1 Разработка игры Arcanoid2D

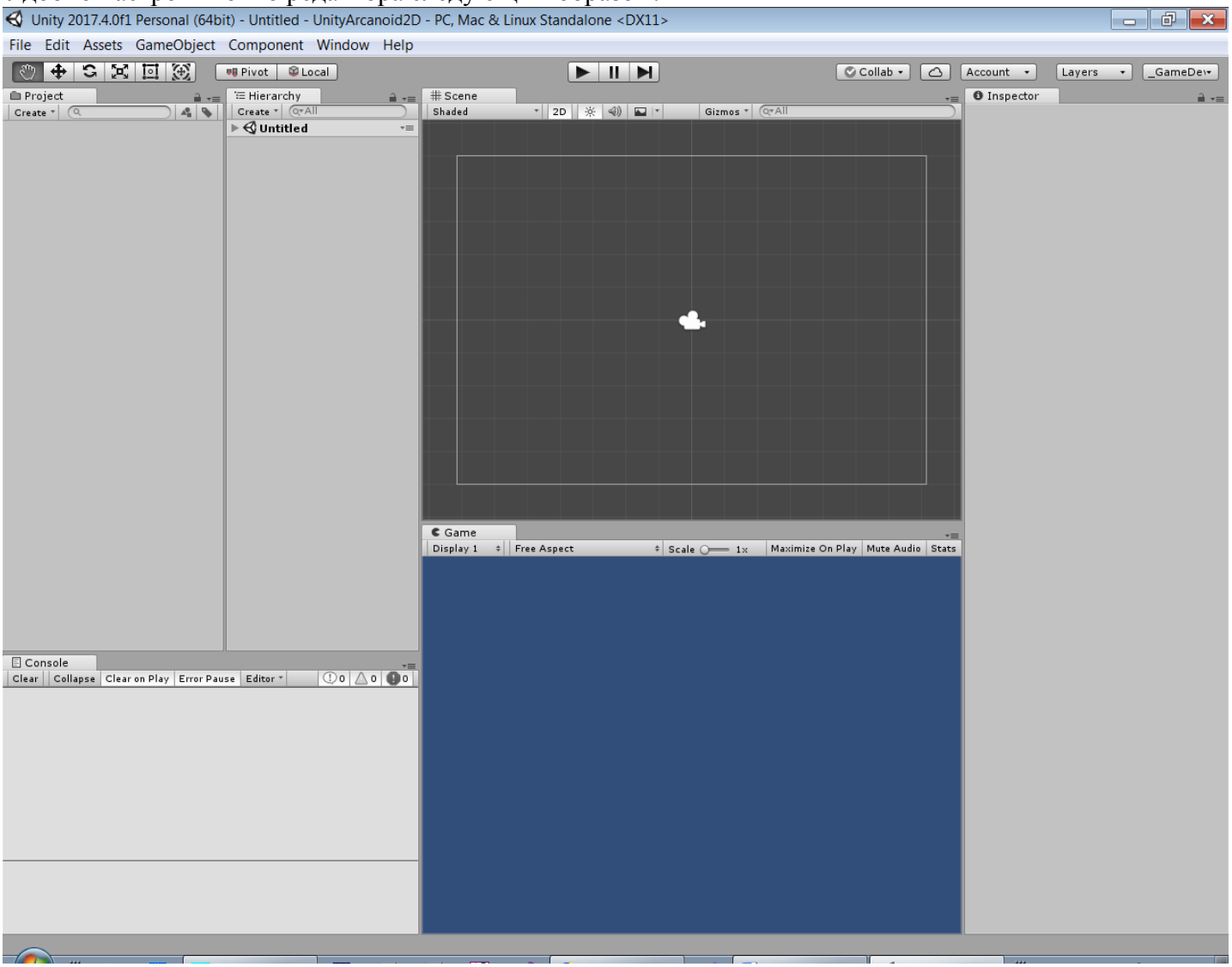
Содержание

1.1 Создание проекта и подключение дополнительных ресурсов

1.1.1 Создание проекта и настройки окна редактора

После запуска Unity создайте новый проект (кнопка New), укажите имя UnityArcanoid2D и установите флажок 2D.

Удобно настроить окно редактора следующим образом:



Такая настройка позволяет одновременно видеть окно игры и в режиме разработки и в игровом режиме.

Для изменения взаимного расположения частей окна редактора достаточно зацепить ярлычок требуемой части и перетащить его на новую позицию.

Кроме того, в окне проекта (Project) установлен режим «в один столбец», что позволяет использовать пространство экрана более экономно (для установки этого режима надо вызвать меню окна проекта (значок в правом верхнем углу) и выполнить команду One Column Layout).

Желательно также иметь на экране окно консоли (Console). Если оно отсутствует, то его можно отобразить командой меню Window | Console.

После завершения настройки конфигурации окна ее целесообразно сохранить, нажав на крайнюю правую кнопку в верхней части окна, выполнив в появившемся меню команду Save Layout… и введя новое имя (это имя появится на кнопке; в данном случае было введено имя _GameDev).

1.1.2 Добавление внешних файлов

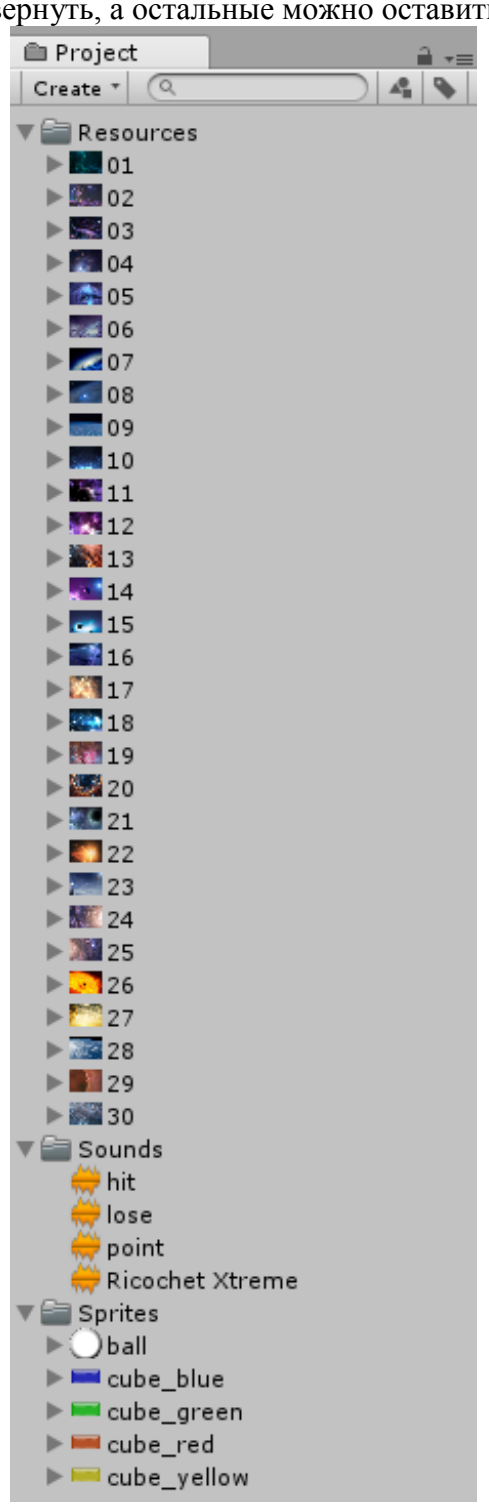
В проекте будет использоваться набор внешних файлов (спрайты для игровых объектов, рисунки для фонов и звуковые файлы). Эти файлы можно добавить в проект непосредственно с помощью любого файлового менеджера, создав для них соответствующие подкаталоги каталога Assets.

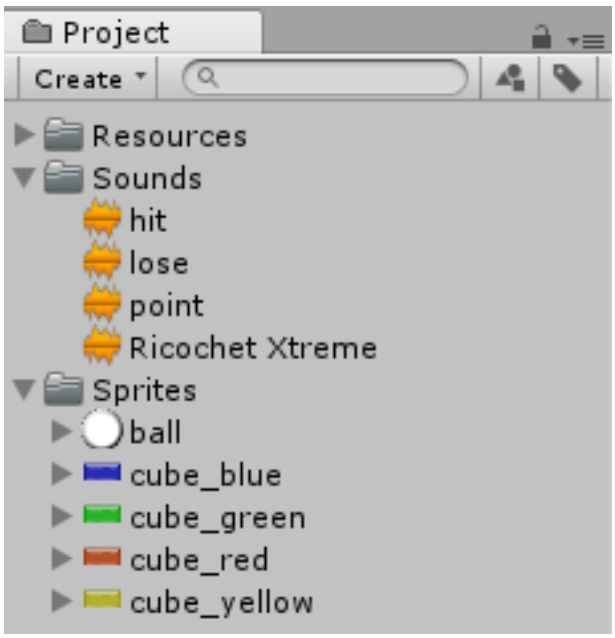
В подкаталог Sounds надо скопировать звуковые файлы hit.mp3 (удар шарика по препятствию), point.mp3 (получение новых очков), lose.mp3 (потеря шарика) и Ricochet Xtreme.mp3 (фоновая музыка продолжительностью примерно 2 мин.).

В подкаталог Sprites надо скопировать изображения шарика и блоков разного цвета, которые он будет разбивать: ball.png, cube_blue.png, cube_red.png, cube_green.png, cube_yellow.png. Блоки синего цвета будут также использоваться для создания стен, а увеличенный шарик (точнее, его верхняя часть) будет использоваться в качестве ракетки полукруглой формы.

В подкаталог Resources надо скопировать 30 изображений для фона (по одному изображению на каждый уровень игры) с именами 01.png, 02.png, …, 30.png. В этих изображениях использована космическая тематика, все изображения имеют формат 1024 x 768.

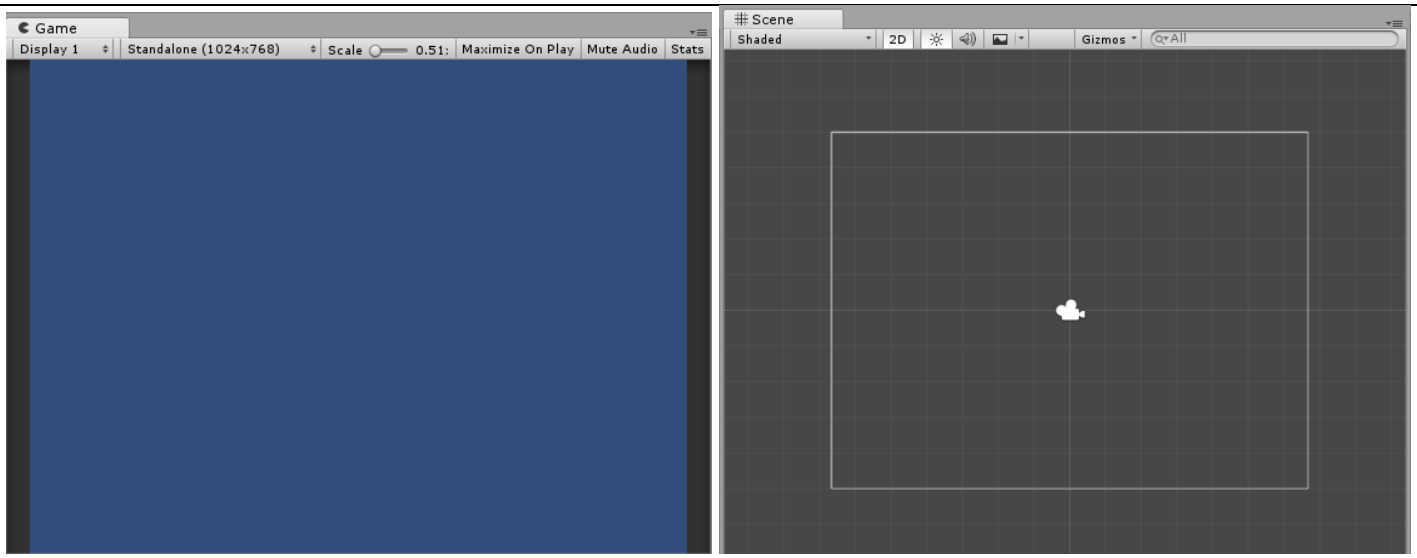
После возврата в окно редактора Unity произойдет автоматическое импортирование всех добавленных файлов, и они появятся в соответствующих папках окна проекта (рисунок слева). Папку Resources удобно свернуть, а остальные можно оставить развернутыми (рисунок справа).





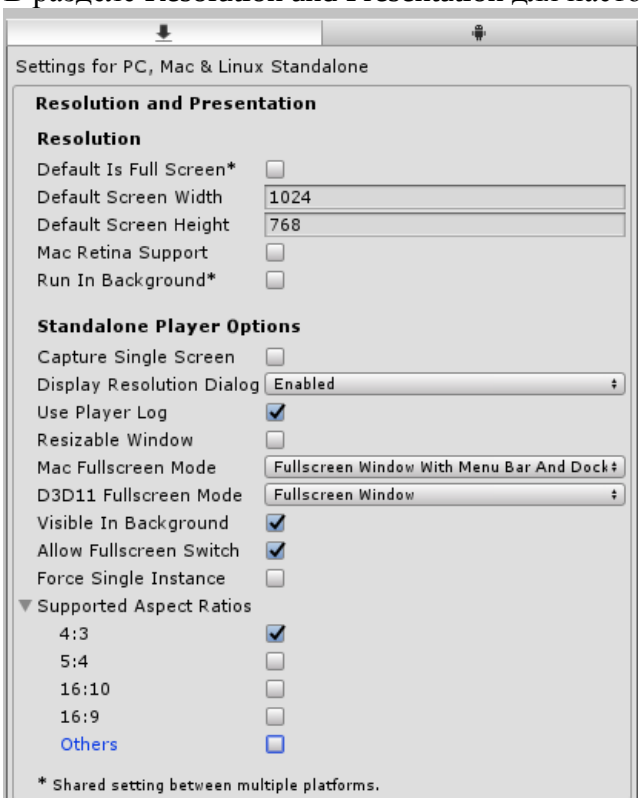
1.1.3 Настройка визуальных характеристик игры

Для окна игры (Game) в списке размеров (второй выпадающий список в верхней части окна) выберите вариант Standalone (1024x768); в результате для сцены будет установлено указанное разрешение (рисунок слева). Рамка сцены в окне сцены (Scene) тоже будет соответствовать этому разрешению; вращая колесико мыши в окне сцены, можно приблизить ее или отдалить (рисунок справа).



Выполните команду Edit | Project Settings | Player; в окне инспектора (Inspector) появится список настроек игры. Убедитесь, что название игры соответствует названию проекта (UnityArcanoid2D); это название появится в заголовке окна запущенной игры. Имя компании (DefaultCompany) можно не изменять.

В разделе Resolution and Presentation для настольных PC укажите следующие настройки.



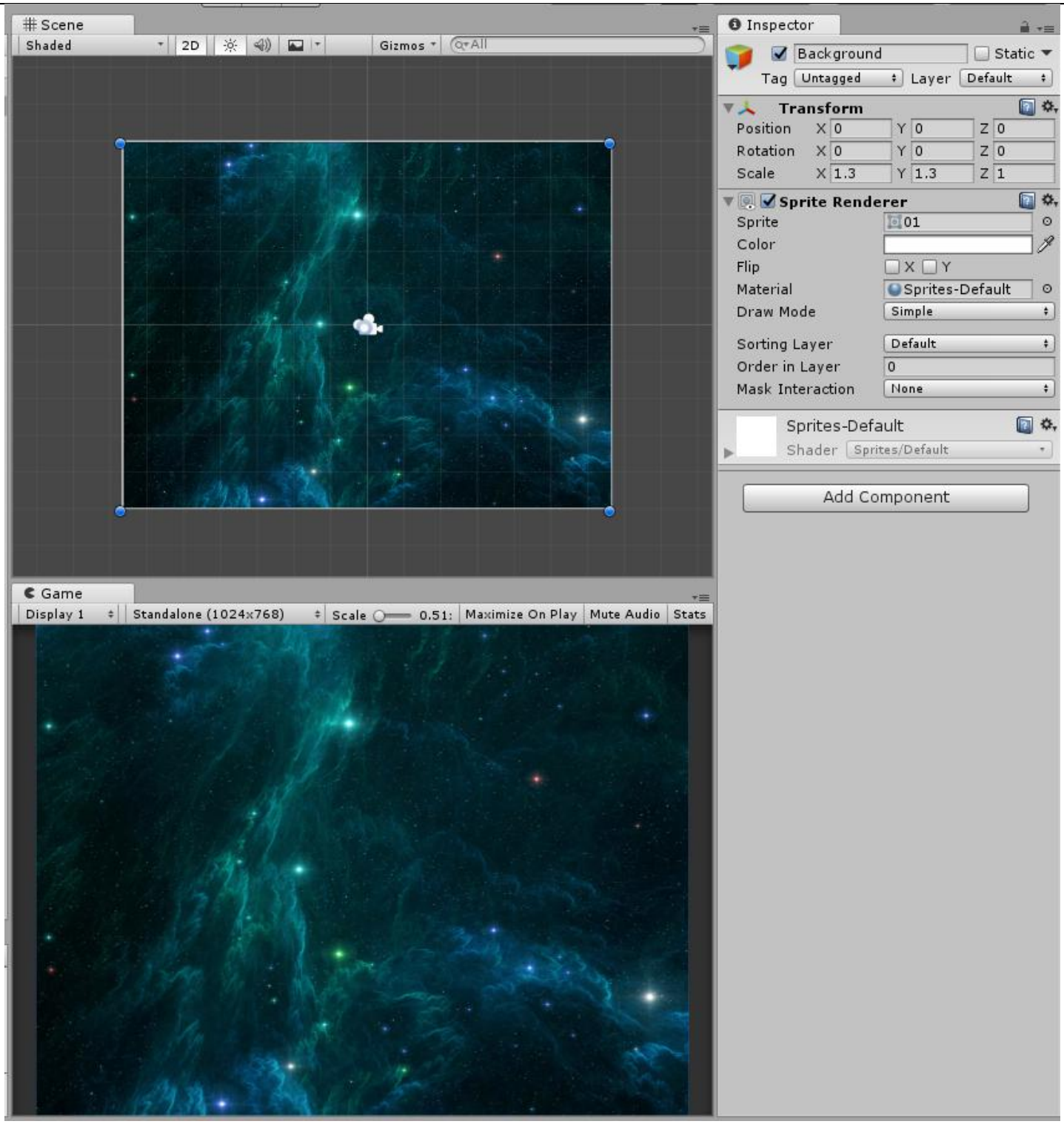
Настройки размеров окна будут установлены автоматически, если снять флажок Default Is Full Screen. Значение Enabled для свойства Display Resolution Dialog позволят выбирать разрешение окна сразу после запуска игры (в версии Unity 2019 эта настройка была удалена). Для поддерживаемых соотношений сторон (Supported Aspect Ratios) оставлен только вариант, соответствующий исходному соотношению (4:3).

1.2 Настройка неподвижных элементов сцены (фона и стен) в режиме дизайна

Для добавления и настройки неподвижных элементов сцены разработка скриптов не требуется, достаточно создавать нужные игровые объекты с помощью окна иерархии, снабжать их дополнительными компонентами и настраивать их свойства.

С помощью кнопки Create окна иерархии создайте новый объект Sprite (2D Object | Sprite) и переименуйте его в Background (это можно сделать, дважды щелкнув на его имени в окна иерархии или изменив верхнее поле в окне инспектора).

На свойство Sprite компонента Sprite Renderer в окне инспектора перетащите файл 01.png из папки Resources окна проекта. Если теперь изменить настройки компонента Transform так, как указано на рисунке, то фоновое изображение полностью заполнит сцену.



После первого изменения сцены целесообразно сохранить ее. Для этого достаточно нажать Ctrl+S. При первом сохранении сцены будет предложено указать ее имя. Введите имя MainScene, после чего соответствующий элемент появится в окне проекта.

В дальнейшем после каждого обновления содержимого сцены следует сохранять ее с помощью той же комбинации клавиш.

Теперь добавим стены. Создадим четыре стены, из которых верхняя, левая и правая будут видны на экране (от них будут отражаться шарики), а нижняя стена не будет видна, но при попадании на нее шарик будет уничтожаться (при этом будет возникать впечатление, что шарик улетел с игрового поля).

