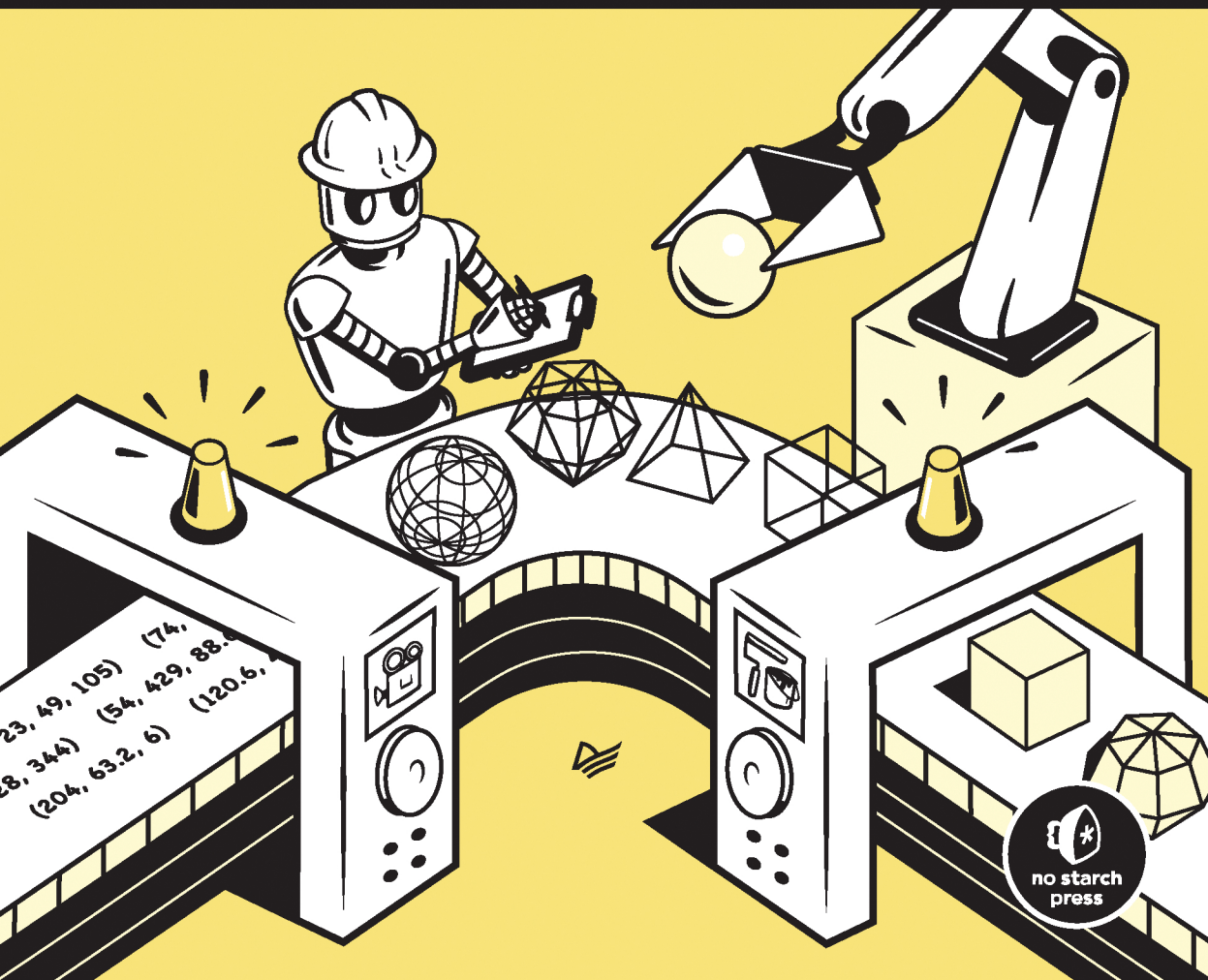
КОМПЬЮТЕРНАЯ ГРАФИКА

РЕЙТРЕЙСИНГ И РАСТЕРИЗАЦИЯ

ГЭБРИЕЛ ГАМБЕТТА



COMPUTER GRAPHICS FROM SCRATCH

A Programmer's Introduction to 3D Rendering

by Gabriel Gambetta

no starch press

San Francisco

КОМПЬЮТЕРНАЯ ГРАФИКА

РЕЙТРЕЙСИНГ И РАСТЕРИЗАЦИЯ

ГЭБРИЕЛ ГАМБЕТТА

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-044.4
УДК 004.92
Г18

Гамбетта Гэбриел

Г18 Компьютерная графика. Рейтрейсинг и растеризация. — СПб.: Питер, 2022. — 224 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1911-0

За красивыми образами анимационного фильма и реалистичной средой популярных видеоигр скрываются загадочные алгоритмы.

В этой книге вы познакомитесь с двумя основными направлениями современной графики: рейтрейсингом и растеризацией. Такая литература пугает новичков из-за большого количества математики. Но только не в этом случае. Познакомьтесь с 3D-рендерингом без длинных формул!

Вы создадите полноценные рабочие рендеры — рейтрейсинг, симулирующий лучи света и их отражение от объектов, растеризатор 3D-моделей, научитесь создавать реалистичные отражения и тени, а также отрисовывать сцены с любой точки обзора.

Наглядные примеры с псевдокодом позволят без проблем создавать рендеры на любом языке, а живые JavaScript-демки каждого алгоритма вдохновят на самостоятельные подвиги.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-044.4
УДК 004.92

ISBN 978-1718500761 англ.

ISBN 978-5-4461-1911-0

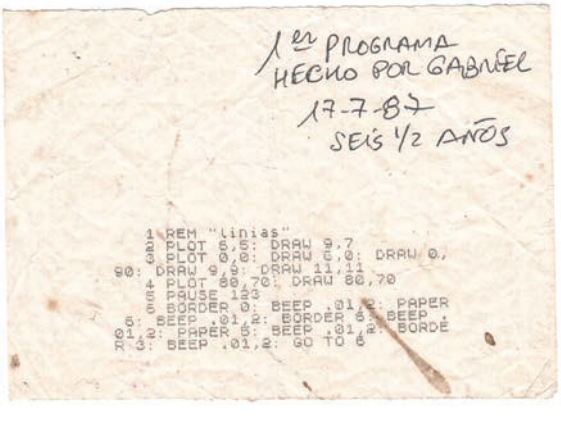
Краткое содержание

Оглавление

ЧАСТЬ I. ТРАССИРОВКА ЛУЧЕЙ

Посвящается моему отцу (1947–2007), архитектору и программисту-самоучке, направившему меня на этот путь.

Мой отец, двухлетний я и ZX81



Моя первая задокументированная программа, рисовавшая линии на экране ZX-Spectrum+, которую я написал в шесть с половиной лет

Об авторе

Габриэл Гамбетта (Gabriel Gambetta) в пять лет начал писать игры на ZX Spectrum. Изучив информатику и поработав в одной крупной организации на родине в Уругвае, он организовал свою компанию по разработке игр, которой руководил десять лет, параллельно преподавая компьютерную графику в университете. С 2011 года Габриэл работает в Google Zürich (за исключением периода, когда он был инженером лондонской компании Improbable, специализирующейся на многопользовательских играх, и одного года в Мадриде, где он осваивал актерское мастерство и киносъемку).

О техническом редакторе

Алехандро Сеговия Азапиан (Alejandro Segovia Azapian) — инженер ПО, уже 14 лет занимающийся компьютерной графикой. За это время он успел поработать в нескольких ведущих компаниях по созданию 3D-графики, включая Autodesk, Electronic Arts, PDI/DreamWorks и WB Games. Там Алехандро принимал участие в разных проектах по реализации графики в реальном времени, будь то приложения, игры, игровые движки или фреймворки. Сейчас он занимается разработкой программного обеспечения для GPU в ведущей компании по производству бытовой электроники в Купертино, штат Калифорния.

Благодарности

Книга, которую вы собираетесь прочесть, создавалась почти 20 лет. Она появилась благодаря стараниям многих людей.

- **Омар Паганини** (Omar Paganini) и **Эрнесто Окампо Эдье** (Ernesto Ocampo Edye). Будучи деканами инженерного факультета и факультета компьютерных наук в Католическом университете Уругвая, они доверили мне, студенту четвертого курса, руководство направлением компьютерной графики, позволив полностью переработать учебную программу по своему усмотрению. Весь первый год моего преподавания профессор **Роберто Люблинерман** (Roberto Lublinerman) был моим наставником.
- **Мои студенты с 2003 по 2008 год.** Они оказались невольными подопытными для оттачивания моего преподавательского мастерства, но это не мешало им принимать и уважать профессора, который был старше их всего на год (а в некоторых случаях даже младше). Я понимал, что все мои труды не напрасны, когда видел их сияющие от радости лица после создания первого изображения по принципу трассировки лучей.
- **Алехандро Сеговия Азапиан**. Он прошел путь от моего студента до ассистента. В итоге Алехандро стал моим близким другом, который время от времени помогал мне в проработке материалов книги. Я горжусь тем, что причастен к его становлению в качестве высококлассного специалиста по оптимизации производительности и рендерингу в реальном времени. Эта книга стала такой благодаря его технической редактуре.
- **Дж. К. Ван Винкель** (J. C. Van Winkel) проделал дополнительную редакторскую работу, внеся много ценных предложений по улучшению содержания.
- **Читатели Hacker News.** Мои заметки лекций, чертежи и демки размещались на первой полосе портала *Hacker News*. Этим интересовались многие, в том числе издательство No Starch Press. Благодаря этому счастливому стечению обстоятельств моя книга увидела свет.
- **Билл Поллок** (Bill Pollock), **Алекс Фрид** (Alex Freed), **Кэсси Андреадис** (Kassie Andreadis) и вся команда No Starch Press. Эти люди помогли довести до ума и переоформить мои, как мне казалось, готовые для публикации конспекты с чертежами в реальную книгу. Никогда бы не подумал, что это может потребовать стольких усилий. На обложке лишь мое имя, но нужно понимать, что книга стала продуктом коллективных стараний.

Введение

Компьютерная графика — удивительная тема. Представьте, как можно перейти от набора геометрических данных и нескольких алгоритмов к спецэффектам для «Звездных войн» и «Мстителей», «Историй игрушек» и «Холодного сердца» либо популярных игр вроде Fortnite или Call of Duty?

При этом область компьютерной графики еще и пугающе огромна: от рендеринга 3D-сцен до создания фильтров изображений, от цифровой типографии до симуляции систем частиц. Одной книги будет мало для раскрытия всех дисциплин, связанных с этой темой. Скорее, потребуется целая библиотека. Здесь же мы сосредоточимся только на рендеринге 3D-сцен.

Книга основана на моем многолетнем опыте преподавания. Это моя скромная попытка доступно представить этот срез компьютерной графики. Книга будет понятна и старшекласснику, но при этом в ней много ценного материала и для профессиональных разработчиков. В издании рассматриваются те же темы, что и в полноценном университетском курсе.

Для кого эта книга

Она будет полезна как школьникам, так и матерым профессионалам. Работая над текстом, я сознательно склонялся в пользу простоты и ясности изложения. Вы увидите это по рассматриваемым идеям и алгоритмам. Везде, где достичь результата можно было несколькими путями, я предпочитал самый понятный, стараясь не усложнять рассуждения и не вносить путаницу. Ориентиром мне служила фраза Альберта Эйнштейна: «Делай просто, насколько это возможно, но не проще».

Здесь вам не пригодятся особые знания или программные и аппаратные зависимости. Единственный используемый в книге примитив — метод, позволяющий устанавливать цвет пикселя. Все максимально приближено к освоению материала *с нуля*.

Алгоритмы здесь просты, а используемая в них математика вполне понятна — максимум немного тригонометрии из курса средней школы. Нам понадобится и линейная алгебра, но в книге есть краткое приложение, где практично изложена вся важная информация.

Охват книги

Начав с азов, мы придем к созданию двух полноценных модулей рендеринга: трассировщика лучей и растеризатора. Несмотря на абсолютно разные подходы, результат будет схожим. Посмотрите на рис. В.1.

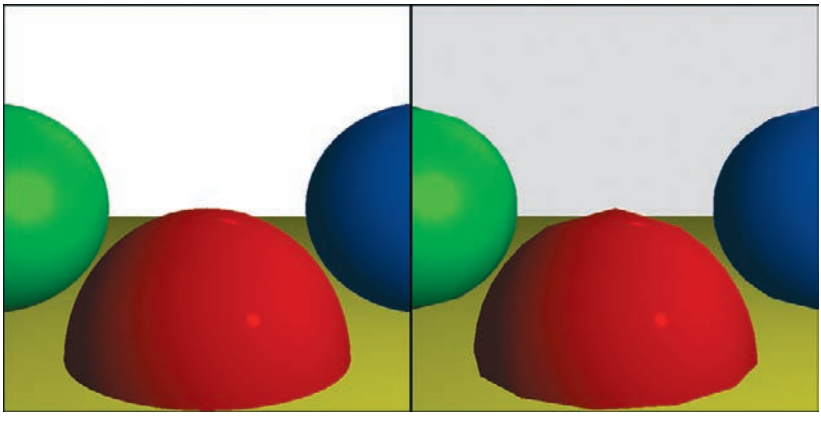


Рис. В.1. Простая сцена, отрисованная трассировщиком лучей *(слева)* и растеризатором *(справа)*, созданными по материалам этой книги

Возможности трассировщика лучей и растеризатора пересекаются, но не одинаковы. В дальнейшем мы рассмотрим их отличительные черты. Некоторые вы можете увидеть на рис. В.2.

Рис. В.2. И у трассировщика лучей, и у растеризатора есть характерные особенности. *Слева:* тени с трассировкой лучей и рекурсивные отражения. *Справа:* растеризованные текстуры

На протяжении книги приводятся неформальный псевдокод и ссылки на полноценные рабочие JavaScript-реализации, которые можно выполнять в любом браузере.

Зачем читать эту книгу

Вы получите знания, необходимые для написания программных модулей рендеринга. Здесь мы не будем использовать или изучать существующие API рендеринга: OpenGL, Vulkan, Metal или DirectX.

Современные графические ускорители мощны и общедоступны. Поэтому редко приходится писать чистый программный рендер, но такой навык будет полезен по ряду причин.

- **Шейдеры и ПО.** Первые GPU из начала 1990-х реализовывали свои алгоритмы рендеринга аппаратно. Их можно было использовать, но не изменять (именно поэтому большинство игр из середины 1990-х выглядят похоже). Сегодня разработчики пишут свои алгоритмы рендеринга *(шейдеры)*, которые выполняются на специальных микросхемах GPU.
- **Знание — сила.** Понимая теории разных техник рендеринга, вы будете не просто копировать недопонятые фрагменты кода и слепо следовать популярным подходам. Вы сможете писать более эффективные шейдеры и конвейеры рендеринга.
- **Графика — это весело.** Работа с графикой — одна из немногих областей компьютерных наук, где мы получаем быструю отдачу. Вы можете бесконечно радоваться, верно выполнив SQL-запрос. Но это *ничто* по сравнению с впечатлением от первого успеха в построении отражений по трассировке лучей. Несколько лет я преподавал в университете компьютерную графику и часто задавался вопросом, почему мне никак не надоест учить одному и тому же семестр за семестром. Но, когда я видел радостные лица студентов, которые ставили свои первые готовые сцены на заставки Рабочего стола, я понимал, что все это не напрасно.

Структура издания

Книга состоит из двух частей: «Трассировка лучей» и «Растеризация», соответствующих двум модулям рендеринга, которые мы будем создавать. В первой главе даются базовые знания. Лучше читать все главы поочередно, но обе части достаточно самостоятельны и для независимого изучения. Для большего удобства прочтите описание каждой главы.

- **Глава 1. Вводные понятия.** Здесь дается определение холсту и единственному инструменту для рисования — `PutPixel`. В этой же главе вы научитесь представлять цвета и управлять ими.

Часть I. Трассировка лучей

- **Глава 2. Базовая трассировка лучей.** Здесь мы создадим простой алгоритм трассировки лучей, способный отрисовывать несколько сфер, которые будут выглядеть как цветные круги.
- **Глава 3. Свет.** В этой главе мы определим модель взаимодействия света с объектами и расширим трассировщик возможностью симуляции света. Теперь сферы будут выглядеть как сферы.
- **Глава 4. Тени и отражения.** Дальше улучшаем внешний вид сфер: теперь они отбрасывают друг на друга тени и могут иметь зеркальные поверхности.
- **Глава 5. Расширение возможностей трассировщика лучей.** Здесь дается обзор дополнительных возможностей трассировщика, которые не вписываются в рамки книги, но по желанию их можно добавить.

Часть II. Растеризация

- **Глава 6. Прямые.** Начинаем с пустого холста и создаем алгоритм для рисования отрезков.
- **Глава 7. Закрашенные треугольники.** Используя некоторые идеи из предыдущей главы, мы разработаем алгоритм рисования треугольников, заполненных одним цветом.
- **Глава 8. Затененные треугольники.** Расширяем алгоритм из главы 7 для закрашивания треугольников плавным цветовым градиентом.
- **Глава 9. Перспективная проекция.** Здесь мы отвлечемся от 2D-рисования, чтобы рассмотреть геометрические и математические принципы для преобразования 3D-точки в 2D-точку, которую можно будет нарисовать на холсте.
- **Глава 10. Описание и рендеринг сцены.** Разрабатываем представление для объектов сцены и изучаем использование перспективной проекции для их отрисовки на холсте.
- **Глава 11. Отсечение.** Создаем алгоритм удаления невидимых для камеры частей сцены, позволяющий безопасно отрисовывать ее из любой позиции камеры.
- **Глава 12. Удаление скрытых поверхностей.** Совмещаем перспективную проекцию и затененные треугольники для рендеринга визуально твердых объектов. Выполнить задачу успешно можно, сделав так, чтобы удаленные объекты не перекрывали более близкие.
- **Глава 13. Затенение.** Учимся применять уравнение освещенности из главы 3 к целым треугольникам.
- **Глава 14. Текстуры.** Разрабатываем алгоритм для имитации деталей поверхности на наших треугольниках.

- **Глава 15. Расширение растеризатора.** Обзор возможностей для растеризатора, выходящих за рамки книги.
- **Приложение. Линейная алгебра.** Знакомимся с используемыми в книге базовыми понятиями из линейной алгебры: точками, векторами и матрицами. Рассматриваем операции, которые можно выполнять с этими элементами, и некоторые примеры их применения.

И еще об авторе

Сейчас я работаю ведущим инженером в Google. До этого же трудился в компании Improbable (http://improbable.com), у которой есть неплохие шансы создать настоящую матрицу (или в корне изменить разработку мультиплеерных игр). Около десяти лет я руководил собственной компанией по разработке компьютерных игр Mystery Studio (http://mysterystudio.com). За это время мы выпустили почти 20 игр, хотя вы о них наверняка даже не слышали.

Пять лет я преподавал компьютерную графику один семестр на третьем курсе и признателен всем моим студентам, благодаря которым у меня была возможность отточить все знания и навыки, лежащие в основе этой книги.

У меня есть и другие увлечения, помимо компьютерной графики, в том числе не инженерного характера. Приглашаю посетить мой сайт http://gabrielgambetta.com, где есть много дополнительной информации и контактные данные для связи.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

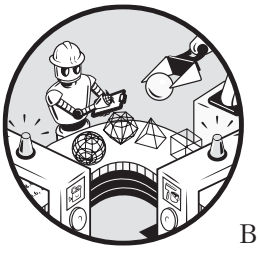
На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

О научном редакторе русскоязычного издания

Дмитрий Соколов окончил мат-мех СПбГУ, хабилитированный доктор, доцент университета Лотарингии, руководитель небольшой научной группы PIXEL (https://pixel.inria.fr/). На платформе GitHub у него 2,6 тысячи подписчиков (https://github.com/ssloy), а его репозитории по компьютерной графике в сумме имеют двадцать тысяч звезд.

1

Вводные понятия



Трассировщик лучей и растеризатор реализуют рендеринг 3D-сцены на 2D-поверхности по-разному. Но у обоих подходов есть несколько базовых общих принципов.

В этой главе мы познакомимся с холстом для отрисовки изображений и системой координат, с помощью которой будем ссылаться на пиксели холста. Еще мы научимся представлять цвета и управлять ими, а также описывать 3D-сцену для ее обработки модулями рендеринга.

Холст

Мы будем рисовать на *холсте* — прямоугольном массиве пикселей, которые могут закрашиваться независимо. В нашем случае неважно, на экране холст или на бумаге. Нам нужно будет представлять 3D-сцену на 2D-холсте, поэтому рендеринг мы будем делать на этом абстрактном прямоугольном массиве пикселей.

Все реализации в этой книге будут строиться на основе одной простой функции, присваивающей цвет пикселю холста:

```
canvas.PutPixel(x, y, color)
```

В этом методе находится три аргумента: координата `x`, координата `y` и цвет. Начнем с координат.

Система координат

У холста есть измеряемые в пикселях ширина и высота, которые мы назовем C_w и C_h. Для обращения к этим пикселям нужна система координат. На большинстве компьютерных экранов исходная точка отсчета будет в верхнем левом углу. При этом x возрастает вправо, а y — вниз, как показано на рис. 1.1.

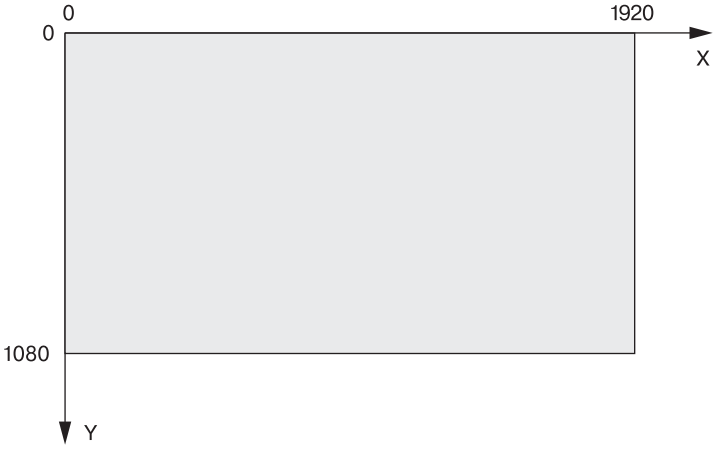


Рис. 1.1. Система координат, используемая на большинстве компьютерных экранов

Эта система координат естественна для компьютера, так как опирается на принцип организации видеопамяти, но не очень удобна для людей. Вместо этого программисты 3D-графики используют систему координат для рисования графиков на бумаге: исходная точка находится в центре, x возрастает вправо и уменьшается влево, а y возрастает вверх и уменьшается вниз, как показано на рис. 1.2.

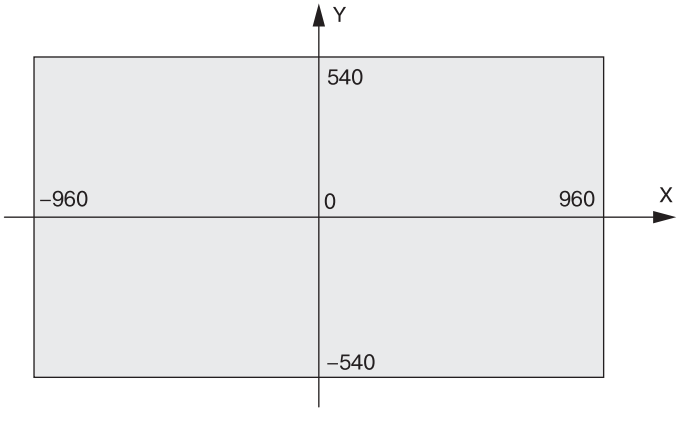


Рис. 1.2. Система координат, которую мы будем использовать для холста

При использовании этой системы координат диапазон x представлен как $\left[\frac{-C_w}{2}, \frac{C_w}{2}\right)$, а y — как $\left[\frac{-C_h}{2}, \frac{C_h}{2}\right)$. Предположим, что использование функции `PutPixel` с выходящими за указанный диапазон координатами ничего не дает.

В наших примерах холст будет отрисовываться на экране, значит, нам придется выполнять преобразование из одной системы координат в другую. Для этого нужно изменить центр системы и обратить направление оси *Y*. Вот итоговые уравнения преобразования:

$$S_x = \frac{C_w}{2} + C_x;$$
$$S_y = \frac{C_h}{2} \quad C_y.$$

Предположим, что `PutPixel` выполняет преобразование автоматически. Теперь можно представлять холст как имеющий исходную точку координат в середине, с x, возрастающим вправо, а y — вверх экрана.

Взглянем на последний аргумент `PutPixel`: цвет.

Цветовые модели

Теория цвета удивительна, но выходит за рамки этой книги, поэтому ниже я приведу упрощенную версию только интересующих нас аспектов.

Когда свет попадает в человеческий глаз, он стимулирует светочувствительные клетки на его задней стенке. Они генерируют сигналы для мозга согласно длине волны воспринимаемого света, которые мы интерпретируем как *цвета*.

В обычных условиях человек не может видеть длину волны, выходящую за *видимый диапазон*. Длина волны и частота находятся в обратной зависимости (чем выше частота волны, тем меньше расстояние между ее пиками). Именно поэтому инфракрасный свет (длина волны больше 740 нм, что соответствует частотам ниже 405 ТГц) безвреден, а ультрафиолетовый (длина волны менее 380 нм, что соответствует частотам выше 790 ТГц) может обжечь кожу.

Любой вообразимый цвет можно описать как разные комбинации этих цветов. «Белый» — это сумма всех цветов, а «черный» — их полное отсутствие. Описывать цвета через их длины волн очень неудобно, но почти все их можно создать линейной комбинацией всего трех *основных цветов*.

Субтрактивная цветовая модель

Субтрактивная цветовая модель — это просто вычурное название для процесса смешивания цветов, знакомого вам с детства. Представьте, что берете белый лист бумаги и красный, синий и желтый карандаши. Сначала вы рисуете желтый круг и пересекающийся с ним синий, получая зеленый цвет. При наложении желтого и красного получается оранжевый, при совмещении красного и синего — пурпурный. Если смешать все три цвета, то выйдет почти черный. Детство — увлекательная

пора, не так ли? На рис. 1.3 показаны основные цвета субтрактивной модели и создаваемые их смешением вариации.

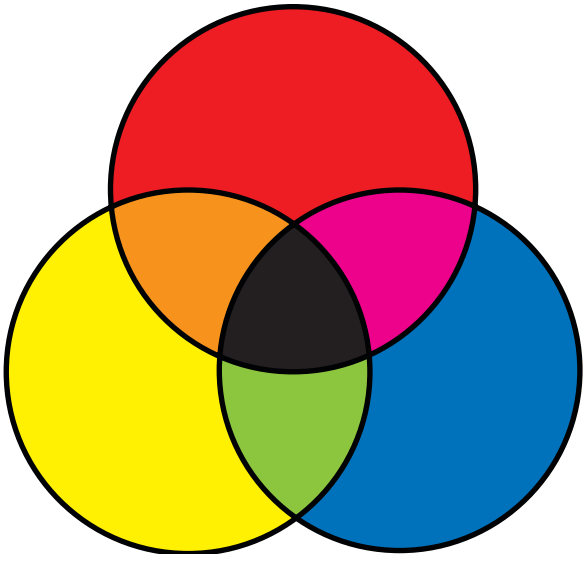


Рис. 1.3. Основные цвета субтрактивной схемы и их комбинации

У объектов разный цвет из-за того, что они по-разному поглощают и отражают свет. Начнем с белого света, наподобие солнечного (солнечный свет нельзя назвать белым, но для наших целей достаточно их сходства). Белый свет содержит весь диапазон световых волн. После столкновения с объектом поверхность последнего поглощает некоторые волны, а остальные отражает, в зависимости от материала. После часть отраженного света достигает наших глаз, и мозг преобразует эту информацию в цвет. Какой именно? Тот, что представляет сумму длин волн, отраженных поверхностью.

А что происходит с карандашами? Мы начинаем с белого света, отражающегося от бумаги. Она белая, поэтому отражает большую часть падающего на нее света. При рисовании желтым карандашом вы добавляете слой материала, поглощающего волны определенной длины, но пропускающего другие. Они отражаются бумагой, снова проходят через желтый слой, достигают наших глаз, и мозг интерпретирует именно эту комбинацию длин волн как «желтый». В этом случае желтый слой *вычитает (субтрактирует)* часть волн из исходного белого света.

Каждый цветной круг можно рассматривать как фильтр. При рисовании синего круга, накладывающегося на желтый, вы отфильтровываете из исходного света еще больше длин волн и глаз достигают те, что не отсеялись синим или желтым кругом, а мозг их воспринимает как «зеленый».

Короче говоря, мы начинаем со всеми длинами волн и для создания нужного цвета вычитаем их часть из основных цветов. В этой модели цвета получаются путем вычитания определенных длин волн из основных цветов, чем и обусловлено ее название.

Но эта модель не совсем верна. На самом деле основные цвета субтрактивной схемы — не синий, красный и желтый, как учат детей и студентов, а циан, маджента и желтый. Более того, смешение этих трех основных цветов создает очень темный тон, который не является полностью черным. Поэтому в качестве четвертого основного добавляется чистый черный цвет. В результате мы получаем *цветовую схему CMYK* (Cyan, Magenta, Yellow, Key или Black) (рис. 1.4).

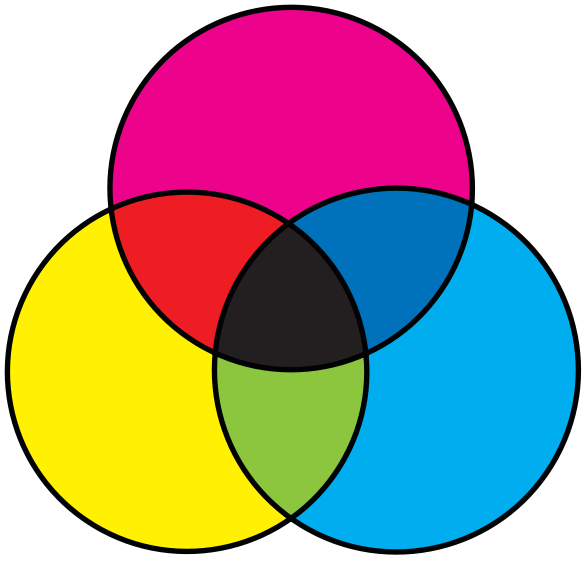


Рис. 1.4. Четыре основных субтрактивных цвета, используемые в картриджах

Эта схема используется в картриджах цветных принтеров, а иногда ее можно увидеть и в контурах картинок дешевых флаеров, где разные цвета немного съезжают друг с друга.

Аддитивная цветовая модель

Субтрактивная модель — это только половина истории. Если вы когда-нибудь разглядывали экран вблизи или через увеличительное стекло (либо, чего уж греха таить, случайно на него чихали), то наверняка видели мелкие цветные точки: красные, зеленые и синие.

Принцип работы экранов компьютеров противоположен бумаге, которая не излучает свет, а только отражает часть получаемого. Здесь мы начинаем с белого цвета и *вычитаем* ненужные длины волн. Экраны же, наоборот, черные, но при этом сами излучают свет. И в таком случае мы начинаем без цвета и *добавляем* волны нужной длины. Для этого необходимы другие основные цвета. Большинство из них можно создать, добавляя к черной поверхности красный, зеленый и синий в разных пропорциях. Это называется *цветовой схемой RGB* или *аддитивной цветовой моделью*, показанной на рис. 1.5.

Комбинация аддитивных основных цветов *светлее* ее компонентов, а комбинация субтрактивных основных цветов — *темнее*. Все основные аддитивные цвета складываются в белый, а субтрактивные — в черный.

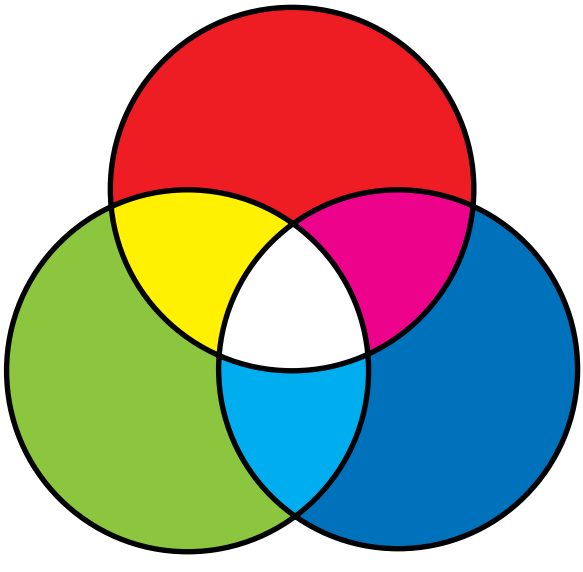


Рис. 1.5. Аддитивные основные цвета и некоторые из их комбинаций

Забудьте лишние детали

Теперь, когда вам все это известно, вы можете выборочно забыть детали и сосредоточиться на самом важном для нашей работы.

Большинство цветов можно выразить в RGB или в CMYK (либо в другой цветовой схеме), преобразовывая при этом одно *цветовое пространство* в другое. Наш приоритет — это рендеринг изображений на экране, поэтому в книге мы будем использовать схему RGB.

Как уже говорилось, объекты поглощают часть достигающего их света, а остальной отражают. Воспринимаемый нами «цвет» поверхности определяется длинами поглощаемых и отражаемых волн. С этого момента мы будем рассматривать цвет как свойство поверхности и забудем о длинах волн.

Глубина цвета и представление

Цвет в мониторе зависит от смешения в разных пропорциях красного, зеленого и синего. Для этого мелкие цветные точки экрана подсвечиваются с разной интенсивностью, которая определяется разницей в подаваемом напряжении.

Сколько же значений интенсивности можно получить? Несмотря на непрерывность напряжения, мы будем управлять цветами с помощью компьютера, использующего дискретные величины (то есть их ограниченное число). Чем больше оттенков красного, зеленого и синего мы можем представить, тем больше цветов можно создать.

В большинстве встречающихся сегодня изображений используется по 8 бит на один основной цвет. Это называется *цветовым каналом*. Используя 8 бит на канал, мы получаем 24 бита на пиксель, итого 2^{24} цветов (примерно 16,7 миллиона). Именно такой формат, известный как *R8G8B8* или просто *888*, мы и будем использовать в книге. Об этом формате можно сказать, что его *глубина цвета* — 24 бита.

Но это не единственный возможный формат. Не так давно для экономии памяти в ходу были 15- и 16-битные аналоги. В случае с 15 битами присваивалось по 5 бит на канал, а с 16 битами — 5 бит для красного, 6 для зеленого и 5 для синего (этот формат назывался *R5G6B5* или *565*). Зеленому выделяется дополнительный бит, потому что наши глаза более чувствительны к изменению именно в этом цвете.

При использовании 16 бит мы получаем 2^{16} цветов (примерно 65 000), то есть по одному для каждых 256 цветов в 24-битном режиме. Несмотря на то что 65 000 цветов — это очень много, на изображениях, где цвета изменяются очень плавно, можно заметить едва уловимые «переходы». Но при использовании 16,7 миллиона цветов их не видно, так как здесь уже достаточно битов для всех промежуточных оттенков. В некоторых специализированных приложениях, таких как цветоустановка для кинофильмов, хорошим решением будет задействовать дополнительные биты на каждый канал для расширения спектра цветовых деталей.

Мы будем использовать для выражения цвета 3 байта со значением 8-битного цветового канала в диапазоне от 0 до 255 в каждом. Цвета мы будем выражать как (*R*, *G*, *B*) — например, (255, 0, 0) представляет чистый красный, (255, 255, 255) — белый, а (255, 0, 128) — красновато-пурпурный.

Управление цветом

Для управления цветом мы будем использовать небольшой набор операций. Если вы знакомы с линейной алгеброй, то можете представить цвета как векторы в трехмерном цветовом пространстве. Если же нет, то ничего страшного, сейчас мы разберем все базовые операции, которые будем использовать.

Интенсивность цвета можно изменять, умножая каждый из его цветовых каналов на константу:

$$k(R, G, B) = (kR, kG, kB).$$

Например, (32, 0, 128) вдвое ярче, чем (16, 0, 64).

Мы можем совместить два цвета, по отдельности сложив их цветовые каналы:

$$(R_1, G_1, B_1) + (R_2, G_2, B_2) = (R_1 + R_2, G_1 + G_2, B_1 + B_2).$$

Если нужно совместить красный (255, 0, 0) и зеленый (0, 255, 0), то мы складываем их поканально и получаем (255, 255, 0) — желтый.

Учтите, что эти операции могут давать недействительные значения. Удваивание интенсивности (192, 64, 32) дает значение R, выходящее за цветовой диапазон. Любое значение больше 255 мы будем принимать за 255, а любое значение меньше 0 — за 0. Такой прием называется *обрезкой* значения до диапазона 0–255. Это чем-то похоже на то, что происходит, когда мы делаем недо- или переэкспонированный фотоснимок: получаем полностью черные либо полностью белые области.

На этом вводная часть по теме цветов и `PutPixel` заканчивается. Теперь уделим немного времени изучению представления 3D-объектов при рендеринге.

Сцена

Вы уже познакомились с холстом (абстрактной поверхностью, на которой закрашиваются пиксели). Теперь перейдем к отображению объектов, рассмотрев еще одну абстракцию — *сцену*.

Сцена — набор объектов, которые вы отрисовываете. Это может быть что угодно: от сферы, дрейфующей в бесконечном пустом пространстве (начнем мы именно с этого), и до невероятно детализированной модели внутренностей угрюмого носа огра.

Для рассмотрения объектов в рамках сцены нам нужна система координат. Использовать ту же систему, что и для холста, мы не можем по двум причинам: во-первых, холст двухмерный, а сцена трехмерная, во-вторых, холст и сцена используют разные

единицы измерения: на холсте — пиксели, а для сцены — реальные единицы (имперская или метрическая системы).

Выбор осей здесь произволен, поэтому мы возьмем самые подходящие для нас. Предположим, что *Y* вертикальная, а *X* и *Z* горизонтальные и все три оси перпендикулярны друг другу. Рассматривайте плоскость *XZ* как «пол», а *XY* и *YZ* —как вертикальные «стены» квадратной комнаты. Это согласуется с системой координат, выбранной нами для холста, где *Y* — вертикаль, а *X* — горизонталь. На рис. 1.6 показано, как все это выглядит.

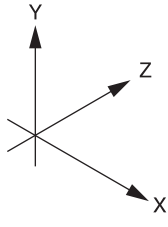


Рис. 1.6. Система координат для сцен

Выбор единиц измерения сцены тоже произволен и зависит от того, что она будет представлять. К примеру, 1 может означать 1 дюйм при моделировании чашки или 1 астрономическую единицу при моделировании Солнечной системы. Неважно, что выражают выбранные единицы измерения, пока мы согласованно их используем. С этого момента можно смело их игнорировать.

Итоги главы

В этой главе вы познакомились с холстом, абстракцией и с базовым методом для построения всего остального: `PutPixel`. Еще мы выбрали систему координат для ориентации в пространстве пикселей холста и рассмотрели способ представления цвета этих пикселей. В завершение было введено понятие сцены и для нее тоже выбрана система координат.

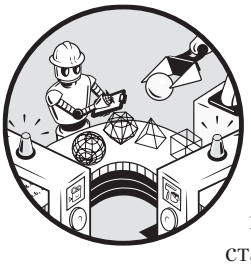
Теперь можно приступать к созданию трассировщика лучей и растеризатора на базе заложенных нами основ.

ЧАСТЬ I

ТРАССИРОВКА ЛУЧЕЙ

2

Базовая трассировка лучей



Эта глава посвящена алгоритму трассировки лучей. Начнем с рассмотрения его сути и написания простого псевдокода, потом вы научитесь представлять лучи света и объекты в сцене. В конце мы выработаем способ вычисления лучей, создающих видимое изображение объектов сцены, и вы узнаете, как представлять их на холсте.

Рендеринг ландшафта Швеции

Представим, что мы направились в одно из экзотических мест планеты, где увидели настолько потрясающий пейзаж, что очень хочется запечатлеть его прелесть. На рис. 2.1 вы видите пример такого пейзажа.

У нас есть холст и кисточка для рисования, но художественный талант напрочь отсутствует. Неужели надежды нет? Вовсе не обязательно. Таланта у нас может и не быть, но зато есть методический подход. Поэтому мы берем москитную сетку, вырезаем из нее прямоугольный кусок, вставляем в рамку и крепим к палке. Затем мы выбираем лучшую точку обзора красот ландшафта и ставим другую палку как отметку точной позиции, где должен находиться глаз.

Рисовать мы еще не начали, но теперь у нас есть фиксированная точка обзора и рамка, через которую можно наблюдать ландшафт. Эта статичная рамка делится сеткой на мелкие клетки. А вот теперь очередь методологии. Мы рисуем на холсте сетку с таким же количеством клеток, что и в москитной рамке. Теперь смотрим на ее верхнюю левую клетку: какой преобладающий цвет мы через нее видим? Небесно-голубой. Значит, левую верхнюю клетку закрашиваем небесно-голубым. Так мы делаем для каждой, и вскоре холст уже начинает выражать неплохую картину пейзажа, похожую на вид через рамку. Получившееся изображение можно увидеть на рис. 2.2.

Рис. 2.1. Потрясающий ландшафт в Швеции

Рис. 2.2. Грубое представление ландшафта

Если подумать, то компьютер — очень методичная машина без всякого художественного таланта. А процесс создания картины можно описать так:

```
Каждую маленькую клетку холста
    Закрасить правильным цветом
```

Очень просто! И все же эта формулировка слишком расплывчата для выполнения на компьютере. Поэтому углубимся в детали:

```
Нужным образом разместить точку обзора и рамку
Для каждой клетки холста:
    Определить, какая клетка сетки соответствует этой клетке на холсте
    Определить цвет, наблюдаемый через клетку сетки
    Закрасить клетку этим цветом
```

Даже такое описание все еще слишком размыто, хотя уже похоже на алгоритм. Вы удивитесь, но это и есть высокоуровневое представление всего алгоритма трассировки лучей. Все верно, ничего сложного.

Основные допущения

Частично шарм компьютерной графики — в рисовании на экране. Для ускорения этого процесса мы сделаем ряд упрощающих допущений.

Эти допущения немного ограничат наши возможности, но в следующих главах мы от них избавимся.

Во-первых, предположим фиксированную точку обзора. Такая точка обзора, где в примере со шведским ландшафтом мы размещаем глаз, обычно называется *позицией камеры*. Назовем ее O. Допустим, что камера занимает одну точку в пространстве, находится в исходной позиции системы координат и никогда не смещается. То есть сейчас $O = (0, 0, 0)$.

Во-вторых, предположим фиксированную ориентацию камеры. Ориентация камеры определяет ее направление. Пусть в нашем случае она смотрит в направлении положительной части оси Z (обозначим ее как $\vec{Z}_+$) при положительной части оси Y ($\vec{Y}_+$), идущей вверх, и положительной части оси X ($\vec{X}_+$), идущей вправо (рис. 2.3).

Теперь позиция и ориентация камеры фиксированы. Но для полного соответствия аналогии не хватает «рамки», через которую мы смотрим на сцену. Допустим, что у нее размеры V_w и V_h и она находится перед камерой (перпендикулярно оси $\vec{Z}_+$). Предположим также, что от камеры рамка находится на расстоянии d, ее стороны параллельны осям X и Y и она центрирована относительно $\vec{Z}_+$. Звучит запутанно, но на деле все просто. Взгляните на рис. 2.4.

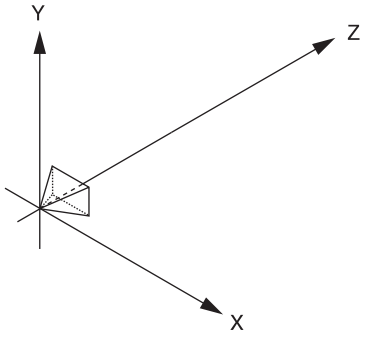


Рис. 2.3. Позиция и ориентация камеры

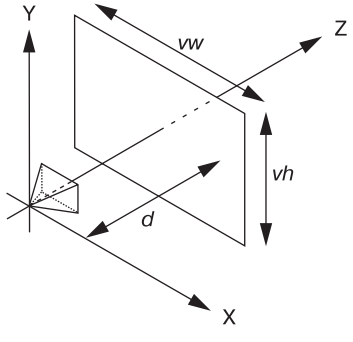


Рис. 2.4. Позиция и ориентация окна просмотра

Прямоугольник, который будет окном в мир, называется *окном просмотра*. Обратите внимание, что его размер и расстояние до камеры определяют угол видимости камеры — *поле зрения*, или FOV (field of view). Человеческое поле зрения охватывает почти 180° по горизонтали (но большая его часть — размытое периферийное зрение без ощущения глубины). Для простоты мы определим $V_w = V_h = d = 1$. В результате FOV будет составлять примерно 53°. Это ведет к получению не сильно искаженных адекватных изображений.

Вернемся к алгоритму и, пользуясь подходящими терминами, пронумеруем шаги в листинге 2.1.

Листинг 2.1. Описание алгоритма трассировки лучей

```
❶ Поместить камеру и окно просмотра нужным образом
  Для каждого пикселя на холсте:
    ❷ Определить, какая клетка окна просмотра соответствует этому пикселю
    ❸ Определить цвет, видимый через эту клетку
    ❹ Закрасить пиксель этим цветом
```

Мы только что закончили шаг ❶ (точнее, пока проигнорировали его). Шаг ❹ тривиален: здесь просто используется `canvas.PutPixel(x, y, color)`. Давайте разберем шаг ❷, а потом сосредоточимся на более изощренных способах выполнения шага ❸. Этому будет посвящено несколько следующих глав.

С холста на окно просмотра

Шаг ❷ нашего алгоритма из листинга 2.1 просит `определить, какая клетка окна просмотра соответствует этому пикселю`. Координаты пикселя на холсте — назовем их C_x и C_y — нам известны. Заметьте, что мы очень удачно разместили окно просмотра и ориентация его осей соответствует ориентации осей холста, а центр совпадает с его центром. Окно просмотра измеряется в глобальных единицах, а холст — в пикселях, поэтому для перехода от координат холста к пространственным координатам нужно лишь изменить масштаб.

$$V_x = C_x \frac{V_w}{C_w};$$

$$V_y = C_y \frac{V_h}{C_h}.$$

Есть еще кое-что. Окно просмотра двухмерное, но встроено в трехмерное пространство. Мы определили его расположение на расстоянии d от камеры. Значит, для каждой точки на этой плоскости (называемой *плоскостью проекции*) по определению $z = d$, следовательно:

$$V_z = d.$$

С этим шагом мы покончили. Для каждого пикселя (C_x, C_y) холста мы можем определить соответствующую ему точку в окне просмотра (V_x, V_y, V_z).

Трассировка лучей

Следующий шаг — выяснить, какой цвет имеет свет, проходящий через (V_x, V_y, V_z) с позиции обзора камеры (O_x, O_y, O_z).

В реальном мире свет исходит от источника (солнце, лампа накаливания и т. д.), отражается от нескольких объектов и достигает наших глаз. Можно попробовать воссоздать путь каждого фотона, покидающего имитированные нами источники света, но для этого нужно *чрезвычайно* много времени. Придется симулировать немыслимое число фотонов (одна лампа на 100 Вт испускает 10^{20} фотонов в секунду), и только их малая часть после прохождения через окно просмотра достигнет (O_x, O_y, O_z).

Такая техника называется *трассировкой* или *отображением фотонов*, но она выходит за рамки этой книги.

Вместо этого мы рассмотрим лучи света «наоборот». Начнем с исходящего от камеры луча, проходящего через точку в окне просмотра и прокладывающего свой путь вплоть до достижения объекта сцены. Данный объект — это то, что камера «видит» через конкретную точку окна просмотра. Итак, в качестве первой аппроксимации мы просто возьмем цвет этого объекта как «цвет света, проходящего через эту точку», как показано на рис. 2.5.

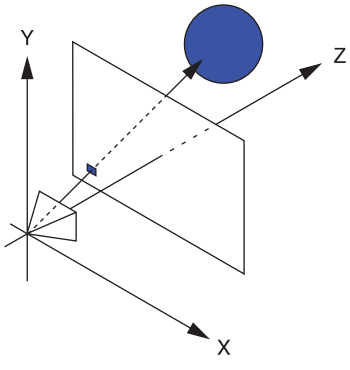


Рис. 2.5. Маленькая клетка в окне просмотра, представляющая один пиксель холста, закрашенный цветом объекта, который камера через него видит

Теперь нам нужны кое-какие уравнения.

Уравнение траектории луча

Для нас удобнее всего будет представить луч с помощью параметрического уравнения. Мы знаем, что он проходит через O, и знаем его направление (от O к V), значит, можно выразить любую точку P в луче как:

$$P = O + t(V - O).$$

Здесь t — любое вещественное число. Путем подстановки в это уравнение каждого значения t от 0 до $+\infty$ мы получаем каждую точку P вдоль луча.

Обозначим $(V - O)$, направление луча, как $\vec{D}$. Тогда уравнение станет таким:

$$P = O + t\vec{D}.$$

Чтобы лучше понять это уравнение, представьте, что мы начинаем луч в исходной точке (O) и «продвигаемся» вдоль его направления $(\vec{D})$ на некоторую величину (t).

Такое продвижение включает все точки вдоль луча. Подробнее об этих векторных операциях можно узнать в приложении «Линейная алгебра». На рис. 2.6 показано наше уравнение в действии.

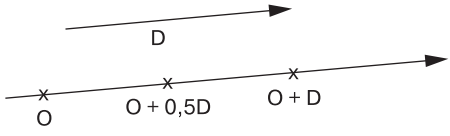


Рис. 2.6. Некоторые точки луча $O + t\vec{D}$ для разных значений t

На рис. 2.6 показаны точки вдоль луча, которые соответствуют $t = 0{,}5$ и $t = 1{,}0$. Каждое значение t дает другую точку вдоль луча.

Уравнение сферы

Теперь нам нужен объект, чтобы лучам было с чем столкнуться. Для строительных элементов сцен мы можем взять любой геометрический примитив. Для трассировки лучей мы воспользуемся сферами, потому что ими проще управлять через уравнения.

Сфера — это множество точек, пролегающих на равном расстоянии от некоторой заданной точки. Это расстояние называется *радиусом* сферы. На рис. 2.7 показана сфера, определенная по ее *центру* C и радиусу r.

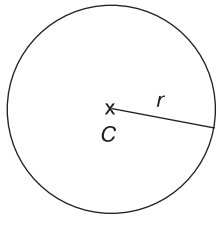


Рис. 2.7. Сфера, определенная по ее центру и радиусу

Согласно вышеприведенному определению, если C будет центром, а r — радиусом сферы, точки P на поверхности этой сферы должны удовлетворять такому уравнению:

$$\text{Расстояние}(P, C) = r.$$

Немного поиграем с этим уравнением. Если какие-то моменты будут вам непонятны, обратитесь к приложению «Линейная алгебра».

