Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

Многопрофильный колледж

УТВЕРЖДАЮ
Директор
_____ / С.А. Махновский
«09» февраля 2022 г.

# МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

**по учебной дисциплине**
**ОПЦ.13 Разработка компьютерных игр**

**для обучающихся специальности**

**09.02.07 Информационные системы и программирование**
**Квалификация: Разработчик веб и мультимедийных приложений**

Магнитогорск, 2022

**ОДОБРЕНО**

Предметно-цикловой комиссией «Информатики и вычислительной техники»
Председатель И.Г.Зорина
Протокол № 5 от 19.01.2022г.

Методической комиссией МпК
Протокол № 4 от 09.02.2022 г.

.

**Разработчик:**
преподаватель ФГБОУ ВО «МГТУ им. Г.И. Носова» Многопрофильный колледж
Д.А. Кузьмин

Методические указания по выполнению практических и лабораторных работ разработаны на основе рабочей программы учебной дисциплины «Разработка компьютерных игр».

Содержание практических и лабораторных работ ориентировано на подготовку обучающихся к освоению профессионального модуля программы подготовки специалистов среднего звена по специальности 09.02.07 Информационные системы и программирование и овладению общими компетенциями.

**СОДЕРЖАНИЕ**

# 1 ВВЕДЕНИЕ

Важную часть теоретической и профессиональной практической подготовки обучающихся составляют практические занятия.

Состав и содержание практических занятий направлены на реализацию Федерального государственного образовательного стандарта среднего профессионального образования.

Ведущей дидактической целью практических занятий является формирование профессиональных практических умений (умений выполнять определенные действия, операции, необходимые в последующем в профессиональной деятельности).

В соответствии с рабочей программой учебной дисциплины «Разработка компьютерных игр» предусмотрено проведение практических занятий.

В результате их выполнения, обучающийся должен:

***уметь:***

- программировать игровую механику и реализовывать геймплей согласно техническому описанию;

– определять и применять в работе инструментальные средства для разработки архитектуры компьютерной игры;

– выбирать и определять методы реализации и представления внутренних данных компьютерной игры;

– рисовать, выбирать, использовать эскизы персонажей, объектов для компьютерной игры;

– выбирать и создавать звуковые и другие эффекты, используемые в компьютерной игре;

– выбирать и применять в работе виртуальный игровой движок;

– определять и учитывать уровни сложности в программировании игры;

– объединять подготовленные части игры;

– дополнять элементы требуемыми эффектами компьютерной игры;

– подготовить модули для редактирования уровней;

– подобрать программные средства для включения анимированных вставок.

ПК 5.4. Производить разработку модулей информационной системы в соответствии с техническим заданием.

ПК 8.3. Осуществлять разработку дизайна веб-приложения с учетом современных тенденций в области веб-разработки.

ПК 9.4. Осуществлять техническое сопровождение и восстановление веб-приложений в соответствии с техническим заданием.

ОК 01. Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам.

ОК 02. Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности.

ОК 04. Работать в коллективе и команде, эффективно взаимодействовать с коллегами, руководством, клиентами.

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке с учетом особенностей социального и культурного контекста.

ОК 09. Использовать информационные технологии в профессиональной деятельности.

ОК 10. Пользоваться профессиональной документацией на государственном и иностранном языках.

Выполнение обучающихся практических работ по учебной дисциплине «Разработка компьютерных игр» направлено на:

- обобщение, систематизацию, углубление, закрепление, развитие и детализацию полученных теоретических знаний по конкретным темам учебной дисциплины;

- формирование умений применять полученные знания на практике, реализацию единства интеллектуальной и практической деятельности;

- формирование и развитие умений: наблюдать, сравнивать, сопоставлять, анализировать, делать выводы и обобщения, самостоятельно вести исследования, пользоваться различными приемами измерений, оформлять результаты в виде таблиц, схем, графиков;

- развитие интеллектуальных умений у будущих специалистов: аналитических, проектировочных и др.;

- выработку при решении поставленных задач профессионально значимых качеств, таких как самостоятельность, ответственность, точность, творческая инициатива.

Практические занятия проводятся после соответствующей темы, которая обеспечивает наличие знаний, необходимых для ее выполнения.

# 2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ

## Тема 2 Знакомство со средой разработки Unity.

## Практическое занятие № 1. Установка и настройка Unity

**Выполнив работу, Вы будете:**
***уметь:***
-устанавливать и настраивать Unity;
-создавать новые проекты в формате 3D и 2D

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Скачать, установить и настроить среду разработки Unity
2. Создать папку со своей фамилией, в папке группы, создать новый проект и сохранить в свою папку

**Порядок выполнения работ:**

**Настройка среды в Unity**

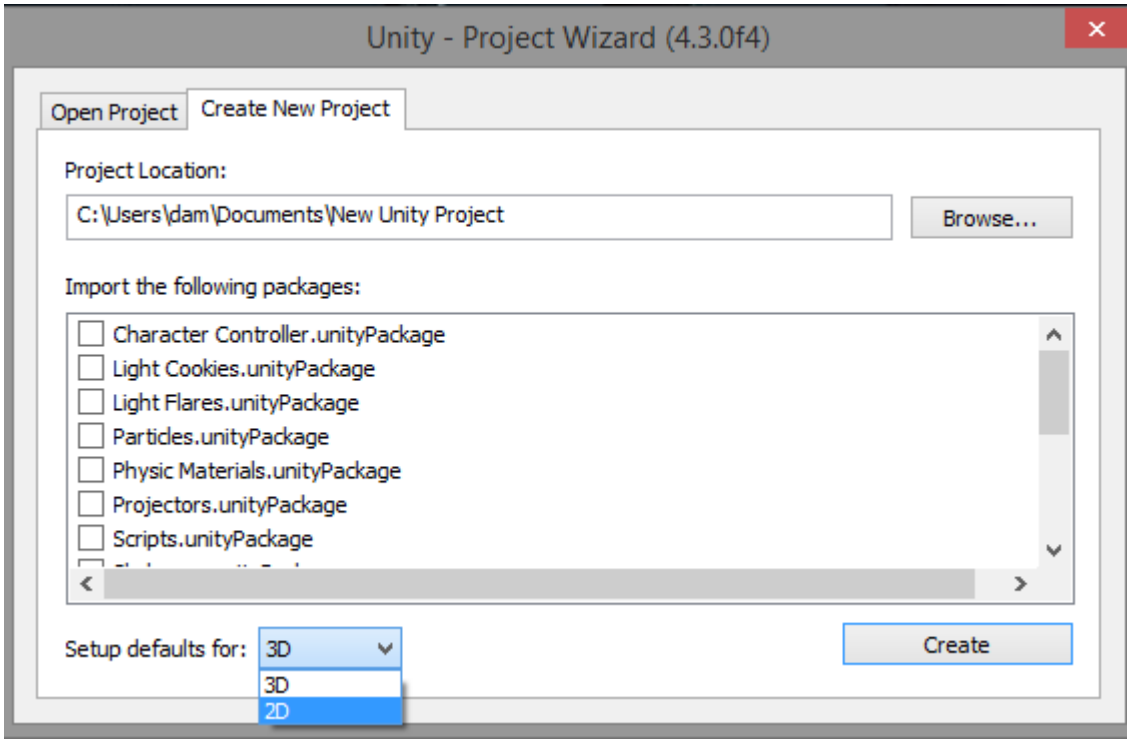
**Загрузка и настройка Unity.**

Загрузите последнюю версию с официального сайта и запустите установочный файл.

Для редактирования кода в Unity (4.0.1 и выше) служит редактор MonoDevelop. Если вы работаете в Windows, вы можете (и я вам советую) использовать альтернативый редактор Visual Studio, после чего в настройках Unity измените редактор по умолчанию на Visual Studio.

**Первая сцена. Создаем новый проект.**

Выберите меню File, а затем создаqnt новый проект. Не выбирайте нивакой стандартный пакет на первое время. Вы можете повторно импортировать их позже, если вы захотите, просто поначалу они будут просто сбивать вас с толку.

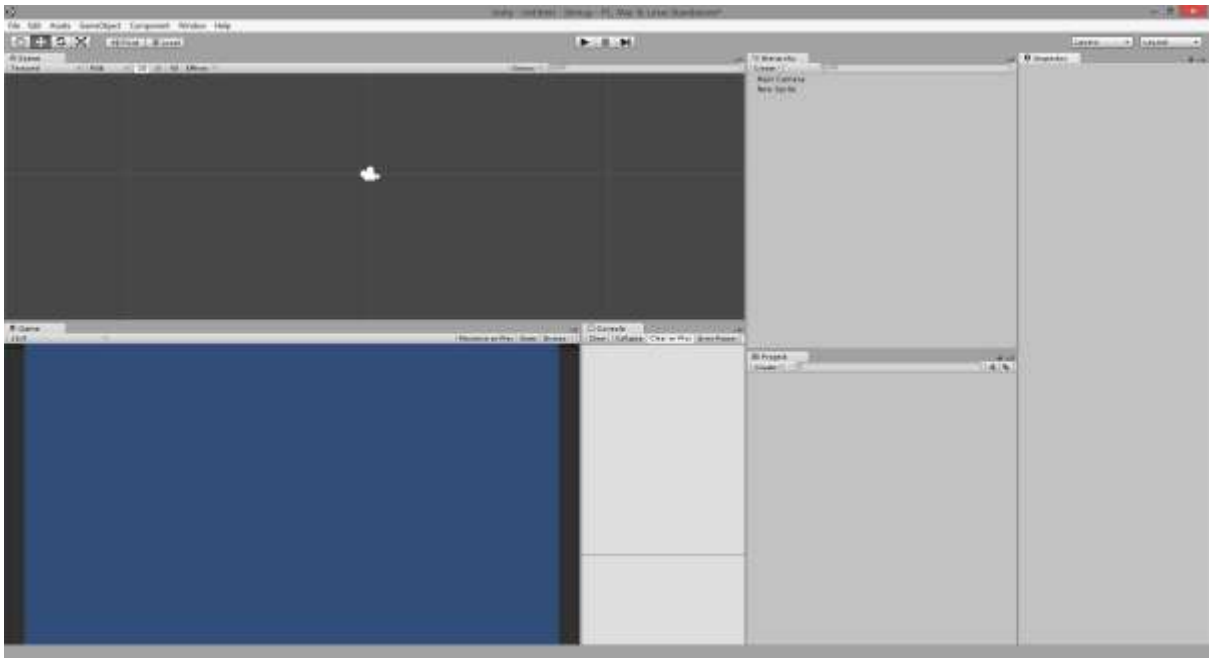



Выберите **2D** настройки. Как и прежде, вы можете изменить этот флаг в настройках проекта позже.

Не беспокойтесь о названии. Оно определяется в настройках, и чтобы изменить имя проекта достаточно просто переименовать папку.

**Разметка и панели Unity**

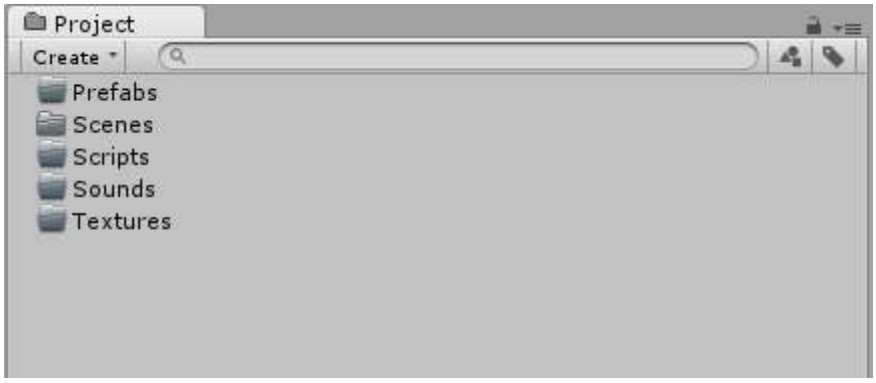
Перед вами пустая страница. С ней вы и будете работать, но вам потребуется время, чтобы настроить интерфейс в соответствии со своими конкретными нуждами. Лично мне удобнее, когда консоль находится рядом с игровым экраном, но если у вас маленький монитор, вы можете заменить панели вкладками.



Прежде чем перейти к созданию игры, уделите несколько минут, чтобы подготовить свой проект и сцены.

Чтобы держать все под рукой, советуем создать папки во вкладке Project (Проект). Эти папки будут созданы в папке Assets вашего проекта.

Внимание: папка Assets – это место, где хранится все, что вы добавляете во вкладке **Project.** Она может быть невидимой в Unity, в зависимости от выбранной разметки вкладки (одна или две колонки), но вы сможете увидеть ее, открыв приложение для экспорта файлов.




Вот пример структуры, которую мы используем в наших проектах. Вы можете адаптировать ее под свои предпочтения.

**Ассеты проекта**

В вашей панели **Project**, вы можете найти различные типы ассетов:

**Префабы**

Многоразовые игровые объекты (например: пули, враги, бонусы).

Префабы можно рассматривать как класс в языке программирования, который может быть обработан в игровых объектах. Это некая форма, которую можно дублировать и изменить по своему желанию в сцене или во время выполнения игры.

**Сцены**

Сцена содержит игровой уровень или меню.

В отличие от других объектов, создаваемых в панели "Проект", сцены создаются в меню "Файл". Если вы хотите создать сцену, нажмите на кнопку "Новая сцена" в подменю и не забудьте потом сохранить ее в папку **Scenes.**

Сцены должны быть сохранены вручную. Это классическая ошибка в Unity - сделать некоторые изменения в сцене и ее элементы и забыть сохранить их после. Ваш инструмент контроля версий не увидите никаких изменений до тех пор, сцена не сохранится.

**Звуки**

Тут все предельно просто. Увидите, если захотите раскидать музыку по разным папкам.

**Scripts**

Весь код находится здесь. Мы используем эту папку в качестве эквивалента корневой папке в C# проекте.

**Textures**

Спрайты и изображения вашей игры. В 2D проекте вы можете переименовать эту папку в "Sprites".

Это неважно для 2D проекта, но, оставив название Textures (Текстуры), вы дадите возможность Unity автоматизировать некоторые задачи.

Заметка о папке Resources: если вы уже работали с Unity, вы знает, что Resources – полезная и уникальная папка. Она позволяет загрузить в скрипт объект или файл (с помощью статичного класса Resources). Она понадобится нам в самом конце (в главе, посвященной меню). Проще говоря, пока мы не будем ее добавлять.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Тема 3 Разработка компьютерной игры.**

**Практическое занятие № 2. Создание статичной сцены**

**Выполнив работу, Вы будете:**
***уметь:***
-создавать статические сцены
-добавлять фон на сцену
**Материальное обеспечение:**
Методические указания для выполнения практических работ

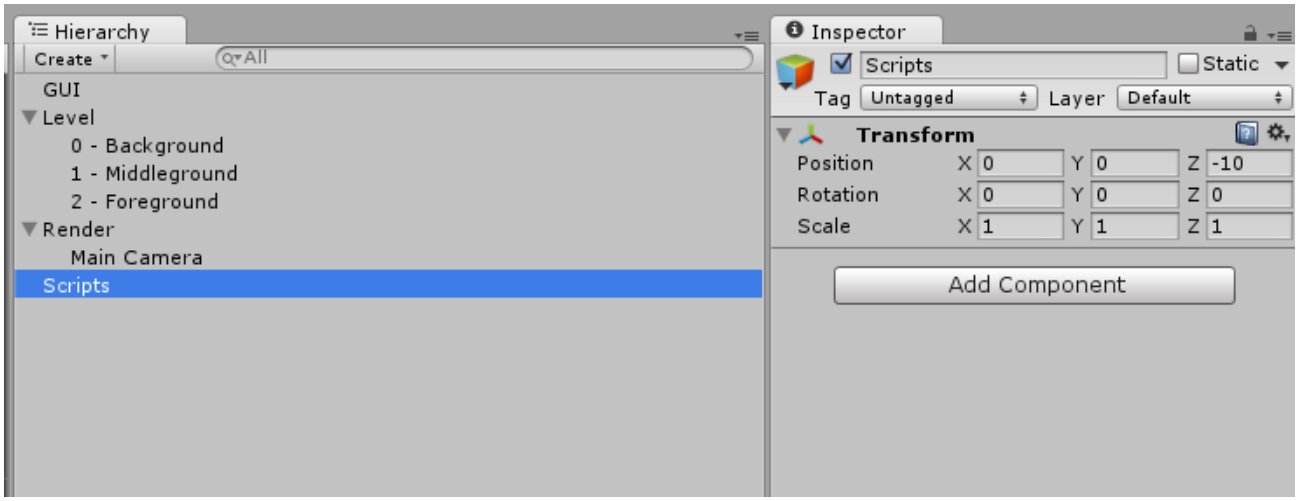
**Задание:**
1. Создать пустые объекты для разделения слоев сцены игры
2. Добавить на сцену статический фон
3. Разделить слои на сцене
4. Добавить платформы на сцену

**Порядок выполнения работ:**

**Первая игровая сцена**

Панель **Hierarchy** (Иерархия) содержит все объекты, которые доступны в сцене. Это то, чем вы манипулируете, когда начинаете игру с помощью кнопки "Play".

Каждый объект сцены является игровым объектом для Unity. Вы можете создать объект в главной сцене, или в другом объекте игры. Также вы можете в любое время переместить объект чтобы изменить его родителя.




Как вы можете видеть здесь, у нас здесь 3 потомка для объекта Level.

Пустые объекты

В **Unity** можно создать пустой объект и использовать его в качестве "папки" для других игровых объектов. Это упростит структуру вашей сцены.


Убедитесь, что все они имеют координаты `(0, 0, 0)` и тогда вы сможете легко их найти! Пустые объекты никак не используют свои координаты, но они влияют на относительные

координаты их потомков. Мы не будем говорить об этой теме в этом уроке, давайте просто обнулим координаты ныших пустых объектов.

Заполнение сцены

По умолчанию, новая сцена создается с объектом Main Camera (Главная камера). Перетащите ее на сцену.

Для начала создайте эти пустые объекты:

Scripts

Мы добавим наши скрипты сюда. Мы используем этот объект, чтобы прикрепить сценарии, которые не связаны с объектом – например, скрипт гейм-менеджера.

Render

Здесь будет наша камера и источники света.

Level

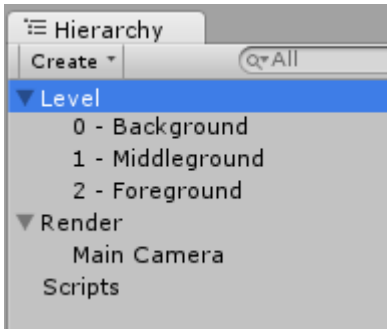
В Level создайте 3 пустых объекта:

0 - Background

1 - Middleground

2 - Foreground

Сохраните сцену в папке `Scenes`. Назовите ее как угодно, например Stage1. Вот, что у нас получилось:

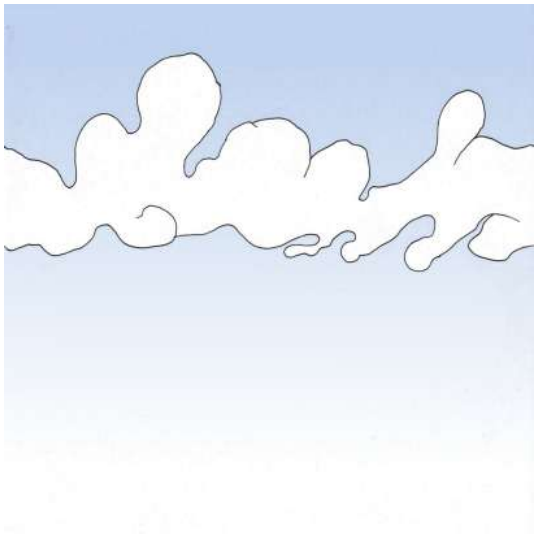



Совет: по умолчанию игровой объект привязан к положению родителя. Это приводит к интересному побочному эффекту при использовании объекта камеры: если камера является дочерним объектом, она автоматически будет отслеживать положение родителя. Если же она является корневым объектом сцены или находится внутри пустого игрового объекта, она всегда показывает один и тот же вид. Однако если вы поместите камеру в движущийся игровой объект, она будет следовать за его передвижениями в пределах сцены. В данном случае нам нужна фиксированная камера, поэтому мы помещаем ее в пустой объект Render. Но запомните это свойство объекта камеры, оно может вам пригодиться. Мы подробно остановимся на этой теме в главе "Паралаксный скроллинг".

Мы только что создали базовую структуру нашей игры. На следующем этапе мы начнем делать забавные вещи: добавим на сцену фон и кое-что еще!

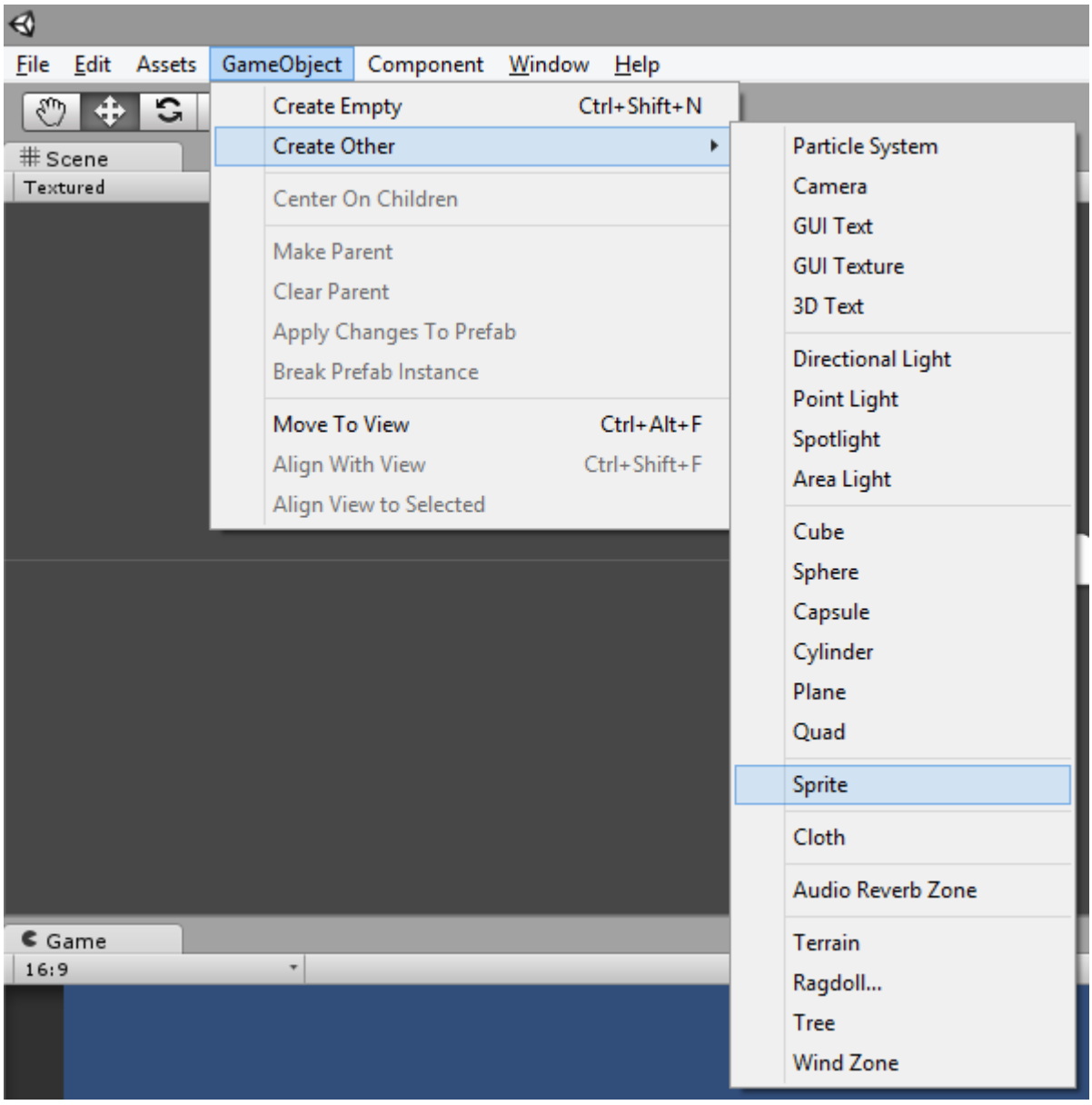
Добавляем фон в сцену

Наш первый фон будет статическим. Воспользуемся следующим изображением:



Импортируйте изображение в папку Textures (Текстуры). Просто скопируйте файл в нее, или перетащите его из проводника. Не беспокойтесь сейчас о настройках импорта.

Создайте в Unity новый игровой объект Sprite на сцене.

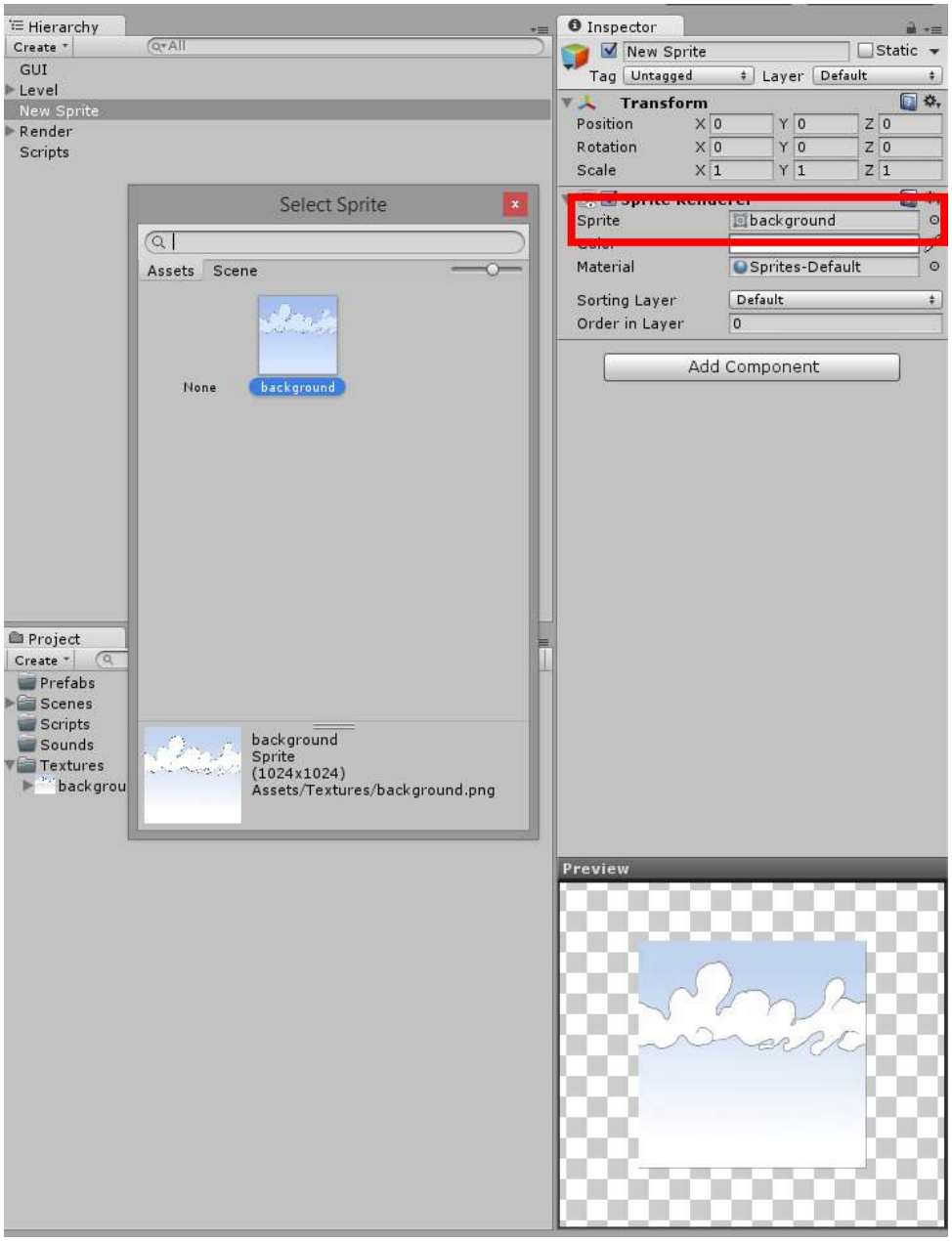



**Что такое спрайт?**

По сути, спрайт – это 2D-изображение, используемое в видео-игре. В данном случае это объект Unity для создания 2D-игр.

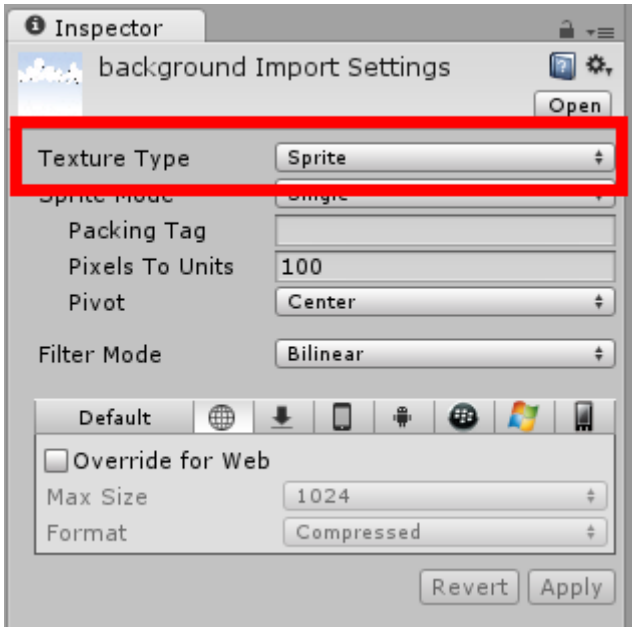
Добавляем текстуру спрайта

Unity может автоматически установить фон для вашего спрайта. Если ничего такого не произошло, или если вы хотите изменить текстуру, перейдите на вкладку инспектора и выберите background: (фон)

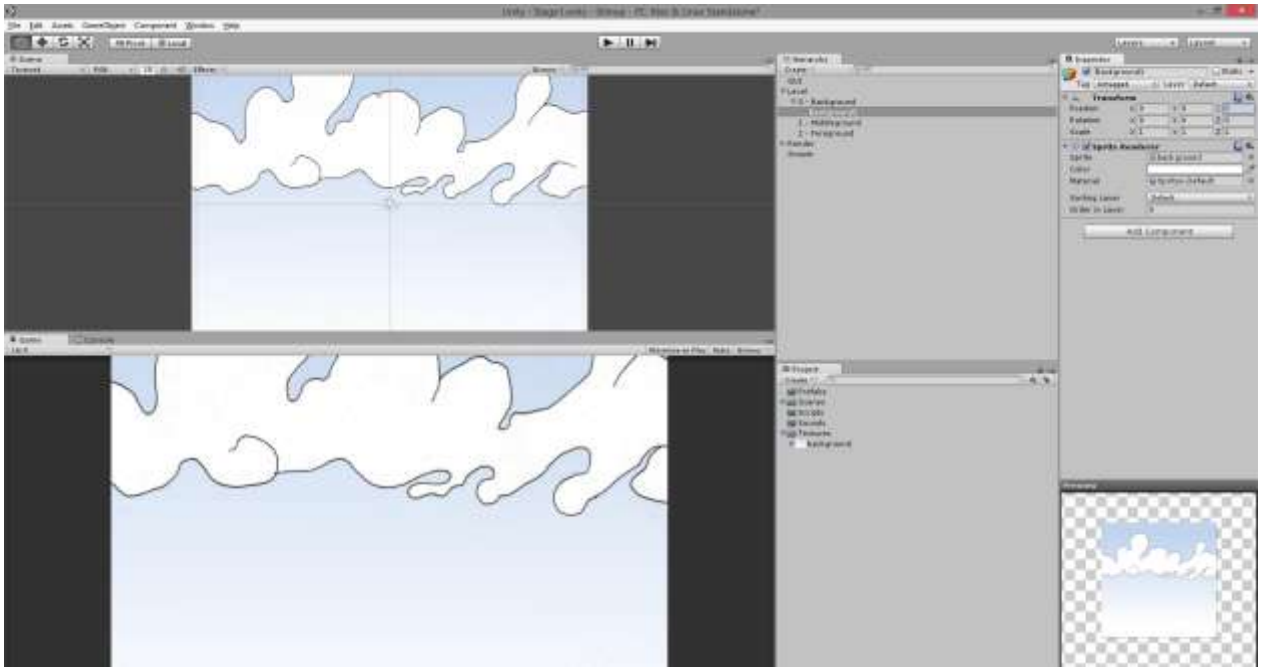



Вы должны нажать на маленький круглый значок справа от поля ввода, чтобы появилось Select Sprite (Выбрать спрайт) в Инспекторе

*Мой спрайт не появляется в диалоговом окне!* Убедитесь, что вы находитесь в вкдадке **Assets** диалогового окна "Select Sprite" (Выбрать спрайт). Если вы видите диалоговое окно пустым, - не пугайтечсь. Дело в том, что для некоторых установок Unity, даже со свежим новым 2D проектом изображения импортируются как "Текстура", а не "Спрайт". Чтобы это исправить, необходимо выбрать изображение на панели "Проект", и в "Инспекторе", изменить свойство "Текстура Type" имущество "Sprite":

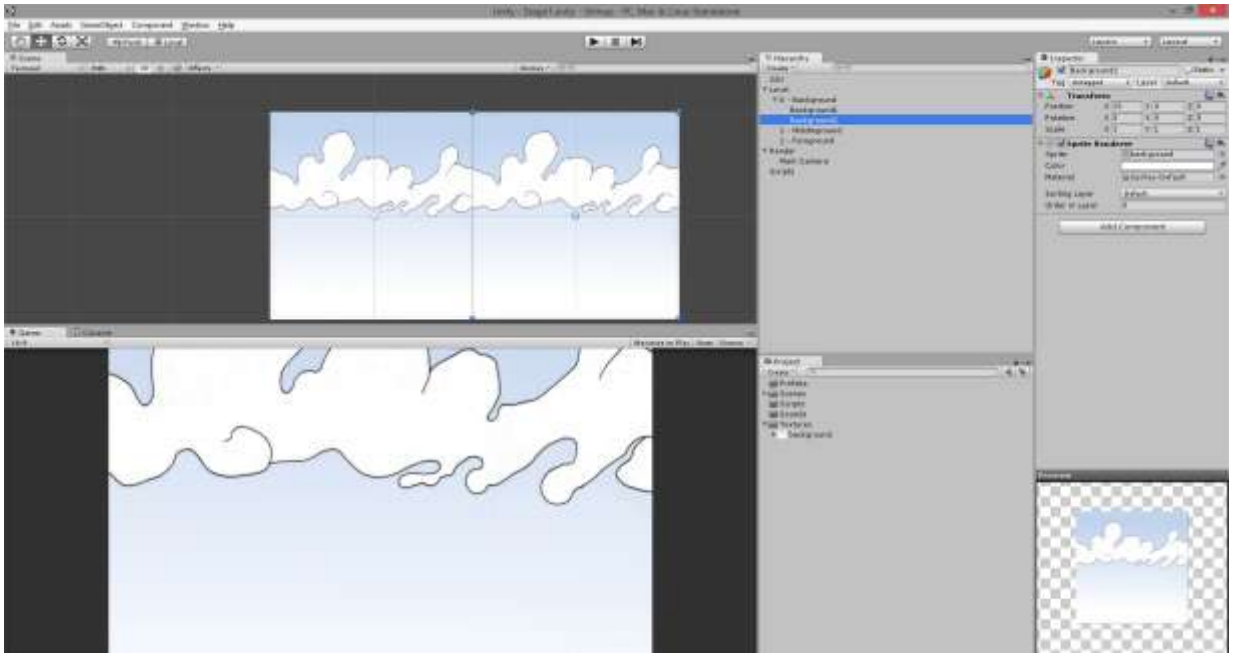



Итак, мы создали простой спрайт отображающий облака на небе. Давайте внесем изменения в сцену. В панели **Hierarchy** (Иерархия) выберите New Sprite. Переименуйте его в Background1 или что-то такое, что легко запомнить. Переименуйте его в Background1 или что-то такое, что легко запомнить. Затем переместите объект в нужное место: Level -> 0 - Background. Измените координаты на (0, 0, 0).



Создайте копию фона и поместите его в (20, 0, 0). Это должно отлично подойти к первой части.

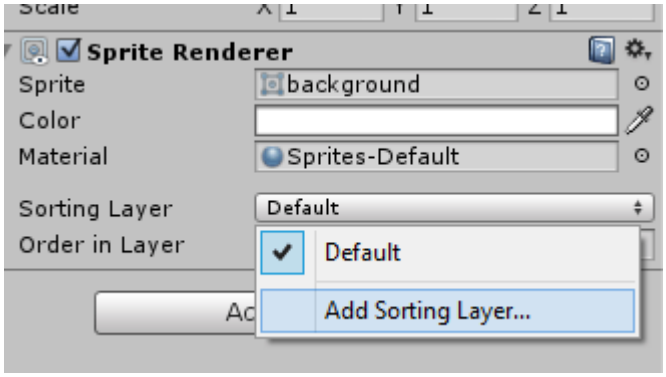
*Tip*: Вы можете создать копию объекта с помощью клавиш cmd + D в OS X или ctrl + D Windows.



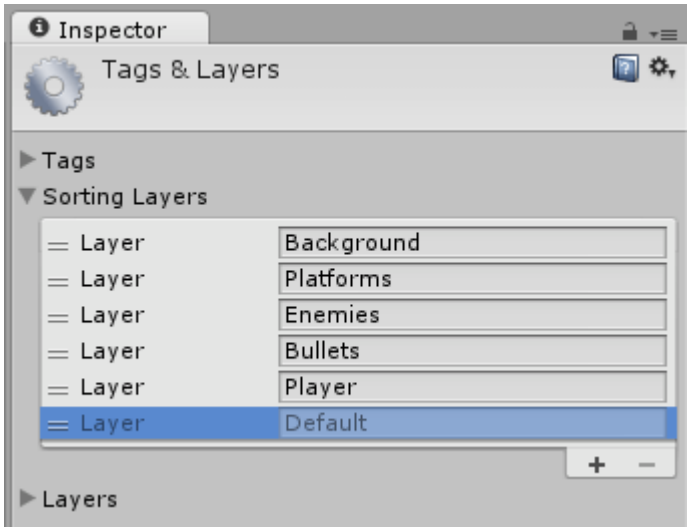
Слои со спрайтами

Следующее утверждение очевидно, но обладает некими неудобствами: мы отображения 2D мир. Это означает, что все изображения на одной и той же глубине, то есть 0. И вы графический движок не знает, что отображать в первую очередь. Слои спрайтов позволяют нам обозначить, что находится спереди, а что сзади.

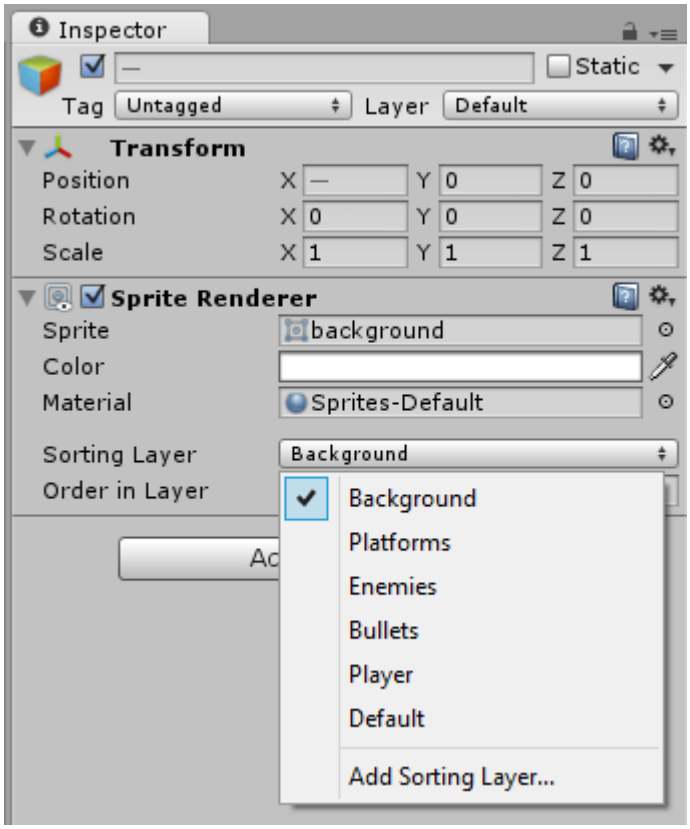
В Unity мы можем изменить "Z" наших элементов, что позволит нам работать со слоями. Это то, что мы делали в этом руководстве перед обновлением до Unity 5, но нам понравилась идея идея использовать слои со спрайтами. У вашего компонента *Sprite Renderer* есть поле с именем *Sorting Layer* с дефолтным значением. Если щелкнуть на нем, то вы увидите:




Давайте добавим несколько слоев под наши нужды (используйте кнопку +):




Добавьте фоновый слой к вашему спрайту фона:

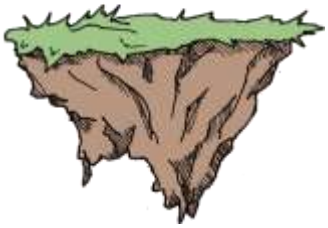



Настройка *Order in Layer* - это способ ограничить подслои. Спрайты с меньшим номером оказываются перед спрайтами с большими числами.

Слой *Default* нельзя удалить, так как это слой, используемый 3D-элементами. Вы можете иметь 3D-объекты в 2D игре, в частности, частицы рассматриваются как 3D-объекты Unity, так что они будут рендериться на этом слое.

Добавление элементов фона

Также известных как *props*. Эти элементы никак не влияют на геймплей, но позволяют усовершенствовать графику игры. Вот некоторые простые спрайты для летающих платформ:



Как видите, мы поместили две платформы в один файл. Это хороший способ научиться обрезать спрайты с помощью новых инструментов **Unity**.

