Крис Дикинсон

Оптимизация игр в Unity 5

Chris Dickinson

Unity 5 Game Optimization

MASTER PERFORMANCE OPTIMIZATION FOR UNITY3D APPLICATIONS WITH TIPS AND TECHNIQUES THAT COVER EVERY ASPECT OF UNITY3D ENGINE

[PACKT] PUBLISHING

BIRMINGHAM – MUMBAI

Крис Дикинсон

Оптимизация игр в Unity 5

Советы и методы оптимизации приложений, охватывающие все аспекты работы с движком Unity3D

Москва, 2017

УДК 004.4'2Unity3D
ББК 32.972
Д44

Дикинсон К.

Д44 Оптимизация игр в Unity 5: пер. с англ. Рагимова Р. Н. – М.: ДМК Пресс, 2017. – 306 с.: ил.

ISBN 978-5-97060-432-8

Ничто так не отпугивает игроков, как сбои при выполнении игры. Задержки при вводе, медленное отображение, физические нестыковки, подергивание, замирание и неожиданное аварийное завершение превращают игру в кошмар, и разработчики игр должны сделать все, чтобы этого никогда не происходило!

Из этой книги вы узнаете, как выявлять и исследовать узкие места во всех основных компонентах движка Unty3D. В каждом конкретном случае описываются способы идентификации проблем, порядок выявления их причин и ряд возможных решений.

Издание адресовано разработчикам игр среднего и продвинутого уровня, имеющим опыт работы с Unity и языком C# и желающим повысить производительность своих приложений.

УДК 004.4'2Unity3D
ББК 32.972

Copyright © Packt Publishing 2015. First published in the English language under the title “Unity 5 Game Optimization” – (9781785884580).

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78588-458-0 (анг.) © 2016 Packt Publishing
ISBN 978-5-97060-432-8 (рус.) © Оформление, перевод, ДМК Пресс, 2017

Содержание

Об авторе **9**

Благодарности **10**

О технических рецензентах **11**

Предисловие **13**

Глава 1. Выявление проблем с производительностью **19**

Профилировщик Unity Profiler 20
  Запуск профилировщика 21
  Окно профилировщика 25
Методы анализа производительности 32
  Проверка присутствия сценария 33
  Проверка количества сценариев 34
  Сведение к минимуму изменений в текущем коде 35
  Сведение к минимуму внутренних помех 35
  Сведение к минимуму внешних помех 37
Выборочное профилирование сегментов кода 37
  Управление профилировщиком из сценариев 38
  Нестандартное профилирование использования центрального процессора 40
Сохранение и загрузка данных профилировщика 44
  Сохранение данных профилировщика 45
  Загрузка данных профилировщика 48
Заключительные соображения о профилировании и анализе 52
  Освоение профилировщика 52
  Уменьшение шума 53
  Сосредоточение внимания на проблеме 54
Итоги 54

Глава 2. Приемы разработки сценариев **56**

Кэширование ссылок на компоненты 56
Самый быстрый метод получения ссылок на компоненты 58
Удаление пустых объявлений обратных вызовов 61
Не используйте методов Find() и SendMessage() 63
  Статические классы 65
  Компоненты-одиночки 67
  Сохранение ссылок на существующие объекты 71
  Глобальная система обмена сообщениями 74
Отключение неиспользуемых сценариев и объектов 85
  Отключение невидимых объектов 85
  Отключение отдаленных объектов 86

Замена расстояния квадратом расстояния 87
Избегайте извлечения строковых свойств объектов игры 89
Метод Update, сопрограммы и метод InvokeRepeating 91
Кэширование изменений компонента Transform 97
Ускорение проверки отсутствия ссылки на игровой объект 98
Итоги 99

Глава 3. Преимущества пакетной обработки 100
Вызовы системы визуализации 101
Материалы и шейдеры 103
Динамическая пакетная обработка 107
  Атрибуты вершин 108
  Однородное масштабирование 109
  Краткие выводы о динамической пакетной обработке 110
Статическая пакетная обработка 111
  Флаг Static 111
  Требования к памяти 112
  Ссылки на материалы 113
  Особенности использования статической пакетной обработки 113
  Краткие выводы о статической пакетной обработке 115
Итоги 116

Глава 4. Привнесение искусства 117
Аудио 118
  Загрузка аудиофайлов 118
  Форматы кодирования и уровни качества 121
  Улучшение производительности аудио 123
Файлы текстур 128
  Форматы сжатия 129
  Улучшение производительности обработки текстур 131
Файлы мешей и анимаций 141
  Уменьшение количества полигонов 141
  Импорт/расчет только необходимого 143
  Встраиваемые анимации 143
  Оптимизация мешей движком Unity 144
  Объединение мешей 145
Итоги 145

Глава 5. Разгон физического движка 147
Внутреннее устройство физического движка 148
  Физические движки и время 148
  Статические и динамические коллайдеры 152
  Обнаружение столкновений 153
  Виды коллайдеров 154
  Матрица столкновений 155
  Активное и неактивное состояния компонента Rigidbody 156
  Отбрасывание лучей и объектов 157
Оптимизация производительности физической системы 157
  Настройка сцены 157

Правильное использование статических коллайдеров 160
Оптимизация матрицы столкновений 161
Предпочтение дискретного обнаружения столкновений 162
Изменение частоты фиксированных обновлений 163
Настройка максимально допустимой длительности 165
Уменьшение отбрасывания лучей и ограничение проверяемого объема 165
Избегайте сложных меш-коллайдеров 167
Избегайте сложных физических компонентов 170
Пусть физические объекты поспят 170
Изменение количества итераций 172
Оптимизация тряпичных кукол 173
Когда следует использовать физическую систему 175
О возможности перехода на Unity 5 176
Итоги 177

Глава 6. Динамическая графика 178
Профилирование проблем отображения 179
Профилирование графического процессора 181
Отладка кадров 184
Поиск методом перебора 185
Основная нагрузка приходится на центральный процессор 185
Узкие места на этапе предварительной обработки 189
Уровень детализации 189
Отключение скининга графическим процессором 191
Уменьшение тесселяции 192
Узкие места на этапе окончательной обработки 192
Скорость заполнения 192
Пропускная способность памяти 206
Ограничения видеопамяти 210
Освещение и затенение 212
Непосредственное отображение 213
Отложенное затенение 214
Обработка освещения в вершинном шейдере (устаревший способ) 214
Обработка теней в реальном времени 215
Оптимизация освещения 215
Оптимизация графики для мобильных устройств 217
Минимизация обращений к системе визуализации 218
Минимизация количества материалов 218
Уменьшение размеров текстур и количества материалов 218
Квадратные текстуры с размером стороны, кратной степени числа 2 219
Использование в шейдерах форматов с минимально допустимой точностью 219
Избегайте альфа-тестирования 219
Итоги 219

Глава 7. Мастерство управления памятью 221
Платформа Mono 222
Процесс компиляции 224

Оптимизация использования памяти 226
Области памяти Unity 227
Значения и ссылки 236
Важность порядка размещения данных 249
Прикладной программный интерфейс Unity 250
Циклы foreach 251
Сопрограммы 252
Замыкания 252
Функции в библиотеке .NET 253
Временные рабочие буферы 254
Пулы объектов 254
Пулы шаблонных объектов 257
Компоненты пула 260
Система пулов шаблонных объектов 263
Пулы шаблонных объектов 267
Активация объектов 268
Предварительное создание экземпляров 269
Деактивация объектов 270
Тестирование пула шаблонных объектов 271
Организация пулов шаблонных объектов и загрузка сцены 272
Итоговые замечения об организации пулов шаблонных объектов 273
Дальнейшее развитие Mono и Unity 274
Итоги 276

Глава 8. Тактические советы и подсказки 278
Подсказки по клавишам быстрого доступа в редакторе 279
Игровые объекты 279
Представление сцены 279
Массивы 280
Интерфейс 280
Прочее 281
Советы, касающиеся интерфейса редактора 281
Общие 281
Представление инспектора 284
Представление проекта 286
Представление иерархии 287
Представления сцены и игры 288
Режим воспроизведения 289
Советы для сценариев 290
Общие 290
Атрибуты 291
Регистрация 293
Полезные ссылки 293
Советы по настройке редактора и меню 294
Советы, не касающиеся Unity напрямую 296
Другие советы 297
Итоги 298

Предметный указатель 300

Об авторе

Крис Дикинсон (Chris Dickinson) вырос в Англии, увлекается естественными науками, математикой и компьютерными играми. Получив в 2005 году степень магистра в области физики и электроники, сразу же переехал в Калифорнию, чтобы заниматься научными исследованиями в самом сердце Кремниевой долины. Затем, разочаровавшись в своем выборе, переключился на работу в области разработки программного обеспечения.

За последнее десятилетие он многого достиг, сделал карьеру в области разработки программного обеспечения, стал ведущим разработчиком. Крис в основном занимался программным обеспечением для автоматизации и создавал инструменты для внутреннего тестирования в процессе разработки, но страсть к видеоиграм не потерял. В 2010 году он занялся изучением секретов разработки игр и трехмерной графики и получил степень бакалавра в области разработки игр и программного моделирования. Написал учебник о применении физических законов в играх (*Learning Game Physics with Bullet Physics and OpenGL*, издана в Packt Publishing). В настоящее время продолжает разрабатывать программное обеспечение, свободное время посвящая работе над независимыми игровыми проектами, использующими Unity 3D.

Благодарности

Всего за 5 лет я сумел усвоить огромный объем знаний. Сделать это было бы невозможно без постоянной поддержки коллег, преподавателей, друзей и семьи.

Спасибо моим коллегам по работе, что с пониманием отнеслись к моему перидическому отсутствию, пока я обучался разработке игр в колледже.

А также спасибо моим преподавателям за то, что смогли оперативно донести до меня самое основное и помогли так много узнать о разработке игр за такое короткое время.

Спасибо друзьям, которые всегда были и остаются постоянным источником вдохновения, за их интерес к моей работе.

Спасибо моей семье за предоставленную возможность получить так много знаний, ощущений и любви за такое короткое время.

И конечно, спасибо моей прекрасной жене и лучшему другу Джейми, окружающей меня заботой и поддержкой и помогающей мне творить.

О технических рецензентах

Клиффорд Чампион (Clifford Champion) обладает обширным, многолетним опытом разработки программного обеспечения, полученным во время работы над 3D-играми, интернет-приложениями и программами искусственного интеллекта. Имеет ученые степени в области математики и информатики, присвоенные в Калифорнийском университете в Сан-Диего и в Калифорнийском университете в Лос-Анджелесе. В прошлом Клиффорд работал в компании Havok (ныне часть Microsoft), занимающейся разработкой игровых технологий, и в известной дизайн-студии PlainJoe Studios. В настоящее время возглавляет группу разработки программного обеспечения в компании zSpace (`zspace.com`), специализирующейся на 3D-приложениях виртуальной реальности для учебных заведений и промышленности.

Клиффорда можно найти в Twitter (`@duckmaestro`), и он готов вести дискуссии на любые темы.

Доктор Себастьян Т. Кёниг (Dr. Sebastian T. Koenig) получил степень доктора наук в области систем «человек–машина» в университете Кентербери, Новая Зеландия, за разработку основы системы виртуальной реальности для индивидуализированной когнитивной реабилитации. Имеет диплом психолога в области клинической нейропсихологии и реабилитации с применением систем виртуальной реальности, полученный в университете Регенсбурга, Германия.

Является основателем и генеральным директором Katana Simulations, где курирует проектирование, разработку и проведение когнитивной экспертизы систем моделирования для профессиональной подготовки. Его профессиональный опыт включает более 10 лет клинической работы в сфере когнитивной реабилитации и более 8 лет в области исследования, разработки и тестирования пользователями виртуальной реальности. Он часто выступает на международных конференциях и приглашается в качестве рецензента научных публикаций в области реабилитации, когнитивной психологии, нейропсихологии, программной инженерии, разработки игр, исследований восприятия пользователями игр и виртуальной реальности.

Разработал ряд прикладных программ для когнитивной оценки и профессиональной подготовки. За работу над задачей виртуальной памяти в 2011 году был удостоен престижной премии Laval Virtual в категории медицины и здравоохранения. Другими его достижениями являются: приложение виртуальной реальности для оценки восприятия, разработанное в сотрудничестве с Фондом Кесслера (Нью-Джерси, США), и запатентованное приложение для моторной и когнитивной подготовки JewelMine/Mystic Isle, основанное на Microsoft Kinect и созданное в Калифорнийском институте креативных технологий (Калифорния, США).

Поддерживает веб-сайт www.virtualgamelab.com, где представлены результаты его исследований и проекты разработки программного обеспечения. Сайт также содержит исчерпывающий список учебных материалов, посвященных игровому движку Unity.

Предисловие

Привлекательность – чрезвычайно важный компонент любой игры. Она определяется не только сценарием игры и игровым процессом, но также гладкостью графики, надежностью соединения с многопользовательскими серверами, скоростью реакции на действия пользователя и даже величиной конечного файла приложения, что важно при распространении из хранилищ или из облака. Барьер включения в разработку игр был значительно снижен благодаря появлению недорогих и качественных инструментов, таких как Unity. Однако требования к конечному продукту, предъявляемые игроками, растут с каждым днем. Следует иметь в виду, что все аспекты игры могут и будут тщательно изучены как игроками, так и критиками.

Цели оптимизации производительности тесно связаны с восприятием игры пользователями. Плохо оптимизированная игра, как правило, имеет низкую частоту кадров, зависает, сбоит, реагирует на действия пользователя с опозданием, долго загружается, ведет себя непоследовательно и дергано во время выполнения, работа физического движка часто нарушается и даже имеет чрезмерно высокое энергопотребление (что крайне важно в эпоху мобильных устройств). Наличие хотя бы одной из этих проблем должно приводить в ужас разработчиков, поскольку авторы обзоров склонны делать упор на отрицательных качествах, оставляя в стороне положительные.

Высокая производительность обеспечивается оптимальным использованием всех имеющихся ресурсов, включая ресурсы центрального процессора, такие как частота процессора и объем оперативной памяти, ресурсы графического процессора, такие как объем видеопамяти и ее пропускная способность, и т. д. Поэтому основной задачей оптимизации является такое распределение ресурсов, которое обеспечит эффективное их использование и выполнение высокоприоритетных задач в первую очередь. Даже небольшие, кратковременные задержки и потери производительности снижают привлекательность игры, ухудшают погружение в нее и ограничивают потенциал разработчиков.

Важно также выбрать момент, когда следует остановиться и прекратить заниматься улучшением производительности. В идеальном мире, где время и ресурсы ничем не ограничены, всегда найдется еще один способ достичь более удачного, быстрого и простого решения.

Но в реальном мире должен существовать момент в процессе разработки, когда следует заявить, что продукт достиг приемлемого качества. Если этого не сделать, дальнейшие усилия принесут мало пользы или вообще не дадут ощутимых результатов.

Лучший способ выбрать момент прекращения работ над улучшением производительности – ответить на вопрос: «А пользователь это заметит?» Если ответ на этот вопрос – нет, дальнейшая оптимизация не стоит затраченных на нее усилий. Существует старая поговорка, относящаяся к разработке программного обеспечения:

Преждевременная оптимизация – корень всех зол.

Под преждевременной оптимизацией понимается переработка и рефакторинг кода для повышения производительности, когда нет твердой уверенности в необходимости этих действий, например при полном отсутствии проблем с производительностью (отвечаем на вопрос о заметности для пользователя) или при наличии предположений, что проблема производительности вызвана определенной областью, но нет доказательств, что так и есть на самом деле. Такие ошибки обходятся разработчикам удручающе большим количеством рабочих часов, потраченных впустую.

Цель этой книги – дать вам инструменты, знания и навыки, необходимые для выявления и исправления проблем производительности, независимо от их причин. Подобные проблемы могут вызывать компоненты оборудования, такие как центральный процессор, графический процессор или оперативная память; подсистемы программного обеспечения, такие как подсистема физики, визуализации или же сам движок Unity. Кроме того, чем больше ресурсов удастся сэкономить, тем больше можно будет сделать с помощью движка Unity на одном и том же оборудовании, что позволит реализовать более интересный и динамичный процесс игры.

А это даст вашей игре больше шансов на успех и позволит ей выделиться из массы других, представленных на рынке, куда ежедневно выбрасываются новые, высококачественные игры.

О чем рассказывается в этой книге

Глава 1 «Выявление проблем с производительностью» посвящена знакомству с профилировщиком Unity Profiler и методами профилирования приложений, выявлению узких мест с точки зрения производительности и анализу основных причин их возникновения.

Глава 2 «Приемы разработки сценариев» описывает эффективные приемы разработки сценариев для Unity на языке C#, минимизации дублирования компонентов, оптимизации взаимодействий между объектами и многое другое.

Глава 3 «Преимущества пакетной обработки» посвящена изучению применения систем динамической и статической пакетной обработки в Unity для снижения нагрузки на систему визуализации.

Глава 4 «Привнесение искусства» поможет разобраться в закулисных процессах работы с ресурсами и узнать, как избежать распространенных ошибок при их импорте, сжатии и перекодировании.

Глава 5 «Разгон физического движка» посвящена особенностям физической системы Unity для 3D- и 2D-игр и правильной организации физических объектов для повышения производительности.

Глава 6 «Динамическая графика» подробно рассматривает систему визуализации и способы решения проблем производительности приложений, возникающих в системе визуализации, узкими местами которой являются графический процессор или центральный процессор, а также специальные методы для мобильных устройств.

Глава 7 «Мастерство управления памятью» анализирует внутренние рабочие процессы движка Unity, фреймворка Mono и управление памятью в этих компонентах, чтобы защитить приложение от распределения памяти в куче и сборки мусора во время выполнения.

Глава 8 «Тактические советы и приемы» знакомит со множеством полезных методов, используемых профессионалами для улучшения управления рабочими процессами и сценами.

Что потребуется для работы с книгой

Основное внимание в этой книге уделяется возможностям версии Unity 5.x. Большинство методов, описанных в книге применимы к проектам, созданным в Unity 4.x, но для некоторых из них требуется обновить Unity 4 до версии Pro (например, окклюзивная выбраковка, статическая пакетная обработка и даже сам профилировщик).

Кому адресована эта книга

Эта книга адресована разработчикам среднего и продвинутого уровня, уже имеющим опыт работы с большей частью функций Unity, заинтересованных в улучшении производительности игр или в решении конкретных проблем. Если вы испытываете проблемы, связанные

с перегрузкой процессора, скачками нагрузки во время выполнения, замедленным доступом к памяти, фрагментацией, уборкой мусора, снижением частоты графического процессора или пропускной способностью памяти, эта книга научит вас выявлять источники проблем и выбирать способы уменьшения их воздействия на приложение.

Вам потребуется знание языка C# при чтении разделов, содержащих сценарии и посвященных использованию памяти, а также базовое понимание библиотеки Cg – при рассмотрении оптимизации шейдеров.

Соглашения

В этой книге используется несколько разных стилей оформления текста для выделения разных видов информации. Ниже приводятся примеры этих стилей и объясняется их назначение.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, фиктивные адреса URL, пользовательский ввод и ссылки в Twitter будут выглядеть так: «Главной функцией обратного вызова Unity является функция обновления `Update()`.»

Блоки программного кода оформляются так:

```
public class TestComponent : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.Space)) {
            PerformProfilingTest();
        }
    }
}
```

Когда потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, он будет выделяться жирным шрифтом:

```
public class TestComponent : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.Space)) {
            PerformProfilingTest();
        }
    }
}
```

Новые термины и **важные слова** будут выделены жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет

оформляться так: «Пороговое значение для перехода в состояние сна может быть изменено при помощи выбора пункта **Edit** ⇨ **Project Settings** ⇨ **Physics** ⇨ **Sleep Threshold** (Редактирование ⇨ Параметры проекта ⇨ Физика ⇨ Интервал для перехода в состояние сна)».

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Загрузка исходного кода примеров

Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы нашли опечатку, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и «Packt» очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Выявление проблем с производительностью

Оценка производительности большинства программных продуктов является исследовательским процессом, включающим: определение максимальных поддерживаемых параметров (количества одновременно работающих пользователей, максимального объема доступной памяти, ресурсов центрального процессора и т. д.); выполнение нагрузочного тестирования путем воссоздания реальных ситуаций; инструментальный сбор результатов тестирования; анализ данных для выявления узких мест; комплексный анализ причин; внесение изменений в конфигурацию или код приложения для исправления проблем; повторение всего перечисленного еще раз.

Разработка игр является творческим процессом, но это вовсе не означает, что ее не следует рассматривать с объективной и исследовательской точек зрения. Игра направлена на конкретную целевую аудиторию, что и определяет ее аппаратные ограничения. Нужно протестировать приложение, собрать данные о работе нескольких компонентов (центральный процессор, графический процессор, память, физика, визуализация и т. д.) и сравнить их с требуемыми характеристиками. Эти данные используются для выявления узких мест в работе приложения, с помощью дополнительных инструментов определяются главные причины проблем, после чего проблемы рассматриваются с разных точек зрения.

Для успешного выполнения этого процесса необходимы инструменты и знания, методы, с которыми знакомит эта глава. Они будут использоваться на протяжении всей книги для выявления проблем с производительностью и определения их главных причин. Такие навыки составляют методику выявления и анализа проблем с произ-

водительностью **Unity**-приложения и выбора мест для внесения изменений. В этой главе будет проделана подготовительная работа для освоения материала остальных глав, описывающего пути решения выявленных проблем.

Начнем с изучения **профилировщика Unity Profiler** и его многочисленных функций. Затем опишем несколько способов выявления узких мест и в завершение приведем несколько советов, касающихся применения этих технологий.

Профилировщик Unity Profiler

Профилировщик Unity Profiler встроен в **редактор Unity** и обеспечивает целенаправленный поиск узких мест производительности путем сбора информации о работе компонентов **Unity3D** во время выполнения, то есть:

- использование центрального процессора отдельно каждым компонентом движка Unity3D;
- затраты на визуализацию;
- использование графического процессора GPU на разных этапах работы графического конвейера;
- использование памяти;
- затраты на воспроизведение аудиоэффектов;
- использование физического движка.

Начиная с **Unity 5.0**, компания Unity Technologies открыла доступ к профилировщику в редакции Personal Edition (прежде эта редакция называлась Free Edition).
Пользователям, использующим редакцию Unity 4 Free Edition, необходимо обновить ее до Unity 5 или приобрести лицензию для редакции Unity 4 Pro Edition.

Однако использование профилировщика влечет дополнительные накладные расходы. При включении флагов профилирования в настройках компилятора будут генерироваться журнал событий времени выполнения и прочий код поддержки профилирования, что вызовет дополнительные затраты ресурсов центрального процессора и памяти во время выполнения. Однако затраты на профилирование не очень значительны и не приводят к непредсказуемой разнице в поведении при включенном и отключенном профилировщике.

Кроме того, всегда избегайте использования режима редактирования во время профилирования и проведения сравнительного анализа

из-за накладных расходов на работу интерфейса редактора и затрат памяти на размещение различных объектов и компонентов. Для получения более точных и надежных данных проверку приложения лучше выполнять автономно и на целевом устройстве.

Читатели, знакомые с процедурой подключения профилировщика к приложениям, могут пропустить следующий раздел и перейти сразу разделу «*Окно профилировщика*».

Запуск профилировщика

Начнем с краткого обзора способов подключения профилировщика Unity в различных контекстах:

- к локальным экземплярам приложения через редактор или автономное профилирование самого редактора;
- к локальным экземплярам приложения в **веб-плеере Unity**;
- к удаленным экземплярам приложения на iOS-устройствах (iPad или iPhone);
- к удаленным экземплярам приложения на Android-устройствах (планшет или телефон с ОС Android).

Далее мы кратко рассмотрим особенности настройки профилировщика в каждом из этих контекстов.

Редактор или автономные экземпляры

Запустить профилировщик можно только из редактора Unity, после чего его следует подключить к запущенному экземпляру приложения. Здесь возможны случаи, когда игра запущена из редактора или как автономное приложение на локальном или удаленном устройстве, или когда требуется выполнить профилирование самого редактора.

Чтобы запустить профилировщик, выберите в меню редактора пункт **Window** ⇨ **Profiler** (Окно ⇨ Профилировщик), как показано на рис. 1.1. При работе редактора в режиме воспроизведения в окне профилировщика появится отчет.

Для профилирования автономных проектов при их сборке следует установить флаги Use Development Mode (Использовать режим разработки) и Autoconnect Profiler (Автоматически подключать профилировщик).

Выбор экземпляра приложения, запущенного в редакторе (в режиме воспроизведения) или автономного (собранного отдельно и рабо-

тающего в фоновом режиме), производится с помощью меню **Active Profiler** (Активный профилировщик) в окне профилировщика, как показано на рис. 1.2.

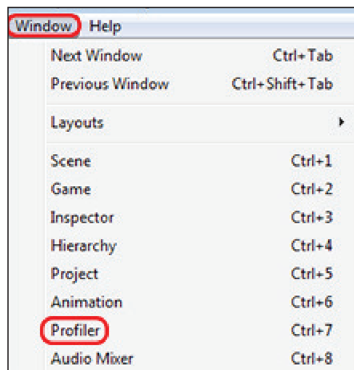


Рис. 1.1 ❖ Запуск профилировщика Unity

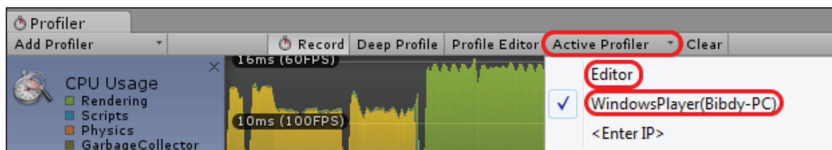


Рис. 1.2 ❖ Профилировщик Unity

Профилирование редактора

Профилирование редактора, например редактора сценариев, можно включить выбором пункта меню **Profile Editor** (Профилирование редактора) в окне профилировщика, как показано на рис. 1.3. Обратите внимание, что для этого требуется отметить параметр **Editor** (Редактор) в меню **Active Profiler** (Активный профилировщик).

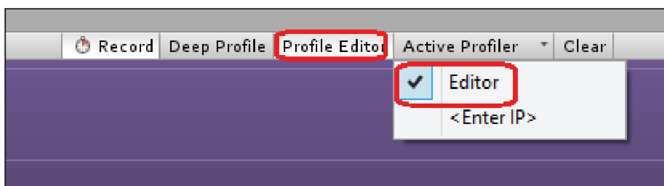


Рис. 1.3 ❖ Профилирование редактора сценариев

Подключение к веб-плееру Unity

Профилировщик можно также подключить к экземпляру приложения в веб-плеере Unity, запущенном в браузере. Это позволит выполнить профилирование веб-приложения в наиболее реальной среде с помощью целевого браузера и протестировать несколько типов браузеров для выявления несоответствий.

1. Перед сборкой приложения для веб-плеера включите флаг **Use Development Mode** (Использовать режим разработки).
2. Запустите скомпилированное для веб-плеера приложение в браузере, активируйте окно браузера и, удерживая нажатой клавишу **Alt** (*Option* на компьютере Mac), щелкните правой кнопкой мыши на объекте веб-плеера в браузере, чтобы открыть меню **Release Channel Selection** (Выбор канала вывода). Затем выберите канал **Development** (Разработка), как показано на рис. 1.4.

Обратите внимание, что изменение параметра Release Channel (Канал вывода) вызовет перезапуск приложения в веб-плеере.

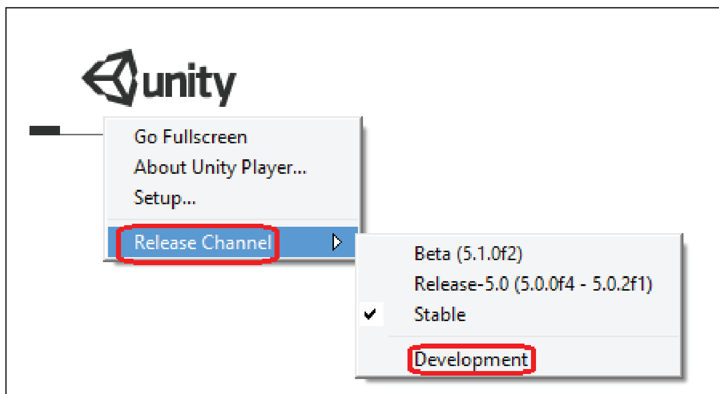


Рис. 1.4 ❖ Выброр канала вывода

3. Откройте профилировщик в окне редактора Unity и выберите пункт **Active Profiler ⇨ WindowsWebPlayer(COMPUTERNAME)** или **Active Profiler ⇨ OSXWebPlayer(COMPUTERNAME)** в зависимости от используемой операционной системы, как показано на рис. 1.5.

После этого в окне профилировщика должна появиться собранная им информация.

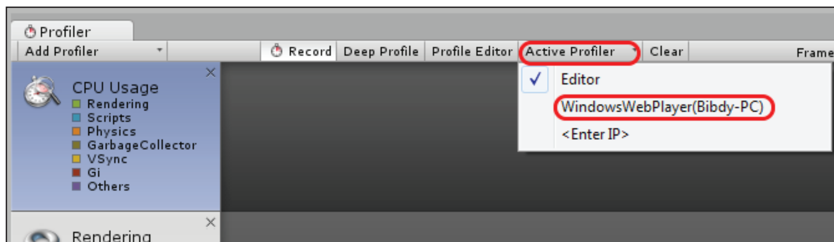
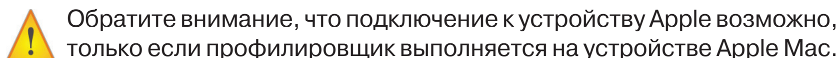


Рис. 1.5 ❖ Выбор веб-плеера

Подключение к удаленным iOS-устройствам

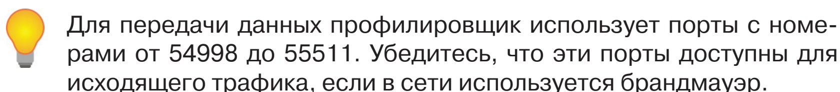
Профилировщик можно также подключить к активному экземпляру приложения, запущенному на удаленном iOS-устройстве, таком как iPad или iPhone, например через Wi-Fi-подключение. Чтобы подключить профилировщик к устройству Apple, выполните следующие действия:



Обратите внимание, что подключение к устройству Apple возможно, только если профилировщик выполняется на устройстве Apple Mac.

1. Перед сборкой приложения включите флаги **Use Development Mode** (Использовать режим разработки) и **Autoconnect Profiler** (Автоматически подключать профилировщик).
2. Подключите оба устройства, iOS и Mac, к локальной сети или сети ADHOC WiFi.
3. Соедините устройства iOS и Mac кабелем USB или Lightning.
4. Запустите сборку приложения, выбрав в меню пункт **Build & Run** (Собрать и запустить), как обычно.
5. Откройте окно профилировщика в редакторе Unity и выберите устройство в меню **Active Profiler** (Активный профилировщик).

После этого в окне профилировщика должна появиться собранная им информация.



Для передачи данных профилировщик использует порты с номерами от 54998 до 55511. Убедитесь, что эти порты доступны для исходящего трафика, если в сети используется брандмауэр.

Подключение к удаленным Android-устройствам

Существуют два способа подключения профилировщика к Android-устройству: через Wi-Fi-соединение или с помощью инструмента отладки **Android Debug Bridge** (**ADB**). ADB – это набор инструментов отладки, входящий в состав пакета Android SDK.

Для профилирования через Wi-Fi-соединение выполните следующие действия:

1. Перед сборкой приложения включите флаги **Use Development Mode** (Использовать режим разработки) и **Autoconnect Profiler** (Автоматически подключать профилировщик).
2. Подключите Android-устройство и компьютер к локальной Wi-Fi-сети.
3. Соедините Android-устройство с компьютером кабелем USB.
4. Запустите сборку приложения, выбрав в меню пункт **Build & Run** (Собрать и запустить), как обычно.
5. Откройте окно профилировщика в редакторе Unity и выберите устройство в меню **Active Profiler** (Активный профилировщик).

После этого в окне профилировщика должна появиться собранная им информация.

Для профилирования через ADB выполните следующие действия:

1. В окне командной строки Windows выполните команду `adb devices`, которая проверит, распознается ли устройство отладчиком ADB (если устройство не распознается, для него придется установить специальные драйверы и/или включить на целевом устройстве отладку через USB).

Обратите внимание, что если команда `adb devices` не будет найдена при попытке запустить ее из командной строки, добавьте путь к папке с Android SDK в переменную окружения `PATH`.

2. Перед сборкой приложения включите флаги **Use Development Mode** (Использовать режим разработки) и **Autoconnect Profiler** (Автоматически подключать профилировщик).
3. Соедините Android-устройство с компьютером кабелем USB.
4. Запустите сборку приложения, выбрав в меню пункт **Build & Run** (Собрать и запустить), как обычно.
5. Откройте окно профилировщика в редакторе Unity и выберите устройство в меню **Active Profiler** (Активный профилировщик).

После этого в окне профилировщика должна появиться собранная им информация.

Окно профилировщика

А теперь рассмотрим основные функции профилировщика.

Обратите внимание, что в этом разделе описываются функции профилировщика Unity 5. В этой версии появились новые функции,

которые в профилировщике Unity 4 могут называться по-другому или вообще отсутствовать.

Окно профилировщика делится на три основные области:

- панель управления;
- временная шкала;
- подробное описание.

Эти области показаны на рис. 1.6.

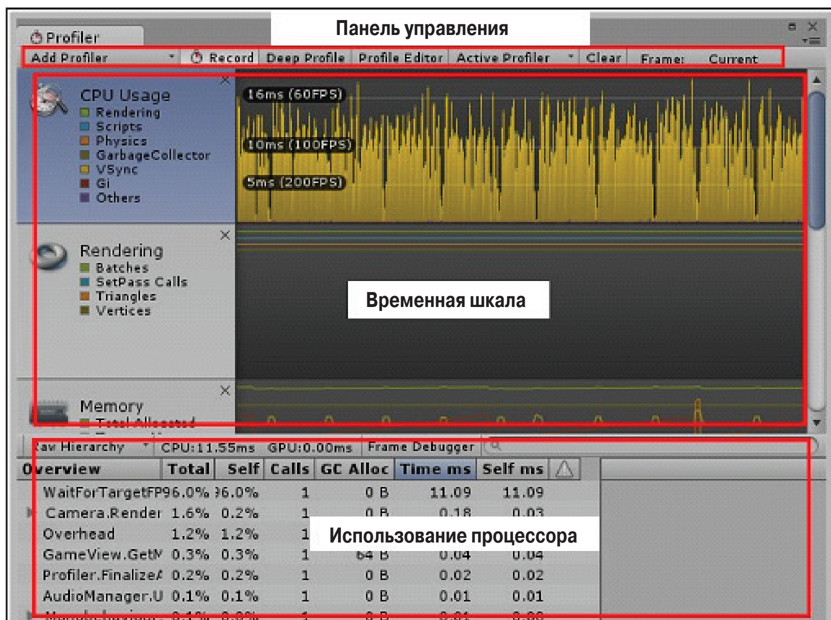


Рис. 1.6 ❖ Окно профилировщика

Элементы управления

Верхняя панель содержит несколько элементов управления профилированием и глубиной сбора системных данных:

- **Add Profiler** (Добавить профилировщик): по умолчанию профилировщик отображает в области временной шкалы несколько компонентов движка Unity, но с помощью пункта **Add Profiler** можно добавлять другие элементы. Полный список компонентов, доступных для профилирования, приводится в разделе «*Временная шкала*».

- **Record** (Запись): включение этого параметра заставит профилировщик непрерывно записывать данные профилирования. Обратите внимание, что данные записываются только в режиме воспроизведения (и если не включена пауза) или если установлен параметр **Profile Editor** (Профилирование редактора).
- **Deep Profile** (Глубокое профилирование): при обычном профилировании записываются только время и память, использованная всеми методами обратного вызова, такими как `Awake()`, `Start()`, `Update()`, `FixedUpdate()`, и т. д. Включение параметра **Deep Profile** (Глубокое профилирование) приведет к повторной компиляции сценариев для обеспечения возможности отслеживания характеристик всех вызываемых методов. Это вызовет увеличение накладных расходов во время выполнения и значительный рост использования памяти, поскольку отбираемые данные будут охватывать весь стек вызовов. Как следствие глубокое профилирование невозможно производить на больших проектах при слабом оборудовании, поскольку Unity может не хватить памяти даже для запуска тестирования!

Обратите внимание, что переход к глубокому профилированию требует повторной компиляции проекта, поэтому следует избегать переключения этого параметра во время выполнения.

  Поскольку установка этого параметра приводит к слепому профилированию всех методов, он редко используется на практике. Это параметр следует использовать, только когда профилирование по умолчанию не дало результатов или для небольших тестовых сцен, использующих небольшое подмножество функций игры.

  Если большой проект или сцена требует глубокого профилирования, а установка параметра **Deep Profile** (Глубокое профилирование) сильно затрудняет выполнение, можно использовать альтернативные способы, описанные в разделе «Выборочное профилирование сегментов кода» ниже;
- **Profile Editor** (Профилирование редактора): этот параметр включает профилирование редактора, то есть производится сбор данных профилирования для самого редактора Unity. Он может пригодиться для профилирования нестандартных редакторов сценариев, разработанных пользователями;

Обратите внимание, что перед использованием этой функции следует выбрать параметр **Editor** (Редактор) в меню **Active Profiler** (Активный профилировщик).

- **Active Profiler** (Активный профилировщик): это раскрывающееся меню с вариантами выбора целевого экземпляра Unity для профилирования. Как уже упоминалось, таким экземпляром может быть редактор, а также экземпляр автономного, локального или удаленного приложения;
- **Clear** (Очистка): очищает данные профилирования в области временной шкалы;
- **Frame Selection** (Выбор кадра): счетчик кадров **Frame** отображает количество кадров, подвергшихся профилированию, и текущий кадр в шкале времени. Две кнопки позволяют перемещаться между кадрами вперед и назад, и еще одна кнопка (**Current** (Текущий)) выбирает последний, текущий кадр, и сохраняет эту позицию. Таким образом, в разделе «Подробное описание» во время профилирования всегда будут отображаться данные для текущего кадра;
- **Timeline View** (Временная шкала): временная шкала отображает данные, организованные по областям, в зависимости от исследуемого компонента движка.

В каждой области выводится несколько графиков разного цвета для различных подразделов соответствующих компонентов. Эти графики могут скрываться и отображаться по желанию.

Каждая область специализируется на профилировании различных компонентов движка Unity. При выборе области во временной шкале в разделе использования процессора появится информация о соответствующем компоненте для выбранного кадра.

В разделе «Подробное описание» может отображаться самая разная информация, в зависимости от выбранной области.

Области можно удалять из временной шкалы щелчком на значке «X» в правом верхнем углу области. Чтобы восстановить закрытую область, выберите соответствующий пункт в меню **Add Profiler** (Добавить профилировщик).

Область использования центрального процессора

Раздел использования центрального процессора отображает информацию об использовании центрального процессора подсистемами Unity, такими как компоненты MonoBehaviour, камеры, процессы

визуализации, процессы моделирования физической среды, пользовательский интерфейс (включая интерфейс редактора при профилировании редактора), аудиопроцессы, сам профилировщик и т. д.

Существуют три режима отображения данных об использовании центрального процессора в разделе «Подробное описание»:

- иерархия;
- простая иерархия;
- шкала времени.

В режиме **Hierarchy Mode** (Режим иерархии) схожие элементы данных и вызовы глобальных функций Unity для удобства группируются вместе, например в этом режиме объединяются вызовы разделителей визуализации, такие как `BeginGUI()` и `EndGUI()`.

В режиме **Raw Hierarchy Mode** (Режим простой иерархии) глобальные вызовы функций Unity не группируются. Это, как правило, усложняет анализ содержимого раздела «Подробное описание», но позволяет подсчитать, сколько раз был вызван конкретный глобальный метод, или определить, не потребовал ли один из этих вызовов больших затрат процессорного времени и памяти, чем ожидалось. Например, вызовы `BeginGUI()` и `EndGUI()` будут представлены отдельными элементами, что приведет к захламлению раздела и затруднит его чтение.

Наиболее полезным, пожалуй, режимом в разделе «Подробное описание» является режим **Timeline Mode** (Режим шкалы времени, не путайте его с разделом «Временная шкала»). В этом режиме отображаются затраты времени центрального процессора для текущего кадра в виде стека вызовов. Блоки в верхней части соответствуют методам, вызванным непосредственно движком Unity (например, методы `Start()`, `Awake()` или `Update()`), а блоки под ними представляют методы, вызванные этими методами, которые, в свою очередь, могут включать вызовы методов других компонентов или объектов.

Ширина каждого блока определяется относительным временем выполнения метода в сравнении с окружающими его блоками. Кроме того, методы, выполняющиеся относительно быстро в сравнении с более *тяжелыми* методами, отображаются серыми прямоугольниками, чтобы не заострять на них внимания.

Раздел «Подробное описание» в режиме временной шкалы предлагает наглядный и хорошо организованный способ, позволяющий выявить наиболее тяжелые методы в стеке вызовов и оценить, как время их работы соотносится со временем выполнения других методов, вызываемых в этом же кадре. Это значительно облегчает выявление виновников проблем.

Например, предположим, что необходимо выявить причину проблемы с производительностью по результатам, изображенным на рис. 1.7. Здесь сразу бросаются в глаза три метода, затрачивающих на выполнение примерно одинаковое время, о чем свидетельствует примерно равная ширина представляющих их блоков.

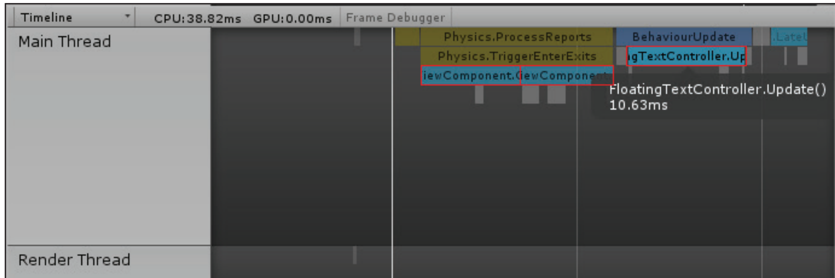


Рис. 1.7 ❖ Результаты профилирования с тремя проблемными методами

Здесь имеются целых три метода, за счет улучшения которых можно повысить производительность, что увеличивает вероятность найти фрагмент, который можно оптимизировать. Но улучшение производительности каждого метода приведет к уменьшению общего времени обработки кадра лишь на одну треть. Следовательно, придется проанализировать и оптимизировать все три метода, чтобы минимизировать время обработки кадра.

Область использования процессора будет подробно рассматриваться в *главе 2 «Приемы разработки сценариев»*.

Область использования графического процессора

Область использования графического процессора (GPU Area) аналогична области использования центрального процессора, но отображает вызовы методов и их длительность, которые выполняются на графическом процессоре. Соответственно, здесь отображаются вызовы методов Unity, связанных с камерами, рисованием, освещением, тенями и т. д.

Область использования графического процессора будет подробно рассматриваться в *главе 6 «Динамическая графика»*.

Область визуализации

Область визуализации (Rendering Area) содержит статистику визуализации, например количество вызовов SetPass, общее количество

пакетов для отображения сцены, количество пакетов, сохраненных при динамической и статической обработке, потребление памяти для текстур и т. д.

Область визуализации будет подробно рассматриваться в *главе 3 «Преимущества пакетной обработки»*.

Область памяти

Область памяти (Memory Area) позволяет исследовать потребление памяти приложением в двух режимах:

- простой режим;
- подробный режим.

Простой режим (Simple Mode) позволяет получить лишь общее представление о потреблении памяти такими компонентами, как низкоуровневый движок Unity, фреймворк Mono (размер общей кучи, обрабатываемой сборщиком мусора), графика, аудио (FMOD) и даже сам профилировщик.

Подробный режим (Detailed Mode) позволяет оценить потребление памяти отдельными игровыми объектами и компонентами в виде их собственных управляемых представлений. В этом режиме отображается дополнительный столбец, поясняющий причину потребления памяти объектом, и когда она может быть освобождена.

Область памяти будет подробно рассматриваться в *главе 7 «Мастерство управления памятью»*.

Область аудио

Область аудио (Audio Area) дает возможность исследовать статистику использования аудиоустройств и позволяет оценить, какую нагрузку оказывает аудиосистема на центральный процессор, а также общий объем памяти, потребляемой **источниками звука** (для выполняющих воспроизведение и в режиме паузы).

Область аудио подробно будет рассматриваться в *главе 4 «Привнесение искусства»*.

Об аудиосистеме часто забывают, когда дело касается улучшения производительности, однако функции воспроизведения аудио могут стать главным источником проблем, если не управлять ими должным образом. Поэтому периодически проверяйте статистику использования памяти и центрального процессора аудиосистемой.

Область физических движков 3D/2D

Имеются две области физических движков: одна – для 3D-системы (Nvidia PhysX) и другая – для 2D-системы (Box2D), – интегрированных в Unity, начиная с версии 4.5. Эти области отображают разнообразные статистические данные для таких компонентов, как `Rigidbody`, `Collider` и счетчики контактов.

Эта область будет рассматриваться в *главе 5 «Разгон физического движка»*.

На момент публикации этой книги в последней версии Unity v5.2.2f1 область Physics3D Area предоставляла незначительное количество элементов, в то время как область Physics2D Area предлагала значительно больше информации.

Методы анализа производительности

Применение надлежащих приемов программирования и управления ресурсами проекта значительно упрощает поиск причин низкой производительности, и в этом случае единственной реальной проблемой остается оптимизация кода. Например, если метод состоит из единственного гигантского цикла `for`, можно с уверенностью утверждать, что проблема связана либо с количеством итераций, либо со временем выполнения отдельных итераций.

Конечно, значительная часть кода не всегда отличается ясностью, поэтому придется время от времени выполнять профилирование подозрительных участков кода. Иногда *грубые, неуниверсальные* решения неизбежны, и не всегда появляется время вернуться назад и реорганизовать код, чтобы привести его в соответствие с рекомендуемыми методиками.

Легко упустить из виду очевидное, когда устранение проблемы и оптимизация производительности являются лишь еще одной формой решения той же задачи. Цель использования профилировщиков и анализа собранных ими данных заключается в поиске источника проблемы и определении ее важности. Очень часто можно получить неверные данные или сделать неправильные выводы из-за нетерпения или отсутствия желания разбираться в тонкостях. Часто при отладке программного обеспечения выясняется, что причину можно было найти гораздо быстрее, если остановиться и еще раз проверить сделанные ранее предположения. Уверенность, что проблема весьма

сложна и специфична, является хорошей предпосылкой для бесцельной траты драгоценного времени и сил. Это же относится и к анализу производительности.

Следующий контрольный список позволит сосредоточиться на конкретных вопросах, а не тратить время на войну с «призраками». Каждый проект своеобразен и имеет свой набор проблем и свою структуру, но следующий перечень применим к любому Unity-проекту:

- проверка присутствия целевого сценария в сцене;
- проверка появления сценария в сцене определенное число раз;
- сведение к минимуму изменений в текущем коде;
- сведение к минимуму внутренних помех;
- сведение к минимуму внешних помех.

Проверка присутствия сценария

Иногда случается так, что мы видим не то, что ожидали увидеть. Обычно такие ситуации легко заметить, потому что человеческий мозг очень хорошо справляется с распознаванием образов. Если что-то не соответствует ожидаемому образу, это сразу бросается в глаза. Но с работой внутренних механизмов дело обстоит иначе. Несоответствие ожиданиям в этом случае заметить, как правило, труднее, потому что мы легко замечаем лишь то, что имеет визуальные отличия. Проверка предполагаемого порядка событий чрезвычайно важна, пренебрежение ею ведет к поспешным выводам и потере драгоценного времени.

В контексте Unity это означает, что важно убедиться в присутствии сценария в сцене, который должен возбуждать события, и вызовы методов производятся в установленном порядке.

Присутствие сценария легко проверить, введя следующий текст в поле ввода в окне **Hierarchy** (Иерархия):

```
t:<monobehaviour name>
```

Например, если ввести текст `t:mytestmonobehaviour` (примечание: регистр букв не важен), в окне **Hierarchy** появится список всех игровых объектов с подключенным компонентом сценария `MyTestMonobehaviour`, как показано на рис. 1.8.

Обратите внимание, что список будет также включать все игровые объекты с компонентами, имена которых содержат имя искомого сценария.

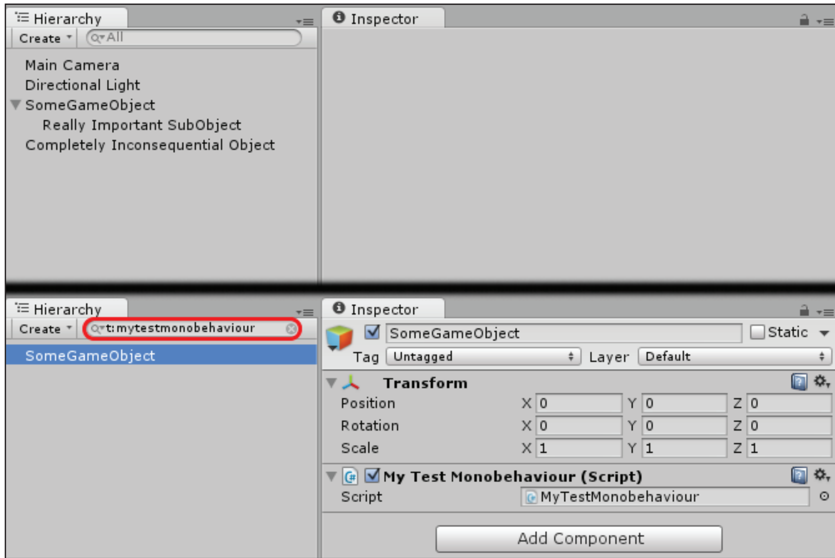


Рис. 1.8 ❖ Игровые объекты с подключенным компонентом сценария `MyTestMonobehaviour`

Дважды перепроверьте список объектов, поскольку они могли быть отключены при тестировании или просто случайно.

Проверка количества сценариев

Если исходить из предположения, что компонент вида MonoBehaviour, вызывающий проблемы с производительностью, присутствует в единственном экземпляре, можно по ошибке проигнорировать вероятность конфликта при многократном вызове метода. Такая позиция весьма опасна, потому что не учитывает возможности создания двух и более объектов в сцене, преднамеренно или случайно. Картина, наблюдаемая в профилировщике, может быть следствием многократного вызова затратного метода. Эту ситуацию следует дважды перепроверить с помощью списка, упомянутого выше.

Если в сцене компонент должен присутствовать в единственном экземпляре, а в списке он наблюдается более одного раза, следует пересмотреть сделанное ранее предположение о причине проблемы. Возможно, имеет смысл написать код инициализации, предотвращающий такое дублирование, или реализовать вспомогательное расши-

рение для редактора, отображающее предупреждение при попытке выполнить действие, которое может привести к этой ошибке.

Предотвращение подобных случайных ошибок имеет важное значение для сохранения хорошей производительности, поскольку, как показывает опыт, если что-то явно не запретить, тогда кто-то, где-то, в какой-то момент, по какой-то причине обязательно сделает это, и поиск такой ошибки потребует затратить впустую массу усилий.

Сведение к минимуму изменений в текущем коде

Я не рекомендую изменять код приложения, пытаясь отыскать причины проблем с производительностью. Такие изменения легко забываются с течением времени. Прием добавления вывода отладочных сообщений выглядит заманчиво, но подумайте, сколько будет затрачено времени на добавление необходимых вызовов методов, повторную компиляцию кода *и* удаление этих вызовов после завершения анализа. Кроме того, если в дальнейшем забыть удалить эти вызовы, они повлекут дополнительные накладные расходы в окончательной сборке, поскольку средства отладки в Unity весьма дорогостоящи в смысле ресурсов центрального процессора и памяти.

Один из способов решения этой проблемы заключается в использовании инструментов управления версиями, позволяющих отслеживать изменения в файлах и при необходимости откатывать их в первоначальное состояние. Это отличный способ исключить ненужные изменения из окончательной версии.

Однако лучше всего при отладке использовать точки останова, поскольку они позволяют исследовать стек вызовов, значения переменных и выполнение кода по условию (например, блоки `if-else`), не требуя вносить ненужные изменения в код или тратить время на повторную компиляцию.

Сведение к минимуму внутренних помех

Редактор Unity не лишен собственных маленьких причуд и нюансов, которые могут вызвать недоумение.

Во-первых, если обработка единственного кадра занимает настолько значительное время, что приводит к заметному замиранию игры, профилировщик может не успеть собрать результаты и вывести их в своем окне. Это особенно раздражает при попытке сбора данных в момент инициализации приложения или сцены. Альтернативные

способы решения этой проблемы описываются в разделе «*Нестандартное профилирование центрального процессора*» ниже.

Еще одна распространенная ошибка (я сам неоднократно допускал ее, работая над книгой): тестируя реакцию на нажатия клавиш, когда окно профилировщика уже открыто, не забудьте предварительно щелкнуть на окне игры! Если последний щелчок был сделан на окне профилировщика, редактор передаст событие нажатия клавиши профилировщику вместо приложения, соответственно, это событие не будет перехвачено ни одним игровым объектом.

Вертикальная синхронизация (другое название – **VSync**) используется для приведения частоты кадров приложения в соответствие с частотой кадров устройства, отвечающего за визуализацию (например, монитора). При включении этой функции профилировщик будет генерировать массу всплесков в графике использования центрального процессора с заголовками **WaitForTargetFPS** (ожидание, связанное с ограничением частоты кадров на целевом устройстве), отражающих преднамеренное замедление приложения для синхронизации с частотой кадров устройства визуализации. Этот ненужный мусор в графике только затрудняет выявление реальных проблем. Обязательно отключайте функцию вертикальной синхронизации перед изучением всплесков нагрузки на центральный процессор, выполняя тестирование производительности. Отключить эту функцию можно, выбрав пункт **Edit** ⇨ **Project Settings** ⇨ **Quality** (Редактор ⇨ Настройки проекта ⇨ Качество) и затем вкладку, соответствующую выбранной платформе.

Также следует удостовериться, что падение производительности не является прямым результатом вывода множества сообщений об исключениях и ошибках в консоль редактора. Метод `Debug.Log()` и аналогичные ему, такие как `Debug.LogError()`, `Debug.LogWarning()` и др., оказывают существенную нагрузку на центральный процессор и используют динамическую память, что, в свою очередь, влечет накопление мусора, на уборку которого затрачивается еще больше тактов процессорного времени.

Эти накладные расходы выглядят незначительными при исследовании проекта в редакторе, где большинство сообщений об ошибках генерируется компилятором или неправильно настроенными объектами. Но они могут стать причиной проблем при выполнении, особенно под управлением профилировщика, когда любые помехи крайне нежелательны. Например, если отсутствует ссылка на объект, которая должна быть присвоена в редакторе, и она используется

в методе `Update()`, единственный компонент MonoBehaviour будет генерировать новое исключение при каждом обновлении. Это приведет к включению массы ненужной информации в данные профилирования.

Обратите внимание, что в режиме воспроизведения проекта можно сбросить флаги **Info** (Информация) и **Warning** (Предупреждения) (как показано на рис. 1.9), правда, на выполнение операторов отладки все равно будут использоваться ресурсы центрального процессора и память, но сами сообщения выводиться больше не будут. Однако лучше оставить эти параметры включенными, чтобы не пропустить ничего важного.

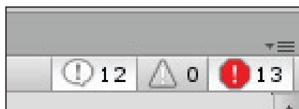


Рис. 1.9 ❖ Флаги **Info** (Информация) и **Warning** (Предупреждения)

Сведение к минимуму внешних помех

Заключается в одном простом, но абсолютно необходимом действии: дважды проверьте отсутствие фоновых процессов со значительным потреблением вычислительных ресурсов или оперативной памяти. Недостаток памяти отрицательно сказывается на тестировании, так как может вызывать увеличение промахов кэша и обращений к жесткому диску для размещения страниц в виртуальной памяти, увеличивая время отклика приложения.

Выборочное профилирование сегментов кода

Если проблему производительности не получилось решить с помощью контрольного списка, приведенного выше, это может означать, что она имеет более глубокие корни и необходимо провести дополнительный анализ. Задание остается прежним: выяснить причину проблемы. Окно профилировщика предоставляет массу информации о производительности. С ее помощью можно найти конкретные кадры, требующие анализа, и определить, какой сценарий и/или метод является причиной проблемы. Затем проблему нужно локализовать,

то есть выяснить, воспроизводима ли проблема, при каких обстоятельствах она возникает и какой фрагмент кода ее порождает.

Чтобы ответить на эти вопросы, нужно выполнить профилирование определенных разделов кода. Решить эту задачу можно несколькими способами, которые в действительности делятся на две категории:

- управление профилировщиком из сценария;
- нестандартные методы хронометража и регистрации.

Обратите внимание, что следующий раздел главным образом посвящен поиску узких мест в сценариях на C#. Поиск узких мест в других компонентах движка будет рассматриваться в соответствующих главах.

Управление профилировщиком из сценариев

Управление профилировщиком из сценариев осуществляется посредством статического класса `Profiler`. Он имеет несколько удобных методов, описание которых можно найти в документации к Unity. Наиболее важными для нас являются методы-разделители, активирующие и деактивирующие профилирование во время выполнения: `Profiler.BeginSample()` и `Profiler.EndSample()`.

Обратите внимание, что методы `BeginSample()` и `EndSample()` компилируются только в отладочном режиме – в режиме сборки окончательной версии они не включаются в выполняемый код и не влекут никаких накладных расходов. Поэтому их без всякого риска можно оставлять в коде на случай, если позднее потребуется повторить профилирование.

Для метода `BeginSample()` имеется перегруженная версия, позволяющая задать произвольное имя для области, отображаемой в окне профилировщика в режиме иерархии. Например, следующий код запустит профилирование и создаст область с указанным заголовком:

```
void DoSomethingCompletelyStupid() {
  Profiler.BeginSample("My Profiler Sample");

  List<int> listOfInts = new List<int>();
  for(int i = 0; i < 1000000; ++i) {
    listOfInts.Add(i);
  }

  Profiler.EndSample();
}
```

Загрузка исходного кода примеров
Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Можно ожидать, что этот метод (создает список с миллионом целых чисел, но никах их не использует) вызовет всплеск потребления процессорного времени, сожрет несколько мегабайт памяти и появится в разделе с подробным описанием под заголовком *My Profiler Sample*, как показано на рис. 1.10.

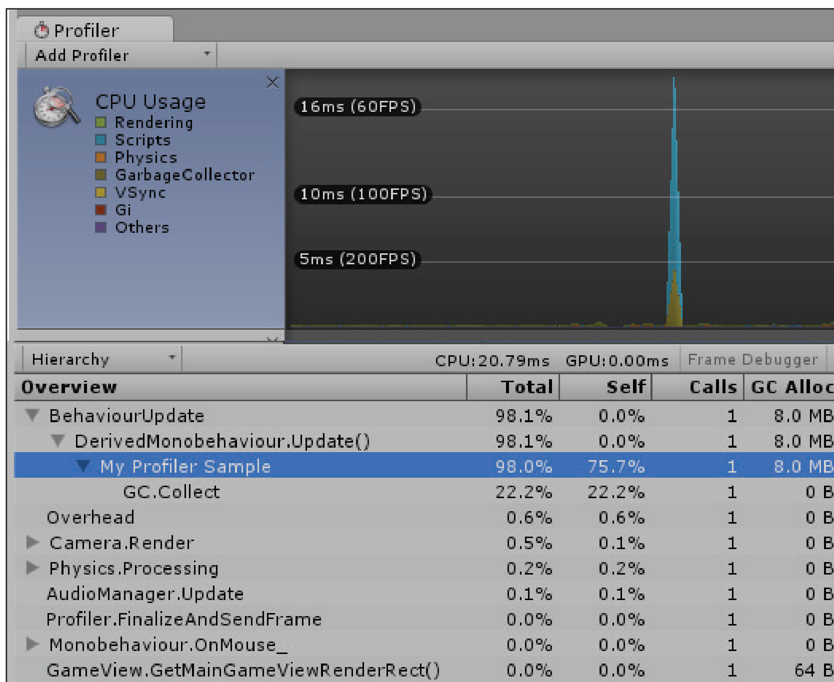


Рис. 1.10 ❖ Всплеск потребления процессорного времени и памяти

Обратите внимание, что при **глубоком профилировании** такие нестандартные имена выводятся на более низких уровнях иерархии. На рис. 1.11 показан тот же раздел с подробным описанием, но в режиме глубокого профилирования:

Hierarchy CPU:115.27ms GPU:0.00ms Frame Debugger

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ BehaviourUpdate	99.5%	0.0%	1	8.0 MB	114.80	0.00
▼ DerivedMonobehaviour.Update()	99.5%	0.0%	1	8.0 MB	114.79	0.00
▼ Profiler.BeginSample()	99.5%	0.0%	1	8.0 MB	114.78	0.10
▼ DerivedMonobehaviour.DoSomething	99.4%	91.3%	1	8.0 MB	114.68	105.36
▼ List`1.Add()	8.0%	0.1%	1000000	8.0 MB	9.32	0.22
▼ List`1.GrowIfNeeded()	7.8%	0.0%	19	8.0 MB	9.10	0.00
▼ List`1.set_Capacity()	7.8%	0.0%	19	8.0 MB	9.09	0.00
▼ Array.Resize()	7.8%	0.0%	19	8.0 MB	9.09	0.01
▼ Array.Resize()	7.8%	3.3%	19	8.0 MB	9.08	3.82
GC.Collect	4.1%	4.1%	1	0 B	4.74	4.74
▼ Array.Copy()	0.4%	0.0%	19	0 B	0.50	0.00
▼ Array.Copy()	0.4%	0.0%	19	0 B	0.50	0.00
Array.FastCopy	0.4%	0.4%	19	0 B	0.49	0.49
Array.GetLowerBc	0.0%	0.0%	38	0 B	0.00	0.00
List`1.get_Capacity()	0.0%	0.0%	19	0 B	0.00	0.00
List`1..ctor()	0.0%	0.0%	1	0 B	0.00	0.00
▼ Profiler.BeginSampleOnly()	0.0%	0.0%	1	0 B	0.00	0.00
My Profiler Sample	0.0%	0.0%	1	0 B	0.00	0.00

Рис. 1.11 ❖ Раздел с подробным описанием в режиме глубокого профилирования

Обратите внимание, что указанное имя оказалось ниже в иерархии, чем следовало бы. Остается неясным, что является причиной этого феномена, но он может привести к путанице при просмотре иерархии в режиме глубокого профилирования, поэтому просто помните об этой особенности.

Нестандартное профилирование использования центрального процессора

Профилировщик – лишь один из доступных инструментов. Иногда бывает необходимо выполнить нестандартное профилирование, например когда возникают сомнения в достоверности данных, получаемых с помощью профилировщика Unity, или накладные расходы на его работу кажутся слишком большими, или просто нужен полный контроль над всеми аспектами приложения. Независимо от мотивов освоение методов независимого анализа кода будет весьма полезно. В конце концов, маловероятно, что единственным нашим занятием до пенсии станет разработка игр в Unity.

Средства профилирования очень сложны, поэтому мы вряд ли сможем в разумные сроки создать собственное решение, сопоставимое с имеющимися. Для проверки использования центрального процессора обычно достаточно точной системы хронометража, быстрого и легковесного способа регистрации и конкретных фрагментов кода для тестирования. Так уж получилось, что библиотека .NET (или, официально, фреймворк Mono) предоставляет класс `Stopwatch` в про-

странстве имен System.Diagnostics. Объект класса Stopwatch (секундомер) можно останавливать, запускать и точно измерять время, прошедшее с момента запуска секундомера.

К сожалению, этот класс не особенно точен – его точность измеряется миллисекундами или, в лучшем случае, десятыми долями миллисекунд. Подсчет реального времени с высокой точностью, сравнимой с тактовой частотой процессора, является на удивление сложной задачей. Поэтому, чтобы не вникать в тонкости решения этой задачи, нужно попытаться найти способ обойтись точностью класса Stopwatch.

Прежде чем ставить перед собой задачу достижения высокой точности, следует выяснить, а так ли это необходимо. Большинство игр выполняется с частотой 30 или 60 кадров в секунду, это значит, что на обработку одного кадра затрачивается около 33 мс или 16 мс соответственно. То есть если, к примеру, нужно снизить время выполнения конкретного блока кода до 10 мс, тогда тысячекратное повторение теста для получения точности до микросекунд не принесет особой пользы.

Однако если точность имеет большое значение, повысить ее можно путем многократного запуска одного и того же теста. Предположим, что некий тест можно с легкостью повторить и само тестирование не занимает слишком много времени, тогда его можно запустить тысячи или даже миллионы раз, а затем разделить общее затраченное время на количество выполненных тестов и получить более точное время прогона одного теста.

Ниже приводится определение класса таймера, определяющего время выполнения заданного количества тестов с помощью объекта Stopwatch:

```
using UnityEngine; using System;
using System.Diagnostics; using System.Collections;

public class CustomTimer : IDisposable {
  private string m_timerName;
  private int m_numTests;
  private Stopwatch m_watch;

  // принимает имя таймера и количество запускаемых тестов
  public CustomTimer(string timerName, int numTests) {
    m_timerName = timerName;
    m_numTests = numTests;
    if (m_numTests <= 0)
      m_numTests = 1;
    m_watch = Stopwatch.StartNew();
  }
```

```
  // вызывается в конце блока 'using'
  public void Dispose() {
    m_watch.Stop();
    float ms = m_watch.ElapsedMilliseconds;
    UnityEngine.Debug.Log(string.Format("{0} finished: {1:0.00}ms total,
{2:0.000000}ms per test for {3} tests", m_timerName, ms, ms/ m_numTests, m_
numTests));
  }
}
```

Далее следует пример использования класса `CustomTimer`:

```
int numTests = 1000;

using (new CustomTimer("My Test", numTests)) {
  for(int i = 0; i < numTests; ++i) {
    TestFunction();
  }
} // метод таймера Dispose() будет вызван здесь автоматически
```

Этот подход имеет три особенности. Во-первых, рассчитывается только средняя продолжительность работы по нескольким вызовам метода. Если продолжительность работы значительно меняется от вызова к вызову, это не будет отражено в полученном среднем значении. Во-вторых, если метод использует общую память, тогда при многократном обращении к одним и тем же блокам памяти скорость будет искусственно завышена из-за кэширования, что уменьшит среднюю продолжительность работы относительно обычного режима. В-третьих, к такому же эффекту приведет JIT-компиляция, выполняемая только при первом вызове метода. Подробнее о JIT-компиляции рассказывается в *главе 7 «Мастерство управления памятью»*.

Блок `using` обычно используется для безопасного уничтожения неуправляемых ресурсов при выходе из него. В конце блока `using` автоматически вызывается метод `Dispose()` объекта, выполняющий необходимые операции очистки. Для этого объект должен реализовать интерфейс `IDisposable`, который требует определить метод `Dispose()`.

Однако эту особенность языка можно использовать также для создания обособленного блока кода, генерирующего временный объект, способный выполнить что-то полезное по завершении блока кода.

Не путайте блок `using` с оператором `using`, который используется в начале файла сценария для определения дополнительного пространства имен. Очень странно, что ключевое слово для управления пространствами имен в языке C# совпадает с другим ключевым словом.

Таким образом, блок `using` и класс `CustomTimer` предоставляют элегантный способ упаковки целевого тестового кода, наглядно демонстрирующий, когда и как он будет выполняться.

Также следует учитывать «разогрев» приложения. Запуск сцены является весьма дорогостоящей операцией, включающей вызовы методов `Start()` и `Awake()` множества игровых объектов, а также инициализацию прочих компонентов, таких как физическая система и система визуализации. Начальная задержка может составлять лишь одну секунду, но она способна значительно исказить результаты тестирования. Поэтому важно начинать тестирование после достижения приложением устойчивого состояния.

Следовательно, было бы целесообразно обернуть целевой блок кода методом, проверяющим ввод с клавиатуры `Input.GetKeyDown()`, для управления моментом его запуска. Например, следующий код выполнит тестирование только после нажатия клавиши *пробела*:

```
if (Input.GetKeyDown(KeyCode.Space)) {
  int numTests = 1000;

  using (new CustomTimer("Controlled Test", numTests)) {
    for(int i = 0; i < numTests; ++i) {
      TestFunction();
    }
  }
}
```

Класс `CustomTimer` имеет три важные конструктивные особенности: он выводит единственное сообщение только по завершении тестирования, извлекая значение времени из объекта класса `Stopwatch` после его остановки и используя `string.Format()` для создания строки.

Как отмечалось выше, механизм регистрации в Unity требует значительных ресурсов. Поэтому не следует использовать его методы внутри тестов профилирования (и вообще в игровом процессе, коли на то пошло). Если понадобятся подробные сведения (например, продолжительность выполнения каждой итерации в цикле, чтобы выявить те из них, что потребовали больше времени), эти сведения разумнее кэшировать и выводить все вместе по завершении тестирования, как в классе `CustomTimer`. Это позволит сократить накладные расходы во время выполнения за счет небольшого увеличения потребления памяти. Альтернативное решение, когда в течение всего тестирования каждое сообщение выводится методом `Debug.Log()` с затратой многих миллисекунд, значительно исказит результаты.

Вторая особенность – остановка объекта класса `Stopwatch` перед чтением его значения. Очевидно, что результат чтения, пока счет времени продолжается, будет отличаться от результата чтения после предварительной остановки и последующего возобновления таймера. Без глубокого анализа исходного кода проекта Mono (и конкретной версии Unity) нельзя точно определить, как объект класса `Stopwatch` осуществляет счет времени и в какие моменты операционная система переключается между приложениями. Поэтому имеет смысл проявить осторожность и остановить счет перед чтением значения.

