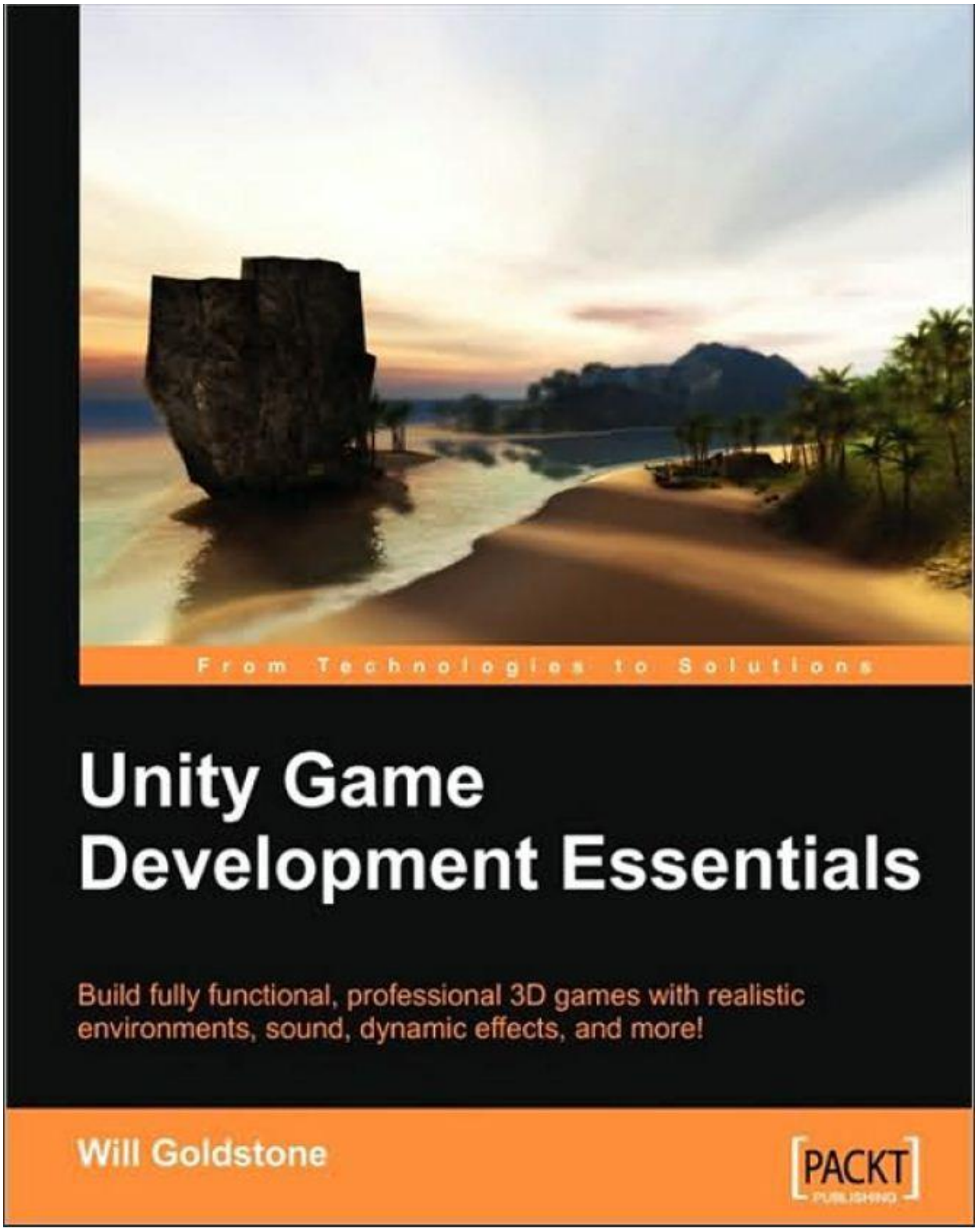
unity


From Technologies to Solutions
Unity Game Development Essentials
Build fully functional, professional 3D games with realistic environments, sound, dynamic effects, and more!
Will Goldstone
PACKT
PUBLISHING

Автор перевода : Sok@I
Автор : Will Goldstone
Источник : Unity Game Development Essentials

1

Добро пожаловать в Третье Измерение

Прежде, чем начать работать с любым трехмерным пакетом, крайне важно понять окружающую среду, в которой Вы будете работать. Поскольку **Unity** - прежде всего трехмерна на основе средство разработки, много понятий всюду по этой книге примут определенный уровень понимания трехмерного развития и двигателей игры. Крайне важно, что Вы оборудуете себя пониманием этих понятий прежде, чем нырнуть в практические элементы остальной части этой книги.

Понимание трехмерного пространства

Давайте смотреть на ключевые элементы трехмерных миров, и как Unity позволяет Вам развивать игры в третьем измерении.

Координаты

Если Вы работали с каким-нибудь трехмерным пакетом прежде, Вы вероятно будете знакомы с понятием **Оси Z.** Ось Z, в дополнение к существующему X для горизонтального и Y для вертикального, представляет глубину. В трехмерных применениях Вы будете видеть информацию относительно объектов, вынутых в X, Y, Z формат - это известно как **Декартовский координационный** метод. Измерения, вращательные ценности, и позиции в трехмерном мире могут все быть описаны таким образом. В этой книге, как в другой документации трехмерных, Вы будете видеть такую информацию, написанную с круглой скобкой, показанной следующим образом:

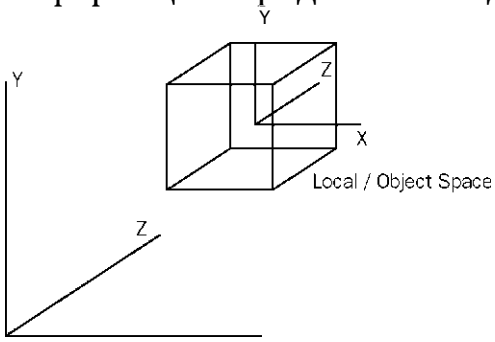
(10, 15, 10)

Это главным образом для опрятности, и также вследствие того, что в программировании, эти ценности должны быть написаны таким образом. Независимо от их представления Вы можете предположить, что любые наборы трех ценностей, отделенных запятыми, будут в X, Y, Z заказ.

Местные координаты вращения против Глобальных

Решающее понятие, чтобы начать смотреть является различием между **Местными** и Глобальны**ми.** В любом трехмерном пакете мир, в котором Вы будете работать, технически бесконечен, и может быть трудно отследить местоположение объектов. В каждом трехмерном мире есть пункт происхождения, часто называемого нолем, поскольку это представлено позицией (0,0,0).

Все глобальные позиции объектов в трехмерном пространстве расположены относительно глобального ноля. Однако, чтобы сделать более простыми вещи, мы также используем Местное координаты (также известный как **Local Rotation)**, чтобы определить объектные позиции относительно друг друга. Местные координаты предполагают, что у каждого объекта есть свой собственный нулевой пункт, который является пунктом, из которого появляются его ручки оси. Это обычно - центр объекта, и создавая отношения между объектами, мы можем сравнить их позиции относительно друг друга. Такие отношения, известные как **родительско-детские отношения**, означают, что мы можем вычислить расстояния от других объектов, используя Местные координаты, с позицией родительского объекта, становящейся новым нулевым пунктом для любого из его детских объектов. Для получения дополнительной информации о родительско-детских отношениях, см. Главу 3.



Векторы

Вы будете также видеть трехмерные векторы, описанные в Декартовских координатах. Как их 2-ые копии, трехмерные векторы - просто линии, оттянутые в трехмерном мире, у которых есть направление и длина. Векторы могут быть перемещены в глобальные координаты, но остаться неизменными непосредственно. Векторы полезны в контексте двигателя игры, поскольку они позволяют нам вычислять расстояния, относительные углы между объектами, и направление объектов.

Камеры

Камеры являются существенными в трехмерном мире, поскольку они действуют как портами вида для экрана. Имея в форме пирамиды поле зрения, камеры могут быть помещены в любом пункте в мире, оживили, или были свойственны характерам или объектам как часть сценария игры. С приспосабливаемым **Полем зрения (FOV)** трехмерные камеры - Ваш порт вида на трехмерном мире. В двигателях игры Вы заметите, что эффекты, такие как освещение, пятна движения, и другие эффекты применены к камере, чтобы помочь с моделированием игры глазного вида человека мира - Вы можете даже добавить несколько кинематографических эффектов, которые человеческий глаз никогда не будет испытывать, такие как вспышки линзы, смотря на солнце!

Большинство современных трехмерных игр использует многократные камеры, чтобы показать части мира игры, что камера характера в настоящее время не видны в подобных кинематографических условиях. Unity делает это с непринужденностью, позволяя много камер в отдельной сцене, которая может быть подготовлена, чтобы действовать как главная камера в любом пункте во время времени выполнения. Многократные камеры могут также использоваться в игре, чтобы управлять предоставлением специфических 2-ух и трехмерных элементов отдельно как часть процесса оптимизации. Например, объекты могут быть сгруппированы в слоях, и камерам можно поручить видеть объекты в специфических слоях.

Многоугольники, края, вершины, и меши

В строительстве трехмерных форм все объекты в конечном счете составлены из связанных 2-ух мерных форм, известных как **многоугольники.** При импортировании моделей от применения моделирования Unity преобразовывает все многоугольники в **треугольники многоугольника.** Треугольники многоугольника в свою очередь составлены из трех связанных **краев.** Местоположения, в которых встречаются эти вершины, известны как **пункты** или **вершины.** Зная эти местоположения, двигатели игры в состоянии сделать вычисления относительно точек падений ракет, известных как **столкновения,** используя сложное обнаружение столкновения с Коллайдерами **Меша**, такой как в стреляющих играх, чтобы обнаружить точное местоположение, в котором пуля поразила другой объект. Комбинируя много связанных многоугольников, трехмерные применения моделирования позволяют нам строить сложные формы, известные как **меши.** В дополнение к построению трехмерных форм у данных, хранивших в петлях, может быть много другого использования. Например, это может использоваться как поверхностные навигационные данные, делая объекты в игре, следующим за вершинами.

В проектах игры для разработчика крайне важно понять важность счета многоугольника. Счет многоугольника - общее количество многоугольников, часто в ссылке на модель, но также и в ссылке на весь уровень игры. Чем выше число многоугольников, тем больше работы Ваш компьютер должно сделать, чтобы отдать объектына экран. Это - то, почему, в прошлое десятилетие, мы видели, что увеличение уровня

деталей от ранних трехмерных игр до таковых сегодня просто сравнивает визуальные детали в игре, такой как *Землетрясение* Ида (1996) с деталями, замеченными в игре, такими как *Механизмы* Эпопеи *войны* (2006). В результате более быстрой технологии разработчики игры которые теперь в состоянии содержать намного более высокий счет многоугольника, и эта тенденция неизбежно продолжится.

Материалы, структуры, и шейдеры

Материалы - общее понятие ко всем трехмерным применениям, поскольку они обеспечивают средства установить визуальное появление трехмерной модели. От основных цветов до рефлексивных основанных на изображении поверхностей материалы обращаются со всем.

Начинаясь с простого цвета и опции использования того или более известный об изображениях как **структуры** - в отдельном материале, материал работает с **шейдером,** который является сценарием, отвечающим за стиль предоставления.

В Unity использование материалов легко. Любые материалы, созданные в Вашем трехмерном пакете моделирования, будут импортированы и обновлены автоматически двигателем и созданы как активы, чтобы использовать позже. Вы можете также создать свои собственные материалы на пустом месте, назначая изображения как файлы структуры, и выбирая шейдериз большой встроенной библиотеки. Вы можете также написать свои собственные шейдеры, или использовать написанные членами сообщества Unity, давая Вам больше свободы для расширения вне включенного набора различных материалов.

Кардинально, создавая структуры для игры в графическом пакете, такие как Фотомагазин, Вы должны знать о решении. Структуры игры, как ожидают, будут квадратными, и измерены к отношению 2 x 2. Это означает, что числа должны работать следующим образом:

- 128 x128
- 256 x 256
- 512 x 512
- 1024 x1024

Создание структур этих размеров будет означать, что они могут успешно умножатся – накладыватся на модель без потери качества, двигателем игры. Вы должны также знать что, чем больший файл структуры который Вы используете, тем больше мошности обработки Вы будете требовать от компьютера игрока. Поэтому, всегда не забывайте пытаться изменить размеры Вашей графики к наименьшей власти 2 возможных измерений, не жертвуя слишком многим в качестве.

Физика Твердого тела

Для разработчиков, работающих с двигателями игры**, двигатели физики** обеспечивают сопровождающий способ моделировать ответы реального мира для объектов в играх. В Unity двигатель игры использует двигатель

Nvidia *PhysX*, популярный и очень точный коммерческий двигатель физики.
В двигателях игры нет никакого предположения, что объект должен быть затронут физикой во-первых, потому что требуется большая обработка данных, и во-вторых потому что это просто не имеет смысла. Например, в трехмерной ведущей игре, имеет смысл для автомобилей быть под влиянием двигателя физики, но не окружающими объектами, такими как деревья, стены, и так далее - они просто не должны быть. Поэтому, делая игры, компонент **Твердого тела** дан любому объекту, который Вы хотите под контролем двигателя физики.

Двигатели физики для игр используют систему динамики Твердого тела создания реалистического движения. Это просто означает, что вместо объектов, являющихся статичным в трехмерном мире, у них могут быть следующие свойства:

- Масса
- Сила тяжести
- Скорость
- Трение

Как власть аппаратных средств и увеличений программного обеспечения, физика твердого тела становится более широко прикладной в играх, поскольку это предлагает потенциал для более различного и реалистического моделирования. Мы будем использовать динамику твердого тела как часть нашей игры в Главе 6.

Обнаружение столкновения

В то время как более решающий для двигателей игры чем в трехмерной мультипликации, обнаружение столкновения - способ, которым мы анализируем наш трехмерный мир для межобъектных столкновений. Давая объект компонент **Коллайдера**, мы эффективно помещаем невидимую сеть вокруг его. Эта сеть подражает своей форме и отвечает за сообщение о любых столкновениях с другими коллайдерами, заставляя двигатель игры ответить соответственно. Например, в игре боулинга с десятью кеглями, простой сферический коллайдер окружит шар, в то время как сами кегли будут иметь или простой краткий коллайдер, или для более реалистического столкновения, использовать коллайдер меша. На воздействии коллайдеры любых затронутых объектов сообщат двигателю физики, который продиктует их реакцию, основанную на направлении воздействия, скорости, и других факторов.

В этом примере, используя коллайдер меша, чтобы соответствовать точно к форме модели булавки было бы более точным, но более дорог в обработке условий. Это просто означает, что это требует больше мошности обработки от компьютера, стоимость которого отражена в более медленной работе следовательно *дорогой* термин.

Существенные понятия Unity

Unity делает процесс производства игры простым, давая Вам ряд логических шагов, чтобы построить любой мыслимый сценарий игры. Известный тем, что был "не типом игры", определенным, Unity предлагает Вам чистый холст и ряд последовательных процедур, чтобы позволить Вашему воображению быть пределом Вашего творческого потенциала. Устанавливая его использование **Объекта Игры (ИДУТ)** понятие, Вы в состоянии сломать части своей игры в легко управляемые объекты, которые сделаны из многих отдельных **Составных частей**. Вы в состоянии бесконечно расширить свою игру в логической прогрессивной манере. У составных частей в свою очередь есть параметры настройки переменной(var) по существу, чтобы управлять ими Регулируя эти переменные, у Вас будет полный контроль над эффектом, который Компонент имеет на Ваш объект. Давайте смотреть на простой пример.

Unity путь

Если бы я желал иметь живой шар как часть игры, то я начал бы со сферы. Это может быстро быть создано через меню Unity, и даст Вам новый Объект Игры с мешем сферы (сеть трехмерной формы), и компонент **Renderer**, чтобы сделать это видимым. Создав это, я могу тогда добавить Твердое тело. Rigidbody (Unity именует фразы наиболее с двумя словами как термин отдельного слова) является компонентом, который говорит Unity применять свой двигатель физики к объекту. С этим прибывает масса, сила тяжести, и способность применить силы к объекту, или когда игрок командует этим или просто когда это сталкивается с другим объектом. Наша сфера теперь упадет к основанию, когда игра будет работать, но как мы заставляем это подпрыгнуть? Это просто! У компонента коллайдера есть переменная под названием **Физический Материал** - это - урегулирование для Rigidbody, определяя, как это будет реагировать на поверхности других объектов. Здесь мы можем выбрать **Бодрый,** доступное, заданное, и вот! Наш живой шар готов, только в нескольких щелчках.

Этот подход для самой основной из задач, таких как предыдущий пример, кажется лёгким сначала. Однако, Вы скоро найдете, что, применяя этот подход к более сложным задачам, они становятся очень простыми в достижении. Вот еще краткий обзор тех ключевых понятий Unity:

Активы

Они - стандартные блоки всех проектов Unity. От графики в форме файлов изображения, через трехмерные модели и звуковые файлы, Unity обращается к файлам, которые Вы будете использовать, чтобы создать Вашу игру как активы. Это - то, почему в любой папке проекта Unity все используемые файлы хранятся в детской папке под названием Активы.

Эта книга состоит из кодовых файлов и активов, загруженных на нашем вебсайте (www.packtpub.com/files/code/8181_Code.zip <http://www.packtpub.com/files/code/8181_Code.zip>) и доступный для извлечения здесь. Пожалуйста извлеките файлы из уже упомянутой

ссылке, чтобы использовать в своих интересах коды актива, неотъемлемую часть развития игры Unity.

Сцены

В Unity Вы должны думать **о сценах** как об отдельных уровнях, или областях содержания игры (таких как меню). Строя Вашу игру со многими сценами, Вы будете в состоянии распределить время загрузки и проверить различные части Вашей игры индивидуально.

Объекты Игры

Когда актив используется в сцене игры, это становится новой Игрой, Упомянутой объектом в условиях особенно Unity в scripting-использовании законтрактованного термина "GameObject". Все GameObjects содержат по крайней мере один компонент для начала, то есть, компонент **Transform (преобразовать)**. Преобразователь просто говорит двигателю Unity позицию, вращение, и масштаб объекта - все описанные в X, Y, Z координата (или в случае масштаба, размерного) заказ. В свою очередь, компонент может тогда быть обращен в scripting, чтобы установить позицию объекта, вращение, или масштаб. От этого начального компонента Вы положитесь на объекты игры с дальнейшими компонентами, добавляющими требуемые функциональные возможности, чтобы построить каждую часть любого сценария игры, который Вы можете вообразить.

Компоненты

Компоненты входят в различные формы. Они могут быть для того, чтобы создать поведение, определяя появление, и влияя на другие аспекты функций объекта в игре. 'Прилагая' компоненты к объекту, Вы можете немедленно применить новые части двигателя игры к Вашему объекту. Общие компоненты производства игры прибывают как встроенные с Unity, таким как компонент Rigidbody, упомянутый ранее, вниз к более простым элементам, таким как огни, камеры, эмитенты частицы, и больше. Чтобы построить далее интерактивные элементы игры, Вы напишете сценарии, которые рассматриваются как компоненты в Unity.

Сценарии

Будучи рассмотренным Unity, чтобы быть Компонентами, **сценарии** - основная часть производства игры, и заслуживают упоминания как ключевое понятия. В этой книге мы напишем наши сценарии в JavaScript, но Вы должны знать, что Unity предлагает Вам возможность написать в C# и Boo (производная языка Питона) также. Я хотел демонстрировать Unity с JavaScript, поскольку это - функциональный язык программирования, с простым слежением за синтаксисом, с которым некоторые из Вас, возможно, уже столкнулись в других программах, таких как развитие Adobe Flash в ActionScript или в использовании JavaScript непосредственно для веб разработки.

Unity не требует, чтобы Вы изучили, как кодирование его собственного двигателя работает или как изменить его, но Вы будете использовать

scripting в почти каждом сценарии игры, который Вы развиваете. Красота использования Unity scripting состоит в том, что любой сценарий, который Вы пишете для своей игры, будет достаточно верным после нескольких примеров, поскольку у Unity есть свое собственный встроенный ряд Classes scripting инструкции поведений для Вас . Для многих новых разработчиков, схватившихся с scripting, может быть перспектива укрощения, и тот, который угрожает отпугнуть новых пользователей Unity, которые просто приучены только проектировать. Я введу в scripting один шаг за один раз, с умом к показу Вас не только важности, но также и власти эффективного scripting для Ваших игр созданных наUnity.

Чтобы написать сценарии, Вы будете использовать автономный редактор сценария Unity. На Mac это - приложение по имени **Unitron,** и на PC, **Uniscite.** Эти отдельные приложения могут быть найдены в прикладной папке Unity на Вашем PC или Mac и будут начаты любое время, Вы редактируете новый сценарий или существующий. Исправление и экономия сценариев в редакторе сценария немедленно обновят сценарий в Unity. Вы можете также назначить своего собственный редактора сценария в предпочтении Unity, если Вы желаете.

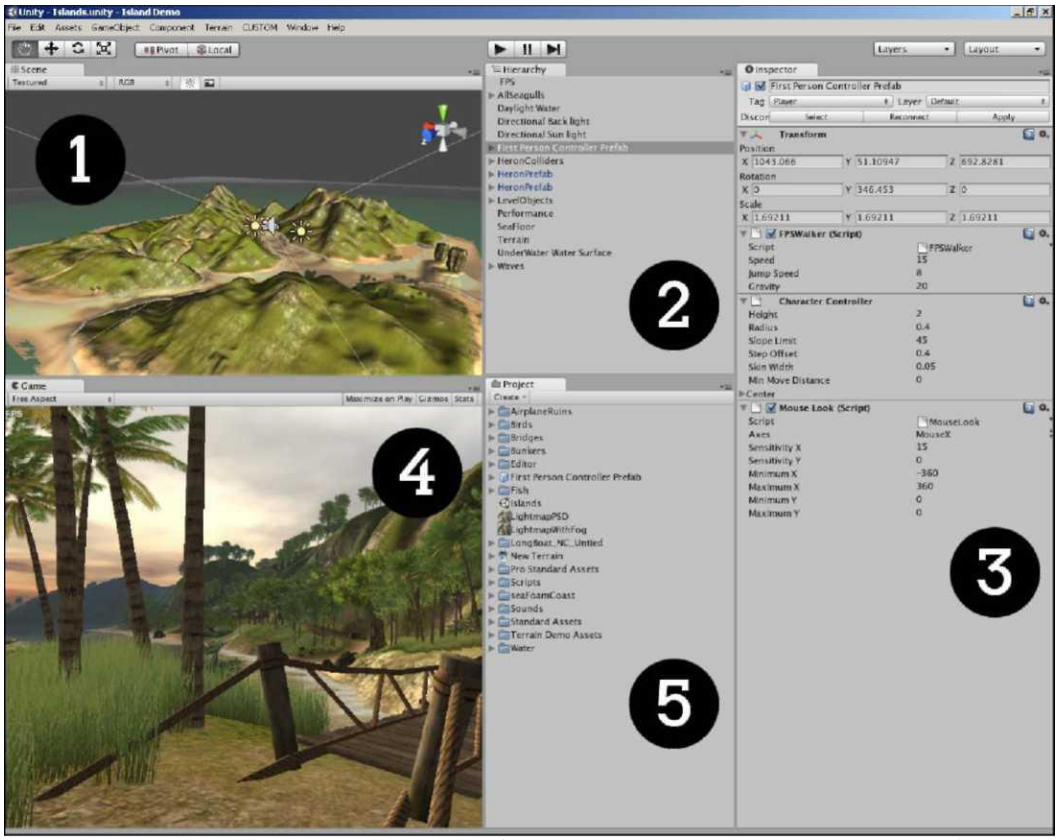
Prefab

Развитие Unity приближается к стержням вокруг понятия GameObject, но у этого также есть умный способ хранить объекты как активы, которые будут снова использованы в различных частях Вашей игры, и затем 'порождены' или 'клонированы' в любое время. Создавая составные объекты с различными компонентами и параметрами настройки, Вы будете эффективно строить шаблон кое для чего, что Вы можете хотеть породить многократные случаи, с каждым случаем, тогда являющимся индивидуально поддающимся изменению. Рассмотрите корзину как пример - Вы, возможно, дали объекту в игре массу, и написали подготовленные поведения для ее разрушения; возможности, Вы будете хотеть использовать этот объект не раз в игре.

Prefab позволяют Вам хранить объект, полный компонентов и текущей конфигурации. Сопоставимый понятию *MovieClip* в Adobe Flash, думайте о prefab просто как о пустых контейнерах, которые Вы можете заполнить объектами, чтобы сформировать шаблон данных, который Вы вероятно переработаете.

Интерфейс

У интерфейса Unity, как много других производственных условий, есть настраиваемое расположение. Состоя из нескольких мест dockable, Вы можете выбрать, какие части интерфейса и где появляются. Давайте смотреть на типичное расположение Unity:



Поскольку предыдущее изображение демонстрирует (показанная версия PC), есть пять различных элементов, с которыми Вы будете иметь дело:

- **Сцена** [1] - где игра построена
- **Иерархия** [2] список GameObjects в сцене
- **Инспектор** [3] - параметры настройки для в настоящее время отбираемого актива/объекта
- **Игра** [4] - окно предварительного просмотра, активное только в способе игры
- **Проект** [5] - список активов Вашего проекта, действия как библиотека

Окно Scene и Иерархия

Окно **Scene** - то, где Вы построите полноту своего проекта игры в Unity. Это окно предлагает перспективу (полный трехмерный) вид, который переключаем к орфографическому (вершина вниз, сторона на, и фронт на) виды. Это действует как полностью предоставленный вид 'Редактора' мира игры, который Вы строите. Перемещение актива к этому окну сделает это активным объектом игры. Вид **Сцены** привязан к **Иерархии,** которая перечисляет все активные объекты в в настоящее время открытой сцене в возрастании на алфавитный порядок.

Окно **Scene** также сопровождается четырьмя полезными кнопками контроля, как показано в предыдущем изображении. Доступный от клавиатуры, используя ключи *Q, W, E,* и *R,* эти ключи выполняют следующие операции:

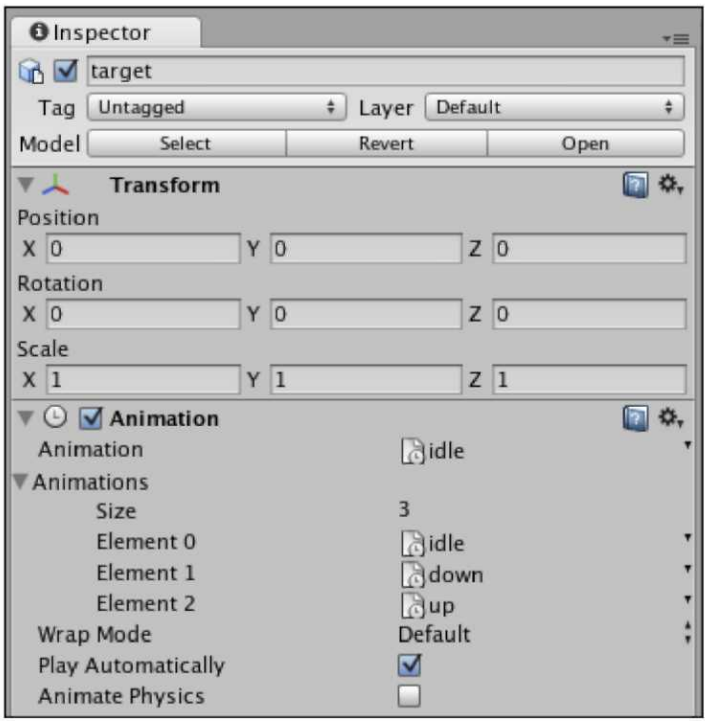
- **Ручной инструмент** [Q]: Это оснащает, позволяет навигацию окна **Scene**. Отдельно, это позволяет Вам тянуться вокруг в окне **Scene**, чтобы подвергнуть резкой критике Ваш взгляд. Отобранный инструмент позволит Вам вращать свой взгляд, с нажатием *командной клавиши CTRL* позволит Вам изменять масштаб изображения. Нажатие *клавиши SHIFT* также ускорит оба из этих функций.
- **Перевести инструмент** *[W]:* Это - Ваш активный инструмент выбора. Поскольку Вы можете полностью взаимодействовать с окном **Scene**, выбирая объекты, или в средствах **Иерархии** или **Сцены** Вы будете в состоянии тянуть ручки оси объекта, чтобы повторно поместить их.
- **Вращать инструмент** [E]: работает таким же образом, как нажатие, используя визуальные 'ручки', чтобы позволить Вам вращать свой объект вокруг каждой оси.
- **Инструмент Масштаба** *[R]:* регулирует размер или масштаб объекта, используя визуальные ручки.

Выбрав объекты или в **Сцене** или в **Иерархии,** они немедленно были отобраны в обоих. Выбор объектов таким образом также покажет свойства объекта в **Инспекторе.** Учитывая, что Вы не можете быть в состоянии видеть объект, который Вы выбрали в **Иерархии** в окне **Scene**, Unity также обеспечивает использование ключа *F*, чтобы сосредоточить Ваш вид **Сцены** относительно того объекта. Просто выберите объект из **Иерархии,** парение Ваш курсор мыши по окну **Scene**, и нажмите *F.*

Инспектор

Думайте об **Инспекторе** как о своем личном наборе инструментов, чтобы приспособить каждый элемент любого объекта игры или актива в Вашем проекте. Понятие, используемое Adobe и Dreamweaver, это - контекстно-зависимое окно. Все это означает, то, что независимо от того, что Вы выбираете, **Инспектор** изменит на показ его соответствующие свойства - это чувствительно к контексту, в котором Вы работаете **Инспектор** покажет каждую составную часть чего-нибудь, что Вы выбираете, и позволяете Вам регулировать переменные этих компонентов, используя простые элементы формы, такие как текстовые окна ввода, весы ползунка, кнопки, и опускаться меню. Многие из этих переменных привязаны в Unity, drag and drop (перетащили и опустили) систему, что означает, что вместо того, чтобы выбрать из опускаться меню, если это более удобно, Вы можете использовать drag and drop (перетащили и опустили), чтобы выбрать параметры настройки.

Это окно используется не только для того, чтобы осмотреть объекты. Это также изменяет показ различных вариантов для Вашего проекта, выбирая их из меню **Edit**, поскольку это действует как идеальное место, чтобы показать Вам изменение предпочтения назад к показу составляющих свойств, как только Вы повторно выбираете объект или актив.



В этом скриншоте **Инспектор** показывает свойства для целевого объекта в игре. Сам объект показывает два компонента - **Преобразовывают** и **Мультипликация. Инспектор** позволит Вам производить изменения в параметрах настройки в любом из них. Также заметьте, чтобы временно повредить любой компонент в любое время - который станет очень полезным для тестирования и экспериментирования - Вы можете просто отсеять количество налево от имени компонента. Аналогично, если Вы желаете выключить весь объект за один раз, тогда Вы можете отсеять количество рядом с ее именем наверху окна **Inspector**.

Окно Project

Окно **Project** - прямой вид папки Активов Вашего проекта. Каждый проект Unity составлен из родительской папки, содержа три Актива подпапок, Библиотеку, и в то время как Редактор Unity работает, папка Временного секретаря. Размещение активов в папку **Активов** означает, что Вы немедленно будете в состоянии видеть их в окне **Project**, и они будут также автоматически импортированы в Ваш проект Unity. Аналогично, изменение любого актива, расположенного в папке Активов, и переэкономия этого от имеющего отношение к третьей стороне применения, такого как Фотомагазин, заставят Unity повторно импортировать актив, отражая Ваши изменения немедленно в Вашем проекте и любых активных сценах, которые используют тот специфический актив.

> Важно помнить, что Вы должны только изменить местоположения актива и названия, используя окно **Project** - использующий Искатель (Mac) или Windows Explorer (PC), чтобы сделать так может сломать связи в Вашем проекте Unity. Поэтому, чтобы переместить или переименовать объекты в Вашей папке Активов, используйте окно **Project** Unity вместо этого.

Окно **Project** сопровождается кнопкой **Create**. Это позволяет создание любых активов, которые могут быть сделаны в пределах Unity, например, сценариев, prefab, и материалов.

Окно Game

Окно **Game** призвано, нажимая кнопку **Play** и действует как реалистический тест Вашей игры. У этого также есть параметры настройки для отношения экрана, которое пригодится, проверяя, сколько из взгляда игрока будет ограничено в определенных отношениях, такой как 4:3 (в противоположность широкому) решения экрана. Нажав **Игру,** крайне важно, что Вы принимаете во внимание следующий совет:

> В способе игры регуляторы Вы делаете к любым частям Вашей сцены игры, являются просто временными - это предназначается как способ тестирования только, и когда Вы нажмете **Игру** снова, чтобы остановить игру, все изменения, произведенные во время способа игры, будут уничтожены. Это может часто сбивать с толку новых пользователей, так что не забывайте об этом!

Окно **Game** может также собираться **Максимизировать**, когда Вы призываете способ игры, давая Вам лучший вид игры в почти fullscreen-, окно расширяется, чтобы заполнить интерфейс. Это стоит отмечать, что Вы можете расширить любую часть интерфейса таким образом, просто толпясь по части, которую Вы желаете расширить и нажим *Клавиши "пробел".*

Резюме

Здесь мы смотрели на ключевые понятия, Вы должны будете понять и закончить упражнения в этой книге. Должен сделать интервалы между ограничениями, я не могу покрыть все подробно, поскольку трехмерное развитие - обширная область исследования. С этим в памяти, я настоятельно рекомендую Вам продолжить читать больше на темах, обсужденных в этой главе, чтобы добавлять Ваше исследование трехмерного развития. У каждой отдельной части программного обеспечения, с которым Вы сталкиваетесь, будут свои собственные специализированные обучающие программы и ресурсы посвященными изучению этого. Если Вы желаете изучить трехмерные художественные

работы, чтобы служить дополнением Вашей работе в Unity, я рекомендую, чтобы Вы ознакомили себя со своим выбранным пакетом, после исследования списка инструментов, которые работают с трубопроводом Unity (см. список в Главе 2), и выбор, какой удовлетворяет Вам лучше всего.

Теперь, когда мы взяли беглый взгляд при трехмерных понятиях и процессах, используемых Unity, чтобы создать игры, мы начнем использовать программное обеспечение, создавая окружающую среду для нашей игры.

В следующей главе мы схватимся с редактором ландшафта. С физическим подходом живописи высоты редактор ландшафта - удобная отправная точка для любой игры с наружной окружающей средой. Мы будем использовать это, чтобы построить остров, и в следующих главах мы добавим особенности к острову, чтобы создать миниигру, в которой пользователь должен осветить походный костер, восстанавливая спички от запертой заставы. Давайте начнем!

2 Окружающая среда

Строя Ваш трехмерный мир, Вы будете использовать два различных типа зданий окружающей среды и пейзажа, построенного в имеющем отношение к третьей стороне трехмерном применении моделирования, и ландшафты создают использованием **редактора ландшафтов** Unity**.**

В этой главе мы будем смотреть на использование обоих, давая краткий обзор необходимых параметров настройки импорта для внешне созданных моделей, но сосредотачиваясь главным образом на использовании собственных инструментов Unity для того, чтобы создать ландшафты. Мы будем определенно смотреть:

- Создание Вашего первого проекта Unity
- Создание и формирование ландшафтов
- Используя комплект инструментов ландшафта, чтобы построить остров
- Освещение сцен
- Используя звук
- Импортирование Упакованных Активов
- Представление Внешних трехмерных Моделей

Внешний modellers

Учитывая, что трехмерный проект - интенсивная дисциплина сам по себе, я рекомендую, чтобы Вы вложили капитал в подобного учебного гида для Вашего применения выбора. Если Вы плохо знакомы с трехмерным моделированием, то вот список трехмерных пакетов моделирования, в настоящее время поддержанных Unity:

- Maya
- 3D Studio Max
- Cheetah 3D
- Cinema 4D
- Blender
- Carara
- Lightwave
- XSI

Они - восемь наиболее подходящих применений моделирования как рекомендующийся Технологиями Unity. Главная причина для этого состоит в том, что они экспортируют модели в формате, который может быть автоматически прочитан и импортирован Unity, когда-то перемещённым в папку Активов Вашего проекта. Эти восемь прикладных форматов будут нести свои меши, структуры, мультипликации, и кости (форма скелетного оснащения) через к Unity, тогда как некоторые меньшие пакеты, возможно, не поддерживают мультипликацию, используя кости на импорт в Unity. Для полного вида последней диаграммы совместимости, посещения: <http://unity3d.com/unity/features/asset-importing>.

Ресурсы

Модели в этой книге будут обеспечены онлайн в формате.fbx (родной формат для использования Unity, которое распространено у большинства трехмерных применений моделирования).

Загружая содержание, чтобы использовать как часть упражнений в этой книге, Вы должны будете использовать систему **пакета** Unity. Доступный от меню вершины **Активов**, импортируя и экспортируя пакеты Unity дает Вам способность передать активы между проектами в то время как включая **зависимости.** Зависимость - просто другой актив, связанный с той, которая Вы - импортирование/экспорт. Например, экспортируя трехмерную модель как часть пакета Unity - переходя сотруднику, или просто между Вашими собственными проектами Unity - Вы должны были бы передать соответствующие материалы и структуры, связанные с моделями, и эти связанные активы будут упоминаться как зависимости модели.

Активы, находящиеся в пакете Unity, форматируют и добавляют их к

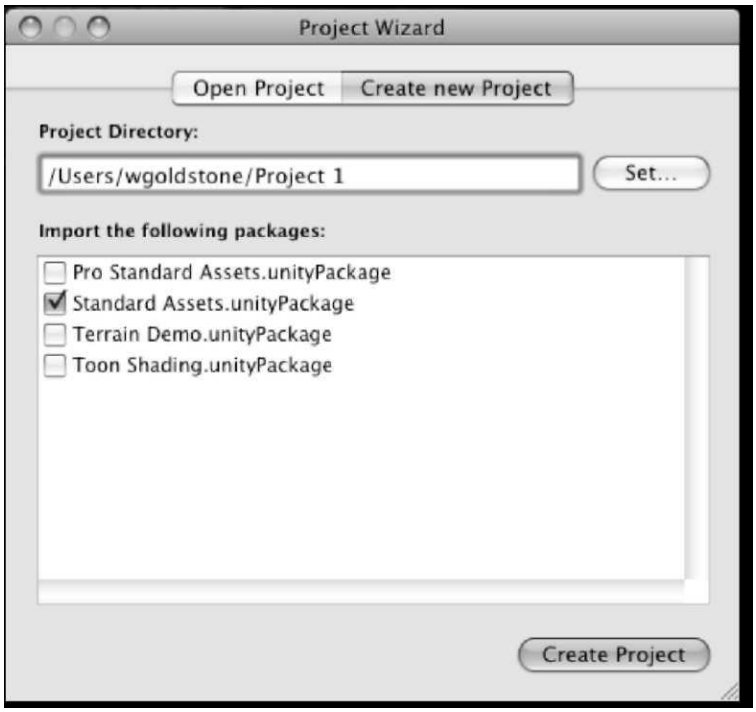
Вашим активам при использовании **Активов | Пакет Импорта.**

Ваш первый проект Unity
Поскольку Unity входит в две различных формы - Инди и Про лицензия разработчика, мы будем придерживаться использования особенностей, к которым у новичка возникают вопросы, и поэтому вероятного Инди-версия лицензии, будет доступнее.

Установив Unity, Ваш первый запуск подарит Вам проект *Демонстрационного примера Острова*. Это - эффективный проект демонстрирующий способности Unity и также помочь новому пользователю выбрать обособленно определенные особенности.
В этой книге Вы будете начинать на пустом месте, и Вы будете нуждаться в новом проекте работы с, так что пойдите в **Файл | Новый Проект.** Это закроет в настоящее время открываемый проект и подарит Вам **Проектного Волшебника,** окно диалога, разрешающее Вам выбрать существующий проект открыться. Вы можете также начать новый, выбирая из нескольких **Пакетов Актива**, чтобы начать с.

> Знайте, что, если в любое время Вы желаете начать Unity и взяты я непосредственно **Проектному Волшебнику,** тогда просто, держите *клавишу ALT* (Мас и я PC), начиная Редактора Unity.

Чтобы начать делать Ваш проект Unity, выберите местоположение, чтобы сохранить Вашу новую проектную папку или определением пути файла в **Проектной Директивной** области или выбирая кнопку **Set** и определяя местоположение в окне диалога, которое появляется. Я называю **Проект** месторождения **1,** но не стесняться называть Вашим как вам нравится. Теперь выберите коробку рядом со **Стандартными Активами.** Это даст Вам ряд свободных активов, обеспеченных Технологиями Unity, чтобы начать с. Когда Вы счастливы тем, где Вы хотите сохранить свою работу, нажать**, Создают Проект.**



Используя редактора ландшафта

В построении любой игры, которая вовлекает наружную окружающую среду, редактор ландшафта - должный - имеют для любого разработчика игры. Unity показал встроенного редактора ландшафта начиная с версии 2.0, и это делает здание полной окружающей средой быстрый и легкий.

В условиях Unity, думайте о ландшафте как про объект игры, у которого есть компонент набора инструментов ландшафта, относился к этому. Начинаясь как **плоскость** - плоская, односторонняя трехмерная форма - ландшафт, который Вы создадите коротко, может быть преобразован в полный комплект реалистической геометрии, с дополнительными деталями, такими как деревья, скалы, листва, и даже атмосферные эффекты, такие как скорость ветра.

Особенности меню ландшафта

Чтобы смотреть на особенности, обрисованные в общих чертах ниже, Вы должны будете создать ландшафт. Столь давайте начнем, вводя новый объект ландшафта игре - это - Актив, который может быть создан в пределах Unity, так просто пойдите в **Ландшафт |, Создают Ландшафт** из главного меню.

Прежде, чем Вы сможете начать изменять свой ландшафт, Вы должны настроить различные параметры настройки для размера и деталей. Меню **Terrain** наверху Unity позволяет, что Вы к не только создаете ландшафт для своей игры, но также и выполняете следующие операции:

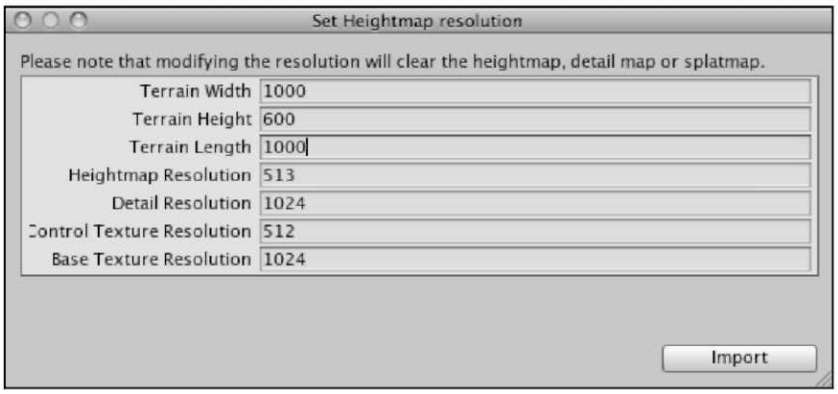
Импортирование и экспорт heightmaps

Карта высот(Heightmaps) - 2-ая графика с легкими и темными областями, чтобы представить топографию ландшафта и могут быть импортированы как альтернатива использованию инструментов живописи высоты Unity.

Созданный в художественном пакете, таком как Фотомагазин и сохранял в.RAW формате, heightmaps часто используются в развитии игры, поскольку они могут быть легко экспортированы и переданы между художественными пакетами и средами проектирования, такими как Unity.

Поскольку мы будем использовать инструменты Ландшафта Unity, чтобы создать нашу окружающую среду, мы не будем использовать внешне созданный heightmaps как часть этой книги.

Набор решение Heightmap



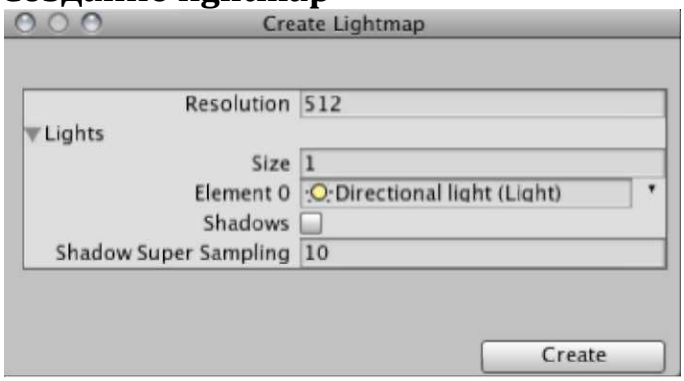
Данное окно диалога позволяет Вам устанавливать много свойств для нового ландшафта, который Вы сделали. Эти параметры настройки должны всегда регулироваться прежде, чем топография ландшафта создана, поскольку наладка их позже может заставить работу над ландшафтом быть перезагруженной.

- **Ширина Ландшафта, Высота** и **Длина:** Измеренный в метрах. Отметьте, что *Высота* здесь устанавливает максимальную высоту, которую может показать топография ландшафта.
- **Решение Heightmap:** решение структуры, которую Unity хранит, чтобы представить топографию в пикселах. Отметьте, что, хотя большинство структур в Unity должно быть властью двух измерений 128, 256, 512 и так далее, heightmap решения всегда, добавляют дополнительный пиксел, потому что каждый пиксел определяет пункт вершины; так в примере 4 x 4 ландшафта, четыре вершины присутствовали бы вдоль каждой секции сетки, но пунктов в который они встреченный включа их конечный точки равные пять.
- **Решение деталей:** решение графики, известный как **карта решения Деталей,** те магазины Unity. Это определяет, как точно Вы можете поместить **детали** относительно ландшафта. Детали - дополнительные ориентиры, такие как заводы, скалы, и кустарники.

Чем больший ценность, тем более точно Вы можете поместить детали относительно ландшафта с точки зрения расположения.

- **Решение Структуры Контроля:** решение структур когда нарисовано на ландшафт. Известный как структуры **Splatmap** в Unity, ценность **Решения Структуры Контроля** управляет размером и, поэтому, детали любых структур, которые Вы подрисовываете. Как со всеми решениями структуры, желательно следить за этой фигурой ниже, чтобы увеличить работу. С этим в памяти, это - хорошая практика, чтобы оставить этот набор значений его неплатежу 512.
- **Основное Решение Структуры:** решение структуры, используемой Unity, чтобы отдать области ландшафта на расстоянии, которые являются дальнейшими от камеры в игре или на старших аппаратных средствах работы.

Создание lightmap



Этот диалог используется, чтобы испечь , освещение топографии на структуры, которые составляют его внешность. Например, если бы мы создали маленький холмистый ландшафт и затем желали включать гору в середине, то тогда мы использовали бы инструменты ландшафта, чтобы создать гору. Однако, учитывая тень набирает ландшафт этой новой горой, мы должны были бы обновить lightmap, чтобы отдать темные области на структуры в недавно заштрихованной области нашего ландшафта.

Использование области **огней** этого диалога позволяет, что Вы, чтобы увеличить число огней имели обыкновение отдавать lightmap. Например, наша сцена может быть освещена главным образом **Направленным светом,** который действует как солнечный свет от определенного направления. Некоторые **Point light** могут быть включены также, такие как те, которые представляют наружные лампы или огни.

Поскольку Вы создаете новую топографию, используя инструменты ландшафта, создавание lightmap Functions является примером одного, Вы, возможно, должны повторно посетить как Ваши пейзажные изменения. Создавая lightmap, чтобы нанести на карту легкие и темные области на ландшафте, Вы также экономите на обработке власти, когда игра работает, поскольку Вы уже вычислили часть освещения. Это против динамического освещения для ландшафта, который более дорог в обработке условий.

Массовые Места Деревьев

Этот функция делает точно, что его имя говорит - размещение

конкретного количества деревьев на ландшафт, с определенным деревом и связанными параметрами, определенными в области **Деревьев Места** компонента сценария ландшафта в Инспекторе.
Этот Functions не рекомендуется для общего использования, поскольку это не дает Вам контроля над позицией деревьев. Я рекомендую, чтобы Вы использовали часть Деревьев Места сценария ландшафта вместо этого, чтобы вручную нарисовать более реалистическое размещение.

Сгладьте Height map

Сгладьтесь Height map присутствует, чтобы позволить Вам сглаживать весь ландшафт в определенной высоте. По умолчанию, Ваша высота ландшафта начинается в ноле, так, если Вы желаете сделать ландшафт с высотой по умолчанию выше этого, такой как, мы делаем для нашего острова, тогда Вы можете определить ценность высоты здесь.

Опытные образцы Деталей и Деревьев

Если Вы произведете изменения в активах, которые составляют любые деревья и детали, которые были уже нарисованы на ландшафт, то Вы должны будете выбрать **Опытные образцы Дерева и Деталей**, чтобы обновить их на ландшафте.

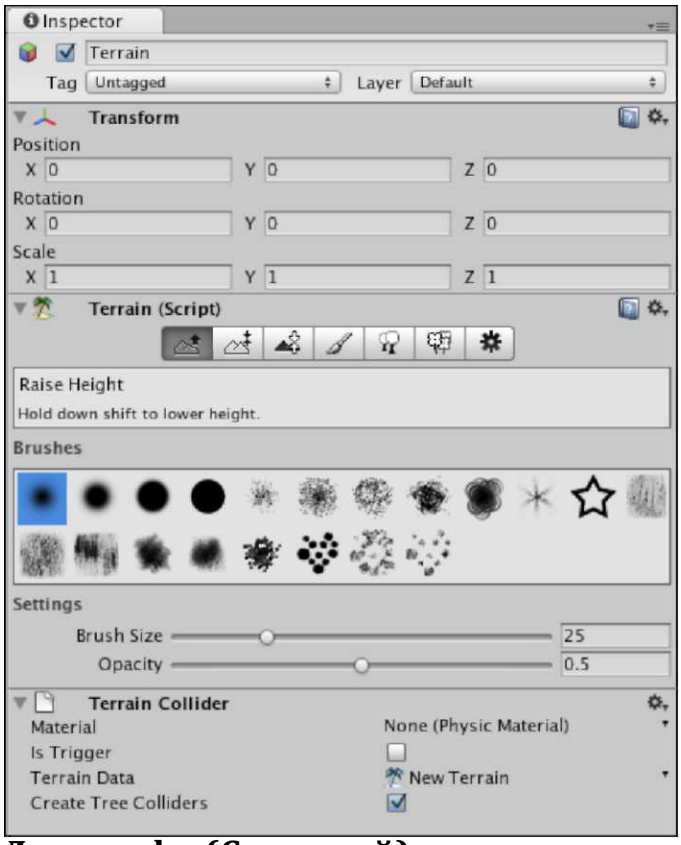
Комплект инструментов ландшафта

Прежде, чем мы начнем использовать их, чтобы построить наш остров, давайте смотреть на инструменты ландшафта так, чтобы Вы могли ознакомить себя с их функциями.

Поскольку Вы только что создали свой ландшафт, это должно быть отображено в окне **Hierarchy**. Если это не отобрано, то выберите это теперь, чтобы показать его свойства в Инспекторе.

Сценарий Ландшафта

На **Инспекторе** комплект инструментов ландшафта упомянут в составляющих условиях как **Ландшафт (Сценарий). Ландшафт (Сценарий)** компонент дает Вам способность использовать различные инструменты и определить параметры настройки для ландшафта в дополнение к functions, доступным в меню ландшафта, обрисованном в общих чертах выше.
В следующем скриншоте Вы можете видеть **сценарий ландшафта.**

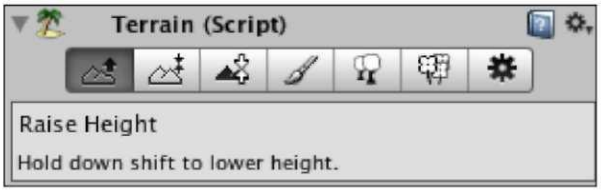


Ландшафт (Сценарий), у компонента есть семь секций к этому, которые легкодоступны от кнопок изображения наверху компонента. Прежде, чем мы начнем, вот быстрый краткий обзор их цели в построении ландшафтов.

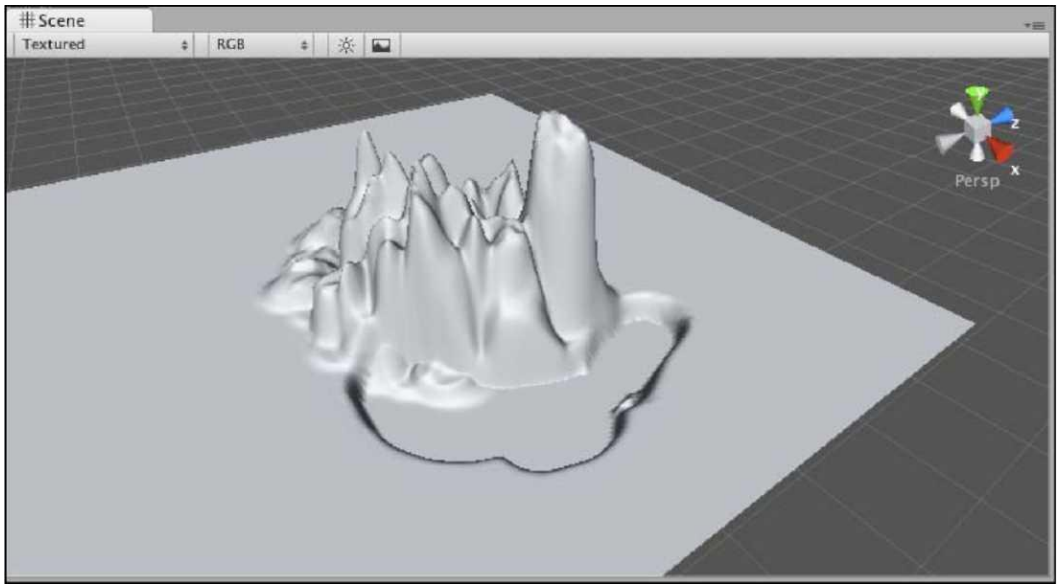
Поднимите Высоту

Этот инструмент позволяет Вам поднимать области, края **инструментом Transform (преобразовать)**

(Сокращение - W ключ).

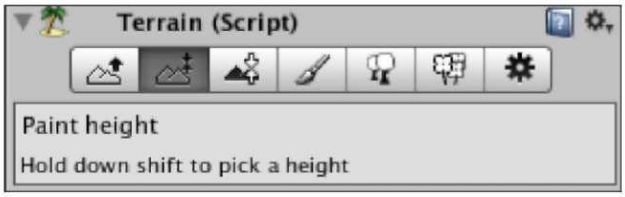


У Вас также есть способность определить, что **Щетка** разрабатывает, **Размер,** и **Opacity** (эффективность) для деформации, которую Вы делаете. Проведение *клавиши SHIFT*, используя этот инструмент вызывает понижающую противоположный эффект высоту.

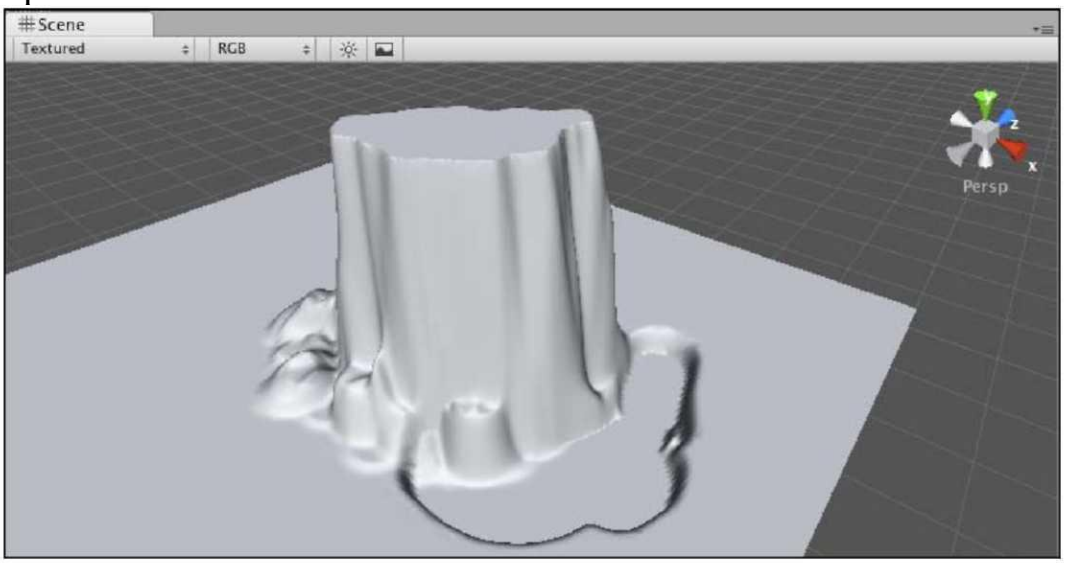


Высота краски

Этот инструмент работает так же на **Поднять** инструмент **Высоты**, но дает Вам дополнительную высоту урегулирования.



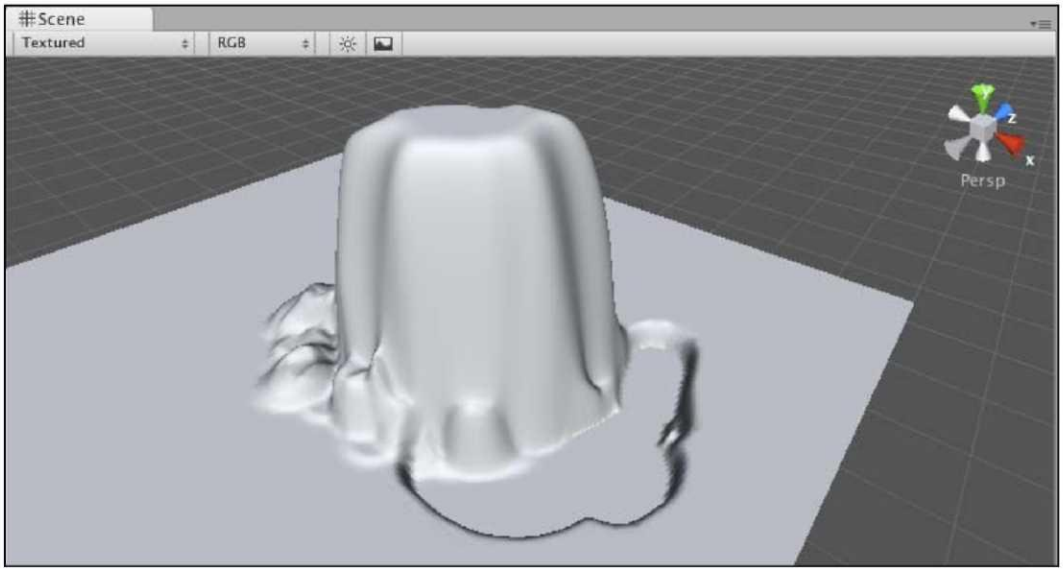
Это означает, что Вы можете определить высоту, чтобы нарисовать область ландшафта, Вы поднимаете пределы указания высоты, это выравнивается, разрешая Вам создать плато, как показано в следующем скриншоте:



Гладкая высота

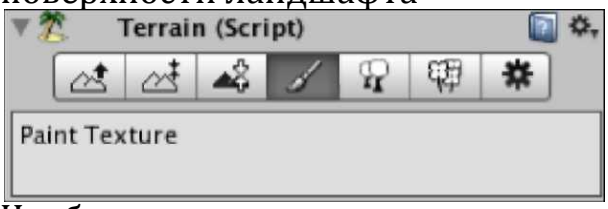
Этот инструмент используется главным образом, чтобы служить дополнением другим инструментам, таким как Высота Краски, чтобы смягчить резкие области топографии.

Например, в предыдущем плато, земля идет прямо, и если я желаю смягчить края поднятой области, я использовал бы этот инструмент, чтобы закруглить резкие края, создавая результат, показанный в следующем скриншоте:



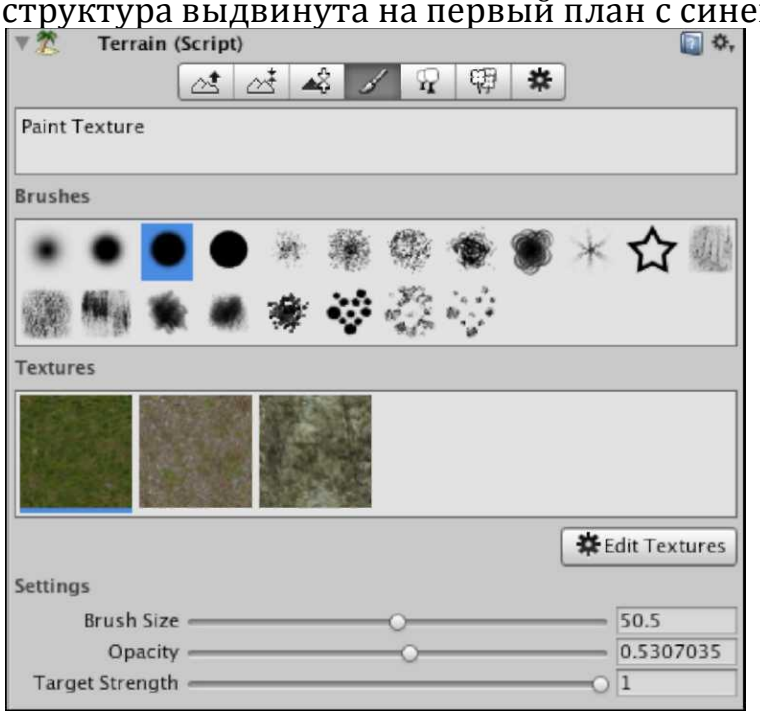
Кисть окраски

Кисть окраски - инструмент, используемый, чтобы покрывать выделенную поверхность текстурой в условиях ландшафта Unity - на поверхности ландшафта

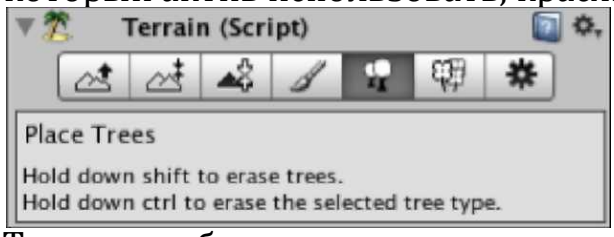


Чтобы рисовать ами, текстуры должны сначала быть добавлены к палитре в области **Текстур** этого инструмента. Текстуры могут быть добавлены, нажимая на кнопку **Edit Textures**, и отбор **Добавления Текстуры,** которая позволит Вам в настоящее время выбирать любой файл структуры в Вашем проекте, наряду с параметрами для того, чтобы крыть выбранную структуру черепицей.

Следующий скриншот - пример этого инструмента с тремя структурами в палитре. Первая структура, которую Вы добавляете к палитре, будет нарисована по всему ландшафту по умолчанию. Тогда, комбинируя несколько структур при изменении opacities и живописи вручную на ландшафт, Вы можете получить некоторые реалистические области любого вида поверхности, которую Вы надеетесь получить.
Чтобы выбрать структуру, чтобы нарисовать, просто нажмите на ее предварительный просмотр в палитре. В настоящее время выбираемая структура выдвинута на первый план с синей схемой.



Добавление дерева
Это - другой инструмент, который делает то, что предлагает его имя. Чистясь с мышью, или используя отдельные щелчки, **Деревья** могут использоваться, чтобы нарисовать деревья на ландшафт, определив который актив использовать, крася.



Таким же образом как определение структур для инструмента **Структуры Краски**, этот инструмент дает Вам кнопку **Edit Trees**, чтобы добавить, отредактировать, и удалить активы из палитры.

В его **Параметрах настройки** Вы можете определить:

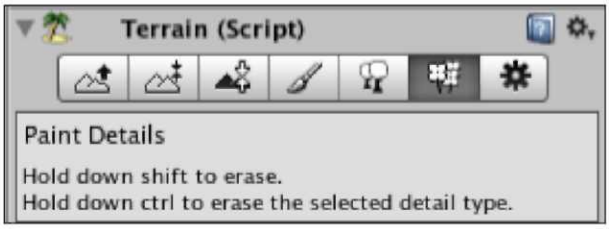
- **Размер Щетки:** количество деревьев, чтобы нарисовать за щелчок

- **Плотность Дерева:** близость деревьев поместила, краса
- **Цветное Изменение:** Применяет случайное цветное изменение к деревьям, краса несколько сразу
- **Ширина/Высота Дерева:** Измеряет актив дерева, которым Вы красите
- **Ширина Дерева / Изменение Высоты:** Дает Вам случайное изменение в калибровке, чтобы создать более реалистично засаженные деревьями области

Этот инструмент также использует *клавишу SHIFT*, чтобы полностью изменить ее эффекты. В этом случае, используя *Изменение* стирает нарисованные деревья и может использоваться в соединении с *клавишей CTRL*, чтобы только стереть деревья типа, отобранного в палитре.

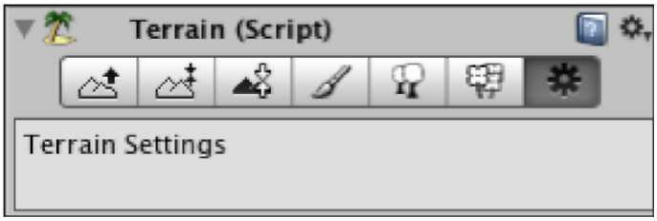
Детали Краски

Этот инструмент работает в подобной манере инструменту Добавления **деревьев**, но разработан, чтобы работать с объектами деталей, такими как цветы и другая листва.



Параметры настройки Ландшафта

Область **Параметров настройки Ландшафта Ландшафта (Сценарий)** содержит различные параметры настройки для рисунка ландшафта GPU компьютера (графическая единица обработки).



Здесь Вы можете определить различные параметры настройки, которые затрагивают **Уровень Деталей (LOD).**

Уровень Деталей в развитии игры определяет количество деталей, определенных в пределах определенного диапазона игрока. В этом ландшафте в качестве примера - мы должны быть в состоянии приспособить параметры настройки, такие как **Расстояние прорисовки,** которое является общим трехмерным понятием игры, которое отдает меньше деталей после определенного расстояния от игрока, чтобы улучшить работу.

В **Параметрах настройки Ландшафта,** например, Вы можете приспособить **Основное Расстояние Карты**, чтобы определить, как далеко далеко, пока ландшафт не заменяет с высокой разрешающей способностью графику для более низкого разрешения, делая объекты, на расстоянии менее мелкие, чтобы видеть.

Следующий скриншот - пример низкого **Основного Расстояния Карты** приблизительно 10 метров. Поскольку Вы можете видеть, структуры далее, чем указанное расстояние оттянуто в намного более низких деталях.

We'll look at the Terrain script's settings further as we begin to build our terrain.

Мы будем смотреть на параметры настройки сценария Ландшафта далее, поскольку мы начинаем строить наш ландшафт.

Солнце, Море, Создание песка остров

Установка 1 ландшафта шага

Теперь, когда мы смотрели на инструменты, доступные, чтобы создать наш ландшафт, давайте начнем, настраивая наш ландшафт, используя меню вершины **Ландшафта**. Гарантируйте, что Ваш Ландшафт все еще отобран в **Иерархии,** и идти в **Ландшафт | Решение Набора.**

Поскольку мы не хотим делать слишком большой остров для нашего первого проекта, установите **ширину** ландшафта и **длину** оба к **1000.** Не забудьте нажимать, *Вступают* после впечатывания этих ценностей так, чтобы Вы эффективно подтвердили их прежде, чем нажать **на Импорт.**

Затем, высота нашего острова должна начаться на ее уровне земли, а не на ноле, который является неплатежом для новых ландшафтов. Если Вы полагаете, что высота ландшафта ноля должна быть морским дном, то мы можем сказать, что наш уровень земли должен быть поднят, чтобы быть поверхностной высотой острова. Пойдите в **Ландшафт |, Сглаживают**

Height map.

Щелкните в строке **Высоты**, и месте в ценности **30** метров, и затем нажмите*, Вступают*, чтобы подтвердить. Нажмите **Сглаживаются**, чтобы закончиться.

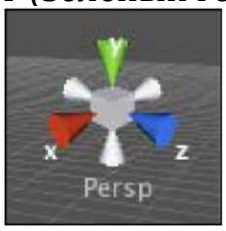
Это изменение ландшафт немного вверх. Однако, причина по которой, мы сделали это, состоит в том, что это - спасти временя - поскольку мы можем теперь сгладить вокруг краев ландшафта, используя инверсию**, Поднимаем Высоту**, чтобы уехать из поднятого острова в центре ландшафта. Это - более эффективный временем метод чем начинание с плоского острова и подъем высоты в центре.

Шаг схема С 2 островами

На **Инспекторе** для **Ландшафта** объекта Terrain **(Сценарий)** компонент, выберите **Поднять** инструмент **Высоты** - первая из этих семи кнопок.

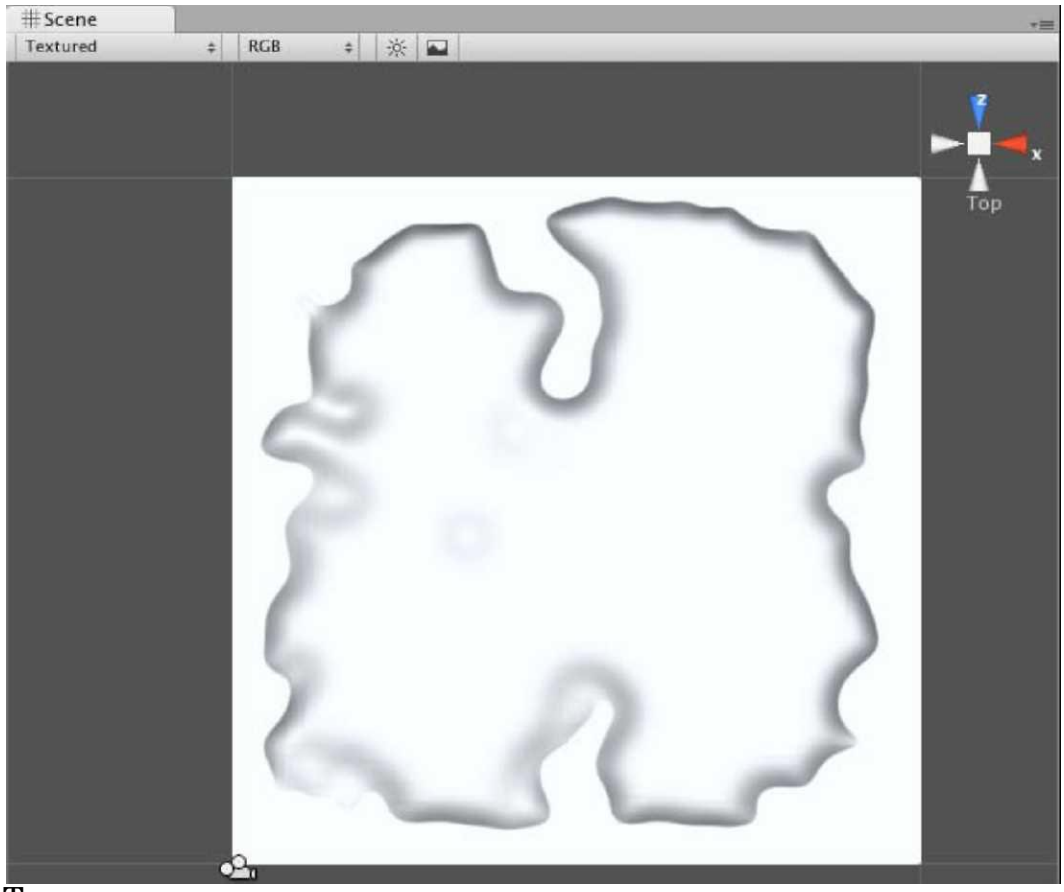
Выберите первую щетку в палитре, и установите ее **Размер Щетки** в **75.** Набор **Opacity** для щетки к **0.5.**

Измените Ваш взгляд в группе **Сцены** к нисходящему виду, нажимая на Ось Y (зеленый говорил) штуковины вида в верхнем правом углу.

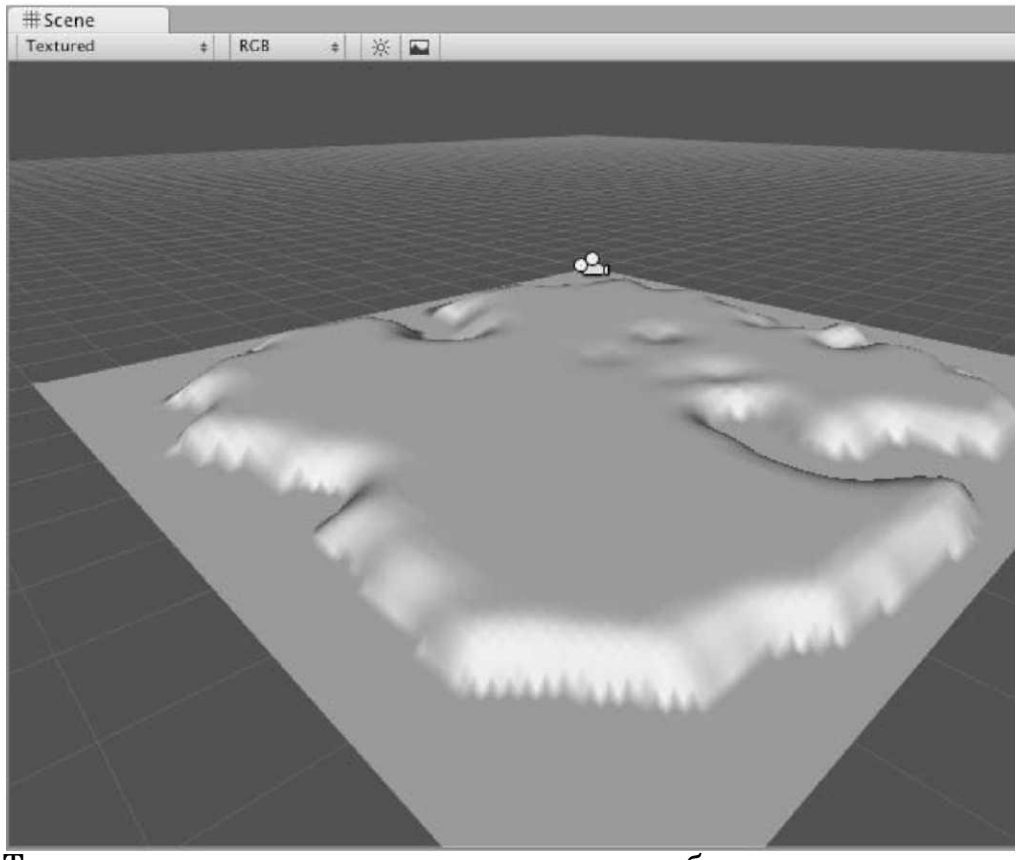


Используя *клавишу SHIFT*, чтобы **понизить высоту,** нарисуйте вокруг схемы ландшафта, создавая береговую линию, которая спускается к нулевой высоте - Вы будете знать, что это достигло ноля, поскольку ландшафт выровняется в этой минимальной высоте.

В то время как нет никакой потребности соответствовать схеме острова, я попытался создать, не сделать дико различную форму также, поскольку Вы будете нуждаться в плоском пространстве земли позже. Как только Вы нарисовали вокруг всей схемы, это должно смотреть кое-что как это:



Теперь переключите назад к перспективе трехмерного вида своей **Сцены**, нажимая на куб центра штуковины вида в верхнем правом из окна **Scene** и восхититесь своей ручной работой. Если Вы не уверены, вот то, на что должно быть похоже :



Теперь проведите некоторое время, пробегаясь через остров, используя **Поднять** инструмент **Высоты**, чтобы создать некоторые дальнейшие топографические детали, и возможно используя Более низкую Высоту (с *клавишей SHIFT*), чтобы добавить входное отверстие или озеро. Уезжайте из плоской области в центре Вашего ландшафта и одном свободном углу Вашей карты, в которой мы собираемся добавить вулкан!

