



# Мастерство MonoGame

Создайте мультиплатформенную  
2D-игру и многоразовый игровой  
движок

---

Джарред Капеллман  
Луи Салин

# **MonoGame Mastery**

**Создайте  
мультиплатформенную  
2D-игру и многоразовый  
игровой движок**

**Джарред Капеллман  
Луи Салин**

# Оглавление

<b>Об авторах .....</b>	<b>xi</b>
<b>Предисловие.....</b>	<b>xvii</b>
<b>Глава 1 Введение.....</b>	<b>1</b>
Для кого эта книга? .....	2
Чем эта книга не является .....	2
Предположения читателей.....	3
Что такое моноигра .....	4
MonoGame по сравнению с движками .....	6
Типы игр, наиболее подходящие для моноигры .....	6
Вертикальные шутеры .....	7
Горизонтальные шутеры .....	8
Боковые скроллеры.....	10
Ролевые игры.....	11
Головоломка .....	12
Стратегия .....	13
Организация этой книги .....	14
Образцы кода .....	15
Резюме.....	16

## TABLE OF CONTENTS

<b>Глава 2: Настройка среды разработки .....</b>	<b>17</b>
Конфигурация среды разработки .....	18
Независимость от платформы.....	18
Настройка среды разработки Windows .....	18
Настройка среды разработки macOS .....	22
Настройка среды разработки Linux .....	27
Дополнительные инструменты.....	28
Инструменты .....	29
Расширения Visual Studio .....	32
Резюме.....	34
<b>Глава 3: Архитектура MonoGame .....</b>	<b>35</b>
Архитектура MonoGame .....	35
Конвейерное приложение.....	35
Класс игры.....	38
Ваши первые визуализированные пиксели .....	40
Создание решения и проекта.....	40
Погружение в проект.....	43
Погружение в MainGamecs .....	45
Порядок выполнения.....	49
Резюме.....	50
<b>Глава 4: Планирование вашего игрового движка .....</b>	<b>51</b>
Дизайн игрового движка .....	51
Ввод игрока .....	52
Искусственный интеллект (ИИ) .....	53
Триггеры событий.....	53
Графический рендеринг .....	53
Рендеринг звука .....	54

## TABLE OF CONTENTS

<b>Физика .....</b>	54
<b>Управление состоянием .....</b>	54
<b>Реализация архитектуры движка.....</b>	58
<b>Создание проекта.....</b>	58
<b>Создание классов состояний .....</b>	59
<b>Создание масштабатора и управления окнами .....</b>	62
<b>Система событий.....</b>	71
<b>Резюме.....</b>	74
 <b>Глава 5: Конвейер активов .....</b>	75
<b>Конвейер активов MonoGame.....</b>	75
<b>Класс ContentManager .....</b>	76
<b>Инструмент конвейера MonoGame .....</b>	78
<b>Интеграция конвейера активов в движок .....</b>	79
<b>BaseGameState .....</b>	79
<b>MainGame .....</b>	81
<b>Добавить спрайт игрока в игру .....</b>	82
<b>Обзор новых активов .....</b>	83
<b>Добавление новых ресурсов к нашему контенту.....</b>	85
<b>Изменения игрового кода .....</b>	89
<b>Запуск приложения.....</b>	91
<b>Резюме.....</b>	93
 <b>Глава 6: Ввод.....</b>	95
<b>Обсуждение различных механизмов ввода .....</b>	96
<b>Состояние клавиатуры.....</b>	96
<b>Состояние мыши .....</b>	100
<b>Состояние геймпада .....</b>	101

## TABLE OF CONTENTS

Прокрутка фона.....	103
Создание универсального диспетчера ввода .....	111
Стрельба пулями .....	118
Резюме.....	123
<b>Глава 7: Аудио .....</b>	<b>125</b>
Рефакторинг движка.....	125
Организация кода .....	131
Звук.....	133
Воспроизведение саундтрека.....	135
Звуковые эффекты .....	140
Резюме.....	145
<b>Глава 8: Частицы.....</b>	<b>147</b>
Анатомия частицы.....	149
Обучение с помощью онлайн-редактора частицы.....	150
Различные формы излучателей частиц .....	152
Добавление системы частиц в нашу игру.....	153
Частица .....	154
EmitterParticleState .....	159
IEmitterType.....	163
ConeEmitterType.....	164
Излучатель.....	166
Добавление следов от ракет и дыма в нашу игру .....	172
Создание игрового состояния Dev для игры.....	172
Добавление игрового объекта «Ракета» в игру .....	184
Резюме.....	189

<b>Глава 9: Обнаружение столкновений .....</b>	<b>191</b>
Техники .....	193
AABB (границчная рамка, выровненная по оси) .....	195
OBB (ориентируемая ограничивающая рамка) .....	196
Сфера .....	198
Равномерные сетки .....	199
Дерево квадрантов .....	200
Другие техники .....	203
Добавление врагов в нашу игру .....	203
Вращение нашего вертолета .....	206
Вращающиеся лопасти .....	208
Заставляем вертолеты двигаться .....	209
Добавление инструмента взрывных частиц .....	216
Добавление обнаружения столкновений .....	219
Ограничивающие рамки .....	220
Обнаружение столкновений AABB .....	229
Резюме .....	234
<b>Глава 10: Анимация и текст .....</b>	<b>237</b>
Немного рефакторинга .....	238
Анимации .....	242
Листы спрайтов .....	242
Атлас текстур .....	244
Недостатки анимации .....	245
Конечные автоматы .....	246
Движок анимации .....	248
Анимация нашего истребителя .....	255

## TABLE OF CONTENTS

<b>Текст .....</b>	<b>262</b>
<b>Шрифты.....</b>	<b>262</b>
<b>Добавление шрифтов в конвейер контента .....</b>	<b>263</b>
<b>Шрифты как игровые объекты.....</b>	<b>264</b>
<b>Отслеживание жизней.....</b>	<b>266</b>
<b>Game Over .....</b>	<b>267</b>
<b>Резюме.....</b>	<b>270</b>
 <b>Глава 11: Дизайн уровней.....</b>	<b>271</b>
<b>Редакторы уровней .....</b>	<b>272</b>
<b>Что такое уровень? .....</b>	<b>273</b>
<b>События уровня .....</b>	<b>275</b>
<b>Читатели уровней и состояние игрового процесса .....</b>	<b>277</b>
<b>Добавление турелей.....</b>	<b>286</b>
<b>Игровое искусство и происхождение.....</b>	<b>287</b>
<b>Турельные пули .....</b>	<b>299</b>
<b>Обнаружение столкновений.....</b>	<b>306</b>
<b>Уборка .....</b>	<b>311</b>
<b>Добавление текста .....</b>	<b>311</b>
<b>Обзор нашего дизайна уровней .....</b>	<b>312</b>
<b>Улучшение игрового процесса .....</b>	<b>313</b>
<b>Резюме.....</b>	<b>315</b>

# Об авторах



Джарред Капельман профессионально занимается разработкой программного обеспечения более 14 лет и является директором по проектированию в SparkCognition в Остине, штат Техас. Он начал делать текстовые игры QBasic, когда ему было 9 лет. Несколько лет спустя Хел выучил C++, прежде чем изучать OpenGL с конечной целью войти в игровую индустрию. Хотя его цель профессионально разрабатывать игры не осуществилась, он продолжал глубоко погружаться в такие

фреймворки, как MonoGame, Vulkan и DirectX, занимая важную часть своего свободного времени.

В свободное от программирования время он любит писать музыку и работает над докторской диссертацией в области кибербезопасности, уделяя особое внимание применению машинного обучения к угрозам безопасности.



Луи Салин был разработчиком более 15 лет в самых разных областях, сначала разрабатывая для Windows на C, C++ и, в конечном итоге, C#, а затем работал разработчиком веб-приложений на базе Linux с использованием различных языков сценариев, таких как Ruby и Python. Его ранняя любовь к кодированию возникла из-за того, что в детстве он копировал видеоигры, написанные на языке Basic, из книг, взятых в библиотеке. Свою первую игру он написал в старшей школе и посещал многие занятия по компьютерной графике.

# Предисловие

Создание видеоигр имеет неоспоримую привлекательность в воображении многих людей из разных слоев общества. Многие дети хотят стать гейм-дизайнерами, когда вырастут, и многие программисты научились искусству написания кода, думая, что однажды они создадут свою собственную игру.

Создание видеоигры — это одновременно и выразительная форма искусства, и ряд логических задач, которые необходимо решить.

Программист игр должен проявлять творческий подход, материализуя содержание своего воображения на мониторе компьютера, и в то же время он должен постоянно преодолевать любые физические ограничения, наложенные на него при формировании своей игры. Для тех из нас, кто любит решать проблемы и имеет склонность к искусству, это поле мечты, будь то хобби или профессиональная работа на полный рабочий день. Никогда не было лучшего времени для обычных людей, чем сегодня, чтобы писать видеоигры! В то время как любители по всему миру создавали игры с 1970-х годов, объем требуемых глубоких технических знаний уменьшился, а входной барьер упал намного ниже.

Последние несколько лет. Игровые инструменты и игровые движки теперь избавляют от сложностей, связанных с отрисовкой чего-либо на экране, а компьютеры стали настолько мощными, что программистам больше не нужно быть столь точными в том, как они управляют памятью и производительностью игры. Кроме того, публикация видеоигры на игровых консолях и компьютерах сегодня стала намного более доступной для любого, кто проявит настойчивость, чтобы довести свою игру до конца, что видно из огромного количества инди-игр, представленных на рынке.

Разработчики программного обеспечения сегодня имеют широкий выбор технологий при создании своей игры. Одним из таких вариантов является MonoGame, фреймворк для создания мощных кроссплатформенных игр.

В этой книге мы стремимся провести опытных программистов на C# в путешествие, объясняя основы разработки игр и создавая с нуля небольшую двумерную вертикальную видеоигру-шутер, используя MonoGame в качестве основы.

## ВВЕДЕНИЕ

К концу этой книги наши читатели не только создадут многоразовый игровой движок, который они смогут использовать в своих будущих играх, но также получат ценные знания, которые помогут им в их будущих проектах, какой бы фреймворк или движок они ни выбрали для его использования.

## ГЛАВА 1

# Вступление

Скорее всего, прочитав эту вступительную строку, вы как минимум будете заинтригованы тем, чтобы начать изучать разработку игры с нуля. Эта книга была написана для того, чтобы привести вас от этой мысли к реализации с использованием MonoGame Framework. Мы начнем с того, что предоставим вам, читатель, прочную основу архитектуры MonoGame и продолжим со спрайтами, звуком и обнаружением столкновений, а затем закончим разделением проблем, подготовив вас к будущим разработкам. В отличие от других книг по разработке игр, эта книга будет развиваться с каждой главой, опираясь на предыдущую, с подходом, основанным на проектах, а не на кусках кода тут и там. В этой книге мы начнем с нуля с вертикального шутера, похожего на те, что были в конце 1980-х и начале 1990-х годов. Вертикальный шутер — отличная отправная точка для начинающих разработчиков игр, поскольку он содержит все элементы, которые можно найти в современных играх:

- Многослойные прокручиваемые фоны
- Обнаружение столкновений снарядов и врагов
- Враги, управляемые компьютером
- Спрайты
- Ввод игрока
- Звуковые эффекты на основе событий
- Уровневая структура

## ГЛАВА 1 ВВЕДЕНИЕ

Кроме того, эта книга расскажет о соответствующем дизайне движка и игровых инструментах, которые, возможно, упускаются из виду во многих книгах по разработке игр.

В этой главе вы узнаете о

- MonoGame на высоком уровне
- Разница между MonoGame и игровыми движками
- Типы игр, подходящие для MonoGame
- Чего ожидать от книги и превью будущих глав

## Для кого эта книга?

Эта книга предназначена для начинающих разработчиков игр, которые хотят создать 2D-игру. Будут предоставлены бесплатные игровые ресурсы для звука, музыки, текстур и спрайтов (все они созданы вашим покорным слугой), что позволит сосредоточить внимание на программных и архитектурных компонентах разработки игр, не беспокоясь о проворачивании игровых ресурсов.

## Чем эта книга не является

Хотя мы рассмотрим типы игр в зависимости от того, что хорошо сочетается с MonoGame, в этой книге не рассматриваются принципы игрового дизайна, создание ресурсов или жизненный цикл разработки игр. Существует множество доступных ресурсов, включая целые книги, посвященные этим отдельным компонентам, но они выходят за рамки этой книги.

## Предположения читателя

Хотя опыт разработки игр не требуется, ожидается, что вы являетесь опытным программистом на C#. Хотя с MonoGame легко начать работу благодаря архитектуре и простому дизайну, фреймворк написан на C#. Кроме того, проект, который мы будем повторять в этой книге, использует многие основные аспекты языка программирования C#, такие как наследование и отражение. Если вы обнаружите, что просматриваете прилагаемый исходный код и испытываете затруднения, я предлагаю взять книгу *C# Programming for Absolute Beginners* также от Apress, чтобы закрыть пробелы.

С точки зрения машины разработки в этой книге будет рассмотрено, как настроить среду разработки MonoGame как на macOS, так и на Windows с помощью Visual Studio. Linux также можно использовать в качестве среды разработки с Visual Studio Code; тем не менее, Windows будет предпочтительной средой для этой книги из-за инструментов, которые предлагает Visual Studio для Windows.

Благодаря тому, что все активы, предоставленные в этой книге, представлены в форматах, которые MonoGame может читать изначально (подробнее об этой функции в главе 5), никаких других инструментов не требуется. Опыт работы с такими инструментами, как Photoshop, 3ds Max и Audition, пригодится для ваших будущих усилий по разработке, даже если это просто начальный уровень навыков.

На момент написания этой статьи последней доступной рабочей версией MonoGame была версия 3.8, выпущенная 10 августа 2020 года. Эта версия будет использоваться для всех примеров кода и фрагментов в этой книге. Версии 3.8.x или более поздние могут быть доступны к тому времени, когда вы читаете это; однако, согласно дорожной карте, образцы должны продолжать работать без проблем.

## Что такое MonoGame

MonoGame на самом высоком уровне представляет собой C# Framework, который предоставляет разработчику холст для быстрого создания игры своей мечты. MonoGame имеет открытый исходный код (публичная лицензия Microsoft) и не требует лицензионных отчислений (более 1000 игр были опубликованы в различных магазинах). В то время как MonoGame предлагает поддержку 3D, сообщество в целом использует его мощную поддержку почти исключительно 2D, и это будет в центре внимания в этой книге (для 3D-игр рекомендуется использование Unity или Unreal Engine). Исходный код MonoGame доступен на GitHub (<https://github.com/MonoGame/MonoGame>).

Как и многие фреймворки и движки, доступные сегодня, MonoGame, как C#, является кроссплатформенным. В настоящее время MonoGame работает на:

- Рабочий стол Windows (7/8.x/10)
- Универсальная платформа Windows
- MacOS
- Linux
- PlayStation 4
- Xbox One
- Nintendo Switch
- Android (4.2 or later)
- iOS

Следует отметить, что для PlayStation 4, Xbox One и Nintendo Switch перед публикацией в соответствующих магазинах требуется дополнительные соглашения с разработчиками.

Учитывая то, что в основе MonoGame лежит C#, по мере того, как новые платформы будут поддерживаться C#, MonoGame не должна сильно отставать.

## ГЛАВА 1 ВВЕДЕНИЕ

На протяжении всей книги мы будем рассматривать любые вопросы, связанные с платформой, такие как разрешение и методы ввода (например, сенсорный экран или клавиатура). К счастью, разработка кроссплатформенной игры не требует больших предварительных усилий с помощью MonoGame.

MonoGame на высоком уровне обеспечивает

- Основной игровой цикл
  - Обработка обновлений
  - Метод рендеринга
- Контент менеджер
- Конвейер контента
- Поддержка OpenGL и DirectX.

Одной из лучших особенностей дизайна MonoGame является эта простота, в отличие от других фреймворков, которые имеют чрезвычайно сложную кривую обучения, чтобы даже отрисовать первый пиксель. В этой книге мы расширим эту структуру для поддержки более сложных сценариев и предоставим богатый расширяемый движок, который не только будет развиваться с каждой главой, но и обеспечит основу для создания вашей собственной игры. В главе 3 мы углубимся в эту архитектуру.

Опытные разработчики в этот момент могут задаться вопросом, какова связь между MonoGame и инфраструктурой Microsoft XNA. На высоком уровне прямой связи нет. Базовая структура, отмеченная ранее, сохраняется, а использование C# в качестве языка является тем местом, где происходит корреляция. MonoGame выросла из желания Хосе Антонио Леала де Фариаса в 2009 году создать XNA Touch. Подобно усилиям Mono Touch по переносу Mono на iOS, цель состояла в том, чтобы перенести XNA на iOS. К тому моменту XNA находилась в застое с выпуском версии 4.0 в 2010 году (которая будет последней выпущенной версией) и официальным заявлением о прекращении поддержки в 2013 году. После этого XNATouch был переименован в MonoGame, а вскоре после этого появилась поддержка Android, Mac и Linux. В конечном итоге MonoGame добралась до GitHub и на момент написания этой статьи имеет более 2200 форков с 267 участниками.

## MonoGame по сравнению с движками

Как уже упоминалось, MonoGame — это чистый фреймворк. С самого начала разработки целью MonoGame было создание гибкого, простого, но мощного фреймворка. Основная причина этого заключалась в том, чтобы позволить использовать MonoGame в широком диапазоне жанров и типов игр, в отличие от движка, который чаще всего адаптирован к определенному жанру (обычно это жанр, в котором игра, в которой разрабатывался движок, был такой как Quakeseries).

И наоборот, движок, подобный движку Unity, Unreal Engine или id Tech, предоставляет комплексный движок и редактор со всеми различными компонентами, составляющими игровой движок, такими как рендеринг, физика, редакторы уровней, конвейеры контента с интеграцией в программы моделирования. В зависимости от уровня отклонения от ядра движка, разработчику может быть очень мало, что нужно расширить самостоятельно. Подход, основанный на движке, позволяет команде художников и дизайнеров создать холст, готовый приступить к реализации игры, вместо того, чтобы ждать, пока программисты создадут движок с нуля или надстроят его на базе такой среды, как MonoGame. Кривые обучения и лицензионные сборы вышеупомянутых двигателей также должны быть приняты во внимание.

Если вы читаете эту книгу, скорее всего, вы хотите погрузиться на более низкий уровень с быстрой кривой обучения — эта книга должна достичь этого.

## Типы игр, наиболее подходящие для MonoGame

Как упоминалось ранее, MonoGame лучше всего подходит для 2D-игр. С возрождением классики 1980-х и 1990-х годов в дополнение к возвращению к простым, но забавным играм, таким как Castle Crashers, это не помеха, а даже преимущество, поскольку структура настроена для этих типов игр.

MonoGame можно использовать в самых разных играх; Ниже приведены несколько примеров типов, которые работают лучше всего. Кроме того, для каждого типа игры будут рассмотрены плюсы и минусы по сравнению с другими типами. При планировании игры взвешивание всех плюсов и минусов определенного типа должно быть основной частью ваших усилий по разработке. Для вашей первой игры после прочтения этой книги настоятельно рекомендуется выбрать более простой в реализации тип игры.

## Вертикальные шутеры

Вертикальные шутеры, популяризированные в Сарком 1942 и улучшенные в 1990-х годах по мере улучшения графики и игрового процесса, могут варьироваться от более научно-фантастических, таких как Major Stryker, до более приземленных, таких как Raptor. Как упоминалось ранее в этой главе, для этой книги мы создадим вертикальный шутер с нуля; скриншот игры из Главы 4 изображен на Рис.1-1.



*Рис. 1-1. Наша 2D-игра из главы 4*

## ГЛАВА 1 ВВЕДЕНИЕ

У разработки вертикальных шутеров есть некоторые преимущества и недостатки:

### Плюсы

- Легко загружаются.
- Элементы управления являются основными.
- Графика проста в реализации.
- Генерация уровней и инструменты просты.
- Легко реализовать.

### Минусы

- Усталый жанр
  - Нужно создать какой-то уникальный геймплей, чтобы он отличался от Raptor и других известных вертикальных шутеров.

## Горизонтальные шутеры

Стала популярной благодаря таким играм, как Einhander в 1990-х годах, похожа на вертикальный шутер, но дает больше разнообразия в игровом процессе. Отличным примером этого в MonoGame является Paladin от Pumpkin Games в Fig. 1-2.

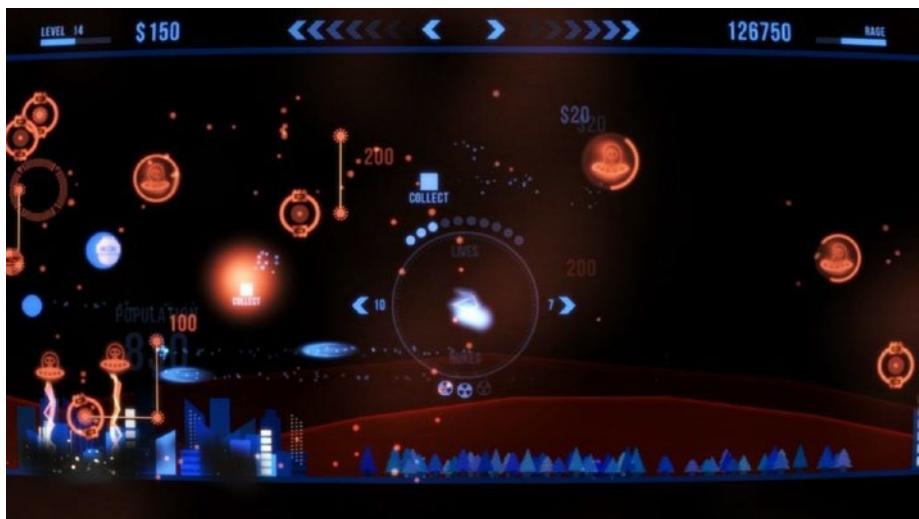


Рис. 1-2. Паладин из *Pumpkin Games*

У разработки горизонтальных шутеров есть некоторые преимущества и недостатки:

#### Плюсы

- Легко загружаются
- Элементы управления являются основными.
- Генерация уровней и инструменты просты.
- Легко реализовать.

#### Минусы

- Графика в этом жанре должна быть высокой из-за конкуренции.
- Пресыщенный жанр
  - Необходимо создать уникальный игровой процесс, чтобы отличать его от других игр.

## Боковые скроллеры

Боковые скроллеры — это жанр, который зародился в конце 1980-х годов и существует по сей день, предлагая широкий спектр приключенческих и экшн-игр с горизонтальной точки зрения. Встроенная в MonoGame поддержка спрайтов и 2D-графики с аппаратным ускорением сделала этот выбор легким для разработки. Tasmanian Tiger 4 от Krome Studios — отличный пример плавной анимации и быстрых действий с использованием MonoGame, как показано на Рис. 1-3.



*Рис. 1-3. Тасманский тигр 4 от Krome Studios*

У разработки сайдскроллеров есть некоторые преимущества и недостатки:

Плюсы

- Разнообразный геймплей достижим.

Минусы

- Графику может быть сложно реализовать в зависимости от игрового процесса.

- AI также может быть сложным в зависимости от игрового процесса.
- Инструменты также могут быть обременительны для разработки.

## Ролевые игры

2Disometric view, ставший популярным благодаря серии Final Fantasy для Super Nintendo, с тех пор используется в 2D-ролевых играх.

Популярным примером этого типа игры с MonoGame является Stardew Valley от ConcernedApe, как показано на рис. 1-4.



Рис. 1-4. Долина Звездной Росы Обеспокоенной Обезьяны

### Плюсы

- Достигим разнообразный геймплей.
- AI может быть легко реализован (в зависимости от уровня взаимодействия NPC).

## ГЛАВА 1 ВВЕДЕНИЕ

### Минусы

- Графическая обработка тайлов и спрайтов может быть громоздкой.
- Инструменты также могут быть обременительны для разработки.

## Головоломка

Игры-головоломки, особенно на мобильных устройствах, учитывая популярность Angry Birds и Bejeweled среди других в последние годы в сочетании с простотой использования MonoGame, идеально подходят. Примером такого типа игры с использованием MonoGame является Raining Blobs Энди Милойкоски, как показано на Рис. 1-5.



*Рис. 1-5. Капли дождя Энди Милойкоски*

### Плюсы

- Разнообразный геймплей достижим.
- Графика может быть легко реализована.

- AI может быть легко реализован.
- Инструменты также могут быть просты в реализации.

Минус

- Добиться уникального и/или увлекательного геймплея на переполненном рынке может быть чрезвычайно сложно.

## Стратегия

Стратегические игры обычно варьируются между пошаговыми играми, играми в реальном времени и гибридами стратегических и ролевых игр. Хотя их намного сложнее разработать и реализовать, они могут предоставить геймерам уникальный опыт. Wayward Terran Frontier от ReasonGenerator Inc — хороший пример использования MonoGame в полной мере на Рис. 1-6.



Рис. 1-6. Заблудшие терранские рубежи Reason Generator Inc.

### Плюс

- Разнообразный геймплей достижим.

### Минус

- Графика может быть сложной для реализации в зависимости от игрового процесса.
- AI также может быть сложным в зависимости от игрового процесса.
- Инструменты также могут быть обременительны для разработки.

## Организация этой книги

Как было сказано в начале этой главы, в этой книге каждая из тем разбита на управляемые и изолированные главы. Ниже приводится обзор книги и тем, которые мы затронем:

Глава 2 подробно описывает, как настроить среду разработки от начала до конца для оставшейся части книги. К концу главы вы сможете запустить пустой проект MonoGame. Подробно будет рассмотрена установка как для macOS, так и для Windows. Linux будет обсуждаться, но не рекомендуется в остальной части книги.

Глава 3 глубоко погружается в архитектуру MonoGame, включая подробное описание 2D-графики, игрового таймера и ввода. Эту главу не следует упускать из виду, даже если вы занимались разработкой игр в прошлом, так как она даст глубокое представление о том, как устроена архитектура MonoGame.

В главе 4 начинается глубокое погружение в создание архитектуры, на основе которой мы будем строить оставшуюся часть книги. Как и в главе 3, эту главу нельзя пропускать, поскольку в ней будут подробно описаны объекты, менеджеры и изменения классов Game.

В [главе 5](#) подробно описывается, как конвейер ассетов работает в MonoGame. Кроме того, будет подробно описана интеграция с Content Manager в Game States. В конце главы мы отрендерим наш первый спрайт.

[Глава 6](#) описывает обработку ввода с помощью клавиатуры и мыши. Кроме того, будут рассмотрены особенности платформы для обработки ввода с геймпада и сенсорного экрана.

В [главе 7](#) рассказывается, как добавить звук в нашу архитектуру и как добавить звуковые триггеры в нашу систему событий. Кроме того, также будут обсуждаться поддерживающие слои фоновой музыки.

В [главе 8](#) подробно рассматривается, как интегрировать частицы в нашу архитектуру, чтобы обрабатывать пули как из нашего объекта игрока, так и из настроек будущих вражеских объектов.

В [главе 9](#) рассматриваются различные методы обнаружения столкновений, используемые в играх. В нашем проекте мы будем использовать столкновение блоков и интегрировать его в нашу архитектуру для обработки не только столкновений объектов игрока, но и столкновений снарядов.

[Глава 10](#) добавляет анимацию в нашу архитектуру и рассматривает подходы, используемые в отрасли. В конце главы в игру добавляются анимации объектов.

[Глава 11](#) рассматривает важность дизайна уровней и подробно описывает, как добавить загрузку уровней в наш игровой движок.

## Примеры кода

Образцы кода, начиная с главы 3, будут упоминаться в каждом разделе. Помимо примеров кода, существует также архив активов, который содержит всю музыку, звуковые эффекты, спрайты и графику, использованные в книге.

## Резюме

В этой главе вы узнали, что такое MonoGame и чем отличаются MonoGame от игровых движков. Кроме того, вы узнали о типах игр, подходящих для MonoGame, и о структуре глав книги.

Далее нужно настроить среду разработки, чтобы начать освоение MonoGame.

## ГЛАВА 2

# Настройка среды разработки

Теперь, когда начальное знакомство с MonoGame завершено, вы готовы приступить к программированию, верно? Не совсем так — нам нужно сначала настроить среду разработки в Visual Studio. К счастью, с годами этот процесс стал гораздо более упорядоченным и не займет много времени. Кроме того, будет рассмотрена настройка Linux для разработки MonoGame; однако, как упоминалось в Главе 1, Windows будет использоваться для всех снимков экрана и включенного примера кода. Как и в большинстве разработок, для разработки можно использовать один инструмент, но есть дополнительные инструменты, плагины и расширения для повышения производительности. Чтобы помочь вам в этом путешествии, в конце этой главы я рассмотрю несколько инструментов, которые помогают мне как в игровой, так и в обычной практике.

В этой главе вы узнаете

- Как настроить среду разработки (Windows, macOS и Linux).
- Поговорим о дополнительных инструментах.
- Рассмотрим расширения Visual Studio.

## Независимая от платформы конфигурация среды разработки

Независимо от используемой платформы все пакеты можно найти на странице [www.monogame.net/downloads/](http://www.monogame.net/downloads/). Как упоминалось в главе 1, на момент написания этой статьи во всех примерах кодирования использовалась версия 3.8. Если к тому времени, когда вы читаете это, появится более поздняя версия и возникнут проблемы с запуском примеров, установите 3.8.0.1641.

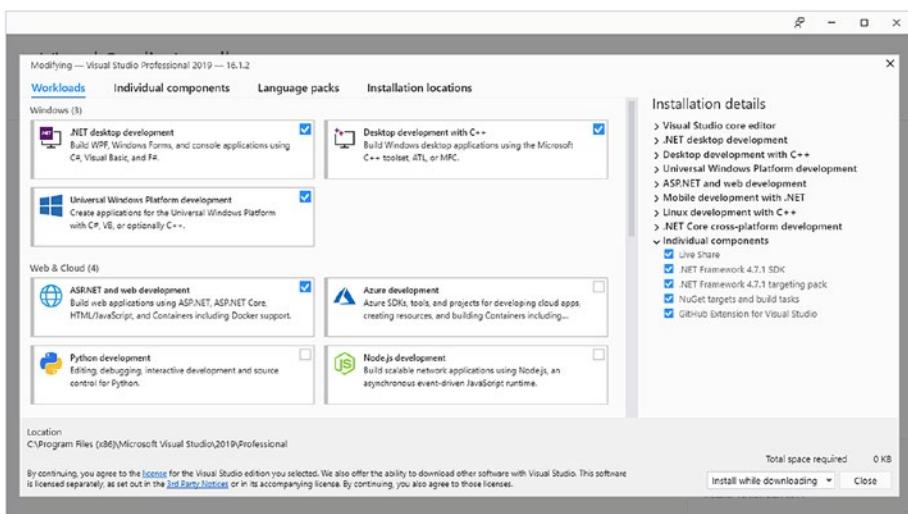
Вы можете скачать MonoGame 3.8 здесь:

<https://github.com/MonoGame/MonoGame/releases/tag/v3.8>.

## Настройте среду разработки Windows

Первым требованием в Windows является установка Visual Studio. Для примеров в этой книге используется Visual Studio 2019 Professional Edition (версия 16.7.2). Версия для сообщества находится в свободном доступе на <https://visualstudio.microsoft.com/>. При установке обязательно установите настольную разработку .NET, как показано на Рис. 2-1. Если вы планируете нацеливаться на UWP или мобильные устройства, также установите разработку на универсальной платформе Windows и разработку для мобильных устройств с .NET соответственно.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ



**Рис. 2-1. Установщик Visual Studio**

Я настоятельно рекомендую установить Visual Studio на SSD или NVMedrive (если есть), так как запуск Visual Studio, особенно процесс компиляции MonoGame, значительно медленнее на традиционном диске. Кроме того, убедитесь, что доступно не менее 8 ГБ ОЗУ.

После установки Visual Studio 2019 на упомянутой ранее странице загрузок щелкните [MonoGame 3.8 для Visual Studio](#) по ранее указанной ссылке GitHub. После загрузки установщика шаблона MonoGame для MonoGame (в частности, MonoGame.Templates.CSharp.3.8.0.1641.vsix) использовался для следующих снимков экрана на Рис. с 2-2 по 2-5), осталось пройти несколько экранов. Они подробно описаны ниже.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

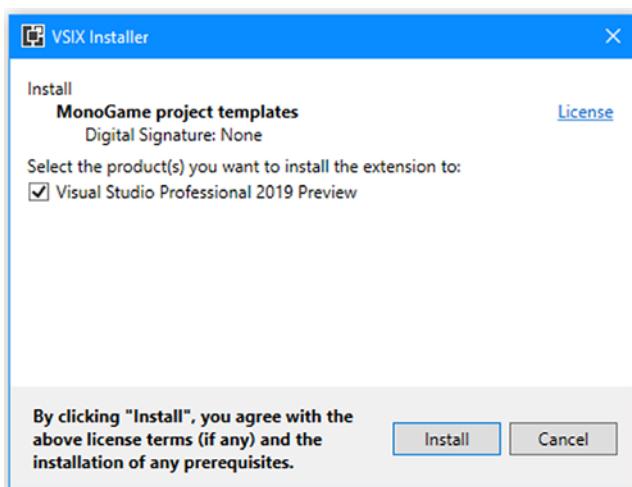


Рис. 2-2. Установка шаблона MonoGame для Windows

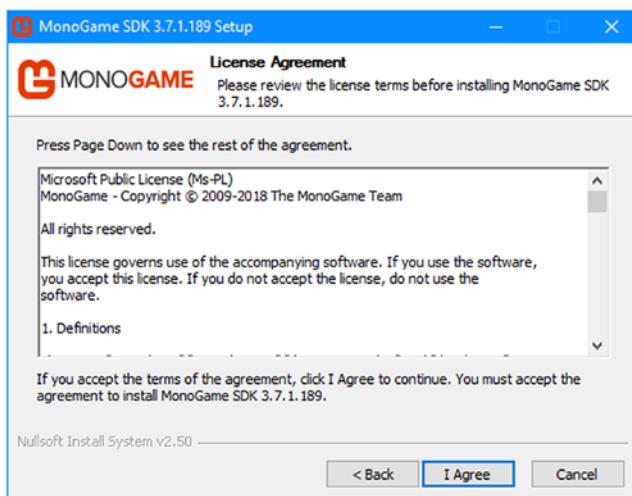
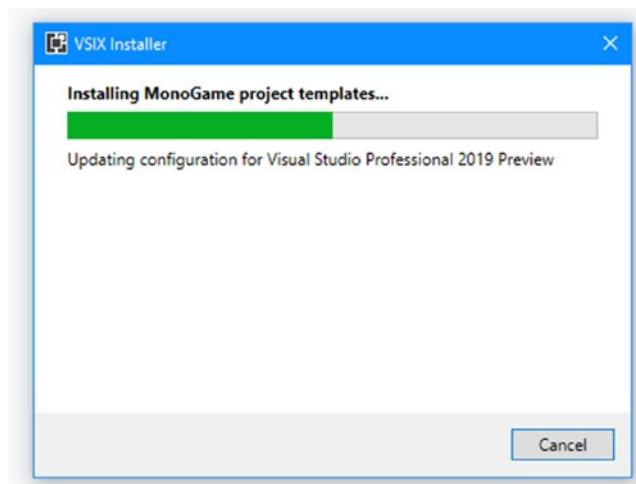
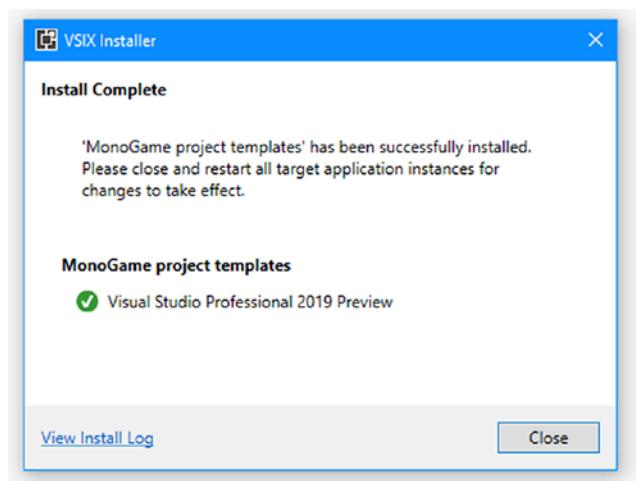


Рис. 2-3. Лицензионное соглашение



*Рис. 2-4. Установка шаблонов проекта*



*Рис. 2-5.Установлены шаблоны MonoGame*

Основным изменением в версии 3.8 по сравнению с предыдущими выпусками стал переход от традиционного установщика к пакетам NuGet и простая установка vsix для шаблонов.

Вам нужно только выбрать Visual Studio 2017 даже при использовании VisualStudio 2019 (конкретная поддержка предусмотрена в следующем выпуске).

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

После завершения, как показано на рис. 2-5, ваша настройка для Windows почти готова к работе.

Если типы проектов MonoGame не отображаются, убедитесь, что вы точно выполнили предыдущие шаги.

## Настройка среды разработки macOS

Первым требованием для macOS является установка Visual Studio для Mac. Примеры, используемые в этой книге, были протестированы на соответствие формату Visual Studio 2019 (версия 8.7.4). Версия для сообщества находится в свободном доступе на <https://visualstudio.microsoft.com/>.

После установки Visual Studio для Mac на упомянутой ранее странице загрузок щелкните ссылку загрузки с префиксом [MonoDevelop.MonoGame\\_IDE\\_VisualStudioForMac](#). После загрузки установщика нужно пройти несколько экранов. Они подробно описаны ниже.

Первый шаг — открыть Visual Studio 2019 для Mac, как показано нарис. 2-6.

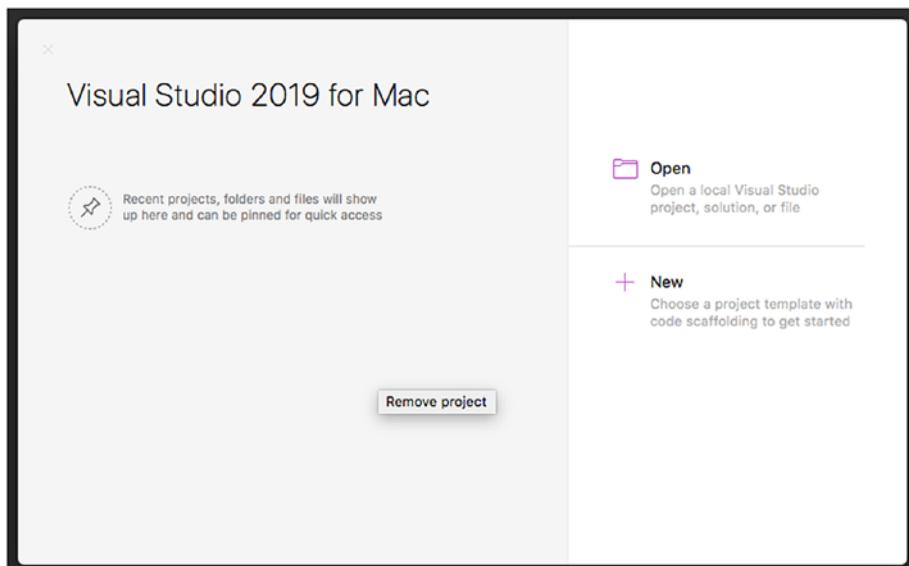
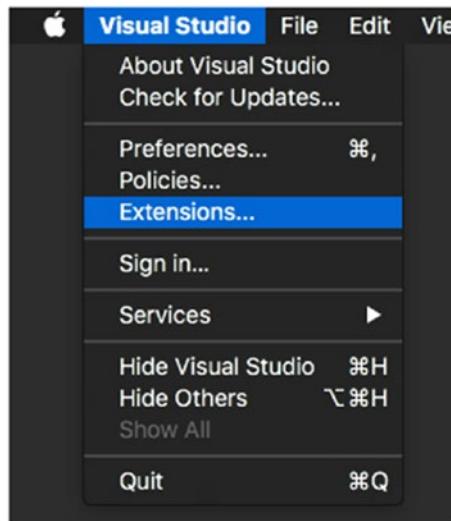


Рис. 2-6. Скрин начальной версии Visual Studio 2019 для Mac

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

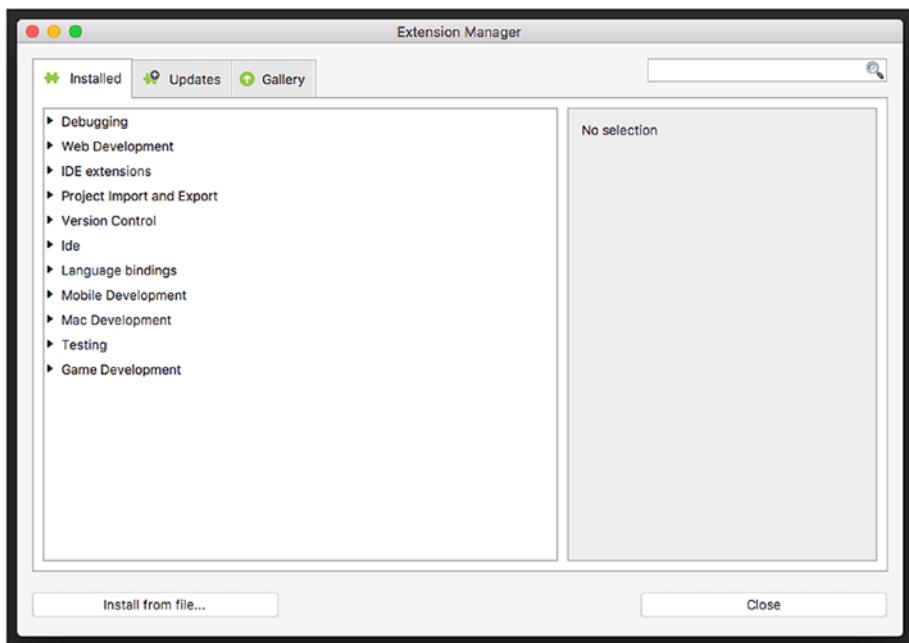
На этом экране перейдите к верхнему меню и выберите «Extensions - Расширения», как показано на Рис. 2-7.



*Рис. 2-7. Visual Studio 2019 для меню, показывающего выбор расширений*

После выбора Extensions Extension Manager - Менеджер расширений будет отображаться, как показано на Рис. 2-8.

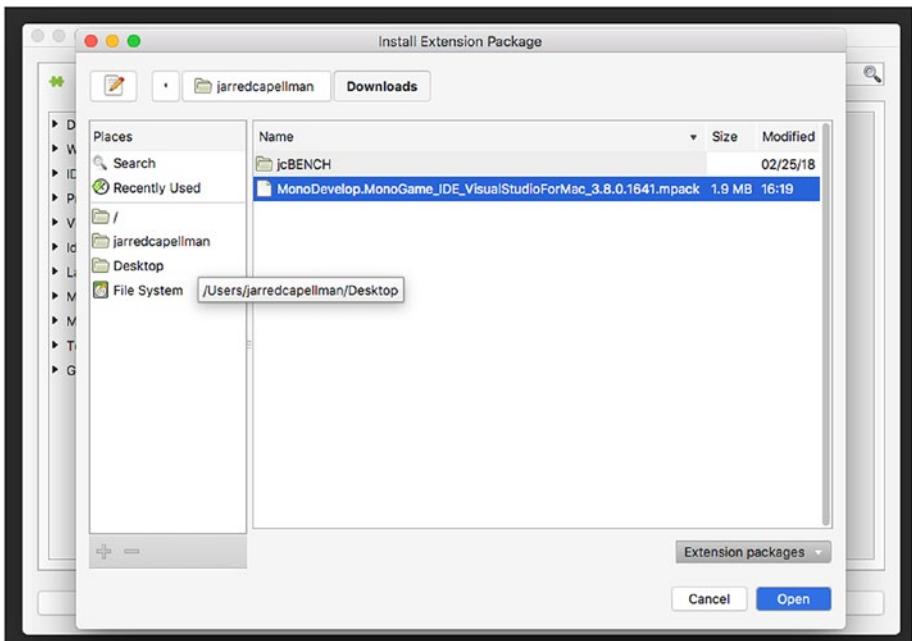
## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ



*Рис. 2-8. Visual Studio 2019 для Mac с Extension Manager*

Нажмите «Install from file... - Установить из файла...», после чего откроется окно выбора файла, как показано на Рис. 2-9.

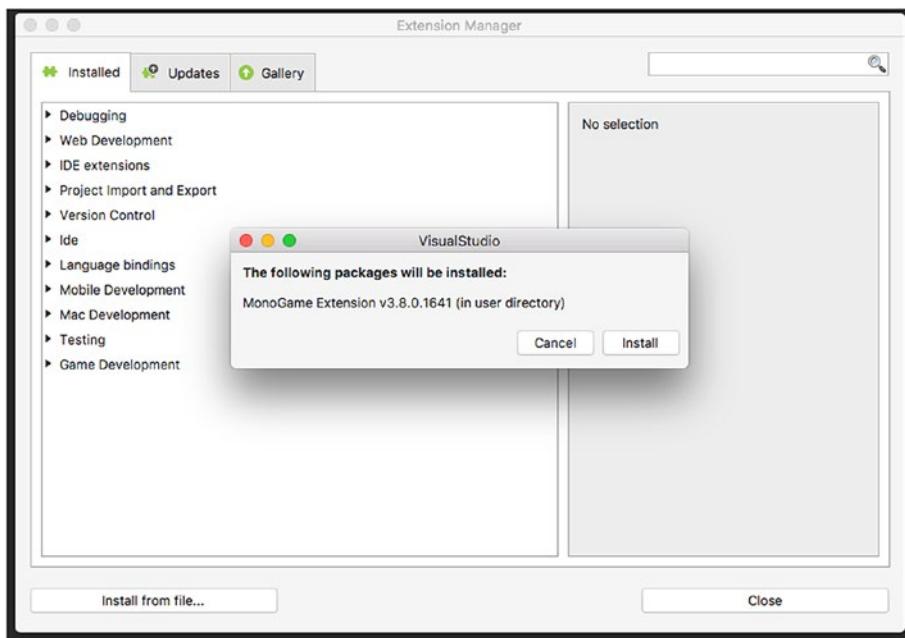
## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ



*Рис. 2-9. Visual Studio 2019 для Mac Окно выбора файла расширения*

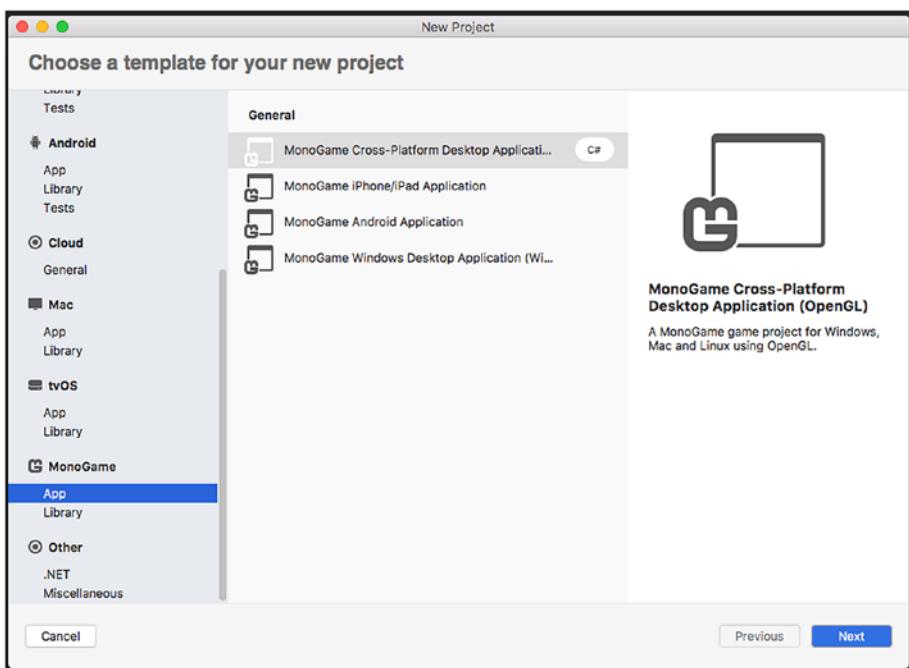
В этом окне найдите, куда вы загрузили **mppackfile** на первом шаге. После нажатия кнопки «Open - Открыть» после выбора появится окно подтверждения, как показано на Рис. 2-10.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ



*Рис. 2-10. Рис. 2-10. Visual Studio 2019 для Mac MonoGame Extension confirmation - Подтверждение расширения MonoGame*

Через несколько секунд установка завершится. Нажмите кнопку «Close - Закрыть», чтобы закрыть диспетчер расширений. После создания нового проекта вы увидите шаблоны MonoGame, как показано на Рис. 2-11.



**Рис. 2-11.** Visual Studio 2019 для Mac с шаблонами -templates MonoGame

Поздравляем, ваш компьютер для разработки macOS теперь подходит для использования с MonoGame!

## Настройка среды разработки Linux

Для разработки под Linux необходимо будет использовать Visual Studio Code. Visual Studio Code — это редактор с открытым исходным кодом, который работает в Windows, macOS и Linux.

Первый шаг — загрузить Visual Studio Code с <https://code.visualstudio.com/>. На момент написания этой книги 1.48.0 является последней производственной сборкой, и эта версия будет использоваться в оставшейся части этой книги. Кроме того, в качестве дистрибутива будет использоваться Ubuntu 18.04 LTS. Другие популярные дистрибутивы, такие как Debian и openSUSE, должны быть похожи, если не идентичны, в дополнение к более новым версиям Ubuntu.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

Начиная с MonoGame 3.8 и перехода на использование .NET Core в качестве основы в отличие от Mono, необходимо установить [.NET Core 3.1](#), поскольку он, скорее всего, не установлен в вашем дистрибутиве Linux по умолчанию.

Первый шаг — добавить репозиторий в ваш дистрибутив следующим образом:

```
wget https://packages.microsoft.com/config/ubuntu/20.04/
packages-microsoft-prod.deb -O /tmp/packages-microsoft-prod.deb
sudo dpkg -i /tmp/packages-microsoft-prod.deb
sudo apt update
```

После добавления репозитория можно установить пакет SDK для .NET Core:

```
sudo apt-get install -y apt-transport-https
sudo apt-get install -y dotnet-sdk-3.1
```

Вы можете проверить установку .NET Core, выполнив команду `dotnet --info` в терминале.

После установки .NET Core 3.1 при вызове следующего кода будут установлены шаблоны MonoGame:

```
dotnet new --install MonoGame.Templates.CSharp
```

После установки ваша среда будет настроена для разработки в Linux.

## Дополнительные инструменты

Простого наличия Visual Studio и MonoGame будет достаточно, чтобы начать свой путь к созданию игры и следовать этой книге; однако есть несколько инструментов и расширений Visual Studio для ускорения и улучшения процесса разработки. Ниже приведены некоторые инструменты, которые я ежедневно использую как для общей разработки, так и для разработки игр.

# Инструменты

## Инструменты управления исходным кодом

### GitHub (Windows, macOS, Linux)

Есть несколько бесплатных инструментов, которые я настоятельно рекомендую установить в дополнение к Visual Studio и MonoGame. Однако первое, что я бы посоветовал, — это создать учетную запись GitHub (<https://github.com>). После того, как Microsoft приобрела GitHub, частные репозитории стали бесплатными и предоставляют вам, как итеративному разработчику, контроль над исходным кодом для вашей работы. Кроме того, недавние дополнения, обеспечивающие бесплатную CI/CD (непрерывную интеграцию и непрерывное развертывание), делают платформу еще более привлекательной для проектов с открытым исходным кодом.

### TortoiseGit (только для Windows)

Хотя Visual Studio предлагает встроенную поддержку Git, я предпочитаю проект с открытым исходным кодом TortoiseGit (<https://tortoisegit.org/>). Параллельные инструменты сравнения, разрешение конфликтов слияния и интеграция с проводником Windows, как мне кажется, обеспечивают лучший контроль над исходным кодом, а также дают мне дополнительный процесс проверки вне Visual Studio перед отправкой запроса на включение или фиксацию. См. Рис. 2-12.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

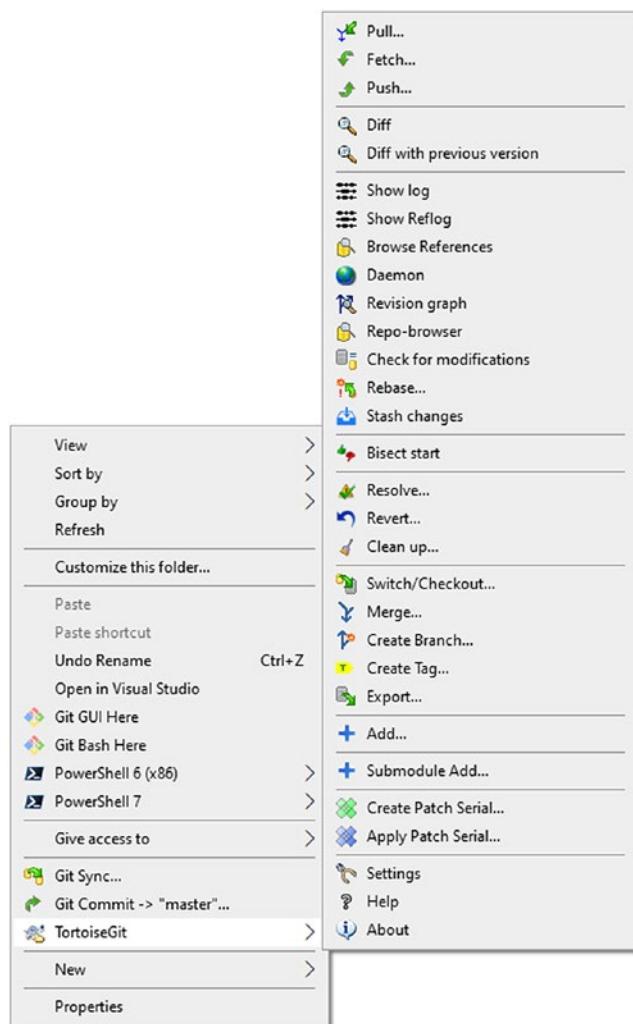
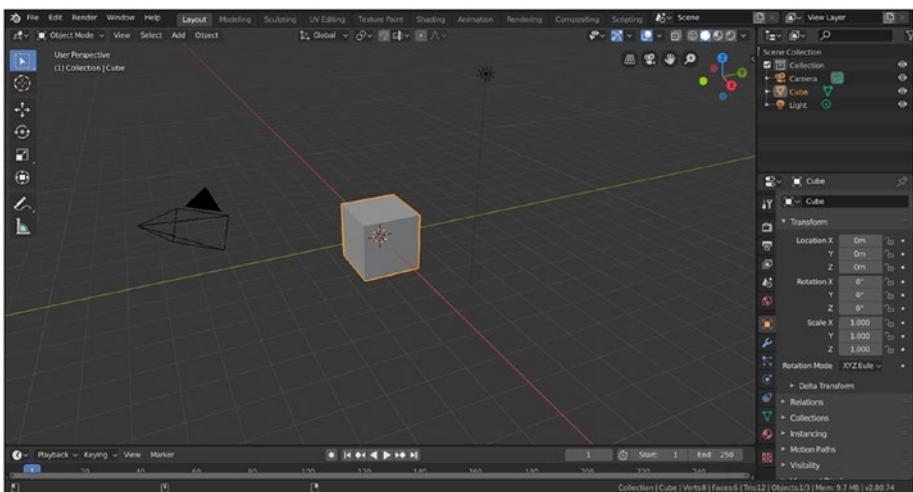


Рис. 2-12. Контекстное меню проводника Windows

## Графические инструменты

### Блендер (Windows, macOS, Linux)

Хотя существует несколько доступных приложений для 3D-моделирования, Blender ([www.blender.org/](http://www.blender.org/)), версия 2.8 на момент написания этой статьи, является отличным бесплатным пакетом для 3D-моделирования и анимации. Кроме того, Blender поддерживает множество форматов файлов как для импорта, так и для экспорта, что позволяет вам или художнику создавать ресурсы для вашего проекта. См. Рис. 2-13.

