Школа:		
Дата:	**ФИО учителя: Сыздыкова А.А.**	
Класс:	**Участвовали:**	**Не участволвали:**
Тема урока: Создание собственного скрипта горизонтального передвижения на языке СИ #		
Цели обучения, которые достигаются на данном уроке	Научить обучающихся созданию собственного скрипта горизонтального передвижения на языке СИ #	
Цели урока	**Все учащиеся**: • Знать виды скриптов для передвижения **Большинство учащихся**: • Распознать назначения скриптов для передвижения **Некоторые учащиеся:** • создавать мини проекты на СИ #	
Критерии оценивания	**Все учащиеся смогут**: • знают разные скрипты для передвижения **Большинство учащихся смогут**: • Распознают назначения скриптов для для передвижения объектов **Некоторые учащиеся смогут:** • создают мини проекты на СИ #	
Воспитание ценностей	Воспитание уважения к мнению друг друга, ответственность, коммуникативные способности, критическое мышление	
Предварительные знания	Создание собственного скрипта горизонтального передвижения на языке СИ #	
Межпредметные связи	Информатика, математика	

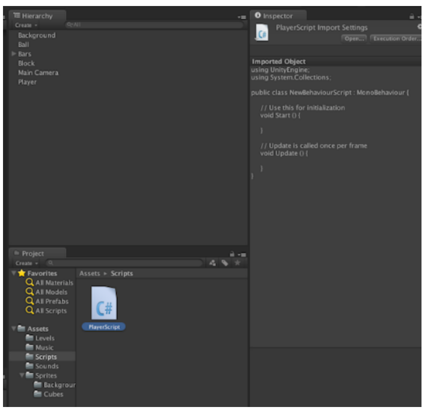
Запланированные этапы урока	**Запланированная деятельность на уроке**	**Ресурсы**
Начало урока **_5__ мин**	**Организационный момент.** Приветствие учащихся. Создание положительного эмоционального фона. Настрой на рабочий лад. Показываю тему урока и цель урока, озвучиваю критерии для цели урока **Деление на подгруппы.** Стратегия «Нумерация»	Карточки с цифры
Середина урока **_30__ мин**	**<u>Ученики работают в группе.</u>** **<u>Выполняют практическую работу №1</u>** **<u>«Передвижение платформы»</u>** **<u>Формативное оценивание:</u>** <u>Стратегия «Светофор»</u> <u>Карточка красного цвета: внимание, мне нужна помошь;</u> <u>Карточка желтого цвета: нужна небольшая помощь;</u> <u>Карточка зеленого цвета: можно двигаться дальше, все понятно.</u>	Практическая работа № 1 Карточки для стратегии светофор

	<u>Выполняют практическую работу №2</u> Ученики создают свои мини проекты на СИ # Формативное оценивание: ученики оценивают проекты с помощью стратегии «Большой палец»	
Конец урока _5__ мин	**Рефлексия:** Метод «Рефлексивные карточки» Было интересно… Я понял, что… У меня получилось… Расскажу дома, что…	«Рефлексивные стикеры»
Дифференциация – каким образом Вы планируете оказать больше поддержки? Какие задачи Вы планируете поставить перед более способными учащимися?	**Оценивание – как Вы планируете проверить уровень усвоения материала учащимися?**	**Охрана здоровья и соблюдение техники безопасности**

<u>Практическую работу №1</u>

<u>«Передвижение платформы»</u>

Итак, для создания скрипта перейдем на вкладку **Project**, найдем там одноименную папку **Scripts** и кликнем на нее правой кнопкой мыши. Выберем **Create -> C# Script**. Появится новый файл с названием NewBehaviourScript. Переименуйте его в PlayerScript для удобства. На вкладке **Inspector** вы можете видеть содержимое скрипта.



Двойным кликом откройте скрипт. Запуститься среда разработки MonoDevelop, которую вы впоследствии можете изменить на любой удобный для вас редактор. Вот то, что вы увидите:

```
using UnityEngine;
using System.Collections;
public class NewBehaviourScript : MonoBehaviour {
    // используйте этот метод для инициализации
  void Start () {
   }
  // Update вызывается при отрисовке каждого кадра игры
  void Update () {


  }
}
```

Все сценарии на Unity имеют по умолчанию два метода:

- Start(): используется для инициализации переменных или параметров, необходимых нам в коде.