Расстояние между P и C — это длина вектора от P к C:

$$|P - C| = r.$$

Длина вектора $|\vec{V}|$ — это квадратный корень из его скалярного произведения с самим собой $\langle \vec{V}, \vec{V} \rangle$:

$$\sqrt{\langle P - C, P - C \rangle} = r.$$

Чтобы избавиться от квадратного корня, возведем обе стороны в квадрат:

$$\langle P - C, P - C \rangle = r^2.$$

Все эти формулировки уравнения равнозначны, но с последней работать будет удобнее.

Луч встречается со сферой

Теперь у нас есть два уравнения: описывающее точки на сфере и описывающее точки на луче:

$$\langle P - C, P - C \rangle = r^2;$$

$$P = O + t\vec{D}.$$

Пересекаются ли луч и сфера? Если да, то где?

Предположим, что луч пересекается со сферой в точке P. Она есть и вдоль луча, и на поверхности сферы, значит, она должна удовлетворять обоим уравнениям. Обратите внимание, что единственная переменная в этих уравнениях — это параметр t, ведь O, $\vec{D}$, C и r даны, а P — искомая точка.

Так как P в обоих уравнениях — это одна и та же точка, можно подставить вместо нее в первом уравнении выражение для P из второго, что даст нам:

$$\langle O + t\vec{D} - C, O + t\vec{D} - C \rangle = r^2.$$

Если мы можем найти значения t, удовлетворяющие этому уравнению, то сможем подставить их в уравнение луча для нахождения точек, в которых луч пересекает сферу.

Сейчас это уравнение выглядит очень громоздко, поэтому мы попробуем его упростить с помощью алгебраических действий.

Во-первых, пусть $\overrightarrow{CO} = O - C$. Тогда можно записать уравнение как:

$$\langle \overrightarrow{CO} + t\vec{D}, \overrightarrow{CO} + t\vec{D} \rangle = r^2.$$

Используя дистрибутивность скалярного произведения, разобьем его на составляющие (пояснения тоже можете найти в приложении «Линейная алгебра»):

$$\langle \overrightarrow{CO} + t\vec{D}, \overrightarrow{CO} \rangle + \langle \overrightarrow{CO} + t\vec{D}, t\vec{D} \rangle = r^2;$$

$$\langle \overrightarrow{CO}, \overrightarrow{CO} \rangle + \langle t\vec{D}, \overrightarrow{CO} \rangle + \langle \overrightarrow{CO}, t\vec{D} \rangle + \langle t\vec{D}, t\vec{D} \rangle = r^2.$$

Слегка изменив порядок выражений, получаем:

$$\langle t\vec{D}, t\vec{D} \rangle + 2\langle \overrightarrow{CO}, t\vec{D} \rangle + \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle = r^2.$$

Вынесение t за скалярные произведения и перемещение r^2 на другую сторону уравнения дает нам:

$$t^2\langle \vec{D}, \vec{D} \rangle + t(2\langle \overrightarrow{CO}, \vec{D} \rangle) + \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle - r^2 = 0.$$

Напомню, что скалярное произведение двух векторов — это вещественное число, значит, каждое выражение между угловыми скобками тоже будет вещественным. Если дать им имена, то получится уже более знакомая форма:

$$a = \langle \vec{D}, \vec{D} \rangle;$$

$$b = 2\langle \overrightarrow{CO}, \vec{D} \rangle;$$

$$c = \langle \overrightarrow{CO}, \overrightarrow{CO} \rangle - r^2;$$

$$at^2 + bt + c = 0.$$

Перед нами старое доброе квадратичное уравнение. Его решения — это значения параметра t, где луч пересекает сферу:

$$\{t_1, t_2\} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

К счастью, это имеет геометрический смысл. Если помните, квадратичное уравнение может не иметь решений, иметь одно двойное решение или же два разных. Это определяется значением дискриминанта $b^2 - 4ac$. Эти варианты в точности соответствуют случаям, когда луч не пересекает сферу, проходит по касательной к сфере или пересекает ее насквозь (рис. 2.8).

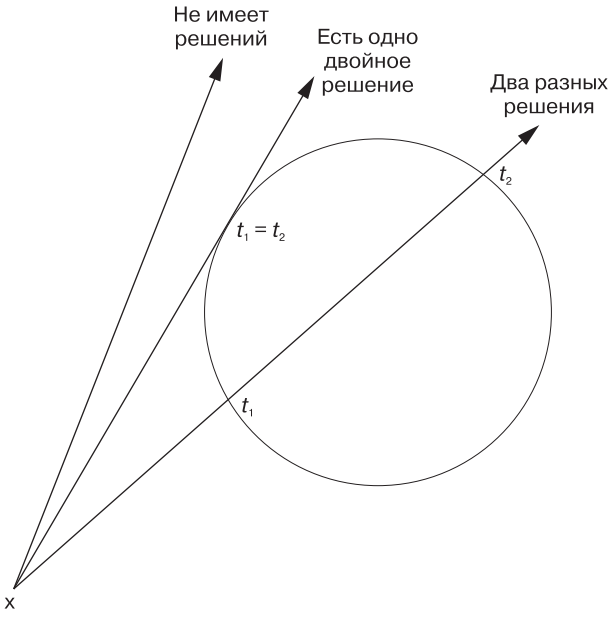


Рис. 2.8. Геометрическая интерпретация квадратичного уравнения: нет решений, одно решение или два решения

Как только мы нашли значение t, можно подставить его обратно в уравнение луча и получить точку пересечения P, соответствующую этому значению.

Рендеринг первых сфер

Подытожим. Для каждого пикселя холста мы можем вычислить соответствующую точку окна просмотра. С учетом позиции камеры можно выразить уравнение исходящего от камеры луча и проходящего через эту точку окна просмотра. С помощью сферы можно вычислить точки, в которых ее пересекает луч.

Все, что нужно сделать, — вычислить пересечения луча с каждой сферой, оставить из них самое близкое к камере и закрасить пиксель на холсте соответствующим цветом. Теперь мы почти готовы к рендерингу наших первых сфер, но параметр t заслуживает больше внимания. Вернемся к уравнению луча:

$$P = O + t(V - O).$$

Исходная точка и направление луча фиксированы, поэтому изменение t в диапазоне вещественных чисел будет давать каждую точку P на этом луче. Обратите внимание,

что для $t = 0$ мы получаем $P = O$, а для $t = 1$ получаем $P = V$. Отрицательные значения t дают точки в противоположном направлении — позади камеры. Поэтому мы можем разделить пространство на три части, как показано в табл. 2.1. На рис. 2.9 оно представлено схематически.

Табл. 2.1. Подразделение параметрического пространства

$t < 0$	Позади камеры
$0 < t < 1$	Между камерой и плоскостью проекции/окном просмотра
$t > 0$	Перед плоскостью проекции/окном просмотра

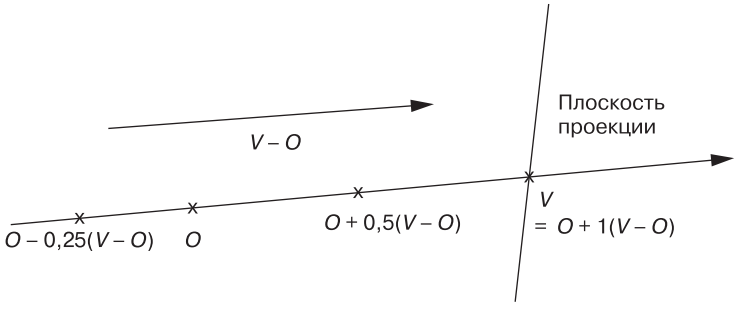


Рис. 2.9. Несколько точек в параметрическом пространстве

Заметьте, ничто в уравнении пересечения не заставляет сферу находиться именно *перед* камерой, и оно без проблем может произвести решения для пересечений *позади* нее. Такой вариант нам не подходит, значит, нужно проигнорировать решения с $t < 0$. Чтобы дальше у нас не возникло проблем, мы ограничим решения диапазоном $t > 1$ — будем отрисовывать все, что находится *за* плоскостью проекции.

Не нужно устанавливать верхний порог значения t с другой стороны, потому что мы хотим видеть все объекты перед камерой, независимо от их удаленности. Но на последующих этапах нам нужно будет сократить лучи, поэтому мы введем эту формальность сейчас, установив значение $+\infty$ верхним порогом t (в языках, которые не могут представлять бесконечность, можно использовать просто очень большое число).

Теперь оформим все проделанное в виде псевдокода. Возьмем за общее правило, что у кода есть доступ к любым нужным данным, чтобы нам не беспокоиться о передаче таких параметров, как холст, а сосредоточиться только на важных.

Основной метод показан в листинге 2.2.

Листинг 2.2. Основной метод

```
O = (0, 0, 0)
for x = -Cw/2 to Cw/2 {
    for y = -Ch/2 to Ch/2 {
        D = CanvasToViewport(x, y) color = TraceRay(O, D, 1, inf)
        canvas.PutPixel(x, y, color)
    }
}
```

Функции `CanvasToViewport` из листинга 2.3 очень просты. Константа `d` — это расстояние между камерой и плоскостью проекции.

Листинг 2.3. Функция CanvasToViewport

```
CanvasToViewport(x, y) {
    return (x*Vw/Cw, y*Vh/Ch, d)
}
```

Метод `TraceRay` (листинг 2.4) вычисляет пересечения луча с каждой сферой и возвращает ее цвет в ближайшем пересечении в запрашиваемом диапазоне *t*.

Листинг 2.4. Метод TraceRay

```
TraceRay(O, D, t_min, t_max) {
    closest_t = inf
    closest_sphere = NULL
    for sphere in scene.spheres {
        t1, t2 = IntersectRaySphere(O, D, sphere)
        if t1 in [t_min, t_max] and t1 < closest_t {
            closest_t = t1
            closest_sphere = sphere
        }
        if t2 in [t_min, t_max] and t2 < closest_t {
            closest_t = t2
            closest_sphere = sphere
        }
    }
    if closest_sphere == NULL {
     ❶ return BACKGROUND_COLOR
    }
    return closest_sphere.color
}
```

В листинге 2.4 `O` — это исходная точка луча. Несмотря на то что мы трассируем его от камеры, размещенной в исходной точке, дальше это не обязательно будет так, значит, он должен быть параметром. То же касается `t_min` и `t_max`.

Обратите внимание, что, когда луч не пересекает никакую сферу, нам все равно нужно возвращать *какой-то* цвет ❶, — я выбрал белый.

В завершение `IntersectRaySphere` (листинг 2.5) просто решает квадратичное уравнение.

Листинг 2.5. Метод IntersectRaySphere

```
IntersectRaySphere(O, D, sphere) {
    r = sphere.radius
    CO = O - sphere.center

    a = dot(D, D)
    b = 2*dot(CO, D)
    c = dot(CO, CO) - r*r

    discriminant = b*b - 4*a*c
    if discriminant < 0 {
        return inf, inf
    }

    t1 = (-b + sqrt(discriminant)) / (2*a)
    t2 = (-b - sqrt(discriminant)) / (2*a)
    return t1, t2
}
```

Чтобы применить все это на практике, определим очень простую сцену (рис. 2.10).

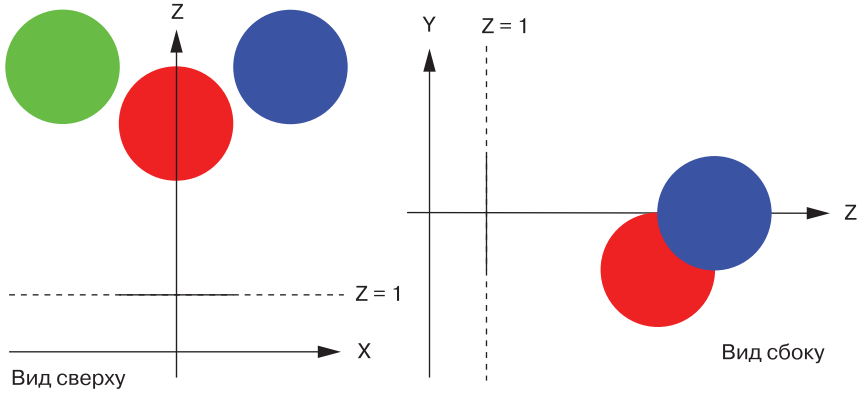


Рис. 2.10. Очень простая сцена, вид сверху *(слева)* и вид сбоку *(справа)*

На языке псевдокода это выглядит так:

```
viewport_size = 1 x 1
projection_plane_d = 1
```

```
sphere {
    center = (0, -1, 3)
    radius = 1
    color = (255, 0, 0) # Красный
}
sphere {
    center = (2, 0, 4)
    radius = 1
    color = (0, 0, 255) # Синий
}
sphere {
    center = (-2, 0, 4)
    radius = 1
    color = (0, 255, 0) # Зеленый
}
```

Выполнив наш алгоритм для этой сцены, мы получаем невероятную картинку на основе трассировки лучей (рис. 2.11).

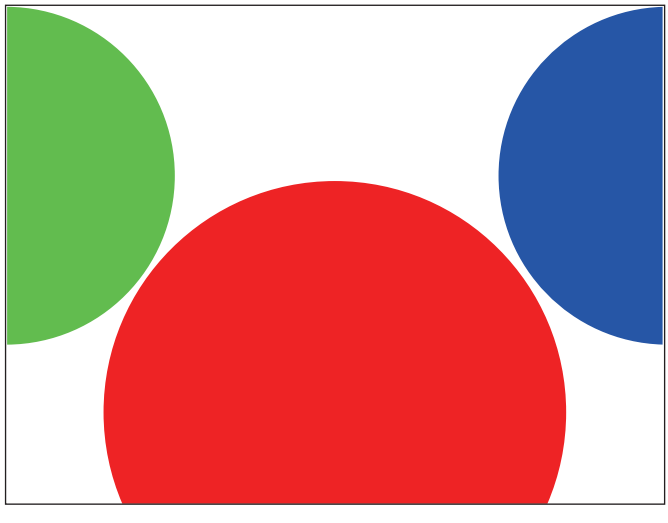


Рис. 2.11. Превосходная сцена на основе трассировки лучей

Реализацию этого алгоритма вы можете найти по адресу https://gabrielgambetta.com/cgfs/basic-rays-demo.

Понимаю, выглядит несколько разочаровывающе. Где здесь отражения, тени и изящный вид? Не беспокойтесь, все это будет. Мы сделали отличный первый шаг. Сферы выглядят как круги, и это лучше, чем если бы они были похожи на кошек. От настоящих сфер их отличает то, что здесь не хватает взаимодействия со светом. Эту тему мы разберем уже в следующей главе.

Итоги главы

Здесь мы заложили основу нашего будущего трассировщика лучей. Мы выбрали фиксированную конфигурацию (позицию и ориентацию камеры и окна просмотра, размер этого окна), представления для сфер и лучей, рассмотрели математическую составляющую для определения взаимодействия лучей со сферами и все это совместили для отрисовки сфер на холсте при помощи сплошных цветов.

В следующих главах на основе этого материала мы будем моделировать взаимодействие света с объектами сцены, повышая уровень детализации.

3

Свет

Начнем с привнесения в наш рендеринг «реализма» за счет добавления в сцену света. Свет — это большая и непростая тема, поэтому для наших целей я ее упрощу. Эта модель построена на принципе работы света в реальном мире, но с ее помощью можно и немного улучшить внешний вид сцен.

Для начала примем ряд упрощающих допущений, которые облегчат нам жизнь, а потом познакомимся с тремя типами света: точечным, направленным и рассеянным. В конце главы мы разберем влияние этих типов освещения на вид объектов, включая диффузное и зеркальное отражение.

Упрощающие допущения

Во-первых, мы объявим весь свет белым. Это позволит нам охарактеризовать любой свет одним вещественным числом i, означающим *интенсивность света*. Имитировать цветной свет несложно (нужно просто использовать три значения интенсивности, по одному для цветового канала, вычисляя весь цвет и освещение поканально). Но, чтобы не усложнять себе жизнь, мы будем придерживаться именно белого света.

Во-вторых, мы проигнорируем атмосферу. В жизни свет при удалении тускнеет — он постепенно поглощается частицами воздуха, через которые проходит. Выполнить это в трассировщике лучей не составит труда, но мы опять же будем придерживаться варианта попроще. В нашей сцене расстояние на яркость света влиять не будет.

Источники света

Свет должен откуда-то исходить. В этом разделе мы определим три типа его возможных источников.

Точечный свет

Точечные источники света испускают его из фиксированной точки в трехмерном пространстве — *позиции*. Свет здесь испускается равномерно во всех направлениях, поэтому такие источники еще называют *всенаправленными*.

Хороший пример из жизни — лампа накаливания, хоть она не всенаправленная и не испускает свет из одной точки.

Пусть вектор $\vec{L}$ будет направлением из точки в сцене P до источника света Q. Мы можем вычислить этот вектор *(световой вектор)* как $Q - P$. Обратите внимание, что Q фиксирована, а P может быть любой точкой в сцене, поэтому $\vec{L}$ будет отличаться для любой из точек сцены, как видно из рис. 3.1.

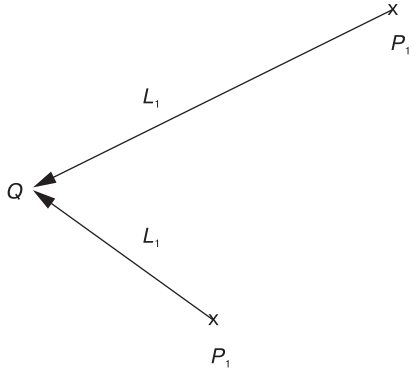


Рис. 3.1. Точечный свет в точке Q. Вектор $\vec{L}$ разный для каждой точки P

Направленный свет

Если точечный свет сопоставим с лампой накаливания, будет ли он так же удачно сопоставим с солнцем? Ответ на этот вопрос зависит от того, что нам нужно отрисовать. В масштабах Солнечной системы Солнце вполне может быть точечным источником света. Оно ведь излучает свет из точки, причем во всех направлениях, значит, по определению подходит.

Но если наша сцена — явление на Земле, тогда Солнце уже не будет удачным примером. Оно слишком далеко от нас, поэтому лучи будут идти из одного направления. Оно может быть точечным источником света, слишком далеким от объектов в сцене. Но расстояние между источником света и этими объектами будет гораздо больше, чем между самими объектами, поэтому возникнет много ошибок точности.

Для грамотной обработки таких ситуаций мы определяем *направленные источники света*. Как и у точечных, у них тоже есть интенсивность, но в отличие от них нет позиции. Им свойственно *фиксированное направление*. Можно рассматривать их как бесконечно удаленные точечные источники, расположенные в заданном направлении.

Если в случае с точечным светом нужно вычислять разный вектор света $\vec{L}$ для каждой точки P в сцене, то здесь $\vec{L}$ дан изначально. В примере с Солнцем и Землей $\vec{L}$ будет *(центр Солнца) – (центр Земли)*. Посмотрите, как это выглядит, на рис. 3.2.

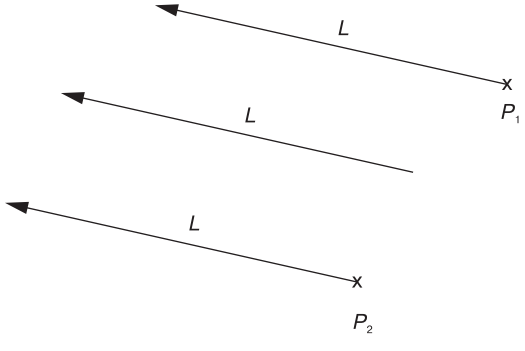


Рис. 3.2. Направленный свет. Вектор $\vec{L}$ одинаков для каждой точки P

Здесь световой вектор направленного света одинаков для каждой точки в сцене. Сравните с рис. 3.1, где световой вектор точечного света, наоборот, для каждой точки в сцене различается.

Рассеянный свет

Можно ли любой реальный свет смоделировать в виде точечного или направленного? Вполне. Будет ли этих двух видов достаточно для освещения сцены? К сожалению, нет.

Рассмотрим Луну. Единственный ближайший источник света у нее — это Солнце. Поэтому одна половина Луны, обращенная к нему, получает весь его свет,

а «тыльная сторона» погружена во тьму. На Земле это явление наблюдается под разными углами и называется фазами Луны.

Ситуация на Земле немного отличается. Даже точки, не получающие прямого света от источника, все равно не погружаются в полную тьму (например, пол под стулом). Как же свет достигает этих точек, если перед ними есть видимое препятствие?

Из раздела «Цветовые модели» в главе 1 мы знаем, что, когда свет достигает объекта, часть излучения поглощается, а оставшаяся — рассеивается обратно в сцену. Но это еще не все. Рассеянный свет будет сталкиваться с очередным объектом, снова частично поглощаться и рассеиваться. Так будет происходить, пока поверхности стены не поглотят всю энергию исходного излучения.

Это значит, что нам нужно рассматривать *каждый объект* как источник света. Как вы понимаете, это сильно усложнит нашу модель, поэтому такой механизм мы в книге изучать не будем. Если же вам любопытно, вбейте в поиск запрос *global illumination* и полюбуйтесь красивыми картинками.

Но нас не устроит прямое освещение объекта или его полное затемнение (если только мы реально не рендерим модель Солнечной системы). Чтобы такого ограничения не было, определим третий тип источника света — *рассеянный свет*. Он характеризуется только интенсивностью. Рассеянный свет привносит часть освещения в каждую точку сцены, независимо от ее расположения. Это сильное упрощение очень сложного взаимодействия между источниками света и поверхностями в сцене, но в нашем случае сработает оно достаточно хорошо.

Как правило, у сцены всего один источник рассеянного света (так как он содержит лишь значение интенсивности, то любое их число можно просто совместить в один) и сколько угодно точечных и направленных источников.

Освещение одной точки

Теперь, когда мы научились определять свет в сцене, нужно разобраться, как он взаимодействует с поверхностями объектов в ней.

Для вычисления освещения одной точки мы узнаем количество света каждого источника и сложим эти значения. В итоге у нас будет единое число — общая величина света, получаемого точкой.

Что происходит, когда любой луч света достигает поверхности объекта сцены?

Можно разделить объекты на два обширных класса в зависимости от их способа отражения света: матовые и глянцевые. Большинство окружающих нас объектов можно классифицировать как матовые, поэтому их мы разберем первыми.

Диффузное отражение

Когда луч света сталкивается с объектом, он рассеивается обратно в сцену равномерно во всех направлениях. Это называется *диффузным отражением*. Именно оно и придает матовым объектам матовость.

Рассмотрите вокруг себя похожие объекты, например стену. Ее цвет не изменится, если вы будете перемещаться. Это значит, что отражаемый от объекта свет, который вы видите, одинаков независимо от точки обзора.

Но количество отраженного света зависит от *угла* между лучом и поверхностью. Причина в том, что переносимая лучом энергия должна распределиться по меньшей или большей площади в зависимости от угла. Поэтому отраженная в сцену энергия на единицу области оказывается большей или меньшей, как показано на рис. 3.3.

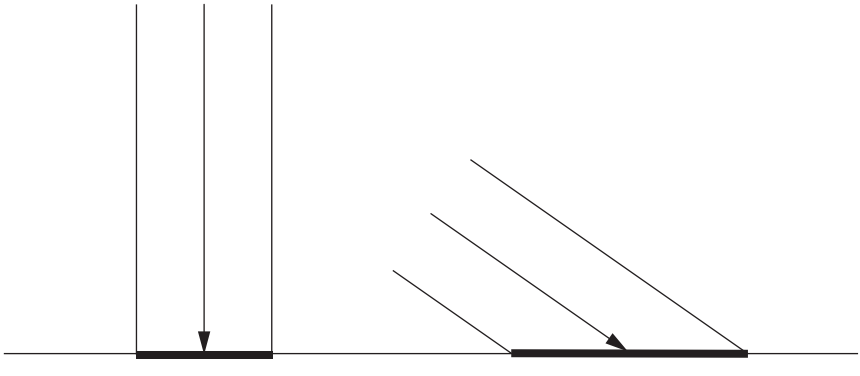


Рис. 3.3. Энергия луча света распределяется по области разной площади в зависимости от угла относительно поверхности

На этом рисунке мы видим два луча света с одинаковой интенсивностью (которая выражена одинаковой шириной), сталкивающихся с поверхностью прямо и под углом. Переносимая этими лучами энергия равномерно распределяется по областям, с которыми они встречаются. Энергия луча справа распределяется по большей площади, чем луча слева. Это значит, что каждая точка в этой области получает меньше энергии, чем в случае с левым лучом.

Чтобы изучить этот момент математически, опишем ориентацию поверхности по ее *вектору нормали*. Вектор нормали поверхности в точке P (или просто «нормаль») перпендикулярен поверхности в точке P. Это также и единичный вектор, то есть он имеет длину 1. Назовем его $\vec{N}$.

Моделирование диффузного отражения

Луч света с направлением $\vec{L}$ и интенсивностью I достигает поверхности с нормалью $\vec{N}$ Какая часть I отразится обратно в сцену в виде функции от I, $\vec{N}$ и $\vec{L}$?

По аналогии с геометрией представим интенсивность света как «ширину» луча. Его энергия распределяется по поверхности размером A. Когда $\vec{N}$ и $\vec{L}$ имеют одинаковое направление, луч перпендикулярен поверхности, тогда $I = A$. То есть энергия, отраженная на единицу области: $\frac{I}{A} = 1$. С другой стороны, по мере того, как угол между $\vec{L}$ и $\vec{N}$ приближается к 90°, A стремится к ∞. Значит, энергия на единицу области стремится к 0: $\lim_{A \to \infty} \frac{I}{A} = 0$. Но что происходит в промежутке?

Эта ситуация отражена на рис. 3.4. Нам известны $\vec{N}$, $\vec{L}$ и P. Для упрощения чертежа я добавил углы α и β наряду с точками Q, R и S.

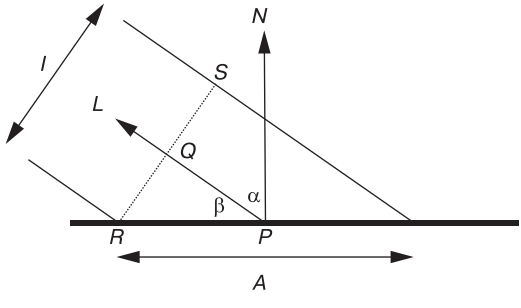


Рис. 3.4. Векторы и углы, задействованные в вычислениях диффузного отражения

Технически у луча света ширины нет, поэтому можем предположить, что все происходит на плоской, бесконечно малой области поверхности. Даже если это поверхность сферы, рассматриваемая область настолько мала, что почти плоская относительно размера всей сферы. Аналогично тому как Земля выглядит плоской при сильном приближении.

Луч света шириной I достигает поверхности в точке P под углом β. Нормаль в P — это $\vec{N}$, и переносимая лучом энергия распределяется по A. Нам нужно вычислить $\frac{I}{A}$.

Рассмотрим RS — «ширину» луча. По определению она перпендикулярна к $\vec{L}$, который является направлением PQ. Поэтому PQ и QR формируют прямой угол, а PQR получается прямоугольным треугольником.

Один из углов PQR = 90°, а другой — β. Оставшийся равен 90 – β. Но обратите внимание, что $\vec{N}$ и PR тоже формируют прямой угол, значит, α + β тоже должны быть равны 90°. Получаем $\widehat{QRP} = \alpha$.

Теперь сосредоточимся на треугольнике *PQR* (рис. 3.5). Его углы — α, β и 90°. Сторона $QR = \frac{I}{2}$, а сторона $PR = \frac{A}{2}$.

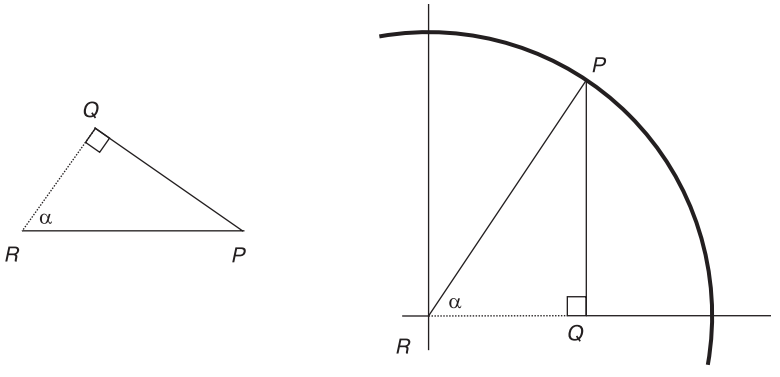


Рис. 3.5. Треугольник *PQR* в тригонометрическом контексте

Дальше нам поможет тригонометрия. По определению $\cos(\alpha) = \frac{QR}{PR}$; заменяя *PQ* на $\frac{I}{2}$, а *PR* на $\frac{A}{2}$, получаем:

$$\cos(\alpha) = \frac{I/2}{A/2},$$

что становится:

$$\cos(\alpha) = \frac{I}{A}.$$

Почти готово. α — это угол между $\vec{N}$ и $\vec{L}$. Мы можем использовать свойства скалярного произведения (не стесняйтесь заглянуть в приложение «Линейная алгебра»), чтобы выразить cos(α) как:

$$\cos(\alpha) = \frac{\langle \vec{N}, \vec{L} \rangle}{|\vec{N}||\vec{L}|}.$$

И в завершение:

$$\frac{I}{A} = \frac{\langle \vec{N}, \vec{L} \rangle}{|\vec{N}||\vec{L}|}.$$

Мы получили простое уравнение, дающее нам отражаемую долю света в виде функции угла между нормалью поверхности и направлением света.

Заметьте, что значение cos(α) для углов больше 90° становится отрицательным. Если мы будем слепо использовать его, то можем прийти к источнику света, который делает поверхность темнее. С позиции физики это бессмыслица. Угол больше 90° просто означает, что свет освещает тыльную сторону поверхности. Поэтому отрицательный cos(α) нужно рассматривать как 0.

Уравнение диффузного отражения

Теперь можно сформулировать уравнение для вычисления общего количества света, получаемого точкой P с нормалью $\vec{N}$ в сцене с рассеянным светом с интенсивностью I_A и n точечных или направленных источников с интенсивностью I_n, а также световыми векторами $\overrightarrow{L_n}$, которые или известны (для направленных источников), или вычисляются для P (для точечных):

$$I_P = I_A + \sum_{i=1}^{n} I_i \frac{\left\langle \vec{N}, \overrightarrow{L_i} \right\rangle}{\left|\vec{N}\right|\left|\overrightarrow{L_i}\right|}.$$

Повторю, что выражения, где $\left\langle \vec{N}, \overrightarrow{L_i} \right\rangle < 0$, к освещению точки добавляться не должны.

Нормали сферы

Недостает только маленькой детали: откуда берутся нормали? Это сложный вопрос, ответ на который мы получим во второй части книги. К счастью, пока мы имеем дело только со сферами, а для них ответ очень прост: вектор нормали любой точки сферы лежит на прямой, проходящей через центр этой сферы. Как видно на рис. 3.6, если C — это центр сферы, направление нормали в точке P будет $P - C$.

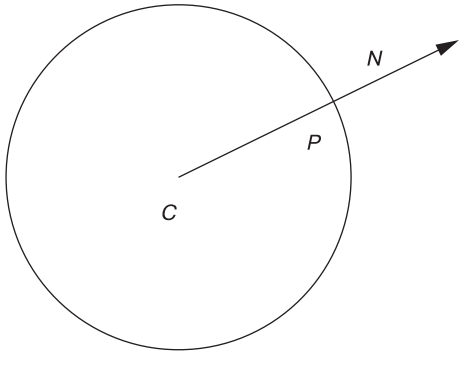


Рис. 3.6. У нормали сферы в точке *P* то же направление, что и у *CP*

Почему «направление нормали», а не «нормаль»? Вектор нормали должен быть перпендикулярен поверхности и иметь длину 1. Чтобы *нормализовать* его в истинный формат, нужно поделить этот вектор на его длину, гарантированно получив результат с длиной 1:

$$\vec{N} = \frac{P - C}{\left|P - C\right|}.$$

Рендеринг с диффузным отражением

Пора перевести все это в псевдокод. Для начала добавим в сцену пару источников света:

```
light {
    type = ambient
    intensity = 0.2
}
light {
    type = point
    intensity = 0.6
    position = (2, 1, 0)
}
light {
    type = directional
    intensity = 0.2
    direction = (1, 4, 4)
}
```

Обратите внимание, что значения интенсивности удобно суммируются в 1,0. Принцип работы уравнения освещенности гарантирует, что никакая точка не имеет интенсивность больше этого значения. В результате мы не будем получать «переэкспонированные» пятна.

Уравнение освещенности перевести в псевдокод достаточно просто (листинг 3.1).

Листинг 3.1. Функция для вычисления освещения с диффузным отражением

```
ComputeLighting(P, N) {
    i = 0.0
    for light in scene.Lights {
        if light.type == ambient {
         ❶ i += light.intensity
        } else {
            if light.type == point {
             ❷ L = light.position - P
            } else {
             ❸ L = light.direction
            }
            n_dot_l = dot(N, L)
         ❹ if n_dot_l > 0 {
             ❺ i += light.intensity * n_dot_l/(length(N) * length(L))
            }
        }
    }
    return i
}
```

В листинге 3.1 мы рассматриваем три типа освещения по-разному. Рассеянный свет самый простой и обрабатывается напрямую ❶. Точечные и направленные источники

делят большую часть кода, в частности вычисление интенсивности ❺. Но векторы направлений вычисляются иначе (❷ и ❸), в зависимости от типа. Условие ❹ гарантирует, что мы не добавляем отрицательные значения.

Осталось только использовать `ComputeLighting` в `TraceRay`. Мы замещаем строку, возвращающую цвет сферы:

```
return closest_sphere.color
```

этим фрагментом:

```
P = O + closest_t * D // Вычисляем пересечение
N = P - closest_sphere.center // Вычисляем нормаль сферы
                              // в пересечении N = N / length(N)
return closest_sphere.color * ComputeLighting(P, N)
```

Ну и ради забавы добавим большую желтую сферу:

```
sphere {
    color = (255, 255, 0) # Желтый
    center = (0, -5001, 0)
    radius = 5000
}
```

Запускаем рендерер, и теперь сферы стали похожи на сферы (рис. 3.7)!

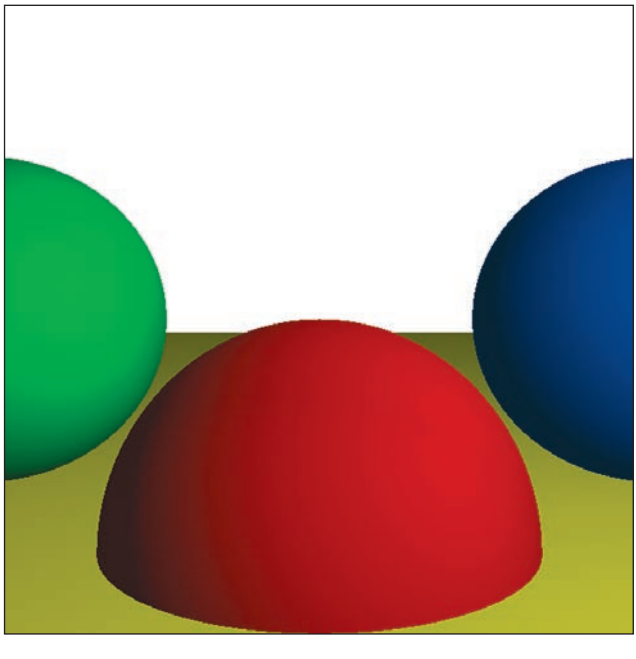


Рис. 3.7. Диффузное отражение добавляет в сцену ощущение глубины и объема

Живую реализацию этого алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/diffuse-demo.

Так, стоп. Как получилось, что большая желтая сфера стала плоским желтым полом? Она не стала. Просто она гораздо больше других сфер, а камера слишком близко. Наша планета тоже кажется плоской, когда мы смотрим себе под ноги.

Зеркальное отражение

Теперь переключим внимание на *глянцевые* объекты. Их вид меняется при смещении точки обзора.

Представьте себе бильярдный шар или машину, только что выехавшую с мойки. Такие вещи обычно отражают на себе яркие пятна, которые будто бы перемещаются при вашем движении вокруг них. В отличие от матовых, видимость нами этих объектов определяется точкой обзора.

Если обойти красный бильярдный шар вокруг, то он останется красным, но смещается яркое белое пятно, придающее ему блеск. Значит, новый эффект, который мы хотим смоделировать, не замещает диффузное отражение, а дополняет его.

Теперь нужно понять причину. Для этого мы внимательно рассмотрим принцип отражения поверхностями света. Из предыдущего раздела мы знаем, что при достижении лучом поверхности матового объекта он рассеивается обратно в сцену равномерно во всех направлениях. Это происходит, потому что поверхность объекта не идеально гладкая и при детальном рассмотрении выглядит как много мельчайших поверхностей, направленных в разные стороны (рис. 3.8).

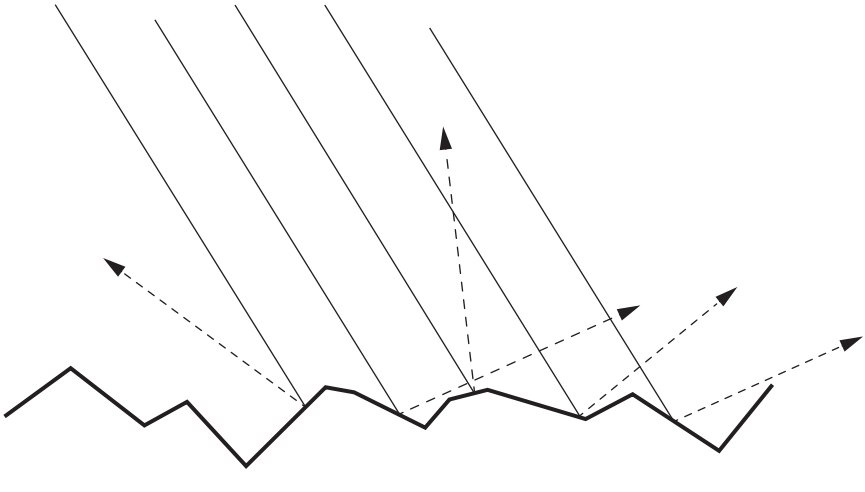


Рис. 3.8. Так может выглядеть шероховатая поверхность матового объекта под микроскопом. Падающие лучи света отражаются произвольно

Но что, если поверхность не будет такой шероховатой? Рассмотрим другую крайность: идеально отполированное зеркало. При столкновении с ним луч отражается в одном направлении. Если назвать направление отраженного света $\vec{R}$ и сохранить условие, что $\vec{L}$ направлен в сторону источника света, то получится ситуация как на рис. 3.9.

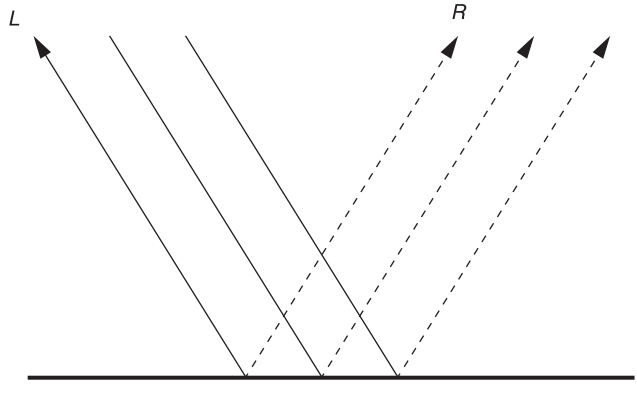


Рис. 3.9. Лучи света, отраженные зеркалом

Гладкая отполированная поверхность в той или иной степени проявляет свойства зеркала. Именно поэтому отражение называется *зеркальным* (specular), от лат. *speculum* — «зеркало».

От идеально отполированного зеркала падающий луч света $\vec{L}$ отражается в одном направлении $\vec{R}$. Поэтому мы и видим отраженные объекты очень ясно: для каждого падающего луча света $\vec{L}$ есть всего один отраженный луч $\vec{R}$. Но не все объекты отполированы идеально. Большинство света отражается в направлении $\vec{R}$, но часть его отражается в близком к $\vec{R}$ направлении. Чем ближе к $\vec{R}$, тем больше света отражается в этом направлении, как можно видеть на рис. 3.10. Именно степень блеска объекта и определяет, насколько быстро отраженный свет уменьшается по мере отклонения от $\vec{R}$.

Нам нужно определить, сколько света от $\vec{L}$ отражается обратно в направлении нашей точки обзора. Если $\vec{V}$ — это вектор обзора, указывающий из точки P в сторону камеры, а α — угол между $\vec{R}$ и $\vec{V}$, мы получаем рис. 3.11.

При $\alpha = 0°$ весь свет отражается в направлении $\vec{V}$. При $\alpha = 90°$ свет вообще не отражается. Как и в случае с диффузным отражением, нам нужно математическое выражение, позволяющее определить, что происходит при промежуточных значениях α.

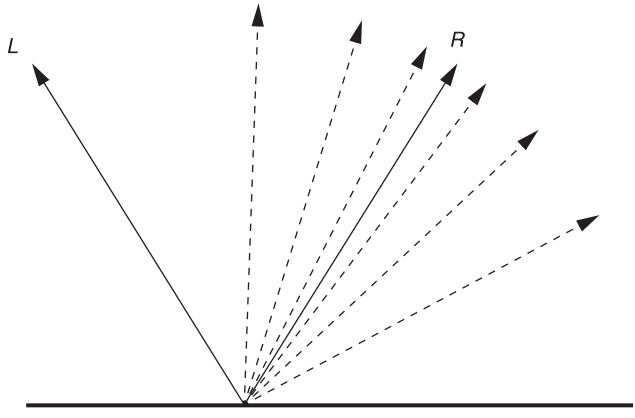


Рис. 3.10. На не идеально полированных поверхностях чем ближе направление к вектору $\vec{R}$, тем больше лучей отражается в этом направлении

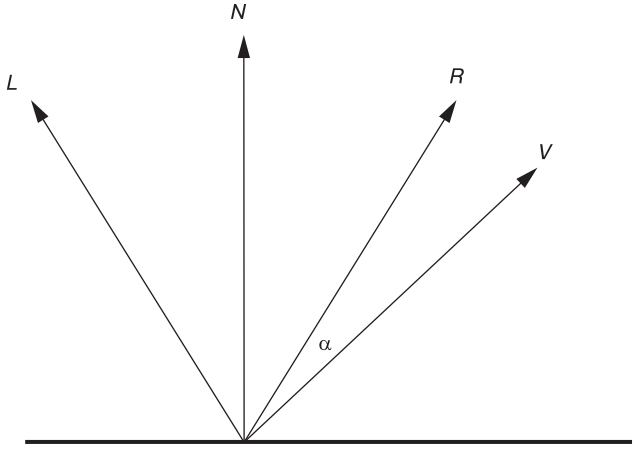


Рис. 3.11. Векторы и углы, задействованные в вычислении зеркального отражения

Моделирование зеркального отражения

В начале этой главы я отметил, что у некоторых моделей нет физических прототипов. Это одна из них. Следующая модель произвольна, но мы ее используем, потому что она проста в вычислении и смотрится симпатично.

Рассмотрим свойства $\cos(\alpha)$: $\cos(0) = 1$ и $\cos(\pm 90) = 0$, как раз то, что нам нужно. При этом значения постепенно уменьшаются от 0 до 90 в виде очень изящной кривой, показанной на рис. 3.12.

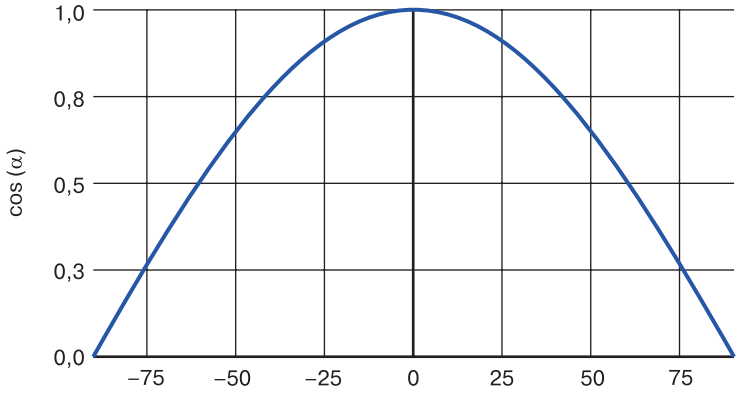


Рис. 3.12. График для cos(α)

Это значит, что cos(α) отвечает всем нашим требованиям к зеркальной функции отражения, так почему бы его не использовать?

Есть еще одна деталь. Если с ходу задействовать эту формулу, то каждый объект окажется одинаково глянцевым. Как же адаптировать уравнение для выражения разных степеней глянца?

Напомню, что глянец (блеск) — это мера того, как быстро уменьшается функция отражения при возрастании α. Простой способ получения разных кривых блеска — возведение cos(α) в положительную степень *s*. $0 \le \cos(\alpha) \le 1$, поэтому мы гарантированно получаем $0 \le \cos(\alpha)^s \le 1$. Следовательно, $\cos(\alpha)^s$ подобен cos(α), только «уже» его. На рис. 3.13 показан график для $\cos(\alpha)^s$ при разных значениях *s*.

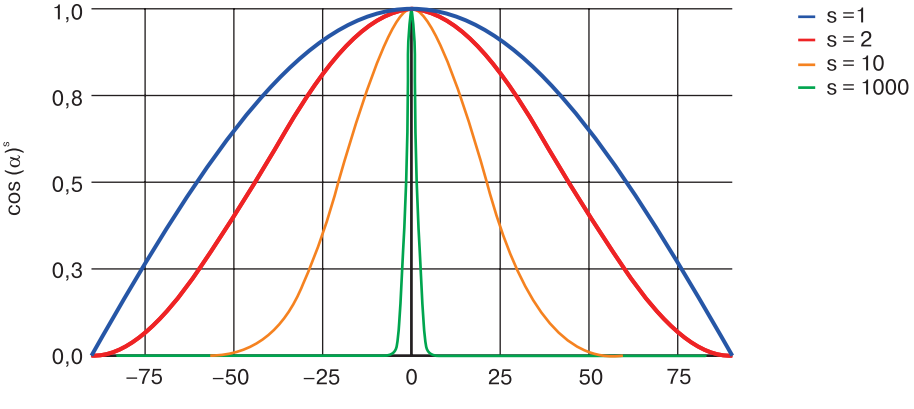


Рис. 3.13. График для $\cos(\alpha)^s$

Чем выше значение *s*, тем «уже» становится функция около 0 и ярче выглядит объект; s называется *зеркальной экспонентой* и относится к свойствам поверхности. Эта модель не основана на физической действительности, поэтому значения *s* можно определить только путем проб и ошибок — по сути, подбирая их, пока они не будут выглядеть «правильно». В случае же с моделью на основе физики можно заглянуть в двухлучевые функции отражательной способности (BDRF).

Теперь объединим все это. Луч света, исходящий из направления $\vec{L}$, сталкивается с поверхностью с зеркальной экспонентой *s* в точке *P*, где нормаль поверхности — это $\vec{N}$. Сколько света отразится в направлении наблюдателя $\vec{V}$?

Согласно нашей модели, это значение $\cos(\alpha)^s$, где α — угол между $\vec{V}$ и $\vec{R}$; $\vec{R}$ — это $\vec{L}$, отраженный по отношению к $\vec{N}$. Поэтому сперва нам нужно вычислить $\vec{R}$ из $\vec{N}$ и $\vec{L}$.

Можно разделить $\vec{L}$ на два вектора ($\overrightarrow{L_P}$ и $\overrightarrow{L_N}$) так, чтобы $\vec{L} = \overrightarrow{L_N} + \overrightarrow{L_P}$, где $\overrightarrow{L_N}$ параллелен $\vec{N}$, а $\overrightarrow{L_P}$ перпендикулярен $\vec{N}$ (рис. 3.14).

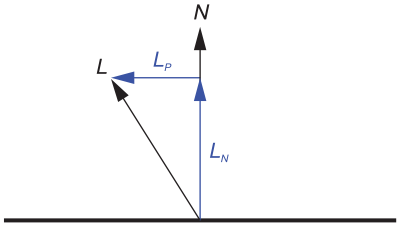


Рис. 3.14. Разделение $\vec{L}$ на его компоненты $\overrightarrow{L_P}$ и $\overrightarrow{L_N}$

$\overrightarrow{L_N}$ — проекция $\vec{L}$ на $\vec{N}$. Исходя из свойств скалярного произведения и того, что $|\vec{N}| = 1$, длина этой проекции равна $\langle \vec{N}, \vec{L} \rangle$. Мы определили L_N параллельным к $\vec{N}$, значит, $\overrightarrow{L_N} = \vec{N} \langle \vec{N}, \vec{L} \rangle$.

Поскольку $\vec{L} = \overrightarrow{L_P} + \overrightarrow{L_N}$, можно сразу получить $\overrightarrow{L_P} = \vec{L} - \overrightarrow{L_N} = \vec{L} - \vec{N} \langle \vec{N}, \vec{L} \rangle$.

Теперь рассмотрим $\vec{R}$. Он симметричен к $\vec{L}$ относительно $\vec{N}$, поэтому его параллельный $\vec{N}$ компонент совпадает с компонентом $\vec{L}$, а его перпендикулярный компонент противоположен компоненту $\vec{L}$. То есть $\vec{R} = \overrightarrow{L_N} - \overrightarrow{L_P}$. Все это отражено на рис. 3.15.

Подставляя найденные выше выражения, получаем:

$$\vec{R} = \vec{N} \langle \vec{N}, \vec{L} \rangle - \vec{L} + \vec{N} \langle \vec{N}, \vec{L} \rangle.$$

А после небольшого упрощения:

$$\vec{R} = 2\vec{N} \langle \vec{N}, \vec{L} \rangle - \vec{L}.$$

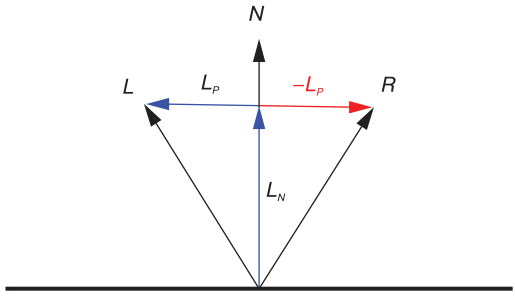


Рис. 3.15. Вычисление $\vec{L}$

Выражение зеркального отражения

Теперь можно составить уравнение для зеркального отражения:

$$\vec{R} = 2\vec{N}\left\langle \vec{N}, \vec{L} \right\rangle - \vec{L};$$

$$I_S = I_L \left(\frac{\left\langle \vec{R}, \vec{V} \right\rangle}{\left|\vec{R}\right|\left|\vec{V}\right|} \right)^s.$$

Как и в случае с диффузным светом, cos(α) может оказаться отрицательным, и нам, как и раньше, нужно это игнорировать. Кроме того, не каждый объект должен быть глянцевым. Для матовых выражение зеркальности вообще не должно вычисляться. Мы отметим это в сцене установкой зеркальной экспоненты на –1 и так обработаем объекты.