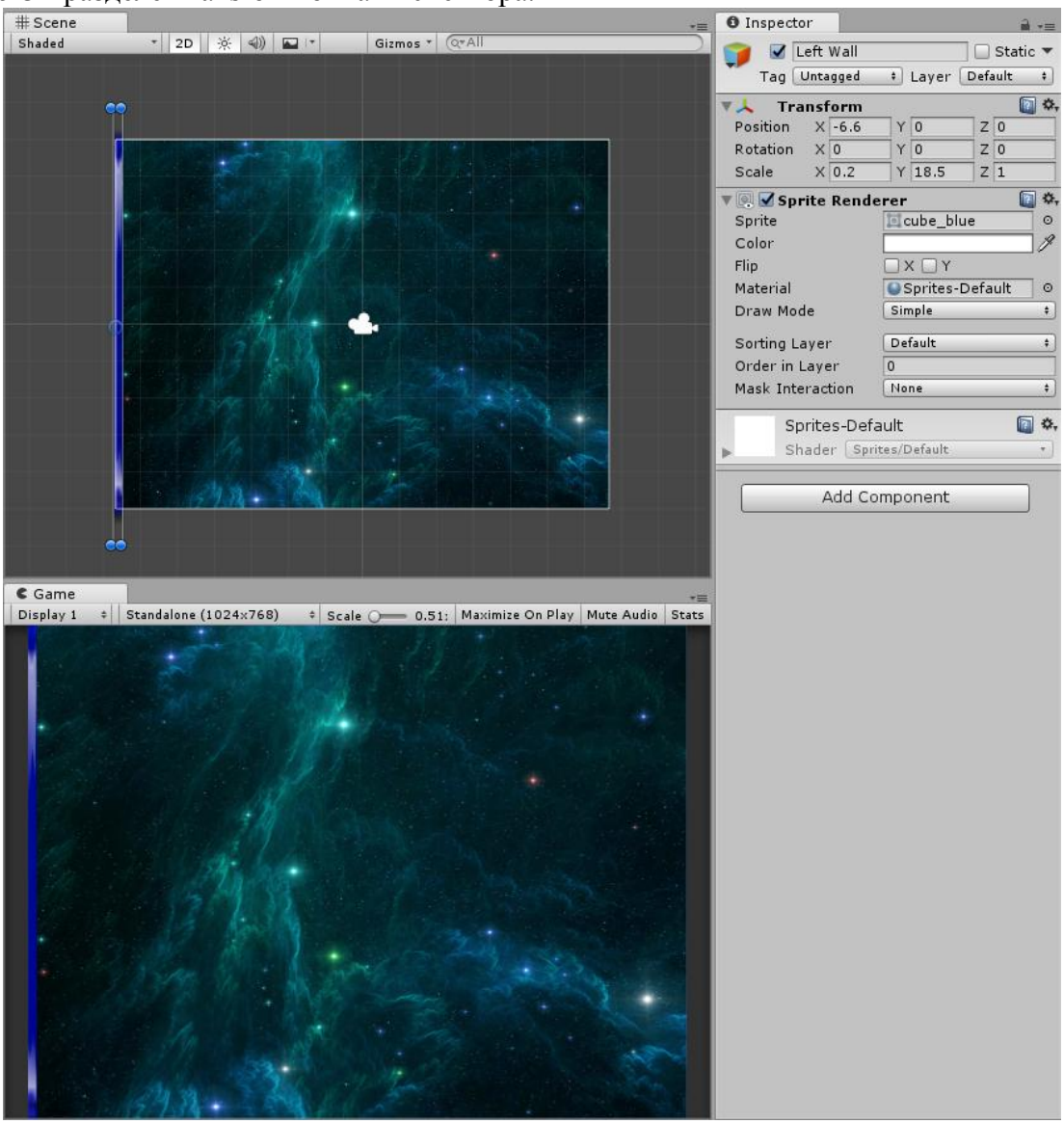
Для создания всех стен будем использовать спрайт для блока синего цвета. Для большей наглядности создадим пустой объект Wall, для которого все стены будут дочерними: Create | Create Empty в окне иерархии; переименуйте этот объект в Walls. Для данного объекта установите значения свойства Position в окне инспектора равными 0 (это важное действие, так как все дочерние объекты позиционируются относительно своего родительского объекта).

Для создания дочерних объектов проще всего щелкнуть правой кнопкой мыши на объекте Walls и выбрать из меню соответствующую команду (в нашем случае 2D Object | Sprite). Переименуйте созданный объект в Left Wall и перетащите в поле Sprite рисунок cube_blue из папки Sprites окна проекта.

Нам надо настроить размеры добавленного объекта так, чтобы он образовал высокую узкую левую стену. Для «грубой» настройки удобно использовать визуальные средства, предоставляемые инструментом Rect Tool, который выбран по умолчанию для игровых 2D-проектов.

Используя этот инструмент, можно перемещать объект по сцене, изменять его размеры, а также вращать его (последнюю возможность мы не будем использовать).

Перетащите объект Left Wall к левой границе сцены, сделайте его узким и высоким. Итоговый вид можно проверить по окну Game. После грубой настройки выполните «тонкую» настройку свойств Position и Scale в разделе Transform окна инспектора.

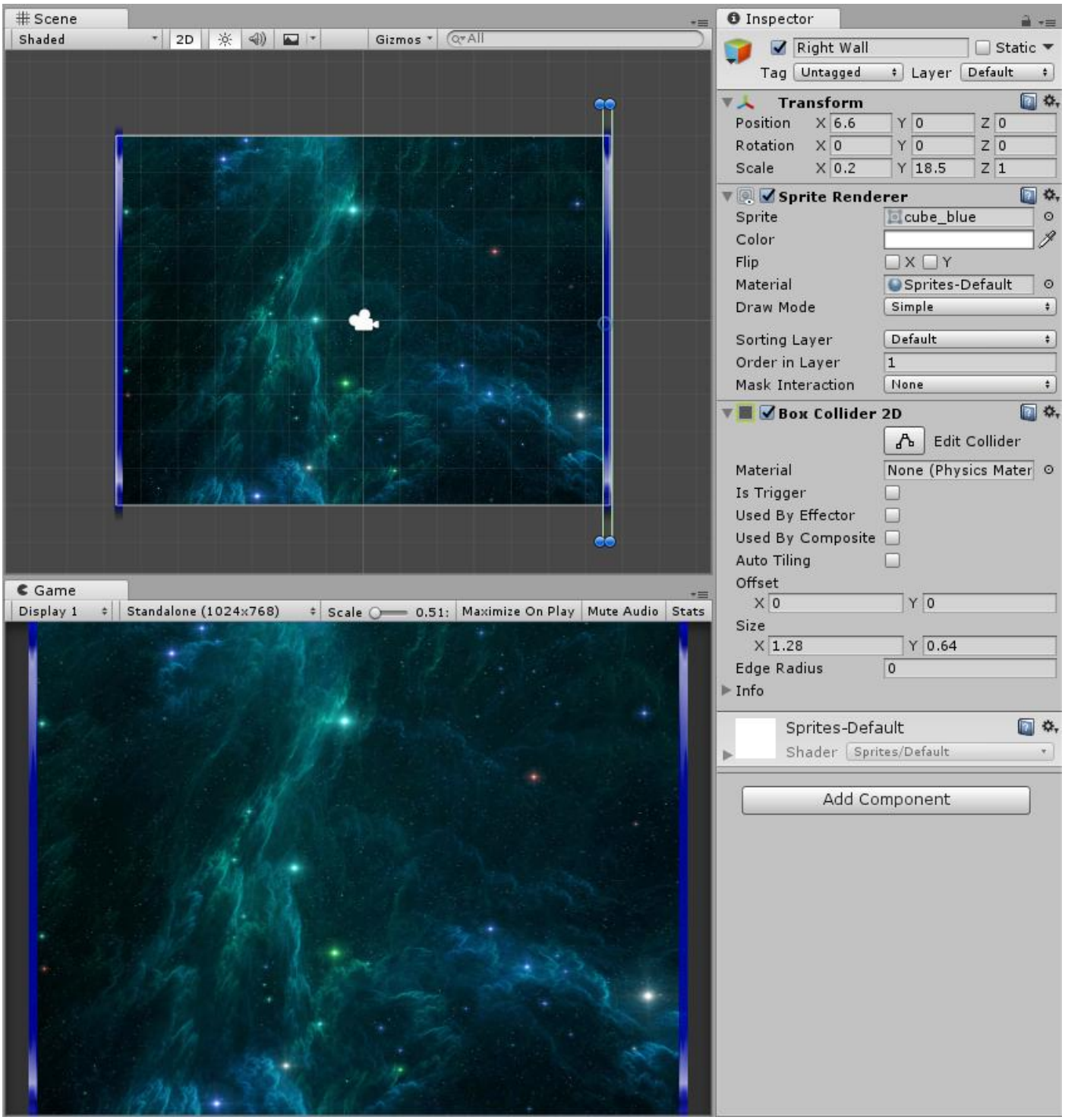


Чтобы обеспечить вывод изображения стены поверх фонового рисунка, укажем свойство Order in Layer равным 1. Заметим, что стена и ранее рисовалась поверх фонового рисунка, однако в дальнейшем, при последующей загрузке данного проекта, такая ситуация может измениться, и гарантировать правильный порядок можно будет только указав для данных объектов разные номера уровней (объект тем *дальше* от зрителя, чем *меньше* его номер).

Прежде чем приступать к созданию новых стен, снабдим левую стену важным дополнительным компонентом, который обеспечит ее взаимодействие с шариком. Чтобы другие объекты реагировали на стену (как на физическое препятствие), необходимо снабдить ее компонентом-*коллайдером*. Для этого нажмите кнопку Add Component в окне инспектора при выбранном объекте Left Wall и выберите в списке вариант Physics 2D | Box Collider 2D. Обратите внимание на то, что в результате границы стены будут обведены зеленой рамкой, размеры которой соответствует размерам стены. Таким образом, физические границы стены (от которых будет отскакивать шарик) будут совпадать с границами ее изображения. Именно поэтому мы выбрали вариант Box Collider 2D, то есть коллайдер прямоугольного размера.

Разумеется, добавить этот компонент можно было и впоследствии, когда мы займемся реализацией взаимодействия игровых объектов, но благодаря такому «раннему» добавлению мы сможем быстрее создавать новые объекты, используя механизм копирования.

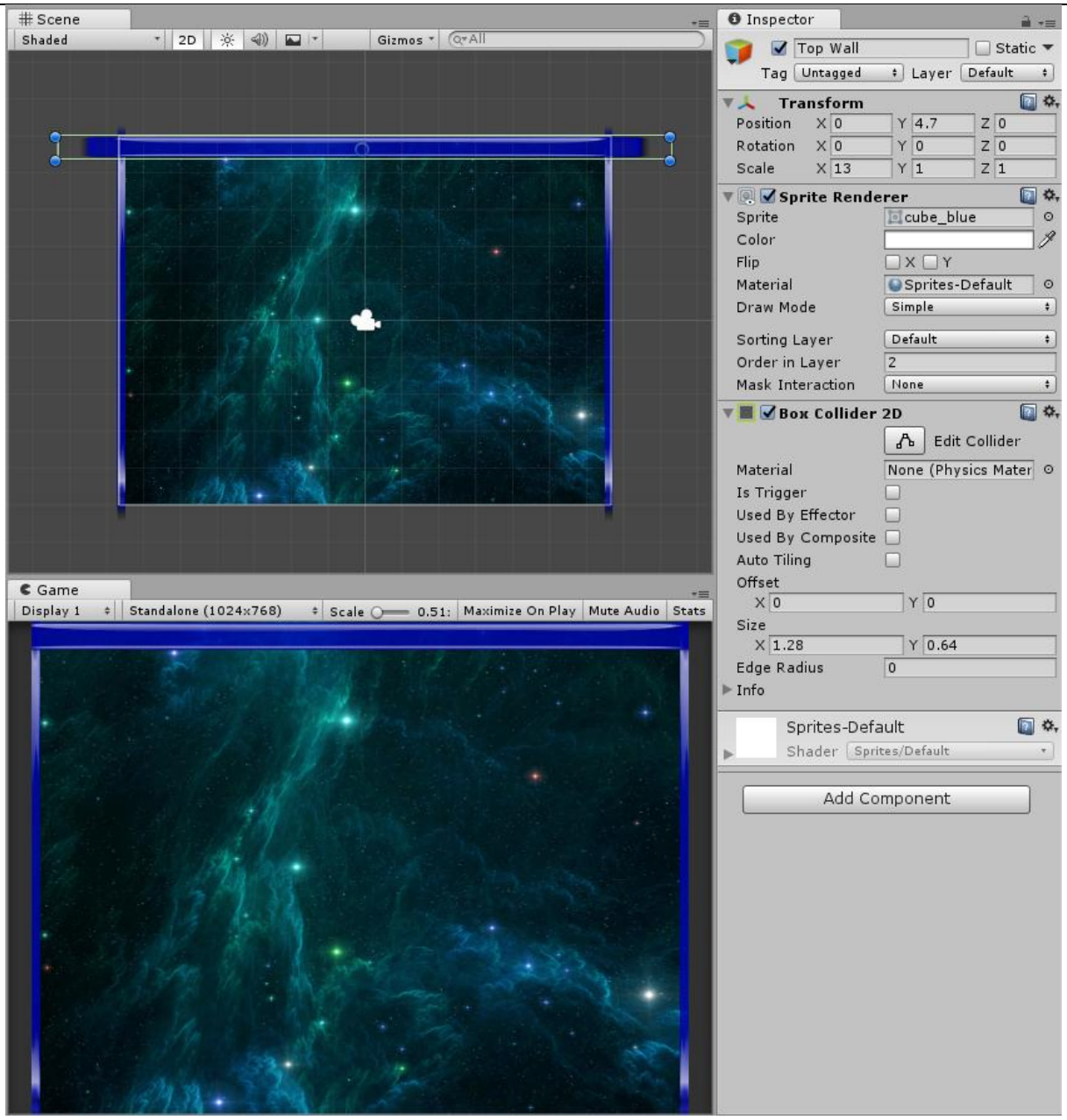
Создадим правую стену как копию объекта Left wall, после чего изменим ее имя и положение. Для этого выделите объект Left Wall в окне иерархии и нажмите Ctrl+D. Будет создан объект Left Wall (1), который надо переименовать в Right Wall и переместить на новую позицию. В данном случае для перемещения уже нет необходимости в инструменте Rect Tool, так как вполне достаточно просто удалить знак «минус» в свойстве Position X (изменив его значение с –6.6 на 6.6). Обратите внимание на то, что созданная копия получила все компоненты оригинала, в том числе и компонент-коллайдер, а ее свойство Order in Layer тоже равно 1.



Теперь создадим верхнюю стену. Для нее мы установим больший размер по вертикали, чем размеры левой и правой стены по горизонтали, так как в дальнейшем мы добавим на эту стену текст с информацией о состоянии игры и доступных клавишах.

Выполните копирование любой из стен Left Wall или Right Wall, переименуйте полученную копию в Top Wall и установите ее свойства из группы Transform так, как указано на рисунке. Обратите внимание на то, что масштаб по вертикали для данной стены равен 1; это значит, что высота верхней стены равна высоте исходного синего блока. Обратите также внимание на то, что размеры коллайдера автоматически подстраиваются под новые размеры объекта.

Чтобы гарантировать, что данная стена будет рисоваться поверх левой и правой стены, задайте для нее значение свойства Order in Layer равным 2.



Нам осталось добавить нижнюю, невидимую для игрока, стену, ударившись о которую шарик должен исчезнуть. Проще всего создать эту стену в виде копии верхней стены, после чего переименовать ее в Bottom Wall и изменить значение ее свойства Position Y на -6 (расстояние от верхнего края нижней стены до нижней границы сцены должно быть примерно равно высоте этой стены, так как диаметр шарика у нас тоже будет примерно равен этой высоте).

Для игровых объектов со сходными функциями полезно задать такую общую характеристику, как метка (Tag). Наряду с уже имеющимся набором стандартных меток, мы можем создавать свои метки. Для этого достаточно развернуть список Tag (для любого игрового объекта) и выполнить команду Add Tag…; в результате появится список созданных нами меток, в котором надо нажать символ «+» и ввести имя новой метки. В нашем случае введите имя Wall.

После этого можно перейти к нужному объекту в окне иерархии и установить для него эту метку. Так как мы хотим установить метку для всех стен, удобно выделить все объекты-стены (выделив первую стену, Left Wall, и затем щелкнув мышью на последней, Bottom Wall, держа нажатой клавишу Shift), после чего сразу задать метку Wall для всех выбранных стен.

Кроме того, полезно определить для всех стен специальный слой Layer (эта настройка расположена рядом с меткой). Предварительно надо создать слой для стен, развернув список слоев и выполнив команду Add Layer…, после чего указать имя нового слоя Wall в первом доступном поле ввода (User Layer 8). После этого, выделив все объекты-стены в окне иерархии, можно сразу задать для них слой Wall.

Заметим, что слои Layer играют другую роль по сравнению со слоями Sorting Layer, которые указываются в компоненте Sprite Renderer и обеспечивают нужный порядок отображения объектов на сцене. Слои Layer позволяют определить, какие объекты игры могут взаимодействовать между собой (как физические тела). В дальнейшем мы воспользуемся этой возможностью.

1.3 Создание ракетки и настройка ее управления с помощью мыши

1.3.1 Создание ракетки и настройка ее свойств

Теперь создадим ракетку — игровой объект, которым может управлять игрок.

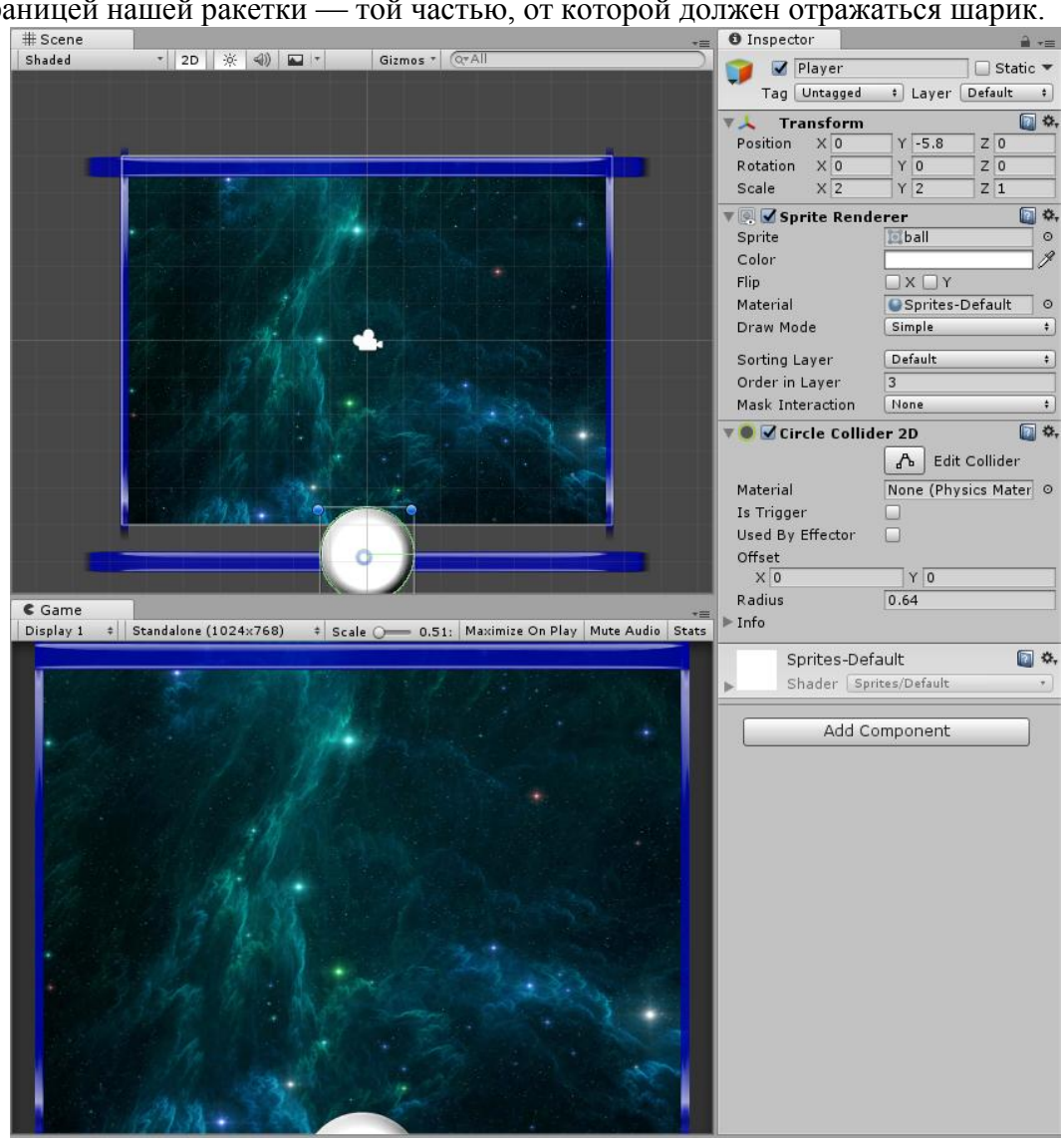
Обычно ракетки в игре «Арканоид» реализуют в виде горизонтальной платформы небольшой ширины. Однако такой вариант ракетки оставляет игроку мало возможностей для управления шариком, поскольку при ударе шарика почти в любой части такой платформы (за исключением краев) он будет отражаться под одним и тем же углом. Если бы ракетка имела овальную форму, то это позволило бы игроку более точно управлять шариком, подстраивая положение ракетки таким образом, чтобы шарик при ударе о нее отразился в желательном направлении. При этом от программиста не потребуется реализовывать сложный алгоритм расчета угла отражения, так как все подобные действия возьмет на себя физический движок системы Unity. Необходимо лишь использовать подходящие коллайдеры для регистрации соударения шарика и овальной ракетки. К сожалению, овальных коллайдеров в Unity не предусмотрено, однако имеется круговые коллайдеры, которыми мы можем воспользоваться.

