохранить игру, или создаёте новую папку. После можете запустить игру через **.exe**

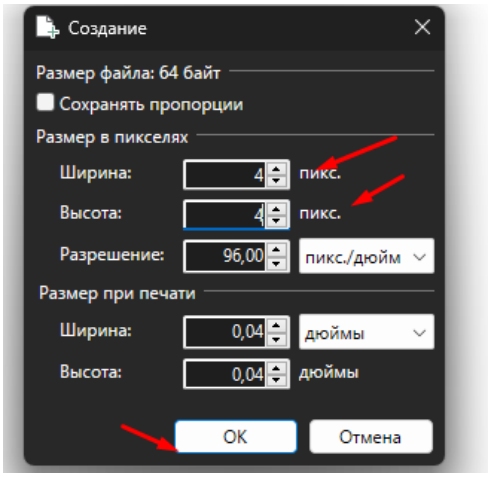
Дополнительная информация:

1. Как сделать пунктирную текстуру на примере paint.net.

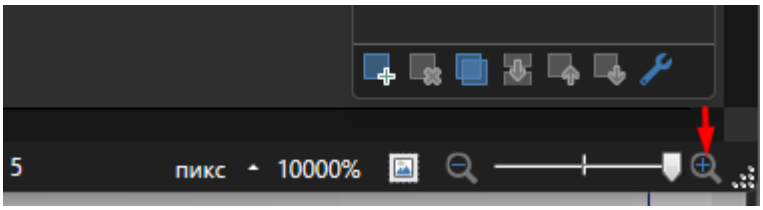
Открываем программу, нажимаем **файл** – **создать**:



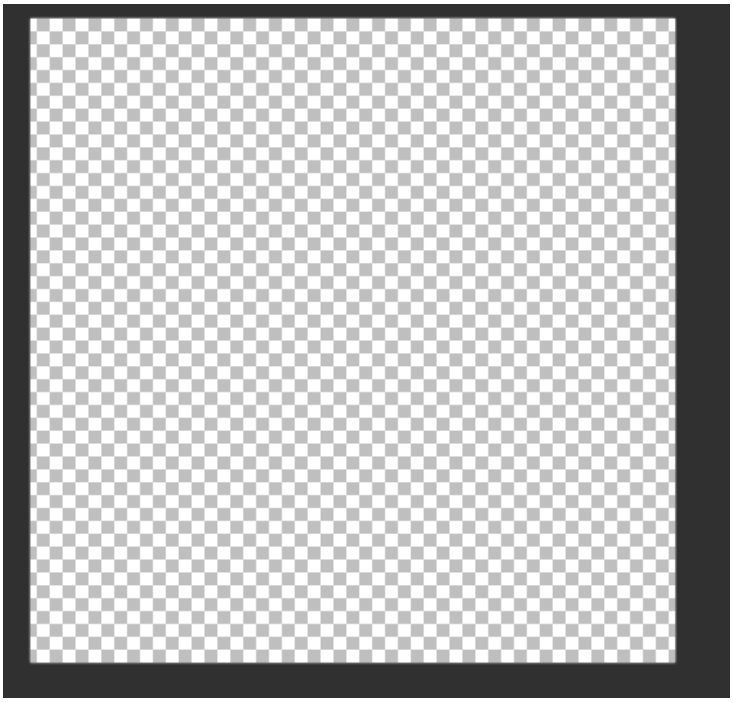
Ширину и высоту меняем на 4 пикселя:



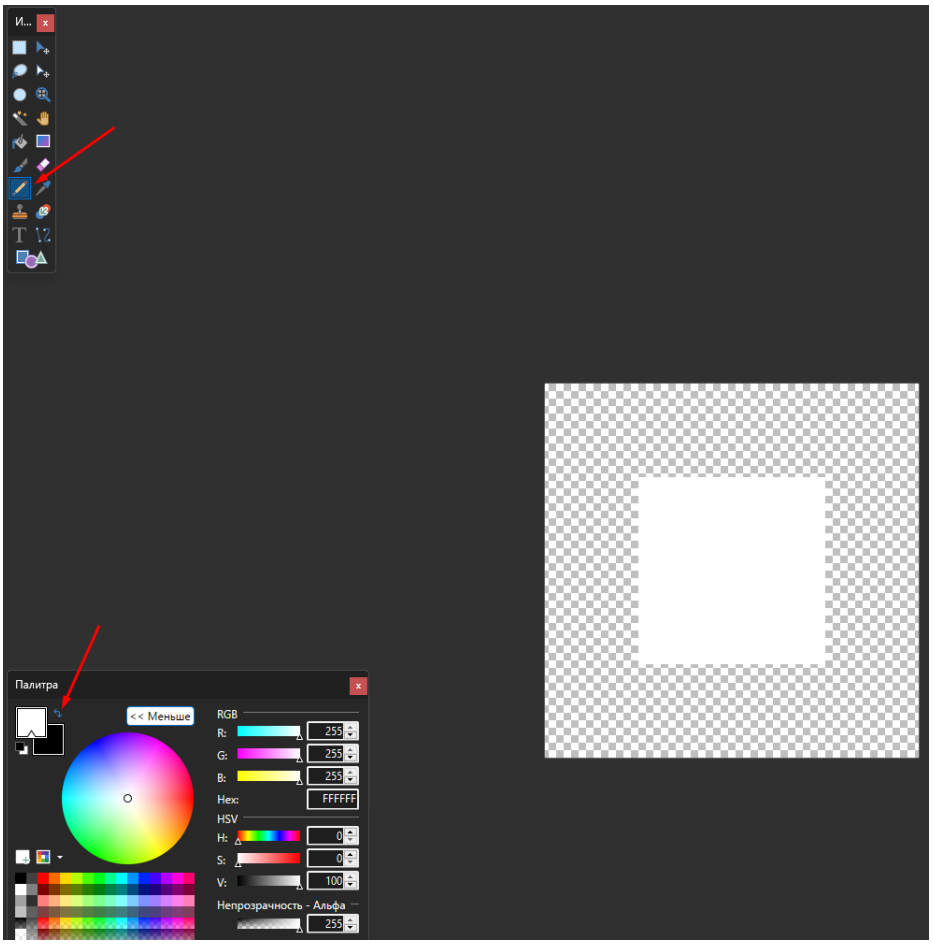
Увеличиваем максимально масштаб (**Ctrl+колесико мыши**):



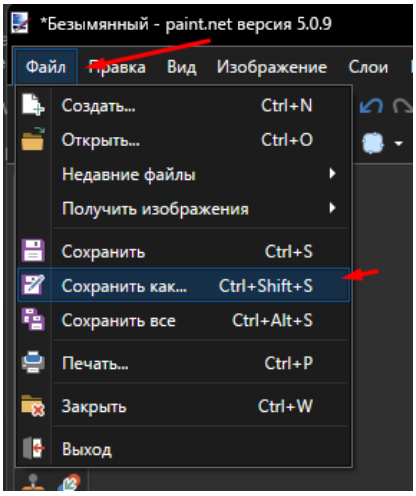
Нажимаем **Ctrl+A** – **Delete**, чтобы удалить белый фон:



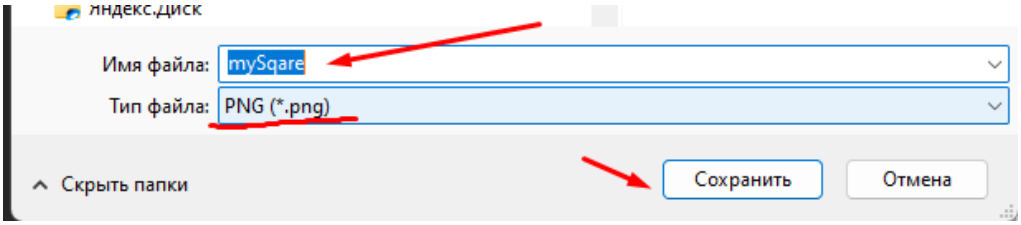
Выбираем карандаш, меняем цвет на белый и в центре рисуем квадрат:



Затем **Файл-Сохранить как:**



Выберите место куда нужно сохранить, введите название - **mySqare**, при этом убедитесь, что формат **PNG,** и нажмите сохранить:



2. Дополнительные способы реализация движения игрока.

4 способ: Перемещение через Transform.Translate

Управление положением объекта напрямую, без использования физического движка.

```
transform.Translate(_direction * this.speed * Time.del-
taTime);
```

Движок полностью игнорируется, данный метод подходит для простых игр, где физика не играет важной роли.

Плюсы:

- Простота реализации.
- Полный контроль над положением объекта.

Минусы:

- Объект не будет взаимодействовать с физическими объектами (например, столкновения).

5 способ: Использование Rigidbody.MovePosition

Управление положением объекта через физический движок, но с расчетом новых координат вручную. Позиция обновляется с учетом физического двигателя, но вы задаете ее напрямую.

```
        Vector2 newPosition = _rigidbody.position + _direc-
tion * this.speed * Time.fixedDeltaTime;
        _rigidbody.MovePosition(newPosition);
```

Плюсы:

- Учитывает физический движок.
- Подходит для точного контроля над перемещением.