Уравнение полного освещения

Мы можем добавить выражение зеркального отражения в создаваемое нами уравнение освещенности и получить единую формулу, описывающую освещение в точке:

$$I_P = I_A + \sum_{i=1}^{n} I_i \left[\frac{\left\langle \vec{N}, \vec{L_i} \right\rangle}{\left|\vec{N}\right|\left|\vec{L_i}\right|} + \left(\frac{\left\langle \vec{R_i}, \vec{V} \right\rangle}{\left|\vec{R_i}\right|\left|\vec{V}\right|} \right)^s \right],$$

где I_P — полное освещение в точке P, I_A — интенсивность рассеянного света, N — нормаль поверхности в P, V — вектор от P к камере, s — зеркальная экспонента поверхности, I_i — интенсивность потока света i, L_i — вектор из P к свету i, а R_i — вектор отражения в P для потока света i.

Рендеринг с зеркальными отражениями

Теперь добавим зеркальные отражения в нашу сцену. Для начала кое-что в ней изменим:

```
sphere {
    center = (0, -1, 3)
    radius = 1
    color = (255, 0, 0) # Красный
    specular = 500 # Глянцевый
}
sphere {
    center = (2, 0, 4)
    radius = 1
    color = (0, 0, 255) # Синий
    specular = 500 # Глянцевый
}
sphere {
    center = (-2, 0, 4)
    radius = 1
    color = (0, 255, 0) # Зеленый
    specular = 10 # Немного глянцевый
}
sphere {
    center = (0, -5001, 0)
    radius = 5000
    color = (255, 255, 0) # Желтый
    specular = 1000 # Сильно глянцевый
}
```

Это та же сцена, что и ранее, но с добавлением в определения сфер зеркальных экспонент.

На уровне кода нам нужно изменить `ComputeLighting` для вычисления выражения зеркальности в случае необходимости и добавления его к полному свету. Обратите внимание, что этой функции теперь требуются $\vec{V}$ и s, как видно из листинга 3.2.

Листинг 3.2. ComputeLighting, поддерживающая диффузное и зеркальное отражения

```
ComputeLighting(P, N, V, s) {
    i = 0.0
    for light in scene.Lights {
        if light.type == ambient {
            i += light.intensity
        } else {
            if light.type == point {
                L = light.position - P
            } else {
                L = light.direction
            }
```

```
            // Диффузное
            n_dot_l = dot(N, L) if n_dot_l > 0 {
                i += light.intensity * n_dot_l/(length(N) * length(L))
            }

            // Зеркальное
        ❶   if s != -1 {
                R = 2 * N * dot(N, L) - L
                r_dot_v = dot(R, V)
          ❷     if r_dot_v > 0 {
                    i += light.intensity * pow(r_dot_v/(length(R) * length(V)), s)
                }
            }
        }
    }
    return i
}
```

Большую часть кода мы не трогаем. Но добавляем фрагмент для обработки зеркальных отражений, чтобы он применялся только к глянцевым объектам ❶, и убеждаемся в том, что не добавляем отрицательную интенсивность света ❷, как делали для диффузного отражения.

В конце нужно изменить `TraceRay` для передачи новых параметров в `ComputeLighting`. С s все просто: она берется прямо из определения сцены. Но откуда берется $\vec{V}$? Это вектор, направленный от объекта к камере. К счастью, в `TraceRay` у нас уже есть вектор, направленный от камеры к объекту, — это $\vec{D}$, направление трассируемого луча. Значит, $\vec{V}$ — это просто $-\vec{D}$.

В листинге 3.3 дается новый вариант `TraceRay`, уже с зеркальным отражением.

Листинг 3.3. TraceRay с зеркальным отражением

```
TraceRay(O, D, t_min, t_max) {
    closest_t = inf
    closest_sphere = NULL
    for sphere in scene.Spheres {
        t1, t2 = IntersectRaySphere(O, D, sphere)
        if t1 in [t_min, t_max] and t1 < closest_t {
            closest_t = t1
            closest_sphere = sphere
        }
        if t2 in [t_min, t_max] and t2 < closest_t {
            closest_t = t2
            closest_sphere = sphere
        }
    }
    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }
```

```
    P = O + closest_t * D // Вычисляем пересечение
    N = P - closest_sphere.center
    // Вычисляем нормаль сферы в пересечении N = N / length(N)
  ❶ return closest_sphere.color * ComputeLighting(P, N, -D, closest_sphere.specular)
}
```

Вычисление цвета 1 сложнее, чем кажется. Напомню, что цвета нужно умножать поканально и результаты вписывать в диапазон канала (в нашем случае 0–255). В этом примере сцены значения интенсивности света суммируются в 1,0. Но после добавления зеркальных отражений эти значения могут выйти из данного диапазона.

Вознаграждение за все это жонглирование векторами вы видите на рис. 3.16.

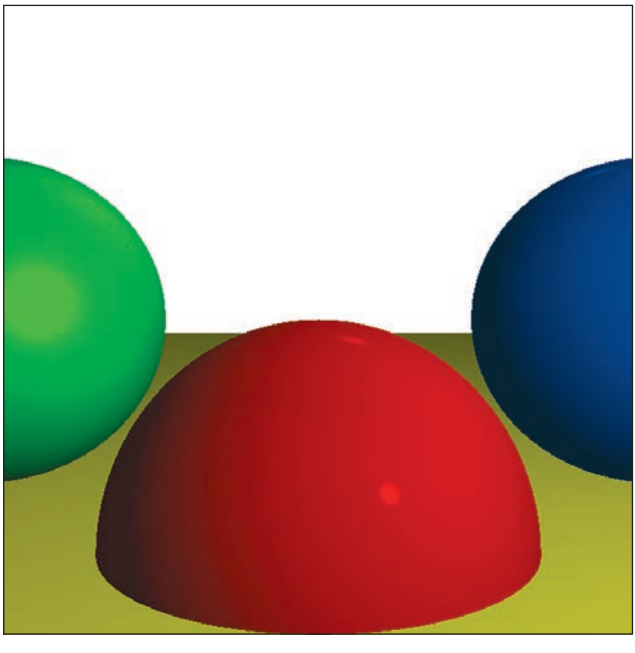


Рис. 3.16. Сцена с рассеянным, диффузным и зеркальным отражениями. В дополнение к глубине и объему здесь у каждой поверхности появились свои отличия

Живую реализацию этого алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/specular-demo.

Обратите внимание, что на рис. 3.16 красная сфера с зеркальной экспонентой, равной 500, отражает на себе более яркое пятно блеска, чем зеленая со значением зеркальной экспоненты 10. Так происходит из-за определенного кадрирования изображения и размещения источников света в сцене. У левой половины зеркальной сферы все еще нет зеркального отражения.

Итоги главы

В этой главе мы взяли очень простой трассировщик лучей из предыдущей и наделили его способностью моделировать источники света, а также определили способ их взаимодействия с объектами сцены.

Есть три типа источников света: точечные, направленные и рассеянные. Вы узнали, как с их помощью представлять разный тип освещения из реальной жизни, и научились описывать их в определении сцены.

После мы перешли к рассмотрению поверхности объектов сцены, разделив их на матовые и глянцевые. Используя принцип взаимодействия лучей света с ними, мы разработали две модели — диффузного и зеркального отражения — для вычисления количества света, отражаемого ими в камеру.

Конечный результат рендеринга сцены оказался намного реалистичнее: теперь мы не просто видим контуры объектов, но и получаем ощущение их глубины, объема и материала, из которого они сделаны. И все же здесь не хватает основного аспекта освещения — теней. О них мы и поговорим в следующей главе.

4

Тени и отражения

Наш квест по реалистичному рендерингу сцены продолжается. В прошлой главе мы смоделировали способ взаимодействия лучей света с поверхностями. В этой главе мы смоделируем два аспекта процесса взаимодействия света со сценой: объекты, отбрасывающие тени и отражающиеся в других объектах.

Тени

Там, где присутствуют свет и объекты, есть и тени. Все это у нас есть, так где же тени?

Принцип формирования теней

Начнем с основного вопроса: откуда вообще берутся тени? Они *возникают*, когда лучи света не могут достичь объекта из-за встреченного на пути препятствия.

Ранее мы рассмотрели только локальные взаимодействия между источником света и поверхностью, проигнорировав все остальные процессы сцены. Чтобы возникли тени, нужно посмотреть глобальнее и проанализировать взаимодействие между источником света, желаемой поверхностью и другими объектами сцены.

Воплотить это несложно. Только следует помнить, что «если между точкой и источником света есть объект, то освещение, исходящее от этого источника, добавлять не нужно».

На рис. 4.1 есть два случая, которые нам необходимо будет различать.

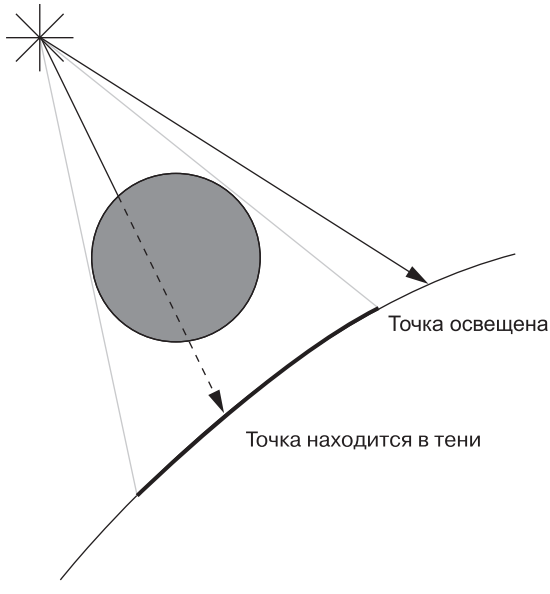


Рис. 4.1. Если между точкой и источником света есть объект, тень всегда будет отбрасываться

У нас уже есть все нужные инструменты. Начнем с направленного света. Нам известна P — интересующая нас точка. Нам известен $\vec{L}$ — часть определения света. Зная P и $\vec{L}$, можно определить луч $(O+t\vec{L})$, проходящий из этой точки поверхности к бесконечно удаленному источнику света. Этот луч с чем-то пересекается? Если ответ отрицательный, то между точкой и источником света ничего нет. Поэтому мы вычисляем освещение от этого источника, как и раньше. Если же пересекается, то точка будет в тени и освещение от этого источника нужно игнорировать.

Мы уже умеем вычислять ближайшее пересечение между лучом и сферой. Для этого у нас есть `TraceRay`, с помощью которой мы трассируем исходящие из камеры лучи. Можно повторно использовать большую ее часть для вычисления ближайшего пересечения луча света с остальной частью сцены.

Хотя при этом параметры функции будут немного другие.

- Теперь луч начинается не от камеры, а от точки P.
- Направление луча не $(V - O)$, а $\vec{L}$.

- Мы не хотим, чтобы объекты *позади P* отбрасывали тени на эту точку, значит, нам нужно $t_{min} = 0$.
- Мы имеем дело с бесконечно удаленными направленными источниками света, поэтому даже очень далекий объект должен все равно отбрасывать тень на P. Значит, $t_{max} = +\infty$.

На рис. 4.2 есть две точки: P_0 и P_1. При трассировке луча, исходящего из P_0 к источнику света, пересечений с объектами нет. Значит, свет может достичь P_0 и тени на ней не будет. Если же с P_1 между лучом и сферой мы находим два пересечения с $t > 0$ (пересечение находится между поверхностью и источником света), точка находится в тени.

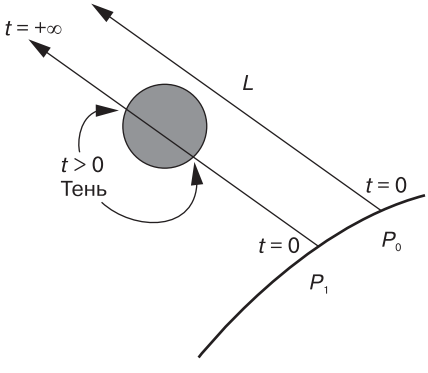


Рис. 4.2. Сфера отбрасывает тень на P_1, но не на P_0

Точечные источники можно рассматривать так же, но есть два исключения. Во-первых, $\vec{L}$ непостоянен, но мы уже умеем вычислять его из P и позиции света. Во-вторых, мы не хотим, чтобы объекты дальше источника света смогли отбрасывать тени на P. Значит, в этом случае нам нужно $t_{max} = 1$, чтобы при достижении источника света луч останавливался.

На рис. 4.3 есть все эти ситуации. Когда мы испускаем луч из P_0 к L_0, то находим пересечения с небольшой сферой. Но для них $t > 1$, то есть они не находятся между источником света и P_0. Значит, мы их игнорируем, поэтому P_0 находится не в тени. С другой стороны, луч из P_1 с направлением L_1 пересекает большую сферу с $0 < t < 1$, и в результате она отбрасывает тень на P_1.

При этом надо учесть буквальный граничный случай. Рассмотрим луч $P + t\vec{L}$. Если мы ищем пересечения начиная с $t_{min} = 0$, то найдем одно в самой точке P! Мы знаем, что P находится на сфере, значит, для $t = 0$ $P + 0\vec{L} = P$. То есть каждая точка будет отбрасывать тень на саму себя.

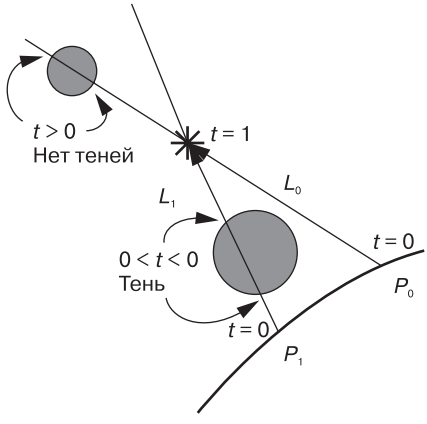


Рис. 4.3. Определяем, отбрасывается ли тень на точку, используя значение t

Простейшее решение — установить t_{min} на очень малое значение вместо 0. С точки зрения геометрии мы говорим, что луч должен начинаться немного над поверхностью, а не из самой *P*. Так диапазон будет [ε, +∞] для направленных источников и [ε, 1] для точечных.

Вы можете захотеть это исправить и не вычислять пересечения между лучом и сферой, которой принадлежит *P*. Для сфер вариант сработает, но не подойдет для объектов с формами посложнее. Например, когда вы прикрываетесь рукой от солнца, то она отбрасывает тень на ваше лицо, обе поверхности в этом случае — часть одного объекта — вашего тела.

Рендеринг с тенями

Теперь переведем все это в псевдокод.

Ранее `TraceRay` сначала вычислял ближайшее пересечение луча со сферой, а потом освещение в нем. Нам нужно извлечь код ближайшего пересечения — он нам понадобится для вычисления теней (листинг 4.1).

Листинг 4.1. Вычисление ближайшего пересечения

```
ClosestIntersection(O, D, t_min, t_max) {
    closest_t = inf
    closest_sphere = NULL
    for sphere in scene.Spheres {
        t1, t2 = IntersectRaySphere(O, D, sphere)
        if t1 in [t_min, t_max] and t1 < closest_t {
            closest_t = t1
            closest_sphere = sphere
        }
```

```
        if t2 in [t_min, t_max] and t2 < closest_t {
            closest_t = t2
            closest_sphere = sphere
        }
    }
    return closest_sphere, closest_t
}
```

Перепишем `TraceRay` для повторного использования этой функции и получим ее упрощенный вариант (листинг 4.2).

Листинг 4.2. Упрощенная версия TraceRay после вычленения ClosestIntersection

```
TraceRay(O, D, t_min, t_max) {
    closest_sphere, closest_t = ClosestIntersection(O, D, t_min, t_max)
    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }
    P = O + closest_t * D
    N = P - closest_sphere.center
    N = N / length(N)
    return closest_sphere.color * ComputeLighting(P, N, -D, closest_sphere.specular)
}
```

Теперь добавим в `ComputeLighting` проверку теней ❶ (листинг 4.3).

Листинг 4.3. ComputeLighting с поддержкой теней

```
ComputeLighting(P, N, V, s) {
    i = 0.0
    for light in scene.Lights {
        if light.type == ambient {
            i += light.intensity
        } else {
            if light.type == point {
                L = light.position - P
                t_max = 1
            } else {
                L = light.direction
                t_max = inf
            }

            // Проверка теней
         ❶ shadow_sphere, shadow_t = ClosestIntersection(P, L, 0.001, t_max)
           if shadow_sphere != NULL {
               continue
           }

            // Диффузное
            n_dot_l = dot(N, L) if n_dot_l > 0 {
                i += light.intensity * n_dot_l / (length(N) * length(L))
            }
```

```
            // Зеркальное
            if s != -1 {
                R = 2 * N * dot(N, L) - L r_dot_v = dot(R, V)
                if r_dot_v > 0 {
                    i += light.intensity * pow(r_dot_v / (length(R) * length(V)), s)
                }
            }
        }
    }
    return i
}
```

На рис. 4.4 показан итоговый вариант нашей сцены.

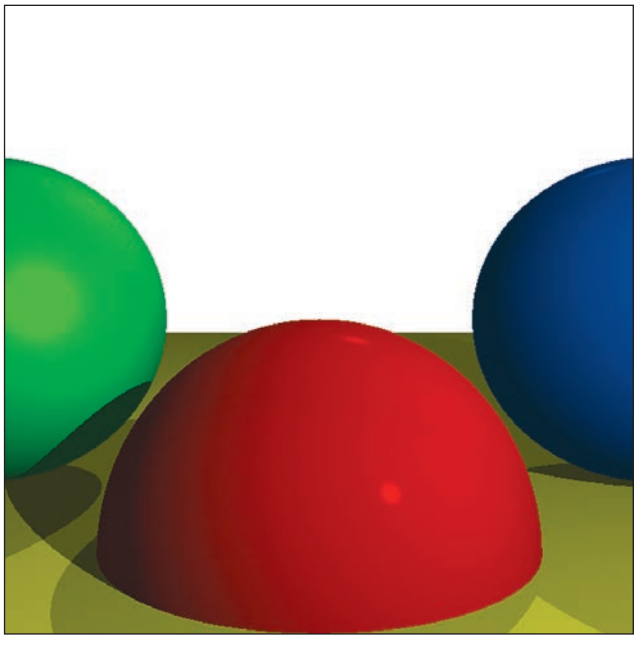


Рис. 4.4. Сцена на основе трассировки лучей, с тенями

Живую реализацию этого алгоритма можно найти по ссылке https://gabrielgambetta.com/cgfs/shadows-demo. В этом демо доступен выбор трассировки из $t = 0$ и из $t = \varepsilon$, чтобы вы смогли понять разницу между этими вариантами.

Вот теперь мы уже кое-чего добились. Объекты в сцене взаимодействуют реалистичнее, отбрасывая друг на друга тени. Дальше мы рассмотрим другие варианты взаимодействия между ними — отражение объектами других объектов.

Отражения

Мы уже говорили о зеркалоподобных поверхностях, но тогда речь шла только о придании им глянца. А можем ли мы использовать объекты, которые будут отражать на своей поверхности другие объекты? Можем, и в трассировщике сделать это будет просто, хотя сначала голова может пойти кругом.

Зеркала и отражения

Разберем принцип работы зеркала. Когда мы смотрим в него, то видим отражаемые им лучи света. Как показано на рис. 4.5, они отражаются симметрично относительно нормали поверхности.

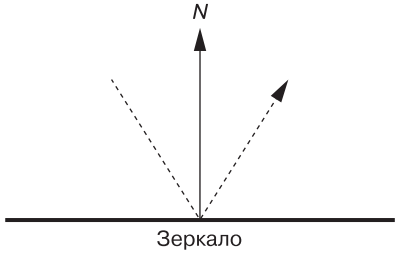


Рис. 4.5. Луч света отражается от зеркала в направлении, симметричном относительно нормали его поверхности

Допустим, мы трассируем луч, и ближайшее пересечение происходит с зеркалом. Какого цвета будет этот луч? Это не цвет самого зеркала, потому что мы наблюдаем отраженный свет. Значит, нужно выяснить, откуда он исходит и каким цветом обладает. Для этого вычислим направление отраженного луча и определим цвет света, идущего из этого направления.

Если бы только у нас была функция, помогающая нам это сделать... Подождите-ка! У нас она есть и имя ей `TraceRay`!

В основном цикле для каждого пикселя мы создаем луч из камеры в сцену и вызываем `TraceRay` для определения цвета, который камера видит в этом направлении. Если `TraceRay` определяет, что камера видит зеркало, то этой функции нужно лишь вычислить направление отраженного луча. Так мы сможем определить цвет света, идущего из этого направления. Причем вызывать `TraceRay` должна... *саму себя*.

Перечитывайте два последних абзаца, пока полностью не поймете их. Если это ваша первая встреча с рекурсивной трассировкой лучей, то прочесть придется

не единожды. Не спешите, я подожду — и как только эйфория от прекрасного момента *«Ага! Понял!»* начнет рассеиваться, мы сформулируем все более четко.

При создании рекурсивного алгоритма (вызывающего самого себя) нам нужно убедиться, что мы не порождаем бесконечный цикл (известный как «Программа не отвечает. Закрыть?»). У этого алгоритма есть два естественных условия выхода: когда луч сталкивается с неотражающим объектом и когда он ни с чем не сталкивается. Но есть простой случай, когда мы можем попасть в западню бесконечного цикла: эффект *бесконечного коридора*. Его можно увидеть, если поставить два зеркала напротив друг друга.

Есть много способов предотвратить это. Но мы просто введем в алгоритм понятие *ограничения рекурсии*, которое поможет нам контролировать, насколько «глубоко» она может уйти. Назовем это ограничение r. Когда $r = 0$, мы видим объекты без отражений. Когда $r = 1$, мы видим объекты и отражения некоторых объектов в них (рис. 4.6).

Рис. 4.6. Отражения некоторых объектов в других объектах ($r = 1$). Мы видим сферы, отраженные от других сфер, но сами они отражений не дают

При $r = 2$ мы видим объекты, отражения некоторых объектов и отражения их отражений. На рис. 4.7 виден результат для $r = 3$. Нет смысла уходить глубже трех уровней, ведь разницы уже почти не будет.

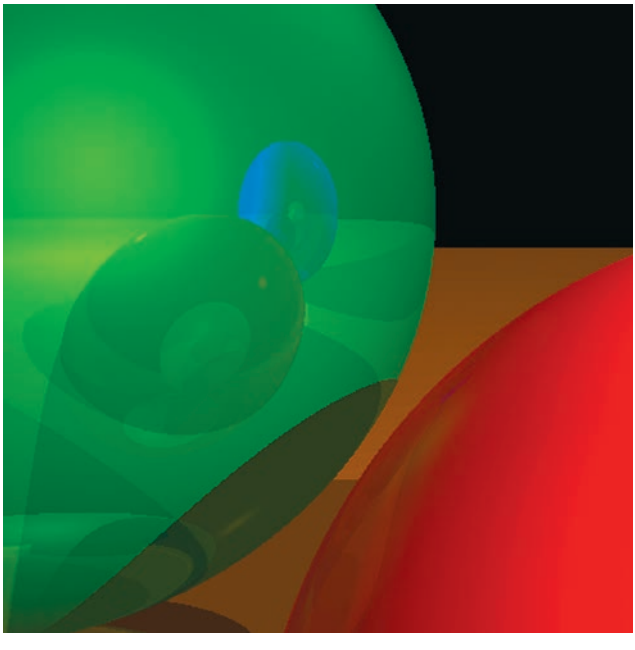


Рис. 4.7. Отражения, ограниченные тремя рекурсивными вызовами ($r = 3$). Теперь мы видим в них отражения отражений сфер

Внесем еще одно отличие: отражаемость не должна определяться как «есть» или «нет» и объекты могут быть только частично отражающими. Мы присвоим каждой поверхности число между 0 и 1 для указания степени ее отражаемости. Потом мы вычислим средневзвешенное локально освещенного и отраженного цвета, используя это число как вес.

Ну и в завершение уточним параметры для рекурсивного вызова `TraceRay`.

- Луч начинается от поверхности объекта в точке P.
- Направление отраженного луча — это направление падающего луча, отраженного от P. В `TraceRay` есть $\vec{D}$, направление падающего луча к P, значит, направление отраженного луча будет $-\vec{D}$, отраженным относительно $\vec{N}$.

- Нам не нужно, чтобы объекты отражали сами себя, поэтому $t_{min} = \varepsilon$.
- Мы хотим видеть отраженный объект независимо от его удаленности, поэтому $t_{max} = +\infty$.
- Ограничение рекурсии будет на 1 меньше его текущей границы (во избежание бесконечной рекурсии).

Теперь можно переводить все это в псевдокод.

Рендеринг с отражениями

Давайте добавим в наш трассировщик отражения. Для начала изменим определение сцены, добавив каждой поверхности свойство `reflective`. Оно будет описывать степень ее отражаемости в диапазоне от 0 (совсем не отражает) до 1 (идеальное зеркало).

```
sphere {
    center = (0, -1, 3)
    radius = 1
    color = (255, 0, 0) # Красная
    specular = 500 # Глянцевая
    reflective = 0.2 # Слегка отражающая
}
sphere {
    center = (-2, 1, 3)
    radius = 1
    color = (0, 0, 255) # Синяя
    specular = 500 # Глянцевая
    reflective = 0.3 # Чуть более отражающая
}
sphere {
    center = (2, 1, 3)
    radius = 1
    color = (0, 255, 0) # Зеленая
    specular = 10 # Немного глянцевая
    reflective = 0.4 # Еще более отражающая
}
sphere {
    color = (255, 255, 0) # Желтая
    center = (0, -5001, 0)
    radius = 5000
    specular = 1000 # Очень глянцевая
    reflective = 0.5 # Полуотражающая
}
```

Мы уже используем формулу отражения луча при вычислении зеркальных отражений, поэтому ее можно исключить. Она получает луч $\vec{R}$ и нормаль $\vec{N}$, возвращая $\vec{R}$, отраженный относительно $\vec{N}$.

```
ReflectRay(R, N) {
    return 2 * N * dot(N, R) - R;
}
```

Единственное, что следует поправить в `ComputeLighting`, — заменить уравнение отражения на вызов `ReflectRay`.

Есть небольшое изменение в основном методе — нужно передавать ограничение рекурсии в высокоуровневый вызов `TraceRay`:

```
color = TraceRay(O, D, 1, inf, recursion_depth)
```

Можно установить начальное значение `recursion_depth` на разумную величину, например 3.

Единственные заметные изменения происходят в конце `TraceRay`, где мы рекурсивно вычисляем отражения. В листинге 4.4 можно увидеть это наглядно.

Листинг 4.4. Псевдокод трассировщика лучей с отражениями

```
TraceRay(O, D, t_min, t_max, recursion_depth) {
    closest_sphere, closest_t = ClosestIntersection(O, D, t_min, t_max)

    if closest_sphere == NULL {
        return BACKGROUND_COLOR
    }

    // Вычисляем локальный цвет
    P = O + closest_t * D
    N = P - closest_sphere.center N = N / length(N)
    local_color = closest_sphere.color * ComputeLighting(P, N, -D,
    closest_sphere.specular)

    // Если достигается граница рекурсии либо объект
    // оказывается неотражающим, процесс завершается
  ❶ r = closest_sphere.reflective
    if recursion_depth <= 0 or r <= 0 {
        return local_color
    }

    // Вычисляем отраженный цвет
    R = ReflectRay(-D, N)
```

```
  ❷ reflected_color = TraceRay(P, R, 0.001, inf, recursion_depth - 1)

  ❸ return local_color * (1 - r) + reflected_color * r
}
```

Изменения в коде очень простые. Сначала выполняется проверка необходимости вычисления отражений ❶. Если сфера оказывается неотражающей или достигается граница рекурсии, мы заканчиваем и просто возвращаем собственный цвет сферы.

Самое интересное изменение в рекурсивном вызове ❷. `TraceRay` вызывает саму себя с нужными параметрами для отражения и уменьшает значение счетчика рекурсий. Вместе с проверкой ❶ это предотвращает появление бесконечного цикла.

После получения локального и отраженного цветов сферы мы их смешиваем ❸, определяя пропорции по «степени отражаемости сферы».

Результаты пусть говорят сами за себя. Смотрим на рис. 4.8.

Живую реализацию этого алгоритма можно найти по адресу: https://gabrielgambetta.com/cgfs/reflections-demo.

Рис. 4.8. Полученная на основе трассировки лучей сцена, теперь с отражениями

Итоги главы

В предыдущих главах мы разработали базовую структуру для рендеринга 3D-сцены на 2D-холсте, смоделировав взаимодействие света с поверхностью объекта. Так мы получили простое начальное представление сцены.

Сейчас мы расширили эту структуру для моделирования взаимодействия разных объектов сцены не только со светом, но и друг с другом, добавив взаимное отбрасывание теней и отражения. Теперь сцена выглядит гораздо реалистичнее.

В следующей главе мы продолжим работу над тем, что начали. И коротко обсудим все, начиная с представления объектов, помимо сфер, и заканчивая практическими вопросами вроде скорости рендеринга.

5

Расширение возможностей трассировщика лучей

В завершение первой части книги рассмотрим несколько интересных тем, которые мы еще не затронули: размещение камеры в любой точке сцены, оптимизацию быстродействия, использование иных фигур, помимо сфер, моделирование объектов с помощью конструктивной блочной геометрии, поддержку прозрачных поверхностей и суперсемплинг (избыточная выборка сглаживания). На практике все это мы не попробуем, но вы можете заняться этим самостоятельно, используя приобретенные здесь знания.

Свободное расположение камеры

В начале рассмотрения трассировки лучей мы сделали три важных допущения: фиксированное положение камеры $(0, 0, 0)$, ее ориентированность в сторону $\vec{Z}_+$ и то, что ее направление «вверх» соответствует $\vec{Y}_+$. Здесь мы устраним эти ограничения, чтобы свободно размещать и направлять камеру.

Начнем с позиции камеры. Вы могли заметить, что во всем псевдокоде *O* используется только единожды: в методе верхнего уровня как отправная точка исходящих из камеры лучей. И чтобы изменить позицию камеры, достаточно просто использовать его другое значение.

Влияет ли это изменение *позиции* на *направление* лучей? Совсем нет. Направление лучей — это вектор от камеры к плоскости проекции. Если камера смещается, то плоскость проекции смещается вместе с ней, а значит, их относительные позиции не меняются. Это согласуется с тем, как мы написали `CanvasToViewport`.

Поговорим об ориентации камеры. Допустим, у вас есть матрица вращения, представляющая нужную ориентацию камеры. При простом вращении камеры

ее *позиция* сохраняется, но изменяется направление, в котором она смотрит. Оно полностью повторяет вращение вслед за камерой. Поэтому если у нас есть направление луча $\vec{D}$ и матрица вращения R, то вращаемый D будет просто $R \times \vec{D}$.

Короче говоря, единственная требующая изменений функция — основная функция в листинге 2.2. Ее обновленная версия приводится в листинге 5.1.

Листинг 5.1. Основной цикл, обновленный для поддержки свободной позиции и ориентации камеры

```
for x in [-Cw/2, Cw/2] {
    for y in [-Ch/2, Ch/2] {
    ❶ D = camera.rotation * CanvasToViewport(x, y)
    ❷ color = TraceRay(camera.position, D, 1, inf)
      canvas.PutPixel(x, y, color)
    }
}
```

Мы применяем матрицу вращения камеры ❶, описывающую ее ориентацию в пространстве, к направлению луча, который собираемся трассировать. Потом используем эту позицию камеры как старую точку луча ❷.

На рис. 5.1 показана сцена при отрисовке с другой позиции и при другой ориентации камеры.

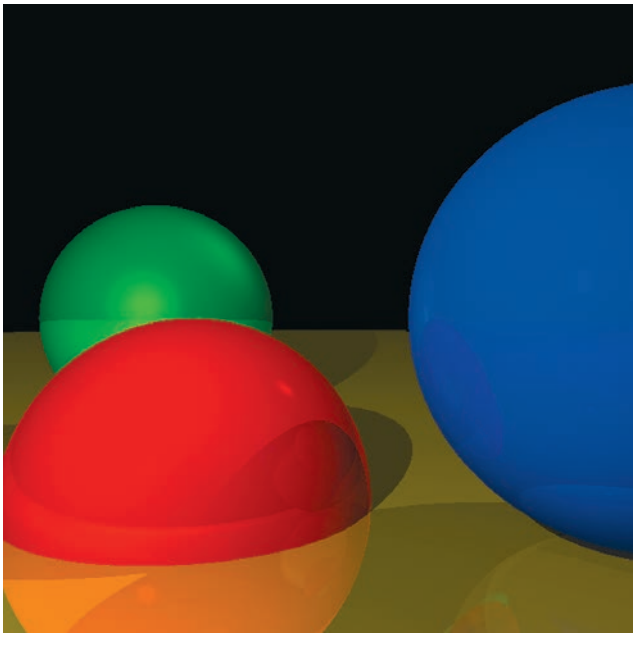


Рис. 5.1. Знакомая сцена, отрисованная при другой позиции и ориентации камеры

Живая реализация этого алгоритма — по адресу https://gabrielgambetta.com/cgfs/camera-demo.

Улучшение производительности

Целью прошлых глав было как можно проще и понятнее объяснить способ работы трассировщика лучей. В итоге он получился функциональным, но медленным. Ниже приведены несколько идей по его ускорению. Просто ради забавы замерьте время выполнения до и после каждой из оптимизаций. Уверен, результаты вас удивят.

Распараллеливание

Самый очевидный способ ускорить трассировщик — это выполнить одновременную трассировку множества лучей. Каждый исходящий из камеры луч независим от других, и данные сцены предполагают только считывание, можно трассировать по одному лучу на ядро ЦПУ без дополнительных издержек или сложностей с синхронизацией. Вообще, в силу своей внутренней природы трассировщики лучей относятся к классу *чрезвычайно параллелизуемых* алгоритмов.

Но порождать поток для каждого луча — не лучшая идея. Это создаст излишнюю нагрузку и перекроет весь прирост скорости. Разумнее будет создать набор «задач». Каждая из них будет отвечать за трассировку части холста (прямоугольной области с точностью до пикселя) и распределять эти задачи по потокам воркеров на физических ядрах по мере их доступности.

Кэширование неизменяемых значений

Кэширование — это способ избежать постоянного повторения одинаковых вычислений. Когда они затратны, а использовать их результат планируется не единожды, лучше сохранить (кэшировать) его значение и потом при надобности использовать уже в готовом виде. Это особенно актуально, когда результат изменяется редко.

Рассмотрим значения, вычисленные в `IntersectionRaySphere`, где трассировщик лучей обычно проводит большую часть времени:

```
a = dot(D, D)
b = 2 * dot(OC, D)
c = dot(OC, OC) - r * r
```

Разные значения остаются неизменными в разные периоды выполнения. Когда мы загрузили сцену и знаем размер сфер, можно вычислить `r * r`. Это значение останется прежним вплоть до изменения размера сферы.

Некоторые значения не меняются в течение как минимум всего кадра. Пример: `dot(OC, OC)`, который изменяется только между кадрами при смещении камеры или сферы (обратите внимание, что тени и отражения трассируют лучи, которые начинаются не в камере. Нужно быть внимательными и убедиться, что кэшированное значение в этом случае не используется).

Некоторые значения неизменны для всего луча. Например, можно вычислить `dot(D, D)` в `ClosestIntersection` и передать его в `IntersectionRaySphere`.

Есть много комбинаций, которые можно задействовать повторно. Проявите фантазию! Хотя не каждое кэшированное значение ускорит процесс целиком. Иногда издержки на его поддержание могут перекрыть выгоду в скорости. Поэтому всегда пользуйтесь бенчмарками для оценки реальной пользы от оптимизации.

Оптимизация теней

Когда точка поверхности находится в тени из-за преграды на пути к свету, вероятность, что соседняя точка тоже будет в тени того же объекта, высока (это называется *совокупностью теней*). Посмотрите пример на рис. 5.2.

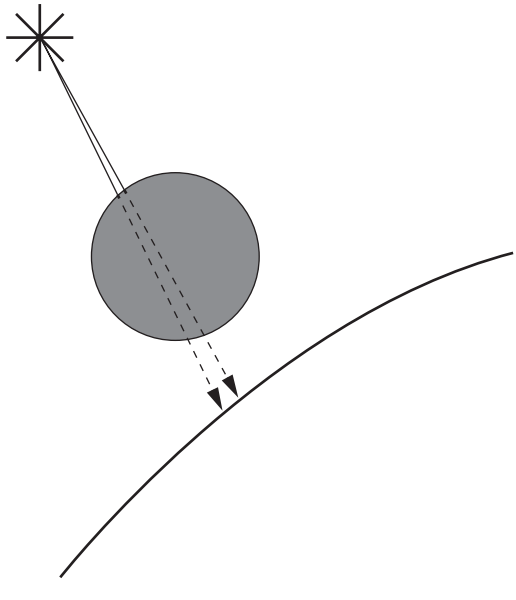


Рис. 5.2. Близкорасположенные точки, вероятнее всего, будут в тени одного объекта

Когда мы ищем объекты между точкой и источником света, чтобы определить, находится ли эта точка в тени, мы проверяем наличие пересечений с остальными

объектами сцены. Но если мы знаем, что соседняя точка в тени какого-то объекта, то сначала можно проверить наличие пересечений именно с ним. Если оно есть, то остальные объекты можно не проверять. Если пересечения нет, мы просто проверяем остальные.

Точно так же при поиске пересечений луча с объектом, чтобы узнать, находится ли точка в тени, не нужно искать *ближайшее* пересечение. Достаточно знать о наличии *хотя бы одного*, так как оно уже будет считаться преградой на пути света к точке. Поэтому можно написать особую версию `ClosestIntersection`, возвращающую результат при первом обнаружении *любого* пересечения. И не нужно вычислять и возвращать `closest_t`, ведь достаточно просто вернуть булево значение.

Иерархические структуры

Очень затратно вычислять пересечение луча с каждой сферой сцены. Есть много разных структур данных, позволяющих отбросить целые группы объектов и избавляющих от необходимости вычисления их пересечений по отдельности.

Представим, что у нас есть несколько близко расположенных сфер. Можно вычислить центр и радиус наименьшей, содержащей их все. Если луч не пересекается с этой *ограничивающей сферой*, можно сделать вывод, что он не пересекает и находящиеся внутри нее, затратив на это всего одну операцию вычисления. Если пересечение будет найдено, придется проверять, пересекает ли луч любую из внутренних сфер.

Можно углубиться и создать несколько уровней ограничивающих сфер (групп, состоящих из других групп сфер), сформировав иерархию. Полностью обойти ее понадобится только в случае пересечения одной из сфер лучом.

Подробное рассмотрение не входит в эту книгу, но вы можете найти интересующую информацию по запросу *«иерархия ограничивающих объемов»*.

Прореживание

Вот простой способ ускорить работу трассировщика лучей в N раз: вычислять в N раз меньше пикселей! Для каждого пикселя холста мы трассируем один луч через окно просмотра, чтобы *сэмплировать* цвет идущего из этого направления света. Если бы лучей у нас было меньше, чем пикселей, это бы означало выполнение *прореживания (subsampling)* сцены. Но как это сделать, не навредив качеству рендеринга?

Представим, что трассируем лучи для пикселей (10, 100) и (12, 100) и эти лучи сталкиваются с одним объектом. Понятно, что луч для пикселя (11, 100) тоже столкнется с этим объектом, значит, можно пропустить начальный поиск пересечений со всеми объектами сцены и сразу перейти к вычислению цвета в этой точке.

Если пропустить каждый второй пиксель по горизонтали и вертикали, то можно сократить количество вычислений пересечения первичных лучей с объектами сцены на 75 % — то есть получить прирост скорости в четыре раза!

Конечно, так можно пропустить очень тонкий объект. Это несовершенная оптимизация. В отличие от рассмотренных выше, она приводит к получению изображения, близкого к его варианту без оптимизации, но не в точности совпадающего с ним. Это можно назвать «жульничеством» через срезание углов. Основная хитрость здесь в понимании того, какие углы можно срезать, чтобы сохранить приемлемое качество. Во многих областях компьютерной графики важна именно субъективная оценка.

Поддержка других примитивов

Ранее мы использовали сферы, потому что уравнения для нахождения пересечений между лучами и сферами несложные. Но, когда у вас уже есть базовый трассировщик, способный отрисовывать сферы, добавить другие примитивы не составит труда.

Обратите внимание, что `TraceRay` должен иметь возможность вычислить для луча и любого объекта всего два элемента: значение t для ближайшего их пересечения и нормаль в нем. Все остальное в трассировщике от объектов не зависит.

Треугольник тоже можно отнести к удобным в рендеринге фигурам. Он простейший из возможных многоугольников — из него можно строить любые многоугольники. Так как математически ими тоже легко управлять, они отлично подходят для представления аппроксимации более сложных поверхностей.

Для добавления поддержки треугольников в трассировщик нужно просто изменить `TraceRay`. Сначала мы вычисляем пересечение между лучом (заданным его источником и направлением) и плоскостью с треугольником (заданным его нормалью и расстоянием от начала координат).

Плоскости бесконечно велики, поэтому лучи почти всегда будут пересекать любую из них (за исключением взаимной параллельности и случая, когда луч удаляется от поверхности). Шаг второй: определить, находится ли пересечение луча и плоскости внутри треугольника. Это можно сделать разными способами, в том числе использовать барицентрические координаты или векторные произведения для проверки того, находится ли точка «внутри» относительно каждой из сторон треугольника.

Если подтверждается, что точка внутри треугольника, значит, нормаль в пересечении — это просто нормаль плоскости. Пусть `TraceRay` вернет соответствующие значения, и больше никаких изменений вносить не нужно.

Конструктивная блочная геометрия

Допустим, нам нужно отрисовать объекты сложнее сфер или изогнутой формы, усложняющей их точное моделирование треугольниками. Примеры таких объектов — линзы (как в увеличительных очках) и Звезда Смерти (замаскированная под Луну...). Их легко описать простым языком. Увеличительная линза выглядит как две обрезанные сферы, склеенные вместе. Звезда Смерти — как сфера, из которой была извлечена часть сферы поменьше.

Это можно выразить формальнее как результат применения операций над множествами (объединение, пересечение или разность) к другим объектам. Линзу можно описать как пересечение двух сфер, а Звезду Смерти — как большую сферу, из которой извлечена часть малой (рис. 5.3).

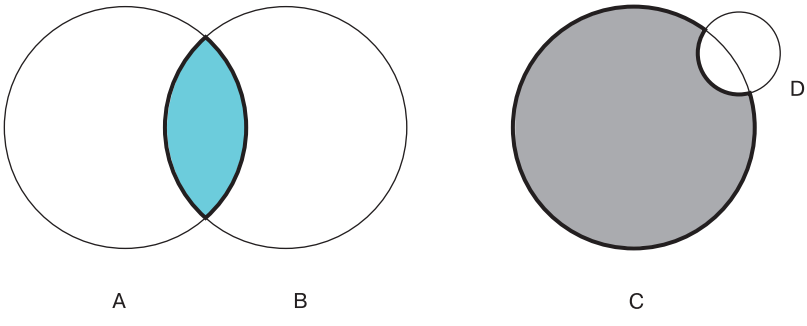


Рис. 5.3. Конструктивная блочная геометрия в действии. $A \cap B$ дает линзу. $C - D$ — Звезду Смерти

Вычисление булевых операций блочных объектов может показаться сложным. Так и есть. К счастью, *конструктивная блочная геометрия* позволяет рендерить результаты операций над множествами между объектами без необходимости вычислять эти результаты явно.

Как можно сделать это в нашем трассировщике? Для каждого объекта можно вычислить точки, в которых луч входит и выходит из него. В примере сферы луч входит в $\min(t_1, t_2)$ и выходит в $\max(t_1, t_2)$. Допустим, вам нужно вычислить пересечение двух сфер. Луч проходит внутри этого пересечения, когда находится внутри *обеих* сфер, а снаружи, когда находится вне *одной из них*. В случае с вычитанием луч проходит внутри, когда находится внутри первого объекта, но не второго. При объединении двух объектов луч проходит внутри, когда находится внутри любого из них.

Проще говоря, если нужно вычислить пересечение между лучом и объектом $A \odot B$ (где $\odot$ — это любая операция над множеством), сначала по отдельности вычисляются пересечения луча с A и луча с B. После этого ближайшее пересечение между

лучом и $A \odot B$ — это наименьшее значение t, которое находится как во «внутреннем» диапазоне объекта, так и между t_{min} и t_{max}. На рис. 5.4 показан внутренний диапазон объединения, пересечения и разницы двух сфер.

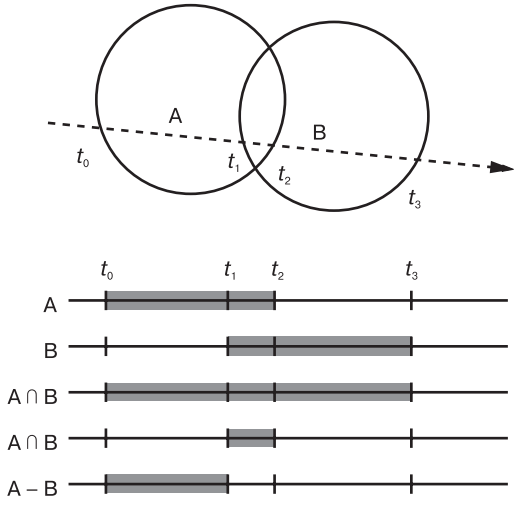


Рис. 5.4. Объединение, пересечение и вычитание двух сфер

Нормаль в этом пересечении — это или нормаль создавшего пересечение объекта, либо ее противоположность. Это зависит от того, смотрим мы «извне» или «изнутри» исходного объекта.

Конечно, A и B не обязаны быть примитивами. Они сами могут быть результатом операций над множествами. Если реализовать все четко, то можно вообще не знать, *чем* являются A и B, пока из них получаются пересечения и нормали. Так можно взять три сферы и вычислить, например, $(A \cup B) \cap C$.

Прозрачность

Раньше мы отрисовывали каждый объект как непрозрачный, но это не всегда должно быть так. Можно рендерить и частично прозрачные объекты, например аквариум.

В этом случае реализация будет похожа на реализацию отражения. Когда луч сталкивается с частично прозрачной поверхностью, мы не только вычисляем локальный и отраженный цвет, но и цвет света, проходящего *через* этот объект. Последний мы получаем через еще один вызов к `TraceRay`. Потом этот цвет смешивается с локальным и отраженным в пропорциях, зависящих от прозрачности объекта. Процесс похож на тот, что мы проделывали при вычислении отражений объектов.

Преломление

В реальности, когда луч проходит через непрозрачный объект, он изменяет направление (поэтому при погружении трубочки в стакан с водой она кажется «сломанной»). Точнее, луч света изменяет направление, когда через одну среду (например, воздух) он попадает в другую (например, воду).

Изменение направления луча определяется свойством каждой среды (материала), называемым *коэффициентом преломления*, по уравнению, называемому законом Снеллиуса:

$$\frac{\sin(\alpha_1)}{\sin(\alpha_2)} = \frac{n_2}{n_1}.$$

Здесь α_1 и α_2 — углы между лучом и нормалью до и после пересечения поверхности, а n_1 и n_2 — коэффициенты преломления материала вне и внутри объектов.

К примеру, $n_{\text{воздуха}}$ равен примерно 1,0, а $n_{\text{воды}}$ — примерно 1,33.

Итак, для каждого луча света, входящего в воду под углом 60°, получаем:

$$\frac{\sin(60)}{\sin(\alpha_2)} = \frac{1{,}33}{1{,}00};$$

$$\sin(\alpha_2) = \frac{\sin(60)}{1{,}33};$$

$$\alpha_2 = \arcsin\left(\frac{\sin(60)}{1{,}33}\right) = 40{,}628°.$$

Этот пример показан на рис. 5.5.

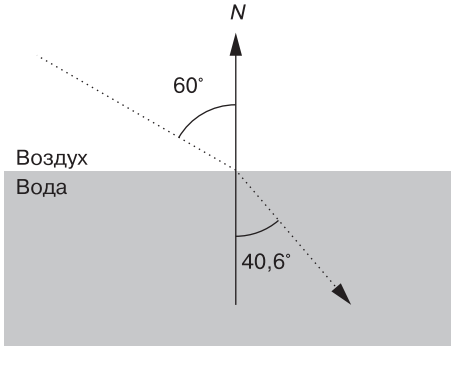


Рис. 5.5. Отклонение луча света при переходе из воздушной среды в водную

На уровне реализации каждый луч содержит дополнительную информацию: коэффициент преломления среды, через которую он проходит. Когда луч пересекает полупрозрачный объект, мы вычисляем его новое направление с этой точки, исходя из коэффициентов преломления текущей и новой среды. После продолжаем движение как обычно.

Давайте рассмотрим вот какой вариант: используя в реализации конструктивную блочную геометрию и прозрачность, вы можете смоделировать увеличительное стекло (пересечение двух сфер), которое физически будет работать как настоящее.

Избыточная выборка

Избыточная выборка (суперсэмплинг) противоположна прореживанию. Здесь наша задача в повышении точности, а не производительности. Допустим, лучи, соответствующие двум смежным пикселям, достигают разных объектов. Тогда каждый пиксель будет закрашиваться подходящим цветом.

Вспомните пример из начала книги: каждый луч должен определять «характерный» цвет каждой *клетки* «сетки», через которую мы смотрим. Используя по одному лучу на пиксель, мы произвольно решаем, что цвет луча света, проходящего через середину клетки, характерен для всей клетки. Это может быть не так.

Решить это можно просто трассировкой большего числа лучей на пиксель — 4, 9, 16 и т. д. — и усреднить их для получения цвета пикселя. Конечно, так трассировщик станет в 4, 9 или 16 раз медленнее по той же причине, по которой прореживание делало его в N раз быстрее. Но есть компромисс. Можно предположить, что свойства объекта плавно изменяются вдоль его поверхности. В результате отправка на пиксель по четыре луча, сталкивающихся с одним и тем же объектом вблизи, не особо улучшит сцену. Поэтому лучше начать с одного луча на пиксель и сравнивать смежные лучи. Если они достигают разных объектов или цвет отличается больше чем на определенный порог, пиксельное деление нужно применять к обоим.

Итоги главы

В этой главе мы ознакомились с некоторыми идеями, которые потом можно изучить самостоятельно. С их помощью можно улучшить разработанный нами базовый трассировщик — повысить его эффективность, научить представлять более сложные объекты или смоделировать лучи света для их большего соответствия физическим законам.

Эта часть книги должна доказать, что трассировщики лучей — отличные представители ПО, способные создавать завораживающие изображения лишь при помощи интуитивно понятных алгоритмов и простой математики.

Печально, но эта чистота требует жертв — производительности. Хоть в этой главе мы и разобрали много способов оптимизации, они все равно слишком затратны для ускорения в реальном времени. Аппаратное обеспечение улучшается с каждым годом, но, несмотря на это, некоторые приложения требуют обработки изображений в 100 раз быстрее и без потери качества. Из них наиболее требовательны видеоигры. Мы ожидаем идеальных изображений, отрисовываемых со скоростью не менее 60 кадров в секунду. Трассировщики лучей на такое просто не способны.

И откуда же тогда такая скорость у игр еще с начала 1990-х? Дело здесь в совершенно другом семействе алгоритмов, которые мы изучим во второй части книги.