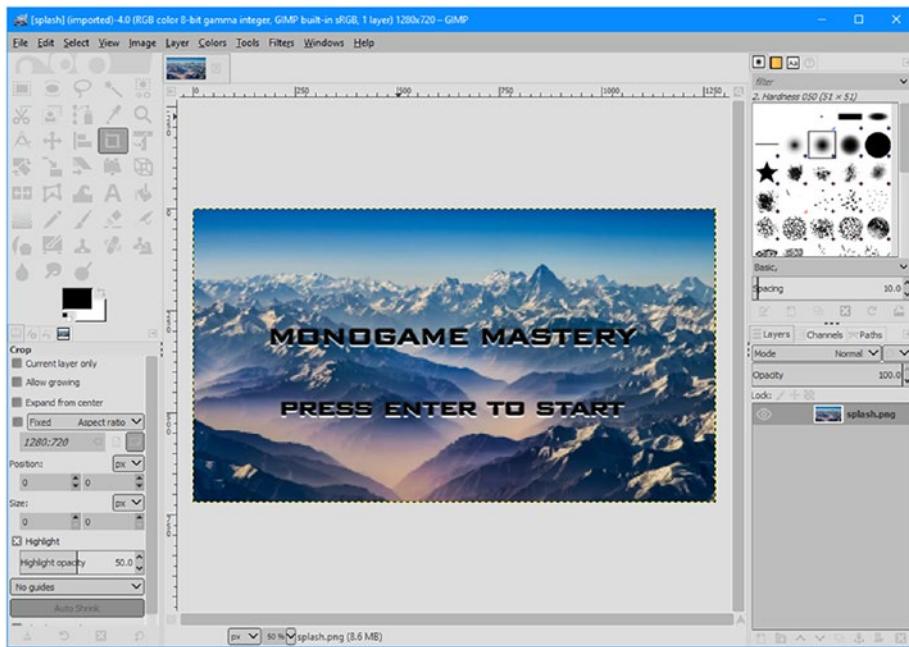


*Рис. 2-13. Блендер 2.8*

### Gimp (Windows, macOS, Linux)

Gimp ([www.gimp.org/](http://www.gimp.org/)), версия 2.10.12 на момент написания этой статьи, предлагает простой в использовании редактор изображений с открытым исходным кодом. Кроме того, будучи кроссплатформенным, Gimp поддерживает все основные форматы файлов как для импорта, так и для экспорта. См. Рис. 2-14.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ



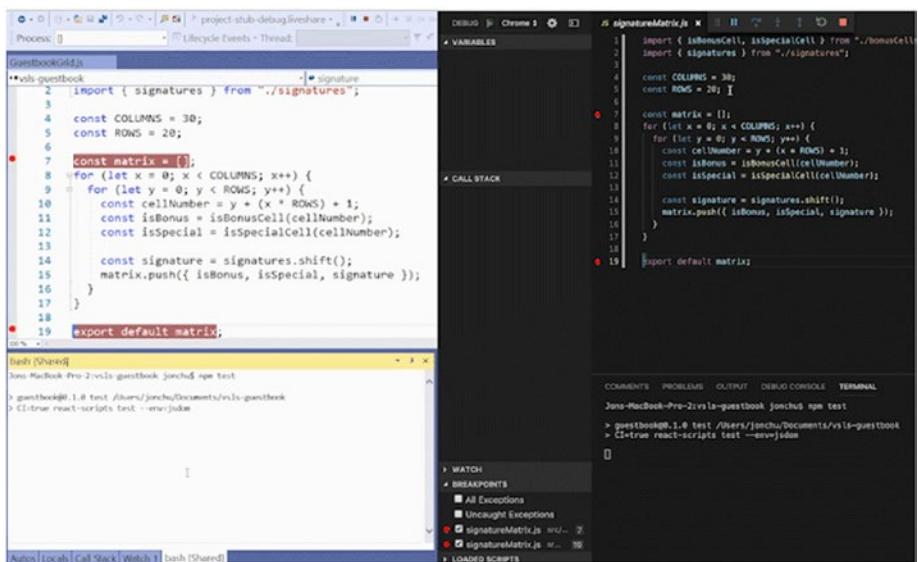
*Рис. 2-14. Gimp 2.10.12*

## Extensions - Расширения Visual Studio

Ниже приведены несколько настоятельно рекомендуемых бесплатных расширений Visual Studio, которые помогут вам расширить возможности Visual Studio/разработки.

### Live Share (2019 и код)

В команде разработчиков, состоящей из нескольких человек, получить оперативную помощь или рецензирование кода может быть непросто. К счастью, бесплатное расширение под названием LiveShare (<https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsls-vs>) предлагает эту функциональность и очень полезно для распределенных команд, где личные встречи затруднены или невозможны. См. Рис. 2-15.



*Рис. 2-15. Совместное использование Visual Studio Live*

## Средство проверки орфографии Visual Studio (2019)

В большом проекте у вас часто будет большое количество строк, переменных и комментариев. Более чем вероятно, что в вашем коде также будут разбросаны орфографические ошибки. Поэтому я настоятельно рекомендую установить расширение проверки правописания Visual Studio (<https://marketplace.visualstudio.com/items?itemName=EWoodruff.VisualStudioCodeSpellCheckerVS2017andLater&ssr=false#overview>).

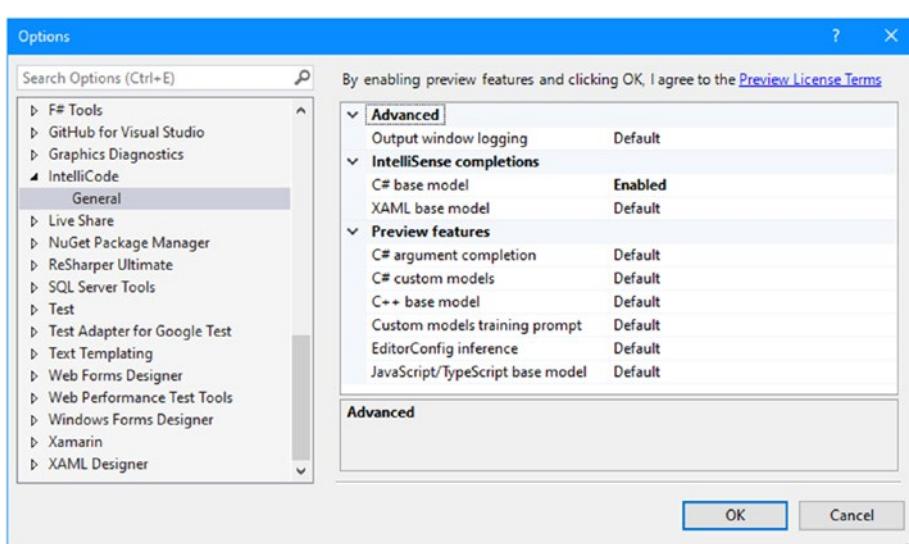
## Visual Studio IntelliCode (2019 and Code)

С появлением машинного обучения, которое стало неотъемлемой частью технологической отрасли, Microsoft предоставила расширение для анализа кода машинного обучения, которое создает пользовательскую модель на основе вашего кода. При работе с распределенной командой или просто в качестве дополнительной проверки согласованности настоятельно рекомендую установить это расширение.

## ГЛАВА 2 НАСТРОЙКА СРЕДЫ РАЗРАБОТКИ

(<https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.VSIntelliCode>.

[visualstudio.com/items?itemName=VisualStudioExptTeam.VSIntelliCode](https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.VSIntelliCode)) и создание модели (всего один клик). См. Рис. 2-16.



**Рис. 2-16.** *Visual Studio IntelliCode*

## Резюме

В этой главе вы узнали, как настроить вашу проект для освоения MonoGame. Кроме того, мы рассмотрели дополнительные инструменты и расширения Visual Studio, которые могут помочь вам в будущих разработках.

Далее следует глубокое погружение в архитектуру MonoGame.

## ГЛАВА 3

# Архитектура MonoGame

После введения MonoGame и настройки вашей среды разработки мы готовы к глубокому погружению в архитектуру MonoGame. Детальное понимание архитектуры поможет вам понять не только будущие главы, но и позволит вам расширять [MonoGame](#) по мере развития ваших навыков.

В этой главе вы узнаете

- О приложении MonoGame Pipeline
- Об игровом классе MonoGame
- Визуализируете свои первые пиксели в MonoGame.

## Архитектура MonoGame

### Конвейерное приложение

Когда речь идет об игре, одними из самых больших компонентов являются такие активы, как текстуры, музыка, спрайты и звуковые эффекты. К счастью, MonoGame делает это очень просто. В основе конвейера ресурсов MonoGame лежит приложение MonoGame Pipeline, инструмент, который берет игровые активы, такие как изображения текстур, звуковые файлы или текстовые шрифты, и преобразует их в двоичные файлы, которые могут быть легко использованы игрой. Кроме того, конвейер ресурсов позволяет разработчику игр легко ссылаться на свои игровые ресурсы в коде.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME

Для тех, кто имел опыт работы с XNA, сборка и компиляция игровых ресурсов была частью процесса сборки и требовала очень много времени сборки почти для каждого проекта. К счастью, MonoGame позволяет вам использовать приложение [assetspipeline](#) для компиляции ресурсов в файл MGCB (MonoGame ContentBinary) независимо от компиляции кода игры в Visual Studio. Таким образом, любое изменение кода не требует пересборки ресурсов.

Рис. 3-1 Иллюстрация приложения MonoGame Pipeline.

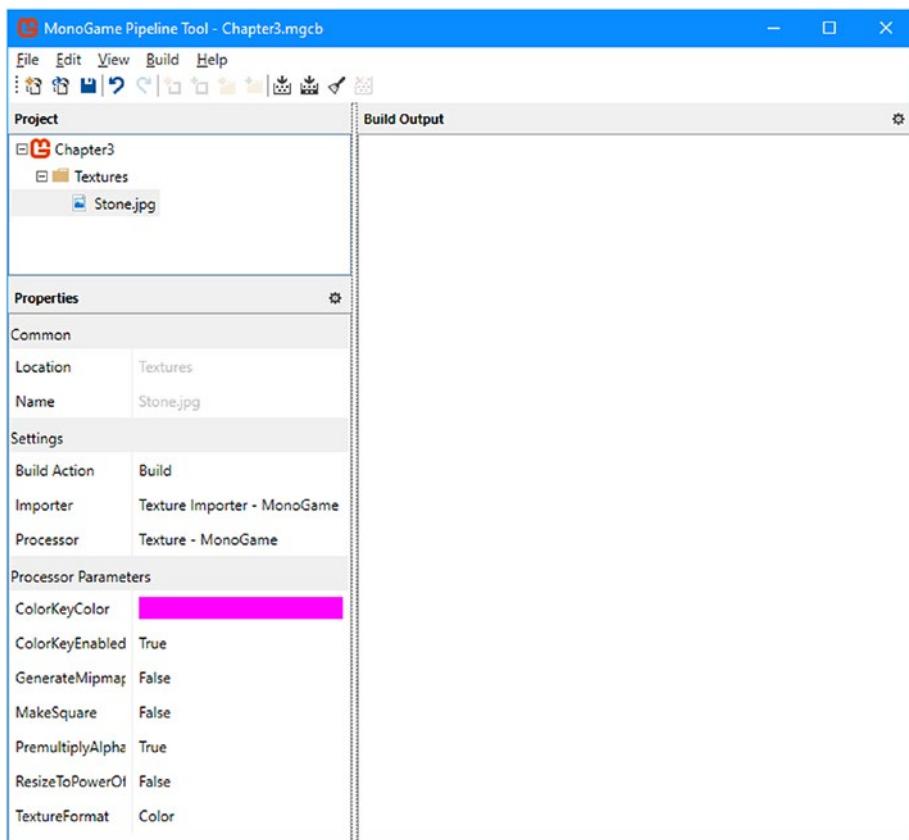


Рис. 3-1. Приложение MonoGame Pipeline

Активы, которые инструмент конвейера поддерживает из коробки

- Эффекты (или шейдеры), представляющие собой небольшие программы, предназначенные для работы на вашей видеокарте и служащие для изменения цвета существующих пикселей на экране.
- 3D models in the .fbx, .X, or Open Asset Import Library formats
- Шрифты, позволяющие разработчику игры рисовать текст на экране
- Видеофайлы в форматах H.264 или .wmv
- Аудиофайлы в форматах .mp3, .ogg, .wav или .wma
- Файлы текстур
- XML-файлы

При добавлении активов в инструмент конвейера необходимо выбрать импортера и обработчика, чтобы сообщить MonoGame, как обрабатывать актив. Эти процессоры и импортеры являются классами в кодовой базе MonoGame, которые знают, как читать содержимое файла, как сериализовать это содержимое в двоичный формат и как преобразовывать это содержимое в структуру данных, которую можно использовать в кодовой базе игры. На [Рис. 3-1](#) анимация в формате PNG добавляется в конвейер ресурсов и настраивается для импорта с использованием средства импорта текстур MonoGame по умолчанию и обработки с использованием процессора текстур MonoGame по умолчанию. Подобный подход позволяет разработчикам позже получить доступ к текстуре в коде, используя что-то вроде этого:

```
contentManager.Load<Texture2D>("Stone");
```

Для типов активов, не поддерживаемых конвейером активов, MonoGame позволяет разработчикам создавать свои собственные импортеры и обработчики.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME

Как правило, для всех ваших ресурсов, не связанных с кодом, вы вручную добавляете файлы в это приложение и перестраиваете свои ресурсы. В случае с текстурами, показанными на рис. 3-1, приложение также будет сжимать, изменять размер и генерировать для вас mip-карты, представляющие собой набор изображений с разным разрешением. В главе 5 более подробно рассматривается, как использовать это приложение для наших целей.

## Класс игры

В основе MonoGame Framework лежит класс Game, который является точкой входа в игру. Его основная функция заключается в настройке окна игры с помощью графического устройства, используемого для рисования на экране, и в настройке важного игрового цикла. Игровой цикл лежит в основе всех видеоигр в дикой природе. По сути, это бесконечный цикл, который постоянно вызывает игровой код, отвечающий за обновление состояния игры, и код, рисующий объекты на экране с помощью методов, называемых [Update\(\)](#) и [Draw\(\)](#).

Игровой цикл должен быть быстрым и эффективным. Подобно мультфильму, где кадры анимации должны отображаться со скоростью 24 кадра в секунду, чтобы создать иллюзию движения, видеоигра должна выглядеть плавно и быстро реагировать на команды игрока. Большинство игр пытаются достичь 60 кадров в секунду, что означает, что каждый вызов как [Update\(\)](#), так и [Draw\(\)](#) в рамках одной итерации игрового цикла должен занимать не более 1/60 секунды. Когда все происходит быстрее, MonoGame делает паузу, чтобы убедиться, что на выполнение цикла ушло ровно 1/60 секунды, и обеспечить согласованный вывод на экран. Когда все происходит медленнее, MonoGame попытается пропустить метод [Draw\(\)](#) несколько раз, чтобы позволить обновлениям наверстать упущенное, что может привести к небольшому заиканию игры. Он также может автоматически снижаться до более низкой частоты кадров.

Чтобы создать игру, мы должны создать собственный игровой класс, который будет наследоваться от класса Game в MonoGame.

Класс Game предоставляет чрезвычайно простой интерфейс для MonoGame Framework с помощью четырех основных функций, как показано на рис. 3-2.

```
public class Game : IDisposable {  
    protected virtual void Initialize();  
    protected virtual void LoadContent();  
    protected virtual void Update(GameTime time);  
    protected virtual void Draw(GameTime time);  
}
```

*Рис. 3-2. Основные методы игрового класса*

Метод [Initialize\(\)](#) предоставляет точку входа для инициализации фоновых потоков, графического устройства и других диспетчеров, не связанных с содержимым. Метод [LoadContent\(\)](#) предоставляет точку входа для создания [SpriteBatch](#) — объекта, который мы будем использовать для отрисовки игровых объектов на экране. Он также используется для загрузки содержимого из вышеупомянутого файла [MGCB](#).

Как мы обсуждали ранее в кратком введении к игровому циклу, методы [Update\(\)](#) и [Draw\(\)](#) последовательно вызываются внутри игрового цикла. [Update\(\)](#) предоставляет точку входа для обработки игровых данных, физики и других не связанных с графикой обновлений игрового таймера. С другой стороны, метод [Draw\(\)](#) предоставляет точку входа для обработки всего графического рендеринга.

В ходе работы над этой книгой мы будем добавлять классы-менеджеры для динамического и мощного управления этими сценариями, чтобы создать игровой проект книги и позволить вам расширять ваши знания для ваших собственных творений.

## Ваши первые визуализированные пиксели

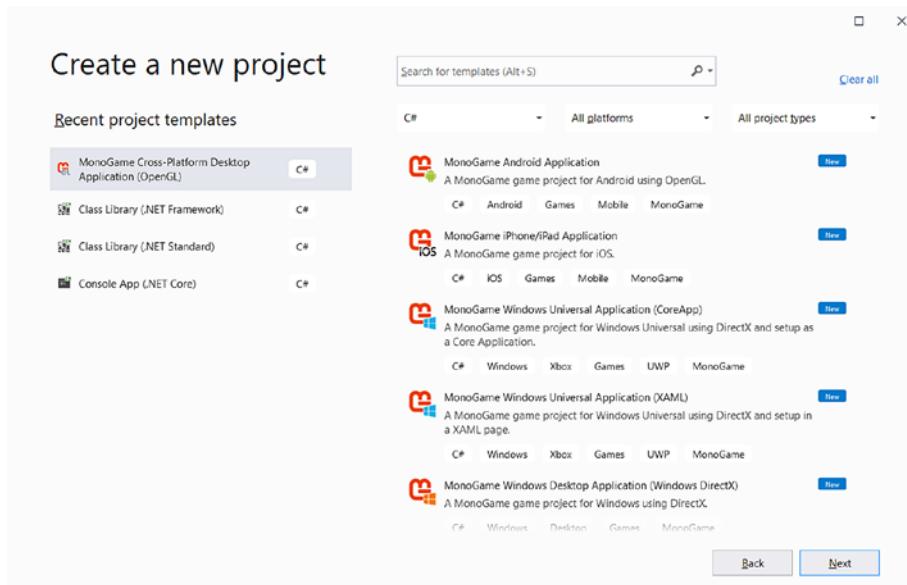
Теперь, когда у вас есть общее представление об архитектуре MonoGame, наконец-то пришло время создать свой первый проект MonoGame в Visual Studio. Те, кому нужен предварительно настроенный проект, могут просмотреть папку [Chapter3](#) для решения и проекта Visual Studio и перейти к подразделу «Diving into the Project - Погружение в проект».

## Создание решения и проекта

Для этого, шаг за шагом, я буду использовать Visual Studio 2019 в [Windows asconFig.d](#) в главе 2. Если вы не установили и не установили Visual Studio или MonoGame, вернитесь к главе 2, а затем вернитесь к этому разделу.

Сначала запустите Visual Studio, и вы увидите диалоговое окно «[Create a new project](#) - Создать новый проект», как показано на Рис. 3-3. Если ваши настройки не позволяют открыть диалоговое окно «Создать новый проект» при запуске, просто нажмите «File▶New▶Project - Файл»▶ «Создать»▶ «Проект».

## ГЛАВА 3

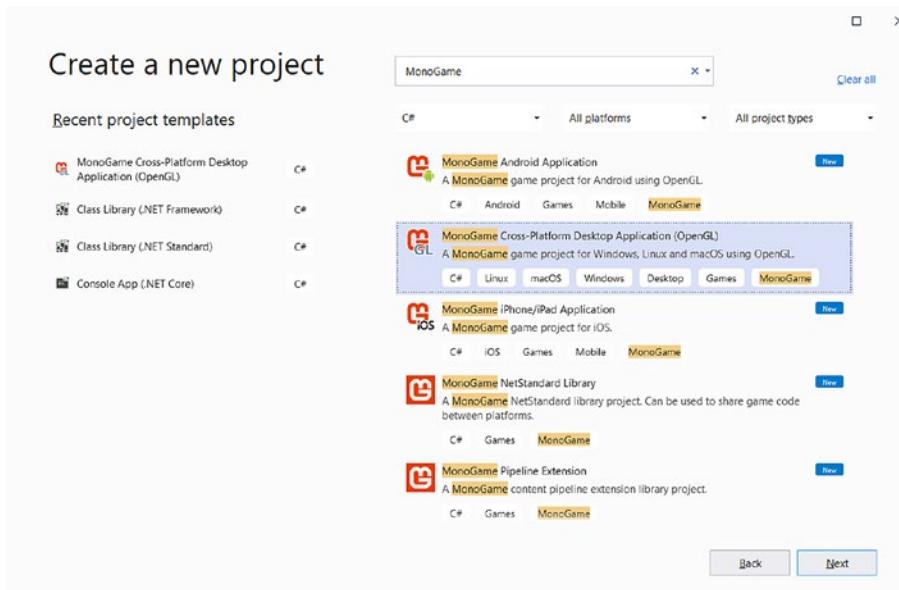


*Рис. 3-3. Visual Studio 2019 Диалог создания нового проекта*

В строке поиска вверху введите «MonoGame». Если Visual Studio не возвращает никаких результатов, убедитесь, что вы правильно установили MonoGame, как описано в Главе 2. В противном случае вы увидите множество различных вариантов. MonoGame — это кроссплатформенная библиотека, которую можно использовать для создания видеоигр на телефонах, планшетах, в Linux или MacOS, в Windows и на консолях, таких как Nintendo Switch, Xbox или PlayStation. Но поскольку для некоторых из этих вариантов требуется определенный код для конкретной платформы, вы должны выбрать проект, который соответствует вашим целям.

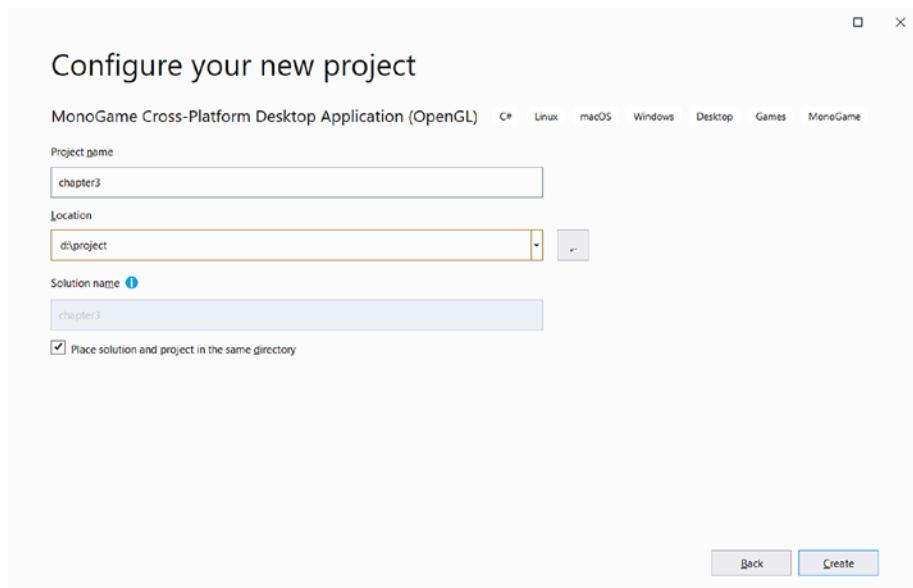
Мы будем создавать игру для ПК, поэтому у нас есть два классических варианта: игра на основе DirectX или OpenGL. Хотя DirectX будет работать в Windows и Xbox, он не будет работать в системах Linux или MacOS. Однако OpenGL — это графическая библиотека, которая долгое время была кроссплатформенной. Выберите «MonoGame Cross-Platform Desktop Application (OpenGL - Кроссплатформенное настольное приложение MonoGame (OpenGL))» и нажмите «Next - Далее», как показано на Рис. 3-4.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME



*Рис. 3-4. Visual Studio 2019 Создание диалогового окна нового проекта с выбранным шаблоном кроссплатформенного настольного приложения (OpenGL) MonoGame*

В следующем диалоговом окне не стесняйтесь давать проекту любое имя по вашему желанию; так как это Глава 3, я дал ей название Chapter3, как показано на Рис. 3-5.



*Рис. 3-5. Visual Studio 2019 Диалоговое окно настройки нового проекта*

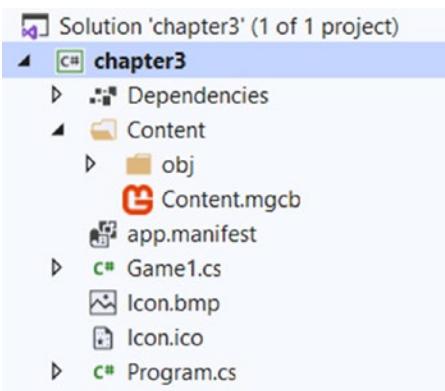
После ввода имени проекта нажмите «Create - Создать». Затем вам будет представлен пустой холст для MonoGame.

## Погружение в проект

Теперь, когда у нас есть проект и решение, давайте просмотрим все файлы, входящие в базовый шаблон MonoGame. Точные файлы могут различаться в зависимости от платформы и будущей версии, но я лично не видел больших различий между версиями на протяжении многих лет.

Начнем с обозревателя решений (Ctrl+W,S в Windows, если в вашей конфигурации по умолчанию он закрыт). Вы должны увидеть что-то очень похожее на рис.3-6.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME

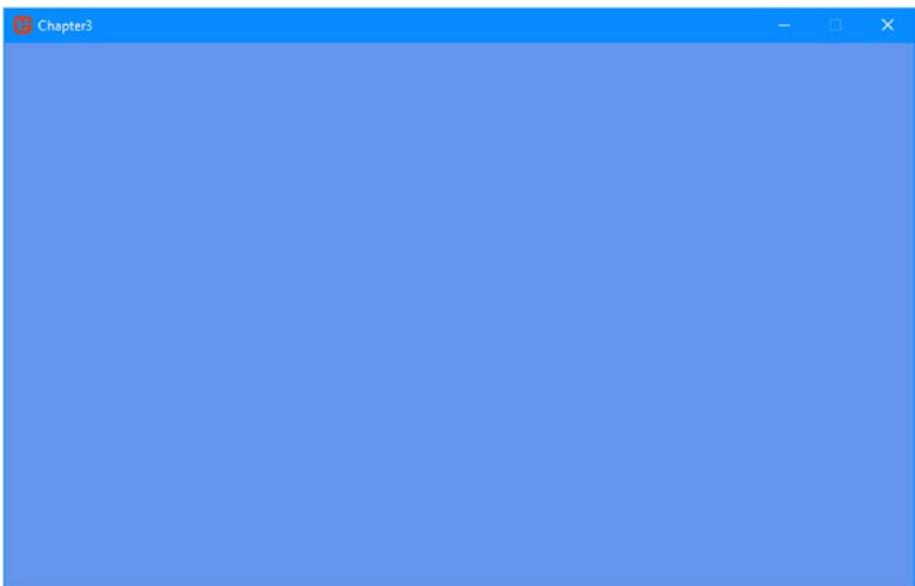


*Рис. 3-6. Visual Studio 2019 Solution Explorer MonoGame for Desktop Template*

Начиная с верхней части списка файлов решений:

- **Content.mgcb**: файл, содержащий все ваши активы, такие как текстуры, музыка и звуковые эффекты, среди прочего. В шаблоне и для этой главы он останется пустым.
- **Game1.cs**: реализация расширенного класса Game по умолчанию, которая будет обновлена позже в этой главе
- **Icon.ico**: значок проекта (по умолчанию значок MonoGame)
- **Program.cs**: содержит метод **Main** программы и вызов класса **Game1**

Обновим проект и переименуем файл Game1.cs. Щелкните файл правой кнопкой мыши и выберите параметр «**Rename - Переименовать**». Переименуйте файл в **MainGame.cs**. Затем соберите и запустите проект; вы должны увидеть всплывающее окно, как на рис. 3-7.



*Рис. 3-7. Глава 3 пример*

## Погружение в MainGame.cs

Откройте файл MainGame.cs. Начиная с верхней части файла, вы можете заметить использование пространств имен Microsoft.XNA по сравнению с MonoGame:

```
using Microsoft.Xna.Framework;  
  
using Microsoft.Xna.Framework.Graphics;using  
Microsoft.Xna.Framework.Input;
```

Причина сохранения прежнего имени XNA заключается в том, что обилие документации, примеров и существующего кода, который необходимо будет обновить, станет огромной задачей для проекта с открытым исходным кодом. Обсуждалось возможное изменение его для будущей основной версии, но ничего на момент написания этой статьи не было введено в действие.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME

Далее в исходном файле идет объявление двух приватных переменных:

```
GraphicsDeviceManager graphics;SpriteBatch spriteBatch;
```

Класс GraphicsDeviceManager предоставляет интерфейс MonoGame для графической карты. Для тех, кто работал с DirectX 12 или Vulkan, это похоже на перечисление устройств и наличие одного объекта класса для запроса команд. Кроме того, класс также предоставляет информацию об устройстве, представленную в свойстве [GraphicsDevice.Adapter](#).

Графическое устройство — это низкоуровневый модуль, отвечающий за отрисовку графических объектов и вывод их на экран. В следующей главе мы углубимся в некоторые другие свойства [GraphicsDeviceManager](#), такие как запрос разрешения видео, полноэкранный режим или нет, а также мультиплатформенность.

Класс SpriteBatch предоставляет основной интерфейс чрезвычайно мощному движку 2D-рендеринга, представленному в MonoGame Framework. Хотя название подразумевает, что он используется только для спрайтов, на самом деле он предоставляет интерфейс рендера для всего 2D-рендеринга.

Далее в файле идет вызов конструктора для [Game1](#). Измените имя класса на [MainGame](#), чтобы оно совпадало с именем файла:

```
public MainGame(){
    graphics = new GraphicsDeviceManager(this);Content.RootDirectory
    = "Content";
}
```

В строке 3 инициализируется графическая переменная, а в строке 4 RootDirectory устанавливается в Content. Вернувшись к рис. 3-6, вы заметите, что файл Content.mgcb находится в подпапке Content. Если вы переименуете эту папку, обязательно обновите эту строку. Эта книга и все примеры сохранят это название.

## ГЛАВА 3

Метод `Initialize` изначально пуст и просто вызывает метод `Initialize` своего базового класса. Давайте посмотрим на `LoadContent`:

```
protected override void LoadContent(){
    //Создаем новый SpriteBatch, который можно использовать
    //для рисования текстур.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: загрузите сюда игровой контент}
```

В строке 4 инициализируется `spriteBatch`. Традиционно этот метод `LoadContent` также дает нам возможность загружать ресурсы из инструмента конвейера, что мы и сделаем в главе 5.

Переходя к следующему методу, `Update`:

```
protected override void Update(GameTime gameTime){
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
        ButtonState.Pressed || Keyboard.GetState().IsKeyDown
        (Keys.Escape))
        Exit();

    // TODO: добавьте сюда свою логику обновления

    base.Update(gameTime);}
```

Строка 3 в предыдущем блоке кода, как вы, вероятно, догадались, проверяет, была ли нажата кнопка `Escape`, и, если да, выход из игры.

## ГЛАВА 3 АРХИТЕКТУРА MONOGAME

Комментарий TODO в строке 6 обычно расширяется, чтобы включить в него вызовы для обнаружения столкновений, AI, физики и других компонентов, не связанных с графикой, в видеоиграх. Здесь поддерживается состояние игры, обновляются координаты игровых объектов в игровом мире, обновляются скорости движения вражеских игровых объектов на основе некоторой переменной ускорения или происходит проверка смерти игрока. Здесь может случиться многое, но мы обязательно структурируем наш код, чтобы этот метод не был перегружен слишком большим количеством деталей и обязанностей.

В строке 8 вызывается метод Update класса Game, который мы сохраним для целей этой книги, поскольку для простоты будем использовать класс Game.

И последнее, но не менее важное — это метод Draw:

```
protected override void Draw(GameTime gameTime){  
    GraphicsDevice.Clear(Color.CornflowerBlue);  
    // TODO: добавьте сюда свой код рисования  
    base.Draw(gameTime);}
```

Вызов метода `GraphicsDevice.Clear` в строке 3 в предыдущем блоке кода очищает экран для `CornflowerBlue`. Очистка экрана перед рендерингом новых объектов важна для предотвращения того, чтобы ранее нарисованные артефакты оставались на экране и чтобы объекты не создавали следов при их перемещении. На протяжении большей части книги мы будем очищать черный цвет, но для демонстрации фактического рендеринга шаблон по умолчанию очищается до василькового синего.

В комментарии TODO в строке 5 ранее мы будем вызывать наш единственный вызов для визуализации всех объектов в следующей главе.

## ГЛАВА 3

Наконец, вызов `base.Draw` в строке 7 вызывает базовый метод `Draw` класса `Game`, что важно для программистов MonoGame, которые хотят использовать игровые компоненты в своем коде. Хотя для нас в этой книге этого не будет, мы рекомендуем оставить базовый вызов на месте на всякий случай.

## Порядок выполнения

Общий вопрос, который я часто задаю себе при использовании нового фреймворка, особенно при расширении или первом глубоком погружении, заключается в том, в каком порядке следует вызывать методы?

Используя проект в качестве ссылки, вот порядок:

1. Program Class ➤ Creates Game object ➤ Calls Runon the Game object
2. MainGame Constructor
3. MainGame.Initialize
4. MainGame.LoadContent
5. Game Loop ➤ (MainGame.Update and thenMainGame.Draw)
6. MainGame.UnloadContent
7. MainGame.Finalizer

Знание этого порядка, получение нулевых исключений, проблемы с содержимым или другие аномалии рендеринга могут быть связаны с выполнением операций не по порядку.

## Резюме

В этой главе вы узнали об архитектуре MonoGame, погрузились в шаблон по умолчанию и настроили свой первый проект. Кроме того, вы запускали и визуализировали свои первые пиксели с помощью MonoGame Framework!

Следующим шагом будет планирование архитектуры остальной части книги с помощью MonoGame.

## ГЛАВА 4

# Планирование игрового движка

Building upon the last chapter, in this chapter we will start architecting the game engine. For each chapter, we will implement another component until completion. Proper planning and architecting are crucial to creating a successful engine, game (or any program for that matter). This chapter will also go over a couple design patterns used in game engines in case you want to explore other patterns in your own projects.

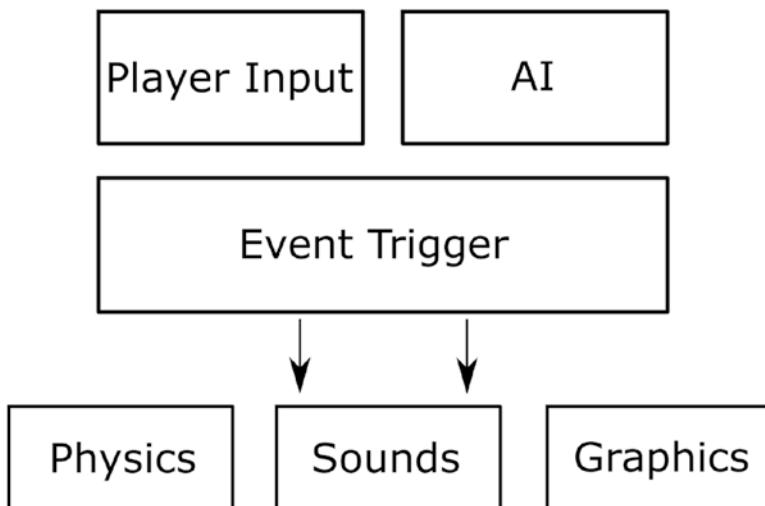
In this chapter, you will learn

- Game engine design patterns
- Programming design patterns
- State management
- MonoGame architecture

## Дизайн игрового движка

Игровые движки используются повсеместно. От простых инди-игр до игр AAA с многомиллионными бюджетами — каждый раз, когда разработчики хотят повторно использовать общий код игры, создается движок. Самые сложные движки представляют собой очень сложные фрагменты кода, на выпуск которых у команд уходят десятки (или более) месяцев, а в некоторых случаях и годы. Unreal Engine 3 содержит более двух миллионов строк кода для справки. С другой стороны, мы могли бы разработать очень маленький и простой движок, который просто рисует для нас игровые объекты и фиксирует действия

игрока, не делая больше ничего. В этой главе мы сначала погрузимся в основные компоненты игрового движка, который будем писать с использованием MonoGameFramework. На рис. 4-1 показана общая архитектура движка.



*Рис. 4-1. Архитектура игрового движка*

## Ввод игрока

Управление многими (если не большинством) взаимодействием вашего игрового движка является вводом. В зависимости от вашей целевой платформы это может включать в себя все: от стандартной комбинации геймпада, клавиатуры и мыши до отслеживания касаний или головы с помощью гарнитуры виртуальной реальности. В главе 6 мы глубоко погрузимся в интеграцию универсального интерфейса, чтобы изящно обрабатывать различные входные данные, которые существуют сегодня для игр, и насколько это возможно для будущего.

## Искусственный интеллект (ИИ)

Искусственный интеллект был важнейшим компонентом игр на протяжении десятилетий. Одним из самых ранних примеров является игра [Space Invaders](#) конца 1970-х годов. Несмотря на примитивность по сегодняшним меркам, [Space Invaders](#) предлагала игроку бросить вызов управляемым компьютером игрокам с двумя разными типами врагов. В современных играх поиск пути и деревья решений движут большинством игр.

## Триггеры событий

В основе нашего движка и многих других лежит система триггеров событий. Идея, стоящая за этим, состоит в том, чтобы определить общее событие, такое как игрок, нажимающий левую кнопку мыши. Фактическая игра будет реагировать на это событие и выполнять одно или несколько действий. Преимущество здесь в том, чтобы свести сложность к минимуму. Более традиционный подход к программированию здесь состоял бы в том, чтобы иметь определенные вызовы Render the Player, но затем, когда игрок нажимал правую кнопку, имел очень похожий код в другом методе Render the Player. Этот подход, как вы можете видеть, также создает DRY - СУХИЕ (не повторяйте себя) нарушения. Позже в этой главе мы создадим основу для нашей подсистемы триггеров событий, которую мы будем использовать в последующих главах.

## Графический рендеринг

Одним из наиболее важных компонентов игрового движка является графика. Рендеринг графики в большинстве современных игровых движков включает в себя спрайты, 3D-модели, частицы и различные проходы текстурирования, и это лишь некоторые из них. К счастью, MonoGame предоставляет простые в использовании интерфейсы, и в этой книге мы сосредоточимся только на 2D-рендеринге. В течение оставшихся глав мы будем расширять возможности рендеринга нашего движка. Кроме того, мы подробно рассмотрим добавление подсистемы частиц в главе 8.

## Рендеринг звука

Часто упускаемый из виду, звуковой рендеринг, возможно, не менее важен для предоставления вашей аудитории высококачественного слухового опыта. Представьте себе, что вы смотрите свой любимый боевик без звука или музыки — в нем отсутствует половина впечатлений. К счастью, в MonoGame очень легко добавить даже базовый уровень к нашему игровому движку, чтобы обеспечить и музыку, и звук. Те, кто занимался разработкой XNA в прошлом, MonoGame переработали интерфейс и не требуют использования XACT (Cross-Platform AudioCreation Tool). На высоком уровне MonoGame предоставляет простой класс Song, который вы, вероятно, предположили для музыки, и SoundEffect для звуковых эффектов. Мы углубимся в звук с MonoGame в главе 7, добавив музыку и звуковые эффекты в наш движок.

## Физика

В зависимости от игры физика может оказаться более важным компонентом, чем звук, ввод или даже графика. Существует растущий жанр игр, в которых основное внимание уделяется физике с относительно простой графикой, таких как Cut the Rope 2 или Angry Birds 2, где птицы бросают из рогатки к ненадежно сбалансированным конструкциям, которые рушатся на землю, когда птица врезается в их основание. Подобно звуковым и графическим триггерам, физические триггеры могут вызвать дополнительные события, такие как столкновение главного героя с врагом, что, в свою очередь, приведет к уничтожению анимации, здоровья и, возможно, врага.

## Управление состоянием

Управление состоянием является распространенным шаблоном, применяемым в играх и, в частности, в MonoGame из-за простой конструкции, которую он предлагает. Идея управления состоянием заключается в том, что независимо от сложности видеигры каждый экран, например стартовое меню, появляющееся при запуске игры, или экран, на котором отображается игровой процесс, можно разбить на собственное уникальное состояние.

Возьмем, к примеру, разные состояния традиционной игры:

- Заставка
- Главное меню
- Геймплей
- Резюме конца уровня

Каждое из этих состояний часто предлагает разные схемы ввода, музыку и звуковые эффекты, не говоря уже о разном отображении ресурсов.

Например, экран-заставка обычно состоит из

- Полнэкранное масштабированное изображение или анимированное видео
- Музыка
- Переходы на основе времени или продвижение на основе ввода
- Диспетчер ввода, который ждет, пока пользователь запустит игру, нажав какую-либо клавишу на своем устройстве ввода.

С другой стороны, состояние игрового процесса будет включать физику, частицы и агентов ИИ, используемых для управления врагами. Он также имеет гораздо более сложный менеджер ввода, точно фиксирующий движения и действия игрока. Состояние игрового процесса также может отвечать за синхронизацию состояния игры по сети, если игрок играет с друзьями в Интернете. Все это говорит о том, что разбиение игры на группы похожих состояний поможет вам начать строить свою игру.

Подобно проектированию на основе наследования, правильное группирование схожих функций и расширение только при необходимости сократит время на поддержку вашего проекта и усилия по разработке.

Чтобы еще больше проиллюстрировать это, давайте рассмотрим несколько состояний [Stardew Valley](#) на базе MonoGame на рисунках 4-2, 4-3 и 4-4.



Рис. 4-2. Главное меню Stardew Valley



Рис. 4-3. Меню Stardew Valley



*Рис. 4-4. Геймплей Stardew Valley*

Начиная с рис. 4-2, состояние главного меню Stardew Valley состоит из

- Многослойные анимированные спрайты (некоторые выровнены)
- Интерактивные кнопки
- Фоновая музыка

Состояние создания персонажа в Рис.4-3 состоит из тех же элементов с добавлением полей ввода и более сложных элементов позиционирования, что позволяет игроку создать нового персонажа с желаемой внешностью.

Наконец, на рис.4-4 показан основной экран игрового процесса и он содержит множество компонентов первых двух состояний, но увеличивает сложность графического рендеринга, добавляя игровые объекты, которые могут меняться со временем, и позволяя игроку перемещаться по игровому миру.

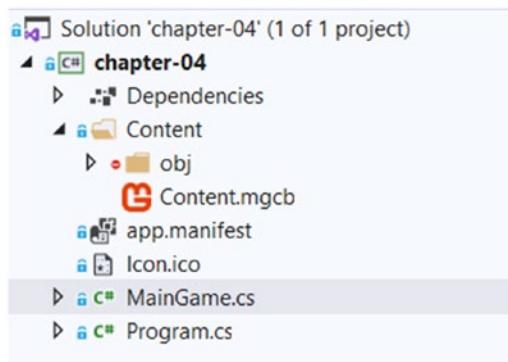
# Реализация архитектуры движка

Теперь, когда мы рассмотрели каждый из компонентов современного игрового движка, пришло время приступить к разработке нашего движка.

Для тех, кто хочет загрузить завершенное решение, см. папку главы 4, где можно найти как пустой проект в начальной папке, так и завершенный проект в конечной папке.

## Создание проекта

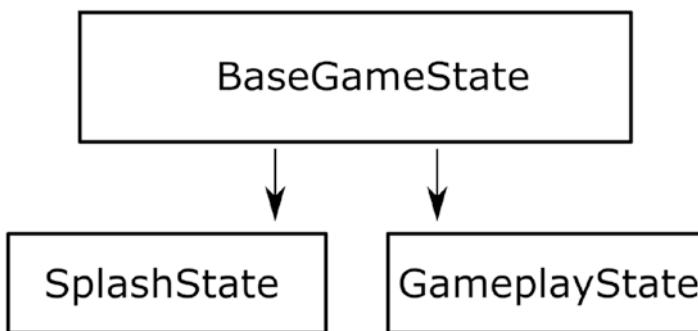
Следуя тем же шагам, которые мы рассмотрели в главе 3, мы создадим тот же тип проекта для этой главы. В дальнейшем имейте в виду, что проект этой главы станет основой для всех остальных глав книги. Как и в предыдущей главе, создайте новый проект кроссплатформенного настольного приложения (OpenGL) MonoGame и переименуйте файл Game1.cs и класс Game1 в MainGame.cs и MainGame - Основная игра. После этого вы должны увидеть проект, подобный показанному на рис.4-5.



*Рис. 4-5. Visual Studio 2019, показывающий пустой проект Chapter4*

## Создание классов состояний

Как было рассмотрено ранее в этой главе, когда мы говорили об управлении состояниями, основной идеологией управления состояниями является модель наследования для создания структуры и сокращения объема повторного использования кода для каждого состояния вашей игры. В рамках этой главы мы создадим класс `initialBaseGameState`, за которым следуют классы `emptySplashState` и `emptyGameplayState`, которые будут заполнены в следующих главах. Рис. 4–6 иллюстрирует взаимосвязь между этими состояниями.



*Рис. 4-6. Состояния игры, которые необходимо реализовать*

В следующих текстах вы найдете начальный код для нашего класса `abstractBaseGameState`, который мы будем использовать в этой книге. Откройте конечное решение главы 4 и посмотрите на класс `BaseGameState.cs` в файле `States\Base\BaseGameState.cs`:

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

using chapter_04.Objects.Base;
```

## ГЛАВА 4 PLANNING YOUR GAME ENGINE

```
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace chapter_04.States.Base
{
    public abstract class BaseGameState
    {
        private readonly List<BaseGameObject> _gameObjects =
            new List<BaseGameObject>();

        public abstract void LoadContent(ContentManager
            contentManager);

        public abstract void UnloadContent(ContentManager
            contentManager);

        public abstract void HandleInput();

        public event EventHandler<BaseGameState> OnStateSwitched;

        protected void SwitchState(BaseGameState gameState)
        {
            OnStateSwitched?.Invoke(this, gameState);
        }

        protected void AddGameObject(BaseGameObject gameObject)
        {
            _gameObjects.Add(gameObject);
        }
    }
}
```

```

public void Render(SpriteBatch spriteBatch)
{
    foreach (var gameObject in _gameObjects.OrderBy
        (a => a.zIndex))
    {
        gameObject.Render(spriteBatch);
    }
}
}

```

Начнем с объявлений абстрактных методов [LoadContent](#) и [UnloadContent](#). Эти методы предоставляют интерфейс для, как вы, вероятно, догадались, загрузки и выгрузки контента. MonoGame использует объект класса [ContentManager](#) для предоставления простого в использовании интерфейса для загрузки содержимого во время выполнения. Мы подробно рассмотрим это в следующей главе, когда углубимся в управление активами. На данный момент имейте в виду, что эти методы будут обрабатывать выгрузку и загрузку содержимого в зависимости от состояния.

Другой абстрактный метод, [HandleInput](#), предоставляет метод для обработки ввода в зависимости от состояния. В этой главе мы сохраним наши реализации простыми. В главе 6, как упоминалось ранее, мы углубимся в абстрагирование обработки ввода.

Событие [OnStateSwitched](#) и метод [SwitchState](#) предоставляют как метод для переключения состояния из другого состояния, так и событие для прослушивания основным классом. Любой класс состояния, реализующий этот класс [BaseGameState](#), сможет вызвать метод [SwitchState](#) и передать новое состояние, на которое мы хотим переключиться. Например, нажатие клавиши Enter в состоянии [SplashScreen](#) вызовет [SwitchState](#) и укажет, что мы хотим теперь использовать состояние [Gameplay](#). Метод [Switch State](#) инициирует событие, на которое наш класс [MainGame](#) будет реагировать, выгружая текущее состояние и затем загружая новое состояние. На следующей итерации игрового цикла начнут вызываться методы [Update](#) и [Draw](#) нового состояния.

Метод `AddGameObject` — это метод состояния для добавления объектов в коллекцию `List BaseGameObjects`, которая используется для отслеживания игровых объектов, которые мы хотим отобразить на экране. В следующих главах мы будем использовать этот метод для добавления спрайтов, статических изображений и других объектов в этот список.

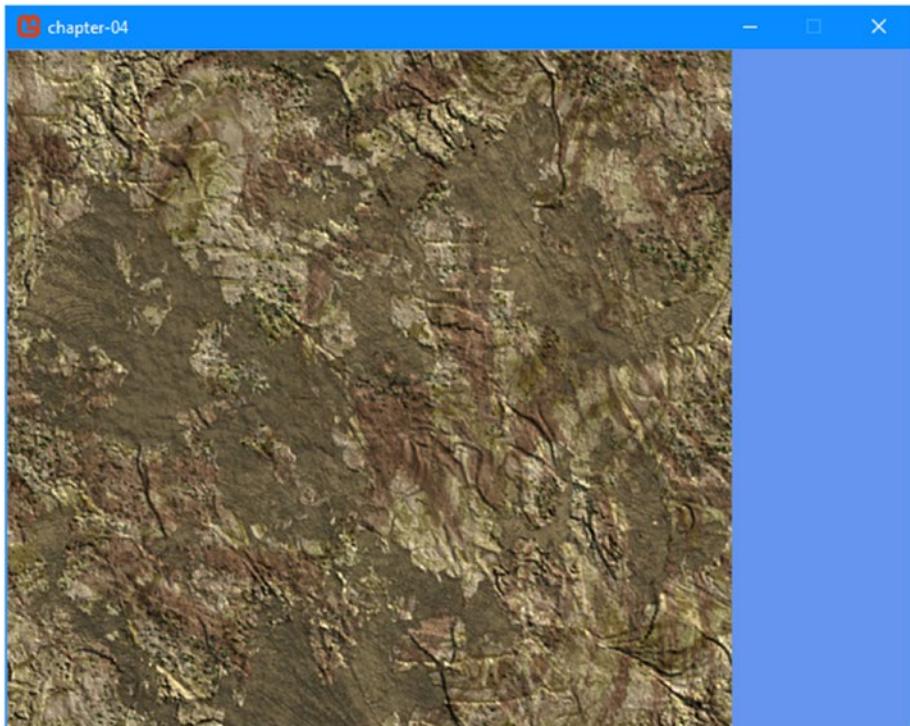
Наконец, метод `Render` предоставляет единый метод для перебора всех игровых объектов, которые мы хотим отобразить на экране. Этот метод вызывается из основного метода `Draw` в классе `MainGame`. Он берет все игровые объекты из другого списка `_gameObjects` и упорядочивает их по `zIndex` перед их отрисовкой. `ZIndex` — это метод упорядочения игровых объектов от самого дальнего к ближайшему. Когда `MonoGame` рисует объекты на экране, каждый нарисованный объект перезаписывает объекты, которые были нарисованы до него. Хотя это желательно в тех случаях, когда объекты, расположенные ближе к зрителю, должны скрывать объекты, находящиеся дальше, мы не хотим делать обратное. Например, облака нужно рисовать перед солнцем, а не позади него. Поэтому, когда мы создаем игровые объекты, мы должны рисовать их по порядку, и для этого мы используем `zIndex`. Почему «з»? Потому что в 2D-играх мы используем систему координат (X, Y), где ось X горизонтальна, а ось Y вертикальна. В трехмерном пространстве есть третья ось, называемая Z, поэтому мы, по сути, представляем глубину с помощью `zIndex`. Обратите внимание: если каждый игровой объект имеет `atZIndex = 0`, то наш класс базового состояния не может гарантировать, что все будет отрисовано в правильном порядке.

## Создание масштабирования и управления окнами

Теперь, когда мы рассмотрели наш базовый начальный код управления состоянием, прежде чем фактически отобразить что-либо на экране, нам нужно обработать масштабирование и поддерживать как оконный, так и полноэкранный режимы.

## Масштабирование окна

Идея масштабирования окна заключается в том, чтобы ваша аудитория наслаждалась игрой так, как вы задумали, независимо от разрешения. Возьмите рис.4-5. Окно в настоящее время имеет ширину 640 и высоту 480 пикселей, а ширина и высота текстуры - 512 пикселей. Учитывая эти размеры и как показано на Рис. 4-7, оно занимает почти весь экран.



*Рис. 4-7. Немасштабированное окно 640x480 с текстурой 512x512*

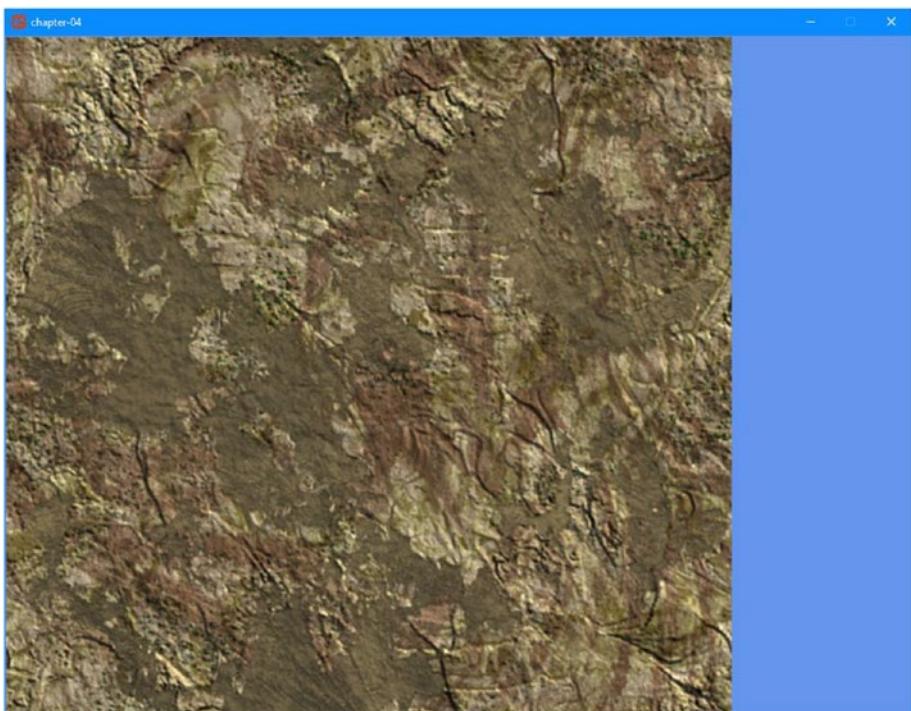
Общим для игр последних двух десятилетий является выбор разрешения, поэтому давайте повторим тот же самый рендеринг с разрешением 1024x768. На рис.4-8 это показано.



**Рис. 4-8.** Рис. 4-8. Немасштабированное окно 1024x768 с текстурой 512x512

Как ясно видно, визуальный опыт для пользователей с более высоким разрешением вашей игры существенно отличается. К счастью, MonoGame предлагает очень простой способ устранить это несоответствие опыта. Подход предполагает, что вы проектируете целевое разрешение, такое как 1080р (1920x1080), если вы ориентируетесь на ПК или домашние консоли. Как только решение принято, все ваши активы должны быть созданы с учетом этого решения. Такие изображения, как всплеск или фоновые изображения, должны иметь это разрешение или выше. Создание активов и управление ими будут более подробно рассмотрены в следующей главе; однако соблюдение этого простого правила поможет вам, когда вы начнете создавать свой контент.

После определения целевого разрешения мы добавим простую шкалу ширины и высоты относительно целевого разрешения. Например, в двух примерах мы используем 640x480 в качестве целевого разрешения и оставляем пользовательское разрешение равным 1024x768. После реализации нашего масштабатора, см. [Рис. 4-9](#). Обратите внимание, что за исключением того, что он больше (как и ожидалось), опыт идентичен снимку экрана 640x480 на [Рис. 4-9](#).



*Рис. 4-9. Масштабированное окно 1024x768 с текстурой 512x512*

Теперь давайте погрузимся в код, который привел к этому изменению. Во-первых, нам нужно определить несколько новых переменных в нашем классе MainGame:

```
private RenderTarget2D _renderTarget;  
private Rectangle _renderScaleRectangle;
```

## ГЛАВА 4 PLANNING YOUR GAME ENGINE

```
private const int DESIGNED_RESOLUTION_WIDTH = 640;
private const int DESIGNED_RESOLUTION_HEIGHT = 480;

private const float DESIGNED_RESOLUTION_ASPECT_RATIO =
DESIGNED_RESOLUTION_WIDTH / (float)DESIGNED_RESOLUTION_HEIGHT;
```

RenderTarget2D будет содержать заданное целевое разрешение, а переменная `_renderScaleRectangle` будет содержать прямоугольник масштаба. Переменные `DESIGNED*` содержат значение, предназначенное для разрешения; не стесняйтесь экспериментировать с этими значениями после добавления этого кода.

После определения новых переменных нам нужно будет инициализировать переменные `RenderTarget` и `Rectangle`, которые будут использоваться в нашем цикле рендеринга, в методе `Initialize`, который мы определили ранее. Кроме того, нам нужно определить новый метод для создания прямоугольника в следующем коде:

```
protected override void Initialize()
{
    _renderTarget = new RenderTarget2D(graphics.GraphicsDevice,
        DESIGNED_RESOLUTION_WIDTH, DESIGNED_RESOLUTION_HEIGHT,
        false,
        SurfaceFormat.Color, DepthFormat.None, 0,
        RenderTargetUsage.DiscardContents);

    _renderScaleRectangle = GetScaleRectangle();

    base.Initialize();
}

private Rectangle GetScaleRectangle()
{
    var variance = 0.5;
    var actualAspectRatio = Window.ClientBounds.Width / (float)
        Window.ClientBounds.Height;

    Rectangle scaleRectangle;
```

```
if (actualAspectRatio <= DESIGNED_RESOLUTION_ASPECT_RATIO)
{
    var presentHeight = (int)(Window.ClientBounds.Width /
DESIGNED_RESOLUTION_ASPECT_RATIO + variance);
    var barHeight = (Window.ClientBounds.Height -
presentHeight) / 2;

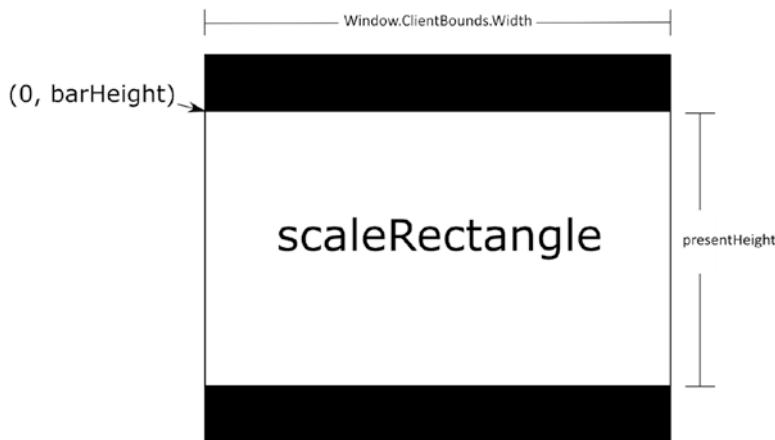
    scaleRectangle = new Rectangle(0, barHeight, Window.
ClientBounds.Width, presentHeight);
}
else
{
    var presentWidth = (int)(Window.ClientBounds.Height *
DESIGNED_RESOLUTION_ASPECT_RATIO + variance);
    var barWidth = (Window.ClientBounds.Width -
presentWidth) / 2;

    scaleRectangle = new Rectangle(barWidth, 0,
presentWidth, Window.ClientBounds.Height);
}

return scaleRectangle;
}
```

[GetScaleRectangle](#) предоставляет черные полосы, похожие на скейлеры на экране вашего телевизора, в зависимости от фактического разрешения и проектного разрешения. Если изображение, отображаемое на экране, не соответствует размеру фактического экрана, телевизор добавит черные полосы по горизонтали или по вертикали, чтобы заполнить недостающее пространство. Этот метод начинается с вычисления соотношения ширины и высоты игрового окна. Если это соотношение ниже расчетного соотношения сторон, которое является нашим желаемым соотношением, тогда нам нужно добавить черные полосы вверху и внизу экрана, чтобы компенсировать это. Для этого мы создаем масштабный прямоугольник, который начинается с координаты (0, barHeight) и имеет ширину, равную окну игры, и высоту,

необходимую для того, чтобы весь прямоугольник помещался на экране. Здесь `barHeight` — это половина необходимого отступа. Рис. 4-10 показывает наш масштабный прямоугольник, если он отображался в окне игры.



*Рис. 4-10. Нахождение масштабного прямоугольника,*

Наконец, мы еще раз изменили метод `Draw` для рендеринга в цель рендеринга, а затем рендеринг заднего буфера следующим образом:

```
protected override void Draw(GameTime gameTime)
{
    // Визуализация в Render
    GraphicsDevice.SetRenderTarget(_renderTarget);

    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    _currentGameState.Render(spriteBatch);

    spriteBatch.End();
```

```
// Теперь визуализируем масштабированное  
graphics.GraphicsDevice.SetRenderTarget(null);  
  
graphics.GraphicsDevice.Clear(ClearOptions.Target, Color.  
Black, 1.0f, 0);  
  
spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.  
Opaque);  
  
spriteBatch.Draw(_renderTarget, _renderScaleRectangle,  
Color.White);  
  
spriteBatch.End();  
  
base.Draw(gameTime);}
```

Здесь происходит несколько вещей. Во-первых, мы устанавливаем цель рендеринга на нашем графическом устройстве. Эта переменная `_renderTarget` создается в конструкторе следующим образом:

```
_renderTarget = new RenderTarget2D(graphics.GraphicsDevice,  
  
DESIGNED_RESOLUTION_WIDTH,DESIGNED_RESOLUTION_HEIGHT, false,  
SurfaceFormat.Color, DepthFormat.None, 0,  
RenderTargetUsage.DiscardContents);
```

Целевой объект рендеринга — это графический буфер, используемый для отрисовки элементов до тех пор, пока мы не будем готовы отправить их на экран. Пока мы рисуем на цели рендеринга, ничего не происходит на экране, пока мы не решим нарисовать эту цель рендеринга. Глядя на параметры, он устанавливает желаемое разрешение окна просмотра игры, область, которую мы хотим нарисовать. Он также устанавливает для флага `mipmap` значение `false`, цвет фона — черный (поскольку `SurfaceFormat.Color` равен нулю) и указывает, что мы не используем какой-либо буфер трафарета глубины и что наш `preferredMultiSampleCount` равен нулю (это используется при выполнении сглаживания). , и все, что мы рисуем в нашей цели рендеринга, не будет сохранено.

Затем графическое устройство очищается васильковым цветом, в результате чего экран окрашивается в этот же цвет. Теперь мы готовы к тому, что текущее состояние игры сделает свое дело и нарисует вещи! Для этого мы используем `spriteBatch`, который создается в методе `LoadContent`:

```
spriteBatch = new SpriteBatch(GraphicsDevice);
```

Мы кратко объяснили пакет спрайтов в главе 3. Это абстракция, которую мы будем использовать для отрисовки наших игровых примитивов на экране. Он в основном используется для наших спрайтов, то есть для наших игровых текстур, но он также может обрабатывать другие 2D-примитивы, такие как линии и прямоугольники. Он называется пакетом спрайтов, потому что мы добавим много спрайтов и примитивов в один пакет, который будет отправлен на видеокарту одним вызовом MonoGame. Более эффективно создавать один пакет на этапе отрисовки игрового цикла, чем несколько пакетов, хотя есть несколько причин, по которым разработчик игры может захотеть создать много пакетов за один этап отрисовки. Чтобы создать новый пакет в нашем движке, мы используем метод `spriteBatch.Begin`. Затем мы вызываем метод `Render` для текущего состояния игры и закрываем пакет спрайтов, вызывая `spriteBatch.End`.

Теперь, когда мы визуализировали один кадр в нашей цели рендеринга, мы готовы отрисовать его на экране, что мы и делаем, устанавливая цель рендеринга графических устройств в значение `null`. Мы начинаем с очистки экрана до черного цвета; затем мы выполняем еще одну фазу пакетной обработки спрайтов, где мы рисуем цель рендеринга в прямоугольнике масштаба, который мы рассчитали ранее. Поскольку экран изначально очищен черным, а цель рендеринга очищена до васильково-синего цвета, если проектное разрешение и разрешение игрового окна не совпадают, мы увидим черные полосы по бокам. Затем мы завершаем пакет спрайтов.

Добавление этой поддержки на раннем этапе разработки нашего движка помогает начать тестирование движка в различных разрешениях и форм-факторах, таких как экран ноутбука и монитор настольного компьютера. Теперь, когда у нас есть масштабирование окна, давайте добавим полноэкранную поддержку нашего окна.

## Полноэкранная поддержка

К счастью, в MonoGame добавить поддержку полноэкранного режима очень просто. Включение поддержки полноэкранного режима — всего одна строка. В следующем коде показано, как включить полноэкранный режим, установив для `graphics.IsFullScreen` значение `true`:

```
public MainGame(){  
    graphics = new GraphicsDeviceManager(this);  
  
    graphics.PreferredBackBufferWidth = 1024;  
    graphics.PreferredBackBufferHeight = 768;graphics.IsFullScreen =  
    true;Content.RootDirectory = "Content";  
}
```

## Система событий

Последним важным изменением в этой главе является добавление начальной работы над системой событий. Идея, лежащая в основе этого шаблона, состоит в том, чтобы иметь один объект вызова или класс, слушающий это конкретное событие, и делать то, что запрограммировано. Этот паттерн позволит нам по ходу книги добавлять все события, чтобы получилась полноценная игра.

## ГЛАВА 4 PLANNING YOUR GAME ENGINE

В рамках этой главы мы добавим одно событие, которое запускает игру для выхода. Чтобы вещи оставались строго типизированными, мы определим нумерацию следующим образом:

```
public enum Events{
```

```
    GAME_QUIT}
```

Затем в классе `BaseGameState` мы добавили новый метод `EventHandler` и:

```
public event EventHandler<Events> OnEventNotification;  
  
protected void NotifyEvent(Events eventType, object argument  
= null)  
{  
    OnEventNotification?.Invoke(this, eventType);  
  
    foreach (var gameObject in _gameObjects){  
        gameObject.OnNotify(eventType);}  
}
```

Идея заключается в том, что мы можем уведомить `MainGame`, который уже прослушивает уведомления о событиях, а также любые игровые объекты, которые существуют в рамках текущего состояния игры, вызвав метод `OnNotify`, который все они наследуют и могут переопределять из базового класса `BaseGameObject`:

```
public virtual void OnNotify(Events eventType) { }
```

Класс MainGame должен будет подключиться к событию OnEventNotification. Поскольку мы уже определили метод SwitchGameState, нам просто нужно добавить событие и определить реализацию следующим образом:

```
private void SwitchGameState(BaseGameState gameState){  
  
    _currentGameState?.UnloadContent(Content);  
  
    _currentGameState = gameState;_currentGameState.LoadContent  
(Content);  
    _currentGameState.OnStateSwitched += CurrentGameState_  
OnStateSwitched;  
    _currentGameState.OnEventNotification += _currentGameState_  
OnEventNotification;  
}  
  
private void _currentGameState_OnEventNotification(object sender,  
Enum.Events e)  
{  
    switch (e){  
        case Events.GAME_QUIT:      Exit();  
        break;}  
}
```

Поскольку для большинства событий не нужно уведомлять класс MainGame, это будет одним из немногих событий, которые вам нужно будет обрабатывать специально.

Последнее изменение, которое нам нужно сделать, — это обработать нажатие кнопки Enter в классе `GameplayState`, чтобы вызвать это событие. Для этого мы будем использовать код, который будет объяснен в главе 6, когда мы будем обсуждать различные способы захвата ввода игрока. Тем временем следующий код проверяет, нажата ли кнопка возврата на геймпаде или клавиша Enter на клавиатуре, и в этом случае он запускает событие `GAME_QUIT`:

```
public override void HandleInput(){  
  
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==  
        ButtonState.Pressed || Keyboard.GetState().IsKeyDown  
        (Keys.Enter))  
    {  
        NotifyEvent(Events.GAME_QUIT);  
    }  
}
```

## Резюме

В этой главе вы узнали о проектировании игрового движка и управлении состояниями, а также реализовали первоначальную архитектуру движка, которая будет способствовать дальнейшему развитию проекта.

В следующей главе мы углубимся в конвейер ассетов, обеспечивающий загрузку спрайтов, чтобы оживить наш недавно созданный движок.

## ГЛАВА 5

# Конвейер активов

Теперь, когда у нас есть четкое представление об архитектуре игрового движка, которая будет рассмотрена в этой книге, пришло время сосредоточиться на следующем важном компоненте нашего движка: ресурсах. Как кратко обсуждалось в [главе 3](#), MonoGame предоставляет простой в использовании и расширяемый интерфейс для доступа к активам. В ходе этой главы мы узнаем:

- Как работает конвейер ресурсов MonoGame.
- Как использовать инструмент MonoGame Asset Tool.
- Об интеграции Asset Pipeline в наш движок
- Как добавить спрайт игрока в игру

## Конвейер активов MonoGame

Для тех, у кого есть опыт работы с XNA, конвейер активов будет очень знаком, поскольку MonoGame строится на конвейере активов XNA. Основное изменение заключается в том, что XNA требует сжатия и упаковки ресурсов во время сборки. Это вызвало серьезную проблему для более крупных проектов, где время сборки и тестирования было значительным. К счастью, MonoGame переработала это, чтобы разделить построение конвейера активов и создание вашего кода, предоставив инструмент конвейера MonoGame (который будет обсуждаться в следующем разделе).

## ГЛАВА 5 ASSET PIPELINE

Кроме того, MonoGame продолжает предоставлять все преимущества конвейера XNA:

1. Расширяемость для поддержки пользовательских форматов файлов
2. Встроенная поддержка XML, видео, музыки, звука и изображений
3. Оптимизация сжатия изображений для каждой платформы
4. (DXTC, например, на ПК)
5. Загрузка системы с использованием дженериков C#

Конвейер на момент написания этой статьи поддерживает оптимизацию активов и таргетинг для

1. PC (Windows, Linux, MacOS X)
2. Консоли (Xbox 360, Xbox One, Switch, PS Vita, PSP, PS4)
3. Мобильный (iOS and Android)
4. Raspberry Pi

## Класс ContentManager

В основе конвейера активов внутри нашего движка, который мы будем продолжать развивать на протяжении всей этой книги, лежит класс [ContentManager](#). Этот класс предоставляет основной интерфейс для загрузки и извлечения контента различных типов, например звука, графики и уровней. На высоком уровне следующие методы являются основными методами обеспечения этой функциональности. Напоминаем, что исходный код этого класса и все примеры из этой книги доступны из прилагаемой папки с кодами.

## T LoadLocalized<T>(string assetName)

Метод LoadLocalized, как следует из названия, принимает параметр `assetName`, а затем строит локализованное имя объекта в цикле следующим образом:

```
string localizedAssetName = assetName + "." + cultureName
```

где `CultureName` является производным как от

`CultureInfo.CurrentCulture.Name`, так и от

`CultureInfo.CurrentCulture.TwoLetterISOLanguageName`. Например, первый вернет «en-US», а второй вернет «en». Для локализованных шрифтов, текстовой графики и аудиофайлов вместо `Load` следует использовать `LoadLocalized`.

Я должен отметить, что если локализованные активы не найдены, MonoGame автоматически возвращается к методу `Load`.

## T Load<T>(string assetName)

Метод `Load` принимает тип `T` и внутренне вызывает метод `ReadAsset`. К сожалению, в настоящее время ограничений на тип `T` нет; поэтому я должен предостеречь от типов, передаваемые внутрь. После успешного чтения актива объект добавляется во внутренний словарь загруженных активов на основе имени актива (оно используется в качестве ключа). Малоизвестно, что если тип `T` и имя\_актива соответствуют ранее существовавшей паре ключ/значение, вызов `ReadAsset` избегается и объект просто возвращается.

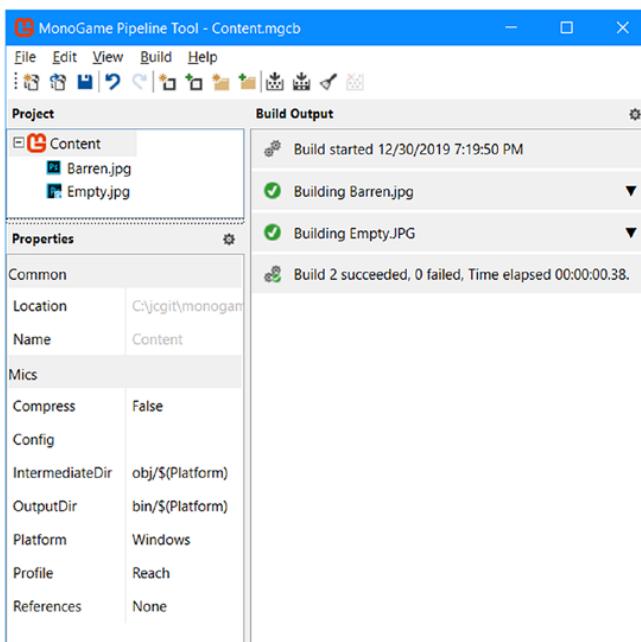
## void Unload()

Метод `Unload`, как следует из названия, вызывает метод `dispose` для всех одноразовых активов, которые были ранее загружены. Кроме того, также очищаются как коллекция «`loadedAssets Dictionary` - Загруженный словарь активов», так и коллекция «`disposableAlerts List` - Список одноразовых предупреждений».

В классе `Game`, который у нас есть и который мы будем использовать в этой книге, класс `ContentManager` доступен через свойство `Content`.

## Инструмент конвейера MonoGame

Ключом к конвейеру является инструмент конвейера MonoGame (изображенный на рис. 5-1). Этот инструмент обеспечивает четкое разделение задач между кодом вашего проекта MonoGame и вашими активами. Это разделение обеспечивает простой в использовании инструмент для художников, звукоинженеров и инженеров-программистов, совместно работающих над более крупными проектами. Проверка скомпилированного файла содержимого и ресурсов в системе управления исходным кодом — это простой способ разработки проекта в соответствии с agile-процессом (быстрым процессом).



*Рис. 5-1. Инструмент конвейера MonoGame*

Для тех, кому интересно, инструмент конвейера в выпуске 3.10 поддерживает MacOS X и Linux, тогда как до этого выпуска инструмент был только для Windows. Одна и та же функциональность существует на всех трех платформах без каких-либо известных различий на момент написания этой статьи.

## Интеграция конвейера активов в движок

Теперь, когда мы рассмотрели встроенную в MonoGame функциональность конвейера ресурсов, давайте реализуем правильный способ обработки загрузки и выгрузки ресурсов в нашем движке.

Вы помните, что в предыдущих примерах мы просто ссылались на [ContentManager](#) внутри методов [LoadContent](#) и [UnloadContent](#) в наших состояниях игры. Преимущество этого заключается в том, что он следует одноэлементному шаблону, но также не дает возможности обрабатывать глобальные активы. Причина в том, что при вызове [Unload](#) в [ContentManager](#) все активы выгружаются. В игре у вас, скорее всего, есть общие активы, такие как шрифты, звуки и графика, которые вы не хотели бы перезагружать при каждом игровом состоянии. Вам может быть интересно, почему [ContentManager](#) не предоставляет способ выгрузки определенных объектов — это обсуждалось и запрашивалось как функция в будущей версии.

Чтобы обойти это ограничение в текущей версии MonoGame, общепринятым решением является просто пропустить основной [ContentManager](#) и создать локальную копию в каждом игровом состоянии. В рамках данной главы мы будем применять именно это решение.

Теперь давайте погрузимся в улучшения нашего движка!

## BaseGameState

Вы, возможно, помните, что абстрактный класс [BaseGameState](#) является основным классом, от которого происходят все состояния нашего игрового движка. В рамках этой главы мы внесем несколько изменений для поддержки новой загрузки контента.

Первое изменение заключается в добавлении постоянной переменной для резервной текстуры (будет обсуждаться в следующем разделе):

```
private const string FallbackTexture = "Empty";
```

## ГЛАВА 5 ASSET PIPELINE

Второе изменение заключается в добавлении закрытой переменной для хранения класса ContentManager:

```
private ContentManager _contentManager;
```

Третье изменение заключается в добавлении нового метода для инициализации переменной privateContentManager:

```
public void Initialize(ContentManager contentManager)
{
    _contentManager = contentManager;
}
```

Четвертое изменение заключается в замене абстрактного метода UnloadContent на реализованный метод, который вызывает наш новый метод Unload в privateContentManager:

```
public void UnloadContent()
{
    _contentManager.Unload();
}
```

Последнее изменение заключается в добавлении обертки вокруг загрузки текстур вместе с откатом к нашей запасной текстуре в случае отсутствия текстурного актива:

```
protected Texture2D LoadTexture(string textureName)
{
    var texture = _contentManager.Load<Texture2D>(textureName);

    return texture ?? _contentManager.Load<Texture2D>
        (FallbackTexture);
}
```

Как отмечалось в подробном обзоре класса ContentManager, такой подход к переносу загрузки текстуры позволяет избежать случайной передачи несовместимого типа T в метод `Load<T> ContentManager` и тем самым вызвать исключение.

Имея эти изменения, мы теперь можем переключить внимание на следующие изменения классов.

## Основная игра

Поскольку большинство изменений происходит в `BaseGameState`, необходимо внести несколько изменений в класс `MainGame`.

Первое изменение заключается в обновлении разработанного разрешения до 1280x720 (720p):

```
private const int DESIGNED_RESOLUTION_WIDTH = 1280;
private const int DESIGNED_RESOLUTION_HEIGHT = 720;
```

Причиной этого изменения по сравнению с ранее использовавшимся разрешением 1024x768 является настоящий актив экрана-заставки, загружаемый в следующем разделе.

Следующее изменение заключается в том, чтобы настроить метод `SwitchGameState` для поддержки нового метода `Initialize` и не передавать `ContentManager` в метод `LoadContent`:

```
private void SwitchGameState(BaseGameState gameState)
{
    _currentGameState?.UnloadContent();

    _currentGameState = gameState;

    _currentGameState.Initialize(Content);

    _currentGameState.LoadContent();
```

```
_currentGameState.OnStateSwitched += CurrentGameState_
    OnStateSwitched;
_currentGameState.OnEventNotification += _currentGameState_
    OnEventNotification;
}
```

С этими изменениями движок теперь предоставляет четкий метод для загрузки текстуры и правильной очистки контента при изменении состояний игры. Изменения и новые активы для завершения этой работы обсуждаются в следующем разделе. В следующих главах мы расширим эту функциональность, включив аудио, шрифты и XML-файлы.

## Добавить спрайт игрока в игру

Теперь, когда движок был обновлен для поддержки загрузки текстур, нам нужно всего лишь внести несколько небольших изменений в код нашей игры. Помните, что вы можете получить доступ к исходному коду, перейдя на [www.apress.com/ISBN](http://www.apress.com/ISBN).

Как и в предыдущих главах, в каталогах главы-05 есть три компонента:

1. ресурсы
2. Начало
3. конец

Папка с ресурсами содержит файлы Adobe Photoshop (PSD) и PNG для новой заставки, образца текстуры и нового бойца. Эти файлы были предоставлены для использования в этой главе и для будущего использования в других проектах.

Стартовая папка содержит код, предшествующий любым изменениям, внесенным в этой главе, чтобы вы могли следовать ему. И наоборот, конечная папка содержит завершенный код, если вы хотите просто вернуться к главе.

## Обзор новых активов

Чтобы продемонстрировать новую функциональность загрузки текстур, в решение были добавлены три новых ресурса.

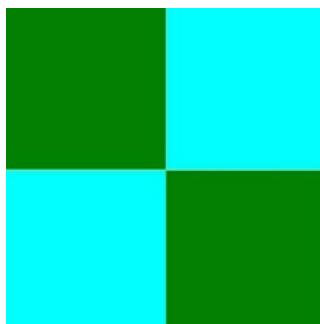
Первый — это настояще изображение экрана-заставки, а не просто использование текстуры земли, которая использовалась ранее. Этот экран-заставка был отрендерен до разрешения 1280x720 (изображен на рис. 5-2).



*Рис. 5-2. Актив экрана-заставки*

Это изображение будет действовать как изображение в нашем классе `SplashState`.

Второй актив представляет собой пустую графику, которую можно использовать, если конкретный актив текстуры не найден или во время загрузки возникает ошибка (изображено на Рис. 5-3).



*Рис. 5-3. Запасная текстура*

Этот подход позволит легко быстро увидеть, если ресурс не загружается. В ваших будущих проектах я настоятельно рекомендую либо использовать эту графику, либо что-то, что больше нигде в вашем проекте не используется, чтобы указать, что при тестировании произошла ошибка.

Последний ассет — это спрайт нашего истребителя, изображенный на рис. 5-4.

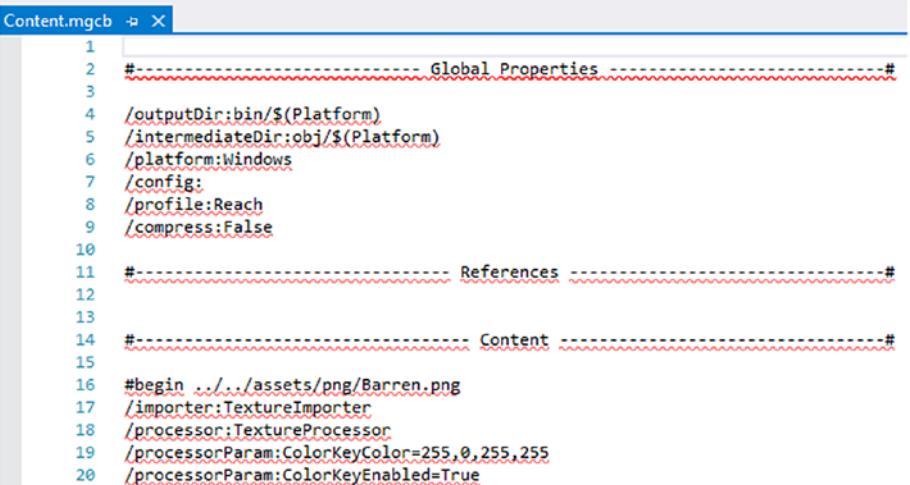


*Рис. 5-4. Спрайт игрока*

Он будет использоваться в качестве спрайта игрока в будущих главах и, как вы могли видеть, содержит прозрачность, которую MonoGame автоматически подберет во время конвейера активов.

## Добавление новых ресурсов к нашему контенту

Чтобы добавить эти новые активы, дважды щелкните Content.mgcb в Visual Studio, как мы это делали ранее. Если файл открывается как XML-файл, как на рис. 5-5, выполните следующие действия.



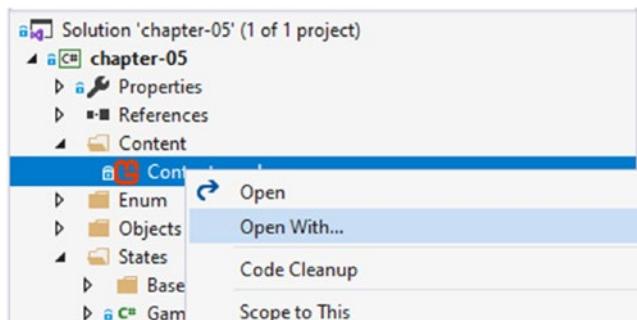
```

Content.mgcb
1
2 #----- Global Properties -----#
3
4 /outputDir:bin/$(Platform)
5 /intermediateDir:obj/$(Platform)
6 /platform:Windows
7 /config:
8 /profile:Reach
9 /compress:False
10
11 #----- References -----#
12
13
14 #----- Content -----#
15
16 #begin ../../assets/png/Barren.png
17 /importer:TextureImporter
18 /processor:TextureProcessor
19 /processorParam:ColorKeyColor=255,0,255,255
20 /processorParam:ColorKeyEnabled=True

```

*Рис. 5-5. Content.mgcb неправильно открыт как файл XML*

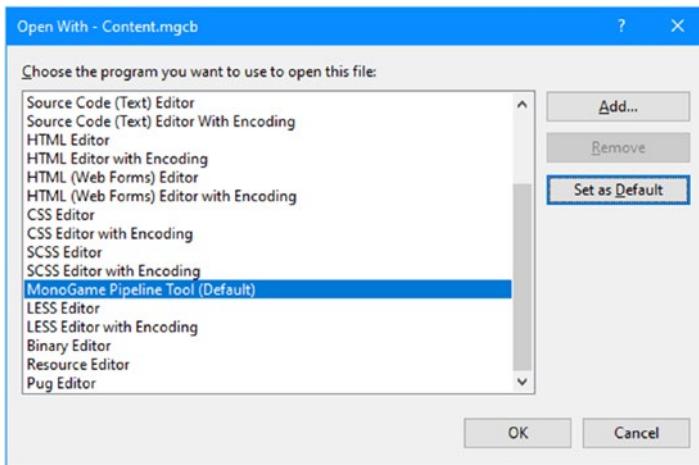
Шаг 1 — щелкнуть правой кнопкой мыши файл Content.mgcb, как показано на рис. 5-6, и выбрать «Open With.... - Открыть с помощью...».



*Рис. 5-6. Контекстное меню Visual Studio в Content.mgcb*

## ГЛАВА 5 ASSET PIPELINE

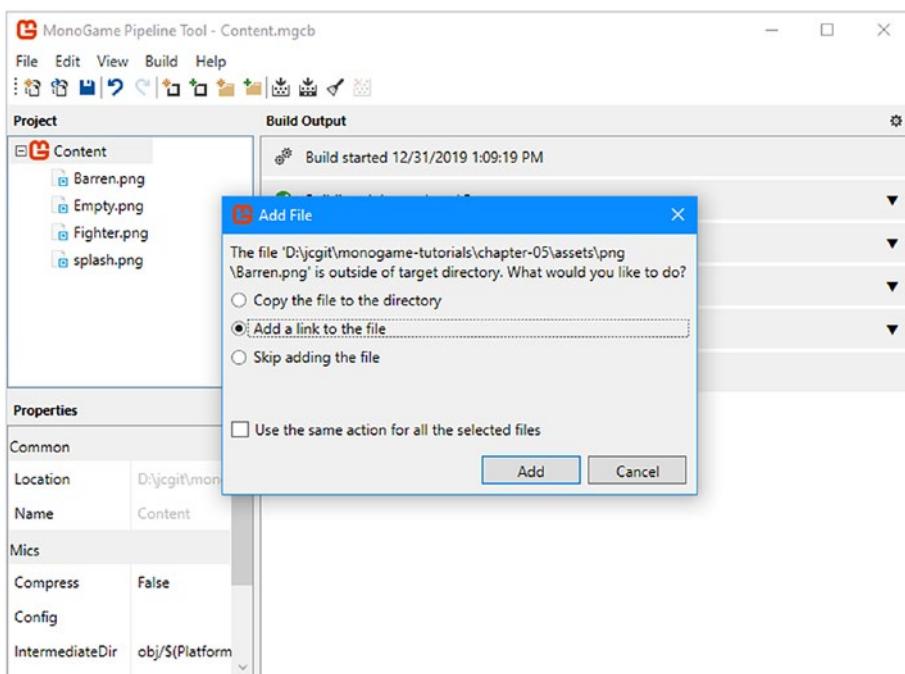
Как только окно откроется, вам будет представлен диалог. Ваше представление может включать другие параметры, не указанные в списке; опция, которую вы ищете, — это MonoGame Pipeline Tool, как выделено на рис. 5-7.



**Рис. 5-7. Visual Studio Открыть с диалоговым окном**

После выбора нажмите «Set as Default - Установить по умолчанию» (чтобы это не повторилось), а затем нажмите «OK». После нажатия OK откроется инструмент MonoGame Pipeline.

При добавлении содержимого, как в предыдущих главах, обязательно выберите «Add a link to the file - Добавить ссылку на файл» вместо «Copy the file to the directory - Скопировать файл в каталог», как показано на Рис. 5-8.



*Рис. 5-8. Диалоговое окно добавления файла MonoGame Pipeline Tool*

Нажав на опцию «Add a link to the file - Добавить ссылку на файл», мы избежим дублирования расположения контента. Когда вы работаете в команде, состоящей из нескольких человек, или просто хотите иметь один источник правды, этот процесс позволяет избежать ненужного оттока и забывания обновить несколько местоположений.

После добавления ресурсов Empty, Fighter и Splash обязательно щелкните Build>Build или нажмите клавишу F6, как показано на Рис. 5-9.

## ГЛАВА 5 ASSET PIPELINE

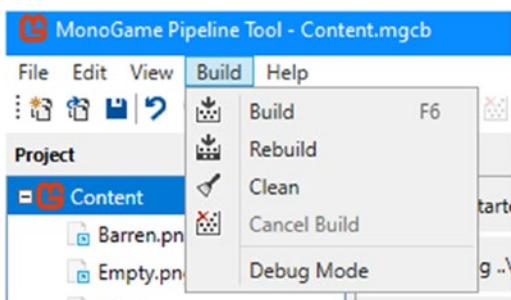


Рис. 5-9. Меню сборки MonoGame Pipeline Tool

После создания пакета содержимого вы должны увидеть то же сообщение, что и на рис. 5-10.

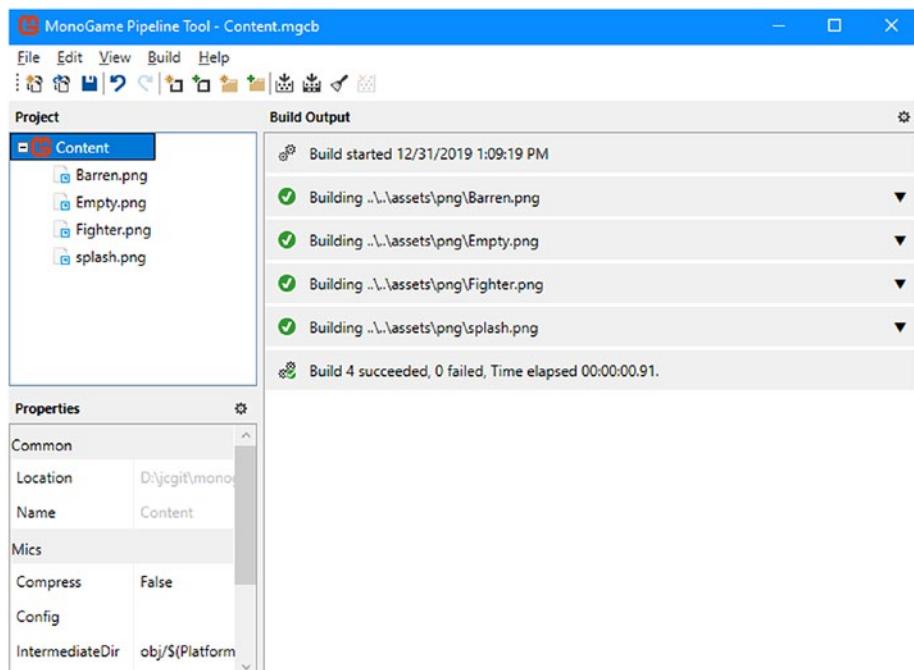


Рис. 5-10. Инструмент MonoGame Pipeline, показывающий завершение сборки

# Изменения игрового кода

Поскольку изменения в нашем движке были завершены в предыдущем разделе, осталось лишь несколько изменений, необходимых для изменения кода нашей игры.

## SplashState

Как вы помните, класс `SplashState` — это состояние, которое запускается при запуске нашей игры. В следующих главах мы расширим функциональные возможности, включив в них эффекты перехода и музыку.

Для объема этой главы необходимо внести три изменения. Если следующий код не ясен, пожалуйста, вернитесь к главе 4 или используйте полный исходный код для этой главы.

Первое изменение — это удаление метода `UnloadContent`. При этом движок автоматически обрабатывает выгрузку контента внутри класса `BaseGameState`. Это удаление упростит и уменьшит дублирующийся код, поскольку мы продолжаем добавлять больше игровых состояний (следуя мантре «Не повторяйся»).

Второе изменение касается метода `LoadContent`. Раньше мы использовали класс `MonoGame ContentManager` напрямую для загрузки текстуры. Как было рассмотрено ранее в этой главе, метод `LoadTexture` обеспечивает абстракцию между `MonoGame` и нашим движком. Для этого есть несколько причин, но некоторые из них включают в себя лучшую обработку изменений в `MonoGame` (т. е. если синтаксис загрузки текстуры меняется, его можно обрабатывать в одном месте, а не во всем коде) в дополнение к расширяемости в рамках вашего собственного кода.

С изменениями, внесенными в движок, теперь мы можем просто вызвать `LoadTexture` следующим образом:

```
public override void LoadContent()
{
    AddGameObject(new SplashImage(LoadTexture("splash")));
}
```

Последнее изменение — в методе `HandleInput`. Раньше этот метод искал любое нажатие клавиши. Продолжая тему превращения `SplashState` в настоящую заставку, метод теперь ищет только нажатие клавиши Enter. Код нового метода:

```
public override void HandleInput(){
    var state = Keyboard.GetState();

    if (state.IsKeyDown(Keys.Enter)){
        SwitchState(new GameState());
    }
}
```

## Состояние игры

Как вы помните, класс `GameState` — это состояние, которое будет удерживать наш основной игровой процесс по мере прохождения глав.

Для объема этой главы необходимы четыре изменения.

Первое изменение — добавление объявления наших текстур как константных переменных, чтобы избежать волшебных строк в нашем коде:

```
private const string PlayerFighter = "fighter"; private const string
BackgroundTexture = "Barren";
```

В следующих главах мы преобразуем эти константы в файлы JSON, чтобы обеспечить большую гибкость и избежать жесткого кодирования текстур.

Второе изменение, как и в классе `SplashState`, заключается в удалении метода `UnloadContent`, так как он обрабатывается внутри базового класса.

Третье изменение, которое необходимо внести, — это загрузка образца текстуры земли и спрайта бойца, которые мы рассмотрели в предыдущем разделе. Мы можем реализовать эту функциональность следующим образом:

```
public override void LoadContent(){
    AddGameObject(new SplashImage(LoadTexture(BackgroundTexture)));
    AddGameObject(new PlayerSprite(LoadTexture(PlayerFighter)));
}
```

Последнее изменение заключается в том, чтобы переключить ввод с клавиатуры так, чтобы он прослушивал только нажатие клавиши Escape, например:

```
public override void HandleInput(){
    var state = Keyboard.GetState();

    if (state.IsKeyDown(Keys.Escape)){
        NotifyEvent(Events.GAME_QUIT);
    }
}
```

В следующей главе мы пересмотрим это, чтобы не привязываться только к вводу с клавиатуры; однако пока мы привязаны только к клавиатуре.

## Запуск приложения

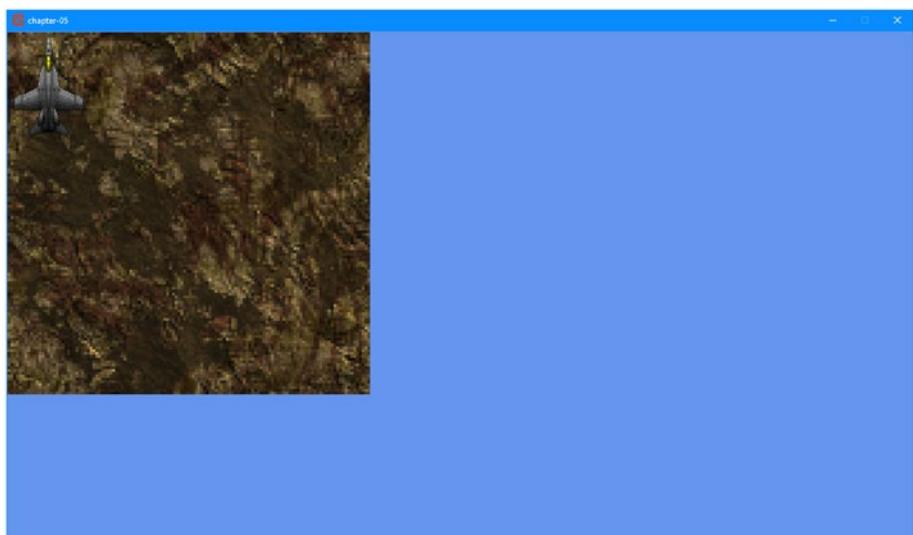
Теперь, когда в код внесены изменения, простой запуск приложения должен отобразить экран-заставку, как показано на Рис. 5-11.

## ГЛАВА 5 ASSET PIPELINE



*Рис. 5-11. Пример из главы 5, показывающий экран-заставку*

После нажатия Enter вы должны увидеть текстуру земли и спрайт бойца нашего игрока, как показано на Рис. 5-12.



*Рис. 5-12. Пример из главы 5, показывающий спрайт игрока, наложенный поверх пустого спрайта*

Чтобы выйти, нажмите клавишу Escape на клавиатуре или нажмите X, чтобы закрыть окно.

## Резюме

В этой главе вы узнали о конвейере ресурсов MonoGame. Мы также погрузились в интеграцию Asset Pipeline в наш движок для поддержки текстур и создания подхода, который мы будем использовать в следующих главах, чтобы быстро добавить поддержку аудио и видео. Наконец, мы продемонстрировали, как использовать нашу новую функциональность для загрузки изображения-заставки в дополнение к спрайту игрока в наш развивающийся игровой движок.

В следующей главе мы углубимся в обработку ввода в нашем игровом движке, добавив поддержку сенсорного ввода, клавиатуры, мыши и контроллера.

## ГЛАВА 6

# Вход

Это не была бы видеоигра, если бы игроки не могли каким-то образом манипулировать состоянием игры.

Теперь мы должны изучить входные данные из трех разных источников, а именно, клавиатуры, мыши или геймпада, и сопоставить эти входные данные с действиями, которые изменят состояние игры, такими как перемещение игрока по экрану и стрельба пулями. Когда мы закончим с этой главой, у вас будет игра, похожая на Рис.6-1. Мы обещаем, что пули будут выглядеть лучше, когда они двигаются.



*Рис. 6-1. Конечный результат*

В этой главе вы

- Реализируете прокручивающийся фон
- Узнаете, как управлять спрайтом игрока с помощью клавиатуры.
- Реализируете универсальный механизм ввода
- Узнаете, как использовать другие источники ввода с помощью MonoGame.

## Обсуждение различных механизмов ввода

MonoGame поддерживает почти все механизмы пользовательского ввода, используемые игроками по всему миру. Являетесь ли вы геймером на ПК с клавиатурой и игрой, консольным игроком с геймпадом или тем, кто любит играть на устройствах с сенсорным экраном, таких как смартфоны, MonoGame предоставит вам готовую защиту. Когда игрок нажимает кнопку на клавиатуре или геймпаде, MonoGame отслеживает состояние этого конкретного устройства ввода. Нашей игре просто нужно регулярно запрашивать это состояние, чтобы реагировать на то, что игрок пытается выполнить.

Чтобы продолжить, откройте решение главы 06, скопированное из папки с кодами: /monogame-mastery. Каталог Chapter-06 содержит два решения: `start` и `end`. Поскольку мы находимся в начале этой главы, давайте откроем начальное решение.

## Состояние клавиатуры

Мы кратко обсуждали ввод с клавиатуры в конце главы 5. Два наших класса состояния игры, `SplashState` и `GameState`, реализуют функцию `HandleInput()`, которая постоянно вызывается классом `MainGame`. Цель этой функции - управлять нашими устройствами ввода и реагировать на действия игрока. Рассмотрим интересующую нас в данный момент часть класса `GameState`. Мы скрыли часть кода для краткости, но в окончательном решении главы 06 будет весь код.

```
using Microsoft.Xna.Framework.Input;

public class GameplayState : BaseGameState
{
    public override void HandleInput()
    {
        var state = Keyboard.GetState();

        if (state.IsKeyDown(Keys.Escape))
        {
            NotifyEvent(Events.GAME_QUIT);
        }
    }
}
```

Поддержка клавиатуры в MonoGame проста. Большинство клавиш либо нажимаются, либо отпускаются, за некоторыми исключениями из этого правила для клавиш, которые включаются и выключаются, например Num Lock и Caps Lock. В предыдущем коде мы начинаем с запроса класса Keyboard о его текущем состоянии. Затем мы запрашиваем у этого состояния информацию, которая имеет отношение к нашей игре. Прямо сейчас мы должны только дать игроку возможность выйти из игры с помощью клавиши Escape. Клавиша Escape нажата в данный момент? Это то, что задает `stateIsKeyDown(Keys.Escape)`. Если это так, мы запускаем событие `GAME_QUIT`, и класс `MainGame` отвечает, сообщая нашей программе о выходе.

Какие ключи мы можем рассматривать? Их слишком много, чтобы их можно было подробно перечислить в книге. Однако мы можем использовать Visual Studio 2019 для проверки доступных параметров с помощью IntelliSense или путем проверки метаданных библиотеки MonoGame.

Удалите Escape-слово из `Keys.Escape` и нажмите Ctrl+Enter, чтобы активировать `IntelliSense`, чтобы открыть всплывающее окно. Вы должны увидеть что-то вроде изображения на [рис. 6-2](#), и прокрутка опций должна дать вам представление о большом разнообразии ключей, которые можно отслеживать.

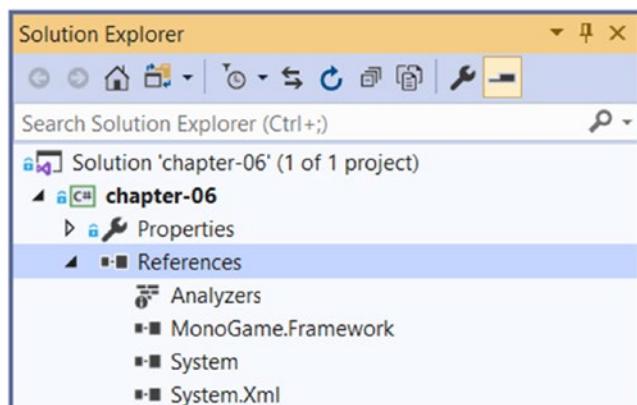
## ГЛАВА 6 INPUT



*Рис. 6-2. Параметры Intellisense для перечисления Keys*

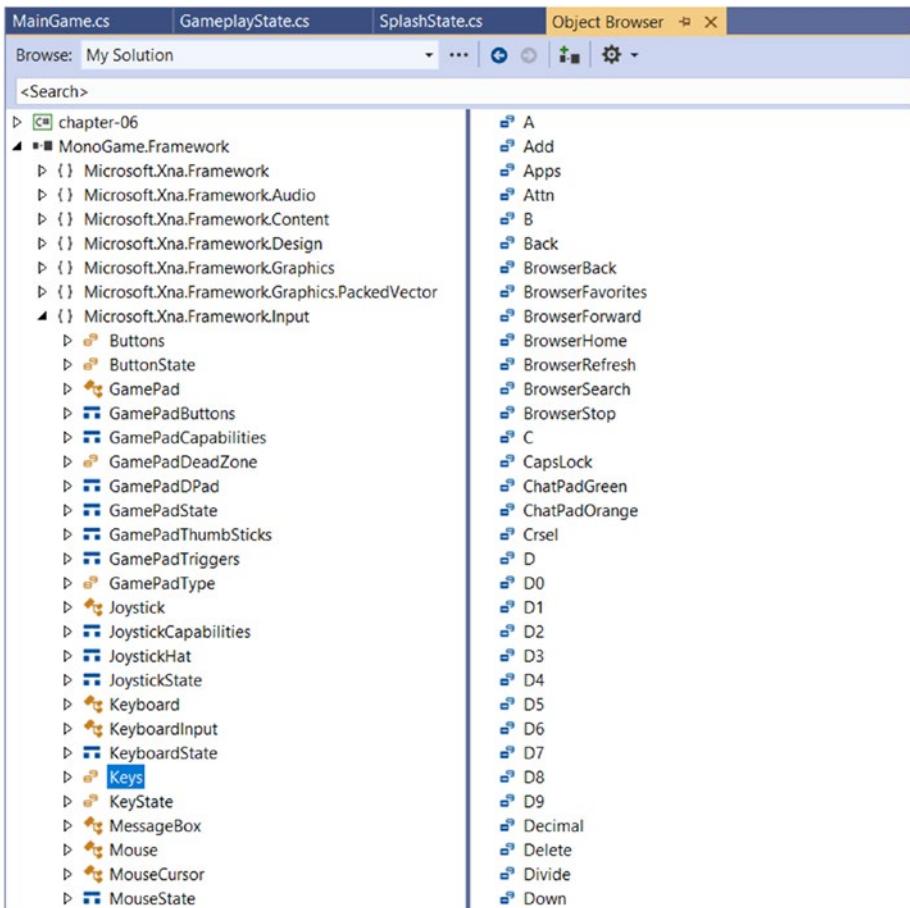
Другой способ просмотреть параметры — просмотреть метаданные библиотеки MonoGame напрямую. Мы предпочитаем этот подход при изучении того, какую функциональность предоставляет нам библиотека. Давайте изучим, что есть в `Microsoft.Xna.Framework.Input`.

На панели `Solution Explorer` разверните список `References`, щелкнув стрелку слева от него. Рисунок 6-3 показывает нам, что `MonoGame.Framework` — это ссылка, которую мы добавили в наш проект.



*Рис. 6-3. Список референсов, используемых в нашей игре*

Теперь мы можем проверить его, дважды щелкнув ссылку [MonoGame.Framework](#). Это откроет обозреватель объектов на новой вкладке в Visual Studio. Оттуда мы можем начать изучение. Разверните [MonoGame.Framework](#) и [Microsoft.Xna.Framework.Input](#), а затем щелкните [Keys - Ключи](#). Вы должны увидеть что-то вроде того, что показано на Рис. 6-4.



*Рис. 6-4. Все нажатые клавиши, которые может отслеживать MonoGame*

Теперь вы можете проверить все значения, которые являются частью этого перечисления Keys. То, что вы также можете увидеть, это все, что MonoGame предлагает нам для всех наших потребностей ввода.

## Состояние мыши

Наша игра не сразу поддерживает использование мыши во время игры, но нам может понадобиться использовать мышь, чтобы указывать и щелкать меню. В большинстве видеоигр используются меню, позволяющие игроку настраивать звук или графику или предлагающие игроку возобновить игру или выйти из нее.

Обычно курсор мыши скрыт от окна просмотра во время игры. Это можно изменить, добавив эту строку в функцию `Initialize()` основной игры:

```
this.IsMouseVisible = true;
```

Однако, даже когда курсор мыши скрыт, состояние мыши все еще можно использовать для наблюдения за тем, как игрок использует свою мышь. Мы можем посмотреть на состояние левой, средней и правой кнопок. Мы также можем отслеживать координаты X и Y мыши и изменения в колесе прокрутки с момента запуска игры, что может быть полезно для реализации функции масштабирования камеры.

Если бы мы хотели, чтобы игрок стрелял пулями при нажатии левой кнопки мыши, мы бы использовали этот код:

```
var mouseState = Mouse.GetState();

if (mouseState.LeftButton == ButtonState.Pressed)
{
    // Perform shooting action!
}
```

Доступ к координатам X и Y мыши осуществляется с помощью следующего кода:

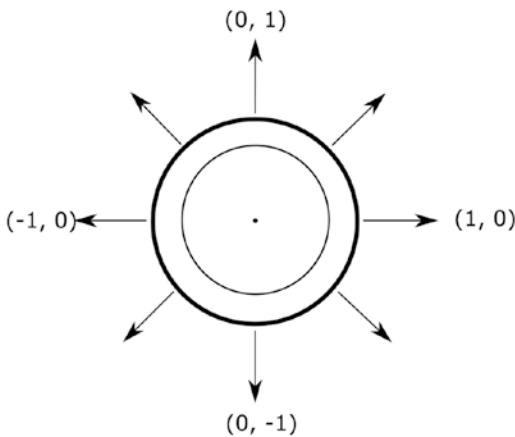
```
var mouseState = Mouse.GetState();  
  
var x = mouseState.Position.X;  
var y = mouseState.Position.Y;
```

Обратите внимание, что это даст вам координаты X и Y на основе (0, 0) начала области просмотра, расположенной в левом верхнем углу окна просмотра игры, чуть ниже строки заголовка.

## Состояние геймпада

Геймпад — невероятно полезное игровое устройство. Это не только основное устройство, используемое в домашних игровых консолях по всему миру, но и игроки иногда предпочитают использовать его на своем компьютере. Мониторинг состояния геймпада не сложнее, чем то, что мы узнали до сих пор. Есть несколько кнопок, которые можно нажимать, а состояние левого или правого джойстика представлено с помощью 2D-вектора со значениями X и Y от -1 до 1, где нулевое значение для обеих координат указывает на то, что джойстик полностью неподвижен, идеально отцентрирован.

У стиков также есть мертвая зона, представляющая собой небольшую область вокруг ее центра, которая не регистрирует никаких движений (рис. 6-5). Эта мертвая зона решает две проблемы. Во-первых, игроки, опирающиеся большими пальцами на стик, все еще могут заставить его двигаться или дрожать, но они не намерены заставлять своих персонажей на экране двигаться. Мертвая зона гарантирует, что это дрожание останется незамеченным игрой. Во-вторых, геймпады со временем изнашиваются, а центры джойстиков могут немного смешаться от положения (0, 0). Мертвая зона в этом случае не позволяет игре заметить любое движение. Без нее персонаж игрока будет двигаться, даже если никто не касается джойстика.



*Рис. 6-5. Значения X и Y джойстика в формате (x, y)*

Чтобы перемещать игрока с помощью джойстика, вам понадобятся две вещи:

- Скорость игрока
- Значения X и Y с помощью джойстика

Когда джойстик сдвинут до упора влево, мы зарегистрируем значение X, равное -1, и значение Y, равное 0. Если бы джойстик находился наполовину влево, значение X было бы равно 0,5.

Код для отслеживания ввода с помощью джойстика будет выглядеть следующим образом:

```
var gamepadState = GamePad.GetState(PlayerIndex.One);
var newPlayerPosition = new Point(
    oldPlayerPosition.X + (gamepadState.ThumbSticks.Left.X *
    playerSpeed),
    oldPlayerPosition.Y + (gamepadState.ThumbSticks.Left.Y *
    playerSpeed)
);
```

Таким образом, если левый мини-джойстик был полностью влево, X должен быть равен -1, а Y должен быть равен 0, таким образом перемещая персонажа, уменьшая его старое положение на значение playerSpeed и вообще не изменяя его положение Y.

Нам также нужно указать, для какого геймпада мы хотим получить состояние. Большинство игровых консолей позволяют играть до четырех игроков. В этом примере нас интересовало, чем занимается игрок 1.

Теперь, когда мы рассмотрели, как различные входы работают в MonoGame, мы готовы добавить собственный код! Наша игра позволит игроку управлять самолетом с помощью клавиш со стрелками на клавиатуре и сбивать врагов с помощью пробела. Но прежде чем мы доберемся до этого, нам нужен еще один элемент в нашей игре: прокручивающийся фон.

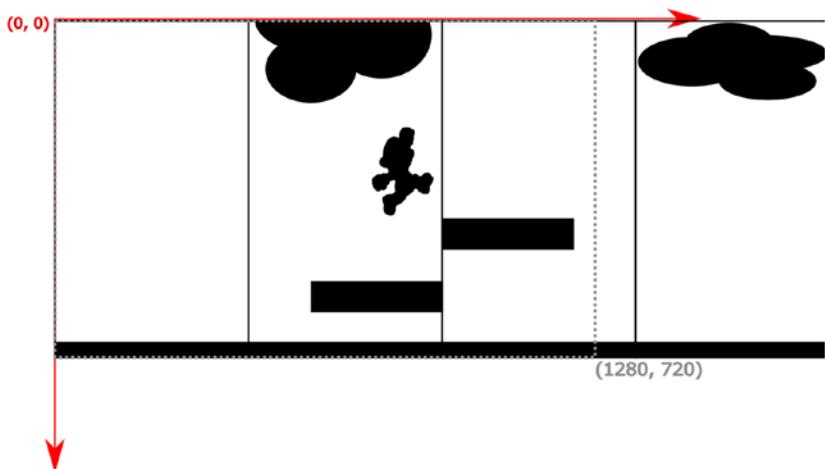
## Прокрутка фона

Мы все знаем, как работают боковые скроллеры. Такие игры, как Super Mario Bros, где персонаж может двигаться только вбок, влево или вправо, а фон «прокручивается», когда персонаж, всегда находящийся в середине экрана, «двигается» вокруг. По сути, персонаж фиксируется на месте. Это фон, который движется и создает иллюзию движения.

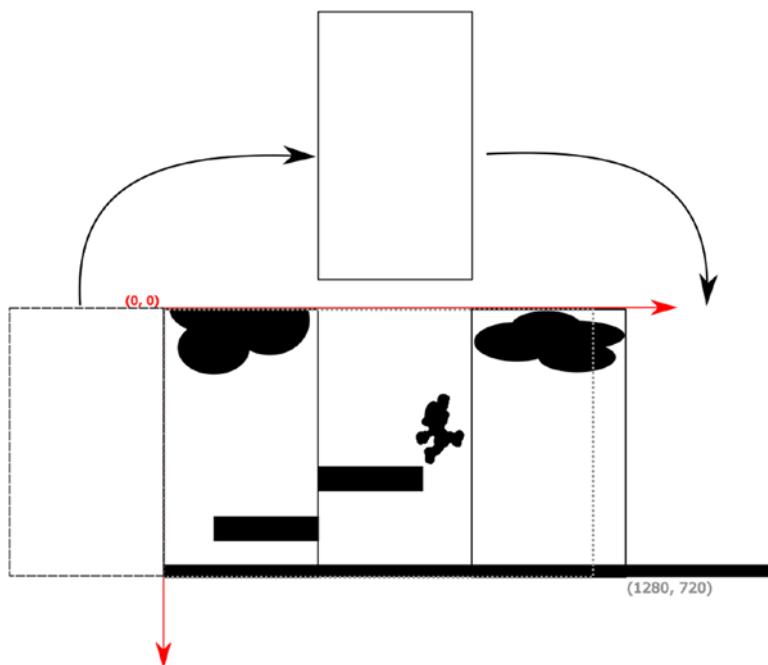
Как разработчики игр достигают этой иллюзии? Они делают это, перемещая фоновые блоки на экране и удерживая персонажа в центре.

Игра заполнена фоновыми блоками, которые заполняют окно просмотра. Когда игрок хочет переместить персонажа вправо, мы вместо этого берем все фоновые блоки и перемещаем их влево. Если бы фонового блока не было вне поля зрения, мы бы заметили зазор, образовавшийся на правом краю экрана. Но поскольку у нас есть фоновый блок в этой области, он сразу же перемещается, чтобы заполнить этот пробел. По мере того, как игрок продолжает перемещать персонажа вправо, крайний левый фоновый блок в конечном итоге полностью оказывается за пределами экрана. В этот момент игра обновит положение этого блока, чтобы он оказался справа от области просмотра, готовый к прокрутке в поле зрения. Вместо этого перемещение символа влево вызовет обратный процесс. [Рис. 6-6](#) и [6-7](#) покажут этот процесс, как он происходит..

## ГЛАВА 6 INPUT



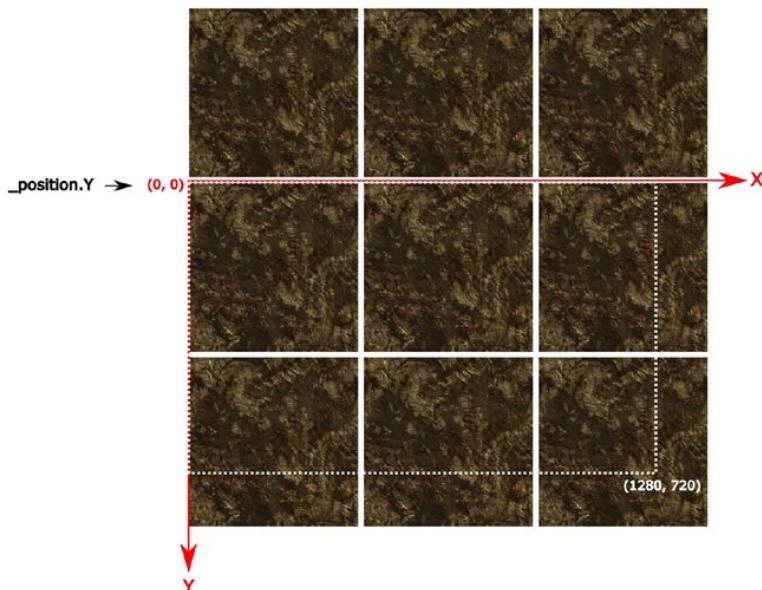
*Рис. 6-6. Фоновые блоки заполняют область просмотра, показанную пунктирной линией.*



*Рис. 6-7. Наш персонаж движется вправо*

Наша игра будет работать аналогичным образом, поскольку нам нужно создать иллюзию того, что наш истребитель движется вперед, в направлении вверх, вместо этого мы будем работать над тем, чтобы фон двигался вниз.

Нашим первым шагом будет заполнение области просмотра плиткой фонового ландшафта и добавление дополнительного ряда плиток прямо над областью просмотра, чтобы они были готовы к прокрутке вниз в поле зрения (рис. 6-8). Наша фоновая текстура была разработана таким образом, чтобы обеспечить бесшовную непрерывность при наложении плитки со всех сторон. По мере того, как игра идет и значение `_position.Y` увеличивается и удаляется от источника, плитки будут двигаться вниз, в результате чего плитки, которые были за пределами экрана в начале, прокручиваются в поле зрения (Рис. 6-9). В конце концов, верхний ряд тайлов будет полностью виден в верхней части области просмотра, а фон вернется в исходное положение по оси Y; затем прокрутка возобновится и создаст иллюзию бесконечной местности (Рис. 6-10).



**Рис. 6-8.** Наш мозаичный фон. Между плитками оставлены пробелы, чтобы проиллюстрировать, что это выглядит как сетка. В игре плитки будут соприкасаться друг с другом, и места не останется.

## ГЛАВА 6 INPUT

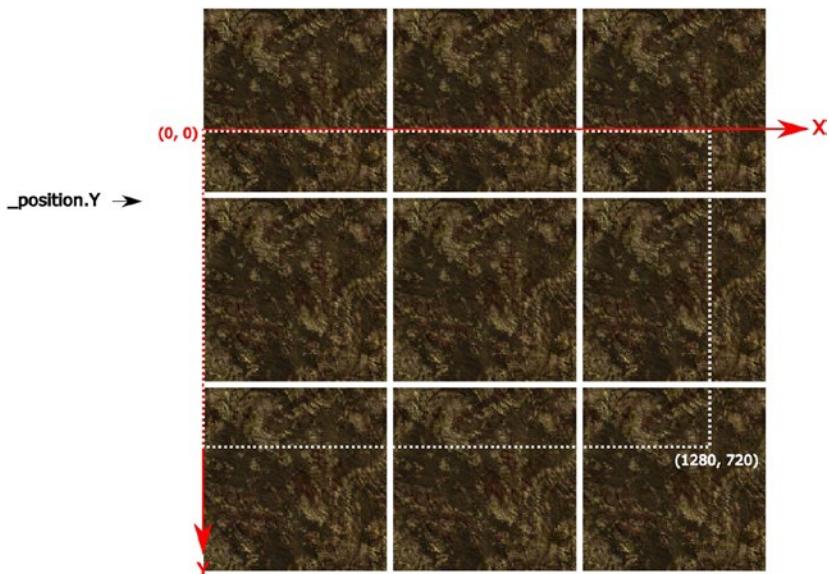


Рис. 6-9. Фон прокручивается вниз

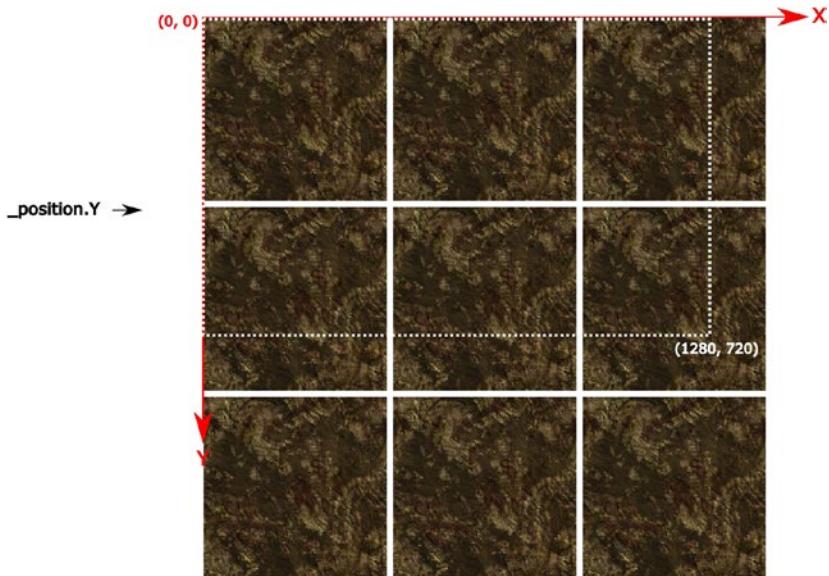


Рис. 6-10. Конец нашей системы прокрутки

Давайте создадим новый класс под названием `TerrainBackground` и добавим его в наследуемый объект `GameObject`. Как и другие объекты базовой игры, он должен быть создан с существующей текстурой и позицией, которую мы установили в  $(0, 0)$ , с целью перемещения этой позиции вниз по оси Y, чтобы она прокручивалась.