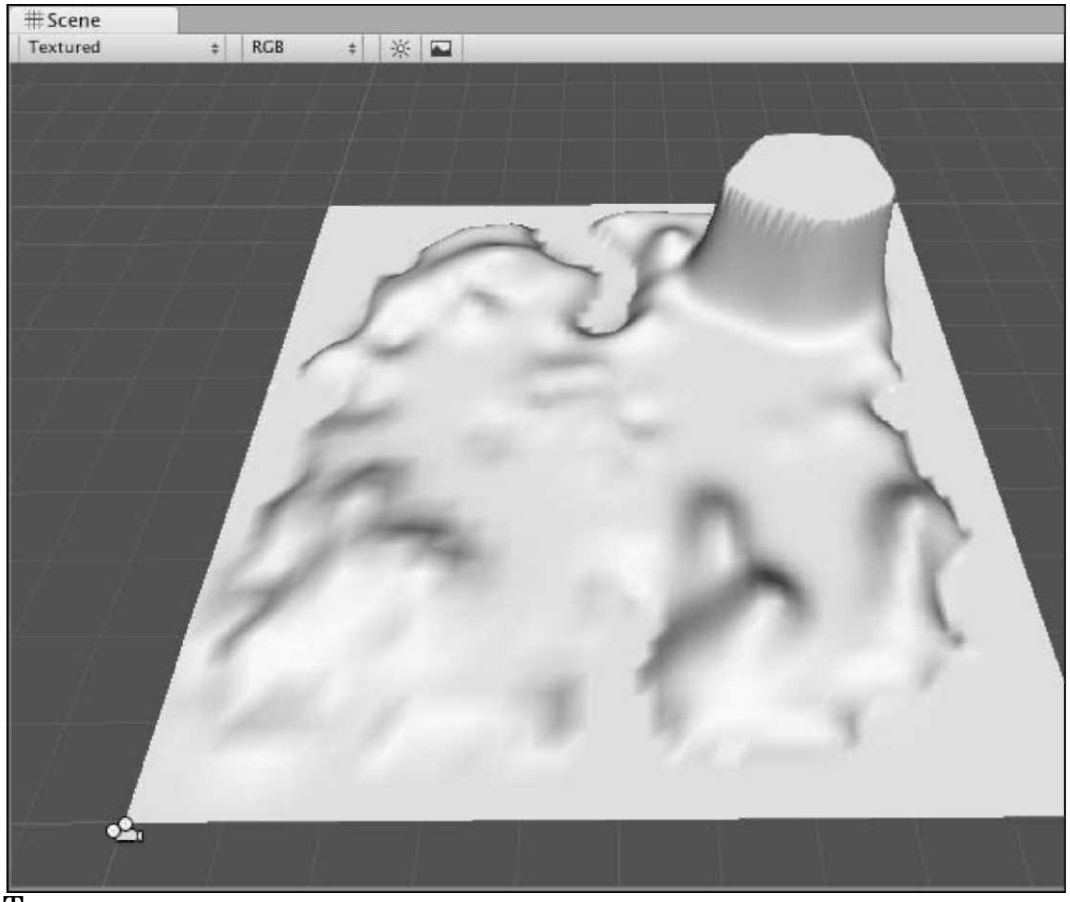
Шаг3, С вулканами!

Теперь создадим наш вулкан! Для этого мы объединим использование **Высоты Краски, Поднимем Высоту,** и **Пригладим** инструменты **Высоты**. Во-первых, выберите инструмент **Высоты Краски**.

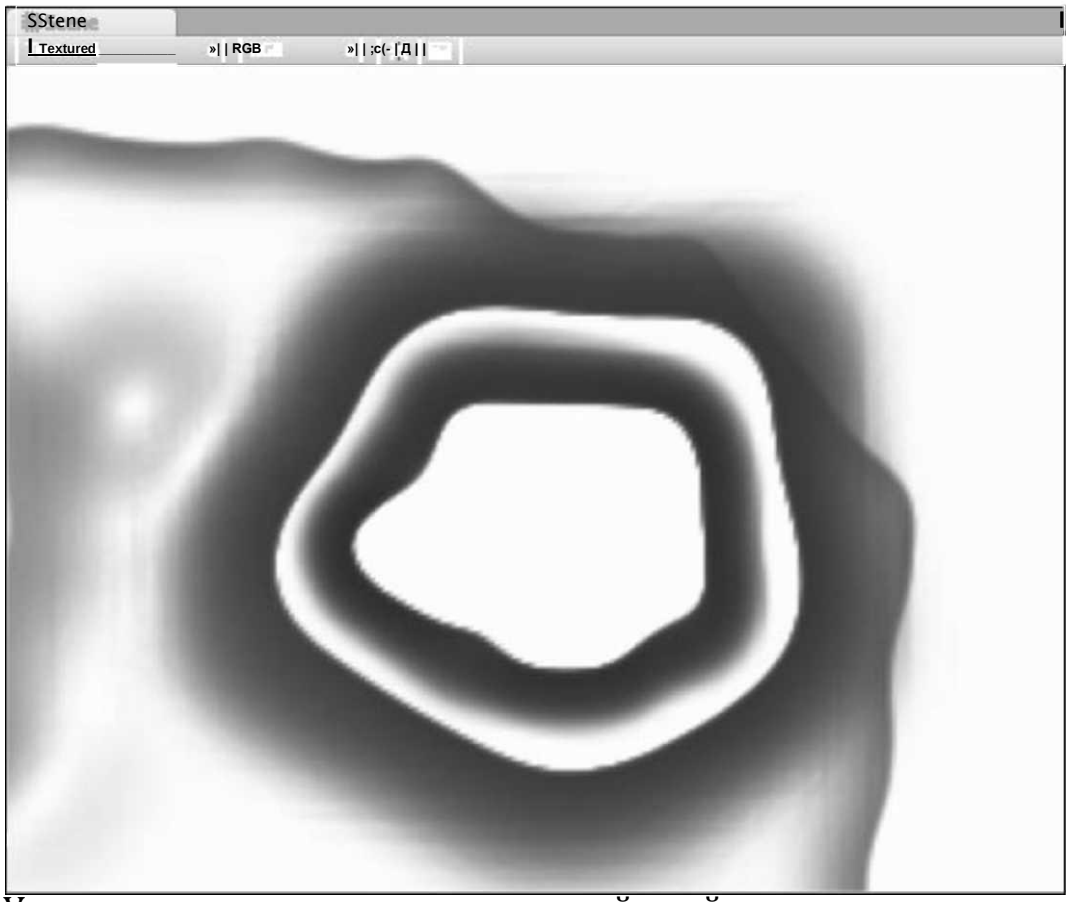
Выберите первую щетку в палитре, и установите **Размер Щетки** в **75, Opacity** к **0.5.** и **Высота** к **200.** Выберите **Вид сверху**, снова используя **штуковину вида,** и подрисуйте плато в углу, который Вы оставили свободным на Вашем ландшафте. Помните, что этот инструмент прекратит затрагивать ландшафт, как только это достигает указанной высоты **200.**

Ваш остров должен теперь быть похожим на это в перспективном виде:

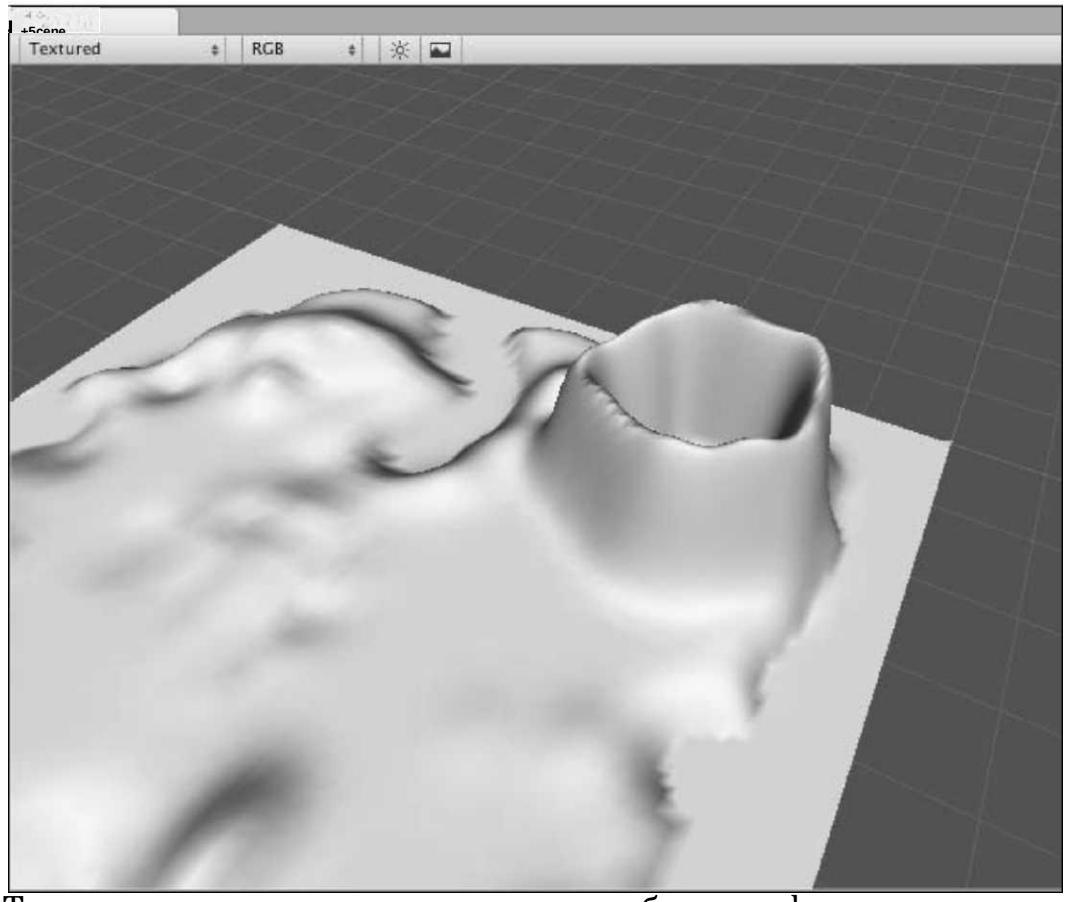
Your island should now look like this in the perspective view:



Теперь это плато может выглядеть неуклюжим - мы исправим это со сглаживанием коротко, но сначала мы должны создать рот вулкана. Так теперь с инструментом **Высоты Краски**, все еще отобранным, измените урегулирование **Высоты** на **20** и **Размер Щетки** к **30.**
Теперь поддержите на нужном уровне мышь и начните красить от центра плато за пределы к его краю в каждом направлении, пока Вы эффективно не выгнули плато, оставляя узкий горный хребет вокруг его окружности, как показано в следующем скриншоте:



У нас все еще есть довольно твердый край к этому горному хребту, и переключаясь на перспективный вид, Вы будете видеть, что это все еще не выглядит совершенно правильным. Это - то, где **Гладкий** инструмент **Высоты** входит. Выберите **Гладкий** инструмент **Высоты** и установите **Размер Щетки** в **30** и **Opacity** к **1.** Используя этот инструмент, нарисуйте вокруг края горного хребта с Вашей мышью, смягчая ее высоту, пока Вы не создали округленный мягкий горный хребет, как показано в следующем скриншоте:

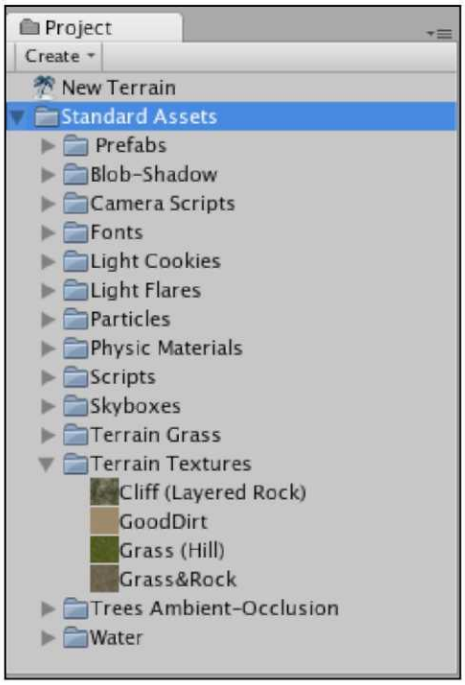


Теперь, когда наш вулкан начал приобретать форму, мы можем приступить к структуре острова, чтобы добавить реализм к нашему ландшафту.

Шаг 4 Добавление текстуры

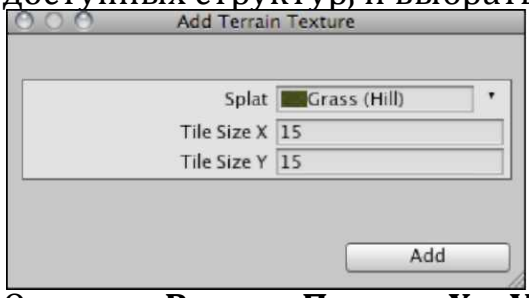
Когда текстурируете Ваш ландшафт, крайне важно помнить, что первая текстура, которую Вы добавляете, покроет ландшафт полностью. С этим в памяти, Вы должны гарантировать, что первая структура, которую Вы добавляете к своей палитре структуры, является структурой, которая представляет большинство Вашего ландшафта.

В **Стандартном Пакете Активов** мы включали, когда мы начали проект, нам дают различные активы со множеством особенностей игры, чтобы начать с. Также, Вы найдете папку под названием **Стандартные Активы** в Вашей **Проектной** группе. Расширьте это вниз, нажимая на серую стрелку налево от этого, и затем расширьте подпапку под названием **Структуры Ландшафта.**



Они - эти четыре текстуры, которые мы будем использовать, чтобы нарисовать наш ландшафт острова, таким образом мы начнем, добавлять их к нашей палитре. Гарантируйте, что объект игры **Ландшафта** все еще отобран в скриншоте, и затем выбирать инструмент **Текстурирования кистью** из **Ландшафта (Сценарий)** компонента в **Инспекторе.**

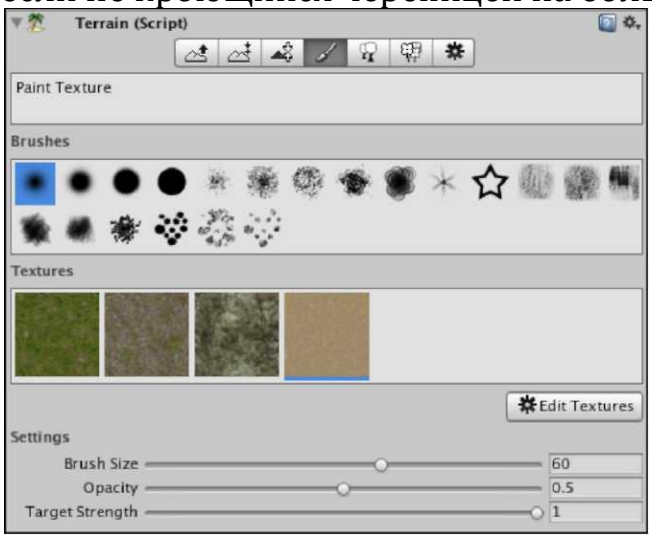
Чтобы ввести эти четыре текстуры для нашего ландшафта, начните, нажимая на кнопку **Edit Textures**, и избранный **Добавляют текстуры** от меню, которое высовывается. Это начнет **Добавить** окно диалога **Структур**, в котором Вы можете в настоящее время выбирать любую текстуру в Вашем проекте. Нажмите вниз стрелка направо от урегулирования **Нащельной рейки**, чтобы выбрать из списка всех доступных структур, и выбрать структуру под названием **Трава (Холм).**



Оставьте **Размер Плитки X** и **Y** на **15** здесь, поскольку эта структура покроет всю карту, и эта маленькая ценность даст нам более детально-выглядящую траву. Нажмите **Добавление** к концу. Это покроет Ваш ландшафт текстурой травы, поскольку это - первый, который мы добавили. Любые будущие текстуры, добавленные к палитре должны будут быть подрисованы вручную.

Повторите предыдущий шаг, чтобы добавить три дальнейших структуры к

палитре, выбирая текстуры под названием **Grass&Rock, Утес (Слоистая Скала),** и **GoodDirt** в то время как оставляя параметры настройки **Размера Плитки**, неизменные для всех кроме текстуры **Утеса (Слоистая Скала)**, у которой должны быть **Размер Плитки X** и **Y 70,** поскольку это будет применено к протянутой области карты и будет выглядеть искаженным если не кроющийся черепицей на больший масштаб.

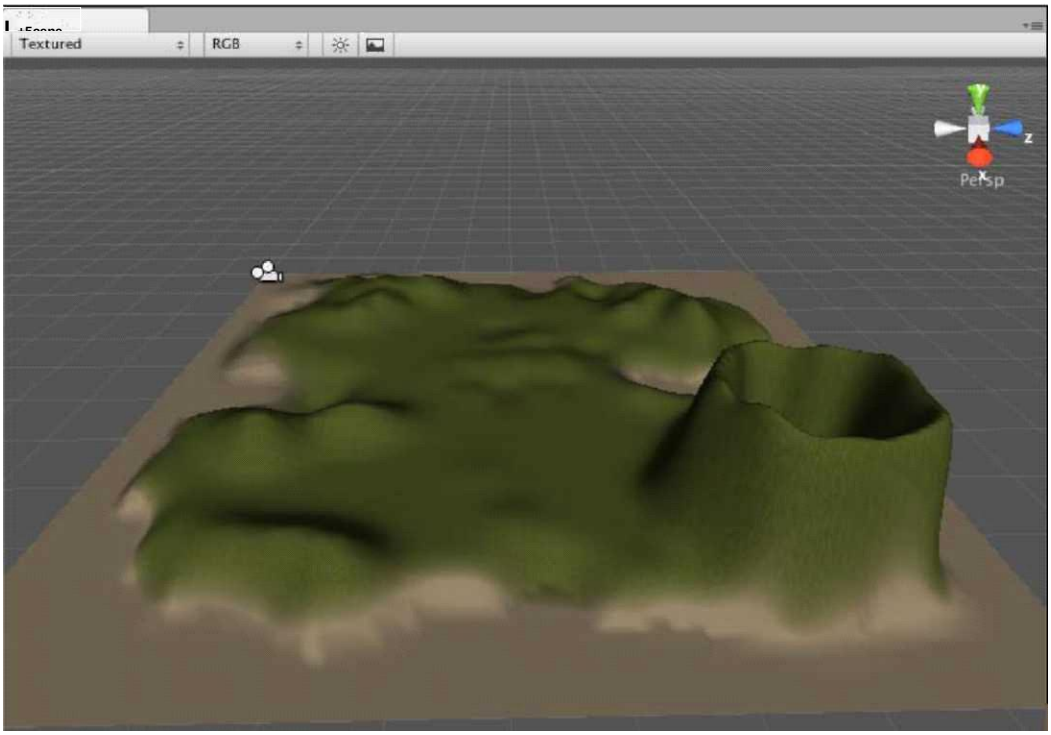


Песчаные области

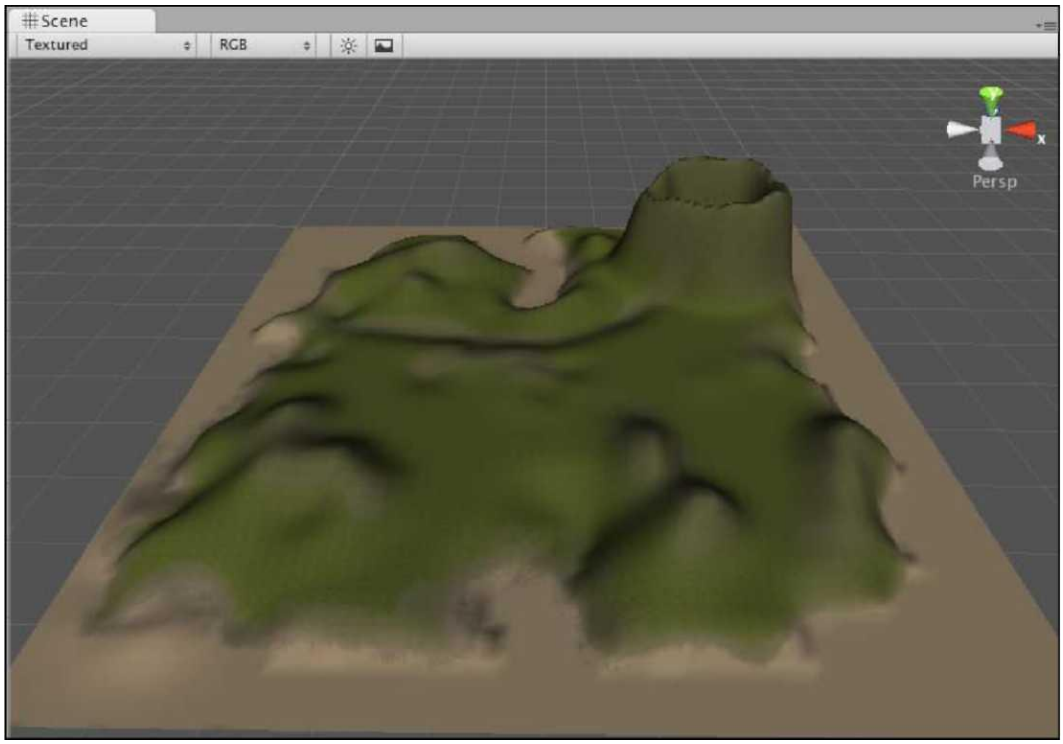
Вы должны теперь иметь все четыре текстуры в наличии в своей палитре. Соответствуя вышеупомянутым параметрам настройки, выберите последнюю добавленную текстуру - **GoodDirt**-и это должно стать выдвинутым на первый план синей подчеркивающей линией как показано. Установите **Размер Щетки** в **60, Opacity** к **0.5,** и **Целевую Силу** к **1.** Вы можете теперь нарисовать вокруг побережья острова, используя или вид **Вершины** или **Перспективы**.

Если Вы будете использовать **Перспективный** вид, чтобы нарисовать структуры, то он поможет помнить, что Вы можете использовать *клавишу ALT* при перемещении мыши, с любым **Рука** (Сокращение - *Q)* или **Преобразовать** (Сокращение - *W)* отобранные инструменты, чтобы вращать Ваш взгляд.

Когда закончено, у Вас должно быть кое-что как это:



Если Вы делаете ошибку, крася, Вы можете или использовать**, Редактируют |, Уничтожают**, чтобы отстраниться один мазок кисти, то есть, отдельный-0.held щелчок мыши, или выбрать структуру из палитры Трава & Скала
Затем выберите структуру **Травы & Скалы**, нажимая на второй добавленный ноготь большого пальца руки. Установите **Размер Щетки** в **25, Opacity** к **0.3,** и **Целевую Силу** к **0.5.** Теперь чиститесь по любым холмистым областям на Вашем ландшафте и вокруг верхней части вулкана, пока Вы не создали кое-что как это:

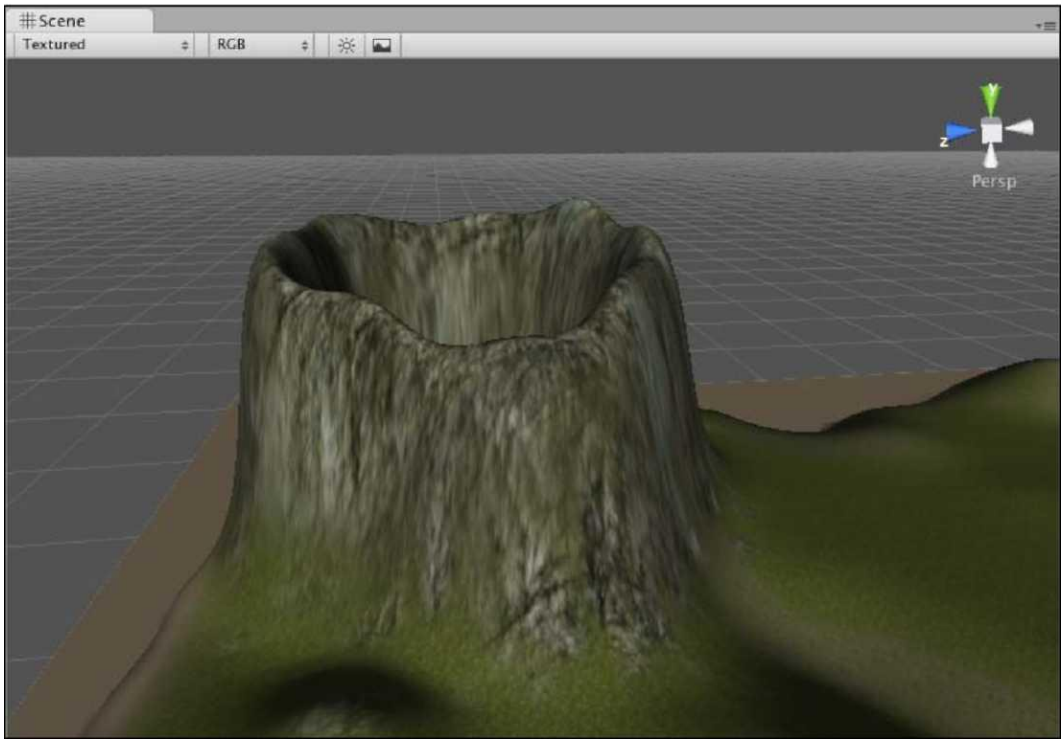


Скала Вулканов!

Теперь мы должны заставить наш вулкан выглядеть более реалистичным, добавляя **Утес (Слоистая Скала)** текстуру к нему. Выберите структуру утеса из палитры, и установите **Размер Щетки** в **25** и **Сила Opacity** и **Цели** к **1.**

Краска чрезмерно внешняя половина и все внутреннее с этими параметрами настройки и затем медленно уменьшают ценности **Размера Opacity** и **Щетки**, поскольку Вы работаете свой путь вниз за пределами вулкана так, чтобы эта структура была применена более тонко к основанию. С более низким opacity Вы можете также хотеть нарисовать по вершинам некоторых из Ваших более высоких hilled областей на ландшафте.

В то время как это возьмет некоторое экспериментирование, когда всё будет закончено, Ваш вулкан должен смотреть кое-что как это:



Шаг 5 время с деревьями
Теперь, когда у нас есть полностью текстурированный остров, мы нуждаемся в деревьях. В нашем Стандартном пакете Активов есть дерево, обеспеченное, чтобы начать нас с редактора ландшафта, и к счастью для нас, это - актив пальмы.

Выберите секцию **Деревьев Места Ландшафта (Сценарий)** компонент, и нажмите на кнопку **Edit Trees**. Из появившегося меню, которое появляется, выберите**, Добавить Дерево.**

Добавить окно диалога **Дерева** появляется. Как с некоторыми из других инструментов ландшафта, это *Добавляет, что* диалог позволяет нам выбирать из любого объекта соответствующего типа от нашей папки Активов.

Это не ограничено деревьям, обеспеченным в **Стандартных Активах,** что означает, что Вы можете смоделировать свои собственные деревья, экономя их в папку Активов Вашего проекта, чтобы использовать их с этим инструментом. Однако, мы собираемся использовать обеспеченную пальму Стандартных Активов. Так выберите актив **Пальмы**, нажимая вниз стрелка направо от урегулирования **Дерева**.
Фактор Изгиба здесь позволяет нашим деревьям колебаться на ветру. Этот эффект в вычислительном отношении, таким образом мы будем просто использовать низкое число. Напечатайте в ценности **2**, и пресса *Вступают*, чтобы подтвердить. Если Вы находите, что это вызывает низкую работу позже в развитии, то Вы можете всегда возвращаться к

этому урегулированию и задерживать его к нолю. Нажмите на кнопку **Add**, чтобы закончить.

С Вашей пальмой в палитре Вы должны видеть маленький предварительный просмотр дерева с синим фоном, чтобы показать, что это отобрано как дерево, чтобы поместить.

Установите **Размер Щетки** в **15** (живопись 15 деревьев за один раз) и **Плотность Дерева** к **0** (предоставление нам широкое распространение деревьев). **Изменение Цвета** Набора к **0.4**, чтобы дать нам различный набор деревьев и **Высоты Дерева / Ширина** к **1.5** с их параметрами настройки Изменения к **0.3.**

Используя отдельные щелчки, деревья места вокруг побережья острова, около песчаных областей, что Вы ожидали бы видеть их. Затем, чтобы служить дополнением ландшафту острова, поместите еще несколько пальм наугад местоположения внутри страны.

Помните, что, если Вы красите деревья неправильно в любое время, Вы можете считать, что *клавиша SHIFT* и щелчок, или краска (бремя) с мышью стирают деревья из ландшафта.

Шаг 6 трава

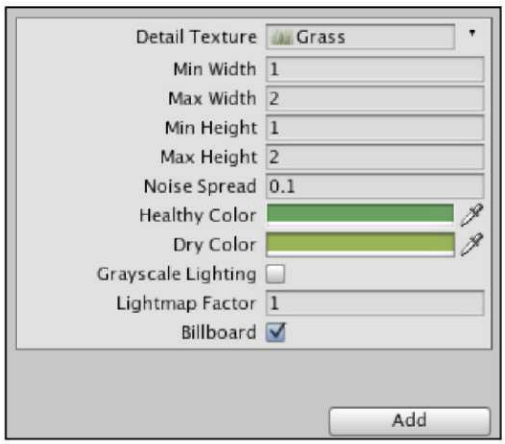
Теперь, когда у нас есть некоторые деревья на нашем острове, мы добавим небольшое количество травы, чтобы служить дополнением структурам травы, которыми мы покрывали ландшафт.

Выберите секцию **Деталей Краски Ландшафта (Сценарий)** компонент и нажмите на кнопку **Edit Details**. Избранный **Добавляют Структуру Травы** от подкидывать вверх меню.

Стандартный пакет Активов предоставляет нам структуру травы, чтобы использовать, таким образом от меню направо от урегулирования **Структуры Деталей**, выберите структуру просто под названием **Трава.**

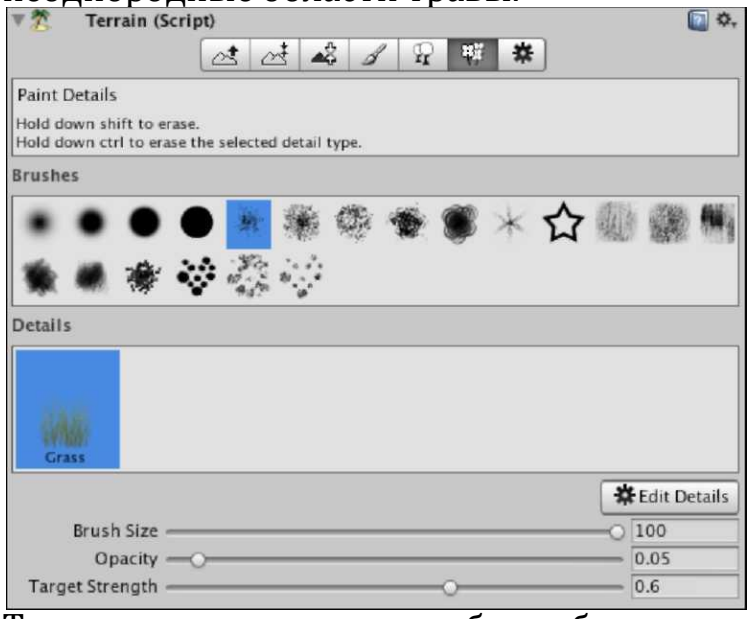
Выбрав структуру **Травы**, оставьте ценности **Ширины** и **Высоты** в их неплатеже и гарантируйте, что **Рекламный щит** отобран у основания этого диалога. Поскольку наши структуры деталей травы являются 2-ыми, мы можем использовать *billboarding,* технику в развитии игры, которое вращает структуру травы, чтобы стоять перед камерой во время игры игры, чтобы заставить траву казаться менее двумерной.

Используя коробки цветного сборщика, гарантируйте, что **Здоровые** и **Сухие** цвета имеют подобный оттенок зеленых к структурам, которые мы нарисовали на ландшафт.



Нажмите на кнопку **Add** у основания окна диалога, чтобы подтвердить добавление этой структуры к Вашей палитре.

Чтобы нарисовать траву на нашу карту, мы можем все снова и снова использовать основанную на мыши чистку подобным способом к другим инструментам ландшафта. Во-первых, мы должны будем выбрать щетку и параметры настройки, чтобы гарантировать широкую и несоизмеримую живопись деталей травы на нашу карту. Учитывая, что предоставление травы является другой дорогой особенностью компьютера, чтобы отдать, мы оставим траву к минимуму, устанавливая **Размер Щетки** в **100,** но **Opacity** к **0.05,** и **Целевую Силу** к **0.6.** Это даст широкое распространение с очень небольшим количеством травы, и выбирая щетку точечного пунктира (см. следующий скриншот), мы можем подрисовать неоднородные области травы.



Теперь увеличение масштаба изображения в ландшафт появляется, выбирая **Ручной инструмент**, и держа *клавишу CTRL* (PC) при перемещении мыши направо. Однажды на близком уровне увеличения

масштаба изображения, Вы будете в состоянии щелкнуть мышью, чтобы подрисовать области травы. Переместите остров, подрисовывающий, некоторые травянистые области это экономно по причинам работы - и позже Вы можете всегда возвращаться и добавлять больше травы, если игра выступает хорошо.

> Крайне важно изменить масштаб изображения в на Вашем ландшафте, крася детали, поскольку вид Сцены Редактора Unity не отдает им явно когда изменено масштаб изображения, чтобы сохранить память - так часто будет казаться, что, когда Вы изменили масштаб изображения, Ваша трава и другие детали исчезнут - не волнуются, дело обстоит не так.

Шаг 7 освещение

Теперь, когда наш ландшафт острова готов исследовать, мы должны будем добавить освещение к нашей сцене. Сначала приближаясь освещающий в Unity, лучше знать о том, для чего используются три различных легких типа:

- **Направленный свет:** Используемый как главный источник света, часто как солнечный свет, Направленный свет не происходит от отдельного пункта, но вместо этого просто едет в отдельном направлении.
- **Point light:** Эти огни происходят от отдельного пункта в трехмерном мире и используются для любого другого источника света, такого как внутреннее освещение, огни, пылающие объекты, и так далее.
- **Свет пятна:** Точно, что это походит, эти легкие сияния в отдельном направлении, но имеет ценность *радиуса*, которая может быть установлена, как сосредоточение фонаря.

Создание солнечного света

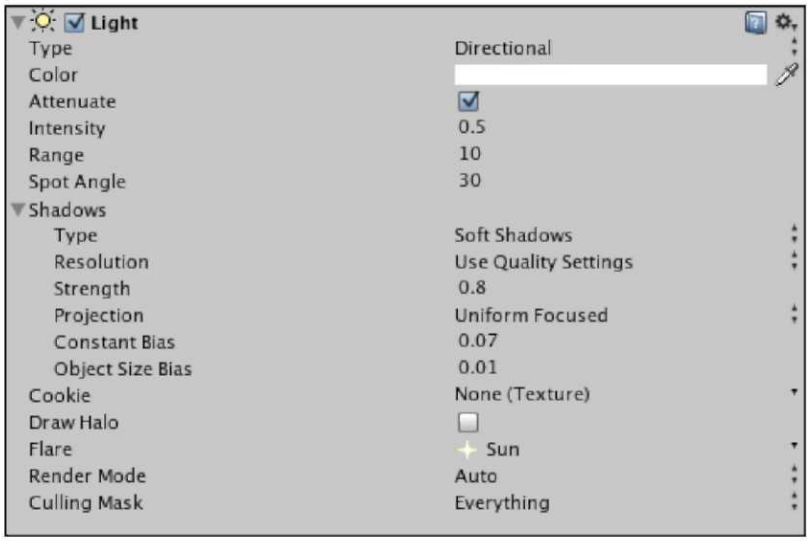
Чтобы представить наш главный источник света, мы добавим Направленный свет. Пойдите в **GameObject |, Создают Другой | Направленный Свет.** Это добавляет свет как объект в нашей сцене, и Вы будете видеть, что это теперь перечислено в **Иерархии.**

Поскольку Направленный свет не происходит от пункта, его позиция является обычно несоответствующей, поскольку он не может быть *замечен* только для игрока свет, он бросает, замечен. Однако, в этой обучающей программе, мы собираемся использовать Направленный свет, чтобы представить солнце, применяя легкую **Вспышку** к свету.

Чтобы представить солнце, мы поместим свет высоко над островом, и гарантируем, что это совместимо с направлением света, который это проливает, мы поместим свет далеко от острова в Оси Z.
Поместите свет в **(0, 500,-200)**, печатая в Инспекторе **Преобразовывают** коробки, и нажим *Вступают*, чтобы подтвердить каждый набор значений. Установите **X вращений** в **8**, чтобы наклонить свет вниз в Оси X, и это

прольет больше света на наш ландшафт.

Наконец, мы должны будем сделать свет видимым. Чтобы сделать это, выберите легкую вспышку, нажимая на опускаться стрелку направо от урегулирования **Вспышки** в **легком** компоненте - это не будет в настоящее время устанавливаться ни в **Один,** но вместо этого, выбирать **Солнце.**



Шаг 8 - Что является звуковым?

Часто пропускаемая область развития игры является звуковой. Для действительно immersive опыта, Ваш игрок должен не только видеть окружающую среду, которую Вы сделали, но и слышать её также.

Звук в Unity обработан через 2 компонента, **Звуковой Источник,** и **Звукового Слушателя.** Думайте об **Источнике** как о спикере в Вашем мире игры, и **Слушателе** как микрофон или ухо игрока. По умолчанию, у объектов игры камеры в Unity есть звуковой компонент слушателя. Столь данный, что у Вас всегда есть камера в Вашей игре, возможности, Вы должны будете только когда-либо настраивать звуковые источники. Это также стоит отмечать, что у **Звукового** компонента **Слушателя** нет никаких свойств приспособиться - это только работает - и что Unity сообщит Вам с ошибкой, если Вы случайно удалите единственного слушателя в какой-нибудь сцене.

Стерео против Моно

Unity обращается с поведением звука, используя звук (с двумя каналами) Стерео как постоянный объем, и Моно (отдельный канал) звук как объем Variables, основанный на близости его источника к слушателю в трехмерном мире.

Например, в следующих целях:

- В музыке игры: звук Стерео был бы лучшим, поскольку это останется постоянным независимо от того, где слушатель игрока входит в игру.

- Звуковой эффект Магнитофона в здании: Моно звук был бы лучшим - хотя Вы можете играть музыку, поскольку Ваш звуковой эффект, используя моно звуковые файлы позволил бы звуковому источнику становиться громче, если бы игрок добирается ближе до источника.

Форматы

Unity примет общие звуковые форматы - **WAV, MP3, AIFF,** и **OGG.** После столкновения со сжатым форматом, таким как MP3, Unity преобразовывает Ваш звуковой файл в **Ogg Vorbis** формат файла, оставляя несжатые звуки, такие как непеределанный WAVs.

Как с другими активами Вы импортируете, звуковые файлы просто преобразоваными, как только Вы переключаетесь между другой программой и Unity, поскольку Unity просматривает содержание папки Активов каждый раз, когда Вы переключаетесь на это, чтобы искать новые файлы.

Холмы живы!

Чтобы заставить наш остров чувствовать себя более реалистичным, мы добавим звуковой источник, играющий наружное окружение, используя звук стерео.

Начните, выбирая объект **Terrain** в **Иерархии.** Пойдите в **Компонент | Аудио | Звуковой Источник** от главного меню. Это добавляет **Звуковой Исходный** компонент к ландшафту. Поскольку объем остается постоянным, используя звуки стерео, и позиция является несоответствующей, мы могли поместить, окружение кажутся источником на любом объекте - это просто имеет логический смысл, что окружающий звук ландшафта должен быть присоединен к тому объекту игры.

В **Инспекторе** ландшафта Вы будете теперь видеть **Звуковой Исходный** компонент, с которым Вы можете или выбрать файл, чтобы играть или оставить, если Вы планируете играть звуки через scripting.

Импортирование Вашего первого пакета

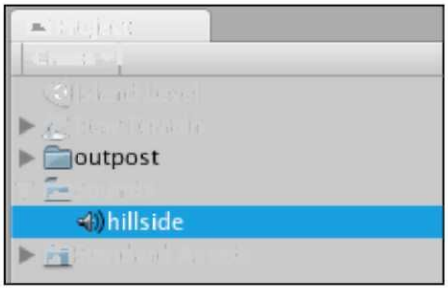
Для этого шага Вы должны будете загрузить первый из нескольких пакетов актива от вебсайта публикации Packt, www.packtpub.com/files/8i8i_Code.zip <http://www.packtpub.com/files/8i8i_Code.zip>. Определите местонахождение файла по имени SoundLunitypackage от списка файлов, связанных с этой книгой, и затем возвратитесь к Unity и пойдите в **Активы | Пакет Импорта.**

Вам подарят диалог **Пакета Импорта** выбора файла, от которого Вы должны провести к местоположению на Вашем жестком диске, что Вы сохранили загруженный файл. Выберите это, и затем нажмите **Открытый**, чтобы выбрать это. Unity тогда покажет Вам список активов, которые содержатся в пакете по умолчанию, это предполагает, что пользователь хочет импортировать все активы в пакете, но способность только

импортировать определенные активы полезна, извлекая отдельные файлы из больших пакетов.

В нашем пакете Sound1 есть просто отдельный файл звука формата MP3 под названием hillside.mp3 - наряду с папкой под названием Звук, чтобы оставить это в - таким образом Вы будете только видеть те два файла.

Нажмите **на Импорт**, чтобы подтвердить добавление этих файлов к Вашему проекту и после короткого конверсионного бара продвижения, Вы должны видеть их в своей проектной группе - Вы должны будете открыть **Звуковую** папку, которую Вы добавили, нажимая на серую стрелку налево от этого, чтобы видеть файл звука **склона**. Звуковые файлы представлены в Вашей **Проектной** группе изображением спикера, как показано в следующем скриншоте:

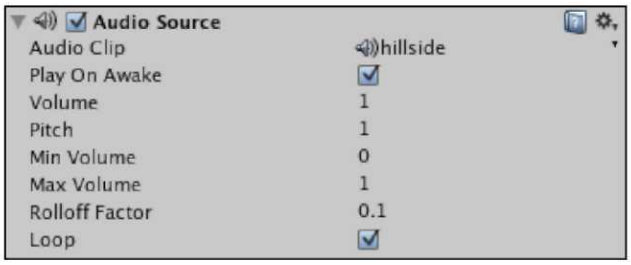


Файл **склона** теперь готов быть примененным к **Звуковому Исходному** компоненту ландшафта.

Нажмите вниз стрелка направо от **Звукового** урегулирования **Скрепки** в **Звуковом Исходном** компоненте объекта игры ландшафта. Это покажет Вам список всех доступных звуковых файлов в Вашем проекте. Поскольку у нас только есть файл аудио **склона** в нашем, который является всем, что Вы будете видеть на этом, опускаются меню, столь выберите это теперь.
Дальнейшие звуковые параметры настройки
У **Звукового Исходного** компонента есть различные параметры настройки, чтобы управлять, как звуковая скрепка звучит, так же как воспроизводит. Для нашего окружающего звука склона, просто гарантируйте, что checkboxes для **Игры На Активном** и **меше** отобраны.

Это будет играть скрепку звука окружения, когда игрок войдет в сцену (или *уровень)* и непрерывно закреплять петлей скрепку, пока из сцены не выходят.

Ваш **Звуковой Исходный** компонент должен быть похожим на это:



Поскольку наш звук находится в Стерео, и поэтому не затронут расстоянием, **Минута / Макс громкость** и параметры настройки **Фактора Rolloff** не обращаются. Однако, используя Моно звук, они делают следующее:

- **Минимальный/Максимальный Объем:** самое тихое/самое громкое звук может быть независимо от звуковой близости слушателя
- **Фактор Rolloff:** Как быстро аудио исчезает как звуковые шаги слушателя к/далеко из источника

Шаг 9 - Смотрите на skybox!

В создании трехмерной окружающей среды, горизонта или расстояния представлен дополнением **skybox.** skybox - **cubemap**-а ряд шести структур, помещенных в кубе и предоставленный легко, чтобы появиться как окружающее небо и горизонт. Этот cubemap фактически горизонт, это не объект, до которого может когда-либо *добираться* игрок.

Чтобы применить skybox к Вашей сцене, пойдите, чтобы **Отредактировать |, Отдают Параметры настройки.** Область **Инспектора** интерфейса теперь изменит на показ предпочтение предоставлению этой сцены. Направо от **Материального** урегулирования **Skybox**, нажмите на опускаться стрелку, и выберите **Закат.** Это применит skybox, но крайне важно понять, что любой skybox, который Вы добавляете, только покажут в группе **Game View** по умолчанию.
Как со многими skyboxes, **Закат** skybox предоставленный Стандартные Активы показывает продвинутый Солнце. Проблема здесь состоит в том, что нам представляла наше солнце визуально вспышка направленного света, таким образом мы должны будем повторно поместить свет, чтобы соответствовать skybox. Но сначала, мы добавим наш **Первый** характер игрока **Диспетчера Человека**, так, чтобы Вы могли видеть эту проблему на собственном опыте.

Шаг 10 Открытая вода

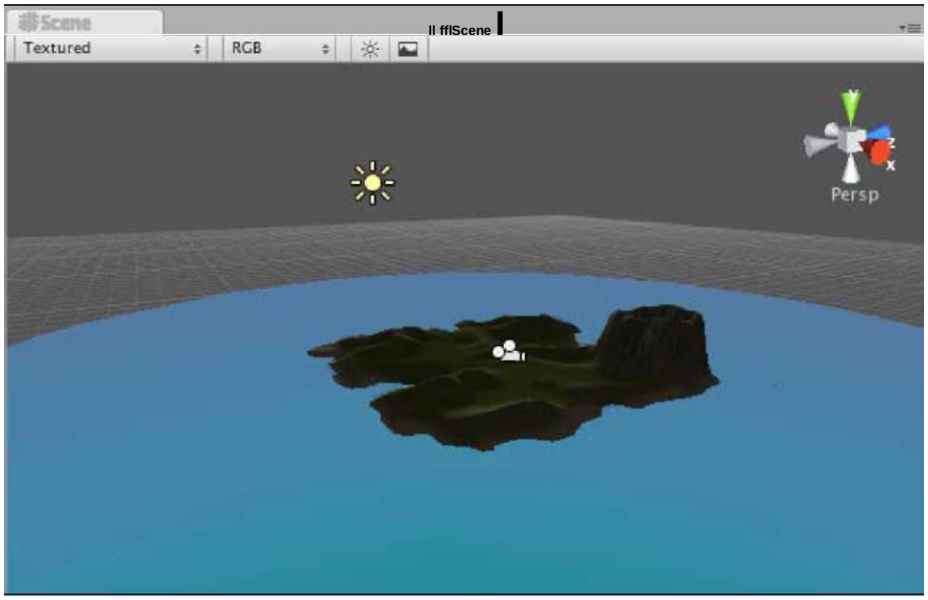
Поскольку мы построили ландшафт острова, из этого следует, что наш континентальный массив должен быть окружен водным путем. Вода в Unity создана, используя оживляемый материал, относился к поверхности. В то время как возможно создать далее динамические водные эффекты, используя частицы, лучший способ добавить, что большое пространство воды должно использовать один из водных материалов, обеспеченных Unity.

Стандартный пакет Активов дает нам две готовых поверхности с водным примененным материалом. С готовностью спасенный как prefab, эти объекты могут быть введены легко от **Project** Panel. Просто откройте **Стандартные Активы | Водная** папка.

Тяните **Дневной свет Простой Водный** prefab в сцену и поместите это в **(500, 4, 500)**, заполняясь в этих X, Y и ценностях Z для **Позиции** в компоненте **Transform (преобразовать)** в **Инспекторе.** Чтобы расширить

масштаб водного prefab, чтобы сформировать море вокруг острова, просто увеличьте и **X** и ценности **Z** под **Масштабом** к **1600.**

Это помещает воду в центр карты и на четыре метра выше морского дна. Это покроет углы Вашего острова, и замаскирует истинную основанную на квадрате природу ландшафта. Это должно быть похожим на это:



Шаг 11 прогулка по острову

В **Проектной** группе, расширьте подпапку **Стандартных Активов** под названием **Prefab.** В здесь Вас найдет отдельный prefab - готовый объект **First Person Controller**. Это может быть брошено в Вашу сцену, и позволит Вам идти вокруг Вашего ландшафта, используя клавиатуру и мышь.

Тянитесь этот prefab от **Проектной** группы на вид **Сцены** - помнят, что Вы не будете в состоянии к, точно помещают этот объект, когда Вы первоначально бросаете это в сцену, поскольку это должно быть сделано впоследствии.

Теперь, когда Ваш объект **First Person Controller** - активный объект игры, Вы можете видеть, что он перечислял в **Иерархии** и ее составных частях, перечисленных в **Инспекторе.**

Повторно поместите объект **First Person Controller** в **(500, 35, 500)**, печатая ценности компонент **Transform (преобразовать)** в **Инспекторе.** Эта позиция помещает характер непосредственно в центр карты в X и Осях Z, потому что мы создали ландшафт с шириной и длиной 1000 метров. Ценность **Y 35** гарантирует, что игрок - наземный уровень - если это не будет, тогда то характер провалится карта после игры. Если Вы, случается, сделали холм в этой позиции, ваяя Ваш ландшафт, просто увеличили ценность **Y**, чтобы поместить характер выше холма в начале игры. Чтобы видеть, где Ваш объект характера помещен более легко, гарантируйте, что он отобран в **Иерархии.** Тогда толпитесь курсор мыши по окну **Scene** и прессе *F* на клавиатуре. Это сосредотачивает вид **Сцены**

относительно отобранного объекта.
Теперь нажмите кнопку **Play**, чтобы проверить игру, и Вы должны быть в состоянии идти вокруг на Вашем ландшафте острова!

Средства управления по умолчанию для характера следующие:

- Up/*W:* Прогулка вперед
- Down/*S:* Прогулка назад
- Left/*A:* Отступите оставленный (также известный как *Обстреливающий)*
- Право / *D:* Обойдите право
- Мышь: Смотрите вокруг/поворот игрока, идя

Прогулка к вершине холма и поворота, пока Вы не можете видеть вспышку Солнца. Тогда смотрите направо от вспышки, и Вы будете видеть, что у skybox также есть освещенная область, чтобы представить солнце, таким образом мы должны повторно поместить объект **Directional Light**, чтобы соответствовать солнцу в skybox, как упомянуто ранее.
Остановите тестирование игры, нажимая кнопку **Play** снова. Крайне важно остановить игру, когда Вы продолжаете редактировать. Если Вы оставляете это игрой или на паузе, то любой редактирует Вас, делают к сцене, только будет временная длительность, пока Вы не нажимаете **Игру** снова или **оставляете** Unity.

Шаг 12 финал tweaked
Выберите объект **Directional Light** в **Иерархии,** и в компоненте **Transform (преобразовать)** в **Инспекторе,** повторно поместите объект в **(-500, 500, 500).** Это поместит свет в сторону карты, которая идет солнце skybox. **Вращайте** объект **120** в Оси Y (коробка центра), и позиция Вашего **Направленного света** будет теперь соответствовать свету в skybox.

Наконец, поскольку мы добавили объект **First Person Controller**, мы больше не нуждаемся в объекте, с которым наша сцена шла неплатежом - **Главная Камера.** Unity напоминает нам об этом, показывая сообщение информации в области предварительного просмотра **пульта** у основания экрана:
Чтобы исправить это, просто удалите объект **Main Camera**, поскольку мы больше не нуждаемся в нем. Чтобы сделать это, выберите это в **Иерархии,** и нажмите *Shift+Delete* использования PC, или на любой платформе, нажмите, чтобы **Отредактировать |, Удаляют.**

Теперь, когда этот объект ушел, Ваш ландшафт острова полон. Сохраните свою сцену **Файл |, Сохранение Сцены** и называют Ваш **Уровень Острова** сцены - Unity предположит, что Вы хотите спасти в папке Активов Вашего проекта, и выберете ту папку как спасти местоположение автоматически, потому что все активы должны быть в этой папке. Чтобы оставить вещи опрятными, Вы можете сделать подпапку **Уровней** теперь и сохранить там, если Вам нравится.

Поздравления, Ваш ландшафт острова готов к исследованию, таким образом поражает кнопку **Play** .Только не забудьте нажимать **Play** снова, когда Вы закончите.

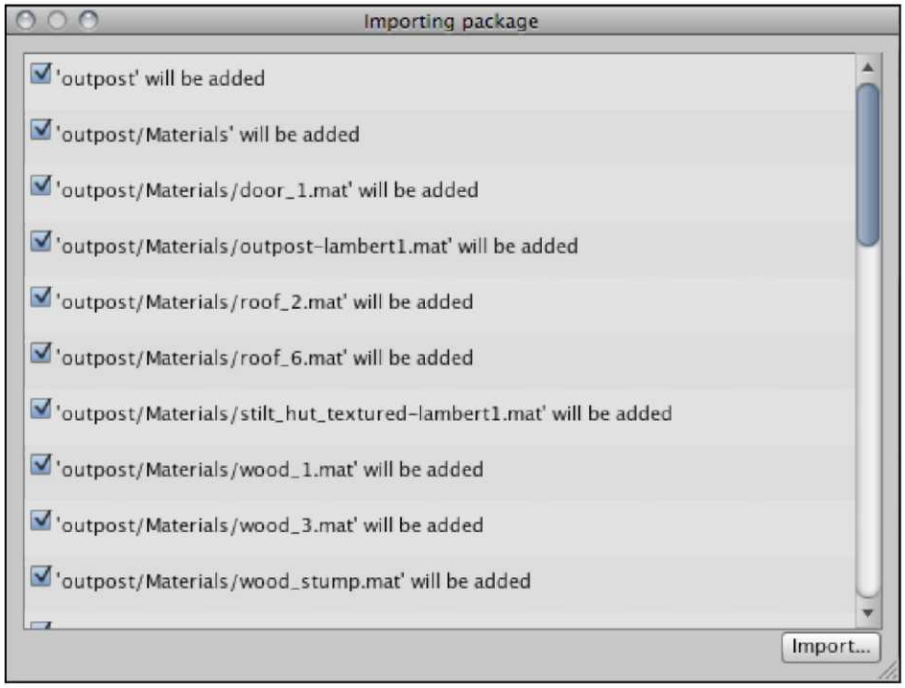
Отведите Меня домой! Представление моделей

Теперь, когда наш ландшафт острова готов к заселению с gameplay, первое ключевое понятие, которое исследует, является введением моделей от внешних применений. Поскольку мы предоставили Вам модели для упражнений в этой книге, Вы должны будете загрузить и импортировать другой пакет Unity, чтобы добавить первый образцовый актив - заставу - к Вашему проекту.

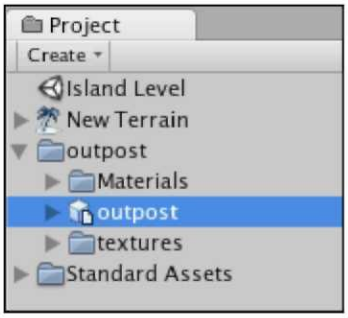
Импортирование образцового пакета

Пойдите в www.packtpub.com/files/8i8i_code.zip <http://www.packtpub.com/files/8i8i_code.zip> и определите местонахождение пакета актива от файлов, доступных в кодовой связке для названного Outpost.unitypackage этой книги.

Затем, возвратитесь к Unity и пойдите в **Активы | Пакет Импорта.**



Оставьте все файлы здесь отобранными, и нажмите на кнопку **Import**, чтобы подтвердить. Теперь, когда Вы импортировали содержание пакета, Вы будете видеть новую папку активов, названных **заставой** в Вашей **Проектной** группе, и поэтому в папке **Активов** Вашего проекта в Вашей операционной системе. Расширьте эту папку, нажимая на серую стрелку рядом с ее именем на **Проектной** группе, чтобы рассмотреть ее содержание.



Импортированная папка содержит модель и две подпапки, один с ее **Материалами,** другой содержащий изображения, которые составляют **текстуры** для тех материалов. Вы можете определить трехмерный образцовый актив в Unity его изображением-а маленькое изображение куба с сопровождающим изображением страницы, дифференцируя это от заранее приготовленного изображения Unity, которое, поскольку Вы обнаружите, является просто изображением куба отдельно.

Общие параметры настройки для моделей

Прежде, чем Вы введете любую модель активной сцене, Вы должны всегда гарантировать, что ее параметры настройки состоят в том, поскольку Вы требуете, чтобы они были в Инспекторе. Когда Unity импортирует новые модели к Вашему проекту, он интерпретирует их со своим Импортером **формата файла FBX**.

При использовании компонента **Импортера FBX** в **Инспекторе** Вы можете выбрать свой образцовый файл в окне **Project** и приспособить параметры настройки для его **Петель, Материалов,** и **Мультипликаций** прежде, чем Ваша модель станет частью Вашей игры.

Меши

В секции **Импортера FBX** Вы можете определить:

- **Коэффициент пропорциональности:** Типично набор к ценности 1, это урегулирование сообщает, что 1 единица должна равняться 1 метру в мире игры. Если Вы желаете, чтобы Ваши модели были измерены по-другому, то Вы можете приспособить их здесь прежде, чем Вы добавите модель к сцене. Однако, Вы можете всегда измерять объекты, как только они находятся в Вашей сцене, используя параметры настройки **Масштаба** компонента **Transform (преобразовать)**.
- **Произведите Коллайдеры:** Этот checkbox найдет каждую отдельную составную часть модели и назначит коллайдер на это. Коллайдер - сложный коллайдер, который может соответствовать к сложным геометрическим формам и, в результате является обычным типом коллайдера, Вы ожидали бы хотеть обратиться ко всем частям карты или трехмерной модели здания.
- **Вычислите Normals:** *нормальной* является передовая поверхность столкновения каждого **меша** в Вашей трехмерной модели, и поэтому

сторона, которая предоставлена как видимая. Позволяя этот checkbox, Вы позволяете Unity гарантировать, что все поверхности правильно настроены, чтобы отдать в игре.

- **Англ Сглаживания:** используя **Вычисление** Functions **Normals**, сглаживание позволяет Вам определять, насколько детальный край должен быть должен считаться твердым краем двигателем игры.
- **Тангенсы Раскола:** Это урегулирование позволяет исправления двигателем для моделей, импортированных с неправильным Ударом Нанесенное на карту освещение. Картография Удара - система использования двух структур, один графика, чтобы представить внешность модели и другой heightmap. Комбинируя эти две структуры, метод карты удара позволяет двигателю предоставления показывать плоские поверхности многоугольников, как будто у них есть трехмерные деформации. Создавая такие эффекты в имеющих отношение к третьей стороне применениях и переходя к Unity, иногда освещение может появиться неправильно, и этот checkbox разработан, чтобы установить это, интерпретируя их материалы по-другому.
- **Обмен UVs:** Это урегулирование позволяет исправление ошибок импорта при освещении shaders введенный от имеющих отношение к третьей стороне применений.

Материалы

Секция **Материалов** позволяет Вам выбирать, как интерпретировать материалы, созданные в Вашем имеющем отношение к третьей стороне трехмерном применении моделирования. Пользователь может выбрать, любой **За Структуру** (создает материал Unity для каждого найденного файла изображения структуры), или **За Материал** (создает Материалы только для существующих материалов в оригинальном файле) от **Поколения**, опускаются меню.

Мультипликации

Секция **Мультипликаций** Импортера позволяет Вам интерпретировать мультипликации, созданные в Вашем применении моделирования многими способами. От **Поколения** опускаются меню, Вы можете выбрать следующие методы:

- **Не Импортируйте:** Заставьте модель не показывать мультипликацию.
- **Магазин в Оригинальных Корнях:** Заставьте модель показывать мультипликации на отдельных родительских объектах, поскольку родитель или объекты корня могут импортировать по-другому в Unity.
- **Магазин в Узлах:** Заставьте модель показывать мультипликации на отдельных детских объектах всюду по модели, позволяя больше контроля за сценарием мультипликации каждой части.
- **Магазин в Корне:** Заставьте модель только показывать

мультипликацию на родительском объекте всей группы.

Секция **Мультипликаций** тогда показывает три далее checkboxes:

- **Испеките Мультипликации:** Скажите Unity интерпретировать суставы в моделях с ИКОМ (Обратный Kinematics) скелетная мультипликация.
- **Уменьшите Keyframes:** Это удаляет ненужный keyframes в экспортируемых моделях от моделирования применений. Это должно всегда отбираться, поскольку Unity не нуждается в keyframes, и выполнение так улучшит работу оживляемой модели.

Мультипликации Раскола: создавая модели, которые будут использоваться с Unity, аниматоры создают основанную на графике времени мультипликацию, и отмечая их диапазоны структуры, они могут добавить каждую область мультипликации в их графике времени, определяя название и структуры, в которых имеет место каждая мультипликация. Преимущество этого состоит в том, что это позволяет Вам звать отдельные мультипликации по имени когда scripting.

Мультипликации Раскола: создавая модели, которые будут использоваться с Unity, аниматоры создают основанную на графике времени мультипликацию, и отмечая их диапазоны структуры, они могут добавить каждую область мультипликации в их графике времени, определяя название и структуры, в которых имеет место каждая мультипликация. Преимущество этого состоит в том, что это позволяет Вам звать отдельные мультипликации по имени когда scripting.

С моделью заставы Вы импортировали и выбрали в **Проектном виде,** мы будем использовать **компонент Импортера FBX** в **Инспекторе**, чтобы приспособить параметры настройки для заставы.
Гарантируйте что:

- Под **мешами: Коэффициент пропорциональности** установлен в **1,** и **Производить Коллайдеры/, Вычисляют, Normals** отобраны
- Под **Материалами: Поколение** установлено в **За Структуру**
- При **Мультипликациях: Уменьшите Мультипликации Keyframes** и **Раскола**, отобраны

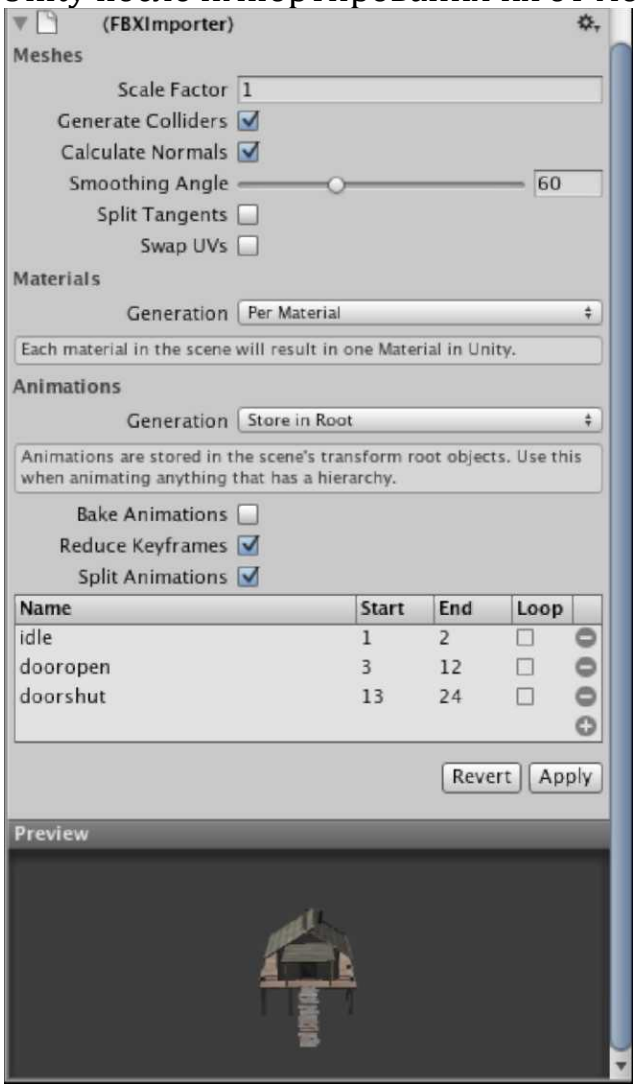
Теперь использование стола базировало область в основании, добавить, три скрепки мультипликации, нажимая **+** (Добавьте символ), кнопка направо.

Первую мультипликацию автоматически называют праздной, который прекрасен, но Вы должны будете определить диапазон структуры. Поэтому, под **Началом,** поместите ценность **1**, чтобы сказать Unity начинать на структуре **1,** и под **Концом,** определять ценность **2.**

Повторите этот шаг, чтобы добавить две дальнейших мультипликации:

- **dooropen**-от структуры 3 - 12
- **doorshut**-от структуры 13 - 24

Примите во внимание, что эти названия мультипликации с учетом регистра когда дело доходит до запроса их с scripting, так гарантируйте, что Вы пишете Ваш буквально как показано всюду по этой книге. Область **Петли** стола Мультипликаций может вводить в заблуждение для новых пользователей. Это не разработано, чтобы закрепить петлей специфическую мультипликацию, которую Вы устанавливаете - это обработано **Способом Обертки** мультипликации, как только это находится в сцене. Вместо этого эта особенность добавляет отдельную дополнительную структуру к мультипликациям, которые будут закреплены петлей, но начало и структуры конца которого не совпадают в Unity после импортирования их от моделирования применений.



При условии, что Ваша модель заставы настроена как описано выше,

нажмите на кнопку **Apply**, чтобы подтвердить эти параметры настройки импорта, и Вы все сделаны - модель должна быть готова быть помещенной в сцену и использоваться в нашей игре.

Резюме

В этой главе мы исследовали основы развития Вашей первой окружающей среды. Начиная только с плоского самолета, Вы теперь создали полностью исследовательский ландшафт острова в коротком периоде времени. Мы также смотрели на освещение и звук, два основных принципа, что Вы обратитесь в каждом виде игры, предполагают, что Вы сталкиваетесь.

Помните, Вы можете всегда возвращаться к инструментам ландшафта, включенным в эту главу в любое время, чтобы добавить больше деталей к Вашему ландшафту, и как только Вы чувствуете себя более уверенными относительно звука, мы возвратимся к добавлению дальнейших звуковых источников к острову позже в книге.

Поскольку Вы продолжаете работать через эту книгу, Вы обнаружите все типы дополнительных нюансов, которые Вы можете принести к окружающей среде, чтобы далее приостановить недоверие в игроке.

Мы будем смотреть на добавление динамического чувства к нашему острову, когда мы будем смотреть на использование частиц, добавляя огни лагеря, и даже перо дыма и пепла от нашего вулкана!

В следующей главе мы будем брать заставу, встраивающую в нашу сцену, и смотреть на то, как мы можем вызвать ее мультипликации, когда игрок приближается к двери. Чтобы сделать это, я представлю Вас письму JavaScript для Unity, и мы возьмем наш первый прыжок в развитие реальных взаимодействий игры.

3

Характеры Игрока

В этой главе мы расширим сценарий острова, который мы создали в предыдущей главе, смотря на постройку характера игрока, который Вы уже добавили к сцене. Сохраненный как prefab (объект шаблона данных) обеспеченный Технологиями Unity как часть Стандартного пакета Активов, этот объект - пример первого перспективного характера игрока человека. Но как делает его комбинацию объектов, и компоненты достигают этого эффекта?

В этой главе мы будем смотреть внутренности этого prefab, смотря, как каждый из компонентов сотрудничает, чтобы создать наш характер игрока. Вы получите свой первый взгляд на scripting в Unity JavaScript. Поскольку мы уже добавили наш prefab к сцене игры, было бы слишком легко продолжиться с развитием и признать, что этот объект только работает. Всякий раз, когда Вы осуществляете любые внешне созданные активы, Вы должны удостовериться, что Вы понимаете, как они работают. Иначе, если что-нибудь будет нуждаться в наладке или идти не так, как надо, то Вы будете в темноте.

С этим в памяти, мы будем смотреть на следующее, чтобы помочь Вам понять, как объединение только несколько объектов и компонентов может создать абсолютный характер:

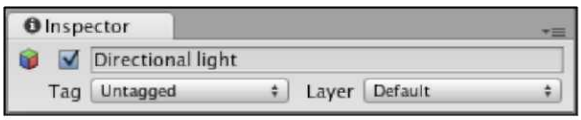
- Признаки, слои, и prefab в **Инспекторе**
- Родительско-детские отношения в объектах
- Основы JavaScript
- Scripting для движения игрока
- Общественное регулирование Variables участника в **Инспекторе**
- Используя камеры, чтобы создать точку зрения

Работа с Инспектором

Поскольку это - наш первый раз, анализируя объект в **Инспекторе, давайте** начнем, смотря на особенности **Инспектора**, которые распространены у всех объектов.

Наверху **Инспектора** Вы будете видеть название объекта, который Вы в настоящее время выбирали, наряду с объектом игры или заранее приготовленным изображением (красный, зеленый, и сине-сторонний куб или светло-голубой куб соответственно) и checkbox, чтобы позволить Вам временно или надолго дезактивировать объект.

Например, недавно создавая объект игры (не от существующего prefab) с нашим **Направленным светом,** вершина **Инспектора** смотрит следующим образом:



Здесь, Вы можете видеть красное, зеленое, и синее изображение, которое представляет стандартный объект игры. Это также стоит отмечать, что коробка названия этой части **Инспектора** может использоваться, чтобы переименовать объект просто, щелкая и печатая.

Ниже изображения, активного checkbox, и названия, Вы будете видеть параметры настройки **Признака** и **Слоя**.

Признаки(Tag)

Признаки - просто ключевые слова, которые могут быть назначены на объект игры. Назначая признак, Вы можете использовать выбранное слово или фразу, чтобы обратиться к объекту игры (или объекты), на который это назначено в пределах Вашей игры scripting (один признак может использоваться много раз). Если Вы работали с Adobe Flash прежде, то Вы могли уподобить признаки понятию *названия случая* во Вспышке - в котором они - ключевые слова, используемые, чтобы представить объекты в сцене.

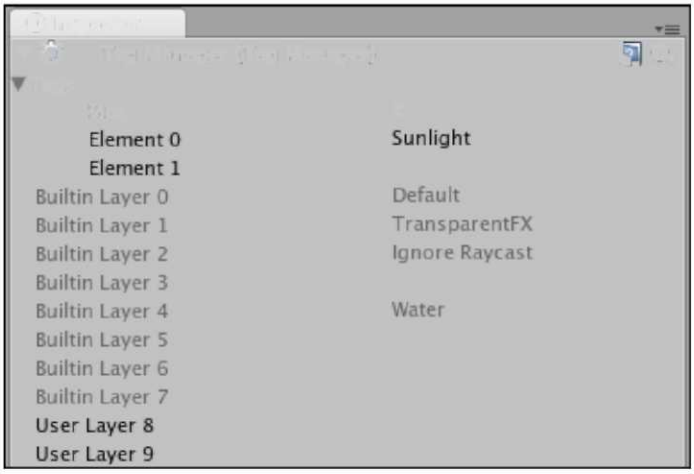
Добавление признака к объекту игры является двухступенчатой процедурой во-первых, Ваш признак должен быть добавлен к списку в пределах **Менеджера Признака,** и затем относиться Ваш выбранный объект.

Чтобы помочь Вам привыкнуть к использованию признаков, давайте сделаем и давайте применим наш первый признак к **Направленному легкому** объекту. С этим объектом, отобранным в группе **Иерархии,** нажмите на **Признак**, опускаются меню, где это в настоящее время **Нет тега.**

Вы будете теперь видеть список существующих признаков. Вы можете или выбрать одного из них или сделать новый. Избранный **Добавляют Признак** в основании, чтобы добавить признак к списку в **Менеджере Признака.**

Группа **Инспектора** теперь переключится на показ **Менеджера Признака,** а не компонентов объекта, который Вы ранее выбрали.

Менеджер Признака показывает и **Признаки** и **Слои.** Чтобы добавить новый признак, Вы должны будете нажать на серую стрелку налево от **Признаков**, чтобы видеть области **Размера** и **Элемента**. Как только Вы сделали это, Вы будете в состоянии напечатать в месте направо от **Элемента 0.**



Напечатайте на название **Солнечный свет** (принимающий во внимание, что признаки можно назвать тем, чему Вы нравитесь), и пресса *Вступает*, чтобы подтвердить. Отметьте, что, как только Вы нажимаете, *Вступают,* приращения ценности **Размера** в следующее число, и добавляет новую область **Элемента** - Вы не должны заполнить это, поскольку это просто доступно для следующего признака, Вы желаете добавить.

Параметр **Размера** здесь - просто количество признаков, в настоящее время настраиваемых. Unity использует систему **Размера** и **Элемента** для многих различных меню, и Вы столкнетесь с этим, поскольку мы продолжаем работать с **Инспектором.**

Вы добавили признак к списку. Однако, это еще не было добавлено к **Направленному легкому** объекту. Поэтому, повторно выберите этот объект в группе **Иерархии**, и затем щелкните еще раз на **Нет тега** рядом с **Признаком** наверху **Инспектора.** Вы будете теперь видеть свой недавно созданный признак под названием **Солнечный свет** у основания Вашего списка признаков. Чтобы применить это, просто выберите это из опускающегося меню, как показано в следующем скриншоте:

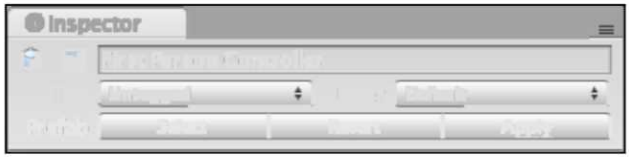
Слои

Слои - дополнительный способ сгруппировать объекты, чтобы применить определенные правила к ним. Они главным образом используются, чтобы сгруппироваться - просят правила предоставления об освещении и камерах. Однако, они могут также использоваться с техникой физики под названием **кастинг Луча**, чтобы выборочно проигнорировать определенные объекты.

Помещая объекты в слой, например, они могут быть отсеяны от параметра **Маски Отбора** света, который исключил бы их из того, чтобы быть затронутым светом. Слои созданы в той же самой манере как

признаки, и доступны от **Менеджера Признака,** замеченный упоминал ниже список **Признаков.**

Prefab и Инспектор



Если активный объект игры, который Вы выбираете из группы **Иерархии**, произойдет из prefab, то Вам покажут некоторые дополнительные параметры настройки.

Ниже областей **Признака** и **Слоя** Вы можете видеть три дополнительных кнопки для того, чтобы взаимодействовать **С prefab** возникновения объекта**.**

- **Выберите(Select):** Это просто определяет местонахождение и выдвигает на первый план prefab, которому принадлежит этот объект в **Проектной** группе.
- **Вернитесь(Revert):** Возвращается любые параметры настройки для компонентов в активном объекте назад к параметрам настройки, используемым в prefab в **Проектной** группе.
- **Обратитесь(Apply):** Изменяет параметры настройки prefab к используемым в в настоящее время отбираемом случае того prefab. Это обновит любые другие копии этого prefab в настоящее время в активной сцене.

Теперь, когда Вы знаете о дополнительных параметрах настройки, доступных на **Инспекторе, давайте** начнем использовать это, чтобы осмотреть наш характер игрока.

Вскрытие противоречия в Первом объекте Диспетчера

Человека

Давайте начнем, смотря на объекты, которые составляют нашего **Первого Диспетчера Человека (FPC)** прежде, чем мы изучим компоненты, которые заставляют его работать.

Нажмите на серую стрелку налево от **Первого Диспетчера Человека** в **Иерархии**, чтобы показать объекты, вложенные внизу. Когда объекты вложены таким образом, мы говорим, что есть родительско-детские отношения. В этом примере **Первый Диспетчер Человека** - *родитель,* в то время как **Графическая** и **Главная Камера** - свои *детские* объекты. В **Иерархии** детские объекты заказаны, чтобы показать их воспитание, как показано в следующем скриншоте:

Родительско-детские проблемы

Считая вложенным или *детскими* объектами, Вы должны отметить, что есть некоторые ключевые правила которые нужно помнить.

Детское положение объекта и ценности вращения относительно их родителя. Это упоминается как **местное положение и местное вращение.** Например, Вы можете рассмотреть свой родительский объект существовать в (500, 35, 500) в мировых координатах, но выбирая ребенка, Вы заметите, что, в то время как его положение (0,0,0), это все еще, кажется, находится в том же самом положении как родитель. Это - то, чем мы подразумеваем *относительно* - помещая детский объект в (0,0,0), мы говорим этому быть в происхождении его родителя, или относительно положения родителя.

В группе **Иерархии**, нажмите **на Графику** ниже родительского объекта FPC, и Вы будете видеть это в действии - FPC находится в центре острова, все же наш **Графический** объект имеет, преобразовывают ценности положения (0,0,0).

В результате этой относительности Вы должны также знать, что всякий раз, когда родительский объект перемещен, его дети будут следовать, поддерживая их местные положения и вращения согласно положению родителя и вращение в мире игры.

First Person Controller

Три части этого объекта следующие:

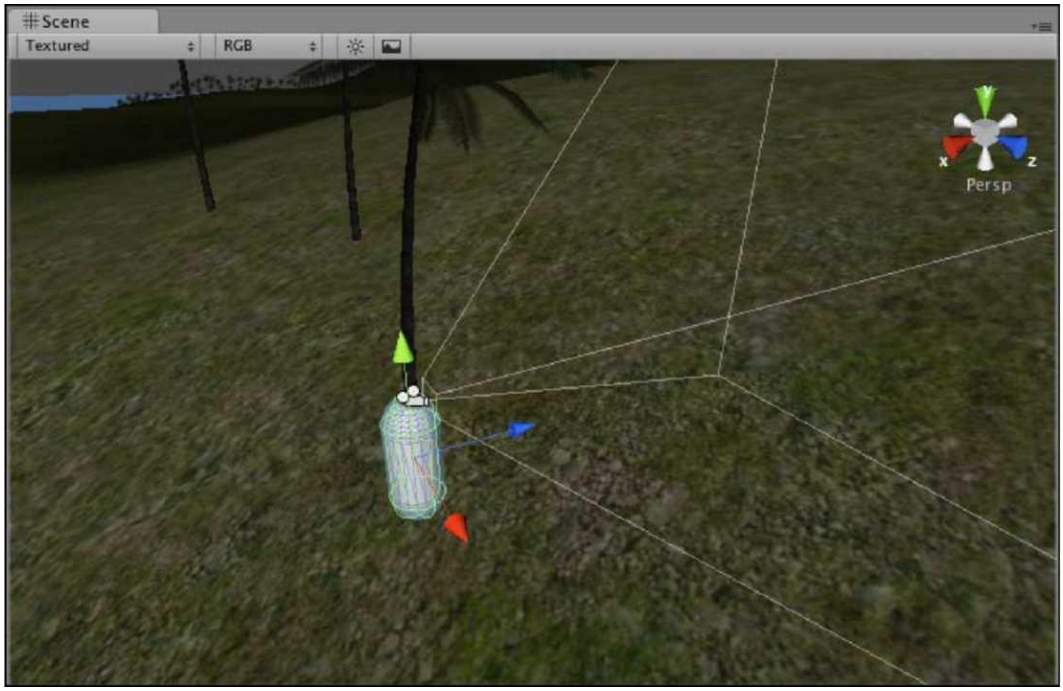
1. **First Person Controller:** объект FPC, или родитель группы, у

которой есть главный scripting и коллайдер, относился к этому и управляет, как это ведет себя. Этому объекту просили подлинник движение, используя клавиатуру и другое вращение разрешения, перемещая левую и правую мышь.