Получение двух спрайтов из одного изображения

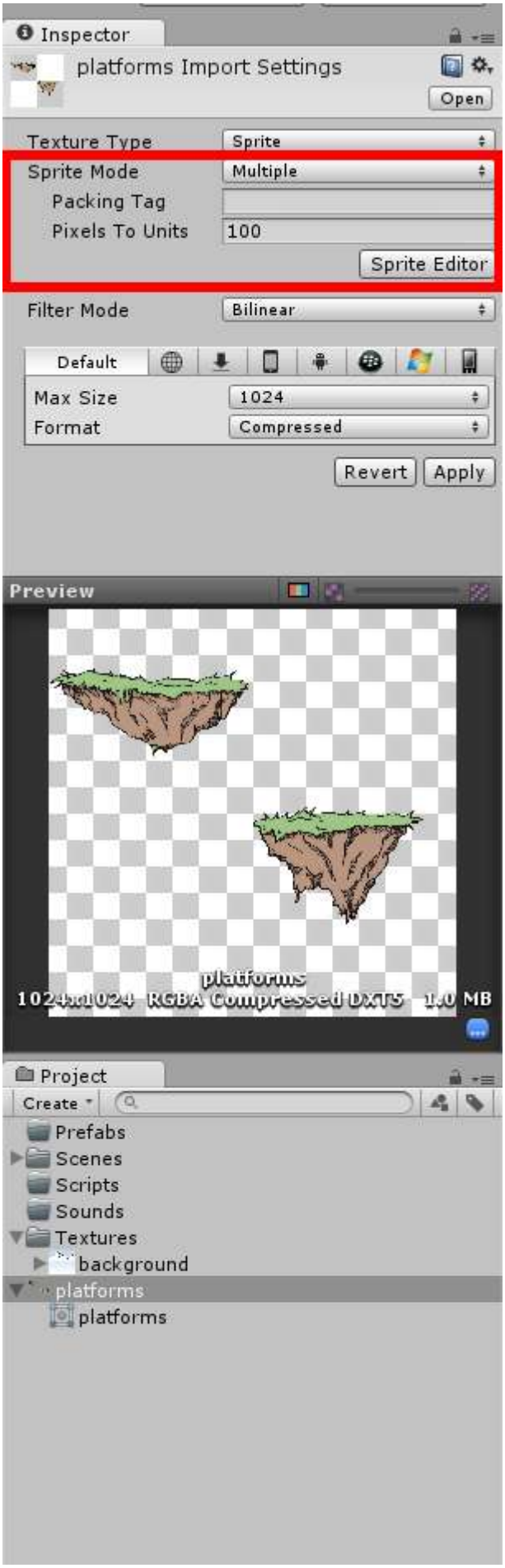
Выполняйте следующие действия:

Импортируйте изображения в папку "Текстуры"

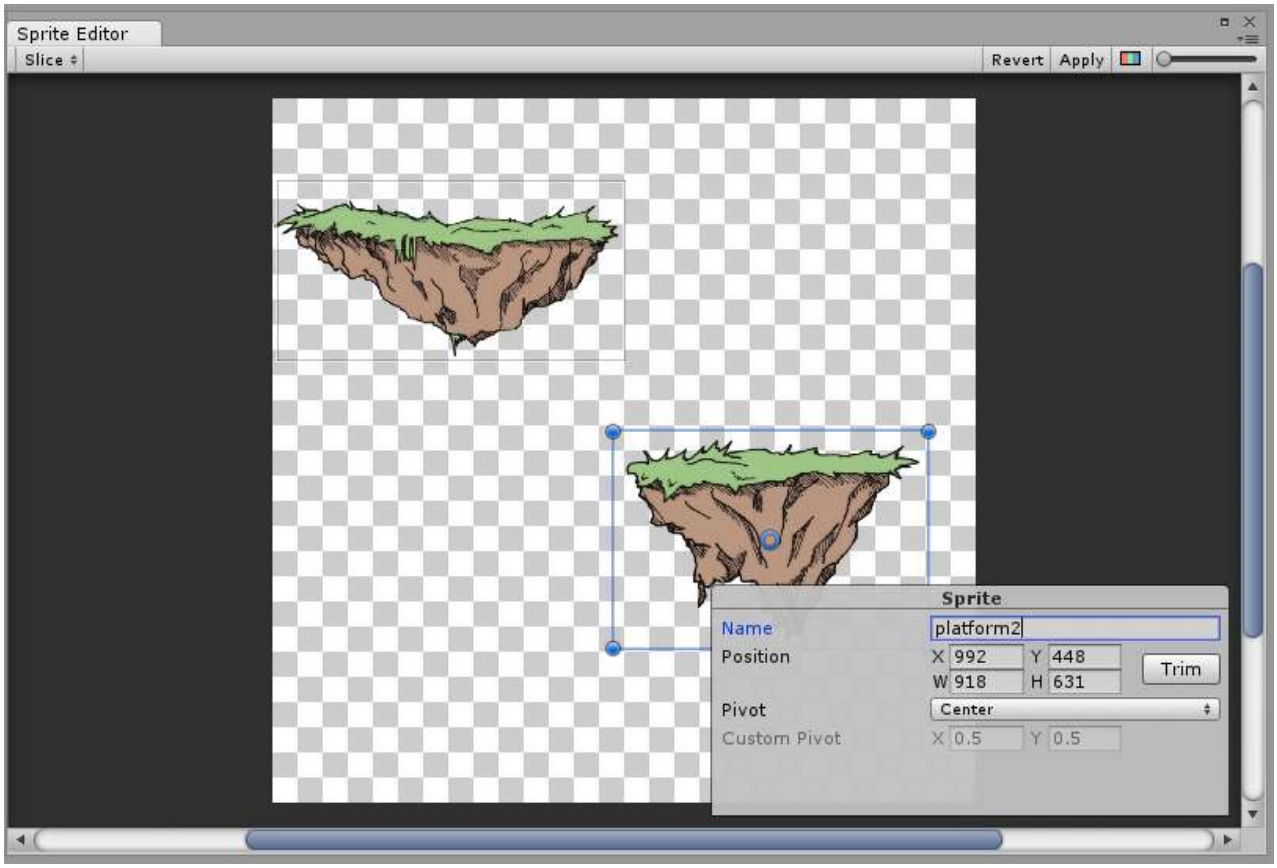
Выберите спрайт Platform и перейдите к панели Инспектор

Измените "Sprite Mode" на "Multiple"

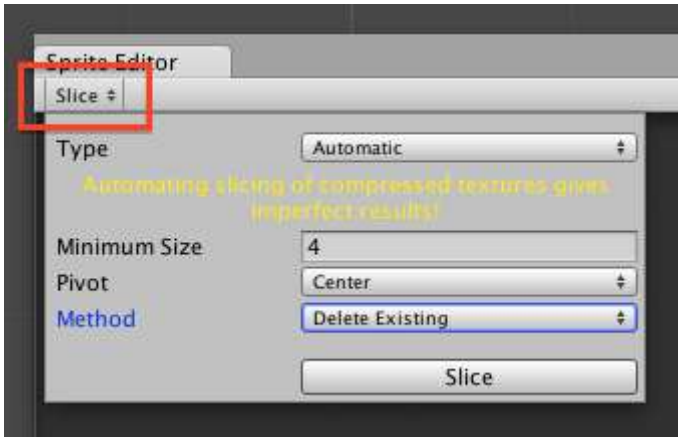
Нажмите на кнопку Sprite Editor (Редактор спрайта)




В новом окне (Sprite Editor) вы можете рисовать прямоугольники вокруг каждой платформы, чтобы разрезать текстуру на более мелкие части:

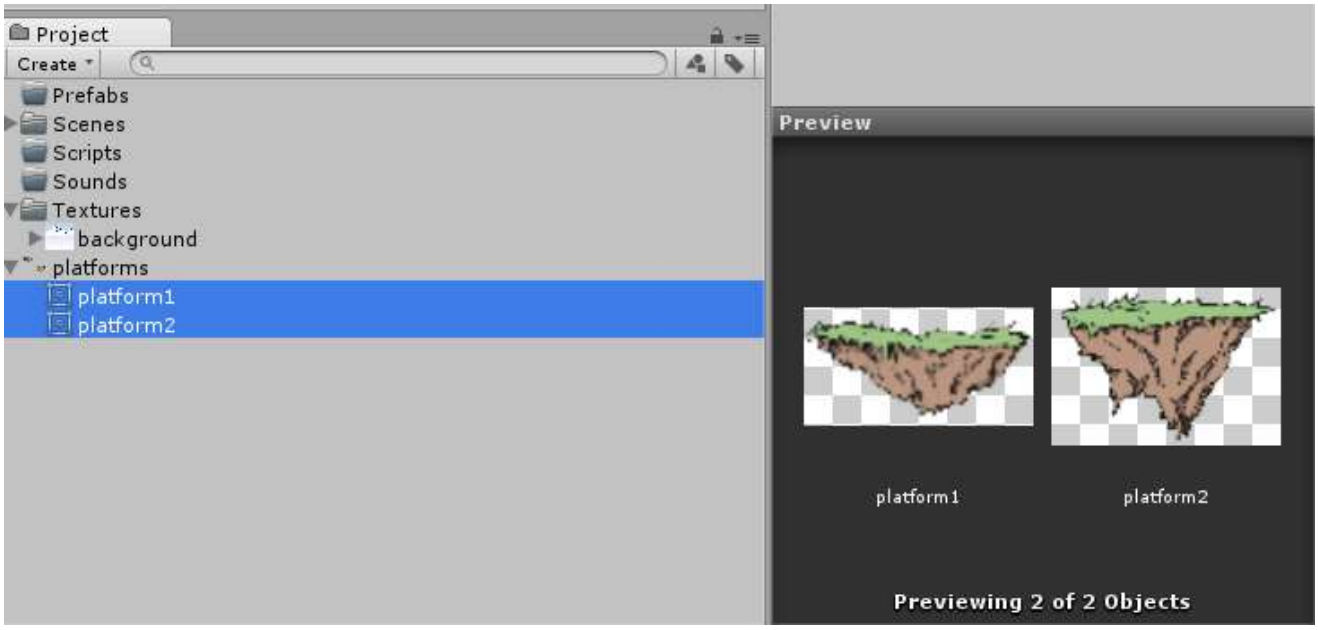



Кнопка Slice в левом верхнем углу позволит вам быстро и автоматически проделать эту утомительную работу:

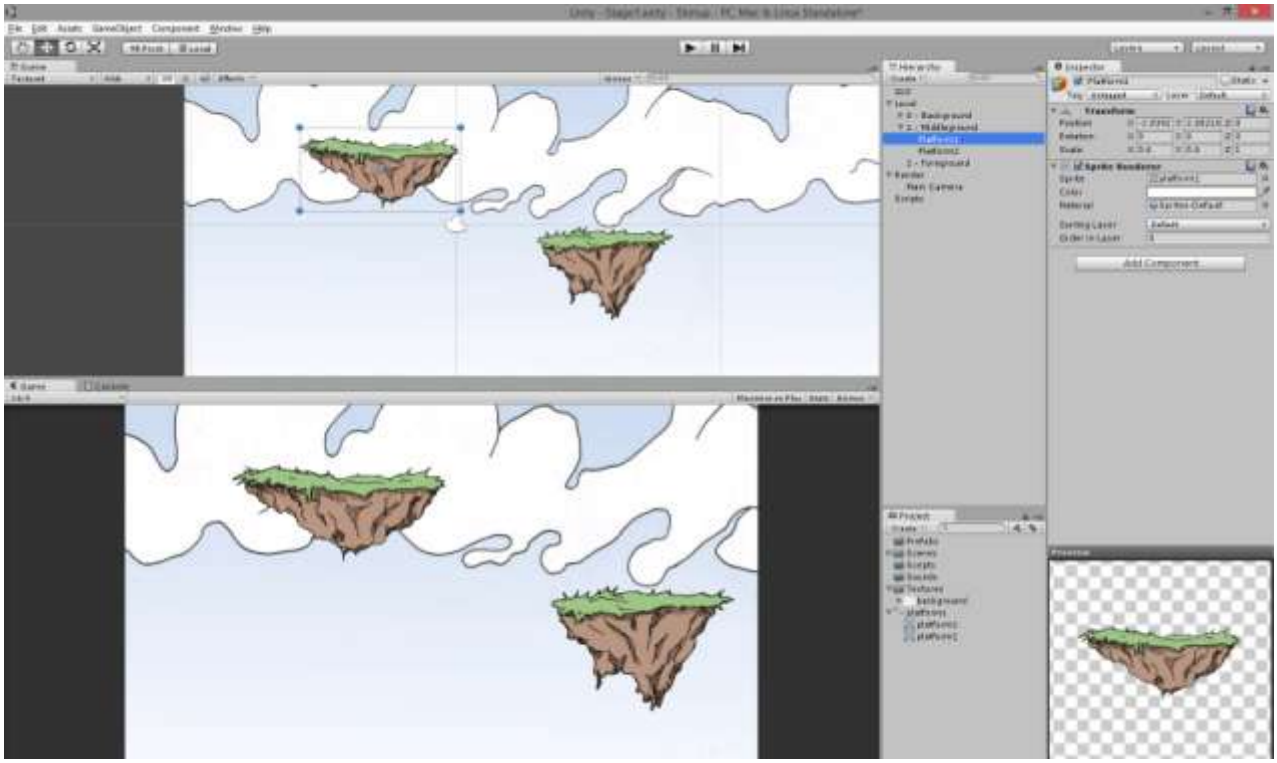



**Unity** найдет объекты внутри изображения и будет нарежет их автоматически. Вы можете установить дефолтное значение для точки вращения или минимальный размер каждого фрагмента. Для простого изображения без артефактов, это необычайно эффективно. Тем не менее, если вы используете этот инструмент, будьте осторожны и проверьте результат, чтобы убедиться, что вы получили то, что хотели.

В этом уроке проделаем эту операцию вручную. Назовите платформы platform1 и platform2. Теперь, под файлом изображения, вы должны увидеть два спрайта отдельно:

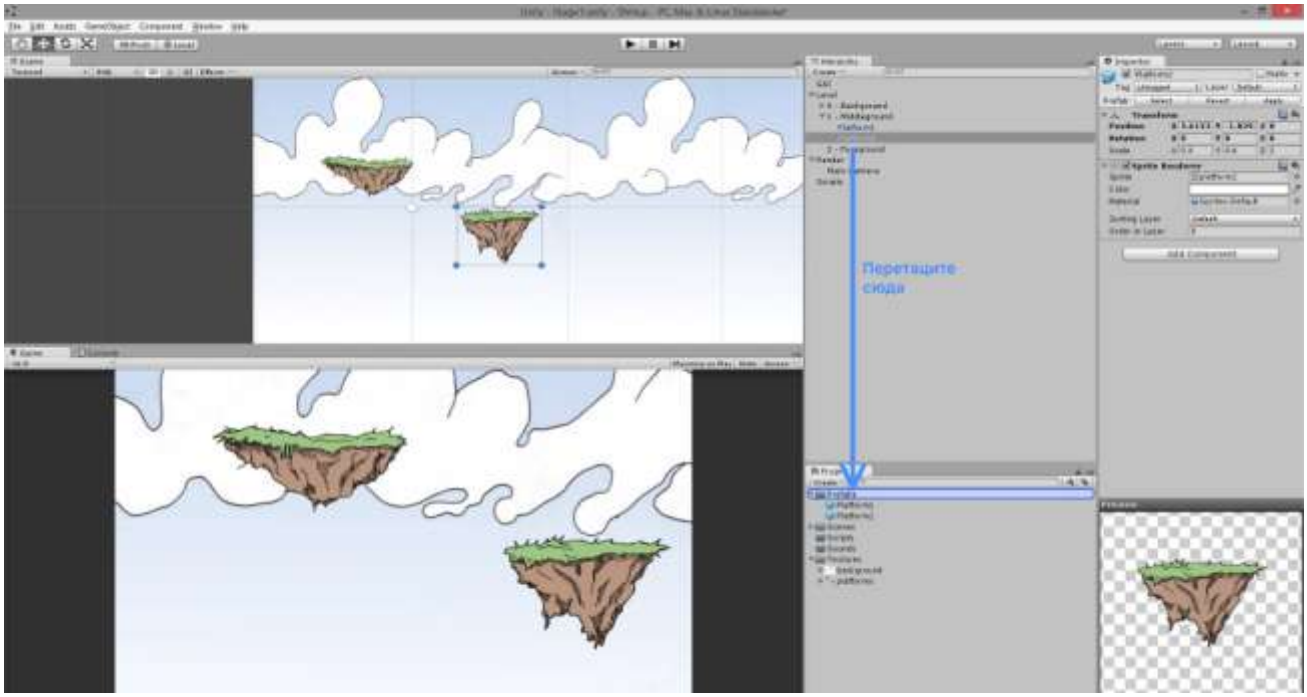



Добавим их в сцену. Для этого мы будем выполнять те же действия что и для фона: создадим новый спрайт и выберим platform1. Потом повторим эти действия для platform2. Поместите их в объект 1 - Middleground. Убедитесь, что их позиция по оси Z равна нулю.

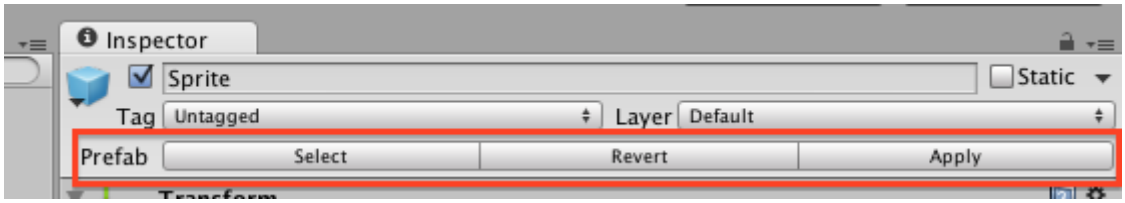


Prefabs (Префабы)

Сохранить эти платформы как префабы. Просто перетащите их в папку Prefabs:




Таким образом вы создадите Prefab, точно отвечающий оригинальному игровому объекту. Вы увидите, что игровой объект, который вы конвертировали в Prefab, представляет собой новый ряд кнопок прямо под его именем:




Заметка о кнопках "Prefab": При последующей модификации игрового объекта, вы можете использовать кнопку "Apply", чтобы применить эти изменения к Prefab, или кнопку "Revert", чтобы отменить все изменения игрового объекта в свойстваъ Prefab. Кнопка "Select" переместит выбранные свойства в ассет Prefab в окне проекта (они будут выделены).

Создание префабов с объектами-платформами упростит их повторное использование. Просто перетащите **Prefab** на сцену, чтобы добавить копию. Попробуйте добавить другую платформу таким же образом.

Теперь вы можете добавить больше платформ, меняющих свои координаты, размеры и плоскости (вы можете поместить их на заднем или переднем плане, просто установите координату Z для платформы на 0).

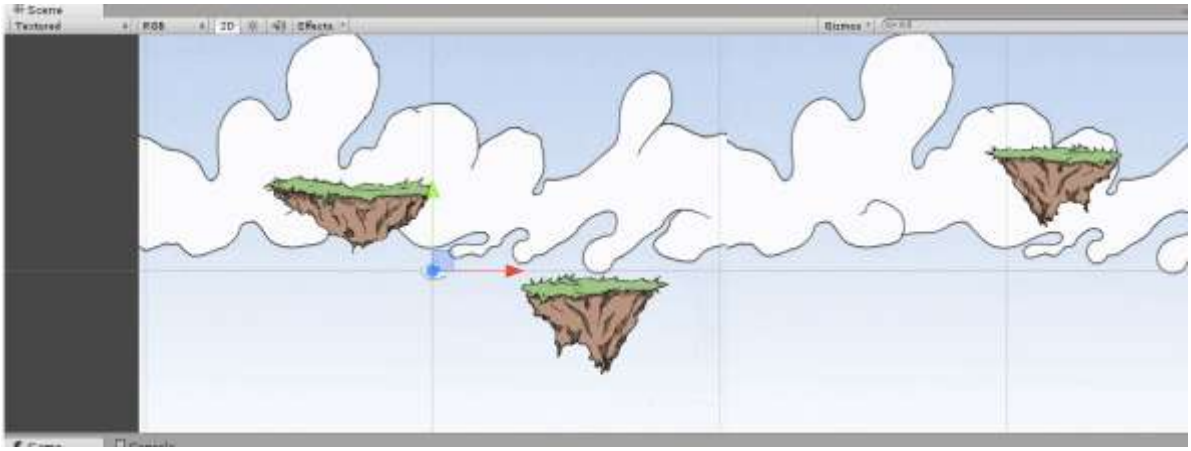
На данном этапе все это выглядит еще сыроватым, но в следующих двух главах мы добавим параллаксный скроллинг, и сцена оживет у нас на глазах.

Слои

Прежде чем двигаться дальше, мы модифицируем наши слои, чтобы избежать каких-либо проблем с порядком их отображения. Для этого просто измените позицию игровых объектов по оси Z во вкладке **Hierarchy** (Иерархия) следующим образом:

| Слой | Позиционирование по оси Z |
|---|---|
| 0 - Задний фон | 10 |
| 1 - Средний фон | 5 |
| 2 - передний фон | 0 |

При переключении из 2D режима в 3D, в окне "Scene" (Сцена) вы будете четко видеть слои:



Кликнув на игровом объекте Main Camera, вы увидите, что флажок Projection установлен на Orthographic. Эта настройка позволяет камере визуализировать 2D игру без учета трехмерных свойств объектов. Имейте в виду, что даже если вы работаете с 2D объектами, Unity по-прежнему использует свой 3D движок для визуализации сцены. Рисунок выше это наглядно демонстрирует.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

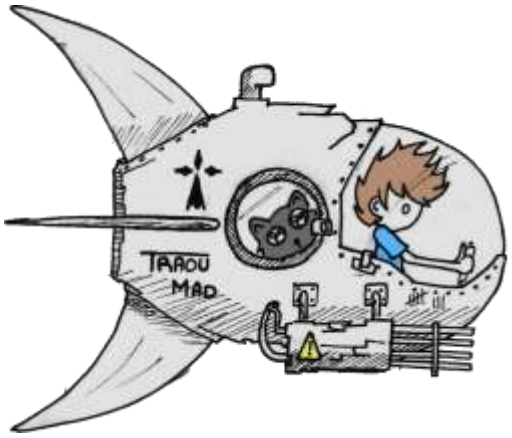
**Практическое занятие № 3. Создание игрока в Unity**

**Выполнив работу, Вы будете:**
***уметь:***
-создавать персонажа под управлением игрока
-задавать управления персонажем при помощи клавиатуры
**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Создать объект, контролируемый игроком
2. Создать управление, для этого объекта

**Порядок выполнения работ:**

Создание объекта, контролируемого игроком, требует наличия определенных элементов: спрайт, способ управления им и способ его взаимодействия с игровым миром. Рассмотрим этот процесс шаг за шагом и начнем, пожалуй, со спрайта. Вот изображение, которое мы будем использовать:




Скорируйте картинку в папку "Textures"

Создайте новый спрайт и назовите его "Player"

Настройте спрайт так, чтобы он отображался в свойстве "Sprite" компонента "Sprite Renderer"

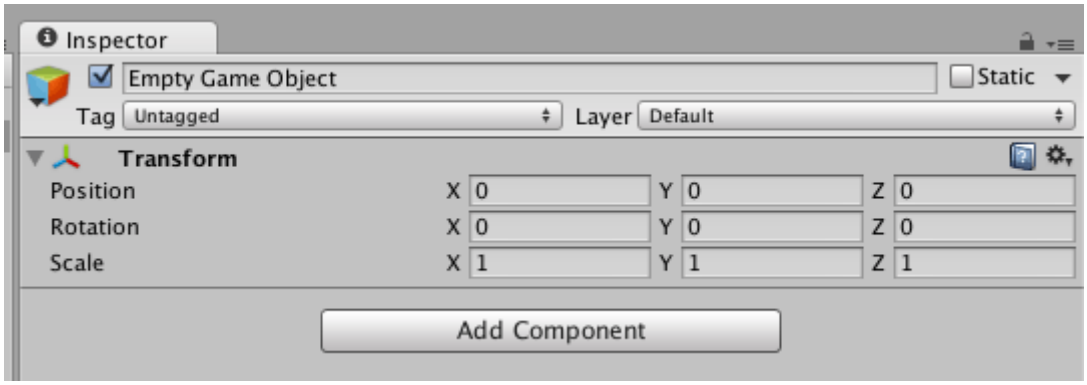
Если у вас возникли проблемы, обратитесь к предыдущему уроку. Мы проделали точно такие же действия для фона и "реквизита".

Поместите игрока в слой "2 - Foreground"

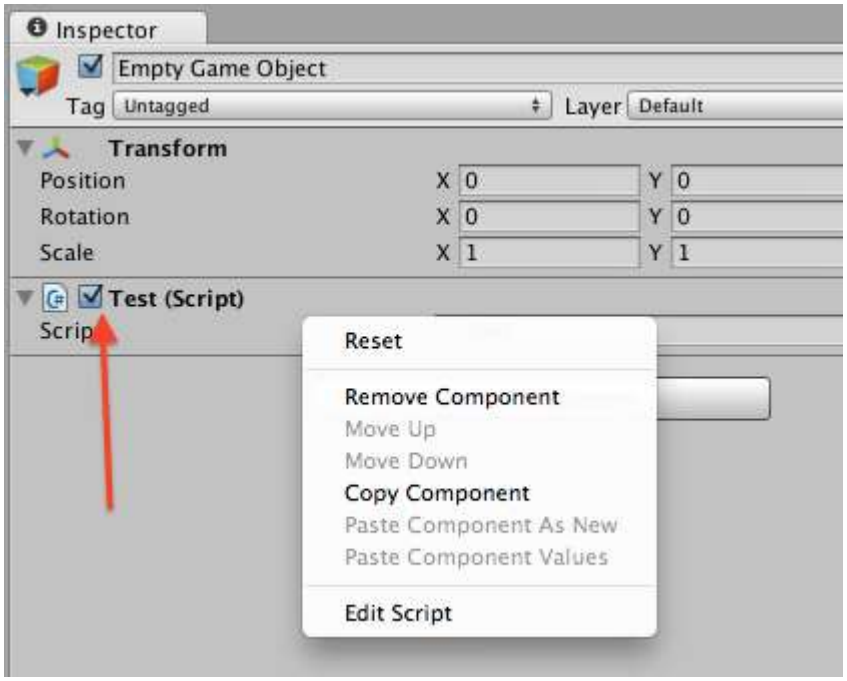
Измените масштаб на (0.2, 0.2, 1)

Теперь несколько слов о компонентах. Мы только что говорили о компоненте "Sprite Renderer". Если вы еще не заметил, объект игры состоит из нескольких компонентов, видимых в панели "Инспектор".

По умолчанию пустой объект игры выглядит так:




Этот объект имеет только один компонент: Transform. Этот компонент является обязательным и не может быть отключен или удален. Вы можете добавить к объекту столько компонентов, сколько захотите. Например, скрипты добавляются в качестве компонента. Большинство компонентов может быть включено или отключено пока существует объект.




*(Вы можете нажать на галочку чекбокса, чтобы отключить его. Вы можете щелкнуть правой кнопкой мыши на компоненте, чтобы вернуть прежнее свойство, удалить его и т.д.*

Компоненты могут взаимодействовать с другими компонентами. Если объект имеет сомпонент, который требуется другому компоненту объекта для работы, вы можете просто перетащить весь объект внутрь этого компонента, и тогда компонент сам найдет все, что ему нужно.

*Sprite Renderer* является компонентом, который способен отображать спрайт-текстуру. Теперь, когда мы узнали о концепции компонента, давайте добавим один к игроку!

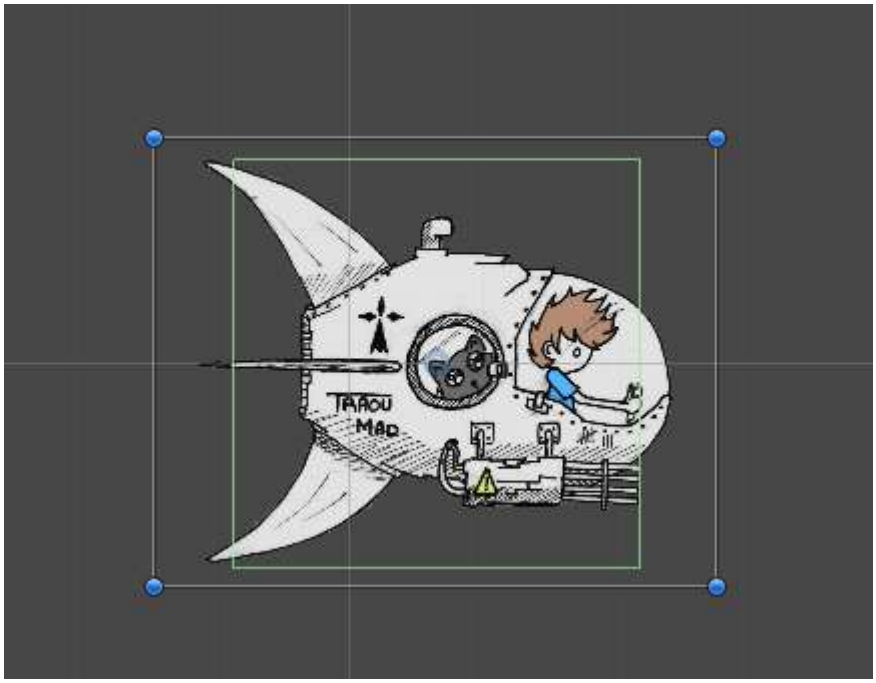
Добавляем бокс-коллайдер (Box Collider)

Нажмите на кнопку "Добавить компонент" объекта игрока. Выберите "Box Collider 2D". Вы можете увидеть коллайдер в редакторе "Сцена" зрения и настройки его размера в "Инспекторе" в поле "Размер" (Size).

Существует еще один способ редактирования бокс-коллайдера. Выберите игровой объект с помощью бокс-коллайдера и зажмите клавишу shift на клавиатуре. Вы увидите, что на бокс-коллайдере (*зеленый прямоугольник* ) появились четыре маленьких рычажка. Перетащите один из них, чтобы изменить форму бокс-коллайдера. Будьте осторожны, *синий прямоугольник представляет собой компонент* Transform вашего игрового объекта, а не коллайдер.

Мы будем устанавливать размер коллайдера равным (10, 10).

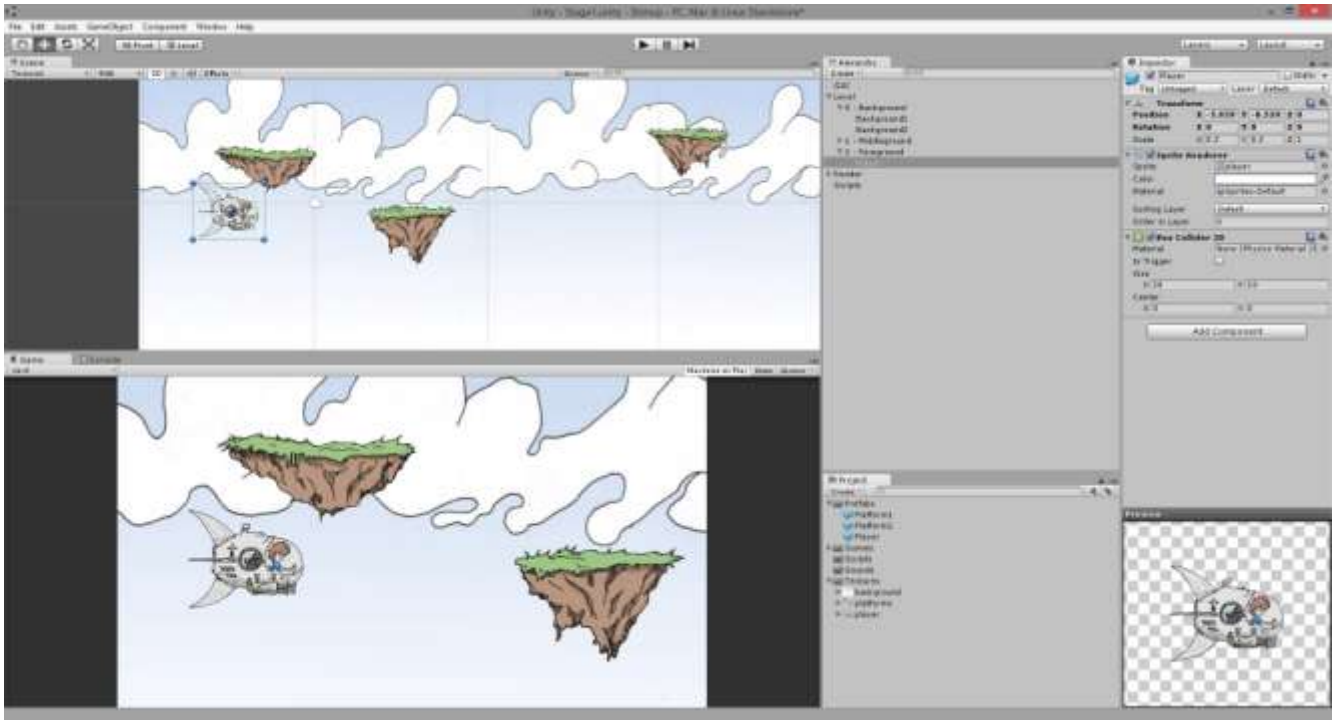
Это слишком много для настоящего *шмапа,* но все же меньше, чем спрайт:



В настоящее время, этого вполне достаточно.

*Совет*: Если вы планируете создать *шмап*, вам придется уделить много времени настройке хитбоксов – они должны точно соответствовать маленькому элементу внутри игрового спрайта. Вы также можете изменить такой параметр коллайдера, как shape – например, с помощью "Circle Collider 2D". Благодаря Unity, его поведение при этом не меняется, но это позволяет немного улучить геймплей.

Сохраним объект игрок как префаб. Теперь у вас есть базовую сущность игрока!



2D полигональный коллайдер

Если вы хотите супер точный и произвольный формы хитбокс, воспользуйтесь компонентом Unity "Полигоннальный коллайдер 2D" (Polygon Collider 2D). Эффект от этого будет незначительный, но зато вы получите такую форму, какую вы хотите.

"Polygon Collider 2D" похож на остальные коллайдеры: вы можете изменять форму с помощью мышки в режиме "Scene". Для удаления точки зажмите cmd или ctrl, а чтобы отрегулировать положение точки или добавить ее в форму коллайдера, используйте shift

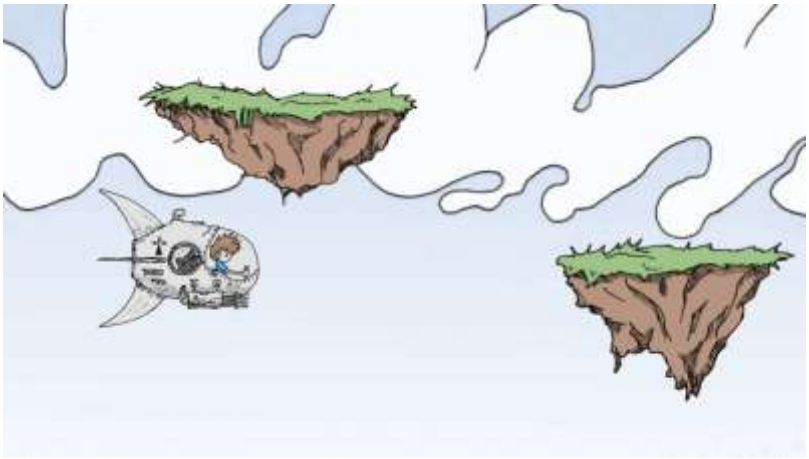
Магия Rigidbody

Последний компонент, необходимый для добавления на нашего игрока: "Rigidbody 2D". Это поможет физическому движку правильно задействовать объект в игровом пространстве. Более того, это позволит вам использовать столкновения в скрипте.

Выберите объект Player в "Hierarchy".

Добавьте компонент "Rigidbody 2D".

Теперь, нажмите кнопку "играть" и смотрите, что у нас вышло:

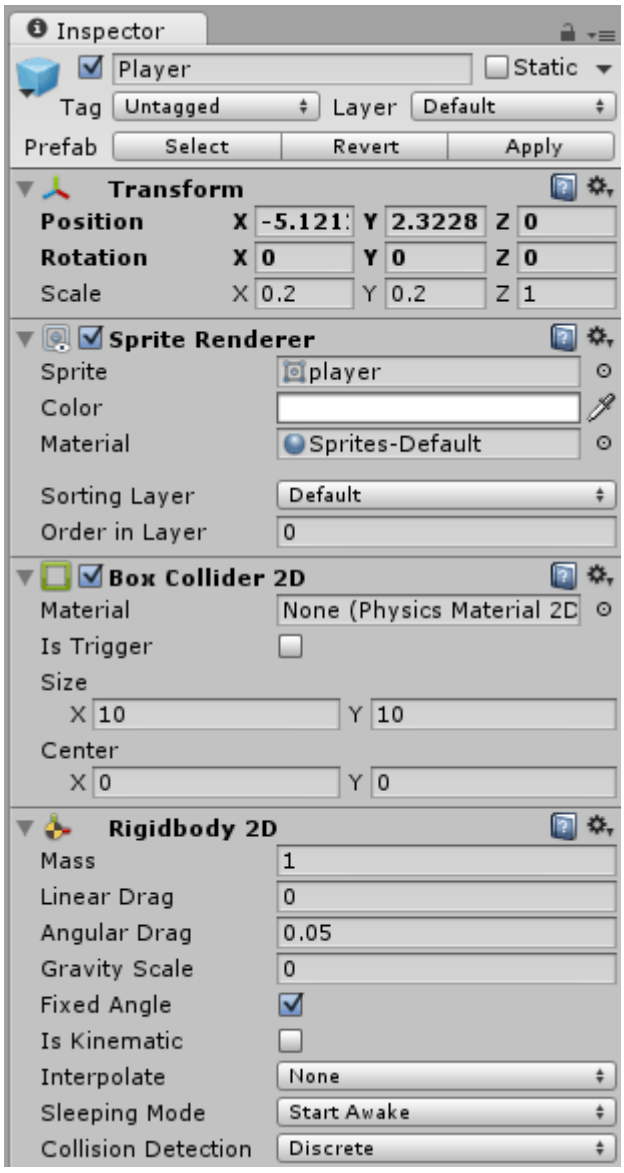


Корабль падает! И как падает! Передвайте привет нашей любимой силе тяжести. По мере того, как сменяются кадры с заранее заданной гравитацией и rigidbodies прибавляет объекту массы, корабль притягивается к нижней части экрана.

По-умолчанию, ускорние свободного падения в Unity равно 9.81, т.е. мы имеем дело с земной гравитацией.

Гравитация может быть использована в любой игре, но нам она не нужна. К счастью, гравитацию на Rigidbody можн легко отключить. Просто установите "гравитационный масштаб" равным нулю. Вот и все, корабль снова летит. Не забудьте поставить галочку в окошке "Fixed Angles", чтобы предотвратить вращение корабля, обусловленное такой физикой.

Окончательные настройки:




Перемещение игрока

Настало время написать скриптик (вы ведь не думали, что все будет двигаться само)? Создайте в Unity C#-скрипт в папке "Scripts" и назовите это "PlayerScript". Откройте ваш любимый редактор или используйте подменю "Sync" (нажмите на "Assets" в строке меню, затем на "Sync MonoDevelop Project") для правки созданного Unity скрипта.

*"Sync MonoDevelop Project"*: Это подменю немного странное.Во-первых, невозможно изменить имя, даже если сменить редактора.
Мы также рекомендуем использовать это меню при создании первого скрипта, так как Unity создаст решения и привяжет их к библиотекам Unity (для Visual Studio, Xamarin Studio или MonoDevelop).

Если вместо этого вы просто откроете скрипт, компилятор вашего IDE, скорее всего, зарегистрирует определенные ошибки, не Unity. Это не имеет значения, потому что вам не придется использовать его напрямую, но функция автоматического завершения объектов Unity не помешает.

По умолчанию в скрипте уже прописаны методы Start и Update. Вот краткий список наиболее часто используемых функций:

Awake() вызывается один раз, когда объект создается. По сути аналог обычной функции-конструктора.

Start() выполняется после Awake(). Отличается тем, что метод Start() не вызывается, если скрипт не включен (remember the checkbox on a component in the "Inspector").

Update() выполняется для каждого кадра in the main game loop.

FixedUpdate() вызывается каждый раз через определеннок число кадров. Вы можете вызывать этот метод вместо Update() когда имеете дело с физикой ("RigidBody" и др.).

Destroy() вызывается, когда объект уничтожается. Это ваш последний шанс, чтобы очистить или выполнить код.

У вас также есть некоторые функции для обработки столкновений:

OnCollisionEnter2D(CollisionInfo2D info)выполняется, когда коллайдер объекта соприкасается с другим коллайдером.

OnCollisionExit2D(CollisionInfo2D info) выполняется, когда коллайдер объекта не соприкасается ни с одним другим коллайдером.

OnTriggerEnter2D(Collider2D otherCollider)выполняется, когда коллайдер объекта соприкасается с другим коллайдером с пометкой "Trigger".

OnTriggerExit2D(Collider2D otherCollider) выполняется, когда коллайдер объекта перестает соприкасаться с коллайдером, помеченным как "Trigger".

Итак, с теорией покончено, пора в бой. Или нет, погодите еще немного: обратите внимание, что почти все, о чем мы говорили с вами имеет, суффикс "2D". Box Collider 2D, a Rigidbody 2D, OnCollisionEnter2D, OnTriggerEnter2D и т.д. Эти новые компоненты или методы появились с Unity 4.3. Используя их, вы работаете с физическим движком, встроенным в Unity 4.3, для 2D-игр (на основе Box2D) вместо движка для 3D-игр (PhysX). Два движка имеют аналогичные концепции и объекты, но они не работают точно так же. Если вы начинаете работать с одним (например, Box2D для 2D-игр), придерживаqntcm его. Именно поэтому мы используем все объекты или методы с суффиксом "2D".