Минусы:

- Может быть сложнее в настройке для сложных физических взаимодействий.

6 способ: Использование Rigidbody.MovePosition

Использование математических расчетов для обновления позиции. Аналогично **Transform.Translate**, но более гибко в плане расчетов.

```
transform.position += (Vector3)_direction * this.speed * Time.deltaTime;
```

Плюсы:

- Максимальная простота и контроль.

Минусы:

- Нет физического взаимодействия.

3. Дополнительная информация про углы мяча и исправление «бага» движения по прямой

Может возникнуть проблема, когда мяч ударяется о ракетку под небольшим углом, нормаль поверхности может направить его движение почти горизонтально. В результате траектория мяча становится прямой.

Например, когда угол станет равен 90° или -9°0, он станет двигаться вверх и вниз. Когда угол будет становиться меньше, чем -15° и 15°, он будет двигаться влево/вправо и немного вверх.

Чтобы нам увидеть текущий угол направления мяча, в скрипте **Ball** прописать метод:

```
private void OnCollisionEnter2D(Collision2D collision)
{
    float angle = Mathf.Atan2(_rigidbody.linearVelocity.y,
_rigibody.linearVelocity.x) * Mathf.Rad2Deg;
    Debug.Log("Ball Angle at Collision: " + angle + "°");
}
```

Для написания символа градуса в Windows следует одной рукой нажать клавишу Alt и, удерживая её, другой рукой ввести на клавиатуре цифры 0 1 7 6.

Объяснение кода:

1. **Mathf.Atan2(y, x)**:
   - Это метод, который вычисляет угол в радианах для вектора, задаваемого координатами (x, y). Он принимает два аргумента:
     - **y**: Координата вертикальной скорости.
     - **x**: Координата горизонтальной скорости.
   - Возвращаемое значение — угол между осью X и вектором скорости.
2. *** Mathf.Rad2Deg**:
   - Переводит угол из радиана в градусы, поскольку Mathf.Atan2 возвращает значение в радианах, а градусы более понятны для восприятия.
3. **_rigidbody.velocity**:
   - Вектор скорости объекта, предоставляемый Rigidbody2D.
   - Его компоненты x и y используются для вычисления угла.
4. **Debug.Log**:
   - Выводит угол в консоль Unity.

Для демонстрации как ведёт себя мяч, при определённом угле, можно переписать метод **AddStartingForce:**

```
public void AddStartingForce(float angleInDegrees)
{
    // Конвертируем угол из градусов в радианы
    float angleInRadians = angleInDegrees * Mathf.Deg2Rad;
    // Вычисляем направление на основе угла
    float x = Mathf.Cos(angleInRadians); // Горизонтальная со-
ставляющая
    float y = Mathf.Sin(angleInRadians); // Вертикальная со-
ставляющая
    // Создаём вектор направления
    Vector2 _direction = new Vector2(x, y).normalized;
    // Применяем силу к мячу
    _rigidbody.AddForce(_direction * speed);
}
```

И теперь в методе **Start** мы можем вручную задавать угол и увидеть, как движется наш мяч:

```
private void Start()
{
    ResetPosition();
    AddStartingForce(25);
}
```

Итак, возвращаем метод **AddStartingForce** к начальным значениям и в методе **Start** убираем параметры:

```
private void Start()
{
    ResetPosition();
    AddStartingForce();
}
Ссылок: 1
public void AddStartingForce()
{
    float x = Random.value < 0.5f ? -1.0f : 1.0f;
    float y = Random.value < 0.5f ? Random.Range(-1.0f, -0.5f) : Random.Range(0.5f, 1.0f);

    Vector2 _direction = new Vector2(x, y);
    _rigidbody.AddForce(_direction * speed);
}
```

Переходим в скрипт **BouncySurface.** И решим проблему, сделав так, что, когда угол становится меньше, например 20°, будем поднимать его до 30°.