И наконец, что касается использования метода `string.Format()`. Более подробно этот метод будет рассмотрен в *главе 7 «Мастерство управления памятью»*, а сейчас лишь поясним, что он используется потому, что оператор + конкатенации строк потребляет удивительно большой объем памяти и тем самым привлекает внимание сборщика мусора. А это противоречит главной цели – точности хронометража и анализа.

Сохранение и загрузка данных профилировщика

В настоящее время профилировщик Unity обладает следующими существенными недостатками, касающимися сохранения и загрузки данных профилирования:

- в окне профилировщика может отображаться не более 300 кадров;
- в интерфейсе пользователя не предусмотрена возможность сохранения данных;
- внутри сценария можно сохранить двоичные данные профилирования в файл, но нет встроенной возможности просмотреть эти данные.

Эти недостатки значительно усложняют выполнение широкомасштабного и длительного тестирования с применением профилировщика Unity. Вопрос о них поднимается на трекере проблем Unity в течение нескольких лет, но пока безрезультатно. Поэтому в решении этой проблемы придется полагаться только на собственную изобретательность.

К счастью, класс Profiler имеет несколько методов управления регистрацией информации профилировщиком:

1. Свойство `Profiler.enabled` можно использовать для включения и выключения профилировщика, оно действует подобно щелчку на кнопке **Record** (Запись) в панели управления.

Обратите внимание, что метод `Profiler.enabled` не оказывает никакого влияние на состояние кнопки Record в панели управления. Это может вызвать путаницу при одновременном управлении профилировщиком из кода и пользовательского интерфейса.

2. Значение свойства `Profiler.logFile` определяет путь к файлу регистрации для вывода данных. Имейте в виду, что в этот файл выводятся только сведения об изменении частоты кадров приложения с течением времени – туда не попадают остальные данные, отображаемые в разделе временной шкалы. Для сохранения этих данных в двоичном виде потребуется использовать один из описанных ниже способов.
3. Свойство `Profiler.enableBinaryLog` включает и выключает запись двоичных данных в дополнительный файл, отображаемый в разделе временной шкалы и подробной информации. Местоположение файла определяется свойством `Profiler.logFile`, но с добавлением расширения `.data`.

С помощью этих свойств можно создать простой инструмент сохранения данных, способный записывать большие объемы данных, разделяя их по нескольким файлам для последующего анализа.

Сохранение данных профилировщика

При создании инструмента, сохраняющего данные, собранные профилировщиком, будут использованы **сопрограммы**. Обычные методы выполняются целиком, от начала до конца. В отличие от них, сопрограммы позволяют создавать методы записи, способные приостанавливаться до определенного момента времени или появления заданного события. Это называется согласованием и реализуется с помощью оператора `yield`. Тип согласования определяет условие возобновления выполнения. Типы согласований перечислены ниже (вместе с объектами, которые передаются оператору `yield`):

- после определенного периода времени (`WaitForSeconds`);
- после выполнения следующего метода Update (`WaitForEndOfFrame`);
- после выполнения следующего метода FixedUpdate (`WaitForFixedUpdate`);
- непосредственно перед выполнением метода LateUpdate (`null`);
- после того как объект `WWW` завершит текущее задание, например загрузит файл (`WWW`);
- после завершения другой сопрограммы (ссылка на другую сопрограмму).

Документация Unity по сопрограммам содержит дополнительную информацию об использовании этих полезных инструментов в движке Unity:

- http://docs.unity3d.com/Manual/Coroutines.html;
- http://docs.unity3d.com/Manual/ExecutionOrder.html.

Не путайте сопрограммы с потоками, которые выполняются независимо от основного потока Unity. Сопрограммы всегда выполняются в главном потоке вместе с остальной частью кода и просто приостанавливают и возобновляют выполнение в определенные моменты в зависимости от объекта, переданного оператору `yield`.

А теперь вернемся к поставленной задаче. Ниже приводится определение класса компонента `ProfilerDataSaverComponent`, который с помощью сопрограммы повторяет действия через каждые 300 кадров:

```
using UnityEngine;
using System.Text;
using System.Collections;

public class ProfilerDataSaverComponent : MonoBehaviour {

  int _count = 0;

  void Start() {
    Profiler.logFile = "";
  }

  void Update () {
    if (Input.GetKey (KeyCode.LeftControl) && Input.GetKeyDown (KeyCode.H)) {
      StopAllCoroutines();
      _count = 0;
      StartCoroutine(SaveProfilerData());
    }
  }

  IEnumerator SaveProfilerData() {
    // этот метод продолжает вызываться, пока
    // не будет выключен режим воспроизведения
    while (true) {

      // сгенерировать путь к файлу
      string filepath = Application.persistentDataPath + "/profilerLog" + _count;

      // включить регистрацию и профилирование
      Profiler.logFile = filepath;
      Profiler.enableBinaryLog = true;
      Profiler.enabled = true;
```

```
      // отсчитать 300 кадров
      for(int i = 0; i < 300; ++i) {

        yield return new WaitForEndOfFrame();

        // чтобы сохранить профилировщик включенным
        if (!Profiler.enabled)
          Profiler.enabled = true;
      }

      // начать снова с другим именем файла
      _count++;
    }
  }
}
```

Попробуйте подключить этот компонент к любому игровому объекту в сцене и нажать одновременно клавиши **Ctrl**+*H* (пользователям OSX нужно заменить метод `KeyCode.LeftControl` на `KeyCode.LeftCommand`). Профилировщик начнет сбор и сохранение информации (даже если окно профилировщика закрыто!) в файлы, находящиеся в каталоге, указанном в параметре `Application.persistantDataPath`.

Обратите внимание, что значение `Application.persistantDataPath` зависит от операционной системы. Более подробную информацию можно найти в документации Unity на странице http://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html.

Не следует сохранять файлы в каталоге `Application.dataPath`, иначе они окажутся в рабочей области проекта. Профилировщик не освобождает последний дескриптор регистрационного файла журнала ни при отключении, ни даже при выходе из режима воспроизведения. В рабочей области проекта это может вызывать конфликты доступа к файлам между редактором Unity, пытающимся читать и записывать дополнительные файлы метаданных, и профилировщиком, захватившим дескриптор для последнего файла регистрации, и приводить к очень неприятным ошибкам доступа к файлам, часто заканчивающимся крахом редактора Unity и потерей всех несохраненных изменений в сцене.

Запись данных каждые 300 кадров вызывает небольшие накладные расходы, связанные с использованием жесткого диска и контекста `IEnumerator`, они появляются при создании каждого файла и составляют несколько миллисекунд процессорного времени (в зависимости от оборудования).

Каждая пара файлов должна содержать данные профилировщика о 300 кадрах, которые можно отобразить в окне профилировщика. Нам осталось лишь найти способ вывода этих данных.

На рис. 1.12 можно видеть список файлов данных, созданных компонентом `ProfilerDataSaverComponent`.

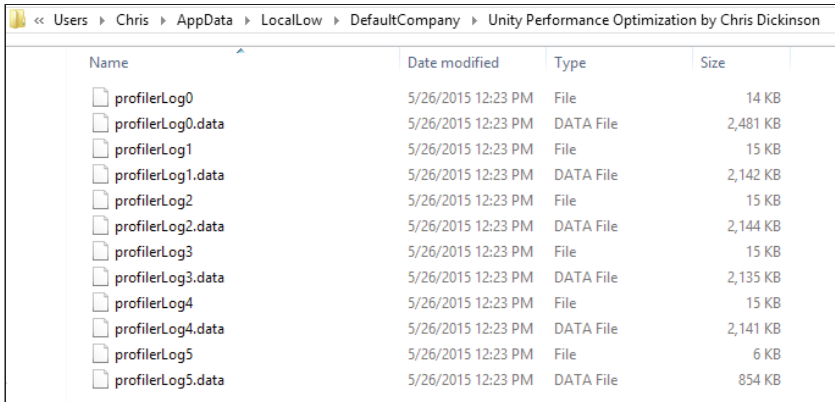


Рис. 1.12 ❖ Список файлов, созданных компонентом `ProfilerDataSaverComponent`

Обратите внимание, что первый файл может содержать менее 300 кадров, если несколько кадров было потеряно за время разгона профилировщика.

Загрузка данных профилировщика

Метод `Profiler.AddFramesFromFile()` загрузит указанную пару файлов (текстовый и двоичный) и поместит их данные во временную шкалу в окне профилировщика, сдвинув присутствующие в ней данные на более поздние моменты времени. Так как каждый файл содержит данные о 300 кадрах, они идеально подходят в данном случае. Остается только создать класс `EditorWindow`, предоставляющий набор кнопок для загрузки файлов в профилировщик.

Обратите внимание, что методу `AddFramesFromFile()` требуется передать только имя оригинального регистрационного файла. Он автоматически найдет соответствующий двоичный файл с расширением .data.

Ниже приводится определение класса для создания окна `ProfilerDataLoaderWindow`:

```
using UnityEngine;
using UnityEditor;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class ProfilerDataLoaderWindow : EditorWindow {

  static List<string> s_cachedFilePaths;
  static int s_chosenIndex = -1;

  [MenuItem ("Window/ProfilerDataLoader")]
  static void Init() {
    ProfilerDataLoaderWindow window = (ProfilerDataLoaderWindow)EditorWindow.
GetWindow (typeof(ProfilerDataLoaderWindow));
    window.Show ();

    ReadProfilerDataFiles ();
  }

  static void ReadProfilerDataFiles() {
    // гарантировать освобождение дескрипторов
    // всех файлов, загружаемых профилировщиком
    Profiler.logFile = "";

    string[] filePaths = Directory.GetFiles(Application.persistentDataPath,
"profilerLog*");

    s_cachedFilePaths = new List<string> ();

    // игнорировать двоичные файлы с расширением .data.
    // Профилировщик найдет их самостоятельно
    Regex test = new Regex (".data$");

    for (int i = 0; i < filePaths.Length; i++) {
      string thisPath = filePaths [i];

      Match match = test.Match (thisPath);

      if (!match.Success) {
        // не двоичный файл, добавить в список
        Debug.Log ("Found file: " + thisPath);
        s_cachedFilePaths.Add (thisPath);
      }
    }
```

```
    s_chosenIndex = -1;
  }

  void OnGUI () {
    if (GUILayout.Button ("Find Files")) {
    ReadProfilerDataFiles();
    }

    if (s_cachedFilePaths == null)
      return;

    EditorGUILayout.Space ();

    EditorGUILayout.LabelField ("Files");

    EditorGUILayout.BeginHorizontal ();

    // создать стили для кнопок и выделить надпись
    // последней выбранной кнопки красным цветом
    GUIStyle defaultStyle = new GUIStyle(GUI.skin.button);
    defaultStyle.fixedWidth = 40f;

    GUIStyle highlightedStyle = new GUIStyle (defaultStyle);
    highlightedStyle.normal.textColor = Color.red;

    for (int i = 0; i < s_cachedFilePaths.Count; ++i) {

      // по 5 элементов списка в строке
      if (i % 5 == 0) {
        EditorGUILayout.EndHorizontal ();
        EditorGUILayout.BeginHorizontal ();
      }

      GUIStyle thisStyle = null;

      if (s_chosenIndex == i) {
        thisStyle = highlightedStyle;
      } else {
        thisStyle = defaultStyle;
      }

      if (GUILayout.Button("" + i, thisStyle)) {
        Profiler.AddFramesFromFile(s_cachedFilePaths[i]);

        s_chosenIndex = i;
      }
    }

    EditorGUILayout.EndHorizontal ();
  }
}
```

При создании любых окон EditorWindow прежде всего нужно создать пункт меню с атрибутом [MenuItem] и экземпляр объекта Window для элементов управления. И то, и другое реализовано в методе Init().

В процессе инициализации также вызывается метод ReadProfilerDataFiles(). Он читает содержимое каталога Application.persistantDataPath (куда сохранялись файлы компонентов ProfilerDataSaverComponent) и добавляет имена найденных файлов в кэш для дальнейшего использования.

И наконец, метод OnGUI(). Этот метод выполняет основную работу. Он создает кнопку для загрузки файлов по требованию, проверяет наличие имен файлов в кэше и генерирует набор кнопок для загрузки отдельных файлов в профилировщик. Он также выделяет последнюю нажатую кнопку, окрашивая надпись на ней в красный цвет с помощью стиля GUIStyle, что позволяет определить, содержимое какого файла отображается в окне профилировщика.

Окрыть окно ProfilerDataLoaderWindow можно, выбрав в меню пункт **Window ⇨ ProfilerDataLoader** (Окна ⇨ Загрузка данных профилировщика), как показано на рис. 1.13.

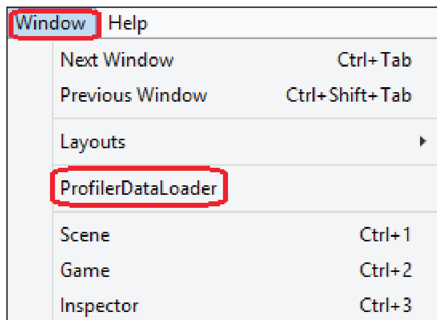


Рис. 1.13 ❖ Доступ к инструменту загрузки данных в профилировщик

На рис. 1.14 показано окно с несколькими файлами, которые можно загрузить. Щелчок на любой из пронумерованных кнопок приведет к загрузке в профилировщик данных, содержащихся в соответствующем файле.

Компонент ProfilerDataSaverComponent и окно ProfilerDataLoaderWindow не могут считаться законченным и полнофункциональным решением. Они служат лишь трамплином для дальнейшего развития этой темы.

Для большинства команд разработчиков и проектов трехсот кадров достаточно, чтобы определить, с чего начинать исправление проблем.

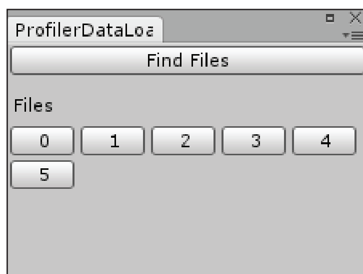


Рис. 1.14 ❖ Кнопки загрузки файлов

Заключительные соображения о профилировании и анализе

Одним из способов оптимизации производительности является «избавление от ненужных задач, на которые затрачиваются ценные ресурсы». Аналогично, путем минимизации затрачиваемых усилий, можно максимизировать свою производительность труда. Эффективное использование имеющихся средств играет первостепенную роль. Они сослужат хорошую службу, если оптимизировать свой рабочий процесс, основываясь на знании рекомендуемых методик.

Большинство, если не все советы по использованию любого рода инструментов сбора данных, может свести к трем:

- освойте инструмент;
- уменьшите шум;
- сфокусируйтесь на проблеме.

Освоение профилировщика

Профилировщик представляет собой хорошо продуманный и интуитивно понятный инструмент, поэтому для освоения большинства его функций не требуется тратить часы на работу с ним в тестовом проекте или чтение документации. Хорошее знание инструмента, его преимуществ и недостатков, возможностей и ограничений сделает более осмысленной предоставляемую им информацию, поэтому стоит потратить время на его использование в условиях, близких к реальным. Ведь никто не хочет столкнуться с сотней проблем производительно-

сти за две недели до выпуска, даже не представляя, как эффективно выполнить анализ производительности!

Например, никогда не упускайте из виду *относительного* характера графического представления временной шкалы. Всплески на временной шкале, которые выглядят огромными и угрожающими, не обязательно обозначают проблему с производительностью. Поскольку временная шкала не предоставляет значений по вертикальной оси и автоматически корректирует эту ось на основании содержимого последних 300 кадров, небольшие пики могут казаться более серьезной проблемой, чем есть на самом деле. Некоторые области шкалы времени содержат контрольные линии, помогающие оценить работу приложения в данный момент времени. Их можно использовать для оценки масштаба возникшей проблемы. Не позволяйте профилировщику обмануть себя, полагая, что высокие всплески всегда вызывают проблемы. И, как всегда, важно только то, что заметит пользователь!

Например, если самый большой всплеск, отражающий использование процессора, не пересекает контрольных линий в 60 или 30 кадров в секунду (в зависимости от частоты кадров целевого устройства), его целесообразнее игнорировать и заняться поиском других причин загрузки центрального процессора, поскольку улучшение соответствующего ему фрагмента кода конечный пользователь просто не заметит и это никак не повлияет на качество продукта.

Уменьшение шума

Под шумом в информатике подразумеваются «*бессмысленные данные*», и пакет данных профилирования, собранный без определенной цели только из соображений полноты информации, интереса не представляет. Большой объем данных требует больше времени на их осмысление и фильтрацию, что очень мешает. Чтобы избежать этого, просто сократите количество обрабатываемых данных, избавившись от информации, которая не является жизненно важной в данной ситуации.

После устранения помех в графическом интерфейсе профилировщика вам проще будет определить компонент, вызвавший всплеск потребления ресурсов. Не забывайте использовать цветные поля в каждой области временной шкалы для сужения пространства поиска. Но имейте в виду, что настройки автоматически сохраняются редактором, поэтому не забудьте снова включить их в следующем сеансе профилирования, иначе в дальнейшем можете упустить из виду что-то важное!

Кроме того, игровые объекты можно отключать, чтобы предотвратить сбор данных о них, если это позволит уменьшить путаницу. Естественно, отключение каждого объекта должно вызвать небольшой рост производительности. Но если после отключения очередного объекта производительность вдруг резко возрастает, можно с уверенностью утверждать, что причина проблемы связана с этим объектом.

Сосредоточение внимания на проблеме

Этот совет может показаться лишним, поскольку мы уже рассмотрели совет уменьшить шум. Но должны ли мы отмахнуться от него? Не совсем так. Под сосредоточенностью понимается умение не отвлекаться на несущественные задачи и сумасбродные затеи.

Напомним, что профилирование с помощью профилировщика Unity привносит определенное снижение производительности. При использовании режима глубокого профилирования **Deep Profiling** затраты значительно увеличиваются. Еще больше увеличивает затраты дополнительная регистрация. Об этом легко забыть, занимаясь профилированием несколько часов подряд.

Мы влияем на производительность, просто измеряя ее. Изменения, сделанные в процессе сбора данных, иногда могут вынудить нас гоняться за несуществующими ошибками, причем массу потраченного зря времени можно было бы сэкономить, просто запустив сценарий без профилирования. Если проблема поддается воспроизведению и заметна при выполнении без профилирования, ее определенно стоит начать исследовать. Но если новые проблемы появляются в процессе исследования существующих проблем, имейте в виду, что они могут быть обусловлены тестированием кода, не являясь в действительности вновь выявленными проблемами.

И наконец, когда профилирование закончено и все исправления внесены, прежде чем двигаться дальше, убедитесь, что изменения привели к желаемому результату, выполнив профилирование приложения еще раз.

Итоги

Эта глава была посвящена поиску проблем производительности в приложении. В ней были рассмотрены многие функции и секреты профилировщика, различные практические приемы поиска причин проблем производительности и дано несколько советов и инструкций. С их помощью можно значительно улучшить производитель-

ность, если подходить к их применению взвешенно и помнить, что использовать их нужно тогда, когда ситуация это позволяет.

Здесь были представлены советы, методики и инструкции для поиска узких мест, которые требуют улучшения. В остальных главах будут рассмотрены методы исправления проблем, позволяющие повысить производительность настолько, насколько это возможно. А теперь поаплодируйте себе, поздравив с успешным прохождением начальной и достаточно нудной части, и переходите к изучению приемов разработки сценариев на языке C#.

Глава 2

Приемы разработки сценариев

Разработка сценариев занимает довольно много времени, поэтому стоит познакомиться с некоторыми передовыми премами их создания. Сценарии – очень объемная тема, поэтому в этой главе будут рассмотрены только примеры, характерные для Unity, и все внимание будет сосредоточено на задачах, связанных с прикладным программным интерфейсом и движком Unity. Особенности и приемы программирования на языке C# с применением библиотек .NET и Mono Framework будут рассматриваться в *главе 7 «Мастерство управления памятью»*.

Эта глава поможет решить накопившиеся конкретные проблемы и познакомит с рядом методик, которые вы сразу сможете использовать для улучшения сценариев и продолжить применять их в дальнейшем. В каждом случае будет выявлена причина проблемы, приведен пример ситуации, когда такая проблема проявляется, и предложено одно или несколько решений.

Кэширование ссылок на компоненты

Распространенной ошибкой в сценариях Unity является чрезмерное использование метода `GetComponent()`. Для примера рассмотрим следующий фрагмент, проверяющий состояние здоровья (значение `health`) персонажа и выполняющий отключение ряда компонентов, если здоровье падает ниже нулевой отметки (`0`), чтобы подготовиться к анимации смерти:

```
void TakeDamage() {
  if (GetComponent<HealthComponent>().health < 0) {
    GetComponent<Rigidbody>().enabled = false;
```

```
    GetComponent<Collider>().enabled = false;
    GetComponent<AIControllerComponent>().enabled = false;
    GetComponent<Animator>().SetTrigger("death");
  }
}
```

При каждом вызове этот метод будет запрашивать ссылки на пять разных компонентов `Component`. Это хорошо с точки зрения потребления памяти (в том смысле, что она вообще не будет потребляться), но это не очень дружелюбно по отношению к центральному процессору. И выглядит особенно проблематичным, если этот метод вызывается из метода `Update()`. Даже если это не так, его вызов все равно может совпасть по времени с другими важными событиями, такими как создание эффектов частиц, подмена объекта объектом ragdoll (при этом включается в работу физический движок) и т. д. Такой стиль программирования может на первый взгляд показаться безобидным, но в перспективе способен вызвать множество проблем и выполнение массы бесполезных операций.

Кэширование этих ссылок для дальнейшего использования потребует очень небольшого объема (32 или 64 бита на каждую ссылку в зависимости от версии Unity, платформы и разрешения фрагментации). То есть если память не является чрезвычайно ограниченным ресурсом, имеет смысл получить все ссылки во время инициализации и хранить их, пока они необходимы:

```
private HealthComponent _healthComponent;
private Rigidbody _rigidbody;
private Collider _collider;
private AIControllerComponent _aiController;
private Animator _animator;

void Awake() {
  _healthComponent = GetComponent<HealthComponent>();
  _rigidbody = GetComponent<Rigidbody>();
  _collider = GetComponent<Collider>();
  _aiController = GetComponent<AIControllerComponent>();
  _animator = GetComponent<Animator>();
}

void TakeDamage() {
  if (_healthComponent.health < 0) {
    _rigidbody.detectCollisions = false;
    _collider.enabled = false;
    _aiController.enabled = false;
```

```
    _animator.SetTrigger("death");
  }
}
```

Кэширование ссылок на компоненты предотвращает их многократное извлечение, сохраняя при этом ресурсы центрального процессора за счет небольшого дополнительного использования памяти.

Самый быстрый метод получения ссылок на компоненты

Существует несколько вариантов метода GetComponent(), поэтому благоразумнее выбрать наиболее быстрый из них. Имеются три перегруженные версии метода: GetComponent(string), GetComponent<T>() и GetComponent(typeof(T)). Однако скорость их работы зависит от используемой версии Unity.

В Unity 4 быстрее всех работает метод GetComponent(typeof(T)). Докажем это с помощью простого теста:

```
int numTests = 1000000;
TestComponent test;

using (new CustomTimer("GetComponent(string)", numTests)) {
  for (var i = 0; i < numTests; ++i) {
    test = (TestComponent)GetComponent("TestComponent");
  }
}

using (new CustomTimer("GetComponent<ComponentName>", numTests)) {
  for (var i = 0; i < numTests; ++i) {
    test = GetComponent<TestComponent>();
  }
}

using (new CustomTimer("GetComponent(typeof(ComponentName))", numTests))  {
  for (var i = 0; i < numTests; ++i) {
    test = (TestComponent)GetComponent(typeof(TestComponent));
  }
}
```

Этот тест вызывает каждую версию метода GetComponent() миллион раз. Это гораздо больше, чем количество вызовов в типичном проекте, и такого числа повторов явно достаточно для доказательства.

Результаты тестирования показаны на рис. 2.1.

```
GetComponent(string) finished: 841.00ms total, 0.000841ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
GetComponent<ComponentName> finished: 169.00ms total, 0.000169ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
GetComponent(typeof(ComponentName)) finished: 122.00ms total, 0.000122ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
```

Рис. 2.1 ❖ Результаты тестирования версий метода GetComponent()

Как видите, версия GetComponent(typeof(T)) заметно быстрее, чем GetComponent<T>(), которая примерно в пять раз быстрее, чем GetComponent(string). Эти результаты получены в версии Unity 4.5.5, и они должны выглядеть похоже на более ранних версиях Unity 3.x.

Версия GetComponent(string) не должна использоваться из-за своей медлительности и сохранена только из соображений полноты.

Результаты тестирования в Unity 5 показаны на рис. 2.2. Разработчики из Unity Technologies немного ускорили передачу ссылок вида System.Type в версии Unity 5.0, и, как следствие, производительность методов GetComponent<T>() и GetComponent(typeof(T)) сравнялась.

```
GetComponent(string) finished: 2961.00ms total, 0.002961ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
GetComponent<ComponentName> finished: 113.00ms total, 0.000113ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
GetComponent(typeof(ComponentName)) finished: 114.00ms total, 0.000114ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
```

Рис. 2.2 ❖ Результаты тестирования версий метода GetComponent() в Unity 5

Как видите, версия GetComponent<T>() лишь незначительно быстрее, чем GetComponent(typeof(T)), в то время как GetComponent(string) оказался почти в 30 раз медленнее своих альтернатив (интересно, что он стал даже медленнее, чем в Unity 4). Многократное повторение тестов, вероятно, приведет к некоторым колебаниям в результатах, но в любом случае в версии Unity 5 можно использовать любую версию метода GetComponent(), основанную на типах, и результат будет примерно одинаковым.

Однако есть один нюанс. В Unity 4 можно пользоваться различными свойствами быстрого доступа, такими как collider, rigidbody, camera и т. д. Эти свойства подобны предварительно кэшированным ссылкам на компоненты и действуют значительно быстрее, чем любые традиционные методы GetComponent():

```
int numTests = 1000000;
Rigidbody test;

using (new CustomTimer("Cached reference", numTests))
{
  for (var i = 0; i < numTests; ++i) {
    test = gameObject.rigidbody;
  }
}
```

Следует отметить, что этот код предназначен для Unity 4 и выдаст ошибку при компиляции в Unity 5 из-за отсутствия в этой версии свойства rigidbody.

Результаты выполнения этого теста в Unity 4 показаны на рис. 2.3.

```
Cached reference finished: 102.00ms total, 0.000102ms per test for 1000000 tests
UnityEngine.Debug:Log(Object)
```

Рис. 2.3 ❖ Результаты тестирования свойства rigidbody в Unity 4

Для уменьшения числа зависимостей и улучшения модульности кода в серверной части движка команда Unity Technologies приняла решение отказаться от поддержки этих переменных в Unity 5. Сохранено только свойство transform.

Пользователи Unity 4, планирующие переход на Unity 5, должны знать, что при обновлении обращения к этим свойствам будут автоматически заменены вызовами метода GetComponent<T>(). Это приведет к появлению некэшированных вызовов метода GetComponent<T>(), разбросанных по всему коду, поэтому перед переходом нужно внести изменения, подобные тем, что были описаны в разделе «Кэширование ссылок на компоненты» выше.

Мораль этой басни в том, что если разработка ведется в Unity 4 и требуется получить один из компонентов, доступный через встроенное свойство игрового объекта, используйте это свойство. Иначе предпочтение нужно отдать методу GetComponent(typeof(T)). При этом в Unity 5 можно пользоваться любой версией, основанной на типах, то есть GetComponent<T>() или GetComponent(typeof(T)).

Удаление пустых объявлений обратных вызовов

При создании нового файла сценария с реализацией `MonoBehaviour` в Unity 4 или 5 редактор Unity автоматически создаст заготовки двух методов:

```
// Используется для инициализации
void Start () {

}

// Вызывается один раз в каждом кадре
void Update () {

}
```

Движок Unity определяет присутствие этих методов на этапе инициализации и добавляет их в список методов обратного вызова. Но если оставить пустые объявления в коде, это приведет к появлению небольших накладных расходов на их вызов движком.

Метод `Start()` вызывается только один раз, в момент создания игрового объекта, когда производится загрузка сцены или создается новый игровой объект. Поэтому наличие пустых объявлений методов `Start()` может никак не отразиться на производительности сцены, если в ней не так много игровых объектов. Однако этот метод добавляет ненужную нагрузку при любом вызове метода `GameObject.Instantiate()`, что обычно происходит во время ключевых событий, и может усугубить и без того сложную ситуацию, когда одновременно происходит много событий.

С другой стороны, метод `Update()` вызывается вский раз, когда сцена перерисовывается. Если сцена содержит тысячи игровых объектов с компонентами, содержащими такие пустые методы `Update()`, на их обработку будет потрачено множество тактов процессорного времени, что отрицательно скажется на частоте кадров.

Проверим это утверждение с помощью простого теста. Создадим тестовую сцену с игровым объектом, имеющим компоненты двух видов – содержащие объявления пустых методов `Update()` и не содержащие:

```
public class CallbackTestComponent : MonoBehaviour {
  void Update () {}
}

public class EmptyTestComponent : MonoBehaviour {
}
```

Ниже представлены результаты тестирования 32 768 компонентов каждого вида. Если включить все объекты, в которых отсутствуют пустые методы, профилировщик не покажет какого-либо увеличения потребления процессора. Вы сможете заметить некоторые изменения в потреблении памяти и прочую активность VSync (Вертикальная синхронизация), но в целом картина останется в пределах нормы. Однако как только будут включены все объекты с объявлениями пустых методов обратного вызова, потребление процессора резко возрастет, как показано на рис. 2.4.

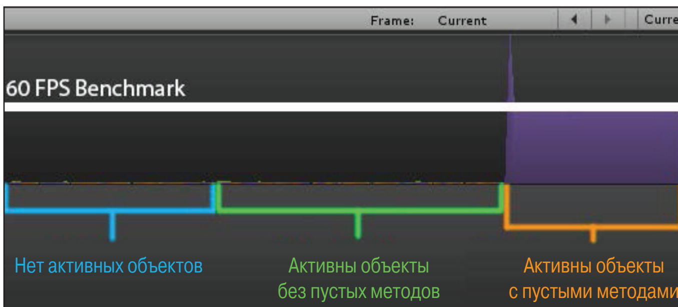


Рис. 2.4 ❖ Влияние пустых методов на производительность

Чтобы исправить эту проблему, достаточно удалить объявления пустых методов. Движок Unity не обнаружит их и не будет вызывать. Иногда *поиск* объявлений пустых методов в большом проекте является довольно сложной задачей, и решить ее вам помогут простые регулярные выражения.

Все популярные инструменты редактирования кода, такие как MonoDevelop, Visual Studio и даже Notepad ++, поддерживают поиск в проекте с применением регулярных выражений. За более подробной информацией обращайтесь к документации для своих инструментов редактирования, поскольку реализация такого поиска может отличаться в разных инструментах и версиях.

Следующее регулярное выражение позволит найти все объявления пустых методов `Update()` в коде:

```
void\s*Update\s*?\(\s*?\)\s*?\n*?\{\n*?\s*?\}
```

С помощью этого регулярного выражения можно найти все стандартные объявления метода `Update()`, включающие избыточные

пробелы и символы новой строки, разбросанные по всему объявлению метода.

Естественно, оно подойдет и для нешаблонных обратных вызовов Unity, таких как `OnGUI()`, `OnEnable()`, `OnDestroy()`, `FixedUpdate()` и т. д. Полный список таких обратных вызовов приводится в документации к Unity с описанием класса `MonoBehaviour` (http://docs.unity3d.com/ScriptReference/MonoBehaviour.html).

Может показаться маловероятным, что кто-то создаст объявления пустых методов в коде, но никогда не говори *никогда*. Например, если во всех компонентах используется общий базовый класс `MonoBehaviour`, одно объявление пустого метода обратного вызова в этом базовом классе будет пронизывать всю игру, что приведет к очень большим затратам. С особой осторожностью следует относиться к методу `OnGUI()`, так как он может несколько раз вызываться в течение одного кадра или события пользовательского интерфейса.

Не используйте методов Find() и SendMessage()

Метод `SendMessage()` и методы семейства `GameObject.Find()` являются крайне дорогостоящими, и их следует избегать при любой возможности. Метод `SendMessage()` выполняется примерно в 2000 раз медленнее вызова простой функции, а медлительность метода `Find()` особенно ярко проявляется в сложных сценах, поскольку он последовательно перебирает все игровые объекты в сцене. Метод `Find()` допускается вызывать во время инициализации сцены, например из методов `Awake()` и `Start()`, для поиска объектов, которые уже существуют в сцене. Однако использование любого из этих методов для организации взаимодействий объектов во время выполнения приведет к очень заметным накладным расходам.

Использование методов `Find()` и `SendMessage()` обычно свидетельствует о плохо продуманной архитектуре, неопытности в программировании на C# и для Unity или просто о проявлении лени при создании прототипов. Их использование стало просто эпидемией в проектах начального и среднего уровня, из-за чего команда Unity Technologies вынуждена постоянно напоминать пользователям о необходимости избегать их в реальных играх, как в документации, так и на конференциях. Их существование оправдано только тем, что они позволяют *новичкам* начать использовать связи между объектами,

а также полезностью в некоторых особых случаях, когда их медлительность может быть оправдана.

Справедливости ради следует отметить, что круг пользователей Unity чрезвычайно неоднороден, он охватывает одиночек-любителей, студентов, страдающих манией величия, малые, средние и большие команды разработчиков. Это приводит к необходимости поддержки невероятно широкого спектра возможностей разработки программного обеспечения. Новичкам в Unity очень трудно понять, почему, собственно, они *должны* делать что-то по-разному, если учесть, что движок Unity не придерживается парадигмы разработки, характерной для многих других игровых движков. В нем имеются привнесенные и необычные концепции, связанные со сценой и шаблонными объектами (prefabs), нет указателей на *вездесущий класс*, и отсутствуют очевидные системы хранения необработанных данных для работы с ним.

Поскольку в этом разделе речь идет об оптимизации сценариев, перейдем к детальному обсуждению вопроса, рассмотрев альтернативные методы организации взаимодействий между объектами.

Начнем с анализа наихудшего случая с использованием методов `Find()` и `SendMessage()` и обсудим способы его оптимизации. Метод в следующем примере создает заданное количество экземпляров врагов из заготовки объекта, а затем уведомляет объект `EnemyManager` об их существовании:

```
public void SpawnEnemies(int numEnemies) {
  for(int i = 0; i < numEnemies; ++i) {
    GameObject enemy =
(GameObject)GameObject.Instantiate(_enemyPrefab, Vector3.zero, Quaternion.
identity);
    GameObject enemyManagerObj = GameObject.Find("EnemyManager");
    enemyManagerObj.SendMessage("AddEnemy", enemy,
    SendMessageOptions.DontRequireReceiver);
  }
}
```

Вызов метода внутри цикла, всегда возвращающий один и тот же результат, должен восприниматься как большой красный флаг, свидетельствующий о низкой производительности, а если это один из самых дорогостоящих методов, таких как `Find()`, нужно найти способ вызывать такой метод настолько редко, насколько это только возможно. Следовательно, лучшим решением являются вынос вызова метода `Find()` за пределы цикла `for` и кэширование результата в локальной переменной.

Дополнительно можно оптимизировать использование метода SendMessage(), заменив его вызовом GetComponent(). Такая замена весьма затратного метода на менее затратный вариант положительно отразится на эффективности.

После оптимизации получаем следующий код:

```
public void SpawnEnemies(int numEnemies) {

  GameObject enemyManagerObj = GameObject.Find("EnemyManager");
  EnemyManagerComponent enemyMgr =
  enemyManagerObj.GetComponent<EnemyManagerComponent>();

  for(int i = 0; i < numEnemies; ++i) {
    GameObject enemyIcon =
(GameObject)GameObject.Instantiate(_enemyPrefab, Vector3.zero,
Quaternion.identity);
    enemyMgr.AddEnemy(enemy);
  }
}
```

Если этот метод вызывается во время инициализации сцены и время ее загрузки не слишком велико, работу по оптимизации можно считать завершенной.

Однако во время выполнения часто требуется создавать новые объекты для поиска существующего объекта и взаимодействия с ним. В этом примере требуется зарегистрировать новые вражеские объекты в компоненте EnemyManagerComponent, чтобы он имел возможность контролировать управление ими в сцене. Нужен надежный и быстрый способ поиска новыми объектами, существующими без использования метода Find().

Существует несколько решений этой задачи, каждое из которых обладает своими преимуществами и недостатками:

- статические классы;
- компоненты-одиночки;
- сохранение ссылок на существующие объекты;
- глобальная система обмена сообщениями.

Статические классы

Это решение основано на создании класса, который всегда доступен из любого места в коде. Объект этого класса существует от момента запуска приложения до его завершения. Глобальные управляющие классы часто критикуют за то, что по их именам невозможно понять, что они должны делать, и они затрудняют отладку, поскольку могут

измениться в любой момент из любой точки в коде. Кроме того, это наименее надежное решение с точки зрения дальнейшего развития проекта. Несмотря на все недостатки, это решение на сегодняшний день является наиболее простым в реализации, и поэтому начнем с него.

Шаблон проектирования «Одиночка» широко используется для создания глобально доступного объекта, который может существовать только в единственном экземпляре. Но *подавляющее* большинство объектов-одиночек (учитывая особенности их использования) в Unity-проектах легко можно заменить более простыми статическими классами C#, не требующими реализации закрытых конструкторов и ненужных свойств доступа к переменным экземпляра. По сути, типичная реализация шаблона проектирования «Одиночка» в C# занимает больше времени и дает тот же результат, который можно получить с помощью статического класса.

Статический класс, который функционирует во многом так же, как `EnemyManagerComponent` в предыдущем примере, можно определить следующим образом:

```
using System.Collections.Generic;

public static class EnemyManager {
  static List<GameObject> _enemies;

  public static void AddEnemy(GameObject enemy) {
    _enemies.Add (enemy);
  }

  public static void RollCall() {
    for(int i = 0; i < _enemies.Count; ++i) {
      Debug.Log (string.Format("Enemy \"{0}\" reporting in...", _ enemies[i].
name));
    }
  }
}
```

Обратите внимание, что при объявлении каждого свойства и метода применяется ключевое слово `static`. Это подразумевает наличие в памяти только одного экземпляра объекта. Статические классы, по определению, не поддерживают нестатических свойств экземпляра, поскольку это будет означать, что объект каким-то образом можно дублировать.

Статические классы могут иметь статический конструктор для инициализации значений свойств, который можно определить, как

показано ниже, и который будет вызываться в момент первого обращения к классу (к его свойству или методу):

```
static EnemyManager() {
  _enemies = new List<GameObject>();
}
```

Такой тип глобального класса, как правило, реализуется в C# проще и нагляднее, чем обычный шаблон проектирования «Одиночка».

Компоненты-одиночки

Недостаток решения на основе статического класса заключается в необходимости наследования самого низкоуровневого класса – `Object`. Это означает, что статические классы не могут наследовать класс `MonoBehaviour` и, соответственно, использовать его инструменты, имеющие отношение к Unity, включая обработку событий и сопрограммы. Кроме того, из-за отсутствия объекта для выбора теряется возможность увидеть данные в инспекторе во время выполнения. Однако эти инструменты доступны в глобальных классах-одиночках.

Типичное решение этой проблемы заключается в реализации класса *«компонента-одиночки»*, порождающего игровой объект и предоставляющего статические методы для глобального доступа. Обратите внимание, что в этом случае фактически необходимо реализовать типичный шаблон проектирования «Одиночка», с локальными переменными статического экземпляра и глобальным методом `Instance` для глобального доступа.

Ниже приведено определение класса `SingletonAsComponent`:

```
public class SingletonAsComponent<T> : MonoBehaviour where T :
SingletonAsComponent<T> {

  private static T Instance;

  protected static SingletonAsComponent<T> _Instance {
    get {
      if(!  Instance) {
        T [] managers =
        GameObject.FindObjectsOfType(typeof(T)) as T[];
        if (managers != null) {
            if(managers.Length == 1) {
                __Instance = managers[0];
                return Instance;
            } else if (managers.Length > 1) {
                Debug.LogError("You have more than one " +
```

```
                typeof(T).Name + " in the scene. You only
                need 1, it's a singleton!");
                for(int i = 0; i < managers.Length; ++i) {
                    T manager = managers[i];
                    Destroy(manager.gameObject);
                }
            }
        }

        GameObject go = new GameObject(typeof(T).Name,
        typeof(T));
        __Instance = go.GetComponent<T>();
        DontDestroyOnLoad( Instance.gameObject);
      }
      return Instance;
    }
    set {
      __Instance = value as T;
    }
  }
}
```

Поскольку объект должен быть глобальным и постоянным, сразу после создания игрового объекта требуется вызвать функцию DontDestroyOnLoad(). Эта специальная функция сообщает Unity, что объект должен сохраняться при смене сцен, вплоть до завершения выполнения приложения. При загрузке новой сцены объект не будет уничтожен и сохранит все свои данные.

Это определение класса предполагает два момента. Во-первых, поскольку в определении поведения используются обобщенные шаблоны, для создания конкретного класса требуется унаследовать конкретные типы. Во-вторых, должен быть определен метод, присваивающий значение переменной _Instance и осуществляющий необходимое приведение типов.

Например, ниже показано, что требуется для успешного создания нового класса MySingletonComponent, наследующего класс SingletonAsComponent:

```
public class MySingletonComponent : SingletonAsComponent<MySingletonCompone
nt> {
  public static MySingletonComponent Instance {
    get { return ((MySingletonComponent)_Instance); }
    set { _Instance = value; }
  }
}
```

Этот класс может использоваться во время выполнения для предоставления постоянного доступа к свойству `Instance` любым другим объектам. Если компонент отсутствует в сцене, базовый класс `SingletonAsComponent` создаст экземпляр своего собственного игрового объекта и подключит к нему экземпляр порожденного класса в качестве компонента. С этого момента свойство `Instance` будет ссылаться на созданный компонент.

Хотя это возможно, не нужно помещать класс, наследующий класс `SingletonAsComponent`, в иерархию сцены, потому что метод `DontDestroyOnLoad()` никогда не будет вызван! Из-за этого игровой объект, к которому подключен компонент-одиночка, не сохранится при загрузке следующей сцены.

Правильная очистка компонента-одиночки может вызывать сложности из-за особенностей уничтожения сцен в Unity. Метод `OnDestroy()` объекта вызывается во время выполнения, когда этот объект должен быть уничтожен. Тот же метод вызывается во время завершения работы приложения, причем каждый компонент, присоединенный к каждому игровому объекту, имеет свой метод `OnDestroy()`, вызываемый Unity. Кроме того, приложение завершается при выходе из режима воспроизведения и возврата в режим редактирования. При этом уничтожение объектов происходит в случайном порядке и не гарантируется, что компонент-одиночка будет уничтожен в последнюю очередь.

Соответственно, если какой-то объект попытается обратиться к компоненту-одиночке в своем методе `OnDestroy()`, произойдет обращение к свойству `Instance`. Если компонент-одиночка на этот момент уже уничтожен, такое обращение приведет к созданию нового экземпляра компонента-одиночки во время завершения работы приложения! Это может повредить файлы сцены, так как экземпляр компонента-одиночки не будет разрушен при уничтожении сцены. Если это произойдет, Unity выведет сообщение об ошибке, как показано на рис. 2.5.

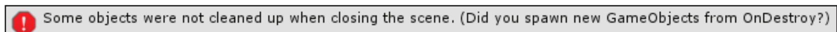


Рис. 2.5 ❖ Сообщение об ошибке в случае создания нового экземпляра компонента-одиночки

Возможность обращения некоторых объектов к компоненту-одиночке при уничтожении объясняется тем, что такие компоненты часто реализуют шаблон «Наблюдатель», позволяя другим объектам

подписываться/отписываться с его помощью на определенные задания, подобно тому как в Unity применяются методы обратного вызова, но в менее автоматизированной манере. Пример такого подхода будет представлен в следующем разделе «*Глобальная система обмена сообщениями*». Объекты, зарегистрировавшиеся при создании, должны отменить регистрацию перед уничтожением, а наиболее удобным местом для этого является метод `OnDestroy()`. Следовательно, такие объекты почти наверняка столкнутся с вышеупомянутой проблемой создания компонентов-одиночек по ошибке во время завершения работы приложения.

Для решения этой проблемы нужно внести три изменения. Во-первых, добавить дополнительный флаг в компонент-одиночку для проверки его состояния и выключения в соответствующие моменты, к которым относятся момент самоуничтожения компонента, а также завершение работы приложения (`OnApplicationQuit()` – еще один полезный метод обратного вызова для наследников `MonoBehaviour`, который будет вызван в этот момент):

```
private bool _alive = true;
void OnDestroy() { _alive = false; }
void OnApplicationQuit() { _alive = false; }
```

Во-вторых, необходимо реализовать проверку текущего состояния компонента для внешних объектов:

```
public static bool IsAlive {
  get {
    if ( Instance == null)
      return false;
    return Instance._alive;
  }
}
```

И наконец, любой объект, который пытается обратиться к компоненту-одиночке из метода `OnDestroy()`, должен перед вызовом `Instance` проверить состояние компонента с помощью свойства `IsAlive`. Например:

```
public class SomeComponent : MonoBehaviour {
    void OnDestroy() {
    if (MySingletonComponent.IsAlive) {
        MySingletonComponent.Instance.SomeMethod();
     }
  }
}
```

Это гарантирует, что никто не попытается получить доступ к `Instance` во время уничтожения. Если не следовать этому правилу, всегда будут возникать проблемы с экземплярами компонентов-одиночек, остающимися в сцене после возврата в режим редактирования.

Паракдоксальность решения на основе компонентов-одиночек заключается в использовании одного из методов `Find()` для проверки существования компонента-одиночки в сцене перед присваиванием ссылки `__Instance`. К счастью, это происходит только при первом обращении к компоненту-одиночке, но, с другой стороны, инициализация компонента не обязательно должна происходить одновременно с инициализацией сцены и может привести к всплеску потери производительности в неподходящий момент. Чтобы избежать этого, необходимо предусмотреть некий вездесущий класс, гарантирующий создание экземпляров важных компонентов-одиночек при инициализации сцены простым обращением к свойству `Instance` каждого из них.

Однако если в дальнейшем будет решено использовать сразу несколько таких управляющих классов или разделить их поведение для удобства, придется изменить *массу* кода.

Имеются также другие альтернативы, например использование встроенного моста Unity между кодом сценария и интерфейсом инспектора.

Сохранение ссылок на существующие объекты

Другое решение проблемы организации связи между объектами заключается в использовании встроенных систем сериализации Unity. Пуристы в области разработки программного обеспечения воинственно настроены в отношении этой функции, поскольку она противоречит принципу инкапсуляции, позволяя открывать доступ к закрытым переменным. Даже при том, что значения переменных становятся доступными только инспектору Unity и никому другому, этого оказалось достаточно, чтобы ломать копья.

Тем не менее это очень эффективный инструмент совершенствования процесса разработки. Он особенно полезен в ситуации, когда художники, дизайнеры и программисты трудятся над созданием одного и того же продукта, причем их уровни владения компьютером и программированием сильно отличаются. Иногда стоит незначительно нарушить правила для достижения более высокой производительности.

Когда мы создаем общедоступную переменную, Unity автоматически сериализует ее значение и выводит в интерфейсе инспектора

при выборе компонента. Однако общедоступные переменные несут потенциальную угрозу, связанную с возможностью их изменения из произвольного фрагмента кода, что затрудняет отслеживание значения переменной и может привести к возникновению неожиданных ошибок.

Как вариант можно выбрать любую закрытую или защищенную переменную-член класса и экспортировать ее в интерфейс инспектора редактора Unity, просто добавив атрибут `[SerializeField]`. Этот подход предпочтительнее использования общедоступных переменных, потому что делает ситуацию более управляемой. Используя его, мы будем уверены, что значения переменных не могут изменяться из внешнего кода (или классами-наследниками) и инкапсуляция с точки зрения кода сценария будет сохранена.

Например, следующий класс экспортирует в инспектор три закрытые переменные:

```
public class EnemySpawnerComponent : MonoBehaviour {

  [SerializeField] private int _numEnemies;
  [SerializeField] private GameObject _enemyPrefab;
  [SerializeField] private EnemyManagerComponent _enemyManager;

  void Start() {
    SpawnEnemies(_numEnemies);
  }

    void SpawnEnemies(int _numEnemies) {
      for(int i = 0; i < _numEnemies; ++i) {
        GameObject enemy = (GameObject)GameObject.Instantiate
                            (_enemyPrefab, Vector3.zero,
                            Quaternion.identity);
        _enemyManager.AddEnemy(enemy);
      }
    }
}
```

Обратите внимание, что спецификатор области видимости `private` в языке C# можно опустить, потому что свойства всегда получают область видимости `private`, если явно не указано иное. Здесь этот спецификатор использован для большей наглядности.

На рис. 2.6 изображена панель **Inspector** с тремя значениями по умолчанию `0` или `null`, которые можно изменять с помощью интерфейса инспектора:

Рис. 2.6 ❖ Панель инспектора с тремя экспортированными значениями

В поле **Enemy Prefab** можно перетащить мышью ссылку на шаблон объекта из окна проектов или даже ссылку на другой игровой объект, присутствующий в сцене. Хотя, учитывая, что игровой объект будет использоваться подобно шаблону, делать это неразумно, поскольку клонированный игровой объект мог быть уже изменен в сцене. Шаблоны объектов предназначены для создания новых экземпляров игровых объектов, и именно их надо использовать для этой цели.

Поле **Enemy Manager** интересно тем, что хранит ссылку на компонент, а не на игровой объект. Если перетащить в это поле игровой объект, его значением станет ссылка на *компонент* данного объекта, а не на сам игровой объект. Если данный объект не содержит нужного компонента, присваивания не произойдет.

Обычно поля ссылок на компоненты используются для сохранения ссылок на компоненты, присоединенные к игровому объекту. Это альтернативный подход рассмотренному в разделе «Кэширование ссылок на компоненты» выше в этой же главе.

Опасность здесь заключается в том, что шаблонные объекты, по сути являющиеся игровыми объектами и содержащие нужный компонент, *могут* быть присвоены этим полям даже вопреки желанию разработчика. Unity загружает шаблонные объекты в память, подобно игровым объектам, и предполагает, что они будут использоваться как шаблонные объекты, то есть только как калька для создания экземпляров. Однако они по-прежнему считаются данными, хранящимися в памяти, и поэтому могут произвольным образом редактироваться, что делает их восприимчивыми к изменениям, которые непосредственно влияют на все будущие игровые объекты, созданные из них.

Хуже того, эти изменения становятся постоянными, даже если сделаны в режиме воспроизведения игры, поскольку шаблонные объекты занимают то же пространство памяти, независимо от того, включен режим воспроизведения или нет. Это означает, что можно случайно

повредить шаблонные объекты при присваивании их неправильным полям. Соответственно, данный подход больше подходит для командной работы над решением задачи поддержки взаимодействий между объектами, но все же не идеален из-за рисков, связанных с тем, что члены команды могут случайно испортить шаблонные объекты или оставить в ссылках значение `null`.

Важно также отметить, что не все объекты могут быть сериализованы. В Unity сериализовать можно все простые типы данных (`int`, `float`, `string` и `bool`), различные встроенные типы, такие как (`Vector3`, `Quaternion` и т. д.) перечисления, классы, структуры, а также массивы и списки с элементами сериализуемых типов. Однако нельзя сериализовать статические поля, значения, доступные только для чтения, свойства и словари.

Некоторые Unity-разработчики используют прием псевдосериализации словарей с применением двух отдельных списков для ключей и значений, а также пользовательского редактора сценариев, или с помощью одного списка объектов, содержащих ключи и значения. Эти решения хотя и громоздки, но вполне допустимы.

Последнее решение, рассматриваемое ниже, использует все лучшее из предыдущих, объединяя простоту реализации с легкостью расширения и строгостью использования, позволяющую избегать большинства ошибок, которые могут сделать разработчики.

Глобальная система обмена сообщениями

Последнее решение задачи организации взаимодействий между объектами –реализация глобальной системы обмена сообщениями, позволяющей любому объекту посылать сообщения любым объектам, заинтересованным в сообщениях конкретного типа. Объекты могут отправлять или получать сообщения. При этом принимающий объект должен определить типы интересующих его сообщений. Отправитель сообщения посылает сообщения, не заботясь о том, кто ожидает получить их. Это решение прекрасно подходит для сохранения кода модульным и изолированным.

Отправляемые сообщения могут быть разного вида, например включать значения данных, ссылки, инструкции для принимающих объектов и прочее, но все они должны содержать общее, базовое объявление, с помощью которого система обмена сообщениями сможет определить, что это за сообщение и для кого оно предназначено.

Ниже приводится определение простого класса для объекта Message:

```
public class BaseMessage {
  public string name;
  public BaseMessage() { name = this.GetType().Name; }
}
```

Конструктор класса BaseMessage сохраняет тип в локальное строковое свойство, которое в дальнейшем будет использовано для классификации и распределения. Это сохранение имеет большое значение, потому что каждый вызов метода GetType().Name добавляет новую строку в кучу, и их число нужно свести к минимуму, насколько только возможно. Пользовательские сообщения должны наследовать этот класс, чтобы добавить любые дополнительные данные и сохранить при этом возможность отправки через систему обмена сообщениями. Обратите внимание, что операция присваивания имени типа в конструкторе базового класса будет сохранять в свойстве name имя производного класса, а не базового.

Теперь перейдем к классу MessagingSystem и определим его особенности:

- он должен быть доступным глобально;
- любой объект (MonoBehaviour или нет) должен иметь возможность оформить или аннулировать подписку на сообщения определенного типа (то есть реализовать шаблон «Наблюдатель»);
- подписываясь на сообщения, объект должен указать метод, вызовом которого ему будут передаваться сообщения;
- система должна передавать сообщение всем получателям в разумные сроки, но не забрасывать их слишком большим количеством одновременных запросов.

Глобально доступный объект

Первое требование делает систему обмена сообщениями явным кандидатом на реализацию шаблона «Одиночка», поскольку в приложении должен иметься только один ее экземпляр. Однако, прежде чем принимать решение о реализации этого шаблона, следует хорошо подумать, а так ли это на самом деле. Если в дальнейшем понадобится создать несколько экземпляров объекта, мы столкнемся с массой трудностей из-за наличия множества зависимостей, которые постепенно включались в систему в процессе ее развития.

Этот пример предполагает абсолютную уверенность, что нужна одна-единственная система, и на этом будет основана вся дальнейшая разработка.

Подписка на сообщения

Второе и третье требования можно удовлетворить, реализовав общедоступные методы, которые обеспечат регистрацию в системе обмена сообщениями. Если обязать принимающий объект указывать делегата для передачи ему сообщения, принимающие объекты смогут определять разные методы для получения разных сообщений. Создание делегатов с именами, соответствующими именам обрабатываемых ими сообщений, значительно улучшит читаемость кода.

Делегаты – невероятно полезные конструкции в языке C#, позволяющие передавать локальные методы в виде аргументов другим методам, которые обычно используются для обратных вызовов. Более подробно узнать о делегировании можно из руководства по программированию MSDN C# на https://msdn.microsoft.com/ru-ru/library/ ms173171.aspx.

Иногда требуется организовать рассылку широковещательного сообщения, такого как, например, «*Порожден враг*», вынуждающего всех получателей что-то выполнить в ответ. В другие моменты может быть желательно отправить сообщение единственному получателю из группы. Например, сообщение «*Уровень здоровья врага изменился*», предназначенное конкретному объекту, отображающему уровень здоровья определенного врага. Если еще реализовать механизм остановки обработки сообщений, можно сэкономить значительное количество ресурсов центрального процессора, когда имеется много получателей, подписанных на сообщения одного вида.

Для этого делегат, используемый для передачи сообщения получателю, должен возвращать ответ, определяющий необходимость дальнейшей обработки сообщения. Решение об остановке обработки можно принимать на основании логического возвращаемого значения, когда `true` означает, что этот получатель обработал сообщение и дальнейшая его обработка не требуется.

Ниже приводится возможная сигнатура делегата:

```
public delegate bool MessageHandlerDelegate(BaseMessage message);
```

Получатели должны определить такой метод и передать ссылку на него при регистрации в `MessagingSystem`, чтобы система обмена сообщениями могла передавать сообщения.

Обработка сообщений

Последнее требование к системе обмена сообщениями заключается в наличии механизма, предотвращающего одновременную рассылку слишком большого количества сообщений. Это означает, что где-то в процессе нужно предусмотреть вызов метода `Update()` и учитывать время.

Реализовать такой механизм можно с помощью статического класса (как было показано выше), которому потребуется другой вездесущий класс, наследующий `MonoBehaviour` и сообщающий об обновлении сцены. Этот механизм можно реализовать также на основе компонента `SingletonAsComponent`, не требующего вездесущего класса. Единственное различие между этими способами заключается в зависимости системы от управления другими объектами.

Подход на основе `SingletonAsComponent` выглядит предпочтительнее, поскольку на практике редко требуется, чтобы система функционировала независимо, когда на нее завязана значительная доля логики игры. Например, даже если игра приостановлена, нежелательно, чтобы ее логика была приостановлена из-за приостановки системы обмена сообщениями. Нужно, чтобы системы обмена сообщениями продолжала получать и обрабатывать сообщения, чтобы, например, обеспечить взаимодействие компонентов пользовательского интерфейса даже после приостановки игрового процесса.

Реализация системы обмена сообщениями

Определим систему обмена сообщениями как наследника класса `SingletonAsComponent` и реализуем метод регистрации объектов в ней:

```
using System.Collections.Generic;

public class MessagingSystem : SingletonAsComponent<MessagingSystem> {

  public static MessagingSystem Instance {
    get { return ((MessagingSystem)_Instance); }
    set { _Instance = value; }
  }

  private Dictionary<string,List<MessageHandlerDelegate>>
_listenerDict = new Dictionary<string,List<MessageHandlerDelegate>>();

  public bool AttachListener(System.Type type, MessageHandlerDelegate
handler) {
    if (type == null) {
      Debug.Log("MessagingSystem: AttachListener failed due to no message
type specified");
```

```
      return false;
    }

    string msgName = type.Name;

    if (!_listenerDict.ContainsKey(msgName)) {
      _listenerDict.Add(msgName, new List<MessageHandlerDelegate>());
    }

    List<MessageHandlerDelegate> listenerList =
_listenerDict[msgName];
    if (listenerList.Contains(handler)) {
      return false; // получатель уже присутствует в списке
    }
    listenerList.Add(handler);
    return true;
  }
}
```

Переменная `_listenerDict` – это словарь строк, отображаемых в списки `MessageHandlerDelegates`. Этот словарь объединяет делегатов в списки по типам обрабатываемых ими сообщений. То есть по типу сообщения можно быстро получить список всех делегатов объектов, подписавшихся на этот тип сообщений, затем обойти список и послать сообщение каждому получателю.

Метод `AttachListener()` требует передачи двух параметров, типа сообщения в форме `System.Type` и делегата `MessageHandlerDelegate`, который будет вызываться для отправки сообщения.

Очередь и обработка сообщений

Для обработки сообщений система `MessagingSystem` должна поддерживать очередь входящих сообщений, чтобы их можно было обрабатывать в порядке поступления:

```
private Queue<BaseMessage> _messageQueue = new Queue<BaseMessage>();

public bool QueueMessage(BaseMessage msg) {
  if (!_listenerDict.ContainsKey(msg.name)) {
    return false;
  }
  _messageQueue.Enqueue(msg);
  return true;
}
```

Метод предварительно проверяет наличие такого типа сообщения в словаре. Дополнительно, для большей эффективности, определя-

ется наличие получателей, подписавшихся на сообщение этого типа, и только потом сообщение добавляется в очередь для последующей обработки. Роль очереди играет закрытое свойство `_messageQueue`.

Далее добавим определение метода `Update()`. Этот метод будет регулярно вызываться движком Unity. Он будет выполнять последовательный обход сообщений в очереди и для каждого будет определять, не слишком ли много времени прошло с момента начала его обработки:

```
private float maxQueueProcessingTime = 0.16667f;

void Update() {
  float timer = 0.0f;
  while (_messageQueue.Count > 0) {
    if (maxQueueProcessingTime > 0.0f) {
      if (timer > maxQueueProcessingTime)
        return;
    }

    BaseMessage msg = _messageQueue.Dequeue();
    if (!TriggerMessage(msg))
      Debug.Log("Error when processing message: " + msg.name);

    if (maxQueueProcessingTime > 0.0f)
      timer += Time.deltaTime;
  }
}
```

Защита на основе подсчета времени состоит в том, чтобы проверить превышение порогового значения. Это позволяет исключить приостановку игры системой обмена сообщениями, если в нее поступит слишком много сообщений за короткий промежуток времени. По истечении интервала времени, отведенного на обработку сообщений, она будет приостановлена и отложена до следующего кадра.

И наконец, нужно определить метод `TriggerMessage()`, передающий сообщения получателям:

```
public bool TriggerMessage(BaseMessage msg) {
  string msgName = msg.name;
  if (!_listenerDict.ContainsKey(msgName)) {
    Debug.Log("MessagingSystem: Message \"" + msgName + "\" has no
listeners!");
    return false; // для сообщения нет получателей, игнорируем его
  }

  List<MessageHandlerDelegate> listenerList =
_listenerDict[msgName];
```

```
  for(int i = 0; i < listenerList.Count; ++i) {
    if (listenerList[i](msg))
      return true; // сообщение обработано делегатом
  }
  return true;
}
```

Метод `TriggerEvent()` – главная рабочая лошадка системы обмена сообщениями. Он извлекает список получателей сообщений данного типа и дает каждому возможность его обработать. Если один из делегатов вернет значение `true`, обработка текущего сообщения прекращается, и метод завершает работу, что позволит методу `Update()` перейти к обработке следующего сообщения.

Обычно для рассылки сообщений используется метод `QueueEvent()`, но с той же целью можно использовать метод `TriggerEvent()`. Он позволяет отправителям заставить получателей обработать сообщение немедленно, не дожидаясь следующего события `Update()`, действуя в обход механизма регулирования, и потому должен использоваться для отправки только критически важных сообщений, когда ожидание следующего кадра нарушит порядок игрового процесса.

Реализация пользовательского сообщения

Система обмена сообщениями готова, но теперь необходимо реализовать пример, помогающий понять, как ею пользоваться. Начнем с определения класса простого сообщения для передачи данных:

```
public class MyCustomMessage : BaseMessage {
  public readonly int _intValue;
  public readonly float _floatValue;
  public MyCustomMessage(int intVal, float floatVal {
    _intValue = intVal;
    _floatValue = floatVal;
  }
}
```

Объявление свойств сообщений как `readonly` гарантирует невозможность изменения данных после создания объекта. Это защитит содержимое сообщений от изменения при передаче от одного получателя к другому.

Регистрация сообщений

Следующий простой класс регистрируется в системе обмена сообщениями, передавая ей ссылку на метод `HandleMyCustomMessage()`, который должен получать сообщения типа `MyCustomMessage`:

```
public class TestMessageListener : MonoBehaviour {

  void Start() {
    MessagingSystem.Instance.AttachListener( typeof(MyCustomMessage), this.
HandleMyCustomMessage);
  }

  bool HandleMyCustomMessage(BaseMessage msg) {
    MyCustomMessage castMsg = msg as MyCustomMessage;
    Debug.Log (string.Format("Got the message! {0}, {1}", castMsg._intValue,
castMsg._floatValue));
    return true;
  }
}
```

При передаче объекта `MyCustomMessage` (откуда угодно!) этот объект будет извлекать сообщение с помощью своего метода `HandleMyCustomMessage()`. Обработчик может выполнить приведение типа сообщения и обработать его своим уникальным способом. На это же сообщение могут подписаться другие классы, выполняющие другие действия в своем собственном методе (предполагается, что делегаты, обработавшие сообщение ранее, не вернули значения `true`).

Тип сообщения в аргументе `msg` метода `HandleMyCustomMessage()` известен, поскольку он был определен при регистрации в вызове метода `AttachListener()`. Благодаря этому можно быть уверенным в безопасности приведения типов и сэкономить время, отказавшись от проверки ссылки на значение `null`, хотя технически ничто не мешает использовать один и тот же делегат для обработки нескольких типов сообщений! В таких случаях придется реализовать определение типа полученного объекта сообщения и обработать его соответственно. Но лучше определить отдельный метод для каждого типа сообщений, чтобы не смешивать их обработку.

Обратите внимание, что определение метода `HandleMyCustomMessage` соответствует сигнатуре функции `MessageHandlerDelegate` и передается в вызов `AttachListener()`. Именно так мы сообщаем системе, какой метод должен вызываться в случае появления сообщения данного типа, и гарантируем безопасность преобразования типов в делегатах. Если сигнатура функции имеет другое возвращаемое значение или другой список аргументов, она не может служить делегатом для передачи в метод `AttachListener()`, и это должно вызвать ошибку компиляции.

Самое замечательное, что делегату можно дать любое имя. На практике чаще выбираются имена, соответствующие именам обраба-

тываемых сообщений. Тогда любой, кто возьмется читать код, легко поймет, для чего используется метод и какой тип сообщений он обрабатывает.

Отправка сообщений

Наконец, реализуем пример отправки сообщения для тестирования системы! Следующий компонент передает экземпляр класса `MyCustomMessage` через систему обмена сообщениями после нажатия клавиши *пробел*:

```
public class TestMessageSender : MonoBehaviour {

  public void Update() {
    if (Input.GetKeyDown (KeyCode.Space)) {
      MessagingSystem.Instance.QueueMessage( new MyCustomMessage(5,
13.355f));
    }
  }
}
```

Если добавить в сцену объекты `TestMessageSender` и `TestMessage-Listener` и нажать клавишу *пробела*, в консоли должно появиться сообщение, информирующее об успехе тестирования, как показано на рис. 2.7.

Рис. 2.7 ❖ Сообщение об успехе тестирования в консоли

Объект-одиночка `MessagingSystem` будет создан сразу после инициализации сцены, в вызове метода `Start()` объекта `TestMessageListener`, который зарегистрирует делегата `HandleMyCustomMessage`. Для создания объекта-одиночки не требуется никаких дополнительных усилий.

Очистка сообщений

Поскольку сообщения являются объектами, они будут создаваться в куче динамически и уничтожаться вскоре после обработки получателями. Однако, как будет описано в *главе 7 «Мастерство управления памятью»*, это ведет к необходимости уборки мусора из-за увеличения объема занятой памяти в куче с течением времени. Если приложение проработает достаточно долго, сборка мусора будет производиться в случайные моменты времени. Поэтому систему обмена

сообщениями следует использовать экономно и избегать частой рассылки ненужных сообщений, сответствующих каждому обновлению.

Еще более важной задачей является реализация своевременного исключения делегатов, объекты которых должны быть уничтожены или заново воссозданы. Если этого не сделать, в системе обмена сообщениями зависнут ссылки на делегатов, которые будут мешать сборщику мусора освобождать память, занятую уже уничтоженными объектами.

По существу, каждому вызову AttachListener() должен соответствовать парный вызов DetachListener() в момент уничтожения объекта или если принято решение, что он больше не нуждается в получении сообщений.

Следующий метод класса MessagingSystem отключает получателя определенного сообщения:

```
public bool DetachListener(System.Type type, MessageHandlerDelegate handler)
  {
    if (type == null) {
      Debug.Log("MessagingSystem: DetachListener failed due to no message
type specified");
      return false;
    }

    string msgName = type.Name;

    if (!_listenerDict.ContainsKey(type.Name)) {
      return false;
    }

    List<MessageHandlerDelegate> listenerList = _listenerDict[msgName];
    if (!listenerList.Contains (handler)) {
      return false;
    }

    listenerList.Remove(handler);
    return true;
  }
```

Ниже приводится пример использования метода DetachListener(), добавленного в класс TestMessageListener:

```
void OnDestroy() {
  if (MessagingSystem.IsAlive) {
    MessagingSystem.Instance.DetachListener( typeof(MyCustomMessage), this.
HandleMyCustomMessage);
  }
}
```

Обратите внимание на использование свойства `IsAlive`, объявленного в классе `SingletonAsComponent`. Это защитит от упомянутых выше проблем при завершении работы приложения, когда нет гарантии, что объект-одиночка будет уничтожен последним.

Обертывание системы обмена сообщениями

Примите поздравления! Мы завершили разработку полнофункциональной системы обмена сообщениями, позволяющей организовать взаимодействие между объектами! Полезной особенностью данного решения является независимость от `MonoBehaviour`, то есть отправители и получатели сообщений не обязаны наследовать класс `MonoBehaviour` для обмена сообщениями. Их класс просто должен предоставлять тип сообщения и функцию с соответствующей сигнатурой.

Если проверить производительность класса `MessagingSystem`, обнаружится, что он способен обрабатывать сотни, если не тысячи сообщений в течение одного кадра с минимальной нагрузкой на процессор (разумеется, в зависимости от мощности центрального процессора). Использование ресурсов центрального процессора одинаково и при отправке одного сообщения 100 различным получателям, и при отправке 100 сообщений одному получателю.

Даже если отправлять сообщения во время игровых событий или событий пользовательского интерфейса, это не должно привести к чрезмерным затратам. То есть если окажется, что система вызывает проблемы с производительностью, причина, скорее всего, кроется в *обработке* сообщений делегатами, а не в передаче их системой обмена сообщениями.