Представим ракетку в виде верхней части шарика (так, чтобы основная часть этого шарика располагалась под игровой сценой). Тогда выступающая над нижней границей сцены часть шарика будет выглядеть как платформа с овальной границей, и нам удастся в точности совместить с этой границей соответствующий круговой коллайдер.

Реализуем эту идею. Поскольку ракетка является основным игровым объектом, назовем ее Player. Создадим в окне иерархии новый игровой 2D-объект Sprite с именем Player и перетащим на свойство Sprite компонента Sprite Renderer рисунок ball из папки Sprites окна проекта.

Настроим размеры и положение объекта Player так, как показано на рисунке. Кроме того, положим значение свойства Order in Layer равным 3, чтобы наша ракетка изображалась поверх стен.

Добавим к созданному объекту круговой коллайдер (Add Component | Physics 2D | Circle Collider 2D). Мы можем убедиться, что граница коллайдера совпадет с границей изображения, то есть, в частности, с верхней границей нашей ракетки — той частью, от которой должен отражаться шарик.



Свяжем созданный объект с меткой Player (эта метка уже присутствует в стандартном наборе меток) и, кроме того, зададим для него слой Player (этот слой потребуется создать).

Физический движок Unity особым образом обрабатывает неподвижные объекты, снабженные коллайдерами (в нашем случае это стены), сразу определяя их физические характеристики и в дальнейшем не выполняя их пересчет. Если объект, снабженный коллайдером, перемещается, то приходится после каждого перемещения пересчитывать его характеристики. Для повышения эффективности такого пересчета надо связывать с любым движущимся объектом, снабженным коллайдером, особый компонент Rigidbody 2D (находящийся в той же группе Physics 2D, что и все двумерные коллайдеры). Это позволит не только более эффективно обрабатывать столкновения данного объекта с другими игровыми объектами, но и обеспечит поведение этого объекта как «настоящего» физического тела, подчиняющегося физическим законам (что пригодится нам при реализации движения шарика).

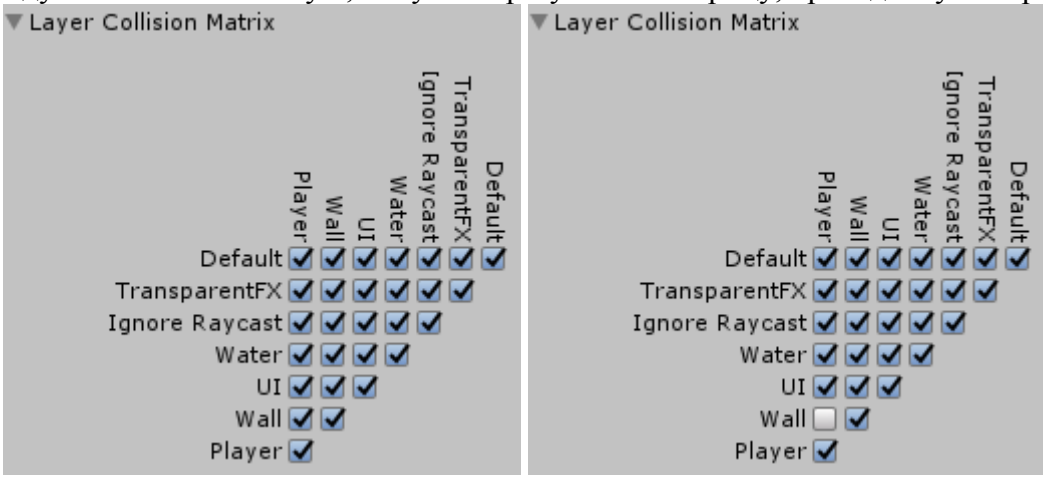
Итак, добавим к объекту Player компонент Rigidbody 2D (Add Component | Physics 2D | Rigidbody 2D). Для ракетки не требуется подчиняться физическим законам, поскольку она будет полностью управляться игроком. Поэтому необходимо положить значение Body Type равным Kinematic. Можно заметить, что после этого многие свойства компонента Rigidbody 2D просто исчезнут из окна инспектора. Заметим, что наряду со значением Body Type равным Dynamic (основным вариантом, обеспечивающим поведение объекта как «физического тела»), предусмотрен вариант Static, который фактически соответствует поведению объекта, имеющего коллайдер, но не снабженного компонентом Rigidbody 2D. Мы могли бы связать компонент Rigidbody 2D, имеющий значение Static, с неподвижными стенами, но это никак не сказалось бы на их поведении (и не сделало бы это поведение более эффективным). Поэтому основными вариантами настройки свойства Body Type являются Dynamic и Kinematic. Заметим также, что объекты с коллайдерами, не имеющие компонента Rigidbody 2D или имеющие компонент со свойствами Static или Kinematic, обладают «бесконечной массой»; это выражается в том, что при столкновениях с другими (динамическими) игровыми объектами они остаются неподвижными (в отличие от динамических объектов, отражающихся от них по законам физики).

1.3.2 Настройка матрицы взаимодействий

Физический движок Unity в большинстве ситуаций не контролирует взаимодействие между коллайдерами статических и «кинематических» объектов. Однако в некоторых ситуациях (когда для коллайдера установлен режим Is Trigger) такое взаимодействие всё же контролируется, что может приводить к излишним вычислениям. Имеется простой и универсальный способ отключить проверку взаимодействий между объектами, для которых это не требуется. Для этого достаточно разместить объекты в разных слоях и настроить матрицу взаимодействий между объектами этих слоев.

Продемонстрируем эту возможность, отключив любые взаимодействия (и проверку их наличия) между объектом Player и стенами на сцене. Выполним команду Edit | Project Settings | Physics 2D; в результате в окне инспектора появятся соответствующие настройки, из которых нас будет интересовать последний элемент, называемый матрицей взаимодействия слоев.

Этот элемент включает все слои, как стандартные, так и добавленные разработчиком, причем по умолчанию разрешены взаимодействия всех слоев (рисунок слева). Нам достаточно отключить взаимодействие между слоями Wall и Player, получив в результате матрицу, приведенную на рисунке справа.



Подчеркнем, что в данном случае это действие не является обязательным, так как оно лишь может обеспечить более эффективное функционирование физического движка. Однако в других ситуациях только корректировка матрицы взаимодействия слоев позволяет организовать правильное функционирование игры (мы встретимся с такой ситуацией, когда будем реализовывать вариант игры с несколькими шариками).

1.3.3 Управление ракеткой

Теперь обеспечим возможность перемещения ракетки игроком. Для этого можно было бы использовать клавиши со стрелками, однако более удобным и быстрым является перемещение ракетки с помощью мыши. При этом программе достаточно реагировать на горизонтальное движение мыши, причем никаких специальных действий по «захвату» ракетки мышью выполнять не требуется.

Реализация описанной возможности уже требует разработки программного скрипта. Все скрипты, создаваемые в проекте, будем размещать в папке Scripts, которую можно создать непосредственно в окне проекта (командой Create | Folder) или в подкаталоге Assets каталога с проектом.

Выделим созданную папку Scripts и создадим новый скрипт (Create | C# Script). Следует немедленно переименовать этот скрипт, поскольку если это не сделать сразу, исходное имя будет использовано при генерации заготовки скрипта. Назовем скрипт PlayerScript.

Чтобы открыть этот скрипт в окне среды MonoDevelop, достаточно выполнить двойной щелчок на имени этого скрипта в окне проекта. Заметим, что изменить среду, предназначенную для редактирования программных скриптов, можно с помощью команды Edit | Preferences | External Tools, указав новый вариант в поле External Script Editor.

Созданная заготовка будет иметь вид

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

В дальнейшем мы будем пользоваться другим стилем расстановки фигурных скобок, при котором открывающая фигурная скобка всегда находится на уровне закрывающей. Чтобы редактор среды MonoDevelop поддерживал такой стиль и обеспечивал автоматическое форматирование, надо выполнить команду Tools | Options и в разделе Source Code | Code Formatting для поля C# source code выбрать вариант Microsoft Visual Studio. Впрочем, эта настройка не повлияет на вид исходных заготовок, которые придется откорректировать следующим образом (можно также удалить комментарии, предваряющие описания методов, и пробелы между именем метода и открывающей скобкой):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerScript : MonoBehaviour
{
    void Start()
    {
    }

    void Update()
    {
    }
}
```

Добавим в методы новые операторы (они выделены полужирным шрифтом):

```
void Start()
{
    Cursor.visible = false;
}

void Update()
{
    var mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
    var pos = transform.position;
    pos.x = mousePos.x;
    transform.position = pos;
}
```

Сохраним измененный скрипт (нажав Ctrl+S), вернемся в редактор Unity и свяжем этот скрипт с объектом Player. Простейшим способом для этого является выбор объекта Player в окне иерархии и перетаскивание созданного скрипта из окна проекта в окно инспектора (которое в данный момент отображает все компоненты объекта Player). В результате в нижней части окна инспектора появится новый компонент объекта Player с именем Player Script.

Компиляция созданного и сохраненного скрипта выполняется автоматически. Если ошибок при компиляции не выявлено (то есть окно Console не содержит соответствующих сообщений), то можно запустить наш проект на выполнение в среде Unity, нажав комбинацию Ctrl+P или кнопку запуска над окном сцены (этой же комбинацией и кнопкой можно остановить выполнение проекта).

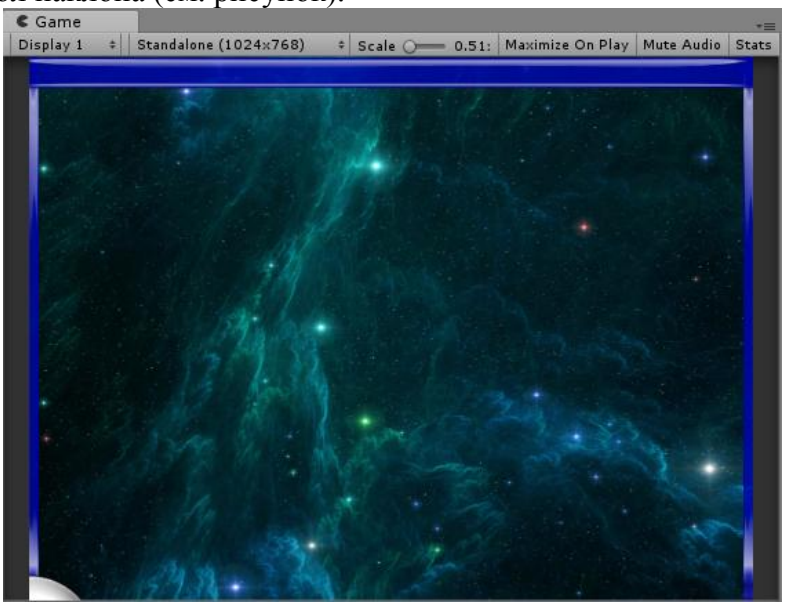
Если при запуске переместить мышь в окно игры, то изображение курсора мыши исчезнет, а ракетка начнет двигаться вслед за горизонтальным перемещением мыши.

Замечание 1. Вместо трех последних операторов метода Update нельзя использовать единственный оператор вида

```
transform.position.x = mousePos.x;
```

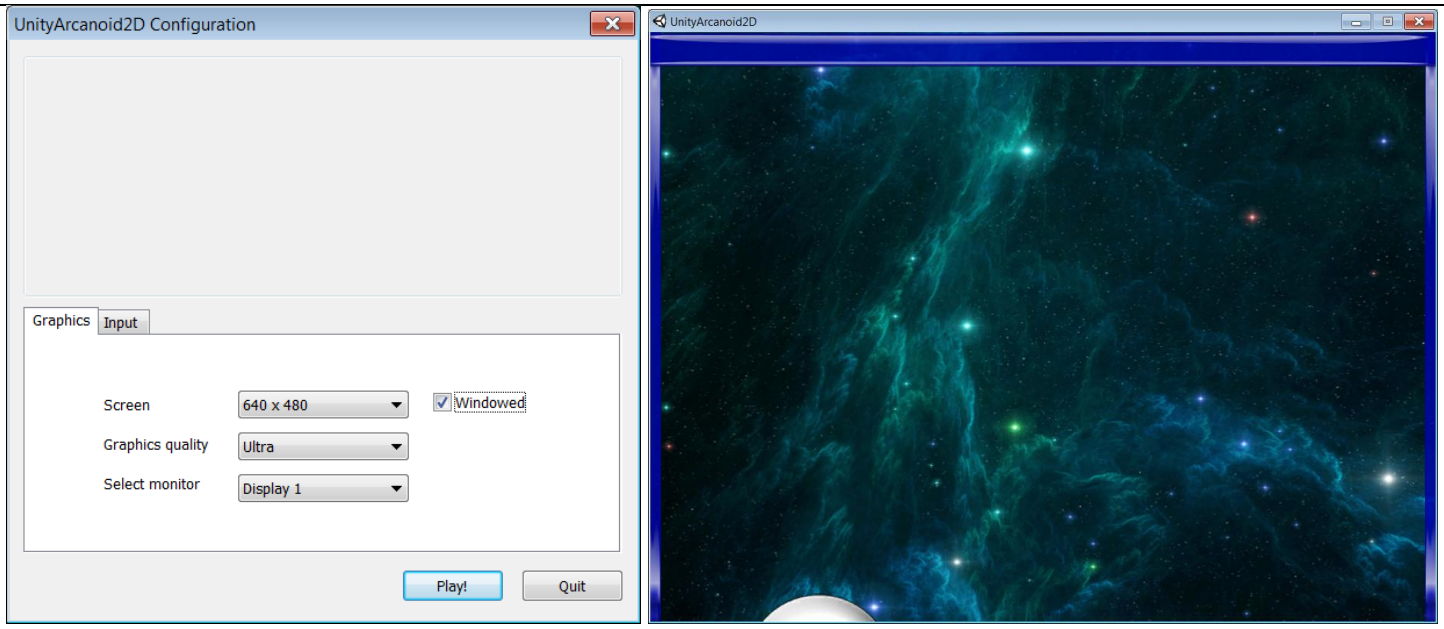
Это связано с тем, что position является не полем, а свойством класса transform, и поэтому не может быть изменено «по частям». Можно лишь сохранить его во вспомогательной переменной, изменить эту переменную и затем присвоить значение этой переменной свойству position «в целом».

Замечание 2. При такой реализации перемещения ракетки ее можно переместить и за границу сцены. Конечно, можно было бы добавить ограничение, не позволяющее ракетке перемещаться за вертикальные стены, однако использованный вариант является не только более простым, но и более удобным, так как позволяет отбивать шарик, летящий в угол сцены, той частью ракетки, которая имеет наиболее подходящий угол наклона (см. рисунок).



Мы можем также запустить проект в виде отдельной программы. Для этого достаточно выполнить команду File | Build and Run или нажать клавишу Ctrl+B. При первом таком запуске будет предложено указать имя exe-файла. Укажем имя, совпадающее с именем проекта: UnityArcanoid2D.

При запуске программы появится окно настроек, в котором можно выбрать разрешение для тестирования программы или установить полноэкранный режим (сняв флажок Windowed).



Пока завершить игру можно только стандартными средствами Windows (в частности, комбинацией Ctrl+F4). В дальнейшем мы реализуем более удобный способ ее завершения.

1.4 Создание шарика и настройка его поведения в различных режимах

1.4.1 Создание объекта Ball и настройка его свойств

Прежде чем мы создадим шарик — объект Ball — и начнем настраивать его свойства, мы создадим вспомогательный ресурс типа Physics Material 2D, который, будучи подключен к объекту Ball, обеспечит для него необходимое поведение.

Поместим этот новый ресурс в отдельную папку Physics, поскольку физические материалы позволяют настроить именно физические свойства тех объектов, для которых задаются. С помощью команды Create окна проекта добавим в эту папку ресурс Physics Material 2D и назовем его BallMaterial. Свойства этого материала зададим следующим образом: Friction положим равным 0 (трение у шарика отсутствует), Bounciness положим равным 1 (упругость у шарика является максимальной).

Теперь создадим в окне иерархии новый игровой 2D-объект типа Sprite, назовем его Ball и настроим ряд его свойств.

На свойство Sprite компонента Sprite Renderer перетащим рисунок ball из окна проекта.

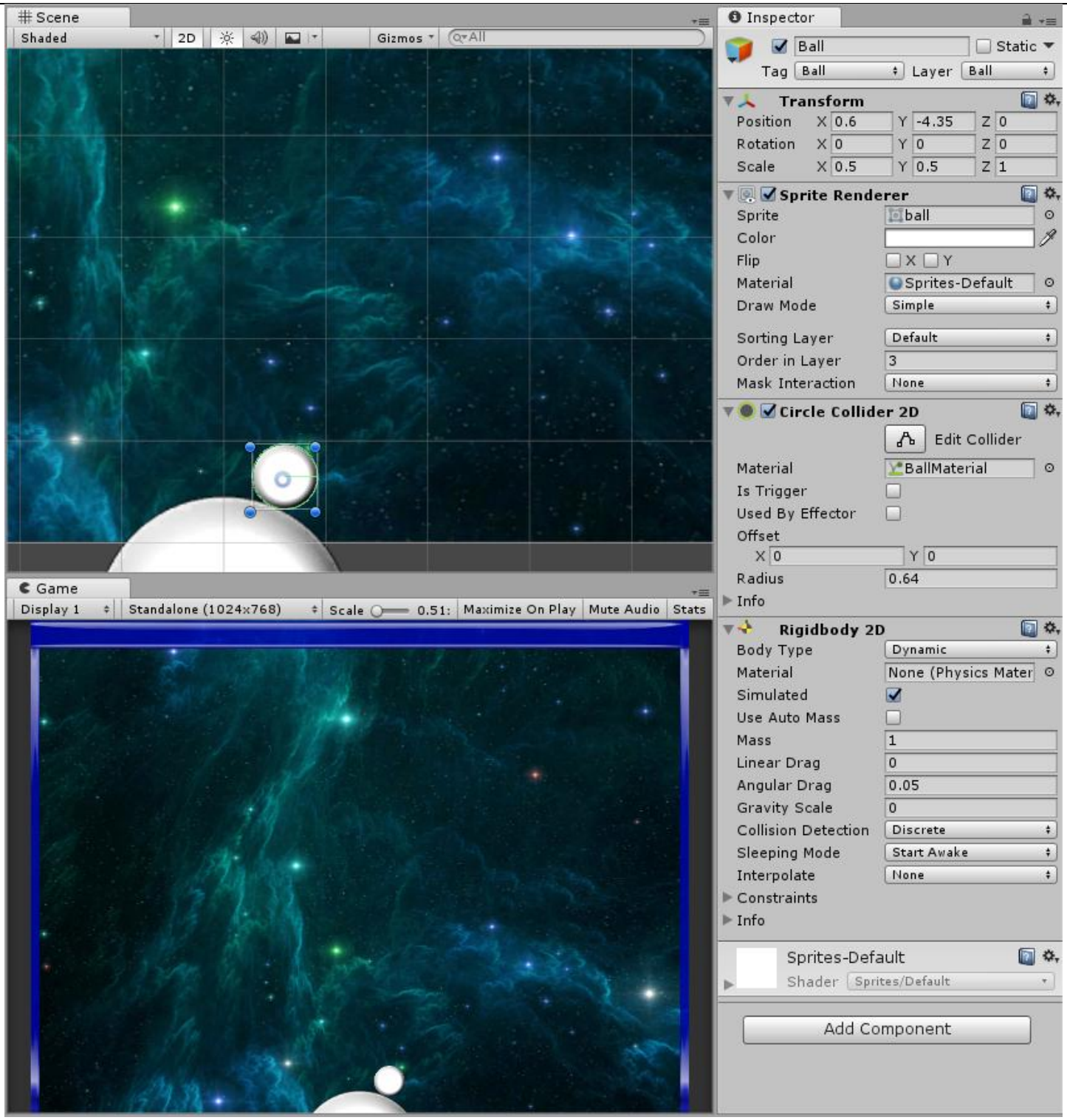
Размеры и положение шарика настроим таким образом, чтобы он располагался на правой половине ракетки, поскольку при запуске он полетит вверх и вправо.

Добавим к объекту Ball круговой коллайдер и объект Rigidbody 2D. Для объекта Rigidbody 2D при значении Body Type, равном Dynamic, обнулим свойство Gravity Scale (чтобы в ситуации, когда шарик начнет двигаться, на него не действовала сила притяжения), после чего установим значение Body Type, равное Kinematic (поскольку в начальный момент игры шарик должен находиться на ракетке и двигаться вместе с ней).

Установим для объекта Ball метку (Tag) и слой (Layer) с таким же именем, создав для этого новую метку и новый слой.

Наконец, зададим для коллайдера шарика созданный ранее физический материал BallMaterial, перетащив этот ресурс из окна проекта на свойство Material компонента Circle Collider 2D (материал компонента Rigidbody 2D менять не требуется).

В результате сделанных настроек свойства объекта Ball должны принять вид, приведенный на рисунке.



1.4.2 Скрипт для реализации поведения шарика

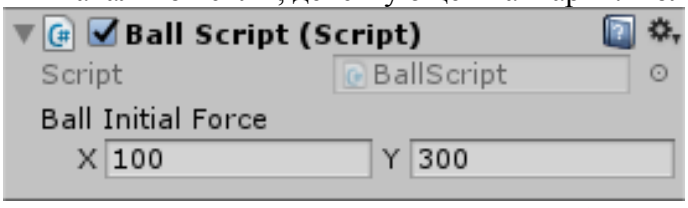
Теперь необходимо разработать скрипт, определяющий поведение шарика. Опишем детали этого поведения. В начальный момент шарик «приклеен» к ракетке и движется вместе с ней. Для запуска шарика надо выполнить щелчок левой кнопкой мыши (или нажать левую клавишу Ctrl). Сразу после этого шарик переходит из «кинематического» состояния в динамическое и начинает двигаться под действием начальной силы, подчиняясь физическим законам и отражаясь от стен и от ракетки. Если шарик вылетит за нижний край сцены (и коснется нижней, невидимой стены), то он разрушится.

Замечательным обстоятельством является то, что наиболее сложные процессы, связанные с отражением шарика от других игровых объектов, снабженных коллайдерами, реализовывать не требуется: они автоматически обеспечиваются физическим движком Unity. Таким образом, нам надо реализовать лишь синхронное перемещение шарика вместе с ракеткой на начальном этапе игры, перевод его в динамическое состояние при выполнении соответствующих действий пользователя (и придание ему начального импульса), а также его разрушение при касании нижней стены.

Создадим новый скрипт, назовем его BallScript и внесем в него следующие добавления (выделенные, как обычно, полужирным шрифтом):

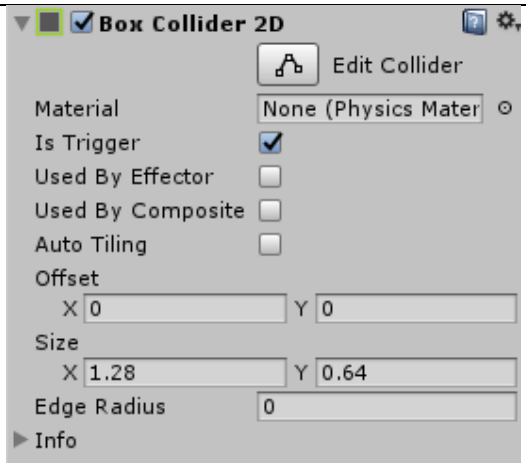
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallScript : MonoBehaviour
{
    public Vector2 ballInitialForce;
    Rigidbody2D rb;
    GameObject playerObj;
    float deltaX;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        playerObj = GameObject.FindGameObjectWithTag("Player");
        deltaX = transform.position.x;
    }

    void Update()
    {
        if (rb.isKinematic)
            if (Input.GetButtonDown("Fire1"))
            {
                rb.isKinematic = false;
                rb.AddForce(ballInitialForce);
            }
            else
            {
                var pos = transform.position;
                pos.x = playerObj.transform.position.x + deltaX;
                transform.position = pos;
            }
    }
}
```