Для этого можно увеличить силу при столкновении

Второй способ - **установить минимальный угол отскока** в методе **OnCollisionEnter2D:**

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Ball ball = collision.gameObject.GetComponent<Ball>();
    if (ball != null)
    {
        Vector2 normal = collision.GetContact(0).normal;
        Vector2 newDirection = -normal * bouncyStrength;
        if (Mathf.Abs(Vector2.Angle(newDirection, Vector2.right)) < 20f)
        {
            newDirection = Quaternion.Euler(0, 0, 30f) * newDirection;
        }
        ball.AddForce(newDirection);
    }
}
```

1. Mathf.Abs

- **Описание**: Возвращает абсолютное значение числа (отрицательные значения превращает в положительные).

- **Почему используется**: Углы могут быть как положительными, так и отрицательными (например, -10° и 10°), но нас интересует их абсолютное значение.
    - Например, Mathf.Abs(-10f) вернёт 10f.

2. Vector2.Angle

- **Описание**: Вычисляет угол (в градусах) между двумя векторами. Результат всегда будет в диапазоне от 0° до 180°.
- **Синтаксис**:

  float angle = Vector2.Angle(Vector2 v1, Vector2 v2);
    - **v1 и v2** — два вектора, между которыми вычисляется угол.
    - Возвращает значение угла между векторами (например, угол между (1, 0) и (0, 1) равен 90°).
- **В данном коде**:

  Vector2.Angle(newDirection, Vector2.right)
    - **newDirection** — текущее направление мяча.
    - **Vector2.right** — это встроенный вектор, равный (1, 0), указывающий вправо вдоль оси X.
    - Этот вызов определяет угол между направлением мяча и горизонтальной осью.

3. Quaternion.Euler

- **Описание**: Создаёт поворот (кватернион), основанный на углах вращения по осям X, Y и Z.
- **Синтаксис**:

  Quaternion rotation = Quaternion.Euler(float x, float y, float z);
    - **x, y, z** — углы вращения в градусах по соответствующим осям.
    - Созданный Quaternion можно использовать для поворота векторов или объектов в трёхмерном пространстве.
- **В данном коде**:

  Quaternion.Euler(0, 0, 30f)
    - Поворот создаётся только по оси Z (2D-пространство), на угол 30°.
    - Это означает, что мы вращаем вектор **newDirection** на 30° против часовой стрелки.

4. Поворот вектора через умножение на Quaternion

- **Что происходит?**

  Умножение вектора на кватернион поворачивает вектор на заданный угол.

  newDirection = Quaternion.Euler(0, 0, 30f) * newDirection;

  - Поворачивает вектор newDirection на 15° против часовой стрелки относительно начала координат.
  - Если бы угол был отрицательным, поворот был бы по часовой стрелке.

Когда вы хотите повернуть вектор направления (или объект), вы вращаете его **вокруг оси Z**, чтобы его ориентация изменялась в плоскости X-Y.

Как работает Quaternion.Euler(0, 0, 30f) в 2D?

- **Первый параметр (X)**: Определяет вращение вокруг оси X (не используется в 2D, так как движение по Y не зависит от этого вращения).
- **Второй параметр (Y)**: Определяет вращение вокруг оси Y (аналогично, не применяется в 2D).
- **Третий параметр (Z)**: Определяет вращение вокруг оси Z, что влияет на то, как объект или вектор "разворачивается" в плоскости X-Y.

В данном случае, угол **30f** вокруг оси **Z** означает, что направление вектора будет повернуто **на 30 градусов против часовой стрелки в плоскости X-Y**.

Также решим проблему **с движением "вверх-вниз".** Она возникает, когда угол отскока становится слишком близким к 90°, и мяч движется практически вертикально. В результате мяч может "застрять" в движении вверх и вниз, особенно если он сталкивается с горизонтальными поверхностями.

Внесём изменения в скрипт **BouncySurface:**

```
// Угол между направлением и осью X
float angle = Mathf.Abs(Vector2.Angle(newDirection, Vec-
tor2.right));
// Если угол слишком мал (почти горизонтально), корректируем
if (angle < 20f)
{
    newDirection = Quaternion.Euler(0, 0, 30f) * newDirection;
}
// Если угол слишком близок к 90° (почти вертикально), корректируем
else if (angle > 80f)
{
    float correctionAngle = angle > 90f ? -30f : 30f;
    newDirection = Quaternion.Euler(0, 0, correctionAngle) * newD-
irection;
}
ball.AddForce(newDirection);
```

4. Проблема замедления мяча

Чтобы гарантировать, что мяч всегда движется с минимальной скоростью, лучше напрямую проверять и корректировать его скорость на основе вектора текущей скорости.

В скрипте **Ball** выведем на консоль отображение текущей скорости:

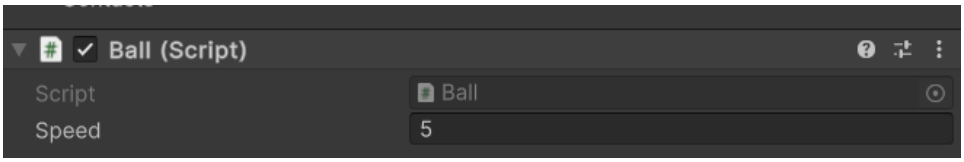
```
private void FixedUpdate()
{
    Debug.Log($"{_rigidbody.linearVelocity.magnitude}");