В скрипт для нашего игрока мы добавим несколько простых элементов управления, а именно: клавиши со стрелками, которые будут перемещать корабль.

```
using UnityEngine;

///

/// Контроллер и поведение игрока
///

public class PlayerScript : MonoBehaviour
{
  ///

  /// 1 - скорость движения
  ///

  public Vector2 speed = new Vector2(50, 50);

  // 2 - направление движения
  private Vector2 movement;

  void Update()
  {
    // 3 -  извлечь информацию оси
    float inputX = Input.GetAxis("Horizontal");
    float inputY = Input.GetAxis("Vertical");

    // 4 - движение в каждом направлении
    movement = new Vector2(
      speed.x * inputX,
      speed.y * inputY);

  }

  void FixedUpdate()
  {
    // 5 - перемещение игрового объекта
    rigidbody2D.velocity = movement;
  }
}
```

Поясню цифры в комментариях к коду:

Сначала определим публичную переменную, которая будет отображаться в окне "Инспектор". Это скорость, используемая для корабля.

Сохраним движение для каждого кадра.

Используем дефолтную ось, которую можно отредактировать в "Edit" -> "Project Settings" -> "Input". При этом мы получим целые значения между [-1, 1], где 0 будет означать, что корабль неподвижен, 1 - движение вправо, -1 - влево.

Умножим направление на скорость.

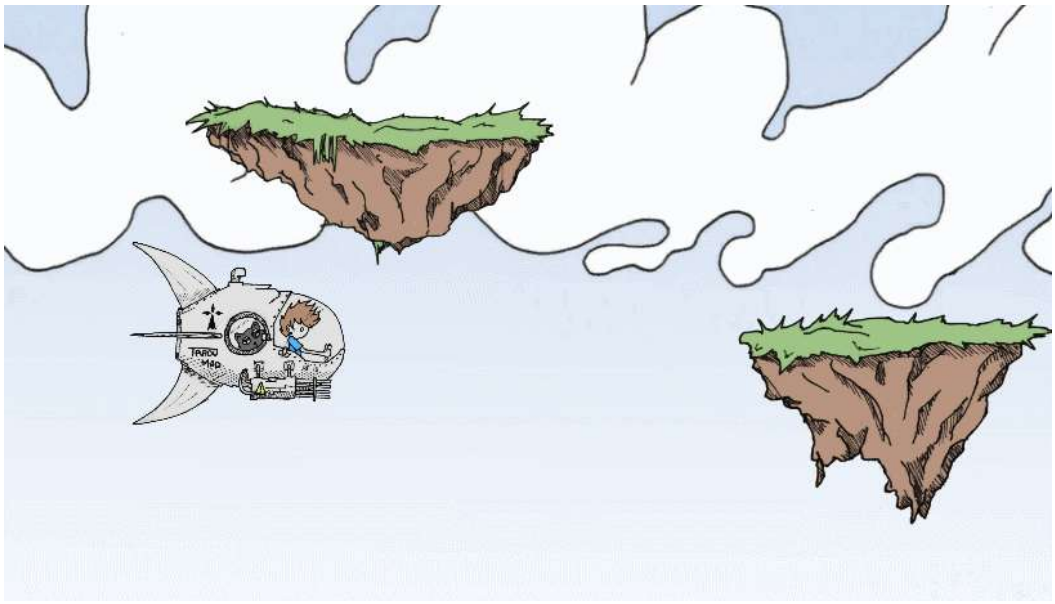
Изменим скорость rigidbody. Это даст движку команду к перемещению объекта. Сделаем это в FixedUpdate(), предназначенном для всего, что связано с физикой.

*Заметка о соглашениях C#* : Посмотрите на видимость speed члена класса – он обозначен как публичный. В C# переменная члена класса должна быть приватной для соответствующего

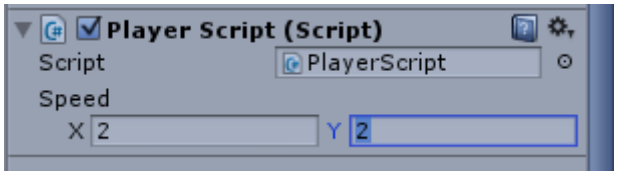
сохранения его внутренней репрезентации. Но смена типа переменной на публичный позволяет редактировать ее в Unity через панель "Inspector", даже в процессе игры. Это одна из самых *мощных* возможностей Unity, позволяющая изменять геймплей без использования кода. Помните, что в данном случае мы создаем скрипты, а это не то же самое, что классическое программирование на C#. Это предполагает некоторых правил и соглашений.

Теперь добавим скрипт к игровому объекту. Для этого перетащите скрипт из окна "Проект" (Project) на игровой объект в "Иерархии" (Hierarchy). Вы также можете нажать на "Add Component" и добвить его вручную.

Нажмите кнопку "Play" в верхней части окна редактора. Корабль движется! Congratulations, Вы только что сделали эквивалент "Hello, World!" для игры :)



Попробуйте настроить скорость: нажмите на игрока, измените значения скорости в "Инспекторе", и посмотрите что из этого получится.




Будьте осторожны: изменения параметров, сделанные во время игры теряются, когда вы ее остановите! Инспекторе - это отличный инструмент для настройки геймплея, но запомните или запишите, что вы делали, если хотите сохранить изменения. Этот же трюк подходит, если вы хотите проверить что-то новое, но не хотите вносить изменения в реальный проект.

Первый враг

Теперь добавим неприятелей, стремящихся уничтожить наш корабль. Пусть им будет зловещий спрут, названный "Poulpi":

Создадим новый спрайт. Для этого:

Скопируйте картинку в папку "Textures".

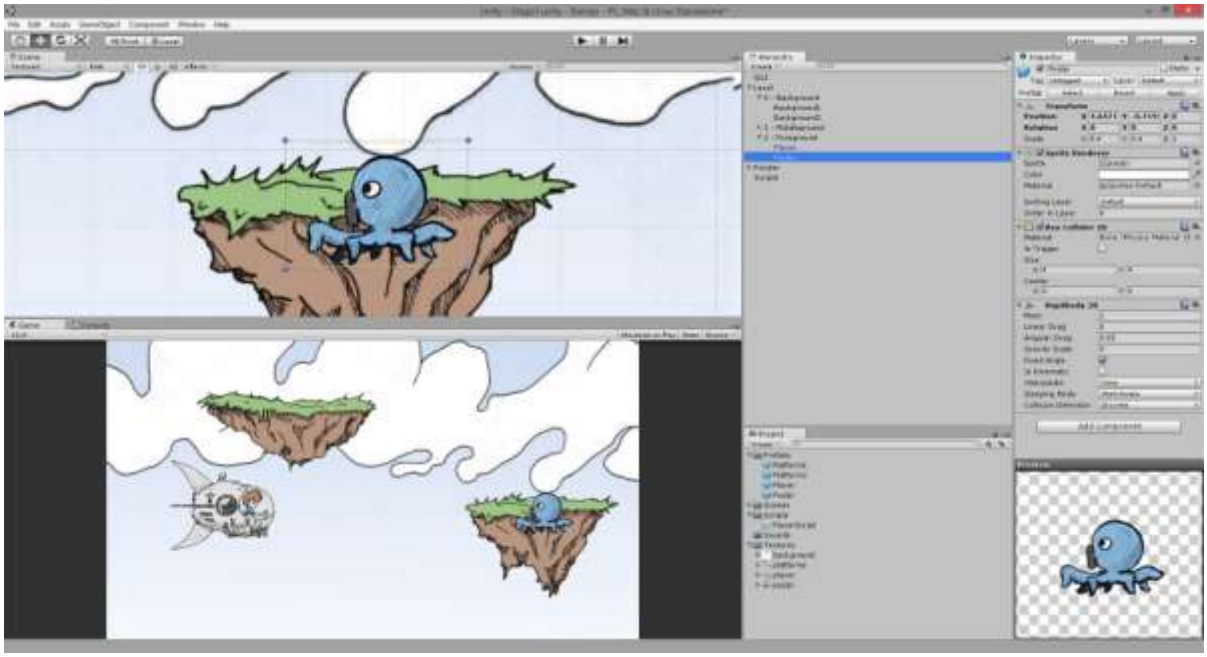
Создайте новый спрайт, используя это изображение.

Измените свойство "Масштаб" (Scale) в разделе Трансформирование (Transform) на (0.4, 0.4, 1).

Добавьте "Box Collider 2D" размером (4, 4).

Add a "Rigidbody 2D" with a "Gravity Scale" of 0 and "Fixed Angles" ticked.

Сохраните префаб и... вуаля!



Скрипт

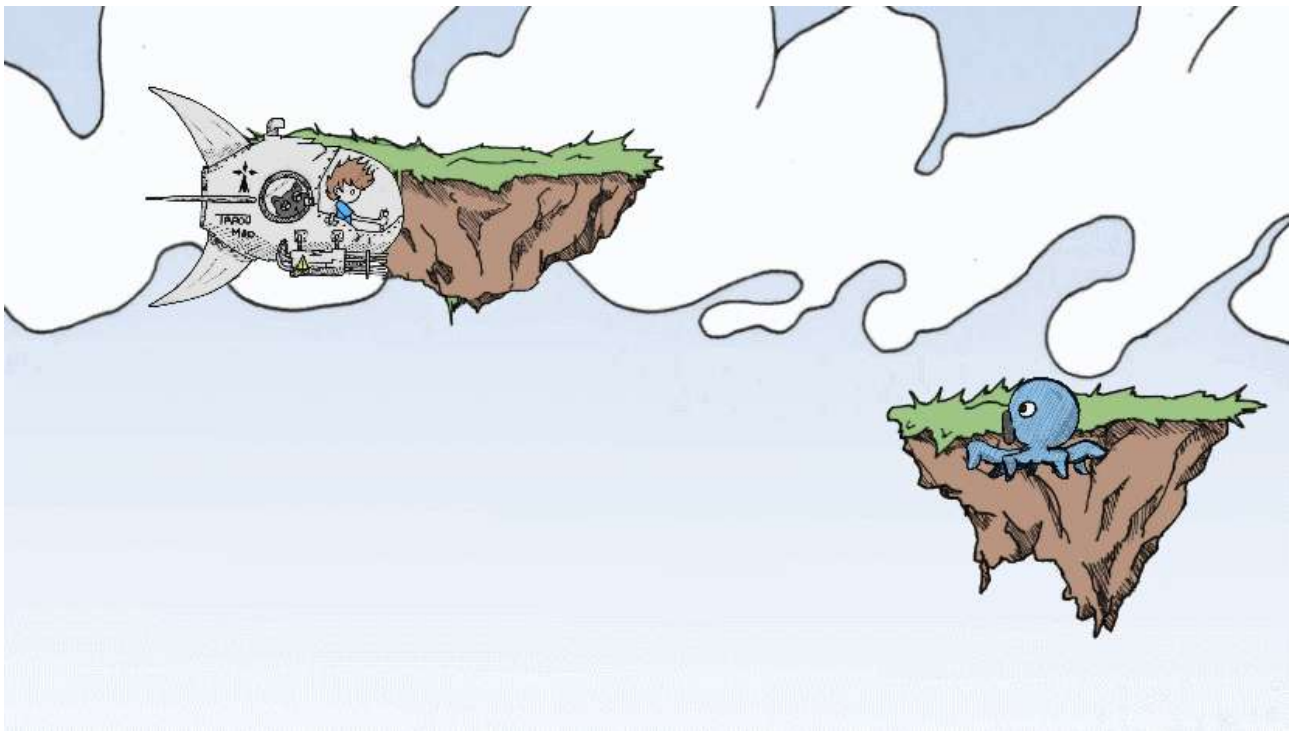
Теперь напишем простенький скрипт, отвечающий за движение осьминога в определенном направлении. Для этого создайте новый скрипт, назвав его "MoveScript".

Модульность обеспечивается системой на основе компонентов Unity. Это отличный способ отделить друг от друга скрипты с различными функциями. Конечно, вы можете написать один гигантский скрипт с большим количеством параметров. Это ваш выбор, но я настоятельно не рекомендую вам делать это.

Скопируем некоторые части кода, который мы написали в «PlayerScript" для движения персонажа. We will add another designer (a public member you can alter in the "Inspector") variable for the direction:

```
using UnityEngine;

/// <summary>
/// Просто перемещает текущий объект игры
/// </summary>
public class MoveScript : MonoBehaviour
{
  // 1 - переменные

  /// <summary>
  /// Скорость объекта
  /// </summary>
  public Vector2 speed = new Vector2(10, 10);

  /// <summary>
  /// Направление движения
  /// </summary>
  public Vector2 direction = new Vector2(-1, 0);

  private Vector2 movement;

  void Update()
  {
    // 2 - Перемещение
    movement = new Vector2(
      speed.x * direction.x,
      speed.y * direction.y);
  }

  void FixedUpdate()
  {
    // Применить движение к Rigidbody
    rigidbody2D.velocity = movement;
  }
}
```

Прикрепите скрипт к осьминогу. Нажмите "Play" и убедитесь, что спрут движется так, как показано на рисунке ниже:



Если вы будете перемещать игрока перед врагом, оба спрайта столкнутся. Они просто заблокируют друг друга, так как мы еще не определили их поведение при столкновении.

**Форма представления результата:**

Отчет о проделанной работе.

**Критерии оценки:**

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Практическое занятие № 4. Создание оружия и боеприпасов**

**Выполнив работу, Вы будете:**
***уметь:***
- создавать пули и оружие у игрока и противника

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Реализовать механику выстрелов у игрока и противника
2. Реализовать механику столкновения пули с противником и игроком

**Порядок выполнения работ:**

**Снаряд**

Первым делом, прежде чем позволить игроку стрелять, нам нужно определить игровой объект, представляющий используемые нами снаряды. Вот наш спрайт

Снаряд – объект, которым мы будем пользоваться очень часто. В игре будет несколько сцен, в которых игрок будет стрелять. Что мы должны использовать в этом случае? Префаб (Prefab), конечно же! Для этого проделайте следующие действия:

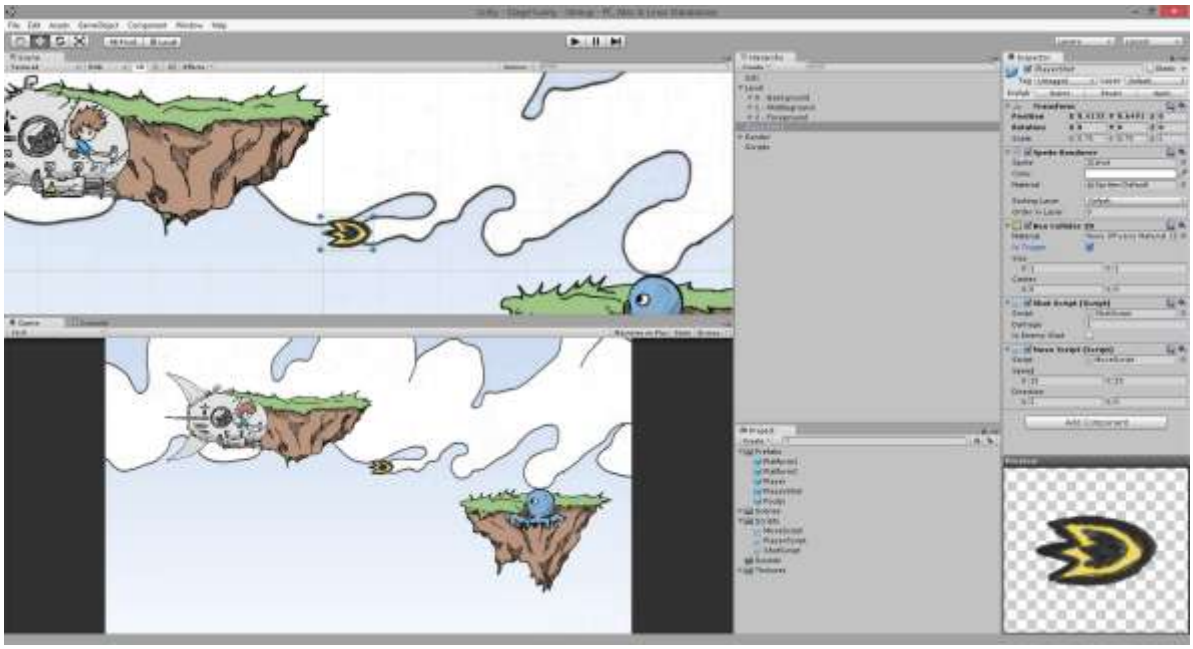
1. Импортируйте текстуру
2. Создайте новый спрайт в сцене
3. Установите изображение на спрайт.
4. Добавьте "Rigidbody 2D" с равным нулю значениями "Gravity Scale" и "Fixed Angles".
5. Добавьте "Box Collider 2D" размером (1, 1).

Сделайте масштаб таким (0,75, 0,75, 1) для лучшего отображения. Теперь, нам нужно установить новый параметр в "Инспекторе" (Inspector). для этого в "Box Collider 2D" поставьте галочку напротив свойства "IsTrigger". Триггер коллайдера создает событие при столкновении, но не используется при моделирования физики. Это значит, что выстрел пройдет сквозь объект при соприкосновении — *никакого «реального» взаимодействия не будет*. А вот у другого коллайдера это спровоцирует событие "OnTriggerEnter2D".

Создайте скрипт, назвав его "ShotScript":

```
using UnityEngine;

/// <summary>
/// Поведение снаряда
/// </summary>
public class ShotScript : MonoBehaviour
{
  // 1 - Переменная дизайнера

  /// <summary>
  /// Причиненный вред
  /// </summary>
  public int damage = 1;

  /// <summary>
  /// Снаряд наносит повреждения игроку или врагам?
  /// </summary>
  public bool isEnemyShot = false;

  void Start()
  {
    // Ограниченное время жизни, чтобы избежать утечек
    Destroy(gameObject, 20); // 20 секунд
  }
}
```

Прикрепите "ShotScript" к спрайту. Также добавьте "MoveScript", т.к. ваши снимки будут двигаться. Теперь перетащите объект выстрел в панель "Проект" для создания Префаба. Он нам совсем скоро понадобится. Вы должны иметь следующую конфигурацию:



Если вы запустите игру с помощью кнопки "Play", вы увидите, что выстрел движется.

**Столкновения и повреждения**

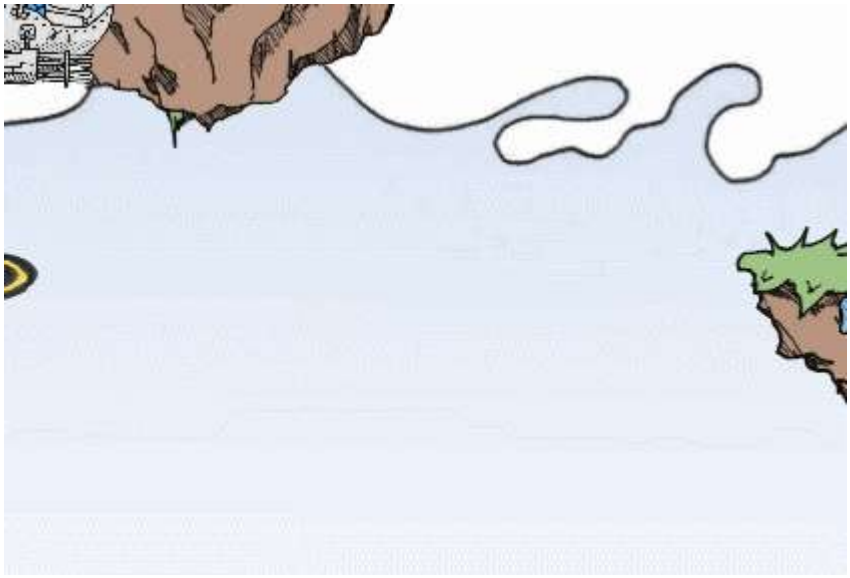
Тем не менее, выстрел (пока) не наносит повреждений. Ничего удивительного, ведь мы не сделали скрипт обработки повреждений. Создадим его, назвав "HealthScript":

```
using UnityEngine;

/// <summary>
/// Handle hitpoints and damages
/// </summary>
public class HealthScript : MonoBehaviour
{
  /// <summary>
  /// Всего хитпоинтов
  /// </summary>
  public int hp = 1;

  /// <summary>
  /// Враг или игрок?
  /// </summary>
  public bool isEnemy = true;

  /// <summary>
  /// Наносим урон и проверяем должен ли объект быть уничтожен
  /// </summary>
  /// <param name="damageCount"></param>
  public void Damage(int damageCount)
  {
    hp -= damageCount;

    if (hp <= 0)
    {
      // Смерть!
      Destroy(gameObject);
    }
  }

  void OnTriggerEnter2D(Collider2D otherCollider)
  {
    // Это выстрел?
    ShotScript shot = otherCollider.gameObject.GetComponent<ShotScript>();
    if (shot != null)
    {
      // Избегайте дружественного огня
      if (shot.isEnemyShot != isEnemy)
      {
        Damage(shot.damage);

        // Уничтожить выстрел
        Destroy(shot.gameObject); // Всегда цельтесь в игровой объект, инач
      }
    }
}
```

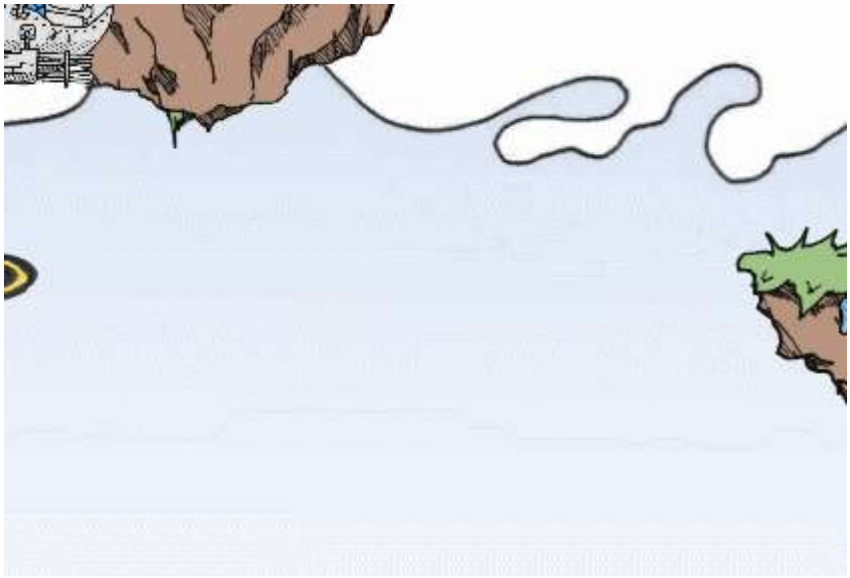
Добавьте "HealthScript" на префаб спрута.

*Внимание*: Лучше всего поработать непосредственно с префабом. При этом каждый экземпляр врага, участвующий в сцене, будет модифицирован так, чтобы отражать префаб. В данном случае это особенно важно, потому что в нашей сцене будет много врагов. Если вы сосредоточили усилия на экземпляр игрового объекта вместо префаба, не волнуйтесь: нажав на кнопку "Применить" сверху вкладки "Инспектор", вы добавите эти изменения и в префаб.

Убедитесь, что выстрел и спрут находятся на одной линии, чтобы проверить столкновение. Напоминаю, что 2D движок ничего не знает про ось Z, поэтому ваши 2D коллайдеры всегда будут в той же плоскости. А теперь, запустите нашу сцену. Вы должны увидеть следующее:



Здоровье врага превосходит урон от выстрела, поэтому он выживет. Попробуйте изменить значение hp в "HealthScript" врага:



**Стрельба**

Удалите выстрел из сцены. Теперь, когда мы с ним закончили, ему нечего там делать. Нам нужен новый скрипт для стрельбы. Создайте его под именем "WeaponScript". Этот скрипт мы будем использовать везде (игроки, враги и т.д.) Его цель заключается в to instantiate снаряда перед игровым объектом, к которому он привязан. Вот полный код, больше, чем обычно. Объяснения ниже:

```
using UnityEngine;

/// <summary>
/// Launch projectile
/// </summary>
public class WeaponScript : MonoBehaviour
{
  //--------------------------------
  // 1 - Переменные дизайнера
  //--------------------------------

  /// <summary>
  /// Префаб снаряда для стрельбы
  /// </summary>
  public Transform shotPrefab;

  /// <summary>
  /// Время перезарядки в секундах
  /// </summary>
  public float shootingRate = 0.25f;

  //--------------------------------
  // 2 - Перезарядка
  //--------------------------------

  private float shootCooldown;

  void Start()
  {
    shootCooldown = 0f;
  }

  void Update()
  {
    if (shootCooldown > 0)
    {
      shootCooldown -= Time.deltaTime;
    }
  }

  //--------------------------------
  // 3 - Стрельба из другого скрипта
  //--------------------------------

  /// <summary>
  /// Создайте новый снаряд, если это возможно
  /// </summary>
  public void Attack(bool isEnemy)
  {
    if (CanAttack)
    {
      shootCooldown = shootingRate;

      // Создайте новый выстрел
      var shotTransform = Instantiate(shotPrefab) as Transform;

      // Определите положение
      shotTransform.position = transform.position;

      // Свойство врага
      ShotScript shot = shotTransform.gameObject.GetComponent<ShotScript>();
      if (shot != null)
      {
        shot.isEnemyShot = isEnemy;
      }

     // Сделайте так, чтобы выстрел всегда был направлен на него
     MoveScript move = shotTransform.gameObject.GetComponent<MoveScript>();
     if (move != null)
     {
       move.direction = this.transform.right; // в двухмерном пространстве это
     }
    }
  }

  /// <summary>
  /// Готово ли оружие выпустить новый снаряд?
  /// </summary>
  public bool CanAttack
  {
    get
    {
      return shootCooldown <= 0f;
    }
  }
}
```

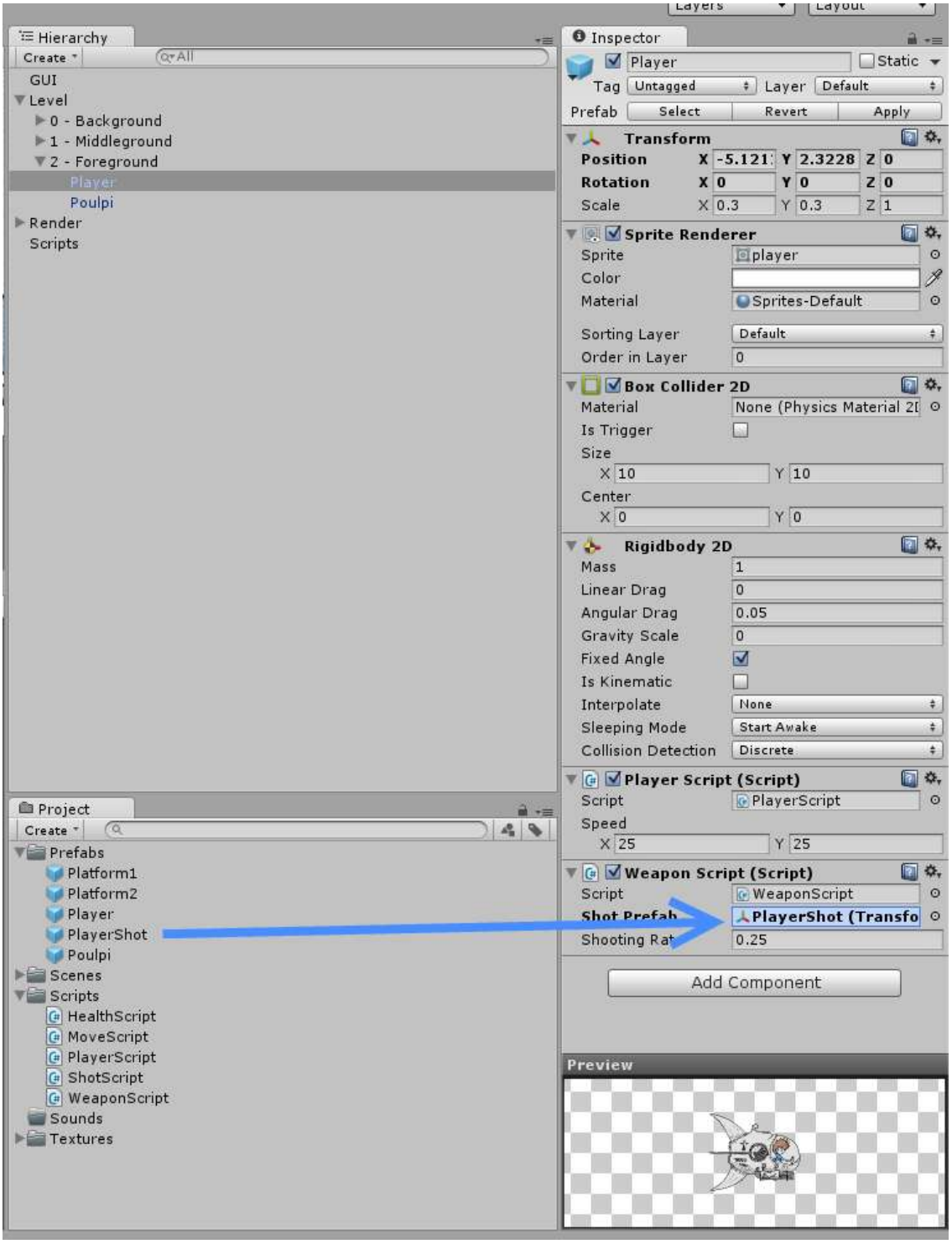
Прикрепите этот скрипт к игроку. Скрипт делится на три части:

Переменные во вкладке "Inspector"

Здесь у нас есть два члена: shotPrefab и shootingRate.

Первый необходим для установки выстрела, который будет использоваться с этим оружием.

Выберите игрока в сцене "Hierarchy". В компоненте "WeaponScript", вы можете увидеть свойство "Shot Prefab" со значением "None". Перетащите префаб "Shot" на это место:




Unity автоматически дополнит скрипт это информацией. Удобно, не так ли?

Переменная shootingRate имеет значение по умолчанию, установленное в коде. Мы не будем менять его на данный момент. Но вы можете начать игру и экспериментировать с ним, чтобы узнать на что она влияет.

Будьте осторожны: изменение значения переменной во вкладке "Инспектор" в Unity не приводит к сохранению этих значений в скрипте. Если добавите этот скрипт в другой объект, значение по умолчанию будет таким, которое написано в скрипте. Если же вы хотите сохранить отредактированные параметры, вы должны открыть свой редактор кода и записать эти значения там.

Перезарядка.

Оружие обладает определенной частотой выстрелов. Без этого параметра можно было бы выпускать неограниченное количество патронов в каждом кадре.

Поэтому нам нужен простой механизм охлаждения. Если его значение превышает 0, мы просто не можем стрелять. Мы вычитаем прошедшее время из каждого кадра.

**Публичный метод создания атаки**

Главная цель этого скрипта – активироваться через другой скрипт. Поэтому для создания снаряда мы используем публичный метод.

Создав екземпляр снаряда, мы извлекаем скрипты объекта выстрела и оверрайдим некоторые переменные.

*Внимание*: С помощью метода GetComponent<TypeOfComponent>() можно создать точный компонент (а значит, и скрипт, потому что скрипт – тоже компонент) объекта. Используйте generic (<TypeOfComponent>) для обозначения конкретного компонента, который вам нужен.
Кроме того, у нас есть GetComponents<TypeOfComponent>(), вызывающий список вместо первого и т.д.

**Использование оружия с классом игрока**

Если вы запустите сейчас игру, то увидите, что ничего не изменилось. Мы создали оружие, но оно совершенно бесполезно.

В самом деле, если "WeaponScript" был бы привязан к классу, мы никогда не смогли бы использовать метод Attack(bool).

Давайте вернемся к нашему "PlayerScript".

В функции Update() добавьте этот кусочек кода:

```
void Update()
{
  // ...

  // 5 - Стрельба
  bool shoot = Input.GetButtonDown("Fire1");
  shoot |= Input.GetButtonDown("Fire2");
  // Замечание: Для пользователей Mac, Ctrl + стрелка - это

  if (shoot)
  {
    WeaponScript weapon = GetComponent<WeaponScript>();
    if (weapon != null)
    {
      // ложь, так как игрок не враг
      weapon.Attack(false);
    }
  }

  // ...
}
```

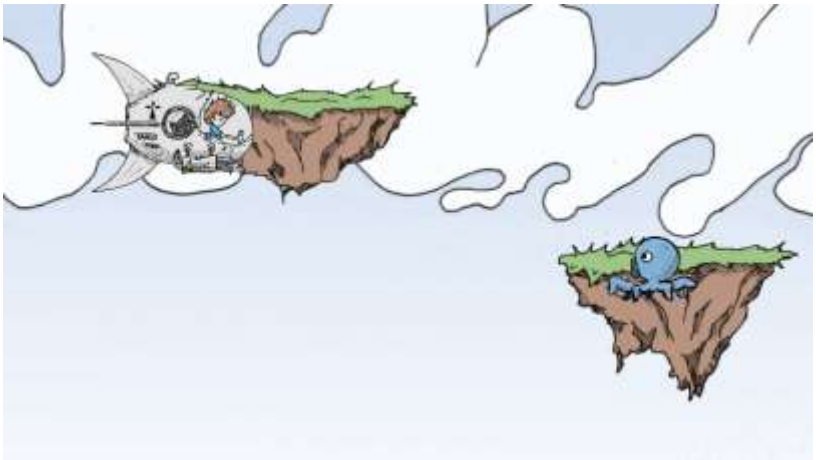
На данном этапе неважно, поставите вы его перед или после движения.

Что мы сделали?

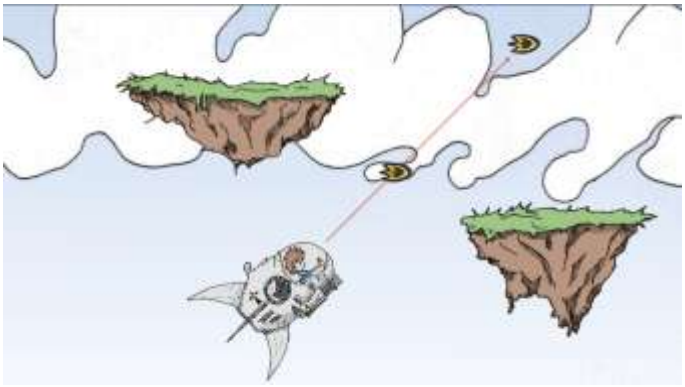
1. Мы определяем нажатие кнопки стрельбы (click или ctrl по умолчанию).
2. Извлекаем скрипт объекта.
3. Мы запускаем Attack(false).

*Button down*: Вы уже наверняка заметили, что мы используем метод GetButtonDown() для обеспечения ввода. "Down" в конце позволяет нам ввести данные при нажании кнопки и *только* один раз. GetButton() будет выводить true в каждом кадре, пока гирок не отпустит кнопку. В нашем случае нам явно необходимо поведение, обеспечиваемое методом GetButtonDown(). Попробуйте использовать GetButton() и почувствуйте разницу.

Запустите игру с помощью кнопки "Play". Вот что вы должны получить:



Пули летят слишком медленно? Поэкспериментируйте с префабом "Shot" чтобы выбрать ортимальное значение. Попробуйте также добавить вращение игроку: (0, 0, 45). Пули двигаются под углом 45 градусов, даже если вращение спрайта выстрела является некорректным – а ведь мы его не изменили.



Итак, у нас уже есть нечто похожее на шутер! Теперь вы умеете создавать оружие, которое может стрелять и уничтожить другие объекты. Давайте двигаться дальше. Мы хотим чтобы враги тоже могли стрелять.

**Вражеский снаряд**

Мы создадим новый снаряд с помощью этого спрайта:

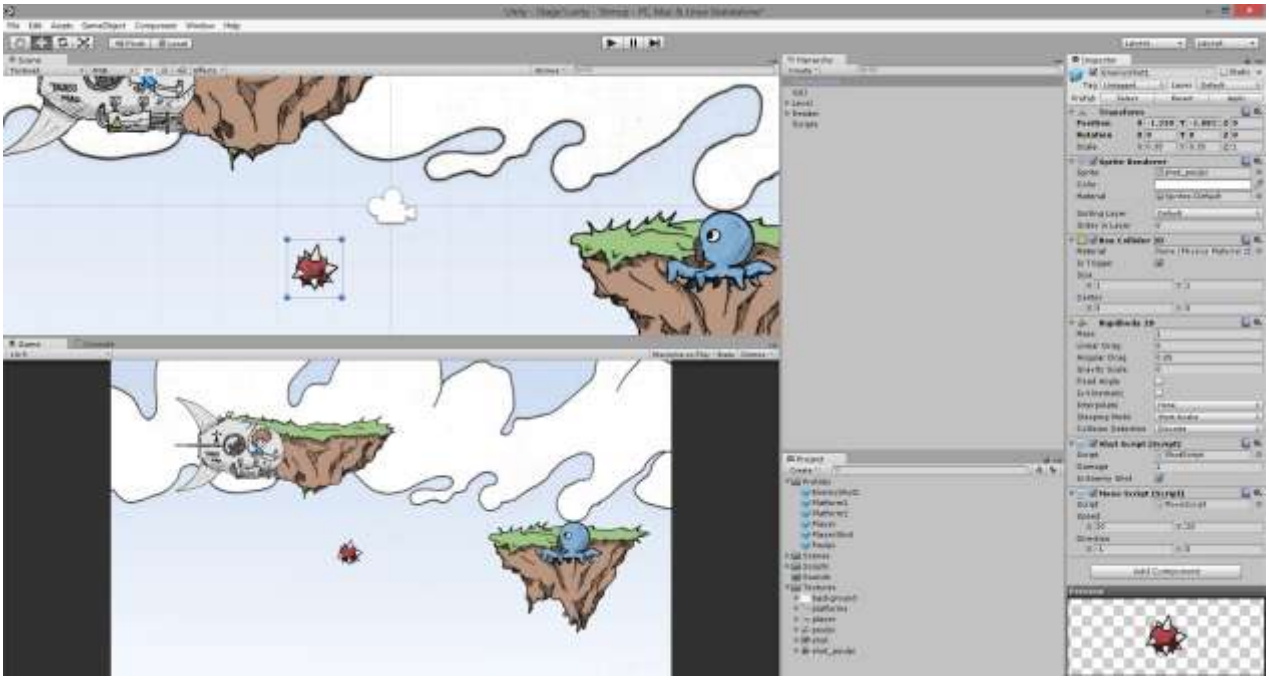
Если вы так же ленивы, как я, продублируйте префаб "PlayerShot", переименуйте его в "EnemyShot1" и измените спрайт, как описано выше.

Для дублирования создайте экземпляр, перетащив его на сцену, переименовав созданный игровой объект и, наконец, сохранив его как `Prefab'.

Или можно просто продублировать Prefab напрямую внутри папки с помощью ярлыков cmd+D (OS X) или ctrl+D (для Windows). Если вы не выбираете легких путей, вы можете создать новый спрайт с параметром rigibody, коллайдером с триггером и т.д.

Правильный масштаб - (0.35, 0.35, 1).

Вот, что у вас должно получиться.



При нажатии "Play" произойдет выстрел, который потенциально может уничтожить врага. Это из-за свойств "ShotScript" (которые по умолчанию плохо совместимы с Poulpi).

Не изменяйте ничего. Помните наш "WeaponScript"? Он то и установит правильные значения.

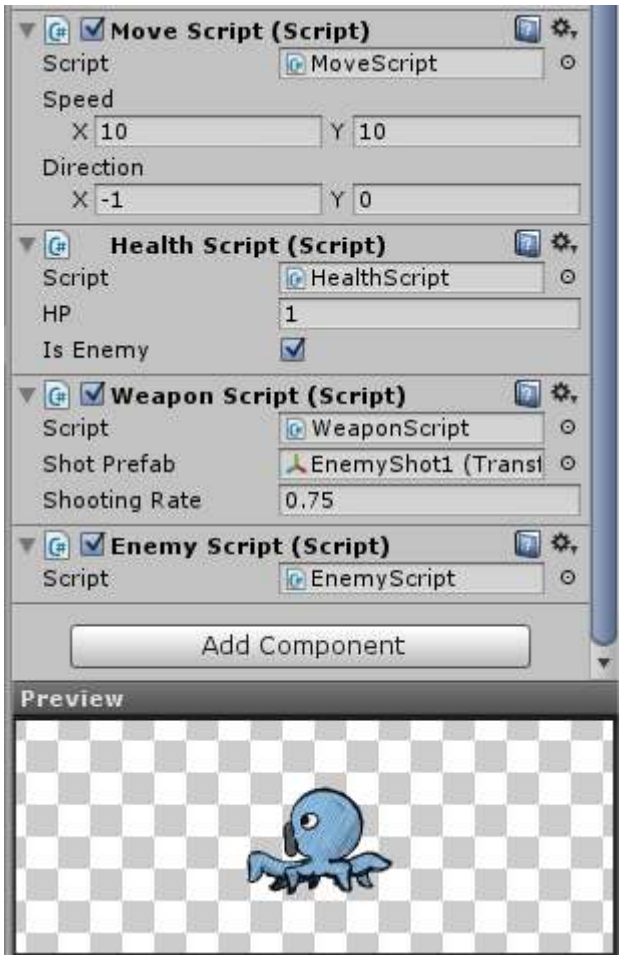
У нас есть префаб "EnemyShot1". Удалите экземпляры со сцены, если они есть.

Также, как мы делали для игрока, также нам нужно добавить оружие и врагу, а потом вызывать Attack() чтобы выстрелить. Вот, что нам надо сделать:

1. Добавьте "WeaponScript" врагу.
2. Перетащите префаб "EnemyShot1" в переменную "Shot Prefab" скрипта.
3. Создайте новый скрипт под названием "EnemyScript". Он просто будет запускать стрельбу в каждом кадре. Что-то вроде автострельбы.