```
public class TerrainBackground : BaseGameObject
{
    private float SCROLLING_SPEED = 2.0f;

    public TerrainBackground(Texture2D texture)
    {
        _texture = texture;
        _position = new Vector2(0, 0);
    }
}
```

Помните, что `_texture` и `_position` являются защищенными переменными в родительском классе `BaseGameObject`, что означает, что они доступны для использования внутри кода `TerrainBackground`, и в этом случае мы инициализируем `_texture` входным параметром и удостоверяемся, что `_position` установлено в исходных координатах  $(0, 0)$ .

Рисование фона будет немного сложнее, чем то, что делает метод `Render()` `BaseGameObject`. Вместо того, чтобы просто рисовать текстуру объекта в одном месте, мы должны заполнить окно просмотра текстурой, а также нарисовать ряд рельефа над экраном. Для этого мы будем использовать другую версию функции `SpritBatch.Draw`, которую использует базовый класс.

Для справки, вот как выглядит функция рендеринга базового класса:

```
public virtual void Render(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(_texture, _position, Color.White);
}
```

## ГЛАВА 6 INPUT

Функция рисования имеет множество перегруженных версий и принимает множество различных параметров. Здесь он ограничивается отрисовкой определенной текстуры в определенной позиции, которую мы указали при создании экземпляра нашего игрового объекта. Параметр `Color.White` — это цветовая маска, которая в данный момент нас не особо интересует, за исключением того, что белая маска указывает MonoGame отображать текстуру как есть.

Однако мы должны использовать другую перегруженную функцию `Draw`:

```
spriteBatch.Draw(_texture, destinationRectangle,
    sourceRectangle, Color.White);
```

Здесь мы сообщаем пакету спрайтов, чтобы он отрисовывал определенный прямоугольник нашей текстуры в определенный прямоугольник области просмотра, используя маску белого цвета. Указание исходного прямоугольника полезно, когда исходная текстура содержит множество спрайтов на одной и той же текстуре, что обычно бывает в играх со спрайтовой анимацией, в которой все кадры анимации игрового объекта, например персонаж, подпрыгивающий вверх и вниз во время бездействия, расположены в сетке в одном файле.

Когда мы указываем целевой прямоугольник другого размера, чем исходный прямоугольник, MonoGame растягивает или сжимает изображение, чтобы оно соответствовало размеру. Однако в нашем случае оба прямоугольника будут одинакового размера, и мы будем использовать всю текстуру ландшафта.

Однако мы должны использовать другую перегруженную функцию `Draw`:

```
var sourceRectangle = new Rectangle(0, 0, _texture.Width, _
    _texture.Height);
var destinationRectangle = new Rectangle(x, y, _texture.Width,
    _texture.Height);
spriteBatch.Draw(_texture, destinationRectangle,
    sourceRectangle, Color.White);
```

Как видите, он имеет ту же ширину и высоту, что и наша игровая текстура объекта, и расположен в координатах `x` и `y`, которые отмечают расположение каждого блока ландшафта в нашей сетке, вычисленной по оси `Y` из `_texture`.

Высота до высоты окна просмотра и от 0 до ширины окна просмотра по оси X. Причина, по которой мы начинаем с `_texture.Height` по оси Y, заключается в том, чтобы создать дополнительный ряд ландшафта, который будет прокручиваться вниз в поле зрения.

Давайте заполним наши блоки текстур:

```
public override void Render(SpriteBatch spriteBatch)
{
    var viewport = spriteBatch.GraphicsDevice.Viewport;
    var sourceRectangle = new Rectangle(0, 0, _texture.Width,
                                        _texture.Height);

    for (int nbVertical = -1;
        nbVertical < viewport.Height / _texture.Height + 1;
        nbVertical++)
    {
        var y = (int) _position.Y + nbVertical *
               _texture.Height;
        for (int nbHorizontal = 0;
            nbHorizontal < viewport.Width / _texture.Width + 1;
            nbHorizontal++)
        {
            var x = (int) _position.X + nbHorizontal *
                   _texture.Width;
            var destRectangle = new Rectangle(x, y,
                                              _texture.Width, _texture.Height);
            spriteBatch.Draw(_texture, destRectangle,
                             sourceRectangle, Color.White);
        }
    }
    _position.Y = (int)(_position.Y + SCROLLING_SPEED) %
                  _texture.Height;
}
```

## ГЛАВА 6 INPUT

Сначала мы перебираем ось **Y** и вычисляем координаты всех блоков нашей текстуры по оси **Y**. Мы знаем, сколько вертикальных блоков нужно отрисовать, разделив высоту области просмотра на высоту нашей текстуры. Внешний цикл предназначен для перехода от -1 к общему количеству блоков, необходимых для размещения дополнительной строки, которую нам нужно нарисовать за пределами экрана. Умножив это число на высоту текстуры, мы получим точную координату **Y**, в которой нужно нарисовать ряд текстурных блоков. Но есть одна хитрость... мы добавляем значение **\_position.Y** игрового объекта к сумме. Эта позиция инициализируется 0, но по ходу игры она будет увеличиваться, и это приведет к тому, что все строки будут прорисовываться ниже при каждом проходе через функцию **Render()**.

Теперь, когда мы знаем координату **Y** нашей строки, нам нужно найти координату **X**. Точно так же мы узнаем, сколько блоков текстуры необходимо, разделив ширину области просмотра на ширину текстуры. Внутренний цикл **for** предназначен для перехода от 0 к общему количеству блоков, необходимых для заполнения экрана. Затем мы умножаем это число на ширину текстуры и добавляем **\_position.X** игрового объекта (которое всегда будет равно нулю в данный момент, но мы никогда не знаем, изменится ли оно в будущем), и мы получаем координату **X** каждого блока.

Вооружившись координатами **x** и **y**, мы теперь можем вычислить наши прямоугольники и заполнить экран блоками. Когда все это сделано, мы увеличиваем **\_position.Y** на **SCROLLING\_SPEED**, что приведет к смещению фона на экране вниз на следующем проходе рендеринга. Но есть еще один нюанс... Нет необходимости прокручивать текстуру вниз более чем на одну ширину. Когда мы достигаем этой точки, мы используем оператор по модулю для сброса значения **\_position.Y**, в результате чего анимация перезапускается с самого начала.

Хорошо, мы многое сделали, и теперь мы должны заменить наш игровой объект **oldterrain** реальным объектом **TerrainBackground**. Откройте класс **GameplayState** и замените эту строку в методе **LoadContent()**

```
AddGameObject(new SplashImage(LoadTexture(BackgroundTexture)));
```

with this line

```
AddGameObject(new TerrainBackground(LoadTexture(Background
Texture)));
```

Он будет использовать наш новый класс `TerrainBackground` и автоматически вечно прокручиваться.

Наконец, нам нужно расположить нашего бойца в нижней части этого экрана, прямо посередине. Измените часть `LoadContent()`, связанную с добавлением нашего игрового объекта истребителя, следующим кодом:

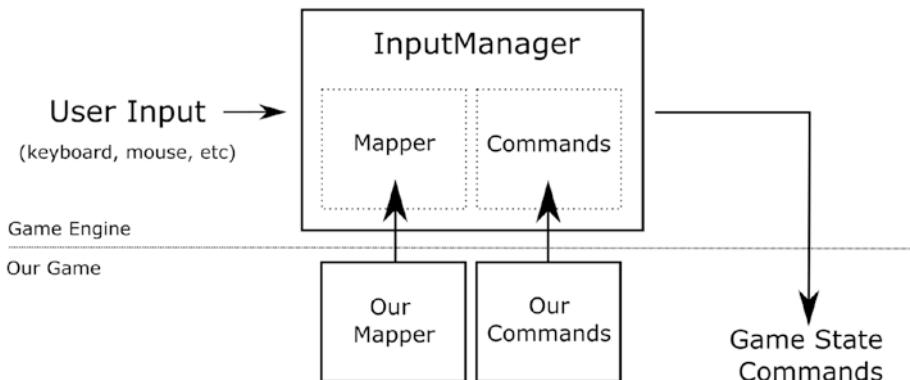
```
_playerSprite = new PlayerSprite(LoadTexture(PlayerFighter));
var playerXPos = _viewportWidth / 2 - _playerSprite.Width / 2;
var playerYPos = _viewportHeight - _playerSprite.Height - 30;
_playerSprite.Position = new Vector2(playerXPos, playerYPos);
```

Этот код как есть не будет компилироваться, потому что мы еще не изменили класс `BaseGameState`, чтобы сохранить ссылку на размеры области просмотра, которые использовались ранее. Вместо этого замените `_viewportWidth` и `_viewportHeight` на 1280 и 720 соответственно или просмотрите код, приведенный в конце главы, чтобы найти полное решение.

Теперь мы разместили истребитель на 30 пикселей выше нижней части области просмотра, прямо посередине экрана, и мы, наконец, готовы начать работу над нашим общим менеджером ввода.

## Создание универсального менеджера ввода

Цель нашего менеджера ввода — обрабатывать входные данные нашего игрока как можно лучше в наборе классов, которые можно повторно использовать в других играх с минимальными изменениями, чтобы заставить их работать из коробки. Мы выбрали шаблон, в котором игра получает ввод с клавиатуры, мыши или геймпада и преобразует ввод в команды, которые игра может вычислить. Каждый из наших классов состояния игры, например класс `GameplayState`, будет обрабатывать только входящие команды, а не напрямую обрабатывать ввод. См. [Рис. 6-11](#).



**Рис. 6-11.** Наша игра будет расширять классы *BaseInputMapper* и *BaseInputCommand* игрового движка и предоставлять расширения для диспетчера ввода.

Например, нажатие клавиши со стрелкой влево на клавиатуре во время игры сгенерирует команду *PlayerMoveLeft*, которая будет использована игровым состоянием и заставит спрайт игрока двигаться влево.

Не зная, какие команды используются нашей игрой или какое игровое состояние действует в данный момент, игровой движок сможет правильно преобразовать или отобразить ввод игрока в игровую команду. Идея состоит в том, что каждое игровое состояние, такое как *SplashState* или *GameState*, предоставит классу *InputManager* в нашем движке определенный класс *Mapper*, который знает, как сопоставлять входные данные с командами, которые важны для текущего состояния.

Здесь стоит отметить, что мы не следуем шаблону *PopularCommand*. Мы просто выбрали имя «*Command* - Команда», чтобы обозначить класс, который можно рассматривать как команду игрока на выполнение некоторого действия на основе пользовательского ввода.

`BaseInputCommand` — это пустой класс, но он используется как возвращаемый тип диспетчером ввода и классами `BaseInputCommand`. Мы расширим один класс картографа и один класс команд для каждого игрового состояния. Давайте посмотрим на класс `GameplayInputCommand`:

```
public class GameplayInputCommand : BaseInputCommand
{
    public class GameExit : GameplayInputCommand { }
    public class PlayerMoveLeft : GameplayInputCommand { }
    public class PlayerMoveRight : GameplayInputCommand { }
    public class PlayerShoots : GameplayInputCommand { }
}
```

Здесь мы видим несколько внутренних классов, унаследованных от `GamePlayInputCommand`, которые мы будем использовать точно так же, как мы используем перечисления, с тем главным отличием, что мы можем ограничивать типы команд нашими игровыми состояниями. Основное преимущество такого подхода заключается в том, что будет легче рассуждать о том, какие команды относятся к какому состоянию игры. По мере добавления новых игровых состояний, таких как меню, экраны опций, мини-игры и т. д., будет важно иметь простой способ поддерживать порядок. Но обо всем по порядку, взгляните на `GamePlayInputMapper`:

```
public class GameplayInputMapper : BaseInputMapper
{
    public override IEnumerable<BaseInputCommand>
        GetKeyboardState(KeyboardState state)
    {
        var commands = new List<GameplayInputCommand>();
        if (state.IsKeyDown(Keys.Escape))
        {
            commands.Add(new GameplayInputCommand.GameExit());
        }
    }
}
```

## ГЛАВА 6 INPUT

```
if (state.IsKeyDown(Keys.Left))
{
    commands.Add(new GameplayInputCommand.
        PlayerMoveLeft());
}

if (state.IsKeyDown(Keys.Right))
{
    commands.Add(new GameplayInputCommand.
        PlayerMoveRight());
}

if (state.IsKeyDown(Keys.Space))
{
    commands.Add(new GameplayInputCommand.
        PlayerShoots());
}

return commands;
}
```

}

В предыдущем преобразователе мы получаем пользовательский ввод из MonoGame, и для каждого интересующего нас ввода мы создаем команду, которую добавляем в список. Когда класс состояния игрового процесса получит эти команды, он сможет выполнять каждую из них и соответствующим образом управлять состоянием игры. Теперь нам нужно внедрить наш преобразователь и команды в менеджер ввода.

Начнем с добавления ссылки на диспетчер ввода в класс BaseGameState.

```
protected InputManager InputManager {get; set;}
```

и инициализируем его в конструкторе (здесь мы получаем размеры окна просмотра, которые мы обсуждали выше):

```
protected abstract void SetInputManager();  
  
public void Initialize(ContentManager contentMng, int  
    viewportWidth, int viewportHeight)  
{  
    _contentManager = contentMng;  
    _viewportHeight = viewportHeight;  
    _viewportWidth = viewportWidth;  
  
    SetInputManager();  
}
```

Здесь цель состоит в том, чтобы наши классы состояния игры, наследующие этот базовый класс, реализовывали функцию [SetInputManager\(\)](#). Класс [GameplayState](#) делает это следующим образом:

```
protected override void SetInputManager()  
{  
    InputManager = new InputManager(new GameplayInputMapper());  
}
```

Теперь мы можем видеть, как класс [GameplayState](#) создает настраиваемый менеджер ввода, который будет использовать наш собственный преобразователь ввода, который возвращает команды игрового процесса, на которые мы можем реагировать. Класс [SplashState](#) также создает свой собственный диспетчер ввода и имеет собственный набор команд, на которые он отвечает, а именно команду [GameSelect](#), которая срабатывает, когда игрок нажимает клавишу [Enter](#).

## ГЛАВА 6 INPUT

Все, что нам осталось сделать, это ответить на наши игровые команды. Давайте перепишем функцию `HandleInput()` класса `GameplayState`.

```
public override void HandleInput()
{
    InputManager.GetCommands(cmd =>
    {
        if (cmd is GameplayInputCommand.GameExit)
        {
            NotifyEvent(Events.GAME_QUIT);
        }

        if (cmd is GameplayInputCommand.PlayerMoveLeft)
        {
            _playerSprite.MoveLeft();
        }

        if (cmd is GameplayInputCommand.PlayerMoveRight)
        {
            _playerSprite.MoveRight();
        }
    });
}
```

Попробуйте! Вы должны увидеть прокручувающийся фон и иметь возможность перемещать бойца влево и вправо... даже за кадром! Подожди, этого не должно было случиться. Нам нужно, чтобы наш игрок оставался в области просмотра! Давайте добавим функцию, которая блокирует выход игрока за пределы экрана:

```
private void KeepPlayerInBounds()
{
    if (_playerSprite.Position.X < 0)
```

```
_playerSprite.Position = new Vector2(0, _playerSprite.  
Position.Y);  
}  
  
if (_playerSprite.Position.X > _viewportWidth -  
_playerSprite.Width)  
{  
    _playerSprite.Position = new Vector2(_viewportWidth -  
    _playerSprite.Width, _playerSprite.Position.Y);  
}  
  
if (_playerSprite.Position.Y < 0)  
{  
    _playerSprite.Position = new Vector2(_playerSprite.  
Position.X, 0);  
}  
  
if (_playerSprite.Position.Y > _viewportHeight -  
_playerSprite.Height)  
{  
    _playerSprite.Position = new Vector2(_playerSprite.  
Position.X, _viewportHeight - _playerSprite.Height);  
}  
}
```

Здесь мы проверяем положение объекта игрока и сбрасываем его, если он выходит за пределы. Если игрок переместится слишком далеко влево, его `Position.X` станет меньше нуля, поэтому мы просто корректируем его до нуля. Если он переместится слишком далеко вправо, спрайт начнет уходить за пределы экрана, когда его значение `Position.X` станет больше, чем ширина окна просмотра минус ширина спрайта. В этом случае мы сбрасываем и эту позицию. Мы также потратили некоторое время на то, чтобы убедиться, что игрок не может подняться или опуститься за пределы экрана, потому что мы предполагаем, что нам может понадобиться начать перемещать самолет вверх и вниз, когда мы начнем добавлять врагов в игру.

Теперь давайте вызовем нашу функцию после того, как игрок переместится:

```
_playerSprite.MoveLeft();  
KeepPlayerInBounds();
```

and

```
_playerSprite.MoveRight();  
KeepPlayerInBounds();
```

## Стрельба пулями

Возможно, вы заметили, что в командах, которые мы рассматривали до сих пор, была команда `PlayerShoots`. Действительно, наш истребитель не был бы очень интересен, если бы мог двигаться только влево и вправо. Итак, давайте стрелять пулями!

Мы добавили спрайт пули в конвейер игры. Вы можете найти в папке с кодами:

`monogame-mastery/blob/master/chapter-06/assets/png/bullet.png`.

Следуйте шагам, описанным в предыдущих главах, чтобы добавить пулью в конвейер игры и создать новый класс `BulletSprite` с функцией, позволяющей пуле перемещаться в верхнюю часть экрана.

```
public class BulletSprite : BaseGameObject  
{  
    private const float BULLET_SPEED = 10.0f;  
  
    public BulletSprite(Texture2D texture)  
    {  
        _texture = texture;  
    }
```

```
public void MoveUp()
{
    Position = new Vector2(Position.X, Position.Y -
        BULLET_SPEED);
}
}
```

Теперь нам нужно подготовить объекты-пули и стрелять ими, когда игрок нажимает пробел. Мы должны обновить функцию `LoadContent()` в классе `GameplayState`, чтобы загрузить текстуру, но не будем сразу создавать игровой объект. Вместо этого мы будем добавлять их в игру только тогда, когда игрок стреляет в них.

```
private const string BulletTexture = "bullet";
private Texture2D _bulletTexture;
private List<BulletSprite> _bulletList;
public override void LoadContent()
{
    // Остальной код для краткости опущен

    _bulletTexture = LoadTexture(BulletTexture);
    _bulletList = new List<BulletSprite>();
}
```

Мы добавляем сюда список пуль, чтобы отслеживать все пули, которые собираются заполнить экран. В конце концов мы должны увидеть, попала ли хоть одна пуля во врага, поэтому нам нужен простой способ проверить все наши пули. Список подходит для этой цели. Когда пули будут созданы, мы добавим их в список игровых объектов (чтобы они отображались игровым движком) и в наш список пуль.

## ГЛАВА 6 INPUT

Нам также нужно отслеживать игровое время, потому что мы не хотим, чтобы игрок мог удерживать пробел и стрелять бесконечным потоком пуль. Мы хотим, чтобы игра была немного сложной, и простой способ сделать это — замедлить. снизить скорость пуль, скажем... 5 в секунду? Вот как это будет работать: когда игрок выпускает залп пуль, мы отмечаем, что игрок в данный момент стреляет, и запоминаем текущее время игры. Если игрок снова выстрелит, удерживая клавишу пробела или нажимая ее слишком быстро, мы не позволим игроку стрелять, если с момента последнего успешного выстрела пуль прошло 0,2 секунды. Итак, давайте обновим код, чтобы отслеживать игровое время:

```
private bool _isShooting;
private TimeSpan _lastShotAt;
public override void HandleInput(GameTime gameTime)
{
    // ...
    if (cmd is GameplayInputCommand.PlayerShoots)
    {
        Shoot(gameTime);
    }
}

private void Shoot(GameTime gameTime)
{
    if (!_isShooting)
    {
        CreateBullets();
        _isShooting = true;
        _lastShotAt = gameTime.TotalGameTime;
    }
}
```

```
private void CreateBullets()
{
    var bulletSpriteLeft = new BulletSprite(_bulletTexture);
    var bulletSpriteRight = new BulletSprite(_bulletTexture);

    // Position bullets around the fighter's nose when they get
    // fired
    var bulletY = _playerSprite.Position.Y + 30;
    var bulletLeftX = _playerSprite.Position.X +
        _playerSprite.Width / 2 - 40;
    var bulletRightX = _playerSprite.Position.X +
        _playerSprite.Width / 2 + 10;

    bulletSpriteLeft.Position = new Vector2(bulletLeftX, bulletY);
    bulletSpriteRight.Position = new Vector2(bulletRightX,
        bulletY);

    _bulletList.Add(bulletSpriteLeft);
    _bulletList.Add(bulletSpriteRight);

    AddGameObject(bulletSpriteLeft);
    AddGameObject(bulletSpriteRight);
}
```

Now we need to make our bullets move up on their own. We'll do that by adding an `Update()` function that will get called from the main game loop. Let's add it in our base class first:

```
public abstract class BaseGameState
{
    public virtual void Update(GameTime gameTime) { }
}
```

## ГЛАВА 6 INPUT

Добавьте эту строку в функцию `Update()` `MainGame`, чтобы вызвать функцию `Update()` класса `currentstate`:

```
_currentGameState.Update(gameTime);
```

Наконец, давайте реализуем функцию `Update()` в нашем классе `GameplayState`, чтобы все маркеры в нашем списке маркеров перемещались вверх:

```
public override void Update(GameTime gameTime)
{
    foreach (var bullet in _bulletList)
    {
        bullet.MoveUp();
    }

    //Нельзя стрелять чаще, чем каждые 0,2 секунды. Если это было
    //больше, разрешите съемку снова
    if (_lastShotAt != null &&
        gameTime.TotalGameTime - _lastShotAt >
        TimeSpan.FromSeconds(0.2))
    {
        _isShooting = false;
    }
}
```

Написание игр — это очень интересно, но в конце концов возникает множество утомительных деталей, о которых нужно позаботиться. Код, который мы рассмотрели, создает множество пуль, летящих вверх по экрану. Единственная проблема заключается в том, что когда пули оказываются за кадром, их все еще отслеживают, и их положение продолжает увеличиваться. Эти пули занимают немного памяти, и хотя компьютеры работают быстро, итерация по нашему списку пуль десятки раз в секунду для обновления их положения в конечном итоге начнет замедлять

игру. Чтобы этого не произошло, нам нужно убирать пули из игры после того, как они исчезли с экрана. Переместив наши пули вверх, давайте очистим их в функции `sameUpdate()`:

```
// Избавляемся от пуль, вышедших из поля зрения
var newBulletList = new List<BulletSprite>();foreach (var bullet
in _bulletList)
{
    var bulletStillOnScreen = bullet.Position.Y > -30;
    if (bulletStillOnScreen)
    {
        newBulletList.Add(bullet);
    }
    else
    {
        RemoveGameObject(bullet);
    }
}
_bulletList = newBulletList;
```

## Резюме

Мы проделали большую работу в этой главе, и наша игра начинает выглядеть как настоящая! Мы добавили прокручивающийся фон после изучения того, как другие игры делают то же самое. Мы также добавили универсальный диспетчер ввода, который можно использовать повторно, просто создавая списки игровых команд и сопоставляя пользовательский ввод с этими командами. Мы также получили небольшое представление об управлении памятью и о том, как нам нужно поддерживать плавную работу игры.

## ГЛАВА 6 INPUT

Окончательную версию кода игры для этой главы можно найти здесь:  
<https://github.com/Apress/monogame-mastery/tree/master/chapter-06/end>.

Не стесняйтесь экспериментировать. В следующей главе мы добавим музыкальную фоновую дорожку и звуковые эффекты, когда игрок стреляет пулями и когда пули поражают врагов. О, это, вероятно, означает, что мы также начнем добавлять врагов в нашу игру!

## ГЛАВА 7

# Аудио

Игра без звука — это как салат без заправки. Это все еще съедобно, но это будет не очень приятно. Точно так же, хотя в нашу игру можно было бы играть без музыки и звуковых эффектов, игрокам будет сложно погрузиться в нее и эмоционально вовлечься. Подумайте обо всех эпических битвах с боссами, которые существуют в большинстве видеоигр. Помимо удивительной механики боссов, все они имели саундтрек, который разжигал инстинкты борьбы или бегства игрока.

Музыка в игре существует для того, чтобы создавать эмоции и задавать тон определенному уровню или сценарию, как в фильмах, а звуковые эффекты нужны для того, чтобы добавить в игру немного реализма, но не быть чрезмерным.

В этой главе вы должны

- Рефакторинг движка, чтобы сделать его более многоразовым.
- Создать звуковой менеджер и добавить его в движок
- Добавлять треки и звуковые эффекты в звуковой менеджер
- Запускать звуковые эффекты через игровые события.

## Рефакторинг движка

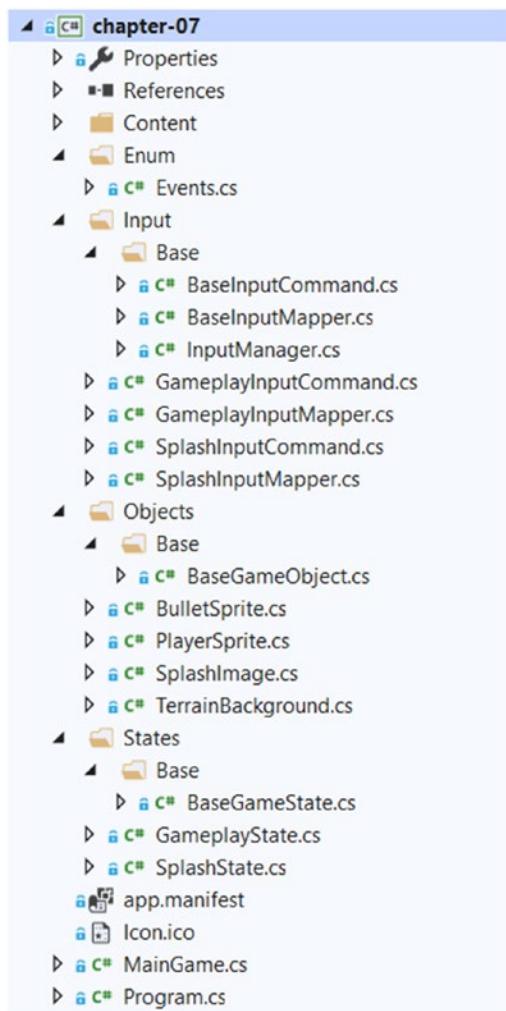
Когда мы начали писать код для этой игры, мы не были слишком сосредоточены на том, чтобы сделать код движка простым для повторного использования в будущих проектах. Несмотря на то, что было разделение ответственности, и мы четко обсудили, какая часть кода будет частью кода движка,

## ГЛАВА 7 AUDIO

а какая часть будет специфичной для игры, все наши файлы были размещены и смешаны вместе. Если бы мы захотели взять код движка из начала этой главы и повторно использовать его в другом проекте, нам пришлось бы выбирать, какие файлы нужно скопировать в новый проект, и этот процесс не идеален.

В лучшем случае вся логика нашего движка будет полностью отделена в библиотеке, которую мы сможем импортировать в другие проекты. Наш игровой движок еще не готов, поэтому мы можем немного отложить этот шаг, пока не закончим написание его кода. Одна вещь, которую мы можем сделать прямо сейчас, это начать организовывать логику движка, чтобы еще больше отделить ее от кода нашей игры.

Давайте взглянем на Рис. 7-1 и текущую организацию нашегокода. Код, который мы рассмотрим и изменим в этой главе, находится здесь: <https://github.com/Apress/monogame-mastery/tree/master/chapter-07/start>. (Находится в папке с кодами)



*Рис. 7-1. Организация кода в начале этой главы*

До сих пор большая часть кода нашего движка была базовыми классами, которые код игры может наследовать и реализовывать. Однако это изменилось, когда мы добавили менеджер ввода в Главе 6. Поскольку мы готовимся добавить звуковой менеджер, нам нужно подумать о том, как мы можем реорганизовать код. Одна из наших проблем

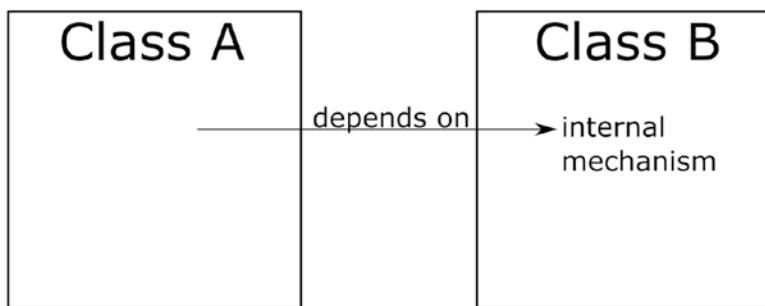
## ГЛАВА 7    AUDIO

заключается в том, что `MainGame.cs` содержит очень общую логику, которая могла бы быть частью игрового движка, если бы не эти строки кода:

```
protected override void LoadContent()
{
    // Создаем новый SpriteBatch, который можно использовать для
    // отрисовки текстур.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    SwitchGameState(new SplashState());
}
```

Все в этом файле ссылается на код движка или код MonoGame, за исключением функции `LoadContent()`, которая должна знать, какой класс `GameState` ей нужно запустить в первую очередь. Одна проблема заключается в том, что в настоящее время он запрограммирован для запуска `SplashState`, который во многом является модулем, специфичным для игры. Если бы вы создавали другую игру, используя тот же движок, скорее всего, `SplashState` был бы совсем другим.

Одним из принципов хорошего проектирования программы является низкая связанность и высокая согласованность между модулями кода. Есть много вещей, которые могут увеличить связь между классами и файлами. Например, на рис. 7-2 у нас есть объект A, обращающийся напрямую к внутренним переменным другого объекта B, что немедленно создало бы зависимость вызывающего объекта A от того, как реализуется внутренний класс B, что усложнило бы изменение класса B в будущем, не вызывая проблем в классе A.



*Рис. 7-2. Объекты класса А зависят от внутренних деталей класса Б*

Между модулями всегда будет связь, но цель должна состоять в том, чтобы уменьшить связь между модулями, относящимися к разным областям интереса. Существует меньше проблем, когда код игрового движка связан сам с собой, чем если бы он был напрямую связан с игрой. Вот где на сцену выходит сплоченность. Код с высокой связностью в своих модулях и низкой связанностью с другими модулями будет легче поддерживать и повторно использовать.

Давайте сделаем `MainGame` более универсальной. Вместо того, чтобы жестко кодировать наш первый класс `GameState` для начала в методе `LoadContent()`, мы предоставим классу эту информацию через параметр конструктора.

Точно так же вместо того, чтобы иметь в классе константы для ширины, высоты и соотношения сторон разрешения экрана, мы передадим в класс желаемую ширину и высоту и вычислим соотношение сторон внутри конструктора.

```

private int _DesignedResolutionWidth;
private int _DesignedResolutionHeight;
private float _designedResolutionAspectRatio;

private BaseGameState _firstGameState;
  
```

## ГЛАВА 7    AUDIO

```
public MainGame(int width, int height, BaseGameState  
firstGameState)  
{  
    Content.RootDirectory = "Content";  
    graphics = new GraphicsDeviceManager(this)  
    {  
        PreferredBackBufferWidth = width,  
        PreferredBackBufferHeight = height,  
        IsFullScreen = false  
    };  
  
    _firstGameState = firstGameState;  
    _DesingedResolutinWidth = width;  
    _DesingedResolutionHeight = height;  
    _designedResolutionAspectRatio = width / (float)height;  
}
```

Теперь [LoadContent\(\)](#) может использовать переменную `_firstGameState` и запускать игру в этом состоянии при выполнении:

```
protected override void LoadContent()  
{  
    // Создаем новый SpriteBatch, который можно использовать для  
    // отрисовки текстур  
    spriteBatch = new SpriteBatch(GraphicsDevice);  
    SwitchGameState(_firstGameState);  
}
```

С этими изменениями наш файл [MainGame.cs](#) теперь ссылается только на код движка и код MonoGame и готов к перемещению в движок!

Ответственность за установку разрешения и начального состояния игры была перенесена в файл Program.cs, поэтому мы можем установить здесь значения WIDTH и HEIGHT и создать наш экземпляр [MainGame](#) с этими константами и экземпляром класса [SplashState](#):

```
private const int WIDTH = 1280;
private const int HEIGHT = 720;
static void Main()
{
    using (var game = new MainGame(WIDTH, HEIGHT, new
        SplashState()))
        game.Run();
}
```

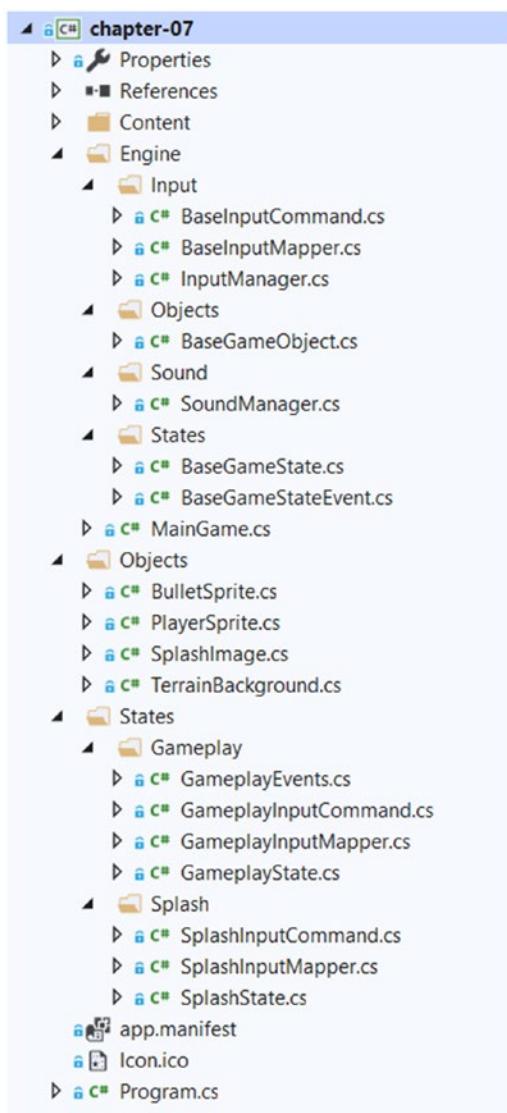
## Организация кода

Существуют разные подходы к организации кода. Существует множество проектов, которые следуют установленным шаблонам, таким как модель/представление/представление-модель (MVVM) или модель/представление/контроллер (MVC), и группируют все контроллеры в каталоге с именем «контроллеры», модели в каталоге с именем «модели», и так далее. Это упрощает поиск кода в проекте. Если программист ищет конкретное представление, ему просто нужно перейти в каталог представления и посмотреть там.

Другая школа мысли состоит в том, чтобы организовать код по функциям, при этом каждый файл кода, связанный с функцией, такой как звук или ввод, должен находиться в одном и том же каталоге. Это школа мысли, которую мы будем использовать здесь при организации кода. Весь звук, ввод, состояние игры и код игровых объектов будут размещены вместе. В то же время специфичный для игры код будет организован по тому, к какому игровому состоянию он принадлежит. Наши сопоставители команд, игровые события и игровые команды будут совмещены по состоянию игры.

На Рис. 7-3 мы видим, что код движка был перемещен в каталог [Engine](#), а код, специфичный для игры, организован по состоянию игры. Теперь мы готовы добавить звук в нашу игру в файле [SoundManager.cs](#), расположенном в папке [Sound](#) движка.

## ГЛАВА 7    AUDIO



*Рис. 7-3. Результатирующее дерево кода после нашей реорганизации*

# Аудио

MonoGame предоставляет нам простой и кроссплатформенный API, который поддерживает форматы файлов mp3, ogg и wav из коробки в инструменте contentpipeline. Мы добавили еще несколько ресурсов в каталог ресурсов главы 7. Вот wav-файлы для фонового саундтрека, который будет воспроизводиться в игре:

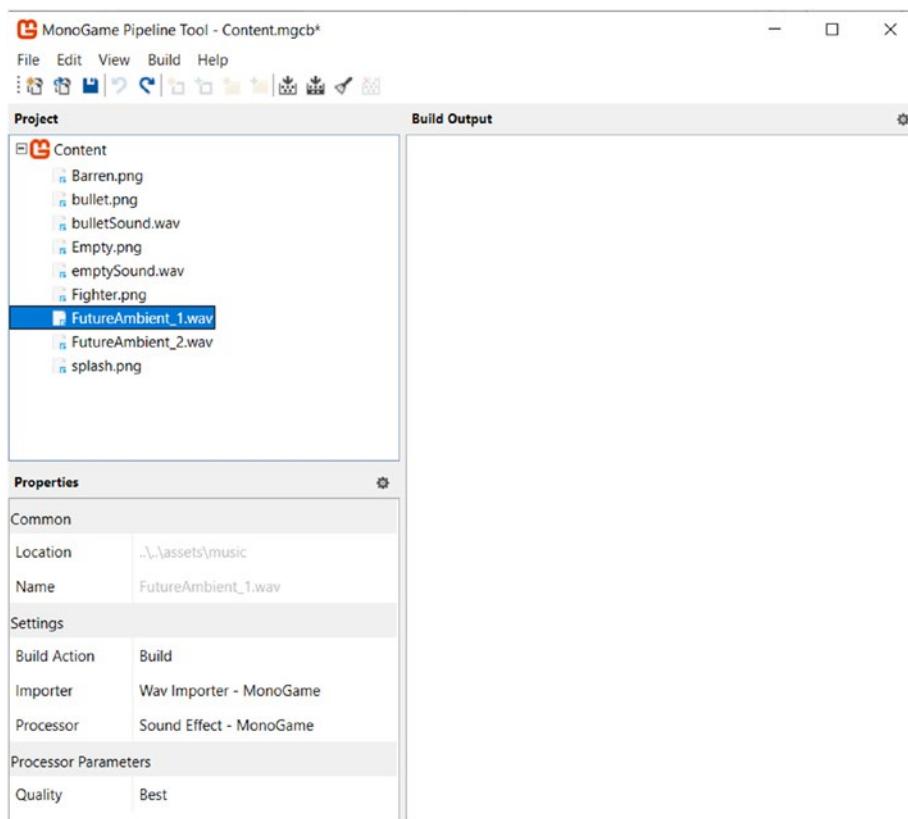
- FutureAmbient \_1.wav
- FutureAmbient \_2.wav
- FutureAmbient \_3.wav
- FutureAmbient \_4.wav

Мы также добавили звуковой эффект пули, который мы будем использовать, когда игрок стреляет, и пустой wav-файл, который будет использоваться, когда движок не загружается.

- bullet.wav
- empty.wav

Откройте [Content Pipeline Tool](#) и добавьте все шесть файлов .wav в список активов, как показано на Рис. 7-4. Мы решили переименовать звуки пули и пустых звуков в конвейере в `bulletSound` и `emptySound`, чтобы их было проще отличить от одноименных изображений.

## ГЛАВА 7     AUDIO



*Рис. 7-4. Добавление звуков в конвейер*

Убедитесь, что звуки для наших новых файлов wav используют Wav Importer и настроены как звуковые эффекты. В поле свойств в нижней части инструмента конвейера установите для импортера значение «Wav Importer — MonoGame», используя раскрывающийся список, и таким же образом установите для процессора значение «Wav Importer – MonoGame - Звуковой эффект — MonoGame». Инструмент конвейера также предлагает средства импорта файлов mp3 и ogg, но будьте осторожны при использовании файлов mp3, поскольку это проприетарный формат, и если ваша игра популярна, вас могут попросить заплатить лицензионные отчисления. Инструмент конвейера также предлагает звуковой процессор, но мы вскоре обсудим, почему мы решили не использовать его для нашей фоновой музыки.

## Воспроизведение саундтрека

Если мы не создаем игровую сцену без какой-либо фоновой музыки для эффекта, который она оказывает на игрока, нам понадобится какой-нибудь саундтрек. Мы могли бы сочинить свой собственный саундтрек, мы могли бы нанять композитора, чтобы он создал для нас несколько треков, мы могли бы купить песни, которые уже существуют, или мы могли бы найти бесплатную музыку, которую мы могли бы легально использовать и распространять. Если вы решите использовать музыку или звуки с лицензией CreativeCommons (CC), дважды проверьте, какие права у вас есть, поскольку создателям доступно несколько вариантов. Этот последний вариант подходит в нашем случае, так как мы создаем простую игру для демонстрационных целей. Тем не менее, у этого есть один недостаток: мы рискуем иметь ту же музыку, что и в другой игре, и это было бы неловко. Мы рекомендуем посетить бесплатный музыкальный архив в Интернете, расположенный здесь: [www.freemusicarchive.org/](http://www.freemusicarchive.org/).

Однако мы уже выбрали несколько файлов .wav, которые хотим использовать в нашей игре, и поместили их в каталог assets\music этой главы.

Есть два способа воспроизведения фоновой музыки с помощью MonoGame: MediaPlayer или использование звуковых эффектов. Медиаплеер был добавлен как способ для игроков Xbox добавлять свою собственную фоновую музыку в игру во время игры, и он отлично подходит для этой цели. Он даже воспроизводит песню с диска в потоковом режиме, а не загружает весь файл в память. Чтобы использовать медиаплеер, нам нужно добавить наши треки в инструмент конвейера контента и выбрать звуковой процессор. Сам класс очень прост в использовании. Добавление этой функции в классы состояния базовой игры позволит игре загрузить песню:

```
using Microsoft.Xna.Framework.Media;
protected Song LoadContent(string songName)
{
    return _contentManager.Load<Song>(songName);
}
```

## ГЛАВА 7    AUDIO

Затем мы могли бы воспроизвести эту песню в наших классах состояния игры:

```
var song = LoadSong("SomeSoundtrack");
MediaPlayer.Play(song);
```

Медиаплеер позволяет разработчикам игр ставить песни в очередь, воспроизводить следующую или предыдущую песню, приостанавливать и возобновлять песни, а также регулировать громкость. Он ведет себя так же, как... медиаплеер!

Однако, к сожалению, медиаплеер страдает одним существенным недостатком: он добавляет небольшую, почти полсекундную паузу между песнями, что исключает возможность зацикливания коротких сэмплов песен для создания, казалось бы, бесконечной дорожки. Чтобы облегчить это, мы могли бы создать очень длинную песню, которая в конечном итоге зациклилась бы. Небольшое молчание все равно будет заметно, но это будет происходить не так часто, что делает его менее проблематичным. Тем не менее, это сделало бы нашу игру менее отполированной.

В нашей игре будут использоваться короткие звуковые сэмплы, которые мы будем повторять. У нас есть четыре таких семпла в нашем каталоге ресурсов, и их можно микшировать вместе в любом порядке. При добавлении в инструмент конвейера контента они должны использовать процессор SoundEffect, как на Рис. 7-4.

Давайте добавим в наш класс `BaseGameState` метод, позволяющий загружать эти образцы:

```
protected SoundEffect LoadSound(string soundName)
{
    return _contentManager.Load<SoundEffect>(soundName);
}
```

Затем звуковой эффект возврата можно использовать в состояниях игры следующим образом:

```
var sound = LoadSound("someSoundEffect");
sound.CreateInstance();
sound.Play();
```

После этого звук будет слышен во время игры. Но он не будет повторяться, что нам нужно для звуковых эффектов, таких как стрельба пульами, но не для

наших минусовок. Обратите также внимание на то, что мы вызываем [CreateInstance\(\)](#) для загруженных объектов звукового эффекта, который возвращает объект [SoundEffectInstance](#), дающий больше возможностей для управления звуком. Объект [SoundEffect](#) можно воспроизвести, и это почти все, но объект [SoundEffectInstance](#) можно воспроизвести, приостановить, возобновить и зациклить. Мы также можем изменить громкость, панорамирование и высоту звукового эффекта во время его воспроизведения.

Чтобы зациклить наш звуковой образец, нам просто нужно установить для свойства [IsLooped](#) значение `true`:

```
sound.IsLooped = true;  
sound.Play();
```

and will play forever until it is stopped with  
`sound.Stop();`

Однако мы не хотим постоянно воспроизводить один и тот же звуковой образец, что сделало бы нашу игру немного монотонной. Вместо этого мы хотели бы чередовать наши сэмплы и воспроизводить следующий каждый раз, когда текущий сэмпл перестает воспроизводиться. К счастью, класс [SoundEffectInstance](#) предлагает нам возможность управлять состоянием сэмпла, считывая свойство [State](#) его объектов, которое может иметь три различных значения: `Paused`, `Playing` или `Stopped`.

Наша стратегия чередования сэмплов песен теперь может обретать форму. Когда мы инициализируем состояние игры, мы загружаем образцы саундтреков и добавляем ссылки на них в `List<SoundEffectInstance>`. Когда метод `Update()` вызывается классом [MainGame](#), мы начинаем проигрывать первый образец, если в данный момент музыка не воспроизводится. В противном случае, если текущая дорожка остановлена, мы перейдем к следующей дорожке и начнем ее воспроизведение. Этот процесс будет повторяться на протяжении всего текущего игрового состояния.

Мы также должны помнить, что мы создаем игровой движок, поэтому ответственность за отслеживание наших сэмплов и воспроизведение их должна лежать на самом движке. Это очень базовая функциональность для игры, и мы бы очень хотели повторно использовать этот код в наших будущих проектах.

## ГЛАВА 7 AUDIO

Создадим класс `SoundManager` в директории `Engine\Sound`:

```
public class SoundManager
{
    private int _soundtrackIndex = -1;
    private List<SoundEffectInstance> _soundtracks =
        new List<SoundEffectInstance>();
    public void SetSoundtrack(List<SoundEffectInstance> tracks)
    {
        _soundtracks = tracks;
        _soundtrackIndex = _soundtracks.Count - 1;
    }

    public void PlaySoundtrack()
    {
        var nbTracks = _soundtracks.Count;

        if (nbTracks <= 0)
        {
            return;
        }

        var currentTrack = _soundtracks[_soundtrackIndex];
        var nextTrack = _soundtracks[(_soundtrackIndex + 1) %
            nbTracks];

        if (currentTrack.State == SoundState.Stopped)
        {
            nextTrack.Play();
            _soundtrackIndex++;

            if (_soundtrackIndex >= _soundtracks.Count)
            {
                _soundtrackIndex = 0;
            }
        }
    }
}
```

```

        }
    }
}

```

Когда мы создаем звуковой менеджер, нам нужно дать ему список дорожек, по которым он будет вращаться. Затем он устанавливает `_soundtrackIndex` в последний элемент в списке, потому что, когда мы говорим ему воспроизвести наши треки, он сначала будет двигаться вперед в списке, и это заставит его вернуться к первому экземпляру `SoundEffectInstance` нашего списка.

Вызов `PlaySoundTrack()` — вот где происходит волшебство. Во-первых, если в нашем плейлисте нет треков, то делать нечего и метод существует. В противном случае мы смотрим на текущий трек и находим наш следующий трек в списке. Если текущая дорожка остановлена, мы проигрываем следующую дорожку и увеличиваем наш индекс. Наконец, если индекс ушел из списка, мы сбрасываем его на первый элемент.

Теперь мы можем изменить наш класс `GamePlayState`, чтобы использовать менеджер звука. Сначала в `BaseGameState` добавьте следующие защищенные переменные, чтобы все наши классы состояния игры имели доступ к диспетчеру звука:

```
protected SoundManager _soundManager = new SoundManager();
```

Then, in the `LoadContent()` method, add the following lines at the end:

```
var track1 = LoadSound("FutureAmbient_1").CreateInstance();
var track2 = LoadSound("FutureAmbient_2").CreateInstance();
_soundManager.SetSoundtrack(new List<SoundEffectInstance>() {
    track1, track2});
```

Поскольку все наши игровые состояния будут пытаться зацикливаться на музыкальном плейлисте, в идеале мы должны обрабатывать вызов диспетчера звука для зацикливания на следующей песне в классе `BaseGameState` каждый раз, когда вызывается функция `Update()`. Но есть одна загвоздка... Функция `Update()` в базовом классе является виртуальной, а это означает, что вызывается только функция `Update()` в дочерних классах, таких как `GameplayState`. Мы могли бы добавить базу.

`Update(gameTime)` во всех наших классах дочерних состояний, но это противоречит цели наличия движка, а также является плохим дизайном. Наши дочерние классы должны знать, что им нужно вызывать метод базового класса с тем же именем, если они хотят, чтобы музыка продолжала играть. Дочерние классы не должны знать таких подробностей реализации, поэтому нам нужна другая стратегия. Вместо этого мы создаем новый абстрактный метод для `BaseGameState`, который будет вызываться функцией `Update()`:

```
public abstract void UpdateGameState(GameTime gameTime);
public void Update(GameTime gameTime)
{
    UpdateGameState(gameTime);
    _soundManager.PlaySoundtrack();
}
```

Обновление по-прежнему постоянно вызывается `MainGame.cs`. Теперь он также взаимодействует с менеджером звука, чтобы поддерживать воспроизведение фоновой музыки, и просит каждый дочерний класс, такой как `GameplayState` и `SplashState`, реализовать свои собственные функции `UpdateGameState`, чтобы у них была возможность обновлять состояние игры с течением времени.

Еще одна интересная особенность этого дизайна заключается в том, что каждый класс состояния игры может иметь свои собственные звуковые дорожки и собственное вращение по циклам. Если мы решим добавить битву с боссом, это будет новое игровое состояние, и вы можете спорить, что его саундтрек будет... эпическим!

## Звуковые эффекты

Теперь, когда у нас играет фоновая музыка, пришло время добавить звуковые эффекты, когда наш боец стреляет пулями.

Звуковые эффекты имеют те же ограничения авторского права, что и саундтреки, поэтому мы также должны быть осторожны, чтобы найти звуковые записи, которые мы можем использовать бесплатно, или записать свои собственные. В данном случае мы решили скачать звук пули с <http://freesound.org>, веб-сайта, который специализируется на...

ну и бесплатные звуки. WAV-файл пули, который мы добавили в конвейер выше, использует лицензию Creative Commons 0, которая позволяет нам делать со звуком все, что мы хотим. Мы также немного отредактировали звук, чтобы сократить его. Вы можете делать такие вещи, используя Audacity ([www.audacityteam.org/](http://www.audacityteam.org/)), бесплатный аудиоредактор с открытым исходным кодом.

Вернемся к нашему проекту, поскольку у нас уже есть менеджер звука, мы собираемся обновить его, чтобы он обрабатывал звуковые эффекты, запуская игровые события, на которые он может реагировать. Когда игровое состояние загружено, оно должно будет загрузить все необходимые звуки в банк звуков и связать каждый звук с типом события. Затем состояние игры будет прослушивать игровые события и направлять их диспетчеру звука, который сможет отреагировать, воспроизведя соответствующий звуковой образец. Нам понадобятся некоторые игровые события, чтобы реагировать на них, так что давайте начнем с этого.

У нас уже есть перечисление `Events`, которое мы могли бы использовать. Игра уже использует его и запускает событие `GAME_QUIT`, когда игрок нажимает клавишу `ESC`, но с ним есть одна проблема. Если мы добавим в это перечисление еще один элемент с именем `PLAYER_SHOOTS`, то перечисление станет специфичным для игры и не может использоваться в движке, потому что в нашей следующей игре игрок может вообще не стрелять. Насколько нам известно, это может быть игра в шахматы. Таким образом, если это перечисление нельзя использовать в движке, значит, наш звуковой менеджер, являющийся частью движка, не может его использовать. И снова нам нужно провести рефакторинг!

В движке добавьте класс `BaseGameStateEvent` в каталог `Engine\States`:

```
public class BaseGameStateEvent
{
    public class GameQuit : BaseGameStateEvent { }
}
```

Это будет базовый класс для всех наших событий состояния игры, и мы собираемся следовать тому же шаблону, который мы использовали для наших команд ввода, где все общедоступные классы явно принадлежат игровому состоянию как часть их типа.

## ГЛАВА 7 AUDIO

Здесь единственное обычное событие, о котором мы можем думать, которое будет частью каждой будущей игры, которую мы создадим, — это событие, которое позволит кому-то выйти из игры. Теперь мы можем избавиться от старого перечисления Events и обновить эти строки функции HandleInput() в GameplayState.

```
if (cmd is GameplayInputCommand.GameExit)
{
    NotifyEvent(Events.GAME_QUIT);
}
```

С этим

```
if (cmd is GameplayInputCommand.GameExit)
{
    NotifyEvent(new BaseGameStateEvent.GameQuit());
}
```

и измените метод NotifyEvent() и все методы, которые он вызывает, чтобы они принимали в качестве параметра наше новое **BaseGameStateEvent** вместо **Eventsenum**.

Теперь мы готовы обновить наш звуковой менеджер. Откройте класс SoundManager и добавьте в него следующий код. Во-первых, нам понадобится словарь для сопоставления типов событий со звуковыми эффектами:

```
private Dictionary<Type, SoundEffect> _soundBank = new
Dictionary<Type, SoundEffect>();
```

Нам также нужен метод, позволяющий игровому состоянию загружать звуки в звуковой банк:

```
public void RegisterSound(BaseGameStateEvent gameEvent,
SoundEffect sound)
{
    _soundBank.Add(gameEvent.GetType(), sound);
}
```

Наконец, мы переходим к методу, позволяющему состоянию игры запускать воспроизведение звука на основе игрового события:

```
public void OnNotify(BaseGameStateEvent gameEvent)
{
    if (_soundBank.ContainsKey(gameEvent.GetType()))
    {
        var sound = _soundBank[gameEvent.GetType()];
        sound.Play();
    }
}
```

Теперь давайте создадим пару событий для нашего [GameplayState](#). Создайте новый класс в каталоге [States\Gameplay](#) с именем [GamePlayEvents](#):

```
public class GamePlayEvents : BaseGameStateEvent
{
    public class PlayerShoots : GamePlayEvents { }
}
```

И обновите класс [GameplayState](#), чтобы это событие запускалось, когда игрок нажимает пробел, чтобы стрелять пулями. В приватном методе [Shoot\(\)](#) внутри блока кода, создающего игровые объекты пули, [callNotifyEvent\(\)](#):

```
if (!_isShooting)
{
    CreateBullets();
    _isShooting = true;
    _lastShotAt = gameTime.TotalGameTime;

    NotifyEvent(new GamePlayEvents.PlayerShoots());
}
```

## ГЛАВА 7 AUDIO

Обновите `NotifyEvent` для вызова менеджера звука:

```
protected void NotifyEvent(BaseGameStateEvent gameEvent)
{
    OnEventNotification?.Invoke(this, gameEvent);
    foreach (var gameObject in _gameObjects)
    {
        gameObject.OnNotify(gameEvent);
    }
    _soundManager.OnNotify(gameEvent);
}
```

Наконец, обновите метод `LoadContent()` состояния игрового процесса, чтобы загрузить звуковой эффект пули и добавить его в банк звуков:

```
public override void LoadContent()
{
    // Код опущен для краткости
    var bulletSound = LoadSound("bulletSound");
    _soundManager.RegisterSound(new GameplayEvents.PlayerShoots(), bulletSound);
}
```

Обратите внимание, что мы не вызываем `CreateInstance()` для этого звукового эффекта после его загрузки, потому что нам на самом деле не нужно приостанавливать или останавливать звуковые эффекты в это время.

## Резюме

И вот оно. В нашей игре есть циклический саундтрек, уникальный для каждого игрового состояния, и каждый саундтрек может состоять из множества звуковых сэмплов, заказанных по желанию разработчика игры. Мы также добавили звуковой менеджер в наш игровой движок и подключили звуковые эффекты к игровым событиям. Наконец, мы реорганизовали довольно много кода и реорганизовали логику, которые не имели такого большого смысла, как в начале, что является фактом жизни для разработчиков программного обеспечения.

Окончательную версию кода для этой главы можно найти здесь:

<https://github.com/Apress/monogame-mastery/tree/master/chapter-07/end>.

В следующей главе мы добавим в игру эффекты частиц. Наш истребитель сможет стрелять ракетами, оставляющими за собой след.

## ГЛАВА 8

# Частицы

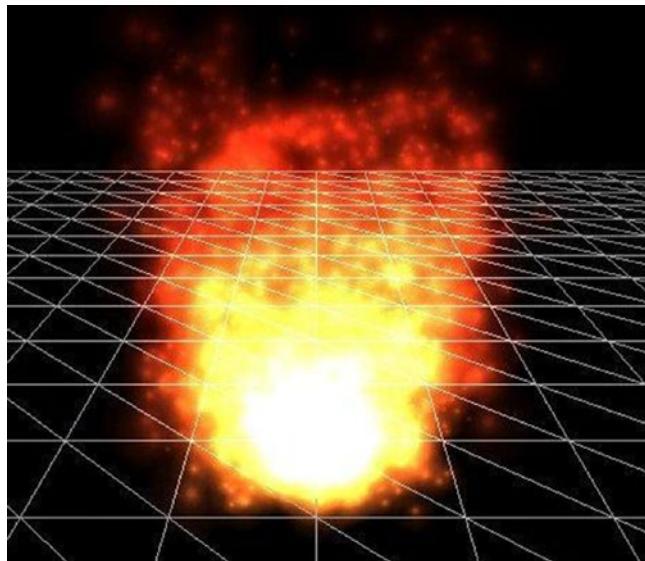
В наши дни трудно найти игру без частиц. Даже простые 2D-игры вроде нашей теперь используют частицы для украшения визуальных эффектов на экране — от мерцающего факела в коридоре до фонтана посреди потрясающего сада. Используя такие системы, разработчик может даже создать стаю птиц или косяк рыб, где каждая частица будет птицей или рыбой.

Идея движка частиц состоит в том, чтобы генерировать сотни или даже тысячи игровых объектов частиц, которые имеют определенные атрибуты, такие как цвет, непрозрачность или масштаб, и эти атрибуты меняются в течение жизни частицы. Чаще всего отдельная частица живет всего несколько секунд и медленно исчезает из поля зрения по мере того, как ее непрозрачность уменьшается до точки, когда она становится полностью прозрачной и невидимой. Каждый раз, когда вызывается игровая функция обновления, несколько десятков частиц создаются в определенном месте на экране, и им дается направление, в котором они летят с определенной скоростью. Как и другие атрибуты частиц, это направление может меняться со временем, направляя частицу на постоянно меняющийся путь.

Например, искры от фейерверков будут притягиваться вниз под действием силы тяжести, даже если они вылетают из центра взрыва в небо.

Посмотрите на Рис.8-1. Частицы вылетают из ярко-белого центра огня по кругу, а затем медленно подхватываются восходящей силой. Каждая частица представляет собой полупрозрачный круг, который сначала белый, затем становится желтым, а затем красным по мере удаления от центра. Частицы также имеют разные размеры, и чем дальше они поднимаются, тем более прозрачными они становятся<sup>1</sup>.

<sup>1</sup>Image source:[https://en.wikipedia.org/wiki/Particle\\_system#/media/File:Particle\\_sys\\_fire.jpg](https://en.wikipedia.org/wiki/Particle_system#/media/File:Particle_sys_fire.jpg)



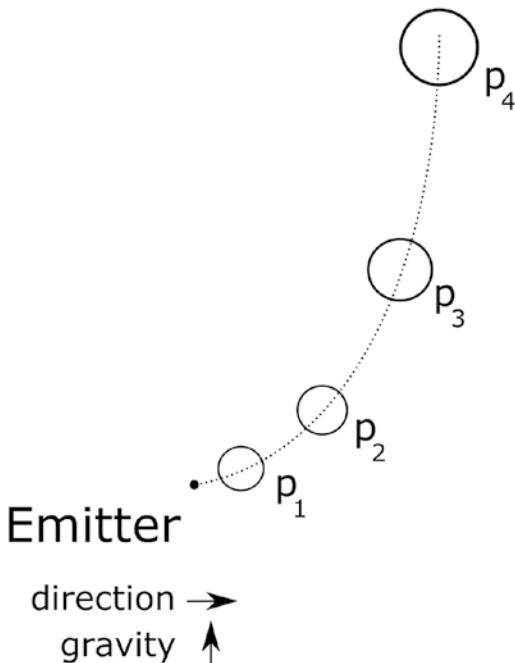
*Рис. 8-1. Огонь с помощью частиц*

В этой главе вы

- Узнаете о частицах
- Поэкспериментируете с онлайн-редактором системы частиц
- Создадите систему частиц для нашей игры.
- Добавите в игру ракеты с дымовым следом

## Анатомия частицы

Что такое частица? Это игровой объект, который будет отображаться нашим игровым движком, точно так же, как наш спрайт игрока или прокручиваемый фон. Имеет расположение на экране и текстуру. Однако одна частица не сильно влияет на визуальный аспект игры. Её сила исходит от концентрации и смешивания частиц на экране. Частица всегда будет двигаться в соответствии с принципами, заданными её излучателем. На рис.8-2 мы можем видеть, как частица вылетает с направлением и скоростью, подвергаясь действию гравитационной силы, которая не всегда направлена вниз. Пунктирная линия представляет траекторию частицы в течение её жизни и то, как частица исчезает с течением времени. Эта единственная частица изображается по мере её старения в возрасте от 1 до 4 лет. Стоит также отметить, что размер частицы также увеличивается и, кажется, ускоряется.

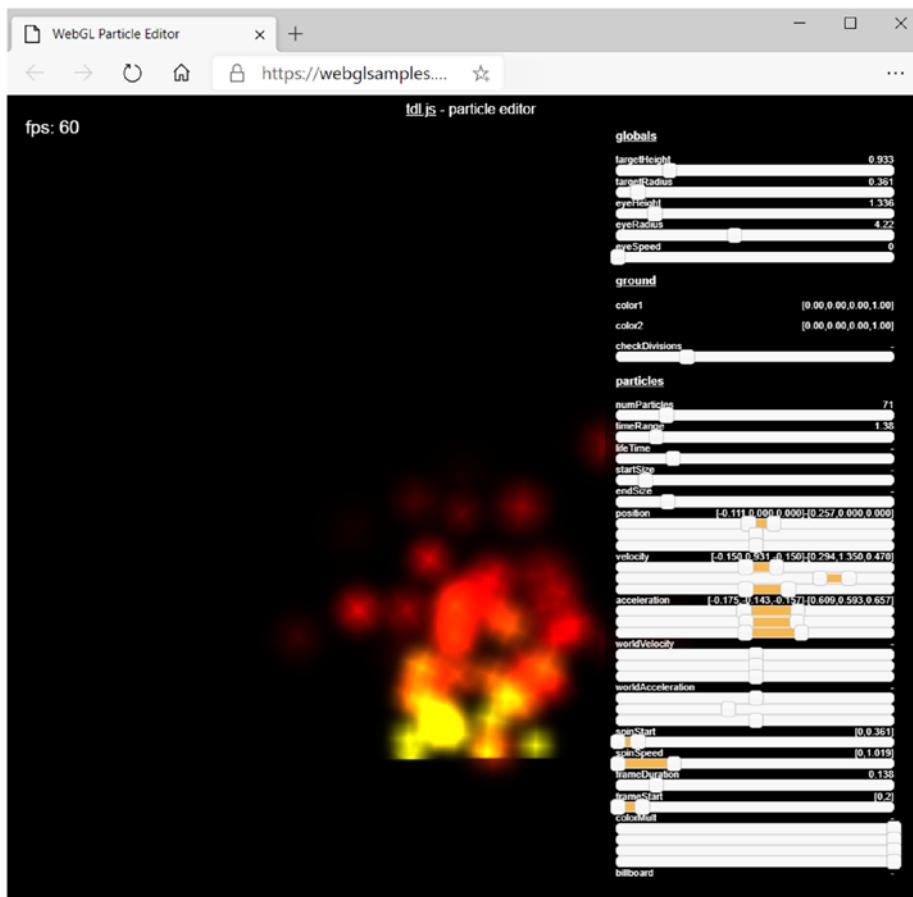


*Рис. 8-2. Изменения частицы по мере ее старения*

По мере старения частицы ее новое положение вычисляется путем добавления вектора направления к вектору силы тяжести и умножения результата на текущую скорость частицы. Это приведет к вектору, который при добавлении к текущему положению частицы даст нам следующее местоположение частицы. Затем текущая скорость частицы увеличивается путем умножения ее на значение ускорения, которое мы обсудим ниже.

## Обучение с онлайн-редактором частиц

Иногда лучший способ научиться — экспериментировать. В Интернете есть несколько онлайн-редакторов частиц, которые могут помочь нам понять, что нам нужно для работы в нашей игре. Что интересно в редакторах, так это то, что все они немного отличаются в том, как они генерируют частицы. Рис. 8-3 показывает нам онлайн-редактор со множеством различных параметров и ползунков, используемых для управления атрибутами частиц.



**Рис. 8-3.** . Онлайн-редактор частиц WebGL  
(<https://webglsamples.org/particleeditor/particleeditor.html>)

Первый атрибут, который мы можем изменить в разделе «частицы», — это количество частиц, которые мы хотим иметь в нашей системе. Чем больше у нас частиц, тем более концентрированными они будут, и в конце концов мы начнем терять отдельные частицы и будем видеть только большое пятно света, меняющее цвет. Давайте рассмотрим несколько других важных атрибутов, которые мы позже реализуем в нашем движке:

- Время жизни: важно кодифицировать, как долго мы хотим, чтобы наши частицы жили, чтобы старые частицы могли исчезнуть с экрана и освободить место для новых.
- Начальный размер: начальный размер частицы при ее создании. Размер изменится позже по мере старения частицы.
- Конечный размер: конечный размер частицы.
- Диапазон положений для осей X, Y и Z: положение частиц при их создании.
- Скорость для осей X, Y и Z: начальная скорость частиц.
- Ускорение по осям X, Y и Z: как частицы ускоряются по всем трем осям. При значении 0 частицы должны сохранять свою начальную скорость.
- Непрозрачность: по мере старения частиц они могут исчезать или становиться более непрозрачными.

Некоторые атрибуты в этом редакторе представляют собой диапазоны с минимальным и максимальным значением. Когда частица излучается и оживает, ей будет присвоено случайное значение в пределах определенного диапазона. Эта небольшая доля случайности помогает создать что-то более естественное.  
робуйте настроить все эти параметры и сделать эту систему частиц похожей на пламя.

## Различные формы излучателей частиц

В онлайн-редакторе частиц, который мы рассматривали, не было возможности изменить форму, создаваемую частицами. Вместо этого он создает частицы на плоскости пола вокруг своего начала и позволяет частицам захватывать восходящие и боковые силы. Если бы мы хотели

создать фейерверк или кольцо огня, нам понадобилось бы что-то еще. Система частиц, которую мы построим, будет учитывать различные формы при создании частиц.

## Добавление системы частиц в нашу игру

Первое использование системы частиц в нашей игре будет заключаться в создании следов дыма на соплах ракет, которыми игрок может выстрелить. На рис. 8-4 показано, как это будет выглядеть.



*Рис. 8-4. Игрок стреляет ракетой*

В будущих главах мы будем использовать нашу систему частиц, чтобы добавить искры при попадании во врага и взрывы при уничтожении различных самолетов, включая нашего собственного игрока. Поскольку взрывы, искры и следы дыма принимают различные формы, нам нужно создать что-то, что сможет справиться со всеми

Одна вещь, о которой мы должны помнить при создании системы частиц, заключается в том, что огромное количество игровых объектов, которые можно добавить в игру очень быстро, может вызвать проблемы с производительностью. Чтобы смягчить потенциальные проблемы, мы ограничим количество испускаемых частиц. Затем частицы достигают своего максимального срока службы или, когда они исчезают за кадром, мы не будем их уничтожать.

Вместо этого мы деактивируем их, чтобы предотвратить их рендеринг, и добавим их в список деактивированных частиц. Когда излучателю потребуются новые частицы, он сможет повторно использовать существующие деактивированные частицы и повторно активировать их с новыми повторно инициализированными параметрами. Это позволяет избежать дополнительных затрат на создание объектов в игре.

Давайте начнем с того, что посмотрим, какие классы мы добавим в наш игровой движок, чтобы поддерживать создание тысяч игровых объектов, каждый из которых имеет свои собственные правила формирования на экране.

## Частица

У нас не может быть системы частиц без частиц! Это будет простой класс, который ограничивается хранением нескольких параметров, которые мы рассматривали ранее, а также отвечает за самообновление каждый раз, когда игра вызывает метод `Update()`. Вот атрибуты, которые отслеживает класс частиц:

- Продолжительность жизни: у частицы есть срок жизни, который будет целым числом и будет увеличиваться на единицу каждый раз, когда вызывается функция `Update()`. Поскольку MonoGame вызывает `Update()` 60 раз в секунду, мы можем легко рассчитать, как долго мы хотим, чтобы частицы сохранялись. Например, если бы мы хотели, чтобы частица существовала 3 секунды, мы бы установили ее продолжительность жизни на 180.
- Направление: частица имеет начальное направление, заданное излучателем. С возрастом он будет стремиться двигаться в этом направлении, но на него также будут влиять гравитация, скорость и ускорение.

- Velocity : этот атрибут является начальной скоростью, с которой частица «выбрасывается» из излучателя. Эта скорость будет меняться со временем. При каждом обновлении это значение будет изменяться путем умножения текущей скорости на атрибут ускорения
- Gravity: Гравитация — это внешняя сила, которая изменяет направление движения частицы с течением времени. Если бы эта сила была направлена вниз, то она действовала бы как реальная гравитация, к которой мы привыкли. Но в системах частиц гравитация может быть направлена в любом направлении и может быть использована для того, чтобы заставить частицы подниматься к небу или для имитации сквозняка, когда гравитация внезапно уходит в сторону.
- Acceleration: представляет собой ускорение частицы по мере ее старения. Поскольку она умножается на скорость при каждом обновлении для вычисления новой скорости, значение от 0 до 1 приведет к тому, что скорость со временем станет меньше, что приведет к замедлению частицы по мере ее старения. Значение больше 1 приведет к увеличению скорости темы частицы на протяжении всей ее жизни.
- Rotation: Несмотря на то, что мы напишем некоторый код для значения вращения трека, в настоящее время мы не будем использовать вращение в нашей игре. Мы добавляем его здесь, потому что он может понадобиться нам в будущих главах, а стоимость его реализации невелика. Обычно спрайт частицы может вращаться по мере старения частицы, и это значение используется для представления текущего угла вращения частицы.
- Age: Возраст частицы в количестве кадров. Мы увеличиваем это число каждый раз, когда вызывается метод Update(). При работе с MonoGame этот метод вызывается 60 раз в секунду.

## CHAPTER 8 PARTICLES

- Position: Текущая позиция нашей частицы вигра. Он определяет, где частица будет отрисовываться на экране.
- Opacity: является ли наша частица прозрачной, непрозрачной или где-то посередине? Этот атрибут имеет значение от 0 до 1, где 0 указывает, что частица полностью прозрачна, а значение 1 означает, что наша частица должна быть полностью непрозрачной.
- OpacityFadingRate: Как и ускорение, этот атрибут определяет, насколько быстро будет меняться непрозрачность по мере старения частицы. Значение между 0 и 1 приведет к медленному уменьшению непрозрачности с течением времени, поскольку оно будет умножено на текущее значение непрозрачности. Точно так же значение выше 1 приведет к увеличению непрозрачности при каждом обновлении.
- Scale: частицы могут увеличиваться или уменьшаться со временем, и этот атрибут используется для отслеживания текущего масштаба частицы. Мы будем использовать это, чтобы уменьшить масштаб нашей игровой текстуры в соответствии с нашими потребностями. В этой главе не планируется изменять масштаб по мере старения частицы.

Давайте посмотрим на наш класс частиц:

```
public class Particle
{
    public Vector2 Position { get; private set; }
    public float Scale { get; private set; }
    public float Opacity { get; private set; }

    private int _lifespan;
    private int _age;
    private Vector2 _direction;
```

```
private Vector2 _gravity;
private float _velocity;
private float _acceleration;
private float _rotation;
private float _opacityFadingRate;

public Particle() { }

public void Activate(int lifespan, Vector2 position,
    Vector2 direction,
    Vector2 gravity,
    float velocity, float acceleration,
    float scale, float rotation, float opacity,
    float opacityFadingRate)
{
    _lifespan = lifespan;
    _direction = direction;
    _velocity = velocity;
    _gravity = gravity;
    _acceleration = acceleration;
    _rotation = rotation;
    _opacityFadingRate = opacityFadingRate;
    _age = 0;

    Position = position;
    Opacity = opacity;
    Scale = scale;
}
```

## CHAPTER 8 PARTICLES

```
// Returns false if it went past its lifespan
public bool Update(GameTime gameTime)
{
    _velocity *= _acceleration;
    _direction += _gravity;

    var positionDelta = _direction * _velocity;

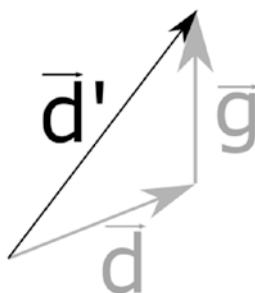
    Position += positionDelta;

    Opacity *= _opacityFadingRate;

    // Returns true if particle can stay alive
    _age++;
    return _age < _lifespan;
}
}
```

Из всех наших атрибутов `Position`, `Opacity` и `Scale` устанавливаются в качестве свойств с общедоступными геттерами, потому что излучатель будет отвечать за их отрисовку, и ему потребуется доступ к этим значениям.

Метод `Update()` вызывается 60 раз в секунду и при каждом вызове обновляет положение, непрозрачность и возраст частицы. Во-первых, скорость умножается на ускорение, либо замедляя, либо ускоряя частицу. Затем, как показано на рис. 8-5, новый вектор направления частицы  $d'$  вычисляется путем добавления вектора силы тяжести  $g$  к текущему вектору направления  $d$ .  $d'$  становится новым направлением для этой частицы. Учитывая достаточное количество времени, направление в конечном итоге совпадет с гравитацией и будет указывать в том же направлении.



*Рис. 8-5. положение векторов  $d$  и  $g$  вместе создает новый вектор  $d'$ .*

При умножении на скорость новый вектор направления растет и дает нам новое положение частицы после прибавления к текущему положению частицы. Сразу после этого мы настраиваем непрозрачность, умножая ее на `opacityFadingRate`, и, наконец, увеличиваем возраст частицы.

Метод `Update()` вернет `true`, если частица должна оставаться в живых, потому что она еще не достигла своего максимального срока жизни. Как только он вернет `false`, это будет сигналом для излучателя деактивировать его и прекратить рисовать на экране.

## EmitterParticleState

В обязанности этого класса входит хранение всех начальных параметров, необходимых излучателю для создания новой частицы. В то время как сам класс частиц отслеживает большинство тех же атрибутов, что и изменение, класс `EmitterParticleState` используется для хранения исходных значений, необходимых для создания каждой новой частицы. Здесь также интересно то, что этот класс допускает определенные вариации значений, которые он хранит. Продолжительность жизни, например, имеет минимальное и максимальное значение, а некоторые атрибуты, такие как скорость и непрозрачность, имеют связанные атрибуты отклонения, которые будут использоваться для генерации значений позже.

Атрибуты, которые отслеживает этот класс:

- MinLifeSpan - Минимальный срок службы
- MaxLifeSpan - Максимальный срок службы
- Velocity - Скорость
- VelocityDeviation - Отклонение скорости
- Acceleration - Ускорение
- Gravity - Сила тяжести
- Opacity - Непрозрачность
- OpacityDeviation - Отклонение непрозрачности
- OpacityFadingRate
- Rotation - Вращение
- RotationDeviation - Отклонение вращения
- Scale - Шкала
- ScaleDeviation - Масштаб отклонения

Каждый из этих атрибутов доступен пользователям класса через методы получения абстрактных свойств, которые должны быть переопределены дочерними классами.

Он также имеет несколько служебных функций, используемых для генерации начальных значений, необходимых при создании частицы. Поскольку мы хотим, чтобы наши частицы слегка отличались друг от друга, мы добавляем элемент случайности всякий раз, когда генерируется значение. Например, новая продолжительность жизни для новой частицы будет случайной и попадет между минимальным и максимальным значениями, определенными ранее. Точно так же начальная скорость частицы будет случайной, но между скоростью минус половина отклонения и скоростью плюс половина отклонения:

```
particle velocity = random(velocity - deviation / 2,
                           velocity + deviation / 2)
```

Результатом этого являются сгенерированные частицы, которые вылетают из излучателя с немного разными скоростями, в результате чего двигатель частиц выглядит немного более естественным. Вот код с включенным классом [RandomNumberGenerator](#) для нескольких вспомогательных функций:

```
public class RandomNumberGenerator
{
    private Random _rnd;

    public RandomNumberGenerator()
    {
        _rnd = new Random();
    }

    public int NextRandom() => _rnd.Next();
    public int NextRandom(int max) => _rnd.Next(max);
    public int NextRandom(int min, int max) => _rnd.Next(min, max);

    public float NextRandom(float max) =>
        (float)_rnd.NextDouble() * max;
    public float NextRandom(float min, float max) =>
        ((float)_rnd.NextDouble() * (max - min)) + min;
}

public abstract class EmitterParticleState
{
    private RandomNumberGenerator _rnd = new
    RandomNumberGenerator();

    public abstract int MinLifespan { get; }
    public abstract int MaxLifespan { get; }

    public abstract float Velocity { get; }
    public abstract float VelocityDeviation { get; }
```

## CHAPTER 8 PARTICLES

```
public abstract float Acceleration { get; }
public abstract Vector2 Gravity { get; }

public abstract float Opacity { get; }
public abstract float OpacityDeviation { get; }
public abstract float OpacityFadingRate { get; }

public abstract float Rotation { get; }
public abstract float RotationDeviation { get; }

public abstract float Scale { get; }
public abstract float ScaleDeviation { get; }

public int GenerateLifespan()
{
    return _rnd.NextRandom(MinLifespan, MaxLifespan);
}

public float GenerateVelocity()
{
    return GenerateFloat(Velocity, VelocityDeviation);
}

public float GenerateOpacity()
{
    return GenerateFloat(Opacity, OpacityDeviation);
}

public float GenerateRotation()
{
    return GenerateFloat(Rotation, RotationDeviation);
}
```

```

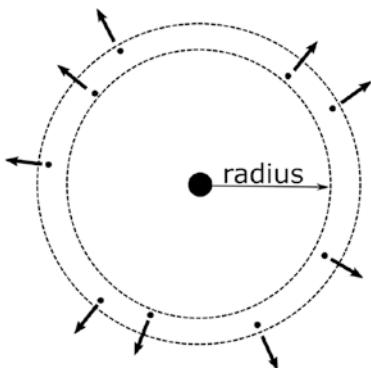
public float GenerateScale()
{
    return GenerateFloat(Scale, ScaleDeviation);
}

protected float GenerateFloat(float startN, float deviation)
{
    var halfDeviation = deviation / 2.0f;
    return _rnd.NextRandom(startN - halfDeviation, startN + halfDeviation);
}
}

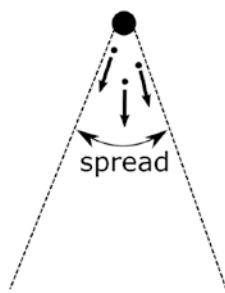
```

## IEmitterType

Излучатели могут быть разных форм. Хотим ли мы, чтобы наши частицы вылетали в форме кольца или конуса, а может, даже квадрата? Нам нужен способ, чтобы наши излучатели знали, где расположить начальные частицы и в каком направлении они должны начать движение. На рис. 8-6 показано несколько типов излучателей.



**Ring Emitter**



**Cone Emitter**

*Рис. 8-6. Две разные формы излучателя*

Этот интерфейс [IEmitterType](#) прост и предоставляет две функции для получения положения частицы и направления частицы.

```
public interface IEmitterType
{
    Vector2 GetParticleDirection();
    Vector2 GetParticlePosition(Vector2 emitterPosition);
}
```

`GetParticlePosition()` будет вычислять положение новой частицы в зависимости от положения самого излучателя. Мы хотим, чтобы новые частицы генерировались вблизи их излучателей.

## ConeEmitterType

Наша игра создаст след дыма, поэтому наш игровой движок предоставит нам излучатель, который генерирует частицы в форме конуса. Конус реализует интерфейс [IEmitterType](#) и будет иметь два новых атрибута: направление и распространение, указывающее ширину конуса. Всякий раз, когда генерируется частица, код размещает ее в том же месте, что и центр излучателя, и ее направление будет генерироваться случайным образом, но будет соответствовать разбросу конуса.

```
public class ConeEmitterType : IEmitterType
{
    public Vector2 Direction { get; private set; }
    public float Spread { get; private set; }

    private RandomNumberGenerator _rnd = new
    RandomNumberGenerator();

    public ConeEmitterType(Vector2 direction, float spread)
    {
        Direction = direction;
```

```

        Spread = spread;
    }

public Vector2 GetParticleDirection()
{
    if (Direction == null)
    {
        return new Vector2(0, 0);
    }

    var angle = (float) Math.Atan2(Direction.Y, Direction.X);
    var newAngle = _rnd.NextRandom(angle - Spread / 2.0f,
        angle + Spread / 2.0f);

    var particleDirection =
        new Vector2((float) Math.Cos(newAngle),
            (float) Math.Sin(newAngle));
    particleDirection.Normalize();
    return particleDirection;
}

public Vector2 GetParticlePosition(Vector2 emitterPosition)
{
    var x = emitterPosition.X;
    var y = emitterPosition.Y;

    return new Vector2(x, y);
}
}

```

Давайте более подробно рассмотрим функцию `GetParticleDirection()`. Во-первых, если направление излучателя, предоставленное конструктору, равно нулю, то направление частицы будет  $(0, 0)$ , что заставит ее не двигаться. В противном случае мы вычисляем угол направления излучателя и создаем новый угол для частицы в пределах предоставленного разброса. Вооружившись новым углом, мы можем

вычислить координаты X и Y для вектора направления частицы, который мы затем нормализуем, что представляет собой операцию, которая изменяет длину вектора на 1,0. Это полезно, потому что это направление в конечном итоге будет умножено на скорость частицы, и мы не хотим, чтобы вектор направления влиял на скорость движения частицы.

## Излучатель

Наконец, у нас есть сам излучатель, центральная часть всей этой операции. В обязанности излучателя входит испускание частиц, присвоение им начальных наборов параметров, визуализация их на экране и отслеживание активных и неактивных частиц.

Излучатель в нашем движке — это игровой объект, то есть у него есть позиция и текстура, которые будут использоваться для отрисовки частиц. Каждая отдельная частица, испускаемая этим классом, будет иметь одну и ту же текстуру, и ее положение будет зависеть от положения излучателя. Положение излучателя также может меняться со временем, например, когда игрок несет факел в темную комнату. Таким образом, все начальные местоположения частиц должны перемещаться вместе с излучателем.

```
public class Emitter : BaseGameObject
{
    private LinkedList<Particle> _activeParticles = new
        LinkedList<Particle>();
    private LinkedList<Particle> _inactiveParticles = new
        LinkedList<Particle>();
    private EmitterParticleState _emitterParticleState;
    private IEmitterType _emitterType;
    private int _nbParticleEmittedPerUpdate = 0;
    private int _maxNbParticle = 0;
```

```

public Emitter(Texture2D texture, Vector2 position,
               EmitterParticleState particleState,
               IEmitterType emitterType, int
               nbParticleEmittedPerUpdate, int
               maxParticles)
{
    _emitterParticleState = particleState;
    _emitterType = emitterType;
    _texture = texture;
    _nbParticleEmittedPerUpdate = nbParticleEmittedPerUpdate;
    _maxNbParticle = maxParticles;
    Position = position;
}
// The rest of the class is omitted while we discuss the
// constructor
}

```

В предыдущем коде мы инициализируем излучатель текстурой, положением, состоянием частиц излучателя, типом, количеством частиц, генерируемых при каждом вызове `Update()`, и максимальным количеством частиц, которые этот излучатель может активировать в любое время. Наши активные и неактивные частицы будут отслеживаться в двух связанных списках, изначально пустых. Для этого хорошо подходят связанные списки. Когда активные частицы умирают из-за возраста, их необходимо удалить из списка активных и добавить в список неактивных. Удаление любого элемента из связанного списка происходит мгновенно, что помогает нашему движку работать лучше.

Испускание одной частицы происходит следующим образом:

```

private void EmitNewParticle(Particle particle)
{
    var lifespan = _emitterParticleState.GenerateLifespan();
    var velocity = _emitterParticleState.GenerateVelocity();
    var scale = _emitterParticleState.GenerateScale();

```

## CHAPTER 8 PARTICLES

```
var rotation = _emitterParticleState.GenerateRotation();
var opacity = _emitterParticleState.GenerateOpacity();
var gravity = _emitterParticleState.Gravity;
var acceleration = _emitterParticleState.Acceleration;
var opacityFadingRate = _emitterParticleState.
    OpacityFadingRate;

var direction = _emitterType.GetParticleDirection();
var position = _emitterType.GetParticlePosition(_position);

particle.Activate(lifespan, position, direction, gravity,
    velocity, acceleration, scale,
    rotation, opacity, opacityFadingRate);
_activeParticles.AddLast(particle);
}
```

Для данного объекта частицы, который был недавно создан и в настоящее время неактивен или активен, эта функция сбросит его атрибуты, активирует его и добавит в список активных частиц. Каждый вызов `_emitterParticleState` здесь предназначен для генерации новых значений атрибутов, которые находятся в пределах диапазонов, определенных в объекте состояния частицы, переданном ранее излучателю. Затем мы спрашиваем тип излучателя о положении частицы и направлении. Мы берем все эти параметры и используем их для сброса частицы. Затем, после активации и добавления в список активных, эта частица будет отображаться на экране, и будет вызываться ее метод `Update()`, позволяющий ей стареть и изменяться с течением времени.

Метод `EmitNewParticle()` вызывается для каждой частицы, которую мы хотим испустить. Каждый раз, когда вызывается метод `Update()`, мы должны испускать количество частиц `_nbParticleEmittedPerUpdate`. Во-первых, мы смотрим на список неактивных частиц, чтобы увидеть, есть ли у нас какие-либо неактивные частицы, которые мы можем использовать повторно. Если у нас их достаточно, мы просто вызываем `EmitNewParticle()` для каждой из них. Если их недостаточно, то мы создаем достаточно новых частиц, чтобы заполнить пробел и также излучать их, при этом следя за тем, чтобы мы никогда не получили больше активных частиц,

чем разрешено `_maxNbParticle`. Основное преимущество такого объединения наших объектов-частиц состоит в том, чтобы избежать накладных расходов на создание новых экземпляров потенциально тысяч игровых объектов-частиц 60 раз в секунду, что может замедлить игру. Здесь, как только наши объекты созданы, они используются повторно, и мы избегаем дополнительных вычислительных затрат.

Вот код, который обрабатывает эту логику:

```
private void EmitParticles()
{
    // Make sure we're not at max particles
    if (_activeParticles.Count >= _maxNbParticle)
    {
        return;
    }

    var maxAmountThatCanBeCreated = _maxNbParticle -
        _activeParticles.Count;
    var neededParticles = Math.Min(maxAmountThatCanBeCreated,
        _nbParticleEmittedPerUpdate);

    // Reuse inactive particles first before creating new ones
    var nbToReuse = Math.Min(_inactiveParticles.Count,
        neededParticles);
    var nbToCreate = neededParticles - nbToReuse;

    for(var i = 0; i < nbToReuse; i++)
    {
        var particleNode = _inactiveParticles.First;
        EmitNewParticle(particleNode.Value);
        _inactiveParticles.Remove(particleNode);
    }
}
```

## CHAPTER 8 PARTICLES

```
for(var i = 0; i < nbToCreate; i++)
{
    EmitNewParticle(new Particle());
}
}
```

Начнем с расчета того, сколько частиц мы можем создать. Из этого числа мы вычисляем, сколько будет получено из списка неактивных частиц и сколько их нужно создать. Каждую неактивную частицу, которую мы повторно излучаем, мы извлекаем из списка и используем в вызове [EmitNewParticle\(\)](#). Для каждой новой частицы, которую нам нужно создать, мы создаем экземпляры и используем их в одном и том же вызове.

Теперь нам нужно вызвать [Update\(\)](#) для всех этих частиц, и мы должны сделать это в излучателе методом [Update\(\)](#):

```
public void Update(GameTime gameTime)
{
    EmitParticles();

    var particleNode = _activeParticles.First;
    while (particleNode != null)
    {
        var nextNode = particleNode.Next;
        var stillAlive = particleNode.Value.Update(gameTime);
        if (!stillAlive)
        {
            _activeParticles.Remove(particleNode);
            _inactiveParticles.AddLast(particleNode.Value);
        }

        particleNode = nextNode;
    }
}
```

Мы немедленно начнем с испускания частиц. Затем мы переберем все наши активные частицы и вызываем `Update()` для каждой из них. Как упоминалось при обсуждении класса `Particle`, функция `Update()` возвращает значение `false`, если возраст частицы превышает ее максимальный значение. Когда это происходит, мы удаляем его из списка активных частиц и добавляем в список неактивных частиц.

Наконец, теперь мы можем визуализировать каждую активную частицу:

```
public override void Render(SpriteBatch spriteBatch)
{
    var sourceRectangle = new Rectangle(0, 0, _texture.Width,
                                        _texture.Height);

    foreach (var particle in _activeParticles)
    {
        spriteBatch.Draw(_texture, particle.Position,
                        sourceRectangle,
                        Color.White * particle.Opacity, 0.0f,
                        new Vector2(0, 0),
                        particle.Scale, SpriteEffects.None,
                        zIndex);
    }
}
```

`Render()` будет перебирать каждую активную частицу и вызывать одну из перегруженных функций `Draw()` в пакете спрайтов, что позволит нам изменить масштаб и непрозрачность частицы.

На этом завершается та часть нашей системы частиц, которую мы добавили в наш игровой движок. Теперь мы должны создать излучатель частиц дымового следа и добавить его в игру!

## Добавление следа от ракет и дыма в нашу игру

Вы можете найти весь код для этой главы по адресу

<https://github.com/Apress/monogame-mastery/tree/master/chapter-08/end>, а все игровые ресурсы здесь:

[athttps://github.com/Apress/monogame-mastery/tree/master/chapter-08/assets](https://github.com/Apress/monogame-mastery/tree/master/chapter-08/assets).

Мы добавили две новые текстуры: одну для дымового следа и одну для ракеты. Мы также добавили новый звуковой эффект для ракеты.

Первый шаг в добавлении в игру нового типа системы частиц, особенно той, которая предназначена для быстрого перемещения за пределы экрана по мере того, как ракета быстро взлетает, — это иметь возможность экспериментировать со следом дыма и наблюдать за ним, не уходя за экран слишком быстро. К счастью, у нас уже есть механизм для этого, и мы создаем наш новый игровой объект `MissileSprite`, не заботясь о пулях, истребителях и прокручиваемом фоне.

Игра строится путем перевода игроков из одного игрового состояния в другое. Сначала игроку показывают игровое состояние `Splash`, а когда он нажимает клавишу `Enter`, он перемещается в игровое состояние. А если бы у нас было состояние `Devgame`, в котором мы могли бы помещать объекты на экран и вообще не заставлять их двигаться? Это игровое состояние будет песочницей, в которой мы можем играть, пока будем строить наши игровые объекты.

## Создание игрового состояния `Dev` для игры

Давайте создадим новую папку с именем `Dev` в каталоге `States` и добавим `DevInputCommand`, `DevInputMapper` и `DevState`.

Это игровое состояние потребует некоторого ввода, так как мы хотим иметь возможность выйти из состояния, и мы хотим убедиться, что ракета работает хорошо, когда мы запускаем ее. Вот команды, которые нам нужны:

```
public class DevInputCommand : BaseInputCommand
{
    public class DevQuit : DevInputCommand { }
    public class DevShoot : DevInputCommand { }
}
```

Как и в реальной игре, нажатие клавиши ESC приведет к выходу из игры, а нажатие клавиши пробела заставит ракету взлететь, когда мы захотим проверить эту функциональность:

```
public class DevInputMapper : BaseInputMapper
{
    public override IEnumerable<BaseInputCommand>
        GetKeyboardState(KeyboardState state)
    {
        var commands = new List<DevInputCommand>();

        if (state.IsKeyDown(Keys.Escape))
        {
            commands.Add(new DevInputCommand.DevQuit());
        }

        if (state.IsKeyDown(Keys.Space))
        {
            commands.Add(new DevInputCommand.DevShoot());
        }

        return commands;
    }
}
```

Давайте теперь добавим наши новые текстуры в конвейер контента. Нам потребуются две новые текстуры, расположенные в папке с кодами в Chapter-08\assets\png с именами Cloud001 и Missile05. Добавьте эти текстуры в конвейер содержимого и назовите их соответственно Cloud и Missile. Наш DevState будет использовать текстуру [Cloud](#) немедленно и в конечном итоге также будет использовать текстуру [Missile](#).

Теперь мы можем создать излучатель частиц дымового следа, который мы просто назвали выхлопом. Возьмите следующий код и поместите его в файл [Exhaust.cs](#), расположенный в новой папке [Particles](#):

```
public class ExhaustParticleState : EmitterParticleState
{
    public override int MinLifespan => 60; // equivalent to 1
                                              // second

    public override int MaxLifespan => 90;
    public override float Velocity => 4.0f;
    public override float VelocityDeviation => 1.0f;
    public override float Acceleration => 0.8f;
    public override Vector2 Gravity => new Vector2(0, 0);
    public override float Opacity => 0.4f;
    public override float OpacityDeviation => 0.1f;
    public override float OpacityFadingRate => 0.86f;
    public override float Rotation => 0.0f;
    public override float RotationDeviation => 0.0f;
    public override float Scale => 0.1f;
    public override float ScaleDeviation => 0.05f;
}

public class ExhaustEmitter : Emitter
{
    private const int NbParticles = 10;
    private const int MaxParticles = 1000;
    private static Vector2 Direction = new Vector2(0.0f, 1.0f);
    private const float Spread = 1.5f;
```

```

public ExhaustEmitter(Texture2D texture, Vector2 position) :
    base(texture, position, new ExhaustParticleState(),
          new ConeEmitterType(Direction, Spread),
          NbParticles, MaxParticles)
{
}

```

Этот файл содержит два класса: `ExhaustParticleState` для отслеживания начального состояния всех наших частиц дыма и класс `ExhaustEmitter`, который является излучателем, но также указывает, сколько частиц испускать при каждом обновлении, максимальное количество частиц, которые мы хотим активировать, направление излучателя вниз и что мы хотим использовать `ConeEmitterType`, который мы добавили ранее.

Наконец, давайте добавим наш класс `DevState`, который будет использоваться как песочница, чтобы мы могли работать с нашими игровыми объектами, не имея при этом необходимости иметь дело со всей игрой одновременно. `DevState`, добавленный в папку `States\Devfolder`, выглядит так:

```

public class DevState : BaseGameState
{
    private const string ExhaustTexture = "Cloud";
    private ExhaustEmitter _exhaustEmitter;

    public override void LoadContent()
    {
        var exhaustPosition = new Vector2(_viewportWidth / 2,
                                         _viewportHeight / 2);
        _exhaustEmitter = new ExhaustEmitter(LoadTexture
                                         (ExhaustTexture), exhaustPosition);
        AddGameObject(_exhaustEmitter);
    }
}

```

## CHAPTER 8 PARTICLES

```
public override void HandleInput(GameTime gameTime)
{
    InputManager.GetCommands(cmd =>
    {
        if (cmd is DevInputCommand.DevQuit)
        {
            NotifyEvent(new BaseGameStateEvent.GameQuit());
        }
    });
}

public override void UpdateGameState(GameTime gameTime)
{
    _exhaustEmitter.Update(gameTime);
}

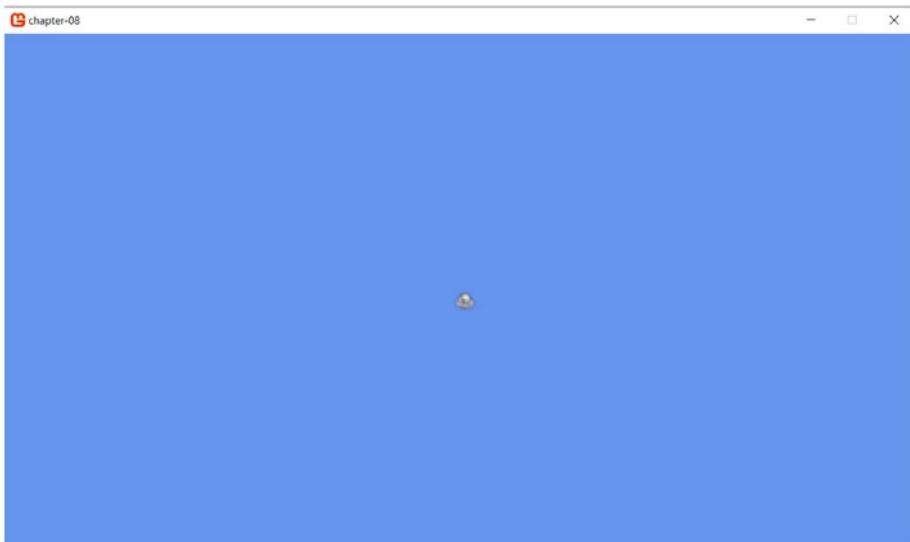
protected override void SetInputManager()
{
    InputManager = new InputManager(new DevInputMapper());
}
}
```

Это должно выглядеть знакомо. [LoadContent\(\)](#) создает излучатель выхлопных газов, помещает его в середину области просмотра и устанавливает его текстуру на текстуру облака. [HandleInput\(\)](#) выполняет команду [DevQuit](#), выходя из игры. [UpdateGameState\(\)](#) вызывает метод [Update\(\)](#) излучателя, а [SetInputManager\(\)](#) подключает наш класс [DevInputMapper](#) к диспетчеру ввода состояния.

Теперь нам нужно, чтобы игра запускалась с нашим `DevState` вместо `SplashState`. Измените метод `Main` программы `Program.cs`, чтобы использовать `DevState` в качестве его начального состояния:

```
static void Main()
{
    using (var game = new MainGame(WIDTH, HEIGHT, new
        DevState()))
        game.Run();
}
```

Запустите программу, и вы должны увидеть что-то вроде Рис. 8-7, с большим количеством частиц дыма, образующихся очень быстро в форме конуса.



*Рис. 8-7. Дым!*

## CHAPTER 8 PARTICLES

Итак, дым генерируется, но мы пока не знаем, как он выглядит при движении. Обновите метод `Update()` класса `DevState`, чтобы со временем перемещать излучатель вверх и уничтожать его, когда он уходит слишком далеко за пределы экрана:

```
public override void UpdateGameState(GameTime gameTime)
{
    _exhaustEmitter.Position =
        new Vector2(_exhaustEmitter.Position.X,
                   _exhaustEmitter.Position.Y - 3f);
    _exhaustEmitter.Update(gameTime);

    if (_exhaustEmitter.Position.Y < -200)
    {
        RemoveGameObject(_exhaustEmitter);
    }
}
```

Запустите игру еще раз, и мы должны увидеть движение дыма. Не стесняйтесь настраивать атрибуты `ExhaustParticleState` или количество частиц `ExhaustEmitter`, чтобы увидеть, как это влияет на след дыма. Поиск подходящего набора параметров — утомительная работа, в ходе которой мы изменяем атрибуты и должны каждый раз запускать нашу игру, чтобы визуализировать результаты. Неудивительно, что существуют редакторы движка частиц, которые могут помочь! Как только вы удовлетворитесь тем, как ваш выхлоп выглядит на экране, пришло время добавить ракету!

Эта `Missile` - Ракета будет составным игровым объектом. У него есть текстура ракеты (текстура ракеты, которую мы добавили в конвейер содержимого ранее) и излучатель выхлопа. Это два игровых объекта в одном, что имеет смысл, поскольку они тесно связаны друг с другом. Создайте новый `MissileSprite.cs` в папке `Objects` и используйте следующий код:

```
public class MissileSprite : BaseGameObject
{
    private const float StartSpeed = 0.5f;
    private const float Acceleration = 0.15f;
```

```
private float _speed = StartSpeed;

// Keep track of scaled-down texture size
private int _missileHeight;
private int _missileWidth;

// Missiles are attached to their own particle emitter
private ExhaustEmitter _exhaustEmitter;

public override Vector2 Position
{
    set
    {
        _position = value;
        _exhaustEmitter.Position =
            new Vector2(_position.X + 18, _position.Y +
            _missileHeight - 10);
    }
}

public MissileSprite(Texture2D missleTexture, Texture2D
exhaustTexture)
{
    _texture = missleTexture;
    _exhaustEmitter = new ExhaustEmitter(exhaustTexture,
        _position);

    var ratio = (float) _texture.Height /
        (float) _texture.Width;
    _missileWidth = 50;
    _missileHeight = (int) (_missileWidth * ratio);
}
```

## CHAPTER 8 PARTICLES

```
public void Update(GameTime gameTime)
{
    _exhaustEmitter.Update(gameTime);

    Position = new Vector2(Position.X, Position.Y - _speed);
    _speed = _speed + Acceleration;
}

public override void Render(SpriteBatch spriteBatch)
{
    // Need to scale down the sprite. The original texture
    // is very big
    var destRectangle =
        new Rectangle((int) Position.X, (int) Position.Y,
                      _missileWidth, _missileHeight);
    spriteBatch.Draw(_texture, destRectangle, Color.White);

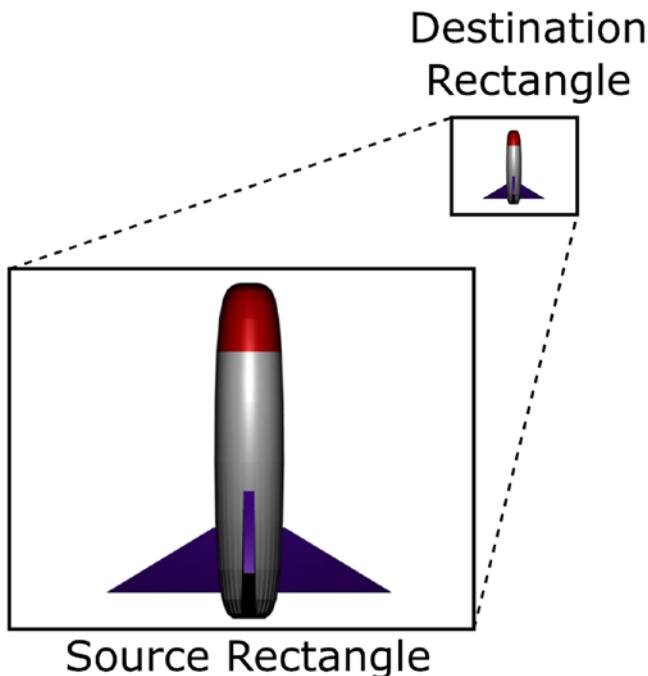
    _exhaustEmitter.Render(spriteBatch);
}
}
```

Основное различие между этим игровым объектом и другими, такими как спрайт игрока, заключается в том, что мы добавили игровой объект излучателя выхлопных газов в спрайт [MissileSprite](#). При создании ракеты мы должны предоставить две текстуры: текстуру ракеты и текстуру дыма для излучателя. Более того, всякий раз, когда положение ракеты изменяется, мы также должны обновлять положение излучателя, о чём мы заботимся в функции установки свойства [Position](#). Каждый вызов метода [Update\(\)](#) объекта [MissileSprite](#) также вызывает метод [Update\(\)](#) излучателя, и аналогичным образом каждый вызов [Render\(\)](#) будет вызывать функцию [Render\(\)](#) излучателя в дополнение к отрисовке текстуры ракеты на экране.

Исходная текстура ракеты очень большая, и перед ее рисованием ее необходимо уменьшить. Ширина 50 пикселей кажется разумной для использования здесь, поэтому нам нужно рассчитать желаемую высоту ракеты, чтобы она поддерживала то же соотношение ширины и высоты,

что и исходный размер текстуры, что делается в конструкторе класса.

При рисовании ракеты мы используем функцию рисования `SpriteBatch`, где мы можем указать исходный прямоугольник текстуры и прямоугольник назначения на экране. Если два прямоугольника имеют разные размеры, MonoGame позаботится о масштабировании исходного прямоугольника до размера целевого прямоугольника, как показано на Рис. 8-8.



*Рис. 8-8. Уменьшение текстуры нашей ракеты*

Давайте добавим экземпляр объекта `MissileSprite` в наше `DevState` и посмотрим, что из этого выйдет! Ради интереса давайте также добавим спрайт игрока и немножко поиграем с объектами в нашей песочнице. Измените класс `DevState`, чтобы он выглядел следующим образом:

```
public class DevState : BaseGameState
{
    private const string ExhaustTexture = "Cloud";// "Облако"
    private const string MissileTexture = "Missile"; "Ракета"
```

## CHAPTER 8 PARTICLES

```
private const string PlayerFighter = "fighter";

private ExhaustEmitter _exhaustEmitter;
private MissileSprite _missile;
private PlayerSprite _player;

public override void LoadContent()
{
    var exhaustPosition = new Vector2(_viewportWidth / 2,
        _viewportHeight / 2);
    _exhaustEmitter = new ExhaustEmitter(LoadTexture(
        ExhaustTexture), exhaustPosition);
    AddGameObject(_exhaustEmitter);

    _player = new PlayerSprite(LoadTexture(PlayerFighter));
    _player.Position = new Vector2(500, 500);
    AddGameObject(_player);
}

public override void HandleInput(GameTime gameTime)
{
    InputManager.GetCommands(cmd =>
    {
        if (cmd is DevInputCommand.DevQuit)
        {
            NotifyEvent(new BaseGameStateEvent.GameQuit());
        }

        if (cmd is DevInputCommand.DevShoot)
        {
            _missile =
                new MissileSprite(LoadTexture(MissileTexture),
                    LoadTexture(Exhaust
                    Texture));
        }
    });
}
```

```
        _missile.Position = new Vector2(_player.  
Position.X, _player.Position.Y - 25);  
        AddGameObject(_missile);  
    }  
});  
}  
  
public override void UpdateGameState(GameTime gameTime)  
{  
    _exhaustEmitter.Position =  
        new Vector2(_exhaustEmitter.Position.X,  
                    _exhaustEmitter.Position.Y - 3f);  
    _exhaustEmitter.Update(gameTime);  
  
    if (_missile != null)  
    {  
        _missile.Update(gameTime);  
  
        if (_missile.Position.Y < -100)  
        {  
            RemoveGameObject(_missile);  
        }  
    }  
  
    if (_exhaustEmitter.Position.Y < -200)  
    {  
        RemoveGameObject(_exhaustEmitter);  
    }  
}  
  
protected override void SetInputManager()  
{  
    InputManager = new InputManager(new DevInputMapper());  
}  
}
```

Мы следим за командой [DevShoot](#) и добавляем в игру ракету всякий раз, когда она срабатывает. Ракета взлетает сразу же, а за ней остается дым, поэтому мы можем легко увидеть удачный результат нашей работы. Однако это не идеальный игровой код. Метод [Update\(\)](#) вызывается 60 раз в секунду, и хотя мы нажимаем клавишу пробела только один раз, она остается нажатой в течение нескольких из этих кадров, и таким образом создается несколько ракет, но все они сохраняются в частной переменной `same_missile`, вызывая некоторые визуальные эффекты. артефакты. Кроме того, ракета не находится по центру спрайта игрока, как мы видим на рис. 8-9. Но давайте не будем тратить силы на решение этого здесь, в песочнице. Мы посвятим время исправлению этих проблем в реальной игре.



*Рис. 8-9. Добавление игровых объектов в нашу песочницу, с интересными визуальными артефактами*

## Добавление игрового объекта «Ракеты» в нашу игру

Чтобы начать добавлять ракеты в нашу игру, нам сначала нужно выполнить [revertProgram.cs](#), чтобы загрузить класс [GameplayState](#) при запуске.

```
static void Main()
{
    using (var game = new MainGame(WIDTH, HEIGHT, new
        SplashState()))
        game.Run();
}
```

Мы хотим, чтобы наши ракеты стреляли только один раз в секунду, когда игрок нажимает пробел, а пули стреляют в то же время. Это сделано для того, чтобы игрок не выпустил слишком много ракет. Нам также нужно добавить звуковой эффект для ракеты. Откройте конвейер содержимого и выполните те же шаги, которые мы использовали в предыдущей главе для звуковых эффектов пули. Добавьте wav-файл assets\sounds\missile и назовите его [RocketSound](#). Обязательно используйте [Wav Importer](#) и процессор звуковых эффектов. Сохраните и создайте конвейер содержимого, прежде чем вернуться к коду.

In the GameState class, add these private variables:

```
private const string ExhaustTexture = "Cloud";
private const string MissileTexture = "Missile";
private Texture2D _missileTexture;
private bool _isShootingMissile;
private TimeSpan _lastMissileShotAt;
private List<MissileSprite> _missileList;
```

Мы будем следовать той же схеме, которую мы использовали для стрельбы пулями. Когда мы запускаем ракету, логическая переменная [\\_isShootingMissile](#) будет установлена в значение [true](#), чтобы мы больше не запускали ракеты. Каждую секунду эта переменная будет устанавливаться в [false](#), чтобы мы могли запустить их снова. Мы также будем отслеживать ракеты на экране с помощью [\\_missileList](#), чтобы мы могли обновлять и отображать их.

Теперь обновите [LoadContent\(\)](#), чтобы загрузить текстуру ракеты и звуковой эффект ракеты, которые нужно добавить в банк звуков нашего менеджера звука.

## CHAPTER 8 PARTICLES

```
_missileTexture = LoadTexture(MissileTexture);
_exhaustTexture = LoadTexture(ExhaustTexture);
_missileList = new List<MissileSprite>();

var missileSound = LoadSound("missileSound");
_soundManager.RegisterSound(
    new GameplayEvents.PlayerShootsMissile(), missileSound,
    0.4f, -0.2f, 0.0f
);
```

Нам нужно добавить новое событие `GameplayEvent`, чтобы менеджер звука знал, что нужно воспроизводить звуковой эффект ракеты всякий раз, когда мы стреляем ракетой. Обновите класс `GameplayEvents`, чтобы он выглядел следующим образом:

```
public class GameplayEvents : BaseGameStateEvent
{
    public class PlayerShootsBullets : GameplayEvents { }
    public class PlayerShootsMissile : GameplayEvents { }
}
```

Метод `Shoot()` вызывается всякий раз, когда игрок нажимает пробел. Добавьте в него следующий код. Он должен выглядеть очень похоже на код, который мы использовали для стрельбы пулями:

```
if (!_isShootingMissile)
{
    CreateMissile();
    _isShootingMissile = true;
    _lastMissileShotAt = gameTime.TotalGameTime;

    NotifyEvent(new GameplayEvents.PlayerShootsMissile());
}
```

Функция [CreateMissile\(\)](#) является новой и должна выглядеть так, создавая, размещая и добавляя ракеты в список игровых объектов:

```
private void CreateMissile()
{
    var missileSprite = new MissileSprite(_missileTexture,
        _exhaustTexture);
    missileSprite.Position =
        new Vector2(_playerSprite.Position.X + 33,
            _playerSprite.Position.Y - 25);

    _missileList.Add(missileSprite);
    AddGameObject(missileSprite);
}
```

Также, как и пули, по мере того, как ракеты уходят за пределы экрана, их нужно удалять из списка игровых объектов. У нас уже есть код для очистки пуль в функции [UpdateGameState\(\)](#), и очистка ракет будет выглядеть очень похоже. Вместо того, чтобы дублировать этот код для ракет, теперь у нас есть возможность создать новый метод, который будет очищать пули и ракеты. Возьмите код очистки из [UpdateGameState\(\)](#), параметризуйте его так, чтобы можно было очистить любой [BaseGameObject](#), и переместите эту логику в новый метод [CleanObjects](#):

```
private List<T> CleanObjects<T>(List<T> objectList) where T :
    BaseGameObject
{
    List<T> listOfItemsToKeep = new List<T>();
    foreach(T item in objectList)
    {
        var stillOnScreen = item.Position.Y > -50;
```

## CHAPTER 8 PARTICLES

```
        if (stillOnScreen)
        {
            listOfItemsToKeep.Add(item);
        }
        else
        {
            RemoveGameObject(item);
        }
    }

    return listOfItemsToKeep;
}
```

Теперь мы можем обновить [UpdateGameState\(\)](#), чтобы перемещать ракеты на экране, вызывав их метод [Update\(\)](#), убедиться, что мы не можем запускать их чаще, чем раз в секунду, и очистить их после того, как они исчезнут с экрана:

```
public override void UpdateGameState(GameTime gameTime)
{
    foreach (var bullet in _bulletList)
    {
        bullet.MoveUp();
    }

    foreach (var missile in _missileList)
    {
        missile.Update(gameTime);
    }

    // Can't shoot bullets more than every 0.2 second
    if (_lastBulletShotAt != null &&
        gameTime.TotalGameTime - _lastBulletShotAt > TimeSpan.
        FromSeconds(0.2))
```

```

{
    _isShootingBullets = false;
}

// Can't shoot missiles more than every 1 second
if (_lastMissileShotAt != null &&
    gameTime.TotalGameTime - _lastMissileShotAt > TimeSpan.
    FromSeconds(1.0))
{
    _isShootingMissile = false;
}

// Get rid of bullets and missiles that have gone out of view
_bulletList = CleanObjects(_bulletList);
_missileList = CleanObjects(_missileList);
}

```

Сделав эти модификации, вы можете запустить игру и увидеть, что игрок может одновременно стрелять пулями и ракетами!

## Резюме

Вы можете потратить много времени на настройку двигателей частиц и параметров эмиттера для достижения желаемых эффектов. В этой главе мы разработали собственный движок частиц, чтобы генерировать след дыма за ракетами, выпущенными нашим игроком. Эффект добавляет немного изысканности в игру, но мог бы быть и лучше. Например, использование линейных преобразований для обновления таких параметров, как непрозрачность или скорость во времени, можно было бы улучшить, используя кривые перехода, где значение атрибута во времени соответствует кривой, а не прямой линии. Другие улучшения заключались бы в реализации вращения и масштабирования частиц с течением времени, изменения цвета частиц с использованием нескольких текстур для разных частиц или даже изменение текстур частиц с течением времени с помощью анимации по мере их старения.

## CHAPTER 8 PARTICLES

Добавление движка частиц потребовало большой работы, но вид дымового следа за нашими ракетами очень аккуратный и добавляет немного реализма в нашу игру. разработчики могут использовать для экспериментов и создания новых игровых объектов, которые после создания можно легко добавить в состояние игрового процесса. В следующей главе мы будем работать над обнаружением столкновений. Что может столкнуться в нашей игре? Наши пули и... враги!

## ГЛАВА 9

# Обнаружение столкновений

Подумайте о видеоигре без обнаружения столкновений в той или иной форме. Это сложно! Обнаружение столкновений используется повсеместно и является частью многих аспектов, составляющих игру. Игрок стоит на платформе или должен падать? Как мы можем предотвратить выход игрока за край карты мира? Ударил ли игрок вращающимся молотом? Удалось ли нам подобрать блестящие монеты? Все эти вещи решаются обнаружением столкновений. Это то, что мы будем исследовать в этой главе. Как мы можем обнаружить, что два объекта столкнулись друг с другом, и что нужно делать, когда такое столкновение произошло? Еще одним интересным вариантом использования обнаружения столкновений является определение того, какой игровой объект щелкнул игрок с помощью мыши или пальца.

Если подумать, мы уже внедрили в игру наивную форму обнаружения столкновений, когда проверяли каждое обновление, если игрок выходит за границы. По сути, мы обнаруживаем столкновение с краями экрана и предотвращаем дальнейшее продвижение спрайта игрока.

Мы потратили некоторое время на рефакторинг нашего кода для этой главы и не будем здесь подробно описывать все эти изменения. Были внесены некоторые изменения, поскольку мы определили области программы с дублирующейся логикой или необходимостью изменить наш класс излучателя частиц, чтобы он мог перестать излучать частицы, не удаляя излучатель из игровых объектов, чтобы позволить существующим частицам исчезать, а не резко удаляться из них. экран. Вы можете скачать код, с которого мы начнем из папки с кодами в здесь:

[monogame-mastery/tree/master/chapter-09/start](https://monogame-mastery/tree/master/chapter-09/start). Окончательный код, включая логику обнаружения столкновений, находится здесь:

[monogame-mastery/tree/master/chapter-09/end](https://monogame-mastery/tree/master/chapter-09/end). Все внесенные нами изменения будут присутствовать в конечном проекте.

Рис. 9-1 дает нам представление о том, как будет выглядеть наша игра в конце этой главы.



*Рис. 9-1. Окончательные результаты*

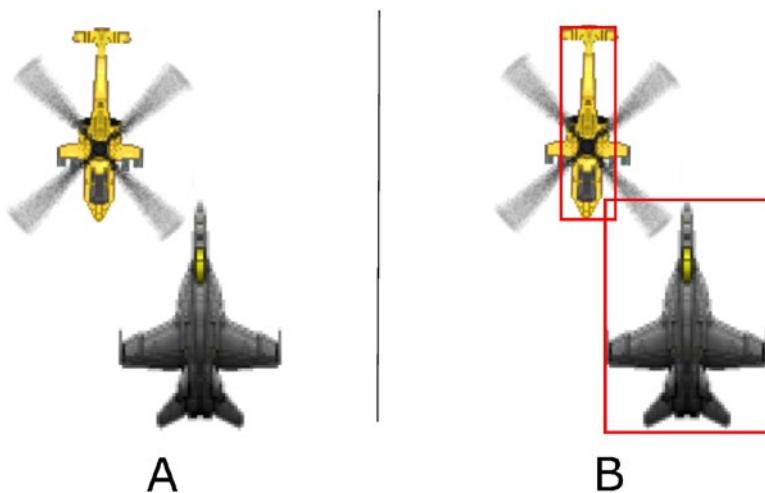
Теперь давайте начнем! В этой главе вы

- Рассмотрите несколько алгоритмов обнаружения столкновений, используемых сегодня в видеоиграх.
- Добавите в игру вражеские вертолеты, чтобы игрок мог стрелять по ним.
- Добавите в двигатель детектор столкновений и уничтожите некоторых врагов

## Техники

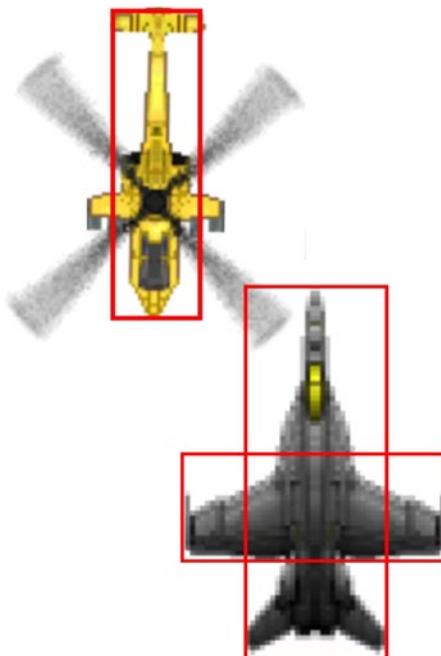
Обнаружение столкновений — настолько обширная тема, что о ней написаны целые книги. Мы изучим основные понятия об этом и несколько алгоритмов, прежде чем решить, как мы будем использовать их в нашей игре.

То, сталкиваются ли две вещи, полностью основано на концепции ограничивающих прямоугольников, которые представляют собой более простой способ представления игровых объектов с целью расчета, сталкиваются ли они друг с другом. Визуально люди могут легко увидеть, соприкоснулись ли два объекта. Однако у компьютеров с этим труднее, поэтому мы должны им помочь. Ограничивающие рамки называются так потому, что они чаще всего представляют внешние границы объекта и обычно представляют собой прямоугольники в двух измерениях или прямоугольники в трех измерениях. Посмотрите на [Рис.9-2](#). Слева мы видим спрайт нашего игрока и вражеский вертолет. Они сталкиваются? Как бы вы написали алгоритм, чтобы определить, что они делают? В правой части изображения вы видите тот же объект с нарисованными ограничивающими рамками. Такой способ представления объектов значительно упрощает и ускоряет обнаружение пересечения двух объектов. Возможно, он не идеален, но работает достаточно хорошо для большинства целей, где не требуется абсолютная точность.



*Рис. 9-2. Наш спрайт-истребитель сталкивается с вертолетом?  
Да, это!*

У нас тут небольшая проблема. Хотя теперь мы можем обнаруживать столкновения между объектами, проверяя, пересекаются ли их ограничивающие рамки, мы можем обнаруживать столкновения, которые на самом деле не происходят, как на предыдущем изображении. Есть несколько способов смягчить это. Во-первых, у прямоугольников здесь нет монополии. С тем же успехом мы могли бы иметь ограничивающую сферу или круг, если бы наш игровой объект лучше соответствовал этой форме. Во-вторых, как мы реализуем позже в этой главе, мы также можем иметь несколько ограничивающих рамок, как показано на [рис. 9-3](#). Использование более чем одного прямоугольника для определения контура наших игровых объектов упрощает подгонку формы, которую мы хотим сопоставить, но добавляет немного дополнительной работы, потому что это один дополнительный квадрат, который нам нужно проверять для каждого игрового объекта на экране.



*Рис. 9-3. Спрайт нашего истребителя сталкивается с вертолетом? Не в этот раз.*

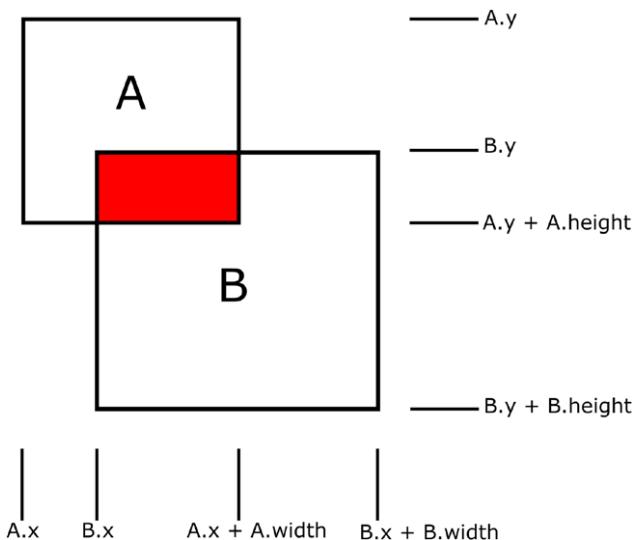
Любой объект, который может столкнуться с другим объектом, должен иметь какую-то ограничивающую рамку или некоторые границы другой формы, такие как сфера или конус. Если у нас есть сотни или тысячи объектов в игре, таких как искры, которые падают на пол и не разрешено проходить через пол, какой алгоритм мы должны использовать для обработки всех этих столкновений? Давайте рассмотрим несколько.

## **AABB (границчная рамка, выровненная по оси)**

До сих пор все ограничивающие прямоугольники, которые мы рассматривали, были выровнены по осям X и Y, а их стороны полностью горизонтальны и вертикальны. Вычисление ограничивающих прямоугольников с двумя выровненными осями несложно: заданы прямоугольники A и B, их положение ( $x, y$ ), , и их размеры (ширина, высота), два прямоугольника пересекаются, если

- $A.x \leq B.x + B.width$  and
- $A.x + A.width \geq B.x$  and
- $A.y \leq B.y + B.height$  and
- $A.y + A.height \geq B.y$

См. [рис. 9-4](#) для визуального представления этих четырех состояний.



*Рис. 9-4. Два пересекающихся прямоугольника*

## ОВВ (Oriented Bounding Box) - ориентированная ограничивающая рамка

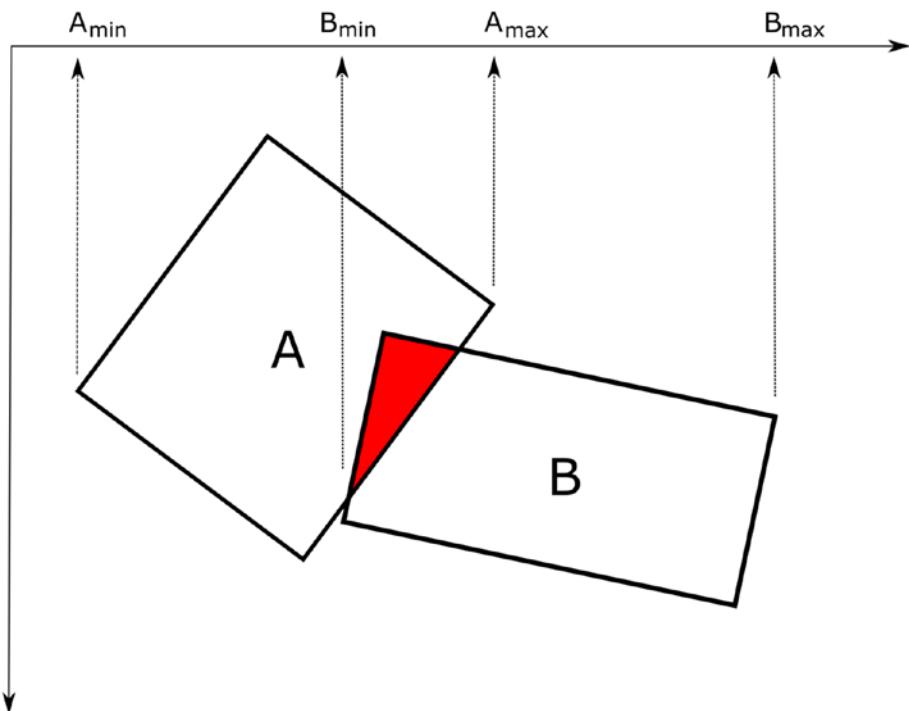
ОВВ аналогичны рамкам с выровненными осями, но их можно вращать по мере вращения игрового объекта, что немногого усложняет вычисления для обнаружения столкновений. Даны две ограничивающие рамки А и В для каждой оси, которые в двумерной игре представляют собой X и Y, спроектируйте все углы ограничивающей рамки А на эту ось и сохраните только наименьшую и наибольшую точки.

Теперь сделайте то же самое для ограничивающей рамки B. Посмотрите на [Рис.9-5](#) пример проецирования углов ограничивающей рамки на ось X.

Теперь расчет похож на AABB.

Если для всех осей верно следующее, то мы имеем коллизию, если

- $A_{\min} \leq B_{\max}$
- $B_{\min} \leq A_{\max}$

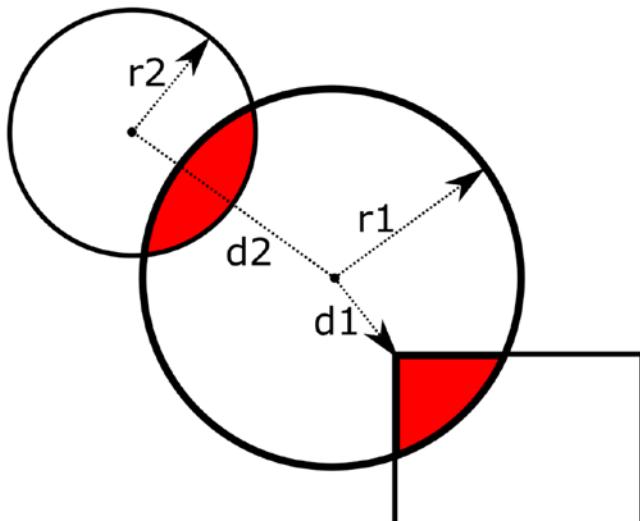


*Рис. 9-5. Ориентированные ограничивающие прямоугольники пересекаются*

## Сфера

Если вы решите использовать сферы или круги в 2D-игре, вам может понадобиться вычислить, пересекается ли ваша сфера с другими сферами или прямоугольниками. На рис. 9-6 средний круг пересекается с другим кругом и прямоугольником. Два круга или сферы сталкиваются, когда расстояние между их центрами меньше суммы их радиусов. На рис.9-6 две окружности сталкиваются, потому что  $d_2$  меньше  $r_1 + r_2$ .

При поиске, сталкивается ли прямоугольник с кругом, вычислите расстояние между центром круга и ближайшей точкой прямоугольника, которая может не быть одним из углов. Если какой-либо из них меньше радиуса окружности, мы имеем столкновение. На Рис.9-6 это представлено тем фактом, что  $d_1$  меньше  $r_1$ .



*Рис. 9-6. Круг пересекается с другим кругом и прямоугольником*

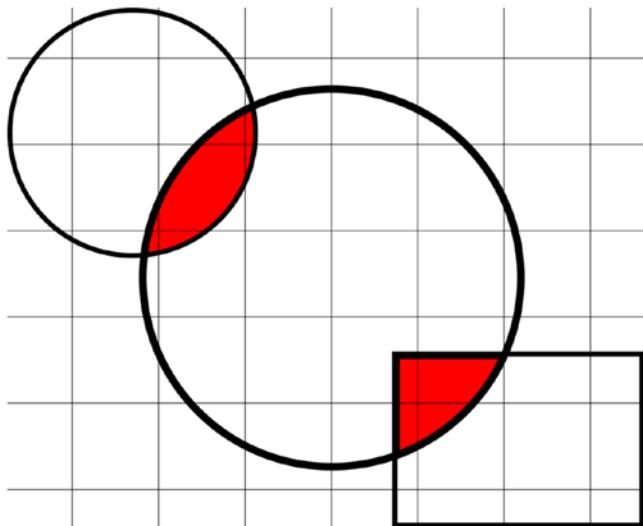
## Равномерные сетки

До сих пор наша стратегия заключалась в том, чтобы собрать все объекты на экране и посмотреть, какие объекты с какими объектами сталкиваются. Это может быть хорошо, если у нас есть только несколько объектов для просмотра, но по мере роста количества объектов количество вычислений также растет экспоненциально. Например, имея на экране 100 объектов, мы выполним не более  $100^2$  обнаружений столкновений. Поскольку мы запускаем их при каждом вызове метода `Update()`, это означает, что мы должны иметь возможность делать это 60 раз в секунду, иначе наша игра рискует замедлиться.

Чтобы уменьшить количество проверок, которые мы выполняем в каждом кадре, было разработано несколько алгоритмов для размещения наших объектов в пространственной структуре данных, чтобы мы знали, какие объекты находятся рядом друг с другом. Затем нам нужно только проверить наличие коллизий между соседними объектами, что значительно сэкономит время вычислений. Единственным недостатком этих следующих нескольких алгоритмов является то, что упорядочивание объектов в структурах данных требует времени, но, надеюсь, не так много, как запуск алгоритма обнаружения столкновений грубой силы.

С учетом сказанного, давайте начнем с равномерных сеток, которые представляют собой двумерный массив в котором каждая ячейка изначально пуста и имеет ширину и высоту. Ячейки сетки, если мы когда-либо удосужились нарисовать их на экране, должны заполнить окно просмотра. Затем мы берем все наши игровые объекты и выясняем, какие ячейки они перекрывают. Для каждой из этих ячеек мы добавляем ссылку на игровой объект. Когда все игровые объекты обработаны, мы проходим по каждой ячейке и смотрим, какие из них связаны более чем с одним объектом. Когда это происходит, мы выполняем обнаружение столкновений на связанных объектах, чтобы увидеть, как ограничивающие прямоугольники или сферы пересекаются. См. пример на Рис. 9-7.

Насколько широка и высока каждая ячейка, полностью зависит от разработчика игры. Поскольку цель состоит в том, чтобы свести к минимуму количество вычислений, которые мы в конечном итоге должны выполнить, большее количество ячеек уменьшит вероятность того, что в них будет связано более одного объекта. Однако это может привести к тому, что одни и те же объекты будут связаны во многих ячейках. С другой стороны, если ячейки слишком велики, мы возвращаемся к исходной точке, если все игровые объекты оказываются в одной и той же ячейке, и нам придется перебирать путь через обнаружение столкновений. Как вы можете видеть на Рис. 9-7, обнаружение столкновений для двух окружностей произойдет шесть раз, потому что только ячейки связаны с двумя объектами. Поскольку нам нужно выполнить обнаружение столкновений только один раз, программисту нужно будет выполнить это действие только один раз для каждой пары игровых объектов.



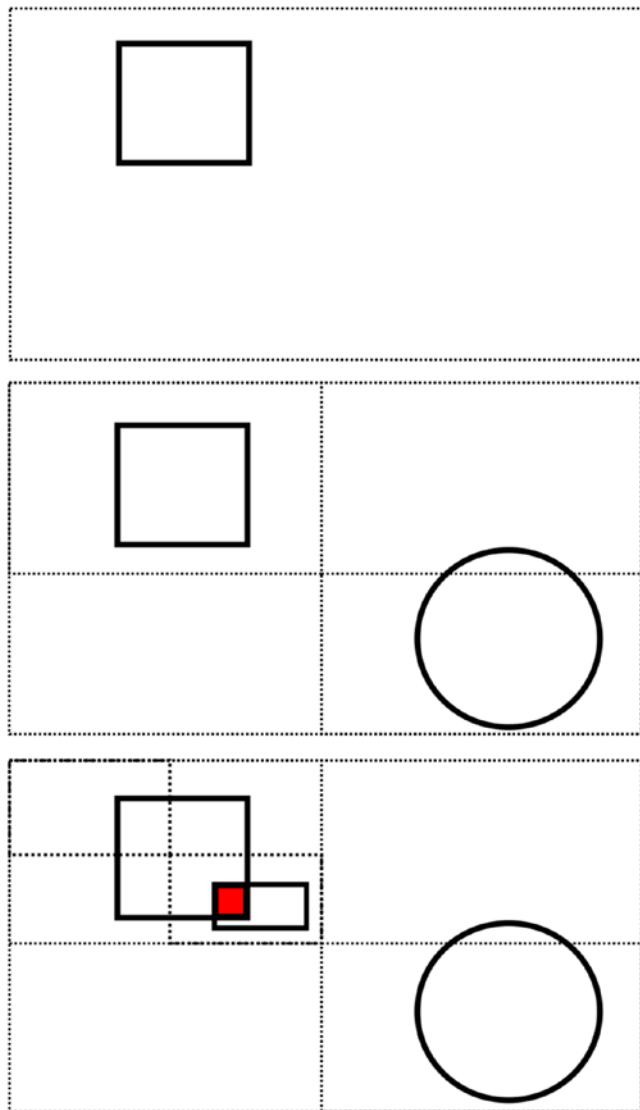
*Рис. 9-7. Представляем наши объекты, если бы они были нарисованы на *a uniformgrid**

## Quadtrees

Quadtrees — это разновидность равномерных сеток, но вместо того, чтобы использовать 2D-массив в качестве структуры данных для связывания игровых объектов, мы используем дерево, в котором каждый узел имеет либо ноль, либо четыре потомка. При создании новогодерева квадрантов мы передаем в качестве параметра размер нашего окна просмотра и максимальное количество игровых объектов, которые может хранить узел. Сначала дерево создаст свой корневой узел, который покрывает весь экран, а затем мы можем добавлять в дерево наши игровые объекты один за другим. Объекты добавляются к корневому узлу до тех пор, пока не будет достигнуто максимальное количество объектов на узел, после чего дерево квадрантов разделится и создаст четыре дочерних пустых узла, а все игровые объекты

в корневых узлах переместятся в дочерние узлы в зависимости от того, где они находятся. Если игровой объект перекрывает два узла, он сохраняется в родительском узле. Затем процесс повторяется до тех пор, пока не будут обработаны все объекты. При создании каждого узла он точно знает, какие части экрана он покрывает, отслеживая координаты верхнего левого угла, ширину и высоту.

На рис. 9-8 приведен пример дерева квадрантов с максимальной пропускной способностью узлов, равной единице, что означает, что узлы будут делиться на четыре всякий раз, когда в них есть два или более объекта. Сначала квадрат помещается в корневой узел. Затем добавляется круг, что означает, что корневой узел имеет два объекта и должен подразделяться на четыре других дочерних узла. Квадрат перемещается в узел, представляющий верхний левый квадрант. Окружность, однако, перекрывает два квадранта, поэтому она останется в корневом узле. Наконец, маленький прямоугольник добавляется к первому квадранту, что приводит к его разделению на четыре. Большой прямоугольник остается в своем узле, потому что он перекрывает дочерние узлы, а меньший прямоугольник помещается в четвертый дочерний квадрант.



*Рис. 9-8. Дерево квадрантов с максимальной емкостью одного узла*

Чтобы выполнить обнаружение столкновений, мы перебираем наши игровые объекты и запрашиваем у дерева квадрантов все объекты, принадлежащие одному и тому же узлу или дочерним узлам. Затем мы выполняем обнаружение столкновений только между этими объектами.

## Другие методы

Существует множество других методов, используемых как для обнаружения столкновений, так и для оптимизации производительности всех этих вычислений. Мы обсудили некоторые из них в этом разделе, но мы рекомендуем вам исследовать этот мир возможностей, чтобы найти то, что соответствует вашим потребностям. В нашем распоряжении множество инструментов, от различных видов древовидных структур до 3D-алгоритмов и подходов на основе рендеринга.

## Добавление врагов в нашу игру

Прежде чем мы реализуем обнаружение столкновений в нашей игре, нам нужно что-то, с чем можно сталкиваться. И поскольку мы реализуем вертикальный шутер, большинство столкновений будет между пулями и врагами. Итак, нам нужны враги!

Мы добавили два новых изображения в каталог assets\png:

- Одно изображение с несколькими вертолетами разного цвета: `chopper-44x99.png`
- Новая текстура взрыва, которую мы будем использовать для нового излучателя частиц, когда враги (или игрок) умирают: `explosion.png`

Добавьте два новых изображения в инструмент конвейера содержимого и назовите изображение вертолета в инструменте `chopper.png`. Сохраните и создайте конвейер контента.

Теперь мы можем создать наш новый игровой объект-враг вертолета и использовать первую текстуру вертолета из этого изображения вертолета. Хотя на изображении много разных вертолетов, включая два разных типа лопастей, мы будем использовать только первый желтый вертолет, добавим размытые лопасти и заставим их вращаться с течением времени. Для этого мы будем использовать версию метода `SpriteBatchRender`, которая позволяет нам выбрать определенную область исходного изображения, которую мы хотим отобразить на экране.

Таким образом, мы можем выбрать желтый вертолет и размытые лопасти и визуализировать их друг над другом на экране.

```
public class ChopperSprite : BaseGameObject
{
    // Which chopper do we want from the texture
    private const int ChopperStartX = 0;
    private const int ChopperStartY = 0;
    private const int ChopperWidth = 44;
    private const int ChopperHeight = 98;

    // Where are the blades on the texture
    private const int BladesStartX = 133;
    private const int BladesStartY = 98;
    private const int BladesWidth = 94;
    private const int BladesHeight = 94;

    // Rotation center of the blades
    private const float BladesCenterX = 47.5f;
    private const float BladesCenterY = 47.5f;

    // Positioning of the blades on the chopper
    private const int ChopperBladePosX = ChopperWidth / 2;
    private const int ChopperBladePosY = 34;

    private int _life = 40;

    public ChopperSprite(Texture2D texture)
    {
        _texture = texture;
    }
}
```

```
public override void OnNotify(BaseGameStateEvent gameEvent)
{
    switch (gameEvent)
    {
        case GameplayEvents.ChopperHitBy m:
            JustHit(m.HitBy);
            SendEvent(new GameplayEvents.EnemyLostLife
(_life));
            break;
    }
}

private void JustHit(IGameObjectWithDamage o)
{
    _hitAt = 0;
    _life -= o.Damage;
}
}
```

Здесь многое происходит. Мы отслеживаем положение желаемого вертолета по текстуре, положение размытых лопастей, где мы хотим, чтобы лопасти располагались поверх вертолета, и центральное положение лопастей при их вращении. Мы также отслеживаем очки жизни на вертолете. Когда в него попадут пули и ракеты, его жизнь будет падать, пока не достигнет нуля, после чего он будет удален из игры и заменен излучателем взрывных частиц. Каждый объект вертолета будет реагировать на новое игровое событие ChopperHitBy, которое сообщает ему, что игровой объект, реализующий интерфейс [IGameObjectWithDamage](#), просто ударил его. Интерфейс, показанный в приведенном ниже коде, предполагает свойство [Damage](#), которое вертолет будет использовать для уменьшения общего количества жизней. После удара вертолет отправит уведомление об изменении своего собственного состояния, что позволит классу [GameplayState](#) отреагировать на смерть вертолета, удалив его из игры и заменив излучателем частиц взрыва.

```
public interface IGameObjectWithDamage
{
    int Damage { get; }
}
```

Теперь нам нужно отобразить наш вертолет на экране, но у нас есть небольшая проблема... Вертолеты в текстуре все обращены вверх, но если наши враги собираются спуститься и атаковать нашего игрока сзади, они должны быть обращены вниз. Нам нужно повернуть их вокруг.

## Вращение нашего вертолета

Вращение не является сложной задачей, так как MonoGame поддерживает этот режим. Но нам нужно знать, где находится центр вращения. Если мы не укажем его, MonoGame будет считать, что центр вращения совпадает с началом текстуры, что означает координаты (0, 0). На рис. 9-9 показан пример того, как будет выглядеть рендеринг, если мы повернули вертолет на 180 градусов без изменения центра вращения. Мы видим оригинальный вертолет, обращенный вверх, и повернутый вертолет. Само по себе это не имеет большого значения, но нам нужно добавить вращающиеся лопасти к этому повернутому врагу, а вращение вертолета вокруг начала координат (0, 0) затрудняет выбор места для размещения текстуры лопастей. Вместо этого мы должны вращать вертолет вокруг соответствующего центра вращения, который может находиться именно там, где должны двигаться лопасти.



*Рис. 9-9. Вращение нашего вертолета на 180 градусов вокруг его (0, 0) начала*

Мы можем начать работать над нашим методом `Render`:

```
public override void Render(SpriteBatch spriteBatch)
{
    var chopperRect =
        new Rectangle(ChopperStartX, ChopperStartY,
                      ChopperWidth, ChopperHeight);
    var chopperDestRect =
        new Rectangle(_positionToPoint(), new
                      Point(ChopperWidth, ChopperHeight));

    var color = Color.White;
    spriteBatch.Draw(_texture, chopperDestRect, chopperRect,
                    color, MathHelper.Pi,
                    new Vector2(ChopperBladePosX,
                               ChopperBladePosY), SpriteEffects.None, 0f);
}
```

Предыдущий код вычисляет исходный прямоугольник тела нашего желтого вертолета на исходном изображении и целевой прямоугольник на экране, где мы хотим нарисовать вертолет. Затем мы рисуем врага и поворачиваем его на 180 градусов, используя угол в радианах в качестве пятого параметра для `spriteBatch.Draw`. 180 градусов в радианах — это PI, поэтому здесь мы используем свойство MonoGame `MathHelper.Pi`.

## Вращающиеся лопасти

Давайте добавим несколько вращающихся лопастей. Во-первых, нам нужно указать скорость вращения лопастей в радианах. Мы будем использовать это число для увеличения угла поворота лопастей при каждом вызове метода `Render()`.

```
private const float BladeSpeed = 0.2f;
private float _angle = 0.0f;
```

Теперь нам нужно добавить вращающееся лопасть к вертолету. Добавьте следующий код в метод `Render()`:

```
var bladesRect = new Rectangle(BladesStartX, BladesStartY,
    BladesWidth, BladesHeight);
var bladesDestRect = new Rectangle(_positionToPoint(), new
    Point(BladesWidth, BladesHeight));

spriteBatch.Draw(_texture, bladesDestRect, bladesRect, Color.
    White, _angle,
    new Vector2(BladesCenterX, BladesCenterY),
    SpriteEffects.None, 0f);

_angle += BladeSpeed;
```

Предыдущий код аналогичен тому, что мы сделали для рендеринга повернутого чоппера, за исключением того, что мы передаем переменную `_angle` методу `Draw()` и увеличиваем угол на константу `BladeSpeed` в каждом кадре.

## Заставить вертолеты двигаться

Чтобы заставить наш вертолет двигаться по экрану, мы предоставим им путь через параметр конструктора, который представляет собой список номеров кадров и векторов направлений. Спрайты будут пробиваться на экран с левой или правой стороны окна просмотра, а через определенное количество кадров менять направление и двигаться вниз по диагонали. Чтобы реализовать это, давайте добавим к нашему объекту переменную скорости и добавим этот путь в конструктор. Затем, когда вызывается метод `Update()`, мы рассчитаем возраст игрового объекта в количестве кадров, прошедших с момента его создания, и сравним этот возраст с путем, который должен пройти вертолет, и соответствующим образом изменим направление вертолета. Во-первых, нам нужны переменные класса:

```
private const float Speed = 4.0f;
private Vector2 _direction = new Vector2(0, 0);
private int _age = 0;

private List<(int, Vector2)> _path;
```

В переменной `_path` хранятся кортежи типа `(int, Vector2)`, которые представляют номер кадра и связанное с ним направление.

Измените конструктор, чтобы указать путь:

```
public ChopperSprite(Texture2D texture, List<(int, Vector2)>
path)
{
    _texture = texture;
    _path = path;
}
```

## CHAPTER 9 COLLISION DETECTION

И добавьте метод Update, который может быть вызван нашим классом GameplayState:

```
public void Update()
{
    // Вертолеты следуют по пути, где направление меняется в
    // определенный кадр,
    // который отслеживается по ageforeach вертолета (var p в _path)

    {
        int pAge = p.Item1;
        Vector2 pDirection = p.Item2;

        if (_age > pAge)
        {
            _direction = pDirection;
        }
    }

    Position = Position + (_direction * Speed);

    _age++;
}
```

Не стесняйтесь экспериментировать с этим новым игровым объектом в классе [DevStateclass](#), нашем классе состояния песочницы, который мы создали в предыдущей главе, чтобы почувствовать, как вертолет может перемещаться по экрану! Дайте им путь и обязательно вызовите для них Update.

Чтобы сгенерировать вертолеты с левой и правой стороны экрана, мы создаем класс [ChopperGenerator](#), отвечающий за создание игровых объектов [ChopperSprite](#) и установку их вне экрана слева или справа. В следующем коде генератор создается с текстурой вертолета, числом вертолетов для создания и обработчик, который класс состояния игры будет использовать для получения уведомления о создании вертолета, чтобы он мог обновить свой список врагов и добавить объекты в список активных игровых объектов.

Когда вызывается `GenerateChoppers()`, генератор будет создавать прерыватель каждые 500 миллисекунд, попеременно располагая его слева или справа от экрана. Он также создаст путь для вертолета и назначит его.

```
public class ChopperGenerator
{
    private bool _generateLeft = true;
    private Vector2 _leftVector = new Vector2(-1, 0);
    private Vector2 _downLeftVector = new Vector2(-1, 1);
    private Vector2 _rightVector = new Vector2(1, 0);
    private Vector2 _downRightVector = new Vector2(1, 1);

    private Texture2D _texture;
    private System.Timers.Timer _timer;
    private Action<ChopperSprite> _chopperHandler;
    private int _maxChoppers = 0;
    private int _choppersGenerated = 0;
    private bool _generating = false;

    public ChopperGenerator(Texture2D texture, int nbChoppers,
                           Action<ChopperSprite> handler)
    {
        _texture = texture;
        _chopperHandler = handler;
        _downLeftVector.Normalize();
        _downRightVector.Normalize();
        _maxChoppers = nbChoppers;
    }
}
```

## CHAPTER 9 COLLISION DETECTION

```
_timer = new System.Timers.Timer(500);
_timer.Elapsed += _timer_Elapsed;
}

public void GenerateChoppers()
{
    if (_generating)
    {
        return;
    }

    _choppersGenerated = 0;
    _timer.Start();
}

public void StopGenerating()
{
    _timer.Stop();
    _generating = false;
}

private void _timer_Elapsed(object sender,
    System.Timers.ElapsedEventArgs e)
{
    List<int, Vector2> path;
    if (_generateLeft)
    {
        path = new List<int, Vector2>
        {
            (0, _rightVector),
            (2 * 60, _downRightVector),
        };
    }
}
```

```

var chopper = new ChopperSprite(_texture, path);
chopper.Position = new Vector2(-200, 100);
_chopperHandler(chopper);
}
else
{
    path = new List<(int, Vector2)>
    {
        (0, _leftVector),
        (2 * 60, _downLeftVector),
    };
    var chopper = new ChopperSprite(_texture, path);
    chopper.Position = new Vector2(1500, 100);
    _chopperHandler(chopper);
}
_generateLeft = !_generateLeft;
_choppersGenerated++;
if (_choppersGenerated == _maxChoppers)
{
    StopGenerating();
}
}
}

```

В нашем классе `GameState` давайте добавим вертолеты в список врагов и заставим их двигаться.

Мы также можем добавить новый излучатель частиц взрыва, пока мы там, что мы рассмотрим в следующем разделе. Однако сейчас мы можем посмотреть, как мы справляемся с этим здесь. В класс излучателя частиц были внесены некоторые изменения, которые мы не будем здесь описывать, чтобы позволить классу состояния игры останавливать излучатель, не уничтожая его:

```
public class GameState : BaseGameState
{
    // ...

    private const string ChopperTexture = "Chopper";
    private const string ExplosionTexture = "explosion";
    private const int MaxExplosionAge = 600; // 10 seconds at
                                              // 60 frames per
                                              // second = 600

    // Emit particles for 1.2 seconds and let them fade out for
    // 10 seconds
    private const int ExplosionActiveLength = 75;
    private Texture2D _chopperTexture;

    private List<ExplosionEmitter> _explosionList = new
        List<ExplosionEmitter>();
    private List<ChopperSprite> _enemyList = new
        List<ChopperSprite>();

    public override void LoadContent()
    {
        _explosionTexture = LoadTexture(ExplosionTexture);
        _chopperTexture = LoadTexture(ChopperTexture);
        _chopperGenerator = new ChopperGenerator(_
            chopperTexture, 4, AddChopper);
        _chopperGenerator.GenerateChoppers();
    }

    public override void UpdateGameState(GameTime gameTime)
    {
        // ...
        foreach (var chopper in _enemyList)
```

```
{  
    chopper.Update();  
}  
_enemyList = CleanObjects(_enemyList);  
}  
  
private void AddChopper(ChopperSprite chopper)  
{  
    chopper.OnObjectChanged +=  
        _chopperSprite_OnObjectChanged;  
    _enemyList.Add(chopper);  
    AddGameObject(chopper);  
}  
  
private void AddExplosion(Vector2 position)  
{  
    var explosion = new ExplosionEmitter(_explosionTexture,  
                                         position);  
    AddGameObject(explosion);  
    _explosionList.Add(explosion);  
}  
  
private void UpdateExplosions(GameTime gameTime)  
{  
    foreach (var explosion in _explosionList)  
    {  
        explosion.Update(gameTime);  
  
        if (explosion.Age > ExplosionActiveLength)  
        {  
            explosion.Deactivate();  
        }  
    }  
}
```

```

        if (explosion.Age > MaxExplosionAge)
        {
            RemoveGameObject(explosion);
        }
    }
}

```

Обратите внимание, что в методе `AddChopper()` мы регистрируемся на событие, которое было добавлено в `BaseGameObject`. Отдельные игровые объекты, такие как `ChopperSprite`, могут инициировать это событие при изменении их внутреннего состояния. Мы будем использовать это событие ниже, когда будем обрабатывать врагов, теряющих все свое количество жизней.

Мы также отслеживаем все добавленные в игру взрывы и обновляем их, чтобы частицы могли испускаться. Как только взрыв достигает определенного возраста, он перестает излучать частицы, из-за чего существующие на некоторое время исчезают, пока излучатель не достигнет своего максимального возраста и не будет удален из игры.

## Добавление двигателя взрывных частиц

Теперь, когда мы можем добавить врагов в нашу игру, врагов, которых мы можем стрелять и уничтожать, нам понадобится новый излучатель частиц, чтобы у нас был хоть какой-тоrudиментарный взрыв, когда враги (или игрок) умирают.

Взрывы не конусообразные. Вместо этого частицы должны генерироваться внутри круга и расти со временем. Для этого нам нужен новый тип излучателя, который не имеет направления и будет генерировать частицы в пределах заданного радиуса. Добавьте этот класс в каталог `Engine\Particles\EmitterTypes`:

```

public class CircleEmitterType : IEmitterType
{
    public float Radius { get; private set; }
}

```

```
private RandomNumberGenerator _rnd = new
RandomNumberGenerator();

public CircleEmitterType(float radius)
{
    Radius = radius;
}

public Vector2 GetParticleDirection()
{
    return new Vector2(0f, 0f);
}

public Vector2 GetParticlePosition(Vector2 emitterPosition)
{
    var newAngle = _rnd.NextRandom(0, 2 * MathHelper.Pi);
    var positionVector = new Vector2(
        (float)Math.Cos(newAngle),
        (float)Math.Sin(newAngle));
    positionVector.Normalize();

    var distance = _rnd.NextRandom(0, Radius);
    var position = positionVector * distance;

    var x = emitterPosition.X + position.X;
    var y = emitterPosition.Y + position.Y;

    return new Vector2(x, y);
}
```

## CHAPTER 9 COLLISION DETECTION

Then, in our games Particles\ directory, add an Explosion.cs file and add the following code to it:

```
public class ExplosionParticleState : EmitterParticleState
{
    public override int MinLifespan => 180; // equivalent to 3
                                                // seconds

    public override int MaxLifespan => 240;

    public override float Velocity => 2.0f;

    public override float VelocityDeviation => 0.0f;

    public override float Acceleration => 0.999f;

    public override Vector2 Gravity => new Vector2(0, 1);

    public override float Opacity => 0.4f;

    public override float OpacityDeviation => 0.1f;

    public override float OpacityFadingRate => 0.92f;

    public override float Rotation => 0.0f;

    public override float RotationDeviation => 0.0f;

    public override float Scale => 0.5f;

    public override float ScaleDeviation => 0.1f;
}

public class ExplosionEmitter : Emitter
{
    private const int NbParticles = 2;
    private const int MaxParticles = 200;
```

```

private const float Radius = 50f;

public ExplosionEmitter(Texture2D texture, Vector2
position) :
    base(texture, position, new ExplosionParticleState(),
    new CircleEmitterType(Radius),
    NbParticles, MaxParticles) { }

}

```

Мы добавили направление гравитации к нашим частицам, поэтому облако текстур взрыва слегка смещается вниз, создавая иллюзию движения. Вы можете добавить экземпляра `ExplosionEmitter` в `DevState` и посмотреть, как он работает сам по себе. В игре, когда враг умирает, мы удалим игровой объект вертолет из списка активных игровых объектов и заменим его излучателем взрыва. Он будет генерировать частицы в течение нескольких секунд перед остановкой.

## Добавление обнаружения столкновений

Теперь мы готовы добавить в нашу игру логику для обнаружения следующих столкновений:

- Попадают ли пули во врага?
- Попадают ли ракеты во врага?
- Столкивается ли враг со спрайтом игрока?

Учитывая, что в нашей игре нет огромного количества объектов в любой момент времени, может быть, десятков пуль, нескольких врагов и, в конце концов, их собственных снарядов, мы можем позволить себе использовать метод грубой силы для обнаружения столкновений для каждой пары объектов в игра. Однако мы можем быть умнее, потому что знаем, что пули никогда не пересекаются друг с другом. Нам тоже все равно, попадут ли ракеты в пули. Мы уже поддерживаем список пуль и список ракет в нашем классе `GameplayState`. Было бы тривиально повторно

использовать эти списки, добавить новый список врагов и выполнить обнаружение столкновений между двумя списками игровых объектов: одним списком пассивных объектов, таких как пули, и списком активных объектов, которые будут уведомлены при попадании в них.

## Ограничивающие рамки

Для обнаружения столкновений между игровыми объектами нам нужны ограничивающие рамки, а MonoGame предоставляет готовую поддержку ограничивающих рамок через класс [BoundingBox](#) в пространстве имен [Microsoft.Xna.Framework](#). Этот класс дает нам несколько полезных функций, таких как слияние двух ограничивающих рамок в одну, создание ограничивающих рамок из сфер и вычисление пересечения двух ограничивающих рамок друг с другом, что может привести к столкновению. К несчастью для нашей 2D-игры, класс [MonoGameBoundingBox](#) использует трехмерные векторы, а это означает, что он в основном используется для 3D-игр. Хотя этот класс можно было бы использовать в 2D-игре, идея не забывать устанавливать все на плоскости  $z=0$ , чтобы избавиться от третьего измерения, не очень привлекательна. Вот почему мы создадим собственный класс 2D [BoundingBox](#) для нашей игры.

Добавьте следующий класс в Engine\Objects:

```
public class BoundingBox
{
    public Vector2 Position { get; set; }
    public float Width { get; set; }
    public float Height { get; set; }

    public Rectangle Rectangle
    {
        get
    }
}
```

```
{  
    return new Rectangle((int)Position.X, (int)  
        Position.Y, (int)Width, (int)Height);  
}  
}  
  
public BoundingBox(Vector2 position, float width,  
float height)  
{  
    Position = position;  
    Width = width;  
    Height = height;  
}  
  
public bool CollidesWith(BoundingBox otherBB)  
{  
    if (Position.X < otherBB.Position.X + otherBB.Width &&  
        Position.X + Width > otherBB.Position.X &&  
        Position.Y < otherBB.Position.Y + otherBB.Height &&  
        Position.Y + Height > otherBB.Position.Y)  
    {  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}  
}
```

## CHAPTER 9 COLLISION DETECTION

Наш класс `BoundingBox` построен с использованием положения на экране, ширины и высоты. Мы также предоставляем служебное свойство `Rectangle`, которое преобразует наш блок в прямоугольник, который мы будем использовать позже для визуализации наших блоков на экране. Это полезно, когда нам нужно отладить, почему наши ограничивающие рамки не находятся в нужном месте. Наконец, класс предлагает вспомогательную функцию, которая возвращает `true`, когда пересекается с другим ограничивающим прямоугольником.

Теперь нам нужно обновить наши игровые объекты, чтобы добавить ограничивающие рамки. Если вы помните рис.9-3, ранее в этой главе мы описывали использование одного блока для вертолета и двух блоков для спрайта игрока. Нам также нужно добавить ограничивающую рамку к нашим пулям и ракетам. Поскольку некоторые игровые объекты требуют использования ограничивающих рамок, имеет смысл добавить эту функциональность в базовый класс. Откройте `BaseGameObject` и добавьте следующий код:

```
protected List<BoundingBox> _boundingBoxes = new  
List<BoundingBox>();  
  
public List<BoundingBox> BoundingBoxes  
{  
    get  
    {  
        return _boundingBoxes;  
    }  
}  
  
public virtual Vector2 Position  
{  
    get { return _position; }  
    set  
    {  
        var deltaX = value.X - _position.X;  
        var deltaY = value.Y - _position.Y;  
        _position = value;  
    }  
}
```

```

        foreach(var bb in _boundingBoxes)
        {
            bb.Position = new Vector2(bb.Position.X + deltaX,
            bb.Position.Y + deltaY);
        }
    }

public void AddBoundingBox(BoundingBox bb)
{
    _boundingBoxes.Add(bb);
}

public void RenderBoundingBoxes(SpriteBatch spriteBatch)
{
    if (_boundingBoxTexture == null)
    {
        CreateBoundingBoxTexture(spriteBatch.GraphicsDevice);
    }

    foreach (var bb in _boundingBoxes)
    {
        spriteBatch.Draw(_boundingBoxTexture, bb.Rectangle,
        Color.Red);
    }
}

```

Наши игровые объекты теперь могут добавлять ограничивающие рамки в список, отслеживаемый базовым классом. Этот список будет доступен нашим классам состояния игры для обнаружения столкновений. Мы также обновили свойство Position, чтобы при его изменении мы также меняли положение ограничивающей рамки на экране.

Наконец, мы также можем визуализировать блоки на экране, просто в виде больших красных прямоугольников, что позволяет разработчикам легко видеть, находятся ли их блоки в правильном положении, и следовать за движущимися объектами.

Давайте посмотрим, как выглядит добавление ограничивающих рамок для нашего объекта [choppergame](#):

```
private int BBPosX = -16;
private int BBPosY = -63;
private int BBWidth = 34;
private int BBHeight = 98;

public ChopperSprite(Texture2D texture, List<(int, Vector2)>
path)
{
    _texture = texture;
    _path = path;
    AddBoundingBox(new Engine.Objects.BoundingBox(
        new Vector2(BBPosX, BBPosY), BBWidth, BBHeight));
}
```

Вот и все! Добавить ограничивающую рамку к объекту так же просто, как вызвать `AddBoundingBox` для каждой рамки, которую мы хотим добавить. Об остальном позаботится `BaseGameObject`.

Вы можете заметить, что переменные `BBPosX` и `BBPosY` отрицательные, что означает, что ограничивающая рамка находится за пределами экрана. Это потому, что изначально положение измельчителя равно `(0, 0)`, а затем мы поворачиваем его на 180 градусов вокруг исходной точки, в результате чего спрайт поворачивается за пределы экрана. Вместо того, чтобы писать код поворота для аналогичного поворота ограничивающего прямоугольника в ту же позицию, мы позволили себе изменить положение прямоугольника непосредственно в переменных.

Наши игровые объекты «`player` - игрок», «`bullet` - пуля» и «`missile` - ракета» имеют схожую логику, но вы можете удивиться, как мы пришли к значениям ограничивающих прямоугольников. Мы можем легко определить положение, ширину и высоту ограничивающей рамки, открыв текстуру в графическом инструменте, таком как `Photoshop` или даже `Paint`, который дает нам пиксельные координаты, когда мы наводим указатель мыши на загруженное изображение.

За исключением текстуры ракеты, масштаб которой уменьшается во время рендеринга, наши ограничивающие рамки можно получить непосредственно с помощью этих инструментов.

```

public class BulletSprite : BaseGameObject ,
IGameObjectWithDamage
{
    private const int BBPosX = 9;
    private const int BBPosY = 4;
    private const int BBWidth = 10;
    private const int BBHeight = 22;

    public int Damage => 10;

    public BulletSprite(Texture2D texture)
    {
        _texture = texture;
        AddBoundingBox(
            new Engine.Objects.BoundingBox(new Vector2
                (BBPosX, BBPosY), BBWidth, BBHeight));
    }
}

```

Обратите внимание, что класс пули теперь реализует интерфейс `IGameObjectWithDamage`, поэтому он должен иметь свойство `Damage`, которое возвращает 10. Класс ракеты, приведенный ниже, также будет иметь этот интерфейс, но его урон немного выше — 25.