Существует много возможностей улучшения и расширения системы обмена сообщениями дополнительными функциями, которые могут пригодиться в дальнейшем, например:

- разрешить отправителям определять задержку (в виде интервала времени или количества кадров) перед обработкой и распространением сообщения;
- позволить получателям определять приоритеты, в каком порядке они должны получать сообщения по отношению к другим получателям, подписавшимся на сообщения того же типа, то есть предусмотреть возможность обслуживания получателя вне очереди, когда он регистрируется позже других;
- для большей безопасности реализовать проверку ситуаций, когда получатель добавляется в список рассылки определенного сообщения в момент, когда этот список уже обрабатыва-

ется, – в языке C# это вызовет исключение, потому что список делегатов будет изменен методом `AttachListener()`, в то время как он обрабатывается в цикле метода `TriggerEvent()`.

На данный момент вы имеете достаточно полное представление о системе обмена сообщениями, поэтому решение этих задач я оставляю вам в качестве самостоятельного упражнения, если вы захотите использовать данное решение в своих играх.

А теперь рассмотрим несколько дополнительных методов повышения производительности с помощью сценариев.

Отключение неиспользуемых сценариев и объектов

Сцены бывают очень загруженными, особенно когда создаются большие открытые миры. Чем больше объектов вызывает код в методе `Update()`, тем хуже масштабирование и медленнее работает игра. Однако многие объекты обрабатываются впустую, поскольку находятся вне поля зрения игрока или просто слишком далеко. Отказ от обработки таких объектов невозможен в больших градостроительных симуляторах, где постоянно должно выполняться полное моделирование, но в играх от первого лица или гонках, где игрок перемещается по достаточно большой области, отключение невидимых объектов не окажет заметного влияния на процесс игры.

Отключение невидимых объектов

Иногда необходимо отключать компоненты или игровые объекты, когда они не видны. Unity имеет встроенные механизмы отключения объектов, которые не видны на камере (функция **Frustum Culling** (Отсекающая выбраковка) работает автоматическим во всех версиях), и объектов, скрытых за другими объектами (**Occlusion Culling** (Окклюзивная выбраковка), которые будут рассматриваться в *главе 6 «Динамическая графика»*), но эти механизмы не воздействуют на неотображаемые компоненты, такие как сценарии искусственного интеллекта. Их поведение необходимо контролировать вручную.

Эту проблему легко решить с помощью методов обратного вызова `OnBecameVisible()` и `OnBecameInvisible()` класса `MonoBehaviour`, которые вызываются, когда объект становится видимым или невидимым для всех камер в сцене. Кроме того, в сцене с несколькими камерами (например, в локальной многопользовательской игре) методы вызы-

ваются, только когда объект становится видимым для любой *одной* камеры и невидимым для *всех* камер. Это означает, что методы будут вызываться именно в нужные моменты.

Поскольку видимость определяется системой визуализации, игровой объект должен иметь присоединенный отображаемый объект, например `Mesh` или `SkinnedMesh`. Компоненты, которые должны получать обратные вызовы, касающиеся видимости, следует прикрепить к видимым игровым объектам, а не к их родительским или дочерним объектам, поскольку в противном случае они не будут вызываться.

Обратите внимание, что Unity учитывает также скрытую камеру области просмотра сцены при выполнении обратных вызовов `OnBecameVisible()` и `OnBecameInvisible()`. Если обнаружится, что они не вызываются во время тестирования в режиме воспроизведения, убедитесь, что камера области просмотра направлена в обратную сторону.

Для включения и отключения отдельных компонентов по событиям изменения видимости можно использовать следующие методы:

```
void OnBecameVisible() { enabled = true; }
void OnBecameInvisible() { enabled = false; }
```

А включение или отключение всего игрового объекта, к которому подключен компонент, можно реализовать так:

```
void OnBecameVisible() { gameObject.SetActive(true); }

void OnBecameInvisible() { gameObject.SetActive(false); }
```

Отключение отдаленных объектов

В других ситуациях желательно отключать компоненты или игровые объекты, оказавшиеся достаточно далеко от игрока и потому едва заметны. Хорошими кандидатами для отключения в таких случаях являются блуждающие создания с искусственным интеллектом, которые должны быть видимыми на большом расстоянии, но, находясь там, они ничего не должны делать.

Ниже приводится простая сопрограмма, которая периодически проверяет расстояние до целевого объекта и отключает себя, если он находится слишком далеко:

```
[SerializeField] GameObject _target;
[SerializeField] float _maxDistance;
```

```
[SerializeField] int _coroutineFrequency;

void Start() {
  StartCoroutine(DisableAtADistance());
}

IEnumerator DisableAtADistance() {
  while(true) {
    float distSqrd = (Transform.position - _target.transform.position).
sqrMagnitude;

    if (distSqrd < _maxDistance * _maxDistance) {
      enabled = true;
    } else {
      enabled = false;
    }

    for (int i = 0; i < _coroutineFrequency; ++i) {
      yield return new WaitForEndOfFrame();
    }
  }
}
```

Нужно присвоить в инспекторе полю `_target` объект игрока (или любой другой объект, расстояние до которого будет измеряться), задать максимальное расстояние в поле `_maxDistance` и изменить частоту вызова сопрограммы с помощью свойства `_coroutineFrequency`. routineFreque удалится от объекта в свойстве `_target` дальше чем на расстояние `_maxDistance`, он будет отключен. Он будет снова включен при приближении на расстояние меньше порогового.

Эта реализация имеет одно малозаметное усовершенствование, положительно сказывающееся на производительности, – сравнение выполняется не с расстоянием непосредственно, а с его квадратом. Это плавно подводит нас к следующему совету.

Замена расстояния квадратом расстояния

Можно смело утверждать, что процессоры относительно хорошо справляются с умножением вещественных чисел, но просто ужасно – с вычислением их квадратного корня. Каждый раз, запрашивая вычисление расстояния с помощью свойства `magnitude` или метода `Distance()` объекта класса `Vector3`, происходит (в соответствии с теоремой Пифагора) весьма дорогостоящее вычисление квадратного корня, в сравнении со многими другими видами вычислений векторной математики.

Однако класс Vector3 также предоставляет свойство sqrMagnitude, возвращающее то же расстояние, только в квадрате. Оно позволяет выполнить то же сравнение без применения дорогостоящей операции извлечения квадратного корня, если сравнивать два квадрата расстояния. То есть говоря математическим языком: если величина A меньше, чем величина B, значит, A^2 будет меньше, чем B^2.

Например, рассмотрим следующий код:

```
float distance = (transform.position -
other.transform.position).Distance();
if (distance < targetDistance) {
  // делаем что-то
}
```

Его можно заменить следующим, добившись при этом практически идентичного результата:

```
float distanceSqrd = (transform.position -
other.transform.position).sqrMagnitude;
if (distanceSqrd < targetDistance * targetDistance) {
  // делаем что-то
}
```

Результаты лишь *почти* идентичны из-за ограниченной точности операций с плавающей запятой. Некоторая потеря точности в сравнении с версией, использующей квадратный корень, возникает из-за того, что значение будет регулироваться в области с другой плотностью представимых чисел. Оно может совпасть или оказаться очень близко к представлению более точного числа, или, что более вероятно, совпадет, но с меньшей точностью. В результате эти сравнения не *совсем* равноценны, но в большинстве случаев близки настолько, чтобы разница стала незаметной, а выигрыш в производительности может быть весьма существенным.

Если такая незначительная потеря точности не имеет значения, следует использовать этот способ повышения производительности. Однако если точность важна (например, при выполнении точных расчетов в крупномасштабном галактическом космическом симуляторе), нужно поискать другой способ улучшения производительности.

Обратите внимание, что этот метод можно использовать в *любых* расчетах с привлечением квадратного корня, а не только при сравнении расстояний. Этот простой, часто встречающийся пример позволяет пролить свет на важное свойство sqrMagnitude класса Vector3 –

свойство, которое Unity Technologies умышленно сделало открытым, чтобы позволить нам использовать его подобным образом.

Избегайте извлечения строковых свойств объектов игры

Обычно извлечение строкового свойства из объекта подобно получению ссылки любого другого типа в языке C# и выполняется без дополнительных затрат памяти. Однако по некой тайной причине, скрытой внутри исходного кода Unity, извлечение строкового свойства из игрового объекта приводит к дублированию строки в памяти и росту кучи. Это привлекает внимание сборщика мусора, который, в свою очередь, может вызвать всплеск потребления вычислительных ресурсов и тем самым повлиять на производительность.

Такими двумя свойствами игрового объекта, страдающими от этого странного поведения, являются `tag` и `name`. Получение любого из этих свойств всегда приводит к ненужному росту кучи. Поэтому нежелательно использовать эти свойства во время игрового процесса и использовать только там, где *производительность несущественна*, например в редакторе сценариев. Однако система тегов часто используется для идентификации во время выполнения, и отказ от нее может стать серьезной проблемой для некоторых команд разработчиков.

Например, следующий код вызывает рост кучи памяти в каждой итерации цикла:

```
for (int i = 0; i < listOfObjects.Count; ++i) {
  if (listOfObjects[i].tag == "Player") {
    // делаем что-то с этим объектом
  }
}
```

Часто предпочтительнее идентифицировать объекты по их компонентам, типам классов и другим значениям, которые не являются строками. Но иногда это невозможно, например из-за использования стороннего, унаследованного кода или для преодоления каких-то других проблем. Предположим, что по какой-то причине нужно работать только с системой тегов, но при этом хотелось бы избежать выделения памяти в куче.

К счастью, свойство `tag` наиболее часто используется в операциях сравнения, и игровые объекты поддерживают альтернативный спо-

соб сравнения свойств `tag`, который не вызывает роста кучи, – метод `CompareTag()`.

Выполним простой тест, чтобы доказать, что это простое изменение способно изменить мир:

```
void Update() {
  int numTests = 10000000;
  if (Input.GetKeyDown(KeyCode.Alpha1)) {
    for(int i = 0; i < numTests; ++i) {
      if (gameObject.tag == "Player") {
        // делаем что-то
      }
    }
  }

  if (Input.GetKeyDown(KeyCode.Alpha2)) {
    for(int i = 0; i < numTests; ++i) {
      if (gameObject.CompareTag ("Player")) {
        // делаем что-то
      }
    }
  }
}
```

Выполнить эти тесты можно щелчком на кнопках *1* и *2*, запускающих соответствующие циклы `for`. Результаты приводятся на рис. 2.8.

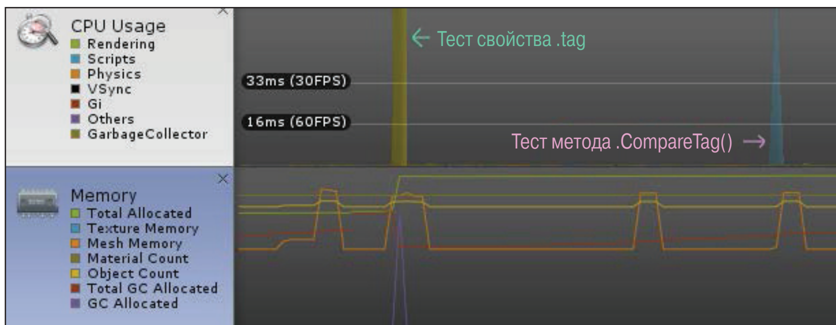


Рис. 2.8 ❖ Временная шкала с результатами тестирования

При просмотре всплесков в области вывода подробной информации (рис. 2.9) можно наблюдать две совершенно разные ситуации.

Overview Тест свойства .tag	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ BehaviourUpdate	99.9%	0.0%	1	362.4 MB	2435.13	0.00
▼ StringAllocationTest.Update()	99.9%	79.9%	1	362.4 MB	2435.12	1946.66
GC.Collect	20.0%	20.0%	1	0 B	488.46	488.46

Overview Тест метода .CompareTag()	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ BehaviourUpdate	99.8%	0.0%	1	0 B	1787.69	0.00
StringAllocationTest.Update()	99.8%	99.8%	1	0 B	1787.69	1787.69
WaitForTargetFPS	0.1%	0.1%	1	0 B	3.11	3.11

Рис. 2.9 ❖ Раздел подробной информации с результатами тестирования

Извлечение свойства tag 10 миллионов раз приводит к выделению около 363 МБ памяти только для одной строки. Это занимает 2435 миллисекунд, из которых 488 миллисекунд тратится на сборку мусора. Тем временем 10 миллионов вызовов метода CompareTag() занимают 1788 миллисекунд и не требуют выделения памяти в куче и поэтому и не создают мусора. Этот пример наглядно демонстрирует, почему следует избегать использования свойств name и tag. То есть если без сравнения тегов не обойтись, следует использовать метод CompareTag().

Обратите внимание, что передача литерала строки «Player» не приводит к росту кучи, поскольку приложение технически выделяет память под это значение при инициализации, а потом просто ссылается на него. Но если строку для сравнения генерировать динамически, возникнет та же проблема роста занятой памяти в куче, поскольку, по сути, каждый раз будет создаваться новый строковый объект.

Более подробно нюансы сборки мусора и использования строк будут рассмотрены в *главе 7 «Мастерство управления памятью»*.

Метод Update, сопрограммы и метод InvokeRepeating

Метод Update вызывается в каждом кадре, но иногда неумелые попытки добиться уменьшения частоты вызовов метода Update приводят к более частому вызову пустого метода:

```
void Update() {
  _timer += Time.deltaTime;
  if (_timer > _aiUpdateFrequency) {
    ProcessAI();
    _timer -= _aiUpdateFrequency;
  }
}
```

Такое определение этой функции приводит к вызову пустой функции практически в каждом кадре. На самом деле все гораздо хуже, потому что выполняется проверка логического значения, которая почти всегда возвращает `false`. Это нормально, если не злоупотреблять таким подходом, но, как было доказано выше, слишком большое количество вызовов ненужных функций в сцене является ударом по производительности.

Это наглядный пример функции, которую нужно преобразовать в сопрограмму, чтобы воспользоваться возможностью задерживать вызовы:

```
void Start() {
  StartCoroutine(UpdateAI());
}

IEnumerator UpdateAI() {
  while (true) {
    yield return new WaitForSeconds(_aiUpdateFrequency);
    ProcessAI();
  }
}
```

Однако этот подход имеет свои недостатки. С одной стороны, сопрограмма вносит дополнительные накладные расходы в сравнении с вызовом обычной функции (медленнее примерно в два раза), а также использует кучу для сохранения своего текущего состояния до следующее вызова. Во-вторых, после инициализации выполнение сопрограммы не зависит от метода `Update()` игрового объекта и будет вызываться независимо от того, активен объект или нет. Прежде чем применять такой подход, следует хорошенько подумать.

Однако в определенных ситуациях преимущества отсутствия вызовов в большинстве кадров перевешивают дополнительные затраты на вызов в каждом кадре. На самом деле, когда в операторах `yield` не выполняется ничего сложного, часто используют более простой вариант этого подхода, основанный на функции `InvokeRepeating()`, которая имеет меньшие накладные расходы (примерно в 1,5 раза медленнее, чем вызов обычной функции):

```
void Start() {
  InvokeRepeating("ProcessAI", 0f, _aiUpdateFrequency);
}
```

Обратите внимание, что функция `InvokeRepeating` также не зависит от метода `Update()` игрового объекта и будет продолжать вызываться, даже когда объект отключен.

Независимо от выбранного подхода существует дополнительный риск вызова слишком большого количества методов в одном и том же кадре. Представьте себе тысячи объектов, которые одновременно инициализируются при инициализации сцены. Каждый раз, по истечении `_aiUpdateFrequency` секунд, они все вместе будут запускать метод `ProcessAI()` в одном и том же кадре, что вызовет огромный всплеск потребления центрального процессора.

Эта проблема имеет следующие возможные решения:

- выполнять задержку на случайный интервал времени при каждом срабатывании таймера или запуске сопрограммы;
- распределить инициализацию сопрограмм по кадрам;
- делегировать ответственность за вызов обновлений некоторому управляющему классу, который ограничит количество вызовов в каждом кадре.

Уменьшив количество излишних определений метода Update с помощью простой сопрограммы, можно ликвидировать массу ненужных накладных расходов, поэтому такое преобразование следует выполнять всегда, когда это возможно. Можно также пересмотреть решение оригинальной проблемы, чтобы предотвратить одновременное повление массы событий.

Другой подход к оптимизации обновлений – отказ от использования метода `Update()`, точнее вызывать его, но только один раз. При вызове метода `Update()` происходит передача управления между низкоуровневой реализацией игрового объекта и его управляемым представлением, что является затратной задачей. Более подробно об этом рассказывается в *главе 7 «Мастерство управления памятью»*, а сейчас просто примем на веру, что каждый обратный вызов в Unity сопровождается затратами на скрытую обработку, которые превышают затрат на вызов обычной функции.

То есть накладные расходы можно уменьшить, ограничив количество передач управления между управляемым и неуправляемым кодами. Сделать это можно с помощью вездесущего класса, вызывающего пользовательские методы обновления во всех пользовательских компонентах. На самом деле многие Unity-разработчики используют этот подход в своих проектах изначально, поскольку он позволяет им полностью контролировать обновления в системе, включая такие моменты, как паузы при выводе всплывающих меню и приостановки при манипуляции эффектами.

Все объекты, подлежащие интегрирации с такой системой, должны иметь общую точку входа. Этого можно достичь с помощью ин-

терфейсного класса. Интерфейсы, по существу, представляют собой соглашение, в соответствии с которым любой класс, реализующий интерфейс, должен определить ряд конкретных методов. Другими словами, если известно, что объект реализует интерфейс, можно быть уверенным в доступности определенных методов. В языке C# классы могут наследовать только один базовый класс, но число реализуемых интерфейсов не ограничено (это позволяет избежать проблемы ромбовидной иерархии – «deadly diamond of death», – знакомой программистам на C++).

В данном случае достаточно реализовать следующий интерфейс, который требует определить единственный метод:

```
public interface IUpdateable {
  void OnUpdate(float dt);
}
```

Далее определим класс `MonoBehaviour`, реализующий этот интерфейс:

```
public class UpdateableMonoBehaviour : MonoBehaviour, IUpdateable
{
  public virtual void OnUpdate(float dt) {}
}
```

Обратите внимание, что метод получил имя `OnUpdate()`, а не `Update()`. Так мы сохранили смысл оригинальной идеи в пользовательской версии и избежали конфликта имен со стандартным методом `Update()`.

Метод `OnUpdate()` в классе `UpdateableMonoBehaviour` извлекает текущее время (`dt`), чтобы избавить нас от массы ненужных вызовов метода `Time.deltaTime`. Кроме того, функция отмечена спецификатором `virtual`, чтобы ее можно было переопределить в производных классах. Как известно, Unity автоматически определяет присутствие методов с именем `Update()` и вызывает их, а так как пользовательские методы обновления имеют другое имя, необходимо реализовать некий вездесущий класс «GameLogic», который будет вызывать эти методы.

При инициализации таких компонентов необходимо сообщать объекту `GameLogic` об их создании и уничтожении, чтобы он знал, когда нужно или не нужно вызывать функцию `OnUpdate()`.

В следующем примере считается, что класс `GameLogic` является компонентом-одиночкой, подобным аналогичному компоненту в разделе «*Компоненты-одиночки*», и имеет статические функции для подписки и отказа от подписки (имейте в виду, что он с легкостью может использовать систему обмена сообщениями!).

Наиболее подходящими для подключения объекта класса MonoBehaviours к системе являются методы Start() и OnDestroy():

```
void Start() {
  GameLogic.Instance.RegisterUpdateableObject(this);
}

void OnDestroy() {
  GameLogic.Instance.DeregisterUpdateableObject(this);
}
```

Выбор метода Start() объясняется просто: к моменту его вызова для всех других компонентов уже будут вызваны их методы Awake(). То есть все критические операции по инициализации объектов уже будут завершены до начала их обновления.

Обратите внимание, что метод Start() определен в базовом классе MonoBehaviour, поэтому, если определить метод Start() в производном классе, он фактически переопределит метод базового класса и Unity будет использовать метод производного класса. Поэтому предпочтительнее реализовать виртуальный метод Initialize(), чтобы производные классы могли переопределить его для настройки инициализации без вмешательства в действия базового класса, такие как уведомление объекта GameLogic о существовании компонента.

Например:

```
void Start() {
  GameLogic.Instance.RegisterUpdateableObject(this);
  Initialize();
}

protected virtual void Initialize() {
  // производные классы должны переопределять этот метод,
  // добавляя в него код инициализации
}
```

Старайтесь автоматизировать этот процесс настолько, насколько это возможно, чтобы избавить себя от необходимости повторно реализовать одни и те же действия в каждом новом компоненте. Если ваш класс наследует класс UpdateableMonoBehaviour, можно быть уверенным, что его метод OnUpdate() обязательно будет вызываться в нужные моменты.

И наконец, реализуем класс GameLogic. Его реализация во многом подобна реализации компонента-одиночки, независимо от использования MessagingSystem в нем. В любом случае, объект класса

UpdateableMonoBehaviour должен оформить подписку или отказаться от нее, как объект IUpdateableObject, а класс GameLogic должен использовать свой метод Update() для обхода всех зарегистрированных объектов и вызова их методов OnUpdate().

Определение класса для системы GameLogic:

```
public class GameLogic : SingletonAsComponent<GameLogic> {
  public static GameLogic Instance {
    get { return ((GameLogic)_Instance); }
  set { _Instance = value; }
  }

List<IUpdateableObject> _updateableObjects = new List<IUpdateableObject>();

public void RegisterUpdateableObject(IUpdateableObject obj) {
  if (!_Instance._updateableObjects.Contains(obj)) {
    _Instance._updateableObjects.Add(obj);
  }
}

public void DeregisterUpdateableObject(IUpdateableObject obj) {
  if (_Instance._updateableObjects.Contains(obj)) {
    _Instance._updateableObjects.Remove(obj);
  }
}

  void Update() {
    float dt = Time.deltaTime;
    for(int i = 0; i < _Instance._updateableObjects.Count; ++i) {
      _Instance._updateableObjects[i].OnUpdate(dt);
    }
  }

}
```

Если все пользовательские классы MonoBehaviours будут наследовать класс UpdateableMonoBehaviour, мы фактически заменим N вызовов функции Update() единственным вызовом метода Update() и N вызовами виртуальных функций. Это предотвратит заметное снижение производительности, поскольку подавляющее большинство операций обновления будет выполняться внутри управляемого кода и передача управления между управляемым и неуправляемым кодами будет осуществляться реже.

В зависимости от стадии разработки текущего проекта внесение таких изменений может стать невероятно сложной и длительной задачей, чреватой появлением массы ошибок, так как для обновления

подсистем потребуется совершенно иной набор зависимостей. Однако преимущества перевесят риски, если имеется запас времени, в чем желательно убедиться, протестировав группы объектов в сцене, обновляемых различными способами.

Кэширование изменений компонента Transform

Компонент `Transform` хранит параметры своего положения только относительно родителей. Это значит, что получение и установка свойств `position`, `rotation` и `scale` компонента `Transform` часто сопровождаются массой матричных расчетов для создания правильного представления свойств компонента относительно свойств его родительских компонентов. Чем глубже находится объект в иерархии, тем больше расчетов понадобится. Хуже того, изменения в компоненте `Transform` приведут к отправке внутренних уведомлений коллайдерам, твердым телам, источникам света и камерам, которые должны быть ими обработаны.

Однако это также значит, что использование свойств `localPosition`, `localRotation` и `localScale` влечет гораздо меньшие затраты, поскольку их значения читаются и записываются в том виде, в каком они передаются. Поэтому везде, где это возможно, следует пользоваться этими локальными свойствами. Однако переход от мировых координат к локальным может усложнить изначально простые (и уже решенные!) задачи, поломать реализацию и внести массу непредвиденных ошибок. Поэтому иногда стоит пожертвовать незначительным снижением производительности в пользу упрощения сложных 3D-математических расчетов!

Кроме того, в ходе обработки сложных событий не редко свойства компонента `Transform` изменяются несколько раз в течение кадра (хотя часто это является признаком излишнего усложнения проекта). Свести к минимуму количество таких изменений можно с помощью кэширования в свойствах класса, с присваиванием окончательных значений по завершении кадра:

```
private bool _positionChanged;
private Vector3 _newPosition;

public void SetPosition(Vector3 position) {
  _newPosition = position;
  _positionChanged = true;
}
```

```
void FixedUpdate() {
  if (_positionChanged) {
    transform.position = _newPosition;
    _positionChanged = false;
  }
}
```

Этот код изменит свойство `position` только в вызове метода `FixedUpdate()` в конце вычислений.

Обратите внимание, что это *не* приведет к видимой инертности игрового процесса, поскольку все физические расчеты выполняются непосредственно после вызова метода `FixedUpdate()`. Кадры не выводятся, пока физический движок не получит возможности отреагировать на изменение свойств компонента `Transform`.

Ускорение проверки отсутствия ссылки на игровой объект

Оказывается, что сравнение ссылки на объект Unity со значением `null` вызывает передачу управления между управляемым и неуправляемым кодом (как упоминалось выше и будет подробно рассматриваться в *главе 7 «Мастерство управления памятью»*), что ожидаемо привносит нежелательные затраты:

```
if (gameObject != null) {
  // делаем что-то с объектом игры
}
```

Существует простая альтернатива, позволяющая выполнить ту же проверку, но почти в два раза быстрее (правда, при этом слегка затеняется смысл кода):

```
if (!System.Object.ReferenceEquals(gameObject, null)) {
  // делаем что-то с объектом игры
}
```

Это относится и к игровым объектам, и к компонентам, и к другим объектам Unity, имеющим управляемое и неуправляемое представления. Однако элементарное тестирование показывает, что оба способа выполняются лишь несколько наносекунд на процессоре Intel Core i5 3570K. Поэтому даже при огромном количестве сравнений ссылок со значением `null` выгода будет едва заметной.

Тем не менее этот пример демонстрирует простые альтернативные способы, помогающие повысить производительность за счет исключения передачи управления между управляемым и неуправляемым кодами.

Итоги

В этой главе было описано множество приемов оптимизации операций с движком Unity, которые следует применять, если (и только если) доказано, что именно эти операции являются причиной проблем с производительностью. Использование некоторых методов требует определенной взвешенности и предварительного профилирования, поскольку они часто привносят дополнительные риски или делают код менее понятным, особенно для новичков. Обычно прочие аспекты игры не менее важны, чем производительность, поэтому перед любыми изменениями следует подумать – не слишком ли многим приходится жертвовать для оптимизации производительности.

Более продвинутые методы оптимизации сценариев будут описаны в *главе 7 «Мастерство управления памятью»*, а пока отвлечемся от кода и рассмотрим способы повышения производительности с помощью встроенных функций Unity, например статическую и динамическую пакетную обработку.

Глава 3

Преимущества пакетной обработки

В 3D-графике и играх понятие **пакетной обработки** представляет собой весьма общий термин, описывающий процесс группировки большого количества непостоянных частей данных и их совместной обработки как единого большого блока данных. Целью этого процесса является сокращение времени вычислений, часто с помощью параллельной обработки, или уменьшение накладных расходов, когда целые пакеты рассматриваются как отдельные элементы. В некоторых случаях пакетная обработка применяется к мешам, множествам вершин, граней, UV-координатам и т. д., используемым для представления трехмерного объекта. Однако тот же термин относится и к пакетной обработке аудиофайлов, спрайтов, текстур и других больших наборов данных.

Поэтому, чтобы избежать путаницы при упоминании пакетной обработки в Unity, под ней понимаются два главных механизма обработки файлов мешей: *статическая* и *динамическая* пакетная обработка. По сути, эти методы являются формой создания экземпляров геометрических структур, позволяющих неоднократно использовать одни и те же данные мешей для отображения одного и того же объекта, без необходимости выполнять подготовку данных несколько раз.

Пакетная обработка позволяет повысить производительность приложения, но только когда она используется разумно. Ее использование связано с массой нюансов и запутанных условий, которые важны для улучшения производительности. На самом деле в некоторых случаях пакетная обработка может даже ухудшать производительность, если используется для обработки наборов данных, непригодных для пакетирования.

Системы пакетной обработки в Unity фактически являются черными ящиками, о внутренней структуре которых компания Unity Technologies не раскрывает подробной технической информации. Но, основываясь на поведении, результатах профилирования и перечне требований к условиям использования, можно многого добиться. В этой главе мы попытаемся развеять мифы, окружающие систему пакетной обработки, и исследуем работу двух ее методов пакетной обработки на примерах. Это позволит принимать обоснованные решения при использовании их для повышения производительности приложений.

Вызовы системы визуализации

Прежде чем перейти к обсуждению статической и динамической пакетной обработки, сначала разберемся, какие задачи они призваны решать в рамках конвейера обработки графики. Постараемся при этом особенно не углубляться в технические детали. Более подробно эта тема будет рассмотрена в *главе 6 «Динамическая графика»*.

Основной целью методов пакетной обработки является сокращение количества обращений к системе визуализации, необходимых для отображения всех объектов в текущем представлении. **Вызов системы визуализации** – это запрос на отображение объекта, посылаемый центральным процессором графическому процессору. Но перед этим должно быть выполнено несколько важных условий. Во-первых, меши и текстуры должны быть перемещены из памяти процессора (RAM) в память графического процессора (VRAM), что обычно происходит при инициализации сцены. Затем центральный процессор должен подготовить графический процессор, определив функции отображения и их параметры, необходимые для обработки объекта.

Взаимодействие между центральным и графическим процессорами осуществляется с помощью низкоуровневого программного интерфейса, такого как DirectX или OpenGL, в зависимости от целевой платформы и оборудования. Эти программные интерфейсы содержат массу сложных взаимосвязанных параметров, переменных состояния и наборов данных, которые подлежат настройке с учетом особенностей данного аппаратного устройства. Массив параметров, настраиваемых перед отображением единственного объекта, обычно обозначается общим термином *состояние визуализации*. Пока состояние визуализации остается неизменным, графический процессор

применяет его ко всем входящим объектам и отображает их одинаковым образом.

Изменение состояния визуализации – длительный процесс. Не вдаваясь в подробности, отмечу лишь, что состояние визуализации – это набор глобальных переменных, влияющих на работу всего графического конвейера. Изменить глобальные переменные в параллельной системе проще на словах, чем на деле. Графический процессор должен выполнить массу действий для синхронизации изменений, в том числе дождаться завершения обработки текущего пакета. В системе с массовым параллелизмом, такой как графический процессор, на ожидание завершения обработки одного пакета может уйти много ценного времени. Такая синхронизация может быть вызвана передачей текстуры, изменением шейдера, изменением информации об освещении, тенях, прозрачности и любых других параметрах, которые только можно представить.

После настройки состояния визуализации центральный процессор должен выбрать меш для вывода объекта, вид материала и местоположение, основываясь на координатах, повороте и масштабе (все это предоставляется в виде единой матрицы преобразования). Для поддержания динамичной связи между центральным и графическим процессорами новые запросы помещаются в буфер команд – список, куда центральный процессор посылает инструкции и откуда графический процессор извлекает их после выполнения предыдущей команды. Буфер команд действует по принципу «первым зашел, первым вышел» (FIFO), и всякий раз, когда графический процессор заканчивает выполнение очередной команды, он извлекает первую команду из очереди, обрабатывает ее и повторяет процесс, пока буфер команд не опустеет.

Обратите внимание, что новый вызов системы визуализации не обязательно приводит к необходимости настройки нового состояния. Если два объекта имеют одно и то же состояние визуализации, графический процессор может немедленно начать отображение нового объекта, поскольку состояние визуализации после вывода предыдущего объекта сохраняется.

Так как процесс визуализации требует работы двух аппаратных компонентов в тандеме, он очень чувствителен к узким местам одного или тем более обоих. Графический процессор способен невероятно быстро отображать отдельные объекты, но если центральный процессор тратит слишком много времени на подготовку команд для системы визуализации (или просто генерирует их слишком много),

графический процессор будет чаще занят ожиданием инструкций, чем непосредственной работой. В этом случае графика приложения будет ограничена возможностями центрального процессора. Больше времени будет уходить на ожидание решений центрального процессора, чем на само рисование. И наоборот, недостаточная производительность графического процессора приведет к заполнению буфера команд необработанными запросами, потому что графический процессор не сможет обрабатывать их со скоростью поступления от центрального процессора.

Более подробно наличие узких мест в центральном и графическом процессорах и решение этих проблем рассматривается в главе 6 «Динамическая графика».

Еще один компонент, способный замедлять обработку графики в этой цепочке, – аппаратный драйвер. Драйвер играет роль посредника при передаче команд, поступающих из нескольких источников, таких как наше приложение, другие приложения и даже сама операционная система (например, при отображении рабочего стола), через графический программный интерфейс. Вследствие этого обновление драйверов иногда приводит к довольно значительному увеличению производительности!

Графические программные интерфейсы следующего поколения, например Microsoft DirectX 12, Apple Metal и Kronos Vulcan, уменьшают накладные расходы, упрощая и распараллеливая обработку определенных задач, в частности передачу инструкций в буфер команд. После того как эти прикладные программные интерфейсы распространятся повсеместно, можно будет более комфортно использовать вызовы системы визуализации. Но пока эти программные интерфейсы не достигли зрелости, следует с большой осторожностью относиться к вызовам системы визуализации, чтобы избежать ограничений, связанных с центральным процессором.

Материалы и шейдеры

Шейдеры – это короткие программы, которые определяют, как графический процессор должен отображать входные данные, описывающие вершины и пиксели. Шейдеры сами по себе не обладают необходимыми сведениями о состоянии, чтобы сделать что-то ценное. Для этого им требуются входные данные, например текстуры, карты нормалей, цвета и т. д., а также переменные состояния визуализации.

Важной обязанностью системы **материалов** в Unity является предоставление этой информации шейдерам. Это значит, что шейдер может использоваться для отображения объекта только вместе с материалом. Каждому шейдеру нужен материал, и каждый материал нуждается в шейдере. Даже вновь импортированным мешам, вводимым в сцену без назначенных материалов, автоматически присваивается материал по умолчанию (скрытый), который придает им базовый диффузный шейдер и белый цвет. Таким образом, эту взаимосвязь разорвать невозможно.

Материал может поддерживать только один шейдер. Использование нескольких шейдеров для одного меша требует назначения нескольких материалов разным частям меша.

Эти две системы охватывают большинство переменных состояния визуализации, которые были рассмотрены в предыдущем разделе. То есть если свести к минимуму количество материалов, используемых для визуализации сцены, автоматически уменьшится количество необходимых изменений состояния визуализации и, соответственно, количество времени, затрачиваемого центральным процессором на подготовку кадров для графического процессора.

Начнем с простой демонстрации поведения материалов и пакетной обработки. Но прежде отключим несколько глобальных параметров механизма визуализации, чтобы не отвлекаться:

- выберите в меню пункт **Edit** ⇨ **Project Settings** ⇨ **Quality** (Правка ⇨ Параметры проекта ⇨ Качество) и установите значение **Disable Shadows** (Отключить тени) в параметре **Shadows** (Тени) или установите уровень качества по умолчанию **Fastest** (Быстрый);
- выберите в меню пункт **Edit** ⇨ **Project Settings** ⇨ **Player** (Правка ⇨ Параметры проекта ⇨ Проигрыватель), откройте вкладку **Other Settings** (Прочие параметры) и сбросьте флажки **Static Batching** (Статическая пакетная обработка), **Dynamic Batching** (Динамическая пакетная обработка) и **GPU Skinning** (Скининг графическим процессором).

Параметры **Static Batching** (Статическая пакетная обработка) и **GPU Skinning** (Скининг графическим процессором) не доступны в редакции Free Unity 4, для их использования необходимо обновиться до редакции Pro Unity 4.

Далее создадим сцену с единственным направленным источником света и восемью мешами – четыре куба и четыре сферы, – каждый из которых имеет уникальный материал, позицию, поворот и масштаб, как показано на рис. 3.1.

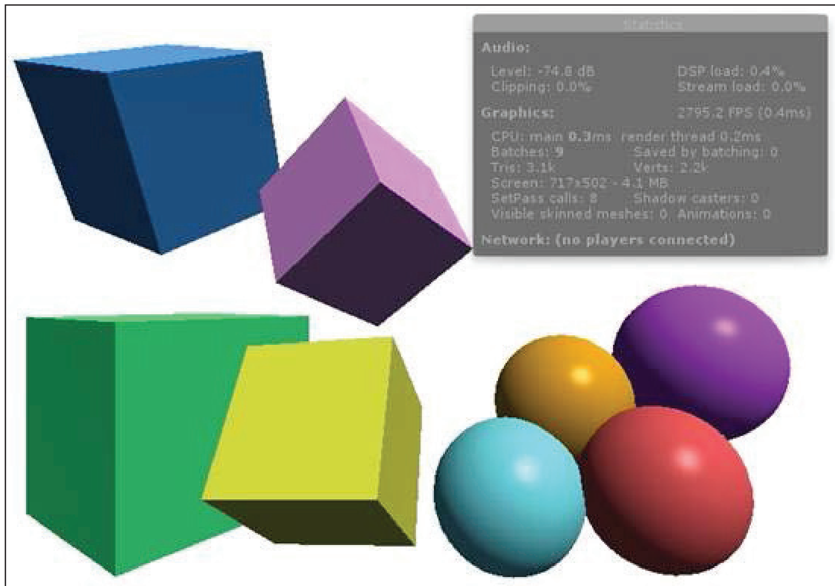


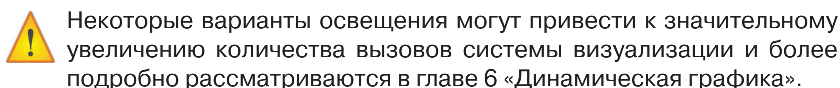
Рис. 3.1 ❖ Тестовая сцена

Взгляните на параметр **Batching** (Пакетная обработка) во всплывающей панели **GameView's Stats** (Статистика представления игры), и вы увидите, что число пакетов равно девяти (это же число стоит в параметре **Draw Calls** (Вызовы системы визуализации) во всплывающей панели **Stats** (Статистика) в Unity 4). Если в параметре **Clear Flags** (Флаги очистки) камеры выбран вариант **Don't Clear** (Не очищать), один пакет будет использован для рисования фона. Это может быть объект Skybox сцены или прямоугольник, заполняющий экран пикселями, окрашенными в цвет, определяемый свойством камеры **Background** (Фон).

Прочие восемь пакетов используются для рисования восьми объектов. В любом случае, каждый вызов системы визуализации включает ее подготовку с учетом свойств материала и посылает графическому

процессору команду отобразить данный меш в его текущей позиции, с текущими поворотом и масштабом. Материал определяет шейдер, управляющий программируемыми частями графического конвейера (управляющими отображением вершин и фрагментов).

Примечательно, что если в параметре **Rendering Path** (Режим отображения) настройки плеера выбран вариант **Forward** (Упреждающий), включение и выключение направленного источника света в сцене не повлияет на количество пакетов, их будет по-прежнему девять. Первый направленный источник света в режиме **Forward** не вносит никаких дополнительных затрат, по крайней мере с точки зрения вызовов системы визуализации. При добавлении других источников света в сцену – направленных, точечных, локальных или пространственных – все объекты обрабатываются с дополнительным «прогоном» шейдеров для каждого источника, в зависимости от значения параметра **Pixel Light Count** (Количество пиксельных источников света) в настройках **Quality** (Качество); приоритет при обработке будут иметь источники света с более высокой яркостью.


Некоторые варианты освещения могут привести к значительному увеличению количества вызовов системы визуализации и более подробно рассматриваются в главе 6 «Динамическая графика».

Как упоминалось выше, теоретически можно минимизировать количество вызовов системы визуализации, уменьшив частоту смены состояний визуализации. То есть один из путей достижения цели – уменьшение количества используемых материалов. Но если для всех объектов использовать один и тот же материал, как показано на рис. 3.2, это ничего не даст, и по-прежнему останется девять пакетов:

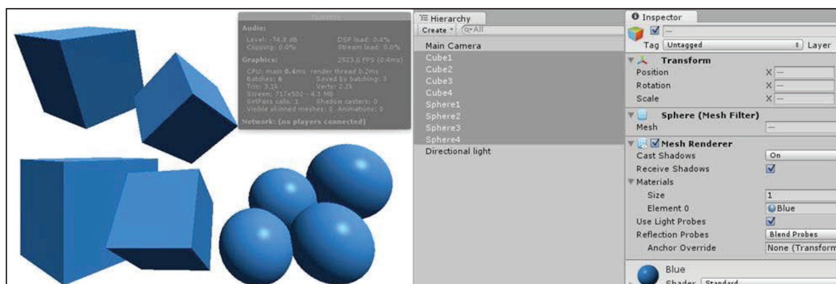


Рис. 3.2 ❖ Все равно осталось девять пакетов

Причина в том, что без группировки информации о мешах число изменений состояния визуализации останется тем же. Без пакетной обработки какого-либо вида система визуализации не сможет понять, что выполняется перезапись одинаковых состояний визуализации при выводе нескольких мешей, и будет перезаписывать их снова и снова. Это дает возможность выводить в процессе визуализации все меши вместе как один объект и избежать ненужных вызовов визуализации. Так в основном и работает динамическая пакетная обработка, уменьшая количество вызовов системы визуализации.

Динамическая пакетная обработка

Цель динамической пакетной обработки – объединить простые меши в большие группы и передать их системе визуализации в виде одного меша. Ее можно применять только к мешам, видимым в настоящий момент для **камеры**, а это значит, что основные вычисления должны производиться во время выполнения и не могут быть выполнены заранее. Соответственно, набор объединяемых объектов будет изменяться от кадра к кадру. Этим и объясняется применение термина «динамическая» к такой пакетной обработке.

Если вернуться на страницу **Player Settings** (Настройки плеера) и включить параметр **Dynamic Batching** (Динамическая пакетная обработка), количество пакетов должно уменьшиться с девяти до шести. Механизм динамической пакетной обработки автоматически определил, что объекты имеют одинаковый материал и схожие параметры мешей, и их можно объединить в один пакет. Это сэкономит ресурсы центрального процессора и высвободит больше времени для решения других задач, например искусственного интеллекта и моделирования законов физики.

Рискуя выглядеть неблагодарными, мы должны все же спросить, почему были исключены только три вызова визуализации, а не шесть. Мы рассчитывали, что система достаточно умна, чтобы, сгруппировав все кубики в один пакет и все сферы в другой, выполнить для их отображения только два вызова (плюс один вызов для отрисовки фона).

Полный список требований, необходимых для успешной динамической пакетной обработки мешей, можно найти в электронной документации Unity по адресу: http://docs.unity3d.com/Manual/DrawCallBatching.html:

- все экземпляры мешей должны использовать одну и ту же ссылку на материал;

- динамическая пакетная обработка применяется только к системам частиц и мешам. Компоненты `SkinnedMesh` (для анимированных персонажей) и другие не могут объединяться;
- общее количество атрибутов вершинного шейдера не должно превышать 900;
- все меши должны использовать либо однородное, либо неоднородное масштабирование, но не их сочетание;
- все экземпляры мешей должны ссылаться на один и тот же файл с картой освещенности;
- шейдеры материалов не должны зависеть от количества проходов;
- экземпляры мешей не должны принимать тени в реальном времени.

Имеется также несколько недокументированных требований, выявленных в ходе конференций Unite:

- пакет вмещает не более 300 мешей;
- пакет не должен содержать более 32 000 индексов мешей.

Последние два требования нельзя назвать интуитивно понятными. Кроме того, их описание отсутствует в документации, поэтому остановимся на них отдельно.

Атрибуты вершин

Атрибут вершины – свойство в файле меша, описывающее вершину и включающее координаты вершины, вектор нормали (часто используется в расчетах освещения), а также UV-координаты (используются для наложения текстуры). Только меши с общим количеством атрибутов вершин не более 900 могут включаться в динамическую пакетную обработку.

Обратите внимание, что файл с исходными данными для меша может содержать меньше атрибутов вершин, чем после его загрузки движком Unity, – в процессе преобразования исходных данных во внутренний формат могут образовываться дополнительные атрибуты.

С увеличением количества атрибутов на одну вершину во вспомогательном шейдере их общее количество может превысить ограничение в 900 атрибутов и уменьшить допустимое количество вершин в меше для участия в динамической пакетной обработке. Например, простой шейдер, использующий только три атрибута для описания вершины, как некоторые из официальных диффузных шейдеров, поддерживает

динамическую пакетную обработку мешей, насчитывающих не более 300 вершин. Более сложные шейдеры, использующие по пять атрибутов на вершину, поддерживают динамическую пакетную обработку мешей, включающих не более 180 вершин.

Именно этим ограничением объясняется, почему при включении динамической пакетной обработки в нашей сцене было сэкономлено только три вызова системы визуализации, несмотря на то что все объекты используют ссылки на один и тот же материал. Куб, сгенерированный в Unity, содержит 8 вершин, для каждой из которых определяются координаты, нормаль и UV-данные, всего 24 атрибута. Это меньше ограничения в 900 атрибутов. Но автоматически созданная сфера содержит 515 вершин, которые явно не могут быть объединены динамически, если подсчитать количество атрибутов (координат, нормалей и UV-координат) всех вершин. Этим объясняется наличие шести вызовов системы визуализации: один – для фона, один – для пакета кубов и четыре – для сфер, рисование которых выполняется отдельными вызовами.

Однородное масштабирование

Документация явно оговаривает, что для включения в динамическую пакетную обработку все объекты должны иметь либо однородное масштабирование, либо неоднородное. На самом деле это ограничение отличается для разных версий Unity из-за продолжающегося совершенствования системы динамической пакетной обработки в Unity 5.

Однородным называется такое масштабирование, когда все три компонента вектора масштабирования (x, y и z) идентичны.

В обеих версиях Unity, если все экземпляры меша имеют один и тот масштаб, они могут быть сгруппированы в один пакет для динамической обработки. В Unity 5 в один пакет могут объединяться все экземпляры меша с неоднородным масштабированием, независимо от масштаба. Однако в Unity 4 объекты с неоднородным масштабированием будет помещены в отдельные пакеты, даже использующие один и тот же меш и материал.

При этом предполагается, что речь идет о положительном масштабировании. С отрицательным масштабированием не все так просто. В Unity 4 отрицательные масштабы не влияют на динамическую пакетную обработку, и к ним применяются те же правила. Но в Unity 5, если меш содержит одно или три отрицательных значения в векторе масштабирования, он не включается в пакет. Если меш содержит два

отрицательных значения, он может объединяться с другими экземплярами. Совершенно не важно, какие из трех значений являются отрицательными, важно, чтобы их было два или ни одного.

Вероятно, это проявление алгоритма, используемого для проверки возможности объединения групп, так как зеркальное отражение меша в двух измерениях математически эквивалентно повороту меша вокруг тех же двух осей на 180 градусов. То есть поведение системы динамической пакетной обработки можно расценивать как автоматическое преобразование.

Итак, имейте в виду, что динамическая пакетная обработка в Unity 5 более эффективна, но и она не лишена компромиссов. При использовании отрицательных масштабов для зеркального отражения меша теряется возможность применять его в динамической пакетной обработке.

Краткие выводы о динамической пакетной обработке

В систему динамической пакетной обработки Unity 5 внесены значительные улучшения, сделавшие ее более универсальной. Однако независимо от версии Unity динамическая пакетная обработка очень полезна при отображении больших наборов простых мешей.

Система динамической пакетной обработки идеально подходит для случав, когда используются простые и схожие внешне меши, такие как:

- большой лес со скалами, деревьями и кустарниками;
- здание, фабрика или космическая станция со множеством похожих элементов (компьютеры, трубы и т. д.), разбросанных по всей сцене;
- игра с множеством подвижных объектов простой геометрической формы и без анимации или с эффектом частиц (в качестве примера можно привести игру Geometry Wars).

Потенциальные преимущества динамической пакетной обработки достаточно значимы, чтобы выделить некоторое время на изучение требований, предъявляемых к сцене. Однако в некоторых ситуациях динамическая пакетная обработка ухудшает производительность.

Наиболее распространенной ошибкой является применение динамической пакетной обработки к сцене, в которой генерируется масса пакетов с большим количеством мешей в каждом. В этом случае накладные расходы на поиск и создание пакетов оказываются больше, чем выигрыш от пакетной обработки.

Кроме того, ухудшение производительности приложения нередко происходит из-за невозможности применения пакетной обработки, когда не выполнено одно из важных ее требований. Все случаи уникальны, поэтому стоит экспериментировать с разными материалами, мешами и шейдерами для определения их пригодности к динамической пакетной обработке.

Статическая пакетная обработка

Еще один механизм пакетной обработки в Unity реализует статическую пакетную обработку. Его назначение – дать возможность объединять в пакеты меши любого размера с той же целью и теми же способами, что и при динамической пакетной обработке, но при другом наборе условий. Существенное отличие между двумя методами пакетной обработки заключается в том, что статическая пакетная обработка происходит во время инициализации приложения, а динамическая – во время выполнения. Поэтому разработчик имеет намного больше возможностей для управления статической пакетной обработкой.

Статическая пакетная обработка доступна во всех редакциях Unity 5, но в Unity 4 потребуется редакция Pro. Для использования этого механизма пользователям редакции Free Unity 4 придется обновиться до редакции Pro.

Система статической пакетной обработки имеет свой набор требований:

- как следует из названия, меш должен иметь флаг Static (Статический);
- для каждого меша, участвующего в статистической пакетной обработке, должна выделяться дополнительная память;
- экземпляры мешей могут поступать из любых источников, но должны использовать один и тот же материал.

Рассмотрим каждое из этих требований подробнее.

Флаг Static

Статическая пакетная обработка может применяться только к объектам с установленным флагом Static (Статический) или, более конкретно, флагом **Batching Static** (Статическая пакетная обработка, также известная как StaticEditorFlag). Если щелкнуть на значке со стрелкой вниз рядом с параметром **Static** (Статический) игрового

объекта, появится список параметров **StaticEditorFlags** (Флаги для статического поведения), изменяющих поведение объекта в различных статических процессах.

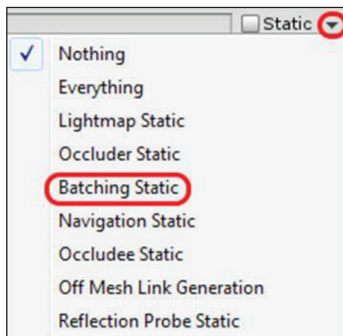


Рис. 3.3 ❖ Список параметров **StaticEditorFlags** (Флаги для статического поведения)

Требования к памяти

Потребность в дополнительной памяти для статической пакетной обработки зависит от количества копий мешей, создаваемых внутри пакета. При статической пакетной обработке все отмеченные меши объединяются в один большой меш, который передается в систему визуализации через единственный вызов. Если все меши, участвующие в статической пакетной обработке, являются *уникальными*, будет использовано памяти ровно столько, как при обычном отображении объектов, то есть не больше, чем необходимо для хранения отдельных мешей.

Но статическая пакетная обработка дубликатов приводит к затратам дополнительной памяти, объем которой равен произведению количества мешей на размер. Обычно при выводе одного, десяти или миллиона копий одного и того же объекта затрачивается одно и то же количество памяти, поскольку все они ссылаются на один и тот же меш. Единственным отличием является преобразование каждого объекта в отдельности. Но поскольку при статической пакетной обработке требуется копировать данные в большой буфер вместе с данными для преобразований, использование ссылки становится невозможным, и оригинальный меш копируется в буфер с жестко заданным преобразованием, независимо от того, был ли он уже скопирован ранее или нет.

То есть при использовании статической пакетной обработки для отображения 1000 идентичных объектов деревьев будет затрачено в 1000 раз больше памяти, чем при отображении тех же объектов по отдельности, поскольку все деревья будут скопированы в буфер статической пакетной обработки как уникальный набор вершин. В этом случае неумелое использование статической пакетной обработки приведет к существенным проблемам использования памяти и потере производительности.

Ссылки на материалы

Теперь, когда мы знаем, что использование ссылок на материалы уменьшает число изменений состояния визуализации, это требование выглядит очевидным. Однако иногда приходится подвергать статической пакетной обработке меши, использующие несколько материалов. В этом случае меши будут сгруппированы по материалам.

В этом случае для отображения всех статических мешей понадобится количество вызовов визуализации, в лучшем случае равное количеству материалов. Зато появляется возможность управлять объединением мешей выбором различных материалов. Фактически можно принудительно начать новый статический пакет, продублировав материал. Чтобы лучше понять суть, познакомимся с некоторыми особенностями применения системы статической пакетной обработки.

Особенности использования статической пакетной обработки

Система статической пакетной обработки не лишена недостатков. Так как сборка пакетов осуществляется путем объединения мешей в один большой меш, система статической пакетной обработки обладает несколькими особенностями, о которых следует знать. В зависимости от сцены эти особенности вызывают проблемы разной серьезности, от незначительных неудобств до крупных недостатков:

- экономия на вызовах системы визуализации не сразу становится заметной в окне **Stats** (Статистика);
- статические объекты не должны вводиться в сцену во время выполнения;
- нельзя изменять параметры преобразования мешей из статических пакетов;
- если любой из мешей в статическом пакете является видимым, отображается вся группа.

Отладка статической пакетной обработки в режиме редактирования

Определение общего эффекта от применения статической пакетной обработки в сцене несколько затруднено, поскольку статическая пакетная обработка никак не отражена в режиме редактирования. Все волшебство происходит в момент инициализации сцены, что не дает возможности оценить реальные преимущества статической пакетной обработки. При этом особенно важно не оставлять реализацию этой функции на завершающий этап работы над проектом, когда потребуется потратить много времени на запуск, настройку и повторный запуск сцены, чтобы убедиться в ожидаемом уменьшении количества вызовов визуализации. Следовательно, применение статической пакетной обработки лучше начинать одновременно с созданием новой сцены.

Статические меши не должны создаваться во время выполнения

Вновь созданные статические объекты в сцене не присоединяются к существующему пакету автоматически, в противном случае это привело бы к огромным накладным расходам, связанным с перерасчетом мешей и синхронизацией с системой визуализации. Поэтому движок Unity даже не пытается сделать это. Это означает, что не следует даже пытаться динамически создавать экземпляры для включения в статическую пакетную обработку. Все меши, включаемые в статические пакеты, должны изначально присутствовать в исходном файле сцены.

Динамическое создание объектов само по себе не требует много ресурсов. Решение этой проблемы будет рассмотрено в *главе 7* «*Мастерство управления памятью*».

Точно так же не следует перемещать меши из статических пакетов после их объединения, поскольку это вызовет огромные нагрузки на процессор. Система статической пакетной обработки объединяет принятые исходные данные в новый объект меша. Перемещение оригинального меша невозможно ни при установленном, ни при сброшенном флаге Static, и изменение параметров преобразования объекта не распознается системой Unity.

Видимость и отображение

Последняя особенность не менее важна. Даже если видна только одна вершина меша, включенного в статический пакет, системе визуализации передается весь меш и обрабатывается ею целиком. Необходимо разумно подходить к использованию статической пакетной обработки и стараться предотвращать вывод гигантских мешей в каждом кадре!

Например, если имеется сцена со множеством смежных комнат из одного и того же материала, включение в статическую пакетную обработку всех комнат приведет к отображению всех комнат, даже если видна только одна и даже если взгляд игрока направлен прямо на стену. В таких сценах желательно использовать механизм окклюзивной выбраковки (рассматривается в *главе 6 «Динамическая графика»*), чтобы предотвратить отображение невидимых комнат.

Кроме того, для больших открытых сцен полезно создавать дубликаты материалов для распределения одинаковых статических объектов по разным группам. Назначение идентичных копий материалов разным разделам игрового мира приведет к размещению их в разных группах. Это увеличит количество обращений к системе визуализации, но позволит повторно использовать те же текстуры, материалы и файлы шейдеров для всей сцены.

Создание копий материалов затрудняет сопровождение, так как изменения в любом материале придется распространить на его копии. На первый взгляд кажется, что эту проблему можно решить с помощью присоединенного компонента сценария, автоматически создающего дубликат материала и присваивающего его мешу при инициализации сцены. К сожалению, статическая пакетная обработка выполняется в самом начале процесса инициализации, и ее нельзя перехватить с помощью подобного пользовательского компонента, поэтому данную задачу придется решать с помощью редактора (хорошая возможность использовать редактор сценариев).

Краткие выводы о статической пакетной обработке

Статическая пакетная обработка – мощный, но опасный инструмент. При неосторожном его использовании легко можно нанести серьезный ущерб производительности, увеличив объем используемой памяти и затраты на отображение. Но он имеет широкие возможности настройки и значительное преимущество: способность работать с на-

бором объектов неограниченного размера. По существу, статическая пакетная обработка ничем не ограничена.

Итоги

Очевидно, что ни статическая, ни динамическая пакетные обработки не являются панацеей. Их нельзя слепо применить к любой сцене и ожидать явных улучшений. Если приложение и сцена соответствуют определенному набору требований, эти методы позволяют очень эффективно снизить нагрузку на процессор и избавиться от узких мест в визуализации. В противном случае придется приложить дополнительные усилия, чтобы привести сцену в соответствие с этими требованиями. В конечном счете только хорошее понимание особенностей системы пакетной обработки поможет определить, где и когда ее применять.

Дополнительные методы оптимизации вывода графики будут рассмотрены в *главе 6 «Динамическая графика»*. А пока займемся другими вопросами, которые касаются некоторых тонкостей способов увеличения производительности, связанных с мастерским управлением ресурсами.

Глава 4

Привнесение искусства

Искусство – весьма субъективная область, в которой преобладают личные мнения и предпочтения. Очень сложно определить, какой из двух разделов искусства «лучше», и вполне вероятно, что здесь невозможно прийти к консенсусу. Технические аспекты искусства, придающие игре артистизм, также очень субъективны. Существует несколько обходных путей, позволяющих улучшить производительность, но они, как правило, ведут к потере качества ради увеличения скорости. Стремясь достигнуть максимальной производительности, очень важно проконсультироваться с членами команды, прежде чем вносить изменения в объекты искусства, поскольку поддержание равновесия само по себе является искусством.

Снизить потребление памяти, уменьшить размер исполняемого файла, достичь максимальной скорости загрузки или обеспечить согласованность частоты кадров можно разными путями. Одни методы всегда приводят к нужным результатам, другие требуют внимательности и предусмотрительности, поскольку могут привести к снижению качества или увеличить риск появления узких мест в других компонентах.

Наши исследования в этой главе мы начнем с аудиофайлов, затем перейдем к файлам текстур и закончим мешами и анимацией. В каждом случае мы рассмотрим, как Unity загружает, хранит и управляет этими ресурсами во время выполнения, а также узнаем, как избежать создания узких мест производительности.

Аудио

В зависимости от назначения проекта его аудиофайлы могут иметь любые размеры, от занимающих солидную часть диска до очень небольших. Unity может использоваться как для создания небольших приложений, требующих минимума звуковых эффектов и единственного трека с фоновой мелодией, так и для разработки ролевых игр, нуждающихся в речевых диалогах с общим объемом в миллионы строк, музыкальных треках и звуках окружения. Поэтому давайте посмотрим, как Unity управляет аудиофайлами, чтобы понять, что необходимо сделать для оптимизации в обоих случаях.

Многие разработчики с удивлением обнаруживают, что процесс обработки аудио может стать значимым потребителем памяти и ресурсов процессора. Звуком часто пренебрегают обе заинтересованные стороны игровой индустрии; разработчики, как правило, не подключают многих ресурсов до самого последнего момента, а пользователи редко обращают внимание на звук. Никто не замечает, когда аудиосопровождение хорошее или просто приемлемое, но как звучит плохое аудио, знают все – оно навязчиво, режет слух и гарантированно привлекает нежелательное внимание. Поэтому важно не слишком сильно жертвовать качеством аудио ради производительности.

Обработка аудио может стать узким местом в силу нескольких причин. Чрезмерное сжатие, большое количество операций с аудио, масса активных аудиоклипов, использование неэффективных методов хранения в памяти и низкая скорость доступа – все это способствует увеличению потребления памяти и ресурсов центрального процессора. Но немного усилий – и разумный подход к выбору параметров настройки позволит избежать негативного восприятия игры пользователем.

Варианты использования аудио в Unity 4 и Unity 5 называются немного по-разному, но функционально они одинаковы. Здесь будут приведены названия для Unity 5.

Загрузка аудиофайлов

Первый параметр импортированного аудиоклипа в редакторе Unity определяет тип загрузки файла (**Load Type** (Тип загрузки)). В этом параметре можно выбрать одно из трех значений:

- **Decompress On Load** (Распаковывать при загрузке);
- **Compressed In Memory** (Хранить в памяти в сжатом виде);
- **Streaming** (Потоковая передача).

Фактически от этого параметра зависят доступные значения и порядок использования остальных параметров, которые мы рассмотрим ниже. Он определяет метод загрузки выбранного аудиофайла.

Если выбрать вариант **Decompress On Load** (Распаковывать при загрузке), аудиофайл, хранящийся на диске в сжатом виде для экономии места, будет распакован при загрузке в память. Это стандартный метод загрузки аудиофайлов, и в большинстве случаев именно он должен применяться.

Если выбрать вариант **Compressed In Memory** (Хранить в памяти в сжатом виде), сжатый файл загрузится в память без обработки и будет распаковываться непосредственно при воспроизведении. Здесь за счет увеличения потребления центрального процессора при воспроизведении увеличивается скорость загрузки клипа и уменьшается потребление памяти во время выполнения. Этот вариант лучше подходит для больших и часто используемых аудиофайлов, или если потребление памяти жестко ограничено, но можно пожертвовать некоторым временем центрального процессора для воспроизведения аудиоклипа.

И наконец, если выбрать вариант **Streaming** (Потоковая передача), файлы будут загружаться, декодироваться и воспроизводиться «на лету» с использованием небольшого буфера. Этот метод имеет минимальное потребление памяти для отдельных аудиоклипов и больше всего нагружает центральный процессор. К сожалению, он не позволяет сослаться на тот же буфер повторно. Попытка запустить несколько потоков воспроизведения одного и того же аудиоклипа приведет к выделению в памяти нового буфера для каждого потока, что при бездумном применении этого метода приведет к значительному потреблению оперативной памяти и ресурсов центрального процессора. Как следствие этот вариант лучше подходит для регулярного воспроизведения единственного экземпляра аудиоклипа, который никогда не пересекается с другими экземплярами. То есть этот параметр следует использовать для воспроизведения фоновой музыки или звуковых эффектов, звучащих большую часть времени в сцене.

Профилирование аудио

Чтобы подтвердить большую часть из описанного выше, воспроизведем несколько экземпляров произвольного аудиоклипа с помощью нескольких источников звука в сцене и проведем сравнительный анализ с помощью аудиопрофилировщика, отображающего потребление памяти и ресурсов центрального процессора при использовании

разных значений в параметре **Load Type** (Тип загрузки). Но имейте в виду, что результаты профилирования в режиме редактора могут ввести в заблуждение, поскольку редактор загружает аудио не так, как это происходит при выполнении приложения.

После загрузки и перехода в режим воспроизведения редактор распакует аудиофайлы, затратив некоторый объем памяти и ресурсов центрального процессора. Затраты памяти на этот процесс можно увидеть в области **Audio Area** профилировщика. Однако после перезапуска сцены затраты памяти на распаковку аудиофайлов внезапно снижаются практически до 0 Кбайт, поскольку файлы уже были распакованы и редактор избавился от данных, которые ему больше не нужны. Это не соответствует реальной ситуации, так как приложению потребуется выполнить эту распаковку.

То есть, чтобы получить более точные результаты, следует производить профилирование автономной или удаленной версии приложения на предполагаемой платформе/устройстве.

Дополнительные варианты загрузки

Имеются еще два параметра, влияющих на поведение загрузки аудиофайла:

- загрузка в фоновом режиме;
- предварительная загрузка аудиоданных.

Обычно, когда аудиоклип назначается компоненту источника звука в сцене, файл загружается в память при инициализации сцены. Но если установлен параметр **Load In Background** (Загружать в фоновом режиме), загрузка аудиоклипа будет передана фоновому заданию, которое запустится после завершения инициализации сцены, и начнется игровой процесс. То есть можно с уверенностью предположить, что установка этого параметра увеличит скорость загрузки сцены, если аудиоклипы потребуются в сцене позднее, как, например, звуковые эффекты, используемые при смене *промежуточных уровней*.

Проверить готовность ресурса, перед тем как им воспользоваться, можно при помощи свойства `loadState` объекта `AudioClip`. Но установка параметра **Load In Background** (Загружать в фоновом режиме) может привести к нестыковке, если попытаться открыть не полностью загруженный звуковой файл. В этом случае воспроизведение будет отложено до завершения фоновой загрузки, что наверняка приведет к появлению звука в неподходящий момент.

Второй параметр – **Preload Audio Data** (Предварительная загрузка аудиоданных) – установлен по умолчанию и требует автоматически

выполнить загрузку файла во время инициализации сцены. Отключение этого параметра откладывает загрузку до первого вызова метода `Play()` или `PlayOneShot()` объекта `AudioSource` во время выполнения. Это вызовет всплеск потребления центрального процессора из-за необходимости загрузить, распаковать (в зависимости от значения параметра **Load Type** (Тип загрузки)), поместить в память и только потом воспроизвести аудиоданные.

Из-за задержки воспроизведения и снижения производительности не рекомендуется полагаться на загрузку в момент, когда потребуется воспроизвести аудиофайл. Вместо этого лучше загрузить файл заранее, вызвав метод `LoadAudioData()` объекта `AudioClip` в удобный для этого момент. Современные игры часто выполняют загрузку в точках приостановки между заданиями, например во время перемещения в лифте с этажа на этаж или по длинным коридорам, где происходит очень мало действий. Обратите внимание, что память, занимаемую аудиофайлом, можно освободить вручную, с помощью метода `UnloadAudioData()` объекта `AudioClip`.

Применение нестандартной загрузки и выгрузки аудиоданных с помощью этих методов может потребоваться в отдельных играх, в зависимости от того, когда возникает необходимость в аудиоклипах, как долго они нужны, как разделены сцены и как игрок должен их пройти. Такая загрузка может потребовать значительного объема специальных изменений, тестирования и настройки управления ресурсами. Поэтому данный подход рекомендуется рассматривать в качестве «ядерного оружия» и использовать, только если применение всех прочих методов не увенчалось успехом.

Форматы кодирования и уровни качества

Unity поддерживает три распространенных формата кодирования аудиофайлов, а также несколько специальных для конкретных платформ (например, HEVAG для PS Vita и XMA для XBox One). Рассмотрим три основных формата кодирования:

- **Compressed**;
- **PCM**;
- **ADPCM**.

Алгоритм сжатия, применяемый при выборе формата **Compressed**, зависит от целевой платформы. В автономных приложениях, WebGL и других немобильных платформах применяется сжатие Ogg-Vorbis, а мобильные платформы используют сжатие MPEG-3 (MP3).

Движок Unity поддерживает импорт аудиофайлов многих популярных аудиоформатов, но в исполняемый файл встраивается поддержка только одного из перечисленных. Статистические данные, получаемые для формата, который определяется параметром **Compression Format** (Формат сжатия), отражают, сколько места на диске сохраняется при сжатии и сколько памяти экономится при воспроизведении во время выполнения (обратите внимание, что эти величины зависят также от выбранного типа загрузки).

Выбор формата сжатия/кодирования коренным образом влияет на качество, размер файла и потребление памяти при воспроизведении во время выполнения, и только при выборе варианта **Compressed** имеется возможность изменять качество независимо от характеристик файла. Форматы **PCM** и **ADPCM** не предоставляют такой удобной возможности, здесь качество жестко определяется размером файла, поэтому, принимая решение пожертвовать качеством ради уменьшения размера файла, нужно сменить характеристики файла.

Каждый формат кодирования и сжатия имеет свои достоинства и недостатки, и для каждого аудиофайла можно пожертвовать одной из характеристик ради улучшения другой, основываясь на его назначении и содержании. Для лучшей оптимизации аудиофайлов всегда следует готовиться использовать все доступные форматы в рамках одного приложения.

Формат **PCM** не предусматривает сжатия, лишен искажений и обеспечивает качество, близкое к качеству аналогового аудио. Он характеризуется большим размером файлов и наивысшим качеством звука. Его рекомендуется использовать для очень коротких звуковых эффектов, требующих высокой четкости, недостижимой при применении сжатия.

С другой стороны, формат **ADPCM** более эффективно использует память и ресурсы центрального процессора, чем PCM, но сжатие привносит изрядное количество шумов. Эти шумы незаметны в коротких звуковых эффектах хаоса, подобных звукам взрывов, столкновений и ударов, где посторонние шумы не являются критичными.

Наконец, формат **Compressed** позволяет уменьшить размеры файлов и получить качество звучания хуже, чем при использовании формата **PCM**, но лучше, чем при использовании формата **ADPCM**, за счет дополнительного использования ресурсов центрального процессора. Этот формат рекомендован к применению в большинстве случаев. При выборе этого варианта появляется возможность настроить уровень качества алгоритма сжатия и достичь нужного баланса

между качеством и размером файла. Работая ползунком **Quality** (Качество), можно найти минимальный уровень качества, не вносящий искажений, заметных пользователю. Тестирование помогает найти «золотую середину» для каждого из файлов.

Имейте в виду, что дополнительные звуковые эффекты, применяемые к файлу во время выполнения, не будут слышны в режиме редактирования, поэтому их влияние можно проверить только в режиме воспроизведения приложения.

Улучшение производительности аудио

А теперь, вооружившись знанием форматов аудиофайлов, методов загрузки и режимов сжатия, рассмотрим приемы повышения производительности путем настройки обработки аудио.

Уменьшение числа активных источников звука

Так как любой активный источник звука потребляет ресурсы центрального процессора, само собой разумеется, что можно снизить нагрузку на центральный процессор, избавившись от избыточных источников звука в сцене. Одно из решений состоит в том, чтобы жестко ограничить количество экземпляров аудиоклипов, воспроизводимых одновременно, поскольку это приводит к избыточности звуковых эффектов. Добиться этого можно, создав объект, управляющий источниками звука. Он должен принимать запросы и воспроизводить звуки через любой незанятый источник звука.