ЧАСТЬ II

РАСТЕРИЗАЦИЯ

6

Прямые

В первой части мы рассмотрели трассировку лучей и разработали трассировщик, способный отрисовывать сцену с освещением, свойствами материалов, тенями и отражением, при помощи несложных алгоритмов и математических принципов. Но эта простота дается ценой производительности. И хотя для таких областей, как архитектурные визуализации или визуальные эффекты в кино, скорости выполнения вполне хватает, ее недостает в других областях, например видеоиграх.

В этой части книги мы рассмотрим совершенно другой набор алгоритмов, направленных на производительность, а не на математическую точность.

Наш трассировщик начинает от камеры и изучает сцену через окно просмотра. Для каждого пикселя холста мы отвечаем на вопрос: «Какой объект сцены отсюда видно?» Теперь же мы используем другой подход: для каждого объекта сцены мы будем отвечать на вопрос: «В каких частях холста этот объект будет видим?»

Оказывается, можно разработать алгоритмы, отвечающие на этот вопрос гораздо быстрее, чем трассировщик. Но здесь нужно идти на компромиссы. Позже вы научитесь использовать эти быстрые алгоритмы для достижения результатов, по качеству равных трассировщику лучей.

Здесь мы снова начнем с азов: у нас есть холст размерами C_w на C_h. Мы можем устанавливать цвет отдельного пикселя через `PutPixel()`, но на этом все. Давайте рассмотрим, как можно нарисовать на этом холсте простейший элемент — отрезок между двух точек.

Описание прямых

Представим, что у нас на холсте две точки, P_0 и P_1, с координатами (x_0, y_0) и (x_1, y_1). Как нарисовать отрезок между P_0 и P_1?

Начнем с представления прямой через параметрические координаты, как делали это с лучами ранее («лучи» можно рассматривать как прямые в 3D). Любую точку P на прямой можно получить, начав в P_0 и продвигаясь на некоторое расстояние вдоль к P_1:

$$P = P_0 + t(P_1 - P_0).$$

Разобьем это уравнение на два, по одному для каждой координаты:

$$x = x_0 + t(x_1 - x_0);$$

$$y = y_0 + t(y_1 - y_0).$$

Возьмем первое и решим его для t:

$$x = x_0 + t(x_1 - x_0);$$

$$x - x_0 = t(x_1 - x_0);$$

$$\frac{x - x_0}{x_1 - x_0} = t.$$

Теперь можно подставить это выражение для t во второе уравнение:

$$y = y_0 + t(y_1 - y_0);$$

$$y = y_0 + \frac{x - x_0}{x_1 - x_0}(y_1 - y_0).$$

Небольшая перестановка:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}.$$

Обратите внимание, что $\frac{y_1 - y_0}{x_1 - x_0}$ — константа, зависящая только от конечных точек отрезка. Назовем ее a. Теперь можно переписать уравнение выше как:

$$y = y_0 + a(x - x_0).$$

Что такое a? В нашем определении она отражает изменение в координате y на единицу изменения в координате x. Другими словами, это мера *уклона* прямой.

Вернемся к уравнению. Развернем умножение:

$$y = y_0 + ax - ax_0.$$

Сгруппируем константы:

$$y = ax + (y_0 - ax_0).$$

Опять же $(y_0 - ax_0)$ зависит только от конечных точек отрезка. Назовем ее b. В итоге мы получаем:

$$y = ax + b.$$

Это стандартная формулировка линейной функции, которую можно использовать для представления *почти* любой прямой. Решая уравнение для t, мы добавили деление на $x_1 - x_0$, не подумав о том, что произойдет, если $x_1 = x_0$. На нуль делить нельзя, поэтому такая формулировка не может представлять прямые с $x_1 = x_0$ — вертикальные.

Пока для решения этой проблемы мы просто проигнорируем вертикальные прямые. Мы вернемся к ним позже.

Рисование прямых

Теперь мы знаем, как получить значение y для каждого интересующего нас значения x. Это дает нам пару (x, y), удовлетворяющую уравнению прямой.

Напишем первую аппроксимацию функции, рисующей отрезок из P_0 в P_1. Пусть `x0` и `y0` будут координатами x и y для P_0, а `x1` и `y1` — для P_1. Предположив, что $x_0 < x_1$, мы можем перейти от x_0 к x_1, вычисляя значение y для каждого значения x и рисуя пиксель в этих координатах:

```
DrawLine(P0, P1, color) {
    a = (y1 - y0) / (x1 - x0)
    b = y0 - a * x0
    for x = x0 to x1 { y = a * x + b
        canvas.PutPixel(x, y, color)
    }
}
```

Заметьте, что оператор деления `/` должен выполнять деление с плавающей точкой, а не целочисленное. Несмотря на то что x и y здесь — целые числа, так как представляют координаты пикселей на холсте.

Циклы должны включать последнее значение диапазона. В C, C++, Java и JavaScript это записывалось бы как `for (x = x0; x <= x1; ++x)`. Такое условное соглашение мы будем использовать до конца книги.

Эта функция — простейшая прямая реализация уравнения выше. Она работает, но можно ли ее ускорить?

Мы не вычисляем значения y для любого произвольного x. Наоборот, мы вычисляем их только в целочисленных возрастаниях x и делаем это по порядку. Сразу после вычисления $y(x)$ мы вычисляем $y(x + 1)$:

$$y(x) = ax + b;$$

$$y(x + 1) = a(x + 1) + b.$$

Второе выражение можно переработать:

$$y(x + 1) = ax + a + b;$$

$$y(x + 1) = (ax + b) + a;$$

$$y(x + 1) = y(x) + a.$$

Не удивляйтесь. Наклонная a — мера того, насколько изменяется y при возрастании x на 1, что здесь у нас и происходит.

То есть можно вычислить очередное значение y, просто взяв предыдущее его значение и добавив этот уклон. Попиксельное умножение здесь не нужно. Это сильно ускоряет функцию. «Предыдущего значения y» нет изначально, поэтому мы начинаем в (x_0, y_0), а после продолжаем прибавлять 1 к x и a к y, пока не приходим к x_1.

Опять же, предполагая, что $x_0 < x_1$, можно переписать эту функцию так:

```
DrawLine(P0, P1, color) {
    a = (y1 - y0) / (x1 - x0)
    y = y0
    for x = x0 to x1 {
        canvas.PutPixel(x, y, color)
        y = y + a
    }
}
```

До этого момента мы предполагали, что $x_0 < x_1$. Есть простой способ поддерживать прямые там, где это окажется не так. Порядок рисования пикселей неважен, поэтому, если мы получаем прямую справа налево, можно просто обменять местами `P0` и `P1`, преобразовав ее в вариант слева направо, а потом нарисовать, как и прежде:

```
DrawLine(P0, P1, color) {
    // Убеждаемся, что x0 < x1if x0 > x1 {
        swap(P0, P1)
    }
    a = (y1 - y0) / (x1 - x0)
```

```
    y = y0
    for x = x0 to x1 {
        canvas.PutPixel(x, y, color)
        y = y + a
    }
}
```

Давайте используем нашу функцию для рисования пары прямых. На рис. 6.1 показан отрезок (–200, –100), (240, 120), а на рис. 6.2 дан крупный план прямой.

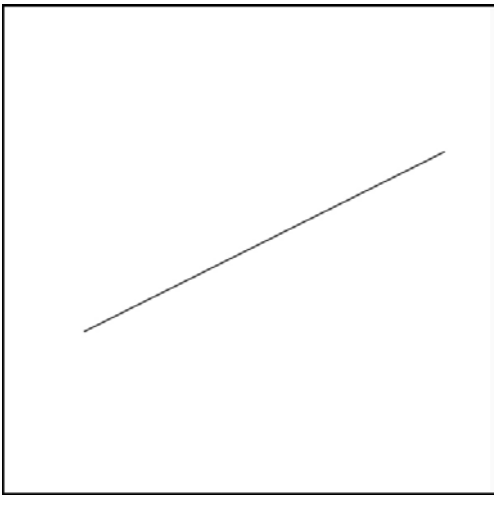


Рис. 6.1. Прямая линия

Рис. 6.2. Прямая линия в приближении

Линия ступенчатая, потому что мы можем закрашивать пиксели только в целочисленных координатах, а математические линии по факту имеют нулевую ширину. То, что мы рисуем, — это разбитая на ступени аппроксимация идеальной прямой из (–200, –100) в (240, 120). Можно нарисовать аппроксимации прямых и красивее

(ознакомьтесь с MSAA, FXAA, SSAA и TAA). Мы заниматься этим не станем по двум причинам: во-первых, это снизит быстродействие, а во-вторых, наша цель не в рисовании красивых линий, а в разработке базовых алгоритмов для рендеринга 3D-сцен.

Создадим еще одну прямую, [(–50, –200), (60, 240)]. На рис. 6.3 показан результат, а на рис. 6.4 — его масштабированная версия.

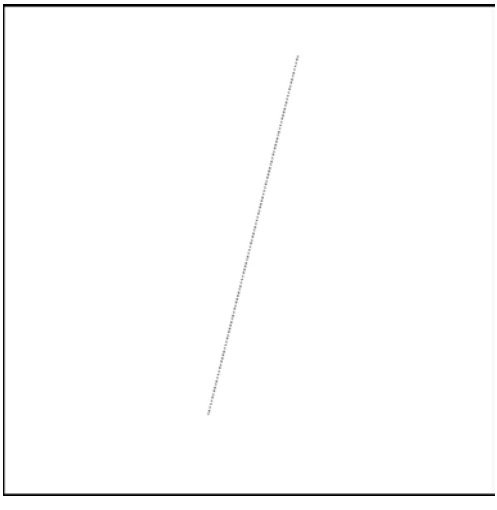


Рис. 6.3. Другая прямая линия с большим уклоном

Рис. 6.4. Вторая прямая линия в приближении

Что случилось?

Алгоритм сделал ровно то, что мы от него просили: прошел слева направо, вычислил по одному значению y для каждого значения x и закрасил соответствующие пиксели. Проблема в том, что он вычислил *одно* значение y для каждого значения x, хотя в этом случае для некоторых значений x нам нужно *несколько* значений y.

Это происходит, потому что мы выбрали формулировку, в которой $y = f(x)$. По этой же причине мы не можем рисовать вертикальные прямые — крайний случай, в котором все значения y соответствуют одному значению x.

Рисование прямых с любым уклоном

Выбор $y = f(x)$ был произвольным. С тем же успехом можно было выразить прямую как $x = f(y)$. Поменяв местами x и y во всех уравнениях, мы получим такой алгоритм:

```
DrawLine(P0, P1, color) {
    // Убеждаемся, что y0 < y1
    if y0 > y1 {
        swap(P0, P1)
    }
    a = (x1 - x0)/(y1 - y0) x = x0
    for y = y0 to y1 {
        canvas.PutPixel(x, y, color)
        x = x + a
    }
}
```

Он такой же, как и предыдущий `DrawLine`, но вычисления x и y здесь поменялись местами. Эта функция уже сможет обработать вертикальные прямые и отрисует (0, 0), (50, 100) как положено. Но обработать горизонтальные прямые или нарисовать корректно (0, 0), (100, 50) у нее не получится. Что же делать?

Можно просто сохранить обе версии функции и выбирать нужную в зависимости от того, какую прямую вы хотите нарисовать. Критерий здесь прост. Имеет ли прямая больше разных значений x, чем разных значений y? Если окажется больше x, чем y, нужно использовать первую версию; в ином случае — вторую.

В листинге 6.1 показана версия `DrawLine`, обрабатывающая все случаи.

Листинг 6.1. Версия DrawLine, обрабатывающая все случаи

```
DrawLine(P0, P1, color) {
    dx = x1 - x0
    dy = y1 - y0
    if abs(dx) > abs(dy) {
        // Прямая скорее горизонтальна
        // Убеждаемся, что x0 < x1 if x0 > x1 {
            swap(P0, P1)
```

```
        }
        a = dy/dx
        y = y0
        for x = x0 to x1 {
            canvas.PutPixel(x, y, color)
            y = y + a
        }
    } else {
        // Прямая скорее вертикальна
        // Убеждаемся, что y0 < y1
        if y0 > y1 {
            swap(P0, P1)
        }
        a = dx/dy
        x = x0
        for y = y0 to y1 {
            canvas.PutPixel(x, y, color)
            x = x + a
        }
    }
}
```

Вариант неэстетичный, но рабочий. Здесь мы видим много повторяющегося кода, а также сильно переплетаются логика выбора функции, логика вычисления значений и сам процесс отрисовки пикселей. Но это исправимо!

Линейная интерполяция

У нас есть две линейные функции: $y = f(x)$ и $x = f(y)$. Забудем о пикселях и напишем их в более общем виде: $d = f(i)$, где i — *независимая* переменная, для которой мы выбираем значения, а d — *зависимая переменная*, чье значение зависит от первой. Ее нам нужно вычислить. В горизонтальном случае x — независимая переменная, а y — зависимая. В вертикальном случае — наоборот.

Любую функцию можно записать как $d = f(i)$. Но мы знаем два момента, полностью определяющие *нашу* функцию: то, что она линейная, и два ее значения — $d_0 = f(i_0)$ и $d_1 = f(i_1)$. Можно написать простую функцию, получающую эти значения и возвращающую список всех промежуточных значений d, предположив, что $i_0 < i_1$:

```
Interpolate (i0, d0, i1, d1) {
    values = []
    a = (d1 - d0) / (i1 - i0)
    d = d0
    for i = i0 to i1 {
        values.append(d)
        d = d + a
    }
    return values
}
```

По форме эта функция похожа на первые две версии `DrawLine`, но переменные здесь — `i` и `d`, а не `x` и `y`. И вместо отрисовки пикселей она сохраняет значения в списке.

Обратите внимание, что значение d, соответствующее i_0, возвращается в `values[0]`, значение $i_0 + 1$ — в `values[1]` и т. д. Как правило, значение для i_n возвращается в `values[i_n - i_0]`, предполагая, что i_n находится в диапазоне $[i_0, i_1]$.

Рассмотрим граничный случай: нам может понадобиться вычислить $d = f(i)$, когда $i_0 = i_1$. Здесь мы не можем вычислить даже a, поэтому просто рассматриваем это как особый случай:

```
Interpolate (i0, d0, i1, d1) {
    if i0 == i1 {
        return [ d0 ]
    }
    values = []
    a = (d1 - d0) / (i1 - i0)
    d = d0
    for i = i0 to i1 {
        values.append(d)
        d = d + a
    }
    return values
}
```

Как деталь реализации и до конца книги значения i всегда будут целыми (поскольку представляют пиксели), а значения d всегда будут с плавающей запятой (поскольку представляют значения общей линейной функции).

Теперь можно написать `DrawLine` с помощью `Interpolate` (листинг 6.2).

Листинг 6.2. Версия DrawLine, использующая Interpolate

```
DrawLine(P0, P1, color) {
    if abs(x1 - x0) > abs(y1 - y0) {
        // Прямая скорее горизонтальна
        // Убеждаемся, что x0 < x1
        if x0 > x1 {
            swap(P0, P1)
        }
        ys = Interpolate(x0, y0, x1, y1)
        for x = x0 to x1 {
            canvas.PutPixel(x, ys[x - x0], color)
        }
    } else {
        // Прямая скорее вертикальна
        // Убеждаемся, что y0 < y1
        if y0 > y1 {
```

```
            swap(P0, P1)
        }
        xs = Interpolate(y0, x0, y1, x1)
        for y = y0 to y1 {
            canvas.PutPixel(xs[y - y0], y, color)
        }
    }
}
```

Этот вариант `DrawLine` может обрабатывать правильно все случаи (рис. 6.5).

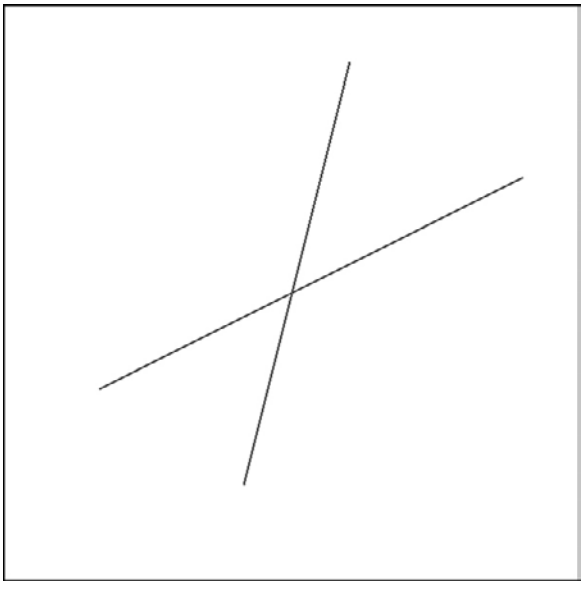


Рис. 6.5. Отрефакторенный алгоритм обрабатывает все случаи верно

Живое демо этого алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/lines-demo.

Эта версия не сильно короче предыдущей, но она более четко отделяет вычисление промежуточных значений y и x от определения, какая переменная является независимой и от самого кода отрисовки пикселей.

Вы удивитесь, но этот алгоритм отрисовки прямых лучший или самый быстрый из существующих. Таким можно назвать и *алгоритм Брезенхэма*. Но мы рассмотрели именно этот по двум причинам. Во-первых, его проще понять, а для нашей книги это базовый принцип. Во-вторых, он дал нам функцию `Interpolate`, которую мы будем активно использовать до конца книги.

Итоги главы

В этой главе вы проделали свои первые шаги по построению растеризатора. С помощью единственного инструмента в нашем арсенале, `PutPixel`, мы разработали алгоритм, способный рисовать на холсте отрезки прямых.

Мы разработали и вспомогательный метод `Interpolate`, представляющий способ эффективного вычисления значений линейной функции. Мы будем часто пользоваться этим методом. Поэтому прежде, чем перейти к новой главе, убедитесь, что вы все поняли в этой.

Дальше с помощью `Interpolate` мы нарисуем на холсте более сложные и интересные фигуры — треугольники.

7

Закрашенные треугольники

Ранее вы сделали первые шаги к отрисовке простых фигур — отрезков прямых. Для этого мы использовали только `PutPixel` и алгоритм на основе простой математики. В этой главе мы повторно задействуем часть этой математики для отрисовки закрашенного треугольника.

Отрисовка каркасных треугольников

С помощью метода `DrawLine` можно нарисовать контуры треугольника:

```
DrawWireframeTriangle (P0, P1, P2, color) {
    DrawLine(P0, P1, color);
    DrawLine(P1, P2, color);
    DrawLine(P2, P0, color);
}
```

Такой вид контуров называется *проволочным каркасом*, потому что треугольник выглядит так, будто сделан из проволоки (рис. 7.1).

Многообещающее начало! Дальше мы разберем заполнение этого треугольника цветом.

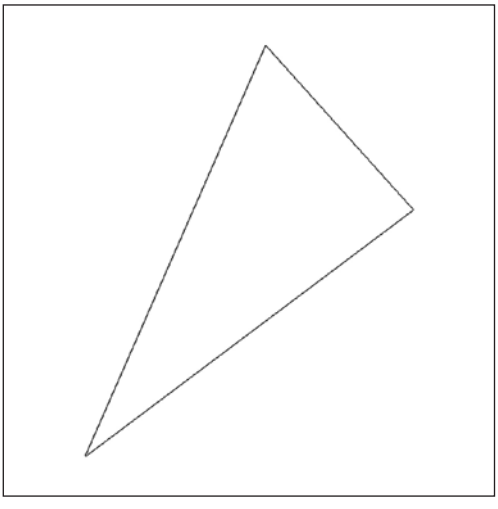


Рис. 7.1. Каркасный треугольник с вершинами (–200, –250), (200, 50) и (20, 250)

Отрисовка закрашенных треугольников

Нам нужно нарисовать треугольник, закрашенный любым цветом на выбор. Сделать это можно несколькими способами. Рисовать закрашенные треугольники мы будем, рассматривая их как коллекции горизонтальных отрезков, которые в совокупности будут выглядеть как треугольник. На рис. 7.2 схематически показано, как выглядит такой треугольник в виде отдельных отрезков.

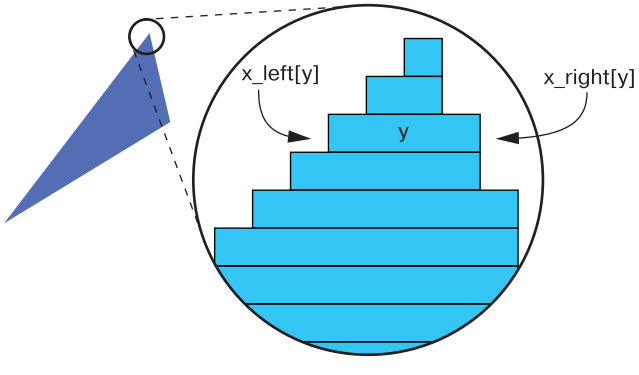


Рис. 7.2. Отрисовка закрашенного треугольника с помощью горизонтальных отрезков

Ниже очень приблизительно обозначено то, что мы собираемся делать:

```
for each horizontal line y between the triangle's top and bottom
    compute x_left and x_right for this y
    DrawLine(x_left, y, x_right, y)
```

Начнем с «между вершиной и основанием треугольника». Треугольник определяется тремя вершинами: P_0, P_1 и P_2. Если эти точки упорядочить по возрастанию значения y так, чтобы $y_0 \le y_1 \le y_2$, тогда диапазон значений y, входящих в треугольник, будет $[y_0, y_2]$:

```
if y1 < y0 { swap(P1, P0) }
if y2 < y0 { swap(P2, P0) }
if y2 < y1 { swap(P2, P1) }
```

Подобное упорядочение вершин все упрощает: после этого можно всегда предполагать, что P_0 — нижняя точка треугольника, а P_2 — верхняя. Благодаря этому у нас не будет путаницы с возможным их порядком.

Дальше вычислим массивы `x_left` и `x_right`. Это будет непросто, потому что у треугольника три стороны, а не две. И все же при рассмотрении только значений y у нас всегда будет длинная сторона от P_0 к P_1 и две коротких: от P_0 к P_1 и от P_1 к P_2.

Есть особый случай, когда $y_0 = y_1$ или $y_1 = y_2$. То есть когда одна из сторон треугольника горизонтальна. В такой ситуации две другие стороны одной высоты, значит, любую из них можно рассматривать как высокую. Нужно ли тогда выбрать левую или правую? К счастью, это неважно. Алгоритм будет поддерживать обе горизонтальные. Поэтому мы можем продолжать придерживаться нашего определения, что длинная сторона — это та, что идет от P_0 к P_2.

Значения для `x_right` будут браться либо из длинной стороны, либо из объединения коротких сторон. Значения для `x_left` будут браться из другого множества. Начнем с вычисления значений x для всех трех сторон. Мы будем рисовать горизонтальные отрезки, поэтому нам нужно ровно по одному значению x для каждого значения y. Это значит, что можно вычислить эти значения с помощью `Interpolate`, использовав y как независимую переменную, а x — как зависимую:

```
x01 = Interpolate(y0, x0, y1, x1)
x12 = Interpolate(y1, x1, y2, x2)
x02 = Interpolate(y0, x0, y2, x2)
```

Значения x для одной из сторон находятся в `x02`. Значения для другой стороны берутся из объединения `x01` и `x12`. Обратите внимание, что в `x01` и `x12` есть повторяющееся значение: x для y_1 — одновременно последнее значение `x01` и первое `x12`. Нам нужно избавиться от одного из них (мы произвольно выбираем последнее значение `x01`), после чего объединить эти массивы:

```
remove_last(x01)
x012 = x01 + x12
```

Мы получили `x02` и `x012`, и теперь нужно определить, какой из них `x_left`, а какой `x_right`. Для этого можно выбрать любую горизонтальную прямую (например, среднюю) и сравнить ее значения x в `x02` и `x012`. Если x в `x02` окажется меньше, чем x в `x012`, тогда `x02` должен быть `x_left`, а если наоборот — `x_right`.

```
m = floor(x02.length / 2)
if x02[m] < x012[m] {
    x_left = x02
    x_right = x012
} else {
    x_left = x012
    x_right = x02
}
```

Теперь у нас есть все данные для отрисовки горизонтальных отрезков. Можно использовать `DrawLine`, но она очень общая, а сейчас мы рисуем горизонтальные прямые слева направо, значит, эффективнее будет использовать цикл `for`. Это даст нам больше контроля над каждым рисуемым пикселем, что особенно пригодится в дальнейшем.

В листинге 7.1 отражена завершенная функция `DrawFilledTriangle`.

Листинг 7.1. Функция для отрисовки закрашенных треугольников

```
DrawFilledTriangle (P0, P1, P2, color) {
  ❶ // Упорядочиваем точки, чтобы y0 <= y1 <= y2
    if y1 < y0 { swap(P1, P0) }
    if y2 < y0 { swap(P2, P0) }
    if y2 < y1 { swap(P2, P1) }

  ❷ // Вычисляем координаты x сторон треугольника
    x01 = Interpolate(y0, x0, y1, x1)
    x12 = Interpolate(y1, x1, y2, x2)
    x02 = Interpolate(y0, x0, y2, x2)

  ❸ // Объединяем короткие стороны
    remove_last(x01)
    x012 = x01 + x12

  ❹ // Определяем, какая левая, а какая правая
    m = floor(x012.length / 2)
    if x02[m] < x012[m] {
        x_left = x02
        x_right = x012
    } else {
        x_left = x012 x_right = x02
    }
  ❺ // Рисуем горизонтальные отрезки
    for y = y0 to y2 {
```

```
        for x = x_left[y - y0] to x_right[y - y0] {
            canvas.PutPixel(x, y, color)
        }
    }
}
```

Разберемся, что здесь происходит. Эта функция получает в качестве аргументов три вершины треугольника в любом порядке. Нашему алгоритму нужно, чтобы они были упорядочены от нижней к верхней, поэтому выполняем сортировку ❶. Потом вычисляем значения `x` для каждого значения `y` этих трех сторон ❷ и объединяем массивы, полученные из двух «коротких» сторон ❸. Затем определяем, какой из них `x_left`, а какой `x_right` ❹. Для каждого горизонтального отрезка между верхом и основанием треугольника мы получаем его левые и правые координаты `x`, отрисовывая этот отрезок попиксельно ❺.

На рис. 7.3 показан результат. Для проверки мы вызываем `DrawFilledTriangle`, а потом `DrawWireFrameTriangle` с одинаковыми координатами, но с разными цветами. Всегда проверяйте полученные результаты — это очень действенный способ обнаружения ошибок в коде.

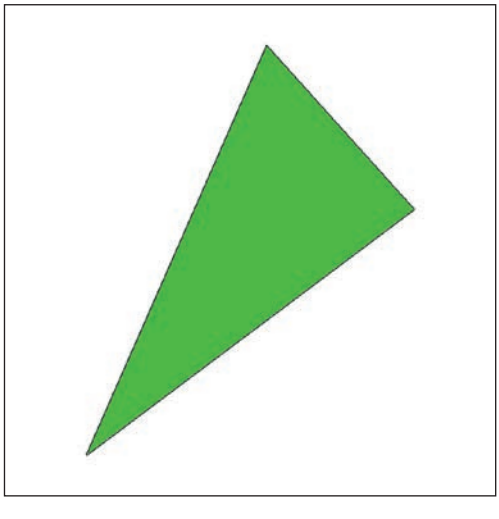


Рис. 7.3. Закрашенный треугольник с каркасными краями для проверки

Живую реализацию алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/triangle-demo.

Вы могли заметить, что черный контур треугольника не *точно* совпадает с зеленой заливкой. Дело в том, что `DrawLine` вычисляет для этой области $y = f(x)$,

а `DrawTriangle` — $x = f(y)$. Это дает немного разные результаты из-за округления. С таким видом ошибки аппроксимации мы смиряемся для получения быстрого алгоритма рендеринга.

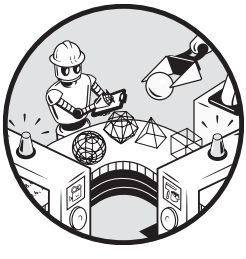
Итоги главы

В этой главе мы разработали алгоритм для отрисовки на холсте закрашенного треугольника. Это следующий шаг после отрисовки отрезков. Еще вы научились рассматривать треугольники как множества горизонтальных отрезков, с которыми можно работать по отдельности.

Дальше мы углубимся в математику и расширим этот алгоритм, чтобы нарисовать треугольник, заполненный цветным градиентом. Стоящие за этим математика и логика станут ключом к остальным возможностям, описанным дальше.

8

Затененные треугольники



В главе 8 мы разработали алгоритм для отрисовки закрашенного однородным цветом треугольника. Здесь наша цель — нарисовать *затененный* треугольник — закрашенный цветовым градиентом.

Определение задачи

Нам надо закрасить треугольник разными *оттенками* одного цвета. Итоговый результат показан на рис. 8.1.

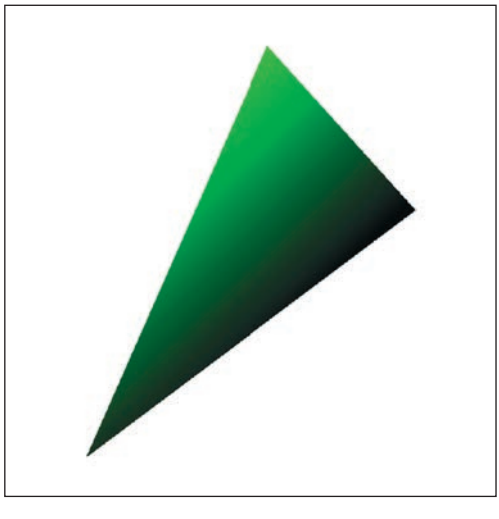


Рис. 8.1. Затененный треугольник

Здесь нам нужно более формально определить, что мы хотим нарисовать. У нас есть базовый цвет C: например, (0, 255, 0), чистый зеленый. Мы присвоим вещественное значение h каждой из его вершин, обозначив интенсивность цвета в них: h находится в диапазоне [0,0, 1,0], где 0,0 — максимально темный оттенок (черный), а 1,0 — самый яркий (исходный цвет — не белый!).

Для вычисления точного оттенка пикселя на основе базового цвета треугольника C и интенсивности этого пикселя h мы выполним поканальное умножение: $C_h = (R_C\,h,\ G_C\,h,\ B_C\,h)$. Так, $h = 0{,}0$ даст чистый черный, $h = 1{,}0$ — исходный цвет C, а $h = 0{,}5$ — цвет с половиной яркости исходного.

Вычисление затенения краев

Чтобы нарисовать затененный треугольник, нужно только вычислить значение h для каждого его пикселя, соответствующий оттенок цвета и закрасить этот пиксель. Очень просто!

Но сейчас мы знаем значения h только для вершин треугольника, потому что сами их выбрали. Как же вычислить значения h для остальной его части?

Начнем с краев. Рассмотрим ребро AB. Нам известны h_A и h_B. Что происходит в M, средней точке AB? Нам нужно, чтобы интенсивность плавно изменялась от A к B, поэтому значение h_M должно быть между h_A и h_B. Так как M в середине AB, почему бы не выбрать для h_M среднее между h_A и h_B?

Другими словами, у нас есть функция $h = f(P)$, сообщающая каждой точке P значение интенсивности h. Мы знаем, что эти значения в A и B равны $h(A) = h_A$ и $h(B) = h_B$, и хотим, чтобы функция была плавной. Так как про $h = f(P)$ мы больше ничего не знаем, можно выбрать любую функцию, совместимую с тем, что нам известно, например линейную (рис. 8.2).

Ничего вам не напоминает? В прошлой главе уже была такая ситуация: у нас была линейная функция $x = f(y)$, мы знали ее значения в вершинах треугольника и хотели вычислить значения x вдоль его сторон. Вычислить значения h вдоль сторон треугольника можно похожим образом, используя `Interpolate` с y как независимой переменной (эти значения мы знаем) и h как зависимой (эти значения нам нужны):

```
x01 = Interpolate(y0, x0, y1, x1)
h01 = Interpolate(y0, h0, y1, h1)

x12 = Interpolate(y1, x1, y2, x2)
h12 = Interpolate(y1, h1, y2, h2)

x02 = Interpolate(y0, x0, y2, x2)
h02 = Interpolate(y0, h0, y2, h2)
```

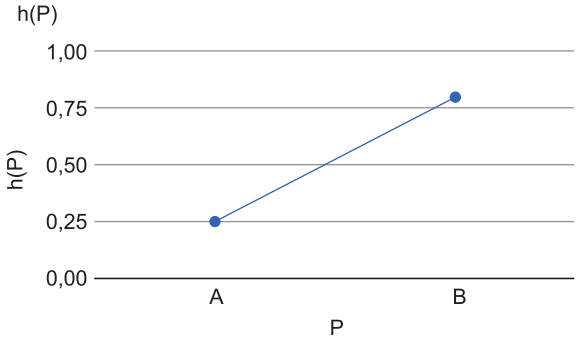


Рис. 8.2. Линейная функция $h(P)$, совместимая с известными нам данными об $h(A)$ и $h(B)$

Затем мы объединили массивы x для коротких сторон, а после определяли, какой из `x02` и `x012` соответствовал `x_left`, а какой `x_right`. Здесь с векторами h можно поступить так же.

Но мы всегда будем использовать значения x для определения левой и правой стороны, а значения h будут просто «следовать за этим». x и h — это свойства фактических точек на экране, значит, мы не можем свободно смешивать и сопоставлять значения левой и правой стороны.

Прописать в коде это можно так:

```
// Объединяем короткие стороны
remove_last(x01)
x012 = x01 + x12

remove_last(h01)
h012 = h01 + h12

// Определяем, какая левая, а какая правая
m = floor(x012.length / 2)
if x02[m] < x012[m] {
    x_left = x02
    h_left = h02

    x_right = x012
    h_right = h012
} else {
    x_left = x012
    h_left = h012

    x_right = x02
    h_right = h02
}
```

Это очень похоже на такой же раздел кода в прошлой главе (см. листинг 7.1). Разница лишь в том, что каждый раз, когда мы что-то делаем с вектором x, мы делаем то же самое с соответствующим вектором h.

Вычисление внутреннего затенения

Последним шагом будет отрисовка фактических горизонтальных отрезков. Для каждого отрезка нам известны `x_left` и `x_right`. Теперь нам известны еще и `h_left` и `h_right`. Но сейчас мы не можем просто выполнить перебор слева направо и отрисовать каждый пиксель с базовым цветом. Нужно вычислить значение h для *каждого пикселя* отрезка.

Можно предположить, что h изменяется линейно относительно x, и использовать `Interpolate` для вычисления этих значений. Тогда независимой переменной будет x, и она смещается от значения `x_left` к `x_right` конкретного горизонтального отрезка, который мы затеняем. Зависимая переменная — это h, а ее соответствующие значения для `x_left` и `x_right` — это `h_left` и `h_right` для этого отрезка:

```
x_left_this_y = x_left[y - y0]
h_left_this_y = h_left[y - y0]

x_right_this_y = x_right[y - y0]
h_right_this_y = h_right[y - y0]

h_segment = Interpolate(x_left_this_y, h_left_this_y,
                        x_right_this_y, h_right_this_y)
```

Или компактнее:

```
h_segment = Interpolate(x_left[y - y0], h_left[y - y0],
                        x_right[y - y0], h_right[y - y0])
```

Осталось только вычислить цвет каждого пикселя и закрасить его. В листинге 8.1 показан весь псевдокод для `DrawShadedTriangle`.

Листинг 8.1. Функция для отрисовки затененных треугольников

```
DrawShadedTriangle (P0, P1, P2, color) {
  ❶ // Упорядочиваем точки, чтобы y0 <= y1 <= y2
    if y1 < y0 { swap(P1, P0) }
    if y2 < y0 { swap(P2, P0) }
    if y2 < y1 { swap(P2, P1) }

    // Вычисляем координаты x и значения h ребер треугольника
    x01 = Interpolate(y0, x0, y1, x1)
    h01 = Interpolate(y0, h0, y1, h1)
```

```
    x12 = Interpolate(y1, x1, y2, x2)
    h12 = Interpolate(y1, h1, y2, h2)

    x02 = Interpolate(y0, x0, y2, x2)
    h02 = Interpolate(y0, h0, y2, h2)

    // Объединяем короткие стороны
    remove_last(x01)
    x012 = x01 + x12

    remove_last(h01)
    h012 = h01 + h12

    // Определяем, какая левая, а какая правая
    m = floor(x012.length / 2)
    if x02[m] < x012[m] {
        x_left = x02
        h_left = h02

        x_right = x012
        h_right = h012
    } else {
        x_left = x012
        h_left = h012

        x_right = x02
        h_right = h02
    }

    // Рисуем горизонтальные отрезки
    ❷ for y = y0 to y2 {
        x_l = x_left[y - y0]
        x_r = x_right[y - y0]

      ❸ h_segment = Interpolate(x_l, h_left[y - y0], x_r, h_right[y - y0])
        for x = x_l to x_r {
          ❹ shaded_color = color * h_segment[x - x_l]
            canvas.PutPixel(x, y, shaded_color)
        }
    }
}
```

Этот псевдокод очень похож на тот, что мы писали в главе 7, в листинге 7.1. Перед отрисовкой горизонтальных отрезков ❷ мы управляем векторами x и h способами, описанными выше. Внутри цикла у нас есть дополнительный вызов `Interpolate` ❸, вычисляющий значения h для каждого пикселя текущего горизонтального отрезка. И во внутреннем цикле мы используем интерполированные значения h для вычисления цвета каждого пикселя ❹.

Заметьте, что мы, как и раньше, упорядочиваем вершины треугольника ❶. Только теперь мы рассматриваем их и их атрибуты, например значение интенсивности h, как неделимое целое. Это значит, что при обмене местами координат двух вершин местами обмениваются и их атрибуты.

Найти живую реализацию этого алгоритма можно по ссылке https://gabrielgambetta.com/cgfs/gradient-demo.

Итоги главы

В этой главе мы расширили код отрисовки треугольника из предыдущей, добавив поддержку треугольников с плавным затенением. Но мы все еще можем применить его для отрисовки одноцветных треугольников, используя 1,0 как значение h для всех трех вершин.

Этот алгоритм гораздо шире, чем кажется. То, что h — значение интенсивности, не влияет на «форму» алгоритма. Смысл у этого значения появляется только в самом конце перед вызовом `PutPixel`. Это значит, что алгоритм можно использовать для вычисления значения любого *атрибута* вершин треугольника любого его пикселя, если это значение изменяется на экране линейно.

Вся информация из этой главы понадобится нам в дальнейшем, так что разберитесь во всем хорошенько, прежде чем мы продолжим. Но в следующей главе мы немного отклонимся от курса и переключим внимание на третье измерение.

9

Перспективная проекция

Вы уже научились рисовать на холсте 2D-треугольники на основе двухмерных координат их вершин. Но книга посвящена отрисовке трехмерных сцен. И здесь мы отвлечемся от работы с 2D-треугольниками и сосредоточимся на преобразовании координат 3D-сцены в координаты 2D-холста. Это поможет нам отрисовать на двухмерном холсте трехмерные треугольники.

Базовые допущения

Как и в главе 2, мы начнем с определения *камеры*. Для этого используем те же допущения: камера находится в $O = (0, 0, 0)$, смотрит в направлении $\vec{Z}_+$, а ее вектор вверх соответствует $\vec{Y}_+$. Теперь определим прямоугольное *окно просмотра* с размерами V_w и V_h, чьи ребра будут параллельны $\vec{X}$ и $\vec{Y}$, расположив его на расстоянии d от камеры. Наша цель — отрисовать на холсте все, что видит камера через окно просмотра.

Рассмотрим точку P где-то перед камерой. Нам нужно найти точку окна просмотра P', через которую камера видит P, как показано на рис. 9.1.

Противоположное мы делали с трассировщиком: он начинал с точки на холсте и определял, что можно увидеть через нее. А здесь мы начинаем с точки в сцене и хотим определить, откуда из окна просмотра она видна.

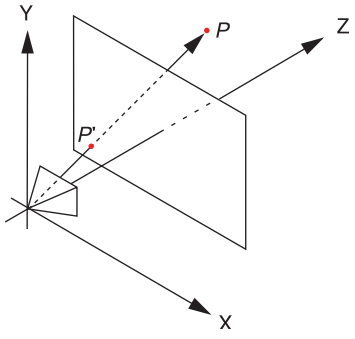


Рис. 9.1. Построение простой перспективной проекции. Камера видит *P* через *P'*, находящуюся на плоскости проекции

Поиск *P'*

Чтобы найти *P'*, посмотрим на конфигурацию с рис. 9.1 под другим углом. На рис. 9.2 показан чертеж конфигурации, видимой справа, как если бы мы стояли на оси $\vec{X}$: $\vec{Y}_+$ указывает вверх, $\vec{Z}_+$ указывает вправо, а $\vec{X}_+$ смотрит на нас.

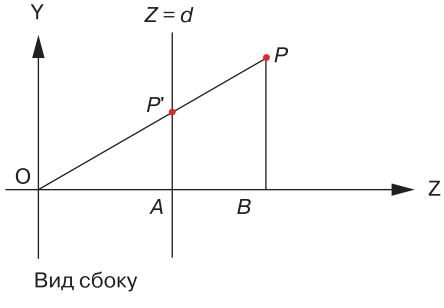


Рис. 9.2. Конфигурация перспективной проекции, вид справа

Чтобы нам было проще, помимо *O*, *P* и *P'*, в этом чертеже отмечены точки *A* и *B*.

Мы знаем, что $P'_z = d$, потому *P'* — точка в окне просмотра, и нам известно, что окно просмотра встроено в плоскость $Z = d$.

Еще мы видим, что треугольники *OP'A* и *OPB* похожи, потому что их соответствующие стороны (*P'A* и *PB*, *OP* и *OP'*, *OA* и *OB*) параллельны. Поэтому пропорции их сторон одинаковы. Например:

$$\frac{|P'A|}{|OA|} = \frac{|PB|}{|OB|}.$$

Отсюда мы получаем:

$$|P'A| = \frac{|PB||OA|}{|OB|}$$

Длина (со знаком) каждого отрезка в этом уравнении — координата точки, которую мы знаем или которой интересуемся: $|P'A| = P'_y$, $|PB| = P_y$, $|OA| = P'_z = d$, а $|OB| = P_z$. В уравнении это выглядит так:

$$P'_y = \frac{P_y d}{P_z}.$$

Можно составить такой же чертеж, но с видом сверху: $\vec{Z}_+$ направлена вверх, $\vec{X}_+$ направлена вправо, а $\vec{Y}_+$ смотрит на нас (рис. 9.3).

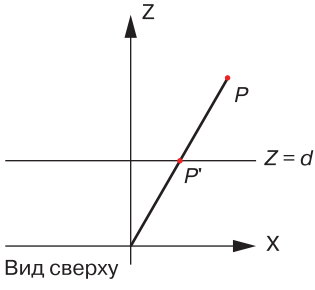


Рис. 9.3. Конфигурация перспективной проекции, вид сверху

Повторив эту процедуру, можно заключить, что:

$$P'_x = \frac{P_x d}{P_z}.$$

У нас есть все три координаты P'.

Уравнение проекции

Теперь все это объединим. У нас есть точка P в сцене и стандартная конфигурация камеры с окном просмотра. Вычислим проецируемую на окно просмотра P, которую назовем P', так:

$$P'_x = \frac{P_x d}{P_z};$$

$$P'_y = \frac{P_y d}{P_z};$$

$$P'_z = d.$$

P' находится в окне просмотра, но это все равно точка в трехмерном пространстве. Как же нам получить такую точку на холсте?

Сразу отбросим P'_z, потому что каждая спроецированная точка находится на плоскости окна просмотра. Теперь нужно преобразовать P'_x и P'_y в координаты холста C_x и C_y. P' — все еще точка в сцене, значит, ее координаты выражены в единицах измерения сцены. Их мы делим на ширину и высоту окна просмотра, которые тоже выражаются в единицах измерения сцены. Получаем значения, временно лишенные единиц измерения. В конце умножаем их на ширину и высоту холста в пикселях:

$$C_x = \frac{P'_x C_w}{V_w};$$
$$C_y = \frac{P'_y C_h}{V_h}.$$

Этот перенос противоположен переносу с холста в окно просмотра, который мы делали, когда занимались трассировкой лучей. И теперь мы можем перейти от точки в сцене к пикселю на экране.

Свойства уравнения проекции

Прежде, чем продолжить, рассмотрим несколько интересных свойств уравнения проекции.

Уравнения выше должны совпадать с нашим повседневным видением вещей. Чем дальше находится объект, тем меньше он выглядит. И действительно, если увеличить P_z, то мы получим меньшие значения P'_x и P'_y.

Но все меняется, когда значение P_z уменьшается слишком сильно. Для отрицательных значений P_z, когда объект *позади* камеры, он все еще проецируется, но перевернутым. И при $P_z = 0$ мы бы делили на нуль, чего делать нельзя. Как нам избежать таких ситуаций? Пока отложим решение этой проблемы до следующей главы и допустим, что каждая точка находится перед камерой. Еще одно основное свойство перспективной проекции в том, что она сохраняет коллинеарность точек. Если три точки коллинеарны в пространстве, их проекции будут коллинеарны в окне просмотра. Другими словами, прямая линия всегда проецируется как прямая линия. Но *угол* между двумя прямыми не сохраняется, ведь в реальности мы видим, как параллельные линии «сходятся» на горизонте.

То, что прямая линия всегда проецируется как прямая линия, нам на руку. Ранее мы рассматривали проецирование точки, но что насчет отрезка или даже треугольника?

Благодаря этому свойству проекция отрезка между двух точек — это отрезок между проекцией двух точек, а проекция треугольника — это треугольник, сформированный проекциями его вершин.

Проецирование первого 3D-объекта

Теперь можно приступать к отрисовке нашего первого 3D-объекта — куба. Мы определяем координаты восьми его вершин и рисуем отрезки между проекциями 12 пар вершин, составляющих ребра, как показано в листинге 9.1.

Листинг 9.1. Отрисовка куба

```
ViewportToCanvas(x, y) {
    return (x * Cw/Vw, y * Ch/Vh);
}
ProjectVertex(v) {
    return ViewportToCanvas(v.x * d / v.z, v.y * d / v.z)
}

// Четыре передних вершины
vAf = [-1, 1, 1]
vBf = [ 1,  1, 1]
vCf = [ 1, -1, 1]
vDf = [-1, -1, 1]

// Четыре задних вершины
vAb = [-1, 1, 2]
vBb = [ 1,  1, 2]
vCb = [ 1, -1, 2]
vDb = [-1, -1, 2]

// Лицевая сторона
DrawLine(ProjectVertex(vAf), ProjectVertex(vBf), BLUE); DrawLine(ProjectVertex(vBf),
ProjectVertex(vCf), BLUE); DrawLine(ProjectVertex(vCf), ProjectVertex(vDf), BLUE);
DrawLine(ProjectVertex(vDf), ProjectVertex(vAf), BLUE);

// Тыльная сторона
DrawLine(ProjectVertex(vAb), ProjectVertex(vBb), RED);

DrawLine(ProjectVertex(vBb), ProjectVertex(vCb), RED); DrawLine(ProjectVertex(vCb),
ProjectVertex(vDb), RED); DrawLine(ProjectVertex(vDb), ProjectVertex(vAb), RED);

// Ребра от лицевой стороны к тыльной
DrawLine(ProjectVertex(vAf), ProjectVertex(vAb), GREEN);
DrawLine(ProjectVertex(vBf), ProjectVertex(vBb), GREEN);
DrawLine(ProjectVertex(vCf), ProjectVertex(vCb), GREEN);
DrawLine(ProjectVertex(vDf), ProjectVertex(vDb), GREEN);
```

Получаем изображение как на рис. 9.4.

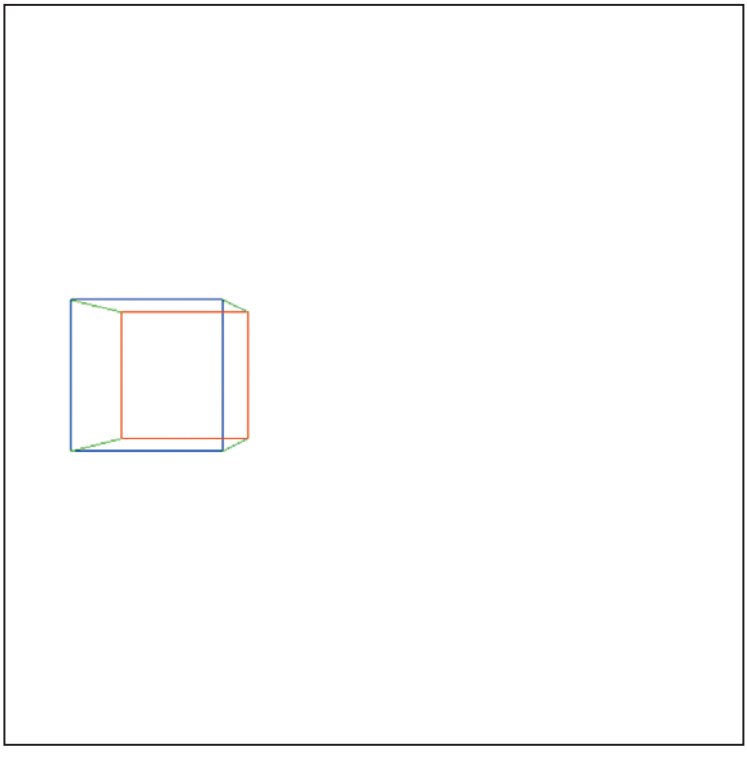


Рис. 9.4. Наш первый 3D-объект, спроецированный на 2D-холст, — куб

Живая реализация этого алгоритма находится по адресу https://gabrielgambetta.com/cgfs/perspective-demo.

Все получилось! Мы смогли перейти от геометрического 3D-представления объекта к его 2D-представлению, видимому с позиции нашей камеры.

Но такой подход очень кустарный и содержит много ограничений. Что, если мы захотим отрисовать *два* куба? Придется ли тогда повторять большую часть кода? Что, если нам нужно будет отрисовать уже не куб? А если мы захотим дать пользователям возможность загружать 3D-модели из файла? Для представления трехмерной геометрии нужен подход, в большей степени основанный на данных.

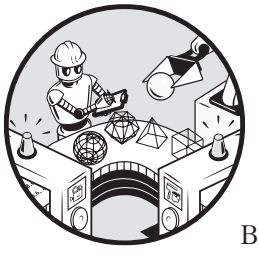
Итоги главы

В этой главе мы разработали математическую модель для перехода от трехмерной точки в сцене к двухмерной на холсте. Мы можем сразу же расширить эту модель до проецирования отрезков и 3D-объектов.

Но у нас остались две неразрешенные проблемы. Во-первых, листинг 9.1 смешивает логику перспективной проекции с геометрией куба. Этот подход явно не масштабируется. Во-вторых, из-за ограничений уравнения перспективной проекции он не может обрабатывать объекты за камерой. Со всем этим мы разберемся в следующих двух главах.

10

Описание и рендеринг сцены



Мы уже разработали алгоритмы для отрисовки 2D-треугольников на холсте при наличии их 2D-координат и разобрали математику для преобразования трехмерных координат точек сцены в двухмерные на холсте.

В конце предыдущей главы мы собрали программу, которая использовала и то и другое для рендеринга 3D-куба на 2D-холсте. Здесь мы расширим эту работу для выполнения рендеринга всей сцены с произвольным количеством объектов.

Представление куба

Давайте еще раз подумаем, как можно представить куб и управлять им, но на этот раз попытаемся найти более универсальный подход. Длина ребер нашего куба — 2 единицы, они расположены параллельно осям координат и центрированы относительно точки отсчета, как показано на рис. 10.1.

Вот координаты его вершин:

- $A = (1, 1, 1)$;
- $B = (-1, 1, 1)$;
- $C = (-1, -1, 1)$;
- $D = (1, -1, 1)$;

- $E = (1, 1, -1)$;
- $F = (-1, 1, -1)$;
- $G = (-1, -1, -1)$;
- $H = (1, -1, -1)$.

Стороны куба — это квадраты, но созданный нами алгоритм работает с треугольниками. Но мы выбрали их не просто так. Как говорилось ранее, любую геометрическую фигуру можно разобрать именно на треугольники. Поэтому каждую квадратную сторону куба мы будем представлять через два треугольника.

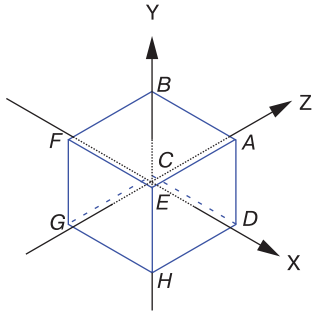


Рис. 10.1. Наш стандартный куб

Но мы не можем взять *любые* три вершины куба и ждать, что они опишут треугольник именно на его поверхности (к примеру, ADG находится внутри куба). Это значит, что сами по себе координаты вершин описывают куб не полностью. Еще нам нужно знать, какие комбинации трех вершин описывают треугольники, составляющие его стороны.

Вот возможный список треугольников нашего куба:

```
A, B, C
A, C, D
E, A, D
E, D, H
F, E, H
F, H, G
B, F, G
B, G, C
E, F, B
E, B, A
C, G, H
C, H, D
```

Он предполагает обобщенную структуру, которую можно использовать для представления *любого* состоящего из треугольников объекта: список `Vertices` с координатами каждой вершины и список `Triangles`, указывающий, какие наборы трех вершин описывают треугольники на поверхности объекта.

В `Triangles` может быть дополнительная информация, помимо вершин. Например, это идеальное место для указания цвета каждого треугольника.

Лучше всего сохранять эту информацию в двух списках, поэтому для обращения к вершинам в списке вершин мы будем использовать индексы списков. В итоге наш куб будет представлен так:

```
Vertices
0 = ( 1,  1, 1)
1 = (-1,  1, 1)
2 = (-1, -1, 1)
3 = ( 1, -1, 1)
4 = ( 1,  1, -1)
5 = (-1,  1, -1)
6 = (-1, -1, -1)
7 = ( 1, -1, -1)

Triangles
0 = 0, 1, 2, red
1 = 0, 2, 3, red
2 = 4, 0, 3, green
3 = 4, 3, 7, green
4 = 5, 4, 7, blue
5 = 5, 7, 6, blue
6 = 1, 5, 6, yellow
7 = 1, 6, 2, yellow
8 = 4, 5, 1, purple
9 = 4, 1, 0, purple
10 = 2, 6, 7, cyan
11 = 2, 7, 3, cyan
```

Рендерить с таким представлением будет несложно: сначала проецируем каждую вершину, сохраняя их во временном списке спроецированных вершин (каждая вершина используется в среднем четыре раза, и это избавляет от лишней работы). После проходим по списку треугольников, отрисовывая каждый из них. В листинге 10.1 дается общая структура кода.

Листинг 10.1. Алгоритм для отрисовки любого объекта из треугольников

```
RenderObject(vertices, triangles) {
    projected = []
    for V in vertices {
        projected.append(ProjectVertex(V))
    }
```

```
    for T in triangles {
        RenderTriangle(T, projected)
    }
}

RenderTriangle(triangle, projected) {
    DrawWireframeTriangle(projected[triangle.v[0]],
                          projected[triangle.v[1]],
                          projected[triangle.v[2]],
                          triangle.color)
}
```

Можно применить его к определенному ранее кубу, но результат получится не очень. Некоторые вершины находятся позади камеры, а это, как мы уже говорили, приводит к странностям. Если посмотреть на координаты вершин и рис. 10.1, можно заметить, что начало координат, позиция камеры, находится *внутри* куба.

Для решения этой проблемы нужно просто сместить куб. Для этого сдвинем каждую его вершину в одном направлении, назовем его $\vec{T}$. Мы переместим куб на 7 единиц вперед, чтобы он точно оказался перед камерой полностью. Теперь перенесем его на 1,5 единицы влево, так он будет смотреться поинтереснее. Поскольку «вперед» — это направление $\vec{Z}_+$, а «влево» — направление $\vec{X}_-$, вектором переноса будет:

$$\vec{T} = \begin{pmatrix} -1{,}5 \\ 0 \\ 7 \end{pmatrix}.$$

Для вычисления перенесенной версии V' каждой вершины V куба нужно просто добавить к ней вектор переноса:

$$V' = V + \vec{T}.$$

Теперь можно взять куб, перенести каждую его вершину и применить алгоритм из листинга 10.1 для получения первого 3D-куба (рис. 10.2).

Живую реализацию этого алгоритма можно найти по ссылке https://gabrielgambetta.com/cgfs/scene-demo.

Модели и экземпляры

А что, если мы хотим отрисовать два куба? Проще всего создать новый набор вершин и треугольников, описывающих второй куб. Но это окажется слишком затратным для памяти. Что, если отрисовать нужно *миллион кубов*?

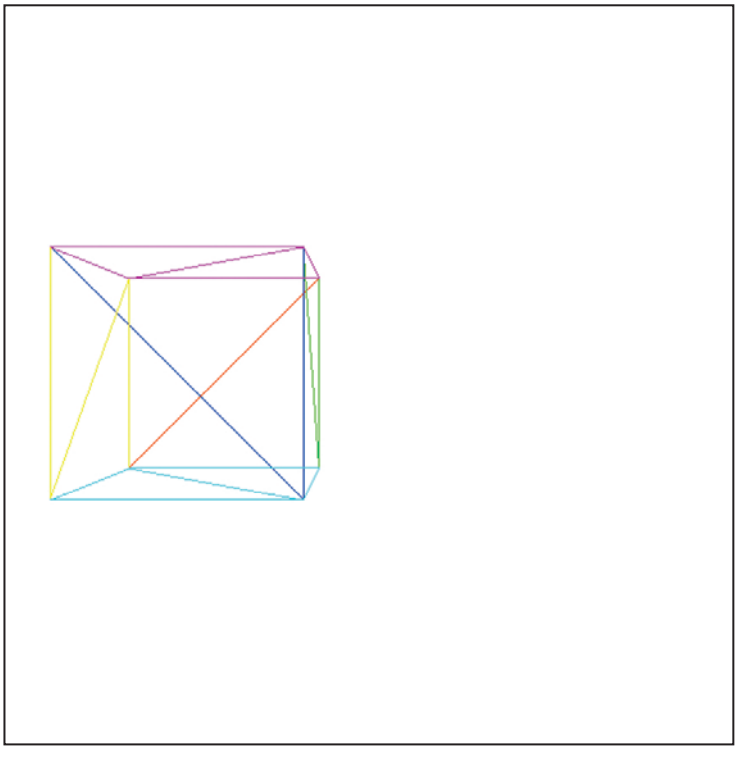


Рис. 10.2. Куб, перенесенный вперед камеры и отрисованный с помощью каркасных треугольников

Более оптимальный вариант — решение на уровне *моделей* и *экземпляров*. Модель — это набор вершин и треугольников, описывающий конкретный объект в общем («у куба есть восемь вершин и шесть сторон»). А вот экземпляр описывает конкретное вхождение модели в сцене («в позиции (0, 0, 5) находится куб»).

Как же применить эту идею на практике? Можно взять по одному описанию для каждого уникального объекта сцены и добавить много его копий, указав нужные координаты. В неформальном виде это будет как-то так: «Вот так выглядит куб, а расположен он там, там и там».

Ниже приведен очень приблизительный пример описания сцены с помощью такого подхода:

```
model {
    name = cube
```

```
    vertices {
        ...
    }
    triangles {
        ...
    }
}

instance {
    model = cube position = (0, 0, 5)
}

instance {
    model = cube position = (1, 2, 3)
}
```

Для отрисовки мы просто перебираем список вариантов. При этом делаем копию вершин модели каждого, переносим их согласно позиции этого экземпляра, а потом отрисовываем, как и раньше (листинг 10.2).

Листинг 10.2. Алгоритм для отрисовки сцены, которая может содержать много вариантов нескольких объектов в разных позициях

```
RenderScene() {
    for I in scene.instances {
        RenderInstance(I);
    }
}

RenderInstance(instance) {
    projected = []
    model = instance.model
    for V in model.vertices {
        V' = V + instance.position
        projected.append(ProjectVertex(V'))
    }
    for T in model.triangles {
        RenderTriangle(T, projected)
    }
}
```

Если мы хотим, чтобы все сработало так, как запланировано, то координаты вершин модели должны быть определены в системе координат, «соответствующей» объекту. Назовем эту систему координат *пространством модели*. К примеру, мы определили куб так, что его центр оказался в позиции (0, 0, 0). То есть когда мы говорим: «Куб расположен в (1, 2, 3)», то имеем в виду: «Куб центрирован вокруг (1, 2, 3)».

После применения этого перевода экземпляров в вершины, определенные в пространстве модели, полученные вершины выражаются в системе координат сцены. Эта система координат называется *глобальным пространством (world space)*.

Для определения пространства модели жестких правил не существует. Все зависит от потребностей приложения. К примеру, если у вас есть модель человека, может оказаться наиболее разумным разместить начало системы координат у его ног.

На рис. 10.3 показана простая сцена с двумя экземплярами куба.

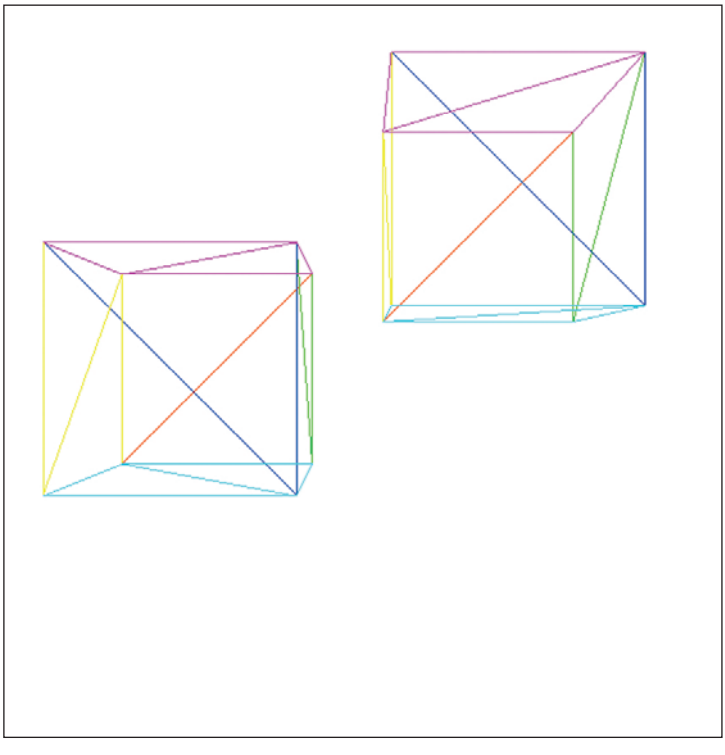


Рис. 10.3. Сцена с двумя экземплярами одной и той же модели куба, размещенными в разных позициях

Живая реализация этого алгоритма находится по адресу https://gabrielgambetta.com/cgfs/instances-demo.

Преобразование модели

Описанное выше определение сцены не дает нам нужной гибкости. Мы можем задавать только *позицию* куба, и инстанцировать их можно сколько угодно, но все они будут смотреть в одну сторону. Но мы хотим указывать их ориентацию в пространстве и по возможности масштаб.

Можно определить *преобразование модели* со следующими тремя элементами: фактором масштабирования, вращением вокруг начала координат в пространстве модели и переносом в конкретную точку сцены:

```
instance {
    model = cube transform {
        scale = 1.5
        rotation = <45 degrees around the Y axis>
        translation = (1, 2, 3)
    }
}
```

Можно расширить алгоритм из листинга 10.2 добавлением новых преобразований. Но здесь важен порядок их *применения*: перенос должен быть последним. В основном нам нужно вращать и масштабировать экземпляры вокруг начала координат в пространстве модели, поэтому следует делать это перед их переносом в глобальное пространство.

Чтобы понять разницу, посмотрите на рис. 10.4. Здесь показано вращение на 45° вокруг начала координат, сопровождаемое переносом вдоль оси *Z*.

На рис. 10.5 показан перенос, выполненный до вращения.

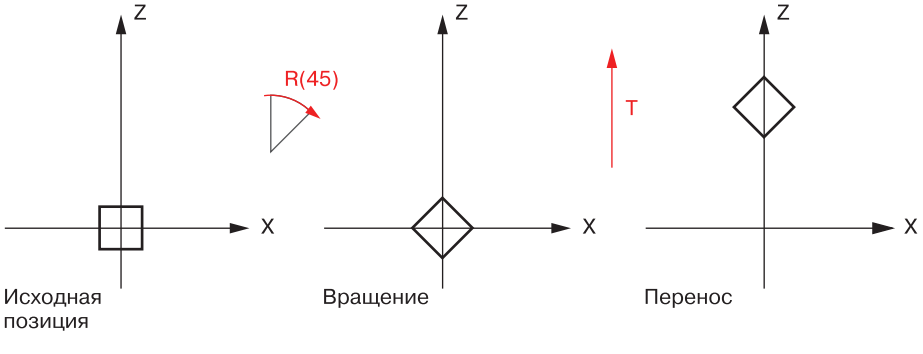


Рис. 10.4. Применение вращения, потом переноса

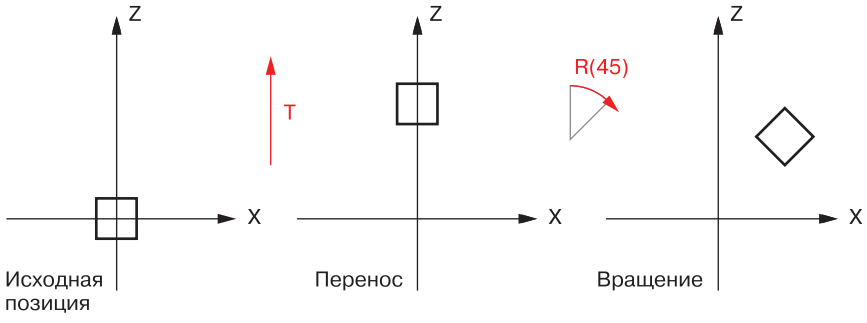


Рис. 10.5. Применение переноса, потом вращения

Вообще, рассматривая сопровождаемое переносом вращение, можно найти такой вариант переноса, сопровождаемого вращением (скорее всего, не вокруг начала координат), который даст такой же результат. Но лучше выразить это преобразование именно так.

Теперь можно написать новую версию `RenderInstance` с поддержкой масштабирования, вращения и позиционирования (листинг 10.3).

Листинг 10.3. Алгоритм для отрисовки сцены, способной содержать много экземпляров нескольких объектов с разным преобразованием

```
RenderInstance(instance) {
    projected = []
    model = instance.model
    for V in model.vertices {
        V' = ApplyTransform(V, instance.transform)
        projected.append(ProjectVertex(V'))
    }
    for T in model.triangles {
        RenderTriangle(T, projected)
    }
}
```

В листинге 10.4 представлен метод `ApplyTransform`.

Листинг 10.4. Функция, применяющая преобразования к вершине в правильном порядке

```
ApplyTransform(vertex, transform) {
    scaled = Scale(vertex, transform.scale)
    rotated = Rotate(scaled, transform.rotation)
    translated = Translate(rotated, transform.translation)
    return translated
}
```

Преобразование камеры

Мы уже разобрали возможность позиционирования моделей в разных точках сцены. В этом разделе мы рассмотрим, как перемещать и вращать в ней камеру.

Представьте себя камерой, плавающей в середине пустой системы координат. Внезапно прямо перед вами возникает красный куб (рис. 10.6).

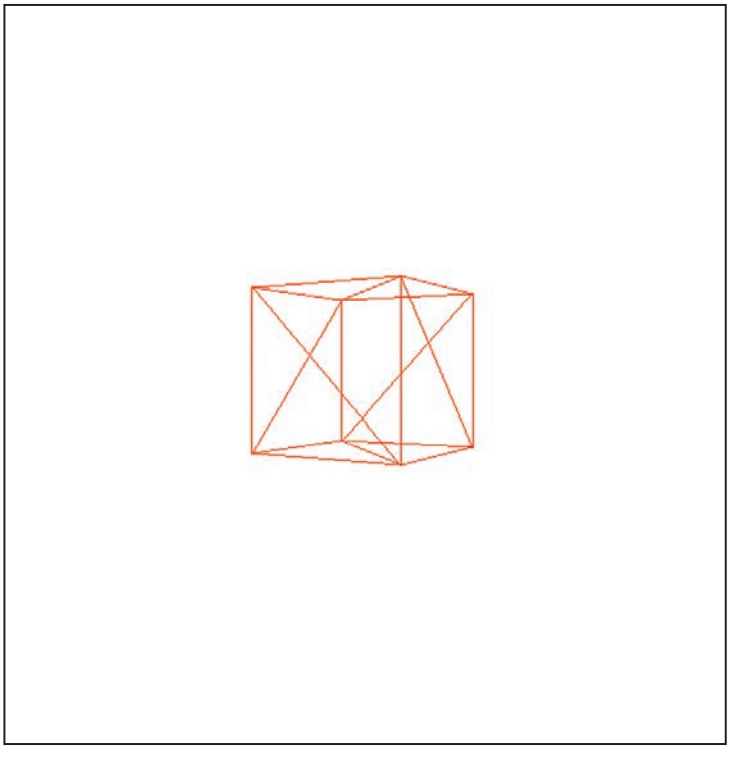


Рис. 10.6. Появление красного куба перед камерой

Спустя секунду куб приближается к вам на единицу (рис. 10.7).

Это куб сдвинулся на одну единицу в вашу сторону или, может, вы приблизились на одну единицу к нему? Ориентиров у нас нет, и система координат вне поля зрения. Мы не сможем узнать наверняка, потому что *относительная* позиция куба и камеры одинаковая в обоих случаях (рис. 10.8).

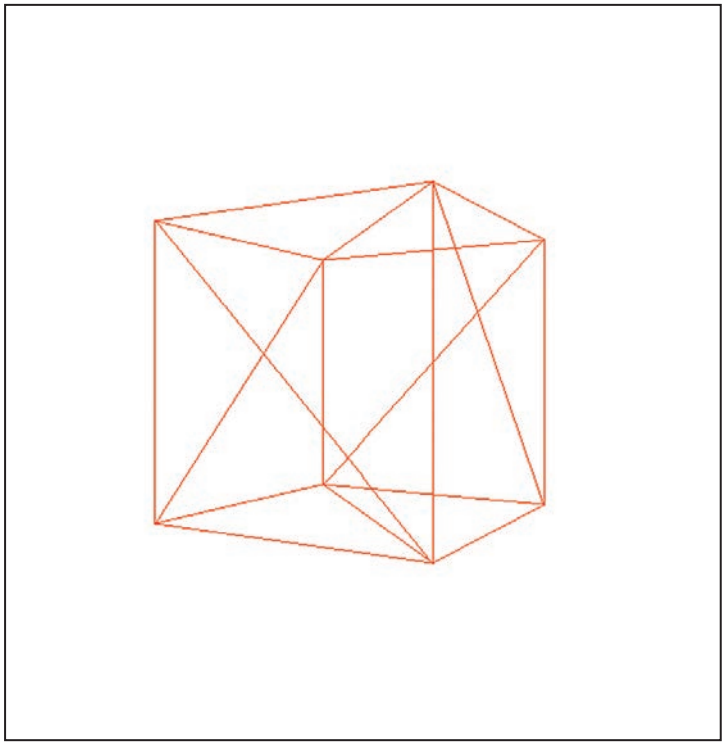


Рис. 10.7. Красный куб перемещается в направлении камеры... точно перемещается?

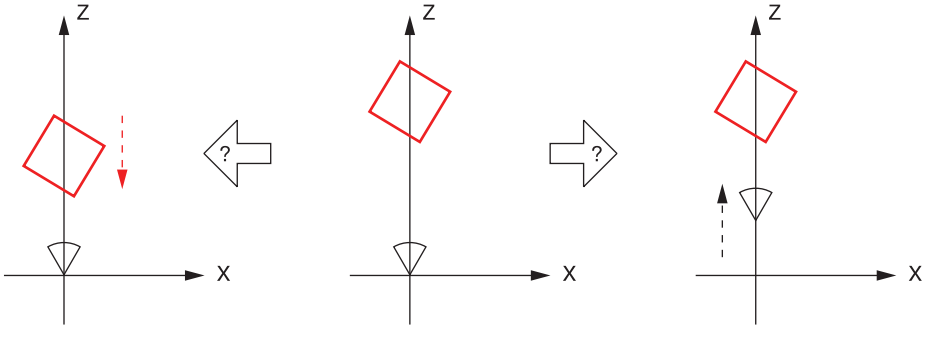


Рис. 10.8. Без системы координат нельзя понять, переместился объект или камера

Теперь куб повернулся вокруг вас по часовой стрелке на угол 45°. Точно он, а не вы повернулись вокруг него по часовой стрелке? Четкого ответа у нас не будет (рис. 10.9).

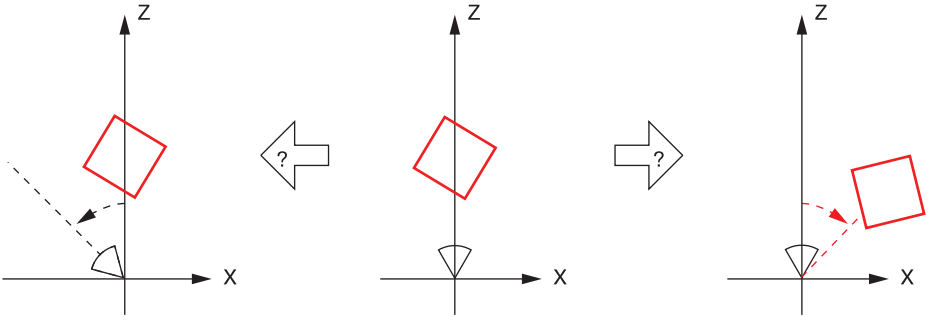


Рис. 10.9. Без системы координат нельзя понять, вращается объект или камера

Благодаря этому эксперименту мы знаем, что нет разницы между перемещением камеры вокруг фиксированной сцены и сохранением ее статичного положения при вращении и переносе сцены вокруг нее.

У такого взгляда есть плюсы. При сохранении позиции камеры в начале координат и ее направленности в сторону $\vec{Z}_+$ можно использовать уравнения проекции из прошлой главы без изменений. Эта система координат камеры называется *пространством камеры*.

Допустим, камера тоже может выполнять преобразование путем переноса и вращения. Для отрисовки сцены с позиции камеры нужно применить к каждой вершине сцены *противоположные* преобразования:

$$V_{\text{translated}} = V_{\text{scene}} - camera.translation;$$

$$V_{\text{cam\_space}} = inverse(camera.rotation) \times V_{\text{translated}};$$

$$V_{\text{projected}} = perspective\_projection(V_{\text{cam\_space}}).$$

Заметьте, что вращение мы представляем с помощью матриц вращения. Подробнее ознакомиться с этой темой можно в приложении «Линейная алгебра».

Матрицы преобразований

Теперь мы можем перемещать по сцене и камеру, и экземпляры моделей. Давайте посмотрим, что происходит с вершиной V_{model} в пространстве модели до ее проецирования в точку холста (cx, cy).

Сначала для перехода от пространства модели к глобальному пространству, применяется преобразование модели:

$$V_{\text{model\_scaled}} = instance.scale \times V_{\text{model}};$$

$$V_{\text{model\_rotated}} = instance.rotation \times V_{\text{model\_scaled}};$$

$$V_{\text{world}} = V_{\text{model\_rotated}} + instance.translation.$$

Чтобы перейти от глобального пространства к пространству камеры, применяем преобразование камеры:

$$V_{\text{translated}} = V_{\text{world}} - camera.translation;$$

$$V_{\text{camera}} = inverse(camera.rotation) \times V_{\text{translated}}.$$

Для получения координат окна просмотра применяем уравнения перспективы:

$$v_x = \frac{V_{\text{camera}}.x d}{V_{\text{camera}}.z};$$

$$v_y = \frac{V_{\text{camera}}.y d}{V_{\text{camera}}.z}.$$

Теперь отображаем координаты окна просмотра в координаты холста:

$$c_x = \frac{v_x c_w}{v_w};\ c_y = \frac{v_y c_h}{v_h}.$$

Как видите, здесь много вычислений и промежуточных значений для каждой вершины. Будет куда интереснее, если мы сможем сжать все это до более компактных размеров.

Давайте выразим преобразования в виде функций, получающих вершину и возвращающих ее преображенный вариант. Пусть C_T и C_R — это перенос и вращение камеры; I_R, I_S и I_T — вращение экземпляра; P — перспективная проекция; а M — отображение из окна просмотра на холст. Если V — это исходная вершина, а V' — точка на холсте, то можно выразить все уравнения выше так:

$$V' = M\left(P\left(C_R^{-1}\left(C_T^{-1}\left(I_T\left(I_R\left(I_S\left(V\right)\right)\right)\right)\right)\right)\right).$$

В идеале нам нужно одно преобразование F, выполняющее всю исходную серию преобразований, но для него есть выражение проще:

$$F = M\,P\,C_R^{-1}\,C_T^{-1}\,I_T\,I_R\,I_S;$$
$$V' = F(V).$$

Сложно найти простой способ представления F. Основное препятствие в том, что мы можем выразить каждое преобразование по-разному. Мы выражаем перенос как сумму точки и вектора, вращение — как умножение матрицы и точки, масштабирование — как умножение вещественного числа и точки, а перспективную проекцию — как умножение и деление вещественных чисел. Но если бы был один способ для выражения всех преобразований и если бы у него был механизм их компоновки, то нужное нам преобразование получить было бы гораздо легче.

Однородные координаты

Рассмотрите выражение $A = (1, 2, 3)$. Здесь A — это 3D-точка или 3D-вектор? Без знания контекста определить невозможно.

Но мы добавим четвертое значение, w, для обозначения A как точки или вектора. Если $w = 0$, то это вектор, если же $w = 1$, то это точка. Итак, точка A выражается как (1, 2, 3, 1), а вектор $\vec{A}$ — как (1, 2, 3, 0).

Поскольку точки и векторы используют общее представление, эти четырехкомпонентные координаты называются *однородными*. Они имеют гораздо больше функций, но эта тема уже выходит за рамки книги. Для нас же они просто удобный инструмент.

Управление точками и векторами, выраженными в однородных координатах, совместимо с их геометрической интерпретацией. Например, вычитание двух точек дает вектор:

$$(8, 4, 2, 1) - (3, 2, 1, 1) = (5, 2, 1, 0).$$

Сложение двух векторов дает другой вектор:

$$(0, 0, 1, 0) + (1, 0, 0, 0) = (1, 0, 1, 0).$$

Точно так же сложение точки с вектором дает точку, умножение вектора на скаляр дает вектор и т. д.

Так что же представляют координаты со значением w, отличным от 0 и 1? Они тоже представляют точки. Любая точка в 3D имеет бесконечное количество представлений в однородных координатах. Решающую роль здесь играет *отношение* между

координатами и значением w. Например, (1, 2, 3, 1) и (2, 4, 6, 2) представляют одну и ту же точку, как и (–3, –6, –9, –3).

Из всех трех представлений то, что содержит $w = 1$, называется *каноническим представлением* точки в однородных координатах. Преобразовать любое другое представление в его каноническую форму или в декартовы координаты несложно:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}.$$

Итак, мы можем преобразовать декартовы координаты в однородные и обратно в декартовы. Но как это поможет найти единое представление для всех преобразований?

Матрица вращений в однородных координатах

Начнем с матрицы вращений. Преобразовать матрицу 3×3 в декартовых координатах в матрицу 4×4 в однородных легко. Координата w точки не должна меняться, поэтому мы добавляем столбец справа, ряд снизу, заполняем их нулями и помещаем в нижний правый элемент 1, сохраняя значение w:

$$\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \rightarrow \begin{pmatrix} A & B & C & 0 \\ D & E & F & 0 \\ G & H & I & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}.$$

Матрица масштабирования в однородных координатах

Получить матрицу масштабирования в однородных координатах тоже несложно. Строится она аналогично матрице вращения:

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} xS_x \\ yS_y \\ zS_z \end{pmatrix} \rightarrow \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} xS_x \\ yS_y \\ zS_z \\ 1 \end{pmatrix}.$$

Матрица для переноса в однородных координатах

С матрицами вращения и масштабирования все просто. Они уже были выражены как матричные умножения в декартовых координатах, и нам оставалось лишь добавить 1 для сохранения координаты w. Но что делать с переносом, который мы выразили в виде дополнения к декартовым координатам?

Мы ищем такую матрицу 4×4, чтобы:

$$\begin{pmatrix} T_x \\ T_y \\ T_z \\ 0 \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}.$$

Сначала сосредоточимся на получении $x + T_x$. Это значение — результат умножения первого ряда матрицы на точку, то есть:

$$\begin{pmatrix} A & B & C & D \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = x + T_x.$$

Расширив это векторное умножение, получим:

$$Ax + By + Cz + D = x + T_x.$$

Так можно сделать вывод, что $A = 1$, $B = C = 0$ и $D = T_x$.

Следуя тому же принципу для остальных координат, мы получаем такое выражение матрицы для переноса:

$$\begin{pmatrix} T_x \\ T_y \\ T_z \\ 0 \end{pmatrix} + \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}.$$

Матрица проекций в однородных координатах

Суммы и умножения легко выражаются как умножения матриц и векторов, ведь они подразумевают суммы и умножения. Но уравнения перспективной проекции содержат деление на z. Как можно выразить его?

Можно предположить, что деление на z — это то же, что умножение на $1/z$, и попробовать решить эту задачу, добавив $1/z$ в матрицу. Но *какую* координату z мы

тогда туда поместим? Нужно, чтобы эта матрица проекций работала для *каждой* входной точки. Значит, жесткое прописывание координаты z *любой* точки не даст нам желаемого.

К счастью, в однородных координатах есть деление на координату w в процессе обратного преобразования в декартовы координаты. Если мы сможем сделать так, чтобы координата z исходной точки выступала как координата w «спроецированной», то после обратного преобразования этой точки в декартовы координаты получим спроецированные x и y:

$$\begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} xd \\ yd \\ z \end{pmatrix} \rightarrow \begin{pmatrix} \frac{xd}{z} \\ \frac{yd}{z} \end{pmatrix}.$$

Заметьте, это матрица 3 × 4. Ее можно умножать на четырехэлементный вектор (преобразованную 3D-точку в однородных координатах), и она будет давать трехэлементный (спроецированную 2D-точку в однородных координатах, которая потом через деление на w преобразуется в двухмерные декартовы координаты). Это дает нам искомые значения x' и y'. Недостает здесь элемента z', который по определению равен d.

По тому же принципу, что и для получения матрицы переносов, можно выразить перспективную проекцию так:

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} xd \\ yd \\ z \end{pmatrix} \rightarrow \begin{pmatrix} \frac{xd}{z} \\ \frac{yd}{z} \end{pmatrix}.$$

Матрица отображения из окна просмотра на холст в однородных координатах

Последний шаг — это отображение точки, спроецированной на окно просмотра, на холст. Это просто двухмерное преобразование масштабирования с $S_x = \frac{c_w}{v_w}$ и $S_y = \frac{c_h}{v_h}$. В итоге матрица получается вот такая:

$$\begin{pmatrix} \frac{c_w}{v_w} & 0 & 0 \\ 0 & \frac{c_h}{v_h} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{x \cdot c_w}{v_w} \\ \frac{y \cdot c_h}{v_h} \\ z \end{pmatrix}.$$

Ее без проблем можно совместить с матрицей проекций для получения простой матрицы переноса из 3D на холст:

$$\begin{pmatrix} \frac{d \cdot c_w}{v_w} & 0 & 0 & 0 \\ 0 & \frac{d \cdot c_h}{v_h} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ & & & \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{x \cdot d \cdot c_w}{v_w} \\ \frac{y \cdot d \cdot c_h}{v_h} \\ z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} \left(\frac{x \cdot d}{z}\right) & \left(\frac{c_w}{v_w}\right) \\ \left(\frac{y \cdot d}{z}\right) & \frac{c_w}{v_w} \end{pmatrix}.$$

Возвращение к матрице преобразований

Мы проделали хорошую работу. Теперь можно выразить любое нужное преобразование для превращения вершины модели V в пиксель холста V' как матрицу. Можно еще скомпоновать эти преобразования перемножением их соответствующих матриц. В то же время можно выразить всю последовательность преобразований в виде одной матрицы:

$$F = M\,P\,C_R^{-1}\,C_T^{-1}\,I_T\,I_R\,I_S.$$

Теперь для преобразования вершины нужно просто вычислить следующее умножение матрицы на точку:

$$V' = F\,V.$$

Можно даже разделить это преобразование на три части:

$$M_{\text{projection}} = M\,P;$$
$$M_{\text{camera}} = C_R^{-1}\,C_T^{-1};$$
$$M_{\text{model}} = I_T\,I_R\,I_S;$$
$$M = M_{\text{projection}}\,M_{\text{camera}}\,M_{\text{model}}.$$

Эти матрицы не нужно заново вычислять для каждой вершины (в этом и смысл использования матрицы). Матричное умножение ассоциативно, и можно повторно использовать неизменяющиеся части выражения.

$M_{\text{projection}}$ должна меняться редко. Она зависит только от размера окна просмотра и холста. Размер холста может изменяться, когда приложение переходит из полноэкранного режима в оконный. Размер окна просмотра меняется только при изменении поля зрения камеры. А это происходит редко.

M_{camera} может изменяться каждый кадр. Она зависит от позиции и ориентации камеры. Если камера движется или поворачивается, это значение нужно снова вычислять. Но после вычисления оно постоянно для каждого отрисовываемого в кадре объекта, так что чаще одного раза в кадр его вычислять не нужно.

M_{model} будет отличаться для каждого экземпляра на сцене. Но для неподвижных (деревьев и строений) она будет постоянной. В таком случае можно вычислить ее только один раз и сохранить в самой сцене. Для движущихся объектов (машин в гоночной игре) ее нужно вычислять при каждом их перемещении (скорее всего, каждый кадр).

В листинге 10.5 приводится высокоуровневое представление псевдокода рендеринга сцены.

Листинг 10.5. Алгоритм для рендеринга сцены с помощью матриц преобразований

```
RenderModel(model, transform) {
    projected = []
    for V in model.vertices {
        projected.append(ProjectVertex(transform * V))
    }
    for T in model.triangles {
        RenderTriangle(T, projected)
    }
}

RenderScene() {
    M_camera = MakeCameraMatrix(camera.position, camera.orientation)

    for I in scene.instances {
        M = M_camera * I.transform
        RenderModel(I.model, M)
    }
}
```

Теперь можно отрисовать сцену с несколькими экземплярами разных моделей, которые можно перемещать и вращать. Теперь мы можем менять и положение камеры в сцене. На рис. 10.10 показаны два экземпляра нашей модели куба, каждый с разным преобразованием (включая вращение и перенос), отрисованные с позиции передвинутой и повернутой камеры.

Живую реализацию алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/transforms-demo.

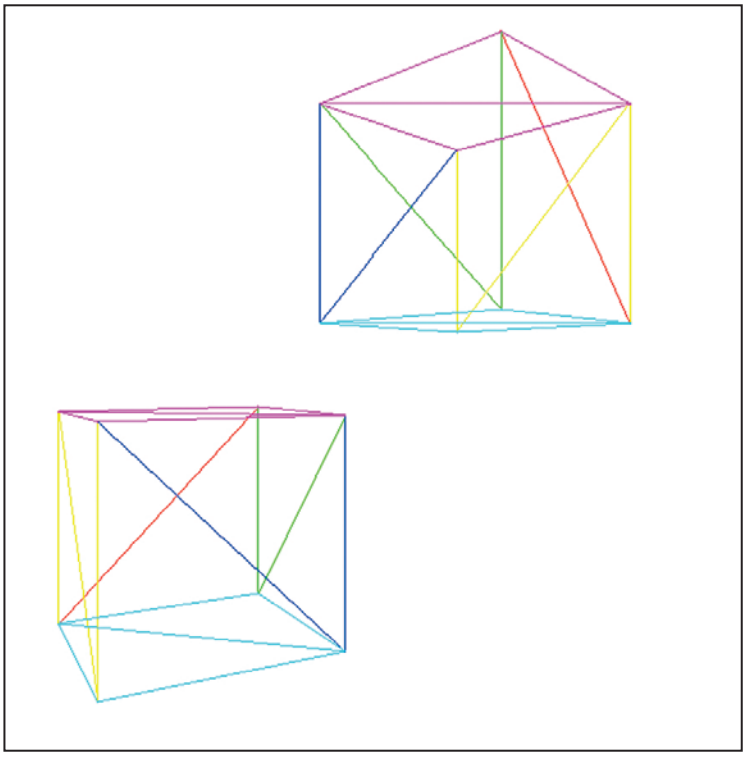


Рис. 10.10. Сцена с двумя экземплярами одной модели куба, имеющей различные преобразования, отрисованные с позиции смещенной и повернутой камеры

Итоги главы

В этой главе мы рассмотрели много всего. Сначала вы научились представлять собранные из треугольников модели. Дальше смогли применить уравнение перспективной проекции из главы 9 к моделям целиком, чтобы переходить от абстрактной 3D-модели к ее представлению на экране.

После вы научились создавать в сцене множество экземпляров модели без множества копий ее самой. Далее мы смогли отбросить одно из ограничений: камера больше не привязана к началу координат и не должна указывать в сторону $\vec{Z}_+$.

В конце вы узнали, как представлять все преобразования для вершины в виде умножения матриц в однородных координатах. Так мы уменьшили объем вычислений

для рендеринга сцены, сжав множество последовательных преобразований всего в три матрицы. Одну — для перспективной проекции и отображения из окна просмотра на холст, другую — для преобразования экземпляра и еще одну — для преобразования камеры.

Это расширило наши возможности представления объектов на сцене и позволило перемещать по ней камеру. Но у нас еще осталось два важных ограничения. Во-первых, из-за перемещения камеры объекты могут оказаться позади нее, а это приведет к проблемам. Во-вторых, отрисовка выглядит не очень презентабельно.

Из практических соображений дальше мы не станем использовать полную матрицу проекций. Вместо этого мы будем задействовать преобразования модели и камеры по отдельности и конвертировать их результаты обратно в декартовы координаты:

$$x' = \frac{x \cdot d \cdot c_w}{z \cdot v_w};$$

$$y' = \frac{y \cdot d \cdot c_h}{z \cdot v_h}.$$

Благодаря этому мы сможем делать в 3D дополнительные операции, которые нельзя выразить в виде матричных преобразований до проецирования точек.

В следующей главе мы поработаем с объектами, которые не должны быть видимы, а оставшуюся часть книги посвятим улучшению внешнего вида отрисованных объектов.

11

Отсечение

В предыдущих главах мы разработали уравнения и алгоритмы для преобразования трехмерного определения сцены в двухмерные формы, которые можно отрисовать на холсте. Мы создали структуру сцены, позволяющую определять 3D-модели и размещать их экземпляры на сцене. Мы даже создали алгоритм, позволяющий отрисовывать сцену с любой точки обзора.

Теперь, когда мы умеем вращать и перемещать камеру, мы снова сталкиваемся с проблемой: уравнения перспективной проекции работают ожидаемо только для точек перед ней. Чтобы этого избежать, мы рассмотрим некоторые важные техники. Вы научитесь определять точки, треугольники и целые объекты позади камеры и обрабатывать их.

Обзор процесса отсечения

В главе 9 у нас получились такие уравнения:

$$P'_x = \frac{P_x d}{P_z};$$

$$P'_y = \frac{P_y d}{P_z}.$$

Опасно делить на P_z — это может спровоцировать деление на нуль. Более того, точки позади камеры имеют отрицательные значения Z, которые мы пока не можем правильно обработать. Даже если точки будут перед камерой, но очень близко, объекты могут сильно исказиться.

Во избежание всех этих неприятностей мы не станем отрисовывать что-то позади плоскости проекции $Z = d$. Эта *отсекающая плоскость* поможет определить, где находится точка: *внутри* или *снаружи отсекаемого объема*, то есть подмножества пространства, которое видно из камеры. Получается, отсекаемый объем — «все, что находится перед $Z = d$», и мы будем отрисовывать только части сцены внутри него.

Отсекаемый объем

Используя только одну плоскость отсечения, чтобы не отрисовывать объекты за камерой, мы добьемся приемлемого результата, но это будет недостаточно эффективно. Некоторые объекты, находящиеся перед камерой, все еще могут быть невидимы. Например, при проецировании объекта перед плоскостью проекции, но смещенного далеко вправо, он будет отображен вне окна просмотра и в результате окажется невидим (рис. 11.1).

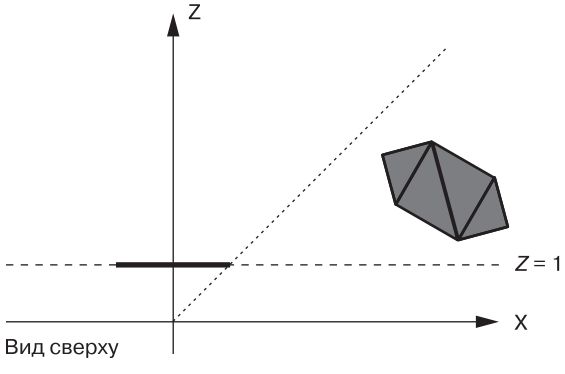


Рис. 11.1. Объект перед плоскостью проекции, но отображается вне окна просмотра

Все усилия для проецирования такого объекта будут потрачены впустую. Лучше просто проигнорировать его.

Для этого нам понадобятся камера и все стороны окна просмотра. С их помощью мы сможем определить дополнительные плоскости, чтобы включить в сцену только то, что должно быть видимым в окне просмотра (рис. 11.2).

Каждая из этих плоскостей делит пространство на две части — *полупространства*. Внутреннее полупространство — это все, что находится перед плоскостью,

а внешнее — все, что позади нее. Внутренняя часть отсекаемого объема, который мы определяем, — это *пересечение* внутренних полупространств, определенных каждой плоскостью отсечения. В нашем случае отсекаемый объем выглядит как бесконечно высокая пирамида с отрубленным верхом.

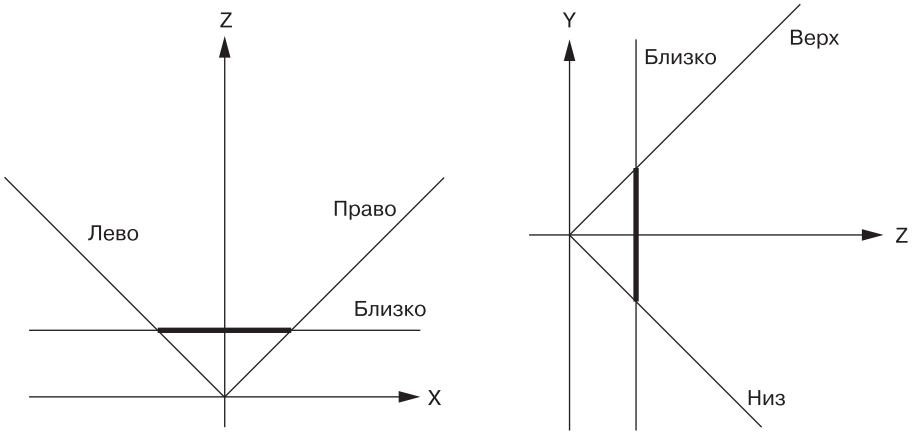


Рис. 11.2. Пять плоскостей, определяющих отсекаемый объем

Значит, для отсечения сцены по отсекаемому объему нужно просто последовательно обрезать ее по каждой плоскости, определяющей этот объем. Все оставшиеся после отсечения по одной плоскости геометрические объекты повторно отсекаются по оставшимся. Оставшиеся геометрические объекты — результат отсечения сцены по отсекаемому объему.

Теперь разберем, как отсекать сцену по каждой плоскости отсечения.

Отсечение сцены по плоскости

Рассмотрим сцену с несколькими объектами, каждый из которых состоит из четырех треугольников (рис. 11.3).

Чем меньше операций мы выполняем, тем быстрее рендеринг. Отсечение сцены по плоскости выполняется в несколько этапов. На каждом все возможные геометрические элементы разделяются на *принятые* и *отброшенные*. Все зависит от того, находится элемент внутри полупространства, определяемого плоскостью отсечения, или нет. Элементы, которые классифицировать нельзя, рассматриваются детальнее на следующем этапе.

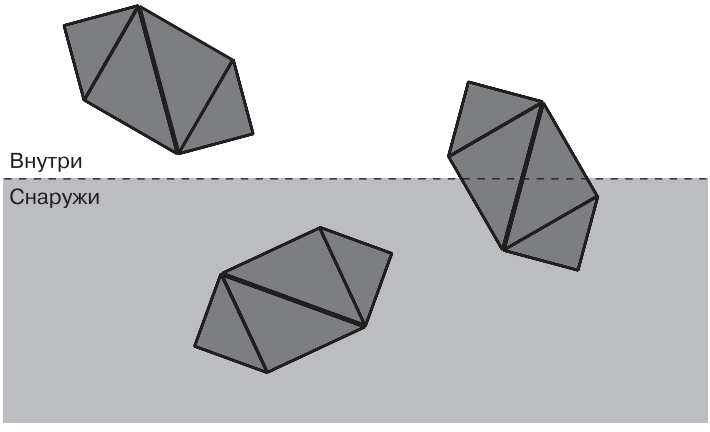


Рис. 11.3. Сцена с тремя объектами

На первом этапе объект классифицируется целиком. Если он полностью внутри отсекаемого объема, он принимается (зеленый на рис. 11.4). Если же он полностью вне — отбрасывается (рис. 11.4).

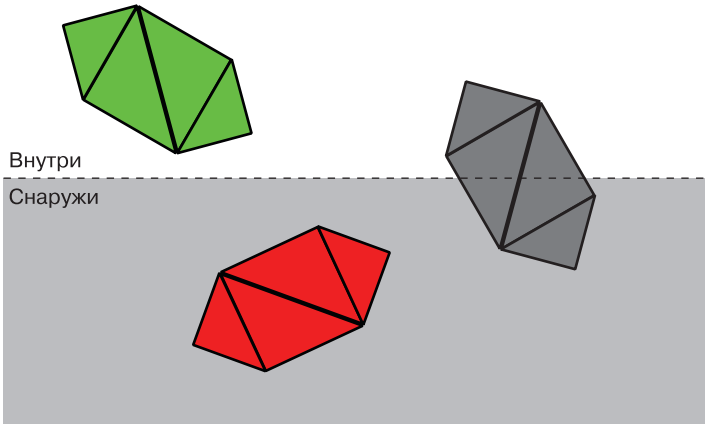


Рис. 11.4. Отсечение на уровне объектов. Зеленый принимается, красный отбрасывается, а серому нужна дополнительная обработка

Если объект нельзя полностью принять или отбросить, он перемещается на следующий этап, где все его треугольники классифицируются отдельно. Если треугольник будет весь внутри отсекаемого объема, он принимается, если наоборот — отбрасывается (рис. 11.5).

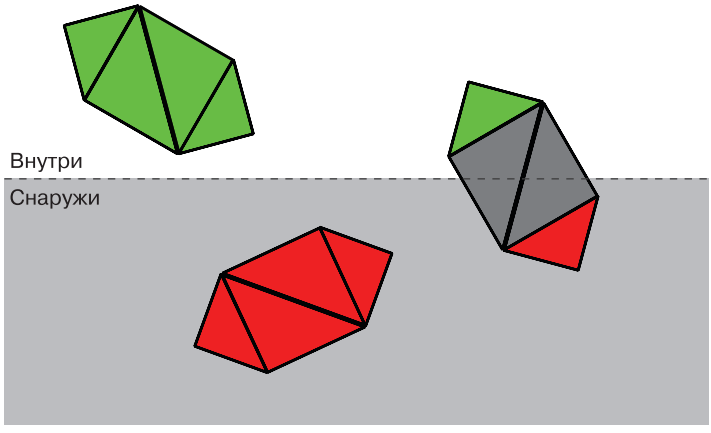


Рис. 11.5. Отсечение на уровне треугольников. Каждый треугольник правого крайнего объекта принимается, отбрасывается или требует дополнительной обработки

Каждый треугольник, который не был отброшен или принят, нужно отсечь. Исходный треугольник удаляется, и один-два новых добавляются для покрытия его части, находящейся внутри отсекаемого объема (рис. 11.6).

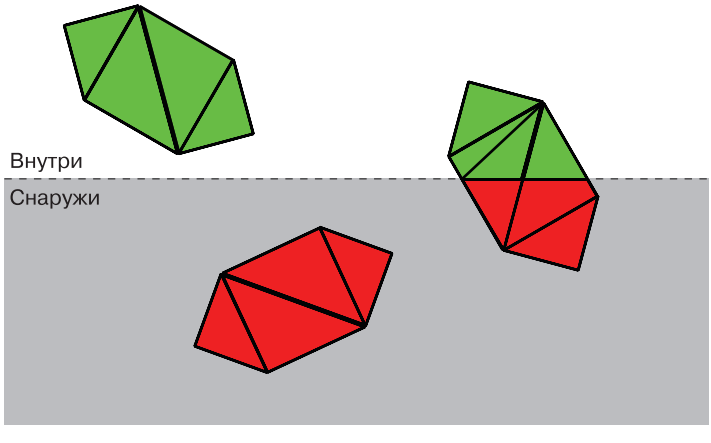


Рис. 11.6. Отсечение на уровне вершин. Каждый треугольник, частично входящий в отсекаемый объем, делится на один или два треугольника, полностью находящихся в этом объеме

Мы разобрались с принципом работы отсечения. Теперь выработаем математические приемы и алгоритмы для создания рабочей реализации.

Определение плоскостей отсечения

Начнем с уравнения плоскости проекции $Z = d$. Ее мы будем использовать как плоскость отсечения. Это уравнение легко визуализировать, но для наших целей оно представлено не в самом удобном виде.

Основное уравнение для 3D-плоскости — $Ax + By + Cz + D = 0$; оно означает, что точка $P = (x, y, z)$ будет удовлетворять этому уравнению тогда и только тогда, когда P находится на плоскости. Если сгруппировать коэффициенты (A, B, C) в вектор $\vec{N}$, можно переписать это уравнение как $\langle \vec{N}, P \rangle + D = 0$.

Если $\langle \vec{N}, P \rangle + D = 0$, тогда $k\langle \vec{N}, P \rangle + kD = 0$ для любого значения k. Мы можем использовать $k = 1/|\vec{N}|$, умножить исходное уравнение и получить новое: $\langle \vec{N}', P \rangle + D' = 0$, в котором $\vec{N}'$ — это единичный вектор. Значит, любую заданную плоскость можно представить уравнением $\langle \vec{N}, P \rangle + D = 0$, где $\vec{N}$ — единичный вектор, а D — вещественное число.

Это очень удобная формулировка: $\vec{N}$ оказывается нормалью плоскости, а $-D$ — *расстоянием со знаком* от начала координат до этой плоскости. На деле для любой точки P выражение $\langle \vec{N}, P \rangle + D$ будет расстоянием со знаком от плоскости до P; *расстояние* = 0 — это особый случай, когда P находится на плоскости.

Если $\vec{N}$ — это нормаль плоскости, то $-\vec{N}$ *тоже*, значит, выбираем $\vec{N}$ так, чтобы она указывала «внутрь» отсекаемого объема. Для плоскости $Z = d$ мы выбираем нормаль (0, 0, 1), указывающую вперед относительно камеры. Точка $(0, 0, d)$ находится на плоскости, поэтому она должна удовлетворять уравнению плоскости. Решим его для D:

$$\langle \vec{N}, P \rangle + D = \langle (0, 0, 1), (0, 0, d) \rangle + D = d + D = 0.$$

А отсюда мы сразу же получим $D = -d$.

Можно было получить $D = -d$ прямо из исходного уравнения плоскости $Z = d$, переписав его как $Z - d = 0$. Но этот метод мы применим для получения уравнений оставшихся плоскостей отсечения.

Мы знаем, что у всех дополнительных плоскостей $D = 0$ (так как они все проходят через начало координат). Поэтому нам нужно только определить их нормали. Для упрощения расчетов выбираем поле зрения (FOV) 90°, то есть плоскости будут под углом 45°.

Рассмотрим левую плоскость отсечения. Ее нормаль имеет направление (1, 0, 1) (это значит 45° вправо и вперед). Длина этого вектора $\sqrt{2}$, поэтому, если его нормализовать, мы получим $\left(\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right)$.

Получаем уравнение левой плоскости отсечения:

$$\langle N, P\rangle + D = \left\langle \left(\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right), P \right\rangle = 0.$$

Нормали для правой, нижней и верхней плоскостей отсечения — $\left(\frac{-1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right)$, $\left(0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ и $\left(0, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$ соответственно. Для вычисления плоскости отсечения для любого произвольного FOV перейдем к тригонометрии.

Наш отсекаемый объем определяется пятью плоскостями:

$$\langle (0, 0, 1), P\rangle - d = 0; \quad \text{(передняя)}$$
$$\left\langle \left(\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right), P \right\rangle = 0; \quad \text{(левая)}$$
$$\left\langle \left(\frac{-1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right), P \right\rangle = 0; \quad \text{(правая)}$$
$$\left\langle \left(0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right), P \right\rangle = 0; \quad \text{(нижняя)}$$
$$\left\langle \left(0, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right), P \right\rangle = 0. \quad \text{(верхняя)}$$

Теперь рассмотрим отсечение геометрических элементов по плоскости детальнее.

Отсечение объектов

Предположим, что помещаем каждую модель в описывающую сферу (рис. 11.7). Назовем ее *ограничивающей сферой* объекта. Вычислить ее гораздо сложнее, чем может показаться. Эта тема выходит за рамки книги. Но грубое представление

можно получить, вычислив центр этой сферы через усреднение координат всех вершин модели и определив радиус как расстояние от этого центра до максимально удаленной вершины.

В любом случае предположим, что знаем центр C и радиус r сферы, охватывающей каждую модель.

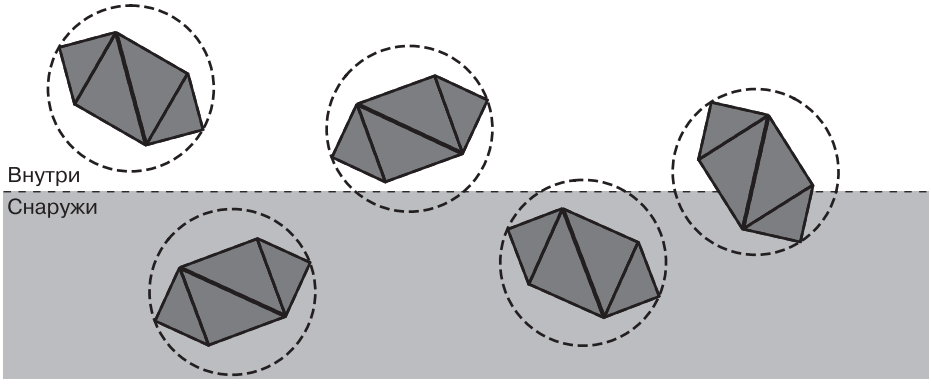


Рис. 11.7. Сцена с несколькими объектами и ограничивающими их сферами

Разделим пространственное отношение между сферой и плоскостью.

- **Сфера полностью перед плоскостью.** В этом случае весь объект принимается и дополнительного отсечения не нужно (но он все еще может быть отсечен по другой плоскости). Смотрим пример на рис. 11.8.

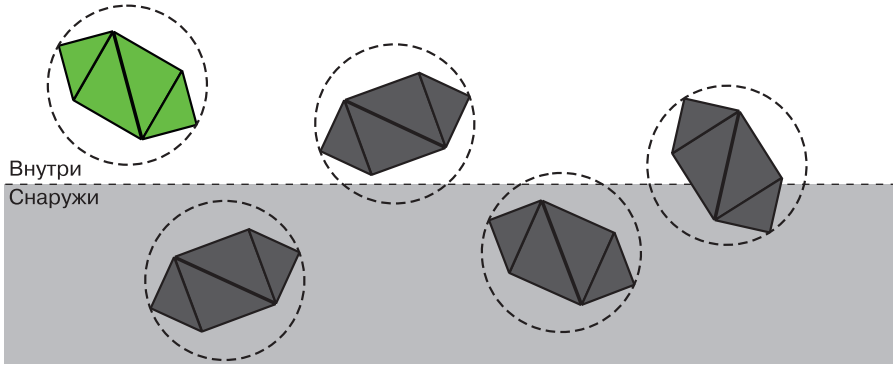


Рис. 11.8. Зеленый объект принимается

- **Сфера полностью позади плоскости.** В этом случае весь объект отбрасывается и дополнительного отсечения не нужно (никакая часть этого объекта уже не попадет в отсекаемый объем). Пример приведен на рис. 11.9.

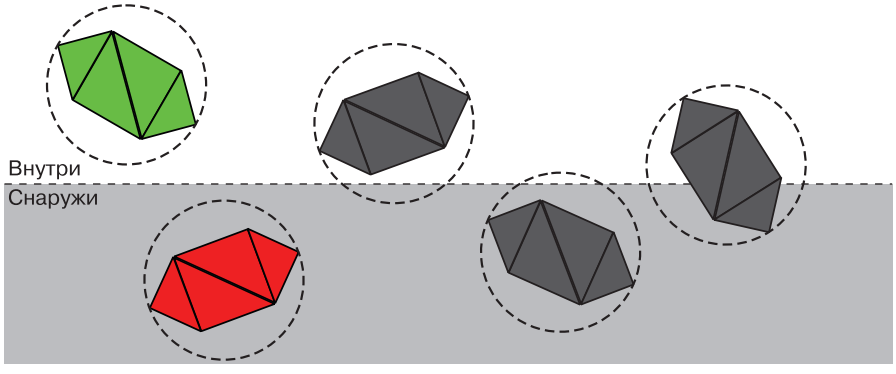


Рис. 11.9. Красный объект отбрасывается

- **Плоскость пересекает сферу.** У нас недостаточно информации об объекте — он может быть внутри полностью или частично. Нужно перейти к следующему этапу и выполнить отсечение модели треугольник за треугольником. Смотрим пример на рис. 11.10.

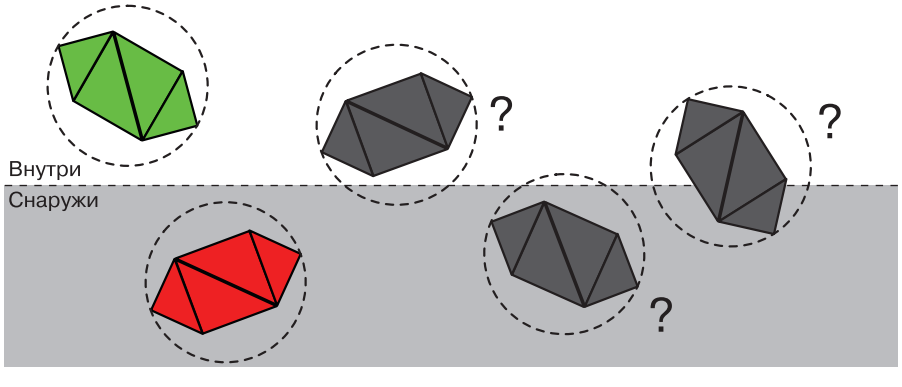


Рис. 11.10. Серые объекты нельзя полностью принять или отбросить

Как же работает это деление на категории? Мы выбрали такой способ выражения плоскостей отсечения, что подстановка любой точки в уравнение плоскости дает нам расстояние со знаком от этой точки до плоскости. Мы можем вычислить подобное

расстояние d от центра ограничивающей сферы до плоскости. Значит, если $d > r$, тогда сфера находится перед плоскостью, а если $d < -r$ — сфера позади нее. В противном случае $|d| < r$ (плоскость пересекает сферу). На рис. 11.11 показаны все три случая.

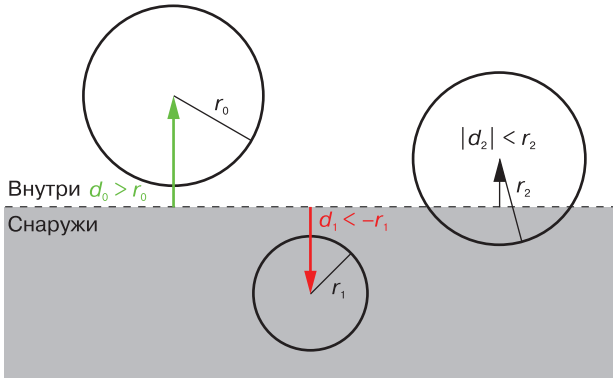


Рис. 11.11. Расстояние со знаком от центра сферы до плоскости отсечения сообщает нам, где находится сфера по отношению к плоскости

Отсечение треугольников

Если предыдущей проверки недостаточно для определения позиции объекта относительно плоскости отсечения, нужно отсечь по ней каждый его треугольник.

Можно разделить вершины в зависимости от их положения относительно плоскости, рассмотрев их расстояние со знаком до нее. Если расстояние равно нулю или положительно, значит, вершина находится перед плоскостью отсечения, если наоборот — позади (рис. 11.12).

Для каждого треугольника есть четыре возможные классификации.

- **Три вершины впереди.** В этом случае весь треугольник находится перед плоскостью отсечения, значит, мы принимаем его и дополнительного отсечения по этой плоскости не нужно.
- **Три вершины позади.** Весь треугольник находится за плоскостью, значит, он отбрасывается и дополнительного отсечения не нужно.
- **Одна вершина впереди.** Пусть A будет вершиной треугольника ABC, находящейся перед плоскостью. В этом случае мы отбрасываем ABC и добавляем новый треугольник $AB'C'$, где B' и C' — это пересечения AB и AC с плоскостью отсечения (рис. 11.13).

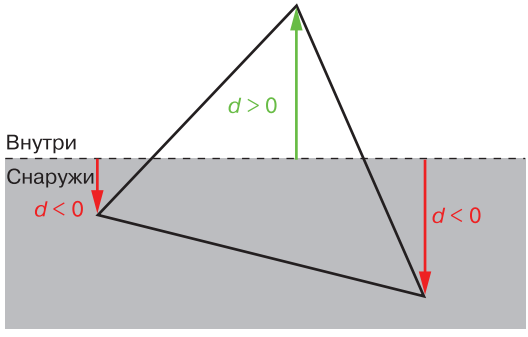


Рис. 11.12. Расстояние со знаком от вершины до плоскости отсечения сообщает, находится вершина перед плоскостью или позади нее

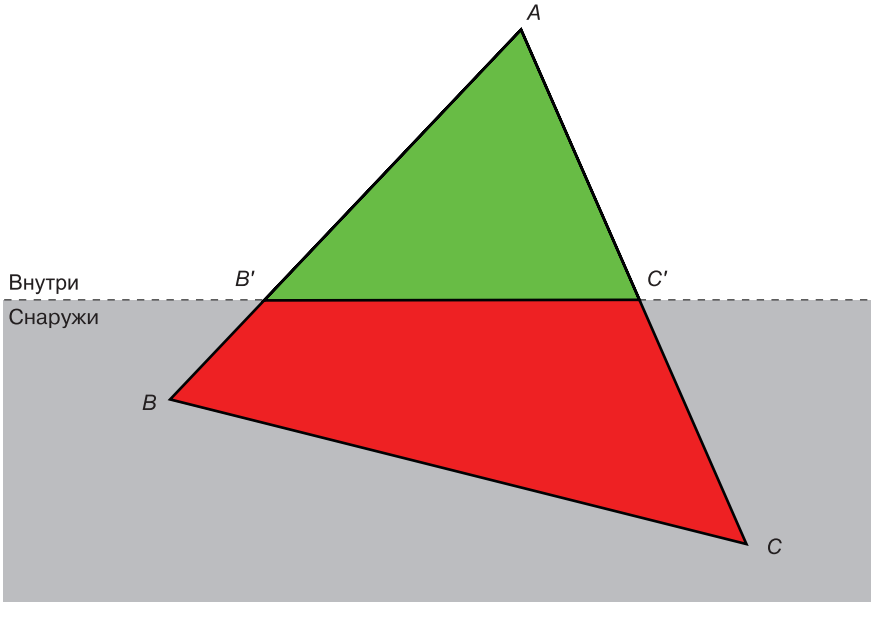


Рис. 11.13. Треугольник ABC с одной вершиной внутри и двумя снаружи отсекаемого объема замещается треугольником $AB'C'$

- **Две вершины впереди.** Пусть A и B будут вершинами треугольника ABC, находящимися перед плоскостью. Мы отбрасываем ABC и добавляем два новых треугольника: ABA' и $A'BB'$, где A' и B' — это пересечения AC и BC с плоскостью отсечения (рис. 11.14).

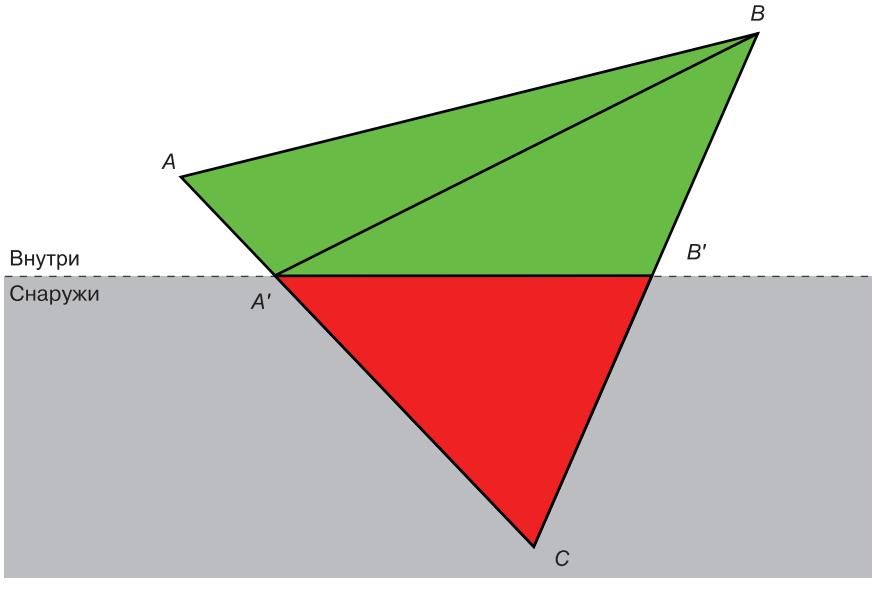


Рис. 11.14. Треугольник *ABC* с одной вершиной снаружи и двумя внутри отсекаемого объема замещается двумя треугольниками *ABA'* и *A'BB'*

Пересечение отрезка с плоскостью

Чтобы сделать отсечение, описанное выше, нужно вычислить пересечение сторон треугольника с отсекающей плоскостью.

У нас есть плоскость отсечения, заданная уравнением $\langle N, P\rangle + D = 0$. Сторону AB треугольника можно выразить через параметрическое уравнение как $P = A + t(B - A)$ для $0 \le t \le 1$. Для вычисления значения области пересечения t мы заменяем P в уравнении плоскости на параметрическое уравнение этого отрезка:

$$\langle N, P\rangle + D = 0;$$

$$P = A + t(B - A);$$

$$\Rightarrow \langle N, A + t(B - A)\rangle + D = 0.$$

Используем линейные свойства скалярного произведения:

$$\langle N, A\rangle + t\langle N, B - A\rangle + D = 0.$$

Решаем для t:

$$t = \frac{-D - \langle N, A\rangle}{\langle N, B - A\rangle}.$$

Мы знаем, что решение существует всегда, так как AB пересекает плоскость. Математически $\langle N, B - A \rangle$ не может быть равно нулю. Это будет подразумевать, что отрезок и нормаль перпендикулярны друг другу. А это значит, что у отрезка не будет пересечения с плоскостью.

После вычисления t пересечение Q будет:

$$Q = A + t(B - A).$$

Если исходные вершины несут дополнительные атрибуты (например, значение интенсивности h, которое мы использовали в главе 7), то нужно вычислить их значения для новых вершин.

В уравнении выше t — часть отрезка AB, в котором происходит пересечение. Пусть αA и αB будут значениями некоторого атрибута α в точках A и B. Предположим, что этот атрибут изменяется по AB линейно, тогда αQ можно вычислить как:

$$\alpha_Q = \alpha_A + t(\alpha_B - \alpha_A).$$

Теперь у нас есть все алгоритмы и уравнения для реализации конвейера отсечения.

Псевдокод отсечения

Пора писать высокоуровневый псевдокод для конвейера отсечения. Здесь мы будем следовать выработанному нами подходу сверху вниз.

Для отсечения сцены мы отсекаем каждый экземпляр на ней (листинг 11.1).

Листинг 11.1. Алгоритм для отсечения сцены по набору отсекающих плоскостей

```
ClipScene(scene, planes) {
    clipped_instances = []
    for I in scene.instances {
        clipped_instance = ClipInstance(I, planes)
        if clipped_instance != NULL {
            clipped_instances.append(clipped_instance)
        }
    }
    clipped_scene = Copy(scene)
    clipped_scene.instances = clipped_instances
    return clipped_scene
}
```

Чтобы отсечь экземпляр, мы принимаем его, отвергаем или отсекаем каждый из его треугольников, опираясь на ограничивающую его сферу (листинг 11.2).

Листинг 11.2. Алгоритм для отсечения экземпляра по набору отсекающих плоскостей

```
ClipInstance(instance, planes) {
    for P in planes {
        instance = ClipInstanceAgainstPlane(instance, plane)
        if instance == NULL {
            return NULL
        }
    }
    return instance
}

ClipInstanceAgainstPlane(instance, plane) {
    d = SignedDistance(plane, instance.bounding_sphere.center)
    if d > r {
        return instance
    } else if d < -r {
        return NULL
    } else {
        clipped_instance = Copy(instance)
        clipped_instance.triangles =
            ClipTrianglesAgainstPlane(instance.triangles, plane)

        return clipped_instance
    }
}
```

И наконец, для отсечения треугольника мы его принимаем, отвергаем или разбиваем на один-два других треугольника, в зависимости от количества вершин перед плоскостью отсечения (листинг 11.3).

Листинг 11.3. Алгоритм для отсечения набора треугольников по отсекающей плоскости

```
ClipTrianglesAgainstPlane(triangles, plane) {
    clipped_triangles = []
    for T in triangles {
        clipped_triangles.append(ClipTriangle(T, plane))
    }
    return clipped_triangles
}

ClipTriangle(triangle, plane) {
    d0 = SignedDistance(plane, triangle.v0)
    d1 = SignedDistance(plane, triangle.v1)
    d2 = SignedDistance(plane, triangle.v2)

    if {d0, d1, d2} are all positive {
        return [triangle]
    } else if {d0, d1, d2} are all negative {
        return []
    } else if only one of {d0, d1, d2} is positive {
        let A be the vertex with a positive distance
        compute B' = Intersection(AB, plane)
```

```
        compute C' = Intersection(AC, plane)
        return [Triangle(A, B', C')]
    } else /* only one of {d0, d1, d2} is negative */ {
        let C be the vertex with a negative distance
        compute A' = Intersection(AC, plane)
        compute B' = Intersection(BC, plane)
        return [Triangle(A, B, A'), Triangle(A', B, B')]
    }
}
```

Вспомогательная функция `SignedDistance` просто подставляет координаты точки в уравнение плоскости (листинг 11.4).

Листинг 11.4. Функция для вычисления расстояния со знаком от плоскости до точки

```
SignedDistance(plane, vertex) {
    normal = plane.normal
    return (vertex.x * normal.x)
        + (vertex.y * normal.y)
        + (vertex.z * normal.z)
        + plane.D
}
```

Живую реализацию этого алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/clipping-demo.

Место отсечения в конвейере рендеринга

Порядок глав в книге не соответствует этапам в конвейере рендеринга. Из введения вы уже знаете, что главы упорядочены так, чтобы вы быстрее и легче усвоили информацию.

Отсечение — это 3D-операция. Она получает 3D-объекты сцены и вычисляет пересечение сцены и отсекаемого объема. Поэтому отсечение должно происходить после размещения объектов в сцене (использовать вершины по завершении трансформаций модели и камеры), но до перспективной проекции.

Техники из этой главы надежные, но очень обобщенные. Чем больше вы заранее знаете о сцене, тем эффективнее будет ее отсечение. Например, многие игры предварительно обрабатывают свои уровни, добавляя в них информацию о видимости. Если поделить сцену на комнаты, то можно сделать табличный список, перечисляющий, какие комнаты видимы из любой другой заданной комнаты. После, во время рендеринга этой сцены, нужно будет только определить, в какой комнате камера. Так можно сэкономить достаточно вычислительных ресурсов, игнорируя все остальные комнаты, отмеченные как невидимые из нее. Но такой подход увеличит время препроцессинга и сделает сцену более жесткой. Если вы заинтересовались, то почитайте о двоичном разбиении пространства и портальной системе.

Итоги главы

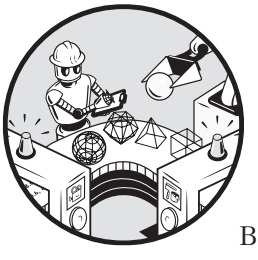
В этой главе мы наконец устранили проблему, из-за которой правильно проецироваться могли только вершины перед камерой. Для этого мы конкретно определили, что значит «находящееся перед камерой» — это все, расположенное внутри отсекаемого объема, который мы определяем с помощью пяти плоскостей.

Мы разработали уравнения и алгоритмы для вычисления геометрического пересечения между сценой и отсекаемым объемом. В итоге мы можем взять сцену и удалить все, что нельзя спроецировать на окно просмотра. Это не только исключает случаи, которые невозможно обработать уравнениями перспективной проекции, но и экономит вычислительные ресурсы, удаляя геометрические элементы, которые спроецировались бы вне окна просмотра.

Но даже после этого все еще можно получить детали, которые *могут* быть видимы на конечном холсте, но *не будут*. В следующей главе мы найдем способы разобраться и с этой ситуацией.

12

Удаление скрытых поверхностей



Теперь мы можем отрисовывать любую сцену с любой точки обзора, но итоговое изображение выглядит примитивно, как схема набора объектов, а не сами объекты.

В оставшихся главах мы сосредоточимся на улучшении визуала отрисованной сцены. К концу этой главы вы сможете создавать объекты, выглядящие целостно без намека на «проволочность». У нас уже есть алгоритм для отрисовки закрашенных треугольников, но корректно использовать его в 3D-сцене будет непросто.

Рендеринг сплошных объектов

Как же нам отрисовать объекты так, чтобы они выглядели монолитными? Первым на ум приходит использование функции `DrawFilledTriangle`, которую мы создали в главе 7 для отрисовки каждого треугольника объектов, заполняемого случайным цветом (рис. 12.1).

Фигуры с рис. 12.1 не особо похожи на кубы, как считаете? Если взглянуть поближе, то проблема очевидна: части задних граней куба нарисованы поверх передних. Причина в том, что мы слепо рисуем двухмерные треугольники на холсте в «случайном порядке». Другими словами, в порядке их определения в списке `Triangles` модели, без учета пространственного отношения между ними.

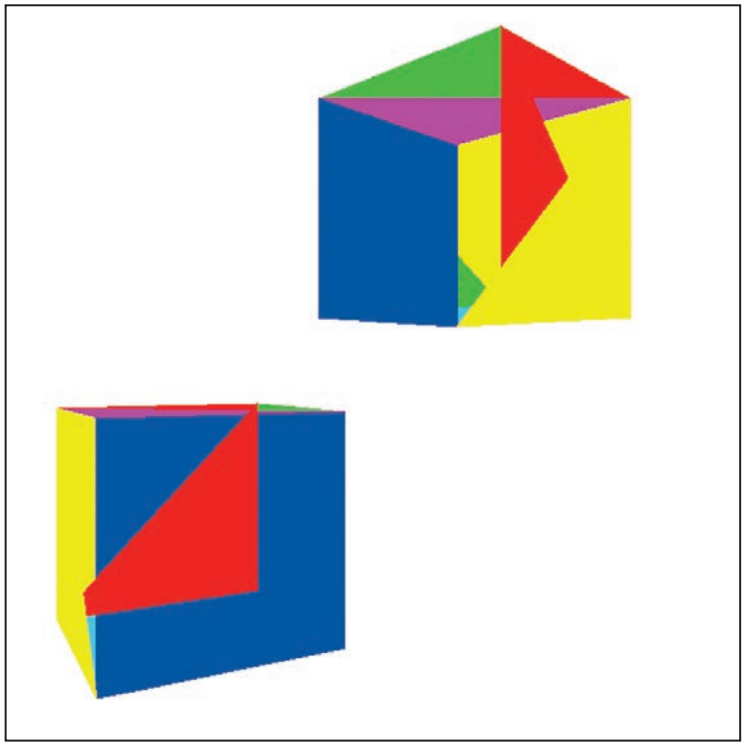


Рис. 12.1. Использование DrawFilledTriangle вместо DrawWireframeTriangle не даст ожидаемого результата

Можно вернуться к определению модели и изменить последовательность треугольника в нем. Но если наша сцена содержит другой экземпляр куба, повернутый на 180°, то мы опять столкнемся с такой же проблемой. Если кратко, то нет единого верного порядка треугольников, который сгодился бы для каждого экземпляра и всех позиций камеры. Что же делать?

Алгоритм художника

Первое решение проблемы — *алгоритм художника*. Реальные художники рисуют сначала задний план, а потом закрывают его части объектами переднего. Взяв это за основу, мы можем рисовать все треугольники сцены, начиная с задних и продвигаясь вперед. Для этого нужно применить преобразования камеры и модели. Еще нужно упорядочить сами треугольники по их удаленности от камеры.

Так мы решим проблему, описанную выше, потому что теперь будем искать подходящий порядок для конкретной относительной позиции объектов и камеры.

Есть у этого подхода и ряд недостатков, делающих его непрактичным. Во-первых, он плохо масштабируется. Наиболее эффективный алгоритм сортировки из известных выглядит как $O(n \log(n))$ и означает, что время выполнения при удваивании числа треугольников увеличивается больше чем вдвое (например, для сортировки 100 треугольников нужно около 200 операций, сортировка 200 треугольников займет уже 460 операций, а не 400, а сортировка 800 треугольников потребует 2322 операции, а не 1840). Проще говоря, этот вариант хорошо работает для малых сцен.

Во-вторых, такому алгоритму сразу нужна информация обо всем списке треугольников. Это требует огромного количества памяти и лишает нас возможности потоковой реализации рендеринга. Нам же нужно, чтобы рендерер был как конвейер, в котором с одной стороны входят треугольники модели, а с другой выходят пиксели. Но этот алгоритм не начинает отрисовку пикселей, пока все треугольники не будут преобразованы и упорядочены.

В-третьих, даже если смириться с ограничениями выше, есть случаи, в которых верного порядка треугольников *вообще не существует*. Посмотрите пример на рис. 12.2.

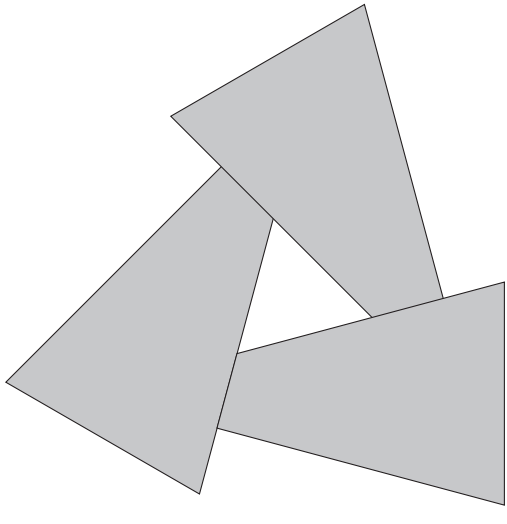


Рис. 12.2. Отсортировать эти треугольники от заднего плана к переднему невозможно

Буферизация глубины

Проблему с упорядочением на уровне треугольников решить не получается, поэтому попробуем перейти на уровень пикселей.

Каждый пиксель на холсте нужно закрасить цветом ближайшего к камере объекта. На рис. 12.3 это P_1.

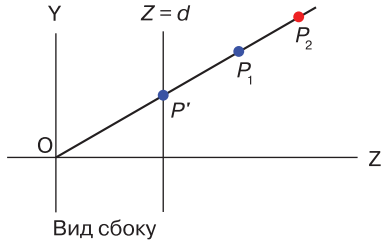


Рис. 12.3. И P_1, и P_2 проецируют на холст одну точку P'. P_1 ближе к камере, чем P_2, поэтому мы закрашиваем P' цветом P_1

В любой момент процесса рендеринга каждый пиксель на холсте представляет одну точку в сцене (до начала отрисовки он представляет бесконечно удаленную точку). Допустим, для каждого пикселя на холсте мы сохранили координату Z точки, которую он представляет. Закрасить пиксель цветом объекта мы можем, только если координата Z точки, которую нужно закрасить, меньше координаты Z уже имеющейся в этом месте точки. Это гарантирует, что поверх пикселя, представляющего точку в сцене, мы никогда не отрисуем другой пиксель, представляющий более удаленную от камеры точку.

Вернемся к рис. 12.3. Предположим, что из-за порядка треугольников в модели нам нужно закрасить сначала P_2, а потом P_1. При закрашивании P_2 пиксель заполняется красным и связанное с ним значение Z становится Z_{P_2}. Дальше нужно закрасить P_1, а поскольку $Z_{P_2} > Z_{P_1}$, мы закрашиваем пиксель синим и получаем желаемый результат.

Именно в нашем случае мы получили бы верный результат, независимо от значений Z, потому что точки были в удобном порядке. Но что, если нужно закрасить сначала P_1, а потом P_2? Сперва мы закрашиваем пиксель синим и сохраняем Z_{P_1}. Но когда нам нужно закрашивать P_2, мы видим, что $Z_{P_2} > Z_{P_1}$, поэтому не делаем этого — так как если его закрасить, то P_2 перекроет P_1. А так мы снова получаем синий пиксель, то есть корректный результат.

Для реализации нам понадобится буфер для хранения координаты Z каждого пикселя на холсте — *буфер глубины*. Его размерность совпадает с размерностью холста, но его элементы — вещественные числа, а не пиксели.

Откуда же берутся значения Z? Ими должны быть значения Z точек после их трансформации, но до выполнения перспективной проекции. Но так мы получим только значения Z для вершин, а нам нужны Z для каждого пикселя каждого треугольника.

Вот и еще одно применение алгоритма отображения атрибутов из главы 8. Почему бы не использовать Z как атрибут и интерполировать его вдоль лицевой стороны треугольника, как мы делали со значениями интенсивности цвета? Но сейчас вы уже

это умеете: нужно взять значения `z0`, `z1` и `z2`, вычислить `z01`, `z02` и `z012`, совместить их для получения `z_left` и `z_right`. Потом для каждого горизонтального отрезка вычислить `z_segment`. В завершение вместо слепого вызова `PutPixel(x, y, color)` нужно сделать следующее:

```
z = z_segment[x - xl]
if (z < depth_buffer[x][y]) {
    canvas.PutPixel(x, y, color)
    depth_buffer[x][y] = z
}
```

Чтобы все сработало, каждая точка в `depth_buffer` должна быть определена как $+\infty$ (или просто с «очень большим значением»). Благодаря этому в первый раз, когда нам понадобится отрисовать пиксель, условие гарантированно будет оценено как верное. Потому что любая точка в сцене ближе к камере, чем бесконечно удаленная.

Теперь мы получим результаты поинтереснее — смотрим рис. 12.4.

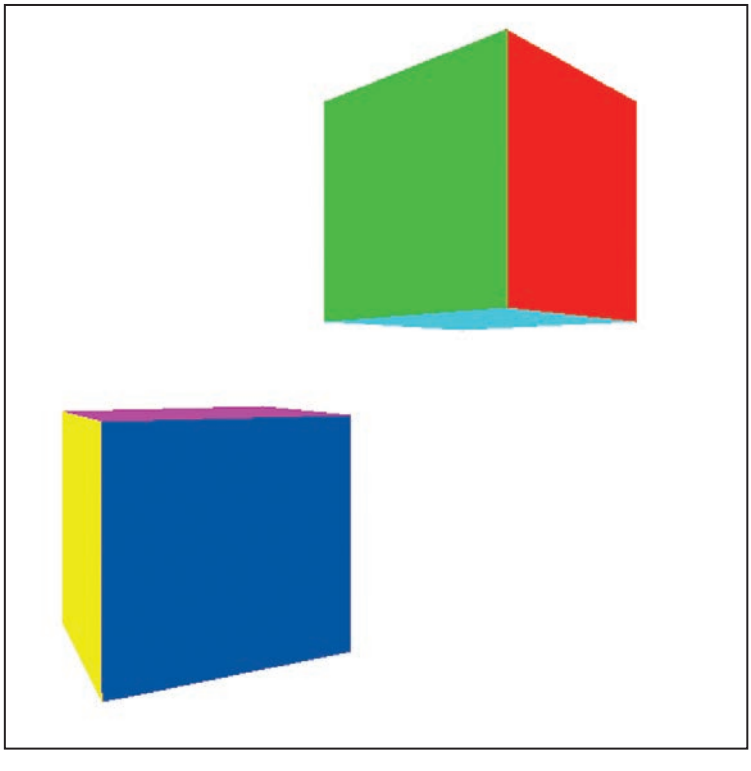


Рис. 12.4. Теперь кубы выглядят как положено, независимо от порядка их треугольников

Живая реализация этого алгоритма находится по адресу https://gabrielgambetta.com/cgfs/depth-demo.

Использование 1/Z вместо Z

Результат получился намного лучше, но наши действия не совсем правильные. Значения Z для вершин верны (они берутся из данных), но в большинстве случаев линейно интерполированные значения Z для оставшихся пикселей будут ошибочны. Пока этого не видно, но может стать проблемой позднее.

Давайте разберемся, почему значения ошибочны. Рассмотрим простой случай отрезка, идущего из $A(-1, 0, 2)$ в $B(1, 0, 10)$. Его средняя точка M находится в $(0, 0, 6)$. Из-за того, что M — середина AB, мы знаем, что $M_z = (A_z + B_z)/2 = 6$. Сам отрезок представлен на рис. 12.5.

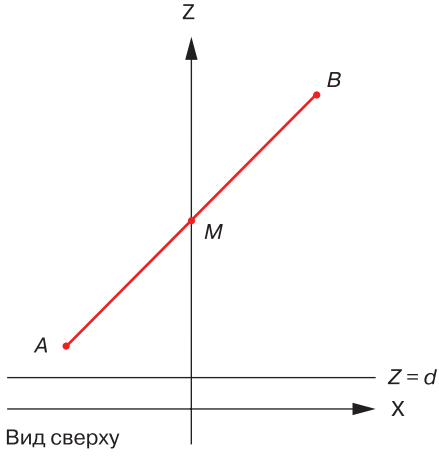


Рис. 12.5. Отрезок AB и его средняя точка M

Вычислим проекцию этих точек при $d = 1$. С помощью уравнения перспективной проекции получаем $A'_x = A_x / A_z = -1/2 = -0{,}5$. По тому же принципу $B'_x = 0{,}1$, а $M'_x = 0$. На рис. 12.6 показаны спроецированные точки.

$A'B'$ — это горизонтальный отрезок в окне просмотра. Нам известны значения A_z и B_z. Давайте посмотрим, что происходит, если попробовать вычислить значение M_z с помощью линейной интерполяции. На рис. 12.7 показана предполагаемая линейная функция.

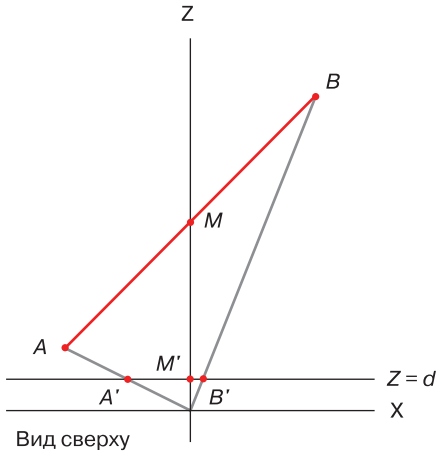


Рис. 12.6. Точки *A*, *B* и *M* спроецированы на плоскость проекции

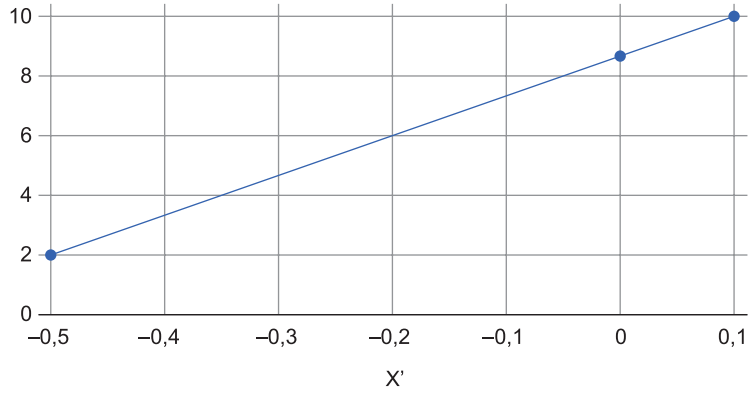


Рис. 12.7. Значения A_z и B_z для A'_x и B'_x определяют линейную функцию $z = f(x')B'_x = 0, 1$

Уклон функции постоянен, значит, можно написать:

$$\frac{M_z - A_z}{M'_x - A'_x} = \frac{B_z - A_z}{B'_x - A'_x}.$$

Переиграв это выражение, можно решить его для M_z:

$$M_z = A_z + \left(M'_x - A'_x\right)\left(\frac{B_z - A_z}{B'_x - A'_x}\right).$$

Если подставить известные значения и проделать необходимые вычисления, получим:

$$M_z = 2 + (0 - (-0{,}5))\left(\frac{10-2}{0{,}1-(-0{,}5)}\right) = 2 + (0{,}5)\left(\frac{8}{0{,}6}\right) = 8{,}666.$$

Здесь у нас получилось, что $M_z = 8{,}666$, но мы знаем, что на деле это 6!

Где мы ошиблись? Мы используем линейную интерполяцию, которая прекрасно работает. Мы передаем верные значения, полученные из данных. Так почему наш результат неверный?

Ошибка кроется в том, что мы допускаем, что интерполируемая нами функция линейная изначально! В данном случае это не так.

Если бы $Z = f(x', y')$ была линейной функцией для x' и y', можно было бы записать ее как $Z = Ax' + By' + C$ для определенных значений A, B и C. У подобной функции есть одно свойство — ее приращение зависит только от расстояния между точками, но не от их положения:

$$f(x' + \Delta x, y' + \Delta y) - f(x', y') = [A(x' + \Delta x) + B(y' + \Delta y) + C] - [Ax' + By' + C] =$$
$$= A(x' + \Delta x - x') + B(y' + \Delta y - y') + C - C = A\Delta x + B\Delta y.$$

Это значит, что для заданной разницы в координатах экрана разница в Z всегда будет одинаковой.

Уравнение плоскости, содержащей рассматриваемый отрезок, будет:

$$Ax + By + Cz + D = 0.$$

С другой стороны, у нас есть уравнения перспективной проекции:

$$x' = \frac{xd}{z};$$
$$y' = \frac{xd}{z}.$$

Мы можем получить обратно x и y отсюда:

$$x = \frac{zx'}{d};$$
$$y = \frac{zy'}{d}.$$

Если заменить x и y в уравнении плоскости на эти выражения, то получим:

$$\frac{Ax'z + By'z}{d} + Cz + D = 0.$$

Умножаем на d, а затем решаем для z:

$$Ax'z + By'z + dCz + dD = 0;$$
$$\left(Ax' + By' + dC\right)z + dD = 0;$$
$$z = \frac{-dD}{Ax' + By' + dC}.$$

Это точно *не* линейная функция для x' и y', и поэтому линейная интерполяция значений z давала ошибочный результат.

Но если вместо z вычислить $1/z$, то получится:

$$1/z = \frac{Ax' + By' + dC}{-dD}.$$

А вот это уже точно линейная функция для x' и y'. Это значит, что можно линейно интерполировать значения $1/z$ и получить верные результаты.

Давайте убедимся, что подход рабочий, и вычислим интерполированное значение для M_z. Но на этот раз с использованием линейной интерполяции $1/z$:

$$\frac{M_{\frac{1}{z}} - A_{\frac{1}{z}}}{M'_x - A'_x} = \frac{B_{\frac{1}{z}} - A_{\frac{1}{z}}}{B'_x - A'_x};$$
$$M_{\frac{1}{z}} = A_{\frac{1}{z}} + \left(M'_x - A'_x\right)\left(\frac{B_{\frac{1}{z}} - A_{\frac{1}{z}}}{B'_x - A'_x}\right);$$
$$M_{\frac{1}{z}} = \frac{1}{2} + \left(0 - (0{,}5)\right)\left(\frac{\frac{1}{10} - \frac{1}{2}}{0{,}1 - (0{,}5)}\right) = 0{,}166666.$$

Из чего следует:

$$M_{\frac{1}{z}} = \frac{1}{M_{\frac{1}{z}}} = \frac{1}{0{,}166666} = 6.$$

Это уже верное значение, совпадающее с исходными расчетами M_z, основанными на геометрических параметрах отрезка.

Это значит, что нам нужно использовать для буферизации глубины значения $1/z$, а не z. В псевдокоде это приводит к двум практическим отличиям: необходимости инициализировать каждую точку в буфере как 0 (что в принципиальном смысле означает $1/+\infty$) и инвертировать выражение сравнения (мы сохраняем большее значение $1/z$, соответствующее меньшему значению z).

Отбрасывание задней грани

Буферизация глубины дает нужный результат. Но можно ли этот процесс ускорить?

Вернемся к кубу. Даже если каждый пиксель в итоге получит правильный цвет, многие из них будут закрашены по многу раз. Например, если задняя грань куба отрисовывается до передней, многие пиксели будут закрашены дважды, а это очень затратно. Раньше мы вычисляли $1/z$ для каждого пикселя, но скоро будем добавлять еще атрибуты (например, освещение). Чем больше будет попиксельных операций, тем затратнее будет вычислять пиксели, которые никогда не будут видны.

Можем ли мы отбросить пиксели до углубления во все эти вычисления? Оказывается, можно отбрасывать *целые треугольники* еще до начала самого рендеринга.

Все это время мы рассуждали о *передних* и *задних* гранях неформально. Представьте, что у каждого треугольника есть две разные стороны, которые невозможно видеть одновременно. Для того чтобы их различать, мы прикрепим на каждый треугольник воображаемую стрелку, перпендикулярную к его поверхности. Теперь возьмем куб и убедимся, что все стрелки указывают наружу (рис. 12.8).

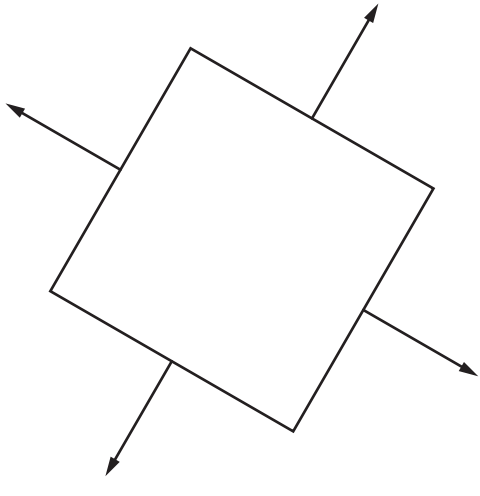


Рис. 12.8. Вид куба сверху со стрелками из каждого треугольника

Благодаря этим стрелкам мы можем определить, «передний» или «задний» будет треугольник, в зависимости от того, куда они указывают. Иначе говоря, если

вектор обзора и эта стрелка (которая является вектором нормали треугольника) формируют угол меньше 90°, треугольник направлен вперед, в противном случае он смотрит назад (рис. 12.9).

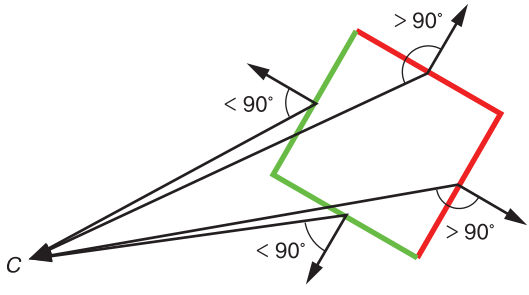


Рис. 12.9. Угол между векторами обзора и нормали треугольника позволяет обозначить его как направленный вперед или назад

На этой стадии нужно наложить на наши 3D-модели ограничение, утверждающее, что они *замкнуты*. Точное определение замкнутости сложное, но нам достаточно интуитивного понимания. Куб, с которым мы работаем, замкнут: мы можем видеть только его внешнюю сторону. Если удалить одну из его граней, он перестанет быть замкнутым, ведь мы сможем видеть его изнутри. Это не значит, что у нас не может быть объектов с отверстиями или вогнутостями. Их можно моделировать просто с помощью тонких «стен». Посмотрите примеры на рис. 12.10.

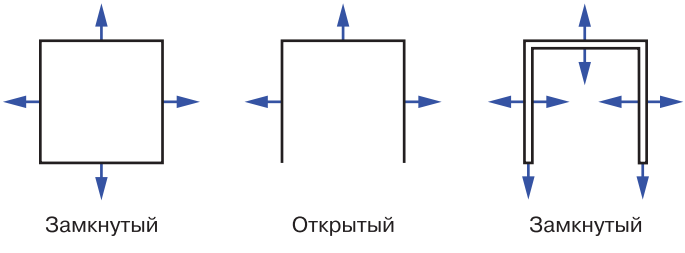


Рис. 12.10. Некоторые примеры замкнутых и открытых объектов

Зачем нам вводить это ограничение? У замкнутых объектов, независимо от ориентации модели или камеры, набор передних граней полностью перекрывает набор задних. Мы можем не рисовать задние грани и сэкономим ценное вычислительное время.

Этот алгоритм так и называется — *отбрасывание задних граней*, а его псевдокод очень прост для алгоритма, который вдвое сокращает время рендеринга (листинг 12.1).

Листинг 12.1. Алгоритм отбрасывания задних граней

```
CullBackFaces(object, camera) {
    for T in object.triangles {
        if T is back-facing {
            remove T from object.triangles
        }
    }
}
```

Дальше мы подробнее рассмотрим, как определить, куда смотрит треугольник.

Классификация треугольников

Предположим, что у нас есть вектор нормали $\vec{N}$ треугольника и вектор $\vec{V}$, идущий из его вершины к камере. Теперь допустим, что $\vec{N}$ направлен наружу объекта. Чтобы классифицировать этот треугольник как смотрящий вперед или назад, мы вычисляем угол между $\vec{N}$ и $\vec{V}$ и проверяем, находятся ли они в пределах 90° относительно друг друга.

Чтобы было проще, воспользуемся свойствами скалярного произведения. Напомню, если α — это угол между $\vec{N}$ и $\vec{V}$, тогда:

$$\frac{\langle \vec{N}, \vec{V} \rangle}{|\vec{N}||\vec{V}|} = \cos(\alpha).$$

Так как $\cos(\alpha)$ для $|\alpha| \le 90°$ не отрицателен, то для определения направленности треугольника вперед или назад нужно только узнать знак этого выражения. Обратите внимание, что $|\vec{N}|$ и $|\vec{V}|$ всегда положительны, то есть на знак выражения повлиять не могут. Получаем:

$$\text{sign}\left(\langle \vec{N}, \vec{V} \rangle\right) = \text{sign}\left(\cos(\alpha)\right).$$

Критерий классификации будет прост:

- $\langle \vec{N}, \vec{V} \rangle \le 0$ смотрит назад;
- $\langle \vec{N}, \vec{V} \rangle > 0$ смотрит вперед.

Есть граничный случай, при котором $\langle \vec{N}, \vec{V} \rangle = 0$, когда мы смотрим на ребро треугольника лоб в лоб. То есть камера и треугольник оказываются в одной плоскости. Этот треугольник можно определить в любую сторону, и на результат это не сильно повлияет. Поэтому мы принимаем его как направленный назад, чтобы не сталкиваться с обработкой вырожденных треугольников.

А откуда берется вектор нормали? Есть такая векторная операция — *векторное произведение* $\vec{A} \times \vec{B}$. Она получает два вектора, $\vec{A}$ и $\vec{B}$, производя вектор, перпендикулярный к ним (подробности в приложении «Линейная алгебра»). Другими словами, векторное произведение двух векторов на поверхности треугольника — это вектор его нормали. Мы можем легко получить два вектора треугольника вычитанием его вершин друг из друга. Так что вычислить направление вектора нормали треугольника ABC просто:

$$\vec{V}_1 = B - A;$$
$$\vec{V}_2 = C - A;$$
$$\vec{N} = \vec{V}_1 \times \vec{V}_2.$$

Заметьте, что «направление вектора нормали» — это не то же, что «вектор нормали», и на это есть две причины. Первая — $\left|\vec{N}\right|$ не обязательно равен 1. Вообще, это неважно, так как нам интересен только знак $\left\langle \vec{N}, \vec{V} \right\rangle$.

Вторая причина в том, что если $\vec{N}$ — это вектор нормали ABC, то и $-\vec{N}$ тоже. В таком случае очень важно направление $\vec{N}$, ведь именно это позволяет нам классифицировать треугольники как направленные вперед или назад.