2. **Графика:** Просто примитивная форма Капсулы (показанный в следующем скриншоте), который позволяет Вам как разработчик видеть, куда Вы поместили
   FPC.
3. **Главная Камера:** Помещенный в том, где Вы ожидали бы, что уровень глаз игрока будет. **Главная Камера** должна там обеспечить точку зрения, и имеет примененный scripting, разрешая нам смотреть вверх и вниз.

Принимая во внимание родительско-детские отношения, мы можем сказать, что везде, куда объект FPC перемещается или вращается к, **Графическая** и **Главная Камера** будет следовать.

Выберите **First Person Controller** в группе **Иерархии**, и с Вашим курсором мыши по окну **Scene**, нажмите ключ *F*, чтобы сосредоточить представление относительно того объекта. Вы должны теперь видеть FPC, как показано в следующем скриншоте:

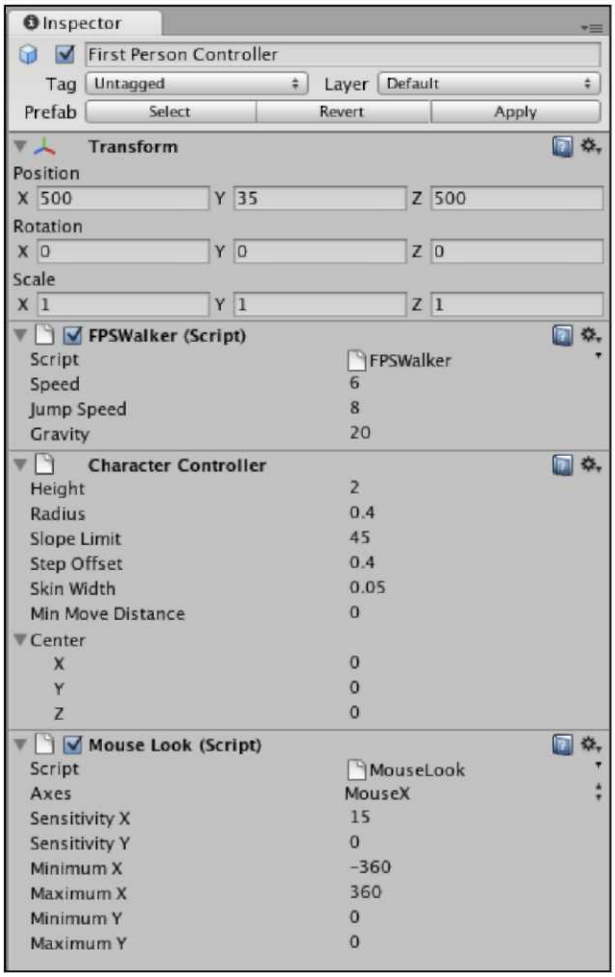


Теперь нажмите кнопку **Play**, щелкните где-нибудь на окне **Game**, и начните перемещать характер, наблюдая это в окне **Scene**. Вы должны заметить, что, поскольку Вы перемещаете характер с ключами и вращаете его с Вашей мышью, детские объекты также перемещаются.

Характеры Игрока

Объект 1: First Person Controller (родитель)

С **First Person Controller**, все еще отобранным в окне **Hierarchy,** смотрите на компоненты, приложенные к этому в **Инспекторе.**



Как показано в предыдущем скриншоте, Вы будете видеть, что есть четыре компонента, составляющие родительский объект FPC - **Преобразовывают, FPSWalker (Подлинник), Диспетчер Характера,** и **Взгляд Мыши (Подлинник).**

Преобразовать

Как со всеми активными объектами игры, у FPC есть компонент **Transform (преобразовать)**, который показывает, и позволяет регулирование его положения, вращения, и масштабных коэффициентов.

FPSWalker (Подлинник)

Этот подлинник, написанный в JavaScript, отвечает за разрешение характера переместиться назад и вперед использование наших **вертикальных ключей оси** (стрелка Стрелки/Вниз или *W/S*), и переместить от стороны к стороне с **горизонтальными ключами оси** (Левая стрелка Стрелки/Права или *A/D*), и подскочить, используя ключ **скачка**, который, по умолчанию, является *Клавишей "пробел"*.

У подлинника есть три **общественных variables участника**, выставленные в **Инспекторе,** разрешая нам приспособить элементы подлинника, даже не открывая это. Они - **Скорость, Скорость Скачка,** и **Сила тяжести.**

В то время как двигатель физики Unity может управлять объектами с компонентом Rigidbody, наш характер не показывает тот. У этого должна быть своя собственная создающая силу тяжести часть подлинника - этот Variables позволяет нам увеличивать свой эффект, в результате заставляя нас упасть более быстро.

> С любым компонентом типа подлинника, первый параметр прежде, чем его общественные variables участника будет **Подлинник.** Это - урегулирование, которое позволяет Вам обменивать это для различного подлинника, не удаляя компонент и добавляя новый.
>
> Это достаточно полезно, но это входит в свое собственное, когда Вы начинаете писать подлинники, которые являются возрастающими усовершенствованиями на Ваших существующих подлинниках - Вы можете просто начать дублировать подлинники, исправить их, и обменять их, выбирая различный из опускаться меню направо от названия подлинника. Делая это, Вы сохраните свои существующие параметры настройки для общественных variables участника в **Инспекторе.**

Character Controller

Этот объект действует как Коллайдер (компонент, дающий наш объект физическое присутствие, которое может взаимодействовать с другими объектами), и определенно разработан, чтобы угодить движению характера и контролю в пределах мира. Это показывает следующие параметры:

- **Высота:** высота характера, который определяет, насколько высокий коллайдер характера формы капсулы будет.
- **Радиус:** Насколько широкий коллайдер формы капсулы будет - у этого есть радиус по умолчанию, который соответствует радиусу **Графического** детского объекта. Однако, если Вы желаете сделать свой характер шире, или ограничить движение или чтобы обнаружить столкновения с большим местом, тогда Вы могли увеличить ценность этого параметра.

- **Наклонный Предел:** Принимая во внимание идущее в гору движение, этот параметр позволяет Вам определять, насколько крутой наклонная поверхность может быть прежде, чем характер больше не может идти по специфической наклонной поверхности. Включением этого параметра **Диспетчер Характера** останавливает игрока от простого хождения по вертикальным стенам или крутым областям земли, которая конечно казалась бы нереалистичной.
- **Погашение Шага:** Поскольку Ваш мир игры может хорошо показать лестницу, этот параметр позволяет Вам определять, как далеко от основания характер может подойти - чем выше ценность, тем больший расстояние они могут подойти.
- **Ширина Кожи:** Поскольку характер столкнется с другими объектами игры в сцене, часто на скорости, этот параметр обеспечен, чтобы позволить Вам определять, как глубоко другие коллайдеры могут пересечься с коллайдером характера без реакции. Это разработано, чтобы помочь уменьшить конфликты с объектами, результатом которых может быть дрожание (небольшая, но постоянная встряска характера) или характер, застревающий в стенах. Это происходит, когда два коллайдера внезапно пересекутся, не позволяя время двигателя игры реагировать соответственно - вместо аварии, двигатель выключит коллайдеры, чтобы остановить контроль или вызвать коллайдеры обособленно неочевидно. Технологии Unity рекомендуют, чтобы Вы установили ширину кожи в 10 процентов параметра радиуса Вашего характера.
- **Расстояние Движения Минуты:** Это - самое низкое количество, которым может быть перемещен Ваш характер. Обычно набор к нолю, устанавливая эту ценность в большее число означает, что Ваш характер не может переместиться, если они не будут перемещены кроме того ценность. Это вообще только используется, чтобы уменьшить дрожание, но в большинстве ситуаций оставлен установленным в **0.**
- **Центр:** Набор как Vector3 (x, y, z ценности). Это позволяет Вам помещать коллайдер характера далеко от его местной центральной точки. Обычно в ноле, это чаще используется для характеров третьего лица, у которых будет более сложный взгляд чем простая капсула. Разрешая Вам переместить координату Y, например это позволяет Вам объяснить, где ноги характера поражают пол как, это **- коллайдер Диспетчера Характера,** который определяет, где объект игрока опирается на основание, а не визуальную петлю ног характера.

Коллайдер Диспетчера Характера представлен в представлении **Сцены** зеленой схемой формы капсулы, таким же образом как другие коллайдеры в Unity. В следующем изображении я имею временно выведенный из строя предоставление визуальной части **Графики** FPC, чтобы помочь Вам определить схему Диспетчера Характера:

Взгляд Мыши (Подлинник)

Написанный в C#, этот подлинник отвечает за превращение нашего характера, поскольку мы перемещаем мышь, оставляя горизонтальные ключи движения, чтобы обращаться с обстрелом. У этого подлинника есть много общественных variables участника, выставленных для регулирования:

- **Топоры:** Набор к **MouseX** в этом случае. Этот Variables позволяет Вам выбирать **MouseX, Мышиный,** или **MouseXAndY.** В нашем FPC мы нуждаемся в этом случае подлинника **Взгляда Мыши** (другой случай, являющийся компонентом **Главного** детского объекта **Камеры**), чтобы быть установленными в Ось X только, поскольку это позволит движение мыши в левом и правом направлении, чтобы вращать весь характер - **Главный** детский включенный объект **Камеры**.
  Вы можете задаваться вопросом, почему у нас просто нет топоров **MouseXAndY** выбранными на **Взгляде Мыши** составляющий для **Главного** объекта **Камеры**. Проблема с этим подходом состояла бы в том, что, в то время как он позволит камере наклонять и готовить в кастрюле в обоих топоры, он не держал бы столкновение характера, где мы смотрим - он просто вращал бы камеру в местном масштабе. В результате мы могли озираться, но когда мы перемещали характер с ключами, мы убежим в случайном направлении. При наличии этого случая подлинника **Взгляда Мыши** на родительском объекте **(Первый Диспетчер Человека),** мы позволяем этому вращать характер, который в свою очередь готовит в кастрюле камеру, потому что это - ребенок и будет соответствовать вращению ее родителя.
- **Чувствительность X/Sensitivity Y:** Поскольку мы только используем Ось X этого подлинника, этот Variables будет только управлять, насколько левое/правильное движение мыши затрагивает вращение объекта. Если бы Variables **Топоров** собирался включать Ось Y, то это управляло бы, насколько чувствительный или вниз движение мыши было при воздействии наклона камеры. Вы заметите, что **Чувствительность Y** была установлена в **0,** поскольку мы не используем ее, в то время как **Чувствительность X** установлена в **15** - чем выше ценность здесь, тем быстрее вращение.

Минимальный X/Maximum X: Это позволяет Вам ограничивать количество, которое Вы можете подвергнуть резкой критике вокруг, поскольку этот подлинник отвечает за превращение характера. Это количество установлено в**-360** и **360** соответственно, разрешая Вам повернуть игрока полностью вокруг на месте.

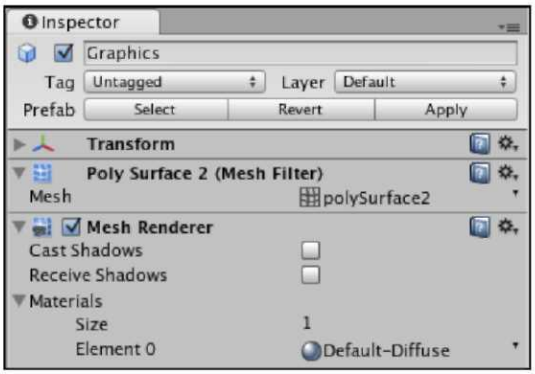
- **Минимум Y/Maximum Y:** Снова, мы не используем Ось Y, таким образом этот Variables установлен в **0** здесь, но обычно это ограничило бы игрока, смотря вверх и вниз, поскольку это

делает в случае этого подлинника, который применен к **Главному** детскому объекту **Камеры**.

Объект 2: Графика

Из-за первой природы человека этого характера игрок никогда не будет видеть их собственное тело, как представлено капсулой. Это просто включено в этот prefab, чтобы помочь Вам, разработчику, легко определить объект в окне **Scene**, развиваясь и проверяя игру. Часто, разработчики выключат предоставление краткой петли, повреждая **Петлю** компонент **Renderer** в **Инспекторе.** Это должно гарантировать, что капсула остается скрытой, особенно в играх, где характер игрока рассматривается от другого угла во время сцен сокращения.

Выберите **Графический** детский объект в группе **Иерархии**, и смотрите на **Инспектора**, чтобы рассмотреть ее компоненты.



Мы возьмем это как читающийся, что у этого объекта есть компонент **Transform (преобразовать)** и что его положение (0,0,0), поскольку это сидит центрально с родительским объектом, **Первым Диспетчером Человека.** Поскольку единственная цель этого объекта состоит в том, чтобы представить нашего игрока визуально, у нее просто есть два ключевых компонента, которые делают ее видимой, **Фильтр Петли** и **Петля Renderer.** Но что делают эти два компонента?

Фильтр петли

Это называют **Поверхностью Poly 2** в этом случае. Название компонента **Фильтра Петли** в любой трехмерной петле обычно - название объекта, который это представляет. Поэтому, когда Вы вводите внешне созданные модели, Вы заметите, что каждый компонент **Фильтра Петли** называют в

честь каждой части модели.

Фильтр Петли - просто компонент, содержащий петлю - трехмерная форма непосредственно. Это тогда работает с renderer, чтобы потянуть поверхность, основанную на петле.

Петля renderer

Петля Renderer должна присутствовать, чтобы потянуть поверхности на петлю трехмерного объекта. Это также отвечает за a), как петля отвечает на освещение и b) материалы, используемые на поверхности, чтобы показать цвет или структуры.

У петли renderers есть следующие параметры:

- **Тени Броска:** заставит Ли легкий бросок на этот объект тень быть брошенной на других поверхностях (только доступный в Unity Про версия).
- **Получите Тени:** Оттянуты ли тени, брошенные другими объектами, на этот объект (только доступный в Unity Про версия).

  В этом примере **Графической** капсулы не поставлен никакой флажок. Во-первых, это - то, потому что мы никогда не будем видеть капсулу, таким образом мы не должны будем получать тени. Во-вторых, игрок не думает, что их характеру сформировали тело как капсула, таким образом видя, что тень формы капсулы после них вокруг выглядела бы довольно странной.
- **Материалы:** Эта область использует систему **Размера/Элемента**, замеченную в **Менеджере Признака** ранее в этой главе. Это позволяет Вам определять один или более материалов и регулировать параметры настройки для них непосредственно, не имея необходимость узнавать материал в использовании, и затем регулировать это отдельно в **Проектной** группе.

Поскольку наш **Графический** объект не требует/показывает никакого цвета или структуры, нет никакого материала к предварительному просмотру, но мы исследуем материальные параметры далее в следующих немногих главах.

Объект 3: Главная Камера

Главная Камера действует как Ваш viewport. В **Первом** prefab **Диспетчера Человека Главная Камера** помещена в уровень глаз (наверху **Графической** капсулы) и управляется подлинниками, разрешая игроку переместить весь родительский объект и камеру независимо. Это позволяет игроку смотреть и также идти вокруг в то же самое время.

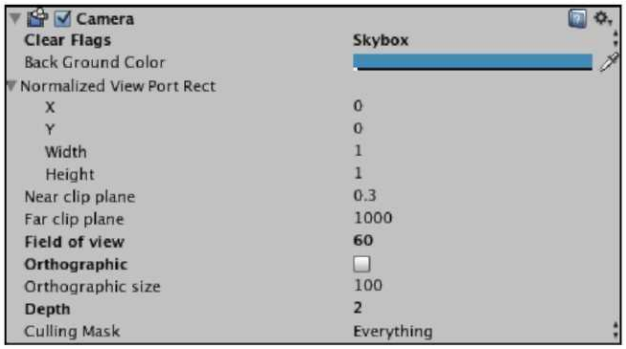
Характеры Игрока

Обычно, объекты игры камеры составлены из трех ключевых компонентов - **Камера, GUILayer,** и **Слой Вспышки,** в дополнение к обычному компоненту **Transform (преобразовать)**. Камеры также идут со **Звуковым Слушателем** для того, чтобы получить звук, но это обычно удаляется, поскольку Unity требует, чтобы Вы только имели один за сцену.

В этом случае у нашей камеры также есть единственный компонент подлинника под названием **Взгляд Мыши (Подлинник),** который обращается с вращением наклона для камеры, основанной на входе от мыши. Понять камеру рассматривает лучше, позволить нам смотреть на то, как основные компоненты работают.

Камера

В то время как обычно сопровождающийся **GUILayer** и компонентами **Слоя Вспышки**, компонент **Камеры** - главный компонент, отвечающий за установление точки зрения. Чтобы понять, как компонент **Камеры** формирует нашу точку зрения, мы исследуем ее параметры.



- **Ясные Флаги:** Обычно это будет установлено в его неплатеж, **Skybox,** чтобы позволить, что камера, чтобы отдать skybox материал в настоящее время относилась к сцене. Но чтобы позволить Вам, разработчику, управлять использованием многократных камер, чтобы потянуть мир игры, **Ясный** параметр **Флагов** существует, чтобы позволить Вам устанавливать определенные камеры, так же как отдавать определенные части мира игры. Однако, маловероятно, что Вы начнете использовать методы, такие как это, пока Вы не схватились с намного большим количеством основ Unity.
- **Обратный Цвет Основания:** второстепенный цвет - цвет, предоставленный позади всех объектов игры, если нет никакого skybox материала, относился к сцене. Нажатие на цветной блок позволит Вам изменять этот цвет, используя цветного сборщика, или Вы можете использовать изображение пипетки чернил направо от цветного блока,

чтобы пробовать цвет от где-нибудь onscreen.

- **Нормализованный Порт Представления Rect:** Этот параметр позволяет Вам определенно заявлять измерения - и положение для - представление камеры. Обычно, это собирается соответствовать всему экрану, и это также верно в примере **Главной Камеры**, приложенной к нашему игроку **[78]**

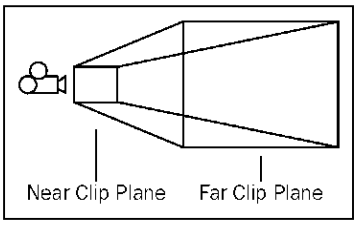
X и координаты **Y**, устанавливаемые в **0** средств, которые наше представление камеры начинает в нижнем левом из экрана. Учитывая то, что ценности **Ширины** и **Высоты** установлены в **1,** наш взгляд от этой камеры заполнит экран, потому что эти ценности находятся в экране Unity, координирует систему, которая колеблется от 0 до 1.

Вы будете видеть эту систему в других 2-ых элементах, что Вы работаете с в Unity, как **графический пользовательский интерфейс (GUI)** элементы.

Преимущество того, чтобы быть способным установить размер и положение нашего viewport состоит в том, что мы можем желать использовать больше чем одну камеру в нашей игре. Например, в мчащейся игре Вы можете желать иметь камеру viewport в главном углу экрана, показывая представление позади автомобиля, позволить игроку разыскивать других водителей, приближающихся к нему.

- **Около обрезают самолет скрепки Самолета / Далекий самолет скрепки: самолеты скрепки** - эффективно расстояния, чтобы потянуть мир игры - близкий самолет, являющийся самым близким расстоянием, чтобы начать тянуть, и далекий самолет, являющийся самым далеким, или скорее где рисунок заканчивается.

  Чтобы экономить на памяти в графическом буфере (часть памяти имела обыкновение хранить информацию на игре visuals в мире), самолеты скрепки часто используются, чтобы отключить пейзаж, который далек от игрока. В старших трехмерных играх эта техника была более очевидной, поскольку у компьютеров назад тогда было меньше RAM, чтобы написать, таким образом больше памяти должно было быть сохранено при использовании более близких далеких самолетов скрепки, чтобы сделать игру управляемой гладко.



- **Область представления:** Этот параметр устанавливает ширину

viewport камеры в степенях. Этот угол установлен в **60** в нашем главном объекте камеры, поскольку это - заметная ценность, чтобы дать эффект человеческого глазного представления.

- **Орфографический** и **Орфографический размер:** Toggling это урегулирование переключил бы Вашу камеру к **орфографическому** представлению, в противоположность стандартному трехмерному представлению, называемому перспективным представлением. Орфографические камеры чаще всего используются в играх, таких как изометрические игры стратегии в реальном времени, или 2-ые игры.

- **Глубина:** урегулирование **Глубины** может быть использовано, используя многократные камеры. При наличии многих камер, которые переключены между использованием подлинников, параметр **Глубины** позволяет Вам определять первоочередной заказ, то есть, камера с более высокой ценностью глубины отдаст перед камерой с более низкой ценностью глубины. Глубина может также работать в соединении с **Нормализованным Портом Представления Rect**, приводящий в порядок, чтобы позволить Вам помещать представления камеры по главному представлению мира игры. В примере зеркала заднего обзора мчащейся игры камера заднего обзора должна была бы иметь более высокую ценность глубины чем главная камера, чтобы быть предоставленной перед главным передовым представлением мира игры.
- **Отбор Маски:** Этот параметр работает со **Слоями** игры Unity**,** как обсуждено ранее, чтобы позволить Вам отдавать выборочно. Помещая определенные элементы Вашей игры на специфическом слое, Вы можете отсеять слой, они идут от Маски Отбора, опускаются меню, чтобы исключить их из того, чтобы быть предоставленным камерой. В настоящее время наша **Главная Камера** собирается отдать все, поскольку это - единственная камера в игре.

GUILayer и Слой Вспышки

Эти два компонента не имеют никаких параметров, но просто позволяют отдавать дополнительных визуальных элементов. **GUILayer** позволяет предоставление 2-ых элементов, таких как меню и в игре **настороженные показы (КОЖУРА). Слой Вспышки** позволяет камере отдавать вспышки освещения, такие как тот, который мы добавили к нашему **Направленному свету** в Главе 2.

Взгляд Мыши (Подлинник)

Это - второй случай **Взгляда Мыши (Подлинник)** компонент, который мы видели пока - другой находящийся на родительском **Первом**

Диспетчере Человека. Однако, на сей раз его параметры **Топоров** установлены к **Мышиному,** и в результате это просто отвечает за взгляд вверх и вниз, вращая главный объект камеры вокруг его Оси X. Это, объединенное с подлинником **Взгляда Мыши**, который вращает родительский объект, дает нам эффект полностью свободного взгляда, используя мышь, потому что, вращая родителя возражают, что наша главная камера вращается левая и правая также.

Звуковой слушатель

Звуковой слушатель действует как наши уши и позволяет нам слышать любые звуковые источники, размещенные в игру. При наличии звукового слушателя на главной камере в первом человеке рассматривают игру, моно (единственный канал) звуки, которые происходят налево от игрока, услышат в левом ухе, разрешая Вам создать окружающую среду с погружением стерео реального мира.

Теперь, когда мы исследовали составные части **Первого Диспетчера Человека, давайте** бросим наш первый взгляд на то, как подлинник **FPSWalker** работает, чтобы схватиться с Unity JavaScript.

Основы Scripting

Scripting - один из наиболее ключевых элементов в становлении разработчиком игр. В то время как Unity является фантастическим при разрешении Вам создать игры с минимальным знанием исходного текста двигателя игры, Вы все еще должны будете понять, как написать кодекс, который командует двигателем Unity. Код, написанный для использования с Unity, догоняет ряд сборных **classes,** о которых Вы должны думать как библиотеки инструкций или *поведений*. При письме подлинников Вы создадите свои собственные classes, догоняя команды в существующем двигателе Unity.

В этой книге мы прежде всего сосредоточимся на том, чтобы писать подлинники в JavaScript Unity, поскольку это - самый легкий язык, чтобы начать с и является главным центром scripting справочной документации Unity. В то время как эта книга представит Вас основам scripting, чрезвычайно рекомендуется, чтобы Вы прочитали Unity *Ссылка Scripting* параллельно на это, и это доступно как часть Вашей установки Unity, и также онлайн в:
<http://unity3d.com/support/documentation/ScriptReference/>

Проблемы столкнулись, в то время как scripting может часто решаться в отношении этого, и если это не работает, то Вы можете попросить помощь на форумах Unity или в **интернет-Чате (IRC)** канал для программного обеспечения. За дополнительной информацией, посещение:
<http://unity3d.com/support/community>

Сочиняя новый подлинник или используя существующий, подлинник станет активным только когда приложено к объекту игры в текущей

сцене. Прилагая подлинник к объекту игры, это становится компонентом того объекта, и поведение, написанное в подлиннике, может относиться к тому объекту. Однако, подлинники не ограничены запросу (термин scripting для *формирования* или *управления*), поведение на игре возражает, что они принадлежат, поскольку на другие объекты могут сослаться название или признак и могут также быть даны инструкции.

Чтобы получить лучшее понимание подлинников, мы собираемся исследовать, позволить нам смотреть на некоторое ядро scripting принципы.

Команды

Команды - инструкции, данные в подлиннике. Хотя команды могут быть свободными в подлиннике, Вы должны попробовать и гарантировать, что они содержатся в пределах Functions, чтобы дать Вам больше контроля, когда их вызывают. Все команды в JavaScript должны быть закончены (законченные) с точкой с запятой следующим образом:

```
speed = 5;
```

Variables

Variables - просто контейнеры для информации. Они объявлены (настроенным) использованием **вара** слова, сопровождаемого словом или нераздельной фразой. Variables можно назвать чем-нибудь, что Вы любите, при условии, что:

- Название не находится в противоречии с существующим словом в коде двигателя Unity.
- Это содержит только алфавитно-цифровые характеры и подчеркивает и не начинается с числа. Например, слово *преобразовывают*, уже существует, чтобы представить компонент Transform (преобразовать) объекта игры, таким образом называя Variables, или Functions с тем словом вызвал бы конфликт.

Variables могут содержать текст, числа, и ссылки на объекты, активы, или компоненты. Вот пример декларации Variables:

```
var speed = 9.0;
```

Наш Variables начинается с `вара` слова, сопровождаемого его именем, `скоростью`. Это тогда установлено (данный ценность), использование сингла равняется символу и закончено как любая другая команда с точкой с запятой.

Типы данных Variables

Объявляя variables, Вы должны также сообщить, какую информацию они будут хранить, определяя **тип данных.** Используя наш предыдущий пример, вот та же самая декларация Variables включая ее тип данных:

```
var speed :  float = 9.0,-
```

Прежде, чем Variables установлен, мы используем двоеточие, чтобы определить тип данных. В этом примере ценность, в которую устанавливается Variables, является числом с десятичным разрядом, и мы заявляем, что у этого должно быть плавание типа данных (короткий для 'плавающей запятой' - значение числа с десятичным разрядом).

Определяя тип данных для любого Variables мы объявляем, мы в состоянии сделать наше scripting более эффективное, поскольку двигатель игры не должен решать на соответствующий тип для Variables, это читает. Вот несколько общих типов данных, с которыми Вы столкнетесь, начинаясь scripting с Unity:

- **последовательность:** комбинация текста и чисел сохранена в кавычках

  "как это"
- **int:** Короткий для целого числа, означая целое число без десятичного разряда
- **плавание:** плавающая запятая или десятичное число поместили числовое значение
- **булевый:** истинная или ложная ценность, обычно используемая как выключатель
- **Vector3:** Ряд ценностей XYZ

Используя variables

После объявления variables информация, которую они содержат, может тогда быть восстановлена или установлена, просто при использовании имени переменной. Например, если бы мы пытались установить Variables скорости, то тогда мы просто сказали бы:

```
speed = 2 0;
```

Поскольку мы уже установили (или объявили), вар Variables только используется в декларации, как тип данных.

Мы можем также подвергнуть сомнению или использовать ценность Variables в частях нашего подлинника. Например, если бы мы желали сохранить ценность, которая была половиной текущего Variables скорости, то тогда мы могли установить новый Variables, как показано в следующем примере:

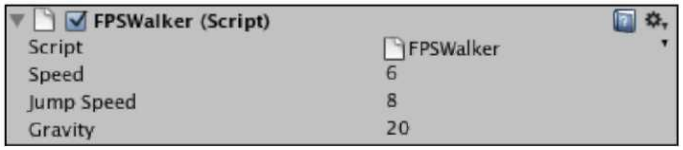
```
var speed : float = 9.0;
var halfSpeed : float;
halfSpeed = speed/2;
```

Кроме того, заметьте, что, где Variables полускорости объявлен, он

не установлен в ценность. Это - то, потому что ценность дана этому в команде ниже этого, деля существующую ценность Variables скорости два.

Общественность против частного

Variables, которые объявлены за пределами Functions в подлиннике, известны как **общественные variables участника,** потому что они автоматически обнаружатся как параметры подлинника, когда он будет рассмотрен как компонент в **Инспекторе.** Например, рассматривая подлинник **FPSWalker**, приложенный как компонент **Первого** объекта **Диспетчера Человека**, мы видим его **Скорость** variables участника**, Скорость Скачка,** и **Силу тяжести:**



Ценность каждого из этих variables может тогда быть приспособлена в **Инспекторе.**

Если Вы вряд ли приспособите ценность Variables в **Инспекторе,** то Вы должны скрыть ее при использовании `частной` приставки**.** Например, если бы мы желали скрыть **Скорость Скачка** Variables**, то** тогда мы приспособили бы декларации Variables в подлиннике следующим образом:

```
var speed = 6.0; private var
jumpSpeed = 8.0; var gravity
= 2 0.0;
```

Это сделало бы наше представление **Инспектора** взгляда подлинника следующим образом:

Знайте, что любая ценность, приспособленная в **Инспекторе**, отвергнет оригинальную ценность, данную Variables в пределах подлинника. Это не будет переписывать ценность, заявил в подлиннике, но просто заменяет это, когда игра работает. Вы можете также вернуться к ценностям, объявленным в подлиннике, нажимая на изображение Винтика направо от компонента и выбирая **Сброс** из опускаться меню, которое появляется.

Functions

Functions могут быть описаны как наборы инструкций, которые можно назвать в определенном пункте во времени выполнения игры. Подлинник может содержать много functions, и каждый может вызвать любую функцию в пределах того же самого подлинника. В Unity scripting, есть много встроенных functions, готовых сделаны

выполнить Ваши команды в предопределенных пунктах вовремя, или в результате пользователя вводит. Вы можете также написать свои собственные functions и назвать их, когда Вы должны выполнить определенные наборы инструкций.

Все functions написаны, определяя название Functions, сопровождаемое рядом скобок, в которые разработчик может передать дополнительные параметры и открыться и согласиться с вьющимися скобами. Давайте смотреть на некоторые примеры существующих functions, которые Вы можете использовать:

Update ()

Новый файл JavaScript, созданный в Unity, начинается с **Update ()** function, который похож на это:

```
function Update(){ }
```

Все инструкции или *команды* для Functions должны произойти между открытием и закрытием скоб, например:

```
function Update(){
  speed = 5;
}
```

Пока команды написаны в пределах вьющихся скоб, их вызовут всякий раз, когда эта функция вызвана двигателем игры.

Игры, которыми управляют в определенном числе **структур в секунду (FPS),** и `Update ()` function, называют, когда каждая структура игры предоставлена. В результате это главным образом используется для любых команд, которые должны постоянно выполняться или для того, чтобы обнаружить изменения в мире игры, которые происходят в режиме реального времени, такие как входные нажатия клавиш команд или щелчки мыши. Имея дело со связанными с физикой командами, альтернативный Functions `FixedUpdate ()` должен использоваться вместо этого, поскольку это - Functions, который держит в синхронизации с двигателем физики, тогда как `Update ()` непосредственно может быть Variables в зависимости от нормы структуры.

OnMouseDown ()

Поскольку пример функции, которая вызвана в определенное время, **OnMouseDown ()** только называют, когда мышь игрока нажимает на объект игры или на элементе GUI в игре.

Это чаще всего используется для управляемых игр мыши или обнаруживающих щелчков в меню. Вот является пример основного `OnMouseDown ()` Functions, который когда приложено к объекту игры, оставит игру, когда на объект нажимают:

```
function OnMouseDown(){
```

```
    Application.Quit();
}
```

Письмо functions

В создании Ваших собственных functions Вы будете в состоянии определить ряд инструкций, которые можно вызвать отовсюду в пределах подлинников, которые Вы пишете. Если мы настраивали некоторые инструкции переместить объект в указанное положение в мире игры, то мы можем написать таможенный Functions, чтобы выполнить необходимые инструкции так, чтобы мы могли вызвать эту функцию от других functions в пределах подлинника.

Например, при падении в ловушку, характер игрока, возможно, должен быть перемещен, если они умерли и возвращаются к началу уровня. Вместо того, чтобы писать инструкции переселения игрока на каждую часть игры, которая заставляет игрока умирать, необходимые инструкции, могут содержаться в пределах единственной функции, которая вызвана много раз. Это могло быть похожим на это:

```
function PlayerDeath(){
 transform.position =
 Vector3(0,10,50); score-=50;
 lives-- ;
}
```

Называя этот `PlayerDeath ()` Functions, все три команды будут нести - положение игрока установлено в X = 0, Y = 10, и Z = 50. Тогда ценность `5 0` вычтена из Variables, названного `счетом`, и 1 вычтен из Variables, названного `жизнями` - два минус символы, замеченные здесь скупой, вычитают 1, и аналогично если Вы сталкиваетесь `c ++` в подлиннике, тогда это просто означает, добавляют 1.

Чтобы вызвать эту функцию, мы просто написали бы следующий как часть нашего подлинника, который заставил игрока умирать:

```
PlayerDeath ();
```

Если еще утверждения

Если утверждение используется в scripting, чтобы проверить на условия. Если его условия соблюдают, то, `если` утверждение выполнит ряд вложенных инструкций. Если они не, то это **еще** может не выполнить своих обязательств к ряду инструкций, названных**.** В следующем примере, `если` утверждение проверяет, установлена ли `основанная логическая переменная в истинный`:

```
var grounded : boolean = false;
var speed  : float; function
Update(){ if(grounded==true){
speed = 5;
 }
```

```
}
```

Если условие в, если скобки утверждения встречен, то есть, если основанный Variables становится верным, то Variables скорости будет установлен в 5. Иначе этому не дадут ценность.

Отметьте, что, устанавливая Variables, сингл равняется символу '=', используется, но проверяя статус Variables, мы используем два '= ='. Это известно, поскольку **сравнительное равняется** - мы сравниваем информацию, хранившую в Variables с ценностью после того, как эти два равняются символам.

Если бы мы хотели настроить условие отступления, то мы еще могли добавить утверждение после, если, который является нашим способом сказать, что, если эти условия не соблюдают, то сделайте что - то еще:

```
var grounded : boolean = false; var
speed  : float; function Update(){
if(grounded==true){ speed = 5;
  }
  else{
    speed = 0;
  }
}
```

Так если не основано верно, скорость будет равняться 0.

Чтобы построить дополнительные потенциальные результаты для того, чтобы проверить условия, мы еще еще можем использовать если перед отступлением. Если мы проверяем ценности, то мы могли написать:

```
if(speed >= 6){ //do
  something
}
else
  if(speed >= 3){
    //do something different
  }
  else{
    //если ни одно из вышеупомянутого не верно, сделайте это
  }
```

Убедитесь, что помнили, что, где я написал//выше, я просто пишу

кодекс, который появился бы как комментарий (невыполненный кодекс).

Многократные условия

Мы можем также проверить на больше чем единственное условие в том `если` утверждение при использовании двух символов амперсанда - **&&**.

Например, мы можем хотеть проверить при условии двух variables сразу и только выполнить наши инструкции, если ценности Variables как определены. Мы написали бы:

```
if(speed >= 3 && grounded == true){
  //do something
}
```

Если мы желали проверить на одно условие или другой, то мы можем использовать два вертикальных выровненных характера '**||**', чтобы означать **ИЛИ.** Мы написали бы этому как:

```
if(speed >= 3 || grounded == true){
  //do something
}
```

Globals и точечный синтаксис

В этой секции мы будем смотреть на путь, которым Вы можете послать информацию между подлинниками, используя кое-что названное **глобальный Variables,** и как при использовании техники письма кодекса, известного как **Точечный Синтаксис**, Вы можете обратиться к информации в иерархической манере.

Используя статичный, чтобы определить globals

Сочиняя подлинники в JavaScript, Ваше название подлинника - название Вашего Classes. Это означает, что, обращаясь к Variables или functions в других подлинниках, Вы можете обратиться к Classes при использовании названия подлинника, сопровождаемого Variables, или функционировать название, используя точечный синтаксис.

Глобальное в условиях scripting может быть определено как ценность или команда, доступная любым подлинником и не просто functions или variables в пределах того же самого подлинника. Чтобы объявить глобальное в пределах Unity файлом JavaScript, `статичная` приставка используется. Например:

```
static var speed :  float = 9.0;
```

Если бы этот Variables `скорости` был одним замеченным в нашем подлиннике **FPSWalker**, то мы были бы в состоянии получить доступ или установить ценность этого Variables от отдельного подлинника, используя название подлинника, точку, и затем название Variables, сопровождаемого, устанавливая ценность следующим образом:

```
FPSWalker.speed = 15.0;
```

Мы именуем этот тип Variables как глобальный Variables, но у нас могут также быть глобальные функции, которые могут быть вызваны от других подлинников. Например, `PlayerDeath ()` Functions, упомянутый ранее, мог быть статическим Functions, который позволит Вам называть это от любого другого подлинника в игре.

Точечный синтаксис

В предыдущем примере мы использовали технику подлинника, названную точечным синтаксисом. Это - термин, который может казаться более сложным, чем это фактически, поскольку это просто означает использовать точку (или точка / период), чтобы отделить элементы, Вы обращаетесь в иерархическом заказе.

Начинаясь с самого общего элемента или ссылки, Вы можете использовать точечный синтаксис, чтобы сузить к определенным параметрам, которые Вы желаете установить.

Например, чтобы установить вертикальное положение объекта игры, мы должны были бы изменить его координату Y. Поэтому, мы надеялись бы обращаться к параметрам положения компонента Transform (преобразовать) объекта. Мы написали бы:

```
transform.position.y = 50;
```

Это означает, что на любой параметр можно сослаться, просто находя компонент, которому он принадлежит при использовании точечного синтаксиса.

Комментарии

Во многих предписьменных подлинниках Вы заметите, что есть некоторые линии, написанные с двумя передовыми разрезами, предварительно устанавливающими их. Они - просто комментарии, написанные автором подлинника. Это - вообще хорошая идея, особенно когда начинающийся с scripting, чтобы написать Ваши собственные комментарии, чтобы напомнить себе, как работает подлинник.

Далее чтение

Поскольку Вы продолжаете работать с Unity, крайне важно, что Вы привыкаете к обращению к *Справочной* документации *Scripting*, которая установлена с Вашей копией Unity и также доступная на вебсайте Unity в следующем адресе: <http://unity3d.com/support/documentation/ScriptReference/>

Вы можете использовать scripting ссылку, чтобы искать правильное использование любого Classes в двигателе Unity. Теперь, когда Вы схватились с основами scripting, давайте смотреть на подлинник

FPSWalker, который отвечает за перемещение нашего **Первого** характера **Диспетчера Человека**.

Подлинник FPSWalker

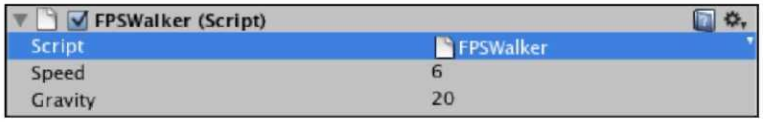
Чтобы рассмотреть любой подлинник в Unity, Вы должны будете открыть это в редакторе подлинника. Чтобы сделать это, просто выберите подлинник в **Проектной** группе, и затем щелкните два раза на ее изображении, чтобы открыть это для того, чтобы отредактировать.

На Mac ушел в спешке неплатеж, редактора подлинника называют Unitron. На PC редактора подлинника называют Uniscite. Они - автономные заявления, которые просто позволяют Вам редактировать различные форматы текстового файла, JavaScript и C# быть двумя такими примерами.

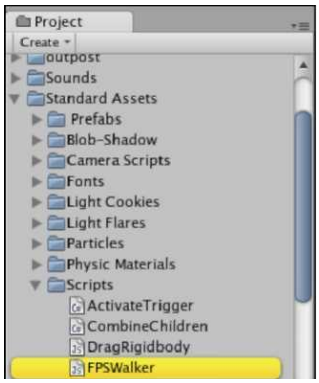
Есть другие свободные редакторы текста, доступные, который Вы можете хотеть использовать при письме подлинников для Unity. Вы можете заставить Unity использовать редактора текста Вашего выбора. В целях этой книги мы обратимся к редакторам подлинника Унитрону по умолчанию и Uniscite.

Запуск подлинника

Выберите **Первый** объект **Диспетчера Человека** в **Иерархии,** и затем нажмите на название файла подлинника под **FPSWalker (Подлинник)** компонент в **Инспекторе** так, чтобы это было выдвинуто на первый план в синем, как показано в следующем скриншоте:

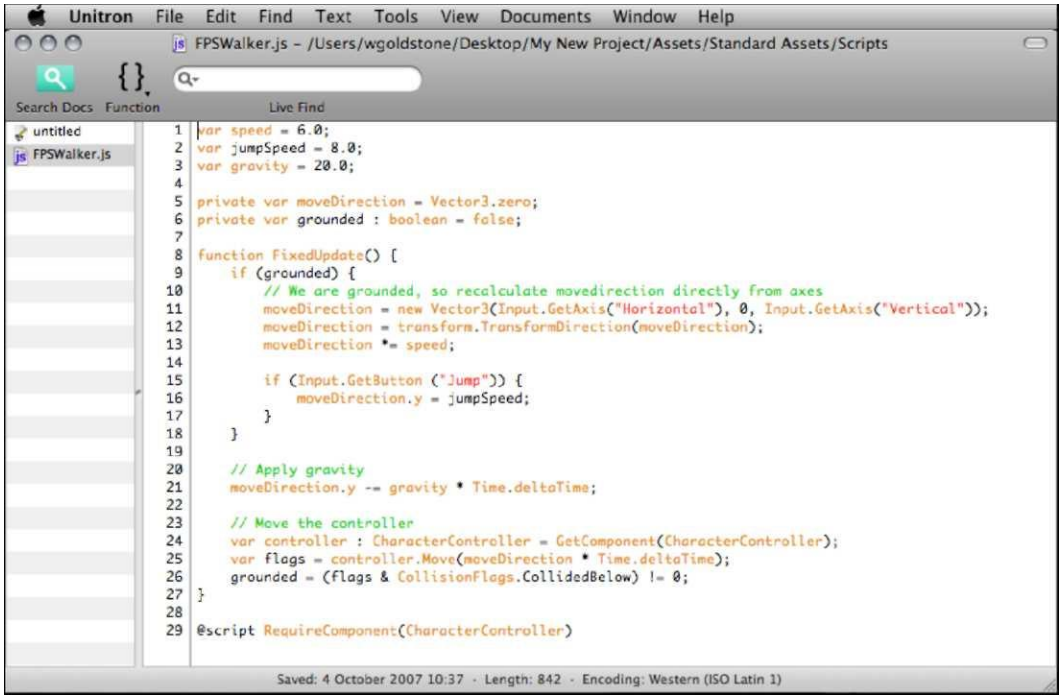


Это выдвинет на первый план местоположение подлинника в **Проектной** группе в желтом (см. следующий скриншот), помочь Вы находить это. Как только Вы определили это, щелкните два раза на изображении файла, чтобы начать это в редакторе подлинника по умолчанию.



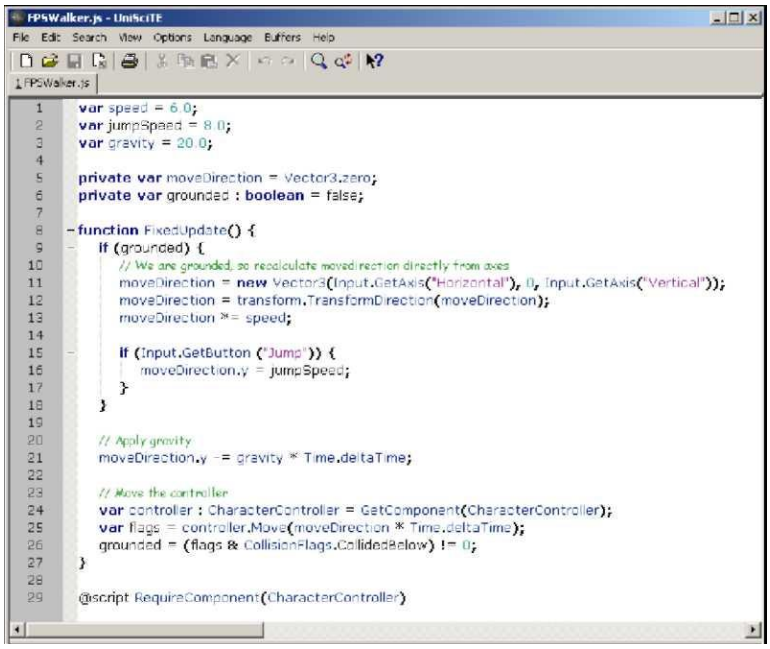
Mac-FPSWalker в Unitron

На Mac **FPSWalker** открывается в редакторе подлинника Унитроне по умолчанию, который появляется как показано в следующем скриншоте:



Windows PC-FPSWalker в Uniscite

На Windows PC **FPSWalker** открывается в редакторе подлинника Унискайте по умолчанию, который появляется как показано в следующем скриншоте:



В то время как это - тот же самый подлинник, Вы будете видеть различные цвета синтаксиса для различных частей подлинника между двумя различными платформами. Это до редактора подлинника непосредственно и не имеет никакого опирания на функциональные возможности scripting.

Вскрытие противоречия в подлиннике

В этой секции мы будем смотреть на части потенциального подлинника, который мы только что изучили в контексте, вскрывая противоречия в их использовании в подлиннике **FPSWalker**.

Декларация Variables

Как с большинством подлинников, **FPSWalker** начинается с ряда деклараций Variables от линий 1 - 6:

```
var speed = 6.0; private var
jumpSpeed = 8.0; var gravity =
2 0.0;
private var moveDirection =
Vector3.zero; private var grounded :
boolean = false;
```

Линии 1 - 3 являются общественными variables участника, используемыми позже в подлиннике как ценности, чтобы умножиться. У них есть десятичные разряды в их числах, таким образом они идеально показали бы набор типов данных, чтобы плавать (поскольку это - простой подлинник в качестве примера от Технологий Unity, не, все variables - напечатанные данные). Линии 5 и 6 являются частными variables, поскольку они будут только использоваться в пределах

подлинника.

Частный Variables `moveDirection` отвечает за хранение текущего передового руководства игрока как Vector3 (набор X, Y, Z координаты). На декларации этот Variables установлен в (0,0,0), чтобы мешать игроку стоять перед произвольным руководством, когда игра начинается.

Частный `основанный` Variables является данными, напечатанными к булевому `(истинный` или `ложный)` тип данных. Это используется позже в подлиннике, чтобы отследить то, является ли игрок опорой на основание, чтобы позволить движение и скачок, который не был бы позволен, если они не были на основании (то есть, если они в настоящее время подскакивают).

Хранить информацию движения

Подлинник продвигается к линии 8 с открытием `FixedUpdate ()` Functions. Подобный `Update ()` function, обсужденному ранее, неподвижное обновление называют каждой неподвижной структурой framerate - это означает, что это является более соответствующим для того, чтобы иметь дело со связанным с физикой scripting, таким как использование rigidbody и эффекты силы тяжести, поскольку стандартный `Update ()` будет меняться в зависимости от нормы структуры игры, зависящей от аппаратных средств.

`FixedUpdate ()` функционируют пробеги от линий 8 - 27, таким образом мы можем предположить, что как все команды и `если` утверждения в пределах этого, они будут проверены после каждой структуры.

> В книге Вы можете иногда сталкиваться с единственной линией кодекса, появляющегося на двух различных линиях. Пожалуйста отметьте, что это было сделано только **я** с целью углубления и должно сделать интервалы между ограничениями. Когда использование такого кодекса удостоверяется, что это находится на одной линии в Вашем файле подлинника.

Первое, `если` утверждение в пробегах Functions от линий 9 - 18 (комментарий разработчика был удален в следующем кодовом отрывке):

```
if (grounded) {
 moveDirection = new
 Vector3(Input.GetAxis("Horizontal"), 0,
  Input.GetAxis("Vertical")); moveDirection =
 transform.TransformDirection(moveDirection);
 moveDirection *= speed;
 if   (Input.GetButton
  ("Jump")) { moveDirection.y =
  jumpSpeed;
```

```
}
```

Заявляя, что его команды и вложенный, если утверждения (линия 15) будут только бежать если (основано), это - стенография для того, чтобы написать:

```
If(grounded == true){
```

Когда основано становится верным, это, если утверждение делает три вещи с Variables moveDirection.

Во-первых, это назначает этому новую ценность Vector3, и помещает текущую ценность Входа. GetAxis (Горизонтальный) к этим X координатам и Входу. GetAxis (Вертикальный) к координате Z, уезжая Y набор к 0:

```
moveDirection = new
  Vector3(Input.GetAxis("Horizontal"), 0,
  Input.GetAxis("Vertical"));
```

В этом примере Технологии Unity написали кодекс, используя слово, 'новое' для приставки Vector3, который они кормят moveDirection Variables. Это - соглашение C# кодирование и было написано таким образом, чтобы сделать их кодекс легче преобразовать между JavaScript и C#. Однако, это не необходимо в JavaScript, который является, почему Вы не будете видеть использование 'новых' в других случаях как это в пределах книги.

Но что является Входом. Команды GetAxis, делающие? Они просто представляют ценности между-1 и 1 согласно горизонтальным и вертикальным входным ключам, которые по умолчанию являются:

- *A/D* или Левая Стрелка/Право горизонтальная стрелкой ось
- W/*S* или Стрелка/Вниз вертикальная стрелкой ось

Когда никакие ключи не будут нажаты, эти ценности будут 0, поскольку они - основанные на оси входы, которым Unity автоматически дает 'праздное' государство. Поэтому, держа клавишу курсора "влево", например, ценность Входа. (Горизонтальный) GetAxis был бы равен-1, держа Правильную стрелку, это будет равно 1, и выпуская любой ключ, ценность рассчитает назад к 0.

Короче говоря, линия:

```
moveDirection = new Vector3(Input.GetAxis("Horizontal"),
  0, Input.GetAxis("Vertical"));
```

дает Variables moveDirection ценность Vector3 с X и ценности Z, основанные на нажатиях клавиш, оставляя набор значений Y 0.

Затем, наш moveDirection Variables изменен снова на линии 12:

```
moveDirection =
transform.TransformDirection(moveDirection);
```

Здесь, мы устанавливаем `moveDirection` в ценность, основанную на `TransformDirection` компонента Transform (преобразовать). Команда **Руководства Transform (преобразовать)** преобразовывает местные ценности XYZ в мировые ценности. Так в этой линии, мы берем ранее набор координаты XYZ `moveDirection` и преобразовываем их в ряд мировых координат. Это - то, почему мы видим `moveDirection` в скобках после `TransformDirection`, потому что это использует набор значений на предыдущей линии и эффективно только изменяет ее формат.

Наконец, `moveDirection` умножен Variables `скорости` на линии 13:

```
moveDirection *= speed;
```

Поскольку `скорость` - Variables участника, умножая ценности XYZ `moveDirection` ею будет означать, что, когда мы увеличиваем ценность `скорости` в **Инспекторе,** мы можем увеличить скорость движения нашего характера, не редактируя подлинник. Это - то, потому что это - проистекающая ценность `moveDirection`, который используется позже в подлиннике, чтобы переместить характер.

Перед нашим, `если (основанное)` утверждение заканчивается, есть, другой гнездился `если` утверждение от линий 15 - 17:

```
if   (Input.GetButton  ("Jump"))
  { moveDirection.y = jumpSpeed;
}
```

Это, `если` утверждение вызвано нажатием клавиши с названием `Скачок`. По умолчанию, кнопка скачка назначена на *Клавишу "пробел"*. Как только этот ключ нажат, ценность Оси Y `moveDirection` Variables установлена в ценность Variables `jumpSpeed`. Поэтому, если `jumpSpeed` не был изменен в **Инспекторе,** `moveDirection.y` будет установлен в ценность `8.0`.

Когда мы переместим характер позже в подлинник, это внезапное дополнение от `0` до `8.0` в Оси Y даст эффект скачка. Но как наш характер возвратится к основанию? Этому объекту характера не прилагали компонент Rigidbody, таким образом он не будет управляться силой тяжести в двигателе физики.

Это - то, почему мы нуждаемся в линии 21, на котором мы вычитаем из ценности

```
moveDirection.y: moveDirection.y -= gravity *
Time.deltaTime;
```

Вы заметите, что мы просто не вычитаем ценность силы тяжести здесь, как результат выполнения, которое не дало бы эффект скачка, но вместо этого брать нас прямо вверх и вниз снова между двумя структурами. Мы вычитаем сумму Variables `силы тяжести`, умноженного командой по имени `Time.deltaTime`.

Умножая любую ценность в пределах `Update ()` или `FixedUpdate ()` функционируют `Time.deltaTime`, Вы отвергаете основанную на структуре природу Functions и преобразовываете эффект Вашей команды в секунды. Так при письме:

```
moveDirection.y -= gravity * Time.deltaTime;
```

мы фактически вычитаем ценность силы тяжести каждую секунду, а не каждую структуру.

Перемещение характера

Как комментарий намечают 23 пункта, линии 24 - 26 отвечают за движение характера.

Во-первых, на линии 24, новый Variables назвал `диспетчера`, установлен и дан тип данных `CharacterController`. Это тогда собирается представить компонент `Диспетчера Характера`, при использовании `GetComponent ()` команда:

```
var controller  :
 CharacterController =
 GetComponent(CharacterController);
```

При использовании `GetComponent ()`, Вы можете получить доступ к любому компоненту, который присоединен к объекту, к которому Ваш подлинник присоединен, просто при использовании его имени в пределах скобок.

Так теперь, всякий раз, когда мы используем справочного `диспетчера` Variables, мы можем получить доступ к любому из параметров того компонента и использовать Functions `Движения`, чтобы переместить объект.

На линии 25, это точно, что мы делаем. Поскольку мы делаем так, мы помещаем это движение в Variables, названный `флагами` как показано:

```
var flags = controller.Move(moveDirection *
Time.deltaTime);
```

`CharacterController`. Functions `движения` ожидает быть переданным ценность Vector3 - чтобы переместить диспетчера характера в направлениях X, Y, и Z-so, мы используем данные, которые мы хранили ранее в нашем `moveDirection` Variables и умножаем `Time.deltaTime` так, чтобы мы двинулись в метры в секунду, а не метры за структуру.

Проверка

Нашему `moveDirection` Variables только дают ценность, если `основанная` логическая переменная установлена в `истинный`. Так, как мы решаем, основаны ли мы или нет?

Коллайдеры Диспетчера Характера, как любые другие коллайдеры, могут обнаружить столкновения с другими объектами. Однако, в отличие от стандартных коллайдеров, у коллайдера диспетчера характера есть четыре определенных сокращения столкновения, настроенные в ряде респондентов по имени `CollisionFlags`. Они следующие:

- Ни один None
- Стороны Sides
- Выше Above
- Ниже Below

Они отвечают за проверку столкновения, с определенной частью коллайдера, который они описывают - за исключением Ни одного, который просто означает, что никакое столкновение не происходит.

Эти флаги используются, чтобы установить наш `основанный` Variables на линии 26:

```
grounded =   (flags & CollisionFlags.CollidedBelow)   !=
0;
```

Это может выглядеть сложным из-за кратного числа, равняется символам, но просто метод стенографии проверки условия и урегулирования ценности в единственной линии.

Во-первых, `основанный` Variables обращен, и затем установил использование, равняется символу. Тогда, в первом наборе скобок, мы используем немного техники **маски**, чтобы определить, соответствуют ли столкновения во `флагах` Variables (движение нашего диспетчера) внутренне определенной ценности `CollidedBelow`:

```
(flags & CollisionFlags.CollidedBelow)
```

Использование единственного символа амперсанда здесь определяет сравнение между двумя ценностями в двухчастной форме, кое-что, что Вы не должны понять на данном этапе, потому что система Classes Unity предлагает стенографию для большинства вычислений этого типа.

Если наш диспетчер действительно столкнется ниже, и поэтому должен будет быть на основании, то это сравнение будет равно 1.

Это сравнение сопровождается `! =0`. Восклицательный знак перед равняется средствам символа, "не равняется". Поэтому, здесь мы заставляем `основанный` не равняться `0`, если мы сталкиваемся ниже, и в

этих сроках, устанавливая `основанный` к `1` то же самое как урегулирование этого к `истинному`.

Команды @Script

`FixedUpdate ()` Functions заканчивается на линии 27, оставляя только единственную команду в остальной части подлинника, то есть, команда `@script` как показано:

```
@script RequireComponent (CharacterController)
```

Команды `@скрипта` используются, чтобы выполнить действия, которые Вы обычно должны были бы выполнить вручную в Редакторе Unity.

В этом примере `RequireComponent ()` выполнен Functions, который вынуждает Unity добавить, что компонент, определенный в скобках, должен объект подлинник добавляться к не, в настоящее время имеют тот.

Поскольку этот подлинник использует компонент `CharacterController`, чтобы стимулировать наш характер, имеет смысл использовать команду `@script`, чтобы гарантировать, что компонент присутствует и, поэтому, может быть обращен. Это также стоит отмечать, что команды `@script` - единственные примеры команд, которые не должны закончиться с точкой с запятой.

Резюме

В этой главе мы смотрели на первый интерактивный элемент в нашей игре пока - **Первый Диспетчер Человека.** Мы также бросили широкий взгляд на scripting для игр Unity, важный первый шаг, на котором мы будем основываться всюду по этой книге.

В следующей главе Вы начнете писать свои собственные подлинники и изучать далее обнаружение столкновения. Чтобы сделать это, мы будем возвращаться к активу модели заставы, который мы импортировали в Главе 2, вводя это нашей сцене игры, и тому, чтобы заставлять наш характер игрока взаимодействовать с этим использующий комбинацию мультипликации и scripting.

4

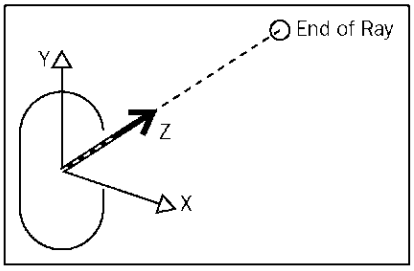
Взаимодействия

В этой главе мы будем смотреть на дальнейшие взаимодействия и

погружение в два из наиболее ключевых элементов развития игры, а именно, **Обнаружения Столкновения** и **Кастинга Луча.**

Чтобы обнаружить физические взаимодействия между объектами игры, наиболее общепринятая методика должна использовать компонент Коллайдера - невидимая сеть, которая окружает форму объекта и отвечает за обнаружение столкновений с другими объектами. Акт обнаружения и восстановления информации от этих столкновений известен как обнаружение столкновения.

Мало того, что мы можем обнаружить, когда два коллайдера взаимодействуют, но и мы можем также давать право на столкновение и выполнить много других полезных задач, используя технику под названием **Кастинг Луча,** который тянет помещенный в Луч просто, невидимую (непредоставленную) векторную линию между двумя пунктами в трехмерном месте - который может также использоваться, чтобы обнаружить пересечение с коллайдером объекта игры. Кастинг луча может также использоваться, чтобы восстановить много другой полезной информации, такой как длина луча (поэтому-расстояние), и точка падения ракеты конца линии.



В данном примере продемонстрирован луч, стоящий перед передовым направлением от нашего характера. В дополнение к направлению лучу можно также дать определенную длину, или позволен бросить, пока это не находит объект.

В течение главы мы будем работать с моделью заставы, которую мы импортировали в Главе 2. Поскольку этот актив был оживлен для нас, мультипликация дверного проема заставы и закрытия готова быть однажды вызванной, модель помещена в нашу сцену. Это может быть сделано или с обнаружением столкновения или с кастингом луча, и мы исследуем то, что Вы должны будете сделать, чтобы осуществить любой подход.

Давайте начнем, смотря на обнаружение столкновения и когда может

быть уместно использовать кастинг луча вместо, или в дополнении к, обнаружение столкновения.

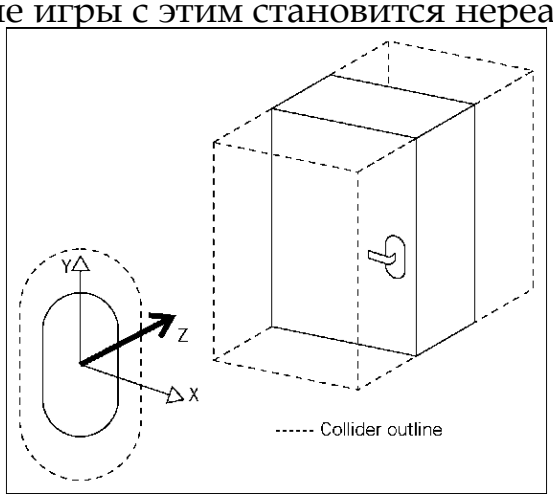
Исследование столкновений

Когда объекты сталкиваются в любом двигателе игры, информация о **случае столкновения** становится доступной. Делая запись множества информации относительно момента воздействия, двигатель игры может ответить в реалистической манере. Например, в игре, вовлекающей физику, если объект падает к основанию от высоты, то двигатель должен знать, какая часть объекта поражала основание сначала. С той информацией это может правильно и реалистично управлять реакцией объекта на воздействие.

Конечно, Unity обращается с этими видами столкновений и хранит информацию от Вашего имени, и Вы только должны восстановить это, чтобы сделать кое-что с этим.

В примере открытия двери мы должны были бы обнаружить столкновения между коллайдером характера игрока и коллайдером на или около двери. Имело бы небольшой смысл обнаруживать столкновения в другом месте, поскольку мы вероятно должны будем вызвать мультипликацию двери, когда игрок будет около достаточно, чтобы идти через это, или ожидать, что это откроется для них.

В результате мы проверили бы на столкновения между коллайдером характера игрока и коллайдером двери. Однако, мы должны были бы расширить глубину коллайдера двери так, чтобы коллайдер характера игрока не должен был быть нажат против двери, чтобы вызвать столкновение, как показано на следующей иллюстрации. Однако, проблема с распространением глубины коллайдера состоит в том, что взаимодействие игры с этим становится нереалистичным.



В примере нашей двери расширенный коллайдер, высовывающийся от визуальной поверхности двери, означал бы, что мы врежемся в невидимую поверхность, которая заставила бы наш характер

останавливаться в течение их следов, и хотя мы будем использовать это столкновение, чтобы вызвать открытие двери через мультипликацию, начальная буква врезаются в расширенный коллайдер, казался бы неестественным игроку и таким образом умалил бы их погружение в игре. Так, в то время как обнаружение столкновения будет работать отлично между коллайдером характера игрока и дверным коллайдером, есть недостатки, которые призывают, чтобы мы как, чтобы творческие разработчики игры искали более интуитивный подход, и это - то, где кастинг луча входит.

Кастинг луча

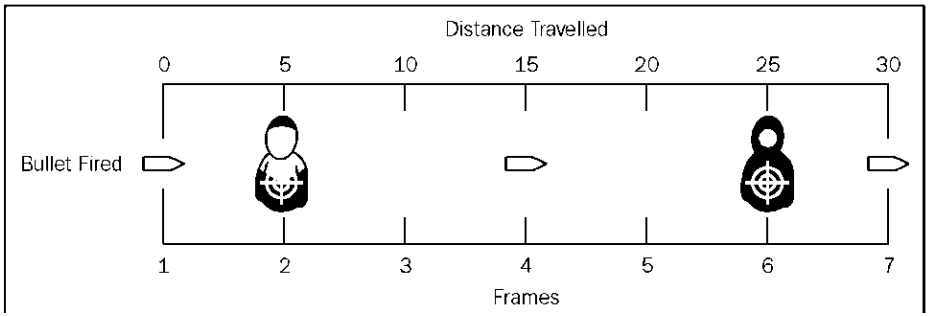
В то время как мы можем обнаружить столкновения между коллайдером характера игрока и коллайдером, который соответствует дверному объекту, более соответствующий метод может быть должен проверить на то, когда характер игрока будет стоять перед дверью, мы ожидаем открываться, и в пределах определенного расстояния этой двери. Это может быть сделано, бросая луч вперед от передового направления игрока и ограничивая его длину. Это означает, что, приближаясь к двери, игрок не должен идти прямо до этого - или врезаться в расширенный коллайдер - для этого, чтобы быть обнаруженным. Это также гарантирует, что игрок не может приблизиться к двери, отворачивающейся от этого, и все еще открыть это - с кастингом луча, они должны стоять перед дверью, чтобы использовать это, которое имеет смысл.

Вместе использование, кастинг луча сделан, где обнаружение столкновения просто слишком неточно, чтобы ответить правильно. Например, реакции, которые должны произойти с покадровым уровнем деталей, могут произойти слишком быстро для столкновения, чтобы иметь место. В этом случае мы должны преимущественно обнаружить, встретится ли столкновение, вероятно, а не столкновение непосредственно. Давайте смотреть на практический пример этой проблемы.

Структура мисс

В примере оружия в трехмерной игре стрелка кастинг луча используется, чтобы предсказать воздействие выстрела, когда оружие запущено. Из-за скорости фактической пули, моделируя курс полета заголовка пули к цели является очень трудным визуально представить в пути, который удовлетворил бы и имел бы смысл игроку. Это до основанной на структуре природы пути, которым предоставлены игры.

Если Вы полагаете, что, когда реальное оружие запущено, оно берет крошечное количество времени, чтобы достигнуть его цели - и насколько наблюдатель обеспокоен, что оно, как могли говорить, случилось немедленно - мы можем предположить, что, отдавая более чем 25 структур нашей игры в секунду, пуля должна была бы достигнуть своей цели в пределах только нескольких структур.



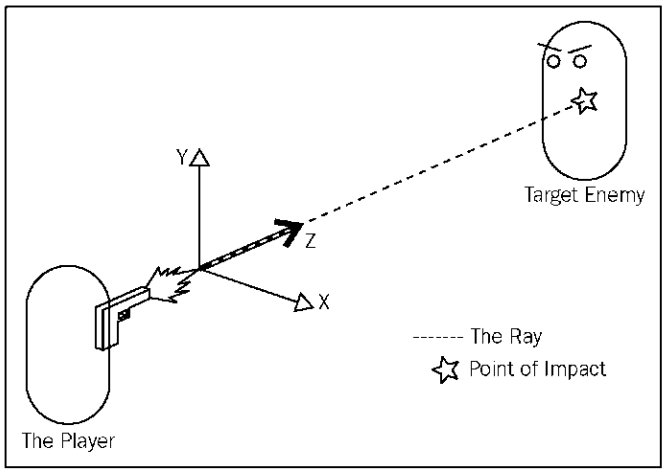
В примере выше, пуля запущена от оружия. Чтобы сделать пулю реалистичной, это должно будет переместиться со скоростью 500 футов в секунду. Если норма структуры - 25 структур в секунду, то шаги пули в 20 футах за структуру. Проблема с этим - человек, приблизительно 2 фута в диаметре, что означает, что пуля будет очень вероятно скучать по врагам, показанным в 5 и на расстоянии в 25 футов, который был бы поражен. Это - то, где предсказание играет роль.

Прогнозирующее обнаружение столкновения

Вместо того, чтобы проверить на столкновение с фактическим объектом пули, мы узнаем, поразит ли запущенная пуля свою цель. Бросая луч отправляют от объекта оружия (таким образом использование его передового направления) на той же самой структуре, что игрок нажимает кнопку огня, мы можем немедленно проверить, какие объекты пересекают луч.

Мы можем сделать это, потому что лучи немедленно оттянуты. Думайте о них как лазерный указатель - когда Вы включаете лазер, мы не видим, что свет продвигается, потому что это едет на скорости света - нам это просто появляется.

Лучи работают таким же образом, так, чтобы всякий раз, когда игрок в основанной на луче игре стрельбы нажимает огонь, они потянули луч в направлении, которое они нацеливают. С этим лучом они могут восстановить информацию относительно коллайдера, который поражен. Кроме того, идентифицируя коллайдер, объект самой игры может быть обращен и подготовлен, чтобы вести себя соответственно. Даже подробная информация, такая как точка падения ракеты, может быть возвращена и использоваться, чтобы затронуть проистекающую реакцию, например, заставляя врага отскочить в специфическом направлении.



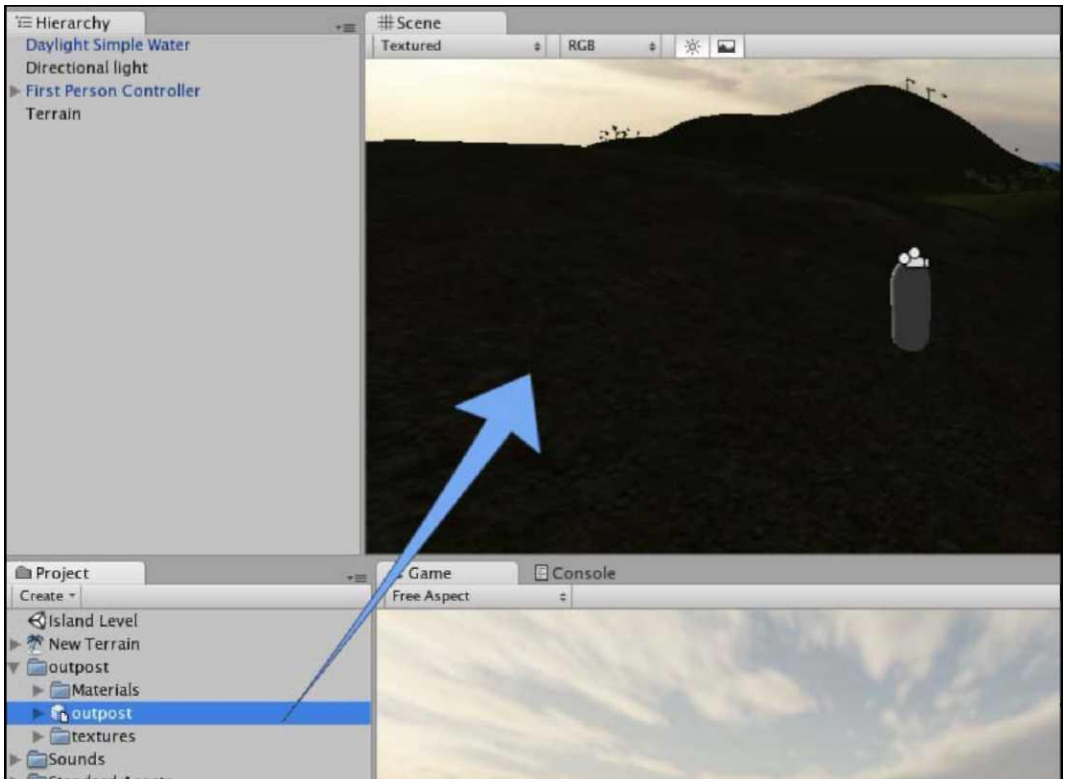
В нашем примере игры стрельбы мы вероятно призвали бы scripting, чтобы убить или физически отразить врага, коллайдер которого хиты луча, и в результате непосредственности лучей, мы можем сделать это на структуре после того, как луч сталкивается с, или *пересекает* вражеский коллайдер. Это дает эффект реального выстрела, потому что реакция немедленно зарегистрирована.

Это также стоит отмечать, что стрельба в игры часто использует иначе невидимые лучи, чтобы отдать краткие видимые линии, чтобы помочь с целью и дать игроку визуальную обратную связь, но не путает эти линии с бросками луча, потому что лучи просто используются как путь для предоставления линии.

Добавление заставы

Прежде, чем мы начнем использовать и обнаружение столкновения и кастинг луча, чтобы открыть дверь нашей заставы, мы должны будем ввести это сцене. В Главе 2 мы настраиваем состояния мультипликации для модели, и это будут те государства, к которым мы будем обращаться, когда мы напишем сценарий позже в главе.

Чтобы начать, тяните модель заставы от **Проектной** группы до **представления Сцены** и понизьте это где-нибудь-медведь в памяти, Вы не можете поместить это, когда Вы drag and drop (перетащили и опустили); это сделано, как только Вы понизили модель (то есть, отпустили от мыши).

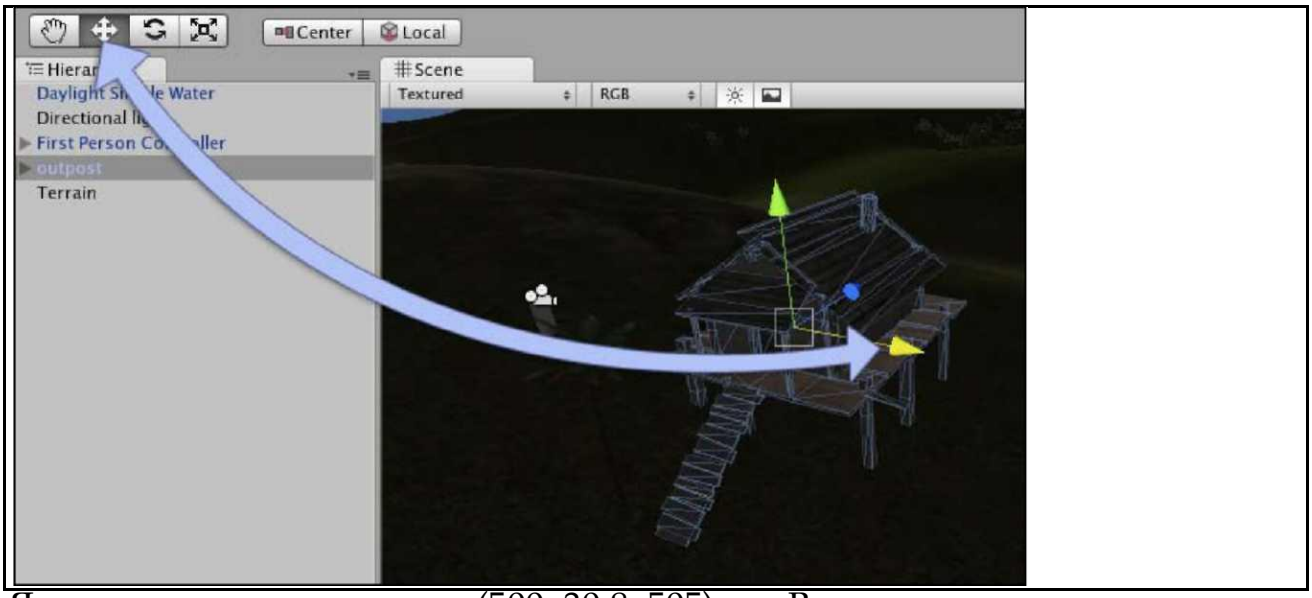


Как только застава находится в **Сцене,** Вы заметите, что ее имя также появилось в группе **Иерархии** и что это автоматически стало отобранным. Теперь Вы готовы поместить и измерить это!

Расположение

Поскольку Ваш проект ландшафта от Главы 2 может быть различным к месторождению, выбрать инструмент **Transform (преобразовать)** и поместить Вашу **заставу** в зону свободной торговли земли при перемещении ручек оси в сцене.

Будьте осторожны, когда использование оси обращается для того, чтобы поместить. Перемещение белого квадрата, где ручки сходятся, приспособит все три топора сразу - не кое-что, что Вы будете хотеть использовать когда в перспективном способе, так гарантируйте, что Вы тянете ручки индивидуально, держа Ваш курсор за пределами белого квадрата.