Объект управления также должен ограничивать количество одновременно воспроизводимых экземпляров одного и того же звукового эффекта и общее количество звуковых эффектов, воспроизводимых одномоментно. При этом предпочтение должно отдаваться звуковым 2D-эффектам или отдельным экземплярам звуковых 3D-эффектов (которые требуют перемещения источника звука в правильное положение на момент воспроизведения).

Почти все инструменты управления звуком, предлагаемые в магазине Asset Store Unity, реализуют функцию ограничения, которую часто называют «объединением в пул», и на то есть веская причина. Эта функция обеспечивает лучшее компромиссное решение для уменьшения воспроизведения избыточных аудиоэффектов с наименьшими усилиями по подготовке аудиоресурсов. По этой причине, а еще и потому, что такие ресурсы часто предоставляют дополнительные функции для увеличения производительности, рекомендуется использо-

вать уже существующие решения вместо создания собственного, так как это потребует написания большого объема кода.

Обратите внимание, что окружающие 3D-звуки все еще придется помещать в определенные места в сцене, чтобы использовать эффект логарифмирования громкости, обеспечивающий псевдо 3D-эффект, поэтому система управления не является идеальным решением. Ограничение воспроизведения аудиоэффектов проще реализовать путем сокращения общего числа источников. С этой целью лучше всего удалять некоторых из них или объединять в один громкий источник звука. Естественно, такой подход влияет на качество восприятия пользователем, поскольку звук будет исходить из одного источника, а не из нескольких, соответственно, данный способ следует использовать с осторожностью.

Уменьшение числа ссылок на аудиоклипы

Каждый источник звука в сцене, содержащий ссылку на аудиоклип с установленным флагом **Preload Audio Data** (Предварительная загрузка аудиоданных), потребляет определенный объем памяти для хранения аудиоклипа (сжатого, несжатого или буферизированного, в зависимости от значения параметра **Load Type** (Тип загрузки)) на протяжении всего времени существования сцены. Как исключение, если два или более источников звука содержат ссылки на один и тот же аудиоклип, дополнительная память не расходуется, потому что все они ссылаются на одну и ту же область в памяти и читают данные из нее по мере необходимости.

Аудиоклипы в Unity являются неуправляемыми ресурсами, а это значит, что присваивание ссылкам на них значения `null` не приводит к освобождению памяти. В Unity предполагается, что эти ресурсы должны загружаться и освобождаться вручную, по мере необходимости. Длительное хранение файлов в памяти разумно для часто используемых звуковых эффектов, поскольку загрузка файла при каждой попытке воспроизвести его вызывает расходование ресурсов центрального процессора.

Однако если часто используемые звуковые эффекты занимают слишком много памяти, в целях ее экономии придется сделать непростой выбор между снижением их качества или полным удалением из памяти. Полное удаление звуковых эффектов – не обязательно худший вариант. Достичь нужного уникального звукового эффекта можно также путем повторного использования существующих звуковых эффектов в сочетании со специальными эффектами и фильтрами.

С другой стороны, хранение в памяти редко используемых звуковых эффектов в течение всего времени существования сцены является существенной проблемой. Звуковые эффекты, воспроизведящиеся лишь один раз, например клип диалога, нет смысла держать в памяти ради однократного применения. Создание источников звука с назначенными им аудиоклипами (через свойство `AudioClip`) ведет к появлению ссылок и является причиной чрезмерного потребления памяти, при том что они будут использованы один раз за все время игры.

Решить проблему можно при помощи методов `Resources.Load()` и `Resources.UnloadAsset()`, загружая аудиоданные только на время воспроизведения, а затем немедленного освобождая память. В следующем примере представлен простейший способ реализации такого приема (отслеживается только один источник звука и один аудиоклип) в виде класса `SingletonAsComponent`, который служит лишь для знакомства с идеей создания подобной системы:

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

public class AudioSystem : SingletonAsComponent<AudioSystem> {

    [SerializeField] AudioSource _source;

    AudioClip _loadedResource;

    public static AudioSystem Instance {
        get { return ((AudioSystem)_Instance); }
        set { _Instance = value; }
    }

    public void PlaySound(string resourceName) {
        _loadedResource = Resources.Load (resourceName) as AudioClip;
        _source.PlayOneShot (_loadedResource);
    }

    void Update() {
        if (!_source.isPlaying && _loadedResource != null) {
            Resources.UnloadAsset(_loadedResource);
            _loadedResource = null;
        }
    }
}
```

Этот класс можно протестировать с помощью следующего компонента, загружающего аудиофайл в ответ на нажатие клавиши *A*:

```
public class AudioSystemTest : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.A)) {
            AudioSystem.Instance.PlaySound("TestSound");
        }
    }
}
```

Обратите внимание: чтобы движок Unity смог найти аудиофайл `TestSound`, его следует поместить в папку `Resources`.

При таком подходе память для аудиоклипа выделяется только на время его воспроизведения, а затем, по завершении воспроизведения, немедленно освобождается. В магазине Asset Store Unity предлагается несколько аудиоинструментов с аналогичными возможностями, но вы без труда сможете доработать класс `AudioSystem` для обработки нескольких источников звука и нескольких аудиоклипов.

Установка флага Force to Mono для 3D-звуков

Установка флага **Force to Mono** (Принудительное моновоспроизведение) для стереоаудиофайла приведет к объединению двух каналов в один и сэкономит 50 процентов пространства, занимаемого файлом на диске и в памяти. Установка этого флага обычно не является хорошей идеей для некоторых звуковых 2D-эффектов, где стереоэффект часто используется для создания определенного восприятия аудио. Но при установке этого флага для звуковых 3D-эффектов, когда отсутствует необходимость в стереоисточнике, и для 2D-звуков, когда не нужен стереоэффект, можно сэкономить достаточно много места.

Понижение частоты дискретизации

Понижение частоты дискретизации импортированных аудиофайлов уменьшит их размеры и объем занимаемой памяти во время выполнения. Это можно сделать с помощью параметра **Sample Rate** (Частота дискретизации) и свойств `SampleRate` объекта аудиоклипа. Некоторые файлы требуют высокой частоты дискретизации, например аудиофайлы со звуками высокого тона или с современной музыкой. Однако в большинстве случаев снижением частоты дискретизации можно существенно уменьшить размер файла без заметного ухудшения качества. Частота `22 050` Гц обычно хорошо подходит для человеческой речи и классической музыки. Для некоторых звуковых

эффектов можно использовать и более низкие значения частоты дискретизации. Однако на каждый звуковой эффект этот параметр влияет по-своему, поэтому имеет смысл сначала поэкспериментировать, прежде чем принимать решение о выборе частоты дискретизации.

Выбор формата кодирования

Если приложение не сильно ограничено объемом памяти и дискового пространства, сократить потребление ресурсов центрального процессора можно с использованием формата WAV, требующего меньше зарат на декодирование аудиоданных при воспроизведении. С другой стороны, при невысоких требованиях к вычислительным ресурсам можно сэкономить пространство, использовав кодирование со сжатием.

Потоковая передача

Потоковую передачу файлов с диска следует применять только к большим файлам, воспроизводимым поодиночке, поскольку это требует обращений к жесткому диску – одного из самых медленных способов передачи данных. Многоуровневые или переходные музыкальные клипы могут «заедать» при воспроизведении этим методом, для них лучше подойдет метод на основе `Resources.Load()`. Также следует избегать потоковой передачи более чем одного файла одновременно, поскольку это приведет к росту промахов дискового кэша и, как следствие, к паузам в игровом процессе.

Применение фильтров через микшер для уменьшения дублирования

Фильтры – это дополнительные компоненты, присоединяемые к источнику звука для модификации воспроизводимого звука. Применение каждого фильтра влечет определенные затраты памяти и ресурсов центрального процессора, поэтому присоединение фильтров ко множеству источников звука в сцене может привести к катастрофическим последствиям. Предпочтительнее с помощью утилиты аудиомикшера Unity создавать общие шаблоны, на которые может ссылаться несколько источников звука, и таким способом уменьшить потребление памяти.

Более подробную информацию можно найти в официальном руководстве по адресу: https://unity3d.com/learn/tutorials/modules/beginner/5-pre-order-beta/audiomixer-and-audiomixer-groups.

Правильное использование свойства «WWW.audioClip»

Для потоковой передачи игрового контента через Интернет можно использовать класс `WWW`. Но при каждом обращении к свойству `audioClip` объект класса `WWW` выделяет новые ресурсы для аудиоклипа, аналогично другим методам получения ресурсов. Этот ресурс необходимо освободить вызовом метода `Resources.UnloadAsset()`, когда он станет не нужен.

Очистка ссылки (присваивание значения `null`) не приведет к автоматическому освобождению выделенной памяти. Следовательно, нужно получить аудиоклип из свойства `audioClip`, извлечь ссылку на ресурс и с этого момента использовать только ее, а затем освободить ссылку, когда аудиоклип станет больше не нужен.

Файлы аудиомодулей для фоновой музыки

Файлы аудиомодулей, иначе называемые трекерными модулями, – прекрасное средство экономии памяти без заметной потери качества. В Unity поддерживаются файлы с расширениями `.it`, `.s3m`, `.xm` и `.mod`. В отличие от обычных аудиофайлов в формате PCM, которые читаются как потоки битов данных, декодируемые во время выполнения, трекерные модули содержат множество небольших высококачественных образцов звука в формате PCM и определение полной звуковой дорожки в виде, схожем с нотной записью музыки, описывающем, когда, где, как громко, с каким тактом и с применением каких специальных эффектов должен воспроизводиться каждый из образцов. Этот подход значительно уменьшает размер файла без потери качества. Поэтому если есть возможность использовать версии музыкальных файлов в виде трекерных модулей, ее не нужно упускать.

Файлы текстур

Термины «текстура» и «спрайт» часто подменяются в разработке игр, поэтому считаю важным уточнить, что в Unity 3D под термином **текстура** подразумевается простой файл изображения, то есть список данных о *цветах*, указывающий интерпретирующей программе, какой цвет придать каждому пикселю изображения. **Спрайт**, напротив, – это 2-мерный эквивалент меша, то есть отдельный прямоугольник на плоскости перед текущей камерой. Имеются также **листы спрайтов** – коллекции отдельных изображений, хранящихся в файле текстуры. Листы спрайтов обычно используются для хранения анимации 2D-персонажа. Выделить отдельные текстуры с кадрами ани-

мации из этих файлов можно при помощи инструмента пакетной обработки спрайтов Unity.

А теперь оставим в стороне всю эту путаницу в терминах и поговорим о простых текстурах – файлах изображений, созданных с помощью графических редакторов, таких как Adobe Photoshop и Gimp. Во время выполнения эти файлы загружаются в память, передаются графическому процессору и выводятся с помощью шейдера на целевом объекте системой визуализации.

Форматы сжатия

По аналогии с аудиофайлами, Unity поддерживает различные методы сжатия файлов текстур для увеличения эффективности их хранения. Импортируя файл текстуры, можно определить несколько параметров (рис. 4.1). Первый параметр: **Texture Type** (Тип текстуры). Он влияет не на сам файл изображения, а на порядок его интерпретации, обработки и сжатия во время сборки исполняемого файла.

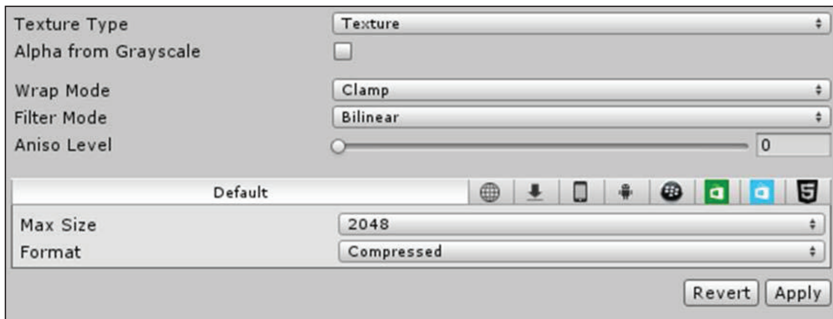


Рис. 4.1 ❖ Параметры импортирования текстур

Для большинства типов текстур в Unity поддерживаются только три типа сжатия: **Compressed** (Сжатый), **16-bit** (16-битная палитра) и **True Color** (24-битный цвет). И только при выборе варианта **Advanced** (Расширенный) в параметре **Texture Type** (Тип текстуры) появляется возможность более тонкой настройки (рис. 4.2). Выбор этого типа текстуры позволяет получить более полный контроль над интерпретацией файла текстуры.

Эта информация еще пригодится ниже, так как, используя ее, мы сможем повысить производительность, организовав особое представление файла текстуры, которая в основном невидима.

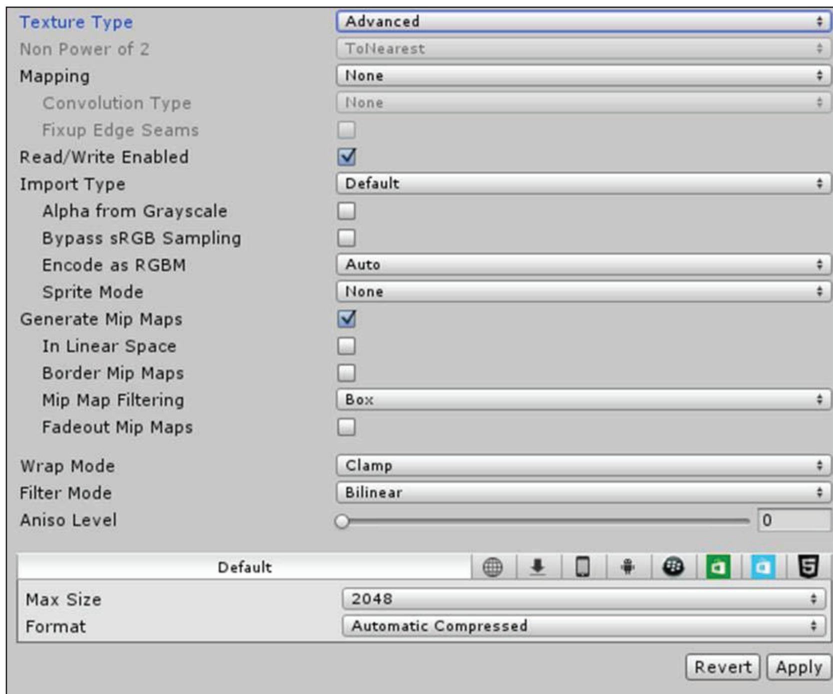


Рис. 4.2 ❖ Дополнительные параметры управления интерпретацией текстуры

Выбор форматов сжатия для типа **Advanced** (Расширенный) шире и разнообразней, причем он одинаков для версий 4 и 5 Unity. Некоторые форматы поддерживают альфа-канал, другие – нет. Это ограничивает выбор, если требуется применить к текстуре альфа-прозрачность (или использовать четвертое значение типа `float` для карты высот!).

Кроме того, разные форматы обусловливают разные уровни производительности на этапе инициализации сцены, когда выполняются распаковка и передача в графический процессор. Причем производительность зависит также от целевой платформы и выбранного формата. Тип **Advanced** (Расширенный) поддерживает вариант **Automatic Compressed** (Автоматический выбор сжатия), когда Unity пытается автоматически подобрать лучший вариант для конкретной платформы и устройства. Этот вариант можно использовать, когда нет полной уверенности в выборе оптимального формата для игры или для

тестирования различных форматов, чтобы определить лучший для целевого устройства.

Обратите внимание, что окно предварительного просмотра в нижней части представления **Inspector** (Инспектор) содержит полезные статистические данные, помогающие определить эффективность выбранного метода сжатия.

Улучшение производительности обработки текстур

Рассмотрим, как с помощью параметров настройки можно увеличить производительность обработки файлов текстур в зависимости от ситуации и содержимого импортируемых файлов. В каждом конкретном случае будет описываться выбор значений параметров и проанализирован эффект их применения: влияние на потребление памяти и ресурсов центрального процессора, увеличение или уменьшение качества текстур и при каких условиях их можно применять.

Кроме того, поскольку в редакции Personal Unity 5 стали доступными несколько функций, прежде поддерживавшихся только в редакции Pro, некоторые пользователи могут не знать о существовании дополнительных методов повышения производительности обработки текстур. Некоторые из этих методов будут описаны в конце данного раздела.

Уменьшение размеров файлов текстур

Чем больше размер файла текстуры, тем большей пропускной способностью должна обладать память графического процессора для ее вывода. Если общий объем данных, передаваемых в единицу времени, превысит пропускную способность памяти графической карты, образуется узкое место, поскольку графический процессор требует загрузки всех текстур до перехода к следующему этапу отображения. Чем меньше текстура, тем проще пропустить ее через графический конвейер, поэтому следует найти золотую середину между качеством и производительностью.

Чтобы убедиться, что узким местом является пропускная способность памяти, попробуйте просто уменьшить разрешение самых сложных и больших файлов текстур в игре и перезапустить сцену. Если после этого частота кадров заметно возрастет, значит, проблема действительно кроется в недостаточной пропускной способности при передаче текстур. Если частота кадров не улучшится или улучшится незначительно, значит, либо пропускная способность памяти еще не была превышена, либо в конвейере визуализации имеются другие узкие места, не позволяющие проявиться улучшениям.

Использование mip-текстур

Не имеет смысла отображать мелкие отдаленные объекты, такие как скалы или деревья, с помощью излишне детализированных текстур, если игрок не сможет рассмотреть детали или если потеря производительности не оправдывает воспроизведения подробностей. Mip-текстуры, призванные для решения таких проблем (а также проблем сглаживания, преследующих видеоигры), – это предварительно подготовленные наборы текстур с разным разрешением. Во время выполнения графический процессор выбирает вариант mip-текстуры, основываясь на размере поверхности, отображаемой в перспективной проекции (по сути, на основе соотношения тексель к пикселю при отображении объекта), а затем масштабирует mip-текстуру.

При установке флага **Generate Mip Maps** (Генерировать mip-текстуры), Unity автоматически генерирует текстуры с низким разрешением. Альтернативные текстуры создаются с использованием методов высококачественной интерполяции и фильтрации в редакторе, а не во время выполнения.

На рис. 4.3 показано, как изображение размером 1024×1024 будет разложено на несколько копий с более низким разрешением:

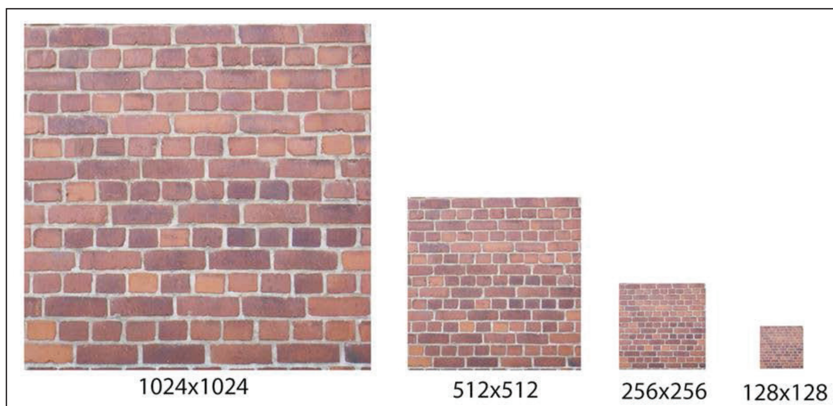


Рис. 4.3 ❖ Mip-текстура

Недостатком mip-текстур является увеличение размера файла и времени их загрузки. В конечном итоге включение mip-текстурирования увеличивает размер файла текстуры приблизительно на 33 процента. Существуют ситуации, когда применение mip-текстурирования

ничего не дает, поэтому проанализируем несколько файлов текстур и выясним, когда применение mip-текстурирования дает положительный эффект.

Применение mip-текстурирования оправдано, только когда дело касается текстур, которые должны отображаться на различных расстояниях от камеры. Для текстур, которые всегда отображаются на одном и том же расстоянии от основной камеры, альтернативные mip-текстуры никогда не используются и просто зря занимают место. В этом случае следует отключить функцию mip-текстурирования, сбросив флаг **Generate Mip Maps** (Генерировать mip-текстуры).

Если используется лишь одна из альтернативных mip-текстур, следует отключить mip-текстурирование и уменьшить разрешение исходного файла текстуры.

Кроме того, кандидатами на отключение функции mip-текстурирования являются:

- практически все файлы текстур в 2D-играх;
- текстуры пользовательского интерфейса (GUI);
- текстуры для мешей, спрайтов и эффектов частиц, которые всегда отображаются рядом с камерой, их примером является сам персонаж игрока, все носимые им объекты и все эффекты частиц, которые всегда окружают игрока.

Управление разрешением текстур извне

Движок Unity предпринимает массу усилий, чтобы упростить включение в проект файлов, созданных с помощью внешних инструментов, например файлов PSD и TIFF, которые часто являются большими изображениями, разделенными на несколько слоев. Unity автоматически создает текстуру из содержимого такого файла, готовую к применению в любом месте движка, и обеспечивает удобную поддержку с помощью системы управления версиями, при этом копия в Unity автоматически обновляется, когда художник вносит изменения в изображение.

Проблема заключается в том, что инструменты автоматического создания и сжатия текстур, имеющиеся в Unity, не являются столь же надежными и эффективными, как инструменты создания исходных файлов, такие как Adobe Photoshop или Gimp. Unity может вносить **артефакты** в автоматически создаваемые текстуры, что нередко побуждает нас импортировать файлы изображений с более высоким разрешением, чем это необходимо. Внешние инструменты порой позволяют снизить разрешение изображения с более высоким каче-

ством и даже добиться приемлемого уровня качества с более низким разрешением и сэкономить дисковое пространство и память.

Мы можем либо отказаться от использования файлов PSD и TIFF в проектах Unity (хранить их в другом месте и импортировать в Unity-версии с уменьшенным разрешением), либо просто периодически проводить тестирование, чтобы убедиться, что не зря увеличен размер файла, затрачивается больше памяти и пропускной способности памяти графического процессора при использовании файлов с большим разрешением, чем необходимо. Это будет стоить определенного удобства управления файлами проекта, но позволит сократить затраты при обработке некоторых из текстур, если потратить время на сравнение производительности при использовании версий текстур с разным разрешением.

Настройка уровня анизотропной фильтрации

Анизотропная фильтрация улучшает качество текстур при их просмотре под острыми углами. На рис. 4.4 показан классический пример сравнения линий разметки на дороге с применением анизотропной фильтрации и без. Без анизотропной фильтрации линии разметки выглядят более размытыми и искаженными с удалением от камеры, тогда как с анизотропной фильтрацией они смотрятся более четкими и ясными.

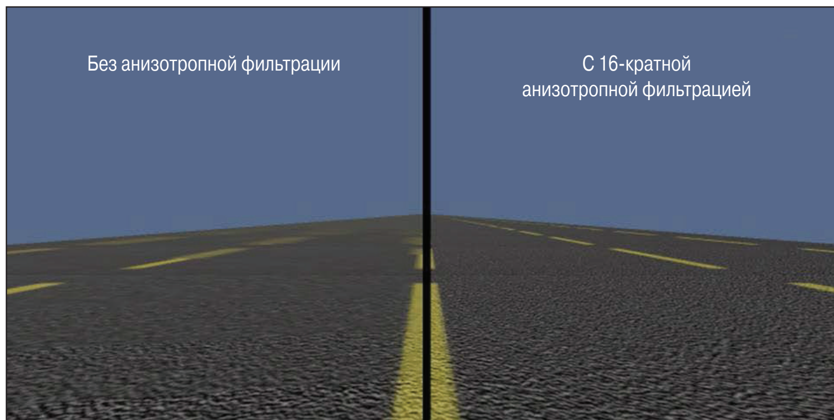


Рис. 4.4 ❖ Пример применения анизотропной фильтрации

Уровень анизотропной фильтрации можно изменить вручную, с помощью параметра **Aniso Level** (Уровень анизотропной фильтра-

ции), кроме того, имеется возможность включить и выключить анизотропную фильтрацию глобально, с помощью параметра **Anisotropic Textures** (Анизотропные текстуры) в разделе **Quality Settings** (Параметры качества).

Подобно mip-текстурированию, анизотропная фильтрация увеличивает потребление ресурсов, и ее применение не всегда оправдано. Если в сцене имеются текстуры, которые никогда не будут отображаться под острым углом (например, спрайты на удаленном фоне или текстура эффекта частиц), для них следует отключить анизотропную фильтрацию для экономии вычислительных ресурсов. Можно также произвести регулировку уровня анизотропной фильтрации для текстур, чтобы определить золотую середину между качеством и производительностью.

Текстурные атласы

Текстурные атласы – это технология объединения множества небольших изолированных текстур в одну большую текстуру для минимизации количества материалов и, как следствие, обращений к системе визуализации при применении динамической пакетной обработки. Концептуально этот метод очень похож на способы минимизации количества используемых материалов, рассмотренных в *главе 3 «Преимущества пакетной обработки»*.

Каждый уникальный материал требует дополнительного обращения к системе визуализации, но каждый материал поддерживает только одну первичную текстуру (это не касается карт нормалей, эмиссий и других вторичных текстур). Объединив все текстуры в одну огромную текстуру, можно уменьшить количество обращений к системе визуализации для отображения объектов с общей текстурой, как показано на рис. 4.5.

Выгоды при этом очевидны: сокращение числа обращений к системе визуализации приведет к уменьшению нагрузки на центральный процессор и повышению частоты кадров, если приложение ограничено возможностями центрального процессора (или просто высвободит центральный процессор для решения других задач). Это произойдет без потери качества и существенного увеличения потребления памяти. Все, что следует сделать, – это применить систему динамической пакетной обработки Unity для уменьшения количества обращений к системе визуализации. Обратите внимание, что использование текстурных атласов не влияет на пропускную способность памяти, поскольку количество передаваемых данных не изменяется. Они просто будут собраны вместе в один большой файл текстуры.

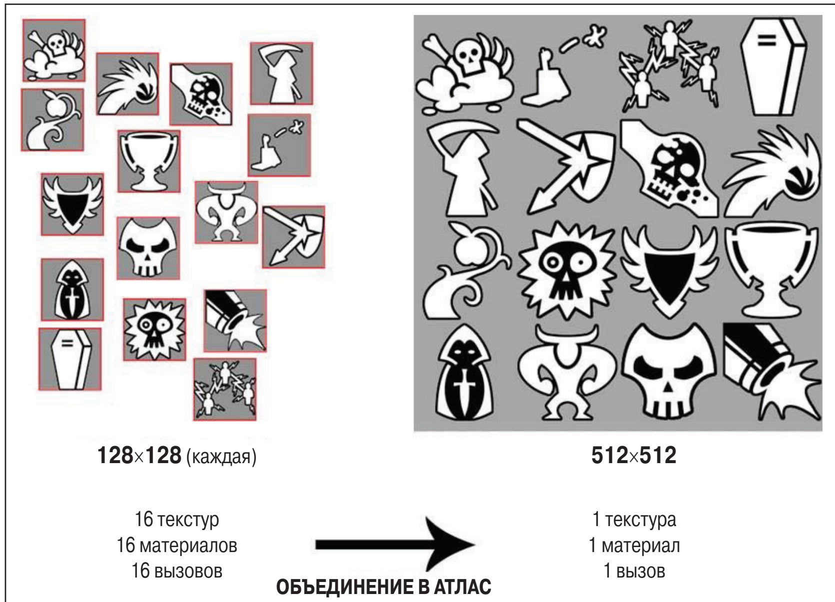


Рис. 4.5 ❖ Эффект применения текстурного атласа

Текстурные атласы применяются, только когда все текстуры используют один и тот же шейдер. Если к некоторым текстурам требуется применять уникальные графические эффекты, их следует изолировать, определив для них собственные материалы, или включить в атлас вместе с другими текстурами, которые используют тот же шейдер.

Текстурные атласы часто применяются для отображения элементов пользовательского интерфейса и в играх с большим количеством 2D-графики и практически необходимы при разработке мобильных игр в Unity, поскольку вызовы визуализации обычно являются узким местом на этих платформах. Однако не хотелось бы создавать файлы атласов вручную. Было бы гораздо проще, имея возможность, продолжать редактировать текстуры отдельно и автоматизировать задачу их объединения в атлас.

В магазине Asset Store Unity доступно большое количество инструментов автоматизированного создания атласов текстур. В Интернете можно найти автономные программы, способные справиться с этой задачей, и сам движок Unity, начиная с версии 4.6, имеет встроенный

инструмент упаковки спрайтов Sprite Packer, который можно легко настроить для упаковки текстур различными способами.

Более подробно об этой полезной функции можно узнать из документации Unity по адресу: http://docs.Unity3D.com/Manual/SpritePacker.HTML.

В любом случае, используйте существующие решения создания атласов, чтобы не изобретать велосипед.

Имейте в виду: чтобы инструмент Sprite Packer мог распознать и упаковать текстуры, они должны иметь тип Sprite (Спрайт).

Текстурные атласы следует применять не только к 2D-графике и элементам пользовательского интерфейса. Этот метод можно использовать и с 3D-мешами, если имеется множество текстур с низким разрешением. 3D-игры с низким разрешением текстур или в художественном стиле плоскостной затушевки с малым количеством полигонов являются идеальными кандидатами для применения атласов.

Однако, так как динамическая пакетная обработка применяется только к мешам без анимации (то есть к компонентам `MeshRenderer`, но не `SkinnedMeshRenderer`), нет никаких оснований объединять в атлас текстуры анимированных персонажей. Поскольку персонажи анимированы, графическому процессору необходимо обработать положение всех костей объекта при текущем состоянии анимации. Это означает, что для каждого персонажа необходимо произвести отдельные расчеты, что повлечет дополнительные обращения к системе визуализации независимо от общности применяемых материалов.

Следовательно, полезность объединения текстур для анимированных персонажей заключается только в удобстве и экономии пространства. Например, в играх с применением художественного стиля плоскостной затушевки с малым количеством полигонов и общей палитрой цвета можно добиться значительной экономии памяти, задействовав одну текстуру для игрового мира, объектов и персонажей.

Главным недостатком атласов является увеличение времени разработки и расходов на нее. Применение текстурных атласов требует много усилий, и еще больше, чтобы оценить, окупится ли затраченный труд. Кроме того, следует остерегаться файлов атласов, слишком больших для целевой платформы.

Некоторые устройства (особенно мобильные) имеют ограниченный размер низкоуровневой кэш-памяти графического процессора. Если атлас слишком велик, его нужно разделить на меньшие текс-

туры, размеры которых соответствуют объему памяти целевого устройства. Если при отображении в каждом вызове потребуется выводить текстуры из различных частей атласа, это не только приведет к промахам кэша, но и уменьшит пропускную способность памяти, поскольку текстуры постоянно будут браться из видеопамяти и низкоуровневого кэша.

Эта проблема не стояла бы, если бы атлас оставался разбитым на отдельные текстуры. Это не избавит от необходимости смены текстур, но уменьшит объем передаваемых данных за счет дополнительных вызовов системы визуализации. Лучшим вариантом решения проблемы будет уменьшение разрешения атласа или создание атласов меньшего размера для улучшения контроля над их динамическим обновлением.

Итак, текстурные атласы не являются идеальным решением. Если их применение не приведет к значительному увеличению производительности, не следует тратить слишком много времени на их реализацию.

Вообще, текстурные атласы следует применять к высококачественным мобильным играм среднего масштаба с самого начала проекта, следя за тем, чтобы размер текстур не превышал ограничений целевой платформы и устройства. С другой стороны, простейшие мобильные игры обычно вообще не нуждаются в использовании атласов.

Для настольных игр с высококачественными текстурами атласы следует применять, только если количество обращений к системе визуализации превышает все разумные возможности аппаратного обеспечения из-за обработки множества текстур с высоким разрешением для обеспечения максимального качества. В настольных играх с низкокачественными текстурами можно вообще отказаться от атласов, так как вызовы системы визуализации не являются в них узким местом.

Конечно, независимо от вида продукта, если узким местом является центральный процессор и применение альтернативных методов ни к чему не привело, грамотное использование атласов может значительно уменьшить количество обращений к системе визуализации.

Настройка сжатия текстур неквадратной формы

Не рекомендуется импортировать в приложение текстуры неквадратной формы и/или с размерами, не равными степени числа 2. Использование текстур неквадратной формы или с размерами, не равными степени числа 2, увеличит затраты, связанные с корректи-

ровкой их размеров. Unity будет автоматически подстраивать текстуры и добавлять к ним дополнительное пустое пространство, чтобы придать требуемую форму и размер, что приведет к дополнительным затратам памяти и передаче графическому процессору, по существу, бесполезных данных.

Поэтому в первую очередь рекомендуется вообще избегать использования текстур неквадратной формы и/или с размерами, не равными одной из степеней числа 2. Если изображения можно разместить внутри квадратной текстуры со стороной, равной степени числа 2 без заметной потери качества из-за сжатия/растяжения, следует это сделать для удобства их обработки центральным и графическим процессорами.

Однако если вы все еще хотите использовать текстуры неквадратной формы, используйте прием, позволяющий получить высокое качество без дополнительных затрат памяти. Поскольку неквадратные текстуры сжимаются с помощью одного из алгоритмов, можно увеличить скорость передачи битов (а следовательно, и качество), выбрав нужный формат сжатия для таких текстур, и таким способом достичь размера, идентичного размеру импортированного файла, и тех же накладных расходов при выполнении. От нас потребуется только потратить время на поиск таких текстур и на проверку различных алгоритмов сжатия, чтобы отобрать версии с наилучшим качеством.

Разреженные текстуры

Разреженные текстуры, иначе называемые мега-текстурами или плиточными текстурами, дают эффективную возможность подкачки данных текстуры с диска во время выполнения. Для сравнения, если предположить, что процессор выполнит операцию за секунды, дисковые операции займут несколько дней. Поэтому во время игры следует всячески избегать обращений к жесткому диску, поскольку этот весьма рискованный подход легко может привести к остановке приложения.

Но применение разреженного текстурирования позволяет нарушить это правило, поддерживая интересные методы сохранения производительности. Разреженное текстурирование заключается в объединении множества текстур в один огромный файл, слишком большой для загрузки в графическую память целиком. В этом отношении разреженная текстура похожа на атлас, кроме невероятно больших размеров (например, 32 768×32 768) и высокого качества (32 бита на пиксель). Идея заключается в том, чтобы сэкономить

значительный объем памяти и пропускную способность за счет динамического выделения небольших подразделов текстуры вручную и предварительной загрузки их с диска до того, как они потребуются в игре. Важной особенностью этой технологии является размер файла (файл с указанным выше разрешением будет занимать 4 Гбайт дискового пространства!). Также придется провести большую работу, связанную с подготовкой сцены.

Игровой мир должен создаваться так, чтобы минимизировать количество переключений текстур. Чтобы исключить подергивания и «разрывы», подразделы текстуры должны извлекаться с диска заранее, незаметно для игрока. Это обеспечивается структурой самого файла текстуры, путем сохранения элементов для данной сцены в одной области текстуры, и структурой сцены, позволяющей подгружать новый раздел текстуры в ключевые моменты игрового процесса. При тщательной разработке и со вниманием к деталям применение разреженного текстурирования поможет получить впечатляющее качество сцены и не менее впечатляющую экономию памяти.

Разреженное текстурирование в версии Unity 4 было доступно только в редакции Pro, но в Unity 5 оно доступно и в редакции Personal. Разреженное текстурирование требует специализированного аппаратного оборудования и поддержки платформой, поэтому данный подход доступен не для всех игр. Это весьма специализированная и редко применяемая в игровой индустрии технология, с чем связано отсутствие ее подробной документации. Документация Unity содержит мало сведений о разреженном текстурировании, но в ней можно найти пример сцены, демонстрирующий эффективность приема: http://docs.unity3d.com/ Manual/SparseTextures.html.

Опытные разработчики, способные разобраться во всем самостоятельно, могут попробовать оценить выгоды применения разреженного текстурирования в своих проектах и свою готовность внести соответствующие изменения в сцены, необходимые для получения преимуществ этой технологии.

Процедурные материалы

Процедурные материалы, иначе называемые субстанциями, – это средство динамического создания текстур во время выполнения путем объединения малых, высококачественных образцов текстур с помощью пользовательских математических формул. Процедурные материалы предназначены для уменьшения объема приложения за счет увеличения потребления памяти и ресурсов центрального про-

цессора во время инициализации. Подобно разреженному текстурированию, раньше эта функция была доступна только в редакции Pro Unity 4, но теперь поддерживается всеми редакциями Unity 5.

Это дает возможность многим пользователям по-другому взглянуть на материалы, так как процедурные материалы представляют более современный подход к разработке игр. Файлы текстур обычно являются одним из самых крупных потребителей дискового пространства в игровом проекте, а это значит, что большой объем текстур отрицательно сказывается на длительности загрузки игры, что отпугивает игроков, пожелавших попробовать игру (даже если она бесплатна!). Процедурные материалы позволяют за счет определенного увеличения времени инициализации и дополнительных вычислительных затрат ускорить процесс загрузки игры. Это очень важно для грядущих поколений мобильных игр, которые пытаются выдержать конкуренцию за счет увеличения качества графики.

В документации Unity процедурные материалы освещены полнее, чем разреженные текстуры, поэтому, чтобы разобраться в работе субстанций и их преимуществах, обращайтесь к документации: http://docs.unity3d.com/Manual/ProceduralMaterials.html.

Файлы мешей и анимаций

Наконец, рассмотрим файлы мешей и анимаций. По сути, эти файлы являются большими массивами данных о вершинах и костях. Существует целый ряд методов, помогающих уменьшить их размеры, сохраняя сходство, если не идентичность, внешнего вида объектов. Также существует несколько способов снижения затрат на отображение больших групп объектов при помощи пакетной обработки. Рассмотрим ряд методов повышения производительности применительно к таким файлам.

Уменьшение количества полигонов

Это наиболее очевидный способ улучшения производительности, но не стоит им пренебрегать. В самом деле, поскольку пакетную обработку нельзя применить к компонентам `SkinnedMeshRenderer`, это единственный доступный способ снижения затрат вычислительных ресурсов центрального и графического процессоров для анимированных объектов.

Сокращение количества полигонов – простой и очевидный способ, обеспечивающий экономию ресурсов центрального процессора и па-

мяти за счет затрат времени художников на чистку мешей. В настоящее время значительную часть деталей объекта можно реализовать исключительно с помощью текстур и сложных шейдеров, поэтому уменьшение числа вершин в современных мешах останется незаметным для подавляющего большинства пользователей.

Настройка сжатия мешей

Настройка сжатия значения для параметра **Mesh Compression** (Сжатие меша), управляющего сжатием импортируемых мешей: **Off** (Отключено), **Low** (Низкое), **Medium** (Среднее) и **High** (Высокое). С увеличением степени сжатия Unity отбрасывает все больше и больше верщин из меша, которые сочтет ненужными, чтобы сократить общий размер файла. По сути, сжатие представляет собой автоматизацию процесса сокращения количества полигонов художником.

Но автоматизированная оптимизация меша – очень сложная задача для решения математическими методами. Даже лучшие алгоритмы, как правило, генерируют большое количество артефактов и нарушений. Изменение значения параметра **Mesh Compression** можно рассматривать только как быстрое решение задачи сокращения количества полигонов, но результат никогда не будет сопоставим с тем, чего может добиться художник.

Инструменты 3D-моделирования обычно предоставляют свои средства автоматизированной оптимизации мешей. Попробуйте воспользоваться ими для оптимизации мешей перед импортированием в Unity.

Правильное использование флага доступности для чтения и записи

Установка флага **Read-Write Enabled** (Доступен для чтения и записи) позволяет изменять меш во время выполнения из сценариев или автоматически из Unity. В этом случае движок будет хранить исходные данные мешей в памяти до их копирования или изменения. Отключение этого параметра позволит Unity освободить память, занимаемую исходными данными, как только будет завершено их использование.

Если на протяжении игры используется только однородно масштабированная версия меша, отключение этого параметра поможет сэкономить память, так как отпадает необходимость хранить исходные данные для создания дубликатов меша оригинального размера (кстати, точно так же Unity объединяет объекты при динамической пакет-

ной обработке). То есть Unity может раньше избавиться от ненужных данных, которые не понадобятся до следующего запуска приложения.

Но если во время выполнения меш часто дублируется в различных масштабах, Unity должен хранить его в памяти, чтобы быстро вычислить новый меш. Поэтому флаг **Read-Write Enabled** (Доступен для чтения и записи) должен быть установлен. Отключение флага потребует от Unity не только повторно загрузить меш, но и выполнить повторное масштабирование дубликата, что вызовет падение производительности.

Во время инициализации Unity пытается определить правильное поведение в зависимости от установки данного флага. Но когда меши создаются и масштабируются динамически, следует самостоятельно установить этот флаг. Это позволит увеличить скорость создания экземпляров объектов, но также увеличит потребление памяти, так как оригинальные меши будут храниться, пока не понадобятся.

Примечательно, что этот же подход применяется при использовании параметра **Generate Colliders** (Генерировать коллайдеры).

Импорт/расчет только необходимого

Это выглядит как еще один очевидный совет, но меши содержат не только координаты вершин. Ненужными могут оказаться **нормали** и **касательные** в меше, которые не используются шейдерами или могут быть автоматически сгенерированы (рис. 4.6), особенно при очень малом **угле сглаживания**. В таких случаях для каждой вершины требуется несколько векторов нормалей для создания фасеточного стиля плоскостной затушёвки.

Внимательно проверьте параметры импортирования меша, оцените размер полученного файла, отображение меша в игре. Поэкспериментируйте с настройками, чтобы увидеть, что можно сделать для очистки от нежелательных и ненужных данных.

Встраиваемые анимации

В некоторых случаях встраиваемые (запекаемые) анимации помогают уменьшить размер файла и занимаемый объем памяти, чем смешанные или наружные анимации. Это зависит от используемого инструмента 3D-моделирования и анимации и общего количества вершин в меше. Встраиваемые анимации сохраняют позиции каждой вершины в каждом кадре, и если количество полигонов достаточно мало, можно наблюдать значительное сохранение ресурсов при внесении достаточно простых изменений.

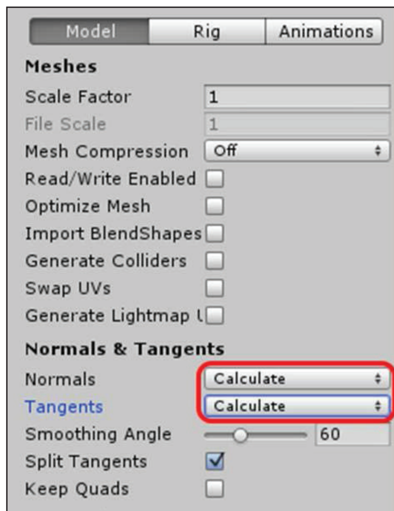


Рис. 4.6 ❖ Настройка автоматического вычисления нормалей и касательных

Кроме того, встраиваемый образец обычно можно настроить в экспортирующем приложении. В порядке упрощенной оценки проведите тестирование на разных частотах дискретизации, чтобы найти оптимальное расположение ключевых моментов анимации.

Оптимизация мешей движком Unity

Установка флага **Optimize Meshes** (Оптимизация мешей) в настройках импорта меша обеспечит реорганизацию вершин для ускорения их чтения и, в некоторых случаях, регенерацию низкоуровневого отображения (вплоть до уровня отдельных точек, штрихов и полос) для оптимизации скорости отображения. Проще говоря, этот флаг должен устанавливаться практически во всех ситуациях, поскольку весьма мало причин запрещать Unity производить такие корректировки. Сбрасывать его стоит только в том случае, если при профилировании точно выяснится, что его установка каким-то образом приводит к снижению производительности.

Если меш генерируется процедурно, для запуска этого процесса движком Unity можно воспользоваться методом `Optimize()` компонента фильтра меша. Этот процесс займет некоторое время, поэтому его следует запускать во время инициализации или в другие удобные для приостановки моменты.

Объединение мешей

Объединение мешей позволяет уменьшить количество обращений к системе визуализации в случаях, если эти меши слишком велики для динамической пакетной обработки и не участвуют в статической пакетной обработке. Такое объединение, по существу, эквивалентно статической пакетной обработке, но выполняется вручную, поэтому вряд ли стоит тратить время на то, что может сделать статическая пакетная обработка автоматически.

Но в редакции Free Unity 4, где статическое пакетная обработка недоступна, и при необходимости перемещать меши по сцене такой подход станет хорошим решением для уменьшения количества вызовов системы визуализации. Однако имейте в виду, что он несет те же риски, что и статическая пакетная обработка. Если хотя бы одна вершина меша видима в сцене, весь объект будет отображаться как единое целое. Это может привести к большим напрасным затратам, если большую часть времени видна только часть меша.

Данный подход имеет еще один недостаток – он создает совершенно новый файл ресурсов мешей, который требуется разместить в сцене. То есть любые изменения в оригинальных мешах не отразятся на объединенном. Это требует выполнять утомительные действия каждый раз после внесения изменений, поэтому лучше отдать предпочтение статической пакетной обработке.

В Интернете можно найти несколько инструментов объединения файлов мешей для применения в Unity. Поищите их в Asset Store или в Google.

Итоги

Существует масса способов увеличения производительности приложения только с помощью тонкой настройки импортируемых ресурсов. Или, говоря иначе, существует масса способов нанести ущерб производительности приложения бездумным управлением ресурсами.

Практически все способы основаны на компромиссе характеристик производительности и рабочих задач. Поэтому будьте осторожны и выбирайте соответствующие способы для соответствующих проектов по соответствующим причинам. К наихудшим последствиям приводит внесение изменений без понимания, чем это чревато. Конечно, интересно случайно получить большой прирост производительности,

но без понимания, как это работает и какие влечет затраты, можно, не осознавая, создать такие проблемы, что их устранение в дальнейшем потребует затрат массы времени и сил.

На этом завершается знакомство с искусством управления ресурсами. Следующая глава будут посвящена физическим движкам Box2D и PhysX, а также изменениям, которые следует внести в сценарии, сцену и проект для ускорения обработки данных этими компонентами, и что делать, чтобы не нарушить их нормальную работу.

Глава 5

Разгон физического движка

Все рассмотренные до сих пор советы по повышению производительности касались главным образом снижения требований к системе и не затрагивали вопроса частоты кадров. Но на самом фундаментальном уровне под увеличением производительности подразуевается улучшение восприятия пользователями. Каждое падение частоты кадров, каждая поломка и каждое требование, слишком дорогостоящее для целевой системы, в конечном итоге ухудшают качество продукта, и нам остается только попытаться успеть выполнить подстройки и исправить ошибки до выхода продукта.

Конечно, существуют возможности настройки поведения физического движка, помогающие решить все упомянутые проблемы, но физический движок также входит в категорию средств, оказывающих непосредственное влияние на качество игры. Если окажется пропущенным важное для игры событие столкновения (например, игрок провалился сквозь пол) или игра зависнет при расчете сложной ситуации, это значительно снизит качество игры. В результате ломается ощущение погружения в игру и остается лишь подбросить монетку, чтобы решить, сочтет ли пользователь это неудобным, отвратительным или забавным. Если игра специально не ориентирована на жанр физической комедии (популярность которого, кстати, в последние годы значительно возросла после появления таких игр, как QWOP и Goat Simulator), то подобных ситуаций следует всячески избегать.

В некоторых играх физический движок решает значительное количество задач, например отслеживает столкновения с другими объектами или отбрасывает невидимые лучи и составляет список объектов в заданной области. Например, самой сутью платформеров и прочих

активных игр является соответствующая настройка физического движка. Реакции персонажа на ввод пользователя и игрового мира на действия персонажа являются двумя наиболее важными аспектами, делающими игру интерактивной и увлекательной. Другие игры используют физический движок лишь для отслеживания отдельных событий игрового процесса, интересных представлений и прочих притягивающих внимание моментов, но чем эффективнее используется физическая система, тем более впечатляюще выглядит игра.

Таким образом, в этой главе будут рассмотрены не только способы снижения пиковых нагрузок на центральный процессор и потребления памяти, связанных с физической системой Unity, но также методы влияния на поведение физического движка для улучшения качества игрового процесса. Кроме того, поскольку физическая система Unity представляет собой «черный ящик», не допуская возможности ее внутренней отладки, часто очень сложно точно определить причины сбоев, возникших в проекте, для принятия соответствующих мер. Поэтому в данной главе будут представлены советы, касающиеся уменьшения нестабильности физической системы и решения проблемных физических ситуаций.

Внутреннее устройство физического движка

Технически в Unity имеются два физических движка: NVIDIA PhysX для поддержки 3D-физики и открытый движок Box2D для поддержки 2D-физики. Однако их реализация абстрагирована, и с точки зрения прикладного программного интерфейса Unity оба движка работают практически идентично.

В любом случае, хорошее понимание физической системы Unity упростит в дальнейшем внесение улучшений. Итак, начнем с теории физических движков Unity.

Физические движки и время

Физические движки, как правило, работают в предположении, что итерации происходят в фиксированные моменты времени, и оба физических движка Unity действуют именно так. Выполняя вычисления, они основываются на конкретных значениях времени, независимо от того, сколько времени потребовалось для отображения предыдущего кадра. Такой подход называется **обновлением с фиксированным**

интервалом времени. По умолчанию считается, что интервал имеет длительность 20 миллисекунд, или 50 обновлений в секунду.

Использование физическим движком переменной скорости значительно затруднит согласование результатов столкновений и воздействия сил между двумя различными компьютерами. Такие движки, как правило, дают противоречивые результаты в многопользовательских системах и при повторном воспроизведении.

Если между отображением последовательных кадров возникает слишком большая пауза (низкая частота кадров), физическая система обновляется несколько раз перед началом нового отображения. И наоборот, если с момента предыдущего отображения прошло недостаточно времени (высокая частота кадров), итерация может быть пропущена до следующего вызова отображения.

Метод `FixedUpdate()` вызывается в момент, когда физическая система начинает очередной шаг моделирования. Это один из самых важных методов обратного вызова в Unity, который можно определить в сценарии MonoBehaviour и использовать для выполнения вычислений, не зависящих от частоты смены кадров, таких как моделирование искусственного интеллекта (работу которого также облегчает предположение о фиксированности частоты обновлений) или изменение объектов `Rigidbody`.

На рис. 5.1 изображен важный фрагмент диаграммы, определяющий порядок работы Unity:

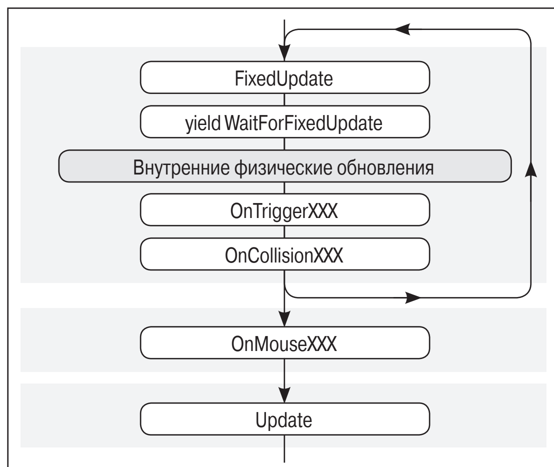


Рис. 5.1

Полную диаграмму, отражающую порядок работы, можно найти по адресу: http://docs.unity3d.com/Manual/ExecutionOrder.html.

Цикл фиксированного обновления

Как видите, метод `FixedUpdate()` вызывается непосредственно перед внутренними операциями физической системы, и эти два действия неразрывно связаны. В начале процесса определяется, прошло ли достаточно времени для начала следующего фиксированного обновления. Результат зависит от времени, прошедшего с момента последнего обновления.

Если прошло достаточно времени, вызываются методы `FixedUpdate()` всех компонентов, а сразу после этого запускаются все сопрограммы, привязанные к фиксированным обновлениям (то есть приостановленные с вызовом `WaitForFixedUpdate()`), затем следуют вызовы обработчиков физических обновлений и триггеров/коллайдеров.

Если с момента последнего фиксированного обновления прошло менее 20 мс, текущее обновление пропускается. В этом случае обработка ввода, игровой логики и отображения должны завершиться до момента, когда Unity проверит необходимость следующего фиксированного обновления, и т. д. Этот процесс постоянно повторяется во время выполнения. При таком подходе фиксированные обновления и физическая система получают более высокий приоритет, чем отображение, что ускоряет физическое моделирование при фиксированной частоте кадров.

Чтобы обеспечить гладкость движения в кадрах, где происходит пропуск обновления, некоторые физические движки (включая движок Unity) экстраполируют состояние от предыдущего к текущему с учетом времени, оставшегося до следующего обновления. Это гарантирует гладкость движений при высокой частоте кадров, хотя обновление выполняется каждые 20 мс.

Максимально допустимая длительность

Важно отметить, что если прошло много времени с момента последнего фиксированного обновления (к примеру, игра на время замерла), физическое и фиксированное обновления будут рассчитываться до момента, пока не «догонят» текущее время. Например, если отображение предыдущего кадра заняло 100 мс (из-за внезапного провала производительности игры до 10 кадров в секунду), физической системе потребуется выполнить пять циклов обновления состояния. При этом перед вызовом метода `Update()` пять раз будет

вызван метод `FixedUpdate()`, так как интервал фиксированных обновлений по умолчанию составляет 20 мс. Если обработка этих пяти обновлений займет 20 мс и более, необходимо будет вызвать обновление шестой раз!

Следовательно, во время тяжелых вычислений физический движок может не успеть завершить обновление за 20 мс, что добавит еще на 20 мс работы, и так до бесконечности, из-за чего не станет невозможно вывести очередной кадр (часто это называют «спиралью смерти»). Чтобы предотвратить блокировку игры физическим движком, определяется максимальное время, которое ему разрешено выполнять обработку между отображениями. Этот порог называют **максимальной допустимой длительностью**, и если длительность текущего фиксированного обновления оказывается больше указанного значения, оно будет просто остановлено и отложено до завершения следующего отображения. Такой подход позволяет системе вывода, по крайней мере хотя бы иногда, отобразить текущее состояние и произвести расчет логики игрового процесса, если уж физические расчеты пошли на взлет (намеренный каламбур).

Физические обновления и изменения во время выполнения

Когда физическая система приступает к обработке следующего временного шага, она должна рассчитать перемещение всех активных объектов `Rigidbody`, обнаружить все столкновения и вызвать соответствующие обработчики столкновений. Именно по этой причине документация Unity четко рекомендует вносить изменения в объекты `Rigidbody` только внутри методов `FixedUpdate()` и других методов-обработчиков, вызываемых физической системой. Эти методы тесно связаны с частотой обновления физического движка, в отличие от других методов игрового цикла, например метода `Update()`.

Это означает, что обратные вызовы, такие как `OnTriggerEnter()`, являются безопасным местом для изменения объектов `Rigidbody`, а такие методы, как `Update()` и сопрограммы, основанные на времени, таким местом не являются. Пренебрежение этой рекомендацией вызывает неадекватное физическое поведение, так как позволяет несколько раз внести изменения в один и тот же объект до того, как физическая система получит шанс обнаружить их и обработать, что является причиной некоторых особенно запутанных странностей в игровом процессе.

Из этого логически следует, что чем больше времени занимает итерация фиксированного обновления, тем меньше остается времени на

следующий этап игрового процесса и отображения. Большую часть времени физический движок не очень загружен работой, и потому обратные вызовы `FixedUpdate()` получают достаточно времени для выполнения.

Однако в некоторых играх физическому движку приходится выполнять много вычислений при каждом обновлении, что отрицательно сказывается на частоте кадров отображения, вызывая ее падение из-за повышенной нагрузки на физическую систему. В действительности система отображения продолжает действовать в обычном режиме, но всякий раз, когда подходит время фиксированного обновления, время создания текущего отображения ограничивается, что вызывает внезапные простановки и появление дополнительных визуальных эффектов, вызванных прерыванием работы физической системы из-за превышения максимального выделяемого ей времени. Все вместе это приводит к ухудшению восприятия игры пользователями.

Следовательно, чтобы добиться гладкой и согласованной смены кадров, необходимо высвободить как можно больше времени для отображения, сведя к минимуму время, затрачиваемое физической системой на обработку. Это относится как к оптимистическому сценарию (ничего не движется), так и к наихудшему (все объекты одновременно врезаются друг в друга). Физическая система имеет несколько параметров настройки, связанных со временем, помогающих избежать подобных потерь производительности.

Статические и динамические коллайдеры

В Unity существует серьезная неоднозначность в толковании термина «static». Мы уже рассмотрели статические игровые объекты, различные флаги из смейства **Static**, статическую пакетную обработку, а ведь есть еще и статические переменные и классы языка C#. Тем не менее оригинальная физическая реализация 3D-объектов в Unity имеет собственную концепцию статических коллайдеров. Позже, с появлением 2D-физики, для поддержания согласованности также были введены статические коллайдеры.

Поэтому важно помнить, что статические коллайдеры не являются объектами с установленным флагом **Static** (Статический), так как их флаги являются лишь концепцией движка Unity. Статические коллайдеры – это обычные коллайдеры, не имеющие присоединенных компонентов `Rigidbody`. С другой стороны, коллайдеры с присоединенным компонентом `Rigidbody` называются динамическими коллайдерами.

Физическая система связывает статические коллайдеры с иной, оптимизированной структурой данных, которая помогает упростить дальнейшую обработку. Такие объекты не реагируют на импульсы, придаваемые им при столкновении с другими объектами, но предотвращают перемещение сквозь них других объектов. Это делает статические коллайдеры идеально подходящими для реализации препятствий, которые не должны двигаться.

Обнаружение столкновений

В Unity поддерживаются три метода обнаружения столкновений: **Discrete** (Дискретное), **Continuous** (Непрерывное) и **Continuous-Dynamic** (Непрерывное динамическое). При выборе дискретного метода выполняется телепортация объектов на малое расстояние, величина которого определяется скоростью и прошедшим временем. После перемещения всех объектов проверяется перекрытие ограничивающих их объемов, и любое перекрытие интерпретируется как столкновение. При этом существует риск пропустить столкновение, если малые объекты движутся слишком быстро.

Оба метода непрерывного обнаружения столкновений сопровождают объекты от начальных до конечных позиций в текущий период времени и определяют любое столкновение вдоль всего пути. При этом уменьшается риск пропустить столкновение, и моделирование получается более точным, но за счет значительно больших затрат ресурсов центрального процессора, чем при использовании дискретного метода.

Только метод непрерывного обнаружения позволяет определять столкновения между заданным и статическими коллайдерами (напомню, что статические коллайдеры – это обычные коллайдеры, к которым не привязан компонент `Rigidbody`). Те же объекты могут одновременно интерпретироваться как использующие дискретное обнаружение, когда дело доходит до определения столкновений с другими динамическими объектами (с присоединенными компонентами `Rigidbody`). Метод непрерывного динамического обнаружения отличается тем, что позволяет определять столкновения заданного коллайдера с любыми другими коллайдерами, статическими и динамическими.

На рис. 5.2 показано, как действуют методы дискретного и непрерывного обнаружения столкновений применительно к паре небольших, быстро движущихся объектов.

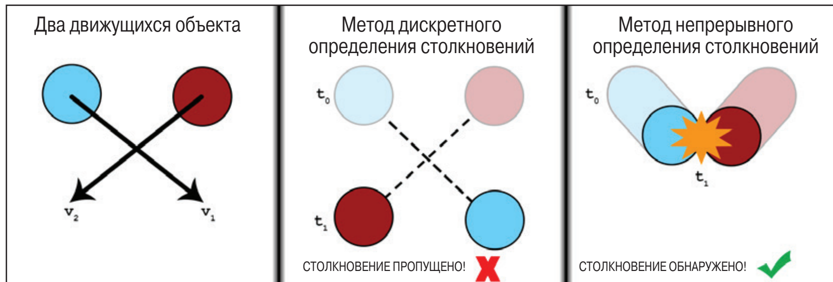


Рис. 5.2 ❖ Дискретное и непрерывное обнаружение столкновений

Этот экстремальный пример показан исключительно для иллюстрации. В этом примере при использовании дискретного метода объекты «телепортируются» на расстояние, в четыре раза превышающее их размер за единицу времени, что характерно только для очень малых объектов, движущихся с очень высокой скоростью, поэтому такие случаи редки в оптимально работающих играх. Чаще за 20 мс объекты преодолевают расстояние гораздо меньше своих размеров и поэтому обнаружить столкновение очень легко.

Виды коллайдеров

В Unity 5 имеются пять видов 3D-коллайдеров. Перечислим их в порядке возрастания накладных расходов: **Sphere** (Сфера), **Capsule** (Капсула), **Cylinder** (Цилиндр), **Box** (Короб) и **Mesh** (Меш). Первые четыре вида коллайдеров считаются простейшими и имеют соответствующую форму, хотя их можно масштабировать в различных направлениях для более полного удовлетворения потребностей. Меш-коллайдеры можно настроить на конкретную форму в зависимости от присвоенного меша.

Имеются также три вида 2D-коллайдеров: Circle (Круг), Box (Короб) и Polygon (Многоугольник), – функционально схожих с коллайдерами Sphere, Box и Mesh. Вся приведенная ниже информация в равной степени относится к эквивалентным 2D-фигурам.

Кроме того, меш-коллайдеры имеют две разновидности: выпуклые и вогнутые. Отличие вогнутого геометрического тела заключается в том, что как минимум один его *внутренний* угол (угол между двумя ребрами тела) больше 180 градусов. На рис. 5.3 показана разница между выпуклым и вогнутым геометрическими фигурами.

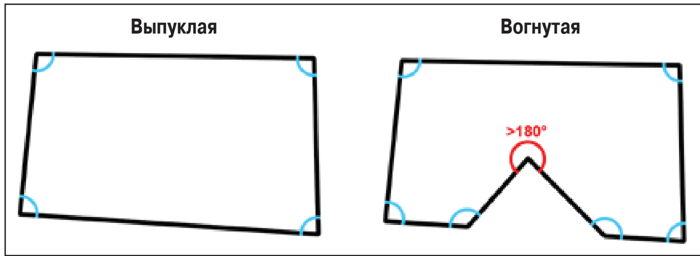


Рис. 5.3 ❖ Выпуклая и вогнутая геометрические фигуры

Разницу между выпуклой и вогнутой фигурами запомнить просто – вогнутая фигура имеет, по крайней мере, одну вогнутость.

Обе разновидноти меш-коллайдеров используют тот же компонент (`MeshCollider`), а собственно разновидность определяется флагом **Convex** (Выпуклый). Установленный флаг позволит объекту сталкиваться с другими выпуклыми меш-коллайдерами, а также с коллайдерами простейшей формы (сферы, короба и др.). Кроме того, если установить флаг **Convex** для меш-коллайдера вогнутой формы, физическая система автоматически упростит вогнутую форму, сгенерировав наиболее близкий по форме выпуклый коллайдер. В примере на рис. 5.3, если для вогнутого меша, показанного справа, установить флаг **Convex**, будет сгенерирован коллайдер, близкий по форме мешу, показанному слева.

В любом случае, физическая система попытается создать коллайдер, соответствующий по форме присоединенному мешу, содержащий до 255 вершин. Если целевой меш имеет больше вершин, это вызовет ошибку при создании меша. Обратите внимание, что в Unity 5 не поддерживаются вогнутые меш-коллайдеры с присоединенными объектами `Rigidbody`. Вогнутую форму могут иметь только статические коллайдеры (например, неподвижный объект с коллайдером) или области событий (например, бассейн необычной формы с кислотой).

Матрица столкновений

Физическая система поддерживает матрицу столкновений, определяющую пары объектов, которые могут сталкиваться. Объекты, не включенные в эту матрицу, автоматически игнорируются физической системой при обнаружении перекрытия их объемов. Это экономит время на этапе определения столкновений и, кроме того, позволяет объектам проходить друг сквозь друга без столкновений.

Матричная система определения столкновений использует в своей работе систему слоев Unity. Матрица хранит все возможные комбинации слоев, а установка флага означает, что на этапе определения столкновений будут проверяться коллайдеры обоих этих слоев. Обратите внимание, что нельзя заставить отреагировать на столкновение только один из двух объектов. Если один слой может сталкиваться с другим слоем, они оба должны реагировать на столкновения (за исключением статических коллайдеров, не имеющих реакции на столкновения).

Чтобы получить доступ к матрице столкновений, выберите пункт **Edit** ⇨ **Project Settings** ⇨ **Physics** (или **Physics2D**) ⇨ **Layer Collision Matrix** (Правка ⇨ Параметры проекта ⇨ Физика (или 2D-физика) ⇨ Матрица столкновений).

Обратите внимание, что общее число слоев в проекте не может превышать 32 (поскольку физическая система использует 32-разрядные битовые маски для определения возможности столкновения между слоями), поэтому необходимо продумать распределение объектов между слоями, которое будет использовано во всем проекте. Если по какой-то причине 32 слоев окажется недостаточно, можно попробовать найти способы повторного использования слоев или удалять слои, ставшие ненужными.

Активное и неактивное состояния компонента Rigidbody

Все современные физические движки поддерживают технологию оптимизации, переводя объекты в состояние покоя путем переключения их внутреннего состояния из активного в неактивное. На обновление неактивного объекта тратится минимальное время процессора, пока он не будет активизирован внешними силами или событиями.

Понятие «покоя» по-разному трактуется разными движками. Для его определения могут использоваться линейная и угловая скорость, кинетическая энергия, импульс и другие физические свойства компонента `Rigidbody`. В любом случае, если выбранный параметр не превышает некоторого порогового значения в течение небольшого промежутка времени, физический движок считает, что объект не должен перемещаться, пока не претерпит новых столкновений или к нему не будет приложена новая сила. До тех пор неактивный объект будет оставаться в текущей позиции.

В сущности, физический движок автоматически прекращает некоторые вычисления для объектов с малой кинетической энергией.

Но это не приводит к их полному удалению из модели. Если к покоящемуся объекту приближается объект с компонентом `Rigidbody`, физический движок определит столкновение и активизирует объект, восстановив его обработку согласно модели. Пороговое значение для перевода объекта в неактивное состояние можно задать, выбрав пункт **Edit** ⇨ **Project Settings** ⇨ **Physics** ⇨ **Sleep Threshold** (Правка ⇨ Параметры проекта ⇨ Физика ⇨ Порог для деактивации). Существует также возможность определять общее число активных объектов с компонентом `Rigidbody` в области Physics профилировщика.

Отбрасывание лучей и объектов

Еще одной общей чертой физических движков является возможность «отбрасывания» луча из одной точки в другую и сбора данных об одном или нескольких объектах, пересекающих луч. Отбрасывание лучей часто применяется для реализации важных игровых функций. Стрельба из орудий обычно реализуется с помощью отбрасывания луча от игрока к цели и определения подходящего объекта на его пути (даже если это просто стена).

Можно также получить список целей в радиусе поражения взрыва, например гранаты или огненного шара, с помощью метода `Physics.OverlapSphere()`, реализующего определение объектов, попадающих в область воздействия эффекта.

С помощью методов `Physics.SphereCast()` и `Physics.CapsuleCast()` можно также отбрасывать в пространство целые объекты. Эти методы часто используются, когда необходимо получить лучи больших размеров или когда нужно увидеть, что находится на пути перемещения персонажа.

Оптимизация производительности физической системы

Теперь, опираясь на понимание наиболее значительных возможностей физического движка Unity, можно перейти к рассмотрению методов оптимизации производительности физической системы.

Настройка сцены

Во-первых, существует ряд методов, применение которых к сцене позволит улучшить согласованность моделирования физических процессов. Обратите внимание, что эти методы не всегда направлены на

уменьшение потребления центрального процессора или памяти, но их применение ведет к снижению вероятности нестабильной работы физического движка.

Масштабирование

Масштаб всех физических объектов игрового мира следует удерживать как можно ближе к соотношению `1:1:1`. Это означает, что для ускорения свободного падения по умолчанию `-9.81` единица масштаба игрового мира должна соответствовать 1 метру, так как сила тяжести на поверхности земли составляет 9,81 м/с^2 (в большинстве игр имитируется именно эта ситуация). Размеры объекта должны отражать подразумеваемый масштаб игрового мира, поскольку слишком большой масштаб приведет к замедлению падений объектов, если это происходит в реальном мире. Обратное также верно: при слишком мелком масштабе объекты будут падать слишком быстро и выглядеть нереалистично.

Настроить подразумеваемый масштаб игрового мира можно, изменив ускорение свободного падения в **Edit** ⇨ **Project Settings** ⇨ **Physics** (или **Physics2D**) ⇨ (Правка ⇨ Параметры проекта ⇨ Физика (или 2D-физика) ⇨ Гравитация). Но имейте в виду, что результаты любых арифметических вычислений с плавающей запятой наиболее точны при значениях, близких к 0, поэтому если значения масштаба некоторых объектов значительно превышают величину (`1,1,1`), даже если они соответствуют подразумеваемым значениям мирового масштаба, все еще будет велика вероятность неустойчивого поведения физической системы. Итак, в начале работы над проектом следует импортировать и масштабировать наиболее распространенные физические объекты с помощью значений масштаба, близких к (`1,1,1`), и затем настроить соответствующее значение ускорения свободного падения. Это послужит ориентиром при работе со вновь вводимыми новыми объектами.

В Unity 4 имеется также значение Speed of Sound (Скорость звука) в настройках Audio (Аудио), используемое для имитации доплеровского аудиоэффекта. По умолчанию используется значение `343`, соответствующее скорости звука в воздухе 343 м/с. Изменение подразумеваемого масштаба игрового мира посредством изменения значения ускорения свободного падения потребует его корректировки для поддержания согласованности. Unity 5 рассчитывает эффект Доплера разными способами, поэтому эта переменная была удалена, чтобы снять этот вопрос.

Позиционирование

Аналогично расположение всех объектов вблизи точки (0,0,0) приведет к увеличению точности операций с плавающей запятой и улучшению согласованности при моделировании. Космические или спортивные тренажеры пытаются имитировать невероятно большие пространства, но они обычно используют технологию, позволяющую скрытно телепортировать (или просто оставлять на том же месте) персонаж в игровом мире. Таким образом, при моделировании путешествия все перемещается или делится на отсеки так, чтобы участвующие в физических расчетах значения всегда были близки к нулю. Это гарантирует близость всех объектов к позиции (0,0,0) для повышения точности результатов операций с плавающей запятой при перемещении игрока на большие расстояния.

Другие виды игр также следует освободить от рисков, вносимых неточностью вычислений с плавающей запятой. Даже если работа над проектом уже ведется (поскольку затягивание с внесением изменений и тестированием лишь добавит хлопот), нужно постараться добиться того, чтобы все физические объекты располагались как можно ближе к позиции (0,0,0). Плюсом этого подхода является убыстрение процесса добавления и позиционирования объектов в игровом мире проекта.

Масса

Документация Unity рекомендует устанавливать значение свойства `mass` (масса) близким к 0,1 и избегать значений выше 10 из-за нестабильности поведения объектов с такой массой: http://docs.unity3d.com/ScriptReference/Rigidbody-mass.html.

Это означает, что не следует привязывать массу к определенным единицам измерения, например фунтам или килограммам, а использовать относительные значения для объектов. Старайтесь поддерживать последовательное и разумное соотношение масс сталкивающихся объектов. Наличие объектов с массой более 1000 наверняка приведет к беспорядочному реагированию из-за большой разницы импульсов и потерь точности вычислений с плавающей запятой. Объекты, участвующие во взаимных столкновениях, должны иметь близкие значения массы, а пары объектов с сильно отличающимися массами желательно изъять из матрицы столкновений (подробнее об этом ниже).

Неправильное соотношение масс – наиболее распространенная причина нестабильности и беспорядочного поведения физической системы.

Обратите внимание, что сила тяжести одинаково действует на все объекты независимо от их массы. Поэтому не имеет значения, если свойство `mass` (масса) резинового мяча и военного корабля равны 1. Нет необходимости регулировать силу тяжести для компенсации любых предположений, касающихся значений относительного свойства `mass` (масса). Но имеет значение сопротивление воздуха, которое действует на падающий объект (именно поэтому перо падает медленнее, чем более плотный объект с той же массой). То есть для поддержания реалистичности происходящего нужно настроить свойство `drag` (аэродинамическое сопротивление) или силу тяжести для каждого из таких объектов в отдельности (для этого следует сбросить флаг **Use Gravity** (Учитывать силу тяжести) и реализовать учет силы тяжести, действующей на объект, с помощью сценария).

Правильное использование статических коллайдеров

Как уже упоминалось, физическая система автоматически создает структуру данных из всех статических коллайдеров (коллайдеров объектов без компонента `Rigidbody`), отдельно от структуры, управляющей динамическими коллайдерами (коллайдерами объектов с компонентами `Rigidbody`). К сожалению, если во время выполнения в структуру вводятся новые объекты, она должна генерироваться повторно. Это может вызвать значительный всплеск потребления ресурсов центрального процессора. Поэтому важно избегать создания новых статических коллайдеров во время игрового процесса.

Кроме того, *перемещение*, *поворот* или *масштабирование* статических коллайдеров вызывает процесс регенерации, и этого также следует избегать. Если имеются коллайдеры, которые нужно перемещать, но они не должны реагировать на столкновения с другими объектами, к ним следует присоединить компонент `Rigidbody`, сделав их динамическими коллайдерами, и установить флаг **Kinematic** (Кинематический) в значение `true`. С этим флагом объекты не будут реагировать на внешние импульсы при столкновении с другими объектами, что позволит им вести себя подобно статическим коллайдерам. Но теперь они будут входить в структуру поддержки движущихся объектов, их можно перемещать с помощью сценариев (в методе `FixedUpdate`!) и передавать импульсы другим объектам.

Именно поэтому флаг **Kinematic** (Кинематический) часто используется для объектов, управляемых игроками. Они способны отталкивать другие объекты, но сами невосприимчивы к толчкам.

Оптимизация матрицы столкновений

Как уже упоминалось, матрица столкновений физической системы определяет объекты в слоях, которые могут сталкиваться с объектами в других слоях. Или, выражаясь более лаконично, какие пары объектов *контролируются* физической системой. Все прочие пары слоев или объектов просто игнорируются физическим движком, что делает этот механизм важным средством уменьшения нагрузки, минимизирующим количество проверок, выполняемых на каждом шаге.

Напоминание: доступ к матрице столкновений можно получить, выбрав **Edit** ⇨ **Project Settings** ⇨ **Physics** (или **Physics2D**) ⇨ **Layer Collision Matrix** (Правка ⇨ Параметры проекта ⇨ Физика (или 2D-физика) ⇨ Матрица столкновений).

На рис. 5.4 показан пример матрицы столкновений для аркадного шутера.

▼ Layer Collision Matrix

	World	Powerups	Enemy Missiles	Player Missiles	Enemies	Player	UI	Water	Ignore Raycast	TransparentFX	Default
Default	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
TransparentFX	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	
Ignore Raycast	☐	☐	☐	☐	☐	☐	☐	☐	☐		
Water	☐	☐	☐	☐	☐	☐	☐	☐			
UI	☐	☐	☐	☐	☐	☐	☐				
Player	☑	☑	☑	☐	☑	☐					
Enemies	☑	☐	☐	☑	☐						
Player Missiles	☑	☐	☐	☐							
Enemy Missiles	☑	☐	☐								
Powerups	☐	☐									
World	☐										

Рис. 5.4

В этом примере сведено к минимуму количество проверок возможных столкновений между объектами. Так как источники энергии (Powerups) может собирать только игрок (Player), нет необходимости отслеживать столкновения между источниками энергии и объектами из других слоев. С другой стороны, нет смысла отслеживать столкновения снарядов с объектами, стреляющими ими, что отраже-

но в исключении столкновений снарядов врагов (Enemy Projectiles) с самими врагами (Enemies) и столкновений снарядов игрока (Player Projectiles) с самим игроком (Player). Мы должны определять столкновения со всеми объектами игрового мира (World, стены и прочие поверхности), но нет необходимости определять столкновения межу снарядами (хотя в некоторых играх реализуется и это!).

Необходимо проверить все комбинации слоев в матрице столкновений на соответствие логике, чтобы убедиться, что драгоценное время не тратится на ненужные проверки между неподходящими парами объектов.

Предпочтение дискретного обнаружения столкновений

Для большинства объектов следует использовать способ определения столкновений по умолчанию **Discrete** (Дискретное). Одноразовая телепортация объектов и выявление совмещения пар близко расположенных друг к другу объектов не составляют особого труда. Тогда как для интерполяции траекторий движения объектов между начальной и конечной позициями с одновременной проверкой даже незначительного совмещения их объемов во всех точках пути требуется значительно больше вычислений.

Вариант **Continuous** (Непрерывное) определения столкновений привносит на порядок больше затрат, чем **Discrete** (Дискетное), а вариант **ContinuousDynamic** (Непрерывное динамическое) – еще на порядок больше, чем **Continuous**! Слишком большое количество объектов с непрерывным способом определения столкновений приведет к серьезным потерям производительности в сложных сценах. В любом случае, затраты умножаются на количество объектов, проверяемых в течение кадра при определении столкновений статических или динамических коллайдеров.

Следовательно, вариант непрерывного определения должен использоваться только в экстремальных обстоятельствах и только для определения важных, часто пропускаемых столкновений со статическими объектами игрового мира, например когда некоторые объекты движутся быстро и нужно гарантировать, что они никогда не покинут игрового мира и не телепортируются сквозь стены. И наконец, вариант **ContinuousDynamic** (Непрерывное динамическое) следует использовать, только когда необходимо определять столкновения между парами быстро движущихся динамических коллайдеров. Если

нет твердой уверенности в выборе варианта, то всегда отдавайте предпочтение варианту **Discrete** (Дискретное).

Но вариант **Discrete** (Дискретное) плохо подходит для крупных масштабов. Например, если основой игры является масса малых физических объектов, дискретный способ определения столкновений просто не сможет эффективно обнаруживать столкновения для поддержания нужного качества продукта. К счастью, имеется возможность изменить частоту фиксированных обновлений, чтобы дать больше шансов механизму определения столкновений, работающему в режиме **Discrete** (Дискретное).

Изменение частоты фиксированных обновлений

Как уже упоминалось ранее, фиксированные обновления и интервал моделирования физической системы тесно связаны между собой. Поэтому изменение частоты фиксированных обновлений приводит не только к изменению частоты, с которой физическая система производит расчеты и осуществляет обратные вызовы, но и к изменению частоты вызовов функций `FixedUpdate()`. Следовательно, изменение этого значения является рискованным шагом, если работа над проектом уже ведется и многое в нем завязано на эти обратные вызовы.

Изменить частоту `FixedUpdate` можно, присвоив нужное значение свойству **Edit ⇨ Project Settings ⇨ Time ⇨ Fixed Timestep** (Правка ⇨ Параметры проекта ⇨ Время ⇨ Интервал фиксированных обновлений) в редакторе (рис. 5.5) или свойству `Time.fixedDeltaTime` в коде сценария.

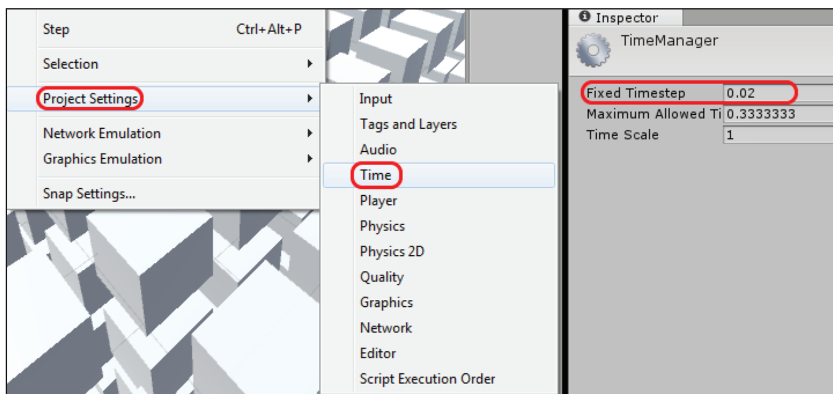


Рис. 5.5 ❖ Свойство в редакторе, управляющее интервалом фиксированных обновлений

Уменьшение этого значения (увеличение частоты) заставит физическую систему выполнять обработку чаще и увеличит вероятность определения столкновений между динамичными объектами в дискретном режиме. Естественно, это увеличит потребление ресурсов центрального процессора, поскольку чаще будут вызываться функции `FixedUpdate()` и физический движок чаще будет выполнять обновления, перемещая объекты и проверяя столкновения.

И наоборот, увеличение этого значения (снижение частоты) освободит центральный процессор для выполнения других задач перед следующей физической обработкой или, если взглянуть с другой точки зрения, даст физической системе больше времени на обработку, перед тем как она перейдет к выполнению следующего этапа. Однако снижение частоты до значения ниже максимальной скорости движения объектов приведет к невозможности дискретного определения столкновений (в зависимости от размеров объектов).

Это делает абсолютно необходимым тщательное тестирование при каждом изменении значения **Fixed Timestep** (Интервал фиксированных обновлений). Даже полностью понимая влияние этого значения, трудно предсказать, как скажется его изменение на игре и как оно повлияет на качество. Соответственно, изменять это значение следует в самом начале работы над проектом и затем изменять его как можно реже и с обязательным тестированием всех возможных физических ситуаций.

Для оценки приемлемости измененного значения параметра **Fixed Timestep** (Интервал фиксированных обновлений) можно создать тестовую сцену, где несколько объектов на высокой скорости сталкиваются друг с другом. Игровой процесс может быть довольно сложным, с большим количеством фоновых задач и непредвиденными действиями игрока, добавляющими работы физической системе или уменьшающими время на обработку текущей итерации. Условия игрового процесса сложно воссоздать в вакууме, и нет другой реальной альтернативы, кроме тестирования нового значения **Fixed Timestep** (Интервал фиксированных обновлений), позволяющего убедиться, что изменения не ухудшают качества игры.

Для исключения наблюдаемой нестабильности всегда, в качестве последнего средства, можно включить режим непрерывного обнаружения столкновений. Но, к сожалению, даже если его использование окажется целесообразным, вероятнее всего, оно вызовет проблемы с производительностью из-за накладных расходов, свойственных непрерывному определению столкновений, о чем уже упоминалось. Имеет смысл выполнить профилирование сцены до и после включе-

ния этого режима, чтобы убедиться, что достигнутые преимущества перевешивают затраты.

Настройка максимально допустимой длительности

При частом превышении ограничения **Maximum Allowed Timestep** (Максимально допустимая длительность) физическое поведение будет выглядеть странно. Объекты с компонентом `Rigidbody` будут замедляться или зависать в пространстве из-за преждевременной остановки физического движка. Это явный признак, что необходимо оптимизировать другие аспекты физической системы. Но, по крайней мере, можно быть уверенным, что пороговое ограничение предотвратит полное зависание при резком увеличении времени обработки физической модели.

Настроить пороговое значение можно, изменив параметр **Edit** ⇨ **Project Settings** ⇨ **Time** ⇨ **Maximum Allowed Timestep** (Правка ⇨ Параметры проекта ⇨ Время ⇨ Максимально допустимая длительность). Значение по умолчанию ограничивает максимальное время на обработку **0,333** секунды, и его превышение приведет к очень заметному снижению частоты кадров (до 3 кадров в секунду!). Если все же наступит момент, когда потребуется изменить этот параметр, это, несомненно, будет вызвано перегрузкой физической системы, поэтому изменяйте его, только когда уже опробованы все прочие подходы.

Уменьшение отбрасывания лучей и ограничение проверяемого объема

Все методы отбрасывания лучей очень полезны, но отрицательно сказываются на производительности (особенно `CapsuleCast()` и `SphereCast()`), поэтому используйте их как можно реже. Избегайте регулярного их вызова в методах-обработчиках и сопрограммах и используйте только в коде сценариев для обслуживания ключевых событий.

Если абсолютно необходимы непрерывные линии, лучи или эффекты столкновения областей в сцене (например, лазерные охранные системы или непрерывно горящие огни), их лучше моделировать с помощью простого триггерного коллайдера вместо непрерывного отбрасывания лучей или проверок перекрытия.

Если такая подмена невозможна и непрерывное отбрасывание лучей действительно необходимо (например, для имитации красной точки лазерного прицела), объем вычислений можно минимизировать с помощью объектов `LayerMask`.

Например, «ленивое» отбрасывание лучей можно реализовать так:

```
[SerializeField] float _maxRaycastDistance;

void PerformRaycast() {
    RaycastHit hitInfo = new RaycastHit();
      if (Physics.Raycast(new Ray(transform.position, transform.forward),
          out hit, _maxRaycastDistance)) {
        // обработка результатов отбрасывания лучей
      }
}
```

Эта перегруженная версия метода Physics.Raycast() позволит лучу столкнуться только с первым объектом из любого слоя, встретившимся на его пути. Метод Physics.Raycast имеет несколько перегруженных версий, принимающих в качестве аргумента объект LayerMask. Этим можно воспользоваться для отбора объектов, которые нужно проверить при отбрасывании луча, что значительно снизит нагрузку на физический движок:

```
[SerializeField] float _maxRaycastDistance;
[SerializeField] LayerMask _layerMask;

void PerformRaycast() {
    RaycastHit hitInfo = new RaycastHit();
      if (Physics.Raycast(new Ray(transform.position, transform. forward),
          out hit, _maxRaycastDistance, _layerMask)) {
        // обработка результатов отбрасывания лучей
      }
}
```

Объект LayerMask можно настраивать через его представление в панели **Inspector** (Инспектор), как показано на рис. 5.6.

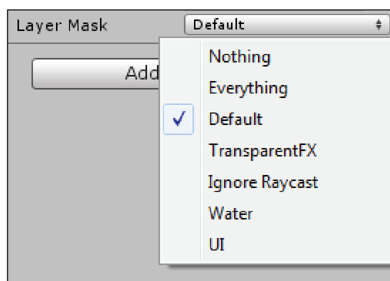


Рис. 5.6 ❖ Представление объекта LayerMask в панели **Inspector** (Инспектор)

Обратите внимание: так как классы `RaycastHit` и `Ray` размещаются в неуправляемой памяти движка Unity, их применение не приводит к выделению памяти, привлекающей внимание сборщика мусора. Более подробно о сборщике мусора рассказывается в главе 7 «Мастерство управления памятью».

Избегайте сложных меш-коллайдеров

Вот как разные виды коллайдеров располагаются в порядке убывания эффективности: сферы, капсулы, цилиндры, короба, выпуклые меш-коллайдеры и, наконец, вогнутые меш-коллайдеры. Причем четыре основных примитива на порядок эффективнее любых меш-коллайдеров, поскольку математические расчеты для обнаружения и разрешения конфликтов между ними довольно кратки и оптимизированы. Определение столкновений между меш-коллайдером выпуклой формы и другими коллайдерами обходится недешево, но определение столкновений между меш-коллайдером вогнутой формы и прочими обходится еще дороже.

Проверка перекрытия объемов пары меш-коллайдеров вогнутой формы – это «математический армагеддон» физического моделирования в реальном времени (и останется таковым, по крайней мере в ближайшие годы!). Чтобы защититься от нашей глупости, Unity категорически запрещает выполнение проверок перекрытия двух меш-коллайдеров вогнутой формы.

Главным парадоксом физической и графической систем 3D-приложений является разница в сложности обработки объектов в форме сферы и короба. Для создания идеального сферического меша требуется бесконечное количество полигонов, но в физическом движке задача определения точек контакта и столкновений сферы решается тривиально просто. Напротив, для отображения обычного короба требуется незначительное количество полигонов и затрат, но для определения точек контакта и столкновений требуется намного больше математических расчетов и затрат на обработку. Это привело к тому, что большинство графических и физических систем заполнило игровой мир графическими объектами с малым количеством полигонов в форме сферы. Однако ненормально выглядел бы угловатый объект, катящийся подобно шару.

При работе с физическим движком важно не забывать, что физическое представление объекта не обязательно должно соответствовать его графическому представлению. Удобство заключается в том, что

графический меш часто можно представить в виде тела более простой формы с очень похожим физическим поведением. Это избавляет от необходимости использовать чрезмерно сложные меш-коллайдеры.

Такое разделение графического и физического представлений позволяет оптимизировать производительность одной системы без (обязательного) негативного влияния на другую. Если это не оказывает заметного влияния на игровой процесс, можно свободно представлять сложные графические объекты с помощью невидимых простейших физических форм. Если игрок никогда этого не заметит, в этом нет никакого вреда!

Эту задачу можно решить одним из двух способов: либо путем аппроксимации физического поведения тела сложной формы посредством одного (или более) стандартных примитивов, либо с помощью значительно упрощенного меш-коллайдера.

Используйте простые примитивы

Большинство объектов можно аппроксимировать одним из четырех примитивных коллайдеров (в порядке убывания эффективности): сфера, капсула, цилиндр или короб. В самом деле, никто не требует представлять объект с помощью единственного коллайдера. Можно использовать несколько коллайдеров, подходящих для определения столкновений объектов сложной формы, путем присоединения дополнительных дочерних игровых объектов с их собственными коллайдерами. Это практически всегда менее затратно, чем применение одного меш-коллайдера, и этому менее сложному решению следует отдавать предпочтение.

На рис. 5.7 показано несколько сложных графических объектов, представленных одним или несколькими простыми примитивами в физической системе.

Использование меш-коллайдера для любого из этих объектов увеличит затраты на их обработку в сравнении с приведенными здесь примитивными коллайдерами. Поэтому стоит потратить время, чтобы узнать все возможности упрощения объектов с помощью таких примитивов для обеспечения прироста производительности.

Например, меши вогнутой формы уникальны в том смысле, что могут содержать полости или отверстия, в которые могут «упасть» или даже пройти насквозь другие меши, что позволит объектам «падать» сквозь игровой мир, если некоторые его области имеют вогнутую форму. С этой целью в наиболее ответственных местах лучше разместить коллайдеры в форме короба.

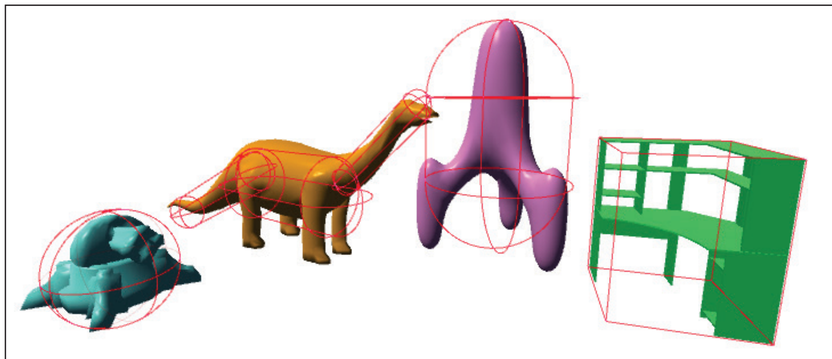


Рис. 5.7 ❖ Сложные объекты, представленные в физической системе простыми примитивами

Используйте простые меш-коллайдеры

Меш, соответствующий меш-коллайдеру, не обязательно должен совпадать с графическим представлением объекта (как делает Unity по умолчанию). Это дает возможность присвоить свойству `mesh` меш-коллайдера другой меш, более простой в сравнении с графическим представлением объекта.

На рис. 5.8 показан пример сложного графического меша, в соответствие которому поставлен более простой меш-коллайдер.

Рис. 5.8 ❖ Графический меш и более простой меш-коллайдер

Отображаемый меш преобразован в меш выпуклой формы с малым количеством полигонов, что значительно уменьшает затраты на определение его перекрытия с другими коллайдерами. В зависимости от того, насколько хорошо произведена оценка исходного объекта, различия в процессе игры могут быть незаметными, особенно в случае топора, который перемещается быстро, раскачиваясь во время нападения, что делает разницу между двумя мешами малозаметной.

Избегайте сложных физических компонентов

Некоторые специальные коллайдеры, такие как коллайдеры ландшафтов, тканей и колес, привносят на несколько порядков больше затрат, чем любые примитивные коллайдеры, а в некоторых случаях и меш-коллайдеры. Не следует включать такие компоненты в сцену, если без них можно обойтись. Например, если имеются удаленные ландшафтные объекты, до которых игрок никогда не доберется, нет никакого смысла присоединять к ним ландшафтный коллайдер.

В игре, содержащей компоненты `Cloth`, которые имитируют ткань, подумайте о замене их другими объектами, когда она выполняется на устройстве с ограниченными ресурсами, или просто используйте анимации, имитирующие ткань (хотя я могу понять тех, кто влюблен в движение материи).

В игре, использующей коллайдеры колес, просто нужно попытаться как можно реже использовать коллайдеры колес! Для транспортных средств с более чем четырьмя колесами достаточно использовать только четыре коллайдера колес, чтобы создать правильное физическое поведение, а для остальных колес оставить лишь графическое представление.

Пусть физические объекты поспят

Функция сна в физическом движке способна создавать проблемы в игре.

Напоминание: предельное время неактивного состояния можно задать с помощью свойства **Edit** ⇨ **Project Settings** ⇨ **Physics** ⇨ **Sleep Threshold** (Правка⇨ Параметры проекта ⇨ Физика ⇨ Порог сна).

Во-первых, некоторые разработчики не замечают, что многие из объектов с компонентом `Rigidbody` находятся в состоянии сна большую часть своего существования. Из-за этого они полагают, что удвоение (к примеру) количества объектов с компонентом `Rigidbody`

повлечет удвоение затрат на их обработку. Это маловероятно. Увеличение частоты столкновений и общего времени нахождения объектов в активном состоянии, скорее всего, повлечет экспоненциальный рост затрат, а не линейный, и непредвиденные затраты при добавлении новых физических объектов. Имейте это в виду, принимая решение об увеличении физической сложности сцен.

Во-вторых, существует опасность образования «анклавов» неактивных физических объектов. Анклавы создаются при соприкосновении друг с другом большого количества объектов с компонентом `Rigidbody`, постепенно впадающих в спячку, – представьте кучу коробок, сваленных в большую кучу. В конце концов, все объекты с компонентом `Rigidbody` перейдут в неактивное состояние, затратив до этого определенную энергию системы, и станут отдыхать. Однако из-за того, что они касаются друг друга, пробуждение одного из этих объектов запустит цепную реакцию пробуждения всех близлежащих объектов с `Rigidbody`. Произойдет всплеск потребления центрального процессора, поскольку десятки объектов вновь включатся в моделирование, и потребуется обработать множество столкновений, пока объекты снова не заснут.

Если найти способ определения сформировавшихся анклавов, можно было бы уничтожить/деактивировать некоторые из них и исключить появление слишком крупных анклавов. Но реализация этого способа зависит от конкретной игры, так как регулярные глобальные проверки и оценка расстояний между всеми объектами с компонентом `Rigidbody` являются весьма затратной задачей. Например, в игре, где игроку требуется переместить множество физических объектов в определенную область (например, игра с заталкиванием овец в загон), можно удалять у объекта динамический коллайдер, как только игрок переместит его в нужное положение, блокировав объект в пункте назначения и ослабив, таким образом, нагрузку на физический движок.

В-третьих, при изменении любого из свойств компонента `Rigidbody` во время выполнения, таких как **mass** (масса), **drag** (аэродинамическое сопротивление), **Use Gravity** (Учитывать силу тяжести) и др., объект также пробуждается. При периодическом изменении этих значений (например, в игре, где со временем изменяются размеры и масса объектов), объекты остаются активными дольше, чем обычно. Это же относится к применению сил, то есть если к объекту применяется нестандартная сила тяжести (как было описано выше, в разделе «*Масса*»), следует избегать применения силы тяжести при каждом фиксированном обновлении, в противном случае объект будет неспособен заснуть.

Объекты в состоянии сна могут быть и благословением, и проклятием. Они экономят вычислительные ресурсы, но затраты на их одновременное пробуждение могут значительно снизить производительность во время игрового процесса, причем моделирующая система в этот момент будет слишком занята, чтобы суметь вернуть их в состояние сна. Старайтесь избегать подобных ситуаций, переводя объекты в состояние сна как можно быстрее и препятствуя их объединению в большие кластеры.

Изменение количества итераций

Моделирование шарниров, пружин и других объектов с компонентами `Rigidbody` является сложной задачей для физических движков. Из-за необходимости взаимозависимого согласования (ограничения перемещений) такого сочленения двух объектов системе обычно требуется несколько попыток для решения необходимых математических уравнений. Чтобы получить точный результат при каждом изменении скорости любого звена цепочки объектов, требуется выполнить множество итераций.

Поэтому следует ограничить максимальное количество попыток решения расчета в конкретной ситуации, пожертвовав точностью результата. Не желательно затрачивать слишком много времени на одно столкновение, поскольку у физического движка имеется масса другой работы, которую он должен выполнить в течение той же итерации. Но, с другой стороны, не следует слишком сильно уменьшать количество итераций, поскольку результат станет чересчур неточным по отношению к результату, который можно было бы получить за большее время.

То же относится к разрешению столкновений и контактов внутри объекта. Правильный результат практически всегда можно получить с помощью единственной итерации, за исключением очень малого числа редких и сложных столкновений. Такими исключениями являются случаи, когда сторонние объекты сталкиваются с шарнирами и для получения окончательного результата требуется несколько итераций вычислений.

Изменить количество итераций можно с помощью параметра **Edit ⇨ Project Settings ⇨ Physics ⇨ Solver Iteration Count** (Правка ⇨ Параметры проекта ⇨ Физика ⇨ Количество итераций решения). В большинстве случаев значение по умолчанию, равное шести итерациям (семь – в Unity 4), вполне приемлемо. Но в играх, включающих

очень сложные способы сочленения, может потребоваться увеличить количество итераций, чтобы избавиться от неустойчивости (или даже разрушения) и слаженности движений персонажей. Однако в некоторых проектах количество итераций можно уменьшить. После изменения этого параметра необходимо провести тестирование, чтобы удостовериться, что проект по-прежнему соответствует нужному уровню качества.

Кстати, если обнаружится, что в игре регулярно возникают подергивания, странности и физические нестыковки, касающиеся объектов со сложными сочленениями (например, тряпичных кукол), следует постепенно увеличивать количество итераций решений, пока проблемы не исчезнут. Эти проблемы обычно возникают, если тряпичные куклы требуют значительных вычислений при столкновении объектов и заданное количество итераций не позволяет довести решение до требуемой точности. На данный момент один из шарниров вот-вот станет сверхновой звездой, перетащим и остальные на его орбиту!

Следует также проверить, соответствуют ли массы компонентов `Rigidbody` тряпичных кукол правилам, изложенным выше в этой главе, чтобы достичь более приемлемых результатов при обмене энергией и скоростями.

Оптимизация тряпичных кукол

Тряпичные куклы наиболее популярны среди персонажей с шарнирами по одной весьма веской причине. Они чрезвычайно забавны! При полном игнорировании современной тенденции забрасывать весь игровой мир трупами все же что-то есть в сложном хороводе движущихся вокруг объектов, а его разрушение подобно психологической «кнопке удовольствия».

Это делает заманчивым одновременное присутствие в сцене множества тряпичных кукол, но при этом возникает риск значительной потери производительности, когда слишком много тряпичных кукол движется и сталкивается с другими объектами. Причиной является количество итераций, необходимых для обработки их столкновений. Итак, давайте рассмотрим способы повышения производительности при использовании тряпичных кукол.

Уменьшите число шарниров и коллайдеров

В Unity имеется простой инструмент создания тряпичных кукол, доступный через пункт **GameObject ⇨ 3D Object ⇨ Ragdoll...** (Игровой объект ⇨ 3D-объект ⇨ Тряпичная кукла...) в Unity 5, и пункт

GameObject ⇨ Create Other ⇨ Ragdoll... (Игровой объект ⇨ Создать прочий ⇨ Тряпичная кукла...) в Unity 4. Этот инструмент можно использовать для создания тряпичной куклы из заданного объекта выбором соответствующих дочерних объектов с приданными коллайдерами, предназначенными для любой части тела или конечности. Этот инструмент всегда создает 11 различных коллайдеров и связывающих шарниров (таз, грудь, голова и по два коллайдера на конечность), но использование только шести коллайдеров (тело, голова и по одному коллайдеру на конечность) существенно сократит накладные расходы за счет реализма тряпичной куклы. Для этого можно удалить нежелательные коллайдеры и присвоить соответствующим свойствам **Connected Body** составных частей персонажа нужные родительские шарниры.

Такое упрощение можно использовать для уменьшения накладных расходов при слабой аппаратной поддержке или в качестве настройки при пониженных требованиях к качеству как компромисс, позволяющий увеличить количество тряпичных кукол в сцене. Его можно даже применять динамически, если определенное количество тряпичных кукол уже присутствует в сцене. Для этого понадобится вездесущий класс, отслеживающий имеющееся количество тряпичных кукол в сцене и осуществляющий переход на более простой вариант ради сохранения плавности игрового процесса.

Исключение столкновений тряпичных кукол между собой

При желании обработку столкновений тряпичных кукол друг с другом можно отключить при помощи матрицы столкновений. Обработка таких столкновений увеличивает затраты в экспоненциальной прогрессии, поскольку разрешение их является весьма сложной задачей, а приближенное ее решение может привести к беспорядочному поведению.

Отключение или удаление неактивных тряпичных кукол

Наконец, можно запретить повторное включение тряпичных кукол в физическое моделирование с момента их перехода в состояние сна. В некоторых играх после достижения тряпичной куклой своего «пункта назначения» игровой мир перестает рассматривать ее как интерактивный объект.

Определить состояние любого заданного коллайдера можно с помощью метода IsSleeping, и как только он перейдет в состояние сна, можно выполнить несколько вариантов действий: отключить все коллайдеры тряпичной куклы, запретив дальнейшее участие в моделировании, полностью удалить тряпичную куклу из сцены, продолжить следить за этой тряпичной куклой и удалить ее только в момент добавления в сцену другой тряпичной куклы.

Независимо от выбранного подхода улучшение производительности тряпичных кукол несомненно приведет к введению ограничений в процесс игры, выражающихся либо в уменьшении количества кукол, либо в их упрощении, либо в сокращении времени их существования, но это станет разумным компромиссом для сохранения производительности игры.

Когда следует использовать физическую систему

Как всегда, лучший способ улучшить производительность компонента – отказаться от излишнего его использования. При анализе особенностей любого движущегося объекта следует воспользоваться моментом, чтобы спросить себя: «А так ли необходимо подключать его к физическому движку?» Если нет, можно поискать возможности замены физической системы чем-то более простым и менее затратным.

Допустим, физическая система используется для выявления попадания игрока в зону поражения, и в игре это определяется достаточно просто: зона поражения находится на определенной высоте. В этом случае можно вообще не использовать физический коллайдер, ограничившись проверкой координаты *y* игрока.

В качестве другого примера рассмотрим моделирование метеоритного дождя, на первый взгляд кажется, что для реализации потребуется создать множество падающих предметов с физическими компонентами Rigidbody, определять моменты столкновений с землей при помощи коллайдеров и генерировать взрывы в точках ударов. Но поверхность земли может быть совершенно плоской, или у нас имеется доступ к карте высот ландшафта, что обеспечит элементарное выявление столкновений. В этом случае перемещение объекта можно упростить с помощью tween-анимации свойства transform.position без применения любых физических компонентов. В обоих случаях затраты на работу физической системы уменьшатся за счет упрощения и замены ее выполнением кода сценария.

Tween-анимация – обобщенный термин, обозначающий математическую интерполяцию свойства между двумя значениями с течением времени. В Asset Store Unity имеется масса полезных (и к тому же бесплатных!) библиотек анимаций, предоставляющих массу полезных функций.

Обратное также возможно. Иногда действия, требующие множества вычислений в сценариях, можно относительно просто реализовать с помощью физической системы. В качестве примера рассмотрим систему подсчета собранных объектов. При нажатии игроком клавиши **Pick up** (Подобрать) следует сравнить положение собираемых объектов с позицией игрока и определить ближайший такой объект. Весь код сценария можно заменить одним вызовом метода `Physics.OverlapSphere()`, чтобы получить все близлежащие объекты, а затем выбрать самый ближний (или просто подобрать их все!). При этом значительно уменьшится общее количество объектов, позиции которых нужно сравнивать при каждом нажатии клавиши.

Ваши возможности ограничены только пределами вашей изобретательности. Способность выявлять и устранять избыточные операции физической системы и/или определять дорогостоящие операции, реализованные с помощью сценариев, и заменять их средствами физического моделирования является важным навыком, который пригодится вам для поддержания высокой производительности текущих и будущих проектов игр.

О возможности перехода на Unity 5

Если вы используете Unity 4, в качестве последней меры для повышения производительности физической системы следует рассмотреть вопрос о переходе на Unity 5. В результате будет выполнено обновление движка PhysX с версии 2.8.3 до версии 3.3, имеющей более высокую производительность. Преимущества этого обновления сложно переоценить, поскольку при этом производительность физической системы увеличивается примерно в два раза, по сравнению с Unity 4. Обновление позволит уменьшить накладные расходы на перемещение статических коллайдеров, улучшит производительность непрерывного определения столкновений, обеспечит поддержку большего количества видов твердых тел, ускорит работу с компонентами коллайдеров ткани и колеса, а также добавит в физическую систему поддержку многоядерных процессоров. Проще говоря, обновление позволит сократить потери производительности или втиснуть в сцену больше физической активности без увеличения затрат.

Однако эти усовершенствования привели к значительным изменениям программного интерфейса для некоторых задач, а это значит, что сценарии и компоненты Unity 4 не полностью совместимы с Unity 5 (и это касается не только физической системы). Поэтому обновление вряд ли будет тривиальным, и не забывайте сделать резервную копию проекта перед обновлением. Объем работы в конечном счете зависит от сложности проекта и использования в нем приобретений из Asset Store. Все сторонние ресурсы необходимо обновить, а лишившиеся поддержки ресурсы заменить другими.

Итоги

Мы рассмотрели множество методов оптимизации физического моделирования игры с точки зрения производительности и согласованности. Лучший способ решения проблем, когда дело касается таких тяжеловесных систем, как физический движок, – избегать их использования. Чем меньше используются такие системы, тем меньше приходится беспокоиться о производительности. В худшем случае может понадобиться ограничить физическую активность только самыми существенными моментами, но, как мы уже знаем, существует много способов уменьшения сложности физической обработки без создания задержек, заметных в процессе игры.

Следующая глава посвящена графической системе Unity. В ней мы рассмотрим оптимизацию отображения графики с использованием ресурсов центрального процессора, высвободившихся за счет оптимизации производительности в предыдущих главах.

Глава 6

Динамическая графика

Нет никаких сомнений в сложности систем отображения на основе современных графических устройств. Даже вывод простого треугольника на экран требует участия многих компонентов, так как графические процессоры предназначены для параллельного выполнения большого количества потоков обработки данных, в отличие от обычных процессоров, способных выполнять практически любой вычислительный сценарий.

Современное отображение графики напоминает быстрый танец обработки и управления памятью, охватывающий программное обеспечение, аппаратное обеспечение, множество областей памяти, языков, процессоров, типов процессоров и большое количество специальных функций, действующих согласованно.

Что еще хуже, каждая ситуация, с которой приходится сталкиваться, требует своего подхода. Выполнение приложения на другом устройстве, даже от одного и того же производителя, часто отличается столь же разительно, как яблоки отличаются от апельсинов, из-за разных возможностей и предоставляемой функциональности. Бывает трудно выявить узкое место в сложной цепи устройств и систем, и нужно всю жизнь проработать в области 3D-графики, чтобы научиться интуитивно находить причины низкой производительности в современных графических систем.

К счастью, и в этом может помочь профилирование. Имея возможность собирать данные обо всех компонентах, сравнивать характеристики производительности и настраивать сцены для наблюдения за влиянием различных параметров на их поведение, можно получить достаточно информации для выявления и исправления проблем.

Итак, в этой главе будет описано, как собрать нужные данные, найти истинный источник проблем в недрах графической системы и выбрать лучший вариант их решения.

В предыдущих главах мы узнали, что центральный и графический процессоры трудятся в тандеме, чтобы определить, какие текстуры, меши, состояния визуализации, шейдеры и т. д. необходимые на текущем этапе отображения сцены. Мы также рассмотрели несколько способов уменьшения затрат на отображение с помощью статической и динамической пакетной обработки и управления файлами мешей и текстур посредством их сжатия, кодирования, mip-текстурирования, составления атласов и даже некоторых процедурных альтернатив.

Однако существует и множество других способов увеличения производительности отображения. Таким образом, в этой главе мы сначала познакомимся с общепринятыми методами, помогающими определить, что является узким местом – центральный или графический процессор – и что можно предпринять в каждом из этих случаев. Затем мы обсудим методы оптимизации, основанные на окклюзивной выбраковке и **уровне детализации (Level Of Detail**, **LOD)**, познакомимся с советами по оптимизации шейдеров, а также крупномасштабных механизмов отображения, таких как освещение и тени. В заключение, учитывая рост популярности мобильных устройств, мы рассмотрим методы увеличения производительности при ограниченных возможностях аппаратных средств.

Профилирование проблем отображения

Низкая производительность механизма отображения проявляется по-разному, в зависимости от того, что является ее причиной – **центральный** или **графический процессор**. В последнем случае корень проблем может располагаться в нескольких местах графического конвейера. Это затрудняет поиск, но, отыскав и решив проблему, можно надеяться на значительное улучшение, и даже небольшие усовершенствования в подсистеме отображения зачастую приводят к впечатляющим результатам.

Тема ограничений, накладываемых центральным и графическим процессорами, кратко рассматривалась в *главе 3 «Преимущества пакетной обработки»*. Резюмируя предыдущее обсуждение, отметим, что центральный процессор посылает инструкции отображения через графический программный интерфейс, затем через аппаратный драйвер они поступают в графический процессор и накапливаются в его

буфере команд. Эти команды поочередно обрабатываются системой параллельной обработки графического процессора, пока буфер не опустеет. Но в данной последовательности гораздо больше нюансов.

На рис. 6.1 представлены диаграмма (сильно упрощенная) типичного конвейера графического процессора (может изменяться в зависимости от применяемых технологий и оптимизации) и обобщенные операции, выполняемые на каждом из этапов отображения.

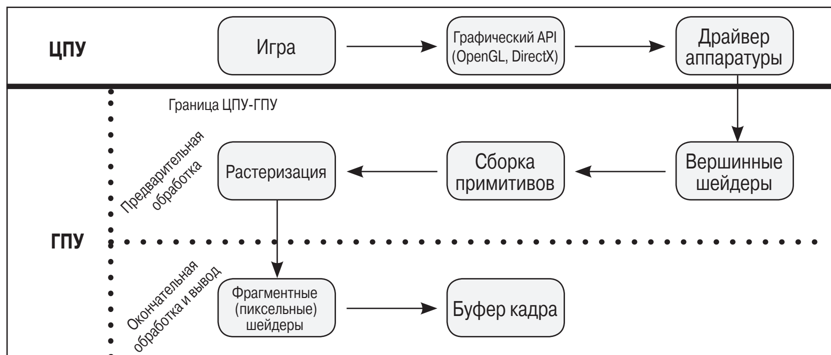


Рис. 6.1 ❖ Диаграмма работы графического конвейера

Верхний ряд отражает работу центрального процессора: вызов программного интерфейса, драйвера аппаратуры и передача команд графическому процессору. Следовательно, производительность центрального процессора главным образом ограничивается сложностью и количественными показателями программного интерфейса.

В то же время производительность графического процессора ограничивается его способностью обрабатывать эти вызовы и опустошать буфер команд в разумные сроки, в соответствии с требуемой частотой кадров. В следующих двух рядах представлены шаги, выполняемые графическим процессором. Из-за сложности устройств их часто делят на две разные части: предварительную и заключительную.

К начальной части процесса отображения относятся: получение мешей, формирование вызовов средств визуализации и передача всей полученной информации вершинному шейдеру. Наконец, механизм растеризации создает пакет фрагментов для обработки в заключительной части. На этом этапе графический процессор выполняет остальную обработку: генерирует фрагменты, проверяет их, выполняет необходимые преобразования, передает их фрагментному шейдеру и далее в буфер кадра в виде пикселей.

Обратите внимание: термин «фрагментный шейдер» является технически более точным, чем термин «пиксельный шейдер». Фрагменты создаются на стадии растеризации и технически становятся пикселями уже после их обработки фрагментным шейдером и помещения в буфер кадра.

Существует несколько подходов к поиску источника проблем отображения графики:

- профилирование графического процессора;
- анализ отдельных кадров с помощью отладчика кадров;
- поиск методом перебора.

Профилирование графического процессора

Поскольку в отображении графики задействованы оба процессора –центральный и графический, для выявления перегруженного компонента следует использовать области **CPU Usage** (Использование центрального процессора) и **GPU Usage** (Использование графического процессора) профилировщика.

Например, на рис. 6.2 представлены результаты профилирования, указывающие на перегруженность центрального процессора. Данное приложение создает тысячи простых объектов без применения методов пакетной обработки. Это приводит к чрезвычайно большому количеству обращений к системе отображения (около 15 000) со стороны центрального процессора, при этом графический процессор загружен незначительно из-за простоты выводимых объектов.

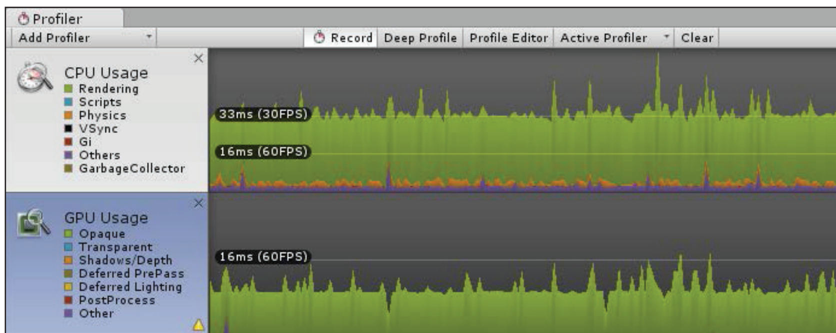


Рис. 6.2 ❖ Результаты профилирования приложения, нагружающего центральный процессор

Данный пример показывает, что большая часть времени, затрачиваемого на решение задачи «отображения», приходится на центральный процессор (около 30 мс на кадр), в то время как графический процессор справляется с обработкой менее чем за 16 мс. То есть узким местом в данном случае является центральный процессор.

Профилирование приложений, где основная нагрузка ложится на графический процессор, осуществляется несколько сложнее. Представьте приложение, создающее небольшое количество объектов с большим количеством полигонов (когда на одну вершину приходится небольшое количество обращений к системе визуализации), использующее десятки точечных источников света, чрезмерно сложный шейдер, обрабатывающий текстуру, текстуры нормалей, карту высот, карту эмиссий, карту окклюзий и т. д. (когда для каждого пикселя выполняется огромный объем операций).

На рис. 6.3 показаны результаты профилирования такой сцены в автономном приложении.

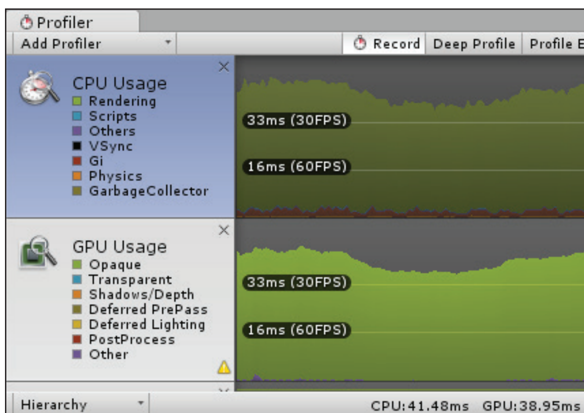
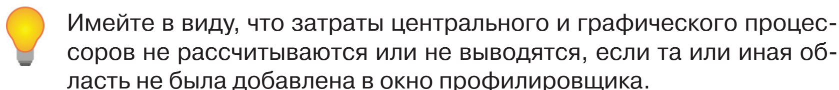


Рис. 6.3 ❖ Результаты профилирования приложения, нагружающего графический процессор

Как видите, график в области **CPU Usage** (Использование центрального процессора) достаточно точно соответствует графику в области **GPU Usage** (Использование графического процессора). Обратите также внимание, что затраты времени центрального и графического процессоров в нижней части окна очень близки (41,48 мс и 38,95 мс). Такой результат кажется странным, потому что графи-

ческому процессору должно было достаться гораздо больше работы, чем центральному процессору.



Имейте в виду, что затраты центрального и графического процессоров не рассчитываются или не выводятся, если та или иная область не была добавлена в окно профилировщика.

А теперь взгляните, что получается, если профилирование той же сцены выполнить в редакторе (рис. 6.4).

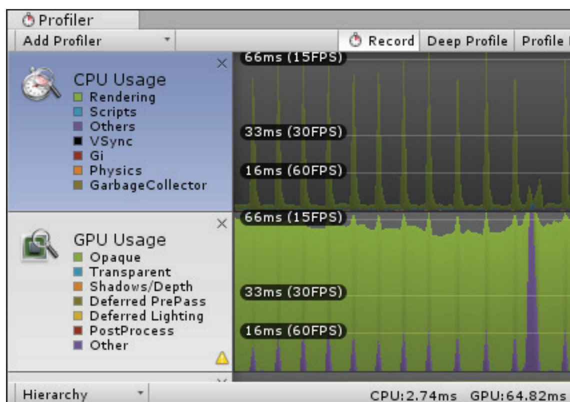
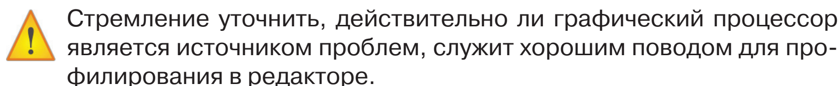


Рис. 6.4 ❖ Результаты профилирования в редакторе

Эти результаты точнее отражают наши ожидания рапределения нагрузки в приложении, нагружающем графический процессор. Как видите, затраты времени центрального и графического процессоров в нижней части окна точно соответствуют нашим предположениям (2,74 мс и 64,82 мс).

Но и эти данные сильно загрязнены. Всплески на грфиках являются результатом обновления окна пользовательского интерфейса профилировщика во время тестирования, а кроме того, накладные расходы редактора также искусственно увеличивают общие затраты времени графического процессора.

Не совсем ясно, что послужило причиной такой интерпретации данных, но это, безусловно, должно измениться в будущих версиях профилировщика, а пока просто учитывайте этот недостаток.



Стремление уточнить, действительно ли графический процессор является источником проблем, служит хорошим поводом для профилирования в редакторе.

Отладка кадров

В Unity 5 появилась новая функция отладки кадров – инструмент, позволяющий определить порядок сборки и отображения сцены отдельными обращениями к системе визуализации. Щелкая на элементах списка, можно наблюдать, как выглядит сцена в тот или иной момент времени. Кроме того, отладка кадров позволяет получить много полезных сведений о выбранном вызове, таких как текущая цель отображения (например, карта теней, текстура глубины камеры, основная камера или прочие нестандартные цели), для чего предназначен вызов (вывод меша, вывод статического пакета, вывод теней глубины и т. д.) и какие параметры использованы (текстура, цвет вершин, встроенные карты освещений, направленное освещение и др.).

На рис. 6.5 показано состояние сцены на момент выбранного вызова в отладчике кадров. Обратите внимание, что тени из встроенной карты освещения отображаются раньше самого объекта.

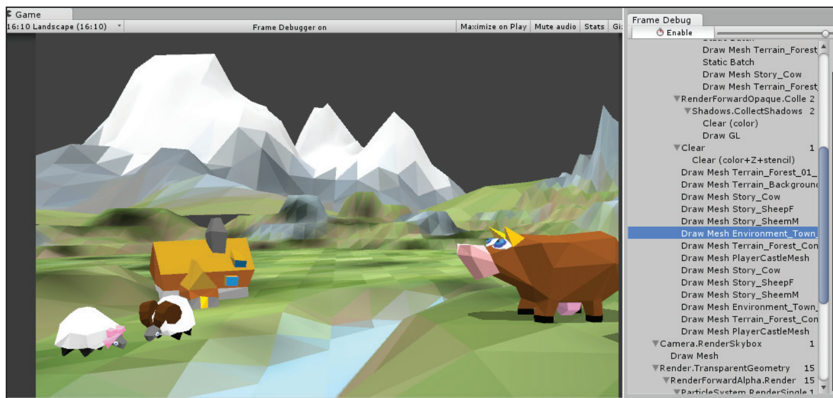


Рис. 6.5 ❖ Состояние сцены на момент выбранного вызова

Если снижение производительности связано с обращениями к системе визуализации, этот инструмент поможет определить, какие вызовы производились, и выяснить, выполнялись ли лишние вызовы, не оказывающие влияния на сцену. Это поможет уменьшить количество вызовов, например путем удаления ненужных объектов или их пакетной обработки. С помощью этого инструмента можно также определить, сколько дополнительных вызовов потребовалось,

например теней, прозрачных объектов и многого другого. Это поможет при создании нескольких уровней качества игры решить, какие функции включить или отключить при низком, среднем и высоком качестве.

Поиск методом перебора

Если при анализе результатов профилирования не удалось найти источник проблем, всегда можно попробовать метод перебора: отключить некоторую активность в сцене и проверить, не привело ли это к значительному росту производительности. Если результатом небольшого изменения станет значительное увеличение скорости, очевидно, что данная активность является узким местом. В этом подходе нет никакого вреда, и если сократить количество неизвестных, анализ данных пойдет в правильном направлении.

Подробнее приемы поиска методом перебора рассматриваются в следующих разделах.

Основная нагрузка приходится на центральный процессор

Если основная нагрузка в приложении приходится на центральный процессор, обычно это характеризуется низкими значениями частоты кадров в области **CPU Usage** (Использование центрального процессора) профилировщика. Однако если включена функция **VSync** (Вертикальная синхронизация), данные часто бывают загрязнены крупными всплесками, отражающими ожидание центральным процессором готовности буфера кадров при синхронизации с частотой обновления кадров экрана. Поэтому, чтобы убедиться, что проблема связана с центральном процессором, отключите функцию **VSync** (рис. 6.6), а затем проанализируйте данные в области **CPU Usage** (Использование центрального процессора).

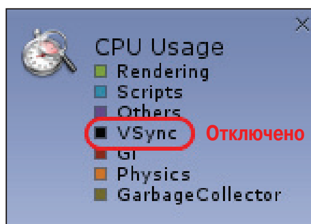


Рис. 6.6 ❖ Отключите функцию **VSync**

Выполняя поиск проблем методом перебора в приложениях, нагружающих центральный процессор, важно сократить количество обращений к системе визуализации. Это требование кажется странным, поскольку для снижения количества вызовов предположительно уже используются такие приемы, как статическая и динамическая пакетная обработка, и возможности дальнейшего сокращения практически исчерпаны.

Однако еще можно отключить все функции, сокращающие количество вызовов системы визуализации, такие как пакетная обработка, и проверить – станет ли ситуация значительно хуже. Если да, это доказывает, что узким местом является центральный процессор. Теперь можно снова включить эти функции и выборочно отключать отображение объектов (предпочтительно с низким уровнем сложности для уменьшения количества обратных вызовов без чрезмерного упрощения отображения сцены). Если производительность значительно улучшится, но вы не сможете найти новых способов пакетной обработки и объединения мешей, к несчастью, придется удалять объекты из сцены, так как только это позволит улучшить производительность.

Существуют и другие возможности уменьшения количества обращений к системе визуализации, включая окклюзивную выбраковку, тонкую настройку освещения и теней, модификацию шейдеров. Они будут рассмотрены в следующих разделах.

Однако система визуализации Unity может работать в многопоточном режиме, в зависимости от целевой платформы, версии Unity и различных настроек. И это может повлиять на общую производительность приложений, в которых узким местом является центральный процессор, и изменить оценку нагруженности центрального процессора.

Многопоточный вывод

Поддержка многопоточного режима впервые появилась в версии Unity 3.5, вышедшей в феврале 2012 года, и включала по умолчанию поддержку многоядерных систем, которые могли бы справиться с требуемой нагрузкой. На то время такими системами были персональные компьютеры, Mac и Xbox 360. Постепенно в этот список были добавлены другие устройства. Начиная с версии Unity 5.0 поддержка многопоточного режима по умолчанию включена для всех основные платформ (и возможно, в некоторых сборках Unity 4).

Мобильные устройства также начинают комплектоваться более мощными процессорами, поддерживающими эту функцию. Поддерж-

ку многопоточного режима в Android (появилась в Unity 4.3) можно включить с помощью флага **Platform Settings ⇨ Other Settings ⇨ Multithreaded Rendering** (Параметры платформы ⇨ Другие параметры ⇨ Многопоточное отображение). Многопоточный режим в iOS можно включить настройкой поддержки программного интерфейса Apple Metal API (появилась в версии Unity 4.6.3), выбрав **Player Settings ⇨ Other Settings ⇨ Graphics API** (Настройки игрока ⇨ Другие параметры ⇨ Графический API).

При включении многопоточного режима отображения задачи, выполняемые через программные интерфейсы отображения (OpenGL, DirectX или Metal), передаются из главного потока в «рабочий». Назначением рабочего потока является принятие на себя тяжелой нагрузки по передаче команд отображения через графический программный интерфейс и драйвер в буфер команд графического процессора. Это может сэкономить немало ресурсов центрального процессора и освободить его для решения других задач в основном потоке. Это означает высвобождение дополнительных ресурсов для обработки движком физических моделей, выполнения кода сценариев и т. д.

Кстати, механизм, с помощью которого основной поток уведомляет рабочий поток о заданиях, действует подобно буферу команд графического процессора, но передает команды более высокого уровня, такие как «отобразить заданный объект с заданным материалом, используя заданный шейдер» или «отобразить *N* экземпляров заданного фрагмента», и т. д. Эта возможность появилась в Unity 5 и позволяет разработчикам напрямую управлять подсистемой отображения из кода на C#. Правда, эти возможности не настолько мощные, как при прямом доступе к программному интерфейсу, тем не менее это шаг в правильном направлении, позволяющий Unity-разработчикам реализовывать уникальные графические эффекты.

По непонятной причине, этот программный интерфейс называется «CommandBuffer», поэтому не путайте его с буфером команд GPU .

Более подробную информацию о буфере команд CommandBuffer можно найти в документации Unity: http://docs.Unity3D.com/ScriptReference/Rendering.CommandBuffer.HTML.

Возвращаясь к теме нехватки ресурсов центрального процессора, следует заметить, что всегда нужно учитывать факт использования многопоточности, поскольку причины низкой производительности несколько отличаются в зависимости от того, включена эта функция или нет.

В однопоточном режиме все вызовы графических программных интерфейсов обрабатываются главным потоком, и в идеальном случае, когда оба компонента работают на пределе своих возможностей, узким местом приложения будет центральный процессор, если 50 и более процентов времени в кадре отводится на работу с графическим программным интерфейсом. Избавиться от этих трудностей можно за счет уменьшения нагрузки на основной поток. Например, значительное сокращение объема работы, приходящегося на долю подсистемы искусственного интеллекта, позволит очень сильно улучшить производительность отображения, поскольку приведет к высвобождению ресурсов центрального процессора для работы с графическим интерфейсом.

Но в многопоточном режиме эта задача возлагается на рабочий поток, а это означает, что работой движка и вызовами графического интерфейса управляют разные потоки. Это полностью независимые процессы, и даже при том, что основной поток все еще выполняет определенную работу по передаче инструкций в рабочий поток (через внутреннюю систему CommandBuffer), эта работа очень незначительная. То есть сокращение объема работы в основном потоке практически не повлияет на производительность отображения.

Обратите внимание, что проблемы, вызванные перегрузкой графического процессора, также не зависят от многопоточного режима отображения.

Скининг графическим процессором

Обсуждая задачу снижения нагрузки на центральный процессор, следует упоямнуть еще об одной возможности – передаче скининга графическому процессору. **Скининг (skinning)** – это процесс трансформации вершин меша на основании текущего положения его анимированных костей. Система анимации, выполняющаяся на центральном процессоре, трансформирует только кости. На следующем шаге отображения нужно позаботиться о трансформации меша, окружающего кости, вычислив средневзвешенные координаты его вершин относительно костей.

Задачу обработки вершин может решать либо центральный, либо графический процессор, в зависимости от значения параметра **GPU Skinning** (Скининг графическим процессором). Переключение можно осуществить с помощью параметра **Edit ⇨ Project Settings ⇨ Player Settings ⇨ Other Settings ⇨ GPU Skinning** (Правка ⇨ Пара-

метры проекта ⇨ Параметры игрока ⇨ Другие параметры ⇨ Скининг графическим процессором).

Узкие места на этапе предварительной обработки

В *главе 4* «*Привнесение искусства*» мы уже рассмотрели некоторые методы оптимизации мешей, позволяющие уменьшить количество атрибутов вершин. Кратко напомню, что нередко используются меши, содержащие массу ненужных данных UV-векторов и векторов нормалей, поэтому их следует тщательно проверить на наличие подобных излишеств. Нужно позволить Unity оптимизировать структуру, что приведет к уменьшению количества промахов кэша при чтении вершин на этапе предварительной обработки.

Ниже, рассматривая этап заключительной обработки и отображения, мы познакомимся с методами оптимизации шейдеров, поскольку многие технологии оптимизации применяются к вершинным и фрагментным шейдерам.

Все такие методы основываются на поиске путей сокращения количества вершин. Очевидным решением являются упрощение и выбраковка, то есть либо команда художников заменяет проблемные меши их версиями с меньшим количеством полигонов, либо для уменьшения общего количества полигонов следует удалить некоторые объекты из сцены. Если использование этих подходов не привело к нужному результату, последним средством остается только поиск компромиссного решения.

Уровень детализации

Поскольку на удалении трудно различить, отображается ли объект в высоком или низком качестве, нет особых причин использовать для таких объектов высококачественные версии. Так почему бы не заменять динамически удаленные объекты их упрощенными версиями?

Под термином **Уровень детализации (LOD)** понимаются широкие возможности динамической замены в зависимости от удаленности относительно камеры. Наиболее часто используется механизм уровней детализации, основанный на мешах: динамическая замена меша на его все менее и менее детализированные версии по мере удаления от камеры. Еще одним примером может служить замена находящихся

вдали анимированных персонажей их версиями с меньшим количеством костей для сокращения затрат на анимацию.

Встроенная поддержка уровней детализации доступна в редакции Pro Unity 4 и во всех редакциях Unity 5. Однако при желании ее можно реализовать с помощью сценария и в редакции Free Unity 4.

При использовании механизма уровней детализации в сцене размещается несколько объектов, присоединяемых к игровому объекту в качестве дочерних через компонент `LODGroup`. Этот компонент формирует прямоугольники, ограничивающие объекты, и выбирает, какой объект выводить, опираясь на размер ограничивающего прямоугольника в поле зрения камеры. Если ограничивающий прямоугольник объекта занимает большую часть видимой области, выбираются меши из нижних групп уровней детализации, а если ограничивающий прямоугольник очень мал, выбираются меши из верхних групп. Если меш находится слишком далеко, можно настроить скрытие всех дочерних объектов. То есть при правильной настройке Unity может использовать более простые версии мешей и исключать их полностью для уменьшения затрат на отображение.

Более подробную информацию о функции детализации уровней можно найти в документации: http://docs.unity3d.com/Manual/LevelOfDetail.html.

Использование этой функции может значительно увеличить время разработки, поскольку художникам придется сгенерировать версии с малым количеством полигонов, а дизайнерам уровней – создать группы детализации, настроить и проверить их, чтобы убедиться, что переходы не вызывают дрожания при удалении и приближении камеры. Кроме того, при таком подходе увеличивается потребление памяти и ресурсов центрального процессора, так как альтернативные меши хранятся в памяти, а компонент `LODGroup` должен регулярно проверять, не переместилась ли камера в новое положение, чтобы гарантировать соответствующее изменение уровней детализации.

В настоящее время возможности графических карт достигли такого уровня, что обработка вершин является не самой важной проблемой. Вместе с дополнительными затратами, сопровождающими применение механизма детализации уровней, разработчикам следует избегать предварительной оптимизации и положиться на механизм детализации уровней. Злоупотребление этим механизмом приведет к потерям производительности в других частях приложения и бес-

смысленной потере времени при разработке. Если не доказано, что именно в этом состоит проблема, это не является проблемой!

Для сцен, отображающих обширные пространства игрового мира при частых перемещениях камеры, следует изначально предусмотреть применение этого метода, поскольку при увеличении расстояния и значительном росте количества видимых объектов произойдет резкое увеличение числа вершин. В сценах, ограниченных помещением, и в сценах, где камера направлена вниз на игровой мир (например, стратегии в реальном времени или боевые многопользовательские игры), следует воздержаться от использования уровней детализации. В играх, занимающих промежуточное положение, следует избегать применения этой функции, пока в ней не возникнет острая необходимость. Все зависит от того, сколько вершин будет видно в любой момент времени и как часто изменяется расстояние до камеры.

Обратите внимание, что некоторые компании, разрабатывающие дополнительное программное обеспечение для разработчиков игр, предлагают инструменты автоматизированного создания мешей для уровней детализации. С ними стоит познакомиться и оценить простоту их применения в сравнении с потерями качества и дополнительными затратами.

Отключение скининга графическим процессором

Как уже говорилось, Unity поддерживает выполнение скининга графическим процессором для снижения нагрузки на центральный процессор за счет увеличения нагрузки на графический процессор. Так как скининг является одним из «впечатляюще параллельных» процессов, которые хорошо сочетаются с параллельной архитектурой графического процессора, передача этой задачи ему часто дает хороший выигрыш. Но на выполнение этой задачи будет тратиться драгоценное время, необходимое на определение вершин фрагментов, поэтому отключение обработки скининга графическим процессором является еще одним направлением для исследования, если узкое место обнаружится в этой области. Отключить скининг графическим процессором можно с помощью параметра **Edit ⇨ Project Settings ⇨ Player Settings ⇨ Other Settings ⇨ GPU Skinning** (Правка ⇨ Параметры проекта ⇨ Параметры игрока ⇨ Другие параметры ⇨ Скининг графическим процессором).

Параметр GPU Skinning доступен в редакции Pro Unity 4 и во всех редакциях Unity 5.

Уменьшение тесселяции

Еще одна задача, выполняющаяся на стадии предварительной обработки, которую следует рассмотреть, – тесселяция. Тесселяция с применением геометрических шейдеров является впечатляющей технологией. Эта часто недооцениваемая технология способна производить незабываемые графические эффекты, которые помогут выделить вашу игру на фоне других игр, использующих только распространенные эффекты. Однако она также привносит огромный объем затрат на обработку на стадии предварительной обработки.

Не существует простых приемов ускорения тесселяции, кроме оптимизации алгоритмов или уменьшения затрат на другие задачи предварительного этапа, чтобы выделить больше ресурсов для тесселяции. В любом случае, если узкое место обнаружится на предварительном этапе, где выполняется тесселяция, следует проверить – не она ли потребляет львиную долю ресурсов.

Узкие места на этапе окончательной обработки

Этап окончательной обработки является наиболее интересной частью конвейера графического процессора, так как здесь создается подавляющее большинство графических эффектов. Следовательно, на этом этапе значительно чаще возникают узкие места.

Здесь можно использовать два пути:

- уменьшить разрешение;
- уменьшить качество текстур.

Эти изменения ослабят рабочую нагрузку на двух важных стадиях заключительной части конвейера, увеличив скорость заполнения и пропускную способность памяти соответственно. В настоящее время скорость заполнения является наиболее распространенной причиной падения производительности при отображении графики, поэтому начнем с нее.

Скорость заполнения

Благодаря уменьшению разрешения экрана системе растеризации придется создавать значительно меньше фрагментов и транспонировать их на меньший по размерам холст пикселей. Это позволит увеличить скорость заполнения и ослабит нагрузку на ключевую часть конвейера визуализации. Следовательно, если при уменьшении раз-

решения экрана резко увеличится производительность, значит, дело в скорости заполнения.

Скорость заполнения охватывает весьма широкий круг понятий, относящихся к скорости, с которой графический процессор способен отображать фрагменты. Но это касается только фрагментов, которые благополучно прошли все проверки на протяжении выполнения заданного шейдера. Фрагмент – это «потенциальный пиксель», и если любая из его проверок окончится неудачей, он немедленно отбрасывается. Это значительно увеличивает производительность, так как конвейер пропускает затратную операцию отображения и переходит к следующему фрагменту.

Примером является Z-тестирование, проверяющее присутствие перед фрагментом данного объекта фрагмента другого, более близкого объекта. Если это так, текущий фрагмент отбрасывается. Если нет, фрагмент передается фрагментному шейдеру и отображается в пиксели целевого устройства. При этом только данный фрагмент повлияет на скорость заполнения. А теперь умножьте эти затраты на тысячи перекрывающихся объектов с сотнями и тысячами фрагментов. При высоком разрешении экрана это составит миллионы или миллиарды фрагментов. Становится очевидно, что пропуск большинства из них приведет к существенному снижению затрат на отображение.

Производители графических карт обычно указывают в рекламных материалах конкретную скорость заполнения, обычно в гигапикселях в секунду, но это несколько неправильно, так как точнее будет сказать, что это скорость в гигафрагментах в секунду. Однако это чисто академический аргумент. В любом случае, большие значения скорости заполнения говорят о том, что устройство потенциально может пропустить через конвейер больше фрагментов. Так, при ограничении в 30 гигапикселей в секунду и частоте кадров целевого устройства в 60 Гц можно позволить себе передачу $30\,000\,000\,000/60 = 500$ миллионов фрагментов на кадр, прежде чем скорость заполнения станет узким местом. При разрешении 2560×1440 и наилучшем стечении обстоятельств, когда каждый пиксель рисуется только один раз, теоретически можно вывести всю сцену около 125 раз без заметных проблем.

К сожалению, этот мир не идеален, и если не предпринять значительных усилий, некоторое количество пикселей в конечном итоге будет выводиться повторно, из-за нарушения порядка отображения объектов. Эта ситуация называется **овердрафтом** (overdraw) и может вызвать существенные затраты, если не проявить должной осторожности.

Причиной, почему изменение разрешения является действенным средством для выявления проблем, связанных со скоростью заполнения, является умножение. Уменьшение разрешения с 2560 × 1440 до 800 × 600 приведет к уменьшению затрат примерно в восемь раз, что может сделать скорость заполнения вполне достаточной для нормального выполнения приложения.

Овердрафт

Визуально оценить затраты на овердрафт позволит отображение объектов с добавлением альфа-канала и применением однотонного прозрачного цвета. Области большего овердрафта будут выглядеть ярче из-за аддитивного наложения полупрозрачных пикселей. Именно так действует режим заливки **Overdraw** (Овердрафт) в панели **Scene** (Сцена), демонстрирующий, насколько сцена пострадала от овердрафта.

На рис. 6.7 показана сцена с несколькими тысячами кубиков и отображение той же сцены в режиме заливки **Overdraw**.

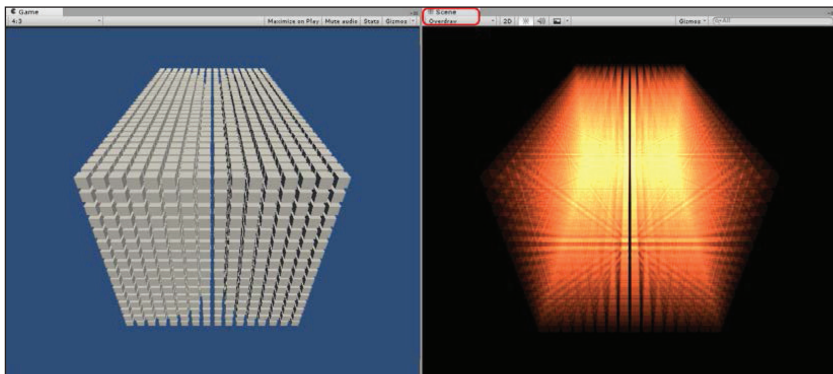


Рис. 6.7

Таким образом, скорость заполнения преподносится как оценка качества. Но это всего лишь маркетинговый термин и в значительной степени является теоретическим. Однако производители используют его как технический для описания производительности этапа окончательной обработки конвейера, на котором фрагменты обрабатываются шейдерами и отображаются на экране.

Если каждый фрагмент требует лишь минимальной обработки (например, когда шейдер возвращает один и тот же цвет), можно вплот-

ную приблизиться к этому теоретическому максимуму. Графический процессор – сложное устройство, но он и не может быть простым. Его особенностью является способность лучше справляться со множеством небольших заданий. Но если задания слишком велики, скорость заполнения падает, так как этап окончательной обработки не в состоянии за раз обработать нужное количество фрагментов, и весь конвейер замирает в ожидании.

Имеется еще несколько дополнительных функций, применение которых может помочь приблизиться к теоретическому максимуму, включая альфа-тестирование, альфа-смешивание, дискретизацию текстур, уменьшение объема данных для описания фрагментов, пропускаемых через шейдеры, и даже смену формата цвета целевой отображаемой текстуры (обычно заключительный буфер кадров), и это не полный список. Но проблема в том, что придется охватить слишком много путей, из-за чего увеличивается риск нарушить весь процесс, однако при этом мы получаем широкие возможности для изучения методов улучшения скорости заполнения.

Окклюзивная выбраковка

Одним из лучших способов для уменьшения овердрафта является использование системы окклюзивной выбраковки в Unity. Она разделяет пространство сцены на серии ячеек и исследует игровой мир с помощью виртуальной камеры, определяя окклюзивные клетки, то есть невидимые из других клеток, опираясь на размеры и положение имеющихся объектов.

Обратите внимание, что окклюзивная выбраковка отличается от отсекающей выбраковки (по пирамиде видимости), когда выбраковываются объекты, не видимые из текущего положения камеры. Эта функция активна всегда и во всех версиях, и с ее помощью выбракованные объекты автоматически игнорируются системой окклюзивной выбраковки.

Окклюзивная выбраковка доступна в редакции Pro Unity 4 и во всех редакциях Unity 5.

Данные окклюзивной выбраковки могут быть сгенерированы только для объектов, отмеченных как **Occluder Static** (Статический, скрывающий и скрываемый при окклюзии) и **Occludee Static** (Статический, только скрываемый при окклюзии) в раскрывающемся списке **Static Flags** (Флаги статичности). Флаг **Occluder Static** обычно устанавливается для статических объектов, если нужно, чтобы он

скрывал другие объекты и был скрыт перекрывающими его большими объектами. Флаг **Occludee Static** представляет особый случай для прозрачных объектов, он позволяет отображать другие объекты позади и скрывать данный объект, если что-то большее загораживает его.

Естественно, из-за необходимости установки флагов статичности окклюзивная выбраковка неприменима к динамическим объектам.

Рисунок 6.8 демонстрирует, насколько эффективно окклюзивная выбраковка может сократить количество видимых объектов в сцене.

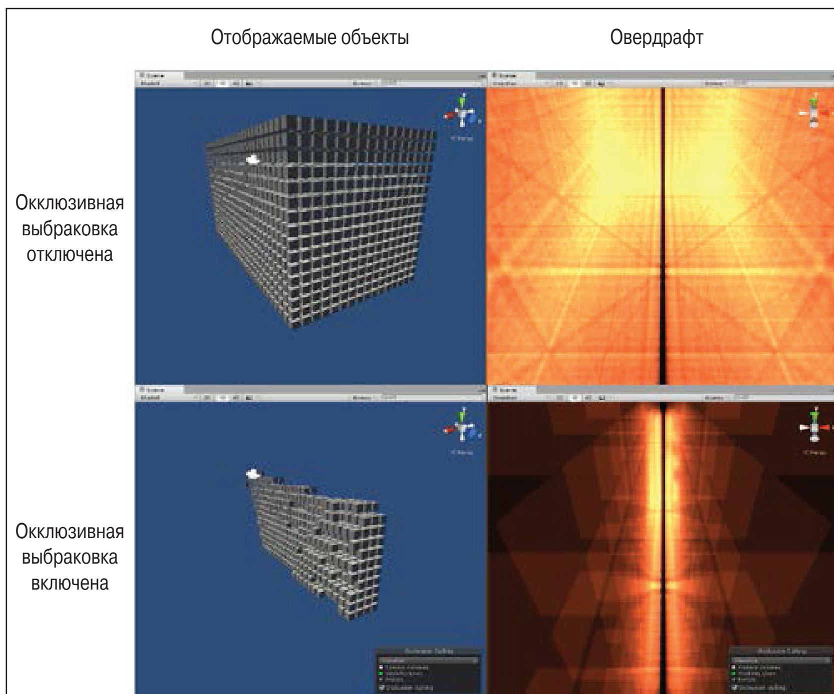


Рис. 6.8 ❖ Эффективность окклюзивной выбраковки

Применение этой системы увеличивает потребление памяти и вычислительных ресурсов во время выполнения. Память расходуется на хранение структуры данных для окклюзивной выбраковки, а вычислительные ресурсы – на определение объектов, перекрытых в каждом кадре.

Структура данных окклюзивной обработки должна быть правильно настроена для создания ячеек с размером, соответствующим сцене,

и чем меньше размер ячеек, тем больше времени расходуется на создание структуры данных. Но при правильной настройке для сцены окклюзивная выбраковка может увеличить скорость заполнения за счет уменьшения овердрафта и уменьшить количество обращений к системе визуализации за счет выбраковки невидимых объектов.

Оптимизация шейдеров

Шейдеры способны значительно снизить скорость заполнения в зависимости от их сложности, количества текстур, числа используемых математических функций и т. д. Шейдеры влияют на скорость заполнения не напрямую, а косвенно, из-за того, что при выполнении шейдеров графическому процессору приходится производить вычисления или получать данные из памяти. Параллельная природа графического процессора означает, что любое узкое место в потоке будет ограничивать количество фрагментов, передаваемых последующим потокам, но параллельная обработка (распределение заданий небольшими фрагментами между несколькими потоками) обеспечивает ускорение относительно последовательной обработки (когда один поток обрабатывает задания одно за другим).

Классическим примером является сборочная линия автомобилей. Полный цикл производства автомобиля состоит из нескольких этапов. Минимальный завершенный цикл включает пять этапов: штамповку, сварку, окраску, сборку и контроль, – и каждый этап может выполняться только одной бригадой. Для каждого отдельного автомобиля следующий этап не может начаться до полного завершения предыдущего, но бригада, закончив штамповку деталей кузова для одного автомобиля, может начать штамповку для следующего. Такая организация позволяет членам каждой бригады стать мастерами в своей области, а попытка занять их чем-то другим приведет, скорее всего, к ухудшению качества партии автомобилей.

Можно удвоить общий результат, удвоив количество бригад, но если любая бригада задержит свою работу, будет потеряно драгоценное время для производства данного автомобиля, а также замедлится будущее производство всех автомобилей, которым будет заниматься эта бригада. Если такие задержки редки, они мало влияют на процесс производства в целом, но если они регулярны и один этап занимает на несколько минут больше, чем положено по норме, это может стать узким местом, угрожающим срыву сроков выпуска всей партии.

Графические процессоры работают похожим образом: каждый поток выполнения можно сравнить со сборочной линией, каждый этап

обработки – с бригадой, а каждый фрагмент – с автомобилем. Если поток тратит слишком много времени на выполнение одного этапа, то задерживается обработка всех фрагментов. Эта задержка будет умножаться, поскольку все фрагменты, которые в дальнейшем пройдут через этот поток, будет задержаны. Это весьма упрощенный подход, но он часто помогает проиллюстрировать, как плохо оптимизированный код шейдера может снизить скорость заполнения и как небольшая оптимизация обеспечивает улучшение производительности заключительной части конвейера.

Программирование и оптимизация шейдеров занимают особое место в области разработки игр. Их абстрактный и узкоспециальный характер требуют мышления определенного рода, отличающегося от того, который требуется для написания кода игрового процесса и для взаимодействия с движком. В шейдерах часто используются необычные математические приемы и обходные пути для обработки данных, например предварительная компиляция значений из файлов текстур. Из-за этого и из-за важности их оптимизации шейдеры, как правило, очень трудно читать и модернизировать.

Поэтому многие разработчики пользуются предустановленными шейдерами или инструментами визуальной разработки шейдеров из Asset Store, такими как Shader Forge или Shader Sandwich. Они упрощают создание начального кода шейдера, который, впрочем, может оказаться не самым эффективным. При использовании предварительно подготовленных шейдеров или созданных с помощью инструментов стоит попытаться оптимизировать их с помощью нескольких проверенных и надежных методов. Итак, рассмотрим несколько способов оптимизации шейдеров.

Шейдеры для мобильных платформ

Мобильные шейдеры, встроенные в Unity, не имеют каких-либо конкретных ограничений, обязывающих их использовать только на мобильных устройствах. Они просто оптимизированы в расчете на минимальные ресурсы (нередко с применением прочих способов оптимизации, перечисленных в этом разделе).

Настольные приложения могут использовать эти шейдеры, но их применение, как правило, влечет потерю качества графики. Дело здесь заключается только в приемлемости снижения качества графики. Поэтому попробуйте поэкспериментировать с мобильными эквивалентами обычных шейдеров, чтобы определить, насколько хорошо они соответствуют вашей игре.

Использование типов данных малого размера

Графический процессор производит вычисления с типами данных малого размера быстрее, чем с типами данных большого размера (особенно на мобильных платформах!), поэтому прежде всего стоит попробовать заменить тип данных float (32-битный, с плавающей точкой) на half (16-битный, с плавающей точкой) или даже fixed (12-битный, с фиксированной точкой).

Размер перечисленных выше типов данных может меняться в зависимости от форматов чисел с плавающей запятой, поддерживаемых целевой платформой. Указанные размеры являются наиболее распространенными. Большое значение для оптимизации имеет относительный размер форматов.

Хорошими кандидатами для понижения точности являются значения цвета, поскольку это не привносит заметных изменений в окраску. Но в графических вычислениях последствия снижения точности весьма непредсказуемы. Поэтому после подобных изменений желательно проверить, повлияло ли снижение точности на правильность отображения графики.

Обратите внимание, что последствия могут значительно отличаться для разных по архитектуре графических процессоров (например, AMD, Nvidia и Intel) и даже для разных графических процессоров одного и того же производителя. В некоторых случаях можно добиться достаточного роста производительности, потратив не слишком много усилий. В других случаях прирост вообще не будет заметен.

Избегайте изменения точности при swizzling-присваивании

Swizzling-присваивание – это прием создания нового вектора (массива значений) из существующего путем перечисления компонентов в том порядке, в каком они должны быть скопированы в новую структуру. Ниже приводится несколько примеров swizzling-присваивания:

```
// инициализация тестового значения
float4 input = float4(1.0, 2.0, 3.0, 4.0);

// swizzling-присваивание двух компонентов
float2 val1 = input.yz;

// swizzling-присваивание трех компонентов в другом порядке
float3 val2 = input.zyx;
```

```
// swizzling-присваивание одного компонента несколько раз
float4 val3 = input.yyy;

// swizzling-присваивание скалярного значения несколько раз
float sclr = input.w;
float3 val4 = sclr.xxx
```

Для обозначения одних и тех же компонентов можно использовать любое из представлений: `xyzw` и `rgba`. Не важно, что это – цвет или вектор; эти обозначения предназначены только для облегчения чтения кода шейдера. Просто перечислите компоненты в требуемом порядке, повторяя их при необходимости.

Приведение типов в шейдерах вообще является достаточно затратной операцией, а приведение типов при swizzling-присваивании особенно нежелательно. При наличии математических операций, использующих swizzling-присваивание двух разных типов точности, разумнее повсеместно перейти на использование единого типа, чтобы избежать необходимости приведения.

Используйте оптимизированные вспомогательные функции

Компилятор шейдеров часто стремится сократить объем математических вычислений, чтобы получить код, оптимизированный для графического процессора. Но скомпилированный пользовательский код вряд ли получится столь же эффективным, как вспомогательные функции из встроенной библиотеки Cg и предоставляемые в Unity в виде подключаемых файлов. При использовании шейдеров, включающих код пользовательской функции, иногда в библиотеке Cg или в Unity можно найти эквивалентную вспомогательную функцию, которая справится с задачей лучше.

Эти подключаемые файлы можно добавить в шейдер с помощью блока `CGPROGRAM` следующим образом:

```
CGPROGRAM
// подключение других файлов
#include "UnityCG.cginc"
// Здесь размещается код шейдера
ENDCG
```

Примерами функций из библиотеки Cg могут служить: `abs()`, возвращающая абсолютное значение, `lerp()`, выполняющая линейную интерполяцию, `mul()`, перемножающая матрицы, и `step()`, обеспечивающая пошаговое выполнение алгоритма. Из функций `UnityCG.cginc`

можно отметить `WorldSpaceViewDir()`, выполняющую расчет направления камеры, и `Luminance()`, преобразующую цвета в оттенки серого.

Полный список функций из библиотеки Cg можно найти по адресу: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html.

Полный актуальный список подключаемых файлов и предоставляемых ими вспомогательных функций можно найти в документации Unity по адресу: http:// docs.unity3d.com/Manual/SL-BuiltinIncludes.html.

Отключайте ненужные механизмы

Иногда можно сэкономить ресурсы, просто отключив шейдерные механизмы, которые не являются жизненно важными. Действительно ли шейдеру требуется выполнить несколько проходов, анализировать прозрачности, осуществлять Z-запись, проводить альфа-тестирование и/или альфа-смешивание? Даст ли тонкая настройка этих параметров или полное отключение этих механизмов желаемый эффект без потери качества отображения графики? Такие изменения порой помогают сохранить скорость заполнения на требуемом уровне.

Удалите ненужные входные данные

Иногда в процессе написания шейдера приходится многократно изменять код и просматривать результат в сцене. В результате некоторые входные данные, которые были необходимы в начале работы, могут стать ненужными после достижения цели, и об этом легко забыть, когда и если процесс затягивается на длительное время. Однако эти избыточные данные вынуждают графический процессор тратить драгоценное время на их извлечение из памяти, хотя они не используется шейдером. Поэтому тщательно проверьте шейдер, чтобы удостовериться, что все входные данные геометрических форм, вершин и фрагментов действительно используются.

Экспортируйте только необходимые переменные

Экспортирование ненужных переменных из шейдера для сопровождающих материалов может увеличивать накладные расходы, так как графический процессор не сможет рассматривать их значения как константы. Из-за этого код шейдера будет скомпилирован в менее оптимальную форму. Эти данные будут пересылаться центральным процессором в каждом проходе, чтобы их можно было изменить в любое время с помощью методов материалов, таких как `SetColor()`,

`SetFloat()` и др. Если до конца проекта такие переменные содержат одни и те же значения, их следует заменить в шейдере константами, чтобы исключить избыточную нагрузку. Единственным недостатком является неясность, понадобятся ли кому-нибудь параметры графического эффекта, поэтому данная оптимизация должна выполняться в самом конце процесса.

Снижайте математическую сложность

Сложные математические вычисления могут значительно замедлить процесс отображения, поэтому сделайте все возможное, чтобы смягчить потери. Сложные математические функции можно заменить текстурой, передаваемой в шейдер, и предварительно созданными таблицами для поиска во время выполнения. Подмена функций, таких как `sin` и `cos`, не даст заметных улучшений, поскольку они хорошо оптимизированы под архитектуру графического процессора, но сложные методы, такие как `pow`, `exp`, `log`, и другие пользовательские математические процедуры можно значительно оптимизировать, и поэтому они являются очевидными кандидатами для упрощения. Для этого потребуется только одно или два входных значения, представляющих координаты текстуры *X* и *Y*, причем математическая точность не имеет первостепенного значения.

При этом будет затрачиваться дополнительная графическая память для хранения текстур во время выполнения (подробнее об этом позже), но если шейдер уже получает текстуры (как это бывает обычно) и альфа-канал не используется, можно передать данные через текстуру с альфа-каналом, что практически не влечет дополнительных затрат, не усложняет код шейдера и графическую систему. Однако для этого понадобится произвести настройку графических ресурсов и включения данных в неиспользуемые каналы цвета, что потребует определенной координации между программистами и художниками, но в результате вы получите очень хороший способ экономии, не увеличивающий затрат на обработку во время выполнения.

В самом деле, свойства материалов и текстур являются отличными отправными точками для избавления шейдера (графического процессора) от ненужных вычислений. Если сложные вычисления не требуют изменения пикселей, можно экспортировать значение как свойство материала и изменять его при необходимости (с учетом накладных расходов, рассмотренных в предыдущем разделе «*Экспортируйте только необходимые переменные*»). Если пиксели изменяются, но не часто, в коде сценария можно создать файл текстуры,

содержащий результаты расчетов в виде RGBA-значений, и передать текстуру в шейдер. Много возможностей возникает, если отвлечься от обычного применения таких систем и рассматривать их только как средство передачи данных.

Сократите поиск в текстурах

Поиск в текстурах не является тривиальной задачей для графического процессора и увеличивает накладные расходы, которые являются наиболее распространенной причиной проблем, особенно если шейдер делает выборку из нескольких текстур или даже несколько выборок из одной текстуры, поскольку это обычно приводит к промахам кэша. Такие ситуации следует упростить, насколько это возможно, чтобы избежать превращения памяти графического процессора в узкое место.

Хуже того, выборка из текстур в случайном порядке обычно приводит к очень затратным промахам кэша графического процессора, поскольку после этого текстуры должны быть перераспределены так, чтобы их можно было выбирать в нужном порядке.

Избегайте условных операторов

Современные центральные процессоры применяют множество методик прогнозирования для обработки условных операторов, чтобы добиться параллелизма на уровне инструкций. Это выражается в попытках центрального процессора предсказать вероятное направление прохождения через условный оператор до его фактического выполнения, из-за чего начинается обработка наиболее вероятного результата условного оператора с помощью компонентов без накладных расходов, которые не используются для вычисления условий (извлечение данных из памяти, копирование чисел с плавающей точкой в неиспользуемые регистры и т. д.). Если выяснится, что прогноз неверен, текущий результат отбрасывается, а выполнение продолжается в правильном направлении.

Поэтому пока затраты на движение по ложному пути меньше времени ожидания до определения правильного направления, что, как правило, и происходит, это приводит к увеличению скорости вычислений.

Однако параллельная природа графических процессоров имеет иное устройство. Ядра графического процессора управляются конструкциями более высокого уровня, которые указывают всем подчиненным ядрам одновременно выполнять одни и те же машинные

инструкции. Поэтому, если фрагментному шейдеру потребуется умножить на 2 значение типа float, процесс начинается с копирования всеми ядрами данных в соответствующие регистры в одном скоординированном шаге. Только после того, как все ядра закончат копирование регистров, им будет поручено перейти ко второму этапу: умножению всех регистров на 2.

То есть когда эта система натыкается на условный оператор, она не может обработать два оператора независимо друг от друга. Она должна определить, какие из ее дочерних ядер будут задействованы для одной ветви условного оператора, прочитать из памяти машинные инструкции для каждой ветви, определить порядок их обработки ядрами и повторить эту последовательность действий для каждой ветви, пока не будут обработаны все возможные пути. Так, для оператора if-else (две ветви) нужно указать одной группе ядер, обработать ветвь «true», а остальным ядрам – ветвь «false». То есть всякий раз должны быть обработаны оба направления.

По этой причине следует избегать ветвления и условных операторов в коде шейдера. Конечно, все зависит от того, насколько условие важно для достижения нужного графического эффекта. Но если пиксели не зависят от условия, часто лучше отказаться от ненужных математических вычислений, чем добавить затраты на ветвление в графическом процессоре. Например, можно проверить, является ли значение ненулевым, прежде чем использовать его в вычислениях или сравнить некоторые глобальные флаги материала перед выбором того или другого действия. В обоих случаях будет достигнута оптимизация путем отказа от условных проверок.

Уменьшайте зависимость данных

Компилятор пытается оптимизировать код шейдера, переводя его на низкоуровневый язык, более дружественный графическому процессору, не предполагая извлечения данных при обработке прочих задач. Например, ниже приводится плохо оптимизируемый код шейдера:

```
float sum = input.color1.r;
sum = sum + input.color2.g;
sum = sum + input.color3.b;
sum = sum + input.color4.a;
float result = calculateSomething(sum);
```

При компиляции шейдера в машинные инструкции выяснится, что этот код содержит зависимости данных, и каждая следующая операция не может начаться, пока не завершится предыдущая, из-за зависи-

мости от переменной sum. Но такие ситуации часто обнаруживаются компилятором шейдеров и оптимизируются в версии, использующие параллелизм на уровне инструкций (следующий код является эквивалентом получившегося машинного кода):

```
float sum1, sum2, sum3, sum4;
sum1 = input.color1.r;
sum2 = input.color2.g;
sum3 = input.color3.b
sum4 = input.color4.a;
float sum = sum1 + sum2 + sum3 + sum4;
float result = CalculateSomething(sum);
```

В этом случае компилятор определит, что можно извлечь четыре значения из памяти параллельно и выполнить суммирование всех четырех независимо выбранных значений параллельно на уровне потоков. Это сэкономит много времени относительно выполнения четырех извлечений друг за другом.

Однако длинные цепочки зависимостей данных пагубно влияют на производительность шейдеров. Создание сложных зависимостей в исходном коде шейдера делает невозможной такую оптимизацию. Например, следующие зависимости отрицательно сказываются на производительности, поскольку ни один шаг не может быть начат без ожидания выборки данных и выполнения соответствующих расчетов.

```
float4 val1 = tex2D(_tex1, input.texcoord.xy);
float4 val2 = tex2D(_tex2, val1.yz);
float4 val3 = tex2D(_tex3, val2.zw);
```

Избегайте сложных зависимостей везде, где только можно.

Шейдеры поверхностей

При использовании шейдеров поверхностей, которые упрощают программирование шейдеров, движок Unity берет на себя преобразование кода шейдера поверхностей и избавляет от необходимости задумываться об оптимизациях, которые были рассмотрены выше. Однако данный подход дает возможность изменять некоторые параметры для снижения точности в целях упрощения математических расчетов. Поверхностные шейдеры в общем случае достаточно эффективны, но при индивидуальном подходе можно добиться большей их оптимизации.

Атрибут approxview оказывает влияние на точность аппроксимации направления взгляда, позволяя сэкономить на этой очень затратной

операции. Атрибут `halfasview` регулирует точность представления векторов, но следует остерегаться его влияния на математические операции с данными, имеющими разные типы точности. Атрибут `noforwardadd` уменьшает количество источников направленного света до одного, сокращая количество вызовов системы визуализации, поскольку шейдер будет выполнять отображение за один проход, но уменьшая при этом сложность освещения. И наконец, атрибут `noambient` отключает в учет рассеянного освещения, исключая дополнительные математические операции, которые в данном случае не нужны.

Реализуйте уровни детализации на основе шейдеров

Имеется возможность заставить Unity использовать для отображения удаленных объектов более простые шейдеры, что эффективно сохранит требуемую скорость заполнения, особенно при развертывании игры на различных платформах или поддержке широкого спектра аппаратных возможностей. В шейдере можно использовать ключевое слово `LOD`, чтобы задать экранный размер, поддерживаемый шейдером. Если текущий уровень детализации не соответствует этому значению, будет выполнен переход к следующему шейдеру и т. д., пока не будет найден шейдер, поддерживающий заданный размер. Значение `LOD` для шейдера можно также задать во время выполнения, используя его свойство `maximumLOD`.

Эта функция напоминает рассмотренное ранее отображение мешей с учетом уровня детализации, использующее те же самые значения уровней для определения размеров объекта.

Пропускная способность памяти

Пропускная способность памяти – еще один важный компонент этапа окончательной обработки и потенциальный источник проблем. Пропускная способность памяти расходуется всякий раз, когда текстура извлекается из основной видеопамяти графического процессора (также называемой VRAM). Графический процессор содержит несколько ядер, имеющих доступ к той же области видеопамяти, но каждый из них имеет намного меньший по размеру локальный текстурный кэш, хранящий текущую текстуру или последнюю, с которой работал графический процессор. Это похоже на многоуровневый кэш центрального процессора, позволяющий передавать данные вверх и вниз по цепочке, чтобы компенсировать нехватку быстрой памяти, которая из-за дороговизны имеет меньший объем, чем медленная память.

Всякий раз, когда фрагментный шейдер производит выборку из текстуры, которая уже находится в локальном текстурном кэше ядра, она выполняется молниеносно и практически не заметно. Но если выполняется выборка из текстуры, отсутствующей в текстурном кэше, перед ее выполнением текстура должна быть извлечена из видеопамяти. Сама передача вызывает расходование значительной доли пропускной способности памяти, точнее, эта доля равна полному размеру файла текстуры, хранящемуся в видеопамяти (который может не совпадать ни с размером исходного файла, ни с размером области в оперативной памяти, в зависимости от выбранного уровня сжатия).

Именно по этой причине, если узким местом производительности является пропускная способность памяти, снижение качества текстур при тестировании методом перебора приведет к резкому повышению производительности. Уменьшение размеров текстур увеличивает пропускную способность памяти графического процессора, позволяя ему извлекать необходимые текстуры намного быстрее. Глобально уменьшить качество текстур можно с помощью параметра **Edit** ⇨ **Project Settings** ⇨ **Quality** ⇨ **Texture Quality** (Правка ⇨ Параметры проекта ⇨ Качество ⇨ Качество текстуры), присвоив ему значение **Half Res** (Одна вторая), **Quarter Res** (Одна четвертая) или **Eighth Res** (Одна восьмая).

Когда узким местом является пропускная способность памяти, графический процессор будет извлекать необходимые текстуры снова и снова, что приведет к задержке всего процесса, поскольку кэш текстур будет ждать появления необходимого фрагмента перед его обработкой. Графический процессор не сможет записать данные в буфер кадров к моменту его отображения на экране, заблокирует весь процесс и приведет к уменьшению частоты кадров.

В конечном счете надлежащее использование пропускной способности памяти определяется правильной калькуляцией. Например, если пропускная способность памяти составляет 96 ГБ/с на ядро, а целевая частота кадров – 60 кадров в секунду, графический процессор может вывести до 96/60 = 1,6 ГБ текстурных данных в каждом кадре, перед тем как пропускная способность памяти станет узким местом.

Значение пропускной способности памяти обычно приводится на одно ядро, но некоторые производители графических сопроцессоров, пытаясь ввести покупателей в заблуждение, умножают пропускную способность памяти на число ядер, чтобы она выглядела

больше, чем есть на самом деле. Поэтому, выполняя калькуляцию, убедитесь, что используете в расчетах пропускную способность памяти для одного ядра.

Обратите внимание, что это значение не является максимальным пределом объема текстур, содержащихся в проекте игры, размещенных в оперативной памяти центрального процессора или даже в видеопамяти. Этот показатель ограничивает только объем текстур, передаваемых для формирования одного кадра. В течение одного кадра одна и та же текстура может несколько раз передаваться туда/обратно, в зависимости от количества шейдеров, которым она необходима, порядка отображения объектов и частоты выборки текстуры. Так, отображение всего нескольких объектов может потребовать передачи гигабайтов данных, если все они требуют высокого качества, массивных текстур, нескольких вторичных текстурных карт (нормалей, эмиссий и т. д.) и не участвуют в пакетной обработке просто из-за нехватки места в текстурном кэше для хранения отдельных текстурных файлов достаточно продолжительное время, чтобы его можно было использовать его в следующем проходе отображения.

Существует несколько решений проблемы недостаточной пропускной способности памяти.

Уменьшайте размеры текстур

Этот подход прост, надежен и применим практически всегда. Обычно уменьшение разрешения текстур или использование типов данных с меньшей точностью приводит к ухудшению качества изображения, но иногда переход на использование 16-битных текстур не вызывает заметного ухудшения.

Механизм mip-текстурирования (*глава 4 «Привнесение искусства»*) – еще один отличный способ уменьшить размеры текстур, которые взад/вперед передаются между видеопамятью и текстурным кэшем. Обратите внимание, что представление сцены имеет режим заливки **Mipmaps**, в котором текстуры, присутствующие в сцене, окрашиваются в синий или красный цвет в зависимости от соответствия текущего масштаба текстуры положению и ориентации камеры. Это поможет определить текстуры, являющиеся кандидатами для дальнейшей оптимизации.

Механизм mip-текстурирования практически всегда следует использовать в 3D-сценах с малоподвижной камерой.

Проверяйте различные форматы сжатия текстур

В *главе 4 «Привнесение искусства»* был описан механизм сжатия текстур, помогающий уменьшить общий размер приложения (размер исполняемого файла) и используемый во время выполнения объем памяти центрального процессора, то есть областей хранения данных всех ресурсов, пока они не понадобятся графическому процессору. Однако после того как данные поступят в графический процессор, для их хранения используется другая форма сжатия. На практике часто используются форматы DXT, PVRTC, ETC и ASTC.

Более того, каждая аппаратная платформа поддерживает различные форматы сжатия, и если устройство не поддерживает данного формата сжатия, он будет обрабатываться программно. Другими словами, центральному процессору придется отложить свою работу и преобразовать текстуры в нужный формат, понятный графическому процессору.

Параметры сжатия доступны, только если в поле **Texture Type** (Тип текстуры) ресурса текстуры выбран вариант **Advanced** (Расширенный). При любом другом значении параметра типа текстуры Unity сам выберет формат для целевой платформы. Такой выбор может быть неидеальным для данного оборудования и приводить к увеличению расходования пропускной способности памяти.

Лучший способ определить правильный формат – проверить ряд различных устройств и методов сжатия текстур и выбрать наиболее подходящий. Например, здравый смысл говорит, что формат ETC является лучшим выбором для Android, так как его поддерживает большинство устройств, но некоторые разработчики полагают, что на некоторых устройствах их игра лучше работает при использовании форматов DXT и PVRTC.

Имейте в виду, что выбор метода сжатия отдельных текстур – это крайний случай, когда испробованы все другие способы уменьшения потребления пропускной способности памяти. Двигаясь в этом направлении, можно дойти до индивидуальной поддержки всех устройств. Многие разработчики предпочитают использовать общий подход, а не персональную настройку для решения подобных проблем.

Минимизируйте выборку текстур

Можно ли модифицировать шейдеры, чтобы избавиться от лишней выборки текстур? Можно ли добавить специализированные текстуры

поиска, чтобы получить прирост скорости заполнения в математических функциях? Если это так, подумайте о возможности снижения разрешения текстур, иначе отмените изменения и попробуйте решить проблему скорости заполнения другими способами. По существу, чем меньше выборок текстур производится, тем меньше расходуется пропускная способность памяти и тем ближе решение проблемы.

Организуйте ресурсы для уменьшения перекачки текстур

Этот подход основывается на пакетной обработке и атласах. Существует ли возможность пакетной обработки наиболее объемных файлов текстур? Если это так, можно избавить графический процессор от необходимости перегонять одни и те же файлы текстуры снова и снова в течение одного кадра. Как крайнее средство можно поискать способ удалить некоторые текстуры из проекта и повторно использовать похожие файлы. Например, если имеются проблемы со скоростью заполнения, можно с помощью шейдеров фрагментов собрать в кучу файлы текстур, применяемых в игре с различными вариациями цветов.

Ограничения видеопамяти

Закончим рассмотрение проблем с текстурами оценкой объема доступной видеопамяти. Большинство текстур передается из центрального процессора в графический процессор во время инициализации, но может возникнуть ситуация, когда в текущем представлении первой потребуется несуществующая текстура. Асинхронность процесса приведет к использованию пустой текстуры, пока заполненная текстура не будет готова к отображению. Поэтому избегайте присутствия слишком большого количества вариантов текстур в сцене.

Предварительная загрузка текстуры

Несмотря на то что это не относится напрямую к производительности, стоит отметить, что пустая текстура, используемая во время асинхронной загрузки, может резко снизить качество игры. Поэтому желательно иметь возможность управлять загрузкой текстур с диска и принудительно загружать текстуры в основную память, а затем и в видеопамять, прежде чем они действительно понадобятся.

Для этого часто создается скрытый игровой объект с нужной текстурой и помещается где-то в сцене на пути игрока к месту, где она

потребуется. Как только текстурированный объект становится кандидатом на отображение (даже если он сейчас технически скрыт), начнется процесс копирования его данных в видеопамять. Это несколько громоздкое, но в то же время простое в реализации решение, достаточно эффективное в большинстве случаев.

Можно также управлять этим процессом из кода сценария, изменяя текстуру скрытого материала:

```
GetComponent<Renderer>().material.texture = textureToPreload;
```

Перезагрузка текстур

В редких случаях, когда загружается слишком много текстурных данных и необходимые текстуры отсутствуют в видеопамяти в данный момент, графическому процессору придется извлечь их из основной памяти и затереть прежде загруженные текстуры, чтобы освободить место. Этот процесс усугубляется с течением времени из-за фрагментации памяти, и возникает риск, что затертые текстуры придется извлекать снова и снова на протяжении одного и того же кадра. Это приведет к серьезному увеличению объема перемещаемых данных, чего следует избегать любой ценой.

Эта проблема не стоит так остро в современных консолях PS4, Xbox One и WiiU, так как у них общее пространство памяти для центрального и графического процессоров. Их конструкция оптимизирована на аппаратном уровне и учитывает тот факт, что на устройстве всегда запускается одно приложение и почти всегда отображается 3D-графика. Однако все прочие платформы должны разделять время и память между несколькими приложениями и иметь возможность работать без графического процессора. Поэтому на них имеется отдельная память для центрального и для графического процессоров, и необходимо гарантировать, что общий объем используемых текстур ни в какой момент времени не превысит объема доступной видеопамяти на целевом оборудовании.

Обратите внимание, что такое увеличение перемещений данных в памяти нельзя считать точным эквивалентом увеличения операций обмена данными с жестким диском, когда происходит двусторонний обмен между основной и виртуальной памятью (файлом подкачки), но они аналогичны. В любом случае производится избыточный обмен данными между двумя областями памяти, потому что запрашивается слишком большой объем данных за слишком короткий промежуток времени при слишком малом объеме двух областей памяти.

Увеличение операций перемещения данных в памяти, например, является распространенной причиной снижения производительности при переносе игр с современных консолей на настольные устройства, о чем следует помнить.

Чтобы избежать такого поведения, может потребоваться настройка качества текстур и размеров файлов отдельно для каждой платформы и устройства. Но имейте в виду, что некоторые игроки почти наверняка заметят эти различия. Как известно, даже небольшие различия в аппаратном обеспечении могут привести к сопоставлению яблок с апельсинами, но завзятые игроки ожидают уровня качества, не зависящего от техники.

Освещение и затенение

Освещение и затенение оказывают влияние на все части графического конвейера, поэтому мы рассмотрим их по отдельности. Освещение и затенение считаются наиболее важными элементами художественного оформления и дизайна игры. Хорошее освещение и затенение могут превратить прозаическую сцену в нечто захватывающее, поскольку профессиональная расцветка привносит в нее нечто магическое и делает ее привлекательнее. Даже художественный стиль с малым количеством полигонов (я имею в виду игру Monument Valley), опираясь на хорошее освещение и затенение, позволяет игроку отличить один объект от другого. Однако эта не книга об искусстве, поэтому сосредоточимся на эксплуатационных характеристики различных функций освещения и затенения.

Unity предлагает два стиля динамического освещения, а также внедрение световых эффектов с помощью карт освещения. Кроме того, предоставляется несколько способов создания теней с различными уровнями сложности и затратами на обработку во время выполнения. Эти два стиля не только имеют массу вариаций, но и множество коварных ловушек, в которые легко попасть, если не проявлять должную осторожность.

В документации Unity очень подробно описаны все эти возможности (начните со страницы http://docs.unity3d.com/Manual/Lighting.html и двигайтесь дальше), поэтому рассмотрим их только с точки зрения влияния на производительность.

Начнем с двух главных режимов освещения. Их настройки находятся в разделе **Edit** ⇨ **Project Settings** ⇨ **Player** ⇨ **Other Settings** ⇨

Rendering (Правка ⇨ Параметры проекта ⇨ Игрок ⇨ Другие параметры ⇨ Отображение) и зависят от платформы.

Непосредственное отображение

Непосредственное отображение является классической формой отображения источников света в сцене. Каждый объект обычно отображается за несколько проходов одного и того же шейдера. Число проходов определяется количеством, расположением и яркостью источников света. Unity пытается определить приоритетный источник направленного света для объекта и получить его первичное отображение в «базовом проходе». Затем выделяется до четырех самых мощных ближайших точечных источников света, и тот же объект отображается повторно за нескольких прогонов одного и того же фрагментного шейдера. Следующие четыре точечных источника света обрабатываются на уровне вершин. Все прочие источники света рассматриваются как гигантское пятно с применением технологии под названием **сферические гармоники**.

Некоторые аспекты этого процесса можно упростить, установив режим отображения освещения, например, в значение **Not Important** (Не важно) и изменив значение поля ввода **Edit** ⇨ **Project Settings** ⇨ **Quality** ⇨ **Pixel Light Count** (Правка ⇨ Параметры проекта ⇨ Качество ⇨ Количество пиксельных источников света). Это значение ограничивает количество источников света, учитываемых фрагментным шейдером, но оно переопределяется любыми источниками света с параметром **Render Mode** (Режим отображения), равным **Important** (Важно). Именно поэтому следует проявлять осторожность при использовании данной комбинации параметров.

Как видите, непосредственное отображение может резко увеличить количество обращений к системе визуализации в сценах с большим количеством точечных источников света из-за заданного количества состояний визуализации и проходов шейдеров во время повторной обработки. Приложениям, действующим в системах с ограниченными возможностями центрального процессора, следует избегать этого режима отображения.

Более подробную информацию о непосредственном отображении можно найти в документации Unity: http://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html.

Отложенное затенение

Отложенное затенение, или отложенное отображение, как его иногда называют, доступно только на графических процессорах, поддерживающих, по крайней мере, Shader Model 3.0. Другими словами, на любых видеокартах, выпущенных после 2004 года. Этот метод был весьма популярен некоторое время, но это не привело к полному отказу от метода непосредственной визуализации из-за применения с оговорками и ограниченной поддержки мобильными устройствами. Сглаживание, прозрачность и анимированные персонажи, получающие тени, не поддерживаются при использовании одного только метода отложенного затенения, и его следует использовать вместе с методом непосредственного отображения, в качестве запасного варианта.

Название «отложенное затенение» было дано потому, что тени фактически отображаются в самом конце процесса, то есть их отображение откладывается на более поздний срок. С точки зрения производительности применение этого метода дает весьма впечатляющие результаты, так как позволяет получать очень качественное попиксельное освещение за счет удивительно небольшого количества обращений к системе визуализации. Его преимущество заключается в возможности обработать большое количество источников освещения всего за один проход шейдера освещения. Главными недостатками являются дополнительные затраты при использовании множества продвинутых возможностей освещения, таких как затенение, и любые шаги, требующие непосредственного отображения, например прозрачность.

Документация Unity служит отличным источником информации о технологии отложенного затенения, ее преимуществах и недостатках: http://docs.unity3d.com/Manual/RenderTech-DeferredShading.html.

Обработка освещения в вершинном шейдере (устаревший способ)

С технической точки зрения существует более двух методов реализации освещения. Unity позволяет также использовать и пару устаревших систем, но только одна из них имеет актуальное значение, а именно обработка освещения в вершинном шейдере. Это значительно упрощенный способ расчета освещения, поскольку выполняется только для вершин, а не для пикселей. Другими словами, в цвет падающего освещения окрашиваются целые поверхности, а не отдельные пиксели.

Эта устаревшая технология почти вышла из употребления, так как отсутствие надлежащего освещения и теней затрудняет визуализацию глубины. Она может применяться для 2D-игр, в которых не планируется использовать тени, карты нормалей и различные другие особенности освещения, но эта технология все еще доступна.

Обработка теней в реальном времени

Мягкие тени весьма затратны, жесткие тени менее затратны, но нет теней, свободных от затрат. Все настройки **Shadow Resolution** (Разрешение теней), **Shadow Projection** (Проекция теней), **Shadow Distance** (Расстояние для теней) и **Shadow Cascades** (Каскадность теней), определяющие поведение и сложность теней, можно найти в разделе **Edit** ⇨ **Project Settings** ⇨ **Quality** ⇨ **Shadows** (Правка ⇨ Параметры проекта ⇨ Качество ⇨ Тени). Это почти все, что вам нужно знать о методах создания теней в реальном времени с точки зрения производительности. Более подробно тени будут рассмотрены в следующем разделе, посвященном оптимизации световых эффектов.

Оптимизация освещения

Кратко рассмотрев все технологии освещения, перейдем к описанию методов снижения затрат, связанных с освещением.

Использование соответствующего режима заливки

Проведите тестирование двух основных методов отображения, чтобы определить, какой лучше подходит для вашего проекта. Отложенное затенение часто используется как резервный метод, применяемый, когда непосредственное отображение пагубно влияет на производительность, но вообще многое зависит от результатов выявления узких мест, так как иногда трудно выяснить разницу между ними.

Использование масок выбраковки

Свойство **Culling Mask** компонента источника освещения – это маска слоев, используемая для определения объектов, которые будут затронуты данным источником света. Оно помогает эффективно сократить затраты на освещение, так как взаимодействие со слоями имеет здесь тот же смысл, что при использовании слоев для оптимизации физической системы. Объекты могут входить только в один слой, и сокращение затрат на физическое моделирование и на освещение может стать плюсом, но в случае конфликтов такой подход перестает быть идеальным.

Обратите внимание, что отложенное затенение накладывает ограничения на применение маски выбраковки. Поскольку освещение обрабатывается глобально, в масках можно использовать только четыре слоя, что ограничивает возможности оптимизации этого метода.

Использование внедренных карт освещения

Внедрение освещения и затенения в сцену требует гораздо меньше затрат, чем создание их во время выполнения. Оборотной стороной этого приема является увеличение размера приложения, потребления и падения пропускной способности памяти. В конечном счете, если световые эффекты реализуются только с помощью устаревшего способа расчета освещения в вершинном шейдере или имеется единственный направленный источник освещения, внедрение карты освещения приведет к значительному снижению затрат. Но если полностью полагаться на расчет освещения и теней в реальном времени, это приведет к катастрофе, если игра пытается выиграть награду за наименьший размер файла приложения всех времен и народов.

Оптимизация теней

На обработку теней приходятся больше всего обращений к системе визуализации и наибольшая доля скорости заполнения, но объем передаваемых данных вершин и выбор параметра **Shadow Projection** (Проекция тени) будет оказывать влияние на способность генерировать источники и приемники теней в начальной части конвейера. Первым делом следует попытаться сократить количество вершин для обработки в начальной части конвейера, и эффект от внесения нужных для этого изменений будет многократно усилен.

Обращения к системе визуализации затрачиваются на затенение при отображении видимых объектов в отдельный буфер (так называемую карту теней) – источника или приемника тени либо обоих. Каждый объект, отображаемый в эту карту, требует дополнительных обращений к системе визуализации, что делает тени причиной многократно возрастающих затрат производительности, поэтому пользователю часто предлагаются настройки для понижения качества, позволяющие уменьшить этот эффект или даже исключить его полностью при использовании слабого оборудования.

Параметр **Shadow Distance** (Расстояние теней) является глобальным множителем, влияющим на отображение теней во время выполнения. Чем меньше теней, тем выше производительность

всего процесса отображения. Нет смысла отображать тени на большом расстоянии от камеры, поэтому данный параметр необходимо настроить для конкретной игры, чтобы добиться требуемой видимости теней во время игры. Также имеет смысл дать пользователю доступ к этому параметру, чтобы он мог уменьшить затраты на отображение теней.

Большие значения параметров **Shadow Resolution** и **Shadow Cascades** увеличивают расходование пропускной способности памяти и скорости заполнения. Эти параметры могут помочь исключить появление артефактов при отображении теней, но за счет увеличения размера карты теней, которую нужно пересылать и требующей холст соответствующего размера.

В документации Unity содержится отличное резюме на тему влияния карты теней, где рассказывается, как параметр Shadow Cascades помогает решить эту проблему: http://docs.unity3d.com/Manual/DirLightShadows.html.

Стоит отметить, что мягкие тени потребляют памяти и ресурсов центрального процессора не больше, чем жесткие, поскольку единственное их различие заключается в применении более сложных шейдеров. Это означает, что в приложениях с достаточно высокой скоростью заполнения можно позволить себе улучшенную графическую четкость мягких теней.

Оптимизация графики для мобильных устройств

Способность движка Unity развертываться на мобильных устройствах обусловила рост его популярности среди любителей и команд разработчиков малого и среднего размера. Поэтому уделим внимание нескольким подходам, относящимся в большей степени к мобильным платформам.

Обратите внимание, что любой из подходов, описанных ниже, может устареть в ближайшее время, если какие-то из них уже не устарели. Рынок мобильных устройств развивается невероятно быстро, и приведенные ниже методы просто отражают накопленный за последние пять лет опыт. Иногда следует проверять предложенные подходы, чтобы узнать, действуют ли все еще описанные ограничения для вновь появляющийхся мобильных устройств.

Минимизация обращений к системе визуализации

Чаще всего узким местом мобильных приложений являются обращения к системе визуализации, а не скорость заполнения. Но не следует игнорировать и проблемы, связанные со скоростью заполнения (все когда-нибудь случается!). В любом мобильном приложении приемлемого качества необходимо с самого начала реализовать объединение мешей, пакетную обработку и атласы. Желательно также применить отложенное отображение, поскольку оно хорошо сочетается с другими особенностями мобильных приложений, например отказом от использования прозрачности и большого числа анимированных персонажей.

Минимизация количества материалов

Эта проблема тесно связана с идеями пакетной обработки и атласами. Чем меньше материалов, тем меньше обращений к системе визуализации. Этот подход помогает также решать проблемы низкой пропускной способности памяти, характерные для мобильных устройств.

Уменьшение размеров текстур и количества материалов

Большинство мобильных устройств имеют намного меньший текстурный кэш, чем более мощные настольные устройства. Например, iPhone 3G поддерживает текстуры размером не более 1024×1024, что обусловлено ограничениями библиотеки OpenGLES1.1 и применением простой технологии отображения вершин. Устройства iPhone 3GS, iPhone 4 и iPad используют библиотеку OpenGLES 2.0, поддерживающую текстуры размеров до 2048×2048. Более поздние поколения поддерживают текстуры размером до 4096×4096. Тщательно проверьте характеристики аппаратных устройств, чтобы удостовериться, что они поддерживают размеры текстур, которые предполагается использовать (таких устройств слишком много, чтобы их можно было перечислить здесь). Имейте в виду, что новейшие устройства никогда не являются самым распространенными. Чтобы игра могла завоевать более широкую аудиторию (и увеличить шансы на коммерческий успех), она должна поддерживать и более слабые устройства.

Обратите внимание, что текстуры, слишком большие для графического процессора, упрощаются центральным процессором во время инициализации, на что тратится время при загрузке, и в результате получаются текстуры сомнительного качества. Кроме того, ограни-

ченный объем видеопамяти и кэша текстур в мобильных устройствах придает особую важность приемам многократного использования текстур.

Квадратные текстуры с размером стороны, кратной степени числа 2

Эта тема была рассмотрена в *главе 4 «Привнесение искусства»*, но стоит еще раз обратиться к вопросу сжатия текстур на уровне графического процессора. Графическому процессору будет сложно или даже невозможно сжать текстуру, если она имеет неквадратную форму, поэтому строго придерживайтесь соглашений и используйте квадратные текстуры со стороной, кратной одной из степеней числа 2.

Использование в шейдерах форматов с минимально допустимой точностью

Мобильные графические процессоры особенно чувствительны к точности вычислений в шейдерах, поэтому предпочтение следует отдавать форматам с наименьшей точностью. Важно также отметить, что по этой же причине желательно избегать преобразования форматов.

Избегайте альфа-тестирования

Мобильными графическими процессорами еще не достигнут уровень оптимизации графических процессоров настольных устройств, и альфа-тестирование на мобильных устройствах все еще остается чрезвычайно затратной операцией. В большинстве случаев ее желательно избегать, стараясь заменить альфа-смешиванием.

Итоги

Если вы дошли до этого места, ничего не пропустив, примите поздравления. В этой главе вы получили много информации, касающейся только одного из компонентов движка Unity, который считается самым сложным из всех, что и потребовало столь подробных объяснений. Надеюсь, знакомство с описанными подходами поможет вам в улучшении производительности отображения и вы узнали о конвейере визуализации достаточно, чтобы подойти к его оптимизации со знанием дела!

Теперь уже понятно, что за исключением улучшения алгоритмов любое увеличение производительности привносит свои затраты,

с которыми придется смириться ради устранения узких мест. Также будьте готовы применить несколько методик, пока из них не будет выжато все, на что они способны.

В этой главе была очень подробно рассмотрена достаточно абстрактная система, а теперь перейдем к более конкретной системе, включающей язык C# и движок, лежащий в основе Unity. В следующей описываются код сценария создания сложного освещения и некоторые методы для улучшения производительности центрального процессора и управления памятью.

Глава 7

Мастерство управления памятью

Правильное использование памяти в движке Unity требует хорошего понимания основных составляющих Unity, движка и фреймворка Mono. Это может удивить некоторых разработчиков, так как многие выбирают Unity, прежде всего чтобы избежать тяжелой низкоуровневой работы, связанной с разработкой движка и управлением памятью. Они предпочли бы вместо этого сосредоточиться на проблемах более высокого уровня, связанных с реализацией игрового процесса, дизайна и управлением художественными ресурсами.

При разработке простых игр вполне можно заниматься только проблемами высокого уровня и никогда не сталкиваться с проблемами, связанными с памятью. И все будет хорошо, пока в один прекрасный момент такие проблемы не встанут перед вами. С этого момента отсутствие понимания важных компонентов движка приведет к попытке осуществления принятых наспех решений, трудных для понимания и реализации без надлежащей поддержки.

Поэтому понимание механизма выделения памяти, особенностей взаимодействий языка C# с платформой Mono и платформы Mono с базовым движком и различными библиотеками имеют первостепенное значение для создания высококачественного эффективного кода сценариев. И данная глава будет посвящена описанию всех тонкостей базового движка Unity, платформы Mono, языка C# и фреймворка .NET.

Начнем со знакомой части движка – платформы Mono, отвечающей за обработку сценариев в игре.

Платформа Mono

Mono является волшебным соусом, приготовленным по рецепту Unity и дающим кросс-платформенные воможности. Mono – это проект с открытым исходным кодом, состоящий из фреймворка и библиотек, основанных на прикладном программном интерфейсе, спецификациях и инструментах фреймворка .NET и общих библиотеках от Microsoft. По сути, он повторяет платформу .NET Framework, которая не предоставляет доступа к исходному коду. Даже при том, что библиотеки Mono являются повторной сборкой с открытым исходным кодом базовой библиотеки классов .NET Microsoft, они полностью совместимы с оригинальными библиотеками от Microsoft.

Целью проекта Mono является создание фреймворка с кросс-платформенной совместимостью, основанного на фреймворке .NET в качестве общего слоя. Это позволяет писать приложения на одном и том же языке программирования и запускать их на различных аппаратных платформах, включая Linux, OS X, Windows, ARM, PowerPC и многие другие. Mono также поддерживает множество языков, а не только уже знакомые C#, Boo и UnityScript. Любой язык, который можно скомпилировать в. NET в **общий промежуточный язык** (**Common Intermediate Language**, **CIL** – подробнее об этом позже), пригоден для интеграции с платформой Mono. К ним относятся такие языки, как F#, Java, Visual Basic .NET, PythonNet и IronPython.

Распространенным заблуждением, касающимся движка Unity, является утверждение, что он разработан на платформе Mono. Это не соответствует действительности, поскольку Mono не в состоянии справиться с такими важными игровыми задачами, как воспроизведение звука, отображение графики, моделирование физических процессов и т. д. Для достижения высокой скорости команда Unity Technologies использовала чистый C++, но предусмотрела возможность управления движком через Mono и интерфейс сценариев. То есть, Mono является лишь компонентом базового движка Unity. Подобным образом реализованы многие другие игровые движки, использующие C++ для важных внутренних задач, таких как визуализация, анимация, управление ресурсами и т. д., одновременно обеспечивающие поддержку языка сценариев для реализации игровой логики. Для решения этих задач команда Unity Technologies выбрала платформу Mono.

Под машинным кодом понимается код, скомпилированный для целевой операционной системы и выполняющийся без применения дополнительных слоев прямо в исполняющей среде. Его примене-

ние характеризуется низкими накладными расходами, но за счет необходимости прямого управления памятью и решения других задач в самом коде.

Языки сценариев обычно абстрагируют сложное управление памятью с помощью автоматической сборки мусора и предоставляют различные безопасные способы, упрощающие программирование за счет накладных расходов во время выполнения. Некоторые языки сценариев интерпретируются во время выполнения, то есть они не требуют компиляции кода перед его выполнением. Исходные команды динамически преобразуются в машинный код и выполняются по мере их считывания во время выполнения. Последняя особенность наиболее важна, поскольку позволяет упростить синтаксис программных инструкций. Это обычно чрезвычайно ускоряет процесс разработки, так как члены команды, не имеющие опыта использования языков, подобных C++, вносят свой вклад в разработку. Это позволяет реализовать такие задачи, как логика игрового процесса, в упрощенном формате, что увеличивает скорость выполнения заданий и улучшает их управляемость.

Обратите внимание, что такие языки часто называют «управляемыми» языками, то есть языками, на которых пишется *управляемый код*. Этот термин был введен компанией Microsoft для обозначения любого исходного кода, который должен выполняться в **общеязыковой среде выполнения** (**Common Language Runtime, CLR**), разработанной в Microsoft, в отличие от кода, который компилируется и выполняется напрямую целевой **операционной системой** (**ОС**). Но из-за распространенности и общих черт, присущих среде CLR и другим языкам, имеющим аналогичные среды выполнения (например, язык Java), термин «управляемый» потерял четкие очертания. Он часто используется для обозначения любого языка или кода, который зависит от собственной среды выполнения, и включает или не включает автоматическую сборку мусора. В остальной части этой главы термин «управляемый» будет относиться к коду, зависящему от отдельной среды выполнения и использующему автоматическую сборку мусора.

С каждым годом сниженная производительность управляемых языков становится все менее и менее важной. Это отчасти объясняется постепенной оптимизацией инструментов и сред выполнения, а отчасти увеличением вычислительной мощности типичных устройств. Но сутью полемики об управляемых языках по-прежнему остается принятое в них автоматическое управление памятью. Управление памятью вручную может стать сложной задачей, решение которой требует многолетнего опыта отладки, но многие разработчики считают,

что управляемые языки решают эту проблему слишком непредсказуемым способом, снижая качество продукта. Такие разработчики утверждают, что управляемый код никогда не достигнет уровня производительности машинного кода, и считают безрассудством создание высокопроизводительных приложений на его основе.

Это верно в том смысле, что применение управляемых языков всегда привносит накладные расходы и ведет к частичной потере контроля над операциями с памятью. Но, как всегда, должен соблюдаться определенный баланс, поскольку неполное использование всех ресурсов не всегда приводит к образованию узких мест, и лучшими не обязательно являются игры, в которых учитывается каждый байт. Например, представьте пользовательский интерфейс, который обновляется каждые 30 микросекунд при использовании машинного кода и 60 микросекунд при применении управляемого кода, из-за 100%-го роста затрат (чрезвычайный случай). Версия с управляемым кодом имеет достаточное быстродействие, поэтому пользователь не заметит никакой разницы, так в чем же вред от использования управляемого кода при реализации такой задачи?

В самом деле, применение управляемых языков обычно означает лишь, что разработчикам следует беспокоиться только об уникальных проблемах, в отличие от разработчиков, работающих с низкоуровневыми языками. То есть выбор управляемого языка является отчасти делом вкуса и отчасти компромиссом между предсказуемостью и скоростью разработки.

Процесс компиляции

Изменения, внесенные в код C#, компилируются обычно сразу же после переключения из интегрированной среды разработки (роль которой обычно играет MonoDevelop или Visual Studio) в редактор Unity. Однако код на C# не преобразуется непосредственно в машинный код, как это происходит при использовании статических компиляторов C++. Вместо этого он преобразуется в код на промежуточном языке Common Intermediate Language (CIL), который является абстракцией над машинным кодом. CIL-код похож на байт-код Java, который является для него базовым, но сам по себе CIL-код совершенно бесполезен, поскольку центральный процессор не имеет ни малейшего представления, как обрабатывать инструкции на этом языке.

Во время выполнения промежуточный код обрабатывается виртуальной машиной (VM) Mono, входящей в инфраструктуру, что позволяет исполнять один и тот же код на нескольких платформах

без изменений. Она является реализацией общеязыковой среды выполнения .NET (Common Language Runtime, CLR). При выполнении в iOS используется виртуальная машина с инфраструктурой, основанной на iOS, а при выполнении в Linux используется другая виртуальная машина, лучше подходящая для Linux.

В среде CLR промежуточный CIL-код компилируется в машинный по мере необходимости. Компиляцию выполняет либо компилятор **Ahead-Of-Time (AOT)**, либо **Just-In-Time (JIT)**, в зависимости от целевой платформы. Оба компилятора компилируют сегменты кода в машинный код (то есть в машинный код для целевой ОС), и основное различие между ними заключается только в том, *когда* осуществляется компиляция.

AOT-компиляция выполняется заранее (ahead of time – предварительно): во время сборки или во время инициализации. В любом случае, код будет скомпилирован заранее, что исключает дополнительные затраты, присущие динамической компиляции, во время выполнения. В текущей версии Unity (версия 5.2.2) AOT-компиляцию подерживали только WebGL (и только при использовании UnityScript) и iOS.

JIT-компиляция выполняется динамически во время выполнения в отдельном потоке и начинается непосредственно перед выполнением («Just-In-Time», то есть «как раз вовремя», перед выполнением). Обычно при динамической компиляции первое выполнение фрагмента кода занимает немного (или намного!) больше времени из-за необходимости компиляции до его выполнения. Но с этого момента и далее при выполнении блока не возникает необходимости в повторной его компиляции, поскольку используется уже скомпилированный ранее машинный код.

Очень часто 90% работы программного обеспечения реализуют лишь 10% кода. А это означает, что JIT-компиляция оказывается более производительной, чем непосредственная интерпретация CIL-кода. Однако, поскольку JIT-компилятор должен компилировать код быстро, он не может использовать ряд оптимизаций, которыми могут воспользоваться статические компиляторы.

Ручная JIT-компиляция

В случаях, когда JIT-компиляция является причиной потери производительности во время выполнения, можно выполнить принудительную JIT-компиляцию метода с помощью механизма отражения. Механизм **отражения (Reflection)** является полезной особенностью языка C#, позволяющей програмному коду исследовать собственную

организацию для получения информации о типах, методах, свойствах и метаданных. Использование механима отражения часто является *весьма* дорогостоящим процессом, поэтому его не следует использовать во время выполнения или, по крайней мере, пользоваться им только во время инициализации или загрузки. Несоблюдение этого правила легко может вызвать значительные пиковые нагрузки центрального процессора и зависание игры.

Для принудительной JIT-компиляции метода достаточно с помощью механизма отражения попытаться получить указатель на него:

```
var method = typeof(MyComponent).GetMethod("MethodName");
if (method != null) {
    method.MethodHandle.GetFunctionPointer();
    Debug.Log("JIT compilation complete!");
}
```

Этот код работает только для общедоступных методов. Для других методов требуется использовать `BindingFlags`:

```
using System.Reflection;
// ...
var method = typeof(MyComponent).GetMethod("MethodName",
BindingFlags.NonPublic | BindingFlags.Instance);
```

Такой код должен выполняться только для *очень* конкретных методов, относительно которых есть твердая уверенность, что их JIT-компиляция вызывает скачки нагрузки на центральный процессор. Это можно проверить, повторно запустив приложение и сравнив результаты профилирования первого вызова метода и всех последующих. Разница покажет размер затрат на JIT-компиляцию.

Обратите внимание, что официально для принудительной JIT-компиляции в библиотеке .NET рекомендуется использовать метод `RuntimeHelpers.PrepareMethod()`, но он имеет неудачную реализацию в текущей версии Mono (версия 2.6.5), поставляемой с Unity. Поэтому пока в Unity не будет обновлена версия Mono, следует использовать приведенный выше обходной путь.

Оптимизация использования памяти

В большинстве игровых движков имеется роскошная возможность перевода неэффективного кода сценариев на быстрый язык C++, если они являются причиной проблем с производительностью. Но эта возможность отсутствует в Unity, если не вкладывать серьезные

деньги в покупку исходного кода Unity, предоставляемого в особых случаях по отдельной лицензии, не связанной с системой лицензирования Free/Personal/Pro. Это вынуждает подавляющее большинство разработчиков стараться сделать код сценариев на C# максимально производительным. Итак, чем же приведенная выше предыстория поможет в решении задач оптимизации производительности?

Во-первых, она не охватила всего, что относится к языкам UnityScript и Boo (хотя большая часть сказанного относится и к ним).

Во-вторых, хотя все сценарии пишутся на C#, следует иметь в виду, что движок Unity собран из нескольких компонентов, каждый из которых управляет собственной областью памяти.

В-третьих, только часть выполняемых задач приводит к пробуждению ужасного сборщика мусора. Есть несколько способов выделения памяти, позволяющих избежать этого.

И наконец, все может резко поменяться при смене целевой платформы, и все решения следует проверять на допустимость, поскольку проблемы, связанные с памятью, могут возникнуть в самых неожиданных местах.

Области памяти Unity

Внутреннюю память движка Unity можно разделить на три области. Каждая хранит различные типы данных и предназначена для разных задач.

Первой является **область собственной памяти**. Это основная память движка Unity, написанного на C++ и компилируемого в машинный код для целевой платформы. Она предназначена для выделения памяти для хранения ресурсов, например текстур и мешей, для различных подсистем, таких как система отображения, физического моделирования, ввода и т. д. И наконец, включает представление важных игровых объектов. Именно здесь базовые классы `Component` хранят свои данные, такие как компоненты `Transform` и `Rigidbody`.

Вторая область, **область управляемой памяти**, которую использует платформа Mono, управляется сборщиком мусора. Все объекты и пользовательские классы сценариев хранятся в этой области памяти. Здесь также хранятся обертки для объектов в собственной области. Это как бы мост между Mono-кодом и соответствующим машинным кодом. В каждой области имеются свои представления одной и той же сущности, и пересечение границ между ними может вызывать достаточно значительное падение производительности в игре, как это было описано в предыдущих главах.

При создании нового игрового объекта или компонента происходит выделение памяти в областях управляемой и собственной памяти. Это позволяет подсистемам, таким как система физического моделирования и система отображения, управлять и отображать объекты, основываясь на их данных преобразования в собственной области памяти, а компоненту `Transform` из кода сценария ссылаться на собственную память и изменять данные преобразования объекта. Перемещение взад и вперед через границы между областями следует сократить настолько, насколько это только возможно, из-за сопутствующих затрат, применив, например, кэширование изменений позиции/поворота до их применения, как это было описано в *главе 2* «*Приемы разработки сценариев*».

Третья и последняя область памяти предназначена для встроенных и внешних DLL-библиотек, таких как DirectX, OpenGL и прочих пользовательских, подключаемых к проекту. Ссылки из кода на языке C# на такие библиотеки сопровождаются примерно такой же сменой пространства памяти, что и между Mono-кодом и машинным кодом.

Собственная память

Не имея прямого доступа к исходному коду движка Unity, нельзя реализовать прямое управление памятью собственной области, однако существует несколько способов косвенного управления с помощью различных функций, доступных на уровне сценариев. Внутренний механизм распределения памяти для таких элементов, как игровые и графические объекты, а также профилировщик, скрыт в глубинах машинного кода.

Однако имеется возможность наблюдать за выделением и резервированием памяти в этой области с помощью представления Memory Area (Область памяти) профилировщика. Распределение собственной памяти помечено как «Unity» (рис. 7.1), и, кроме того, можно получить дополнительную информацию с помощью представления **Detailed** (Подробно) при выборе текущего кадра.

```
Simple
Used Total: 140.6 MB   Unity: 34.7 MB   Mono: 14.3 MB   GfxDriver: 8.6 MB   FMOD: 2.1 MB   Profiler: 83.1 MB
Reserved Total: 286.8 MB   Unity: 175.4 MB   Mono: 14.7 MB   GfxDriver: 8.6 MB   FMOD: 2.1 MB   Profiler: 88.0 MB
Total System Memory Usage: 440.0 MB
(WP8) Commited Limit: 0 B   Commited Total: 0 B
```

Рис. 7.1 ❖ Распределение собственной памяти

В разделе Scene Memory (память сцены) можно видеть, что объекты `MonoBehaviour` всегда потребляют постоянный объем памяти,

независимо от содержащихся в них данных. Память выделяется под представление объектов в собственной области памяти. Обратите внимание, что объект `MonoBehaviour` в режиме редактора занимает 376 байт памяти и только 156 байт при профилировании через автономное приложение.

Потребление памяти в режиме редактора всегда резко отличается от затрат в автономном варианте из-за присутствия ловушек отладчика и редактора. Это является дополнительным стимулом избегать использования режима редактора при профилировании и сравнительном анализа (хотя есть ситуации, когда это оказывается полезным).

Узнать объем собственной памяти движка, выделенной под определенный объект, можно также с помощью метода `Profiler.GetRuntimeMemorySize()`.

Управляемые представления объектов неразрывно связаны с их представлениями в области собственной памяти. Лучшим способом свести к минимуму затраты собственной памяти является оптимизация использования соответствующей управляемой памяти.

Управляемая память

Динамическая память в современных операционных системах разделена на две категории: стек и кучу. **Стек** – это специальное зарезервированное пространство в памяти, предназначенное для хранения временных данных, которые автоматически освобождаются в момент выхода из области видимости. Стек хранит локальные переменные, а также используется для обработки вызовов функций и может расти и сжиматься. Освобождение стека практически не вносит затрат, поскольку его данные, по сути, мгновенно забываются и становятся недоступными. Новые данные просто затирают старые, поскольку начало следующего выделенного блока памяти всегда известно и нет никаких причин выполнять операцию очистки.

Поскольку данные в стеке хранятся очень недолго, полный размер стека обычно очень мал и составляет порядка нескольких мегабайт. Выделение большего объема, чем может уместиться на стеке, может привести к его переполнению. Такое возможно при исключительно большом количестве вызовов функций (например, в бесконечных циклах) или при огромном количестве локальных переменных, но чаще переполнение стека не возникает, несмотря на его относительно скромные размеры.

Куча – это все остальное пространство памяти. Она используется в основном для динамического выделения памяти. Всякий раз, когда тип данных значений слишком велик для размещения в стеке или значения данных должны существовать вне функции, где они объявляются, возникает необходимость выделения памяти в куче. **Куча** Mono характерна тем, что управляется сборщиком мусора (такие кучи иногда называют **управляемыми кучами**). Во время инициализации приложения фреймворк Mono запрашивает у операционной системы область памяти и использует ее для создания кучи. В первый момент куча имеет небольшой размер, менее одного мегабайта, но растет по мере потребления памяти сценариями.

Увидеть размер памяти, выделенной для кучи, можно в разделе Memory Area (область памяти) профилировщика, где он подписан словом «Mono» (рис. 7.2).