А еще векторное произведение двух векторов не коммутативно: $\vec{V}_1 \times \vec{V}_2 = -\left(\vec{V}_2 \times \vec{V}_1\right)$. Другими словами, порядок векторов важен в этой операции. И мы определили V_1 и V_2 через A, B и C, значит, порядок вершин в треугольниках тоже имеет значение. Мы больше не можем рассматривать треугольники ABC и ACB как один.

Это все не случайно. У нас есть определение операции векторного произведения и система координат, которой мы пользуемся (X — вправо, Y — вверх, Z — вперед). В таком случае есть простое правило, определяющее направление вектора нормали. Если вершины треугольника ABC расположены по часовой стрелке с точки зрения камеры, то вектор нормали будет указывать в сторону камеры, то есть камера смотрит на переднюю грань треугольника.

Помните об этом при проектировании 3D-моделей вручную и перечисляйте вершины каждого треугольника по часовой стрелке, глядя на переднюю грань, чтобы их нормали указывали наружу. Наш пример модели куба, конечно, этому правилу следует.

Итоги главы

Раньше наш рендерер мог отрисовывать только проволочные объекты. В этой главе мы научили его рендерить монолитные структуры. Это уже сложнее, чем простое использование `DrawFilledTriangles` вместо `DrawWireframeTriangles`, ведь здесь нужно, чтобы треугольники ближе к камере заслоняли те, которые находятся дальше от нее.

Сначала мы хотели отрисовывать треугольник, начиная с задних и приближаясь к ближним. Но у этого подхода были недостатки. Так мы решили рассмотреть ситуацию на уровне пикселей и пришли к технике буферизации глубины. Ее результаты не зависели от порядка отрисовки треугольников.

Еще мы рассмотрели необязательную, но ценную технику — отбрасывание задних граней. Она не влияет на корректность результатов, но может избавить от рендеринга примерно половины треугольников сцены. Все направленные назад треугольники закрытого объекта перекрываются теми, что направлены вперед, поэтому отрисовывать их не нужно. Мы даже представили простой алгебраический способ для определения того, направлен треугольник вперед или назад.

Вот вы и научились отрисовывать сплошные объекты. Оставшаяся часть книги будет посвящена приданию им реалистичности.

13

Затенение

Мы продолжаем делать наши изображения реалистичнее. В этой главе вы научитесь добавлять в сцену источники света и подсвечивать объекты в ней. Но сначала давайте разберемся с терминологией.

Затенение и освещение

Эта глава называется «Затенение», а не «Освещение», но эти понятия тесно связаны. *Освещение* относится к математическим операциям и алгоритмам для вычисления воздействия света на одну точку в сцене. *Затенение (шейдинг)* же работает с техниками, расширяющими этот эффект света от дискретного набора точек до целых объектов.

Благодаря информации из главы 3 вы умеете определять рассеянное, точечное и направленное освещение, вычислять его в любой точке сцены с учетом ее позиции и нормали поверхности в этой точке:

$$I_P = I_A + \sum_{i=1}^{n} I_i \left[\frac{\langle \vec{N}, \vec{L}_i \rangle}{|\vec{N}||\vec{L}_i|} + \left(\frac{\langle \vec{R}_i, \vec{V} \rangle}{|\vec{R}_i||\vec{V}|} \right)^s \right].$$

Это уравнение освещенности, выражающее влияние света на точку в сцене. В растеризаторе оно будет работать так же, как и в трассировщике лучей.

В этой же главе мы рассмотрим кое-что поинтереснее: расширение разработанных нами алгоритмов «освещения в точке» в алгоритмы «освещения в каждой точке треугольника».

Плоское затенение

Начнем с простого. Мы умеем вычислять освещение в точке и можем просто выбрать любую точку в треугольнике (например, центр), вычислить освещение в ней и использовать его для затенения всего треугольника. Для этого нам нужно будет просто умножить цвет треугольника на вычисленное значение освещения. Результат виден на рис. 13.1.

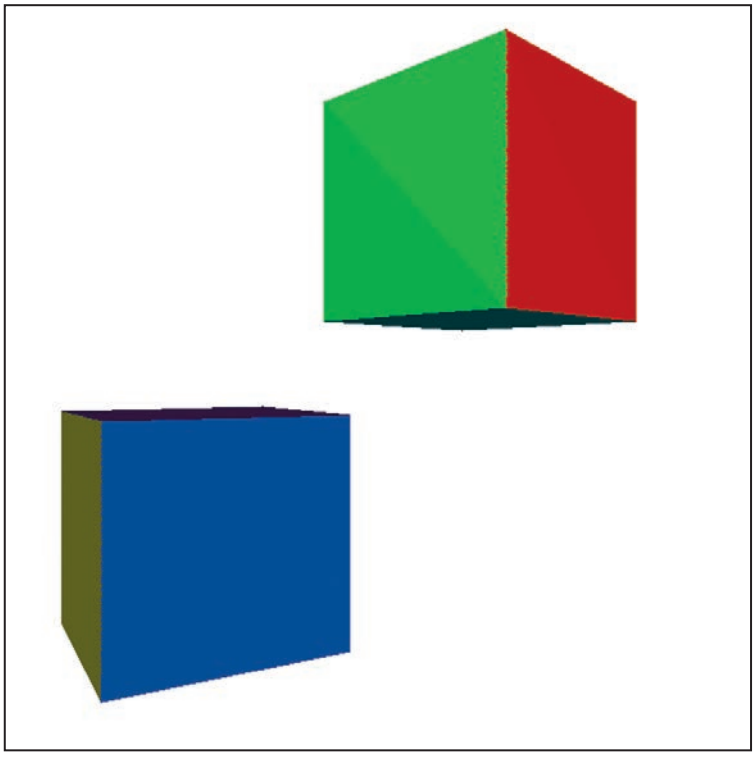


Рис. 13.1. В плоской модели затенения сначала вычисляется освещение в центре треугольника, потом оно используется для всего треугольника

Выглядит обнадеживающе. Каждая точка треугольника имеет одну и ту же нормаль при условии, что свет достаточно удален, световые векторы для каждой точки

приблизительно параллельны и каждая точка получает *приблизительно* одинаковое количество света. Неоднородность между двумя треугольниками, из которых состоят стороны куба, особенно заметна на его зеленой грани (см. рис. 13.1). Это следствие *приблизительной*, но не *точной* параллельности световых векторов.

Что же будет, если применить эту технику к объекту, у которого каждая точка имеет свою нормаль, как в случае со сферой на рис. 13.2?

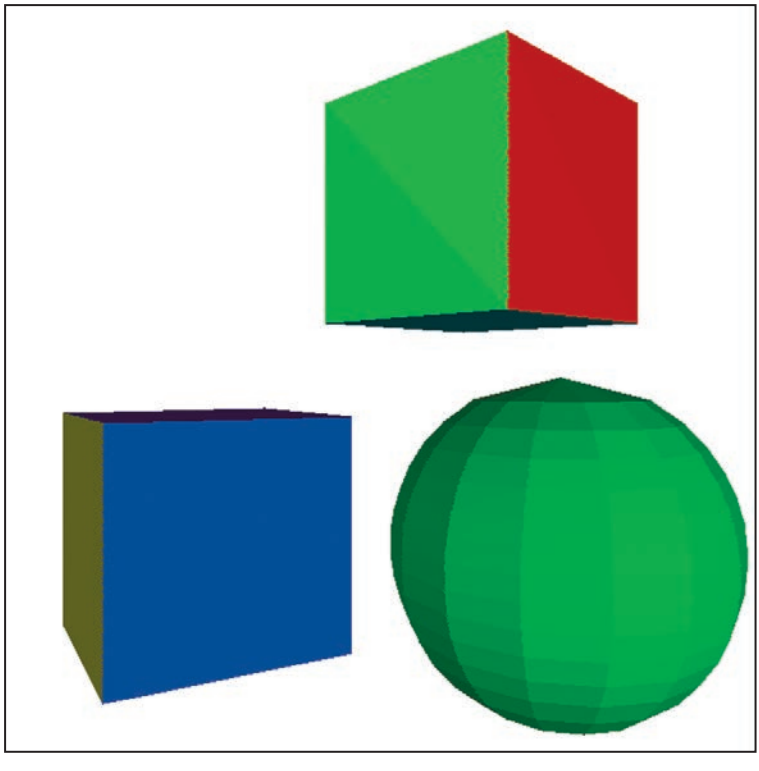


Рис. 13.2. Плоское затенение неплохо работает для объектов с плоскими гранями, но меньше подходит для кривых линий

Выглядит не очень. Сразу понятно, что это не настоящая сфера, а ее приближенная версия из плоских треугольных частей. Именно из-за такой особенности эта техника и называется *плоским затенением*.

Затенение по Гуро

Как же нам избавиться от этих неоднородностей? Можно вычислить освещение не в центре треугольника, а в трех его вершинах. Так мы получим три значения между 0,0 и 1,0, по одному для каждой из вершин. Это ставит нас в ситуацию из главы 8: мы можем напрямую применить `DrawShadedTriangle`, использовав полученные значения освещения как атрибут «интенсивности».

Эта техника называется *затенением по Гуро*, в честь Анри Гуро, разработавшего ее в 1971 году. На рис. 13.3 показаны результаты применения этой техники к кубу и сфере.

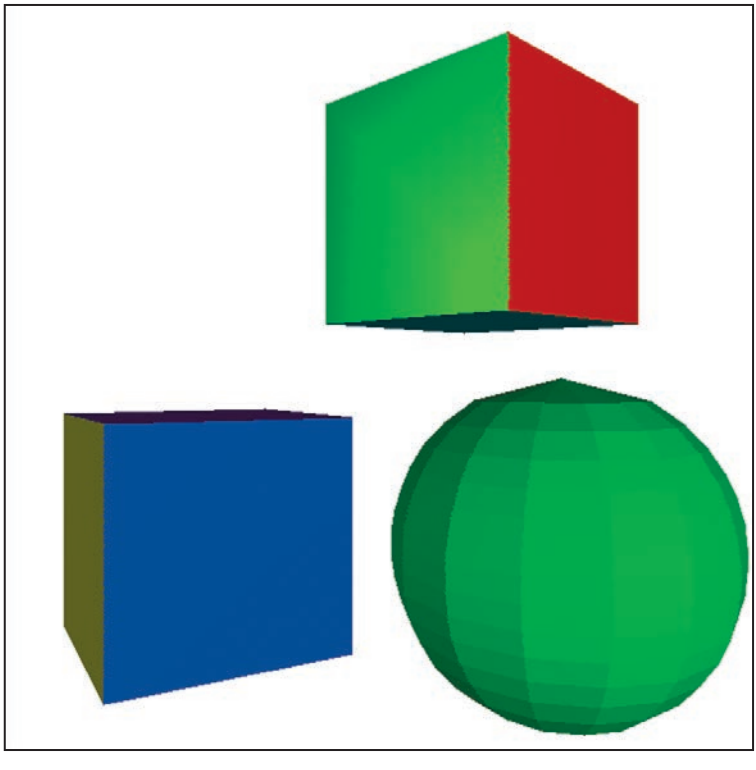


Рис. 13.3. В методе затенения по Гуро сначала вычисляется освещение в вершинах треугольника, а потом полученные значения интерполируются по всей его поверхности

Куб выглядит уже лучше. Неоднородность пропала, так как оба треугольника грани имеют две общие вершины и одну нормаль. В результате освещение этих вершин оказывается одинаковым для обоих треугольников.

А вот сфера все еще кажется граненой, и неоднородности ее поверхности прорисованы ошибочно. Так происходит из-за того, что мы рассматриваем сферу как коллекцию плоских поверхностей. И хотя каждый треугольник разделяет вершины с соседними, их нормали различаются. На рис. 13.4 эта проблема отражена наглядно.

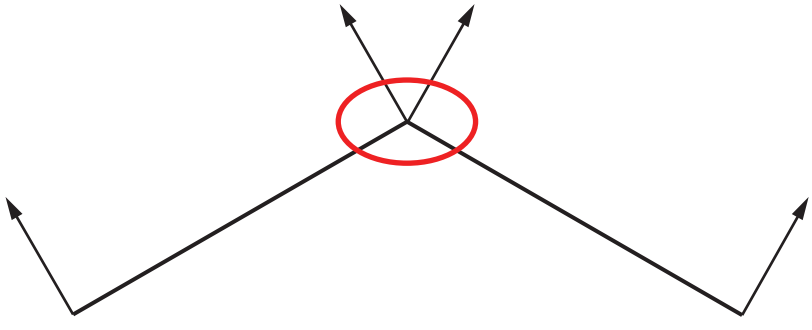


Рис. 13.4. Мы получаем разные значения для освещения общей вершины. Они определяются нормалями треугольников, а эти нормали разные

Вернемся немного назад. Использование треугольников для изогнутого объекта ограничивает наши техники, но не свойство самого объекта.

Каждая вершина в модели сферы соответствует точке на ней. Но треугольники, которые этими точками определяются, — только аппроксимация ее поверхности. Лучше сделать так, чтобы вершины в модели представляли точки на сфере максимально точно, — использовать фактические нормали сферы для каждой вершины (рис. 13.5).

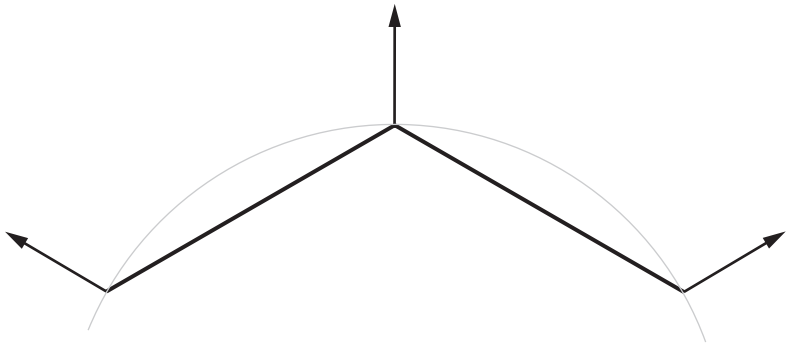


Рис. 13.5. Для каждой вершины можно установить нормаль поверхности, которую она представляет

К кубу это не относится. Каждую его грань нужно затенять независимо от остальных. Нет единой верной нормали для всех вершин куба.

Наш рендерер не знает, должна модель быть аппроксимацией криволинейного объекта или же точным представлением плоского. В конце концов, куб — это очень грубая аппроксимация сферы! Чтобы решить эту проблему, мы сделаем нормали треугольника частью модели, позволив дизайнеру принять это решение.

Некоторые объекты (сфера) имеют по одной нормали для каждой вершины. В других (куб) нормали различаются для каждого треугольника с общей вершиной. Значит, можно сделать нормали свойством вершин — они должны быть свойством использующих их треугольников:

```
model {
    name = cube
    vertices {
        0 = (-1, -1, -1)
        1 = (-1, -1, 1)
        2 = (-1,  1, 1)
        ...
    }
    triangles {
        0 = {
            vertices = [0, 1, 2]
            normals = [(-1, 0, 0), (-1, 0, 0), (-1, 0, 0)]
        }
    ...
    }
}
```

На рис. 13.6 показана сцена, отрисованная с использованием затенения по Гуро и соответствующих нормалей вершин.

Кубы все так же выглядят как кубы, а сфера теперь действительно похожа на сферу. Понять, что она состоит из треугольников, можно только по ее контурам. Но это тоже можно исправить, если использовать более мелкие треугольники. Однако для этого потребуется больше вычислительных ресурсов.

Затенение по Гуро становится неэффективным, когда дело доходит до рендеринга объектов с отблесками. Мы видим, что блик на сфере выглядит нереалистично. Это указывает на более общую проблему. Когда мы перемещаем точечный свет очень близко к большой грани, то ждем, что она станет ярче и эффекты отражения будут более выражены. Но затенение по Гуро дает ровно противоположный эффект (рис. 13.7).

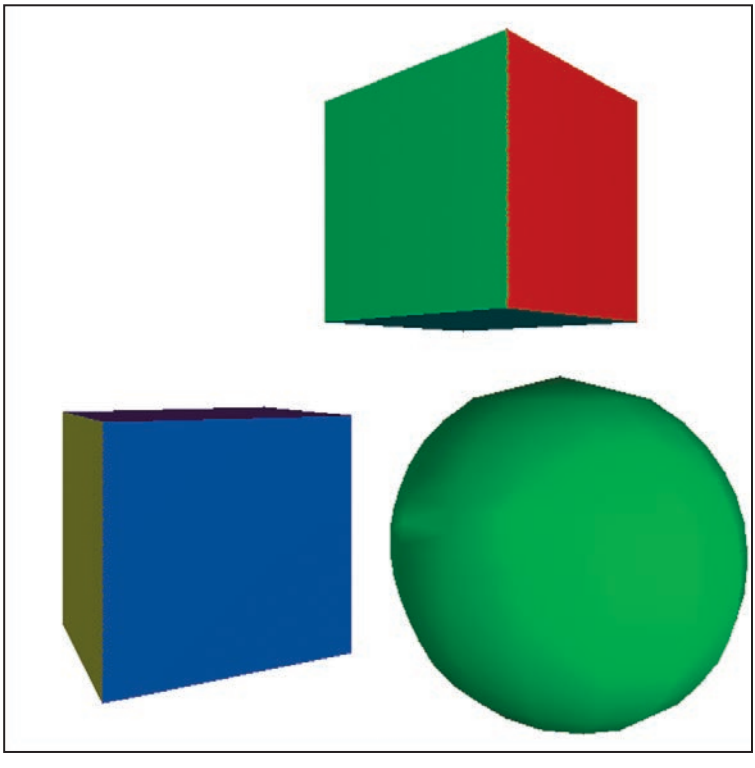


Рис. 13.6. Затенение по Гуро с заданными в модели векторами нормалей

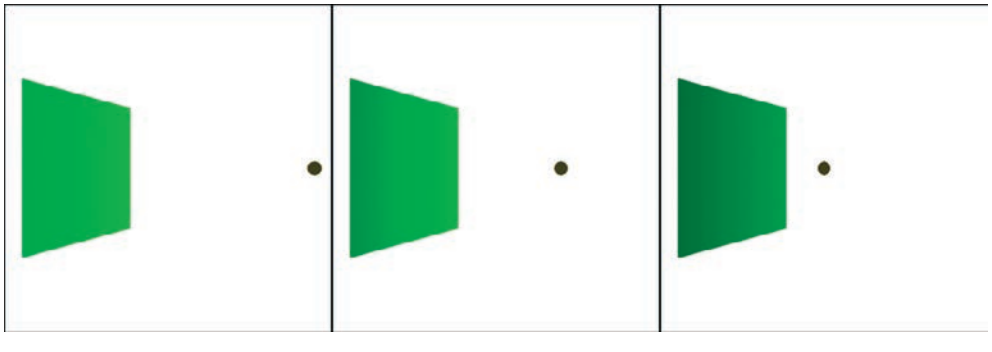


Рис. 13.7. Чем ближе точечный свет к грани, тем темнее она выглядит

Мы ожидаем, что точки возле центра треугольника получат много света, потому что $\vec{L}$ и $\vec{N}$ примерно параллельны. Но мы вычисляем освещение не в центре треугольника, а в его вершинах. Поэтому чем ближе источник света к поверхности, тем *больше* угол между лучом и нормалью и точки получают меньше света. Значит, каждый внутренний пиксель получит значение интенсивности, которое будет результатом интерполяции между двух малых значений. Иначе говоря, оно тоже окажется мало. На рис. 13.8 это представлено наглядно.

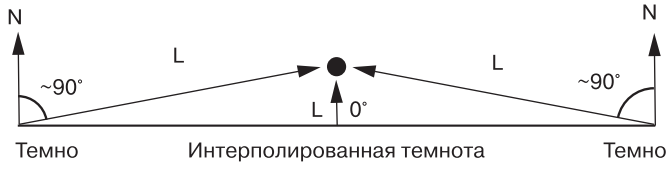


Рис. 13.8. Интерполирование освещения от темных вершин приводит к темноте в центре, но его нормаль в этой точке параллельна световому вектору

Что же делать?

Затенение по Фонгу

В случае с затенением по Гуро ситуацию можно исправить, но придется пойти на компромисс между качеством и затратой ресурсов.

Плоское затенение подразумевало по одному вычислению освещения для каждого треугольника. В затенении по Гуро требуется уже три таких вычисления на треугольник плюс интерполяция одного атрибута, освещения, по всему треугольнику. Очередной шаг в повышении качества требует вычисления освещения для каждого пикселя треугольника.

В теории это звучит несложно. Мы уже вычисляем освещение в одной или трех точках, а при трассировке лучей нам приходилось вычислять освещение для каждого пикселя. Загвоздка здесь в том, чтобы определить, откуда брать входные данные для уравнения освещенности.

Напомню, что полное уравнение освещенности с рассеянным, диффузным и зеркальным компонентом выглядит так:

$$I_P = I_A + \sum_{i=1}^{n} I_i \left(\frac{\left\langle \vec{N}, \vec{L}_i \right\rangle}{\left|\vec{N}\right|\left|\vec{L}_i\right|} + \left(\frac{\left\langle \vec{R}, \vec{V} \right\rangle}{\left|\vec{R}\right|\left|\vec{V}\right|} \right)^s \right).$$

Во-первых, нам нужен $\vec{L}$. Для направленных источников света он дан. Для точечных $\vec{L}$ определяется как вектор из точки в сцене P до позиции источника Q. А вот Q для каждого пикселя треугольника мы не знаем. Нам известны только его значения для вершин.

Но у нас есть проекция P — координаты x' и y', которые мы собираемся отрисовать на холсте. Нам известно, что:

$$x' = \frac{xd}{z};$$
$$y' = \frac{yd}{z}.$$

Еще у нас есть интерполированное, но геометрически верное значение $\frac{1}{z}$ — часть алгоритма буферизации глубины, поэтому:

$$x' = xd\frac{1}{z};$$
$$y' = yd\frac{1}{z}.$$

Из этих значений можно воссоздать P:

$$x = \frac{x'}{d\frac{1}{z}};$$
$$y = \frac{y'}{d\frac{1}{z}};$$
$$z = \frac{1}{z}.$$

Нам также нужен $\vec{V}$ — вектор от камеры (чью позицию мы знаем) к P (которую мы только что вычислили), значит, $\vec{V}$ — это просто отрезок $P - C$.

Теперь нас интересует $\vec{N}$. Мы знаем нормали только в вершинах треугольника. Когда из инструментов у вас только молоток, то любая задача покажется гвоздем. Наш молоток — это линейная интерполяция значений атрибута. Так что давайте возьмем N_x, N_y и N_z в каждой вершине и рассмотрим каждый из них как атрибут, который можно линейно интерполировать. Потом в каждом пикселе мы пересоберем полученные интерполированные компоненты в вектор, нормализуем его и используем как нормаль этого пикселя.

Эта техника называется *затенением по Фонгу*, в честь Ву Тонг Фонга, разработавшего ее в 1973 году. Результат можно увидеть на рис. 13.9.

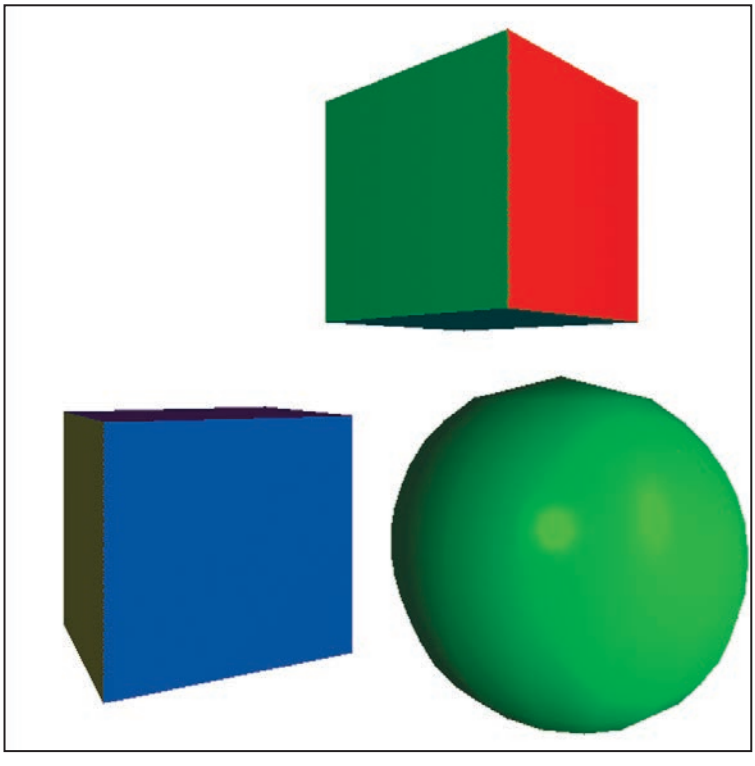


Рис. 13.9. Затенение по Фонгу. Поверхность сферы выглядит гладкой, с четким световым бликом

Живую реализацию алгоритма можно найти по адресу https://gabrielgambetta.com/cgfs/shading-demo.

Вот теперь сфера смотрится намного лучше. Она правильной круглой формы, и на ней отчетливо виден блик света. Но по контуру все еще видно, что мы отрисовываем аппроксимацию из треугольников. Это не недостаток алгоритма затенения, ведь он просто определяет цвет каждого пикселя поверхности треугольников без контроля над их формой. В нашей аппроксимации сферы их 420 штук. Для большей гладкости контура нужно больше треугольников, но это снизит производительность.

Затенение по Фонгу решает проблему утраты яркости с приближением света к грани. Теперь мы можем видеть ожидаемый результат (рис. 13.10).

Теперь, когда мы реализовали с помощью растеризатора все возможности трассировщика лучей из части I, кроме теней и отражений, на рис. 13.11 показан итоговый вывод.

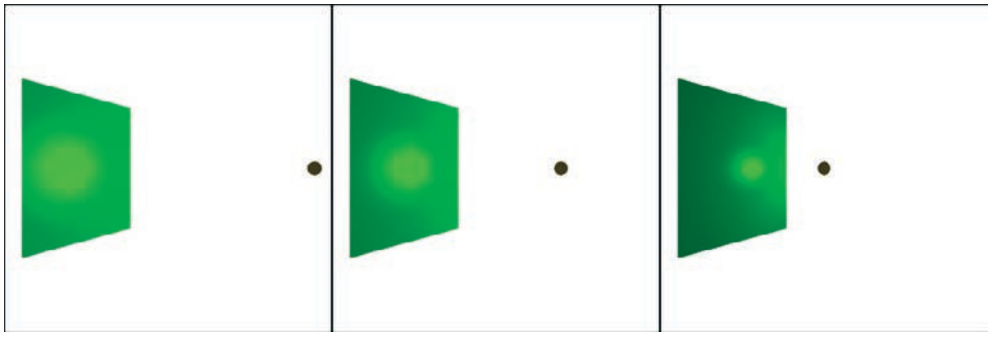


Рис. 13.10. Чем ближе источник света к поверхности, тем ярче и четче световой блик

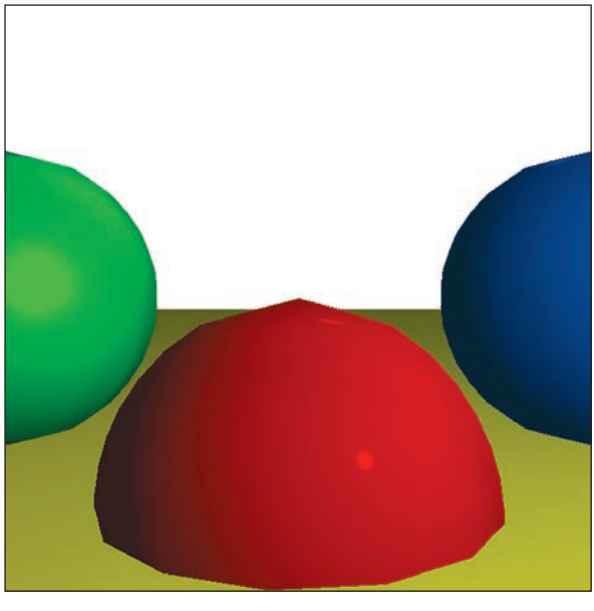


Рис. 13.11. Эталонная сцена, отрисованная растеризатором

Сравним эту сцену с версией той же сцены, полученной через трассировку лучей (рис. 13.12).

Несмотря на использование абсолютно разных техник, оба варианта выглядят почти одинаково. Это вполне ожидаемо, ведь определение сцены осталось идентичным. Единственное видимое отличие в контуре сфер: трассировщик отрисовывает их в виде математически идеальных объектов, а в растеризаторе используется аппроксимация из треугольников.

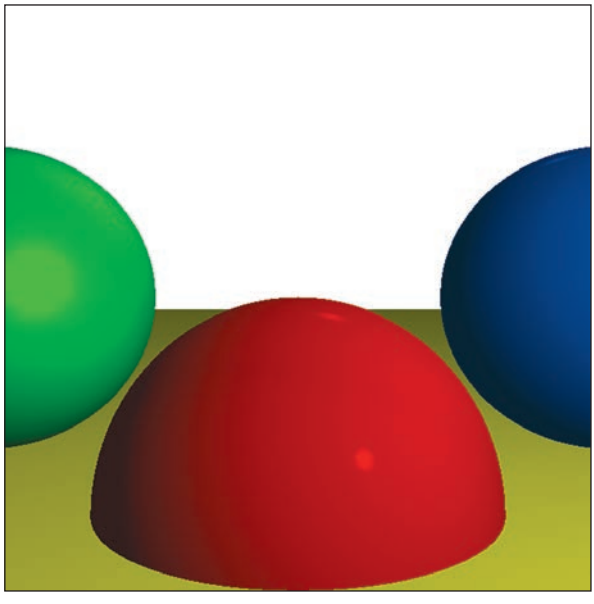


Рис. 13.12. Эталонная сцена, отрисованная трассировщиком лучей

Еще одно отличие — в скорости этих двух рендереров. Она сильно зависит от аппаратного обеспечения и реализации. Но если обобщить, то растеризатор может отрисовывать полноэкранную сцену до 60 раз в секунду и более, поэтому он отлично подходит для таких интерактивных приложений, как видеоигры. Трассировщикам же иногда может понадобиться до нескольких секунд на отрисовку всего одной сцены. Хотя в будущем эта разница может сократиться и даже исчезнуть. Технологические усовершенствования в сфере вычислительного оборудования все больше приближают быстродействие трассировщиков к растеризаторам.

Итоги главы

В этой главе мы добавили в растеризатор освещение при помощи уравнения из главы 3, потому что используем одну и ту же модель освещения. Но если трассировщик вычислял это уравнение для каждого пикселя, то растеризатор может поддерживать разные техники, позволяющие добиться нужного баланса между скоростью и качеством изображения.

Самый быстрый, но создающий самое некачественное изображение алгоритм шейдинга — это плоское затенение. В нем значение освещения вычисляется в одной точке треугольника и используется для всех его пикселей. Итоговые изображения имеют граненый вид, особенно это касается объектов вроде сфер.

Следующий шаг в сторону повышения качества — затенение по Гуро. Здесь освещение вычисляется уже для трех вершин треугольника, а потом это значение интерполируется на всю его поверхность. Так даже криволинейные объекты получаются более гладкими. Но с более тонкими световыми эффектами, например бликами, эта техника не справляется.

Наконец, мы пришли к затенению по Фонгу. Аналогично трассировщику в этой технике уравнение освещенности вычисляется в каждом пикселе. Это дает лучший результат, но худшую скорость. Сложность этого приема — в нахождении способа получения всех нужных для уравнения освещенности значений. И снова решением оказывается линейная интерполяция — в нашем случае векторов нормалей.

В следующей главе мы добавим еще больше деталей на поверхности треугольников при помощи наложения текстур — техники, которую мы еще не рассматривали.

14

Текстуры

Наш растеризатор может отрисовывать кубы или сферы. Но нам неинтересны абстрактные фигуры вроде кубов и сфер. Нас интересуют разные реальные объекты, например ящики и планеты или игральные кубики и шарики для рулетки. В этой главе вы научитесь добавлять на поверхность объектов разные визуальные детали с помощью текстур.

Закрашивание ящика

Допустим, нам в сцене нужен деревянный ящик. Как можно превратить в него куб? Первый вариант — добавить много треугольников для воссоздания фактуры дерева, шляпок гвоздей и т. д. Это сработает, но скажется на быстродействии.

Есть еще вариант — сымитировать детали: вместо изменения геометрии объекта мы просто «нарисуем» поверх него что-то похожее на дерево. Если не разглядывать ящик вплотную, то и отличий заметить не удастся, а вычислительная стоимость окажется намного ниже.

Эти два варианта можно совмещать: выбирать правильный баланс между добавлением геометрических деталей и рисованием поверх них для достижения требуемого качества и скорости отрисовки. Мы уже знаем, как работать с геометрией, поэтому сразу перейдем к изучению второго варианта.

Для начала нам нужно изображение, которое мы будем рисовать на треугольниках. Его мы будем звать *текстурой*. На рис. 14.1 показана текстура деревянного ящика.

Рис. 14.1. Текстура деревянного ящика Filter Forge — Attribution 2.0 Generic (CC BY 2.0)

Дальше нужно указать, как эта текстура должна применяться к модели. Это наложение можно определить по треугольникам, сопоставив точки текстуры с их вершинами (рис. 14.2).

Для определения этого сопоставления нам понадобится система координат, позволяющая обращаться к точкам на текстуре. Напомню, что текстура — это просто изображение в виде прямоугольного массива пикселей. Можно было взять координаты x и y и рассматривать пиксели текстуры относительно них, но мы уже используем эти обозначения для холста. Поэтому координатами текстуры у нас будут u и v, а ее пиксели назовем *текселями* (сокр. от англ. *texture elements* — элементы текстуры).

Начало этой системы координат (u, v) мы установим в верхнем левом углу текстуры. Еще объявим, что u и v — это вещественные числа в диапазоне $[0, 1]$, независимо от фактических размеров текселей. Это будет очень удобно по нескольким причинам. Например, нам может понадобиться использовать текстуры с разрешением выше или ниже, в зависимости от объема доступной памяти ОЗУ. Мы не привязаны к реальным размерам пикселей, поэтому можно изменять разрешение, не корректируя саму модель. Мы можем умножить u и v на ширину и высоту текстуры для получения фактических индексов tx и ty.

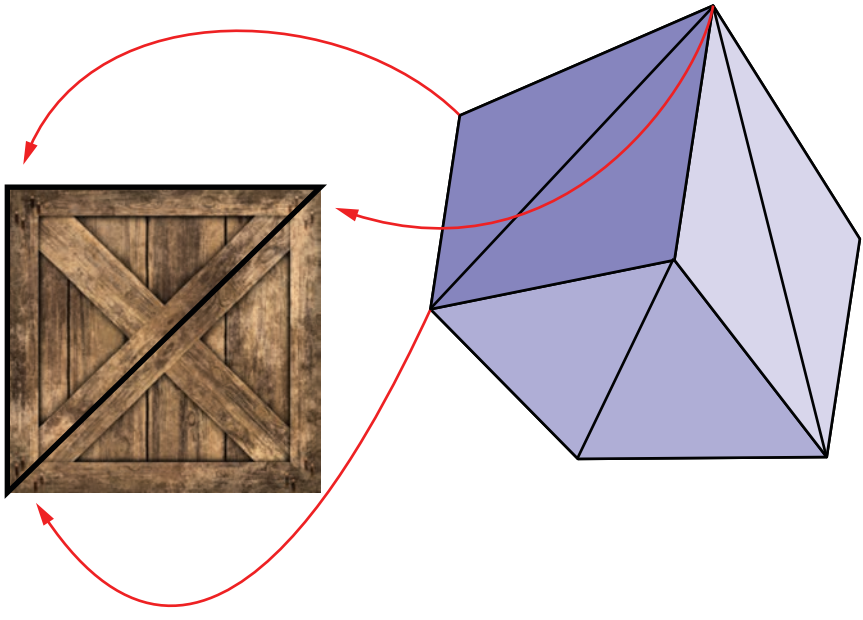


Рис. 14.2. Сопоставление точек на текстуре с вершинами треугольника

Основная идея наложения текстур проста: вычислить координаты (u, v) для каждого пикселя треугольника, получить соответствующий тексель и закрасить пиксель его цветом. Но модель указывает координаты u и v только для трех вершин треугольника, а нам они нужны для каждого пикселя...

Вы уже должны понемногу вникать в происходящее. Да, здесь снова появляется наш старый добрый друг — линейная интерполяция. Можно использовать отображение атрибутов, чтобы интерполировать значения u и v по всей грани треугольника. Это даст нам (u, v) для каждого пикселя. С их помощью мы сможем вычислить (tx, ty), получить тексель, применить затенение и закрасить пиксель итоговым цветом. Результат всего этого можно посмотреть на рис. 14.3.

Результат получается посредственный. Внешне ящики выглядят неплохо. Но если присмотреться к диагональным доскам, мы увидим, что они деформировались. Что же пошло не так?

Как и в главе 12, мы снова сделали неявное ошибочное допущение: то, что u и v изменяются по экрану линейно. Это не так. Рассмотрим стену очень длинного коридора, закрашенную чередующимися черными и белыми полосами. Чем дальше будет стена, тем тоньше будут полосы. Если мы сделаем так, чтобы координата u изменялась линейно относительно x', то получим неверный результат, как показано на рис. 14.4.

Рис. 14.3. При применении к объектам текстура выглядит деформированной

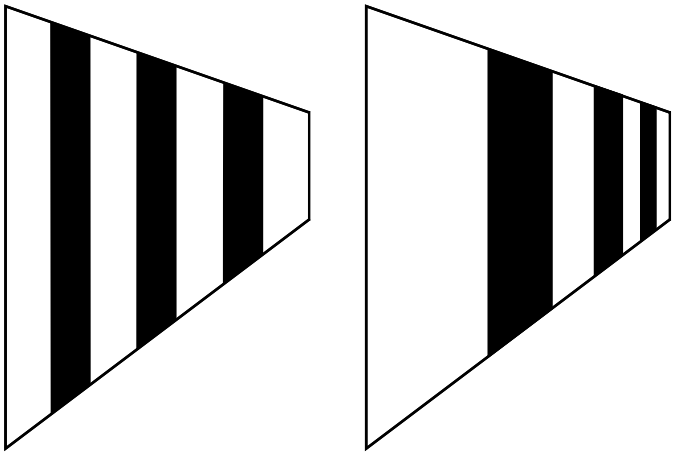


Рис. 14.4. Линейная интерполяция *u* и *v* *(слева)* не дает ожидаемого результата с правильной перспективой *(справа)*

Ситуация схожа с той, что была у нас в главе 12, и решение тоже будет очень похожим: несмотря на то что u и v — не линейные в координатах экрана, такими являются $\frac{u}{z}$ и $\frac{v}{z}$ (подтверждение этому будет похоже на подтверждение $\frac{1}{z}$: представьте, что u изменяется линейно в 3D-пространстве, и подставьте x и y с их выражениями для области экрана). У нас уже есть интерполированные значения $\frac{1}{z}$ в каждом пикселе, этого достаточно для интерполяции $\frac{u}{z}$ и $\frac{v}{z}$ и получения назад u и v:

$$u = \frac{u/z}{1/z};$$

$$v = \frac{v/z}{1/z}.$$

Как видно из рис. 14.5, это даст ожидаемый результат.

Рис. 14.5. Линейная интерполяция u/z и v/z дает верные с позиции перспективы результаты

На рис. 14.6 сравниваются оба результата, чтобы можно было увидеть разницу.

Рис. 14.6. Сравнение результата для линейных *u* и *v* *(слева)* с результатом для линейных *u/z* и *v/z* *(справа)*

Живая реализация алгоритма находится по ссылке https://gabrielgambetta.com/cgfs/textures-demo.

Эти примеры смотрятся красиво, потому что размер текстуры и пиксельный размер треугольников под ней приблизительно одинаковые. А если треугольник будет в несколько раз больше или меньше текстуры? Эту ситуацию мы разберем следующей.

Билинейная фильтрация

Предположим, что камеру разместили очень близко к одному из кубов. Мы увидим что-то похожее на рис. 14.7.

Изображение выглядит очень пикселизованным. Так получилось, потому что у треугольника на экране больше пикселей, чем у текстуры текселей, поэтому каждый тексель накладывается на множество смежных пикселей.

Рис. 14.7. Отрисовка текстурированного объекта вблизи

Мы интерполируем координаты текстуры u и v, вещественные значения в диапазоне от 0,0 до 1,0. Позднее с помощью размеров w и h мы отобразим координаты u и v в координаты текселей tx и ty, умножив их на w и h. Но так как текстура — это массив пикселей с целочисленными индексами, мы округляем tx и ty вниз до ближайшего целого числа. Такая базовая техника называется *фильтрацией по ближайшему соседу*.

Даже если (u, v) изменяется по грани треугольника плавно, получающиеся координаты текселей «перескакивают» от одного целого пикселя к следующему, делая внешний вид объекта пикселизованным, как мы видели на рис. 14.7.

Чтобы результат был получше, вместо округления tx и ty вниз можно интерпретировать дробную координату текселя (tx, ty) как описывающую позиции между четырех целочисленных координат текселей (полученных сочетанием округления tx и ty вверх и вниз). Можно взять четыре цвета окружающих целочисленных текселей и вычислить линейно интерполированный цвет для дробного текселя. Так мы получим заметно более плавный результат (рис. 14.8).

Назовем четыре окружающих пикселя *TL*, *TR*, *BL* и *BR* (верхний левый, верхний правый, нижний левый и нижний правый). Теперь возьмем дробные части tx и ty, присвоив им имена fx и fy. На рис. 14.9 показана описанная (tx, ty) точная позиция *C* вместе с окружающими ее текселями, расположенными в целочисленных координатах. Здесь же указано расстояние от этой позиции до них.

Рис. 14.8. Текстурированный объект, отрисованный с использованием интерполированных цветов

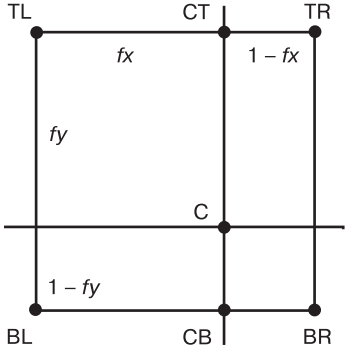


Рис. 14.9. Линейная интерполяция цвета в C на основе четырех окружающих ее текселей

Сначала мы линейно интерполируем цвет в *CT*, которая находится между *TL* и *TR*:

$$CT = (1 - fx)TL + fx\,TR.$$

Заметьте, что вес для *TR* — это *fx*, а не (1 – *fx*). Причина в том, что по мере приближения *fx* к 1,0 *CT* должна приближаться к *TR*. Действительно, если *fx* = 0,0, тогда *CT* = *TL*, а если *fx* = 1,0, тогда *CT* = *TR*.

Вычислить *CB*, расположенную между *TL* и *TR*, можно так же:

$$CB = (1 - fx)BL + fx\,BR.$$

Теперь вычисляем *C*, делая линейную интерполяцию между *CT* и *CB*:

$$C = (1 - fy)BT + fy\,CB.$$

В псевдокоде можно написать функцию для получения интерполированного цвета, соответствующего дробному текселю:

```
GetTexel(texture, tx, ty) {
    fx = frac(tx)
    fy = frac(ty)
    tx = floor(tx)
    ty = floor(ty)

    TL = texture[tx][ty]
    TR = texture[tx+1][ty]
    BL = texture[tx][ty+1]
    BR = texture[tx+1][ty+1]

    CT = fx * TR + (1 - fx) * TL
    CB = fx * BR + (1 - fx) * BL

    return fy * CB + (1 - fy) * CT
}
```

Эта функция использует `floor()`, округляющую число вниз до ближайшего целого, и `frac()`, которая возвращает дробную часть числа и может быть определена как `x – floor(x)`.

Данная техника называется *билинейной фильтрацией* (потому что линейная интерполяция выполняется дважды, по разу для каждого измерения).

MIP-текстурирование

Теперь представим ситуацию, где объект отрисовывается издалека. Здесь у текстуры намного больше текселей, чем у треугольника пикселей. Проблема может быть не так очевидна, я специально подобрал случай, чтобы ее показать.

Рассмотрим квадратную текстуру, в которой в шахматном порядке расположены черные и белые пиксели (рис. 14.10).

Допустим, мы накладываем эту текстуру на квадрат так, чтобы при ее отрисовке на холсте его ширина в пикселях составляла ровно половину ширины текстуры в текселях. Так будет задействована только одна четвертая часть текселей.

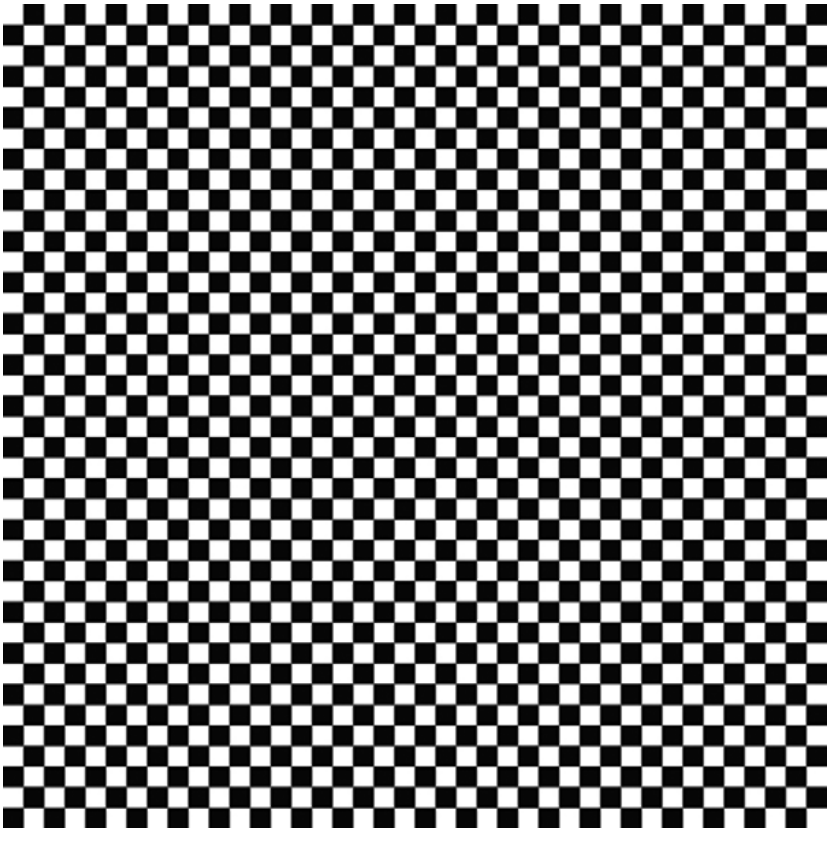


Рис. 14.10. Шахматная черно-белая текстура

Ожидается, что квадрат будет серым. Но при нашем способе наложения текстуры все пиксели могут получиться белыми или черными. Да, если повезет, то мы получим их комбинацию 50/50, но ожидаемый 50%-ный серый не гарантируется. Посмотрите на рис. 14.11, где показан неудачный случай.

Как это исправить? Каждый пиксель квадрата представляет область текселя размером 2×2. Так что можно вычислить усредненный цвет этой области и использовать его для этого пикселя. Усреднение черных и белых пикселей даст нам серый цвет.

Но такой процесс тратит много ресурсов. Представьте: квадрат настолько далеко, что занимает одну десятую от ширины текстуры. Это значит, что каждый пиксель в этом квадрате представляет область текстуры площадью 10×10 текселей. В этом случае нам бы пришлось вычислять среднее между 100 текселями для каждого отрисовываемого пикселя.

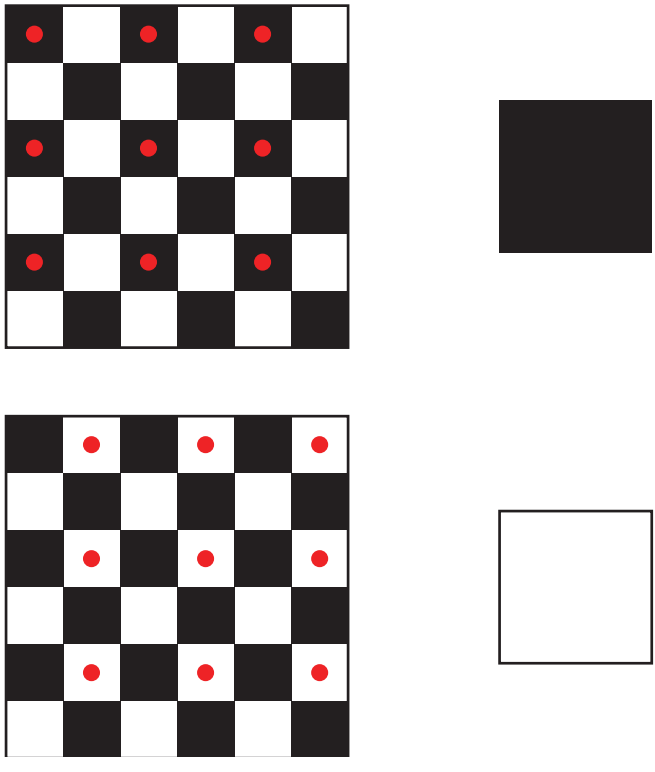


Рис. 14.11. Наложение большой текстуры на малый объект может привести к неожиданным результатам

К счастью, это одна из ситуаций, когда можно заменить объемные вычисления небольшим объемом дополнительной памяти. Вернемся к изначальной картине, где квадрат представлял половину ширины текстуры. Мы можем не вычислять каждый раз средний из четырех текселей для каждого пикселя. Вместо этого мы можем *предварительно вычислить* уменьшенную вдвое текстуру, в которой каждый тексель будет средним от соответствующих четырех текселей в исходной текстуре. Теперь, когда нам нужно будет отрисовать пиксель, мы просто будем находить тексель в этой уменьшенной текстуре или даже применять билинейную фильтрацию, как в прошлом разделе. Так, при усреднении четырех пикселей наш рендеринг будет качественнее, но заплатим мы за это всего одним поиском по текстуре. Для этого нужно чуть больше времени (например, при загрузке текстуры) и намного больше памяти (для сохранения текстуры в полном и половинчатом размерах), но результат того стоит.

А как насчет сценария с размером 10×10, о котором говорилось выше? Мы разовьем эту технику и предварительно вычислим версии 1/4, 1/8 и 1/16 от оригинальной

текстуры (при желании вплоть до текстуры 1×1). Дальше во время рендеринга треугольника мы используем текстуру с подходящим по размеру масштабом. Так мы получим все выгоды без дополнительных затрат для времени выполнения.

Эта мощная техника называется *MIP-текстурированием* (от лат. *multum in parvo* — «многое в малом»). Для вычисления всех этих уменьшенных текстур нужна дополнительная память, но на удивление меньше, чем можно ожидать.

Допустим, исходная область текстуры в текселях — это A, а ее ширина — это w. Половинчатая ширина текстуры будет равна $\frac{w}{2}$, но нужно всего $\frac{A}{4}$ текселей. Поделенная вчетверо ширина потребует $\frac{A}{16}$ текселей и т. д. На рис. 14.12 показана исходная текстура и первые три уменьшенные версии.