```

public class PlayerSprite : BaseGameObject
{
    private const int BB1PosX = 29;
    private const int BB1PosY = 2;
    private const int BB1Width = 57;
    private const int BB1Height = 147;

    private const int BB2PosX = 2;
    private const int BB2PosY = 77;
    private const int BB2Width = 111;

```

## CHAPTER 9 COLLISION DETECTION

```
private const int BB2Height = 37;

public PlayerSprite(Texture2D texture)
{
    _texture = texture;
    AddBoundingBox(
        new Engine.Objects.BoundingBox(new Vector2(BB1PosX,
            BB1PosY), BB1Width, BB1Height));
    AddBoundingBox(
        new Engine.Objects.BoundingBox(new Vector2(BB2PosX,
            BB2PosY), BB2Width, BB2Height));
}

public class MissileSprite : BaseGameObject ,
IGameObjectWithDamage
{
    private int _missileHeight;
    private int _missileWidth;

    public int Damage => 25;

    public MissileSprite(Texture2D missleTexture, Texture2D
exhaustTexture)
    {
        _texture = missleTexture;
        _exhaustEmitter = new ExhaustEmitter(exhaustTexture,
            _position);

        var ratio = (float) _texture.Height /
                    (float) _texture.Width;
        _missileWidth = 50;
        _missileHeight = (int) (_missileWidth * ratio);
    }
}
```

```

// Note that the missile is scaled down! So its
// bounding box must be scaled down as well
var bbRatio = (float) _missileWidth / _texture.Width;

var bbOriginalPositionX = 352;
var bbOriginalPositionY = 7;
var bbOriginalWidth = 150;
var bbOriginalHeight = 500;

var bbPositionX = bbOriginalPositionX * bbRatio;
var bbPositionY = bbOriginalPositionY * bbRatio;
var bbWidth = bbOriginalWidth * bbRatio;
var bbHeight = bbOriginalHeight * bbRatio;

AddBoundingBox(
    new Engine.Objects.BoundingBox(new
        Vector2(bbPositionX, bbPositionY), bbWidth,
        bbHeight));
}

}

```

Стоит потратить несколько минут на ограничивающие прямоугольники ракеты. Исходная текстура ракеты намного больше, чем мы хотим, чтобы они были нарисованы, и масштабируется во время рендеринга. Чтобы вычислить его поле рендеринга, мы извлекаем положение, ширину и высоту из исходного изображения и умножаем эти значения на коэффициент масштабирования, чтобы получить новые значения, соответствующие уменьшенной текстуре ракеты.

Чтобы визуализировать ограничивающие рамки, мы можем добавить переменную `_debug` в `BaseGameState`, которая по умолчанию имеет значение `false`:

```
protected bool _debug = false;
```

and use it in the rendering code:

```
public void Render(SpriteBatch spriteBatch)
{
    foreach (var gameObject in _gameObjects.OrderBy(a =>
        a.zIndex))
    {
        gameObject.Render(spriteBatch);

        if (_debug)
        {
            gameObject.RenderBoundingBoxes(spriteBatch);
        }
    }
}
```

Это заставит игровой движок рисовать ограничивающие рамки поверх игровых объектов в каждом кадре, если включен флаг `_debug`, как показано на [Рис. 9-10](#), где мы видим, что все ограничивающие рамки расположены и масштабируются правильно.

---

<sup>1</sup> Мы обновили код в главе 10, чтобы вместо этого рисовать ограничивающие рамки под игровыми объектами, поскольку это упрощает визуализацию.



*Рис. 9-10. Отрисовка ограничивающих прямоугольников поверх игровых объектов для отладки*

## Обнаружение столкновений AABB

Теперь мы готовы обнаруживать столкновения. Поскольку мы собираемся выполнять обнаружение столкновений в стиле AABB, добавьте класс в `Engine\Objects` с именем `AABCollisionDetector`:

```
public class AABBCollisionDetector<P, A>
    where P : BaseGameObject
    where A : BaseGameObject
{
    private List<P> _passiveObjects;

    public AABBCollisionDetector(List<P> passiveObjects)
    {
        _passiveObjects = passiveObjects;
    }
}
```

## CHAPTER 9 COLLISION DETECTION

```
public void DetectCollisions(A activeObject, Action<P, A>
collisionHandler)
{
    foreach(var passiveObject in _passiveObjects)
    {
        if (DetectCollision(passiveObject, activeObject))
        {
            collisionHandler(passiveObject, activeObject);
        }
    }
}

public void DetectCollisions(List<A> activeObjects,
Action<P, A> collisionHandler)
{
    foreach(var passiveObject in _passiveObjects)
    {
        foreach(var activeObject in activeObjects)
        {
            if (DetectCollision(passiveObject,
activeObject))
            {
                collisionHandler(passiveObject,
activeObject);
            }
        }
    }
}

private bool DetectCollision(P passiveObject, A
activeObject)
{
    foreach(var passiveBB in passiveObject.BoundingBoxes)
```

```

{
    foreach(var activeBB in activeObject.BoundingBoxes)
    {
        if (passiveBB.CollidesWith(activeBB))
        {
            return true;
        }
    }
}

return false;
}
}

```

Этот класс создается со списком пассивных объектов, таких как пули, которые мы будем использовать позже для проверки столкновений с одним активным объектом или списком активных объектов с помощью двух перегруженных методов [DetectCollisions\(\)](#).

Оба метода будут перебирать каждый пассивный и активный объекты и проверять наличие коллизий с помощью алгоритма ААВВ, который мы обсуждали выше в этой главе. Когда столкновение обнаружено, вызывается функция обработчика столкновений, и оба сталкивающихся объекта передаются в качестве параметров, давая вызывающей стороне возможность отреагировать на каждое столкновение.

В классе `GameState` добавьте следующий код в метод [UpdateGameState\(\)](#):

`DetectCollisions();`

И реализуйте этот метод [DetectCollisions\(\)](#):

```

private void DetectCollisions()
{
    var bulletCollisionDetector =
        new AABBCollisionDetector<BulletSprite,
        ChopperSprite>(_bulletList);

```

## CHAPTER 9 COLLISION DETECTION

```
var missileCollisionDetector =
    new AABBCollisionDetector<MissileSprite,
    ChopperSprite>(_missileList);
var playerCollisionDetector =
    new AABBCollisionDetector<ChopperSprite,
    PlayerSprite>(_enemyList);

bulletCollisionDetector.DetectCollisions(
    _enemyList, (bullet, chopper) =>
{
    var hitEvent = new GameplayEvents.ChopperHitBy(bullet);
    chopper.OnNotify(hitEvent);
    _soundManager.OnNotify(hitEvent);
    bullet.Destroy();
});

missileCollisionDetector.DetectCollisions(_enemyList,
    (missile, chopper) =>
{
    var hitEvent = new GameplayEvents.ChopperHitBy(missile);
    chopper.OnNotify(hitEvent);
    _soundManager.OnNotify(hitEvent);
    missile.Destroy();
});

playerCollisionDetector.DetectCollisions(_playerSprite,
    (chopper, player) =>
{
    KillPlayer();
});
}
```

Сначала мы создаем три отдельных детектора столкновений, по одному для каждого интересующего нас сценария. Затем мы вызываем [DetectCollisions\(\)](#) для каждого из наших сценариев.

- Пули попали в вертолеты
- Ракеты поражают вертолеты
- Вертолеты бьют игрока

Мы также передаем лямбда-функцию в качестве нашего обработчика, чтобы реагировать на столкновения. Когда пули или ракеты попадают в вертолеты, мы генерируем новое игровое событие и уведомляем вертолет, который затем обновляет свое собственное количество жизней. Мы также уведомляем менеджера звука, если у нас есть звуковой эффект, который необходимо воспроизвести. Наконец, вместо того, чтобы удалять из игры столкнувшуюся пушку или снаряд, мы помечаем их как уничтоженные. Метод [CleanObject](#) был обновлен, чтобы также очищать уничтоженные объекты, поэтому помеченные объекты удаляются при вызове [everyUpdate\(\)](#).

Мы также должны реагировать на полное изменение жизни вертолетов. Когда он достигает нуля, мы хотим убрать вертолет и заменить его взрывом. Мы делаем это в следующем методе, который мы связали с событием [OnObjectChanged](#) вертолета:

```
private void _chopperSprite_OnObjectChanged(object sender,
BaseGameStateEvent e)
{
    var chopper = (ChopperSprite)sender;
    switch (e)
    {
        case GameplayEvents.EnemyLostLife ge:
            if (ge.CurrentLife <= 0)
            {
                AddExplosion(new Vector2(chopper.Position.X -
                    40, chopper.Position.Y - 40));
                chopper.Destroy();
            }
    }
}
```

```
        }
        break;
    }
}
```

Здесь мы уничтожаем вертолет, если его общее количество жизней равно нулю или меньше, а это означает, что метод [CleanObjects\(\)](#) позаботится об удалении его из игры при следующем вызове. Затем мы вызываем [AddExplosion\(\)](#), который создаст небольшой взрыв на экране именно там, где раньше был вертолет.

Наконец, функция [KillPlayer\(\)](#) реализована так:

```
private async void KillPlayer()
{
    _playerDead = true;

    AddExplosion(_playerSprite.Position);
    RemoveGameObject(_playerSprite);

    await Task.Delay(TimeSpan.FromSeconds(2));
    ResetGame();
}
```

Игрок помечен как мертвый, чтобы он не стрелял пулями. Взрыв располагается поверх спрайта игрока непосредственно перед удалением игрового объекта из игры. Затем мы даем игроку две секунды, чтобы понять, что только что произошло, и перезагружаем игру, чтобы можно было начать уровень заново.

Загрузите конечное решение [главы 9](#) в Visual Studio и попробуйте!

## Резюме

В этой главе мы рассмотрели несколько различных алгоритмов обнаружения столкновений и то, как мы можем подключить их к коду игрового процесса и реагировать на столкновения. Враги могут быть поражены пулями и ракетами, потерять жизнь и в конечном итоге взорваться огненным облаком.

Мы многое сделали в этой главе, но, к сожалению, не смогли подробно рассмотреть весь код, который был добавлен, например, как вертолеты кратковременно меняют цвет при попадании пуль или как игра перезагружается для перезагрузки уровня. Мы рекомендуем вам исследовать это самостоятельно, и мы считаем, что в настоящее время у вас есть все инструменты, необходимые для реализации этого дома. При разработке видеоигры дьявол кроется в деталях:

мы могли бы потратить больше времени на добавление искровых частиц к вертолетам, когда они попадут в цель. Мы могли бы использовать обнаружение столкновений в стиле ОВВ и вращать вертолеты, когда они движутся по диагонали. Вероятно, нам следует добавить звуковые эффекты, когда пули пролетают мимо или когда они взрываются. Мы также находимся на том этапе, когда нам нужно добавить жизни игроку, чтобы мы могли повторить уровень три раза, прежде чем игра закончится.

Однако дело в том, что теперь у вас есть все инструменты, необходимые для реализации этих деталей. Чего не хватает, так это возможности добавлять текст в игру и отображать оставшиеся жизни игрока. Мы вернемся к этому в следующей главе, но перед этим мы хотели бы перейти к другому интересному предмету: анимации!

## ГЛАВА 10

# Анимация и текст

Если вы не создаете Tetrisclone или текстовую игру, такую как классическая игра Zork, вам понадобятся анимации в вашей видеоигре, чтобы сделать ее более плавной. Даже в игре Space Invader 1978 года была очень простая и грубая анимация, где враги создавали иллюзию движения, переключаясь между двумя спрайтами каждую секунду или около того: в один момент руки инопланетян опущены, а в следующую секунду руки подняты.

Анимации дают нам иллюзию жизни.

Еще одна тема, которая до сих пор отсутствовала в нашей игре, — это текст. Каждая игра имеет ту или иную форму текста. Даже когда в игровом процессе нет текста, что бывает редко, в играх обычно есть какая-то форма системы меню, в которой игрок может выбирать между такими пунктами, как «Начать игру», «Продолжить» или «Выйти». После этого у игроков есть фиксированное количество жизней, и они могут накапливать очки, при этом оба они отображаются на экране.

Рис.10-1 дает нам пример текста, который будет в нашей игре к концу главы.



**Рис. 10-1. Текст!**

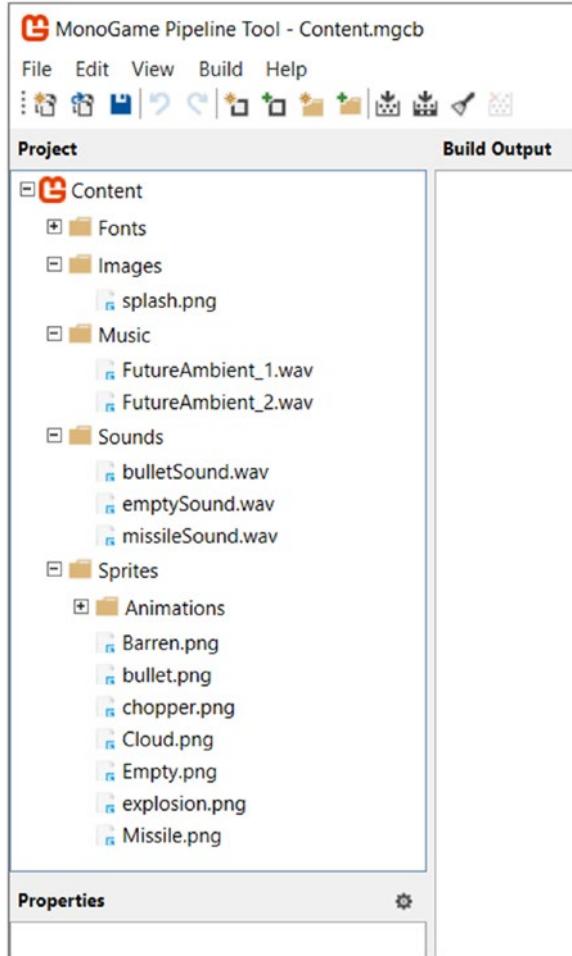
В этой главе вы будете

- Изучать основы анимации
- Добавлять анимацию истребителя при его движении.
- Добавлять в игру текст, отображающий количество оставшихся жизней игрока и наложение Game Over, когда у игрока заканчиваются жизни.

## Немного рефакторинга

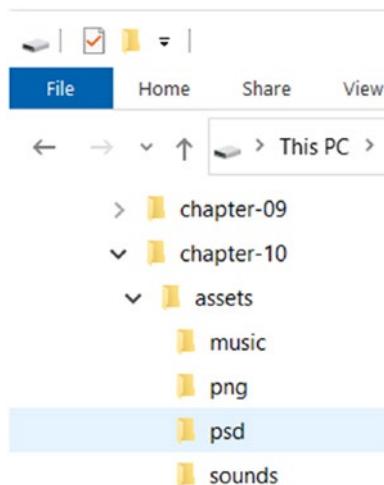
Начальный код для этой главы можно в папке с кодами:  
`monogame-mastery/tree/master/chapter-10/start`.

Первое, что мы сделали перед написанием нового кода для этой главы, — это реорганизовали Content Pipeline. Расположение всех звуков и изображений в одной корневой папке становилось немного запутанным. У нас есть четыре типа активов: спрайты, изображения, музыка и звуки. Мы добавили папку в диспетчере контента для каждого из этих типов и переместили наши активы в правильные папки, как показано на Рис. 10-2.



*Рис. 10-2. Недавно реорганизованный конвейер контента*

На данный момент вы можете игнорировать папку Fonts and Animations, так как они будут добавлены позже в этой главе. Чтобы выполнить эти изменения, мы сначала создали фактические каталоги на жестком диске в нашей папке «Assets - Активы». См. Рис. 10-3.



*Рис. 10-3. Новая физическая организация активов*

Затем мы открыли XML-файл конвейера содержимого, расположенный в [Content\Content.mgcb](#) в [Visual Studio](#), чтобы отредактировать XML-файл напрямую, поскольку пользовательский интерфейс конвейера содержимого не дает нам возможности перемещать активы. Для каждого из наших активов мы обновили местоположение физического актива и его местонахождение в конвейере контента. Например, это

```
#begin ../../assets /FutureAmbient_1.wav
/importer:WavImporter
/processor:SoundEffectProcessor
/processorParam:Quality=Best
/build:../../assets/FutureAmbient_1.wav;FutureAmbient_1.wav
```

был обновлен с соответствующими путями к файлам и расположением содержимого:

```
#begin ../../assets/music/FutureAmbient_1.wav
/importer:WavImporter
/processor:SoundEffectProcessor
```

```
/processorParam:Quality=Best  
/build:../../assets/music/FutureAmbient_1.wav;Music/  
FutureAmbient_1.wav
```

Мы сделали это для всех наших активов, открыли пользовательский интерфейс конвейера контента и перестроили наш конвейер контента, чтобы убедиться, что у нас нет ошибок. Наконец, теперь нам нужно было обновить код игры, чтобы загружать наши звуки, музыку и текстуры из их новых место нахождения. Например:

```
public class GameState : BaseGameState  
{  
    private const string BackgroundTexture = "Sprites/Barren";  
    private const string PlayerFighter = "Sprites/Animations/  
FighterSpriteSheet";  
    private const string BulletTexture = "Sprites/bullet";  
    private const string ExhaustTexture = "Sprites/Cloud";  
    private const string MissileTexture = "Sprites/Missile";  
    private const string ChopperTexture = "Sprites/Chopper";  
    private const string ExplosionTexture = "Sprites/  
explosion";  
  
    private const string TextFont = "Fonts/Lives";  
    private const string GameOverFont = "Fonts/GameOver";  
  
    private const string BulletSound = "Sounds/bulletSound";  
    private const string MissileSound = "Sounds/missileSound";  
  
    private const string Soundtrack1 = "Music/FutureAmbient_1";  
    private const string Soundtrack2 = "Music/FutureAmbient_2";  
}
```

## Анимации

2D-анимации просты и следуют тем же принципам, которые аниматоры использовали с начала двадцатого века, когда изображения отображаются последовательно и достаточно быстро, чтобы создать иллюзию движения. В двухмерных играх программисты загружают в память несколько спрайтов и говорят игре отображать их последовательно, по одному за раз, и контролируют скорость анимации, устанавливая количество кадров или ограничение по времени для каждого спрайта. Посмотрите на Рис. 10-4 все спрайты, необходимые для анимации смерти персонажа-скелета. Пролистывая все эти спрайты, скелет оживает... только для того, чтобы умереть.

Посмотрите на Рис. 10-4, где все спрайты, необходимые для анимации смерти персонажа-скелета. Пролистывая все эти спрайты, скелет оживает... только для того, чтобы умереть.



*Рис. 10-4. Смерть скелета, по одному спрайту за раз*

Насколько плавно выглядит анимация, зависит от того, сколько спрайтов используется и как долго длится анимация. При меньшем количестве кадров требуется более быстрая анимация, чтобы избежать прерывистости, когда игрок может явно заметить каждый кадр. Однако такой эффект может быть желателен в некоторых играх, где, например, персонаж игрока бездействует и подпрыгивает вверх и вниз. Так же, как и пришельцы из Space Invaders, этому бездействующему персонажу может потребоваться всего два спрайта: один для нижней части анимации бобра и один для верхней.

## Листы спрайтов

Если бы мы включили умирающий скелет из Рис. 10-4 в нашу игру и анимировали его, пришлось бы нам добавлять семь спрайтов в конвейер контента, давать им имена и загружать их в нашу игру? К счастью, нет, так как это потребует много работы, потому что это всего лишь одна анимация, где скелет смотрит вправо.

В игре могут быть скелеты, обращенные влево, вверх и вниз, каждый со своей анимацией, всего до 28 спрайтов! Вместо этого разработчики игр используют листы спрайтов, которые очень похожи на наши обычные файлы текстур, но со всеми необходимыми спрайтами игровой анимации в одном файле, обычно с одной строкой на анимацию. Один лист спрайтов, охватывающий все анимации, необходимые для нашего скелета, может иметь строку для каждого из следующих элементов:

- Простой лицом влево
- Простой лицом вправо
- В режиме ожидания лицом вверх
- Бездействие лицом вниз
- Идти налево
- Идти направо
- Пешком до
- Прогулка вниз
- Умереть лицом влево
- Умереть лицом направо
- Умереть лицом вверх
- Умереть лицом вниз

Художник создавал все эти спрайты, помещал их в файл таблицы спрайтов и предоставлял разработчику игры размеры каждого спрайта. С помощью этой информации мы можем затем вычислить местоположение любого спрайта на листе спрайтов. См. Рис. 10-5 для примера большого листа спрайтов, содержащего несколько анимаций. В этом случае каждая строка имеет один тип игрового персонажа, и несколько анимаций находятся в одной строке.

<https://opengameart.org/content/a-platformer-in-the-forest>.

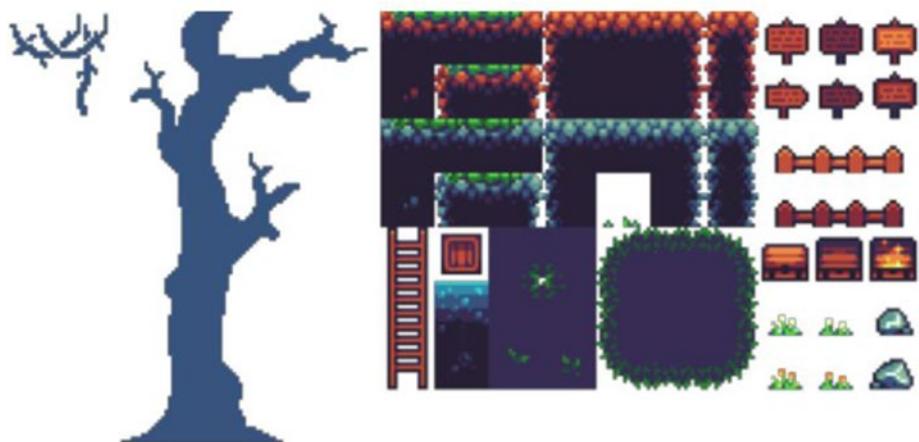
На этой веб-странице мы видим, что художник написал инструкции по поиску спрайтов, необходимых для различных анимаций.



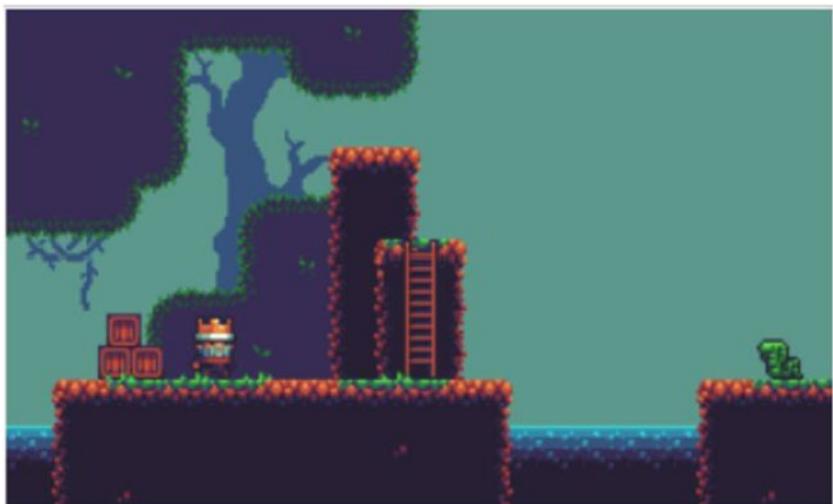
*Рис. 10-5. Большой лист спрайтов со спрайтами для многих анимаций, для любых персонажей*

## Атлас текстур

Атласы текстур похожи на листы спрайтов, и мы должны кратко рассмотреть их здесь, потому что их иногда смешивают. Принцип тот же: создается один файл изображения, содержащий несколько текстур. Основное различие между листом спрайтов и атласом заключается в том, что атлас обычно не используется для анимации. Вместо этого он будет содержать текстуры, необходимые для игры в целом. См. на Рис. 10-6 пример атласа. На Рис. 10-7 показано, как атлас используется для построения мира.



*Рис. 10-6. Атлас игр*



*Рис. 10-7. Использование атласа для построения мира*

## Недостатки анимации

Когда я играю в боевые видеоигры, такие как NieR:Automata, The Witcher 3 или Monster Hunter: World, я предпочитаю быстрый и отзывчивый стиль боя. Все эти игры предлагают игроку выбор между медленным и мощным оружием или быстрым и более слабым оружием, которое наносит меньший урон, чем их тяжелые аналоги, но бьет чаще, что приводит к аналогичному сокрушимому урону. Каждый раз, когда я беру тяжелое оружие в этих играх и нажимаю кнопку атаки, мне приходится смотреть анимацию взмаха оружия, которая занимает много времени. Мой персонаж делает шаг назад, начинает размахивать оружием вокруг себя, а затем изо всех сил старается обрушить оружие на цель. Иногда анимация занимает целых три секунды, что кажется вечностью, и в это время я не могу больше ничего делать.

Эти длинные анимации выявляют один из недостатков техники. В видеоигре может быть сложно плавно перейти от анимации, потому что дизайнерам сложно создавать различные способности и типы движений таким образом, чтобы их можно было смешивать и сопоставлять. Что, если на полпути к длинному замаху мечом я решу начать бежать назад? В реальной жизни это могло бы быть выполнимо: пусть инерция меча развернет меня, пока я приближаю меч к своему телу и, наконец, вставляю его в ножны и начинаю бежать. Однако в видеоиграх геймдизайнеры редко учитывают внезапное изменение движения, поэтому игрокам запрещено делать что-либо подобное. Вы не можете отменить взмах меча, точно так же, как вы не можете обнажить меч, пока он не будет полностью вложен в ножны. Это заставит анимацию извлечения из ножен начинаться с самого начала, когда анимация помещения в ножны находится на полпути, заставляя меч «прыгать» вниз в одном кадре, разрушая иллюзию движения.

Гораздо проще разработать игру, если анимация вынуждена идти до завершения или если отмена анимации для замены ее другой не вызывает ничего странного визуально.

## Конечные автоматы

Конечные автоматы — полезный способ управления, среди прочего, <sup>1</sup>анимацией игровых объектов. Мы не будем использовать конечные автоматы в нашей игре, но мы подумали, что было бы неплохо упомянуть о них, поскольку они являются полезным инструментом, помогающим контролировать и организовывать сложность управления всеми возможными состояниями игровых объектов. Например, свисая с края здания в Assassin's Creed, игрок не может просто начать идти. Сначала он должен либо отпустить руку и упасть, либо взобраться на выступ и встать на крыше. Мы могли бы добавить такой код в наш класс состояния игрового процесса и микроуправлять игровым объектом игрока:

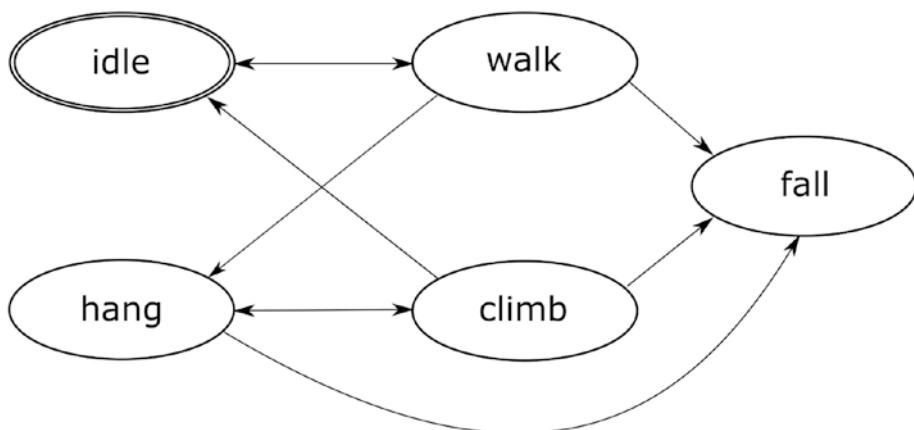
---

<sup>1</sup>Другим применением конечных автоматов может быть управление действиями с командами игрока. Например, нажатие стрелки вниз может означать перемещение персонажа вниз или удаление элемента, в зависимости от того, в каком состоянии находится персонаж.

```
if (cmd is GameplayInputCommand.MoveUp && !player.IsHanging &&
    !player.IsFalling && !player.IsDead)
{
    player.MoveUp();
}
Else if (cmd is GameplayInputCommand.MoveUp && player.IsHanging
&& !player.IsDead)
{
    player.Climb();
}
```

Однако по мере развития игры управление различными способами, которыми игрок может двигаться и взаимодействовать с миром, становится сложным и трудным для понимания. Здесь могут помочь конечные автоматы. Конечный автомат — это набор состояний, соединенных друг с другом направленными ребрами. Если у состояния A есть ребро, ведущее к состоянию B, это означает, что мы можем перейти из A в B. Однако, если состояние B не имеет собственного ребра, ведущего к состоянию B, состояние A, мы не можем вернуться назад.

Если мы посмотрим на наш предыдущий пример с игроком Assassin's Creed, висящим на уступе, мы увидим граф состояний и ребер, изображенный на рис. 10-8. Этот конечный автомат имеет пять различных состояний, и все начинается с того, что игрок изначально находится в состоянии ожидания. Оттуда игрок может только начать ходить, после чего может повиснуть на уступе или упасть. Подвешивая, они могут подниматься или падать.



*Рис. 10-8. Простой конечный автомат*

Каждое состояние в нашем конечном автомате имеет свою собственную связанную анимацию, поэтому, когда приходит время рисовать нашу анимацию, состояние игрового процесса просто должно попросить игровой объект игрока нарисовать анимацию своего текущего состояния, не заботясь о деталях игры.

Хотя мы не будем реализовывать конечный автомат в нашей игре, мы подумали, что стоит упомянуть эту технику, потому что она широко используется в играх. Если вы когда-нибудь решите создать игру с помощью Unity, вам придется использовать конечные автоматы и связывать состояния с анимацией.

## Движок анимации

Любой игровой объект должен иметь только одну активную анимацию за раз или не иметь активной анимации при отображении обычного неподвижного спрайта, что они уже могут делать. Наша стратегия реализации анимации в нашем движке будет состоять в том, чтобы добавить два класса: `AnimationFrame` и `Animation`, и позволить игровым объектам управлять своей собственной анимацией. Хотя было бы полезно добавить в движок конечный автомат, нам не понадобится эта дополнительная сложность для простых анимаций, используемых в вертикальных шутерах.

Начнем с самого простого из двух классов: `AnimationFrame`. Его цель — хранить информацию о каждом кадре анимации:

- Где его спрайт находится на листе спрайтов. Мы будем использовать `aRectangle`, чтобы сохранить это в классе, чтобы упростить рисование спрайта позже, потому что метод `SpriteBatch.Draw` требует `Rectangles`.
- Как долго должен длиться этот кадр, в количестве кадров.

Некоторые механизмы анимации рассчитывают длину каждого кадра анимации в секундах, что полезно при работе с инфраструктурой, которая не гарантирует фиксированное количество кадров в секунду. В нашем случае MonoGame по умолчанию отображает 60 кадров в секунду, поэтому мы можем легко рассуждать о нашей анимации с точки зрения количества кадров. Однако важно отметить, что это просто значение по умолчанию. Некоторые игры целенаправленно работают со скоростью 30 кадров в секунду, и в этом случае длину кадров анимации необходимо обновить, чтобы они работали так же быстро, как игры с частотой 60 кадров в секунду. В других случаях медленное игровое оборудование может помешать MonoGame работать с желаемой скоростью обновления. В этом случае анимации, которые измеряются с использованием времени, а не кадров, будут продолжать работать с той же скоростью и могут пропускать кадры анимации, в то время как анимации, подобные нашей, которые используют подсчет кадров, будут казаться медленнее.

Создайте класс `AnimationFrame` в директории `Engine\Objects\Animations`:

```
public class AnimationFrame
{
    public Rectangle SourceRectangle { get; private set; }
    public int Lifespan { get; private set; }
```

## CHAPTER 10 ANIMATIONS AND TEXT

```
public AnimationFrame(Rectangle sourceRectangle, int lifespan)
{
    SourceRectangle = sourceRectangle;
    Lifespan = lifespan;
}
}
```

Теперь давайте начнем работать над нашим классом Animation, который немного сложнее. Он будет содержать список кадров анимации и сможет указать нашей игре, какой анимационный спрайт нужно отрисовать в любое время, учитывая возраст анимации. Он также должен знать, является ли анимация зацикленной или нет. Например, для бездействующего персонажа, подпрыгивающего вверх и вниз, будет использоваться циклическая анимация, в то время как персонаж, взирающийся на уступ, не будет ее использовать. Наконец, у нас должна быть возможность сбросить анимацию и вернуть ее к кадру номер 1, а возможность создать перевернутую копию анимации может оказаться полезной.

Создайте класс Animation в директории Engine\Objects\Animations со следующими переменными класса:

```
public class Animation
{
    private List<AnimationFrame> _frames = new
        List<AnimationFrame>();
    private int _animationAge = 0;
    private int _lifespan = -1;
    private bool _isLoop = false;
}
```

Здесь у нас есть список `_frames` кадров анимации, возраст анимации и является ли анимация циклом. У нас также есть `_lifespan`, который будет использоваться для хранения общей продолжительности анимации в количестве кадров. Он рассчитывается следующим образом:

```
public int Lifespan {
    get
    {
        if (_lifespan < 0)
        {
            _lifespan = 0;
            foreach (var frame in _frames)
            {
                _lifespan += frame.Lifespan;
            }
        }
        return _lifespan;
    }
}
```

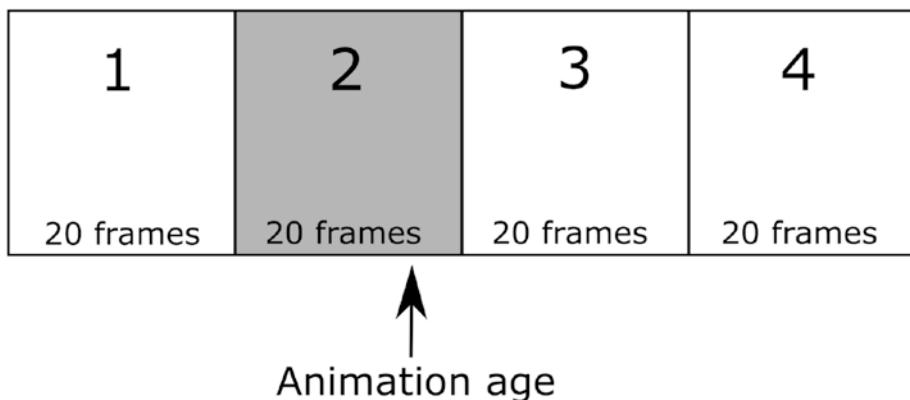
Это свойство берет сумму продолжительности жизни всех кадров анимации и кэширует ее в `_lifespan` перед возвратом. Мы можем кэшировать это значение, потому что однажды созданная анимация уже никогда не изменится.

Получение текущего кадра анимации является ядром этого класса. Алгоритм работает следующим образом:

Мы будем перебирать кадры анимации и добавлять их возраст в переменную-аккумулятор. Когда мы посещаем каждый кадр, если текущий возраст анимации меньше, чем сумма накопления плюс продолжительность жизни текущего кадра, то мы находим наш текущий кадр и возвращаем его. В противном случае увеличиваем аккумулятор на срок жизни текущего кадра и переходим к следующему кадру. Если после перебора всего нашего кадра мы так и не нашли наш текущий кадр, мы просто вернем последний.

На рис. 10-9 показан простой пример анимации с четырьмя кадрами, каждый из которых длится 20 кадров. Если возраст нашей анимации в настоящее время составляет 35 лет, то для вычисления нашего текущего кадра нам нужно пройти наш алгоритм.

- Frame 1: Аккумулятор = 0, поэтому возраст нашей анимации 35 не меньше, чем аккумулятор + 20. Далее!
- Frame 2: Аккумулятор теперь равен 20, что означает, что возраст анимации 35 меньше, чем аккумулятор + 20. Мы нашли наш кадр! Возврат кадра номер 2.



*Рис. 10-9. Вычисление текущего кадра анимации на основе возраста анимации*

Код, реализующий этот алгоритм, выглядит следующим образом:

```
public AnimationFrame CurrentFrame
{
    get
    {
        AnimationFrame currentFrame = null;

        var framesLifespan = 0;
        foreach (var frame in _frames)
```

```
{  
    if (framesLifespan + frame.Lifespan >=  
        _animationAge)  
    {  
        currentFrame = frame;  
        break;  
    }  
    else  
    {  
        framesLifespan += frame.Lifespan;  
    }  
}  
  
if (currentFrame == null)  
{  
    currentFrame = _frames.LastOrDefault();  
}  
  
return currentFrame;  
}  
}  
}
```

При создании экземпляра класса Animation мы должны указать, является ли это зацикленной анимацией:

```
public Animation(bool looping)  
{  
    _isLoop = looping;  
}
```

## CHAPTER 10 ANIMATIONS AND TEXT

Затем наши игровые объекты, которые вскоре будут использовать наш новый класс анимации, могут добавлять кадры в анимацию один за другим с помощью метода AddFrame, указав на листе спрайтов прямоугольник, соответствующий добавляемому кадру, и продолжительность жизни кадра:

```
public void AddFrame(Rectangle sourceRectangle, int lifespan)
{
    _frames.Add(new AnimationFrame(sourceRectangle, lifespan));
}
```

Наконец, нам нужно увеличивать возраст анимации при каждом обновлении. Мы также должны иметь возможность сбросить анимацию, установив ее возраст обратно на ноль.

```
public void Update(GameTime gametime)
{
    _animationAge++;

    if (_isLoop && _animationAge > Lifespan)
    {
        _animationAge = 0;
    }
}

public void Reset()
{
    _animationAge = 0;
}
```

Обратите внимание, что в случае с зацикленной анимацией, когда мы увеличиваем возраст во время метода Update, мы сбрасываем возраст, если возраст анимации превышает ее общее время жизни, что приводит к ее запуску заново.

Наконец, мы добавим нашу вспомогательную функцию для обращения анимации, которая возвращает новую анимацию с перевернутым списком кадров. Это пригодится позже.

```
public Animation ReverseAnimation
{
    get
    {
        var newAnimation = new Animation(_isLoop);
        for (int i = _frames.Count - 1; i >= 0; i--)
        {
            newAnimation.AddFrame(_frames[i].SourceRectangle,
                _frames[i].Lifespan);
        }

        return newAnimation;
    }
}
```

## Анимация нашего истребителя

Теперь мы должны добавить анимацию в игру, и мы считаем, что истребитель нуждается в некотором улучшении, когда он движется влево или вправо. В реальной жизни, когда истребитель движется боком, он начинает наклоняться в этом направлении, заставляя его двигаться именно в этом направлении. Было бы интересно, если бы спрайт наших игроков наклонялся таким же образом, когда он перемещается по нижней части экрана, поэтому мы создали лист спрайтов с несколькими приращениями наклона в обоих направлениях. На [рис. 10-10](#) показан лист спрайтов, который мы создали. Однако вместо того, чтобы создавать две строки для левой и правой анимации, они все должны находиться в одной строке.



*Рис. 10-10. Все кадры нашего движущегося истребителя*

Лист спрайтов находится в каталоге `assets\png` в папке с кодами. Откройте инструмент `Content Pipeline` и добавьте папку `Animations` в папку `Sprites`. Внутри папки `Animations` добавьте ссылку на наш лист спрайтов и назовите его `FighterSpriteSheet.png`. Сохраните и создайте конвейер контента.

Откройте класс `PlayerSprite` в Visual Studio и добавьте следующие закрытые переменные:

```
private Animation _turnLeftAnimation = new Animation(false);
private Animation _turnRightAnimation = new Animation(false);
private Animation _leftToCenterAnimation = new Animation(false);
private Animation _rightToCenterAnimation = new Animation(false);
private const int AnimationSpeed = 3;
private const int AnimationCellWidth = 116;
private const int AnimationCellHeight= 152;

private Animation _currentAnimation;
private Rectangle _idleRectangle;

private bool _movingLeft = false;
private bool _movingRight = false;
```

Здесь мы добавляем несколько анимаций к игровому объекту. Наш истребитель наклоняется влево или вправо. Он также может вернуться в центр, когда игрок перестанет двигаться. Мы также определяем несколько значений, которые будем использовать позже. Анимация наклона имеет скорость 3, которую мы будем использовать в качестве продолжительности жизни каждого кадра анимации позже. Это очень быстро! Однако все, что медленнее, казалось слишком прерывистым, поэтому мы выбрали быструю анимацию вместо анимации с большим количеством промежуточных кадров.

Наши частные переменные также отслеживают высоту и ширину каждого анимационного спрайта на листе спрайтов. Наконец, мы отслеживаем, какая анимация воспроизводится в данный момент, какой спрайт является бездействующим в листе спрайтов и движется ли игрок в настоящее время влево или вправо. Давайте теперь создадим нашу анимацию. Перейдите к конструктору `PlayerSprite` и добавьте следующий код:

```
_idleRectangle = new Rectangle(348, 0, AnimationCellWidth,
    AnimationCellHeight);
_turnLeftAnimation.AddFrame(new Rectangle(348, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
_turnLeftAnimation.AddFrame(new Rectangle(232, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
_turnLeftAnimation.AddFrame(new Rectangle(116, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
_turnLeftAnimation.AddFrame(new Rectangle(0, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);

_turnRightAnimation.AddFrame(new Rectangle(348, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
_turnRightAnimation.AddFrame(new Rectangle(464, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
_turnRightAnimation.AddFrame(new Rectangle(580, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);
```

## CHAPTER 10 ANIMATIONS AND TEXT

```
_turnRightAnimation.AddFrame(new Rectangle(696, 0,
    AnimationCellWidth, AnimationCellHeight),
    AnimationSpeed);

_leftToCenterAnimation = _turnLeftAnimation.ReverseAnimation;
_rightToCenterAnimation = _turnRightAnimation.ReverseAnimation;
```

Предыдущий код устанавливает незанятый прямоугольник, представляющий самолет-истребитель в его неактивном состоянии, который является четвертым спрайтом в нашем листе спрайтов. Координаты в прямоугольнике точно соответствуют расположению спрайта на листе спрайтов. Затем мы создаем `_turnLeftAnimation`, добавляя первые четыре спрайта на лист спрайтов. За этим следует создание `_turnRightAnimation`, которое добавляет последние четыре спрайта на лист спрайтов.

Наконец, чтобы создать анимацию для возврата самолета в исходное положение, мы просто меняем местами две созданные нами анимации. Теперь у нас есть четыре анимации для нашего истребителя. Давайте использовать их!

В нашем объекте `PlayerSprite` уже есть методы `MoveLeft()` и `MoveRight()` для изменения положения самолета в игре. Мы будем повторно использовать эти методы, чтобы также изменить текущую анимацию объекта. Мы также добавим метод `StopMoving()`, чтобы иметь возможность воспроизводить соответствующую обратную анимацию.

```
public void StopMoving()
{
    if (_movingLeft)
    {
        _currentAnimation = _leftToCenterAnimation;
        _movingLeft = false;
    }

    if (_movingRight)
    {
        _currentAnimation = _rightToCenterAnimation;
    }
}
```

```
        _movingRight = false;
    }
}

public void MoveLeft()
{
    _movingLeft = true;
    _movingRight = false;
    _currentAnimation = _turnLeftAnimation;
    _leftToCenterAnimation.Reset();
    _turnRightAnimation.Reset();
    Position = new Vector2(Position.X - PlayerSpeed, Position.Y);
}

public void MoveRight()
{
    _movingRight = true;
    _movingLeft = false;
    _currentAnimation = _turnRightAnimation;
    _rightToCenterAnimation.Reset();
    _turnLeftAnimation.Reset();
    Position = new Vector2(Position.X + PlayerSpeed, Position.Y);
}
```

В `MoveRight()` происходит несколько вещей. Во-первых, мы соответствующим образом устанавливаем булевые переменные `_movingRight` и `_movingLeft`, чтобы наш объект знал, что в данный момент он движется вправо. Затем мы устанавливаем `_currentAnimation` на `_turnRightAnimation`. Поскольку мы знаем, что в данный момент не используем другие анимации, мы сбрасываем их, чтобы их можно было повторно использовать позже. Наконец, мы обновляем положение объекта на экране. Метод `MoveLeft()` работает аналогичным образом.

## CHAPTER 10 ANIMATIONS AND TEXT

Однако метод `StopMoving()` должен знать, в какую сторону поворачивалась плоскость, чтобы вернуть ее в центр. Если он двигался влево, то мы используем `_leftToCenterAnimation`. В противном случае мы используем `_rightToCenterAnimation`. Наконец, поскольку мы больше не движемся, были установлены обе булевые переменные движения.

Это немного отслеживаемое состояние, но все же управляемое. Однако, если бы мы начали добавлять больше функциональности к нашему игровому объекту-истребителю, например, перемещение по вертикали, нам, возможно, пришлось бы подумать об использовании вместо этого конечного автомата.

Осталось совсем немного, чтобы наш новый истребитель можно было использовать. Нам нужен метод `Update()`, чтобы мы могли увеличить возраст текущей анимации, и нам также нужно найти кадр текущей анимации, чтобы мы могли его отрисовать. Или, если текущей анимации нет из-за того, что игрок перестал двигаться, рисуем спрайт бездействия.

```
public void Update(GameTime gametime)
{
    if (_currentAnimation != null)
    {
        _currentAnimation.Update(gametime);
    }
}

public override void Render(SpriteBatch spriteBatch)
{
    var destinationRectangle =
        new Rectangle((int)_position.X, (int)_position.Y,
                      AnimationCellWidth, AnimationCellHeight);
    var sourceRectangle = _idleRectangle;
    if (_currentAnimation != null)
    {
        var currentFrame = _currentAnimation.CurrentFrame;
        if (currentFrame != null)
```

```

    {
        sourceRectangle = currentFrame.SourceRectangle;
    }
}

spriteBatch.Draw(_texture, destinationRectangle,
sourceRectangle, Color.White);
}

```

И вот оно. Вы можете поиграть с этим новым классом [FighterSprite](#) в классе [DevState](#) и посмотреть, как он работает при поворотах влево и вправо.

Класс [GameplayState](#) был изменен с учетом новой команды [GameplayInputCommand](#):

```
public class PlayerStopsMoving : GameplayInputCommand { }
```

Эта команда выдается, когда игрок не нажимает клавиши со стрелками влево или вправо в нашем преобразователе ввода, который был изменен для этого:

```

if (state.IsKeyDown(Keys.Right))
{
    commands.Add(new GameplayInputCommand.PlayerMoveRight());
}
else if (state.IsKeyDown(Keys.Left))
{
    commands.Add(new GameplayInputCommand.PlayerMoveLeft());
}
else
{
    commands.Add(new GameplayInputCommand.PlayerStopsMoving());
}

```

С учетом этого метод `HandleInput()` в классе `GameplayState` может отслеживать эту команду и реагировать соответствующим образом:

```
if (cmd is GameplayInputCommand.PlayerStopsMoving &&
!_playerDead)
{
    _playerSprite.StopMoving();
}
```

Наконец, метод `UpdateGameState()` объекта `GameplayState` должен вызывать метод `Update()` класса `PlayerSprite`. Это делается с помощью этого простого вызова:

```
_playerSprite.Update(gameTime);
```

## Текст

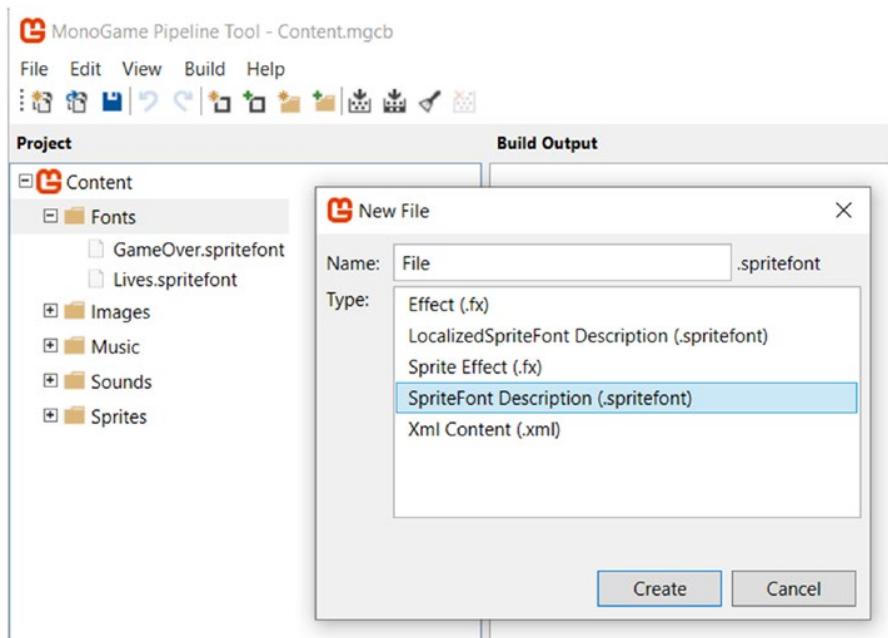
Текст является важной частью игр во всем мире. Он используется для различных целей, таких как отображение счета игрока, письма, полученного игроком в ММО, или меню на экране, позволяющее игроку изменить свои графические настройки. Мы будем использовать текст в нашей игре, чтобы сообщить игроку, сколько жизней у него осталось, и дать ему экран Game Over, когда у него закончатся жизни.

## Шрифты

Для отображения текста на экране нужны спрайты. К счастью, нам не нужно создавать собственные атласы спрайтов, содержащие все шрифтовые символы, необходимые для нашей игры. MonoGame позаботится об этом за нас, растеризовав шрифты, которые уже есть на наших компьютерах. Если вы можете использовать определенный шрифт при редактировании текста в документе Word, то этот шрифт доступен для MonoGame, которая будет рад создать для вас шрифтовые спрайты.

## Добавление шрифтов в конвейер контента

Создание спрайтов шрифтов для нашей игры довольно просто. Мы будем использовать инструмент Content Pipeline, чтобы добавить наши игровые шрифты, но нам понадобится один дополнительный шаг, чтобы правильно построить наши шрифты. Откройте [Content Pipeline Tool](#) и добавьте папку Fonts в Contents. Как показано на Рис. 10-11, в папке Fonts создайте два новых элемента и выберите параметр SpriteFont Description. Назовите каждый предмет GameOver и Lives. Причина, по которой нам нужны два разных шрифта, заключается в том, что игра над текстом будет больше, чем живой текст, поэтому нам нужно независимо растировать два шрифта в два разных спрайт-шрифта. Обратите внимание, что в конвейере содержимого существует еще один вариант шрифта спрайта для локализованного текста. Это используется, когда игры продаются на международном уровне, и текст должен меняться в зависимости от региона мира, в котором живет игрок. Мы не будем рассматривать этот сценарий в этой главе.



**Рис. 10-11.** Добавление *SpriteFonts* в конвейер контента

Сохраните конвейер содержимого, но пока не закрывайте окно. Мы только что создали два файла в нашем игровом решении, расположенные в [Content\Fonts\](#) с именами [GameOver.spritefont](#) и [Lives.spritefont](#). Откройте оба файла в [VisualStudio](#). Как видите, оба файла также являются XML-файлами и могут редактироваться. Обратите внимание, что в шрифте [Lives.sprite](#) шрифт установлен на Arial в строке 14:

```
<FontName>Arial</FontName>
```

Не стесняйтесь экспериментировать с разными шрифтами, но пока мы решили использовать шрифт по умолчанию. Однако нам нужно изменить размер шрифта на 14, изменив эту строку в файле:

```
<Size>14</Size>
```

Сохраните файл, откройте [GameOver.spritefont](#) и измените размер шрифта в этом файле на 50, а затем сохраните и закройте оба файла. Теперь вы можете запустить сборку в инструменте конвейера контента.

## Шрифты как игровые объекты

Мы будем рассматривать текст как игровые объекты, потому что, как и другие игровые объекты, они рисуются на экране и имеют положение. Давайте добавим в наш движок новый класс в директорию [Engine\Objects](#) с именем [BaseTextObject](#):

```
public class BaseTextObject : BaseGameObject
{
    protected SpriteFont _font;

    public string Text { get; set; }

    public override void Render(SpriteBatch spriteBatch)
    {
        spriteBatch.DrawString(_font, Text, _position,
            Color.White);
    }
}
```

Этот класс наследуется от `BaseGameObject`, предоставляя ему всю функциональность `GameObject`, но мы будем игнорировать приватную переменную `_texture` в базовом классе. Вместо этого мы будем отслеживать шрифт спрайта, используемый каждым текстовым объектом, и предоставим метод рендеринга по умолчанию, который использует новый метод `SpriteBatch`, называемый `DrawString()`, который используется для рисования текста с использованием шрифта спрайта, строки и позиции.

Давайте добавим два наших текстовых объекта в нашу игру. Добавьте в директорию `Objects\Text` два класса — `GameOverText` и `LivesText`:

```
public class GameOverText : BaseTextObject
{
    public GameOverText(SpriteFont font)
    {
        _font = font;
        Text = "Game Over";
    }
}

public class LivesText : BaseTextObject
{
    private int _nbLives = -1;

    public int NbLives {
        get
        {
            return _nbLives;
        }
        set
        {
            _nbLives = value;
            Text = $"Lives: {_nbLives}";
        }
    }
}
```

```
public LivesText(SpriteFont font)
{
    _font = font;
}
}
```

Класс `GameOverText` очень прост и содержит текст «GameOver».

Класс `LivesText`, с другой стороны, несет дополнительную ответственность за обновление своего текста в зависимости от количества оставшихся жизней игрока. Когда игрок теряет жизнь, `GameplayState` устанавливает свойство `NbLives`, что приводит к тому, что отображаемый текст обновляется.

## Отслеживание жизней

Давайте теперь отследим жизнь нашего игрока. Когда игра начнется, у нашего игрока будет три жизни. Добавьте эти частные переменные в класс `GameplayState`:

```
private const int StartingPlayerLives = 3;
private int _playerLives = StartingPlayerLives;

private const string TextFont = "Fonts/Lives";
private const string GameOverFont = "Fonts/GameOver";
private LivesText _livesText;
```

Здесь мы храним наше начальное количество жизней, расположение шрифтов спрайтов и экземпляр класса `LivesText`, экземпляра которого создается в методе `LoadContent()`:

```
_livesText = new LivesText(LoadFont(TextFont));
_livesText.NbLives = StartingPlayerLives;
_livesText.Position = new Vector2(10.0f, 690.0f);
AddGameObject(_livesText);
```

В предыдущем коде используется новая функция `LoadFont()`, добавленная в класс `BaseGameState`, который использует функцию `Load` менеджера контента для классов `SpriteFont`:

```
protected SpriteFont LoadFont(string fontName)
{
    return _contentManager.Load<SpriteFont>(fontName);
}
```

С этими элементами в игре и отрисовкой нашего текстового объекта `LivesText` мы готовы уменьшить количество жизней всякий раз, когда вызывается метод `KillPlayer()`:

```
_playerLives -= 1;
_livesText.NbLives = _playerLives;
if (_playerLives > 0)
{
    ResetGame();
}
else
{
    GameOver();
}
```

Этот код уменьшает `_playerLives` на единицу и обновляет количество жизней в объекте `_livesText`. Если это число больше нуля, мы сбрасываем игру. В противном случае мы вызываем метод `GameOver()`, который вскоре реализуем.

## Game Over

Мы могли бы просто отображать большой текст «Игра окончена» в середине экрана, когда у игрока заканчиваются жизни, но это немного скучно, поэтому мы добавим дополнительные штрихи, а также немного затемним экран, чтобы по-настоящему донести мысль, что игра действительно окончена. Во-первых, давайте реализуем метод `GameOver()`. Начните с добавления этой закрытой переменной в класс `GameplayState`, чтобы отслеживать, окончена игра или нет:

```
private bool _gameOver = false;
```

Then add the GameOver() method:

```
private void GameOver()
{
    var font = LoadFont(GameOverFont);
    var gameOverText = new GameOverText(font);
    var textPositionOnScreen = new Vector2(460, 300);

    gameOverText.Position = textPositionOnScreen;
    AddGameObject(gameOverText);
    _gameOver = true;
}
```

Аналогично тому, как мы добавляем объект [LivesText](#) в игру, мы загружаем шрифт GameOver и создаем экземпляр объекта GameOverText, который затем размещается в центре экрана перед добавлением к активным игровым объектам. Наконец, мы устанавливаем для переменной `_gameOver` значение `true`, что позволит затемнить экран.

Выбранная стратегия немного затемнить экран состоит в том, чтобы нарисовать большой прямоугольник, который заполняет область просмотра, используя черный цвет и настройку прозрачности 30%. Для этого мы переопределим метод [Render\(\)](#) класса [BaseGameState](#) и сами нарисуем наши игровые объекты, после чего нарисуем эти полупрозрачные темные прямоугольники, чтобы добиться желаемого эффекта, если игра окончена. Во-первых, измените метод [Render\(\)](#) [BaseGameState](#) на виртуальный, чтобы его можно было переопределить:

```
public virtual void Render(SpriteBatch spriteBatch)
```

Затем добавьте следующий код в класс `GameplayState`:

```
public override void Render(SpriteBatch spriteBatch)
{
    base.Render(spriteBatch);

    if (_gameOver)
    {
        // Draw black rectangle at 30% transparency
        var screenBoxTexture = GetScreenBoxTexture(spriteBatch.
GraphicsDevice);
        var viewportRectangle = new Rectangle(0, 0,
        _viewportWidth, _viewportHeight);
        spriteBatch.Draw(screenBoxTexture, viewportRectangle,
        Color.Black * 0.3f);
    }
}

private Texture2D GetScreenBoxTexture(GraphicsDevice
graphicsDevice)
{
    if (_screenBoxTexture == null)
    {
        _screenBoxTexture = new Texture2D(graphicsDevice, 1, 1);
        _screenBoxTexture.SetData<Color>(new Color[] { Color.
White });
    }
    return _screenBoxTexture;
}
```

Метод `Render()` класса `GameplayState` начинается с вызова метода базового класса, который мы переопределяем. Он заботится о рисовании всех активных игровых объектов на экране. Затем, если игра окончена, мы создаем пустую текстуру, которую кэшируем для будущего использования. Текстура белая, но она быстро меняется на полупрозрачную черную в функции `spriteBatch.Draw()` путем умножения значения `Color.Black` на 0,3.

Теперь, когда игрок теряет три жизни, экран темнеет, отображается текст игры, и игра может продолжаться без истребителя, пока игрок не нажмет клавишу выхода, чтобы выйти из игры.

## Резюме

В этой главе мы знакомим наших читателей с тем, как MonoGame обрабатывает текст, позволяя отображать информацию на экране и открывая будущие возможности, такие как экран паузы и экраны меню. Мы также рассмотрели 2D-анимации и то, как добавить их в нашу игру, уделив некоторое время изучению конечных автоматов.

Мы считаем, что к этому моменту книги у вас есть все инструменты, необходимые для создания игр. Но, как вы могли заметить в этом путешествии, некоторые аспекты программирования игр утомительны. У нас также нет полной игры прямо сейчас. Чего не хватает на данный момент, так это надлежащего уровня для прохождения.

В следующей главе мы рассмотрим внешние игровые инструменты, которые помогут ускорить разработку игр. Мы также будем работать над уровнем для прохождения и создадим редактор уровней, чтобы мы могли создавать будущие уровни.

## ГЛАВА 11

# Создание уровня

На этом этапе книги у вас есть все инструменты, необходимые для создания собственных 2D-игр. Но, как вы могли заметить в этом путешествии, некоторые аспекты программирования игр утомительны, а дьявол кроется в деталях. Несмотря на все наши усилия до сих пор, у нас все еще нет полной игры, не говоря уже о полном уровне для прохождения. Все, что у нас есть в игре, это игрок, у которого есть три попытки уничтожить четырех врагов-вертолетов, а затем фон просто продолжает прокручиваться, пока игрок не выйдет из игры. Нет никакого вызова и никакого чувства вознаграждения.

В этой главе вы будете

- Создавать свой собственный игровой уровень
- Загружать игровой уровень в игру
- Добавлять турели в качестве нового и очень сложного врага

Рис. 11-1 — это то, что можете ожидать, как игра будет выглядеть к тому времени, когда вы закончите работу над этой главой.



*Рис. 11-1. Наш финал*

## Редакторы уровней

Редакторы уровней — один из самых мощных инструментов для игрового дизайна. С их помощью вы можете взять идеи, которые мы реализовали, игровую механику и наши игровые объекты, и расположить их на уровне таким образом, чтобы создать вызов для игрока и сделать игру увлекательной.

Возможно, вы слышали о том, что мы считаем сейчас самым популярным редактором уровней в мире: [Super Mario Maker](#), который позволяет игрокам со всего мира проектировать и создавать уровни Super Mario, которые они затем могут загрузить в Интернет, чтобы другие игроки могли попробовать их.

В начале 1990-х годов одной из самых популярных игр того времени была Software's Doom, классический шутер от первого лица, который действительно сделал весь жанр мейнстрилом. Вскоре после выпуска редактор уровней Doom стал доступен для скачивания, и тысячи энтузиастов Doom начали создавать свои собственные уровни, а затем играть в них.

Эти редакторы были искушенными. Они предоставили пользователям графический интерфейс для построения уровней. С редактором уровней Doom мы потратили часы, соединяя линии вместе, чтобы сформировать стены и назначая свойства каждой стене, чтобы придать ей текстуру или отметить двери. С помощью Mario Maker игрок размещает игровые объекты на 2D-уровне и может визуально видеть, что они строят. В итоге игра выглядит именно так, как видит пользователь в редакторе. Это требует нового уровня усилий, который включает в себя создание отдельного приложения (или приложения внутри игры), чтобы позволить пользователям размещать игровые объекты в режиме «что видишь, то и получаешь».

Наша игра, однако, не очень сложна. Чтобы спроектировать наши уровни, мы можем расположить врагов и препятствия в виде сетки в текстовом документе. Хотя это и не так наглядно, как редактор уровней Doom, мы все же можем расположить вещи так, чтобы было легко мысленно визуализировать игрока, бегущего по нашему конструкции.

## Что такое уровень?

Если мы подумаем о той игре, которую мы создаем, то это набор фоновых плиток, размещенных на сетке, с генерируемыми в определенное время вертолетами. Позже мы добавим в игру турели, и они будут размещены на тайлах фона, поэтому организация нашего уровня в виде длинной вертикальной сетки с игроком, начинающим снизу, имеет смысл. Когда игра начинается, она считывает первую строку сетки, ту, что внизу, интерпретирует ее и размещает предметы в игре. Затем он читает следующий ряд над ним и размещает на экране другие элементы, и так далее, пока не закончатся ряды для чтения, и в этот момент игрок завершает уровень.

Посмотрите на этот фрагмент уровня, который мы разработали для игры на [Рис.11-2](#). Представьте, что спрайт игрока находится посередине внизу изображения и медленно движется вверх. Каждая строка сетки состоит из одиннадцати элементов: десять элементов игровых объектов, за которыми следует один последний символ, представляющий глобальное событие. Полный текстовый файл уровня можно найти в папке с кодами: monogame-mastery/blob/master/chapter-11/end/Levels/LevelData/Level1.txt.

## ГЛАВА 11 LEVEL DESIGN

```
0,0,0,1,0,0,1,0,0,0,_  
0,0,1,0,0,0,0,1,0,0,g2  
0,0,0,0,0,0,0,0,0,0,_  
0,0,0,0,0,0,0,0,0,0,g2  
0,0,1,0,0,0,0,1,0,0,_  
0,0,0,0,0,0,0,0,0,0,g2  
0,1,0,0,0,0,0,0,0,0,_  
0,0,0,0,0,0,0,0,0,0,g4  
0,0,0,0,0,0,0,0,0,0,_  
0,0,0,0,0,0,0,0,0,0,g2  
0,0,0,0,0,0,0,0,0,s
```

*Рис. 11-2. Фрагмент уровня, преобразованный в текст*

Наша сетка имеет ряды из десяти элементов, которые представляют игровые объекты, расположенные на экране. В конце каждой строки также есть одиннадцатый элемент, представляющий общигровые события, такие как генерация врагов. Посмотрите на следующую надпись для значения каждого элемента:

- 0: Здесь ничего не происходит. Не добавляйте ничего в игру в этой области сетки.
- 1: Добавить турель в этом месте на экране.
- \_: Эта строка уровня не запускает никаких игровых событий.
- gN: буква «g», за которой следует число, запускает генерацию N вражеских вертолетов всякий раз, когда эта строка оценивается игрой.

- s: Указывает на начало уровня, позволяя игре позже пожелать удачи игроку.
- e: указывает на конец уровня, позволяя игре позже поздравить игрока.

По мере оценки каждой строки игрового уровня окно просмотра будет разделено на десять равных секций сетки. Когда игра решит, что турель необходимо разместить, она будет размещена чуть выше верхней части экрана в соответствующем горизонтальном положении на сетке, чтобы ее можно было медленно прокручивать вниз в игре.

Как быстро мы должны выполнять каждую строку? Это полностью зависит от геймдизайнера, но мы выбрали двухсекундную скорость чтения. Наш текущий уровень, как и было задумано, имеет в общей сложности 48 линий, поэтому прохождение всего уровня займет 96 секунд, что кажется правильным для общей продолжительности, необходимой для этого типа игры.

Этот файл может быть отредактирован в любое время разработчиком игры. Количество сгенерированных прерывателей можно изменить, или же можно изменить время их создания, заменив элемент «gN» на «\_», чтобы удалить генерацию прерывателей в этой строке уровня. Точно так же «\_» можно заменить элементами «gN», чтобы добавить генерацию прерывателей. Перемещать башни также легко, меняя «0» на «1» в нашей сетке. Всякий раз, когда игра перекомпилируется и запускается, игра будет воспроизводить только что отредактированный уровень.

## События уровня

Когда мы читаем строки уровня из текстового файла уровня, мы будем генерировать события уровня, на которые может регистрироваться любой объект состояния игры. У нас будет по одному событию для каждого типа ситуаций, описанных в предыдущем разделе. Давайте начнем добавлять код! Создайте каталог Levels в проекте и добавьте файл LevelEvents.cs со следующим кодом:

## ГЛАВА 11 LEVEL DESIGN

```
public class LevelEvents : BaseGameStateEvent
{
    public class GenerateEnemies : LevelEvents
    {
        public int NbEnemies { get; private set; }
        public GenerateEnemies(int nbEnemies)
        {
            NbEnemies = nbEnemies;
        }
    }

    public class GenerateTurret : LevelEvents
    {
        public float XPosition { get; private set; }
        public GenerateTurret(float xPosition)
        {
            XPosition = xPosition;
        }
    }

    public class StartLevel : LevelEvents { }
    public class EndLevel : LevelEvents { }
    public class NoRowEvent : LevelEvents { }
}
```

Эти события будут запускаться классом `Level` ниже, поскольку он интерпретирует каждый элемент каждой строки в нашей сетке уровней. Эти события позволяют нам сгенерировать любое количество вражеских вертолетов или расположить турели в определенной координате `X` над окном просмотра.

## Читатели уровней, уровни и состояние нашего игрового процесса

Нам нужна возможность читать текстовые файлы нашего уровня, но прежде чем мы это сделаем, мы должны создать наш первый уровень. Вы можете либо скопировать файл level.txt по указанному ранее URL-адресу, либо создать новый для своих нужд. В любом случае поместите его в `Levels\LevelData\Level1.txt` и убедитесь, что он добавлен в проект в Visual Studio. Затем откройте свойства файла в Visual Studio и убедитесь, что для параметра «Build Action - Действие сборки» установлено значение «Embedded Resource - Встроенный ресурс», как показано на рис. 11-3.



*Рис. 11-3. Level1.txt — это встроенный ресурс.*

Наличие наших уровней в качестве встроенного ресурса — это хорошо, потому что внутреннее представление наших уровней не будет напрямую доступно для пользователей, которые могут сами пойти и отредактировать файлы и потенциально испортить игру, перезаписав исходные файлы чем-то, что либо сломает игру, либо разрушит дизайн каждого уровня. При встраивании файлов уровни становятся ресурсами, которые находятся непосредственно в исполняемом файле и к которым намного сложнее получить доступ.

Теперь мы можем добавить класс `LevelReader`, отвечающий за загрузку файла и преобразование его в сетку уровней. Добавьте класс в директорию `Levels`:

```
public class LevelReader
{
    private int _viewportWidth;
```

## ГЛАВА 11 LEVEL DESIGN

```
private const int NB_ROWS = 11;
private const int NB_TILE_ROWS = 10;

public LevelReader(int viewportWidth)
{
    _viewportWidth = viewportWidth;
}

}
```

Мы создаем экземпляр класса с параметром `viewportWidth`, который будет использоваться позже для преобразования номера сетки в координату X, соответствующую тому, где на экране должен находиться элемент строки. У нас также есть две константы, которые показывают, что наша сетка состоит из одиннадцати элементов, десять из которых предназначены для размещения игровых объектов на экране.

Чтобы преобразовать каждый элемент строки в игровое событие, мы будем использовать эту функцию:

```
private BaseGameStateEvent ToEvent(int elementNumber, string
input)
{
    switch (input)
    {
        case "0":
            return new BaseGameStateEvent.Nothing();

        case "_":
            return new LevelEvents.NoRowEvent();

        case "1":
            var xPosition = elementNumber * _viewportWidth /
NB_TILE_ROWS;
            return new LevelEvents.GenerateTurret(xPosition);

        case "s":
            return new LevelEvents.StartLevel();
    }
}
```

```

    case "e":
        return new LevelEvents.EndLevel();

    case string g when g.StartsWith("g"):
        var nb = int.Parse(g.Substring(1));
        return new LevelEvents.GenerateEnemies(nb);

    default:
        return new BaseGameStateEvent.Nothing();
    }
}

```

Функция принимает строку, представляющую один элемент строки, и положение этого элемента в строке. Если это «**0**» или «**\_**», мы возвращаем соответствующее событие, чтобы указать, что ничего не происходит. Основной причиной использования здесь двух отдельных событий является разница между тем, чтобы ничего не добавлять на экран, и глобальным событием, которое можно использовать для отмены предыдущего глобального события. Например, событие начального уровня «**s**» отменяется, когда инициируется следующее событие **NoRowEvent**. Если наш символ равен «**1**», то мы вычисляем координату X текущего элемента строки и передаем ее в событие. Если наш символ — «**s**» или «**e**», то мы запускаем события **StartLevel** или **EndLevel**. Наконец, когда строка начинается с «**g**», мы читаем остальную часть строки и преобразовываем ее в целое число, которое говорит нам, сколько прерывателей нужно создать.

Следующая функция будет отвечать за перебор каждой строки и извлечение из нее всех событий уровня. Она возьмет строку уровня и разделит ее запятыми, затем переберет каждый элемент и преобразует его в событие, прежде чем вернуть список событий для всей строки:

```

private List<BaseGameStateEvent> ToEventRow(string rowString)
{
    var elements = rowString.Split(',');
    var newRow = new List<BaseGameStateEvent>();

```

## ГЛАВА 11 LEVEL DESIGN

```
for (int i = 0; i < NB_ROWS; i++)
{
    newRow.Add(ToEvent(i, elements[i]));
}

return newRow;
}
```

Наконец, мы добавим общедоступную функцию, которая будет загружать любой уровень по его номеру, находя встроенный текстовый файл внутри сборки, считывая его в память и преобразовывая в сетку событий уровня.