Я поместил свою заставу в (500, 30.8, 505), но Вы, возможно, должны повторно поместить свой объект вручную. Помните, что, как только Вы поместили использование ручек оси в окно **Scene**, Вы можете войти в определенные ценности в ценности **Позиции** компонента **Transform (преобразовать)** в **Инспекторе.**

Вычисление

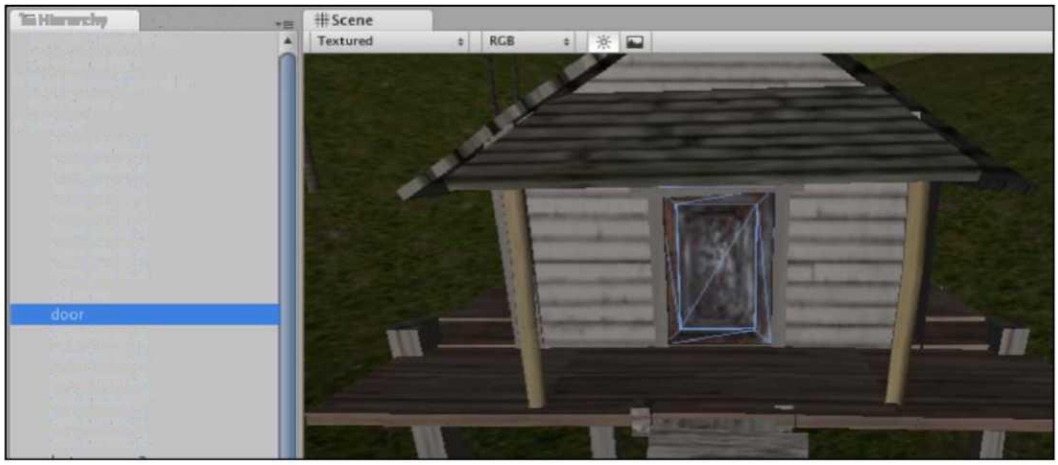
Поскольку мы работаем с двумя применениями - применение моделирования, наш актив **заставы** был создан в и Unity непосредственно, масштаб импортированного актива не будет соответствовать единицам метра в Unity и будет нуждаться в расширении или вниз.

В то время как мы можем настроить ценности **Масштаба** в **Инспекторе,** это лучше везде, где возможный, чтобы настроить Ваш масштаб при использовании урегулирования **Коэффициента пропорциональности** на оригинальном активе. Выберите модель **заставы** в папке **Заставы** в **Проектной** группе. Теперь в **Инспекторе,** Вы будете видеть компонент под названием **FBXImporter.** У этого компонента есть параметры настройки импорта, доступные, который затронет любые случаи модели, которую Вы помещаете в Сцену. Здесь Вы должны установить ценность **Коэффициента пропорциональности** от **1** до **2.** Это делает модель заставы достаточно большой для нашего **Диспетчера Характера**, чтобы соответствовать через заслонку рабочего окна и создание этого реалистично размерной комнате, как только мы проникаем внутрь.

Коллайдеры и маркировка двери

Чтобы открыть дверь, мы должны идентифицировать это как отдельный объект, когда с этим сталкивается игрок - это может быть сделано, потому что у объекта есть компонент коллайдера, и через это мы можем проверить объект на определенный признак. Расширьте объект

родителя **заставы**, нажимая на темно-серую стрелку налево от ее имени в группе **Иерархии**.



Вы должны теперь видеть список всех детских объектов ниже этого. Выберите объект, названный **дверью** и затем с Вашим курсором мыши по окну **Scene**, нажмите *F* на клавиатуре, чтобы сосредоточить Ваш взгляд относительно этого.

Вы должны теперь видеть дверь в окне **Scene**, и в результате отбора объекта, Вы должны также видеть его компоненты, перечисленные в группе **Инспектора**. Вы должны заметить, что один из компонентов - **Коллайдер Петли.** Это - детальный коллайдер, назначенный на все петли, найденные на различных детях модели, когда Вы выбираете**, Производят Коллайдеры,** поскольку мы сделали для актива **заставы** в Главе 2.

> Коллайдер петли назначен на каждый детский элемент, поскольку Unity не знает, сколько деталей будет присутствовать в любой данной модели, которую Вы могли хотеть импортировать. В результате это не выполняет своих обязательств к назначению коллайдеров петли для каждой части, поскольку они будут естественно соответствовать к форме петли, они сталкиваются. Поскольку наша дверь - просто форма куба, мы должны заменить этот коллайдер петли более простым и более эффективным коллайдером коробки.
>
> От главного меню, пойдите в **Компонент | Физика | Коллайдер Коробки.**
>
> Вы тогда получите два, вызывает. Во-первых, Вам скажут, что добавление нового компонента заставит этот объект терять свою связь с родительским объектом в **Проектной** группе. Это окно диалога, названный **Проигрывающий Prefab,** просто означает, что Ваша копия в **Сцене больше не** будет соответствовать

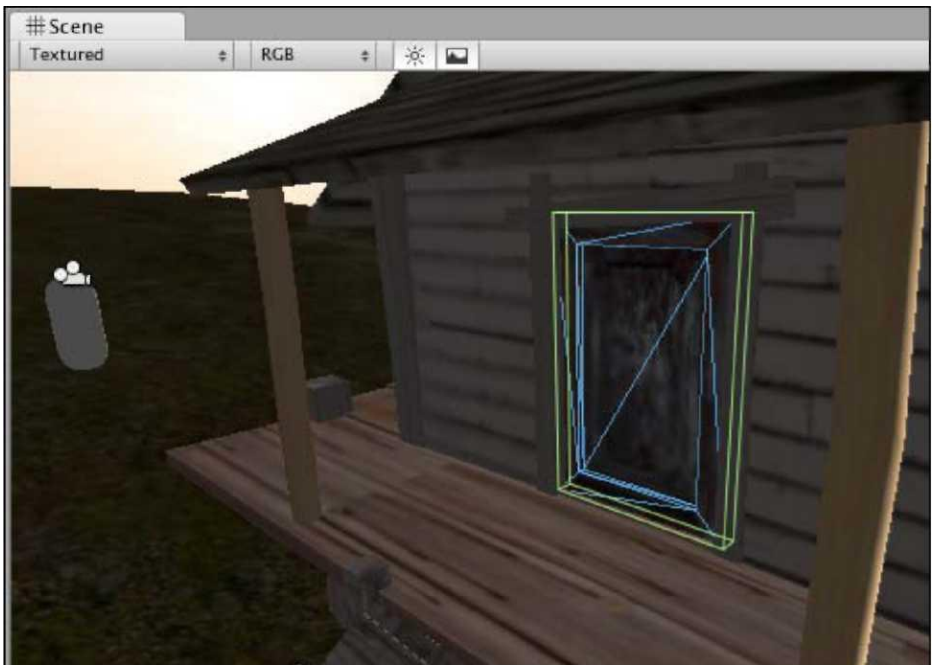
оригинальному активу, и в результате любые изменения, произведенные в активе в **Проектной** группе в Unity, не будут отражены в копии в **Сцене.** Просто нажмите на кнопку **Add**, чтобы подтвердить, что это - то, что Вы хотите сделать.

Это случится всякий раз, когда Вы начинаете настраивать свои импортированные модели в Unity, и это - ничто, чтобы волноваться о. Это - то, потому что, вообще, Вы должны будете добавить компоненты к модели, которая является, почему Unity дает Вам возможность создать Ваши собственные prefab.

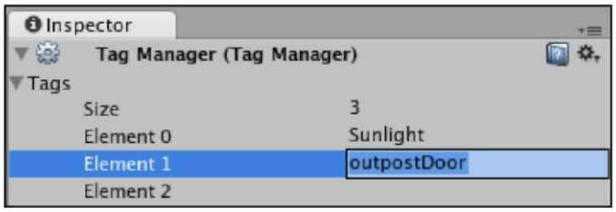
Во-вторых, поскольку объекту уже назначали коллайдер на это, Вы будете побуждены, желаете ли Вы **Добавить, Заменить,** или **Отменить** этот коллайдер к Вашему объекту. Вообще, Вы будете использовать единственный коллайдер за объект, поскольку это работает лучше для двигателя физики в Unity. Это - то, почему Unity спрашивает, хотели ли бы Вы **Добавить** или **Заменить** вместо того, чтобы принять дополнение коллайдеров.

Поскольку мы не имеем никакой дальнейшей потребности в коллайдере петли, выбираем**, Заменяют.**

Вы будете теперь видеть зеленую схему вокруг двери, представляющей компонент **Коллайдера Коробки, который** Вы добавили.



Коллайдер коробки - пример **Примитивного Коллайдера,** так называемого, поскольку это - один из нескольких масштабируемых примитивных коллайдеров формы во включающей Unity Коробке, Сфере, Капсуле, и Колесе - которые предопределили формы коллайдера, и в Unity, все примитивные коллайдеры показывают с этой зеленой схемой. Вы, возможно, заметили это, рассматривая коллайдер диспетчера характера, который является технически краткой формой коллайдера, и как таковой также показывает в зеленом.



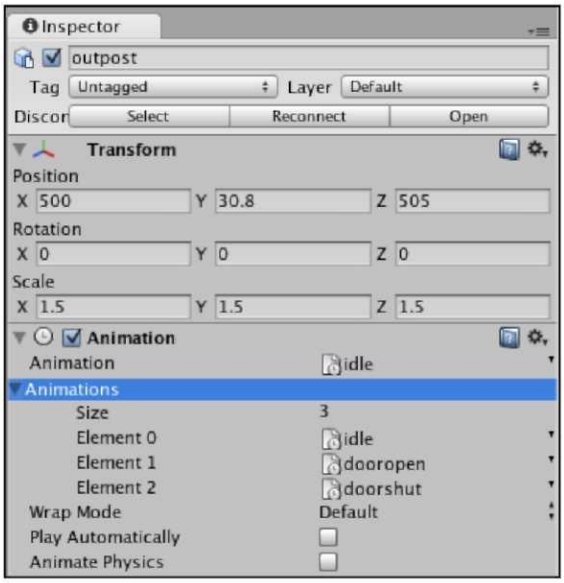
Наконец, мы должны пометить дверной объект, поскольку мы должны будем обратиться к этому объекту в нашем scripting позже. С дверным детским объектом, все еще отобранным, нажмите на **признак**, опускаются наверху группы **Инспектора**, и выбирают, **Добавляет Признак.** В **Менеджере Признака**, который заменяет Ваше текущее представление **Инспектора**, добавьте, что признак **outpostD**Because добавляющие признаки является двумя процессами шага, Вы должны будете повторно выбрать **дверной** детский объект в группе **Иерархии**, и выбрать Ваш недавно добавленный признак **outpostDoor** из **Признака** опускаются меню, чтобы закончить добавлять признак.

Выведение из строя автоматической мультипликации

По умолчанию, Unity предполагает, что все оживляемые объекты, введенные сцене должны будут играться автоматически. Хотя это - то, почему мы создаем праздную мультипликацию - в котором наш актив ничего не делает; разрешение Unity играть автоматически будет часто заставлять оживляемые объекты казаться структура в одну из их намеченных мультипликаций. Чтобы исправить эту проблему, мы просто отсеиваем **Игру Автоматически** checkbox в **Инспекторе** для родительского объекта нашей модели. Обычно, мы не должны были бы делать это, если бы наш актив, постоянно был просто закрепленной петлей мультипликацией играя в мире игры. Например, вздымающийся флаг или вращающаяся лампа маяка, но мы нуждаемся в нашей заставе, чтобы не оживить, пока игрок не достигает двери, таким образом мы избегаем автоматически играть любую мультипликацию.

Чтобы сделать это, повторно выберите родительский объект, названный **заставой** в группе **Иерархии**, и в компоненте **Мультипликации** в группе **Инспектора**, отсейте **Игру Автоматически.**

Групповое представление **Инспектора** объекта заставы должно теперь быть похожим на следующий скриншот:



Отметьте, что я расширил параметр **Мультипликаций** здесь**, я** в компоненте **Мультипликации**, чтобы рассмотреть мультипликацию сообщаю, что **я** в настоящее время обращался к этому объекту.

Объект заставы теперь готов взаимодействоваться нашим характером игрока, таким образом мы должны будем начать scripting, чтобы обнаружить столкновения с нашей недавно теговой дверью, используя или обнаружение столкновения или луч, бросая как обрисовано в общих чертах выше.

Открытие заставы

В этой секции мы будем смотреть на два отличающихся подхода для того, чтобы вызвать мультипликацию, дающую Вам краткий обзор двух методов, которые оба станут полезными во многих других ситуациях развития игры. В первом подходе мы будем использовать обнаружение-а столкновения решающее понятие, чтобы схватиться с тем, поскольку Вы начинаете воздействовать на игры в Unity. Во втором подходе мы осуществим простой бросок луча вперед от игрока.

Приблизьтесь к обнаружению С 1 столкновением

Чтобы начать писать сценарий, который вызовет мультипликацию дверного проема и таким образом предоставит доступ к заставе, мы должны рассмотреть который объект написать сценарий для.

В развитии игры часто более эффективно написать единственный сценарий для объекта, который будет взаимодействовать со многими другими объектами, вместо того, чтобы писать много отдельных сценариев, которые проверяют на единственный объект. С этим в памяти, при письме сценариев для игры, таких как это, мы напишем сценарий, который будет применен к характеру игрока, чтобы проверить на столкновения со многими объектами в нашей окружающей среде, а не сценарий, сделанный для каждого объекта, с которым может взаимодействовать игрок, который проверяет на игрока.

Создание новых активов

Прежде, чем мы введем любой новый вид актива в наш проект, это - хорошая практика, чтобы создать папку, в которой мы будем держать активы того типа. В **Проектной** группе, нажмите на кнопку **Create**, и выберите **Папку** из опускаться меню, которое появляется.

Переименуйте эти **Сценарии** папки, выбирая это и нажимая *Возвращение* (Mac) или нажимая *F2* (PC).

Затем, создайте новый файл JavaScript в пределах этой папки просто, оставляя отобранную папку **Сценариев** и нажимая на кнопку **Create Проектной** группы снова, на сей раз выбирая **JavaScript.**

> Выбирая папку, Вы хотите, чтобы недавно созданный актив был в том, прежде, чем Вы создадите их, Вы не должны будете создать и затем переместить **меня** Ваш актив, поскольку новый актив будет сделан в пределах отобранной папки.

Переименуйте недавно созданный сценарий от default-`NewBehaviourScript`-to `PlayerCollisions`. У файлов JavaScript есть расширение файла `.js`, но группа **Проекта** Unity скрывает расширения файла, таким образом нет никакой потребности попытаться добавить это, переименовывая Ваши активы.

Вы можете также определить тип файла сценария, смотря на его изображение в
Проектная группа. У файлов JavaScript есть 'JS', написанный на них, C# файлы просто
имейте 'C#', и у файлов Boo есть изображение призрака Pacman, миленького
информативная игра слов от парней в Технологиях Unity!

Scripting для обнаружения столкновения характера

Чтобы начать редактировать сценарий, щелкните два раза на его изображении в **Проектной** группе, чтобы начать это в редакторе сценария для Вашей платформы-Unitron на Mac, или Uniscite на PC.

Работа с OnControllerColliderHit

По умолчанию, все новые JavaScripts включают `Update ()` function, и это - то, почему Вы найдете, что он представляет, когда Вы открываете сценарий впервые. Давайте начнем, объявляя variables, которые мы можем использовать всюду по сценарию.

Наш сценарий начинается с определения четырех variables, общественных variables участника и двух частных variables. Их цели следующие:

- `doorIsOpen`: частный `истинный/ложный` (булевый) Variables типа, действующий как выключатель для сценария, чтобы проверить, открыта ли дверь в настоящее время.
- `doorTimer`: частный (помещенный в десятичное число) Variables числа с плавающей запятой, который используется как таймер так, чтобы, как только наша дверь была открыта, сценарий, может посчитать определенное количество времени прежде закрывающимся автоматически дверь.
- `currentDoor`: частный `GameObject` хранение Variables имел обыкновение хранить определенную в настоящее время открываемую дверь. Если Вы желаете добавить больше чем одну заставу к игре позднее, тогда это гарантирует, что открытие одной из дверей не открывает их всех, которых это делает, помня новый дверной хит.
- `doorOpenTime`: с плавающей запятой (потенциально десятичный) числовой общественный Variables участника, который будет использоваться, чтобы позволить нам устанавливать количество времени, мы желаем, чтобы дверь осталась открытой в **Инспекторе.**

- `doorOpenSound/doorShutSound:` Два общественных variables участника типа данных `AudioClip,` для того, чтобы позволить звуковую скрепку drag and drop (перетащили и опустили) назначение в группе **Инспектора**.

Определите variables выше при письме следующего наверху сценария **PlayerCollisions, который** Вы редактируете:

```
private var doorIsOpen : boolean = false;
private var doorTimer  :  float =
0.0,-private var currentDoor  :
GameObject;
var doorOpenTime  :  float =
3.0,-var doorOpenSound :
AudioClip; var doorShutSound :
AudioClip;
```

Затем, мы оставим `Update ()` function кратко, в то время как мы устанавливаем Functions обнаружения столкновения непосредственно. Спустите две линии от:

```
function Update(){ }
```

И напишите в следующем Functions:

```
function OnControllerColliderHit(hit  :
ControllerColliderHit){
}
```

Это устанавливает новый Functions по имени `OnControllerColliderHit.` Этот Functions обнаружения столкновения - определенно для использования с характерами игрока такой как нашим, которые используют компонент **CharacterController**. Его единственный `хит` параметра - Variables, который хранит информацию на любом столкновении, которое происходит. Обращаясь к Variables `хита`, мы можем подвергнуть сомнению информацию относительно столкновения, включая - для стартеров - определенная игра возражает, что наш игрок столкнулся с.

Мы сделаем это, добавляя `если` утверждение нашему Functions. Так в пределах скоб Functions, добавьте следующий `если` утверждение:

```
function OnControllerColliderHit(hit:
ControllerColliderHit){
  if(hit.gameObject.tag == "outpostDoor"  && doorIsOpen
  == false){

  }
}
```

oor, как показано в следующем скриншоте:

В этом, если утверждение, мы проверяем два условия, во-первых который объект мы поражаем, помечен с признаком outpostDoor и во-вторых что Variables doorOpen в настоящее время устанавливается в ложный. Помните здесь, что два равняется символам (==), используются как сравнительное, и два символа амперсанда (&&) просто говорят 'и также'. Исход означает, что, если мы поражаем коллайдер двери, который мы пометили и если мы уже не открыли дверь, тогда это может выполнить ряд инструкций.

Мы использовали точечный синтаксис, чтобы обратиться к объекту, с которым мы проверяем на столкновения, сужая от хита (наша информация хранящего Variables на столкновениях) к gameObject (объектный хит) к признаку на том объекте.

Если это, если утверждение действительно, то мы должны выполнить ряд инструкций открыть дверь. Это вовлечет игру звука, игра одной из мультипликации пристегивается модель, и урегулирование нашей логической переменной doorOpen к истинному. Поскольку мы должны вызвать многократные инструкции - и, возможно, быть должны вызвать эти инструкции в результате различного условия позже, когда мы осуществим подход кастинга луча - мы разместим их в наш собственный Functions под названием Открытый.

Мы напишем этот Functions коротко, но сначала, мы вызовем функцию в, если утверждение мы будем иметь, добавляя:

```
OpenDoor();
```

Таким образом Ваш полный Functions столкновения должен теперь быть похожим на это:

```
function OnControllerColliderHit(hit:
ControllerColliderHit){
  if(hit.gameObject.tag == "outpostDoor"  && doorIsOpen
    == false){ OpenDoor();
  }
}
```

Письмо таможенных functions

Хранение наборов инструкций, которые Вы можете желать вызвать в любое время, должно быть сделано при письме Ваших собственных

functions. Вместо того, чтобы иметь необходимость выписать ряд инструкций или "команд" много раз в пределах сценария, сочиняя Ваши собственные functions, содержащие инструкции, означает, что Вы можете просто вызвать ту функцию в любое время, чтобы управлять тем набором инструкций снова. Это также делает ошибки прослеживания в известном о коде как **Отладка много** более простого, поскольку есть меньше мест, чтобы проверить на ошибки.

В нашем Functions обнаружения столкновения мы написали звонок в Functions под названием `Открытый`. Скобки после `Открытый` используются, чтобы сохранить параметры, которые мы можем желать послать в использование Functions скобкам Functions, Вы можете заставить дополнительное поведение проходить к инструкциям в Functions. Мы будем смотреть на это в большей глубине позже в этой главе под **Эффективностью Functions** заголовка**.** Наши скобки пусты здесь, поскольку мы не желаем передавать любое поведение к Functions все же.

Объявление Functions

Чтобы написать Functions, мы должны звонить, мы просто начинаем при письме:

```
function OpenDoor(){
}
```

Промежуточный скобы Functions, очень таким же образом как инструкции, `если` утверждение, мы помещаем какие-нибудь инструкции, которые будут выполнены, когда эта функция вызвана.

Игра аудио

Наша первая инструкция состоит в том, чтобы играть звуковую скрепку, назначенную на Variables, названный `doorOpenSound`. Чтобы сделать это, добавьте следующую линию к своему Functions, помещая это в пределах вьющихся скоб после `{`"и прежде"`}`:

```
audio.PlayOneShot(doorOpenSound);
```

Чтобы убедиться, это должно быть похожим на это:

```
function OpenDoor(){
  audio.PlayOneShot(doorOpenSound);
}
```

Здесь мы обращаемся к **Звуковому Исходному** компоненту, приложенному к объекту игры, что к этому сценарию относятся (наш объект характера игрока, `Первый Диспетчер Человека`), и как таковой, мы должны будем гарантировать позже, что нам приложили этот компонент; иначе, эта команда вызовет ошибку.

Адресация к звуковому источнику, использующему термин аудио, предоставляет нам доступ к четырем functions, Игра (), Остановка (), Пауза (), и PlayOneShot (). Мы используем PlayOneShot, потому что это - лучший способ играть единственный случай звука, в противоположность игре звука и затем переключению скрепок, которые были бы более соответствующими для непрерывной музыки чем звуковые эффекты. В скобках команды PlayOneShot мы передаем Variables doorOpenSound, который вызовет любой звуковой файл, поручен на тот Variables в **Инспекторе** играть. Мы загрузим и назначим этому и скрепке для того, чтобы закрыть дверь после письма сценария.

Проверка дверного статуса

Одно условие нашего, если утверждение в пределах нашего Functions обнаружения столкновения было то, что наша логическая переменная doorIsOpen должна быть установлена в ложный. В результате вторая команда в нашем Открытом () Functions должна установить этот Variables в истинный.

Это - то, потому что характер игрока может столкнуться с дверью несколько раз, врезаясь в это, и без этого булевого, они могли потенциально вызвать Открытое () Functions много раз, вызывая звук и мультипликацию возвратиться и перезапустить с каждым столкновением. Добавляя в Variables, который когда ложный позволяет Открытому () Functions работать и затем отвергает его, устанавливая doorIsOpen Variables в истинный немедленно, дальнейшие столкновения не будут повторно вызывать Открытое () Functions.

Добавьте линию:

```
doorOpen = true;
```

к Вашему Открытому () Functions теперь, помещая это между вьющимися скобами после предыдущей команды Вы только добавили.

Игра мультипликации

В Главе 2 мы импортировали пакет актива заставы и смотрели на различные параметры настройки на активе прежде, чем ввести это игре в этой главе. Одной из задач, выполненных в процессе импорта, было настраивание скрепок мультипликации, используя **Инспектора.** Выбирая актив в **Проектной** группе, мы определили в **Инспекторе**, что это покажет три скрепки:

- idle ('ничто' не сообщает),
- dooropen
- doorshut

В нашем открытом () Functions, мы призовем названную скрепку, используя **Последовательность** текста обращаться к этому. Однако, сначала мы должны будем сообщить, какой объект в нашей сцене

содержит мультипликацию, которую мы желаем играть. Поскольку сценарий, который мы пишем, быть присоединен к игроку, мы должны обратиться к другому объекту прежде, чем обратиться к компоненту мультипликации. Мы делаем это, сообщая линию:

```
var myOutpost  : GameObject =
GameObject.Find("outpost"),-
```

Здесь мы объявляем новый Variables названным `myOutpost`, заставляя его тип быть `GameObject` и затем выбирая объект игры с `заставой` названия при использовании `GameObject. Найти.` Команда `Находки` выбирает объект в текущей сцене ее именем в **Иерархии** и может использоваться как альтернатива использованию признаков.

Теперь, когда у нас есть Variables, представляющий наш объект игры заставы, мы можем использовать этот Variables с точечным синтаксисом, чтобы назвать мультипликацию приложенной к нему, сообщая:

```
myOutpost.animation.Play("dooropen");
```

Это просто находит компонент мультипликации приложенным к объекту `заставы` и играет мультипликацию, названную **dooropen.** `Игра ()` команду можно передать любая последовательность текстовых характеров, но это будет только работать, если скрепки мультипликации были настроены на рассматриваемом объекте.

Ваше законченное `Открытое ()` таможенный Functions должно теперь быть похожим на это:

```
function OpenDoor(){
  audio.PlayOneShot(doorOpenSound);
  doorIsOpen = true;
  var myOutpost  : GameObject =
  GameObject.Find("outpost");
  myOutpost.animation.Play("dooropen");
}
```

Изменение процедуры

Теперь, когда мы создали ряд инструкций, которые откроют дверь, как мы закроем это, как только это открыто? Чтобы помочь пригодности для игры, мы не будем вынуждать игрока активно закрыть дверь, но вместо этого установить некоторый код, который заставит это закрываться после определенного периода времени.

Это - то, где наш `doorTimer` Variables играет роль. Мы начнем рассчитывать, как только дверь становится открытой, добавляя ценность времени к этому Variables, и затем проверять, когда этот Variables достиг специфической ценности при использовании `если` утверждение.

Поскольку мы будем иметь дело со временем, мы должны использовать Functions, который будет постоянно обновлять, такие как `Update ()`

function, у нас было ожидание нас, когда мы создали сценарий ранее.

Создайте некоторые пустые линии в `Update ()` function, перемещая его заключительную вьющуюся скобу `}` несколько линий вниз.

Во-первых, мы должны проверить, была ли дверь открыта, поскольку нет никакого смысла в увеличивании нашего Variables таймера, если дверь не в настоящее время открыта. Напишите в следующем, `если` утверждение, чтобы увеличить Variables таймера со временем, если `doorIsOpen` Variables установлен в `истинный`:

```
if(doorIsOpen){
    doorTimer += Time.deltaTime;
}
```

Здесь мы проверяем, открыта ли дверь - это - Variables, который по умолчанию установлен в `ложный`, и только станет `верным` в результате столкновения между объектом игрока и дверью. Если `doorIsOpen` Variables `верен`, то мы добавляем ценность `Time (Времени).deltaTime` к `doorTimer` Variables. Примите во внимание, что просто при письме имени переменной, поскольку мы прикончили наш, `если` условие утверждения - то же самое как пишущий `doorIsOpen == истинный`.

> `Time.deltaTime` - Classes Time (Времени), который будет работать независимый от нормы структуры игры. Это важно, потому что Вашей игрой можно управлять на переменных аппаратных средствах когда развернуто, и было бы странным, если бы время, замедленное на более медленных компьютерах и, было быстрее, когда лучшие компьютеры управляли этим. В результате, добавляя время, мы можем использовать `Time.deltaTime`, чтобы вычислить время, потраченное, чтобы закончить последнюю структуру и с этой информацией, мы можем автоматически исправить подсчет в реальном времени.

Затем, мы должны проверить, достиг ли наш Variables таймера, `doorTimer`, определенной ценности, что означает, что определенное количество времени прошло. Мы сделаем это вложением, `если` утверждение в том, который мы только добавили - это будет означать, что, `если` утверждение мы собираемся добавить, будет только проверен, если `doorIsOpen`, `если` условие действительно.

Добавьте следующий код ниже времени, увеличивая линию в существующем `если` утверждение:

```
if(doorTimer > doorOpenTime){
  shutDoor();
  doorTimer = 0.0,}
```

Это дополнение к нашему коду будет постоянно проверяться, как только doorIsOpen Variables становится верным и ждет, пока ценность doorTimer не превышает ценность doorOpenTime Variables, который, потому что мы используем Time.deltaTime как возрастающую ценность, будет означать, что три секунды в реальном времени прошли. Это конечно, если Вы не изменяете ценность этого Variables от его неплатежа 3 в **Инспекторе.**

Как только doorTimer превысил ценность 3, функция, названная shutDoor (), вызвана, и doorTimer Variables перезагружен к нолю так, чтобы это могло использоваться снова в следующий раз, когда дверь вызвана. Если это не будет включено, то doorTimer застрянет выше ценности 3, и как только дверь была открыта, это закроется в результате.

Ваш законченный Update () function должен теперь быть похожим на это:

```
        function Update(){
           if(doorIsOpen){
        doorTimer += Time.deltaTime;

        if(doorTimer > 3){
          shutDoor(); doorTimer
          = 0.0;
        }

    }
```

Теперь, добавьте следующую функцию, вызванную shutDoor () к основанию Вашего сценария. Поскольку это выполняет в значительной степени тот же самый Functions как открытый (), мы не будем обсуждать это подробно. Просто заметьте, что различную мультипликацию вызывают на заставе и что наш doorIsOpen Variables перезагружен к ложному так, чтобы вся процедура могла начаться:

```
function shutDoor(){
audio.PlayOneShot(doorShutSound)
; doorIsOpen = false;

var myOutpost  : GameObject =
GameObject.Find("outpost"),-myOutpost.animation
.Play("doorshut");
}
```

Эффективность Functions

Теперь, когда у нас есть сценарий, отвечающий за открытие и закрытие нашей двери, давайте смотреть на то, как мы можем расширить наше знание сделанных на заказ functions, чтобы сделать наше scripting более эффективное.

В настоящее время у нас есть два functions, которые мы именуем как обычай или сделанный на заказ - открытый () и shutDoor (). Эти functions выполняют те же самые три задачи - они играют звук, устанавливают логическую переменную, и играют мультипликацию. Итак, почему бы не создавать единственный Functions и добавить параметры, чтобы позволить этому играть отличающиеся звуки и сделали, чтобы это выбрало или истинный или ложный для булевого и игры, отличающейся мультипликации? Превращение этих трех задач в параметры Functions позволит нам делать только это. Параметры - параметры настройки, определенные в скобках Functions. Они должны быть отделены запятыми и должны быть даны указанный тип когда объявлено в Functions.

```
function Door(aClip : AudioClip, openCheck : boolean,
animName : String, thisDoor : GameObject){
  audio.PlayOneShot(aClip);
  doorIsOpen = openCheck;

  thisDoor.transform.parent.animation.Play(animName);
}
```

У основания Вашего сценария, добавьте следующий Functions:Вы заметите, что этот Functions выглядит подобным нашим существующим открытым и закрытым functions, но имеет четыре параметра в его декларации-aClip, openCheck, animName, и thisDoor. Они - эффективно variables, которые назначены, когда функция вызвана, и ценность, назначенная на них, используется в Functions. Например, когда мы желаем передать ценности для открытия двери в этот Functions, мы вызвали бы функцию и установили бы каждые параметры при письме:

```
Door(doorOpenSound, true, "dooropen", currentDoor);
```

Это кормит Variables doorOpenSound к aClip параметру, ценности верного для openCheck параметра, последовательности текста "dooropen" к animName параметру, и посылает Variables currentDoor в thisDoor параметр.

Теперь мы можем заменить звонок в открытое () Functions в Functions обнаружения столкновения. Однако, мы должны все же установить currentDoor Variables, таким образом мы должны будем сделать это также. Во-первых, удалите следующую линию, которая называет Открытое () Functions внутри OnControllerColliderHit () Functions:

```
OpenDoor();
```

Замените это следующими двумя линиями:

```
currentDoor = hit.gameObject;
```

```
Door(doorOpenSound, true, "dooropen", currentDoor);
```

Здесь мы устанавливаем `currentDoor` Variables в то, какой бы ни с объектом последний раз столкнулись. Это позволяет нам тогда передавать эту информацию к нашему Functions, гарантируя, что мы только открываемся - который является более аккуратной мультипликацией на - определенная застава, мы сталкиваемся с, а не любая застава с теговой дверью.

В последней линии этого Functions мы играем правильную мультипликацию при использовании `thisDoor.transform.parent.animation`-при-использовании точечный синтаксис здесь, мы прослеживаем, отстраняется к компоненту `мультипликации, к которому` мы должны обратиться. `thisDoor` Variables питался объект, последний раз сохраненный хитом в `currentDoor` Variables - и мы тогда обращаемся к родительскому объекту двери - застава непосредственно, поскольку это - это, которому приложили компонент мультипликации к этому, не дверь. Например, мы не могли сказать:

```
thisDoor.animation.Play(animName);
```

Unity сказал бы нам, что нет никакого компонента мультипликации. Так, вместо этого, мы обращаемся к дверному детскому `родителю transform (преобразовать)` объекта - объект, которому это принадлежит в **Иерархии** - и затем затем выбирать компонент `мультипликации` оттуда.

Наконец, потому что мы используем этот новый метод открытия и закрытия дверей, мы должны будем исправить дверной код закрытия в пределах `Update ()` function. В пределах, `если` утверждение, что проверки на `doorTimer` Variables, превышающий ценность `doorOpenTime` Variables, замените звонок в `shutDoor ()` Functions с этой линией:

```
Door(doorShutSound,   false,   "doorshut",
currentDoor);
```

Вы можете теперь удалить оригинальные два functions - `открытый ()` и `shutDoor ()` как наша настраиваемая `Дверь ()`, Functions теперь заменяет их оба. Создавая functions таким образом, мы не повторяем нас в scripting, и это делает наш сценарий более эффективным и экономит время, сочиняя два functions.

Окончание сценария

Чтобы закончить сценарий, мы удостоверимся, что у объекта, к которому он добавлен, есть компонент `AudioSource` - это необходимо, чтобы воспроизвести звуковые скрепки, поскольку мы делаем в нашем `Дверном` Functions.

Добавьте следующую линию к самому основанию сценария, гарантируя, что линия НЕ закончена с точкой с запятой, поскольку Вы ожидали бы.

Это - то, потому что эта команда является определенной для Unity, и не часть ожидаемых требований JavaScript.

```
@script RequireComponent (AudioSource)
```

Приложение сценария

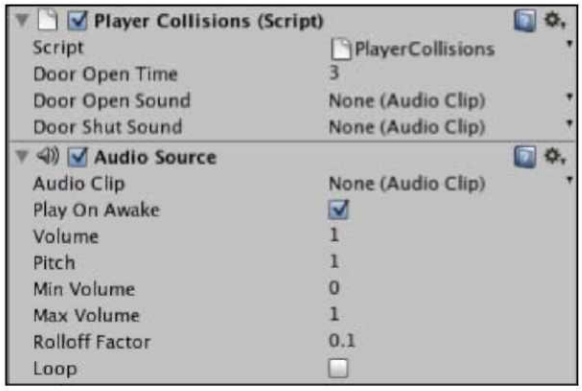
Спасите свой сценарий в редакторе сценария, чтобы гарантировать, что редактор Unity получает любые обновления, которые Вы сделали - это должно быть сделано с любыми изменениями, которые Вы производите в своих сценариях; иначе Unity не может повторно собрать сценарий.

Затем, переключитесь назад на Unity. Проверьте нижний бар интерфейса Unity, поскольку это - то, где любые ошибки, сделанные в сценарии, покажут. Если есть любые ошибки, то щелкают два раза на ошибке и гарантируют что Ваши спички сценария написанное выше. Поскольку Вы продолжаете работать с scripting в Unity, Вы привыкнете к использованию ошибки, сообщая, чтобы помочь Вам исправить любые ошибки, которые Вы можете сделать. Лучшее место, чтобы начать перепроверять это, Вы не сделали ошибок в своем коде, должно гарантировать, что у Вас есть четное число открытия и закрытия вьющихся скоб - это означает, что все functions и `если` утверждения правильно закрыты.

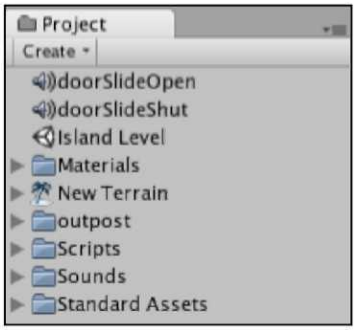
Если у Вас нет никаких ошибок, то просто выбирают объект, к которому Вы желаете применить сценарий в **Иерархии** - объект **First Person Controller**.

Есть два метода приложения сценария к любому объекту в Unity. Во-первых, Вы можете drag and drop (перетащили и опустили) сценарий непосредственно от **Проектной** группы на объект в **Иерархии**, или **Сцена рассматривает,** или просто выбирать объект, Вы желаете применить сценарий к (поскольку Вы только сделали), и от главного меню, пойдите в **Компонент | Сценарии | Столкновения Игрока.** Подменю **Scripts** меню **Component** просто перечисляет любые сценарии, которые это находит в текущем **Проекте;** чтобы скоро, как Ваш сценарий создан и спасен, это будет доступно, чтобы выбрать из того меню.

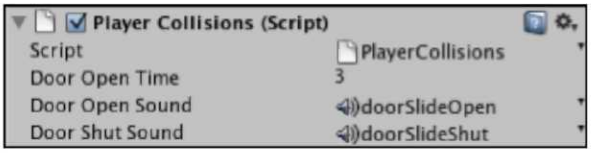
Группа **Инспектора** для **Первого Диспетчера Человека** должна теперь показать **Сценарий Столкновений Игрока** как компонент наряду со **Звуковым Исходным** компонентом, который автоматически был добавлен в результате использования команды `RequireComponent` в конце нашего сценария.



Вы должны отметить, что наши общественные variables участника, `doorOpenTime`, `doorOpenSound`, и `doorShutSound` появляются в **Инспекторе** так, чтобы Вы могли приспособить ценность `doorOpenTime` и drag and drop (перетащили и опустили) звуковые файлы от **Проектной** группы, чтобы назначить на два звуковых variables. Это верно для любых variables участника, с которыми Вы можете столкнуться при письме, что сценарии - помнят, что Вы можете скрыть общественные variables участника от появления в этой манере просто при использовании `частной` приставки, поскольку мы сделали с другими тремя variables, используемыми в сценарии. Примите во внимание, что, используя частные variables, им нужно назначить тип или ценность в сценарии, чтобы избежать ошибок, иначе они будут "`пустыми`" - не имеют никакой ценности. `Открытая` Дверь и `Дверь Близко` звучат, скрепки доступны в кодовой связке, обеспеченной на packtpub.com <http://packtpub.com> (`www.packtpub.com/files/code/8181_Code.zip` <http://www.packtpub.com/files/code/8181_Code.zip>). Извлеките файлы и определите местонахождение пакета, названного `doorSounds.unitypackage`. Чтобы импортировать этот пакет, возвратитесь к Unity и пойдите в **Активы | Пакет Импорта,** проведите к файлу, Вы загрузили и выбираете это. Окно диалога **Активов Импорта** будет казаться перечисляющим эти два файла в пакете. Просто нажмите на кнопку **Import** на этом окне, чтобы подтвердить импорт. У Вас должны теперь быть **doorSlideOpen** и **doorSlideShut** звуковые скрепки в списке Вашей **Проектной** группы активов, как показано в следующем скриншоте. Чтобы держать вещи опрятными, тяните их обоих в существующую папку **Звуков**, где у нас уже есть наш склон окружающая звуковая скрепка.



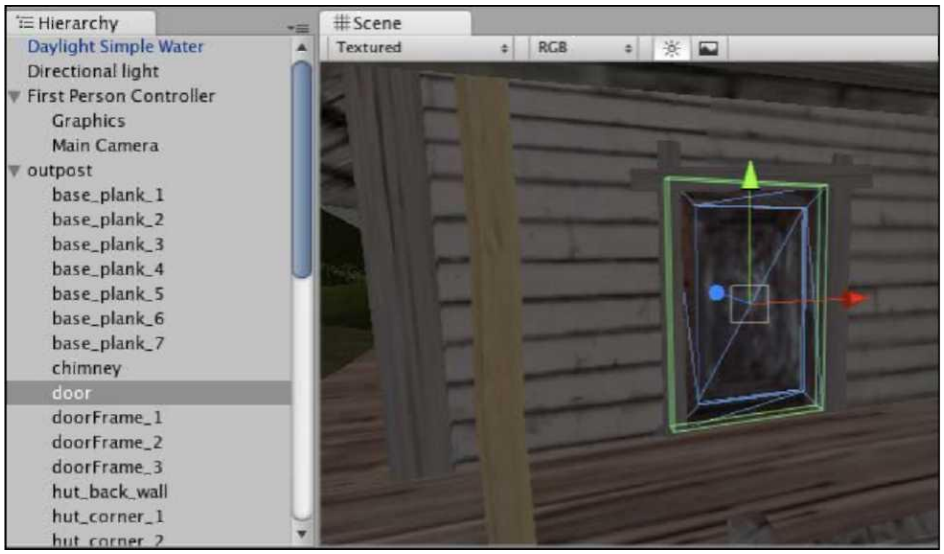
Теперь, когда активы находятся в Вашей проектной папке, drag and drop (перетащили и опустили) соответствующую скрепку к общественным variables участника в **Инспекторе** для **Столкновений Игрока (Сценарий)** компонент. Как только это сделано, компонент сценария должен быть похожим на это:



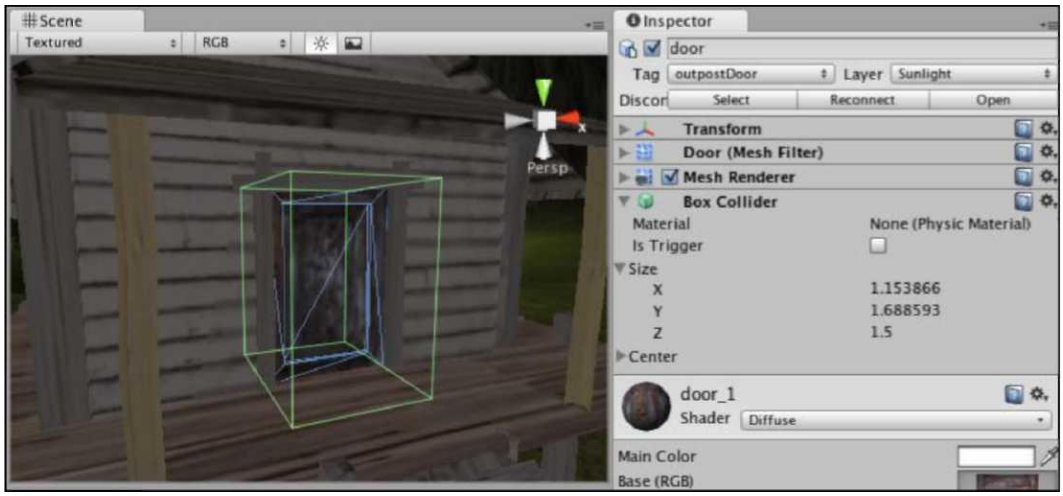
Теперь попытайтесь играть в игру и открыть дверь. Щелкните кнопкой **Play** наверху интерфейса, чтобы начать проверить, и приближаться к двери заставы и врезаться в это.

Столкновение между игроком и дверью будет обнаружено, заставляя дверь открыться, и после **3** секунд, которые дверь закроет автоматически. Нажмите кнопку **Play** снова, чтобы закончить проверять.

Чтобы позволить двери открываться, не будучи нажатым против этого, мы расширим коллайдер двери, как обсуждено в начале главы. Расширьте детские объекты ниже объекта игры **заставы** в группе **Иерархии**, нажимая на серую стрелку рядом с объектом, и затем выберите детский объект, названный **дверью.** Теперь с Вашим курсором мыши по представлению **Сцены**, нажмите *F*, чтобы сосредоточить представление относительно того объекта:



В **Инспекторе** для этого объекта, расширьте параметр **Размера** компонента **Коллайдера Коробки**, так, чтобы Вы могли приспособить ценность **Z** (глубина) коллайдера непосредственно. Измените существующую ценность на ценность **1.5,** так, чтобы коллайдер стал расширенным, как показано в следующем скриншоте:



Теперь попытайтесь играть тест снова, и Вы заметите, что дверь открывается скорее после приближения к нему, поскольку граница коллайдера простирается далее от видимой двери.

Однако, Вы должны также отметить физическое врезание в коллайдер, который происходит. Этому можно противостоять, используя Более

аккуратный способ коллайдера, но мы будем изучать спусковые механизмы далее в следующей главе. Так пока, мы осуществим обсужденный кастинг более раннего луча второго подхода. Бросая луч отправляют от игрока, мы будем отрицать потребность в нашем коллайдере характера игрока, чтобы фактически коснуться коллайдера двери.

Приблизьтесь к кастингу С 2 лучами

В этой секции мы осуществим дополнительный подход к открытию двери с обнаружением коллизий. Хотя обнаружение столкновения или более аккуратное обнаружение были бы действительным подходом, вводя понятие кастинга луча, мы можем гарантировать, что наш игрок только открывает дверь заставы, когда они стоят перед этим, потому что луч будет всегда стоять перед направлением, перед которым оказывается Первый Диспетчер Человека, и как таковой не пересекают дверь, если, например, игрок поддерживает к этому.

Выведение из строя комментариев использования обнаружения столкновения

Чтобы избежать потребности написать дополнительный сценарий, мы просто прокомментируем то есть, временно, дезактивируют часть кода, который содержит наш Functions обнаружения столкновения. Чтобы сделать это, мы добавим характеры, чтобы превратить наш рабочий код столкновения в комментарий.

В программировании комментарии можно оставить как напоминание, но никогда не выполняются. Так дезактивировать часть кода, Вы можете превратить это в комментарий - мы именуем это как **комментирующий.**

Переключитесь назад на Вашего редактора сценария (Unitron/Uniscite) и перед линией:

```
function OnControllerColliderHit(hit:
ControllerColliderHit){
```

place the following characters:

```
/*
```

Помещение передового разреза и звездочки в Ваш сценарий начинает массовый комментарий (в противоположность двум передовым разрезам, которые просто комментируют единственная линия). После Functions обнаружения столкновения, поместите перемену этого, то есть, звездочка, сопровождаемая передовым разрезом. Ваш весь Functions должен был изменить цвет синтаксиса в редакторе сценария и быть написанным как это:

```
/*
function OnControllerColliderHit(hit:
ControllerColliderHit){
```

```
    if(hit.gameObject.tag == "outpostDoor"  && doorIsOpen
      == false){ currentDoor = hit.gameObject;
      Door(doorOpenSound, true, "dooropen", currentDoor);
    }
  }
```

Сброс дверного коллайдера

Поскольку мы не хотим эффекта врезания в невидимый коллайдер на двери, возвратитесь к компоненту **Коллайдера Коробки** двери, выбирая **дверной детский** объект в **Инспекторе** и затем устанавливая ценность **Размера** Оси Z к **0.1.** Это будет соответствовать глубине двери визуально.

Добавление луча

Переключитесь назад на редактора сценария и переместите свой курсор несколько линий вниз от открытия `Update ()` function. Мы помещаем броски луча в `Update ()` function, поскольку мы должны технически бросить наш луч вперед каждая структура. Добавьте в следующем коде:

```
var hit  : RaycastHit;

  if(Physics.Raycast (transform.position,
              transform.forward,  hit,   5)) {
  if(hit.collider.gameObject.tag=="outpostDoor"
    && doorIsOpen == false){ currentDoor =
    hit.collider.gameObject; Door(doorOpenSound,
    true, "dooropen", currentDoor);
  }

}
```

В начале луч создан, устанавливая частный Variables, названный `хитом`, который имеет тип `RaycastHit`. Отметьте, что это не нуждается в `частной` приставке, которая не будет замечена в **Инспекторе** - это является просто частным по умолчанию, потому что это объявлено в Functions. Это будет использоваться, чтобы хранить информацию на луче, когда это пересечет коллайдеры. Всякий раз, когда мы обращаемся к лучу, мы используем этот Variables.

Тогда мы используем два `если` утверждения. Родитель, `если` отвечает за кастинг луча и использует Variables, мы создали. Поскольку мы помещаем кастинг луча в, `если` утверждение, мы в состоянии только вызвать вложенный, `если` утверждение, если луч поражает объект, делая более эффективный сценарий.

Наше первое, `если` содержит `Физику. Raycast`, фактическая команда, которая бросает луч. У этой команды есть четыре параметра в пределах ее собственных скобок:

- Позиция, чтобы создать луч `(transform.position` - позиция

объекта этот сценарий относится, который является **Первым Диспетчером Человека),**

- Направление луча (`transform.forward` - передовое направление объекта этот сценарий относится),
- Структура данных `RaycastHit` мы настраиваем вызванный `хит` - луч, сохраненный как Variables
- Длина луча (`5`-а расстояние в единицах игры, метрах)

Тогда у нас есть вложенный, `если` утверждение, что первые проверки Variables `хита` для столкновения с коллайдерами в мире игры, определенно поразили ли мы коллайдер, принадлежащий объекту игры, помечали `outpostDoor`, таким образом:

```
hit.collider.gameObject.tag
```

Во-вторых, это, `если` утверждение, как гарантированное, наш Functions обнаружения столкновения сделал - что `doorIsOpen` логическая переменная `ложна`. Снова, это должно гарантировать, что дверь не будет повторно вызывать много раз, как только она начала открываться.

Как только оба, `если` условия утверждений соблюдают, мы просто, устанавливают `currentDoor` Variables в объект, хранивший в `хите`, и затем называют `Дверь ()` Functions таким же образом, что мы сделали в нашем Functions обнаружения столкновения:

```
currentDoor                          =
hit.collider.gameObject;
Door(doorOpenSound,              true,
"dooropen");
```

Playtest игра еще раз и Вы заметите, что, приближаясь к двери, это не только открывается прежде, чем Вы врежетесь в это, но также и только открывается, когда Вы оказываетесь перед этим, потому что луч, который обнаруживает дверь, брошен в том же самом направлении, перед которым стоит характер игрока.

Резюме

В этой главе мы исследовали два ключевых метода для того, чтобы обнаружить взаимодействия между объектами в трехмерных играх. У и кастинга луча и обнаружения столкновения есть много отдельного использования, и они - ключевые навыки, которые Вы должны ожидать к повторному использованию в Вашем будущем использовании Unity.

В следующей главе мы изучим другой метод обнаружения столкновения, используя коллайдеры набора объектов, чтобы вызвать способ. Этот способ позволяет обнаружение столкновения, но удаляет физическое присутствие объекта, учитывая пример - коллекция объектов без игрока, врезающегося в них. Попытайтесь думать о спусковых механизмах как о столкновениях без воздействия. В то время как мы, возможно, проявили этот подход с дверью заставы, важно изучить основное понятие

обнаружения столкновения сначала, которое является, почему мы учимся в этом специфическом заказе.

Мы создадим игру коллекции, в которой игрок должен найти четыре батареи, чтобы перезарядить замок на двери заставы, и не позволить вход в заставу, если эти батареи не были собраны.

5

Prefab, Collection, и HUD

В этой главе мы продолжим нашу работу от Главы 4. Работая с подобным подходом к предыдущей главе, мы будем расширять наше знание обнаружения столкновения при использовании третьего метода столкновения, то есть, при использовании коллайдеров как **Спусковые механизмы.**

Спусковые механизмы часто упоминаются как фактические компоненты. Однако, в простых условиях, они - примитивные коллайдеры, с которыми Вы уже знакомы, но с более аккуратным набором способа в **Инспекторе**, использующем**, Более аккуратный** checkbox, как показано в следующем скриншоте:

Поскольку мы уже настроили заставу с вводной дверью, мы теперь ограничим доступ игрока, заставляя их найти объекты, чтобы открыть дверь. Создавая onscreen инструкции, когда игрок приближается к двери, мы сообщим ему, что дверь требует, чтобы источник энергии был открыт. Мы тогда добавим 2-ой показ пустой батареи на экране. Это побудит игрока искать больше объектов, то есть, батарей, которые рассеяны поблизости, чтобы завысить цену достаточной власти открыть дверь.

Создавая эту простую загадку, Вы изучите следующее:

- Работа с 2-ыми объектами, используя Структуры GUI
- Как управлять onscreen текстом с Текстовыми элементами GUI
- Как использовать prefab, чтобы создать кратные числа объекта игры, которые сохранены как актив

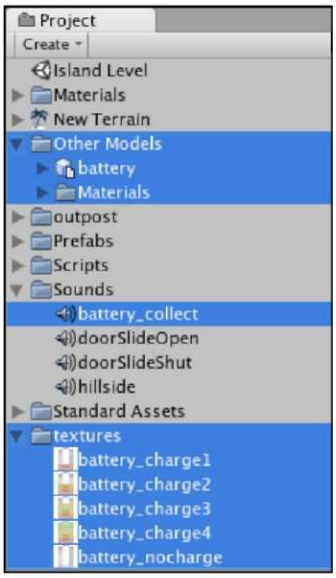
Создание prefab батареи

В этой секции мы импортируем загруженную модель батареи и превратим ее в шаблон данных prefab-a Unity, который мы можем использовать, чтобы сделать многократные копии модели с предопределенными параметрами настройки примененными. Если Вы работали с Adobe Flash прежде, то Вы могли бы сравнить эту идею понятию *MovieClip*, в чем Вы можете создать много идентичных копий, или изменить отдельное создание копий.

Загрузка, импорт, и место

Чтобы начать создавать загадку, Вы будете нуждаться в пакете актива батареи, который доступен в кодовой связке, обеспеченной на packtpub.com (www.packtpub.com/files/<http://www.packtpub.com/files/>code/8181_Code.zip). Определите местонахождение пакета, названного batteries.unitypackage от извлеченных файлов. Как только Вы сделали это, выключатель назад к Unity и идете в **Активы | Пакет Импорта.** Рассмотрите к местоположению, которое Вы загружали пакет к, и выбирать это как файл, чтобы импортировать. Вам подарят список активов в этом пакете в окне диалога **Активов Импорта**, затем нажмете **на Импорт**, чтобы импортировать эти активы. В этом пакете Вами предоставляют:

- Трехмерная модель батареи
- Пять файлов изображения составления счетов батареи с обвинением
- Звуковая скрепка, которая будет играться, после коллекции батареи игроком



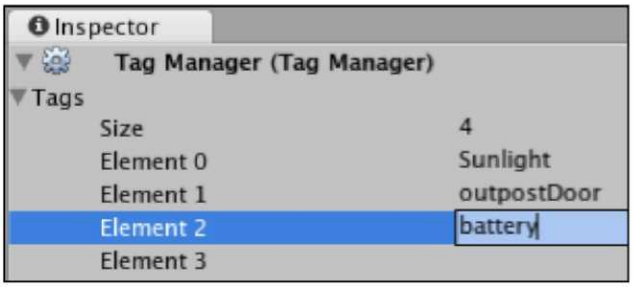
Просто щелкните **Импортом**, чтобы подтвердить здесь. У Вас должны тогда быть следующие файлы в Вашем проекте.

Drag and drop (перетащили и опустили) модель батареи от **Другой** папки **Моделей** в **Проектной** группе на вид **Сцены**. Тогда толпитесь свой курсор по виду **Сцены** и прессе *F*, чтобы сосредоточить вид относительно этого. Ваша батарея будет помещена в произвольную позицию - мы повторно поместим ее, как только мы закончили делать наш prefab.

Маркировка батареи

Поскольку мы должны обнаружить столкновение с объектом **батареи**, мы должны дать ему признак, чтобы помочь нам идентифицировать объект в scripting, который мы напишем коротко. Нажмите на **Признак**, опускаются, меню, и у основания избранного меню **Добавляет Признак.**

Группа **Инспектора** тогда переключается, чтобы показать **Менеджера Признака.** В следующей доступной щели **Элемента**, добавьте признак, названный **батареей,** как показано в следующем скриншоте:



Нажмите *Вступает*, чтобы подтвердить признак, затем повторно выбрать объект **батареи** в группе **Иерархии**, и выбирать новый признак **батареи** из **Признака** опускаются меню в **Инспекторе** для того объекта.

Масштаб, коллайдер, и вращение

Теперь, мы подготовим **батарею** как prefab, применяя компоненты и параметры настройки, которые мы хотели бы оставить в каждой новой копии **батареи.**

Увеличение батареи

Мы создаем кое-что, что игрок обязан собирать, таким образом мы должны гарантировать, что объект имеет разумный размер для игрока, чтобы определить в игре. Поскольку мы уже смотрели на вычисление объектов в FBXImporter для актива, мы будем смотреть на простое изменение размеров с компонентом Transform **(преобразовать)** в **Инспекторе.** С объектом **батареи**, все еще отобранным в **Иерархии,** измените все ценности **Масштаба** в компоненте Transform **(преобразовать) Инспектора** к 6.

Добавление более аккуратного коллайдера

Затем, мы должны добавить примитивный коллайдер к **батарее**, чтобы позволить игроку взаимодействовать с этим. Пойдите в **Компонент | Физика | Краткий Коллайдер.** Мы выбрали этот тип коллайдера, поскольку это - самая близкая форма на нашу **батарею.** Поскольку мы не хотим, чтобы игрок врезался в этот объект, собирая это, мы заставим его коллайдер вызывать способ. Так, на недавно добавленном **Кратком** компоненте **Коллайдера**, выберите коробку направо от**, Более аккуратное** урегулирование.

Создание эффекта вращения

Теперь мы напишем сценарий, чтобы заставить объект **батареи** вращать, чтобы добавить визуальный эффект и облегчить для игрока замечать это. На **Проектной** группе, выберите папку `Сценариев`, чтобы гарантировать, что сценарий, который мы собираемся создать, создан в

пределах той папки. Пойдите в кнопку Create на **Проектной** группе, и выберите JavaScript. Нажмите *F2* и тип, чтобы переименовать недавно созданный файл от NewBehaviourScript до RotateObject. Щелкните два раза его изображением, чтобы начать это в редакторе сценария.

Наверху Вашего нового сценария вне Update () function, создайте общественный Variables участника с плавающей запятой, названный rotationAmount, и установите его ценность в 5.0, добавляя следующую линию:

```
var rotationAmount    :    float = 5.0;
```

Мы будем использовать этот Variables, чтобы определить, как быстро объект **батареи** вращается. Поскольку это - общественный Variables участника (за пределами любого Functions и не объявленный как частный), мы также будем в состоянии приспособить эту ценность в **Инспекторе**, как только сценарий присоединен к **батарее.**

В пределах Update () function, добавьте следующую команду, чтобы вращать нашу **батарею** вокруг ее Оси Y с ценностью, равной rotationAmount Variables:

```
function Update    () {
  transform.Rotate(Vector3(0,rotationAmount,0)),-
}
```

Вращение () команда ожидает Vector3 (X, Y, Z) ценность, и мы обеспечиваем ценности 0 для X и Z,устанавливая ценность Variables в Ось Y. Поскольку мы написали эту команду в пределах Update () function, она будет выполнена в каждой структуре, и таким образом батарея будет вращаться 5 степенями каждая структура. В редакторе сценария, пойдите в **Файл |, Экономят** и затем переключаются назад на Unity.

Чтобы приложить этот сценарий к нашему объекту **батареи**, гарантируйте, что это отобрано в группе **Иерархии**. Тогда, пойдите в **Компонент | Сценарии |** RotateObject или просто drag and drop (перетащили и опустили) сценарий от **Проектной** группы на объект в группе **Иерархии**.
Нажмите кнопку Play наверху интерфейса, и наблюдайте **батарею** в виде **Сцены**, чтобы удостовериться, что вращение работает. Если это не работает, то возвратитесь к своему сценарию и проверьте, что Вы не сделали ошибок - перепроверка, применили ли Вы сценарий к правильному объекту! Не забудьте нажимать **Игру** снова, чтобы закончить Ваше тестирование перед продолжением.

Экономия как prefab

Теперь, когда объект **батареи** полон, мы должны будем клонировать объект три раза, давая нам в общей сложности четыре батареи. Лучший

способ сделать это с заранее приготовленной системой Unity. Создание prefab просто означает делать шаблон данных, который хранит параметры настройки объекта, созданного в сцене. Это может тогда быть клонировано, редактируя, или иллюстрироваться примерами (созданный на лету в игре) во времени выполнения.

Создайте новую папку в пределах **Проектной** группы для того, чтобы сохранить это и любые будущие prefab. Чтобы сделать это, сначала гарантируйте, что Вам не выбирали существующую папку, щелкая в некотором сером месте ниже объектов в Вашей **Проектной** группе. Нажмите на кнопку **Create** и выберите **Папку,** затем переименуйте проистекающую новую папку к **Prefab**, нажимая *Возвратитесь* (Mac) или *F2* (PC) и перепечатывание.

Выберите недавно сделанную папку **Prefab**. От кнопки **Create**, выберите **Prefab.** Это создает новый пустой prefab в пределах папки **Prefab**.

> Вы можете определить пустой prefab, поскольку у этого будет серое изображение куба, как отклонено **я** к законченному prefab, у которого есть светло-голубое изображение куба.

Переименуйте новый пустой prefab на **батарею.** Теперь тянитесь, **батарея** возражают, что мы продолжали работать от группы **Иерархии**, и понижаем ее на пустой объект prefab **батареи** в **Проектной** группе.

Это превращает Вашу батарею в prefab и также делает копию в текущей сцене связанной с тем prefab, что означает, что любые изменения, произведенные в prefab в **Проектной** группе, будут отражены в копии в сцене. Объекты в сцене, связанной с активами в проекте, показывают в группе **Иерархии** с синим текстом, в противоположность черному тексту объектов только для сцены.

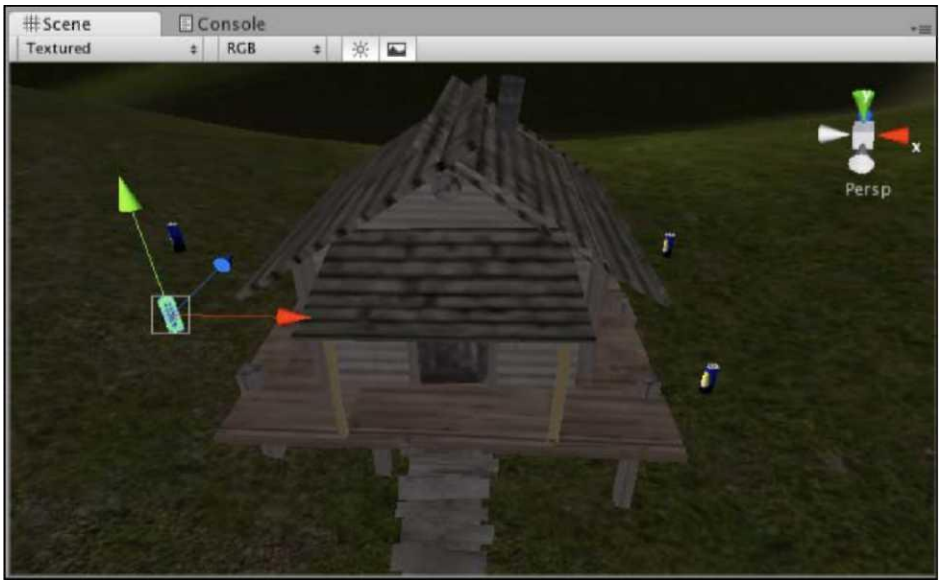
Рассеивание батарей

Теперь, когда нам сохранили наш объект батареи как prefab, когда мы дублируем копию в сцене, мы создаем дальнейшие случаи prefab. Гарантируйте, что Вам все еще выбрали **батарею** в **Иерархии** и затем дублируете объект **батареи** три раза так, чтобы Вы имели четыре всего, это может быть сделано или собираясь **Редактировать | Дубликат** или используя сокращенную *Команду* клавиатуры *+ D* (на Mac) или *Ctrl + D* (на PC).

> То, когда объекты в сцене дублированы, дубликаты созданы в той же самой позиции - не позволяет этому смущать Вас. Unity просто делает это, чтобы стандартизировать, где новые объекты в сцене заканчиваются, и это - то, потому что, когда объект дублирован, каждое урегулирование - идентичное включение позиция. Кроме того, легче помнить, что они находятся в той же самой позиции как оригинал и просто нуждаются в перемещении от той позиции.

Теперь, выберите каждую из этих четырех батарей в группе **Иерархии**, и используйте **инструмент Transform (преобразовать)**, чтобы повторно поместить их вокруг заставы. Помните, Вы можете использовать штуковину вида в верхнем правом из вида **Сцены**, чтобы переключиться от перспективного вида, чтобы превысить, понять, и виды сбоку. Не забудьте помещать батареи в высоту, в которой игрок может поднять их, так что не заставляйте их слишком высоко достигать.

Как только Вы поместили эти четыре батареи вокруг заставы, у Вас должно быть кое-что как это:



Показ батареи GUI

Теперь, когда у нас есть наши предметы коллекционирования батареи в месте, мы должны будем показать игроку визуальное представление того, что они собрали. Структуры, импортированные с пакетом батарей, были разработаны, чтобы ясно показать, что игрок должен будет собрать четыре батареи, чтобы полностью зарядить дверь. Обменивая пустое изображение батареи onscreen для одного с 1 единицей обвинения, затем для изображения с 2 единицами обвинения, и так далее, мы можем создать иллюзию динамического элемента интерфейса.

Создание объекта Texture GUI

Часть активов, импортированных с пакетом батарей, была папкой `структур`. Эта папка содержит пять файлов изображения - один из пустого элемента аккумуляторной батареи и другие четырех стадий заряда батареи. Созданный в Adobe Photoshop, эти образы имеют прозрачный фон, и сохранены в **PNG (Портативная Графика Сети)** формат. Формат PNG был отобран, потому что он сжат, но все еще

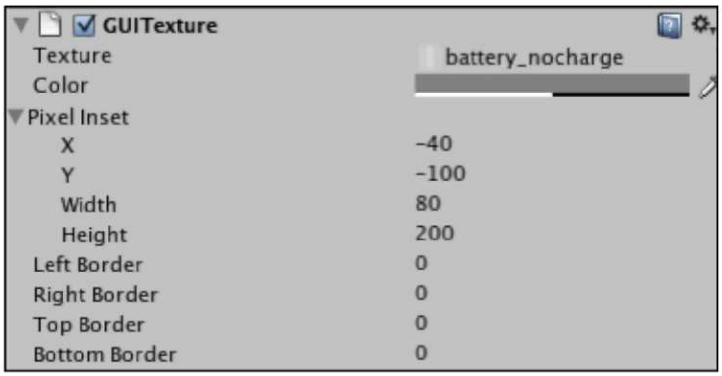
поддерживает высококачественные **альфа-каналы.** Альфа-каналы - то, что много частей программного обеспечения именуют как канал изображения (помимо обычных красных, зеленых, и синих каналов), который создает прозрачность.

Мы начнем создавать GUI показ продвижения батареи, добавляя пустую батарею, графическую к сцене, используя компонент **Структуры** Unity **GUI**. В **Проектной** группе, расширьте папку **структур**, и выберите файл, названный **battery_nocharge.**

В пути, различном к нашему нормальному методу перемещения и бросания активов в вид **Сцены**, мы должны определенно создать новый объект с компонентом Структуры GUI и определить battery_nocharge графику как структуру для того компонента, чтобы использовать.

В то время как это - технически процедура с тремя шагами, это может быть сделано в отдельном шаге, выбирая структуру, которая будет использоваться в **Проектной** группе, и затем движение к **GameObject | Создает Другой | Структура GUI** от главного меню. Сделайте это теперь.

Новый объект будет создан с приложенным компонентом **GUITexture**. В **Инспекторе** будут уже определены области **Вставки Пиксела** (см. следующий скриншот), основанный на измерениях выбранной структуры, которые Unity читает от файла:



Области **Вставки Пиксела** определяют измерения и показывают область объекта. Типично параметры **Ширины** и **Высоты** должны быть определены, чтобы соответствовать оригинальным измерениям Вашего файла структуры, и **X** и ценности **Y** должны быть установлены в половину измерений. Снова, это настроено для Вас, когда Вы создаете объект, выбрав структуру сначала.

Выбирая графику Вы желаете использовать сначала, Unity знает, что, когда это создает новый объект с компонентом **GUITexture**, это должно выбрать тот файл как структуру и заполниться в измерениях структуры также.

Кроме того, когда Вы выбираете графику и создаете объект **Texture GUI**, используя главное меню, созданный объект называют после Вашего актива структуры ради удобства - помнят это, поскольку это поможет Вам найти объект в группе **Иерархии**, отправьте создание.

Выберите объект, который Вы создали в группе **Иерархии**, выбирая **battery_nocharge,** и переименовываете ее на **Батарею GUI**, нажимая *Возвращение* (Mac) или *F2* (PC) и перепечатывание.

Расположение Структуры GUI

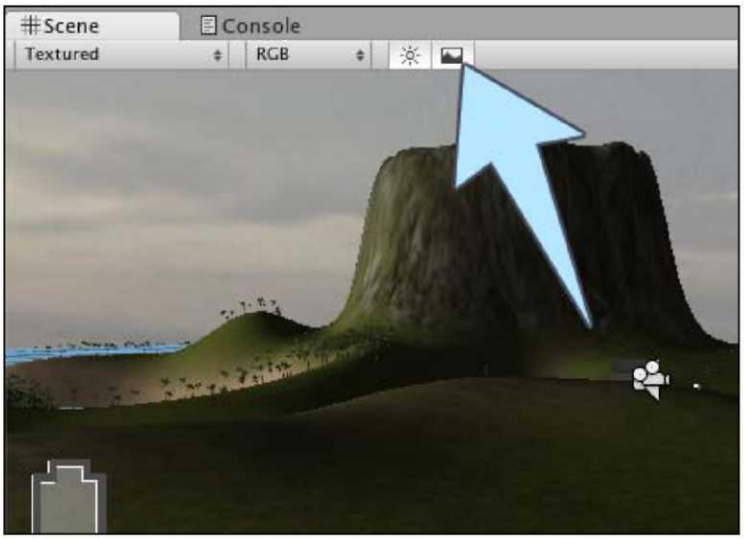
Имея дело с 2-ыми элементами, Вы должны будете работать в координатах экрана. Поскольку они являются 2-ыми, они только работают в X и топорах Y - с Осью Z, используемой для приоритета слоя между многократными элементами Структуры GUI.

Координаты экрана идут с приращением в десятичных числах от 0 до 1, и чтобы поместить **Батарею GUI**, где мы хотим видеть это (в более низко-левом экрана), мы должны напечатать ценности в **X** и коробки **Y** компонента **Transform (преобразовать)** объекта **GUI Батареи**.

Заполнитесь в ценности **0.05** в **X** и **0.2** в **Y.**

Вы должны теперь видеть графику, показанную на виде **Игры**, как показано в следующем скриншоте:

Вы можете также показать 2-ые детали в виде **Сцены**, нажимая на кнопку Game Overlay, как показано в следующем скриншоте:



Scripting для изменения GUI

Теперь, когда у нас есть несколько батарей, чтобы собраться, и поскольку это - первая стадия нашей батареи GUI onscreen, мы должны написать сценарий, чтобы обратиться к компоненту Структуры GUI и обменять его структуру, основанную на том, сколько батарей было собрано.

Выберите папку **Сценариев** в **Проектной** группе, и затем нажмите на кнопку Create и выберите JavaScript. Переименуйте NewBehaviourScript, **который** Вы создали к BatteryCollect, и затем щелкаете два раза его изображением, чтобы открыть его в редакторе сценария.

Поскольку мы пишем сценарий, чтобы управлять компонентом Структуры объекта **GUI Батареи** GUI, лучше писать сценарий, чтобы быть присоединенным к тому объекту, так, чтобы связанные сценарии были присоединены к своим соответствующим объектам.

Сочиняя следующий сценарий, структура, которая используется onscreen, будет основана на том, сколько батарей было собрано - эта информация будет храниться в целом числе (целое число) Variables. После того, как мы будем закончены, сочиняя этот сценарий, мы приложим к нашему сценарию PlayerCollisions, добавляя более аккуратное обнаружение столкновения для батарей. Каждый раз, когда столкновение обнаружено с батареей, мы увеличим целое число в нашем сценарии BatteryCollect, который обменяет структуру компонента Структуры GUI на **Батарее GUI.**

Наш сценарий начинается, устанавливая `статический` Variables-a, глобальный, доступный от других сценариев - чтобы сохранить

количество собранного обвинения. Мы делаем этот Variables целым числом, поскольку нет никакого шанса его не быть целым числом. Добавьте следующую линию к вершине сценария теперь:

```
static var charge    :    int = 0;
```

Тогда мы нуждаемся в пяти variables, чтобы сохранить структуры, которые представляют пять различных государств нашей GUI-пустой **Батареи**, плюс четыре стадии обвинения. Так, мы добавим пять неназначенных variables типа Texture2D:

```
Var charge1tex    :    Texture2D;
var charge2tex    :    Texture2D;
var charge3tex    :    Texture2D;
var charge4tex    :    Texture2D;
var charge0tex    :    Texture2D;
```

Поскольку они - общественные variables участника, мы просто назначим файлы изображения им от **Проектной** группы, использование drag and drop (перетащили и опустили) к щелям Variables, которые будут созданы в **Инспекторе** (составляющий) вид этого сценария.

Чтобы настроить неплатежи для компонента Структуры GUI, добавьте следующее `Начало` () Functions к сценарию, ниже variables, которые Вы только объявили:

```
function Start(){
  guiTexture.enabled = false; charge = 0;
}
```

`Начало` () Functions выполнит однажды, когда текущий уровень (или сцена, в условиях Unity) начался. Помещая в линии:

```
guiTexture.enabled = false;
```

Мы гарантируем, что компонент не позволен, так, чтобы батарея не была видима, когда игра начнется. Мы также установили Variables обвинения в 0, когда игра начинается, чтобы гарантировать, что сценарий предполагает, что никакие батареи не собраны на начале сцены.

Затем, переместите заключительную правильную вьющуюся скобу Update () function вниз несколькими линиями, и поместите следующий `если` утверждение в этом:

```
if(charge == 1){
  guiTexture.texture = charge1tex;
  guiTexture.enabled = true;
}
```

Здесь мы проверяем Variables `обвинения` на ценность 1, и затем выполняем две команды:

- guiTexture.texture = charge1tex;: Эта линия устанавливает щель структуры компонента Структуры GUI (объекта, к которому этот сценарий присоединен), чтобы использовать файл изображения, назначенный на charge1tex Variables.
- guiTexture.enabled =true;: Эта линия позволяет компонент непосредственно. Мы заставили **Батарею GUI** не быть видимой (то есть, не позволили), когда игра начинается, чтобы не смутить пользователя, и избежать загромождать экран - мы только хотим, чтобы они видели пустую батарею GUI, как только они попытались открыть дверь.

  Эта линия сценария отвечает за предоставление возможности GUI, когда игрок собрал одну батарею, потому что этим управляют `если` утверждение. Позже, мы добавим scripting к дверному столкновению луча в нашем сценарии PlayerCollisions, чтобы включить пустую батарею GUI, если пользователь, случится, попытается войти в дверь перед собиранием его первой батареи.

Затем мы добавим еще три `если` утверждения Update () function, проверяющему на другие государства Variables `обвинения`. Как никогда не будет случая, где обвинение равняется и 1 и другая ценность, мы `еще` будем использовать, `если` утверждения, поскольку они проверены, когда любой из другого, `если` утверждения уже находятся в игре.

Если мы должны были использовать, `если` для каждого из следующих утверждений, они будут все проверены одновременно, делая неэффективный сценарий. Добавьте следующий код после заключительной вьющейся скобы Вашего существующего `если` утверждение:

```
else if(charge == 2){
    guiTexture.texture = charge2tex;
}
else if(charge == 3){
    guiTexture.texture = charge3tex;
}
else if(charge >= 4){
    guiTexture.texture = charge4tex;
}
```

В них `еще`, `если` утверждения, мы просто заставляем структуру, показанную компонентом Структуры GUI использовать отличающиеся структуры при использовании общественных variables участника, установленных наверху сценария (мы назначим файлы изображения на эти variables в **Инспекторе** позже). Мы не должны `еще` позволять компонент в любом из них, `если`

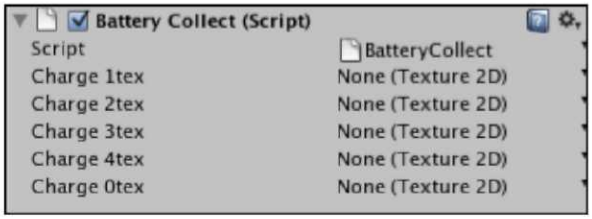
утверждения, поскольку он был бы уже позволен, когда игрок поднял свою первую батарею.

Наконец, чтобы обратиться, когда наш Variables `обвинения` в 0, мы `еще` можем просто добавить утверждение до конца нашего, `если` и `еще если` утверждения, который эффективно говорит, "если ни одно из вышеупомянутого не верно, затем сделайте следующий." Поэтому, как наше существующее, `если` и `еще если` утверждения `еще` ищут ценность обвинения 1 - 4, следующий, утверждение будет заботиться о показе, когда игра начнется (когда Variables обвинения будет равняться 0). Это по существу поддерживает `Начало` () Functions:

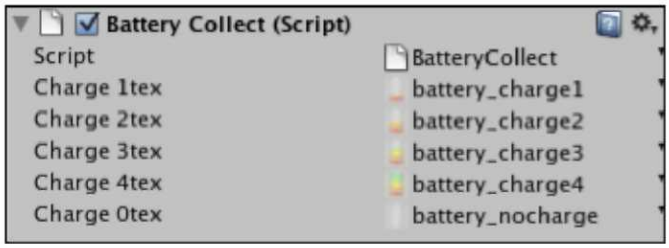
```
else{
  guiTexture.texture = charge0tex;
}
```

Теперь, когда сценарий полон, пойдите в **Файл |, Экономят** в редакторе сценария и переключаются назад на Unity.