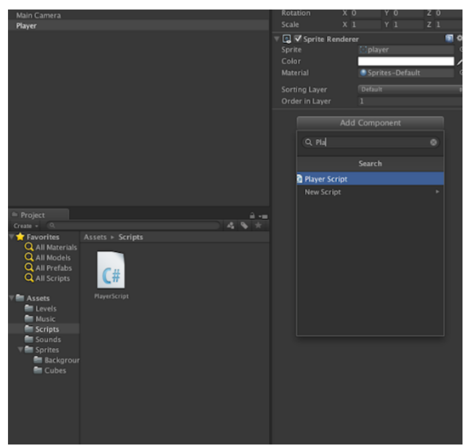
- Update(): вызывается каждый кадр игры, необходим для обновления состояния игры.

Для того, чтобы сдвинуть платформу, нам потребуется два вида информации: позиция и скорость. Таким образом, необходимо создать две переменные для сохранения этой информации:

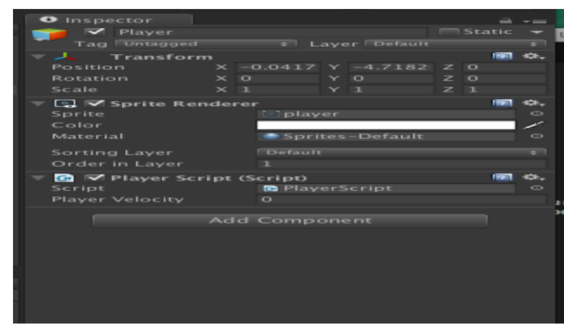
```
public float playerVelocity;
private Vector3 playerPosition;
```

Обратите внимание, что одна переменная объявлена публично, а вторая — приватно. Для чего это делается? Дело в том, что Unity позволяет редактировать значения публичных переменных не переходя в редактор MonoDevelop, без необходимости изменения кода. Эта возможность очень полезна в тех случаях. когда необходимо «на лету» корректировать какое-то значение. Скорость платформы — одно из таких значений, и именно поэтому мы объявили его публично.

Сохраните сценарий в редакторе MonoDevelop и перейдите в редактор Unity. Теперь у нас есть сценарий и нам нужно присвоить его какому то объекту, в нашем случае — платформе. Выберите нашу платформу во вкладке **Hierarchy** и в окне **Inspector**добавьте компонент, кликнув на кнопку **Add Component**.



Добавление нашего скрипта в компонент можно сделать и по-другому. Перетащите наш сценарий в область кнопки **Add Component**. Во вкладке **Inspector** вы должны увидеть что-то подобное:



Обратите внимание, что в компоненте скрипта появилось поле **Player Velocity**, которое можно тут же изменить. Это получилось возможным благодаря публичному объявлению переменной. Установите параметр в значение 0.3 и перейдите в редактор MonoDevelop.

Теперь нам надо узнать позицию платформы: playerPosition. Для того, чтобы инициализировать переменную, следует обратиться к объекту сценария в методе Start():

```
// используйте этот метод для инициализации
void Start () {
        // получим начальную позицию платформы
        playerPosition = gameObject.transform.position;
}
```

Отлично, мы определили начальную позицию платформы, и теперь можно ее двигать. Так как нам надо, чтобы платформа перемещалась только по оси X, то мы сможем использовать метод GetAxis класса Input. Этой функции мы передадим строку Horizontal, и она вернет нам 1, если была нажата клавиша «вправо», и -1 — «влево». Умножив полученное число на скорость и прибавив эту величину к текущей позиции игрока, мы и получим движение.

Также добавим проверку на выход из приложения по нажатию на клавишу Esc.