}
```

возвращает вектор скорости, а его свойство ***magnitude*** *позволяет получить числовое значение этой скорости.*

В скрипте **Ball:**

```
private void FixedUpdate()
{
    // Получаем текущую скорость (величину вектора скорости)
    float currentSpeed = _rigidbody.linearVelocity.magnitude;
    //Это длина (модуль) вектора скорости мяча, которая пред-
ставляет текущую скорость.
    Debug.Log($"{_rigidbody.linearVelocity.magnitude}");
    // Проверяем, если текущая скорость меньше минимальной
    if (currentSpeed < speed)
    {
        // Нормализуем вектор скорости и задаём новую скорость
        _rigidbody.linearVelocity = _rigidbody.linearVeloc-
ity.normalized * speed;
    }
}
```

Меняем скорость на правильную для скрипта **Ball** на 5:



5. Вылеты мяча за пределы экрана

Когда мяч становится слишком быстрым и вылетает за пределы игрового поля, связана с ограничениями физического движка Unity. Если объект движется слишком

быстро, его движение может не успевать корректно обрабатываться между кадрами симуляции, что приводит к "пролёту" через коллайдеры.

1 способ, как решить проблему. **Ограничьте максимальную скорость мяча**

В моём случае при скорости больше 51, мяч вылетает за пределы границ. Чтобы это обойти внесём правки в скрипт **Ball.** Объявим максимальное значение скорости:

```
private float maxSpeed = 51f;
```

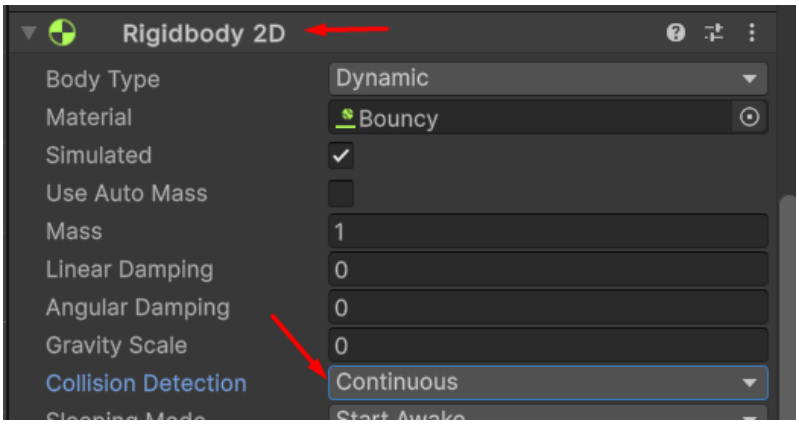
Затем в методе **FixedUpdate** внесём правки:

```
private void FixedUpdate()
{
    float currentSpeed = _rigidbody.linearVelocity.magnitude;
    Debug.Log($"{_rigidbody.linearVelocity.magnitude}");
    if (currentSpeed < speed)
    {
        _rigidbody.linearVelocity =
          _rigidbody.linearVelocity.normalized * speed;
    }
    // Ограничение скорости
    else if (currentSpeed > maxSpeed)
    {
        _rigidbody.linearVelocity =
          _rigidbody.linearVelocity.normalized * maxSpeed;
    }
}
```

2 способ. Использование режима **Continuous** для коллизий.

Для начала удалим предыдущие правки, чтобы не было лимита скорости.

В компоненте **Rigidbody2D** нашего объекта мяча **Ball,** установите **Collision Detection** в значение **Continuous:**



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.

Из документации:

Дискретный	Если вы устанавливаете **Collision Detection** на **Discrete** , GameObjects с Rigidbody 2D и Collider 2D могут перекрываться или проходить друг сквозь друга во время обновления физики, если они движутся достаточно быстро. Контакты столкновений генерируются только в новой позиции.
Непрерывный	Если **Collision Detection** установлен на **Continuous** , GameObjects с Rigidbody 2D и Collider 2D не проходят друг сквозь друга во время обновления. Вместо этого Unity вычисляет первую точку удара любого из Collider 2D и перемещает GameObject туда. Обратите внимание, что это занимает больше процессорного времени, чем **Discrete** .