Выберите объект **GUI Батареи** в группе **Иерархии** и пойдите в **Компонент | Сценарии | BatteryCollect** от главных меню. Это назначает наш сценарий на объект **GUI Батареи**, и Вы должны видеть это в **Инспекторе** с пятью общественными variables участника, ждущими назначения, как показано в следующем скриншоте:



Drag and drop (перетащили и опустили) пять файлов структуры от папки **структур Проектной** группы, которую Вы импортировали ранее к соответствующим variables участника, как показано в следующем скриншоте:



Теперь, когда наш onscreen GUI для коллекции батареи готов, мы просто должны добавить это к сценарию PlayerCollisions, приложенному к **Первому Диспетчеру Человека,** чтобы обнаружить наше взаимодействие характера игрока с более аккуратными коллайдерами

объектов батареи.

Коллекция батареи со спусковыми механизмами

Чтобы вызвать отличающиеся государства нашей **Батареи GUI,** мы будем использовать Functions по имени OnTriggerEnter (), чтобы обнаружить взаимодействие с объектами, у которых есть более аккуратные коллайдеры способа, то есть, наши коллекционируемые батареи.

Прежде, чем мы добавим этот Functions к нашему сценарию PlayerCollisions, мы добавим общественный Variables участника наверху, чтобы считать, что звуковая скрепка играется, когда игрок поднимет батарею как форму основанной на аудио обратной связи, чтобы поддержать элемент GUI.

Откройте сценарий **PlayerCollisions**, щелкая два раза на его изображении в папке **Сценариев Проектной** группы. Это начнет сценарий в редакторе сценария, или просто переключится назад на него для Вас, если Вам уже откроют файл. Добавьте общественный Variables участника для звуковой скрепки к вершине сценария, добавляя линию:

```
var batteryCollect     : AudioClip;
```

Помните, что этому не назначают в сценарии, но просто оставлено с типом данных **(AudioClip)** так, чтобы это могло быть назначено в **Инспекторе** позже.

Затем, поместите свой курсор выше заключительной линии сценария:

```
@script RequireComponent (AudioSource) И
```

добавляют в следующем Functions:

```
function OnTriggerEnter(collisionInfo     : Collider){
}
```

Это - Functions определенно для того, чтобы обнаружить столкновения с более аккуратными коллайдерами способа. Любое столкновение с таким коллайдером помещено в параметр, названный collisionInfo, который имеет `Коллайдер` типа, означая, что информация, подвергнутая сомнению от этого, должна быть сделана в ссылке на объект, приложенный к коллайдеру, который был поражен.

В OnTriggerEnter () Functions (то есть, перед его столкновением права [закрывающим] вьющуюся скобу), добавляют следующие, `если` утверждение, чтобы подвергнуть сомнению любые спусковые механизмы, с которыми сталкиваются:

```
    if(collisionInfo.gameObject.tag == "battery"){
```

```
        BatteryCollect.charge++;
        audio.PlayOneShot(batteryCollect);
        Destroy(collisionInfo.gameObject);
    }
```

С этим, `если` утверждение, мы подвергаем сомнению collisionInfo параметр Functions, проверяя, присоединен ли коллайдер в текущем столкновении к объекту игры, помеченному с `батареей как` слова, которая наши предметы коллекционирования батареи. Если дело обстоит так, то мы делаем три вещи:

- Обратитесь `к обвинению` сценария BatteryCollect статический (глобальный) Variables, используя точечный синтаксис, и добавьте тот к его ценности, используя ++ метод. Добавляя к ценности `обвиняют` в каждом столкновении с батареей, мы делаем сценарий BatteryCollect обменивает структуры, которые представляют обвинение, заканчивая связь между нашими столкновениями и onscreen GUI.
- Играйте один выстрел (отдельный случай) звуковой скрепки, назначенной на batteryCollect Variables в **Инспекторе.**
- Удалите объект, с которым сталкиваются, с от текущей сцены используя `Разрушение` () команда. Используя точечный синтаксис снова, мы определили, что объект, который будет разрушен, является тем, в настоящее время сталкивался с. Поскольку этим управляют, `если` утверждение, вызывая это, только если столкнутой - с объектом является теговая `батарея`, тогда нет никакого шанса разрушения неправильного объекта. `Разрушение` () команда просто берет объект игры как свой главный параметр в этом случае, но это может также использоваться со вторым параметром после запятой, то есть, ценность плавания, чтобы представить задержку вовремя прежде, чем это удалит объект.

Теперь спасите сценарий, идя в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity. Выберите объект First Person Controller в **Иерархии,** и определите местонахождение **Столкновений Игрока (Сценарий)** компонент в **Инспекторе.** Вы должны теперь видеть, что есть **Батарея** Variables нового общественного участника, которой позвонили за счёт абонента, которая ждет назначения **Звукового** актива **Скрепки**.
Drag and drop (перетащили и опустили) звуковой актив скрепки, названный battery_collect от папки **Звуков** в **Проектной** группе, или нажмите на серость вниз стрелка направо ни от **Одного (Звуковая Скрепка),** и выберите battery_collect из раскрывающегося списка, который появляется.

Нажмите кнопку Play и проверьте игру, Вы должны теперь быть в состоянии идти характер игрока в каждую из батарей, и они должны исчезнуть, вызывая батарее GUI казаться onscreen и затем увеличить с каждой собранной батареей. Вы должны также услышать звуковой

эффект, играемый с каждой батареей, которую Вы собираете, так что удостоверьтесь, что аудио Вашего компьютера поднят! Нажмите **Игру** снова, чтобы прекратить проверять, как только Вы гарантировали, что Вы можете собрать все батареи.

Ограничение доступа заставы

Главный пункт этого осуществления должен продемонстрировать, как Вы можете управлять определенными ситуациями в пределах своего развития игры. В этом примере мы желаем позволить доступ игрока к заставе, только если они зарядили раздвижную дверь, собирая четыре батареи.

Эта загадка представляет нас как разработчиков с двумя главными проблемами, которые будут обращены:

- Как игрок знает, что они должны собрать батареи, чтобы войти в дверь?
- Как мы можем закодировать нашу игру, чтобы открыть дверь, только если все батареи были собраны?

Чтобы обратиться к первой проблеме, мы будем стремиться поддерживать элемент тайны о нашей игре. Мы только подарим игроку намек, чтобы собраться, батареи должен они быть достаточно любознательными, чтобы попытаться войти в заставу. Чтобы сделать это, мы должны будем добавить onscreen инструкции, когда к двери приблизятся впервые. Мы добавим дальнейшие инструкции, если к двери приблизятся снова, если игрок собрал некоторых, но не все требуемые батареи.

Чтобы проверить, может ли игрок открыть дверь - и поэтому, показать ли инструкции - мы можем использовать сценарий BatteryCollect, `который` мы уже написали. Поскольку у этого сценария есть количество батарей, собранных и сохраненных в статическом Variables, названном `обвинением`, мы можем подвергнуть сомнению ценность `обвинения`, чтобы видеть, равняется ли это 4 - значение, что все батареи были собраны.

Ограничение доступа

Прежде, чем мы дадим инструкции игрока, мы ограничим часть дверного проема нашего сценария PlayerCollisions, чтобы активизировать только, когда игрок собрал все эти четыре батареи. В предыдущей главе мы написали таможенный Functions в сценарии PlayerCollisions под названием `Дверь ()`, для которого мы могли накормить параметры, чтобы использовать это для открытия или закрытия дверей, с которыми столкнулся игрок. Это было похоже на это:

```
function Door(aClip : AudioClip, openCheck : boolean, animName : String,
  thisDoor     : GameObject){ audio.PlayOneShot(aClip); doorIsOpen =
  openCheck;
```

```
    thisDoor.transform.parent.animation.Play(animName);
}
```

Этот Functions играет звук, он гарантирует, что дверь не может быть вновь открыта, и затем играет мультипликацию его открытие. Мы вызывали эту функцию в пределах кода броска луча, который мы создали в Update () function (векторная линия, которая стоит перед тем же самым направлением как игрок, проверяющий на дверь). Чтобы напомнить Вам, это было похоже на это:

```
var hit     : RaycastHit;
  if       (Physics.Raycast       (transform.position, transform.forward, hit,
    5)) {
    if(hit.collider.gameObject.tag=="outpostDoor" && doorIsOpen ==
      false){
      currentDoor = hit.collider.gameObject;
      Door(doorOpenSound, true, "dooropen", currentDoor);
    }
  }
```

Откройте сценарий PlayerCollisions, щелкая два раза на этом в **Проектной** группе, и затем определите местонахождение последнего кодового отрывка в Update () function около вершины сценария, который называет `Дверь` () Functions, чтобы открыться:

```
if(hit.collider.gameObject.tag=="outpostDoor" && doorIsOpen ==
  false){
  Door(doorOpenSound, true, "dooropen", currentDoor);
}
```

Чтобы ограничить доступ перед, все эти четыре батареи собраны, просто добавляют третье условие к этому `если` утверждение при использовании другой пары ampersands-&&, как показано в следующем кодовом отрывке:

```
if(hit.collider.gameObject.tag=="outpostDoor" && doorIsOpen == false
  && BatteryCollect.charge >= 4){ Door(doorOpenSound, true,
  "dooropen", currentDoor);
}
```

Добавляя следующее условие:

```
BatteryCollect.charge> = 4
```

Мы проверяем `обвинение` сценария BatteryCollect статический Variables на ценность 4. Дверь не будет открываться, пока мы не попытаемся войти в дверь, собиравшую все батареи, которые мы

вынули.

Использование GetComponent ()

Когда мы получим доступ, мы должны будем удалить **Батарею GUI** из экрана. Чтобы сделать это, мы должны будем получить доступ к компоненту Структуры GUI на том объекте. Учитывая, что вызов двери находится в сценарии PlayerCollisions, мы в настоящее время продолжаем работать, мы должны будем обратиться к компоненту, который не находится на том же самом объекте как этот сценарий **(Первый Диспетчер Человека),** но вместо этого находится на объекте **GUI Батареи**. В то время как в сценарии BatteryCollect, мы могли просто сказать:

```
guiTexture.enabled = false;
```

Это - то, потому что это было присоединено к тому же самому объекту как компонент **GUITexture** - это не может быть сделано в сценарии, который не находится на объекте с компонентом **GUITexture**.

Это - то, где GetComponent () команда входит. Ссылаясь на объект, у которого есть специфический компонент, к которому Вы должны обратиться и сопровождаемый GetComponent () команда, Вы можете легко приспособить компоненты на внешних объектах.

Ниже линии, которая называет `Дверь` () Functions в, `если` утверждение Вы только изменились, добавьте следующую линию кода:

```
GameObject.Find("Battery
GUI").GetComponent(GUITexture).enabled=false;
```

Здесь, мы используем GameObject. `Найдите`, определяя название объекта в **Иерархии,** и тогда использующий точечный синтаксис - мы обращаемся к GUITexture как к параметру GetComponent (). Наконец, мы повреждаем компонент, устанавливая это в `ложный` в обычной манере - позволил = `ложный`;.

Мы не хотим разрушать объект, поскольку мы все еще нуждаемся в сценарии, чтобы существовать в пределах игры, так, чтобы мы могли сослаться на количество батарей, уже собранных. Это - то, почему мы просто повреждаем визуальный компонент.

Пойдите в **Файл |, Экономят** в редакторе сценария, и затем переключаются назад на Unity.

Нажмите кнопку **Play** теперь, чтобы проверить игру, и гарантировать,

что Вы не можете войти в дверь заставы не собрав все эти четыре батареи. Если дело обстоит не так, то перепроверьте свой код, чтобы гарантировать, что он соответствует тому, что мы обрисовали в общих чертах пока. Не забудьте нажимать **Игру** снова, чтобы прекратить проверять.

Намеки для игрока

Что, если игрок, менее заинтригованный батареями чем дверь заставы непосредственно, подходит к двери и пытается вступить? Мы должны идеально сделать две вещи:

- Скажите игроку в текстовой форме, что дверная зарядка потребностей - в то время как мы могли легко сказать, "Собирает некоторые батареи!", намного лучше в условиях gameplay обеспечить намеки, такие как "дверь, кажется, испытывает недостаток во власти..".
- Включите Батарею GUI так, чтобы они могли сказать, что они должны завысить цену батареи.

Последний тех двух намеков намного легче осуществить, таким образом мы обратимся что сначала.

Батарея намек GUI

Переключитесь назад на редактора сценария, где у Вас все еще есть открытый сценарий PlayerCollisions. Если Вы закрыли это по какой-нибудь причине, то помните, что Вы можете просто вновь открыть это от **Проектной** группы.

После заключительной вьющейся скобы, `если` утверждение мы продолжали работать, `еще` добавьте, `если` утверждение, что проверки на те же самые условия как оригинальный - кроме этого времени, мы проверим, является ли Variables `обвинения` BatteryCollect меньше чем 4:

```
else if(hit.collider.gameObject.tag=="outpostDoor" &&
  doorIsOpen == false && BatteryCollect.charge < 4){
}
```

В этом `еще`, `если` утверждение, мы будем использовать ту же самую линию, которую мы ранее добавили к первому, `если` утверждение, которое повредило компонент gui `Батареи` GUITexture, но на сей раз мы позволим это. `Еще` поместите следующую линию в `если` утверждение:

ТеперьGameObject.Find("Battery GUI").

GetComponent(GUITexture).enabled=true;

, пойдите в **Файл |, Экономят** в редакторе сценария и переключаются назад на Unity. Нажмите кнопку Play и тест, что, когда Вы приближаетесь

к двери без любых четырех батарей, Батареи, GUI появляется как визуальный намек. Нажмите **Игру** снова, чтобы прекратить проверять.

Текстовый намек GUI

Всякий раз, когда Вы должны написать текст на экране в 2-ом, самый прямой способ сделать, это при использовании компонента **GUIText**. Создавая новый объект Text GUI из главных меню, Вы получите новый объект и с Преобразовать и с Текстовые компоненты GUI. Создайте одного из них теперь, идя в **GameObject |, Создают Другой | Текст GUI.** У Вас должен теперь быть новый объект в **Иерархии** под названием **Текст GUI** и некоторый 2-ой текст на экране, который говорит **Гуй Текста,** как показано в следующем скриншоте:

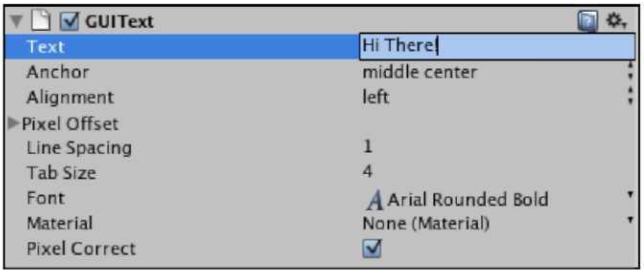
Переименуйте этот объект в **Иерархии**, выбирая это и нажимая *Возвращение* (Mac) или *F2* (PC). Назовите это **TextHint GUI.**

Выберите объект **Text GUI** в группе **Иерархии**, чтобы видеть компонент непосредственно в **Инспекторе.** Поскольку мы хотим сообщить игроку, мы оставим настоящее положение внутри компонент **Transform (преобразовать)** как **0.5** в **X** и **Текстовые** работы расположения **Y-GUI** в координатах экрана также, таким образом **0.5** в обоих топорах поместит это в середине экрана, требуя внимание игрока.

В дополнение к расположению всего элемента **Текст GUI** также показывает параметр **Выравнивания**, который работает в подобной манере к параметрам настройки оправдания в программном обеспечении обработки текста. Щелкните/вниз стрелки направо от

параметра **Выравнивания** и выберите **средний центр** - это означает, что текст распространится из центра экрана вместо того, чтобы начаться в центре и затем заполнить направо.

В то время как Вы можете легко напечатать то, что Вы желаете, чтобы этот **Текстовый** элемент **GUI** сказал в **Текстовом** параметре в **Инспекторе** (см. следующий скриншот) - мы вместо этого собираемся управлять тем, что это говорит динамически, через scripting.



Создайте новый файл JavaScript, выбирая папку **Сценариев** в **Проектной** группе. От опускаться меню кнопки **Create**, выберите **JavaScript.**

Переименуйте **NewBehaviourScript, который** это создает к **TextHints,** и затем щелкать два раза его изображением, чтобы начать это в редакторе сценария.

Начните сценарий, объявляя три variables:

```
static var textOn :     boolean = false;
static var message     : String; private
var timer     :     float = 0.0;
```

Наш первый статический Variables-textO - булевое, поскольку он должен просто действовать как выключатель. Мы покроем больше на этом коротко.

Тогда, мы устанавливаем тип последовательности статический Variables, названный `сообщением`, которое мы будем использовать, чтобы передать любую информацию к `Текстовому` параметру компонента. Эти два variables статичны, поскольку мы должны будем обратиться к ним от отдельного сценария, PlayerCollisions.

Наш третий Variables - таймер - является частным, поскольку это не должно быть обращено другими сценариями, ни замечено в **Инспекторе.** Мы будем использовать этот `таймер`, чтобы рассчитать с момента, сообщение кажется onscreen, чтобы заставить это исчезнуть после определенного количества времени.

Затем, добавьте следующее `Начало` () Functions, чтобы установить определенные государства, когда сцена начинается:

```
function Start(){ timer =
  0.0; textOn = false;
  guiText.text = "";
}
```

Здесь, мы просто гарантируем, что таймер установлен в 0, что textOn Variables ложен (мы не должны видеть инструкцию, когда игра начинается), и что никакой текст в настоящее время не находится в текстовом параметре guiText компонента. Мы делаем это, просто используя две кавычки без текста внутри. Это известно как **пустая последовательность.**

Затем, приложите следующий код к Update () function, который уже существует в сценарии:

```
function Update       () {
  if(textOn){
    guiText.enabled          =          true;
    guiText.text  =  message;  timer  +=
    Time.deltaTime;

  }
  if(timer >=5){ textOn = false;
    guiText.enabled = false; timer = 0.0;

  }
}
```

Эта простая двухступенчатая процедура делает следующее:

- Проверки на textOn Variables, чтобы стать верный. Когда это, это:
  - ° Включает компонент **GUIText**
  - ° Устанавливает текстовые параметры, равные вызванному Variables последовательности
    сообщение
  - ° Начинает увеличивать Variables таймера, используя Time (Времени). команда deltaTime
- Проверки на Variables таймера, чтобы достигнуть ценности 5 секунд. Если это имеет, то это:
  - ° Наборы textOn булевое к ложному (значение первого, если утверждение больше не будет действительно),
  - ° Повреждает компонент **GUIText**
  - ° Сброс Variables таймера к 0.0

Пойдите в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity. Выберите объект GUI TextHint в **Иерархии** и пойдите в **Компонент | Сценарии | TextHints**, чтобы добавить сценарий, который Вы только что написали объекту.

У нас теперь есть сценарий, который управляет нашим TextHint GUI, и всем, что мы должны сделать, более аккуратно это в действие, вызывая статические variables, которые это содержит. Учитывая, что момент, мы желаем вызвать инструкции, связан со столкновениями с дверью, мы обратимся к этим статическим variables от открытого для сценария PlayerCollisions это в редакторе сценария теперь.

Еще найдите, если утверждение, которое мы последний раз добавили к Update () function этого сценария, который сопровождает обнаружение броска луча двери, и добавляют следующие две линии:

```
TextHints.message = "дверь, кажется, должен более
двинуться на большой скорости.."; TextHints.textOn =
верный;
```

Здесь мы посылаем некоторый текст в сообщение статический Variables сценария TextHints и устанавливаем textOn логическую переменную того же самого сценария в истинный. Вот еще полное если утверждение:

```
else if(hit.collider.gameObject.tag=="outpostDoor" &&
  doorIsOpen == false && BatteryCollect.charge < 4){
  GameObject.Find("Battery GUI").GetComponent(GUITexture).
  enabled=true;
  TextHints.message = "The door seems to need more power..";
  TextHints.textOn = true;
}
```

Поскольку мы включили компонент, устанавливая textOn к истинному, наш сценарий TextHints сделает остальных! Пойдите в **Файл |, Экономят** в редакторе сценария, затем переключаются назад на Unity и **Игру** прессы, чтобы проверить игру. Приблизьтесь к двери без четырех необходимых батарей, и Вам покажут пять секунд основанного на тексте намека на экран. Нажмите **Игру** снова, чтобы прекратить проверять игру.

Пойдите в **Файл |, Экономят** в Unity, чтобы обновить Ваше продвижение.

Наконец, давайте улучшим вид нашего onscreen текстового намека, добавляя наш собственный шрифт.

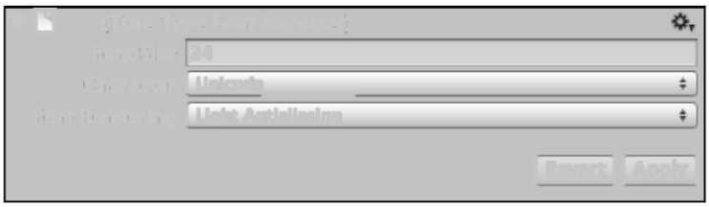
Используя шрифты

Используя шрифты в любом проекте Unity, они должны быть импортированы как актив таким же образом как любая другая часть

СМИ, которые Вы включаете. Это может быть сделано, просто добавляя любой **TTF (шрифт TrueType)** или **OTF (шрифт OpenType)** файл к Вашей папке **Активов** в Искателе (Mac) или Windows Explorer (PC) или в Unity непосредственно, идя в **Активы | Импорт Новый Актив.**

<http://www.dafont> или этот пример, я буду использовать коммерчески свободный к использованию шрифт от www.dafont <http://www.dafont>. com, который является вебсайтом свободных к использованию шрифтов, который очень полезен когда начинающийся в любом виде типографского проекта. Знайте, что некоторые вебсайты шрифта предоставляют шрифтам ограничение, сообщая, что Вы должны использовать их в проектах, которые могли сделать шрифт извлекаемым - это не проблема с Unity, поскольку все шрифты преобразованы в структуры в экспортируемом, строят из игры.

Посетите этот участок теперь и загрузите шрифт, взгляд которого Вы любите, и который легок читать. Помните, Вы будете использовать этот шрифт, чтобы дать инструкции, таким образом что-нибудь чрезмерно сложное будет противоинтуитивно игроку. Если Вы хотели бы использовать тот же самый шрифт, который я использую, затем искать шрифт по имени **Sugo.** Загрузите шрифт, расстегните молнию на нем, и затем используйте методы, только обрисованные в общих чертах, чтобы добавить файл Sugo.ttf как актив.



Как только это находится в Вашем проекте, найти шрифт в **Проектной** группе и выбрать это. Тогда, чтобы облегчить читать, мы увеличим размер шрифта для того, всякий раз, когда этот шрифт используется. В **Инспекторе** для шрифта **Sugo** при **Истинном Импортере Шрифта Типа** (обращаются к следующему скриншоту), установите **Размер Шрифта** в **24** и затем нажмите кнопку **Apply**. Если Вы выбрали собственный шрифт, знают, что калибровка может отличаться, но Вы можете возвратить и изменить **Размер Шрифта**, устанавливающий в любое время.

Затем, выберите это как шрифт, чтобы использовать для **TextHint GUI**, выбирая это в группе **Иерархии** и затем выбирая Ваш шрифт из опускаться меню направо от параметра **Шрифта** компонента **GUIText** в

Инспекторе. Альтернативно, помните, что Вы можете также drag and drop (перетащили и опустили), чтобы назначить в Unity, при перемещении актива шрифта от **Проектной** группы и понижая это на этот параметр в **Инспекторе.**

Теперь нажмите кнопку **Play**, и проверьте Вашу игру. На сей раз, когда Вы приближаетесь к двери с меньше чем четырьмя батареями, Ваше сообщение появится на экране. Когда Вы больше не столкнетесь с дверью, это исчезнет после этих пяти секунд мы определили ранее. Нажмите **Игра** снова, чтобы прекратить проверить игру и идти в **Файл | Экономят**, чтобы обновить Ваше продвижение

Резюме

В этой главе мы успешно создали и решили сценарий игры. Оценивая, что Ваш игрок будет ожидать видеть в игре, что Вы представляете им - за пределами Вашего предшествующего знания ее работ - Вы можете лучше всего разработать подход, который Вы должны проявить как разработчик.

Попытайтесь считать каждый новый элемент в своей игре от перспективной игры игрока существующими играми, думать о сценариях реального мира, и больше всего, не принять предшествующее знание. Самый интуитивный gameplay всегда находится в играх, которые ударяют баланс между трудностями в достижении набора задач и должным образом оборудовании игрока для задачи с точки зрения информации и дружественных отношений с намеченным подходом. Соответствующая обратная связь для игрока крайне важна здесь, быть этим визуальный или звуковой базирующийся всегда рассматривают, какую обратную связь игрок имеет всегда, проектируя любую игру.

Теперь, когда мы исследовали основной сценарий игры и смотрели на то, как мы можем построить и управлять элементами GUI, в следующей главе, мы будем идти дальше к более продвинутому сценарию игры, в котором мы будем смотреть на два более решающих понятия - физика **Твердого тела** и **Экземпляр**.

Экземпляр и Твердые тела

В этой главе мы будем исследовать два решающих понятия в трехмерном

проекте игры. В первой половине мы будем смотреть на понятие **Экземпляра** - процесс создания объектов во время времени выполнения. Мы тогда исследуем практический пример экземпляра, поскольку мы узнаем об использовании физики **Твердого тела**.

Когда Вы сначала начнете строить сцены игры, Вы поймете, что не все объекты, требуемые в пределах любой данной сцены, присутствовали бы в начале игра. Это верно для большого разнообразия жанров игры как головоломки, такие как *Tetris.* Части загадки случайных форм созданы или *иллюстрируются примерами* наверху экрана в интервалах набора, потому что все они не могут быть сохранены наверху экрана бесконечно.

Теперь возьмите нашу игру исследования острова как другой пример. В этой главе мы будем смотреть на физику твердого тела, создавая метод для нашего характера игрока, чтобы играть в простую кокосовую застенчивую игру, но кокосовые орехи, которые будут брошены, не будут присутствовать в сцене игры, когда это начнется. Это - то, где экземпляр входит снова. Определяя актив (наиболее вероятно prefab объекта), позиция, и вращение, объекты могут быть созданы, в то время как в игру играют - это позволит нам создавать новый кокосовый орех всякий раз, когда игрок нажимает кнопку огня.

Чтобы связать эту новую часть нашей игры в игру как, это стоит, мы будем удалять одну из батарей, которая обязана входить в заставу от игры. Мы дадим игроку шанс выиграть заключительную батарею, которой они требуют, играя в кокосовую застенчивую игру. Вам предоставят модель кокосовой застенчивой платформы. Вы должны будете создать кокосовый prefab и управлять мультипликацией целей в пределах игры через scripting-обнаружение столкновений между кокосовыми орехами и целями.

Как цели мы обеспечиваем, у Вас с есть "сбитый" и мультипликация 'сброса', мы также напишем scripting, чтобы гарантировать, что цели перезагружали, будучи сбитым для определенного числа секунд. Это означает, что игрок может выиграть эту миниигру, только если все три цели снижаются в то же самое время. Это добавляет дополнительный слой навыка для игрока.

В этой главе Вы изучите следующее:

- Подготовка prefab для экземпляра
- Команда экземпляра осуществлена
- Проверка игрока введена
- Добавление физики к объекту с твердыми телами
- Обеспечение обратной связи для игрока

Представление экземпляра

В этой секции мы узнаем, как породить и дублировать объекты, в то время как игра работает. Это - понятие, которое используется во многих играх, чтобы создать снаряды, коллекционируемые объекты, и даже характеры,

такие как враги.

В понятии
Экземпляр - просто метод создания объекты от шаблона (prefab в сроках Unity) во время времени выполнения. Это может также использоваться, чтобы дублировать существующие объекты игры уже в сцене.

Подход, используя экземпляр будет обычно принимать эту форму:

- Создайте объект, который Вы желаете иллюстрировать примерами в Вашей сцене, и добавить компоненты по мере необходимости
- Создайте новый prefab в своем проекте, и понизьте объект, Вы продолжали работать в тот prefab
- Удалите оригинальный объект из сцены так, чтобы это было только сохранено как заранее приготовленный актив
- Напишите сценарий, который вовлекает Instantiate (Проиллюстрировать) () команда, приложить это к активному объекту игры, и установить prefab, который Вы создали как объект что
  Instantiate (Проиллюстрировать) (), команда создает

В коде
В его ядре Instantiate (Проиллюстрировать) () команда имеет три параметра и написана следующим образом:

```
Instantiate (Проиллюстрировать) (объект, чтобы создать,
  поместить, чтобы создать это, вращение, чтобы дать
  это);
```

Понимая, как назначить эти три параметра, Вы будете в хорошем земельном участке, чтобы заняться любым использованием Instantiate (Проиллюстрировать) (), приказывают, чтобы Вы могли столкнуться.

Прохождение в объекте
Чтобы пройти в объекте, Вы можете просто питаться, общественное имя переменной участника к первому параметру Instantiate (Проиллюстрировать) (). Создавая общественный Variables участника типа GameObj ect наверху сценария, Вы будете в состоянии drag and drop (перетащили и опустили) prefab на это в **Инспекторе** Unity и затем использовать этот Variables как объектный параметр, следующим образом:

```
var myPrefab : GameObject;
Instantiate(myPrefab,    position to create it,    rotation to give it);
```

Мы можем также гарантировать, что только определенные типы prefab могут быть брошены в этот myPrefab Variables, будучи более определенными в его печатании данных, например, мы могли сказать:

```
var myPrefab : Rigidbody;
```

Instantiate(myPrefab, position to create it, rotation to give it);

Это гарантирует, что только заранее приготовленные объекты с компонентом Rigidbody могут использоваться.

Позиция и вращение

Позиция и вращение объектов, которые будут созданы, должны быть определены как ценности Vector3 (X, Y, Z). Их можно передать непосредственно следующим образом:

```
Instantiate (Проиллюстрировать) (myPrefab, Vector3 (0, 12, 30), Vector3 (0, 0, 90));
```

Они могут также быть унаследованы от другого объекта, беря ценности, которые представляют его позицию.

Назначая позицию объекта иллюстрироваться примерами, Вы должны рассмотреть, где Ваш объект будет создан и должно ли это быть создано в местном космическом или мировом месте.

Например, создавая наши кокосовые prefab, мы будем создавать их в пункте в мире, определенном пустым объектом игры, который будет ребенком нашего объекта характера игрока. В результате мы можем сказать, что это будет создано в местном месте - не в том же самом месте каждый раз, а относительно того, где наш характер игрока выдерживает и стоит.

Это решение помогает нам решить, где написать наш код, то есть, к который объект приложить сценарий. Прилагая сценарий к пустому объекту, который представляет позицию, где кокосовые орехи должны быть созданы, мы можем просто использовать точечный синтаксис и ссылку transform.position как позиция для Instantiate (Проиллюстрировать) () команда. Делая это, созданный объект наследует позицию компонента пустого объекта Transform, потому что это - то, к чему присоединен сценарий. Это может быть сделано для вращения, также дающего недавно порожденный объект вращение, которое соответствует пустому родительскому объекту.

Экземпляр и Твердые тела

Это дало бы нам, Instantiate (Проиллюстрировать) () приказывает, чтобы был похож на это:

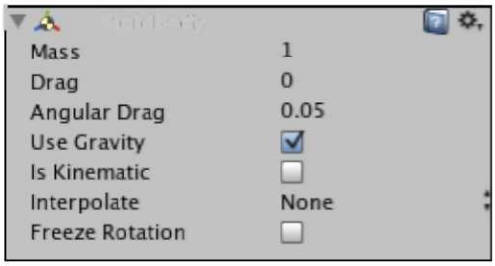
```
var myPrefab : GameObject;
Instantiate(myPrefab, transform.position, transform.rotation);
```

Мы проведем в жизнь это позже в главе, но сначала позволим нам смотреть на физику твердого тела и ее важность в играх.

Представление твердых тел
Двигатели физики дают играм средство моделирования реализма в физических сроках, и они - особенность в почти всех двигателях игры или прирожденно или как плагин. Unity использует **Nvidia PhysX** двигатель физики, точный современный двигатель физики, который используется во многих коммерческих играх в промышленности. Имея двигатель физики означает, что не только физические реакции, такие как вес и сила тяжести являются возможными, но реалистическими ответами на трение, вращающий момент, и основанное на массе воздействие, также возможны.

Силы
Влияние двигателя физики на объектах известно как **сила,** и силы могут быть применены во множестве путей через компоненты или scripting. Чтобы применить силы физики, объект должен быть тем, что известно как объект твердого тела.

Компонент Rigidbody
Чтобы призвать двигатель физики в Unity, Вы должны дать объекту rigidbody компонент. Это просто говорит двигателю применять двигатель физики к специфическому объекту - что Вы не должны применить это ко всей сцене. Это просто работает на заднем плане.



Добавив компонент **Rigidbody**, Вы видели бы параметры настройки для этого в **Инспекторе** таким же образом как любой другой объект, как показано в следующем скриншоте:
У компонентов Rigidbody есть следующие параметры, которые будут регулироваться или управляться через scripting:

- **Масса:** вес объекта в килограммах. Примите во внимание, что урегулирование массы на множестве различных твердых тел заставит их вести себя реалистично. Например, тяжелый объект, поражающий более легкий объект, заставит легкий объект быть отраженным далее.

- **Бремя:** Бремя, как в реальном исчислении, является просто количеством сопротивления воздуха, затрагивающего объект, поскольку это перемещается. Чем выше ценность, тем более быстрое желание объект замедляется когда просто затронуто воздушным путем.
- **Угловое Бремя:** Подобный предыдущему параметру, но угловому бремени просто затрагивает вращательную скорость, определяя, сколько воздуха затрагивает объект, замедляя это к вращательной остановке.
- **Используйте Силу тяжести:** Делает точно, поскольку это сообщает. Это - урегулирование, которое определяет, будет ли объект твердого тела затронут силой тяжести или нет. С этим инвалидом опции объект будет все еще затронут силами и воздействиями от двигателя физики и будет реагировать соответственно, но как будто в невесомости.
- **Является Кинематическим:** Эта опция позволяет Вам иметь объект твердого тела, который не затронут двигателем физики. Например, если Вы желали иметь объект, отражают твердое тело с силой тяжести на, такой как спусковой механизм в автомате для игры в пинбол, поражающем шар - но без воздействия, заставляющего спусковой механизм быть тогда затронутым, Вы могли бы использовать это урегулирование.
- **Интерполируйте:** Это урегулирование может использоваться, если Ваши объекты твердого тела дрожат. Вставка и экстраполяция могут быть выбраны, чтобы пригладить движение transform (преобразовать), основанное на предыдущей структуре, или предсказали следующую структуру соответственно.
- **Вращение Замораживания:** Это может использоваться, чтобы захватить объекты так, чтобы они не вращались в результате сил, примененных двигателем физики. Это особенно полезно для объектов, которые должны использовать силу тяжести, но вертикальное пребывание, такое как характеры игры.

Создание миниигры

Чтобы осуществить, на что мы только что смотрели, мы создадим кокосовую застенчивую игру, которая набрасывается на наш доступ к заставе. Играя в игру, игрок будет вознагражден с заключительной батареей, которой они требуют, чтобы зарядить дверь заставы.

Поскольку мы уже настроили элемент заряда батареи игры, мы просто должны удалить одну из батарей от существующей сцены, оставляя игрока с одним меньше.

Выберите один из объектов, названных **батареей** в группе **Иерархии**, и затем удалите это с *Командой + Клавиша Backspace* (Mac) или *Удалите* (PC).

Создание кокосового prefab

Теперь, когда мы узнали об экземпляре, мы начнем нашу миниигру,

создавая объект, который будет брошен, то есть, кокосовый орех.

Пойдите в **Объект Игры |, Создают Другой | Сфера.**

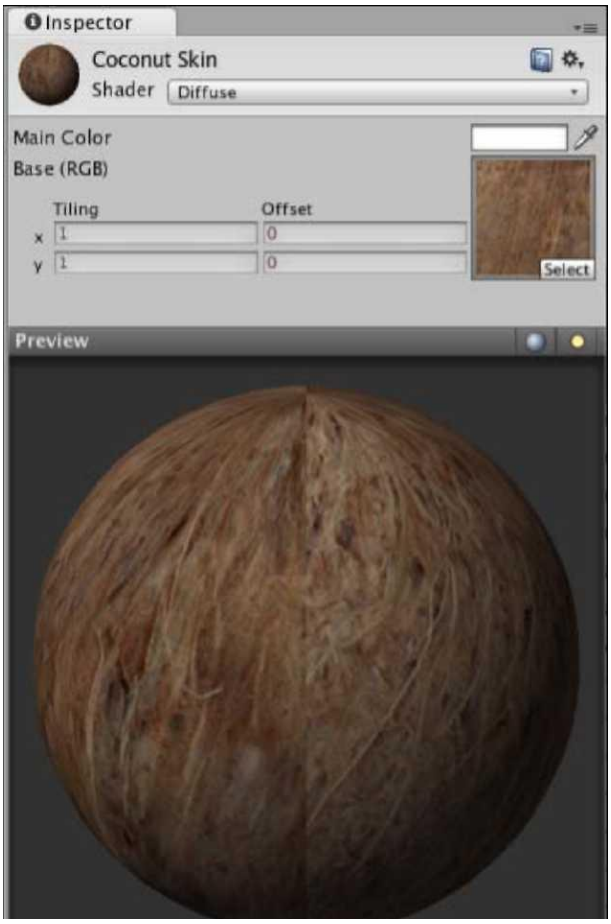
Это создает новую сферу примитивный объект в сцене. В то время как это не будет создано непосредственно перед редактором viewport, Вы можете легко изменить масштаб изображения к этому, толпясь Ваш курсор по представлению **Сцены** и нажимая *F* (центр) на клавиатуре. Переименуйте этот объект от **Сферы** до **Кокосового ореха,** выбирая объект в **Иерархии** и нажимая *Возвращение* (Mac) или *F2* (PC) и перепечатывание.

Затем, мы сделаем этот объект более соответствующим размером и формой для кокосового ореха, сокращаясь и тонко расширяя его размер в Оси Z. В компоненте **Transform (преобразовать) Инспектора** для объекта **Coconut**, измените ценность **Масштаба** для **X** и **Z** к **0.5** и **Y** к **0.6.**
Структуры, которые будут использоваться на кокосовом орехе, доступны в кодовой связке, обеспеченной на packtpub.com <http://packtpub.com> (www.packtpub.com/files/code/8181_Code.zip). Определите местонахождение пакета по имени CoconutGame.unitypackage.
Импортируйте этот пакет, идя в **Активы | Пакет Импорта.** Проведите к файлу, Вы извлекли, выбираете его, и подтверждаете **Импорт.**

У Вас должна теперь быть папка в Вашей **Проектной** группе под названием Кокосовая Игра, содержащая следующее:

- Кокосовая структура изображения
- Структура перекрестия
- Четыре звуковых скрепки
- Две трехмерных модели - одна вызванная **платформа,** и другая вызванная **цель**
- Папка Материалов для трехмерных моделей

Создание текстурированного кокосового ореха
Чтобы применить кокосовую структуру, мы должны будем сделать новый материал, чтобы применить это к. На **Проектной** группе, нажмите на кнопку **Create**, и от опускаться меню, которое появляется, выбрать **Материал.** Переименуйте новый материал, который Вы сделали к **Кокосовой Коже**, нажимая *Возвращение* (Mac) или *F2* (PC), и перепечатывание.
Чтобы применить структуру к этому материалу, просто drag and drop (перетащили и опустили) **кокосовую** структуру от **Проектной** группы к пустому квадрату направо от **Основы (RGB)**, устанавливающий для материала в **Инспекторе.** Когда это сделано, Вы должны видеть предварительный просмотр материала в окне в более низкой половине **Инспектора,** как показано в следующем скриншоте:



Помните, что этот предварительный просмотр просто демонстрирует внешность материала на сфере - это - неплатеж для предварительных просмотров материала Unity и не имеет никакого отношения к факту, что мы будем использовать это на сферическом объекте.

Затем, мы должны применить **Кокосовый** материал **Кожи** к **Кокосовому** объекту игры, который мы поместили в нашу сцену. Чтобы сделать это, мы просто drag and drop (перетащили и опустили) материал от окна **Project** или до имени объекта в **Иерархии** или к объекту непосредственно в представлении **Сцены**.

Добавление физики

Теперь, потому что наш **Кокосовый** объект игры должен вести себя реалистично, мы должны будем добавить компонент Rigidbody, чтобы призвать двигатель физики для этого объекта. Выберите объект в **Иерархии** и пойдите в **Компонент | Физика | Rigidbody.**

Это добавляет компонент Rigidbody, который применит силу тяжести. В результате, когда игрок бросает объект вперед, он упадет в течение долгого времени, поскольку мы ожидали бы это к в действительности. Установки по умолчанию компонента Rigidbody можно оставить

неприспособленными, таким образом Вы не должны будете изменять их на данном этапе.

Мы можем проверить это кокосовые падения и рулоны теперь, нажимая кнопку **Play** и наблюдая объект в представлениях **Сцены** или **Игры**. Когда Вы удовлетворены, нажмите кнопку **Play** снова, чтобы прекратить проверять.

Экономия как prefab

Теперь, когда наш кокосовый объект игры полон, мы должны будем сохранить его как prefab в нашем проекте, чтобы гарантировать, что мы можем иллюстрировать примерами его использующий код так много раз, как нам нравится, вместо того, чтобы просто иметь единственный объект.

На **Проектной** группе, выберите папку **Prefab**, и затем нажмите на кнопку **Create** и выберите **Prefab.** Переименуйте новый заранее приготовленный **Кокосовый Prefab.** Drag and drop (перетащили и опустили) **Кокосовый** объект игры от **Иерархии** на новый prefab в **Проектной** группе, чтобы спасти это, и затем удалить оригинальную копию со сцены, выбирая это в **Иерархии** и нажимая *Команду + Клавиша Backspace* (Mac) или *Удалить* (PC).

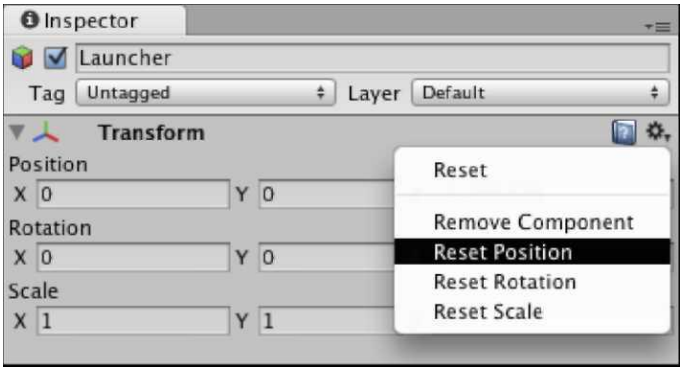
Создание объекта Launcher

Поскольку мы собираемся позволить нашему игроку бросать кокосовые prefab, мы только что создали, мы будем нуждаться в двух вещах-а сценарий, чтобы обращаться с экземпляром и пустым объектом игры, чтобы действовать как контрольная точка для позиции, чтобы создать объекты в мире.

В действительности, когда мы бросаем шар (стиль сверхруки), он входит в представление стороне нашей головы, поскольку наша рука выступает вперед, чтобы выпустить шар. Поэтому, мы должны поместить объект только за пределами области нашего игрока представления и удостовериться, что это будет следовать везде, где они смотрят. Поскольку представление игрока обработано **Главным** детским объектом **Камеры Первого Диспетчера Человека,** делая пустой объект, который ребенок этого позволит этому перемещать с камерой, поскольку ее позиция будет всегда оставаться относительно ее родителя.

Начните, создавая новый пустой объект игры в Вашей сцене, идя в **GameObject |, Создают Пустой.** Выберите этот новый объект в **Иерархии** (по умолчанию, это назовут **GameObject)**, и переименуйте это **Пусковая установка.** Затем, расширьте объект **First Person Controller**, нажимая на

серую стрелку налево от ее имени в **Иерархии.** Теперь drag and drop (перетащили и опустили) объект **Launcher** на **Главную Камеру** так, чтобы это стало ребенком этого - Вы будете знать, что Вы сделали это правильно, если серая стрелка появляется рядом с **Главной Камерой,** указывая, что это может быть расширено, чтобы показать ее детские объекты. Объект **Launcher** заказан ниже этого, как показано в следующем скриншоте:
Делая этот объект, ребенок **Главной Камеры** будет подразумевать, что это перемещается и вращается с ее родителем, но это все еще нуждается в перерасположении. Начните, перезагружая позицию объекта **Launcher** в компоненте **Transform (преобразовать)** в **Инспекторе.** Это может быть сделано или заменяя все ценности **0,** или экономить время, Вы можете использовать кнопку Cog, чтобы перезагрузить, выбирая **Позицию Сброса** из меню популярности, как показано в следующем скриншоте:



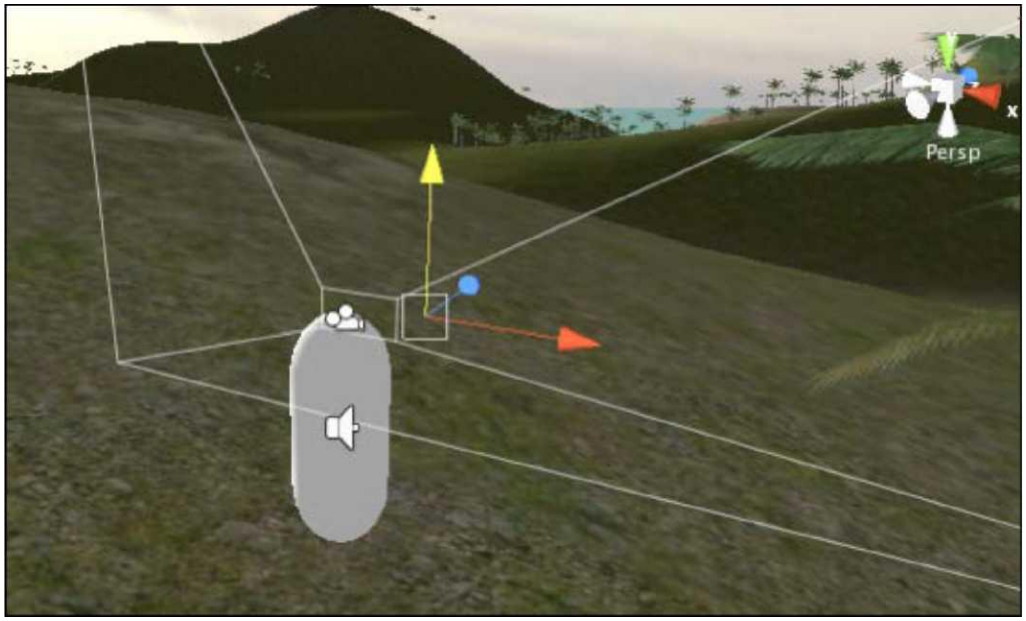
Кнопка Cog действует как быстрый способ выполнить операции на **Я**
" (Компоненты *Л,* и Вы будете видеть один рядом с каждым компонентом в **Инспекторе.**
Они также полезны, поскольку они позволяют Вам удалять компоненты Вы не
более длинная потребность или которые были добавлены по ошибке.

Урегулирование ребенка **Пусковой установки** возражает против позиции **0** средств, это находится точно в центре (та же самая позиция) как ее родитель. Конечно, это не то, что мы хотим, но это - хорошая отправная точка, чтобы пойти от. Мы не должны оставить пусковую установку в этой позиции по двум причинам:

- Когда кокосовые орехи брошены, они, казалось бы, прибывали бы из головы игрока, и это будет выглядеть странным.
- Когда Вы иллюстрируете примерами объекты, Вы должны гарантировать, что они не созданы в позиции, где их коллайдер пересечется с другим коллайдером, потому что это вынуждает

двигатель физики выдвинуть коллайдеры обособленно и могло прервать силу, примененную, бросая кокосовый орех.

Чтобы избежать этого, мы просто должны продвинуть объект **Launcher** и направо от его настоящего положения. В компоненте **Transform (преобразовать)**, набор **X** и позиции **Z** к ценности **1.** Ваш объект **Launcher** должен теперь быть помещен в пункте, где Вы ожидали бы выпускать объект, брошенный правой рукой характера игрока, как показано в следующем скриншоте:



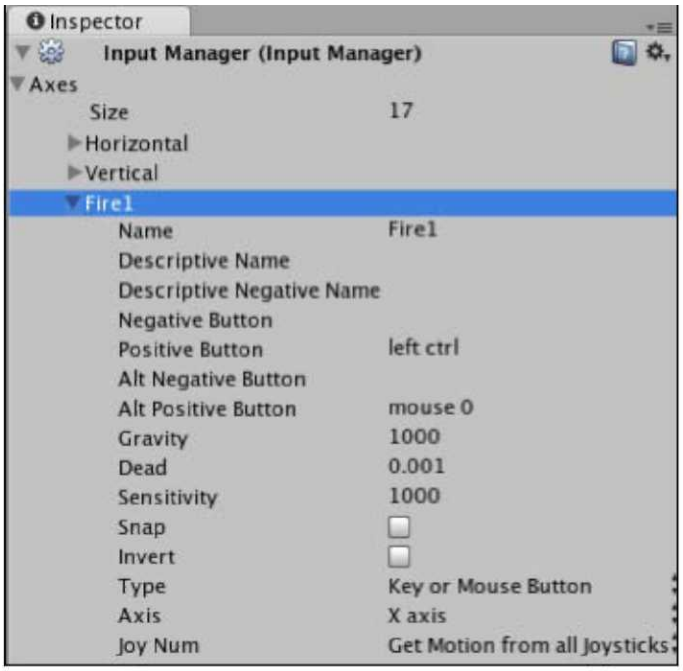
Наконец, чтобы заставить брошенный кокосовый орех направиться к центру нашего взгляда, мы должны вращать пусковую установку немного вокруг Оси Y. При **Вращении** в компоненте **Transform (преобразовать)**, поместите в ценности **352**, чтобы вращать это 8 степенями.

Затем, мы будем нуждаться к сценарию в экземпляре кокосового ореха и его толчка, когда игрок нажмет кнопку огня.

Броски кокосового ореха Scripting

Поскольку мы должны начать кокосовые орехи, когда игрок нажмет огонь, мы должны будем проверить, нажимают ли они ключ - привязанный к входу в Unity - каждая структура. Ключи и топоры/кнопки мыши привязаны, чтобы не выполнить своих обязательств названные входами во **Входном Менеджере** Unity**,** но они могут быть изменены на Вашем досуге, собираясь **Редактировать | Проектные Параметры настройки | Вход.** Сделайте это теперь, затем расширьте **Топоры**, щелкая серой стрелкой налево от нее, и затем наконец расширьте вход оси по имени **Fire1.**

Три решающих области, чтобы наблюдать вот являются параметром **Названия**, **Положительным** параметром, и **Высоким звуком Положительный** параметр. Мы будем обращаться к этой оси ее именем, и **Положительными** и **Положительным Высоким звуком** являются фактические ключи непосредственно, чтобы прислушаться. Новые топоры могут быть созданы, просто увеличивая ценность **Размера** наверху **Входа**, Получающегося менеджером в новом добавляемом входе, который Вы можете тогда настроить.



Для игрока, чтобы начать кокосовые орехи в целях - который мы поместим в нашу сцену позже - у них должен быть сценарий, который осуществляет два ключевых шага:

- Экземпляр объекта **Coconut**, который будет брошен в прессу кнопки огня
- Назначение скорости к компоненту Rigidbody продвинуть кокосовый орех вперед, как только это было создано

Чтобы достигнуть этого, сначала создайте новый файл JavaScript:

- Выберите папку **Сценариев** в **Проектной** группе
- Нажмите на кнопку **Create**, опускаются меню и выбирают **JavaScript**
- Переименуйте **NewBehaviourScript** к **CoconutThrow**, нажимая *Возвращение* (Mac) или *F2* (PC)

- Начните это в редакторе сценария, щелкая два раза его изображением

Проверка игрока введена
Учитывая, что мы должны прислушаться к нажатиям клавиш игрока каждая структура, мы должны написать наш код для пусковой установки в Update () function. Переместите заключительную правильную вьющуюся скобу} этого Functions вниз несколькими линиями, и затем добавьте следующий если утверждение, чтобы прислушаться к Fire1 keypress:

```
if(Input.GetButtonUp("Fire1")){ }
```

Это проверяет Входной Classes и ждет кнопок, привязанных к входу **Fire1** (Левая *клавиша CTRL* и оставленная кнопка мыши), чтобы быть выпущенным. В это, если бы утверждение, мы должны будем поместить действия, мы хотели бы, чтобы сценарий выступил, когда игрок выпускает любую кнопку.

Во-первых, мы должны играть звук, который действует как основанная на аудио обратная связь для того, чтобы бросить. Если бы мы создавали стреляющую игру, то это вероятно было бы звуком запускаемого оружия. Однако, в этом случае, у нас просто есть тонкий звук whooshing, чтобы представить запуск кокосового ореха.

Игра звука обратной связи
Мы будем нуждаться в Variables, чтобы представить звуковую скрепку, которую мы должны играть, так поместите следующий общественный Variables участника в очень главный из сценария прежде, чем мы продолжим:

```
var throwSound : AudioClip;
```

Это создает то, что известно как общественный Variables участника, что означает, что мы будем в состоянии назначить фактическую звуковую скрепку на этот Variables, используя **Инспектора**, как только мы закончены, сочиняя этот сценарий.

Теперь, давайте настраивать игру этой звуковой скрепки в нашем если утверждение. После вводной вьющейся скобы, добавьте следующую линию:

```
audio.PlayOneShot(throwSound); Это будет играть звук,
```

поскольку игрок выпускает кнопку **Fire1**.

Instantiate (Проиллюстрировать) кокосового ореха
Затем, мы должны создать фактический кокосовый орех непосредственно, также в пределах потока если утверждение. Учитывая, что мы создали кокосовый орех и сохранили его как prefab, мы должны установить другой общественный Variables участника так, чтобы мы могли назначить наш

prefab на Variables в **Инспекторе** позже. Наверху сценария, ниже Вашей существующей throwSound линии Variables, помещают следующее:

```
var coconutObject     : Rigidbody;
```

Это помещает в общественном Variables участника с типом данных Rigidbody. Хотя наш кокосовый орех сохранен как заранее приготовленный актив, когда мы будем иллюстрировать примерами его, мы будем создавать объект игры с Rigidbody, приложенным в нашей сцене, и поэтому типе данных. Это гарантирует, что мы не можем тянуть объект non-rigidbody к этому Variables в **Инспекторе.** Строго печатанием данных к Rigidbody это также означает, что, если бы мы желаем обратиться к компоненту Rigidbody этого объекта, тогда мы не должны были бы использовать GetComponent () команда, чтобы выбрать компонент Rigidbody сначала - мы можем просто написать код, который говорит непосредственно с Rigidbody.

Теперь, в, если утверждение в Update (), поместите следующую линию ниже существующей звуковой линии:

```
var newCoconut     :     Rigidbody = Instantiate(coconutObject,
   transform.position, transform.rotation);
```

Здесь, мы устанавливаем частный Variables, вызванный, newCoconut-это является частным, потому что это в пределах Update () function, и так не должно быть неявно написано с частной приставкой. В этот Variables мы передаем создание (Экземпляр) нового
GameObject-и поэтому тип данных.

Помните, что три параметра Instantiate (Проиллюстрировать) (), объект, позиция, и вращение. Вы будете видеть, что мы использовали общественный Variables участника, чтобы создать случай нашего prefab и затем унаследовали позицию и вращение от объекта, этот сценарий приложен к - объект **Launcher**.

Обозначение случаев

Всякий раз, когда Вы создаете объекты во время времени выполнения с, Instantiate (Проиллюстрировать) (), Unity берет название prefab и следует, это с текстом "клонируется", называя новые случаи. Поскольку это - скорее неуклюжее название к ссылке в коде - который мы должны будем сделать для наших целей позже - мы можем просто назвать случаи, которые созданы, добавляя следующую линию ниже линии, которую мы только добавили:

```
newCoconut.name = "coconut";
```

Здесь, мы просто использовали имя переменной, которое мы создали, который посылает к новому случаю prefab, и используемому точечному синтаксису обратиться к параметру названия.

Назначение скорости

В то время как этот Variables создаст случай нашего кокосового ореха, наш сценарий еще не полон, поскольку мы должны назначить скорость на недавно созданный кокосовый орех также. Иначе, это будет просто создано и падение к основанию. Чтобы позволить нам регулировать скорость брошенного кокосового ореха, мы можем создать другой общественный Variables участника наверху нашего сценария, чтобы обращаться с этим. Чтобы дать нам точность, мы сделаем этот Variables типом данных плавания, разрешая нам напечатать в ценности с десятичным разрядом:

```
var throwForce     : float; Теперь, ниже Instantiate (Проиллюстрировать) ()
```

линия в Вашем, если утверждение, добавьте следующую линию:

```
newCoconut.rigidbody.velocity = transform.
  TransformDirection(Vector3(0,0, throwForce));
```

Здесь, мы ссылаемся на недавно иллюстрировавший примерами кокосовый орех его именем переменной, затем используя точечный синтаксис, чтобы обратиться к компоненту Rigidbody, и затем устанавливая скорость для твердого тела.

Мы установили скоростное использование, преобразовывают. TransformDirection, поскольку эта команда создает направление от местного до мирового места. Мы должны сделать это, потому что наша **Пусковая установка** будет постоянно перемещать и никогда не стоять перед последовательным направлением.

Ось Z мира *действительно* стоит перед последовательным направлением. Назначая скорость, мы можем взять определенную местную ось просто, назначая это ценность в Vector3. Это - то, почему у нас есть ценности 0 в X и Y параметрах Vector3.

Обеспечение составляющего присутствия

Если мы желаем охранять против ошибок, следующих из попытки обратиться к компоненту, который не приложен, то мы можем проверить, существует ли компонент, и добавлять это, если это не присутствует. Например, с нашим rigidbody, мы могли проверить новый случай newCoconut для компонента Rigidbody, говоря:

```
if(!newCoconut.rigidbody) {
  newCoconut.AddComponent(Rigidbody);
}
```

Здесь мы говорим, что, если нет никакого rigidbody, приложенного к этому случаю Variables, то добавьте компонент того типа. Мы не говорим 'rigidbody', просто помещая восклицательный знак перед утверждением в если обычная практика условия-а в scripting. Поскольку мы уже подготовили наш prefab с rigidbody, мы не должны делать это в этом случае.

Охрана столкновений
В то время как мы настроили нашу **Пусковую установку** в позиции, которая вдали от коллайдера характера игрока (Коллайдер Диспетчера) - мы должны все еще включать эту последнюю часть кода, чтобы охранять против иллюстрирования примерами новых кокосовых орехов, которые случайно пересекают коллайдер нашего игрока.

Это может быть сделано, используя IgnoreCollision () команда Classes Физики. Эта команда типично берет три аргумента:

IgnoreCollision(Collider A, Collider B, whether to ignore or not);

В результате мы просто должны накормить это этими двумя коллайдерами, которые мы не хотим, чтобы двигатель физики реагировал на, и установил третьи параметры к истинному.

Добавьте следующую линию ниже последней линии, которую Вы добавили:

```
Physics.IgnoreCollision(transform.root.collider,
  newCoconut.collider, true);
```

Здесь мы находим коллайдер характера игрока при использовании transform.root - это просто находит окончательный родительский объект любых объектов, к которым присоединена **Пусковая установка**. В то время как **Пусковая установка** - ребенок объекта **Main Camera**, у самой камеры нет коллайдера. Так, мы действительно хотим найти объект, к которому это присоединено - **Первый Диспетчер Человека,** которого мы находим при использовании transform.root.

Тогда мы просто проходим в имени переменной newCoconut, который представляет наш недавно иллюстрировавший примерами кокосовый орех. Для обоих параметры мы используем точечный синтаксис, чтобы обратиться к компоненту коллайдера.

Здесь мы должны были найти окончательного родителя этих объектов, но если Вы, которых **я** только отсылаю к родителю объекта, Вы можете обратиться к этому использующий **I**transform.parent. **Я**

Включая Звуковой Исходный компонент
Наконец, поскольку Ваше действие броска вовлекает аудио игры, мы можем использовать команду @script, чтобы заставить Unity включать Звуковой Исходный компонент, когда этот сценарий добавлен к объекту.

Ниже закрытия Update () function, который является в самом основании сценария, добавляет следующая линия:

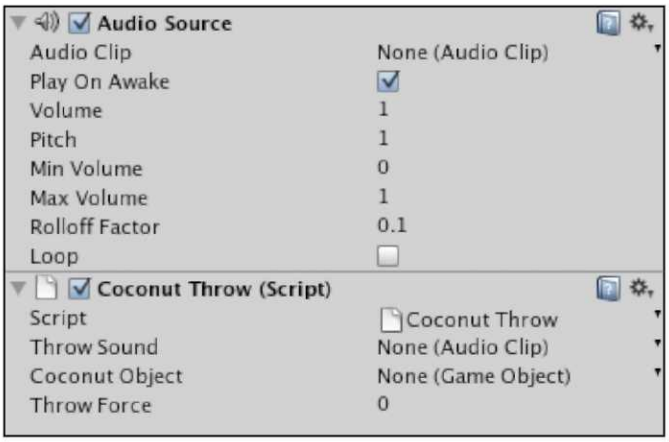
@script RequireComponent (AudioSource) Экономят Ваш

сценарий теперь, идя в **Файл |, Экономят** и возвращаются к Unity.

Сценарий и назначение Variables

Гарантируйте, что объект **Launcher** все еще отобран в группе **Иерархии**, и затем идти в **Компонент | Сценарии | Кокосовый Бросок** от главных меню. Unity добавит сценарий, который Вы только что написали как компонент, так же как Звуковой Исходный компонент Вы были обязаны добавлять.

Вы должны заметить, что общественные variables участника сценария CoconutThrow должны быть назначенными ценностями/активами, поскольку мы не делали этого вручную в сценарии.



Два общественных variables участника, которые требуют, чтобы активы были назначены на них, являются **Звуком Броска** и **Кокосовым Объектом** - Вы можете легко определить это, поскольку они указывают тип данных, который был установлен для них в сценарии. Третий общественный Variables участника, хотя данные, напечатанные в сценарии, просто предоставляет Вам неплатеж, оценивает в соответствующем числовом значении формата-а **0.**

Нажмите на опускаться стрелку направо ни от **Одного (Звуковая Скрепка)** и выберите скрепку аудио **броска** из списка доступных звуковых активов скрепки. Нажмите на стрелку направо ни от **Одного (Rigidbody)** и выберите **Кокосовый Prefab** как объект, чтобы назначить от списка, который появляется. Это связывает кокосовый заранее приготовленный актив непосредственно с этим Variables, означая, что сценарий создаст случаи того объекта. Наконец, назначьте Variables **Силы Броска** ценность **25.** Примите во внимание, что изменение ценностей в **Инспекторе** не регулирует то, что написано в сценарии, но просто отвергает любые назначенные на сценарий ценности. Помните, что эти публично выставленные variables участника могут быть заменены, не имея необходимость

повторно собирать или редактировать исходный текст. Привыкните к наладке этих ценностей в **Инспекторе,** поскольку удобство использования общественных variables участника таким образом является оперативным спасателем.

Теперь, пришло время проверять сценарий броска игрой, проверяющей игру. Нажмите кнопку **Play** теперь и или щелкните левой кнопкой мыши, или нажмите левую *клавишу CTRL* на клавиатуре, чтобы бросить кокосовый орех! Нажмите кнопку **Play** снова, чтобы прекратить проверять, как только Вы удовлетворены, что это работает правильно. Если что-нибудь не работает правильно, то возвратитесь к своему сценарию и перепроверке, что это соответствует полному сценарию, который является следующие:

```
var throwSound : AudioClip; var
coconutObject     : Rigidbody; var
throwForce     : float; function Update
() {
  if(Input.GetButtonUp("Fire1")){
    audio.PlayOneShot(throwSound);
    var newCoconut     :     Rigidbody = Instantiate(coconutObject,
      transform.position, transform.rotation);
    newCoconut.name = "coconut";
    newCoconut.rigidbody.velocity = transform.TransformDirection
    (Vector3(0,0, throwForce));
    Physics.IgnoreCollision(transform.root.collider,
    newCoconut.collider, true);

  }
}

@script RequireComponent(AudioSource)
```

Instantiate (Проиллюстрировать) ограничение и объектную уборку

Instantiate (Проиллюстрировать) объектов таким образом, в которых мы сделали, является идеальным использованием заранее приготовленной системы Unity, облегчая строить любой объект в сцене и создавать много клонов этого во время времени выполнения.

Однако, создание многих клонов объекта с твердым телом может оказаться дорогостоящим, поскольку каждый призывает двигатель физики, и поскольку это договаривается о своем пути вокруг трехмерного взаимодействия мира другие объекты - это будет использовать циклы центрального процессора (обрабатывающий власть). Теперь вообразите, должны ли Вы были позволить Вашему

игроку создавать бесконечное количество управляемых физикой объектов, и Вы можете ценить, что Ваша игра может замедлиться через некоторое время. Поскольку Ваша игра использует слишком много циклов центрального процессора и памяти, норма структуры становится ниже, создавая вяленое мясо смотрит на Вашу ранее гладко показывавшую жестом игру. Это конечно будет плохим опытом для игрока, и в коммерческом смысле, убило бы Вашу игру.

Вместо того, чтобы надеяться, что игрок не бросает много кокосовых орехов, мы вместо этого сделаем две вещи, чтобы избежать слишком многих объектов, забивающих норму структуры игры:

- Позвольте игроку бросать кокосовые орехи только в то время как в определенном пятне в мире игры
- Напишите сценарий, чтобы удалить кокосовые орехи из мира после определенного времени начиная с их экземпляра

Если бы мы воздействовали на экземпляр большего масштаба, например, оружия, то мы также добавили бы основанную на времени задержку, чтобы гарантировать 'перезарядить' период. Это не необходимо в этом случае, поскольку мы избегаем слишком многих кокосовых орехов, бросаемых сразу при использовании GetButtonUp () команда - игрок должен выпустить ключ прежде, чем кокосовый орех будет брошен.

Формирование кокосового броска

Мы обратимся к первому пункту, просто имея переключающуюся логическую переменную, которая должна быть верной для игрока, чтобы бросить кокосовые орехи, и мы только установим этот Variables в верный, когда характер игрока будет стоять на части модели платформы - который будет нашей кокосовой целевой ареной.

Наличие игрока, беспорядочно бросающего кокосовые орехи вокруг уровня далеко от этой миниигры, действительно не имело бы смысла, таким образом это - хорошая вещь, чтобы ограничить это действие вообще независимо от наших рассмотрений работы.

Вновь откройте сценарий **CoconutThrow**, если Вы закрыли его, щелкая два раза его изображением в **Проектной** группе. Иначе, просто переключитесь назад на редактора сценария и продолжите воздействовать на это. Ранее, мы смотрели на формирование и дезактивацию сценариев через позволенный параметр компонента. Точно так же в этом случае, мы могли очень хорошо использовать GetComponent () команда, выбрать этот сценарий, и повредить это, когда мы не хотим, чтобы игрок бросил кокосовые орехи. Однако, как со всеми проблемами scripting, есть много решений любой проблемы, и в этом случае, мы будем смотреть на использование статических variables, чтобы общаться через сценарии. Добавьте следующую линию в очень главном из сценария **CoconutThrow**:

```
статический вар canThrow: булевый = ложный;
```

Эта статическая приставка перед нашим Variables - Джевэскрипт Unity

способ создать глобальную-а стоимость, к которой могут получить доступ другие сценарии. В результате этого мы будем добавлять другое условие к нашему существующему, если утверждение, которое позволяет нам бросать, так что найдет что линия в пределах Update () function. Это должно быть похожим на это:

```
if(Input.GetButtonUp("Fire1")){
```

Чтобы добавить второе условие, просто добавьте два символа амперсанда перед правильной заключительной скобкой если утверждение наряду с названием статического Variables, следующим образом:

```
if(Input.GetButtonUp("Fire1") && canThrow){
```

Примите во внимание здесь, что просто написание имени Variables является более коротким способом сообщить:

```
if(Input.GetButtonUp("Fire1") && canThrow==true){
```

Поскольку мы установили canThrow Variables в сценарии к ложному, когда это было объявлено, и потому что это статично (поэтому не общественный Variables участника), мы должны будем использовать другую часть scripting, чтобы установить этот Variables в истинный. Учитывая, что наш игрок должен стоять в определенном месте, наш лучший курс действия для этого должен использовать обнаружение столкновения, чтобы проверить, сталкивается ли игрок со специфическим объектом, и если так, установил этот статический Variables в истинный, позволяя им бросить.

Откройте сценарий **PlayerCollisions** теперь и определите местонахождение прокомментированного OnControllerColliderHit () Functions. Это должно быть похожим на это:

```
/*
function OnControllerColliderHit(hit: ControllerColliderHit){
  if(hit.gameObject.tag == "outpostDoor" && doorIsOpen == false){
  Door(doorOpenSound, true, "dooropen");
  }
}
*/
```

Удалите/* и */характеры, чтобы непрокомментировать это, делая это активный код снова. Однако, удалите, если утверждение, поскольку мы больше не нуждаемся в этом Functions, чтобы обращаться с дверным проемом и закрытием. У Вас должен теперь только быть Functions, непосредственно остающийся, как это:

```
function OnControllerColliderHit(hit: ControllerColliderHit){
}
```

В модели **платформы** Вы загрузили, есть часть модели, названной

циновкой, на которой игрок должен стоять, чтобы бросить кокосовые орехи, таким образом мы гарантируем, что они сталкиваются с этим объектом. В этом Functions, еще добавьте следующий если утверждение:

```
if(hit.collider == GameObject.Find("mat").collider){
  CoconutThrow.canThrow=true; }else{
  CoconutThrow.canThrow=false;
}
```

Здесь мы проверяем, равен ли текущий поражаемый коллайдер (двойной, равняется, сравнительное), коллайдер объекта игры в сцене с циновкой названия. Если это условие соблюдают, то мы просто обращаемся к статическому Variables canThrow в сценарии по имени CoconutThrow использование точечного синтаксиса, и устанавливаем этот Variables в истинный. Это позволяет Instantiate (Проиллюстрировать) () команды сценария CoconutThrow работать. Иначе, мы удостоверяемся, что этот Variables, который это установило в ложный, означая, что игрок не будет в состоянии бросить кокосовые орехи, когда он не будет стоять на циновке броска платформы.

В редакторе сценария, пойдите в **Файл |, Экономят** и возвращаются к Unity.

Удаление кокосовых орехов

Как объяснено ранее, слишком много управляемых физикой объектов в Вашей сцене могут серьезно затронуть работу. Поэтому, в случаях, таких как это, где Ваши объекты - просто холостые объекты (возражает, что не должен быть сохранен), мы можем написать сценарий, чтобы автоматически удалить их после определенного количества времени.

Выберите папку **Сценариев** в **Проектной** группе, нажмите на кнопку **Create**, и выберите **JavaScript**, чтобы сделать новый сценарий. Переименуйте этот сценарий **CoconutTidy**, нажимая *Возвращение* (Mac) или *F2* (PC) и перепечатывание. Тогда щелкните два раза изображением сценария, чтобы начать это в редакторе сценария.

Удалите Update () function по умолчанию из этого сценария, поскольку мы не нуждаемся в этом. Чтобы удалить любой объект из сцены, мы можем просто использовать **Разрушение ()** команда. Осуществите его второй параметр, чтобы установить период ожидания. Добавьте следующий Functions и команду к Вашему сценарию теперь:

```
function Start(){
  Destroy(gameObject, 5);
}
```

При использовании Начала () команда, которую мы вызовем, Разрушает (), как только этот объект появляется в мире, то есть, как только это иллюстрируется примерами игроком, нажимающим кнопку огня. Обращаясь к gameObject, мы просто говорим 'объект, к которому этот

сценарий присоединен'. После запятой мы просто сообщаем время в секундах, чтобы ждать до формирования этой команды.

В результате этого сценария - как только кокосовый орех брошен, это останется в мире в течение пяти секунд, и затем будет удалено. Пойдите в **Файл |, Экономят** в редакторе сценария и возвращаются к Unity.

Ранее, когда мы написали сценарии, мы приложили их к объектам в сцене, мы продолжали работать. Однако, в этом случае, мы уже закончили воздействовать на наш кокосовый prefab, и у нас больше нет копии в сцене. Есть два способа применить сценарий, который мы только что написали prefab. Чтобы сделать это легкий путь, Вы можете:

- Выберите **Кокосовый** prefab, который Вы сделали ранее в **Проектной** группе, и идти в **Компонент | Сценарии | CoconutTidy**

Или следовать более многоречивым маршрутом, Вы можете изменить prefab в **Сцене** следующим образом:

- Тяните **Кокосовый prefab** к окну **Scene** или группе **Иерархии**
- Пойдите в **Компонент | Сценарии | CoconutTidy,** щелчок **Добавляет** к corvfirm когда сказано, что **'Добавление компонента потеряет заранее приготовленного родителя'**
- Отзовитесь эхом это обновление к оригинальному prefab, идя в **GameObject | Применяют Изменения к prefab**
- Удалите случай в сцене, используя сокращенную *Команду + Клавиша Backspace*

  (Mac) или *Удаляет* (PC).

В этом случае я рекомендую прежнему, то есть, единственному маршруту шага, столь давайте сделаем что теперь. Но в некоторых случаях может быть полезно забрать prefab в сцену, и изменить это прежде, чем Вы примените изменения к prefab. Например, если бы Вы воздействуете кое на что визуальное, такое как система частицы, тогда Вы должны были бы видеть то, что производит Ваши регуляторы, или недавно добавленные компоненты будут иметь. Поэтому, взятие prefab такого объекта в сцену, чтобы отредактировать было бы существенным.

Пойдите в **Файл |, Экономят Проект** в Unity теперь, чтобы обновить Ваше продвижение пока.

Добавление кокосовой застенчивой платформы

Теперь мы готовы осуществить нашу миниигру от активов, которые Вы загружали ранее. Помещая платформу и три цели в сцену, мы проверим на столкновения между кокосовыми орехами и целями, и напишем сценарий, чтобы проверить, сбиты ли все три цели в некогда цели миниигры.

В **Проектной** группе есть **Кокосовая** папка **Игры**, которая была

импортирована, когда Вы загружали активы, чтобы закончить эту главу. Выберите трехмерную модель в этой папке, названной **платформой**, чтобы видеть ее свойства в **Инспекторе.**

Параметры настройки импорта
Прежде, чем мы поместим платформу и цели в сцену, мы гарантируем, что они правильно измерены и могут идтись на игроком, производя коллайдеры для каждой части моделей.

Платформа
В компоненте **FBXImporter** в **Инспекторе,** набор **Коэффициент пропорциональности** к **1.25,**
тогда выберите коробку для**,** **Производят Коллайдеры**, чтобы гарантировать, что Unity назначает коллайдер петли на каждую часть модели, означая, что характер игрока будет в состоянии идти на платформе.

Чтобы подтвердить это изменение, щелкните кнопкой **Apply** у основания **Инспектора** теперь.

Теперь тяните модель от **Проектной** группы до окна **Scene**, и используйте **инструмент Transform (преобразовать)**, чтобы поместить это где-нибудь около заставы, чтобы гарантировать, что игрок понимает, что две особенности связаны. Удостоверьтесь, что Вы понижаете модель платформы в основание так, чтобы характер игрока был в состоянии идти по шагам впереди модели. Вот расположение, которое я выбрал:

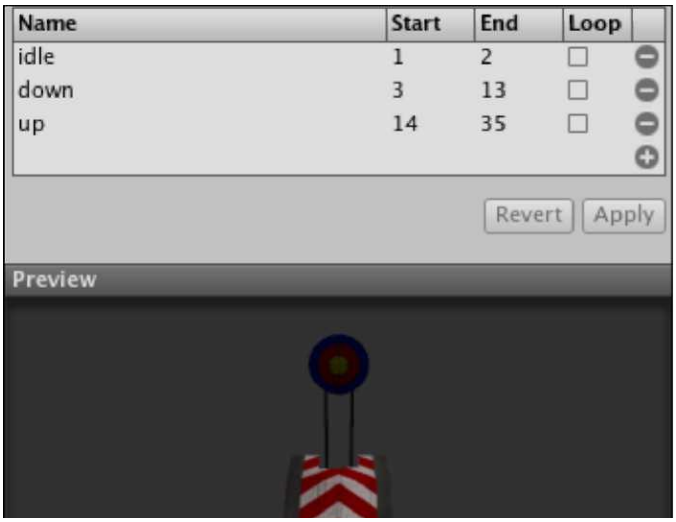
Цели и кокосовые столкновения

Теперь найдите **целевую** модель в **Кокосовой** папке **Игры** в **Проектной** группе, и выберите это, чтобы видеть, что различный импорт устанавливает компоненты в **Инспекторе.**

В компоненте **FBXImporter** в **Инспекторе,** выберите коробку для**, Производят Коллайдеры**, чтобы гарантировать, что любая часть модели, что кокосовые хиты должны заставить это отражать - помнит, что без коллайдеров, трехмерные объекты пройдут через друг друга. Кроме того, установите **Коэффициент пропорциональности** в ценность **1** здесь.