Вот то, что у нас должно получиться в итоге:

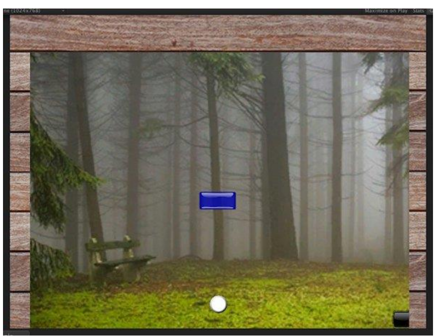
```
using UnityEngine;
using System.Collections;


public class PlayerScript : MonoBehaviour {


  public float playerVelocity;
```

```
    private Vector3 playerPosition;

    // используйте этот метод для инициализации
    void Start () {
        // получим начальную позицию платформы
        playerPosition = gameObject.transform.position;
    }

    // Update вызывается при отрисовке каждого кадра игры
    void Update () {
        // горизонтальное движение
        playerPosition.x += Input.GetAxis ("Horizontal") * playerVelocity;

        // выход из игры
        if (Input.GetKeyDown(KeyCode.Escape)){
            Application.Quit();
        }

        // обновим позицию платформы
        transform.position = playerPosition;
    }
}
```

Сохраните скрипт и вернитесь в редактор Unity. Нажмите кнопку **Play** и попробуйте передвинуть платформу при помощи кнопок «влево» и «вправо».

Определение игровой области

Вы, скорее всего, заметили, что платформа может двигаться и за пределами игрового поля. Дело в том, что у нас нет никаких проверок на выход за пределы каких-то границ.

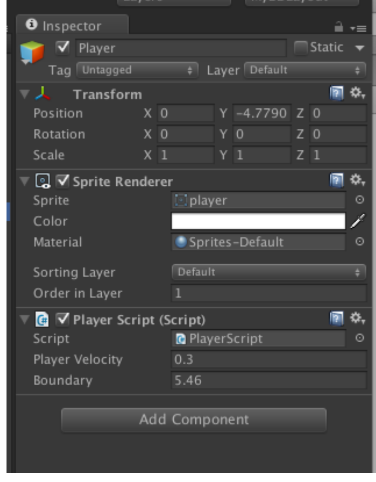


Давайте добавим в наш существующий скрипт еще одну публичную переменную и назовем его boundary.

Эта переменная будет хранить максимальную координату платформы по оси X. Так как мы собираемся строить уровни в симметричной форме вокруг точки с координатами (0, 0, 0), то абсолютное значение переменной boundary будет одинаковым и для положительной части оси X, и для отрицательной.

А теперь добавим пару условий. Поступим достаточно просто: если вычисленная нами позиция будет больше boundary или меньше -boundary, то мы просто зададим новую позицию по оси X, равную значению переменной boundary. Таким образом, мы гарантируем, что платформа не уедет за пределы наших границ и никогда не покинет игровую зону. Вот код:

```
using UnityEngine;
using System.Collections;

public class PlayerScript : MonoBehaviour {

    public float playerVelocity;
```




```
private Vector3 playerPosition;

// используйте этот метод для инициализации
void Start () {
    // получим начальную позицию платформы
    playerPosition = gameObject.transform.position;
}

// Update вызывается при отрисовке каждого кадра игры
    void Update () {
        // горизонтальное движение
        playerPosition.x += Input.GetAxis ("Horizontal") * playerVelocity;

        // выход из игры
        if (Input.GetKeyDown(KeyCode.Escape)){
            Application.Quit();
        }

        // обновим позицию платформы
        transform.position = playerPosition;

                              // проверка выхода за границы
        if (playerPosition.x < -boundary) {
            transform.position = new Vector3 (-boundary, playerPosition.y, playerPosition.z);
        }
        if (playerPosition.x > boundary) {
            transform.position = new Vector3(boundary, playerPosition.y, playerPosition.z);
        }
    }
}
```

Теперь вернитесь в редактор и, переключаясь в игру, найдите оптимальное значение переменной boundary. В нашем случае подошло число 5.46. Откройте **Inspector** и сбросьте позицию платформы по оси X на 0, а параметр **Boundary** выставьте согласно найденному вами значению.