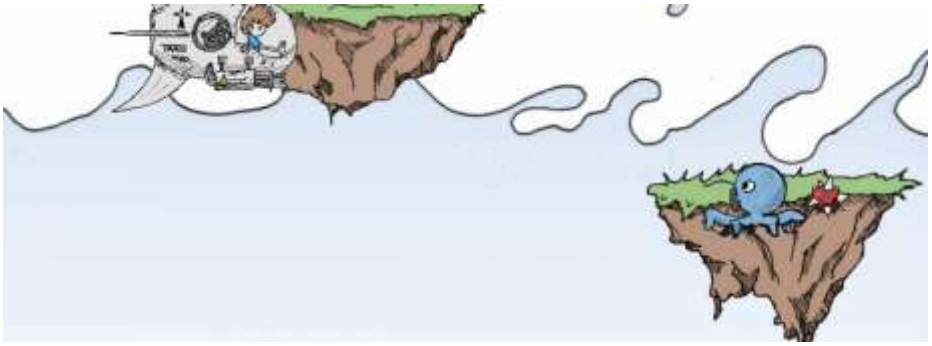
```
using UnityEngine;

/// <summary>
/// Enemy generic behavior
/// </summary>
public class EnemyScript : MonoBehaviour
{
  private WeaponScript weapon;

  void Awake()
  {
    // Получить оружие только один раз
    weapon = GetComponent<WeaponScript>();
  }

  void Update()
  {
    // автоматическая стрельба
    if (weapon != null && weapon.CanAttack)
    {
      weapon.Attack(true);
    }
  }
}
```

Прикрепите этот скрипт к осьминогу. У вас должно получиться следующее (заметьте, что частота стрельбы немного звеличилась до0.75):



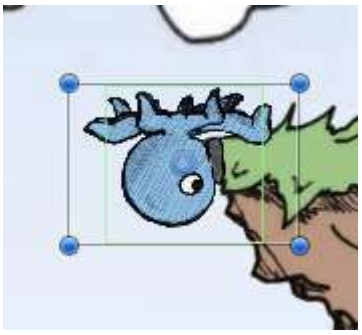
*Замечание*: Если вы модифицируете игровой объект в сцене, не забудьте сохранить все изменения в префабе, использовав кнопку "Применить" справа сверху от панели "Инспектор".

Попробуйте сыграть и посмотреть!



Итак, мы сделали то, что хотели и теперь и по нам тоже стреляют.

Если повернуть врага, вы можете сделать его стреляющим в его слева, но, хм... спрайт повернулся вверх ногами, а нам это не нужно.



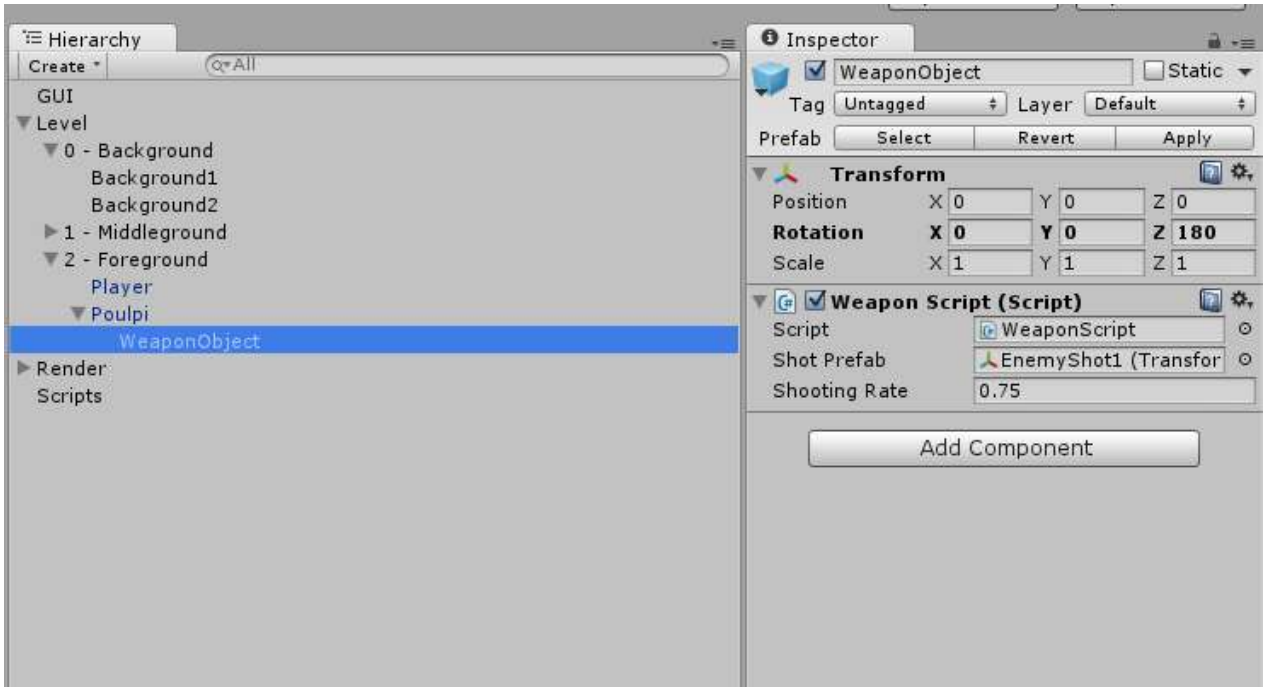
Давайте исправим это недоразумение.

**Стрельба в любом направлении.**

"WeaponScript" был написан особым образом: вы можете выбрать направление стрельбы, просто вращая прикрепленный игровой объект. Мы уже видели это раньше, когда вращали спрайт врага. Суть в том, чтобы создать пустой игровой объект как ребенка префаба врага. Итак, нам нужно:

1. Создать пустой игровой объект. Назовем его "WeaponObject".
2. Удалим "WeaponScript", прикрепленный к префабу врага.
3. Добавим "WeaponScript" к "WeaponObject" и установить свойства префба выстрела как мы это делали раньше.
4. Повернем "WeaponObject" вот так (0, 0, 180).

Если вы проделали это все на игровом объекте, а не на префабе, то не забудьте нажать на кнопку "Применить" для сохранения изменений. Вот, что у нас получилось:




However, we have a small change to make on the "EnemyScript" script.

В своем нынешнем состоянии вызов GetComponent<WeaponScript>() в "EnemyScript" возвращает null. В самом деле, "WeaponScript" больше не привязан к одному объекту игры.

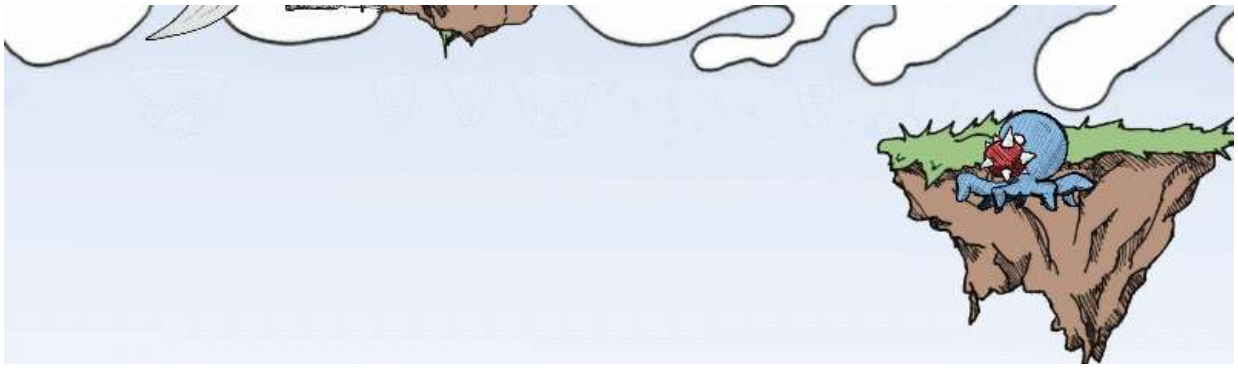
К счастью, в Unity также доступен метод, использующий детскую иерархию игрового объекта, который называется GetComponentInChildren<Type>().

Для GetComponent<>(), GetComponentInChildren<>() также существует в форме множественного числа: GetComponentsInChildren<Type>(). Обратите внимание на *s* после "Component". Этот метод возвращает список вместо первого соответствующего компонента.

На самом деле, просто для удовольствия, мы также добавили возможность управления несколькими видами оружия. Мы просто манипулируем списком вместо одного экземпляра компонента. Взгляните на весь "EnemyScript":

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

/// <summary>
/// Enemy generic behavior
/// </summary>
public class EnemyScript : MonoBehaviour
{
  private WeaponScript[] weapons;

  void Awake()
  {
    // Получить оружие только один раз
    weapons = GetComponentsInChildren<WeaponScript>();
  }

  void Update()
  {
    foreach (WeaponScript weapon in weapons)
    {
      // автоматическая стрельба
      if (weapon != null && weapon.CanAttack)
      {
        weapon.Attack(true);
      }
    }
  }
}
```

Наконец, нужно обновить скорость выстрела путем настройки публичной пременной "MoveScript" из префаба "EnemyShot1". Скорость выстрела должна быть больше скорости движения спрута:



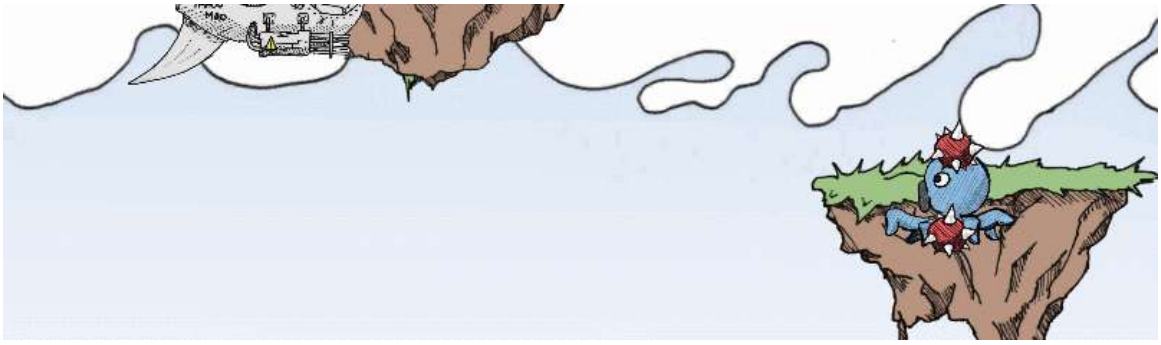
Мы сделали великого и ужасного осьминога. А давайте еще реализуем стрельбу в двух направлениях?

**Стрельба а двух направлениях**

Эта задача реализуеся всего в пару кликов. Для этого не нужны никакие скрипты:

1. Добавьте другое оружие врагу (дублируя первый "WeaponObject").
2. Измените угол поворота второго "WeaponObject".

Враг должен сейчас стрелять в двух направлениях. Возможный результат:



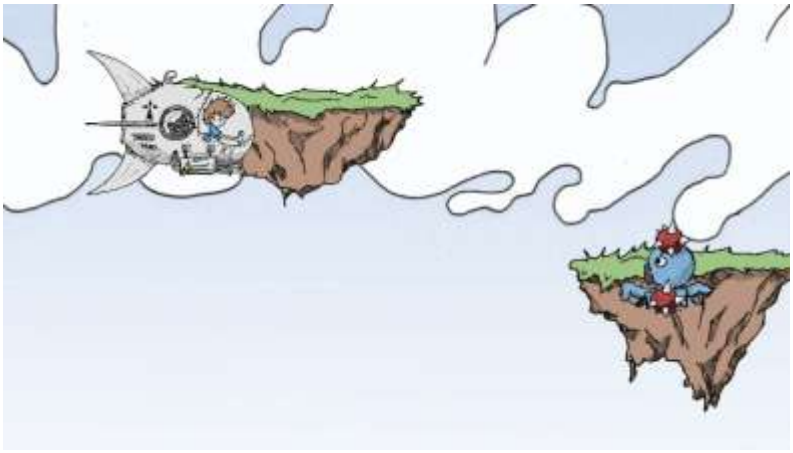
Это хороший пример правильной работы в Unity: создавая независимые скрипты вроде этого и делая публичными некоторые полезные переменные, можно значительно уменшить количество кода. Меньше кода - меньше ошибок.

**Нанесение урона игроку**

Наши осьминоги внушают ужас? Как бы не так! Да, они могут стрелять, но это не наносит повреждения игроку. Может, у них холостые патроны? Давайте разбираться.

Просто добавьте "HealthScript" на игрока. Убедитесь, что сняли галку с поля "IsEnemy".


Запустите игру и почувствуйте разницу:



**Бонус**

Вот вам некоторые советы для улучшения аспекта стрельбы в вашей будущей игре. Вы можете пропустить эту часть, если вас не интересуют подробности, относящиеся к жанру *шмапа*.

**Солкновения игрока с врагом**

Давайте посмотрим, как мы можем обработать столкновения между игроком и врагом, поскольку сейчас они сталкиваются друг с другом без последствий. Столкновение - это результат

пересечения двух не-триггерных 2D коллайдеров. Нам просто нужно обрабатывать событие OnCollisionEnter2D в PlayerScript:

```
//PlayerScript.cs
//....

void OnCollisionEnter2D(Collision2D collision)
  {
    bool damagePlayer = false;

    // Столкновение с врагом
    EnemyScript enemy = collision.gameObject.GetComponent<EnemyScript>();
    if (enemy != null)
    {
      // Смерть врага
      HealthScript enemyHealth = enemy.GetComponent<HealthScript>();
      if (enemyHealth != null) enemyHealth.Damage(enemyHealth.hp);

      damagePlayer = true;
    }

    // Повреждения у игрока
    if (damagePlayer)
    {
      HealthScript playerHealth = this.GetComponent<HealthScript>();
      if (playerHealth != null) playerHealth.Damage(1);
    }
  }
```

При столкновении мы наносим урон как врагу, так и игроку благодаря наличию компонента HealthScript. К нему привязано все, что относится к здоровью/урону.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Практическое занятие № 5. Создание сцены с эффектом параллакс-скроллинга**

**Выполнив работу, Вы будете:**
***уметь:***
- создавать сцены с эффектом параллакс – скроллинга

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Добавить на вашу сцену в игре эффект параллакс – скроллинга

**Порядок выполнения работ:**

Эффект параллакса

Эффект, который вы найдете в каждой 2D игре за последние 15 лет называется параллакс-эффектом. Короче говоря, идея заключается в том, чтобы перемещать фоновые слои с разной скоростью (т.е. чем дальше слой, тем медленнее он движется). Если все сделано правильно, то создается иллюзия глубины. Это здорово, красиво и просто в использовании.

Давайте реализуем все это в Unity. Добавление оси скроллинга – задача непростая, и нам придется поразмыслить, как написать остальной код игры с учетом этого аспекта. Над этим стоит задуматься, прежде чем начать кодировать :) Мы можем воспользоваться несколькими решениями:

Игрок и камера двигаются, все остальное неподвижно

Игрок и камера неподвижны. Все остальное движется

Первый вариант представляет собой достаточно сложную задачу, если у вас режим камеры Perspective (камера будет отображать объекты в режиме перспективного просмотра). Параллакс очевиден: фоновые элементы отличаются большей глубиной. Поэтому кажется, что они движутся медленнее.

Но в стандартной 2D игре в **Unity** мы используем ортографическую камеру (которая будет отображать объекты в ортогональном режиме, т.е. без эффекта глубины). У нас не будет такой величины, как глубина, в принципе.

*Немного о камере*: помните такое свойство нашей камеры, как "Projection"? В нашей игре оно установлено на Orthographic. Режим Perspective означает, что мы имеем дело с классической 3D камерой с управлением глубиной. Когда мы переключаетмся на ортографическую камеру, все объекты отображаются с одинаковой глубиной. Это особенно полезно для графического интерфейса или 2D игры.

Чтобы добавить в нашу игру эффект параллакс-скроллинга, нам придется использовать оба эти способа. Итого у нас будет два скроллинга:

Игрок движется вперед вместе с камерой.

Фоновые элементы движутся с разной скоростью (в дополнение к движению камеры).

Вы можете спросить: "Почему бы нам просто не установить камеру как дочерний объект объекта игрока?". Действительно, в Unity, если вы установите объект (напр. камера ) в качестве

дочернего по отношению к объекту игры, то этот объект будет сохранять свою относительную позицию по отношению к своему родителю. Так что, если камера дочерний объект по отношению к игроку и направлена на него, она будет следовать за ним. Да, можно сказать, что это решение задачи, но оно не будет соответствовать нашему геймплею.

В играх жанра *shmup*, камера *ограничивает* передвижения игрока. Если камера перемещается вслед за игроком и по горизонтальной, и по вертикальной оси, игрок свободен в своих передвижениях и может идти, куда захочет. Но в нашем случае игрок *ДОЛЖЕН* находиться внутри четко обозначенной области.

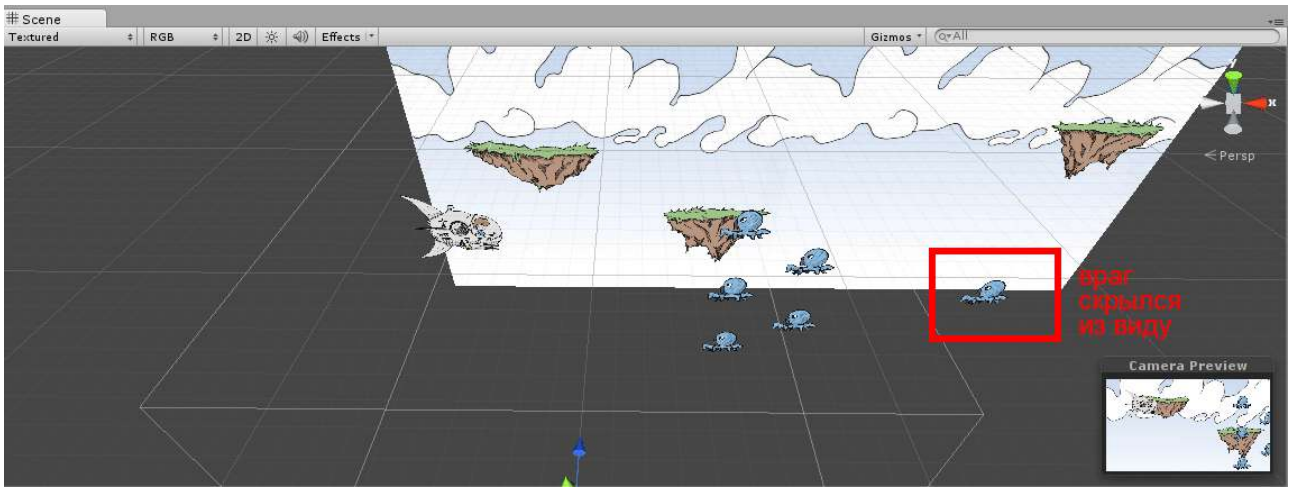
Мы бы также порекомендовали использовать камеру в качестве самостоятельного объекта в 2D играх. Даже в платформере камера не строго привязана к игроку: она следует за его передвижениями с некоторыми ограничениями. В качестве одного из лучших примеров реализации камеры в платформере можно привести всем известную игру Super Mario World.

Спавн (место появления) врагов

Однако, добавление в игру параллакс-скроллинга приводит к определенным последствиям, особенно это касается врагов. На данном этапе, они просто передвигаются по карте и стреляют с первой секунды игры. Но нам надо, чтобы они ждали и оставались неуязвимыми до начала спавна.

Как построить спавн врагов? Конечно, это в первую очередь зависит от игры. Вы можете создать события, которые запускают спавн врагов, точки спавна, заданные положения и т.д.

*Вот что мы сделаем* : Мы расположим осьминогов на карте просто перетянув префабы на нужное место. По умолчанию они статичны и неуязвимы пока не попадут в камеру, которая активирует их.



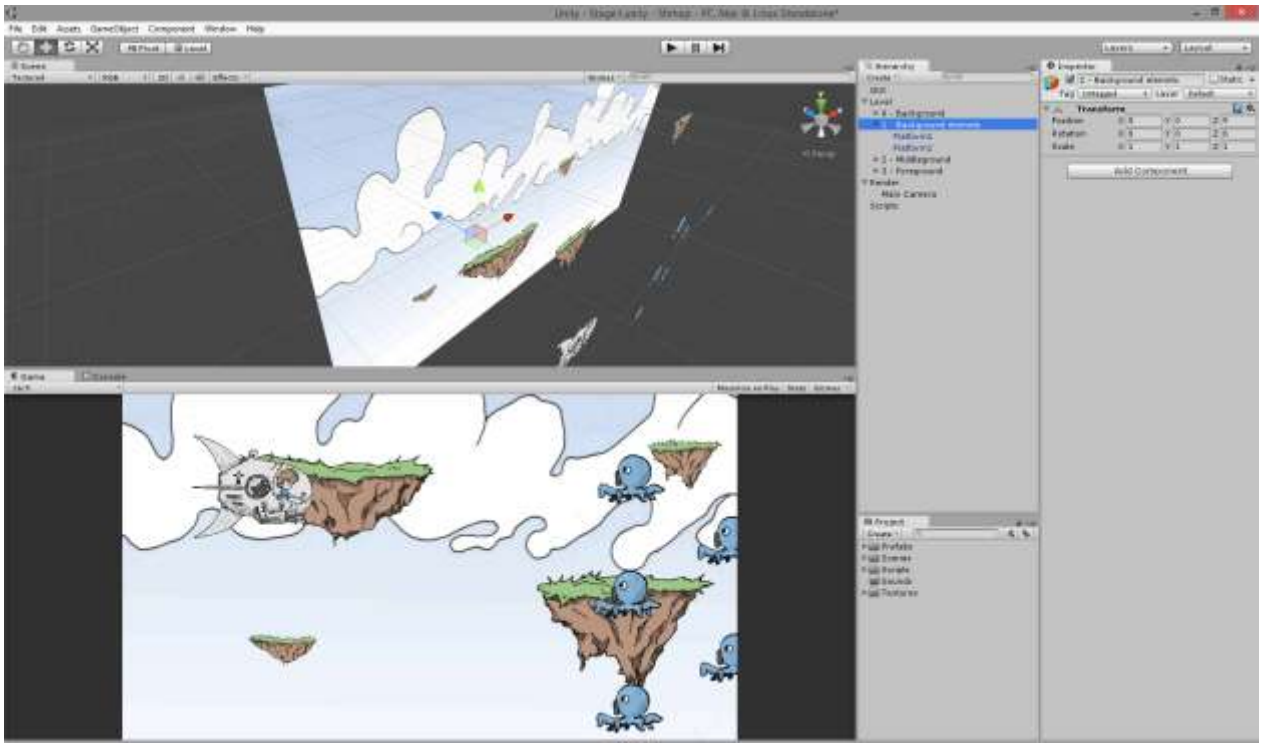

Что хорошо – так это то, что для настройки врагов можно использовать редактор Unity. Вы прочитали правильно: не прилагая абсолютно никаких усилий *вы получаете готовый редактор уровней* .

Мы действительно думаем, что вам стоит использовать редактор **Unity** в качестве редактора уровней, если, конечно, у вас нет недостатка во времени, средствах и собственных редакторах уровней, для которых нужны специальные инструменты.

Слои

Для начала нам нужно определиться с нашими слоями и указать, какие из них будут закольцованы. Закольцованный фон будет повторяться снова и снова на протяжении уровня. Это особенно полезно для таких вещей, как небо. Добавьте новый слой на сцену для фоновых элементов. Вот, что у нас должно получиться:

| Слой | Повторяющееся | Позиция |
|---|---|---|
| Задний фон с небом | Да | (0, 0, 10) |
| Задний фон (1-й ряд летающих платформ) | Нет | (0, 0, 9) |
| Средний фон (2-й ряд летающих платформ) | Нет | (0, 0, 5) |
| Передний план с игроками и врагами | Нет | (0, 0, 0) |



Мы также можем добавить слои впереди игрока. Просто следите за тем, чтобы координаты оси Z находились между[0, 10], иначе вам придется долго настраивать камеру.

Добавляя слои перед игроком, находящимся на переднем плане, следите за тем, чтобы все объекты были четко видны и отличимы на фоне друг друга. Во многих играх эта техника не используется, так как она влияет на четкость графики.

В стандартных пакетах Unity есть кое-какие скрипты для параллакс-скроллинга (взгляните на демо 2D платформера в Asset Store). Вы, конечно, можете воспользоваться ими, но я подумал, что было бы интересно написать его с нуля. Вообще, старайтесь не использовать стандартные пакеты, поскольку они не дают вашему разуму развиваться. Вы ведь не ходите создать бесполезный клон, который никто и не вспомнит?

Простой скроллинг

Мы начнем с легкого: прокрутка фона без цикла. Помните, "MoveScript", который мы использовали ранее? Принцип тот же: скорость и направление, применяемые на протяжении определенного промежутка времени.

Создайте новый скрипт и назовите его "ScrollingScript":

```
using UnityEngine;

/// Скрипт параллакс-скроллинга, который нужно прописать для слоя
public class ScrollingScript : MonoBehaviour
{
  // Скорость прокрутки
  public Vector2 speed = new Vector2(2, 2);

  // Направление движения
  public Vector2 direction = new Vector2(-1, 0);

  // Движения должны быть применены к камере
  public bool isLinkedToCamera = false;

  void Update()
  {
    // Перемещение
    Vector3 movement = new Vector3(
      speed.x * direction.x,
      speed.y * direction.y,
      0);

    movement *= Time.deltaTime;
    transform.Translate(movement);

    // Перемещение камеры
    if (isLinkedToCamera)
    {
      Camera.main.transform.Translate(movement);
    }
  }
}
```

Прикрепите скрипт к объектам игры со следующими значениями:

| **Слой** | **Скорость** | **Направление** | **Связан с камерой** |
|---|---|---|---|
| 0 - Задний фон | (1, 1) | (-1, 0, 0) | Нет |
| 1 - Фоновые элементы | (1.5, 1.5) | (-1, 0, 0) | Нет |
| 2 - Средний слой | (2.5, 2.5) | (-1, 0, 0) | Нет |
| 3 - Передний план | (1, 1) | (1, 0, 0) | Да |

А теперь, давайте добавим элементы на сцену:

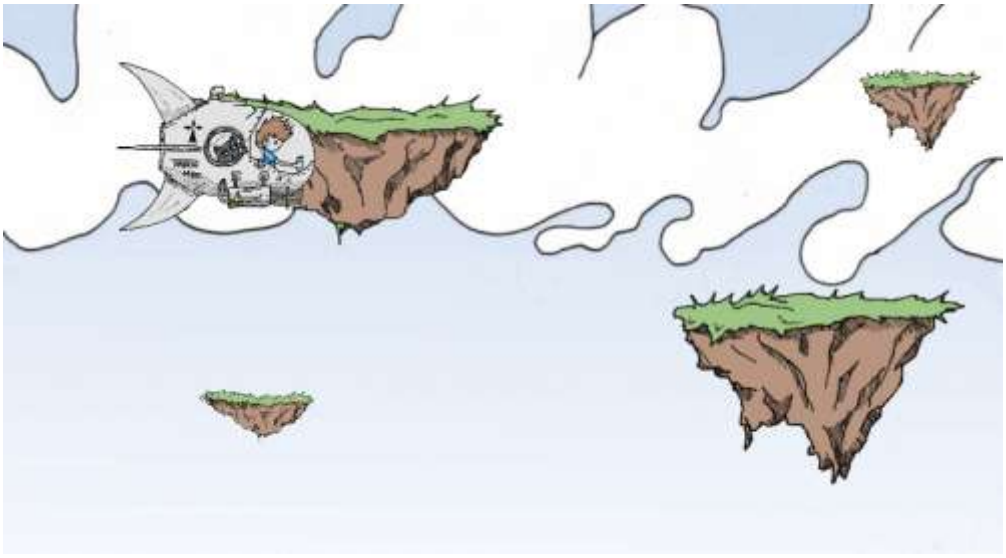
Добавьте третий слой после двух предыдущих.

Добавьте несколько небольших платформ в слое 1 - Фоновые элементы.

Добавьте платформы в слое 2 - Средний слой.

Добавьте врагов с правой стороны на слое 3 - Передний план, подальше от камеры.

Результат:



Неплохо! Но мы видим, что враги двигаются и стреляют когда они находятся вне камеры, даже до начала спавна!

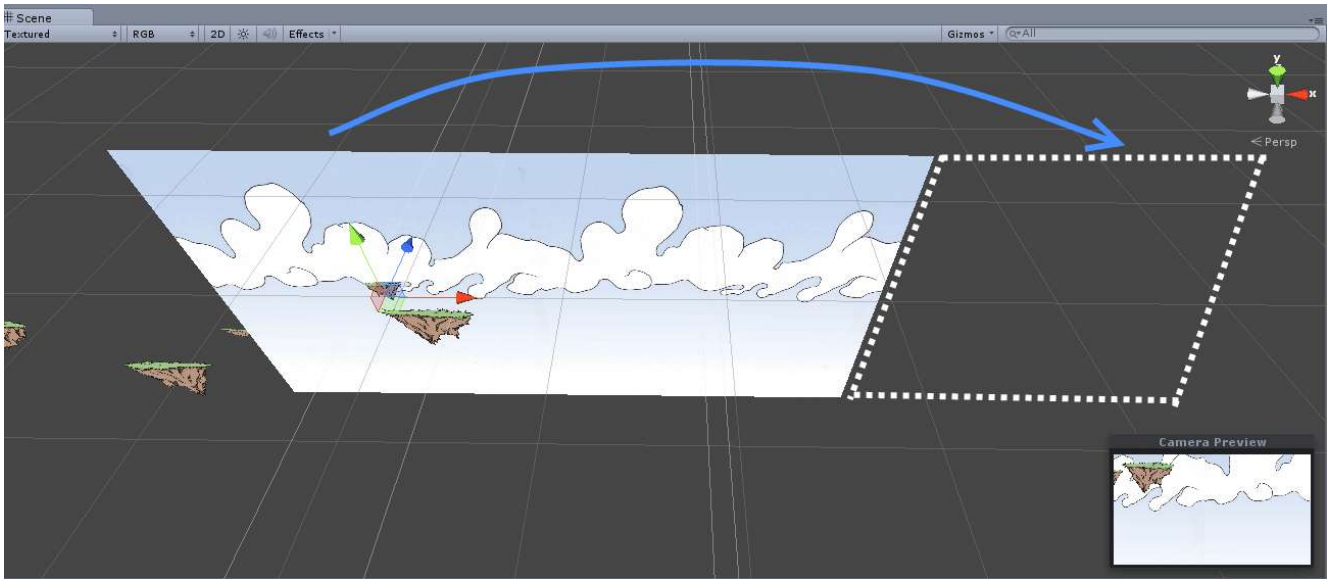
Более того, проходя мимо игрока, они не исчезают (уменьшите изображение в режиме "Scene" и посмотрите налево: Poulpies все еще двигаются). Поэкспериментируйте со значениями.

Мы исправим эти проблемы в дальнейшем. Во-первых, мы должны управлять бесконечным фоном (небо).

Бесконечный фон

Для того, чтобы получить бесконечный фон, нам нужно всего лишь проследить за детским объектом слева от бесконечного фона.

Когда этот объект выходить за левый край кадра, мы передвигаем его в правую часть слоя. *И так до бесконечности* .




Слой, заполненный изображениями, должен полностью покрывать все пространство кадра, чтобы мы не могли рассмотреть, что находится за ним. В данном случае слой неба состоит из трех частей, но это чисто индивидуальное решение.

Найдите правильный баланс между потреблением ресурсов и гибкостью для вашей игры.

В нашем случае смысл состоит в том, чтобы разместить все детские объекты в пределах слоя и установить для них рендерер.

*Небольшая заметка о том, как правильно использовать рендерер*: Этот метод не работает для видимых объектов (то есть, тех к которым привязаны скрипты). Но вряд ли вам когда-нибудь понадобиться применять его для невидимых объектов.

*Расширение*: Язык C# позволяет расширить класс без использования базового исходного кода класса. Создадим статичный метод, начав с первого параметра, который выглядит так: this Type currentInstance. В классе Type теперь доступен новый метод везде, где доступен ваш собственный класс. В рамках метода расширения можно использовать текущий экземпляр класса, применяя метод с помощью параметра currentInstance вместо этого.

Скрипт "RendererExtensions"

Создайте новый C# файл "RendererExtensions.cs" и напишите в нем:

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

/// Скрипт параллакс-скроллинга, который должен быть прописан для слоя
public class ScrollingScript : MonoBehaviour
{
  // Скорость прокрутки
  public Vector2 speed = new Vector2(10, 10);

  // Направление движения
  public Vector2 direction = new Vector2(-1, 0);

  // Движения должны быть применены к камере
  public bool isLinkedToCamera = false;

  // 1 - Бесконечный фон
  public bool isLooping = false;

  // 2 - Список детей с рендерером
  private List<Transform> backgroundPart;

  // 3 - Получаем всех детишек))
  void Start()
  {
    // Только для безконечного фона
    if (isLooping)
    {
      // Задействовать всех детей слоя с рендерером
      backgroundPart = new List<Transform>();

      for (int i = 0; i < transform.childCount; i++)
      {
        Transform child = transform.GetChild(i);

        // Добавить только видимых детей
        if (child.renderer != null)
        {
          backgroundPart.Add(child);
        }
      }

      // Сортировка по позиции.
      // Примечание: получаем детей слева направо.
      // Мы должны добавить несколько условий для обработки
      // разных направлений прокрутки.
      backgroundPart = backgroundPart.OrderBy(
        t => t.position.x
      ).ToList();
    }
  }
```

```
void Update()
{
  // Перемещение
  Vector3 movement = new Vector3(
    speed.x * direction.x,
    speed.y * direction.y,
    0);

  movement *= Time.deltaTime;
  transform.Translate(movement);

  // Перемещение камеры
  if (isLinkedToCamera)
  {
    Camera.main.transform.Translate(movement);
  }

  // 4 - Loop
  if (isLooping)
  {
    // Получение первого объекта.
     // Список упорядочен слева (позиция по оси X) направо.
    Transform firstChild = backgroundPart.FirstOrDefault();

    if (firstChild != null)
    {
      // Проверить, находится ли ребенок (частично) перед камерой.
      // Первым делом мы тестируем позицию, т.к. метод IsVisibleFrom
      // немного сложнее воплотить в жизнь
      if (firstChild.position.x < Camera.main.transform.position.x)
      {
        // Если ребенок уже слева от камеры,
        // мы проверяем, покинул ли он область кадра, чтобы использовать его
        // повторно.
        if (firstChild.renderer.IsVisibleFrom(Camera.main) == false)
        {
          // Получить последнюю позицию ребенка.
          Transform lastChild = backgroundPart.LastOrDefault();
          Vector3 lastPosition = lastChild.transform.position;
          Vector3 lastSize = (lastChild.renderer.bounds.max - lastChild.renderer.bounds.min

          // Переместить повторно используемый объект так, чтобы он располагался ПОСЛЕ
          // последнего ребенка
          // Примечание: Пока работает только для горизонтального скроллинга.
          firstChild.position = new Vector3(lastPosition.x + lastSize.x, firstChild.positio

          // Поставить повторно используемый объект
          // в конец списка backgroundPart.
          backgroundPart.Remove(firstChild);
          backgroundPart.Add(firstChild);
        }
      }
    }
  }
}
```

```
работает только для горизонтального скроллинга.
= new Vector3(lastPosition.x + lastSize.x, firstChild.position.y, firstChild.position.z);
```

Поясним комментарии с цифрами:

Нам нужна публичная переменная для включения режима «замыкание» в Инспекторе.

Там также нужна частная переменная для хранения детских объектов слоя.

Используя метод Start(), мы добавляем в список backgroundPart детей, у которых есть рендерер. Благодаря небольшому количеству, мы ранжируем их по положению на оси X и ставим самый левый объект на первое место в массиве.

При использовании метода Update(), если для свойства isLooping установлено значение true, мы извлекаем первый детский объект из списка backgroundPart. Затем проверяем, находится ли он полностью за пределами кадра. Если да – изменяем его положение, помещая его после последнего (самого правого) ребенка. Наконец, мы ставим его в конец списка backgroundPart.

Действительно, backgroundPart точно отражает то, что происходит в нашей сцене.

Не забудьте включить свойство "Is Looping" в "ScrollingScript" для 0 - Background на панели "Inspector". В противном случае (и это очевидно) мы ничего не добьемся.

Нажмите на картинку выше, чтобы увидеть анимацию.

Почему мы не используем методы OnBecameVisible() и OnBecameInvisible()? *Потому что здесь они не работают.* Смысл этих методов заключается в исполнении фрагмента кода при появлении объекта на экране (или наоборот). Они работают как методы Start() или Stop() (если вы решили использовать один из них, просто добавьте нужный метод в MonoBehaviour, Unity применит его). Проблема в том, что эти методы также применяются при использовании режима "Scene" в редакторе Unity. А значит, поведение объектов в редакторе Unity и самой игре (независимо от платформы) будет отличаться. Это опасно и абсурдно. *Мы всячески рекомендуем избегать этих методов.*

Бонус: Улучшение написанных скриптов

Давайте обновим наши предыдущие скрипты.

Враг, версия 2 со спавном

Мы ранее говорили, что враги должны быть отключены, пока они не видны камерой.

Они также должны использоваться повторно, полностью пропав из поля зрения.

Мы должны обновить "EnemyScript", сделав вот что:

Отключить движения, коллайдер и авто-огонь (при инициализации).

Проверить, попал ли рендерер в камеру.

Запустить самоактивацию.

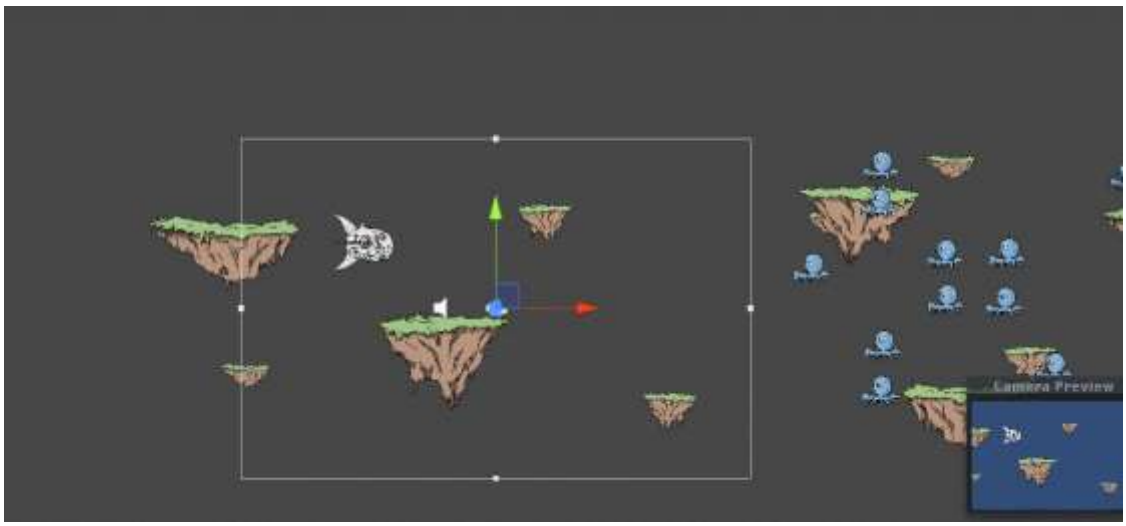
Уничтожить объект игры, когда он находится вне камеры.

*Цифры указывают на комментарии в коде*

```
using UnityEngine;

/// Общенное поведение врага
public class EnemyScript : MonoBehaviour
{
  private bool hasSpawn;
  private MoveScript moveScript;
  private WeaponScript[] weapons;

  void Awake()
  {
    // Получить оружие только один раз
    weapons = GetComponentsInChildren<WeaponScript>();

    // Отключить скрипты, чтобы деактивировать объекты при отсутста
    moveScript = GetComponent<MoveScript>();
  }

  // 1 - Отключить все
  void Start()
  {
    hasSpawn = false;

    // Отключить
    // -- коллайдеры
    collider2D.enabled = false;
    // -- Перемещение
    moveScript.enabled = false;
    // -- стрельбу
    foreach (WeaponScript weapon in weapons)
    {
      weapon.enabled = false;
    }
  }
```

```
  void Update()
  {
    // 2 - Проверить, начался ли спавн врагов.
    if (hasSpawn == false)
    {
      if (renderer.IsVisibleFrom(Camera.main))
      {
        Spawn();
      }
    }
    else
    {
      // автоматическая стрельба
      foreach (WeaponScript weapon in weapons)
      {
        if (weapon != null && weapon.enabled && weapon.CanAttack)
        {
          weapon.Attack(true);
        }
      }

      // 4 - Выход за рамки камеры? Уничтожить игровой объект.
      if (renderer.IsVisibleFrom(Camera.main) == false)
      {
        Destroy(gameObject);
      }
    }
  }

  // 3 - Самоактивация.
  private void Spawn()
  {
    hasSpawn = true;

    // Включить все
    // -- Коллайдеры
    collider2D.enabled = true;
    // -- Перемещение
    moveScript.enabled = true;
    // -- Стрельбу
    foreach (WeaponScript weapon in weapons)
    {
      weapon.enabled = true;
    }
  }
}
```

Отключение "MoveScript" привело к нежелательному результату: игрок никогда не дойдет до врагов, так как все они двигаются вместе со скроллингом слоя 3 - Foreground:




Помните: мы добавили "ScrollingScript" к этому слою, чтобы перемещатьть камеру вместе с игроком.