В компоненте **Мультипликаций** мы должны будем определить, что структуры и дать названия для каждой мультипликации сообщают, что мы хотели бы, чтобы эта модель имела, таким же образом поскольку мы сделали для дверных мультипликаций заставы. Добавляя эти государства мультипликации, мы можем призвать их в наших сценариях, если столкновение между кокосовым орехом и правильной частью цели происходит.

Добавьте три государства мультипликации, как показано в следующем скриншоте, нажимая плюс **(+)** изображение направо от стола мультипликаций, тогда fflling на **Название** и структуры **Начала** и **Конца**:



Когда Вы закончили стол **Мультипликаций**, не забудьте нажимать кнопку **Apply** в основании, чтобы подтвердить это и другие изменения импорта, которые Вы произвели в активе.

Размещение

Чтобы поместить цели легко в платформу, мы добавим их как дети объекта **платформы** уже в нашей сцене. Чтобы сделать это, просто тяните целевую модель от **Кокосовой** папки **Игры** в **Проектной** группе, и понизьте это на объект родителя **платформы** в группе **Иерархии**.

Вы будете побуждены с окном диалога, сообщающим Вам, что добавление этого детского объекта **Потеряет заранее приготовленную связь.** Просто щелкните кнопкой **Continue** здесь. Это просто сообщает Вам, что изменения сделали к компонентам приложенный к оригинальной модели в **Проектной** группе например, сценарий параметры, больше не относятся к этой копии в сцене, потому что Вы разъединяете связь между этим случаем и оригинальным активом или prefab.

Добавляя цель, поскольку ребенок платформы заставит платформу расширять и показывать свои существующие детские объекты наряду с целью, которую Вы только что добавили.

Поскольку цель будет помещена в центр платформы по умолчанию, переместит это в правильную позицию, изменяя ценности в компоненте **Transform (преобразовать)** в **Размещении инспектора** это в **(0, 1.7, 1.7).**

Кокосовый сценарий обнаружения

Поскольку мы должны обнаружить столкновения между целевой частью целевой модели - в противоположность стеблю или основе, для примера - мы должны будем написать сценарий с обнаружением коллизий, чтобы быть примененными к той части модели. Выберите папку **Сценариев** в **Проектной** группе, затем нажмите на кнопку **Create**, и выберите **JavaScript**

из опускаться меню. Переименуйте это от **NewBehaviourScript** до **CoconutCollision,** и затем щелкните два раза изображением, чтобы открыть это в редакторе сценария.

Установление variables

Во-первых, мы должны установить пять variables, и они следующие:

- Общественный Variables участника GameObject, хранящий целевой объект непосредственно
- beenHit булевое, чтобы проверить, снижается ли цель в настоящее время
- Перезагружен частный Variables таймера с плавающей запятой, чтобы ждать определенного количества времени перед целью
- hitSound звуковой общественный Variables участника
- resetSound звуковой общественный Variables участника

Чтобы сделать это, добавьте следующий код к началу сценария:

```
var targetRoot : GameObject; private var beenHit :
boolean = false; private var timer     :     float = 0.0;
var hitSound     : AudioClip; var resetSound     :
AudioClip;
```

Отметьте здесь, что beenHit и таймер - частные variables, поскольку они не должны быть назначены в **Инспекторе** - их ценности установлены и используются только в пределах сценария. Урегулирование частной приставки скроет их от представления в **Инспекторе,** означая, что они не общественные variables участника.

Обнаружение столкновения

Затем, переместите существующий Update () function вниз несколькими линиями и напишите в следующем Functions обнаружения столкновения:

```
function OnCollisionEnter(theObject     :     Collision) {
  if(beenHit==false && theObject.gameObject.name=="coconut"){
    audio.PlayOneShot(hitSound); targetRoot.animation.Play("down");
    beenHit=true;

  }
}
```

Отличающийся от OnControllerColliderHit () и OnTriggerEnter () функционирует, мы использовали ранее, OnCollisionEnter () обращается с обычными столкновениями между объектами с примитивными коллайдерами, то есть, не коллайдерами диспетчера характера и не коллайдерами в более аккуратном способе.

В этом Functions параметр theObject является случаем Classes

Столкновения, который хранит информацию на скоростях, rigidbodies, коллайдерах, преобразовать, GameObjects, и пункты контакта, вовлеченные в столкновение. Поэтому, здесь мы просто согласовываем Classes с, если утверждение, проверяя, снабдил ли тот параметр gameObject кокосовым орехом названия. Чтобы гарантировать, что цель не может быть поражена много раз, у нас есть дополнительное условие в, если утверждение, которое проверяет, что beenHit установлен в ложный. Это будет иметь место, когда игра начнется, и когда это столкновение происходит, beenHit установлен в верный так, чтобы мы не могли случайно вызвать это дважды.

Этот Functions также играет звуковой файл, назначенный на Variables hitSound, и играет государство мультипликации, названное вниз на любом объекте, который мы drag and drop (перетащили и опустили) к targetRoot Variables. После письма сценария мы назначим целевой объект родителя модели на этот Variables в **Инспекторе.**

Сброс цели
Теперь в Update () function, мы должны будем использовать beenHit Variables, чтобы начать таймер подсчитывание к 3 секундам. Это означает, что от структуры, что она поражена кокосовым орехом, таймер рассчитает к 3 прежде, чем вызвать сброс. Для этого мы будем нуждаться два, если утверждения, один, чтобы обращаться с проверкой beenHit, чтобы быть верным и увеличивание таймера и другого, чтобы проверить, достиг ли таймер 3 секунд. Переместите заключительную правильную вьющуюся скобу Update () function вниз несколькими линиями, и добавьте следующий код:

```
if(beenHit){
   timer += Time.deltaTime;
if(timer > 3){
   audio.PlayOneShot(resetSound);
   targetRoot.animation.Play("up");
   beenHit=false;
   timer=0.0;
}
```

Наше первое, если утверждение ждет beenHit, чтобы быть верным, и затем добавляет к Variables таймера, используя прилавок Time.deltaTime. Это - определенная норма неструктуры, и поэтому рассчитывает в режиме реального времени.

Второе, если утверждение ждет Variables таймера, чтобы превысить 3 секунды, то игры звук, назначенный на resetSound Variables, играет государство мультипликации модели, назначенной на targetRoot, и перезагружает beenHit и таймер к их оригинальным государствам так, чтобы цель могла быть поражена снова.

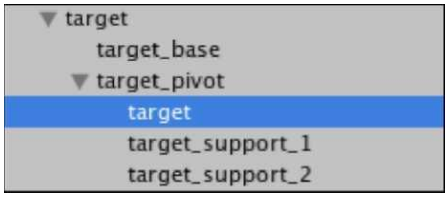
Включая звуковой источник

Поскольку мы играем звуки, мы должны будем добавить обычную команду RequireComponent к основанию нашего сценария, чтобы гарантировать, что звуковой источник добавлен к объекту, к которому приложен этот сценарий. Поместите следующую линию в самое основание сценария после заключительной вьющейся скобы Update () function:

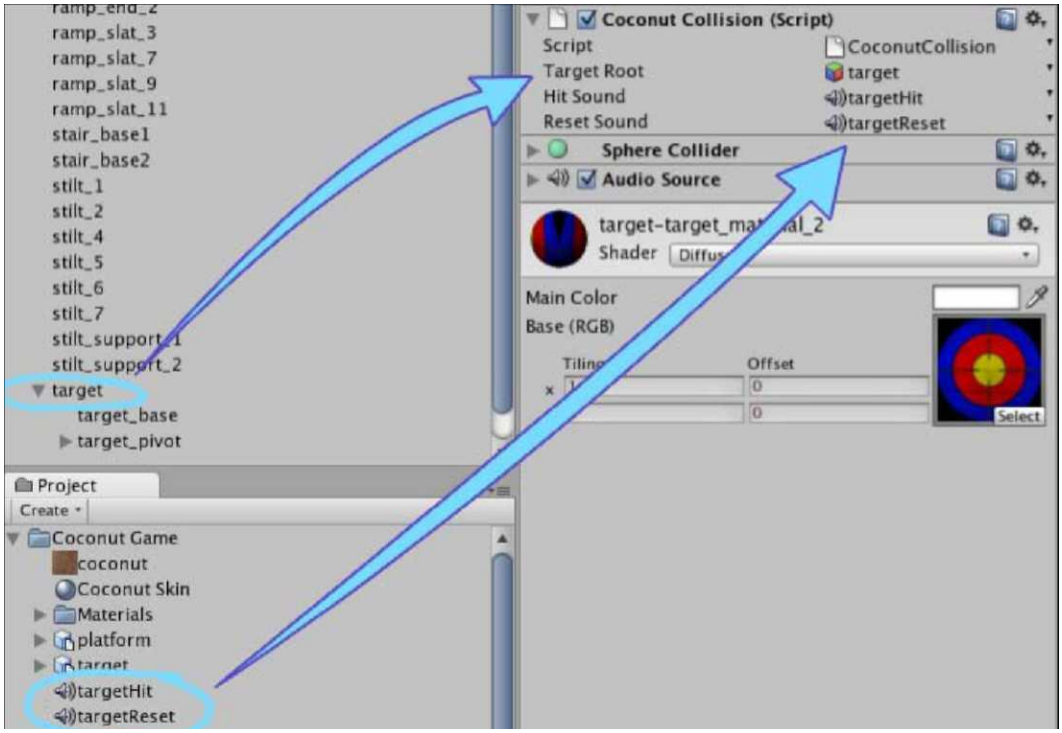
```
@script RequireComponent (AudioSource)
```

Пойдите в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity теперь.

Назначение сценария
В группе **Иерархии**, расширьте **целевую** модель, которую Вы добавили к платформе, чтобы видеть ее составные части, и затем расширяете **target_pivot** детскую группу, чтобы показать цель непосредственно и ее поддержки. Они должны быть похожими на это:



В последнем скриншоте я выбрал цель непосредственно. Вы должны сделать то же самое теперь, поскольку мы должны гарантировать, что сценарий, который мы только написали, назначен на этот объект, в противоположность родительской целевой модели. Мы должны проверить на столкновения с этой детской частью, поскольку мы не хотим, чтобы столкновения были вызваны, если игрок бросает кокосовый орех в основу или поддержки, например.
С этим отобранным объектом, пойдите в **Компонент | Сценарии | CoconutCollision.** Сценарий, который мы только написали, будет добавлен, наряду со **Звуковым Исходным** компонентом. Теперь тяните **целевого** родителя от группы **Иерархии** до **Целевого** Variables участника общественности **Корня** и **targetHit** и **targetReset** звуковых файлов от **Проектной** группы до соответствующих общественных variables участника в **Инспекторе,** как иллюстрировано в следующем скриншоте:



Теперь пойдите в **Файл |, Экономят Сцену** в Unity, чтобы обновить Ваше продвижение. Нажмите кнопку **Play**, чтобы проверить игру и прогулку к платформе, удостоверяясь, что Вы стоите на циновке - Вы должны теперь быть в состоянии бросить кокосовые орехи и сбить цель. Нажмите **Игру** снова, чтобы прекратить проверять.

Чтобы закончить нашу установку миниигры, мы сделаем еще три цели, используя заранее приготовленную систему.

Создание большего количества целей
Чтобы сделать больше целей, мы дублируем нашу существующую цель, превратив это в prefab. Сделайте новый prefab, выбирая папку **Prefab** в **Проектной** группе и идя в кнопку **Create** и выбирая **Prefab.** Переименуйте это от **нового prefab**, чтобы **Предназначаться для Prefab.** Теперь, drag and drop (перетащили и опустили) целевой объект родителя в **Иерархии** на этот новый prefab, чтобы спасти это.

Текст родительской **цели** в **Иерархии** должен стать синим, указывая, что это связано с prefab в проекте.

Теперь выберите родительскую **цель** в группе **Иерархии**, и *Команду* прессы *+ D* (Mac) или *Ctrl + D* (PC), чтобы дублировать этот объект.

С отобранным дубликатом, устанавливает его **X** позиций в компоненте **Transform (преобразовать)** в **Инспекторе** к **1.8.** Повторите этот шаг дублирования снова теперь, чтобы сделать третью цель, но на сей раз, устанавливая **X** позиций в**-1.8.**

Победа игры

Чтобы закончить Functions нашей миниигры - чтобы дать игроку заключительную батарею, они должны зарядить дверь заставы - мы должны будем написать сценарий, который проверяет, сбиты ли все три цели сразу.

Выберите папку **Сценариев** в **Проектной** группе, и используйте кнопку **Create**, чтобы сделать новый файл **JavaScript**. Переименуйте этот сценарий **CoconutWin,** и затем щелкните два раза его изображением, чтобы начать это в редакторе сценария.

Установка Variables

Наверху сценария, добавьте следующие четыре variables:

```
static var targets      :     int = 0;
private var haveWon     : boolean = false;
var win : AudioClip;
var battery     : GameObject;
```

Здесь мы начинаем со статического Variables, названного целями, который является эффективно в противоречии с магазином, сколько целей в настоящее время пробивается вниз - этому назначит изменение, которое мы произведем в нашем сценарии CoconutCollision позже. У нас тогда есть частный Variables, названный haveWon, который будет мешать этой миниигре быть переигранным, просто будучи установленным в истинный после первой победы.

У нас тогда есть два общественных variables участника один, чтобы сохранить скрепку аудио победы, и другой, чтобы сохранить prefab батареи, так, чтобы этот сценарий мог иллюстрировать примерами его, когда игрок победил.

Проверка победу

Теперь добавьте следующий код к Update () function:

```
if(targets==3 && haveWon == false){ targets=0;
  audio.PlayOneShot(win);
  Instantiate(battery, Vector3(transform.position.x,
    transform.position.y+2, transform.position.z),
    transform.rotation);
  haveWon = true;
}
```

Это, если у утверждения есть два условия. Это гарантирует, что целевой счет достиг 3, подразумевая, что они должны все быть сбиты, и также что haveWon Variables ложен, означая, что это - первый раз, когда игрок делал попытку игры этой миниигры.

Когда эти условия соблюдают, следующие команды выполнены:

- Сценарий перезагружает целевой Variables к 0 (это - просто другая мера, чтобы гарантировать, что, если утверждение не повторно вызывает),
- Скрепка аудио победы играется как обратная связь игрока
- Случай объекта, который мы назначим на Variables батареи, иллюстрируется примерами, беря его позицию с ручным Vector3, который использует элемент X и ценности Z платформы (как, именно это к этому сценарию будут относиться), добавляя 2 к ценности y
- Мы устанавливаем haveWon Variables в истинный, что означает, что игра не может быть выиграна снова, и мешает игроку произвести больше батарей

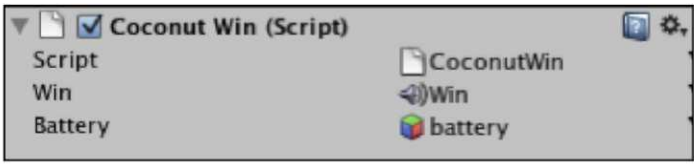
Наконец, поскольку мы играем звук, добавьте следующую линию к самому основанию сценария:

```
@script RequireComponent (AudioSource)
```

Теперь пойдите в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity.

Назначение сценария
Выберите объект родителя платформы в группе **Иерархии** и пойдите в **Компонент | Сценарии | Кокосовая Победа.**

Чтобы закончить это, назначьте prefab **батареи** от папки **Prefab** в **Проектной** группе к Variables участника **батареи**, и звуковой эффект **Победы** от **Кокосовой** папки **Игры** до Variables участника **Победы**. Когда сделано, компонент сценария должен быть похожим на это:



Увеличивание и цели decrementing
Наконец, чтобы сделать работу игры, мы должны возвратиться к нашему сценарию **CoconutCollision**, и добавить тот к статическим целям Variables в CoconutWin, когда мы сбиваем цель, и вычитаем тот, когда цель перезагружает.

Это просто сделать, поскольку наш сценарий уже настроен, чтобы обращаться с теми двумя событиями. Щелкните два раза на изображении сценария **CoconutCollision** в папке **Сценариев**, чтобы начать это в редакторе сценария.

Добавление
В Functions-OnCollisionEnter обнаружения столкновения () - если утверждение показывает линию, устанавливающую beenHit к истинному. Найдите линию:

beenHit=true; и после этого,

добавьте следующую линию:

CoconutWin.targets ++;

Вычитание
В Update () function, второе, если ручки утверждения, перезагружающие цель, подразумевая, мы должны вычесть от целевого счета. Найдите линию:

beenHit=false; и затем добавьте

следующую линию после этого:

CoconutWin.targets-;

В обоих этих случаях мы используем точечный синтаксис, чтобы обратиться к сценарию (эффективно Classes), сопровождаемый названием Variables, а именно, целей.

Пойдите в **Файл |, Экономят** в редакторе сценария, и возвращаются к Unity.

Нажмите кнопку **Play** теперь и проверьте игру. Бросок кокосовых орехов и сбивание всех трех целей сразу должны заставить платформу иллюстрировать примерами батарею для Вас, чтобы собраться. Когда это делает, Вы просто нажимаете *Клавишу "пробел"*, чтобы перепрыгнуть через барьер и собрать батарею. Нажмите **Игру** снова, чтобы прекратить проверять игру, и пойти в **Файл | Экономят Сцену** в Unity, чтобы обновить Ваше продвижение.

Последние штрихи

Чтобы заставить эту миниигру чувствовать себя немного больше полируемой, мы добавим перекрестие к настороженному показу, когда игрок будет стоять на циновке броска, и использовать наш существующий объект **GUI TextHint**, чтобы дать инструкции игрока относительно того, что сделать, чтобы выиграть кокосовую застенчивую игру.

Добавление перекрестия

Чтобы добавить перекрестие к экрану, сделайте следующее:

- Откройте **Кокосовую** папку **Игры** в **Проектной** группе.

- Выберите файл структуры **Перекрестия**.
- Пойдите в **GameObject |, Создают Другой | Структура GUI.** Это возьмет измерения файла структуры и назначит им для Вас, создавая новый объект Texture GUI, названный в честь структуры **Перекрестия**.

Это автоматически отобрать в группе **Иерархии**, таким образом Вы сможете видеть ее компонент **GUITexture** в **Инспекторе.** Графическое перекрестие должно было стать видимым в представлении **Игры** и в представлении **Сцены**, если у Вас есть кнопка **Game Overlay** toggled. Поскольку это сосредоточено по умолчанию, это работает отлично с нашими маленькими 64 x 64 файла структуры. Когда мы создали файл структуры для этого примера, было важно, что мы использовали легкие и темные края так, чтобы крест был легок видеть независимо от того, смотрит ли игрок кое на что легкое или темное.

Примите во внимание, что, если не выбирая файл структуры и просто создавая Структуру GUI, Вам подарят эмблему Unity. Вы тогда должны обменять это для своей структуры в **Инспекторе,** так же как заполниться в измерениях вручную. Поэтому это всегда лучше, чтобы выбрать структуру, Вы желаете сформировать свой объект Texture GUI сначала.

Toggling перекрестие Структура GUI

Откройте сценарий **PlayerCollisions** от папки Сценариев **Проектной** группы. В
OnControllerColliderHit () Functions, Вы заметите, что у нас уже есть scripting, который проверяет, находимся ли мы на циновке броска. Поскольку мы хотим, чтобы перекрестие было видимо только, когда мы будем на этой циновке, мы добавим больше кода к ним если утверждения.

Перед, если утверждение, которое читает:

```
if(hit.collider == GameObject.Find("mat").collider){
  CoconutThrow.canThrow=true;
}
```

Добавьте следующие две линии:

```
var crosshairObj       : GameObject = GameObject.Find("Crosshair"); var
crosshair : GUITexture = crosshairObj. GetComponent(GUITexture);
```

Здесь мы устанавливаем два variables, тот, который находит объект перекрестия, использующий GameObject. Найдите и имя объекта и другой, который использует GetComponent (), чтобы представить компонент предыдущего Variables GUITexture. Делая это, мы можем просто позволить и повредить перекрестие, еще используя если и утверждения.

Теперь в, если утверждение, показанное ранее, мы добавляем следующую линию:

```
crosshair.enabled = true;
```

И в сопровождении еще утверждение, мы добавляем линию:

```
crosshair.enabled=false;
```

Информирование игроку
Чтобы помочь игроку понять, что они должны сделать, мы будем использовать наш объект **GUI TextHints** от Главы 5, чтобы показать сообщение на экране, когда игрок будет стоять на циновке броска.

В, если утверждение OnControllerColliderHit () в **PlayerCollisions,** добавьте следующие линии:

```
TextHints.textOn=true;
  TextHints.message = "Knock down all 3 at once to win a battery!";
  GameObject.Find("TextHint GUI").transform.position.y = 0.2,-
```

Здесь мы включаем **TextHint GUI**, устанавливая его статический Variables textOn к истинному, тогда посылая последовательность текста к его сообщению статический Variables. Таймер в пределах сценария TextHints будет заботиться о выключении этого сообщения, как только игрок оставляет циновку.

Здесь мы также использовали GameObject. Найдите, чтобы обратиться к TextHint `GUI` объект непосредственно и установить его у позицию в 0.2. Это - то, потому что, по умолчанию, наши TextHint GUI сообщения появляются в центре экрана, который означал бы, что они будут в том же самом месте как наше перекрестие. Поскольку мы не хотим это, мы используем последнего этих трех линий в предыдущем кодовом отрывке, чтобы установить позицию ниже на экране.

Теперь, в еще утверждение OnControllerColliderHit (), добавьте следующее:

```
GameObject.Find("TextHint GUI").transform.position.y = 0.5;
```

Это просто перезагружает TextHint `GUI` объект назад к его оригинальной позиции, когда игрок сделан с кокосовой игрой и оставляет циновку.

Ваш законченный OnControllerColliderHit () Functions должен быть похожим на это:

```
Пойдитеfunction OnControllerColliderHit(hit: ControllerColliderHit){ var
  crosshairObj       : GameObject = GameObject.Find("Crosshair"),-var
  crosshair     : GUITexture =
    crosshairObj.GetComponent(GUITexture); if(hit.collider ==
  GameObject.Find("mat").collider){
    CoconutThrow.canThrow=true;
```

```
    crosshair.enabled = true;
    TextHints.textOn=true;
TextHints.message = "Knock down all 3 to win a battery!";
GameObject.Find("TextHint GUI").transform.position.y = 0.2,}
else{
    CoconutThrow.canThrow=false;
    crosshair.enabled = false;
    GameObject.Find("TextHint GUI").transform.position.y = 0.5;
}
```

} в **Файл |, Экономят** в редакторе сценария, чтобы обновить Ваше продвижение и возвратиться к Unity.

Нажмите кнопку **Play** и проверьте игру. Теперь, стоя на циновке броска, Вы должны видеть перекрестие и сообщение, говоря Вам, что сделать. Оставьте циновку, и перекрестие должно исчезнуть из экрана, сопровождаемого сообщением после нескольких секунд. Нажмите **Игру** снова, чтобы прекратить проверять игру, и пойти в **Файл | Экономят Проект** обновить Ваше продвижение пока.

Резюме

В этой главе мы затронули различные темы, которые Вы найдете крайне важным, создавая любой сценарий игры. Мы смотрели на осуществление объектов твердого тела, которые используют двигатель физики. Это - кое-что, на что Вы вероятно расширитесь во многих других сценариях игры, работая с Unity. Мы также исследовали понятие экземпляра, кое-что, что очень важно, чтобы схватиться с, поскольку это означает, что Вы можете создать или клонировать любой заранее приготовленный актив или объект игры во время очень полезного инструмента во-время-выполнения-а в Вашем проектирующем игру арсенале.

Мы также дали игроку дальнейшую обратную связь, снова используя наш объект GUI TextHint, сделанный в Главе 5, и работали через сценарии, чтобы послать информацию в этот объект.

Они - понятия, которые Вы продолжите использовать в остальной части этой книги и в Ваших будущих проектах Unity. В следующей главе мы будем отдыхать от кодирования и смотреть на большее количество эстетических эффектов Unity. Мы исследуем использование систем частицы, чтобы создать огонь вне предоставления каюты заставы игрок визуальная награда за то, чтобы выиграть миниигру и открыть дверь.

7

Системы Частицы

В этой главе мы будем смотреть на некоторые из эффектов предоставления, доступных для Вас как разработчик Unity. Чтобы создать более динамические трехмерные миры, отдавая эффекты за пределами простых материалов и texturing используются, чтобы моделировать, и часто подчеркнуть, особенности реального мира. Много трехмерных игр приняли визуальные соглашения захваченных камерой образов, вводя такие эффекты как вспышки линзы и полоски света как часть моделируемой точки зрения, которая в реальном исчислении никогда не свидетельствовала бы такие эффекты.

Мы уже использовали в своих интересах эффект предоставления вспышки линзы в Главе 2, где мы использовали вспышку линзы Солнца на легком компоненте нашего главного направленного света. В этой главе мы будем смотреть на более универсальные эффекты, которые могут быть достигнуты при использовании систем частицы в пределах Вашего трехмерного мира. Игры используют эффекты частицы достигнуть обширного диапазона эффектов от тумана и дыма к искрам, лазерам, и простым образцам. В этой главе мы будем смотреть на то, как мы можем использовать две системы частицы, чтобы достигнуть эффекта моделируемого огня.

В этой главе Вы будете учиться:

- Что составляет систему частицы - ее компоненты и параметры настройки
- Как построить системы частицы, чтобы моделировать огонь и дым
- Дальнейшая работа с инструкциями игрока на экране и обратной связью
- Используя variables, чтобы активизировать системы частицы во время времени выполнения

Какова система частицы?

Система частицы упомянута в Unity как система - а не компонент как, требуется много компонентов, сотрудничающих, чтобы функционировать должным образом. Прежде, чем мы начнем работать с

системами непосредственно, мы должны понять составные части и их роль в системе.

Эмитент частицы

В пределах любой системы частицы компонент эмитента отвечает за иллюстрирование примерами отдельных частиц. В Unity есть **Эллиптический Эмитент Частицы** и доступный **Эмитент Частицы Петли**.

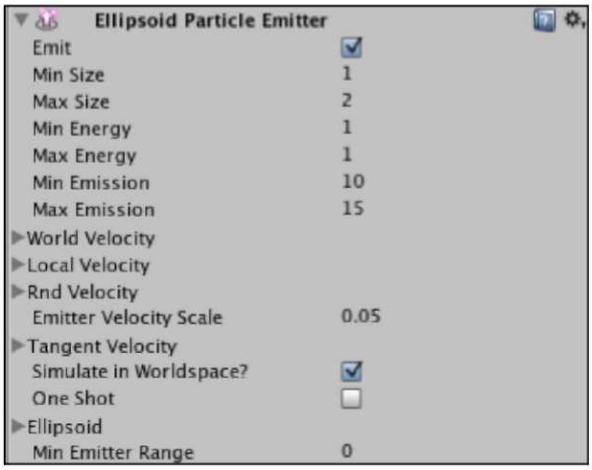
Эллипсоид обычно используется для эффектов, таких как дым, пыль, и другие такие экологические элементы, которые могут быть созданы в определенном месте. Это упоминается как эллипсоид, потому что это создает частицы в пределах сферы, которая может быть протянута, чтобы приспособить систему.

Эмитент петли создает частицы, которые привязаны непосредственно к трехмерной петле и могут или быть оживлены вдоль вершин петли или просто испущены на пункты петли. Это более обычно используется, когда есть потребность в прямом управлении предоставлением позиции частицы разработчик, способность следовать за вершинами петли означает, что они могут проектировать точные системы частицы в любой трехмерной форме, которой они желают.

В составляющих сроках у обоих эмитентов есть следующие параметры настройки вместе:

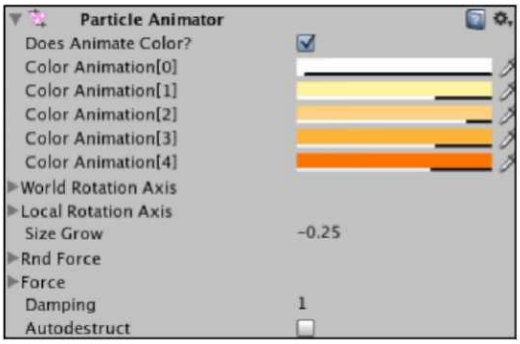
- Размер: видимый размер отдельной частицы
- Энергия: количество времени частица существует в мире перед авторазрушением
- Эмиссия: число частиц, испускаемых за один раз
- Скорости: скорость, на которой едут частицы

Мы будем смотреть на некоторые из более детальных параметров настройки, определенных для любого вида эмитента, поскольку мы идем далее в главе. Для задачи создания огня мы будем использовать эллиптического эмитента - это позволит более случайный эффект, поскольку это не привязано к петле.



Аниматор Частицы

Аниматор Частицы отвечает, как отдельные частицы ведут себя в течение долгого времени. Чтобы казаться динамичными, у частиц есть продолжительность жизни, после которой они авторазрушат. Как в нашем примере создания огня, должны идеально быть оживлены частицы так, чтобы внешность каждой отдельной частицы изменилась во время ее продолжительности жизни в мире. Поскольку огонь разрывается и усиливает реальный огонь - мы можем использовать компонент аниматора, чтобы применить различие цвета и видимости, так же как применить силы к частицам непосредственно.



Частица Renderer

Частица renderers определяет визуальное появление отдельных частиц. Частицы - эффективно квадратные эльфы (2-ая графика), и в большинстве случаев предоставлены в той же самой манере как трава, которую мы добавили к нашему ландшафту в Главе 2 - при использовании billboarding техники, чтобы дать появление всегда столкновения перед камерой. Частица renderers в Unity может показать частицы другими способами, но billboarding является соответствующим

для большинства использования.

Частица компонент Renderer также обращается с материалами, относилась к системе частицы. Поскольку частицы предоставлены просто, поскольку эльфы, применяя частицу shaders к материалам в renderer компоненте означают, что Вы можете создать иллюзию неквадратных эльфов при использовании альфа-канала окруженные структуры (прозрачности). Вот пример, который мы будем использовать коротко на нашем материале огня - темная область здесь - прозрачная часть:

В результате использования прозрачности для частиц у них больше нет квадратного появления - поскольку предоставленный 2-ыми эльфами обычно был бы. Комбинируя отдавание только видимой части этой структуры с более высокими ценностями эмиссии эмитента, эффект плотности может быть достигнут.

Частица renderers может также оживить частицы, используя УЛЬТРАФИОЛЕТОВУЮ мультипликацию, при использовании сетки изображений, чтобы эффективно обменять структуры во время продолжительности жизни частицы, но это немного более продвинуто чем область этой книги, таким образом рекомендуется, чтобы Вы обратились к руководству Unity для получения дополнительной информации об этом.

В резюме

Система частицы работает, потому что у нее есть ряд компонентов, которые сотрудничают, а именно, эмитент, создающий частицы, аниматор, определяющий их поведение / изменение в течение долгого времени, и renderer определение их эстетических материалов использования, и показывают параметры.

Теперь, мы будем смотреть на создание следующей части нашей учебной игры, которая достигнет высшей точки в зажигании огня, сделанного из двух систем частицы, один для огня и другого создания пера дыма.

Создание задачи

Наша существующая игра состоит из задачи для игрока закончить, чтобы войти в заставу - они должны собрать четыре батареи, чтобы привести дверь в действие, один из которых должен быть выигран,

выигрывая игру, которую мы добавили в предыдущей главе.

В настоящее время, войдя в заставу, игрок встречен со смыслом разочарования как нет ничего, чтобы быть найденным внутри. В этой главе мы изменим это, добавляя коробок спичек, который будет поднят игроком, когда они войдут в заставу. Мы тогда создадим наш огонь снаружи, который может только быть зажжен, если игрок несет коробок спичек. Таким образом, мы можем показать игроку ряд регистраций, ждущих, чтобы быть освещенными, принуждая им попытаться найти спички, заканчивая задачи вынутый (то есть, открывая дверь).

Чтобы создать эту игру, мы должны будем осуществить следующее в Unity:

- Определите местонахождение пакета актива, и добавьте модель дров к нашей сцене около заставы.
- Создайте системы частицы для огня и дым для того, когда огонь зажжен. Тогда заставьте их не испускать пока не вызвано сценарием.
- Настроенное обнаружение столкновения между характером игрока и дровами возражает, чтобы зажечь огонь, включая компонент эмитента.
- Добавьте модель спичек к заставе и настройте обнаружение столкновения, чтобы функционировать как коллекция объекта, и заставить это ограничить, может ли огонь быть зажжен.
- Используйте наш объект GUI `TextHint`, чтобы показать игроку намек, если они приближаются к огню без спичек.

Загрузка актива

Чтобы заставить активы, необходимые заканчивать это осуществление, определите местонахождение файла, названного `firePack.unitypackage` от нашей извлеченной кодовой связки. В Unity, пойдите в **Активы | Пакет Импорта,** и проведите к местоположению, Вы загрузили пакет к, и выбираете его.

Это импортирует несколько файлов, требуемых создать осуществление огня:

- Трехмерная модель регистраций походного костра
- Структура Пламени для нашего материала системы частицы огня
- Структура Дыма для нашего материала системы частицы дыма
- Звуковая скрепка треска огня
- Материальные папки для трехмерных моделей

Файлы будут импортированы в папку в **Проектной** группе под

названием **особенность Огня.**

Добавление груды регистрации

В папке **особенности Огня** в проектной группе Вы найдете модель названной **походным костром.** Выберите это, и измените **Коэффициент пропорциональности** на **0.5** в компоненте **Импортера FBX Инспектора.** Это гарантирует, что модель импортирована в нашу сцену в разумном размере по сравнению с другими объектами, уже представляют. Нажмите кнопку **Apply** у основания компонентов в **Инспекторе**, чтобы подтвердить это изменение.

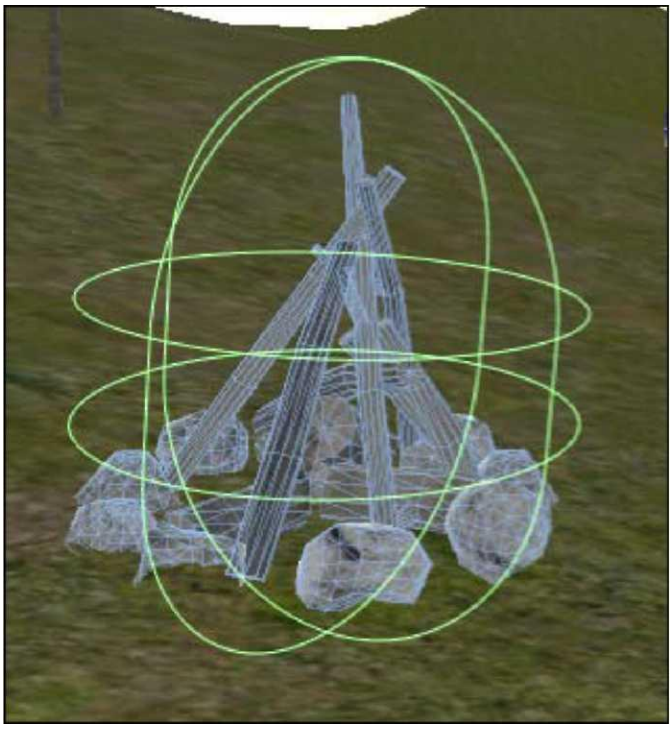
Теперь тяните эту модель в **Сцену,** и используйте **инструмент Transform (преобразовать)**, чтобы поместить это около заставы и кокосового ореха, застенчивого, которые уже присутствуют. Вот расположение, которое я использовал:

Теперь, потому что мы не хотим, чтобы наш игрок шел через эту модель, у этого должен быть коллайдер. Обычно, со сложными моделями, такими как это, мы использовали бы Импортера FBX, чтобы произвести коллайдеры петли для каждой отдельной петли в модели. Однако, при условии, что мы только должны гарантировать, что игрок врезается в этот объект, мы можем просто использовать краткий коллайдер вместо этого. Это будет работать точно также в цели и экономить на обработке власти, потому что процессор не должен будет построить коллайдер для каждой части.

Выберите объект **походного костра** в **Иерархии** и пойдите в **Компонент | Физика | Краткий Коллайдер.** Вы будете побуждены подтвердить, что Вы желаете разъединить этот случай от оригинала с высказыванием окна диалога**, Теряющим Заранее приготовленного Родителя просто**,

нажимают, **Продолжаются** как обычно.

Это даст Вам маленький сферически выглядящий коллайдер в основе модели огня. Мы должны увеличить размер этого, чтобы покрыть границу огня - в недавно добавленном **Кратком** компоненте **Коллайдера** в **Инспекторе,** установить **Радиус** в **2, Высоту** к **5,** и в ценностях для **Центра,** установить **Y** в ценность **1.5.** Ваш коллайдер должен теперь смотреть кое-что как то, что показывают в следующем скриншоте, и после тестирования игры, должно заставить игрока врезаться в объект.



Создание систем частицы огня

В этой секции мы создадим две различных системы частицы, один для огня и другого для дыма, происходящего от этого. При использовании двух отдельных систем мы будем иметь больше контроля над мультипликацией частиц в течение долгого времени, поскольку мы не должны будем оживлять от огня, чтобы курить в единственной системе.

Создание огня

Чтобы начать строить наши системы частицы огня, мы можем добавить объект игры со всеми тремя существенными компонентами частицы. Пойдите в **GameObject |, Создают Другой | Система Частицы.** Это создает новый объект игры в **Сцене/Иерархии** под названием **Система Частицы.** Нажмите *Возвращение* (Mac) или *F2* (PC) и тип, чтобы переименовать это к **FireSystem,** гарантируя, что нет никакого места на

название - это крайне важно для нашего scripting позже.

Примите во внимание, что на данном этапе, мы не должны поместить огонь, пока мы не закончили делать его. Поэтому, это может быть разработано в произвольном пункте в трехмерном мире, что редактор Unity поместил это в.
По умолчанию, у Вашей системы частицы есть появление частицы по умолчанию смягченных точек белого, в группе, не слишком несходной от светлячков в группе. Это просто демонстрирует большинство родовых параметров настройки для каждого компонента - эмитент частицы испускает и имеет скромное число частиц, аниматор просто заставляет частицы усилиться и, и у renderer еще нет никакого материала, настроенного.

Давайте пройдем каждый компонент теперь и давайте настроим их индивидуально. Вы будете в состоянии видеть предварительный просмотр системы частицы в представлении **Сцены**, так часы, поскольку Вы регулируете каждое урегулирование в **Инспекторе.**

Эллиптические Параметры настройки Эмитента Частицы

Начните, устанавливая ценность **Размера Минуты** в **0.5**, и **Макс Сайз** оценивают **2.** Наш огонь значительно больше чем подобные светлячку точки, замеченные в установках по умолчанию. Как со всеми параметрами настройки Минуты и Макса, эмитент породит частицы размера между этими двумя ценностями.

Теперь **Энергия Минуты** набора к **1** и **Макс Энерджи** к **1.5** - это - продолжительность жизни частиц, и в результате частицы продлятся между 1 и 1.5 секундами перед авторазрушением.

Установите ценность **Эмиссии Минуты** в **15**, и **Макс Эмишен** оценивают **20.** Это - определение ценности, сколько частиц находится в сцене в любой момент времени. Чем выше эти ценности идут, тем больше частиц будет на экране, давая больше плотности огню. Однако, частицы могут быть дорогими, чтобы отдать для центрального процессора. Так, вообще говоря, лучше держать ценности эмиссии как они к самому низкому урегулированию, которое Вы можете эстетически предоставить.

Установите ценность **Y Мировой Скорости** к **0.1.** Это заставит частицы повышаться во время их продолжительности жизни, поскольку огонь был бы из-за естественной высокой температуры.

Мы делаем это с **Мировой Скоростью** с объектами, которые должны всегда повышаться в мире игры. Если бы это было сделано на объекте, который был подвижен с **Местной Скоростью** и тем объектом, вращаемым во время игры, то ее местная Ось Y больше не стояла бы Мир Y.

Например, пылающий баррель с физикой может упасть, но его огонь должен все еще повыситься в Мировых сроках. В то время как наш походный костер не подвижен, это - хорошая практика, чтобы понять

различие между Местным и Мир здесь.

Тогда установите ценность **Y Скорости Rnd** (Случайной) к **0.2** - это заставит случайный случайный огонь прыгать немного выше чем другие.

Скорость Тангенса должна быть установлена в **0.2** в **X** и Осях Z, оставляя **Ось Y** на **0. Скорость Тангенса** определяет стартовую скорость отдельных частиц. Так, устанавливая маленькую ценность в **X** и **Z,** мы стимулируем огню огня горизонтально.

Скоростной Масштаб Эмитента может быть установлен в **0,** поскольку это только относится к движущимся системам частицы, поскольку он управляет, как быстро сами частицы перемещаются, если родительский объект системы перемещен. Поскольку наш огонь останется статичным, это можно оставить в **0.**

Моделируйте в Worldspace, может быть оставлен отсеянным, поскольку мы хотели бы, чтобы наши частицы были созданы относительно позиции системы, в противоположность мировой космической позиции.

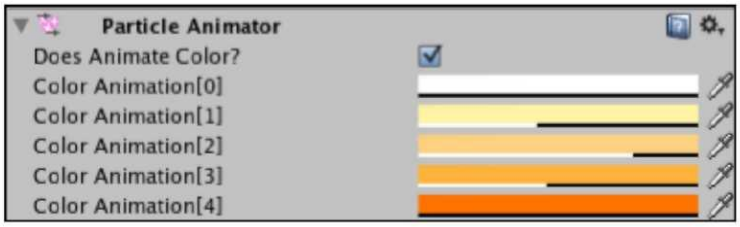
Один Выстрел можно оставить отсеянным, поскольку мы нуждаемся в наших частицах, чтобы течь непрерывно. **Один Выстрел** был бы более полезным кое в чем, таком как затяжка дыма от орудия например.

Эллиптические ценности могут быть установлены в **0.1** для всех топоров - это - маленькая ценность, но наша система частицы только в мелком масштабе - эллиптическая ценность для системы частицы пера дыма, например, должна будет быть намного более высокой.

Параметры настройки Аниматора Частицы

Затем мы должны настроить компонент Аниматора Частицы, который определит поведение наших частиц в течение их продолжительности жизни. Здесь мы должны будем заставить частицы усилиться и, и живой через цвета (оттенки красных/оранжевых), так же как применить силы к ним, которые заставят огонь прыгать и вздыматься в сторонах более реалистично.

Начните, гарантируя, что это **Действительно Оживляет Цвет**, отобран, который заставит пять **Цветных** коробок **Мультипликации** играть роль. Теперь пройдите каждую коробку, нажимая на цветное место - и в цветном сборщике, который появляется, выбранные цвета, которые исчезают от белого в цвете **Мультипликация [0]** через к темно-оранжевому в цвете **Мультипликация [4].**



Ваша система частицы в представлении **Сцены** должна теперь

вздыматься более естественно и оживлять через цвета и альфа-ценности, которые Вы определили.
[193]
Поскольку Вы делаете это, также устанавливаете (альфа) ценность у основания цветного сборщика к ценности так, чтобы **у Цветной Мультипликации [0]** и **Цветной Мультипликации [4]** была альфа **0,** и государства промежуточный постепенно уменьшают и назад. Альфа иллюстрирована в **Инспекторе** белым/черным пятном ниже каждого цветного блока. См. следующий скриншот для визуального представления какой

Поскольку Вы делаете это, также устанавливаете (альфа) ценность у основания цветного сборщика к ценности так, чтобы **у Цветной Мультипликации [0]** и **Цветной Мультипликации [4]** была альфа **0,** и государства промежуточный постепенно уменьшают и назад. Альфа иллюстрирована в **Инспекторе** белым/черным пятном ниже каждого цветного блока. См. следующий скриншот для визуального представления того, что Вы должны сделать здесь:

Эффект **Цветной** системы **Мультипликации** может выглядеть более или менее эффективным в зависимости от того, относились ли материалы к специфическому **Shaders** использования системы Вашей частицы - некоторый показ красит больше так чем другие; например, shaders с прозрачностью, может казаться, показывает цвета менее эффективно.

После отъезда параметров настройки **Оси Вращения** в их неплатеже **0** во всех топорах, **Размер** набора **Растет** к ценности**-0.3.** Мы устанавливаем это в минус ценность, чтобы заставить частицы сжиматься во время их продолжительности жизни, давая более динамический взгляд огню.

Затем мы добавим силы, чтобы дать более реалистическое движение огню. Силы различны к скоростям, добавленным в параметрах настройки **Эмитента**, поскольку они применены, когда частицы - иллюстрировавший примерами выз повышение скорости после замедлением. Расширьте параметр **Силы Rnd**, нажимая на серую

стрелку налево от этого и поместите ценность **1** в **X** и Оси Z. Тогда расширьте параметр **Силы** таким же образом, и установите ценность **Оси Y** в **1.**

Установите ценность для **Демпфирования** к **0.8.** Демпфирование определяет количество, которое частицы замедляют во время их продолжительности жизни. С ценностью по умолчанию 1 значения не происходит никакое демпфирование, ценности между 0 и 1 причиной замедляются, 0 являющийся самым интенсивным замедлением. Мы устанавливаем умеренную ценность 0.8 так, чтобы наши частицы не замедлились слишком противоестественно.

Заключительное урегулирование, **Авторазрушьте,** может быть оставлен отсеянным. Все частицы непосредственно естественно авторазрушают в конце их продолжительности жизни, но авторазрушать урегулирование здесь имеет отношение с родительским объектом игры непосредственно - если отобрано, и все частицы в системе авторазрушили, тогда объект игры будет разрушен. Это только играет роль, используя урегулирование **Выстрела Того** в компоненте эмитента - в примере взрыва орудия, разработчик вероятно иллюстрировал бы примерами случай системы частицы с одним выстрелом с, авторазрушают отобранный. Это означает, что, как только это было создано, как только все частицы умерли, объект игры будет разрушен, таким образом экономя ресурсы системы, такие как центральный процессор, GPU, и RAM.

Частица параметры настройки Renderer

В нашей системе частицы огня мы просто должны применить заштрихованный частицей материал, содержащий огонь, графический, Вы загружали и импортировали. Но прежде, чем мы делаем это, мы гарантируем, что **Частица Renderer** настроена правильно в нашей цели.

Поскольку частицы или огонь огня должны технически излучать свет, мы отсеем и **Бросок** и **Получим** тени, чтобы гарантировать, что никакая тень не набрана или медведем огня в памяти, что это только действительно для пользователей Unity Про версия, поскольку Инди не показывает динамические тени во время письма.

В настоящее время нет никаких материалов, относился к этой системе частицы, таким образом нет никаких записей в области **Материалов** - мы исправим это коротко. Затем, гарантируйте, что **Скоростной Масштаб Камеры** установлен в **0** - это только использовалось бы, если бы мы не были рекламным щитом, отдающим наши частицы. Если бы мы запланировали протянуть частицы, то мы использовали бы это урегулирование, чтобы определить, сколько из движения камеры эффекта имело на протяжении частицы.

Частицы Протяжения должны быть установлены в **Рекламный щит** как обсужденное более раннее обеспечение, что независимо от того, откуда игрок рассматривает огонь, частицы оттянуты, стоя перед ними. Поскольку **Длина** и **Скоростной** масштаб только используются в

протянутых частицах, мы можем счастливо оставить эти два параметров настройки в с 0 изменениями, эти ценности не будут затрагивать billboarded частицы.

Урегулирование финала к рассмотренный имеет в памяти, что мы не используем УЛЬТРАФИОЛЕТОВЫЙ, Мультипликация **Размер Макса Партикла.** Это урегулирование управляет, насколько большой частица может быть относительно высоты экрана. Например если установлено в **1,** частицы могут иметь размер до высоты экрана, в **0.5** они могут быть до половины высоты экрана, и так далее. Поскольку наши частицы огня никогда не должны будут иметь размер столь же большой как высота экрана - мы можем оставить это урегулирование на его неплатеже **0.25.**

Добавление материала

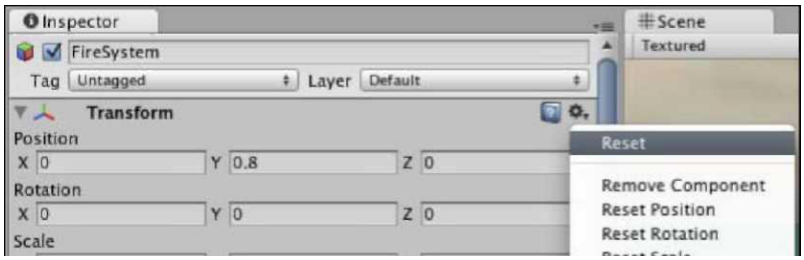
Теперь, когда наша система частицы настроена, все, что оставляют сделать, должен создать материал для этого использующий нашу структуру огня. Выберите папку **особенности Огня** в **Проектной** группе, и нажмите на кнопку **Create** наверху группы, и выберите **Материал.** Это заставит новый актив под названием **Новый Материал** в папке просто переименовать это **Пламя,** и затем выбрать это, чтобы видеть его свойства в **Инспекторе.**

От **Shader** опускаются меню, выбирают **Частицы | (мягкая) Добавка.** Это даст нам, основанная на альфе частица с мягким отдает структуры, которую мы применяем. Теперь, тяните структуру под названием **Огонь 1** от папки **Особенности Огня** в **Проектной** группе, и понизьте это на пустой квадрат направо от **Структуры Частицы**, где это в настоящее время говорит
Ни один (Texture2D).

Чтобы применить этот материал, просто тяните материал от папки **особенности Огня** в **Проектной** группе и понизьте это на объект **FireSystem** в группе **Иерархии**.

Расположение FireSystem

Чтобы поместить систему частицы огня более легко, мы должны будем сделать это ребенком объекта **походного костра** уже в нашей сцене. В группе **Иерархии**, тяните объект **FireSystem**, и понизьте это на родительский объект, названный **походным костром**, чтобы сделать это детским объектом. Тогда перезагрузите его позицию, нажимая на изображение **Винтика** направо от компонента **Transform (преобразовать) FireSystem,** и выберите **Сброс:**



Теперь, когда Вы перезагрузили позицию, Ваша система частицы будет непосредственно в центре ее родительского объекта; поскольку Вы заметите, это слишком низко снижается. Если Вы не можете видеть это, то выберите объект в **Иерархии** и парении Ваш курсор мыши по представлению **Сцены** и нажмите *F*, чтобы сосредоточить Ваш взгляд относительно этого. Все еще в компоненте **Transform (преобразовать)**, набор позиция **Оси Y** к **0.8**, чтобы поднять это немного.

Время, чтобы Проверить!

Нажмите кнопку **Play**, чтобы проверить игру теперь и восхититься Вашей ручной работой! Не забудьте нажимать **Игру** снова, чтобы прекратить проверять прежде, чем Вы продолжите.

Создание дыма

Как говорится, "нет дыма без огня", и наоборот. С этим в памяти, мы будем нуждаться в пере дыма, появляющемся из выше нашего походного костра, если это должно выглядеть реалистичным вообще.

Начните, добавляя новую систему частицы к Сцене; пойдите в **GameObject |, Создают Другой | Система Частицы.** Переименуйте **Систему Частицы**, которую Вы только что сделали в Иерархии к **SmokeSystem** (снова, отметьте, что я не использую место на это объектное название).

Эллиптические параметры настройки Эмитента Частицы

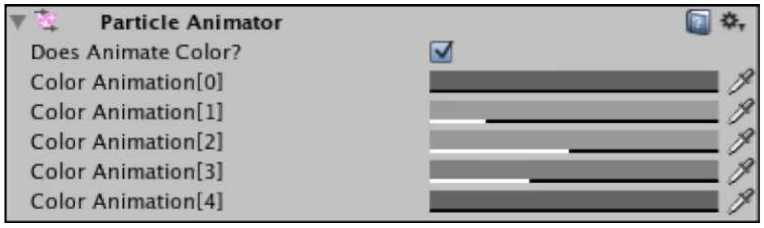
Поскольку мы уже обсудили значения параметров настройки в предыдущем шаге, теперь Вы можете просто использовать следующий список и наблюдать изменения, поскольку Вы делаете их. Любые параметры настройки, не перечисленные, нужно оставить при их установке по умолчанию:

- **Размер Мин: 0.8(Min Size: 0.8)**
- **Макс Сайз: 2.5(Max Size: 2.5)**
- **Энергия Мин: 8(Min Energy: 8)**
- **Макс Энерджи: 10(Max Energy: 10)**
- **Эмиссия Мин: 10(Min Emission: 10)**
- **Макс Эмишен: 15(Max Emission: 15)**

- **Мировая Скорость Y: 1.5(World Velocity Y: 1.5)**
- **Скорость Rnd Y: 0.2(Rnd Velocity Y: 0.2)**
- **Скоростной Масштаб Эмитента: 0.1(Emitter Velocity Scale: 0.1)**

Параметры настройки Аниматора Частицы

Настройте своего аниматора частицы, чтобы оживить через оттенки серых, как показано в изображении ниже:



Снова, здесь Вы должны заметить, что частицы должны быть оживлены между двумя нулевыми альфами так, чтобы они начали и закончились незримо. Это избежит визуальной "популярности", поскольку частицы удалены - если они видимы, когда они удалены, удаление намного более примечательное, и менее естественным.

Теперь измените следующие параметры настройки:

- **Размер Растет: 0.2(Size Grow: 0.2)**
- **Сила Rnd: (1.2, 0.8, 1.2)(Rnd Force: (1.2, 0.8, 1.2))**
- **Сила: (0.1, 0 0.1)(Force: (0.1, 0 0.1))**
- **Авторазрушьте(Autodestruct):** Не отобранный

Частица параметры настройки Renderer

Мы создадим материал, чтобы относиться к дыму коротко, но сначала, гарантировать, что следующие параметры настройки применены к частице renderer компонент:

- **Бросьте/Получите Тени(Cast/Receive Shadows):** оба отсеяли
- **Частицы Протяжения: Рекламный щит(Stretch Particles: Billboard)**
- **Скоростной Масштаб Камеры, Характерный радиус взаимодействия, Скоростной Масштаб: 0(Camera Velocity Scale, Length Scale, Velocity Scale: 0)**
- **Размер Макса Партикла: 0.25(Max Particle Size: 0.25)**

Теперь, следуйте, шаг назвал **Добавление Материала** при **Создании огня** выше. Однако, на сей раз, назовите свой материальный **Дым** и назначьте файл структуры, названный **дымом 1**, и понизьте материал, который Вы делаете на объект игры **SmokeSystem**.

Расположение

Повторите шаг для того, чтобы поместить огня для системы дыма при перемещении и понижая объект **SmokeSystem** на родителя **походного костра** в **Иерархии.** Тогда используя изображение **Винтика**, выберите **Сброс,** и установите ценность позиции **Y** в **0.9.** Нажмите кнопку **Play** теперь, и Ваш походный костер должен осветить, и перо дыма должно повыситься, смотря кое-что как следующий скриншот. Как всегда, не забудьте нажимать **Игру** снова, чтобы прекратить проверять.

Добавление аудио к огню

Теперь, когда наш огонь выглядит эстетически приятным, мы должны будем добавить атмосферный звук треска огня. В пакете Вы загружали, Вам предоставляют звуковой файл, названный `fire_atmosphere`, который является моно звуковой скрепкой, которая может быть закреплена петлей. Быть моно, когда мы применяем этот файл к звуковому источнику, убегая от огня, будет означать, что аудио исчезнет с расстоянием, которое должно иметь смысл игроку.

Выберите объект **походного костра** в **Иерархии,** и пойдите в **Компонент | Аудио | Звуковой Источник.** Это добавляет звуковой исходный компонент к объекту, так теперь просто тяните **fire_atmosphere** звуковую скрепку от папки **особенности Огня** в **Проектной** группе, и понизьте его на **Звуковой** параметр **Скрепки Звукового Источника** в **Инспекторе.** Чтобы закончить, просто выберите коробку рядом с параметром **Петли**.

Нажмите кнопку **Play** и приблизьтесь к огню, прислушиваясь к звуку треска огня; тогда уйдите, и слушайте исчезающий эффект.

Зажигание огня

Теперь, когда наш огонь полон, давайте выключать системы частицы и аудио так, чтобы это не было освещено в начале игры.

- Выберите объект **походного костра** в **Иерархии,** и в **Звуковом Источнике**
  компонент в **Инспекторе,** отсейте **Игру На Активном** параметр.
- Выберите детский объект **FireSystem** походного костра, и в **Эмитенте Частицы**
  компонент, отсейте **Испускать** параметр.
- Выберите детский объект **SmokeSystem** походного костра, и в компоненте **Эмитента Частицы**, отсейте **Испускать** параметр.

Теперь мы должны создать коллекцию спичек и показ интерфейса факта, у нас есть спички. Чтобы сделать это, мы должны будем сделать следующее:

- Добавьте, что модель **спичечной коробки** загружала ранее к заставе, делая это цель из получения входа.
- Сделайте Структуру GUI, используя структуру в папке **особенности Огня**, и сохраните это как prefab.
- Добавьте к существующему сценарию **PlayerCollisions**, строящему в обнаружении столкновения для спичечной коробки, чтобы разрушить это и иллюстрировать примерами спичечную коробку prefab GUI.

Добавление спичек

Выберите модель **спичечной коробки** в папке **особенности Огня Проектной** группы, тяните это к **Иерархии,** и понизьте это на объект **заставы**, делая это детский объект. Это облегчит позиции.

В компоненте **Transform (преобразовать)** для спичечной коробки в **Инспекторе,** устанавливает ценности **Позиции** в **(0, 3.5, 0).** Теперь установите все ценности для **Масштаба** в том же самом компоненте к **5.**

Поскольку мы должны собрать этот объект, это будет нуждаться в коллайдере, чтобы обнаружить столкновения между этим и игроком. Со **спичечной коробкой**, все еще отобранной, пойдите в **Компонент | Физика | Коллайдер Коробки,** и нажмите, **Добавляют** когда подарено **Проигрывающее Заранее приготовленное** окно диалога. Чтобы избежать игрока, врезающегося в объект, мы поместим этот коллайдер в более аккуратный способ. Чтобы сделать это, просто выберите коробку рядом с, **Более аккуратное** урегулирование в компоненте **Коллайдера Коробки Инспектора.**

Чтобы сделать этот объект более очевидно коллекционируемым пунктом, мы добавим сценарий, мы имели обыкновение вращать

батареи в нашей игре. Пойдите в **Компонент |, Сценарии | Вращают Объект.** Тогда, в недавно добавленном компоненте сценария в **Инспекторе,** устанавливает Variables **Количества Вращения** в **2.**

Создание Спичек GUI

Выберите файл структуры **MatchGUI** в папке **особенности Огня** в **Проектной** группе. Пойдите в **GameObject |, Создают Другой | GUITexture.** Мы не должны поместить эту структуру на экран, поскольку это может быть сделано, когда мы иллюстрируем примерами это как prefab.

Выберите папку **особенности Огня** в **Проектной** группе, и от кнопки **Create**, выберите **Prefab.** Переименуйте этот prefab к **MatchGUIprefab** и затем drag and drop (перетащили и опустили) объект **MatchGUI** от **Иерархии** на этот пустой prefab в **Проектной** группе.

Наконец, выберите оригинальный объект **MatchGUI** в **Иерархии** снова, и удалите это из сцены, нажимая *Команду* + *Клавиша Backspace* (Mac) или *Удалите* (PC).

Сбор спичек

От папки **Сценариев** в Проектной группе, откройте сценарий по имени **PlayerCollisions.** Чтобы помнить, собрали ли мы спички или нет, мы добавим логическую переменную к сценарию, который будет установлен в `верный`, как только спички собраны. Добавьте следующую линию к вершине сценария:

```
private var haveMatches : boolean = false;
```

Мы будем также нуждаться в общественном Variables участника, чтобы представить prefab Спичек Структура GUI так, чтобы мы могли иллюстрировать примерами это, когда игрок собирает спичечную коробку. Добавьте следующий Variables к вершине сценария:

```
var matchGUI : GameObject;
```

Теперь прокрутите вниз к `OnTriggerEnter ()` Functions сценария, и ниже существующего, добавьте в следующем `если` утверждение:

```
if(collisionInfo.gameObject.name == "matchbox"){
Destroy(collisionInfo.gameObject);
haveMatches=true;
audio.PlayOneShot(batteryCollect);
var matchGUIobj = Instantiate(matchGUI, Vector3(0.15,0.1,0),
transform.rotation);
matchGUIobj.name = matchGUI;
    }
```

Здесь мы проверяем наш существующий `collisionInfo` параметр,

который регистрируется, любые объекты столкнулись с. Мы проверяем, содержит ли `collisionInfo` объект, названный `спичечной коробкой` при использовании `gameObject.name` и сравнения этого к последовательности текста `"спичечная коробка"`. Если дело обстоит так, то мы выполняем следующие команды:

- Разрушьте объект `спичечной коробки`, обращаясь к потоку `gameObject` в `collisionInfo.`
- Установите `haveMatches` Variables в `истинный.`
- Играйте звуковую скрепку, назначенную на наш `batteryCollect` Variables. Поскольку игрок уже привык к этому звуку для того, чтобы собрать объекты, имеет смысл снова использовать это.
- Instantiate (Проиллюстрировать) случай объекта игры, назначенного на общественный Variables участника `matchGUI,` в позиции экрана (0.15, 0.1, 0), принимая во внимание, что `Ось Z` в 2-ых объектах просто для иерархического представления, следовательно ценность 0. Мы также использовали команду `transform.rotation`, чтобы унаследовать вращение от родительского объекта

  поскольку вращение является несоответствующим в 2-ых объектах.
- Назовите недавно иллюстрировавший примерами объект при использовании Variables `matchGUIobj,` который мы создаем в линии экземпляра. Это будет использоваться позже, удаляя GUI из экрана.

Пойдите в **Файл |, Экономят** в редакторе сценария теперь и переключаются назад на Unity. Нажмите кнопку **Play** и гарантируйте, что после входа в заставу, Вы можете поднять спички, идя в них. Структура спичек должна также появиться в более низком, оставленном экрана. Теперь мы будем использовать `haveMatches` Variables, чтобы решить, может ли огонь быть зажжен или нет.

Поджигая

Чтобы зажечь огонь вообще, мы должны проверить на столкновения между игроком и объектом походного костра. Для этого, возвратитесь к редактору сценария, чтобы изменить сценарий **Столкновений Игрока**, или если Вы закрыли его, вновь откройте его от папки **Сценариев Проектной** группы.

Определите местонахождение `OnControllerColliderHit ()` Functions вводная линия:

```
function OnControllerColliderHit(hit:
```

`ControllerColliderHit){`тогда спускаются к линии ниже этого.

Добавьте следующий `если` утверждение:

```
if(hit.collider.gameObject ==
GameObject.Find("campfire")){

}
```

Это проверит, что мы поразили объект **походного костра**. Однако, мы должны выполнить другую проверку относительно того, несет ли игрок спички, так внутри, что, если утверждение, поместите следующее:

```
    if(haveMatches){
      haveMatches = false;
      lightFire(); }else{
      TextHints.textOn=true;
      TextHints.message = "I'll need some matches to light
this camp fire..";
    }
```

Здесь мы проверяем, установлен ли haveMatches Variables в истинный, и если не (еще), мы используем сценарий TextHints, чтобы включить наш TextHint GUI и показать предложение игроку на экране.

Если haveMatches верен, мы вызываем функцию, названную lightFire (), который мы должны будем написать затем, чтобы включить системы частицы и аудио. Свиток к основанию сценария, и добавляет следующий Functions перед @script линией:

```
function lightFire(){
  var campfire  : GameObject =
  GameObject.Find("campfire"); var campSound :
  AudioSource = campfire.GetComponent(AudioSource);
  campSound.Play();

  var flames  : GameObject =
  GameObject.Find("FireSystem");
  var flameEmitter  :  ParticleEmitter =
flames.GetComponent(ParticleE mitter);
    flameEmitter.emit = true;

var smoke  : GameObject = GameObject.Find("SmokeSystem");
var smokeEmitter  :  ParticleEmitter =
smoke.GetComponent(ParticleEm itter);
    smokeEmitter.emit = true;

    Destroy(GameObject.Find("matchGUI"));
}
```

В этом Functions мы должны были выполнить четыре операции:

Старт звуковой петли для огня,

Включающего систему частицы Огня,

Включающую систему частицы Дыма

Удаление на экране Matches GUI, чтобы предположить, что спички "использовались"

В первых трех операциях мы должны были обратиться к объекту, создавая Variables, чтобы представить это, например:

```
var campfire  : GameObject = GameObject.Find("campfire");
```

Таким образом здесь мы назвали Variables, установили его тип данных в `GameObject`, и установили это равный объекту игры, названному `походным костром`, используя команду `Находки`.

Затем мы обратились к определенному компоненту того объекта при использовании Variables, только установленного, и команда `GetComponent`:

```
var campSound : AudioSource =
campfire.GetComponent(AudioSource);
```

Снова, мы устанавливаем новый Variables, чтобы представить компонент, устанавливая тип данных в `AudioSource`, и наконец мы используем этот Variables, чтобы вызвать команду:

```
campSound.Play();
```

Поскольку команда `Игры` - определенная команда к `Звуковым Исходным` компонентам, Unity знает точно, что сделать и просто играет звуковую скрепку, назначенную в **Инспекторе.**

Этот подход повторен во второй и третьей операции, кроме этого времени, мы обращаемся к `испускать` параметру компонента **Эмитента Частицы**, например:

```
flameEmitter.emit = true;
```

Последняя операция в этом Functions - `Разрушение ()` команда, которая просто находит объект GUI, показывающий спички, который называют, `matchGUI` на экземпляр (обратитесь к **Сбору** секции **спичек**).

Пойдите в **Файл |, Экономят** в редакторе сценария теперь, и переключаются назад на Unity.

Поскольку мы создали общественный Variables участника для **MatchGUIprefab** в сценарии **PlayerCollisions**, он будет нуждаться в назначении. В **Иерархии,** крах любые расширенные родительские объекты, нажимая на их серую стрелку, затем выбирают объект **First Person Controller**, чтобы видеть его список компонентов в **Инспекторе.**

Найдите **Столкновения Игрока (сценарий)** компонент и затем тяните актив **MatchGUIprefab** от папки **особенности Огня** в **Проектной** группе,

и понизьте это на неназначенный общественный Variables участника под названием **Состязание GUI.**

Поздравления! Ваш элемент освещения огня теперь полон. Пойдите в **Файл |, Экономят Сцену** в Unity, чтобы гарантировать, что Ваш проект современен.

Тестирование и подтверждение

Как с любой новой особенностью Вашей игры, тестирование крайне важно. В Главе 9 мы будем смотреть на оптимизацию Вашей игры и обеспечение, что тест строит работу, поскольку они ожидаются к, наряду с различными вариантами для того, чтобы предоставить Вашу игру.

Пока, Вы должны гарантировать, что Ваша игра функционирует должным образом пока. Даже если у Вас нет никаких ошибок, показывая в части пульта Unity *(Команда +, Изменение + C* показывает эту группу по Mac, *Ctrl + Изменение + C* на PC), Вы должны все еще удостовериться, что, поскольку Вы играете через игру, никакие ошибки не происходят, поскольку игрок использует каждую часть игры.

Нажмите кнопку **Play** и игру через коллекцию батареи, кокосовую застенчивую игру, коллекцию состязания, и освещение огня, чтобы гарантировать, что все элементы в настоящее время работают. Если какие-нибудь ошибки происходят, то вернулись к Вашему scripting и проверяют, что все соответствует кодовым спискам в этой книге.

> Если Вы столкнетесь с ошибками, проверяя, то кнопка **Pause** наверху редактора Unity позволит, что Вы, чтобы сделать паузу, играть, и смотреть на ошибку перечисляли в пульте. Сталкиваясь с ошибкой, просто щелкните два раза на этом в **Console,** и Вы будете взяты к линии сценария, который содержит ошибку - или по крайней мере туда, где сценарий сталкивается с проблемой.
>
> Отсюда, Вы можете диагностировать ошибку или проверить Ваш сценарий против Unity scripting ссылка, чтобы удостовериться, что у Вас есть правильный подход. Если Вы все еще застреваете с ошибками, просите помощь на форумах сообщества Unity или в канале IRC. За дополнительной информацией, посетите следующую страницу:
> <http://unity3d.com/support/community>

Резюме

В этой главе мы смотрели на использование частиц, чтобы дать более динамическое чувство нашей игре. Частицы используются в широком множестве различных ситуаций игры, от автомобиля и выхлопа космического корабля на оружие и пар сапуна, и лучший способ укрепить то, что мы только что изучили, должен экспериментировать. Есть много параметров, чтобы играть с и, как таковое, лучшие

результаты найдены, вынимая некоторое время из проекта только, чтобы видеть, каких эффектов Вы можете достигнуть.

В следующей главе мы будем смотреть на создание меню для Вашей игры, и это вовлечет scripting с Classes Unity GUI, так же как использующий активы GUI Кожи, чтобы разработать и создать поведения для Ваших интерфейсов. Classes GUI - определенная часть двигателя Unity, который используется определенно для того, чтобы сделать меню,HUD (Возглавляет Показы), и когда использующийся в соединении с активами HUD GUI, становится полностью визуально настраиваемым и многократного использования. Это - то, потому что к активам Кожи можно относиться так много сценариев Classes GUI, как вам угодно создающих последовательный стиль проекта всюду по Вашим проектам Unity.

Проект Меню

Чтобы создать округленную игру в качестве примера, в этой главе, мы будем смотреть на создание отдельной сцены к нашей существующей сцене острова, чтобы действовать как меню. Проект меню в Unity может быть достигнут во многих способах использовать комбинацию встроенных поведений и 2-ого предоставления структуры.

Названия игры, которые Вы вводите, должны быть добавлены, используя Структуры GUI, такие как экран всплеска с эмблемами разработчика или загружая экран. Однако, добавляя интерактивные меню, Вы должны рассмотреть два разных подхода, одно использование Структуры GUI - область, которую мы уже исследовали, осуществляя наши Спички GUI в предыдущей главе и перекрестие в Главе 6, и другое использование classes UnityGUI, включая GUI активы skin.

В этой главе Вы изучите следующее:

- Создание двух разных подходов, чтобы соединять проект
- Контроль компонентов Texture GUI с подготовленными событиями мыши
- Письмо основного script UnityGUI
- Параметры настройки для активов skin GUI

- Загружая сцены, чтобы провести меню и загружая

уровень игры Мы изучим два подхода для того, чтобы добавить интерактивные меню:

- **Подход 1:** Структуры GUI и случай мыши scripting
  Первый подход вовлечет создание GUI, которому Texture возражает, и с scripting основанный на мыши событий мыши, мыши вниз, и мыши например, обменивая структуры, используемые этими объектами. С этим подходом меньше scripting обязано создавать кнопки непосредственно, но все действия должны управляться и слушаться через scripts.
- **Подход 2:** Classes UnityGUI scripting и skins GUI
  Второй подход возьмет больше script - интенсивная методология и вовлечет производство нашего всего меню, используя scripts, в противоположность созданию отдельных объектов игры для пунктов меню в подходе 1.
  С этим подходом больше scripting обязано создавать элементы меню первоначально, но активы skin GUI могут быть созданы, чтобы разработать появление и назначить поведение через **Inspector** для событий мыши.

Последний подход - вообще более принятый метод создания полных меню игры, поскольку это дает разработчику больше гибкости. Пункты самого меню установлены в script, но разрабатывали использование GUI skins в подходе, сопоставимом HTML и развитию CSS в веб дизайне - **CSS (Льющийся каскадом Таблицы стилей)** управление формой с HTML, обеспечивающим содержание.

Работая с skins GUI, Вы также заметите несколько соглашений CSS неожиданное возникновение, таких как края, дополнение, и оправдание. Это будет полезно, если Вы будете иметь какой-нибудь опыт веб разработки, но не будете волноваться, не является ли это случаем-GUI, параметры настройки skin разработаны, как большинство особенностей Unity, чтобы потребовать минимального предшествующего знания.

Интерфейсы и меню

Меню обычно используются, чтобы настроить средства управления и приспособить параметры настройки игры, такие как графика и звук, или загрузить спасенные государства игры. В любой данной игре крайне важно, что сопровождающее меню не препятствует доступу к игре или любым из ее параметров настройки. Когда мы думаем о большой игре, мы всегда помним это для фактической игры непосредственно, а не меню - если они не были особенно интересны, или особенно ужасно проектировали.

Много игр стремятся связать меню своей игры с проектом игры или темами. Например, в превосходном *World Of Goo* 2-ого Мальчика, курсор

изменен на форму шара липкой вещи со следом, который следует за этим в меню и игре, связывая визуальное понятие игры интерфейсом игры. Это - хороший пример, поскольку сама игра уже дает игроку кое-что, чтобы играть, поскольку они проводят через вводное меню.

В *LittleBigPlanet* Молекулы СМИ это понятие взято к другому уровню, давая игроку меню, которое требует, чтобы они узнали, как управлять характером игрока прежде, чем они проведут через меню игры.

Как с любым проектом, последовательность - ключ, и в нашем примере, мы будем гарантировать, что мы поддерживаем ряд цветов дома и последовательного использования книгопечатания. Вы, возможно, столкнулись с плохо разработанными играми в прошлом и заметили, что они используют слишком много шрифтов, или сталкивающиеся цвета, который данный, что меню игры - первая вещь, которую игрок будет видеть - является ужасно нерасполагающим фактором в создании приятной игры и коммерчески жизнеспособный.

Структуры, которые будут использоваться в создании меню, доступны в кодовой связке, обеспеченной на packtpub.com <http://packtpub.com> (`www.packtpub.com/files/code/8181_Code.zip` <http://www.packtpub.com/files/code/8181_Code.zip>).

Определите местонахождение файла по имени `Menu.unitypackage` от извлеченных файлов и затем возвратитесь к Unity. Пойдите в **Активы | пакет Импорта,** и введите активы, рассматривая и выбирая пакет, который Вы извлекли.

Почтовый импорт, Вы должны видеть, что Вы добавили папку под названием **Меню** к **Project** panel. В этой папке Вы найдете следующее:

- Структуры для главного названия игры и трех кнопок - **Игра, Инструкции,** и **Оставленный**
- Звуковая скрепка интерфейса подает звуковой сигнал для кнопок

Создание главного меню

В этой секции мы будем брать нашу существующую работу проекта острова непосредственно и использовать это как фон для наших меню. Это дает игрокам предварительный просмотр вида окружающей среды, они будут исследовать и устанавливают визуальный контекст для игры.

Дублируя наш существующий остров, рассматриваемый от расстояния, используя отдаленную камеру, мы наложим 2-ые элементы интерфейса, используя два ранее упомянутых подхода.

Создание сцены

Для нашего меню мы будем стремиться использовать окружающую среду острова, которую мы уже создали. Помещая остров на заднем

плане нашего меню, мы эффективно дразним игрока с окружающей средой, которую они могут исследовать, если они начинают играть в игру. Визуальные стимулы, такие как они могли бы казаться незначительными, но они могут работать хорошо как путь подсознательного поощрения игрока попробовать игру.

Визуальный пример

Вот то, на что будет похоже наше меню, когда это будет закончено. Этот пример взят от подхода 1; приблизьтесь 2's, пункты меню будут выглядеть различными:

Дублирование острова

Чтобы начать, мы снова используем нашу сцену **Island Level**, которую мы создали. Чтобы сделать вещи легче справиться, мы будем группироваться существенные активы окружающей среды, чтобы держать их отдельными от объектов, в которых мы не нуждаемся, такие как застава и батареи. Этот путь, когда дело доходит до дублирования уровня, мы можем просто удалить все кроме группы, которую мы собираемся сделать. Группировка в Unity столь же легка как гнездящиеся объекты как дети под пустым родительским объектом.

Группировка объектов окружающей среды

- Пойдите в **Объект Игры |, Создают Пустой.**
- Это делает новый объект названным просто **GameObject,** так что переименуйте это как **Окружающую среду** теперь.
- В группе **Hierarchy**, drag and drop (перетащили и опустили) **Направленный Свет, Дневной свет, Простую Воду,** и объекты **Ландшафта** на **Окружающую среду** пустой объект. Теперь мы готовы дублировать этот уровень, чтобы создать наше меню в отдельной сцене.

Дублирование сцены

Следуйте за этими шагами, чтобы дублировать сцену:

1. Гарантируйте, что сцена **Island Level** спасена, идя в **Файл |, Экономят Сцену,** и затем выбирают актив **Island Level** в **Project** panel.
2. Пойдите, чтобы **Отредактировать | Дубликат,** или сокращенная *Команда* клавиатуры использования + *D* (Mac) или *Ctrl* + *D* (PC). Дублируя, Unity просто увеличивает названия объекта/актива с числом, таким образом Ваш дубликат назовут, **Island Level 1** - гарантируют, что дубликат отобран в **Project** panel и переименовывать это к **Меню** теперь.
3. Загрузите эту сцену, чтобы начать воздействовать на это, щелкая два раза на этом в **Project** panel.
4. С открытой сценой **Меню** мы можем теперь удалить любые объекты, в которых мы не нуждаемся. Если Вы находитесь на Mac, то поддерживаете на нужном уровне *командную клавишу*, или если на PC, то поддерживаете на нужном уровне *клавишу CTRL* и выбираете все объекты в **Hierarchy** за исключением группы **Окружающей среды**, нажимая на них один за другим.
5. Теперь удалите их из этой сцены при использовании сокращенной *Команды* клавиатуры + *Клавиша Backspace* на Mac, или *Изменение* + *Удаляет* на PC.

Как показано в предыдущем скриншоте, наше меню будет выстрелом острова от вдалеке помещенного в более низкое право на экран, с названием и меню, добавленным чрезмерно в пустом месте, таком как небо и море. Чтобы достигнуть этого, мы должны будем ввести новую камеру сцене, поскольку ранее единственная камера была той, приложенной к **Первому Диспетчеру Человека.** Как в настоящее время нет никакой камеры в сцене, **Game** view должен теперь быть полностью чистым, поскольку нет никакого viewport на нашем трехмерном мире.

Чтобы создать камеру, пойдите в **GameObject |, Создают Другой | Камера.** Гарантируя, что этот новый объект **Камеры** отобран в **Hierarchy,** войдите в следующее **Положение** в компонент **Transform (преобразовать)** в **Inspector - (150, 250,-650).**

Отмена mip картографии

Картография Mip - способ произвести меньшие версии структур, чтобы спасти работу, когда они рассматриваются издалека в двигателе игры. Это может улучшить работу на 33 процента в двигателе Unity. Однако, это применяется только к структурам, которые являются частью трехмерного мира - не используемые как 2-ые структуры, такие как те, мы собираемся использовать для нашего названия и трех кнопок меню. Мы должны выключить эту особенность в параметрах настройки **Импорта** для каждого актива.

Начните, выбирая файл структуры **MainTitle** в папке **Меню** в **Project** panel. Вы будете теперь видеть параметры настройки **Импорта** для этой

структуры в **Inspector,** так просто отсеивать checkbox для**, Производят Карты Mip.** Это экономит работу, потому что Unity больше не будет производить меньшие версии во времени выполнения.

Повторите этот шаг для других структур в папке **Меню**:

- **InstructionsBtn**
- **InstructionsBtnOver**
- **PlayBtn**
- **PlayBtnOver**
- **QuitBtn**
- **QuitBtnOver**

Добавление titling

Затем мы нуждаемся в эмблеме для нашей игры. Самый легкий способ добавить это со структурой, которую Вы проектировали, который настроен в Unity как Texture GUI.

GUI форматы Texture

В папке **Меню Project** panel, выберите структуру по имени **MainTitle.** Эта структура, разработанная в Фотомагазине, спасена как **РАЗМОЛВКА (Теговый Формат файла Изображения).** Это - хороший формат, чтобы использовать для любых структур, которые Вы намереваетесь использовать как Texture GUI. Это обеспечивает высококачественную несжатую прозрачность и может избежать некоторых проблем с белыми схемами, которые Вы можете видеть, используя другие форматы, такие как **ДЖИФ (Графический Формат Обмена)** или **PNG (Портативная Графика Сети).**

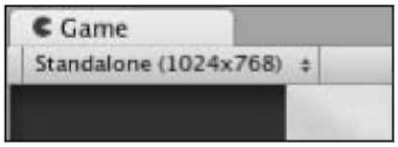
Создание объекта

С этой отобранной структурой, создайте новый Texture GUI на основе объект, идя в **GameObject | Create Other | GUI Texture.** Это автоматически читает измерения отобранной структуры и заставляет ее как структуру использовать в новом объекте, как обсуждено ранее. Вы теперь найдете объект по имени **MainTitle** в **Hierarchy,** поскольку объект взял бы название от файла, Вы выбрали во время его создания.

Расположение

Поскольку большинство компьютеров в настоящее время может обращаться с решениями в или выше 1024x768 пикселы, мы выберем это как стандарт, чтобы проверить наше меню и гарантировать, что оно работает в других решениях при использовании Classes `Экрана`, чтобы установить динамические положения. В верхне оставленном из **Game** view Вы будете видеть, опускаться меню (обратитесь к следующему скриншоту). Это меню позволит Вам определять различные отношения экрана или решения. От этого опускаются меню, выбирают **Автономный (1024x768)**, чтобы переключиться на

предварительный просмотр того размера.



Примите во внимание что, если решение Вашего собственного компьютера будет бежать близко к этому размеру, то **Game** view не будет показывать точное представление этого, поскольку это может составить меньшую часть интерфейса Unity.

К пуговице **Game** view (и любая из групп интерфейса) в fullscreen способ, Вы можете толпиться по этому с мышью и выявить *Клавишу "пробел"*. Поскольку мы помещаем элементы GUI в эту главу, используем этот метод для предварительного просмотра, на что интерфейс будет похож в законченной игре.

По умолчанию, все Структуры GUI начинаются со своего положения в (0.5, 0.5, 0), который данный, что 2-ые элементы работают в координатах экрана от 0 до 1 - находится в середине экрана.

В компоненте **Transform (преобразовать)** для **MainTitle** возражают в **Hierarchy,** набор положение к **(0.5, 0.8, 0).** Это помещает эмблему в верхний центр экрана. Теперь, когда мы добавили главную эмблему названия игры, мы должны будем добавить три кнопки, использующие далее GUI объекты Texture.

Создание подхода меню 1

В этом первом подходе мы создадим меню, которое использует прозрачную второстепенную структуру как Texture GUI, таким же образом поскольку мы только что сделали с нашей главной эмблемой названия.

Мы тогда должны будем написать script, чтобы заставить структуру получить события мыши для мыши, вступают, выход мыши, и мышь вниз/.

Добавление кнопки игры

В папке **Меню** в **Project** panel, выберите структуру по имени **PlayBtn.** Пойдите в **GameObject | Create Other | GUI Texture.** Выберите объект **PlayBtn** в **Hierarchy**, который Вы только что сделали. В **Inspector,** набор его **Transform** position к **(0.5, 0.6, 0).**

Кнопка Texture GUI script

Это является первым из наших трех кнопок, и потому что у них всех есть общие functions, мы теперь напишем script, который может использоваться на всех этих трех кнопках, используя общественные

variables участника, чтобы приспособить параметры настройки. Например, каждая кнопка будет:

- Играйте звук когда нажато
- Загрузите различный уровень (или **Сцена** в сроках Unity) когда нажато
- Структура обмена, когда мышь по ним, чтобы выдвинуть на первый план их

Выберите папку **Scripts** в **Project** panel, и от **Create** кнопка опускается меню, выбирает **JavaScript.** Переименуйте **NewBehaviorScript** к **MainMenuBtns,** и затем щелкните два раза его изображением, чтобы начать это в редакторе script.

Начните, устанавливая четыре общественных variables участника наверху script, как показано в следующем кодовом отрывке:

```
var levelToLoad : String; var
normalTexture  : Texture2D; var
rollOverTexture  : Texture2D; var
beep  : AudioClip;
```

Первым Variables, который мы будем использовать, чтобы сохранить название уровня, который будет загружен, когда кнопка этот script будет применен к, щелкают. Помещая эту информацию в Variables, мы в состоянии применить этот script ко всем трем кнопкам, и просто использовать имя переменной, чтобы загрузить уровень.

Вторые и третьи variables объявлены как variables типа `Texture2D`, но не установлены в определенный актив так, чтобы структуры могли быть назначены, используя, drag and drop (перетащили и опустили) в **Inspector.**

Наконец, Variables типа `AudioClip` установлен, который будет играться, когда мышью щелкнут.

Теперь добавьте следующий Functions, чтобы установить структуру, используемую компонентом Texture GUI, когда мышь входит в область, которую занимает структура. Это обычно упоминается как государство одновременного нажатия клавиш или парение:

```
function OnMouseEnter(){
 guiTexture.texture = rollOverTexture;
}
```

В пакете `Меню`, который Вы импортировали, Вам предоставляют нормальное и по государству для каждой структуры кнопки. Это сначала `OnMouseEnter ()` Functions просто устанавливает область структуры, привыкшую компонентом к тому, что было назначено на общественный Variables участника `rollOverTexture` в **Inspector.** Чтобы знать, когда мышь переезжает или выходит из границы этой структуры, добавьте следующий Functions:

```
function OnMouseExit(){
  guiTexture.texture = normalTexture;
}
```

Если бы это не присутствовало, то `rollOverTexture` просто остался бы на появлении игроку, как будто тот определенный выбор меню был все еще выдвинут на первый план.

Теперь, чтобы обращаться со звуком и погрузкой соответствующей сцены, добавьте следующий Functions:

```
function OnMouseUp(){
  audio.PlayOneShot(beep);
  yield new WaitForSeconds(0.3 5);
Application.LoadLevel(levelToLoad);
```

Мы играем звук как первую команду, чтобы гарантировать, что это играет прежде, чем следующая сцена загружена, и что это не отключено. Помещая `урожай` командуют промежуточный script для того, чтобы играть звук и загрузить следующую сцену, мы в состоянии остановить script для определенного числа секунд. Используя урожаи в scripting быстрый способ создать задержку, не имея необходимость писать таймер.

Здесь мы просто создаем задержку с урожаем, но они могут также использоваться, чтобы выполнить весь набор инструкций перед выполнением следующей линии кодекса. Это известно как **coroutine,** поскольку **рутина** - общий термин для инструкций, выполняемых в программировании сроков.

После урожая наша игра загрузит сцену, используя `Заявление. LoadLevel ()` команда, беря любой текст мы пишем в `levelToLoad` общественный Variables участника в **Inspector** и использовании что найти соответствующий файл сцены.

Создавая интерфейсы, обычно советуют поместить действия в мышь случай, а не мышь вниз случай. Это дает игроку шанс отодвинуть их курсор, если они выбрали неправильный пункт.

Наконец, поскольку мы играем звук, гарантируем, что у Вашего объекта есть компонент `AudioSource`, добавляя следующую линию `RequireComponent` к основанию Вашего script:

```
@script RequireComponent (AudioSource)
```

Пойдите в **Файл |, Экономят** в редакторе script и возвращаются к Unity теперь. Выберите объект **PlayBtn** в **Hierarchy,** и пойдите в **Компонент | Scripts | Главное Меню Btns**, чтобы применить script, который Вы только что написали. Это должно тогда появиться в списке **Inspector** компонентов для объекта **PlayBtn**. В результате линии `RequireComponent` у Вас также будет **Звуковой Исходный** компонент на Вашем объекте.

Назначение общественных variables участника

Поскольку Вы будете видеть, общественные variables участника должны будут быть назначены прежде, чем этот script может функционировать. В **Уровне, Чтобы Загрузить** Variables, напечатайте на название **Island Level**, чтобы гарантировать, что это загружено, когда на кнопку нажимают. Тогда тяните **PlayBtn** и структуры **PlayBtnOver** от папки **Меню** в **Project** panel к **Нормальному Texture** и **Переверните** variables **Texture** соответственно. Наконец, тяните **menu_beep** звуковую скрепку от той же самой папки до Variables **Звукового сигнала**. Когда закончено, Ваш компонент должен быть похожим на это:

Тестирование кнопки

Теперь нажмите кнопку **Play**, чтобы проверить сцену. Когда Вы перемещаете свой курсор мыши через кнопку **Play Game**, он должен обменять структуру к структуре **PlayBtnOver**, которая покрашена и имеет мотив пламени направо. Отодвигая курсор от структуры, это должно переключиться назад.

Теперь попытайтесь щелкнуть кнопкой. У основания экрана Вы будете видеть ошибку в пульте, который является баром у основания высказывания экрана:

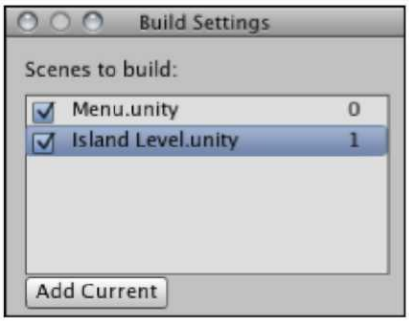
Island Level уровня не мог быть загружен, потому что он не добавлен к построить параметрам настройки.

Это - способ Unity гарантировать, что Вы не забываете добавлять все включенные уровни к **Построить Параметрам настройки.** Постройте параметры настройки, эффективно экспортные параметры настройки Вашей игры (в сроках развития игры, законченном, или проверьте продукт, упоминается как то, чтобы строить). Построить параметры настройки в Unity должны перечислить все сцены, включенные в Вашу игру. Чтобы исправить это, удостоверьтесь, что Вы нажимаете **Игру**, чтобы прекратить проверять игру, и затем пойти в **Файл | Строят Параметры настройки.**

С **Построить** открытой группой **Параметров настройки**, ниже **Сцен, чтобы построить** секцию, нажимают на кнопку **Add Current**, чтобы добавить сцену Меню, мы продолжаем работать. У сцен в построить списке параметров настройки есть заказ, представленный числом направо от названия сцены, и Вы должны удостовериться,

что первая сцена, которую Вы должны загрузить, всегда находится в положении **0.**

Drag and drop (перетащили и опустили) **Island Level** от **Project** panel, и понизьте это на текущий список сцен так, чтобы это было перечислено ниже **Menu.unity,** как показано в следующем скриншоте:



Знайте, что нет никакого подтверждения или экономить кнопку параметров настройки на **Построить** диалоге **Параметров настройки**, так просто близко это и затем повторно проверять игру. Нажмите **Игру** в Unity и затем попытайтесь нажать на Вашу кнопку **Play Game - Island Level** должен загрузить после того, как **menu_beep** звуковой эффект играется. Теперь нажмите **Игру** снова, чтобы прекратить проверять, и Вы возвратитесь в сцену **Меню**.

Добавление кнопки инструкций

Чтобы добавить вторую кнопку в нашем меню, просто выберите структуру **InstructionsBtn** в папке **Меню** в **Project** panel, и пойдите в **GameObject | Create Other | GUI Texture.** Это создает объект также по имени **InstructionsBtn** в **Hierarchy.** В компоненте **Transform (преобразовать)** этого объекта, устанавливает ценности **Положения** в **(0.5, 0.5, 0).**

Поскольку scripting уже сделан, просто пойдите в **Компонент | Scripts | Главное Меню Btns**, чтобы добавить script к этой кнопке, и затем назначить соответствующие структуры и **menu_beep** в той же самой манере, поскольку мы сделали в *Назначающей общественной* секции *variables участника* ранее. Поскольку мы еще не сделали сцену инструкций в Unity, просто заполнитесь на название **Инструкции** в `levelToLoad` Variables, и мы гарантируем, что нашу сцену называют этим позже.

Добавление оставленной кнопки

Эта кнопка работает в подобной манере к первым двум, но не загружает сцену. Вместо этого это призывает **Прикладной** Classes build, используя **Оставленный ()** команда заканчивать игру как заявление так, чтобы

Ваша операционная система закрыла это.

Это означает, что мы должны будем изменить наш script `MainMenuBtns`, чтобы составлять это изменение. Щелкните два раза этим script в папке **Scripts Project** panel, чтобы начать это в редакторе script.

Начните, добавляя следующий булевый общественный Variables участника к вершине script:

```
вар QuitButton: булевый = ложный;
```

Это, которое мы будем использовать как пуговица - если это будет установлено в `истинный`, тогда это заставит щелчок кнопки - `OnMouseUp ()` Functions - управлять `оставленным ()` команда. Если это будет `ложно` (то есть, его государство по умолчанию), то это загрузит уровень, относился к `levelToLoad` Variables как нормальному.

Чтобы осуществить это, `еще` реструктурируйте свой `OnMouseUp ()` Functions с `если` утверждение, как показано в следующем кодовом отрывке:

```
function OnMouseUp(){
 audio.PlayOneShot(beep);
 yield new WaitForSeconds(0.3 5);
 if(QuitButton){
  Application.Quit();
 }
 else{
  Application.LoadLevel(levelToLoad);
 }
}
```

Здесь мы просто изменили Functions, чтобы играть звук и паузу `(урожай)`, независимо от того, какой застегивают, это. Однако, мы должны выбрать между двумя вариантами - если `QuitButton верен`, то `Заявление. Оставленный ()` команду называют, иначе `(еще)`, уровень загружен как нормальный.

Пойдите в **Файл |, Экономят** в редакторе script, и переключаются назад на Unity.

Выберите структуру **QuitBtn** в папке **Меню** в **Project** panel, и пойдите в **GameObject | Create Other | GUI Texture.** Это создает объект по имени **QuitBtn** в **Hierarchy.** В компоненте **Transform (преобразовать)** для этого объекта, устанавливает ценности **Положения** в **(0.5, 0.4, 0).**
С **QuitBtn**, все еще отобранным в **Hierarchy,** пойдите в **Компонент | Scripts | Главное Меню Btns**, чтобы добавить script. В **Inspector,** заполнитесь в общественных variables участника как прежде, но на сей

раз оставьте **Уровень, Чтобы Загрузить** бланк, и выбрать коробку рядом с недавно добавленным **Оставленным** Variables **Кнопки**.

Чтобы перепроверить Ваш script, здесь это полностью:

```
var levelToLoad : String; var
normalTexture  : Texture2D; var
rollOverTexture  : Texture2D; var
beep  : AudioClip; var QuitButton
: boolean = false; function
OnMouseEnter(){
 guiTexture.texture = rollOverTexture;
}

function OnMouseExit(){
 guiTexture.texture = normalTexture;
}

function OnMouseUp(){
 audio.PlayOneShot(beep); yield
 new WaitForSeconds(0.35);
 if(QuitButton){
  Application.Quit();
 }
 else{
  Application.LoadLevel(levelToLoad);
 }
}

@script RequireComponent(AudioSource)
```

Теперь пойдите в **Файл |, Экономят Сцену**, чтобы обновить проект, и затем нажать **Игру**, чтобы проверить меню. Нажим кнопки меню **Play Game** должен загрузить Island Level. Кнопка **Instructions** вызовет 'уровень, не мог быть загружен' ошибка, которую мы видели ранее, поскольку мы еще не создали это. Кнопка **Quit Game** не будет вызывать ошибку, но не будет также анонсировать в редакторе Unity, таким образом мы не будем в состоянии проверить это, пока мы не создадим строить позже.

Не забудьте нажимать **Игру** снова, чтобы закончить проверять. Поскольку первый подход при создании создания меню теперь полон, мы будем теперь смотреть на другой метод создания функционального меню в Unity, используя **OnGUI ()** Functions, упомянутый в Unity как часть GUI 2.0 системы, потому что это было начато с версией 2.0 Unity.

Используя отладку командует, чтобы проверить scripts

Несмотря на `Заявление. Оставленный ()` команда, не анонсирующая в Редакторе Unity, мы должны гарантировать, что кнопка **Quit** действительно работает, вместо того, чтобы принять это дело обстоит

так. Чтобы проверить любую часть script, Вы можете просто поместить в команде отладки. Это пошлет сообщение в часть пульта Unity, который анонсируется у основания интерфейса.

Давайте испытаем это теперь. Возвратитесь к своему script **MainMenuBtns** в редакторе script, и определите местонахождение части `QuitButton` кодекса:

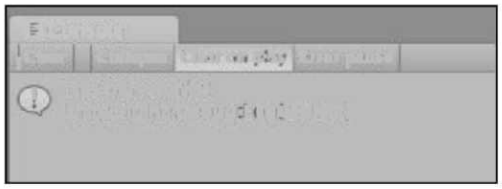
```
if(QuitButton){
 Application.Quit();
}
```

Считывание отладки может быть загружено список с другими его вида наряду с ошибками в пульте. Они обычно похожи на это:

```
Debug.Log("This part works!");
```

Сочиняя эту линию кодекса туда, где Вы ожидаете, что script выполнит, Вы можете обнаружить, работают ли специфические части Вашего script. В нашем примере мы поместили бы эту команду после `Заявления. Оставленный ()`, поскольку это доказало бы, что команда была выполнена без проблемы. Включите это так, чтобы это было похоже на следующий кодовый отрывок:

```
if(QuitButton){
 Application.Quit();
 Debug.Log("This part works!");
}
```



Спасите script, идя в **Файл |, Экономят** в редакторе script, и возвращаются к Unity. Теперь проверьте свою сцену меню снова, нажимая кнопку **Play**, и Вы будете видеть, что команда отладки печатает у основания интерфейса Unity. Если Вы откроете часть **Console** Unity *(Команда + Изменение + C* на Mac, *Ctrl+Shift + C* на PC), то Вы будете видеть, что это перечисляло там также, как показано в следующем скриншоте:

Эта техника может оказаться очень полезной, диагностируя проблемы script или проектируя части теоретического script, для которого у Вас нет команд, чтобы заполнить все же.

Создание подхода меню 2

Поскольку у нас уже есть рабочее меню, вместо того, чтобы удалить это из нашей сцены, мы временно повредим объекты, которые составляют

это. Делая это, Вы можете выбрать, какое меню Вы предпочитаете позже и восстанавливаете по мере необходимости.

Выведение из строя Объектов Игры

По одному, выберите **PlayBtn, InstructionsBtn,** и **QuitBtn** в **Hierarchy,** и дезактивируйте их, делая следующее:

- В **Inspector,** отсейте checkbox налево от названия объекта
- Гарантируйте, что это повернуло текст объекта к светло-серому в **Hierarchy** и что сам элемент исчез из предварительного просмотра **Game** view

Письмо OnGUI () script для простого меню

Теперь, создайте новый пустой объект, идя в **GameObject |, Создают Пустой.** Это делает новый объект в **Hierarchy** по имени **GameObject** с только компонентом **Transform (преобразовать)** приложенным. Это будет объектом держателя для нашего GUI 2.0 меню. Это - то, потому что script, который мы собираемся написать, должен будет быть приложен как компонент, чтобы функционировать. Имеет смысл посвящать объект этому ради организации.

Поскольку положение элементов OnGUI обработано через scripting, положение transform (преобразовать) этого объекта является несоответствующим, таким образом мы не должны будем регулировать это. Просто переименуйте объект от его названия по умолчанию до **Menu2** в обычной манере.

Выберите папку **Scripts** в **Project** panel и нажмите на кнопку **Create**, выбирая **JavaScript** как тип актива, чтобы создать. Переименуйте этот script к **MainMenuGUI2** и затем щелкните два раза его изображением, чтобы начать это в редакторе script.

Установленный против расположения

Для этого примера мы будем использовать Classes Расположения Unity GUI, кое-что, что более гибко для того, чтобы поместить в противоположность Classes GUI, у которого есть неподвижное расположение.

Расположение GUI одобрено многими разработчиками как это автоматически элементы положений, такие как формы и кнопки, ниже друг друга. Это позволяет разработчику проводить больше времени, разрабатывая их расположение с активом skin GUI, который когда относящийся `OnGUI ()` script, functions подобным способом к stylesheets для веб дизайна.

Общественные variables участника

Начните свой script, устанавливая четыре общественных variables

участника:

```
var beep  : AudioClip; var
menuSkin  : GUISkin; var
areaWidth  : float; var
areaHeight   : float;
```

Здесь мы создаем ту же самую скрепку аудио звукового сигнала как замечено в нашем первом подходе script, затем щель для skin GUI, который будет применен и два числовых variables, которые мы можем использовать, чтобы определить полный размер области нашего GUI.

OnGUI () Functions

Затем, установите следующий Functions в своем script:

```
function OnGUI(){
 GUI.skin = menuSkin;
}
```

Это устанавливает `OnGUI ()` Functions, и настраивает некоторые ключевые элементы. Сначала мы применяем актив skin, представленный `menuSkin` Variables. Это означает, что любыми элементами GUI, помещенными в этот Functions, такими как кнопки, формы, и так далее будет управлять стиль skin, относился к этому Variables. Это облегчает обменивать skins, и таким образом полностью повторно разрабатывать Ваш GUIs сразу.

Гибкое расположение для GUIs

Затем мы должны установить область для кнопок, в которых мы должны быть привлечены. У нас уже есть `areaWidth` и `areaHeight` variables, ждущие, чтобы использоваться, но мы должны удостовериться, что область, где мы тянем прямоугольное место для нашего GUI, собирается быть гибкой, в зависимости от того, в каком решении экрана игрой управляют.

Если бы мы не сделали этого и дали определенные измерения, то GUI выглядел бы различным на различных решениях и казался бы непрофессиональным. Чтобы противостоять этому, мы создадим некоторые частные variables в пределах нашего `OnGUI ()` Functions, который сохранит пункт центра на экране. Мы не должны использовать `частную` приставку, поскольку мы устанавливаем variables в Functions. Поэтому, они являются неотъемлемо частными.

После линии `GUI.skin` Вы только добавили, установить следующие два variables:

```
var ScreenX =    ((Screen.width * 0.5)    -    (areaWidth *
0.5)),-


var ScreenY =    ((Screen.height * 0.5)    -    (areaHeight *
0.5));
```

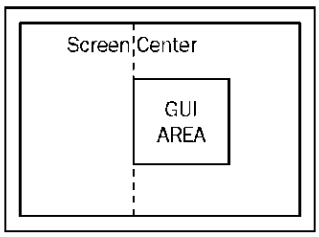
Здесь мы создаем два variables, которые равны сумме. У самой суммы есть две части:

```
((Screen.width * 0.5) - (areaWidth * 0.5));
```

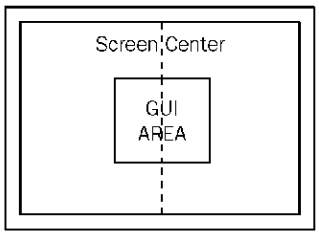
В вышеупомянутой линии, `(Screen.width * 0.5)` использует параметр `ширины` Classes `Экрана`, чтобы приобрести текущую ширину экрана игры. Мы делим это на два, чтобы найти пункт центра.

Отметьте, что здесь мы заменяем `"/2" * 0.5`. Это - то, потому что **я** умножение требую меньшего количества циклов центрального процессора чем подразделение - вокруг **я** 100 - находя ценности в Unity. **Я**

Мы тогда видим `(areaWidth * 0.5)`, который берет ширину нашей области GUI и находит пункт центра этого, делясь на два. Итак, почему мы вычитаем это из пункта центра экрана? Это - то, потому что области GUI всегда оттягиваются из их левого края, таким образом находя, что пункт центра экрана и рисунок оттуда привели бы к предоставлению вне центра, как показано к следующему изображению:



Вычитая половину ширины области GUI, мы достигнем центрального положения, как показано в следующем изображении:



Две части суммы помещены в пределах их собственных скобок так, чтобы их рассматривали как единственная сумма, которую Variables получает как ценность. Мы тогда повторяем этот процесс для второго Variables, `ScreenY`, чтобы получить вертикальное положение для нашей области GUI.

[223]

Области крайне важны для `GUILayout`-без них, `OnGUI ()` предполагает, что Вы желаете потянуть меню, используя все место экрана, начинаясь с верхнего левого. Устанавливая область с `BeginArea ()`, мы в состоянии определить четыре параметра:

```
GUILayout.BeginArea(Rect( distance from left of screen,
  distance from top of screen, width, height ));
```

`Rect ()` команда просто устанавливает прямоугольную область для `BeginArea ()` команда, чтобы использовать, таким образом при использовании частных variables `ScreenX` и `ScreenY`, мы будем в состоянии обеспечить положение для прямоугольной области, которая будет оттянута. Для `ширины` и `высоты`, мы будем использовать общественные variables участника, установленные наверху script.

Давайте добавим этот кодекс к нашему `OnGUI ()` Functions теперь ниже двух частных variables, которые Вы только установили:

```
GUILayout. BeginArea (Rect (ScreenX, ScreenY,
  areaWidth, areaHeight));
```

Область должна также быть закрыта с `EndArea ()` команда. Поскольку это закрывает нашу область, остальная часть нашего кодекса GUI должна быть помещена перед этой линией в Functions. Добавьте следующую линию, чтобы закрыть область GUI и затем спустить эту линию так, чтобы у Вас было место, чтобы написать в кодексе перед этим:

```
GUILayout. EndArea ();
```

Добавление кнопок UnityGUI

Перед `EndArea ()` линия, добавьте следующие линии, чтобы установить первую кнопку:

```
if(GUILayout.Button ("Play")){
  OpenLevel("Island Level");
}
```

Это устанавливает новый `GUILayout. Кнопка` со словом **Игра** на этом. Помещая это в, `если` утверждение, мы не только создаем это, но говорим Unity, что сделать, когда кнопка нажата. Инструкция, которую

мы даем этому, состоит в том, чтобы вызвать таможенную функцию по имени OpenLevel () с единственным параметром - наше название уровня. Мы напишем OpenLevel () Functions после окончания OnGUI () Functions.

Затем, добавьте следующие два если утверждения, чтобы создать другие две кнопки:

```
if(GUILayout.Button
  ("Instructions")){
  OpenLevel("Instructions");
}
if(GUILayout.Button ("Quit")){
  Application.Quit();
```

Со вторым, если кнопка утверждения мы называем тот же самый таможенный OpenLevel () Functions, но на сей раз мы посылаем различную последовательность в ее единственный параметр - название все еще, чтобы быть созданным уровнем Инструкций.

Третье, если кнопка утверждения не загружает уровень, но просто называет Заявление. Оставленный () команда вместо этого, как замечено в нашем подходе 1 GUI.

Вводные сцены с таможенными functions

Теперь, мы должны написать таможенную функцию, которая может быть вызвана, чтобы загрузить указанный уровень. Ниже заключительной правильной вьющейся скобы OnGUI () Functions, установите Functions следующим образом:

```
function OpenLevel(level    : String){
}
```

Здесь мы создаем Functions с параметром, названным уровнем, которому дают тип данных Последовательности - подразумевение, что, пока мы передаем последовательность текста к этому, вызывая функцию, мы можем использовать уровень слова, чтобы представить независимо от того, что текст передают к этому. В OnGUI () Functions, мы только добавили это требование:

```
OpenLevel (Island Level);
```

В этом примере слова "Island Level" передают к параметру уровня OpenLevel () Functions. Отметьте, что мы не должны говорить уровень = "Island Level", поскольку это автоматически знает, чтобы применить этот текст к параметру, это находит в OpenLevel () Functions. Если бы мы не включали правильный тип данных здесь, например передавая число или имя переменной, то мы получили бы ошибку, поскольку это не будет соответствующим для нашего параметра

уровня.

В результате использования этого параметра, чтобы послать последовательность текста к, везде, где параметр уровня используется, кодекс будет читать в последовательности, посланной в него.

Теперь в пределах этого Functions, поместите следующие три команды:

```
audio.PlayOneShot(beep);
yield new WaitForSeconds(0.3 5);
Application.LoadLevel(level);
```

Мы использовали эти команды в подходе 1. Обратитесь к ним, если Вы нуждаетесь к, но основное отличие, чтобы отметить вот наше использование параметра уровня, чтобы передать последовательность в Заявление. LoadLevel (). Чтобы закончить script, гарантируйте, что наш звуковой файл будет играть, добавляя обычную линию RequireComponent у основания script:

```
@script RequireComponent (AudioSource)
```

Пойдите в **Файл |, Экономят** в редакторе script теперь, и переключаются назад на Unity. Чтобы перепроверить Ваш script, здесь это полностью:

```
var beep  : AudioClip; var
menuSkin  : GUISkin; var
areaWidth  : float; var
areaHeight   : float;
function OnGUI(){
 GUI.skin = menuSkin;
 var ScreenX =  ((Screen.width * 0.5)   -   (areaWidth *
 0.5)); var ScreenY =   ((Screen.height * 0.5)   -
 (areaHeight * 0.5)); GUILayout.BeginArea  (Rect
 (ScreenX,ScreenY, areaWidth, areaHeight));
 if(GUILayout.Button ("Play")){
  OpenLevel("Island Level");
 }
 if(GUILayout.Button ("Instructions")){
  OpenLevel("Instructions");
 }
 if(GUILayout.Button ("Quit")){
  Application.Quit();
 }
 GUILayout.EndArea();
}
function OpenLevel(level   :
```

```
    String){ audio.PlayOneShot(beep);
    yield new WaitForSeconds(0.35);
    Application.LoadLevel(level);
  }

  @script RequireComponent(AudioSource)
```

Применение и моделирование

Назад в Unity, выберите свой пустой объект игры **Menu2** в группе **Hierarchy**. Пойдите в **Компонент | Scripts | Главное Меню GUI2**, чтобы добавить script к Вашему объекту.

Тяните **menu_beep** звуковую скрепку от папки **Меню** в **Project** panel к Variables участника общественности **Звукового сигнала** в этом script, чтобы назначить это. Теперь заполнитесь в **Высоте Ширины** и **Области Области** ценностью **200.**

Нажмите кнопку **Play**, чтобы рассмотреть меню. Поскольку Classes GUI собран от script, он только отдаст, когда игра будет проверена. В настоящее время меню мы создали через взгляды script, немного унылые, как показано в следующем скриншоте:

Таким образом мы должны будем применить стиль к этому, чтобы заставить это выглядеть немного более опрятным, который является, где **skins GUI** входит.

У `menuSkin` Variables должен быть актив skin GUI, назначающий на это, таким образом мы должны создать тот теперь. Выберите папку `Меню` в **Project** panel, и затем нажмите на кнопку **Create** наверху **Project** panel. От опускаться меню, выберите **Кожу GUI.** Это заставляет новый актив под названием **Новый GUISkin,** просто переименовать это к **MainMenu.**

GUI параметры настройки skin

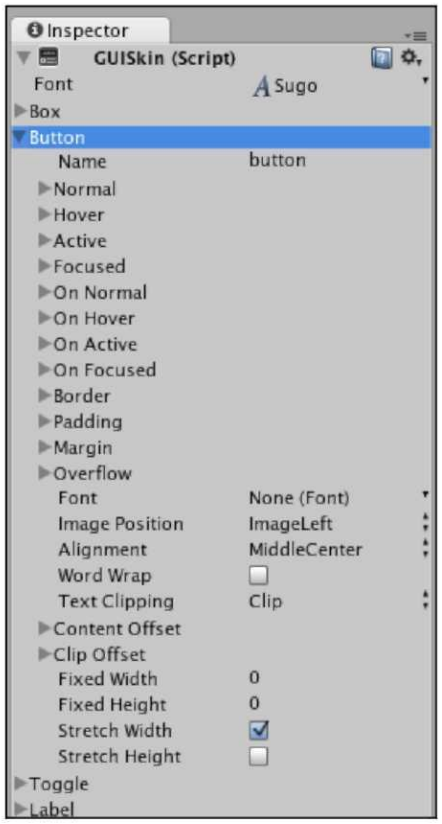
GUI параметры настройки предложения skins для каждого элемента в Classes GUI:

- Коробки
- Кнопки
- Пуговицы
- Лейблы
- Текстовые области и области
- Ползунки, scrollbars, и scrollviews

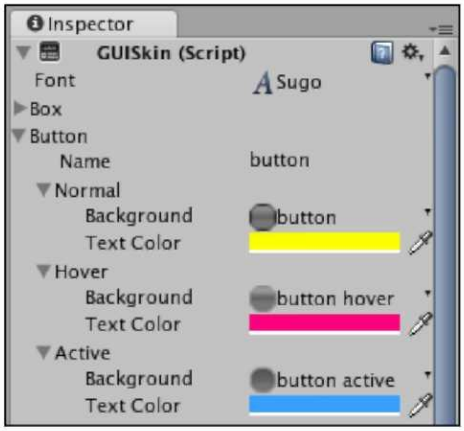
Первый "**шрифт** параметра" универсален ко всем элементам, которыми управляет skin. Так, начните, беря шрифт, который Вы вводили игре (который был **Sugo,** если Вы следовали за тем, что я сделал), и понизьте это от **Project** panel на этот параметр.

Мы будем использовать этот skin, чтобы разработать элементы кнопки.

Гарантируйте, что у Вас есть секция **Кнопки GUISkin**, расширенного в **Inspector**, нажимая на серую стрелку налево от этого так, чтобы Вы могли видеть ее параметры настройки, как показано в следующем скриншоте:



Расширьтесь **Нормальный, Парение,** и **Активный** так, чтобы Вы могли видеть **Цвет Фона** и **Текста** каждого, как показано в следующем скриншоте. **Второстепенный** параметр устанавливает структуру для фона кнопки по умолчанию, Unity предоставляет профессионально выглядящему округл-краю, графическому выдвинутые на первый план государства для **Парения** и **Активный.** Мы будем придерживаться их для этой книги. Однако, создавая меню для Ваших собственных игр, это - определенно кое-что, с чем Вы должны экспериментировать. Пока, нажмите на каждый **Текстовый** блок **Цвета**, и используйте цветного сборщика, который, кажется, выбирает цвет для **Нормального, Парения,** и **Активный** (придавленная мышь) государства кнопки:

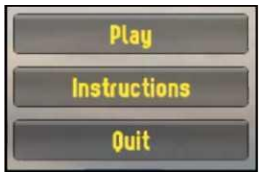


Затем, расширьте параметры настройки для **Дополнения** и установите **Вершину** и **Минимальные значения** оба к **6.** Это даст нам больше комнаты в кнопке выше и ниже текста непосредственно, точно так же, как дополнение делает в CSS.

Чтобы держать наши кнопки располагаемыми вертикально далее обособленно, мы должны будем увеличить нижнее поле. Расширьте параметр **Края**, и установите **Минимальное значение** в **10.**

Мы теперь сделали регуляторы skin и готовы применить его к нашему script GUI. Выберите объект **Menu2** в группе **Hierarchy**, и в **главном** компоненте **Меню GUI2** script, назначьте Variables участника общественности **Меню Кожи** при перемещении и понижая skin **MainMenu, который** Вы только что сделали от **Project** panel до щели Variables.
[229]

Теперь тест, который Ваш skin применен, нажимая кнопку **Play**, чтобы видеть, что script GUI отдает. Это должно быть похожим на это:



Это выглядит более профессиональным. Поскольку мы использовали

главный шрифт игры, это соединяется лучше с эмблемой игры. Законченный fullscreen должен теперь быть похожим на это:

Нажмите кнопку **Play** снова, чтобы прекратить проверять игру, и пойти в **Файл | Экономят сцену** в Unity.

Время решения

Теперь это - Ваша очередь взять на себя некоторое творческое управление! Выберите подход, взгляд которого Вы предпочитаете. Основанный на том, что Вы изучили в секции *объектов игры Выведения из строя* ранее, или держите второй подход и оставляют трех инвалидов объектов игры Texture GUI, или повреждают **Menu2** и восстанавливают их использующий checkboxes наверху **Inspector.**

Однако, рекомендуется, чтобы Вы продолжили работать со вторым подходом, используя UnityGUI, поскольку у этого есть много использования в дополнение к простому обеспечивающему меню представлению статистики во время тестирования, например, или построение параметров настройки для игрока, чтобы приспособиться. Эти более продвинутые темы - кое-что, с чем Вы, вероятно, столкнетесь, когда Вы будете прогрессировать из этой книги.

Резюме

В этой главе мы смотрели на два основных способа создать элементы интерфейса в Unity-GUI scripting и Структурах GUI. К настоящему времени у Вас должно быть хорошее понимание того, как осуществить любой подход, чтобы построить интерфейсы. В то время как есть еще

много вещей, Вы можете сделать с GUI scripting, методами для того, чтобы основать элементы, которые мы покрыли, вот основы, в которых Вы будете нуждаться каждый раз, когда Вы пишете script GUI.

Мы должны все еще произвести сцену **Инструкций**, содержащую информацию для игрока, но не волнуемся, мы будем создавать это, поскольку мы смотрим на небольшое количество новых методов для мультипликации в следующей главе, среди других последних штрихов для игры непосредственно.

Последние штрихи

В этой главе мы возьмем нашу игру от простого примера кое до чего, что мы можем развернуть, добавляя некоторые последние штрихи к острову. Поскольку мы смотрели на различные новые навыки всюду по этой книге, мы добавили отдельный пример за один раз. В этой главе мы укрепим некоторые из навыков, которые мы изучили пока, и также смотрим более подробно на некоторые заключительные эффекты, что мы можем добавить, что не крайне важны для, gameplay-который то, почему лучше оставлять их до конца цикла развития.

Строя любую игру, механика крайне важна - физические рабочие элементы игры должны быть в месте перед дополнительными художественными работами, и экологический талант может быть введен. Как в большинстве случаев, строя игру, крайние сроки будут установлены или, как часть независимой дисциплины разработчика, или издателем, для которого Вы работаете. Оставляя последние штрихи в конце цикла развития, Вы гарантируете, что Вы не потеряли времени, воздействуя на получение самого важного element-gameplay-just права.

В целях книги мы предположим, что наша механика игры является полной и рабочей как ожидалось. Теперь, повернитесь к тому, что мы можем добавить к нашей окружающей среде острова и игре вообще, чтобы добавить некоторый заканчивающийся талант.

Для этого мы будем добавлять следующий к нашей игре в этой главе:

- Система частицы в части вулкана ландшафта

- Основанный на близости звук грохота вулкана
- Полоски света для нашей кокосовой застенчивой миниигры, чтобы показать кокосовую траекторию
- Туман, чтобы добавить реализм к лучу обзора
- Оживляемая сцена **Инструкций**, чтобы объяснить цель игры игроку
- Переход постепенного появления изображения для **Уровня Острова**, используя Структуру GUI и Альфу scripting
- Сообщение победы игры игроку

Мы закончим, строя и смотря на различные проблемы для того, чтобы проверить Вашу игру, так же как возможности взятия Вашей игры на рынок через независимые каналы разработчика.

Во-первых, давайте начнем с нашими последними штрихами, делая наш более динамичный вулкан.

Вулкан!

Для этого следующего шага, гарантируйте, что у Вас есть сцена **Уровня Острова**, открытая в Unity. Если Вы не имеете, то открываете это теперь или щелкая два раза на файле **Сцены** в **Проектной** группе, или идя в **Файл | Открытая Сцена** и затем выбирая это из папки **Активов**. Файлы сцены легки определить, поскольку они используют то же самое изображение как редактор Unity непосредственно - эмблема Unity.

В Главе 2 мы построили ландшафт острова с редактором ландшафта, включая угол острова, посвященного рту вулкана. Чтобы заставить этот вулкан казаться немного более реалистичным, мы добавим перо дыма и источника аудио моноскрепки, чтобы создать основанный на близости звук вулкана, пузырящегося с литой лавой. Добавляя и звуковой и визуальный элемент, мы, мы надеемся, достигнем более динамического и реалистического чувства к острову и поддержим погружение игрока в нашей игре.

Начните, создавая новую систему частицы в Unity, идя в **GameObject |, Создают Другой | Система Частицы.** Это создает новую систему частицы под названием **Система Частицы** в **Иерархии.** Гарантируйте, что это отобрано теперь, и переименовывать это **Частицы Вулкана.**

Расположение системы частицы

Поскольку наш вулкан - просто часть объекта ландшафта непосредственно и не независимого объекта, расположение наших частиц в относительных условиях не возможно в обычной манере. Обычно, к относительно помещают объект, мы сделали бы новый объект ребенком объекта, которым мы желаем, чтобы это было рядом и перезагрузило свою местную позицию к (0, 0, 0).

В этом случае однако, мы должны будем использовать в своих интересах групповую штуковину взгляда **Сцены**. Начните, нажимая на Ось Y (зеленая ручка) штуковины, чтобы измениться от перспективного вида до нисходящего или вида с высоты птичьего полета острова. Если сделано правильно, штуковина тогда показывает слово **Вершина** ниже этого:

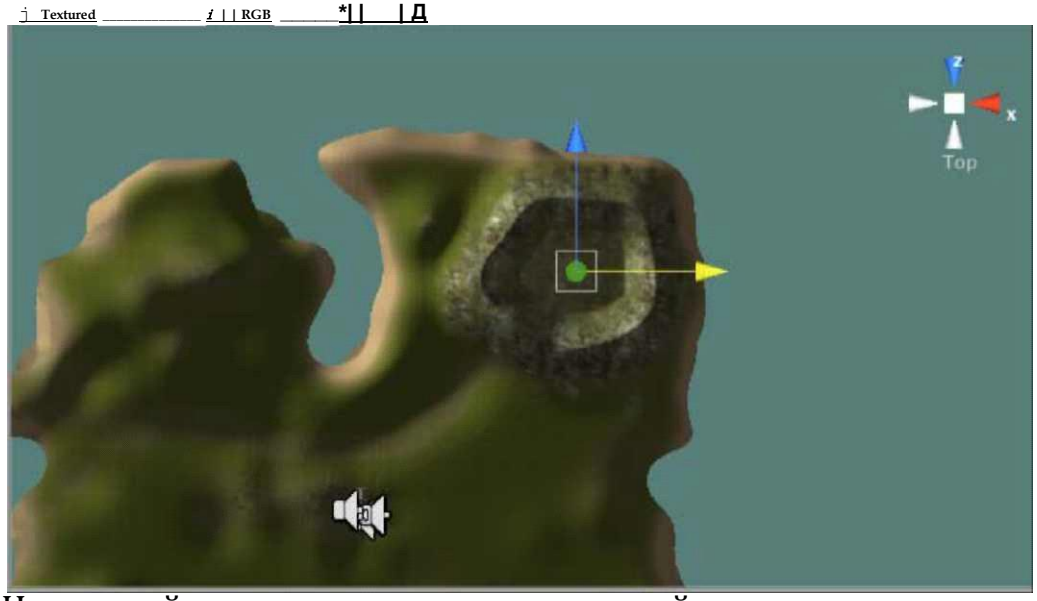
Затем, чтобы видеть, где Ваша система частицы расположена, гарантируйте, что она отобрана в группе **Иерархии**, и затем выбирать инструмент **Transform (преобразовать)** (горячая клавиша: *W)*, чтобы видеть его топоры в Сцене.

Поскольку новая система частицы будет создана в центре Вашего текущего вида **Сцены**, мы должны будем повторно поместить это в вулкане, используя этот **Вид сверху**. Удостоверьтесь, что Вы можете видеть и топоры системы частицы и вулкан непосредственно - Вы можете просто должны быть изменить масштаб изображения, чтобы видеть оба на экране. Чтобы сделать это, переключитесь на инструмент вида (горячая клавиша: Q), и проведение *командной клавиши* (Mac) или *клавиша CTRL* (PC), тянут мышь налево, чтобы изменить масштаб изображения, затем переключаются назад на инструмент Transform (преобразовать) *(W)*, чтобы видеть ручки оси Вашего объекта снова.

Теперь использование инструмент **Transform (преобразовать)**, тяните **X** (красноты) и **Z** (синие) ручки оси независимо, пока Вы не поместили свою систему частицы в центр рта вулкана от этой перспективы, как показано в следующем скриншоте:

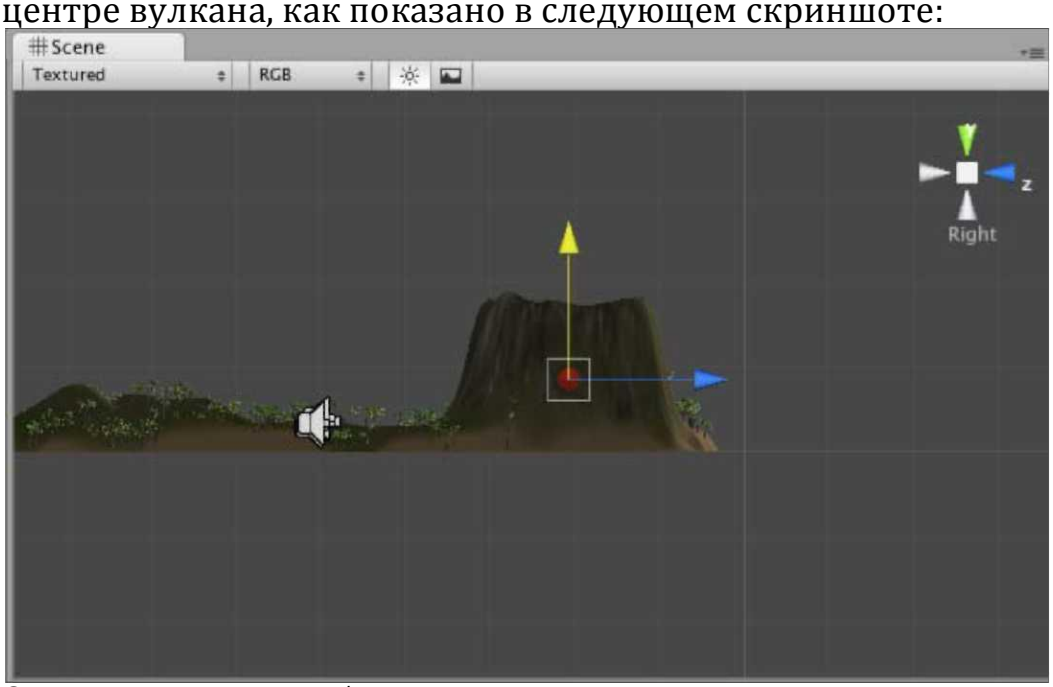
I ftsttnt .

j Textured _____ ɪ | | RGB _____ *| | | Д



Не смущайтесь выдвижением на первый план ручек в желтом, когда Вам выбрали их - у Вас все еще есть правильная ручка!

Теперь нажмите на красную ручку Оси X **Штуковины Сцены,** чтобы дать Вам сторону - ввиду острова. Теперь используйте этот вид, наряду с инструментом **Transform (преобразовать)**, чтобы тянуть ручку **Оси Y** (зеленую) из Вашей системы частицы, чтобы получить это к позиции в центре вулкана, как показано в следующем скриншоте:



Отметьте, что в изображении, зеленая ручка в настоящее время отбирается и таким образом выдвинута на первый план в желтом. Наконец, переключитесь назад на Перспективный вид, нажимая на **белый куб** в

центре **Штуковины Сцены.**

Загрузка активов

Затем, мы должны будем добавить некоторые активы к нашему проекту закончить вулкан. Активы доступны в кодовой связке, обеспеченной на packtpub.com <http://packtpub.com> (www.packtpub <http://www.packtpub>. com/files/code/8i8i_code.zip). Определите местонахождение пакета, названного volcanoPack. unitypackage от наших извлеченных файлов, и затем переключитесь назад на Unity.
Пойдите в **Активы | Пакет Импорта,** проведите к файлу, который Вы только загрузили на своем компьютере, и выбирать его как пакет, чтобы импортировать. Подтвердите это, когда Вам покажут список активов, включенных, и Вы найдете, что Вы добавили папку **Вулкана** к своему проекту. В этой папке Вы найдете:

- Структура для дыма вулкана
- Звуковой файл, чтобы представить грохочущую лаву вулкана
- Файл структуры вызвал белого, которого мы будем использовать позже для нашего постепенного появления изображения уровня

Создание материала дыма

Теперь, когда мы импортировали соответствующие активы, мы должны будем сделать материал для нашей структуры дыма вулкана. Чтобы оставить вещи опрятными, мы создадим эту внутреннюю часть папка **Вулкана**. Выберите папку **Вулкана** в **Проектной** группе, и затем нажмите на кнопку **Create**, выбирая **Материал** из опускаться меню.

Переименуйте этот новый материальный **Материал Дыма Вулкана,** и гарантируйте, что он отобран в **Проектной** группе, чтобы видеть ее параметры настройки в **Инспекторе.** От **Shader** опускаются меню, выбирают**, Частицы | Умножаются.** Это установит стиль предоставления для материала к одному подходящему для частиц - **Умножаются**, покажет прозрачный фон структур частицы и смягченные края. Drag and drop (перетащили и опустили) **volcano_smoke** файл структуры от папки **Вулкана** на пустую щель направо от урегулирования **Структуры Частицы**, оставляя параметры **Черепицы** и **Погашения** в их неплатежах.

Теперь тяните **Материал Дыма Вулкана** от папки **Вулкана** в **Проектной** группе, и понизьте это на систему частицы **Дыма Вулкана** в **Иерархии**, чтобы применить это.

Параметры настройки системы частицы

Как с любым визуальным эффектом, особенно относительно систем много частицы экспериментирования необходимо, чтобы достигнуть эффекта, что Вы лично чувствуете хорошие взгляды. С этим в памяти, я рекомендую

Вам просто использовать параметры настройки, которые я предлагаю здесь как гид, и затем не тороплюсь, чтобы попытаться приспособить несколько параметров настройки, чтобы достигнуть эффекта что:

- Вам нравится вид
- Работы хорошо со стилем рта вулкана Вы создали на своем ландшафте.

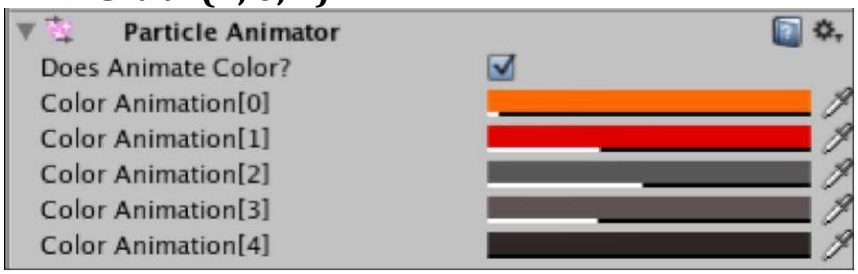
Отметьте, что параметры настройки, перечисленные здесь, являются только теми, которые были приспособлены от неплатежа.

Эллиптические параметры настройки Эмитента Частицы

- **Размер Минуты: 40**
- **Макс Сайз: 60**
- **Энергия Минуты: 10**
- **Макс Энерджи: 40**
- **Эмиссия Минуты: 2**
- **Макс Эмишен: 8**
- **Мировая Скоростная Ось Y: 30**

Параметры настройки Аниматора Частицы

- **Цветная Мультипликация [0]: Оранжевый цвет, 5%-ая Альфа**
- **Цветная Мультипликация [1]: Красный цвет, 25%-ая Альфа**
- **Цветная Мультипликация [2]: Середина Серого цвета, 40%-ой Альфы**
- **Цветная Мультипликация [3]: Более темный Серый цвет, 25%-ая Альфа**
- **Цветная Мультипликация [4]: Черный цвет, 5%-ая Альфа**
- **Размер Растет: 0.15**
- **Сила Rnd: (25, 0, 25)**
- **Сила: (1, 0, 1)**



Теперь не торопитесь, чтобы приспособить эти параметры настройки, чтобы заставить частицы удовлетворить Вашему ландшафту а

немного лучше.

Добавление аудио к вулкану

Чтобы закончить эффект подлинного вулкана, мы добавим звуковой источник теперь с нашей вулканической звуковой петлей, играющей на этом.

Как отмечено ранее, наш вулкан не фактический объект игры, таким образом в этом случае также, мы не можем добавить компонент к нему из-за этого факта. Однако, у нас действительно теперь есть объект в пункте центра нашего вулкана - система частицы. Это означает, что мы можем использовать тот объект как тот, чтобы добавить наш звуковой компонент к.

Гарантируйте, что объект **Volcano Smoke** отобран в **Иерархии** и затем идти в **Компонент | Аудио | Звуковой Источник.**

Это добавляет звуковой исходный компонент к основанию списка компонентов в **Инспекторе.** Поскольку у системы частицы уже есть несколько компонентов, заставляющих это работать, Вы, возможно, должны прокрутить вниз в **Инспекторе**, чтобы найти **Звуковой Источник.**

Назначьте **volcano_rumble** звуковую скрепку от папки **Вулкана** до **Звукового** параметра **Скрепки**, и гарантируйте, что **Игра На Активном** отобрана - это гарантирует, что звук не будет нуждаться в вызове, но просто играет, когда сцена будет загружена. Регулируйте **Громкость** и параметры **Макса Вольюма** оба к 200. Это гарантирует что:

- Звук вулкана достаточно громок, чтобы пересилить звук стерео окружения склона, относился к ландшафту
- Звук достигает этого уровня, когда игрок около звукового источника, то есть, стендов рядом с вулканом

Мы оставим **Объем Минуты** на **0**, поскольку это означает, что звук может полностью исчезнуть, если игрок достаточно далек от вулкана.

Затем, мы можем определить, как далеко далеко игрок должен быть для звука, чтобы постепенно исчезнуть, регулируя урегулирование **Фактора Rolloff**. На моем ландшафте урегулирование **0.025** является хорошим количеством - с этим урегулированием, чем выше ценность, тем ближе игрок должен быть источнику прежде, чем услышать это так при использовании низкой ценности, поскольку я имею, этот звук, несет для большого расстояния, поскольку Вы ожидали бы, что звук грохочущего вулкана сделает. Начните с этой ценности, и попробуйте различные параметры настройки, когда Вы играете тест игра коротко. Наконец, выберите коробку для **Петли**, чтобы гарантировать, что звук продолжает играть неопределенно

Тестирование вулкана

Теперь, когда наш вулкан полон, мы должны проверить его эффективность. Во-первых, пойдите в **Файл |, Экономят Сцену**, чтобы гарантировать, что мы не теряем неспасенного продвижения. Тогда нажмите кнопку **Play**, и попытайтесь идти от текущего местоположения Вашего игрока, к вулкану. Частицы должны повышаться в воздух. Поскольку Вы приближаетесь к вулкану, объем звука вулкана должен стать громче. Если это не достаточно громко, просто tweaked (щипнуть) ценности в областях **Объема**, или если звук не несет достаточно далеко, то понижает ценность для **Фактора Rolloff** в **Звуковом Исходном** компоненте.

> Помните, что сделанное использование любых изменений **Инспектора** во время тестирования игры будет уничтожено, как только Вы поражаете кнопку **Play** снова, чтобы прекратить проверять. Просто используйте это как буквальный период тестирования, и гарантируйте, что Вы помещаете ценности, на которые Вы обосновываетесь в **Инспекторе** снова, когда Вы прекратили проверять игру.

Это стоит отмечать, что для того, чтобы проверить цели, Вы можете желать увеличить скорость **Первого Диспетчера Человека**, который позволит Вам идти вокруг Вашего острова более быстро. Чтобы сделать это, в то время как Вы - тестирование игры, выбирает объект **First Person Controller** в **Иерархии.** Установите Variables участника общественности **Скорости FPSWalker (сценарий)**, компонент к Вашей желательной ценности - помнит, что это просто для того, чтобы проверить, таким образом нереалистичные скорости прекрасны! Поскольку Вы делаете это во время тестирования, это урегулирование **Иерархии** вернется назад, как только Вы нажимаете кнопку **Play** снова, чтобы остановить испытательное значение, Вы не будете терять свою оригинальную намеченную скорость.

Кокосовые следы

Затем мы добавим некоторый талант к нашей кокосовой застенчивой игре, добавляя полоски света к нашим кокосовым prefab. Делая это, когда игрок бросает их, они будут видеть, что полоска света следует за траекторией снаряда, который должен дать хороший визуальный эффект.

Редактирование Prefab

Чтобы осуществить это изменение, мы должны будем возвратиться к нашему кокосовому prefab от Главы 6, как **След** компонент **Renderer, который** мы будем использовать, должен быть присоединен к этому объекту. Откройте папку **Prefab** в **Проектной** группе, и определите местонахождение **Кокосового Заранее приготовленного** актива. Тяните это в сцену так, чтобы мы могли воздействовать на эти-активы, может работаться на непосредственно от их местоположения в **Проектной** группе, но чтобы анонсировать и проверить эффект, который мы создаем,

лучше тянуть их в сцену, и видеть то, что мы делаем "в действии". Помните, что, нажимая *F* с Вашим курсором по виду **Сцены**, Вы можете изменить масштаб изображения прямо к местоположению отобранного объекта.

Тащите компонент Renderer

Чтобы добавить компонент, гарантируйте, что **Кокосовый Prefab** все еще отобран в группе **Иерархии**, и идти в **Компонент | Частицы | След Renderer.** Вы будете побуждены, объясняя, что Вы теряете связь с prefab, просто продолжаете в этом пункте - мы обновим prefab, как только мы закончили делать наш след.

Этот компонент просто тянет образующую дугу векторную линию, готовя ряд очков от объекта, поскольку он перемещается через трехмерный мир. Определяя длину, материал, и начало/конец widths линии, мы будем в состоянии достигнуть эффекта, который мы хотим. Чтобы видеть установку по умолчанию следа renderer, нажмите кнопку **Play** теперь, и наблюдайте кокосовое падение к основанию, оставляя след позади этого. Вы должны видеть, что противное широкое черное пятно предоставлено - не хороший!

Во-первых мы обратимся к некоторым проблемам работы - для Unity Про пользователи версии, способность использовать динамические тени прибывает как стандарт; однако, поскольку мы не нуждаемся в линии, чтобы бросить или получить любые тени, отсейте первые два параметра на компоненте в **Инспекторе.** Тени вообще дороги, и поскольку сам след renderer добавляет к напряжению, надевает власть обработки компьютера игрока, что-нибудь, что мы можем сделать, чтобы уменьшить напряжение, определенно хорошая идея.

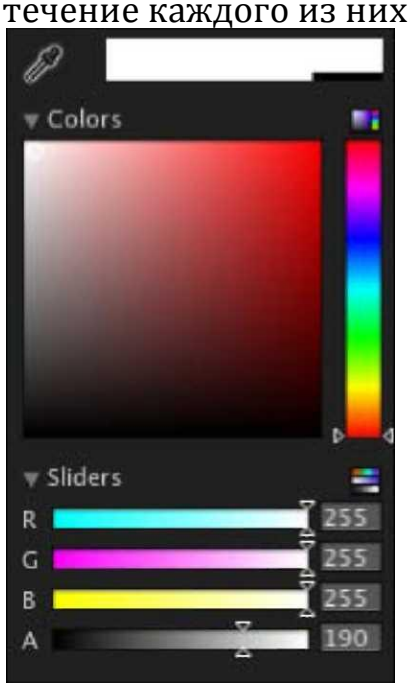
Затем расширьте параметр **Материалов**, чтобы видеть **Размер** и **Элемент 0** параметров настройки. Здесь мы можем поручить структуре использовать для следа. Поскольку мы уже сделали материал пламени, мы снова используем это, потому что это использует соответствующий тип shader для следа - (мягкая) Добавка. Откройте папку **Особенности Огня** в **Проектной** группе, и определите местонахождение материала **Пламени**, затем drag and drop (перетащили и опустили) это на **Элемент 0** урегулирования для **Следа** компонент **Renderer** в **Инспекторе.**

Теперь, гарантировать след не чрезмерно длинно, установите параметры **Time (Времени)** к ценности **1.** Это означает, что след - вторые длинные пункты в конце следа, удалены после того, как они существовали за это количество времени.

Теперь установите **Ширину Начала** в **0.25** и **Ширину Конца** к **0.15** - они определяют ширину предоставленного материала с обоих концов следа, и, вообще говоря, имеет смысл делать начало шире чем конец, чтобы сузиться след.

Наконец, расширьте параметр **Цветов** так, чтобы Вы могли видеть каждую коробку для цвета. С этим мы можем оживить появление следа через цвет и также видимость, используя альфа-параметры настройки. Поскольку у нас есть цвет в нашей структуре пламени, мы оставим цвета этих

параметров настройки, но просто заставим след исчезнуть к его концу. Нажмите на каждый **Цветной** блок в свою очередь. При использовании **(Альфа**) ценность, ценность набора от 80 процентов вниз к 0 процентам в течение каждого из них:



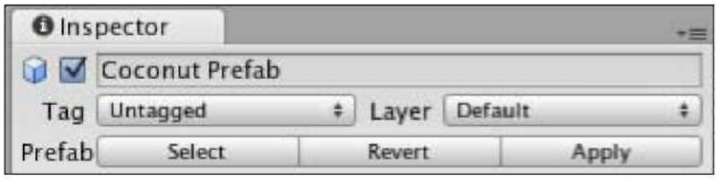
Продолжите входить в эти параметры настройки, пока у Вас нет кое-чего как показанный ниже:

Остающиеся параметры настройки можно оставить в их неплатежах - **Расстояние Вершины Минуты** просто определяет то, чем самое короткое расстояние между двумя пунктами в линии может быть - чем больше подарка пунктов, тем более детальный линия, но также и более дорогостоящее, которое это обрабатывает мудрый. **Авторазрушьте** не должен быть позволен также, поскольку у самого объекта есть сценарий, обращающийся с удалением этих prefab от мира - **Кокосовый орех Приводит в порядок** сценарий, который мы написали в Главе 6.

Обновление prefab

Поскольку мы эффективно воздействуем на случай prefab, который потерял его связь с оригинальным активом следовательно предупреждение, когда мы добавили, что след renderer-мы должен применить изменения, мы сделали к оригинальному активу, чтобы иметь все новые случаи prefab, чтобы показать этот след.

Чтобы сделать это, у Вас есть два варианта: или выберите **Кокосовый Prefab** в **Иерархии** и пойдите в **GameObject |, Применяют Изменения к Prefab,** или используют кнопку **Apply** наверху **Инспектора** для этого объекта:



Теперь, когда Вы обновили оригинальный заранее приготовленный актив, мы больше не нуждаемся в случае в сцене. Таким образом мы просто выбираем это и удаляем это использующий сокращенную *Команду* клавиатуры *+, Клавиша Backspace* (Mac) или *Изменение + Удаляет* (PC).

Чтобы видеть эффект, игра проверяет игру теперь, и испытывает кокосовую застенчивую миниигру. Вы должны видеть, что пылающие следы следуют за каждым кокосовым орехом, который Вы бросаете!

Работа tweaked (щипнуть)

В этой секции мы будем смотреть на пути, которыми Вы можете повысить работу своей игры как продукт конца. Также известный как оптимизация, этот процесс крайне важен, чтобы сделать, как только Вы гарантировали, что Ваша игра работает как ожидалось.

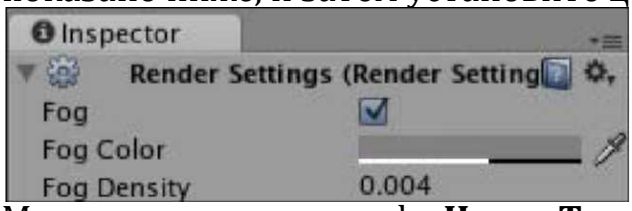
Самолеты Скрепки Камеры и туман

Чтобы добавить более хорошее визуальное появление к нашему острову, мы позволим туман. В Unity туман может быть позволен очень просто и может использоваться в соединении с **Далеким** урегулированием **Самолета Скрепки** Камеры, чтобы приспособить расстояние ничьей - то, чтобы заставлять объекты вне определенного расстояния не быть предоставленным. Это улучшит работу. Включением тумана Вы будете в состоянии замаскировать сокращение предоставления отдаленного предоставления объектов менее неуклюжее чувство к исследованию острова. Мы обсуждали **Далекие** параметры настройки **Самолета Скрепки** в Главе 3, когда мы вскрывали противоречия в Первом Диспетчере Человека. Теперь, давайте приспособим ценность далекого самолета, чтобы улучшить работу, сокращая расстояние, на котором объекты все еще предоставлены камерой.

- Расширьте **Первую** группу родителя **Диспетчера Человека**, щелкая ее серой стрелкой налево от ее имени в группе **Иерархии**.
- Выберите детский объект, названный **Главной Камерой.**
- В **Инспекторе,** найдите компонент **Камеры**, и установите **Далекий Самолет Скрепки**

оцените **600.** Это - более короткое расстояние, описанное в метрах, и хотя это сокращает визуальное расстояние взгляда нашего игрока, это будет

замаскировано туманом, который мы собираемся добавить. Пойдите, чтобы **Отредактировать |, Отдают Параметры настройки.** Это поднимает**, Отдают Параметры настройки** вместо **Инспектора.** Просто выберите коробку для **Тумана** здесь, и затем нажмите на **Цветной** блок направо от **Цвета Тумана**, чтобы открыть параметры настройки для Цвета и Альфы. Установите Альфа-ценность (A) приблизительно в 60 %, как показано ниже, и затем установите ценность **Плотности Тумана** в **0.004:**



Мы устанавливаем альфу **Цвета Тумана** и **Плотность Тумана** к нижнему значению, поскольку неплатеж и более высокие ценности маскируют вид настолько хорошо, что частицы от вулкана стали бы невидимыми до стендов игрока весьма близко к вулкану.

Окружающее освещение

В **Отдают Параметры настройки,** Вы можете также установить **Рассеянный свет** сцены. В то время как наш **Направленный Свет** обращается с главным действием освещения как с солнцем в этом примере - рассеянный свет позволит Вам устанавливать общую полную яркость, означая, что Вы можете создать сцены, которые похожи на определенное время ночи или дня. Попытайтесь приспособить это урегулирование теперь, нажимая на цветной блок направо от урегулирования и экспериментирования с цветным сборщиком.

Сцена инструкций

Чтобы закончить нашу игру, мы закончим меню, которое мы сделали в Главе 8, создавая сцену **Инструкций** для пользователя, чтобы читать. В этом мы осуществим некоторую мультипликацию, используя scripting, и мы изучим новую команду, которую мы не использовали еще названный **линейной вставкой,** или **lerp** для короткого.

Поскольку наша сцена **Инструкций** должна подражать остальной части меню, мы начнем со сцены **Меню** как основание. Прежде, чем мы сделаем это, однако, гарантирует, что сцена **Уровня Острова** спасена, идя в **Файл |, Экономят Сцену.** Дублируйте сцену **Меню**, выбирая это в **Проектной** группе и используя сокращенную *Команду* клавиатуры *+ D* (Mac) или *Ctrl + D* (PC).

Это дублирует сцену **Меню**, и даст ей название **Меню 1,** переименовать это **Инструкции,** и открыть ее, щелкая два раза на его изображении теперь.

Добавление текста экрана

Мы должны будем написать наши инструкции для игрока на этом экране, таким образом мы используем объект Text GUI с этой целью. Пойдите в

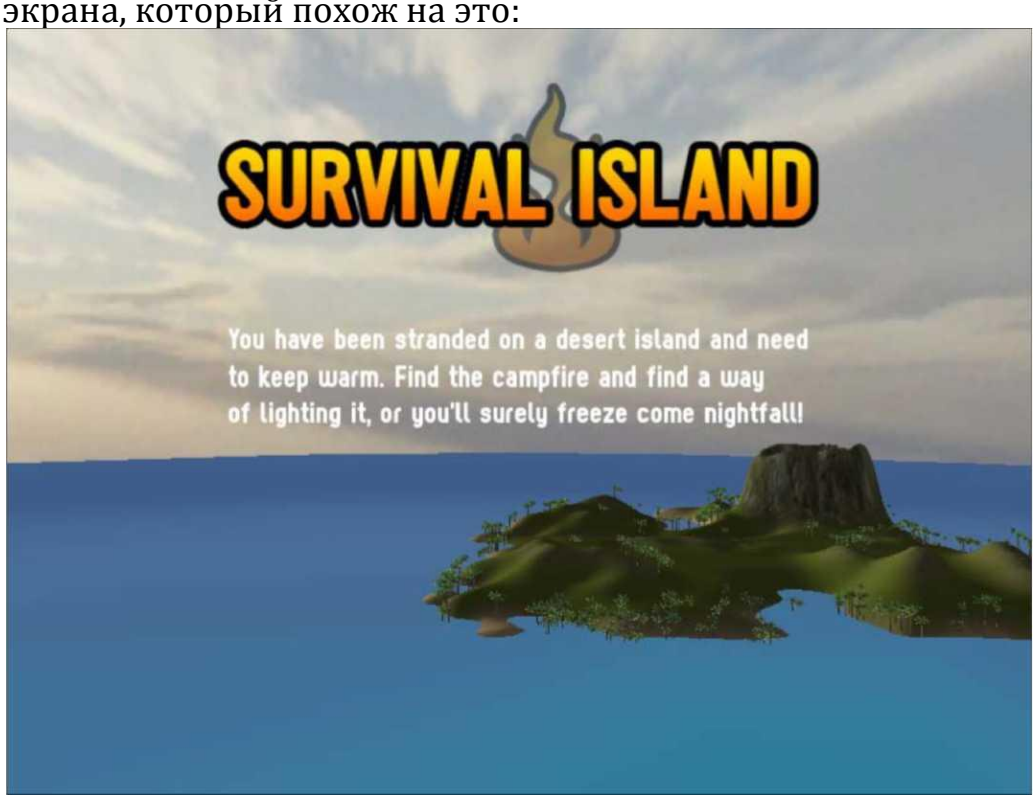
GameObject |, Создают Другой | Текст GUI. Это создает новый объект с Текстовым компонентом GUI под названием **Текст GUI** в **Иерархии.** Выберите это и переименуйте это **Текст Инструкции.**

Затем, чтобы поддержать последовательность, назначьте шрифт, который Вы имели обыкновение к настоящему времени в Вашей игре как шрифт для Структуры GUI использовать. Я использовал свободное, чтобы загрузить шрифт, **Sugo,** таким образом я просто тянул это от **Проектной** группы в область **Шрифта** в **Текстовом** компоненте **GUI**.

Затем напишите в коротком параграфе, объясняя цель игры игроку. Они должны осветить походный костер спичками, чтобы выжить, таким образом я написал:

> "*Вы заблудились на необитаемом острове и нуждаетесь в костре! Найдите походный костер и найдите способ запалить его, или Вы конечно замерзнете, прибывают сумерки...*"

Чтобы написать в тексте на многократных линиях в **Текстовой** области этого компонента, просто закончите свою линию, и двиньтесь в следующую линию при использовании сокращенного *Высокого звука* клавиатуры *+, Вступают* (Mac и PC).
Наконец, поместите этот текст, помещая следующие ценности в область позиции **Трэнсформа - (0.5, 0.55, 0).** Я хотел помещать свой текст в три линии так, чтобы он визуально выстроился в линию с эмблемой наверху экрана, который похож на это:



Текстовая Мультипликация, используя Линейную Вставку (Lerp)

Затем мы оживим этот текст, используя сценарий, так выберите папку **Меню** в **Проектной** группе и нажмите**, Создают,** выбирая **Javascript** из опускаться меню.

Переименуйте **NewBehaviorScript, который** Вы создали, вызывая это **Аниматор.** Теперь щелкните два раза изображением сценария, чтобы начать это в редакторе сценария.
Мы начнем, устанавливая некоторые общественные variables участника, которые мы можем использовать в **Инспекторе**, чтобы управлять поведением нашей текстовой мультипликации. Добавьте следующий к вершине Вашего сценария:

```
var startPosition :     float = -1.0; var
endPosition :     float = 0.5; var speed :     float =
1.0; private var StartTime     : float;
```

Мы будем использовать эти variables в следующей части сценария, но мы назвали их основанными на том, что они делают - начало и заканчивают, Variables позиции отвечают за позицию в координатах экрана (следовательно позиция начала по умолчанию-1, означая от экрана). Учитывая, что они - общественные variables участника, мы будем в состоянии приспособить их в **Инспекторе.** Variables скорости будет просто использоваться, чтобы умножить скорость мультипликации в течение долгого времени, таким образом неплатеж будет 1.

Наконец, мы включаем Variables по имени StartTime, Variables плавающей запятой, который только используется в пределах сценария и поэтому объявлен как частный. Мы будем использовать этот Variables, чтобы сохранить ценность по умолчанию времени, когда сцена будет загружена. Если мы не делаем этого, после того, как первая погрузка этой сцены, любые посещения возвращения экрана инструкций покажут инструкции уже относительно экрана, как ценность времени в Unity, мы собираемся использовать счет от первого груза игры.

Затем, чтобы настроить наше назначение этого Variables, мы захватим команду времени Unity, когда сцена загрузит, добавляя следующий Functions:

```
Теперь, кofunction Start(){
    StartTime = Time.time;
}
```

гда сцена загружает, захваты StartTime та ценность. Мы будем использовать это, устанавливая мультипликацию.

Теперь, переместите заключительную вьющуюся скобу Update () function вниз несколькими линиями так, чтобы Вы могли добавить некоторый код к Functions. Добавьте следующую линию:

```
transform.position.x = Mathf.Lerp(startPosition,     endPosition, (Time.
time-StartTime)*speed);
```

Здесь мы выбираем определенную ось ценностей позиции компонента transform (преобразовать), поэтому:

```
transform.position.x
```

Тогда мы заставляем это равняться Functions-Mathf Математики - названный Lerp, который является Functions что линейно интерполир-путешествия непосредственно - между двумя ценностями. Ценности, которые мы посылаем этому Functions, определены нашим startPosition и endPosition variables, таким образом Functions Lerp обеспечит число для X позиций оси между теми двумя числами.

Третий параметр для Functions Lerp - количество, чтобы интерполировать - ценность 1 означала бы, что ценность возвратила путешествия полностью из начала, чтобы закончить ценность, и ценность 0 не будет означать изменения. Мы хотим, чтобы эта вставка произошла в течение долгого времени здесь, таким образом вместо отдельной ценности, мы используем Unity, построил в команде Time.time, чтобы подсчитать и вычитание ценности startTime, который мы устанавливаем ранее - эффективно сброс его ценности так, чтобы это могло рассчитать от 0. Наконец, мы изменяем время, умножаясь нашим Variables скорости. В настоящее время Variables скорости установлен в 1.0, таким образом никакое изменение не будет произведено, но никакая ценность, более чем 1 увеличит скорость, и ценности ниже чем 1 уменьшат это.

Эта команда должна быть сделана в Update () function, поскольку это требует возрастающего изменения - который наш lerp обеспечивает - каждая структура, таким образом помещая это в Начало (), Functions, например, не работал бы.

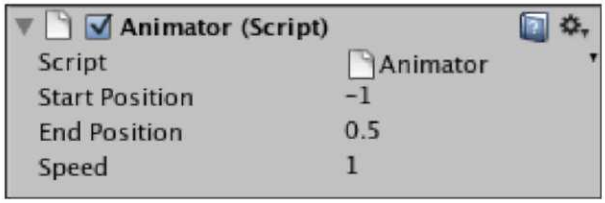
Чтобы перепроверить Ваш сценарий, здесь это закончено:

```
var startPosition :     float = -1.0; var
endPosition :     float = 0.5; var speed :     float
= 1.0; private var StartTime     : float;

function Start(){
    StartTime = Time.time;
}
function Update () {
    transform.position.x = Mathf.Lerp(startPosition, endPosition,
 (Time.time-StartTime)*speed);
}
```

Пойдите в **Файл |, Экономят** в редакторе сценария теперь, и переключаются назад на Unity.

Теперь мы применим сценарий и приспособим параметры настройки общественных variables участника в **Инспекторе**, чтобы гарантировать, что он делает то, что мы хотим. Выберите объект **Instructions Text** в группе **Иерархии**, и пойдите в **Компонент | Сценарии | Аниматор.**

Это добавляет сценарий как компонент. В параметрах настройки общественных variables участника Вы должны видеть ценности по умолчанию, установленные в сценарии, который Вы только написали:



Чтобы видеть этот эффект в действии, мы должны будем играть сцену, но сначала, мы должны удалить меню, которое было перенесено от дублирования оригинальной сцены меню. Если Вы выбирали меню Texture-based GUI, то Вы должны будете проверить все три объекта кнопки. Если Вы используете меню GUI-scripted, то просто проверяют объект, у которого есть тот сценарий как компонент. Чтобы напомнить себе о дезактивации, см. предыдущую главу.

Как только Вы дезактивировали меню, возвратитесь к объекту **Instructions Text**. В составляющем **Тексте GUI**, **Якорь** набора **Среднему Центру** и **Выравнивание** к **Левому.** Теперь нажмите кнопку **Play**, чтобы видеть эффект мультипликации. Ваш текст должен переместиться в **X** осей от за кадром левого к центральной позиции **0.5.** Нажмите **Игру** снова, чтобы закончить Ваш тест перед продолжением.

Если Вы хотели бы, чтобы мультипликация произошла от за кадром права, то Вы могли заменить Variables участника общественности **Позиции Начала** к числу выше чем 1.0 - поскольку это - правильный край экрана. Однако, если бы Вы хотели, чтобы мультипликация произошла в Оси Y, то Вы должны были бы возвратиться к Вашему сценарию, и изменить ось эффекта там. Например, transform.position.y = Mathf (...)

Возвращение меню

Поскольку мы воздействуем на сцену, которая является отдельной к нашему меню, мы должны будем включать кнопку, чтобы возвратить пользователя сцене **Меню** непосредственно. Иначе они застрянут на экране **Instructions**!

Для этого мы будем использовать GUI scripting техника, обсужденная в предыдущей главе, и экономить некоторое время, дублируя часть существующей работы, которую мы уже сделали. В папке **Сценариев Проектной** группы, определите местонахождение сценария **MainMenuGUI2,** и дублируйте это использующий сокращенную *Команду* клавиатуры *+ D* (Mac) или *Ctrl + D* (PC). Как объекты чисел Unity и активы, которые это производит, Ваш дубликат назовут **MainMenuGUI3,** так что переименуйте этот **BackButtonGUI.** Тогда щелкните два раза его изображением, чтобы начать это в редакторе сценария.
В первом, если утверждение onGUi () Functions, приспособьте текст на кнопке, чтобы ответить вместо Игры. В звонке в OpenLevel () Functions,

набор последовательность, чтобы сказать Меню, вместо Уровня острова. Это должно быть похожим на это:

```
if(GUILayout.Button ("Back")){
    OpenLevel("Menu");
}
```

Поскольку мы только хотим, чтобы этот сценарий произвел одну кнопку, удалил другие два если утверждения, оставляя Вас с только отдельным, чтобы возвратиться к сцене Меню. Единственная другая проблема здесь - расположение кнопки. Поскольку мы используем screenY Variables от нашего сценария **MainMenuGUI2** - это помещает нашу область GUI, чтобы потянуть кнопку в центре экрана. Поскольку наш текст будет в пути этого, мы идеально должны отдать кнопку ниже вниз. Чтобы обойти это, просто исправьте учреждение Variables ScreenY, чтобы использовать немного нижнее значение:

вар ScreenY = ((Screen.height / 1.7) - (areaHeight / 2)); Это

поместит кнопку ниже чем учебный текст. Законченный

сценарий **BackButtonGUI** должен быть похожим на это:

```
var beep     : AudioClip; var menuSkin :
GUiSkin; var areaWidth : float; var
areaHeight       : float;

function OnGUi(){

    GUI.skin = menuSkin;

    var ScreenX =       ((Screen.width / 2)       -       (areaWidth / 2)); var
    ScreenY =       ((Screen.height / 1.7)       -       (areaHeight / 2));

    GUILayout.BeginArea     (Rect (ScreenX,ScreenY,
                            areaWidth, areaHeight));

    if(GUILayout.Button ("Back")){
        OpenLevel("Menu");
    }

    GUILayout.EndArea();
}

function OpenLevel(level       : String){
        audio.PlayOneShot(beep);
        yield new WaitForSeconds(0.35);
```

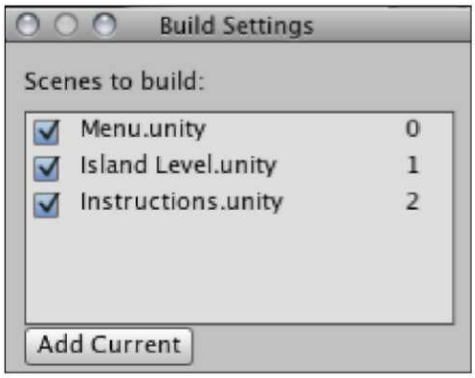
```
        Application.LoadLevel(level);
    }

@script RequireComponent(AudioSource)
```

Пойдите в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity теперь.

Как со всеми GUI-подготовленными элементами интерфейса, мы должны приложить сценарий к объекту для этой кнопки, чтобы появиться. Создайте новый пустой объект игры, чтобы жить, этот сценарий, идя в **GameObject | Создают Пустой.** С новым объектом, отобранным в **Иерархии,** переименуйте это **Назад Кнопка.** Приложите сценарий мы только сделанный, идя в **Компонент | Сценарии | Обратная Кнопка GUI.** Поскольку Вы заметите, общественные variables участника назначения потребности сценария, таким образом Вы должны будете сделать следующее:

- Тяните звуковую скрепку **menu_beep** от папки **Меню** в **Проектной** группе к Variables участника **Звукового сигнала**
- Тяните **Главный** актив **Меню кожи** от папки **Меню** до Variables **Меню Кожи**
- Установите Variables **Ширины Области** в **200**
- Установите Variables **Высоты Области** в **75,** поскольку у нас только есть отдельная кнопка на сей раз

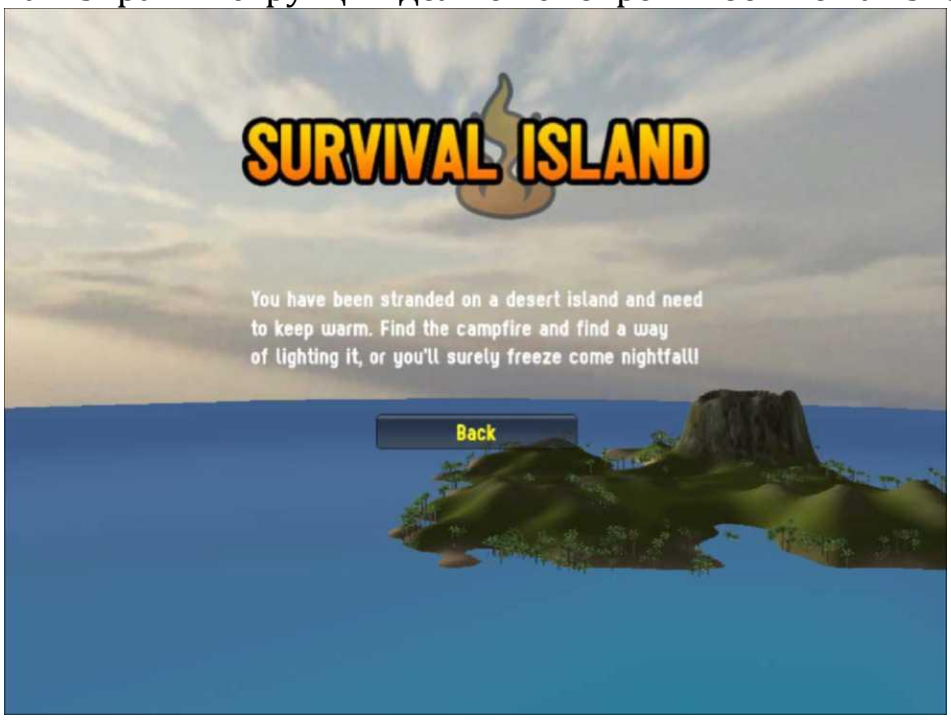
Теперь, когда наша **Обратная** кнопка полна, мы можем проверить все меню. Но сначала мы должны добавить, что сцена **Инструкций** к проекту **Строит Параметры настройки,** для Unity, чтобы загрузить это во время тестирования. Пойдите в **Файл |, Строят Параметры настройки,** и нажимают на кнопку **Add Current**. Ваш список уровней в **Построить Параметрах настройки** должен теперь быть похожим на это:



Закройте **Построить** диалог **Параметров настройки**, и Вы готовы проверить. Нажмите кнопку **Play**, чтобы проверить сцену - **Обратная**

кнопка должна появиться, и текст должен оживить в. Щелкните **Обратной** кнопкой, и Вы должны быть взяты к сцене **Меню**. Если любое из этого не работает, то перепроверьте те свои спички сценария упомянутые выше изменения.

Ваш экран инструкций должен смотреть кое-что как это:



Теперь прекратите проверять сцену, и пойдите в **Файл |, Экономят Сцену**, чтобы обновить это.

Постепенное появление изображения уровня острова

Чтобы освободить игрока в окружающую среду, когда они входят в игру, мы создадим постепенное появление изображения в начале нашего фактического уровня игры, используя структуру GUI, которая покрывает экран и постепенно исчезается, в течение долгого времени используя технику Lerp, которую мы только что изучили.

Щелкните два раза изображением **Уровня Острова**, чтобы открыть ту сцену. Теперь, в папке **Вулкана** Вы импортировали ранее, Вы найдете файл структуры названным **белым.** Эта структура, плоский белый цвет, созданный в Фотомагазине, имеет размер 64x64 пикселы; это может казаться довольно маленьким, чтобы покрыть экран, но поскольку это - просто плоский цвет, это не должно быть большим - мы просто протянем это к размеру экрана.

Выберите структуру теперь и в **Инспекторе,** отсейте**, Производят Карты Mip** в компоненте **Импортера Структуры**, чтобы мешать Unity отдать меньшие версии этого для трехмерной прессы использования тогда кнопка **Apply** в основании, чтобы подтвердить изменение. Мы будем

использовать эту структуру с далее UnityGUI и Lerp scripting, чтобы протянуть это к полному экрану и исчезнуть это, когда сцена начнет - это заставит вид игры усилиться от белого.

Выберите папку **Сценариев**, и нажмите на кнопку **Create**, выбирая **Javascript** из опускаться меню. Переименуйте этот сценарий **FadeTexture.** Щелкните два раза изображением сценария, чтобы начать это в редакторе сценария.

Начните, устанавливая два variables, один общественный участник и другой частный Variables:

```
var theTexture     : Texture2D; private var
StartTime     : float;
```

Первый Variables здесь будет считать белый актив структуры готовым быть протянутым по экрану, и вторым является число с плавающей запятой, которое мы будем использовать, чтобы сохранить ценность времени.

Затем, захватите ценность Time.time в Variables StartTime, когда уровень загружает, добавляя следующий Functions к Вашему сценарию ниже variables, Вы только добавили:

```
function OnLevelWasLoaded(){
    StartTime = Time.time;
}
```

Мы должны использовать этот Variables, чтобы захватить текущее законченное количество времени, потому что Time.time начинается с того, когда первая сцена загружена - то есть, меню. Поскольку игрок рассмотрит меню сначала, мы знаем, что Time.time не будет равняться 0, когда сцена **Уровня Острова** будет загружена. Мы будем использовать этот Variables коротко, чтобы вычесть из текущего чтения времени и таким образом заставить пункт рассчитывать от.

Теперь в Update () function, поместите в следующем коде:

```
    if(Time.time-StartTime >= 3){
        Destroy(gameObject);
    }
```

Это, если утверждение проверяет ценность Time.time, минус время, это было в том, когда текущая нагруженная-StartTime сцена - и если это больше чем или равно 3, тогда мы разрушаем объект игры, что этот сценарий присоединен. Это просто удаляет объект игры, к которому что мы приложим этот сценарий после спустя три секунды как после того, как исчезнуть эффект произойдет, мы больше не нуждаемся в объекте в сцене.

Предоставление структуры UnityGUI

Установите новый OnGUI () Functions в Вашем сценарии, и место в

следующих линиях кода:

```
функционируйте OnGUI () {
        GUI.color = Color.white;
        GUI.color.a = Mathf. Lerp (1.0, 0.0, (Time.time-StartTime));
        GUI.DrawTexture (Rect (0, 0, Screen.width,
                                        Screen.height), theTexture);
}
```

Здесь мы обращаемся к Classes GUI непосредственно. В первой линии мы обращаемся к цветному параметру, устанавливая это во встроенную ссылку Цветного Classes, названного "белым".

Тогда с этим цветным набором, мы обращаемся к его альфа-параметру - его видимости - говоря GUI.color.a. Мы используем тот же самый Mathf. Lerp приказывают, чтобы мы имели обыкновение оживлять наш объект Text GUI ранее, интерполируя альфу от 1.0 (полностью видимый) к 0.0 (невидимый). Третий параметр в петле - количество, чтобы интерполировать - потому что мы используем Time.time-StartTime здесь, мы эффективно начинаем прилавок от 0.0, который увеличится как проходы времени, таким образом Lerp произойдет в течение долгого времени, создавая исчезновение.

Третья линия фактически отдает структуру непосредственно, используя команду DrawTexture Classes GUI. Определяя Rect (прямоугольное место) для этого, чтобы быть оттянутыми, начинаясь в 0,0 - верхний левый из экрана - мы удостоверяемся, что эта структура простирается, чтобы заполнить экран при использовании Screen.width и Screen.height. Это делает исчезнуть работу в том, какой бы ни решение, в котором игрок управляет игрой, поскольку это автоматизирует размер структуры на экране.

Пойдите в **Файл |, Экономят** в редакторе сценария, и возвращаются к Unity теперь.

Назад в Unity, пойдите в **GameObject |, Создают Пустой.** Это создает новый объект в **Иерархии** по имени **GameObject.** Переименуйте это к **Микшеру** теперь. Добавьте сценарий, который Вы только что написали, идя в **Компонент |, Сценарии | Исчезают Структура.** Это добавляет сценарий как компонент. Теперь, просто drag and drop (перетащили и опустили) **белую** структуру от папки **Вулкана** в **Проектной** группе к общественному Variables участника **theTexture** в **Инспекторе.**
Теперь нажмите кнопку **Play**, и Вы должны видеть, что экран усиливается от белого и что **белый** объект игры в **Иерархии** удален после трех секунд. Прекратите проверять, и пойдите в **Файл |, Экономят Сцену**, чтобы обновить проект теперь.

Уведомление победы игры

Как заключительный последний штрих, мы скажем игроку, что они успешно выиграли игру, когда огонь был зажжен, поскольку это - цель нашей игры.

Откройте сценарий **PlayerCollisions** в папке **Сценариев Проектной** группы, и свиток к основанию. Последний Functions в сценарии - lightFire (), и в это мы добавим еще некоторые команды перед его правом завершения вьющаяся скоба. Спустите несколько линий от потока последняя линия:

```
Destroy(GameObject.Find("matchGUI")),-- И
```

место в следующих командах:

```
TextHints.textOn=true;
TextHints.message = "You Lit the Fire,     you'll survive, well done!";

yield new WaitForSeconds(5);
Application.LoadLevel("Menu");
```

Здесь мы переключаемся назад на нашем TextHints GUI от ранее, и посылаем сообщение, "Вы Зажгли Огонь..." как последовательность текста, чтобы показать на экране. Мы тогда используем команду урожая, чтобы остановить сценарий в течение 5 секунд, и затем загрузить уровень Меню игры, так, чтобы игрок мог играть снова.

Теперь нажмите кнопку **Play**, и игру через целую игру. Вы должны быть в состоянии собрать три батареи, выиграть четвертую батарею от кокосовой застенчивой миниигры, войти в заставу, собрать спички, и зажечь огонь. Когда огонь зажжен, Вы должны быть уведомлены, и взяты к главному экрану меню.

Как только Вы проверили это, прекратите проверять, и пойдите в **Файл |, Экономят сцену**, чтобы обновить проект.

Резюме

В этой главе мы смотрели на различные последние штрихи для Вашей игры. Визуальные эффекты, освещение, и мультипликация, обсужденная здесь только, царапают поверхность того, что Вы можете сделать с Unity, но в то время как Unity облегчает добавлять эти особенности полировки, чтобы заставить Вашу игру действительно выделиться, крайне важно иметь в виду, что их нужно только рассмотреть, как только gameplay Вашего проекта - зат-последние-штрихи, отличный способ закончить Ваш проект, но пригодность для игры должна всегда быть на первом месте.

Теперь, когда мы закончили игру, мы потратим следующее смотрение главы на здание, тестирование, и восстановление, и значения развертывания Вашей игры. Мы будем также смотреть на дальнейшую оптимизацию для Вашей игры и обсуждать получение Вашей игры, замеченной как независимый разработчик.

10

Здание и Разделение

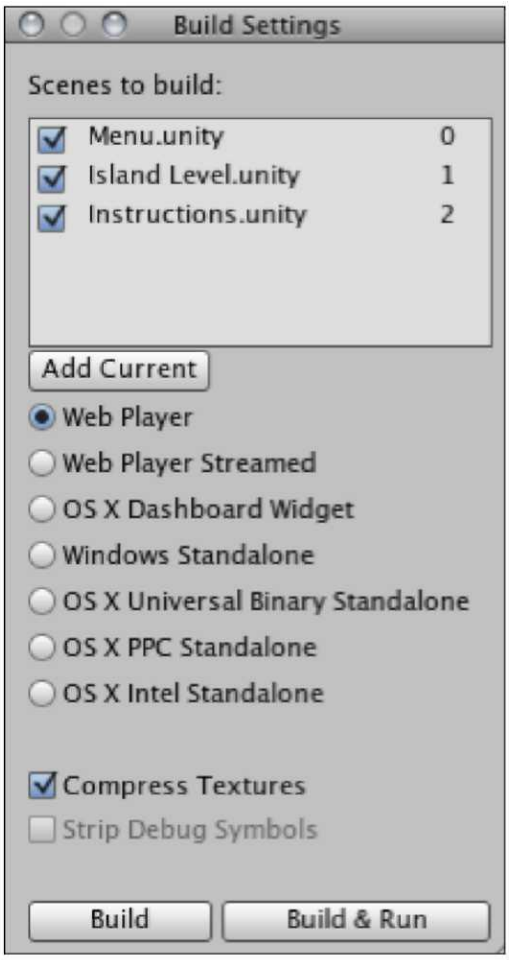
Чтобы взять нашу игру от простого примера кое до чего, мы можем поделиться с тестерами игры, мы должны рассмотреть различные платформы развертывания и как мы можем приспособить игру, которая будет экспортироваться в Сеть. Лучший метод для Вас как разработчик должен разделить Вашу работу. Унити учитывает различные весы финала, строят из Вашей игры и сожмет структуры и различные другие активы как соответствующие для Вас. Вы должны также знать об обнаружении платформы для сети, строит, чтобы приспособить определенные параметры настройки, когда развертывание онлайн, в противоположность полному автономному рабочему столу строит.

Стандартная Инди и Про выпуски Unity предлагают Вам шанс построить для рабочего стола Mac, рабочего стола Windows, как Виджет для Mac средство **'Приборной панели'** X OS, и как плагин web-браузера.

В этой последней главе мы будем смотреть на тот, как настроить активы, чтобы создать сеть, строят, и автономный рабочий стол строят. Мы затронем следующие темы:

- Работа со **Строит Параметры настройки**, чтобы экспортировать Вашу игру
- Построение сети и автономной версии Вашей игры
- Обнаружение платформы, чтобы удалить элементы из сети строит
- Пути, которыми Вы можете поделиться своими играми с другими и получить дальнейшую помощь с Вашим развитием Unity

Постройте Параметры настройки
В Unity, пойдите в **Файл |, Строят Параметры настройки** теперь, и смотрят по вариантам, которые Вы имеете. Вы должны видеть различные варианты, упомянутые ранее:



В **Построить Параметрах настройки** Мас строит, отмечены приставкой **OS X** - текущее поколение операционной системы. Мас строит, также дают Вам различные варианты, поскольку есть различные поколения компьютера Мас, чтобы рассмотреть - старшее поколение, бегущее на процессоре **PowerPC**, и текущем поколении построенный вокруг процессоров **Интела**. **Универсальное Двойное** урегулирование построит OS X наборов из двух предметов, которые управляют на обоих старшими системами PowerPC так же как новыми системами Интела. Это приводит к большему файлу, поскольку Вы эффективно включаете две копии своей игры в одном применении.
В нашем примере **Построить Параметры настройки** показывают список сцен, которые мы добавили к нашему проекту пока, начиная со сцены **Меню**. Важно иметь первую сцену, которую Вы хотели бы, чтобы Ваш игрок видел как первый пункт в **Сценах, чтобы построить** список. Если Ваше меню или первая сцена не являются первыми в списке, то Вы можете просто тянуть названия сцен, чтобы переупорядочить их.

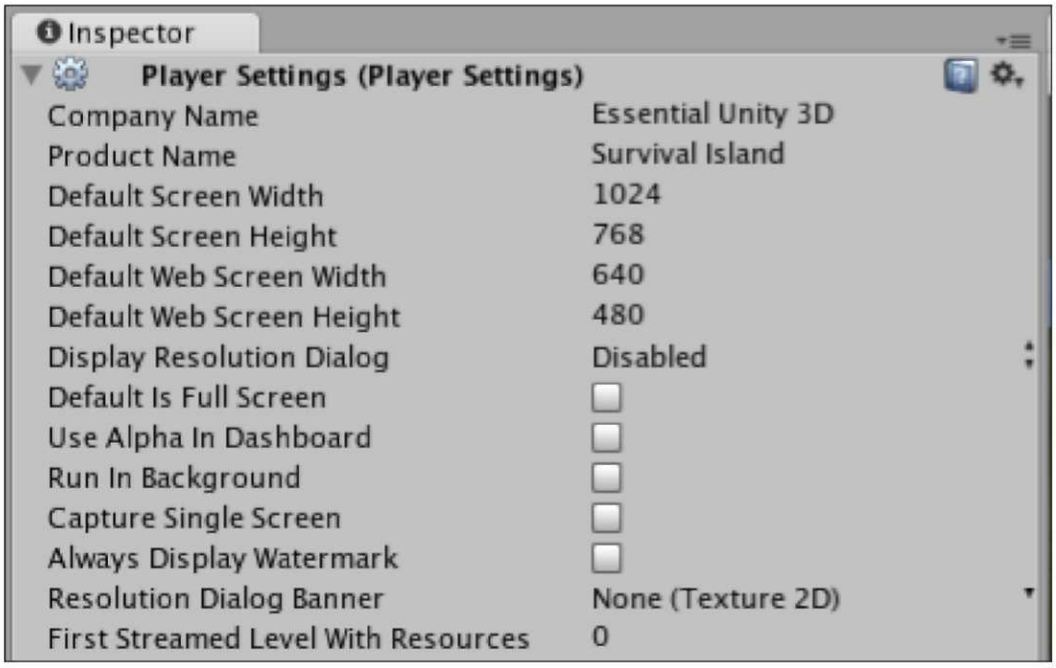
Давайте начнем, смотря на различные варианты в большей глубине и что Вы прошли бы построение каждой опции.

Игрок Сети

Помещая любое основанное на плагине содержание на Сеть, это должно быть включено как объект, который вызывает установленный плагин. Зрители сети Unity строят, будет обязан загружать плагин для их браузера почти таким же способом, поскольку содержание Adobe Flash требует, чтобы пользователи загрузили Игрока Вспышки. Игрок сети строит, создают файл игры с расширением.unity3d, который вызывает плагин Игрока Сети Unity, наряду с сопровождающим файлом HTML, содержащим необходимый объемлющий код. Этот объемлющий HTML может тогда быть взят наряду с файлом игры и включен в веб-страницу Вашего собственного проекта.

Параметры настройки Игрока

Поскольку игроки сети полагаются на программное обеспечение браузера, чтобы загрузить HTML, содержащий звонок в плагин, любой компьютер, управляющий Вашей игрой, поскольку игрок сети уже израсходовал власть обработки на браузере. С этим в памяти, это может помочь обеспечить Вашу игру при более низком решении, чем Вы были бы для рабочего стола строить. Мы проектировали нашу игру к настольному решению начального уровня 1024x768 пикселы. Однако, когда развертывание в сеть строит, размер экрана должен быть уменьшен кое до чего меньшего, такой как 640x480. Это делает груз на центральном процессоре менее интенсивным, поскольку меньшие структуры оттягиваются, таким образом давая лучшую работу.
Чтобы приспособить параметры настройки, такие как это, мы должны будем смотреть на **Параметры настройки Игрока.** Пойдите, чтобы **Отредактировать | Проектные Параметры настройки | Игрок** теперь, чтобы переключить группу **Инспектора**, чтобы показать параметры настройки для проекта. Когда Вы строите свою игру, Вы эффективно помещаете свой законченный проект в **Игрока** - на Сети, файл игрока, вызывающий плагин непосредственно. В **Параметрах настройки Игрока** Вы можете определить определенные элементы - такие как размер экрана - для игрока, чтобы использовать.



На этом экране Вы должны начать, заполняясь в деталях для Вашего проекта. Добавьте в **Названии компании** и **Названии продукта,** которое на этой стадии не должно быть ничем формальным. Тогда заполнитесь в **Ширине Экрана По умолчанию** и **Высоте** с **1024**x**768**, мы первоначально проектировали для.

Теперь заполнитесь в **Ширине Экрана Сети По умолчанию** и **Высоте** с **640** и **480** соответственно, как показано в предыдущем скриншоте. Единственное другое урегулирование, относящееся к нашему развертыванию игрока сети, является урегулированием для **Первого, Тек Уровень С Ресурсами,** который позволяет Вам выбирать первый уровень в Вашем строящего список, у которого есть ряд активов, чтобы загрузить в. С этим Вы можете создать экраны всплеска, который segway в Вашу игру. Если бы это имело место, то Вы использовали бы это урегулирование, чтобы выбрать специфический уровень, который сначала показывает активы, сообщая его число в заказе построить списка.
В нашей игре первый экран меню содержит активы - остров - и мы можем счастливо оставить эту установку по умолчанию на **0.** Мы возвратимся к **Параметрам настройки Игрока** коротко, когда мы будем смотреть на рабочий стол, строит.

Игрок Сети Тек
Игрок Сети, Текший-а отдельный, строит опцию в списке - позволяет Вам строить развертывание сети, которое не вынуждает игрока ждать слишком долго бара погрузки, чтобы закончить.

Сталкиваясь любой ждет онлайн, это характерно для пользователей, чтобы быть нетерпеливым, и важно, что Вы ограничиваете времена ожидания в

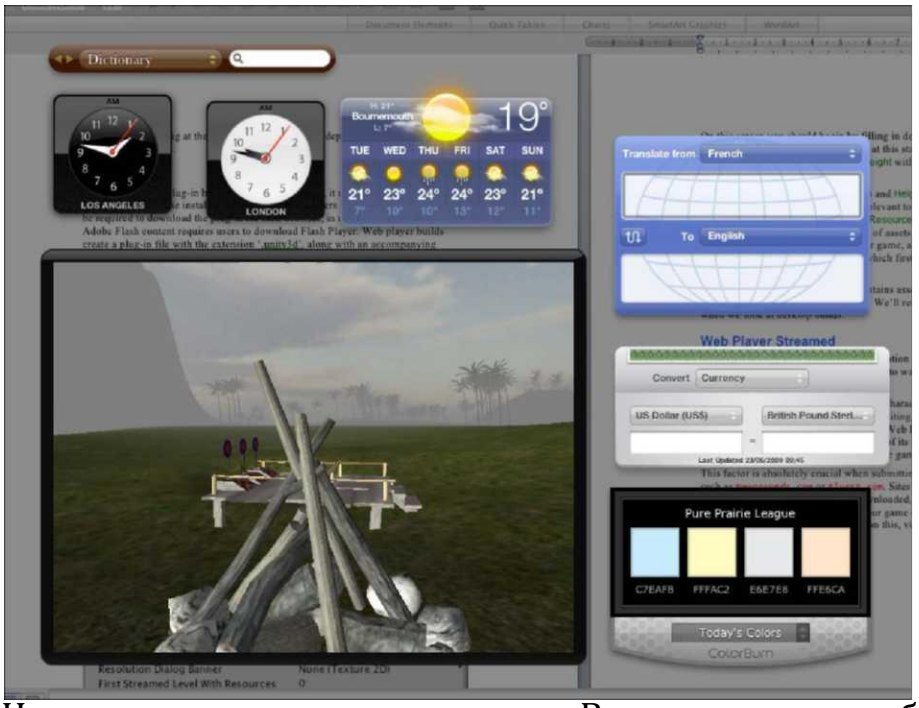
максимально возможной степени, когда создание сети строит из Вашей игры. Используя **Сеть Игрок Тек** средства, что Ваша игра может начать играться перед полнотой ее активов загружены тогда, поскольку игрок взаимодействует с первой сценой, остальная часть активов игры продолжают загружать.

Этот фактор абсолютно крайне важен, представляя Вашу игру участкам двери игр, такой как www.shockwave.com <http://www.shockwave.com>, www.wooglie.com <http://www.wooglie.com>, или www.blurst.com <http://www.blurst.com>. Участки, такие как они ожидают, что Ваша игра будет играема после того, как приблизительно 1 МБ данных был загружен, и придерживаясь этих руководящих принципов, Вы, более вероятно, будете в состоянии получить свою игру на такие участки, давая Вам подвергание для Вашей работы. Для получения дополнительной информации об этом, посетите вебсайт Unity.

OS X Виджетов Приборной панели

У операционных систем Mac (версия 10.4 вперед) есть особенность, названная Приборной панелью. Это - ряд простых инструментов и применений, известных как **Виджеты,** которые могут быть подняты в любое время как оверлей по экрану. Unity может издать Вашу игру как Виджет, и это просто иначе для Вас, чтобы предоставить Вашу игру людям. Если Вы делаете простую загадку или timewaster игру, это могло бы быть соответствующим как метод развертывания. Однако, с игрой, такой как наш первый человек walkaround, Приборная панель является менее соответствующей.

Вот то, на что наша игра похожа как часть Мак Дэшбоарда:



Идеально, игры, развернутые как Виджеты, должны быть основными, потому что лучше избегать загружать массы данных в Виджет Приборной панели, поскольку это должно остаться житель в памяти компьютера так, чтобы игра могла продолжиться, когда Приборная панель активизирована и дезактивирована.

OS Автономный X/Windows

Автономный или настольный строит, абсолютные применения, которые могут быть поставлены таким же образом как коммерческая игра.

Построение Вашей игры для Mac, OS X автономный построит отдельный прикладной файл со всеми необходимыми активами, связанными внутри, строя для автономной Windows PC, создаст папку, содержащую (выполнимый).exe и активы, требуемые управлять игрой.

Построение автономного является лучшим способом гарантировать максимальную работу от Вашей игры, поскольку файлы хранятся в местном масштабе, не онлайн, и уже не используют власть обработки, управляя браузером или OS X Приборных панелей.

Построение игры

Теперь, когда мы готовы построить игру, Вы должны считать переменные методы развертывания обсужденными ранее, и приспособить проект, который будет построен для Сети так же как автономной игры.

Приспосабливание к сети строит

В Unity трехмерный мир, с которым Вы работаете, полностью измерен

двигателем, который будет представлен в любом решении, которое Вы определяете в **Параметрах настройки Игрока.** Мы также проектировали меню в этой книге, чтобы быть масштабируемыми в различных решениях, используя Classes Экрана к позиции GUIs, основанный на текущем решении. Однако, чтобы узнать об обнаружении платформы мы удалим элемент, мы не хотим быть замеченными в нашей версии сети - кнопка **Quit**. В папке Сценариев в **Проектной** группе, щелкните два раза изображением для MainMenuGUi2, чтобы начать это в редакторе сценария теперь.

Оставленная автоматизация платформы кнопки

Поскольку это - сеть, строят, кнопка **Quit, которую** мы добавили к меню, бессмысленна. Это то, потому что Применение. Оставленный () команды не функционируют, когда в игру Unity играют через игроков браузера вместо этого, просто закрывают счет, содержащий игру, или проводят далеко, когда они закончены, играя. Мы должны исключить эту кнопку из нашего меню сети, но мы не хотим удалять это из нашего сценария, потому что мы все еще хотим, чтобы сценарий отдал кнопку **Quit** в автономном, строят.

Чтобы решить эту проблему, мы используем другую часть Прикладного Classes, названного платформой, которую мы можем использовать, чтобы обнаружить, какое развертывание (рабочий стол, сеть, или Приборная панель) игра строится как.

Мы сделаем это при письме следующего если утверждение:

```
if(Application.platform == RuntimePlatform.OSXWebPlayer ||
  Application.platform == RuntimePlatform.WindowsWebPlayer)
```

Здесь мы просто проверяем параметр платформы Применения, относительно того, управляют ли этим на OSXWebPlayer (Mac) или WindowsWebPlayer (P**Q** - символ || между двумя просто средства 'ИЛИ'. Таким образом здесь мы говорим, если этим управляют на игроке сети, то сделайте кое-что! Теперь мы просто еще должны объединить это с утверждение, потому что мы действительно хотим сделать условие для того, когда игра не развернута для сети, и отдавать кнопку **Quit**.
В 0nGUi () Functions, определите местонахождение, если утверждение, отвечающее за создание кнопки **Quit**, это должно быть похожим на это:

```
if(GUILayout.Button ("Quit")){
  Application.Quit();
}
```

Теперь еще добавьте в следующем если структура, еще помещая оригинальную кнопку **Quit** если утверждение выше в секция. Это должно теперь быть похожим на это:

```
if(Application.platform == RuntimePlatform.OSXWebPlayer ||
  Application.platform == RuntimePlatform.WindowsWebPlayer){
}
else{
```

```
        if(GUILayout.Button ("Quit")){
          Application.Quit();
        }
      }
```

Заметьте здесь, что мы просто оставили пустую линию для, если условие, которое соблюдают, и еще, поместило часть кнопки **Quit** кода в часть, так, чтобы это было только оттянуто, если игра не будет обнаружена как играемый на Сети.

Теперь определите местонахождение двух, если утверждения, которые отдают кнопки **Play и Instructions**:

```
        if(GUILayout.Button ("Play")){
          OpenLevel("Island Level");
        }
        if(GUILayout.Button ("Instructions")){
          OpenLevel("Instructions");
        }
```

Разместите их в, если часть утверждения, которое мы только добавили, так, чтобы у Вас было это:

```
    if      (Application.platform == RuntimePlatform.OSXWebPlayer ||
      Application.platform == RuntimePlatform.WindowsWebPlayer){
        if(GUILayout.Button ("Play")){
          OpenLevel("Island Level");
        }
        if(GUILayout.Button ("Instructions")){
          OpenLevel("Instructions");
        }
    }
    else{
      if(GUILayout.Button ("Quit")){
        Application.Quit();
      }
    }
```

Теперь, Ваше начальное обнаружение платформы, если утверждение отдаст кнопки **Play и Instructions**, если это обнаружит быть управляемым онлайн, и кнопку **Quit**, если это не онлайн.

Но ждите! Мы нуждаемся в кнопках **Play и Instructions** для автономного построить слишком правильный? Конечно - еще копируют/приклеивают код для кнопок **Play и Instructions** в утверждение также. Ваше заключительное утверждение обнаружения платформы должно быть похожим на это:

```
Здесь Выif       (Application.platform ==
  RuntimePlatform.OSXWebPlayer || Application.platform ==
  RuntimePlatform.WindowsWebPlayer){
  if(GUILayout.Button ("Play")){
    OpenLevel("Island Level");
  }
  if(GUILayout.Button ("Instructions")){
    OpenLevel("Instructions");
  }
}
else{
  if(GUILayout.Button ("Play")){
    OpenLevel("Island Level");
  }
  if(GUILayout.Button ("Instructions")){
    OpenLevel("Instructions");
  }
  if(GUILayout.Button ("Quit")){
    Application.Quit();
  }
}
```

можете видеть, что у нас еще есть только Игра и Инструкции в если условие и Игра, Инструкции, и Оставленный в. Пойдите в **Файл |, Экономят** в редакторе сценария, и затем переключаются назад на Unity. Это теперь автоматизировало проект только отдать кнопку **Quit**, если это не играется на Сети.

Теперь благодаря Прикладной автоматизации Classes мы осуществили. У Вас теперь есть ряд активов сцены, которые могут быть помещены в **Построить** список **Параметров настройки** наряду с **Уровнем Острова** и построены, поскольку **Игрок Игрока** или **Сети Сети Тек.**

Теперь давайте смотреть на построение и сеть и автономные версии игры.

Сжатие структуры и демонтаж отладки

Строя для любой платформы, у Вас будет два дополнительных параметров настройки у основания окна **Build Settings - Структуры** Компресса и **Символы Отладки Полосы.**

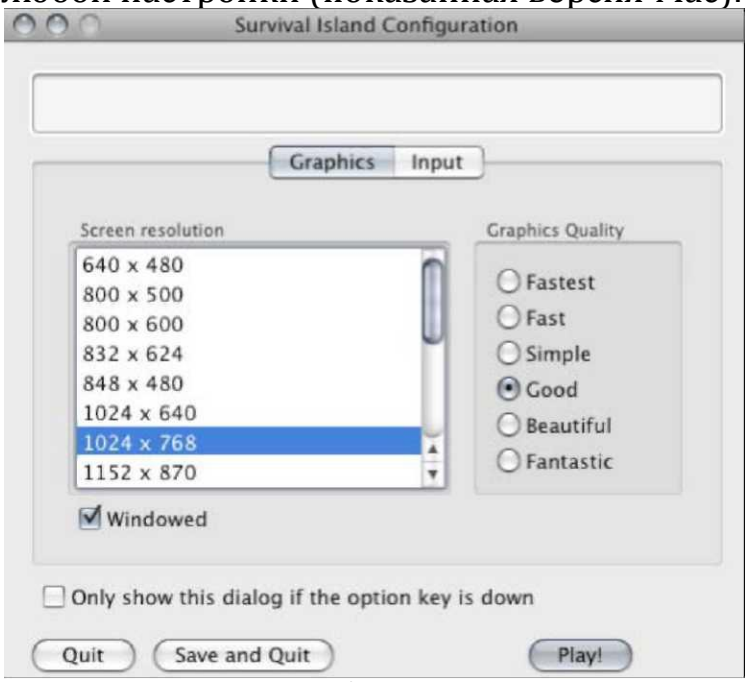
Помещенный просто, Unity обращается со сжатием активов структуры, используемых в Вашей игре для Вас основанный на их параметрах настройки импорта, и всем, что Вы должны сделать, гарантируют, что эта коробка отобрана, строя Вашу игру. Вы должны также выбрать коробку для **Символов Отладки Полосы** в большинстве случаев, когда Ваша игра достаточно полна, чтобы построить. Это заставляет Unity удалить любой нежелательный код Classes Отладки из Ваших сценариев, таких как

Отладка. Команда регистрации мы обсуждали в Главе 8.

Автономное здание

Автономный строит из Вашей игры запуском по умолчанию с пользователями разрешения страницы всплеска, чтобы настроить решение, при котором нужно играть в Вашу игру. Известный в Unity как **Диалог Решения,** это окно также позволяет Вашему игроку настраивать **Входные** параметры настройки и **Графическое Качество.** Как разработчик, Вы можете заклеймить это окно диалога изображением баннера.

Вот то, на что наше окно **Resolution Dialog** будет похоже без любой настройки (показанная версия Mac):



Большинство разработчиков предпочитает оставлять этот экран позволенным, поскольку он позволяет игроку определять их выбранное решение и уровни графического качества, которые удовлетворяют спецификации их компьютера. Однако, это может быть инвалид, если Вы хотели бы вынудить пользователей играть при специфическом решении.

Чтобы повредить это, Вы можете пойти, чтобы **Отредактировать | Проектные Параметры настройки | Игрок** и изменить урегулирование **Диалога Решения Показа**, чтобы **Повредить.** Наконец, Вы можете выбрать коробку для **Неплатежа, Полный Экран** здесь, чтобы вынудить Ваш компьютер играть в игру в fullscreen способе, если Вы не желаете, чтобы игра открылась в окне, которое является установкой по умолчанию. Примите во внимание, что игроки, которые получают, строят, где окно **Resolution Dialog** - инвалид, может вынудить это казаться, держа *клавишу ALT* (также известным как *Опция* на Mac), начиная строить - это может быть полезно, когда отсылка теста строит.

Проектировать баннер, графический, чтобы украсить верхнюю часть коробки Я
диалог решения, проект и сохранили файл изображения в Вашем проекте Я
измерьте 432x163 пикселы, и избранный это от **Баннера Диалога Решения** Я
опуститесь меню на экране **Player Settings**. Я

Теперь пойдите в **Файл |, Строят Параметры настройки,** и гарантируют, что Ваши три файла сцены перечислены в **Сценах, чтобы построить** область:

- **Menu.unity**
- **Остров Левел.унити**
- **Instructions.unity**

Если какие-нибудь сцены не появляются в списке, помните, что их можно тянуть я и понизились от **Проектной** группы до **Построить** списка **Параметров настройки**.

Важно, что **Меню** - первая сцена в списке, поскольку мы нуждаемся в этом, чтобы загрузить это сначала, так что удостоверьтесь, что это в позиции **0.**

Структуры Компресса отобраны по умолчанию, и это должно быть сохранено тем путем, поскольку он гарантирует, что Ваши структуры сжаты, чтобы загрузить игру более быстро. В первый раз Вы строите свою игру, может занять больше времени чем следующий, строит, поскольку структуры сжаты впервые.

Теперь просто выберите, какой формат Вы хотели бы построить для, или Windows, который создаст строить, чтобы воздействовать на Windows XP, Перспектива, и 7 +, или Mac - принимающий во внимание или PowerPC, Интел, или оба **(универсальный набор из двух предметов).**
Нажмите на кнопку **Build** у основания этого окна диалога, и Вы будете побуждены для местоположения спасти свою игру. Проведите к своему желательному местоположению и назовите свою построенную игру в **Том, чтобы сохранять Как** область. Тогда нажмите кнопку **Save**, чтобы подтвердить.

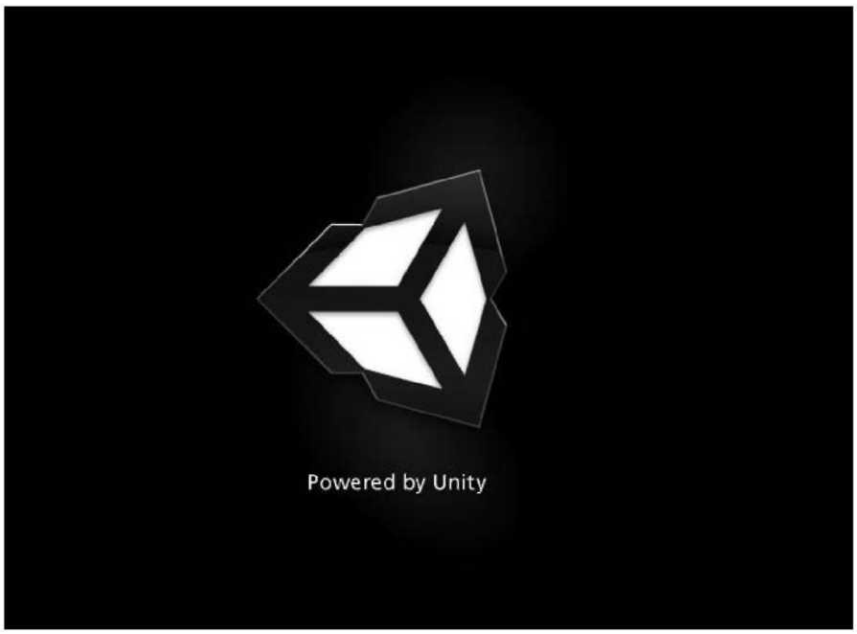
Ждите, в то время как Unity строит Вашу игру! Вам покажут бары продвижения, показывая активы, сжимаемые, сопровождаемые баром погрузки, поскольку каждый уровень добавлен к тому, чтобы строить. Как только здание полно, Ваша игра строят, будет открыт в окне операционной системы, чтобы показать Вам, что это готово.

На Мас, щелкните два раза применением, чтобы начать это, или на PC, открыть папку, содержащую игру и щелкнуть два раза. файл exe, чтобы начать Вашу игру. Примите во внимание, что, если Вы строите версию PC

на Мас или наоборот, они не могут быть проверены прирожденно.

Инди против Про

Строя Вашу игру с Инди-версией Unity, Вы должны знать, что Вашему игроку покажет **Приведенный в действие** экран всплеска **Unity** перед Вашими грузами игры. Эта отметка уровня воды не присутствует, строя с более дорогой Про версией Unity:



Другая Инди против Про различий, таких как нехватка динамических теней в Инди, может быть найдена в полном столе в следующем URL: <http://unity3d.com/unity/licenses.html>

Построение для Сети

В Unity открытом, **Построить Параметры настройки**, идя в **Файл | Строят Параметры настройки,** и выбирают**, Игрок Сети** от радио-списка кнопки типов, чтобы построить, и затем просто нажать **Строят.** Unity будет обращаться с соответствующими преобразованиями и сжатиями, необходимыми, чтобы создать строить, которое будет работать хорошо на Сети, и это - только одно из ее многого очарования!

Вы будете побуждены определить название и местоположение, чтобы сохранить файл. Так войдите в это и затем нажмите кнопку **Save** в основании, чтобы подтвердить.

Когда закончено, операционная система переключит к окну Ваш строила, спасен в, показывая Вам два файла, которые заставляют сеть построить работу - игра непосредственно (.unityweb дополнительный файл), и файл HTML, содержащий объемлющий код и JavaScript, требуемый загрузить

или вызвать загрузку плагина Игрока Сети Unity.

Чтобы играть в игру, откройте файл HTML в web-браузере Вашего выбора, и это начнет игру.

Приспосабливание игрока сети строит
Как Unity по умолчанию строят, предоставляет Вам необходимый код HTML/JavaScript, чтобы включать.unityweb файл в веб-страницу, должен Вы хотеть поместить это в Ваши собственные веб-страницы HTML/CSS, Вы просто должны будете взять код от <головы> и <тела> области страницы, чтобы поместить, это встраивает в Ваш собственный проект.

Сценарий обнаружения - <ГОЛОВА>
Откройте сопровождение build страницы HTML в редакторе сценария, который идет с Unity или Вашим привилегированным редактором HTML, например, Dreamweaver, SciTE, TextMate, и TextWrangler. Выберите весь код из <главной> части HTML - от вводного <сценарий> к закрытию </сценарий> признак, и скопируйте это к новому документу JavaScript, сохраняя это как UnityDetect.js.

Тогда Вы можете сохранить этот файл к той же самой папке как Ваши веб-страницы и включать этот JavaScript, вызывая это в <голова> Вашей собственной страницы со следующей линией HTML:

```
<script type="text/javascript" src="UnityDetect.js"></script>
```

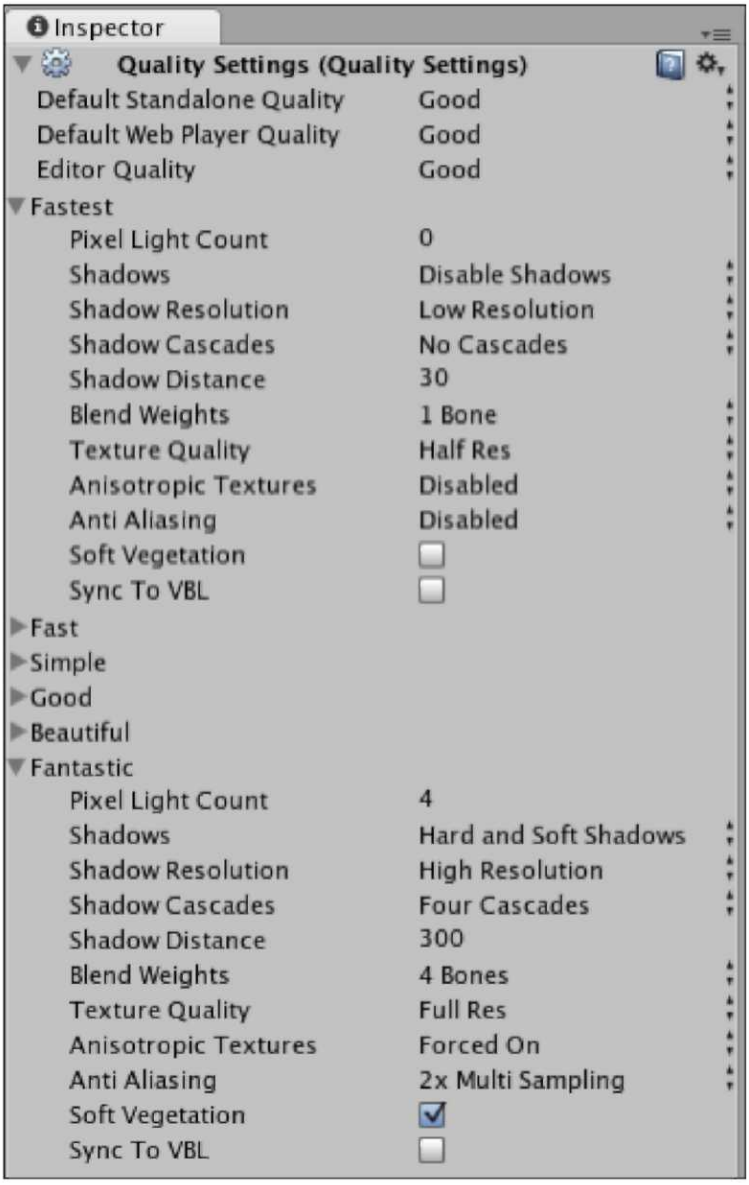
Объект включает - <ТЕЛО>
Тогда, возьмите вложение объекта непосредственно, копируя все в <тело> страницы от вводного <сценарий> признак, вниз к закрытию </noscript> признак. Возьмите этот кусок HTML и поместите это в свою собственную веб-страницу, чтобы вызвать игрока Unity.

Помните, что для этого, чтобы работать, этот код предполагает, что.unityweb файл находится в том же самом справочнике как HTML, который вызывает это.

Качественные Параметры настройки
Экспортируя от Unity, Вы не ограничены никакому отдельному уровню качества. Вы имеете большой контроль над качеством Вашей продукции, которая входит в форму **Качественных Параметров настройки.** Откройте это теперь в части **Инспектора** интерфейса, собираясь **Редактировать | Проектные Параметры настройки | Качество.**

Здесь Вы найдете, что способность установить Ваши различные три строит к одному из шести различных качеств, задает - **Самый быстрый, Быстрый, Простой, Хороший, Красивый,** и **Фантастический.** Вы можете тогда отредактировать, они задают себя, чтобы достигнуть точных результатов, поскольку Вы нуждаетесь к. Поскольку Вы только начинаете с Unity, давайте смотреть на противоположные концы масштаба,

сравниваясь **Самый быстрый** с **Фантастическим:**



Поскольку Вы можете видеть от предыдущего скриншота, параметры настройки весьма отличаются от каждого конца масштаба, столь давайте смотреть на то, что делают отдельные параметры настройки:

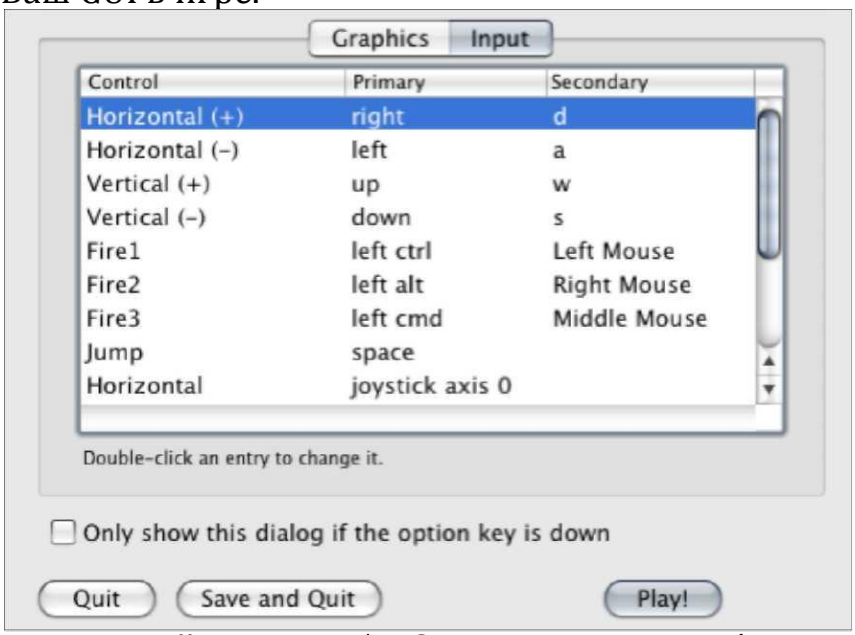
- **Свет Пиксела граф:** число огней пиксела, которые могут использоваться в Вашей сцене. Огни в Unity предоставлены как пиксел или вершина, пиксел, выглядящий лучшими, но являющийся более дорогим мудрые обработкой. С этим урегулированием Вы можете позволить определенное число огней пиксела, с остальными предоставляемыми как огни вершины. Это - то, почему у низкого конца масштаба**, Самого быстрого** заданный, есть **Свет Пиксела** набор **графа** к **0** по умолчанию.

- **Тени:** Эта особенность доступна только в Unity Про версия и позволяет Вам не определять динамические тени, твердые тени только, или твердые и мягкие тени (два уровня теневого качества).
- **Теневое Решение:** Снова это относится к Unity, Про только, и это урегулирование позволяет Вам выбирать качество, устанавливающее определенно для предоставляемых теней. Это может быть полезно, чтобы спасти работу, когда наличие многократных объектов с динамическими тенями в Вашем урегулировании сцены их к низкому решению могло означать различие между выключением их и хранением теней полностью во время оптимизации.
- **Теневые Каскады:** Про Unity может использовать в своих интересах **Каскадные Теневые Карты,** которые могут улучшить появление теней на направленных огнях в Вашей сцене. При рисунке той же самой теневой карты прогрессивно большие пространства, зависящие от более близкого к близости к камере игрока, получает больше теневых деталей пиксела карты, улучшая качество.
- **Теневое Расстояние:** Подобный оптимизации ограничения далекого самолета скрепки камеры, это - другой уровень деталей, tweaked (щипнуть). Это может использоваться, чтобы просто установить расстояние, после которого не предоставлены тени.
- **Веса Смеси:** Это урегулирование используется для манипулируемых характеров с очищенным от костей скелетом, и управляет числом весов (уровни) мультипликации, которая может быть смешана между. Технологии Unity рекомендуют две кости как хороший обмен между работой и появлением.
- **Качество Структуры:** Точно, как это звучит, количество, к которому Unity сожмет Ваши структуры.
- **Анизотропные Структуры: Анизотропное фильтрование** может помочь улучшить появление структур когда рассматривающийся под крутым углом, как холмы, но является дорогостоящим с точки зрения работы. Примите во внимание, что Вы можете также настроить это фильтрование на основе отдельной структуры в **Параметрах настройки Импорта** для активов.
- **Анти-Совмещение имен:** Это урегулирование смягчает края трехмерных элементов, делая Ваш взгляд игры намного лучше. Однако, как с другими фильтрами, это прибывает по стоимости работы.
- **Мягкая Растительность:** Это позволяет элементам ландшафта Unity, таким как растительность и деревья использовать **альфа-смешивание,** которое значительно улучшается, появление прозрачных областей структур имело обыкновение создавать растительность.
- **Синхронизация к VBL:** Это вынуждает Вашу игру быть синхронизированной к разряду освежительного напитка монитора игрока. Это вообще ухудшает работу, но избежит 'рваться' элементов в Вашей игре - появление некоаксиальности вершин, где структуры кажутся 'порванными' друг от друга.

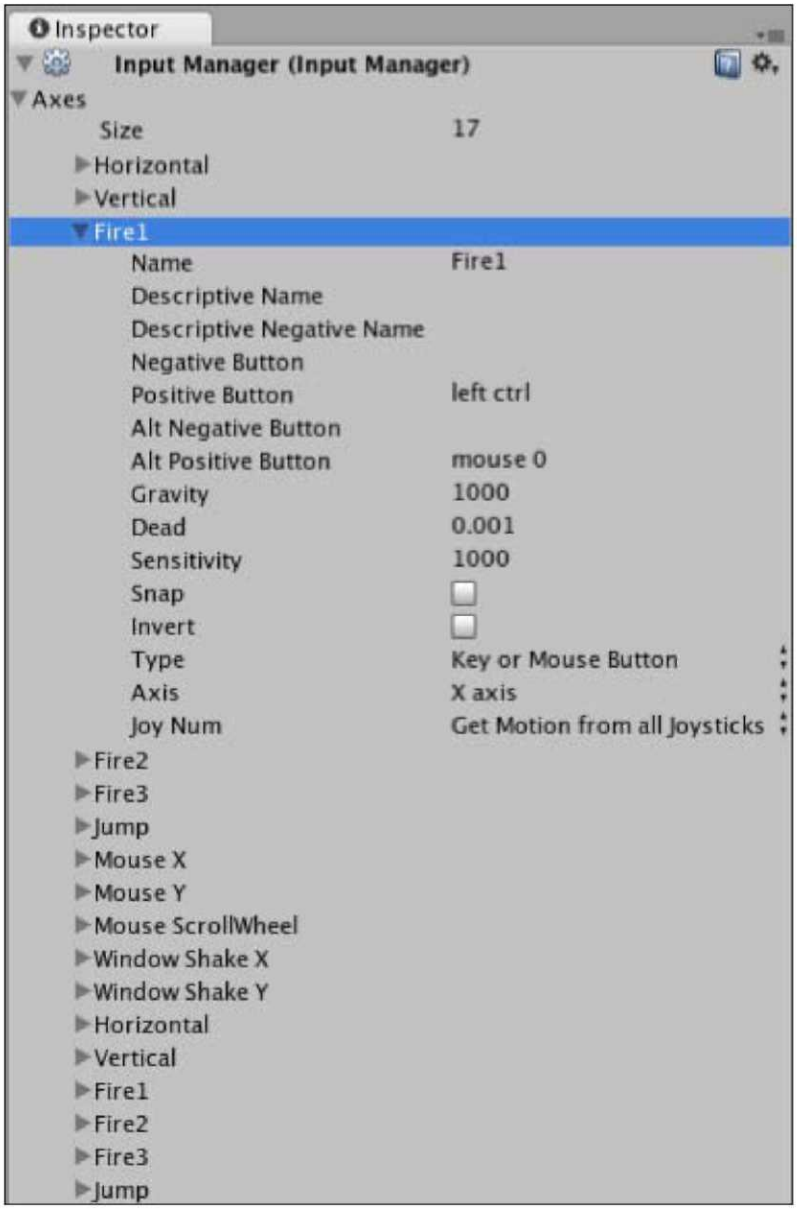
Вы должны использовать, они задают, чтобы установить варианты, которые принесут пользу игроку, поскольку у них будет способность выбрать из них в окне **Resolution Dialog** (см. *Строящую Автономную* секцию ранее), начиная Вашу игру как автономное, если у Вас нет инвалида это. Однако, довольно безопасно в большинстве случаев использовать собственный Unity, задает как гид, и просто tweaked (щипнуть) определенные параметры настройки, когда Вы нуждаетесь к.

Входные параметры настройки Игрока

В то время как окно **Resolution Dialog** дает автономное, строят игрока способность приспособить входные средства управления Вашей игры во **Входном** счете (см. следующий скриншот), важно знать, что Вы можете определить свои собственные неплатежи для контроля Вашей игры во **Входных параметрах настройки Игрока.** Это особенно полезно для сети, строит, поскольку у игрока нет никакой способности изменить параметры настройки контроля, когда они загружают игру. Поэтому, это лучше, что Вы настраиваете их заметно и предоставляете информацию игроку через Ваш GUI в игре.



В Unity, пойдите, чтобы **Отредактировать | Проектные Параметры настройки | Вход**, чтобы открыть входные параметры настройки в части **Инспектора** интерфейса. Вам тогда подарят существующие топоры контроля в Unity. Ценность **Размера** просто сообщает, сколько средств управления существует. Увеличивая эту ценность, Вы можете построить в своих собственных средствах управления, или альтернативно Вы можете просто расширить любой из существующих, нажимая на серую стрелку налево от их имени и регулируя ценности там.



Вы можете видеть, как это сходится с кодом, который мы написали ранее; смотря
сценарий CoconutThrow мы написали:

если (Вход. GetButtonUp ("Fire1")

Здесь, на ось Fire1 ссылается ее имя. Изменяя параметр **Названия** во входных параметрах настройки, Вы можете определить что потребности быть написанными в scripting. Для получения дополнительной информации о ключах Вы можете связать с в этих параметрах настройки, обратиться к **Входной** странице руководства Unity в следующем адресе:

<http://unity3d.com/support/documentation/Manual/Input.html>

Разделение Вашей работы

В дополнение к размещению Вашего игрока сети встраивают в Ваш собственный вебсайт, есть также несколько независимых участков двери игры, доступных, которые действуют как сообщество для разработчиков, разделяющих их работу.

Вот некоторые рекомендуемые участки, которые Вы должны посетить, как только Вы готовы поделиться своей работой с сообществом онлайн:

- www.shockwave.com <http://www.shockwave.com>
- www.wooglie.com <http://www.wooglie.com>
- www.blurst.com <http://www.blurst.com>
- www.tigsource.com <http://www.tigsource.com> (источник Игр "Индепендент")
- www.thegameslist.com <http://www.thegameslist.com>
- <http://форум unity3d.com> (область Витрины)
- www.todaysfreegame.com <http://www.todaysfreegame.com>
- www.learnUnity3d.com <http://www.learnUnity3d.com>

Важно, что Вы поделились своей работой с другими не только, чтобы хвастаться Вашими навыками развития но также и получить обратную связь на Вашей игре и позволить членам общественности без предшествующего знания Вашего проекта проверить, как это работает.

Этот вид непредубежденной обратной связи крайне важен, поскольку это позволяет Вам полоть ошибки и расследовать неинтуитивные части Вашей игры, которая может иметь смысл Вам, но сбить с толку обычного игрока.

Кроме того, знайте, что некоторые участки не можете быть в состоянии принять свою игру в формате.unityweb, но будет желать связаться с Вашим собственным блогом с включенной игрой или принять автономную версию для загрузки.

Резюме

В этой главе мы смотрели на то, как Вы можете экспортировать свою игру в Сеть и как автономный проект. В заключении мы оглянемся назад на то, что Вы изучили в течение этой книги и предлагаете пути, которыми Вы можете прогрессировать далее с существующими навыками, которые Вы развили и где искать длительную помощь с Вашим развитием Unity.

11

Тестирование и Дальнейшее Исследование

В течение этой книги мы затронули существенные темы, чтобы начать Вас в развитии с двигателем игры Unity. В работе с Unity Вы обнаружите, что с каждым новым элементом игры Вы развиваетесь, новые авеню возможности открываются в Вашем знании. Новые идеи и понятия игры прибудут более легко, поскольку Вы добавляете далее scripting знание к своему skillset. В этой главе мы завершим Ваше введение в Unity, смотря:

- Подходы к тестированию и завершению Вашей работы
- Измерение разрядов структуры от испытательных пользователей
- Где пойти для помощи с Unity и что учиться затем

С этим в памяти, смотря вперед туда, где продолжить расширять Ваши навыки, Вы должны занять время, чтобы расширить Ваше знание следующих областей:

- Scripting
- Scripting
- Scripting

Правильно, это не шутка - в то время как Unity гордится обеспечением интуитивного комплекта инструментов для того, чтобы развиться в визуальной манере и использовать GUI Редактора, чтобы построить сцены и объекты игры, нет никакой замены для того, чтобы изучить classes и команды, которые составляют двигатель Unity непосредственно. Прочитывая руководство Unity, сопровождаемое Составляющей Ссылкой и Сценарием, Доступным для ссылки и онлайн и как часть Вашей установки программного обеспечения Unity - Вы начнете понимать, как лучше всего создать все типы элементов игры, которые, возможно, не относятся к Вашему текущему проекту, но должны изложить в деталях Ваше понимание, чтобы помочь Вам работать более эффективно в долгосрочной перспективе:

Составляющая Ссылка (http://www.unity3d.com/support/Документация/Компоненты/)

Ссылка Scripting (<http://www.unity3d.com/support/documentation/> ScriptReference/)

- Руководство Unity (<http://www.unity3d.com/support/documentation/Manual/>)

Тестирование и завершение
Рассматривая развитие игры, Вы должны очень знать о важности тестирования Вашей игры среди пользователей, у которых нет никаких предвзятых мнений этого вообще. Воздействуя на любой творческий проект, Вы должны знать, что, чтобы поддержать творческую объективность, Вы должны быть открытыми для критики и что тестирование - столько же часть, которой поскольку это - техническая потребность. Слишком легко привыкнуть к рассказу Вашей игры или механике, и часто неспособный видеть "лес для деревьев" с точки зрения того, как игрок ответит на это.

Общественное тестирование
То, когда смотрящий на тест Ваша игра, попробуйте и пошлите тест, строит к диапазону пользователей, которые могут предоставить испытательную обратную связь Вам со следующими изменениями:

- Компьютерная спецификация: Гарантируйте, что Вы проверяете в ряде по-другому приведенных в действие машин, и получаете обратную связь на работе
- Формат: Попытайтесь послать строить и за Мас и за PC где только возможно
- Язык: Ваши испытательные пользователи все говорят на том же самом языке как Вы? Они могут сказать Вам, если Вы объясняете элементы игры в интерфейсах?

Вручая Ваш игра закончена коллекции общественных тестеров, Вы вручаете им, что упоминается как **эксплуатационное испытание** Вашей игры - **альфа**, являющаяся испытательной версией Вы и другой тест разработчиков. Формализуя процесс, Вы можете сделать обратную связь, которую Вы получаете о своей игре как полезная насколько-возможно-ничья анкетный опрос, который излагает те же самые вопросы всем тестерам, спрашивая не только вопросы об их ответах на игру, но также и информация о них как игрок. Таким образом, Вы можете начать делать утверждения о своей игре, такие как:

"Игроки в возрасте 18 - 24 любили механика и поняли игру, но игроки 45 + не понимали это, не читая инструкции."

в дополнение к технической информации, такой как:

"Игроки с компьютерами под 2.4ghz обработка скорости нашли, что

игра ответила вяло."

Обратная связь разряда структуры

Чтобы предоставить тестерам Вашей игры с помощью обеспечения определенной обратной связи на технических особенностях, таких как разряд структуры (скорость, на которой структуры игры оттянуты во время игры), Вы можете обеспечить, Ваш тест строят с элементом GUI, говоря им текущий разряд структуры.

Чтобы добавить это к любой сцене, давайте смотреть на практический пример. Откройте сцену, Вы желаете добавить, что оверлей экрана разряда структуры к, и создать новый объект Text GUI, чтобы показать идти информация в **GameObject | Создает Другой | Текст GUI.** Переименуйте этот объект FPS **displayer,** и затем в **Текстовом** компоненте **GUI Инспектора,** заставьте **Якорь** в **верхний центр,** и **Выравнивание сосредотачиваться.**

Теперь создайте новый сценарий в своем **Проекте**, выбирая папку, в которой Вы хотели бы создать его, и затем щелкать **Создают,** и выбирают **Javascript** из опускаться меню. Переименуйте свой сценарий **FPSdisplay,** затем щелкните два раза его изображением, чтобы начать это в редакторе сценария.

Поскольку структура оценивает Ваши пробеги игры в, Variables в зависимости от аппаратных средств и конфигурации программного обеспечения, мы должны выполнить сумму, которая принимает во внимание, сколько структур предоставлялось в пределах временных рамок игры каждую секунду. Мы начнем, устанавливая следующие variables наверху сценария:

```
private   var updatePeriod = 0.5;
private   var nextUpdate    :    float = 0;
private   var frames : float = 0;
private   var fps : float = 0;
```

Мы устанавливаем эти четыре variables здесь по следующим причинам:

- updatePeriod: как часто в секундах мы хотели бы, чтобы текст GUI обновил, и поэтому период, в который мы пробуем количество предоставленных структур. Мы установили это в 0.5, чтобы показать, что новое чтение каждой половины секунды, облегчающей для испытательного пользователя читать - устанавливающий эту ценность немного, понижается, закончился бы в числах, обновляющих слишком часто, чтобы читать легко. Устанавливая любого выше например, каждый секунда результат в менее точном чтении, поскольку разряд структуры может измениться в пределах второго.
- nextUpdate: это - число, чтобы сохранить пункт вовремя, в котором мы должны проверить и обновить структуры в секунду.
- структуры: число увеличило каждую структуру, поэтому храня количество предоставленных структур.

- fps: число, чтобы сохранить структуры в секунду, которой назначит поток fps ценность, беря ценность Variables структур и деля это updatePeriod Variables.

Теперь, в Update () function, поместите следующий код:

```
frames++;
if(Time.time > nextUpdate){
        fps = Mathf.Round(frames / updatePeriod);
            guiText.text = "Frames Per Second:         " + fps;
            nextUpdate = Time.time + updatePeriod; frames = 0;
}
```

Здесь мы делаем следующее:

- Увеличьте Variables структур к 1 каждому разу, когда Update () function происходит (после того, как каждая структура).
- Ждите Time.time - прилавок в реальном времени, который рассчитывает от начала игры - чтобы достигнуть ценности вне ценности nextUpdate Variables. Поскольку nextUpdate начинает игру за ценность 0, это немедленно происходит, когда сцена этот сценарий находится в грузах.
- Это, если утверждение тогда заставляет fps Variables равняться округленной ценности счета структур, разделенных на updatePeriod, чтобы давать нам целое число сколько структур в секунду, а не половину секунды - эти 0.5 ценности updatePeriod.
- Мы обращаемся к текстовому параметру guiText компонента, и устанавливаем его в последовательность, говоря "Структуры В Секунду:", и добавьте число в fps Variables до конца.
- Кардинально, мы тогда устанавливаем nextupdate в текущую ценность времени, плюс updatePeriod, подразумевая это, если утверждение повторно вызовет в половину секундного времени.
- Наконец, мы перезагружаем счет структур так, чтобы осуществление выборки могло продолжиться от ноля.

Теперь, после закрытия Functions обновления, гарантируйте, что Вы включаете следующую линию, чтобы удостовериться, что Вы добавляете это к объекту с Текстовым компонентом GUI:

```
©script RequireComponent (GUIText)
```

Пойдите в **Файл |, Экономят** в редакторе сценария, и переключаются назад на Unity. Гарантируйте, что **FPS displayer** объект все еще отобран в **Иерархии** и затем идти в **Компонент | Сценарии | показ FPS.** Теперь проверьте свою сцену, и Вы будете видеть, что Текст GUI показывает разряд структуры наверху экрана.

Ваша игра выступит по-другому за пределами редактора Unity. Также,

чтения от этого FPS displayer должны только быть отмечены, как только игра построена и работает или как игрок сети или как автономная версия; попросите, чтобы Ваши испытательные пользователи отметили самые низкие и самые высокие разряды структуры, чтобы дать Вам диапазон чтений, чтобы рассмотреть, и если есть особенно требовательные части Вашей игры, например сложная мультипликация или эффекты частицы, то это могло стоить просить чтения с этих времен отдельно.

Поскольку игра выступает по-другому за пределами редактора Unity, ценности от этого показа разряда структуры будут отличаться от данных во встроенном счете **Stats** вида **Игры**:

Повышение работы

Улучшение работы Вашей игры в результате тестирования является легко всей областью исследования сам по себе. Однако, чтобы улучшить Ваше будущее развитие, Вы должны будете знать об основной экономии следующими способами:

- Счет многоугольника: вводя трехмерные модели, они должны быть разработаны с низким счетом многоугольника в памяти. Так попробуйте и упростите свои модели в максимально возможной степени, чтобы улучшить работу.
- Расстояние ничьей: Полагайте, что сокращение расстояния Вашего Далекого Самолета Скрепки в Ваших камерах сокращает количество пейзажа, который должна отдать игра.
- Размеры структуры: Включая более высокое решение структуры могут улучшить визуальную ясность Вашей игры, но они также делают машинную работу тяжелее. Так попробуйте и уменьшите размеры структуры в максимально возможной степени, используя и Ваше программное обеспечение редактирования изображение и используя параметры настройки Импорта Ваших активов структуры.

Сценарий эффективно: Как есть много подходов к отличающимся решениям в scripting, пробуют и находят более эффективные способы написать Ваши сценарии. Начало, читая Unity ведет к эффективному scripting онлайн (<http://unity3d>. com/support/documentation/ScriptReference/index.Performance_ Optimization.html)

Наблюдайте представление Оптимизации Работы Джоакимом Антом, победите программиста в Технологиях Unity (<http://unity3d.com/support/> resources/unite-presentations/performance-optimization)

Подходы к изучению

Поскольку Вы прогрессируете из этой книги, Вы должны будете развить подход, чтобы далее учиться, который сохраняет равновесие между личной настойчивостью и потребностью попросить помощь от более опытных разработчиков Unity. Последуйте совету, вынутому ниже, и Вы должны быть хорошо на Вашем пути к помощи другим членам сообщества, поскольку Вы расширяете свое знание.

Покрытие так много оснований насколько возможно

Изучая любой новый пакет программ / язык программирования, это часто имеет место, что Вы работаете к крайнему сроку, быть этим как частью Вашей работы или как внештатный сотрудник. Это может часто приводить, "берут только, к чему Вы должны" приблизиться к изучению. В то время как это может часто быть потребностью из-за рабочих требований, это может часто быть вредно для Вашего изучения, поскольку Вы можете развить плохие привычки, которые остаются с Вами в течение Вашего времени, работая с программным обеспечением - в конечном счете приведение к неэффективным подходам.

Принимая это, я рекомендую, чтобы Вы заняли время, чтобы прочитать официальную документацию всякий раз, когда Вы можете - даже если Вы застреваете на определенной проблеме развития, она может часто помогать отвлечься далеко от того, с чем Вы застреваете. Пойдите и читайте на несвязанном сценарии, и возвратитесь с новой перспективой.

Если Вы не знаете, только спрашиваете!

Другой полезный подход к изучению должен, конечно, смотреть на то, как другие приближаются к каждому новому элементу игры, который Вы пытаетесь создать. В развитии игры то, что Вы обнаружите, - то, что часто есть много подходов к той же самой проблеме как, которую мы изучили в Главе 5, заставляя наш характер открыть дверь заставы. В результате часто заманчиво переработать изученное решение навыков предыдущей проблемы - но я всегда рекомендую перепроверить тот Ваш подход, самый

эффективный путь.

Спрашивая на форуме Unity или на IRC (интернет-Чат Реле) канал, Вы будете в состоянии получить согласие по самому эффективному способу выполнить задачу развития - и чаще чем не, обнаружить, что способ, которым Вы сначала думали о приближении к Вашей проблеме, был более сложным, чем это должно было быть!

Задавая вопросы в любом из ранее упомянутых мест, всегда не забывайте включать следующие пункты когда бы ни было возможно:

- Чего Вы пытаетесь достигнуть?
- То, что делает Вы думаете, является правильным подходом?
- Что Вы попробовали пока?

Это даст другим лучший выстрел в помощь Вам - давая так много информации насколько возможно, даже если Вы будете думать, что это, возможно, не относится к делу, то Вы дадите себе лучший шанс достижения Вашего развития.
Большая вещь о сообществе Unity - то, что оно поощряет учиться примером. У третьего лица основанный на Wiki участок (www.unifycommunity.com/wiki <http://www.unifycommunity.com/wiki>) есть широкие располагающиеся примеры всего от сценариев до плагинов, к обучающим программам, и больше. Там Вы найдете, что полезные свободные к использованию кодовые отрывки добавляют примеры подготовленных элементов в пределах ссылки сценария и даже информации относительно того, как осуществить это с примером загружаемые проекты.

Резюме

В этой главе мы обсудили способы, которыми Вы должны идти дальше из этой книги, и как Вы можете собрать информацию от испытательных пользователей, чтобы улучшить Вашу игру.

Все, что остается, должно желать Вам лучшую из удачи с Вашим будущим развитием игры в Unity. От меня непосредственно и всех связанных с этой книгой, благодарит читать, и я надеюсь, что Вы наслаждались изучением-это - только что начало!