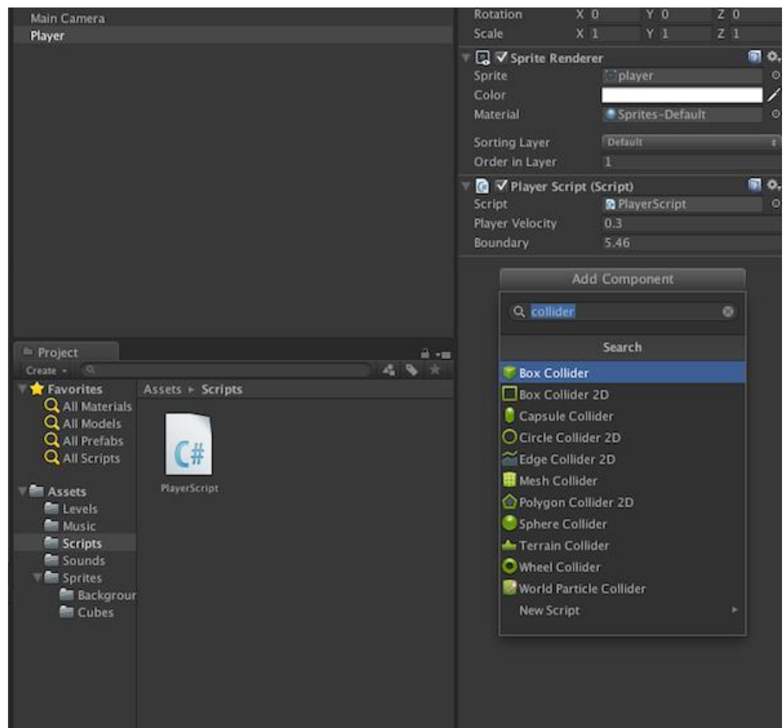
Нажмите кнопку **Play** и убедитесь в том, что вы все сделали правильно. Платформа должна двигаться только в пределах игрового поля.

Включение физики

Чтобы столкновения были более реалистичные — воспользуемся симуляцией физики. В этой статье мы добавим физические свойства мячику, платформе и границам поля. Так как мы пишем 2D игру, то будем использовать 2D коллайдеры. Коллайдер — это отдельный тип компонентов, который позволяет объекту реагировать на коллайдеры других объектов.

В окне **Hierarchy** выберем нашу платформу, перейдем в **Inspector** и нажмем на кнопку **Add Component**. В появившемся окошке наберем collider. Как вы можете увидеть — вариантов

достаточно много. Каждый коллайдер имеет специфические свойства, соответствующие связанным объектам — прямоугольникам, кругам и т.д.



Так как наша платформа имеет прямоугольную форму, мы будем использовать **Box Collider 2D**. Выберите именно его, и компонент автоматически определит размеры платформы: вам не нужно будет задавать их вручную, Unity сделает это за вас.

Сделайте то же самое и для 3 границ (**Hierarchy -> Inspector -> Add Component -> Box Collider 2D**).

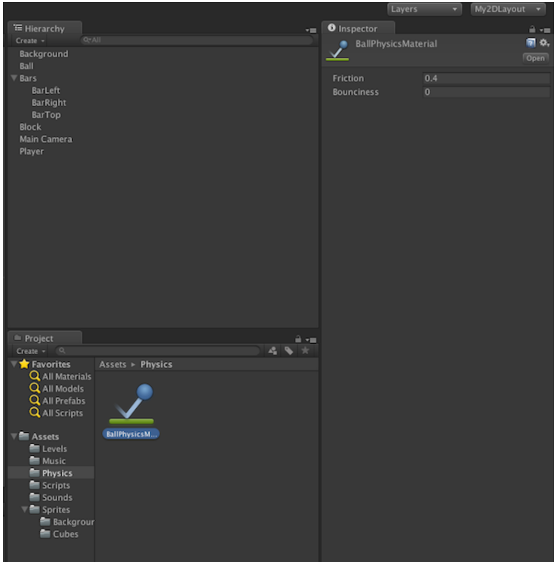
С мячиком чуть-чуть по другому: он имеет круглую форму. Выберем мяч и добавим для нее компонент **Circle Collider 2D**.

На самом деле коллайдер окружности и прямоугольника очень похожи, за исключением того, что вместо параметра **Size**, определяющим ширину и длину, в окружности используется **Radius**. Объяснения здесь, думаем, излишни.

Упругое столкновение

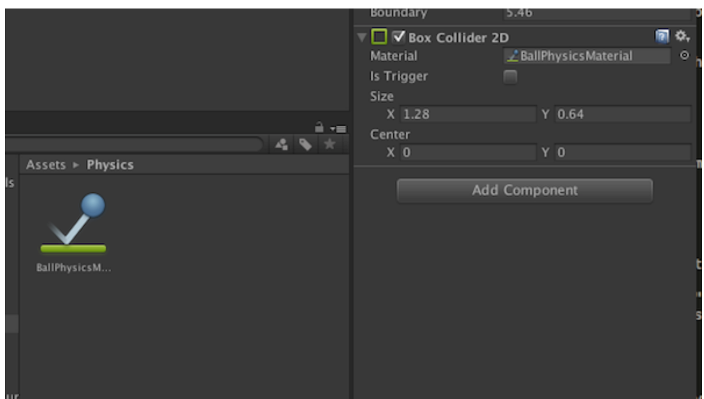
Для того, чтобы наш мячик отскакивал от блоков, стен и платформы, нам следует задать поверхность *(material)* для физического компонента, добавленного ранее. В Unity все уже имеется, нам остается только добавить нужный материал.

Откройте окно **Project** и внутри папки **Asset** создайте новую папку под названием Physics. Кликните по только что созданной папке правой кнопкой мыши и выберите **Create -> Physics2D Material**. Задайте название BallPhysicsMaterial.



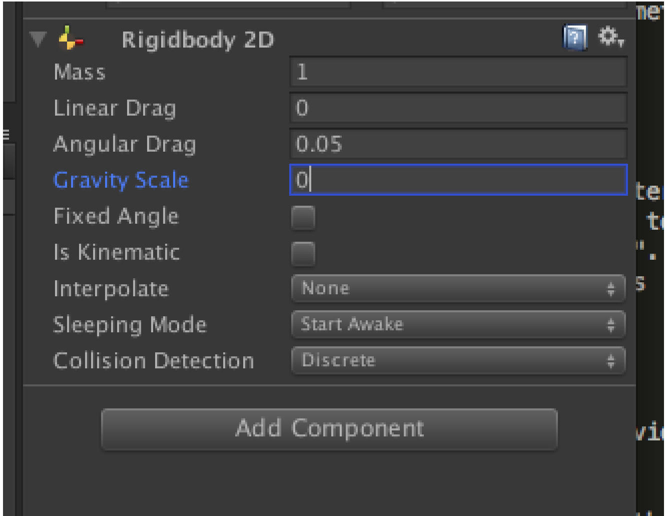
Каждая поверхность в Unity имеет два параметра: трение *(friction)* и упругость *(bounciness)*. Более подробно вы можете прочитать про физический движок и ряд физических параметров в одной из наших статей про физический движок. Если вам требуется абсолютно упругое тело, то следует выставить трение на 0, а упругость на 1.

Сейчас у нас есть готовый материал, но он пока никак не связан с мячом. Выберите объект мяча во вкладке **Hierarchy** и в окне **Inspector** вы увидите поле **Material**компонента **Circle Collider 2D**. Перетащите сюда недавно созданный материал.



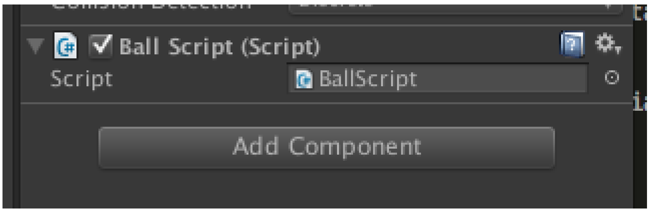
Добавление компонента **Rigid Body**

Для того, чтобы наш мячик двигался под контролем физики, мы должны добавить ему еще один компонент: **Rigid Body 2D**. Выберите объект мяча в окне **Hierarchy** и добавьте вышеупомянутый компонент — хоть он и имеет несколько параметров, нас интересует только один: **Gravity Scale**. Так как наш шарик будет двигаться только за счет отскоков, то мы зададим этому параметру 0 — таким образом мы гарантируем, что гравитация не будет реагировать на объект. Все остальное можно не менять.



Поведения шарика

Давайте создадим для шарика отдельный скрипт (снова воспользуемся C# в качестве языка программирования) и назовем его BallScript. Свяжите созданный скрипт с объект (**Hierarchy -> Inspector -> Add Component**).



Перед тем, как начать писать скрипт, давайте определим поведение шарика:

1. Шар имеет два состояния: неактивное (когда он в начале игры находится на платформе) и активное (когда находится в движении).
2. Шар будет становиться активным только один раз.
3. Когда шар становится активным, мы применяем к нему силу для того, что он начал движение.
4. Если шар вышел за пределы игрового поля, он переводится в неактивное состояние и помещается на платформу.