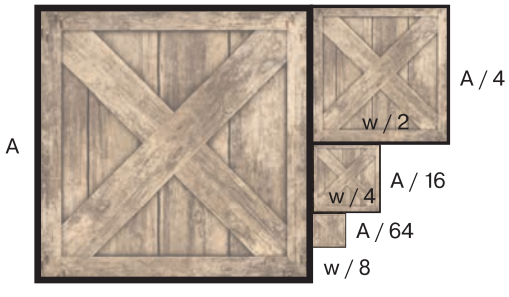


Рис. 14.12. Текстура и ее уменьшающиеся MIP-карты

Сумму размеров текстур можно выразить как бесконечную последовательность:

$$A + \frac{A}{4} + \frac{A}{16} + \frac{A}{64} + \ldots = \sum_{n=0}^{\infty} \frac{A}{4^n}.$$

Она сходится к $A\frac{3}{4}$ либо к $A \cdot 1{,}3333$. Это значит, что все уменьшенные текстуры, вплоть до размера 1×1 тексель, занимают всего на $1/3$ больше места, чем их оригинал.

Трилинейная фильтрация

Пойдем еще дальше. Представьте себе объект вдали от камеры. Мы отрисовываем его с помощью самого подходящего уровня MIP-карты.

А теперь представьте, что камера наезжает на этот объект. В какой-то момент выбор подходящего уровня MIP-карты изменится и возникнет слабое, но заметное различие.

Мы выбираем тот уровень MIP-карты, который ближе всего соответствует относительному размеру текстуры и квадрата. Например, для квадрата в 10 раз меньше текстуры мы можем выбрать уровень MIP-карты в 8 раз меньше исходной текстуры и применить к ней билинейную фильтрацию. Но можно рассмотреть и *два* уровня MIP-карт, больше всего подходящих относительному размеру (в этом случае они будут меньше в 8 и 16 раз). Между ними нужно провести линейную интерполяцию, определяемую «расстоянием» между соотношением размера MIP-карты и фактическим соотношением размеров.

Цвета из каждого уровня MIP-карты билинейно интерполируются, и мы применяем поверх этого еще одну линейную интерполяцию, поэтому такая техника называется *трилинейной интерполяцией*.

Итоги главы

В этой главе мы сильно улучшили качество нашего растеризатора. До этого каждый треугольник мог быть только одного цвета — теперь мы можем отрисовывать на них изображения разной сложности.

Теперь вы умеете и делать изображения текстурированных треугольников четче, независимо от соотношения их размеров с размерами текстуры. Вы познакомились с техникой билинейной фильтрации, MIP-текстурированием и трилинейной фильтрацией, которые использовали для решения самых распространенных случаев падения качества текстур.

15

Расширение растеризатора

Вторую часть книги я завершу, как и первую, набором возможных расширений растеризатора из предыдущих глав.

Карты нормалей

В главе 13 вы видели, как сильно влияют векторы нормалей поверхности на ее внешность. При верном выборе нормалей мы можем превратить граненый объект в плавно изогнутый. Все из-за того, что грамотный выбор нормалей определяет правильное взаимодействие света с поверхностью. К сожалению, с помощью интерполяции нормалей, помимо придания плавности изгибов поверхностям, больше мы особо ничего сделать не можем.

В главе 14 мы узнали, как можно сымитировать детали поверхности, «закрашивая» ее. Эта техника называется *наложением текстур* и дает широкий и точный контроль над внешним видом поверхности. Но она не меняет форму треугольников — они по-прежнему остаются плоскими.

Карты нормалей совмещают в себе обе идеи. Нормали можно использовать для изменения способа взаимодействия света с поверхностью, меняя ее наблюдаемую форму. С помощью отображения атрибутов мы можем присваивать разным частям треугольника разные значения атрибута. При совмещении этих двух идей можно определить нормали поверхностей на уровне пикселей. Для этого мы ассоциируем *карту нормалей* с каждым треугольником.

Карта нормалей как карта текстур, но ее элементы — не цвета, а векторы нормалей. Во время рендеринга мы не вычисляем интерполированную нормаль, как в шейдинге по Фонгу. Через карту мы получаем вектор нормали для конкретного отрисовываемого пикселя, точно так же, как получается для него цвет при отображении текстур. Потом этот вектор используется для вычисления освещения в пикселе.

На рис. 15.1 показана плоская поверхность с применением карты текстур и эффекты от разной направленности света при дополнительном применении карты нормалей.

а) Без карты нормалей

б) С картой нормалей и освещением слева

в) С картой нормалей и освещением справа

Рис. 15.1. Эффект от применения карты нормалей на плоской поверхности

Все три изображения на рис. 15.1 отображают рендеринг плоского квадрата (то есть двух треугольников) с текстурой *(а)*. При добавлении карты нормалей

и подходящего попиксельного затенения мы создаем иллюзию дополнительной геометрической детали. На рисунках *(б)* и *(в)* затенение ромбов зависит от направления падения света, и наш мозг начинает «видеть» их объем.

Но нужно кое-что учитывать. Во-первых, ориентация векторов в карте нормалей относительна поверхности треугольника, к которому они применяются. Используемая для этого система координат называется *касательным пространством*, в котором две из осей (обычно X и Z) расположены касательно к (то есть вложены в) поверхности, а оставшийся вектор перпендикулярен к ней. Во время рендеринга выраженный в пространстве камеры вектор нормали треугольника изменяется согласно вектору в карте нормалей для получения окончательного вектора нормалей, который можно использовать в уравнениях освещенности. Так карта будет независимой от позиции и ориентации объекта в сцене.

Во-вторых, очень часто карты нормалей кодируют в виде текстур, отображая значения X, Y и Z в значения R, G и B. Это придает им очень характерный фиолетовый цвет. Как сочетание красного и синего, фиолетовый кодирует плоские поверхности. На рис. 15.2 показана карта нормалей, используемая в примерах на рис. 15.1.

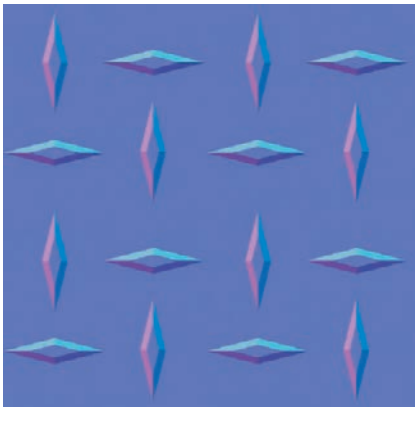


Рис. 15.2. Карта нормалей, используемая для примеров на рис. 15.1, представлена в виде RGB-текстуры

С помощью этой техники мы можем сделать поверхности в сцене реалистичнее. Но есть ограничения. Например, плоские поверхности остаются плоскими, поэтому она не может изменять силуэт объекта. Иллюзия рушится и когда полученная отображением нормалей поверхность просматривается под крайним углом или в упор. Когда признаки в карте нормалей оказываются слишком велики относительно размера поверхности, техника тоже не работает. Лучше всего она подходит для тонких деталей, таких как поры кожи, фактура оштукатуренной стены или шероховатости апельсиновой кожуры. Именно поэтому технику называют *рельефным текстурированием*.

Наложение карты среды

Одна из наиболее поразительных особенностей созданного нами трассировщика лучей — возможность показывать отражающие друг друга объекты. Это можно сделать и в растеризаторе, но выглядеть она будет не так убедительно.

Представьте, что у нас есть сцена в виде комнаты в доме и посреди нее мы хотим отрисовать отражающий объект. Для каждого пикселя поверхности этого объекта нам известны 3D-координаты его точки, нормаль поверхности в ней. Еще мы знаем позицию камеры, поэтому тоже можем вычислить вектор обзора этой точки. Его можно отразить относительно нормали поверхности для получения вектора отражения, как мы делали в главе 4.

В этот момент нам нужно знать цвет света из направления вектора отражения. В случае трассировщика лучей нам было бы достаточно просто отследить идущий в этом направлении луч. Но здесь у нас растеризатор, так что же делать?

Наложение карт среды может ответить на этот вопрос. Допустим, перед рендерингом объектов в комнате мы помещаем камеру в ее центр и отрисовываем сцену шесть раз — по разу для каждого перпендикулярного направления (вверх, вниз, влево, вправо, вперед и назад). Можете представить камеру внутри воображаемого куба, каждая сторона которого будет окном просмотра одной из этих отрисовок. Полученные шесть изображений мы сохраним как текстуры. Такой набор называется *картой куба*, отсюда и название техники — *наложение куба*.

Дальше идет отрисовка отражающего объекта. Когда нам становится нужен отраженный цвет, мы просто используем направление отраженного вектора для выбора одной из текстур карты куба и текселя этой текстуры. В итоге мы получаем аппроксимацию цвета, наблюдаемого в этом направлении, — и никакой трассировки лучей!

Есть у этой техники и недостатки. Карта куба захватывает вид сцены из одной точки. Если отражающий объект будет в этой точке, то объекты будут отражены не так, как ожидалось, и станет понятно, что перед нами просто аппроксимация. Особенно видно это будет, если отражающий объект в комнате начнет перемещаться — отраженная сцена не будет подстраиваться под его перемещение.

Это ограничение также предлагает лучшие варианты применения техники. Когда перемещение объекта небольшое по сравнению с размером комнаты, разница между настоящим отражением и предварительно отрисованными картами среды может стать незаметной. Это отлично сработает для сцены с отражающим космическим кораблем в космосе, ведь «комната» (далекие звезды и галактики) бесконечно удалена для любых практических задач.

Еще одна сложность — нам нужно делить объекты в сцене на две категории: статичные, являющиеся частью «комнаты», которые мы видим в отражениях,

и динамичные, способные отражать. В некоторых случаях это может быть очевидно (стены и мебель — часть комнаты, а люди — нет), но даже тогда динамические объекты не будут отражаться от других динамических объектов.

Последний минус относится к разрешению карт куба. Если в трассировщике лучей мы могли трассировать очень точные отражения, то здесь нам нужно идти на компромисс между точностью (текстуры куба с более высоким разрешением дают более четкие отражения) и потреблением памяти (чем выше разрешение текстур, тем больше нужно памяти). Это значит, что карты среды не будут давать такие же четкие отражения, как настоящие отражения при трассировке лучей, особенно при наблюдении отражающих объектов вблизи.

Тени

Наш трассировщик умеет отрисовывать геометрически правильные и четко определенные тени. Для его алгоритма это было естественным расширением. А вот архитектура растеризатора усложняет реализацию теней, но не делает ее невозможной.

Сначала сформулируем задачу, которую хотим решить. Для правильной отрисовки теней при каждом вычислении уравнения освещенности для пикселя и света нужно знать, освещается ли этот пиксель напрямую или находится в тени объекта относительно данного источника света.

В трассировщике узнать об этом мы можем путем трассировки луча от поверхности к источнику света. В растеризаторе же такой возможности нет. Поэтому мы используем другой подход. Здесь у нас два варианта.

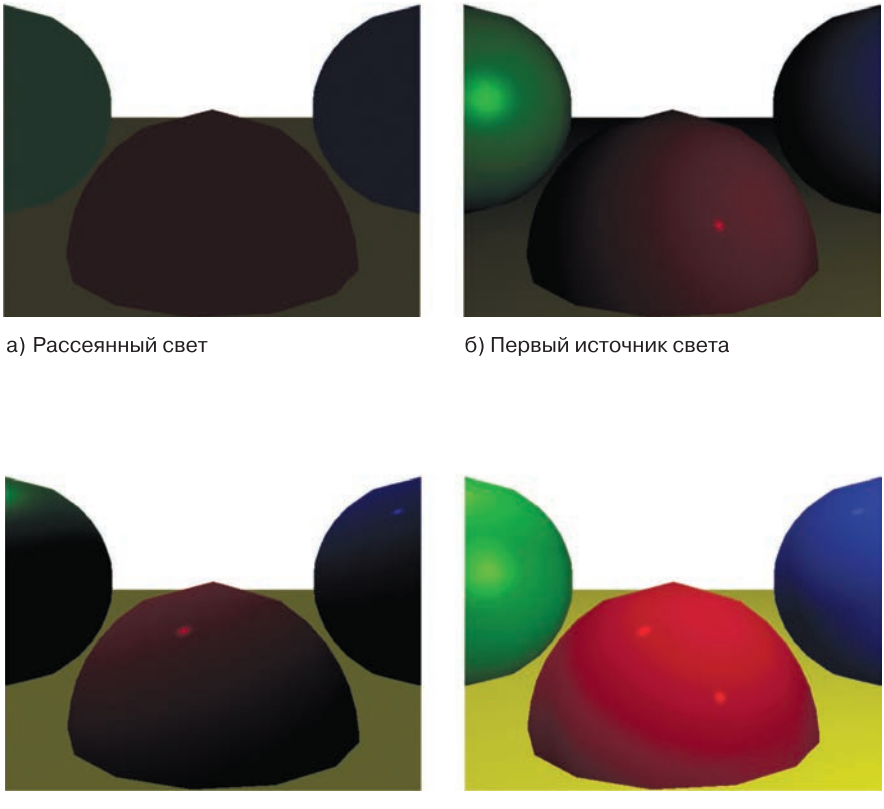
Трафаретные тени

Трафаретные тени (жесткие тени) — это техника отрисовки теней с четкими контурами (например, тени объектов в очень солнечный день).

Наш растеризатор отрисовывает сцену в один *проход*. Он проходит по каждому треугольнику в сцене и отрисовывает его на холсте, каждый раз вычисляя уравнение глобального освещения (для каждого треугольника, каждой вершины или каждого пикселя, что определяется алгоритмом затенения). В итоге на холсте будет заключительная отрисовка сцены.

Начнем с изменения растеризатора для рендеринга сцены в несколько *проходов*, по одному для каждого источника света (включая рассеянный). Как и раньше, в каждый проход будет обрабатываться каждый треугольник, но при вычислении освещения будет учитываться только связанный с этим проходом источник.

Так мы получим набор изображений сцены, освещенной каждым источником света отдельно. Эти изображения мы *совместим*, сложим попиксельно, получив итоговую отрисовку сцены. Финальное изображение будет такое же, как и в версии с одним проходом. На рис. 15.3 показаны три прохода и итоговый вариант нашей эталонной сцены.



а) Рассеянный свет

б) Первый источник света

в) Второй источник света

г) Итоговое общее изображение

Рис. 15.3. Сцена, отрисованная с использованием одного подхода для каждого источника света

Так мы сможем перейти от «отрисовки сцены с тенями от нескольких источников света» до «многократной отрисовки сцены с тенями от одного источника света». А как отрисовать сцену, освещенную одним источником света, оставив пиксели в тени относительно него полностью черными?

Для этого мы задействуем *трафаретный буфер*. Как и буфер глубины, он имеет размеры холста, но элементы в нем — целые числа. Его можно использовать как трафарет для отрисовки. Например, изменив код рендеринга для рисования пикселя на холсте только в случае, когда значение такого элемента в трафаретном буфере равно нулю.

Если настроить этот буфер так, чтобы освещаемые пиксели были со значением нуль, а пиксели в тени имели ненулевое значение, то можно отрисовывать с его помощью только освещенные пиксели.

Создание теневых объемов

Для настройки трафаретного буфера используются *теневые объемы*. Теневой объем в 3D-полигоне «оборачивается» вокруг объема пространства в тени относительно света.

Теневой объем мы построим для каждого объекта, способного отбрасывать тень на сцену. Сначала определим, какие ребра будут частью силуэта объекта: между лицевыми и тыльными треугольниками (для классификации треугольников можно использовать скалярное произведение, как в технике отбрасывания задней грани в главе 12). Каждое полученное ребро мы вытягиваем в противоположном от источника света направлении до бесконечности или просто за края сцены.

Так мы получим стороны теневого объема. Лицевая сторона состоит из лицевых треугольников объекта, а тыльную сторону объема можно вычислить созданием многоугольника, чьи ребра будут дальними ребрами вытянутых сторон.

На рис. 15.4 показан созданный этим способом теневой объем для куба относительно точечного света.

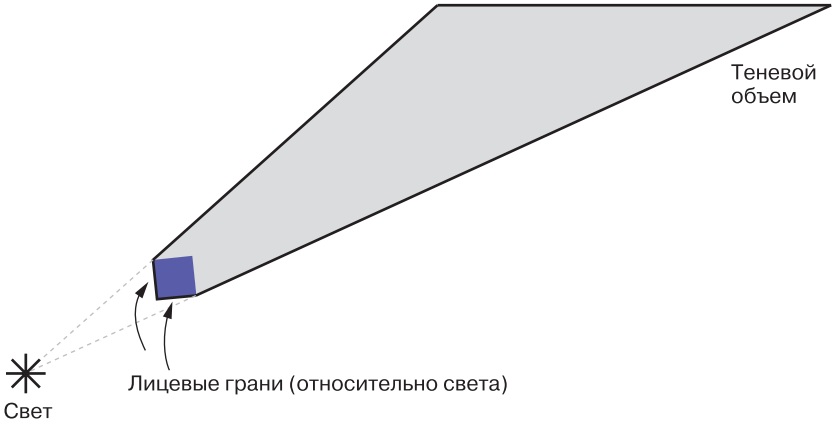


Рис. 15.4. Теневой объем куба относительно точечного света

Дальше мы рассмотрим, как с помощью полученных теневых объемов определять, какой пиксель холста находится в тени относительно источника света.

Подсчет пересечений луча и теневого объема

Представьте луч от камеры, устремляющийся в сцену до достижения поверхности. На своем пути он может пересекать любое количество теневых объемов.

Это можно отслеживать с помощью счетчика, начинающегося с нуля. Каждый раз, когда луч входит в теневой объем, значение счетчика повышается на 1; каждый раз, когда он его покидает, значение счетчика уменьшается. По достижении лучом поверхности мы этот процесс прекращаем и смотрим на счетчик. Если он показывает нуль, значит, луч покинул такое же количество теневых объемов, в какое вошел, и точку нужно осветить. Если же счетчик не равен нулю, значит, луч внутри как минимум одного теневого объема и точка должна быть в тени (рис. 15.5).

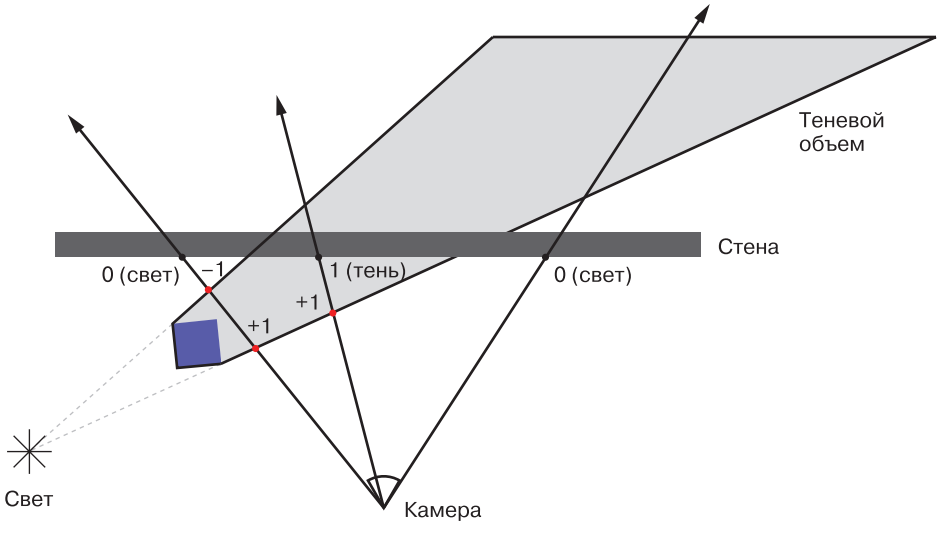


Рис. 15.5. Подсчет пересечений лучей с теневыми объемами показывает, освещены точки на их путях или находятся в тени

Но все это работает, только если сама камера не находится внутри теневого объема. Если луч начнется внутри него и не покинет его до момента достижения поверхности, наш алгоритм ошибочно решит, что поверхность освещена.

Можно проверить выполнение этого условия и скорректировать счетчик, но подсчет количества теневых объемов, в которых находится точка, — дорогостоящая

операция. Можно обойти это ограничение. Этот способ будет проще и дешевле, хоть и не очень интуитивен.

Лучи бесконечны, но теневые объемы — нет. Значит, луч всегда начинается и заканчивается вне теневого объема и поэтому всегда входит в теневой объем столько же раз, сколько его покидает. Другими словами, счетчик для всего луча всегда должен быть равен нулю.

Представим, что отслеживаем пересечения луча и теневого объема *после* достижения лучом поверхности. Если на счетчике нуль, это значение должно быть нулевым и *до того*, как луч ее достиг. Если же значение счетчика ненулевое, на другой стороне этой поверхности у него должно быть противоположное значение.

Получается, подсчет пересечений луча и теневого объема до достижения лучом поверхности равнозначен подсчету таких пересечений после. Но в этом случае нам не нужно беспокоиться о позиции камеры! На рис. 15.6 показано, как эта техника всегда дает верный результат.

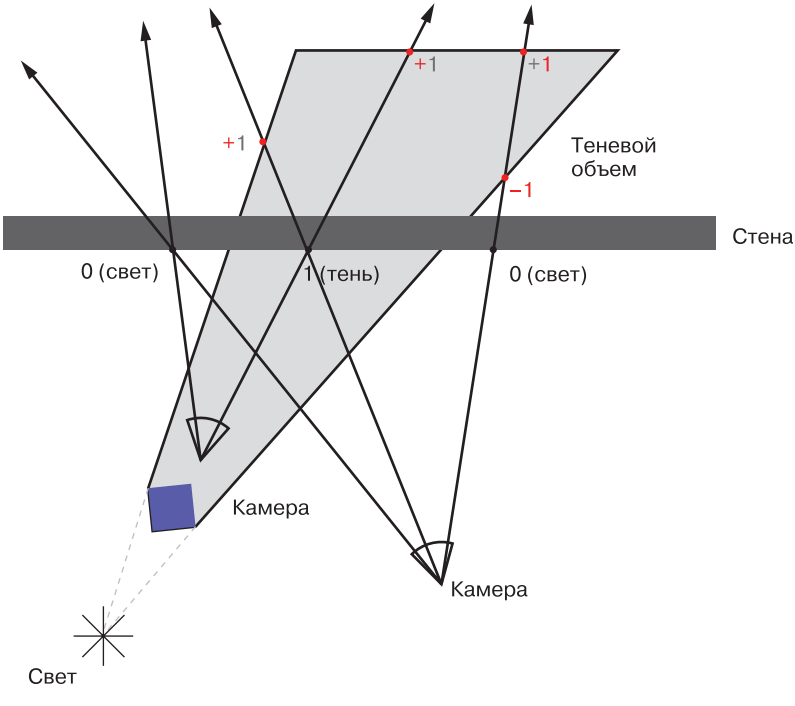


Рис. 15.6. Счетчики имеют нулевое значение для точек, получающих свет, и ненулевое для тех, которые в тени. Положение камеры относительно теневого объема здесь неважно

Настройка трафаретного буфера

Мы работаем с растеризатором, а не трассировщиком, поэтому нужно найти способ сохранить эти счетчики без вычисления любых пересечений лучей и теневых объемов. Это можно сделать с помощью трафаретного буфера.

Для начала отрисовываем сцену, освещенную только рассеянным светом. Такой свет теней не отбрасывает, значит, не придется вносить изменения в растеризатор. В итоге мы получим одно из изображений для составления финальной отрисовки. Но оно еще и даст нам информацию из буфера глубины о глубине сцены с позиции камеры. Этот буфер нужно сохранить — он пригодится дальше.

Теперь для каждого источника света мы проделываем следующие шаги.

1. Рендерим задние грани теневых объемов в трафаретном буфере, инкрементируя его значение, когда пиксель *проваливает* тест буфера глубины. Так мы подсчитаем, сколько раз луч покидает теневой объем после достижения ближайшей поверхности.
2. Рендерим лицевые грани теневых объемов в трафаретном буфере, уменьшая его значение каждый раз, когда пиксель *проваливает* тест буфера глубины. Так мы подсчитаем, сколько раз луч входит в теневой объем по достижении ближайшей поверхности.

Обратите внимание, что во время рендеринга нас интересует только изменение трафаретного буфера. Не нужно записывать пиксели на холст, поэтому не нужно и вычислять освещение или текстурирование. Еще мы не делаем записи в буфер глубины, потому что стороны теневых объемов — это не физические объекты сцены. Вместо этого мы используем буфер глубины во время прохода рассеянного освещения.

После этого в трафаретном буфере будут нули для освещенных пикселей и другие значения для пикселей в тени. Итак, мы отрисовываем сцену как обычно, освещенную одним источником света, соответствующим текущему проходу, вызывая `PutPixel` только для пикселей, для которых трафаретный буфер содержит нуль.

Повторяя этот процесс для каждого источника света, мы получаем набор соответствующих сцене изображений, освещенных каждым источником света, с правильно учтенными тенями. Последний шаг — объединить все эти изображения в итоговую отрисовку сцены, сложив их попиксельно.

Идея использования трафаретного буфера для рендеринга теней зародилась еще в начале 90-х, но у первого ее применения были минусы. Описанный здесь вариант глубинного провала был независимо обнаружен несколько раз в 1999 и 2000 годах. Впервые его заметил Джон Кармак во время работы над Doom 3, поэтому этот вариант также известен как *реверс Кармака*.

Теневая карта

Еще одна известная техника рендеринга теней в растеризаторе называется *теневой картой* (или *мягкими тенями*). Она отрисовывает тени с менее отчетливыми краями (например, тени от объектов в облачный день).

Напомню, что мы пытаемся понять, как при наличии источника света и точки узнать, освещена ли точка им (определить, есть ли между источником света и рассматриваемой точкой объект).

При использовании трассировщика мы прослеживали луч из точки к источнику света. Так мы в некотором смысле спрашиваем, «видит» ли эта точка источник или «видит» ли сам источник ее.

И мы подходим к основной идее теневой карты. Мы отрисовываем сцену с точки зрения источника света, сохраняя буфер глубины. По аналогии с созданием карт среды мы отрисовываем сцену шесть раз и получаем шесть буферов глубины *(теневые карты)*. С их помощью мы можем определить расстояние до ближайшей поверхности, которую источник света может «видеть» в любом заданном направлении.

Ситуация усложняется для направленных источников, ведь у них нет позиции, из которой можно сделать рендеринг. Вместо этого нам нужно отрисовать сцену из *направления*. Здесь следует использовать не перспективную проекцию, а *ортографическую*. В первом случае каждый луч начинается в точке. В случае же ортографической проекции и направленных источников света лучи параллельны друг другу и распространяются в одном направлении.

Чтобы определить, находится ли точка в тени, мы вычисляем расстояние и направление от источника света к этой точке. Результат мы используем для поиска соответствующей записи в теневой карте. Если это значение глубины меньше расстояния от точки до источника света, значит, есть поверхность ближе к нему, чем освещаемая нами точка. Получается, эта точка находится в тени этой поверхности, если же наоборот — источник света может «видеть» точку без препятствий, значит, она им освещается.

Заметьте, что у теневых карт ограниченное разрешение, которое обычно ниже разрешения холста. В зависимости от расстояния и относительной ориентации точки и света тени могут получиться пикселизованными. Во избежание этого можно дополнительно сделать выборку по окружающим записям глубины, чтобы определить, находится ли точка на краю тени (об этом скажет рассогласованность значений в этих записях). Если это так, то можно использовать технику вроде билинейной фильтрации, как в главе 14, чтобы получить значение между 0,0 и 1,0. Так мы узнаем, *насколько* точка видима из позиции источника света, и умножим это значение на яркость света. Созданные с помощью теневой карты тени приобретут характерный

размытый вид. В других подходах во избежание пикселизованного вида теней выборка из теневой карты делается иначе. Например, с помощью техники фильтрации *percentage closer filtering (PCF)*.

Итоги главы

Здесь мы снова кратко ознакомились с несколькими идеями, которые можно разобрать самостоятельно. С их помощью можно расширить возможности нашего растеризатора, приблизив их к трассировщику лучей, но с сохранением преимуществ в быстродействии. Без компромиссов никогда не обходится. Здесь, в зависимости от алгоритма, они проявляются в виде менее точных результатов или повышенного потребления памяти.

Послесловие

Мои поздравления! Теперь вы знаете, как работает 3D-рендеринг. Вы разработали трассировщик лучей и растеризатор, а еще познакомились с алгоритмами и математикой, стоящими за ними.

Но еще во введении я говорил, что невозможно охватить всю область 3D-рендеринга в одной книге. Вот несколько тем, которые вы можете разобрать для повышения своего уровня.

- **Глобальное освещение, включая метод излучения (radiosity) и трассировку пути.** Углубление в тему рассеянного света.
- **Рендеринг, основанный на физике.** Освещение и модели затенения, которые выглядят красиво и реалистично.
- **Рендеринг вокселей.** Представьте Minecraft или снимки МРТ в больницах.
- **Алгоритмы уровня детализации.** Сюда относятся техники офлайн- и динамического упрощения полигональной сетки, импосторы и билборды. Эти алгоритмы объясняют, как эффективно отрисовывать леса с миллиардами растений, толпы из миллионов людей или чрезвычайно детализированные 3D-модели.
- **Структуры ускоряющие.** Сюда входят деревья двоичного разбиения пространства, k-мерные деревья, квадродеревья и октодеревья. С их помощью можно отрисовывать массивные сцены, например целые города.
- **Рендеринг ландшафта.** Научитесь качественно отрисовывать модель местности размером со страну и детализированную до масштаба человека.
- **Атмосферные эффекты и системы частиц.** Работа с туманом, дождем и дымом, с материалами вроде травы или волос.
- **Освещение на основе изображений.** Аналогично картам среды, но используется для диффузного освещения.
- **Расширенный динамический диапазон, гамма-коррекция.** Углубление в тему представления цвета.
- **Каустика.** Кратко можно описать как «игра света на дне плавательного бассейна».

- **Процедурная генерация текстур и моделей.** Научитесь добавлять в сцены вариативность и делать их бесконечно большими.
- **Аппаратное ускорение.** Использование OpenGL, Vulkan, DirectX и других инструментов для выполнения графических алгоритмов на GPU.

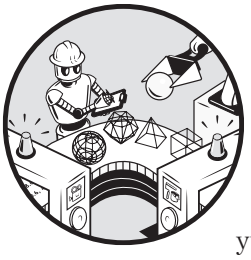
Есть еще много тем, причем все это лишь касательно 3D-рендеринга! Компьютерная графика в целом еще обширнее. Вот наиболее интересные для ознакомления области.

- **Рендеринг шрифтов.** Эта тема гораздо сложнее, чем можно подумать.
- **Сжатие изображений.** Как сохранять изображения в меньшем объеме памяти.
- **Обработка изображений (например, преобразование и фильтрация).** Представьте фильтры в «Инстаграме».
- **Распознание изображений.** На картинке кошка или собака?
- **Рендеринг кривых, включая кривые Безье и сплайны.** Узнайте, что на самом деле представляют собой эти странные стрелки на изогнутых линиях в программах рисования!
- **Вычислительная фотография.** Как камера вашего телефона может делать хорошие снимки почти без света?
- **Сегментация изображений.** Чтобы «размыть задний план» во время видеовызова, понадобится определить, какие пиксели относятся к заднему плану, а какие — нет.

Еще раз поздравляю с совершением первого шага в мир компьютерной графики. Теперь ваша очередь выбирать, куда по нему идти дальше!

Приложение

Линейная алгебра



Это приложение — ваша шпаргалка по линейной алгебре. Здесь вы найдете наборы инструментов, их свойства и возможности применения. Теорию по этой теме вы можете найти в любом учебнике по линейной алгебре.

Здесь мы сосредоточимся на двухмерной и трехмерной алгебре. Именно эти темы пригодятся вам в книге.

Точки

Точка — это позиция внутри системы координат.

Выражается она как последовательность чисел в скобках, например (4, 3). Для обозначения точек используются заглавные буквы, например P или Q.

Каждое число, выражающее точку, называется *координатой*. Их количество представляет *размерность* точки. Точка с двумя координатами называется двухмерной или 2D.

Порядок описывающих координаты чисел важен. Сочетание (4, 3) не будет равнозначно сочетанию (3, 4). По соглашению координаты в двухмерном пространстве называются x и y, а в трехмерном — x, y и z.

Следовательно, у точки (4, 3) координатой x будет 4, а координатой y — 3. На рис. П.1 показана P — двухмерная точка с координатами (4, 3).

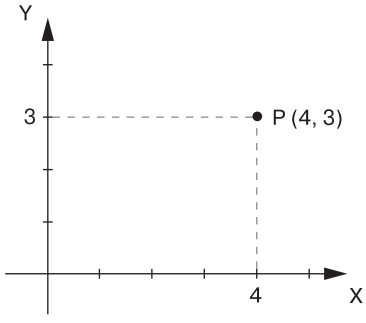


Рис. П.1. Двухмерная точка *P* с координатами (4, 3)

Конкретные координаты точки можно обозначать и с помощью нижнего индекса, например P_x или Q_y. Значит, точку P можно записать как (P_x, P_y, P_z), когда это удобно.

Векторы

Вектор — это направление, разница между двумя точками. Чтобы было понятнее, представьте вектор как стрелку, соединяющую одну точку с другой. Еще его можно представить как инструкцию для попадания из одной точки в другую.

Представление векторов

Вектор, как и точка, выражается как набор чисел в скобках и обозначается с помощью заглавной буквы. Поэтому для отличия над именем вектора добавляется стрелка. Например, (2, 1) — это вектор, который можно назвать $\vec{A}$. На рис. П.2 показаны два равных вектора, $\vec{A}$ и $\vec{B}$.

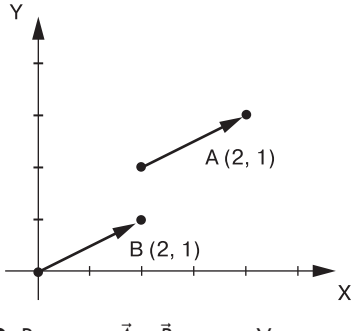


Рис. П.2. Векторы $\vec{A}$ и $\vec{B}$ равны. У них нет позиции

Несмотря на использование одинакового с точками способа представления, векторы не выражают и не имеют позиции. Они выражают *разницу* между двумя позициями.

Когда у вас чертеж как на рис. А.2, вам нужно где-то нарисовать оба вектора, хотя по факту $\vec{A}$ и $\vec{B}$ равны, потому что представляют одинаковое смещение.

Кроме того, точка (2, 1) и вектор (2, 1) не связаны. Этот вектор, конечно, проходит от (0, 0) в (2, 1), но также верно сказать, что он проходит из, например, (5, 5) в (7, 6).

Векторы характеризуются своим *направлением* (углом, под которым они направлены) и *модулем* (то есть протяженностью).

Модуль вектора

Модуль вектора можно вычислить на основе его координат. Модуль также называется *длиной* или *нормой* вектора. Обозначается он вертикальными линиями, например $\left|\vec{V}\right|$, и вычисляется так:

$$\left|\vec{V}\right| = \sqrt{V_x^2 + V_y^2 + V_z^2}.$$

Вектор с модулем, равным 1, называется *единичным вектором*.

Операции с точками и векторами

Мы рассмотрели определения точек и векторов. Пора узнать, что же с ними делать.

Вычитание точек

Вектор — это разница между двумя точками. Другими словами, его можно получить вычитанием между двумя точками.

$$\vec{V} = P - Q.$$

В этом случае можно рассматривать $\vec{V}$ как проходящий из Q в P (рис. П.3).

В алгебраическом представлении мы вычитаем каждую координату по отдельности:

$$(V_x, V_y, V_z) = (P_x, P_y, P_z) - (Q_x, Q_y, Q_z) = (P_x - Q_x, P_y - Q_y, P_z - Q_z).$$

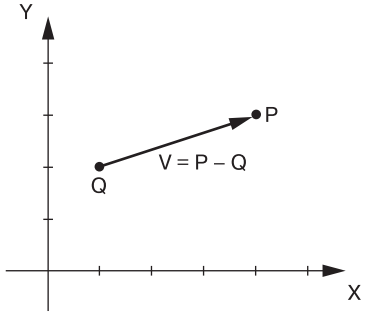


Рис. П.3. Вектор $\vec{V}$ представляет разницу между P и Q

Сложение точки и вектора

Уравнение выше можно переписать по координатам:

$$V_x = P_x - Q_x;\ V_y = P_y - Q_y;\ V_z = P_z - Q_z.$$

Это всего лишь числа, значит, к ним применимы все стандартные правила, и вполне можно сделать так:

$$Q_x + V_x = P_x;\ Q_y + V_y = P_y;\ Q_z + V_z = P_z.$$

И снова сгруппировать координаты:

$$Q + \vec{V} = P.$$

Проще говоря, можно прибавить вектор к точке и получить новую точку. В этом есть логика и геометрический смысл. При наличии стартовой позиции (точки) и смещения (вектора) мы получаем новую позицию (другую точку). На рис. П.4 показан пример.

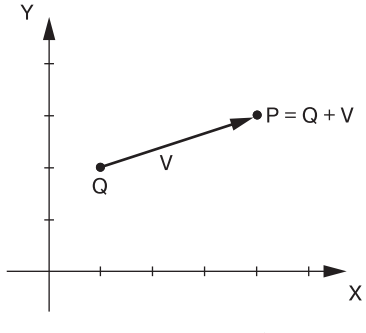


Рис. П.4. Прибавление $\vec{V}$ к Q дает *P*

Сложение векторов

Можно сложить и два вектора. В геометрическом смысле нужно просто представить, что один вектор помещается за другим, как показано на рис. П.5.

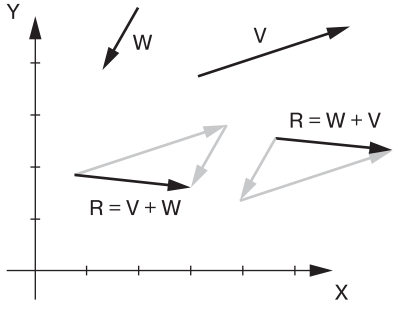


Рис. П.5. Сложение двух векторов. Операция сложения коммутативна. Помните, что у векторов нет позиции

Отмечу, что сложение векторов является коммутативным, то есть порядок операндов значения не имеет. В этом чертеже мы видим, что $\vec{V}+\vec{W}=\vec{W}+\vec{V}$.

В алгебраическом представлении все координаты складываются отдельно:

$$\vec{V}+\vec{W}=\left(V_x,V_y,V_z\right)+\left(W_x,W_y,W_z\right)=\left(V_x+W_x,V_y+W_y,V_z+W_z\right).$$

Умножение вектора на число

Вектор можно умножить на число. Это называется *умножением на скаляр* (не путать со скалярным произведением, определенным ниже). Так вектор становится длиннее или короче, как можно видеть на рис. П.6.

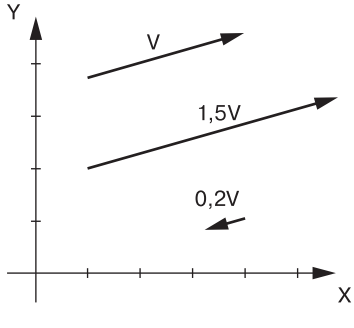


Рис. П.6. Умножение вектора на число

Если число будет отрицательным, то направление вектора изменится на противоположное. Но умножение вектора на число никогда не изменит его наклона — то есть он продолжит лежать вдоль той же линии.

В алгебраическом представлении умножение координат происходит по отдельности:

$$k\vec{V} = k\left(V_x, V_y, V_z\right) = \left(kV_x, kV_y, kV_z\right).$$

Вектор можно разделить на некоторое число. Как и в случае с числами, деление на k равнозначно умножению на $\frac{1}{k}$. Делить на нуль нельзя.

Одним из применений такого умножения и деления является *нормализация* — преобразование вектора в единичный. В результате такой операции модуль вектора изменяется на 1, но его свойства остаются прежними. Для этого нужно просто разделить вектор на его длину:

$$\vec{V}_{\text{норм}} = \frac{\vec{V}}{\left|\vec{V}\right|}.$$

Перемножение векторов

Векторы можно перемножать. Есть много способов это сделать. Мы же сосредоточимся на двух видах умножения, которые для нас наиболее полезны: скалярном и векторном произведениях.

Скалярное произведение

Скалярное произведение двух векторов дает число (скаляр). При этом он записывается между угловыми скобками, например $\left\langle \vec{V}, \vec{W} \right\rangle$.

В алгебраическом представлении координаты по отдельности перемножаются и складываются.

$$\left\langle \vec{V}, \vec{W} \right\rangle = \left\langle \left(V_x, V_y, V_z\right), \left(W_x, W_y, W_z\right) \right\rangle = V_x \cdot W_x + V_y \cdot W_y + V_z \cdot W_z$$

В геометрии скалярное произведение $\vec{V}$ и $\vec{W}$ определяется их длинами и углом α между ними. Формула ниже аккуратно сочетает линейную алгебру и тригонометрию:

$$\left\langle \vec{V}, \vec{W} \right\rangle = \left|\vec{V}\right| \cdot \left|\vec{W}\right| \cos(\alpha).$$

По любой из этих формул видно, что скалярное произведение коммутативно (то есть $\left\langle \vec{V}, \vec{W} \right\rangle = \left\langle \vec{W}, \vec{V} \right\rangle$) и дистрибутивно относительно умножения на скаляр (то есть $k \cdot \left\langle \vec{V}, \vec{W} \right\rangle = \left\langle k \cdot \vec{V}, \vec{W} \right\rangle$).

У второй формулы интересное следствие: если $\vec{V}$ и $\vec{W}$ будут перпендикулярны, то cos(α) = 0, а значит, $\langle \vec{V}, \vec{W} \rangle$ тоже равно нулю. Если $\vec{V}$ и $\vec{W}$ единичные векторы, то $\langle \vec{V}, \vec{W} \rangle$ всегда будет между –1 и 1. При этом 1 будет означать их равенство, а –1 — противоположность.

По второй формуле тоже понятно, что скалярное произведение можно использовать для вычисления угла *между* двумя векторами:

$$\alpha = \cos^{-1}\left(\frac{\langle \vec{V}, \vec{W} \rangle}{|\vec{V}| \cdot |\vec{W}|}\right).$$

Обратите внимание, что скалярное произведение вектора с самим собой, $\langle \vec{V}, \vec{V} \rangle$, сокращается до квадрата его длины:

$$\langle \vec{V}, \vec{V} \rangle = V_x^2 + V_y^2 + V_z^2 = |\vec{V}|^2.$$

Это предполагает еще один способ вычисления длины вектора, а именно как квадратного корня из его скалярного произведения с самим собой:

$$|\vec{V}| = \sqrt{\langle \vec{V}, \vec{V} \rangle}.$$

Векторное произведение

Векторное произведение двух векторов дает другой вектор. Записывается эта операция с помощью крестика, например $\vec{V} \times \vec{W}$.

Произведение векторов — это вектор, перпендикулярный к ним обоим. В этой книге оно определено только для 3D-векторов (рис. П.7).

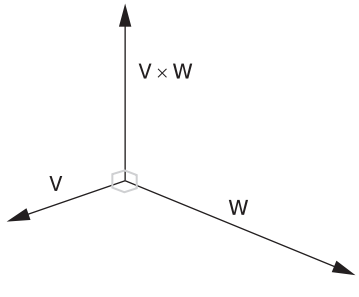


Рис. П.7. Произведение двух векторов — это вектор, перпендикулярный к ним обоим

Вычисление здесь сложнее, чем в скалярном произведении. Если $\vec{R} = \vec{V} \times \vec{W}$, то:

$$R_x = V_y \cdot W_z - V_z \cdot W_y;$$

$$R_y = V_z \cdot W_x - V_x \cdot W_z;$$

$$R_z = V_x \cdot W_y - V_y \cdot W_x.$$

Векторное произведение антикоммутативно, то есть $\vec{V} \times \vec{W} = -\left(\vec{W} \times \vec{V}\right)$.

Эта операция используется для вычисления *вектора нормали* поверхности — единичного вектора, перпендикулярного к этой поверхности. Для этого берутся два вектора на поверхности, вычисляется их векторное произведение, а результат нормализуется.

Матрицы

Матрица — это прямоугольный массив чисел. В этой книге матрицы используются для *преобразования*, их можно применять к точкам или векторам. Мы называем их с помощью заглавной буквы, например *M*. Таким же способом мы обозначаем точки, но по контексту всегда будет понятно, что имеется в виду.

Описывается матрица ее размером в строках и столбцах. Например, вот матрица 3×4:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ -3 & -6 & 9 & 12 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Операции с матрицами

Посмотрим, какие операции можно производить с матрицами и векторами.

Сложение матриц

Две матрицы можно сложить при условии их одинакового размера. Такое сложение выполняется поэлементно:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} + \begin{pmatrix} j & k & l \\ m & n & o \\ p & q & r \end{pmatrix} = \begin{pmatrix} a+j & b+k & c+l \\ d+m & e+n & f+o \\ g+p & h+q & i+r \end{pmatrix}.$$

Умножение матрицы на число

Матрицу можно умножить на число. Для этого нужно умножить на это число каждый ее элемент:

$$n\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} na & nb & nc \\ nd & ne & nf \\ ng & nh & ni \end{pmatrix}.$$

Перемножение матриц

Матрицы можно умножать между собой, если их размеры совместимы: то есть количество столбцов в первой должно совпадать с количеством строк во второй. Например, можно умножить матрицу 2×3 на матрицу 3×4, но не наоборот. В матрице, в отличие от чисел, порядок умножения важен, даже если операция проводится для двух квадратных матриц, которые можно умножать в любом порядке.

В результате перемножения двух матриц получается третья, с таким же количеством строк, что и в левосторонней матрице, и тем же количеством столбцов, что в правосторонней. Продолжая пример выше, отмечу, что результатом умножения матрицы 2×3 на матрицу 3×4 станет матрица 2×4.

Теперь посмотрим, как выполняется умножение матриц A и B:

$$A = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix};$$

$$B = \begin{pmatrix} g & h & i & j \\ k & l & m & n \\ o & p & q & r \end{pmatrix}.$$

Чтобы было понятнее, мы сгруппируем значения в A и B в векторы: A мы напишем как столбец векторов строк, а B — как ряд векторов столбцов. Например, первая строка в A — это вектор (a, b, c), а второй столбец в B — это вектор (h, l, p):

$$A = \begin{pmatrix} (a, b, c) \\ (d, e, f) \end{pmatrix};$$

$$B = \left(\begin{pmatrix} g \\ k \\ o \end{pmatrix} \begin{pmatrix} h \\ l \\ p \end{pmatrix} \begin{pmatrix} i \\ m \\ q \end{pmatrix} \begin{pmatrix} j \\ n \\ r \end{pmatrix} \right).$$

Проименуем эти векторы:

$$A = \begin{pmatrix} -\vec{A}_0- \\ -\vec{A}_1- \end{pmatrix};\ B = \begin{pmatrix} | & | & | & | \\ \vec{B}_0 & \vec{B}_1 & \vec{B}_2 & \vec{B}_3 \\ | & | & | & | \end{pmatrix}.$$

Нам известно, что A — это 2×3, а B — это 3×4, значит, результатом будет матрица 2×4:

$$\begin{pmatrix} -\vec{A}_0- \\ -\vec{A}_1- \end{pmatrix}\begin{pmatrix} | & | & | & | \\ \vec{B}_0 & \vec{B}_1 & \vec{B}_2 & \vec{B}_3 \\ | & | & | & | \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \end{pmatrix}.$$

Теперь для элементов получившейся матрицы можно использовать простую формулировку: значение элемента в строке r и столбце c результата — то есть c_{rc} — скалярное произведение соответствующего вектора строки в **A** и вектора столбца в **B**, то есть $\vec{A}_r$ и $\vec{B}_c$:

$$\begin{pmatrix} -\vec{A}_0- \\ -\vec{A}_1- \end{pmatrix}\begin{pmatrix} | & | & | & | \\ \vec{B}_0 & \vec{B}_1 & \vec{B}_2 & \vec{B}_3 \\ | & | & | & | \end{pmatrix} = \begin{pmatrix} \langle \vec{A}_0, \vec{B}_0 \rangle & \langle \vec{A}_0, \vec{B}_1 \rangle & \langle \vec{A}_0, \vec{B}_2 \rangle & \langle \vec{A}_0, \vec{B}_3 \rangle \\ \langle \vec{A}_1, \vec{B}_0 \rangle & \langle \vec{A}_1, \vec{B}_1 \rangle & \langle \vec{A}_1, \vec{B}_2 \rangle & \langle \vec{A}_1, \vec{B}_3 \rangle \end{pmatrix}.$$

К примеру, $c_{01} = \langle \vec{A}_0, \vec{B}_1 \rangle$, что раскладывается на $ah + bl + cp$.

Умножение матрицы на вектор

Вектор с размерностью n можно рассматривать как вертикальную матрицу $n \times 1$ или горизонтальную матрицу $1 \times n$ и выполнять умножение так же, как в случае с двумя совместимыми матрицами. Так будет выглядеть умножение матрицы 2×3 и 3D-вектора:

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x + b \cdot y + c \cdot z \\ d \cdot x + e \cdot y + f \cdot z \end{pmatrix}.$$

В результате умножения матрицы на вектор (или вектора на матрицу) получается вектор, и у нас матрицы представляют преобразования, поэтому можно сказать, что матрица *преобразует* вектор.

Гэбриел Гамбетта

Компьютерная графика. Рейтрейсинг и растеризация

Перевел с английского *Д. Брайт*

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Д. Соколов*
Литературный редактор	*Т. Сажина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2022. Наименование: книжная продукция. Срок годности: не ограничен.

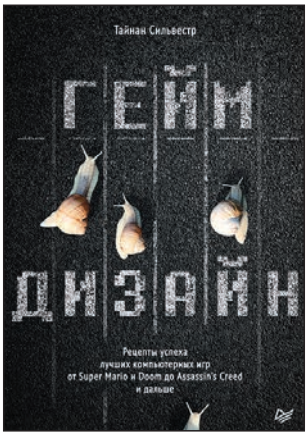
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 31.01.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 18,060. Тираж 1000. Заказ 0000.

Тайнан Сильвестр

ГЕЙМДИЗАЙН. РЕЦЕПТЫ УСПЕХА ЛУЧШИХ КОМПЬЮТЕРНЫХ ИГР ОТ SUPER MARIO И DOOM ДО ASSASSIN'S CREED И ДАЛЬШЕ



Что такое ГЕЙМДИЗАЙН? Это не код, графика или звук. Это не создание персонажей или раскрашивание игрового поля. Геймдизайн — это симулятор мечты, набор правил, благодаря которым игра оживает.

Как создать игру, которую полюбят, от которой не смогут оторваться? Знаменитый геймдизайнер Тайнан Сильвестр на примере кейсов из самых популярных игр рассказывает, как объединить эмоции и впечатления, игровую механику и мотивацию игроков. Познакомьтесь с принципами дизайна, которыми пользуются ведущие студии мира!

Тайнан Сильвестр занимается геймдизайном больше 15 лет. За это время он успел поработать как над инди-проектами, так и над студийным блокбастером BioShock Infinite, но больше всего он известен благодаря RimWorld.