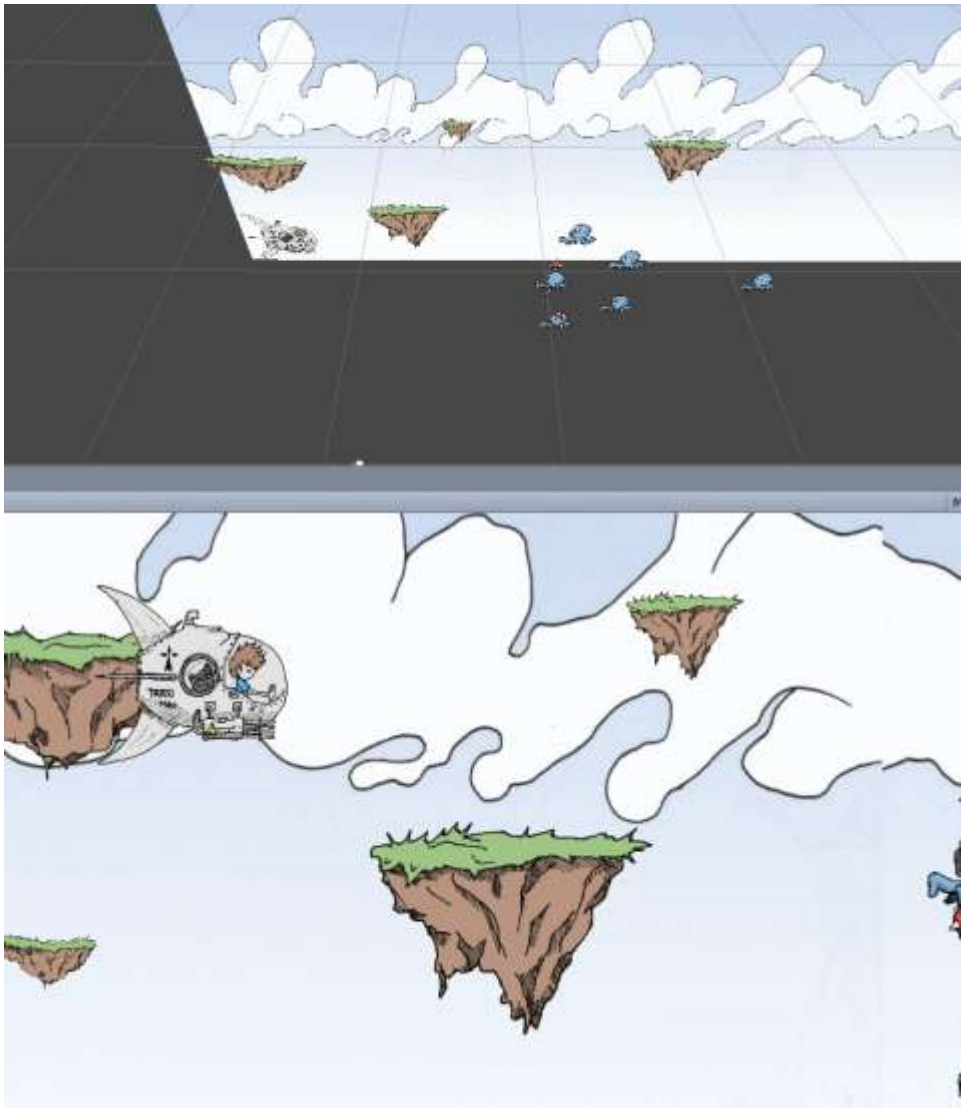
Но есть простое решение: переместить "ScrollingScript" со слоя 3 - Foreground на игрока!

В конце концов, почему бы и нет? Единственный объект, который двигается в этом слое – это сам игрок, и скрипт не прописан специально для определенного типа объекта.

Нажмите кнопку "Play" и убедитесь, что все работает.

Враги отключены до начала спавна (то есть, пока камера не достигнет точки, в которой они находятся).

Затем они исчезают, когда они находятся за пределами камеры.



Вы наверняка заметили, что игрок еще не ограничен рамками камеры. Нажмите "Play", затем кнопку со стрелочкой влево – и он покинет кадр.

Мы должны это исправить. Откройте "PlayerScript", и добавить в конце метод "Update()":

```
void Update()
  {
    // ...

    // 6 – Убедиться, что игрок не выходит за рамки кадра
    var dist = (transform.position - Camera.main.transform.positi

    var leftBorder = Camera.main.ViewportToWorldPoint(
      new Vector3(0, 0, dist)
    ).x;

    var rightBorder = Camera.main.ViewportToWorldPoint(
      new Vector3(1, 0, dist)
    ).x;

    var topBorder = Camera.main.ViewportToWorldPoint(
      new Vector3(0, 0, dist)
    ).y;

    var bottomBorder = Camera.main.ViewportToWorldPoint(
      new Vector3(0, 1, dist)
    ).y;

    transform.position = new Vector3(
      Mathf.Clamp(transform.position.x, leftBorder, rightBorder),
      Mathf.Clamp(transform.position.y, topBorder, bottomBorder),
      transform.position.z
    );

    // Вот и весь метод Update
  }
```

```
var dist = (transform.position - Camera.main.transform.position).z;
```

Мы определяем границы камеры и делаем так, чтобы положение игрока (*центр спрайта* ) не выходил за границы кадра.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

## Практическое занятие № 6. Совершенствование визуальной составляющей игры

**Выполнив работу, Вы будете:**
***уметь:***
- создавать анимацию персонажа
- создавать анимацию взрыва при столкновении

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Добавить в игру анимацию столкновения пули и противника

**Порядок выполнения работ:**

Игры с частицами в Unity

Частицы (в основном) - это простые спрайты, которые будут повторяться и отображаются в течение очень короткого промежутка времени. Подумайте о взрывах, лазерах, дыме и т.д. Эти эффекты достигаются через использование частиц – в большинстве случаев (взрыв может быть просто анимированным спрайтом). В Unity встроен мощный редактор частиц *Shuriken Engine*. Давайте с ним поработаем.

Префаб взрыва

Мы сделаем взрыв, который будет возникать когда враг или игрок погибают. Это включает в себя следующие действия:

Создать систему частиц нашего взрыва (как префаб).

Создать экземпляр и воспроизвести его при необходимости.

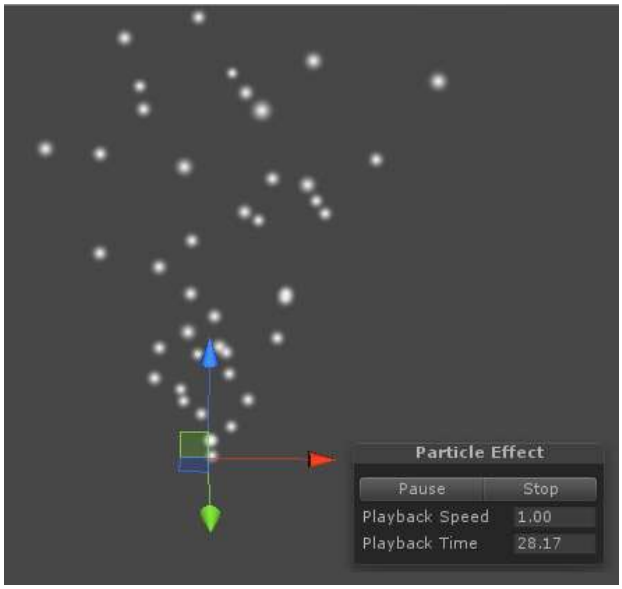
Взрыв, как правило, состоит из двух составляющих: огонь и дым.

Создаем частицы дыма в Unity

Создайте новую систему частиц ("Particle System") ("Game Object" -> "Create Other" -> "Particle System").

Советую вам работать на пустой части сцены (или в пустой сцене), чтобы вы четко могли видеть, что происходит. Если вы хотите сфокусироваться на объекте в виде "Сцена" ("Scene"), - дважды щелкните на объекте в «Иерархии», или нажмите клавишу F внутри окна "Сцена".

Взглянув на свою систему частиц в увеличенном масштабе, вы увидите сплошной поток искр, испускаемых объектом частиц:



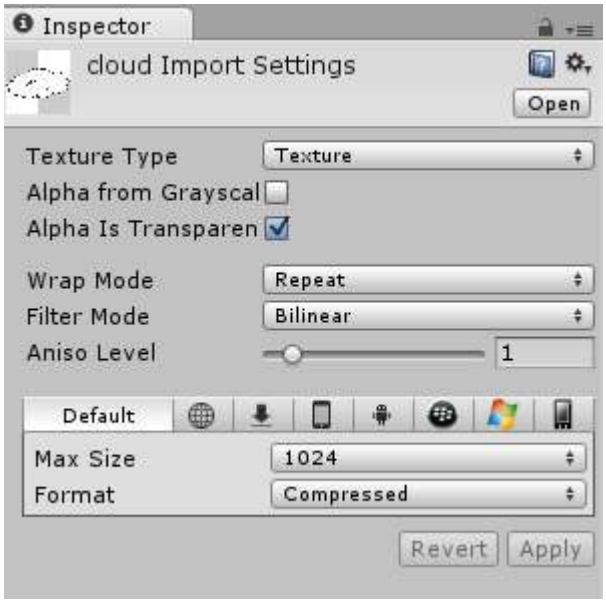

Обратите внимание на новое окно (с кнопками "Пауза" и "Стоп") в виде "Сцена" зрения, или в панели Инспектор (Inspector). Да, теперь она заполнена десятками полей, относящихся к системе частиц.

Когда вы выбираете систему частиц в "Hierarchy",она начинает симулировать систему. Если вы снимите галочку, симуляция остановится. Очень полезно понаблюдать, как работает созданная вами система (а она запускается мгновенно).

Мы будем использовать этот спрайт для частиц дыма (сохраните его на жестком диске):

Если у вас есть проблемы с прозрачностью при использовании вашего ассета, убедитесь, что прозрачность установлена для черных пикселей с альфой, равной 0. Действительно, даже если пиксель не виден, он по-прежнему имеет значение. Которое используется компьютером.

Скопируйте изображение в папке "Текстуры" (Textures). Измените "Тип текстуры" (Texture Type) на "Texture" и установите галочку рядом с "Альфа-прозрачность" (Alpha Is Transparent). У вас должно быть:

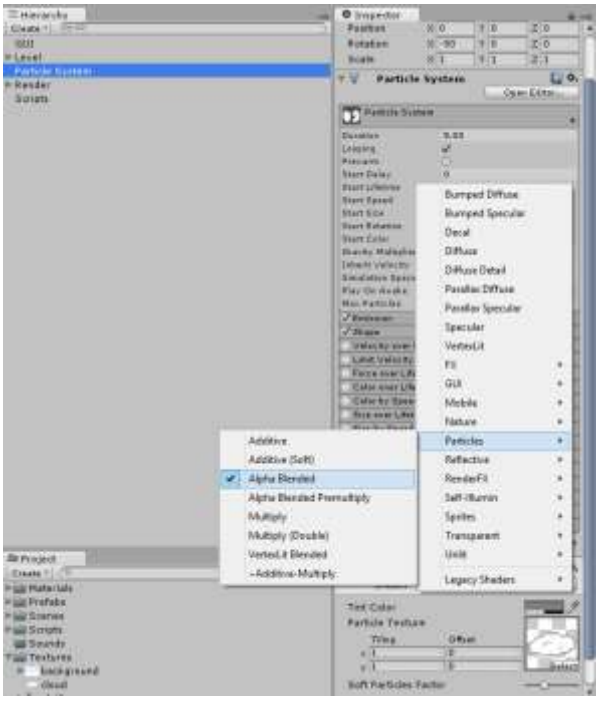



Мы используем свойство Unity 3D, а не 2D. Вообще-то это не имеет значения. Используя 2D инструменты, вы используете только подклассы Unity. Но полный потенциал Unity все еще остается незатронутым.

Назначьте текстуру частицы:

Перетащите текстуру на систему частиц в режиме "Inspector" (или на объект частиц в "Hierarchy", который привяжет текстуру к соответствующему свойству в "Inspector").

Измените шейдер "Particles" на "Alpha Blended":



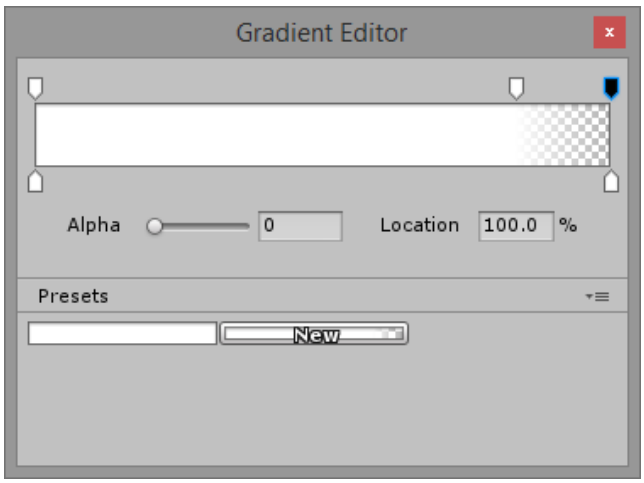
Чтобы создать идеальные частицы дыма, мы должны изменить многие параметры в системе частиц на вкладке "Инспектор". Рекомендую попробовать:

| **Категория** | **Параметр** | **Значение** |
|---|---|---|
| General | Duration (продолжительность) | 1 |

| | | |
|---|---|---|
| General | Max Particles (макс. число частиц) | 15 |
| General | Start Lifetime (время рождения) | 1 |
| General | Start Color (начальный цвет) | Gray (серый) |
| General | Start Speed (начальная скорость) | 3 |
| General | Start Size (начальный размер) | 2 |
| Emission | Bursts (всплески) | 0 : 15 |
| Shape (форма) | Shape (форма) | Sphere (сфера) |
| Color Over Lifetime (цвет в течении всей жизни) | Color (цвет) | See below (N°1) |
| Size Over Lifetime (размер в течении всей жизни) | Size (размер) | See below (N°2) |

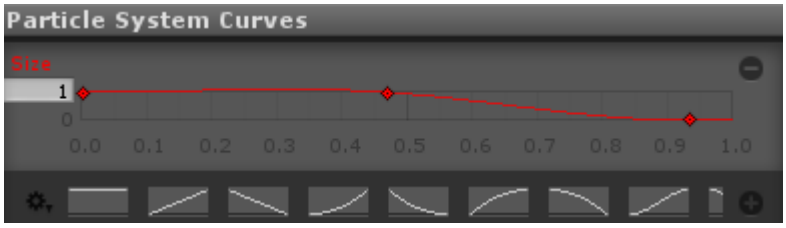
N°1 — цвет в течении всей жизни

Настройте эффект выцветания для свойства «Альфа-прозрачность»:

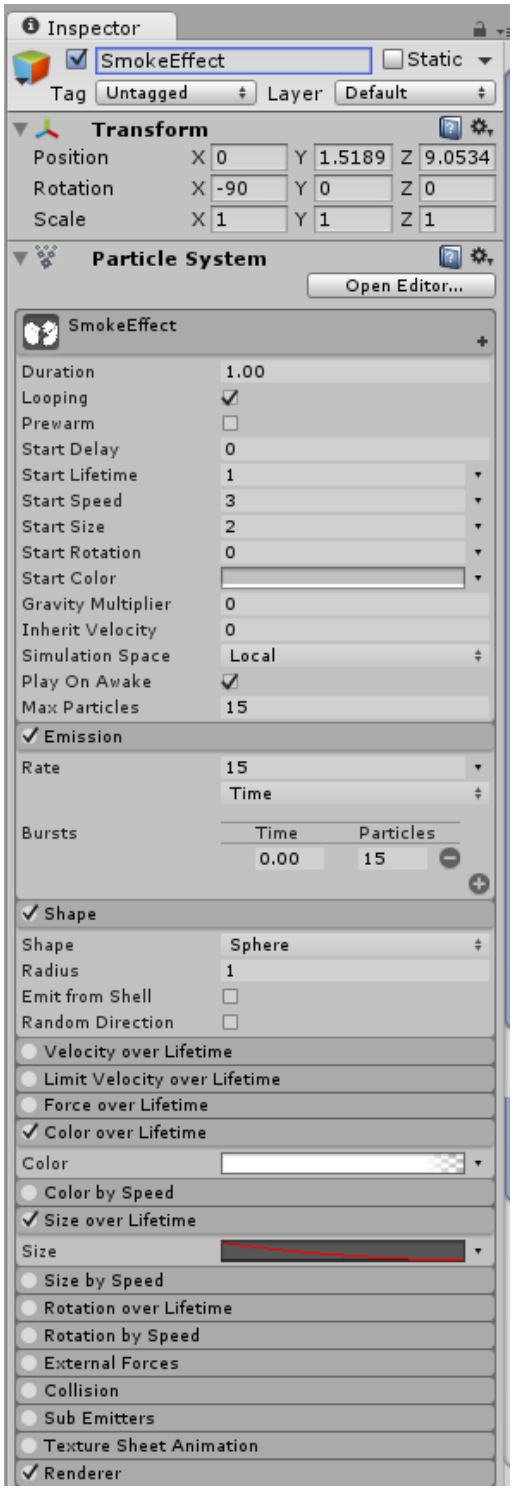



N°2 — размер в течении всей жизни

Выберите убывающую кривую:

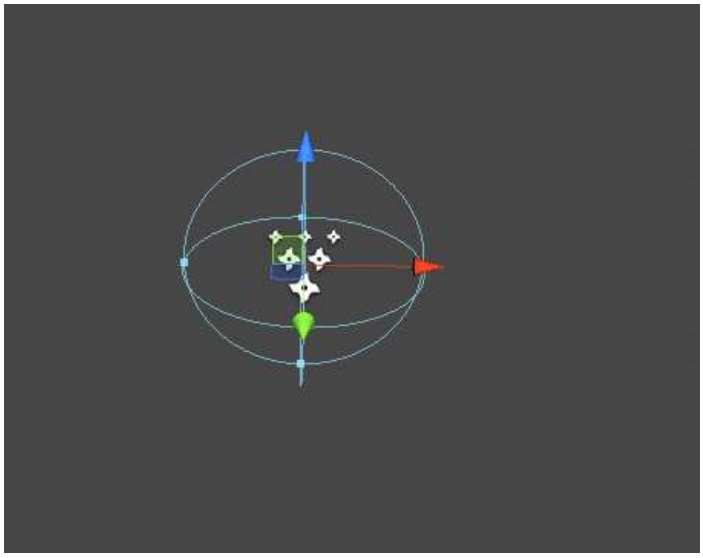



Вот, что у вас должно быть:




Не стесняйтесь настраивать систему. Поиграйте с редактором, чтобы увидеть, как это работает. Это ваша игра в конце концов. :) Когда вы будете удовлетворены, снимите флажок "Цикл" (Looping).

Обратите внимание на результат:



Это явно не идеально, но обратите внимание, как просто было создать этот эффект. Добавление частиц может превратить скучную игру в ту, на которую приятно смотреть.

Сохраните облако как префаб: создайте папку "Prefabs/Particles" и назовите его "SmokeEffect".

Частицы огня

Здесь все тоже самое:

Создайте новую систему частиц, так же, как вы делали выше.

Используйте материалы по умолчанию для огня ("Renderer/Material" - "Default-Particle"). Этого достаточно для наших нужд.

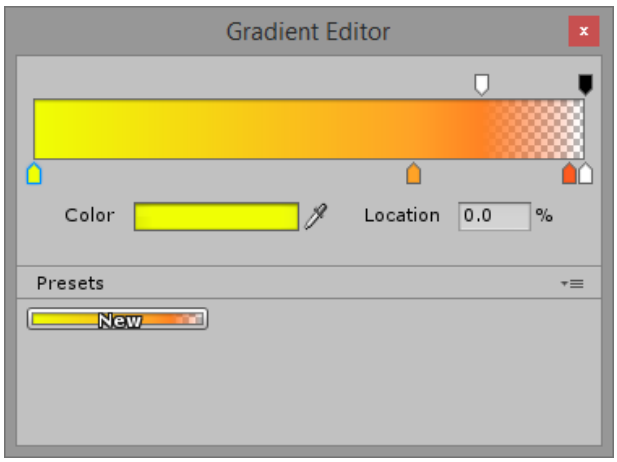
Мы рекомендуем использовать:

| **Категория** | **Параметр** | **Значение** |
|---|---|---|
| General | Looping | false |
| General | Duration (Продолжительность) | 1 |
| General | Max Particles (Макс. число частиц) | 10 |
| General | Start Lifetime (Время появления) | 1 |
| General | Start Speed (Начальная скорость) | 0.5 |
| General | Start Size (начальный размер) | 2 |
| Emission | Bursts (всплески) | 0 : 10 |
| Shape | Shape | Box |

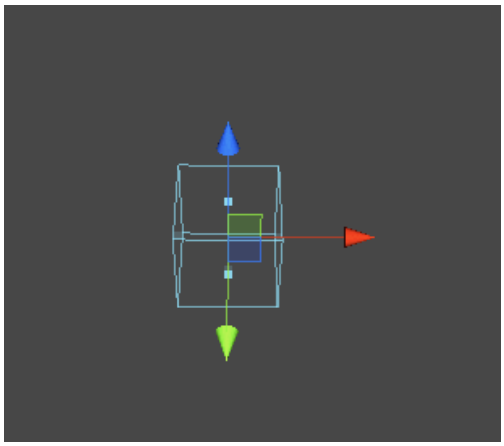
Color Over Lifetime Color See below
(цвет в течение всей жизни) (цвет) (N°1)

N°1 — цвет в течение всей жизни

Создаем красивый градиент от желтого до оранжевого, с затуханием в конце:




Вы должны получить:



Сохраните как префаб, дав ему имя "FireEffect". Теперь, мы будем использовать эти префабы в скрипте. Инстанцирование префабов частиц идентично инстанцированию игрока или выстрела. Тем не менее, вы должны помнить, что они должны удаляться, когда больше не нужны. Мы также скомбинируем системы огненных и дымовых частиц в скрипте, чтобы создать взрыв. Создайте скрипт "SpecialEffectsHelper":

```
using UnityEngine;

// Создание экземпляра частиц
public class SpecialEffectsHelper : MonoBehaviour
{
  // Синглтон
  public static SpecialEffectsHelper Instance;

  public ParticleSystem smokeEffect;
  public ParticleSystem fireEffect;

  void Awake()
  {
    // регистрация синглтона
    if (Instance != null)
    {
      Debug.LogError("Несколько экземпляров SpecialEffectsHelper!")
    }

    Instance = this;
  }

  // Создать взрыв в данной точке
  public void Explosion(Vector3 position)
  {
    // Дым над водой
    instantiate(smokeEffect, position);

    // да-даам

    // Огонь в небе
    instantiate(fireEffect, position);
  }

  // Создание экземпляра системы частиц из префаба
  private ParticleSystem instantiate(ParticleSystem prefab, Vector3
  {
    ParticleSystem newParticleSystem = Instantiate(
      prefab,
      position,
      Quaternion.identity
    ) as ParticleSystem;

    // Убедитесь, что это будет уничтожено
    Destroy(
      newParticleSystem.gameObject,
      newParticleSystem.startLifetime
    );

    return newParticleSystem;
  }
}
```

```
ние экземпляра системы частиц из префаба
ParticleSystem instantiate(ParticleSystem prefab, Vector3 position)
```

Поскольку у нас может быть несколько частиц в сцене в одно и то же время, мы вынуждены каждый раз создавать новый префаб. Если бы мы были уверены, что в сцене не может

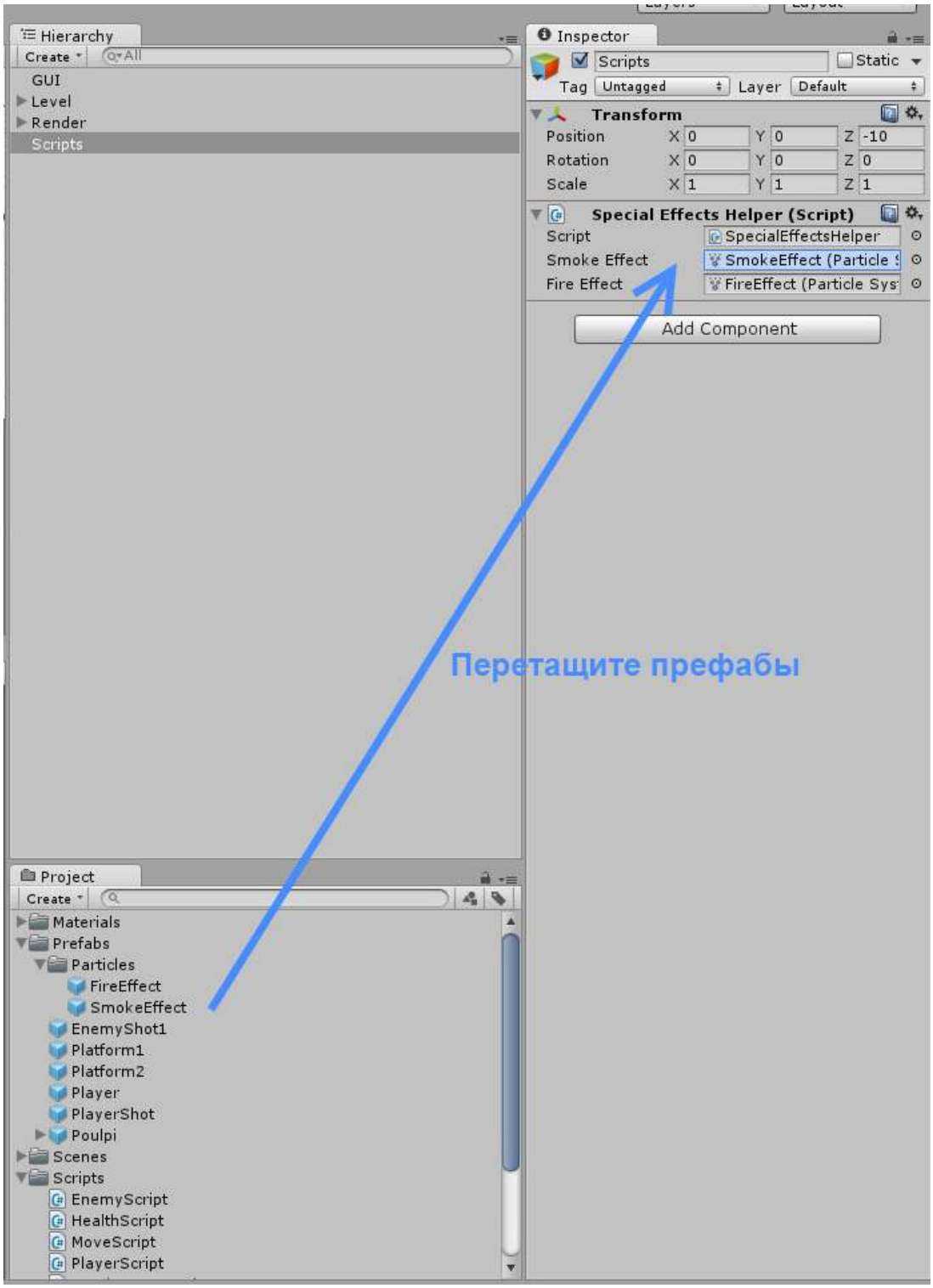
использоваться более одной системы одновременно, мы бы создали ссылку и использовали бы ее каждый раз.

Мы создали синглтон, доступ к которому можно получить в любой момент с помощью SpecialEffectsHelper.Instance.

Синглтон – это паттерн, благодаря которому объект не инстанцируется более одного раза. Мы немного отклонились от классического варианта кода, но суть осталась неизменной.

Привяжите скрипт к объекту "Scripts" в "Hierarchy".

Проинспектируйте его и заполните поля соответствующими префабами.




Настало время вызвать наш скрипт.

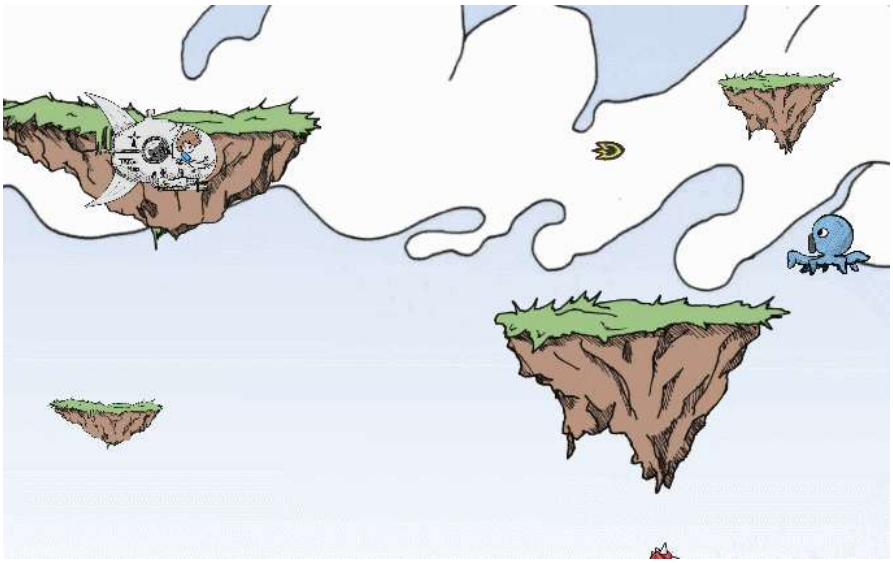
Откройте "HealthScript". Мы будем отслеживать разрушение игрового объекта и в нужный момент покажем наш взрыв. Нам нужно добавить одну строку:

```
SpecialEffectsHelper.Instance.Explosion(transform.position);
```

Into the `OnTriggerEnter()` method of the "HealthScript":

```
public Damage(int damageCount)
  {
    // ...

    if (hp <= 0)
    {
      // 'Splosion!
      SpecialEffectsHelper.Instance.Explosion(transform.position);

      // Dead!
      Destroy(gameObject);
    }

    // ...
  }
```

Запустите игру. Попробуйте стрелять во врагов.



Не плохо, не так ли? Хотя, не исключено, что есть способ и получше. Теперь, когда вы знаете, как работают частицы, вы сможете создавать действительно зрелищные взрывы.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

## Практическое занятие № 7. Работа со звуком

**Выполнив работу, Вы будете:**
***уметь:***
- добавлять звуковое сопровождение
- воспроизводить звук в игре

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Добавить в игру звуковое сопровождение
2. Добавить звуки при выстреле и столкновении

**Порядок выполнения работ:**

Начнем подыскивать звуковый файлы для нашей игры. Самое главное — сохраняйте все свои звуки в несжатом 8bit .wav формате. В дальнейшем, в самом **Unity** можно будет сжать звук, не изменяя исходного .wav файла. Вот, что мы можем сделать:

Купить мелодию.

Попросить знакомого музыканта.

Использовать бесплатные звуки из звуковых банков (например, FindSounds or Freesound).

Записать звук самому.

С музыкой немного сложнее, но тоже все решаемо:

На сайте Jamendo много музыки разных музыкальных направлений. Будьте осторожны с лицензиями для коммерческих целей.

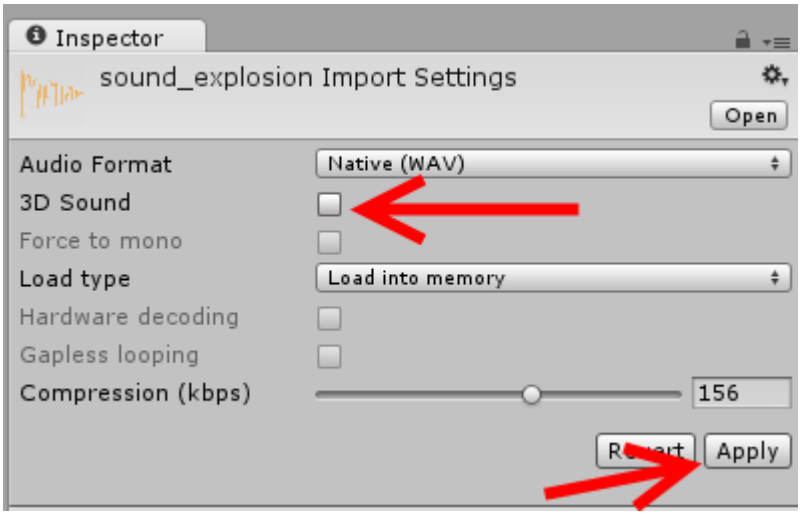
Иногда любители выкладывают свою музыку на рутрекере.

Воспользуйтесь продуктом Bosca Ceoil от Терри Кавана для создания своей, неповторимой мелодии.

Мы будем использовать один из треков игры "Больное бумажное приключение", Спинтроник для этого урока:

Импорт звука в Unity

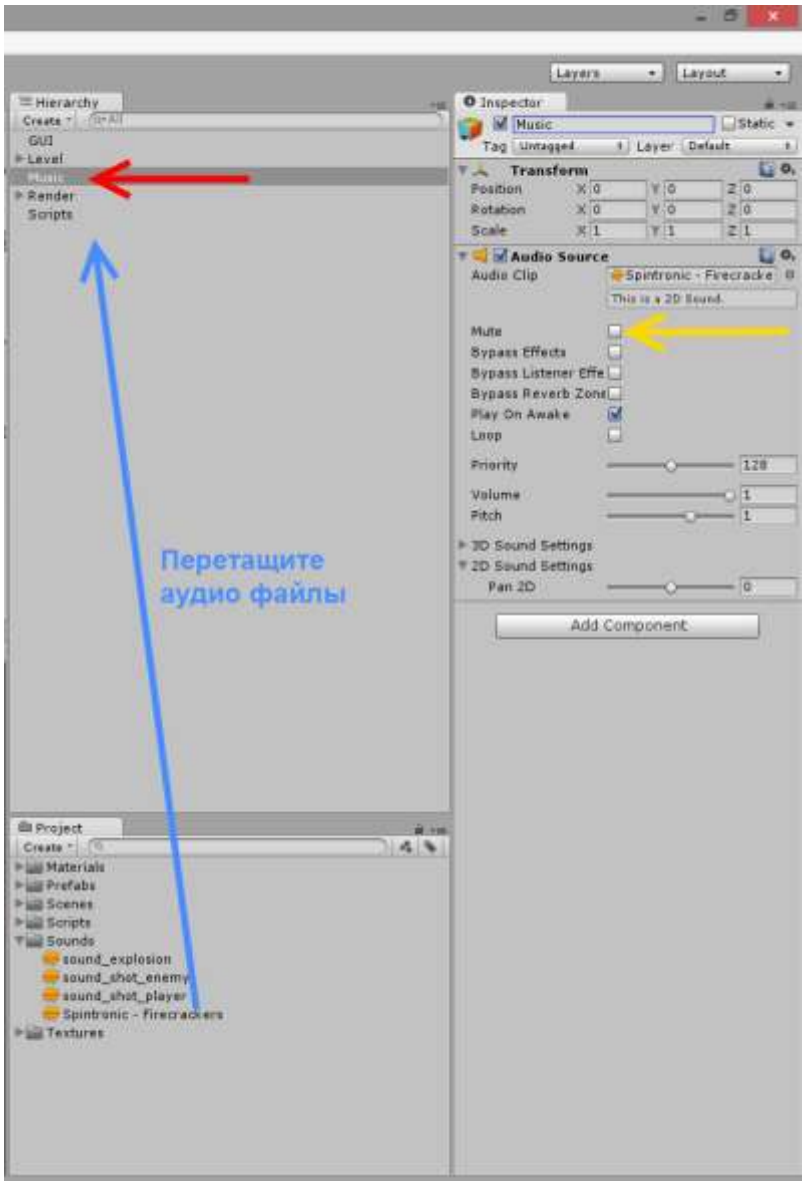
Переместите 4 звуковые дорожки в папку "Sounds". Убедитесь, что для каждой из во вкладке "Инспектор" отключен "3D звук" (так как мы делаем 2D игру) и нажата кнопка "Применить". Вот и все.




Для воспроизведения музыки, просто перетащите песню во вкладку «Иерархия» (Hierarchy). Мы приглашаем Вас:

Переименуйте новый игровой объект в «Музыка».

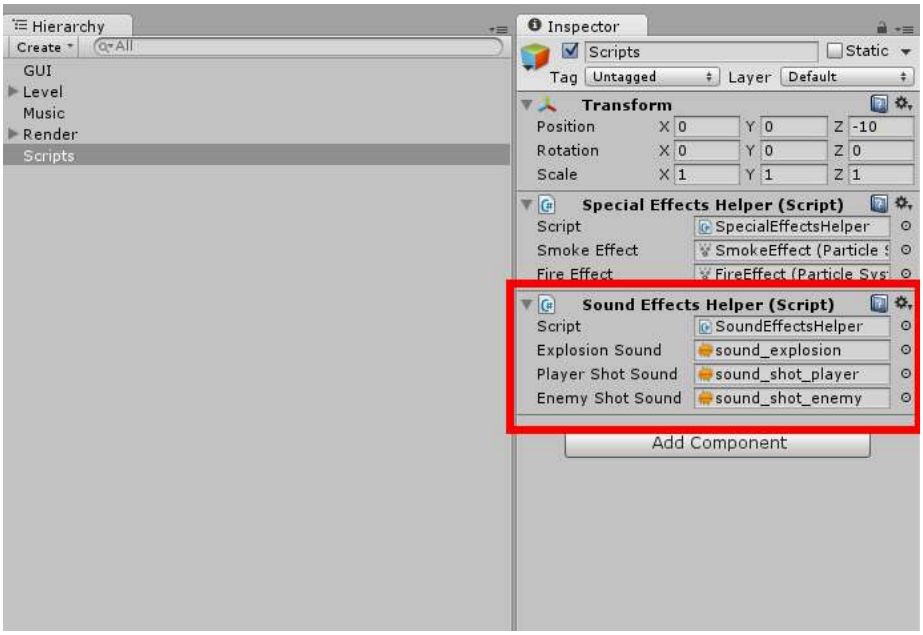
Поместите его на (0, 0, 0).



Обратите внимание на флажок "Mute". Это может вам пригодиться в ваших тестах. Со звуками придется повозиться, поскольку они должны проигрываться в определенное время. Для решения этой задачи напишем новый скрипт под названием "SoundEffectsHelper":

```
using UnityEngine;
using System.Collections;

// Создаем экземпляр класса для звука на основе кода без усилий
public class SoundEffectsHelper : MonoBehaviour
{

  // Синглтон
  public static SoundEffectsHelper Instance;

  public AudioClip explosionSound;
  public AudioClip playerShotSound;
  public AudioClip enemyShotSound;

  void Awake()
  {
    // регистрируем синглтон
    if (Instance != null)
    {
      Debug.LogError("Несколько экземпляров SoundEffectsHelper!");
    }
    Instance = this;
  }

  public void MakeExplosionSound()
  {
    MakeSound(explosionSound);
  }

  public void MakePlayerShotSound()
  {
    MakeSound(playerShotSound);
  }

  public void MakeEnemyShotSound()
  {
    MakeSound(enemyShotSound);
  }

  // Играть данный звук
  private void MakeSound(AudioClip originalClip)
  {
    // Поскольку это не 3D-звук, его положение на сцене не имеет зн
    AudioSource.PlayClipAtPoint(originalClip, transform.position);
  }
}
```

Добавьте этот скрипт в игровой объект и заполните его поля со звуковыми клипами:



Затем выполните:

SoundEffectsHelper.Instance.MakeExplosionSound(); в "HealthScript", только после воздействия частиц.

SoundEffectsHelper.Instance.MakePlayerShotSound(); в "PlayerScript", сразу после weapon.Attack(false);.

SoundEffectsHelper.Instance.MakeEnemyShotSound(); в "EnemyScript", сразу после weapon.Attack(true);.

Запустите игру и прислушайтесь. Да, у нас теперь есть звуки и музыка!

Этот метод годится для небольших игр. Для большого игрового проекта он, скорей всего не подойдет, так как вы не сможете легко управлять сотнями звуков.

Мы только что узнали, как использовать звуки и музыку в нашей игре. Теперь давайте добавим меню, чтобы мы могли начать и перезапустить наш уровень.

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Практическое занятие № 8 Анимационные клипы.**

**Выполнив работу, Вы будете:**
***уметь:***
- анимировать персонажа
- воспроизводить анимацию при возникновении событий

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Создать и добавить босса в игру
2. Анимировать босса
3. Добавить анимацию покоя, получения урона, атаки

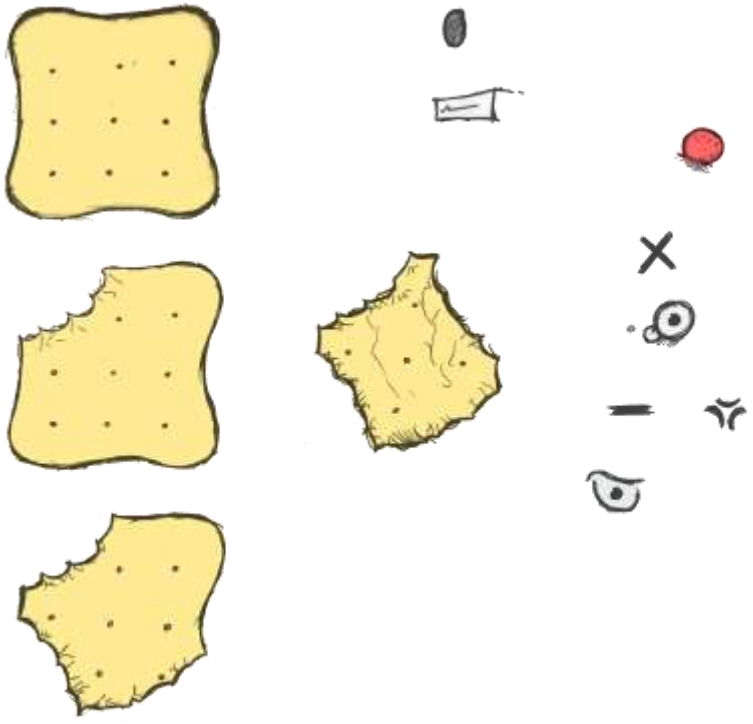
**Порядок выполнения работ:**

На текущем этапе есть один аспект, которому нам стоит уделить больше внимания - анимация. Даже если визуальный стиль может оправдать отсутствие анимации (ведь мы используем растрированные рисунки), мы все равно можем их анимировать. Давайте добавим нового врага, супер-босса, и создадим для него эффекты анимации. Анимации в Unity состоят из нескольких элементов:

Несколько анимационных клипов, определяющие ключевые кадры для каждого анимированного свойства.

Контроллер анимации, определяющий порядок и переход клипов объекта (или префаба).

Спрайт состоит из нескольких изображений: тело, глаза и пр. Сохраните рисунок ниже, он нам понадобится.