Сохраним данный скрипт в редакторе MonoDevelop, вернемся в редактор Unity и перетащим этот скрипт в окно инспектора при выбранном объекте Ball. В результате в окне инспектора появится раздел для этого скрипта, в котором дополнительно будет содержаться поле Ball Initial Force. В это поле надо ввести компоненты начальной силы, действующей на шарик. Положим компоненты равными 100 и 300.



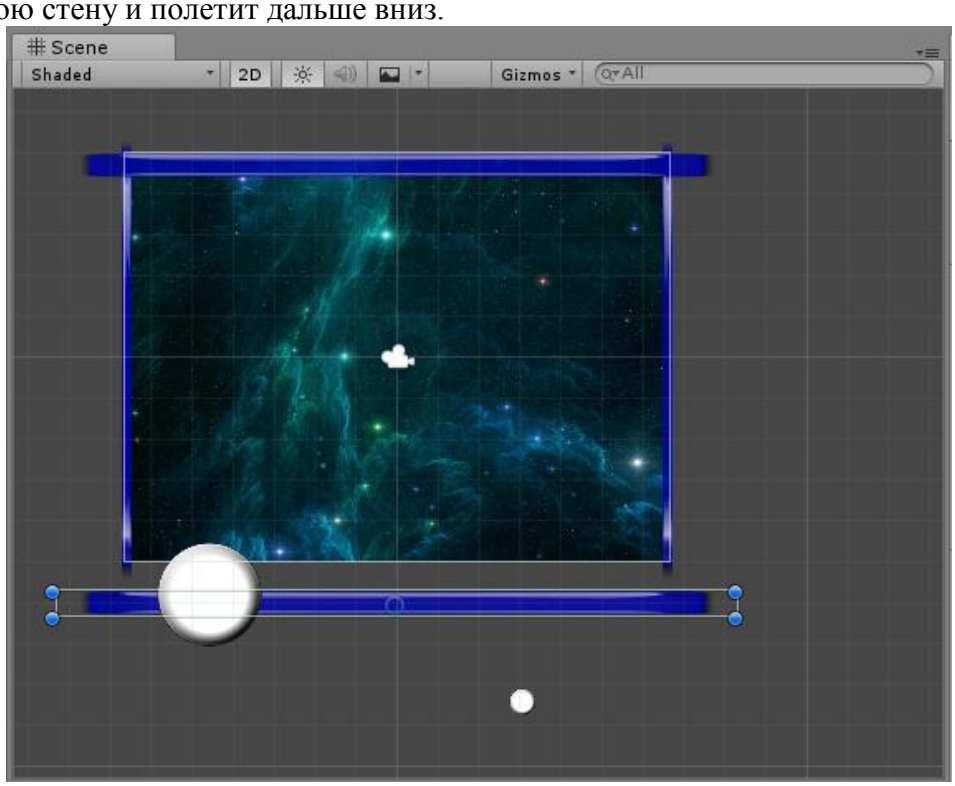
Теперь после запуска программы шарик будет двигаться вместе с ракеткой, пока не будет нажата левая кнопка мыши или левая клавиша Ctrl, после чего шарик перейдет в динамический режим и начнет летать по сцене, отражаясь от стен (в том числе и от невидимой нижней стены), а также от ракетки. Обратите внимание на то, что за счет выбора позиции соударения шарика и ракетки он будет отражаться от нее под различными углами.

Осталось реализовать исчезновение шарика при вылете за нижнюю границу сцены. Для этого выделите объект Bottom Wall и установите для него флажок Is Trigger в компоненте Box Collider 2D:



Тем самым мы изменили поведение коллайдера, связанного с нижней стеной. Теперь он будет вести себя не как «твердое тело» при соударении с другим объектом (обеспечивая отражение этого объекта), а как «датчик» касания, который будет срабатывать при попадании на него другого объекта.

Если теперь уменьшить масштаб сцены, чтобы было видно пространство под нижней стеной, запустить программу и дождаться падения шарика на нижнюю стену, то мы увидим, что шарик пролетит сквозь нижнюю стену и полетит дальше вниз.



Нам осталось написать код, который распознает столкновение шарика с нижней стеной и обеспечивает немедленное уничтожение шарика. Этот код выглядит как обработчик особого события OnTriggerEnter2D, который надо поместить в описание объекта BallScript:

```
void OnTriggerEnter2D(Collider2D other)
{
    Destroy(gameObject);
}
```

С помощью параметра other можно определить тот объект, который вызвал данное событие, но, поскольку свойством коллайдера-триггера обладает только нижняя стена, в анализе этого параметра нет необходимости.

Если сохранить измененный скрипт и опять запустить программу, то мы увидим, что как только шарик достигнет нижней стены, он немедленно исчезнет.

1.4.3 Добавление звуковых эффектов

При разработке игр в системе Unity очень легко снабдить их различными звуковыми эффектами, которые значительно повысят наглядность и привлекательность игры для пользователей. Добавим к нашей программе компоненты и программный код, обеспечивающий воспроизведение двух звуковых эффектов: при

ударе шарика о препятствие и при его исчезновении за нижней границей. Соответствующие звуковые файлы небольшого размера уже присутствуют в нашем проекте в каталоге Sounds: это hit (звук удара) и lose (звук исчезновения).

Для интеграции их в программу необходимо использовать компонент для воспроизведения звука. Поскольку в дальнейшем мы будем использовать и настраивать этот компонент не только в объекте Ball, но и в других объектах, целесообразно разместить его в каком-либо стандартном объекте нашей игры, доступ к которому является максимально простым. Хорошим кандидатом является объект Main Camera. В то же время, сами музыкальные клипы можно добавить непосредственно в скрипт BallScript, поскольку они будут использоваться только в нем.

Итак, вначале выберем объект Main Camera и добавим к нему новый компонент: Add Component | Audio | Audio Source.

Теперь вернемся к скрипту BallScript в редакторе MonoDevelop и добавим к нему новые фрагменты (метод Update не требует изменения, поэтому его не указываем):

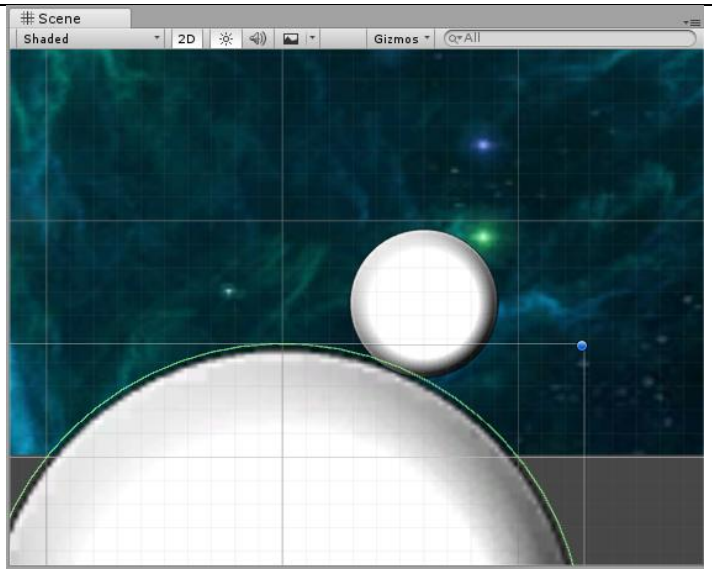
```
public Vector2 ballInitialForce;
Rigidbody2D rb;
GameObject playerObj;
float deltaX;
AudioSource audioSrc;
public AudioClip hitSound;
public AudioClip loseSound;

void Start()
{
    rb = GetComponent<Rigidbody2D>();
    playerObj = GameObject.FindGameObjectWithTag("Player");
    deltaX = transform.position.x;
    audioSrc = Camera.main.GetComponent<AudioSource>();
}

void OnTriggerEnter2D(Collider2D other)
{
    audioSrc.PlayOneShot(loseSound);
    Destroy(gameObject);
}

void OnCollisionEnter2D(Collision2D collision)
{
    audioSrc.PlayOneShot(hitSound);
}
```

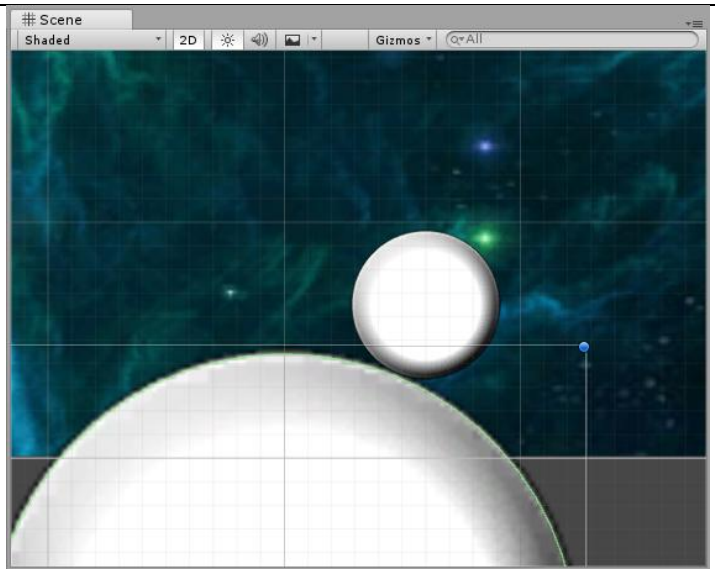
Обратите внимание на то, что при соприкосновении коллайдеров, для которых не установлено свойство Is Trigger, возникает событие OnCollisionEnter2D с параметром другого типа: Collision2D (из которого, впрочем, тоже можно извлечь информацию и о другом коллайдере, и о связанном с ним объекте).

После сохранения измененного скрипта и возврата в редактор Unity мы увидим, что вид скрипта BallScript изменился: в нем появились поля Hit Sound и Lose Sound, на которые надо перетащить звуковые клипы hit и lose из папки Sounds окна проекта.

Если теперь запустить программу, то при каждом соударении шарика со стеной или с ракеткой будет слышен щелчок, а при исчезновении шарика под сценой — звук разрушения шарика.

Недочет. Щелчок слышен и в начальный момент, когда шарик отрывается от ракетки. Это объясняется тем, что при указанных настройках положения шарика и ракетки их коллайдеры соприкасаются. Если увеличить фрагмент сцены с ракеткой и шариком, то можно заметить, что контур коллайдера для ракетки несколько больше, чем видимый белый контур самой ракетки (рисунок слева).





Поэтому можно немного уменьшить радиус коллайдера, связанного с ракеткой, например, с 0.64 до 0.62 (рисунок справа). Теперь при запуске программы щелчок в начальный момент не будет звучать.

1.5 Создание и настройка объектов-блоков: спрайты с текстом, работа с шаблонами (prefabs)

Игра «Арканоид» состоит в том, что шарик, управляемой ракеткой, разбивает блоки, расположенные на игровом поле, и тем самым зарабатывает очки. Для придания дополнительного интереса можно использовать блоки с различными свойствами, например, блоки, более или менее устойчивые к удару (более устойчивые разбиваются только после определенного числа ударов шарика и при этом приносят больше очков). Можно также предусмотреть особый вид блоков, ведущих себя подобно стенам, то есть не разрушающихся даже при большом числе ударов. Однако и для них можно предусмотреть «предел прочности», сделав так, чтобы они все же разрушались, например, после 30 ударов (хотя их разрушение и не является обязательным условием для перехода к следующему уровню).

Интересные дополнительные возможности связаны с падающими вниз «бонусами», которые появляются при разрушении блоков специального вида и активизируются в случае попадания на ракетку. Варианты бонусов могут быть очень разнообразными, причем как «хорошими» (дополнительные очки, дополнительные резервные шарики или шарики, сразу появляющиеся на игровом поле, замедление шариков, увеличение размеров ракетки, особые «огненные» шарики, разбивающие за один удар любые блоки, и т. д.), так и «плохими» (уменьшение размеров ракетки, ускорение шариков, «бомба», уничтожающая ракетку, отмена «огненных» шариков и т. д.). Кроме того, можно использовать блоки, движущиеся по определенной траектории. Однако эти возможности мы не будем реализовывать.

1.5.1 Создание объекта для блока и настройка его основных свойств

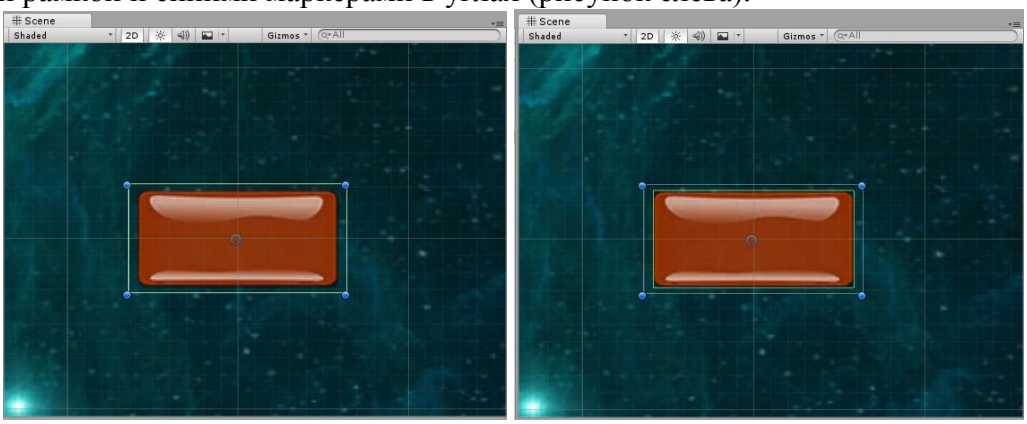
В наборе имеющихся изображений предусмотрены блоки четырех видов, различающиеся цветами. Свяжем с каждым цветом блоков особые свойства: пусть синие блоки (которые мы уже использовали при создании стен) будут особо прочными блоками, разрушающимися только после 30 ударов и приносящими 100 очков (хотя разрушать их необязательно), красные блоки потребуют для своего разрушения 4 ударов и дадут 40 очков, зеленые блоки потребуют 2 удара и дадут 20 очков, а желтые будут разрушаться после первого же удара и приносить по 10 очков. При желании все эти числовые характеристики можно будет изменять, причем в режиме дизайна, не внося изменения в программные скрипты.

Для большей наглядности будем изображать на блоках, требующих более чем одного удара для своего разрушения, число, показывающее, сколько ударов осталось нанести (исключением будут только особо прочные блоки, на которых эти данные не отображаются). Так как связывание с двумерными спрайтами текстовых данных является в Unity достаточно сложной задачей, начнем с разработки именно такого типа блоков.

Итак, добавим на сцену красный блок и настроим его свойства. Создадим в окне иерархии новый 2D-объект Sprite, переименуем его в Red Block, перетащим на свойство Sprite компонента Sprite Renderer рисунок cube_red из папки Sprites окна проекта и установим значение 1 для свойства Order in Layer того же компонента. Таким образом, блок будет отображаться перед фоном, «на одном уровне» со стенами и шариком (это не приведет к тому, что какой-то объект будет заслонять другой, так как блоки не будут перекрываться

ни со стенами, ни с шариком). Для большей наглядности разместим созданный объект чуть выше основной камеры, положив Position X = 0, Y = 1, Z = 0. Масштабировать блоки мы не будем.

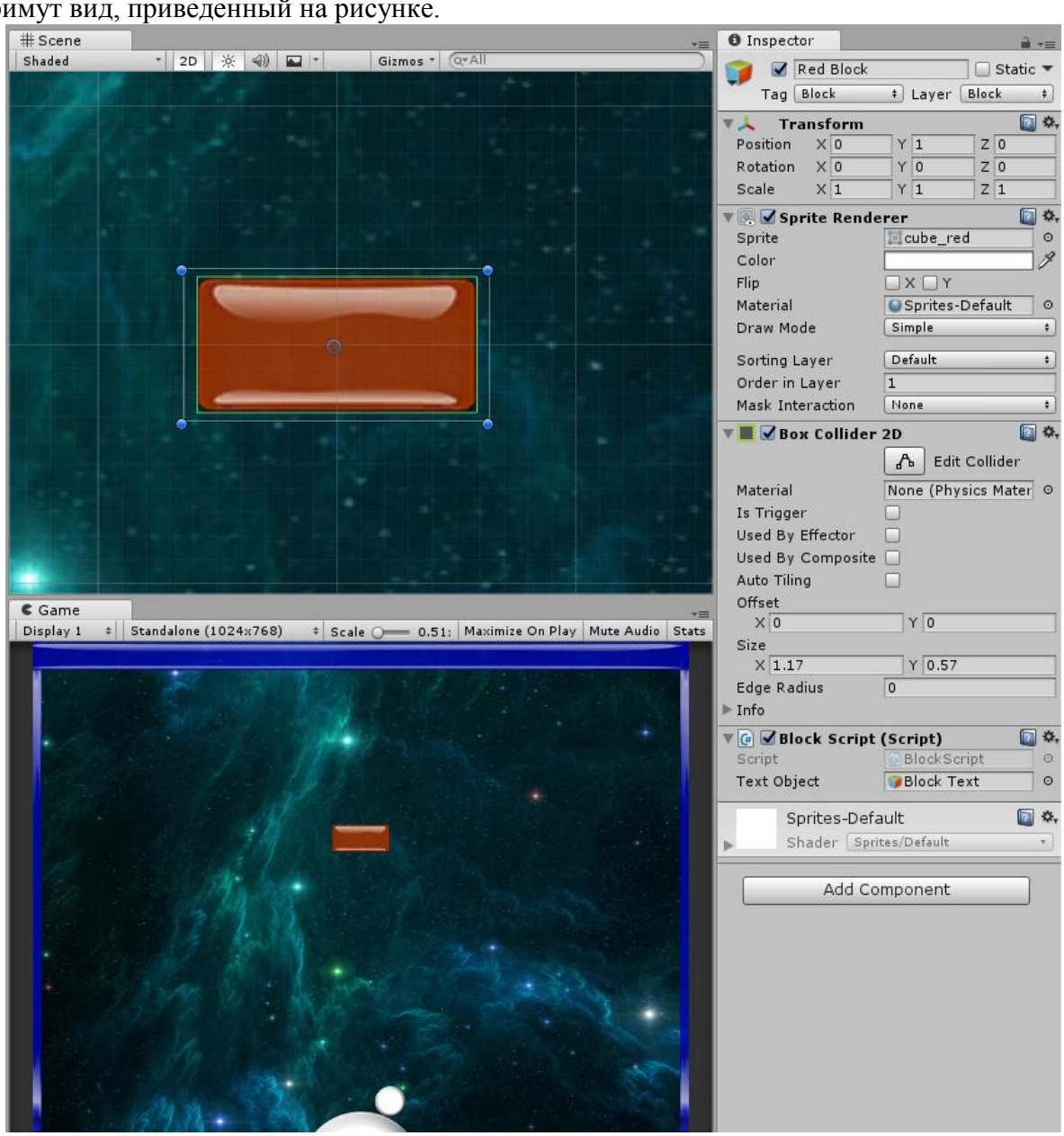
Кроме того, добавим к объекту Red Block прямоугольный коллайдер — компонент Physics 2D | Box Collider 2D; размеры коллайдера автоматически подстроятся под размеры блока. Впрочем, если увеличить масштаб отображения сцены, то можно заметить, что размеры коллайдера (зеленая рамка) всё же несколько превышают размеры видимой части спрайта, хотя и совпадают с размером изображения, помеченного синей рамкой и синими маркерами в углах (рисунок слева).



Чтобы это исправить, откорректируем размеры коллайдера, изменив в компоненте Box Collider 2D значение Size X с 1.28 на 1.17, а значение Size Y с 0.64 на 0.57 (рисунок справа).

Наконец, свяжем с созданным объектом новую метку Block и новый слой с этим же именем.

После выполнения всех описанных настроек сцена и окно инспектора для нового объекта Red Block примут вид, приведенный на рисунке.



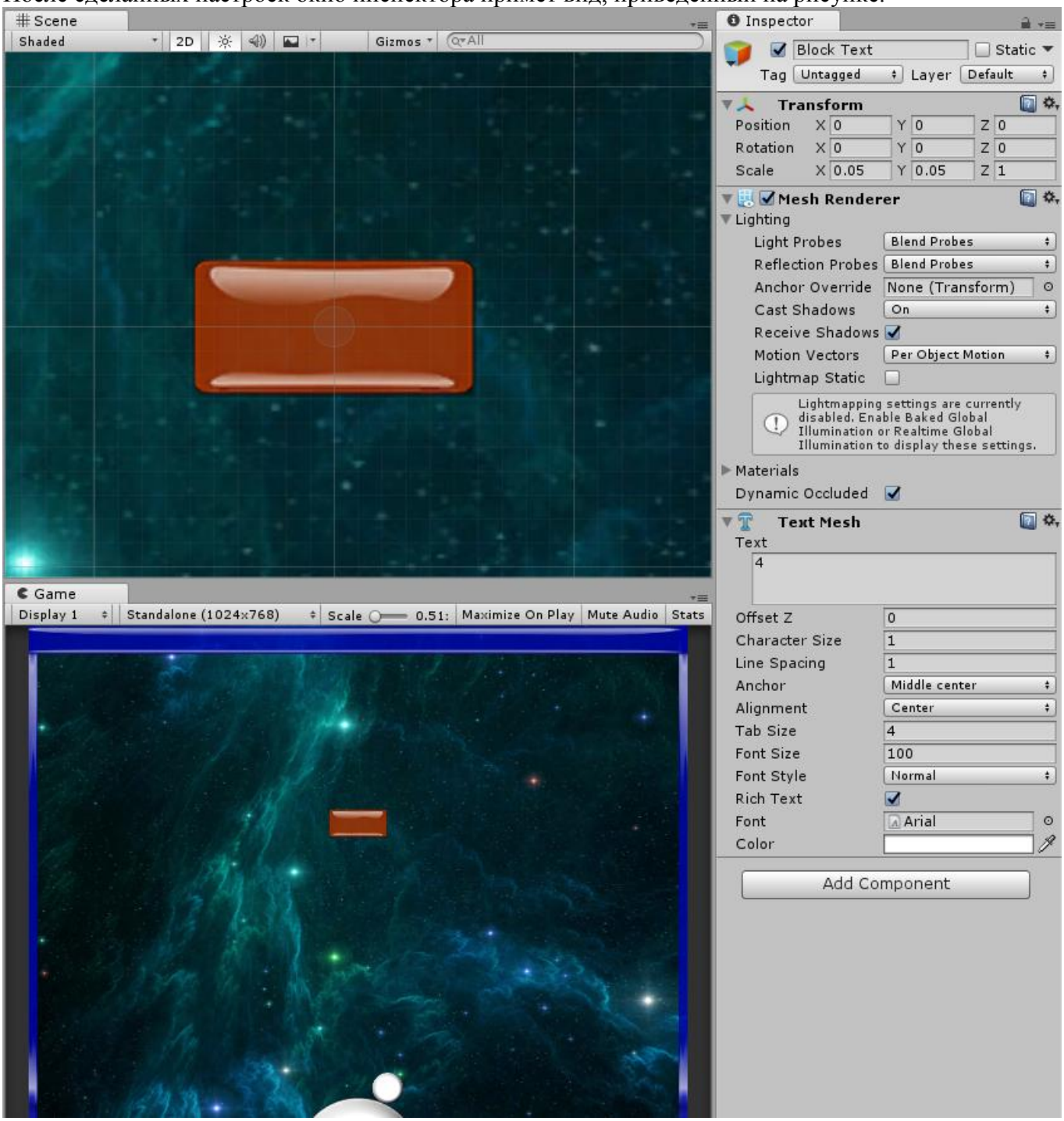
1.5.2 Добавление к блоку текста

Теперь надо поместить на данный объект текст «4» — количество ударов, требующихся для разрушения блока. К сожалению, в набор двумерных объектов не входит объект «Текст», хотя для трехмерного случая такой объект предусмотрен: это 3D Text из группы 3D Object. Как правило, трехмерные объекты в двумерных играх не используются, однако трехмерный текст вполне можно поместить на двумерный спрайт, правда, отобразить сам текст при этом удастся только при запуске игры, а некоторые важные свойства объекта 3D Text можно будет установить только программным способом.

Объект 3D Text сделаем дочерним объектом по отношению к объекту Red Block. Для этого выделим объект Red Block в окне иерархии, вызовем его контекстное меню и в этом меню выполним команду 3D Object | 3D Text. Перейдя на созданный объект (с именем New Text), изменим его имя на Block Text и зададим новые значения для некоторых свойств компонента Transform и Text Mesh.

Для компонента Transform положим значения Scale равными X = 0.05, Y = 0.05, для компонента Text Mesh изменим свойство Text на 4 (хотя это и необязательно, так как значение этого свойства будет настраиваться программно), свойство Anchor положим равным Middle center, свойство Alignment — равным Center, а свойство Font Size — равным 100. Значения метки и слоя оставим стандартными: Untagged и Default соответственно.

После сделанных настроек окно инспектора примет вид, приведенный на рисунке.

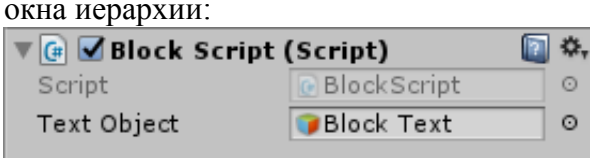


Как отмечалось ранее, мы не увидим требуемый текст на блоке, размещенном на сцене. Чтобы он появился при запуске программы, нам необходимо создать скрипт, содержащий настройки, которые невозможно установить в режиме редактора.

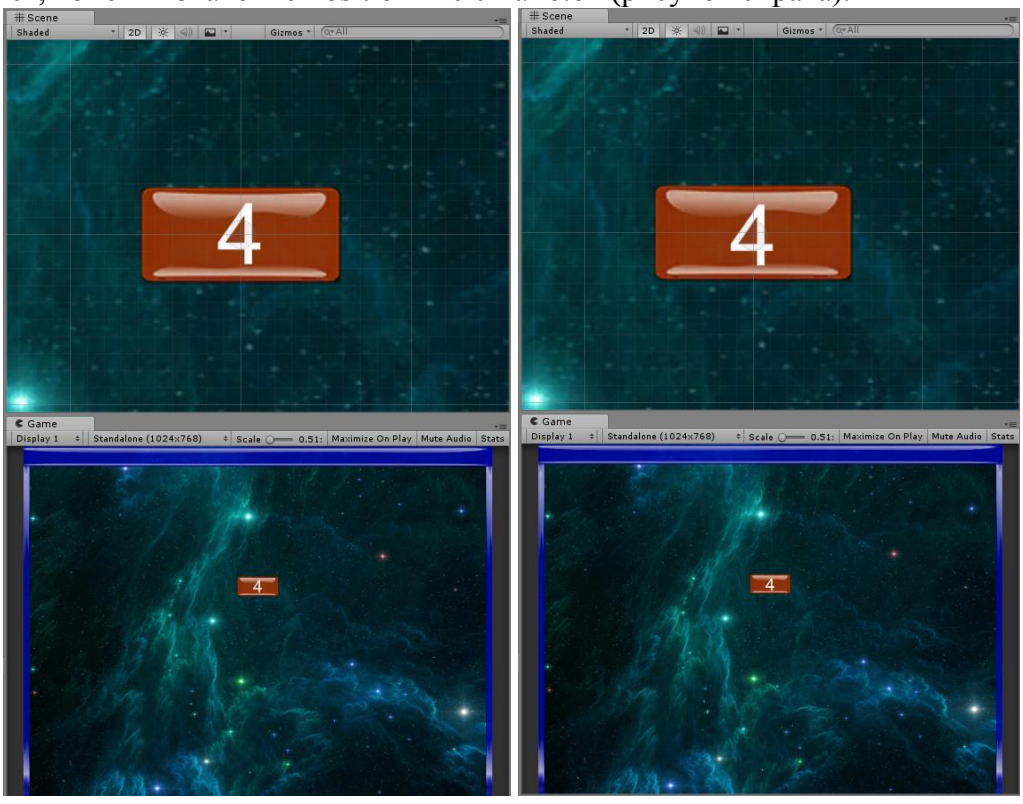
Создадим новый скрипт с именем BlockScript и внесем в него следующие изменения:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BlockScript : MonoBehaviour
{
    public GameObject textObject;
    TextMesh textMesh;

    void Start()
    {
        if (textObject != null)
        {
            textObject.GetComponent<Renderer>().sortingLayerName = "Default";
            textObject.GetComponent<Renderer>().sortingOrder = 2;
            textMesh = textObject.GetComponent<TextMesh>();
            textMesh.text = "4";
        }
    }
}
```

Сохраните этот скрипт, вернитесь в редактор Unity и перейдите на объект Red Block в окне иерархии. Перетащите скрипт BlockScript в окно инспектора для данного объекта. При этом в появившемся разделе Block Script (Script) дополнительно появится поле Text Object, в которое надо перетащить объект Block Text из окна иерархии:



Теперь после запуска программы мы увидим текст на блоке (рисунок слева). Можно немного опустить текст, изменив значение Position Y с 0 на -0.02 (рисунок справа).



1.5.3 Завершающие корректировки скрипта, связанные с настройкой характеристик блока

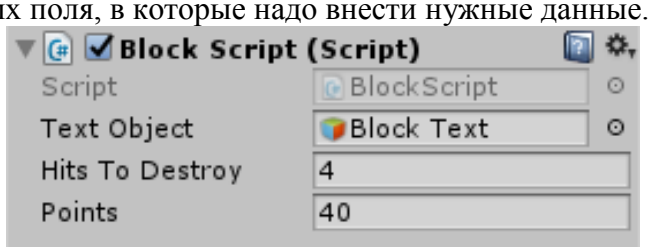
Дополним скрипт BlockScript, добавив открытые поля, определяющие требуемое число ударов (hitsToDestroy) и количество очков, получаемых за разрушение блока (points). При каждом ударе шарика о блок будем уменьшать значение поля hitsToDestroy и корректировать текст на блоке. Эти действия будут выполняться в обработчике OnCollisionEnter2D (аналогичный обработчик мы использовали для объекта Ball для генерации звука удара). Если значение hitsToDestroy станет равным 0, то блок будет уничтожен. Пока мы никак не будем обрабатывать поле points, поскольку мы еще не реализовали в игре подсчет очков.

Получаем следующий вариант скрипта.

```
public class BlockScript : MonoBehaviour
{
    public GameObject textObject;
    TextMesh textMesh;
    public int hitsToDestroy;
    public int points;

    void Start()
    {
        if (textObject != null)
        {
            textObject.GetComponent<Renderer>().sortingLayerName = "Default";
            textObject.GetComponent<Renderer>().sortingOrder = 2;
            textMesh = textObject.GetComponent<TextMesh>();
            textMesh.text = hitsToDestroy.ToString();
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        {
            hitsToDestroy--;
            if (hitsToDestroy == 0)
                Destroy(gameObject);
            else if (textMesh != null)
                textMesh.text = hitsToDestroy.ToString();
        }
    }
}
```

Мы также откорректировали последний оператор в методе Start, чтобы начальный текст соответствовал начальному значению поля hitsToDestroy. Обратите внимание на то, что действия с полями textObject и textMesh выполняются только при наличии текстового объекта, присоединенного к объекту-блоку. Это позволит использовать данный скрипт и для тех вариантов блоков, для которых не нужно будет выводить текстовую информацию.

После сохранения нового варианта скрипта его вид в окне инспектора изменится: в нем появятся два новых поля, в которые надо внести нужные данные.

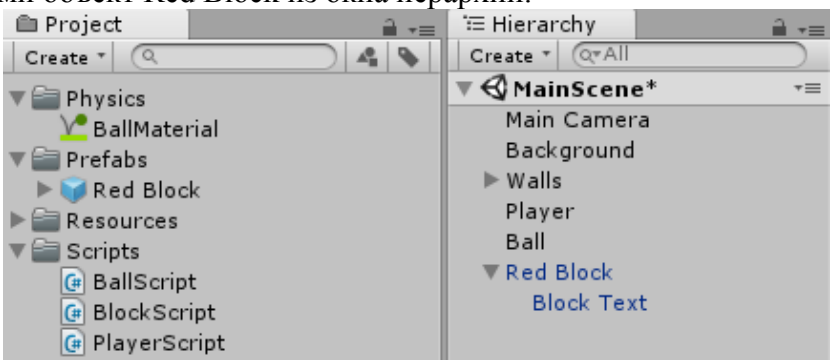


Теперь при запуске программы после каждого удара шарика о блок число на блоке будет уменьшаться, а при достижении требуемого числа ударов блок исчезнет.

1.5.4 Оформление объекта как шаблона (prefab)

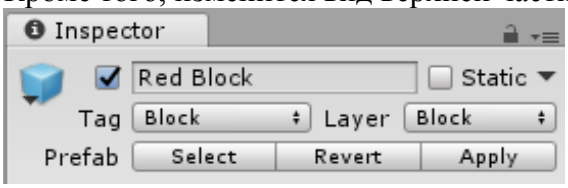
Разработанный нами объект Red Block может использоваться для создания большого числа блоков на различных уровнях игры. Эти экземпляры блоков можно создавать как в режиме редактора Unity, так и программным путем; особенно удобно это делать, если оформить объект Red Block как шаблон (в системе Unity для таких шаблонов используется термин prefab).

Чтобы создать шаблон достаточно добавить в окно проекта папку Prefabs и перетащить в нее созданный нами объект Red Block из окна иерархии.

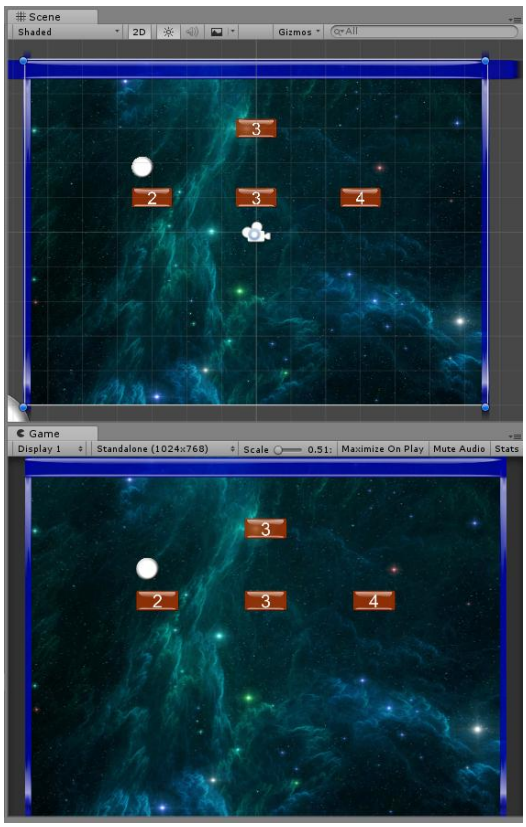


Можно заметить, что после выполнения этого действия объект Red Block в окне иерархии (и его дочерний объект BlockText) изменит цвет на синий. Это означает, что данный объект является копией шаблона, и если в шаблоне будут сделаны изменения, то они автоматически будут перенесены на объекты окна иерархии, связанные с этим шаблоном. Например, если в шаблоне изменить поле points, то аналогичным образом изменится и это поле в объекте в окне иерархии. В то же время, если изменить свойство в объекте окна иерархии, то это не повлечет изменения шаблона, причем в этом случае измененное значение свойства в объекте окна иерархии будет выделено полужирным шрифтом (признак того, что значение отличается от аналогичного значения для шаблона). Если в этом случае изменить данное свойство в шаблоне, то автоматического изменения свойства в объекте окна иерархии не произойдет.

Кроме того, изменится вид верхней части описания объекта в окне инспектора.



Теперь в нем появятся три дополнительные кнопки. Нажатие на первую из них (Select) немедленно выделяет в окне проекта шаблон, связанный с данным объектом. Нажатие на вторую (Revert) отменяет все изменения, которые были сделаны в объекте по сравнению с шаблоном (не изменяются только данные в компоненте Transform, связанные с позицией объекта). Наконец, кнопка Apply позволяет выполнить обратное действие: изменить шаблон в соответствии с текущими настройками объекта (причем изменятся и данные, связанные с позицией объекта).



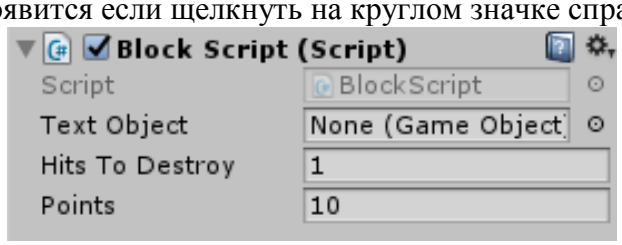
Мы можем перетащить из папки Prefabs созданный шаблон в окно иерархии несколько раз, и при этом на сцене будут созданы новые экземпляры данного шаблона. Останется лишь изменить их положение. При запуске программы они будут вести себя совершенно одинаковым образом.

Можно поступить и по-другому: полностью удалить объекты Red Block со сцены и создать их экземпляры программно, в момент начала игры (или загрузки очередного уровня). Именно так мы и поступим в дальнейшем.

Корректируя свойства одного из добавленных блоков, мы можем создать другой вариант блока, который тоже можно оформить в виде шаблона. Проделаем это на примере создания зеленого блока. Выберем один из блоков, созданных на основе шаблона Red Block (например, левый), и изменим три его свойства: перетащим в поле Sprite компонента Sprite Renderer рисунок cube_green из папки Sprites окна проекта, а в компоненте Block Script изменим значение поля Hits To Destroy на 2, а значение по-

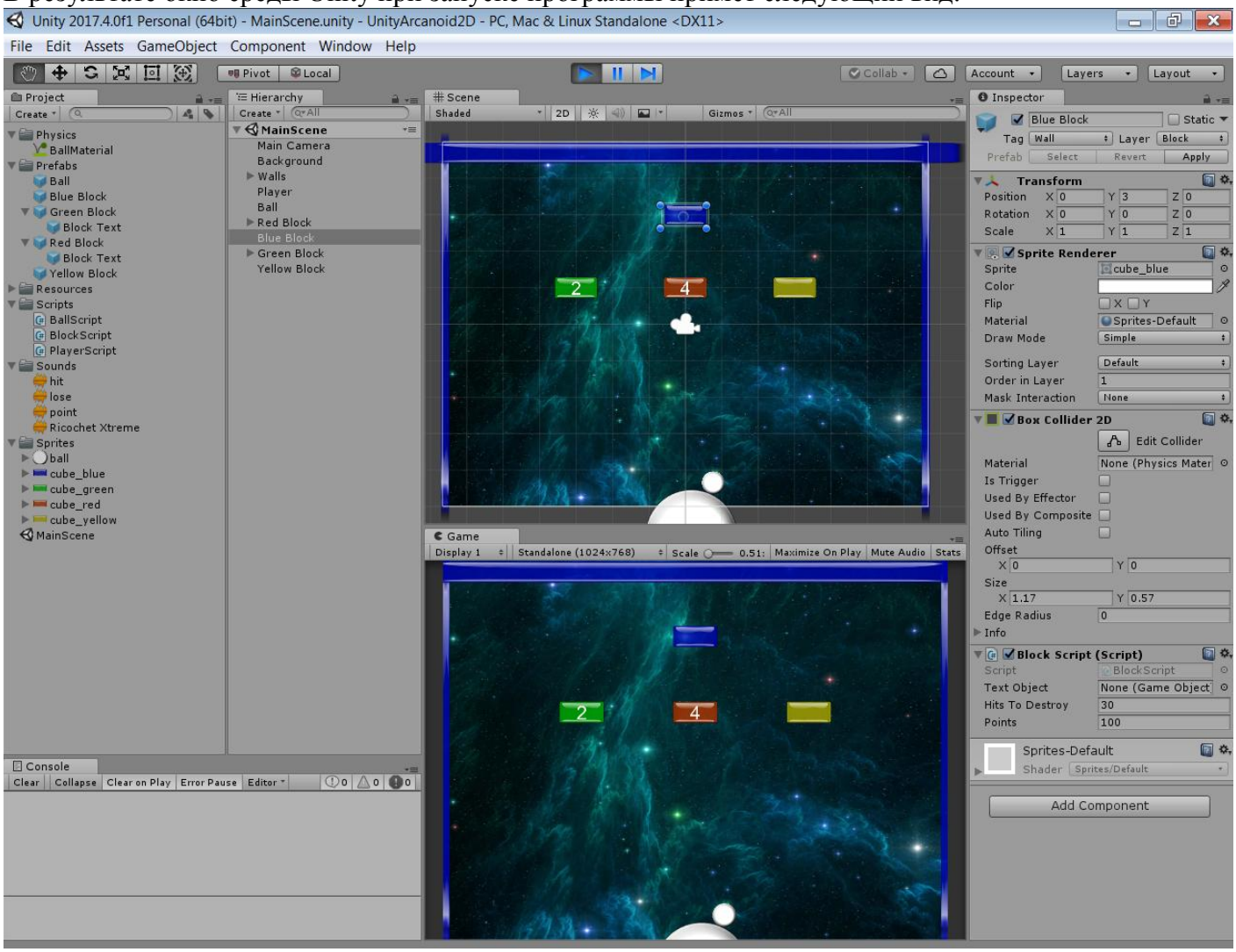
ля Points на 20. Кроме того, переименуем этот объект в окне иерархии, назвав его Green Block. Осталось перетащить этот объект в папку Prefabs окна проекта.

Аналогичными действиями создадим шаблон для желтого блока. Для этого изменим свойства, например, правого блока сцены (имя — Yellow Block, рисунок cube _yellow, Hits To Destroy = 1, Points = 10). В данном случае выводить текст «1» на блоке не нужно, поэтому перед созданием шаблона удалим дочерний объект Block Text, а в свойстве Text Object компонента Block Script выберем вариант None (из списка, который появится если щелкнуть на круглом значке справа от этого свойства).



Те же действия выполним для создания шаблона для синего блока, настроив верхний образец блока на сцене (имя — Blue Block, рисунок cube _blue, Hits To Destroy = 30, Points = 100). Для этого шаблона тоже надо удалить дочерний объект и положить поле Text Object равным None. Кроме того, для синего блока мы изменим метку, положив ее равной Wall.

В результате окно среды Unity при запуске программы примет следующий вид:



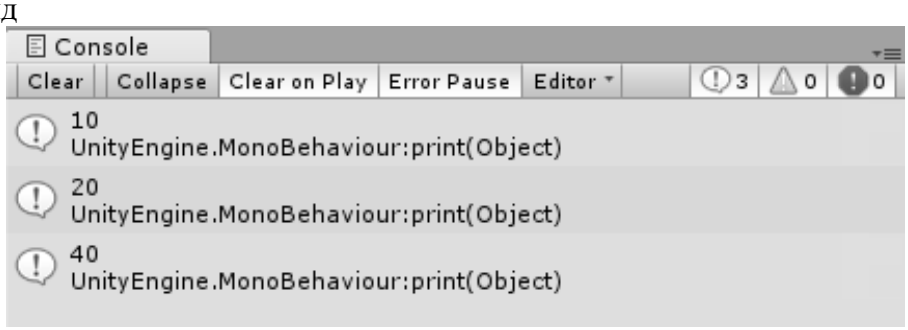
Можно проверить, что для разрушения зеленого блока требуется два удара, желтого блока — один удар, а синий блок будет оставаться на месте после достаточно большого количества ударов, то есть играть роль «внутреннего участка стены» (хотя после 30 ударов и он разрушится).

Чтобы дополнительно проверить, что при разрушении блоков будет начисляться требуемое число очков, можно просто выводить это число в консольном окне. Для этого достаточно внести следующее изменение в метод OnCollisionEnter2D скрипта BlockScript:

```
    void OnCollisionEnter2D(Collision2D collision)
    {
        {
            hitsToDestroy--;
```

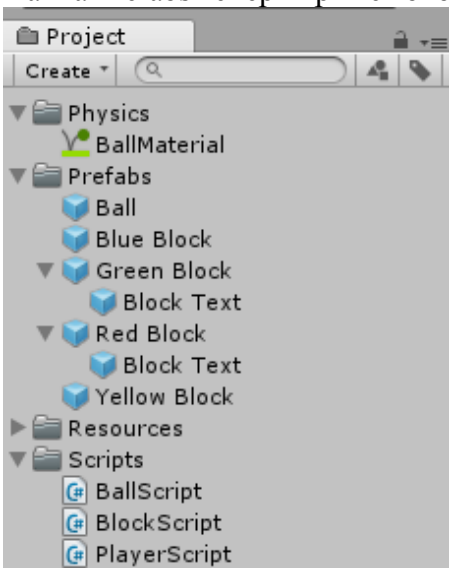
```
            if (hitsToDestroy == 0)
            {
                print(points);
                Destroy(gameObject);
            }
            else if (textMesh != null)
                textMesh.text = hitsToDestroy.ToString();
        }
    }
```

Если теперь последовательно разрушить желтый, зеленый и красный блок, то консольное окно примет вид



Чтобы завершить раздел, связанный с созданием шаблонов, создадим также шаблон для шарика. Для этого достаточно перетащить объект Ball из окна иерархии в папку Prefabs окна проекта.

Папка Prefabs теперь примет окончательный вид:



1.6 Генерация уровней игры

Применим разработанные шаблоны для генерации различных уровней игры. При разработке игр можно либо создавать каждый новый уровень в виде отдельной сцены, используя средства редактора Unity, либо программно генерировать компоненты каждого уровня на основе единственной сцены. Воспользуемся вторым подходом.

Каждый уровень игры будет различаться составом и расположением блоков, фоновым рисунком, а также начальной скоростью шариков (чем выше уровень, тем начальная скорость больше). С увеличением уровня будет также увеличиваться количество блоков. Мы реализуем простейший вариант генерации блоков, при котором требуемое количество блоков различного типа будет случайным образом располагаться в верхней половине игровой сцены (хотя в дальнейшем можно предусмотреть вариант загрузки каждой конфигурации блоков из специальных настроечных файлов, а для подготовки таких файлов можно разработать конструктор уровней). При активации каждого уровня будем добавлять на ракетку не один, а два шарика, чтобы, с одной стороны, усложнить управление ими, а с другой — сразу реализовать вариант игровой ситуации, когда число шариков на сцене может быть больше одного. Пока при потере всех шариков будем просто восстанавливать оба шарика, а при разбивании всех блоков — выводить сообщение о прохождении уровня в консольном окне (управление уровнями игры будет реализовано в следующем разделе).

Ограничим число уровней значением 30 (по количество имеющихся фоновых изображений).

1.6.1 Создание новых игровых объектов на основе шаблонов

При создании нового уровня мы будем заново генерировать на сцене все игровые объекты, кроме ракетки и стен (используя имеющиеся шаблоны), поэтому удалите со сцены расположенные на ней блоки, а также шарик.

Добавьте следующий код в описание класса PlayerScript:

```
const int maxLevel = 30;

[Range(1, maxLevel)]
public int level = 1;
public float ballVelocityMult = 0.02f;
public GameObject bluePrefab;
public GameObject redPrefab;
public GameObject greenPrefab;
public GameObject yellowPrefab;
public GameObject ballPrefab;

void CreateBlocks(GameObject prefab, float xMax, float yMax, int count, int maxCount)
{
    if (count > maxCount)
        count = maxCount;
    for (int i = 0; i < count; i++)
        Instantiate(prefab,
            new Vector3((Random.value * 2 - 1) * xMax, Random.value * yMax, 0),
            Quaternion.identity);
}

void CreateBalls()
{
    int count = 2;
    for (int i = 0; i < count; i++)
    {
        var obj = Instantiate(ballPrefab);
        var ball = obj.GetComponent<BallScript>();
        ball.ballInitialForce += new Vector2(10 * i, 0);
        ball.ballInitialForce *= 1 + level * ballVelocityMult;
    }
}

void StartLevel()
{
    var yMax = Camera.main.orthographicSize * 0.8f;
    var xMax = Camera.main.orthographicSize * Camera.main.aspect * 0.85f;
    CreateBlocks(bluePrefab, xMax, yMax, level, 8);
    CreateBlocks(redPrefab, xMax, yMax, 1 + level, 10);
    CreateBlocks(greenPrefab, xMax, yMax, 1 + level, 12);
    CreateBlocks(yellowPrefab, xMax, yMax, 2 + level, 15);
    CreateBalls();
}

void Start()
```

```
    {
        Cursor.visible = false;
        StartLevel();
    }
```

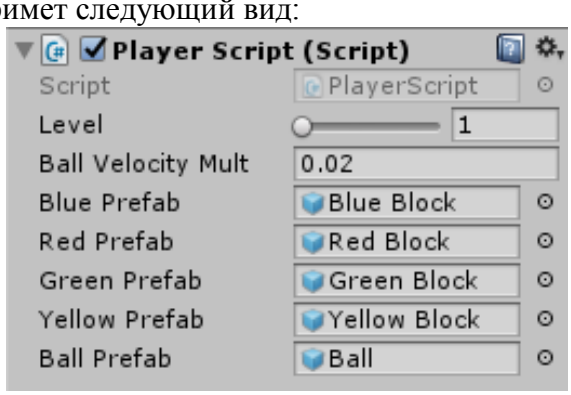
Все действия по созданию нового уровня мы реализовали в функции StartLevel, которая вызывается в методе Start. В этой функции вначале определяется размер области, в которой будут генерироваться блоки (эта информация определяется на основе данных, получаемых от главной камеры). Затем создается требуемое число блоков каждого вида (это число зависит от номера уровня, а также не может превосходить указанного максимального значения). В конце создаются два шарика.

Для создания объекта на основе шаблона предусмотрен метод Instantiate. В нем можно явно указать новое положение объекта или использовать позицию, указанную для шаблона (первый вариант мы применили для блоков, второй — для шариков).

После создания шарика мы дважды модифицируем начальную силу, которая на него действует: первая модификация обеспечивает небольшое изменение траектории для второго шарика, а вторая ускоряет движение шариков в зависимости от номера уровня.

Чтобы упростить тестирование уровней, мы описали поля level и ballVelocityMult как открытые; это позволит настраивать их значения непосредственно в редакторе Unity. Поле level снабжено атрибутом, который обеспечивает отображение в редакторе не только текстового поля для ввода значения, но и ползунка. Благодаря этому в качестве значения можно указать только целые числа в диапазоне от 1 до константы maxLevel, равной 30.

В виде открытых полей также представлены используемые шаблоны. Перед запуском программы на каждое из этих полей в редакторе надо перетащить нужный шаблон. В результате компонент Player Script примет следующий вид:



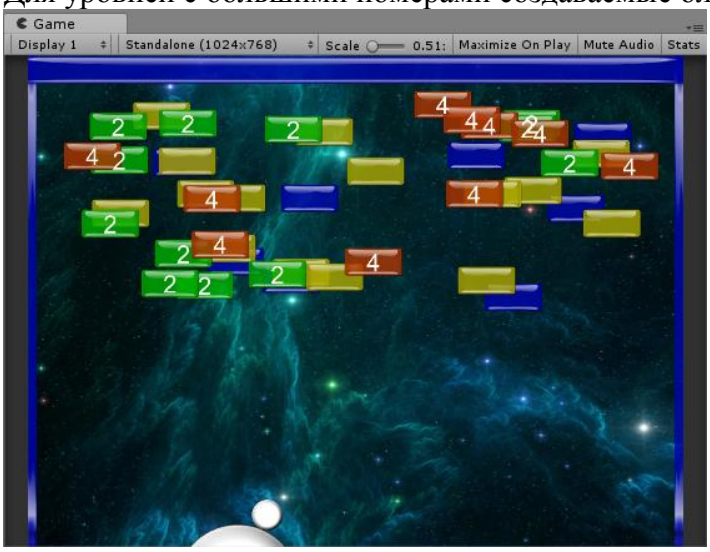
Установив значение уровня Level равным 30 и запустив программу, можно проверить величину максимального увеличения скорости шарика и при необходимости откорректировать ее, увеличив или уменьшив значение поля Ball Velocity Mult.

1.6.2 Отключение взаимодействия шариков

При запуске программы можно обнаружить, что шарики тоже взаимодействуют между собой (отражаются друг от друга). Обычно в игре Arcanoid шарики не взаимодействуют, поэтому в нашей игре мы тоже отключим эту возможность. Для этого достаточно внести дополнительные изменения в матрицу взаимодействия слоев, сняв в ней флажок на пересечении пары слоев Ball – Ball (напомним, что отобразить матрицу в окне инспектора можно с помощью команды Edit | Project Settings | Physics 2D).

1.6.3 Размещение блоков без наложений

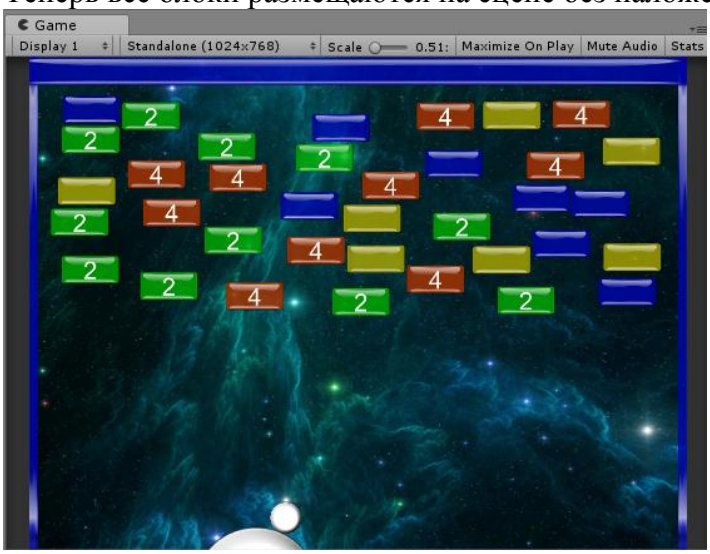
Для уровней с большими номерами создаваемые блоки будут часто накладываться друг на друга.



Чтобы это исправить, добавим с описание класса PlayerScript два вспомогательных поля и откорректируем метод CreateBlocks:

```
static Collider2D[] colliders = new Collider2D[50];
static ContactFilter2D contactFilter = new ContactFilter2D();

void CreateBlocks(GameObject prefab, float xMax, float yMax, int count, int maxCount)
{
    if (count > maxCount)
        count = maxCount;
    for (int i = 0; i < count; i++)
        for (int k = 0; k < 20; k++)
        {
            var obj = Instantiate(prefab,
                new Vector3((Random.value * 2 - 1) * xMax, Random.value * yMax, 0),
                Quaternion.identity);
            if (obj.GetComponent<Collider2D>()
                  .OverlapCollider(contactFilter.NoFilter(), colliders) == 0)
                break;
            Destroy(obj);
        }
}
```

Теперь все блоки размещаются на сцене без наложений.



Мы использовали метод OverlapCollider, позволяющий найти для любого коллайдера все коллайдеры, которые пересекаются с ним. Найденные коллайдеры заносятся в массив, который надо создать перед вызовом метода и указать в качестве последнего параметра. Чтобы не создавать такой массив многократно, мы описали его как статическое поле класса PlayerScript (таким образом, этот массив на протяжении всей игры будет существовать в единственном экземпляре). Другой вспомогательный параметр метода OverlapCollider мы тоже описали как статическое поле класса.

Поскольку возможна ситуация, когда при большом числе уже имеющихся блоков новый блок не удастся добавить на сцену без наложений, мы предусмотрели максимум 20 попыток для добавления нового блока. Если все 20 попыток оказались неудачными, то новый блок не добавляется.

1.6.4 Обновление фона: использование ресурсов

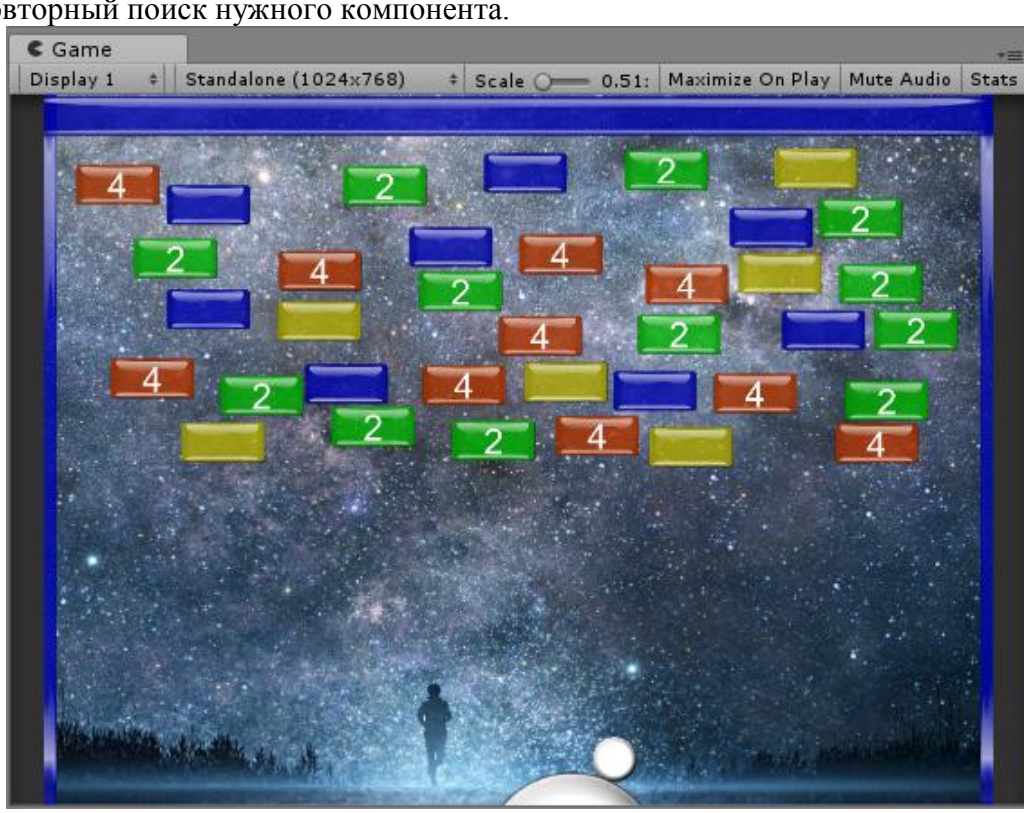
Для завершения создания уровня нам осталось обновить фон сцены в соответствии с номером уровня. Поскольку мы разместили все фоновые изображения в папке Resources и снабдили их порядковыми номерами от 1 до 30, для их загрузки достаточно использовать метод Resources.Load.

Опишите в классе PlayerScript метод SetBackground и добавьте его вызов в начало метода StartLevel:

```
void SetBackground()
{
    var bg = GameObject.Find("Background").GetComponent<SpriteRenderer>();
    bg.sprite = Resources.Load(level.ToString("d2"), typeof(Sprite)) as Sprite;
}

void StartLevel()
{
    SetBackground();
    ...
}
```

При преобразовании номера уровня в строку мы использовали форматную настройку d2, которая дополняет целое число слева нулями до двух цифр.

Вызов метода SetBackground в начале метода StartLevel ускоряет поиск компонента Background, поскольку на данном этапе сцена содержит только фон, стены и ракетку. Можно было бы описать объект bg в качестве поля класса PlayerScript и один раз задать его значение в методе Start, но это нецелесообразно, так как в других частях программного кода этот объект не будет использоваться, а метод SetBackground будет вызываться достаточно редко, и поэтому вполне допустимо, чтобы при каждом его вызове производился повторный поиск нужного компонента.



1.6.5 Восстановление шариков при их потере: сопрограммы

Если шарик улетает за нижнюю границу, то он уничтожается, поэтому для отслеживания ситуации, при которой потеряны все шарики, достаточно проверить, что на сцене отсутствуют объекты с меткой Ball. В этом случае надо создать шарики заново:

```
if (FindGameObjectsWithTag("Ball").Length == 0)
    CreateBalls();
```

Где разместить данный фрагмент? Естественно включить его в «главный» объект игры, то есть в класс PlayerScript. Простейшим вариантом было бы помещение этого фрагмента в метод Update; это гарантирует, что уже в первом кадре, в котором будут отсутствовать все шарики, они создадутся заново. Однако такой вариант является неэффективным, поскольку он требует достаточно большого объема вычислений при обновлении каждого кадра, тогда как анализируемое событие (исчезновение всех шариков) будет происходить редко.

Поэтому лучше связать требуемые действия с ситуацией, когда исчезает один из шариков. Эту ситуацию легко распознать в объекте BallScript, поскольку она связана с его событием OnTriggerEnter2D. Итак, опишем в классе PlayerScript метод BallDestroyed и вызовем его в методе OnTriggerEnter2D класса BallScript:

```
// новый метод класса PlayerScript
    public void BallDestroyed()
    {
        if (FindGameObjectsWithTag("Ball").Length == 0)
            CreateBalls();
    }

// измененный метод класса BallScript
    void OnTriggerEnter2D(Collider2D other)
    {
        audioSrc.PlayOneShot(loseSound);
        Destroy(gameObject);
        playerObj.GetComponent<PlayerScript>().BallDestroyed();
    }
```

Однако при запуске нового варианта игры мы обнаружим, что в случае исчезновения всех шариков новые шарики не создаются. Добавим оператор отладочной печати в метод BallDestroyed:

```
print(FindGameObjectsWithTag("Ball").Length);
```

Запустив игру повторно, мы можем убедиться, что при потере шарика метод BallDestroyed вызывается, но возвращаемое в нем число шариков все еще учитывает только что потерянный шарик (при потере первого шарика выводится число 2, а при потере второго — число 1). Это связано с тем, что вызов метода Destroy не приводит к немедленному удалению объекта со сцены (хотя удаление и происходит очень быстро).

Одним из вариантов решения выявленной проблемы является «отложенное» определение текущего числа шариков (например, через 0.1 с после исчезновения очередного шарика). Для реализации отложенных действий в Unity предусмотрено несколько механизмов, одним из которых являются сопрограммы.

При выполнении сопрограммы можно указать в ней требуемую паузу, которая приостановит выполнение этой сопрограммы, не приостанавливая выполнение всей игры. В нашем случае сопрограмму надо запускать из объекта PlayerScript, так как если ее запустить из объекта BallScript, она будет немедленно завершена при разрушении этого объекта, и поэтому не успеет выполнить требуемые действия.

Итак, опишем вспомогательную сопрограмму BallDestroyedCoroutine в классе PlayerObject, переместив в нее действия, ранее указанные в методе BallDestroyed, а в методе BallDestroyed просто вызовем эту сопрограмму:

```
IEnumerator BallDestroyedCoroutine()
{
    yield return new WaitForSeconds(0.1f);
    if (FindGameObjectsWithTag("Ball").Length == 0)
        CreateBalls();
}

public void BallDestroyed()
```

```
{
    StartCoroutine(BallDestroyedCoroutine());
}
```

Теперь при исчезновении всех шариков на ракетке автоматически появляются два новых шарика, для запуска которых, как обычно, надо щелкнуть левой кнопкой мыши или нажать левую клавишу Ctrl.

1.6.6 Проверка прохождения уровня

Уровень считается пройденным, если разрушены все блоки (за исключением особо прочных синих блоков, которые разрушать необязательно). Заметим, что все блоки, кроме синих, имеют метку Block, поэтому для проверки прохождения уровня достаточно проанализировать число блоков, оставшихся на сцене, причем этот анализ можно проводить лишь в момент разрушения какого-либо блока. Таким образом, ситуация очень похожа на ранее рассмотренную ситуацию с анализом числа оставшихся шариков.

Опишем с классе PlayerScript новый метод BlockDestroyed и связанную с ним сопрограмму BlockDestroyedCoroutine. Предусмотрим в методе BlockDestroyed параметр points, в котором будет указываться число очков, полученных в результате разрушения блока (этот параметр мы используем впоследствии, когда будем реализовывать систему подсчета очков). Факт прохождения уровня будем пока отмечать лишь выводом соответствующего сообщения в консольном окне.

```
IEnumerator BlockDestroyedCoroutine()
{
    yield return new WaitForSeconds(0.1f);
    if (GameObject.FindGameObjectsWithTag("Block").Length == 0)
        print("Level complete!");
}

public void BlockDestroyed(int points)
{
    StartCoroutine(BlockDestroyedCoroutine());
}
```

Метод BlockDestroyed будет вызываться из метода OnCollisionEnter2D класса BlockScript. Для того чтобы из этого метода можно было обратиться к методам класса PlayerScript, добавим к классу BlockScript поле playerScript и инициализируем его в конце метода Start:

```
PlayerScript playerScript;

void Start()
{
    ...
    playerScript = GameObject.FindGameObjectWithTag("Player")
       .GetComponent<PlayerScript>();
}

void OnCollisionEnter2D(Collision2D collision)
{
    {
        hitsToDestroy--;
        if (hitsToDestroy == 0)
        {
            Destroy(gameObject);
            playerScript.BlockDestroyed(points);
        }
        else if (textMesh != null)
            textMesh.text = hitsToDestroy.ToString();
    }
}
```

1.7 Хранение общих данных игры и переключение между уровнями

Смена уровня в игре обычно сопровождается загрузкой новой сцены; при этом старая сцена разрушается вместе со всеми находящимися на ней объектами (хотя у класса GameObject предусмотрен метод DontDestroyOnLoad(obj), который отменяет разрушение объекта obj при смене сцены). В то же время для каждой игры имеется набор данных, который должен сохраняться при смене уровней (в этот набор, в частности, может входить текущий номер уровня). Этот набор можно хранить в виде статических полей какого-либо скрипта, поскольку статические поля класса сохраняются при разрушении его экземпляров. Однако среда Unity предоставляет альтернативный механизм для создания «постоянных» объектов, существующих не только на протяжении всей игры, но даже после ее завершения (при условии ее запуска из редактора Unity). Это, в частности, упрощает отладку игры, так как дает возможность явной настройки ее параметров в редакторе Unity непосредственно перед запуском. Отмеченная возможность обеспечивается с помощью особых объектов, являющихся потомками класса ScriptableObject. Проиллюстрируем ее на примере нашей игры.

1.7.1 Создание объекта типа ScriptableObject и его подключение к игровому объекту

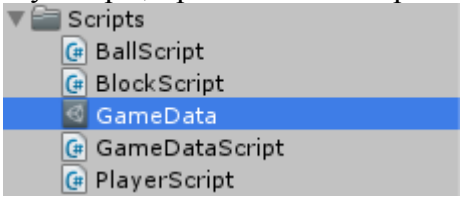
Создайте новый скрипт с именем GameDataScript и измените его заготовку следующим образом:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName = "GameData", menuName = "Game Data", order = 51)]
public class GameDataScript : ScriptableObject
{
    public int level = 1;
    public int balls = 6;
    public int points = 0;
}
```

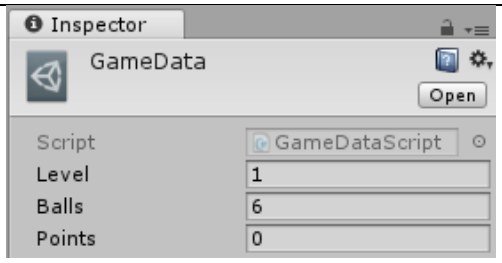
Обратите внимание на то, что предком данного класса должен быть не класс MonoBehaviour (как предлагается по умолчанию), а класс ScriptableObject. Мы поместили в класс GameDataScript поля, содержащие информацию об игре «в целом»: текущий номер уровня, количество имеющихся шариков и число набранных очков.

Любой объект типа ScriptableObject надо создавать на этапе разработки игры. Для реализации этой возможности перед описанием соответствующего класса указывается атрибут CreateAssetMenu, добавляющий в меню Assets | Create новую команду для создания экземпляра данного класса. В параметре fileName указывается имя, под которым созданный экземпляр появится в окне проекта (в дальнейшем это имя можно изменить), в параметре menuName указывается имя команды, которая будет добавлена в меню, а параметр order позволяет задать положение этой команды среди имеющихся команд (значение 51 помещает команду сразу после команды Folder, поскольку каждая группа команд меню Assets | Create связывается с диапазоном из 50 номеров).

Сохраните скрипт GameDataScript, вернитесь в редактор Unity и выполните команду Game Data, которая появится в меню Assets | Create. В результате в окне проекта появится объект GameData. Перетащите его в папку Scripts; при этом папка примет следующий вид:



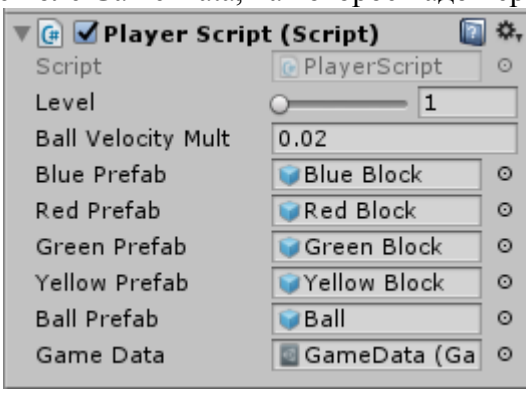
При выделении объекта GameData в окне инспектора отображаются его поля:



Подключите созданный объект к игровому объекту Player, добавив в описание класса PlayerScript дополнительное открытое поле:

```
public GameDataScript gameData;
```

После сохранения измененного текста скрипта PlayerScript в его разделе в окне инспектора появится новое поле Game Data, на которое надо перетащить объект GameData из окна проекта:



Теперь мы можем получать и изменять данные объекта GameData из объекта PlayerScript, причем эти данные сохранятся даже при разрушении объекта Player (которое будет происходить при смене сцены).

1.7.2 Подсчет очков

Вначале реализуем простейшую возможность: подсчет очков и их сохранение в объекте GameData. Для этого достаточно добавить единственный новый оператор в метод BlockDestroyed класса PlayerScript:

```
public void BlockDestroyed(int points)
{
    gameData.points += points;
    StartCoroutine(BlockDestroyedCoroutine());
}
```

Для проверки запустите игру, предварительно выделив в окне проекта объект GameData, чтобы его данные отображались в окне инспектора. При разрушении блоков число очков в поле Points будет увеличиваться.

Когда игра будет остановлена, данные в поле Points сохранятся. Это отличает объекты типа SriptableObject от объектов MonoBehavior, для которых после завершения игры восстанавливаются данные, указанные перед ее запуском. Такое поведение объекта SriptableObject может оказаться удобным для отладки игры, но в стандартном режиме необходимо начинать игру с исходных настроек. Поэтому предусмотрим вариант автоматического сброса настроек объекта GameData.

Измените скрипт GameDataScript:

```
public class GameDataScript : ScriptableObject
{
    public bool resetOnStart;
    public int level = 1;
    public int balls = 6;
    public int points = 0;

    public void Reset()
    {
        level = 1;
        balls = 6;
        points = 0;
```

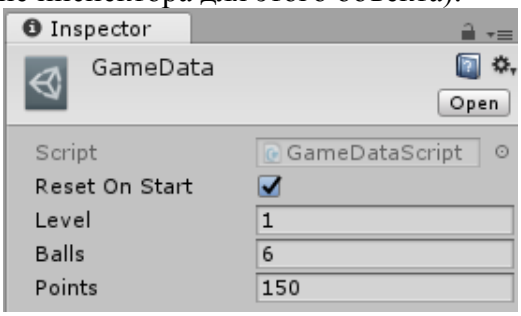
```
    }
}
```

Кроме того, измените скрипт PlayerScript, добавив к нему новое статическое поле и дополнив метод Start:

```
static bool gameStarted = false;

void Start()
{
    Cursor.visible = false;
    if (!gameStarted)
    {
        gameStarted = true;
        if (gameData.resetOnStart)
            gameData.Reset();
    }
    StartLevel();
}
```

Теперь в начале игры, когда объект PlayerScript создается первый раз, его статическое поле gameStarted будет равно false, в методе Start его значение будет изменено на true и в дальнейшем уже не будет изменяться на протяжении всей игры. В момент изменения поля gameStarted на true проверяется значение поля gameData.resetOnStart, и если оно равно true, происходит сброс всех настроек объекта GameData.

Таким образом, если мы хотим, чтобы при запуске игры сохраненные ранее настройки объекта GameData автоматически сбрасывались, нам достаточно установить флажок Reset On Start (который теперь появится в окне инспектора для этого объекта):



Если теперь запустить игру снова, то можно увидеть, что число очков обнулится.

Подчеркнем, что данные объекта GameData сохраняются после остановки игры только в редакторе Unity. Если сгенерировать exe-файл для нашей игры, то при каждом его запуске объект GameData будет инициализироваться заново, с учетом тех настроек, которые указаны в окне инспектора.

1.7.3 Смена уровней

Теперь реализуем смену уровней при разрушении всех блоков. Для этого достаточно внести в скрипт PlayerScript три небольших изменения.

Добавьте в список директив using новую директиву:

```
using UnityEngine.SceneManagement;
```

Измените метод BlockDestroyedCoroutine:

```
IEnumerator BlockDestroyedCoroutine()
{
    yield return new WaitForSeconds(0.1f);
    if (GameObject.FindGameObjectsWithTag("Block").Length == 0)
        print("Level complete!");
    {
        if (level < maxLevel)
            gameData.level++;
        SceneManager.LoadScene("MainScene");
    }
```

```
    }
```

Добавьте новый оператор в метод Start:

```
    void Start()
    {
        Cursor.visible = false;
        if (!gameStarted)
        {
            gameStarted = true;
            if (gameData.resetOnStart)
                gameData.Reset();
        }
        level = gameData.level;
        StartLevel();
    }
```

Теперь при разрушении всех блоков происходит автоматический переход на новый уровень, за исключением ситуации, когда уже пройден последний, 30 уровень. Если пройден 30 уровень, то он будет загружен заново.

Замечание. Альтернативным вариантом при успешном прохождении 30 уровня является завершение текущей игры с выводом на экран соответствующего сообщения. Этот вариант требует работы с интерфейсными элементами, которые мы здесь подробно не рассматриваем.

1.7.4 Действия при потере шарика

Наконец, реализуем дополнительные действия при потере шарика. Во-первых, при каждой такой потере надо уменьшать число доступных шариков (в объекте GameData), во-вторых, при потере всех шариков на игровом поле надо их восстанавливать за счет имеющегося резерва.

Кроме того, надо предусмотреть специальные действия при исчерпании резерва шариков. В этой ситуации (как и в ситуации с прохождением 30 уровня) можно было бы завершать игру с выводом сообщения, но мы реализуем более простой вариант: немедленно запустим новую игру, загрузив ее 1 уровень.

Как и при смене уровней, для реализации всех описанных возможностей достаточно внести небольшие изменения в скрипт PlayerScript.

В методе CreateBalls предусмотрим ситуацию, когда в резерве остался только один шарик, и поэтому при восстановлении шариков на сцене придется создавать единственный шарик:

```
    void CreateBalls()
    {
        int count = 2;
        if (gameData.balls == 1)
            count = 1;
        for (int i = 0; i < count; i++)
        {
            var obj = Instantiate(ballPrefab);
            var ball = obj.GetComponent<BallScript>();
            ball.ballInitialForce += new Vector2(10 * i, 0);
            ball.ballInitialForce *= 1 + level * ballVelocityMult;
        }
    }
```

При потере шарика будем уменьшать их резерв:

```
    public void BallDestroyed()
    {
        gameData.balls--;
        StartCoroutine(BallDestroyedCoroutine());
    }
```

При потере всех шариков на игровом поле будем восстанавливать шарики из резерва, а если резерв исчерпан, то начинать игру заново:

```
IEnumerator BallDestroyedCoroutine()
{
    yield return new WaitForSeconds(0.1f);
    if (GameObject.FindGameObjectsWithTag("Ball").Length == 0)
        if (gameData.balls > 0)
            CreateBalls();
        else
        {
            gameData.Reset();
            SceneManager.LoadScene("MainScene");
        }
}
```

Чтобы ускорить тестирование добавленных возможностей, удобно снять флажок Reset On Start для объекта GameData и уменьшить число резервных шариков (до трех, двух или одного). Для тестирования ситуации, связанной с началом новой игры (и загрузкой первого уровня), можно задать начальный уровень равным 2.

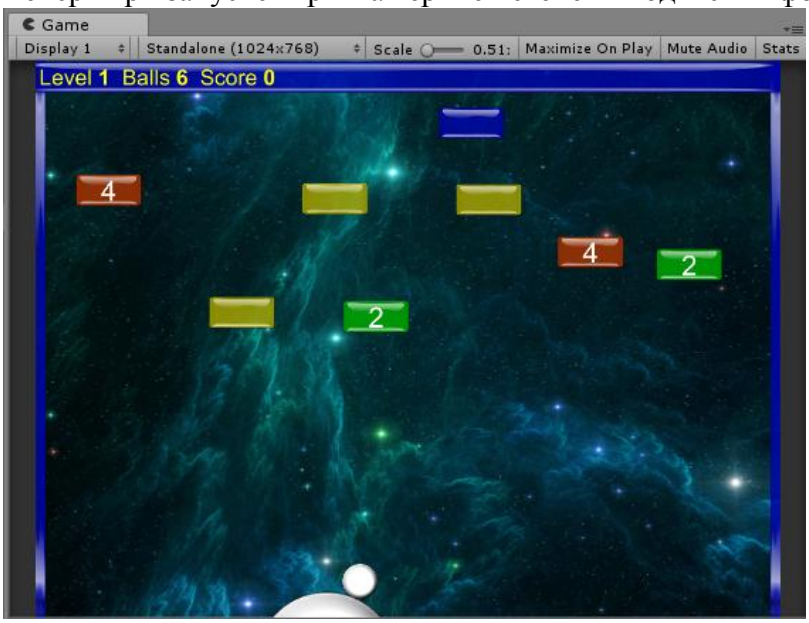
1.8 Отображение текущего состояния игры

В настоящее время данные, связанные с игрой (номер уровня, число доступных шариков и количество набранных очков) отображаются только в окне инспектора при выборе объекта GameData. Добавим эту информацию на сцену. Для этого воспользуемся простейшим вариантом игрового интерфейса, основанным на обработчике события OnGUI (заметим, что для последних версий Unity рекомендуется использовать другой вариант графического интерфейса, в котором все интерфейсные объекты создаются и редактируются подобно обычным игровым объектам).

Добавьте в описание класса PlayerScript новый метод:

```
void OnGUI()
{
    GUI.Label(new Rect(5, 4, Screen.width - 10, 100),
        string.Format(
            "<color=yellow><size=30>Level <b>{0}</b>  Balls <b>{1}</b>"+
            "  Score <b>{2}</b></size></color>",
            gameData.level, gameData.balls, gameData.points));
 }
```

Теперь при запуске игры на верхней стене выводится информация, связанная с ее текущим состоянием.



Обратите внимание на то, что для настройки размера и цвета шрифта, а также его стиля (полужирный или курсивный) можно использовать теги, аналогичные тегам HTML.

1.9 Дополнительные звуковые эффекты и управление ими

В данном разделе мы добавим к игре дополнительные звуковые эффекты и предусмотрим средства для их отключения.

1.9.1 Сигнал при увеличении количества очков

Вначале добавим звуковой эффект, связанный с увеличением очков. Для этого добавим поля для компонента Audio Source и соответствующего звукового файла к классу PlayerScript, в методе Start подключим нужный компонент и вызовем звуковой файл в методе BlockDestroyed:

```
AudioSource audioSrc;
public AudioClip pointSound;

void Start()
{
    audioSrc = Camera.main.GetComponent<AudioSource>();
    ...
}

public void BlockDestroyed(int points)
{
    gameData.points += points;
    audioSrc.PlayOneShot(pointSound);
    StartCoroutine(BlockDestroyedCoroutine());
}
```

После сохранения скрипта и возврата в редактор Unity надо перетащить звуковой файл point на поле Point Sound компонента Player Script.

Теперь при увеличении числа очков выдается особый звуковой сигнал.

1.9.2 Фоновая музыка

Добавим также фоновую музыку. Для этого достаточно перетащить звуковой файл Ricochet Xtreme на поле AudioClip компонента Audio Source объекта Main Camera. При этом желательно уменьшить уровень громкости до значения 0.2. Кроме того, необходимо установить флажок Loop, чтобы фоновая музыка воспроизводилась непрерывно на протяжении прохождения всего уровня.

Недочет. При запуске игры можно обнаружить, что теперь звуковые эффекты воспроизводятся тише, чем раньше. Это объясняется тем, что при их воспроизведении используется уровень громкости, установленный в компоненте Audio Source. Чтобы увеличить громкость звуковых эффектов, не изменяя громкости фоновой музыки, достаточно использовать вариант метода PlayOnShot с двумя параметрами, вторым из которых является множитель, на который увеличивается установленный уровень громкости:

```
// изменение в методе BlockDestroyed класса PlayerScript
        audioSrc.PlayOneShot(pointSound, 5);
// изменение в методе OnTriggerEnter2D класса BallScript
        audioSrc.PlayOneShot(loseSound, 5);
// изменение в методе OnCollisionEnter2D класса BallScript
        audioSrc.PlayOneShot(hitSound, 5);
```

1.9.3 Отключение звуковых эффектов и фоновой музыки

Предусмотрим также команды, позволяющие отключать (а в дальнейшем повторно включать) звуковые эффекты и фоновую музыку. Поскольку эти действия должны влиять на игру в целом, а не на отдельные уровни, следует связать соответствующие настройки с объектом GameData. Поэтому добавьте в скрипт GameDataScript два новых поля типа bool:

```
public bool music = true;
public bool sound = true;
```

В классе PlayerScript опишите вспомогательную функцию

```
void SetMusic()
```

```
    {
        if (gameData.music)
            audioSrc.Play();
        else
            audioSrc.Stop();
    }
```

В метод Start добавьте вызов этой функции:

```
    void Start()
    {
        ...
        SetMusic();
        StartLevel();
    }
```

В метод Update добавьте обработку клавиш M и S (первая переключает режим воспроизведения фоновой музыки, вторая включает или отключает звуковые эффекты):

```
    void Update()
    {
        var mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        var pos = transform.position;
        pos.x = mousePos.x;
        transform.position = pos;
        if (Input.GetKeyDown(KeyCode.M))
        {
            gameData.music = !gameData.music;
            SetMusic();
        }
        if (Input.GetKeyDown(KeyCode.S))
            gameData.sound = !gameData.sound;
    }
```

Наконец, добавьте соответствующую проверку в метод BlockDestroyed:

```
    public void BlockDestroyed(int points)
    {
        gameData.points += points;
        if (gameData.sound)
            audioSrc.PlayOneShot(pointSound, 5);
        StartCoroutine(BlockDestroyedCoroutine());
    }
```

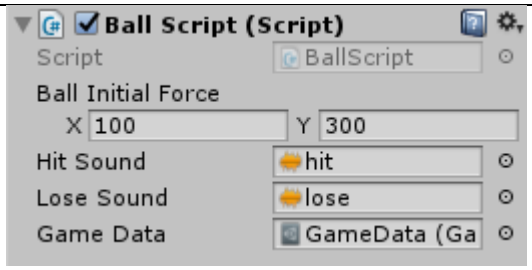
Аналогичные проверки надо добавить и в скрипт BallScript, однако перед этим надо связать данный скрипт с объектом GameData. Для этого добавьте в набор полей класса BallScript новое поле:

```
    public GameDataScript gameData;
```

Используйте проверку поля gameData.sound перед воспроизведением звуковых эффектов:

```
// изменение в методе OnTriggerEnter2D класса BallScript
        if (gameData.sound)
            audioSrc.PlayOneShot(loseSound, 5);
// изменение в методе OnCollisionEnter2D класса BallScript
        if (gameData.sound)
            audioSrc.PlayOneShot(hitSound, 5);
```

После сохранения скрипта BallScript и возвращения в редактор Unity необходимо выделить шаблон Ball в папке Prefabs и перетащить объект GameData на поле Game Data компонента Ball Script в окне инспектора (в котором в этот момент будут отображаться компоненты шаблона Ball):



При тестировании игры удобно отобразить в окне инспектора объект GameData; в этом случае при переключении звуковых режимов мы дополнительно увидим, как изменяются состояния флажков music и sound. Чтобы при тестировании новых возможностей игры временно отключать звуковые эффекты, достаточно снять соответствующие флажки в окне инспектора для объекта GameData.

Перед генерацией exe-файла необходимо установить оба флажка, связанных со звуковыми эффектами, если мы хотим, чтобы эти эффекты были включены при запуске игры.

1.10 Дополнительные игровые возможности

Завершая разработку нашей игры, добавим в нее еще несколько полезных возможностей.

1.10.1 Небольшое изменение траектории шариков

В процессе игры можно заметить, что иногда шарики выходят на «стационарную» траекторию, например, начинают летать по горизонтали от одной стены до другой. Предусмотрим специальную клавишу J для небольшого изменения траектории, позволяющего вывести шарики из стационарного режима. Для этого добавим в метод Update класса BallScript новый фрагмент:

```
if (!rb.isKinematic && Input.GetKeyDown(KeyCode.J))
{
    var v = rb.velocity;
    if (Random.Range(0,2) == 0)
        v.Set(v.x - 0.1f, v.y + 0.1f);
    else
        v.Set(v.x + 0.1f, v.y - 0.1f);
    rb.velocity = v;
}
```

В данном фрагменте происходит небольшое изменение скорости шарика, причем случайным образом выбирается один из двух вариантов. Для выбора варианта используется датчик случайных чисел — объект Range, входящий в стандартную библиотеку Unity.

1.10.2 Добавление резервных шариков при получении достаточного числа очков

Естественно ожидать, что начального количества из 6 резервных шариков окажется недостаточно для прохождения всех уровней. Поскольку мы не планируем включать в игру возможности, связанные с падающими вниз бонусами, которые можно подбирать ракеткой, реализуем следующую возможность: добавление нового резервного шарика после накопления определенного количества очков. Это количество будет зависеть от номера уровня: чем больше уровень, тем больше очков надо набрать, находясь на этом уровне, чтобы получить новый резервный шарик. Накопленные очки, разумеется, будут переходить на следующий уровень, так что новый шарик можно будет получить, например, накопив очки при прохождении двух или трех уровней. Будем считать, что для первого уровня число требуемых очков равно 400, а для каждого последующего уровня это число будет увеличиваться на 20 (так что на последнем, 30 уровне для получения нового шарика потребуется накопить уже 980 очков).

В объекте GameData будем хранить текущую сумму накопленных очков, еще не «переведенную» в шарик; назовем ее pointsToBall. В объекте Player заведем свойство только для чтения requiredPointsToBall; это свойство будет возвращать количество очков, необходимое для получения нового шарика на текущем уровне.

Итак, внесем в скрипт GameDataScript следующие изменения:

```
public class GameDataScript : ScriptableObject
{
    public bool resetOnStart;
    public int level = 1;
```

```
    public int balls = 6;
    public int points = 0;
    public int pointsToBall = 0;
    public bool music = true;
    public bool sound = true;

    public void Reset()
    {
        level = 1;
        balls = 6;
        points = 0;
        pointsToBall = 0;
    }
}
```

В скрипт PlayerScript добавим описание нового свойства requiredPointsToBall и дополним метод BlockDestroyed:

```
    int requiredPointsToBall
      { get { return 400 + (level - 1) * 20; } }

    public void BlockDestroyed(int points)
    {
        gameData.points += points;
        if (gameData.sound)
            audioSrc.PlayOneShot(pointSound, 5);
        gameData.pointsToBall += points;
        if (gameData.pointsToBall >= requiredPointsToBall)
        {
            gameData.balls++;
            gameData.pointsToBall -= requiredPointsToBall;
        }
        StartCoroutine(BlockDestroyedCoroutine());
    }
```

Чтобы упростить тестирование добавленной возможности, удобно сразу настроить данные для объекта GameData таким образом, чтобы получить требуемое число очков, разрушив лишь небольшое количество блоков (например, можно сразу положить поле Points To Ball равным 380). При этом, разумеется надо снять флажок Reset On Start, чтобы значения полей объекта GameData не были сброшены в начале игры.

Приобретение нового резервного шарика можно также отметить каким-либо звуковым эффектом. Вместо того чтобы добавлять новый эффект, поступим по-другому: воспроизведем тот же эффект, что и при получении очков, но повторим его 10 раз с паузами в 0.2 секунды. Для подобных действий хорошо подходят сопрограммы, поэтому реализуем еще одну сопрограмму, связанную с методом BlockDestroyed, и вызовем ее в этом методе:

```
    IEnumerator BlockDestroyedCoroutine2()
    {
        for (int i = 0; i < 10; i++)
        {
            yield return new WaitForSeconds(0.2f);
            audioSrc.PlayOneShot(pointSound, 5);
        }
    }

    public void BlockDestroyed(int points)
    {
```

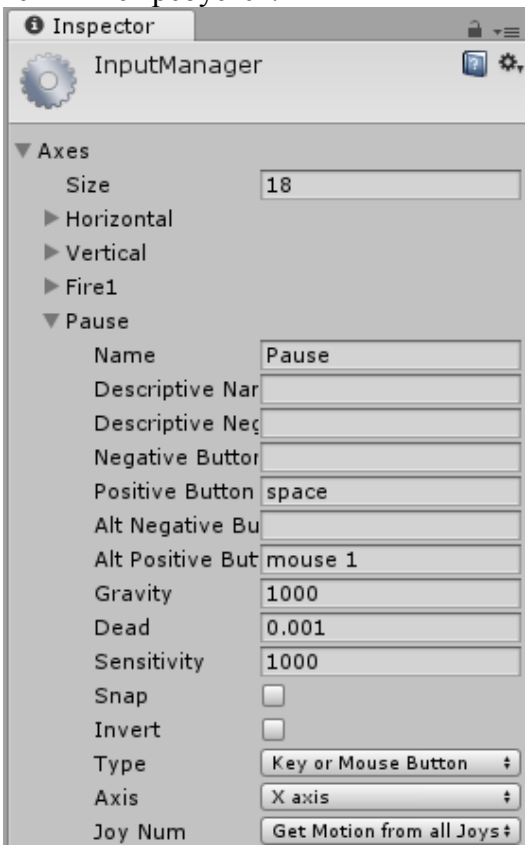
```
        gameData.points += points;
        if (gameData.sound)
            audioSrc.PlayOneShot(pointSound, 5);
        gameData.pointsToBall += points;
        if (gameData.pointsToBall >= requiredPointsToBall)
        {
            gameData.balls++;
            gameData.pointsToBall -= requiredPointsToBall;
            if (gameData.sound)
                StartCoroutine(BlockDestroyedCoroutine2());
        }
        StartCoroutine(BlockDestroyedCoroutine());
    }
```

1.10.3 Режим паузы

Свяжем клавишу пробела и правую кнопку мыши с включением и выключением режима паузы.

Для единообразной обработки описанных действий воспользуемся специальной возможностью Unity — настройкой особых команд, называемых «осями» (axes), с которыми можно связать любые клавиши, а также нажатия кнопок мыши. Заметим, что ранее мы уже воспользовались одной из осей (Fire1), которая обеспечивает запуск шарика (эта ось связана с левой клавишей Ctrl и левой кнопкой мыши). Сейчас мы используем еще одну ось, причем предварительно настроим ее свойства.

Выполните команду Edit | Project Settings | Input. При этом в окне инспектора появится список «осей» Axes. Разверните этот список (если он является свернутым) и перейдите на первую ось Fire2. Измените ее имя на Pause, в поле Positive Button введите space, убедитесь, что в поле Alt Positive Button указано значение mouse 1. Проверьте также, что поле Type содержит вариант Key or Mouse Button. Остальные значения полей изменять не требуется.



Для проверки того, что ось Pause правильно настроена, добавьте следующий фрагмент в метод Update класса PlayerScript:

```
        if (Input.GetButtonDown("Pause"))
            print("Pause");
```

Если теперь после запуска игры нажать на пробел или на правую кнопку мыши, то в консольном окне будет выведен текст Pause.

Чтобы остановить движение всех игровых объектов, достаточно «остановить игровое время», положив его масштабный множитель Time.timeScale равным нулю:

```
if (Input.GetButtonDown("Pause"))
    print("Pause");
    if (Time.timeScale > 0)
        Time.timeScale = 0;
    else
        Time.timeScale = 1;
```

Однако этого недостаточно, чтобы отключить управление ракеткой. Для отключения управления ракеткой добавьте в начало метода Update еще одну проверку:

```
if (Time.timeScale > 0)
{
    var mousePos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
    var pos = transform.position;
    pos.x = mousePos.x;
    transform.position = pos;
}
```

1.10.4 Выход с сохранением текущего состояния игры и восстановление сохраненного состояния

Если игру требуется прервать, то было бы удобно сохранить ее текущее состояние, чтобы в следующий раз не начинать игру сначала. В нашем случае все настройки игры собраны в объекте GameData, поэтому достаточно сохранить все поля данного объекта. Оформим это сохранение в виде вспомогательного метода класса GameDataScript:

```
public void Save()
{
    PlayerPrefs.SetInt("level", level);
    PlayerPrefs.SetInt("balls", balls);
    PlayerPrefs.SetInt("points", points);
    PlayerPrefs.SetInt("pointsToBall", pointsToBall);
    PlayerPrefs.SetInt("music", music ? 1 : 0);
    PlayerPrefs.SetInt("sound", sound ? 1 : 0);
}
```

Вызывать этот метод будем в классе PlayerScript в специальном обработчике, который вызывается при завершении программы:

```
void OnApplicationQuit()
{
    gameData.Save();
}
```

В классе GameDataScript необходимо также описать метод для восстановления сохраненных данных:

```
public void Load()
{
    level = PlayerPrefs.GetInt("level", 1);
    balls = PlayerPrefs.GetInt("balls", 6);
    points = PlayerPrefs.GetInt("points", 0);
    pointsToBall = PlayerPrefs.GetInt("pointsToBall", 0);
    music = PlayerPrefs.GetInt("music", 1) == 1;
    sound = PlayerPrefs.GetInt("sound", 1) == 1;
}
```

Этот метод надо вызывать при запуске программы. Соответствующие действия уже предусмотрены в методе Start класса PlayerScript; осталось внести в этот метод небольшое изменение:

```
void Start()
```

```
    {
        audioSrc = Camera.main.GetComponent<AudioSource>();
        Cursor.visible = false;
        if (!gameStarted)
        {
            gameStarted = true;
            if (gameData.resetOnStart)
                gameData.Reset();
                gameData.Load();
        }
        level = gameData.level;
        SetMusic();
        StartLevel();
    }
```

Протестировать новую возможность можно непосредственно в редакторе Unity: при остановке игры и ее возобновлении восстанавливается последний уровень, число резервных шариков и количество набранных баллов. Кроме того, восстанавливается предыдущий звуковой режим. Теперь флажок Reset On Start не сбрасывает поля в их начальное состояние, а обеспечивает восстановление ранее сохраненных значений. Если снять флажок Reset On Start, то по-прежнему можно задавать начальные настройки, используя окно инспектора.

При компиляции игры для создания exe-файла необходимо установить флажок Reset On Start.

1.10.5 Быстрые команды для начала новой игры и ее завершения

Игрок может захотеть начать новую игру, не дожидаясь завершения предыдущей. Поскольку теперь выход из программы и ее последующий запуск не приводит к началу новой игры, предусмотрим для этого действия особую клавишу N и добавим соответствующий код в метод Update класса PlayerScript:

```
    void Update()
    {
        ...
        if (Input.GetKeyDown(KeyCode.N))
        {
            gameData.Reset();
            SceneManager.LoadScene("MainScene");
        }
    }
```

Предусмотрим также быстрый способ завершения программы — нажатие клавиши Esc. Соответствующий фрагмент тоже поместим в метод Update класса PlayerScript:

```
    void Update()
    {
        ...
        if (Input.GetKeyDown(KeyCode.Escape))
            Application.Quit();
    }
```

В отличие от предыдущих возможностей, эту возможность можно протестировать только при запуске откомпилированного exe-файла.

1.10.6 Вывод информации о назначении клавиш

Мы реализовали в нашей игре много команд, связанных с клавиатурными комбинациями. Было бы удобно отобразить на игровом экране эти комбинации с кратким описанием команд. Кроме того, для команд, связанных с настройкой звука и режимом паузы, можно также отображать информацию о том: к чему приведет данная команда: к включению или отключению данной звуковой настройки или режима паузы.

Воспользуемся для этого тем же простейшим способом, основанным на обработчике OnGUI. Предварительно опишем вспомогательную функцию OnOff(boolVal), возвращающую текст «on» или «off» в зависимости от значения логического параметра boolVal:

```
string OnOff(bool boolVal)
{
    return boolVal ? "on" : "off";
}

void OnGUI()
{
    GUI.Label(new Rect(5, 4, Screen.width - 10, 100),
        string.Format(
            "<color=yellow><size=30>Level <b>{0}</b>  Balls <b>{1}</b>"+
            "  Score <b>{2}</b></size></color>",
            gameData.level, gameData.balls, gameData.points));
    GUIStyle style = new GUIStyle();
    style.alignment = TextAnchor.UpperRight;
    GUI.Label(new Rect(5, 14, Screen.width - 10, 100),
        string.Format(
            "<color=yellow><size=20><color=white>Space</color>-pause {0}" +
            " <color=white>N</color>-new" +
            " <color=white>J</color>-jump" +
            " <color=white>M</color>-music {1}" +
            " <color=white>S</color>-sound {2}" +
            " <color=white>Esc</color>-exit</size></color>",
            OnOff(Time.timeScale > 0), OnOff(!gameData.music), OnOff(!gameData.sound)),
        style);
}
```

Для вывода поясняющей информации мы использовали шрифт меньшего размера и выровняли эту информацию по правому краю игрового окна.

На рисунке приведен вид окна в режиме паузы при включенных звуковых эффектах и отключенной фоновой музыке.

На этом разработку нашего варианта игры Arcanoid можно считать законченной.