```
Simple
Used Total  75.6 MB   Unity: 10.3 MB   Mono: 280.0 KB   GfxDriver: 4.8 MB   FMOD: 0.8 MB   Profiler: 60.2 MB
Reserved Total  123.4 MB   Unity: 50.1 MB   Mono: 0.5 MB   GfxDriver: 4.8 MB   FMOD: 0.8 MB   Profiler: 68.0 MB
Total System Memory Usage: 181.0 MB
(WP8) Commited Limit: 0 B   Commited Total: 0 B
```

Рис. 7.2 ❖ Размер памяти, выделенной для кучи

Аналогично текущие размеры используемой и зарезервированной памяти в куче можно определить во время выполнения с помощью методов `Profiler.GetMonoUsedSize()` и `Profiler.GetMonoHeapSize()` соответственно.

Сборка мусора

Когда поступает запрос на выделение памяти и в куче имеется достаточно места , Mono выделяет блок памяти и передает его в распоряжение запросившего. Но если в куче *не* хватает места, пробуждается сборщик мусора, который сначала сканирует все выделенные блоки в поисках неиспользуемых и, найдя, очищает их, а только потом выполняется попытка увеличить текущий размер кучи.

В версии Mono, используемой Unity, применяется так называемый трассирующий сборщик мусора, работающий по принципу «пометить, а затем очистить». Алгоритм делится на два этапа. Сначала каждый объект, для которого выделена память, отмечается дополнительным битом. Этот флаг сообщает, отмечен объект или нет. Изначально флаги имеют значения `0` (или `false`).

В процессе сборки все объекты, все еще доступные программе, маркируются установкой флага в значение `1` или `true`. Доступным

считается объект, на который указывает прямая ссылка, если объект статический или хранится в локальной переменной на стеке, или косвенная – хранящаяся в поле другого объекта, доступного прямо или косвенно. Таким способом выявляются объекты, на которые в настоящий момент можно сослаться.

На втором этапе выполняется перебор ссылок на объекты в куче (которые Mono отслеживает на протяжении всего жизненного цикла приложения) и проверяется состояние флага. Если флаг установлен, объект игнорируется. Если нет, объект становится кандидатом на освобождение. На этом этапе все отмеченные объекты пропускаются, и их флаги сбрасываются в значение `false` в порядке подготовки к первому этапу следующего цикла сборки мусора.

После завершения второго этапа пространство, выделенное под все неотмеченные объекты, освобождается, и вновь оценивается первоначальный запрос на создание объекта. Если места для объекта достаточно, ему выделяется память, и на этом процесс заканчивается. Иначе операционной системе посылается запрос на выделение еще одного блока памяти для кучи. Когда этот вопрос будет решен, процесс наконец-то завершается.

В идеальном случае, когда выделение и освобождение памяти для объектов протекают с равной скоростью, куча имеет постоянный размер, поскольку всегда имеется место для размещения очередного объекта. Однако объекты редко освобождаются в порядке создания, и еще реже им требуется один и тот же объем памяти. Это приводит к фрагментации памяти.

Фрагментация памяти

Фрагментация возникает при выделении и освобождении памяти для объектов в произвольном порядке. Лучше всего это можно пояснить на примере. На рис. 7.3 показаны четыре этапа выделения памяти в типичной куче памяти.

Выделение памяти происходило следующим образом:

1. Работа начинается с пустой кучи (1).
2. Затем в куче выделяется память для четырех объектов – *A*, *B*, *C* и *D* – каждый размером 64 байта (2).
3. Позднее два объекта – *A* и *C* – были освобождены (3).
4. Технически было освобождено 128 байт памяти, но поскольку объекты не были смежными (непосредственно примыкающими друг к другу), на самом деле были освобождены две отдельные области памяти, размером 64 байт каждая. Если через опреде-

ленное время потребуется выделить память для нового объекта размером более 64 байт (4), его не получится разместить в освобожденной памяти, оставшейся от объектов *A* и *C*, так как ни один из освобожденных участков в отдельности не сможет вместить новый объект (объекты всегда должны занимать непрерывный участок памяти). Таким образом, для нового объекта должны быть выделены следующие доступные 128 смежных байтов кучи.

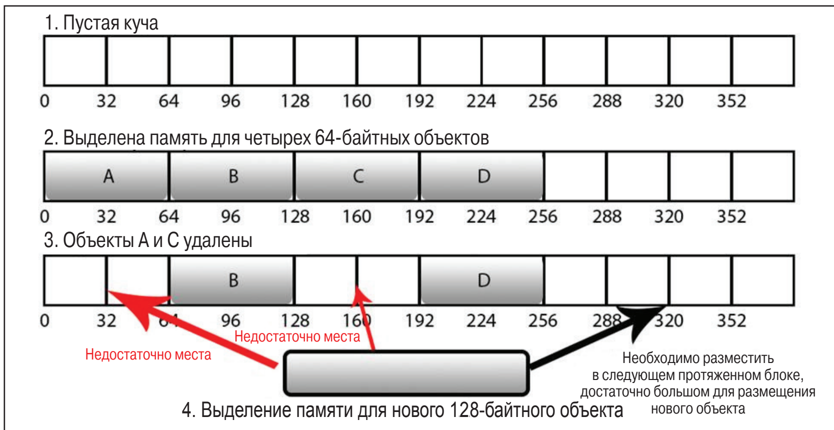


Рис. 7.3 ❖ Четыре этапа выделения памяти в типичной куче

Со временем память в куче будет наводняться все большим количеством маленьких незанятых участков, оставшихся после удаления объектов разных размеров. Далее система будет пытаться выделять память для новых объектов в наименьших доступных участках, подходящих по размерам. Чем меньше становятся эти участки, тем они менее полезны для выделения памяти под новые объекты. В результате пространство памяти становится похожим на швейцарский сыр со множеством маленьких отверстий, которые невозможно использовать. При отсутствии фоновых механизмов, которые позволяют автоматически избавляться от фрагментации, такой эффект будет возникать буквально в памяти любого вида: оперативной, куче и даже на жестком диске, который просто больше, медленнее и предназначен для более длительного хранения (вот почему имеет смысл периодически запускать дефрагментацию жестких дисков!).

Фрагментация памяти вызывает две проблемы. Во-первых, она эффективно снижает общий объем памяти, доступный для новых

объектов, в зависимости от частоты размещения и освобождения. Во-вторых, она увеличивает время выделения новых блоков памяти из-за необходимости поиска достаточно больших участков памяти, пригодных для размещения новых объектов.

При этом важным становится не только *количество* свободного пространства, но и *где* находятся свободные участки. Даже если технически имеются 128 байт свободного пространства для размещения нового объекта, но это пространство не является непрерывным, придется либо продолжить поиск до успешного обнаружения свободного блока достаточно большого размера, либо затребовать выделить новый блок памяти у операционной системы для увеличения размера всей кучи.

Сбор мусора во время выполнения

Итак, в худшем случае, когда игра запрашивает выделение нового блока памяти, центральному процессору придется затратить время на решение следующих задач:

1. Проверить наличие непрерывного свободного пространства для размещения нового объекта.
2. Если такого пространства нет, перебрать все существующие прямые и косвенные ссылки и пометить их как достижимые.
3. Выполнить обход всех блоков памяти в куче и освободить блоки со сброшенным флагом.
4. Еще раз проверить наличие непрерывного свободного пространства для размещения нового объекта.
5. Если такого пространства нет, запросить новый блок памяти у операционной системы для увеличения кучи.
6. Разместить новый объект в начале вновь выделенного блока.

Все это может потребовать слишком много времени центрального процессора, особенно если память выделяется для такого важного игрового объекта, как эффект частиц, новый персонаж, вводимый в сцену, ролик перехода и т. д. Пользователи могут заметить моменты приостановки игрового процесса сборщиком мусора при выполнении этих задач. Положение усугубляется еще больше с увеличением размера кучи, поскольку обход нескольких мегабайт памяти выполняется значительно быстрее, чем сканирование нескольких гигабайт.

Все это делает чрезвычайно важным правильное управление памятью кучи. Плохая тактика использования памяти приводит к практически экспоненциальному росту затрат на сборку мусора. Поэтому, как ни парадоксально, несмотря на все усилия решить эту пробле-

му, разработчики, использующие управляемые языки, сталкиваются с проблемой распределения памяти ничуть не реже, если не *чаще,* чем разработчики, использующие низкоуровневые языки!

Многопоточная сборка мусора

Сборщик мусора работает в двух отдельных потоках: **основном** и **завершающем**. Изначально сборщик запускается в основном потоке и занимается маркировкой освобождаемых блоков памяти. Но освобождение происходит не сразу. Завершающий поток, управляемый фреймворком Mono, вступает в работу с задержкой в несколько секунд и заканчивает процесс освобождения памяти, делая ее доступной для перераспределения.

Это можно пронаблюдать в панели **Total Allocated block** (Всего выделенных блоков) профилировщика (зеленая линия, цвет которой оправдан лишь тем, что только 5% населения имеют дейтеранопию/дейтераномалию). Как результат выделение памяти может занять до нескольких секунд из-за выполнения сборки мусора. Поэтому не следует полагаться на память, доступную для освобождения в текущий момент, и тратить время на проверку возможности использования каждого байта памяти. Всегда необходимо иметь своего рода буферную область для выделения памяти в будущем.

Время от времени сборщик мусора возвращает освобожденные блоки памяти операционной системе, чтобы уменьшить область, зарезервированную под кучу. Это позволяет выделить память на что-то другое, например для использования другим приложением. Однако этот процесс совершенно непредсказуем и зависит от целевой платформы, поэтому не стоит на него полагаться. Единственное, в чем можно быть уверенным, – после выделения памяти для фреймворка Mono она станет недоступна ни для области собственной памяти, ни для любого другого приложения, работающего в той же системе.

Тактика сборки мусора

Одним из способов минимизации затрат на сборку мусора является их сокрытие, то есть принудительный вызов сборщика мусора в благоприятные моменты, когда игрок этого не сможет заметить, обращением к методу:

```
System.GC.Collect();
```

Удобными моментами для принудительного запуска сборки мусора являются, например: загрузка следующего уровня, приостановка

игрового процесса, открытие интерфейсного меню, запуск ролика перехода и практически любой перерыв в игре, когда игрок не заметит внезапной потери производительности. Кроме того, во время выполнения можно использовать методы `Profiler.GetMonoUsedSize()` и `Profiler.GetMonoHeapSize()`, чтобы оценить необходимость вызова сборщика мусора.

Можно также освобождать память, выделенную для конкретных объектов. Если объект является оберткой, например, игрового объекта или компонента `MonoBehaviour`, перед его уничтожением сборщик мусора вызовет метод `Dispose()`, чтобы освободить область собственной памяти. При этом будет освобождена и память в управляемой области. В редких случаях, если обертка реализует интерфейс `IDisposable` (то есть имеет метод `Dispose()`, доступный из сценария), этим процессом можно управлять и освобождать память немедленно.

Единственный пример (известный автору в настоящее время), когда такое освобождение памяти доступно и полезно, – класс `WWW`. Чаще всего этот класс используется для подключения к веб-серверу и загрузки ресурсов во время выполнения. Этот класс выделяет несколько буферов в области собственной памяти Unity. Он также выделяет место для сжатого файла, буфера декомпрессии и итогового распакованного файла. Хранение всего этого в памяти длительное время станет колоссальной потерей драгоценного места в памяти. Поэтому, вызывая метод `Dispose()` из сценария, можно гарантировать быстрое и своевременное освобождение буферов в памяти.

Все прочие объекты ресурсов предлагают свои методы очистки памяти, занимаемой неиспользуемыми данными, например метод `Resources.UnloadUnusedAssets()`. На самом деле ресурсы хранятся в области собственной памяти, поэтому технически сборщик мусора здесь ни при чем, но идея освобождения та же. Она заключается в переборе всех ресурсов конкретного типа для проверки отсутствия ссылок на них и освобождения. Но, опять же, это асинхронный процесс, не гарантирующий немедленного освобождения памяти. Этот метод автоматически вызывается после загрузки сцены, но он не гарантирует мгновенного освобождения.

Метод `Resources.UnloadAsset()` (для освобождения памяти, выделенной под один конкретный ресурс) выглядит предпочтительнее для очистки данных ресурсов, поскольку, по крайней мере, не тратит времени на обход всей коллекции. Однако стоит отметить, что эти методы освобождения памяти в Unity 5 стали многопоточными, что значительно улучшило производительность на большинстве платформ.

Но лучшей стратегией по-прежнему остается исключение сборки мусора, то есть уменьшение выделения памяти под кучу и максимальный контроль над ее использованием, что позволит не беспокоиться о больших затратах, связанных со сборкой мусора. Такая тактика будет описана в остальной части этой главы, но сначала познакомимся с теоретическими вопросами выделения памяти.

Значения и ссылки

Выделение памяти в Mono не обязательно связано с кучей. Во фреймворке .NET (и в расширении языка C#, реализующем спецификацию .NET) поддерживаются понятия значений и ссылок, из которых только последние помечаются сборщиком мусора в алгоритме «пометить, а затем очистить». Ссылочные данные подразумевают (или это требуется) длительное нахождение в памяти из-за их сложности, размера и характера использования. Большие наборы данных и любые объекты, являющиеся экземплярами класса, представляются в виде ссылок. К ним относятся массивы (независимо от того, являются ли их элементы значениями или ссылками), делегаты, все классы, такие как `MonoBehaviour`, `GameObject`, и любые пользовательские классы.

Значения обычно размещаются в стеке. Данные простых типов – логические, целые числа и числа с плавающей запятой – являются примерами значений, но только если они автономны и не содержатся внутри данных ссылочного типа. Если данные простых типов входят в данные ссылочного типа, например класс или массив, предполагается, что они либо слишком велики для стека, либо длительность их существования должна превышать время существования текущей области видимости и их следует хранить в куче.

Все это лучше пояснить на примерах. Следующий фрагмент кода создает значение целого типа, которое временно хранится на стеке:

```
public class TestComponent : MonoBehaviour {
    void Start() {
        int data = 5; // помещается на стек
        DoSomething(data);
    } // здесь целое значение удаляется со стека
}
```

По завершении метода `Start()` целое значение удаляется со стека. Эта операция, по сути, не вносит никаких затрат, как уже упоминалось выше, она не требует последующей сборки мусора, поскольку указатель стека просто возвращается к предыдущему положению в стеке вызовов. Любые дальнейшие данные, размещаемые на стеке,

просто затрут устаревшие данные. И, что самое главное, не производится выделение памяти в куче, и, следовательно, сборщику мусора об этом значении ничего не известно.

Но если целочисленная переменная находится внутри определения класса MonoBehaviour, она будет принадлежать ссылочному типу (классу) и попадет в кучу вместе со своим контейнером:

```
public class TestComponent : MonoBehaviour {

    private int _data = 5;

    void Start() {
        DoSomething(_data);
    }
}
```

Аналогично, если поместить целое число в собственный класс, к нему будут применяться правила для ссылок, и объект опять же окажется в куче:

```
public class TestData {
    public int data = 5;
}

public class TestComponent : MonoBehaviour {
    void Start() {
        TestData dataObj = new TestData(); // помещается в кучу
        DoSomething(dataObj.data);
    } // 'dataObj' не будет освобожден прямо здесь, но станет
      // кандидатом на уничтожение при следующей сборке мусора
}
```

Итак, имеется значительная разница между временным выделением памяти в методе класса и хранением долгосрочных данных в свойствах класса. В обоих случаях для хранения данных используется ссылочный тип (класс), что означает возможность внешней ссылки на него. Например, представьте, что метод DoSomething() сохраняет ссылку на dataObj в свойстве:

```
private TestData _testDataObj;

void DoSomething (TestData dataObj) {
    _testDataObj = dataObj; // создана новая ссылка! Ссылка на объект
                            // будет отмечаться в ходе работы алгоритма
                            // сборки мусора как достижимая
}
```

В этом случае нельзя освободить объект, указанный в dataObj, с завершением метода Start(), поскольку общее количество ссылок на объект при этом лишь уменьшится с 2 до 1. Это не 0, и потому сборщик мусора будет по-прежнему отмечать объект как достижимый. Мы должны присвоить полю _testDataObj значение null или сделать со ссылкой что-то еще, чтобы объект стал недоступным.

Обратите внимание, что переменная с типом значения не может иметь значения null. Если значению в стеке присвоить ссылочный тип, это приведет к простому копированию данных. То же относится и к массивам значений:

```
public class TestClass {
    private int[] _intArray = new int[1000]; // Ссылочный тип, заполненный
                                             // значениями
    void StoreANumber(int num) {
        _intArray[0] = num; // сохранение значения в массиве
    }
}
```

При создании исходного массива (во время инициализации объекта) в куче будет выделено место для 1000 целых чисел, и всем им будет присвоено значение 0. Метод StoreANumber() просто скопирует значение переменной num в нулевой элемент массива, а не присвоит ссылку на нее.

Следует учитывать все тонкости преобразования значений в ссылки и стараться использовать автономные значения всякий раз, когда имеется такая возможность, поскольку они хранятся в стеке, а не в куче. В любой ситуации, когда фрагмент данных должен существовать только в пределах текущей области видимости, следует использовать значения, а не ссылочные типы. Это не вызывает проблем при передаче данных в другой метод того же класса или в метод другого класса, поскольку они по-прежнему остаются значениями, которые существуют на стеке, пока не будет завершено выполнение создавшего их метода.

Передача по значению и по ссылке

Технически при передаче значения в виде аргумента от одного метода к другому происходит его копирование. Это верно как для обычных значений, так и для ссылок. Это называют **передачей по значению**. Главным отличием при передаче значения ссылочного типа является передача указателя, занимающего всегда 4 или 8 байт памяти (для 32-разрядной или 64-разрядной архитектуры), независимо от того, на что эта ссылка указывает. Если ссылочный тип передается

в виде аргумента, на самом деле передается значение указателя, которое дублируется очень быстро, поскольку длина его невелика.

В то же время обычные значения содержат все биты данных, хранящиеся в объекте. Следовательно, все данные обычных значений копируются при каждой передаче между методами или при сохранении в других значениях. В некоторых случаях это приводит к передаче обычных значений большого размера в аргументах, что может стать даже более затратным, чем передача значения ссылочного типа. Для большинства типов значений это не проблема, поскольку они близки по своим размерам к указателям. Но это важно для структур, о которых речь пойдет в следующем разделе.

Данные могут также передаваться по ссылке с помощью ключевого слова `ref`, но этот способ сильно отличается от идеи значений и ссылок, и очень важно знать его особенности, чтобы разобраться в том, что при этом происходит. Обычное значение можно передать по значению или по ссылке, и значение ссылочного типа также передать по значению или по ссылке. Это означает, что существуют четыре варианта передачи данных, причем конкретный вариант определяется типом передаваемого значения и использованием при передаче ключевого слова **ref**.

Когда данные передаются по ссылке (даже если это просто значение!), последующие попытки их изменения приведут к изменению оригинала. Например, следующий код выведет значение `10`:

```
void Start() {
    int myInt = 5;
    DoSomething(ref myInt);
    Debug.Log(String.Format("Value = {0}", myInt));
}

void DoSomething(ref int val) {
    val = 10;
}
```

Если удалить ключевое слово `ref` в двух местах, этот фрагмент выведет число `5`. Понимание этой особенности поможет при работе с более сложными типами данных. А именно со структурами, массивами и строками.

Структуры являются значениями

Структуры – особый случай в языке C#. Перешедшие с языка C++ на C#, вероятно, полагают, что единственное отличие структуры от

класса заключается в том, что структура имеет спецификатор доступа по умолчанию `public`, в то время как в классах по умолчанию используется спецификатор `private`. В действительности структуры в C# подобны классам: они могут иметь поля со спецификаторами `private`/`protected`/`public`, включать методы, использоваться для создания экземпляров во время выполнения и т. д. Основное различие между ними состоит в том, что структуры являются значениями, а классы – ссылками.

Существует также несколько других важных отличий между структурами и классами в C#: структуры не поддерживают наследования, их свойствам нельзя присвоить значения по умолчанию (они всегда имеют значения по умолчанию `0` или `null`, в зависимости от типа свойства), и их конструкторы по умолчанию не могут переопределяться. Это значительно ограничивает их использование, по сравнению с классами, поэтому простую замену всех классов на структуры (чтобы память для них выделялась только на стеке) нельзя назвать разумным решением.

Однако если класс используется только для передачи большого двоичного объекта куда-то в другое место в приложении и он не нужен нигде за пределами текущей области видимости, вместо него следует использовать структуру, поскольку применение класса приведет к размещению данных в куче, что не желательно:

```
public class DamageResult {
    public Character attacker;
    public Character defender;
    public int totalDamageDealt;
    public DamageType damageType;
    public int damageBlocked;
    // и т. д.
}

public void DealDamage(Character _target) {
    DamageResult result =
    CombatSystem.Instance.CalculateDamage(this, _target);
    CreateFloatingDamageText(result);
}
```

В этом примере класс переносит данные из одной подсистемы (системы боя) в другую (систему пользовательского интерфейса). Единственная цель этих данных заключается в расчете и считывании в разных подсистемах, что делает их хорошим кандидатом для преобразования в структуру.

Простая замена ключевого слова class на struct в определении DamageResult избавит от ненужных затрат на сборку мусора, поскольку память будет выделена на стеке.

```
public struct DamageResult {
    // ...
}
```

Это решение не является самым оптимальным. Поскольку структуры являются значениями, это делает достаточно уникальной их передачу в аргументах между методами. Как уже упоминалось выше, при передаче значения в виде аргумента от одной функции к другой, оно дублируется, то есть происходит передача по значению. При этом создается дубликат значения для принимающего метода, который будет уничтожен по его завершении, и так при каждой передаче. То есть если структура передается по значению в длинной цепи из пяти разных методов, в стеке будут одновременно существовать пять копий. Как уже говорилось, освобождение памяти в стеке не сопряжено с накладными расходами, но к копированию данных это не относится.

Копирование небольших по размеру значений, таких как отдельное целое или вещественное число, сопровождает незначительные затраты, но многократная передача достаточно большого набора данных через структуру, очевидно, не является столь же тривиальной задачей, и ее следует избегать. В таких случаях разумнее передавать структуру по ссылке с помощью ключевого слова ref, чтобы свести к минимуму объем копируемых данных (лишь 32- или 64-разрядное целое число для ссылок). Однако это может быть опасным, поскольку передача по ссылке позволит любому из методов в последовательности изменить структуру. Поэтому разумнее сделать значение доступным только для чтения, с помощью ключевого слова readonly (такие значения инициализируются в конструкторе, и их нельзя изменить нигде больше, даже в функциях-членах), для предотвращения последующих изменений.

Все вышесказанное справедливо и для структур внутри данных ссылочных типов:

```
public struct DataStruct {
    public int val;
}
public class StructHolder {
    public DataStruct _memberStruct;
    public void StoreStruct(DataStruct ds) {
```

```
        _memberStruct = ds;
    }
}
```

При недостатке опыта может показаться, что предыдущий код преобразует хранящуюся на стеке структуру в ссылку. То есть объект `StructHolder` в куче ссылается на объект в стеке? А что случится, когда метод `StoreStruct()` завершится и структура будет уничтожена? Это неверная постановка вопроса.

На самом деле хотя память для объекта `DataStruct` (`_memberStruct`) была выделена в куче, внутри объекта `StructHolder`, он по-прежнему является значением и не превращается волшебным образом в ссылку. То есть к нему применяются все обычные правила для значений. Переменная `_memberStruct` не может иметь значения `null`, и все ее поля будут инициализированы значениями `0` или `null`. При вызове функции `StoreStruct()` данные из `ds` копируются в `_memberStruct`. Нет никаких ссылок на объект в стеке, и не стоит волноваться по поводу потери данных.

Массивы являются ссылками

Массивы способны хранить огромные объемы данных, что усложняет их обработку как значений, поскольку есть вероятность, что для них не хватит места на стеке. Поэтому они обрабатываются как ссылки, что позволяет передавать весь набор данных передачей одной-единственной ссылки, без копирования всего массива, независимо от того, содержит ли массив значения или ссылки.

Это означает, что следующий код разместит массив в куче:

```
TestStruct[] dataObj = new TestStruct[1000];
for(int i = 0; i < 1000; ++i) {
    dataObj[i].data = i;
    DoSomething(dataObj[i]);
}
```

Следующий код функционально эквивалентен предыдущему, но он не использует кучу, потому что структуры являются значениями и хранятся на стеке:

```
for(int i = 0; i < 1000; ++i) {
    TestStruct dataObj = new TestStruct();
    dataObj.data = i;
    DoSomething(dataObj);
}
```

Тонкое отличие второго фрагмента заключается в том, что в каждый момент на стеке присутствует только один экземпляр структуры `TestStruct` (по крайней мере, в этой функции, так как `DoSomething()` может создать дополнительные экземпляры), а в первом фрагменте выделяется память для массива из 1000 элементов. Я понимаю, что эти методы выглядят дико, но они написаны только для иллюстрации рассматриваемых понятий. Компилятор недостаточно умен, чтобы автоматически сориентироваться в такой ситуации и внести соответствующие изменения. Возможности оптимизации использования памяти с помощью перехода на значения полностью зависят от способности их обнаружить и понять, какие изменения приведут к выделению памяти в стеке, а не в куче.

Обратите внимание, что при размещении массива ссылок создается массив, каждый элемент которого ссылается на определенное место в куче. А при размещении массива значений создается упакованный список значений в куче. Каждое из этих значений будет инициализировано значением `0` (или эквивалентным ему), поскольку им нельзя присвоить `null`, а каждая ссылка в массиве будет инициализирована значением `null`.

Строки являются неизменяемыми ссылками

Тема строк была кратко затронута в *главе 2* «*Приемы разработки сценариев*», теперь остановимся на ней более подробно, поскольку правильное использование строк чрезвычайно важно.

По своей сути строки являются массивами символов, поэтому они относятся к ссылочным типам и к ним применимы все правила для ссылок, то есть при их копировании и передаче между функциями используется указатель, и они размещаются в куче.

Путаница начинается, когда обнаруживается, что строки являются **неизменяемыми**, то есть их нельзя изменить после размещения в памяти. Будучи массивом, строка должна храниться в непрерывном блоке памяти, что делает невозможным расширение и сжатие строк в динамической памяти (как можно быстро и безопасно расширить строку, если память, следующая непосредственно за ней, уже занята?).

Это означает, что при изменении строки необходимо разместить в памяти новую строку, содержащую измененную копию оригинала в виде нового массива символов. В этом случае в программе не останется ссылок на старую версию строки, она не будет отмечена сборщиком мусора на этапе маркировки и в конечном итоге будет унич-

тожена. Как результат небрежное программирование может привести к выделению массы ненужной памяти в куче с последующей ее обработкой при сборке мусора.

Например, если предположить, что строки обрабатываются так же, как другие ссылочные типы, для вывода строки `World!`, мы могли бы написать следующий код:

```
void Start() {
    string testString = "Hello";
    DoSomething(testString);
    Debug.Log(testString);
}

void DoSomething(string localString) {
    localString = "World!";
}
```

Однако это не так, и данный код выведет слово `Hello`. В действительности переменная `localString` внутри функции `DoSomething()` ссылается на то же место в памяти, что и переменная `testString`, поскольку значение передается по ссылке. В результате появляются две ссылки, указывающие на одно и то же место в памяти, чего и следовало ожидать при применении любого ссылочного типа. Пока все в порядке.

Но при изменении значения переменной `localString` возникает конфликт. Строки являются неизменяемыми, поэтому, чтобы создать иллюзию изменения строки, нужно выделить память под новую строку со словом `World!` и присвоить полученную ссылку переменной `localString`, после чего останется единственная ссылка на строку `Hello`. Исходная строка (`Hello`) останется в памяти без изменений, поскольку значение переменной `testString` не будет изменено, и это сохраненное значение будет выведено в `Debug.Log()`. Функция `DoSomething()` всего лишь создаст новую строку в куче, которая будет уничтожена сборщиком мусора и ничего не изменит. Это учебное пособие по расточительности.

Если добавить ключевое слово `ref` в определение метода `DoSomething()`, чтобы он получал строку по ссылке, код действительно выведет слово `World!`. Именно такого результата мы ожидали от использования значения, и подобные ожидания заставляют многих разработчиков неправильно полагать, что строки являются значениями. Но это уже пример четвертого и последнего варианта передачи данных, когда ссылка передается по ссылке, что позволяет изменить то, на что *ссылается* исходная ссылка!

Итак, подведем итоги. Если аргумент передается по значению, для изменения будет доступна только копия значения. Если аргумент передается по ссылке, для изменения будет доступно фактическое значение оригинала. Если в аргументе передается ссылка по значению, для изменения будет доступен объект, на который ссылается исходная ссылка. И наконец, если в аргументе передается ссылка по ссылке, для изменения будет доступна исходная ссылка.

Объединение строк

Под объединением (конкатенацией) понимается операция добавления одной строки в конец другой для формирования более длинной строки. Как уже упоминалось, это приводит к избыточным затратам памяти в куче. Дополнительные затраты возникают при объединении строк оператором + или += из-за эффекта цепного распределения памяти.

Например, следующий код объединяет несколько строк для вывода информации о результате боя:

```
void CreateFloatingDamageText(DamageResult result) {}
    string outputText = result.attacker.GetCharacterName() + "
    dealt " + result.totalDamageDealt.ToString() + " " +
    result.damageType.ToString() + " damage to " +
    result.defender.GetCharacterName() + " (" +
    result.damageBlocked.ToString() + " blocked)";
    // ...
}
```

Эта функция, например, может вывести такую строку:

```
Dwarf dealt 15 Slashing damage to Orc (3 blocked)
```

Эта функция содержит несколько строковых литералов (которые размещаются в памяти в момент инициализации приложения), таких как `dealt`, `damage to` и `blocked`. Но из-за участия переменных в операции объединения итоговая строка не может быть собрана во время компиляции и должна создаваться динамически во время выполнения.

Каждый оператор + и += будет вызывать распределение нового блока памяти в куче, и за один раз будет объединяться только одна пара строк, что приведет к неоднократному выделению памяти. Результат слияния первой пары послужит источником для объединения с еще одной строкой и т. д., пока не будет собрана вся строка целиком.

То есть предыдущий пример создаст 9 различных строк за один проход. При выполнении этой инструкции для каждой строки будт

выделена память, которая в конечном итоге будет утилизирована сборщиком мусора (обратите внимание, что операторы выполняются справа налево):

```
"3 blocked)"
" (3 blocked)"
"Orc (3 blocked)"
" damage to Orc (3 blocked)"
"Slashing damage to Orc (3 blocked)"
" Slashing damage to Orc (3 blocked)"
"15 Slashing damage to Orc (3 blocked)"
" dealt 15 Slashing damage to Orc (3 blocked)"
"Dwarf dealt 15 Slashing damage to Orc (3 blocked)"
```

Здесь будет выделено памяти для 262 символов вместо 49, или, учитывая, что каждый символ занимает 2 байта, 524 байта памяти вместо 98. Этот код может вызываться в приложении много раз, что приведет к многократным повторениям весьма затратного объединения и ненужным расходам тонн памяти на создание лишних строк.

Обратите внимание, что длинные строковые литералы можно безопасно объединять с помощью операторов `+` и `+=` для придания лучшей читаемости коду, но только если в результате будет получена строковая константа. В этом случае компилятор выполнит слияние строковых литералов в одно целое во время компиляции.

Для конструирования строк программным способом рекомендуется использовать класс `StringBuilder` или методы класса `string`.

Класс `StringBuilder` представляет изменяемую строку (доступную для изменения). Он выделяет буферы для целевых строк и резервирует дополнительное пространство по мере необходимости. Извлечь сконструированную строку можно вызовом метода `ToString()`, что, естественно, приведет к выделению памяти для строки, но, по крайней мере, удастся избежать выделения памяти под промежуточные строки при использовании операторов + и +=.

Обычно окончательный размер строки результата примерно известен, что позволяет заблаговременно выделить соответствующий буфер и застраховаться от чрезмерного выделения памяти. В примере выше можно было бы выделить буфер емкостью в 100 символов, чтобы в него поместилось самое длинное имя персонажа и описание нанесенных повреждений:

```
using System.Text;
// ...
StringBuilder sb = new StringBuilder(100);
sb.Append(result.attacker.GetCharacterName());
sb.Append(" dealt " );
sb.Append(result.totalDamageDealt.ToString());
// и т. д. ...
string result = sb.ToString();
```

Если окончательный размер неизвестен, есть риск при использовании класса `StringBuilder` создать слишком большой или слишком маленький буфер. Слишком большой буфер приведет к напрасной трате времени и памяти, а слишком маленький, что еще хуже, – к его переполнению при создании строки результата. Если полный размер строки заранее не известен, лучше использовать один из методов класса `string`.

Класс `string` обладает тремя методами для создания строк: `Format()`, `Join()` и `Concat()`. Каждый из них работает по-своему, но результат всегда один и тот же, а именно: в памяти размещается новая строка, содержащая строки, переданные в метод, и все это делается в одно действие без выделения памяти под промежуточные строки.

Порой очень сложно сделать выбор между этими двумя подходами, поскольку каждый имеет массу неочевидных нюансов. На эту тему постоянно ведутся дискуссии (просто поищите в Google по строке «C# производительность конкатенации строк», и вы сами увидите, что я имею в виду), поэтому выбор наилучшего подхода должен основываться на приведенных выше соображениях. В случае потери производительности при использовании одного из подходов попробуйте применить другой, проведите их профилирование и выберите наилучший вариант.

Упаковка

Формально все в языке C# является объектами (с некоторыми оговорками). Даже простые типы данных, например `int`, `float` и `bool`, являются низкоуровневыми производными от `System.Object` (ссылочный тип!), что позволяет им иметь доступ к вспомогательным методам, например к методу `ToString()` для получения их строкового представления.

Но эти простые типы рассматриваются как особые случаи и интерпретируются как значения. Всякий раз, когда значение неявно обрабатывается как объект, среда CLR автоматически создает временный

объект для хранения или «упаковки» значений, чтобы обрабатывать их как типичные объекты ссылочного типа. Как вы уже догадались, это приводит к выделению памяти в куче.

Обратите внимание, что упаковка не похожа на использование значений в качестве свойств ссылочных типов. Она предназначена для интерпретации значений как объектов.

Например, следующий код приведет к упаковке переменной i в объект obj:

```
int i = 128;
object obj = i;
```

Следующий код использует представление в виде объекта obj для изменения значения целого числа и его «распаковки» обратно в целое число с сохранением в переменной i. В результате переменная i получит значение 256:

```
obj = 256;
i = (int)obj;
```

Такие типы могут динамически изменяться. Ниже приводится вполне допустимый код на C#, использующий все тот же объект obj, который изначально был упаковкой типа int:

```
obj = 512f;
float f = (float)obj;
```

Следующий код также допустим:

```
obj = false;
bool b = (bool)obj;
```

Обратите внимание, что попытка распаковать obj в тип, не совпадающий с типом последнего присвоенного значения, приведет к возбуждению исключения InvalidCastException. Все это сложно воспринимать, пока не придет осознание, что все это – в конце концов, просто несколько битов в памяти. Важно лишь знать, что есть возможность обрабатывать значения простых типов как объекты, упаковывать их, преобразовывать их типы и затем распаковывать их в другой тип.

Обратите внимание, что для преобразования типа упакованного объекта можно использовать один из многочисленных методов System.Convert.To...().

Упаковка может быть явной, как в примерах выше, и неявной, путем приведения к типу `System.Object`. Распаковка всегда должна быть явной, с приведением к исходному типу. Всякий раз при передаче значения в метод, который ожидает получить аргумент типа `System.Object`, производится неявная упаковка.

Типичным примером методов, принимающих аргументы типа `System.Object`, является метод `String.Format()`. Обычно в этих аргументах передаются значения простых типов, таких как `int`, `float`, `bool` и др. Автоматически выполняемая при этом упаковка сопровождается выделением памяти в куче, о чем следует помнить. Еще один пример – тип `Collections.Generic.ArrayList`, экземпляры которого всегд содержат ссылки типа `System.Object`.

При передаче значений любой функции, принимающей аргументы типа `System.Object`, следует помнить, что при этом происходит неявное распределение памяти в куче из-за упаковки.

Важность порядка размещения данных

На удивление легко забыть о порядке размещения данных в памяти, а между тем правильная их организация может дать значительный прирост производительности. Избегайте промахов кэша, насколько это возможно. Эта рекомендация означает, что массивы данных, располагающихся в памяти непрерывно, должны обходиться последовательно и ни в каком другом порядке.

Порядок размещения данных также имеет большое значение для сборки мусора, потому что сборка выполняется последовательно, и способы, позволяющие сборщику мусора пропускать проблемные области, помогают сэкономить массу времени при обходе.

По сути, имеет смысл хранить большие группы ссылок отдельно от больших групп значений. При наличии даже одной-единственной ссылки в значении, например в структуре, сборщик мусора считает весь объект и все его элементы объектами, на которые может иметься ссылка. Когда приходит время для маркировки с последующей очисткой, это приводит к проверке всех полей объекта. Но, разделив массив на несколько массивов с элементами разных типов, можно добиться пропуска сборщиком мусора большей части данных.

Например, если имеется массив структур, как показано ниже, сборщику мусора понадобится довольно много времени, чтобы перебрать все свойства всех структур:

```
public struct MyStruct {
    int myInt;
    float myFloat;
    bool myBool;
    string myString;
}
MyStruct[] arrayOfStructs = new MyStruct[1000];
```

Но если разместить эти же данные в простых массивах, сборщик мусора сможет проигнорировать все простые типы данных и проверить только строки. Это приведет к значительному ускорению процесса сборки мусора:

```
int[] myInts = new int[1000];
float[] myFloats = new float[1000];
bool[] myBools = new bool[1000];
string[] myStrings = new string[1000];
```

Причина заключается в уменьшении количества косвенных ссылок, проверяемых сборщиком мусора. При разделении данных на отдельные массивы (ссылки) сборщик мусора выделяет три массива значений, помечает их и сразу же двигается дальше, потому что ему нет никакого смысла помечать значения. Но он должен обойти все строки в массиве строк, так как каждая является ссылочным типом, и ее необходимо проверить на наличие косвенных ссылок внутри. Разумеется, строки не могут содержать косвенных ссылок, но сборщик мусора работает на уровне, где объекты делятся только на ссылки и значения. Однако мы все еще имеем неплохой выигрыш, так как сборщик мусора может пропустить 3000 элементов данных (по 1000 значений типа `int`, `float` и `bool`).

Прикладной программный интерфейс Unity

В прикладном программном интерфейсе Unity имеется несколько команд, которые приводят к выделению памяти в куче, о чем следует знать. В их число в основном входят команды, возвращающие массивы данных. Например, вызов следующих методов приведет к выделению памяти в куче:

```
GetComponents<T>(); // (T[])
Mesh.vertices; // (Vector3[])
Camera.allCameras; // (Camera[])
```

Использования таких методов следует избегать по мере возможности или, по крайней мере, вызывать их однократно и помещать результат в кэш, чтобы не вызывать излишнего выделения памяти.

Разработчики из Unity Technologies сообщили о возможности создания версий этих методов, не выделяющих память в куче, уже для Unity 5. Предположительно при этом может быть применен тот же подход, что и для создания систем частиц, позволяющий получить доступ к данным с помощью массива ссылок `Particle[]`, содержащего указатели на данные. Это позволяет избежать распределения памяти в куче за счет повторного использования одного и того же буфера.

Циклы foreach

Вокруг использования ключевого слова `foreach` для организации циклов ведутся споры в кругах Unity-разработчиков. Это вызвано тем, что обычно применение циклов `foreach` в коде на C# приводит к излишнему выделению памяти в куче из-за использования объекта `Enumerator`, размещаемого в куче, а не структуры на стеке. Но это зависит от особенностей реализации метода `GetEnumerator()` коллекции.

Как оказывается, все коллекции, реализованные в версии Mono, поставляемой с Unity (версия 2.6.5 Mono), создают классы вместо структур, что приводит к выделению памяти в куче. К их числу относятся коллекции `List<T>`, `LinkedList<T>`, `Dictionary<K,V>`, `ArrayList` и др. Но обратите внимание, что к массивам это не относится! При работе с массивами, компилятор Mono просто преобразует циклы `foreach` в `for`.

Затраты оказываются незначительными, так как размер памяти, выделенной в куче, не увеличивается пропорционально количеству итераций. Выделяется память только под один объект `Enumerator`, который используется многократно, что составляет всего несколько байтов памяти. То есть даже если цикл `foreach` запускается при каждом обновлении (что обычно опасно само по себе), затраты в небольших проектах будут очень незначительными. Время, затраченное на преобразование всех таких циклов в циклы `for`, может не стоить полученных при этом выгод. Но об этой особенности следует помнить, начиная работу над новым проектом.

Знатоки C#, Visual Studio и ручной компиляции сборок Mono могут исправить эту ошибку для универсальных коллекций, выполнить в Visual Studio компиляцию кода и скопировать полученную сборку DLL в папку `Assets`.

Обратите внимание на типичный перебор в цикле `foreach` дочерних элементов компонента `Transform`. Например:

```
foreach (Transform child in transform) {
    // делаем что-то с 'child'
}
```

Однако это приводит к такому же выделению памяти в куче, как это было описано выше. Поэтому желательно использовать следующий стиль кодирования:

```
for (int i = 0; i < transform.childCount; ++i) {
    Transform child = transform.GetChild(i);
    // делаем что-то с 'child'
}
```

Сопрограммы

Изначально сопрограмма потребляет небольшое количество памяти, но обратите внимание, что затраты увеличиваются при вызове методов *yield*. Если возникают значительные проблемы, связанные с потреблением памяти и сборкой мусора, нужно избегать сопрограмм с очень коротким циклом существования и частых вызовов `StartCoroutine()` во время выполнения.

Замыкания

Замыкания являются полезным, но и опасным инструментом. Анонимные методы и лямбда-выражения не всегда являются замыканиями, но могут быть ими. Все зависит от того, использует ли метод данные из-за пределов своей области видимости и своего списка параметров.

Например, следующая анонимная функция не является замыканием, поскольку она автономна и функционально эквивалентна любой другой локально определенной функции:

```
System.Func<int,int> anon = (x) => { return x; };
int result = anon(5);
```

Но если анонимная функция использует внешние данные, она превращается в замыкание, так как «замыкает в себе среду», окружающую требуемые данные. Следующий код образует замыкание:

```
int i = 1024;
System.Func<int,int> anon = (x) => { return x + i; };
int result = anon(5);
```

Для завершения этой транзакции компилятор должен определить новый класс, ссылающийся на окружение, где доступно значение

переменной `i`. Во время выполнения будет создан соответствующий объект в куче и передан анонимной функции. Обратите внимание, что объект включает значение (как в примере выше), обычно размещаемое в стеке, поэтому следует постараться поместить его в стек. В следствие этого каждый вызов второго метода приведет к выделению памяти в куче и неизбежной сборке мусора.

Функции в библиотеке .NET

Библиотека .NET содержит огромное количество общих функций, помогающих разработчикам решать множество повседневных задач. Большинство из этих классов и функций оптимизировано для общих случаев, что не делает их оптимальными для каждой конкретной ситуации. Однако библиотечные классы можно заменять пользовательскими реализациями, лучше подходящими для конкретного случая.

В библиотеке .NET имеются две основные функции, которые часто вызывают значительное падение производительности при их использовании. Это вызвано тем, они были созданы только как быстрое и частичное решение данной проблемы, без приложения особых усилий для лучшей оптимизации. Этими функциями являются LINQ и регулярные выражения.

LINQ – это инструмент обработки массивов как миниатюрной базы данных, выполнения запросов, написанных на SQL-подобном языке. Простота стиля кодирования и сложность базовой системы (из-за использования замыканий) определяют связанные с ним значительные накладные расходы. LINQ – удобный инструмент, но он не предназначен для получения высокой производительности, требуемый приложениям реального времени, таким как игры, и не работает на платформах, не поддерживающих JIT-компиляцию, таких как iOS.

Регулярные выражения, использующие класс `Regex`, позволяют выполнять сложный анализ строк с целью поиска совпадений с определенным шаблоном, замены строк или сборки строк из различных исходных элементов. Регулярное выражение – это еще один очень полезный инструмент, которым, впрочем, не стоит злоупотреблять, если в нем нет особой необходимости или если его использование кажется на первый взгляд «умным» подходом к реализации функций, таких как локализация текстов, когда простая замена в строке гораздо более эффективна.

Описание конкретных способов оптимизации этих функций выходит далеко за рамки данной книги, поскольку только ему можно посвятить целые тома. Старайтесь использовать их как можно реже,

заменяйте чем-то менее дорогостоящим, обратитесь к экспертам по LINQ и Regex для решения возникших проблем или выполните поиск в Google материалов по их оптимизации.

Один из лучших способов найти в сети правильный ответ – разместить неправильный вопрос! Люди либо помогут вам по доброте душевной, либо их настолько возмутит ваша реализация, что они сочтут своим гражданским долгом поправить вас! Просто не забудьте начать с некоторого самостоятельного исследования этого вопроса. Даже самые занятые люди обычно рады помочь, если убедятся, что на решение вопроса уже были затрачены определенные усилия.

Временные рабочие буферы

Если у вас вошло привычку использовать для решения задач большие временные буферы, имеет смысл организовать повторное их использование вместо размещения в памяти снова и снова, так как это снижает накладные расходы на распределение, а также на сборку мусора (так называемую «нагрузку на память»). Такую функциональность целесообразно вынести за пределы конкретных классов и поместить ее в общий вездесущий класс, содержащий большие рабочие области многократного использования.

Пулы объектов

Подобно временным рабочим буферам, пулы объектов являются отличным способом минимизации и контроля над использованием памяти, позволяющим избежать ее освобождения и повторного распределения. Идея заключается в разработке собственной системы создания объектов, позволяющей скрывать объекты, для которых уже была выделена память, или повторно использовать ранее размещенные. Для описания этого процесса часто используются термины «активация» и «деактивация» объектов, а не «создание» и «удаление», поскольку при деактивации они просто переводятся в неактивное состояние, пока не понадобятся вновь.

Рассмотрим быструю реализацию системы пула объектов.

Первая задача – обеспечить поддержку пула в объектах, которые должны сами себя активировать в нужный момент времени. Ее прекрасно решает следующий интерфейс:

```
public interface IPoolableObject{
    void New();
    void Respawn();
}
```

Этот интерфейс определяет два метода: New() и Respawn(). Они вызываются при первоначальном создании объекта и повторной активации соответственно.

Вторая задача – базовая реализация интерфейса, позволяющая учитывать любые требования при первоначальном создании и активации объектов.

Следующее определение класса ObjectPool является простейшей реализацией идеи пула объектов. В ней используются универсальные шаблоны для поддержки объектов любого типа, соответствующих двум критериям: они должны реализовать интерфейс IPoolableObject и включать конструктор без параметров (ключевое слово new() в объявлении класса).

```
public class ObjectPool<T> where T : IPoolableObject, new() {
    private Stack<T> _pool;
    private int _currentIndex = 0;
    public ObjectPool(int initialCapacity) {
        _pool = new Stack<T>(initialCapacity);
        for(int i = 0; i < initialCapacity; ++i) {
            Spawn (); // создание экземпляра пула N объектов
        }
        Reset ();
    }

    public int Count {
        get { return _pool.Count; }
    }
    public void Reset() {
        _currentIndex = 0;
    }
    public T Spawn() {
        if (_currentIndex < Count) {
            T obj = _pool.Pop ();
            _currentIndex++;

            IPoolableObject ip = obj as IPoolableObject;
            ip.Respawn();

            return obj;
        } else {
            T obj = new T();
            _pool.Push(obj);
            _currentIndex++;

            IPoolableObject ip = obj as IPoolableObject;
            ip.New();
            return obj;
```

```
        }
    }
}
```

Ниже приводится пример объекта с поддержкой пула. Он должен реализовать два общедоступных метода – `New()` и `Respawn()`, – которые вызываются классом `ObjectPool` в нужные моменты времени:

```
public class TestObject : IPoolableObject {
    public void New() {
        // первоначальная инициализация
    }
    public void Respawn() {
        // реализация восстановления данных перед
        //  повторным использованием объекта
    }
}
```

И наконец, пример создания пула с сотней объектов `TestObject`:

```
private ObjectPool<TestObject> _objectPool = new ObjectPool<TestObject>(100);
```

Первые 100 вызовов метода `Spawn()` объекта `_objectPool` приведут к активации объектов и передаче их вызывающему. Если хватит места в стеке, можно разместить и большее количество объектов `TestObject`. Наконец, метод `Reset()` объекта `_objectPool` воссоздает объекты и возвращает их вызывающему коду.

Обратите внимание, что такое решение не будет работать с еще не определенными классами и не наследуется, подобно классам `Vector3` и `Quaternion`. Для этих случаев необходимо определить контейнерный класс:

```
public class PoolableVector3 : IPoolableObject {
    public Vector3 vector = new Vector3();

    public void New() {
        Reset();
    }

    public void Respawn() {
        Reset();
    }

    public void Reset() {
        vector.x = vector.y = vector.z = 0f;
    }
}
```

Эту систему можно расширить в разных направлениях, например определить метод `Despawn()` для уничтожения объекта, предусмотреть использование интерфейса `IDisposable` и блоков `using` для автоматической активации и деактивации объектов в небольшой области или обеспечить добавление в пул не входящих в него объектов.

Пулы шаблонных объектов

Приведенное выше решение организации пула предназначено для обычных классов, оно не будет работать со специальными объектами Unity, такими как объекты `GameObject` и `MonoBehaviour`. А именно эти объекты обычно потребляют значительную часть памяти во время выполнения, значительное количество ресурсов центрального процессора при их создании и уничтожении и увеличивают риск сборки больших объемов мусора во время выполнения. Иными словами, главной целью организации пулов шаблонных объектов является создание подавляющего большинства экземпляров объектов во время инициализации сцены взамен их создания во время выполнения. Это может обеспечить значительную экономию ресурсов центрального процессора и избавить от массы неприятностей, вызванных созданием/уничтожением объектов и сборкой мусора, за счет увеличения времени загрузки сцены и потребления памяти во время выполнения. Как результат в Asset Store появилось достаточно много решений для организации пула различной сложности, качества и возможностей.

Применение пулов особенно рекомендуется в играх для мобильных устройств из-за больших накладных расходов на выделение и освобождение памяти, по сравнению с настольными приложениями.

Однако организация пула – очень интересная тема, и реализация подобного решения с нуля дает отличную возможность досконально разобраться во внутренних особенностях движка Unity. Кроме того, знание устройства такой системы облегчит работу с ней для удовлетворения потребностей конкретной игры, в отличие от использования готовых решений.

Общая идея организации пулов шаблонных объектов заключается в создании системы, содержащей списки активных и неактивных игровых объектов, созданных из одной и той же ссылки на шаблонный объект. На рис. 7.4 представлена диаграмма, иллюстрирующая,

как выглядит система пулов после нескольких циклов активации/деактивации различных объектов, созданных на основе четырех разных шаблонных объектов (орка Orc, тролля Troll, огра Ogre и дракона Dragon).

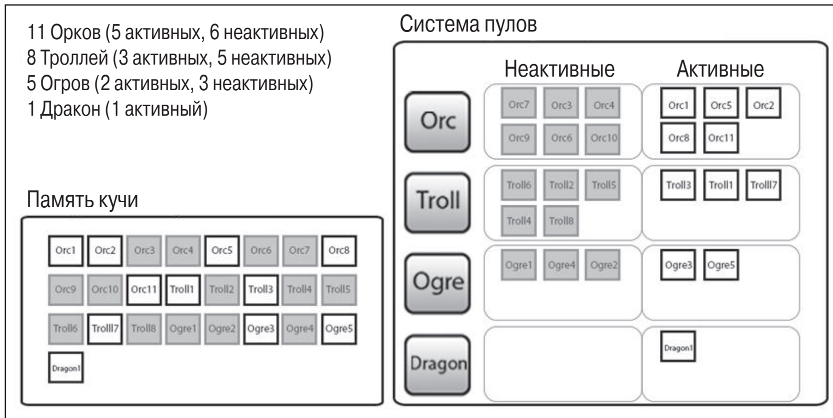


Рис. 7.4 ❖ Состояние системы после нескольких появлений, исчезновений и повторных появлений различных объектов

Обратите внимание, что в памяти кучи находятся все существующие объекты, в то время как в пуле находятся ссылки на эти же объекты.

В этом примере создано несколько экземпляров каждого из шаблонных объектов (11 орков, 8 троллей, 5 огров и 1 дракон). В настоящее время только одиннадцать из них активны, в то время как другие четырнадцать ранее были деактивированы. Обратите внимание, что неактивные объекты остаются в памяти, хотя они не видны и не могут взаимодействовать с игровым миром, пока не будут активированы вновь. Естественно, поддержка неактивных объектов во время выполнения обеспечивается выделением постоянного количества памяти в куче, но при создании нового объекта повторно используется один из существующих неактивных объектов и для удовлетворения запроса не требуется *дополнительная* память. Это значительно уменьшает потребление ресурсов центрального процессора во время создания и уничтожения объектов и позволяет избежать сборки мусора.

На рис. 7.5 показана цепочка событий при появлении нового орка.

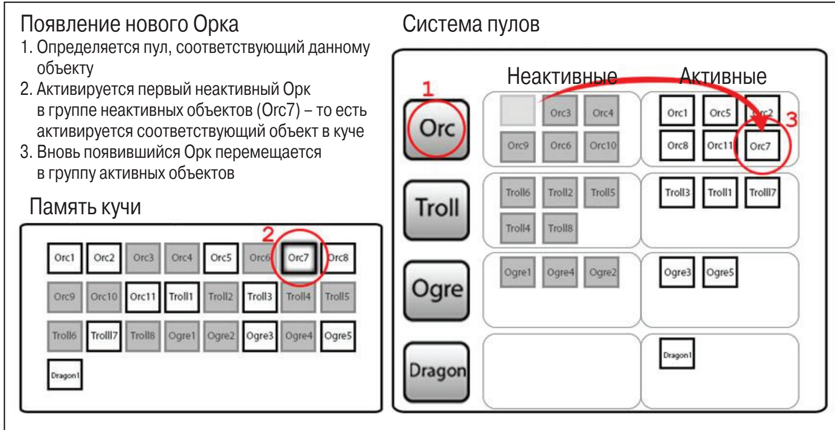


Рис. 7.5 ❖ Цепочка событий при активации нового орка

Первый неактивный орк в группе неактивных объектов (Orc7) становится активным и перемещается в группу активных объектов. Теперь 6 орков активны и 5 неактивны.

На рис. 7.6 показан порядок событий, сопровождающих деактивацию огра.

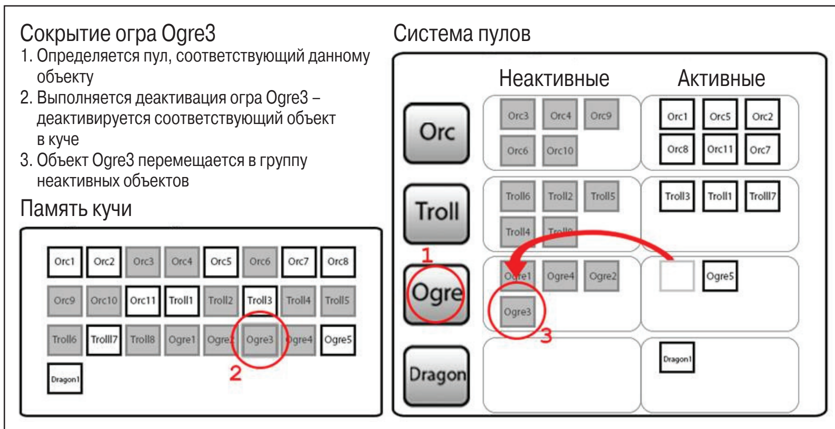


Рис. 7.6 ❖ Порядок событий, сопровождающих деактивацию огра

На этот раз объект деактивируется и перемещается из группы активных в группу неактивных объектов, после чего остается 1 активный огр и 4 неактивных.

И наконец, на рис. 7.7 показана череда событий, связанных с активацией нового объекта, когда в пуле нет готовых неактивных объектов.

Рис. 7.7 ❖ Активация нового объекта в отсутствие неактивных объектов в пуле

В этом случае выделяется еще память для нового объекта дракона `Dragon`, поскольку неактивных и готовых к повторному использованию объектов `Dragon` в пуле нет. Именно поэтому, чтобы избежать выделения памяти под игровые объекты во время выполнения, важно заранее знать, сколько их понадобится. Это будет зависеть от типа объекта, кроме того, потребуются тестирование и отладка, чтобы убедится, что имеется разумное количество экземпляров каждого шаблонного объекта.

Учитывая все это, создадим систему пула для шаблонных объектов!

Компоненты пула

Начнем с определения интерфейса компонента пула:

```
public interface IPoolableComponent {
    void Spawned();
    void Despawned();
}
```

Реализация интерфейса IPoolableComponent значительно отличается от реализации интерфейса IPoolableObject. На этот раз создаваться будут игровые объекты, с которыми гораздо сложнее работать, чем со стандартными объектами, поскольку их поведение во время выполнения обеспечивается движком Unity, доступ к которому сильно ограничен.

Игровые объекты не предоставляют доступа к методу, эквивалентному методу New(), который можно вызвать для создания объекта, и для его реализации нельзя унаследовать класс GameObject. Игровые объекты создаются либо путем размещения их на сцене, либо путем создания экземпляров во время выполнения вызовом метода GameObject.Instantiate(), и им можно задать только начальное положение и поворот. Конечно, их компоненты имеют метод Awake(), вызываемый при первом вводе компонента в действие, который можно определить, но это просто компонент, а не родительский объект, который должен активироваться или деактивироваться.

Итак, поскольку нам доступны только компоненты класса GameObject, предполагается, что интерфейс IPoolableComponent реализуется, *по крайней мере, для одного* из компонентов, прикрепленных к игровому объекту, который требуется включить в пул.

Метод Spawned() должен вызываться для каждой реализации компонента всякий раз, когда игровой объект, входящий в пул, активируется, а метод Despawned() – когда деактивируется. Это дает нам точки входа для управления переменными и поведением во время создания и уничтожения родительского объекта игры.

Действия, выполняемые в случае деактивации игрового объекта, весьма тривиальны: нужно сбросить флаг активности в false (вызовом SetActive()), деактивировать компоненты физических расчетов Collider и Rigidbody, удалить объект из списка отображаемых и тем самым полностью отключить от взаимодействий со всеми встроенными подсистемами движка Unity. Исключением являются только сопрограммы, содержащие ссылки на этот объект, поскольку, как было описано в *главе 2 «Приемы разработки сценариев»*, сопрограммы вызываются независимо от вызовов методов Update() и активности игровых объектов. Поэтому необходимо дополнительно вызвать метод StopCoroutine() или StopAllCoroutines().

Кроме того, компоненты обычно привязываются к пользовательским подсистемам, реализующим логику игры, поэтому метод Despawn() дает возможность компонентам позаботиться об очистке таких связей перед отключением. Например, метод Despawn() можно

использовать для отмены регистрации компонента в системе обмена сообщениями, описанной в *главе 2 «Приемы разработки сценариев»*.

К сожалению, повторная активация объекта реализуется сложнее. Объект имеет множество параметров, значения которых сохраняются с момента, когда объект был активным прежде, и они должны быть сброшены, чтобы избежать неувязок в его поведении. Наиболее часто проблемы возникают со скоростью движения в компоненте Rigidbody. Если это значение не сбросить явно до активации объекта, вновь активированный объект будет продолжать двигаться с той же скоростью, которую он имел до исчезновения.

Эта проблема еще больше осложняется тем, что встроенные компоненты наглухо закрыты и их невозможно унаследовать. Поэтому, чтобы избежать проблем, можно создать пользовательский компонент, сбрасывающий присоединенный компонент Rigidbody всякий раз при появлении:

```
public class ResetPooledRigidbodyComponent : MonoBehaviour,
IPoolableComponent {
    Rigidbody _body;
    public void Spawned() { }
    public void Despawned() {
        if (_body == null) {
            _body = GetComponent<Rigidbody>();
            if (_body == null) {
                // нет компонента Rigidbody!
                return;
            }
        }
        _body.velocity = Vector3.zero;
        _body.angularVelocity = Vector3.zero;
    }
}
```

Обратите внимание, что лучше всего эту задачу выполнять в момент деактивации, поскольку заранее неизвестно, в каком порядке будут вызываться методы Spawned() компонентов игрового объекта. Маловероятно, что другой интерфейс IPoolableComponent изменит скорость объекта во время деактивации, но вполне возможно, что другому интерфейсу IPoolableComponent, присоединенному к тому же объекту, потребуется установить нужную начальную скорость в компоненте Rigidbody при вызове метода Spawned(). Следовательно, сброс скорости в методе Spawned() класса ResetPooledRigidbodyComponent мо-

жет привести к конфликту с другими компонентами и стать причиной плохо выявляемой ошибки.

В самом деле, создание компонентов, поддерживающих пулы, которые не являются автономными и склонны взаимодействовать с другими компонентами подобным способом, таит в себе большую опасность при реализации системы пулов. Следует свести к минимуму подобные реализации и регулярно проверять их при отладке странных проблем, возникающих в игре.

В целях иллюстрации ниже приводится определение простого компонента с поддержкой пула, который заменяет класс `TestMessageListener`, описанный в *главе 2 «Приемы разработки сценариев»*. Этот компонент автоматически выполняет несколько базовых операций при активации/деактивации:

```
public class PoolableTestMessageListener : MonoBehaviour, IPoolableComponent
{
    public void Spawned() {
        MessagingSystem.Instance.AttachListener
        (typeof(MyCustomMessage), this.HandleMyCustomMessage);
    }

bool HandleMyCustomMessage(BaseMessage msg) {
        MyCustomMessage castMsg = msg as MyCustomMessage;
        Debug.Log (string.Format("Got the message! {0}, {1}",
        castMsg._intValue, castMsg._floatValue));
        return true;
    }

    public void Despawned() {
        if (MessagingSystem.IsAlive) {
            MessagingSystem.Instance.DetachListener(
                typeof(MyCustomMessage),
                this.HandleMyCustomMessage);
        }
    }
}
```

Система пулов шаблонных объектов

Надеюсь, теперь вы получили полное понимание требований к системе пулов, так что осталось только ее реализовать. Эти требования заключаются в следующем:

- система должна принимать запросы на активацию экземпляров шаблонных объектов в начальной позиции и с начальным поворотом:

  - если существует неактивная версия, активировать ее;
  - если ее не существует, создать новый экземпляр игрового объекта из шаблонного объекта;
  - в любом случае во всех интерфейсах `IPoolableComponent`, присоединенных к игровому объекту, должен быть вызван метод `Spawned()`;
- система должна принимать запросы на деактивацию конкретных игровых объектов:
  - если объект управляется пулом, его следует деактивировать и вызвать метод `Despawned()` для всех интерфейсов `IPoolableComponent`, присоединенных к объекту;
  - если объект не управляется пулом, необходимо вывести сообщение об ошибке.

Требования просты, но их реализация требует некоторого исследования, чтобы сделать решение более производительным. Во-первых, хорошим выбором для реализации главной точки входа может стать обычный синглтон, так как система должна быть глобально доступна из любой точки игрового мира:

```
public static class PrefabPoolingSystem {
}
```

Основная задача активации объекта включает получение ссылки на шаблонный объект и поиск соответствующего неактивного объекта. Для решения этой задачи система должна хранить два списка со ссылками на экземпляры шаблонных объектов: активных и неактивных. Эти данные лучше выделить в отдельный класс, который мы назовем `PrefabPool`.

Для увеличения производительности системы (и, следовательно, получения максимальной выгоды относительно обычного выделения и освобождения памяти под объекты) используем некоторые структуры данных для извлечения соответствующих объектов `PrefabPool` при поступлении запроса на появление или исчезновение.

Поскольку запрос на активацию включает ссылку на шаблонный объект, нам понадобится структура данных, которая позволит быстро найти соответствующий пул `PrefabPool` по шаблонному объекту. А так как запрос на деактивацию включает конкретный игровой объект, нужна еще одна структура данных, которая позволит быстро найти соответствующий пул `PrefabPool` по игровому объекту. Для удовлетворения обеих потребностей отлично подойдет словарь.

Определим эти словари в системе пулов:

```
public static class PrefabPoolingSystem {
    static Dictionary<GameObject,PrefabPool> _prefabToPoolMap =
        new Dictionary<GameObject,PrefabPool>();
    static Dictionary<GameObject,PrefabPool> _goToPoolMap =
        new Dictionary<GameObject,PrefabPool>();
}
```

Затем определим действия, выполняемые при активации объекта:

```
public static GameObject Spawn(GameObject prefab, Vector3 position,
Quaternion rotation) {
    if (!_prefabToPoolMap.ContainsKey (prefab)) {
        _prefabToPoolMap.Add (prefab, new PrefabPool());
    }
    PrefabPool pool = _prefabToPoolMap[prefab];
    GameObject go = pool.Spawn(prefab, position, rotation);
    _goToPoolMap.Add (go, pool);
    return go;
}
```

Метод `Spawn()` получает ссылку на шаблонный объект, начальное положение и поворот. Он должен найти пул `PrefabPool`, соответствующий шаблонному объекту (если таковой вообще имеется), дать ему команду активировать новый игровой объект с использованием предоставленных данных, а затем вернуть объект вызывающей программе. Итак, сначала с помощью словаря «шаблонные объекты/пулы» выполняется поиск пула. Если пул не найден, его следует создать. Затем нужно дать команду объекту класса `PrefabPool` активировать новый объект. В ответ объект `PrefabPool` либо активирует объект, деактивированный прежде, либо создаст новый (если не осталось ни одного неактивного экземпляра).

В любом случае этому классу просто требуется получить экземпляр `PrefabPool` из словаря «игровые объекты /пулы» и вернуть запросившему.

Для удобства можно также определить перегруженный метод, помещающий объект в центр игрового мира (полезно для невидимых игровых объектов, которые просто должны где-то существовать):

```
public static GameObject Spawn(GameObject prefab) {
    return Spawn (prefab, Vector3.zero, Quaternion.identity);
}
```

Обратите внимание, что собственно активация и деактивация пока не реализованы. Эта задача будет решаться в классе `PrefabPool`.

Для деактивации задается игровой объект, по которому определяется контролирующий его объект класса `PrefabPool`. Это можно сделать путем перебора объектов `PrefabPool` и проверки присутствия заданного объекта `GameObject`. Но когда имеется много объектов класса `PrefabPool`, этот итеративный процесс может занять определенное время. В конечном итоге количество объектов класса `PrefabPool` будет соответствовать количеству шаблонных объектов (по крайней мере, если они управляются через систему пулов). Большинство проектов, как правило, содержат десятки, сотни, если не тысячи, различных шаблонных объектов.

Поэтому для быстрого доступа к объекту пула `PrefabPool`, где находится активный объект, используется словарь «игровые объекты/пулы». С его помощью также можно быстро проверить, контролировался ли данный объект игры системой пулов изначально. Ниже приводится определение метода деактивации, решающего эти задачи:

```
public static bool Despawn(GameObject obj) {
    if (!_goToPoolMap.ContainsKey(obj)) {
        Debug.LogError (string.Format (
            "Object {0} not managed by pool system!", obj.name));
        return false;
    }

    PrefabPool pool = _goToPoolMap[obj];
    if (pool.Despawn (obj)) {
        _goToPoolMap.Remove (obj); return true;
    }
    return false;
}
```

Обратите внимание, что метод `Despawn()` в классах `PrefabPoolingSystem` и `PrefabPool` возвращает логический признак успешной деактивации объекта.

Благодаря поддержке двух словарей мы получаем быстрый доступ к объекту пула `PrefabPool`, управляющему заданной ссылкой, и это решение подойдет для любого числа шаблонных объектов, управляемых системой.

Пулы шаблонных объектов

На настоящий момент у нас имеется система, автоматически поддерживающая несколько пулов шаблонных объектов, осталось только определить их поведение. Как упоминалось выше, класс PrefabPool должен управлять двумя структурами данных: одна – для активных объектов, созданных из заданного шаблонного объекта, и другая – для неактивных объектов.

Технически класс PrefabPoolingSystem уже поддерживает словари, управляемые объектами PrefabPool, поэтому можно сэкономить немного памяти, подчинив класс PrefabPool классу PrefabPoolingSystem, не нарушая порядка управления шаблонными объектами. Поэтому две структуры данных определены как простые свойства класса PrefabPool.

Однако для каждого объекта, помещаемого в пул, этот класс должен поддерживать список всех его ссылок на интерфейсы IPoolableComponent, чтобы вызывать его методы Spawned() и Despawned(). Получение может обойтись очень дорого, поэтому имеет смысл кэшировать эти данные в простую структуру:

```
public struct PoolablePrefabData {
    public GameObject go;
    public IPoolableComponent[] poolableComponents;
}
```

Она содержит ссылку на объект GameObject и список его интерфейсов IPoolableComponents.

Теперь можно определить свойства класса PrefabPool:

```
public class PrefabPool {
    Dictionary<GameObject,PoolablePrefabData> _activeList = new
    Dictionary<GameObject,PoolablePrefabData>();
    Queue<PoolablePrefabData> _inactiveList = new
    Queue<PoolablePrefabData>();
}
```

Структура данных для списка активных объектов должна быть словарем, чтобы быстро находить соответствующий объект PoolablePrefabData по любой ссылке на GameObject. Это понадобится для быстрой деактивации объекта.

Структура с неактивными объектами, напротив, определяется как очередь, но с этой ролью одинаково хорошо справился бы список, стек и практически любая структура данных, способная расширять-

ся и сжиматься, обеспечивающая быстрый доступ к последнему поступившему элементу, потому что выбор того или иного неактивного объекта не имеет значения. Важно только получить один из них. В данном случае очередь полезна еще и тем, что позволяет извлекать и удалять элементы единственным вызовом.

Активация объектов

Давайте определимся, что означает активация игрового объекта в контексте системы пулов: в определенный момент объект PrefabPool получит запрос на активацию игрового объекта по ссылке на шаблонный объект, а также координаты в игровом мире и поворот. Получив такой запрос, он должен, во-первых, проверить наличие неактивных экземпляров этого шаблонного объекта. Если такие имеются – извлечь первый доступный и активировать его. Если нет – создать новый объект GameObject из шаблонного объекта вызовом метода GameObject.Instantiate(). В этот момент он должен также создать объект класса PoolablePrefabData для хранения ссылки на игровой объект и получить список прикрепленных к нему интерфейсов IPoolableComponents.

Теперь требуется активировать игровой объект, установить его положение и поворот и вызвать метод Spawned() для всех его интерфейсов IPoolableComponent. После активации объект следует добавить в список активных объектов и вернуть запросившему.

Ниже приводится определение метода Spawn(), который реализует это:

```
public GameObject Spawn(GameObject prefab, Vector3 position, Quaternion
rotation) {
    PoolablePrefabData data;
    if (_inactiveList.Count > 0) {
        data = _inactiveList.Dequeue();
    } else {
        // создать новый экземпляр объекта
        GameObject newGO = GameObject.Instantiate(prefab,
            position, rotation) as GameObject;
        data = new PoolablePrefabData();
        data.go = newGO; data.poolableComponents =
            newGO.GetComponents<IPoolableComponent>();
    }

    data.go.SetActive (true);
    data.go.transform.position = position;
    data.go.transform.rotation = rotation;
```

```
    for(int i = 0; i < data.poolableComponents.Length; ++i) {
        data.poolableComponents[i].Spawned ();
    }
    _activeList.Add (data.go, data);

    return data.go;
}
```

Предварительное создание экземпляров

Поскольку метод `GameObject.Instantiate()` вызывается всякий раз, когда в пуле заканчиваются неактивные экземпляры, система не исключает создания экземпляров во время выполнения и, следовательно, не дает возможности полностью избежать выделения памяти в куче. Поэтому важно предусмотреть предварительное создание прогнозируемого количества экземпляров, которые понадобятся в течение существования текущей сцены, чтобы не создавать их во время выполнения.

Весьма расточительным станет решение создать эффекты частиц для 100 взрывов, если одновременно в сцене могут присутствовать лишь три или четыре. И наоборот, создание недостаточного количества экземпляров приведет к слишком большому числу операций выделения памяти во время выполнения, в то время как целью этой системы является принудительное распределение большей части памяти при запуске сцены. Будьте осторожны, выбирая количество экземпляров, постоянно находящихся в памяти, чтобы не затратить на это больше памяти, чем необходимо.

Определим в классе `PrefabPoolingSystem` метод для быстрого создания заданного числа экземпляров из шаблонных объектов. По сути, он создает N экземпляров, а затем сразу же деактивирует их:

```
public static void Prespawn(GameObject prefab, int numToSpawn) {
    List<GameObject> spawnedObjects = new List<GameObject>();

    for(int i = 0; i < numToSpawn; i++) {
        spawnedObjects.Add (Spawn (prefab));
    }

    for(int i = 0; i < numToSpawn; i++) {
        Despawn(spawnedObjects[i]);
    }

    spawnedObjects.Clear ();
}
```

Этот метод можно вызывать во время инициализации сцены для создания коллекции объектов, которые могут понадобиться позднее. Например:

```
public class OrcPreSpawner : MonoBehaviour
    [SerializeField] GameObject _orcPrefab;
    [SerializeField] int _numToSpawn = 20;
    void Start() {
        PrefabPoolingSystem.Prespawn(_orcPrefab, _numToSpawn);
    }
}
```

Деактивация объектов

Как уже упоминалось выше, деактивация прежде всего предполагает отключение объекта, но следует также позаботиться о различных учетных заданиях и вызове метода `Despawned()` для всех ссылок на интерфейс `IPoolableComponent`.

Ниже приводится определение метода `Despawn()` в классе `PrefabPool`:

```
public bool Despawn(GameObject objToDespawn) {
    if (!_activeList.ContainsKey(objToDespawn)) {
        Debug.LogError ("This Object is not managed by this object pool!");
        return false;
    }

    PoolablePrefabData data = _activeList[objToDespawn];

    for(int i = 0; i < data.poolableComponents.Length; ++i) {
        data.poolableComponents[i].Despawned ();
    }
    data.go.SetActive (false);

    _activeList.Remove (objToDespawn);
    _inactiveList.Enqueue(data);
    return true;
}
```

Он сначала проверяет присутствие объекта в пуле, а затем извлекает соответствующий объект класса `PoolablePrefabData` для доступа к списку ссылок на интерфейс `IPoolableComponent`. После вызова метода `Despawned()` всех интерфейсов объект деактивируется, удаляется из списка активных и помещается в очередь неактивных объектов для активации позднее.

Тестирование пула шаблонных объектов

Следующий класс позволит протестировать класс PrefabPoolingSystem. Он поддерживает три шаблонных объекта и создает по пять экземпляров каждого во время инициализации приложения. Активацию экземпляров каждого типа можно выполнять клавишами *1*, *2* и *3*, а деактивацию случайно выбранных экземпляров каждого типа – клавишами *Q*, *W* и *E*.