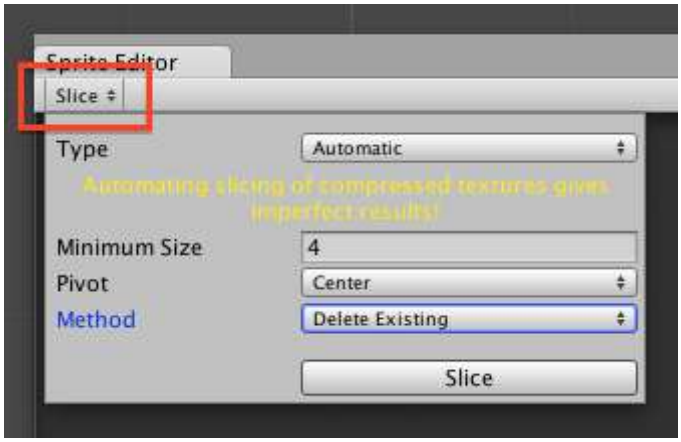


Импортируйте изображение в Unity.

Установите "Sprite Mode" значение "Multiple" в "Inspector".

Нажмите на кнопку "Sprite Editor"

Используйте функцию автоматического разрезания (32 должно идеально подойти):




Не забудьте "Применить" нарезки.

Супер-босс

Босс выполнен из четырех частей:

Тело

Два глаза

Рот

Чтобы иметь правильную конфигурацию, создайте пустой объект игры. Тогда:

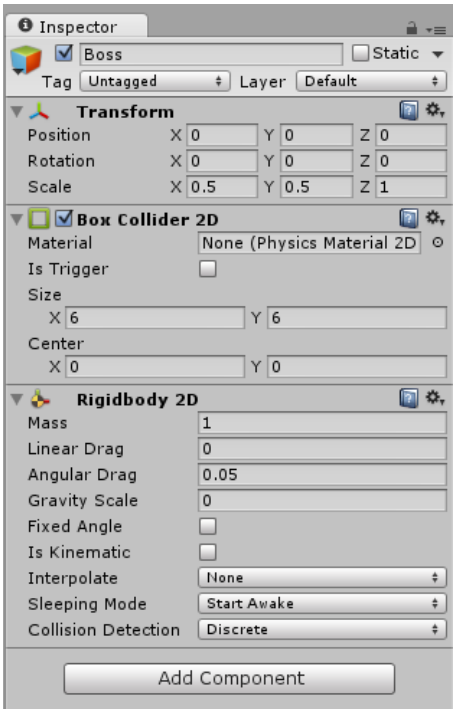
Назовите его "Boss".

Добавьте "Rigidbody 2D" без гравитации/постоянных углов.

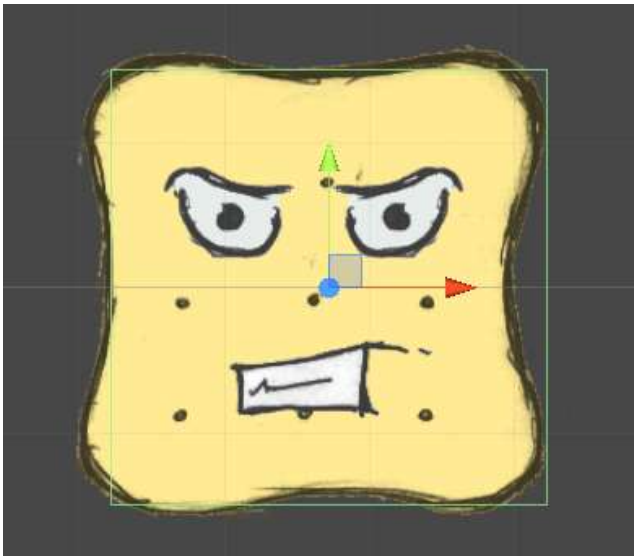
Добавьте "Box collider 2D" с размерами (6, 6).

Установите его масштаб (0.5, 0.5, 1).

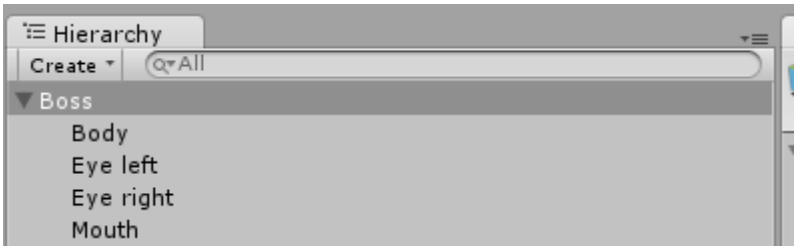
Этот объект ничего не отображает. Поэтому здесь и нет "Sprite Renderer".



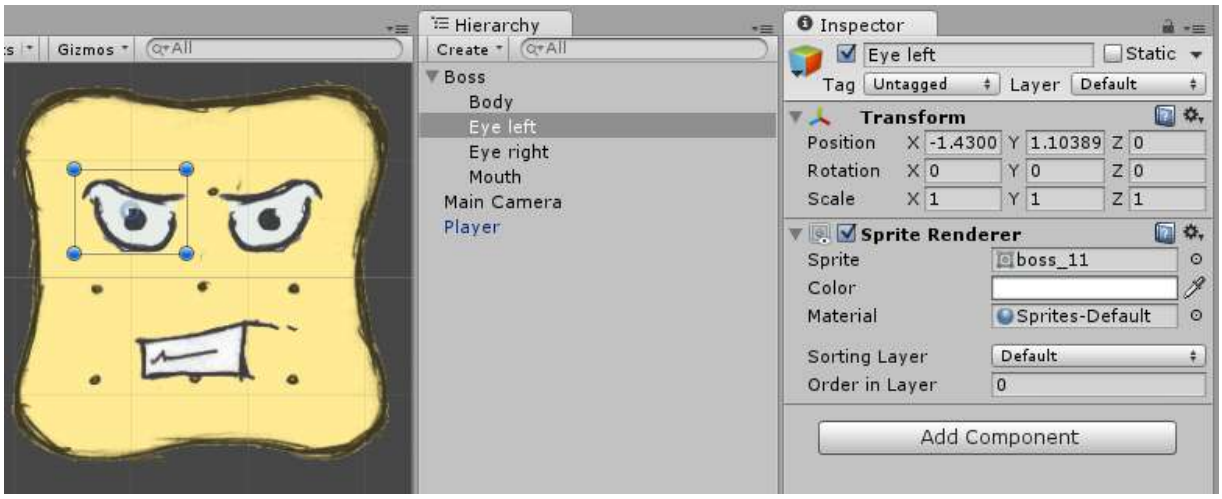
Создать 4 пустых игровых объекты, как дети объекта "Boss". Для каждого из них добавьте "Sprite Renderer" и выберите соответствующее изображение (тело, глаза или рот). Измените их положение, чтобы получить что-то вроде этого:



Используя также это в качестве иерархии:

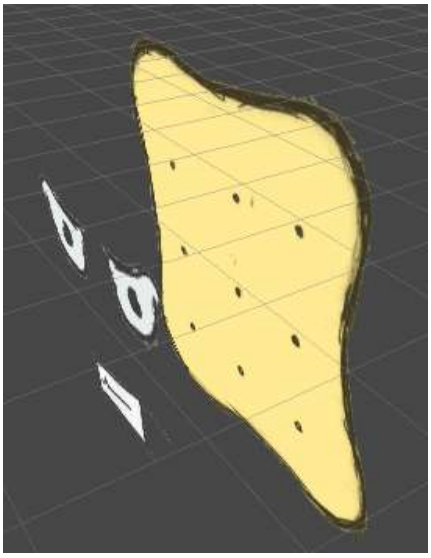



Например, левый глаз игрового объекта позиционируется с помощью мыши в редакторе. Вот что у нас:




*Совет*: У нас есть только один спрайт для глаз. Мы просто перевернем другой спрайт с помощью зеркального отражения. Для этого в Unity установите значение масштаба на противоположное. В частности, мы установим масштаб глаз равным (-1, 1, 1). Вы можете использовать этот простой совет везде и даже у свойства y (зеркальное отражение по вертикали).

Установите объект "тело" немного глубже, чем остальные, вот так: (0, 0, 1). 3D-вид покажет наши слои:



Используя объект со множеством подспрайтов, мы можем манипулировать ими отдельно. Сохраните объект "Boss", как префаб. Теперь мы готовы анимировать его!

Старая школа анимации предлагает нам анимировать объект с помощью одного листа спрайтов и изменять положение всего изображения каждые N секунд. Этот способ также работает в Unity, но это не цель этой главы..

Анимационные клипы

Мы собираемся создать несколько клипов для определения состояния нашего босса:

Idle — просто плавающий вокруг.

Attack — запуск снарядов.

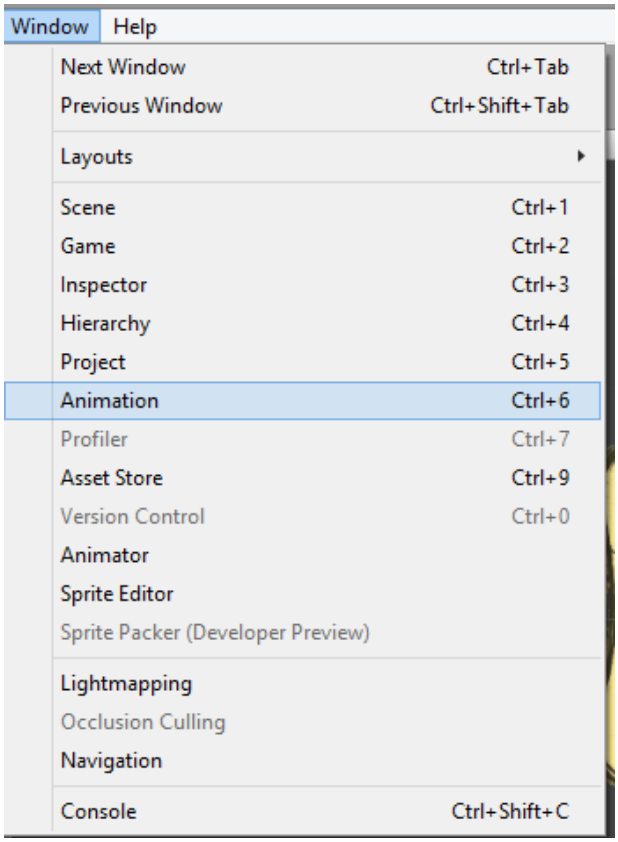
Hit — попадания выстрела.

Для этого урока мы опишем первую анимацию настолько детально, насколько это возможно. Затем мы отобразим остальные с помощью гиф-анимации, а вы получите задание воспроизвести их. Это хороший способ узнать, как управлять инструментом анимации. И, конечно, сделать их лучше. :)

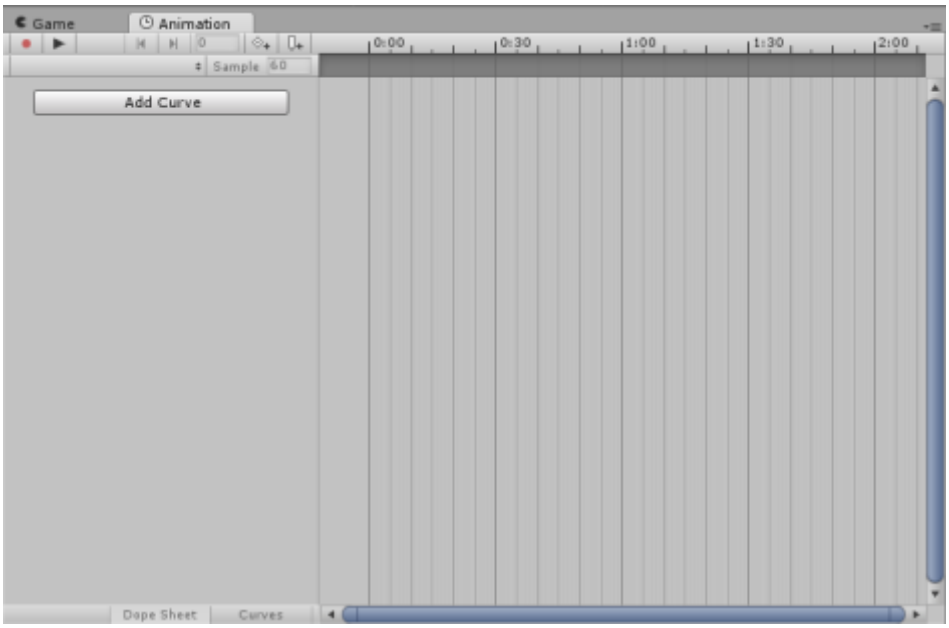
*Совет*: Когда вы работаете над анимацией, делайте это в пустой сцене. Таким образом, вы можете четко видеть, что происходит и ни на что не будете отвлекаться. Здесь мы будем работать в сцене под названием TestAnimations. Эта сцена как раз для дизайнеров и никогда не должна быть включена или запущена в окончательном проекте.

Idle - анимация. Создаем новый клип

Open the "Animation" window (**not "Animator"!**):




Новое окно должно отображаться:




Выберите объект boss в вашей сцене (экземпляр, если это необходимо, вы не можете работать непосредственно на Prefab). Теперь создайте новый клип. Опция доступна в списке анимация:


Создайте папку "Animations" и сохраните новый клип как "Boss_Idle". Клип будет выбран в анимационной панели.

*Новый клип*: На самом деле, с помощью кнопки «Создать новый клип", с точки зрения анимации, Unity делает три вещи. Во-первых, он создает "Аниматор-Контроллер" для выбранного игрового объекта (в данном случае - это босс). Потом он добавляет к этому аниматору «Анимацию». «Аниматор» также добавляется к игровому объекту в качестве компонента, указывающего на «Аниматор-Контроллер». Если у игрового объекта уже есть «Аниматор», к нему просто добавляется анимация. Загляните в папку «Анимации»: рядом с анимацией "Boss_Idle" вы увидите аниматор “Boss”. Мы будем говорить об этом чуть позже, пока не трогайте его.

Добавление ключевых кадров

Для начала работы над клипом, нажмите на кнопку "Record" (красная точка) в верхнем левом углу вкладки "Анимация". Теперь, все, что вы будете делать в сцене для нашего объекта (босса) будет записываться в качестве ключевого кадра анимации.

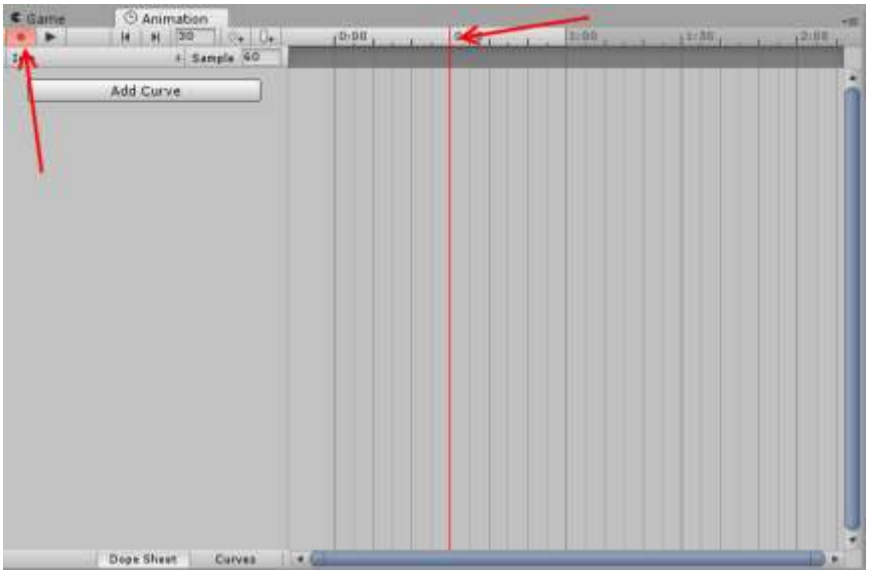
*Внимание*: Клик на линейке времени включит кнопку "Запись". Будьте осторожны! Этот способ можно использовать для ускорения работы: просто кликните на время и добавьте необходимые изменения (запись начнется, когда на линейке появится красный маркер).

*Ключевой кадр* - это *набор значений* , относящихся к определенному моменту во *времени*.

Чтобы добавить ключевой кадр:

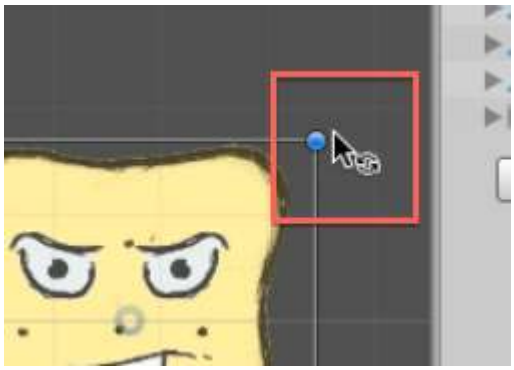
Вы можете нажать на кнопку "Запись"

Выберите время, нажав на линейке времени. Следует перевести селектор (красная линия).

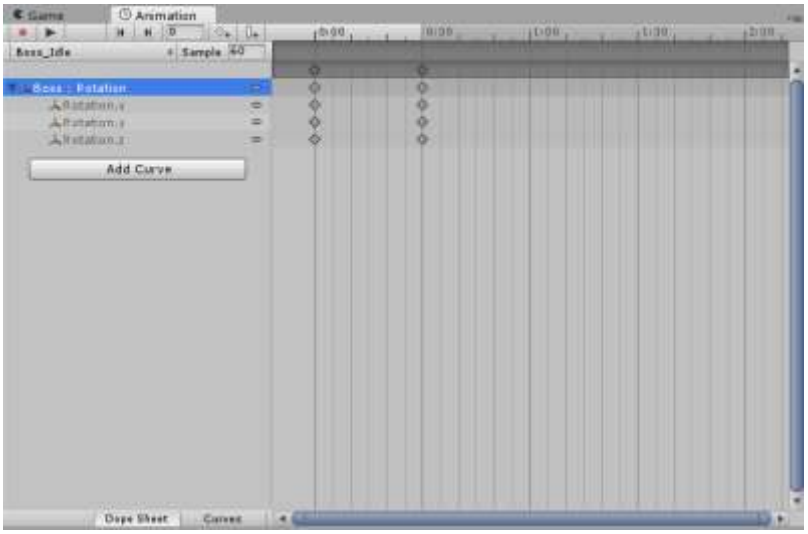



Выберите объект "Босс" в вашей сцене. В 0:30 измените поворот на (0, 0, 30).

*Поворот*: Во вкладке "Сцена" (Scene) если вы наведете указатель на уголок рамки выделенного объекта, то вы увидите как изменился указатель мыши. Удерживая и перемещая мышь, вы можете изменить угол поворота выбранного объекта.

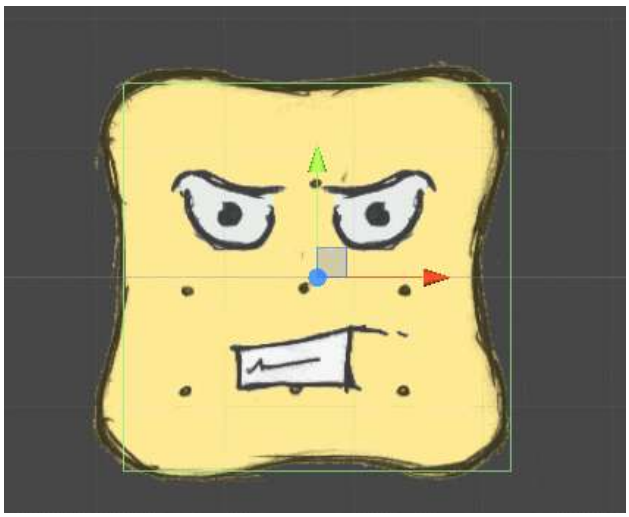


Вы можете видеть, что ключевой кадр создан:




*Горячие клавиши*: клавиша alt может быть использован для перемещения по линейке времени. Колесо мыши можно использовать для увеличения или уменьшения масштаба, наведя курсор на необходимый объект и прокручивая его вверх или вниз.

Если вы нажмете кнопку "Play" во вкладке "Анимация" (Animation) то, вы увидите анимацию в панелях "Игра" (Game) или "Редактор" (Editor):

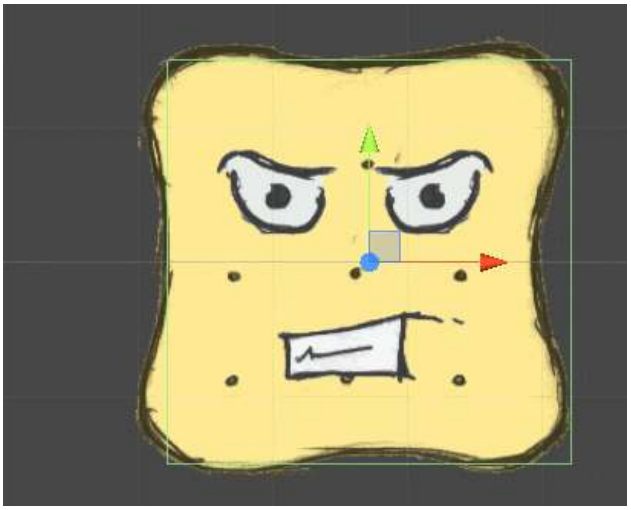


Начнем! Наш Босс выглядит глупо, поэтому мы добавим два новых ключевых кадра:

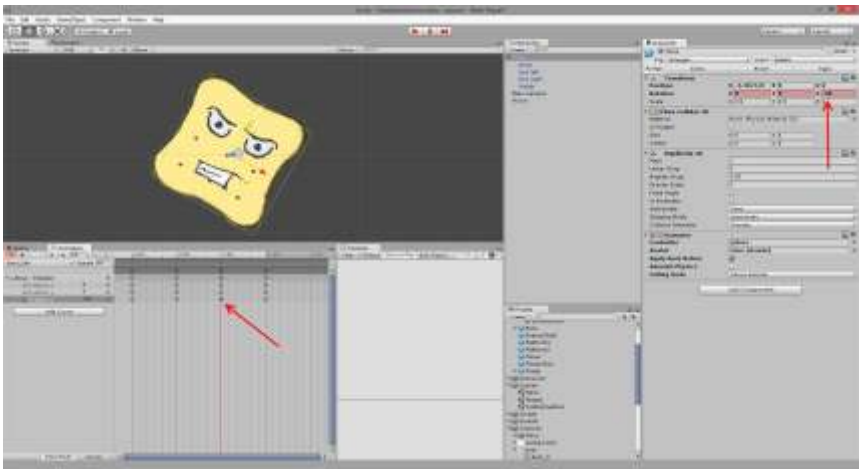
1:00 - "Boss" object rotation of (0, 0, 30).

1:30 - "Boss" object rotation of (0, 0, 0).

Теперь, анимация отображается плавно (потому что вращение происходит одинаково в начале и в конце).



Если вы кликните на значение ключевого кадра в редакторе, Unity перейдет в режим записи и выделит изменяемые свойства красным в «Инспекторе».



*Изменение свойство объекта "Boss"*: Если вы измените свойство родительского объекта (объект "Boss" в данном случае), будьте осторожны, особенно со свойством "Позиция" ("Position"). Например, если вы измените свойство "Position" для объекта "Boss", его положение в сцене заменится положением в анимации каждый раз, когда вы будете ее проигрывать. Он может блокировать объект. В большинстве случаев вам нужно будет изменить детские свойства, а не сам контейнер. Здесь мы работаем с родителем, потому что нам надо, чтобы все вращалось вместе. Но может быть, нам нужен еще и пустой родительский объект, к которому не применяется анимация.

Играtv с кривыми

Между каждым ключевым кадром **Unity** использует линейную интерполяцию для вывода промежуточных значений. Если мы говорим:

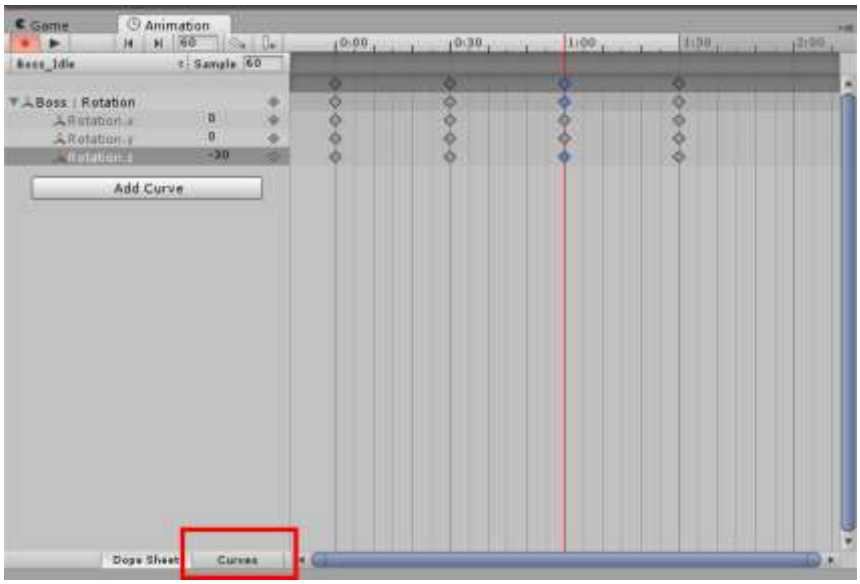
Время 0:00, значение 0.

Время 0:30, значение 30.

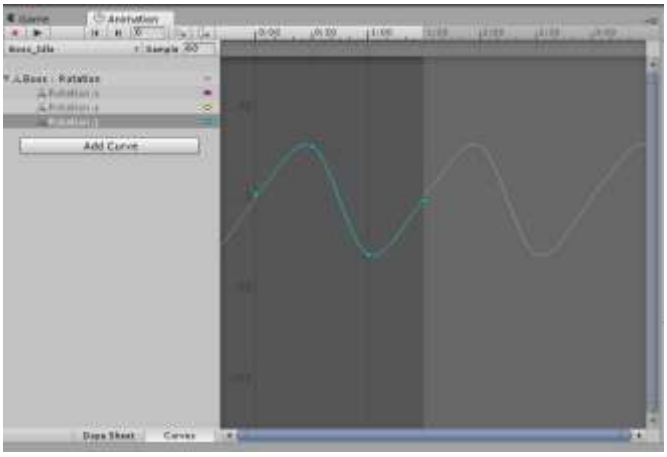
Тогда Unity может вывести:

Time 0:15, value 15.

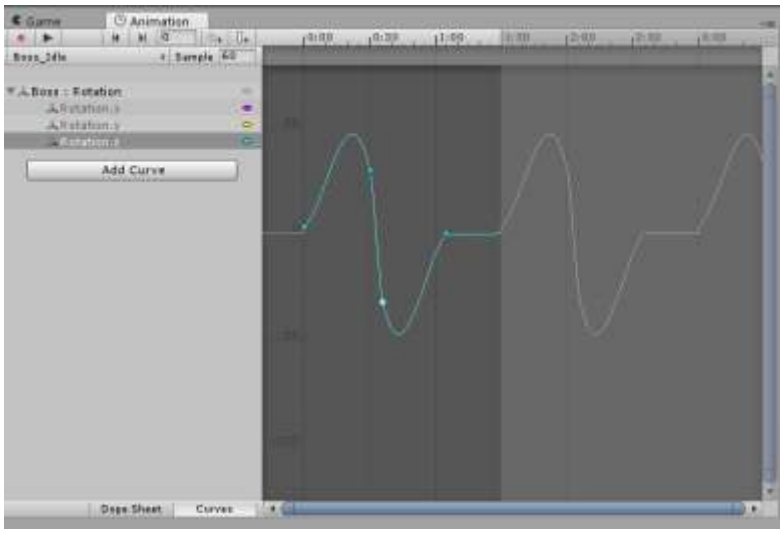
Тем не менее, вы можете применить нелинейную интерполяцию (например, начинается быстро и заканчивая медленно). Для этого есть два решения: можно добавить другие ключевые кадры между ними, или поиграть с *Curves*. Это специальная вкладка, которую можно переключать в левом нижнем углу вкладки "Анимация":



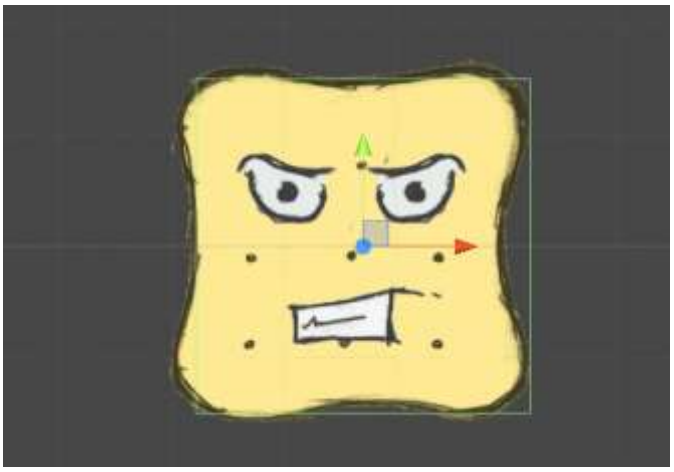

Сейчас наша нынешняя простоя анимация выглядит как обычная синусоида:



Мы можем поиграть с кривой путем перетаскивания точек. Он будет обновлять ключевые кадры и связанные с ними значения. Например, если мы внесем немного беспорядка:



Анимация выглядит уже по-другому: она воспроизводится быстрее, с паузами между циклами.



Tweaks

Мы просто сделали анимацию, используя вращение, но вы можете обновить все свойства объекта и его детей. Например:

Вы можете обновить позицию (*не делайте этого на контейнере*).

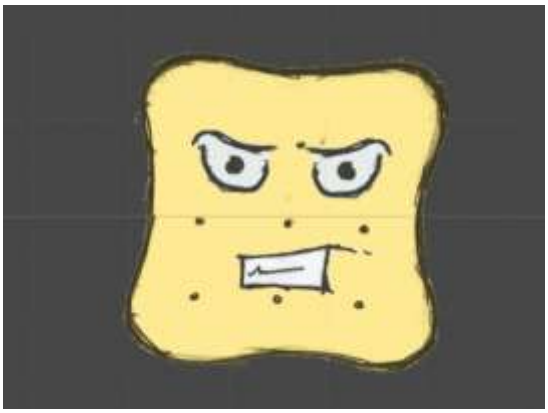
Изменить спрайт (из "Sprite Renderer").

Добавить и удалить детей.

Включить или выключить компоненты.

Обновить размер коллайдера, чтобы он соответствовал анимации.

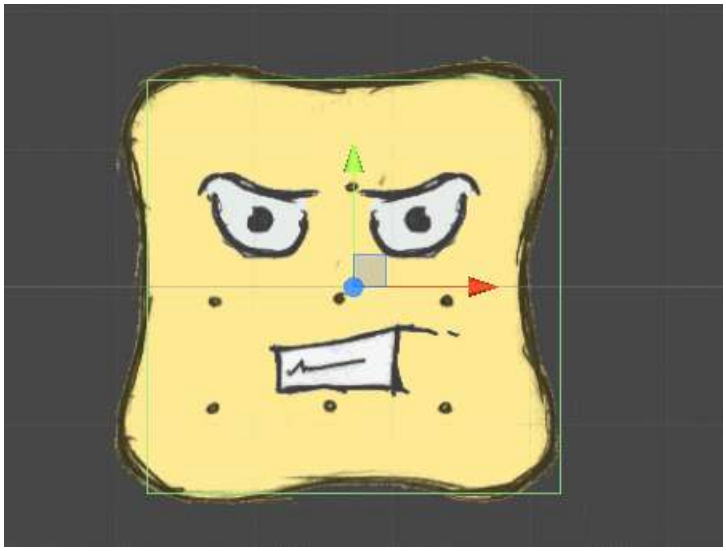
Теперь, настало время для экспериментиров! Вот что у вас в конце концов должно получиться:



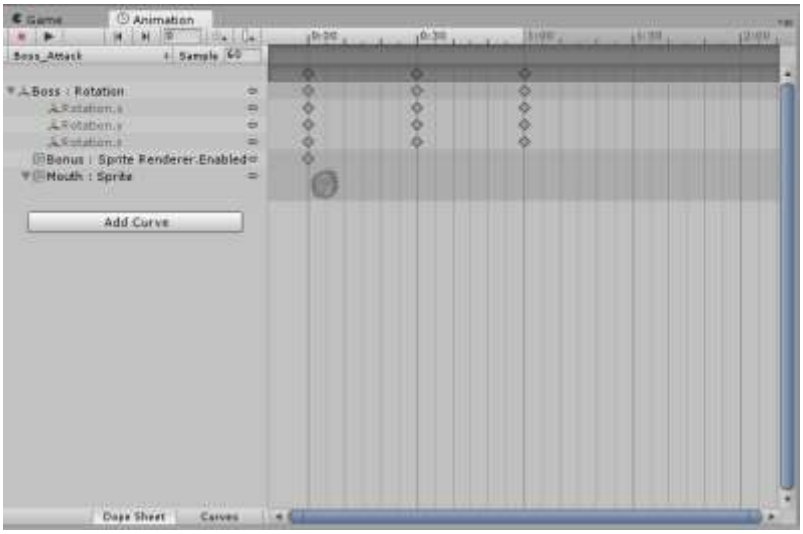
Анимация атаки и удара

Для этих анимаций мы покажем что вы можете достичь, но мы рекомендуем вам весело провести время и сделать свои собственные. Это хорошая возможность проверить, что вы узнали раньше. Просто убедитесь, что имя клипов - "Boss_Attack" и "Boss_Hit". In your "Boss" Prefab, добавьте нового ребенка спрайта "Bonus". Этот объект будет служить для отображения дополнительного изображения, когда это потребуется. Большую часть времени оно будет скрыто (например, в режиме ожидания анимации).

Атака

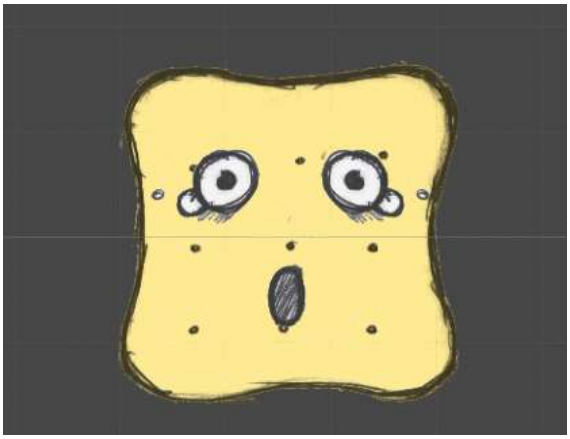


Ключевые кадры должны выглядеть следующим образом:



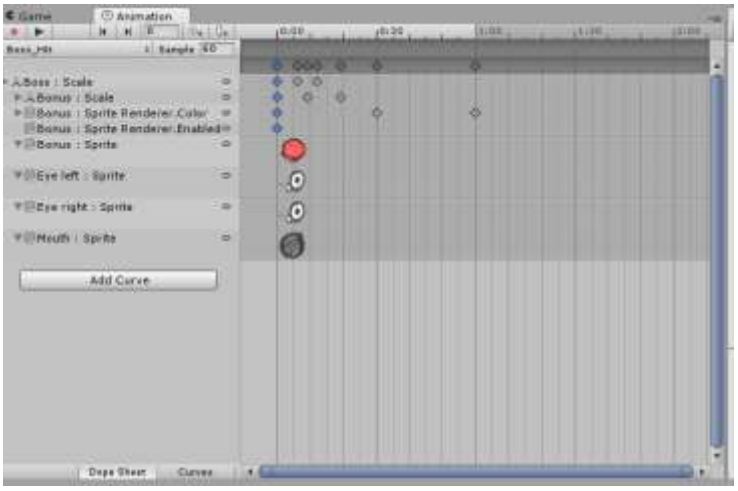

Посмотрим на детей "Mouth" (другое изображение) and "Bonus" (включен).

Удар



В зацикленном виде эта анимация выглядит странно, но наша цель – быстрая анимация, которая проигрывается всего один раз в ответ на выстрел игрока.

Ключевые кадры:




Чтобы отключить цикл, нажмите на анимацию "Boss_Hit" на панели "Проект" ("Project"). Затем, снимите флажок "Loop Time". На первый взгляд может показаться что ничего не изменилось. Запустите игру и смотрите в окно "Аниматор" ("Animator"), чтобы увидеть разницу.

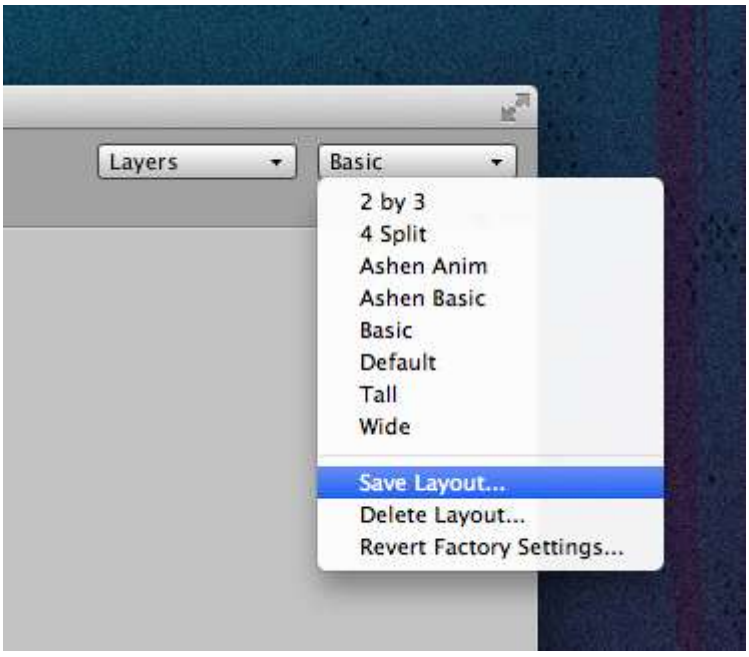
Спец-слой

Мы рекомендуем использовать выделенный слой для работы с анимацией. Это очень удобно:

Вкладка "Сцена" (Scene) находится рядом с вкладкой "Анимация". Вы можете нажать на спрайт, чтобы выбрать и изменить его. Если анимация записывается, вы увидите новый ключ на линейке времени после каждого изменения.

В режиме "Игра" всегда можно отобразить анимацию, нажав на кнопку "Play" в режиме "Animation" (без запуска игры).

Не забудьте сохранить макет чтобы использовать его в будущем.

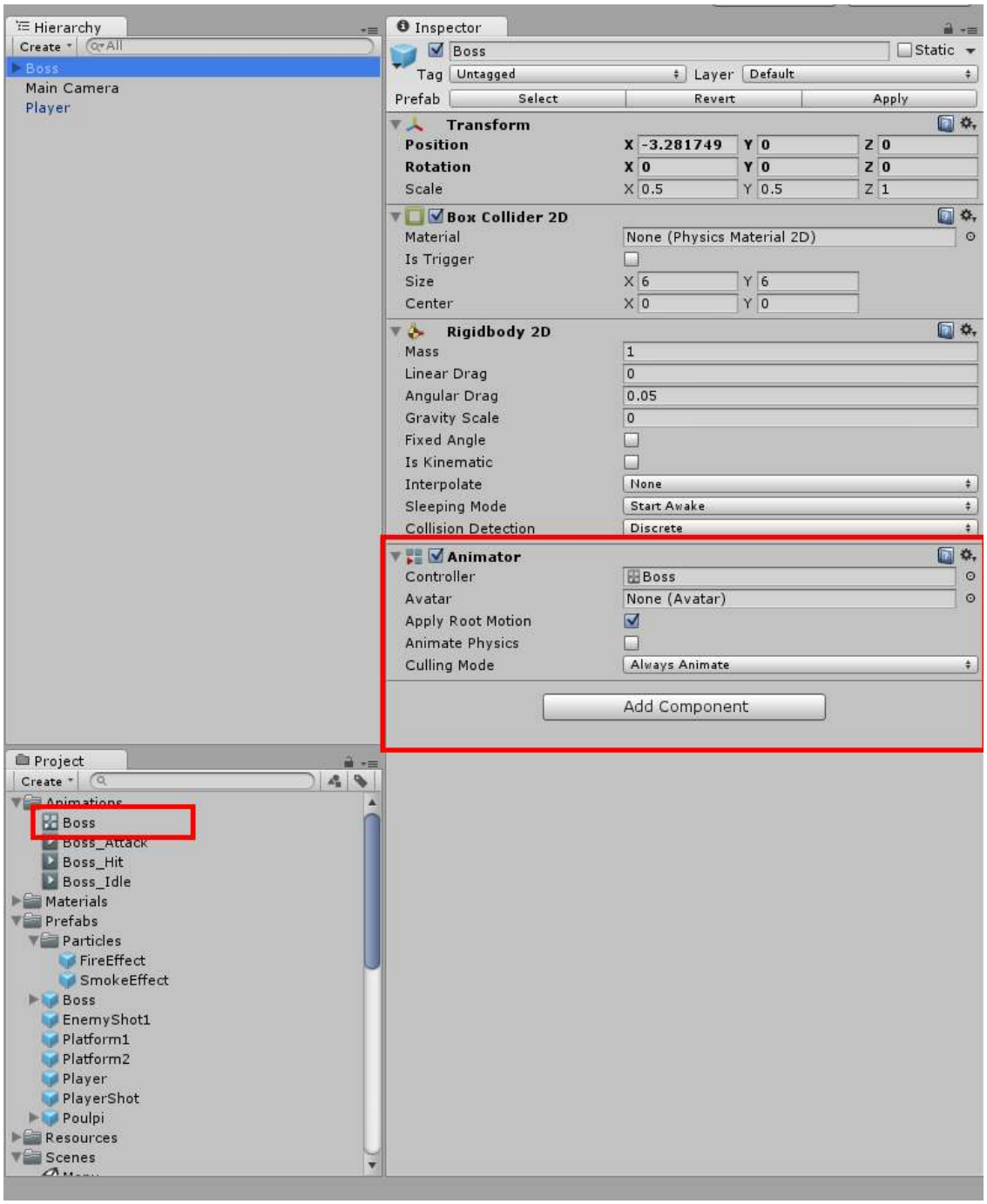



Мы сделали несколько анимационных клипов для нашего нового врага. Но сейчас, они нигде не используются. Действительно, клип просто слой, содержащий информацию о том как оживить объект. Нам потребуется *"Аниматор"* (Animator). "Animator" является компонентом, который вы перенесли на объект, ссылающийся на контроллер Аниматор (Animator Controller). "Контроллер Аниматор" - контроллер, который позволяет выполнять работу с анимациями в аниматоре - добавлять анимации, слои, и т.д. Давайте посмотрим как им пользоваться.

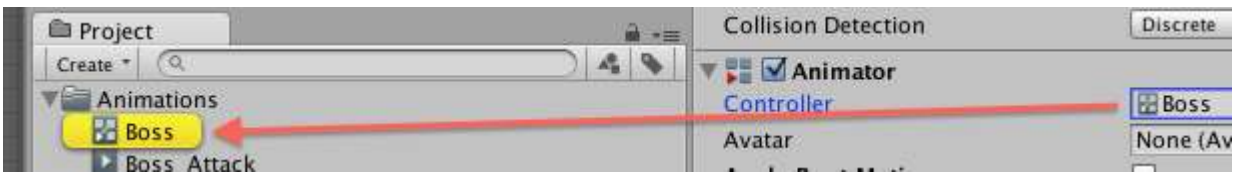
Аниматор (Animator)

Вы, возможно, уже обратили внимание что объект "Boss", над которым мы работали, автоматически получил новый компонент: "Аниматор". В то же время, в папку "Animations" был добавлен новый файл, названный так же, как Ваш объект. Это контроллер Аниматор.

Теперь взглянем на свойство "Controller" компонента "Animator":




Файл "Boss" является контроллером. Вы можете проверить это, кликнув на поле контроллера — выделится прикрепленный файл:



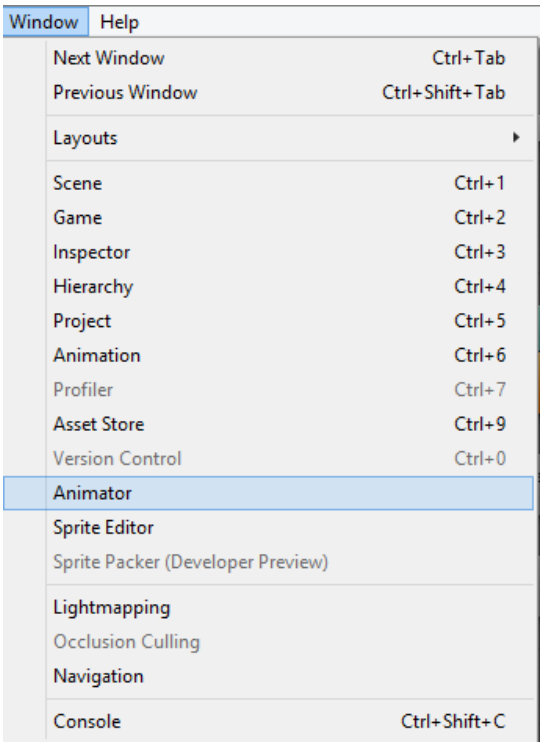

С помощью этого контроллера мы будем определять, как и когда **Unity** должно проиграть анимационные ролики. В принципе, компонент "Аниматор" - просто связь между объектом и

контроллером. Этот компонент можно извлечь и манипулировать им с помощью кода. Если ваш префаб "Boss" не имеет компонент "Аниматор", добавьте его вручную и перетащите контроллер "Boss" внутрь свойства.

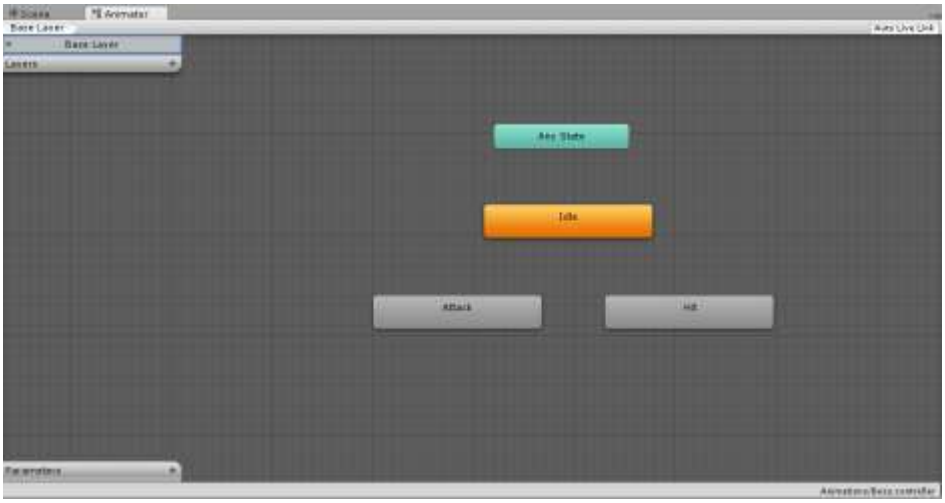
Компонент "Аниматор" имеет и некоторые другие параметры. "Apply Root Motion" следует отключить при использовании анимации, как мы делаем в этой статье. Но здесь это не важно, потому что у нас очень простой объект без гравитации.

Внутри аниматора

Теперь мы должны открыть окно, называемое "Аниматор" (Animator) (не спутайте с "Анимация" (Animation))! Для этого Вы можете дважды щелкнуть по файлу контроллера ("Animations/Boss"), чтобы открыть контроллер, или вы можете найти его в меню "Окно" (Window):

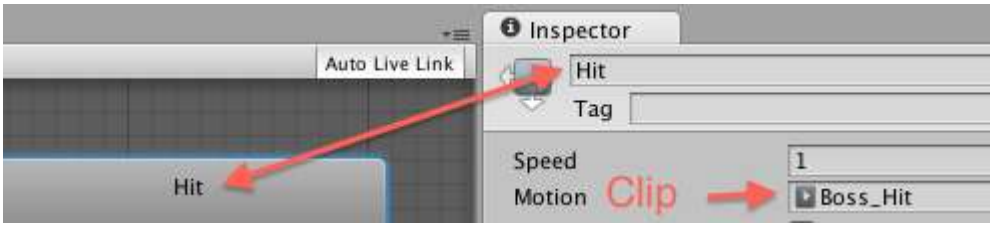



Вы должны получить что-то вроде этого:

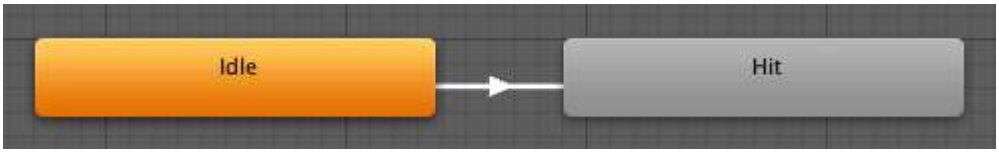


Вы можете видеть, что у нас есть *состояния* - *states* (прямоугольники), созданные автоматически с нашими клипами, плюс один специальный по имени "Any State". Помните, когда

вы использовали кнопку "Создать новый клип" во вкладке "Анимация", **Unity** при этом добавило состояние в контроллере объекта, связанное с созданным Вами файлом анимации. Пройдитесь по каждому состоянию аниматора и переименовать их, удалив префикс "Boss_":




"Аниматор-контроллер" – это *машина с конечным числом состояний*. Каждое состояние **может** быть анимацией, и вы можете определить переходы между ними. Переход рассказывает Unity, когда и почему она должна перейти от одного состояния в другое.




На этой картинке, мы создали связь между двумя состояниями. Чтобы проанимировать объект с помощью анимации "Hit", сначала нам нужно использовать состояние "Idle". Мы сосредоточимся немного больше на переходах чуть позже, а сейчас давайте посмотрим на три разных типа состояний.

1. Дефолтное состояние

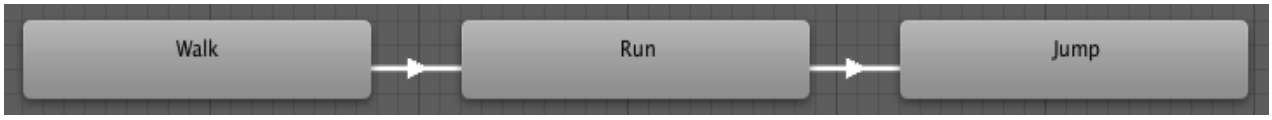
*Оранжевое состояние* является дефолтным: это изначальное состояние, в котором запускается игра.


В этом случае, "Idle" состояние по-умолчанию одно (если это не так, щелкните правой кнопкой мыши и выберите "Установить по умолчанию" (Set As Default)). Это означает, что, когда игра началась, объект "Boss" будет автоматически воспроизводить "холостую" анимацию (бесконечно, если включить "Loop Time" в "Inspector" — как и должно быть в данной игре).

2. "Любое состояние"

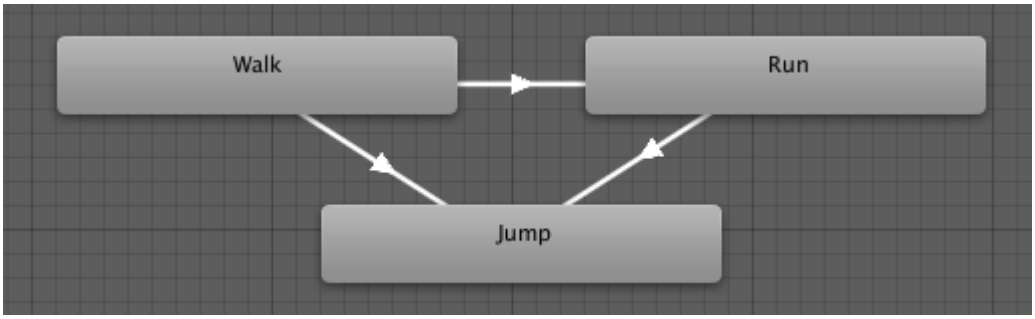
*Зеленое состояние*, которое называется "любое состояние" – особый случай.


Это хороший способ упростить наш контроллер. Как можно догадаться по названию, оно представляет собой каждое состояние в заданный момент времени. В контроллере босса это состояние одновременно представляет собой состояния "Idle", "Hit" и "Attack". Поясним это на нескольких примерах. Предположим, мы используем следующую машину с конечным числом состояний:

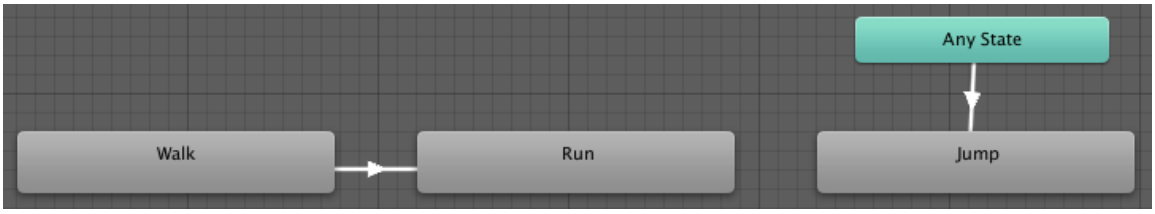



Для того, чтобы наступило событие "Jump" (прыжок), вы должно выполняться "Walk" (ходить), затем "Run" (Выполнить) и лишь потом "Jump". Это означает, что анимация "Jump" не будет воспроизводиться, если ваш объект до этого не находится в состоянии "Run".

Это не идеально, не так ли? Мы должны переходить в состояние "Jump", когда находимся в состоянии "Walk" тоже! Хорошо, давайте попробуем сделать это:



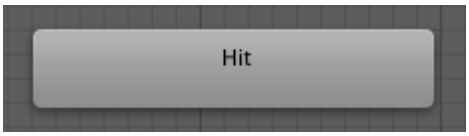

Отлично, теперь мы можем перейти в "Jump" из состояний "Walk" и "Run". Тем не менее, если вы добавите несколько новых состояний, то вам потребуется создать еще переходы в "Jump" от каждого состояния. Решение этой задачи заключается в использовании "Any State":



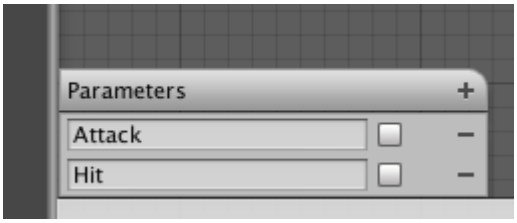

Благодаря этому контроллеру, «любое состояние» может перейти в "Jump". Разве это не гениально?

3. Нормальное состояние

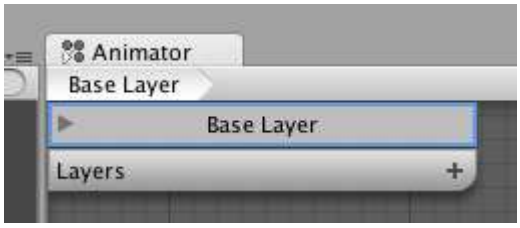
*Серые состояния* являются нормальными и содержат одну анимацию либо не содержат ее вообще.



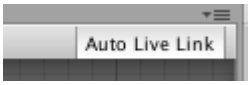

На левом нижнем углу вкладки "Аниматор", вы можете найти список параметров. Эти параметры используются для условий переходов. Подробнее об этом ниже.




В верхнем левом углу, вы можете увидеть слои. This is a way to have multiple state machines for one object. Мы не будем использовать эту функцию, в этом руководстве.

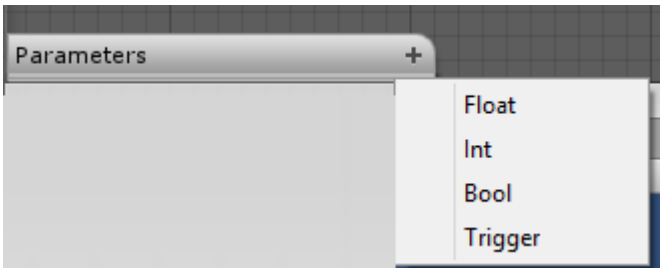



И, наконец, кнопка "Auto live Link" в правом верхнем углу, который позволяет видеть в режиме реального времени состояния, которых в настоящее время проигрываются. Оставьте ее включенной.




Добавление параметров

Параметр – это значение или триггер для нашей машины с конечным числом состояний. В дальнейшем мы будем использовать их в условиях наших переходов. Доступно 4 типа параметров:




Int — просто целое число.

Float — просто вещественное число.

Bool — true или false значение.

Trigger — флаг, который остается включен, пока он не используется. Затем он будет снят.

Числа понадобятся нам для особых случаев вроде горизонтальной или вертикальной скорости. Вам можете понадобиться другая анимация для ходьбы или бега, но все они зависят от скорости движения игрока, которая выражается с помощью определенного параметра.

Для нашей игре, давайте добавим два новых параметра:

"Hit" - **trigger**

"Attack" - **boolean**


*(К сожалению, редактор внешне не различает триггеры и булевские значения)*

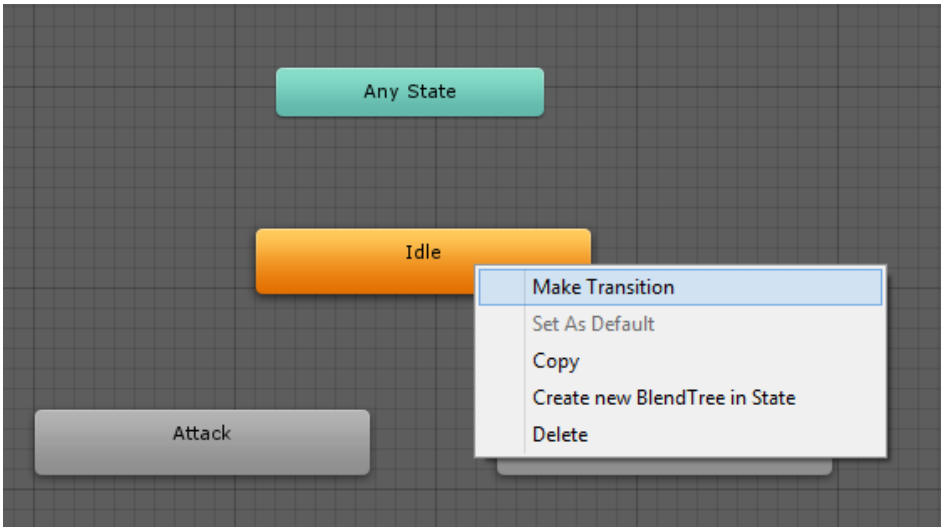
Теперь, давайте посмотрим зачем они нам.

Переходы

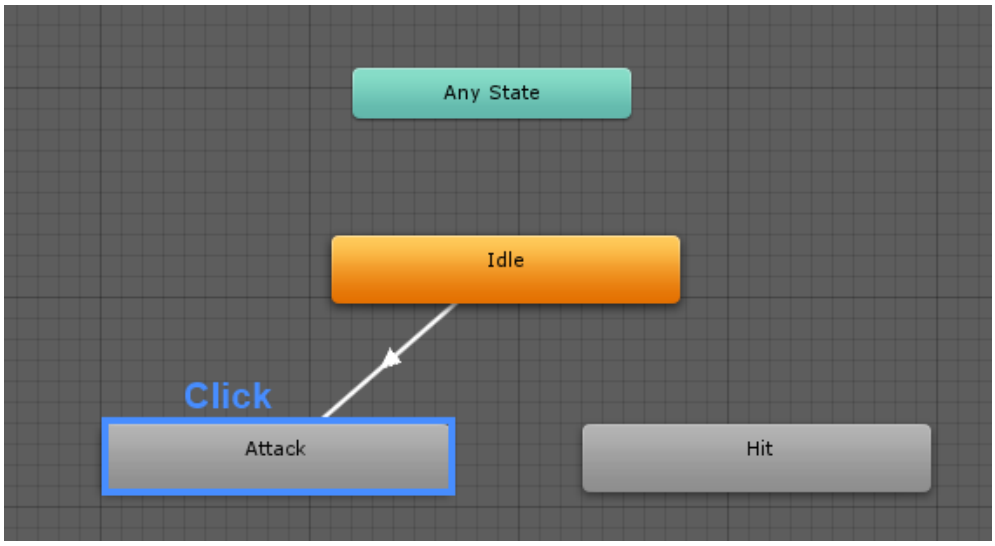
Переход является связующим звеном между двумя состояниями. Он определяет, как именно одно из них должно превращаться в другое.

1. "Idle to Attack"

Чтобы создать переход, нажмите правой кнопкой мыши на состоянии источника. Давайте сначала проделаем это для "Idle to Attack": щелкните правой кнопкой мыши на состоянии "Idle", а затем выберите "Make transition":

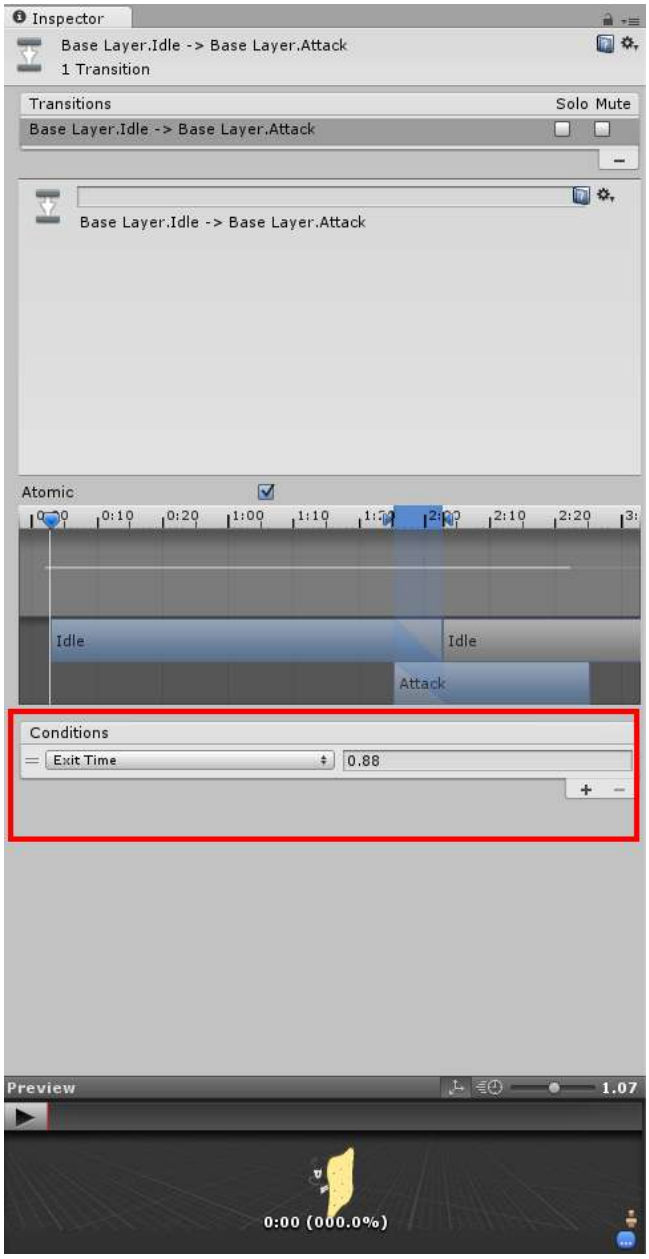



Теперь кликните на состояние назначения:



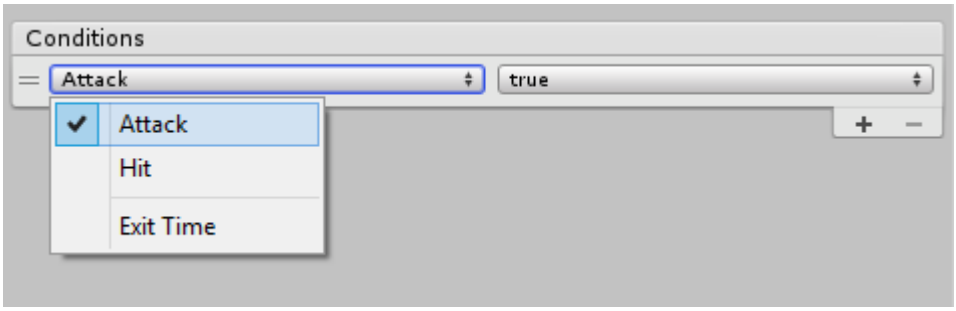

Вот и все!

Вы можете выбрать переход, нажав на ссылку."Инспектор" (Inspector) покажет много интересных параметров, в частности начальные условия:

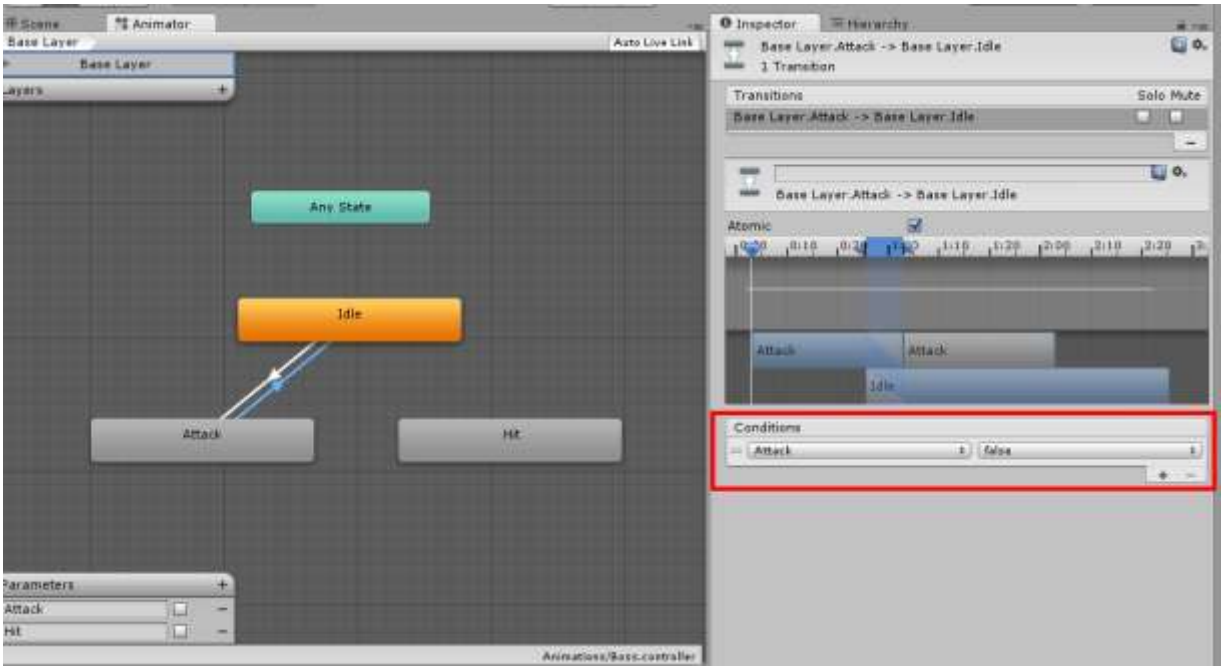



*Exit Time*: Условие "Exit Time" – дефолтное условие для перехода. Оно значит, что переход может быть осуществлен, когда закончится исходная анимация.

Это то, что мы будем редактировать. Измените "Exit Time" для параметра "Attack", который мы определили ранее.




Это условие означает: "Если значение Attack true, то воспроизвести анимацию "Attack". Точно так же, добавить переход между "Attack» и "Idle" с условием "Attack - false".




Это означает, "Если значение *Attack* равно false, то остановить анимацию "Attack" и вернуться в режим ожидания".

Как видите, нам пришлось определить значения для обоих переходов. В противном случае контроллер не вернулся бы к состоянию "Idle" после атаки.

Мы сделаем почти то же самое для Idle анимации.

2. "Idle to Hit"

Вы еще не забыли про "Any State"? Сейчас он нам понадобится. Сделайте два новых подключения:

От "Any State" к "Hit", условие Hit.

От "Hit" к "Idle", условие Exit Time.

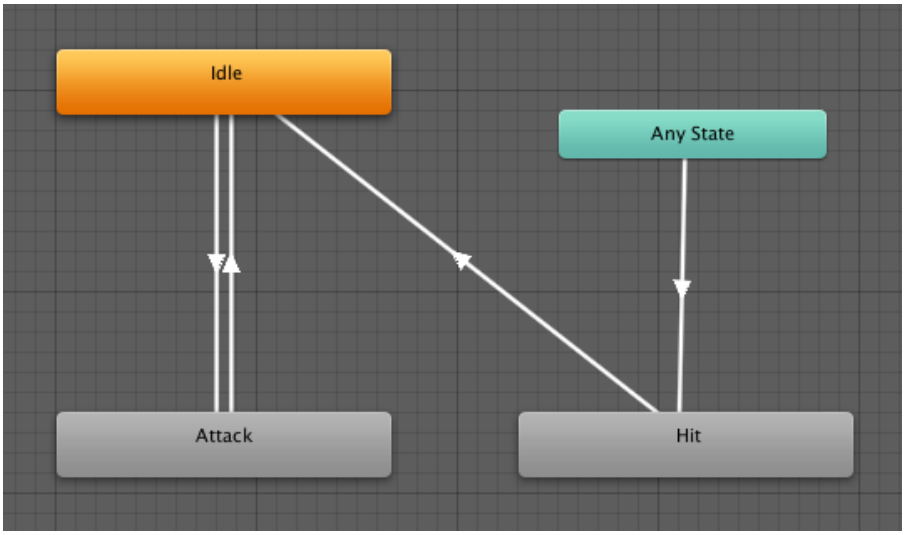
Если установлен триггер "Hit", мы проиграем один раз анимацию "Hit" и возвращаемся к "Idle".

"Any State" придется здесь как раз кстати, поскольку триггером для Hit являются такие состояния босса, как "Idle" или "Attack". Вместо того, что определять оба состояния, мы просто используем "Any State".

Как видите, когда вы используете триггер, не нужно указывать значение. Действительно, триггер - это просто способ сказать контроллеру: "Если состояние действительно - делай переход".

Конечный график

График нашей анимации должен выглядеть так:




Последнее, что нам понадобится – это код, с помощью которого все эти переходы происходили бы в самой игре (иначе аниматор останется в состоянии "Idle").

*График анимации*: Создание графика в аниматоре – вовсе не точная наука. В зависимости от того, какой код вы используете, чего хотите добиться или в какой последовательности выполняете те или иные действия, этот алгоритм может меняться. Например, в нашем случае мы могли бы осуществить переход к атаке из "Any State". Но поскольку мы используем всего три короткие анимации, это не имеет особого значения. По мере того, как ваш график будет расти, вам придется принимать решения, которые повлияют на вашу игру.

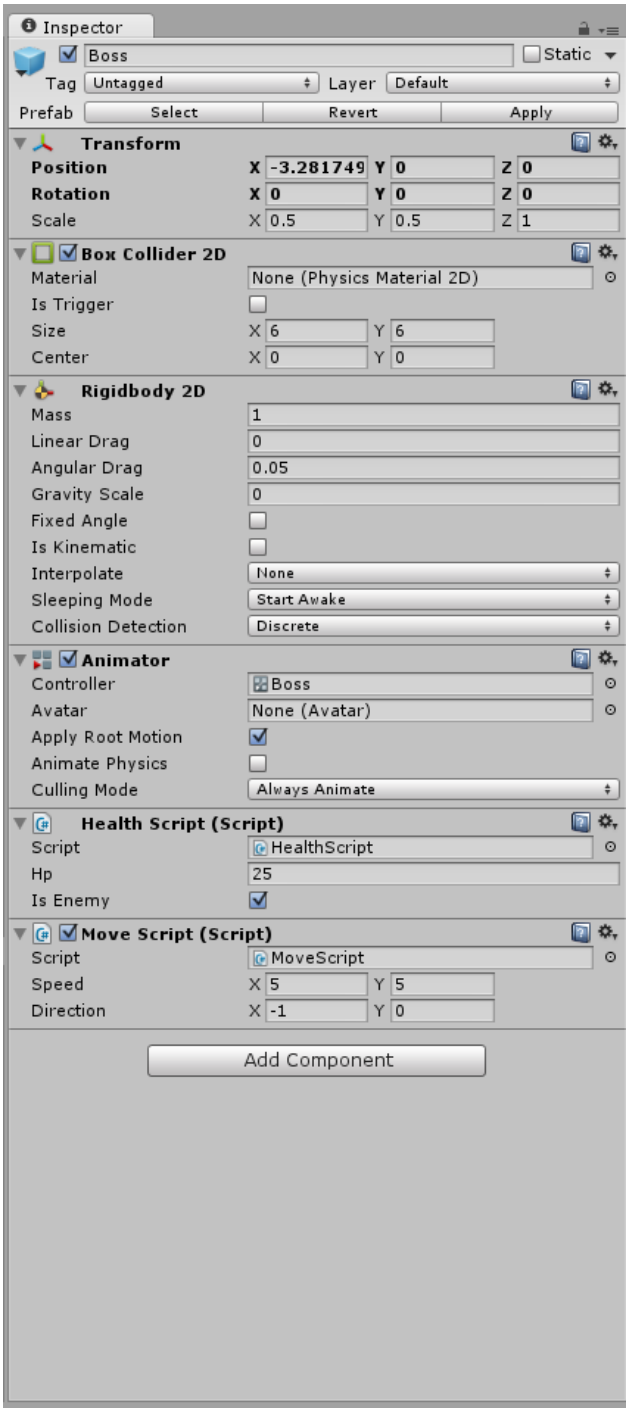
А где же Boss?

Прежде чем перейти к действительно интересному вопросу, нам нужно позаботиться о том, чтобы в игре был босс.

Не будем затягивать, ведь эта глава все-таки посвящена анимациям.

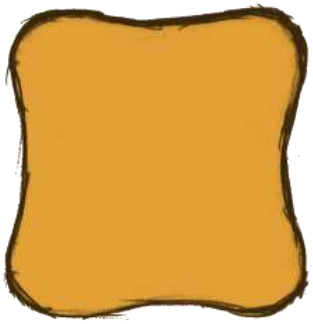
Добавьте "HealthScript" в Боссу, и задайте ему много очков здоровья (например, 50).

Добавить "MoveScript". Для хорошего перемещения попробуйте скорость (5, 5).

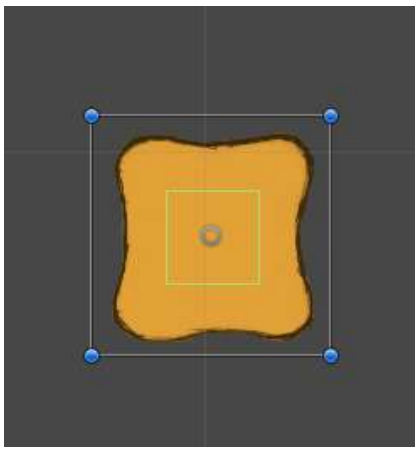


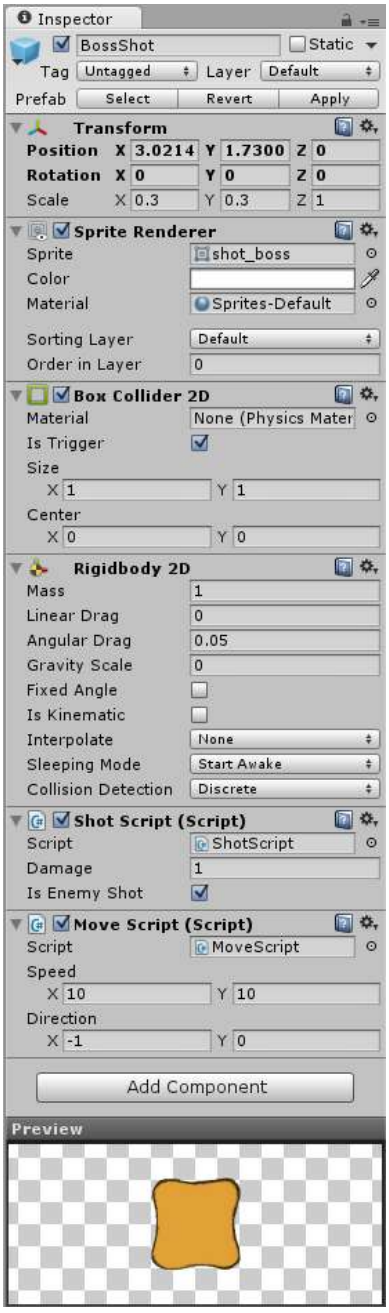

Снаряд

Нам нужен новый снаряд, которым Босс будет атаковать игрока. Продублирйте "EnemyShot1" и измените изображение на новое:



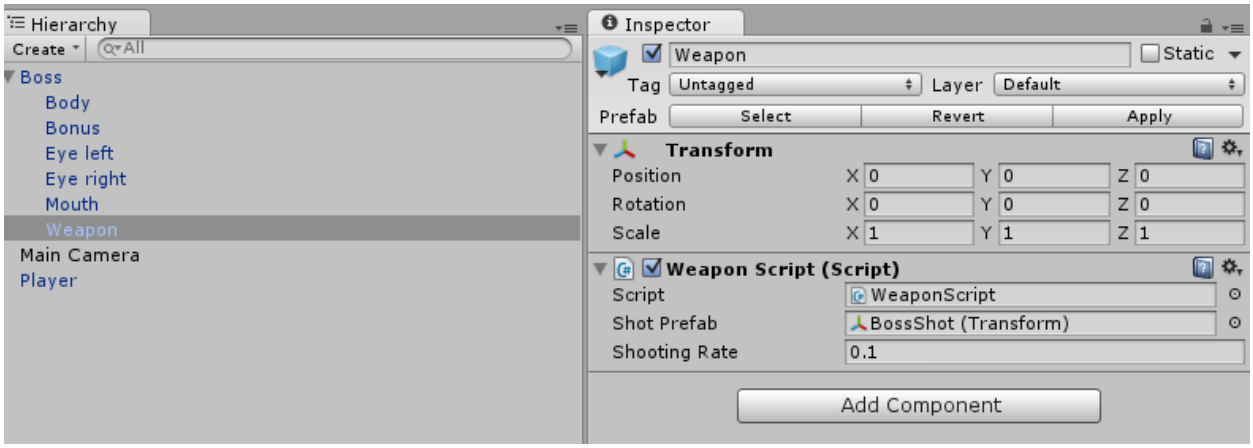
Далее установите масштаб (0.3, 0.3, 1) и сохраните как новый префаб. У Вас должно получиться что-то вроде этого:






Оружие

Точно так же, как мы делали с "Poulpi", присоедините к боссу ребенка оружия (пустой игровой объект с "WeaponScript").




Отлично. Теперь напишем скрипт для Босса. Назовем это "BossScript".

```
using UnityEngine;

///

/// Общее поведение для врагов
///

public class BossScript : MonoBehaviour
{
  private bool hasSpawn;

  // Параметры компонентов
  private MoveScript moveScript;
  private WeaponScript[] weapons;
  private Animator animator;
  private SpriteRenderer[] renderers;

  // Поведение босса (не совсем AI)
  public float minAttackCooldown = 0.5f;
  public float maxAttackCooldown = 2f;

  private float aiCooldown;
  private bool isAttacking;
  private Vector2 positionTarget;

  void Awake()
  {
    // Получить оружие только один раз
    weapons = GetComponentsInChildren();

    // Отключить скрипты при отсутствии спауна
    moveScript = GetComponent();

    // Получить аниматор
    animator = GetComponent();

    // Получить рендереры в детях
    renderers = GetComponentsInChildren();
  }

  void Start()
  {
    hasSpawn = false;

    // Отключить все
    // -- Collider
    collider2D.enabled = false;
    // -- Движение
    moveScript.enabled = false;
    // -- Стрельба
    foreach (WeaponScript weapon in weapons)
    {
      weapon.enabled = false;
    }

    // Дефолтное поведение
    isAttacking = false;
    aiCooldown = maxAttackCooldown;
  }
```

```
void Update()
{
  // Проверим появился ли враг
  if (hasSpawn == false)
  {
    // Для простоты проверим только первый рендерер
    // Но мы не знаем, если это тело, а глаз или рот ...
    if (renderers[0].IsVisibleFrom(Camera.main))
    {
      Spawn();
    }
  }
  else
  {
    // AI
    //--------------------------------
    // Перемещение или атака.
    aiCooldown -= Time.deltaTime;

    if (aiCooldown <= 0f)
    {
      isAttacking = !isAttacking;
      aiCooldown = Random.Range(minAttackCooldown, maxAttackCoold
      positionTarget = Vector2.zero;

      // Настроить или сбросить анимацию атаки
      animator.SetBool("Attack", isAttacking);
    }

    // Атака
    //----------
    if (isAttacking)
    {
      // Остановить все движения
      moveScript.direction = Vector2.zero;

      foreach (WeaponScript weapon in weapons)
      {
        if (weapon != null && weapon.enabled && weapon.CanAttack)
        {
          weapon.Attack(true);
          SoundEffectsHelper.Instance.MakeEnemyShotSound();
        }
      }
    }
    // Перемещение
    //----------
    else
    {
      // Выбрать цель?
      if (positionTarget == Vector2.zero)
      {
        // Получить точку на экране, преобразовать ее в цель в ми
        Vector2 randomPoint = new Vector2(Random.Range(0f, 1f), R

        positionTarget = Camera.main.ViewportToWorldPoint(randomP
      }
```

```
aiCooldown = Random.Range(minAttackCooldown, maxAttackCooldown);
positionTarget = Vector2.zero;
```

```
domPoint = new Vector2(Random.Range(0f, 1f), Random.Range(0f, 1f));
```

```
positionTarget = Camera.main.ViewportToWorldPoint(randomPoint);
```

```
    else
    {
      // Выбрать цель?
      if (positionTarget == Vector2.zero)
      {
        // Получить точку на экране, преобразовать ее в цель в ми
        Vector2 randomPoint = new Vector2(Random.Range(0f, 1f), R

        positionTarget = Camera.main.ViewportToWorldPoint(randomP
      }

      // У нас есть цель? Если да, найти новую
      if (collider2D.OverlapPoint(positionTarget))
      {
        // Сбросить, выбрать в следующем кадре
        positionTarget = Vector2.zero;
      }

      // Идти к точке
      Vector3 direction = ((Vector3)positionTarget - this.transfo

      // Помните об использовании скрипта движения
      moveScript.direction = Vector3.Normalize(direction);
    }
  }
}
```

```
direction = ((Vector3)positionTarget - this.transform.position);
```

```
private void Spawn()
{
  hasSpawn = true;

  // Включить все
  // -- Коллайдер
  collider2D.enabled = true;
  // -- Движение
  moveScript.enabled = true;
  // -- Стрельба
  foreach (WeaponScript weapon in weapons)
  {
    weapon.enabled = true;
  }

  // Остановить основной скроллинг
  foreach (ScrollingScript scrolling in FindObjectsOfType())
  {
    if (scrolling.isLinkedToCamera)
    {
      scrolling.speed = Vector2.zero;
    }
  }
}

void OnTriggerEnter2D(Collider2D otherCollider2D)
{
  // В случае попадания изменить анимацию
  ShotScript shot = otherCollider2D.gameObject.GetComponent();
  if (shot != null)
  {
    if (shot.isEnemyShot == false)
    {
      // Stop attacks and start moving awya
      aiCooldown = Random.Range(minAttackCooldown, maxAttackCoold
      isAttacking = false;

      // Изменить анимацию
      animator.SetTrigger("Hit");
    }
  }
}

void OnDrawGizmos()
{
  // Небольшой совет: Вы можете отобразить отладочную информацию
  if (hasSpawn && isAttacking == false)
  {
    Gizmos.DrawSphere(positionTarget, 0.25f);
  }
}
}
```

```
Random.Range(minAttackCooldown, maxAttackCooldown);
= false;
```

**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.
Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.
Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.
Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Практическое занятие № 9 Создание меню игры.**

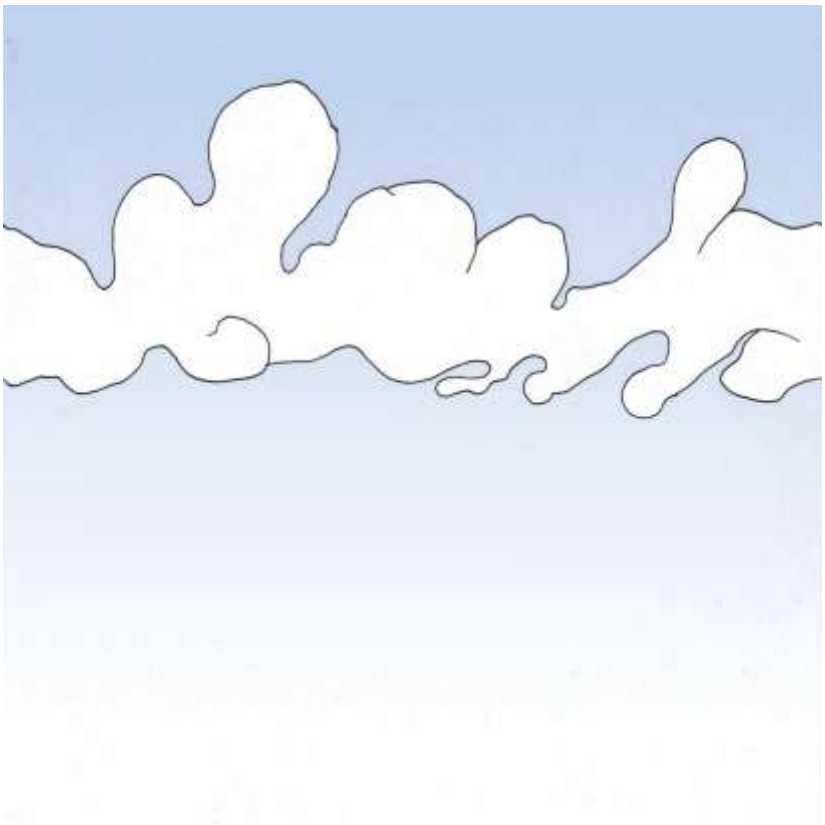
**Выполнив работу, Вы будете:**
***уметь:***
- создавать основное меню игры
- ставить игру на паузу
- создавать меню паузы

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Создать меню
2. Создать меню паузы

**Порядок выполнения работ:**

К сожалению, Unity не располагает инструментами для создания создать красивых меню и, даже используя сторонние библиотеки, приходится повозиться. В этом уроке мы не будем строить сложный графический интерфейс, поэтому воспользуемся встроенными инструментами. Давайте начнем с основ. Сохраните следующие картинки:




Это наши фон и логотип. Импортируйте их в проект. Вы можете поместить их в "Меню" - подпапку "Текстуры". В противном случае "фон" сотрет предыдущий файл игры. Для кнопок мы будем использовать стандартные (уродливые) кнопки **Unity.**

Главное меню

Настало время создать сцену с главным меню. Это то, что показывается юзеру при запуске игры. Для начала, создайте новую сцену:

"File" -> "New scene".