```
public List<List<BaseGameStateEvent>> LoadLevel(int nb)
{
    var assembly = Assembly.GetExecutingAssembly();
    var assemblyName = assembly.FullName.Split(',')[0];
    var fileName = $"{assemblyName}.Levels.LevelData.Level{nb}.
txt";

    var stream = assembly.GetManifestResourceStream(fileName);
    var reader = new StreamReader(stream);
    var levelString = reader.ReadToEnd();

    var rows = levelString.Split(Environment.NewLine.
        ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
    var convertedRows = from r in rows
                        select ToEventRow(r);

    return convertedRows.Reverse().ToList();
}
```

Поскольку текстовый файл читается в памяти сверху вниз и мы должны выполнить уровень в обратном направлении, мы переворачиваем список в конце, чтобы события могли воспроизвестись в соответствующем порядке.

Теперь мы можем читать текстовые файлы уровней и преобразовывать их в сетку событий уровней. Все, что осталось реализовать, — это класс `Level`, который будет принимать сетку событий уровня и считывать каждую строку событий каждые две секунды для нашего игрового состояния. Создайте класс `Level` в той же директории, где мы разместили наши классы `LevelReader` и `LevelEvents`:

```
public class Level
{
    private LevelReader _levelReader;
    private List<List<BaseGameStateEvent>> _currentLevel;
    private int _currentLevelNumber;
    private int _currentLevelRow;

    private TimeSpan _startGameTime;
    private readonly TimeSpan TickTimeSpan = new TimeSpan(0,
        0, 2);

    public event EventHandler<LevelEvents.GenerateEnemies>
        OnGenerateEnemies;
    public event EventHandler<LevelEvents.GenerateTurret>
        OnGenerateTurret;
    public event EventHandler<LevelEvents.StartLevel>
        OnLevelStart;
    public event EventHandler<LevelEvents.EndLevel> OnLevelEnd;
    public event EventHandler<LevelEvents.NoRowEvent>
        OnLevelNoRowEvent;
```

## ГЛАВА 11 LEVEL DESIGN

```
public Level(LevelReader reader)
{
    _levelReader = reader;
    _currentLevelNumber = 1;
    _currentLevelRow = 0;

    _currentLevel = _levelReader.LoadLevel(
        _currentLevelNumber);
}

}
```

Класс `Level` имеет доступ к считывателю уровня, который передается как параметр через конструктор, который устанавливает начальный номер текущего уровня в 1 и строку текущего уровня в 0, а затем использует считыватель для загрузки уровня 1. Самое главное, константа `TickTimeSpan` установлен на промежуток времени в две секунды, который будет использоваться ниже, чтобы определить, как долго ждать, пока мы не сможем прочитать следующую строку событий уровня.  
Класс также предлагает некоторые полезные функции, такие как сброс уровня, когда игрок умирает, или загрузка следующего уровня для нас:

```
public void LoadNextLevel()
{
    _currentLevelNumber++;
    _currentLevel = _levelReader.LoadLevel(
        _currentLevelNumber);
}

public void Reset()
{
    _currentLevelRow = 0;
}
```

Наконец, основной метод следующего класса считывает события уровня текущей строки уровня, если прошло полных две секунды. Для каждого события он запускает событие .Net, на которое будет зарегистрирован класс состояния игрового процесса:

```
public void GenerateLevelEvents(GameTime gameTime)
{
    // Только генерировать события каждые 2 секунды
    if (_startGameTime == null)
    {
        _startGameTime = gameTime.TotalGameTime;
    }

    // Ничего не делать до времени тика
    if (gameTime.TotalGameTime - _startGameTime < TickTimeSpan)
    {
        return;
    }

    _startGameTime = gameTime.TotalGameTime;

    foreach (var e in _currentLevel[_currentLevelRow])
    {
        switch (e)
        {
            case LevelEvents.GenerateEnemies g:
                OnGenerateEnemies?.Invoke(this, g);
                break;

            case LevelEvents.GenerateTurret g:
                OnGenerateTurret?.Invoke(this, g);
                break;

            case LevelEvents.StartLevel s:
                OnLevelStart?.Invoke(this, s);
                break;
        }
    }
}
```

```

        case LevelEvents.EndLevel s:
            OnLevelEnd?.Invoke(this, s);
            break;

        case LevelEvents.NoRowEvent n:
            OnLevelNoRowEvent?.Invoke(this, n);
            break;
    }

}

_currentLevelRow++;
}

```

Теперь мы должны загрузить наш первый уровень, хотя мы пока не будем обрабатывать турели. Откройте класс `GameState` и добавьте приватную переменную для нашего уровня:

```
private Level _level;
```

Затем обновите метод `LoadContent()`, чтобы создать уровень:

```

var levelReader = new LevelReader(_viewportWidth);
_level = new Level(levelReader);

_level.OnGenerateEnemies += _level_OnGenerateEnemies;
_level.OnGenerateTurret += _level_OnGenerateTurret;
_level.OnLevelStart += _level_OnLevelStart;
_level.OnLevelEnd += _level_OnLevelEnd;
_level.OnLevelNoRowEvent += _level_OnLevelNoRowEvent;

```

Давайте теперь создадим каждый обработчик событий, который мы использовали для регистрации игровых событий. Мы пока оставим большинство из них пустыми, чтобы мы могли сосредоточиться на правильном выполнении наших уровней. Единственное событие, которое мы можем обработать — это генерация врагов, потому что у нас уже есть объект генератора вертолетов, хотя нам нужно было реорганизовать его, чтобы один и тот же экземпляр генератора можно было повторно использовать для разного количества врагов.

```
private void _level_OnLevelStart(object sender,
LevelEvents.StartLevel e)
{
    // Пока намеренно оставлено пустым
}

private void _level_OnLevelEnd(object sender,
LevelEvents.EndLevel e)
{
    // Пока намеренно оставлено пустым
}

private void _level_OnLevelNoRowEvent(object sender,
LevelEvents.NoRowEvent e)
{
    // Пока намеренно оставлено пустым
}

private void _level_OnGenerateTurret(object sender,
LevelEvents.GenerateTurret e)
{
    // Пока намеренно оставлено пустым
}

private void _level_OnGenerateEnemies(object sender,
LevelEvents.GenerateEnemies e)
{
    _chopperGenerator.GenerateChoppers(e.NbEnemies);
}
```

Мы не будем здесь освещать изменения, необходимые для обновления генератора прерывателя. Не стесняйтесь искать обновленный класс в нашем репозитории GitHub или попытаться реорганизовать его и посмотреть, сможете ли вы соответствовать тому, как мы называли его ранее.

## ГЛАВА 11 LEVEL DESIGN

Чтобы уровни запускались правильно, нам нужно генерировать события при каждом обновлении. Добавьте эту строку в метод [UpdateGameState\(\)](#):

```
_level.GenerateLevelEvents(gameTime);
```

Когда игрок умирает, нам нужно сбросить уровень, чтобы у него был второй или третий шанс пройти его. Обновите метод [ResetGame\(\)](#) и добавьте эту строку внизу:

```
_level.Reset();
```

Запустите игру и посмотрите, сможет ли она правильно прочитать уровень и сгенерировать то количество врагов-вертолетов, которое вы установили в каждом ряду уровня!

## Добавление турелей

Наша игра пока не слишком сложная. Враги-вертолеты генерируются и пролетают сквозь экран, но в нижних углах экрана есть несколько безопасных мест, где спрайт игрока никогда ничем не заденет. Чтобы изменить это, мы могли бы добавить в игру новые пути для вертолетов или турели, стреляющие в игрока, где бы он ни находился. Это устранит любое безопасное место из игры, но создаст новую проблему. Поскольку турели будут прокручиваться вниз с той же скоростью, что и фон, они в конечном итоге достигнут нижней части экрана, выровняются по горизонтали с игроком и будут стрелять пулями, от которых игрок не сможет уклониться. Чтобы дать игроку шанс выжить в этой ситуации, мы должны позволить ему двигаться вверх и вниз! Это простая функциональность для добавления и требует лишь нескольких изменений кода. Давайте посмотрим, сможете ли вы сделать это самостоятельно!

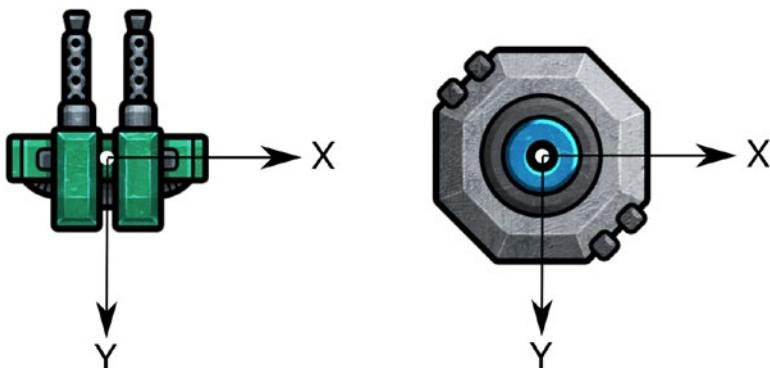
Мы внесли еще два изменения в код. Во-первых, скорость прокрутки фона была перемещена из класса [TerrainBackground](#) в класс [GameState](#), чтобы ее можно было использовать совместно с нашими турелями, расположенными ниже. Второе изменение заключалось в обновлении обработчика событий [\\_onObjectChanged\(\)](#), чтобы он мог обрабатывать любой тип [BaseGameObject](#), а не только объекты [ChopperSprite](#).

## Игровое искусство и происхождение

Игровой арт для новых турелей состоит из трех спрайтов. Основание башни ([Tower.png](#)), двойная пушка ([MG2.png](#)) и спрайт пули ([Bullet\\_MG.pgn](#)).

Эти активы находятся в директории `Assets\png`, и их просто нужно добавить в конвейер контента в новой папке `Sprites\Turrets`. Мы сохранили имена их конвейеров без изменений.

База и пушка должны быть собраны друг над другом, при этом пушка вращается над базой, следя за игроком. Посмотрите на Рис. 11-4, на котором показаны спрайты основания пушки и башни с исходной точкой и нарисованными над ними осями X и Y. Белые круги, расположенные между двумя стволами пушек над его основанием, указывают на центр вращения спрайта пушки. Это важно отметить, потому что, хотя координата X этого центра вращения находится точно на половине ширины текстуры, его координата Y не совпадает и установлена на 158 пикселей. Центр вращения также называется точкой отсчета изображения, потому что все игровые объекты будут вращаться вокруг своей точки отсчета, к которой мы вкратце коснулись, заставляя вращаться лопасти вертолета в предыдущей главе.



*Рис. 11-4. Орудийная часть башни с исходными координатами X и Y*

## ГЛАВА 11 LEVEL DESIGN

Другим важным аспектом этого изображения пушки является то, что не повернутое под углом ноль градусов, оно направлено вверх, а не вправо. Из-за этого важно различать угол наклона нашей турели и фактическое направление, на которое она указывает. Игровые объекты редко фиксируются и часто вращаются, поэтому мы должны включить это в наш [BaseGameObject](#). Откройте класс и добавьте следующий код:

```
public class BaseGameObject
{
    protected float _angle;
    protected Vector2 _direction;

    protected Vector2 CalculateDirection(
        float angleOffset = 0.0f)
    {
        _direction = new Vector2((float)Math.Cos(
            _angle - angleOffset),
            (float)Math.Sin(_angle - angleOffset));
        _direction.Normalize();
        return _direction;
    }
}
```

Это позволит всем нашим игровым объектам сохранять угол и направление, которые можно обновлять на основе защищенной переменной [\\_angle](#). При необходимости игровой объект может использовать угол и смещение для получения нормализованного вектора, представляющего, куда указывает объект. Смещение по умолчанию равно нулю, и в этом случае [CalculateDirection\(\)](#) вернет вектор, выровненный с осью X, когда переменная [\\_angle](#) также равна нулю. Однако, как мы увидим далее, наша турельная пушка направлена вверх, а не вправо, когда ее угол равен нулю, поэтому мы обеспечим смещение, чтобы получить точное направление.

Давайте создадим наш класс `TurretSprite` в директории `Objects\` и начнем с его инициализации:

```
public class TurretSprite : BaseGameObject
{
    private Texture2D _baseTexture;
    private Texture2D _cannonTexture;

    private float _moveSpeed;

    // With an angle of zero, the turret points up
    // so track offset for calculations when tracking player
    private const float AngleOffset = MathHelper.Pi / 2;
    private const float Scale = 0.3f;
    private const float AngleSpeed = 0.02f;
    private const int BulletsPerShot = 3;
    private const float CannonCenterPosY = 158;

    private int _hitAt = 100;
    private int _life = 50;

    private Vector2 _baseCenterPosition;
    private Vector2 _cannonCenterPosition;
    private float _baseTextureWidth;
    private float _baseTextureHeight;
    private bool _isShootingBullets;
    private TimeSpan _lastBulletShotAt;
    private int _bulletsRemaining;
    private bool _attackMode;

    public bool Active { get; set; }

    public event EventHandler<GameplayEvents.TurretShoots>
        OnTurretShoots;
```

## ГЛАВА 11 LEVEL DESIGN

```
public TurretSprite(Texture2D baseTexture, Texture2D
cannonTexture, float moveSpeed)
{
    _isShootingBullets = false;
    _moveSpeed = moveSpeed;
    _baseTexture = baseTexture;
    _cannonTexture = cannonTexture;
    _angle = MathHelper.Pi; // point down by default
    _bulletsRemaining = BulletsPerShot;
    _attackMode = false;
    Active = false;

    _direction = CalculateDirection(AngleOffset);

    _baseTextureWidth = _baseTexture.Width * Scale;
    _baseTextureHeight = _baseTexture.Height * Scale;

    _baseCenterPosition = new Vector2(_baseTextureWidth /
        2f, _baseTextureHeight / 2f);
    _cannonCenterPosition = new Vector2(_cannonTexture.
        Width / 2f, CannonCenterPosY);

    AddBoundingBox(new Engine.Objects.BoundingBox(new
        Vector2(0, 0),
        _baseTexture.Width * Scale,
        _baseTexture.Height * Scale));
}
```

Давайте рассмотрим все эти частные переменные одну за другой. Во-первых, турель должна отслеживать две текстуры для своей базы и своей пушки, поскольку она будет отвечать за обработку обоих. Поскольку эта турель будет двигаться вниз с той же скоростью, что и фон, нам нужно знать, с какой скоростью она должна двигаться, и для этого мы используем переменную [\\_moveSpeed](#).

Затем у нас есть несколько констант, помогающих формировать поведение турели. Угловое смещение — это угол между осью X и направлением башни, когда `_angle` равен нулю. Мы будем использовать это для вычисления вектора `_direction` всякий раз, когда пушка вращается. Изображение спрайта башни также немного велико, поэтому мы уменьшим его в 0,3 раза. `AngleSpeed` используется для установки скорости вращения пушки, а `BulletsPerShots` говорит нам, сколько пуль выпустит башня за один залп. Как только он обнаружит игрока, он остановится и выпустит три пули. Поскольку она перестает двигаться, это дает игроку возможность увернуться от пуль и изменить позицию, а в идеале выстрелить в саму турель, пока она занята стрельбой, туда, где раньше был игрок.

Теперь поговорим об углах. Угол башни установлен на `MathHelper.Pi`, чтобы заставить ее первоначально указывать вниз. Этот угол задается в радианах, поэтому `PI` представляет собой изменение на 180 градусов, и, поскольку начальное изображение по умолчанию направлено вверх, это изменение приводит к тому, что оно переворачивается и указывает вниз. Это чисто косметика. Поскольку башня видна сверху экрана, нам кажется, что она выглядит лучше, если направлена вниз. Учитывая это начальное значение `_angle`, равное 180 градусам, и учитывая, что наведение вниз в MonoGame означает поворот по часовой стрелке на 90 градусов, мы имеем расхождение в 90 градусов между углом объекта и направлением. Чтобы компенсировать это, мы устанавливаем константу `AngleOffset` на `PI/2`, что составляет 90 градусов.

Булева переменная `_active` используется для предотвращения вращения или атаки турелей, поскольку турелям не очень весело атаковать игрока, когда он находится за кадром. Кроме того, как и наш вражеский вертолет, наши турели можно атаковать, поэтому у них есть определенное количество очков жизни. `_hitA` точно так же инициализируется большим числом, как и вертолеты, и используется в методе `Render()`, чтобы турель вспыхивала при попадании.

Наконец, мы отслеживаем некоторые координаты и используем `_isShootingBullets`, `_lastBulletShotAt`, `_bulletsRemaining` и `_attackMode`, чтобы управлять скоростью стрельбы турели, количеством пуль, оставшихся в текущем залпе, или должна ли турель двигаться или стрелять. То, как стреляет турель, похоже на то, как состояние игрового процесса определяет, как часто игрок может стрелять.

## ГЛАВА 11 LEVEL DESIGN

Конструктор [TurretSprite](#) инициализирует все эти приватные переменные, принимая во внимание тот факт, что наше изображение уменьшено. Он также добавляет большое поле к игровому объекту.

Каждый раз, когда наша башня вращается с помощью этих двух вспомогательных функций, мы вычисляем направление пушки, всегда принимая во внимание [AngleOffset](#):

```
public void MoveLeft()
{
    _angle -= AngleSpeed;
    _direction = CalculateDirection(AngleOffset);
}

public void MoveRight()
{
    _angle += AngleSpeed;
    _direction = CalculateDirection(AngleOffset);
}
```

Единственное, что заставит пушку вращаться, — это местоположение игрока на экране, поэтому давайте добавим в класс [TurretSprite](#) метод [Update](#), который принимает текущее местоположение игрока и время игры в качестве параметров:

```
public void Update(GameTime gameTime, Vector2 currentPlayerCenter)
{
    // Опустить башню
    Position = Vector2.Add(_position, new Vector2(0,
        _moveSpeed));

    // Если турель не активна, она не может вращаться или стрелять
    if (!Active)
    {
        return;
    }
```

```
// Может либо атаковать и выстрелить 3 пули, либо двигаться. Не оба
// действия сразу

if (_attackMode && _bulletsRemaining > 0)
{
    Shoot(gameTime);
}
else
{
    var centerOfCannon = Vector2.Add(_position,
        _cannonCenterPosition * Scale);
    var playerVector = Vector2.Subtract(currentPlayerCenter,
        centerOfCannon);
    playerVector.Normalize();

    var angleTurret = Math.Atan2(_direction.Y,
        _direction.X);
    var anglePlayer = Math.Atan2(playerVector.Y,
        playerVector.X);
    var angleDiff = angleTurret - anglePlayer;

    var tolerance = 0.1f;
    if (angleDiff > tolerance)
    {
        MoveLeft();
    }
    else if (angleDiff < -tolerance)
    {
        MoveRight();
    }

    if (angleTurret >= anglePlayer - tolerance &&
        angleTurret <= anglePlayer + tolerance)
    {
        _attackMode = true;
    }
}
```

## ГЛАВА 11 LEVEL DESIGN

```
        Shoot(gameTime);
    }
}

if (_bulletsRemaining <= 0)
{
    _attackMode = false;
}

// Предотвращаем слишком быструю стрельбу пулями
if (_lastBulletShotAt != null &&
    gameTime.TotalGameTime - _lastBulletShotAt >
    TimeSpan.FromSeconds(0.3))
{
    _isShootingBullets = false;
}

// Перезаряжаем пули каждые 2 секунды
if (gameTime.TotalGameTime - _lastBulletShotAt >
    TimeSpan.FromSeconds(2))
{
    _bulletsRemaining = BulletsPerShot;
}

}
```

Здесь много чего происходит, поэтому нам потребуется некоторое время, чтобы пройти это шаг за шагом. Во-первых, мы перемещаем турель вниз, чтобы создать иллюзию того, что игрок летит вверх. Независимо от того, неактивна ли турель, стреляет ли она или следит за игроком, она также должна двигаться вниз к нижней части экрана. Затем, если турель неактивна, мы выходим из метода, потому что мы не хотим отслеживать или стрелять в игрока, пока турель не будет активирована. Далее у нас есть две ветки: турель стреляет тремя пулями в одном направлении или вращается, чтобы зафиксировать центральную позицию игрока. Если турель находится в режиме атаки и у нее осталось стрелять, она продолжает стрелять. В противном случае он крутится.

При вращении турель должна определить, где находится игрок по отношению к себе. Мы передаем положение центра спрайта игрока в качестве параметра методу `Update()`, но прежде чем мы сможем его использовать, мы должны его преобразовать. Посмотрите на Рис.11-5 для примера. Векторы **a** и **b** обозначают положение нашей башни и объектов истребителя на экране. Поскольку метод `Update()` принадлежит классу `TurretSprite`, у нас есть доступ к вектору **b** через переменную `_position`, а позиция центра игрока передается в качестве параметра методу, который дает нам вектор **a**. Мы также вычисляли вектор **d**, который является направлением башни, всякий раз, когда изменяется переменная `_angle`. Однако, чтобы определить, направлена ли башня на игрока, нам нужно вычислить угол между вектором **d** и вектором **c**, чего у нас сейчас нет. Но есть простое решение.

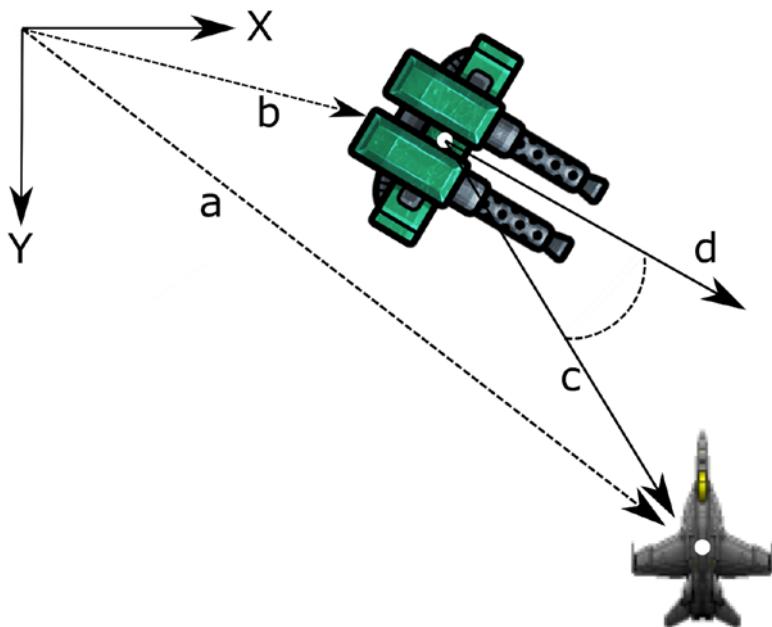


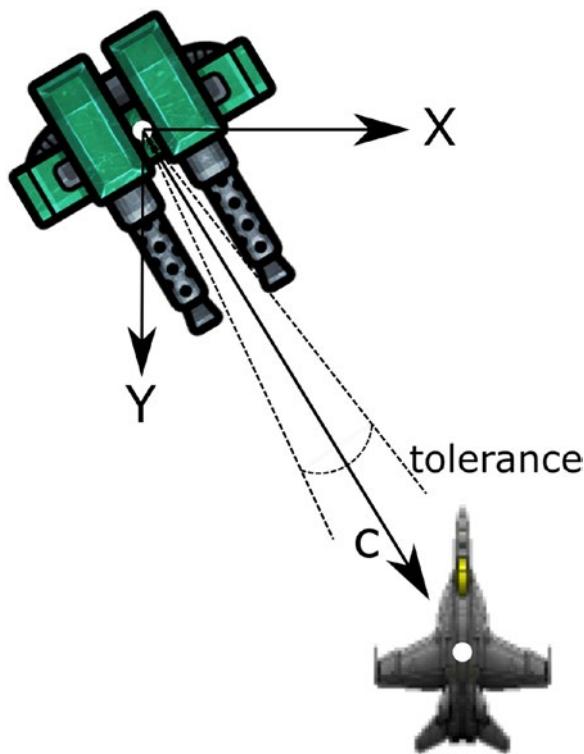
Рис. 11-5. Векторы и углы. Направлена ли башня на игрока?

## ГЛАВА 11 LEVEL DESIGN

Если мы вычтем вектор **b** из обеих позиций объекта, мы по сути, вернем центр башни обратно в исходную точку (0, 0) игрового экрана. Поскольку мы также вычитаем этот вектор **b** из положения истребителя, самолет останется в том же месте относительно башни. Это дает нам новый способ думать об объектах, как показано на [рис. 11-6](#). Обратите внимание, что на самом деле мы не перемещаем игровые объекты на экране, а просто манипулируем переменными для расчетов. Другой способ думать об этом состоит в том, что, поскольку  $a = b + c$ , то  $c = a - b$ . Итак, вот что делают эти две строки:

```
var centerOfCannon = Vector2.Add(_position,  
_cannonCenterPosition * Scale);  
var playerVector = Vector2.Subtract(currentPlayerCenter,  
centerOfCannon);
```

Первая строка находит центральную точку пушки, используя положение объекта и уменьшенное положение центра пушки на изображении текстуры. Затем мы вычитаем вектор `centerOfCannon` из текущего `PlayerCenter` и получаем наш вектор **c**.



**Рис. 11-6.** Башня, расположенная в начале координат, указывает на объект игрока.

Получив вектор  $c$ , мы можем вычислить его угол и угол вектора направления башни, а затем сравнить их друг с другом. Если они попадают в определенную область допуска, мы считаем, что турель направлена на игрока. Основная причина этой зоны допуска заключается в том, что при вращении турели она может пролететь мимо игрока. Без каких-либо допусков алгоритм заставит турель вернуться назад и снова пролететь над игроком в другом направлении, что приведет к изменению направления и так далее и тому подобное. Башня на экране будет казаться размытой, поскольку она постоянно безжалостно вращается вверх и вниз. Зона допуска позволяет ему оставаться неподвижным и обеспечивает гораздо лучший визуальный эффект. Если разница углов между  $c$  и направлением башни попадает в эту зону допуска, башня переходит в режим атаки и стреляет.

Затем проверяем, остались ли в башне патроны. Если их нет, отключаем режим атаки, чтобы турель снова могла вращаться. Наконец, у нас есть две функции синхронизации. Первый гарантирует, что турель не сможет стрелять быстрее, чем три пули в секунду. Эта проверка очень похожа на то, как мы предотвращаем слишком быструю стрельбу игрока в классе `GameplayState`. Наконец, мы хотим убедиться, что турель дает игроку двухсекундный перерыв, прежде чем она снова начнет стрелять.

Теперь нам нужно отрисовать башню на экране, а поскольку мы рисуем два спрайта друг над другом и хотим, чтобы при выстреле турель мигала, как вертолеты, мы переопределим метод `Render()` базового класса:

```
public override void Render(SpriteBatch spriteBatch)
{
    // If the turret was just hit and is flashing, Color should
    // alternate between OrangeRed and White
    var color = GetColor();

    var cannonPosX = _position.X + _baseCenterPosition.X;
    var cannonPosY = _position.Y + _baseCenterPosition.Y;
    var cannonPosition = new Vector2(cannonPosX, cannonPosY);

    spriteBatch.Draw(_baseTexture, _position,
        _baseTexture.Bounds, color, 0, new Vector2(0, 0),
        Scale, SpriteEffects.None, 0f);
    spriteBatch.Draw(_cannonTexture, cannonPosition,
        _cannonTexture.Bounds, Color.White,
        _angle, _cannonCenterPosition, Scale,
        SpriteEffects.None, 0f);
}
```

Метод `Render()` начинается с определения положения центра пушки. Затем рисуется основание башни без особой помпы. Наконец, он рисует пушку и становится немного сложнее. Этот метод `secondDraw()` используется с углом поворота, исходным вектором и масштабом. Вот что происходит под одеялом. MonoGame берет нашу текстуру и перемещает ее так, чтобы начало координат  $(0, 0)$  находилось именно там, где находится `_cannonCenterPosition`. Затем он поворачивает текстуру вокруг начала координат, прежде чем вернуть ее в позицию `cannonPosition`.

Остальной код `TurretSprite`, такой как `GetColor()`, `OnNotify()` и `JustHit()`, точно такой же, как код `ChopperSprite`, который мы рассмотрели в главе об обнаружении столкновений.

## Турельные пули

Последним оставшимся элементом полнофункциональной турели является создание пуль, которые будут более сложными, чем пули, которыми может стрелять игрок. На этот раз турельные пули немного сложнее. Пули нашего истребителя было легко реализовать, потому что они движутся прямо вверх по экрану и имеют выровненные по оси ограничивающие рамки для обнаружения столкновений. Турельные пули, с другой стороны, большую часть времени будут двигаться по диагонали и не могут иметь выровненных по оси ограничивающих рамок, потому что это может вызвать столкновения между пулей и самолетом, даже если они не касаются друг друга.

Нам нужно определить, где мы хотим, чтобы пули изначально располагались на экране, то есть точно за двумя стволами. Мы также хотим, чтобы пули были расположены под углом и шли в том же направлении, что и турель. Башня отвечает за указание классу состояния геймплея через событие `TurretShoots` создать реальные игровые объекты и передать классу состояния угол и направление пуль. Метод `Shoot()` реализован следующим образом:

```
public void Shoot(GameTime gameTime)
{
```

## ГЛАВА 11 LEVEL DESIGN

```
if (!_isShootingBullets && _bulletsRemaining > 0)
{
    var centerOfCannon = Vector2.Add(_position,
        _baseCenterPosition);

    // Find perpendicular vectors to position bullets left
    // and right of the center of the cannon
    var perpendicularClockwiseDirection = new Vector2(
        _direction.Y, -_direction.X);
    var perpendicularCounterClockwiseDirection = new
    Vector2(-_direction.Y, _direction.X);

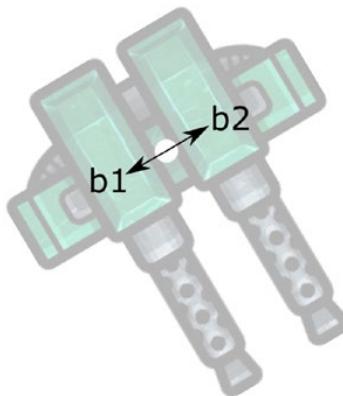
    var bullet1Pos =
        Vector2.Add(centerOfCannon,
            perpendicularClockwiseDirection * 10);
    var bullet2Pos =
        Vector2.Add(centerOfCannon,
            perpendicularCounterClockwiseDirection * 10);

    var bulletInfo =
        new GameplayEvents.TurretShoots(bullet1Pos,
            bullet2Pos, _angle, _direction);

    _bulletsRemaining--;
    _isShootingBullets = true;
    _lastBulletShotAt = gameTime.TotalGameTime;

    OnTurretShoots?.Invoke(this, bulletInfo);
}
}
```

Во-первых, мы проверяем, разрешено ли нам стрелять еще раз. Для `_isShootingBullets` устанавливается значение `true` с помощью метода `Update()`, когда мы впервые начинаем стрелять, но оно не будет ложным, пока не пройдет 0,3 секунды. осталось несколько патронов. Если эти два условия соблюдены, мы можем начать процесс создания объекта маркера. Нам нужно найти два вектора, которые перпендикулярны текущему направлению турели и противоположны друг другу, как показано на Рис.11-7. Это будет начальное положение пуль `b1` и `b2`. Как только у нас есть эти два вектора, которые нормализованы, потому что они вычисляются из вектора направления, который также нормализован, мы умножаем их на 10. Нормализованный вектор имеет длину 1, поэтому, чтобы быть полезным, этот вектор должен быть немного длиннее, а 10, кажется, быть как раз правильным числом, чтобы переместить пули от центра пушки под стволы.



*Рис. 11-7. Исходное расположение двух револьверных пуль `b1` и `b2`*

Добавим игровое событие `TurretShoots` в класс `GameplayEvents`:

```
public class TurretShoots : GameplayEvents
{
    public Vector2 Direction { get; private set; }
    public Vector2 Bullet1Position { get; private set; }
    public Vector2 Bullet2Position { get; private set; }
    public float Angle { get; private set; }
```

## ГЛАВА 11 LEVEL DESIGN

```
public TurretShoots(Vector2 bullet1Pos, Vector2 bullet2Pos,
                    float angle, Vector2 direction)
{
    Direction = direction;
    Bullet1Position = bullet1Pos;
    Bullet2Position = bullet2Pos;
    Angle = angle;
}
}
```

Теперь нам нужно изменить класс игрового процесса, чтобы он что-то делал с этим событием. Пока мы в классе, давайте также добавим код, необходимый для создания игровых турелей:

```
public class GameState : BaseGameState
{
    private const string TurretTexture = "Sprites/Turrets/Tower";
    private const string TurretMG2Texture = "Sprites/Turrets/MG2";
    private const string TurretBulletTexture = "Sprites/Turrets/
    Bullet_MG";

    private List<TurretSprite> _turretList = new
    List<TurretSprite>();
    private List<TurretBulletSprite> _turretBulletList = new
    List<TurretBulletSprite>();

    private void _level_OnGenerateTurret(object sender,
    LevelEvents.GenerateTurret e)
    {
        var turret = new TurretSprite(LoadTexture(TurretTexture),
                                    LoadTexture(TurretMG2
                                    Texture),
                                    SCOLLING_SPEED);
```

```

// Расположите турель за кадром на верхней турели.
turret.Position = new Vector2(e.XPosition, -100);

turret.OnTurretShoots += _turret_OnTurretShoots;
turret.OnObjectChanged += _onObjectChanged;
AddGameObject(turret);

_turretList.Add(turret);
}
}

```

Этот код создает пустые списки турелей и турели, которые мы будем отслеживать с течением времени, и реализует метод `_level_OnGenerateTurret()`, который мы оставили пустым в начале этой главы, работая над редактором уровней. Создание башни похоже на создание других игровых объектов в том смысле, что мы создаем новый экземпляр игрового объекта и передаем ему текстуры вместе со скоростью прокрутки, которая используется совместно с классом `TerrainBackground`. Затем мы помещаем турель прямо за пределы экрана вверху и регистрируем два события, указывающие на то, что турель стреляет пулями или в нее попали, чтобы мы могли справиться с ее смертью. Затем мы добавляем объект в список активных игровых объектов и в наш список турелей.

Теперь давайте реализуем обработчик события:

```

private void _turret_OnTurretShoots(object sender,
GameplayEvents.TurretShoots e)
{
    var bullet1 =
        new TurretBulletSprite(LoadTexture(TurretBulletTexture),
        e.Direction, e.Angle);
    bullet1.Position = e.Bullet1Position;
    bullet1.zIndex = -10;
}

```

## ГЛАВА 11 LEVEL DESIGN

```
var bullet2 =  
    new TurretBulletSprite(LoadTexture(TurretBulletTexture),  
    e.Direction, e.Angle);  
bullet2.Position = e.Bullet2Position;  
bullet2.zIndex = -10;  
  
AddGameObject(bullet1);  
AddGameObject(bullet2);  
  
_turretBulletList.Add(bullet1);  
_turretBulletList.Add(bullet2);  
}
```

Здесь мы создаем экземпляры двух игровых объектов `TurretBulletSprite`, используя информацию, предоставленную нам объектом `TurretSprite` через событие `TurretShoots`. Одна примечательная новая вещь — использование свойства `zIndex` всех наших игровых объектов. Если вы помните из предыдущих глав этой книги, `zIndex` представляет порядок, в котором мы должны рисовать наши объекты. Так как мы хотим, чтобы пули турели были нарисованы под башней, чтобы казалось, что они вылетают из стволов орудий, мы устанавливаем `zIndex` равным `-10`, в то время как `zIndex` башни по умолчанию равен нулю. Это заставит нашу основную функцию рисования сначала нарисовать пули, а затем поместить турель сверху. Мы до сих пор не использовали `zIndex`, потому что в нем не было необходимости.

Мы готовы добавить наши револьверные пули в кодовую базу. Создайте новый класс под названием `TurretBulletSprite` в каталоге `Objects`.

```
public class TurretBulletSprite : BaseGameObject  
{  
  
    private const float BULLET_SPEED = 18.0f;  
    private Vector2 _bulletCenterPosition;  
  
    public Segment CollisionSegment  
    {
```

```
get
{
    var segment = _direction * _texture.Height;
    return new Segment(_position, Vector2.Add(
        _position, segment));
}
}

public TurretBulletSprite(Texture2D texture, Vector2
direction, float angle)
{
    _texture = texture;
    _direction = direction;
    _direction.Normalize();

    _bulletCenterPosition = new Vector2(_texture.Width / 2,
    _texture.Height / 2);
    _angle = angle;
}

public void Update()
{
    Position = Position + _direction * BULLET_SPEED;
}

public override void Render(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(_texture, _position, _texture.Bounds,
    Color.White, _angle,
        _bulletCenterPosition, 1f,
        SpriteEffects.None, 0f);
}
```

Здесь не так много кода. Пуля создается с текстурой, направлением и углом, которые не меняются со временем. Конструктор нормализует направление на тот случай, если оно еще не нормализовано, а затем вычисляет центральную точку текстуры пули, чтобы мы могли вращать ее при рисовании. Метод `Update()` обеспечивает быстрое перемещение маркера по экрану, а метод `Render()` рисует маркер.

Остается свойство `CollisionSegment`, которое мы будем использовать для обнаружения столкновений с объектом игрока.

## Обнаружение столкновений

Поскольку мы решили не использовать наш алгоритм обнаружения столкновений с выравниванием по осям (AABB), нам пришлось использовать другой способ обработки столкновений. У нас есть несколько вариантов алгоритма на выбор: мы можем использовать алгоритм ориентированной ограничивающей рамки (OBB), стратегию столкновения сферы/поля или обнаружить пересечение между линией и ограничивающей рамкой, выровненной по оси. В алгоритме OBB косые ограничивающие рамки будут использоваться пулями, поскольку они почти всегда расположены под углом, и нам нужно будет вычислить пересечение наклонных ограничивающих рамок пуль с рамками, выровненными по оси спрайта игрока. С другой стороны, если бы мы представили пули в виде небольшой сферы рядом с центром спрайта, алгоритм обнаружения столкновений был бы очень простым и эффективным для вычислений. В конце концов, однако, мы решили вычислить пересечение между сегментом и игроком, что немного сложнее и немного менее эффективно, чем обнаружение столкновений сферы/AABB, но также немного более точно.

Итак, что такое сегмент? Это просто линия с начальным местоположением `P1` и конечным местоположением `P2`, реализованная следующим образом:

```
public class Segment
{
    public Vector2 P1 { get; private set; }
    public Vector2 P2 { get; private set; }
```

```

public Segment(Vector2 p1, Vector2 p2)
{
    P1 = p1;
    P2 = p2;
}
}

```

Итак, когда `TurretBulletSprite` вычисляет свой `CollisionSegment`, он умножает нормализованный вектор `_direction` на высоту текстуры. Это дает нам вектор такой же длины, как и сама пуля. Затем мы создаем сегмент, который начинается в позиции пули и заканчивается на высоте текстуры, идущей в правильном направлении.

Для обнаружения столкновений мы реализуем класс `newSegmentAABCollisionDetector`, который работает аналогично классу `AABCollisionDetector`, но определяет, пересекается ли сегмент с выровненной ограничивающей осью рамкой. Алгоритм прост и подобен алгоритму AABB. Если значение X начальной точки сегмента находится в пределах значений X ограничивающей рамки AND значение Y точки находится в пределах значений Y ограничивающей рамки, то у нас есть столкновение. Затем мы делаем ту же проверку для точки в конце сегмента. Вот код:

```

public class SegmentAABBCollisionDetector<A>
    where A : BaseGameObject
{
    private A _passiveObject;

    public SegmentAABBCollisionDetector(A passiveObject)
    {
        _passiveObject = passiveObject;
    }
}

```

## ГЛАВА 11 LEVEL DESIGN

```
public void DetectCollisions(Segment segment, Action<A>
collisionHandler)
{
    if (DetectCollision(_passiveObject, segment))
    {
        collisionHandler(_passiveObject);
    }
}

public void DetectCollisions(List<Segment> segments,
Action<A> collisionHandler)
{
    foreach(var segment in segments)
    {
        if (DetectCollision(_passiveObject, segment))
        {
            collisionHandler(_passiveObject);
        }
    }
}

private bool DetectCollision(A passiveObject, Segment
segment)
{
    foreach(var activeBB in passiveObject.BoundingBoxes)
    {
        if (DetectCollision(segment.P1, activeBB) ||
            DetectCollision(segment.P2, activeBB))
        {
            return true;
        }
    }
}
```

```
        else
        {
            return false;
        }
    }

    return false;
}

private bool DetectCollision(Vector2 p, BoundingBox bb)
{
    if (p.X < bb.Position.X + bb.Width &&
        p.X > bb.Position.X &&
        p.Y < bb.Position.Y + bb.Height &&
        p.Y > bb.Position.Y)
    {
        return true;
    }
    else
    {
        return false;
    }
}
}
```

Теперь мы можем объединить все это в классе `GameplayState`. Добавьте следующий код в класс `DetectCollisions()`, чтобы добавить столкновения пуль и ракет с турелью, чтобы игрок мог их уничтожить, что является ключевой стратегией выживания:

```
var turretBulletCollisionDetector =
    new SegmentAABBollisionDetector<PlayerSprite>(
    _playerSprite);
```

## ГЛАВА 11 LEVEL DESIGN

```
bulletCollisionDetector.DetectCollisions(_turretList, (bullet,
turret) =>
{
    var hitEvent = new GameplayEvents.ObjectHitBy(bullet);
    turret.OnNotify(hitEvent);
    _soundManager.OnNotify(hitEvent);
    bullet.Destroy();
});

missileCollisionDetector.DetectCollisions(_turretList,
(missile, turret) =>
{
    var hitEvent = new GameplayEvents.ObjectHitBy(missile);
    turret.OnNotify(hitEvent);
    _soundManager.OnNotify(hitEvent);
    missile.Destroy();
});
```

Затем в том же методе измените раздел `if (!_playerDead)`, чтобы добавить столкновения пуль турелей с игроком:

```
if (_playerDead)
{
    var segments = new List<Segment>();
    foreach (var bullet in _turretBulletList)
    {
        segments.Add(bullet.CollisionSegment);
    }

    turretBulletCollisionDetector.DetectCollisions(segments, _ =>
    {
        KillPlayer();
    });
}
```

```
playerCollisionDetector.DetectCollisions(_playerSprite,  
    (chopper, player) =>  
    {  
        KillPlayer();  
    });  
}
```

## Очистка

Теперь, когда все подключено, нам нужно выполнить нашу обычную чистку. При уничтожении турелей их необходимо убрать из игры. Когда турели и турельные снаряды находятся за пределами экрана (но не тогда, когда они только что были добавлены в игру и находятся прямо над экраном), их необходимо удалить. Мы не будем рассматривать эту задачу здесь, но вы можете увидеть, как мы справились с ней, в конечном решении этой главы.

## Добавление текста

Последнее, что мы хотим добавить, это игровой текст, чтобы пожелать игроку удачи, когда он начнет уровень, и поздравить его, когда он закончит его. Для этого мы будем повторно использовать наш объект GameOverText, так как шрифт имеет соответствующий размер. Чтобы это произошло, нам просто нужно добавить новый текстовый игровой объект и реализовать три последних события уровня, которые мы оставили пустыми в начале этой главы:

```
private GameOverText _levelStartEndText;  
public override void LoadContent()  
{  
    _levelStartEndText = new GameOverText(LoadFont(GameOverFont));  
}
```

## ГЛАВА 11 LEVEL DESIGN

```
private void _level_OnLevelStart(object sender,
LevelEvents.StartLevel e)
{
    _levelStartEndText.Text = "Good luck, Player 1!";
    _levelStartEndText.Position = new Vector2(350, 300);
    AddGameObject(_levelStartEndText);
}

private void _level_OnLevelEnd(object sender, LevelEvents.
EndLevel e)
{
    _levelStartEndText.Text = "You escaped. Congrats!";
    _levelStartEndText.Position = new Vector2(300, 300);
    AddGameObject(_levelStartEndText);
}

private void _level_OnLevelNoRowEvent(object sender,
LevelEvents.NoRowEvent e)
{
    RemoveGameObject(_levelStartEndText);
}
```

Эта небольшая дополнительная деталь действительно здорово меняет нашу игру.

## Обзор нашего дизайна уровней

В нашей игре теперь есть возможность загружать текстовый файл уровня и проходить его, но является ли наш уровень сложным или увлекательным? Мы бы сказали, что это сложно. Как и было задумано, этот уровень почти невозможно пройти, потому что турели жестоки и с ними невероятно трудно справиться. Кроме того, в самом начале уровня у нас есть четыре турели на экране одновременно, и очень сложно двигаться дальше этой точки.

К счастью, эту проблему легко решить, отредактировав файл `Level1.txt` и изменив расположение турелей, чтобы упростить уровень.

Когда мы играли на своем уровне после того, как были добавлены турели, мы поняли, что они добавили элемент неожиданности для игрока, который потом думает, что все действительно становится серьезно. Игра не шутит и убьет их, если они сделают хотя бы одну ошибку. Этот элемент неожиданности в сочетании с одновременной видимостью четырех турелей заставил нас нахмуриться и по-настоящему сосредоточиться. Игра стала веселей!

Попробуйте обновить файл `Level1.txt`, чтобы добавить или удалить турели, или изменить количество сгенерированных вражеских вертолетов, или когда они сгенерируются, запустите игру и посмотрите, как она проходит. Однако если вы продолжаете умирать, но все же хотите досмотреть уровень до конца, мы добавили переменную `_indestructible` в `BaseGameState`. Если установлено значение `true`, игрок не умрет.

## Улучшение игрового процесса

Пока мы работали над кодом для этой главы, стало очевидно, что геймплей немного подтормаживает, а спрайты не плавно перемещаются по экрану. Есть много факторов, которые могут вызвать заикание игры. MonoGame пытается работать максимально эффективно и по умолчанию вызывает функцию `Update()` 60 раз в секунду. Однако из-за определенных вещей игра время от времени пропускает отрисовку кадра. Обычно виновником является сборщик мусора `.Net`, выполняющий работу по удалению экземпляров объектов из памяти, когда на них больше не ссылаются. У нас есть возможности для улучшения. При каждом вызове `Update()` мы деактивируем вражеские объекты, находящиеся за пределами экрана, и оставляем их в памяти. В конце концов, сборщик мусора заберет их. Улучшения, которые мы могли бы сделать здесь, включали бы наличие пула предварительно созданных объектов и повторное использование игровых объектов так же, как мы перерабатывали наши объекты-частицы в главе 8.

Однако в нашем случае заикание происходило даже тогда, когда сборщик мусора не работал, поэтому проблема должна быть где-то в другом месте. Совместное выполнение функций `Update()` и `Draw()` не должно занимать более 1/60 секунды; иначе мы не сможем добиться 60 кадров в секунду. В этом случае MonoGame отвечает, устанавливая для флага `GameTime.IsRunningSlowly` значение `true`, чтобы позволить разработчику отреагировать. Он также пропустит вызов функции `Draw()` несколько раз, чтобы иметь возможность `callUpdate()` в течение времени, которое, как мы надеемся, может наверстать упущенное. Из-за этого игра будет выглядеть рывками, но это не наша проблема. Хотя мы могли бы добавить счетчик в наш класс `GameplayState`, который мы увеличиваем при каждом вызове `Render()` и отображать количество кадров в секунду на экране, мы вполне уверены, что наша игра работает намного быстрее ограничений по времени.

Другим фактором, который может вызвать заикание, является то, как MonoGames синхронизируется с частотой обновления вашего монитора, когда он пытается отрисовать что-то на экране, потому что в конечном итоге мы не можем перерисовывать экран чаще, чем может выдержать монитор. Хотя это более глубокая тема для данной книги, мы обнаружили, что указание MonoGame не синхронизироваться с монитором приводит к более плавной работе.

Мы настроили функцию `Main()` в нашем классе `Program`, чтобы указать MonoGame использовать фиксированный временной шаг, установив флаг `game.IsFixedTimeStep` в значение `true`, что заставляет игру работать с определенным количеством кадров в секунду. Поскольку мы по-прежнему хотим запускать игру со скоростью 60 кадров в секунду, мы должны сообщить MonoGame, сколько времени занимает каждый кадр, что составляет 1000 миллисекунд, разделенных на 60. Теперь функция `Main()` нашей программы выглядит следующим образом:

```
static void Main()
{
    using (var game = new MainGame(WIDTH, HEIGHT, new
        SplashState()))
    {
        game.IsFixedTimeStep = true;
```

```
game.TargetElapsedTime = TimeSpan.FromMilliseconds(1000.0f /  
60); game.Run();  
}
```

Затем, чтобы отключить синхронизацию с монитором, мы устанавливаем флаг `Graphics.SynchronizeWithVerticalRetrace` в `false` в методе `Initialize()` `MainGame`:

```
graphics.SynchronizeWithVerticalRetrace = false;
```

С этими вещами игра стала работать намного плавнее.

## Резюме

В этой главе мы узнали, как создать внутреннее представление уровня в виде сетки и как загрузить данные из текстового файла в эту сетку. Мы также реализовали класс `Level`, который может использовать эту внутреннюю сетку событий уровней и вызывать добавление реальных игровых объектов на экран. Наконец, мы добавили турели, которые могут стрелять в игрока, что до сих пор было самым сложным игровым объектом для реализации, включая повороты и векторы, а также новый способ обнаружения столкновений между сегментами и выровненными по осям прямоугольниками. Мы проделали здесь большую работу, и наша игра выглядит намного лучше.

Хотя это последняя глава книги, наше путешествие в мир разработки игр только начинается. Разработка игр на самом деле представляет собой набор методов и алгоритмов, сшитых воедино для создания игры, и есть еще много всего, что можно открыть для себя. Например, мы использовали `zIndices` для определения порядка отрисовки наших объектов, но существуют методы наслойения, когда все, с чем может столкнуться игрок, рисуется на нижнем уровне, а игрок и враги рисуются на более высоком уровне. Мы также слегка поцарапали поверхность движков частиц. Мы не добавили возможность трансформировать наши текстуры по мере их старения! А как насчет обнаружения пути и агентов ИИ, позволяющих нашим игровым объектам действовать независимо от состояния игрового процесса?

Нам предстоит еще многому научиться, но эта книга дает вам хорошую основу и хорошую отправную точку для создания игр с помощью MonoGame. Итак, что вы добавите в нашу игру? Вот лишь несколько возможностей, которыми вы могли бы заняться дальше:

- Больше фоновых элементов, таких как реки и мосты, городские пейзажи.
- Более широкий выбор врагов.
- Бои с боссами!
- Добавление разного оружия игрокам.
- Когда враги умирают, они могут генерировать расходуемые игровые объекты, которые дают игроку некоторое усиление, если он их поднимает, например бонусы.

Мы с нетерпением ждем возможности сыграть в игру, которую вы создадите в будущем!