Основываясь на этой информации, давайте создадим глобальные переменные ballIsActive, ballPosition и ballInitialForce:

```
private bool ballIsActive;
private Vector3 ballPosition;
private Vector2 ballInitialForce;
```

Теперь, когда у нас есть набор переменных, мы должны подготовить объект. В методе Start() мы должны:

- создать силу, которая будет применена к шару;
- перевести шар в неактивное состояние;

- запомнить позицию шара.

Вот, как это можно сделать:

```
void Start () {
		// создаем силу
		ballInitialForce = new Vector2 (100.0f,300.0f);

		// переводим в неактивное состояние
		ballIsActive = false;

		// запоминаем положение
		ballPosition = transform.position;
}
```

Как вы могли заметить, сила прилагается не строго вертикальная, а наклоненная вправо — шарик будет двигаться по диагонали.

Далее, в методе Update() мы воспользуемся стандартной кнопкой Unity — Jump. Изначально под этой кнопкой подразумевается пробел, но это можно изменить:

```
void Update () {
   // проверка нажатия на пробел
   if (Input.GetButtonDown ("Jump") == true) {

   }
}
```

Следующим шагом является проверка состояния шара, поскольку задать силу нам надо только в том случае, если шар находится в неактивном состоянии:

```
void Update () {
   // проверка нажатия на пробел
   if (Input.GetButtonDown ("Jump") == true) {
			// проверка состояния
			if (!ballIsActive){

			}
		}
}
```

Если предположить, что мы находимся в начале игры, то мы должны применить силу к шару и установить его в активное состояние:

```
void Update () {
   // проверка нажатия на пробел
   if (Input.GetButtonDown ("Jump") == true) {
			// проверка состояния
```

```
                if (!ballIsActive){
                        // применим силу
                        rigidbody2D.AddForce(ballInitialForce);
                        // зададим активное состояние
                        ballIsActive = !ballIsActive;
                }
        }
}
```

Если теперь вы включите игру, то, нажав на пробел, шар действительно начнет движение. Однако вы можете заметить, что мяч в неактивном состоянии ведет себя не совсем правильно: если мы будем двигать платформу, то мяч должен двигаться вместе с ней, но на самом деле остается на прежней позиции. Остановите игру, давайте исправим это.

В методе Update мы должны проверять состояние шарика, и в случае если оно неактивное, нам надо задать позицию мячика по оси X таким же, какое оно у платформы.

Решение достаточно простое, но как нам получить координату совсем другого объекта? Элементарно — мы создадим переменную типа GameObject и сохраним ссылку на объект платформы:

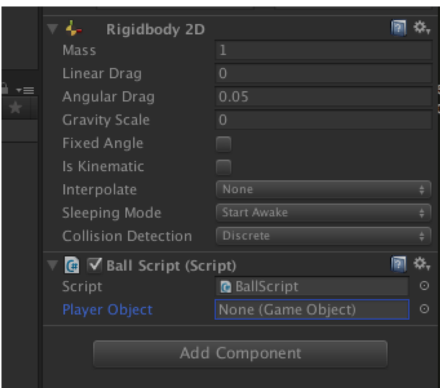
```
public GameObject playerObject;
```

Вернемся к методу Update():

```
void Update () {
  // проверка нажатия на пробел
  if (Input.GetButtonDown ("Jump") == true) {
                // проверка состояния
                if (!ballIsActive){
                        // применим силу
                        rigidbody2D.AddForce(ballInitialForce);
                        // зададим активное состояние
                        ballIsActive = !ballIsActive;
                }

                if (!ballIsActive && playerObject != null){
      // задаем новую позицию шарика
      ballPosition.x = playerObject.transform.position.x;

      // устанавливаем позицию шара
      transform.position = ballPosition;
    }
  }
}
```

Сохраните скрипт и вернитесь в редактор Unity. Вы наверняка заметили, что переменная playerObject объявлена, используется, но нигде не инициализирована. Да, так и есть.

Чтобы ее проинициализировать, перейдите во вкладку **Hierarchy**, найдите шар и в окне **Inspector** найдите компонент **Ball Script**. У данного компонента есть параметр **Player Object**, в настоящее время пустующий:



Найдите во вкладке **Hierarchy** нашу платформу и перетащите ее на поле **Player Object**. Запустите игру, нажав кнопку **Play**, и убедитесь, что все работает.