Сохраните ее в папке "Scenes" как "Menu".

Вы можете также нажать cmd+N (OS X) или ctrl+N (Windows).

Наше главное меню будет состоять из:

Фона.

Логотипа.

Скрипта, который будет отображать кнопки.

Для фона:

Создайте новый спрайт.

Установите его в (0, 0, 1).

Задайте размер (2, 2, 1).

Для логотипа:

Создайте новый спрайт.

Установите его в (0, 2, 0)

Задайте размер (0.75, 0.75, 1)

Вот, что у вас должно получиться:


Конечно, вы можете добавить свое имя, инструкции, шутки и анимацию. К созданию меню можно подойти творчески, просто имейте в виду, что люди хотят начать играть как можно быстрее и ваши изыски им по большей части "по барабану."

Теперь мы добавим кнопку "начать игру" с помощью скрипта. Создать скрипт по имени "MenuScript" в папке "Scripts", и приложите его к новому пустому игровому объекту под названием "Scripts":

```
using UnityEngine;

///

/// Скрипт главного меню
///

public class MenuScript : MonoBehaviour
{
  void OnGUI()
  {
    const int buttonWidth = 84;
    const int buttonHeight = 60;

    // Определяем место кнопки на экране:
    // по оси X - в центре, по оси Y - 2/3 от высоты
    Rect buttonRect = new Rect(
          Screen.width / 2 - (buttonWidth / 2),
          (2 * Screen.height / 3) - (buttonHeight / 2),
          buttonWidth,
          buttonHeight
        );

    // Нарисуйте кнопку, чтобы начать игру
    if(GUI.Button(buttonRect,"Start!"))
    {
      // По щелчку по кнопке, загрузите первый уровень.
      // "Stage1" - название первой сцены, которую мы создали.
      // Ее то мы и загрузим.
      Application.LoadLevel("Stage1");
    }
  }
}
```

Мы просто рисуем кнопку, которая будет загружать сцене "Stage1", когда игрок нажимает на нее.

Метод OnGUI применяется в каждом кадре и вставляет весь код, в котором отображен элемент GUI: полоса жизни, меню, интерфейс и т.д. Объект GUI позволяет нам быстро создать компоненты GUI из кода, как кнопка с методом GUI.Button.

Теперь запустите игру и полюбуйтесь нашим замечательным меню:


Жмем и... Катастрофа!

Level 'Stage1' (-1) не может быть загружено, поскольку

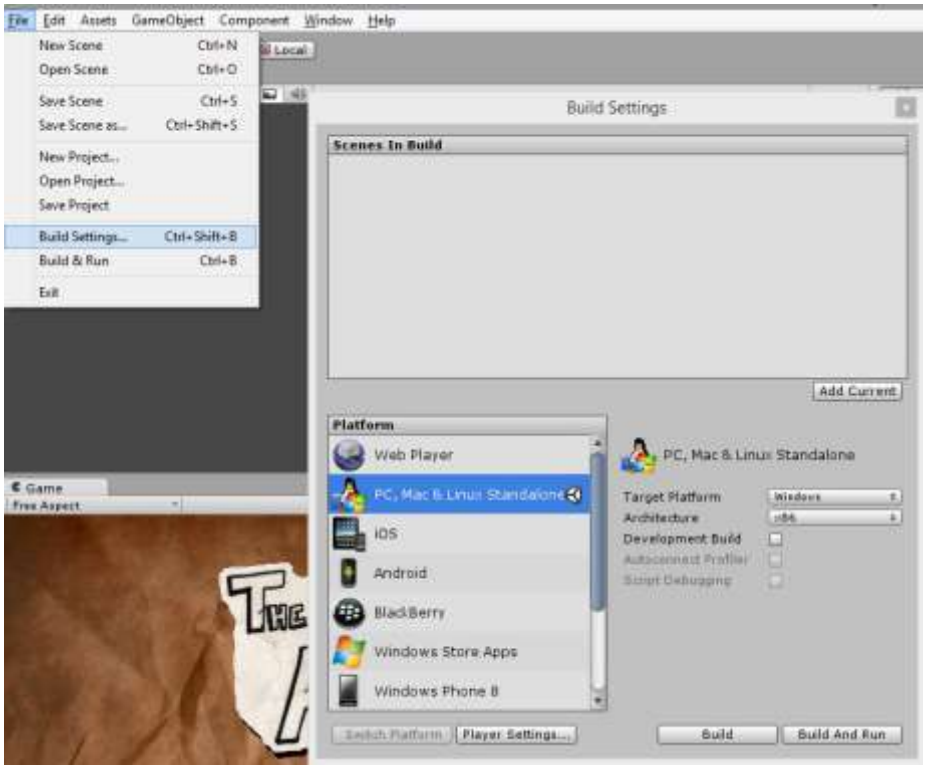
не добавлено в настройки сборки. Чтобы добавить уровень в

настройки сборки, используйте меню File->Build Settings...

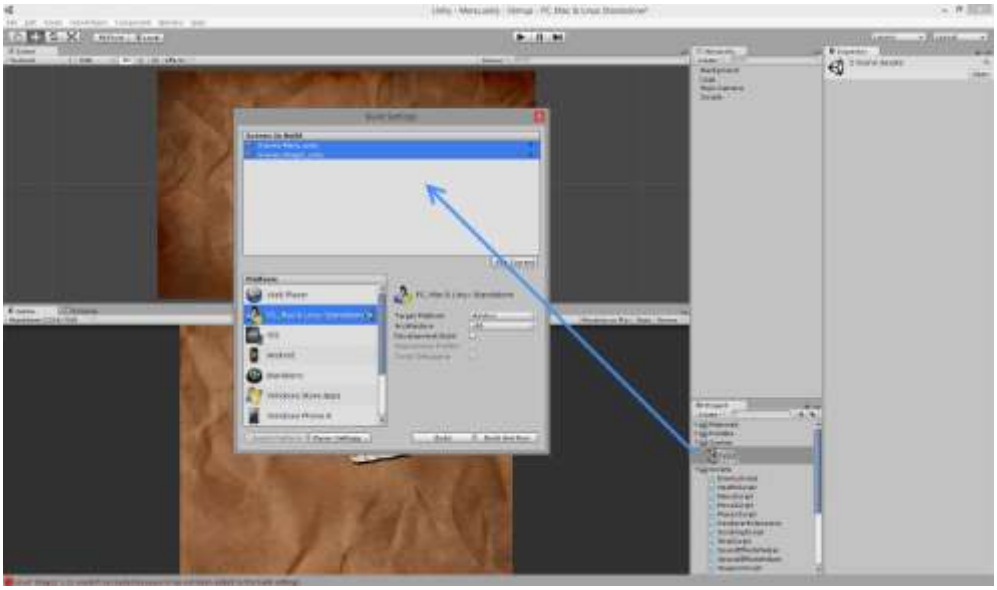
То, что нам нужно сделать четко написано в сообщении об ошибке.

Добавление сцен в сборку

Нажмите в меню Unity на "File", затем на "Build Settings":




Теперь перетащите все сцены, которые должны быть в вашей игре. Здесь все просто: это "Меню" и "Stage1".



Вернитесь в меню. Нажмите на кнопку и ... играть!


Смысл метода Application.LoadLevel() - очистить текущую сцену и инстанцировать все игровые объекты в следующей. Иногда нам нужно, чтобы игровой объект из первой сцены был перенесен во вторую (например, чтобы при переходе между двумя меню непрерывно играла одна и та же музыка).

Для этих случаев в Unity есть метод DontDestroyOnLoad(aGameObject). Просто примените его к игровому объекту – и он не исчезнет при загрузке новой сцены. Он вообще не исчезнет. Поэтому если в следующей сцене вам понадобится его убрать, придется уничтожать его вручную.

Наконец, мы позволим игроку начать игру сначала после того, как его персонаж умер. И, как вы, возможно, заметили, заметили происходит достаточно часто. Давайте "упростим" игру. Вот, что у нас сейчас происходит:

Игрок попадает под пули.

HealthScript.OnCollisionEnter запускается.

Игрок теряет 1 единицу здоровья.

"HealthScript" уничтожае игрока, так как у него меньше, чем 1 единица здоровья.

Мы добавим два новых действия:

Вызывается PlayerScript.OnDestroy.

Создается "GameOverScript" и добавляется к сцене.

Создайте в папке "Scripts" новый скрипт по имени "GameOverScript". Это маленький кусочек кода, который будет отображать кнопки "Начать сначала" и "Назад в меню":

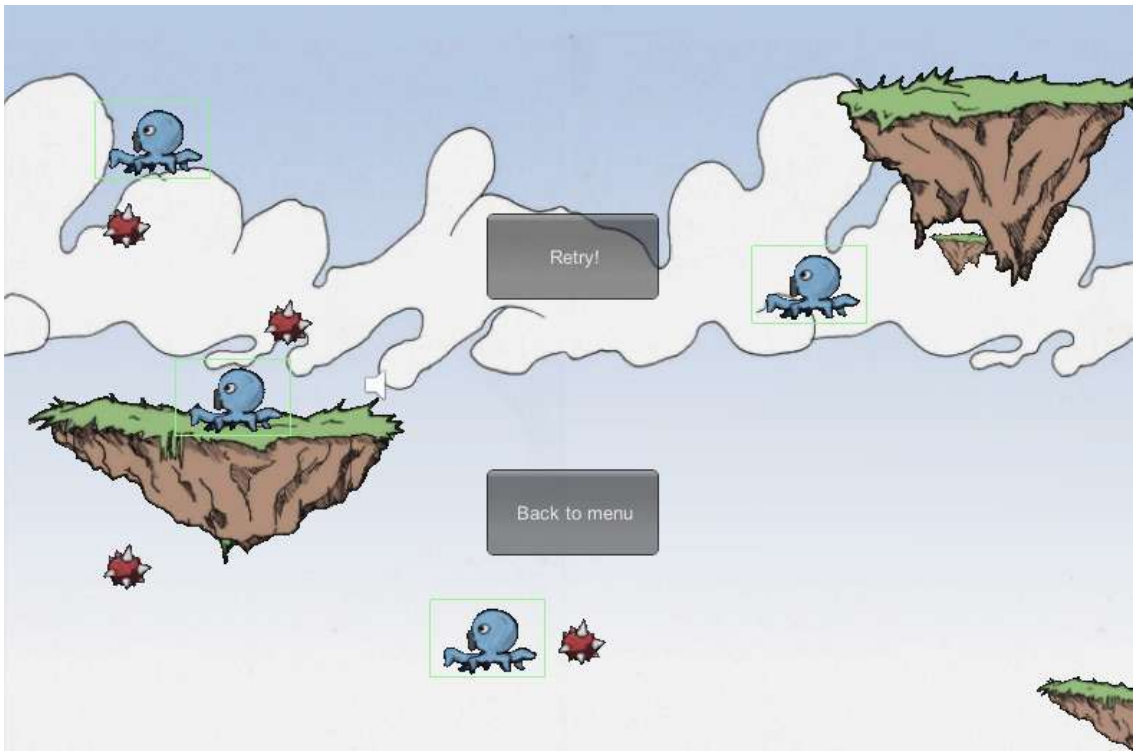
```
using UnityEngine;

// Начало или конец игры
public class GameOverScript : MonoBehaviour
{
  void OnGUI()
  {
    const int buttonWidth = 120;
    const int buttonHeight = 60;

    if (
      GUI.Button(
        // по оси X - по середине, по оси Y - 1/3 от высоты
        new Rect(
          Screen.width / 2 - (buttonWidth / 2),
          (1 * Screen.height / 3) - (buttonHeight / 2),
          buttonWidth,
          buttonHeight
        ),
        "Начать сначала!"
      )
    )
    {
      // загрузить уровень Stage1
      Application.LoadLevel("Stage1");
    }

    if (
      GUI.Button(
        // по оси X - по середине, по оси Y - 2/3 от высоты
        new Rect(
          Screen.width / 2 - (buttonWidth / 2),
          (2 * Screen.height / 3) - (buttonHeight / 2),
          buttonWidth,
          buttonHeight
        ),
        "Назад в меню"
      )
    )
    {
      // загрузить уровень Menu
      Application.LoadLevel("Menu");
    }
  }
}
```

Это абсолютно идентично первому скрипту, который мы написали, с двумя кнопками.

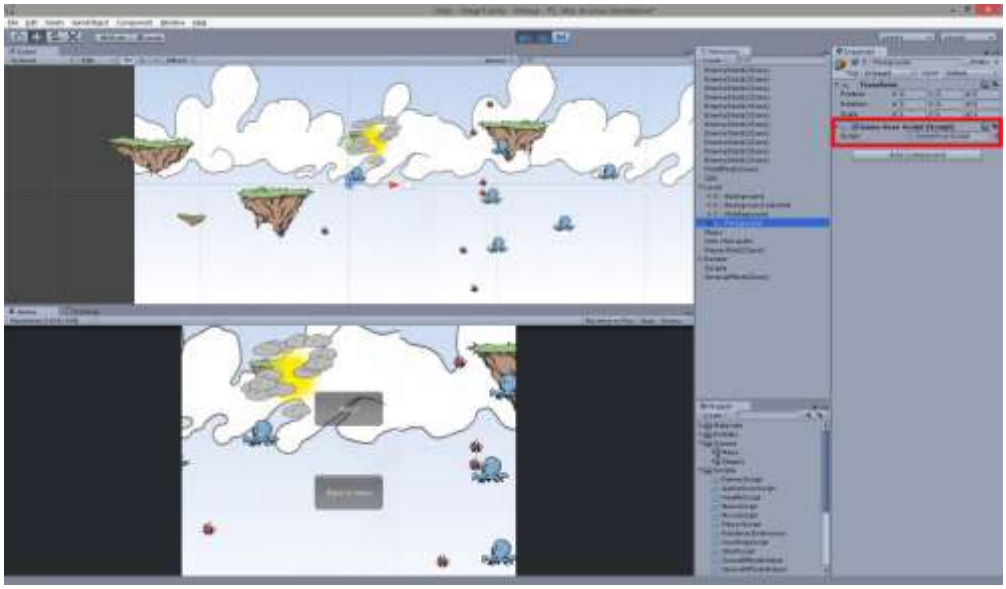
Теперь, в "PlayerScript", нам нужно привязать этот новый скрипт к смерти:

```
void OnDestroy()
{
  // Игра окончена.
  // Добавьте скрипт к родителю, поскольку текущий игровой
  // объект, скорее всего, будет тут же уничтожен.
  transform.parent.gameObject.AddComponent<GameOverScript>();
}
```

Запустите игру и попытайтесь умереть (это не должно занять много времени):



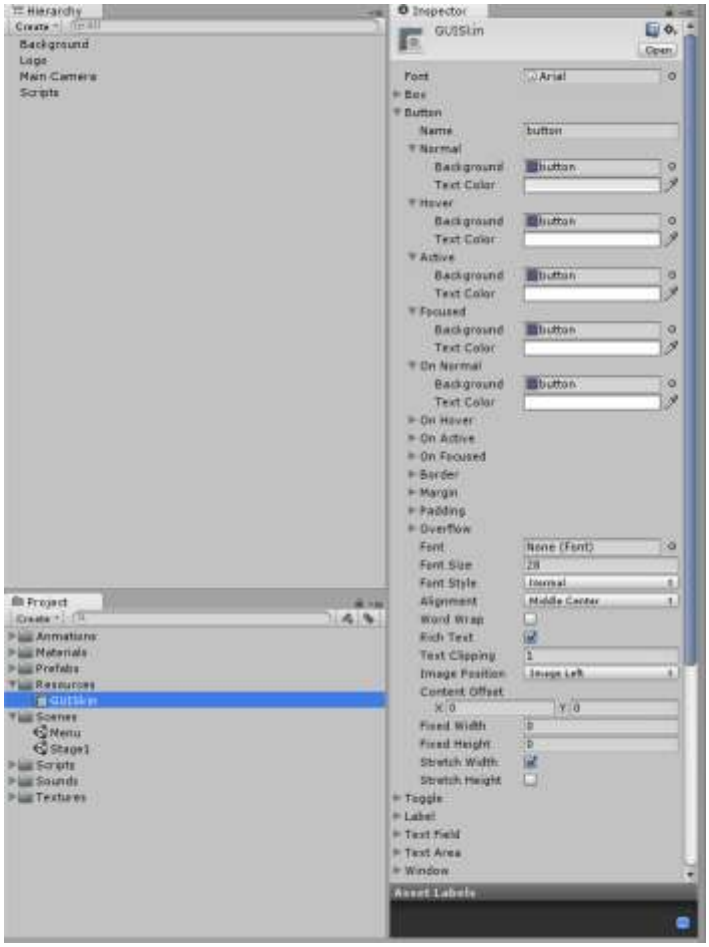

Вы можете найти скрипт где-то на сцене:



Конечно, вы можете улучшить скрипт, например, добавив в него счет или анимациИ.

Если вы хотите провернуть что-нибудь эдакое, можете создать "GUI Skin".

"Assets" -> "Create" -> "Gui Skin":




Вы можете изменить интерфейс в "Inspector", чтобы сделать игру более интересной. Убедитесь в том, что этот скин лежит в папке «Resources» (Ресурсы).

Папка "Resources" занимает в Unity особое место. Чтобы загрузить ее содержимое, используйте метод Resources.Load(). Так вы получите возможность добавлять объекты прямо во время игры, и эти объекты могут быть созданы самими пользователями (как насчет модов?).

Тем не менее, скин не применяется до тех пор, пока вы не вызовите его в вашем скрипте. Во всех наших предыдущих сценариях GUI, мы должны загрузить скин только один раз, а не в каждом кадре, используя GUI.skin = Resources.Load ("GUISkin");.

Вот пример в "MenuScript" (проследите, как действует метод Start()):

```
using UnityEngine;

// Скрипт главного меню
public class MenuScript : MonoBehaviour
{
  private GUISkin skin;

  void Start()
  {
    // Закрузить скин для кнопки
    skin = Resources.Load("GUISkin") as GUISkin;
  }

  void OnGUI()
  {
    const int buttonWidth = 128;
    const int buttonHeight = 60;

    // задать скин
    GUI.skin = skin;

    // Нарисуйте кнопку для начала игры
    if (GUI.Button(
      // Центр в X, 2/3 высоты Y
      new Rect(Screen.width / 2 - (buttonWidth / 2), (2 * Screen.he
      "START"
      ))
    {
      // По щелчку загрузить первый уровень.
      Application.LoadLevel("Stage1"); // "Stage1" - название сцены
    }
  }
}
```

```
creen.height / 3) - (buttonHeight / 2), buttonWidth, buttonHeight),
```

**Форма представления результата:**

Отчет о проделанной работе.

**Критерии оценки:**

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

**Тема 4 Разработка компьютерной игры.**

**Практическое занятие № 10 Выбор платформы.**

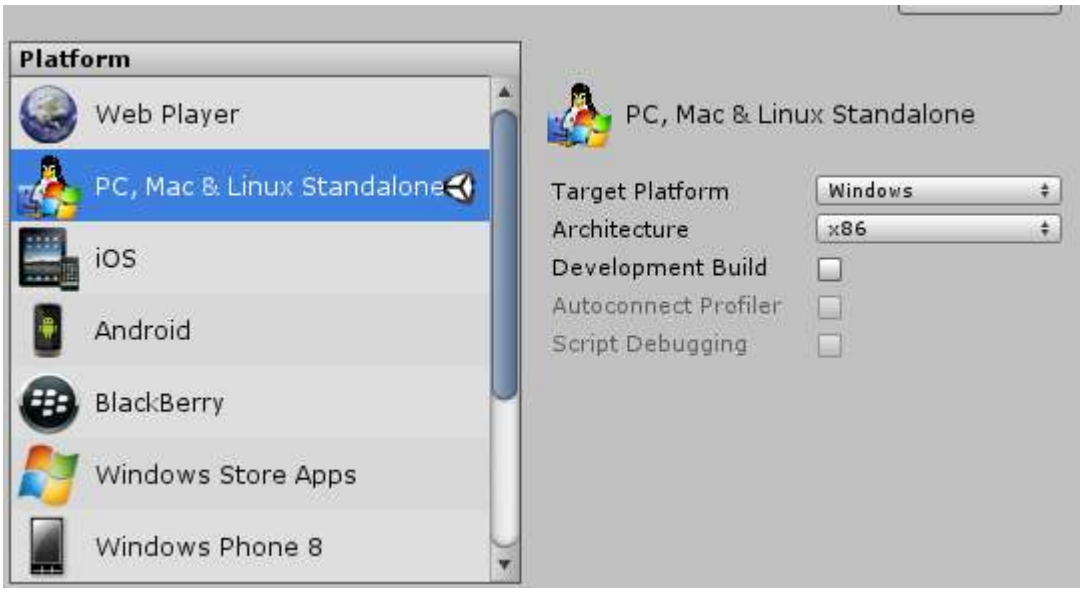
**Выполнив работу, Вы будете:**
***уметь:***
- выбирать платформу для компилирования проекта
- компилировать проект под определенную платформу

**Материальное обеспечение:**
Методические указания для выполнения практических работ

**Задание:**
1. Выбрать несколько платформ для компиляции проекта
2. Скомпилировать проект для нескольких платформ
3. Протестировать проект на этих платформах

**Порядок выполнения работ:**

Когда-то мы уже использовали это окно, теперь пришло время вернуться к нему снова. Откройте окно "File" → "Build Settings". Слева вы можете выбрать платформу, на которой будет работать ваша игра. При этом настройки выбранной платформы появится справа.




Выберите ту, которую вы хотите и нажмите "Build & Run".

Давайте попробуем с Web Player:

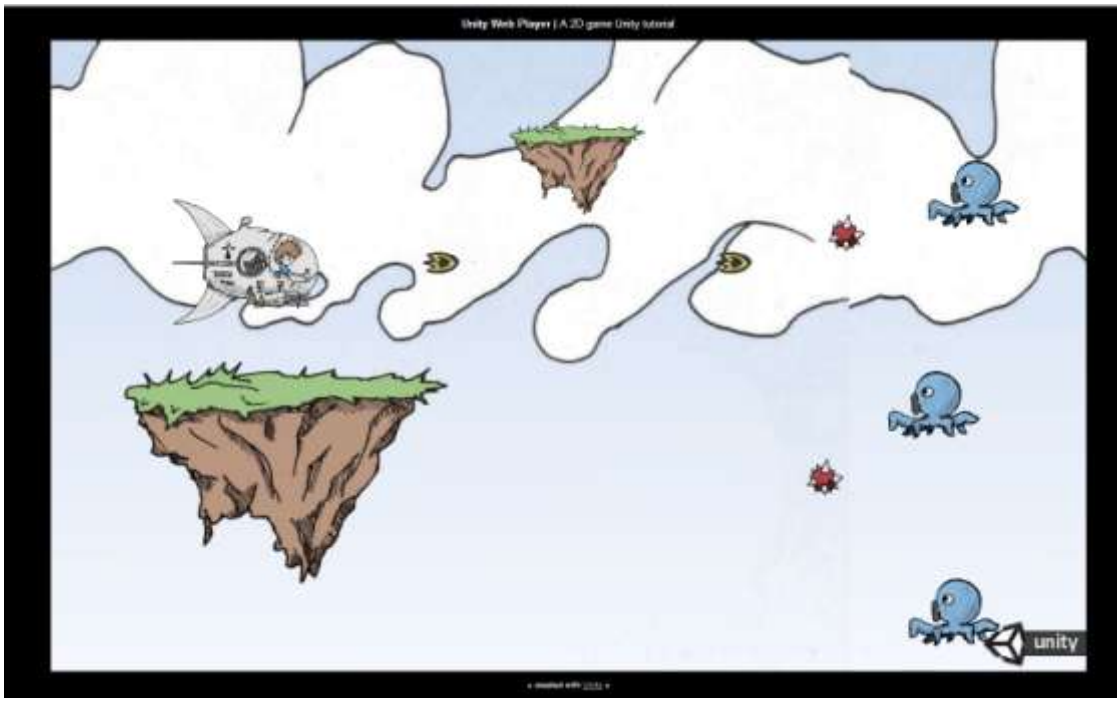
Выберите "Web Player" в "Platform"

Создайте игру.

Обратите внимание: при этом создается страница HTML со встроенной игрой.

Запустите ее.

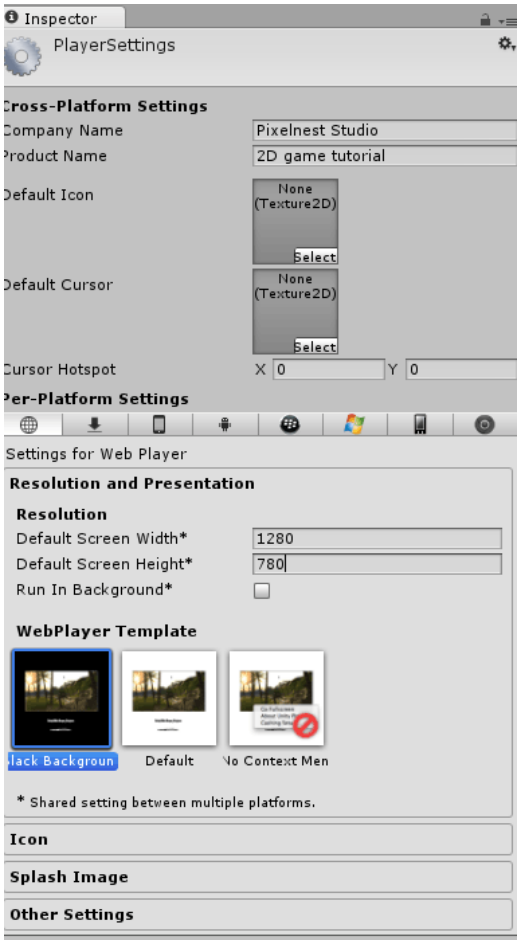
Это первый и самый простой способ распространять свои игры. Поместите эти два файла на сервер и ни о чем не беспокойтесь.




Настроки плеера в Unity

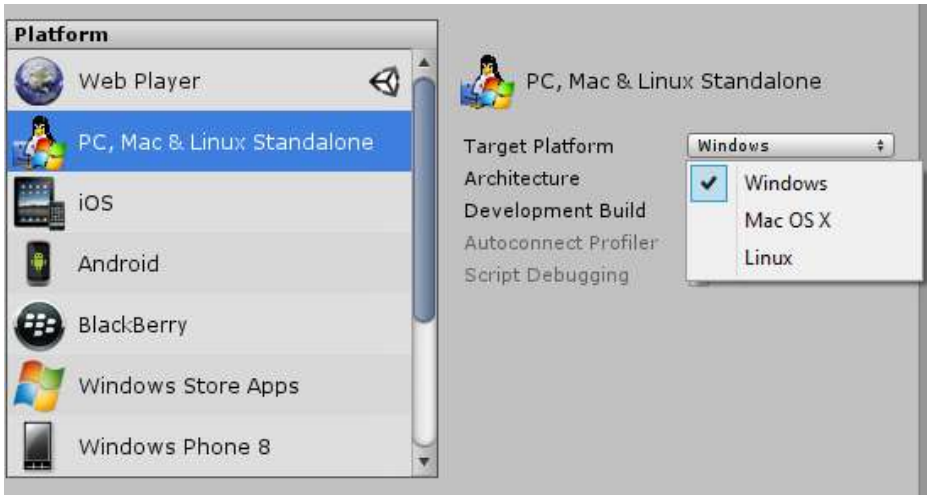
Возможно, вам потребуется изменить некоторые настройки (например, разрешение, название игры или некоторые ресурсы) для конкретной платформы.

Вы можете сделать это через панель "Player Settings": "File" → "Build Settings" → "Player Settings" или "Edit" → "Project Settings" → "Player". Здесь мы устанавливаем разрешение веб-плеера 1280 * 780:




Развертывание на Windows, Mac и Linux

Об этих платформах, в общем-то, нечего сказать. Выбрав "PC, Mac & Linux Standalone", вы сможете уточнить, для какой платформы создаете игру.




И это (почти) все! В Unity действительно легко создавать и развертывать приложения.

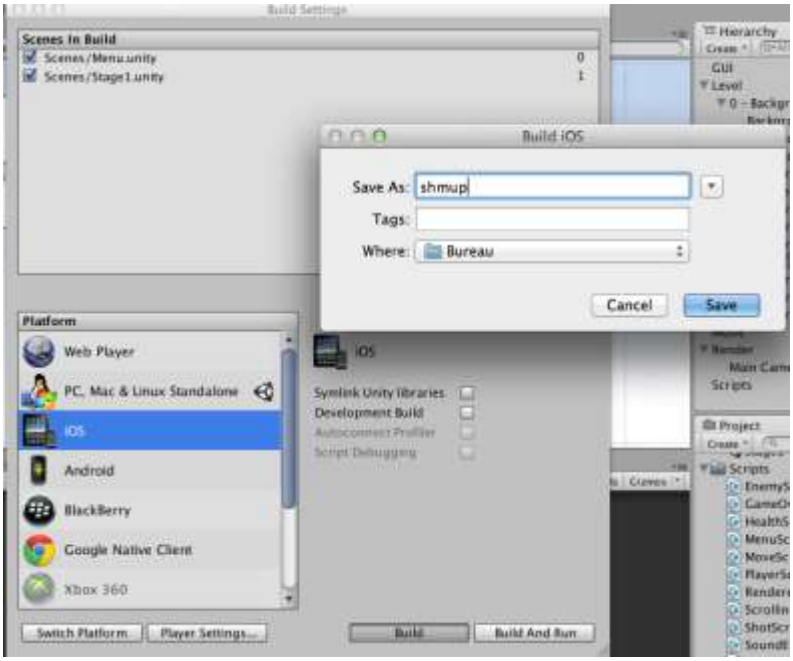
Бонус для пользователей Mac: Развертывание в iOS

Мобильное развертывание немного сложнее. Вы должны иметь последнюю SDK (официальные средства разработки), установленную для данной платформы. Это также означает, что у вас должен быть Mac OS X, чтобы выпустить игру iOS.

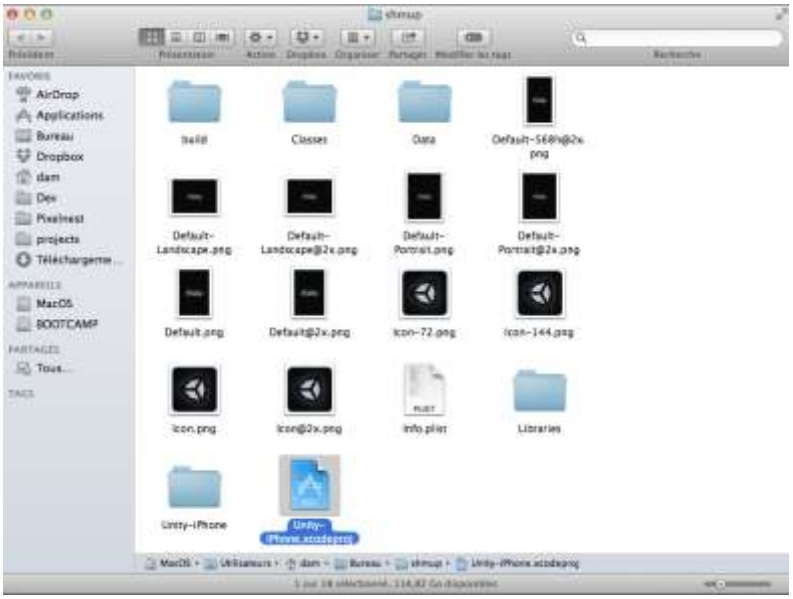
Рассмотрим процесс развертывания игр под iOS (для игр под Android практически все тоже самое). Во-первых, выберите пункт "iOS" в окне сборки:

Откройте вкладку "Настройки плеера" (Player settings), чтобы изменить параметры (минимальный SDK, иконку и т.д.). Для тестирования, вы можете выполнить следующий трюк: в IOS во вкладке "Player settings", найдите поле "SDK version". Затем выберите "Simulator SDK":


Создайте проект. Unity спросит вас, где вы хотите его сохранить:

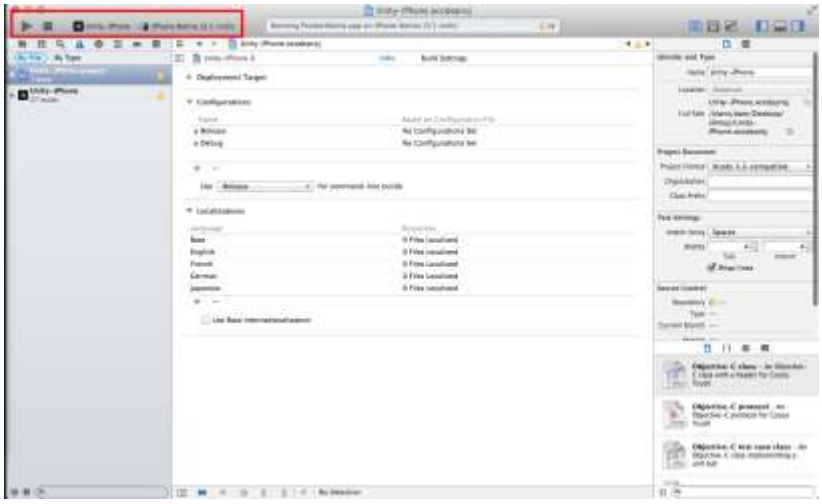



На самом деле, Unity сгенерировала Xcode проект:

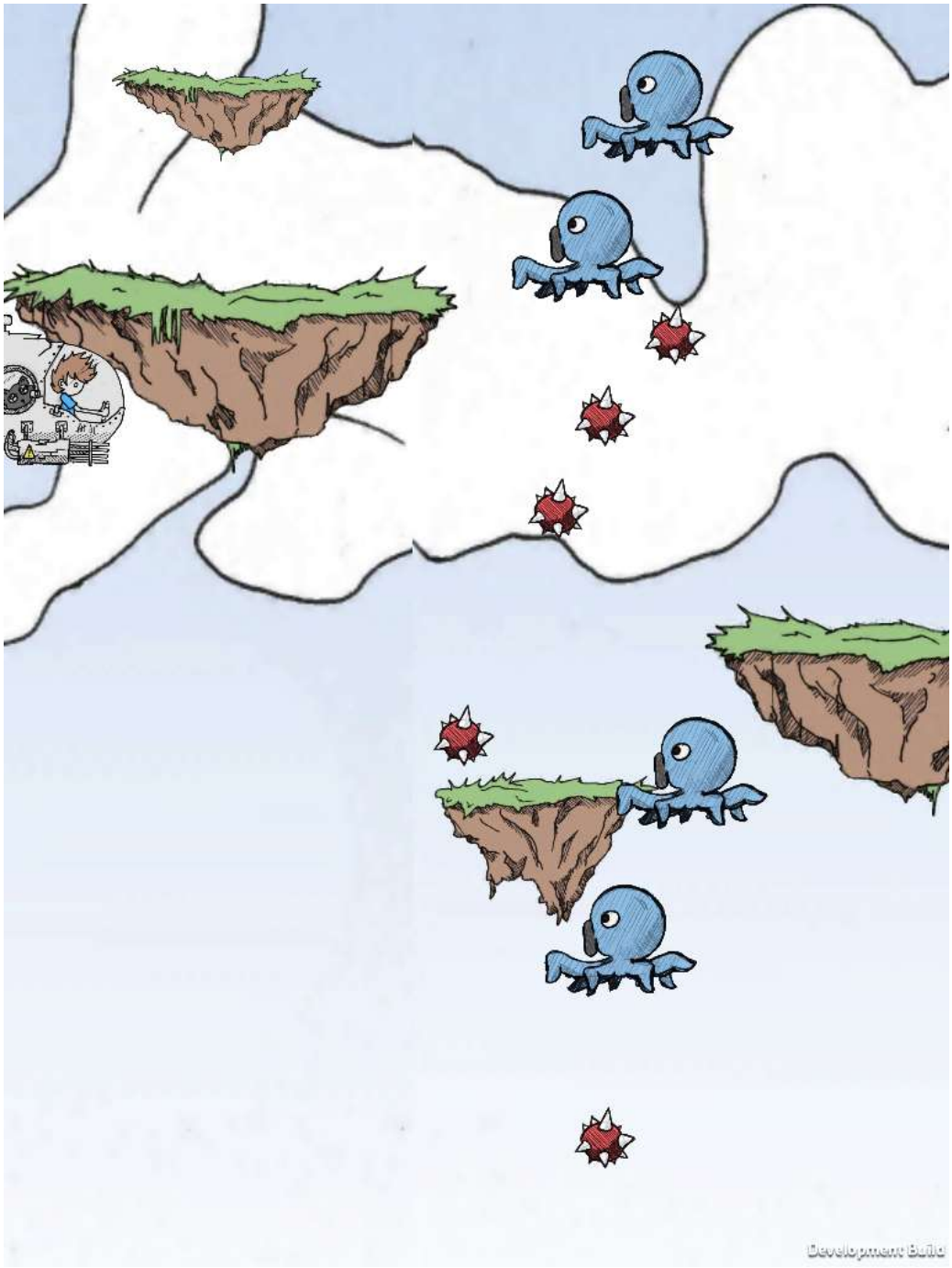



Вот почему вам действительно нужно установить все средства разработки, в противном случае вы не сможете даже запустить проект на устройстве iOS или симуляторе.

Откройте Xcode-файл .xcodeproj. К счастью, больше нам ничего не придется делать, кроме как наконец-то опробовать все на деле:



Попробуйте запустить игру. Она должна пойти на симуляторе. Например, на iPad:



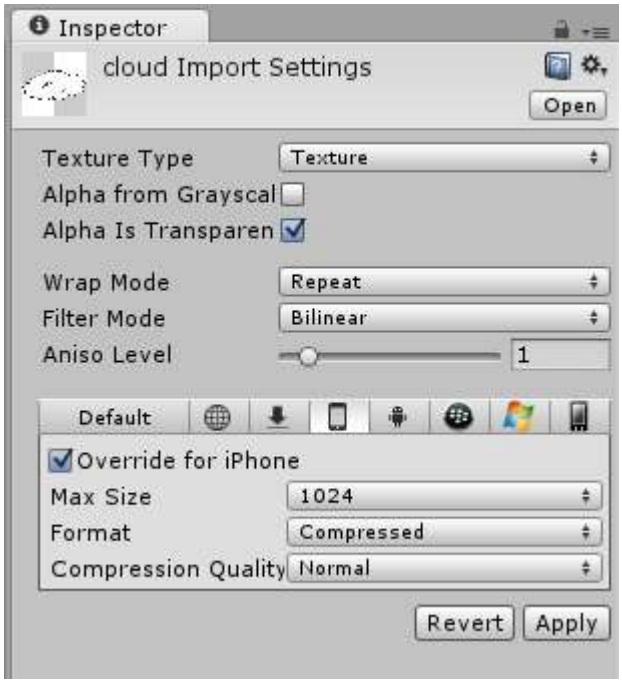
Спрайты корректно отображаются, игра загружается ... Но играть в нее невозможно, потому что мы не подключили тач-управление (кроме тэпа для выстрела по умолчанию). Разрешение и ориентация также не обрабатывается.

И наконец, запустив игру с планшета, вы сможете сами убедиться, какая она корявая.

Вот почему с мобильными играми все не так непросто: нужно оптимизировать и настроить свою игру, чтобы она достойно смотрелась на смартфонах и планшетах.

Качество ассетов для каждой платформы

Для некоторых ассетов, возможно, потребуется увеличить (или уменьшить) качество для выбранной платформы. Посмотрите на изображение, приведенное в качестве примера. Вы можете снизить качество для мобильных устройств, но увеличить его для персональных компьютеров.




**Форма представления результата:**
Отчет о проделанной работе.

**Критерии оценки:**
Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.