```
public class PoolTester : MonoBehaviour {

    [SerializeField] GameObject _prefab1;
    [SerializeField] GameObject _prefab2;
    [SerializeField] GameObject _prefab3;

    List<GameObject> _go1 = new List<GameObject>();
    List<GameObject> _go2 = new List<GameObject>();
    List<GameObject> _go3 = new List<GameObject>();

    void Start() {
        PrefabPoolSystem_AsSingleton.Prespawn(_prefab1, 5);
        PrefabPoolSystem_AsSingleton.Prespawn(_prefab2, 5);
        PrefabPoolSystem_AsSingleton.Prespawn(_prefab3, 5);
    }

    void Update () {
        if (Input.GetKeyDown(KeyCode.Alpha1))
            {SpawnObject(_prefab1, _go1);}
        if (Input.GetKeyDown(KeyCode.Alpha2))
            {SpawnObject(_prefab2, _go2);}
        if (Input.GetKeyDown(KeyCode.Alpha3))
            {SpawnObject(_prefab3, _go3);}
        if (Input.GetKeyDown(KeyCode.Q)) { DespawnRandomObject (_go1); }
        if (Input.GetKeyDown(KeyCode.W)) { DespawnRandomObject (_go2); }
        if (Input.GetKeyDown(KeyCode.E)) { DespawnRandomObject (_go3); }
    }

    void SpawnObject(GameObject prefab, List<GameObject> list) {
        GameObject obj = PrefabPoolingSystem.Spawn (prefab,
            Random.insideUnitSphere * 8f, Quaternion.identity);
        list.Add (obj);
    }

    void DespawnRandomObject(List<GameObject> list) {
        if (list.Count == 0) {
            // Нет объектов для деактивации
            return;
```

```
        }

        int i = Random.Range (0, list.Count);
        PrefabPoolingSystem.Despawn(list[i]);
        list.RemoveAt(i);
    }
}
```

После активации пяти экземпляров одного из шаблонных объектов, чтобы активировать еще один, потребуется создать новый экземпляр в памяти, что выльется в необходимость распределения памяти. Но если понаблюдать за областью использования памяти в профилировщике, можно убедиться, что активация и деактивация уже существующих экземпляров не вызывают абсолютно никаких распределений.

Организация пулов шаблонных объектов и загрузка сцены

Эта система имеет еще один тонкий нюанс, который не был упомянут: класс `PrefabPoolingSystem` продолжит свое существование после уничтожения сцены, поскольку это статический класс. Это означает, что при загрузке новой сцены словари системы пулов сохранят ссылки на все экземпляры из предыдущей сцены, помещенные в пулы, хотя Unity принудительно уничтожит эти объекты независимо от наличия ссылок на них (если только к ним не был применен метод `DontDestroyOnLoad()`!), и поэтому словари окажутся заполненными ссылками со значением `null`. Это вызовет серьезные проблемы в следующей сцене.

Поэтому нужно создать в классе `PrefabPoolingSystem` метод инициализации системы пулов для подготовки к таким изменениям. Следующий метод нужно вызывать перед загрузкой новой сцены, чтобы он подготовил следующую сцену к вызовам методов `Prespawn()`:

```
public static void Reset() {
    _prefabToPoolMap.Clear ();
    _goToPoolMap.Clear ();
}
```

Обратите внимание, что при принудительном вызове сборки мусора во время смены сцен отпадает необходимость явной очистки ссылок на объекты `PrefabPools` в словарях. Так как это только ссылки на объекты `PrefabPool`, они будут освобождены во время следующей сборки мусора. Если при смене сцен сборка мусора не выполняется,

объекты `PrefabPool` и `PooledPrefabData` останутся в памяти до следующей сборки мусора.

Итоговые замечания об организации пулов шаблонных объектов

Итак, проблема выделения памяти для игровых и шаблонных объектов решена, но для применения данного решения следует помнить:

- о важности сброса данных в активируемых объектах (например, скорость в компоненте Rigidbody);
- о необходимости создания не слишком малого и не слишком большого количества экземпляров шаблонных объектов;
- об осторожности относительно порядка выполнения методов `Spawned()` и `Despawned()` интерфейсов `IPoolableComponents;`
- о необходимости вызвать метод `Reset()` объекта `PrefabPoolingSystem` перед загрузкой сцены.

Дополнительно можно было бы реализовать еще несколько функций. Это станет вашим самостоятельным упражнением, если вы захотите расширить данную систему.

- Компоненты `IPoolableComponent`, добавленные в игровой объект после инициализации, не будут вызываться. Это можно исправить, добавив в класс `PrefabPool` сохранение ссылок на компоненты `IPoolableComponent` при каждом вызове `Spawned()` и `Despawned()`, правда, это увеличит накладные расходы во время активации/деактивации.
- Интерфейсы `IPoolableComponent`, присоединенные к дочерним объектам корневого шаблонного объекта, не будут учитываться. Это можно исправить, если задействовать в классе `PrefabPool` метод `GetComponentsInChildren<T>`, но это увеличит накладные расходы при использовании шаблонных объектов с глубокой иерархией.
- Экземпляры шаблонных объектов, уже существующие в сцене, не будут управляться системой пулов. Можно создать компонент, который будет прилагаться к таким объектам, уведомлять класс `PrefabPoolingSystem` о своем существовании и передавать ссылку в соответствующий объект `PrefabPool`.
- Можно в интерфейсе создать `IPoolableComponents` некое подобие приоритетов и устанавливать их в момент приобретения объекта, чтобы контролировать порядок выполнения методов `Spawned()` и `Despawned()`.

- Можно добавить счетчики, отслеживающие продолжительность нахождения объектов в списке неактивных по отношению к общей продолжительности существования сцены, и вывести данные при выключении. Это позволит оценить правильность выбора количества экземпляров каждого из шаблонных объектов.
- Эта система некорректно взаимодействует с экземплярами шаблонных объектов, для которых вызывался метод `DontDestroyOnLoad()`. Имеет смысл добавить логическое значение для каждого вызова `Spawn()`, определяющее необходимость сохранения данных об объекте при очистке вызовом метода `Reset()`.
- Можно изменить метод `Spawn()`, добавив в него дополнительные аргументы, позволяющие запросившему передавать пользовательские данные в функцию `Spawned()` интерфейса `IPoolableObject` в целях инициализации. Можно использовать систему, аналогичную системе обмена пользовательскими сообщениями, наследуемую от класса `BaseMessage`, как описывается в *главе 2 «Приемы разработки сценариев»*.

Дальнейшее развитие Mono и Unity

Как известно, Unity использует не самую последнюю и лучшую версию проекта Mono (http://www.mono-project.com), а внутренние нестандартные версии, содержащие определенное количество внутренних ошибок.

Описание изменений в Mono, внесенных разработчиками из Unity Technologies, можно найти в репозитории GitHub: https://github.com/Unity-Technologies/mono/.

Печальным следствием лицензирования различных компонентов фреймворка Mono является нечастая возможность обновлять его версию в Unity. Последний раз это произошло перед выпуском Unity 4 4, когда фреймворк Mono был обновлен до версии 2.6, а вскоре после этого, в середине 2010 года, до версии 2.6.5, которая поддерживает функции .NET 2.0/3.5. Но на момент публикации последней является версия Mono 4.0, выпущенная в мае 2015 года и поддерживающая функции .NET. Это привело к отставанию Unity примерно на 5 лет в отношении поддержки языка C# и фреймворка .NET, что вызывает негодование у многих Unity-разработчиков.

Unity Technologies предполагает обновить Mono для Unity 5, но последние официальные сообщения об этом появились еще в августе 2014 года. В планах (https://unity3d.com/unity/roadmap) это обновление помечено как отложенное на «длительный и неопределенный» срок, и выпущенная в начале сентября 2015 года версия 5.2 Unity не содержит этого обновления. Поэтому трудно сказать, когда появится столь необходимое обновление Mono. Unity Technologies также сотрудничает с Microsoft для изменения некоторых компонентов Mono (например, обновления версии статического, JIT- и AOT-компиляторов, а также среды CLR).

Microsoft анонсировала дальнейшее развитие .NET на http://blogs.msdn.com/b/dotnet/archive/2014/04/03/the-next-generation-of-net.aspx.

Тем временем Unity Technologies работает над задачей отказа в перспективе от сторонних зависимостей. С выходом Unity 5 появился новый подход к компиляции кода сценариев, который изначально задумывался как способ улучшения поддержки сценариев для Unity-приложений, основанных на WebGL, но затем был преобразован в базовое решение на основе IL2CPP.

Анонс IL2CPP Unity Technologies можно найти на http://blogs.unity3d.com/2014/05/20/the-future-of- scripting-in-unity/.

Аббревиатура **IL2CPP** расшифровывается как **Intermediate Language To C++** (из промежуточного языка в C++). IL2CPP – это отдельная среда выполнения .NET, которая может быть развернута на нескольких платформах. Основная идея заключается в переводе кода сценариев C# на промежуточный язык, а затем, в момент сборки, на C++. Полученный в результате код на C++ будет передаваться одному из доступных компиляторов, в зависимости от целевой платформы.

Этот процесс практически полностью скрыт от пользователя, что, естественно, приведет к потере контроля, так как код будет проходить сквозь последовательность «фильтров» с различными уровнями оптимизации. Еще предстоит выяснить, будут ли предусмотрены средства управления процессом компиляции для опытных разработчиков. Редактор по-прежнему будет запускать среду выполнения C#.NET для ускорения разработки.

Как предполагается, такое решение должно улучшить производительность (поскольку даже близкий к родному машинный код, созданный JIT-компилятором, все еще далек от статически скомпи-

лированного кода C++), обеспечить быструю адаптацию и учет особенностей разработки движка Unity, а также улучшение сборки мусора. В Unity 5 первой платформой для применения IL2CPP стала WebGL, но в конечном итоге планируется обеспечить поддержку всех других платформ, по мере развития этой системы и увеличения ее надежности.

За дополнительной информацией о IL2CPP обращайтесь к блогам Unity Technologies, посвященным этому вопросу. Следующая статья в одном из таких блогов содержит массу информации, а также дополнительные ссылки на прочие важные темы: http://blogs.unity3d.com/2015/05/06/an-introduction-to-ilcpp-internals/.

Почему Unity Technologies просто не предоставит прикладной программный интерфейс C++ для Unity? Об этом можно только догадываться, но наиболее правдоподобной причиной кажется неудобство взаимодействия с C++ напрямую для большинства Unity-разработчиков. Потеря доступа к языкам C# и UnityScript сильно изменит продукт. В долгосрочной перспективе это, скорее всего, разобьет клиентов на два лагеря, что не сулит ничего хорошего, поскольку направление, которое приносит наибольший доход, станет лучше поддерживаться за счет прочих (если провести интересную аналогию с играми, которые обычно страдают от той же проблемы, так как расширения и пакеты карт часто разделяют игроков многопользовательских версий).

Кроме того, существует множество .NET-языков, совместно использующих один и тот же промежуточный код (CIL) и имеющих общий двоичный интерфейс, что значительно облегчает поддержку совместимости для разных платформ и библиотек, что невозможно для C++. Предположительно IL2CPP является компромиссным решением.

Таким образом, если коротко, движок Unity коренным образом меняется, и слишком рано говорить о том, будет ли подход IL2CPP достаточно хорошо работать на всех платформах. Одно можно утверждать точно, что в ближайшие пару лет Unity ждут интересные перемены!

Итоги

В этой главе мы рассмотрели огромное количество теоретических понятий, которые должны пролить некоторый свет на внутреннее устройство движка Unity и работу языка C#. Эти инструменты стара-

ются избавить нас от тяжелого бремени управления памятью, но остается еще масса вопросов, которые следует иметь в виду при разработке игр: процесс компиляции, наличие нескольких областей памяти, типы значений и ссылочные типы, передача по значению и передача по ссылке, упаковка, организация пулов объектов и прочие особенности программного интерфейса Unity. Однако по мере приобретения опыта эти проблемы можно будет решать, не обращаясь к гигантским фолиантам, подобным этому!

В этой главе собраны все доступные методы, призванные улучшить производительность приложения. Оптимизация рабочего процесса всегда актуальна, но существует масса мелких нюансов, касающихся движка Unity, слишком малоизвестных и плохо документированных, о которых можно узнать только с приобретением опыта и при общении с другими членами сообщества. Поэтому следующая глава будет посвящена советам и подсказкам по эффективному управлению проектом и сценами, улучшению работы в редакторе. Следование им позволит сэкономить время на фактическую реализацию всех методов оптимизации, описанных в этой книге.

Глава 8

Тактические советы и подсказки

Имеется множество мелких нюансов, свзанных с использованием движка Unity, знание которых поможет оптимизировать процесс работы над проектом. Однако многие функции редактора недостаточно хорошо документированы, плохо известны или вообще действуют иначе, чем представляют многие разработчики до, пока не столкнутся с фактом, что одна из них идеально подходила для решения конкретной проблемы, с которой они столкнулись полгода назад.

Интернет забит статьями в блогах и на форумах, авторы которых пытаются помочь другим Unity-разработчикам освоить эти полезные функции, но эти советы, как правило, посвящены решению только сиюминутных проблем. В Интернете отсутствуют ресурсы, в которых эти советы были бы сконцентрированы в одном месте. В результате у пользователей среднего и продвинутого уровня уже не хватает места для закладок со ссылками на эти советы, позволяющими решать проблемы, и в итоге эти закладки бесполезно занимают место, ожидая очередной чистки.

Так как эта книга адресована главным образом пользователям среднего и продвинутого уровня, имеет смысл в этой короткой главе собрать такие советы и подсказки в одном месте. Получится что-то вроде списка ссылок, который позволит сэкономить массу времени при разработке.

Подсказки по клавишам быстрого доступа в редакторе

Редактор изобилует удобными клавиатурными комбинациями. О них можно прочесть в документации. Но, если честно, никто не читает руководство, пока оно не понадобится для решения какого-либо конкретного вопроса. Ниже приводится описание наиболее полезных, но мало известных клавиш быстрого доступа, которыми можно воспользоваться при работе в редакторе Unity.

Здесь приводятся комбинации для Windows. Если в OSX используется другая комбинация, она будет приведена в скобках.

Игровые объекты

Чтобы скопировать игровой объект, его следует выбрать в иерархии и нажать комбинацию клавиш **Ctrl+D** (**Cmd+D**).

Новый пустой игровой объект можно создать с помощью комбинации **Ctrl+Shift+N** (**Cmd+Shift+N**).

Комбинация **Ctrl+Shift+A** (**Cmd+Shift+A**) откроет меню **Add Component** (Добавить компонент), где можно ввести имя добавляемого компонента.

Представление сцены

Комбинация **Shift+F**, или двойное нажатие клавиши **F**, заставит найти объект в представлении сцены, что поможет отследить объект, движущийся с высокой скоростью, или выяснить, почему объект выпадает из сцены.

Удержание клавиши **Alt** при перетаскивании левой кнопкой мыши заставит камеру просмотра сцены вращаться вокруг выбранного объекта. Удержание клавиши **Alt** и перетаскивание правой кнопкой мыши изменяют масштаб камеры.

Удержание клавиши **Ctrl** и перетаскивание левой кнопкой мыши вызовут перемещение выбранного объекта с привязкой к сетке. То же самое можно сделать и для поворота, удерживая клавишу **Ctrl** при настройке поворота объекта. Если выбрать пункт **Edit ⇨ Snap Settings…** (Правка ⇨ Параметры привязки…), откроется окно, где можно изменить шаг сетки, к которой привязываются объекты по осям.

Мы можем заставить объекты привязываться по вершинам, удерживая клавишу **V** при перемещении. Выбранный объект будет привя-

зываться к вершине, ближайшей к указателю. Это очень полезно при выравнивании элементов на разных уровнях, например платформ, без необходимости ручной корректировки позиций векторов.

В версиях Unity с 4.2 по 4.6 можно было, удерживая клавишу Shift при выбранном объекте с коллайдером, вызвать появление небольших захватов, с помощью которых можно было перемещать коллайдер по представлению сцены. Эта возможность была исключена из последних версий Unity из-за конфликтов с другими элементами управления. Для доступа к этой функции используйте кнопку **Edit Collider** (Редактировать коллайдер) компонента коллайдера.

Массивы

Нажатием комбинации **Ctrl+D (Cmd+D)** можно скопировать элементы массива, выбранные в представлении **Inspector** (Инспектор). Скопированные элементы будут вставлены в массив сразу за текущим выделением.

Комбинацией **Shift+Delete (Cmd+Delete)** можно удалять объекты из массива ссылок (например, из массива игровых объектов) . Это приведет к исключению элемента и сжатию массива. Обратите внимание, что первое нажатие вызовет очистку ссылки и присвоит ей значение `null`, а второе – удалит элемент. Удаление элементов простых типов из массива (`int`, `float` и т. д.) можно выполнить нажатием клавиши **Delete**, без клавиши модификатора **Shift (Cmd)**.

Клавиши **W**, **A**, **S**, **D** можно использовать при перетаскивании камеры правой кнопкой мыши в представлении сцены, чтобы организовать ее облет в стиле обычного управления камерой персонажа от первого лица. Клавиши **Q** и **E**, соответственно, можно использовать для перемещения камеры по вертикальной оси.

Интерфейс

Удерживая клавишу **Alt** и щелкнув на любой стрелке в иерархии (маленькая серая стрелка слева от имени любого родительского объекта), можно раскрыть всю иерархию, а не только следующий уровень. Этот прием работает для игровых объектов в представлении **Hierarchy** (Иерархия), папок и шаблонных объектов в представлении **Project** (Проект), списков в представлении **Inspector** (Инспектор) и т. д.

Можно сохранить и восстановить выделение объектов в представлениях **Hierarchy** (Иерархия) и **Project** (Проект) в типичном стиле RTS-игры! Создайте выделение и нажмите комбинацию

Ctrl+**Alt**+<**0–9**> (**Cmd**+**Alt**+<**0–9**>) для сохранения выделения. Нажмите **Ctrl**+**Shift**+<**0–9**> (**Cmd**+**Shift**+<**0–9**>) для его восстановления. Это исключительно полезно при многократном выделении одних и тех же объектов.

Нажмите комбинацию **Shift**+**Пробел**, чтобы распахнуть текущее окно на весь экран редактора. Повторное нажатие приведет к восстановлению его предыдущего положения и размера.

Комбинация **Ctrl**+**Shift**+**P** (**Cmd**+**Shift**+**P**) переключает кнопку **Pause** (Пауза) в режиме воспроизведения.

Прочее

Доступ к документации с описанием любого класса или ключевого слова можно быстро получить, выделив его в MonoDevelop и нажав комбинацию **Ctrl**+**’** (**Cmd**+**’**). В результате откроется обозреватель по умолчанию и будет выполнен поиск данного ключевого слова или класса в документации.

Обратите внимание, что пользователям европейских клавиатур может также потребоваться удерживать нажатой клавишу **Shift**.

В последних версиях **Visual Studio Tools for Unity** (**VSTU**) появилась возможность доступа к документации, как это делается в Visual Studio, – нажатием комбинации **Ctrl**+**Alt**+**M**, вслед за которой следует нажать **Ctrl**+**H** (естественно, эквивалент для OSX отсутствует).

Советы, касающиеся интерфейса редактора

Следующая коллекция советов посвящена редактору и элементам управления его интерфейса.

Общие

Приоритеты выполнения методов Update и FixedUpdate сценариев можно определить, выполнив переход **Edit** ⇨ **Project Settings** ⇨ **Script Execution Order** (Правка ⇨ Параметры проекта ⇨ Порядок выполнения сценариев). За исключением некоторых систем, чувствительных ко времени выполнения, таких как обработка аудио, если возникнет необходимость решить проблемы с помощью этой функции, это означает, что между компонентами было создано непрочное и напряженное соединение. С точки зрения правильности разработки

программного обеспечения, это может послужить предупреждением, что следует взглянуть на проблему с другой стороны. Однако иногда полезно заставить функции `Update()` и `LateUpdate()` отдельных объектов вызываться перед функциями других объектов.

Порой бывает непросто интегрировать проект Unity с системой управления версиями. В таких случаях первым делом следует настроить в проекте принудительное создание файла ресурсов с расширением `.meta`, иначе всем, кто попытается поместить данные в локальный Unity-проект, придется создавать собственные файлы метаданных. Это легко может привести к конфликтам, поэтому важно, чтобы все использовали одни и те же их версии. Включить видимость файлов метаданных можно с помощью параметра **Edit ⇨ Project Settings ⇨ Editor ⇨ Version Control ⇨ Mode ⇨ Visible Meta Files** (Правка ⇨ Параметры проекта ⇨ Редактор ⇨ Контроль версий ⇨ Режим ⇨ Видимость файлов метаданных). Теперь все файлы метаданных с расширением `.meta` станут видимыми и доступными для загрузки в систему управления версиями.

Это также полезно для преобразования некоторых ресурсов в текстовый формат вместо двоичного, доступный для редактирования вручную. В результате многие файлы данных будут преобразованы в гораздо более удобочитаемый формат YAML. Например, если пользовательские данные хранятся в классе `ScriptableObjects`, мы сможем использовать текстовый редактор для поиска и редактирования его файлов без необходимости делать это в редакторе Unity. Это позволит сэкономить много времени, особенно когда речь идет о конкретном значении или о множественном изменении различных производных типов. Включить эту возможность можно с помощью параметра **Edit ⇨ Project Settings ⇨ Editor ⇨ Asset Serialization ⇨ Mode ⇨ Force Text** (Правка⇨ Параметры проекта ⇨ Редактор ⇨ Сериализация ресурсов ⇨ Режим ⇨ Принудительно в текст).

Редактор имеет файл регистрации, который можно открыть в окне консоли (куда выводятся регистрационные сообщения), щелкнув на значке «бутерброда» (с тремя горизонтальными линиями) в правом верхнем углу и выбрав пункт **Open Editor Log** (Открыть регистрационные сообщения редактора), как показано на рис. 8.1. Если проект собирался недавно, здесь будет содержаться перечень размеров сжатых файлов всех ресурсов, упакованных в исполняемый файл. Это очень удобный способ выяснить, какие ресурсы больше всего увеличивают размер файла приложения (подсказка: это почти всегда файлы текстур) и какие файлы занимают больше места, чем должны.

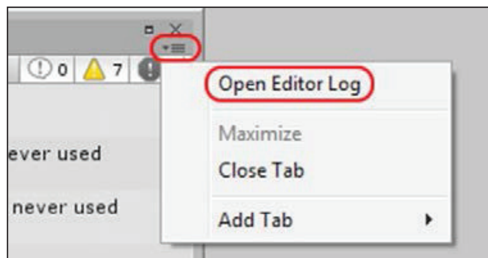


Рис. 8.1 ❖ Доступ к файлу регистрации

Щелкнув правой кнопкой мыши на заголовке любого окна и выбрав пункт **Add Tab** (Добавить вкладку), как показано на рис. 8.2, можно добавить в редактор дополнительные окна. Таким способом можно добавлять *дубликаты* окон, например одновременно открыть несколько представлений **Inspector** (Инспектор).

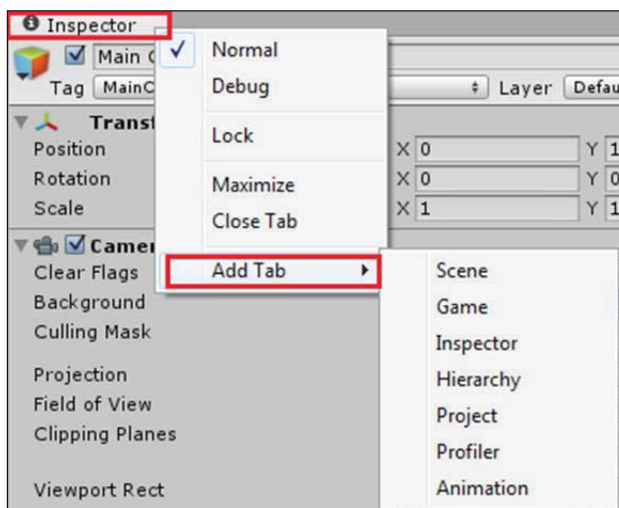


Рис. 8.2 ❖ Добавление дополнительных окон

Дублирование представлений не имеет смысла, если не использовать значок с изображением «замка» для блокировки текущего выделения в данном представлении (рис. 8.3). При выборе объекта все представления **Inspector** (Инспектор) обновятся для отображения данных об объекте, за исключением заблокированных, которые по-

прежнему будут отображать данные объекта, выбранного на момент блокировки.

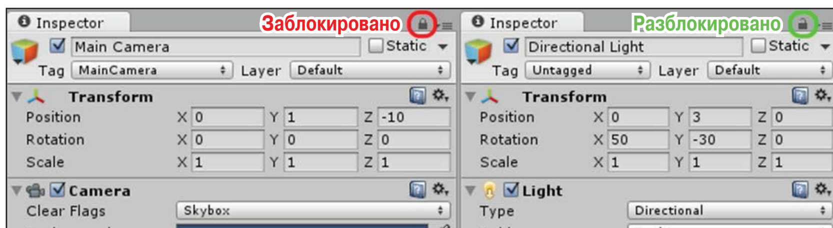


Рис. 8.3 ❖ Значок блокировки текущего выделения

Ниже дается несколько советов по использованию блокировок окон:

- используйте одинаковые представления (**Inspector** (Инспектор), **Animation** (Анимация) и др.) для сравнения данных двух объектов, поместив их рядом друг с другом, или для упрощения копирования данных;
- применяйте дублирование представлений **Project** для перемещения больших наборов данных;
- выполняйте тестирование зависимых объектов при настройке объекта во время выполнения;
- выберите несколько объектов в представлении **Project** (Проект), а затем перетащите их в сериализованный массив, в представлении **Inspector** (Инспектор), без потери начального выделения.

Представление инспектора

Числовые поля в представлении инспектора можно использовать для вычислений. Например, если в поле с типом `int` ввести выражение `4*128`, оно получит значение `512`. Таким способом можно избавить себя от выполнения расчетов на калькуляторе или в уме.

Копировать и удалять элементы массива (как это делалось с помощью клавиш быстрого доступа) можно щелчком правой кнопкой мыши и выбором пункта **Duplicate Array Element** (Скопировать элемент массива) или **Delete Array Element** (Удалить элемент массива).

Чтобы вывести контекстное меню компонента, щелкните правой кнопкой мыши на значке шестеренки в правом верхнем углу или на имени компонента. Контекстное меню всех компонентов содержит

пункт **Reset** (Сброс) для установки всех полей в значения по умолчанию, что избавляет от ручного ввода. Это особенно удобно при работе с компонентами Transform, так как при выборе этого пункта позиция и поворот объекта сбрасываются в (0,0,0), а масштаб в (1,1,1).

Всем известно, что если игровой объект порожден от шаблонного объекта, его можно вернуть в первоначальное состояние с помощью кнопки **Revert** (Вернуть) в представлении **Inspector** (Инспектор). Но мало кто знает, что отдельные значения можно восстановить, щелкнув правой кнопкой мыши на имени поля и выбрав пункт **Revert Value to Prefab** (Вернуть значение из шаблонного объекта). При этом восстановится только выбранное значение, остальные остаются нетронутыми.

Представление **Inspector** (Инспектор) имеет режим отладки, к которому можно получить доступ, щелкнув на значке бутерброда рядом со значком замка и выбрав пункт **Debug** (Отладка). При этом будут отключены все пользовательские отображения в представлении инспектора и выведены данные из выбранного игрового объекта и его компонентов, в том числе и из закрытых полей. Закрытые поля окрашиваются в серый цвет и недоступны для изменения в представлении **Inspector** (Инспектор), но их значения доступны для просмотра в режиме воспроизведения. Представление **Debug** (Отладка) также выводит внутренние идентификаторы объектов, которые могут пригодиться при взаимодействии с системой сериализации Unity и для разрешения возникающих конфликтов.

Если имеется массив, сериализуемый в представлении **Inspector**, его элементы обычно помечены как Element <N>, где <N> – индекс элемента в массиве. Это усложняет поиск конкретного элемента, если массив содержит сериализованные классы или структуры, которые могут иметь дочерние элементы. Но если *самое первое поле* объекта является строкой, названием элемента будет значение строкового поля, как показано на рис. 8.4.

Рис. 8.4 ❖ Элементы массива в представлении инспектора

При выборе объекта меша окно **Preview** (Предварительный просмотр) в нижней части представления **Inspector** (Инспектор) обыч-

но имеет небольшой размер, что затрудняет просмотр деталей меша и оценку его внешнего вида в сцене. Но если щелкнуть правой кнопкой мыши на верхней панели окна **Preview** (Предварительный просмотр), оно будет отсоединено и увеличено в размере, что упростит осмотр. Не нужно заботиться о возврате окон в их исходное положение, потому что после закрытия отсоединенного окна окно **Preview** (Предварительный просмотр) вернется на свое место в нижней части представления **Inspector** (Инспектор).

Представление проекта

Строка поиска в представлении **Project** (Проект) позволяет выполнить фильтрацию объектов по типу щелчком на маленьком значке справа от строки поиска. После щелчка появится список типов для фильтрации, выбор в котором приведен к выводу всех объектов этого типа в рамках целого проекта. В результате выбора любого пункта в строку поиска помещается строка вида `t:<type>`, что и вызовет применение соответствующего фильтра.

То есть можно просто вводить эквивалентные строки в строку поиска, что часто оказывается быстрее. Например, при вводе `t:prefab` будут выведены все шаблонные объекты независимо от их местоположения в иерархии; при вводе `t:texture` будут выведены все текстуры, при вводе `t:scene` – все сцены и т. д. Ввод нескольких фильтров приведет к выводу объектов всех типов (а не тех, что удовлетворяют обоим фильтрам). К фильтрам могут добавляться модификаторы, определяющие имена, то есть, добавив в фильтр обычный текст, можно отфильтровать объекты по типам и именам. К примеру, для фильтра `t:texture normalmap` будут выведены все файлы текстур, имена которых содержат слово `normalmap`.

Если используются `AssetBundles` (упаковки ресурсов) и встроенная система меток, с помощью строки поиска в представлении **Project** (Проект) можно отобрать такие упаковки по их метке, введя строку вида `l:<label type>`.

Если сценарий `MonoBehaviour` содержит сериализованные ссылки (объявленные с атрибутом `[SerializeField]` или `public`) на ресурсы, такие как меши и текстуры, им можно назначать значения по умолчанию непосредственно в сценарии. Выберите файл сценария в представлении **Project** (Проект), и в представлении **Inspector** (Инспектор) должно появиться поле для ресурса, доступное для перетаскивания, назначаемого по умолчанию ресурса, как показано на рис. 8.5.

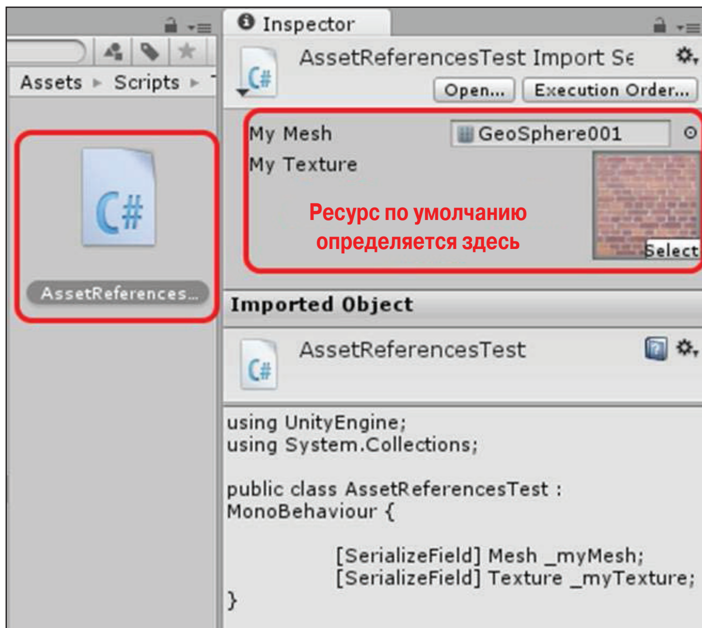


Рис. 8.5 ❖ Назначение ресурса по умолчанию

По умолчанию представление **Project** (Проект) разбивает файлы и папки на две колонки и обрабатывает их отдельно. Для отображения обычной иерархии папок и файлов нужно выбрать пункт **One Column Layout** (Макет в один столбец) в контекстном меню (значок бутерброда в правом верхнем углу). Такое отображение часто позволяет сэкономить место в макете редактора.

Щелкните правой кнопкой мыши на любом объекте в представлении **Project** (Проект) и выберите пункт **Select Dependencies** (Выбор зависимостей). Это приведет к выводу перечня всех объектов, от которых зависит этот ресурс, таких как текстуры, меши, файлы сценариев `MonoBehaviour` и т. д. Для файлов сцен такой перечень будет содержать ссылки на все сущности в сцене. Эта функция полезна при проведении чистки ненужных ресурсов.

Представление иерархии

Представление **Hierarchy** (Иерархия) имеет малоизвестную функцию, позволяющую фильтровать объекты в активной сцене по компо-

нентам. По непонятной причине, здесь используется тот же синтаксис, что при фильтрации по типам в представлении **Project** (Проект), и, следовательно, требуется ввести строку вида `t:<component_name>`. Например, если ввести `t:light` в строку поиска, представление **Hierarchy** (Иерархия) выведет перечень всех объектов в сцене, к которым присоединен компонент освещения.

Регистр символов не имеет значения, но в строке поиска должно указываться полное имя компонента. Кроме того, в перечень включаются компоненты, наследующие заданный тип, то есть для строки поиска `t:renderer` представление выведет все объекты с компонентами, наследующими `renderer`, такими как MeshRenderer, SkinnedMeshRenderer и т. д.

Представления сцены и игры

Камера представления сцены не видна из представления игры, но она значительно облегчает перемещение и позиционирование с помощью клавиш быстрого доступа, описанных выше. Редактор позволяет задать позицию выбранного объекта и поворот камеры сцены выбором пункта **GameObject ⇨ Align with View** (Объект игры ⇨ Выравнивание по обзору) (**Ctrl**+**Shift**+**F** / **Cmd**+**Shift**+**F**). То есть, воспользовавшись элементами управления камеры, можно поместить камеру сцены туда, где должен находиться объект, и совместить его с камерой.

Аналогично можно выровнять обзор сцены по выбранному объекту, выбрав пункт **GameObject ⇨ Align View to Selected** (Объект игры ⇨ Выровнять обзор по выбранному объекту). Таким способом можно убедиться, что данный объект повернут в нужном направлении.

В представлении сцены можно выполнять фильтрацию по компонентам по аналогии с представлением **Hierarchy** (Иерархия), введя в строку поиска фильтр вида `t:<component>`.

Справа в самом верху редактора Unity имеется раскрывающееся меню **Layers** (Слои), как показано на рис. 8.6. Оно содержит набор фильтров слоев и блокировки для представления сцены. Включение/выключение значка с изображением глаза отображает/скрывает все объекты этого слоя в представлении сцены. Переключение значка с изображением замка разрешает или запрещает выбор объектов заданного слоя. Это полезно для предотвращения случайного выбора и перемещения объектов фона, которые на данный момент идеально позиционированы.

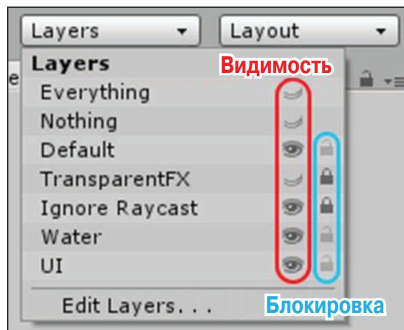


Рис. 8.6 ❖ Раскрывающееся меню **Layers** (Слои)

Широко известной особенностью редактора является возможность добавления к игровым объектам специальных значков или меток для упрощения их поиска в представлении сцены. Эта особенность значительно упрощает поиск неотображаемых объектов, таких как источники света и камеры, которые имеют встроенные значки, позволяющие легко выделить их в представлении сцены.

Для отображения объектов Gizmo в представлении **Game** (Игра) щелкните на кнопке **Gizmos** в верхнем правом углу представления **Game** (Игра). В раскрывающемся списке этой кнопки можно определить, какие именно объекты Gizmo станут видимыми при ее включении.

Режим воспроизведения

Поскольку изменения в режиме воспроизведения не сохраняются автоматически, имеет смысл изменить оттенок цвета, применяемого в режиме воспроизведения, чтобы подчеркнуть особенность этого режима. Сделать это можно, изменив параметр **Edit** ⇨ **Preferences** ⇨ **Colors** ⇨ **Playmode tint** (Правка ⇨ Предпочтения ⇨ Цвета ⇨ Оттенок режима воспроизведения).

Сохранить изменения, сделанные в режиме воспроизведения, можно с помощью буфера обмена. В случае успешной настройки параметров объекта в режиме воспроизведения объект можно скопировать в буфер обмена комбинацией клавиш **Ctrl**+**C** (**Cmd**+**C**), а затем, после выхода из режима воспроизведения, вставить его обратно в сцену комбинацией **Ctrl**+**V** (**Cmd**+**V**). То же можно проделать с отдельными значениями или целыми компонентами, воспользовавшись пунк-

тами **Copy Component** (Копировать компонент) и **Paste Component** (Вставить компонент) контекстного меню компонента. Однако буфер обмена может одновременно содержать данные только одного объекта, компонента *или* значения.

Чтобы в режиме воспроизведения сохранить несколько объектов, можно создать шаблонный объект, перетаскивая нужные объекты в окно **Project** во время выполнения после удачной их настройки. Если исходный объект создан из шаблонного объекта и необходимо обновить все его экземпляры, следует затереть старый шаблонный объект вновь созданным, перетащив его на старый. Обратите внимание, что этот прием работает также в режиме выполнения, что может быть опасно, так как диалог для подтверждения перезаписи в этом случае не выводится. Будьте осторожны, чтобы случайно не перезаписать нужный шаблонный объект.

Для покадрового просмотра можно пользоваться кнопкой **Frame Skip** (Вперед на кадр, справа от кнопки паузы редактора). Это может пригодиться для отладки физических взаимодействий или логики игры. Имейте в виду, что в этом случае на каждом шаге выполняется еще один вызов функции `FixedUpdate` и функции `Update`, что не соответствует обычному процессу выполнения, когда, как правило, эти функции вызываются с разной частотой.

Если при переходе в режим воспроизведения нажата кнопка паузы, игра будет приостановлена сразу после первого кадра, давая возможность наблюдать аномалии, возникшие при инициализации сцены.

Советы для сценариев

Следующие советы пригодятся при написании сценариев.

Общие

Имеется возможность изменить шаблоны новых файлов сценариев, шейдеров и вычислительных шейдеров, например чтобы избавиться от пустой функции `Update`, наличие которой, как уже упоминалось ранее, влечет дополнительные накладные расходы во время выполнения. Файлы шаблонов можно найти:

- для Windows: `<Каталог_Unity>\Editor\Data\Resources\ScriptTemplates\`;
- для OSX: `/Applications/Unity/Editor/Data/Resources/ScriptTemplates/`.

Недавно вышедшая версия Unity 5.1 поддерживает класс `Assert` и отладку с применением проверок, которую многие разработчики считают более удобной, чем отладку, основанную на исключениях. Более подробную информацию об этом можно найти в документации: http://docs.unity3d.com/ScriptReference/Assertions.Assert.html.

Вызов метода `Debug.Break()` функционально эквивалентен паузе в режиме воспроизведения, что можно использовать для выявления нестыковок в графике или в качестве удобной альтернативы абсурдной комбинации клавиш для приостановки сцены (**Ctrl**+**Shift**+**P**).

Атрибуты

Атрибуты – весьма полезные теги мета-уровня, которые в языке C# можно использовать практически для любых целей. Чаще всего атрибуты добавляют к членам данных (полям) и классам для наделения их особыми свойствами и обработки по-разному. Unity-разработчикам среднего и продвинутого уровня стоит ознакомиться с их применением в документации языка C# и рассмотреть использование собственных атрибутов, которые помогут ускорить разработку. Существует несколько атрибутов, встроенных в движок Unity, которые оказываются весьма полезными при правильном использовании.

Опытные пользователи знают, что атрибуты часто добавляются к перечислениям, делегатам, методам, параметрам, событиям, модулям и даже сборкам.

Атрибуты переменных

Общедоступные переменные являются довольно опасной вещью при добавлении в компоненты, поскольку их значения могут изменяться откуда угодно во время выполнения, что затрудняет трассировку ошибок. Кроме того, для предотвращения присваивания недопустимых значений требуется специальный механизм. Однако общедоступные переменные являются основным способом отображения значений в представлении **Inspector** (Инспектор) для Unity-разработчиков. В некоторых случаях переменным абсолютно необходимо быть общедоступными, но они не должны отображаться в представлении **Inspector** (Инспектор). Для скрытия таких переменных в представлении **Inspector** (Инспектор) можно использовать атрибут `[HideInInspector]`.

С другой стороны, вывод закрытых и защищенных переменных в редакторе можно осуществить с помощью атрибута `[SerializeField]`.

Этот атрибут позволяет вынести такие переменные в представление **Inspector** (Инспектор) без риска, что другие компоненты могут случайно изменить их во время выполнения.

К полям с целочисленными значениями и значениями с плавающей точкой можно добавить атрибут `[Range]` для отображения их в виде ползунка в представлении **Inspector** (Инспектор). В атрибуте можно определить минимальную и максимальную величины значения.

Обычно при переименовании переменной в интегрированной среде разработки (MonoDevelop или Visual Studio) ее значение теряется, как только Unity повторно скомпилирует сценарий и внесет соответствующие изменения во все экземпляры компонента. Однако если снабдить переменную атрибутом `[FormerlySerializedAs]`, при переименовании ее прежнее сериализованное значение будет скопировано в момент компиляции. Теперь данные не будут теряться при переименовании!

Обратите внимание, что удаление атрибута `[FormerlySerializedAs]` после завершения преобразования *небезопасно*, если только переменная не была изменена вручную и повторно сохранена. Файл данных «.prefab» по-прежнему содержит старое имя переменной, и поэтому атрибут `[FormerlySerializedField]` все еще необходим Unity, чтобы знать, куда поместить данные при следующей загрузке файла (например, когда редактор будет закрыт и вновь открыт). Это очень полезный атрибут, но неумеренное его использование излишне загромождает код.

Атрибуты классов

Атрибут `[SelectionBase]` отмечает все игровые объекты, к которым выбранный в представлении сцены компонент присоединен как корневой. Это особенно полезно в случаях, когда имеются меши, являющиеся дочерними по отношению к другим объектам, и требуется, чтобы при первом щелчке выбирался родительский объект, а не объект с компонентом `MeshRenderer`.

Атрибут `[RequireComponent]` можно использовать, чтобы обязать разработчиков присоединять к игровому объекту прочие необходимые компоненты, когда они пытаются присоединить данный компонент. Это гарантирует, что все нужные зависимости будут удовлетворены, и поможет избежать написания обширной документации.

Атрибут `[ExecuteInEditMode]` принуждает вызывать методы `Update()`, `OnGUI()` и `OnRenderObject()` объекта даже в режиме редактирования. Однако при этом следует учитывать, что:

- метод Update() вызывается, только если что-то изменится в сцене;
- метод OnGUI() вызывается, только если событие возникнет в представлении игры, но не в других представлениях, например в представлении сцены;
- метод OnRenderObject() вызывается по любым событиям отображения сцены и представления игры.

Этот атрибут используется, чтобы придать объекту другой набор обработчиков событий и точек входа, не как в обычном редакторе сценариев.

Регистрация

В отладочные сообщения можно включать теги форматирования, такие как <size> (размер), <b> (полужирный), <i> (наклонный) и <color> (цвет). Это поможет отличать разные виды сообщений и выделять конкретные элементы, как показано на рис. 8.7.

```
Debug.Log ("<color=red>[ERROR]</color>This is a <i>very</i>
<size=14><b>specific</b></size> kind of log message");
```

Рис. 8.7 ❖ Результат применения тегов форматирования

Класс MonoBehaviour имеет удобный метод print(), эквивалентный методу Debug.Log().

Он может помочь в создании пользовательского класса регистрации, автоматически добавляющего \n\n в конец каждого сообщения, и избавиться от вызова метода UnityEngine.Debug:Log(Object), вносящего беспорядок в окно консоли.

Полезные ссылки

Unity technologies предоставляет массу полезных руководств по использованию различных возможностей, которые главным образом ориентированы на разработчиков среднего уровня и начинающих. Эти руководства можно найти по адресу: https://unity3d.com/learn/tutorials/topics/scripting.

На Unity Answer можно найти статью со ссылками на описание многих ошибок компиляции: http://answers.unity3d.com/questions/723845/what-are-the-c-error-messages.html.

Объекты класса `ScriptableObject` представляют отличный способ хранения данных и не требуют их создания во время выполнения. Они подобны объектам любого другого класса, в том смысле, что могут иметь методы и переменные, поддерживают сериализацию и полиморфизм и т. д. Но могут создаваться только с помощью сценариев и должны загружаться в память вызовом метода `Resources.Load()`. Это позволяет контролировать объекты класса `ScriptableObject`, присутствующие в памяти, и облегчает контроль над потреблением памяти во время выполнения. Описание нюансов использования объектов `ScriptableObject` займет слишком много места, но вы можете просмотреть отличный начальный видеокурс, доступный по адресу: https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/scriptable-objects.

Обратите внимание, что видеокурс охватывает и другие близкие темы. Новичкам лучше сначала сосредоточить свое внимание на работе с шаблонными объектами и только потом приступать к изучению других важных тем, касающихся класса `ScriptableObject` и сериализации.

Вложенные сопрограммы – еще одна интересная и полезная область сценариев, которая не очень хорошо документирована. Однако существует сторонний блог с описанием множества интересных подробностей, которые следует учитывать при работе со вложенными сопрограммами: http://www.zingweb.com/blog/2013/02/05/unity-coroutine-wrapper.

Выяснить, когда функция была добавлена в прикладной программный интерфейс Unity, можно на странице, посвященной истории программного интерфейса: http://docs.unity3d.com/ScriptReference/40_history.html.

Советы по настройке редактора и меню

Имеется возможность назначить клавиши быстрого доступа для пунктов меню. Например, обеспечить быстрый доступ к некоторому пункту меню с помощью клавиши *K* можно, определив атрибут `MenuItem`:

```
[MenuItem("My Menu/Menu Item _k")]
```

Также можно указать модификатор клавиш **Ctrl (Cmd)**, **Shift** и **Alt**, использовав символ **%**, **#** и **&** соответственно.

Атрибут MenuItem поддерживает еще два дополнительных параметра: логическое значение, определяющее необходимость вызова метода валидации, и целое число, задающее приоритет пункта меню в иерархии.

Полный список доступных модификаторов клавиш, специальных клавиш и описание создания методов валидации пунктов меню можно найти по адресу: http://docs.unity3d.com/ScriptReference/MenuItem.html.

Имеется возможность послать сигнал проверки (ping) объекту в иерархии, подобно тому, как это делается при щелчке на ссылке в представлении **Inspector** (Инспектор), для чего следует вызвать метод EditorGUIUtility.PingObject().

Первоначальная реализация класса Editor приучила многих разработчиков размещать логику сценариев и методы отображения в одном и том же классе. Однако с помощью класса PropertyDrawer можно делегировать отображение инспектора разным классам. Он позволяет отделить ввод и валидацию от отображения и получить более полный контроль над отображением каждого из полей и эффективнее использовать код. Класс PropertyDrawer можно даже использовать для переопределения отображения по умолчанию встроенных объектов Unity, таких как векторы и кватернионы.

Класс PropertyDrawer использует SerializedPropertie для сериализации отдельных полей, и ему следует отдавать предпочтение при написании сценариев редактора, поскольку он способен использовать встроенные функции отмены, повтора и мультиредактирования. Проверка данных может быть несколько усложнена, и лучшим решением является использование метода OnValidate() при установке свойств для полей. Выступление разработчика из команды Unity Technologies, Тима Купера (Tim Cooper), на конференции Unite 2013, в котором он подробно объясняет преимущества и недостатки различных подходов к сериализации и валидации, можно найти по адресу: https://www.youtube.com/watch?v=Ozc_hXzp_KU.

С помощью атрибутов [ContextMenu] и [ContextMenuItem] можно добавить пункты в контекстное меню компонентов и даже в контекстное меню отдельных полей. Это упрощает настройку работы инспектора с компонентами и не требует написания классов редактора или пользовательских инспекторов.

Продвинутые пользователи могут счесть полезным хранение пользовательских данных в файлах метаданных Unity посредством переменной AssetImporter.userData. Также имеется множество воз-

можностей для использования механизма отражения в сценариях Unity. Описание огромного количество приемов применения отражения в редакторе Unity можно найти в выступлении Райана Хиппла (Ryan Hipple) на Unite 2014: https://www.youtube.com/watch?v=SyR4OYZpVqQ.

В Unity 4.5 появилась недокументированная возможность – **переупорядочиваемые списки**. Они позволяют иметь в представлении инспектора обычный список `List<T>`, который легко переуроpядочить перетаскиванием элементов. Однако эта функция выглядит незавершенной, так как для полноценного использования требует наличия пользовательского класса редактора. Следующая статья на Unity Answers достаточно лаконично разъясняет использование переупорядочиваемых списков: http://answers.unity3d.com/questions/826062/re-orderable-object-lists-in-inspector.html.

Советы, не касающиеся Unity напрямую

Следующие советы и рекомендации касаются вопросов, не относящихся к самому редактору Unity, но они могут пригодиться при разработке в Unity.

Поиск в Google по темам, связанным Unity, будет эффективнее, если начать строку поиска с текста «`site:unity3d.com`».

В случае краха редактора Unity, независимо от причин, можно восстановить сцену, переименовав файл с расширением `.unity` (расширение файлов сцен) и скопировав его в папку `Assets`:

```
\<project folder>\Temp\_EditModeScene
```

При разработке в Windows предпочтительнее пользоваться средой Visual Studio. Среда MonoDevelop спотыкается и поскрипывает уже на протяжении многих лет, и многие разработчики переключаются на редактор Visual Studio Community, который обладает большими возможностями, перекрывающими практически все потребности, особенно вместе с невероятно полезными подключаемыми модулями, такими как Resharper.

Для некоторых разработчиков единственной причиной продолжать пользоваться MonoDevelop была возможность отладки во время выполнения кода, но с выходом пакета Visual Studio Tools for Unity (VSTU) среда разработки Visual Studio приобрела более тесную интеграцию с редактором Unity. Появилась даже возможность отладки кода на C# во время выполнения в самой среде Visual Studio. Если вас

смущает незнакомый интерфейс или тот факт, что изготовителем является большая и нехорошая компания Microsoft, советую все равно попробовать поработать с Visual Studio Community, чтобы познакомиться со способами ускорения разработки кода сценариев, о которых вы даже не подозревали.

Более подробную информацию о Visual Studio и ее интеграции с Unity посредством VSTU можно найти в видеообзоре: https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2015-Final-Release-Event/Building-Unity-games-in-Visual-Studio.

В Интернете имеется отличный ресурс, содержащий шаблоны программирования игр (точнее, типовые шаблоны программирования, имеющие отношение к разработке игр), которые совершенно бесплатны и доступны для скачивания: http://gameprogrammingpatterns.com/contents.html.

Не забывайте просматривать видеоролики с выступлениями всех конференций Unite (а еще лучше попробуйте принять в них участие). На каждой конференции имеется пара секций, в рамках которых опытные разработчики обмениваются полезными и интересными сведениями, касающимися использования движка и редактора. Кроме того, поддерживайте связь с Unity-сообществом через форумы на unity3d.com, Twitter, reddit, Stack Overflow, Unity Answers и многочисленные конференции, участившиеся в последние годы.

Все советы, включенные в эту книгу, основаны на идеях и знаниях, которые где-то были выложены на всеобщее обозрение. Поэтому, чтобы быть в курсе самых актуальных подсказок, приемов и методов, следите за развитием Unity, принимайте активное участие в деятельности сообщества.

Другие советы

И наконец, в этом разделе содержатся советы, которые не вошли ни в одну из категорий.

Старайтесь структурировать сцены с помощью пустых игровых объектов с понятными названиями. Единственным недостатком этого метода является учет преобразований пустого объекта при изменении позиции или поворота и при пересчете. Но ссылки на нужный объект, кэширование изменений преобразования и/или использование `localPosition`/`localRotation` позволят решить эту проблему. Почти всегда преимущества структурирования сцены более ценны, чем небольшие потери производительности.

Возможность переопределения контроллеров анимации, появившаяся еще в Unity 4.3, была быстро забыта и практически нигде не упоминается. Она может послужить альтернативой стандартным контроллерам анимации, так как позволяет сослаться на существующий контроллер анимации, а затем переопределить конкретные состояния для использования различных анимационных файлов. Это значительно ускоряет разработку, поскольку делает ненужным дублирование и настройку контроллеров анимации, так как требует лишь изменения нескольких состояний анимации.

При запуске Unity 5 автоматически открывается мастер проекта **Project Wizard**, позволяющий открыть последний проект. Однако если вы предпочитаете порядок запуска, принятый по умолчанию в Unity 4, где автоматически открывался предыдущий проект, измените порядок запуска, установив флаг **Edit ⇨ Preferences ⇨ General ⇨ Load Previous Project on startup** (Правка ⇨ Предпочтения ⇨ Общие ⇨ Загрузка предыдущего проекта при запуске). Обратите внимание, что этот флаг в Unity 4 имеет другое название и выполняет обратное действие. Его можно найти, перейдя к **Edit ⇨ Preferences ⇨ General ⇨ Always Show Project Wizard** (Правка ⇨ Предпочтения ⇨ Общие ⇨ Отображать мастер проекта). Обратите внимание, что если мастер **Project Wizard** включен, можно открыть несколько экземпляров редактора Unity.

Удивительная гибкость редактора Unity и его постоянно растущие возможности способствуют добавлению новых способов облегчения процесса разработки к массе уже известных. На сайте Asset Store предлагается огромное количество ресурсов для решения конкретных проблем, возникающих в процессе разработки, что делает его лучшим местом для поиска решений, которые избавят от массы хлопот за очень скромную плату. Поскольку эти ресурсы нацелены на самую широкую аудиторию, цена их обычно невелика, и это позволяет приобрести отличные полезные инструменты и сценарии по удивительно низкой цене. На самостоятельную разработку подобных решений почти всегда требуется затратить значительное количество часов. Если вы цените свое время, Asset Store поможет вам сэкономить его.

Итоги

Книга подошла к концу, и я надеюсь, что вы получили удовольствие от ее прочтения. Самый главный совет этой книги: при появлении проблем с производительностью проведите профилирование, пре-

жде чем вносить какие-либо изменения. Не стоит тратить времени на погоню за призраками, когда пять минут тестирования с помощью профилировщика могут заменить целый день работы вслепую. Кроме того, для принятия решения часто требуется убедиться, что улучшение производительности в одном месте не приведет к значительным потерям в другом. Удостоверьтесь, что правильно определили причины, чтобы избежать ненужного риска.

Задача увеличения производительности может оказаться весьма интересной. Из-за высокой сложности современного компьютерного оборудования небольшие подстройки порой приводят к значительным результатам. Существует множество способов увеличения производительности приложения или ускорения процесса разработки. Некоторые из них трудно реализовать в полной мере, не имея соответствующего опыта и навыков, необходимых для их осуществления за разумное время. Но в большинстве случаев процесс внесения исправлений относительно прост, если уже найден источник проблемы. Поэтому двигайтесь вперед и используйте полученные знания, чтобы сделать ваши игры лучше!

Предметный указатель

Символы

2D-коллайдеры
- короб, 154
- круг, 154
- многоугольник, 154

3D-коллайдеры
- капсула, 154
- короб, 154
- меш, 154
- сфера, 154
- цилиндр, 154

.NET
- URL-адрес, 275
- функции библиотеки, 253

А

Артефакты, 133
Атласинг, 135
Атрибуты
- классов, 292
- переменных, 291

Аудио
- определение, 118
- профилирование, 119
- улучшение производительности, 123
- уровни качества, раскодирование, 121
- файлы, загрузка, 118
- форматы кодирования, 121

Аудиофайлы
- дополнительные варианты загрузки, 120
- загрузка, 120

Б

Блок общего распределения, 234
Буфер команд, URL-адрес, 187

В

Вложенные сопрограммы, URL-адрес, 294
Внутреннее устройство физического движка
- активное состояние компонента RigidBody, 156
- виды коллайдеров, 154
- время, 148
- динамические коллайдеры, 152
- максимально допустимая длительность, 150
- матрица столкновений, 155
- обнаружение столкновений, 153
- описание, 148
- отбрсывание лучей, 157
- состояние сна компонента RigidBody, 156
- статические коллайдеры, 152
- физические обновления, 151
- цикл фиксированного обновления, 150

Временные рабочие буфера, 254

Д

Динамическая пакетная обработка
- атрибуты вершин, 108
- единообразное масштабирование, 109
- краткие выводы, 110
- применение, 107
- требования, 107

Динамические коллайдеры, 152

З

Задерживание центральным процессором
- аппаратный скининг, 188

– многопоточный вывод, 186
– описание, 185
Замыкания, 252
Значение массы, URL-адрес, 159
Значения и ссылки, структуры, 239

И

Изменение частоты фиксированных обновлений, 163

К

Клавиши быстрого доступа редактора
– доступ к документации, 281
– игровые объекты, 279
– интерфейс, 280
– массивы, 280
– описание, 279
– представление сцены, 279
Количество полигонов
– использование флага доступности чтения и записи, 142
– настройка сжатия мешей, 142
– уменьшение, 141
Куча, 230

Л

Листы спрайтов, 128

М

Максимально допустимая длительность
– настройка, 165
– описание, 150
Матрица столкновений, 155
Метод FixedUpdate(), 163
Метод RuntimeHelpers. PrepareMethod(), 226
Метод UnloadAudioData(), 121
Методы улучшения производительности
– встаиваемые анимации, 143
– импорт только необходимого, 143
– настройка сжатия мешей, 142
– объединение мешей, 145
– оптимизация мешей, 144
– расчет только необходимого, 143
– уменьшение количества полигонов, 141
Мешевые коллайдеры
– вогнутые, 154
– выпуклые, 154
Меши, описание, 141

Н

Настройка сцены
– массы, 159
– масштабирование, 158
– позиционирование, 159
Неизменяемые ссылки
– объединение строк, 245
– описание, 243
Непосредственное отображение, URL-адрес, 213

О

Обволакивание, 188
Области памяти Unity
– описание, 227
– собственная память, 228
– управляемая память, 229
Общий промежуточный язык (CIL), 222
Объекты класса ScriptableObject, 294
Овердрафт, 194
Ограничения видеопамяти
– описание, 210
– перезагрузка текстур, 211
– предварительная загрузка текстур, 210
Операционная система (ОС), 223
Оптимизация графики для мобильных устройств
– избегание альфа-тестирования, 219
– использование в шейдерах форматов с минимальной точностью, 219

– квадратные текстуры, 219
– минимизация количества материалов, 218
– минимизация обращений к системе визуализации, 218
– описание, 217
– текстуры со стороной, кратной степени 2, 219
– уменьшение размеров текстур, 218

Оптимизация использования памяти, 226
– области памяти Unity, 227
– функции библиотеки .NET, 253

Оптимизация освещения
– использование внедренных карт освещения, 216
– использование масок выбраковки, 215
– использование соответствующего режима заливки, 215
– описание, 215
– оптимизация теней, 216

Оптимизация памяти
– важность порядка размещения данных, 249
– временные рабочие буфера, 254
– замыкания, 252
– организация пула объектов, 254
– прикладной программный интерфейс Unity, 250
– сопрограммы, 252
– циклы foreach, 251

Оптимизация производительности физической системы
– Unity 5, переход, 176
– дискретное обнаружение столкновений, предпочтение, 162
– изменение количества итераций, 172
– максимально допустимая длительность, настройка, 165
– матрица столкновений, оптимизация, 161
– описание, 157
– отбрасывание лучей и ограничение проверяемого объема, 165
– сложный мешевый коллайдер, избегание, 167
– статические коллайдеры, использвание, 160
– тряпичные куклы, оптимизация, 173
– физическая система, использование, 170, 175
– физические объекты, пусть поспят, 170
– частота фиксированных обновлений, изменение, 163

Организация пулов предварительно подготовленных объектов
– компоненты пула, 260
– описание, 257
– тестирование, 271

Организация пулов шаблонных объектов
– активация объектов, 268
– деактивация объектов, 270
– загрузка сцены, 272
– предварительное создание экземпляров, 269
– резюме, 273

Освещение и затенение
– URL-адрес, 212
– непосредственное отображение, 213
– обработка теней в реальном времени, 215
– описание, 212
– освещение в вершинном шейдере, 214
– отложенное затенение, 214

Отладка кадров, 184

Отсроченное затенение, URL-адрес, 214

П

Пакетная обработка
- краткие выводы, 110
- требования к памяти, 112
- флаг Static, 111

Передача по значению, 238
Передача по ссылке, 238
Перераспределяемые списки
- URL-адрес, 296
- описание, 296

Платформа Mono
- URL-адрес, 274
- дальнейшее развитие, 274
- описание, 222
- отключение или удаление тряпичных кукол, 174

Поиск методом перебора, 185
Прикладной программный интерфейс Unity, 250
Пропускная способность памяти
- использование меньших текстур, 208
- описание, 206
- организация ресурсов для уменьшения перекачки текстур, 210
- сведение к минимуму выборки текстур, 209
- тестирование различных форматов сжатия текстур, 209

Профилирование графического процессора, 181
Профилирование проблем отображения
- задерживание центральным процессором, 185
- описание, 179
- отладка кадров, 184
- поиск методом перебора, 185
- профилироване графического процессора, 181

Процедурные материалы
- URL-адрес, 141
- описание, 140

Процесс компиляции, платформа Mono
- описание, 224
- ручная JIT-компиляция, 225

Р

Разреженные текстуры
- URL-адрес, 140
- описание, 139

Редактор
- интерфейс, 280
- клавиши быстрого доступа, 279
- описание, 279

Редактор Unite, отражение, URL-адрес, 296

С

Сборка мусора, управление памятью
- во время выполнения, 233
- описание, 230
- потоки, 234
- тактика, 234
- фрагментация, 231

Сборщик мусора
- завершающий поток, 234
- основной поток, 234

Система пулов шаблонных объектов
- требования к системе, 263

Скорость заполнения
- овердрафт, 194
- окклюзивная выбраковка, 195
- описание, 192
- оптимизация шейдеров, 197
- шейдеры для мобильных платформ, 198

Сложный мешевый коллайдер
- избегание, 167
- оптимизация тряпичных кукол, 173
- простые мешевые коллайдеры, использование, 169
- простые примитивы, использование, 168
- столкновение между собой тряпичных кукол, уменьшение, 174

– число шарниров и коллайдеров, уменьшение, 173
Собственная область памяти, 227
Советы для сценариев
– атрибуты, 291
– атрибуты классов, 292
– общие, 290
– описание, 290
– регистрация, 293
Советы, касающиеся интерфейса редактора
– описание, 281
– представление игры, 288
– представление иерархии, 287
– представление инспектора, 284
– представление проекта, 286
– представление сцены, 288
– режим воспроизведения, 289
Советы, не касающиеся напрямую Unity
– другие советы, 297
– описание, 296
Сопрограммы, описание, 252
Спрайт, 128
Статическая пакетная обработка, оговорки
– видимость, 115
– избегание создания статических мешей, 114
– описание, 113
– отладка в режиме редактирования, 114
– отображение, 115
Статические коллайдеры
– использование, 155
– определение, 152
Стек, 229
Сферические гармоники, 213

Т

Текстура, 128
Типы ссылок
– массивы, 242
– описание, 238

У

Узкие места заключительной части
– видеопамять, 210
– пропускная способность памяти, 206
– скорость заполнения, 192
Узкие места начальной части
– описание, 189
– отключение аппаратного скининга, 191
– уменьшение тесселяции, 192
– уровень детализации, 189
Узкие места на этапе окончательной обработки, описание, 192
Улучшение производительности аудио
– описание, 123
– осторожность при применении потоковой передачи, 127
– понижение частот, ресамплинг, 126
– сведение к минимуму ссылок на аудиоклипы, 124
– сведение к минимуму числа активных источников, 123
– свойство WWW.audioClip, использование, 128
– установка флага Force to Mono для 3D-звуков, 126
– эффект от применения фильтра, 127
Улучшение производительности обработки тексту, описание, 131
Улучшение производительности обработки текстур
– mip-текстуры, использование, 132
– атласинг, обсуждение, 135
– настройка сжатия текстур неквадратной формы, 138
– процедурные материалы, 140
– разреженные текстуры, 139
– уменьшение размеров файлов текстур, 131

– управление разрешением текстур извне, 133
– уровень анизотропной фильтрации, настройка, 134
Упаковка, 247
Управляемая куча, 229
Управляемая область памяти, 228
Уровень детализации (LOD), описание, 179, 189

Ф

Файлы анимаций, 141
Файлы текстур
– описание, 128
– улучшение производительности, 131
– форматы сжатия, 129
Физический движок
– использование, 147
– описание, 147
Функции стандартной библиотеки Cg, URL-адрес, 201
Функция Shadow Cascades, URL-адрес, 217

Ц, Ч

Циклы foreach, 251
Черный ящик, 148

Ш

Шаблоны программирования игр, 297
Шейдеры для мобильных платформ
– избегайте условных операторов, 203
– избегание изменения точности при swizzling-присвоении, 199
– использвание оптимизированных вспомогательных функций, 200
– использование, 198
– использование типов данных малого размера, 199
– отключение ненужных функций, 201
– снижение математической сложности, 202
– сократите поиск текстур, 203
– удаление ненужных входных данных, 201
– уменьшение зависимости данных, 204
– уровни детализации на основе шейдеров, 206
– шейдеры поверхностей, 205
– экспонируйте только необходимые переменные, 201
Шейдеры фрагментов, 180

A, C

Ahead-Of-Time (AOT), 225
Common Language Runtime (CLR), 223

I

Intermediate Language To C++ (IL2CPP)
– URL-адрес, 275
– описание, 275

J, M

Just-In-Time (JIT), 225
Mono
– описание, 222
– процесс компиляции, 222

U

Unity
– URL-адрес, 275
– дальнейшее развитие, 274
– документация, 137

V

Visual Studio, URL-адрес, 297
Visual Studio Tools for Unity (VSTU), 296

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Крис Дикинсон

Оптимизация игр в Unity 5

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Киселев А. Н.*
Перевод *Рагимов Р. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 28,6875. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф