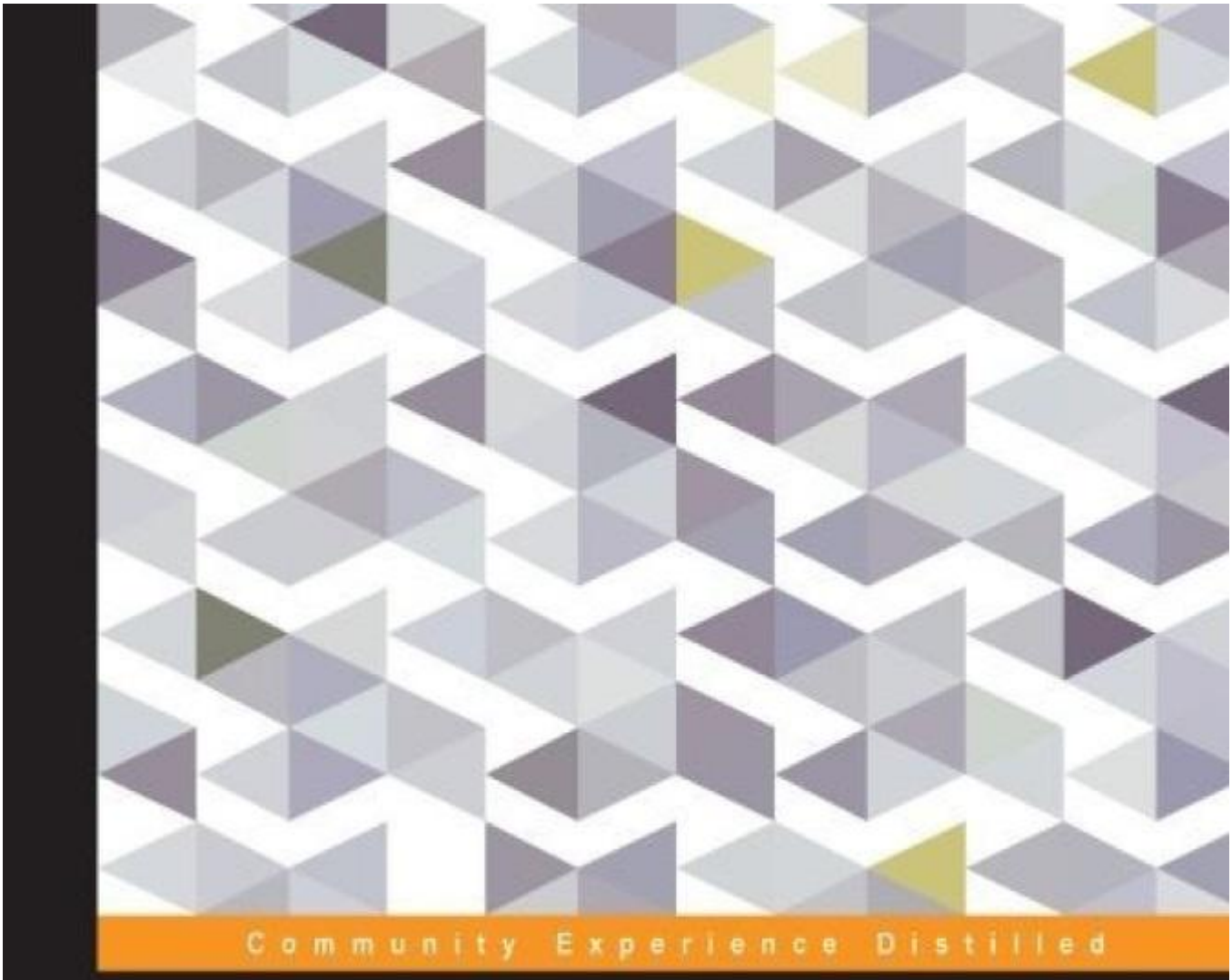


Изучаем C++ создавая игры в UE4

Изучайте программирование C++ с интересным применением реально мира, что позволит вам создавать ваши собственные игры!

Уилльям Шериф

[PACKT] PUBLISHING

Изучаем C++ создавая игры в UE4

Оглавление

Изучаем C++ создавая игры в UE4 1

Предисловие 10

Глава 1. Написание кода с C++ 17

Установка нашего проекта 17

Используем Microsoft Visual C++ на Windows 17

Используем XCode на Mac 22

Создание вашей первой программы C++ 25

Точка с запятой 29

Исправление ошибок 29

Предупреждения 30

Что такое построение и компиляция? 30

Скрипт 31

Выводы 32

Глава 2. Переменные и память 33

Переменные 34

Объявление переменных – затрагивание кремния 34

Числа это всё 36

Больше о переменных 37

Математика в C++ 39

Обобщённый синтаксис переменных 41

Примитивные типы 41

Типы объектов 42

Указатели 45

Что могут делать указатели? 46

Адрес оператора & 47

cin 49

printf() 49

Выводы 51

Глава 3. If, Else и Switch 52

Разветвление 52

Контролирование выполнения вашей программы 53

Оператор равенства == 53

Пишем код утверждений с if 54

Пишем код с оператором Else 55

Проверка неравенств с применением других операторов сравнения (>, >=, <, <=, и !=) 56

Использование логических операторов 57
Оператор Не (!) 57
Оператор И (&&) 59
Оператор Или (||) 59
Наш первый пример с Unreal Engine 60
Упражнение 65
Решение 65
Разветвление кода более чем в двух направлениях 66
Утверждения с else if 66
Оператор switch 68
Выводы 74
Глава 4. Циклы 75
Цикл while 75
Бесконечные циклы 77
Упражнения 78
Решения 78
Цикл do/while 79
Цикл for 80
Упражнения 81
Решения 81
Выполнение цикла с Unreal Engine 82
Выводы 84
Глава 5.Функции и Макросы 85
Функции 85
Пример функции – sqrt() библиотеки <cmath> 86
Написание нашей собственной функции 88
Образец трассировки программы 89
Упражнение 91
Решение 91
Функции с аргументами 91
Функции возвращающие значения 92
Упражнения 93
Решения 94
Переменные, пересмотр 95
Глобальные переменные 95
Локальные переменные 96

Область действия переменной 96
Статические локальные переменные 98
Const переменные 98
Прототипы функций 99
prototypes.h содержит 100
funcs.cpp содержит 101
main.cpp содержит 101
Переменные extern 102
Макросы 102
Совет – попробуйте применить переменные const там, где это возможно 103
Макросы с аргументами 103
Совет – примените встроенные функции вместо макросов с аргументами 104
Выводы 105
Глава 6. Объекты, Классы и Наследование 106
Объекты struct 107
Функция-член 107
Строковые типы являются объектами? 108
Запуск функции-члена 109
Private и инкапсуляция 111
Некоторым программистам нравится public 113
Класс против struct 113
Геттеры и сеттеры 114
Геттеры 114
Сеттеры 115
Но в чём же всё-таки идея операций get/set? 116
Конструкторы и деструкторы 117
Наследование класса 118
Производные классы 118
Отношение “это” 122
Защищённые переменные 124
Виртуальные функции 124
Чисто виртуальные функции (и абстрактные классы) 124
Множественное наследование 125
Частное наследование 126
Помещаем ваш класс в заголовочный файл 126
.h и .cpp 129

Упражнение 130
Выводы 130
Глава 7. Динамическое распределение памяти 131
Динамическое распределение памяти 132
Ключевое слово delete 132
Утечка памяти 133
Обычные массивы 134
Синтаксис массивов 135
Упражнения 136
Решения 136
C++ стиль массивов динамического размера (new[] и delete[]) 136
Массивы динамического C-стиля 137
Выводы 139
Глава 8. Действующие лица и пешки 140
Actor против pawn 140
Создание мира для размещения в нём ваших акторов 140
Редактор UE4 144
Управление редактором 144
Управление режимом игры 144
Добавление объектов в сцену 145
Начинаем с нуля 147
Добавление источников света 149
Объёмы столкновения 150
Добавление актора в сцену 152
Создание сущности игрока 152
Наследование от класса GameFramework в UE4 152
Загружаем сетку 156
Написание C++ кода контролирующего игрового персонажа 164
Делаем игрока экземпляром класса Avatar 164
Устанавливаем ввод управления 166
Рыскание и тангаж 170
Создание объекта неигрового персонажа 171
Отображение цитат из каждого диалогового окна NPC 175
Отображение сообщений в HUD 175
Применение TArray<Message> 178
Срабатывание события по приближению к NPC 180

Выводы.....184
Глава 9. Шаблоны и обычно используемые контейнеры......185
Отладка выходных данных в UE4.....185
TArray<T> в UE4.....186
Пример использования TArray<T>.....186
Итерация TArray.....188
Выясняем находится ли элемент в TArray.....190
TSet<T>.....190
Итерация TSet.....191
Пересечение TSet.....191
Объединение TSet.....191
Нахождение TSet.....192
TMap<T, S>.....192
Список предметов для инвентаря игрока.....192
Итерация TMap.....193
STL C++ версии часто используемых контейнеров.....194
Набор STL C++.....195
Нахождение элемента в <set>.....196
Карта STL C++.....197
Выводы.....199
Глава 10. Система Инвентаризации и Подбор Предметов.....200
Объявляем рюкзак.....200
Предварительное объявление.....200
Импортирование ассетов.....202
Прикрепляем действия карты к клавише.....206
Базовый класс PickupItem.....208
Корневой компонент.....210
Изображаем инвентарь игрока.....213
Используем HUD::DrawTexture().....214
Определение щелчка в инвентарной системе.....217
Выводы.....220
Глава 11. Монстры.....221
Пейзаж.....221
Скульптурирование пейзажа.....224
Монстры.....226
Базовый интеллект монстров.....230

Монстр нападает на игрока....237
Рукопашная атака....238
Сокеты....244
Снаряды или дальняя атака....261
Отбрасывание игрока....268
Выводы....269
Глава 12. Книга заклинаний....270
Система частиц....271
Изменение свойств частиц....274
Настройки для заклинания метель....276
Актор класса заклинания....283
Блупринт для наших заклинаний....286
Выбор заклинаний....287
Прикрепляем правый клик к посыланию заклинания....290
Написание функции CastSpell аватара....291
Написание AMyHUD::MouseRightClicked()....292
Создание других заклинаний....295
Огненное заклинание....296
Упражнения....297
Выводы....297

Изучаем C++ создавая игры в UE4

Об авторе

Уильям Шериф является программистом C++ с опытом программирования более восьми лет. Он имеет обширный опыт в мире программирования, от игрового программирования до веб программирования. Он также работал сессионным преподавателем курсов в университете, в течении семи лет.

Он выпустил несколько приложений в магазине iTunes, включая симулятор игры на гитаре и MARSHALL OF THE ELITE SQUADRON.

В прошлом он получил признание благодаря лёгкой для понимания манере изложения курса.

Предисловие

Итак, вы хотите программировать ваши собственные игры, используя Unreal Engine 4 (UE4). И у вас есть великое множество причин на это:

- UE4 мощный: UE4 предоставляет самый передовой уровень искусства, красоту, реалистичное освещение и физические эффекты, и всего что применяемо AAA Студиями.
- UE4 приспособлен для работы на всех устройствах: код написанный для UE4 будет работать на стационарных компьютерах как на Windows, так и для Mac, и на устройствах как на Android, так и на iOS (всё это в момент написания этой книги, а в будущем будет поддерживаться ещё больше устройств).

Так что вы можете использовать UE4, чтобы сразу писать главные части своей игры, и после этого беспрепятственно выкладывать на торговые площадки iOS и Android. (Конечно, будет пара моментов: приложения iOS и Android должны быть написаны отдельно.)

Что же такое игровой движок?

Игровой движок аналогичен двигателю машины. Игровой движок - это то, что приводит в действие игру. Вы будете говорить движку, чего вы хотите и (используя код C++ и редактор UE4) движок будет ответственен за то, чтобы на самом деле заставить всё это происходить.

Вы будете строить свою игру вокруг игрового движка UE4, сопоставимо с тем как кузов и колёса выстроены вокруг настоящего автомобильного двигателя. Когда вы отгружаете игру с UE4, вы основательно приспосабливаете движок UE4 и подгоняете под свои собственные составляющие игры, такие как графика, звук и код.

Во что мне обойдётся использование UE4?

Ответ вкратце, 19$ и 5 процентов от продаж.

“Что?” спросите вы. 19$?

Верно. Всего лишь за 19$, вы получаете полный доступ к AAA Движку мирового класса, в комплекте с исходниками. Это потрясающая сделка, особенно учитывая факт того, что другие движки могут стоить где-то от 500$ до 1000$ за одну лишь только лицензию.

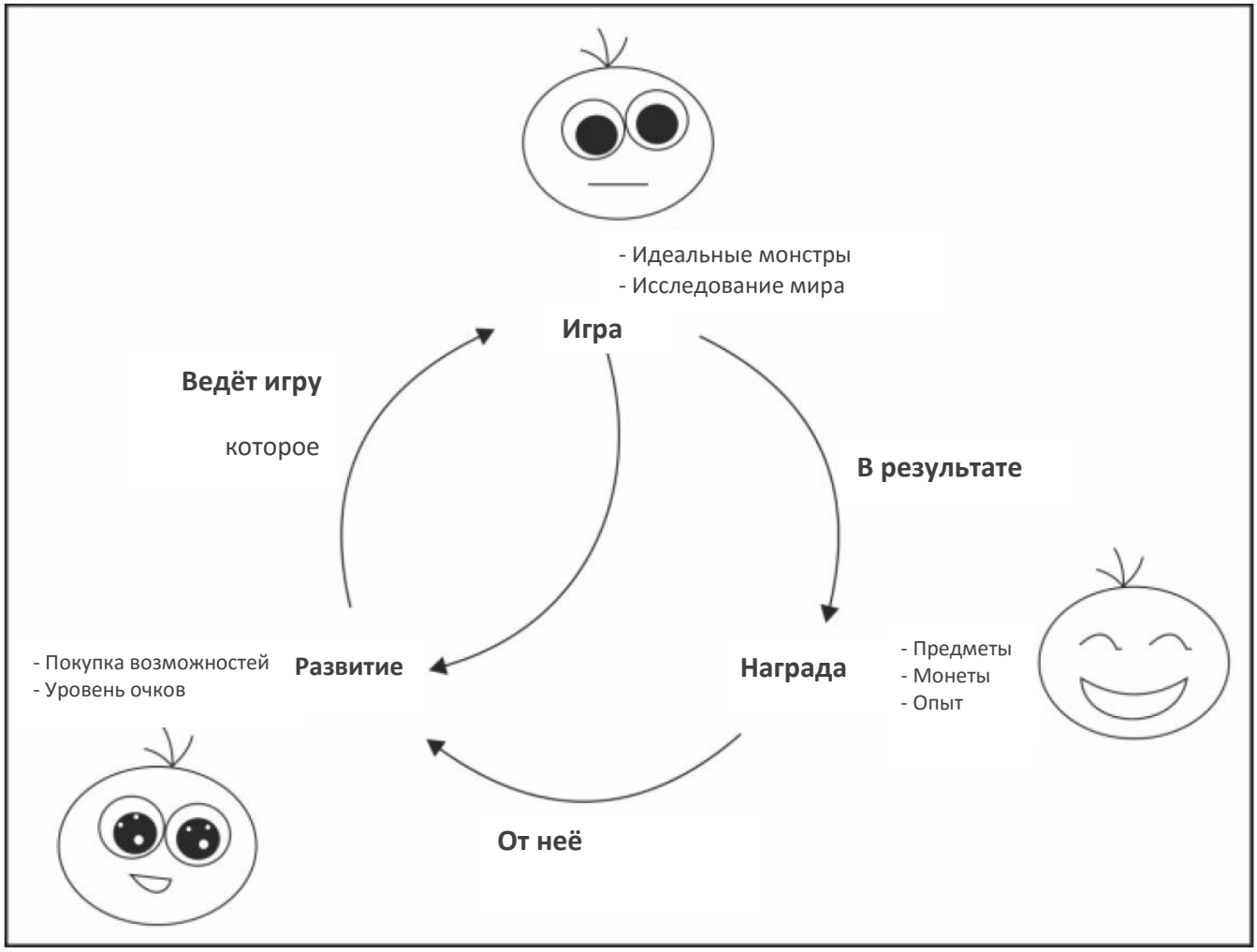
Почему я просто не спрограммирую свой собственный движок и не сохраню 5%?

Поверьте мне, если вы хотите создавать игры в пределах приемлемых временных рамок и у вас нет большой команды специализированных по движку программистов, чтобы помочь вам, то вы захотите сфокусировать свои усилия на том, что вы продаёте (на ваших играх).

Не обязывая фокусироваться на программировании, игровой движок даёт вам свободу думать только о том, как делать саму игру. И если вам не нужно содержать в исправности ваш собственный движок и отлаживать в нём баги, то попутно вы избавляетесь от большой загруженности на свою голову.

Обзор игры – цикл Игра-Награда-Рост

Я хочу показать вам эту диаграмму сейчас, потому что она содержит основную концепцию, которую многие новоиспечённые разработчики упускают во время написания своей первой игры. Игра может быть наполнена звуковыми эффектами, графикой, реалистичной физикой и всё ещё не ощущаться игрой. Почему так?



Начиная с верхушки цикла, в ходе действий Игры (таких как одержать победу над монстром) складывается результат награды для игрока (такой как золото или опыт). Эти награды в свою очередь могут быть использованы для Развития внутри игры (как например повышение статистики либо открытие новых миров для исследования). Это Развитие затем ведёт ход игры по новому интересному пути. На пример новое оружие может изменить базовую механику ведения схватки, новые заклинания позволяют вам бросать вызов группам монстров с абсолютно другим подходом, или новые режимы перемещения могут позволить вам достигать областей ранее не доступных.

Это основной базовый цикл, который создаёт интересный ход игры. Ключ в том, что ход Игры должен складывать результат какого-то рода Вознаграждения – подумайте о блестящих золотых кусочках, вылетающих из мерзкого злодея. Для награды должны быть очки, которые должны служить результатом какого-то рода Развития в ходе игры. Подумайте о том, как много новых мест было открыто круговым ударом в *The Legend of Zelda*.

Игра, в которую можно лишь просто играть (без Вознаграждения и Развития) не будет ощущаться игрой, на самом деле она будет ощущаться лишь базовым прототипом игры. К примеру, представьте симулятор полёта, в котором просто открытый мир и никаких целей и никаких стремлений. Так же никакой возможности прокачивать ваш самолёт или оружие. Это не будет прямо таки игрой.

Игра, в которую можно играть и лишь получать Вознаграждение (без Развития), будет ощущаться примитивной и простой. Награда не будет приносить игроку удовольствия, если её нельзя будет ни для чего использовать.

Игра, в которую можно играть и Развиваться (без Вознаграждения), будет просто казаться бессмысленным возрастанием сложности, и не будет давать игроку чувства наслаждения от его достижений.

Игра со всеми трёмя элементами будет держать игрока вовлечённым в увлекательный ход Игры. Ход Игры имеет вознаграждающий результат (обретение трофеев и последовательность истории), которые влияют на результат Развития мира игры. Держать этот цикл в уме, когда продумываете свою игру, действительно поможет вам спроектировать полноценную игру.

Полезный совет

Прототип игры это подтверждение концепции игры. Скажем, вы хотите создать вашу собственную, уникальную версию игры *Blackjack*. Первое что вы должны сделать, написать программу прототипа, чтобы показать каким образом будет идти ход игры.

Монетизация

Кое о чём вам необходимо подумать заранее во время разработки вашей игры. О стратегии монетизации. Как ваша игра будет делать деньги? Если вы пытаетесь основать компанию, то вам уже на раннем этапе следует подумать о том, каковы будут ваши источники дохода.

Будете ли вы пробовать делать деньги на покупочной цене, как *Jamestown, The Banner Saga, Castle Crashers* или *Crypt of the Necrodancer*? Либо вы сфокусируетесь на распространении бесплатной игры с вложенными покупками, как *Clash of Clans, Candy Crush Saga*, или *Subway Surfers*?

Класс игр для мобильных устройств (на пример игры на построение на iOS) делает много денег, позволяя пользователю пропускать порядок хода Игры и

перепрыгивать прямо к частям цикла с награждениями и Развитием. Преимущество в том чтобы так делать очень большое. Многие люди тратят сотни долларов на одной игре.

Почему C++

UE4 спрограммировано на C++. И чтобы писать код для UE4, вы должны знать C++.

C++ это общий выбор программистов игр, потому что он предлагает очень хорошее исполнение, комбинированное с элементами объектно-ориентированного программирования. Это очень мощный и гибкий язык.

Что охватывает эта книга

Глава 1. *Написание кода с C++*, поговорим о запуске и дальнейшей работе вашей первой программы на C++.

Глава 2. *Переменные и Память*, поговорим о том, как создавать, читать и писать переменные в компьютерной памяти.

Глава 3. *If, Else,* и *Switch*, поговорим о разветвлении кода: что позволяет выполнять различные секции кода, в зависимости от условий программы.

Глава 4. *Циклирование*, обсудим то, как мы повторяем определённые секции кода, столько раз, сколько нужно.

Глава 5. *Функции и Макросы*, поговорим о функциях, которые являются связками кода и могут быть вызваны любое количество раз, в зависимости от ваших пожеланий.

Глава 6. *Объекты, Классы и Наследование*, поговорим об определениях классов и конкретизация некоторых объектов основанных на определении классов.

Глава 7. *Динамическое распределение памяти*, обсудим объекты, распределённые на «куче», а так же низкоуровневые стили массивов C и C++.

Глава 8. *Действующие лица и Пешки*, первая глава, где мы уже как следует, поработаем в коде UE4. Мы начнём с создания игрового мира, чтобы поместить персонажей в него. И сделаем происхождение класса Аватар от специально выполненного действующего лица.

Глава 9. *Шаблоны и Обычно Используемые Контейнеры*, исследуем семейство собрания данных UE4 и C++ STL называемых контейнеры. Зачастую проблема программирования может быть упрощена во много раз при выборе правильного типа контейнера.

Глава 10. *Система Инвентаризации и Подбор Предметов*, обсудим создание системы инвентаризации с возможностью подбирания новых предметов.

Глава 11. *Монстры*, научимся создавать монстров, которые преследуют игрока и атакуют его, применяя оружие.

Глава 12. *Книга Заклинаний*, научимся создавать и насылать заклинания в нашей игре.

Что вам нужно для этой книги

Чтобы работать с этим текстом, вам понадобятся две программы. Первая ваша интегрированная среда разработки, ИСР. Вторая часть программного обеспечения это, конечно же, сам Unreal Engine.

Если вы используете Microsoft Windows, то вам понадобится Microsoft Visual Studio 2013 Express Edition для Windows Desktop. Если вы используете Mac, то вам понадобится Xcode. Скачать Unreal Engine вы можете отсюда https://www.unrealengine.com/.

Для кого эта книга

Эта книга для любого кто хочет писать приложения Unreal Engine. Текст книги начинает рассказывать вам, как компилировать и запускать ваше первое приложение Unreal Engine, последующие главы описывают правила языка программирования C++. После вступительных глав по C++, вы начнёте строить ваше собственное игровое приложение на C++.

Условные обозначения

В этой книге вы обнаружите несколько стилей текста, которые различаются в зависимости от рода информации. Здесь несколько примеров этих стилей и объяснение их значений.

Слова кода в тексте, названия таблиц баз данных, имена папок, имена файлов, расширения файлов, названия путей, URL, ввод пользователя, и имена в Twitter показаны следующим образом: «Тип переменной variableType будет говорить вам, какой тип данных мы собираемся хранить в нашей переменной. Имя переменной variableName это обозначение, которое мы будем использовать, чтобы считывать либо записывать эту часть памяти».

Блок кода будет изображён следующим образом:

```
struct Player
{
	string name;
	int hp;
	// A member function that reduces player hp by some amount
	void damage( int amount ) {
	hp -= amount;
	}
	void recover( int amount ) {
	hp += amount;
	}
};
```

Новые термины и **важные слова** выделены жирным шрифтом. Текст, который отображается на экране, будет показан так: В меню **File**, выберите **New Project**…

Примечания

Дополнительная информация, которая относится к делу, но немного отстранена, будет появляться в блоках как этот.

Подсказки

Подсказки и советы будут показаны так.

Глава 1. Написание кода с C++

Если ты программист-новичок. Тебе нужно много выучить!

Преподаватели университетов часто описывают принципы программирования в теории, но практическое применение любят оставлять на кого-нибудь другого. Предпочтительно на кого-то из этой индустрии. Мы в этой книге не поступаем так. В этой книге мы распишем теорию на фоне принципов и осуществления нашей собственной игры на C++ как таковой.

Первое что я посоветую, это что бы вы делали упражнения. Вы не сможете научиться программировать только читая. Вы должны работать с теорией в упражнениях.

Мы начнём с программирования очень простой программы на C++. Я знаю, что вы хотите начать играть в свою уже законченную, готовую игру прямо сейчас. Тем не менее, вы должны начать с самого начала, чтобы её закончить (если вы действительно хотите этого, то перейдите сразу на Главу 12, *Книга Заклинаний*, либо откройте некоторые из примеров, чтобы проникнуться тем к чему мы идём).

В этой главе, мы пройдём следующие темы:

- Установка нового проекта (в Visual Studio и Xcode)
- Ваш первый проект C++
- Как исправлять ошибки
- Что такое построение и компиляция?

Установка нашего проекта

Наша первая программа на C++ будет написана вне UE4. Для того чтобы начать, я покажу все шаги и для Xcode, и для Visual Studio 2013. Однако после этой главы я буду стараться говорить только о коде C++, не ссылаясь на то, используете ли вы Microsoft Windows, либо Mac OS.

Используем Microsoft Visual C++ на Windows

В этой части, мы установим редактор кода для Windows, Visual Studio от Microsoft. Пожалуйста, перейдите сразу в следующую часть, если вы пользуетесь Mac.

Примечание

Express издание Visual Studio – это бесплатная версия Visual Studio, которую Microsoft предоставляет на официальном сайте. Чтобы начать процесс установки,

перейдите на https://www.visualstudio.com/ru-ru/products/visual-studio-express-vs.aspx

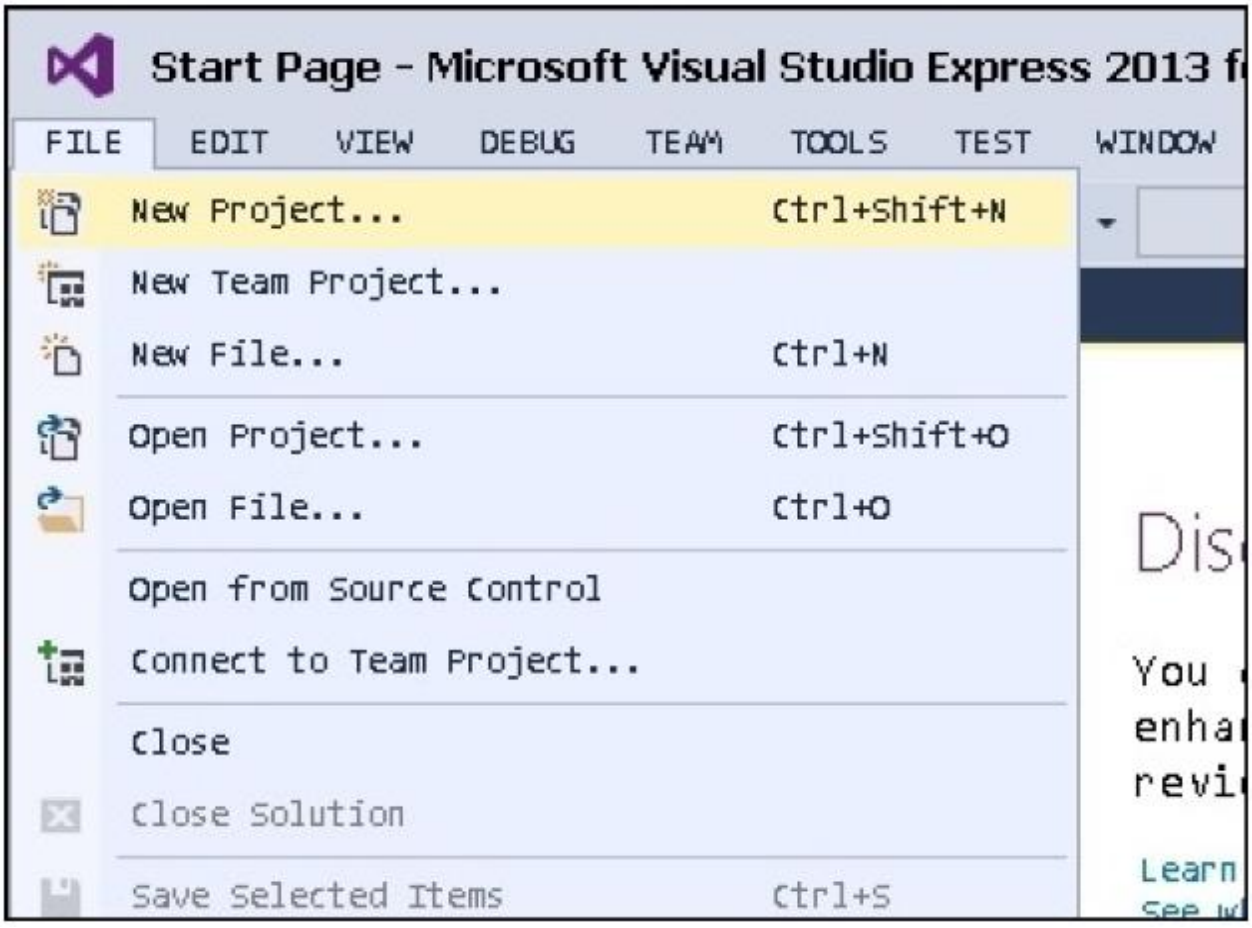
Чтобы начать вам нужно качать и установить **Microsoft Visual Studio Express 2013 для Windows Desktop**. Вот как выглядит значок для программного обеспечения:

Совет

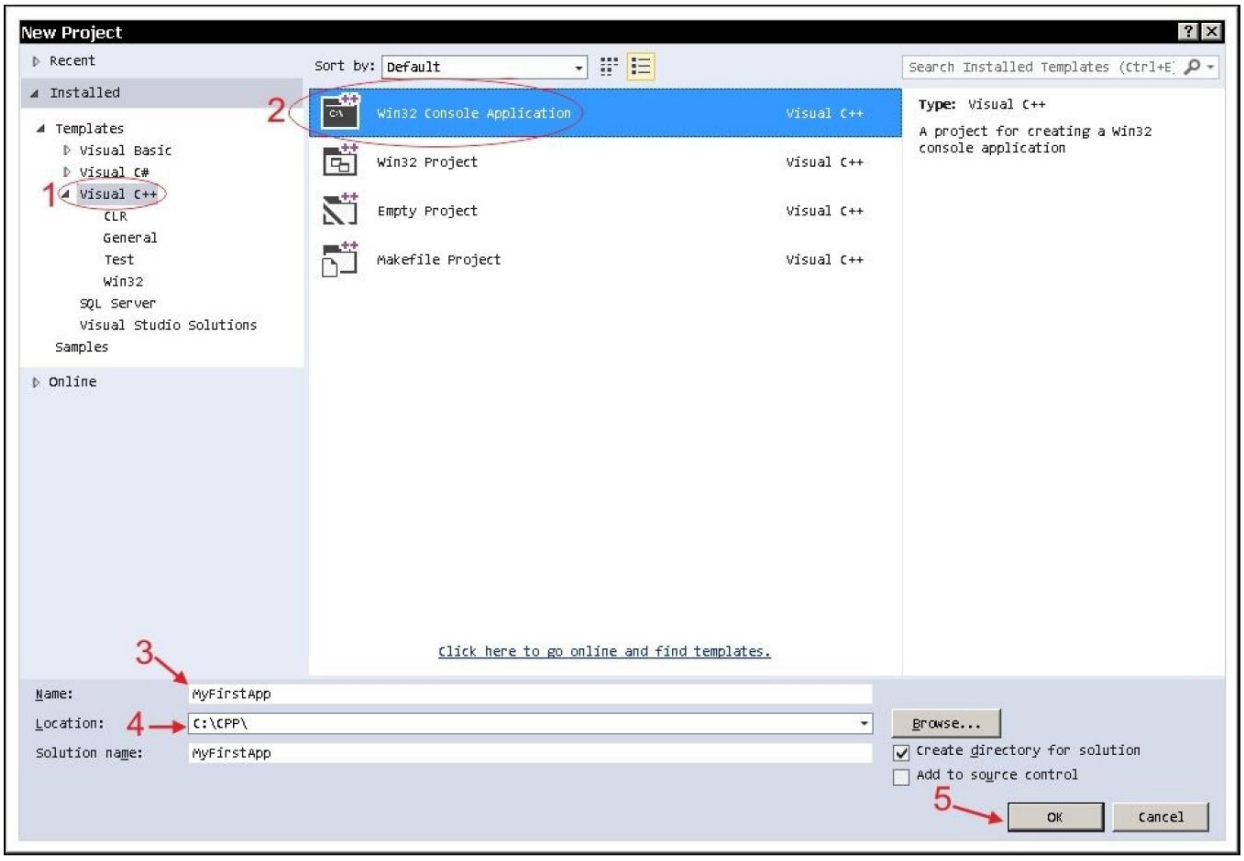
Не устанавливайте **Express 2013 для Windows**. Это другой пакет и он используется для других вещей, нежели мы здесь делаем.

Как только у вас установится Visual Studio 2013 Express, откройте его. Выполните следующие шаги, чтобы дойти до места, где вы уже и сможете писать код:

1. В меню **File**, выберите **New Project**, как показано на следующем скриншоте:



2. У вас откроется следующее диалоговое окно:



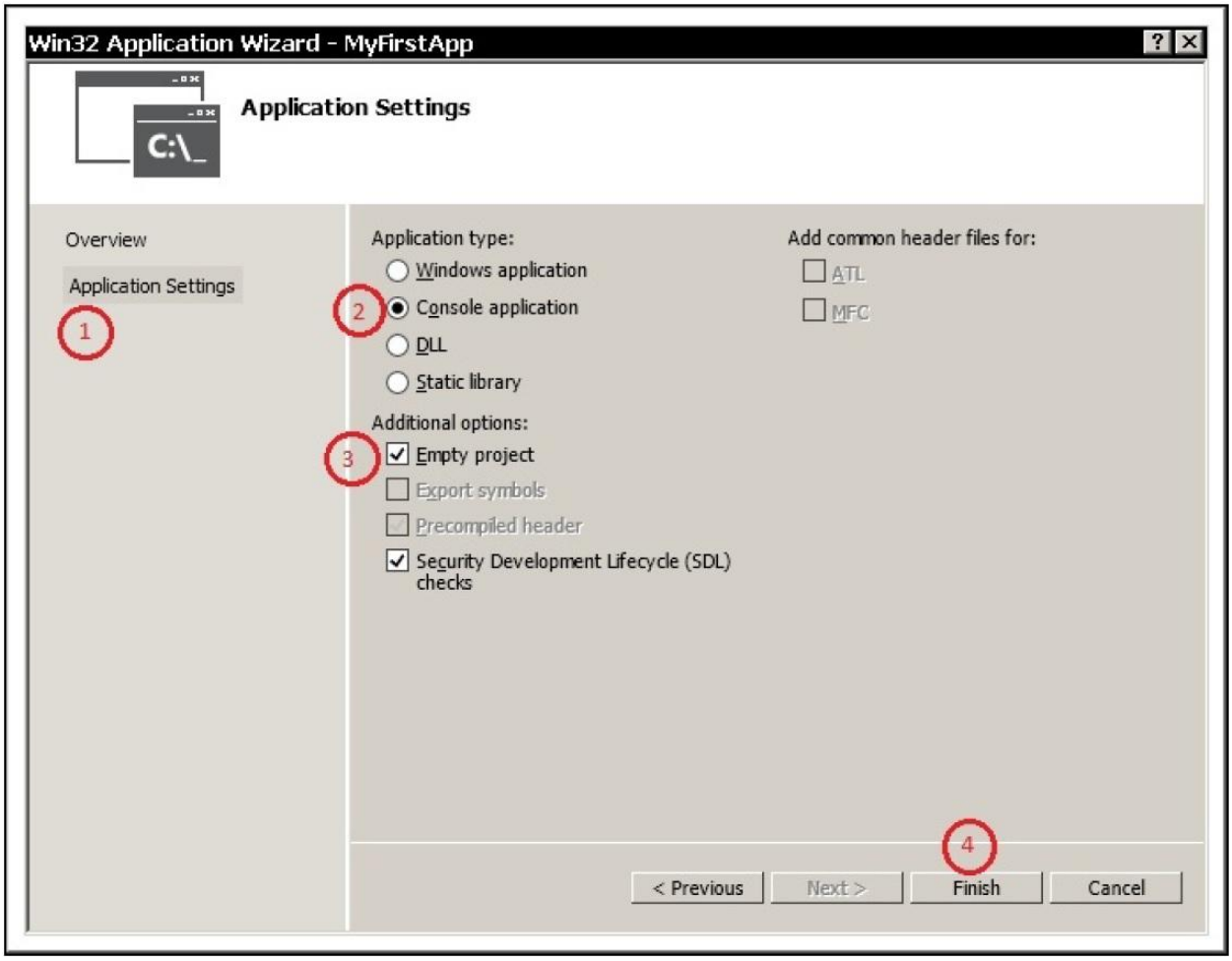
Совет

Обратите внимание, там есть поле с текстом **Solution name** (имя решения). В основном решения **Visual Studio**, могут содержать много проектов. Хотя эта книга и работает с одним единственным проектом, в какое-то время вы можете найти полезным, совместить много проектов в одном решении.

3. Здесь пять этапов, которые нужно выполнить в следующем порядке:

    1. Выберите **Visual C++** на панели слева.
    2. Выберите **Win32 Console Application** на панели справа.
    3. Назовите своё приложение (Я назвал своё MyFirstApp).
    4. Выберите папку для хранения вашего кода.
    5. Нажмите кнопку **OK**.

4. После этого откроется диалоговое окно **Application Wizard**, как показано на следующем скриншоте:

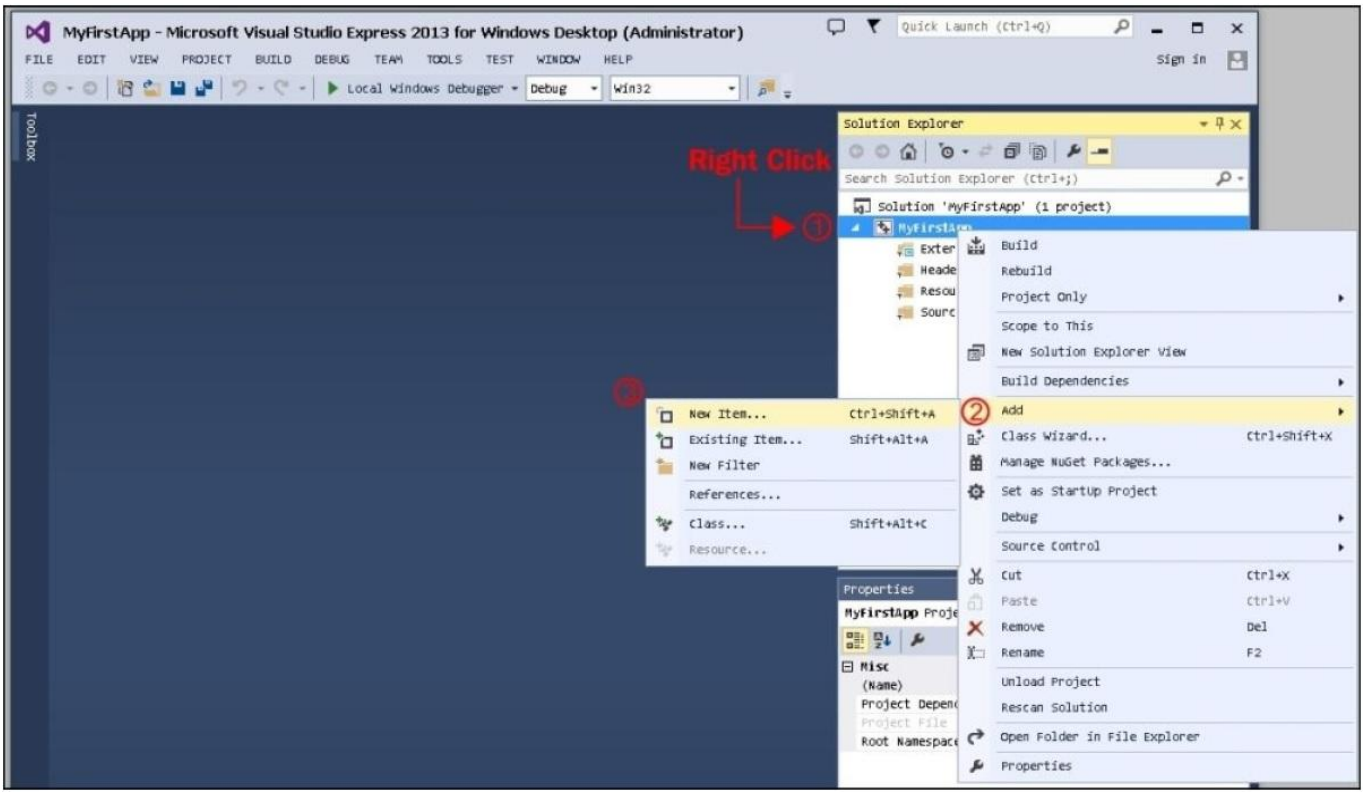


5. Нам нужно выполнить четыре следующих этапа в этом диалоговом окне:

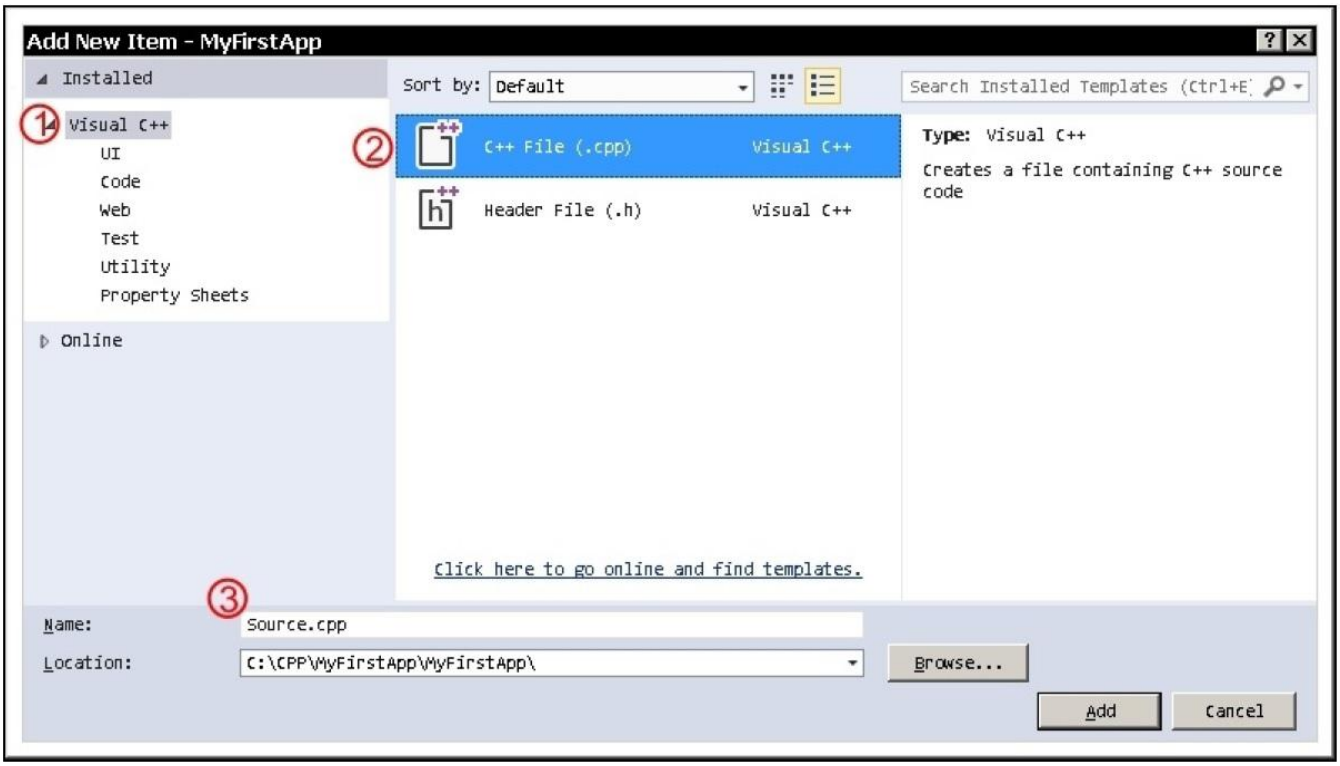
    1. Щёлкните по **Application Settings** на панели слева.
    2. Убедитесь что выбрано **Console application**.
    3. Выберите **Empty project**.
    4. Нажмите **Finish**.

Теперь вы в среде Visual Studio 2013. Это место где вы будете делать всю свою работу, и писать код.

Тем не менее, нам нужен файл, в котором мы будем писать код. Так что мы добавим в наш проект файл C++ кода, как показано на следующем скриншоте:



Добавьте ваш новый файл исходного кода, как показано на следующем скриншоте:



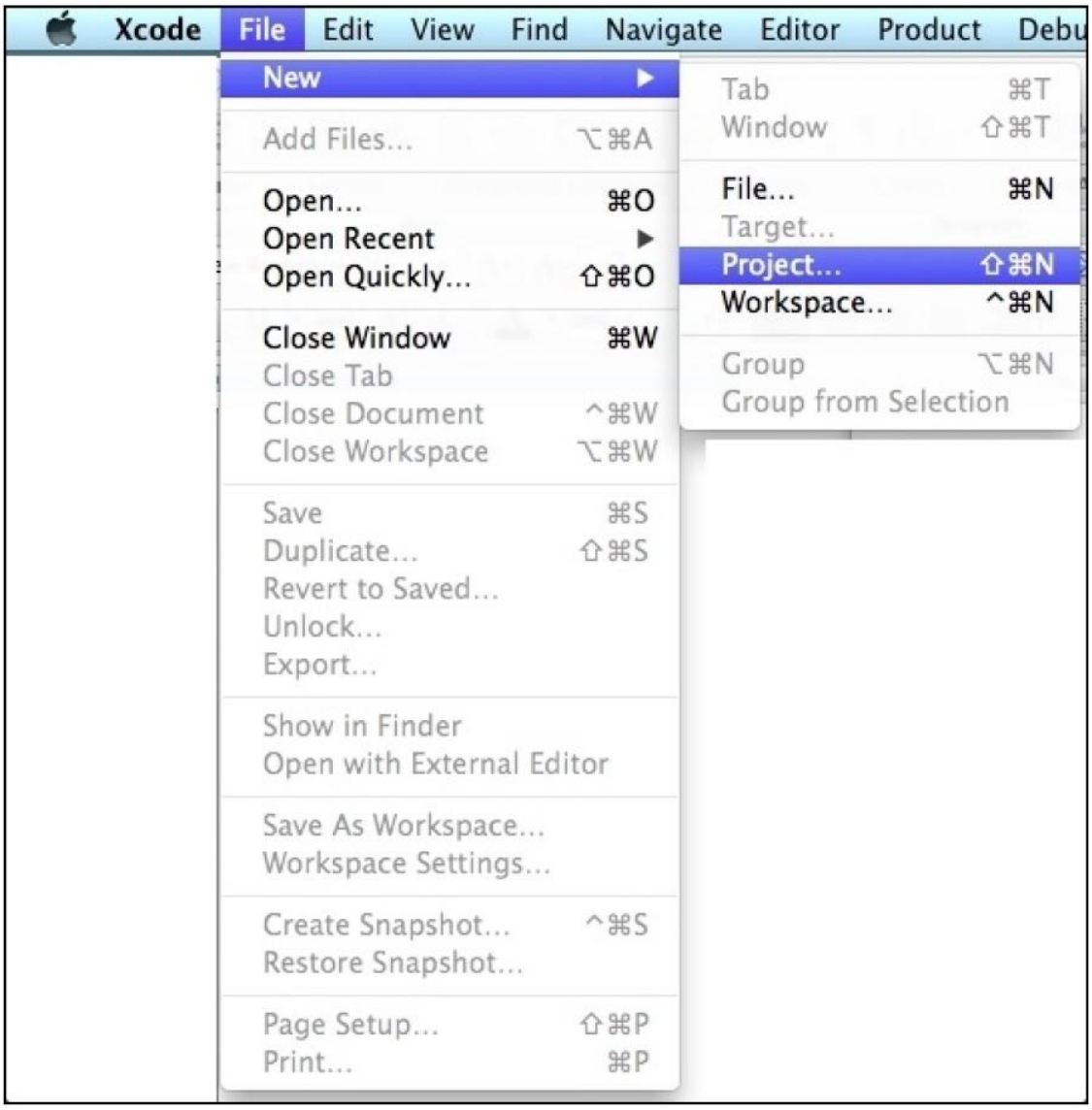
Теперь вы будете редактировать Source.cpp. Переходите к секции Вашей Первой C++ Программы и пишите внутри свой код.

Используем XCode на Mac

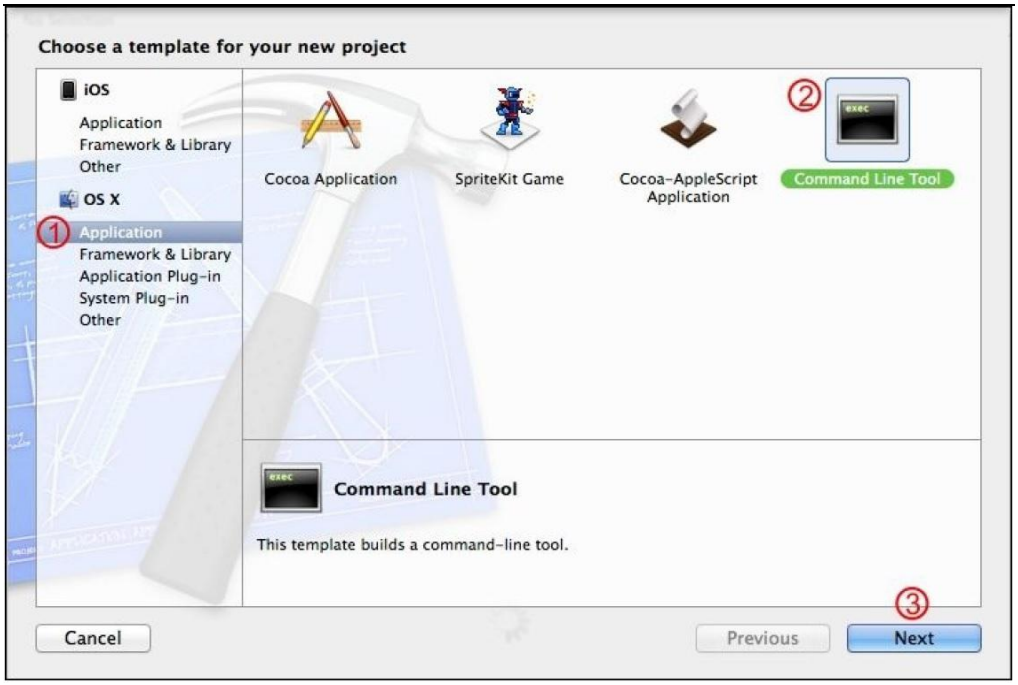
В этом разделе мы поговорим о том как установить XCode на Mac. Пожалуйста, перейдите сразу к следующему разделу, если вы пользуетесь Windows.

Xcode доступен на всех устройствах Mac. Вы можете получить XCode используя Apple App Store (бесплатно), как показано здесь:

1. Как только вы установили XCode, откройте его. Затем из системной панели меню вверху вашего экрана перейдите в **File** | **New** |**Project**… как показано на следующем скриншоте:



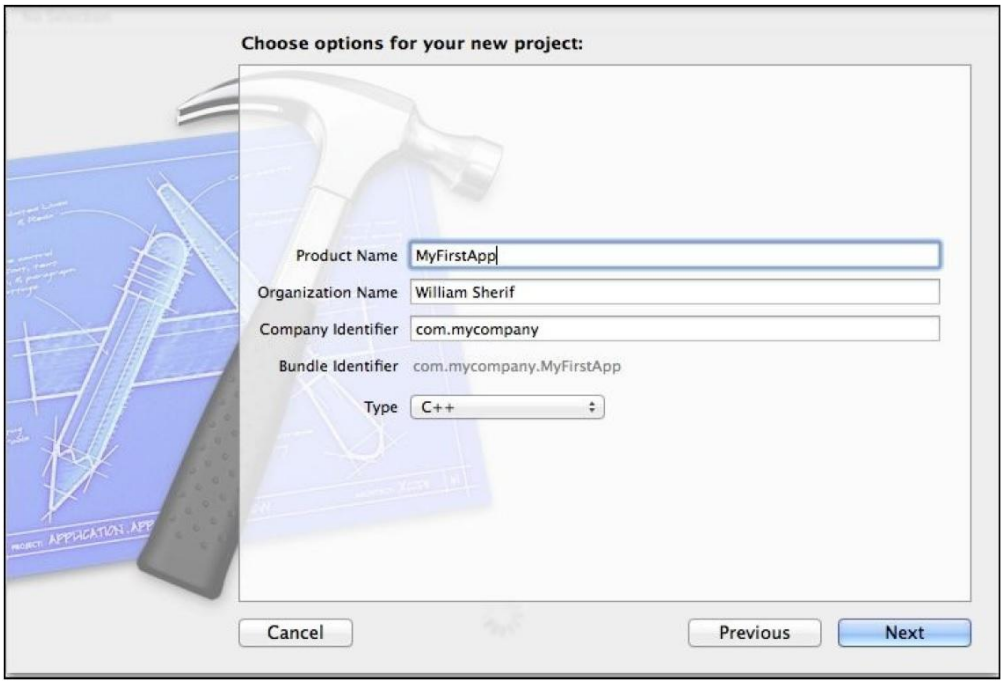
2. В диалоговом окне New Project, с левой стороны экрана выберите **Application** прямо под **OS X**, а с правой стороны выберите **Command Line Tool**. Затем, нажмите **Next**:



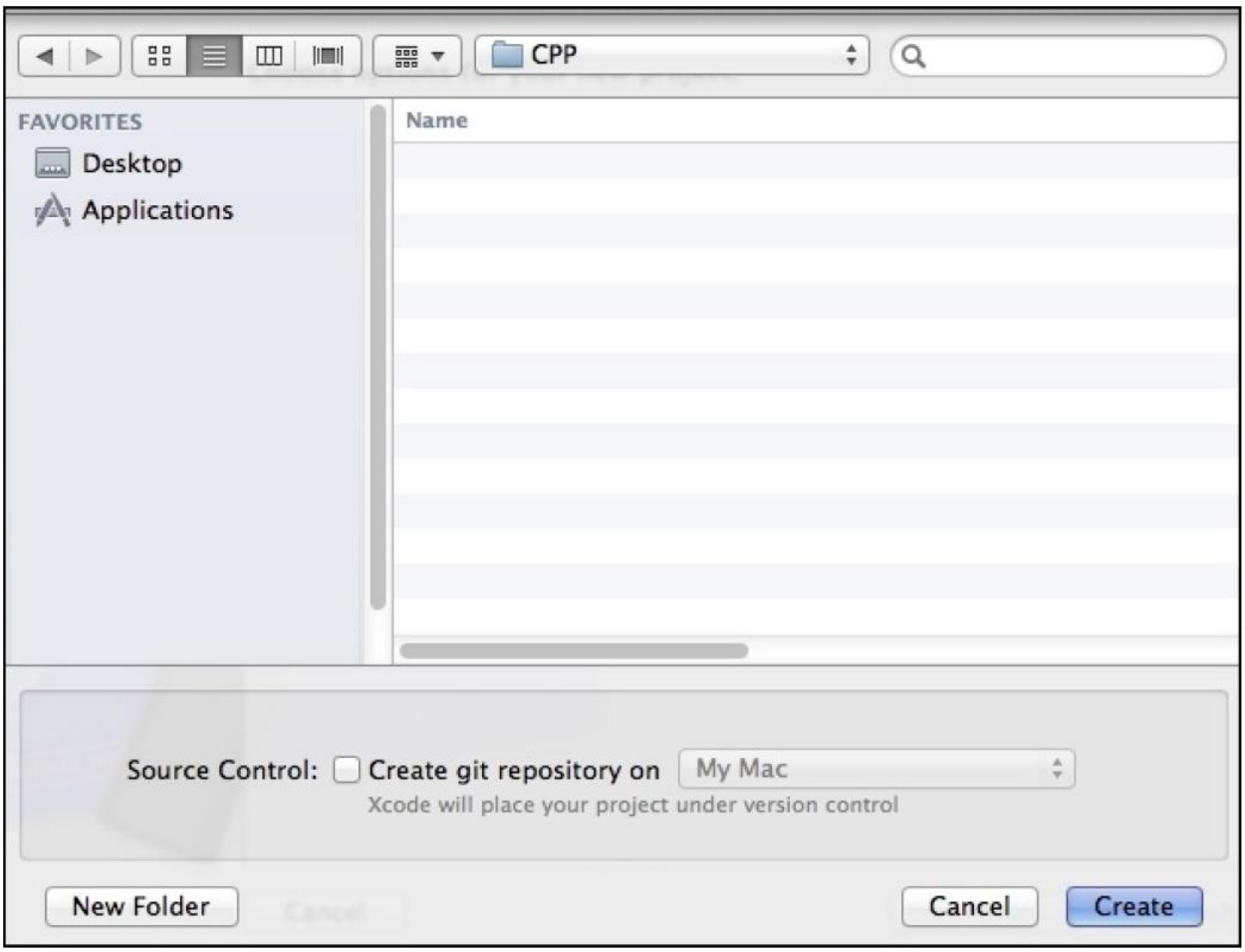
Примечание

Возможно, вас привлечёт значок SpriteKit Game и вы захотите кликнуть по нему, но не делайте этого.

3. В следующем диалоговом окне дайте имя своему проекту. Обязательно заполните все поля, иначе XCode не даст вам продолжить. Убедитесь что тип проекта, а именно пункт Type установлен на C++, а затем нажмите на кнопку Next, как показано здесь:



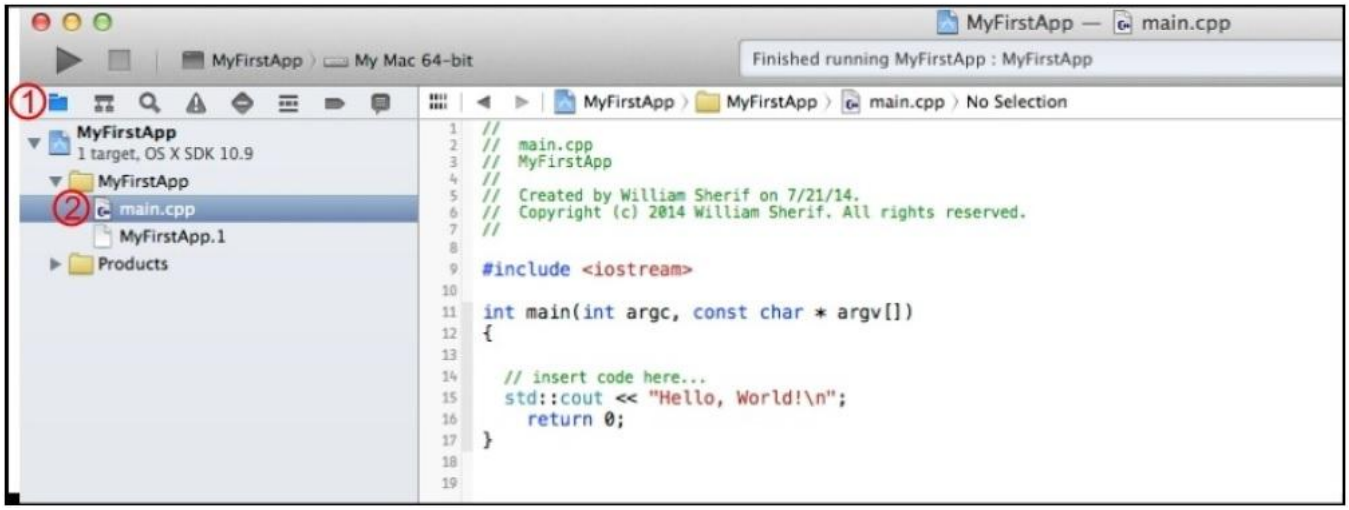
4. Следующее всплывающее окно попросит вас выбрать расположение для сохранения вашего проекта. Выберите место на своём жёстком диске и сохраните проект там. По умолчанию XCode создаёт Git хранилище для каждого созданного вами проекта. Вы можете убрать галочку с **Create git repository** (мы не будем проходить Git в этой главе), как показано на следующем скриншоте:



Совет

Git является **Системой управления версиями**. В основном это означает, что Git хранит копии состояния всего кода в вашем проекте в определённый момент времени (каждый раз, когда вы обращаетесь к хранилищу). Другими популярными инструментами **управления контролем источников** являются Mercurial, Perforce и Subversion. Когда множество людей совместно работают над одним проектом, эти инструменты имеют способность автоматически объединять и копировать изменения других людей из хранилища в вашу локальную базу кода.

Отлично! Вы всё установили. Щёлкните по файлу **main.cpp** на панели с левой стороны XCode. Если файл не появился, убедитесь, что сначала выбран значок папки наверху панели с лева, как показано на следующем скриншоте:



Создание вашей первой программы C++

Сейчас мы собираемся писать исходный код C++. Есть очень хорошая причина, по которой мы называем его именно исходный код: это исходник из которого мы будем строить наш бинарный исполняемый код. Один и тот же исходный C++ код может быть построен на различных платформах, таких как Mac, Windows и iOS. И в теории исполняемый код делает абсолютно одно и то же на каждой соответствующей платформе, производя должный результат.

В не таком далёком прошлом, до выхода языков C и C++, программисты писали код индивидуально для каждой конкретной машины, для которой он предназначался. Они писали код на языке называемом - язык ассемблера. Но сейчас когда доступны C и C++, программисты могут написать код один раз и он будет развёрнут на разных машинах, просто путём отправления одного и того же код через разные компиляторы.

Совет

На практике существуют некоторые различия между разновидностью C++ Visual Studio и разновидностью C++ XCode, но эти различия выявляются в основном при работе с продвинутыми концепциями C++, такими как шаблоны.

Одна из причин, по которой использование UE4 так помогает, это потому что UE4 стирает множество различий между Windows и Mac. Команда UE4 проделала не мало по настоящему волшебной работы, чтобы получить одинаковый код для работы и на Wndows и на Mac.

Примечание

Совет от real-world

Очень важно чтобы код выполнялся одинаковым образом на всех машинах, особенно для сетевых игр или игр которые позволяют коллективный перезапуск. Это может быть достигнуто при использовании стандартов. Например, стандарт плавающей точки IEEE используется, чтобы осуществлять математические действия с десятичной дробью на всех компиляторах C++. Это означает, что результат вычислений, таких как 200*3.14159 должны быть одинаковыми на всех машинах.

Напишите следующий код в Microsoft Visual Studio или в Xcode:

```
#include <iostream>    // Импорт библиотеки ввода-вывода
using namespace std; // позволяет нам писать оператор cout
                     // вместо std::cout
int main()
{
	cout << "Hello, world" << endl;
	cout << "I am now a C++ programmer." << endl;
	return 0; // "return" to the operating sys
}
```

Нажмите *Ctrl+F5*, чтобы запустить этот код в Visual Studio, либо нажмите ⌘*+R*, чтобы запустить его в Xcode.

Когда вы впервые нажмёте *Ctrl+F5* в Visual Studio, вы увидите это диалоговое окно:

Поставьте галочку на **Do not show this dialog again** (Не показывать это окно снова) и нажмите **Yes**. Поверьте мне, так вы избежите дальнейших проблем.

Первое что может прийти вам на ум, это: “Эй! Что за неразбериха!”.

И в самом деле, вы редко видите применение символа решётки # (разве что в пределах Twitter), а также фигурных скобок { }, в обычных текстах. Тем не менее, в коде C++ эти странные символы изобилуют. Вам просто нужно привыкнуть к ним.

Итак, давайте растолкуем эту программу, начиная с первой строки.

Вот первая строка программы:

```
#include <iostream>    // Импорт библиотеки ввода-вывода
```

В этой строке два важных пункта, на которые следует обратить внимание:

1. Первое что мы видим это оператор #include. Мы просим C++ скопировать и вставить содержимое другого исходного файла C++, который называется <iostream> , прямо в наш файл кода. <iostream> это стандартная C++ библиотека, которая управляет всем трудным кодом, который позволяет нам выводить текст на экран.
2. Второе на что мы обращаем внимание это //комментарий. C++ игнорирует любой текст после двойной косой черты //, до конца этой строки. Комментарии очень полезны и используются для добавления чётких объяснений того, что делает код. Также в источнике вы можете увидеть комментарии C-стиля /* */. Окружая любой текст в C или C++ косой чертой и звёздочкой/* и звёздочкой и косой чертой */, вы даёте инструкцию на удаление этого кода компилятором.

Вот вторая строка кода:

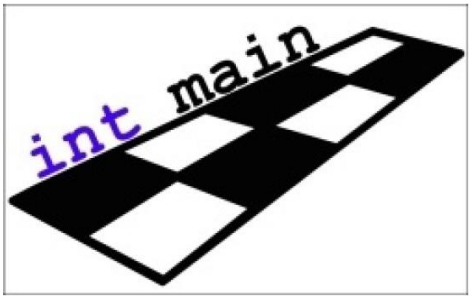
```
using namespace std; // позволяет нам писать оператор cout
                     // вместо std::cout
```

Комментарии с этой строкой объясняют, что делает оператор using: он просто позволяет вам использовать сокращение (например, cout) вместо полностью квалифицированного названия (что в данном случае было бы std::cout) для множества наших команд кода C++. Некоторым программистам не нравится утверждение using namespace std;. Они предпочитают писать std::cout каждый раз, когда они хотят применить cout. Вы можете вступать в долгие споры, о подобных вещах. В этой части текста мы предпочитаем сокращение, которое мы получаем от утверждения using namespace std;.

Вот следующая строка:

```
int main()
```

Это место начала применения. Вы можете представить main, как полосу старта в гонках. Утверждение int main(), даёт знать вашей C++ программе, откуда начинать. Взгляните на следующее изображение:



Если у вас в программе нет отметки int main(), или если main написано неверно, то ваша программа просто не захочет работать, потому что она не знает, откуда начать.

На следующей строке находится знак, который вы видите не часто:

{

Этот знак { - это не усы в стороны. Этот знак называется фигурная скобка, и он указывает на точку начала вашей программы.

Следующие две строки выводят текст на экран:

```
cout << "Привет мир!" << endl;
cout << "Теперь я программист C++." << endl;
```

Оператор cout отвечает за вывод на консоль. Текст между двойными кавычками будет выведен на консоль в точности так же как он представлен между кавычек. Между двойных кавычек вы можете писать всё что угодно, кроме других двойных кавычек, чтобы код оставался действительным.

Совет

Чтобы ввести двойные кавычки, между парой двойных кавычек вам необходимо поставить обратную косую черту перед теми кавычками, которые вы хотите иметь внутри строки. Как показано здесь:

```
cout << "Джон крикнул в пещеру: \"Эй!\", и в пещере раздалось эхо."
```

Этот знак \ обратная косая черта, является примером управляющей последовательности. Есть и другие управляющие последовательности, которые вы можете использовать. Самая распространённая управляющая последовательность, которую вы обнаружите это \n. Которая используется, чтобы переносить текст на следующую строку.

Последняя строка программы это оператор return:

```
return 0;
```

Эта строка кода указывает на то, что программа завершается. Вы можете думать об операторе return, как о возвращении к операционной системе.

И наконец то, само завершение вашей программы обозначается закрывающей фигурной скобкой:

}

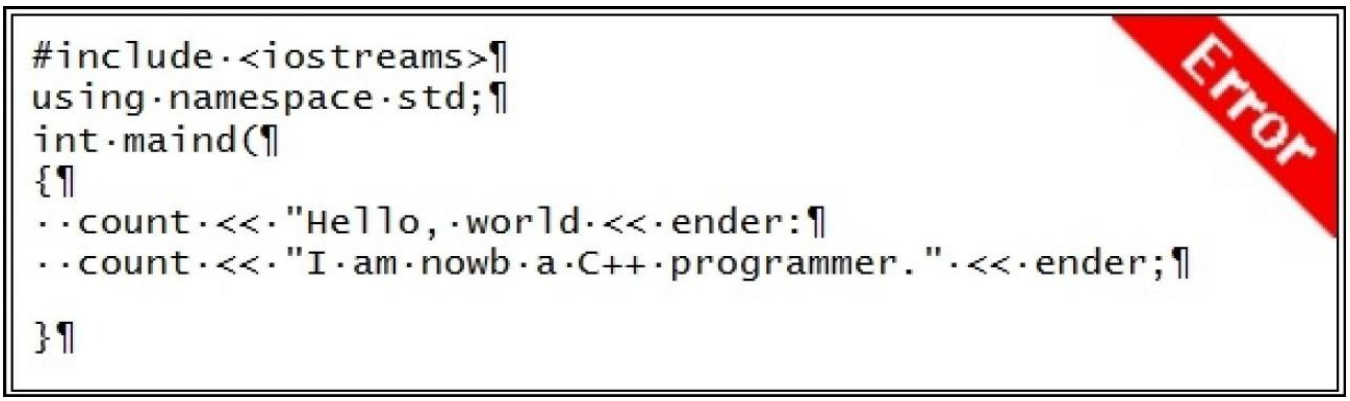
Точка с запятой

Точка с запятой (;) очень важный знак в C++ программировании. Обратите внимание, что в предыдущих примерах кода, большинство строк заканчиваются точкой с запятой. Если вы не будете заканчивать каждую строку точкой с запятой, то ваш ход не компилируется, а если это случится, то вас могут уволить с работы.

Исправление ошибок

Если вы допустили ошибку, пока вводили код, то у вас будет синтаксическая шибка. Встречая синтаксическую ошибку, C++ кричит: „Караул!“. И ваша программа даже не будет компилироваться, и естественно запущенна тоже не будет.

Давайте попробуем ввести пару ошибок в предыдущий пример нашего кода:



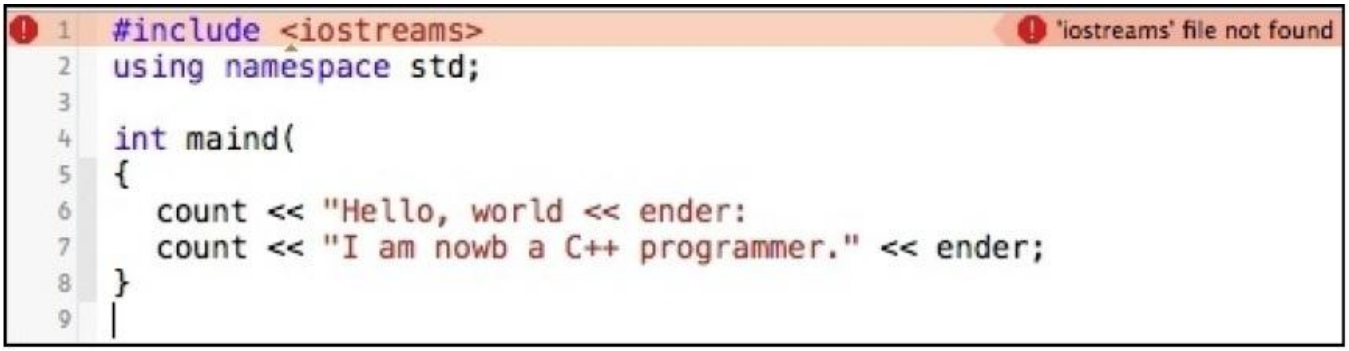
Внимание! Этот код содержит ошибки. И будет хорошей тренировкой найти все ошибки и исправить их!

В качестве упражнения постарайтесь найти и исправить все ошибки в этой программе.

Примечание

Заметьте, что если вы совсем новичок в C++, то это может быть трудным упражнением. Тем не менее, оно покажет вам насколько аккуратным нужно быть, когда пишешь код C++.

Исправление ошибок компиляции, может оказаться мучительным делом. Но зато, если вы поместите текст этой программы в ваш редактор кода и попробуете компилировать его, то компилятор доложит вам обо всех ошибках. Исправьте ошибки сразу и затем попробуйте заново компилировать. Либо всплывут новые ошибки, либо программа просто заработает, как показано на следующем скришоте:



Xcode показывает вам ошибки в вашем коде, когда вы пробуете компилировать его.

Причина, по которой я показываю вам этот пример программы, это чтобы поддержать последующий рабочий процесс. И так как C++ для вас ново:

1. Всегда начинайте с рабочих примеров кода C++. Из раздела *Ваша первая программа C++*, вы можете идти во многих разных направлениях C++ программ.
2. Модифицируйте свой код по маленьким шажкам. Если вы новичок, то компилируйте код после написания каждой новой строки. Не пишите код целый час или два, чтобы потом всё компилировать за раз.
3. Вам следует ожидать, что пройдёт пара месяцев, прежде чем вы начнёте писать код, который будет с первого раза выполняться, так как вы и ожидали. И не теряйте самообладания. Изучать код, на самом деле весело.

Предупреждения

Компилятор будет ставить флаг на то, что он считает ошибкой. Есть и другой класс замечаний компилятора, известный как предупреждения. Предупреждения – это проблемы в вашем коде, которые вам не обязательно исправлять, чтобы запустить код. Но компилятор крайне рекомендует их исправить. Предупреждения зачастую указывают на код, который не совсем идеален. И исправление предупреждений считается хорошей практикой.

Тем не менее, не все предупреждения послужат причиной проблем в вашем коде. Некоторые программисты предпочитают отключать предупреждения, которые они не считают важными (например, предупреждение 4018 уведомляет о несоответствии типов со знаком и без знака, которое вы вероятнее всего увидите далее).

Что такое построение и компиляция?

Вы может быть, слышали о компьютерном процессном термине компиляция. Компиляция – это процесс преобразования вашей C++ программы в код, который

может быть запущен на центральном процессоре. Построение вашего исходного кода означает то же, что и компилирование кода.

Посмотрите, ваш исходный файл code.cpp на самом деле не будет запускаться на компьютере. Он должен быть компилирован, прежде чем запуститься.

Это всё об использовании Microsoft Visual Studio Express и Xcode. Visual Studio и Xcode оба являются компиляторами. Вы можете писать исходный код C++ в любой программе редактирования текста, даже в Notepad. Но вам нужен компилятор, чтобы запустить этот код на вашей машине.

В каждой операционной системе обычно имеется один или больше C++ компиляторов, которые могут компилировать C++ код, чтобы запускать его на этой платформе. На Windows у вас есть компиляторы Visual Studio и Intel C++ Studio. На Mac есть Xcode, а также на всех Windows, Mac и Linux есть **GNU Compiler Collection (GCC).**

Тот же C++ код, что мы пишем (Исходник) может быть компилирован при использовании различных компиляторов для различных операционных систем. И в теории они должны выдавать одинаковый результат. Возможность компилировать дин и тот же код на разных платформах, называется портируемость. В целом портируемость это хорошая вещь.

Скрипт

Существует ещё один класс языков программирования называемых скриптовыми языками. Это такие языки как PHP, Python, ActionScript. Скриптовые языки или языки сценариев, не компилируются. Для JavaScript, PHP и ActionScript нет шага компиляции. Вместо этого они интерпретируются из источника, когда программа запускается. У скриптовых языков есть хорошее свойство, которое заключается в том, что обычно они кроссплатформенны. И неважно, на какой именно платформе изначально шло написание, потому что интерпретаторы очень старательно разработаны, чтобы они стали кроссплатформенными.

Осуществление искусства ASCII

Игровые программисты любят искусство ASCII. Вы можете нарисовать картинку, используя только знаки. Вот пример изобразительного искусства ASCII:

```
cout << "*****************" << endl;
cout << "*.............*.*" << endl;
cout << "*.*.*******..*.*" << endl;
cout << "*.*.*.........*" << endl;
cout << "*.*.*.**********" << endl;
cout << "***.***........*" << endl;
```

Составьте свою собственную графику в C++ коде или нарисуйте картинку знаками.

Выводы

Подводим итоги. Мы узнали, как писать нашу первую программу на языке программирования C++, в нашей интегрированной среде разработки (ИСР, Visual Studio, Xcode). Это была простая программа, но вам следует считать компиляцию и запуск вашей программы как вашу первую победу. В следующих главах, мы объединим больше сложных программ и начнём использовать Unreal Engine для наших игр.

```
1   #include <iostream>  // Import the input-output library
2   using namespace std; // allows us to write cout
3                        // instead of std::cout
4
5   int main()
6   {
7       cout << "Hello, world" << endl;
8       cout << "I am now a C++ programmer." << endl;
9       return 0;
10  }
11
```

Сверху скриншот вашей первой C++ программы, а следующий скриншот – это её вывод и ваша первая победа:

Глава 2. Переменные и память

Чтобы писать вашу C++ программу игры, вам нужно будет, чтобы ваш компьютер запоминал множество вещей. Таких, как например: где в мире находится игрок, сколько у него здоровья, как много боеприпасов у него осталось, где в мире расположены предметы и какие улучшения они дают, а также буквы, которые формируют имя игрока на экране.

В вашем компьютере имеется своего рода система ввода и редактирования графической информации, называемая *память* или ОЗУ (оперативное запоминающее устройство). Физически компьютерная память сделана из кремния и выглядит как на следующем изображении:

Это ОЗУ похоже на парковку? Такую метафору мы будем использовать.

Оперативная память также означает RAM аббревиатура от Random Access Memory – память с произвольным доступом. С произвольным доступом, потому что вы можете иметь доступ к любой её части в любое время. Если у вас до сих пор есть CD, валяющиеся где то, то они как раз таки являются примером не произвольного доступа. CD предназначен для чтения и воспроизведения по порядку. Я всё ещё помню переходы треков в альбоме *Dangerous* Майкла Джексона, когда я перематывал назад переключая треки на диске, что занимало кучу времени! Зато переходы и доступ к разным ячейкам ОЗУ вообще не занимают много времени. ОЗУ является типом памяти быстрого доступа, известного как флеш-память.

ОЗУ названа энергозависимой флеш-памятью, потому что когда компьютер выключался, содержимое ОЗУ очищалось и прежнее содержимое ОЗУ терялось, пока оно сначала не было сохранено на жёсткий диск.

Для постоянного хранения, вам необходимо сохранять ваши данные на жёсткий диск. Есть два основных типа жёстких дисков: **HDD** (***hard (magnetic) disk drive*** - *накопитель на жёстких магнитных дисках*) и **SSD** (***solid-state drive*** - *твердотельный накопитель*). SSD которые к тому же являются не механическими более современные чем HDD, так как они используют принцип ОЗУ, быстрого доступа к (флеш) памяти. Однако в отличие от ОЗУ, данные на SSD остаются и после

того как компьютер был выключен. Если у вас есть возможность, я крайне рекомендую вам использовать их. Магнитные жёсткие диски уже устарели.

Нам нужен способ резервировать место в ОЗУ и считывать оттуда, и записывать туда. И к счастью, C++ легко выполняет это.

Переменные

Сохранённые локации в компьютерной памяти, которые мы можем считывать или записывать называются *переменными*.

Переменная – это компонент значение которого может меняться. В компьютерной программе, вы можете считать переменную неким контейнером, в котором вы можете хранить данные. В C++ эти контейнеры данных (переменные) имеют разные типы. Вам нужно применять верный тип контейнера данных, чтобы сохранить ваши данные в вашей программе.

Если вы хотите сохранить целое число, такое как 1, 0 или 20, то вы будете применять тип контейнера *int*. Вы можете применять контейнер плавающего типа, чтобы работать со значениями (десятичных дробей) с плавающей точкой, такими как 38.87. И вы можете применять строковые переменные, чтобы работать с буквенными строками (думайте об этом как о “жемчужинах на нитке”, где каждая буква это жемчужина).

Вы можете думать о своём занятом месте в ОЗУ как о занятом парковочном месте на гаражной стоянке. Как только мы объявляем нашу переменную и получаем место для неё, то операционная система больше никому (даже другим программам, работающим на этой же машине) не даст эту часть ОЗУ. Оперативная память рядом с вашей переменной может либо быть не использованной, либо использованной другими программами.

Подсказка

Операционная система существует, чтобы не давать программам наступать друг другу на ноги и не получать доступ к одним и тем же битам компьютерного железа в одно и то же время. В целом, обывательские компьютерные программы не должны писать и читать в памяти друг друга. Однако, некоторые типы обманных программ (например, maphack – взламывающие карты в играх) тайно проникают в память вашей программы. Такие программы как PunkBuster были представлены в свет, чтобы предотвращать читерство в онлайн играх.

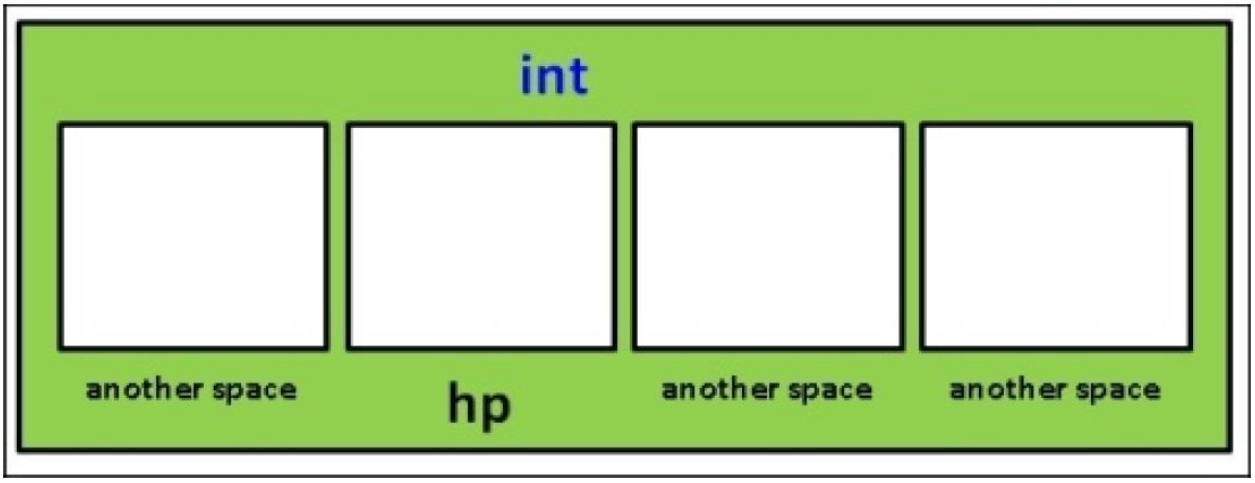
Объявление переменных – затрагивание кремния

Занимать место в компьютерной памяти используя C++ легко. Нам надо будет назвать наш участок памяти, в котором мы будем хранить наши данные, хорошо наглядным именем.

Например, скажем, мы знаем, что **hit points** (**hp**) игрока, а это подразумевает единицы здоровья, будут целыми числами, такими как 1, 2, 3 или 100. Чтобы получить часть кремния для хранения hp игрока в памяти, мы объявим следующую строку кода:

```
int hp; // объявляем переменную, чтобы хранить hp игрока
```

Эта строка кода занимает маленький участок оперативной памяти, чтобы хранить целое число, названное hp (int является сокращением от intiger – целое число). Далее идёт пример нашего участка оперативной памяти, используемого для хранения hp игрока. Так занимается парковочное место для нас в памяти (посреди других парковочных мест) и мы можем ссылаться к этому месту в памяти посредством ярлыка hp.



Посреди всех других участков в памяти, мы получаем одно место для хранения наших hp данных

Обратите внимание, как область переменной отмечена в этой диаграмме типом **int**: если это область для двойной или другого типа переменной. C++ запоминает области, которые вы занимаете для своей программы, не только по имени, но также и по типу переменной.

Обратите внимание, что мы ещё ничего не положили в ящик hp! Мы сделаем это позже, а прямо сейчас значение переменной hp не установлено. Так что она будет иметь значение, которое было оставлено здесь предыдущим владельцем (возможно значение, оставшееся после другой программы). Сообщать C++ тип переменной очень важно! Позже мы объявим переменную для хранения значения десятичной дроби, такой как 3.75.

Чтение и запись в занятом вами месте в памяти

Записывать значение в память легко! Как только у вас появилась переменная hp, вы просто приписываете ей значение, используя знак =:

```
hp = 500;
```

Вуаля! У игрока есть 500 единиц.

Чтение переменной также просто. Чтобы вывести значение переменной, просто введите это:

```
cout << hp << endl;
```

Это выведет значение, хранящееся в переменной hp. Если вы изменяете значение hp, а затем применяете cout снова, то будет выведено самое последнее значение, как показано здесь:

```
hp = 1200;
cout << hp << endl; // теперь показывает 1200
```

Числа это всё

Кое-что к чему вам нужно привыкнуть, когда вы начинаете компьютерное программирование, это то, что поразительное число вещей может храниться в компьютерной памяти просто как числа. Единицы здоровья игрока? Как мы только что видели в предыдущем разделе, единицы здоровья могут быть целыми числами. Если игроку нанесён урон, то мы понижаем это число. Если игрок приобрёл здоровье, то мы повышаем это число.

Цвета так же могут храниться в числах! Если вы используете стандартные программы редактирования изображения, то обычно имеются ползунки, которые указывают цвет, показывая как много красного, зелёного и синего было использовано. Как ползунки цвета в Pixelmator. Цвет в свою очередь представлен трёмя числами. Фиолетовый цвет, показанный на рисунке 1, является (R=127, G=34, B=203).

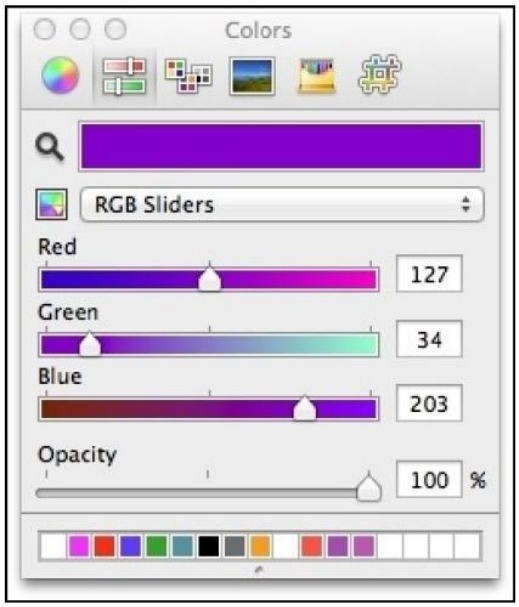


Рисунок 1

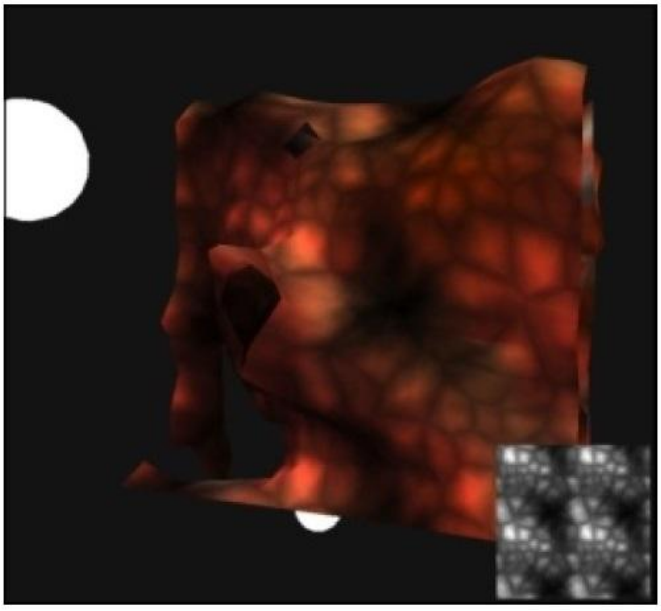


Рисунок 2

Что насчёт мировой геометрии? Это также просто числа: всё, что нам нужно делать, это хранить список точек 3D пространства (координаты x, y и z) и затем хранить ещё один список точек, который объясняет, как те точки могут быть соединены, чтобы формировать треугольники. На рисунке 2 мы можем видеть, как использованы точки 3D пространства, чтобы представить мировую геометрию.

Комбинация чисел для цветов и чисел для точек 3D пространства позволят вам нарисовать большие и цветные пейзажи в вашем игровом мире.

Фокус с предшествующими примерами в том, как мы истолковываем сохранённые числа, так мы можем заставить их значить то, что мы хотим, чтоб они значили.

Больше о переменных

Вы можете думать о переменных, как о клетках для перевозки животных. Сумку-переноску для кошки, можно использовать, чтобы переносить кошку, но не собаку. Таким же образом вы должны использовать переменную плавающего типа, чтобы переносить значения десятичных дробей. Если вы храните значение десятичной дроби внутри переменной int, то это не подойдёт:

```
int x = 38.87f;
cout << x << endl; // выводит 38, а не 38.87
```

Что на самом деле происходит здесь, это то, что C++ производит автоматическое преобразование значения 38.87, *таинственным образом превращая* его в целое число, чтобы оно подходило для содержащего его контейнера int. Десятичная дробь 38.87 преобразуется в целочисленное значение 38.

Так например, мы можем модифицировать код, чтобы включить применение трёх типов переменных, как показано в следующем коде:

```
#include <iostream>
#include <string> // это необходимо для использования строковых переменных!
using namespace std;
int main()
{
	string name;
	int goldPieces;
	float hp;
	name = "William"; // Это моё имя
	goldPieces = 322; // начинаю с таким количеством золота
	hp = 75.5f;		// единицы здоровья выражены значением десятичной дроби
	cout << "Character " << name << " has "
		<< hp << " hp and "
		<< goldPieces << " gold.";
}
```

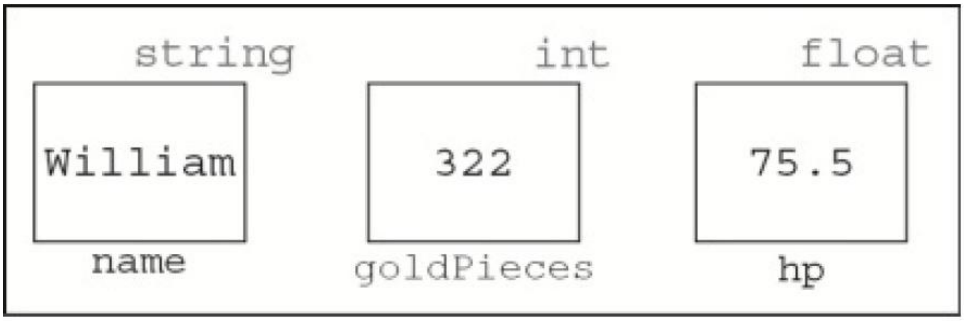
На первых трёх строках, мы объявляем три ящика для хранения в них частей наших данных как показано здесь:

```
string name;
int goldPieces;
float hp;
```

Эти три строки занимают три места в памяти (подобно парковочным местам). А следующие три строки заполняют переменные значениями, которые мы хотим, как здесь:

```
name = "William";
goldPieces = 322;
hp = 75.5f;
```

В компьютерной памяти, это выглядит, как показано на следующем изображении:



Вы можете менять содержимое переменной в любое время. Вы можете писать переменную, используя оператор назначения =, как здесь:

```
golfdPieces = 522; // знак = называется “оператор назначения”
```

Также в любое время вы можете считывать содержимое переменной. Это то, что делают три следующие строки кода здесь:

```
cout << "Character " << name << " has "
     << hp << " hp and "
     << goldPieces << " gold.";
```

Взгляните на эту строку:

```
cout << "I have " << hp << " hp." << endl;
```

Есть два применения hp на этой строке. Одно между двойных кавычек, а другое нет. Слова между двойных кавычек всегда выводятся именно так, как вы их там и написали. Когда не используются двойные кавычки (например, << hp <), выполняется просмотр переменной. Если переменная не существует, тогда вы получите ошибку компилятора (undeclared identifier – необъявленный идентификатор).

В памяти есть место для размещения имени – name, для того сколько золота есть у игрока – goldPieces, и для здоровья игрока – hp.

Совет

В целом, вы всегда должны стараться хранить соответствующий тип данных в соответствующей переменной. Если вы случайно сохраните несоответствующий тип данных, то ваш код поведёт себя не верно.

Математика в C++

Математические действия в C++ выполнять легко.

Все операции: + (плюс), - (минус), * (умножить), / (разделить); выполняются в порядке приоритета: Скобки, Возведение в степень, Деление, Умножение, Сложение и Вычитание. Например, мы можем написать, как показано в следующем коде:

```
int answer = 277 + 5 * 4 / 2 + 20;
```

Ещё один оператор, с которым вы возможно ещё не знакомы – это % (модуль). Модуль (например, 10 % 3) находит остаток от деления, когда x разделён на y. Посмотрите на следующую таблицу примеров:

Оператор (название)	**Пример**	**Ответ**
+ (плюс)	7 + 3	10
- (минус)	8 - 5	3
* (умножить)	5 * 6	30
/ (разделить)	12 / 6	2
% (модуль)	10 % 3	1 (потому что 10/3 даёт остаток = 1)

Тем не менее, зачастую мы не хотим выполнять математические действия в такой манере. Вместо этого, мы обычно хотим менять значение переменной на определённый вычисленный результат. Этот принцип понять тяжелее. Скажем, игрок встречает чертёнка и получает урон в 15 единиц.

Следующая строка кода будет применяться для уменьшения hp игрока на 15 (верите вы или нет):

```
hp = hp - 15;     // возможно это вас смутит :)
```

Вы можете спросить почему. Потому что по правую сторону мы вычисляем новое значение для hp (hp – 15). После того как новое значение для hp найдено (на 15 меньше чем было до этого), новое значение записано в переменную hp.

Подсказка

Тонкости

Переменная без начального значения имеет образец бита, который держался в памяти для неё до этого. Объявление переменной не очищает память. Итак, скажем мы используем следующую строку кода:

```
int hp;
hp = hp - 15;
```

Вторая строка кода уменьшает hp на 15 от предыдущего значения. Каково будет предыдущее значение, если мы вообще не устанавливали hp = 100 или ещё чему либо? Оно может быть 0, но не всегда.

Одна из самых распространённых ошибок, это запускать код с использованием переменной, для которой сперва не было присвоено значение.

Далее сокращённый синтаксис для hp = hp – 15:

```
hp -= 15;
```

Помимо -=, вы можете применять: += для добавления определённой величины к переменной, *= для умножения переменной на величину, и /= для деления переменной на величину.

Упражнения

Запишите значение **x** после выполнения следующих действий; затем свертись со своим компилятором:

Упражнения	**Решения**
int x = 4; x += 4;	8
int x = 9; x -= 2;	7

int x = 900; x /= 2;	450
int x = 50; x *= 2;	100
int x = 1; x += 1;	2
int x = 2; x -= 200;	-198
int x = 5; x *= 5;	25

Обобщённый синтаксис переменных

В предыдущем разделе, вы узнали, что каждая часть данных, которую вы сохраняете в C++ имеет определенный тип. Все переменные созданы одним образом; в C++ объявление переменной имеет следующую форму:

переменнойТип переменнойИмя

Тип переменной сообщает, какой тип данных мы собираемся хранить в нашей переменной. Имя переменной - это идентификатор, который мы используем для чтения либо записи этой части памяти.

Примитивные типы

Ранее мы говорили о том, как все данные в компьютере, с какого-то момента будут числами. Ваш компьютерный код ответственен за верную интерпретацию этих чисел.

Это говорит о том, что C++ определяет только несколько базовых типов данных, как показано в следующей таблице:

Char	Единственная буква, как ‘a’, ‘b’ или ‘+’
Short	Целое число от -32.767 до +32.768

Int	Целое число от -2.147.483.647 до +2.147.483.648
Float	Любое десятичное значение приблизительно от -1x1038 до 1x1038
Double	Любое десятичное значение приблизительно от -1x10308 до 1x10308
Bool	true или false

Есть беззнаковые версии каждого типа переменных, упомянутые в предыдущей таблице. Беззнаковая переменная может содержать натуральные числа, включая 0 (x >= 0). Например, беззнаковый short может иметь значение между 0 и 65535.

Примечание

Если в дальнейшем вам будет интересна разница между float и double, то вы свободно можете найти это в интернете. Я буду продолжать объяснять только самые важные принципы C++ используемые для игр. Если вам любопытно, что-то из того что было охвачено здесь, вы можете спокойно найти нужную информацию.

Всё оборачивается иначе, когда эти простые типы данных сами по себе могут быть использованы, чтобы без проблем строить сложные программы. “Как?” спросите вы. Разве не трудно строить 3D игры используя лишь плавающие типы и целочисленные?

На самом деле не трудно строить игру с типов float и int, но более сложные типы помогают. Будет нудно и неаккуратно программировать, если мы будем использовать распущенные значения с плавающей точкой для расположения игрока.

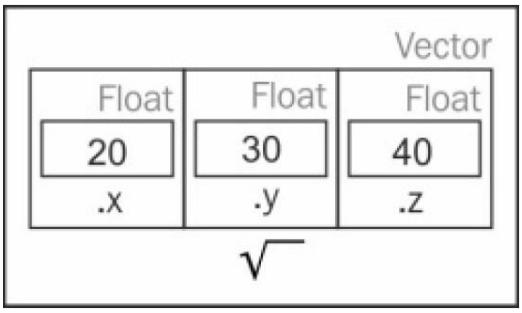
Типы объектов

C++ даёт вам структуры для группирования переменных вместе, что сделает вашу жизнь намного легче. Рассмотрите пример следующего блока кода:

```
#include <iostream>
using namespace std;
struct Vector // НАЧАЛО ОПРЕДЕЛЕНИЯ ОБЪЕКТА Vector
{
    float x, y, z; // положения x, y и  z все типа float
}; // КОНЕЦ ОПРЕДЕЛЕНИЯ ОБЪЕКТА Vector
  // Теперь компьютер знает чем является Vector
  // Так что мы можем его создавать
```

```
int main()
{
    Vector v; // Создаём экземпляр Vector названный v
    v.x=20, v.y=30, v.z=40; // присваиваем значения
    cout << "A 3-space vector at " << v.x << ", " << v.y << ", " << v.z <<
    endl;
}
```

В памяти это выглядит довольно таки наглядно; Vector является просто участком памяти с трёмя типами float, как показано здесь:



Подсказка

Не путайте структуру Vector на предыдущем изображении с std: :vector от STL. Объект Vector сверху предназначен для представления трёхмерного вектора, в то время как тип std: :vector от STL представляет размерное собрание значений.

Вот пара заметок о предыдущем коде:

Во первых, ещё до того как мы применим наш объектный тип Vector, нам нужно определить его. C++ не идёт со встроенными типами для математических векторов (а поддерживает только скалярные величины и думалось, что этого хватит!). Итак, C++ позволяет вам строить ваши собственные объектные конструкции, чтобы сделать вашу жизнь легче. Сначала у нас есть следующее определение:

```
struct Vector       // НАЧАЛО ОПРЕДЕЛЕНИЯ ОБЪЕКТА
{
      float x, y, z; // x, y и z положения, все типа float
};                   // КОНЕЦ ОПРЕДЕЛЕНИЯ  ОБЪЕКТА Vector
```

Это сообщает компьютеру, чем является вектор (три значения плавающего типа, которые объявлены и будут расположены друг за другом в памяти). То как будет выглядеть Vector в памяти, показано на предыдущем изображении.

Далее мы используем определение нашего объекта Vector, чтобы создать экземпляр от Vector названный v:

```
Vector v; // Создаёт экземпляр от Vector названный v
```

Определение struct Vector на самом деле не создаёт объект вектор. Вы можете выполнить: Vector.x = 1. “О каком экземпляре объекта вы говорите?” спросит

компилятор C++. Сначала вам надо создать экземпляр Vector, такой как Vector v1. Затем вы можете делать назначение экземпляру v1, такое как v1x = 0.

Затем мы используем этот экземпляр, чтобы записать значения в v:

```
v.x=20, v.y=30, v.z=40; // assign some values
```

Подсказка

Мы использовали запятые в предыдущем коде, чтобы присвоить значения нескольким переменным на одной строке. Это нормально в C++. Конечно вы можете сделать для каждой переменной свою строку, но и способ показанный здесь, тоже хорош.

Так v будет выглядеть как на предыдущем скриншоте. Затем мы выводим их:

```
cout << "A 3-space vector at " << v.x << ", " << v.y << ", " << v.z << endl;
```

Здесь, на обеих строках кода, мы получаем доступ к индивидуальным элементам данных внутри объекта просто используя точку (.). v.x ссылается к элементу x внутри объекта. В каждом объекте Vector также будет три значения плавающего типа: одно из которых названо x, другое y и ещё одно z.

Пример – Player

Определите C++ данные struct для объекта Player. Затем, создайте экземпляр вашего класса Player и заполните каждый элемент значениями.

Решение

Давайте объявим наш объект Player. Мы хотим всё сгруппировать вместе для игрока внутри объекта Player. Мы делаем это так, чтобы код был аккуратным и опрятным. Код, который вы читаете в Unreal Engine будет использовать такие объекты повсюду. Так что будьте внимательны:

```
struct Player
{
	string name;
	int hp;
	Vector position;
}; // Не забудьте эту точку с запятой в конце!
int main()
{
	// создаём объект типа Player,
	Player me; // экземпляру дано имя 'me'
	me.name = "William";
	me.hp = 100.0f;
	me.position.x = me.position.y = me.position.z=0;
}
```

Определение struct Player это то, что говорит компьютеру, как расположен объект Player в памяти.

Совет

Я надеюсь, вы заметили обязательность точки с запятой в конце объявления структуры (struct). Объявлениям объекта struct, нужно иметь точку с запятой в конце, а функциям нет. Это просто правило C++, которое вы должны запомнить.

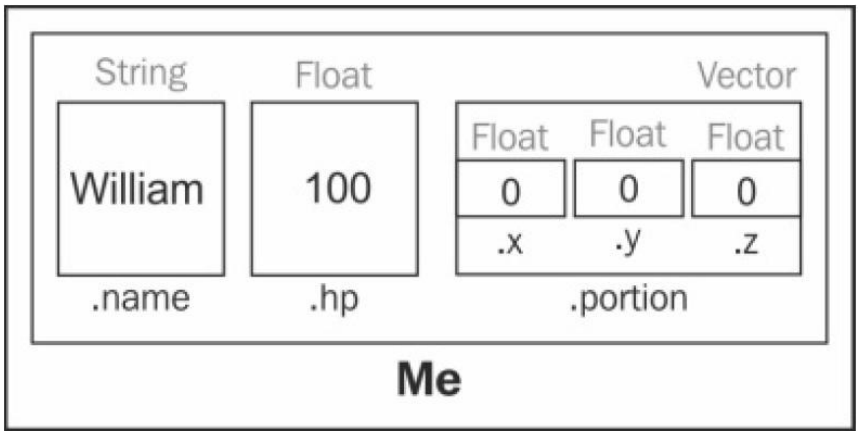
Внутри объекта Player, мы объявили строковый тип для имени игрока, тип с плавающей точкой для его hp и объект Vector для полных положений x, y, z.

Когда я говорю объект, я имею в виду C++ struct (или позже мы введём термин класс).

Погодите! Мы поместили объект Vector внутри объекта Player! Да, мы можем так делать.

После определения того, что имеется внутри объекта Player, мы и создаём экземпляр объекта Player, именуемый me и присваиваем ему значения.

После присвоения значений, объект me выглядит, так как показано на следующем изображении:



Указатели

Особенно мудрёным принципом для постижения, является принцип указателей. Указатели не так трудны для понимания, но может потребоваться время, чтобы разобраться с ними.

Скажем, у нас в памяти есть, как и до этого, объявленная переменная типа Player:

```
Player me;
me.name = "William";
me.hp = 100.0f;
```

Сейчас мы объявляем указатель к Player:

```
Player* ptrMe;                    // Объявление указателя
```

Знак * обычно придаёт особенность. В данном случае знак * делает особенным ptrMe. Знак звёздочка *, является тем, что делает ptrMe типом указатель.

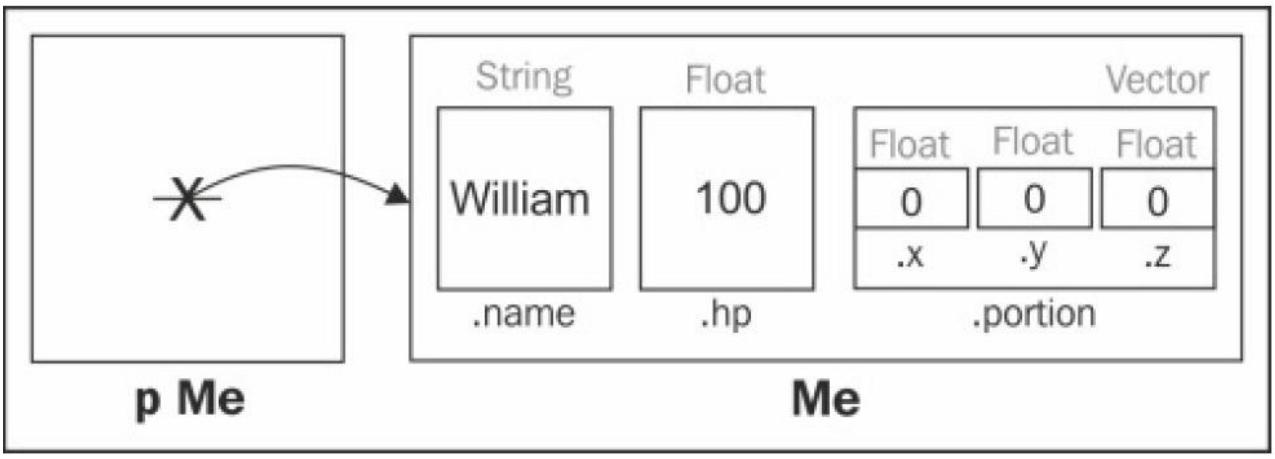
Сейчас мы хотим связать ptrMe к me:

```
ptrMe = &me;                      // СВЯЗЫВАНИЕ
```

Совет

Этот шаг связывания очень важен. Если вы не привяжите указатель к объекту перед использованием этого указателя, то у вас будет нарушение с получением доступа в памяти.

Сейчас ptrMe ссылается на тот же объект что и me. Изменение ptrMe изменит и me, как показано на следующем изображении:



Что могут делать указатели?

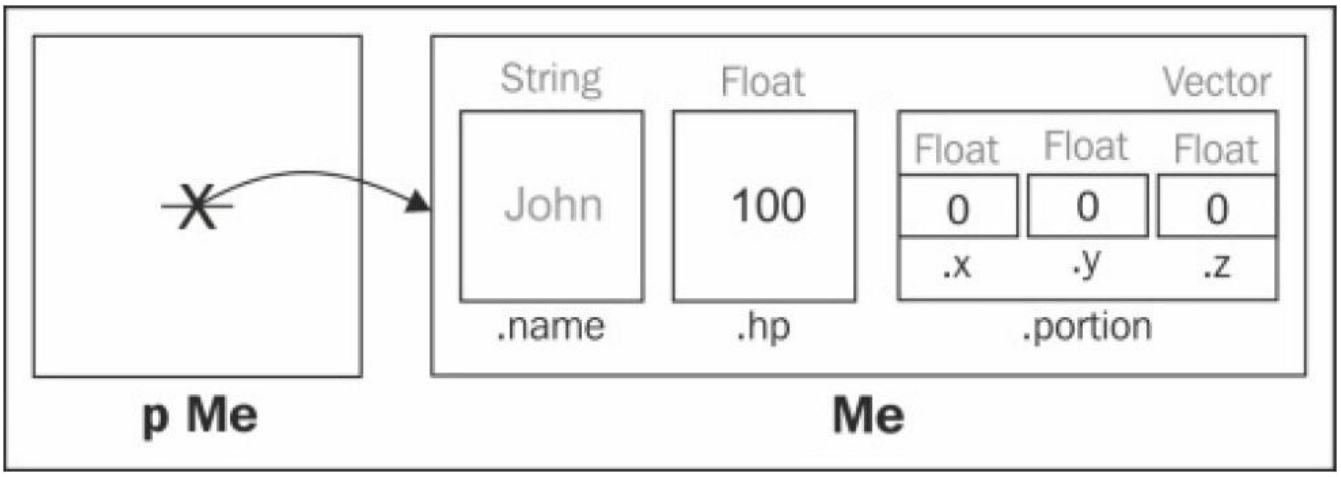
Когда мы устанавливаем привязку между переменной-указателем и тем на что она указывает, мы можем управлять указываемой переменной, через указатель.

Во первых, указатель можно использовать, чтобы ссылаться на один объект из нескольких различных локаций кода. Объект Player хороший кандидат на указание. Вы можете создавать так много указателей для одного объекта, сколько пожелаете. Объекты, на которые идёт указание, не обязательно могут знать, что на них идёт указание, тем не менее, изменения объекта могут быть сделаны через указатели.

Например, скажем, игрок бал атакован. В результате произойдёт уменьшение его hp, и это уменьшение будет выполнено при использовании указателя, как показано в следующем коде:

```
ptrMe->hp -= 33;            // reduced the player's hp by 33
ptrMe->name = "John";// changed his name to John
```

Вот как теперь выглядит объект Player:



Итак, мы изменили me.name изменив ptrMe->name. Потому что ptrMe указывает на me. Изменения через ptrMe прямо влияют на me.

Помимо того интересный синтаксис стрелок (используйте -> когда переменная является указателем), это принцип который не трудно понять.

Адрес оператора &

Обратите внимание на применение символа амперсанд - & в предыдущем примере кода. Оператор & получает адрес переменной в памяти. Адрес переменной в памяти – это место где живёт переменная в пространстве памяти компьютера. C++ имеет способность получать адрес любого объекта в памяти вашей программы. Адрес переменной уникален, и своего рода случаен.

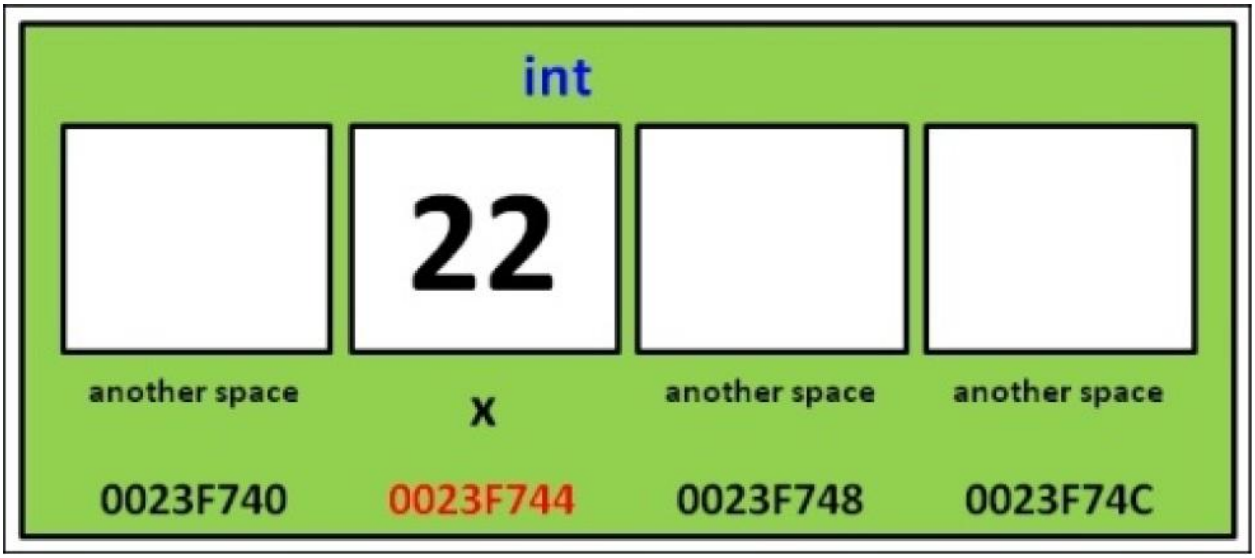
Скажем, мы выводим адрес целочисленной переменной x:

```
int x = 22;
cout << &x << endl; // выводит адрес x
```

При первом запуске, мой компьютер выводит следующее:

```
0023F744
```

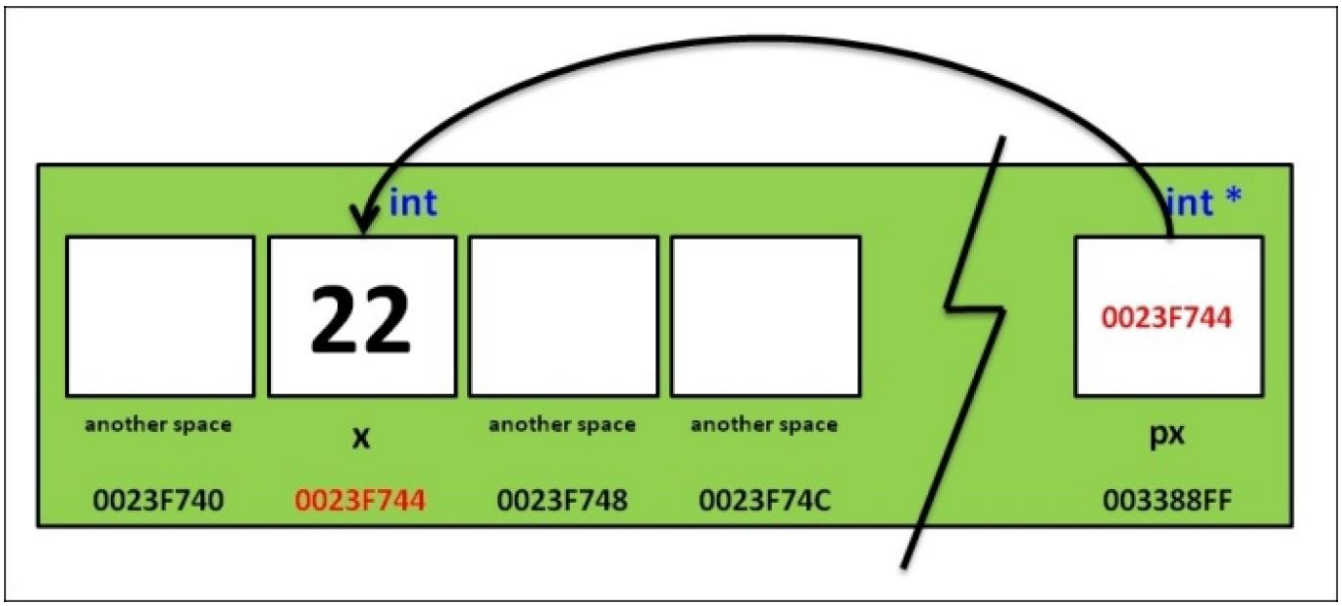
Это число (значение принадлежащее &) просто клетка памяти, в которой хранится переменная x. Это означает что именно в этом запуске программы, переменная x расположена в клетке памяти под этим номером 0023F744, как показано на следующем изображении (где another space – другое пространство):



Теперь, создайте и назначьте переменную указателя на адрес x:

```
int *px;
px = &x;
```

Что мы делаем здесь? Сохраняем адрес памяти x в переменной px. Таким образом, мы буквально указываем на переменную x совсем другую переменную именуемую px. Это может выглядеть, как показано на следующем изображении:

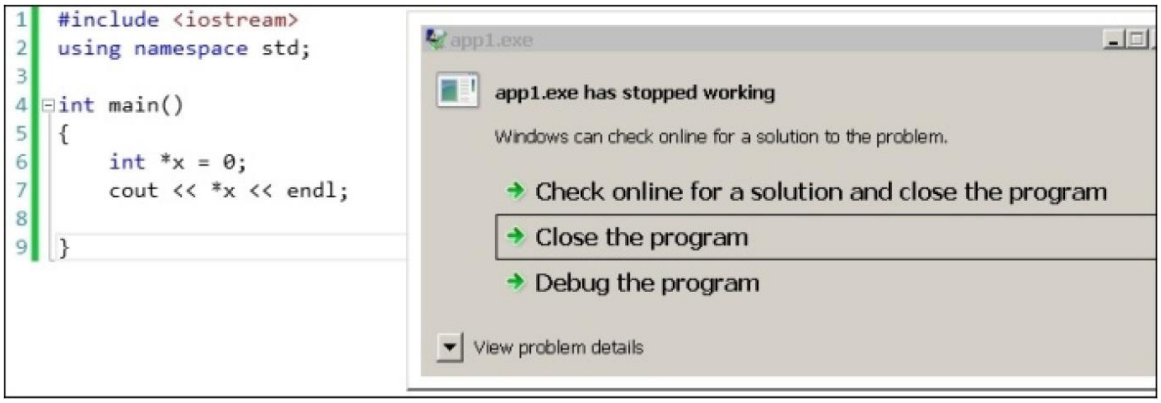


Здесь переменная px содержит в себе адрес переменной x. Другими словами, переменная px является ссылкой на другую переменную. Дифференцирование px означает доступ к переменной, на которую и ссылается px. Дифференцирование выполнено используя знак *:

```
cout << *px << endl;
```

Указатели Null

Нулевой указатель – это переменная указателя со значением 0. В основном, большинство программистов любят присваивать указателю начальное значение Null (0), при создании новой переменной указателя. Компьютерные программы в основном не могут получить доступ адресу памяти 0 (он занят), так что если вы попытаетесь обратиться к указателю Null, ваша программа выйдет из строя, как показано на следующем скриншоте:



Совет

Приколы с указателями от Binky, весёлое видео про указатели. Посмотрите на http://www.youtube.com/watch?v=i49_SNt4yfk.

cin

cin – это способ которым C++ традиционно принимает ввод от пользователя в программу. cin легко использовать, потому что оно смотрит на тип переменной, в которую будет помещать значение, когда помещает это значение. Например, скажем мы хотим спросить возраст пользователя и сохранить его в переменной типа int. Мы можем сделать это следующим образом:

```
cout << "What is your age?" << endl;
int age;
cin >> age;
```

printf()

Хотя мы применили cout, чтобы выводить переменные, вам необходимо знать о ещё одной распространённой функции, которая применяется для вывода на консоль. Эта функция называется printf. Функция printf включена в библиотеку <iostream>, так что вам нет надобности включать - #include что то дополнительное, чтобы применять её. Некоторые программисты в индустрии игр предпочитают printf вместо cout (я знаю), так что позвольте представить вам это.

Давайте продолжим и посмотрим, как работает printf(), как показано в следующем коде:

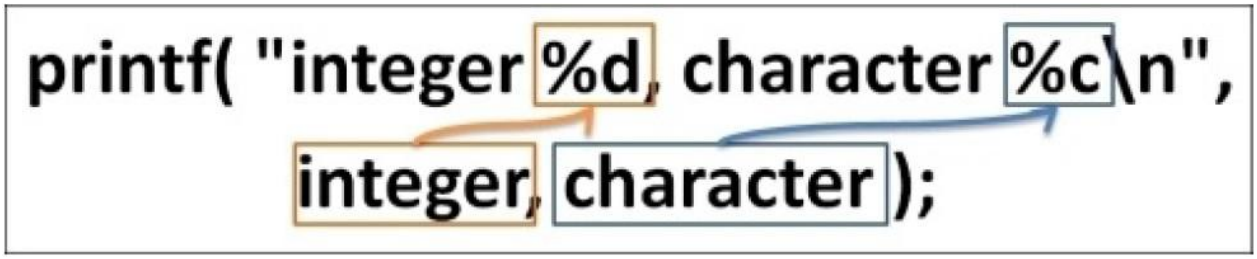
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
	char character = 'A';
	int integer = 1;
	printf( "integer %d, character %c\n", integer, character );
}
```

Совет

Скачивание примера кода

Вы можете скачать файл образца кода со своего аккаунта http://www.packtpub.com, для всех книг от Packt Publishing, которые вы купили. Если вы купили их где то ещё, вы можете посетить http://www.packtpub.com/support и зарегистрироваться там, чтобы получить файлы, которые будут присланы вам прямо на электронную почту.

Мы начнём со строкового формата. Формат string подобен кадру рисунка, а переменные будут вставлены в локацию от % в формате string. Затем, всё будет выдано на консоль. В предыдущем примере, целочисленная переменная будет вставлена в локацию первого % (%d), а знак будет вставлен в локацию второго % (%c), как показано на следующем скриншоте:



Вам необходимо использовать верный форматный код, чтобы получить вывод формата правильно. Посмотрите на следующую таблицу:

Тип данных	Код формата
Int	%d
Char	%c

String	%s

Чтобы вывести строку в C++, вы должны использовать функцию string.c_str():

```
string s = "Hello";
printf( "string %s\n", s.c_str() );
```

Если вы используете неверный форматный код, то вывод не появится должным образом, либо программа может выйти из строя.

Упражнение

Спросите у пользователя его имя и возраст, и примите их используя cin. Затем, выдайте ему приветствие на консоли, используя printf() (не cout).

Решение

Вот как программа будет выглядеть:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
	cout << "Имя?" << endl;
	string name;
	cin >> name;
	cout << "Возраст?" << endl;
	int age;
	cin >> age;
	cout << "Привет " << name << " я смотрю тебе исполнилось " << age << "лет.
	Поздравляю." << endl;
}
```

Подсказка

Строка на самом деле объектный тип. Внутри неё просто цепочка знаков!

Выводы

В этой главе мы говорили о переменных и памяти. Мы поговорили о математических операциях над переменными, и о тм насколько они просты в C++.

Мы также обсудили, насколько сложными могут быть выстроены типы данных, используя комбинации более простых типов данных, таких как плавающие типы, целочисленные типы и знаковые. Такие конструкции называются объектами.

Глава 3. If, Else и Switch

В предыдущей главе мы обсудили важность памяти и то, как она может использоваться для хранения данных в компьютере. Мы поговорили о том, как ваша программа занимает память, используя переменные, и как мы можем включать различные типы информации в наши переменные.

В этой главе мы поговорим о том, как контролировать поток нашей программы и как мы можем менять результат выполнения кода, разветвляя код, используя операторы контроля выполнения. Здесь мы обсудим различные типы контроля выполнения:

- Утверждения с If
- Как проверять равенство, применяя оператор ==
- Утверждения с Else
- Как проверять неравенства (а именно, как проверять больше ли одно число другого, либо меньше ли одно число другого, используя операторы: >, >=, <, <=, !=)
- Использование логических операторов (таких как: не (!), и (&&), или (||))
- Наш первый пример проекта с Unreal Engine
- Разветвление более чем на пути:
  - Утверждение с else, if
  - Утверждение с оператором switch

Разветвление

Компьютерный код, который мы писали в Главе 2, *Переменные и память*, выполнялся в одном направлении: прямо вниз. Иногда, нам может понадобиться пропускать части кода. Нам может понадобиться, чтобы код мог выполняться не в одном направлении, а разветвляться. Схематически, мы можем представить это следующим образом:

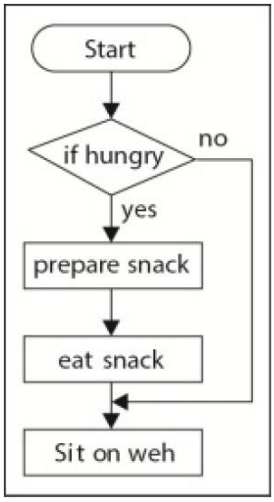


Схема процесса: Старт – если голоден – готовит перекусить – ест – присаживается на диван; если нет – (то сразу) присаживается на диван

Другими словами, нам нужен вариант выбора, не запускать определённые строки при определённых условиях. Предыдущее изображение называется схема процесса. Согласно этой схеме, если (if) и только если мы голодны, тогда мы приготовим бутерброд, съедим его, а затем пойдём, приляжем отдохнуть. Если мы не голодны, тогда нет нужды делать бутерброд и мы просто отдохнём на кушетке.

Мы будем прибегать к схемам процесса лишь иногда, но в UE4 вы можете использовать схемы процесса даже, чтобы программировать вашу игру (а именно использовать то, что называется blueprint – план).

Примечание

Эта книга о коде C++, так что мы всегда будем преобразовывать наши схемы процесса в код C++ в этой книге.

Контролирование выполнения вашей программы

В конечном итоге, мы хотим, чтобы код ответвлялся в одну сторону при определённых условиях. Команды кода, которые меняют последовательность выполнения строк, называются операторами управления потоком. Самый базовый оператор управления потоком или выполнением, это оператор if. Для того чтобы написать код для оператора if, нам сначала нужен способ проверить значение переменной.

Итак, чтобы начать, позвольте представить знак == (оператор равенства), который используется для проверки значения переменной.

Оператор равенства ==

Для того чтобы что то сравнивать в C++, нам нужно использовать не один, а два знака равно (==), друг за другом, как показано здесь:

```
int x = 5; // как вы знаете, мы используем один знак равно
int y = 4; // для присваивания…
// но нам нужно использовать два знака равно
// чтобы проверить, равны ли переменные друг другу
cout << "Is x equal to y? C++ says: " << (x == y) << endl; // Равны ли x и y?
```

Если вы запустите этот код, вы заметите, что выводится следующее:

```
Is x equal to y? C++ says: 0
```

В C++, 1 означает true, а 0 означает false. Если вы хотите, чтобы вместо 0 и 1, показывались слова true и false, вы можете использовать потоковый манипулятор boolalpha, на строке кода cout как показано здесь:

```
cout << "Is x equal to y? C++ says: " << boolalpha << (x == y) << endl;
```

Оператор == является типом операторов сравнения. Причина, по которой C++ использует ==, чтобы проверять равенство, а не просто =, заключается в том, что мы уже используем знак = как оператор присваивания! (смотрите *Больше о переменных* в Главе 2, *Переменные и память*). Если мы применим один знак =, C++ посчитает, что мы хотим переписать x на y, а не сравнить их.

Пишем код утверждений с if

Теперь, когда у нас есть в арсенале двойной знак равно, давайте напишем код для семы процесса. Код для предыдущей схемы таков:

```
bool isHungry = true; // можем установить это на false если не
                      // голоден!
if( isHungry == true ) // просто заходит внутрь { если isHungry (голоден) true
    {
        cout << "Готовит перекусить..." << endl;
        cout << "Ест... " << endl;
    }
    cout << "Сидит на диване..." << endl;
}
```

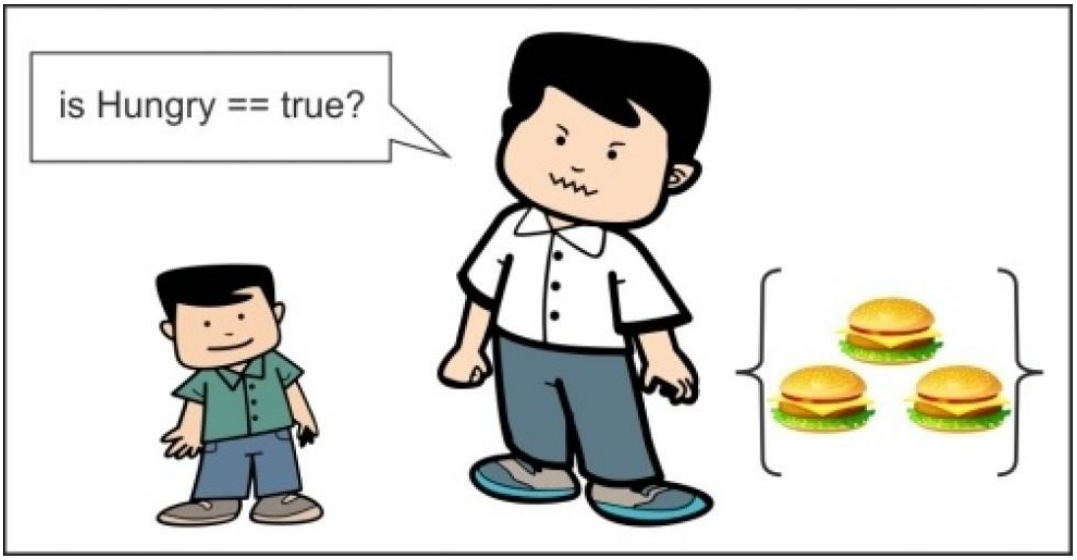
Совет

Это первый раз, когда мы используем переменную bool! Переменная bool либо содержит значение true, либо false.

Во первых, мы начинаем с переменной bool названной isHungry (являетсяГолодным) и назначаем ей true.

```
if( isHungry == true )
```

Оператор if действует как охранник для блока кода под ним. (Запомните, что блок кода это группа заключённая между фигурных скобок { и }.)



Вы можете читать код между { и }, если только isHungry == true

Код внутри фигурных скобок вы сможете получить лишь, если isHungry == true. В противном случае, в доступе вам будет отказано и вы будете вынуждены пропустить весь этот блок кода.

Совет

Мы можем получить такой же эффект, просто написав следующую строку кода:

if(isHungry) // иди только сюда, если isHungry будет true

Это может быть применено как альтернатива для следующего:

if(isHungry == true)

Причина, по которой программисты могут использовать форму if(isHungry), заключается в желании избежать возможности совершения ошибок. Если случайно написать if(isHungry = true), isHungry будет устанавливаться на true каждый раз при встрече с утверждением с if! Во избежание такой возможности, вместо этого мы можем просто написать if(isHungry). Альтернативно, некоторые (мудрые) программисты используют то, что называется условия Yoda, чтобы проверять утверждения с if: if(true == isHungry). Причина писать утверждение с if таким образом, в том, что если мы случайно напишем if(true = isHungry), то компилятор выявит ошибку, и мы узнаем где она.

Попробуйте запустить этот сегмент кода, что бы увидеть, что я имею в виду:

```
int x = 4, y = 5;
cout << "Is x equal to y? C++ says: " << (x = y) << endl; //bad!
// строка сверху переписывает значение в x на то, что было в y,
// и так как строка сверху содержит присвоение x = y
// нам следовало бы использовать (x == y) теперь.
cout << "x = " << x << ", y = " << y << endl;
```

Следующая строка показывает вывод предыдущей строки кода:

```
Is x equal to y? C++ says: 5
x = 5, y = 5
```

Строка кода, которая имеет (x = y) переписывает предыдущее значение x (которое было 4) на значение y (которым является 5). И хотя мы пытались проверить, равны ли переменные x и y, в предыдущем утверждении, переменной x было назначено значение переменной y.

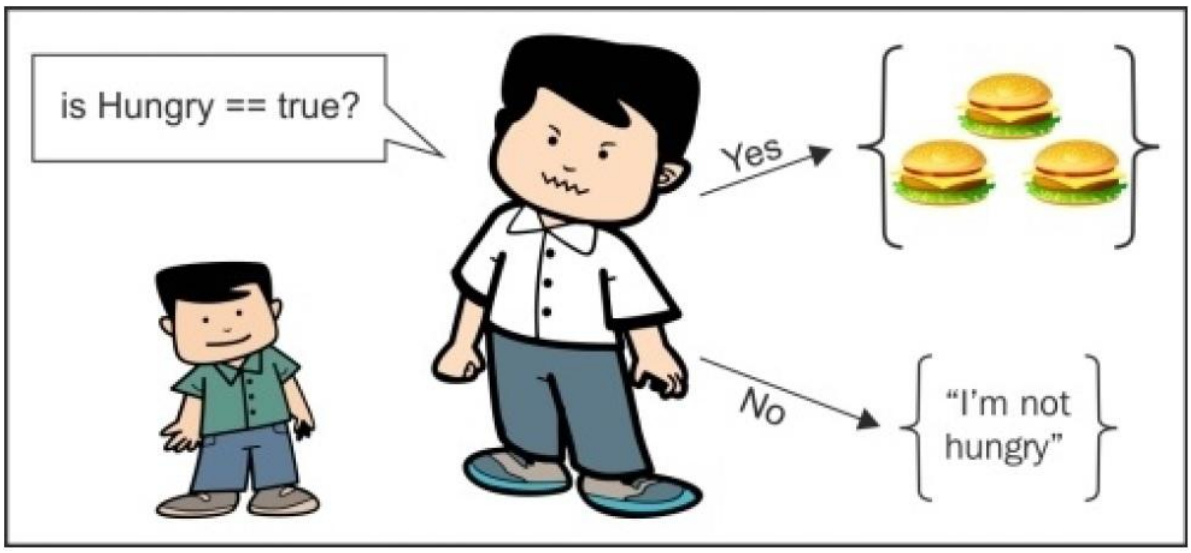
Пишем код с оператором Else

Оператор else используется для того, чтобы в нашем коде были ещё какие-то действия, в случае если часть оператора if не задействована.
Например, у нас есть ещё что то, что мы хотели бы сделать, в случае если мы не голодны, как показано в следующей части кода:

```
bool isHungry = true;
if( isHungry ) // обратите внимание подразумевается == true!
{
        cout << "Готовит перекусить..." << endl;
        cout << "Ест... " << endl;
}
else            // мы идём сюда, если isHungry будет FALSE
{
    cout << "I'm not hungry" << endl; // Я не голоден
}
    cout << "Сидит на диване..." << endl;
}
```

Вот пара важных пунктов о ключевом слове else, которые вам нужно запомнить:

- Оператор else всегда должен идти сразу за оператором if. У вас не должно быть каких-либо дополнительных строк между завершением блока if и соответствующим блоком else.
- Вы никогда не должны идти в оба блока if и else. Всегда запускается либо один, либо другой.



Утверждение с else – это путь, по которому вы идёте, если isHungry не равно true

Вы можете представить операторы if/else, неким стражем, который направляет людей либо направо, либо налево. Каждый человек пойдёт либо к еде (когда isHungry == true), либо от еды (когда isHungry == false).

Проверка неравенств с применением других операторов сравнения (>, >=, <, <=, и !=)

Другие логические операторы сравнения могут быть с лёгкостью применены в C++. Знаки > и < означают то же что и обычно в математике. Соответственно это знаки больше чем (>) и меньше чем (<). Знак >= означает тоже, что и этот знак больше

или равно ≥ в математике. Этот знак <=, подразумевает знак меньше или равно ≤ для кода C++. И так как у нас на клавиатуре нет ни этого ≥, и ни того знака ≤, нам приходится писать код C++ с применением двух знаков (>=,<=) для этих обозначений. != - это как мы говорим “не равно” в C++. Итак, скажем у нас есть следующие строки кода:

```
int x = 9;
int y = 7;
```

Мы можем спросить компьютер x > y или x < y, как показано здесь:

```
cout << "x больше чем y? " << (x > y) << endl;
cout << "x больше чем ИЛИ РАВЕН y? " << (x >= y) << endl;
cout << "x меньше чем y? " << (x < y) << endl;
cout << "x меньше чем ИЛИ РАВЕН y? " << (x <= y) << endl;
cout << "x не равен y? " << (x != y) << endl;
```

Подсказка

Нам нужны скобки вокруг сравнений x и y, для соблюдения последовательности действий. Если у нас не будет скобок, C++ запутается между такими операторами, как << и <. Это имеет определённый смысл и вы поймёте его позже. Но вам необходимо, чтобы C++ оценивало сравнение (x < y) перед тем как вы выведите результат (<<). Здесь есть отличная таблица, к которой можно обращаться: http://en.cppreference.com/w/cpp/language/operator_precedence.

Использование логических операторов

Логические операторы позволяют вам выполнять более сложные проверки, нежели обычная проверка на равность или неравность. Например, условия для входа в особую комнату, требуют от игрока наличие и красной, и зелёной ключ-карты. И мы хотим проверить, содержат ли оба условия true, в одно и то же время. А чтобы выполнить такого сложного типа проверку, существуют три дополнительных конструкции, которые нам нужно узнать: операторы *не* (!), *и* (&&), *или* (||).

Оператор Не (!)

Оператор *не* ! удобен для противоположного обращения переменной типа boolean. Посмотрите на следующий пример кода:

```
bool wearingSocks = true; // носки надеты
if( !wearingSocks ) // тоже самое что и if( false == wearingSocks )
{
cout << "Надеть носки!" << endl;
} else
  {
   cout << "Вы уже в носках" << endl;
  }
```

Данное утверждение проверяет, надеты ли на вас носки. Затем вы даёте команду, надеть носки. Оператор *не* ! обращает любое значение в переменной типа boolean на противоположное.

Мы используем так называемую “таблицу правды”, чтобы показать все возможные результаты использования оператора *не* ! для булевых переменных:

носкиНадеты **wearingSocks**	**!носкиНадеты** **!wearingSocks**
true	**false**
false	**true**

Итак, в то время как wearingSocks имеет значение true, !wearingSocks имеет противоположное значение false.

Упражнения

1. Как вы думаете, каким будет значение !!wearingSocks, если значение wearingSocks будет true?
2. Каково значение переменной isVisible, после запуска следующего кода?

```
bool hidden = true;          // bool спрятан = true
bool isVisible = !hidden;    // bool являетсяВидимым = !спрятан
```

Решение

1. Если wearingSocks равно true, тогда !wearingSocks равно false. Из чего следует, что !!wearingSocks снова обретает значение true. Это как сказать:”Я не не голоден”. "Не не" даёт двоенное отрицание, так что это предложение означает, что я вообще то голоден.
2. Ответ на второй вопрос будет false. Переменная hidden имела значение true, так что !hidden даёт false. После чего false сохраняется в переменной isVisible.

Подсказка

Оператор *не* ! иногда в просторечии называется bang (что либо внезапное, как хлопок). Предыдущая операция bang bang (!!) является двойным негативом и двойной логической инверсией. Если вы применили bang-bang для булевой переменной, то нет никого цепного изменения переменной. Если вы примените bang-bang для целочисленной переменной (int), то эта переменная просто станет

булевой переменной (true или false). Если значение целочисленной переменной больше нуля, оно упрощается до значения true. Если значение целочисленной переменной уже является 0, оно упрощается до значения false.

Оператор И (&&)

Скажем, мы хотим запустить секцию кода, если верны (true) два условия. Например, мы полностью оденемся только, если на нас надеты и носки и остальная одежда. Вы можете использовать следующий код, чтобы проверить это:

```
bool wearingSocks = true;           // носки надеты
bool wearingClothes = false;        // одежда надета
if( wearingSocks && wearingClothes )// Знак и && требует, чтобы ОБЕ переменные имели true
{
   cout << "Вы одеты!" << endl;
}
else
{
    cout << "Вы ещё не одеты" << endl;
}
```

Оператор Или (||)

Иногда мы хотим запустить секцию кода, если хотя бы одна из переменных имеет значение true.

Итак, например, игрок выигрывает определённый бонус, если он находит специальную звезду в уровне, либо если он проходит уровень меньше чем за 60 секунд. В этом случае вы можете использовать следующий код:

```
bool foundStar = true;                  // найдена Звезда
float levelCompleteTime = 25.f;   // время Завершения Уровня
float maxTimeForBonus = 60.f;   // максимальное Время Для Бонуса
  // Знак или || требует, чтобы ХОТЬ ОДНА переменная имела true, чтобы попасть внутрь {
if( foundStar || levelCompleteTime < maxTimeForBonus )
{
    cout << "Награждаетесь бонусом!" << endl;
}
else
{
   cout << "Никакого бонуса." << endl;
}
```

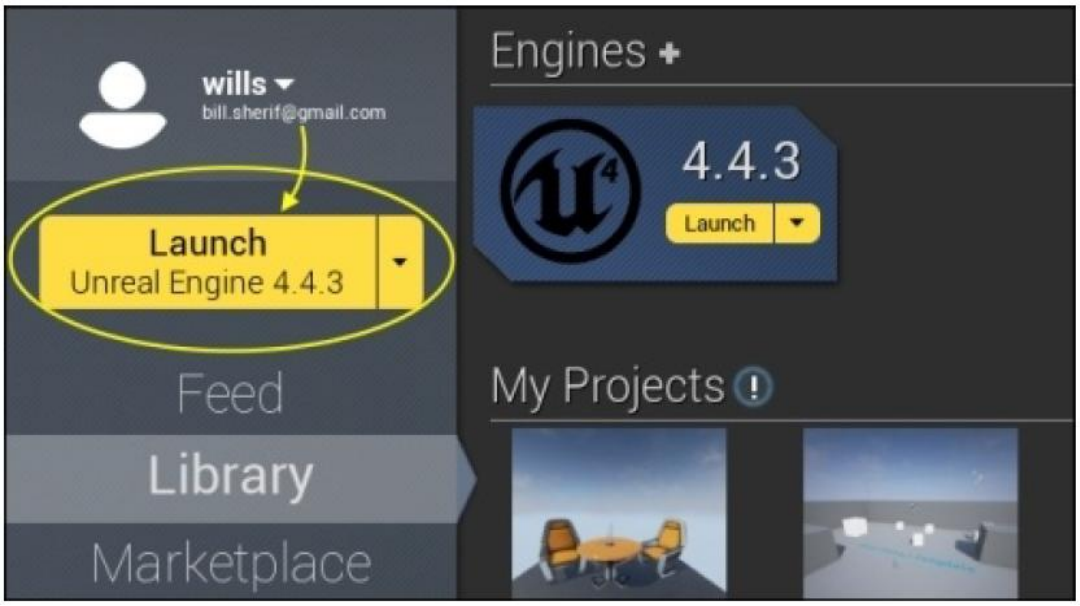
Наш первый пример с Unreal Engine

Нам нужно приступать к Unreal Engine.

Совет

На заметку: когда вы откроете ваш первый проект Unreal, вы обнаружите, что код выглядит очень сложным. Но не унывайте. Просто сфокусируйтесь на выделенных частях. В течении вашей карьеры программиста, вы будете часто иметь дело с очень большим базами кода, содержащими секции которые вы не понимаете. Тем не менее, фокусирование на частях которые вы понимаете, сделает эти секции продуктивными.

Откройте приложение **Unreal Engine Launcher** (у него значок UE4 синего цвета). Выберите **Launch Unreal Engine 4.4.3**, как показано на следующем скриншоте:

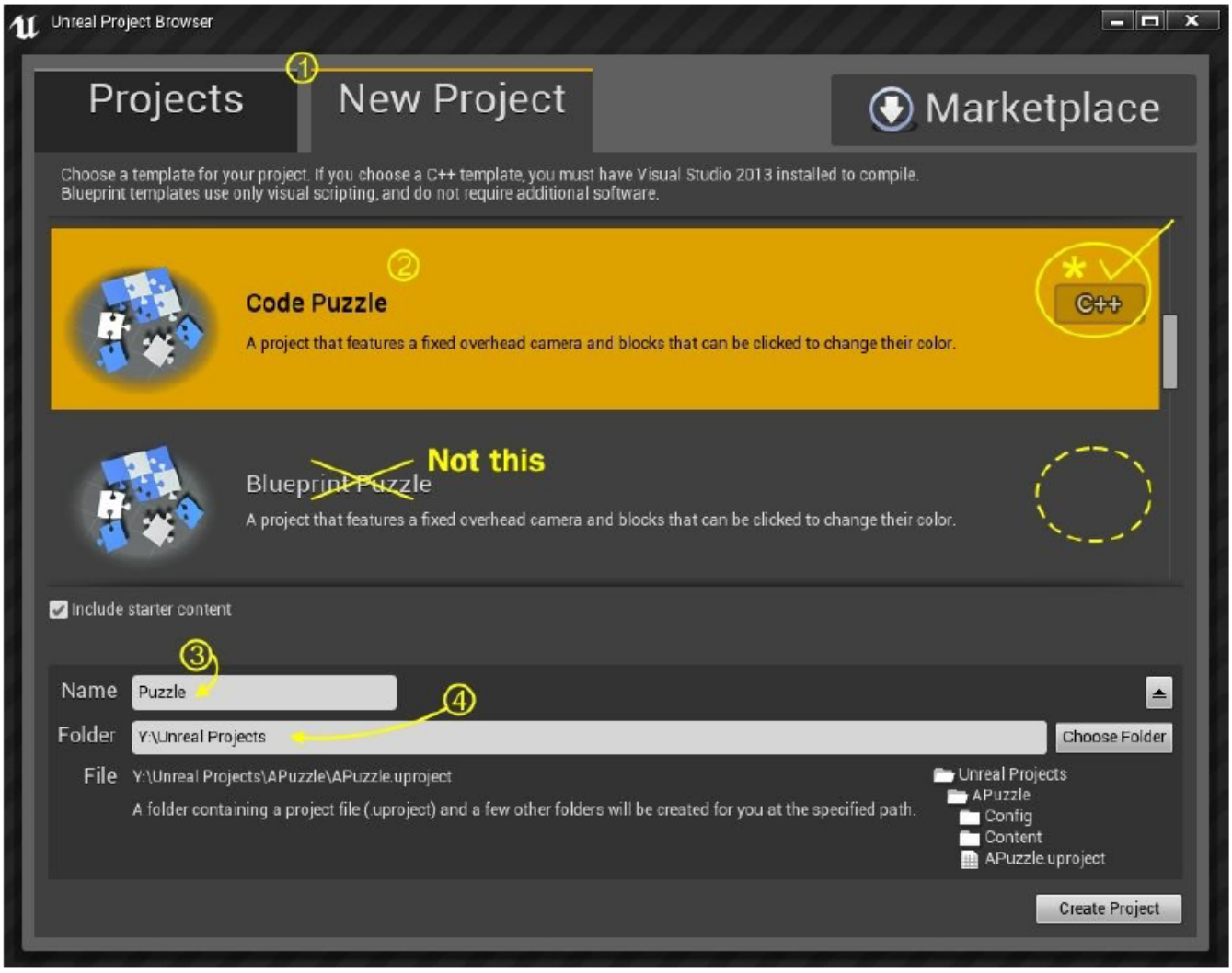


Совет

Если кнопка **Launch** не доступна, вам нужно перейти во вкладку **Library** и скачать движок (~3 ГБ).

Как только движок запущен (что может занять несколько секунд), вы будете на экране **Unreal Project Browser** (значок UE4 чёрного цвета), как показано на следующем скриншоте.

Теперь выберите вкладку **New Project** в браузере проектов UE4. Прокручивайте вниз пока не дойдёте до **Code Puzzle**. Это один из наиболее простых проектов, в котором не так много кода. Так что с него хорошо начинать. Мы дойдём до 3D проектов немного позже.



Вот несколько пунктов, на которые следует обратить внимание на этом экране:

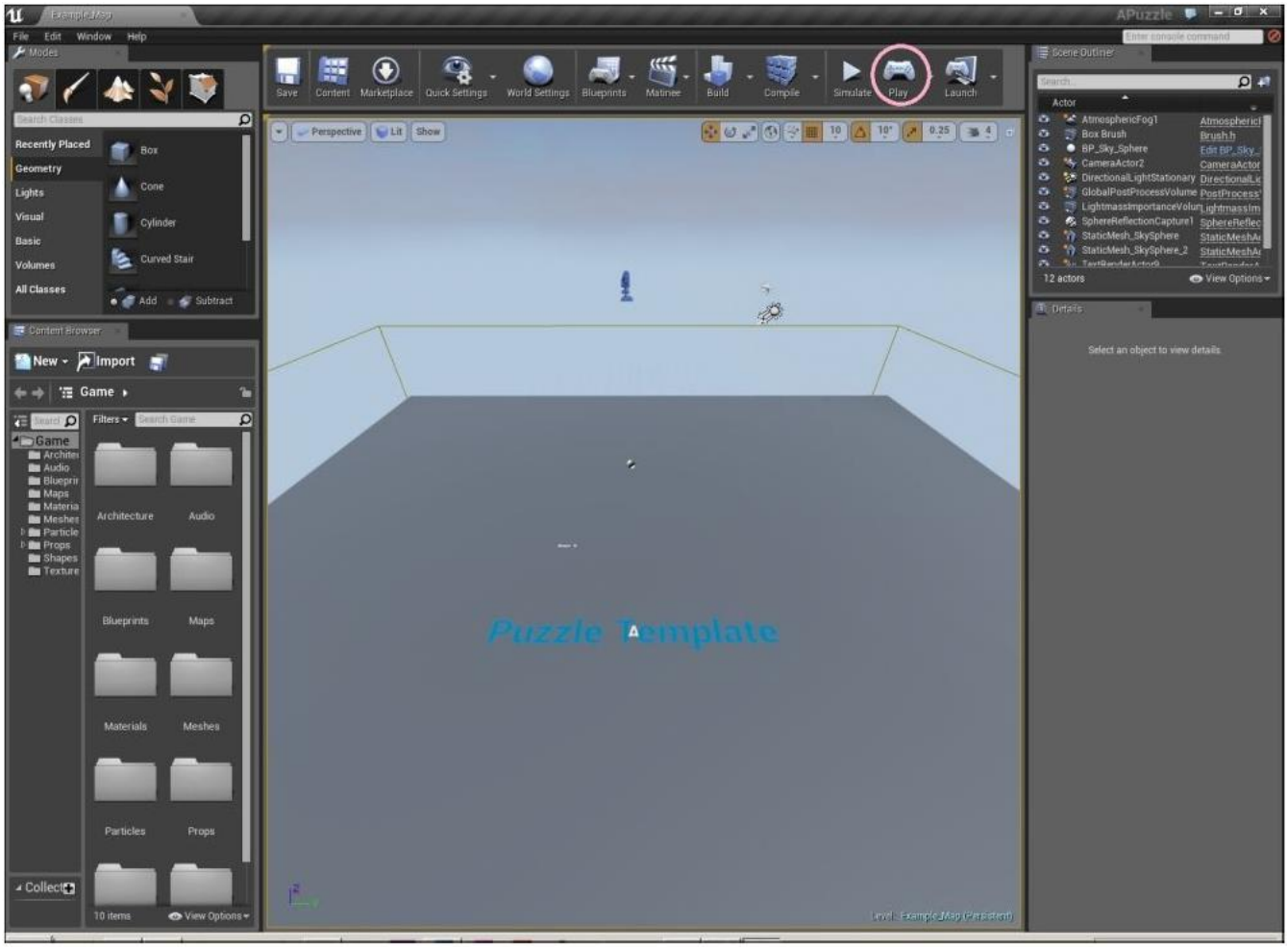
- Убедитесь что вы во вкладке **New Project**
- Когда вы нажимаете на **Code Puzzle**, убедитесь что это тот проект со значком **C++** справа, а не **Blueprint Puzzle**
- Введите имя Puzzle для вашего проекта, в поле **Name** (это важно для примера кода, который я дам вам далее)
- Если вы хотите изменить папку хранения (на другую), щёлкните по стрелке вниз. Так появятся папки. Затем, укажите папку, в которой вы хотите хранить ваш проект.

После того как вы сделаете всё это, выберите **Create Project**.

Visual Studio 2013 откроется с кодом вашего проекта.

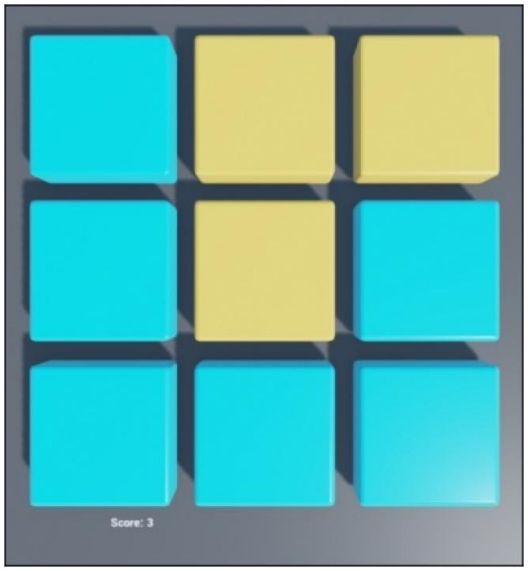
Нажмите *Ctrl+F5*, чтобы построить и запустить проект.

Как только проект компилируется и запустится, вы должны увидеть Unreal Editor, как показано на следующем скриншоте:



Выглядит сложно? Ну конечно! Мы исследуем кое-что из функционала панели инструментов немного позже. А сейчас, просто нажмите **Play** (обведён розовым), как показано на скриншоте выше.

Это запускает игру. Вот как она должна выглядеть:



Теперь попробуйте пощёлкать по блокам. Как только вы щёлкните по блоку, он станет жёлтым и это повысит ваш счёт.

Мы собираемся найти секцию, которая выполняет это, и немного изменить поведение

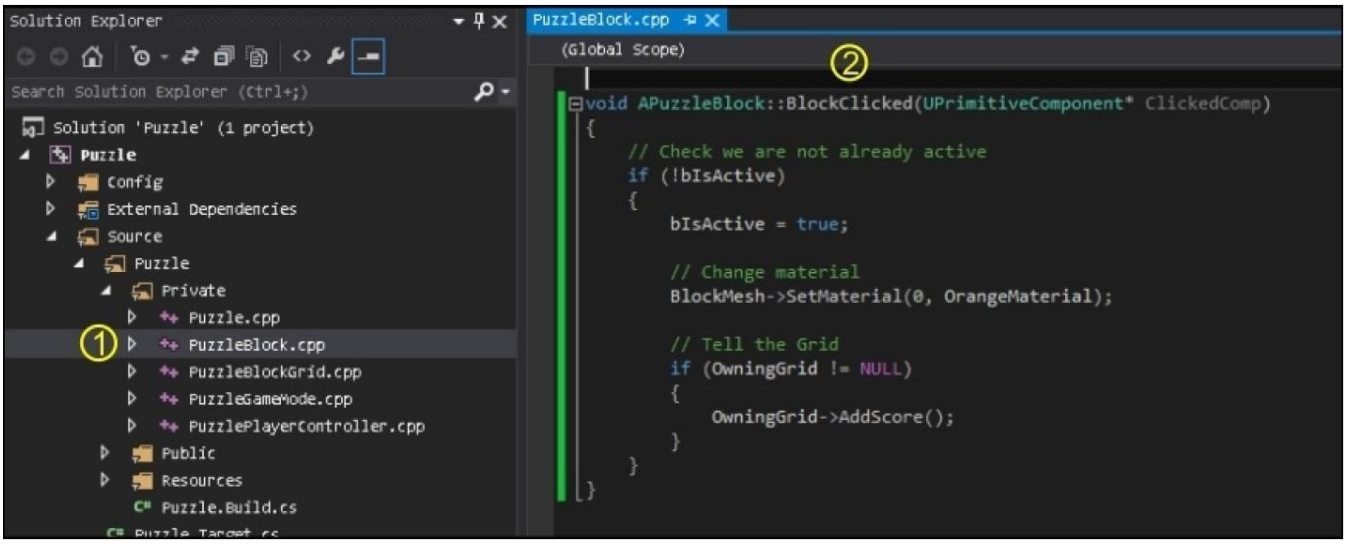
Найдите и откройте файл PuzzleBlock.ccp.

Подсказка

В Visual Studio список файлов проекта находится в **Solution Explorer**. Если ваш **Solution Explorer** скрыт, просто щёлкните по **View/ Solution Explorer** вверху меню.

В этом файле, прокрутите вниз до конца, где вы найдёте секцию, начинающуюся следующим образом:

void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp)



APuzzleBlock – это имя класса, а BlockClicked – это имя функции. Когда вы щёлкаете по блоку пазла, запускается секция кода начинающаяся с { и заканчивающаяся }. Надеемся, что манера, в которой это происходит, прояснит что то дальше.

Это напоминает манеру оператора if. Если деталь пазла была нажата, то эта часть кода запускается для этого элемента пазла.

Мы пройдём каждый шаг для смены цвета блока после щелчка по нему (так второй щелчок будет менять цвет обратно с жёлтого на голубой).

Выполните следующие шаги с максимальным вниманием:

1. Откройте файл PuzzleBlock.h. После строки 25 (на которой находится этот код):

```
/** Указатель на жёлтый материал используется на активных блоках */
UPROPERTY()
class UMaterialInstance* OrangeMaterial;
```

Введите следующий код, после предыдущей строки кода:

```
UPROPERTY()
class UMaterialInstance* BlueMaterial;
```

2. Теперь, откройте файл PuzzleBlock.ccp. После строки 40 (на которой находится этот код):

```
// Сохраните указатель на жёлтый материал
OrangeMaterial = ConstructorStatics.OrangeMaterial.Get();
```

Введите следующий код, после предыдущей строки кода:

```
BlueMaterial = ConstructorStatics.BlueMaterial.Get();
```

3. Наконец, в PuzzleBlock.ccp замените содержание секции кода (строка 44) APuzzleBlock::BlockClicked на следующий код:

```
void APuzzleBlock::BlockClicked(UPrimitiveComponent* ClickedComp)
{
    // --ЗАМЕНЯЙТЕ ОТСЮДА--
    bIsActive = !bIsActive; // flip the value of bIsActive
    // (если это было true, то станет false или наоборот)
    if ( bIsActive )
    {
        BlockMesh->SetMaterial(0, OrangeMaterial);
    }
    else
        {
            BlockMesh->SetMaterial(0, BlueMaterial);
        }
        // Сообщите Grid (сетку)
        if(OwningGrid != NULL)
        {
            OwningGrid->AddScore();
        }
    // --ДО СЮДА--
}
```

Подсказка

Замените только внутри утверждения void APuzzleBlock::BlockClicked (UPrimitiveComponent* ClickedComp).

Не заменяйте строку, которая начинается с void APuzzleBlock::BlockClicked. Вы можете получить ошибку (если вы не назвали свой проект Puzzle). Вы были предупреждены.

Итак, давайте проанализируем это. Вот первая строка кода:

```
bIsActive = !bIsActive; // меняет значение переменной bIsActive
```

Эта строка кода просто меняет значение в bIsActive. Переменная bIsActive булевая (она создана в APuzzleBlock.h). Если bIsActive будет true, то !bIsActive будет false. Итак, если эта строка кода запущена (что происходит при нажатии на любой блок), значение в bIsActive меняется на противоположное (с true на false, либо с false на true).

Давайте рассмотрим следующий блок кода:

```
if ( bIsActive )
{
  BlockMesh->SetMaterial(0, OrangeMaterial);
}
else
{
   BlockMesh->SetMaterial(0, BlueMaterial);
}
```

Мы просто меняем цвет блока. Если bIsActive будет true, то блок становится жёлтым. В противном случае, блок становится голубым.

Упражнение

Теперь, вы уже должны были заметить, что самый лучший способ повысить навыки программирования это программировать. Вам нужно много практиковаться, чтобы значительно продвинуться в программировании.

Создайте две целочисленные переменные, x и y. И заполните их от имени пользователя. Напишите утверждение if/else, которое выводит имя переменной с наибольшим значением.

Решение

Решение предыдущего упражнения показано в следующем блоке кода:

```
int x, y;
cout << "Введите два целых числа разделённых пробелом " << endl;
cin >> x >> y;
if( x < y )
{
   cout << "x меньше чем y" << endl;
} else
   {
      cout << "x больше чем y" << endl;
   }
```

Подсказка

Не пишите букву, в то время как cin ожидает число. Если это случится, cin может совершить ошибку и выдать плохое значение для вашей переменной.

Разветвление кода более чем в двух направлениях

В предыдущих частях мы могли ответвлять код только в одном из двух направлений. В псевдокоде у нас есть следующий код:

```
if( какое то условие является true )
{
   выполняется это;
} else // в другом случае
    {
        выполняется вот это;
    }
```

Подсказка

Псевдокод – это фиктивный код. Написание псевдокода это отличный метод мозгового штурма и планирования вашего кода, особенно если вы не особо освоились в C++.

Этот код отчасти похож на развилку дороги, где выбирать приходится лишь одно направление из двух.

Иногда нам может понадобиться разветвить код в трёх направлениях или даже больше. К примеру, направления, в которых может пойти код в зависимости от того, какой предмет в данный момент держит игрок. Игрок, возможно будет держать один из трёх предметов, таких как: монета, ключ или морской ёж. И C++ позволяет это! По факту, вы можете разветвлять код, в каком угодно количестве направлений.

Утверждения с else if

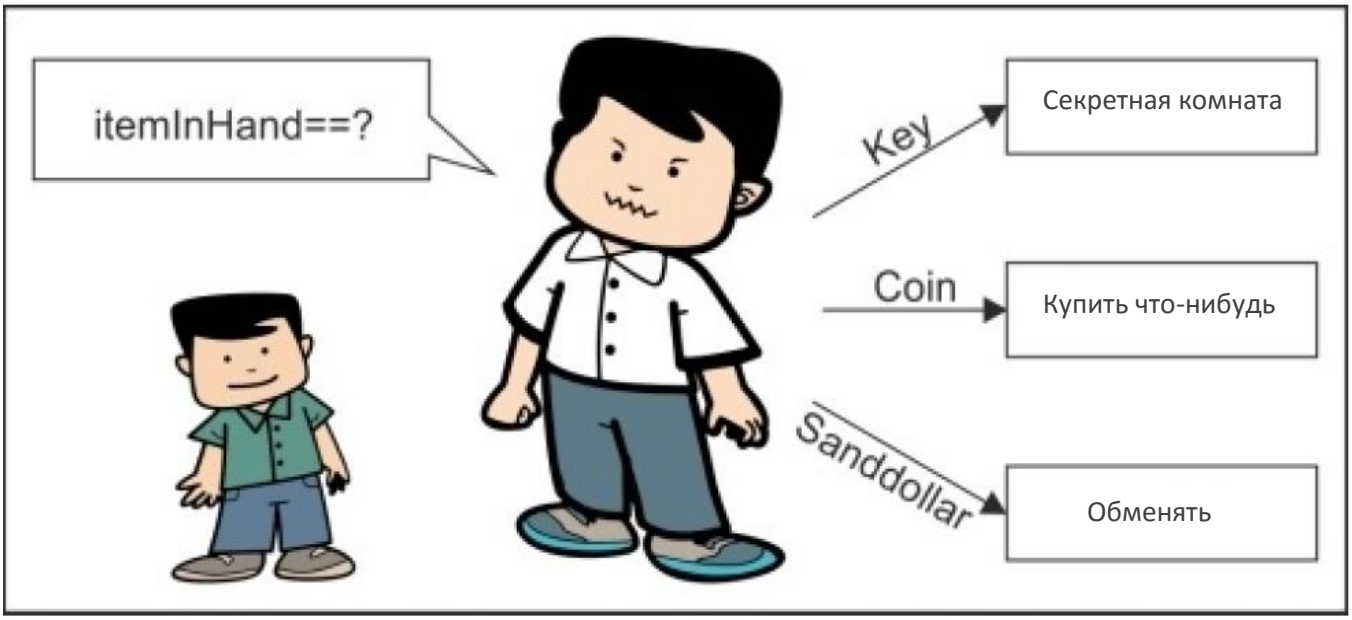
Конструкция *else if* – это способ писать код с более чем двумя возможными направлениями. В следующем примере, код пойдёт по одному из трёх различных путей, в зависимости от того держит ли игрок объект Coin (монету), Key (ключ) или Sanddollar (морской плоский щитообразный ёж):

```
#include <iostream>
using namespace std;
int main()
{
   enum Item // enum определяет новый тип переменной!
   {
      Coin, Key, Sanddollar // переменные типа Item могут содержать
                            // любое из этих трёх значений
   }
   Item itemInHand = Key; // Попробуйте поменять это значение на Coin,
                          // Sanddollar
   if( itemInHand == Key ) // (предмет В Руке == Ключ)
   {
```

```
        cout << "Ключ с основанием в форме головы льва." << endl;
        cout << "Вы попадаете в секретную комнату используя Ключ!" << endl;
    }
    else if( itemInHand == Coin ) // (предмет В Руке == Монета)
    {
        cout << "Монета проржавевшего латунного цвета. На ней изображение дамы в юбке."
        << endl;
        cout << "На эту монету вы можете купить пару вещей" << endl;
    }
    else if( itemInHand == Sanddollar ) // (предмет В Руке == Морской ёж)
    {
        cout << "На этом морском еже маленькая звёздочка." << endl;
        cout << "Возможно вы сможете обменять его на что-нибудь." << endl;
    }
    return 0;
}
```

Примечание

Обратите внимание, предыдущий код идёт только по одному из трёх отдельных путей! По серии путей проверок if, else if, и ещё по одному else if, мы войдём только в один блок кода.



Упражнение

Используйте C++ программу, чтобы ответить на следующие вопросы. Обязательно постарайтесь выполнить эти упражнения по порядку, чтобы натренировать беглость с этими операторами равенства.

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    int y;
```

```
    cout << "Введите целочисленное значение для x:" << endl;
    cin >> x; // Так значение будет считываться с консоли
                // Считываемое значение будет храниться как целочисленная
                // переменная x, так то лучше записывать целое число в знание!
    cout << "Enter an integer value for y:" << endl;
    cin >> y;
    cout << "x = " << x << ", y = " << y << endl;
    // *** Пишите новые строки кода здесь
}
```

Напишите новые строки кода с места где говорится (//*** Пишите новые…):

1. Проверьте равны ли x и y. Если они равны, выведите: x и y равны. Если нет, выведите: x и y не равны.
2. Упражнение на неравенства: проверьте больше ли x чем y. Если да, то выведите: x больше чем y. Если нет, то выведите: y больше чем x.

Решение

Чтобы выразить равенство, введите следующий код:

```
if( x == y )
{
 cout << "x и y равны " << endl;
} else
{
 cout << "x и y не равны" << endl;
}
```

Чтобы проверить какое значение больше, введите следующий код:

```
if( x > y )
{
   cout << "x больше чем y" << endl;
} else if( x < y)
    {
     cout << "y больше чем x" << endl;
    } else // в этом случае ни x > y, ни y > x
        {
         cout << "x и y равны" << endl;
        }
```

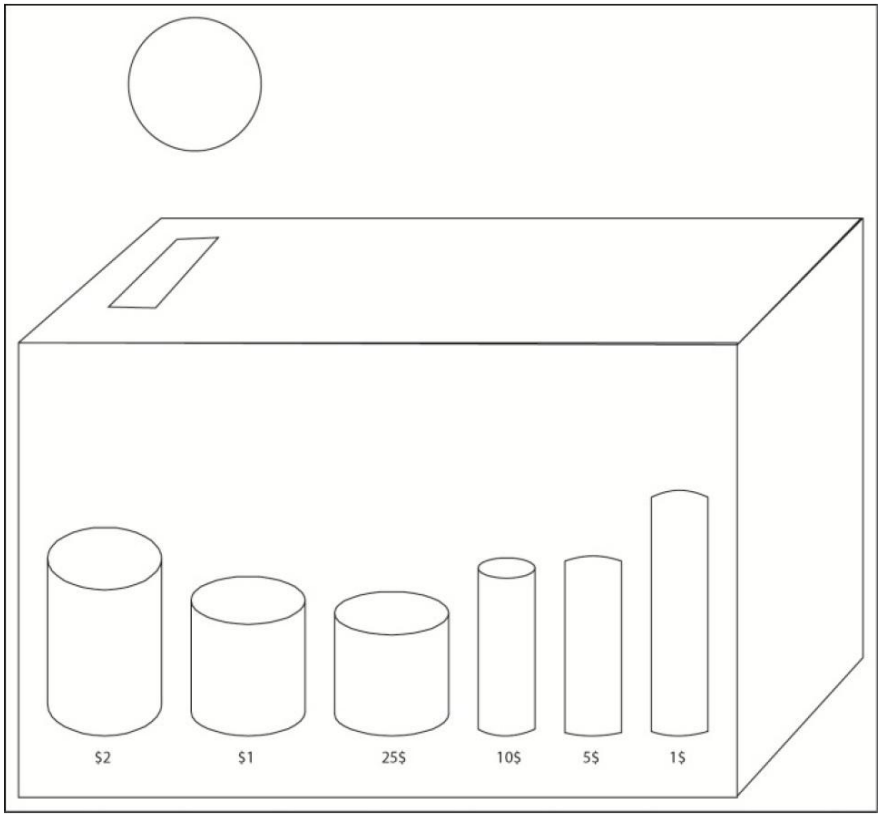
Оператор switch

Оператор switch позволяет вашему коду разветвляться по множеству путей. Оператор switch будет смотреть на значение переменной и в зависимости от этого значения, код пойдёт по одному из различных направлений.

Здесь мы также представим новую конструкцию enum:

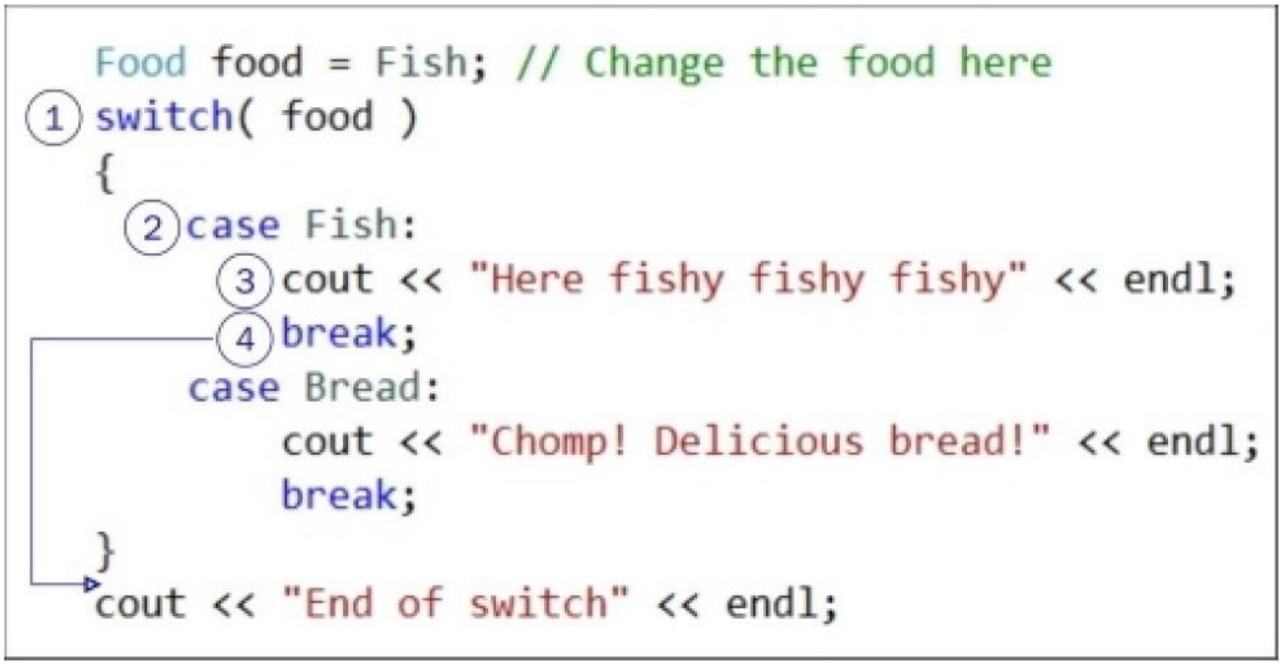
```
#include <iostream>
using namespace std;
enum Food // enum определяет новый тип переменной!
```

```
{
    // переменная типа Food может иметь любое из этих значений
    Fish,
    Bread,
    Apple,
    Orange
};
int main()
{
    Food food = Bread; // Смените еду здесь
    switch( food )
    {
      case Fish:
          cout << "Вот и рыба" << endl;
          break;
      case Bread:
          cout << "Хряп! Какой же вкусный хлеб!" << endl;
          break;
      case Apple:
          cout << "Ммм фрукты! Очень полезны." << endl;
          break;
      case Orange:
          cout << "Апельсин! Вы наверно рады, что я сказал не банан." << endl;
          break;
      default: // Сюда вы идёте в случае
                  // если не попалось ни одно из верхних условий
          cout << "Непригодная еда." << endl;
          break;
    }
    return 0;
}
```



Оператор switch как сортировщик монет. Когда вы опускаете 25 центов в сортировщик монет, он определяет путь к стопке монет номиналом в 25 центов. Подобно этому, оператор switch просто позволяет коду проскакивать к подходящей секции. Пример сортировки монет показан на изображении слева.

Код в утверждении switch будет продолжать выполняться, строка за строкой пока не дойдёт до оператора `break;`. Оператор break выводит вас из утверждения switch. Взгляните на следующую диаграмму, чтобы понять как работает switch:



1. Сначала проверяется переменная Food. Какое в ней значение? В данном случае Fish.
2. Команда switch проскакивает вниз к подходящей метке случая. (Если не будет никаких подходящих меток для этого случая, то утверждение switch просто пропускается).
3. Запускается утверждение cout и “Here fishy fishy fishy” появляется на консоли.
4. После проверки переменной и вывода ответа пользователю, срабатывает оператор break. Это заставляет нас прекратить выполнение строк кода в switch и выйти из switch. Следующая строка кода, которая будет запущена, это строка кода в программе, которая и была бы запущена, если бы и не было утверждения switch вовсе (после закрывающей фигурной скобки утверждения switch). Это утверждение для вывода внизу, которое говорит “End of switch”.

Switch против if

Switch похож на цепочку if / else if / else, что мы видели ранее. Однако, switch быстрее образует код чем цепочка if / else if / else. Интуитивно switch сразу переходит к выполнению подходящей секции кода. Цепочка if / else if / else может включать в себя более сложные сравнения (включая логические сравнения), которые занимают у ЦПУ больше времени. Главная причина того что вы будите

использовать оператор if, в том что вы можете сделать больше со своими специально подобранными сравнениями внутри скобок.

Подсказка

Оператор enum на самом деле является типом int. Чтобы подтвердить это, применим следующий код:

```
cout << "Fish=" << Fish <<
        " Bread=" << Bread <<
        " Apple=" << Apple <<
        " Orange=" << Orange << endl;
```

Вы увидите целочисленные значения enum, так что теперь вы знаете.

Иногда программистам нужно группировать множество значений под одной меткой case в switch. Скажем, у нас есть enum такого вида:

```
enum Vegetables { Potato, Cabbage, Broccoli, Zucchini };
```

Программист хочет сгруппировать всю зелень вместе, так что он пишет утверждение switch следующим образом:

```
switch( veg )
{
case Zucchini: // цукини пролетает мимо, так как break
case Broccoli: // не был написан здесь
   cout << "Зелень!" << endl;
   break;
default:
   cout << "Нет зелени!" << endl;
   break;
}
```

В этом случае Zucchini терпит неудачу и выполняется код Broccoli. Не зелёные овощи в метке случая default. Чтобы предотвратить не выполнение как в предыдущем примере, вы должны не забывать вводить завершающий оператор break после каждой метки case.

Мы можем написать ещё одну версию switch, которая не пропускает Zucchini с применением завершающего оператора break:

```
switch( veg )
{
case Zucchini: // цукини больше не пропускается, так как есть оператор break
   cout << "Цукини зелёное" << endl;
   break;// останавливает на случае цукини, не доходя до
case Broccoli: // того что написано здесь
   cout << "Брокколи зелёная" << endl;
   break;
default:
```

```
    cout << "Нет зелени!" << endl;
    break;
}
```

Заметьте, что хорошая привычка в программировании, ставить оператор break и после случая default, хоть это и последний случай в перечислении.

Упражнение

Завершите следующую программу, которая имеет объект enum с рядом верховых животных, из которых можно выбирать. Напишите утверждение switch, которое выводит следующие сообщения для выбранного животного:

Конь	Конь отважный и мощный.
Кобыла	Эта лошадь белая и красивая.
Мул	Вам дали мула для езды. Вы заслужили это.
Овца	Бее! Овца может спокойно нести вашу поклажу.
Чокобо	Чокобо!

Помните, что объект enum является на самом деле типом int. Первое вводное в enum по умолчанию 0, но вы можете дать объекту enum любое стартовое значение по своему желанию, используя оператор =. Последовательные значения в объекте enum это целые числа, расположенные по порядку.

Подсказка

Побитовый сдвиг enum

Распространённое явление с объектом enum, назначать значение побитового сдвига для каждого вводного:

```
enum WindowProperties
{
    Bordered     = 1 << 0, // бинарное 001
    Transparent = 1 << 1, // бинарное 010
    Modal        = 1 << 2 // бинарное 100
};
```

Значения побитового сдвига должны быть способны комбинировать свойства окна. Вот как назначение будет выглядеть:

```
// побитово ИЛИ комбинированием свойств
WindowProperties wp = Bordered | Modal;
```

Проверка того, какие WindowProperties (свойства окна) были установлены, включает в себя проверку с использованием побитового И:

```
// побитовое И проверяет если wp является Modal
if( wp & Modal )
{
  cout << "Вы смотрите на модальное окно" << endl;
}
```

Техника побитового сдвига слегка выходит за рамки этой книги, но я включил эту подсказку просто, чтобы вы знали об этом.

Решение

Решение предыдущего упражнения показано в следующем коде:

```
#include <iostream>
using namespace std;
enum Mount
{
    Horse=1, Mare, Mule, Sheep, Chocobo
    // Так как Horse=1, Mare=2, Mule=3, Sheep=4 и Chocobo=5.
};
int main()
{
    int mount; // Мы будем использовать переменную типа int для наших верховых животных
                // так что cin работает
    cout << "Выбирайте на ком ехать верхом:" << endl;
    cout << Horse << " Конь" << endl;
    cout << Mare << " Кобыла" << endl;
    cout << Mule << " Мул" << endl;
    cout << Sheep << " Овца" << endl;
    cout << Chocobo << " Чокобо" << endl;
    cout << "Введите число от 1 до 5, чтобы сделать выбор" << endl;
    cin >> mount;
        // Пишите своё утверждение switch здесь. Опишите что происходит
        // когда вы едете верхом на каждом животном в утверждении switch
    switch( mount )
    {
        default:
        cout << "Такого не оседлаешь" << endl;
        break;
    }
return 0;
}
```

Выводы

В этой главе вы узнали, как разветвлять код. Разветвление делает возможным развитие кода в разных направлениях, вместо строгого выполнения вниз.

В следующей главе, мы двинемся к различным операторам потокам управления, которые позволят вам возвращаться и повторять строку кода определённое количество раз. Секции кода, которые повторяются, будут называться циклами.

Глава 4. Циклы

В предыдущей главе, мы обсуждали оператор if. Оператор if даёт вам возможность указывать условия для выполнения блока кода.

Вы этой главе мы изучим циклы. Они являются структурами кода, позволяющими нам повторять блок кода при определённых условиях. Мы прекращаем повторение этого блока, как только условия становятся несоответствующими (false).

В этой главе мы изучим следующие темы:

- Цикл while
- Цикл do/while
- Цикл for
- Простой практический пример цикла в Unreal Engine

Цикл while

Цикл while (пока) применяется для повторяющегося запуска секции кода. Это удобно если у вас набор действий, которые должны выполняться повторно для достижения какой то цели. Например, в следующем коде, цикл while повторно выводит значение переменной x, по мере того, как оно возрастает на единицу с 1 до 5:

```
int x = 1;
while( x <= 5 ) // только при x<=5, тело while может быть выполнено
{
    cout << "x равен " << x << endl;
    x++;
}
cout << "Готово" << endl;
```

Вот что выведет эта программа:

```
x равен 1
x равен 2
x равен 3
x равен 4
```

```
x равен 5
Готово
```

На первой строке кода, создаётся целочисленная переменная x и ей устанавливается значение 1. Затем, мы задаём условия для цикла while. Условия while говорят, что пока x меньше или равно 5, вы должны оставаться в блоке кода, который идёт следом.

Каждая итерация цикла (итерация означает пройти цикл один раз) немного продвигает выполнение задачи (по выведению чисел от 1 до 5). Мы программируем цикл, чтобы автоматически выходить, как только задача выполнена (как только x <= 5 больше не является true).

Подобно оператор if в предыдущее главе, вход в блок под циклом while допустим только, если встречается условие указанное в скобках данного цикла (в предыдущем примере x <= 5). Вы можете попробовать мысленно заменить цикл while циклом if, как показано в следующем коде:

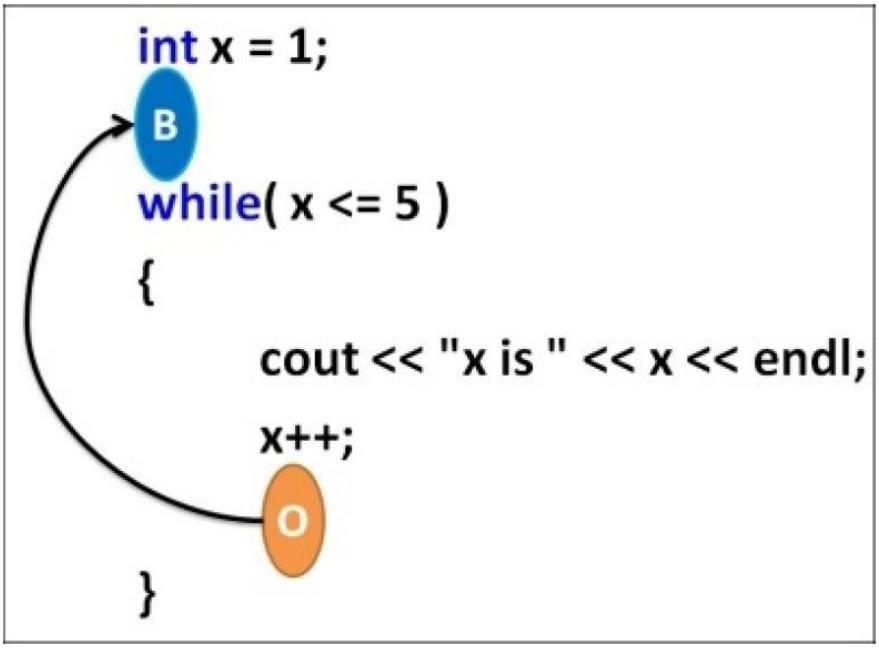
```
int x = 1;
if( x <= 5 ) // вы можете только войти в блок снизу когда x<=5
{
  cout << "x равен " << x << endl;
  x++;
}
cout << "Конец программы" << endl;
```

Этот пример кода будет выводить лишь x равен 1. Итак, цикл while такой же как утверждение с if, но в нём есть специальное автоматическое самоповторение, которое продолжается пока соблюдается условие определённое в скобках.

Примечание

Я бы хотел объяснить повторение цикла while, прибегнув к видео игре. Если вы не знаете Portal от Valve, вам нужно поиграть в неё, просто чтобы понять циклы. Посмотрите вот это https://www.youtube.com/watch?v=TluRVBhmf8w демонстрационное видео.

В циклах while есть магический *портал* внизу, благодаря которому цикл повторяется. Следующий скриншот изображает, что я имею в виду:



В конце цикла while есть портал, который возвращает вас назад к началу.

На предыдущем скриншоте, мы идём назад по циклу от оранжевого портала (помеченного буквой **O**), к синему порталу (помеченного буквой **B**). Это впервые, когда мы можем возвращаться в коде. Это как путешествие во времени, только в коде. Как здорово!

Лишь не встретив заданного условия, можно миновать блок цикла while. В предыдущем примере, как только значение становится 6 (так не соблюдается условие x <= 5,что означает false), мы больше не заходим в цикл while. Поскольку оранжевый портал внутри цикла, мы сможем завершить, как только x станет 6.

Бесконечные циклы

Вы можете навсегда остаться в цикле. Посмотрите как следует на модифицированную программу в следующем блоке кода (что вы думаете, будет выводиться?):

```
int x = 1;
while( x <= 5 ) // в тело цикла while можно войти только когда x<=5
{
  cout << "x равен " << x << endl;
}
cout << "Конец программы" << endl;
```

Вот как будет выглядеть вывод:

```
x is 1
x is 1
x is 1…
(повторяется вечно)
```

Цикл повторяется вечно, потому что мы убрали строку кода которая меняла значение переменной x. Если значение x не возрастает и остаётся тем же, мы застрянем в теле цикла while. Это потому что несоблюдение условия (значение x становится 6) не может быть встречено, если x не меняется внутри тела цикла.

В следующих упражнениях используются все принципы из предыдущих глав, такие как операции приращения += и -= отрицательного приращения (декремент). Если вы что-то забыли, то вернитесь и повторите предыдущие разделы.

Упражнения

1. Напишите цикл while, который будет выводить числа от 1 до 10.
2. Напишите цикл while, который будет выводить числа от 10 до 1 (на убывание).
3. Напишите цикл while, который будет выводить числа от 2 до 20, с инкрементацией на два (например: 2, 4, 6 и 8).
4. Напишите цикл while, который будет выводить числа от 1 до 16 и помимо того, их же в квадратной степени.

Вот пример вывода программы по примеру упражнения 4:

1	**1**
2	**4**
3	**9**
4	**16**
5	**25**

Решения

Коды решений к предыдущим упражнениям:

1. Решение цикла while, которое выводит числа от 1 до 10:

```
int x = 1;
while( x <= 10 )
{
```

```
 cout << x << endl;
 x++;
}
```

2. Решение цикла while, которое выводит числа от 1 до 10, в обратном порядке:

```
int x = 10; //  x high
while( x >= 1 ) // продолжается пока x не станет 0 или меньше
{
 cout << x << endl;
 x--; // понижает значение x на 1
}
```

3. Решение цикла while, которое выводит числа от 2 до 20, инкрементируя на 2:

```
int x = 2;
while( x <= 20 )
{
 cout << x << endl;
 x+=2; // повышает значение x на 2
}
```

4. Решение цикла while, которое выводит числа от 1 до 16 с возведением их в квадратную степень:

```
int x = 1;
while( x <= 16 )
{
 cout << x << " " << x*x << endl; // выводит значения x и их квадрат
 x++;
}
```

Цикл do/while

Цикл do/while (делать/пока) почти идентичен циклу while. Вот пример цикла do/while, который эквивалентен первому циклу while, что мы рассматривали:

```
int x = 1;
do
{
  cout << "x равен " << x << endl;
  x++;
} while( x <= 5 ); // может идти назад по циклу, если x<=5
cout << "Конец программы" << endl;
```

Единственная разница здесь в том, что нам не надо проверять условие while при нашем первом входе в цикл. Это значит, что тело цикла do/while всегда выполняется хотя бы один раз (в то время как цикл while может быть пропущен полностью, если условие для входа в цикл while даёт не true, а false, когда мы проверяем его первый раз).

Цикл for

Цикл for имеет слегка иную анатомию, чем цикл while, и оба они очень похожи.

Давайте рассмотрим анатомию цикла for сравнённую с эквивалентным циклом while. Вот пример отрывков кода:

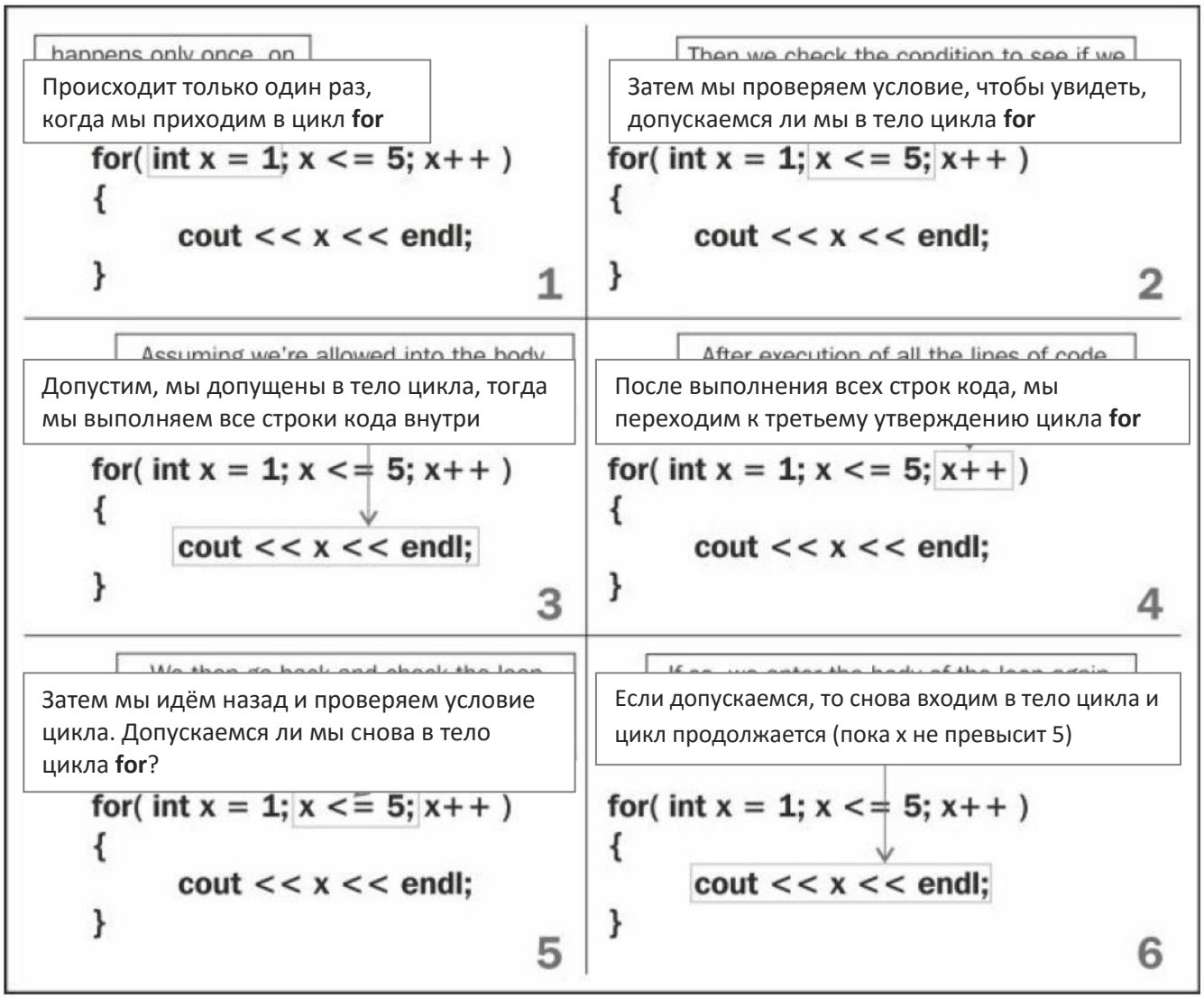
Цикл for	**Эквивалентный цикл while**

for(int x = 1; x <= 5; x++)
{
cout << x << endl;
}
	int x = 1;
	while(x <= 5)
	{
	cout << x << endl;
	x++;
	}

Внутри скобок цикла for три утверждения. Давайте рассмотрим их по порядку.

Первое утверждение цикла for (int x = 1;) выполняется лишь один раз, когда мы впервые входим в тело данного цикла. Оно обычно используется для присвоения начального значения встречаемой переменной (в данном случае x). Второе утверждение в цикле for (x <= 5) является условием повторения цикла. Пока x <= 5, мы должны оставаться в теле цикла for. Последнее утверждение внутри скобок цикла for (x++;) выполняется каждый раз после того как мы проходим тело цикла.

Следующая последовательность диаграмм объясняет прогрессию цикла for:



Упражнения

1. Напишите цикл for, который будет собирать сумму чисел от 1 до 10.
2. Напишите цикл for, который будет выводить числа от 6 до 30, с приращением 6 (6, 12, 18, 24, 30).
3. Напишите цикл for, который будет выводить числа от 2 до 100 с приращением 2 (например, 2, 4, 6, 8 и так далее).
4. Напишите цикл for, который будет выводить числа от 1 до 16 и их в квадрате.

Решения

Вот решения к предыдущим упражнениям:

1. Решение для цикла for, который выводит сумму чисел от 1 до 10:

```
int sum = 0;
for( int x = 1; x <= 10; x++ )
{
 sum += x;
```

```
 cout << x << endl;
}
```

2. Решение для цикла for, который выводит числа от 6 до 30, с приращением 6:

```
for( int x = 6; x <= 30; x += 6 )
{
 cout << x << endl;
}
```

3. Решение для цикла for, который выводит числа от 2 до 100, с приращением 2:

```
for( int x = 2; x <= 100; x += 2 )
{
 cout << x << endl;
}
```

4. Решение для цикла for, который выводит числа от 1 до 16 и их квадраты:

```
for( int x = 1; x <= 16; x++ )
{
 cout << x << " " << x*x << endl;
}
```

Выполнение цикла с Unreal Engine

В вашем редакторе кода откройте ваш проект Unreal Puzzle из Главы 3, *If, Else и Switch*.

Есть несколько способов открыть ваш проект в Unreal. Вероятно самый простой способ перейти в папку Unreal Projects (которая по умолчанию находится в вашей пользовательской папке Документы в Windows) и дважды щёлкните по файлу .sln в проводнике Windows, как показано на следующем скриншоте:

В Windows откройте файл .sln для редактирования кода вашего проекта

Теперь, откройте файл PuzzleBlockGrid.cpp. В этом файле прокрутите вниз до секции, которая начинается со следующего утверждения:

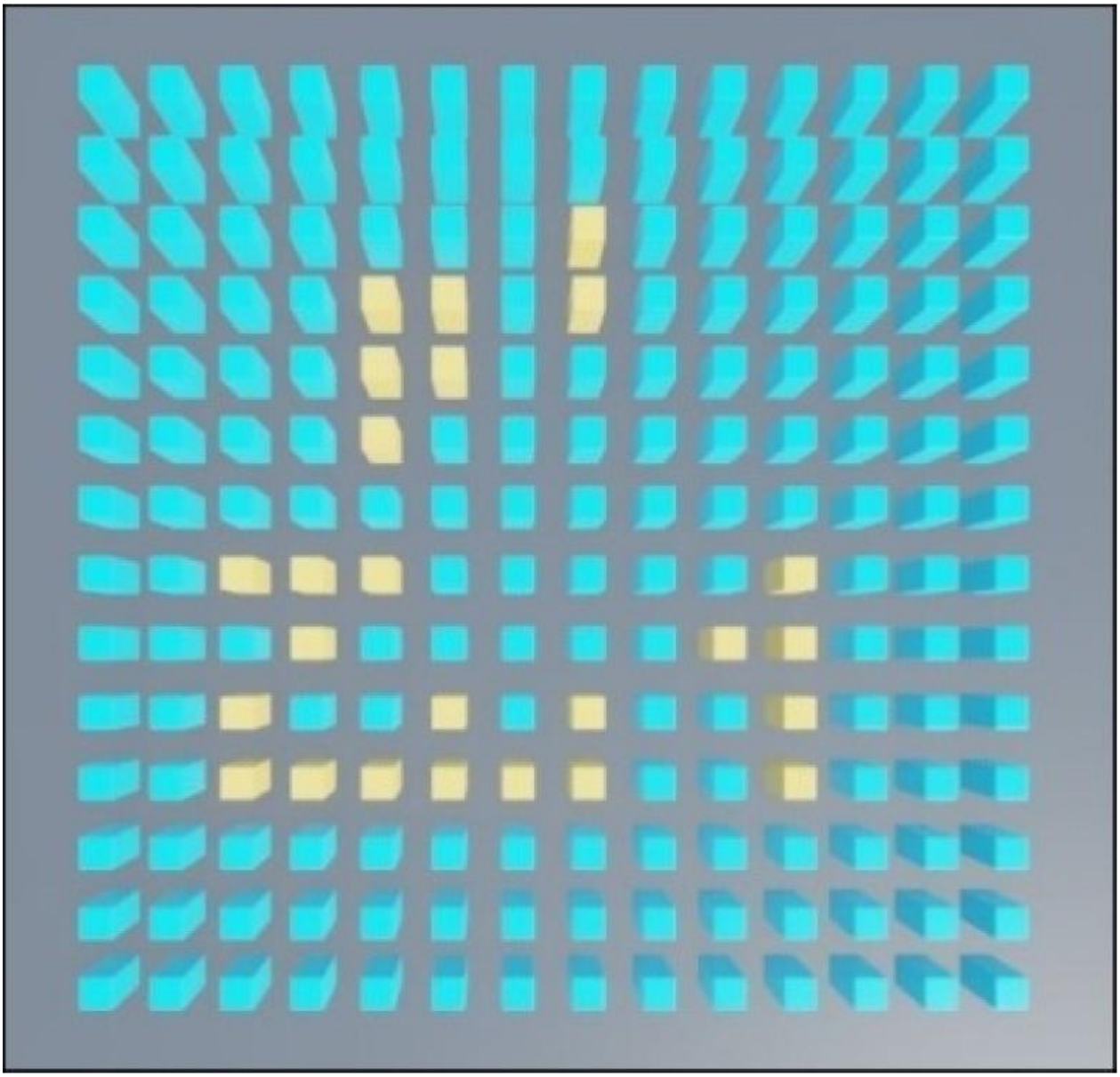
```
void APuzzleBlockGrid::BeginPlay()
```

Заметьте тут есть цикл for для порождения начальных девяти блоков, как показано в следующем коде:

```
// Цикл для порождения каждого блока
for( int32 BlockIndex=0; BlockIndex < NumBlocks; BlockIndex++ )
{
  // ...
}
```

Поскольку NumBlocks (который используется для определения когда останавливать цикл) вычисляется как Size*Size, мы можем легко изменить число порождающихся блоков сменив значение переменной Size. Перейдите на строку 23 файла PuzzleBlockGrid.cpp и измените значение переменной Size на четыре или пять. Затем, запустите код снова.

Вы должны увидеть, что блоков на экране стало больше, как показано на следующем скриншоте:



Установив размер на 14, вы создадите на много больше блоков

Выводы

В этой главе вы узнали, как повторять строки кода выполняя цикл этого кода, что позволило вам возвращаться в нём. Это можно применять для повторного использования одной и той же строки кода, в порядке выполнения задачи. Представьте, каково выводить числа от 1 до 10 не прибегая к циклу.

В следующей главе, мы исследуем функции, что являются базовыми компонентами повторно используемого кода.

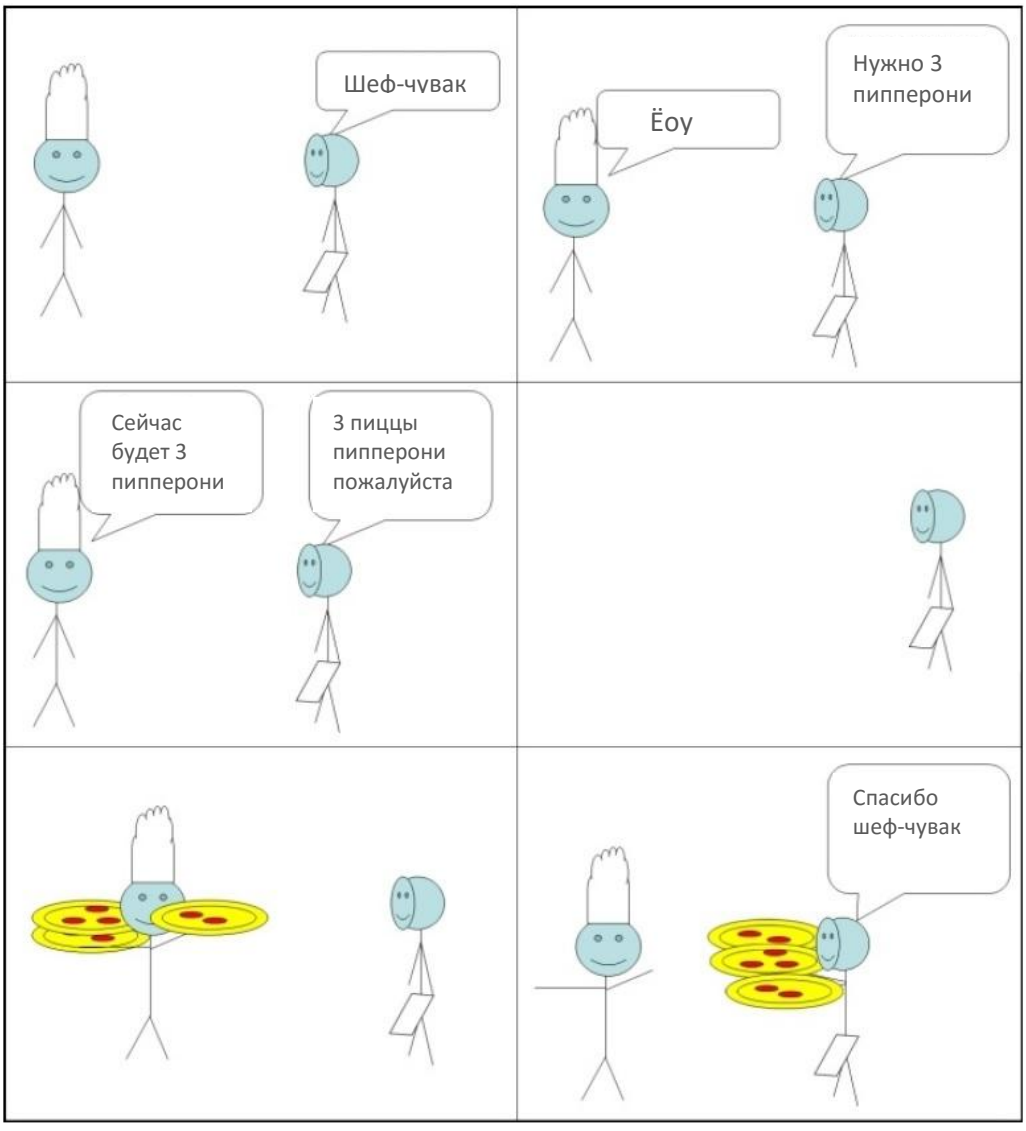
Глава 5.Функции и Макросы

Функции

Есть то, что нам нужно повторять. И не только код. Функция это связка кода, которая может вызываться любое количество раз, сколько пожелаете.

Аналогии это хорошо. Давайте рассмотрим аналогию с официантами, поварами, пиццой и функциями. В английском языке, когда мы говорим, что у человека есть функция, мы имеем в виду, что этот человек выполняет какую-то (обычно очень важную) специальную задачу. Эти люди могут выполнять эту задачу снова и снова, и когда бы их об этом не попросили.

Следующий комикс демонстрирует взаимодействие официанта (вызывающий) и шеф-повара (вызываемый). Официанту требуется еда на столик который он обслуживает, так что он просит (вызывает) повара приготовить еду заказанную на ожидающий столик. Повар приготовил еду и выдал заказ официанту.



Здесь шеф-повар выполняет свою функцию по приготовлению еды. Шеф-повар принял параметры на приготовление еды (три пиццы пипперони). Затем шеф уходит делать работу, и возвращается с трёмя пиццами. Заметьте, что официант не знает и не беспокоится о том как шеф-повар готовит пиццу. Шеф-повар абстрагирует процесс приготовления пиццы от официанта. Так что приготовление пиццы лишь простая однострочная команда для официанта. Официант просто хочет, чтоб его запрос выполнился и пицца вернулась к нему.

Когда функция (шеф-повар) вызвана с некоторыми аргументами (виды пиццы на приготовление), функция выполняет некоторые действия (по приготовлению пиццы) и возвращает результат (готовая пицца).

Пример функции – sqrt() библиотеки <cmath>

Теперь давайте поговорим о более практичных примерах и сопоставим их примеру с пиццей.

В библиотеке <cmath> есть функция называемая sqrt(). Давайте я быстро покажу её применение в следующем коде:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double rootOf5 = sqrt( 5 ); // функция вызывает функцию sqrt
  cout << rootOf5 << endl;
}
```

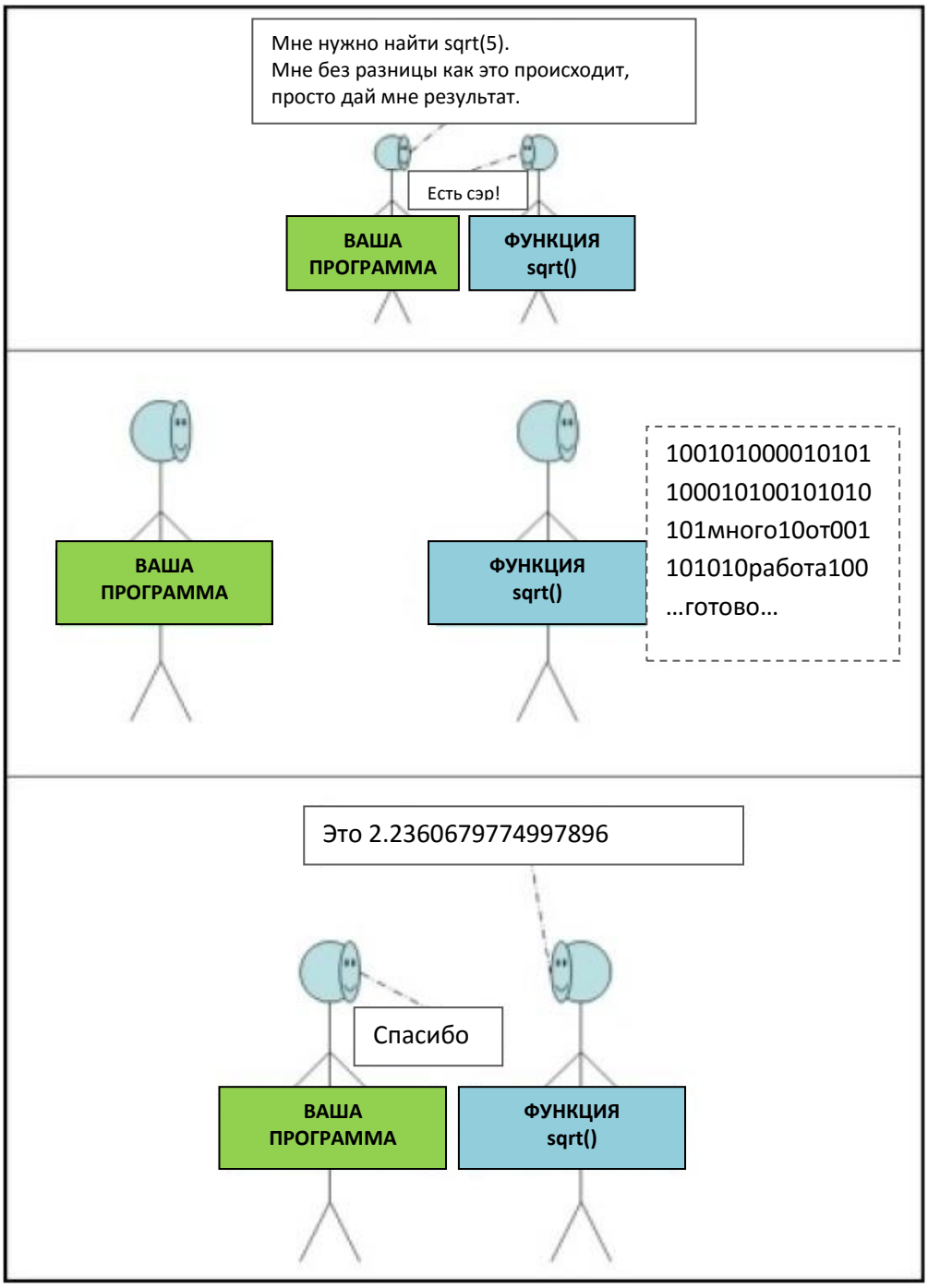
Так sqrt( ) может найти математический квадратный корень любого данного ему числа.

Вы знаете как найти квадратный корень такого трудного числа как 5? Это не просто. Умные души сели и написали функцию, которая может найти квадратный корень любого типа чисел. Обязательно ли вам вникать в математику по нахождению квадратного корня пяти, применяемую в вызове функции sqrt(5)? Конечно нет! Так же как и официанту не обязательно знать процесс приготовления, чтобы забрать готовую пиццу, так и вызывающему в библиотеке функций C++ для эффективного применения не обязательно полностью понимать как эта библиотека функций работает изнутри.

Вот преимущества использования функций:

1. Функции делают сложную задачу простой, упрощая рутину. Это делает требуемый для приготовления пиццы код, просто однострочной командой для вызывающего (вызывающим типично является ваша программа).
2. Функции избегают повторения кода, там где это не требуется. Скажем у нас 20 или больше строк кода, которые могут найти квадратный корень двойного значения. Мы сворачиваем эти строки в псевдотип функции. Вместо того чтобы повторно копировать и вставлять эти 20 строк кода, мы просто вызываем функцию sqrt (с числом для вычисления корня) когда бы нам ни понадобился корень.

Следующее изображение демонстрирует процесс по нахождению квадратного корня:



Написание нашей собственной функции

Скажем, мы хотим написать код, который выводит полосу дороги, как показано здесь:

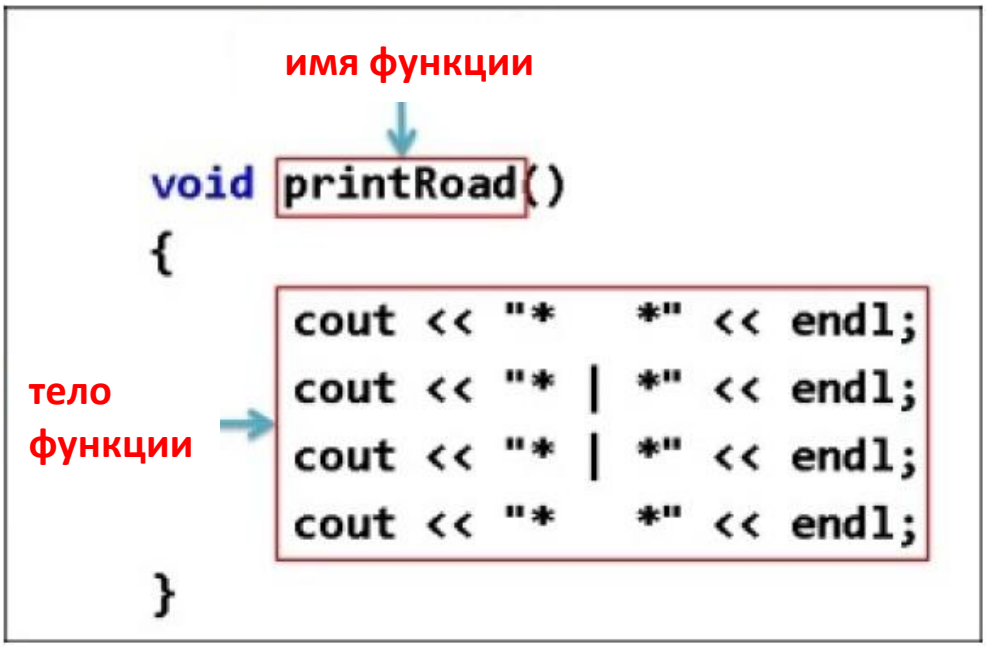
```
cout << "*   *" << endl;
cout << "* | *" << endl;
cout << "* | *" << endl;
cout << "*   *" << endl;
```

Скажем теперь, мы хотим вывести две такие полосы дороги, в ряд. Или три полосы. Или скажем, мы хотим выводить любое количество таких полос. Тогда нам нужно повторить четыре строки кода, которые образуют первую полосу дороги, один раз на каждую полосу дороги, что мы пытаемся вывести.

Что если мы введём нашу собственную C++ команду, которая позволяет нам выводить полосу дороги при вызове этой команды. Вот как это работает:

```
void printRoad() // напечатать дорогу
{
    cout << "*   *" << endl;
    cout << "* | *" << endl;
    cout << "* | *" << endl;
    cout << "*   *" << endl;
}
```

В этом отличие функции. Функция C++ имеет следующую анатомию:



Функцию применять просто: мы лишь вызываем функцию, которую хотим выполнить по имени, за которым идут две круглые скобки (). Например, вызов функции printRoad() запустит эту самую функцию. Давайте выполним трассировку примера программы, чтобы полностью понять, что это означает.

Образец трассировки программы

Вот полный пример того как работает вызов функции:

```
#include <iostream>
using namespace std;
void printRoad()
{
    cout << "*   *" << endl;
    cout << "* | *" << endl;
    cout << "* | *" << endl;
    cout << "*   *" << endl;
}
int main()
{
    cout << "Программа начинается!" << endl;
    printRoad();
    cout << "Программа заканчивается" << endl;
    return 0;
}
```

Давайте произведём трассировку выполнения программы от начала до конца. Запомните, что для всех программ C++, выполнение начинается на первой строке main().

Примечание

main() также является функцией. Она следит за выполнением всей программы. Как только main() выполняет оператор return, ваша программа заканчивается.

Когда достигнута последняя строка функции main(), программа заканчивается.

Строка за строкой отслеживается выполнение предыдущей программы:

```
void printRoad()
{
   cout << "*   *" << endl;                          // 3: затем мы переходим сюда
   cout << "* | *" << endl;                          // 4: запускаем это
   cout << "* | *" << endl;                          // 5: и это
   cout << "*   *" << endl;                          // 6: и это
}
int main()
{
   cout << "Программа начинается!" << endl;         // 1: первая строка на выполнение
   printRoad();                                     // 2: вторая строка...
   cout << "Программа заканчивается" << endl;       // 7: наконец последняя строка
   return 0;                                        // 8: и возвращение к ОС
}
```

Вот как будет выглядеть вывод этой программы:

```
Программа начинается!

*   *
* | *
* | *
*   *
Программа заканчивается
```

Вот объяснение предыдущего кода, строка за строкой:

1. Выполнение программы начинается на первой строке main(), которая выводит: “Программа начинается!”.
2. Следующая выполняемая строка вызывает printRoad(). Она переводит счётчик команд на первую строку printRoad(). Затем все строки printRoad() выполняются по порядку (шаги 3-6).
3. И наконец, после завершения вызова printRoad(), контроль возвращается к оператору main(). Затем мы видим вывод: “Программа заканчивается”.

Подсказка

Не забывайте круглые скобки после имени функции printRoad() при вызове. Вызываемая функция всегда должна сопровождаться круглыми скобками (), иначе она не сработает и компилятор выдаст вам ошибку.

Следующий код применим для вывода четырёх полос дороги:

```
int main()
{
   printRoad();
   printRoad();
   printRoad();
   printRoad();
}
```

Альтернативно, вы можете использовать следующий код:

```
for( int i = 0; i < 4; i++ )
printRoad();
```

Так что вместо повторения четырёх строк cout каждый раз, когда сегмент отображён, мы просто вызываем функцию printRoad(), чтобы выполнить его отображение. Также, если мы хотим изменить то как выглядит отображаемая дорога, нам просто нужно модифицировать осуществление функции printRoad().

Вызов функции влечёт за собой запуск всего тела функции, строка за строкой. После того как вызов функции завершён, управление программы возобновляется с места где была вызвана функция.

Упражнение

В качестве упражнения выясните, что не так с этим кодом:

```
#include <iostream>
using namespace std;
void myFunction()
{
  cout << "Вы вызывали?" << endl;
} int main()
{
  cout << "Я сейчас собираюсь вызвать myFunction." << endl;
  myFunction;
}
```

Решение

Верный ответ на эту проблему заключается в том, что вызов myFunction (на последней строке main()) не сопровождается круглыми скобками. Каждый вызов функции должен сопровождаться круглыми скобками. Последняя строка main() должна быть записана как myFunction(); ,а не просто myFunction.

Функции с аргументами

Как мы можем расширить функцию printRoad(),чтобы выводить дорогу с заданным количеством сегментов? Ответ прост. Мы можем позволить функции printRoad() принимать параметр, названный numSegments, чтобы выводить определённое количество сегментов дороги.

Следующий отрывок кода показывает, как это будет выглядеть:

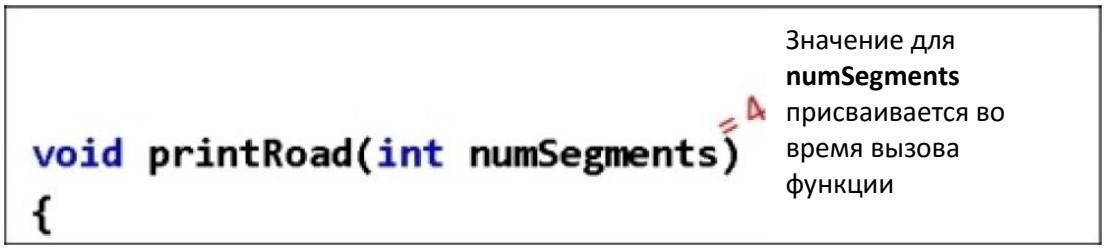
```
void printRoad(int numSegments)
{
   // используйте цикл for, чтобы вывести дорожные сегменты numSegments
   for( int i = 0; i < numSegments; i++)
   {
      cout << "*   *" << endl;
      cout << "* | *" << endl;
      cout << "* | *" << endl;
      cout << "*   *" << endl;
   }
}
```

Следующее изображение показывает анатомию функции принимающей аргумент:

Вызовите эту новую версию printRoad(), запрашивая её отобразить четыре сегмента:

```
printRoad( 4 );   // вызов функции
```

Цифра 4 между скобок вызываемой функции в предыдущем утверждении назначается переменной numSegments функции printRoad(int numSegments). Вот как передаётся значение 4 к numSegments:



Как printRoad(4) присваивает значение 4 переменной numSegment

Так при вызове printRoad() в скобках присваивается значение для numSegments.

Функции возвращающие значения

Функция sqrt() является примером функции, которая возвращает значение. Функция sqrt() принимает единственный параметр в своих скобках (число для вычисления корня) и возвращает действительный корень числа.

Вот пример использования функции sqrt:

```
cout << sqrt( 4 ) << endl;
```

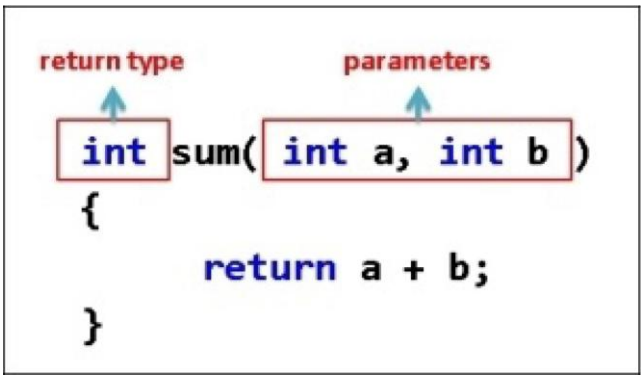
Функция sqrt() производит некие аналогии с тем, что делает шеф-повар, когда готовит пиццу.

Как вызывающий функции, вы не заботитесь о том, что происходит в теле функции sqrt(). Эта информация неважна, так как всё чего вы хотите это результат квадратного корня, задаваемого вами числа.

Давайте объявим нашу собственную простую функцию, которая возвращает значение, как показано в следующем коде:

```
int sum(int a, int b)
{
  return a + b;
}
```

Следующий скриншот демонстрирует анатомию функции с параметрами и возвратным значением:



Функция sum очень базовая. Всё что она делает, так принимает целые числа int a и b, суммирует их и возвращает результат. Вы можете сказать, что нам даже не нужна целая функция просто, чтобы сложить два числа. И вы правы, но давайте продолжим на минутку. Мы будем использовать эту простую функцию, чтобы объяснить принцип возвратного значения.

Вы будете использовать функцию sum таким образом (из main()):

```
int sum( int a, int b )
{
  return a + b;
}
int main()
{
  cout << "Сумма 5 и 6 равна " << sum( 5,6 ) << endl;
}
```

Для завершения команды cout, должен быть выражен вызов функции sum(5, 6). Прямо с места, где происходит вызов функции sum(5, 6), помещается возвратное значение от sum(5, 6).

Другими словами, это строка кода, которую и видит cout после выражения вызова функции sum(5, 6).

```
cout << "Сумма чисел 5 и 6 равна " << 11 << endl;
```

Возвратное значение из sum(5, 6) эффективно вырезается и вставляется в место вызова функции.

Значение всегда должно возвращаться функцией, которая обещает это (если возвратный тип функции что угодно кроме void).

Упражнения

1. Напишите функцию isPositive, которая возвращает true, когда двойной параметр переданный ей действительно положительный.
2. Завершите следующее определение функции:

```
// функция возвращает true, когда величина 'a'
// равна величине 'b' (абсолютное значение)
bool absEqual(int a, int b){
// при завершении этого упражнения, попробуйте не применять
// функции библиотеки cmath
}
```

3. Напишите функцию getGrade(), которая принимает целочисленное значение (более 100) и возвращает оценку (A, B, C, D, или F).
4. Математическая функция формы f(x) = 3x + 4. Напишите функцию C++, которая возвращает значения для f(x).

Решения

1. Функция isPositive принимает двойной параметр и возвращает булевое значение:

```
bool isPositive( double value )
{
  return value > 0;
}
```

2. Далее завершённая функция absEqual:

```
bool absEqual( int a, int b )
{
      // Сделайте a и b положительными
   if( a < 0 )
      a = -a;
   if( b < 0 )
      b = -b;
   // теперь, когда они оба положительные,
   // нам просто нужно сравнить на равность a и b
   return a == b;
}
```

3. В следующем коде дана функция getGrade():

```
char getGrade( int grade )
{
   if( grade >= 80 )
     return 'A';
   else if( grade >= 70 )
     return 'B';
   else if( grade >= 60 )
     return 'C';
   else if( grade >= 50 )
     return 'D';
   else
     return 'F';
}
```

4. Это простая программа, которая должна развлечь вас. Происхождение функции name в C++ исходит из математического мира, как показано в следующем коде:

```
double f( double x )
{
   return 3*x + 4;
}
```

Переменные, пересмотр

Всегда здорово пересмотреть темы, которые вы прошли ранее. Особенно теперь, когда вы гораздо глубже понимаете написание кода на C++.

Глобальные переменные

Теперь, когда мы ознакомились с принципом функций, можно ознакомиться и с принципом глобальной переменной.

Что такое глобальная переменная? Глобальная переменная – это любая переменная, доступная всем функциям программы. Как мы можем сделать переменную доступной всем функциям программы? Мы просто объявляем переменную вверху кода файла. Обычно после или не далеко от утверждения #include.

Вот пример программы с глобальными переменными:

```
#include <iostream>
#include <string>
using namespace std;

string g_string;    // глобальная строковая переменная,
// доступная всем функциям в пределах программы
// (потому что она объявлена до любой функции
// ниже!)

void addA(){ g_string += "A"; }
void addB(){ g_string += "B"; }
void addC(){ g_string += "C"; }

int main()
{
  addA();
  addB();
  cout << g_string << endl;
  addC();
  cout << g_string << endl;
}
```

Здесь одна и та же глобальная переменная g_string доступна всем четырём функциям в программе (addA(), addB(), addC(), и main()). Глобальные переменные активны в течении программы.

Подсказка

Иногда программисты предпочитают ставить префикс g_ глобальной переменной, но это не является обязательным условием.

Локальные переменные

Локальной переменной является переменная, определённая в блоке кода. Область действия локальной переменной завершается в конце блока, в котором они объявлены. Далее, в разделе *Область действия переменной*, мы рассмотрим несколько примеров.

Область действия переменной

Область кода, где может использоваться переменная, является областью действия этой переменной. Область действия любой переменной в основном это блок, в котором она была определена. Мы можем продемонстрировать область действия переменной следующим примером:

```
int g_int; // область глобальной int до конца файла
void func( int arg )
{
  int fx;
} // </fx> уже не досягаемо, </arg> заканчивается

int main()
{
   int x; // область действия переменной <x> начинается здесь...
          // до конца main()
   if( x == 0 )
   {
     int y; // область действия переменной <y> начинается здесь,
            // и ограничивается закрытием фигурной скобки ниже
   } // </y> уже не досягаема
   if( int x2 = x ) // создаётся переменная <x2> и устанавливается равной <x>
   {
    // входим сюда если x2 была равна не нулю
   } // </x2> заканчивается

for( int c = 0; c < 5; c++ ) // переменная c создаётся и имеет
   { // область действия внутри фигурных скобок цикла
     cout << c << endl;
   } // </c> становится недосягаема только при выходе из цикла
} // </x> завершается
```

Главное что определяет область действия переменной это блок. Давайте обсудим область действия пары переменных определённых в предыдущем примере кода:

- g_int: Это глобальная переменная целочисленного типа, область действия которой простирается с места её объявления до конца кода этого файла. И это говорит о том, что g_int может использоваться внутри func() и main(), но не может использоваться в коде других файлов. Чтобы иметь одну глобальную переменную, которая будет использоваться в коде множества файлов, вам нужна внешняя переменная.
- arg (аргумент func()): Может использоваться с первой строки func() (после открывающей фигурной скобки {) до последней строки func() (до закрывающей фигурной скобки }).
- fx: Может использоваться где угодно внутри func() до закрывающей фигурной скобки этой самой func() }.
- main() (переменные внутри main()): Может использоваться как написано в комментариях.

Обратите внимание, что переменные объявленные в скобках списка аргументов функции могут использоваться только в блоке под объявлением этой функции. Например, переменная arg передаётся к func():

```
void func( int arg )
{
  int fx;
} // </fx> уже не досягаемо, </arg> заканчивается
```

Переменная arg перестанет быть доступной после закрывающей фигурной скобки (}) функции func(). Будет нелогичным если круглые скобки технически будут вне фигурных скобок, которые определяют { блок }.

То же касается и переменных, объявленных внутри круглых скобок цикла for. Взгляните на следующий пример цикла for:

```
for( int c = 0; c < 5; c++ )
{
  cout << c << endl;
} // c прекращается здесь
```

Переменная int c может использоваться в круглых скобках объявления цикла for или в блоке под этим объявлением цикла. Переменная c будет недосягаема после закрывающей фигурной скобки цикла for, в котором она объявлена. Если вы хотите, чтобы переменная c, была действительна и после фигурных скобок цикла for, то вам надо объявить эту переменную перед циклом for, как показано здесь:

```
int c;
for( c = 0; c < 5; c++ )
{
   cout << c << endl;
```

```
} // с не прекращает быть активной здесь
```

Статические локальные переменные

Локальные переменные static прямо как глобальные переменные, только у них локальная область действия, как показано в следующем коде:

```
void testFunc()
{
  static int runCount = 0; // это запускается только ОДИН РАЗ, даже при
  // последующем вызове testFunc()!
  cout << "Запустили эту функцию " << ++runCount << " раз" << endl;
} // runCount больше не области действия, но не прекращает быть активной здесь
int main()
{
  testFunc(); // говорится 1 раз
  testFunc(); // говорится 2 раза!
}
```

При использовании ключевого слова static в testFunc(), переменная runCount запоминает её значение между вызовами testFunc(). Так что вывод двух отдельных запусков testFunc() будет:

```
Запустили эту функцию 1 раз
Запустили эту функцию 2 раза
```

Это потому что статические переменные создаются и обретают начальное значение только раз (первый раз, когда функция объявлена и запускается), и после этого статические переменные сохраняют своё старое значение. Скажем, мы объявляем runCount как регулярную, локальную, нестатическую переменную:

```
int runCount = 0; // если объявить таким образом, runCount будет локальной
```

Затем, вывод будет выглядеть так:

```
Запустили эту функцию 1 раз
Запустили эту функцию 1 раз
```

Здесь мы видим testFunc говорящую: “Запустили эту функцию 1 раз”, оба раза. Как у локальной переменной, значение runCount не сохраняется между вызовами функции.

Вы не должны злоупотреблять статическими локальными переменными. По факту, вы должны применять статические локальные переменные лишь, когда это совсем необходимо.

Const переменные

Переменная const – это переменная значение которой вы пообещали компилятору не менять после первого присвоения этого значения. Мы можем для примера объявить одну, со значением для pi:

```
const double pi = 3.14159;
```

Поскольку pi это универсальная константа (одна из немногих на которую вы можете полагаться как на неизменную), не должно быть никакой нужды менять pi после первого присвоения значения ей. По факту, изменение pi должно быть запрещено компилятором. Попробуйте например, назначить новое значение для pi:

```
pi *= 2;
```

Мы получим следующую ошибку компилятора:

```
error C3892: 'pi' : you cannot assign to a variable that is const
```

Эта ошибка ясно даёт понять, что помимо начального присвоения значения, нам не должно быть доступно дальнейшее изменение значения pi – переменной, которая является константой.

Прототипы функций

Прототип функции – это описание функции без тела. Например, давайте сделаем прототипы функций isPositive, absEqual и getGrade из следующих упражнений:

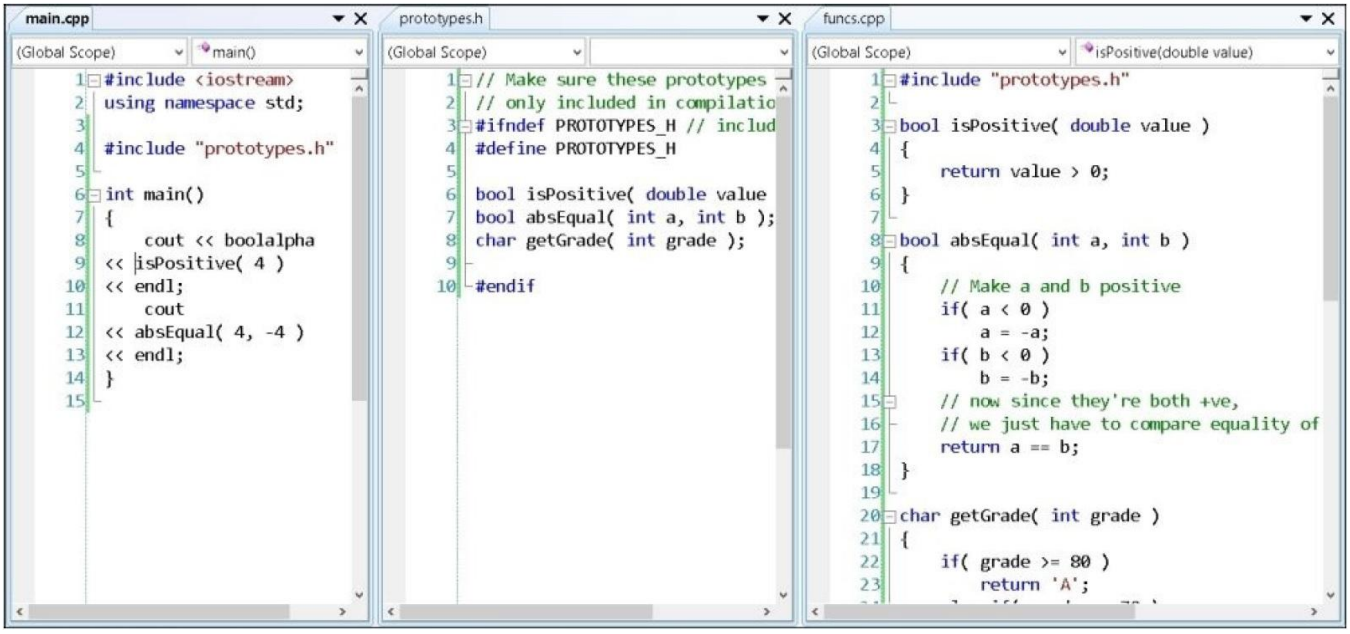
```
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

Обратите внимание, что прототип функции это просто возвратный тип, имя функции и список аргументов, которые запрашивает функция. Прототипы функций не имеют тела. Тело функции, как правило, помещается в .cpp файле.

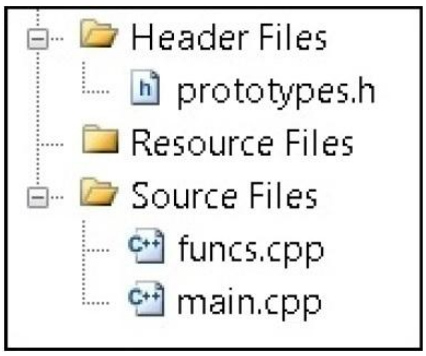
Файлы .h и .cpp

Обычно ваши прототипы функции располагаются в файле .h, а тело функций в файле .cpp. Причина этого в том, что вы можете включить ваш .h файл в связку файлов .cpp и не получить множество ошибок определения.

Следующий скриншот даёт вам ясное изображение файлов .h и .cpp:



Здесь у нас есть три файла в этом проекте Visual C++:



prototypes.h содержит

```
// Убедитесь, что эти прототипы
// включены в компиляцию только ОДИН РАЗ
#pragma once
extern int superglobal; // extern: "прототип" переменной
// прототипы функций
bool isPositive( double value );
bool absEqual( int a, int b );
char getGrade( int grade );
```

Файл prototypes.h содержит прототипы функций. Мы объясним что делает ключевое слово extern, через пару параграфов.

funcs.cpp содержит

```
#include "prototypes.h" // каждый файл, который использует isPositive,
// absEqual или getGrade должен #include (включать) "prototypes.h"
int superglobal; // "реализация" переменной
// Сами определения функций здесь, в файле .cpp
bool isPositive( double value )
{
   return value > 0;
}
bool absEqual( int a, int b)
{
   // Делаем a и b положительными
   if( a < 0 )
      a = -a;
   if( b < 0 )
      b = -b;
   // теперь, когда они оба положительные,
   // на нужно лишь сравнить на равенство a и b
   return a == b;
}
char getGrade( int grade )
{
   if( grade >= 80 )
     return 'A';
   else if( grade >= 70 )
     return 'B';
   else if( grade >= 60 )
     return 'C';
   else if( grade >= 50 )
     return 'D';
   else
     return 'F';
}
```

main.cpp содержит

```
#include <iostream>
using namespace std;
#include "prototypes.h" // для использования функций isPositive, absEqual
int main()
{
  cout << boolalpha << isPositive( 4 ) << endl;
  cout << absEqual( 4, -4 ) << endl;
}
```

Когда вы разбиваете код на .h и .cpp файлы, то файл .h (заголовочный файл) называется интерфейс, а файл .cpp (в котором сами функции) называется реализацией.

Поначалу озадачивающая часть для некоторых программистов. Как например, в следующих вопросах. Откуда C++ знает где находится тело функции isPositive и

тело функции getGrade, если мы включаем (#include) только прототипы? Не должны ли мы также включать (#include) файл func.cpp в main.cpp?

Ответ на всё это – *магия*. Вам нужно лишь включать заголовочный файл rototypes.h и в main.cpp и в func.cpp. В то время как оба файла .cpp включены в ваш C++ проект **Интегрированной Среды Разработки** (**ИСР**) (это значит, они появляются в дереве обзора **Solution Explorer**, с левой стороны), присоединение прототипов к телу функций выполняется автоматически компилятором.

Переменные extern

Объявление extern схоже с прототипом функции, но применяется к переменным. Вы можете расположить объявление глобальной переменной extern в файле .h и включить этот .h файл во всю связку других файлов. Таким образом, у вас может быть единственная глобальная переменная, которая может совместно использоваться множеством исходных фалов, без получения множества ошибок определенных символов найденных компоновщиком. Вы располагаете само объявление переменной в файле .cpp, так что переменная объявляется только раз. Переменная extern есть в файле prototypes.h в предыдущем примере.

Макросы

Макросы C++ выходят из класса команд C++ называемых директивами препроцессора. Директивы препроцессора выполняются до компиляции.

Макросы начинаются с #define. Например, у нас есть следующий макрос:

```
#define PI 3.14159
```

На самом низком уровне макросы просто копируют и вставляют операции, которые происходят перед временем компиляции. В предыдущем утверждении макроса, 3.14159 буквально будет скопировано и вставлено всюду где в программе встречается символ PI.

Посмотрите на следующий пример кода:

```
#include <iostream>
using namespace std;
#define PI 3.14159
int main()
{
  double r = 4;
  cout << "Окружность равна " << 2*PI*r << endl;
}
```

Что будет делать препроцессор C++? Сначала пройдёт по коду и просмотрит есть ли использование символа PI. Он найдёт такое использование на этой строке:

```
cout << "Окружность равна " << 2*PI*r << endl;
```

Предыдущая строка кода до компиляции будет преобразовываться в следующую строку:

```
cout << "Окружность равна " << 2*3.14159*r << endl;
```

Итак, всё что происходит с утверждением #define это замещение всех используемых случаев (например PI) на алгебраическое число 3.14159 перед компиляцией. Смысл использования макросов таким образом, в том, чтобы избегать числа усложняющие код. Символы обычно легче читать, чем большие, длинные числа.

Совет – попробуйте применить переменные const там, где это возможно

Вы можете использовать макросы, чтобы определять переменные константы. Также вместо этого вы можете использовать выражения переменных const. Итак, скажем у нас есть следующая строка кода:

```
#define PI 3.14159
```

В помощь нам будет предложено следующее:

```
const double PI = 3.14159;
```

Применение переменной const будет предпочтительным, потому что она хранит ваше значение внутри самой переменной. Переменная определена, а определённые данные это хорошо.

Макросы с аргументами

Мы также можем писать макросы, которые принимают аргументы. Вот пример макроса с аргументом:

```
#define println(X) cout << X << endl;
```

Что этот макрос будет делать? Каждый раз когда println (“Некое значение”) будет встречаться в коде, код справой стороны (cout << "Some value" << endl) будет скопирован и вставлен на консоли. Заметьте что аргумент между скобок скопирован на место X. Скажем у нас есть следующая строка кода:

```
println( "Всем привет" )
```

Она будет замещена следующим утверждением:

```
cout << "Всем привет" << endl;
```

Макросы с аргументами прямо как короткие функции. Макросы не могут содержать символы-разделители строк.

Совет – примените встроенные функции вместо макросов с аргументами

Вы должны знать как работают макросы с аргументами, потому что вы будете часто встречать их в коде C++. Тем не менее, когда возможно, многие C++ программисты предпочитают использовать встроенные функции нежели макросы с аргументами.

Обычные функции вызывают выполнение, включающее инструкцию перехода к функции и затем выполнение функции. Встроенная функция – это та функция, чьи строки скопированы в место вызова и никакой переход не требуется. Применение встроенных функций обычно имеет смысл для очень маленьких, простых функций, которые имеют не большое количество строк. Например, мы можем встроить простую функцию max, которая находит большее из двух значений:

```
inline int max( int a, int b )
{
  if( a > b ) return a;
  else return b;
}
```

Везде где пользуется эта функция max, код для тела функции будет скопирован и вставлен в место вызова функции. Когда не нужен переход к функции, экономится время выполнения, что делает встроенные функции эффективными подобно макросам.

Есть ухищрение по использованию встроенных функций. Встроенные функции должны иметь своё собственное тело, полностью содержащееся в заголовочном файле .h. Так компилятор может производить оптимизацию и собственно встраивать функцию туда, где она используется. Функции делаются встроенными типично ради скорости (поскольку вам не надо переходить к другому телу кода, чтобы выполнить функцию), но ценой разрастания кода.

Вот причины, по которым встроенные функции предпочтительнее макросов:

1. Макросы предрасположены к ошибкам: не обозначен аргумент макроса.
2. Макрос должны быть написаны на одной строке, иначе вы увидите их с применением символов перевода на новую строку:

   ```
   \
   знаки перехода строки \
   как эти \
   усложняют чтение
   ```

3. Если макрос написан не внимательно, это послужит сложной для исправления ошибкой компилятора. Например, если вы не правильно поместили в скобки ваш аргумент, то ваш код просто будет не верным.

4. Большой макрос трудно отлаживать.

Следует сказать, что макросы позволяют вам выполнять некоторую магию компилятора препроцессоров. UE4 предпринимает много действий с использованием макросов с аргументами, что вы увидите далее.

Выводы

Вызов функции позволяет вам повторно использовать основной код. Повторное использование кода важно по ряду причин. В основном, потому что программирование само по себе итак является сложным и нужно избегать насколько возможно, повторного выполнения одной и той же сложной работы. Усилия программиста, который написал функцию sqrt(), не должны повторяться другими программистами, которые хотят решить ту же проблему.

Глава 6. Объекты, Классы и Наследование

В предыдущей главе мы обсудили функции как способ связать вместе строки кода имеющие одно отношение. Мы поговорили о том, как функции абстрагируются от деталей осуществления, и как функция sqrt() не требует от вас понимания её внутренней работы, чтобы применить её для нахождения корня. Всё это было хорошо, преимущественно потому что это экономит время и усилия программиста, так как делает саму работу по нахождению квадратного корня легче. Этот принцип *абстрагирования* поднимется здесь снова, когда мы будем обсуждать объекты.

Вкратце сказать, объекты связывают вместе методы и относящиеся к ним данные в одну структуру. И эти структуры называются *классами*. Главная идея использования объектов заключается в создании представления кода для всего, что есть вашей игре. Каждый представленный в коде объект будет иметь данные и связанные с ними функции, которые оперируют этими данными. Так что у вас будет объект для представления экземпляра вашего игрока и относящихся к нему функций, которые позволяют игроку прыгать(), стрелять() и братьПредмет(). А также у вас будет объект для представления каждого экземпляра монстра и относящихся уже к нему функций, таких как рычать(), атаковать() и возможно преследовать().

Объекты являются типом переменных и будут оставаться в памяти так долго, как вы сами будете держать их там. Вы создаёте экземпляр объекта один раз, когда создано то, что он представляет в вашей игре. И вы аннулируете экземпляр объекта, когда “умирает” то, что он представлял в вашей игре.

Объекты могут быть использованы для представления того, что находится в игре, а также для представление любых других типов вещей. Например, вы можете хранить изображение как объект. Поля данных будут шириной изображения, его высотой и набором пикселей внутри него. Строковые типы C++ также являются объектами.

Совет

Глава содержит множество ключевых слов, которые могут быть сложными для понимания с первого раза, включая *виртуальный* и *абстрактный*.

Не позволяйте более сложным разделам этой главы утянуть вас ещё глубже в непонимание. Я внёс описания для многих продвинутых принципов в качестве дополнения. К тому же, уясните для себя одно, что вам не обязательно понимать абсолютно всё в этой главе, чтобы написать рабочий код C++ в UE4. Всё это помогает понять тему, но если вы не можете понять смысл чего то, не стопоритесь на этом. Просто прочитайте как следует и двигайтесь дальше. Возможно, будет так, что вы не поймёте это поначалу, но не забывайте обращаться к этому вопросу,

когда пишите код. И когда вы снова откроете эту книгу, то “вуаля!”, начнёт проясняться смысл.

Объекты struct

Объект в C++ обычно является любым типом переменной, которая образована из конгломерата более простых типов. Самый базовый объект в C++ это struct. Мы используем ключевое слово struct, чтобы склеивать связку более мелких переменных в одну большую переменную. Если вы не забыли, мы знакомились вкратце со struct в Главе 2, *Переменные и Память*. Давайте ещё раз посмотрим на этот простой пример:

```
struct Player
{
string name;
int hp;
};
```

Это определение структуры для образования объекта Player. У игрока есть строковый тип (string) для его имени (name), и целочисленный тип (int) для значения его единиц здоровья (hp).

Если вы вспомните из Главы 2, *Переменные и Память*, способ которым мы производили экземпляр объекта Player таков:

```
Player me; // создаём экземпляр Player, названный me
```

Отсюда мы можем иметь доступ к полям объекта me таким образом:

```
me.name = "Tom";
me.hp = 100;
```

Функция-член

Теперь, исполнительная часть. Мы можем прилагать функцию-член к определению struct, просто записав эту функцию внутри определения struct.

```
struct Player
{
  string name;
  int hp;
  // Функция-член уменьшающая hp игрока на какую-либо сумму
  void damage( int amount )
  {
    hp -= amount;
  }
  void recover( int amount )
  {
    hp += amount;
  }
};
```

Функция-член – это просто функция C++ объявленная внутри struct или определении класса. Великолепная идея, не правда ли?

Отчасти эта идея забавна, так что я просто сказал как есть. Переменные struct Player доступны всем функциям в struct Player. Внутри каждой функции-члена struct Player, мы собственно можем иметь доступ к переменным name и hp, как если бы они были локальными для функции. Другими словами, переменные name и hp объекта struct Player совместно используются между всеми функциями-членами этого объекта.

Ключевое слово this

В некотором C++ коде (в дальнейших главах), вы увидите больше обращений к ключевому слову this. Ключевое слово this – это указатель, который ссылается на действующий объект. Внутри функции Player::damage(), мы можем, например написать нашу ссылку прямо с ключевым словом this:

```
void damage( int amount )
{
  this->hp -= amount;
}
```

Ключевое слово this имеет смысл лишь внутри функции-члена. Мы можем прямо включить использование ключевого слова this внутри функции-члена, но без написания this, имеется в виду, что мы говорим о hp данного объекта.

Строковые типы являются объектами?

Да! Каждый раз, когда в прошлом вы использовали строковую переменную, вы использовали объект. Давайте испытаем некоторые функции-члены в классе string.

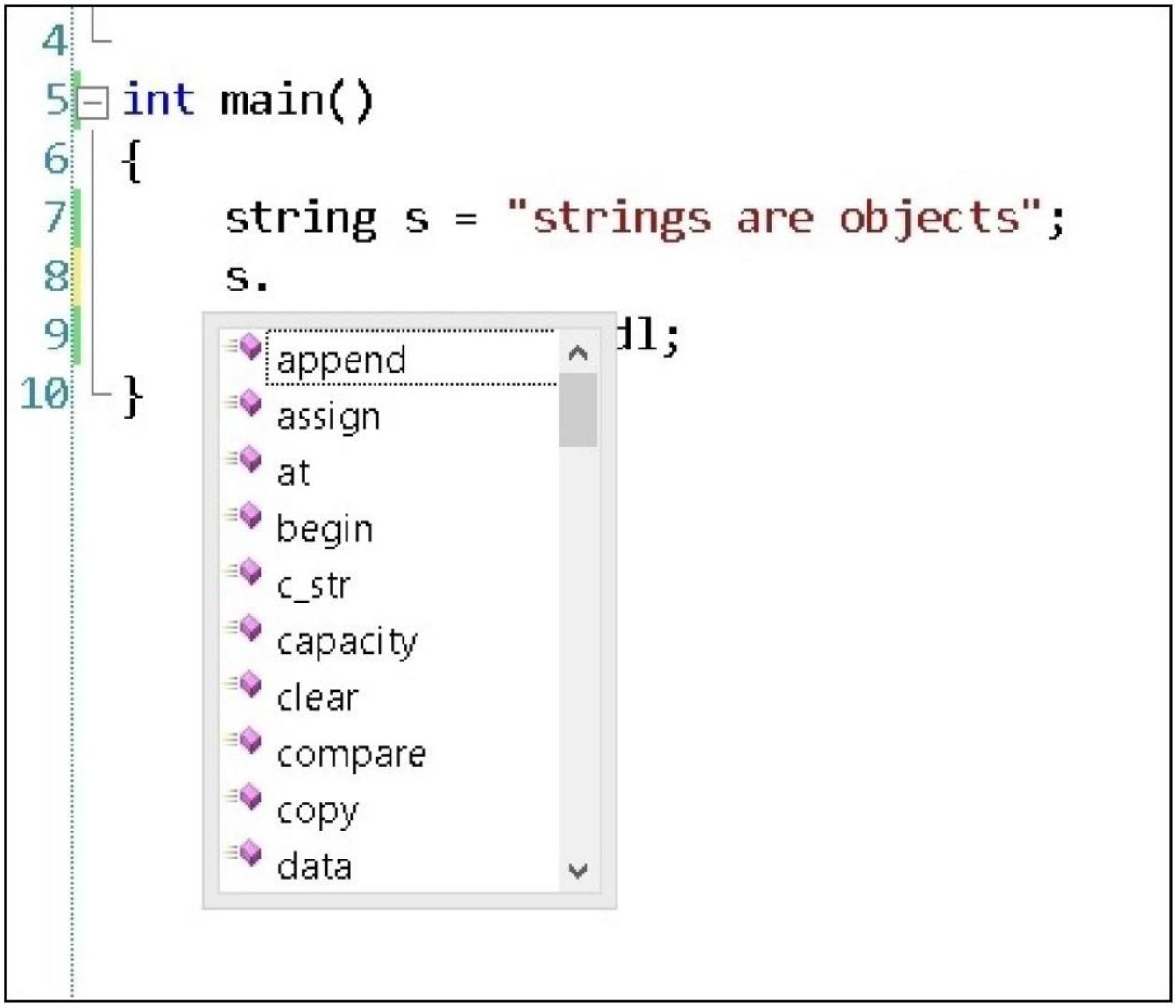
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string s = "strings are objects";
  s.append( "!!" ); // добавьте "!!" в конец строки!
  cout << s << endl;
}
```

Что мы здесь сделали? Применили функцию-член append(), чтобы добавить два дополнительных знака на конец строки (!!). Функция-член всегда применяется к объекту, который и вызывает эту функцию-член (объект слева от точки).

Подсказка

Чтобы увидеть список функций и функций-членов доступных в объекте, напишите имя переменной объекта в Visual Studio, затем поставьте точку (.), затем нажмите

Ctrl и пробел. И появится список.



Нажатие Ctrl и пробела вызывает список членов

Запуск функции-члена

Функция-член может быть запущена при помощи следующего синтаксиса:

```
objectName.memberFunction();
```

Объект запускающий функцию-член находится слева от точки. Функция-член для вызова справа от точки. Запуск функции-члена всегда сопровождается круглыми скобками (), даже если в них не передан аргумент.

Так, в той части программы, где атакует монстр, мы можем уменьшить значение hp игрока таким образом:

```
player.damage( 15 ); // игрок получает урон в 15 единиц
```

Что не более читабельно, чем следующее:

```
player.hp -= 15; // игрок получает урон в 15 единиц
```

Совет

Когда функции-члены и объекты используются эффективно, ваш код будет читаться больше как проза или поэзия, нежели как куча операторных символов столпотворённых вместе.

Помимо красоты и читабельности, каково значение написания функций-членов? Вне объекта Player, с помощью единственной строки кода, мы сейчас можем делать больше, чем просто уменьшать элемент hp на 15. Мы можем также делать другие вещи как уменьшение hp игрока, такие как: принимать в учётную запись броню игрока; проверять, неуязвим ли игрок или иметь другие эффекты, когда игроку наносится урон. То, что происходит, когда игроку нанесён урон, должно получаться от функции damage().

Теперь представим, что у игрока есть класс брони. Давайте добавим поле в struct Player для класса брони:

```
struct Player
{
  string name;
  int hp;
  int armorClass;
};
```

Нам понадобилось бы уменьшить урон полученный игроком, с помощью класса брони (armor) игрока. Так что теперь мы написали бы формулу, чтобы уменьшать hp. Мы можем сделать это не объектно-ориентированным способом, прямо войдя в поле данных объекта player:

```
player.hp -= 15 – player.armorClass; // не ООП
```

Или мы можем сделать это объектно-ориентированным способом, написав функцию-член, которая меняет данные элементов объекта player как надо. Внутри объекта player, мы можем написать функцию-член damage():

```
struct Player
{
  string name;
  int hp;
  int armorClass;
  void damage( int dmgAmount )
  {
    hp -= dmgAmount - armorClass;
  }
};
```

Упражнения

1. Есть лёгкий баг в функции damage в предыдущем коде. Можете вы найти и исправить его. Подсказка: Что произойдёт, если заданный урон (damage) меньше чем armorClass?
2. Наличие только числа для класса брони, не даёт достаточно информации о броне! Каково название брони? Как она выглядит? Придумайте функцию struct для брони объекта Player с полями для имени, класса брони и рейтинга продолжительности.

Решения

Решения для кода объекта struct Player перечислены в следующем разделе, *Private и инкапсуляция*.

Как на счёт следующего кода:

```
struct Armor
{
  string name;
  int armorClass;
  double durability;  // двойная продолжительность
};
```

Экземпляр Armor, затем будет помещён внутри struct Player:

```
struct Player
{
  string name;
  int hp;
  Armor armor; // Player обладает Armor
};
```

Это означает, что у игрока есть броня. Мы исследуем отношения *имеет* против *является* позже.

Private и инкапсуляция

Итак, теперь мы определили пару функций-членов, чья цель модифицировать и поддерживать элементы данных нашего объекта Player. Но некоторые программисты предлагают аргументы.

Аргументы следующие:

- Элементы данных объекта всегда должны быть доступны только через функции-члены, и никогда напрямую.

Это означает, что вы вообще не должны иметь доступ к элементам данных объекта напрямую извне объекта, то есть модифицировать hp игрока напрямую:

player.hp -= 15 – player.armorClass; // плохо: прямой доступ к элементу

Этот принцип называется инкапсуляция. Инкапсуляция – это концепция, по которой взаимодействие с каждым объектом должно осуществляться только посредством функций-членов. Инкапсуляция говорит, что необработанные элементы данных, должны быть недоступны напрямую.

Причины инкапсуляции таковы:

- **Чтобы сделать содержание класса безопасным**: Главная идея инкапсуляции заключается в том, что объекты работают лучше, когда они спрограммированы так, что они управляют и поддерживают свои внутренние переменные без надобности писать код за пределами класса, чтобы рассматривать частные данные этого класса. Когда код объектов написан таким образом, то работать с объектами становится гораздо легче, их легче читать и поддерживать. Чтобы заставить объект игрока прыгать, вам просто нужно вызвать player.jump() и позволить состоянию управления объекта игрока сменить его положение высоты по оси **y** (y-height), что заставляет игрока подпрыгнуть. Когда внутренние элементы объекта не открыты, взаимодействие с этим объектом гораздо легче и эффективней. Взаимодействие лишь с публичной функцией-членом объекта, позволяет объекту управлять своим внутренним состоянием (мы скоро объясним ключевые слова private (частный) и public (публичный)).
- **Чтобы избегать нарушений кода**: Когда код вне класса взаимодействует только с публичными функциями-членами этого класса (публичный интерфейс класса), то внутреннее управление состоянием может свободно меняться, без каких-либо нарушений вызываемого кода. Таким образом, если внутренние элементы данных объекта меняются по какой-либо причине, весь код использующий объект будет оставаться в порядке, пока в порядке остаются функции-члены.

Итак, как же мы можем не допустить, чтоб программист делал что то неправильно и имел доступ к элементам данных напрямую? Тут C++ вводит концепцию *модификаторов доступа*, чтобы предотвратить доступ к внутренним данным объекта.

Это как мы используем модификаторы доступа, чтобы запретить доступ к определённым секциям объекта struct Player извне.

Первое, что вам понадобится сделать, это решить какие секции определения struct вы хотите, чтобы были доступными снаружи класса. Эти секции будут отмечены как public. Все другие области, которые будут недоступны за пределами struct, будут отмечены как private.

```
struct Player
{
private:    // начинается частная секция... к ней нет доступа
            // снаружи класса, до…
  string name;
  int hp;
  int armorClass;
  public: // …до СЮДА. Так начинается публичная секция
  // Эта функция-член доступна снаружи struct,
  // потому что она в секции отмеченной как public:
  void damage( int amount )
  {
     int reduction = amount – armorClass;
     if( reduction < 0 ) // убедитесь что не отрицательное!
     reduction = 0;
     hp -= reduction;
  }
};
```

Некоторым программистам нравится public

Некоторые программисты без зазрения используют публичные элементы данных и не инкапсулируют свои объекты. Это дело вкуса, хоть и считается плохо практикой объектно-ориентированного программирования.

Тем не менее, классы в UE4 используют публичные элементы иногда. Это субъективное решение; должны элементы данных быть public или private на самом деле решать самому программисту.

С опытом, вы обнаружите, что иногда вы оказываетесь в положении, которое отчасти требует реорганизации кода, и вы делаете *публичными* элементы данных, которые должны были быть *частными*.

Класс против struct

Возможно, вы видели разные способы объявления объекта, используя ключевое слово class, вместо struct, как показано следующем коде:

```
class Player // здесь мы применили class вместо  struct!
{
  string name;
  //
};
```

Ключевые слова class и struct в C++ почти идентичны. Есть только одно отличие между class и struct. И оно в том, что элементы данных внутри struct будут объявлены как public по умолчанию. В то время как в class элементы данных будут объявлены по умолчанию как private. (Вот почему я ввёл объекты использующие struct. Я не хотел без объяснения ставить public, как первую строку для объекта class.)

В основном struct предпочтительно для простых типов, которые не используют инкапсуляцию, имеют не много функций-членов и должны быть обратно совместимы с зыком С. Классы всё ещё используются повсюду.

Отныне, давайте применять ключевое слово class вместо struct.

Геттеры и сеттеры

Вы должно быть заметили, что как только мы поставили private к определению класса Player, мы больше не можем считывать или записывать имя игрока извне класса Player.

Если мы попытаемся и считаем имя с помощью следующего кода:

```
Player me;
cout << me.name << endl;
```

Или запишем имя вот так:

```
me.name = "William";
```

Используя определение struct Player с private элементами, мы получим следующую ошибку:

main.cpp(24) : error C2248: 'Player::name' : cannot access private member declared in class 'Player' (невозможно получить доступ к частным элементам, объявленным в классе Player)

Это просто то, чего мы и просили, когда обозначали поле name как private. Мы сделали его совсем недоступным снаружи класса Player.

Геттеры

Геттер – получатель (также известен как функция доступа), используется, чтобы передавать обратно вызывающему, копии внутренних элементов данных. Чтобы считать имя игрока, мы украшаем класс Player функцией-членом специально, чтобы извлечь копию этого частного (private) элемента данных.

```
class Player
{
private:
   string name; // недоступно снаружи этого класса!
                // остальная часть класса, как и прежде
public:
   // Функция геттер извлекает копию переменной для вас
   string getName()
{
    return name;
}
};
```

Итак, теперь возможно считывать информацию об имени игрока. Мы можем делать это, используя следующее утверждение кода:

```
cout << player.getName() << endl;
```

Геттеры используются для извлечения частных элементов, которые иначе будут недоступны снаружи класса.

Совет

Совет от real world – ключевое слово const

Внутри класса, вы можете добавить ключевое слово const к объявлению функции-члена. Что делает это ключевое слово? Оно обещает компилятору, что внутреннее состояние объекта не будет изменяться в результате запуска этой функции. Присоединение ключевого слова const будет выглядеть вот так:

```
string getName() const
{
  return name;
}
```

Никакого присвоения элементам данных не будет происходить внутри функции-члена, отмеченной как const. И поскольку внутреннее состояние объекта гарантированно не будет меняться в результате запуска const функции, то компилятор может выполнять оптимизацию вызова функции для const функции-члена.

Сеттеры

Сеттер – установщик (также известен как модификатор функции и мутатор функции), функция-член, чьей единственной целью является изменение значения внутренней переменной в классе, как показано в следующем коде:

```
class Player
{
private:
   string name; // недоступно снаружи этого класса!
                // остальная часть класса, как и прежде
public:
   // Функция геттер извлекает копию переменной для вас
   string getName()
{
return name;
}
void setName( string newName )
{
   name = newName;
}
};
```

Так что мы всё ещё можем менять частную (private) функцию класса, снаружи этой функции, но только если мы делаем через функцию сеттер.

Но в чём же всё-таки идея операций get/set?

Итак, первый вопрос, который возникает в голове новоиспечённого программиста, когда он впервые сталкивается с операциями get/set для частных переменных, это: “не являются ли они обречёнными на провал?”. Я имею в виду, в чём смысл скрывать доступ к элементам данных, когда мы собираемся открыть эти же самые данные снова другим путём? Это как сказать: “Вы не можете взять шоколадки, потому что они, чьи то собственные, до тех пор, пока не скажете: «пожалуйста дайМнеШоколадку()». А потом, можете брать их”.

Некоторые опытные программисты даже сокращают функции get/set до одной строки:

```
string getName(){ return name; }
void setName( string newName ){ name = newName; }
```

Давайте ответим на этот вопрос. Не разрушает ли пара get/set инкапсуляцию, полностью открывая данные?

Ответ двоякий. Во первых, функция-член get в основном возвращает только копию элемента данных к которому создавала доступ. Это означает, что значение оригинального элемента данных остаётся защищённым не подлежит модификации через операцию get().

Set() (метод мутатор) операция немного противоречивая однако. Если сеттер проходит через (passthru) операцию, такую как void setName(string newName) { name=newName; }, затем наличие сеттера может показаться бессмысленным. В чём преимущество использования метода мутатор вместо переписывания переменной напрямую?

Аргумент для использования метода мутатора это написать дополнительный код перед назначением переменной, чтобы защитить переменную от принятия некорректного значения. Например, скажем, у нас есть сеттер для элемента данных hp, что выглядит таким образом:

```
void setHp( int newHp )
{
    // защита переменной hp от принятия отрицательного значения
    if( newHp < 0 )
    {
        cout << "Ошибка, hp игрока не может быть меньше чем 0" << endl;
        newHp = 0;
    }
    hp = newHp;
}
```

Метод мутатора предполагает предотвращение принятия отрицательного значения внутренним элементом данных hp. Вы можете рассматривать метод мутатора отчасти имеющим обратную силу. Должна ли ответственность лежать на вызывающем коде, в проверке значения, которое он установил до вызова setHp(-2), и не допускать этого, как только оно попадается в методе мутатора? Можете ли вы использовать публичную переменную-член и возложить ответственность в проверке того, что переменная не принимает недействительных значений в вызове кода, а не в сеттере? Да, вы можете.

Тем не менее, это суть применения метода мутатора. Идея метода мутатора в том, что вызывающий код может передавать любое значение которое он хочет для функции setHp (например, setHp(-2)), без необходимости беспокоиться о том, будет ли значение, передаваемое функции, пригодным или нет. Затем функция setHp принимает ответственность, в том, чтобы убедиться, является ли значение для переменной hp пригодным.

Некоторые программисты считают прямые функции мутатор, такие как getHp()/setHp() дурно пахнущим кодом. Дурно пахнущий код – это в целом плохая практика программирования, которую откровенно не замечают, за исключением лёгкого чувства, что, что-то было сделано не оптимально. Они спорят о том, что вместо мутаторов могут быть написаны функции-члены более высокого уровня. Например, вместо функции-члена setHp() у нас должны быть публичные функции-члены, такие как восстанавливать() и наноситьУрон(). Статья по этой теме доступна на http://c2.com/cgi/wiki?AccessorsAreEvil.

Конструкторы и деструкторы

Конструктор в вашем C++ коде, это простая маленькая функция, которая запускается один раз, когда впервые создан C++ объект. Деструктор запускается один раз, когда объект C++ ликвидируется. Скажем, у нас есть следующая программа:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
private:
    string name; // недоступно снаружи этого класса!
public:
    string getName(){ return name; }
// Конструктор!
    Player()
    {
        cout << "Объект Player сконструирован" << endl;
        name = "Diplo";
    }
    // ~Деструктор (~ это не опечатка!)
```

```
    ~Player()
    {
        cout << "Объект Player ликвидирован" << endl;
    }
};
int main()
    {
        Player player;
        cout << "Player назван "' << player.getName() << "'" << endl;
    }
    // объект игрока ликвидирован здесь
```

Итак, здесь мы создали объект Player. Вывод этого кода будем следующим:

```
Объект Player сконструирован
Player назван 'Diplo'
Объект Player ликвидирован
```

Первое, что происходит в ходе конструирования, собственно запускается конструктор. Что выводит строку: "Объект Player сконструирован". Следом за этим, выводится строка с именем игрока: "Player назван 'Diplo'". Почему игрок назван Diplo? Потому что это имя назначено в конструкторе Player().

И вот, в конце программы, вызывается деструктор игрока, и мы видим: " Объект Player ликвидирован". Объект игрока ликвидируется, когда он выходит за область действия в конце main() (за предел } утверждения main).

Итак, для чего хороши конструкторы и деструкторы? Определённо они появляются для: установки и удаления объекта. Конструктор может использоваться для назначения начальных полей данных, а деструктор для вызова удаления любых динамически распределённых ресурсов (мы ещё не проходили динамически распределённые ресурсы, так что не беспокойтесь об этом пока).

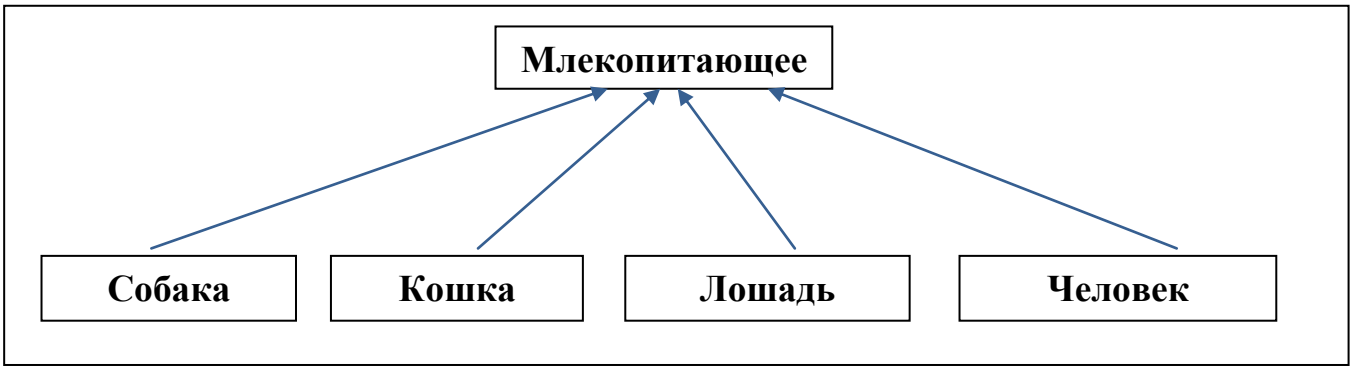
Наследование класса

Вы используете наследование, когда вы хотите создать новый, более функциональный класс кода, основанный на существующем классе кода. Наследование важная тема для прохождения. Давайте начнём с понятия производный класс (или подкласс).

Производные классы

Наиболее естественный способ рассматривать наследование, путём аналогии с

царством животных. Классификация живых существ показана на следующем изображении:



Эта диаграмма означает, что **Собака**, **Кошка**, **Лошадь** и **Человек** являются **Млекопитающими**. Это означает, что собака, кошка, лошадь и человек обладают общими характеристиками, такими как общая система органов (мозг с неокортексом, лёгкие, печень и матка для особей женского пола), в то же время абсолютно отличаясь друг от друга в других отношениях.

Что бы это значило, если бы вы писали код существ? Вам нужно было бы лишь раз спрограммировать общую функциональность. Затем, вы применили бы код для различных частей, специально для каждого класса собак, кошек, лошадей и людей.

Конкретный пример для схемы сверху:

```
#include <iostream>
using namespace std;
class Mammal // класс Млекопитающее
{
protected:
    // защищённые (protected) переменные похожи на частные (private)  переменные.
    // они доступны в этом классе, но не снаружи класса.
    // разница между protected и private в том,
    // что protected означает доступность и для происходящих подклассов
int hp;
    double speed;

public:
    // Конструктор млекопитающих – запускается ПЕРВЫМ до конструкторов классов-потомков!
Mammal()
{
    hp = 100;
    speed = 1.0;
    cout << "Млекопитающее создано!" << endl;
}
~Mammal()
{
    cout << "Млекопитающее скончалось!" << endl;
}
// Общая функция для всех Млекопитающих и производных
    void breathe()
    {
        cout << "Вдох... выдох" << endl;
    }
```

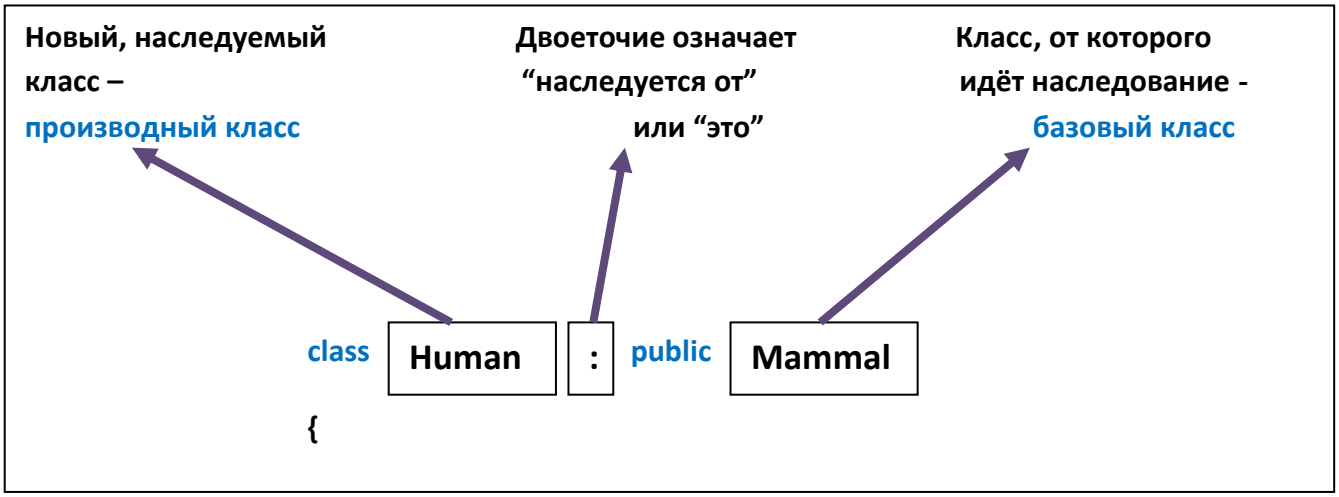
```
    virtual void talk()
    {
        cout << "Млекопитающее говорит… подмените эту функцию!" << endl;
    }
    // Чистая виртуальная функция, (объясняется ниже)
    virtual void walk() = 0;
};
// Следующая строка говорит "класс Собака наследуется от класса Млекопитающее"
class Dog : public Mammal // : двоеточие используется для наследования
{
public:
    Dog()
    {
        cout << "Собака родилась!" << endl;
    } ~
    Dog()
    {
        cout << "Собака ушла в мир иной" << endl;
    }
    virtual void talk() override
    {
        cout << "Гав!" << endl; // собаки говорят только гав!
    }
    // осуществление ходьбы для собаки
    virtual void walk() override
    {
        cout << "Левая передняя лапа и задняя правая лапа, правая передняя лапа и левая
    задняя лапа… со скоростью" << speed << endl;
    }
};

class Cat : public Mammal // класс Кошка наследуется от класса Млекопитающее

{
public:
    Cat()
    {
        cout << "Кошка родилась" << endl;
    }
    ~Cat()
    {
        cout << "Кошка ушла в мир иной" << endl;
    }
    virtual void talk() override
    {
        cout << "Мяу!" << endl;
    }
    // осуществление ходьбы для кошки… такое же как и для собаки!
    virtual void walk() override
    {
        cout << " Левая передняя лапа и задняя правая лапа, правая передняя лапа и левая
    задняя лапа… со скоростью " << speed << endl;
    }
};
class Human : public Mammal // класс Человек наследуется от класса Млекопитающее
{
```

```
// Уникальный элемент данных для Человека (в отличии  от других Млекопитающих)
    bool civilized;
public:
    Human()
    {
        cout << "Новый человек родился" << endl;
        speed = 2.0; // меняем скорость. Так как конструктор класса-потомка
                     // запускается после конструктора базового класса,
                     // устанавливаются специальные для этого класса
                     // переменные-члены
        civilized = true;
    }
    ~Human()
    {
        cout << "Человек ушёл в мир иной" << endl;
    }
    virtual void talk() override
    {
        cout << "Я хорошо выгляжу для... человека" << endl;
    }
    // осуществление ходьбы для человека..
    virtual void walk() override
    {
        cout << "Левой, правой, левой, правой со скоростью " << speed << endl;
    }
    // функция-член уникальна для производного человек
    void attack( Human & other )
    {
    // Человек не будет нападать, если он цивилизован
    if( civilized )
        cout << "Почему один человек должен нападать на другого? Я отказываюсь" << endl;
    else
        cout << "Человек нападает на другого!" << endl;
    }
};
int main()
{
    Human human;
    human.breathe(); // для дыхания используется функционал базового класса Млекопитающее
    human.talk();
    human.walk();

    Cat cat;
    cat.breathe(); // для дыхания используется функционал базового класса Млекопитающее
    cat.talk();
    cat.walk();

    Dog dog;
    dog.breathe();
    dog.talk();
    dog.walk();
}
```

Собака, Кошка и Человек, все наследуются от класса Млекопитающее. Это значит, что собака, кошка и человек являются млекопитающими, и даже больше.

Синтаксис наследования

Синтаксис наследования довольно прост. Давайте в качестве примера возьмём определение класса Человек. Далее демонстрируется типичное утверждение наследования:



Класс слева от двоеточия, это новый производный класс. А класс справа от двоеточия, это базовый класс.

Что делает наследование?

Идея наследования заключается в том, чтобы производный класс принимал все характеристики (элементы данных, функции-члены) базового класса, а затем расширял их с ещё большей функциональностью. Например, все млекопитающие обладают функцией breathe() (дышать). Наследуясь от класса Млекопитающих, классы Собака, Кошка и Человек автоматически приобретают возможность дышать (breathe()).

Наследование сокращает повторение кода, так как нам не надо повторно осуществлять общую функциональность (такую как breathe()) для Собаки, Кошки и Человека. Вместо этого, каждый из этих производных классов использует вместе с остальными функцию breathe(), определённую в классе Млекопитающее.

Тем не менее, только класс Человек имеет функцию-член attack(). Это означает, что в нашем коде только класс Человек нападает, атакует. Функция cat.attack() вызовет ошибку компилятора, если только вы не напишете функцию-член attack() внутри class Cat (или в class Mammal).

Отношение “это”

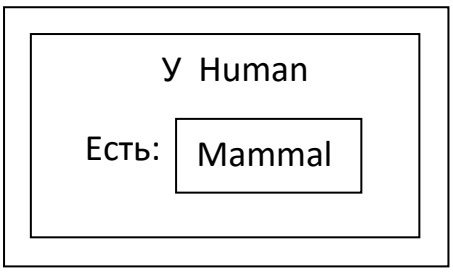
Наследование часто говорит “это” отношение. Когда класс Человек наследуется от класса Млекопитающее, то это говорит о том, что человек “это” млекопитающее.

Человек наследует все особенности присущие Млекопитающему

Например, объект Human, содержит функцию Mammal внутри, как в примере ниже:

```
class Human
{
   Mammal mammal;
};
```

В этом примере, мы говорим, что у Human где-то есть Mammal (что имело бы смысл, если бы человек, а именно, разумеется женщина, была бы беременна или как-либо ещё держала бы ребёнка (mammal), например, нянчила бы).



У этого класса Human есть некоторого рода млекопитающее (mammal) добавленное к нему

Помните как ранее объекту Player, мы давали Armor. И не имело бы смысла объекту Player наследоваться от класса Armor, потому что не будет смысла, если сказать Игрок (Player) это Броня (Armor). Когда при разработке кода решаете, наследоваться ли одном классу от другого или нет (например, класс Human наследуется от класса Mammal), вы всегда должны быть способны сказать что-то в таком роде как: Человек *это* Млекопитающее. Если *это* звучит неправильно, то похоже, наследование является неверным отношением для этой пары объектов.

В предыдущем примере мы ввели пару новых ключевых слов C++. Первое это protected.

Защищённые переменные

Переменная-член с ключевым словом protected (защищённая), отличается от переменных с private или public. Все три этих класса переменных доступны внутри класса, в котором они определенны. Разница между ними в отношении доступности снаружи класса. Публичная (public) переменная доступна везде, как внутри класса, так и снаружи. Частная (private) переменная доступна внутри класса, но не снаружи. Защищённая (protected) переменная доступна внутри класса, а также внутри происходящего подкласса, но не снаружи класса. Так что элементы hp и speed класса Mammal будут доступны классах-потомках Dog, Cat и Human, но не снаружи этих классов (например в main()).

Виртуальные функции

Виртуальная функция – это функция-член, чьё осуществление может быть подменено в классе-потомке. В этом примере функция-член talk() (определена в class Mammal) отмечена как virtual. Это означает, что классы-потомки могут выбирать, либо могут не выбирать осуществлять их собственную версию того, что функция-член talk() значит.

Чисто виртуальные функции (и абстрактные классы)

Чисто виртуальные функции это из тех, чьё осуществление вам требуется подменить в классах-потомках. Функция walk() в классе Mammal чисто виртуальная. Она была объявлена таким образом:

```
virtual void walk() = 0;
```

Вот эта “= 0” часть в конце предыдущего кода, это то, что делает функцию чисто виртуальной.

Функция walk() в классе Mammal чисто виртуальная и это делает класс Mammal абстрактным. Абстрактный класс в C++ это любой класс, имеющий как минимум одну виртуальную функцию.

Если класс содержит чисто виртуальную функцию и является абстрактным, то этот класс не может быть использован напрямую для создания экземпляра. То есть, вы не можете создать объект Mammal сейчас, от записи чисто виртуальной функции walk(). Если вы попытаетесь выполнить следующий код, вы получите ошибку:

```
int main()
{
   Mammal mammal;
}
```

Если вы попытаетесь создать объект Mammal, то вы получите следующую ошибку:

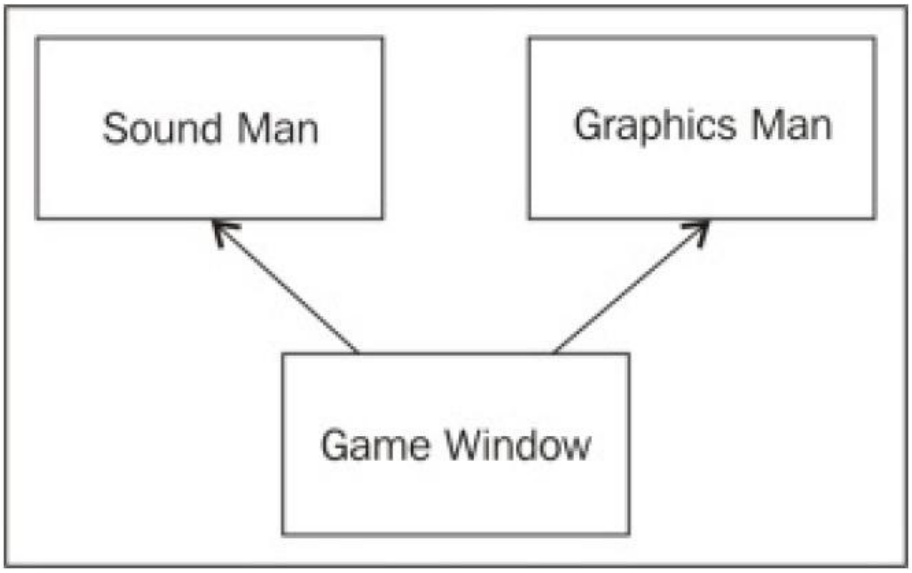
error C2259: 'Mammal' : cannot instantiate abstract class (невозможно создать экземпляр абстрактного класса)

Однако вы можете создавать экземпляры производных от класса Mammal, пока классы-потомки будут иметь осуществлённые виртуальные функции-члены.

Множественное наследование

Не всё множественное так хорошо, как это звучит. Множественное наследование – это когда производный класс наследуется более чем от одного базового класса. Обычно, это работает без помех, если множественные базовые классы от которых мы наследуем, не имеют совершенно ничего общего друг с другом.

Например, у нас может быть класс Window, который наследуется от базовых классов SoundManager и GraphicManager. Если SoundManager предоставляет функцию-член playSound(), а GraphicManager предоставляет функцию-член drawSprite(), тогда класс Window будет в состоянии использовать эти дополнительные характеристики без проблем.



Game Window наследуемое от Sound Man и Graphic Man, означает, что Game Window будет иметь обе установленные характеристики

Тем не менее, множественное наследование может иметь негативные последствия. Скажем, мы хотим создать класс Мул, который происходит от классов Ослица и Конь. Классы Ослица и Конь в свою очередь наследуются от класса Млекопитающее. У нас мгновенно возникает вопрос! Если мы вызовем функцию mule.talk() (мул.говорить()), а для мула мы не подменяли функцию talk(), то какая функция-член должна быть запущена, от класса Ослица или от класса Конь? Это не ясно.

Частное наследование

Мало сказано об особенности C++ частном наследовании. Когда один класс происходит от другого класса публично, то это становится известно всему коду, к родительскому классу которого он принадлежит. Например:

```
class Cat : public Mammal
```

Это означает, что весь этот код будет знать, что Cat это объект от Mammal. И будет возможно указывать на экземпляр Cat* используя указатель базового класса Mammal. Например, следующий код будет действительным:

```
Cat cat;
Mammal* mammalPtr = &cat; // Указывает на Cat так если бы он был
                          // Mammal
```

Этот код хорош, если Cat наследуется от Mammal публично. Частное наследование это, когда коду снаружи класса Cat не позволено знать родительский класс:

```
class Cat : private Mammal
```

Здесь, внешне вызываемый код не будет “знать”, что класс Cat происходит от класса Mammal. Привидение типов экземпляра Cat к базовому классу Mammal не разрешено компилятором, когда наследование частное (private). Используйте частное наследование, когда вам нужно скрыть факт того, что определённый класс происходит от определённого родительского класса.

Однако частное наследование редко применяется на практике. Большинство классов используют лишь публичное наследование. Если вы желаете знать больше о частном наследовании, посмотрите:

http://stackoverflow.com/questions/406081/why-should-i-avoid-multiple-inheritance-in-c.

Помещаем ваш класс в заголовочный файл

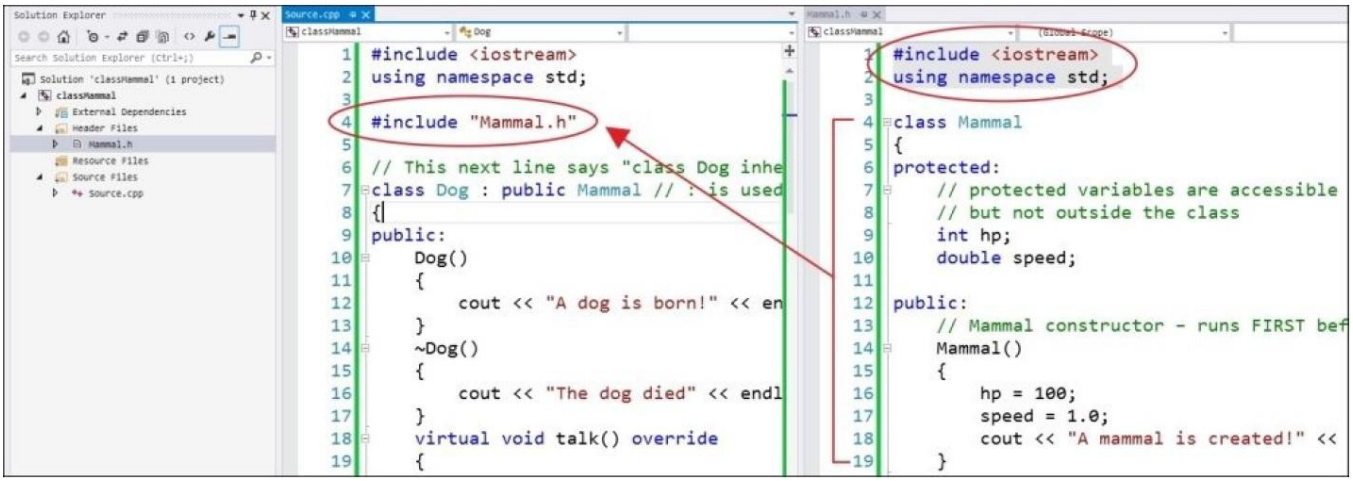
Уже давно, наши классы просто вставлялись перед main(). Если вы продолжите программировать таким образом, то ваш код весь будет в одном файле и выглядеть как один большой беспорядок.

Поэтому, хорошая практика в программировании это организовывать ваши классы в отдельных файлах. Так редактировать каждый код класса индивидуально, становится гораздо легче, когда в проекте множество классов.

Возьмите класс Mammal и его производные классы из того, что мы делали ранее. Мы как полагается, организуем этот пример в отдельные файлы. Для этого давайте выполним следующие шаги:

1. Создайте новый файл в вашем проекте C++ и назовите его Mammal.h. Вырежьте и вставьте весь класс Mammal целиком в этот файл. Заметьте, что как только класс Mammal включает в себя, использование cout, мы пишем утверждение #include <iostream> в этом файле также.
2. Напишите утверждение “#include Mammal.h” вверху вашего файла Source.cpp.

Пример того, как это выглядит, показан на следующем скриншоте:



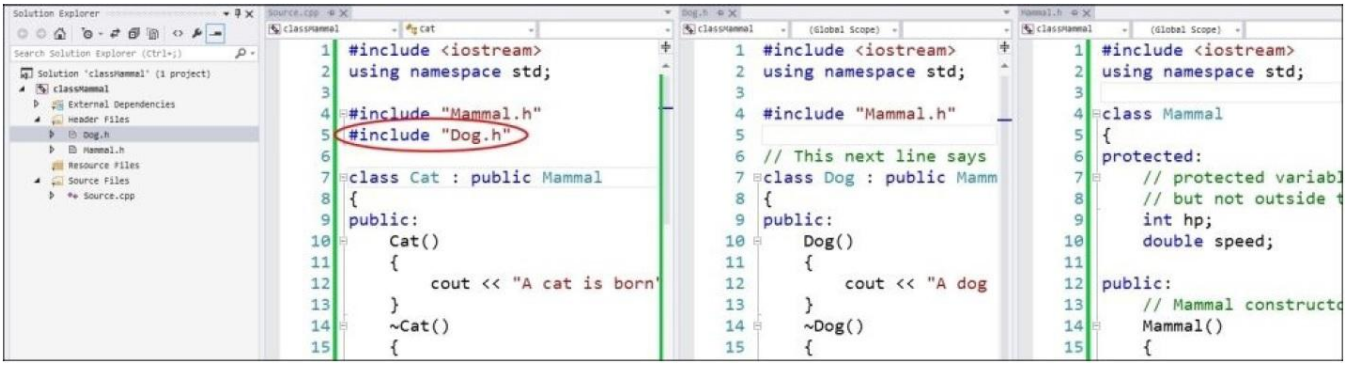
Что происходит здесь, когда код компилирован? Весь класс Mammal копируется и вставляется (#include) в файл Source.cpp, который содержит функцию main(), а все остальные классы происходят от Mammal. Поскольку #include это функция копирования и вставления, то код будет работать абсолютно так же, как и до этого. Отличие только в том, что он будет гораздо лучше организован и его будет удобней и легче просматривать. С этого шага компилируйте и запустите ваш код, чтобы убедиться, что он по-прежнему работает.

Совет

Почаще проверяйте, чтобы ваш код компилировался и запускался. Особенно когда проводите рефакторинг. Если вы не знаете правил, вы неизбежно наделаете много ошибок. Вот почему вам следует выполнять рефакторинг только небольшими шагами. Рефакторингом называются действия, которые мы выполняем сейчас – мы реорганизуем источник, чтобы облегчить понимание другим программистам читающим нашу кодовую базу. Рефакторинг обычно не влечёт к переписыванию слишком многого.

Следующее, что вам надо сделать, это изолировать классы Dog, Cat и Human в их собственные файлы. Чтобы сделать это, создайте файлы Dog.h, Cat.h, Human.h и добавьте их в ваш проект.

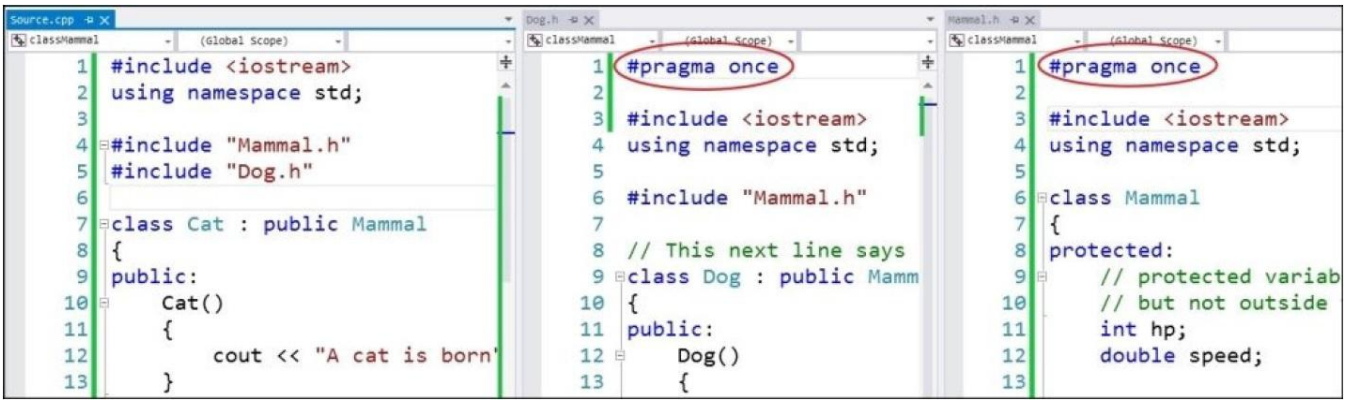
Давайте начнём с класса Dog, как показано на следующем скриншоте:



Если вы примените точно такую же установку и попытаетесь компилировать, и запустить ваш проект, вы увидите ошибку: **‘Mammal’ : ‘class’ type redefinition** (‘Mammal’ : переопределение типа ‘class’); как показано на следующем скриншоте:

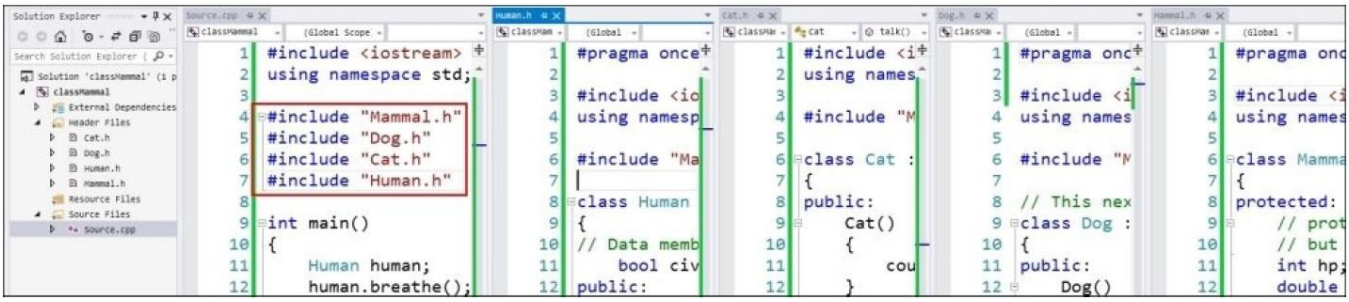
Что означает эта ошибка? То, что Mammal.h дважды внесён в ваш проект. Один раз в Source.cpp и затем, ещё раз в Dog.h. Это означает, что фактически две версии класса Mammal были добавлены в компилируемый код и C++ не знает, какую версию использовать.

Есть два способа решить этот вопрос, но наилегчайший (и тот который применяется Unreal Engine) это макрос #pragma once, как показано на следующем скриншоте:



Мы пишем #pragma once вверху каждого заголовочного файла. Таким образом, когда Mammal.h включается второй раз, компилятор не копирует и не вставляет его содержимое снова, поскольку он уже был включен до этого и его содержимое уже на самом деле в компилируемой группе файлов.

Сделайте то же самое для Cat.h и Human.h. Затем включите их в ваш файл Source.cpp, где постоянно находится ваша функция main().



Все классы включены

Теперь, когда мы включили все классы в ваш проект, код должен компилироваться и запускаться.

.h и .cpp

Следующий уровень организации, это оставить объявления класса в заголовочном файле (.h) и поместить сами тела осуществления функций в новые файлы .cpp. А также оставить существующие элементы в объявлении класса Mammal.

Для каждого файла выполните следующие операции:

1. Удалите тело всех функций (код между { и }) и просто замените их на двоеточие. Для класса Mammal, это будет выглядеть так:

```
// Mammal.h
#pragma once
class Mammal
{
protected:
    int hp;
    double speed;

public:
    Mammal();
    ~Mammal();
    void breathe();
    virtual void talk();
    // чистая виртуальная функция,
    virtual void walk() = 0;
};
```

2. Создайте новый файл .cpp и назовите его Mammal.cpp. Затем просто поместите тело функций в этот класс:

```
// Mammal.cpp
#include <iostream>
using namespace std;

#include "Mammal.h"
Mammal::Mammal() // Обратите внимание на: use of :: (оператор разрешения области
                 // видимости)
{
```

```
    hp = 100;
    speed = 1.0;
    cout << "Млекопитающее создано!" << endl;
}
Mammal::~Mammal()
{
    cout << "Млекопитающее скончалось!" << endl;
}
void Mammal::breathe()
{
    cout << "Вдох… выдох" << endl;
}
void Mammal::talk()
{
    cout << "Млекопитающее говорит... подмените эту функцию!" << endl;
}
```

Важно обратить внимание на использование имени класса и на оператор разрешения области видимости (двойное двоеточие), когда объявляете тело функции-члена. Всем функциям-членам, принадлежащим к классу Mammal, мы добавляем приставку Mammal::.

Обратите внимание на то, что чисто виртуальная функция не имеет тела, собственно так и должно быть! Чисто виртуальные функции просто объявлены (и им присвоено начальное значение 0) в базовом классе, но осуществляются позже в производных классах.

Упражнение

Завершите разделение классов разных существ сверху в заголовочный класс (.h) и файл определения класса (.cpp).

Выводы

Вы узнали об объектах в C++, которые являются частями кода связывающими элементы данных и функции-члены в связки кода называемые классами (class) или структурами (struct). Объектно-ориентированное программирование означает, что ваш код будет наполнен предметами, а не просто переменными int, float и char. У вас будет переменная, которая представляет бочку (Barrel), ещё переменная, которая представляет игрока (Player) и так далее. То есть будут переменные для представления всех сущностей в вашей игре. Вы будите способны повторно использовать код при помощи наследования. Если вам надо было написать код для реализации классов Cat и Dog, вы могли написать код общей функциональности в базовом классе Mammal. Мы также обсудили инкапсуляцию и то, насколько легче и эффективней программировать объекты так, чтобы они поддерживали своё внутреннее состояние.

Глава 7. Динамическое распределение памяти

В предыдущей главе, мы говорили об определениях классов и о том как продумывать ваш собственный класс. Мы обсудили как, продумывая наши собственные классы, мы можем сконструировать переменные представляющие сущности в вашей игре или программе.

В этой главе, мы поговорим о динамическом распределении памяти и о том, как создавать место в памяти для группы объектов.

Предположим, что мы имеем упрощённую версию класса Player, как ранее, лишь с конструктором и деструктором:

```
class Player
{
    string name;
    int hp;
public:
    Player(){ cout << "Player born" << endl; }   // Игрок родился
    ~Player(){ cout << "Player died" << endl; } // Игрок умер
};
```

Ранее мы говорили об области действия переменной в C++. Чтобы освежить воспоминания, напомним, что область действия переменной в целом ограничивается блоком, в котором она объявлена. А блок - это просто, любая секция кода содержащаяся между фигурными скобками ({ и }). Вот простая программа демонстрирующая область действия переменной:

```
int main()
{                                   начинаем блок main()
    int x;                          область действия x отсюда и до конца main()

    if( какое-нибудь_условие )
    {                               начинаем блок if
        int y;                      область действия y отсюда и до конца if
    }                               заканчиваем блок if [y ликвидируется]
}                                   заканчиваем блок main() [x ликвидируется]
```

В этом примере программы, область действия переменной x проходит по всему main(). А область действия переменной y только внутри блока if.

Ранее мы упоминали, что переменные ликвидируются, когда они выходят из области действия. Давайте испробуем эту идею с экземплярами класса Player:

```
int main()
{
    Player player; // "Игрок родился"
}                  // "Игрок сгинул" – объект игрока ликвидируется здесь
```

Вывод этой программы следующий:

```
Игрок родился
Игрок умер
```

Деструктор для объекта игрока вызван в конце области действия этого объекта. Поскольку область действия переменной это блок в котором она определена, что составляет три строки кода, то объект Player будет ликвидирован немедленно в конце main(), когда он выходит из области действия.

Динамическое распределение памяти

Теперь, давайте попробуем динамически распределить объект Player. Что это значит?

Мы используем ключевое слово new, чтобы распределить объект!

```
int main()
{
    // "динамическое распределение" – используя ключевое слово new!
    // этот стиль распределения означает, что объект игрока
    // НЕ будет автоматически удалён в конце блока, в котором
    // он объявлен!
Player *player = new Player();
}  // НИКАКОГО автоматического удаления!
```

Вывод этой программы таков:

```
Игрок родился
```

Игрок не умирает! Как мы убиваем игрока? Мы должны прямо вызвать delete на указателе player.

Ключевое слово delete

Оператор delete запускает деструктор для объекта, который удалён, как показано в следующем коде:

```
int main()
{
    // "динамическое распределение" – используя ключевое слово new!
    Player *player = new Player();
    delete player; // удаление запускает деструктор
}
```

Вывод этой программы следующий:

```
Игрок родился
Игрок умер
```

Итак, только “нормального” (или “автоматического”, также называемые как типы не-указатели) типа переменные ликвидируются в конце блока, в котором они были объявлены. Тип указатель (переменная объявленная со знаком * и ключевым словом new) не ликвидируются автоматически, даже когда они выходят из области действия.

Каково предназначение этого? Динамическое распределение даёт вам контроль когда объект создаётся и ликвидируется. Позже это станет более понятно.

Утечка памяти

Итак, динамически распределённые объекты, созданные со словом new, не удаляются автоматически, пока вы напрямую не вызываете delete для них. Тут есть риск! Это называется *утечка памяти*. Утечка памяти происходит, когда объект, распределённый со словом new, вообще не удалился. И что может случиться? Если много объектов в вашей программе распределены со словом new и затем вы прекращаете использовать их, то в конце концов, в вашем компьютере иссякнет память из-за утечки памяти.

Вот нелепый пример программы для демонстрации проблемы:

```
#include <iostream>
#include <string>
using namespace std;
class Player
{
    string name;
    int hp;
public:
    Player(){ cout << "Игрок родился" << endl; }
    ~Player(){ cout << "Игрок умер" << endl; }
};

int main()
{
    while( true ) // продолжается вечно,
    {
        // распределение...
        Player *player = new Player();
        // без delete == Утечка памяти!
    }
}
```

Эта программа, если позволить ей запускаться достаточно долго, в конце концов, поглотит память компьютера, как показано на следующем скриншоте:

Name	Status	CPU	Memory
dynmem.exe (32 bit)		26.3%	1,961.9 MB

2 ГБ ОЗУ используется для объектов Player!

Заметьте, что никто намеренно не пишет программу с проблемой такого типа в ней! Проблема утечки памяти происходит случайно. Вы должны быть внимательными с вашим распределением памяти и удалять (delete) объекты, которые больше не используются.

Обычные массивы

Массив в C++ может быть объявлен следующим образом:

```
#include <iostream>
using namespace std;
int main()
{
    int array[ 5 ]; // объявляем "массив" пяти элементов
                    // заполняем ячейки 0-4 значениями
array[ 0 ] = 1;
array[ 1 ] = 2;
array[ 2 ] = 3;
array[ 3 ] = 4;
array[ 4 ] = 5;
    // выводим содержание
    for( int index = 0; index < 5; index++ )
        cout << array[ index ] << endl;
}
```

В памяти это выглядит как то так:

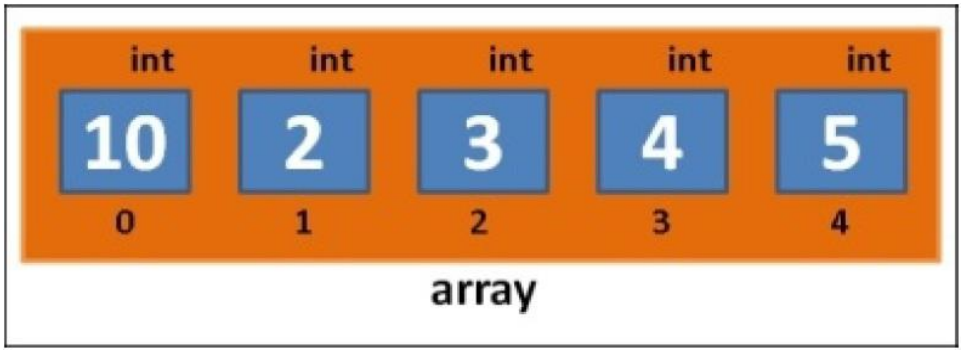
Вот так, в переменной массиве пять ячеек или элементов. В каждой ячейке обычная переменная типа int.

Синтаксис массивов

Итак, как вы получите доступ к одному из int значений в массиве? Чтобы получить доступ к индивидуальному элементу массива, мы используем квадратные скобки, как показано в следующей строке кода:

```
array[ 0 ] = 10;
```

Эта строка кода меняет элемент в ячейке массива 0 на 10:



В целом, чтобы получить конкретную ячейку массива, вы будете писать следующее:

```
array[ slotNumber ] = value to put into array;
```

Запомните, что индексация ячеек массива всегда начинается с 0. Чтобы попасть в первую ячейку массива, используйте array[0]. Вторая ячейка массива это array[1] (не array[2]). Последняя ячейка массива, что показан у нас сверху это array[4] (не array[5]). Тип данных array[5] за пределами массива! (В массиве с предыдущего изображения нет ячейки с индексом 5. Последний индекс в данном массиве это 4.)

Не выходите за границы массива! Иногда это может сработать, но во всех остальных случаях ваша программа выйдет из строя с **нарушением доступа памяти** (попытка получения доступа к памяти, которая не принадлежит вашей программе). В целом, доступ к памяти, которая не принадлежит вашей программе, послужит причиной выхода из строя вашего приложения. И если это и не произойдёт сразу, то будет скрытый баг в вашей программе, который создаст проблемы. Вы всегда должны быть внимательны при индексации массива.

Массивы встроены в C++, поэтому, вам не надо вносить что-то специальное, чтобы использовать массивы сразу. У вас могут быть массивы любого тип данных, какого захотите. Например, массивы типов int, double, string и даже ваши собственные объектные типы (Player).

Упражнения

1. Создайте массив из пяти строковых значений и поместите в него имена (любые, не имеет значения).
2. Создайте двумерный (double) массив с трёмя элементами и назовите его temps. И сохраните в нём температуру за последние три дня.

Решения

1. Вот пример программы с массивом пяти строковых значений:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string array[ 5 ]; // объявляем массив - "array" из 5 строковых значений
                      // заполняем ячейки 0-4 значениями
array[ 0 ] = "Mariam McGonical";
array[ 1 ] = "Wesley Snice";
array[ 2 ] = "Kate Winslett";
array[ 3 ] = "Erika Badu";
array[ 4 ] = "Mohammed Benaziza";
   // выводим содержание
   for( int index = 0; index < 5; index++ )
      cout << array[ index ] << endl;
}
```

2. Здесь просто массив:

```
double temps[ 3 ];
// заполняем ячейки 0-2 значениями
temps[ 0 ] = 0;
temps[ 1 ] = 4.5;
temps[ 2 ] = 11;
```

C++ стиль массивов динамического размера (new[] и delete[])

Возможно, случалось и с вами такое, что мы не можем всегда знать размер массива, когда начинаем писать программу. Нам нужно распределять размер массива динамически.

Тем не менее, если вы попробовали сделать это, вы заметили, что это не работает!

Давайте попробуем и применим команду cin, чтобы принять размер массива от пользователя. Давайте спросим пользователя, насколько большой ему нужен массив и попробуем создать массив такого размера для него:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Насколько большой?" << endl;
    int size; // пробуем применить переменную для размера...
    cin >> size; // получаем размер от пользователя
    int array[ size ]; // получаем ошибку: "unknown size" (неизвестный размер)
}
```

Мы получаем следующую ошибку:

```
error C2133: 'array' : unknown size (неизвестный размер)
```

Проблема в том, что компилятор хочет распределить размер массива. И пока размер переменной не будет обозначен как const, компилятор не будет уверен в его значении во время компиляции. Компилятор C++ не может задавать размер массиву во время компиляции, так что он выдаёт ошибку.

Чтобы исправить это, нам нужно распределять массив динамически (на “куче”):

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Насколько большой?" << endl;
    int size; // пробуем применить переменную для размера...
    cin >> size;
    int *array = new int[ size ]; // это работает
    // заполняем массив и выводим
for( int index = 0; index < size; index++ )
{
    array[ index ] = index * 2;
    cout << array[ index ] << endl;
}
delete[] array; // должны вызвать delete[] на распределении массива со словом
                // new[]!
}
```

Итак, урок тут заключается в следующем:

- Чтобы распределить массив какого-нибудь типа (например int) динамически, вы должны использовать новый int[числоЭлементовМассива].
- Массивы распределённые с new[], позже должны быть удалены с delete[], иначе вы получите утечку памяти! (этот delete[] с квадратными скобками! А не обычное удаление).

Массивы динамического C-стиля

Массивы C-стиля являются темой наследия, но их всё ещё стоит обсуждать, так как, несмотря на то, что они стары, вы всё ещё можете увидеть их иногда.

Мы объявляем массив С-стиля следующим образом:

```
 #include <iostream>
using namespace std;
int main()
{
    cout << "Насколько большой?" << endl;
    int size; // пробуем применить переменную для размера...
    cin >> size;
    // следующая строка будет выглядеть странно...
    int *array = (int*)malloc( size*sizeof(int) ); // C-style
    // заполняем массив и выводим
for( int index = 0; index < size; index++ )
    {
        array[ index ] = index * 2;
        cout << array[ index ] << endl;
    }
free( array ); // должны вызвать free() для массива распределённого с
                // malloc() (не delete[]!)
}
```

Разница здесь выделена жирным шрифтом.

Массив С-стиля создан с использованием функции malloc(). Слово malloc означает “memory allocate” – “распределяет память”. Эта функция требует, чтобы для создания вы передали размер массива в битах, а не просто количество желаемых элементов в массиве. По этой причине, мы умножаем число запрашиваемых элементов (размер) посредством sizeof – размера типа внутри массива. Размеры в битах, нескольких частых С++ типов, перечислены в следующей таблице:

Примитивный С++ тип	**sizeof (размер в битах)**
int	4
float	4
double	8
long long	8

Память, распределённая функцией malloc(), позже должна быть освобождена с применением free().

Выводы

Эта глава ознакомила вас со стилями массивов C и C++. В большинстве кода UE4, вы будете использовать коллекцию классов (TArray<T>) встроенных в редактор UE4. Тем не менее вам необходимо быть знакомым с базовыми стилями массивов C и C++, чтобы быть очень хорошим программистом.

Глава 8. Действующие лица и пешки

Теперь мы реально погрузимся в код UE4. Поначалу, это всё может выглядеть обескураживающим. Фреймворк класса UE4 очень обширный, но не беспокойтесь по этому поводу. Фреймворк обширный, а ваш код не обязан быть таким. Вы обнаружите, что вы можете выполнить много всего и отобразить много всего на экране, используя относительно небольшой код. Это потому что код движка UE4 такой всесторонний и хорошо спрограммированный, что делает возможным выполнение любой относящейся к игре задачи с лёгкостью. Просто вызовите правильную функцию и вуаля, то что вы хотите увидеть, появится на экране. Вся идея фреймворка в том, что он разработан так, чтобы позволить вам получить желаемый вами сюжет игры без надобности проводить кучу времени убиваясь над деталями.

Actor против pawn

В этой главе, мы обсудим такие понятия как actor (актор) и pawn (пешка). Хотя это и звучит так, как будто pawn будет более базовым классом, чем actor, на самом деле всё наоборот. Объект актор в UE4 (класс Actor) – это базовый тип того, что может быть помещено в игровом мире UE4. Перед тем как поместить что-либо в мире UE4, вам нужно выполнить происхождение этого от класса Actor.

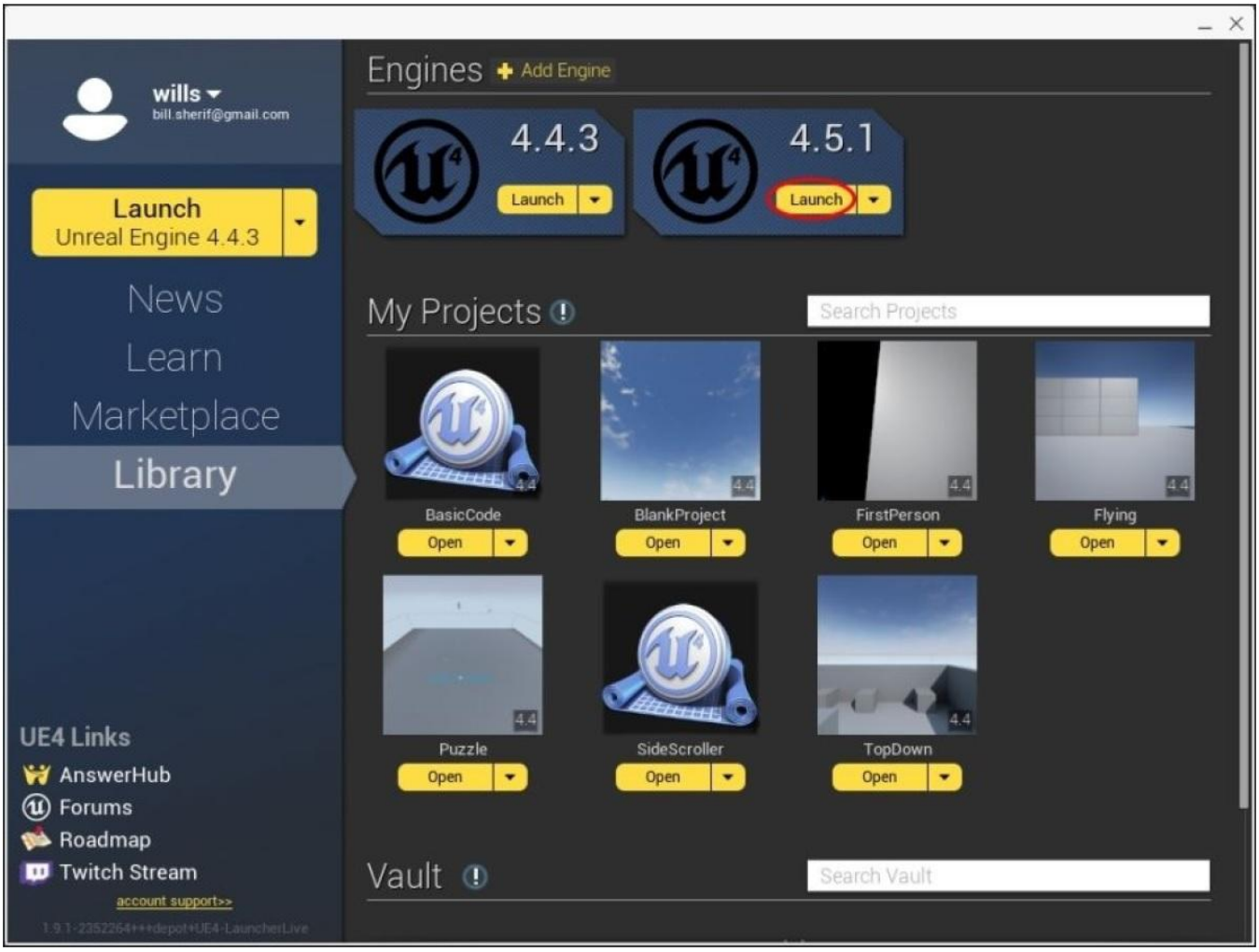
Pawn (пешка) – это объект, представляющий что-либо, что вы или Искусственный Интеллект (AI – Artificial Intelligence) компьютера можете контролировать на экране. Класс Pawn происходит от класса Actor с дополнительной способностью быть контролируемым напрямую игроком или скриптом AI. Когда пешка или актор управляются контроллером или AI, это говорит о том, что этот контроллер или AI обладают ими.

Представьте класс Actor как персонаж в игре. Ваш игровой мир будет сформирован из связки акторов, которые действуя все вместе, заставляют работать сюжет игры. Персонажи игры, Неигровые Персонажи (**Non-player Character** (**NPC**)) и даже сундуки с сокровищами, все будут акторами.

Создание мира для размещения в нём ваших акторов

Здесь, мы начнём с нуля и создадим базовый уровень, в который мы сможем помещать наших игровых персонажей. Команда UE4 уже проделала великолепную работу по представлению того, как редактор мира может использоваться для создания мира в UE4. Я хочу, чтобы вы потратили немного времени и создали ваш собственный мир.

Во первых, чтобы начать создайте новый, пустой проект UE4. Для этого в лаунчере Unreal нажмите кнопку **Launch** (Запустить), которая рядом с вашими самыми недавними установками, как показано на следующем скриншоте:



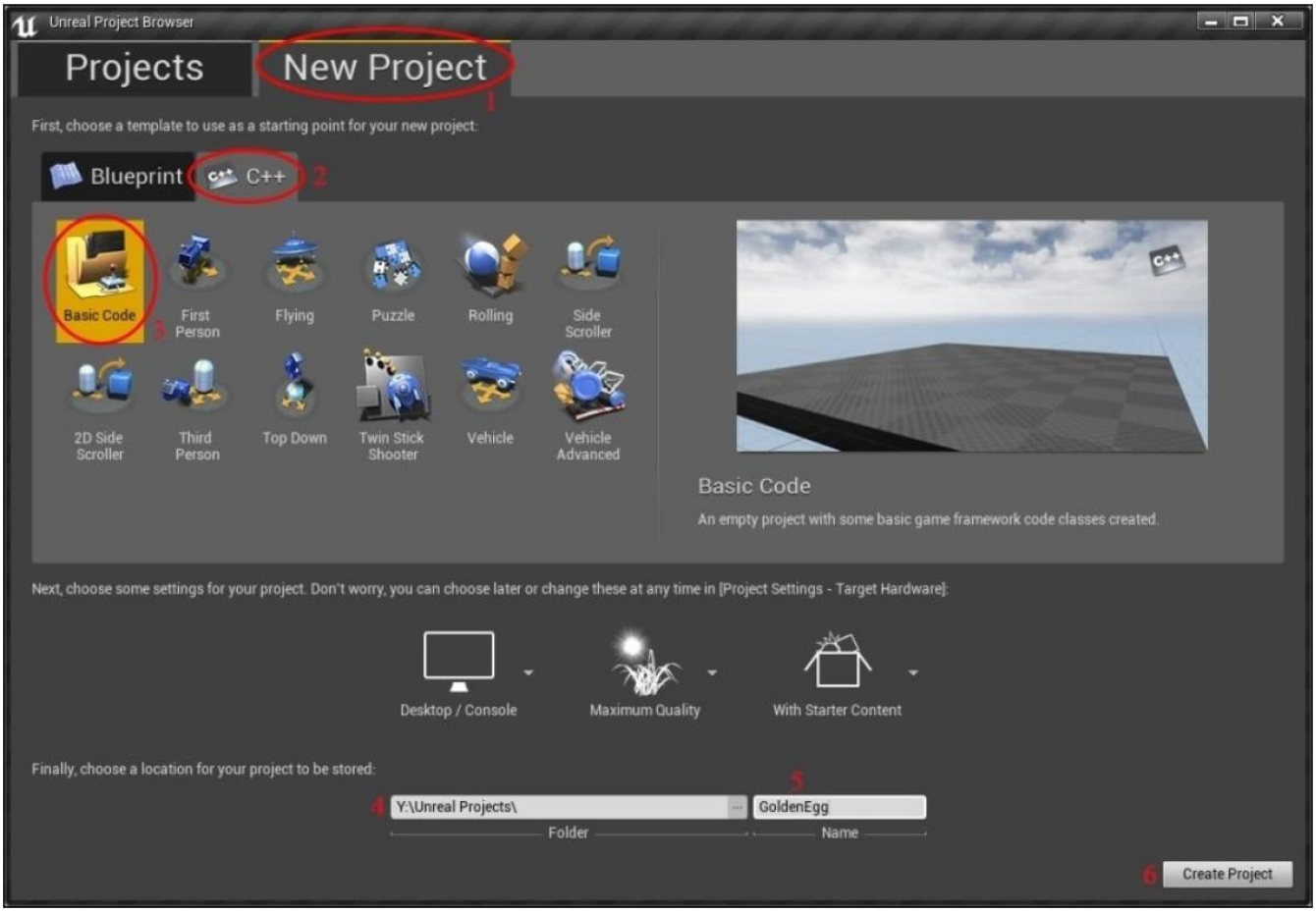
Это запустит редактор (Unreal Editor). Редактор Unreal используется, чтобы визуально редактировать ваш игровой мир. Вы проведёте много времени в редакторе Unreal, так что пожалуйста используйте своё время, чтобы поэкспериментировать и всё попробовать.

Я лишь охвачу основы того, как работать с редактором UE4. Вам понадобится высвободить свою фантазию и пожертвовать немного времени, чтобы познакомиться с редактором как следует.

Совет

Чтобы узнать больше о редакторе UE4, посмотрите плэйлист *Getting Started: Introduction to the UE4 Editor*, который доступен на: https://www.youtube.com/playlist?list=PLZlv_N0_O1gasd4IcOe9Cx9wHoBB7rxFl.

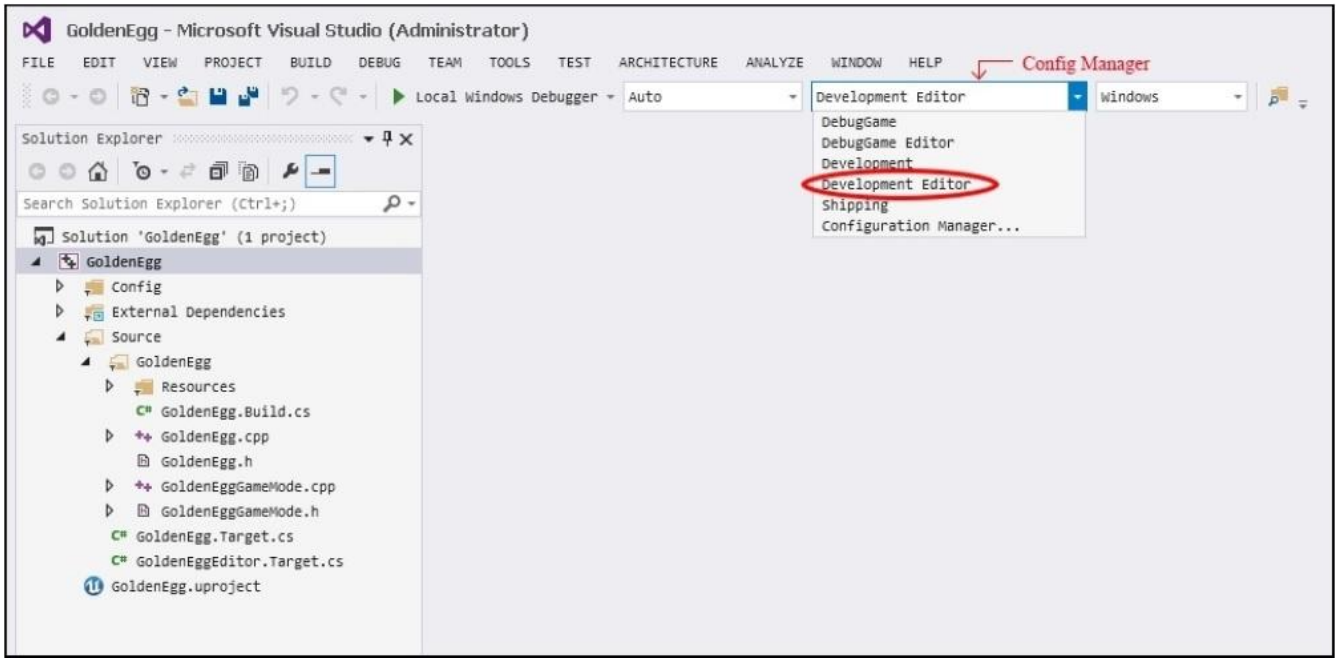
Как только вы запустили редактор UE4, вам будет представлен диалоговое окно **Project**. Следующий скриншот демонстрирует пронумерованные по порядку шаги:



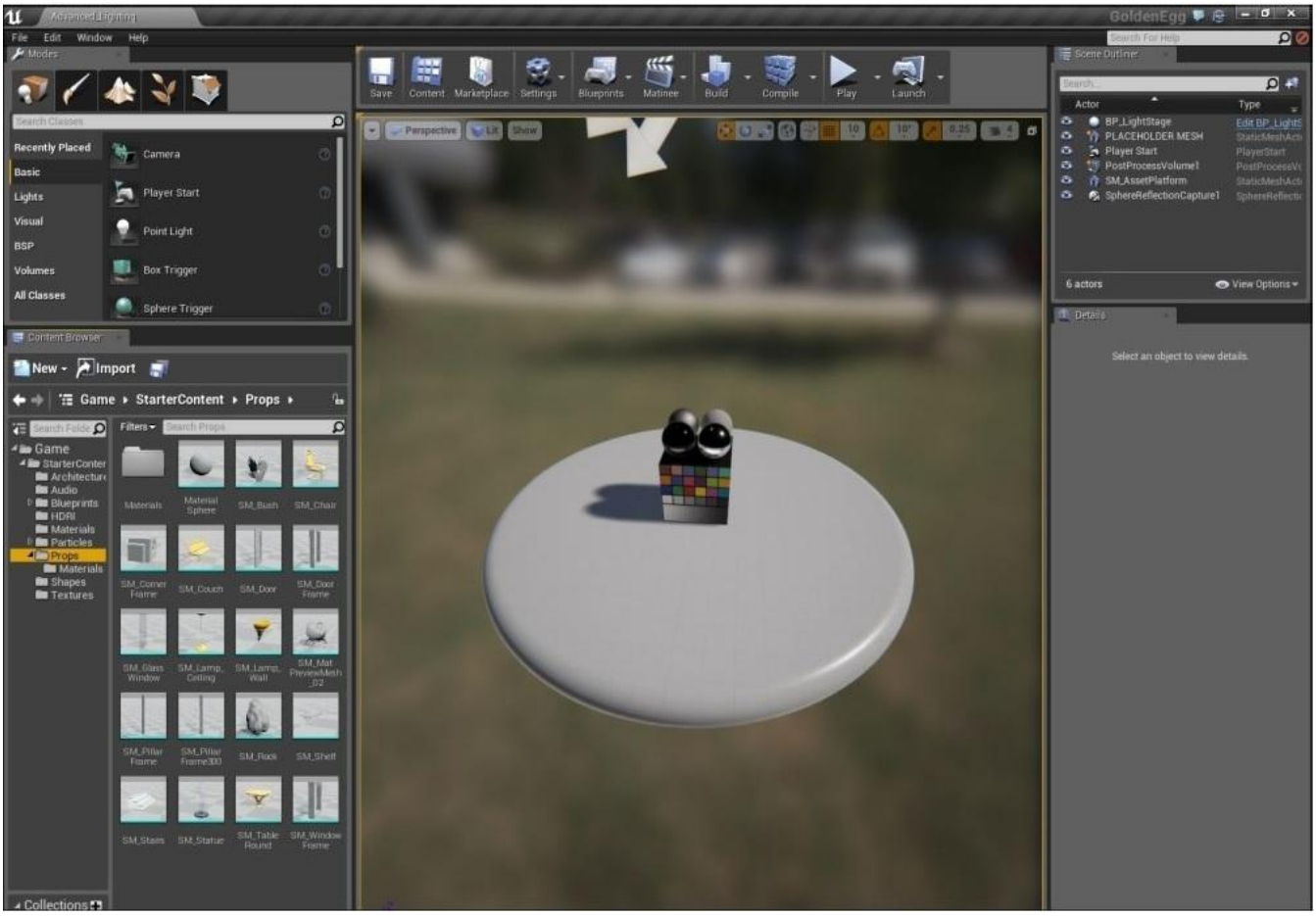
Выполните следующие шаги, чтобы создать проект:

1. Выберите вкладку **New Project** вверху экрана.
2. Щёлкните по вкладке **C++** (вторая подвкладка).
3. Затем выберите **Basic Code** из списка доступных проектов.
4. Выберите место хранения проекта (у меня **Y:\Unreal Project**\). Выберите расположение на жёстком диске, где много места (итоговый проект будет около 1.5 ГБ).
5. Назовите ваш проект. Свой я назвал **GoldenEgg** (ЗолотоеЯйцо).
6. Нажмите **Create Project**, чтобы закончить создание проекта.

Как только вы выполните это, лаунчер UE4 запустит Visual Studio. Будет только два исходных файла в Visual Studio, но мы пока не собираемся их трогать. Убедитесь что **Development Editor** (редактор разработки) выбран в выпадающем меню **Configuration Manager** (менеджер конфигурации), вверху экрана, как показано на следующем скриншоте:



Теперь запустите ваш проект, нажав *Ctrl+F5* в Visual Studio. Вы кажитесь в редакторе UE4, как показано на следующем скриншоте:



Редактор UE4

Здесь мы исследуем редактор UE4. Мы начнём с управления, поскольку важно знать, как управляться в Unreal.

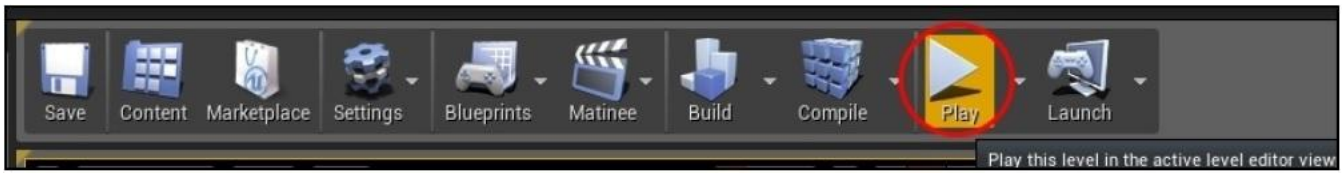
Управление редактором

Если вы никогда до этого не пользовались 3D редактором, то изучение управления может быть довольно сложным. Вот основное навигационное управление в режиме редактора:

- Используйте клавиши со стрелками, чтобы перемещаться по сцене
- Нажимайте клавиши *Page Up* и *Page Down*, чтобы перемещаться вертикально
- Зажимайте левую кнопку мыши + тащите влево или вправо, чтобы изменить направление, в которое вы обращены
- Зажимайте левую кнопку мыши + тащите вверх или вниз, чтобы двигать камеру вперёд и назад (тоже самое, что и нажимать клавиши со стрелкой вверх или вниз)
- Зажимайте правую кнопку мыши + тащите, чтобы изменить направление, в которое вы обращены
- Зажимайте колёсико мыши + тащите, чтобы панорамировать вид
- Зажатие правой кнопки мыши и клавиши W, A, S, D используются для перемещения по сцене

Управление режимом игры

Нажмите кнопку **Play** на панели сверху, как показано на следующем скриншоте. Это запустит режим игры.



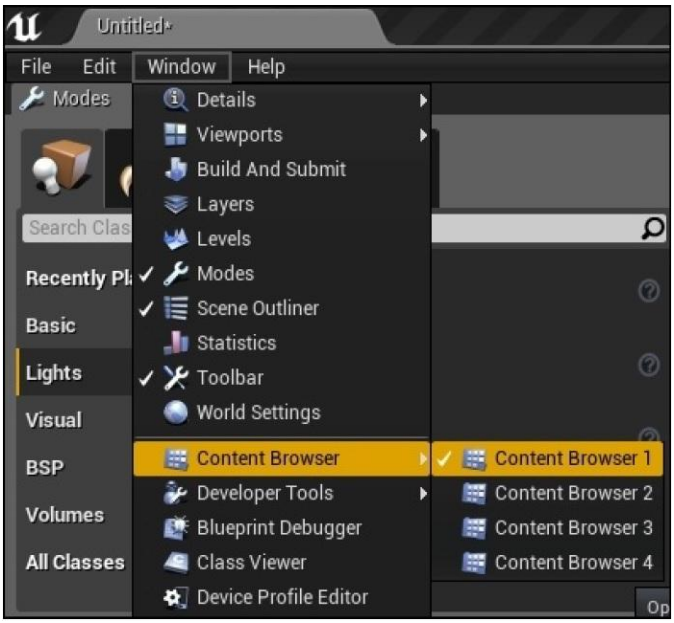
Как только вы нажмёте кнопку **Play**, управление переменится. В игровом режиме управление следующее:

- Клавиши W, A, S, D для перемещения
- Клавиши со стрелками влево и вправо, чтобы смотреть влево и вправо соответственно
- Движение мыши для изменения направления, в котором вы смотрите
- Клавиша *Esc* для выхода из режима игры и возвращения к режиму редактора

Сейчас я предлагаю вам, попробовать добавить несколько фигур и объектов в сцену и попробовать окрасить их различными *материалами*.

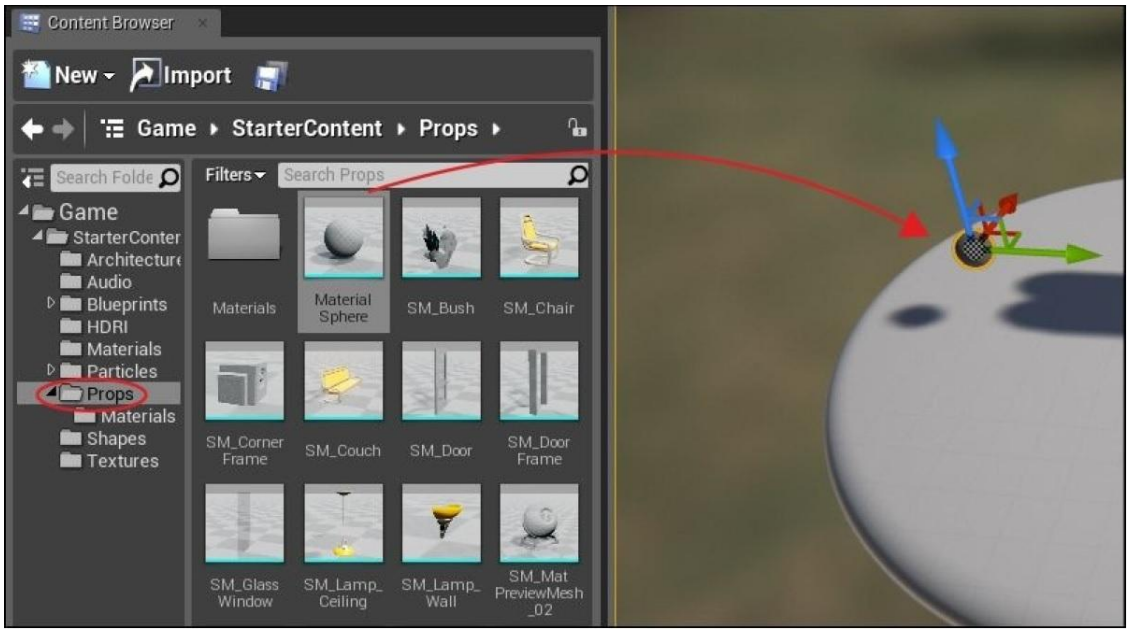
Добавление объектов в сцену

Добавить объекты в сцену также просто как перетащить их из вкладки **Content Browser**. Вкладка **Content Browser** по умолчанию находится с левой стороны окна. Если её там нет, просто выберите **Window** и перейдите к **Content Browser**, чтобы эта вкладка появилась.



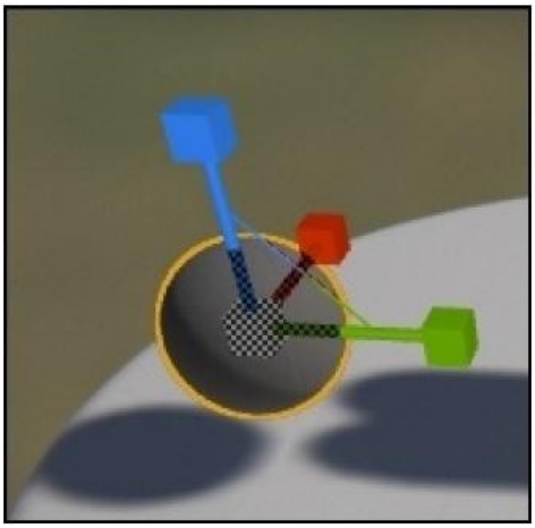
Убедитесь, что Content Browser видим, перед тем, как добавлять объекты в ваш уровень

Далее выберите папку **Props** с левой стороны **Content Browser**.



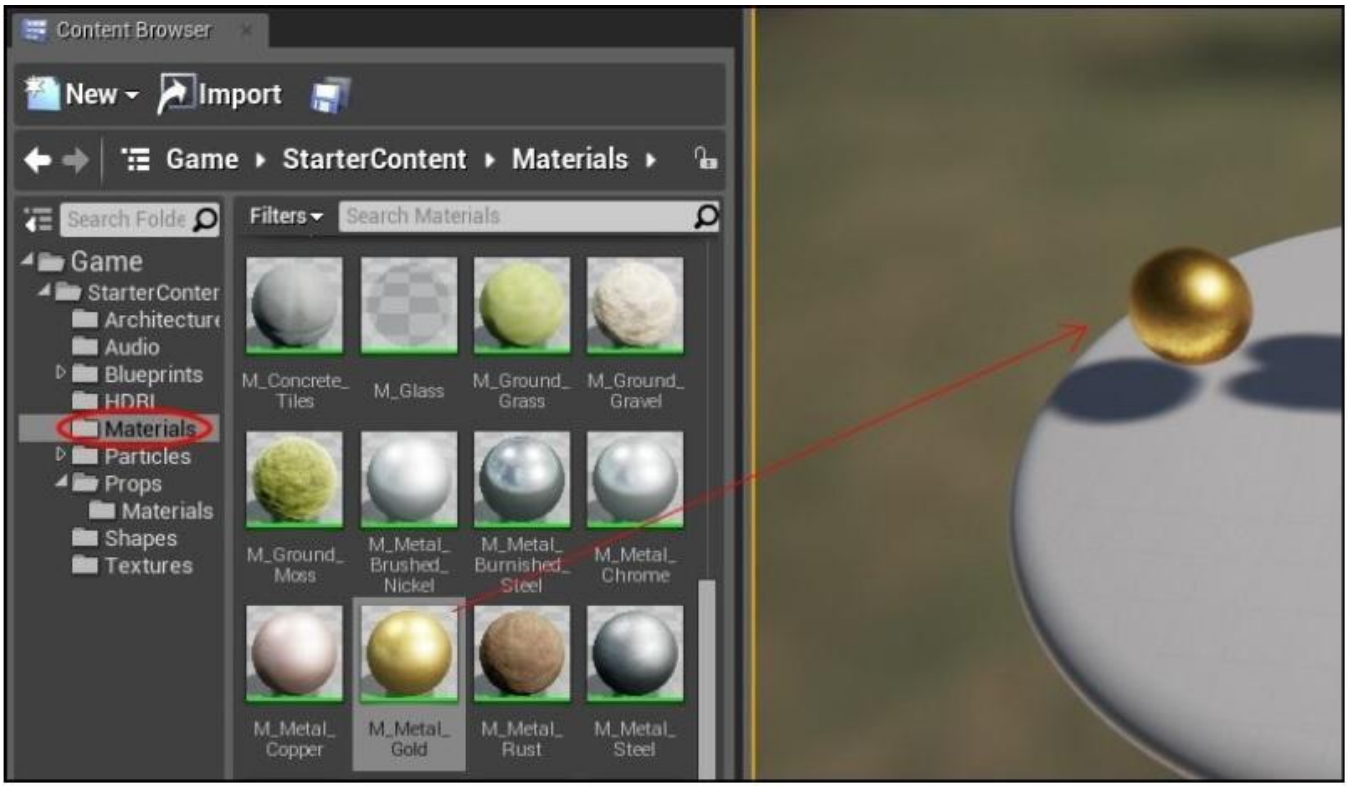
Перетаскивайте предметы из Content Browser в ваш игровой мир

Чтобы изменить размер объекта, нажмите клавишу R на клавиатуре. Манипуляторы режима изменения размера, появятся вокруг объекта в виде кубиков.



Нажмите R на клавиатуре для изменения размера объекта

Перед тем как, менять материал, который используется для окраски объектов, просто перетащите новый материал из окна **Content Browser** в папку **Materials**.



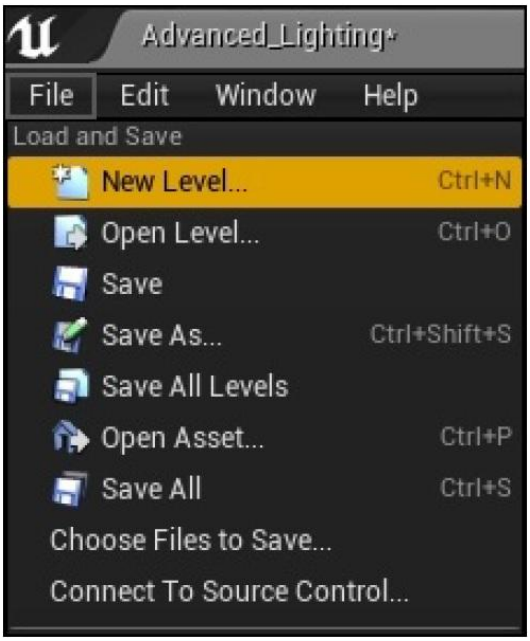
Перетащите материал с окна Content Browser из папки Materials на объект, чтобы окрасить его в новый цвет

Материалы, они как краски. Вы можете покрыть объект любым материалом, каким хотите, просто перетаскивая нужный материал на объект, который вы желаете

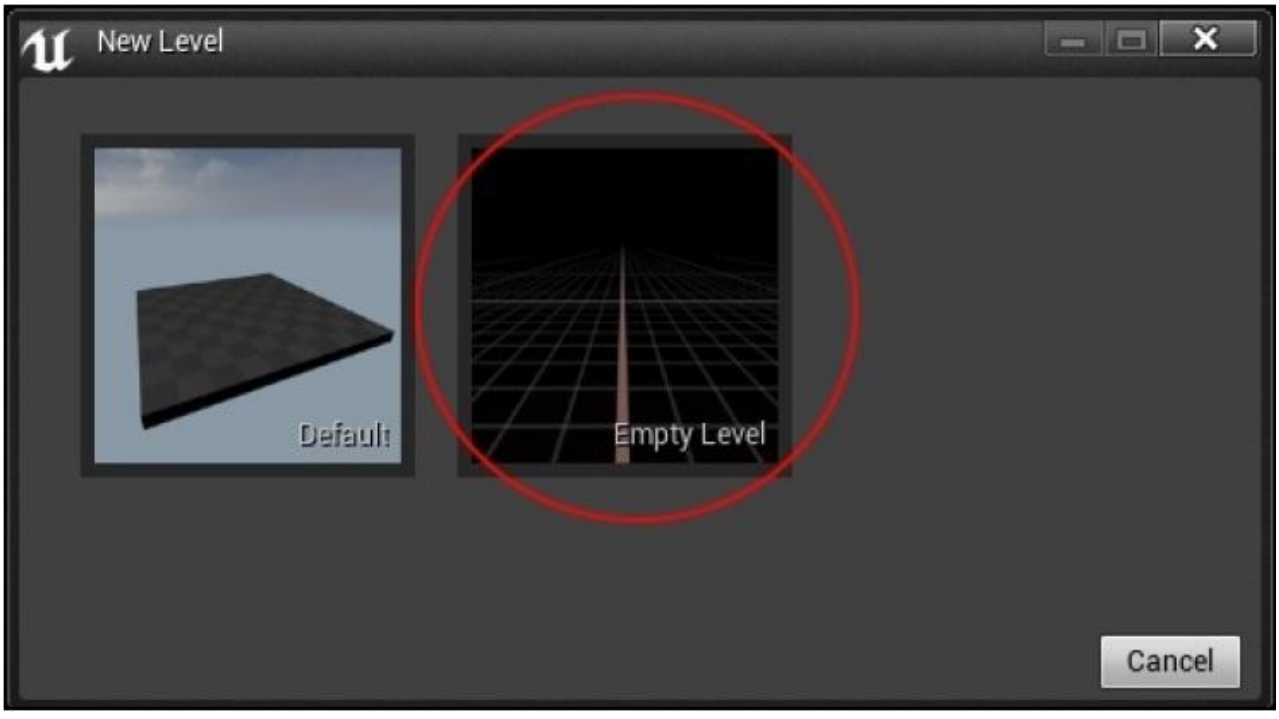
покрыть. Материалы поверхностны, они не меняют другие свойства объекта (такие как вес).

Начинаем с нуля

Если вы хотите создать уровень с нуля, просто нажмите **File** и перейдите к **New Level…**, как показано здесь:



Затем вы можете выбрать между **Default** (по умолчанию) и **Empty Level** (пустой уровень). Я думаю выбрать **Empty Level** это хорошая идея, по причинам, которые будут названы позже.

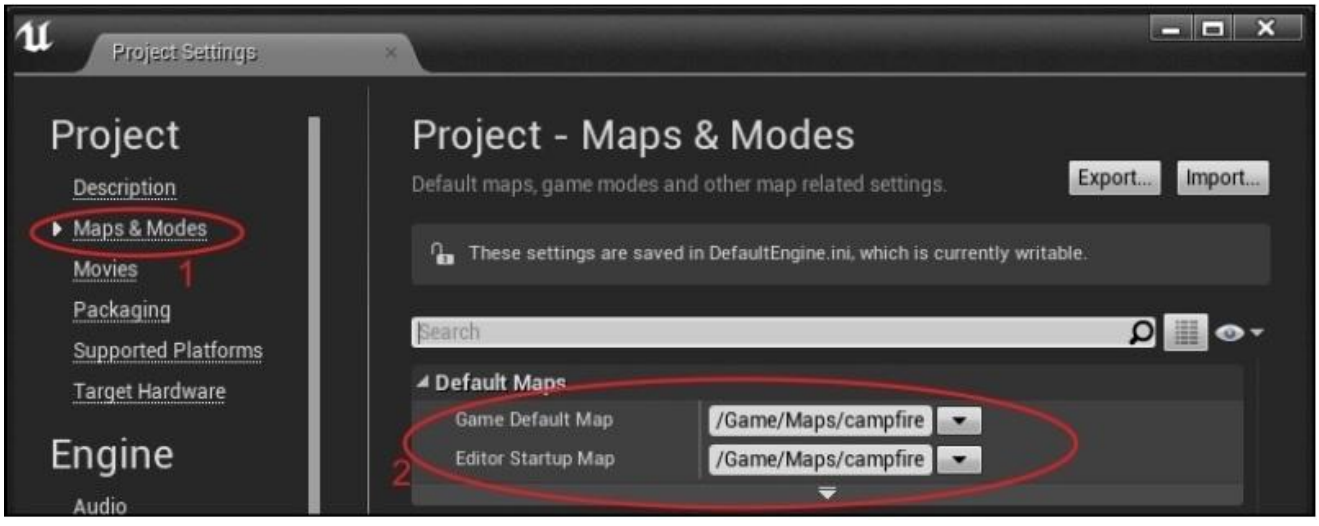


Новый уровень будет совершенно чёрным. Попробуйте снова перетащить объекты из вкладки **Content Browser**.

На этот раз, я добавил формы/блоки с изменённым размером для поверхности земли, текстурированной мхом, а также **Props** / **SM_Rocks**, **Particles** / **P_Fire**, и что наиболее важно источник света.

Обязательно сохраните вашу карту. Вот снимок моей карты (а как выглядит ваша?):

Если вы хотите изменить уровень по умолчанию, который открывается, когда вы запускаете редактор, то перейдите к **Project Settings** | **Map & Models**, затем вы увидите настройки **Game Default Map** и **Editor Startup Map**, как показано на следующем скриншоте:



Добавление источников света

Заметьте, что если ваша сцена предстаёт абсолютно чёрной, то возможно вы забыли поместить в неё источник света.

В предыдущей сцене, точка излучения **P_Fire** действует как источник света, но она испускает лишь небольшой объём света. Чтобы всё было хорошо освещено в вашей сцене, вам нужно добавить источник света следующим образом:

1. Перейдите к **Window** и щёлкните по **Modes**, чтобы панель источников света была показана:

2. Затем из панели **Modes**, перетащите один из объектов Lights на сцену:

3. Выберите значок с лампочкой и кубом.
4. Щёлкните по **Lights**, на панели слева.
5. Выберите тип света, который хотите и просто перетащите его в свою сцену.

Если у вас не будет источника света, ваша сцена буде абсолютно чёрной.

Объёмы столкновения

Возможно вы давно уже заметили, что камера просто проходит по всей геометрии сцены, даже в режиме игры. Это не хорошо. Давайте сделаем так, чтобы игрок не мог гулять сквозь скалы в нашей сцене.

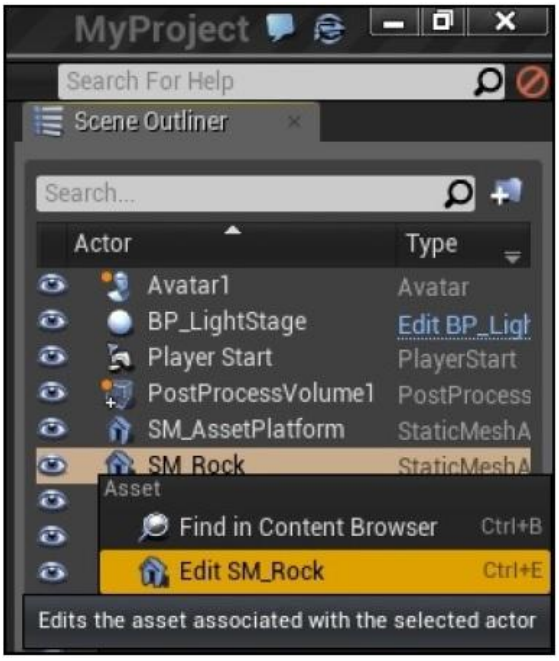
Есть несколько разных типов столкновения. В основном, идеальные столкновения mesh-mesh (сетка-сетка) слишком затратный способ, чтобы выполнять его в ходе игры. Вместо этого мы используем приблизительность (граничащие объёмы), чтобы угадывать объём столкновения.

Добавление обнаружения столкновения для редактора объектов

Первое, что нам нужно сделать, это ассоциировать объём столкновения с каждой скалой в сцене.

Мы можем сделать это из редактора UE4 следующим образом:

1. Щёлкните по каждому объекту в сцене, которому вы хотите добавить объём столкновения.
2. Щелчок правой кнопкой мыши по этому объекту во вкладке **Scene Outliner** (по умолчанию расположенной с правой стороны экрана) и выберите редактировать, как показано на следующем скриншоте:

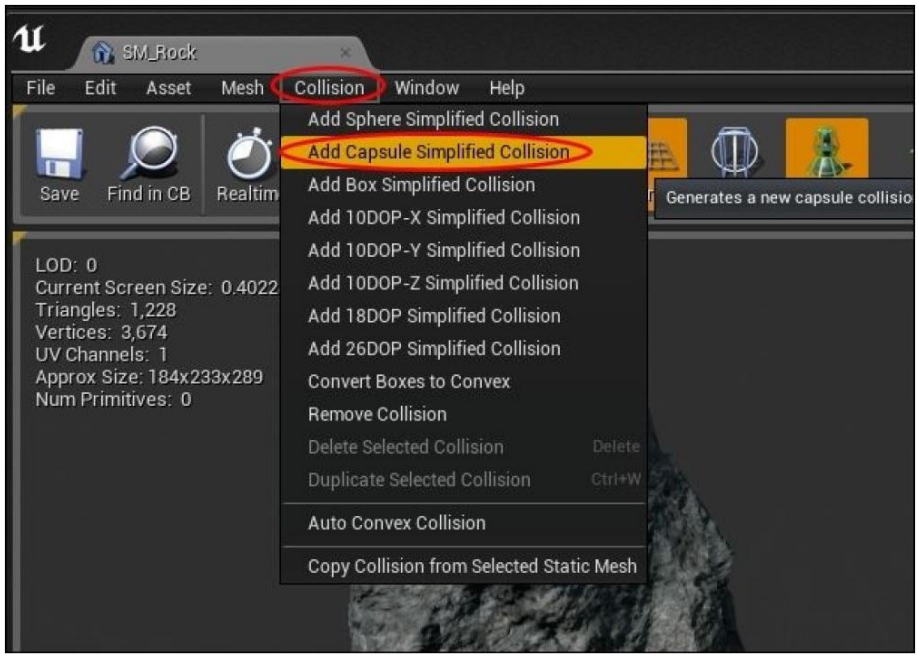


Примечание

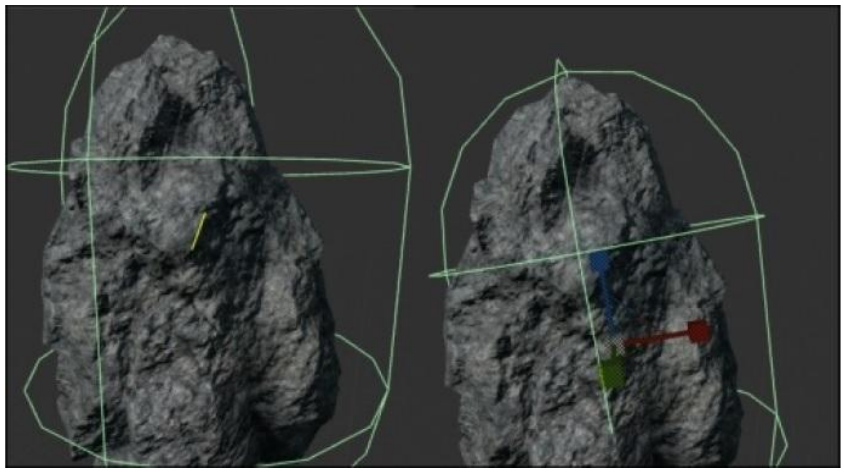
Вы окажитесь в редакторе сетки.

3. Убедитесь, что объём столкновения выделен, в верху экрана:

4. Перейдите в меню **Collision** и щёлкните по **Add Capsule Simplified Collision**:



5. Затем объём столкновения, если добавлен успешно, появится в виде линий окружающих объект, как показано на следующем изображении:



Капсула столкновения по умолчанию (слева) и версия с изменённым вручную размером (справа)

6. Вы можете изменять размер (*R*), вращать (*E*), двигать (*W*) и изменять объём столкновения, как пожелаете. Таким же образом вы можете манипулировать объектом в редакторе UE4.
7. Когда вы закончите с добавлением сетки столкновения, попробуйте нажать **Play**. Вы увидите, что вы больше не можете проходить через объекты, для которых установлено столкновение.

Добавление актора в сцену

Сейчас, когда мы подготовили и запустили сцену, нам надо добавить в неё актор. Давайте сначала добавим аватар для игрока, с готовым объёмом столкновения. Чтобы сделать это, нам надо выполнить наследование от класса GameFramework в UE4.

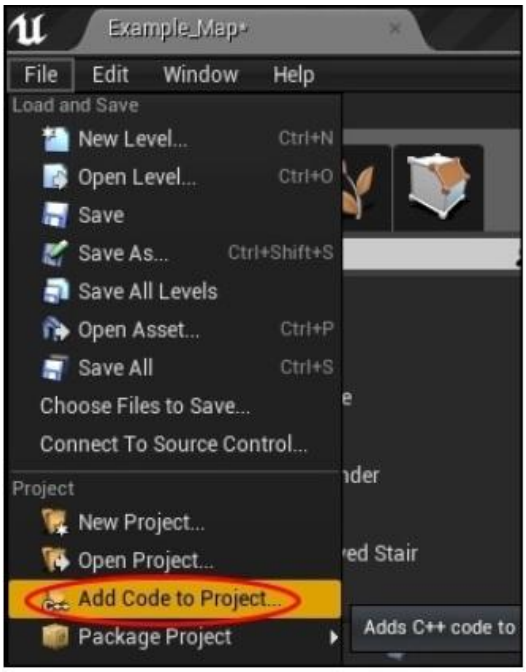
Создание сущности игрока

Перед тем как создать представление игрока на экране, нам надо выполнить происхождение от класса Character в Unreal.

Наследование от класса GameFramework в UE4

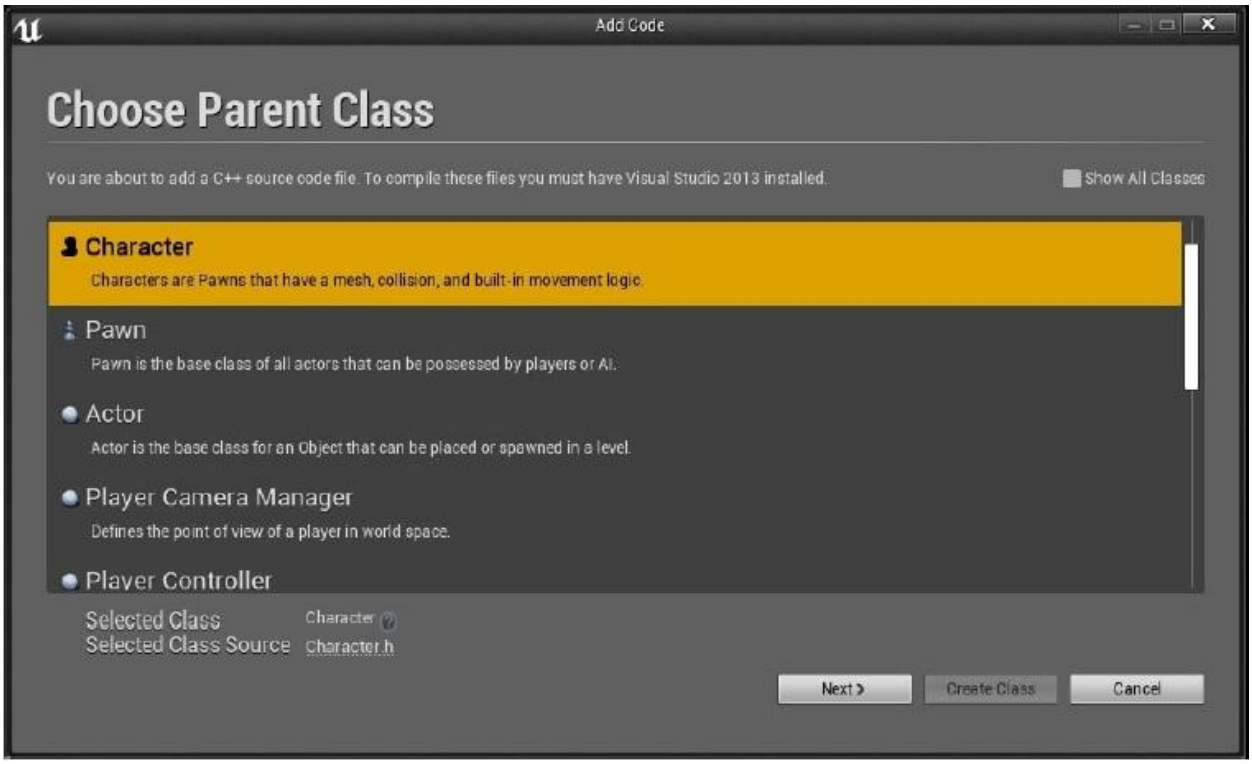
UE4 делает лёгким наследование от базового класса фреймворка. Всё что вам нужно сделать, выполнить следующие шаги:

1. Откройте ваш проект в редакторе UE4.
2. Перейдите в **File** и затем выберите **Add Code to Project** (добавить код в проект)…



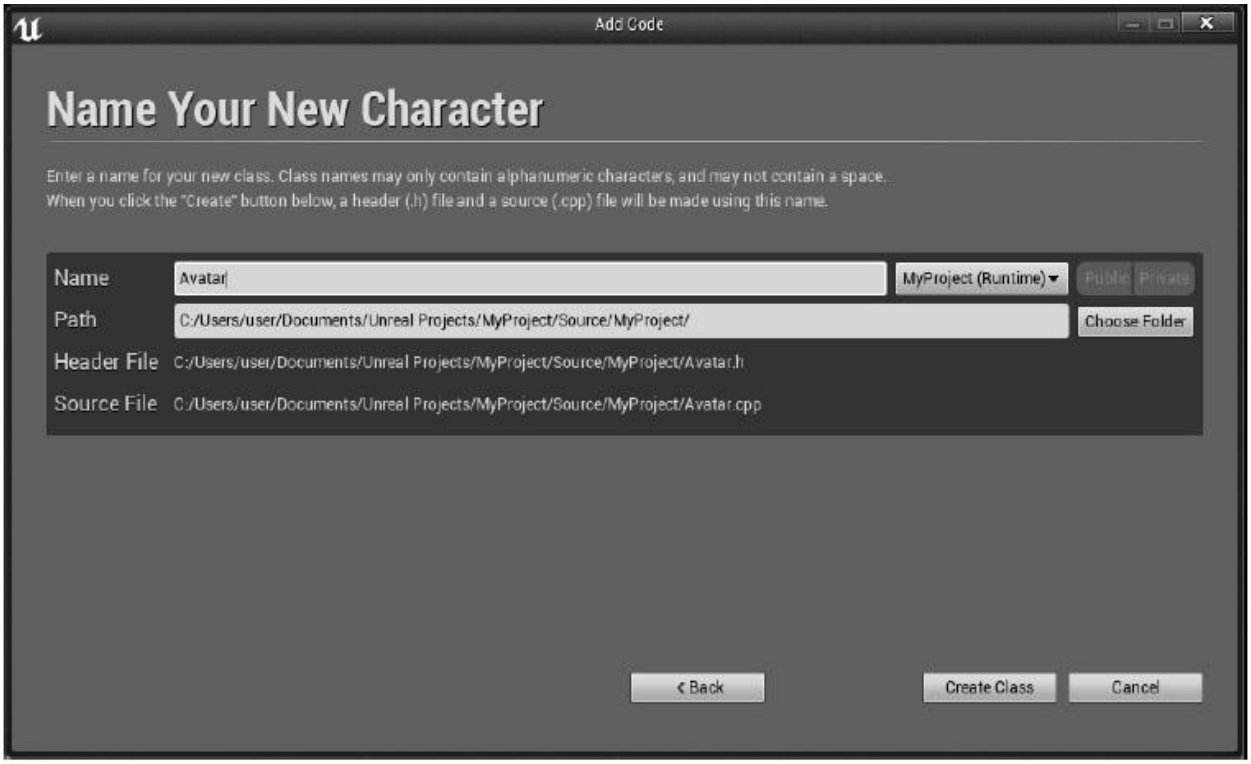
File | Add Code to Project… позволит вам выполнить происхождение от любого из классов GameFramework в UE4

3. Теперь, выберите базовый класс, от которого вы хотите произойти. У вас есть **Character**, **Pawn**, **Actor** и так далее, но сейчас мы выполним происхождение от **Character**:



Выберите класс UE4 от которого вы хотите произойти

4. Нажмите **Next >**, чтобы получить это диалоговое окно, в котором вы даёте имя классу. Свой класс игрока я назвал Avatar.



5. И наконец то, нажмите **Create Class**, чтобы создать класс в коде, как показано на предыдущем скриншоте.

Позвольте UE4 обновить ваш проект Visual Studio, когда он попросит вас. Откройте новый файл Avatar.h из **Solution Explorer**.

Код, который генерирует UE4, будет выглядеть немного странно. Вспомните макросы, которые я предлагал вам избегать в Главе 5. *Функции и макросы*. UE4 широко использует макросы. Макросы используются, чтобы копировать и вставлять начальный шаблон кода, что позволяет вашему коду интегрироваться с редактором UE4.

Содержание файла Avatar.h показано в следующем коде:

```
#pragma once
// код файла Avatar.h
#include "GameFramework/Character.h"
#include "Avatar.generated.h"
UCLASS()
class MYPROJECT_API AAvatar : public ACharacter
{
   GENERATED_UCLASS_BODY()
};
```

Давайте немного поговорим о макросах.

Макрос UCLASS() в основном делает ваш класс кода C++ доступным в редакторе UE4. Макрос GENERATED_UCLASS_BODY() копирует и вставляет код, который нужен UE4, чтобы заставить ваш класс функционировать, как полагается классу UE4.

Подсказка

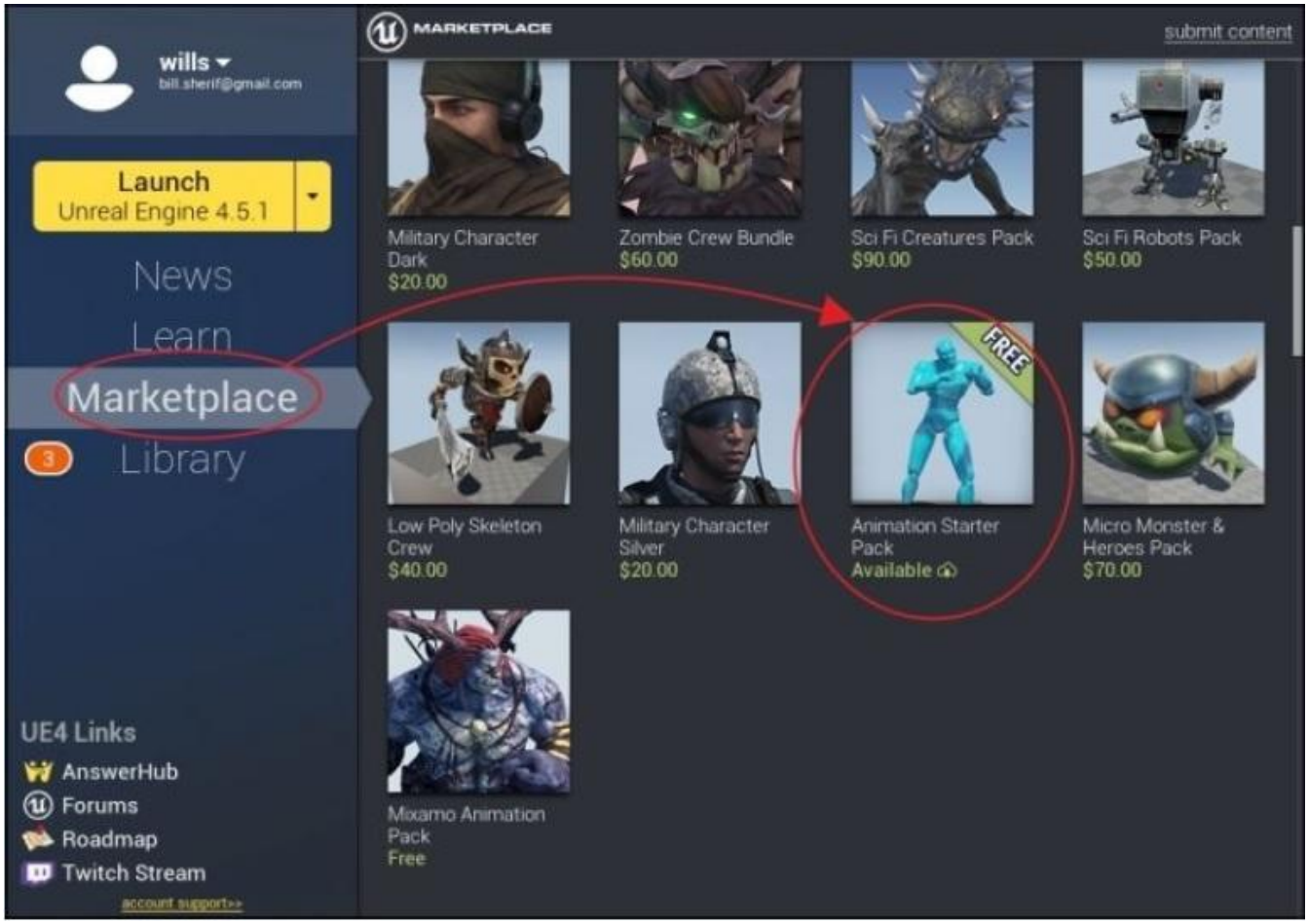
Для UCLASS() и GENERATED_UCLASS_BODY() вам на самом деле не обязательно понимать, как UE4 проделывает свою магию. Вам лишь надо удостовериться, что они присутствуют в нужном месте (там, где они были, когда вы сгенерировали класс).

Ассоциирование модели с классом Avatar

Сейчас нам надо ассоциировать модель с объектом нашего персонажа. Перед этим, нам нужна модель для игры. К счастью есть полный набор образцов моделей доступных бесплатно в магазине UE4.

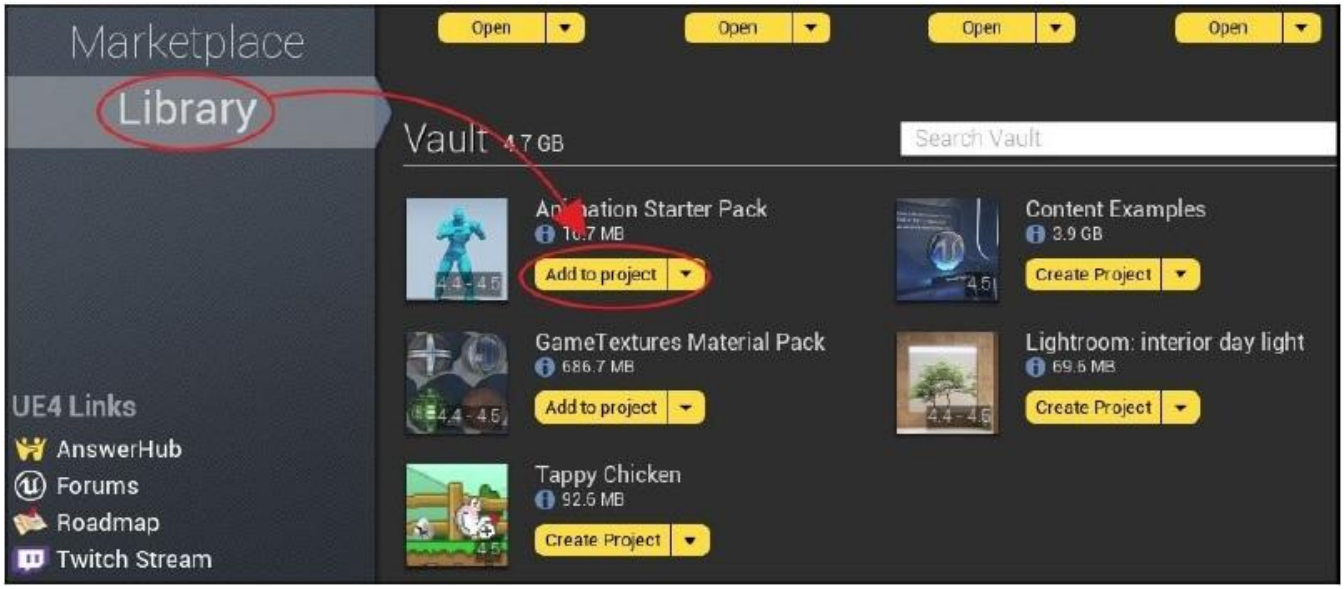
Скачивание бесплатных моделей

Чтобы создать объект игрока, мы скачаем файл **Animation Starter Pack** (который бесплатен) из вкладки **Marketplace**.

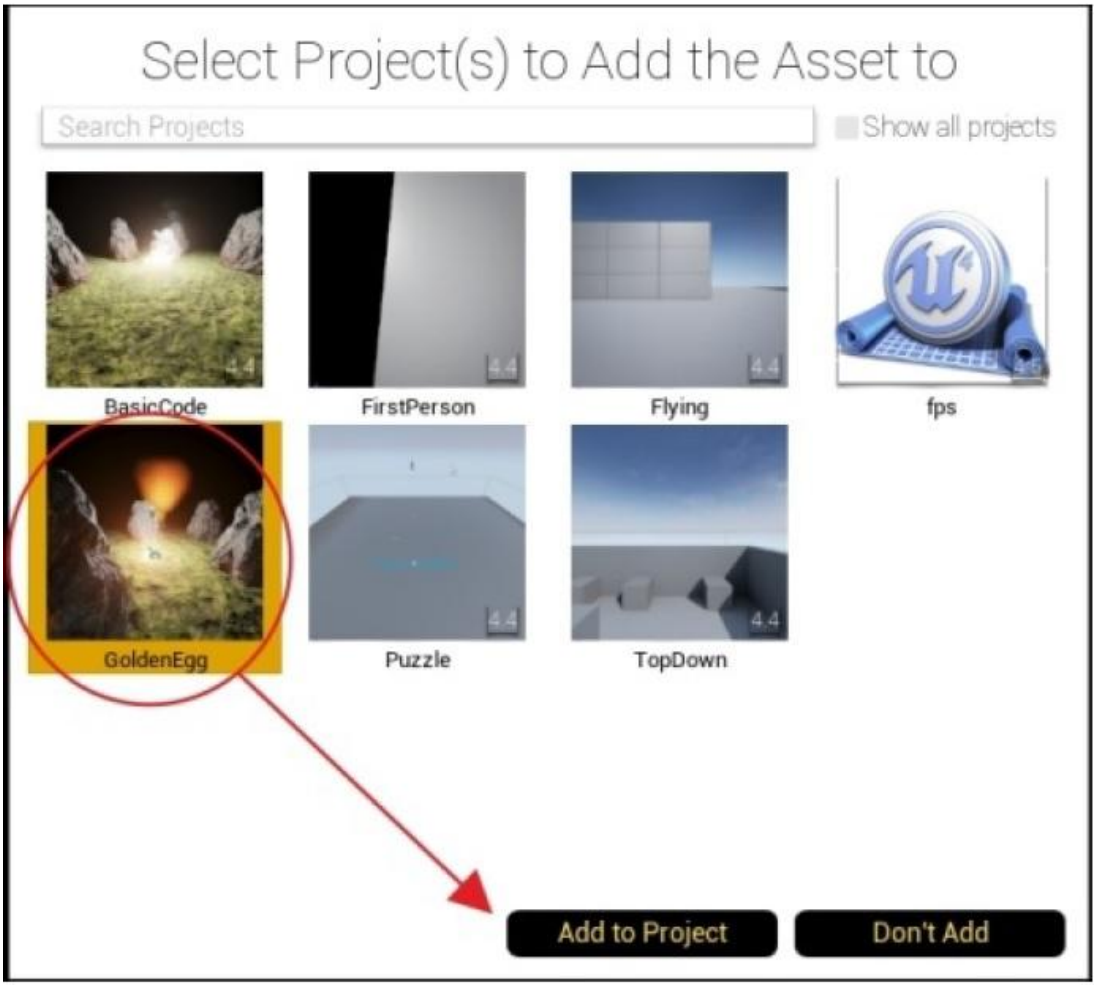


В Unreal Launcher щёлкните по Marketplace найдите Animation Starter Pack, который был бесплатным на момент написания этой книги

После того, как вы скачаете файл **Animation Starter Pack**, вы сможете добавить его в любой ранее созданный вами проект, как показано на следующем скриншоте:



Когда вы нажмёте **Add to project** под **Animation Starter Pack**, у вас появится это всплывающее окно, спрашивающее, в какой проект добавить пакет:



Просто выберите ваш проект, и новая фигура станет доступна в вашем **Content Browser**.

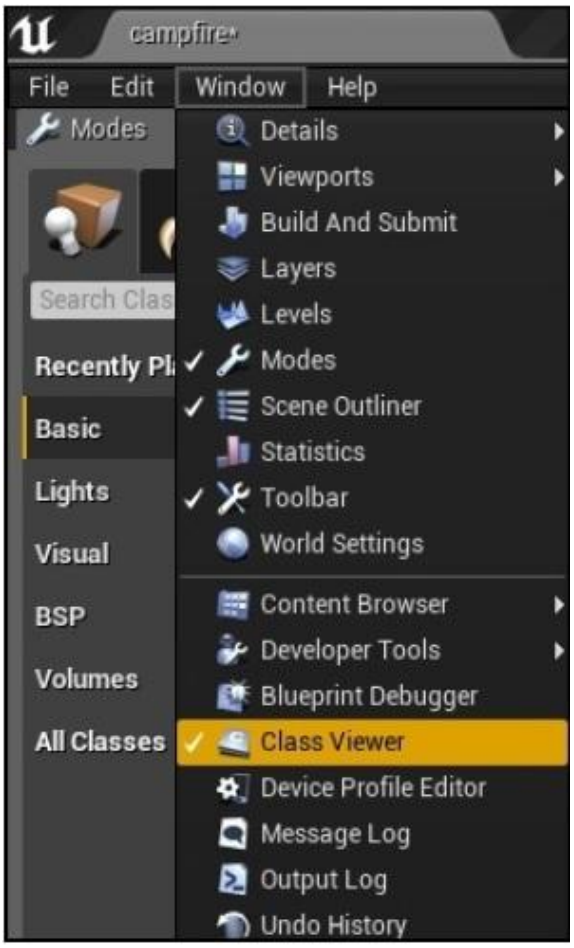
Загружаем сетку

В целом считается плохой практикой жёстко закодированные в игре ассеты (ресурс представляющий часть игрового контента). Жёстко кодировать или хардкодить означает, что вы пишете C++ код, из-за которого ассеты грузятся. И хардкодинг означает, что загруженный ассет это часть итогового исполняемого файла, что в свою очередь будет означать, что изменение уже загруженного ассета будет невозможно в ходе игры. Это плохая практика. Гораздо лучше, если ассет можно изменять в ходе игры.

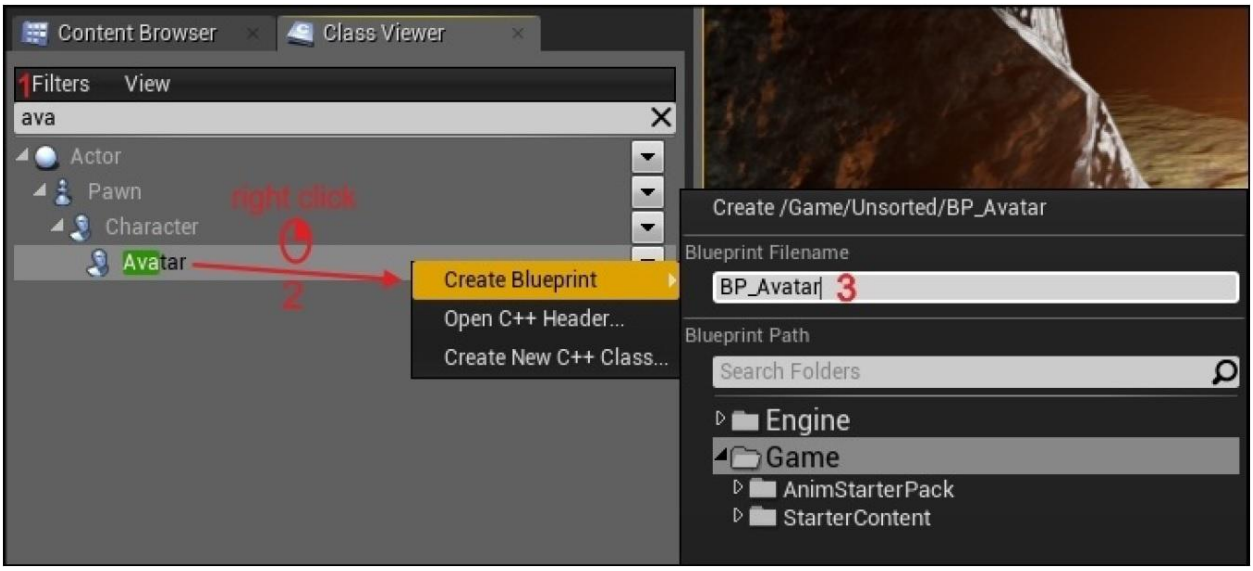
По этой причине, мы собираемся использовать схемы (blueprint) UE4, чтобы установить сетку модели и капсулу столкновения класса Avatar.

Создание blueprint от нашего C++ класса

1. Это действительно легко. Откройте вкладку **Class Viewer**, перейдя в **Window**, а затем щёлкнув по **Class Viewer**, как показано здесь:



2. В диалоговом окне **Class Viewer**, начните писать имя вашего C++ класса. Если вы правильно создали и экспортировали класс из вашего C++ кода, то он будет выглядеть как показано на следующем скриншоте:

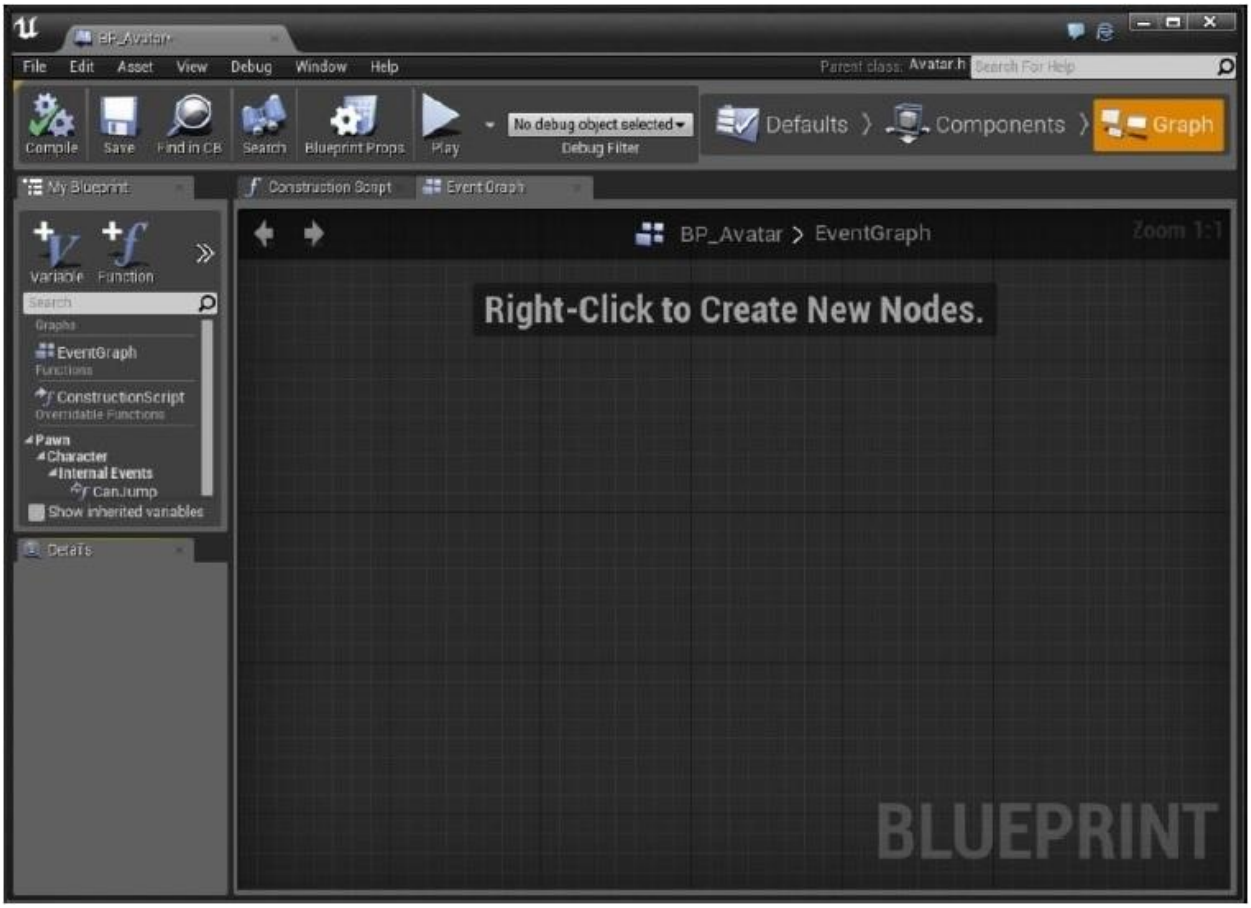


Подсказка

Если ваш класс Avatar не показался, то закройте редактор и снова компилируйте/запустите C++ проект.

3. Щёлкните правой кнопкой мыши по классу, из которого вы хотите создать blueprint (в моём случае это мой класс **Avatar**).
4. Дайте своему blueprint какое-нибудь особенное имя. Свой blueprint я назвал **BP_Avatar**.
5. Теперь откройте этот blueprint для редактирования, дважды щёлкнув по имени **BP_Avatar** (оно появится во вкладке **Class Viewer** после того, как вы добавите его, сразу под **Avatar**), как показано на следующем скриншоте:

6. Вам откроется окно blueprint для вашего нового объекта **BP_Avatar**, как показано здесь:

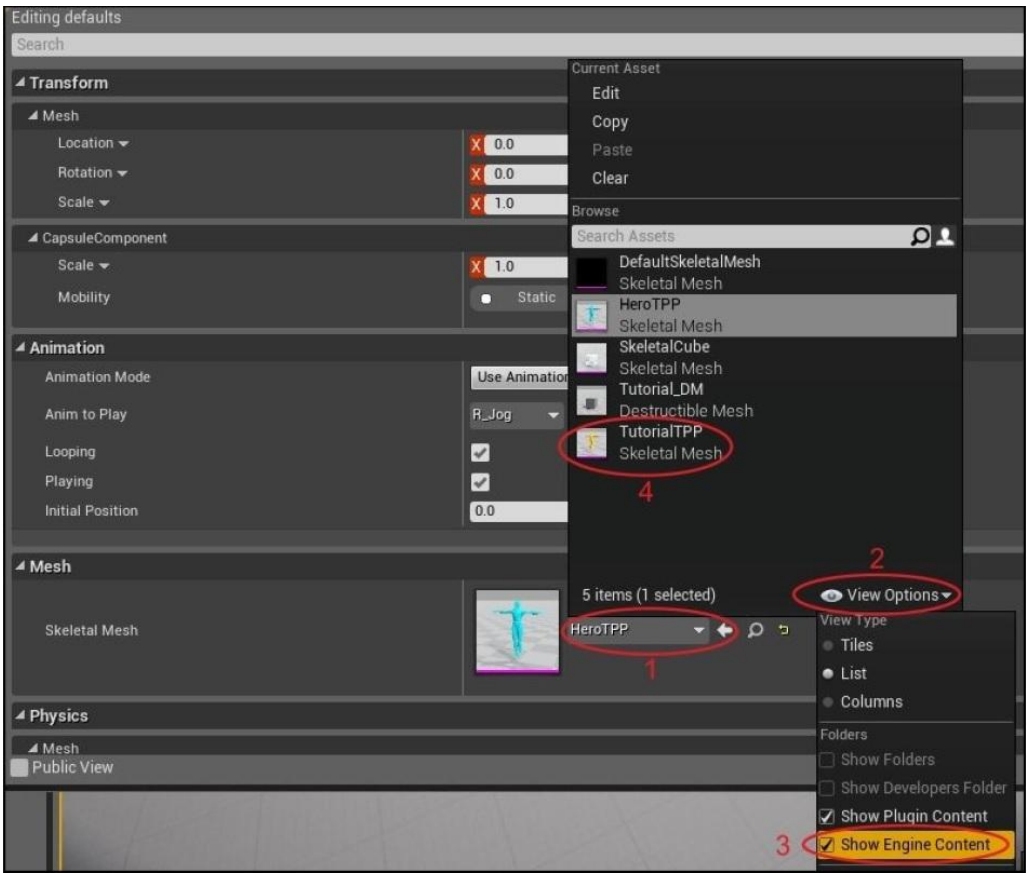


Подсказка

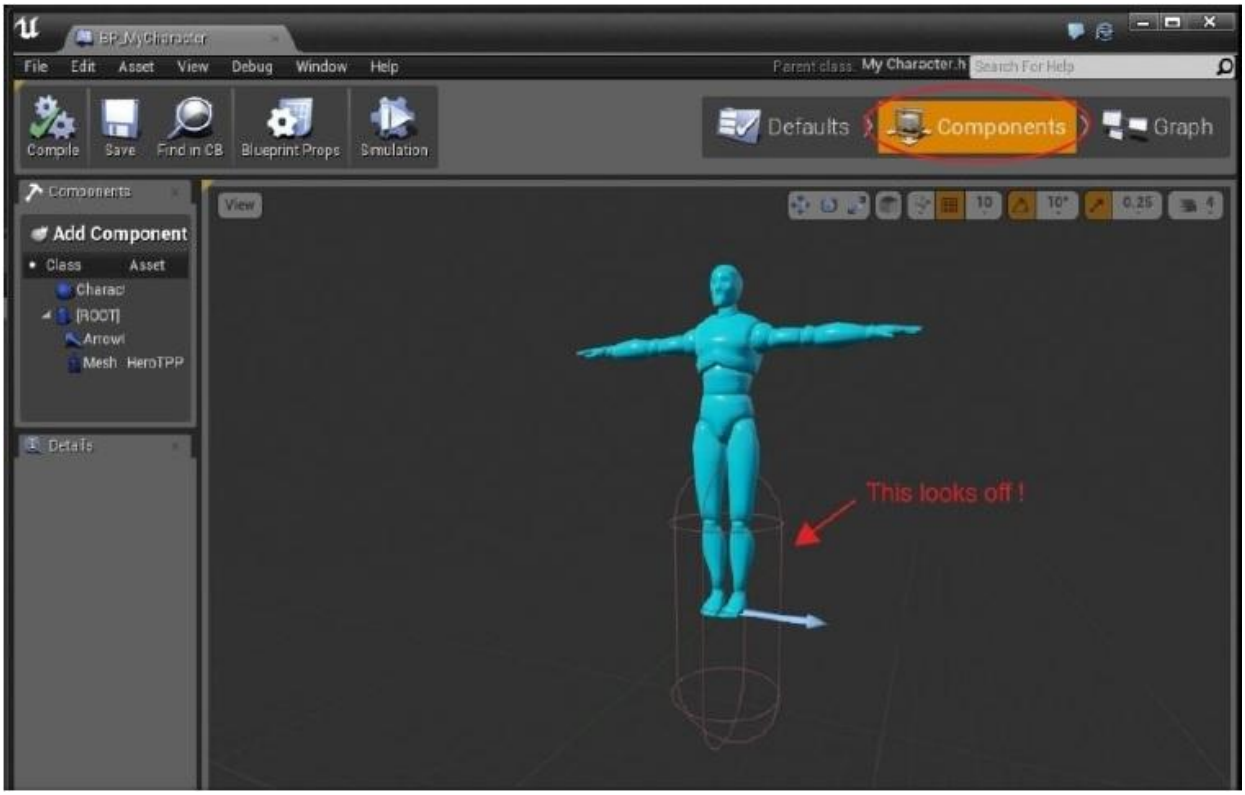
Из этого окна вы можете прикрепить модель к классу Avatar визуально. Ещё раз, это рекомендованная схема, так как художники обычно будут устанавливать свои ассеты для разработчиков игры, чтобы играть с ними.

7. Чтобы установить сетку по умолчанию, нажмите кнопку **Default** сверху. Прокрутите свойства вниз, до **Mesh**.

8. Щёлкните по выпадающему меню и выберите **HeroTPP** для вашей сетки, как показано на предыдущем скриншоте.
9. Если **HeroTPP** отсутствует в выпадающем меню, убедитесь, что вы скачали и добавили **Animation Starter Pack** в ваш проект. Альтернативно, вы можете добавить жёлтую модель **TutorialTPP** в ваш проект, если вы выбрали **Show Engine Content** (показать состав движка) под **View Options** (обзор опций):



10. Что на счёт объёма столкновения? Щёлкните по вкладке **Components** в редакторе blueprint для вашего аватара:

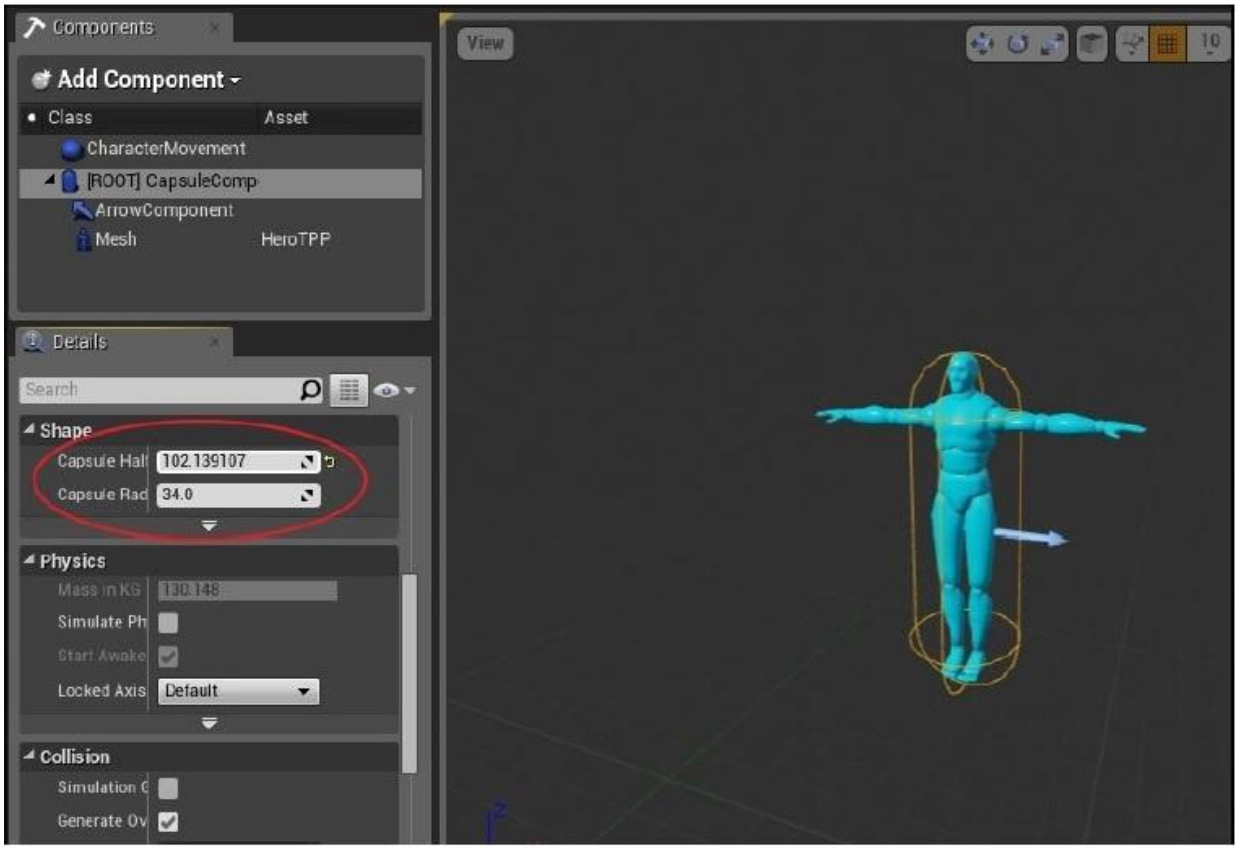


Если ваша капсула не охватывает вашу модель, отрегулируйте модель так, чтобы она была равномерно расположена в капсуле

Примечание

Если ваша модель окажется в таком же расположения как у меня, соответственно это значит что капсула не на месте! Вам нужно отрегулировать её.

11. Щёлкните по голубой модели аватара и нажмите клавишу W. Двигайте модель вниз, пока она не окажется в капсуле равномерно. Если капсула не достаточно большая, вы можете отрегулировать её размер во вкладке **Details** под **Capsule Height** и **Capsule Radius**, как показано на следующем скриншоте:



Вы можете растянуть вашу капсулу, регулируя свойство Capsule Height

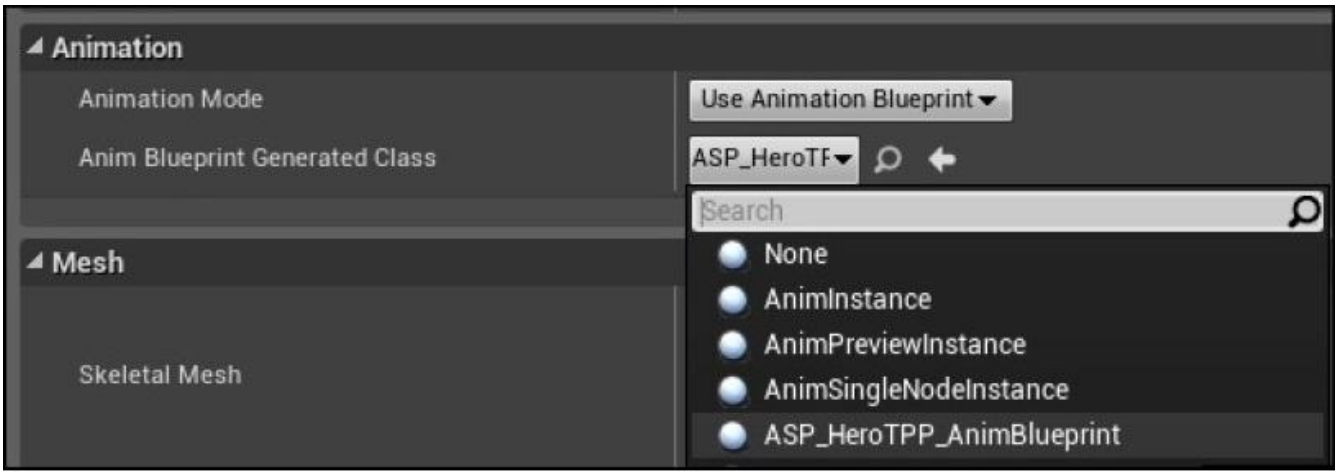
12. Теперь мы готовы добавить этот аватар в игровой мир. Щёлкните по вашей модели **BP_Avatar** и удерживая, перетащите из вкладки **Class Viewer** в вашу сцену в редакторе UE4.

Наш класс Avatar добавлен в сцену, в Т-позе

Поза аватара называется – Т-поза. Аниматоры часто оставляют своих персонажей в этой позе по умолчанию. К персонажу можно добавить анимации, которые затем сменят эту позу по умолчанию на что-либо более интересное. Вы говорите, что хотите заставить его двигаться! Что ж это легко.

В редакторе Blueprint, под вкладкой **Defaults**, прямо над **Mesh**, есть раздел **Animation**, где вы можете вы можете выбрать действующую анимацию к своей сетке (**Mesh**). Если вы желаете использовать определённый ассет анимации, просто щёлкните по выпадающему меню и выберите анимацию, которую вы желаете показать.

Однако лучше всего для анимации использовать схему (blueprint). Таким путём, художник может как следует установить анимацию, основанную на том, что делает персонаж. Если в **Animation Mode** (режим анимации) вы выбрали **Use Animation Blueprint** (использовать схему анимации), а затем выбрали **ASP_HeroTPP_AnimBlueprint** в выпадающем меню, то поведение персонажа в игре будет намного лучше, потому что анимация будет настроена посредством схемы (blueprint) (которая в свою очередь будет выполнена художником) по мере движения персонажа.



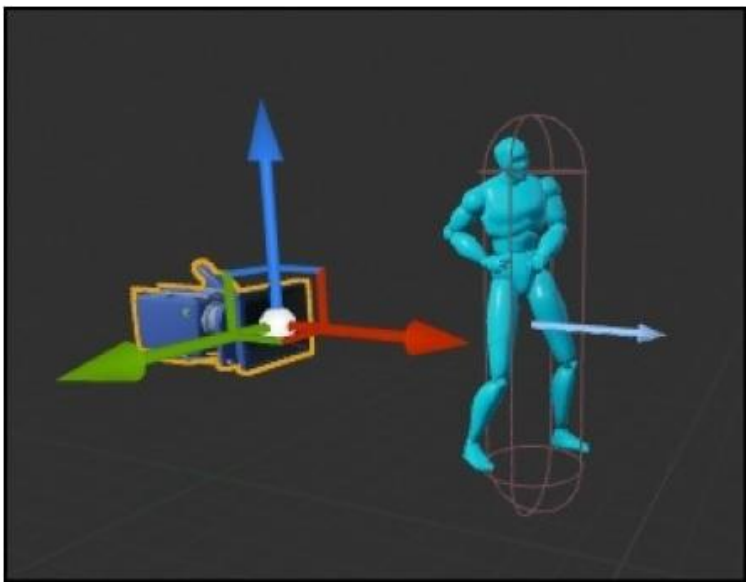
Совет

Мы не можем охватить здесь всё. Схемы анимации охватываются в Главе 11. *Монстры*. Если вы действительно заинтересованы анимацией, то также будет не плохой идеей уделить внимание паре учебных пособий от Gnomon Workshop по инверсной кинематике, анимации и риггингу. Таким как *Rigging 101* от Алекса Альвареза http://www.thegnomonworkshop.com/store/product/768/Rigging-101.

Ещё одна вещь: давайте сделаем так, чтобы камера находилась позади аватара. Это даст вам вид от третьего лица, что позволит вам видеть персонажа целиком, как показано на следующем скриншоте с шагами в соответствующем порядке выполнения:

1. В редакторе **BP_Avatar**, щёлкните по вкладке **Components**.
2. Щёлкните по **Add Component** (добавить компонент).
3. Для добавления выберите **Camera**.

Камера появится в окне просмотра. Вы можете щёлкнуть по камере и двигать её как угодно. Разместите камеру где-нибудь позади игрока. Голубая стрелка персонажа игрока должна обязательно указывать в том же направлении что и камера. Если это не так, поверните сетку модели Avatar так, чтобы она смотрела в том же направлении что и её голубая стрелка.



Голубая стрелка сетки вашей модели указывает направление вперёд для сетки модели. Убедитесь, что камера обращена в том же направлении что направляющий вектор персонажа

Написание C++ кода контролирующего игрового персонажа

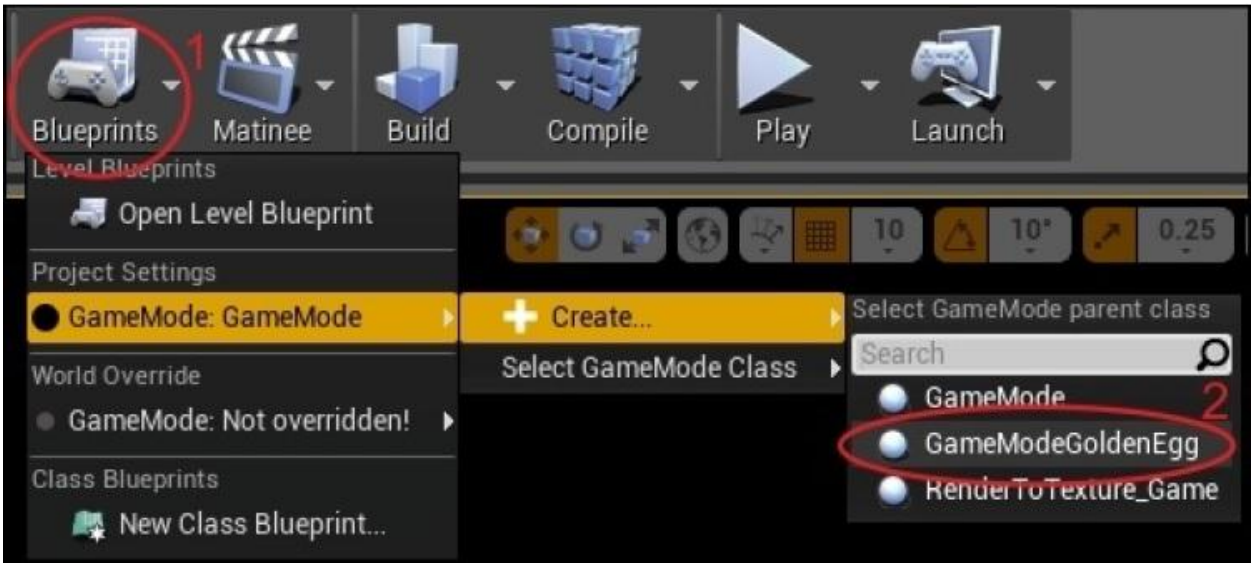
Когда вы запускаете вашу UE4 игру, вы должно быть замечаете, что камера в режиме по умолчанию, камера свободного полёта. Что мы сделаем сейчас? Сделаем стартовый персонаж экземпляром нашего класса Avatar и будем управлять нашим персонажем используя клавиатуру.

Делаем игрока экземпляром класса Avatar

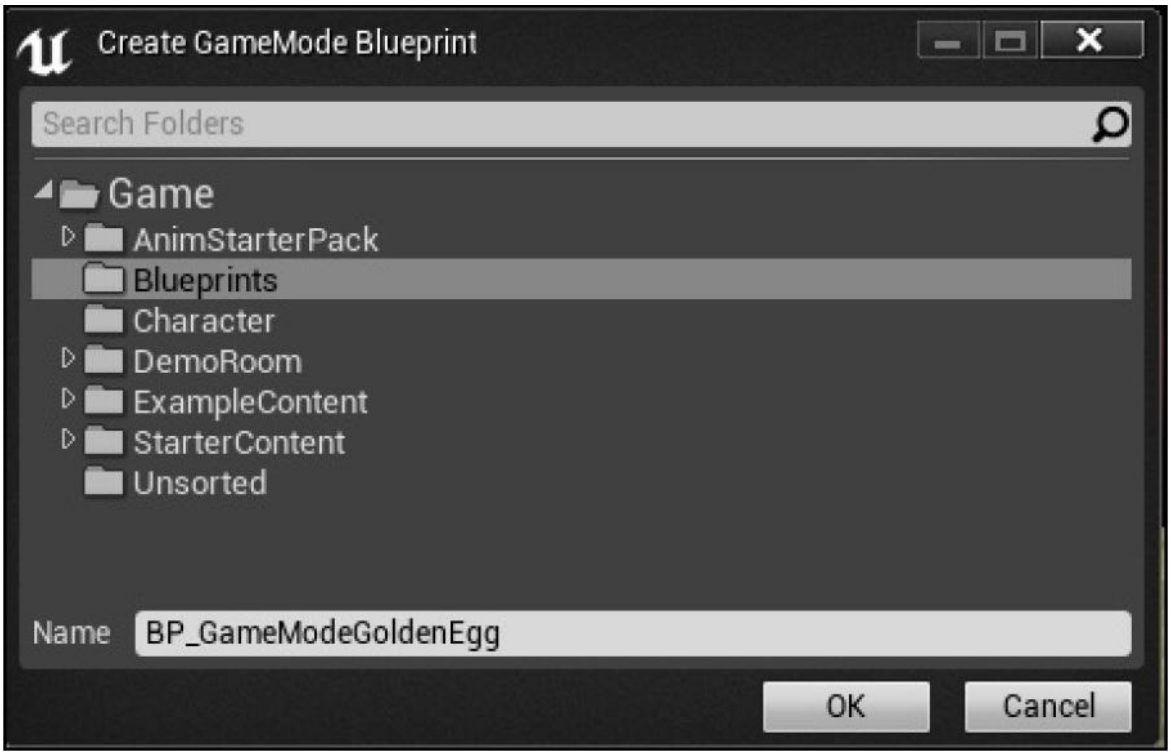
В Unreal Editor, создайте подкласс от **Game Mode** перейдя к **File** | **Add Code To Project…** и выберите **Game Mode**. Свой я назвал **GameModeGoldenEgg**.

GameMode в UE4 содержит правила игры и описания того, как игра проигрывается движком. Мы поработаем больше с нашим классом GameMode позже. Сейчас нам нужно сделать для него подкласс.

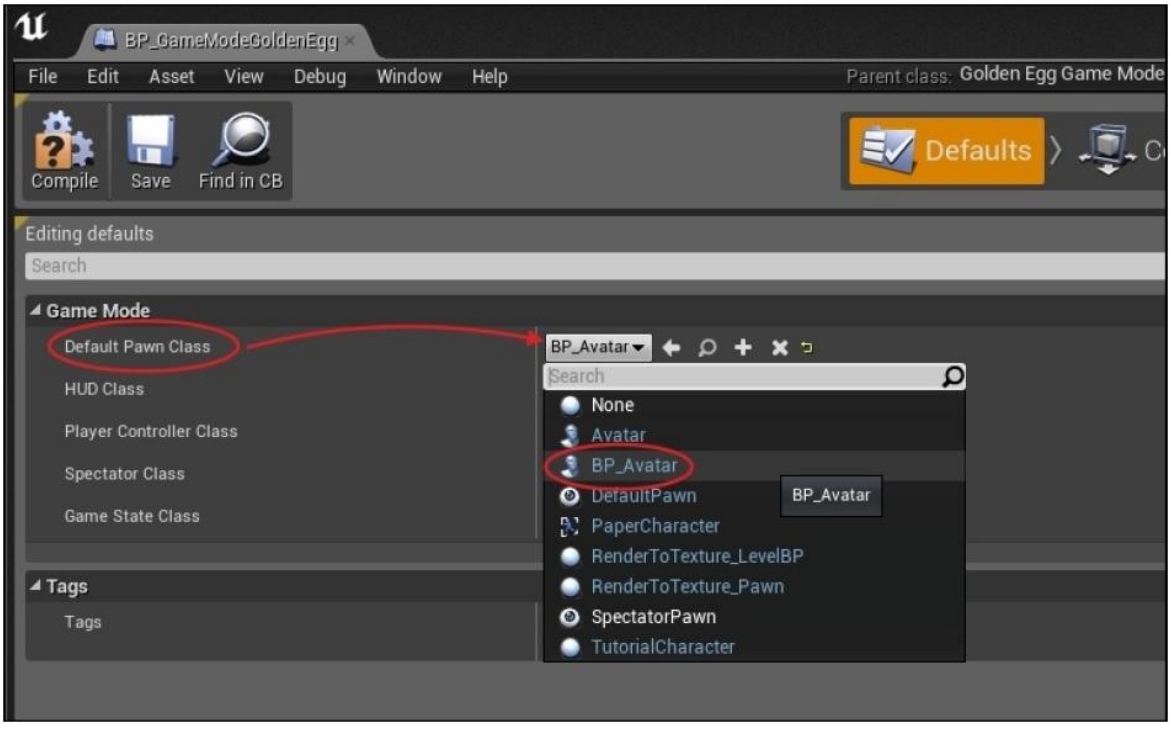
Заново компилируйте ваш проект из Visual Studio, так вы сможете создать схему (blueprint) **GameModeGoldenEgg**. Создайте схему **GameMode** перейдя к значку **Blueprints** панели меню сверху, и щёлкните по **GameMode**, а затем выберите **+Create** | **GameModeGoldenEgg** (или то, как вы назовёте ваш подкласс **GameMode** в шаге 1).



1. Дайте имя своему блупринту. Свой я назвал **BP_GameModeGoldenEgg**, как показано на следующем скриншоте:



2. Ваш только что созданный блупринт откроется в редакторе для blueprint. Если не откроется, вы можете открыть класс **BP_GameModeGoldenEgg** из вкладки **Class Viewer**.
3. Выберите ваш класс **BP_Avatar** на панели **Default Pawn Class**, как показано на следующем скриншоте. Панель **Default Pawn Class** это тип объекта, который будет применён для игрока.

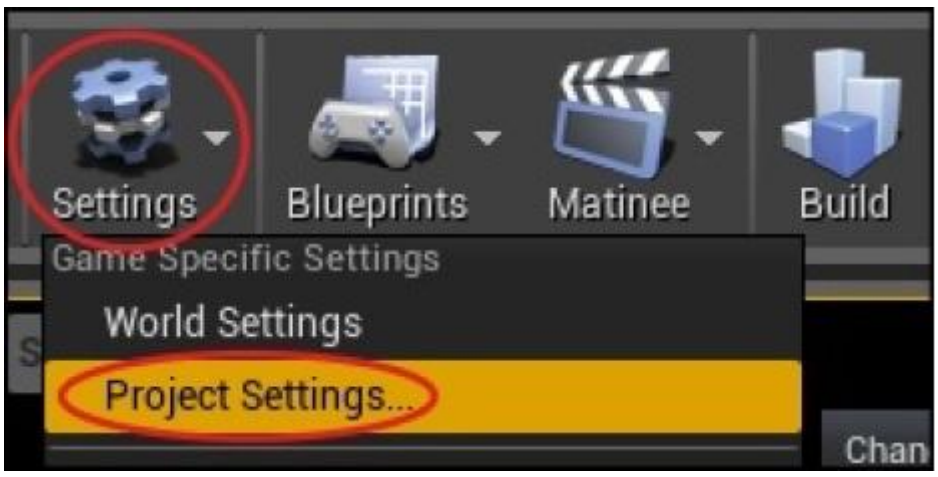


4. Теперь, запустите свою игру. Вы увидите вид сзади, так как камера расположена сзади, как показано здесь:

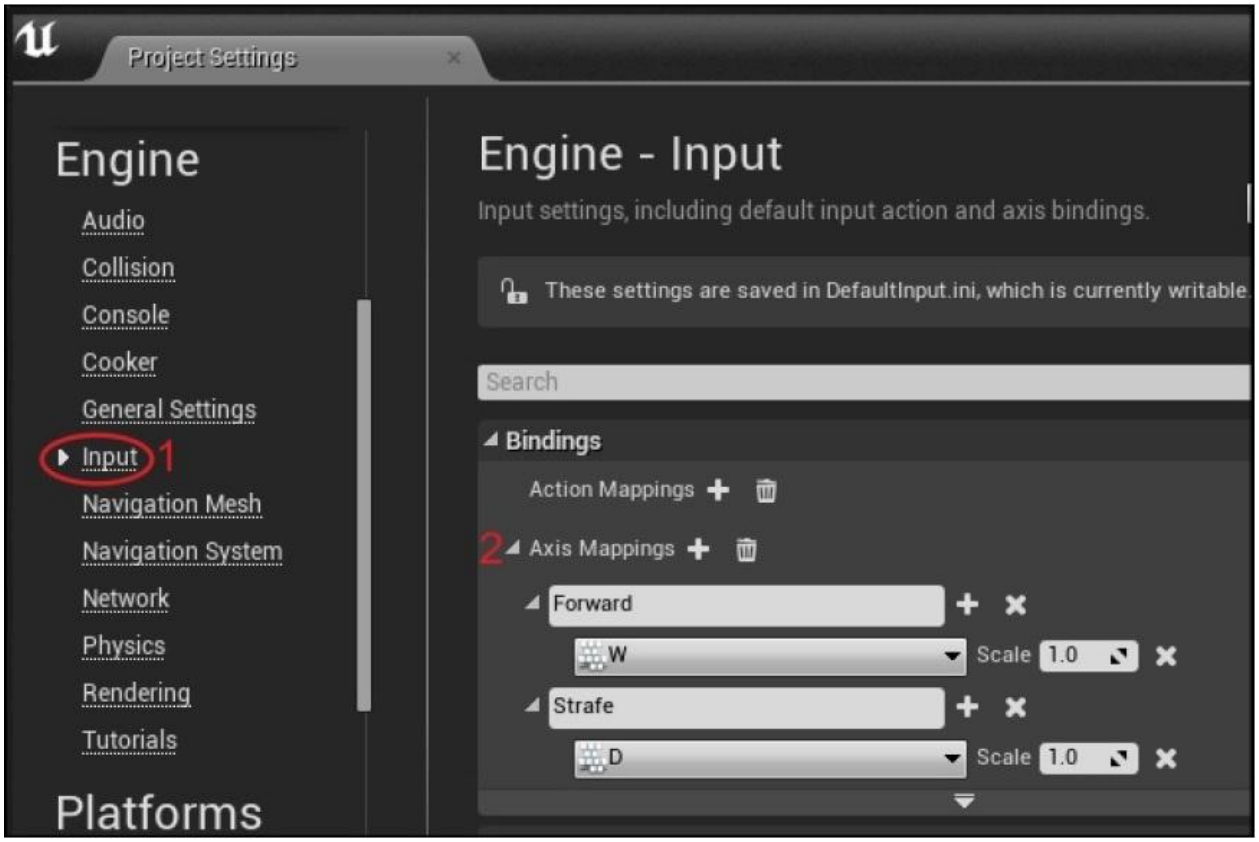
Вы заметите, что вы не можете двигаться. Почему бы это? Ответ, потому что мы ещё не установили ввод управления.

Устанавливаем ввод управления

1. Чтобы установить ввод управления, перейдите в **Settings** | **Project Settings…**:



2. Далее, в панели по левую сторону, под **Engine** прокручивайте вниз, пока не увидите **Input**.



3. По правую сторону, вы можете установить привязки (**Bindings**). Щёлкните по маленькой стрелке после **Axis Mapping**, чтобы открыть меню. Добавьте только две оси на карту, чтобы начать. Одна называется **Forward** – Вперёд (связана с буквой W на клавиатуре), а вторая называется **Strafe** – Обстрел (связанная с буквой D на клавиатуре). Запомните названия, которые вы даёте, мы скоро обратимся к ним в коде C++.
4. Закройте диалоговое окно **Project Settings**. Теперь откройте ваш C++ код.

В конструкторе Avatar.h, вам нужно добавить три объявления функции-члена, как показано здесь:

```
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
    GENERATED_UCLASS_BODY()
    // Новые! Эти три новых объявления функций-членов
    // будут применяться для движения нашего игрока!
    void SetupPlayerInputComponent(class UInputComponent* InputComponent)
override;
    void MoveForward( float amount );
    void MoveRight( float amount );
};
```

Обратите внимание, что первая функция-член, которую мы добавляем (SetupPlayerInputComponent) является подменой для виртуальной функции. SetupPlayerInputComponent это виртуальная функция в базовом классе Apawn.

В файле Avatar.cpp, вам нужно поместить тела функций. Добавьте следующие определения функции-члена:

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
InputComponent)
{
    check(InputComponent);
    InputComponent->BindAxis("Forward", this, &AAvatar::MoveForward);
    InputComponent->BindAxis("Strafe", this, &AAvatar::MoveRight);
}
```

Эта функция-член ищет привязки осей Forward и Strafe, которые мы только что создали в Unreal Editor и связывает их к функции-члену внутри этого класса. К какой функции-члену должны мы привязать? Почему мы должны привязывать к AAvatar::MoveForward и AAvatar::MoveRight. Вот определение функции-члена для этих двух функций:

```
void AAvatar::MoveForward( float amount )
{
    // Не вводите тело этой функции, если контроллер
    // ещё не установлен или если сумма для движения равна 0
    if( Controller && amount )
    {
        FVector fwd = GetActorForwardVector();
        // мы вызываем AddMovementInput, чтобы собственно двигать
        // игрока `суммой` в направлениях  `fwd`
        AddMovementInput(fwd, amount);
    }
}
void AAvatar::MoveRight( float amount )
{
    if( Controller && amount )
    {
        FVector right = GetActorRightVector();
        AddMovementInput(right, amount);
     }
}
```

Подсказка

Объект Controller и функции AddMovementInput определены в базовом классе APawn. Поскольку класс Avatar происходит от ACharacter, который в свою очередь происходит от APawn, мы можем свободно пользоваться всеми функциями-членами в базовом классе APawn. Теперь вы видите красоту наследования и повторного использования кода?

Упражнения

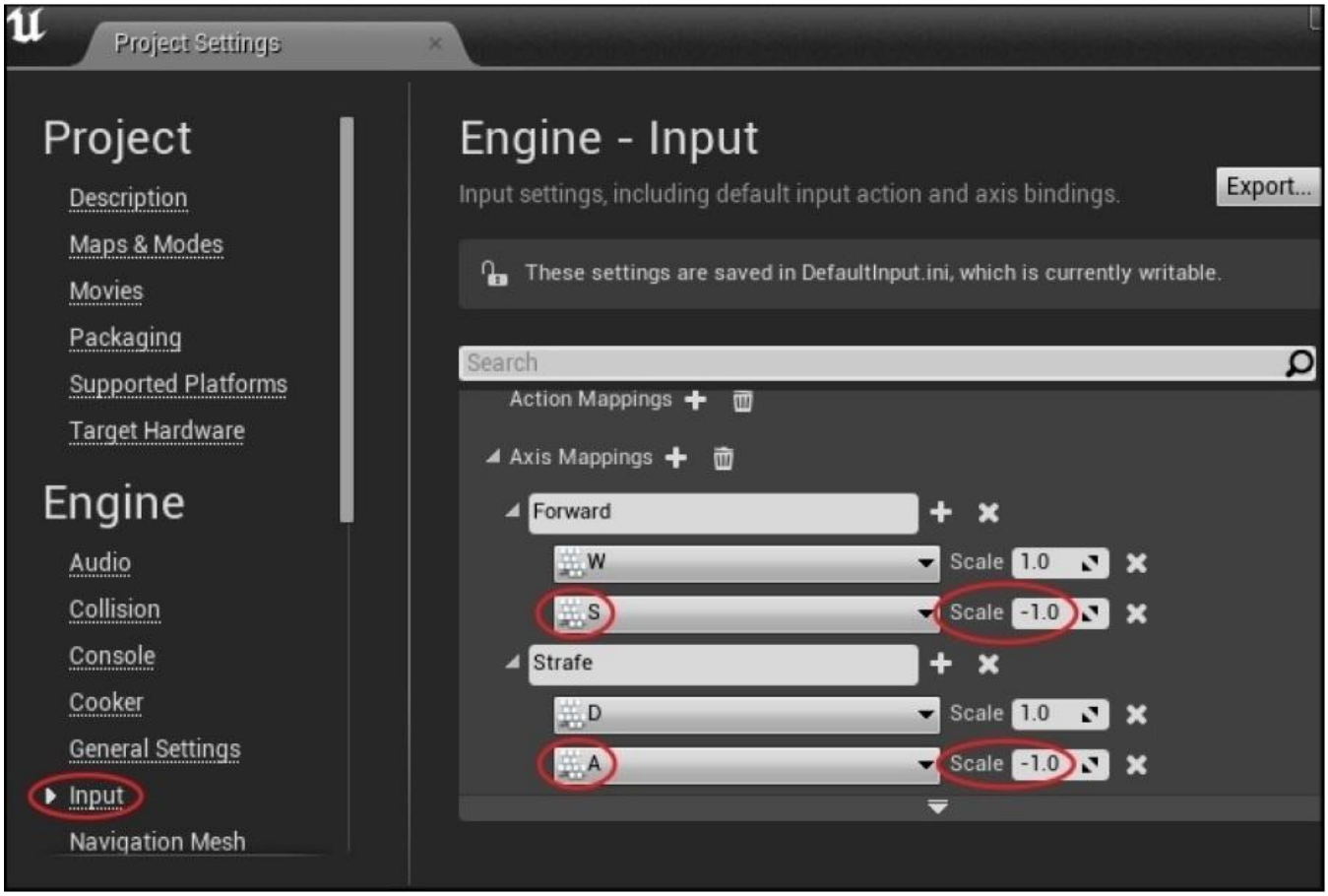
Добавьте привязки осей и функции C++, чтобы двигать игрока влево и назад.

Примечание

Вот подсказка: вам нужно только добавить привязки осей, если вы понимаете, что идти назад это негатив ходьбы вперёд.

Решение

Введите две дополнительные привязки осей, перейдя в **Settings** | **Project Settings…** | **Input**, как показано здесь:



В поле ввода Scale для S и A введите -1. Это инвертирует ось. Так что при нажатии клавиши S в игре, игрок будет идти назад. Попробуйте!

Альтернативно, вы можете определить две совсем раздельные функции-члены в вашем классе AAvatar,как показано далее и привязать клавиши A и S к AAvatar::MoveLeft и AAvatar::MoveBack, соответственно:

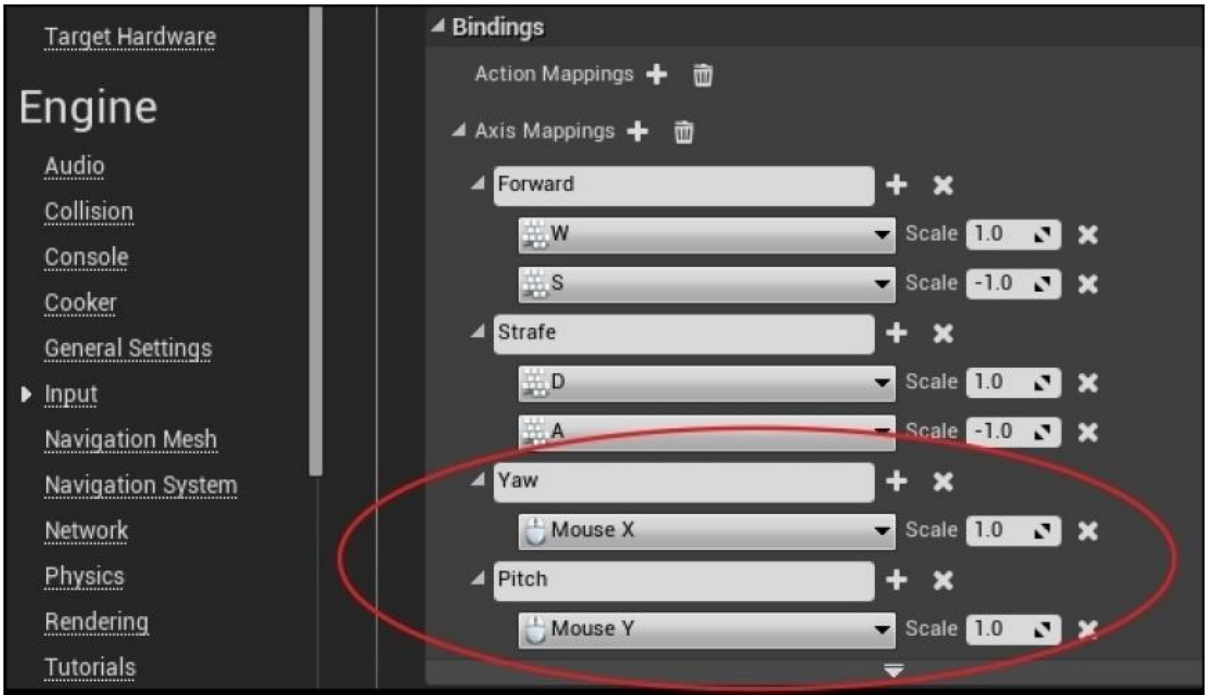
```
void AAvatar::MoveLeft( float amount )
{
    if( Controller && amount )
   {
       FVector left = -GetActorRightVector();
```

```
        AddMovementInput(left, amount);
    }
} void AAvatar::MoveBack( float amount )
{
    if( Controller && amount )
    {
        FVector back = -GetActorForwardVector();
        AddMovementInput(back, amount);
    }
}
```

Рыскание и тангаж

Мы можем менять направление, в котором смотрит игрок, настраивая рыскание – **Yaw** и тангаж – **Pitch** контроллера.

Всё что нам нужно сделать здесь, это добавить привязки осей для мыши, как показано на следующем скриншоте:



Из C++, вам нужно добавить два новых объявления функций-членов в AAvatar.h:

```
void Yaw( float amount );
void Pitch( float amount );
```

Тела этих функций-членов отправятся в файл AAvatar.cpp:

```
void AAvatar::Yaw( float amount )
{
    AddControllerYawInput(200.f * amount * GetWorld()- >GetDeltaSeconds());
}
void AAvatar::Pitch( float amount )
{
    AddControllerPitchInput(200.f * amount * GetWorld()- >GetDeltaSeconds());
}
```

Затем, добавьте две строки к SetupPlayerInputComponent:

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*
InputComponent)
{
    // … как и до этого, плюс:
    InputComponent->BindAxis("Yaw", this, &AAvatar::Yaw);
    InputComponent->BindAxis("Pitch", this, &AAvatar::Pitch);
}
```

Здесь обратите внимание на то, как мы умножили значение amount в функциях Pitch на 200. Это число представляет чувствительность мыши. Вы можете (должны) добавить элемент float в класс AAvatar, чтобы избежать хардкодинга этого числа чувствительности.

GetWorld()->GetDeltaSeconds() даёт вам количество времени, которое проходит между последним кадром и данным кадром. И это не много: GetDeltaSeconds() большинство времени (если ваша игра идёт 60 кадров в секунду) это должно быть около 16 миллисекунд (0.016 секунд).

Итак, сейчас у нас есть вод и контроль игрока. Чтобы добавить новую функциональность вашему аватару, всё, что вам нужно сделать это:

1. Привязать ваши действия клавиш или мыши перейдя к **Settings** | **Project Settings** | **Input**.
2. Добавить функцию-член для выполнения, когда эта клавиша нажата.
3. Добавить строку к SetupPlayerInputComponent, связывая имя ввода границы к функции-члену, которую мы хотим запускать, когда клавиша нажата.

Создание объекта неигрового персонажа

Итак, нам надо создать несколько **NPC** (**non-playable character** – неигровой персонаж). NPC это персонажи в игре, которые помогают игроку. Некоторые предлагают особые предметы, некоторые это продавцы в лавках, а некоторые делятся с игроком информацией. В этой игре они будут реагировать на игрока, когда он оказывается рядом. Давайте спрограммируем что-то из этого поведения.

Сначала создадим ещё один подкласс для **Character**. В редакторе UE4 перейдите в **File** | **Add Code To Project…** выберите класс **Character**, из которого вы сможете сделать подкласс. Назовите ваш подкласс NPC.

Теперь отредактируйте ваш код в Visual Studio. У каждого NPC будет сообщение для игрока, так что мы добавим свойство UPROPERTY() FString в класс NPC.

Совет

FStrings это версия UE4 типа C++ <string>. Когда программируете в UE4, вам следует использовать объекты FString вместо C++ объектов string стандартной библиотеки

шаблонов. В целом предпочтительно вам следует использовать встроенные в UE4 типы, так как они гарантируют кроссплатформенную совместимость.

Теперь, добавьте свойство UPROPERTY() FString в класс NPC, как показано в следующем коде:

```
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
    GENERATED_UCLASS_BODY()
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Collision)
    TSubobjectPtr<class USphereComponent> ProxSphere;
    // Это сообщение NPC, которое он должен сказать нам.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
    FString NpcMessage;
    // Когда вы создаёте blueptint из этого класса, вы хотите, чтобы вы могли
    // редактировать это сообщение в blueprint,
    // вот почему у нас есть
    // свойства  EditAnywhere и BlueprintReadWrite.
}
```

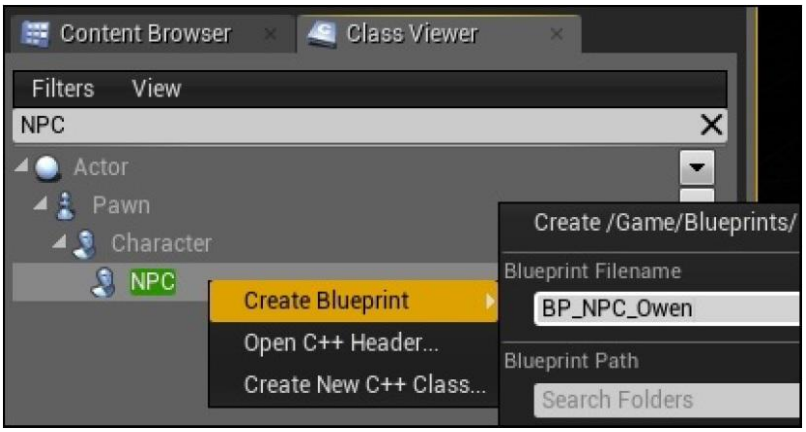
Обратите внимание, что мы помещаем свойства EditAnywhere и BlueprintReadWrite в макрос UPROPERTY. Это делает NpcMessage (сообщение NPC) редактируемым в blueprint.

Совет

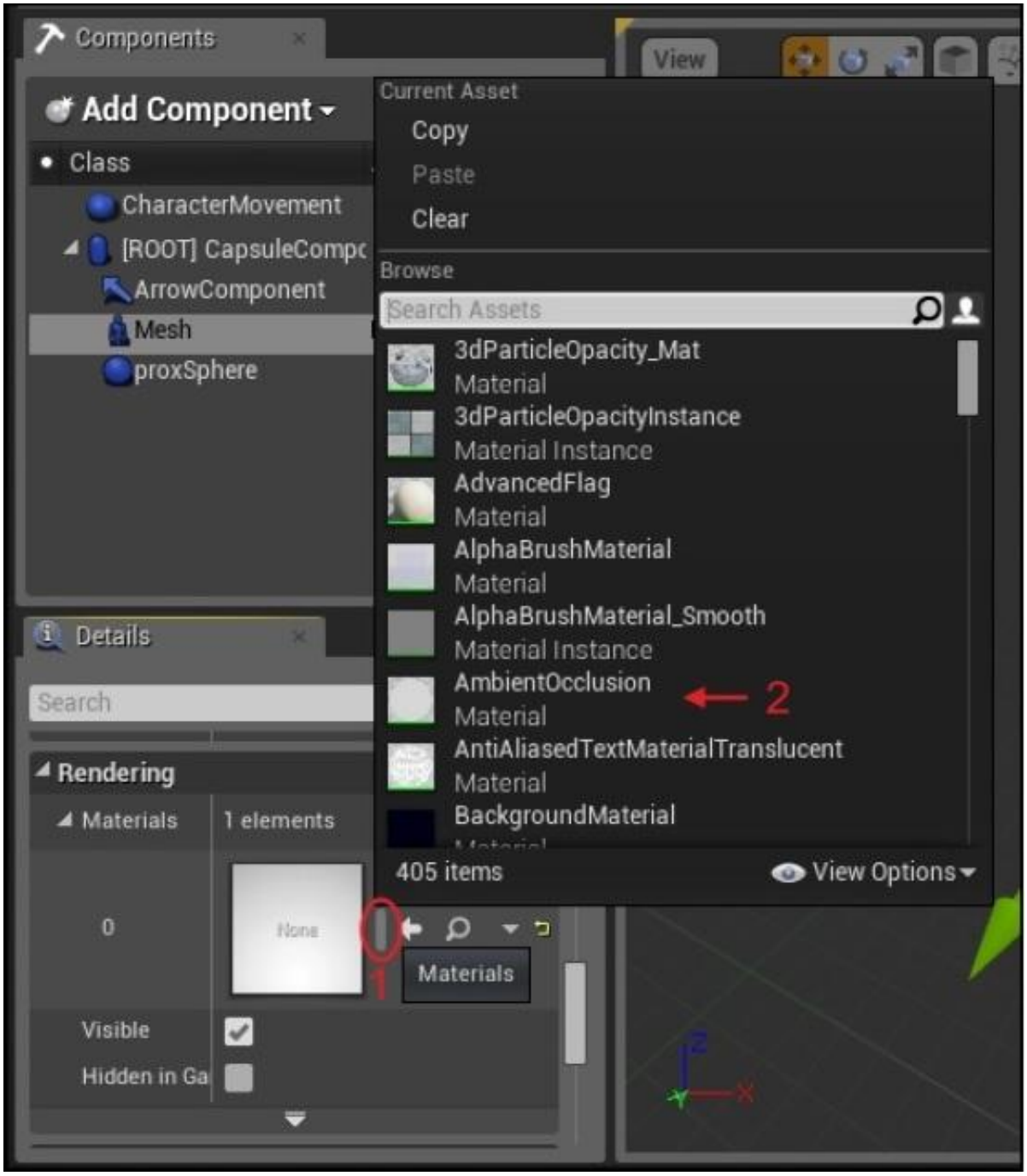
Полное описание особенностей свойств UE4, доступно на: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/Properties/

Заново компилируйте ваш проект (как мы делали это для класса Avatar). Затем, перейдите в **Class Viewer**, щёлкните правой кнопкой мыши по вашему классу NPC и создайте схему (blueprint) из него.

Каждый персонаж NPC, который вы хотите создать может быть основан на схеме (blueprint) вне класса NPC. Дайте каждому NPC уникальное имя, так как мы будем выбирать разные сетки моделей для каждого NPC, который у нас будет, как показано на следующем скриншоте:

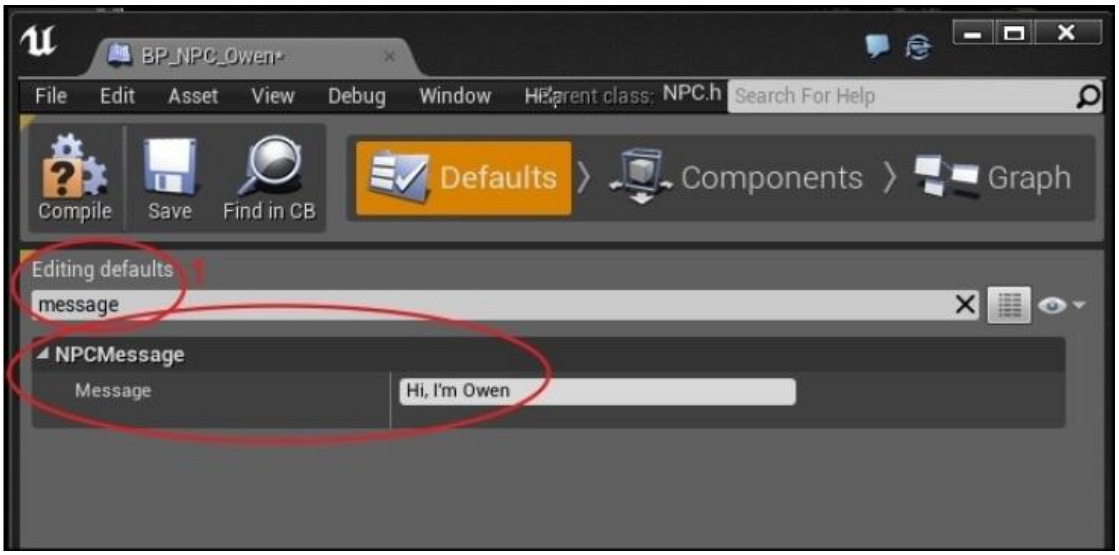


Теперь, откройте blueprint, выберите скелетную сетку (**mesh**) в **Add Components**, и отрегулируйте капсулу (как мы делали это для **BP_Avatar**). Вы также можете сменить материал вашего нового персонажа, так чтобы он отличался от игрока.



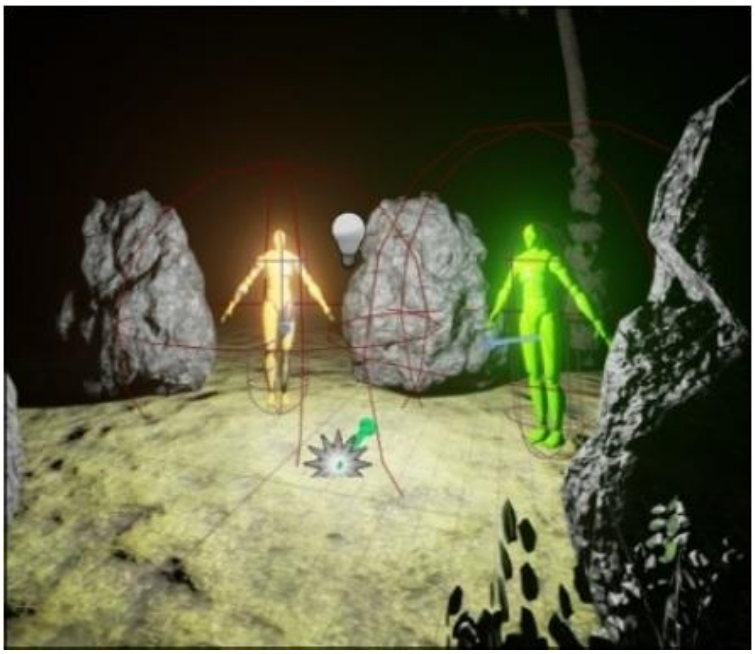
Смените материал вашего персонажа в свойствах вашей сетки. Под вкладкой Rendering (визуализация), щёлкните по значку +, чтобы добавить материал. Затем, щёлкните по объекту в форме капсулы, чтобы выбрать материал для визуализации.

Во вкладке **Defaults**, найдите свойство NpcMessage. Это наша связь между кодом C++ и схемой (blueprint), потому что мы ввели функцию UPROPERTY() для переменной FString NpcMessage. Это свойство будет редактируемым в UE4, как показано на следующем скриншоте:



Теперь перетащите BP_NPC_Owen в сцену. Вы также можете создать второго персонажа и третьего, и обязательно давайте им не повторяющиеся, уникальные имена, вид и сообщения.

Я создал две схемы (blueprint) для NPC основанных на базовом классе NPC. Это BP_NPC_Justin и BP_NPS_Owen. У них разный вид и разные сообщения для игрока.



Джастин и Оуен в сцене

Отображение цитат из каждого диалогового окна NPC

Чтобы отображать диалоговое окно, нам нужно приспособить **HUD** (heads-up display – отображение на уровне глаз). HUD отображает части графического пользовательского интерфейса и служит для предоставления важной информации игроку прямо в течение игры. В большинстве случаев элементы HUD (уровень, очки жизни, опыта, время, прицел, радар, карта, доступные способности и прочее) располагаются по периферии экрана, чтобы не мешать игроку.

В редакторе UE4 перейдите в **File** | **Add Code To Project…** и выберите класс HUD, из которого создаётся подкласс. Назовите свой подкласс как пожелаете. Свой я назвал MyHUD.

После того как вы создали класс MyHUD, дайте Visual Studio перезагрузиться. Мы немного отредактируем код.

Отображение сообщений в HUD

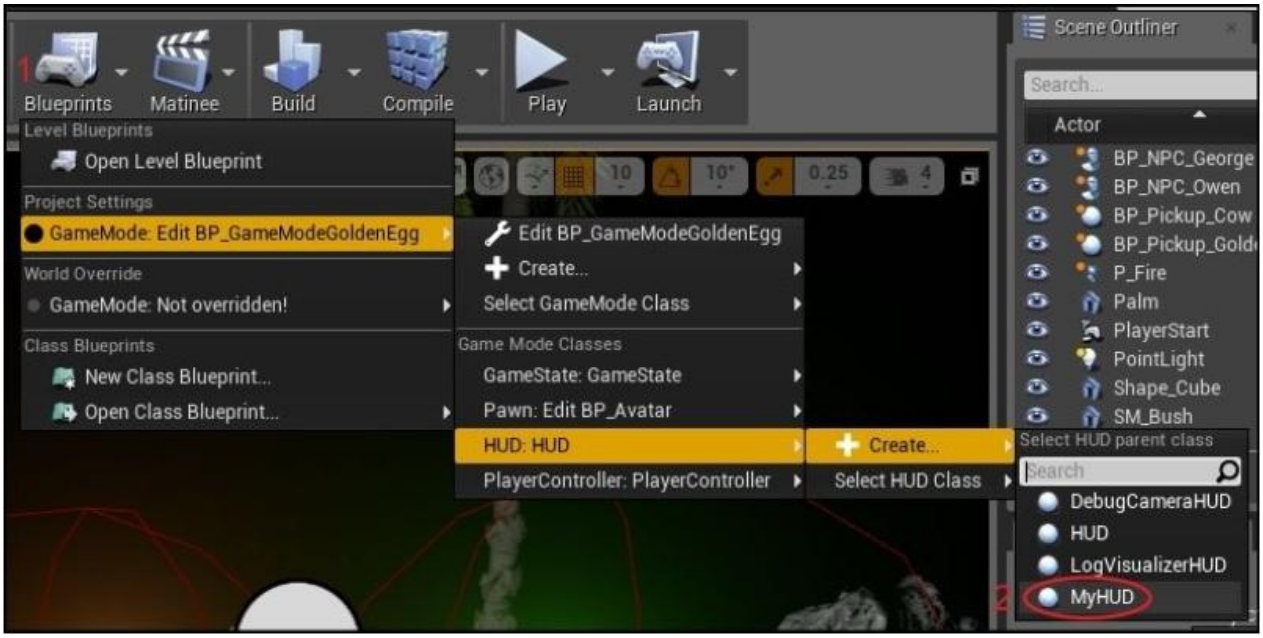
Внутри класса AMyHUD, нам нужно выполнить функцию DrawHUD() (изобразить HUD), чтобы изобразить наши сообщения в HUD и чтобы назначить шрифт для HUD, как показано в следующем коде:

```
UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
   GENERATED_UCLASS_BODY()
   // Шрифт, используемый для изображения текста в HUD.
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
   UFont* hudFont;
   // Добавьте эту функцию, чтобы изобразить HUD!
   virtual void DrawHUD() override;
};
```

Шрифт HUD будет установлен в спроектированной версии (blueprint) класса AMyHUD. Функция DrawHUD() запускается один раз за кадр. Перед тем как изображать внутри кадра, добавьте функцию в файл AMyHUD.cpp:

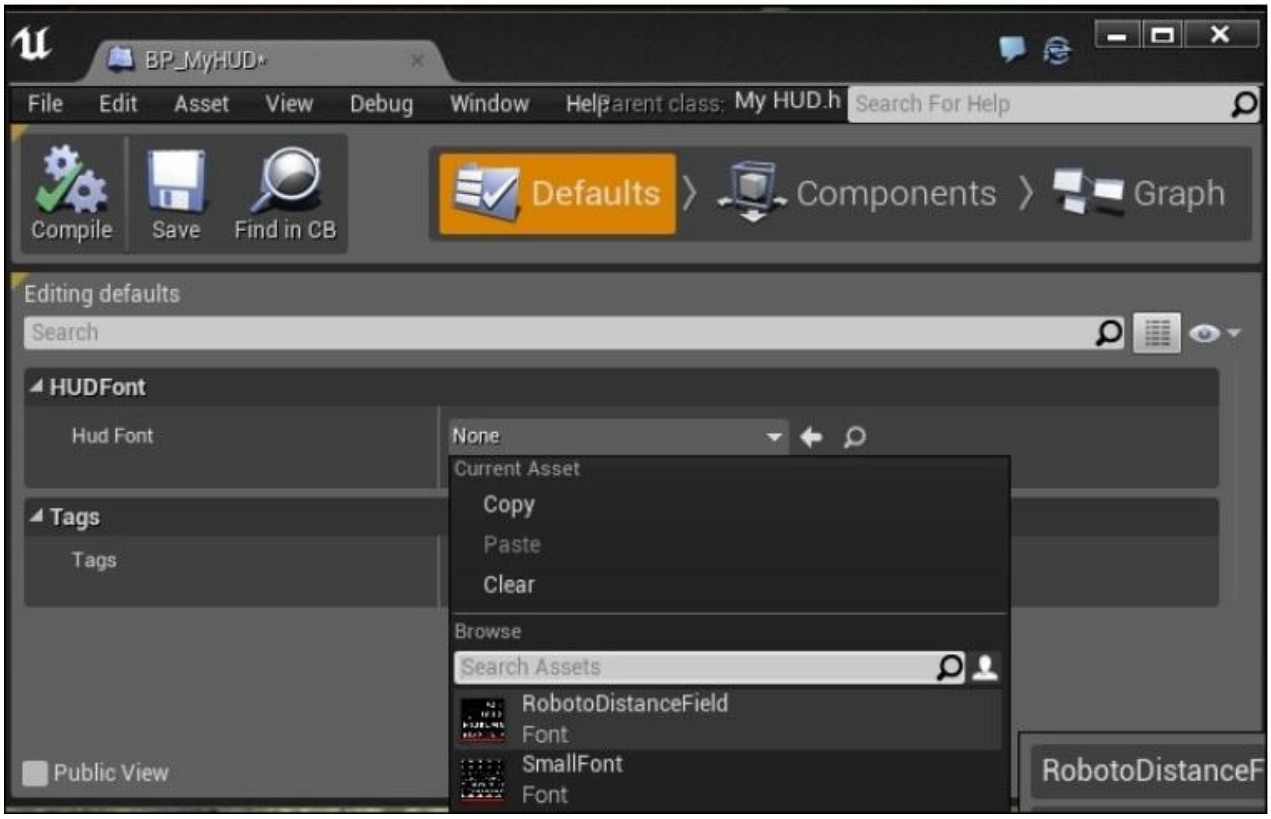
```
void AMyHUD::DrawHUD()
{
   // сначала вызываем функцию суперкласса DrawHUD()
   Super::DrawHUD();
   // затем переходим к изображению вашей задумки.
   // мы можем изобразить линии..
   DrawLine( 200, 300, 400, 500, FLinearColor::Blue );
   // и мы можем изобразить текст!
   DrawText( "Unreal приветствует вас!", FVector2D( 0, 0 ), hudFont,
   FVector2D( 1, 1 ), FColor::White );
}
```

Подождите! Мы ещё не назначили шрифт. Чтобы сделать это, нам нужно установить его в схеме (blueprint). Компилируйте и запустите ваш проект Visual Studio. Как только вы окажитесь в редакторе, перейдите в меню **Blueprints** сверху и перейдите к **GameMode** | **HUD** | **+ Create** | **MyHUD**.



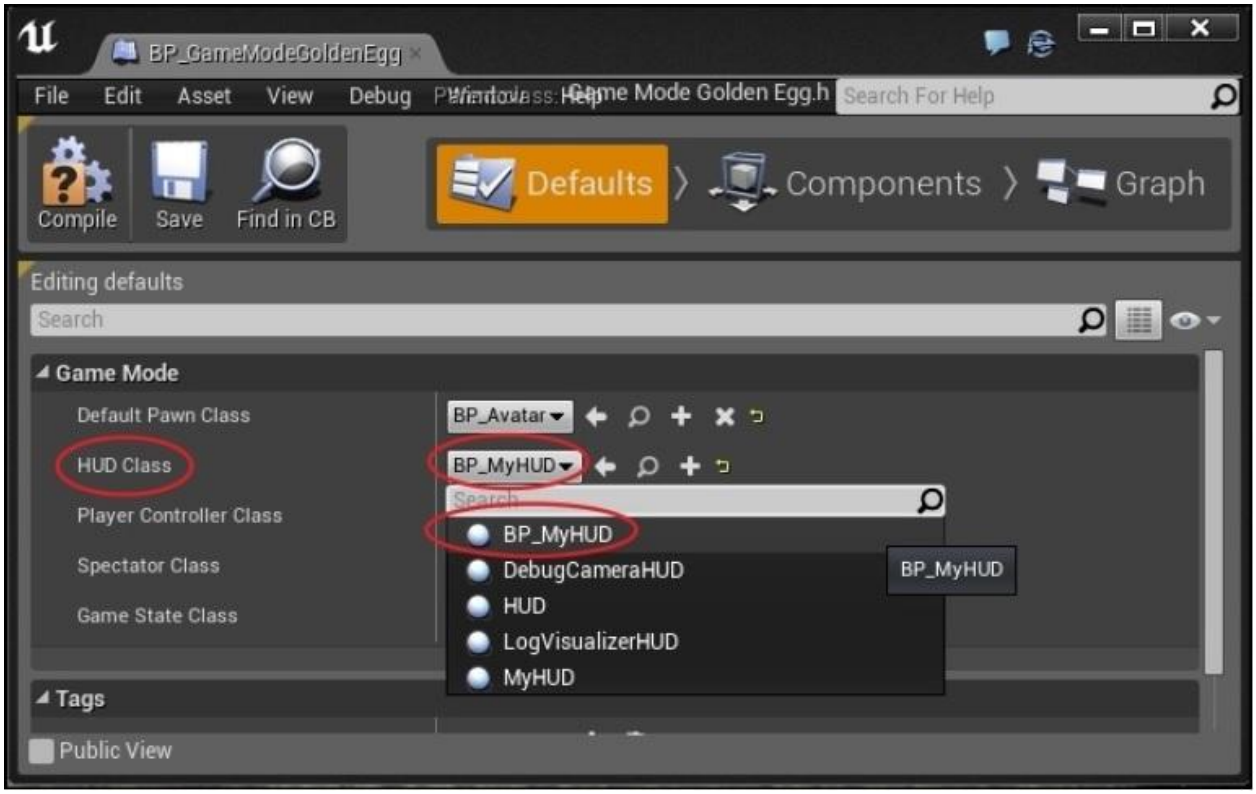
Создание blueprint класса MyHUD

Я дал имя BP_MyHUD. Отредактируйте BP_MyHUD и выберите шрифт в выпадающем меню под HUDFont (HUD шрифт):



Я выбрал RobotoDistanceField в качестве шрифта моего HUD

Затем отредактируйте ваш blueprint **Game Mode** (**BP_GameModeGoldenEgg**) и выберите ваш новый BP_MyHUD (не класс MyHUD) для панели **HUD Class**:



Протестируйте вашу программу, запустив её! Вы должны увидеть текст на экране.

Применение TArray<Message>

Каждое сообщение, которое мы хотим вывести игроку, будет иметь несколько свойств:

- Переменная FString для сообщения
- Переменная float для времени отображения
- Переменная FColor для цвета сообщения

Так что для нас есть смысл написать маленькую функцию struct, чтобы она содержала всю эту информацию.

Вверху MyHUD, введите следующее объявление struct:

```
struct Message
{
  FString message;
  float time;
  FColor color;
  Message()
  {
     // Устанавливаем время по умолчания.
     time = 5.f;
     color = FColor::White;
  }
  Message( FString iMessage, float iTime, FColor iColor )
  {
     message = iMessage;
     time = iTime;
     color = iColor;
  }
};
```

Примечание

Улучшенная версия для структуры Message (с цветом фона) в пакете кода для этой главы. Здесь мы применили код попроще, чтобы было легче понять главу.

Теперь, внутри класса AMyHUD, мы хотим добавить TArray этих сообщений. TArray - это специальный определённый в UE4 тип динамически увеличивающегося массива C++. Мы пройдём детальное применение TArray в следующей главе, но данное простое использование TArray должно быть хорошим первым знакомством, чтобы подпитать ваш интерес к пользе и важности массивов в играх. Объявление должно быть как TArray<Message>:

```
 UCLASS()
class GOLDENEGG_API AMyHUD : public AHUD
{
  GENERATED_UCLASS_BODY()
  // Шрифт, используемый для изображения текста в HUD.
  UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = HUDFont)
```

```
  UFont* hudFont;
  // Новое! Массив сообщений для отображения
  TArray<Message> messages;
  virtual void DrawHUD() override;
  // Новое! Функция для возможности добавлять сообщение для отображения
  void addMessage( Message msg );
};
```

Теперь, когда у NPC есть сообщение для отображения, нам просто нужно будет вызвать AMyHud::addMessage() с нашим сообщением. Сообщение будет добавлено в TArray, массив, где находятся сообщения, которые будут отображены. Когда время для отображения сообщения истекает, оно будет удалено из HUD:

В файле AMyHUD.cpp, добавьте следующий код:

```
 void AMyHUD::DrawHUD()
{
  Super::DrawHUD();
  // выполняем итерацию задом наперёд, так что если мы удаляем
  // пункт во время итерации, не возникнет никаких проблем
  for( int c = messages.Num() - 1; c >= 0; c-- )
  {
     // изображаем фоновое окно для сообщения
     // нужного размера
     float outputWidth, outputHeight, pad=10.f;
     GetTextSize( messages[c].message, outputWidth, outputHeight, hudFont,
     1.f );
     float messageH = outputHeight + 2.f*pad;
     float x = 0.f, y = c*messageH;
     // чёрный фон
     DrawRect( FLinearColor::Black, x, y, Canvas->SizeX, messageH );
     // изображаем наше сообщение используя hudFont
     DrawText( messages[c].message, messages[c].color, x + pad, y + pad,
     hudFont );
     // сокращаем время отображения на время прошедшее
     // с последнего кадра.
     messages[c].time -= GetWorld()->GetDeltaSeconds();
     // если время сообщения вышло, удаляем его
     if( messages[c].time < 0 )
     {
        messages.RemoveAt( c );
     }
  }
}
void AMyHUD::addMessage( Message msg )
{
  messages.Add( msg );
}
```

Функция AMyHUD::DrawHUD() сейчас изображает все сообщения в массиве messages и упорядочивает каждое сообщение в этом массиве количеству времени прошедшему с последнего кадра. Истёкшие сообщения удаляются из собрания messages, как только их значение time падает ниже 0.

Упражнение

Выполните рефакторинг функции DrawHUD(), так чтобы код, который изображает сообщения на экране, был в раздельных функциях названных DrawMessages().

Переменная Canvas доступна только в DrawHUD(), так что вам будет нужно сохранить Canvas->SizeX и Canvas->SizeY в переменных уровневого класса.

Примечание

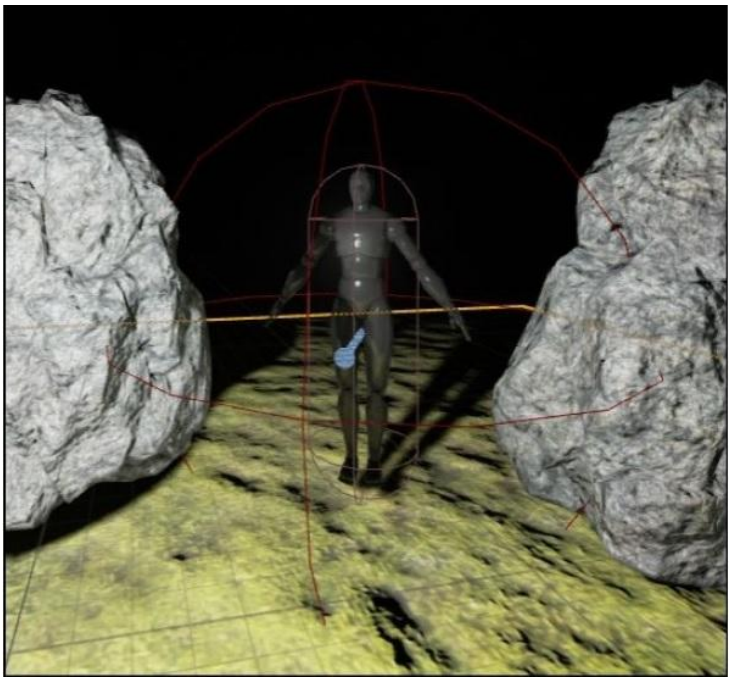
Рефакторинг означает внутреннюю смену способа работы кода, так чтобы он был более организованным и более удобным для чтения, но в то же время имел тот же видимый результат для пользователя, запускающего программу. Зачастую рефакторинг это хорошая практика. Причина, по которой проводится рефакторинг – в том, что никто на момент начала написания кода, не знает, как должен будет выглядеть код в итоге.

Решение

Посмотрите на функцию AMyHUD::DrawMessages() в пакете кода для этой главы.

Срабатывание события по приближению к NPC

Чтобы событие сработало рядом с NPC, нам нужно установить дополнительный объём выявления столкновения, который немного шире, чем форма капсулы по умолчанию. Дополнительный объём выявления столкновения будет представлять из себя сферу вокруг каждого NPC. Когда игрок водит в сферу NPC, то NPC реагирует и выводит сообщение.



Мы собираемся добавить тёмно-красную сферу для NPC, чтобы он мог говорить, когда рядом находится игрок

Внутри вашего файла класса NPC.h, добавьте следующий код, чтобы объявить ProxSphere и UFUNCTION названную Prox:

```
UCLASS()
class GOLDENEGG_API ANPC : public ACharacter
{
   GENERATED_UCLASS_BODY()
   // Это сообщение NPC, которое он должен сказать нам.
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
   FString NpcMessage;
   // Сфера, с которой может сталкиваться игрок, чтобы получить предмет
   UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Collision)
   TSubobjectPtr<class USphereComponent> ProxSphere;
   // Соответствующее тело этой функции это
   // ANPC::Prox_Implementation, __не__ ANPC::Prox()!
   // Это немного странно и не то, что вы ожидали,
   // но это случается, потому что это BlueprintNativeEvent (родное событие схемы (blueprint))
   UFUNCTION(BlueprintNativeEvent, Category = "Collision")
   void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
   OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult );
};
```

Выглядит немного запутанно, но на самом деле не так всё сложно. Здесь мы объявляем дополнительную сферу объёма границ, названную ProxSphere, которая выявляет, когда игрок находится рядом с NPC.

В файле NPC.cpp, нам нужно добавить следующий код, чтобы завершить выявление близости:

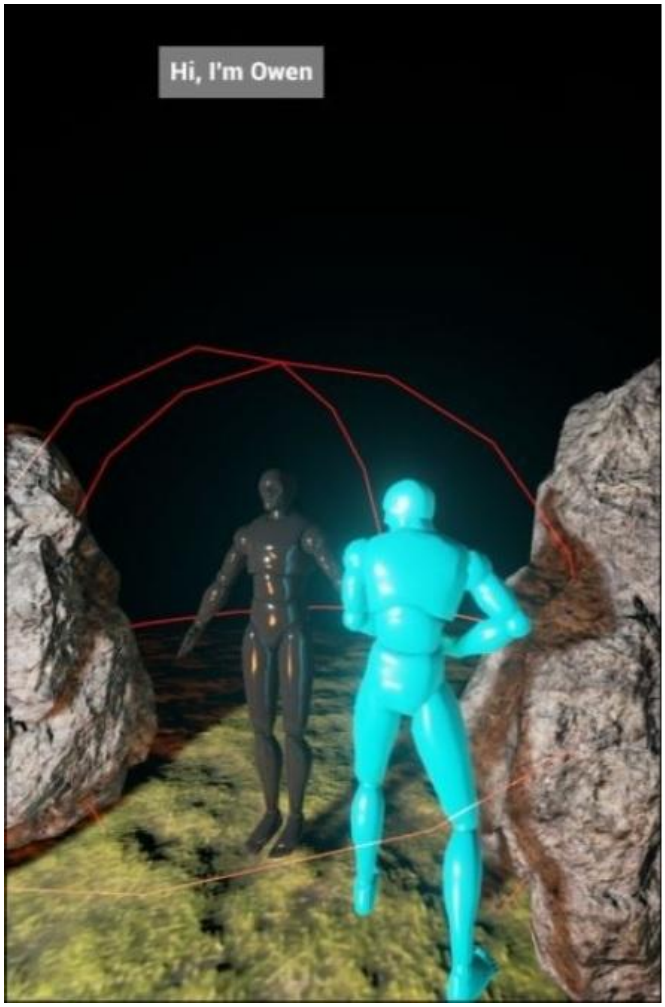
```
ANPC::ANPC(const class FPostConstructInitializeProperties& PCIP) :
Super(PCIP)
{
   ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this,
   TEXT("Proximity Sphere"));
   ProxSphere->AttachTo( RootComponent );
   ProxSphere->SetSphereRadius( 32.f );
   // Код для запуска ANPC::Prox(), когда эта сфера приближения
   // пересекается с другим актором.
   ProxSphere->OnComponentBeginOverlap.AddDynamic( this, &ANPC::Prox );
   NpcMessage = "Hi, I'm Owen";// сообщение по умолчанию, может быть отредактировано
                                    // в blueprint
}
// Обратите внимание! Хотя это ANPC::Prox() и было объявлено в заголовочном файле,
// теперь это вот здесь ANPC::Prox_Implementation.
void ANPC::Prox_Implementation( AActor* OtherActor, UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
SweepResult )
{
   // Здесь наш код будет выполнять то, что будет происходить
   // при пересечении
}
```

Код для вывода сообщения на HUD, если кто-то рядом с NPC

Когда игрок рядом со сферой объёма столкновения NPC, мы выведем сообщение на HUD, которое даёт знать игроку, что говорит NPC.

Вот полное осуществление ANPC::Prox_Implementation:

```
void ANPC::Prox_Implementation( AActor* OtherActor, UPrimitiveComponent*
OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult &
SweepResult )
{
    // если актор, с которым произошло пересечение, это не игрок,
    // вы просто должны вернуться из функции
    if( Cast<AAvatar>( OtherActor ) == nullptr )
    {
       return;
    }
    APlayerController* PController = GetWorld()- >GetFirstPlayerController();
    if( PController )
    {
       AMyHUD * hud = Cast<AMyHUD>( PController->GetHUD() );
       hud->addMessage( Message( NpcMessage, 5.f, FColor::White ) );
    }
}
```



Приветствие Оуена

Первое, что мы делаем в функции это приведение OtherActor (другой актор) (то, что появляется рядом с NPC) к AAvatar. Приведение происходит (и не nullptr), когда OtherActor это объект AAvatar. Мы получаем объект HUD (что происходит с прикреплением к контроллеру игрока) и передаём сообщение от NPC на HUD. Сообщение появляется, всегда когда игрок находится в пределах границ красной сферы NPC.

Упражнения

1. Добавьте имя функции UPROPERTY для имени NPC, чтобы имя NPC был редактируемым в схеме (blueprint), подобно сообщению, которое есть у NPC для игрока. Покажите имя NPC в выводе на экран.
2. Добавьте функцию (напишите UTexture2D*) для лицевой текстуры NPC. Вставьте лицо NPC рядом с его сообщением в выводе.
3. Визуализируйте на экране HP игрока как панель (заполненный прямоугольник).

Решения

Добавьте это свойство к классу ANPC:

```
// Это имя NPC
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
FString name;
```

Затем, в ANPC::Prox_Implementation, смените строку переданную в HUD на:

```
name + FString(": ") + message
```

Таким образом, имя NPC будет добавлено в сообщение.

Добавьте это свойство к классу ANPC:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NPCMessage)
UTexture2D* Face;
```

Затем вы можете выбрать изображения лиц, чтобы прикрепить к лицу NPC в схеме (blueprint).

Прикрепите текстуру к вашему struct Message:

```
UTexture2D* tex;
```

Чтобы визуализировать на экране эти изображения, вам нужно добавить вызов DrawTexture() с нужной текстурой переданной в неё:

```
DrawTexture( messages[c].tex, x, y, messageH, messageH, 0, 0, 1, 1 );
```

Обязательно проверьте рабочая ли текстура, перед тем как визуализировать её. Изображения должны выглядеть подобно тому, что показано здесь, вверху экрана:

Это то, как функция изображает оставшееся здоровье игрока в форме полосы:

```
void AMyHUD::DrawHealthbar()
{
    // Изображаем полосу здоровья.
    AAvatar *avatar = Cast<AAvatar>(
    UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
    float barWidth=200, barHeight=50, barPad=12, barMargin=50;
    float percHp = avatar->Hp / avatar->MaxHp;
    DrawRect( FLinearColor( 0, 0, 0, 1 ), Canvas->SizeX - barWidth - barPad
    - barMargin, Canvas->SizeY - barHeight - barPad - barMargin, barWidth +
    2*barPad, barHeight + 2*barPad );
    DrawRect( FLinearColor( 1-percHp, percHp, 0, 1 ), Canvas->SizeX -
    barWidth - barMargin, Canvas->SizeY - barHeight - barMargin,
    barWidth*percHp, barHeight );
}
```

Выводы

В этой главе, прошли много материала. Мы показали вам, как создавать персонажа и отображать его на экране, управлять вашим персонажем с помощью привязок осей, и как создавать и отображать NPC, который может передавать сообщения на HUD.

В дальнейших главах, мы будем развивать нашу игру дальше, добавляя *Систему Инвентаризации и Подбор Предметов* в Главе 10, а также код и концепцию подсчёта того что есть у игрока. Но перед этим, мы тщательней объясним некоторые типы контейнеров UE4 в Главе 9. *Шаблоны и обычно используемые контейнеры*.

Глава 9. Шаблоны и обычно используемые контейнеры.

В Главе 7. *Динамическое распределение памяти*, мы говорили о том, как вы будете использовать динамическое распределение памяти, если вы хотите создать новый массив, размер которого неизвестен в момент компиляции. Форма динамического распределения памяти такова int * array = new int[number_of_elements].

Вы также видите это динамическое распределение, использует ключевое слово new[], требующее от вас позже вызов delete[] в массиве, иначе у вас будет утечка памяти. Такое управление памятью это тяжёлая работа.

Есть ли способ создать массив динамического размера и чтобы память автоматически управлялась для вас самим C++? Ответ да. Есть объектные типы C++ (называемый контейнеры), которые управляют динамическим распределением и перемещением памяти автоматически. UE4 предоставляет пару типов для хранения ваших данных в динамически изменяемых собраниях.

Есть две разные группы шаблонных контейнеров. Есть семья UE4 контейнеров (начинающаяся с T*) и семья контейнеров **Стандартной библиотеки шаблонов** (**Standard Template Library**) (**STL**) C++. И есть некоторые отличия между контейнерами UE4 и контейнерами STL, но отличия небольшие. Наборы контейнеров UE4 написаны с выполнением игры в уме. Контейнеры STL C++ также работают хорошо и их интерфейсы немного более последовательны (последовательность в API это то, что вы предпочли бы). Какой набор контейнеров использовать, решать вам. Однако вам рекомендуется использовать набор контейнеров UE4, так как это гарантирует, что у вас не возникнет проблем с кроссплатформеностью, когда вы попробуете компилировать ваш код.

Отладка выходных данных в UE4

Для всего кода в этой главе (также как и в последующих главах) требуется, чтобы вы работали в проекте UE4. В целях тестирования TArray, я создал проект базового кода и назвал его TArrays. В конструкторе ATArraysGameMode::ATArraysGameMode я использую элемент отладки вывода, чтобы выводить текст на консоль.

Вот как будет выглядеть код:

```
ATArraysGameMode::ATArraysGameMode(const class
FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
    if( GEngine )
    {
        GEngine->AddOnScreenDebugMessage( 0, 30.f, FColor::Red, "Hello!" );
    }
}
```

Если вы компилируете и запустите этот проект, вы увидите текст отладки в верхнем левом углу окна вашей игры, когда вы начнёте игру. Вы можете использовать вывод отладки, чтобы увидеть внутренние данные вашей программы в любое время. Просто убедитесь, что объект GEngine существует о время отладки вывода. Вывод предыдущего кода показан на следующем скриншоте:

TArray<T> в UE4

TArray это версия динамического массива в UE4. Чтобы понять, что такое переменная TArray<T>, вам сначала надо узнать, для чего служит опция <T>, между угловых скобок. Опция <T> означает, что тип данных хранящихся в массиве это переменная. Вам нужен массив int? Тогда создайте переменную TArray<int>. Нужна переменная TArray с типом double? Создайте переменную TArray<double>.

В общем, когда есть <T>, вы можете вставлять тип C++ на ваш выбор. Давайте пойдём дальше и покажем это на примере.

Пример использования TArray<T>

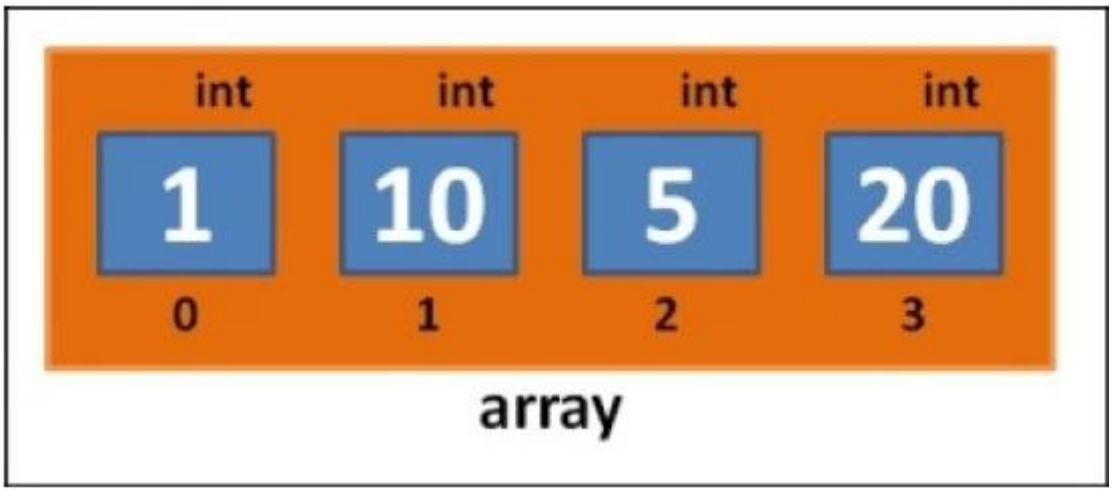
Переменная TArray<T> это просто массив типов int. Переменная TArray<Player*> будет массивом указателей Player*. Массив с динамически изменяемым размером, и элементы могут быть добавлены в конец этого массива, после их создания.
Чтобы создать переменную TArray<T>, всё что вам нужно сделать, это применить синтаксис распределения нормальной переменной:

```
TArray<int> array;
```

Изменения в переменной TArray производятся с использованием функций-членов. Есть пара функций-членов, которые вы можете применять для переменной TArray. Первая функция-член, о которой вам надо знать, это способ, которым вы добавляете значение в массив, как показано в следующем коде:

```
array.Add( 1 );
array.Add( 10 );
array.Add( 5 );
array.Add( 20 );
```

Эти четыре строки кода произведут значение массива в памяти, как показано на следующем изображении:

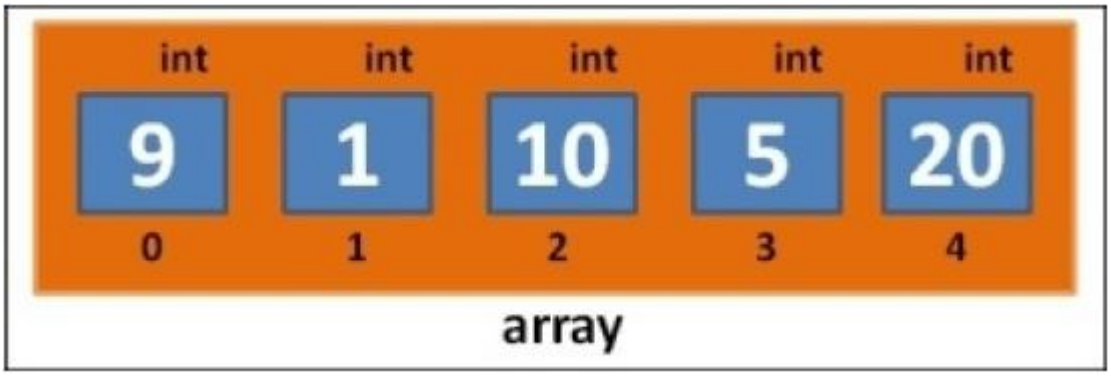


Когда вы вызываете array.Add(number), новое число идёт в конец массива. Так как мы добавили в массив числа **1**, **10**, **5** и **20** в таком порядке, то в таком порядке они и идут в массиве.

Если вы хотите ввести число в начале или в середине массива, то это тоже возможно. Всё что вам нужно сделать, это применить функцию array.Insert(value, index), как показано в следующей строке кода:

```
array.Insert( 9, 0 );
```

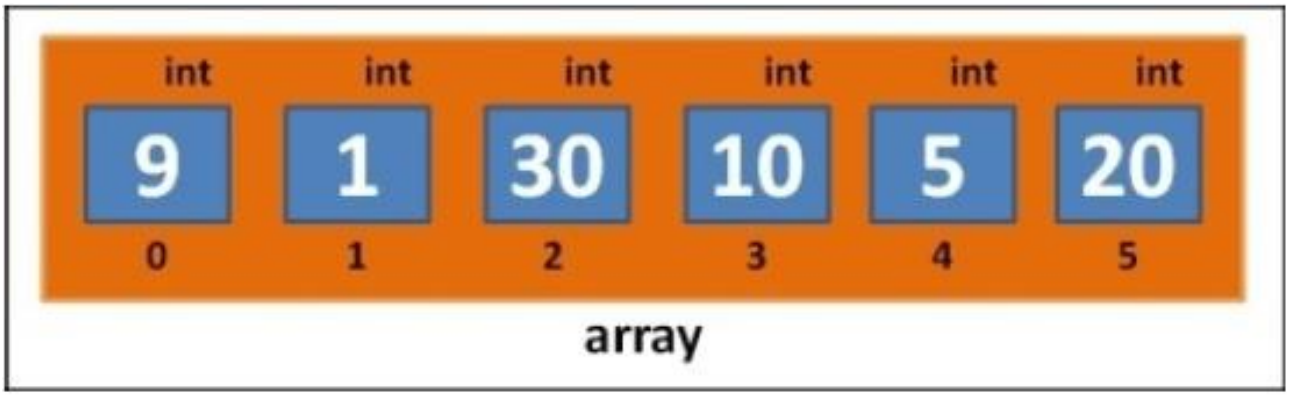
Эта функция поставит число **9** на позицию массива **0** (в начало). Это значит, что остальные элементы массива пододвинутся вправо, как показано на следующем изображении:



Мы можем ввести ещё один элемент на позицию массива **2**, применив следующую строку кода:

```
array.Insert( 30, 2 );
```

Эта функция переупорядочит массив, как показано на следующем изображении:



Подсказка

Если вы введёте число на позицию в массиве, которая находится за пределами массива, UE4 выйдет из строя. Так что будьте осторожны и не делайте так.

Итерация TArray

Вы можете итерировать (проходить) элементы переменной TArray двумя способами: либо используя целочисленную индексацию, либо используя итератор. Я покажу вам оба этих способа здесь.

Ваниль цикла for и запись в квадратных скобках

Использование целых чисел для индексации элементов массива иногда называется “ваниль” цикла for. Доступ к элементам массива можно получить, используя array[index], где индекс это номерная позиция элемента в массиве:

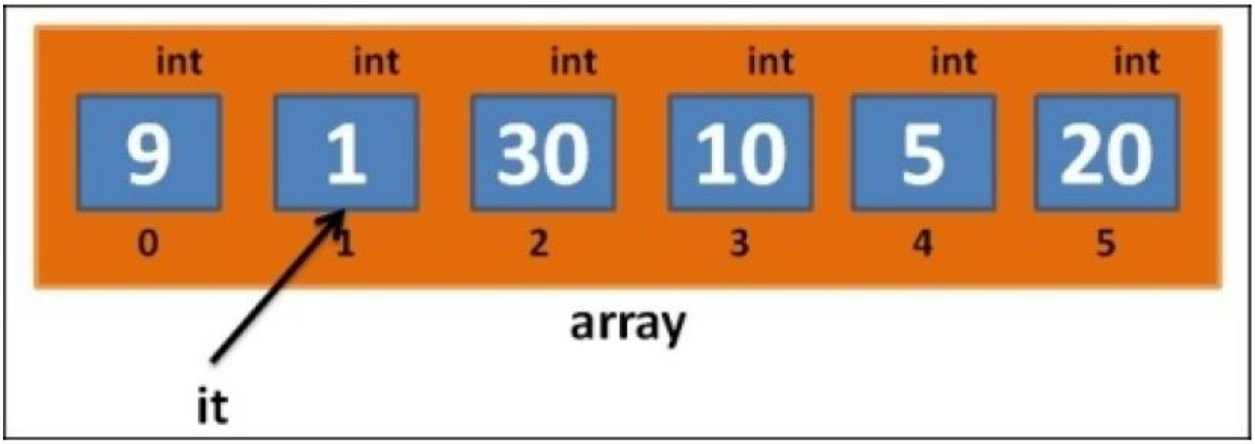
```
for( int index = 0; index < array.Num(); index++ )
{
    // выводим элемент массива на экран, применяя сообщение отладки
    GEngine->AddOnScreenDebugMessage( index, 30.f, FColor::Red,
FString::FromInt( array[ index ] ) );
}
```

Итераторы

Вы также можете использовать итератор, чтобы проходить по элементам массива один за другим, как показано в следующем коде:

```
int count = 0; // отслеживаем номерной индекс в массиве
for( TArray<int>::TIterator it = array.CreateIterator(); it; ++it )
{
    GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red,
    FString::FromInt( *it ) );
}
```

Итераторы являются указателями в массиве. Итераторы могут использоваться, чтобы просматривать или изменять значения внутри массива. Пример итератора показан на следующем изображении:



Суть итератора: это внешний объект, который может заглядывать и просматривать значения массива. Выполнение ++ продвигает итератор к просмотру следующего элемента.

Итератор должен подходить для собрания элементов, через которые он проходит. Чтобы пройти через переменную TArray<int>, вам нужен тип итератора TArray<int>::TIterator.

Мы используем знак *, чтобы смотреть значение позади итератора. В предыдущем коде, мы использовали (*it), чтобы получить целочисленное значение от итератора. Это называется разыменование. Разыменование итератора означает посмотреть его значение.

Операция ++it, которая происходит в конце каждой итерации цикла for, инкрементирует итератор, продвигая его к месту следующего элемента в списке.

Введите код в программу и теперь проверьте, как она работает. Вот пример программы, которую мы создавали до этого, используя TArray (всё в конструкторе ATArraysGameMode::ATArraysGameMode()):

```
ATArraysGameMode::ATArraysGameMode(const class
FPostConstructInitializeProperties& PCIP) : Super(PCIP)
{
    TArray<int> array;
    array.Add( 1 );
    array.Add( 10 );
    array.Add( 5 );
    array.Add( 20 );
    array.Insert( 9, 0 );// ставим 9 спереди
    array.Insert( 30, 2 );// ставим 30 на индекс 2
    if( GEngine )
    {
      for( int index = 0; index < array.Num(); index++ )
      {
        GEngine->AddOnScreenDebugMessage( index, 30.f, FColor::Red,
        FString::FromInt( array[ index ] ) );
      }
    }
}
```

Вывод предыдущего кода, показан на следующем скриншоте:

Выясняем находится ли элемент в TArray

Находить контейнеры UE4 легко. Это делается с применением функции-члена Find. Используя массив, который создали ранее, мы можем найти индекс значения 10, написав следующий код:

```
int index = array.Find( 10 ); // на изображении сверху, это был бы индекс 3
```

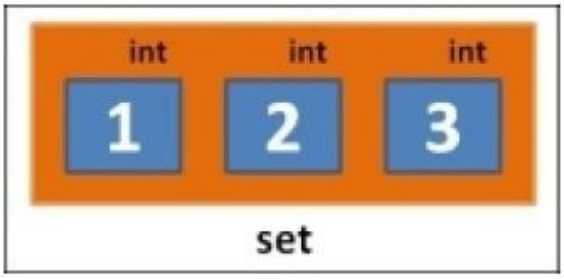
TSet<T>

Переменная TSet<int> хранит набор целых чисел. Переменная TSet<FString> хранит набор строковых значений. Главное отличие между TSet и TArray в том, что TSet не допускает дублирования, то есть все элементы внутри TSet гарантированно будут уникальными. Переменная TArray не препятствует дублированию элементов.

Чтобы добавить числа в TSet, просто вызовите Add. Посмотрите на пример следующего объявления:

```
TSet<int> set;
set.Add( 1 );
set.Add( 2 );
set.Add( 3 );
set.Add( 1 );// дублирование! Добавлено не будет
set.Add( 1 );// дублирование! Добавлено не будет
```

Вот как будет выглядеть TSet:



Дублирование, то есть повторный ввод одинаковых значений в TSet не будет допускаться. Обратите внимание, что введённые данные в TSet не пронумерованы, как это было в TArray. Вы не можете использовать квадратные скобки для доступа к элементам массива TSet.

Итерация TSet

Чтобы заглядывать в массив TSet вы должны использовать итератор. Вы не можете использовать запись квадратных скобок для доступа к элементам TSet:

```
int count = 0; // keep track of numerical index in set
for( TSet<int>::TIterator it = set.CreateIterator(); it; ++it )
{
   GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red, FString::FromInt( *it ) );
}
```

Пересечение TSet

У массива TSet есть две специальные функции, которых нет у переменной TArray. Пересечение двух массивов TSet, в целом это общие для них элементы. Если у нас есть два массива TSet, таких как X и У и мы сделаем их пересечение, то в результате появится третий, новый массив TSet, который будет содержать только общие для них элементы. Взгляните на следующий пример:

```
TSet<int> X;
X.Add( 1 );
X.Add( 2 );
X.Add( 3 );
TSet<int> Y;
Y.Add( 2 );
Y.Add( 4 );
Y.Add( 8 );
TSet<int> common = X.Intersect(Y); // Intersect – Пересекать. Результат 2
```

Из общих элементов массивов X и У, будет только 2.

Объединение TSet

Математически, объединение двух наборов, это когда вы вводите все элементы в один набор. Так как мы говорим о наборах здесь, то не будет никаких дубликатов.

Если мы возьмём наборы массивов X и У, из предыдущего примера, и создадим объединение, то у нас получится новый набор:

```
TSet<int> uni = X.Union(Y); // 1, 2, 3, 4, 8
```

Нахождение TSet

Вы можете определить, находится ли элемент в TSet или нет, применив для набора функцию-член Find(). TSet вернёт указатель на элемент в TSet, который соответствует вашему запросу, если он есть в TSet. Или вернёт NULL, если запрашиваемого вами элемента в TSet нет.

TMap<T, S>

TMap<T, S> создаёт таблицу видов в ОЗУ. TMap представляет карту ключей слева от значений с правой стороны. Вы можете представить TMap как таблице с двумя столбцами, с ключами в левом столбце и значениями в правом столбце.

Список предметов для инвентаря игрока

Скажем, мы захотели создать структуру данных C++, чтобы хранить список предметов для инвентаря игрока. В левом столбце таблицы (ключи), у нас будет FString для названия предметов. В правом столбце (значения), у нас будет int для количества этих предметов.

Предмет (Ключ)	Количество (Значение)
яблоки	4
пончики	12
мечи	1
щиты	2

Чтобы воплотить это в коде, мы просто пишем следующее:

```
TMap<FString, int> items;
items.Add( "яблоки", 4 );
items.Add( "пончики", 12 );
items.Add( "мечи", 1 );
items.Add( "щиты", 2 );
```

Как только вы создали ваш TMap, вы можете иметь доступ к значениям внутри TMap, используя квадратные скобки и передавая ключ в скобках. Например, в карте items - предметы, в предыдущем коде, items["яблоки"] это 4.

Подсказка

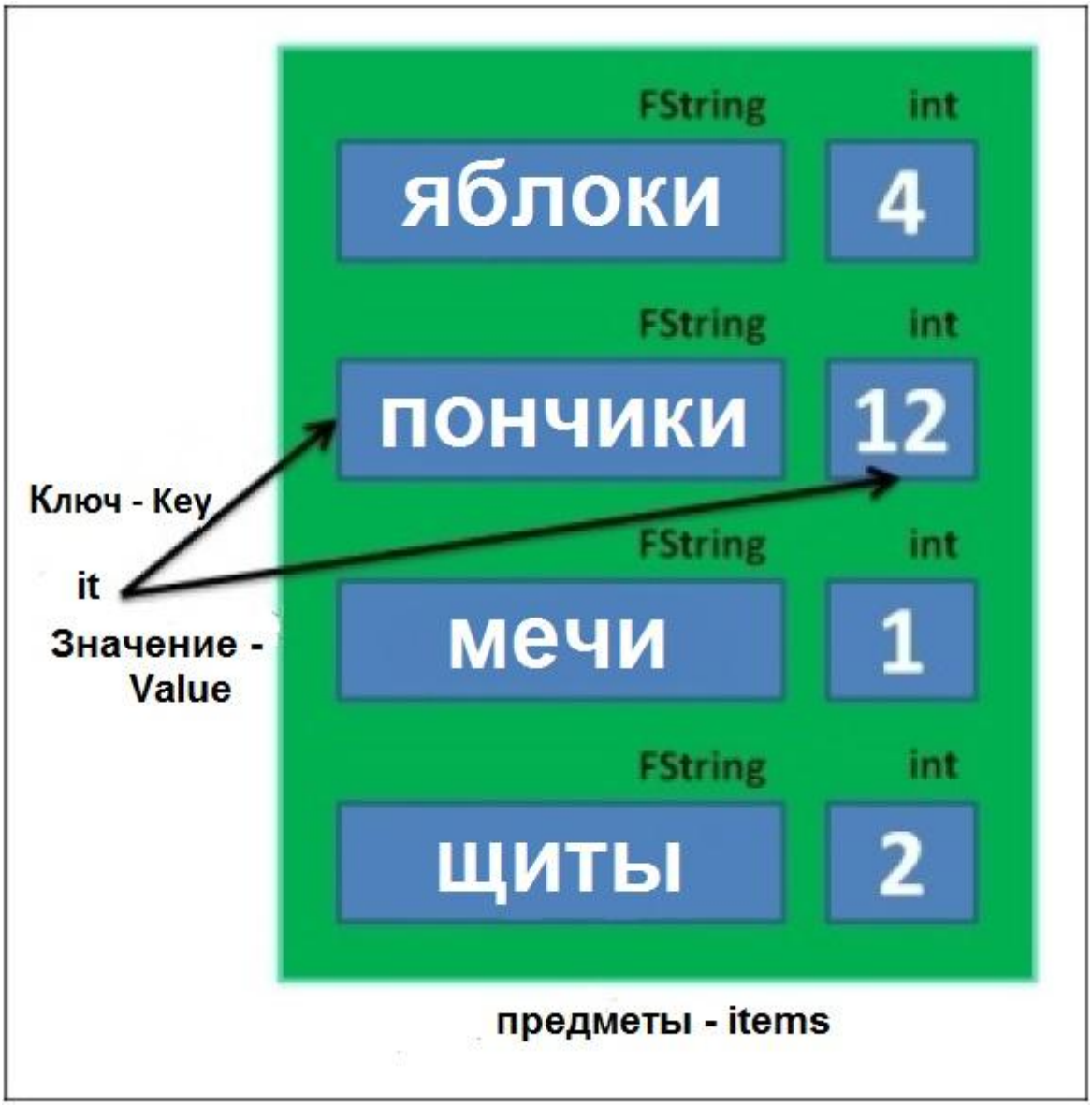
UE4 выйдет из строя, если вы используете квадратные, чтобы получить доступ к ключу, которого ещё нет в карте. Так что будьте осторожны! A STL C++ не выходит из строя, если вы делаете это.

Итерация TMap

Чтобы итерировать TMap, мы конечно используем итератор:

```
for( TMap<FString, int>::TIterator it = items.CreateIterator(); it; ++it )
{
    GEngine->AddOnScreenDebugMessage( count++, 30.f, FColor::Red, it->Key + FString(": ") +
    FString::FromInt( it->Value ) );
}
```

Итераторы TMap немного отличаются от итераторов TArray и TSet. Итератор TMap содержит и Ключ и Значение. Мы можем получить к ключу внутри TMap с it->Key и к значению с it->Value.



STL C++ версии часто используемых контейнеров

Я хочу рассказать о паре версий контейнеров STL C++. STL – это стандартная библиотека шаблонов (standard template library), которая работает с большинством компиляторов C++. Причина по которой я хочу рассказать об этих версиях STL, в том, что их поведение отличается от UE4 версий тех же контейнеров. В каких-то направлениях они ведут себя очень хорошо, но игровые программисты часто жалуются, что у STL есть проблемы с производительностью. В частности, я хочу затронуть STL контейнеры set и map.

Примечание

Если вам нравится интерфейс STL, но вы хотите лучшей производительности, то есть хорошо известная реализация библиотеки STL от Electronic Arts, называемая EASTL, которую вы можете использовать. Она предоставляет тот же функционал,

что и STL, но осуществлена с лучшей производительностью (в основном делая такие вещи как, удаление проверки границ). Она доступна на GitHub https://github.com/paulhodge/EASTL.

Набор STL C++

Набор – set C++ это связка предметов, которые уникальны и отсортированы. Хорошая характеристика STL set, то что он хранит набор элементов отсортированным. Быстрый и грязный способ отсортировать группу значений, это засунуть их в один набор. set позаботится о сортировке за вас.

Мы можем вернуться к простому приложению консоли C++ для использования наборов. Чтобы использовать набор STL C++, вам надо включить <set>, как показано здесь:

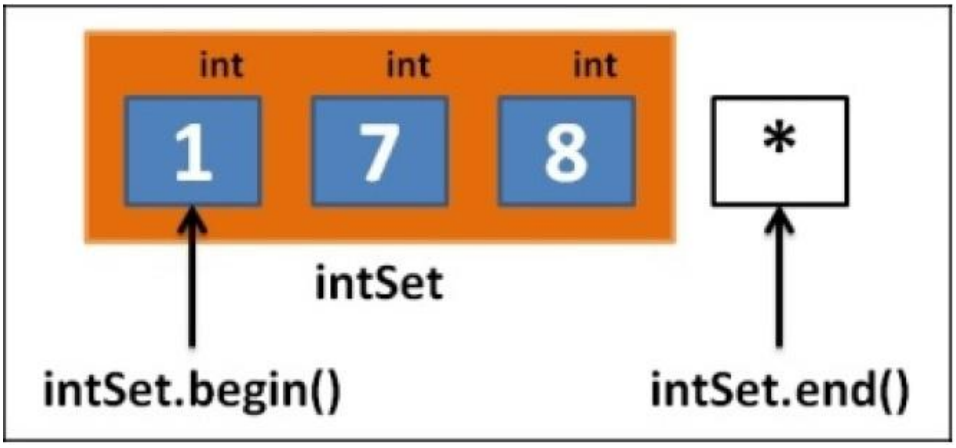
```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    set<int> intSet;
    intSet.insert( 7 );
    intSet.insert( 7 );
    intSet.insert( 8 );
    intSet.insert( 1 );

    for( set<int>::iterator it = intSet.begin(); it != intSet.end(); ++it )
    {
        cout << *it << endl;
    }
}
```

Вот вывод предыдущего кода:

```
1

7

8
```

Дубликат 7 отфильтрован, и элементы содержаться в возрастающем порядке внутри set. Способ, которым мы выполняем итерацию элементов контейнера STL, сходен с итерацией для массива UE4 TSet. Функция intSet.begin() возвращает итератор, который указывает в начало intSet.

Условие для остановки итератора, это когда it становится intSet.end(), и на самом деле это на одну позицию дальше конца набора, как показано на следующем изображении:



Нахождение элемента в <set>

Чтобы найти элемент в наборе STL, мы можем использовать функцию-член find(). Если пункт, который мы ищем, обнаруживается в наборе – set, то наш итератор указывает на элемент, который мы искали. Если пункта, который мы ищем, нет наборе – set, то мы получаем set.end(), как показано здесь:

```
set<int>::iterator it = intSet.find( 7 );
if( it != intSet.end() )
{
    // 7 было в intSet, и *it имеет своё значение
    cout << "Found " << *it << endl;
}
```

Упражнение

Попросите у пользователя набор из трёх неповторяющихся имён. Примите каждое имя, одно за одним, и затем выведите их в отсортированном порядке. Если пользователь повторяет имя, то попросите его другое имя, пока не будет три имени.

Решение

Решение предыдущего упражнения в следующем коде:

```
#include <iostream>
#include <string>
#include <set>
using namespace std;
int main()
{
    set<string> names;
    // пока у нас не будет 3 имён, продолжаем цикл
    while( names.size() < 3 )
    {
        cout << names.size() << " Введите имя " << endl;
        string name;
```

```
        cin >> name;
        names.insert( name ); // не надо вводить если уже есть,
    }
    // теперь выводим имена. Набор будет отсортирован в порядке
    for( set<string>::iterator it = names.begin(); it != names.end(); ++it )
    {
        cout << *it << endl;
    }
}
```

Карта STL C++

STL C++ объект map – карта во многом похож на объект UE4 TMap. Единственное, что делает map и не делает TMap, это поддерживает отсортированный порядок в карте. Сортировка стоит дополнительных затрат, но если вы хотите, чтобы ваша карта была отсортирована, то выбор STL версии может быть хорошим выбором.

Чтобы использовать C++ STL объект map, мы включаем <map>. В следующем примере программы, мы заполняем карту предметов парами ключ-значение (key-value):

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
    map<string, int> items;
    items.insert( make_pair( "apple", 12 ) );
    items.insert( make_pair( "orange", 1 ) );
    items.insert( make_pair( "banana", 3 ) );
    // мы можем также использовать квадратные скобки, чтобы вносить предметы в STL map
    items[ "kiwis" ] = 44;

    for( map<string, int>::iterator it = items.begin(); it != items.end(); ++it )
    {
        cout << "items[ " << it->first << " ] = " << it->second << endl;
    }
}
```

Вот вывод предыдущей программы:

```
items[ apple ] = 12
items[ banana ] = 3
items[ kiwis ] = 44
items[ orange ] = 1
```

Обратите внимание, что синтаксис итератора для map STL немного отличается от TMap. Доступ к ключу мы получаем используя it->first и к значению получаем доступ, используя it->second.

Обратите внимание, что C++ STL также предлагает от части синтаксический сахар для TMap. То есть вы можете использовать квадратные скобки для ввода в C++ STL map. Вы не можете использовать квадратные скобки для ввода в TMap.

Нахождение элемента в <map>

Вы можете выполнять поиск карты для пары <key,value> используя функцию-член карты STL find.

Упражнение

Попросите пользователя ввести пять предметов и их количество, в пустую карту – map. Выведите результаты в сортированном порядке.

Решение

Для решения предыдущего упражнения применяется следующий код:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
    map<string, int> items;
    cout << "Введите 5 предметов и их количество" << endl;
    while( items.size() < 5 )
    {
        cout << "Введите предмет" << endl;
        string item;
        cin >> item;
        cout << "Введите количество" << endl;
        int qty;
        cin >> qty;
        items[ item ] = qty; // сохраняем в карте, запись квадратных
        // скобок
    }

    for( map<string, int>::iterator it = items.begin(); it != items.end(); ++it )
    {
        cout << "items[ " << it->first << " ] = " << it->second << endl;
    }

}
```

В коде решения, мы начинаем с создания map<string, int> items ,чтобы хранить все предметы, которые мы собираемся принять. Спрашиваем пользователя о предметах и их количестве, затем мы сохраняем item в карте items, применяя запись квадратных скобок.

Выводы

Контейнеры UE4 и семья контейнеров C++ STL превосходны для хранения игровых данных. Зачастую проблемы программирования могут быть намного упрощены, если выбрать правильный тип контейнера.

В следующей главе, мы приступим к начальному программированию нашей игры, отслеживая, что держит игрок, и сохраняя эту информацию в объекте TMap.

Глава 10. Система Инвентаризации и Подбор Предметов

Мы хотим, чтобы наш игрок мог подбирать предметы в игровом мире. В этой главе, мы будем писать код и разрабатывать рюкзак, чтобы наш игрок хранил предметы. Мы будем отображать, что есть у игрока, когда пользователь нажимает клавишу I.

В качестве представления данных, мы можем использовать предметы TMap<FString, int>, пройденные в предыдущей главе, чтобы хранить наши предметы. Когда игрок подбирает предмет, мы добавляем этот предмет в карту. Если предмет уже в карте, мы просто повышаем значение количества подобранного предмета.

Объявляем рюкзак

Мы можем представить рюкзак игрока просто как предмет TMap<FString, int>. Чтобы позволить вашему игроку собирать предметы в мире, откройте файл Avatar.h и добавьте следующее объявление TMap:

```
class APickupItem; // forward declare the APickupItem class,
                   // since it will be "mentioned" in a member function
decl below
UCLASS()
class GOLDENEGG_API AAvatar : public ACharacter
{
   GENERATED_UCLASS_BODY()

   // Карта для рюкзака – backpack игрока
   TMap<FString, int> Backpack;

   // Значки для предметов в рюкзаке, просматриваются строкой
   TMap<FString, UTexture2D*> Icons;

   // Флаг выводящий нам Интерфейс Пользователя показывает
   bool inventoryShowing;
   // функция-член даущая аватару иметь предмет
   void Pickup( APickupItem *item );
   // ... остальная часть файла Avatar.h такая же как до этого
};
```

Предварительное объявление

Перед классом AAvatar, заметьте, что у нас есть предварительное объявление класса APickupItem. Дальнейшие объявления нужны в кодовом файле, когда класс затрагивается (такой как прототип функции APickupItem::Pickup(APickupItem *item);), но нет кода в файле, который использует объект этого типа внутри файла. Так как заголовочный файл Avatar.h не содержит выполняемый код, который

использует объект типа APickupItem, то предварительно объявление то, что нам нужно. Отсутствие предварительного объявления даст ошибку компилятора, так как компилятор не слышал бы о классе APickupItem до компилирования кода в классе AAvatar. Ошибка компилятора появится в объявлении APickupItem::Pickup(APickupItem*item); объявлении функции прототипа.

Мы объявили два объекта TMap в классе AAvatar. Объекты будут выглядеть, как показано в следующей таблице:

FString (имя)	Int (количество)	UTexture2D* (изображение)
GoldenEgg	2	
MetalDonut	1	
Cow	2	

В рюкзаке TMap, мы храним переменную FString предмета, который держит игрок. В карте Icons, мы храним единственную ссылку на изображение предмета, который держит игрок.
Во время визуализации, мы можем использовать две карты работающие вместе, чтобы просматривать и количество предметов имеющихся у игрока (в его карте Backpack), и ссылку ресурса текстуры этого предмета (в карте Icons). Следующий скриншот показывает, как будет выглядеть визуализация HUD:

Примечание

Обратите внимание, что мы также можем использовать массив struct с переменной FString и UTexture2D* в нём, вместо использования двух карт.

Например, мы можем держать TArray<Item> Backpack с помощью переменной struct, как показано в следующем коде:

```
struct Item
{
    FString name;
    int qty;
    UTexture2D* tex;
};
```

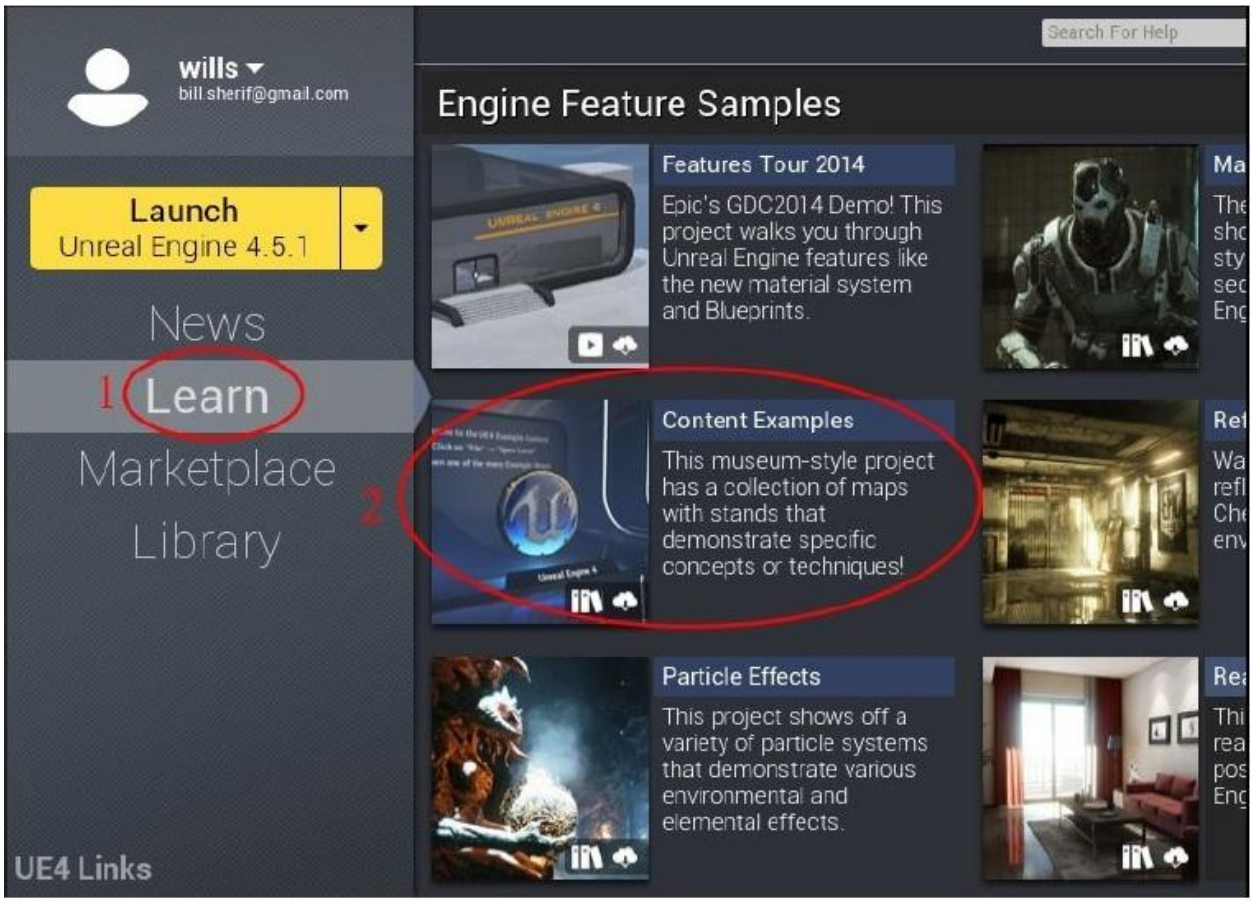
Затем, по мере того, как мы подбираем предметы, они будут добавлены в линейный массив. Однако подсчёт каждого предмета, который есть у нас в рюкзаке, потребует постоянную переоценку, посредством итерации через массив предметов, каждый раз, когда мы хотим видеть счёт. Например, чтобы увидеть, сколько расчёсок есть у вас, вам потребуется сделать передачу через весь массив. И это не так эффективно, как использование карты.

Импортирование ассетов

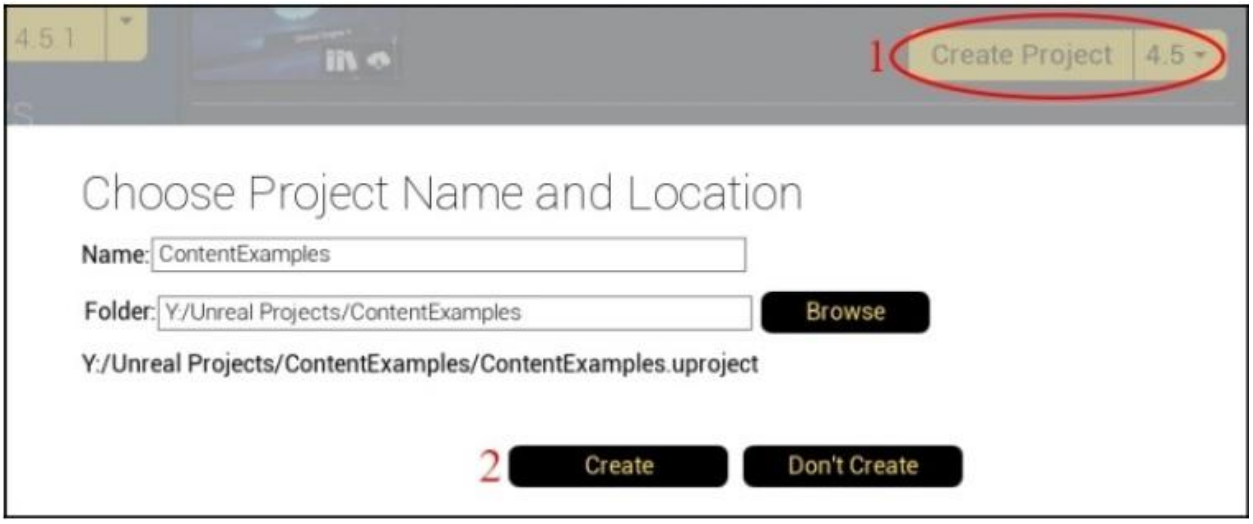
Вы должно быть заметили ассет (ресурс) **Cow** на предыдущем скриншоте, который не является частью стандартного набора ассетов предоставляемых UE4 в новом проекте. Чтобы использовать ассет **Cow** (корова), вам нужно импортировать корову из проекта **Content Examples** (образцы контента). Это стандартная процедура импортирования, которую использует UE4.

На следующем скриншоте, я обвёл процедуру для импортирования ассета **Cow**. Другие ассеты будут импортированы из других проектов в UE4, используя тот же метод. Выполните следующие шаги, чтобы импортировать ассет **Cow**:

1. Скачайте и откройте проект UE4 **Content Exampels**:

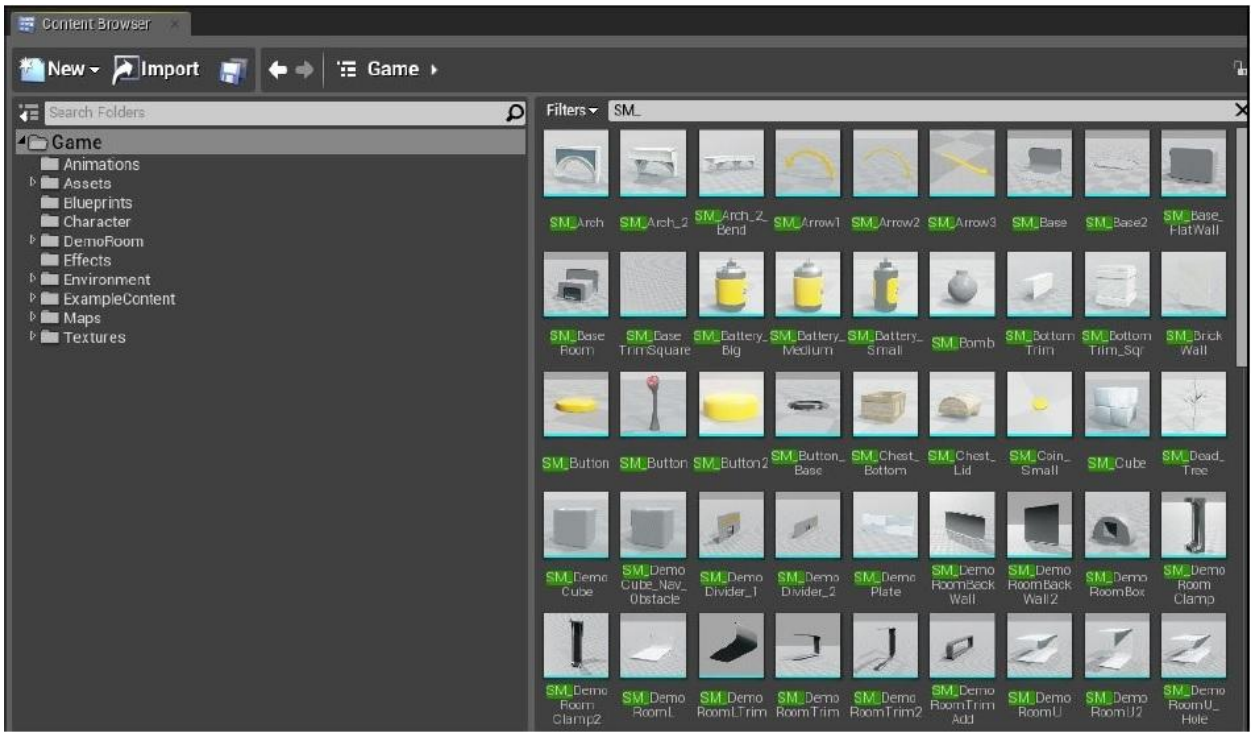


2. После скачивания **Content Examples**, откройте его и нажмите **Create Project**:



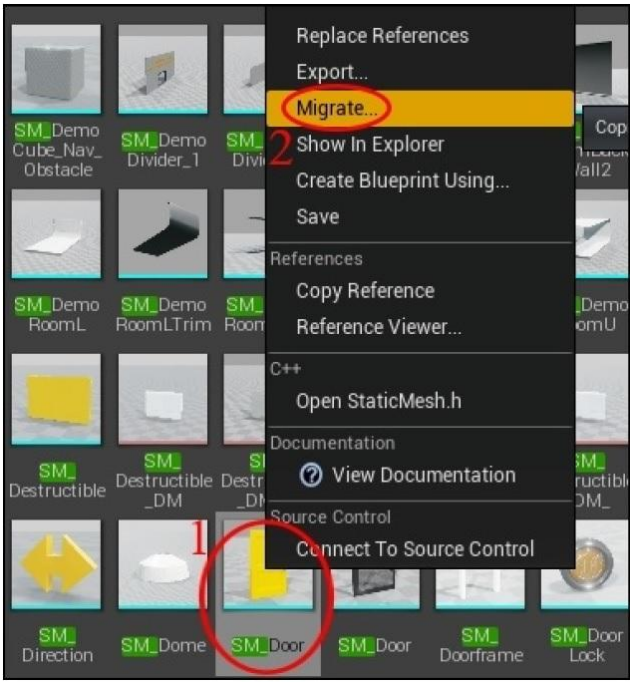
3. Далее, назовите папку, в которую вы поместите ваш ContentExamples и нажмите **Create**.

4. Откройте ваш проект ContentExamples из библиотеки. Просмотрите доступные ассеты в проекте, и найдите такой, что понравится вам. Поиск для SM_ поможет, так как все статичные сетки обычно начинаются с SM_ по соглашению.

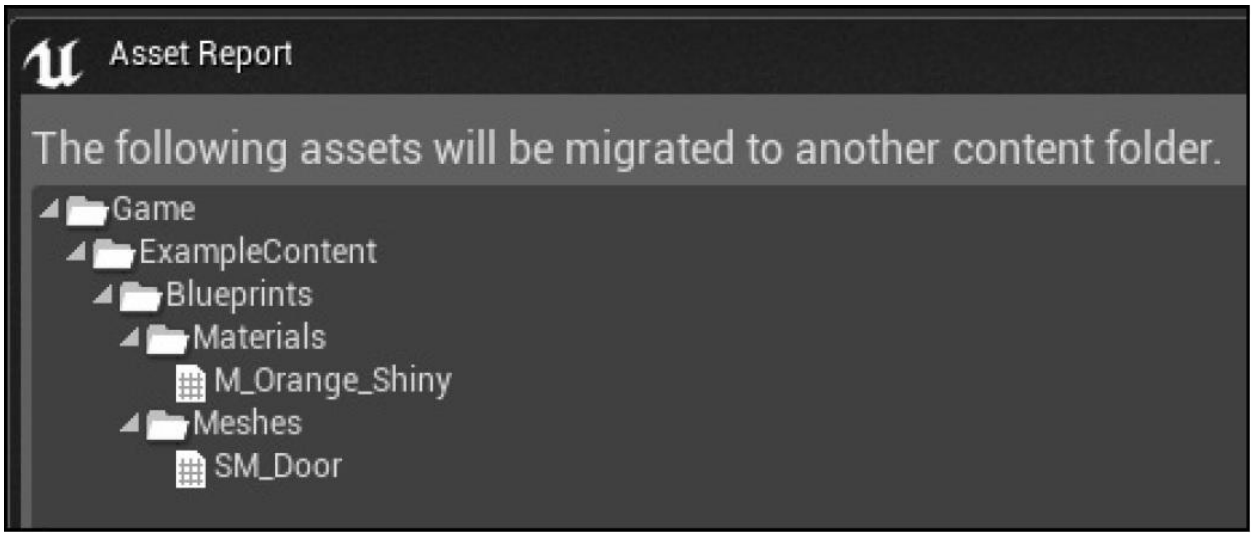


*Список статических сеток, все начинаются с SM_*

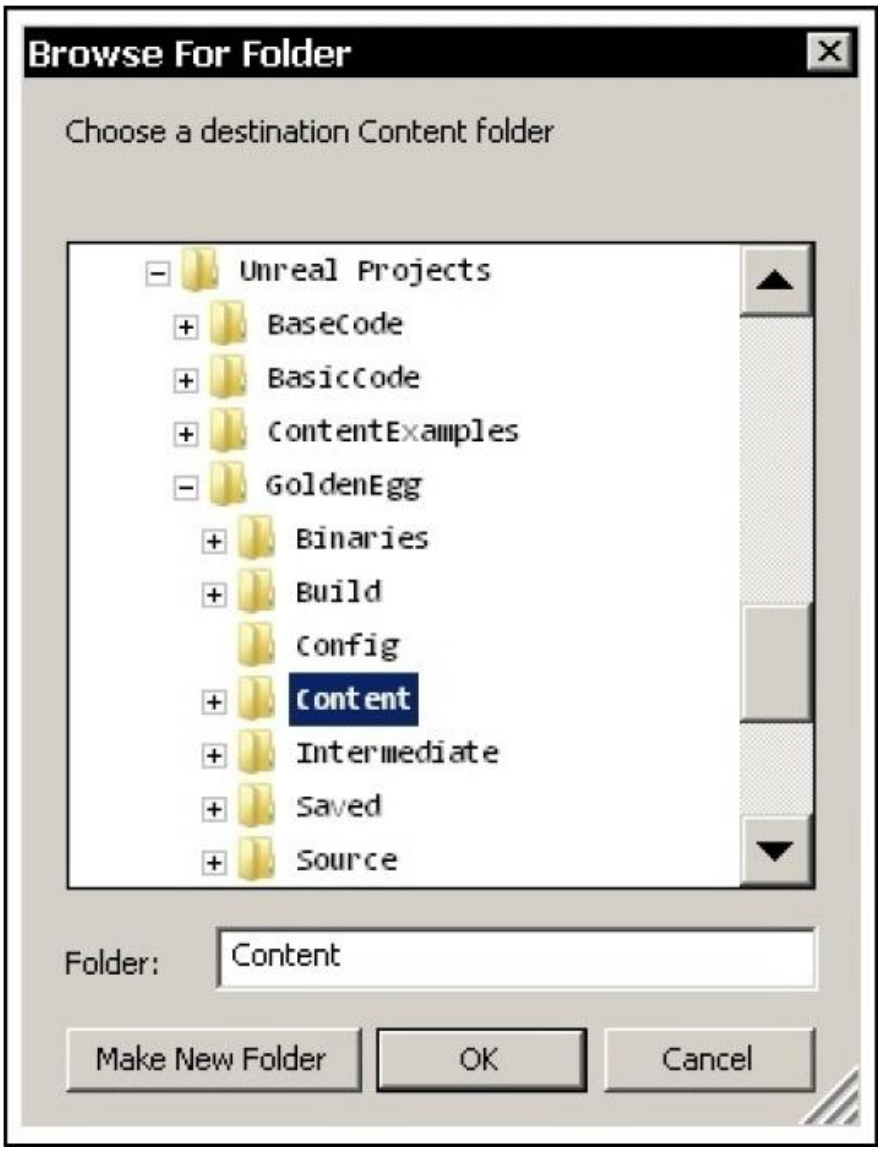
5. Когда вы подберёте ассет, который вам понравился, импортируйте его в ваш проект, щёлкнув правой кнопкой мыши по ассету, а затем нажмите **Migrate…**:



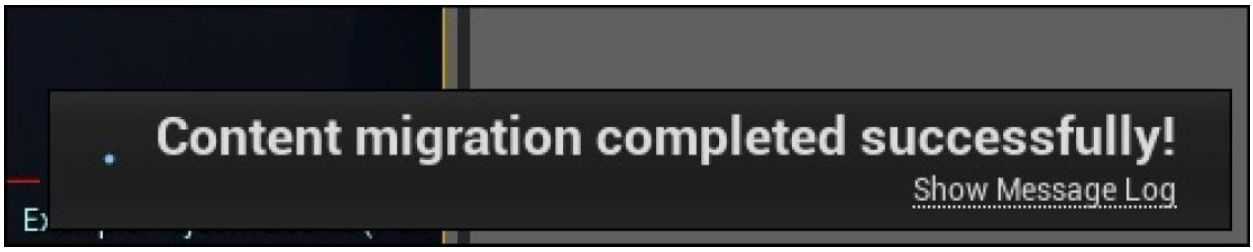
6. Нажмите **OK** в диалоговом окне **Asset Report**:



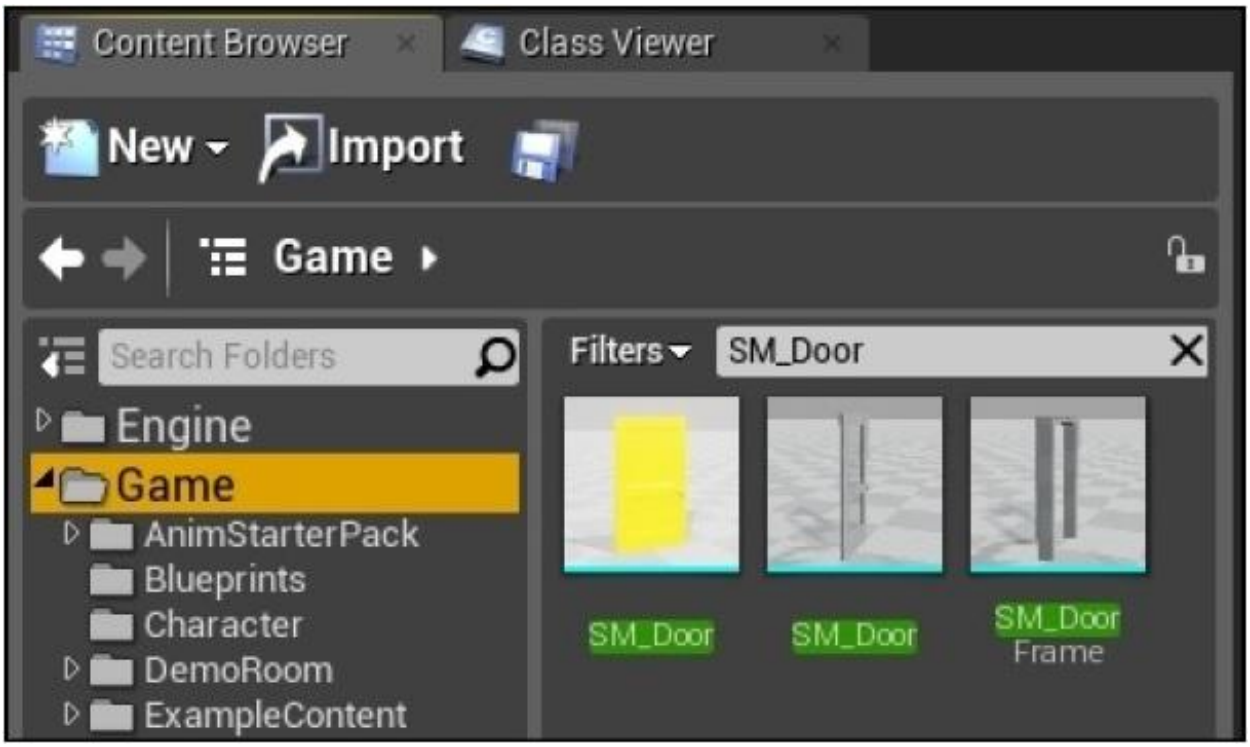
7. Выберите папку **Content** из вашего проекта, в которую вы хотите добавить файл **SM_Door**. В моём случае, я хочу добавить его в Y:/Unreal Projects/GoldenEgg/Content, как показано на следующем скриншоте:



8. Если импорт завершился успешно, вы увидите следующее сообщение:



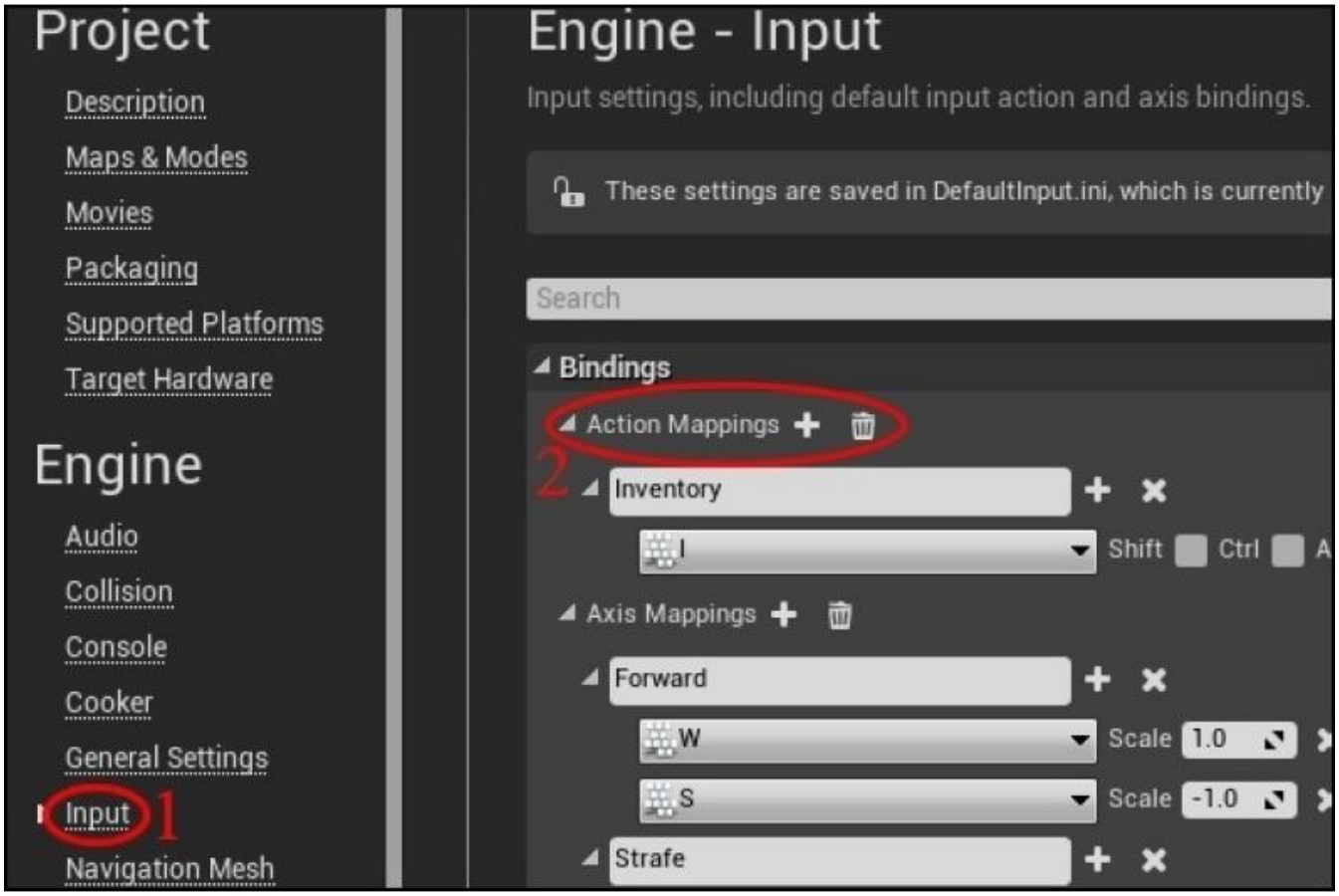
9. Как только вы импортируете свой ассет, вы увидите, что он появится в вашем браузере ассетов, в вашем проекте:



Затем вы можете спокойно использовать ассет в вашем проекте.

Прикрепляем действия карты к клавише

Нам нужно прикрепить клавишу, чтобы активировать отображение инвентаря игрока. В редакторе UE4, добавьте **Action Mappings +** названный Inventory и назначьте его к клавише I:



В файле Avatar.h, добавьте функцию-член, запускаемую, когда нужно отображать инвентарь игрока:

```
void ToggleInventory();
```

В файле Avatar.cpp, осуществите функцию ToggleInventory(), как показано в следующем коде:

```
void AAvatar::ToggleInventory()
{
   if( GEngine )
   {
      GEngine->AddOnScreenDebugMessage( 0, 5.f, FColor::Red, "Showinginventory…" );
   }
}
```

Затем, соедините действие “Inventory” с AAvatar::ToggleInventory() в SetupPlayerInputComponent():

```
void AAvatar::SetupPlayerInputComponent(class UInputComponent*InputComponent)
{
   InputComponent->BindAction( "Inventory", IE_Pressed, this,&AAvatar::ToggleInventory );
   // остальная часть SetupPlayerInputComponent такая же как до этого
}
```

Базовый класс PickupItem

Мы должны определить, как подбор предметов будет выглядеть в коде. Каждый подбираемый предмет будет происходить от общего базового класса. Давайте сейчас сконструируем базовый класс для класса PickupItem (подбор предметов).

Базовый класс PickupItem должен наследоваться от класса AActor. Подобно тому, как мы создали множество схем (blueprint) NPC от базового класса NPC, мы можем создать и множество схем (blueprint) PickupItem от единственного базового класса PickpItem, как показано на следующем скриншоте:

Как только вы создадите класс PickupItem, откройте его код в Visual Studio.

Классу APickupItem понадобится несколько элементов:

- Переменная FString для имени подбираемого предмета
- Переменная int32 для количества подбираемого предмета
- Переменная USphereComponent для сферы которой вы будете сталкиваться, чтобы подобрать предмет
- Переменная UStaticMeshComponent, чтобы содержать сетку
- Переменная UTexture2D для значка, который представляет предмет
- Указатель для HUD (который мы инициализируем позже)

Вот как выглядит код в APickupItem.h:

```
UCLASS()
class GOLDENEGG_API APickupItem : public AActor
{
   GENERATED_UCLASS_BODY()

   // Имя предмета, который вы заполучаете
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
   FString Name;

   // Как много вы заполучаете
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
   int32 Quantity;

   // сфера, с которой вы сталкиваетесь, что подобрать предмет
   UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Item)
   TSubobjectPtr<USphereComponent> ProxSphere;
```

```
    // Сетка предмета
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Item)
    TSubobjectPtr<UStaticMeshComponent> Mesh;

    // Значок, который представляет объект в интерфейсе пользователя/на поверхности
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
    UTexture2D* Icon;

    // Когда что-то попадает в ProxSphere, эта функция запускается
    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
    bFromSweep, const FHitResult & SweepResult );
};
```

Суть всех этих объявлений UPROPERTY(), в том чтобы сделать APickupItem полностью конфигурируемым посредством схем (blueprint). Например, предметы в категории **Pickup**, в редакторе blueprint будут отображены следующим образом:

В файле PickupItem.cpp, мы завершаем конструктор для класса APickupItem, как показано в следующем коде:

```
APickupItem::APickupItem(const class FPostConstructInitializeProperties&
PCIP) : Super(PCIP)
{
    Name = "UNKNOWN ITEM"; .// “НЕИЗВЕСТНЫЙ ПРЕДМЕТ”
    Quantity = 0;

    // Задаём объект Unreal
    ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this, TEXT("ProxSphere"));
    Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this, TEXT("Mesh"));

    // делаем корневым объектом Mesh
    RootComponent = Mesh;
    Mesh->SetSimulatePhysics(true);

    // Пишем код для запуска APickupItem::Prox(), когда эта
    // сфера близости объекта пересекается с другим актором.
    ProxSphere->OnComponentBeginOverlap.AddDynamic(this,
    &APickupItem::Prox);
    ProxSphere->AttachTo( Mesh ); // очень важно!
}
```

На первых двух строках мы выполняем установку значений имени (Name) и количества (Quantity), которые должны предстать перед разработчиком игры как изначально заданные. Я использовал заглавные буквы, чтобы разработчик мог видеть, что до этого переменная ещё не устанавливалась.

Затем, мы присваиваем начальные значения компонентам ProxSphere и Mesh используя PCIP.CreateDefaultSubobject. Недавно инициализированные объекты могут иметь некоторые из своих начально заданных значений, но Mesh (сетка) будет стартовать пустой. Вам понадобится загрузить саму сетку позже, в блупринтах.

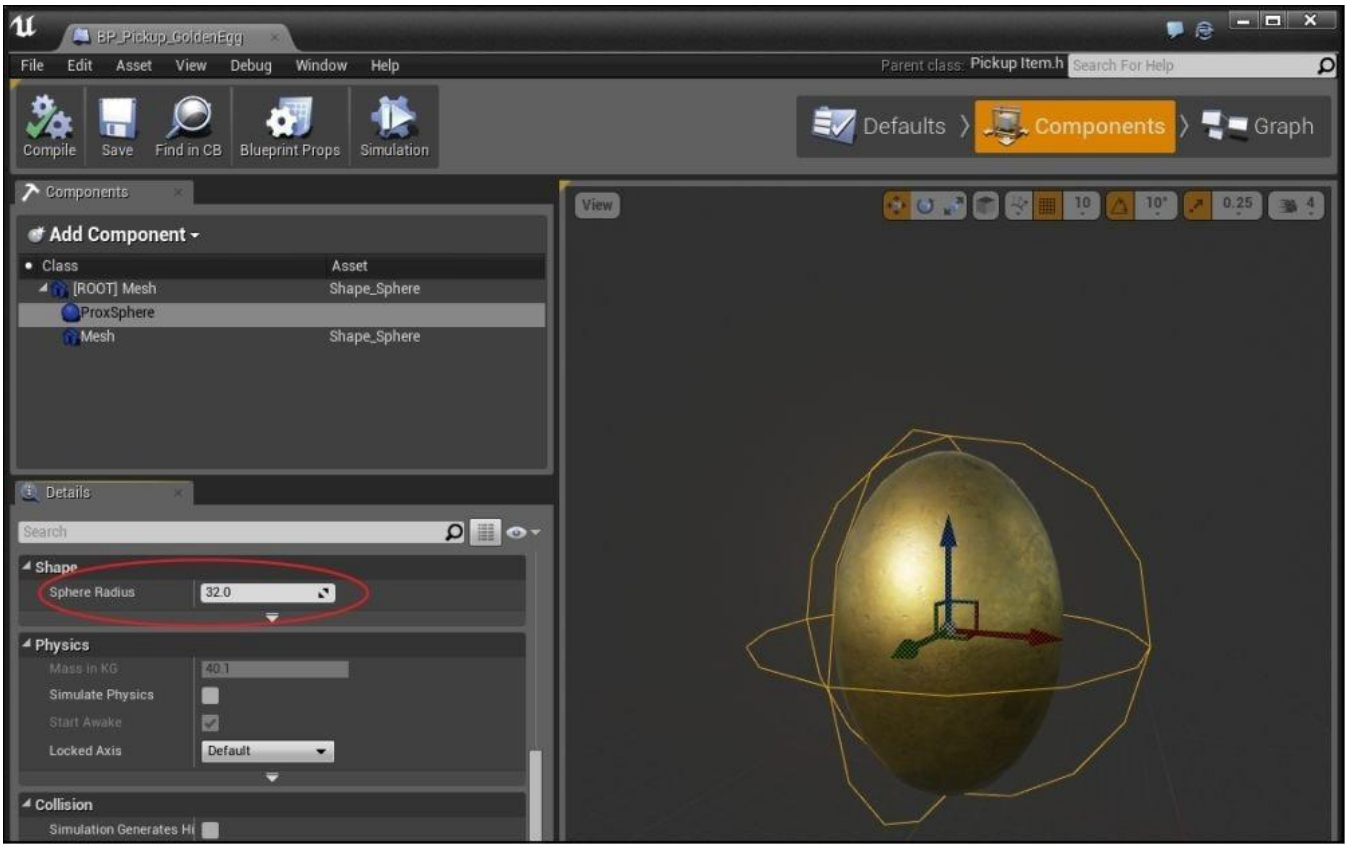
Сетку мы устанавливаем, чтобы симулировать реалистичную физику, так что если предметы бросать или двигать, то они будут отскакивать и катиться или переворачиваться. Обратите особое внимание на строку ProxSphere->AttachTo(Mesh). Это трока говорит вам убедиться, что компонент подбираемого предмета ProxSphere прикреплён к корневому компоненту Mesh. Это означает, что когда сетка двигается в уровне, то ProxSphere следует за ней. Если вы забудете этот шаг (или если вы сделали его как то по другому), то ProxSphere не будет следовать за сеткой, когда она подпрыгивает.

Корневой компонент

В предыдущем коде мы назначили RootComponent (корневой компонент) класса APickupItem объекту Mesh. Элемент RootComponent является частью базового класса AActor, так что AActor и его производные имеют корневой компонент. Корневой компонент в основном подразумевает то, что он будет ядром объекта, а также будет определять как вы сталкиваетесь с объектом. Объект RootComponent определён в файле Actor.h, как показано в следующем коде:

```
/**
 * Примитив столкновения, который определяет преобразование (местоположение, вращение, масштаб) этого актора.
 */
UPROPERTY()
class USceneComponent* RootComponent;
```

Таким образом, создатели UE4 намеревались сделать так, чтобы RootComponent всегда был ссылкой на примитив столкновения. Иногда примитив столкновения может быть в форме капсулы, в остальных случаях в форме сферы или даже прямоугольной формы, или может быть произвольной формы, как в нашем случае, с сеткой. Это редкий случай, когда персонаж должен иметь корневой компонент прямоугольной формы, потому что углы прямоугольника могут задевать стены. Обычно предпочитаемы круглые формы. Свойство RootComponent показывается в блупринтах, где вы можете просматривать его и манипулировать им.



Вы можете редактировать корневой компонент ProxSphere в блупринтах, как только вы создадите блупринт основанный на классе PickupItem

И наконец, функция Prox_Implementation осуществляется следующим образом:

```
void APickupItem::Prox_Implementation( AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult & SweepResult )
{
    // если актор, с которым произошло пересечение, НЕ является игроком,
    // вы просто должны вернуться
    if( Cast<AAvatar>( OtherActor ) == nullptr )
    {
        return;
    }

    // Получаем ссылку на аватар игрока, чтобы дать ему предмет
    AAvatar *avatar = Cast<AAvatar>( UGameplayStatics::GetPlayerPawn( GetWorld(), 0 ) );

    // Позволяем игроку подбирать предмет
    // Обратите на ключевое слово this!
    // Это то, как _this_ Pickup может ссылаться на само себя.
    avatar->Pickup( this );

    // Получаем ссылку на контроллер
    APlayerController* PController = GetWorld()- >GetFirstPlayerController();

    // Получаем ссылку на HUD из контроллера
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->addMessage( Message( Icon, FString("Picked up ") + FString::FromInt(Quantity) +
    FString(" ") + Name, 5.f, FColor::White, FColor::Black ) );
```

```
    Destroy();
}
```

Вот пара очень важных подсказок: первая, нам надо получить доступ к паре *глобальных функций*, чтобы получить доступ к объекту, который нам нужен. Есть три главных объекта, к которым у нас будет доступ через эти функции, которые управляют HUD: контроллер (APlayerController), HUD (AMyHUD), и сам игрок (AAvatar). Есть только один из каждого из этих типов объектов в экземпляре игры. UE4 делает нахождение их лёгким.

Получаем аватар

Объект класса player можно найти в любое время, из любого места в коде, просто вызвав следующий код:

```
AAvatar *avatar = Cast<AAvatar>( UGameplayStatics::GetPlayerPawn( GetWorld(), 0 ) );
```

Затем мы передаём ему предмет, вызывая функцию AAvatar::Pickup() определённую ранее.

Так как объект PlayerPawn на самом деле экземпляр AAvatar, мы приводим результат к классу AAvatar, используя команду Cast<AAvatar>. Семья функций UGameplayStatics доступна повсюду в вашем коде, это глобальные функции.

Получаем контроллер игрока

Также можно получать контроллер игрока из *суперглобальных*:

```
APlayerController* PController = GetWorld()->GetFirstPlayerController();
```

Функция GetWorld() определена в базовом классе UObject. Так как все объекты UE4 происходят от UObject, то любой объект в игре имеет доступ к объекту world.

Получаем HUD

Хотя эта организация и может показаться странной поначалу, HUD на самом деле прикреплён к контроллеру игрока. Вы можете взять HUD следующим образом:

```
AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
```

Мы приводим объект HUD, так как ранее мы установили HUD экземпляром AMyHUD в блупринтах. Так как мы будем часто использовать HUD, мы можем держать постоянный указатель на HUD внутри нашего класса APickupItem. И это мы обсудим далее.

Далее мы осуществляем AAvatar::Pickup, что добавляет объект типа APickupItem в рюкзак аватара:

```
void AAvatar::Pickup( APickupItem *item )
{
   if( Backpack.Find( item->Name ) )
   {
      // предмет уже был в рюкзаке… увеличиваем его количество
      Backpack[ item->Name ] += item->Quantity;
   }
   else
   {
      // предмета не было в рюкзаке, сейчас добавляем его туда
      Backpack.Add(item->Name, item->Quantity);
      // записываем ссылку на текстуру, когда предмет взят первый раз
      Icons.Add(item->Name, item->Icon);
   }
}
```

В предыдущем коде, мы проверяем, есть ли уже подбираемый предмет у игрока. Если есть, мы увеличиваем его количество. Если нет, то мы добавляем его и в рюкзак, и в карту Icons (значки).

Чтобы добавить подбираемый предмет, используйте следующий код:

```
avatar->Pickup( this );
```

APickupItem::Prox_Implementation это способ вызвать эту функцию-член.

Теперь, нам нужно отобразить содержимое нашего рюкзака на HUD, когда игрок нажимает I.

Изображаем инвентарь игрока

Экран инвентаря, в таких играх как Diablo наделён всплывающими окнами в организованной сетке, со значками предметов, которые вы подобрали. В UE4 мы можем достичь такой тип поведения.

Есть несколько методов для изображения интерфейса пользователя (User Interface - UI) в UE4. Наиболее основной метод, просто использовать вызов HUD::DrawTexture(). Другой способ использовать Slate. Ещё один способ использовать новейшую функциональность пользовательского интерфейса UE4: дизайнер Unreal Motion Graphics (UMG).

Slate использует синтаксис объявлений, чтобы выкладывать элементы UI в C++. Slate наилучшим образом подходит для меню и системы оценки. UMG ново в UE 4.5 и использует сильно основанный на blueprint рабочий процесс. Поскольку мы фокусируемся здесь на упражнениях использующих код C++, мы остановимся на осуществлении HUD::DrawTexture(). Это значит, что нам будет нужно управлять всеми данными, которые связанны с инвентаризацией в нашем коде.

Используем HUD::DrawTexture()

Мы дойдём до этого в два шага. Первый шаг, помещать содержимое нашего инвентаря в HUD, когда пользователь нажимает клавишу I. Второй шаг, собственно визуализировать значки на HUD в форме сетки.

Чтобы содержать всю информацию о том, как графический элемент (widget) может быть визуализирован, мы объявляем простую структуру для содержания информации касающейся того, какой значок она использует, это текущая позиция и текущий размер.

Вот как выглядят структуры Icon (значок) и Widget (графический элемент):

```
struct Icon
{
    FString name;
    UTexture2D* tex;
    Icon(){ name = "UNKNOWN ICON"; tex = 0; }
    Icon( FString& iName, UTexture2D* iTex )
    {
      name = iName;
      tex = iTex;
    }
};


struct Widget
{
    Icon icon;
    FVector2D pos, size;
    Widget(Icon iicon)
    {
      icon = iicon;
    }
    float left(){ return pos.X; }
    float right(){ return pos.X + size.X; }
    float top(){ return pos.Y; }
    float bottom(){ return pos.Y + size.Y; }
};
```

Вы можете добавить эти объявления структур вверху MyHUD.h, либо вы можете добавить их отдельный файл и включать этот файл везде, где эти структуры используются.

Обратите внимание на четыре функции-члены в структуре Widget, функции для перемещения влево - left(), вправо - right(), вверх – top() и вниз – bottom() по графическому элементу. Мы будем использовать их позже для определения, находится ли место щелчка мыши в окне.

Далее, мы объявляем функцию, которая будет визуализировать графические элементы на экране, в классе AMyHUD:

```
void AMyHUD::DrawWidgets()
{
   for( int c = 0; c < widgets.Num(); c++ )
   {
     DrawTexture( widgets[c].icon.tex, widgets[c].pos.X, widgets[c].pos.Y, widgets[c].size.X,
     widgets[c].size.Y, 0, 0, 1, 1 );
     DrawText( widgets[c].icon.name, FLinearColor::Yellow, widgets[c].pos.X, widgets[c].pos.Y,
     hudFont, .6f, false );
   }
}
```

Вызов функции DrawWidgets() должен быть добавлен к функции DrawHUD():

```
void AMyHUD::DrawHUD()
{
   Super::DrawHUD();
   // dims существуют только здесь в stock переменной Canvas
   // Обновите их, чтобы использовать в addWidget()
   dims.X = Canvas->SizeX;
   dims.Y = Canvas->SizeY;
   DrawMessages();
   DrawWidgets();
}
```

Далее, мы заполним функцию ToggleInventory() (ПереключатьИнвентарь). Вот функция, которая запускается, когда пользователь нажимает I:

```
void AAvatar::ToggleInventory()
{
   // Получаем контроллер и HUD
   APlayerController* PController = GetWorld()- >GetFirstPlayerController();
   AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );

   // Если инвентарь отображается, то прекращаем отображать его.
   if( inventoryShowing )
   {
     hud->clearWidgets();
     inventoryShowing = false;
     PController->bShowMouseCursor = false;
     return;
   }

   // Иначе отображаем инвентарь игрока
   inventoryShowing = true;
   PController->bShowMouseCursor = true;
   for( TMap<FString,int>::TIterator it = Backpack.CreateIterator(); it; ++it )
   {
     // Комбинируем имя предмета с количеством, то есть Cow x 5
     FString fs = it->Key + FString::Printf( TEXT(" x %d"), it- >Value );
     UTexture2D* tex;
     if( Icons.Find( it->Key ) )
     tex = Icons[it->Key];
     hud->addWidget( Widget( Icon( fs, tex ) ) );
   }
}
```

Чтобы компилировать предыдущий код, нам нужно добавить функцию в AMyHUD:

```
void AMyHUD::addWidget( Widget widget )
{
   // находим положение графического элемента, расположенного в сетке.
   // изображаем значки...
   FVector2D start( 200, 200 ), pad( 12, 12 );
   widget.size = FVector2D( 100, 100 );
   widget.pos = start;
   // вычисляем положение здесь
   for( int c = 0; c < widgets.Num(); c++ )
   {
     // Немного смещаем положение вправо.
     widget.pos.X += widget.size.X + pad.X;
     // Если справа больше нет места, то
     // переходим на следующую строку
     if( widget.pos.X + widget.size.X > dims.X )
     {
       widget.pos.X = start.X;
       widget.pos.Y += widget.size.Y + pad.Y;
     }
   }
   widgets.Add( widget );
}
```

Мы продолжаем применять переменную типа Boolean в inventoryShowing, чтобы сообщать нам отображается ли сейчас инвентарь или нет. Когда инвентарь показан, мы также показываем мышь, чтобы пользователь знал на, что он нажимает. Также когда отображается инвентарь, свободно движение игрока невозможно. Самый лёгкий способ отключить свободное движение игрока, просто вернуться из функции движения, до самого движения. Следующий код даёт пример:

```
void AAvatar::Yaw( float amount )
{
   if( inventoryShowing )
   {
     return; // когда мой инвентарь показывается,
             // игрок не может двигаться
   }
   AddControllerYawInput(200.f*amount * GetWorld()- >GetDeltaSeconds());
}
```

Упражнение

Проверьте каждую функцию движения с коротким циклом возвращения if(inventoryShowing) { return;}.

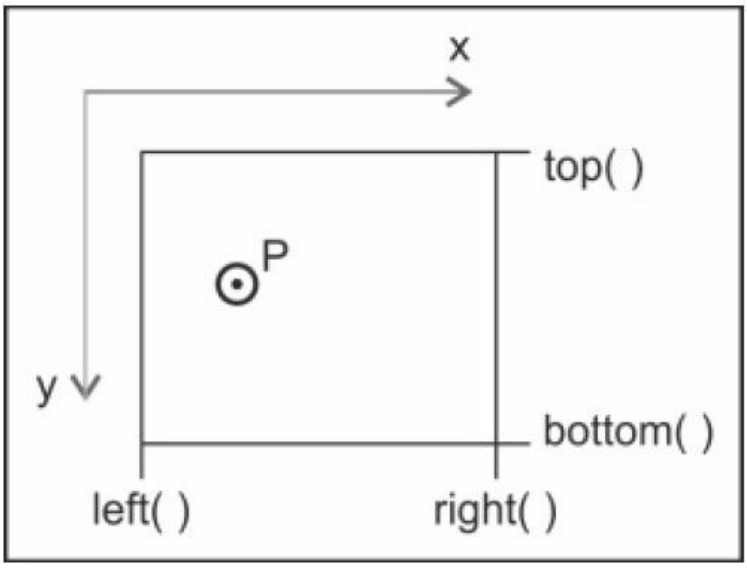
Определение щелчка в инвентарной системе

Мы можем определить кликает ли кто-нибудь по нашим предметам инвентаря, просто выполнив проверку указателя в прямоугольнике. Тест указателя в прямоугольнике, выполняется проверкой указателя в контенте прямоугольника.

Добавьте следующую функцию-член в struct Widget:

```
struct Widget
{
    // .. остальная часть структуры такая же как прежде ..
    bool hit( FVector2D p )
    {
      // +---+ top (0)
      // | |
      // +---+ bottom (2) (bottom > top)
      // L R
      return p.X > left() && p.X < right() && p.Y > top() && p.Y < bottom();
    }
};
```

Тест указателя в прямоугольнике выглядит следующим образом:



Итак, нажатие происходит, если p.X выполняет следующие условия:

- Справа от left() (p.X > left())
- Слева от right() (p.X < right())
- Снизу от top() (p.Y > top())
- Сверху от bottom() (p.Y < bottom())

Запомните, что в UE4 (и визуализация UI в целом) ось *y* инвертирована. Другими словами, ось *y* в UE4 идёт вниз. Это означает, что top() меньше чем bottom(), так как начало отсчёта координат (точка (0,0)) находится в верхнем левом углу экрана.

Перетаскивание элементов

Мы можем легко перетаскивать элементы. Первый шаг, чтобы сделать возможным перетаскивание, это отклик на нажатие левой кнопки мыши. Во первых, мы напишем функцию, для выполнения когда нажата левая кнопка мыши. В файле Avatar.h, добавьте следующий прототип к объявлению класса:

```
void MouseClicked();
```

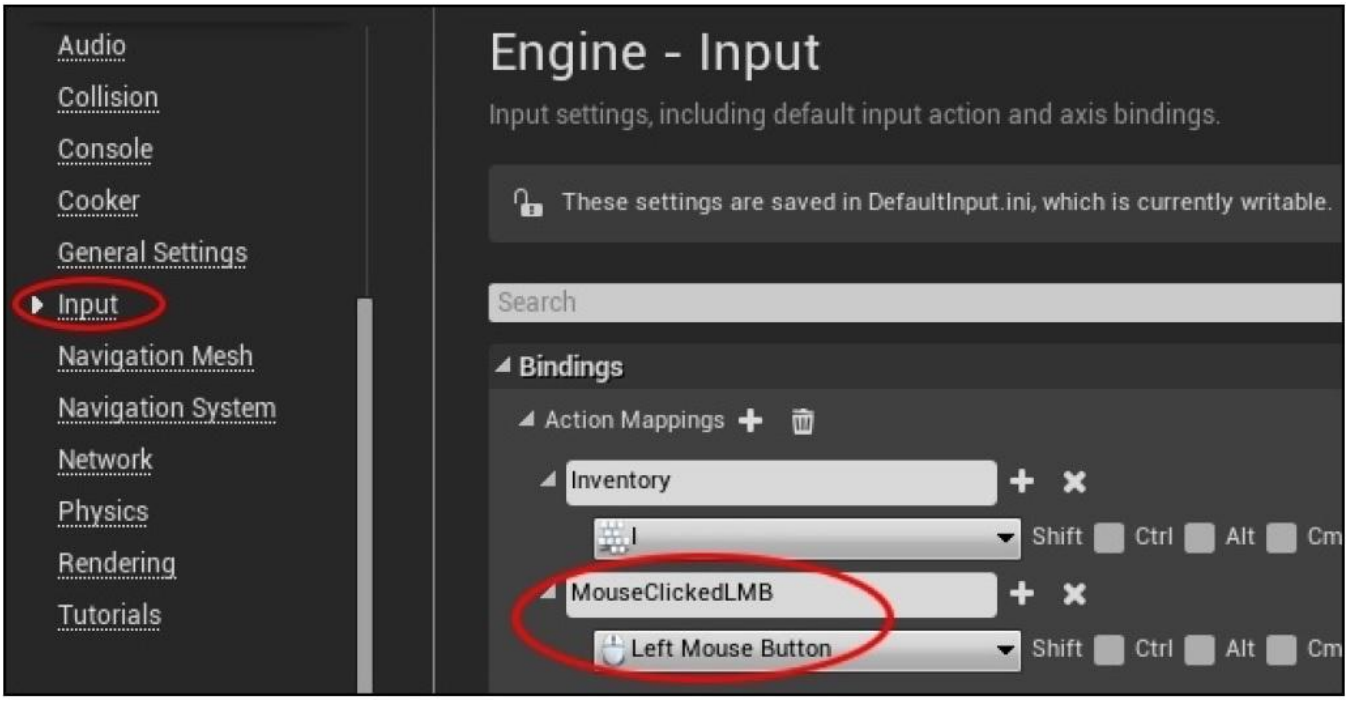
В файле Avatar.cpp, мы можем прикрепить функцию для выполнения по щелчку мыши и передачи запроса щелчка в HUD, следующим образом:

```
void AAvatar::MouseClicked()
{
    APlayerController* PController = GetWorld()- >GetFirstPlayerController();
    AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
    hud->MouseClicked();
}
```

Затем в AAvatar::SetupPlayerInputComponent, нам нужно добавить наш ответчик:

```
InputComponent->BindAction( "MouseClickedLMB", IE_Pressed, this, &AAvatar::MouseClicked );
```

Следующий скриншот демонстрирует, как вы можете добавить визуализацию:



Добавьте элемент в класс AMyHUD:

```
Widget* heldWidget; // держит последний затронутый Widget в памяти
```

Далее, в AMyHUD::MouseClicked(),мы начинаем выполнять поиск нажатия по Widget:

```
void AMyHUD::MouseClicked()
{
   FVector2D mouse;
   PController->GetMousePosition( mouse.X, mouse.Y );
   heldWidget = NULL; // очищает последний содержанный графический элемент
   // смотрим если положение щелчка мыши xy попадает по какому-нибудь элементу
   for( int c = 0; c < widgets.Num(); c++ )
   {
      if( widgets[c].hit( mouse ) )
      {
         heldWidget = &widgets[c];// сохраняем графический элемент
         return;                  // прекращаем поиск
      }
   }
}
```

В функции AMyHUD::MouseClicked, мы проходим цикл по всем графическим элементам, которые находятся на экране, и проверяем совпадение с текущим положением мыши. Вы можете получить текущее положение мыши от контроллера, в любое время, просто посмотрев PController->GetMousePosition().

Каждый графический элемент (widget) проверяется на текущее положение мыши, и тот элемент, на котором нажата кнопка мыши, будет перемещаться, когда перетаскивается мышь. Как только мы определим, по какому элементу был щёлчок, мы можем остановить проверку, так что мы получаем значение return от функции MouseClicked().

Однако нажать на графический элемент не достаточно. Нам нужно тащить нажатый элемент, когда двигается мышь. Для этого нам нужно осуществить функцию MouseMoved() в AMyHUD:

```
void AMyHUD::MouseMoved()
{
   static FVector2D lastMouse;
   FVector2D thisMouse, dMouse;
   PController->GetMousePosition( thisMouse.X, thisMouse.Y );
   dMouse = thisMouse - lastMouse;
   // Смотрим если левая кнопка мыши удерживается зажатой
   // более чем 0 секунд. Если она зажата,
   // то можно начать перетаскивание.
   float time = PController->GetInputKeyTimeDown( EKeys::LeftMouseButton );
   if( time > 0.f && heldWidget )
   {
      // мышь зажата.
      // двигаем графический элемент, изменяя положение amt
      heldWidget->pos.X += dMouse.X;
      heldWidget->pos.Y += dMouse.Y; // y inverted
   }
   lastMouse = thisMouse;
}
```

Не забудьте включить объявление в файл MyHUD.h.

Функция перетаскивания просматривает разницу в положения мыши между последним кадром и этим кадром, и двигает выбранный элемент на этот объём. Переменная static (глобальная с локальной областью действия) используется, чтобы запоминать lastMouse положение (последнее положение мыши) между вызовами функции MouseMoved().

Как мы можем связать движение мыши с запущенной функцией MouseMoved() в AMyHUD? Если вы помните, мы уже связывали движение мыши в классе Avatar. Две функции, которые мы использовали, это AAvatar::Pitch() (ось y) и AAvatar::Yaw() (ось x). Расширение этих функций даст вам возможность передавать вводные мыши в HUD. Сейчас я покажу вам функцию Yaw и вы сможете экстраполировать отсюда как работает Pitch:

```
void AAvatar::Yaw( float amount )
{
   // ось x
   if( inventoryShowing )
   {
      // Когда инвентарь показан,
      // передаём вводные в HUD
      APlayerController* PController = GetWorld()->GetFirstPlayerController();
      AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
      hud->MouseMoved();
      return;
   }
   else
   {
      AddControllerYawInput(200.f*amount * GetWorld()- >GetDeltaSeconds());
   }
}
```

Функция AAvatar::Yaw() сначала проверяет показывается ли инвентарь или нет. Если показывается, вводные направляются прямо в HUD, без эффекта на Avatar. Если HUD не показывается, то вводные просто идут в Avatar.

Упражнения

1. Закончите функцию AAvatar::Pitch() (ось y), чтобы направлять вводные в HUD, а не в Avatar.
2. Сделайте NPC персонажей из Главы 8. *Действующие лица и пешки*, и дайте игроку предмет (такой как GoldenEgg), когда он проходит рядом с ними.

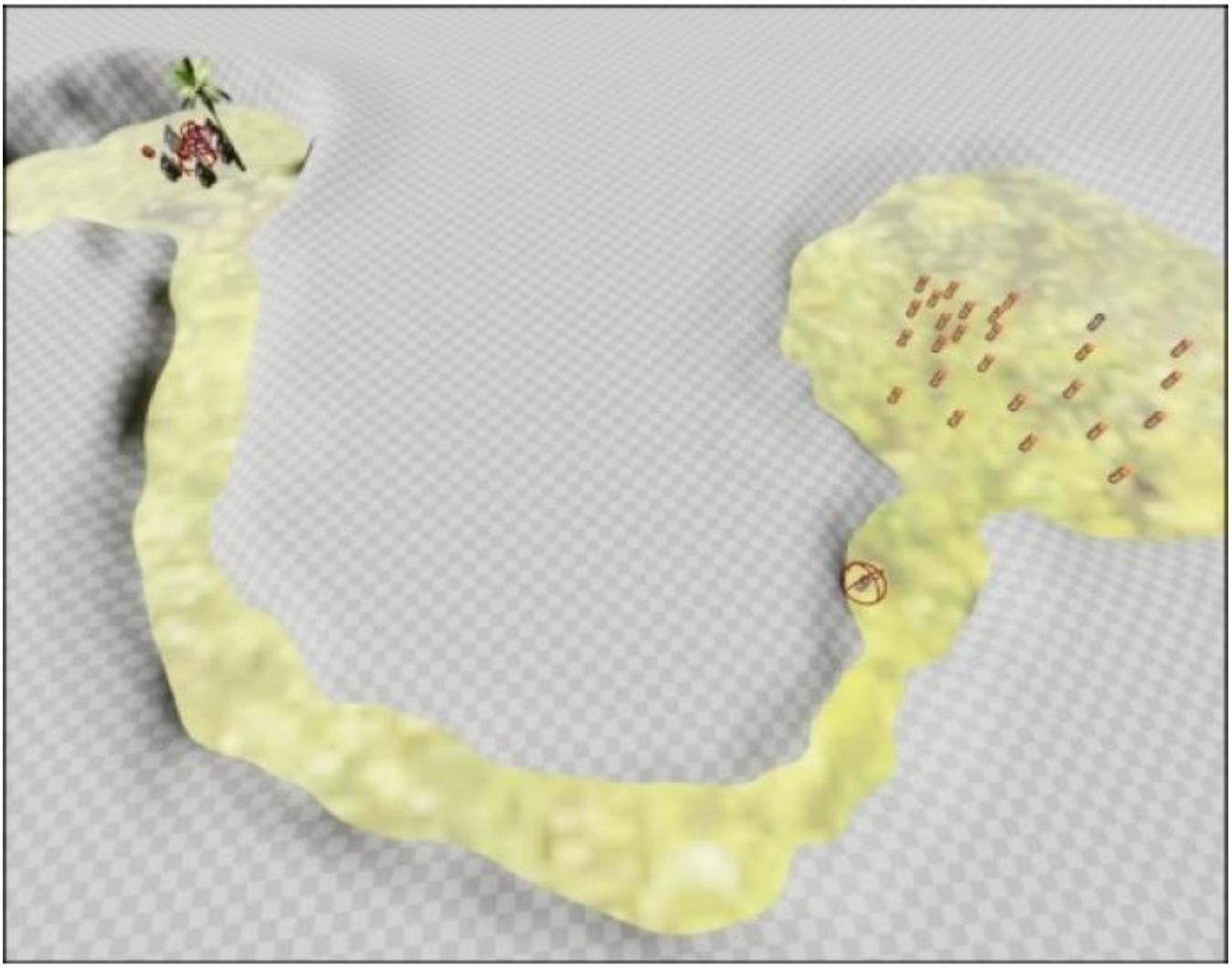
Выводы

В этой главе, мы прошли как устанавливать множество подбираемых предметов для игрока, чтобы видеть их отображёнными в уровне и подбирать. В следующей главе мы вводим *Монстров,* и игрок будет способен обороняться против них, используя магические заклинания.

Глава 11. Монстры

Мы добавим группу оппонентов для игрока.

Что мы сделаем в этой игре? Добавим пейзаж, например. Игрок пойдёт по дорожке, сделанной для него и затем, встретит армию. Есть NPC, которых он встретит до армии, который предложит совет.

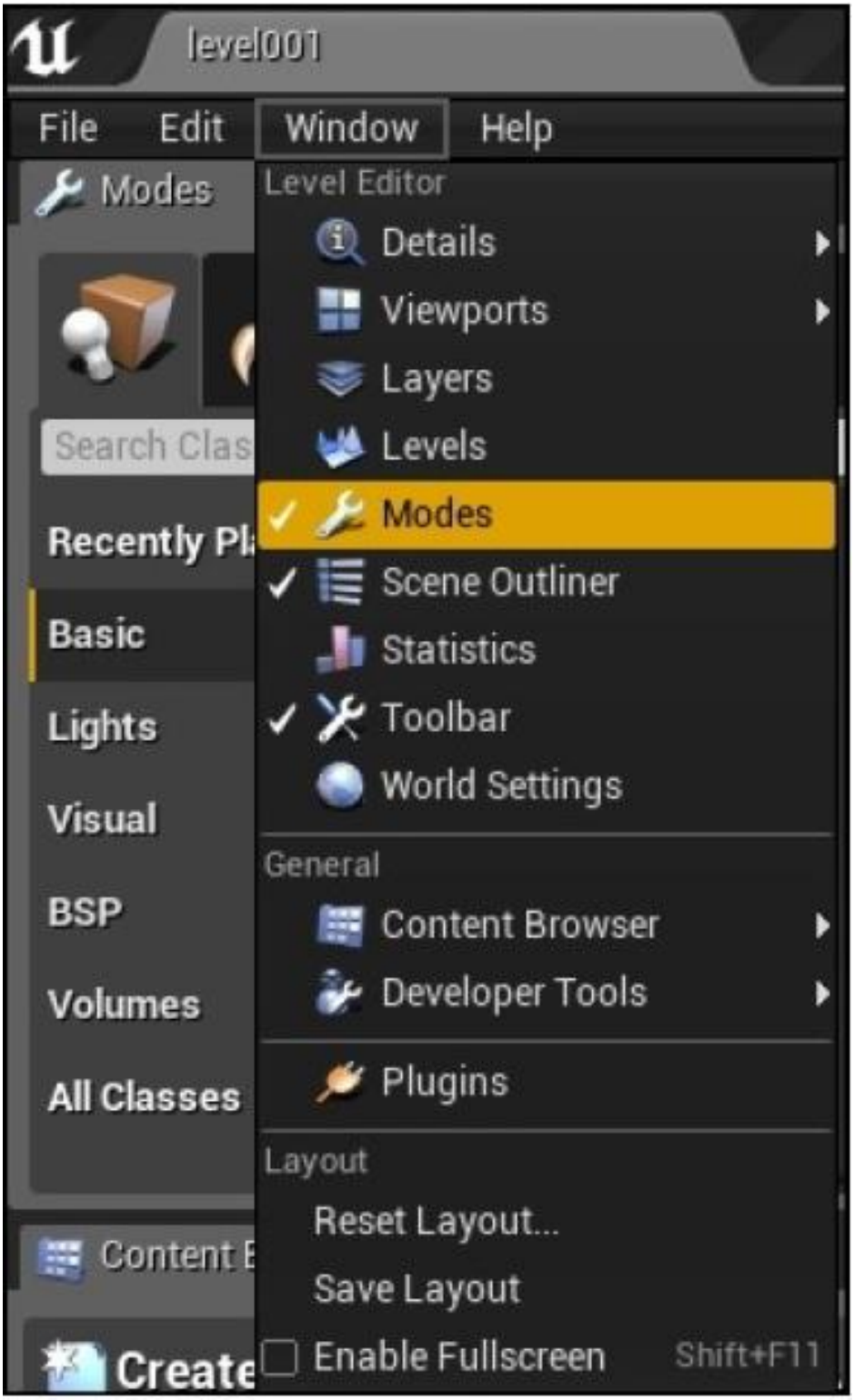


Сцена начинает выглядеть как игра

Пейзаж

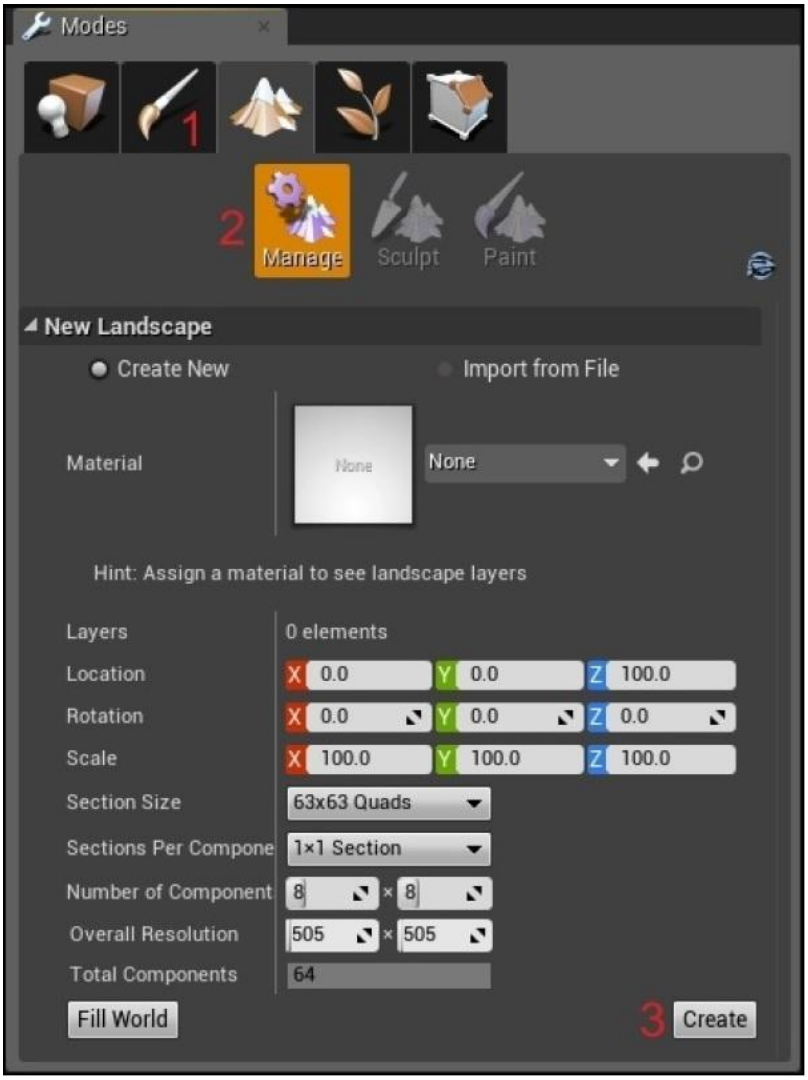
В этой книге мы ещё не проходили как формировать пейзаж, но мы сделаем это здесь. Во первых, у вас должен быть пейзаж, с которым работать. Создайте новый файл, перейдя в **File** | **New**. Вы можете выбрать пустой уровень, либо уровень с небом. Я в этом примере выбрал без неба.

Чтобы создать пейзаж, нам нужно поработать с панелью **Modes**. Убедитесь, что панель Modes отображена, перейдя в **Window** | **Modes**:



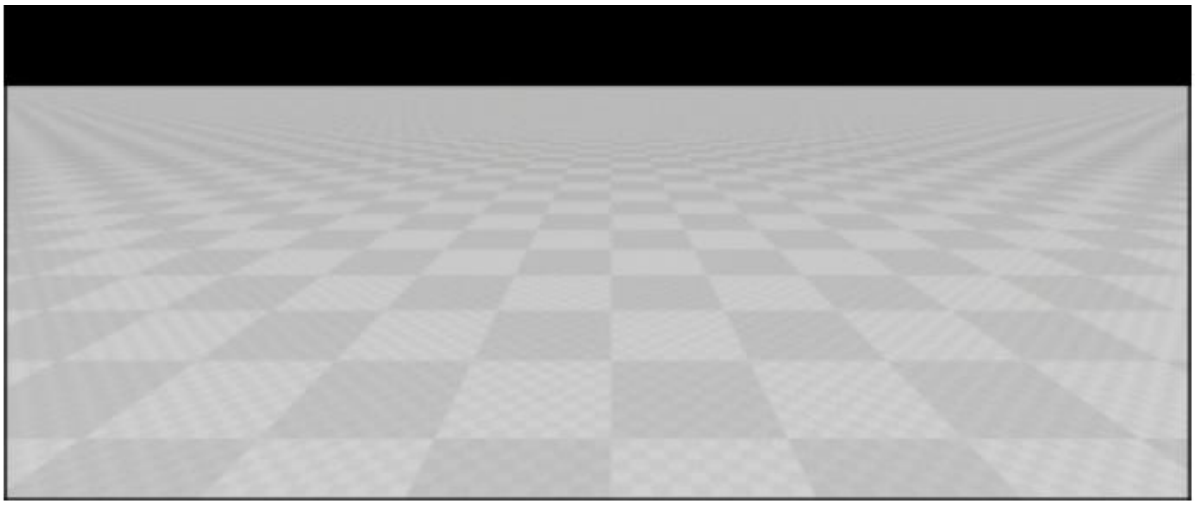
Панель Modes отображается

Пейзаж можно создать в три шага, которые показаны на следующем скриншоте, за которым идут соответствующие шаги:

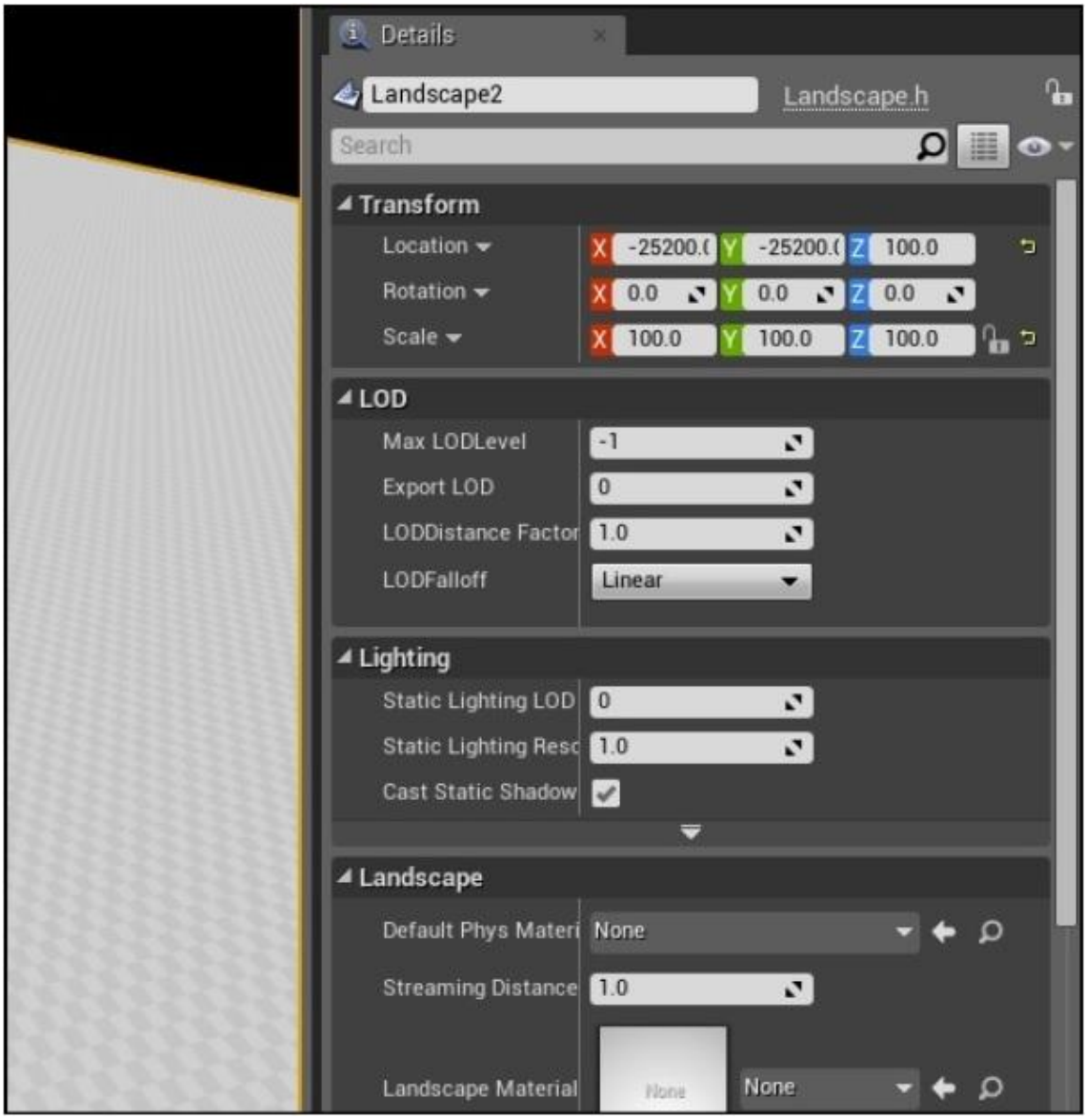


1. Щёлкните по значку пейзажа (с изображением гор) на панели **Modes**.
2. Нажмите кнопку **Manage**.
3. Затем, нажмите кнопку **Create** в нижнем правом углу экрана.

Теперь у вас должен быть пейзаж, с которым можно работать. Он появится как серая, мозаичная площадь в главном окне:



Первое, что вы захотите сделать со своим пейзажем сцены, это добавить цвета в него. Какой пейзаж без сцены? Щёлкните правой кнопкой мыши где угодно на своём сером, мозаичном пейзаже. На панели Details, справа, вы увидите, что там полно информации, как показано на следующем скриншоте:



Прокрутите вниз, пока не увидите свойство **Landscape Material**. Вы можете выбрать материал **M_Ground_Grass** (земля_трава) для реалистично выглядящей поверхности земли.

Далее добавьте свет в сцену. Вам наверно следует применить направленный свет, так чтобы вся поверхность земли имела одинаковое освещение.

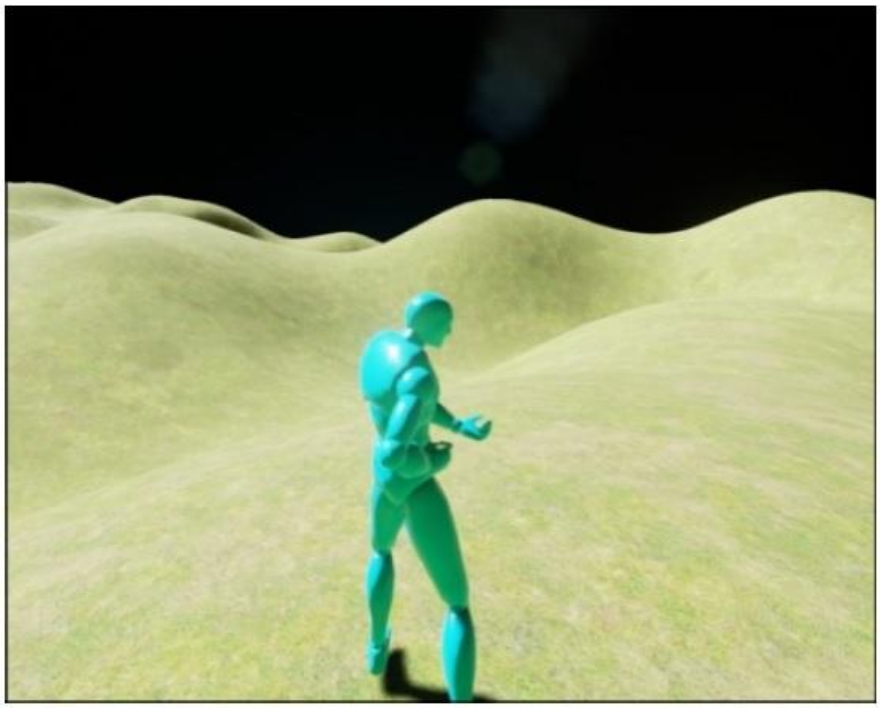
Скульптурирование пейзажа

Плоский пейзаж, вероятно будет скучным. Мы как минимум добавим немного изгибов и холмов. Чтобы сделать это, нажмите кнопку **Sculpt** на панели **Modes**:

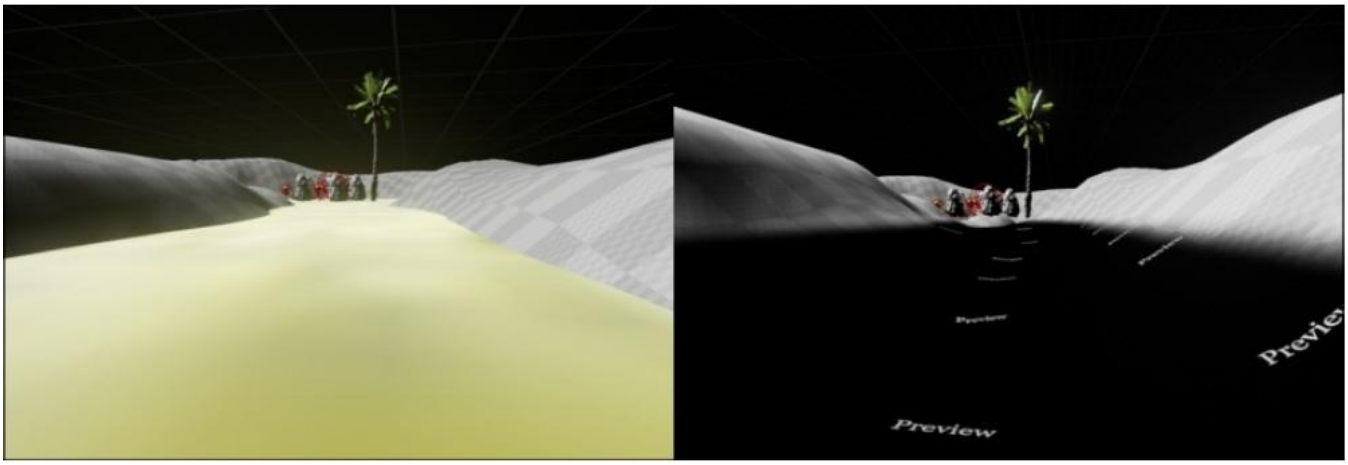
Чтобы изменить пейзаж, нажмите кнопку Sculpt

Размер и сила вашей кисти определены параметрами **Brush Size** и **Tool Strength** в окне **Modes**.

Щёлкните по вашему пейзажу удерживая кнопку, тащите мышь, чтобы изменить высоту бугров. Как только вы будете довольны результатом, нажмите кнопку **Play**, чтобы проверить. Итоговый результат, можно увидеть на следующем скриншоте:



Поэкспериментируйте со своим пейзажем и создайте сцену. Я занизил пейзаж, сделав поверхность более прямой, чтобы игрок мог ходить по хорошо определённой плоской территории, как показано на следующем скриншоте:



Можете делать со своим пейзажем всё, что угодно. В качестве примера вы можете использовать, то что я сделал, если хотите. Я советую вам импортировать ассеты из **ContentExamples** или из **StrategyGame**, чтобы применить их в вашей игре. Чтобы сделать это, обратитесь к разделу Импортирование ассетов в Главе 10. *Система инвентаризации и подбор предметов*. Когда вы закончите с импортом ассетов, мы сможем перейти к добавлению монстров в ваш мир.

Монстры

Мы начнём программировать монстров таким же способом, как мы программировали NPC и PickupItem. Во первых, мы напишем базовый класс (происходящий от персонажа), чтобы представить класс Monster. Затем, мы выполним происхождение группы схем (blueprint) для каждого типа монстров. У каждого монстра будет пара свойств общих со всеми, которые определяют их поведение. Вот общие свойства:

- Переменная типа float для скорости.
- Переменная типа float для значения HitPoints, обозначающего урон (я обычно использую тип float для HP, так что мы можем легко сможем смоделировать эффекты истощающие HP, такие как проходить через поток лавы).
- Переменная int32 для опыта, получаемого при победе над монстрами.
- Функция UClass для добычи оставленной монстром.
- Переменная типа float для BaseAttackDamage (базовый урон атаки) выполняемы при каждой атаке.

- Переменная типа float для AttackTimeout, что является объёмом времени, на который монстр прерывается между атаками.
- Два объекта USphereComponents: Один из них это SightSphere – как далеко он может видеть. Другой это AttackRangeSphere – как далеко могут достигать его атаки. Объект AttackRangeSphere всегда меньше чем SightSphre.

Выполните происхождение от класса Character, чтобы создать ваш класс Monster. В UE4 вы можете сделать это перейдя в **File** | **Add Code To Project…** и затем выбрав опцию **Character** в меню для вашего базового класса.

Заполните класс Monster базовыми свойствами. Убедитесь, что вы объявляете UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties), так что свойства монстров могут быть изменены в блупринтах:

```
UCLASS()
class GOLDENEGG_API AMonster : public ACharacter
{
   GENERATED_UCLASS_BODY()

   // Насколько он быстр
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   float Speed;

   // Единицы здоровья монстра
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   float HitPoints;

   // Опыт получаемый от побед
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   int32 Experience;

   // Blueprint типа предмета обронённого монстром
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   UClass* BPLoot;

   // Объём урона получаемого от атаки
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   float BaseAttackDamage;

   // Объём времени, необходимый монстру для отдыха
   // между атаками
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
   float AttackTimeout;

   // Время с последнего удара монстра, читаемое в blueprint
   UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = MonsterProperties)
   float TimeSinceLastStrike;

   // Расстояние на котором он видит
   UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision)
   USphereComponent* SightSphere;

   // Дальность его атаки. Визуализируемая в редакторе как сфера
   UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision)
```

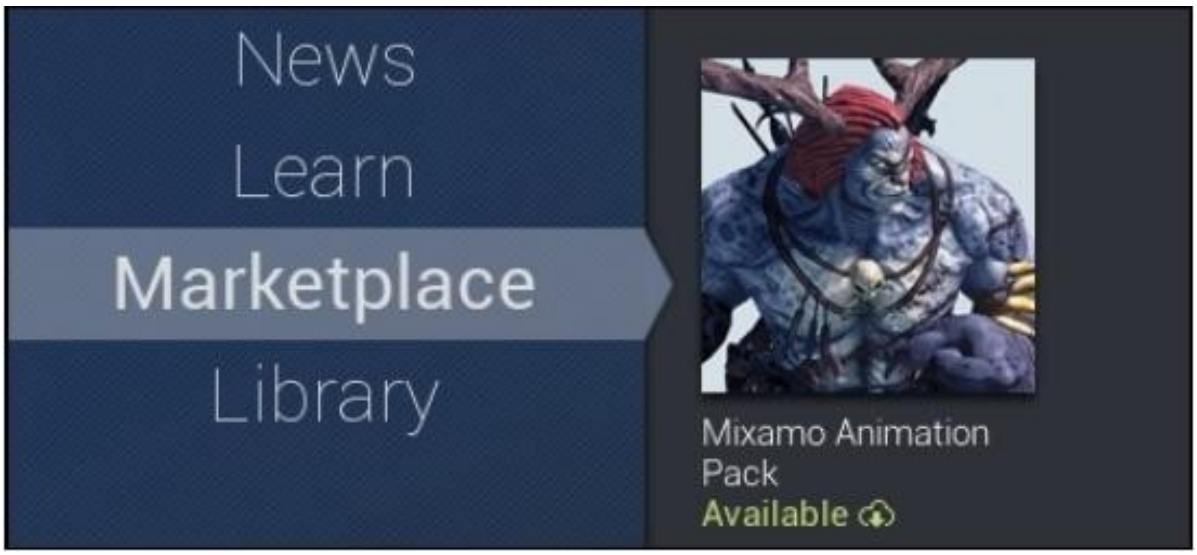
```
    USphereComponent* AttackRangeSphere;
};
```

Вам понадобится минимальный код в вашем конструкторе Monster, чтобы задать свойства монстра. Используйте следующий код в файле Monster.cpp:

```
AMonster::AMonster(const class FObjectInitializer& PCIP) : Super(PCIP)
{
    Speed = 20;                     // Скорость
    HitPoints = 20;                 // Здоровье
    Experience = 0;                 // Опыт
    BPLoot = NULL;                  // Трофеи
    BaseAttackDamage = 1;           // Урон базовой атаки
    AttackTimeout = 1.5f;           // Длительность атаки
    TimeSinceLastStrike = 0;        // Прошло времени с последнего удара

    SightSphere = PCIP.CreateDefaultSubobject<USphereComponent> (this,
    TEXT("SightSphere"));
    SightSphere->AttachTo( RootComponent );

    AttackRangeSphere = PCIP.CreateDefaultSubobject <USphereComponent>(this,
    TEXT("AttackRangeSphere"));
    AttackRangeSphere->AttachTo( RootComponent );
}
```

Компилируйте и запустите код. Отройте Unreal Editor и выполните происхождение блупринт основанного на вашем классе Monster (назовите его BP_Monster). Теперь мы можем начать конфигурировать свойства ваших монстров.

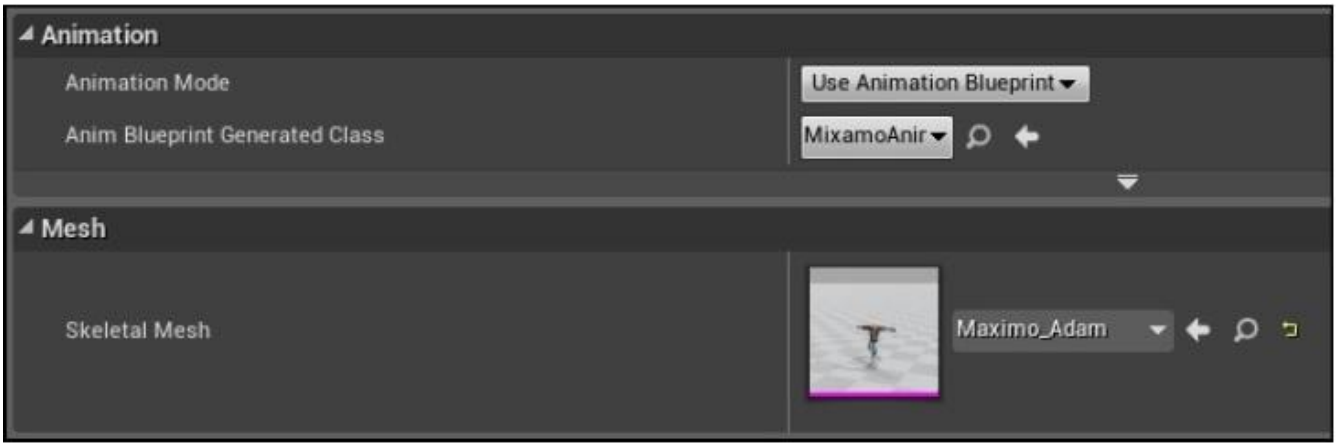
Для скелетной сетки мы хотим использовать модель HeroTPP для монстра, потому что нам нужно, чтобы монстр мог выполнять рукопашную атаку, а HeroTPP не идёт с рукопашной атакой. Однако некоторые модели в файле **Maximo Animation Pack** обладают анимацией рукопашной атаки. Итак, скачайте файл **Maximo Animation Pack** в **Marketplace** (бесплатно).



Некоторые модели в пакете довольно крупные, которых я бы избежал, но есть и вполне хорошие

Далее вам надо добавить файл Maximo Animation Pack в ваш проект, как показано на следующем скриншоте:

Теперь, создайте блупринт и назовите его BP_Monster основанный на вашем классе Monster. Отредактируйте свойства класса блупринт и выберите **Mixamo_Adam** (в текущем выпуске пакета на самом деле написано как **Maximo_Adam**) как скелетную сетку. Также выберите **MixamoAnimBP_Adam** как блупринт анимации.

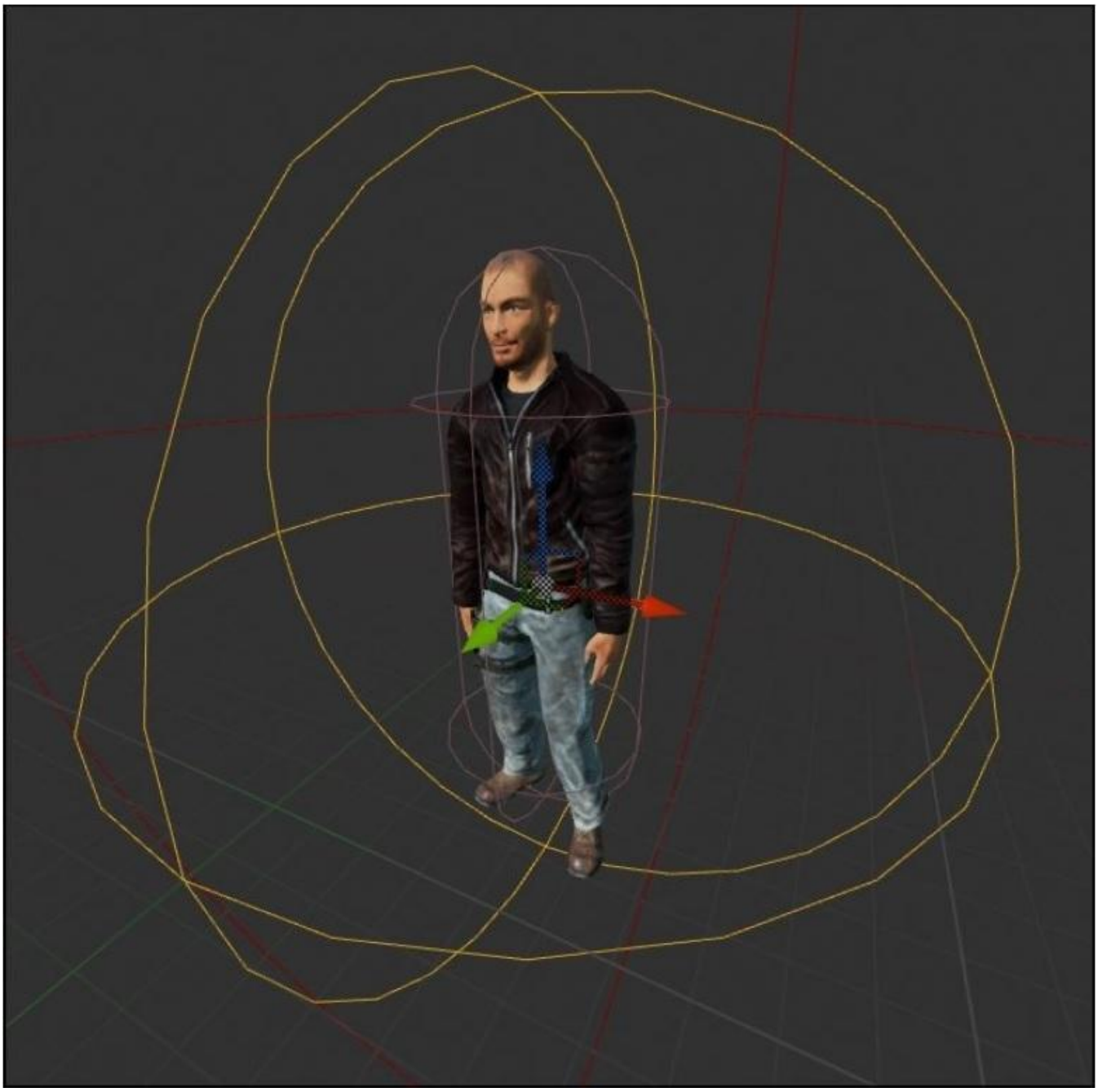


Выберите Skeletal Mesh Maximo_Adam и MixamoAnimBP_Adam для Anim Blueprint Generated class

Мы модифицируем блупринт, чтобы позднее корректно совместить с анимацией рукопашной атаки.

Когда редактируете ваш блупринт BP_Monster, измените размер объектов SightSphere и AttackRangeSphere на значения, которые имеют для вас смысл. Я сделал объект AttackRangeSphere моего монстра достаточным для расстояния вытянутой руки (60 единиц) и его объект SightSphere в 25 раз больше этого (1500 единиц).

Запомните, что монстр начнёт двигаться на игрока, как только игрок войдёт SightSphere монстра, и монстр начнёт атаковать игрока, как только игрок окажется в объекте AttackRangeSphere монстра.



Mixamo Adam с его объектом AttackRangeSphere подсвеченным оранжевым

Поместите несколько ваших экземпляров BP_Monster в вашу игру, компилируйте и запустите. Без кода движения персонажа Monster, ваши монстры будут просто стоять.

Базовый интеллект монстров

В нашей игре, мы добавим только базовый интеллект персонажам Monster. Монстры будут знать, как делать две базовые вещи:

- Выслеживать игрока и преследовать его
- Атаковать игрока

Монстр больше ничего не будет делать. Вы можете сделать, чтобы монстр дразнил игрока, когда увидит первый раз, но это мы припасли в качестве упражнения для вас.

Движение монстра – управляемое поведение

Монстры в самых базовых играх, обычно не обладают сложным поведением движения. Обычно они просто идут на цель и атакуют. Мы спрограммируем такой тип монстров в этой игре. Но представьте себе, вы можете получить более интересную игру с монстрами, которые сами располагаются с преимуществом на местности, чтобы выполнять удалённые атаки и так далее. Этого мы не будем программировать здесь, но об этом стоит подумать.

Для того, чтобы персонаж Monster двигался на игрока, нам нужно динамически обновлять направление двигающегося персонажа Monster в каждом кадре. Чтобы обновлять направление, в котором расположен монстр, мы пишем код в методе Monster::Tick().

Функция Tick запускается в каждом кадре игры. Запись функции Tick такова:

```
virtual void Tick(float DeltaSeconds) override;
```

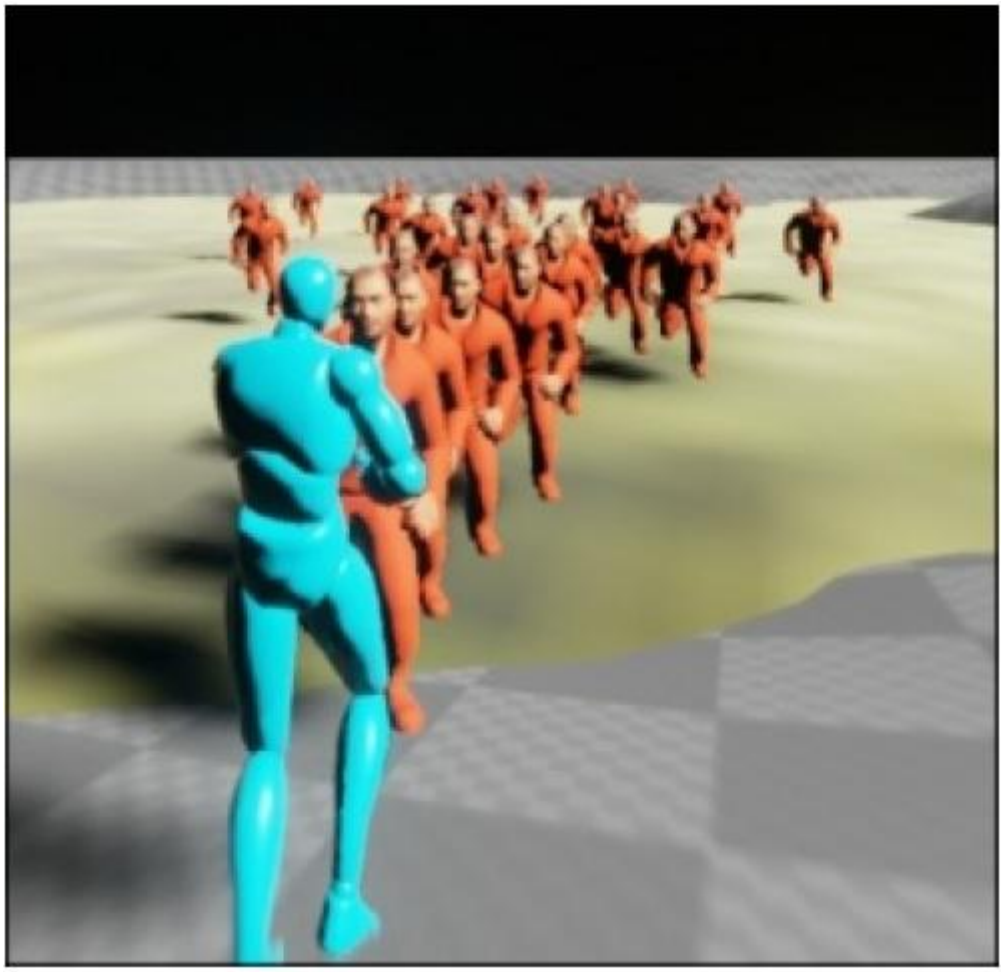
Вам нужно добавить прототип этой функции в ваш класс AMonster, в вашем файле Monster.h. Если мы подменяем Tick, мы можем устроить своё собственное поведение, которое персонаж Monster должен выполнять в каждом кадре. Вот базовый код который будет двигать монстра на игрока в течении каждого кадра:

```
void AMonster::Tick(float DeltaSeconds)
{
    Super::Tick( DeltaSeconds );

    // базовый интеллект: двигает монстра на игрока
    AAvatar *avatar = Cast<AAvatar>( UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
    if( !avatar ) return;

    FVector toPlayer = avatar->GetActorLocation() - GetActorLocation();
    toPlayer.Normalize(); // reduce to unit vector

    // Собственно двигаем монстра на игрока
    AddMovementInput(toPlayer, Speed*DeltaSeconds);

    // Обращение лицом к цели
    // Получаете ротатор для поворачивания того,
    // что смотрит в направлении игрока `toPlayer`
    FRotator toPlayerRotation = toPlayer.Rotation();
    toPlayerRotation.Pitch = 0; // 0 off the pitch
    RootComponent->SetWorldRotation( toPlayerRotation );
}
```

Чтобы работало AddMovementInput, у вас должен быть выбран контроллер внизу панели **AIController Class** в вашем блупринте, как показано на следующем скриншоте:

Если у вас выбрано None (ничего), то вызов AddMovementInput не будет иметь никакого эффекта. Чтобы предотвратить это, выберите либо класс AIController, либо класс PlayerController в качестве вашего **AIController Class**.

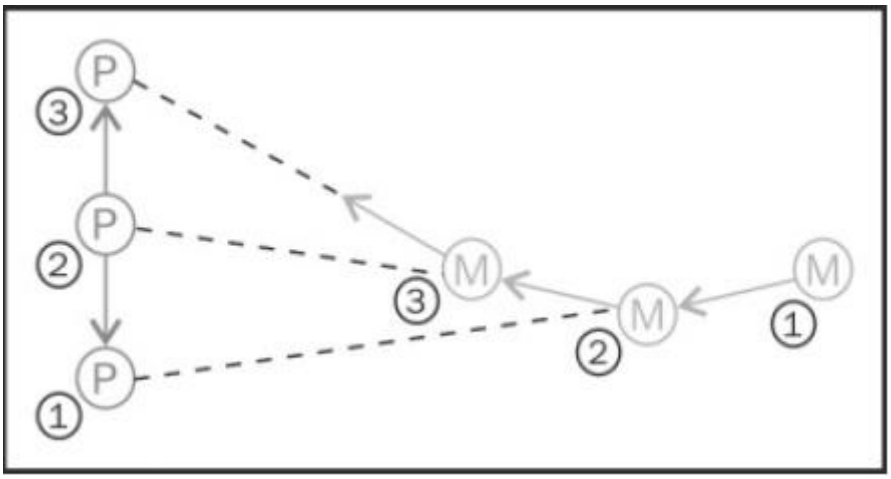
Предыдущий код очень простой. Он содержит самую базовую форму интеллекта противника: просто движение на игрока посредством небольшого инкрементного возрастания в каждом кадре.



Наша не особо интеллектуальная армия гонится за игроком

В результате серии кадров, монстр выслеживает и преследует игрока в уровне. Чтобы понять, как это работает, вы должны помнить, что функция Tick, в среднем вызывается 60 раз в секунду. Что это значит? То что в каждом кадре, монстр понемногу продвигается к игроку. Поскольку монстр движется очень маленькими

шажками, его действия выглядят плавными и продолжительными (в то время как на самом деле, он делает маленькие скачки и прыжки в каждом кадре).



Детальная суть отслеживания: движение монстра в трёх наложенных кадрах

Подсказка

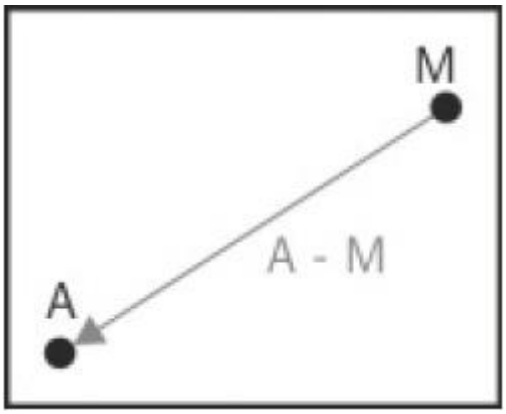
Почему монстр двигается 60 кадров в секунду? Из-за аппаратного ограничения. Частота обновления типичного монитора составляет 60Гц, так что это действует как практический ограничитель того, сколько обновлений в секунду имеет смысл. Обновление частоты кадров быстрее, чем обновление частоты, возможно, но это в этом не будет пользы для игр, так как вы будете видеть новую картинку только каждую 1/60 секунды на большинстве компьютеров. Некоторые продвинутые симуляторы физического моделирования выполняют почти 1,000 обновлений в секунду. Но тут можно поспорить. Вам не нужно разрешение такого типа для игры и вы займёте лишнее время ЦПУ предназначенное для того, чем будет довольствоваться игрок, как например лучшие алгоритмы AI (искусственного интеллекта). Некоторые более новые аппаратные устройства разгоняют частоту обновления до 120 Гц (посмотрите игровые мониторы, но не говорите вашим родителям, то сказал вам спустить все ваши деньги на это).

Детальная суть движения монстра

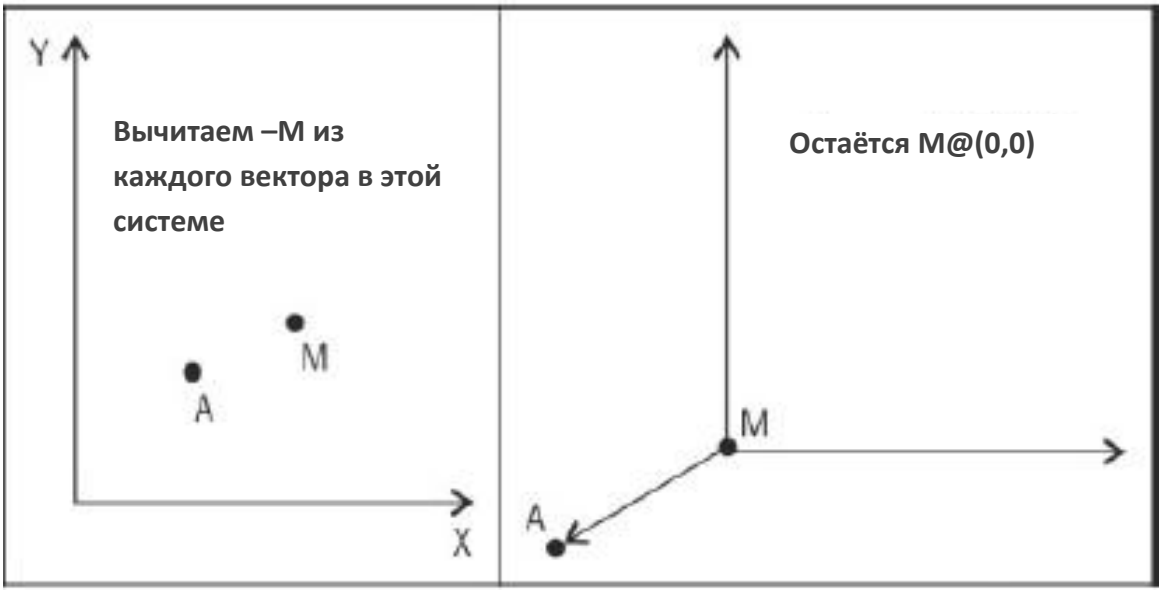
Компьютерные игры разбиты на детали. На предыдущем изображении последовательности наложенных кадров, движение игрока (P) представлено прямо вверх на экране, маленькими шажками. Движение монстра также в маленьких шажках. В каждом кадре, монстр выполняет один маленький шаг в сторону игрока. Монстр следует видимой кривой дорожкой, двигаясь прямо туда где находится игрок в каждом кадре.

Чтобы двигать монстра на игрока, нам сначала нужно получить позицию игрока. И так как игрок доступен в глобальной функции, UGameplayStatics::GetPlayerPawn, мы просто берём наш указатель на игрока, используя эту функцию. Далее мы находим указывающий вектор из функции Monster (GetActorLocation()), который указывает

от монстра на аватар. Чтобы сделать это, вам нужно вычесть местоположение монстра из местоположения аватара, как показано на следующем изображении:



Это простое математическое правило и его нужно запомнить. Но часто оно понимается неправильно. Чтобы получить правильный вектор, всегда вычитайте исходный вектор (начальная точка) от целевого вектора (конечная точка). В нашей системе, нам нужно вычитать вектор Monster из вектора Avatar. Это работает, потому что вычитание вектора Monster из системы, двигает вектор Monster к началу координат и вектор Avatar будет снизу слева от вектора Monster:



Вычитание вектора Monster из системы координат двигает вектор Monster к (0,0)

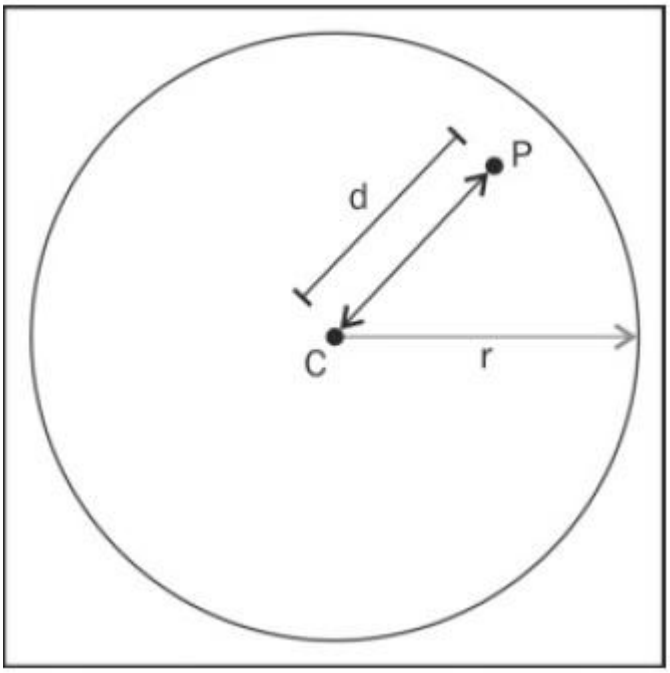
Обязательно испытайте ваш код. С этого момента монстры будут бегать за вашим игроком и толпиться вокруг него. С предыдущим кодом, который выделен, они не будут атаковать. Они просто будут следовать за ним повсюду, как показано на следующем скриншоте:

SightSphere монстра

Прямо сейчас, монстры не обращают внимания на компонент SightSphere. Где бы ни находился игрок в мире, в текущей установке монстры будут идти на него. И мы хотим изменить это сейчас.

Чтобы сделать это, нам нужно, чтобы Monster принимал во внимание ограничение SightSphere. Если игрок внутри объекта SightSphere монстра, то тогда монстр устроит погоню. Иначе, монстры будут в неведении местоположения игрока и естественно не будут за ним гнаться.

Проверять находится ли объект внутри сферы легко. На следующем изображении, точка **P** находится внутри сферы, если расстояние **d** между **P** и центром **C** меньше, чем радиус сферы **r**:

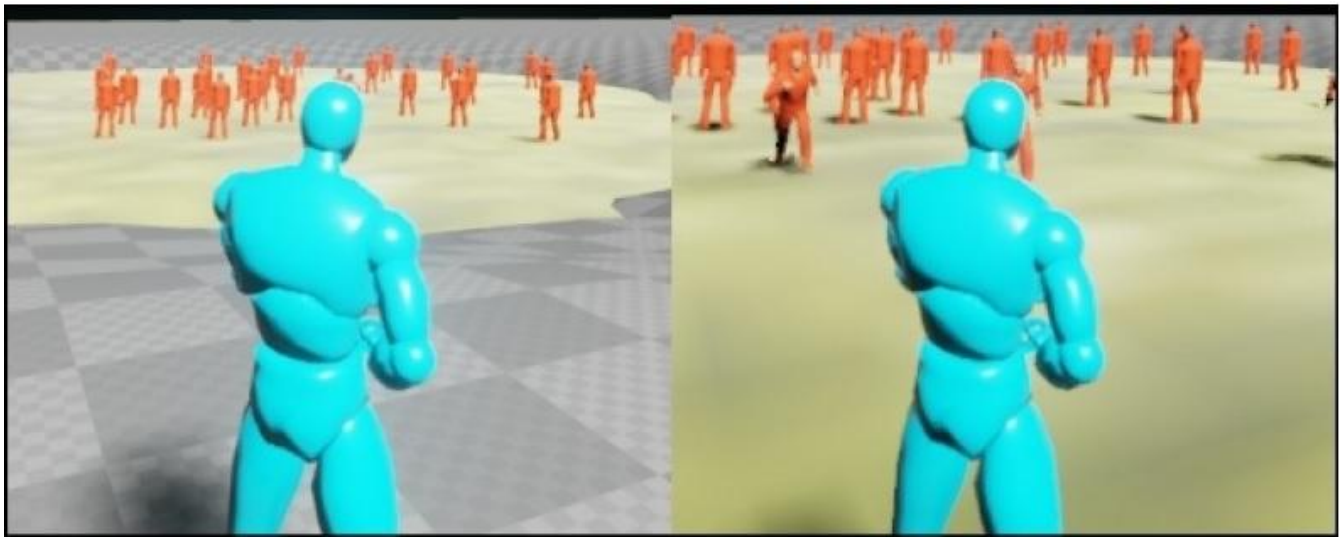


P находится внутри сферы, когда d меньше чем r

Итак, в нашем коде, предыдущее изображение переходит в следующий код:

```
void AMonster::Tick(float DeltaSeconds)
{
    Super::Tick( DeltaSeconds );
    AAvatar *avatar = Cast<AAvatar>( UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
    if( !avatar ) return;
        FVector toPlayer = avatar->GetActorLocation() - GetActorLocation();
    float distanceToPlayer = toPlayer.Size();
    // Если игрок не в SightSphere монстра,
    // идём назад
    if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
    {
        // Если игрок в не поля зрения,
        // то монстр не может гнаться за ним
        return;
    }

    toPlayer /= distanceToPlayer; // нормализуем вектор
    // Собственно двигаем монстра на игрока по немногу
    AddMovementInput(toPlayer, Speed*DeltaSeconds);
    // (остальная часть функции такая же как прежде (вращение))
}
```

Этот кот добавляет дополнительный интеллект персонажу Monster. Персонаж Monster сейчас может прекращать погоню за игроком, если игрок вне объекта SightSphere монстра. Вот как будет выглядеть результат:



Хорошо было бы здесь внести сравнение расстояния в простую встроенную функцию. Мы можем вести эти две встроенные функции-члены в заголовочный файл Monster:

```
inline bool isInSightRange( float d )
{ return d < SightSphere->GetScaledSphereRadius(); }
inline bool isInAttackRange( float d )
{ return d < AttackRangeSphere->GetScaledSphereRadius(); }
```

Эти функции возвращают значение true, когда переданный параметр d внутри рассматриваемой сферы.

Подсказка

Функция inline (встроенная) означает, что функция больше походит на макрос, чем на функцию. Макросы копируются и вставляются в вызываемую локацию, в то время как функции перепрыгивают по средством C++ и выполняются на своих локациях. Встроенные функции хороши, потому что они предоставляют хорошую производительность, наряду с тем, что код остаётся лёгок для чтения, и их можно использовать повторно.

Монстр нападает на игрока

Есть несколько различных типов атаки, что может выполнять монстр. В зависимости от типа персонажа Monster, атака монстра может быть рукопашной (близкая дистанция) или же удалённой (стрелковое оружие).

Персонаж Monster будет атаковать игрока всегда, когда игрок находится в его AttackRangeSphere (сфера расстояния атаки). Если игрок за пределами расстояния атаки монстра (AttackRangeSphere), но в пределах поля зрения монстра (объект

SightSphere), то монстр пойдёт на сближение с игроком, пока игрок не окажется в AttackRangeSphere монстра.

Рукопашная атака

Словарное определение слова *melee* – рукопашный, сбивает с толку массу народа. Рукопашная атака (melee attack) выполняется на близком расстоянии. Представьте группу *зерглингов* сражающихся с группой *ультралисков* (если вы играете в StarCraft, то вы знаете, что и *зерглинги* и *ультралиски* ведут рукопашные схватки). Рукопашные атаки обычно ведутся на близком расстоянии, на расстоянии вытянутой руки. Чтобы осуществить рукопашную атаку, вам нужна анимация рукопашной атаки, которая включается, когда монстр начинает свою рукопашную атаку. Для этого, вам надо отредактировать блупринт анимации в *Persona*, UE4 редакторе анимации.

Совет

Серии *Persona* Зака Пэриша, это великолепное место, с которого можно начать, перед тем как программировать анимации в блупринтах: https://www.youtube.com/watch?v=AqYmC2wn7Cg&list=PL6VDVOqa_mdNW6JEu9UAS_s40OCD_u6yp&index=8.А сейчас мы просто спрограммируем рукопашную атаку, а затем позаботимся о модификации анимации в блупринтах.

Определение оружия для ближнего боя

Будут три части, определяющие наше оружие для рукопашной схватки, то есть для ближнего боя. Первая часть это C++ код, который представляет это. Вторая это модель, а третья часть будет связывать код и модель, используя блупринт UE4.

Пишем код для оружия для ближнего боя

Мы определим новый класс, AMeleeWeapon (происходящий от AActor), чтобы представить оружие которое держится в руке. Я добавлю пару редактируемых в блупринт свойств в класс AMeleeWeapon, и этот класс будет выглядеть следующим образом:

```
class AMonster;

UCLASS()
class GOLDENEGG_API AMeleeWeapon : public AActor
{
   GENERATED_UCLASS_BODY()

   // Объём урона от атаки его оружием
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MeleeWeapon)
   float AttackDamage;
```

```
    // Список того, что оружие уже ударило этим взмахом
    // Убеждаемся, что всё через что проходит меч, получает удар один раз
    TArray<AActor*> ThingsHit;

    // предотвращаем урон, в кадрах
    // где не было взмаха меча
    bool Swinging;

    // "Перестань бить себя" – используется, чтобы проверить если
    // актор держащий оружие бьёт сам себя
    AMonster *WeaponHolder;

    // ограничивающий блок, определяющий, когда бьёт оружие
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    MeleeWeapon)
    UBoxComponent* ProxBox;

    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category =
    MeleeWeapon)
    UStaticMeshComponent* Mesh;

    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
    OtherBodyIndex, bool bFromSweep, const FHitResult & SweepResult );
    void Swing();
    void Rest();
};
```

Обратите внимание, как я использовал ограничивающий блок (также известный как ограничивающий параллелепипед) для ProxBox, а не ограничивающую сферу. Это потому что мечи и топоры будут лучше обрабатываться посредством блоков нежели сфер. Есть две функции-члены, Rest() и Swing(), которые дают MeleeWeapon знать, в каком состоянии находится актор (стоит спокойно или делает взмахи). Есть также свойство TArray<AActor*> ThingsHit в этом классе, которое отслеживает удары актора, при каждом взмахе оружием. Мы программируем это так, что оружие может наносить только один удар при одном взмахе.

Файл AMeleeWeapon.cpp будет содержать только базовый конструктор и какой-то простой код, чтобы посылать повреждения другому актору – OtherActor, когда наш меч бьёт его. Файл MeleeWeapon.cpp содержит следующий код:

```
AMeleeWeapon::AMeleeWeapon(const class FObjectInitializer& PCIP) :
Super(PCIP)
{
    AttackDamage = 1;
    Swinging = false;
    WeaponHolder = NULL;

    Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this, TEXT("Mesh"));
    RootComponent = Mesh;

    ProxBox = PCIP.CreateDefaultSubobject<UBoxComponent>(this, TEXT("ProxBox"));
```

```
    ProxBox->OnComponentBeginOverlap.AddDynamic( this, &AMeleeWeapon::Prox );
    ProxBox->AttachTo( RootComponent );
}

void AMeleeWeapon::Prox_Implementation( AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult & SweepResult )
{
    // не бьём не корневые компоненты
    if( OtherComp != OtherActor->GetRootComponent() )
    {
        return;
    }

    // избегаем нанесение ударов когда нет замахов мечём,
    // избегаем нанесение удаоров по себе, и
    // избегаем нанесение ударов по одному OtherActor дважды
    if( Swinging && OtherActor != WeaponHolder && !ThingsHit.Contains(OtherActor) )
    {
        OtherActor->TakeDamage( AttackDamage + WeaponHolder- >BaseAttackDamage,
        FDamageEvent(), NULL, this );
        ThingsHit.Add( OtherActor );
    }
}
void AMeleeWeapon::Swing()
{
    ThingsHit.Empty(); // опустошаем список
    Swinging = true;
}
void AMeleeWeapon::Rest()
{
    ThingsHit.Empty();
    Swinging = false;
}
```

Скачиваем меч

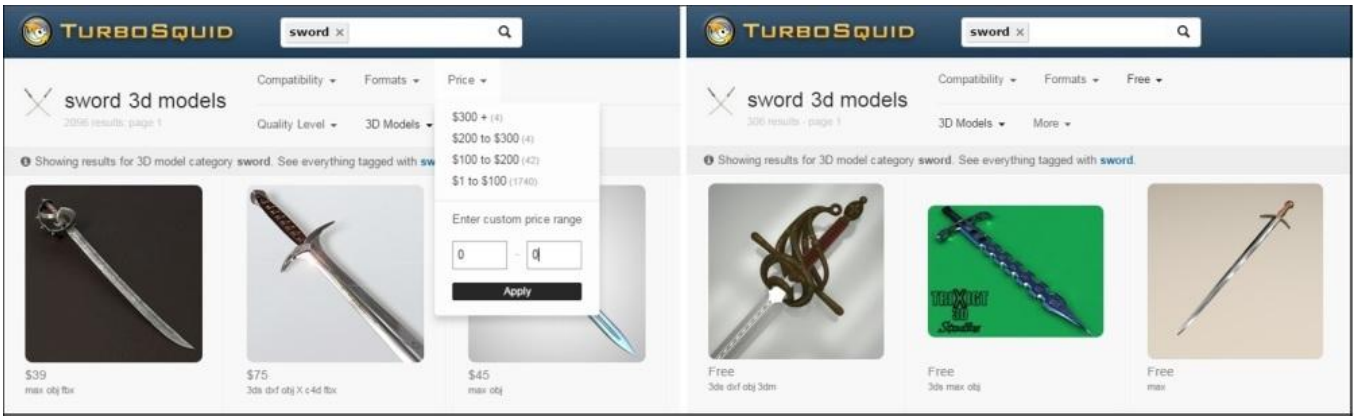
Чтобы закончить это упражнение, нам нужен меч, который можно поместить в руку модели. Я добавил меч в проект, который называется *Kilic* с http://tf3dm.com/3d-model/sword-95782.html от Каана Гюльхана. Вот список других мест, где вы можете взять бесплатные модели:

- http://www.turbosquid.com/
- http://tf3dm.com/
- http://archive3d.net/
- http://www.3dtotal.com/

Подсказка

Секретный совет

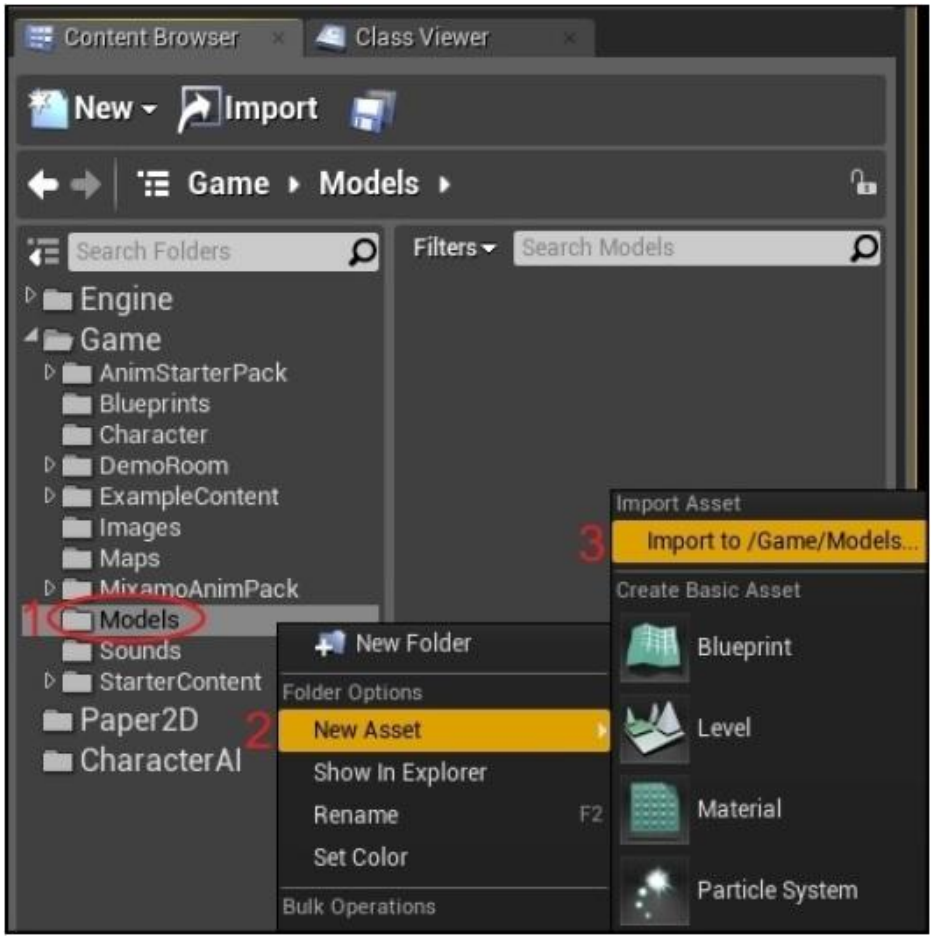
Сначала на TurboSquid.com будет выглядеть так, будто нет бесплатных моделей. На самом деле есть секрет. В поиске вы vожете цену 0-0, что является бесплатно.



Поиск бесплатных мечей в TurboSquid

Мне нужно немного отредактировать сетку меча *Kilic*, чтобы поправить начальные размер и поворот. Вы можете импортировать любую сетку в формат **Filmbox** (**FBX**) вашей игре. Модель меча Kilic есть в пакете примера кода для Главы 11. *Монстры*.

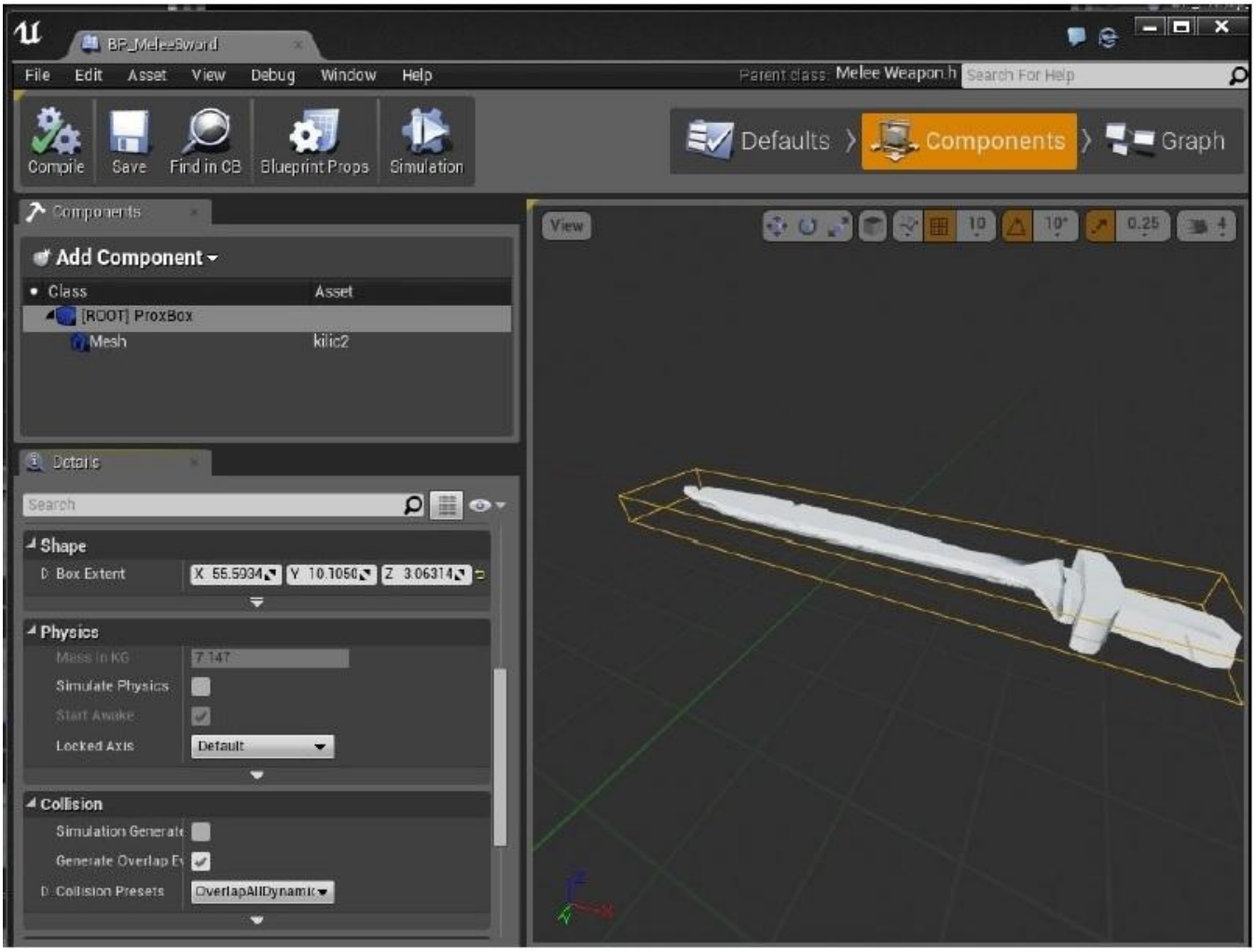
Чтобы импортировать ваш меч в редактор UE4 щёлкните правой кнопкой мыши по любой папке, в которую вы хотите добавить модель. Перейдите в **New Asset** | **Import to** | **Game** | **Models…**, и из обозревателя файлов, который появится, выберите новый ассет, который вы хотите добавить. Если папки **Models** не существует, вы можете создать её, просто нажав правой кнопкой мыши в дереве обзора слева и выбрать **New Folder** в панели слева от вкладки **Content Browser**. Я выбрал ассет kilic.fbx на своём рабочем столе.



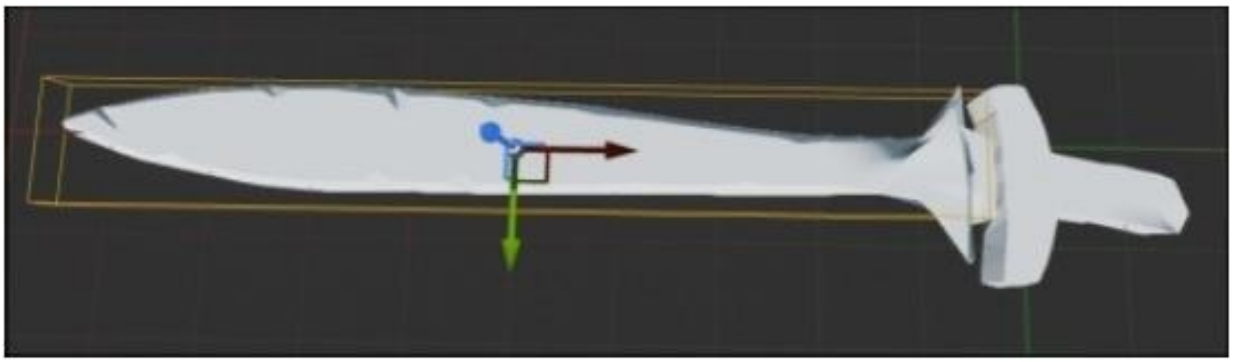
Импортирование в ваш проект

Создаём блупринт для вашего оружия

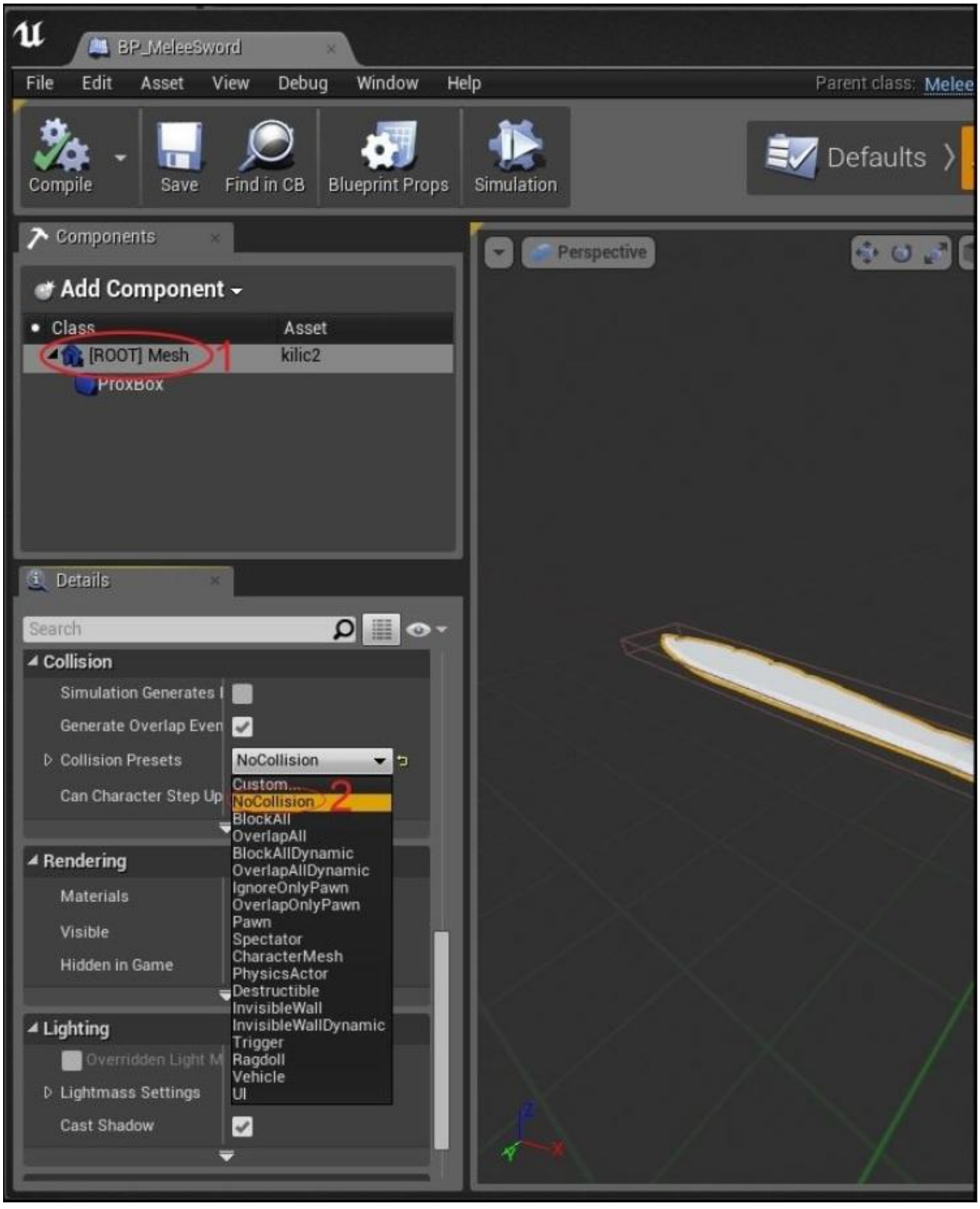
В редакторе UE4, создайте блупринт не основанный а AMeleeWeapon и назовите его BP_MeleeSword. Конфигурируйте BP_MeleeSword, чтобы использовать лезвие одели *kilic* (или лезвие любой модели выбранной вами), как показано на следующем скриншоте:



Класс ProxBox будет определять, было ли что-нибудь ударено оружием, так что мы модифицируем класс ProxBox так, чтобы он охватывал лезвие меча, как показано на следующем скриншоте:



Также под панелью **Collision Presets**, важно выбрать опцию **NoCollision** для сетки (не **BlockAll**). Это изображено на следующем скриншоте:



Если вы выберите **BlockAll**, то игровой движок будет автоматически решать взаимопроникновения между мечём и персонажем, отталкивая то чего коснулся меч при любом взмахе. В результате ваш персонаж будет подлетать при любом взмахе меча.

Сокеты

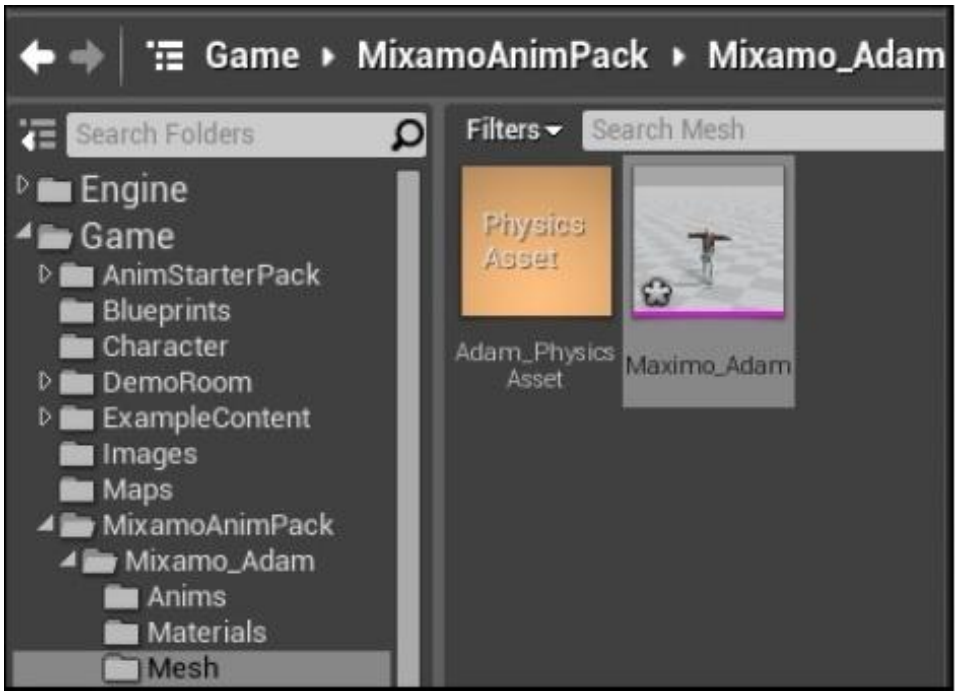
Сокет в UE4, это вместилище или можно сказать гнездо в скелетной сетке для другого актора. Вы можете располагать сокет где угодно на теле скелетной сетки. После того, как вы корректно расположите сокет, вы можете прикреплять другой актор к этому сокету в коде UE4.

Например, если мы хотим поместить меч в руку нашего монстра, то нам просто нужно создать сокет в руке нашего монстра. Мы можем добавить шлем игроку, создав сокет не его голове.

Создаём сокет скелетной сетки в руке монстра

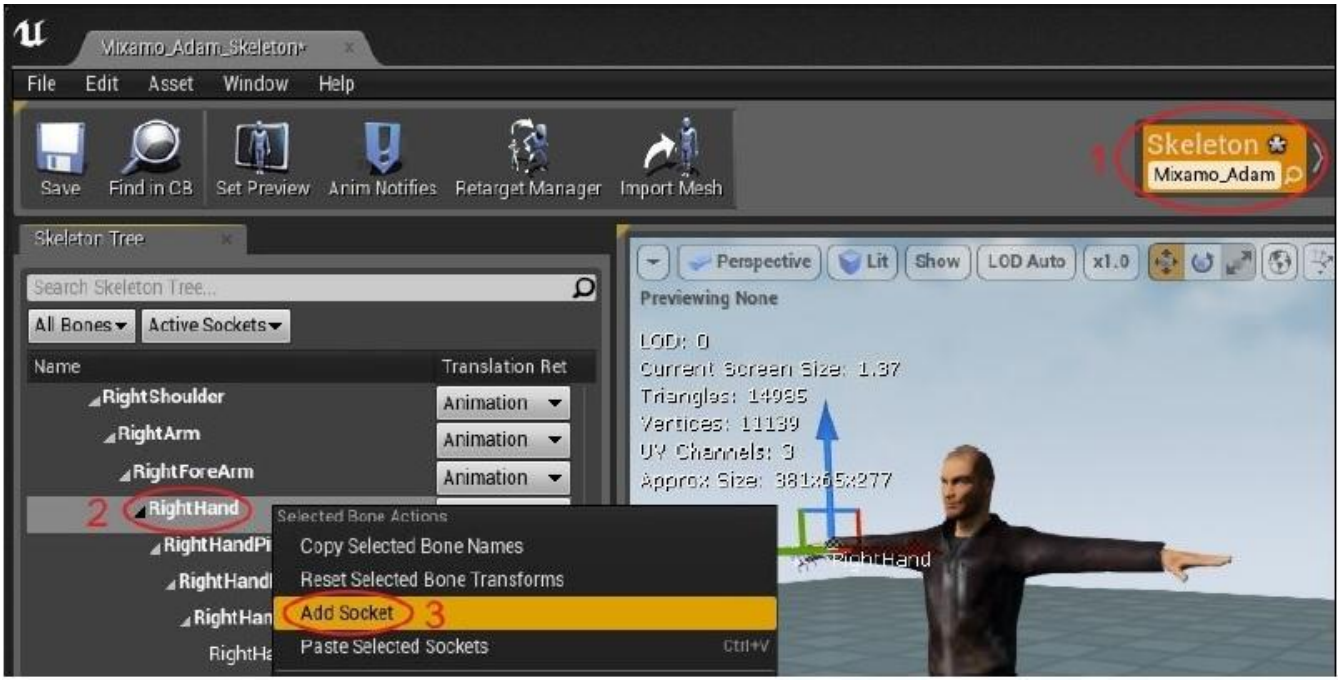
Чтобы добавить сокет к руке монстра, нам нужно отредактировать скелетную сетку, используемую монстром. И поскольку для монстра мы используем скелетную сетку Mixamo_Adam, нам нужно открыть и отредактировать эту сетку.

Для этого, двойной щелчок по скелетной сетке **Mixamo_Adam** во вкладке **Content Browser** (она появится в Т-позе), чтобы открыть редактор скелетной сетки. Если вы не видите **Mixamo_Adam** в вашей вкладке **Content Browser**, то убедитесь, что вы импортировали файл **Mixamo Animation Pack** в ваш проект из Unreal Launcher.



Редактируем Maximo_Adam двойным кликом по объекту скелетной сетки Maximo-Adam

Щёлкните по **Skeleton** в верхнем правом углу экрана. Прокрутите вниз дерево костей в панели слева, до **RightHand**. Мы прикрепим сокет к этой кости. Щёлкните правой кнопкой мыши по кости **RightHand** и выберите **Add Socket** (добавить разъём), как показано на следующем скриншоте:

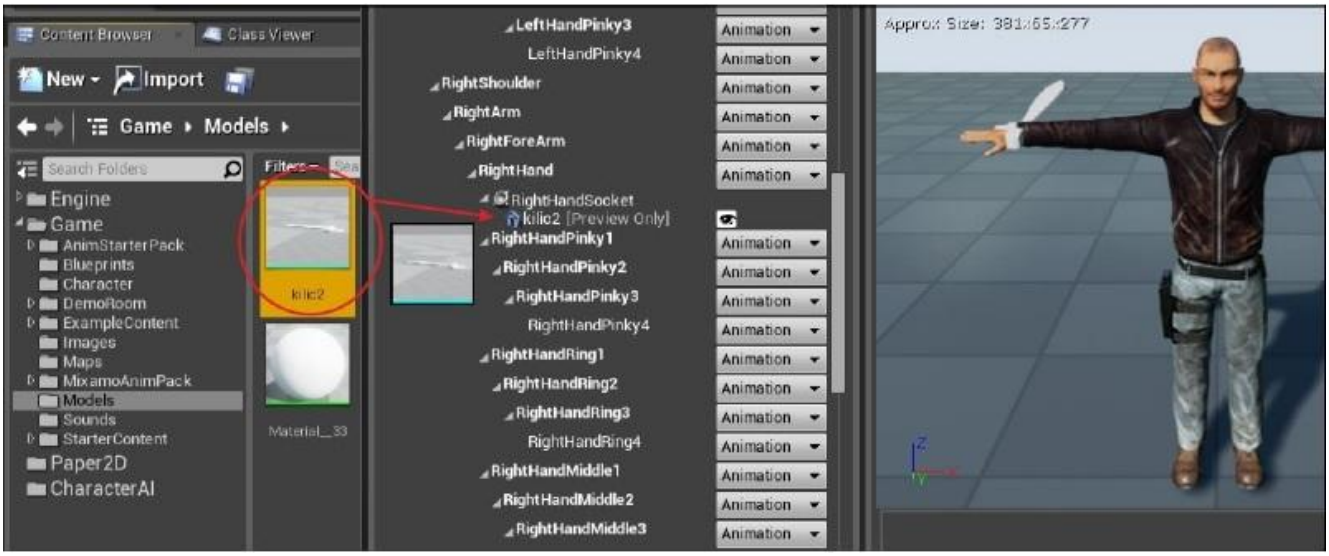


Вы можете оставить имя по умолчанию (**RightHandSocket**) или назвать по своему если хотите:

Далее нам нужно добавить меч в руку игрока.

Прикрепляем меч к модели

Открыв скелетную сетку Adam, найдите опцию RightHandSocket в дереве обзора. Так как Адам будет взмахивать своей правой рукой, вам нужно закрепить меч в его правой руке. Перетащите вашу модель меча в опцию RightHandSocket. Вы увидите, что Адам держит меч в изображении модели справа на следующем скриншоте:



Теперь, щёлкните по **RightHandSocket** и увеличьте для руки Адама. Нам нужно отрегулировать расположение сокета в предпросмотре так, чтобы меч подходил в него как следует. Подвиньте и поверните меч, чтобы поместить его в руке корректно.

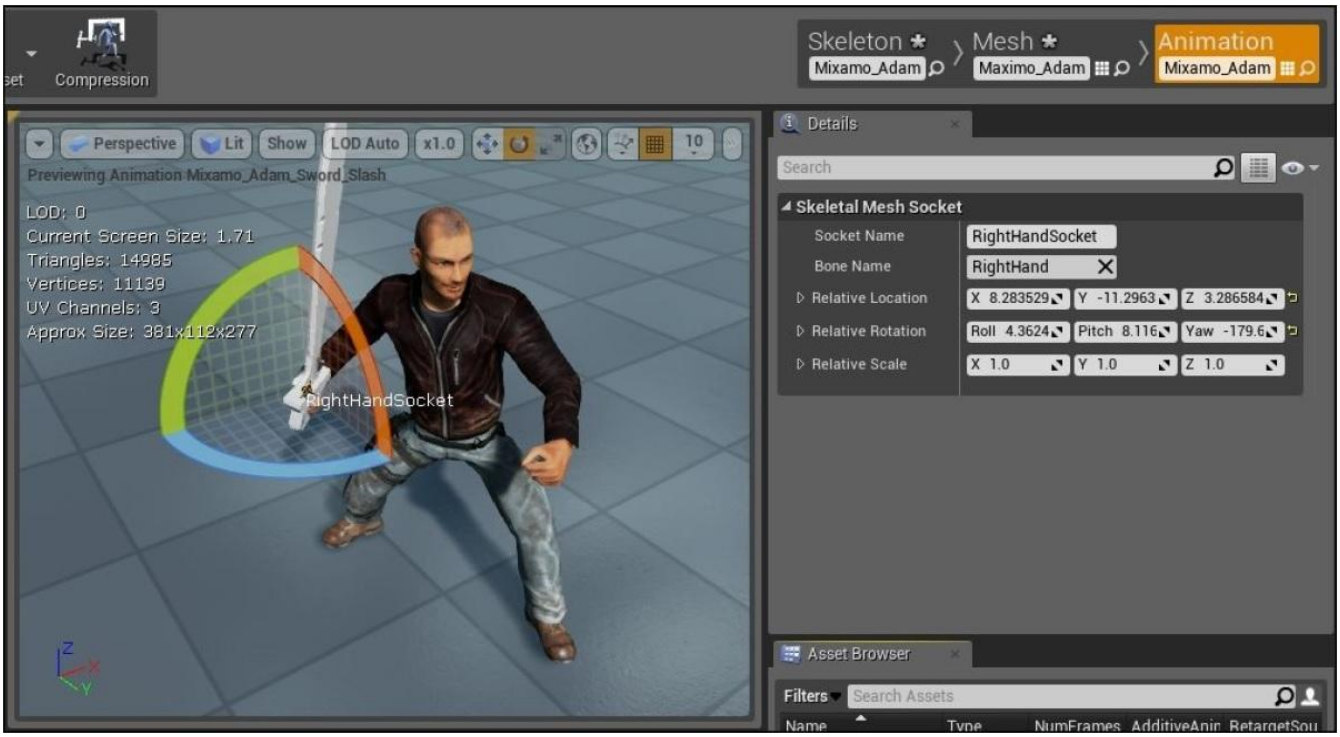
Располагаем сокет в правой руке так, чтобы меч лежал как следует

Совет

Совет от real-world

Если у вас несколько моделей меча, которые вы хотите менять постоянно в одном и том же сокете правой руки **RightHandSocket**, тогда вам будет нужно хорошо убедиться в единообразии (отсутствии отклонений от нормы), между разными мечами, предназначенными для одного сокета.

Вы можете предварительно просматривать свою анимацию с мечом в руке, перейдя во вкладку **Animation**, в правом верхнем углу экрана.



Вооружаем модель мечём

Тем не менее, если вы загружаете свою игру, у Адама не будет меча в руке. Это потому, что добавление меча к сокету *Персонажа*, идёт только в целях предварительного просмотра.

Код для оснащения игрока мечём

Чтобы оснастить игрока мечём через код и привязать меч к действующему лицу на постоянной основе, присвойте AMeleeWeapon экземпляр и прикрепите его на RightHandSocket после того как экземпляр монстра инициализирован. Мы делаем это в PostInitializeComponents(), так как в этой функции объект слияния был уже полностью инициализирован.

В файле Monster.h, добавьте метод, чтобы выбрать имя класса **Blueprint** (UClass) для оружия рукопашного боя. Также добавьте крюк для переменной, чтобы собственно хранить экземпляр MeleeWeapon, используя следующий код:

```
// Класс MeleeWeapon, который использует монстр
// Если он не установлен, то монстр использует просто рукопашную атаку
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
UClass* BPMeleeWeapon;

// Экземпляр MeleeWeapon (устанавливается если персонаж использует
// оружие для рукопашной схватки)
AActor* MeleeWeapon;
```

Теперь, выберите блупринт BP_MleeSword в вашем классе блупринта монстра.

В коде C++, вам нужно инициализировать оружие. Чтобы сделать это, нам нужно объявить и осуществить функцию PostInitializeComponents для класса Monster. В файле Monster.h добавьте объявление прототипа:

```
virtual void PostInitializeComponents() override;
```

PostInitializeComponents запускается после того, как завершается конструктор объекта и все компоненты объекта инициализированы (включая конструктор блупринта). Так это идеальное время, чтобы проверить прикреплён ли блупринт MeleeWeapon к монстру или нет, и если прикреплён, сделать экземпляр этого оружия. Следующий код добавляется, чтобы установить экземпляр оружия в осуществлении Monster.cpp для AMonster::PostInitializeComponents():

```
void AMonster::PostInitializeComponents()
{
    Super::PostInitializeComponents();

    // создаём экземпляр оружия, если блупринт был выбран
    if( BPMeleeWeapon )
    {
        MeleeWeapon = GetWorld()->SpawnActor<AMeleeWeapon>( BPMeleeWeapon, FVector(),
        FRotator() );

        if( MeleeWeapon )
        {
        const USkeletalMeshSocket *socket = Mesh->GetSocketByName( "RightHandSocket" );
        // убедитесь, что используете верное
        // имя сокета!
        socket->AttachActor( MeleeWeapon, Mesh );
        }
    }
}
```

Теперь монстры появляются с мечём в руке, если BPMeleeWeaon выбран для блупринта монстра.

Монстры держат оружие

Срабатывание анимации атаки

По умолчанию, нет связи между нашим C++ классом Monster и срабатыванию анимации атаки. Другими словами, класс MixamoAnimBP_Adam не имеет способа знать, когда монстр находится в состоянии атаки.

Поэтому, нам нужно обновить блупринт анимации скелета Адама (MixamoAnimBP_Adam), чтобы включить запрос в класс Monster, проходящий по переменным и проверяя, находится ли монстр в состоянии атаки. До этого мы не работали с блупринтами анимации (и вообще в целом с блупринтами) в этой книге, но следуйте шаг за шагом и вы увидите как всё складывается.

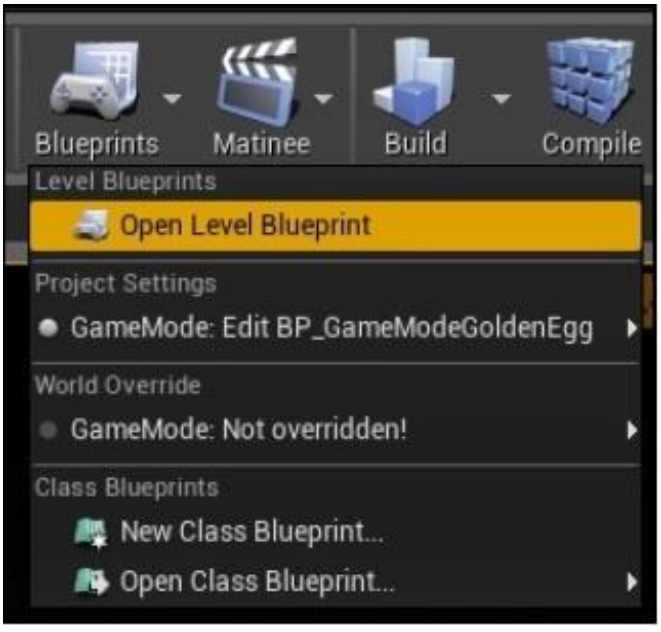
Основы blueprint

Блупринт UE4, это визуальная реализация кода (не путать с тем, когда люди говорят, что класс C++ это метафорически blueprint – схема, экземпляра класса). В блупринтах UE4 вместо самого написания кода, вы перетаскиваете элементы в график и соединяете их, чтобы достичь желаемой игры. Соединяя нужные узлы с нужными элементами, вы можете программировать всё, что хотите в своей игре.

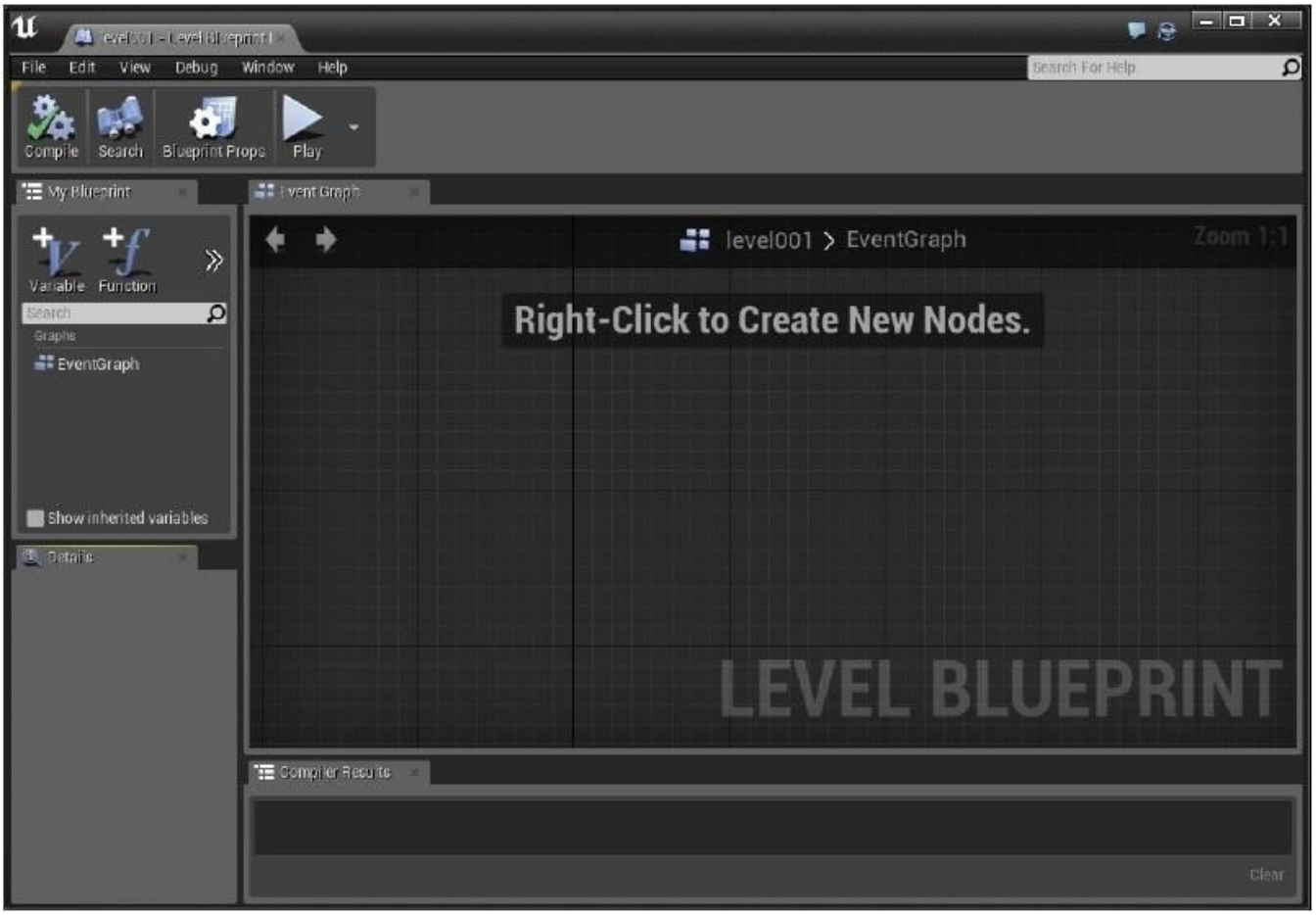
Подсказка

Эта книга не вдохновляет вас на использование блупринтов, так как вместо этого, мы стараемся вдохновить вас на написание вашего собственного кода. Тем не менее анимации наилучшим образом работают с блупринтами, потому что это то, что будет знать и художник и разработчик.

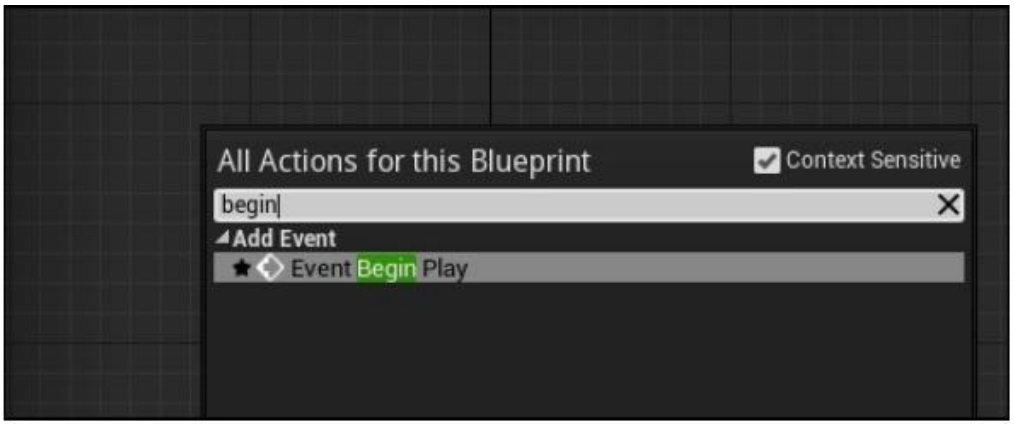
Давайте начнём писать примерный блупринт, чтобы почувствовать как они работают. Для начала щёлкните по вкладке мену блупринт сверху и выберите Open Level Blueprint, как показано на следующем скриншоте:



Опция Level Blueprint выполняется автоматически, когда вы начинаете уровень. Когда вы откроете окно, вы увидите пустой реестр для создания в нём игрового сюжета, как показано здесь:

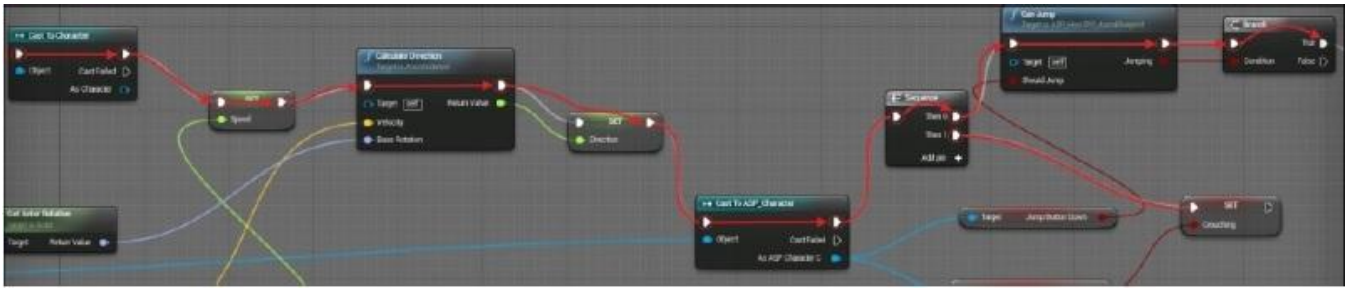


Щёлкните правой кнопкой мыши в любом месте графика. Начните печатать begin и в появившемся списке нажмите опцию **Event Begin Play**. Убедитесь, что стоит галочка на **Context Sensitive**, как показано на следующем скриншоте:



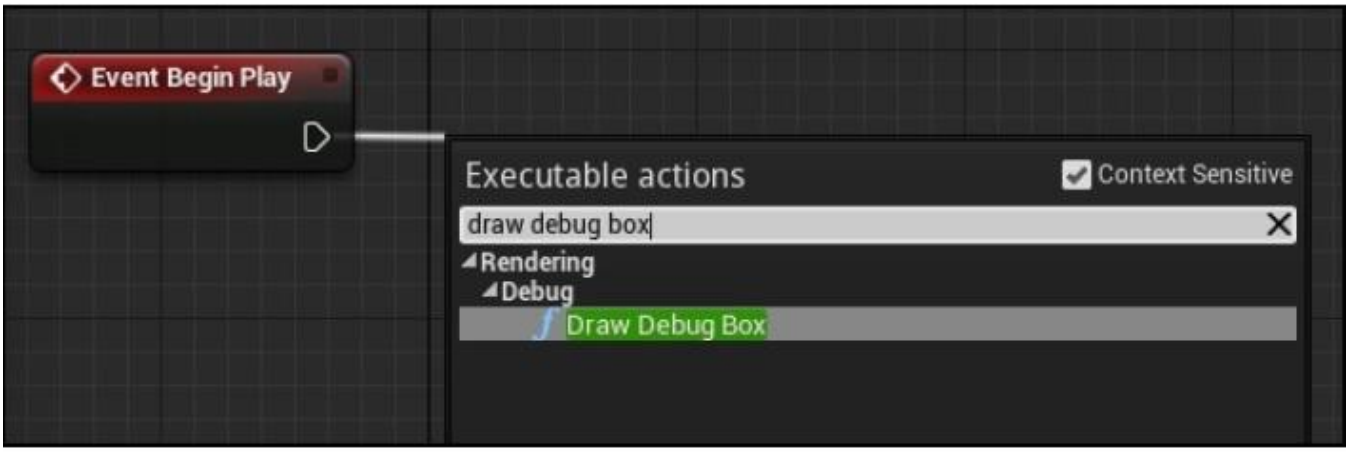
Как только вы нажмёте опцию Event Begin Play, сразу появится красный блок на вашем экране. У него будет один белый контакт справа. Это называется контакт исполнения (execution pin), как показано здесь:

Первое, о чём вам нужно знать в блупринтах анимации, это путь белого контакта выполнения (белая линия). Если вы видели графики блупринт раньше, то вы должно быть заметили, что белая линия проходит по графику, как показано на следующем скриншоте:

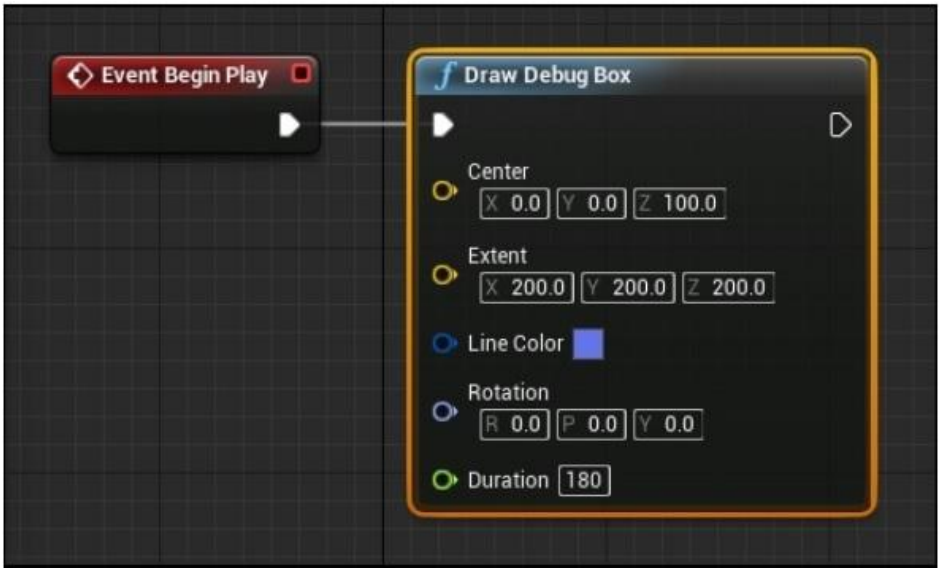


Белый путь контакта выполнения очень схож со строками кода выстроенными в линию и запускаемыми одна за другой. Белая линия определяет, какой узел будет выполнен и в каком порядке. Если узел не имеет прикреплённого белого контакта выполнения, то этот узел не будет выполняться вообще.

Потащите белый контакт выполнения из **Event Begin Play**. В диалоговом окне Executable actions (выполняемые действия) начните печатать draw debug box (нарисовать блок отладки). Выберите первое, что там появляется (**f Draw Debug Box**), как показано здесь:

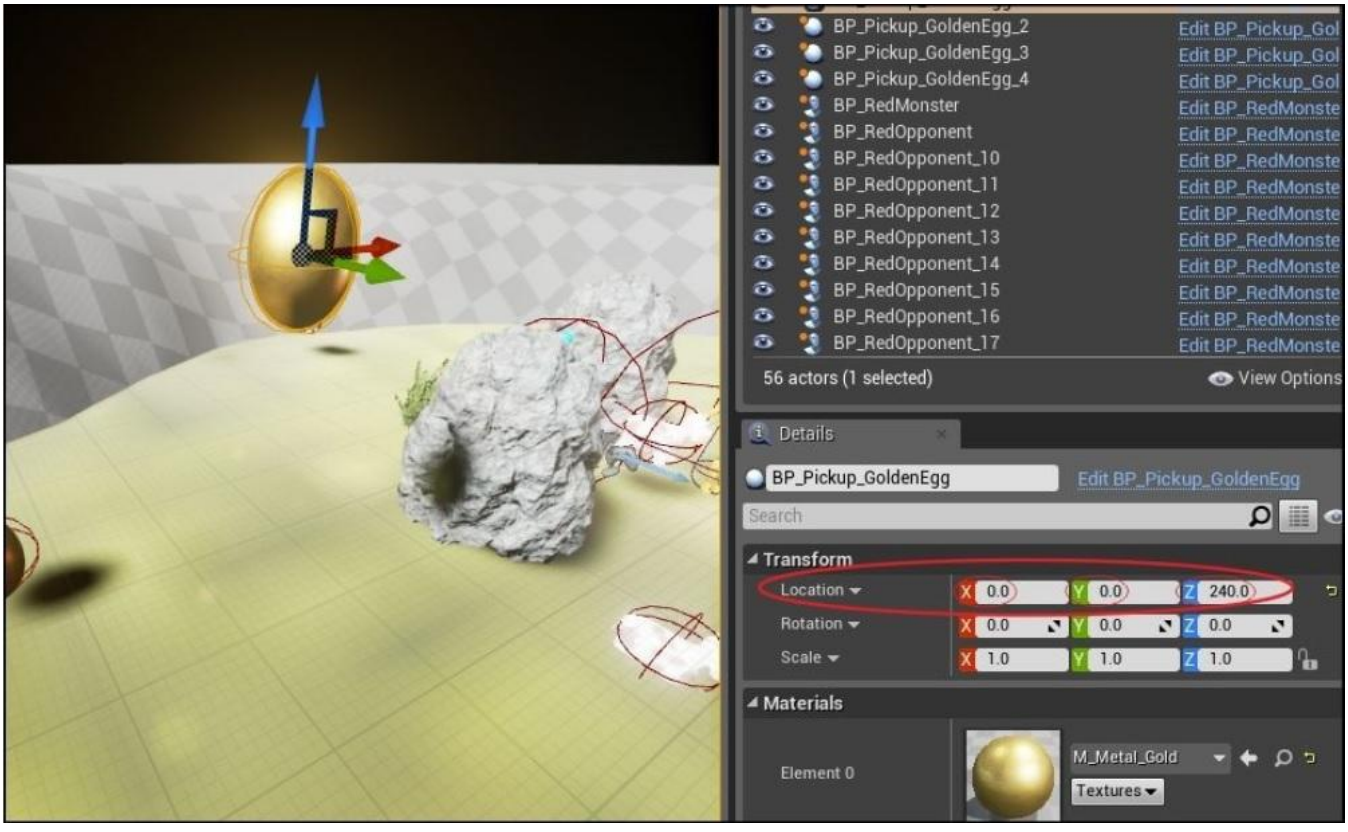


Заполните некоторые детали того, как вы хотели бы, чтобы выглядел блок. Я выбрал синий цвет для блока, центр блока (0, 0, 100), размер блока (200, 200, 200) и продолжительность (**Duration**) 180 секунд (обязательно введите продолжительность, которая достаточна для того, чтобы увидеть результат), как показано на следующем скриншоте:

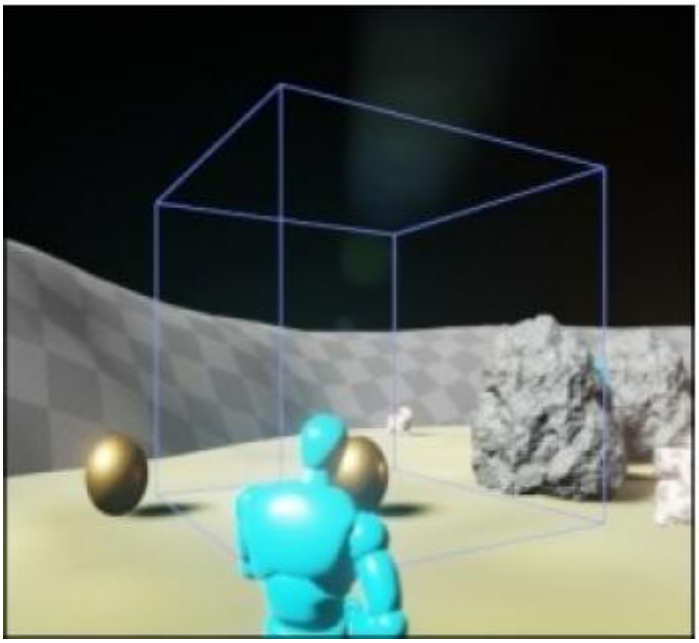


Теперь нажмите кнопку **Play**, чтобы реализовать график. Помните, что вам нужно найти начало отсчёта мировой системы координат, чтобы видеть блок отладки.

Чтобы найти начало координат, поместите золотое яйцо на (0, 0, (какое-нибудь значение z)), как показано на следующем скриншоте:



Вот как блок будет выглядеть в уровне:

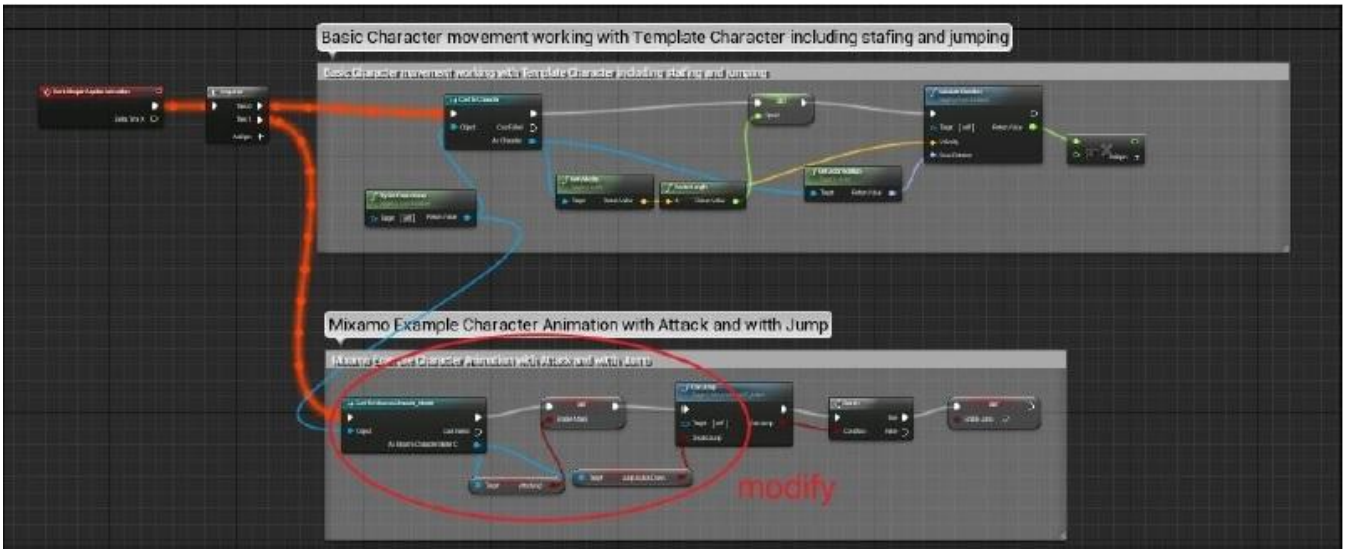


Блок отладки выведен в начале координат

Модифицируем блупринт анимации для Mixamo Adam

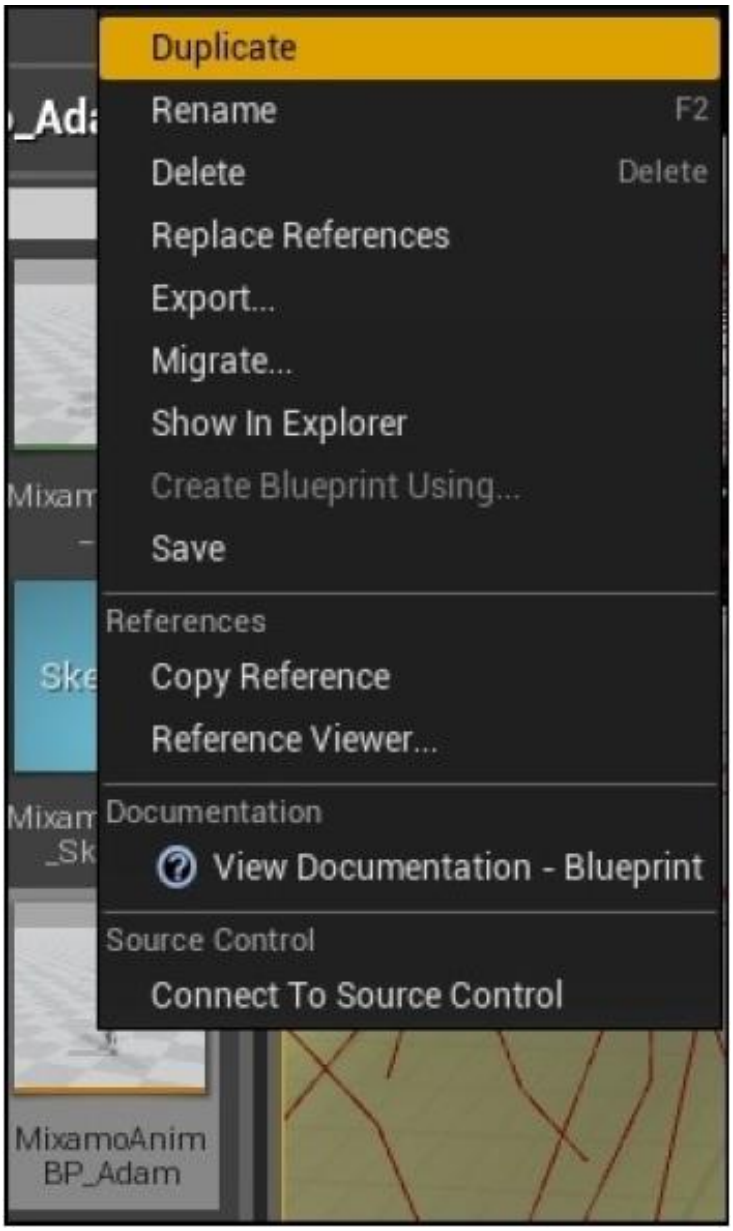
Чтобы интегрировать нашу анимацию атаки, нам нужно модифицировать блупринт. Под **Content Browser** откройте MixamoAnimBP_Adam.

Первое, что вы заметите, это то, что график имеет две секции: верхняя секция и нижняя секция. Верхняя секция отмечена как “**Basic Character movement…**”, в то время как нижняя секция говорит “**Mixamo Example Character Animation…**”. Базовое движение персонажа отвечает за ходьбу и бег модели. Работать в секции **Mixamo Example Character Animation with Attack and Jump**, которая ответственна за анимацию атаки. Мы будем работать о второй секции графика, показанной на следующем скриншоте:



Когда вы впервые открываете график, то сначала это с увеличение масштаба, в разделе снизу. Чтобы прокручивать вверх, правый щелчок мыши и тащим вверх. Вы также можете уменьшать масштаб, используя колёсико мыши, либо удерживая клавишу *Alt* и правую кнопку мыши двигая мышь вверх.

Перед тем как продолжить, вы вероятно захотите дублировать ресурс **MixamoAnimBP_Adam**, чтобы не повредить оригинал, в случае если позже вам понадобится вернутся и что-либо исправить. Это позволяет вам легко возвращаться и исправлять что-нибудь, если вы обнаружите, что сделали ошибку в одной из ваших модификаций, без надобности переустанавливать свежую копию всего пакета анимации в ваш проект.



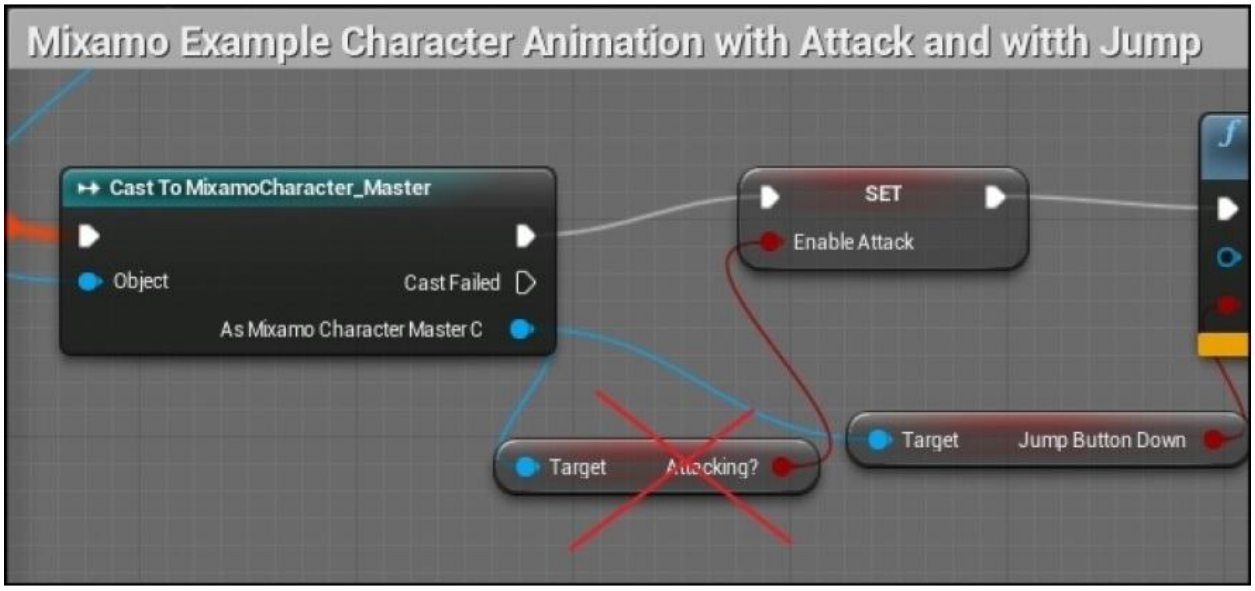
Делаем дубликат ресурса MixamoAnimBP_Adam, во избежание повреждений оригинала ассета

Совет

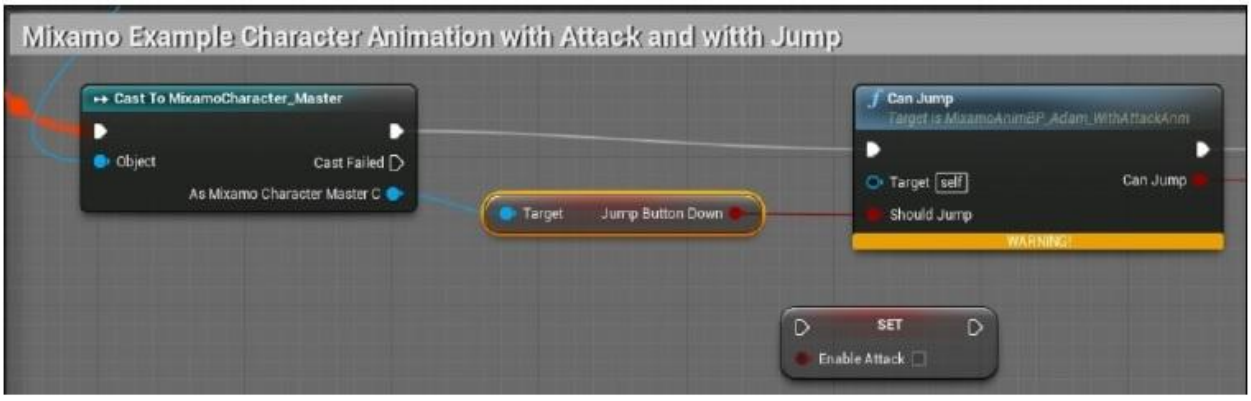
Когда ассеты добавлены в проект из Unreal Launcher, делается копия оригинала ассета, так что сейчас вы можете модифицировать **MixamoAnimBP_Adam** в вашем проекте и позже получать свежую копию оригинала ассета в проекте.

Мы собираемся сделать только пару вещей, чтобы заставить Адама махать мечём, когда он атакует. Давайте сделаем это по порядку.

1. Удаляем узел (node), который говорит **Attacking?** (атаковать?):

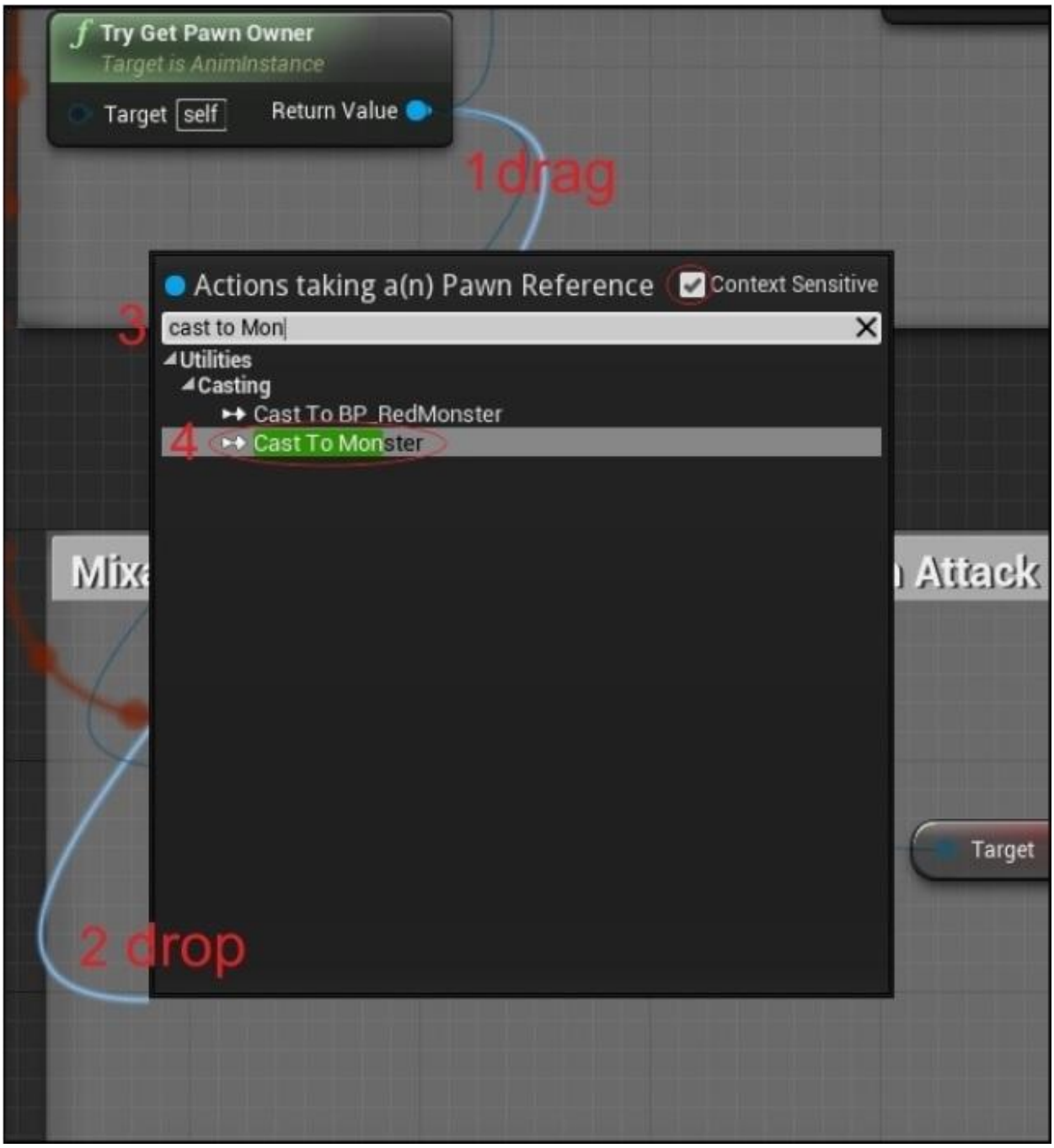


2. Переорганизовываем узлы следующим образом, узел **Enable Attack** (включить атаку) внизу:

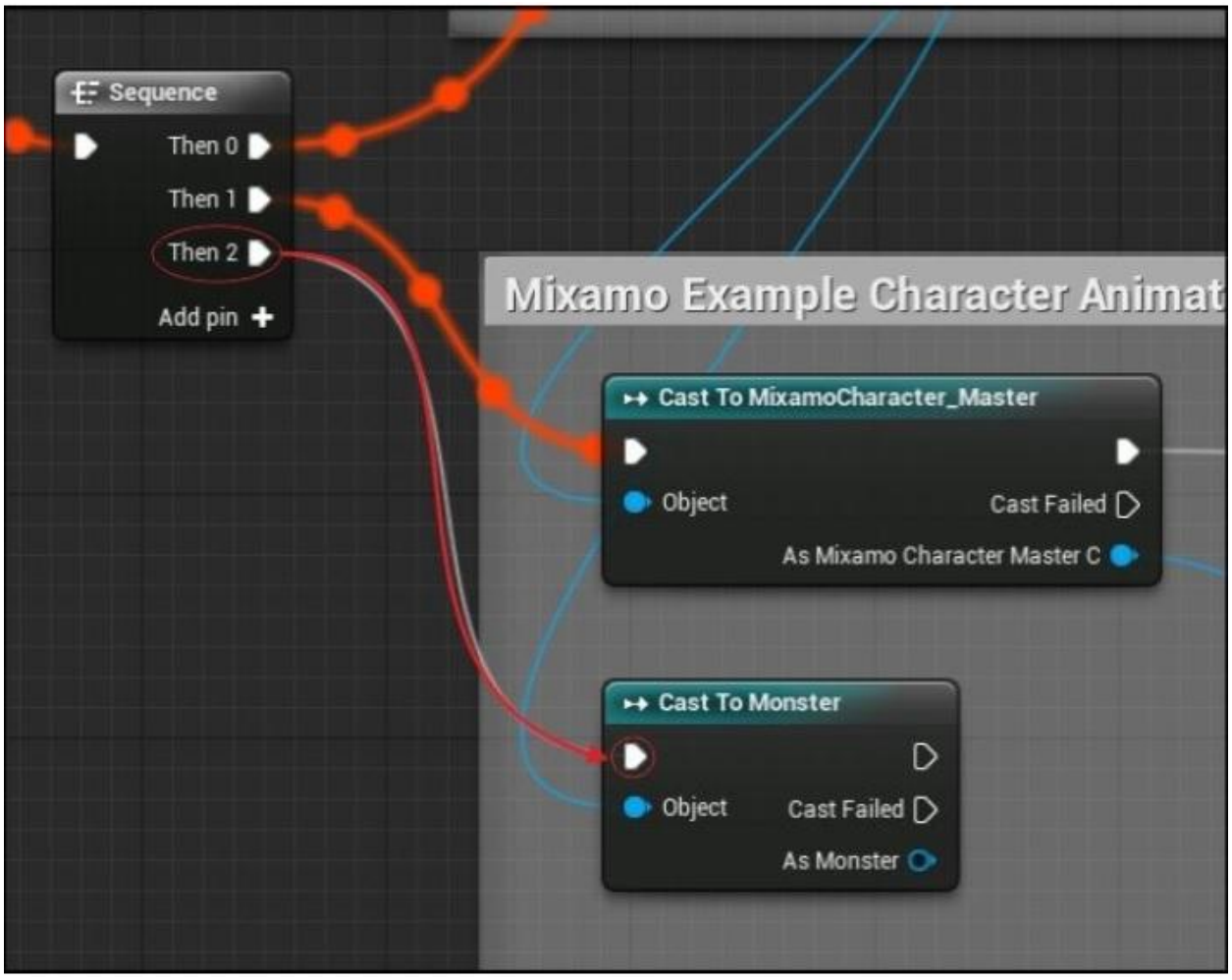


3. Далее мы поработаем с монстром, чтобы анимировать его анимацию. Прокрутите график немного вверх и ищите голубую точку отмеченную как **Return Value** – Возвратное Значение, в диалоговое окно **Try Get Pawn Owner** (Пробуем Получить Владельца Пешки). Бросьте это в своём графике и когда

появится всплывающее меню, выберите **Cast to Monster** (Привести Монстру) (убедитесь, что **Context Sensitive** отмечено галочкой, иначе опция **Cast to Monster** не покажется). Опция Try Get Pawn Owner поучает экземпляр монстра, который обладает анимацией, которая в свою очередь просто объект класса AMonster, как показано на следующем скриншоте:

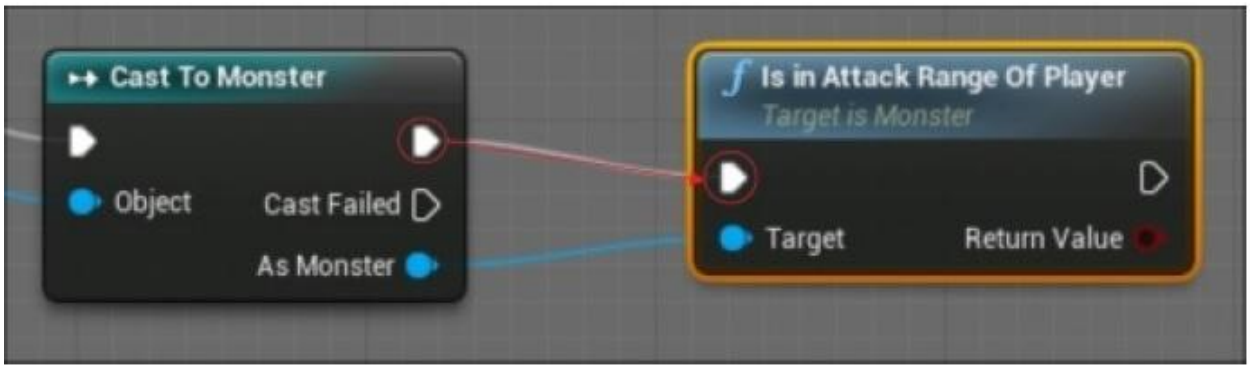


4. Кликните по значку **+** в окне **Sequence** (Последовательность) и перетащите ещё один контакт выполнения из группы **Sequence** в экземпляр узла **Cast to Monster**, как показано на следующем скриншоте. Это гарантирует, что экземпляр **Cast to Monster** будет выполнен.



5. Следующий шаг для того, чтобы вытянуть контакт из терминала **As Monster** узла **Cast to Monster** и найти свойство **Is in Attack Range Of Player** (находится ли на расстоянии таки игрока):

6. Возьмите белый контакт выполнения в узле **Cast to Monster** слева и перетащите его в узел **Is in Attack Range of Player** справа. Что обеспечит передачу контроля операции **Cast to Monster** в узел **Is in Attack Range Of Player**:



7. Вытяните красный и белый контакты в узел **SET**, как показано здесь:

Подсказка

Эквивалентный псевдокод предыдущего блупринта (схемы), это что похожее на это:

```
if( Monster.isInAttackRangeOfPlayer() )
{
   Monster.Animation = The Attack Animation;
}
```

Протестируйте свою анимацию. Монстр должен делать взмахи только, когда он находится в досягаемости к игроку.

Код для взмахов мечём

Мы хоти добавить событие извещения анимации, когда мечём махнули. Сначала объявим и добавим C++ функцию, которую можно вызывать из блупринта, в ваш класс Monster:

```
// в Monster.h:
UFUNCTION( BlueprintCallable, Category = Collision )
void SwordSwung();
```

Утверждение BlueprintCallable означает, что эту функцию возможно вызывать из блупринтов. Другими словами, SwordSwung() (МечёмМахнули) будет C++ функцией, которую мы можем запускать из узла блупринта, как показано здесь:

```
// в Monster.cpp
void AMonster::SwordSwung()
{
    if( MeleeWeapon )
    {
        MeleeWeapon->Swing();
    }
}
```

Далее откройте анимацию **Mixamo_Adam_Sword_Slash**, дважды щёлкнув по ней в вашем **Content Browser** (это должно быть в **MixamoAnimPack/Mixamo_Adam/Anims/Mixamo_Adam_Sword_Slash**). Отодвиньте анимацию до того места, где Адам начинает махать своим мечём. Щёлкните правой кнопкой мыши по полосе анимации и выберите **New Notify** (новое уведомление) под **Add Notify…**, как показано здесь:

Назовите уведомление SwordSwung:

Имя уведомления должно появиться в вашей временной шкале, следующим образом:

Сохраните анимацию и затем снова откройте свою версию **MixamoAnimBP_Adam**. Под группой узлов **SET** создайте следующий график:

Узел **AnimNotify_SwordSwung** появляется, когда вы щёлкаете правой кнопкой мыши в графике (с включенным **Context Sensitive**) и начинаете печатать SwordSwung. Узел **Cast To Monster** снова запитан от узла **Try Get Pawn Owner**, как в шаге 2 раздела *Модифицируем блупринт анимации для Mixamo Adam*. И наконец наша C++ функция вызываемая из блупринта в классе AMonster.

Если вы начнёте игру сейчас, ваши монстры будут уже выполнять свою анимацию атаки. Когда ограничивающий прямоугольник меча придёт в контакт с вами, вы увидите, что ваша полоса HP убывает понемногу (вспомните, что полоса HP была добавлена в конце Главы 8. *Действующие лица и пешки*, в качестве упражнения).

Монстры атакуют игрока

Снаряды или дальняя атака

Атака с расстояния обычно включает какого-либо рода снаряды. Снаряды – это например, пули, но не только. Они могут также представлять из себя атаки магическими разрядами молнии или огненным шаром. Чтобы спрограммировать атаку снарядами, вам нужно породить новый объект и применять урон к игроку, только если снаряд достигает игрока.

Чтобы осуществить базовые пули в UE4, нам следует выполнить происхождение объекта нового типа. Я выполнил происхождение класса ABullet от класса AActor, как показано в следующем коде:

```
UCLASS()
class GOLDENEGG_API ABullet : public AActor
{
    GENERATED_UCLASS_BODY()

    // Сколько урона наносит пуля.
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Properties)
    float Damage;

    // Видимая сетка – Meshдля компонента, чтобы мы могли
    // стреляющий объект
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision)
    UStaticMeshComponent* Mesh;
```

```
    // сфера, с которой вы сталкиваетесь, чтобы нанести урон
    UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision)
    USphereComponent* ProxSphere;
    UFUNCTION(BlueprintNativeEvent, Category = Collision)
    void Prox( AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
    bFromSweep, const FHitResult & SweepResult );
};
```

Класс ABullet имеет пару важных элементов:

- Переменная типа float для урона, который наносит пуля при контакте
- Переменная Mesh для тела пули
- Переменная ProxSphere, чтобы определять, когда пуля попала во что-то
- Функция для запуска, когда Prox замечен рядом с объектом

Конструктор для класса ABullet должен иметь инициализацию переменной Mesh и переменной ProxSphere. В конструкторе мы устанавливаем, что RootComponent будет переменная Mesh, а затем добавим переменную ProxSphere к переменной Mesh. Переменная ProxSphere будет использоваться для проверки столкновения, а проверка столкновения для переменной Mesh должна быть выключена, как показано в следующем коде:

```
ABullet::ABullet(const class FObjectInitializer& PCIP) : Super(PCIP)
{
    Mesh = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this, TEXT("Mesh"));
    RootComponent = Mesh;

    ProxSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this, TEXT("ProxSphere"));
    ProxSphere->AttachTo( RootComponent );

    ProxSphere->OnComponentBeginOverlap.AddDynamic( this, &ABullet::Prox );
    Damage = 1;
}
```

В конструкторе мы присвоили начальное значение 1, переменной Damage (урон). Но его можно изменять в редакторе UE4, как только мы создадим блупринт класса ABullet. Далее функция ABullet::Prox_Implementation() должна работать с уроном актора, получившего удар, если мы сталкиваемся с другим актором от RootComponent, используя следующий код:

```
void ABullet::Prox_Implementation( AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult &SweepResult )
{
    if( OtherComp != OtherActor->GetRootComponent() )
    {
        // не сталкиваемся ни с чем другим
        // кроме актора root component
        return;
    }

    OtherActor->TakeDamage( Damage, FDamageEvent(), NULL, this );
    Destroy();
}
```

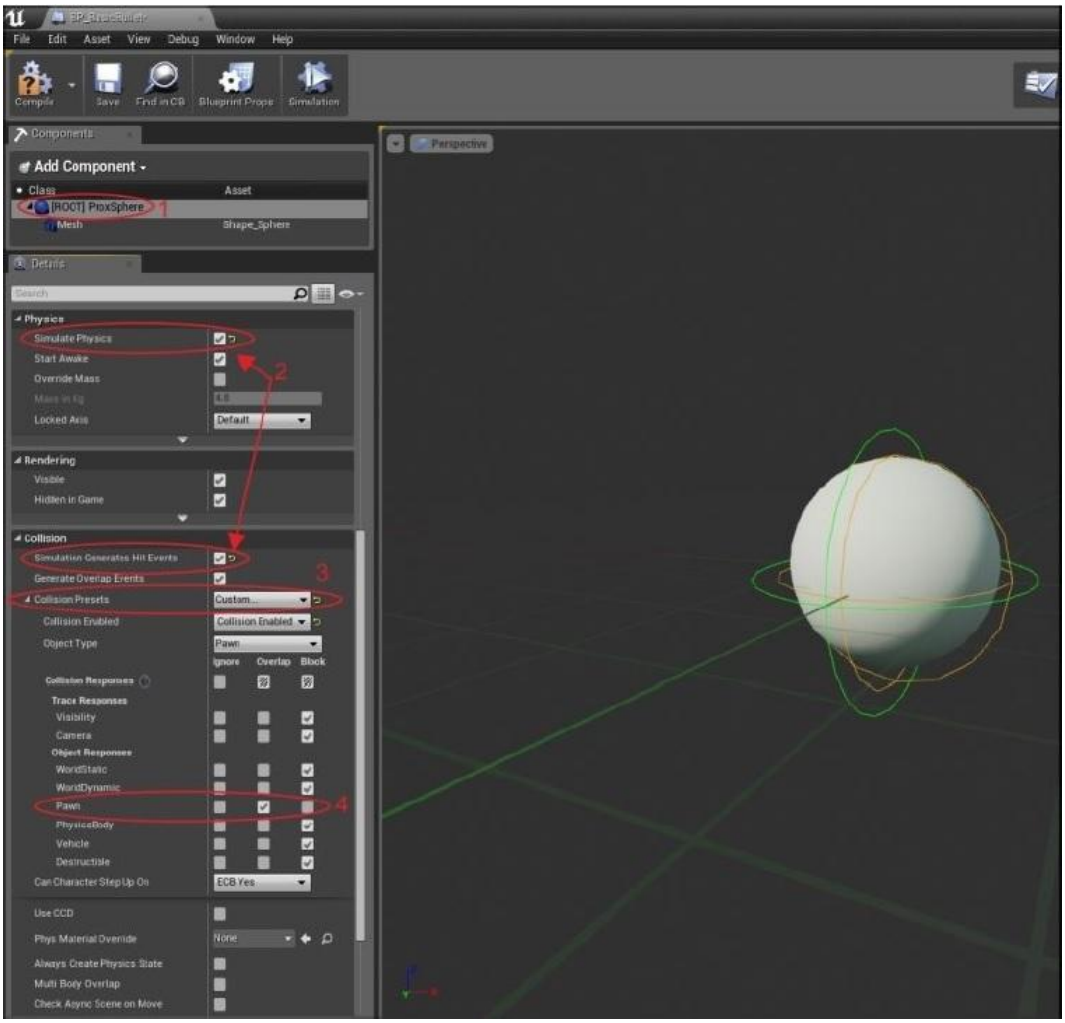
Физика пули

Чтобы заставить пулю лететь по уровню, вы можете использовать физический движок UE4.

Создайте блупринт основанный на классе ABullet. Я выбрал **Shape_Sphere** (форма_сферы) для сетки. У сетки пули не должно быть физики столкновения, вместо этого у нас будет физика столкновения на ограничивающей сфере пули.

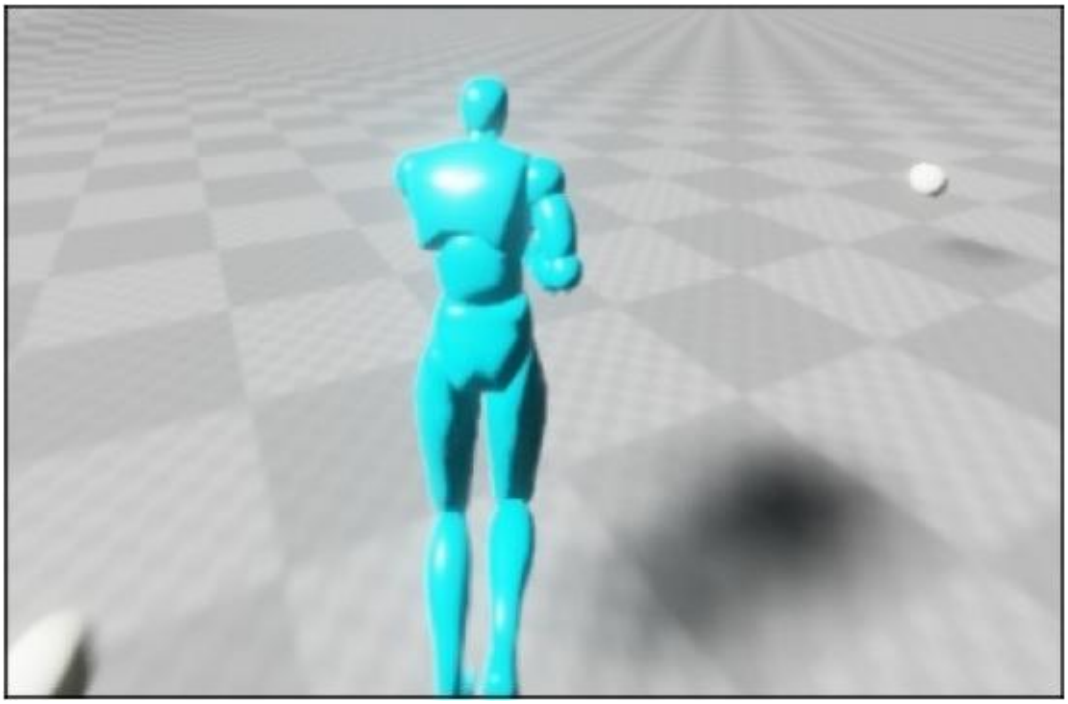
Конфигурируем пулю для качественного поведения, что немного замысловато. Выполняем это в следующих шагах:

1. Выберите **[ROOT] ProxSphere** во вкладке **Components**. Переменная ProxSphere должна быть корневым компонентом и должна быть вверху иерархии.
2. Во вкладке **Details** ставим галочку и на **Simulate Physics** и на **Simulation Generates Hit Events**.
3. В выпадающем списке **Collision Presets** выберите **Custom…**.
4. Отметьте ячейки **Collision Responses** как показано; отметьте **Block** для большинства типов (**WorldStatic**, **WorldDynamic** и так далее) и отметьте **Overlap** только для **Pawn**:



Галочка на **Simulate Physics** даёт свойству ProxSphere опыт гравитации и импульсных сил влияющих на него. Импульс это мера воздействия силы на тело за данный промежуток времени, которую мы будем использовать для управления выстрелом пули. Если вы не поставите галочку на **Simulation Generate Hit Events**, то пуля просто упадёт на землю. А **BlockAll Collision Preset** не даёт пуле пролетать насквозь при попадании во что-нибудь.

Если вы сейчас перетащите пару этих объектов BP_Bullet из вкладки **Content Browser** прямо в мир, то они пока будут просто падать на землю. Вы можете пнуть их, когда они на земле. Следующий скриншот демонстрирует шары представляющие пули на земле:



И однако, мы не хотим, чтобы наши поли падали на землю. Мы хотим, чтобы они летели при выстреле. Так что давайте поместим наши пули в класс Monster.

Добавление пуль в класс Monster

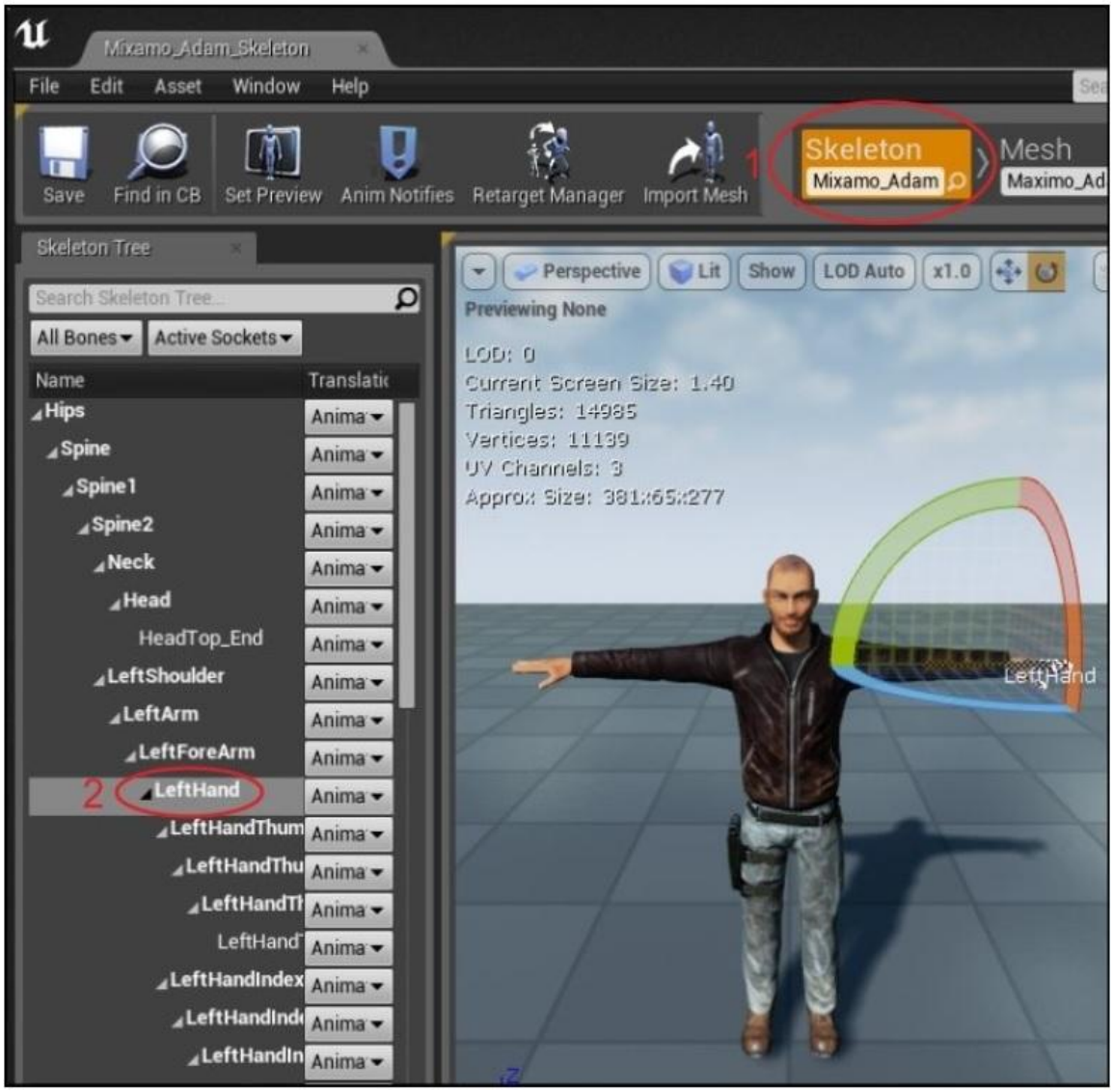
Добавьте элемент в класс Monster, который получает ссылку экземпляра блупринта. Вот для чего нужен объектный тип UClass. Также добавьте свойство float конфигурируемого блупринта, чтобы регулировать силу с которой выстреливается пуля, как показано в следующем коде:

```
// Блупринт класса пули, который использует монстр
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
UClass* BPBullet;
// Тяга позади запуска пули
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MonsterProperties)
float BulletLaunchImpulse;
```

Компилируйте и запустите C++ проект и откройте ваш блупринт BP_Monster. Теперь вы можете выбрать класс блупринта для BPBullet, как показано на следующем скриншоте:

Когда вы выберите классовый тип блупринта, чтобы создать экземпляр, когда монстр стреляет, то вам нужно будет спрограммировать, чтобы монстр стрелял, когда игрок находился в его досягаемости.

Откуда стреляет монстр? Вообще то он должен стрелять из кости. Если вы не знакомы с терминологией, кости это просто соответствующие точки на сетке модели. Сетка модели как правило выполнена из многих “костей”. Чтобы увидеть кости, откройте сетку **Mixamo_Adam** дважды щёлкнув по ассету во вкладке **Content Browser**, как оказано на следующем скриншоте:



Перейдите во вкладку Skeleton и вы увидите кости монстра в списке дерева обзора слева. Мы хотим выбрать кость из которой будут вылетать пули. Я здесь выбрал LeftHand.

Подсказка

Художник как правило введёт дополнительную кость в сетку модели, для выпуска пуль, которая как правило будет на конце ствола оружия.

Работая из базовой сетки модели, мы можем получить местоположение костей Mesh, и в коде заставить монстра выпускать экземпляры Bullet из кости.

Готовые функции монстра Tick и Attack могут быть получены при использовании следующего кода:

```
void AMonster::Tick(float DeltaSeconds)
{
    Super::Tick( DeltaSeconds );

    // двигаем монстра на игрока
    AAvatar *avatar = Cast<AAvatar>(
    UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
    if( !avatar ) return;
    FVector playerPos = avatar->GetActorLocation();
    FVector toPlayer = playerPos - GetActorLocation();
    float distanceToPlayer = toPlayer.Size();

    // если игрок за пределами SightSphere монстра,
    // возвращаемся
    if( distanceToPlayer > SightSphere->GetScaledSphereRadius() )
    {
        // Если игрок это OS, то монстр не может за ним гнаться
        return;
    }

    toPlayer /= distanceToPlayer; // нормализуем вектор

    // Обращение лицом к цели
    // Получаете ротатор для поворачивания того,
    // что смотрит в нправлении игрока `toPlayer`
    FRotator toPlayerRotation = toPlayer.Rotation();
    toPlayerRotation.Pitch = 0; // 0 от тангажа
    RootComponent->SetWorldRotation( toPlayerRotation );

    if( isInAttackRange(distanceToPlayer) )
    {
        // Выполняем атаку
        if( !TimeSinceLastStrike )
        {
            Attack(avatar);
        }

        TimeSinceLastStrike += DeltaSeconds;
```

```
        if( TimeSinceLastStrike > AttackTimeout )
        {
            TimeSinceLastStrike = 0;
        }
        return; // больше ничего не делаем
    }
    else
    {
        // не в зоне досягаемости атаки, тогда идём к игроку
        AddMovementInput(toPlayer, Speed*DeltaSeconds);
    }
}
```

Функция AMonster::Attack относительно проста. Конечно, нам сначала нужно добавить объявление прототипа в файле Monster.h, чтобы написать нашу функцию в файле .cpp:

```
void AMonster::Attack(AActor* thing);
```

В Monster.cpp мы осуществляем функцию Attack, следующим образом:

```
void AMonster::Attack(AActor* thing)
{
    if( MeleeWeapon )
    {
        // код для взмахов рукопашным оружием, если
        // рукопашное оружие используется
        MeleeWeapon->Swing();
    }
    else if( BPBullet )
    {
        // If a blueprint for a bullet to use was assigned,
        // then use that. Note we wouldn't execute this code
        // bullet firing code if a MeleeWeapon was equipped
        FVector fwd = GetActorForwardVector();
        FVector nozzle = GetMesh()->GetBoneLocation( "RightHand" );
        nozzle += fwd * 155;// двигаем её от монстра, так что
        // она не сталкивается с моделью монстра
        FVector toOpponent = thing->GetActorLocation() - nozzle;
        toOpponent.Normalize();
        ABullet *bullet = GetWorld()->SpawnActor<ABullet>( BPBullet, nozzle, RootComponent-
        >GetComponentRotation());

        if( bullet )
        {
            bullet->Firer = this;
            bullet->ProxSphere->AddImpulse( fwd*BulletLaunchImpulse );
        }
        else
        {
        GEngine->AddOnScreenDebugMessage( 0, 5.f,
        FColor::Yellow, "monster: no bullet actor could be spawned. is the bullet overlapping something?"
        );
        }
    }
}
```

Мы оставляем код, который осуществляет рукопашную атаку. Предполагая, что монстр не держит оружие для рукопашной атаки, мы затем проверяем установлен ли элемент BPBullet. Если этот элемент установлен, то это значит, что монстр создаст экземпляр BPBullet из блупринта класса и будет стрелять.

Обратите особое внимание на следующую строку:

```
ABullet *bullet = GetWorld()->SpawnActor<ABullet>(BPBullet, nozzle, RootComponent-
>GetComponentRotation() );
```

Это то, как мы добавляем новый актор в мир. Функция SpawnActor() помещает экземпляр UCLASS, который вы передали, в spawnLoc, с некоторыми начальными ориентировками.

После того, как мы порождаем пулю, мы вызываем функцию AddImpulse() к её переменной ProxSphere, чтобы запускать её вперёд.

Отбрасывание игрока

Чтобы добавить игроку отбрасывание, я добавил переменную член в класс Avatar и назвал её knockback – отбрасывание. Отбрасывание происходит, когда аватар получает ранение:

```
FVector knockback; // в классе AAvatar
```

Перед тем как обдумать направление отбрасывание игрока, когда он получает удар, нам нужно добавить код к AAvatar::TakeDamage. Вычисляем направление вектора со стороны атакующего к игроку и сохраняем этот вектор в переменной knockback:

```
float AAvatar::TakeDamage(float Damage, struct FDamageEvent const&
DamageEvent, AController* EventInstigator, AActor* DamageCauser)
{
    // добавляем отбрасывание, которое применятся в нескольких кадрах
    knockback = GetActorLocation() - DamageCauser- >GetActorLocation();
    knockback.Normalize();
    knockback *= Damage * 500; // отрасывание пропорциональное урону
}
```

В AAvatar::Tick, мы применяем отбрасывание к положению аватара:

```
void AAvatar::Tick( float DeltaSeconds )
{
    Super::Tick( DeltaSeconds );

    // применяем вектор отбрасывания
    AddMovementInput( knockback, 1.f );

    // делим пополам объём отбрасывания каждый кадр
    knockback *= 0.5f;
}
```

Поскольку вектор отбрасывания уменьшается каждым кадром, слабеет со временем, пока вектор отбрасывания не обновится с очередным ударом.

Выводы

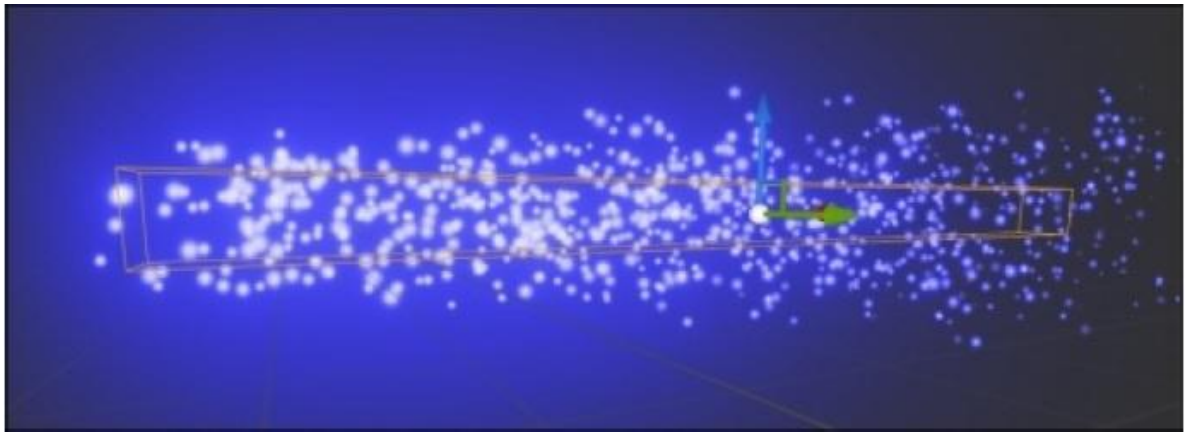
В этой главе, мы исследовали, как создавать экземпляры монстров на экране, которые бегают за игроком и нападают на него. В следующей главе, мы дадим игроку способность обороняться, позволяя ему посылать заклинания, которые нанося урон монстрам.

Глава 12. Книга заклинаний

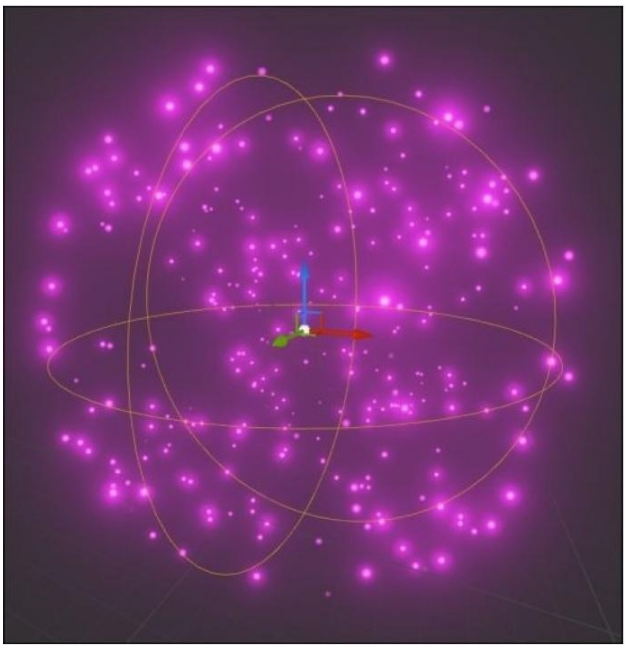
У игрока пока ещё нет средств обороны. Мы снабдим игрока очень полезным и интересным способом, магическими заклинаниями. Магические заклинания будут применяться игроком, чтобы влиять на монстров поблизости.

На практике, заклинания будут комбинацией системы частиц и область действия, представленной огранивающим объёмом. Ограничивающий объём проверяется на содержание акторов в каждом кадре. Когда актор оказывается внутри ограничивающего объёма заклинания, то на этот актор будет оказываться действие заклинания.

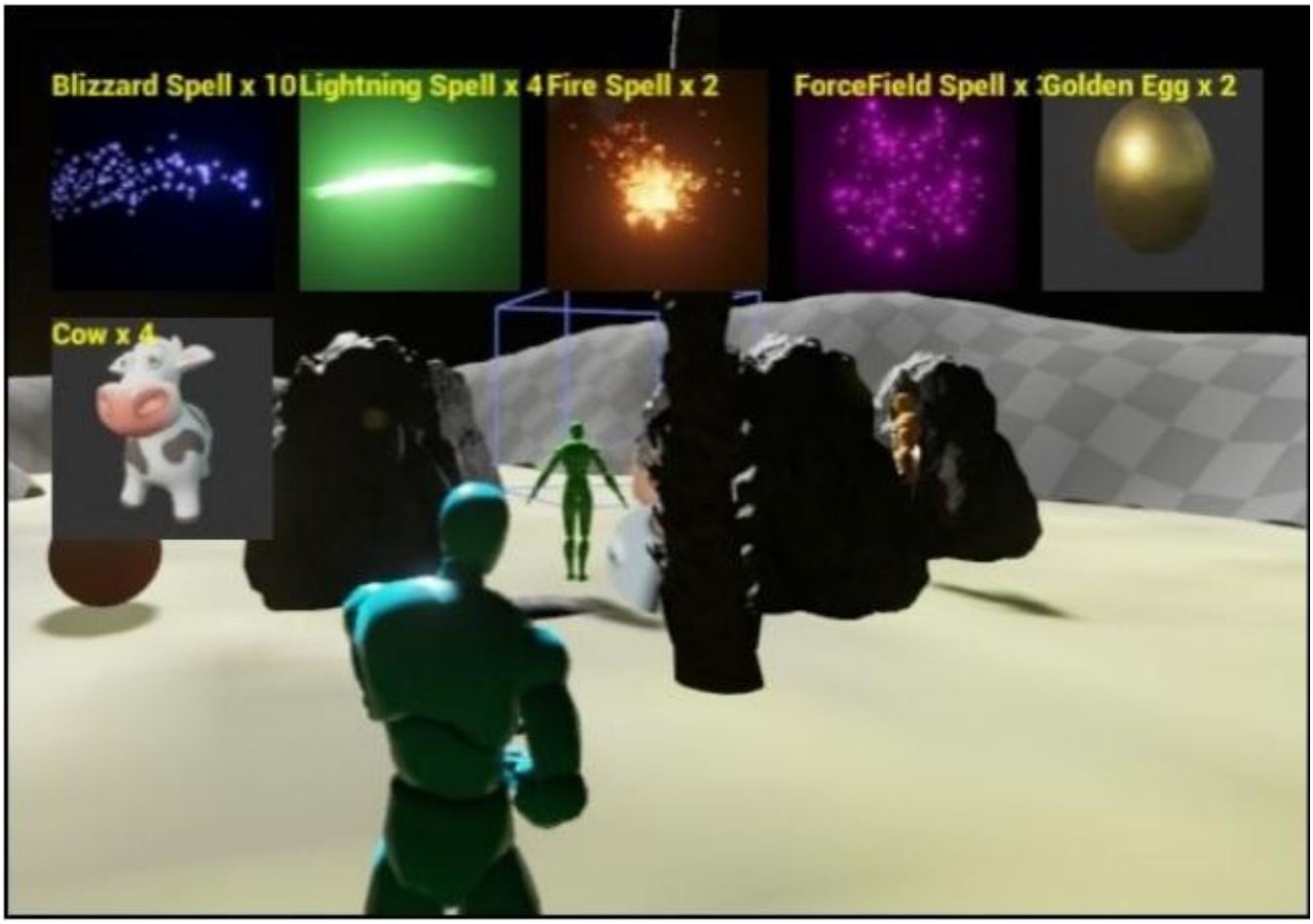
Следующий скриншот демонстрирует метель и поле силы заклинания, с его ограничивающим объёмом, который выделен оранжевым цветом:



Визуализацию заклинания метели, можно видеть сверху, с длинным, в форме прямоугольника ограничивающим объёмом. Визуализация поля силы заклинания, со сферическим ограничивающим объёмом для отталкивания монстров, показана на следующем скриншоте:



В каждом кадре, ограничивающий объём проверяется на содержание акторов. Любой актор находящийся в ограничивающем объёме заклинания, будет попадать под действие этого заклинания только для этого кадра. Если актор выходит из ограничивающего объёма заклинания, то он больше не будет попадать под действие этого заклинания. Запомните, что система частиц заклинания это всего лишь визуализация. Сами частицы не то, что будет влиять на акторов игры. Класс PickupItem, который мы создали в Главе 8. *Действующие лица и пешки*, может использоваться, чтобы позволить игроку брать предметы представляющие заклинания. Мы расширим класс PickupItem и добавим блупринт заклинания, чтобы приводить каждый PickupItem. Клик по графическому элементу заклинания в HUD, будет посылать эти заклинания. Интерфейс будет выглядеть как то так:



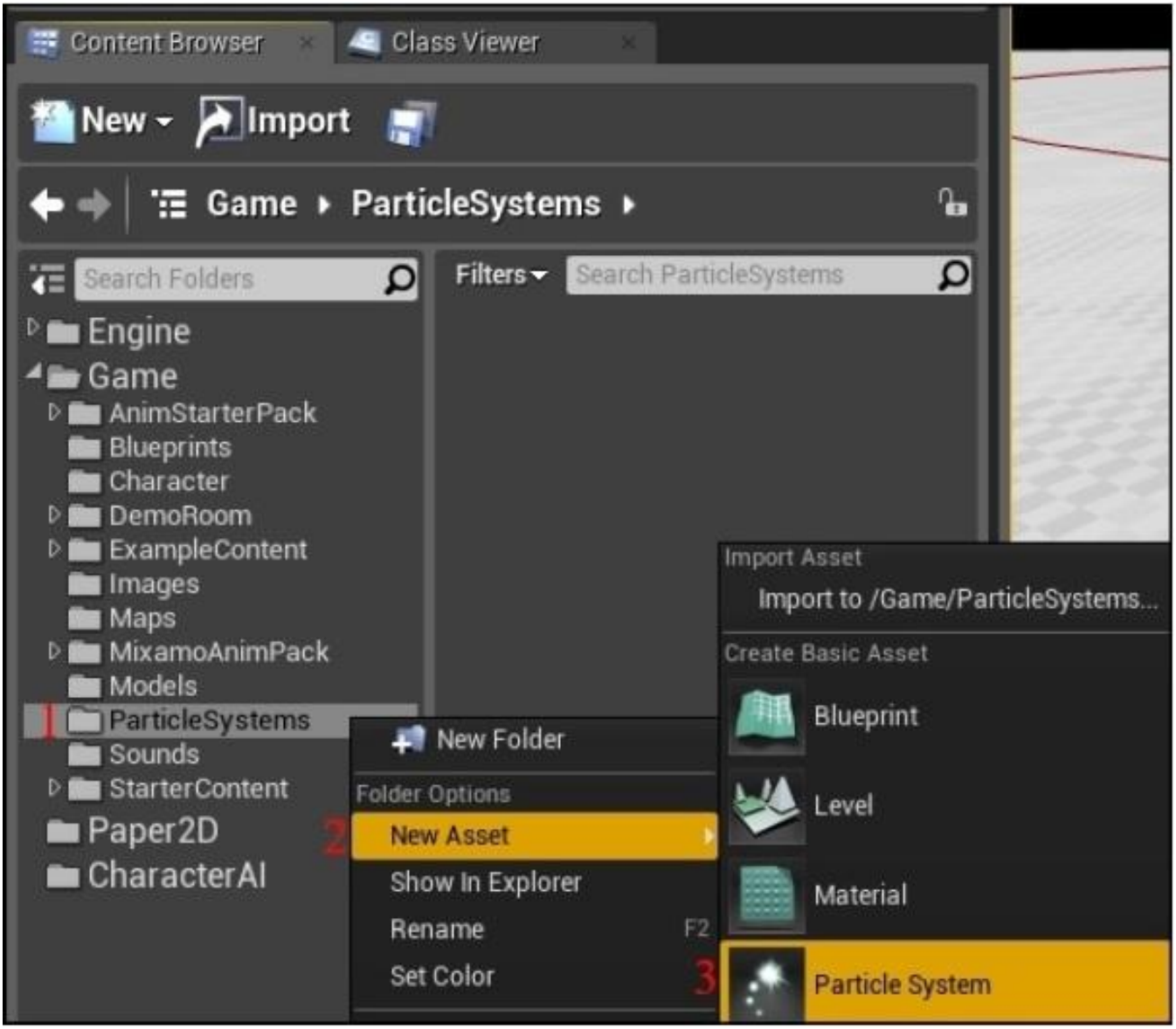
Предметы, которые подобрал игрок, а также различные заклинания

Мы начнём главу с описания того, как создать нашу собственную систему частиц. Затем мы пойдём дальше и заключим источник частиц в класс Spell (заклинание), и напишем функцию CastSpell() (наслать заклинание) для того, чтобы аватар мог насылать заклинания.

Система частиц

Сначала, нам нужно место, в которое мы поместим все броские эффекты. Щёлкните правой кнопкой в вашей вкладке **Content Browser**, по корневому

элементу **Game** и создайте новую папку, назвав её **ParticleSystems** (Системы Частиц). Щёлкните правой кнопкой по новой папке и выберите **New Asset** | **Particle System**, как показано на следующем скриншоте:

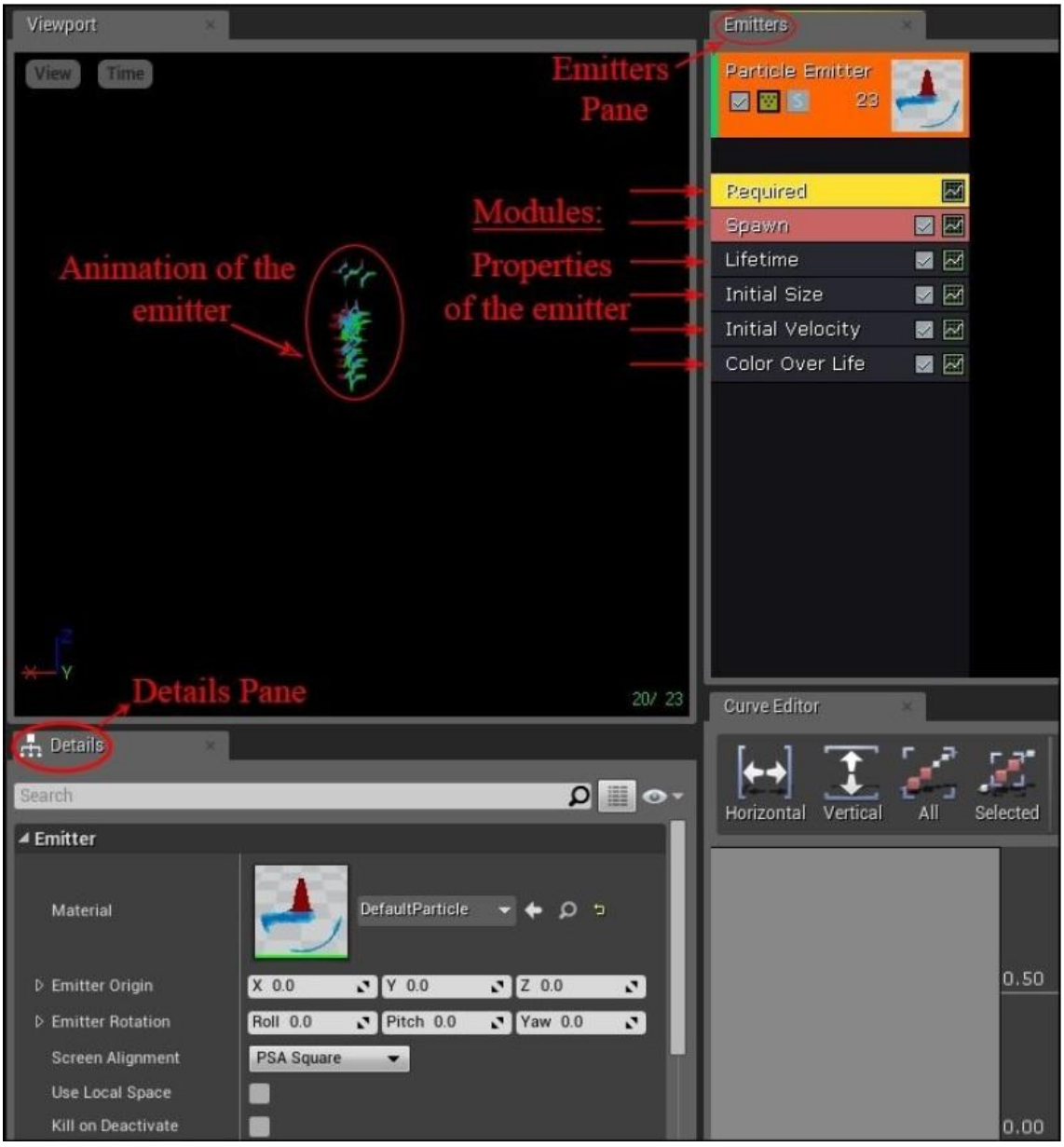


Совет

Посмотрите это руководство от Unreal Engine 4 по системам частиц, в целях информации того, как работают источники частиц в Unreal: https://www.youtube.com/watch?v=OXK2Xbd7D9w&index=1&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t.

Дважды щёлкните по значку NewParticleSystem, который появится, как показано на следующем скриншоте:

Вы окажитесь в Cascade, редакторе частиц. Описание среды показано на следующем скриншоте:

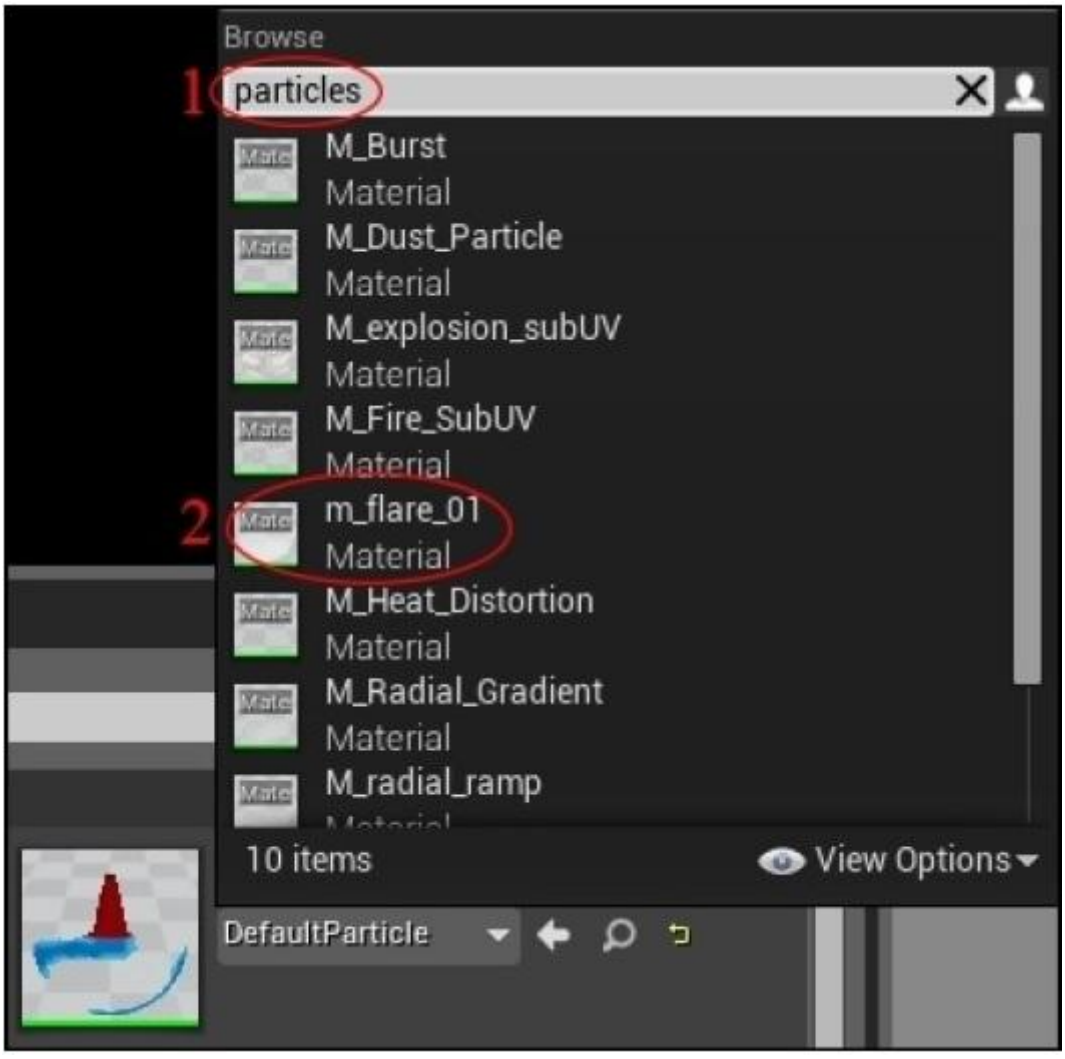


Здесь есть несколько разных панелей, каждая из которых показывает различную информацию:

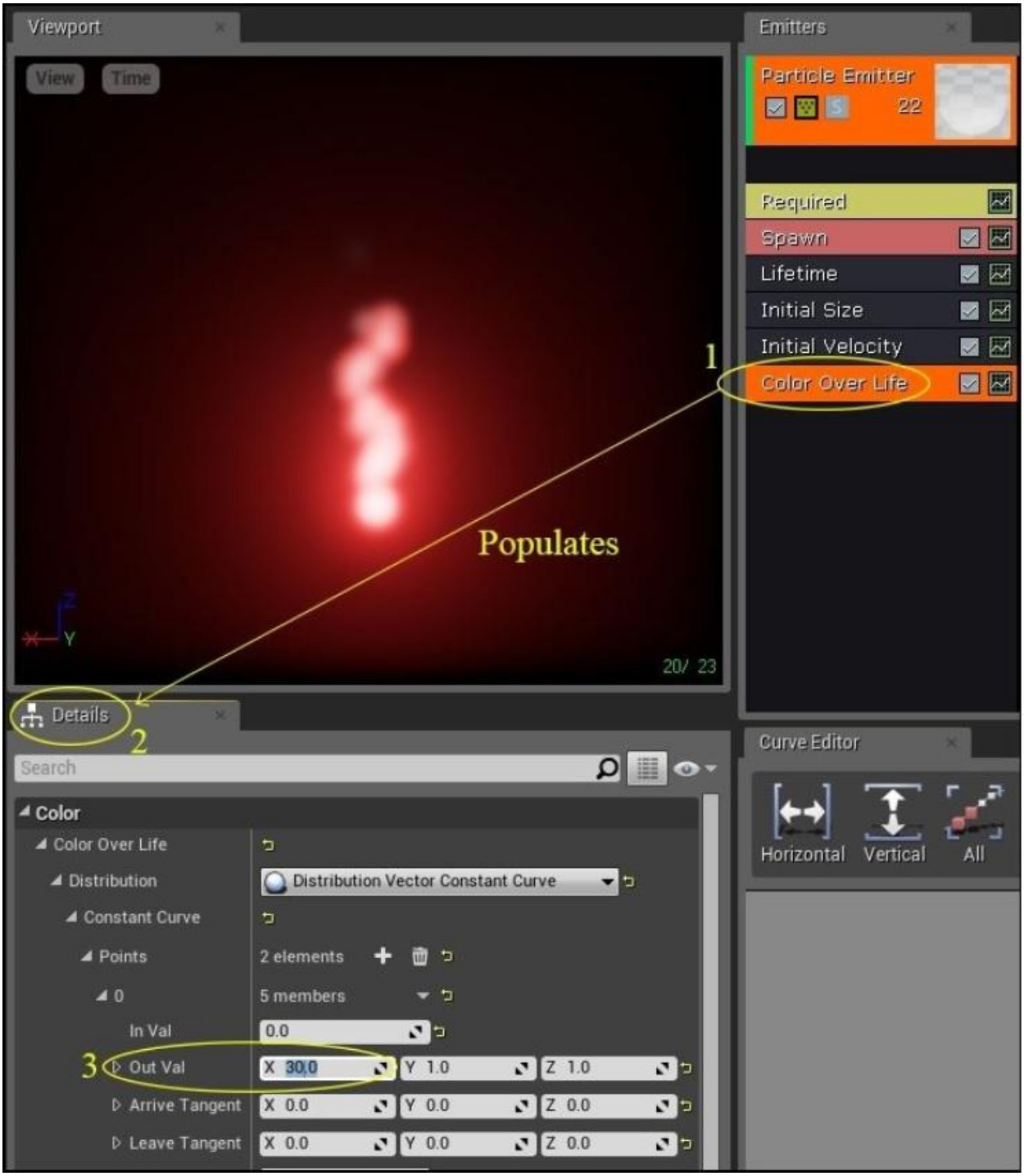
- Сверху слева находится панель **Viewport**. Она показывает анимацию текущего источника, как он сейчас работает.
- Справа находится панель **Emitters**. В ней вы можете видеть единственный объект, называемый **Particle Emitter** (вы можете иметь больше чем один источник в вашей системе частиц, но мы сейчас не будем этого делать). Список модулей **Particle Emitter** находится под ним. На этом скриншоте у нас есть модули: **Required**, **Spawn**, **Lifetime**, **Initial Size**, **Initial Velocity** и **Color Over Life**.

Изменение свойств частиц

Источник частиц по умолчанию испускает формы прицела. Мы хотим изменить это на что-то более интересное. Щёлкните по жёлтому полю **Required** под панелью **Emitters**, затем под **Material** в панели **Details** напишите particles. Появится список всех материалов частиц переменной. Выберите опцию **m_flare_01**, чтобы создать нашу первую систему частиц, как показано на следующем скриншоте:

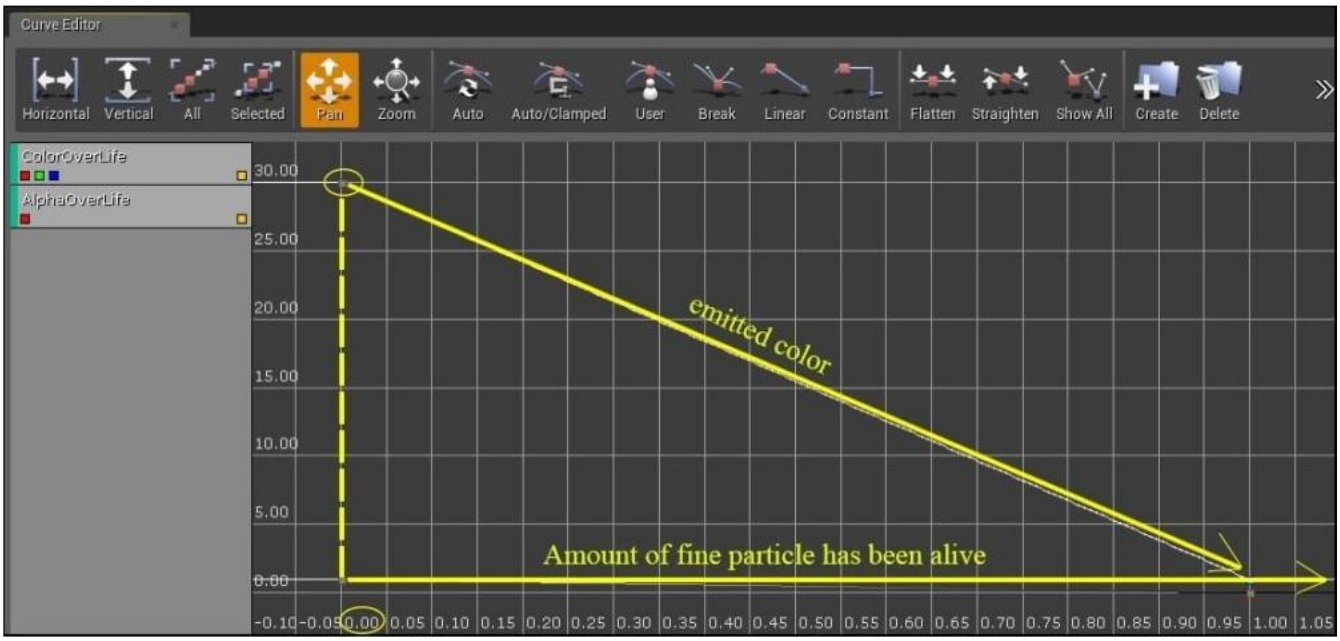


Теперь давайте изменим поведение системы частиц. Щёлкните по **Color Over Life** под панелью **Emitters**. Панель **Details** снизу, показывает информацию различных параметров:

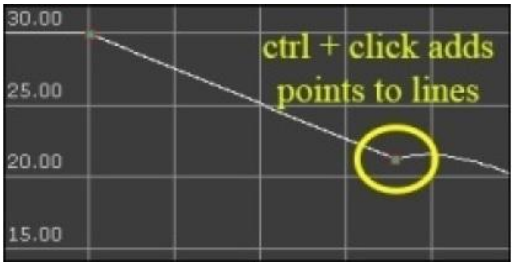


В панели **Details** пункта **Color Over Life**, я повысил **X**, но не **Y** и не **Z**. Это придаёт системе частиц, красное свечение. (**X** это красный, **Y** это зелёный, а **Z** это синий).

Вместо того, чтобы редактировать просто числа, вы можете изменять цвет частиц более наглядно. Если вы нажмёте зелёную кнопку с зигзагом в пункте **Color Over Life**, то вы увидите график, отображённый для **Color Over Life** во вкладке **Curve Editor** (редактор кривой), как показано на следующем скриншоте:



Мы можем сейчас изменять параметры **Color Over Life**. График во вкладке **Curve Editor** отображает излучаемый цвет против объёма времени, в течении которого живёт частица. Вы можете регулировать значения, перетаскивая точки. Нажав *Ctrl* + левую кнопку мыши, добавьте новую точку на линию:



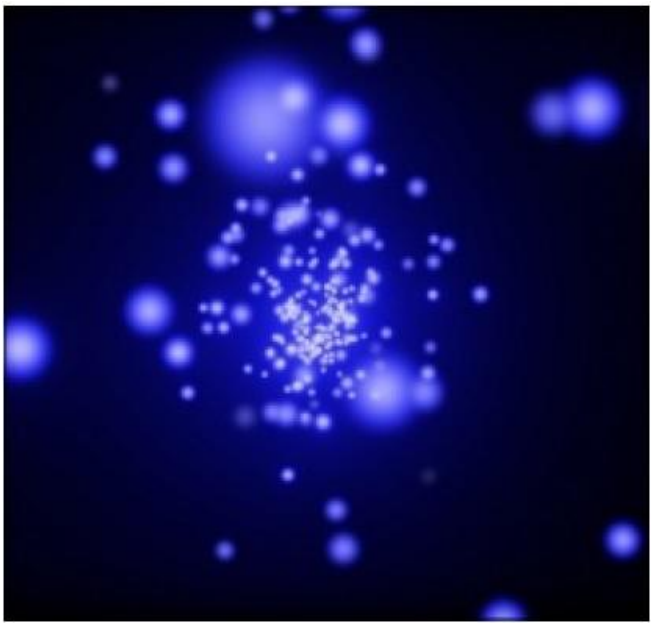
Ctrl + клик добавляет точку на линию

Вы можете поэкспериментировать с настройками источника частиц, чтобы создать вашу собственную визуализацию заклинания.

Настройки для заклинания метель

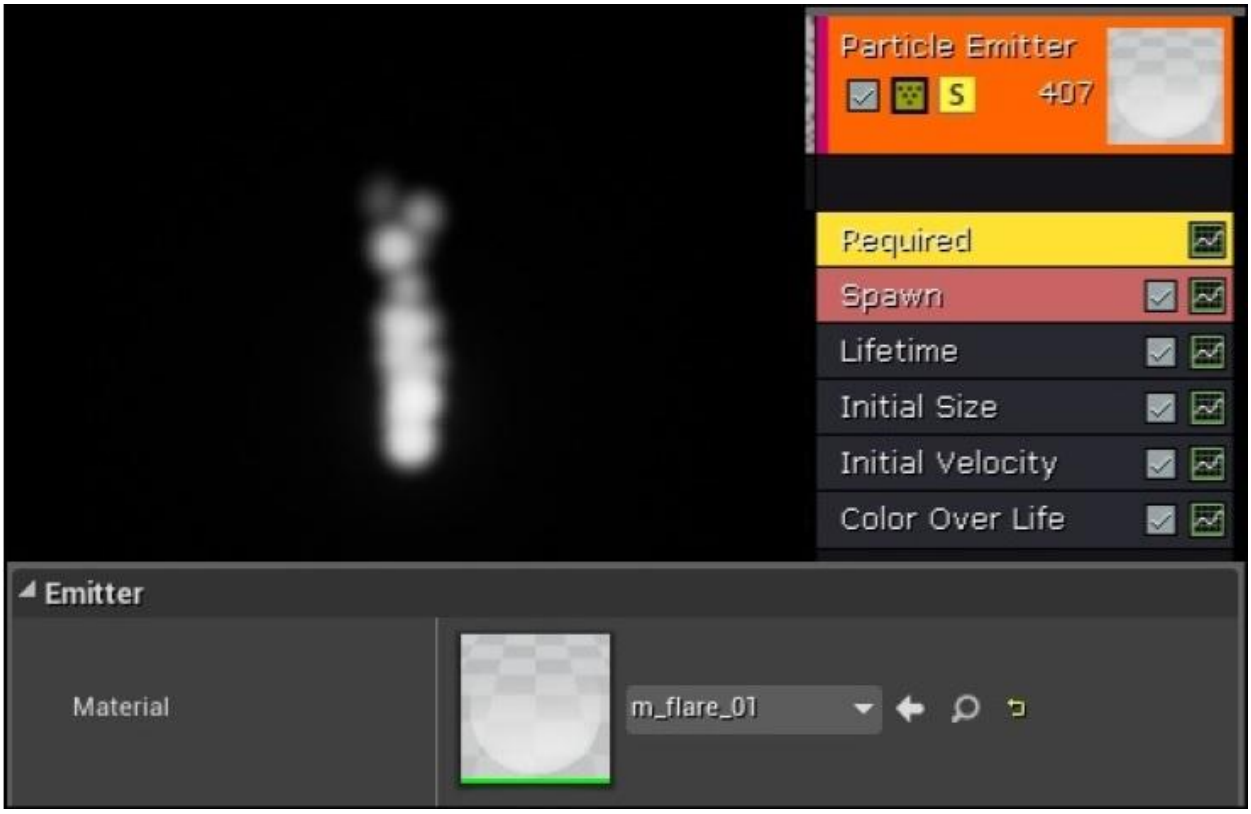
С этого момента, мы должны переименовать нашу систему частиц с NewParticle System, на что-то более подходящее. Давайте переименуем её на **P_Blizzard**. Вы можете переименовать вашу систему частиц, просто щёлкнув по ней и нажав *F2*.

Нажимаем F2 по объекту в Content Browser, чтобы переименовать его

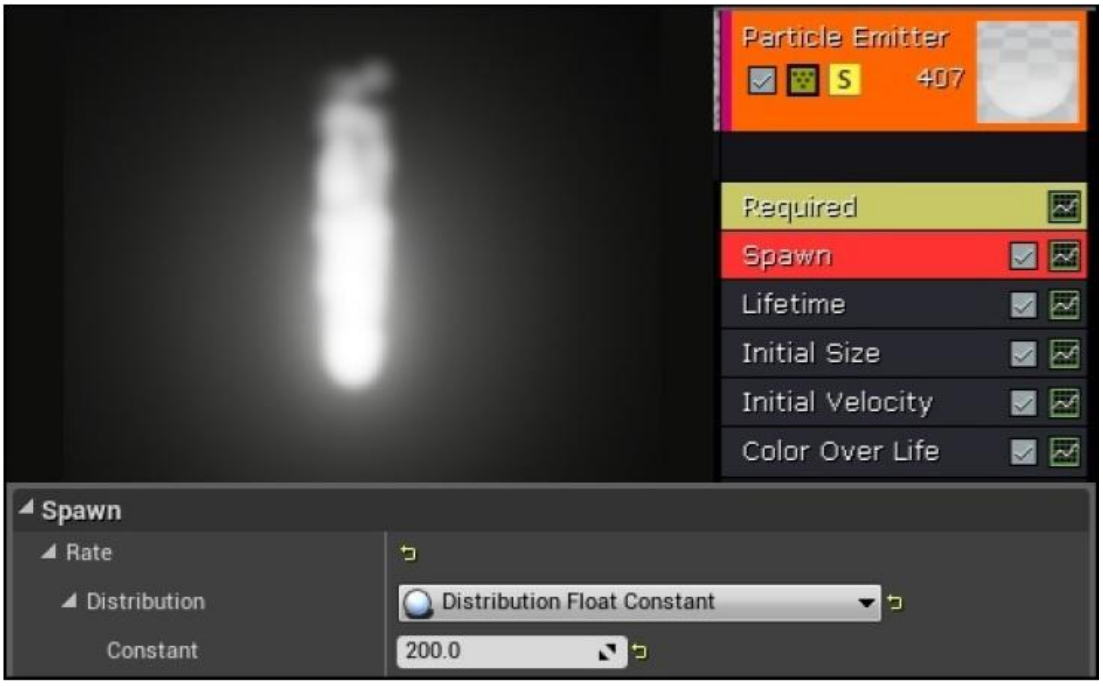


Мы отрегулируем некоторые настройки, чтобы получить заклинание с эффектом частиц метели. Выполните следующие шаги:

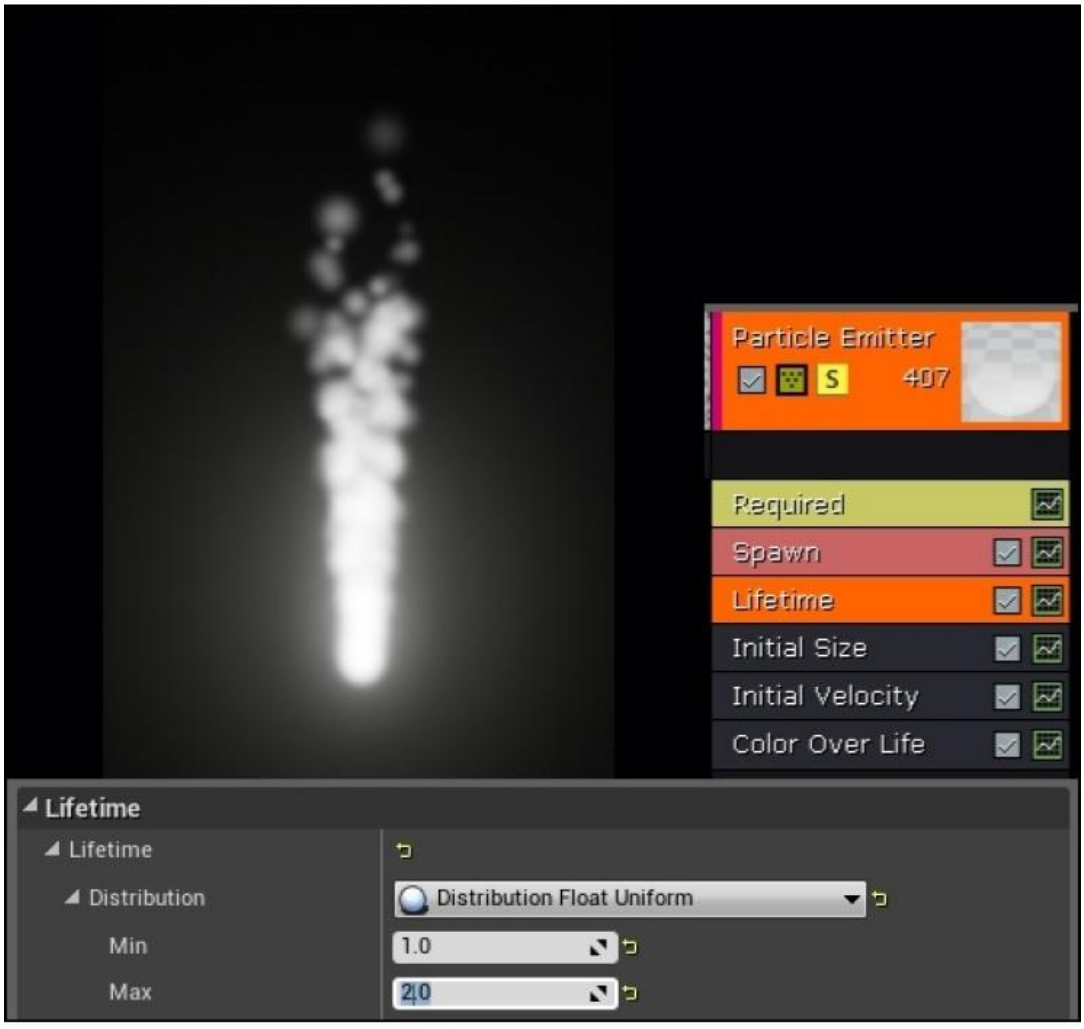
1. Под вкладкой **Emitters**, щёлкните по пункту **Required**. В панели **Details** измените материал **Emitter** на **m_flare_01**:



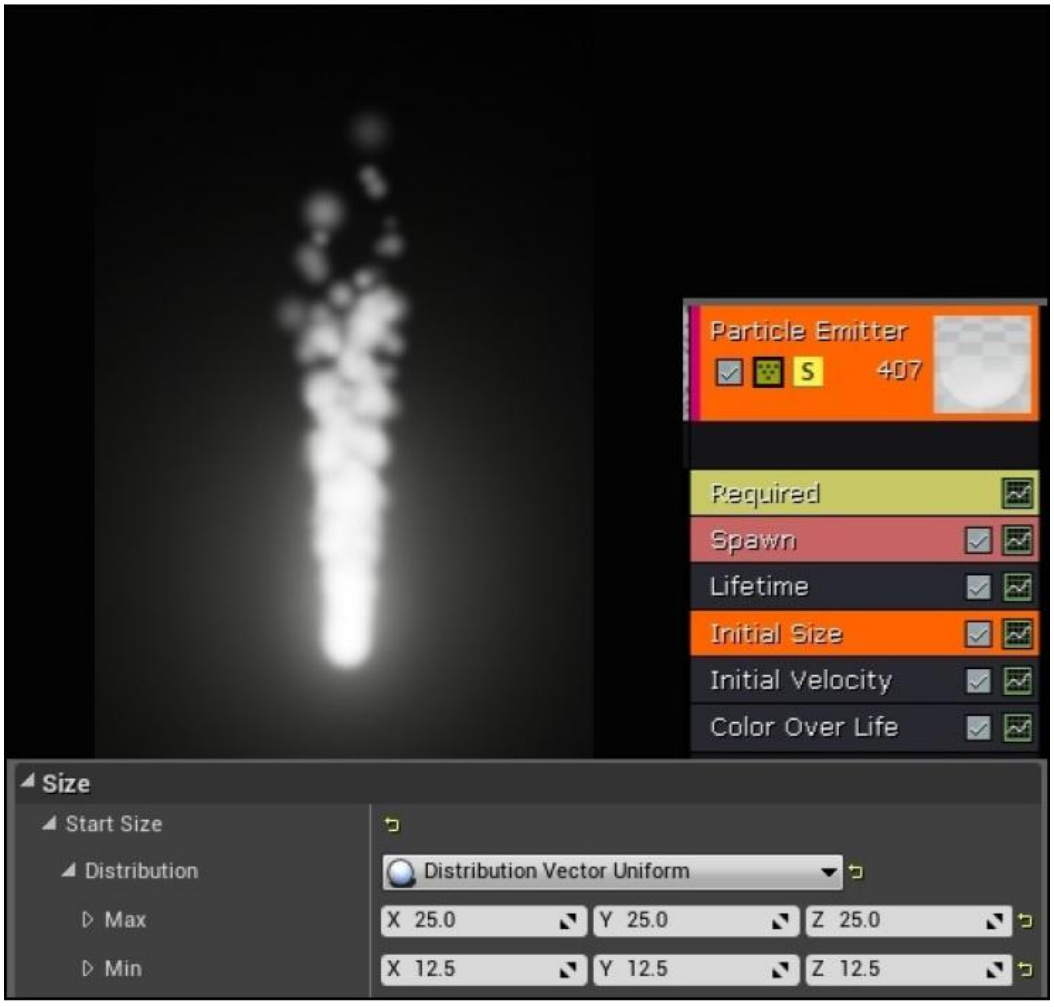
2. Под модулем Spawn (размножать), измените уровень размножения на 200. Это повышает плотность визуализации, как показано здесь:



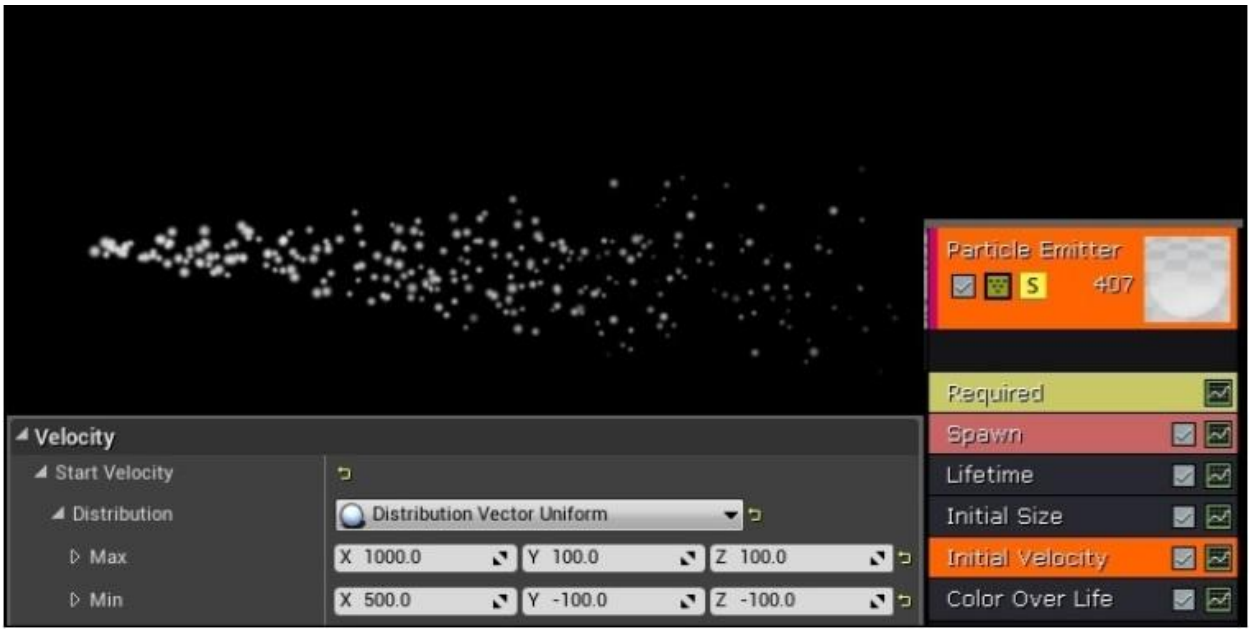
3. Под модулем **Lifetime**, повысьте свойство **Max** с 1.0 до 2.0. Это вводит некоторую вариацию во время жизни частиц. То есть некоторые частицы буду жить дольше, чем другие.



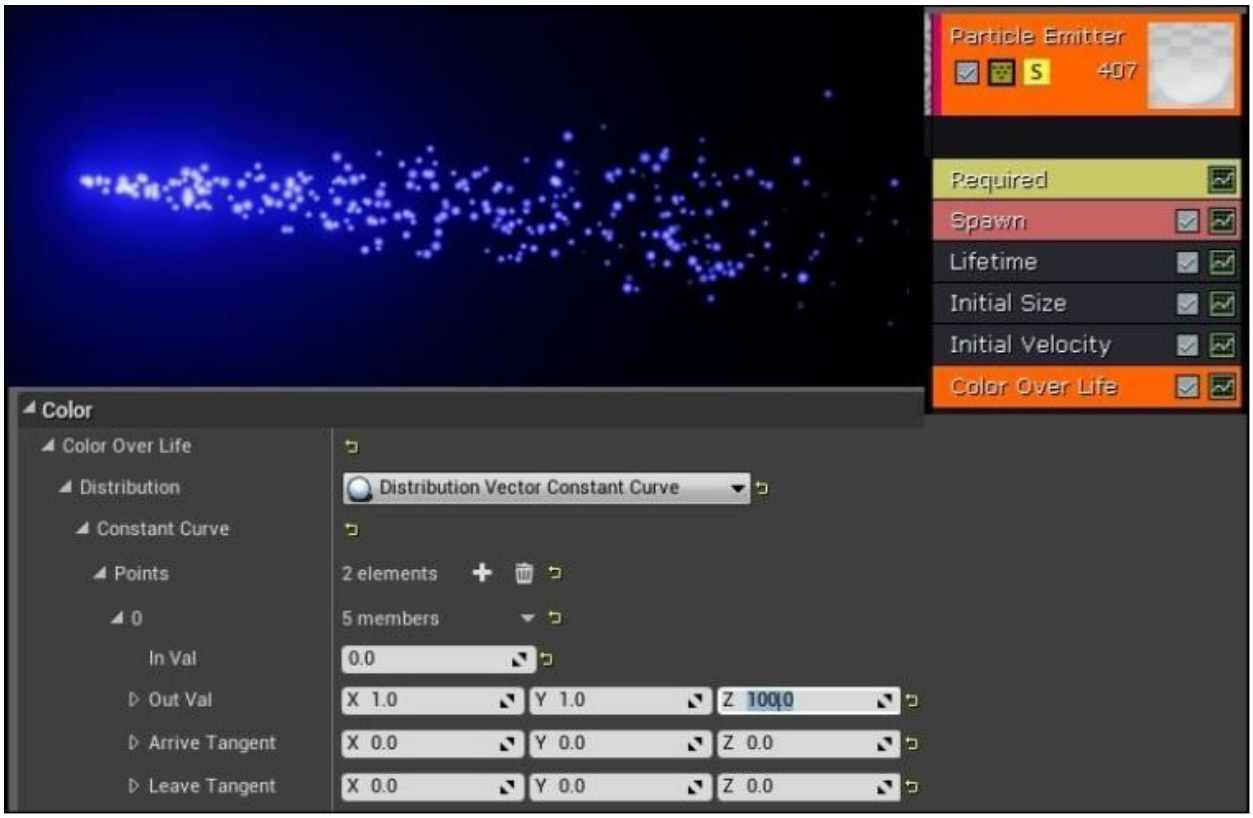
4. Под модулем **Initial Size** (начальный размер), измените свойство размера **Min** на 12.5 в **X**, **Y** и **Z**:



5. Под модулем **Initial Velocity** (изначальное ускорение), измените значения **Min**/**Max** на показанные здесь значения:

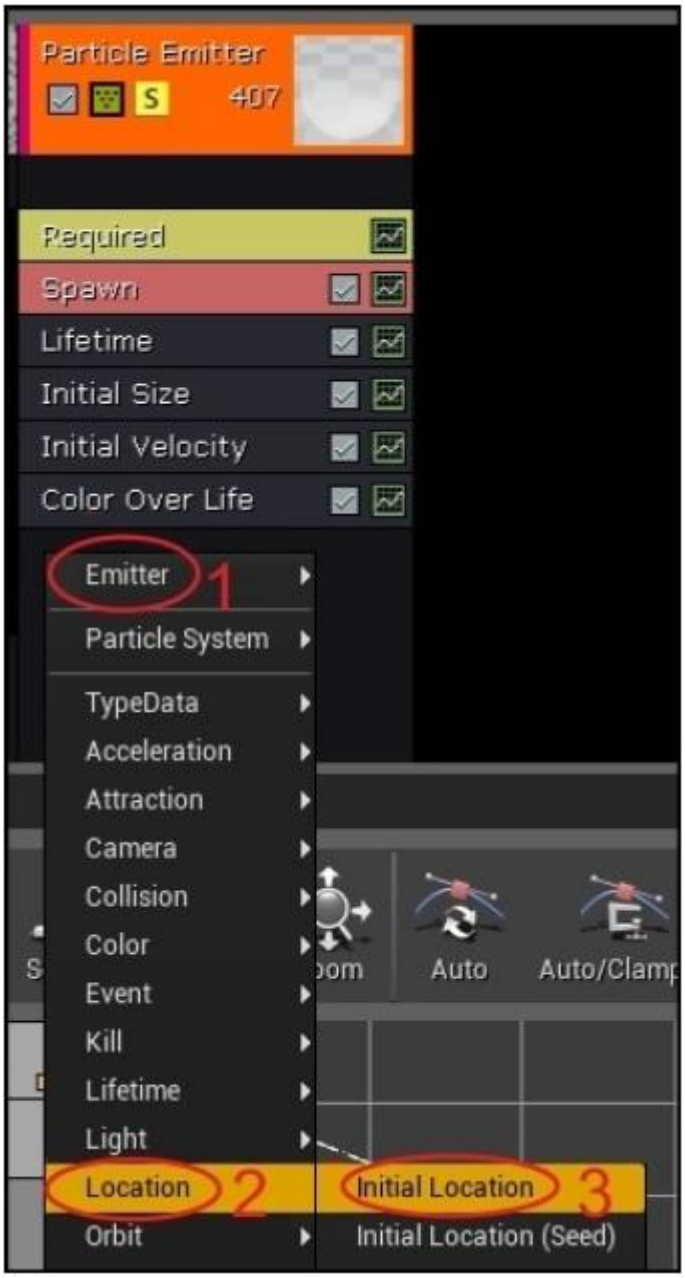


6. Причина, по которой у нас метель дует +X, это потому что направление игрока вперёд начинается в +X. И так как заклинание будет исходить от рук игрока, мы хотим, чтобы заклинание указывало в ту же сторону куда обращён игрок.
7. Под меню **Color Over Life** измените значение синего (**Z**) на 100.0. Вы увидите мгновенное изменение в синем свечении:

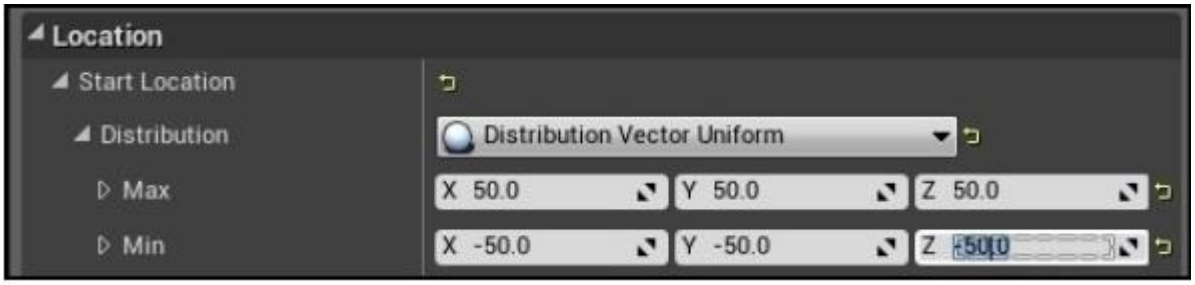


Теперь это выглядит волшебно!

8. Щёлкните правой кнопкой мыши по черноватой области под модулем **Color Over Life**. Выберите **Location** | **Initial Location**:

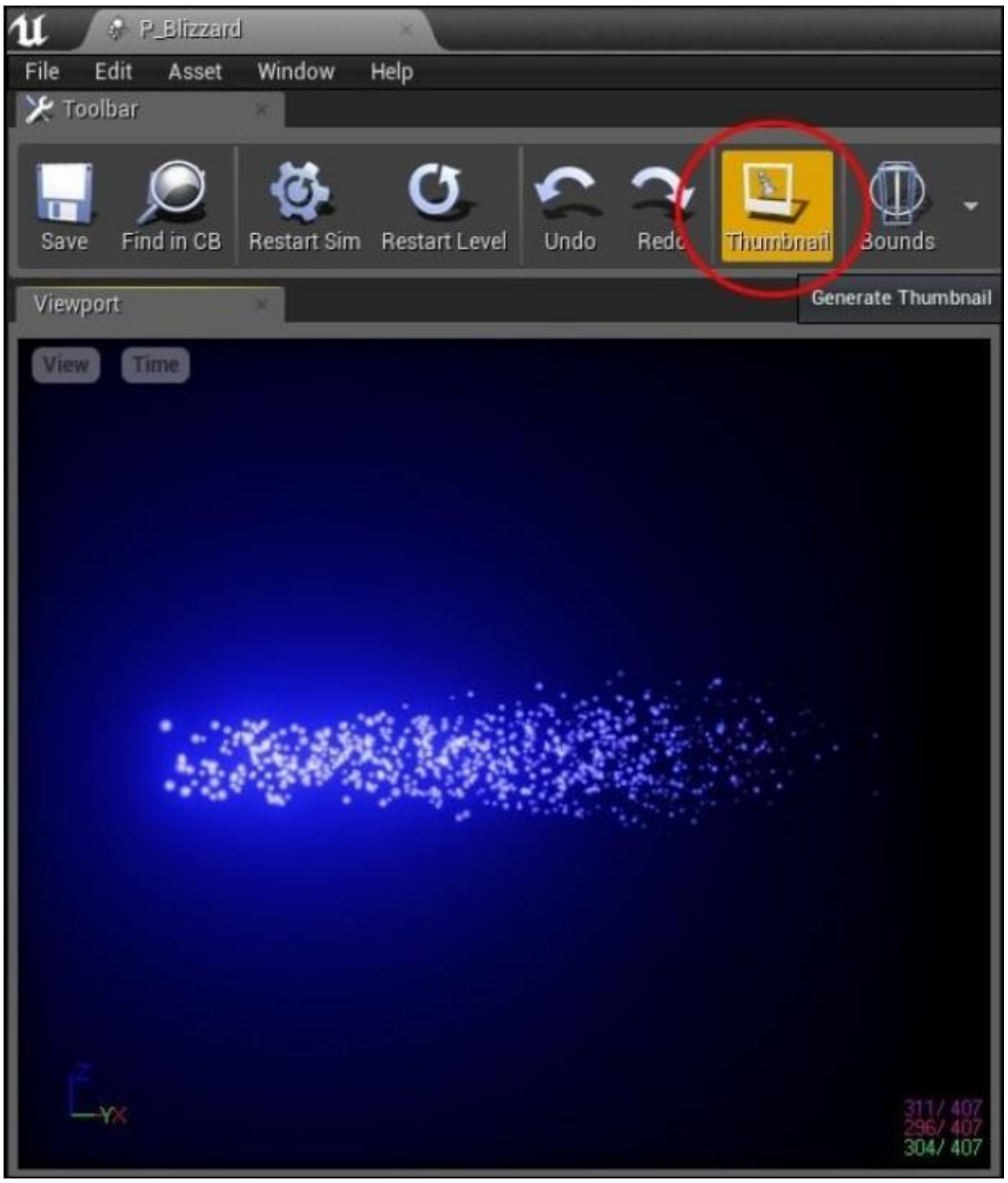


9. Введите значения под Start Location | Distribution как показано ниже:



10. У вас должна получиться вот такая метель:

11. Подвиньте камеру в положение, которое вам нравится, затем щёлкните по опции **Thumbnail** в панели меню сверху. Это сгенерирует значок/иконку для вашей системы частиц во вкладке **Content Browser**.



Щёлкая Thumbnail в панели меню сверху, генерирует мини иконку для вашей системы частиц

Актор класса заклинания

Класс Spell будет наносить крайний урон всем монстрам. Ближе к концу нам нужно содержать и систему частиц, и ограничивающий блок в классе актора Spell. Когда класс Spell приводится аватаром, экземпляр объекта Spell будет создан в уровне и начнёт функционировать Tick. При каждом Tick() объекта Spell на любого монстра содержащего внутри ограничивающий объём заклинания, будет производиться эффект этого заклинания – Spell.

Класс Spell должен выглядеть как в следующем коде:

```
UCLASS()
class GOLDENEGG_API ASpell : public AActor
{
   GENERATED_UCLASS_BODY()

   // параллелепипед (box) определяющий объём урон
   UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Spell)
   TSubobjectPtr<UBoxComponent> ProxBox;

   // визуализация частиц заклинания
   UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Spell)
   TSubobjectPtr<UParticleSystemComponent> Particles;

   // Сколько урона наносит заклинание за секунду
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
   float DamagePerSecond;

   // Как долго длится заклинания
   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Spell)
   float Duration;

   // Отрезок времени, в котором заклинание живёт в уровне
   float TimeAlive;

   // Настоящий насылатель заклинаний (чтобы игрок не попал
   // сам в себя)
   AActor* Caster;

   // Делаем родителем этого заклинания актор насылающий заклинание
   void SetCaster( AActor* caster );

   // Запускаем каждый кадр. Подменяем функцию Tick, чтобы работать с уроном
   // для чего угодно в ProxBox каждый кадр.
   virtual void Tick( float DeltaSeconds ) override;
};
```

Есть только три функции об осуществлении которых нам стоит беспокоиться. А именно: конструктор ASpell::ASpell(), функция ASpell::SetCaster() и функция ASpell::Tick().

Откройте файл Spell.cpp. Добавьте строку, чтобы включить файл Monster.h, так что мы сможем иметь доступ к определению объекта Monster в файле Spell.cpp, как показано в следующей строке кода:

```
#include "Monster.h"
```

Сначала конструктор, который устанавливает заклинание и инициализирует все компоненты показанные в следующем коде:

```
ASpell::ASpell(const class FPostConstructInitializeProperties& PCIP) :
Super(PCIP)
{
    ProxBox = PCIP.CreateDefaultSubobject<UBoxComponent>(this, TEXT("ProxBox"));
    Particles = PCIP.CreateDefaultSubobject<UParticleSystemComponent>(this,
    TEXT("ParticleSystem"));

    // Particles является корневым компонентом, а ProxBox
    // является дочерним объектом системы частиц - Particle.
    // Если бы это был другой способ, то масштабирование ProxBox
    // также масштабировало бы Particles, чего мы не хотим
    RootComponent = Particles;
    ProxBox->AttachTo( RootComponent );

    Duration = 3;                 // Продолжительность
    DamagePerSecond = 1;   // Урон за секунду
    TimeAlive = 0;                // Время жизни

    PrimaryActorTick.bCanEverTick = true; // требуется, чтобы заклинания
    // “тикали” (tick)!
}
```

Особенно важна здесь, последняя строка PrimaryActorTick.bCanEverTick = true. Если вы не установили это, то ваш объект Spell не вызовет Tick().

Далее у нас есть метод SetCaster(). Он вызывается так, что тот, кто посылает заклинание, известен объекту Spell. Мы можем гарантировать, что посылающий заклинания не сможет навредить сам себе, своими собственными заклинаниями, используя следующий код:

```
void ASpell::SetCaster( AActor *caster )
{
    Caster = caster;
    AttachRootComponentTo( caster->GetRootComponent() );
}
```

И наконец то, у нас есть метод ASpell::Tick(), который собственно наносит урон ко всем содержащимся акторам, как показано в следующем коде:

```
void ASpell::Tick( float DeltaSeconds )
{
    Super::Tick( DeltaSeconds );

    // ищем ProxBox для всех акторов в объёме.
```

```
    TArray<AActor*> actors;
    ProxBox->GetOverlappingActors( actors );

    // наносим урон каждому актору попадающему в область блока
    for( int c = 0; c < actors.Num(); c++ )
    {
        // чтобы не навредить тому, кто посылает заклинание
        if( actors[ c ] != Caster )
        {
            // Наносим урон только, если блок пересекается с
            // КОРНЕВЫМ компонентом актора.
            // Таким образом, урона не будет, если будет просто
            // пересечение с SightSphere монстра
            AMonster *monster = Cast<AMonster>( actors[c] );

            if( monster && ProxBox->IsOverlappingComponent( monster->CapsuleComponent ) )
            {
                monster->TakeDamage( DamagePerSecond*DeltaSeconds, FDamageEvent(),0, this );
            }
            // чтобы наносить урон другим типам класса, попробуйте проверенное насылание
            // здесь...
        }
    }

    TimeAlive += DeltaSeconds;
    if( TimeAlive > Duration )
    {
        Destroy();
    }
}
```

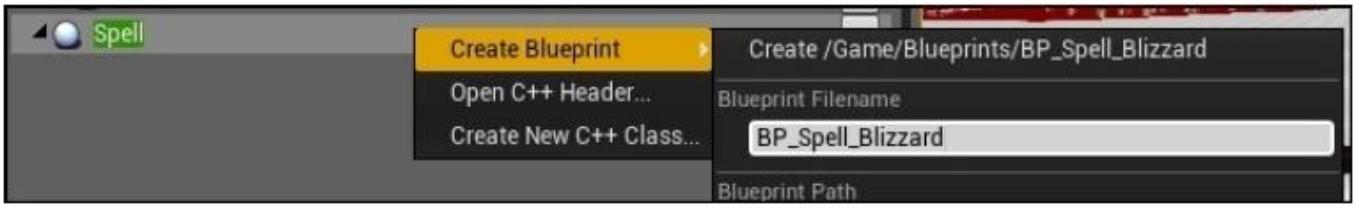
Функция ASpell::Tick() выполняет следующее:

- Получает всех акторов пересекающихся с ProxBox. Любой актор не являющийся посылателем заклинаний, получает урон, если пересекаемый компонент является корневым компонентом объекта. Причина, по которой нам нужно проверять на пересечение с корневым компонентом, заключается в том, что если мы этого не сделаем, то заклинание может пересечься с SightSphere монстра, что означает, что атака будет происходить с очень большого расстояния, чего мы не хотим.
- Заметьте, что если у нас есть другой класс, который должен получать урон, мы бы могли попробовать посылать заклинание на каждый тип объекта специально. Каждый тип класса может иметь различные типы объёма ограничения, которыми они должны сталкиваться. Другие типы могут даже не иметь CapsuleComponent (они могут иметь ProxBox или ProxSphere).
- Повышает объём времени жизни заклинания. Если продолжительность времени выделенного на заклинание заканчивается, то оно удаляется с уровня.

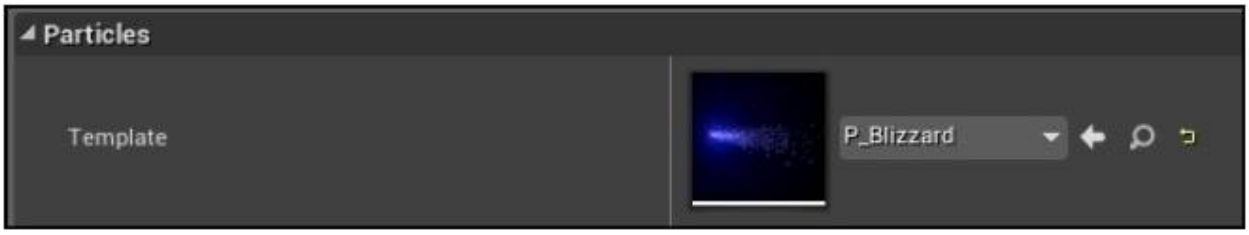
Теперь давайте сфокусируемся на том, как игрок может приобретать заклинания, создавая индивидуальный PickupItem для каждого объекта заклинания, который может брать игрок.

Блупринт для наших заклинаний

Компилируйте и запустите ваш C++ проект с классом Spell, который мы только что добавили. Нам нужно создать блупринты для каждого из заклинаний, которые мы хотим насылать. Во вкладке **Class Viewer**, начните печатать Spell и вы увидите как появится класс Spell. Щёлкните правой кнопкой по **Spell** и создайте блупринт, и назовите его **BP_Spell_Blizzard**, а затем дважды щёлкните по нему, чтобы открыть, как показано на следующем скриншоте:



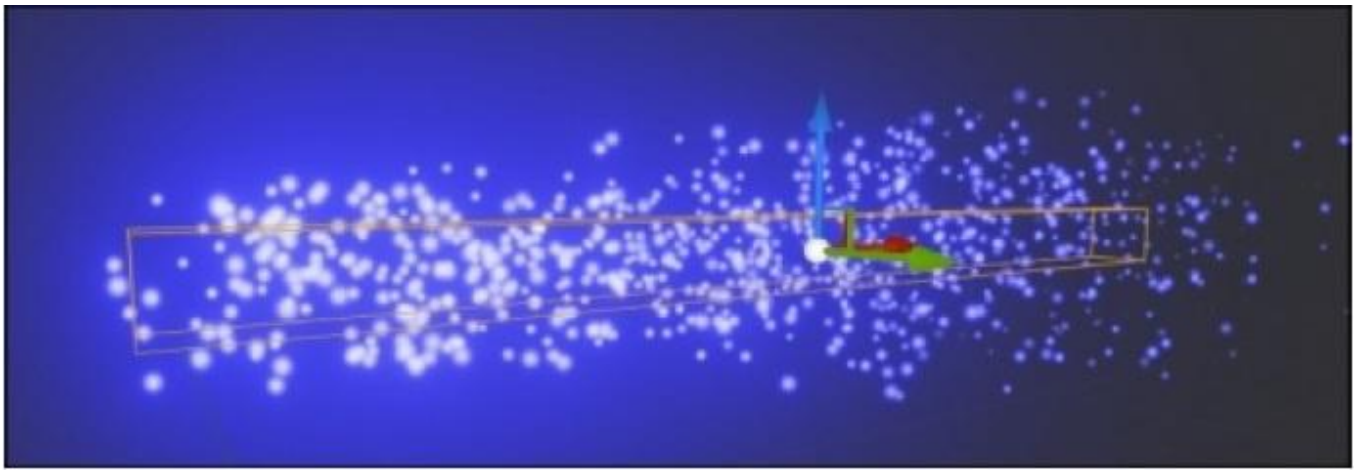
В свойствах заклинания выберите заклинание **P_Blizzard** для источника частиц, как показано на следующем скршоте:



Прокрутите вниз до категории **Spell** (заклинание) и измените параметры **Damage Per Second** (урон за секунду) и **Duration** (продолжительность) на значения по своему усмотрению. Здесь заклинание метель будет длиться 3 секунды и наносить урон 16.0 единиц за секунду. После трёх секунд, метель будет исчезать.

После того как вы конфигурировали свойства **Default** (по умолчанию), перейдите на вкладку **Components**, чтобы выполнить дальнейшие модификации. Щёлкните по ProBox и измените его форму, на более подходящую. Форма параллелепипеда

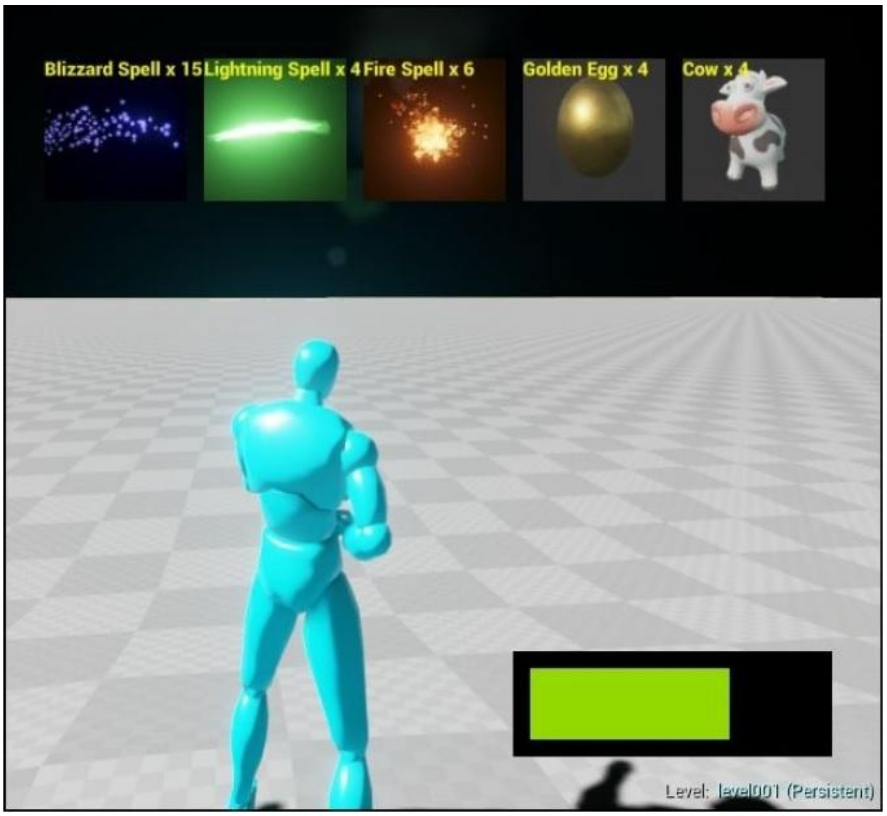
должна охватывать самую выраженную часть системы частиц и только не перебарщивайте. Объект ProxBox не должен быть слишком большим, потому что потом ваше заклинание метели будет действовать на то, что даже не задело. Как показано на следующем скриншоте, немного частиц за формой параллелепипеда ProxBox, это нормально.



Блупринт вашего заклинания метели готов к использованию игроком.

Выбор заклинаний

Вспомните, что ранее мы программировали наш инвентарь для отображения предметов имеющихся у игрока, когда игрок нажимает клавишу *I*. И теперь мы сделаем ещё больше.



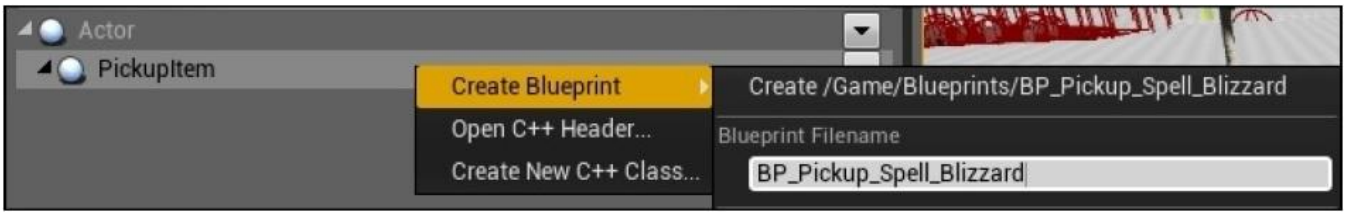
Чтобы позволить игроку выбирать заклинания, мы модифицируем класс PickupItem, внести слот для блупринта заклинания, которое посылает игрок, используя следующий код:

```
// в классе APickupItem:
// Если этот предмет посылает заклинание, установить его здесь
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Item)
UClass* Spell;
```

Как только вы добавите свойство UClass* Spell в класс APickupItem, снова компилируйте и запустите ваш C++ проект. Теперь вы можете перейти к созданию блупринта экземпляра PickupItem для вашего объекта Spell.

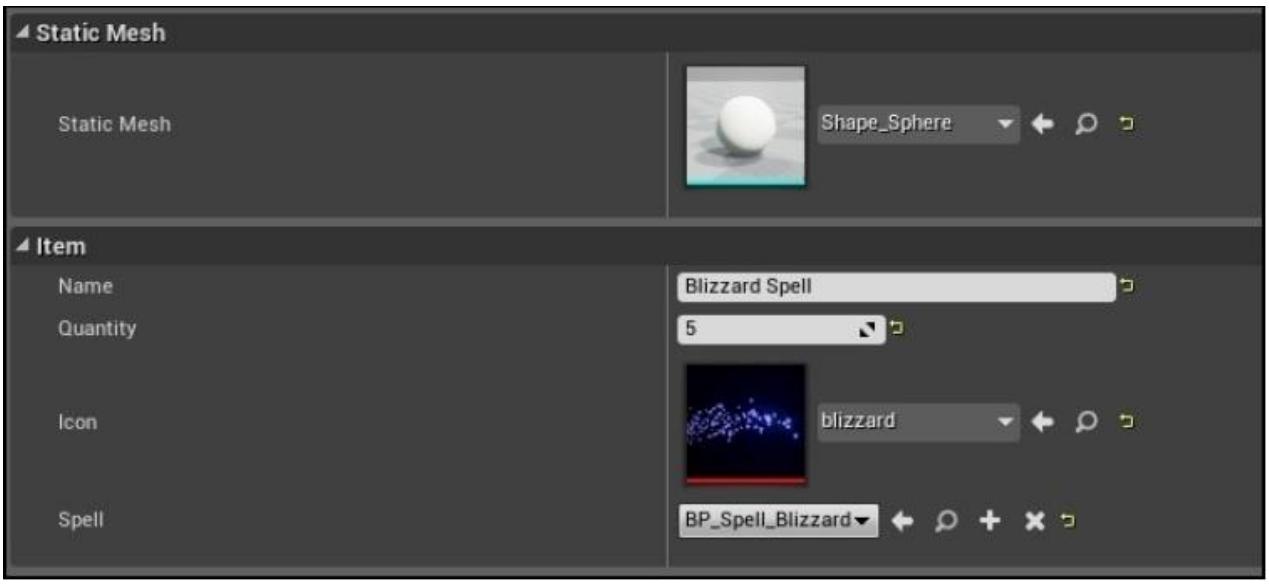
Создание блупринта для PickupItem, которые посылают заклинания

Создайте блупринт для PickupItem и назовите его BP_Pickup_Spell_Blizzard. Дважды щёлкните по нему, чтобы редактировать его свойства, как показано на следующем скриншоте:



Я установил свойства метели, следующим образом:

Имя предмета **Blizzard Spell** (заклинание метель) и пять в каждой группе. Я взял скриншот системы частиц и импортировал его в проект, так что в **Icon** я выбрал это изображение. Для **Spell** я выбрал **BP_Spell_Blizzard** в качестве имени посылаемого заклинания (не **BP_PickupItem_Spell_Blizzard**), как показано на следующем скриншоте:



Я выбрал синюю сферу для класса Mesh от класса PickupItem. Для **Icon** я взял скриншот заклинания метели в предпросмотровом окне частиц, сохранил его на диск и импортировал это изображение в проект (посмотрите папку с изображениями во вкладке **Content Browser** в примере проекта).

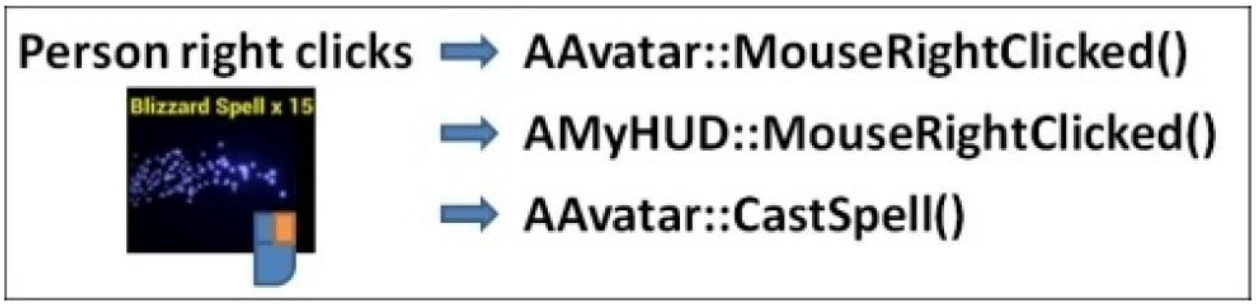
Поместите несколько этих объектов PickupItem в ваш уровень. Если мы их подбираем, то у нас есть заклинания метели в нашем инвентаре.

Слева: предметы заклинания Blizzard – метель, нашем игровом мире. Справа: предмет заклинания Blizzard в инвентаре.

Теперь нам нужно активировать метель. Так как в Главе 10. *Система инвентаризации и подбор предметов,* мы уже прикрепили левый клик мыши для перетаскивания значков, давайте прикрепим правый клик для того, чтобы слать заклинания.

Прикрепляем правый клик к посыланию заклинания

Правый клик мыши должен будет пройти через пару вызов функции перед вызовом метода CastSpell аватара. График вызова будет выглядеть подобным образом:



Пару вещей происходят между правым кликом и посыланием заклинания:

- Как мы видели до этого, все взаимодействия мыши пользователя и клавиатуры, проходят через объект Avatar. Когда объект Avatar определяет правый клик, он передаёт событие клика в HUD через AAvatar::MouseRightClicked().
- Вспомните в Главе 10. *Инвентарь и подбор предметов*, мы использовали класс struct Widget, чтобы отслеживать предметы которые взял игрок. Struct Widget имеет только три элемента:

  ```
  struct Widget
  {
     Icon icon;
     FVector2D pos, size;
     ///.. и некоторые функции-члены
  };
  ```

  Нам нужно будет добавить дополнительные свойства для класса struct Widget, чтобы запомнить заклинание которое он посылает.

  HUD определяет если событие клика было внутри Widget в AMyHUD::MouseRightClicked().

- Если клик был по графическому элементу (Widget), который посылает заклинание, затем HUD вызывает аватар обратно с запросом послать заклинание, вызывая AAvatar::CastSpell().

Написание функции CastSpell аватара

Мы применим предыдущий график вызова в обратном порядке. Мы начнём с написания функции, которая собственно посылает заклинание в игре, AAvatar::CastSpell(), как показано в следующем коде:

```
void AAvatar::CastSpell( UClass* bpSpell )
{
   // создаём экземпляр заклинания и прикрепляем его к персонажу
   ASpell *spell = GetWorld()->SpawnActor<ASpell>(bpSpell, FVector(0),
   FRotator(0) );

   if( spell )
   {
      spell->SetCaster( this );
   }
   else
   {
      GEngine->AddOnScreenDebugMessage( 1, 5.f, FColor::Yellow, FString("заклинание не
      послать ") + bpSpell->GetName() );
   }
}
```

Вы можете обнаружить, что сам вызов заклинания примечательно прост. Тут два базовых шага, чтобы посылать заклинание:

- Создайте экземпляр объекта заклинания, используя функцию мирового объекта SpawnActor (породить актора)
- Прикрепите его к аватару

Как только экземпляр объекта заклинания создан, его функция Tick() будет запускаться каждый кадр, когда это заклинание в уровне. Каждый раз при Tick(), объект Spell будет автоматически находить монстров в пределах уровня, и наносить им урон. С каждой строкой кода упомянутой ранее происходит многое, так что давайте обсудим каждую строку в отдельности.

Создание экземпляра заклинания – GetWorld()->SpawnActor()

Чтобы создать объект Spell из блупринта, нам нужно вызвать функцию SpawnActor() из объекта World. Функция SpawnActor() может принимать любой блуринт и создавать его экземпляр в уровне. И объект Avatar (и вообще-то любой объект Actor) может справиться с объектом World в любое время, просто вызывая функцию-член GetWorld().

Строка кода, которая приносит объект Spell в уровень:

```
ASpell *spell = GetWorld()->SpawnActor<ASpell>( bpSpell, FVector(0), FRotator(0) );
```

В этой строке есть пара вещей, на которые стоит обратить внимание:

- bpSpell должен быть созданным блупринтом объекта Spell. Объект <ASpell> в угловых скобках указывает это ожидание.
- Новый объект Spell начинает с начала отсчёта координат (0, 0, 0) и применен к нему без дополнительного вращения. Это потому что мы прикрепим объект Spell к объекту Avatar, который будет предоставлять передачу и направление компонентов для объекта Spell.

If(spell)

Мы всегда тестируем, если вызов SpawnActor<ASpell>() проходит успешно, проверяя if(spell). Если блупринт переданный объекту CastSpell, не является блупринтом основанным на классе ASpell, то функция SpawnActor() возвращает указатель NULL вместо объекта Spell. Если это происходит, мы выводим на экран сообщение об ошибке, указывающее, что что-то пошлее неправильно во время посылания заклинания.

spell->SetCaster(this)

При начальной установке значений, если заклинание проходит успешно, то мы прикрепляем его к объекту Avatar, вызывая spell->SetCaster(this). Помните, что в контексте программирования в пределах класса Avatar, метод this является ссылкой на объект Avatar.

Теперь, как мы собственно соединим посылание заклинания от вводных данных UI, к вызову функции AAvatar::CastSpell() на первом месте? Нам нужно снова выполнить немного программирования HUD.

Написание AMyHUD::MouseRightClicked()

Команда посылания заклинания, в конечном счете, будет приходить из HUD. Нам нужно написать C++ функцию, которая будет проходить через все графические элементы HUD и выявлять клик по какому-либо из них. Если клик на объекте widget, то этот объект widget должен отвечать заклинанием, если оно ему назначено. Нам нужно расширить наш объект Widget, чтобы иметь переменную, которая будет содержать блупринт посылания заклинания. Добавьте элемент в ваш объект struct Widget, используя следующий код:

```
struct Widget
{
    Icon icon;
    // bpSpell является блупринтом заклинания, которое посылает этот графический элемент
    UClass *bpSpell;
    FVector2D pos, size;
    Widget(Icon iicon, UClass *iClassName)
}
```

Теперь вспомните, что наш PickupItem имел прикреплённый к нему ранее блупринт заклинания, которое он посылает. Однако когда класс PickupItem (подбираемый предмет) подбирался в уровне игроком, то класс PickupItem ликвидировался.

```
// Из APickupItem::Prox_Implementation():
avatar->Pickup( this ); // give this item to the avatar
// удаляем подбираемый предмет, как только он был подобран
Destroy();
```

Так что нам нужно удерживать информацию о том, какое заклинание посылает каждый PickupItem. Мы можем сделать это, кода этот PickupItem подбирается первый раз.

Внутри класса AAvatar добавьте дополнительную карту, чтобы запоминать блупринт заклинания, которое посылает предмет, по имени предмета:

```
// Поместите это в Avatar.h
TMap<FString, UClass*> Spells;
```

Теперь в AAvatar::Pickup(), запомните класс заклинания, экземпляр которого создаёт класс PickupItem, с помощью следующей строки кода:

```
// заклинание связанное с предметом
Spells.Add(item->Name, item->Spell);
```

Теперь в AAvatar::ToggleInventory() мы может иметь объект Widget, который отображается на экране. Вспомните, какое заклинание он должен посылать, посмотрев на карту Spells.

Найдите строку, на которой мы создали графический элемент (widget), и сразу под ней добавьте назначение объектов bpSpell, которые посылает Widget:

```
// В AAvatar::ToggleInventory()
Widget w( Icon( fs, tex ) );
w.bpSpell = Spells[it->Key];
```

Добавьте следующую функцию к AMyHUD, которую установим для запуска, когда правая кнопка мыши щёлкает по значку:

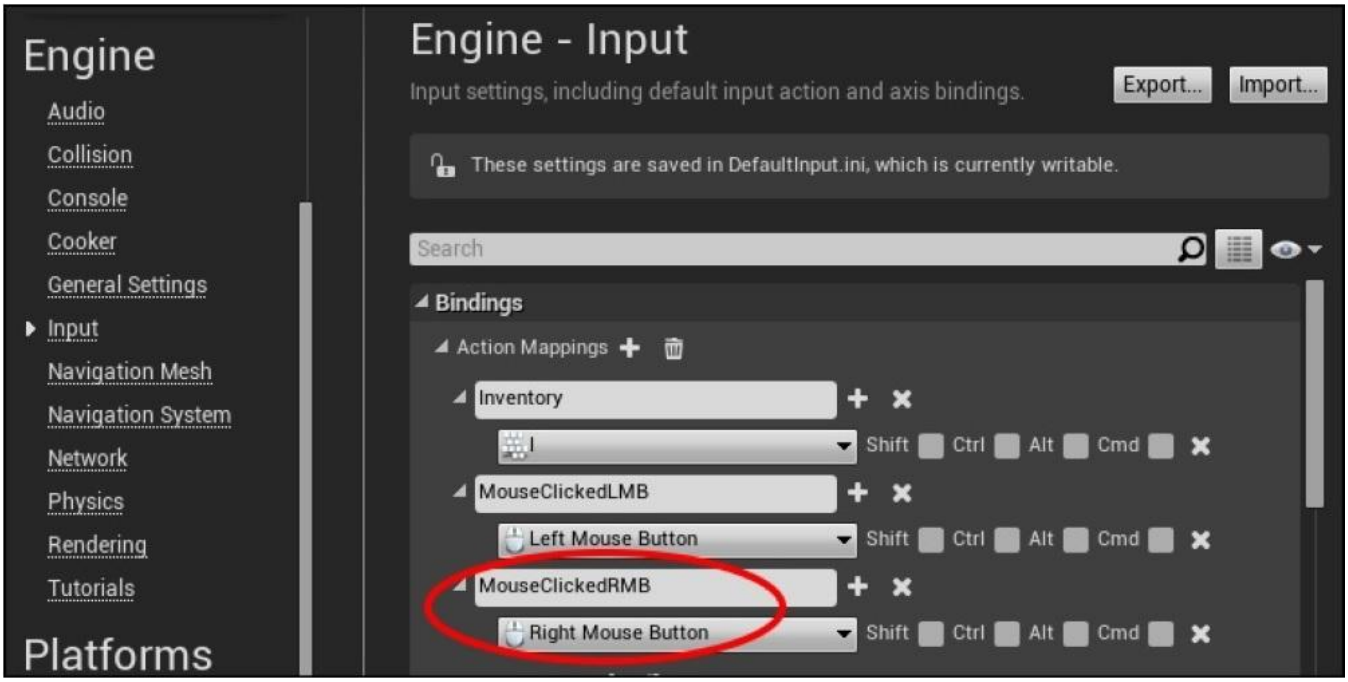
```
void AMyHUD::MouseRightClicked()
{
   FVector2D mouse;
   APlayerController *PController = GetWorld()- >GetFirstPlayerController();
   PController->GetMousePosition( mouse.X, mouse.Y );
   for( int c = 0; c < widgets.Num(); c++ )
   {
      if( widgets[c].hit( mouse ) )
      {
         AAvatar *avatar = Cast<AAvatar>( UGameplayStatics::GetPlayerPawn(GetWorld(), 0) );
         if( widgets[c].spellName )
         avatar->CastSpell( widgets[c].spellName );
      }
```

```
    }
}
```

Это очень похоже на нашу функцию левого клика мыши. Мы просто проверяем положение клика в отношении графического элемента. Если какой-либо Widget был нажат правым кликом и этот Widget имеет связанный с ним объект Spell, то заклинание будет послано вызовом метода аватара CastSpell().

Активация клика правой кнопки мыши

Чтобы соединить эту HUD функцию с запуском, нам нужно добавить обработчик событий к правому клику мыши. Мы можем сделать это перейдя в **Settings** | **Project Settings**, и в появившемся диалоговом окне добавить опцию **Input** для **Right Mouse Button**, как показано на следующем скриншоте:



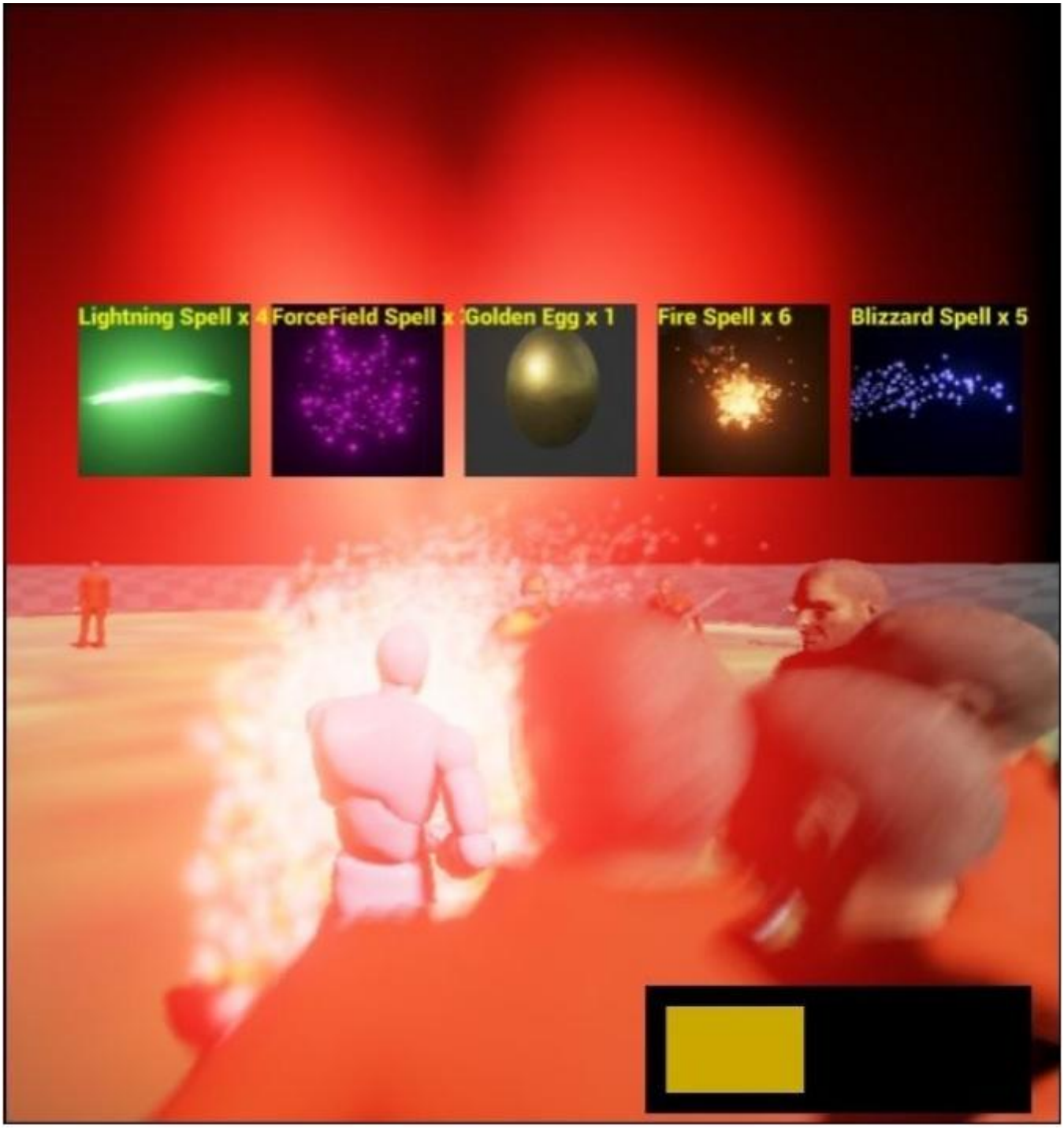
Объявите функцию под названием MouseRightClick() в Avatar.h/Avatar.cpp, используя следующий код:

```
void AAvatar::MouseRightClicked()
{
    if( inventoryShowing )
    {
        APlayerController* PController = GetWorld()->GetFirstPlayerController();
        AMyHUD* hud = Cast<AMyHUD>( PController->GetHUD() );
        hud->MouseRightClicked();
    }
}
```

Затем в AAvatar::SetupPlayerInputComponent(), нам нужно прикрепить событие MouseClickedRMB к функции MouseRightClicked():

```
// В AAvatar::SetupPlayerInputComponent():
InputComponent->BindAction( "MouseClickedRMB", IE_Pressed, this,
&AAvatar::MouseRightClicked );
```

Мы наконец то подключили посылание заклинаний. Испытайте их, функционал игры теперь довольно классный:

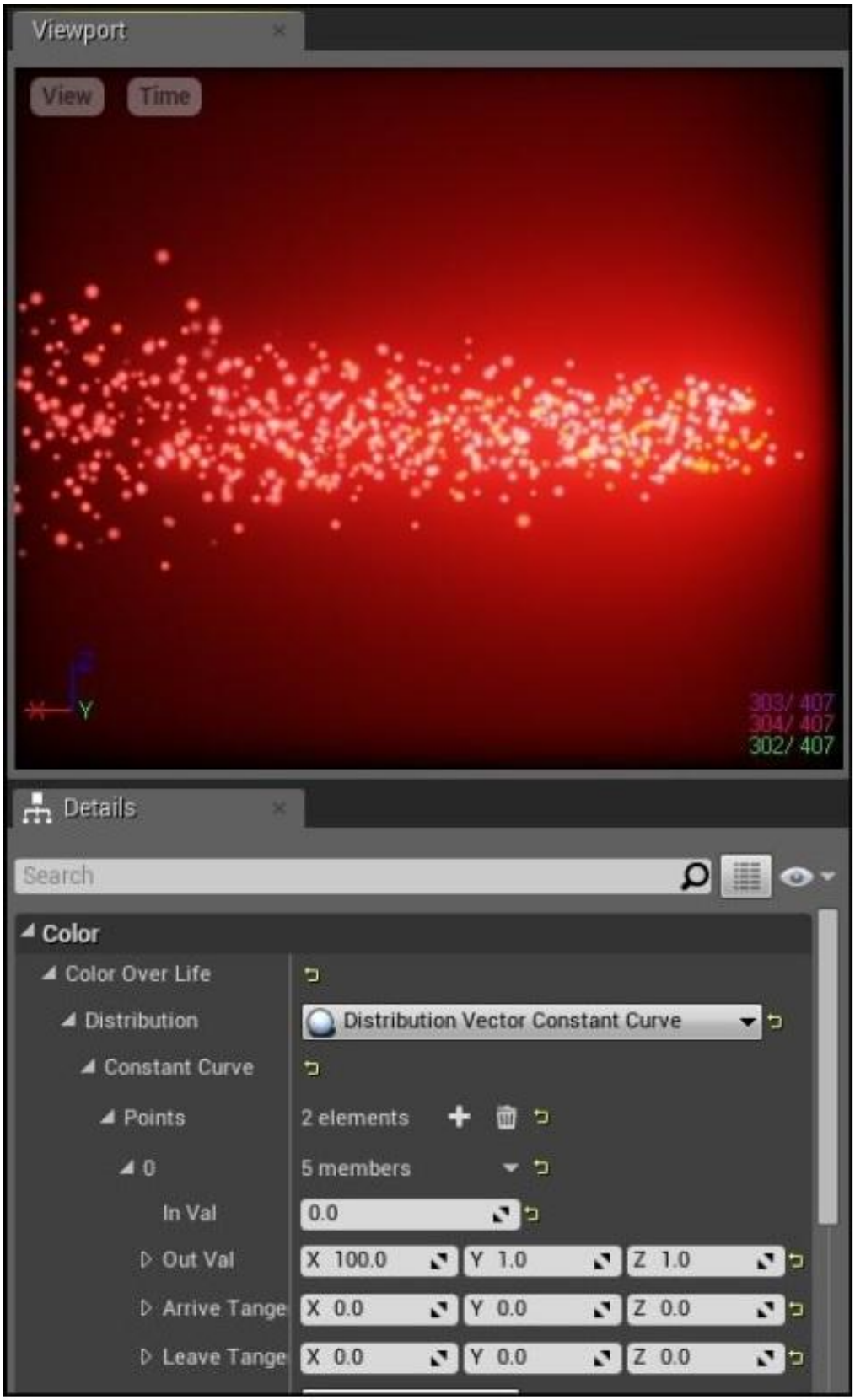


Создание других заклинаний

Экспериментируя с системой частиц, вы можете создать разнообразие различных заклинаний, которые производят разные эффекты.

Огненное заклинание

Вы можете легко создать огненный вариант нашего заклинания метели, изменив цвет системы частиц на красный:



Параметр цвета Out Val изменён на красный

Упражнения

Попробуйте следующие упражнения:

1. **Заклинание молнии**: Создайте заклинание молнии, используя луч частиц. Посмотрите пособие Зака в качестве примера, как создаются и выстреливаются в каком-либо направлении лучи:

   https://www.youtube.com/watch?v=ywd3lFOuMV8&list=PLZlv_N0_O1gYDLyB3LVfjYIcbBe8NqR8t&index=7.

2. **Заклинание силового поля**: Силовое поле будет отклонять атаки. Это необходимо любому игроку. Предложенное осуществление: Выполните наследование подкласса от ASpell под названием ASpellForceField. Добавьте ограничивающую сферу к классу и используйте её в функции ASpellForceField::Tick(), чтобы отталкивать монстров.

Что дальше? Я бы очень посоветовал, чтобы вы расширили нашу маленькую игру. Вот несколько идей для расширения:

- Создайте больше элементов окружения, расширьте местность, добавьте дома и другие здания
- Добавьте квесты, которые идут от NPC
- Определите больше оружия, такого как мечи
- Определите броню, защиту для игрока, например щиты
- Добавьте лавки, в которых продаётся оружие для игрока
- Добавьте больше типов монстров
- Осуществите трофеи, получаемые от монстров

У вас впереди буквально тысячи часов работы. Если так сложилось что вы программист одиночка, то сработайтесь с кем-то ещё. Вы не выживите на игровом рынке в одиночку.

Это опасно делать всё самому. Заведите друзей.

Выводы

Это была заключительная глава. Вы прошли долгий путь. От незнающего вообще ничего в C++ программировании, к надеюсь, будучи способными связывать основы программирования игр в UE4.