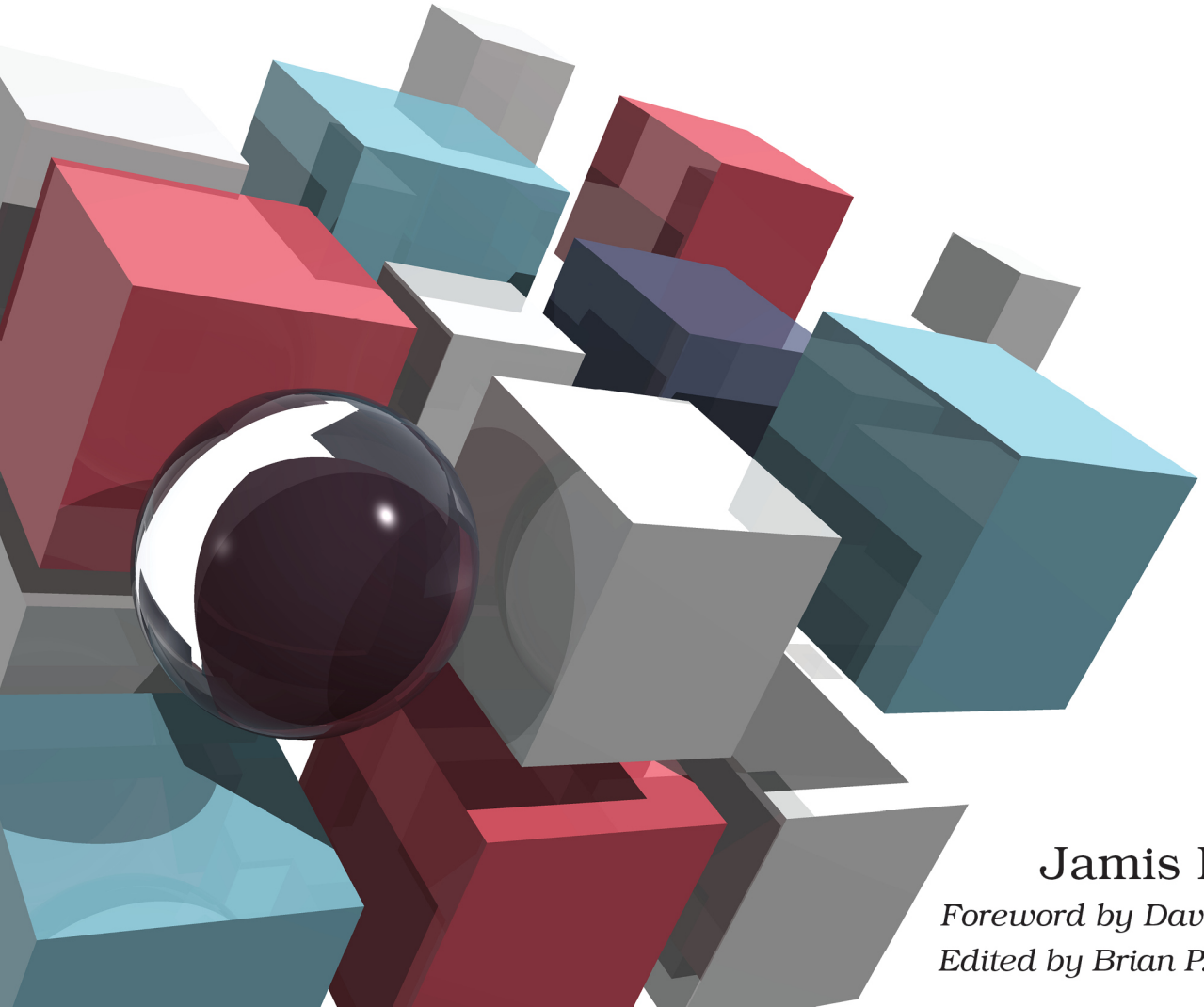
The Pragmatic Programmers

# The Ray Tracer Challenge

## A Test-Driven Guide to Your First 3D Renderer



Jamis Buck

*Foreword by David Buck*

*Edited by Brian P. Hogan*

**Elogio antecipado para O Desafio Ray Tracer**

Seguindo os passos de seu livro, *Labirintos para programadores,* Buck mais uma vez pega um conceito desafiador, apresenta-o em um formato fácil de entender e nos lembra que a programação pode ser divertida e recompensadora.

➤ Dr. Sebastian Raaphorst
Observatório Gemini

Este é um domínio de problema que sempre quis abordar, mas tenho lutado para encontrar algo acessível para alguém que não sabe ou não é bom em C ou C ++. Este livro é uma dádiva de Deus.

➤ Danielle Kefford
Engenheiro de software

Este livro é tortuoso. Aos poucos, um teste aqui e outro ali, você criará uma biblioteca de traçado de raios incrivelmente sofisticada. Como cada mudança é muito pequena, o seu ray tracer irá se esgueirar até você. Essa é a parte tortuosa: no final, você terá construído um software incrivelmente complexo, mas nunca será difícil!

➤ Cory Forsyth
Sócio Fundador, 201 Created, Inc.

No *O Desafio Ray Tracer* Jamis Buck doma um tópico difícil usando uma abordagem prática e divertida que até mesmo os avessos à matemática irão gostar. A abordagem testada desafia e recompensa o leitor com experiências e artefatos que lembram até mesmo o mesquinho mesquinho de software dos momentos alegres no desenvolvimento de software que nos inspiraram a buscar a engenharia em primeiro lugar.

➤ Justin Ball
CTO, Atomic Jolt

Criar um ray tracer é um rito de passagem que recomendo a todos os desenvolvedores que se esforcem para concluir. Jamis faz um ótimo trabalho apresentando tópicos complexos de forma simples e permitindo que o leitor se concentre nas partes mais interessantes do projeto. É quase certo que trabalhar com este livro aumentará suas habilidades de programação.

➤ Jason Pike
Diretor, Engenharia de Software, Atlas RFID Solutions

*O Desafio Ray Tracer* é uma introdução deliciosa à iluminação e renderização 3D por meio de traçado de raios. Sim, há matemática, mas Jamis fornece ótimos exemplos, e os exercícios ilustram conceitos em um estilo que é muito mais divertido do que qualquer aula de matemática que fiz na faculdade!

➤ Matthew Margolis
Diretor, Engenharia de Software

Aceitar o Desafio Ray Tracer foi muito divertido. Começando com alguns testes curtos, você criará imagens lindamente renderizadas com apenas um pouco de matemática e código.

➤ Justin Weiss
Engenheiro de software sênior, Aha!

Com este livro, posso usar o que aprendi na universidade há treze anos, e agora é divertido! *O Desafio Ray Tracer* devolveu minha alegria por projetos de estimação. Eu recomendo a todos!

➤ Gábor László Hajba
Consultor de TI Sênior

Um dos truques para evitar o esgotamento do programador é encontrar um projeto de paixão. Neste livro, você encontrará exatamente isso: um projeto pessoal incrível que você pode realizar independentemente do seu conhecimento de idioma. Jamis's *O Desafio Ray Tracer*
nos mostra que os melhores projetos de paixão são compartilhados.

➤ Kevin Gisi
Engenheiro UX Sênior

Deixamos esta página em branco para que
os números das páginas sejam os mesmos
no eletrônico e
livros de papel.

Tentamos apenas omitir, mas então as
pessoas nos escreveram perguntando sobre
as páginas que faltavam.

De qualquer forma, Eddy, o Gerbil,
queria dizer "olá".

# O Desafio Ray Tracer

Um guia orientado a testes para seu primeiro renderizador 3D

Jamis Buck

A estante pragmática

Raleigh, Carolina do Norte

Muitas das designações usadas por fabricantes e vendedores para distinguir seus produtos são reivindicadas como marcas registradas. Onde essas designações aparecem neste livro e a The Pragmatic Programmers, LLC estava ciente de uma reivindicação de marca registrada, as designações foram impressas em letras maiúsculas iniciais ou em todas as letras maiúsculas. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg e a vinculação *g* device são marcas comerciais da The Pragmatic Programmers, LLC.

Todas as precauções foram tomadas na preparação deste livro. No entanto, o editor não assume nenhuma responsabilidade por erros ou omissões ou por danos que possam resultar do uso das informações (incluindo listas de programas) aqui contidas.

Nossos livros, screencasts e audiolivros pragmáticos podem ajudar você e sua equipe a criar um software melhor e a se divertir mais. Visite-nos em **https://pragprog.com** .

A equipe que produziu este livro inclui:

Editor: Andy Hunt
VP de operações: Janet Furlow Editor
administrativo: Susan Conant Development Editor:
Brian P. Hogan Editor de texto: L. Sakhi MacMillan
Indexação: Potomac Indexing, LLC Layout: Gilson
Graphics

Para vendas, licenciamento por volume e suporte, entre em contato **support@pragprog.com** .

Para direitos internacionais, por favor entre em contato **rights@pragprog.com** .

Copyright © 2019 The Pragmatic Programmers, LLC.

Todos os direitos reservados. Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida, em qualquer forma, ou por animêmeos, eletrônicos, mecânicos, fotocópias, gravações ou outros, sem o consentimento prévio do editor.

ISBN-13: 978-1-68050-271-8

Versão do livro: P1.0 - fevereiro de 2019

# Conteúdo


Agradecimentos . . . . . XI
Prefácio . . . . . xiii
Começando . . . . . xv

1 Tuplas, pontos e vetores . . . . . 1
Tuplas 2
Operações 5
Juntar as peças 12

2 Desenhar em uma tela . . . . . 15
Representando Cores 15
Implementando operações de cores 16
Criação de uma tela 19
Salvando uma tela 19
Juntar as peças 22

3 Matrizes . . . . . 25
Criação de uma matriz 26
Matrizes de multiplicação 28
A Matriz de Identidade 31
Matrizes de Transposição 32
Matrizes Invertidas 33
Juntar as peças 42

4 Transformações matriciais . . . . . 43
Tradução 44
Dimensionamento 46
Rotação 47
Tosquia 51
Transformações de encadeamento 53
Juntar as peças 54

5 Intersecções Ray-Sphere . . . . . 57
Criação de raios 57
Interseção de raios com esferas de 59
rastreamento de interseções 63
Identificação de sucessos 64
Transformando Raios e Esferas Juntando tudo 66
70

6 Luz e Sombreamento . . . . . 75
Normais de superfície 76
Vetores Refletindo 82
O modelo de reflexão Phong reunindo 83
tudo 89

7 Fazendo uma cena . . . . . 91
Construindo um Mundo 92
Definindo uma Transformação de Visualização 97
Implementando uma Câmera 100
Juntar as peças 105

8 Sombras . . . . . 109
Iluminação nas sombras 110
Teste de sombras 111
Renderizando sombras 113
Juntar as peças 116

9 Aviões . . . . . 117
Formas de Refatoração 117
Implementando um Avião 122
Juntar as peças 124

10 Padrões . . . . . 127
Fazendo um padrão listrado, 128
transformando padrões 130
Padrões de generalização 132
Fazendo um padrão de gradiente Fazendo 134
um padrão de anel 135
Fazendo um padrão Checker 3D Juntando 136
tudo 138

11 Reflexão e Refração . . . . . 141
Reflexão 142

Transparência e Refração 149
Efeito Fresnel 160
Juntar as peças 165

12 Cubos . . . . . 167
Cruzando um Raio com um Cubo 168
Encontrando o Normal em um Cubo 173
Juntando-o 175

13 Cilindros . . . . . 177
Cruzando um Raio com um Cilindro Encontrando o 178
Normal em um Cilindro Cilindros Truncados 180
181
Cilindros Tampados 184
Cones 188
Juntar as peças 191

14 Grupos . . . . . 193
Implementando Grupos 194
Encontrando o normal em um objeto filho 197
Usando caixas delimitadoras para otimizar cenas grandes Juntando 200
tudo 203

15 Triângulos . . . . . 207
Triângulos 208
Arquivos Wavefront OBJ 212
Triângulos suaves 218
Triângulos suaves em arquivos OBJ 223
Juntando tudo 225

16 Geometria Sólida Construtiva (CSG) . . . . . 227
Implementando CSG 229
Colorir CSG Shapes 236
Juntar as peças 237

17 Próximos passos . . . . . 239
Luzes de área e luzes de sombras suaves 239
240
Desfocagem focal 241
Desfoque de movimento 242
Anti-aliasing 243
Mapas de textura 244

Perturbação normal 245
Toro primitivo 246
Embrulhando-o 247

A1. Renderizando a imagem da capa . . . . . 249
Índice . . . . . 255

# Agradecimentos

Este livro existe porque meu filho queria aprender a escrever software. Eu havia mexido em ray tracers anos atrás, hackeando POV-Ray e escrevendo meus próprios renderizadores simples, então, quando ele pediu minha ajuda, meu primeiro pensamento foi guiá-lo por escrever um de sua autoria. Desse experimento nasceu uma ideia, que se tornou um esboço, que se tornou uma proposta de livro, que se tornou este livro.

Então, obrigado, Nathaniel!

Muito obrigado à minha esposa, Tessa. Ela não apenas suportou esse projeto, mas me incentivou ativamente nele, mesmo enquanto estava mergulhada até o pescoço em sua própria jornada rumo ao segundo grau de bacharel. Não teria como escrever este livro sem o apoio dela.

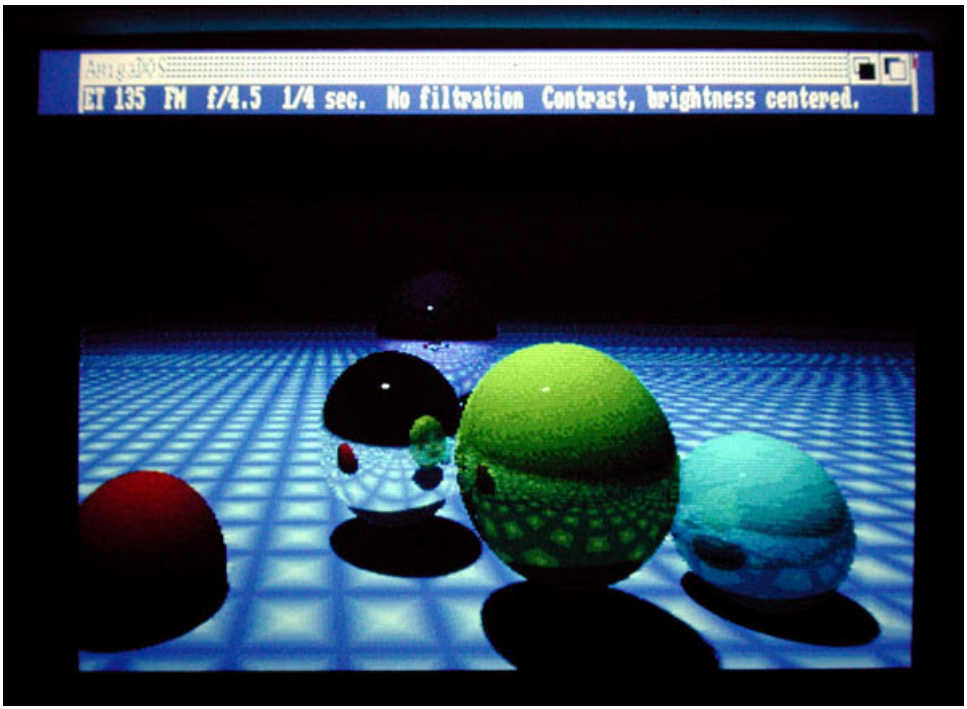
Agradeço também ao pequeno exército de revisores que sacrificaram seu tempo (e talvez sanidade!) Para verificar novamente minha prosa, testes e pseudocódigo. Especificamente: Nathan Anderson, Justin Ball, David Buck (que também gentilmente forneceu o prefácio!), Cory Forsyth, Kevin Gisi, Jeff Holland, Gábor László Hajba, Javan Makhmali, Matthew Margolis, Bowen Masco, David Owen, Jason Pike, Sebastian Raaphorst , Lasse Skindstad Ebert, Bruce Williams e Justin Weiss. Este livro seria infinitamente mais pobre sem seus comentários, correções e sugestões.

Certamente não posso assinar sem agradecer ao meu editor, Brian Hogan, que foi incansável (implacável?) Em me ajudar a transformar minha ideia em algo polido e apresentável. Seu nome merece seu lugar na capa deste livro tanto quanto o meu.

E, por fim, agradeço à minha editora, a Pragmatic Bookshelf, por apostar na minha ideia. Eles realmente são fantásticos para escrever.

# Prefácio

Minhas aventuras no rastreamento de raios começaram em 1986. Eu tinha um computador Amiga e um amigo veio com um disquete contendo o código-fonte C para um rastreamento de raios escrito para Unix. Achei que seria interessante experimentá-lo, então pedi para compilar no meu Amiga e renderizei minha primeira foto. Produziu uma imagem em preto e branco de esferas sobre um plano plano. Fiquei instantaneamente hipnotizado. O pensamento de que um programa de computador fosse capaz de desenhar uma imagem tão realista foi incrível para mim. Adaptei o programa para renderizar imagens coloridas em vez de apenas preto e branco e achei o resultado ainda mais espetacular.




Minha jornada no rastreamento de raios me levou a escrever meu próprio programa de rastreamento de raios chamado DKBTrace, que lancei como freeware. Achei que tinha me divertido escrevendo e queria que outras pessoas se divertissem usando. DKBTrace começou a se tornar

bastante popular no final dos anos 1980, então trabalhei com um grupo de desenvolvedores para transformá-lo em um ray tracer chamado POV-Ray. O POV-Ray é agora o programa freeware de rastreamento de raios mais popular disponível atualmente. Embora eu não tenha feito nenhum desenvolvimento no POV-Ray desde o início dos anos 1990, a equipe do POV-Ray o transformou em um ray tracer de alto nível, capaz de produzir imagens realmente impressionantes.

Agora, Jamis Buck (sem parentesco) escreveu um livro mostrando como escrever seu próprio programa de rastreamento de raios. Suas explicações são claras e divertidas de seguir. Ele o conduz pelo desenvolvimento escrevendo testes primeiro e, em seguida, fazendo com que os testes sejam executados. O livro é agnóstico em relação à linguagem de programação, portanto, você pode escrever o ray tracer em qualquer linguagem de programação que desejar. Na verdade, escrevi todos os exercícios em Smalltalk - minha linguagem de programação favorita. Pude reviver a emoção e a alegria de criar um programa de rastreamento de raios do zero e visualizar imagens que criei com meu software. Este não é um livro que você acabou de ler. É um livro que o orienta a escrever seus próprios programas. Leva você a uma jornada divertida e lhe dá a satisfação de criar suas próprias imagens impressionantes.

Agora convido você a nos acompanhar nessa jornada. Ao longo do caminho, você aprenderá sobre computação gráfica e desenvolvimento de software. Você aprenderá as técnicas básicas usadas para renderizar filmes como *Era do Gelo* e *Carros.* O mais importante, porém, é que você terá a satisfação de escrever seu próprio software que pode surpreendê-lo. Este livro traça o caminho e o conduz. Agora é hora de você dar os primeiros passos. Aproveite a jornada.

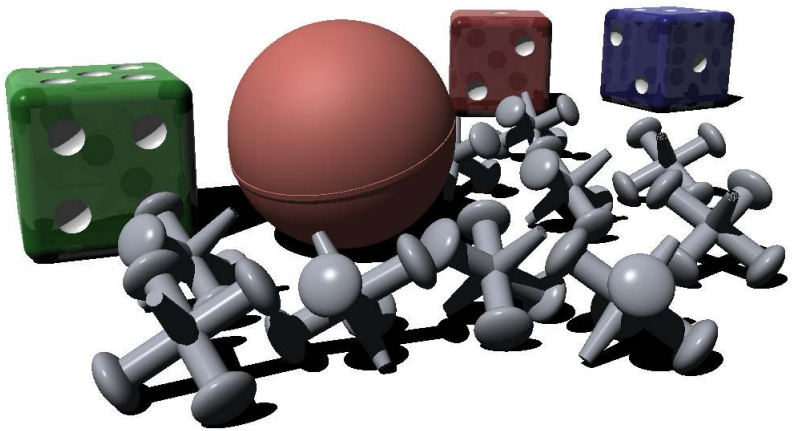
David Buck

Autor de DKBTrace e co-autor do POV-Ray

# Começando

OK. Você é oficialmente incrível. Você é um dos *Essa* programadores, aqueles que buscam ativamente novas maneiras de aplicar sua arte e enfrentar os desafios pela emoção dela. Você está em boa companhia!

Com este livro, você vai construir um renderizador 3D do zero. Especificamente, você construirá um *ray tracer,* lançando raios de luz para trás em uma cena e seguindo seus caminhos enquanto eles saltam em direção a uma fonte de luz. Geralmente não é uma técnica muito rápida (e, portanto, não é adequada para renderização em tempo real), mas pode produzir resultados muito realistas. Ao final deste livro, você será capaz de renderizar cenas como esta:



E você não precisa ser um matemático ou cientista da computação para fazer isso!

Começando na parte inferior, você construirá uma base de rotinas e ferramentas básicas. Você os usará para iniciar outras rotinas, criando raios de luz, formas e funções para prever como eles interagirão. Então, as coisas começam a se mover rapidamente e, em alguns capítulos, você estará produzindo imagens realistas de esferas 3D. Você adicionará sombras e efeitos visuais como padrões geométricos, reflexos de espelho e vidro. Seguem outras formas - planos, cubos, cilindros e muito mais.

No final do livro, você pegará essas formas primitivas e as combinará de maneiras complexas usando operações de conjunto. Não haverá como pará-lo!

O algoritmo específico que você implementará é chamado *Traçado de raio branco,* [1] nomeado em homenagem a Turner Whitted, o pesquisador que o descreveu em 1979. É frequentemente referido como *recursivo* traçado de raio, porque funciona através da desova recursiva *raios*
(linhas representando raios de luz) e saltando-os ao redor da cena para descobrir que cor cada pixel da imagem final deve ter. Em suma, o algoritmo funciona assim para cada um dos pixels da imagem:

1. Lance um raio na cena e descubra onde ele atinge uma superfície.
2. Lance um raio daquele ponto em direção a cada fonte de luz para determinar quais luzes iluminam aquele ponto.
3. Se a superfície for reflexiva, lance um novo raio na direção da reflexão e determine recursivamente qual cor é refletida ali.
4. Se a superfície for transparente, faça a mesma coisa na direção da refração.
5. Combine todas as cores que contribuem para o ponto (a cor da superfície, o reflexo e a refração) e retorne isso como a cor do pixel.

Ao longo deste livro, você implementará cada uma dessas etapas, aprendendo como calcular vetores de reflexão, como aproximar a refração, como interceptar raios com várias formas primitivas e muito mais. Mais cedo do que você imagina, você renderizará cenas 3D incríveis!

## Para quem é este livro

Em última análise, este livro é para quem adora escrever código, mas você obterá o máximo dele se:

- Você tem experiência anterior em escrever software (talvez um ano ou mais).
- Você já escreveu testes de unidade antes.
- Você gosta de mexer e experimentar códigos e algoritmos.

Realmente não importa qual ambiente de programação ou sistema operacional você prefere. O único código neste livro é pseudocódigo. É certo que as explicações *Faz* tendem a linguagens imperativas, procedimentais e orientadas a objetos, mas os próprios conceitos e testes podem ser traduzidos para qualquer ambiente que você desejar.

1 en.wikipedia.org/wiki/Ray_tracing_(graphics)#Recursive_ray_tracing_algorithm

# Como ler este livro

Cada capítulo é apresentado como uma série de testes cobrindo uma pequena parte do ray tracer geral. Uma vez que cada um se baseia nos capítulos anteriores, você terá mais sucesso se os ler em sequência.

Você implementará seu ray tracer no estilo teste-primeiro, escrevendo alguns testes por vez e fazendo-os passar ao implementar as funções e recursos correspondentes no código. A primeira metade do livro está estruturada para levá-lo suavemente de teste para teste, mas conforme você avança para a segunda metade do livro, o ritmo aumenta. Com maior experiência vem maior responsabilidade! Você ainda terá os testes, mas haverá menos mãos e os testes serão apresentados de uma forma mais linear, quase como uma lista de verificação.

Cada capítulo apresenta um ou mais novos recursos, discute como o recurso funciona em alto nível e, em seguida, orienta você pelos testes e como fazê-los passar. Os testes são apresentados como cenários de pepino,[2] mas é absolutamente
*não* necessário usar o pepino para implementá-los. Fique à vontade para usar o sistema de sua preferência para escrever seus testes!

Normalmente, Cucumber é usado para descrever interações de alto nível entre um usuário e um aplicativo, mas os testes neste livro o usam de forma diferente. Aqui, você verá que é usado para descrever as interações de nível inferior, por exemplo, como várias entradas para uma função específica podem afetar a saída da função. Isso permite que o livro o guie na construção de uma API, passo a passo, em vez de apenas mostrar o comportamento de alto nível que você precisa tentar emular. Por exemplo, considere a seguinte especificação hipotética que descreve o comportamento de concatenar duas matrizes.

```
Cenário : Concatenar duas matrizes deve criar uma nova matriz
    Dado uma ← matriz (1, 2, 3)
        E b ← matriz (3, 4, 5)
    Quando c ← a + b
    Então c = matriz (1, 2, 3, 3, 4, 5)
```

É estruturado como qualquer cenário Cucumber, mas descreve interações de API de baixo nível:

- Começa com duas suposições (“ `Dado` … `E` ”), Que deve ser verdadeiro para começar. Eles usam as setas para a esquerda ( ←) para atribuir dois arrays a duas variáveis, `uma` e `b`.

2. Tecnicamente, os testes são escritos em Gherkin, que é a linguagem em que as especificações do Pepino são escritas. Ver pepino.io

- Depois que tudo foi inicializado, uma ação ocorre (“ Quando ”). O resultado dessa ação é o que deve ser testado. Observe que isso também usa a seta para a esquerda, atribuindo o resultado da concatenação uma e b para outra variável, c.

- Finalmente, uma afirmação é feita (“ Então ”), Que deve ser verdade. Isso usa o operador igual (=) para afirmar que a variável c é igual ao array fornecido.

Seu trabalho como leitor é implementar cada teste e, em seguida, fazer cada aprovação. Você pode fazer isso no Cucumber, se quiser - na verdade, os testes do Cucumber podem ser baixados do editor,[3] para poupar o esforço de digitá-los todos manualmente. Mas se o Cucumber não é sua praia, você pode ter o mesmo sucesso traduzindo os cenários do Cucumber em qualquer sistema de teste de sua preferência. Honestamente, parte do quebra-cabeça - parte da diversão! - é traduzir cada especificação em um teste de unidade funcional. O cenário informa qual deve ser o comportamento. *Vocês* decidir como fazer isso acontecer.

Enquanto trabalha neste livro, você descobrirá que uma implementação que funcionou para um teste pode não funcionar bem (ou nem funcionar) para um teste posterior. Você precisará ser flexível e estar disposto a refatorar à medida que descobrir novos requisitos. Isso ou leia o livro inteiro antes de iniciar sua implementação para saber o que está por vir.

Além disso, esteja ciente de que tomei muitas das decisões arquitetônicas neste livro com o objetivo de ser fácil de explicar. Freqüentemente, haverá maneiras mais eficientes de implementar uma função ou de arquitetar um recurso. Você pode discordar do livro às vezes, e tudo bem! Este livro é um roteiro, descrevendo apenas uma das muitas maneiras possíveis de chegar ao objetivo. Siga seu próprio senso estético. Faça seu próprio código.

Por fim, no final de cada capítulo há uma seção chamada “Juntando tudo”. É aqui que você encontrará uma descrição de algo que se baseia no código que você escreveu para aquele capítulo e lhe dá a chance de brincar e experimentar seu novo código. Às vezes, será um pequeno projeto e, outras vezes, uma lista de coisas possíveis para tentar ou direções para explorar. Certamente é possível pular essas seções se você estiver com pressa, mas se o fizer, estará perdendo uma das partes mais agradáveis da viagem.

## Coisas a ter em atenção

Um ray tracer é pesado em matemática. Não há como contornar isso. Ele faz sua mágica ao processar números, encontrar interseções entre linhas e formas, computar reflexos e refrações e mesclar cores. Então, sim, haverá

3 pragprog.com/book/jbtracer/the-ray-tracer-challenge

Deve haver muita matemática aqui, mas irei principalmente dar a você, pronto para implementar. Você encontrará pouco ou nenhum foco sobre a origem da matemática, nenhuma derivação de fórmulas, nenhuma explicação de por que uma equação faz o que faz. Você verá as fórmulas e, quando necessário, explicará como implementá-las, mas não irá percorrer provas e derivações. Se as provas e derivações são o que você particularmente gosta, você sempre pode encontrar uma grande quantidade de informações sobre elas online.

Além disso, a análise de números tende a consumir bastante a CPU. Um ray tracer oferece muitas oportunidades para otimizar o código, mas esse não é o foco deste livro. Se você acompanhar e implementar apenas o que está descrito, seu código provavelmente não será muito eficiente ou muito rápido, mas *vontade* trabalhar. Pense na otimização como um exercício bônus!

Outras coisas a serem observadas, sem nenhuma ordem específica, são estas:

*Comparando números de ponto flutuante*

Especialmente em testes, você precisará comparar dois números de ponto flutuante para determinar se eles são aproximadamente iguais. As especificações no livro representam essa comparação vaga com um sinal de igual simples. Na prática, você precisará ser mais explícito e testar se os dois números estão dentro de um valor de erro ao qual o livro se refere como *EPSILON,* algo assim: | *a* - *b* | <*EPSILON*. Na prática, usando um valor minúsculo como 0,0001 para EPSILON geralmente está bem.

*Comparando estruturas de dados*

Assim como acontece com a comparação de números, também presume-se que você precisará comparar estruturas de dados para ver se eles são iguais. Por exemplo, você precisa ser capaz de ver se dois pontos são iguais. Essas rotinas de comparação não são descritas explicitamente no livro, mas você precisará implementá-las todas da mesma forma. Não faria mal adicionar testes para essas rotinas também, apesar de não terem sido dados no livro.

*Representando o infinito*

Em capítulos posteriores, como Capítulo 12, Cubos, na página 167 , e Capítulo 13, Cilindros, na página 177 , você precisará ser capaz de comparar os números com o infinito. Se sua linguagem de programação pode representar o infinito nativamente, isso é ótimo! Caso contrário, você geralmente pode fingir usando um número muito grande. (Algo como $1 \times 10^{10}$ geralmente é suficiente. Em muitas linguagens de programação, você pode escrever isso como 1e10.)

*Use seus próprios nomes e arquitetura!*

Os nomes das funções e variáveis dados no livro são apenas recomendações. As funções são projetadas para que o primeiro argumento seja o

"Parte responsável" ou a entidade responsável pelo domínio em questão. Em termos orientados a objetos, o primeiro argumento seria o auto
objeto. Mas não deixe que isso o impeça de reatribuir essas responsabilidades, se preferir. Você deve sempre se sentir à vontade para escolher nomes mais apropriados para sua própria arquitetura.

Além disso, o ray tracer será descrito imperativamente, mas você deve procurar maneiras de adaptar essas descrições aos pontos fortes e aos idiomas de seu ambiente de programação. Abrace suas classes, módulos, namespaces, atores e mônadas e faça deste ray tracer o seu!

Muito trabalho foi feito para garantir que tudo neste livro seja preciso e sem erros, mas ninguém é perfeito. Se você encontrar um erro em algum lugar, por favor, me avise. Você pode relatar erratas no site do livro.[4] E não deixe de visitar o fórum de discussão do livro,[5] onde você pode fazer perguntas, compartilhar dicas e truques e publicar colírio para os olhos que renderizou com o seu ray tracer. Este fórum é puramente meu e não é afiliado de forma alguma à Pragmatic Bookshelf.

Com tudo isso fora do caminho, prepare-se - vamos começar imediatamente. Isto vai ser divertido!

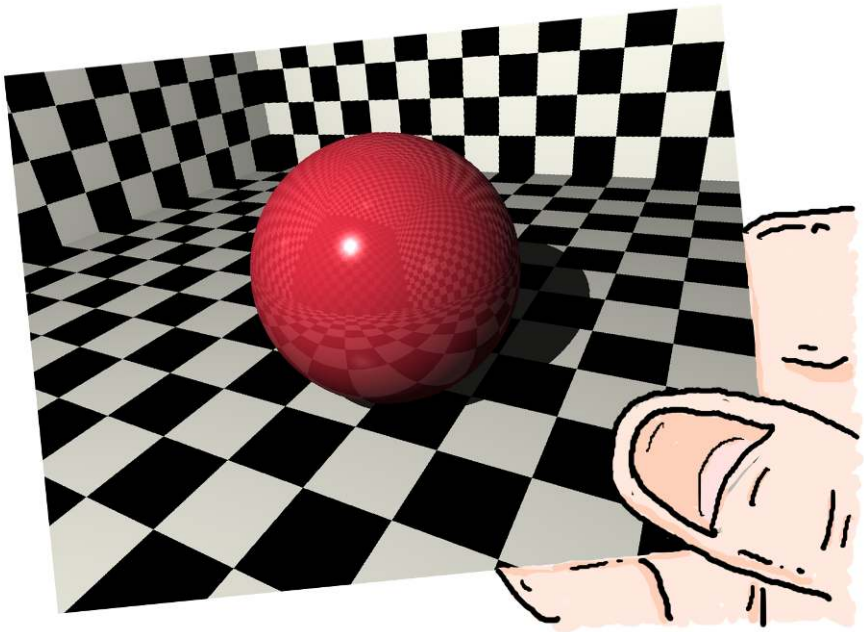
4 pragprog.com/book/jbtracer/the-ray-tracer-challenge

5 forum.raytracerchallenge.com

CAPÍTULO 1

# Tuplas, pontos e vetores

Uma esfera fica sozinha em uma sala com paredes e piso xadrez. Ele reflete a luz de uma lâmpada em algum lugar acima dele. Na verdade, ele reflete quase tudo: a parede quadriculada atrás da câmera, o teto acima e até (se você olhar de perto) sua própria sombra.



Mmmm. Não é lindo? Você não quer apenas tocar aquela esfera? Bem, chegue perto e deixe-me lhe contar um segredinho. Esta pronto?

*Você renderizará cenas como esta com um software que você mesmo escreveu antes de terminar a leitura deste livro.*

É a verdade. Juro. É tudo modelagem de luz e objetos e como eles interagem.

Mas antes de modelar coisas como luz e objetos, você precisa ser capaz de representar conceitos fundamentais como *posição* e *direção* e *distância.*
Por exemplo, essa esfera deve ser localizada *em algum lugar* em uma cena antes que seu renderizador possa desenhá-la. O sombreamento realista depende muito da direção de uma superfície até a fonte de luz, e os reflexos têm a ver com seguir a mudança de direção de um raio de luz.

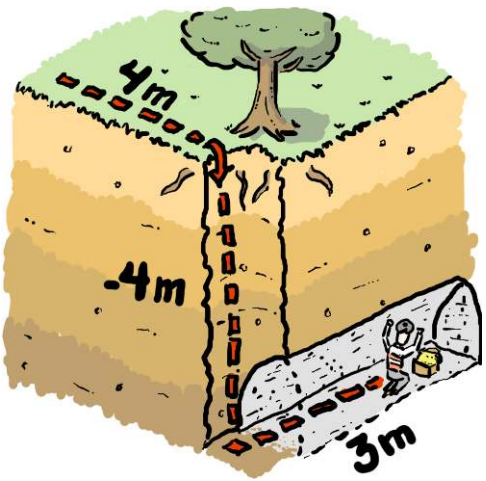
Felizmente, esses conceitos - posição, direção e distância - são perfeitamente encapsulados em uma pequena coisa chamada tupla.

Vamos começar o primeiro capítulo!

## Tuplas

UMA *tupla* é apenas uma lista ordenada de coisas, como números. Isso é muito abstrato, então vamos usar o conceito de posição para ilustrá-lo.

Digamos que você esteja caminhando em um parque um dia. Você avança quatro metros e, de repente, o chão cai debaixo de você. Você desce, pousando quatro metros depois. Lá você descobre um túnel misterioso à esquerda, ao longo do qual você rasteja por mais três metros. Nesse ponto, você descobre um baú cheio de moedas de ouro e comemora. Yay!



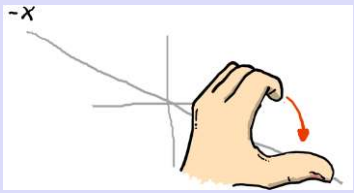

Digamos que seus primeiros quatro metros foram no x direção, o segundo quatro (quando você caiu), no negativo (para baixo) y direção, e os últimos três (no túnel) no z direção. Essas três distâncias, então, podem representar a posição do tesouro, caso em que as escreveríamos como ( 4, -4, 3). Esta

é uma tupla, e esta *específico* tupla também é chamada de *ponto (* porque representa um ponto no espaço).

## Coordenadas canhotas vs. destras

Com o y eixo apontando para cima, e o x eixo apontando para a direita, o z eixo pode ser definido para apontar qualquer *em direção a* você ou *longe de* vocês.

Este livro usa um *canhoto* sistema de coordenadas. Se você pegar o polegar da mão esquerda e apontar no + x direção e, em seguida, aponte os dedos da mão na direção de + y, você descobrirá que, se enrolar os dedos na direção da palma, eles se desviarão de você. Essa é a direção do z eixo para os fins deste livro.




Muitos sites, documentos, artigos, livros e APIs usam um *destro* sistema de coordenadas, no qual o z eixo aponta para você. Não há nada de errado com nenhuma dessas abordagens. Eu escolhi ficar com o sistema para canhotos porque ele é usado em alguns renderizadores populares, incluindo o sistema RenderMan da Pixar, uma a unidade b motor de jogo e o POV-Ray ray tracer de código aberto. c

uma. renderman.pixar.com
b. Unit3d.com
c. www.povray.org

As direções funcionam da mesma maneira. Digamos que você esteja ao lado da arca do tesouro (agora vazia), se orientando. Você toma um momento e mentalmente desenha uma seta apontando da sua posição atual para onde você começou. Esta linha irá apontar *negativo* quatro metros no x direção, *positivo* quatro metros no y direção, e *negativo* três metros no z direção, ou (- 4, 4, -3). Esta tupla - um *vetor* agora - nos diz não apenas a direção em que devemos olhar, mas também o quão longe devemos ir nessa direção. Muito legal!

Mas olhando para ( 4, -4, 3) e (- 4, 4, -3), é impossível saber que um é um ponto e o outro é um vetor. Vamos adicionar um quarto componente a estes ( x, y, z) tuplas, chamadas C, para nos ajudar a diferenciá-los. Definir C a 1 para pontos e 0 para vetores. Assim, seu ponto se torna ( 4, -4, 3, 1), e seu vetor se torna (- 4, 4, -3, 0).

Agora, a escolha de 0 ou 1 para C provavelmente parece arbitrário agora, mas fique quieto! Vai fazer mais sentido quando você chegar a Capítulo 3, Matrizes, na página 25 , onde acaba sendo bastante importante para a multiplicação de matrizes e tuplas.

Tudo isso é muito fundamental, por isso é importante que funcione corretamente. Para isso, você vai escrever alguns testes - de preferência antes de escrever qualquer código real - para ter certeza de que funcionará corretamente.

Evite tipos de dados complexos tanto quanto possível ao implementar suas tuplas. Por exemplo, você deve preferir números de ponto flutuante nativos em vez de abstrações de precisão arbitrária. Essas tuplas serão alguns dos cavalos de batalha do seu ray tracer, então você vai querer que elas sejam enxutas e rápidas!

Use as duas especificações a seguir para orientar seus testes. O primeiro mostra que uma tupla é um ponto quando C é 1, e um segundo mostra que uma tupla é um vetor quando C é 0 Use esses testes para demonstrar como sua implementação acessa os componentes individuais da tupla também.

**features / tuples.feature**

```
Cenário : Uma tupla com w = 1,0 é um ponto
    Dado uma ← tupla (4,3, -4,2, 3,1, 1,0)
    Então ax = 4,3
        E ay = -4,2
        E az = 3,1
        E aw = 1.0
        E a é um ponto
        E a não é um vetor

Cenário : Uma tupla com w = 0 é um vetor
    Dado uma ← tupla (4,3, -4,2, 3,1, 0,0)
    Então ax = 4,3
        E ay = -4,2
        E az = 3,1
        E aw = 0,0
        E a não é um ponto
        E a é um vetor
```

Você usará muito essa distinção, portanto, pode fazer sentido ter algumas funções de fábrica para facilitar a criação desses dois tipos de tuplas. Escreva mais dois testes, um para mostrar que uma função ponto (x, y, z) cria pontos, e outro para mostrar que uma função vetor (x, y, z) cria vetores.

**features / tuples.feature**

```
Cenário : ponto () cria tuplas com w = 1
    Dado p ← ponto (4, -4, 3)
    Então p = tupla (4, -4, 3, 1)

Cenário : vetor () cria tuplas com w = 0
    Dado v ← vetor (4, -4, 3)
    Então v = tupla (4, -4, 3, 0)
```

Legal! Isso fornece uma base sólida para a criação de tuplas, pontos e vetores. A seguir, vamos ver algumas coisas que você pode fazer com eles.

**Comparando Números de Ponto Flutuante**

Cuidado ao comparar valores de ponto flutuante usando equivalência simples. O erro de arredondamento pode fazer dois números que *deve* ser equivalente, em vez de ser ligeiramente diferente.

Quando você precisar testar dois números de ponto flutuante para equivalência, compare suas diferenças. Se o valor absoluto de sua diferença for menor que algum valor (chamado EPSILON), você pode considerá-los iguais. O pseudocódigo para esta comparação é assim:

```
constante EPSILON ← 0,00001

função igual (a, b)
    E se abdômen (a - b) <EPSILON
        Retorna verdade
    outro
        Retorna falso
    fim se
função final
```

## Operações

Agora que você tem essas tuplas, se depara com a questão de como usá-las. Em última análise, esses serão os alicerces do seu ray tracer - eles surgirão em cálculos em todos os lugares, desde o cálculo da interseção de um raio com objetos em sua cena até descobrir como um ponto específico em uma superfície deve ser sombreado. Mas para inserir esses vetores e pontos em seus cálculos, você precisa implementar algumas operações básicas sobre eles.

Vamos começar com algumas operações familiares da aritmética.

Se ainda não o fez, reserve um minuto para escrever uma função que comparará duas tuplas para igualdade. Isso vai te poupar alguma duplicação! Ao fazer isso, tenha em mente os comentários sobre as comparações de ponto flutuante em Comparando Números de Ponto Flutuante, na página 5 .

### Adicionando tuplas

Imagine que você tem razão ( 3, -2,5,1) e um vetor (- 2,3,1,0), e você quer saber onde estaria se seguisse o vetor a partir desse ponto. A resposta vem via *Adição* —Adicionando as duas tuplas. Vá em frente e escreva um teste que demonstre isso, como o seguinte:

**features / tuples.feature**

```
Cenário : Adicionando duas tuplas
    Dado a1 ← tupla (3, -2, 5, 1)
        E a2 ← tupla (-2, 3, 1, 0)
    Então a1 + a2 = tupla (1, 1, 6, 1)
```

Você faz uma nova tupla adicionando os componentes correspondentes de cada um dos operandos - o x soma para produzir o novo x, y para produzir um novo y, e assim por diante.

E veja como isso C coordenar coopera. Você adiciona um ponto ( C de 1) e um vetor ( C de 0), e o resultado tem um C de 1 - outro ponto! Da mesma forma, você pode adicionar dois vetores ( C de 0) e obter um vetor, porque o C A soma é 0. No entanto, adicionar um ponto a um ponto realmente não faz sentido. Tente. Você verá que obtém uma tupla com um C de 2, que não é um vetor nem um ponto!

## Subtraindo tuplas

Subtrair tuplas também é útil. Será útil quando você chegar a Capítulo 6, Luz e Sombreamento, na página 75 , quando você precisa encontrar o vetor que aponta para sua fonte de luz.

Adicione o seguinte teste para mostrar que subtrair tuplas funciona subtraindo os elementos correspondentes das tuplas.

**features / tuples.feature**

```
Cenário : Subtraindo dois pontos
    Dado p1 ← ponto (3, 2, 1)
        E p2 ← ponto (5, 6, 7)
    Então p1 -   p2 = vetor (-2, -4, -6)
```

Não é legal? Os dois C componentes (ambos iguais a 1) se cancelam, e a tupla resultante tem um C de 0 - um vetor! Especificamente, é o vetor
apontando de p 2 topo 1: (- 2, -4, -6).

Da mesma forma, você pode subtrair um vetor ( C de 0) de um ponto ( C de 1) e obtenha outra tupla com um C de 1 - um ponto. Conceitualmente, isso está apenas mudando *para trás* pelo vetor fornecido. Adicione este próximo teste para demonstrar isso.

**features / tuples.feature**

```
Cenário : Subtraindo um vetor de um ponto
    Dado p ← ponto (3, 2, 1)
        E v ← vetor (5, 6, 7)
    Então p - v = ponto (-2, -4, -6)
```

Por último, subtrair dois vetores nos dá uma tupla com um C de 0 - outro vetor, representando a mudança de direção entre os dois. Escreva outro teste para mostrar que isso funciona.

**features / tuples.feature**

```
Cenário : Subtraindo dois vetores
    Dado v1 ← vetor (3, 2, 1)
        E v2 ← vetor (5, 6, 7)
    Então v1 - v2 = vetor (-2, -4, -6)
```

Porém, assim como acontece com a adição, nem todas as combinações fazem sentido. Por exemplo, se você subtrair um ponto ( w = 1) de um vetor ( w = 0), você vai acabar com um negativo
C componente, que não é ponto nem vetor. Vejamos a seguir uma contraparte para a subtração.

## Negando tuplas

Às vezes você vai querer saber o que *oposto* de algum vetor é. Ou seja, dado um vetor que aponta de uma superfície em direção a uma fonte de luz, que vetor aponta da fonte de luz de volta para a superfície? (Você encontrará este caso específico em Capítulo 6, Luz e Sombreamento, na página 75 , também.) Matematicamente, você pode obtê-lo subtraindo o vetor da tupla ( 0, 0, 0, 0). Vá em frente e escreva um teste como o seguinte, para demonstrar isso:

**features / tuples.feature**

```
Cenário : Subtraindo um vetor do vetor zero
    Dado zero ← vetor (0, 0, 0)
        E v ← vetor (1, -2, 3)
    Então zero - v = vetor (-1, 2, -3)
```

Mas ( 0, 0, 0, 0) é difícil de pensar (é um vetor, mas onde está?
*apontando?),* e a operação em si é difícil de escrever. Você pode simplificar isso implementando um *negar* operação, que nega cada componente da tupla. Adicione o seguinte teste mostrando o efeito da negação em uma tupla.

**features / tuples.feature**

```
Cenário : Negando uma tupla
    Dado uma ← tupla (1, -2, 3, -4)
    Então - a = tupla (-1, 2, -3, 4)
```

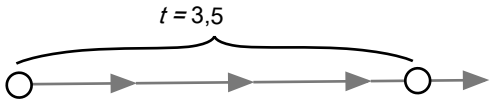
É basicamente assim que funciona: ( x, y, z, w) torna-se (- x, -y, -z, -w).

Se sua linguagem suportar sobrecarga de operador, a negação pode ser implementada como um operador menos unário (- tupla). Caso contrário, um método ( tuple.negate ()) ou uma função ( negar (tupla)) funciona bem. Neste livro, presume-se que - tupla é o operador de negação.

## Multiplicação e divisão escalar

Agora, digamos que você tenha algum vetor e queira saber qual ponto está 3,5 vezes mais longe nessa direção. (Isso vai aparecer em Capítulo 5, Ray-Sphere

Cruzamentos, na página 57 , quando você está descobrindo onde um raio intercepta uma esfera.) Então você posiciona esse vetor ponta a ponta 3,5 vezes para ver a que distância o ponto está do início, como na ilustração a seguir.




Acontece que multiplicar o vetor por 3,5 faz exatamente o que você precisa. O 3.5 aqui está um *valor escalar* porque multiplicando por isso *escalas* o vetor (muda seu comprimento uniformemente). Para fazer isso, você multiplica cada componente da tupla pelo escalar. Escreva esses testes para demonstrar o efeito:

**features / tuples.feature**

```
Cenário : Multiplicando uma tupla por um escalar
    Dado uma ← tupla (1, -2, 3, -4)
    Então a * 3,5 = tupla (3,5, -7, 10,5, -14)

Cenário : Multiplicando uma tupla por uma fração
    Dado uma ← tupla (1, -2, 3, -4)
    Então a * 0,5 = tupla (0,5, -1, 1,5, -2)
```

Observe o último teste, onde você multiplica a tupla por 0,5. Isso é essencialmente a mesma coisa que dividir a tupla por 2, certo? Você sempre pode implementar a divisão com multiplicação, mas às vezes é mais simples descrever uma operação como divisão. Funciona como você espera - dividindo cada componente da tupla pelo escalar. Adicione o seguinte teste para demonstrar isso.

**features / tuples.feature**

```
Cenário : Dividindo uma tupla por um escalar
    Dado uma ← tupla (1, -2, 3, -4)
    Então a / 2 = tupla (0,5, -1, 1,5, -2)
```

Esse é o último dos operadores aritméticos familiares. A seguir, vamos dar uma olhada em alguns novos que serão úteis principalmente com vetores.

## Magnitude

Lembra, no início deste capítulo, quando você leu que um vetor era um valor que codificava direção e distância? A distância representada por um vetor é chamada de *magnitude,* ou *comprimento.* É a distância que você viajaria em linha reta se andasse de uma ponta a outra do vetor. Adicione alguns testes como o seguinte, mostrando a magnitude de vários vetores diferentes.

**features / tuples.feature**

```
Cenário : Calculando a magnitude de vetor (1, 0, 0)
    Dado v ← vetor (1, 0, 0)
    Então magnitude (v) = 1
```

**Cenário :** Calculando a magnitude de vetor (0, 1, 0)

**Dado** v ← vetor (0, 1, 0)

**Então** magnitude (v) = 1

**Cenário :** Calculando a magnitude de vetor (0, 0, 1)

**Dado** v ← vetor (0, 0, 1)

**Então** magnitude (v) = 1

**Cenário :** Calculando a magnitude de vetor (1, 2, 3)

**Dado** v ← vetor (1, 2, 3)

**Então** magnitude (v) = √ 14

**Cenário :** Calculando a magnitude de vetor (-1, -2, -3)

**Dado** v ← vetor (-1, -2, -3)

**Então** magnitude (v) = √ 14

O teorema de Pitágoras nos ensinou como calcular isso, com alguns quadrados e uma raiz quadrada:

$$\text{magnitude}(v) = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Vetores com magnitudes de 1 são chamados de *vetores unitários,* e eles serão muito úteis. Você vai usá-los ao computar sua matriz de visão em Definindo uma Transformação de Visualização, na página 97 , ao determinar a direção perpendicular a uma superfície ( Calculando o Normal em uma Esfera, na página 77 ), e mesmo ao gerar os raios que você deseja lançar em sua cena ( Capítulo 5, Intersecções Ray-Sphere, na página 57 )

Porém, você nem sempre será capaz de começar com um vetor unitário bom e organizado. Muitas vezes, na verdade, você estará começando com uma diferença entre dois pontos e precisará ser capaz de pegar isso e transformá-lo em um vetor unitário, preservando sua direção.

*Normalização* para o resgate!

## Normalização

Normalização é o processo de pegar um vetor arbitrário e convertê-lo em um vetor unitário. Ele manterá seus cálculos ancorados em uma escala comum (o vetor unitário), o que é muito importante. Se você pular a normalização de seus vetores de raios ou normais de superfície, seus cálculos serão escalados de forma diferente para cada raio que você lançar, e suas cenas parecerão terríveis (se forem renderizadas).

Adicione os seguintes testes ao seu conjunto, mostrando o efeito da normalização de alguns vetores diferentes e também confirmando que o comprimento de um vetor normalizado é 1.

```
features / tuples.feature
Cenário : Normalizando vetor (4, 0, 0) dá (1, 0, 0)
    Dado v ← vetor (4, 0, 0)
    Então normalizar (v) = vetor (1, 0, 0)

Cenário : Normalizando vetor (1, 2, 3)
    Dado v ← vetor (1, 2, 3)
                                                    # vetor (1 / √14,          2 / √14,        3 / √14)
    Então normalizar (v) = aproximadamente vetor (0,26726, 0,53452, 0,80178)

Cenário : A magnitude de um vetor normalizado
    Dado v ← vetor (1, 2, 3)
    Quando norma ← normalizar (v)
    Então magnitude (norma) = 1
```

Você normaliza uma tupla dividindo cada um de seus componentes por sua magnitude. Em pseudocódigo, é mais ou menos assim:

```
função normalizar (v)
    Retorna tupla (vx / magnitude (v),
                   vy / magnitude (v),
                   vz / magnitude (v),
                   vw / magnitude (v))
função final
```

Com isso, você pode transformar qualquer vetor (ou melhor, qualquer vetor com magnitude diferente de zero) em um vetor unitário.

## Produto interno

Ao lidar com vetores, um *produto escalar (* também chamado de *produto escalar,* ou *produto Interno)* vai aparecer quando você começar a cruzar raios com objetos, bem como quando você calcular o sombreamento em uma superfície. O produto escalar pega dois vetores e retorna um valor escalar. Adicione este teste para demonstrar o efeito do produto escalar.

```
features / tuples.feature
Cenário : O produto escalar de duas tuplas
    Dado uma ← vetor (1, 2, 3)
        E b ← vetor (2, 3, 4)
    Então ponto (a, b) = 20
```

Dados esses dois vetores, o produto escalar é calculado como a soma dos produtos dos componentes correspondentes de cada vetor. Aqui está o pseudocódigo mostrando como é:

```
função ponto (a, b)
    Retorna ax * bx +
            ay * por +
            az * bz +
            aw * bw
função final
```

O produto escalar pode parecer muito abstrato, mas aqui está uma maneira rápida de internalizá-lo: o *menor* o produto escalar, o *maior* o ângulo entre os vetores. Por exemplo, dados dois vetores unitários, um produto escalar de 1 significa que os vetores são idênticos e um produto escalar de -1 significa que eles apontam em direções opostas. Mais especificamente, e novamente se os dois vetores são vetores unitários, o produto escalar é realmente o *cosseno* do ângulo entre eles, fato que será útil quando você chegar a Capítulo 6, Luz e Sombreamento, na página 75 . Se você quiser ler mais sobre o que significa produto escalar e como entendê-lo, eu

recomende o seguinte artigo: betterexplained.com/articles/vector-calculus-understandingthe-dot-product .

Você certamente não precisa de nenhum conhecimento profundo do produto escalar para implementá-lo. (Sorte sua!) Por enquanto, apenas faça o teste passar e siga em frente.

**Joe pergunta:**

## O produto escalar precisa do C componente?

Se você já foi exposto a produtos escalares antes, pode se perguntar se C pertence a este cálculo, uma vez que o produto escalar só faz sentido em vetores, e todos os nossos vetores terão um C de 0

A resposta é: depende. Optei por incluí-lo aqui porque o produto escalar se aplica a vetores de qualquer dimensão, não apenas três, e porque preserva uma certa simetria com as outras operações. Além disso, se acontecer de você usar o produto escalar em pontos em vez de vetores acidentalmente, mantendo o C na computação pode ajudá-lo a identificar o bug mais cedo ou mais tarde!

### Produto cruzado

Ok, último. O *produto cruzado* é outra operação vetorial, mas ao contrário do produto escalar, ela retorna outro vetor em vez de um escalar, o que o teste a seguir demonstra. Vá em frente e adicione-o à sua suíte.

**features / tuples.feature**

```
Cenário : O produto vetorial de dois vetores
    Dado uma ← vetor (1, 2, 3)
        E b ← vetor (2, 3, 4)
    Então cruzar (a, b) = vetor (-1, 2, -1)
        E cruzar (b, a) = vetor (1, -2, 1)
```

Observe que isso é especificamente um teste *vetores,* não *tuplas.* Isso ocorre porque o produto vetorial quadridimensional é significativamente mais complicado do que o

produto cruzado tridimensional, e seu ray tracer realmente só precisa da versão tridimensional de qualquer maneira.

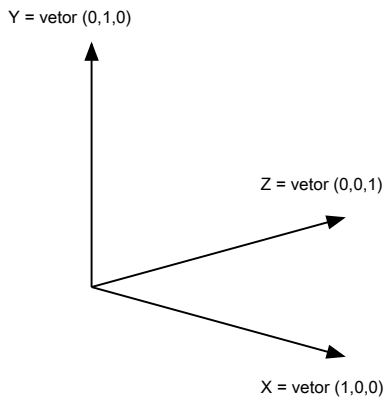
Além disso, observe que se você alterar a ordem dos operandos, alterará a direção do vetor resultante. Lembre-se disso ao usar o produto vetorial: o pedido é importante!

No pseudocódigo, o produto vetorial de dois vetores tridimensionais se junta assim:

```
função cruzar (a, b)
    Retorna vetor (ay * bz - az * por,
                   az * bx - ax * bz,
                   machado * por - ay * bx)
função final
```

Você obtém um novo vetor que é perpendicular a ambos os vetores originais.

O que isto significa? Considere os três vetores perpendiculares a seguir, X, Y, e Z.




Se você pegar o produto cruzado de X e Y, você pega Z. Similarmente, Y cruzar Z pega você X, e Z cruzar X é Y. Os resultados são sempre perpendiculares às entradas.

Novamente, a ordem é importante aqui. X cruzar Y da-te Z, mas Y cruzar X da-te - Z!

Você usará isso principalmente ao trabalhar com transformações de visualização (em Capítulo 7, Fazendo uma cena, na página 91 ), mas também aparecerá quando você começar a renderizar triângulos (em Capítulo 15, Triângulos, na página 207 )

## Juntar as peças

Quanto aos primeiros passos, este não foi tão ruim e lançou as bases para algumas coisas ótimas. Você tem uma implementação funcional de pontos e

vetores! Essas coisas vão aparecer em todos os lugares. Infelizmente, você ainda não tem nenhum ray tracer para conectar seu código, mas ainda pode se divertir um pouco com ele.

Experimente brincar com este pequeno programa, disparando projéteis virtuais e ver até onde eles vão. Isso permitirá que você exercite as rotinas de vetor e ponto que escreveu. Comece com as duas estruturas de dados a seguir:

- UMA *projétil* tem um *posição (* um ponto) e um *velocidade* um vetor).
- A *ambiente* tem *gravidade (* um vetor) e *vento (* um vetor).

Em seguida, adicione um tick (ambiente, projétil) função que retorna um novo projétil, representando o projétil dado após uma unidade de tempo ter passado. (As unidades reais aqui não importam - talvez sejam segundos ou milissegundos. Tanto faz. Vamos chamá-las apenas de “tiques”.)

Em pseudocódigo, o marcação() função deve fazer o seguinte:

```
função marcação (env, proj)
    posição ← proj.position + proj.velocity velocity ← proj.velocity + env.gravity + env.wind

    Retorna projétil (posição, velocidade)
função final
```

Agora, inicialize um projétil e um ambiente. Use os valores que desejar, mas estes podem ajudá-lo a começar:

```
# projétil inicia uma unidade acima da origem.
# velocidade é normalizada para 1 unidade / escala.
p ← projétil ( ponto (0, 1, 0), normalizar ( vetor (1, 1, 0)))

# gravidade -0,1 unidade / tique, e o vento é -0,01 unidade / tique.
e ← ambiente ( vetor (0, -0,1, 0), vetor (-0,01, 0, 0))
```

Em seguida, execute tick repetidamente até que o projétil y position é menor ou igual a 0. Relate a posição do projétil após cada tick, e o número de ticks necessários para que o projétil atinja o solo. Experimente multiplicar a velocidade inicial do projétil por números cada vez maiores para ver quanto mais longe o projétil vai!

Assim que tiver a chance de brincar um pouco com este canhão virtual, vá para o próximo capítulo. Você vai implementar o lado visual do seu ray tracer, o *tela de pintura* para o qual tudo acabará sendo atraído.

CAPÍTULO 2

# Desenhar em uma tela

Pontos e vetores podem ser fundamentais para um ray tracer, mas sem uma maneira de transformá-los em algo visual, a maioria das pessoas não ligará. É uma boa coisa que você está prestes a implementar um *tela de pintura,* então, não é?

Uma tela é uma espécie de prancheta de desenho virtual, que seu ray tracer usará para transformar suas cenas em imagens que você pode realmente *Vejo.* Neste capítulo, você criará uma tela que suporta milhões de cores e que poderá posteriormente salvar como um arquivo de imagem.

Para chegar lá, falaremos sobre cores e como representá-las, bem como algumas operações de cores que você precisará oferecer suporte. Assim que tiver controle sobre isso, você passará para a tela em si e concluirá com um pequeno projeto para revisualizar seu lançador de projéteis do capítulo anterior.

Vamos começar.

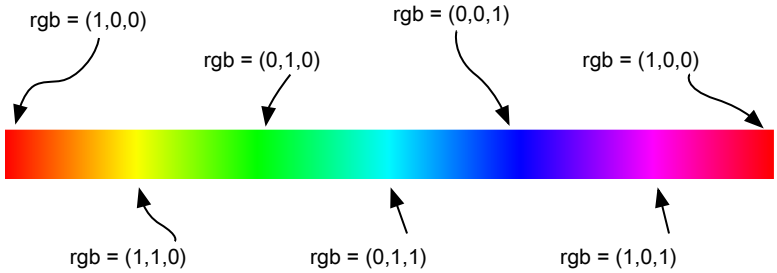
## Representando Cores

Cada pixel no monitor do computador é uma composição de três cores: vermelho, verde e azul. Se você pegar essas três cores e misturá-las em quantidades diferentes, obterá quase todas as outras cores que puder imaginar, desde vermelho, amarelo e verde, até ciano, azul e roxo, e tudo mais.

Se você permitir que vermelho, verde e azul tenham um valor entre 0 e 1 (com 0 significando que a cor está totalmente ausente e 1 significando que a cor está totalmente presente), então o figura na página 16 mostra algumas cores possíveis que você pode obter combinando-as.

Se todos os componentes forem 1, você obtém o branco. Se todos os componentes forem 0, você ficará preto.

E você percebeu isso? Uma cor é uma tupla, assim como vetores e pontos! Na verdade, quando chega a hora de tornar isso real, pode fazer sentido construir seu




implementação de cor no topo de sua implementação de tupla, em vez de começar do zero.

De uma forma ou de outra, você precisará ser capaz de criar uma cor a partir de um (vermelho, verde, azul) tupla. Adicione o seguinte teste, que faz exatamente isso.

**features / tuples.feature**

```
Cenário : As cores são tuplas (vermelho, verde, azul)
    Dado c ← cor (-0,5, 0,4, 1,7)
    Então c.red = -0,5
        E c.green = 0,4
        E c.azul = 1,7
```

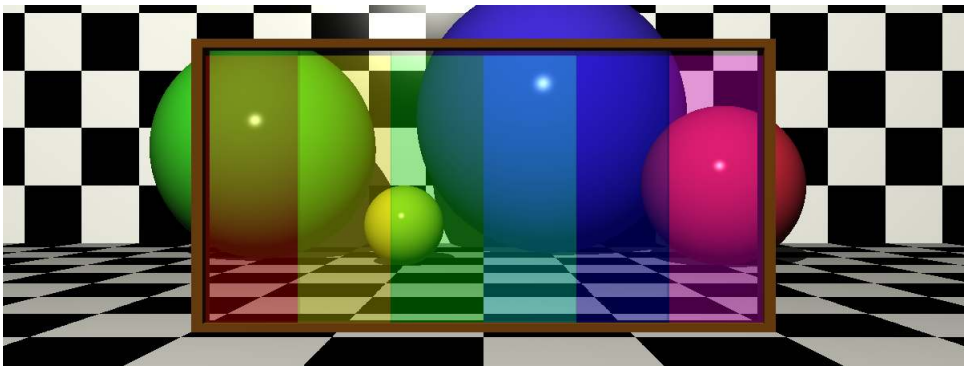
Na prática, você só usará números entre 0 e 1 para esses componentes, mas não coloque nenhuma restrição sobre eles ainda. Se uma cor for especialmente clara ou escura em algum lugar de sua cena, ela pode passar por múltiplas transformações antes de atingir seu “olho” virtual, caindo para menos de 0 ou aumentando para maior que 1 em qualquer ponto ao longo do caminho. Limitar a cor prematuramente pode tornar partes da cena muito claras ou escuras na imagem final.

Assim que o teste for aprovado, siga em frente. Falaremos sobre as diferentes operações que sua implementação de cores precisará oferecer suporte.

## Implementando operações de cores

As cores, como você verá, tendem a interagir umas com as outras. Quer seja uma luz verde refletindo em uma superfície amarela, ou uma superfície azul vista através de um vidro vermelho, ou alguma outra combinação de transparência e reflexo, as cores podem afetar umas às outras. Por exemplo, o figura na página 17 mostra como diferentes painéis de vidro coloridos afetam as cores vistas através deles.

Felizmente, podemos lidar com todas essas combinações com apenas quatro operações: adicionar e subtrair cores, multiplicar uma cor por um escalar e multiplicar uma cor por outra cor.



É aqui que as cores mostram especialmente sua relação com vetores e pontos. Adição, subtração e multiplicação por um escalar funcionam exatamente como você viu com tuplas em Capítulo 1, Tuplas, Pontos e Vetores, na página 1 . Escreva os seguintes testes para enfatizar isso, mostrando que você espera o mesmo comportamento com as cores.

**features / tuples.feature**

```
Cenário : Adicionando cores
    Dado c1 ← cor (0,9, 0,6, 0,75)
        E c2 ← cor (0,7, 0,1, 0,25)
      Então c1 + c2 = cor (1,6, 0,7, 1,0)

Cenário : Subtraindo cores
    Dado c1 ← cor (0,9, 0,6, 0,75)
        E c2 ← cor (0,7, 0,1, 0,25)
      Então c1 - c2 = cor (0,2, 0,5, 0,5)

Cenário : Multiplicando uma cor por um escalar
    Dado c ← cor (0,2, 0,3, 0,4)
    Então c * 2 = cor (0,4, 0,6, 0,8)
```

A operação de cor final, multiplicando uma cor por outra cor, é usada para misturar duas cores. Você o usará quando (por exemplo) quiser saber a cor visível de uma superfície verde-amarela iluminada por uma luz roxo-avermelhada. Implemente o seguinte teste para mostrar o que você espera que aconteça.

**features / tuples.feature**

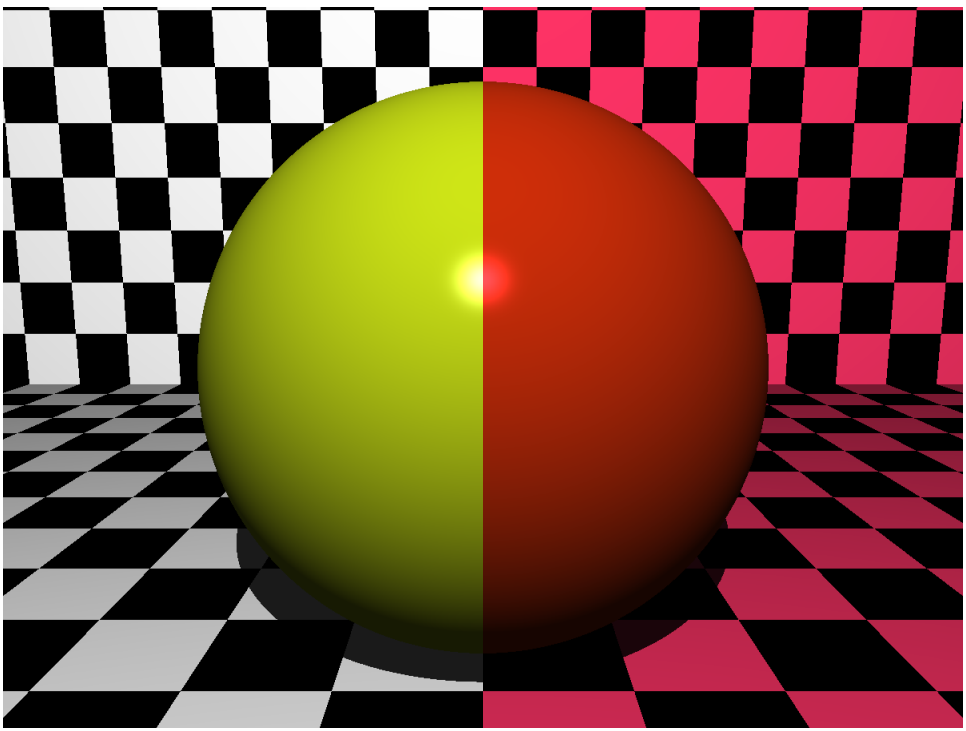
```
Cenário : Multiplicando cores
    Dado c1 ← cor (1, 0,2, 0,4)
        E c2 ← cor (0,9, 1, 0,1)
      Então c1 * c2 = cor (0,9, 0,2, 0,04)
```

Este método de mesclar duas cores funciona multiplicando os componentes correspondentes de cada cor para formar uma nova cor. É tecnicamente chamado de *Produto Hadamard (* ou *Produto Schur),* mas realmente não importa o que você

chame-o. Ele só precisa produzir uma nova cor onde o novo componente vermelho é o produto dos componentes vermelhos das outras cores, e assim por diante para o azul e o verde. Em pseudocódigo, é assim:

```
função hadamard_product (c1, c2)
    r ← c1.red * c2.red g ← c1.green * c2.green b
    ← c1.blue * c2.blue

    Retorna cor (r, g, b)
função final
```

Considere este teste novamente. Diz que se você visse aquela superfície verde-amarela ( c2) sob uma luz roxo-avermelhada ( c1), a cor resultante parecerá vermelha (porque seu componente vermelho, 0,9, é o maior). A imagem a seguir compara aquela esfera verde-amarela em luz branca com a luz roxa-avermelhada e mostra visualmente o que o teste está afirmando.



Por enquanto, isso é tudo que você precisa fazer com as cores. Assim que esses testes forem aprovados, você estará pronto para a próxima etapa: uma tela de imagem adequada!

## Criação de uma tela

UMA *tela de pintura* é apenas uma grade retangular de pixels - muito parecida com a tela do computador. Sua implementação permitirá que seu tamanho seja configurável, para que você possa especificar a largura e a altura que a tela deve ter.

Adicione o seguinte teste à sua suíte. Ele demonstra como uma tela é criada e mostra que cada pixel na tela deve ser inicializado em preto ( cor (0, 0, 0)).

```
features / canvas.feature
Cenário : Criação de uma tela
    Dado c ← tela de pintura (10, 20)
    Então c.width = 10
        E c.altura = 20
        E cada pixel de c é cor (0, 0, 0)
```

Os pixels são atraídos para a tela especificando uma posição e uma cor. Escreva o seguinte teste, apresentando uma função chamada write_pixel (canvas, x, y, color) e mostrando como ele é usado.

```
features / canvas.feature
Cenário : Gravando pixels em uma tela
    Dado c ← tela de pintura (10, 20)
        E vermelho ← cor (1, 0, 0)
    Quando write_pixel (c, 2, 3, vermelho)
    Então pixel_at (c, 2, 3) = vermelho
```

Observe que o x e y os parâmetros são assumidos como baseados em 0 neste livro. Quer dizer, x pode estar em qualquer lugar de 0 a largura - 1 ( inclusive), e y pode estar em qualquer lugar de 0 a altura - 1 ( inclusive).

Você não precisará de nenhum outro método para gravar em sua tela, já que seu ray tracer funcionará pixel a pixel em toda a cena. Faça todos os testes passarem e então podemos falar sobre como salvar essa tela no disco em um formato que seja realmente significativo.

## Salvando uma tela

A tela, por si só, é apenas uma etapa intermediária. Pode representar uma imagem de sua cena, mas você não pode olhá-la diretamente. Você não pode mostrar a ninguém. Você não pode usá-lo para se gabar de como sua cena renderizada em 3D parece incrível ou de como seu ray tracer é incrível. Para fazer isso, você precisa ser capaz de pegar as informações em sua tela e gravá-las em um arquivo, que pode então ser visto, enviado por e-mail, tweetado, Instagrammed ou qualquer outra coisa.

Vamos fazer isso acontecer.

Você pode escolher entre um *muito* de diferentes formatos de imagem, mas você só vai implementar um deles: o formato Portable Pixmap (PPM) do projeto Netpbm. [1] Existem vários tipos de formato PPM, mas a versão que você implementará (chamada PPM "simples") é um texto simples.

**Joe pergunta:**

**Como posso ver um arquivo PPM?**

Se você usa um Mac, está com sorte, porque Preview.app (que faz parte do sistema operacional) pode abrir arquivos PPM. No localizador, basta clicar duas vezes no arquivo PPM que deseja visualizar ou digitar abra minha-imagem.ppm a partir da linha de comando.

A história é mais complicada para Linux e Windows, mas não terrivelmente. Existem muitas ferramentas que você pode obter para qualquer plataforma que abrirá arquivos PPM, mas você realmente não pode dar errado com o GNU Image Manipulation Program (GIMP). [uma] É gratuito, é multiplataforma, é de código aberto e é bem mantido.

uma. www.gimp.org

Todo arquivo PPM simples começa com um cabeçalho que consiste em três linhas de texto. A figura a seguir mostra um cabeçalho possível.

```
P3
80 40
255
```

A primeira linha é a string P3 (que é o identificador, ou "número mágico", para o tipo de PPM que estamos usando), seguida por uma nova linha. A segunda linha consiste em dois números que descrevem a largura e altura da imagem em pixels. O cabeçalho da figura anterior descreve uma imagem com 80 pixels de largura e 40 de altura. A terceira linha (255) especifica o *valor máximo da cor,* o que significa que cada valor de vermelho, verde e azul será dimensionado para ficar entre 0 e 255, inclusive.

Escreva o seguinte teste. Ele apresenta uma função chamada canvas_to_ppm (canvas)
que retorna uma string formatada por PPM. Este teste ajudará a garantir que o cabeçalho seja criado corretamente.

**features / canvas.feature**

```
Cenário : Construindo o cabeçalho PPM
    Dado c ← tela de pintura (5, 3)
    Quando ppm ← canvas_to_ppm (c)
```

1 netpbm.sourceforge.net

```
Então as linhas 1-3 de ppm são
    """
    P3
    5 3
    255
    """
```

Imediatamente após este cabeçalho estão os dados de pixel, que contém cada pixel representado como três inteiros: vermelho, verde e azul. Cada componente deve ser escalado entre 0 e o valor máximo de cor fornecido no cabeçalho (por exemplo, 255), e cada valor deve ser separado de seus vizinhos por um espaço.

Adicione o seguinte teste à sua suíte para mostrar que os dados de pixel PPM são construídos corretamente para uma tela onde três pixels foram coloridos. Observe que os componentes de cor que seriam maiores que 255 são limitados (ou *preso)*
a 255, e os componentes que seriam menores que 0 são fixados em 0.

**features / canvas.feature**

```
Cenário : Construindo os dados de pixel PPM
    Dado c ← tela de pintura (5, 3)
        E c1 ← cor (1,5, 0, 0)
        E c2 ← cor (0, 0,5, 0)
        E c3 ← cor (-0,5, 0, 1)
    Quando write_pixel (c, 0, 0, c1)
        E write_pixel (c, 2, 1, c2)
        E write_pixel (c, 4, 2, c3)
        E ppm ← canvas_to_ppm (c)
    Então linhas 4-6 de ppm são
        """
        255 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 128 0 0 0 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 """
```

Observe como a primeira linha de pixels vem primeiro, depois a segunda linha e assim por diante. Além disso, cada linha é encerrada por uma nova linha.

Além disso, nenhuma linha em um arquivo PPM deve ter mais de 70 caracteres. A maioria dos programas de imagem tende a aceitar imagens PPM com linhas mais longas do que isso, mas é uma boa ideia adicionar novas linhas conforme necessário para mantê-las mais curtas. (Apenas tome cuidado para colocar a nova linha onde um espaço teria sido colocado, para não dividir um número pela metade!)

Implemente o seguinte teste para garantir que as linhas de dados de pixel não excedam 70 caracteres.

**features / canvas.feature**

**Cenário :** Divisão de longas linhas em arquivos PPM

**Dado** c ← tela de pintura (10, 2)

**Quando** cada pixel de c é definido como cor (1, 0,8, 0,6)

**E** ppm ← canvas_to_ppm (c)

**Então** linhas 4-7 de ppm são

```
""" "
255 204 153 255 204 153 255 255 204 153 255 204 153 255 204 153 255 255 204 153 255 204 153 255 204 153 255 204 153
255 204 153
255 204 153 255 204 153 255 255 204 153 255 204 153 255 204 153 255 255 204 153 255 204 153 255 204 153 255 204 153
255 204 153
""" "
```

Mais uma coisa. Alguns programas de imagem (notavelmente ImageMagick [2]) não processará arquivos PPM corretamente, a menos que os arquivos sejam encerrados por um caractere de nova linha. Adicione o seguinte teste para satisfazer esses consumidores exigentes.

**features / canvas.feature**

**Cenário :** Os arquivos PPM são encerrados por um caractere de nova linha

**Dado** c ← tela de pintura (5, 3)

**Quando** ppm ← canvas_to_ppm (c)

**Então** ppm termina com um caractere de nova linha

Isso é realmente tudo que há para arquivos PPM. O próximo passo é embrulhar tudo com um laço e fazer algo divertido com ele! Vamos revisitar o programa que você escreveu no capítulo anterior.

## Juntar as peças

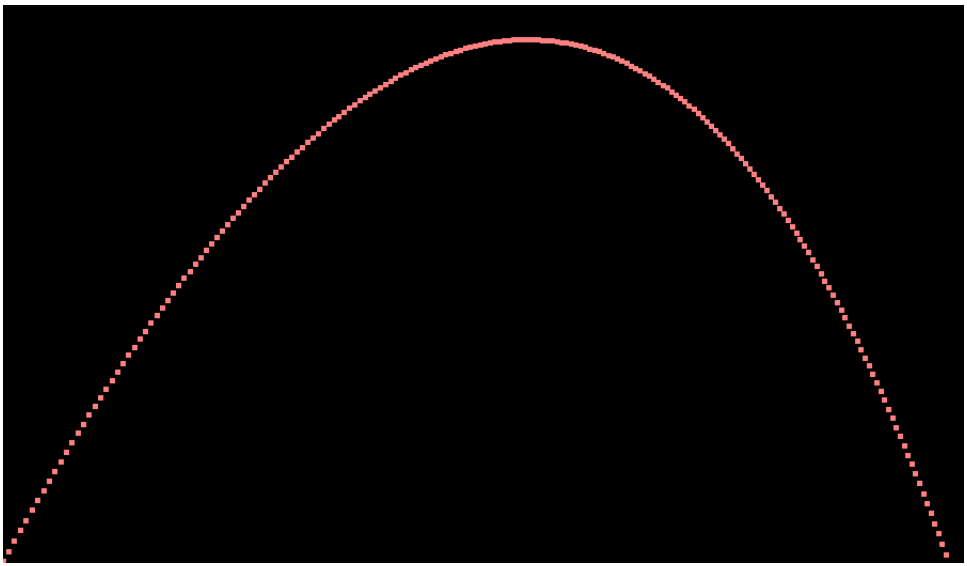
No capítulo anterior, você escreveu um programa para calcular a trajetória de um projétil, usando nada além de pontos e vetores. Mas agora você tem uma nova ferramenta!

Para este desafio, você calculará mais uma vez a trajetória de um projétil, como antes, mas desta vez traçará seu curso em sua tela totalmente nova. Após cada tique, pegue as coordenadas do projétil e pinte o pixel correspondente na tela. Quando o loop terminar, salve sua tela no disco e veja o resultado. Deve ser parecido com o figura na página 23 . Os tamanhos dos pixels foram exagerados aqui, plotados como quadrados em vez de pontos únicos, para torná-los visíveis na impressão.

Ao enfrentar este desafio, observe algumas coisas:

1. A tela y a coordenada está de cabeça para baixo em comparação com a sua coordenada mundial nates. É zero na parte superior da tela e aumenta à medida que você desce.

2 www.imagemagick.org



Para converter as coordenadas do seu projétil em coordenadas da tela, subtraia as do projétil y da altura da tela.

2 Vai ser muito, muito fácil acidentalmente traçar um ponto que está fora dos limites de sua tela. Certifique-se de lidar com este caso, fazendo com que a tela ignore os pontos fora de seus limites ou evitando que seu programa plote esses pontos em primeiro lugar.

3 Suas coordenadas de projétil serão números de ponto flutuante. Os pixels em sua tela, no entanto, estão em coordenadas inteiras. Certifique-se de converter seu projétil x e y coordenadas para inteiros antes de traçá-los.

4 Depois que seu loop terminar, certifique-se de salvar sua tela em um arquivo! Afinal, esse é o objetivo deste exercício.

Você pode precisar experimentar um pouco para encontrar o tamanho da tela e a velocidade do projétil que se complementam. Inicialmente, você provavelmente descobrirá que o seu projétil mal faz um pontinho na tela ou vai se espalhar pela lateral na velocidade da luz! Se ajudar, a imagem acima foi feita com as seguintes configurações:

começar ← ponto (0, 1, 0) velocidade ← normalizar ( vetor (1, 1,8, 0)) * 11,25 p ← projétil (início, velocidade)

gravidade ← vetor (0, -0,1, 0) vento ← vetor (-0,01, 0, 0) e ← ambiente (gravidade, vento)

c ← tela de pintura (900, 550)

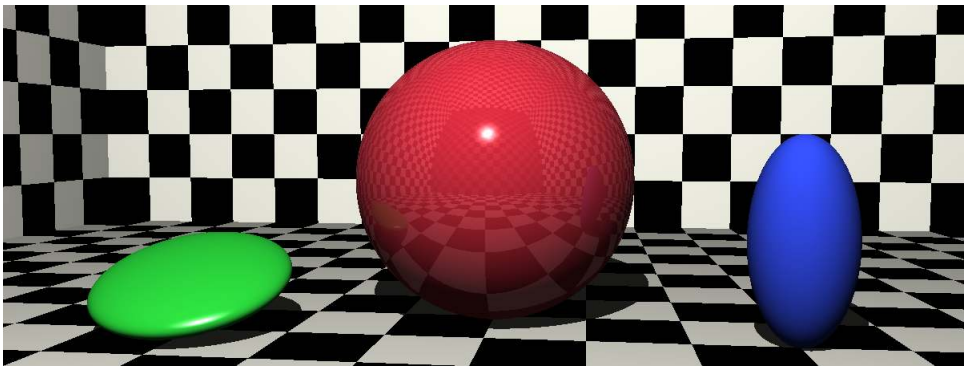
A velocidade do projétil foi normalizada para um vetor unitário e então multiplicada por 11,25 para aumentar sua magnitude. Isso, o vetor de velocidade e o tamanho da tela foram todos determinados empiricamente. Experimente diferentes vetores e velocidades iniciais e veja o que acontece!

Depois de brincar um pouco com isso, siga em frente! Voltaremos ao modo matemático nos próximos capítulos para desenvolver mais alguns fundamentos de que você precisará para o seu ray tracer.

CAPÍTULO 3

# Matrizes

Ei, olhe. Aquela esfera vermelha brilhante de antes tem companhia agora. Seus amigos parecem ser um ovoide azul fosco com aparência de charuto e uma coisa de plástico verde amassada que está virada em nossa direção, como se estivesse curiosa para ver quem está olhando.



Você ficaria surpreso em saber que tudo isso são apenas esferas? Eles também foram movidos, dimensionados e girados um pouco, mas, no fundo, ainda são perfeitamente esféricos. Essas transformações são todas graças a uma coisinha chamada *matriz.*

Uma matriz é uma grade de números que você pode manipular como uma única unidade. Pra exemplo, aqui está uma matriz 2x2. É ha [ s dois] linhas e duas colunas.

$$\begin{matrix} 3 & 1 \\ 2 & 7 \end{matrix}$$

E aqui está uma matriz 3x5, com • três linhas e • cinco colunas:

$$\bullet \begin{matrix} 9 & 1 & 2 & 0 & 3 \\ 0 & 0 & 2 & 3 & 1 \\ 8 & 7 & 5 & 4 & 6 \end{matrix} \bullet$$

Para o seu ray tracer, você usará principalmente matrizes 4x4 - aquelas com exatamente quatro linhas e quatro colunas, como esta •

s:

•

1 2 0 0

0 1 4 1 •

0 1 1 3 •

0 0 0 1

•

Neste capítulo, você implementará uma estrutura de dados de matriz 4x4 e algumas operações gerais de matriz. No capítulo após este, Capítulo 4, Transformações de matriz, na página 43 , você desenvolverá essas operações, adicionando funcionalidade para facilitar a manipulação de pontos e vetores (e, em última análise, formas).

Preparar? Vamos fazer isso!

## Criação de uma matriz

Primeiras coisas primeiro. Você precisa ser capaz de descrever uma nova matriz. Escreva um teste como o seguinte, que mostra que uma matriz é composta por quatro linhas de quatro números de ponto flutuante cada, para um total de dezesseis números. Deve também mostrar como se referir aos elementos da matriz.

**features / matrices.feature**

```
Cenário : Construindo e inspecionando uma matriz 4x4
    Dado a seguinte matriz 4x4 M:
        | 1     | 2     | 3     | 4     |
        | 5,5 | 6,5 | 7,5 | 8,5 | | 9
                | 10    | 11    | 12    |
        | 13,5 | 14,5 | 15,5 | 16,5 |
    Então M [0,0] = 1
        E M [0,3] = 4
        E M [1,0] = 5,5
        E M [1,2] = 7,5
        E M [2,2] = 11
        E M [3,0] = 13,5
        E M [3,2] = 15,5
```

A primeira coisa a notar é quando falamos sobre os elementos individuais da matriz, especificamos o elemento *fileira* primeiro, e então é *coluna.* Por exemplo,
elemento $M_{23}$ é aquele na linha 2, coluna 3. Observe também neste livro, os índices de linha e coluna serão baseados em zero, portanto, a linha 2 é na verdade a terceira linha.

Mais tarde, em Invertendo matrizes, na página 33 , você precisará ser capaz de instanciar matrizes 2x2 e 3x3 além de matrizes 4x4, portanto, reserve um momento para se certificar de que também será possível criar matrizes desses tamanhos. Adicione os seguintes testes para mostrar que seu código suporta essas dimensões:

**features / matrices.feature**

**Cenário :** Uma matriz 2x2 deve ser representável

**Dado** a seguinte matriz 2x2 M:

| -3 | 5 |

| 1 -2 |

**Então** M [0,0] = -3

**E** M [0,1] = 5

**E** M [1,0] = 1

**E** M [1,1] = -2

**Cenário :** Uma matriz 3x3 deve ser representável

**Dado** a seguinte matriz 3x3 M:

| -3 | 5 | 0 |

| 1 -2 | -7 |

| 0 | 1 1

**Então** M [0,0] = -3

**E** M [1,1] = -2

**E** M [2,2] = 1

Mantenha sua implementação de matriz o mais simples possível. Prefira tipos nativos sempre que puder e evite abstrações complicadas. Suas matrizes farão muito trabalho!

Outra parte crítica de sua implementação de matriz é *comparação de matrizes.*

Você estará comparando muito as matrizes, especialmente neste capítulo e no próximo, por isso é importante fazer a coisa certa. Os dois testes a seguir não são exaustivos, mas devem indicar a direção certa. Por exemplo, você deseja certificar-se de que números muito semelhantes sejam tratados corretamente ao comparar matrizes, conforme descrito em Comparando Números de Ponto Flutuante, na página 5 .

**features / matrices.feature**

**Cenário :** Igualdade de matriz com matrizes idênticas

**Dado** a seguinte matriz A:

| 1 2 | 3 | 4 | 5 | 6 7 8 | 9 8 7 6 | 5

| 4 3 | 2 |

**E** a seguinte matriz B:

| 1 2 | 3 | 4 | 5 | 6 7 8 | 9 8 7 6 | 5

| 4 3 | 2 |

**Então** A = B

**Cenário :** Igualdade de matriz com matrizes diferentes

**Dado** a seguinte matriz A:

| 1 2 | 3 | 4 | 5 | 6 7 8 | 9 8 7 6 | 5

| 4 3 | 2 |

**E** a seguinte matriz B:

| 2 | 3 | 4 5 | | 6 7 8 9 | 8 7 6 5 | |

4 3 | 2 | 1

**Então** A! = B

Assim que você tiver a estrutura básica de dados da matriz funcionando, a álgebra linear será sua ostra. Faremos algumas coisas malucas com matrizes, mas começaremos devagar; vamos falar sobre como multiplicá-los.

## Multiplicando Matrizes

Multiplicação é a ferramenta que você usará para realizar transformações como dimensionamento, rotação e translação. Certamente é possível aplicá-los um de cada vez, sequencialmente, mas na prática, você frequentemente desejará aplicar várias transformações de uma vez. Multiplicá-los é como você faz isso acontecer, como você verá quando chegar a Capítulo 4, Transformações de matriz, na página 43 .

Então, vamos falar sobre multiplicação de matrizes. Ele pega duas matrizes e produz outra matriz multiplicando seus elementos componentes de uma maneira específica. Você verá como isso funciona em breve, mas comece primeiro escrevendo um teste que descreve o que você espera que aconteça ao multiplicar duas matrizes 4x4. Não se preocupe com matrizes 2x2 ou 3x3 aqui; seu ray tracer não precisará multiplicá-los.

**features / matrices.feature**

**Cenário** : Multiplicando duas matrizes

**Dado** Os seguintes matriz A:

| 1 2 | 3 | 4 |

| 5 | 6 7 8 |

| 9 8 7 6 |

| 5 | 4 3 | 2 |

**E** Os seguintes matriz B:

| -2 | 1 2 | 3 |

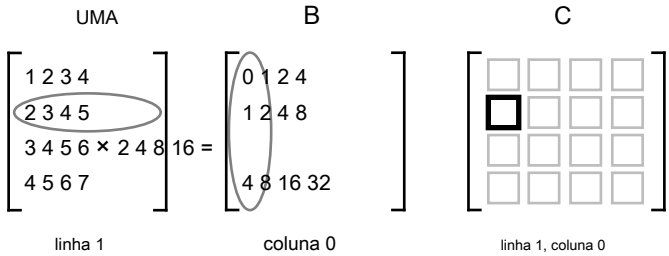
| 3 | 2 | 1 -1 | | 4 3 | 6 5 | | 1 2 | 7 8

**Então** A * B é a seguinte matriz 4x4:

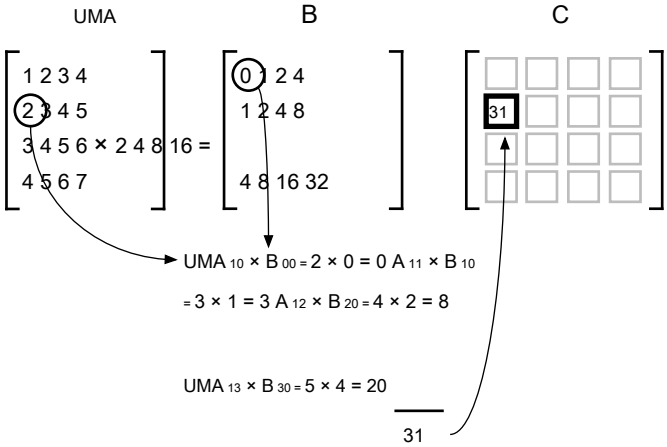
| 20 22 50 48 | 44 54 114 108 | 40 58 110 102 | 16 26 46 42

Vejamos como isso é feito para um único elemento de uma matriz, indo passo a passo para encontrar o produto para o elemento $C_{10}$, destacado no figura na página 29 .

Elemento $C_{10}$ está na linha 1, coluna 0, então você precisa olhar para a linha 1 do UMA matriz e coluna 0 do B matriz, conforme mostrado na figura a seguir.




Então, você multiplica pares correspondentes de elementos juntos ($A_{10}$ e $B_{00}$, UMA $_{11}$ e $B_{10}$, UMA $_{12}$ e $B_{20}$, e $A_{13}$ e $B_{30}$), e adicione os produtos. A figura a seguir mostra como isso funciona.




O resultado, aqui, é 31, e para encontrar os outros elementos, você executa o mesmo processo para cada combinação linha-coluna das duas matrizes.

Declarada como um algoritmo, a multiplicação de duas matrizes 4x4 tem a seguinte aparência:

1. Deixe UMA e B sejam as matrizes a serem multiplicadas, e deixe M seja o resultado.
2. Para cada linha r no UMA, e cada coluna c no B:

3. Deixe $M_{rc} = UMA_{r0} * B_{0c} + UMA_{r1} * B_{1c} + UMA_{r2} * B_{2c} + UMA_{r3} * B_{3c}$

Como pseudocódigo, o algoritmo pode ter a seguinte aparência:

```
função matrix_multiply (A, B)
    M ← matriz ()

    pra fileira ← 0 a 3
        pra col ← 0 a 3
            M [ linha, col ] ← UMA [ linha, 0 ] * B [ 0, col ] +
                                UMA [ linha, 1 ] * B [ 1, col ] + A [ linha, 2 ] * B [ 2,
                                col ] + A [ linha, 3 ] * B [ 3, col ]

        fim para
    fim para

    Retorna M
função final
```

Se tudo isso parece familiar, pode ser porque você já implementou algo muito semelhante - o produto escalar de dois vetores na página 10 . Sim, é verdade. A multiplicação de matrizes calcula o produto escalar de cada combinação linha-coluna nas duas matrizes! Muito legal.

Agora, não terminamos ainda. As matrizes podem ser multiplicadas por *tuplas,* além de outras matrizes. Multiplicar uma matriz por uma tupla produz outra tupla. Comece com um teste novamente, como o seguinte, para expressar o que você espera que aconteça ao multiplicar uma matriz e uma tupla.

**features / matrices.feature**

```
Cenário : UMA        matriz multiplicada por uma tupla seguindo a matriz
    Dado a           A:
        | 1          2 | 3 | 4
        | 2 |        4 4 2 |
        | 8          6 4 1
        | 0 |        0 | 0 | 1
    E b ← tupla (1, 2, 3, 1)
    Então A * b = tupla (18, 24, 33, 1)
```

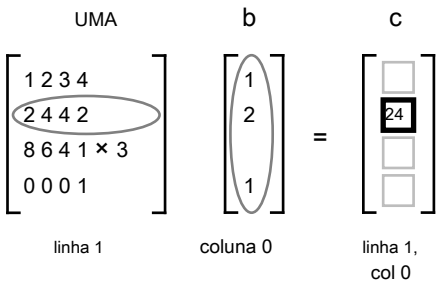
Como funciona? O truque começa tratando a tupla como uma matriz muito fina (uma coluna!), Assim:

$$(1, 2, 3, 1) \Rightarrow 3 \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

Quatro filas. Uma coluna.

Ele vem junto com a multiplicação de duas matrizes 4x4, mas agora você está lidando apenas com uma única coluna na segunda "matriz". O

A figura a seguir ilustra isso, destacando a linha e a coluna usadas ao calcular o valor de $c_{10}$




Para calcular o valor de $c_{10}$, você considera apenas a linha 1 da matriz UMA, e coluna 0 (o *só* coluna!) de tupla b. Se você pensar nessa linha da matriz como um
tupla, então a resposta é encontrada tomando o produto escalar dessa linha e a outra tupla:

$$2 \times 1 + 4 \times 2 + 4 \times 3 + 2 \times 1 = 24$$

Os outros elementos de c são calculados de forma semelhante. Na verdade, é exatamente o mesmo algoritmo usado para multiplicar duas matrizes, com a única diferença sendo o número de colunas na segunda "matriz".

Se você estiver se sentindo desconfortável com a quantidade de magia que existe nesses algoritmos, consulte "Um guia intuitivo para álgebra linear [1] "

em BetterExplained.com. Ele faz um bom trabalho em dar sentido a essas coisas!

Faça uma pausa aqui para que os testes que você escreveu até agora sejam aprovados. Depois de colocá-los em funcionamento, continue! Vamos examinar uma matriz muito especial e usar a multiplicação para entender um pouco do que a torna tão especial.

## A Matriz de Identidade

Você sabe que pode multiplicar qualquer número por 1 e obter o número original. O número 1 é chamado de *identidade multiplicativa* por essa razão. Bem o
*matriz de identidade* é como o número 1, mas para matrizes. Se você multiplicar qualquer matriz ou tupla pela matriz identidade, você obtém a matriz ou tupla com a qual começou.

Isso pode parecer totalmente inútil agora, mas considere o seguinte: se a multiplicação pela matriz de identidade apenas retorna o valor original, significa que você pode usá-lo como a transformação padrão para qualquer objeto em sua cena. Você não precisa

1 betterexplained.com/articles/linear-algebra-guide

quaisquer casos especiais para dizer a diferença entre uma forma com uma transformação e uma forma sem. Isso é, na verdade, exatamente para o que você vai usá-lo quando chegar a Capítulo 5, Intersecções Ray-Sphere, na página 57 .

Adicione os seguintes testes para ilustrar o (não) efeito da multiplicação pela matriz de identidade.

**features / matrices.feature**

```
Cenário : Multiplicando                    uma matriz pela matriz de identidade matriz A:
    Dado Os seguintes
        | 0 | 1 2 | 4                      |
        | 1 2 | 4 8                        |
        | 2 | 4 8 16 4 8 16 32             |
                                           |
    Então A * identity_matrix = A

Cenário : Multiplicando a matriz de identidade por uma tupla
    Dado uma ← tupla (1, 2, 3, 4)
    Então identidade_matriz * a = a
```

A matriz de identidade é composta de zeros, exceto para os elementos ao longo da diagonal, que são definidos como 1:

$$\text{identidade} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Novamente, você só precisa se preocupar com a matriz de identidade 4x4 do seu ray tracer. A seguir, vamos examinar outra operação de matriz.

## TransposingMatrices

*Transposição de matriz* será útil quando você chegar a Capítulo 6, Luz e Sombreamento, na página 75 . Você o usará ao traduzir certos vetores (chamados *normal* vetores) entre o espaço do objeto e o espaço do mundo. Isso pode soar como ficção científica, mas é crucial para sombrear seus objetos corretamente.

Ao transpor uma matriz, você transforma suas linhas em colunas e suas colunas em linhas:

$$\text{transpor} \begin{bmatrix} 0 & 9 & 3 & 0 \\ 9 & 8 & 0 & 8 \\ 1 & 8 & 5 & 3 \\ 0 & 0 & 5 & 8 \end{bmatrix} = \begin{bmatrix} 0 & 9 & 1 & 0 \\ 9 & 8 & 8 & 0 \\ 3 & 0 & 5 & 5 \\ 0 & 8 & 3 & 8 \end{bmatrix}$$

Transpor uma matriz transforma a primeira linha na primeira coluna, a segunda linha na segunda coluna e assim por diante. Aqui está um teste que demonstra isso.

**features / matrices.feature**

**Cenário :** Transpondo uma matriz

**Dado** a seguinte matriz A:

| 0 | 9 3 | 0 | | 9 8 0 | 8 | 1 8 5 | 3

| | 0 | 0 | 5 | 8

**Então** transpor (A) é a seguinte matriz:

| 0 | 9 1 0 | | 9 8 8 0 | | 3 | 0 | 5 |

5 | | 0 | 8 3 | 8

E, curiosamente, a transposição da matriz de identidade sempre fornece a matriz de identidade. Implemente o seguinte teste para mostrar que isso é verdade.

**features / matrices.feature**

**Cenário :** Transpondo a matriz de identidade

**Dado** UMA ← transpor (matriz de identidade)

**Então** A = identidade_matriz

Ver? Diversão boa e limpa. Faça esses testes passarem e siga em frente. É hora de falar sobre *inversão de matriz.*

## InvertingMatrices

Se você multiplicar 5 por 4, obterá 20. Se mais tarde decidir desfazer essa operação, você pode multiplicar 20 pelo *inverso* de 4 (ou $1/4$) e pegue 5 novamente.

Essa também é a ideia para matrizes. Se você multiplicar alguma matriz UMA por outra matriz B, produzindo C, você pode multiplicar C pelo inverso de B para obter UMA de novo. Você usará esta abordagem a *muito,* começando em Capítulo 5, Intersecções Ray-Sphere, na página 57 , porque a inversão de matrizes é a chave para transformar e deformar formas em um ray tracer.

No entanto, inverter matrizes é um pouco mais complicado do que inverter números. Você vai empregar um método conhecido como *expansão do cofator.* Se isso parece intimidador, anime-se! Faremos uma abordagem agradável e lenta, um passo de cada vez. Começando com rotinas para calcular o determinante de uma matriz 2x2, passaremos de forma incremental por coisas que parecem misteriosas, como submatrizes, menores e cofatores, e depois voltaremos aos determinantes novamente. Finalmente, encerraremos este capítulo com o próprio algoritmo para inversão de matrizes.

Vamos começar com o *determinante.*

## Determinando Determinantes

O *determinante* é um número derivado dos elementos de uma matriz. O nome vem do uso de matrizes para resolver sistemas de equações, onde é usado para *determinar* se o sistema tem ou não uma solução. Se o determinante for zero, o sistema de equações correspondente não terá solução.

No entanto, você não usará matrizes para resolver equações aqui. Para você, o determinante é apenas uma das peças que você usará para calcular o inverso de uma matriz.

Começaremos pequeno, construindo o algoritmo de baixo para cima. É aqui que essas matrizes 2x2 são úteis, porque a inversão de matrizes maiores começa por encontrar os determinantes de matrizes 2x2. Adicione o seguinte teste ao seu conjunto, para mostrar que seu código pode fazer exatamente isso.

**features / matrices.feature**

```
Cenário : Calculando o determinante de uma matriz 2x2
    Dado a seguinte matriz A 2x2:
        | 1 5 |
        | -3 | 2 |
    Então determinante (A) = 17
```

Funciona assim:

$$\text{determinante} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

Não é adorável? Essa é toda a mágica de que você precisa para encontrar o determinante de uma matriz 2x2! Isso aí é a semente para tudo o mais envolvido na inversão de matrizes.

Você precisa de mais algumas ferramentas antes de encontrar o determinante de uma matriz maior. Ser paciente! Faça o novo teste passar e continue lendo. O próximo conceito que você precisa implementar é o de *submatrizes,* que será usado para ajudar a reduzir matrizes maiores para tamanhos com os quais você sabe trabalhar.

## Spotting Submatrices

UMA *submatriz* é o que resta quando você exclui uma única linha e coluna de uma matriz. Como você sempre remove uma linha e uma coluna, efetivamente reduz o tamanho da matriz em um. A submatriz de uma matriz 4x4 é 3x3 e a submatriz de uma matriz 3x3 é 2x2. E adivinha? Você sabe como encontrar o determinante de matrizes 2x2! As submatrizes são as próprias ferramentas que você usará para dividir e conquistar os animais maiores.

Adicione os dois testes a seguir que mostram o que você obtém ao extrair uma submatriz de uma matriz. Eles apresentam uma nova função, submatriz (matriz, linha, coluna),
que retorna uma cópia da matriz fornecida com a linha e coluna fornecidas removidas.

**features / matrices.feature**

```
Cenário : Uma submatriz de uma matriz 3x3 é uma matriz 2x2
    Dado a seguinte matriz A 3x3:
        | 1 5 | 0 |
        | -3 | 2 | 7
        | 0 | 6 -3 |
    Então submatriz (A, 0, 2) é a seguinte matriz 2x2:
        | -3 | 2 |
        | 0 | 6

Cenário : Uma submatriz                 de uma matriz 4x4 é uma matriz 3x3 4x4 matriz A:
    Dado Os seguintes
        | -6 | 1 1                     6
        | -8 | 5 | 8                   6
        | -1 | 0 | 8                   2 |
        | -7 | 1 -1 |                  1
    Então submatriz (A, 2, 1) é a seguinte matriz 3x3:
        | -6 | 1 6
        | -8 | 8 6
        | -7 | -1 | 1
```

Não há mágica ali e, realmente, não há matemática. Eu não disse que íamos levar isso bem devagar? Vá em frente e faça esses testes passarem. Os próximos são
*menores.*

## Manipulando Minores

Ok, então agora você está familiarizado com *determinantes* e *submatrizes.* Isso é perfeito, porque agora você tem todas as ferramentas de que precisa para calcular o *menores*
de uma matriz 3x3. (Ainda não é bem 4x4, mas você está chegando perto!)

O *menor* de um elemento na linha eu e coluna j é o *determinante* do *submatriz* no ( eu j). Implemente o seguinte teste, que apresenta uma nova função,
menor (matriz, linha, coluna).

**features / matrices.feature**

```
Cenário : Calculando um menor de uma matriz 3x3
    Dado a seguinte matriz A 3x3:
            | 3 | 5 | 0 |
            | 2 | -1 | -7 |
            | 6 -1 | 5 |
        E B ← submatriz (A, 1, 0)
    Então determinante (B) = 25
        E menor (A, 1, 0) = 25
```

Veja isso? Você encontra a submatriz no local fornecido e, a seguir, calcula o determinante dessa submatriz. A resposta é a menor. (Você tem que admitir: "menor" é mais fácil de dizer do que "determinante da submatriz".)

Faça esse teste passar e, em seguida, veremos o último conceito de que precisamos para começar a montar esse quebra-cabeça de inversão de matrizes.

## Co-fatores de computação

*Cofatores* são a última ferramenta de que você precisará para calcular os determinantes de matrizes maiores. São menores de idade que (possivelmente) tiveram seu sinal alterado. Adicione o seguinte teste para demonstrar o que é esperado do cofator. É uma introdução
produz uma nova função, cofator (matriz, linha, coluna).

**features / matrices.feature**

```
Cenário : Calculando um cofator de uma matriz 3x3
    Dado a seguinte matriz A 3x3:
            | 3 | 5 | 0 |
            | 2 | -1 | -7 |
            | 6 -1 | 5 |
    Então menor (A, 0, 0) = -12
        E cofator (A, 0, 0) = -12
        E menor (A, 1, 0) = 25
        E cofator (A, 1, 0) = -25
```

Então, como isso funciona? Bem, primeiro você calcula o menor na linha e coluna fornecidas. Então você considera essa linha e coluna para determinar se ou não
para negar o resultado. O seguinte • g figura é • útil:

```
  +     - + •
• - + -
  +     - +
```

Se a linha e a coluna identificam um ponto com um +, o sinal do menor não muda. Se a linha e a coluna identificam um ponto com um •, então você nega o menor.

Claro, você também pode fazer isso sem olhar para uma figura: se linha + coluna é um número ímpar, então você nega o menor. Caso contrário, basta devolver o menor como está. Faça esse teste passar e depois continue a ler!

## Determinando determinantes de matrizes maiores

Agora que você tem essas três ideias prontas - determinantes, menores e cofatores - você pode finalmente implementar o determinante das matrizes 3x3 e 4x4. (Na verdade, a ideia também se generaliza para matrizes arbitrariamente grandes, mas para seus propósitos aqui, você não precisa ir além de 4x4.)

Primeiro, prepare o cenário escrevendo os dois testes a seguir, mostrando o determinante e alguns dos cofatores de uma matriz 3x3 e 4x4. (Por que os cofatores? Espere um pouco. Tudo ficará claro em breve!)

**features / matrices.feature**

```
Cenário : Calculando o determinante de uma matriz 3x3
    Dado a seguinte matriz A 3x3:
        | 1 2 | 6
        | -5 | 8 -4 |
        | 2 | 6 4
    Então cofator (A, 0, 0) = 56
        E cofator (A, 0, 1) = 12
        E cofator (A, 0, 2) = -46
        E determinante (A) = -196

Cenário : Calculando o determinante de uma matriz 4x4
    Dado a seguinte matriz A 4x4:
        | -2 | -8 | 3 | 5 | | -3 | 1 7 3 | | 1 2 | -9 | 6 |
        -6 | 7 7 -9 |


    Então cofator (A, 0, 0) = 690
        E cofator (A, 0, 1) = 447
        E cofator (A, 0, 2) = 210
        E cofator (A, 0, 3) = 51
        E determinante (A) = -4071
```

Esses testes ainda não deveriam estar passando. Vamos consertar isso.

Encontrar o determinante de matrizes maiores que 2x2 funciona recursivamente. Vigarista-

Considere a matriz 3x3 do p • testes revious •.

1 2 6 •
• - 5 8 - 4
2 6 4

Para encontrar o determinante, olhe para qualquer uma das linhas ou colunas. É realmente não importa qual, então vamos apenas • eu escolho • a primeira linha.

1 2 6 •
• . . .
. . .

Em seguida, para cada um desses elementos, você multiplicará o elemento por seu cofator e adicionará os produtos.

$$1 \cdot 56 + 2 \cdot 12 + 6 \cdot - 46 = - 196$$

E esse é o determinante! O mágico é que não importa a linha ou coluna que você escolher. *Simplesmente funciona.*

E funciona para matrizes 4x4 também. Aqui, considere a matriz do teste que você escreveu:

| | | | |
|---|---|---|---|
| - 2 | - 8 | 3 | 5 |
| - 3 | 1 | 7 | 3 |
| 1 | 2 | - 9 | 6 |
| - 6 | 7 | 7 | - 9 |

Mais uma vez, você só precisa olhar para uma única linha ou coluna, então vamos escolher a primeira linha.

| | | | |
|---|---|---|---|
| - 2 | - 8 | 3 | 5 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |

Em seguida, multiplique cada elemento por seu cofator e adicione os resultados.

$$- 2 \cdot 690 + - 8 \cdot 447 + 3 \cdot 210 + 5 \cdot 51 = - 4071$$

*Voilà!* O determinante!

Não há como negar que é muito para processar, no entanto. Para lhe dar uma vantagem, aqui está um pouco de pseudocódigo para esse algoritmo:

```
função determinante (M)
    det ← 0

    E se M.size = 2
        det ← M [ 0, 0 ] * M [ 1, 1 ] - M [ 0, 1 ] * M [ 1, 0 ]

    outro
        pra coluna ← 0 a M.size - 1
            det ← det + M [ 0, coluna ] * cofator (M, 0, coluna)
        fim para
    fim se

    Retorna det
função final
```

Vá em frente e faça esses testes passarem. Você está na reta final agora. Com um determinante totalmente funcional, você está pronto para enfrentar *inversão.*

## Implementando Inversão

Ok, você está no ponto culminante de todo esse processo agora. É aqui que tudo se junta! Lembre-se de que inversão é a operação que permite reverter o efeito da multiplicação por uma matriz. Será crucial para a transformação de formas em seu ray tracer, permitindo que você mova as formas, torne-as maiores ou menores, gire-as e muito mais. Não é exagero dizer que, sem inversão, não adianta construir mais nada!

Agora, uma das coisas complicadas sobre a inversão de matriz é que nem toda matriz é invertível. Antes de mergulhar de cabeça na inversão de matrizes, você deve primeiro ser capaz de identificar se essa tarefa é mesmo possível!

Adicione os seguintes testes para mostrar que seu código pode diferenciar matrizes invertíveis das não invertíveis.

**features / matrices.feature**

```
Cenário : Testando uma matriz invertível para invertibilidade
    Dado a seguinte matriz A 4x4:
        | 6 4 4 4 | 5 | 5 | 7 6 | 4 -9 | 3 | -7 | | 9 1 7
        -6 |


    Então determinante (A) = -2120
        E A é invertível

Cenário : Testando uma matriz não reversível para invertibilidade
    Dado a seguinte matriz A 4x4:
        | -4 | 2 | -2 | -3 | | 9 6 2 | 6 | 0 | -5 | 1 -5 |
        | 0 | 0 | 0 | 0 |


    Então determinante (A) = 0
        E A não é invertível
```

E assim como os testes sugerem, o determinante é a chave. Se o determinante for sempre 0, a matriz não pode ser invertida. Qualquer outra coisa está bem.

Assim que estiver funcionando, adicione o seguinte teste. Ele exerce uma nova função chamada inverso (matriz), que produz o inverso da matriz fornecida.

**features / matrices.feature**

```
Cenário : Calculando o                          inversa de uma matriz matriz A:
    Dado o seguinte 4x4
            | -5 | 2 | 6 -8 | 1 -5 | 1 8 7 7 -6 | -7 | 1  |
            -3 | 7 4                                      |
                                                           |
                                                           |
        E B ← inverso (UMA)
    Então determinante (A) = 532
        E cofator (A, 2, 3) = -160
        E B [3,2] = -160/532
        E cofator (A, 3, 2) = 105
        E B [2,3] = 105/532
        E B é a seguinte matriz 4x4:
            | 0,21805 | 0,45113 | 0,24060 | -0,04511 | | -0,80827 | -1,45677 | -0,44361 | 0,52068 |
            | -0,07895 | -0,22368 | -0,05263 | 0,19737 | | -0,52256 | -0,81391 | -0,30075 | 0,30639
            |
```

Não é por acaso que o teste também calcula alguns cofatores e determinantes - tudo está relacionado ao algoritmo de inversão em si. Esse algoritmo consiste em várias etapas, começando com a construção de um *matriz de cofatores.*

Ou seja, você cria uma matriz que consiste nos cofatores de cada um dos elementos originais:

$$\begin{bmatrix} -5 & 2 & 6-8 \\ 1-5 & 1 & 8 \\ 7 & 7-6-7 & \\ 1-3 & 7 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 116-430 & -42-278 \\ 240-775-119-433 & \\ 128-236 & -28-160 \\ -24 & 277 \quad 105 \quad 163 \end{bmatrix}$$

Então, tra • suponha que o cofator matr • ix: •

$$\begin{bmatrix} 116-430 & -42-278 \\ 240-775-119-433 & \\ 128-236 & -28-160 \\ -24 & 277 \quad 105 \quad 163 \end{bmatrix} \Rightarrow \begin{bmatrix} 116 & 240 & 128-24 \\ -430-775-236\ 277 & & \\ -42-119 & -28\ 105 & \\ -278-433-160\ 163 & & \end{bmatrix}$$

Finalmente, divida cada um dos elementos resultantes pelo determinante da matriz original.

$$\begin{bmatrix} 116 & 240 & 128-24 \\ -430-775-236\ 277 & & \\ -42-119 & -28\ 105 & \\ -278-433-160\ 163 & & \end{bmatrix} \div 532 \Rightarrow \begin{bmatrix} 0{,}21805 & 0{,}45113 & 0{,}24060-0{,}04511 \\ -0{,}80827-1.45677-0{,}44361 & & 0{,}52068 \\ -0{,}07895-0{,}22368-0{,}05263 & & 0{,}19737 \\ -0{,}52256-0{,}81391-0{,}30075 & & 0{,}30639 \end{bmatrix}$$

Uau! E isso é o inverso. Que viagem!

Embora seja certamente possível implementar isso fazendo exatamente o que os exemplos anteriores sugerem (encontrar a matriz de cofatores e, em seguida, transpô-la e assim por diante), você pode realmente fazer isso com um pouco mais de eficiência combinando as operações. Aqui está um pseudocódigo que demonstra o que quero dizer:

```
função inverso (M)
    falhou E se M não é invertível

    M2 ← nova matriz do mesmo tamanho que M

    pra fileira ← 0 a M.size - 1
        pra col ← 0 a M.size - 1
            c ← cofator (M, linha, col)

            # observe que "coluna, linha" aqui, em vez de "linha, coluna",
            # realiza a operação de transposição!
            M2 [ coluna, linha ] ← c / determinante (M)
        fim para
    fim para

    Retorna M2
função final
```

É importante que tudo isso esteja correto. Quaisquer erros neste código causarão inúmeras dores de cabeça no caminho. Adicione os dois testes a seguir para fornecer um pouco mais de cobertura para suas rotinas de matriz.

**features / matrices.feature**

```
Cenário : Calculando o inverso de outra matriz
    Dado a seguinte matriz A 4x4:
        | 8 -5 | 9 2 | | 7 5 | 6 1 | -6 | 0 | 9 6 | -3 | 0
        | -9 | -4 |

    Então inverso (A) é a seguinte matriz 4x4:
        | -0.15385 | -0.15385 | -0,28205 | -0,53846 | | -0,07692 | 0,12308 | 0,02564 | 0,03077
        | | 0,35897 | 0,35897 | 0,43590 | 0,92308 | | -0,69231 | -0,69231 | -0,76923 | -1,92308
        |

Cenário : Calculando o inverso de uma terceira matriz
    Dado a seguinte matriz A 4x4:
        | 9 3 | 0 | 9 | -5 | -2 | -6 | -3 | | -4 | 9 6 4 |
        -7 | 6 6 2 |

    Então inverso (A) é a seguinte matriz 4x4:
        | -0,04074 | -0,07778 | 0,14444 | -0,22222 | | -0,07778 | 0,03333 | 0,36667 | -0,33333
        | | -0,02901 | -0,14630 | -0,10926 | 0,12963 | | 0,17778 | 0,06667 | -0,26667 | 0,33333
        |
```

Uma última coisa a se notar sobre o inverso: no início desta seção, você lê que “se você multiplicar alguma matriz UMA por outra matriz B, produzindo C,
você pode multiplicar C pelo inverso de B para obter UMA de novo." Bem, não podemos deixar essa afirmação passar sem comprovação! Adicione mais um teste para mostrar que o inverso, na verdade, se comporta conforme descrito.

**features / matrices.feature**

```
Cenário : Multiplicando um produto pelo seu inverso
    Dado a seguinte matriz A 4x4:
            | 3 | -9 | 7 3 | | 3 | -8 | 2 | -9 | | -4 | 4 4 1 |
            -6 | 5 | -1 | 1

        E a seguinte matriz B 4x4:
            | 8 2 | 2 | 2 | | 3 | -1 | 7 0 | | 7 0 | 5 | 4 | 6
            -2 | 0 | 5 |

        E C ← A * B
    Então C * inverso (B) = A
```

Certifique-se de que todos os seus testes estão passando agora. Quando tudo estiver verde, respire fundo e dê um tapinha nas costas. Você acabou de implementar um dos pilares da álgebra linear - com testes, até!

## Juntar as peças

Agora você tem matrizes 4x4 com suporte para multiplicação, transposição e inversão. Nada mal!

Infelizmente, não há muito relacionado ao traçado de raios que você possa fazer com essas rotinas ainda, mas você cuidará de *que* pequeno problema no próximo capítulo, Capítulo 4, Transformações de matriz, na página 43 . No entanto, sempre há espaço para um pouco de experimentação. Antes de prosseguir, reserve alguns minutos para explorar um pouco mais.

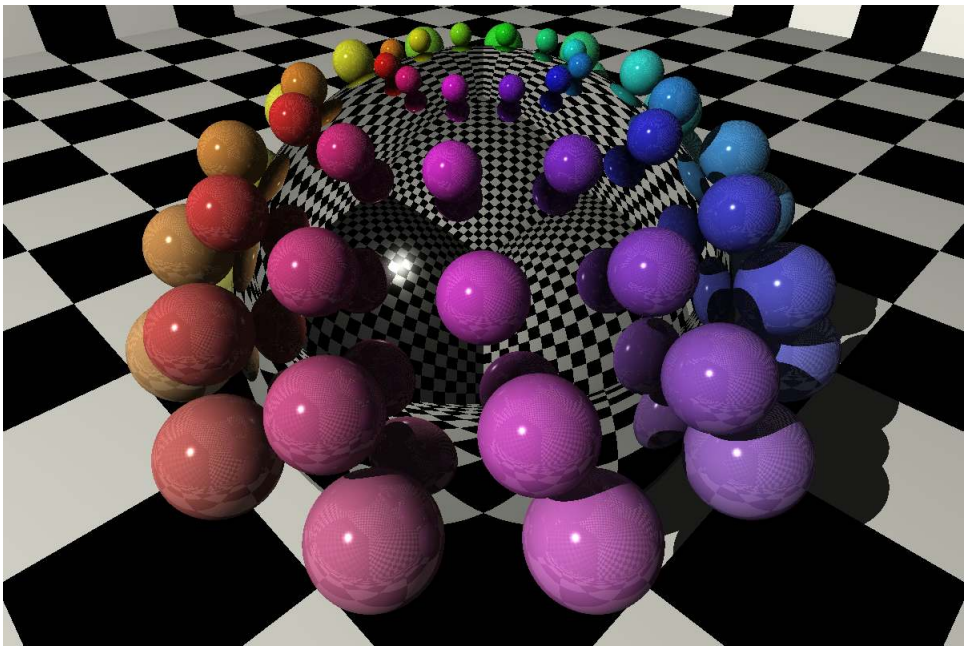
1. O que acontece quando você inverte a matriz de identidade?
2. O que você obtém quando multiplica uma matriz pelo seu inverso?
3. Existe alguma diferença entre o *inverso* do *transpor* de uma matriz, e a *transpor* do *inverso?*
4. Lembra-se de como multiplicar a matriz de identidade por uma tupla resulta na tupla, inalterada? Agora, tente alterar qualquer elemento único da matriz de identidade para um número diferente e, em seguida, multiplique-o por uma tupla. O que acontece com a tupla?

Quando estiver pronto, vire a página! No próximo capítulo, você usará suas matrizes para implementar *transformações,* entidades que o ajudarão a posicionar e orientar os objetos em suas cenas.

CAPÍTULO 4

# Transformações matriciais

Awesomesauce! Você está prestes a pegar a base das operações de matriz que implementou no capítulo anterior e começar a fazer algumas coisas práticas, como *transformações,* que o seu ray tracer irá (eventualmente) usar para mover e deformar objetos. Considere a seguinte cena.



Seu ray tracer não será capaz de renderizar essas reflexões até Capítulo 11, Reflexão e Refração, na página 141 , mas a cena em si não é muito complicada - algumas esferas coloridas, alguns planos quadriculados. A parte relevante aqui, porém, é como cada uma dessas esferas é dimensionada e posicionada. Sem transformações, você teria que descrever explicitamente o raio e a localização de cada esfera, o que seria tedioso (em um sentido decididamente trigonométrico) para obter

correto. Talvez surpreendentemente, isso também aumentaria a complexidade do seu ray tracer, como você verá em Capítulo 5, Intersecções Ray-Sphere, na página 57 .

*Com* transformações, no entanto, você adiciona cada uma dessas esferas menores à cena na origem e, em seguida, aplica uma série de transformações a elas: escala, translação e algumas rotações. Sem matemática complicada ou cálculos tediosos envolvidos!

O melhor de tudo é que essas transformações usam as operações de matriz que você acabou de concluir. Veremos como construir uma matriz para representar cada uma dessas transformações, bem como encadear várias delas como uma única matriz.

Preparar? Vamos começar com a tradução.

## Tradução

*Tradução* é uma transformação que move um ponto, assim.

Ele muda as coordenadas do ponto adicionando ou subtraindo deles. Por exemplo, se o ponto tivesse um x coordenada de 3, e você moveu 4 unidades em x, acabaria com um x coordenada de 7.

**Joe pergunta:**

**Não podemos simplesmente usar vetores para traduzir pontos?**

Bem, sim, na verdade, podemos. Você viu em *Tuplas, pontos e vetores* como adicionar um vetor a um ponto e, assim, transladar o ponto na direção do vetor. Isso funciona bem.

O problema com isso é que ele só pode fazer translação - não podemos usar a mesma operação (ou seja, adicionar um vetor) e obter rotação, escala ou cisalhamento. O que queremos é uma única operação que possa produzir qualquer uma dessas transformações e concatená-las em ordem arbitrária.

A multiplicação de matrizes é uma dessas ferramentas.

O burro de carga aqui será um novo `tradução(x, y, z)` função que deve retornar uma matriz de tradução 4x4. Implemente o seguinte teste para mostrá-lo em ação. Não se preocupe em fazer os próximos testes passarem; Eu vou te mostrar o molho secreto em breve.

**features / transformations.feature**

```
Cenário : Multiplicando por uma matriz de tradução
    Dado transformar ← tradução (5, -3, 2)
        E p ← ponto (-3, 4, 5)
      Então transformar * p = ponto (2, 1, 7)
```

Além disso, se você pegar o inverso de uma matriz de translação, obterá outra matriz de translação que move os pontos ao contrário. Adicione o seguinte teste ao seu conjunto para demonstrar isso.

**features / transformations.feature**

```
Cenário : Multiplicando pelo inverso de uma matriz de tradução
    Dado transformar ← tradução (5, -3, 2)
        E inv ← inverso (transformar)
        E p ← ponto (-3, 4, 5)
      Então inv * p = ponto (-8, 7, 3)
```

Agora vamos jogar uma chave inglesa nas coisas: multiplicar uma matriz de tradução por um *vetor* não deve mudar o vetor! Lembre-se de que um vetor é apenas uma seta. Movê-lo no espaço não muda a direção que aponta. Adicione o seguinte teste para mostrar que os vetores não são alterados pela tradução:

**features / transformations.feature**

```
Cenário : A tradução não afeta os vetores
    Dado transformar ← tradução (5, -3, 2)
        E v ← vetor (-3, 4, 5)
      Então transformar * v = v
```

Você pode se perguntar como vai fazer isso. Uma matriz que afeta pontos, mas não vetores? Pode realmente ser assim?

Reúna-se!

No *Tuplas, pontos e vetores* , você leu que a diferença entre um ponto e um vetor era apenas que um vetor tinha um 0 em seu C componente. É aqui que esse recurso paga dividendos. Acontece que a forma como uma matriz de tradução é construída faz com que um 0 no C componente de uma tupla fará com que a tradução seja ignorada.

Vamos dar uma olhada nesta matriz de tradução misteriosa (spoiler: não realmente misteriosa) e ver como ela está estruturada. Comece com uma matriz de identidade t, e então adicione o desejado x, y, e z valores para (respectivamente) $t_{03}$, $t_{13}$, e $T_{23}$ elementos, conforme mostrado na figura a seguir.

$$\text{tradução}(x, y, z) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

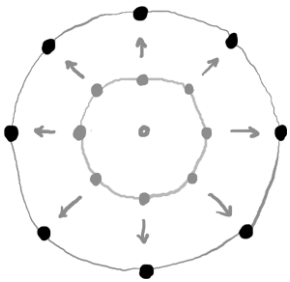
Você vai descobrir que, quando multiplicado por um vetor, o 0 no C faz com que esses valores de tradução desapareçam, como mágica. Com um ponto, porém, o 1 no C tem o efeito desejado e faz com que o ponto se mova.

Liso!

Então essa é a tradução. Faça esses testes passarem e veremos *escala* Próximo.

## Dimensionamento

Onde a tradução move um ponto adicionando a ele, *escala* move-o por multiplicação. Quando aplicada a um objeto centralizado na origem, esta transformação dimensiona todos os pontos no objeto, tornando-o efetivamente maior (se o valor da escala for maior que 1) ou menor (se o valor da escala for menor que 1), conforme mostrado no figura.



Você precisará de uma nova função, chamada escala (x, y, z), que retorna uma matriz de tradução 4x4. Adicione o seguinte teste para demonstrar como ele é usado para dimensionar um ponto.

**features / transformations.feature**

```
Cenário : Uma matriz de escala aplicada a um ponto
    Dado transformar ← escala (2, 3, 4)
        E p ← ponto (-4, 6, 8)
      Então transformar * p = ponto (-8, 18, 32)
```

Agora, ao contrário da tradução, o dimensionamento também se aplica aos vetores, alterando seu comprimento. Adicione o seguinte teste para mostrar como os vetores são afetados pela escala.

**features / transformations.feature**

```
Cenário : Uma matriz de escala aplicada a um vetor
    Dado transformar ← escala (2, 3, 4)
        E v ← vetor (-4, 6, 8)
      Então transformar * v = vetor (-8, 18, 32)
```

E, como você pode esperar, multiplicar uma tupla pelo inverso de uma matriz de escalonamento escalará a tupla da maneira oposta (diminuindo em vez de aumentar, ou vice-versa). Adicione o seguinte teste para mostrar que é assim.

**features / transformations.feature**

```
Cenário : Multiplicando pelo inverso de uma matriz de escala
    Dado transformar ← escala (2, 3, 4)
        E inv ← inverso (transformar)
        E v ← vetor (-4, 6, 8)
      Então inv * v = vetor (-2, 2, 2)
```

Para construir uma matriz de escalonamento, pegue uma matriz de identidade t e mudar os valores em $t_{00}$, $t_{11}$, e $T_{22}$ ser (respectivamente) o x, y, e z s • valores de calibração.

$$\text{dimensionamento}(x, y, z) = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Já que estamos no assunto de dimensionamento, vamos tomar um momento e discutir seu primo mais próximo: *reflexão.* A reflexão é uma transformação que leva um ponto e *reflete* - movendo-o para o outro lado de um eixo. Pode ser útil quando você tem um objeto em sua cena que deseja virar (ou espelhar) em alguma direção. Talvez o modelo esteja inclinado para o lado errado, voltado para a direção errada. Talvez seja um rosto que está olhando para a direita quando você deseja que olhe para a esquerda. Em vez de quebrar um modelador 3D e editar o modelo, você pode simplesmente refletir o modelo no eixo apropriado.

A reflexão é essencialmente a mesma coisa que dimensionar por um valor negativo. Implemente o seguinte teste, que mostra como um ponto pode ser refletido no x eixo escalando o x componente por - 1

**features / transformations.feature**

```
Cenário : A reflexão está aumentando em um valor negativo
    Dado transformar ← escala (-1, 1, 1)
        E p ← ponto (2, 3, 4)
      Então transformar * p = ponto (-2, 3, 4)
```

Assim, o ponto foi movido do lado positivo do x eixo, para o negativo.

Faça seus testes passarem e, em seguida, vamos prosseguir para a rotação.

## Rotação

Multiplicando uma tupla por um *rotação* matriz irá girar essa tupla em torno de um eixo. Isso pode ficar complicado se você estiver tentando girar em torno de uma linha arbitrária, então não vamos seguir esse caminho. Vamos apenas lidar com as rotações mais simples aqui - girando em torno do x, y, e z machados.

**Funções trigonométricas**

Matrizes de rotação dependem do *seno* e *cosseno* funções de trigonometria. Não se preocupe em dragar suas memórias de matemática do ensino médio. Verifique sua linguagem de implementação para um Matemática namespace, onde você geralmente encontrará as funções chamadas pecado e cos.

A rotação parecerá ser no sentido horário em torno do eixo correspondente quando vista ao longo desse eixo, em direção à extremidade negativa. Então, se você estiver girando em torno do x eixo, ele irá girar conforme ilustrado na figura a seguir.

Outra maneira de descrever isso é dizer que as rotações em seu ray tracer obedecerão ao *regra da mão esquerda,* que remonta a Coordenadas para canhotos vs. destros, na página 3 : se você apontar o polegar de sua mão esquerda na direção do eixo de rotação, a rotação em si seguirá a direção de seus dedos restantes conforme você os enrola em direção à palma da sua mão.

Cada um dos três eixos requer uma matriz diferente para implementar a rotação, portanto, examinaremos cada um deles. Os ângulos serão dados em radianos, portanto, se sua biblioteca de matemática preferir outras unidades (como graus), você precisará se adaptar de acordo.

**Joe pergunta:**

### O que são radianos?

Um círculo completo (360 graus) consiste em 2 π radianos, o que significa um semicírculo (180 graus) é π radianos e um quarto de círculo (90 graus) é π/2 radianos. Se você não está acostumado a pensar em termos de radianos, pode ser útil escrever uma função para converter eles a partir de graus. A fórmula é semelhante a esta:

$$\text{radianos ( deg)} = \frac{\text{deg}}{180}\pi$$

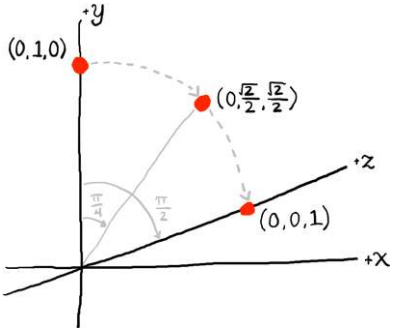
## Rotação em torno do eixo X

Esta primeira matriz de rotação gira uma tupla algum número de radianos em torno do x eixo, e será criado pela introdução de um novo rotação_x (radianos) função. Prove que funciona adicionando o seguinte teste, que mostra a rotação de um ponto em torno do x eixo.

**features / transformations.feature**

```
Cenário : Girando um ponto em torno do eixo x
    Dado p ← ponto (0, 1, 0)
        E half_quarter ← rotação_x ( π / 4)
        E full_quarter ← rotação_x ( π / 2)
    Então half_quarter * p = ponto (0, √ 2/2, √ 2/2)
        E full_quarter * p = ponto (0, 0, 1)
```

Visualmente, o teste realiza as duas rotações a seguir:




Em seguida, adicione outro teste mostrando que o inverso dessa matriz de rotação simplesmente gira na direção oposta.

**features / transformations.feature**

**Cenário :** O inverso de uma rotação x gira na direção oposta

**Dado** p ← ponto (0, 1, 0)

**E** half_quarter ← rotação_x ( π / 4)

**E** inv ← inverso (half_quarter)

**Então** inv * p = ponto (0, √ 2/2, - √ 2/2)

A matriz de transformação para rotação r radianos ao redor do x eixo é construído assim:

rotação x ( r) = •

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 porque r - pecado r 0 | | | |
| 0 pecado r | | porque r | 0 |
| 0 | 0 | 0 | 1 |

Muito agradável. Agora, para o próximo eixo.

## Rotação em torno do eixo Y

O y a rotação do eixo funciona exatamente como o x rotação do eixo, mudando apenas o eixo. Adicione o seguinte teste para demonstrar a diferença.

**features / transformations.feature**

**Cenário :** Girando um ponto em torno do eixo y

**Dado** p ← ponto (0, 0, 1)

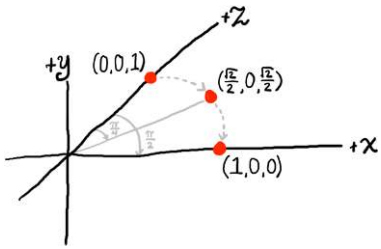
**E** half_quarter ← rotação_y ( π / 4)

**E** full_quarter ← rotação_y ( π / 2)

**Então** half_quarter * p = ponto ( √ 2/2, 0, √ 2/2)

**E** full_quarter * p = ponto (1, 0, 0)

Mais uma vez, visualmente, essa rotação se parece com isto:




A matriz de transformação para rotação r radianos ao redor do y eixo é construído assim:

rotação y ( r) = •

| | | | |
|---|---|---|---|
| porque r | 0 pecado r | | 0 |
| 0 | 1 | 0 | 0 |
| - pecado r 0 porque r 0 | | | |
| 0 | 0 | 0 | 1 |

Só então. Falta mais um eixo!

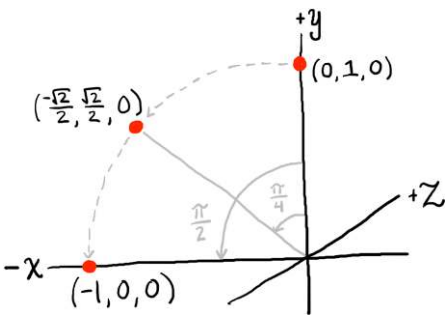
## Rotação em torno do eixo Z

E por último, mas não menos importante: o z rotação do eixo. Mostre que funciona exatamente como as outras rotações, implementando o seguinte teste.

**features / transformations.feature**

```
Cenário : Girando um ponto em torno do eixo z
    Dado p ← ponto (0, 1, 0)
        E half_quarter ← rotação_z ( π / 4)
        E full_quarter ← rotação_z ( π / 2)
    Então half_quarter * p = ponto (- √ 2/2, √ 2/2, 0)
        E full_quarter * p = ponto (-1, 0, 0)
```

E aqui está a visualização correspondente:



Esta rotação pode parecer para trás, mas quebre a regra da mão esquerda e verifique.
Aponte seu polegar esquerdo ao longo do positivo
z eixo e, em seguida, curve os dedos. Eles se curvam em direção ao negativo
x eixo, conforme ilustrado!

Finalmente, a matriz de transformação • em si é este:

rotação z ( r) = • pecado r

| | | |
|---|---|---|
| porque r - s | no r | 0 0 |
| | porque r | 0 0 • |
| 0 | 0 | 1 0 • |
| 0 | 0 | 0 1 |

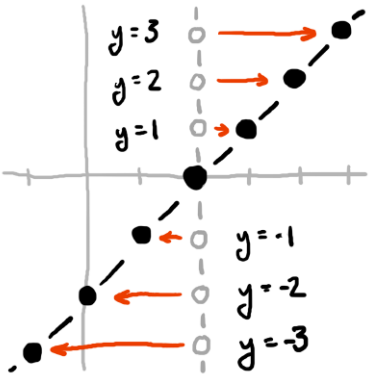
Isso se encarrega de girar um ponto ou vetor em torno de qualquer um de nossos três eixos primários. Faça esses testes passarem e siga em frente. Vamos ver mais uma transformação.

## Tosquia

UMA *tosquia (* ou *enviesado)* a transformação tem o efeito de tornar as linhas retas inclinadas. É provavelmente a transformação mais (visualmente) complexa que implementaremos, embora a implementação não seja mais complicada do que qualquer uma das outras.

Quando aplicada a uma tupla, uma transformação de cisalhamento altera cada componente da tupla em proporção aos outros dois componentes. Então o x mudanças de componentes em proporção a y e z, y mudanças em proporção a x e z, e z
mudanças em proporção a x e y.

A ilustração a seguir mostra como isso funciona em duas dimensões. Especificamente, observe como a mesma transformação afeta de forma diferente cada ponto em x Enquanto o y
alterações de componentes.



Isso é o que "mudar em proporção" significa: quanto mais longe y coordenada é de zero, quanto mais o x mudanças de valor.

Em três dimensões, cada componente pode ser afetado por qualquer um dos outros dois componentes, portanto, há um total de seis parâmetros que podem ser usados para definir a transformação de cisalhamento:

- x em proporção para y
- x em proporção para z
- y em proporção para x
- y em proporção para z
- z em proporção para x
- z em proporção para y

Escreva os seguintes testes, demonstrando como um ponto é afetado por cada um desses parâmetros. Em cada um, observe como a coordenada que está sendo movida se move pela quantidade da outra coordenada. Por exemplo, neste primeiro teste x é inicialmente

2, mas se movendo x em proporção para y adiciona 1 vezes y ( ou 3) para x (2) e produz um novo x de 5

**features / transformations.feature**

```
Cenário : Uma transformação de cisalhamento move x em proporção a y
    Dado transformar ← tosquia (1, 0, 0, 0, 0, 0)
        E p ← ponto (2, 3, 4)
    Então transformar * p = ponto (5, 3, 4)
```

Os testes restantes funcionam de maneira semelhante, adicionando os dois componentes para obter o novo valor do componente.

**features / transformations.feature**

```
Cenário : Uma transformação de cisalhamento move x em proporção a z
    Dado transformar ← tosquia (0, 1, 0, 0, 0, 0)
        E p ← ponto (2, 3, 4)
    Então transformar * p = ponto (6, 3, 4)

Cenário : Uma transformação de cisalhamento move y em proporção a x
    Dado transformar ← tosquia (0, 0, 1, 0, 0, 0)
        E p ← ponto (2, 3, 4)
    Então transformar * p = ponto (2, 5, 4)

Cenário : Uma transformação de cisalhamento move y em proporção a z
    Dado transformar ← tosquia (0, 0, 0, 1, 0, 0)
        E p ← ponto (2, 3, 4)
    Então transformar * p = ponto (2, 7, 4)

Cenário : Uma transformação de cisalhamento move z em proporção a x
    Dado transformar ← tosquia (0, 0, 0, 0, 1, 0)
        E p ← ponto (2, 3, 4)
    Então transformar * p = ponto (2, 3, 6)
```

**Cenário :** Uma transformação de cisalhamento move z em proporção a y

**Dado** transformar ← tosquia (0, 0, 0, 0, 0, 1)

**E** p ← ponto (2, 3, 4)

**Então** transformar * p = ponto (2, 3, 7)

A matriz de transformação para uma transformação de cisalhamento é dada na figura a seguir, onde (por exemplo) $x_y$ meios " x movido em proporção a y, "E repre-
envia a quantidade pela qual multiplicar y antes de • está adicionando a x. •

$$\text{tosquia}(x_y, x_z, y_x, y_z, z_x, z_y) = \begin{bmatrix} 1 & x_y & x_z & 0 \\ y_x & 1 & y_z & 0 \\ z_x & z_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Essa é a última das matrizes de transformação que abordaremos aqui. Reserve um tempo agora para verificar se todos os testes estão passando antes de prosseguir. Quando estiver pronto, vamos falar sobre como você pode combinar essas matrizes para criar transformações mais complexas.

## Transformações de encadeamento

Como você viu, você pode criar matrizes de transformação para traduzir, dimensionar, girar e inclinar. Mas e se você quiser fazer mais de um de cada vez?

É uma expectativa totalmente razoável. Digamos que você (eventualmente) renderize um bule de chá. O modelo que você está renderizando está na origem e é pequeno em relação ao resto da cena. O modelo também é inclinado para o lado. Você gostaria de girá-lo para que fique com o lado certo para cima, dimensionar para um tamanho razoável e, em seguida, traduzi-lo para que fique sobre uma mesa, em vez do chão.

Você pode aplicar cada transformação em sequência, assim:

```
# gire o bule para que fique com o lado direito para cima
UMA ← rotação_x ( π / 2) bule ← Um bule
de chá

# a seguir, aumente o bule 5x maior
B ← escala (5, 5, 5) bule ← B * bule


# finalmente, mova o bule para uma mesa
C ← tradução (10, 5, 7) bule ← Bule C *
```

Mas isso é exatamente o mesmo:

```
UMA ← rotação_x ( π / 2) B ← escala (5, 5, 5) C ← tradução
(10, 5, 7)


bule de chá ← C * (B * (bule A *))
```

Ou, uma vez que a multiplicação da matriz é associativa:

```
bule de chá ← ( C * B * A) * bule
```

Observe que a ordem das multiplicações é importante! A multiplicação da matriz é associativa, mas *não* comutativo. Quando se trata de matrizes, A × B não é garantido que seja o mesmo que B × A.

Portanto, se você quiser uma única matriz que gire, e depois dimensione e depois traduza, pode multiplicar a matriz de translação pela matriz de escala e, a seguir, pela matriz de rotação. Ou seja, você deve concatenar as transformações em *ordem reversa* para aplicá-los na ordem que você deseja! Adicione os seguintes testes para demonstrar este resultado (particularmente contra-intuitivo).

**features / transformations.feature**

```
Cenário :       As transformações individuais são aplicadas em sequência
    Dado p      ←   ponto (1, 0, 1)
        E UMA   ←   rotação_x ( π / 2)
        E B     ←   escala (5, 5, 5)
        E C     ←   tradução (10, 5, 7)
    # Aplique   rotação primeiro
    Quando p2   ←   A * p
    Então p2    = ponto (1, -1, 0)
    # em seguida, aplique escala
    Quando p3 ← B * p2
    Então p3 = ponto (5, -5, 0)
    # em seguida, aplique a tradução
    Quando p4 ← C * p3
    Então p4 = ponto (15, 0, 7)

Cenário : As transformações encadeadas devem ser aplicadas na ordem reversa
    Dado p ← ponto (1, 0, 1)
        E UMA ← rotação_x ( π / 2)
        E B ← escala (5, 5, 5)
        E C ← tradução (10, 5, 7)
    Quando T ← C * B * A
    Então T * p = ponto (15, 0, 7)
```

Impressionante! Agora você tem vetores e pontos e transformações de matriz. Esta é uma base fantástica para o resto do seu ray tracer! Vamos encontrar algo para fazer com essas peças antes de prosseguir.

## Juntar as peças

Aqui está um programa para você escrever. Imagine um relógio analógico. Existem (normalmente) doze posições ao redor da borda, representando as horas. Entendi? OK. Seu

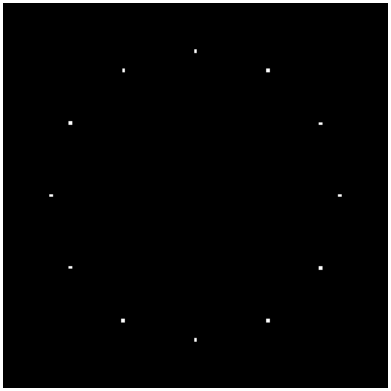
**APIs fluentes**

Dependendo de sua linguagem de implementação, você pode apresentar uma interface mais intuitiva para concatenar matrizes de transformação. UMA *API fluente,* por exemplo, poderia permitir que você declarasse suas transformações em uma ordem natural como esta:

```
transformar ← identidade().
                rotate_x ( π / 2).
                escala (5, 5, 5).
                traduzir (10, 5, 7)
```

A chamada para identidade() retorna a matriz de identidade, e rotate_x (π / 2) é então invocado nele. Isso multiplica a matriz de rotação correspondente pelo chamador, "rotação" vezes "identidade", invertendo efetivamente a ordem das operações. Cada chamada subsequente nesta cadeia multiplica sua matriz pelo resultado da chamada anterior, eventualmente virando toda a cadeia "do avesso".

O desafio é escrever um programa que use uma matriz de rotação para calcular as posições dessas horas no mostrador do relógio e desenhar um pixel em uma tela para cada uma delas. O resultado deve ser mais ou menos assim:



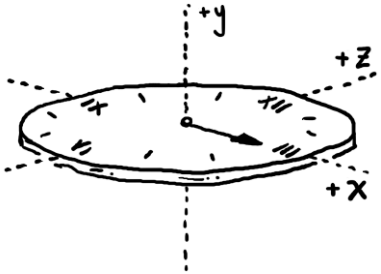
Aqui estão quatro dicas para você começar. (Sinta-se à vontade para parar de ler agora se quiser ver se consegue fazer funcionar sem nenhuma dica!)

## Dica # 1

Primeiro, suponha que o relógio esteja centralizado na origem, ponto (0,0,0). Deixe a origem estar no meio de sua tela.

## Dica # 2

A seguir, escolha um eixo para orientar o relógio. Se, por exemplo, é orientado ao longo do y eixo e você está olhando para ele de frente, então você está olhando para a extremidade negativa do y eixo. A figura a seguir mostra essa orientação.




Isso significa que doze horas está no + z eixo em ponto (0,0,1), e três horas é no + x eixo em ponto (1,0,0).

## Dica # 3

Agora, gire o ponto das doze horas em torno do y eixo para encontrar a outra hora posições. Há 2 π radianos em um círculo, então cada hora é girada 2 π/12 ( ou π/6) radianos. Em pseudocódigo, então, seria algo assim:

```
# calcula a rotação do eixo y para a hora # 3
r ← rotação_y (3 * π / 6)

# fornecido: posição das doze horas
doze ← ponto (0,0,1)

# calcula a posição de três horas girando doze horas
três ← r * doze
```

Neste caso, você deve descobrir que três horas são em ponto (1,0,0).

## Dica # 4

Decida o tamanho do relógio a ser desenhado em sua tela. Por exemplo, se sua tela é quadrada, você pode deixar o raio do relógio ser 3/8 a largura da tela.

Para cada ponto que você computar, multiplique o x e z componentes por esse raio e, em seguida, mova-os para o centro da tela, adicionando as coordenadas do ponto central. Deixei x seja o x coordenada do pixel, e z

seja o y coordenada.

Não se esqueça de salvar sua tela como um arquivo PPM quando terminar!

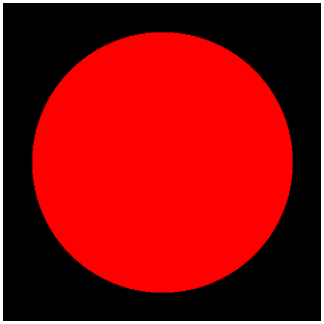
Assim que você tiver isso definido, siga em frente. É hora de começar a cruzar raios e esferas!

CAPÍTULO 5

# Intersecções Ray-Sphere

Ótimas notícias! Você acabou de fazer o trabalho básico e agora pode começar na carne de um traçador de raio real. De agora em diante, cada capítulo culminará em algo concreto, algo *visual,* que irá adicionar ao seu estoque crescente de colírio para os olhos.

Para este capítulo, a parte visual não será particularmente impressionante. Será apenas um círculo preenchido humilde desenhado em sua tela, como este:



Primitivo? Sem dúvida! Mas você vai desenhá-lo exercitando o músculo mais básico do corpo de um traçador de raios: a projeção de raios.

*Ray casting* é o processo de criação de um *raio,* ou linha e encontrando as interseções desse raio com os objetos em uma cena. Abordaremos tudo isso neste capítulo, usando material dos capítulos anteriores à medida que avançamos.

Vamos fazer isso!

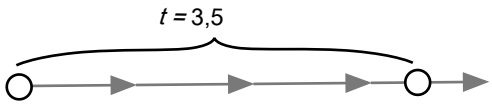
## Criação de raios

Cada raio criado pelo seu ray tracer terá um ponto de partida chamado de *origem,*
e um vetor chamado de *direção* que diz para onde aponta. Escreva o seguinte teste, mostrando como você cria um raio e quais devem ser seus atributos primários:

**features / rays.feature**

```
Cenário : Criação e consulta de um raio
    Dado origem ← ponto (1, 2, 3)
        E direção ← vetor (4, 5, 6)
    Quando r ← raio (origem, direção)
    Então r.origin = origem
        E r.direction = direction
```

Armado com a origem e direção de um raio, você pode encontrar pontos que estão a qualquer distância t ao longo do raio. Por que t? Culpe os matemáticos! Ele significa *Tempo,* o que só faz sentido quando você pensa no raio *vetor de direção* como é *Rapidez.* Por exemplo, se o raio se move uma unidade a cada segundo, então a figura a seguir de Multiplicação e divisão escalar, na página 7 , mostra a distância que o raio viaja em 3,5 segundos.




Faça o seguinte teste, que apresenta uma nova função chamada posição (raio, t). Esta função deve calcular o ponto na distância dada t ao longo do raio.

**features / rays.feature**

```
Cenário : Calculando um ponto à distância
    Dado r ← raio ( ponto (2, 3, 4), vetor (1, 0, 0))
    Então posição (r, 0) = ponto (2, 3, 4)
        E posição (r, 1) = ponto (3, 3, 4)
        E posição (r, -1) = ponto (1, 3, 4)
        E posição (r, 2,5) = ponto (4,5, 3, 4)
```

Para encontrar a posição, você multiplica a direção do raio por t para encontrar a distância total percorrida e, em seguida, adicione-a à origem do raio. Em pseudocódigo, é assim:

```
função posição (raio, t)
    Retorna ray.origin + ray.direction * t
função final
```

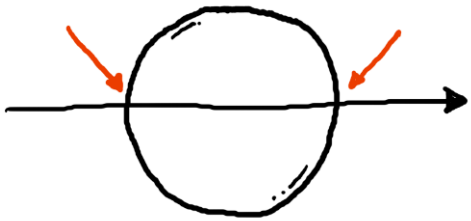
Você fará bom uso disso em Capítulo 6, Luz e Sombreamento, na página 75 , quando você começa a transformar cruzamentos em informações de superfície reais. Faz parte do processo de computação de sombreamento realista para suas cenas.

Certifique-se de que seus testes estão passando antes de prosseguir. Na próxima seção, veremos a interseção desses raios com esferas.

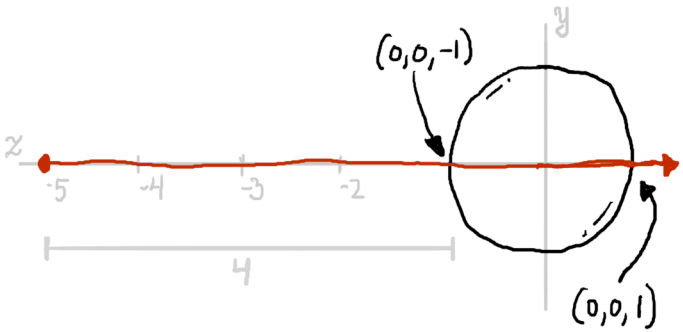
## Interseção de raios com esferas

Vamos tornar a sua vida o mais simples possível, assumindo que a origem de cada esfera (seu ponto central) está situada na origem do mundo (que é ponto (0, 0, 0)). Também presumiremos que esses são todos *esferas unitárias,* com raios de 1.

Se você fosse lançar um raio através do centro de uma dessas esferas, você veria o raio se cruzar em dois lugares, assim:



Mais especificamente, se o raio se origina em ( 0, 0, -5), e passa diretamente pela origem, deve interceptar a esfera em ( 0, 0, -1) e ( 0, 0, 1), 4 e 6 unidades (respectivamente) de distância da origem do raio, como mostra a figura a seguir.




Adicione o seguinte teste para demonstrar isso. Ele apresenta duas novas funções:
esfera(), que retorna um novo objeto de esfera, e cruzar (esfera, raio), que retorna a coleção de t valores onde o raio cruza a esfera.

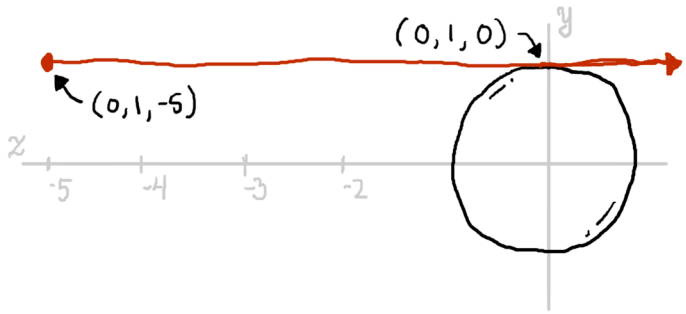
**features / spheres.feature**

```
Cenário : Um raio cruza uma esfera em dois pontos
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 2
        E xs [0] = 4,0
        E xs [1] = 6,0
```

O esfera() A função deve retornar um valor exclusivo sempre que for chamada. Dependendo da sua linguagem de programação, você pode precisar passar alguma coisa

único (um inteiro ou uma string) para a função como a nova esfera eu ia. Você adicionará alguns atributos à esfera posteriormente neste capítulo (quando começar a incorporar transformações de matriz), mas por enquanto ela não tem dados associados. Apenas certifique-se de que não haja duas invocações de esfera() retorna o mesmo valor.

Agora, se você mover o ponto inicial do seu raio 1 unidade no positivo y direção, o raio será *tangente* para a esfera. Ele se cruzará em um ponto, apenas olhando para fora da borda, assim:




Implemente o seguinte teste, que corresponde a este cenário. Deve instanciar uma unidade de raio 1 mais distante no y direção e interseccione-a com a mesma esfera unitária. Mesmo que ele realmente se cruze em apenas um único ponto, para simplificar, você terá seu código de retorno *dois* intersecções, com o mesmo ponto em cada uma. (Isso ajudará mais tarde na determinação de sobreposições de objetos, em

Capítulo 16, Geometria Sólida Construtiva (CSG), na página 227 .) Afirme que ambas as interseções estão no mesmo ponto.

**features / spheres.feature**

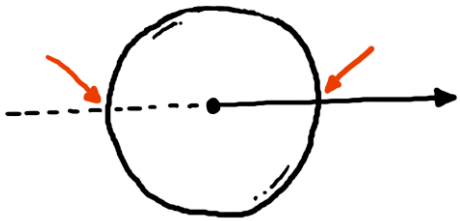
```
Cenário : Um raio cruza uma esfera em uma tangente
    Dado r ← raio ( ponto (0, 1, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 2
        E xs [0] = 5,0
        E xs [1] = 5,0
```

Agora mova o ponto de partida do seu raio um pouco mais ao longo do positivo y direção. O raio deve perder totalmente a esfera, passando acima da esfera e não a cruzando de forma alguma. Escreva o seguinte teste para mostrar que isso é verdade.

**features / spheres.feature**

```
Cenário : Um raio perde uma esfera
    Dado r ← raio ( ponto (0, 2, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 0
```

Antes de fazer esses testes passarem, existem alguns casos extremos a serem considerados. Por exemplo, o que acontece se o seu raio se originar dentro da esfera? Bem, deve haver uma interseção na frente do raio e outra atrás dele, como ilustra a figura a seguir.

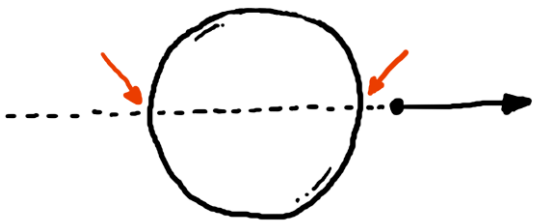


Sim, o raio realmente se estende *atras do* o ponto de partida, mas não vamos nos distrair com as definições! Vá em frente e escreva o seguinte teste, mostrando que quando o raio começa no centro de uma esfera, a primeira interseção está atrás da origem do raio e a segunda está na frente dela.

**features / spheres.feature**

```
Cenário : Um raio se origina dentro de uma esfera
    Dado r ← raio ( ponto (0, 0, 0), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 2
        E xs [0] = -1,0
        E xs [1] = 1,0
```

Por último, se a esfera está completamente atrás do raio, você ainda deve ver duas interseções - ambas com um negativo t valor. A figura a seguir mostra a aparência disso.



O teste a seguir mostra que é assim, com ambas as interseções ocorrendo atrás da origem do raio.

```
features / spheres.feature
Cenário : Uma esfera está atrás de um raio
    Dado r ← raio ( ponto (0, 0, 5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 2
        E xs [0] = -6,0
        E xs [1] = -4,0
```

Vamos dar uma olhada agora no que precisa acontecer para que esses testes sejam aprovados. Para calcular a interseção de um raio e uma esfera, você precisará das rotinas que implementou até este ponto, incluindo aritmética de tupla, o produto escalar e até mesmo (mais adiante neste capítulo) inversão e transformações de matriz. É bom que você já os tenha funcionando, não é?

A matemática por trás da intersecção de um raio e uma esfera é realmente muito elegante, mas por uma questão de brevidade, vamos pular as derivações e ir direto para a implementação. Se você realmente deseja se aprofundar na matemática, encontrará muitos recursos online. Confira o artigo “Intersecção linha-esfera” na Wikipedia, [1] o tutorial “RaySphere Intersection” no Lighthouse3d, [2] ou o post “Ray-Sphere Intersection” da série Scratchapixel sobre “A Minimal Ray-Tracer. [3]

”

Comece o algoritmo calculando o *discriminante* - um número que indica se o raio cruza a esfera. Em pseudocódigo, os cálculos são assim:

```
# o vetor do centro da esfera até a origem do raio
# lembre-se: a esfera está centrada na origem do mundo
esfera_para_raia ← ray.origin - ponto (0, 0, 0)

uma ← ponto (ray.direction, ray.direction) b ← 2 * ponto (raio.direcionamento,
esfera_para_raio) c ← ponto (esfera_para_raia, esfera_para_raia) - 1


discriminante ← b² - 4 * a * c
```

Que discriminante valor é a chave. Se for negativo, então o raio falha e nenhuma interseção ocorre entre a esfera e o raio.

```
E se discriminante <0 então
    Retorna ()
fim se
```

1 en.wikipedia.org/wiki/Line–sphere_intersection

2 www.lighthouse3d.com/tutorials/maths/ray-sphere-intersection

3 www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphereintersection

Caso contrário, você verá um (para raios que atingem a esfera em uma tangente perfeita) ou duas interseções, mas sua função sempre deve retornar dois em ambos os casos. Para o caso da tangente, ambas as interseções terão o mesmo t valor, conforme mencionado anteriormente. Além disso, certifique-se de que as interseções sejam retornadas em ordem crescente para facilitar a determinação posterior de quais interseções são significativas.

```
t1 ← (- b - √ ( discriminante)) / (2 * a) t2 ← (- b + √ ( discriminante)) / (2 *
a)

Retorna ( t1, t2)
```

Neste ponto, todos os seus testes devem estar passando. Yay! Dê palmadinhas nas suas próprias costas e exulte pelo facto de ter implementado o coração de um verdadeiro ray tracer!

No entanto, isso é apenas parte da solução. Seu ray tracer eventualmente precisará saber mais do que o t valores em cada intersecção. A seguir, vamos ver como controlar essas informações adicionais.

## Rastreando interseções

Atualmente, o seu cruzar função retorna um conjunto de t valores, mas imagine por um momento uma cena lindamente complexa, cheia de esferas, cubos, cilindros, cones e dezenas de combinações criativas. Você lança seu raio naquela cena e obtém de volta um punhado duplo de cruzamentos. Agora você sabe onde ocorreram os cruzamentos (graças ao t valores), mas você não tem ideia de como desenhá-los. Qual objeto foi cruzado naquele ponto? Que cor é essa? Quais são suas propriedades materiais? Deve haver uma reflexão ou não? Você simplesmente não sabe.

Com a adição de mais uma propriedade, você terá a base do que precisa para responder a essas perguntas. Você vai criar uma nova estrutura de dados, chamada de *interseção,* que irá (por enquanto) agregar duas coisas:

1. O t valor da interseção, e
2. O objeto que foi cruzado.

Você adicionará propriedades adicionais em capítulos posteriores, mas elas serão suficientes por enquanto. Vá em frente e adicione o seguinte teste para mostrar como criar uma interseção e como suas propriedades são acessadas.

**features / intersections.feature**

```
Cenário : Uma interseção encapsula t e objeto
    Dado s ← esfera ()
    Quando eu ← interseção (3,5, s)
    Então it = 3,5
        E i.object = s
```

Você também precisará de uma maneira de agregar esses objetos de interseção para que possa trabalhar com várias interseções de uma vez. (Considere sua rotina de interseção de esferas, que pode retornar zero, uma ou duas interseções.) Escreva o seguinte teste, que apresenta uma nova função chamada intersecções (i1, i2, ...). Isso deve retornar uma nova coleção dos objetos de interseção fornecidos.

**features / intersections.feature**

```
Cenário : Agregando cruzamentos
    Dado s ← esfera ()
        E i1 ← interseção (1, s)
        E i2 ← interseção (2, s)
    Quando xs ← cruzamentos (i1, i2)
    Então xs.count = 2
        E xs [0] .t = 1
        E xs [1] .t = 2
```

Esta lista de interseções pode ser apenas uma matriz primitiva em sua linguagem de implementação, mas observe que você adicionará uma função em breve (em Identificando sucessos, na página 64 ) que opera nessas listas de cruzamentos.

Agora é hora de quebrar alguns códigos! Modifique seus testes existentes para que eles assumam seu cruzar função retorna uma lista desses registros de interseção, em vez de t valores. Além disso, adicione o seguinte teste, que mostrará que o

objeto propriedade está sendo definida por se cruzam.

**features / spheres.feature**

```
Cenário : Intersect define o objeto na interseção
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando xs ← cruzar (s, r)
    Então xs.count = 2
        E xs [0] .object = s
        E xs [1] .objeto = s
```

Faça seus testes passarem novamente, modificando seu cruzar função para criar um registro para cada interseção, em vez de retornar o t valores diretamente. Tudo que você precisa agora é ser capaz de decidir com qual dessas interseções você realmente se importa, o que apresenta o sucesso.

## Identificação de sucessos

Ao renderizar sua cena, você precisará ser capaz de identificar qual de todas as interseções é realmente visível da origem do raio. Alguns podem estar atrás do raio, e outros podem estar escondidos atrás (ou *ocluído por)* outros objetos. Para fins de discussão, vamos chamar a interseção visível de *acertar*. Este é realmente o único cruzamento que importa para a maioria das coisas.

O hit nunca estará atrás da origem do raio, uma vez que está efetivamente atrás da câmera, então você pode ignorar todas as interseções com negativo t valores ao determinar o acerto. Na verdade, o hit será sempre a interseção com o menor valor não negativo t valor.

**Joe pergunta:**

## Por que devo manter todos os cruzamentos?

Você acabou de ler que pode ignorar todos os cruzamentos com negativos t valores ao determinar o acerto. Então, por que mantê-los por perto? Não seria mais fácil simplesmente não devolvê-los do intersectar () função em primeiro lugar?

Certamente. É uma otimização justa, até chegar a Capítulo 11, Reflexão e Refração, na página 141 . Nesse ponto, esses cruzamentos aparentemente irrelevantes de repente se tornam importantes! Eles serão usados para ajudar a determinar quais formas contêm outras formas. Isso também será útil em Capítulo 16, Geometria Sólida Construtiva (CSG), na página 227 , para informar como renderizar coleções de objetos relacionados por operações booleanas.

Então, agarre-se àqueles negativos t valores por enquanto! Seu futuro eu vai agradecer.

Escreva os seguintes testes, que apresentam uma função chamada acertar (cruzamentos).
Esta função retorna a ocorrência de uma coleção de registros de interseção. Escrever esses testes mostrará como acertar deve se comportar em algumas situações diferentes.

**features / intersections.feature**

```
Cenário : O acerto, quando todas as interseções têm t positivo
    Dado s ← esfera ()
        E i1 ← interseção (1, s)
        E i2 ← interseção (2, s)
        E xs ← cruzamentos (i2, i1)
    Quando eu ← acertar (xs)
    Então i = i1

Cenário : O acerto, quando alguns cruzamentos têm t negativo
    Dado s ← esfera ()
        E i1 ← interseção (-1, s)
        E i2 ← interseção (1, s)
        E xs ← cruzamentos (i2, i1)
    Quando eu ← acertar (xs)
    Então i = i2

Cenário : O acerto, quando todas as interseções têm t negativo
    Dado s ← esfera ()
        E i1 ← interseção (-2, s)
        E i2 ← interseção (-1, s)
        E xs ← cruzamentos (i2, i1)
    Quando eu ← acertar (xs)
```

```
    Então eu não sou nada

Cenário : O acerto é sempre a interseção não negativa mais baixa
    Dado s ← esfera ()
    E i1 ← interseção (5, s)
    E i2 ← interseção (7, s)
    E i3 ← interseção (-3, s)
    E i4 ← interseção (2, s)
    E xs ← cruzamentos (i1, i2, i3, i4)
Quando eu ← acertar (xs)
Então i = i4
```

Não deixe que o último teste o atrapalhe! As interseções são dadas intencionalmente em ordem aleatória; depende do seu cruzamentos () função para manter uma lista ordenada ou, pelo menos, ordenar a lista a pedido. Isso será importante no futuro, quando você tiver cenas mais complicadas com vários objetos. Não será viável para cada forma preservar manualmente a ordem de classificação dessa lista de interseção.

Isso completa seu conjunto de funcionalidades relacionadas a interseções. Faça todos esses testes passarem e, em seguida, vamos dar uma olhada em como mover, redimensionar, girar e deformar suas esferas.

## Transformando raios e esferas

Uma esfera unitária fixada na origem é (na melhor das hipóteses) pouco útil. Você certamente não poderia ter mais de um, o que torna difícil fazer qualquer tipo de cena com eles. O que você deseja é ser capaz de transformar essa esfera - dimensione-a para maior ou menor, mova-a e talvez (se um lado tiver uma textura diferente) gire-a um pouco.

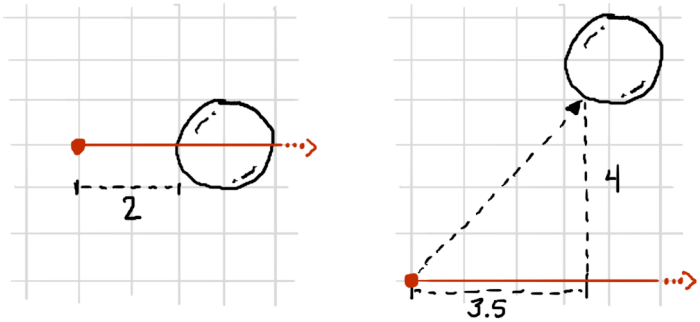
Se você permitir o movimento da esfera, no entanto, seu belo algoritmo de interseção de raio-esfera tem que mudar, porque ele assume que a esfera está sempre na origem e sempre tem um raio de 1 Seria ótimo se você pudesse manter essa suposição, enquanto ainda permite que as esferas sejam redimensionadas e reposicionadas. Isso tornaria sua implementação muito mais limpa e simples.

Bem, vamos considerar isso. Vocês *dizer* você quer mover a esfera, mas o que você *mesmo* quer, fundamentalmente, que a distância entre a esfera e a origem do raio aumente ou diminua, ou a relação entre a direção do raio e a posição da esfera mude, como as duas imagens mostrado na página 67 .

.....

.....

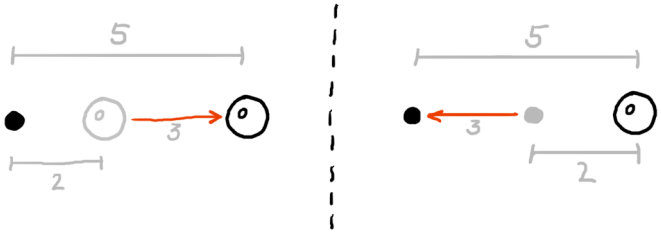
No da esquerda, a origem do raio e a esfera são separadas por 2 unidades. No da direita, eles se afastaram ainda mais. Mas pense nisso por um momento: *a esfera se moveu ou o raio?* Isso importa mesmo? Independentemente




de qual deles mudou, a distância entre eles aumentou, certo? Então, aqui está uma ideia maluca. E se, em vez de mover a esfera, *você move o raio?*

(Eu sei. É muito selvagem.)

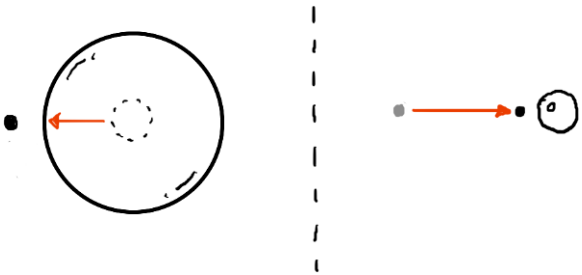
Quer traduzir sua esfera para longe do raio? Isso é o mesmo que traduzir o *raio* longe de *esfera,* na direção oposta, como mostram as figuras a seguir.




Na figura à esquerda, a esfera é afastada do ponto (talvez a origem de um raio). À direita, o ponto é movido para longe da esfera. Em ambos os casos, a esfera e o ponto terminam separados por 5 unidades.

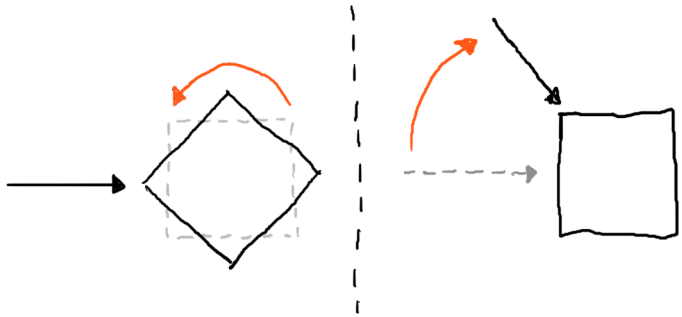
Mas e quanto ao escalonamento? E se você quiser tornar sua esfera maior? Acontece que isso é exatamente o mesmo que *diminuindo a distância* entre o raio e a esfera. É uma relação inversa. Você escala o raio pelo *inverso*

de como você queria escalar a esfera, como na figura a seguir:



Ok, mas e a rotação? Certamente não pode ser tão simples para algo como rotação? Oh, mas pode! Considere a seguinte figura. À esquerda, você vê que girar um objeto expõe um lado diferente desse objeto ao raio. À direita, o mesmo resultado é obtido girando o *raio* em volta do

*objeto.*



Se você quiser girar sua esfera, você gira o raio pelo *inverso* da rotação que você deseja aplicar à esfera.

Em outras palavras: qualquer transformação que você deseja aplicar à esfera, aplique o *inverso* dessa transformação para o raio, ao invés. Louco, certo? Mas funciona!

**Espaço Mundial vs. Espaço Objeto**

Outra maneira de pensar sobre matrizes de transformação é pensar nelas como pontos de conversão entre dois sistemas de coordenadas diferentes. No nível da cena, tudo está em *espaço do mundo* coordenadas, em relação ao mundo em geral. Mas no nível do objeto, tudo está em *espaço do objeto* coordenadas, em relação ao próprio objeto.

Multiplicar um ponto no espaço do objeto por uma matriz de transformação converte esse ponto no espaço do mundo - dimensionando-o, traduzindo-o, girando-o ou o que quer que seja. Multiplicando um ponto no espaço mundial pelo *inverso* da matriz de transformação converte esse ponto de volta para o espaço do objeto.

Quer cruzar um raio no espaço do mundo com uma esfera no espaço do objeto? Basta converter a origem e a direção do raio para o mesmo espaço de objeto e você estará dourado.

Então, primeiro, certifique-se de que seu raio seja transformável. Adicione os seguintes testes ao seu pacote, introduzindo um transformar (raio, matriz) função que aplica a matriz de transformação dada ao raio dado, e retorna um novo raio com origem e direção transformadas. Certifique-se de que ele retorna um novo raio, ao invés de modificar o raio no lugar! Você precisa manter o raio original não transformado, para que possa usá-lo para calcular localizações no espaço mundial posteriormente.

**features / rays.feature**

```
Cenário : Traduzindo um raio
    Dado r ← raio ( ponto (1, 2, 3), vetor (0, 1, 0))
        E m ← tradução (3, 4, 5)
    Quando r2 ← transformar (r, m)
    Então r2.origin = ponto (4, 6, 8)
        E r2.direction = vetor (0, 1, 0)

Cenário : Escalando um raio
    Dado r ← raio ( ponto (1, 2, 3), vetor (0, 1, 0))
        E m ← escala (2, 3, 4)
    Quando r2 ← transformar (r, m)
    Então r2.origin = ponto (2, 6, 12)
        E r2.direction = vetor (0, 3, 0)
```

Observe como, no segundo teste, o vetor de direção do raio é deixado não normalizado. Isso é intencional e importante! Transformar um raio tem o efeito de (potencialmente) esticar ou encolher seu vetor de direção. Você tem que deixar esse vetor com seu novo comprimento, para que quando o t valor é eventualmente calculado, ele representa uma interseção na distância correta (no espaço mundial!) da origem do raio.

Pause aqui e faça esses testes passarem implementando o transformar (raio, matriz) função.

Uma vez que seus raios podem ser transformados, o próximo passo é permitir que uma transformação seja atribuída a uma esfera. Implemente os seguintes testes para demonstrar que uma esfera tem uma transformação padrão e que sua transformação pode ser atribuída.

**features / spheres.feature**

```
Cenário : A transformação padrão de uma esfera
    Dado s ← esfera ()
    Então s.transform = identity_matrix

Cenário : Mudando a transformação de uma esfera
    Dado s ← esfera ()
        E t ← tradução (2, 3, 4)
    Quando set_transform (s, t)
    Então s.transform = t
```

Finalmente, faça com que seu cruzar função transforma o raio antes de fazer o cálculo. Adicione os seguintes testes para ilustrar dois cenários possíveis.

**features / spheres.feature**

```
Cenário : Cruzando uma esfera em escala com um raio
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando set_transform (s, escala (2, 2, 2))
        E xs ← cruzar (s, r)
```

```
    Então xs.count = 2
        E xs [0] .t = 3
        E xs [1] .t = 7

Cenário : Cruzando uma esfera traduzida com um raio
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← esfera ()
    Quando set_transform (s, tradução (5, 0, 0))
        E xs ← cruzar (s, r)
    Então xs.count = 0
```

Agora vá e faça esses testes passarem. Você precisará certificar-se de que o raio passou para cruzar é transformado pelo inverso da matriz de transformação da esfera. Em pseudocódigo, significa adicionar uma linha no topo da função, assim:

```
função cruzar (esfera, raio)
    ray2 ← transformar (raio, inverso (sphere.transform))

    # . .
função final
```

Certifique-se de usar o novo raio em outros cálculos da função também.

Depois que tudo estiver funcionando, dê um tapinha nas costas! Não é lindo? Você consegue manter sua adorável esfera unitária e ainda deformá-la de todas as maneiras. Você pode transformá-lo em um elipsóide dimensionando-o de forma não uniforme, distorcendo-o com uma transformação de cisalhamento e traduzindo-o onde quiser em uma cena - tudo isso aplicando o inverso da transformação ao raio.

É mágico!

Você ainda não pode renderizar uma cena 3D, mas está mais perto do que estava. Na verdade, você está chegando muito perto. É hora de colocar alguns desses conceitos juntos em algo concreto e mostrar o quão próximo você é.

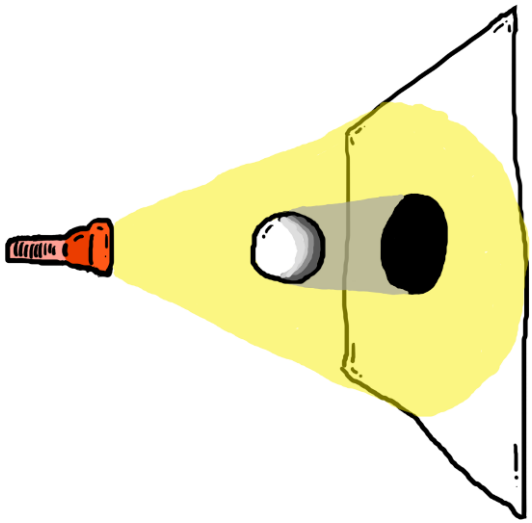
## Juntar as peças

Sua tarefa final neste capítulo é escrever um programa que lança raios em uma esfera e desenha a imagem em uma tela. Qualquer raio que atinja a esfera deve resultar em um pixel colorido (vermelho, por exemplo), e qualquer erro deve ser desenhado em preto. O resultado será uma silhueta da esfera - não tridimensional, mas definitivamente redonda!

Aqui estão algumas dicas para ajudá-lo. Pare de ler a qualquer momento se achar que tem uma solução!

### Dica # 1

Pense como se você estivesse tentando projetar a sombra do seu objeto em alguma parede atrás dele, como na figura a seguir.



Você projeta cada raio de algum ponto inicial em direção a algum ponto na parede que corresponde a uma posição em sua tela. Se o raio interceptar a esfera, uma sombra é projetada, que você marcará com um pixel colorido.

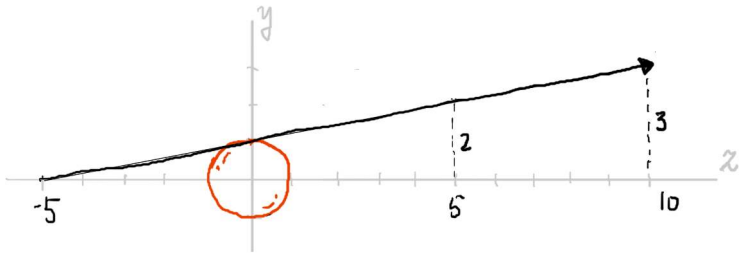
### Dica # 2

Descubra a que distância a origem do seu raio está da esfera. Além disso, decida onde ficará sua parede. Mover a origem do raio para mais perto da esfera tornará a esfera no desenho maior. Movê-lo para mais longe tornará a esfera menor. Mover a parede fará o mesmo. Para saber por onde começar, experimente estes valores:

```
# inicie o raio em z = -5
ray_origin ← ponto (0, 0, -5)

# coloque a parede em z = 10
wall_z ← 10
```

Em seguida, decida o quão grande sua parede deve ser. Porque você está usando esferas unitárias, o máximo y valor para a esfera vai ser 1 Com isso, você pode extrapolar entre a origem do raio e a parede para ver o quão grande a parede deve ser, conforme mostrado no figura na página 72 .

.....




Então, com a parede em z = 10, ele precisa ter pelo menos 6 unidades de diâmetro para capturar toda a sombra da esfera. Dê a si mesmo um pouco de margem e chame-o
7. (Suponha que a parede seja um quadrado.)

```
wall_size ← 7,0
```

## Dica # 3

Decida o tamanho que você deseja que sua tela seja (em pixels). Uma tela de 100 pixels de lado é provavelmente boa para começar. (Imagens maiores levarão mais tempo exponencialmente para renderizar.)

```
canvas_pixels ← 100
```

Depois de saber como muitos pixels cabem ao longo de cada lado da parede, você pode dividir o tamanho da parede pelo número de pixels para obter o tamanho de um único pixel (em unidades de espaço mundial).

```
pixel_size ← wall_size / canvas_pixels
```

Então, suponha que você está olhando diretamente para o centro da esfera. Metade da parede ficará à esquerda e a outra metade à direita. Calcule esse tamanho.

```
metade ← wall_size / 2
```

Uma vez que a parede está centrada em torno da origem (porque a esfera está na origem), isso significa que este metade variável descreve o mínimo e máximo
x e y coordenadas de sua parede.

## Dica # 4

Agora que você conhece a origem de cada raio, as dimensões da sua tela e o tamanho da sua parede, pode calcular, lançar e cruzar os raios. A seguir está uma forma possível de abordá-lo, em pseudocódigo:

```
tela de pintura ← tela de pintura (canvas_pixels, canvas_pixels) cor ← cor (1, 0, 0) # vermelho

forma ← esfera ()

# para cada linha de pixels na tela
pra y ← 0 a canvas_pixels - 1
```

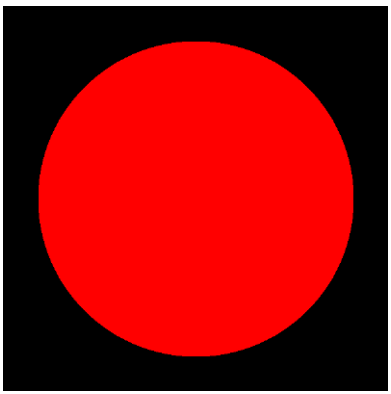
```
➢   # calcula a coordenada y mundial (parte superior = + metade, parte inferior = metade)
➢   world_y ← meio - pixel_size * y

    # para cada pixel na linha
    pra x ← 0 a canvas_pixels - 1

        # calcula a coordenada x mundial (esquerda = metade, direita = metade)
        world_x ← - meio + pixel_size * x

        # descreve o ponto na parede que o raio terá como alvo
        posição ← ponto (world_x, world_y, wall_z)

        r ← raio (ray_origin, normalizar (posição - ray_origin)) xs ← cruzar (forma, r)


        E se acertar (xs) está definido
            write_pixel (tela, x, y, cor)
        fim se

    fim para

fim para
```

Não se esqueça de salvar a tela em um arquivo no final!

Observe as linhas destacadas, onde o mundo `y` coordenada é calculada. No espaço mundial, o `y` a coordenada aumenta à medida que você sobe e diminui à medida que você desce. Mas na tela, o topo está em `y = 0`, e `y` aumenta conforme você avança *baixa*. Assim, para renderizar o círculo corretamente, você deve virar o `y` coordenada, que é realizada subtraindo-a de seu valor máximo (o topo da parede,

ou metade).

Se tudo correr bem, você deverá ver um círculo, muito parecido com o seguinte:



Parabéns! Esta é a silhueta de sua esfera, desenhada em sua tela, um raio de cada vez.

Uma vez que você tenha trabalhado tanto, tente deformar a esfera com algumas transformações e veja o que acontece. Aqui estão algumas idéias:

```
# reduza ao longo do eixo y
shape.transform ← escala (1, 0,5, 1)

# reduza ao longo do eixo x
shape.transform ← escala (0,5, 1, 1)

# encolher e girar!
shape.transform ← rotação_z (pi / 4) * escala (0,5, 1, 1)

# encolher e inclinar!
shape.transform ← tosquia (1, 0, 0, 0, 0, 0) * escala (0,5, 1, 1)
```

Quando você tiver se divertido tanto quanto pode suportar, siga em frente. Uma silhueta é eficaz, mas você pode fazer muito melhor. No próximo capítulo, você adicionará iluminação e sombreamento para fazer com que a esfera pareça tridimensional!

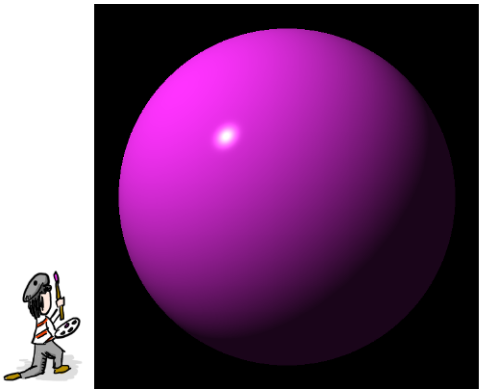
CAPÍTULO 6

# Luz e Sombreamento

Diggity quente! Você é imparável. Você acabou de desenhar a silhueta de uma esfera tridimensional com nada além de algum código e matemática! Isso é, tipo, coisa de assistente nível10.

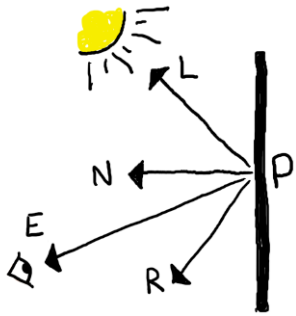
Mesmo assim - triste, mas é verdade! - os resultados não são exatamente o que a maioria das pessoas chama de "renderização em 3D". É hora de consertar isso.

Neste capítulo, você implementará um modelo para simular o reflexo da luz de uma superfície, o que finalmente permitirá que você desenhe essa esfera e faça com que pareça tridimensional. Na verdade, no final do capítulo, você terá renderizado uma imagem muito parecida com esta:



Para fazer isso, você adicionará uma fonte de luz e, em seguida, implementará um algoritmo de sombreamento para aproximar a intensidade da luz que ilumina as superfícies nas quais ela incide. Pode parecer complicado, mas não é. A verdade é que a maioria dos traçadores de raios prefere aproximações em vez de simulações fisicamente precisas, de modo que, para sombrear qualquer ponto, você só precisa conhecer quatro vetores. Estes são ilustrados no

figura na página 76 .




Se P é onde seu raio cruza um objeto, esses quatro vetores são definidos como:

- E é o *vetor do olho,* apontando de P à origem do raio (geralmente, onde existe o olho que está olhando para a cena).
- eu é o *vetor de luz,* apontando de P para a posição da fonte de luz.
- N é o *superfície normal,* um vetor que é perpendicular à superfície em P.
- R é o *vetor de reflexão,* apontando na direção em que a luz incidente iria ricochetear ou refletir.

Você já tem as ferramentas para calcular os dois primeiros vetores:

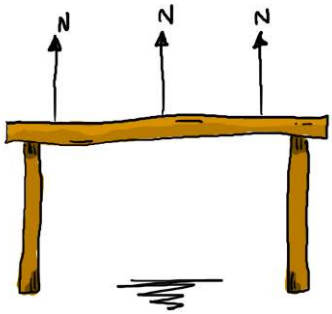
- Encontrar E, você pode negar o vetor de direção do raio, girando-o para apontar de volta para sua origem.

- Encontrar EU, você subtrai P da posição da fonte de luz, dando a você o vetor apontando em direção à luz.

A normal de superfície e o vetor de reflexão, entretanto ... são novos. Antes que você possa usá-los, precisamos fazer uma pausa e falar sobre como calculá-los.

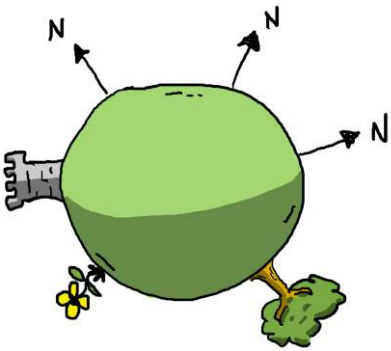
## Normais de superfície

UMA *superfície normal (* ou apenas *normal)* é um vetor que aponta perpendicularmente a uma superfície em um determinado ponto. Considere uma tabela, conforme mostrado na figura a seguir.



Uma superfície plana como uma mesa terá a mesma normal em todos os pontos de sua superfície, conforme mostrado pelos vetores rotulados N. Se a mesa estiver nivelada, os normais serão os mesmos que “para cima”, mas mesmo se inclinarmos a mesa, eles ainda serão perpendiculares à superfície da mesa, como mostra a figura a seguir.

As coisas ficam um pouco mais complicadas quando começamos a falar sobre superfícies não planas (aquelas que não são uniformemente planas). Pegue o planetóide na figura a seguir, por exemplo.




Os três vetores normais certamente não estão apontando todos na mesma direção! Mas cada um é perpendicular à superfície da esfera no ponto onde vive.

Vejamos como realmente computar esses vetores normais.

## Calculando o Normal em uma Esfera

Comece escrevendo os seguintes testes para demonstrar o cálculo do normal em vários pontos de uma esfera. Apresente uma nova função, normal_at (esfera, ponto),
que retornará o normal na esfera dada, no ponto determinado. Você pode presumir que o ponto sempre estará na superfície da esfera.

**features / spheres.feature**

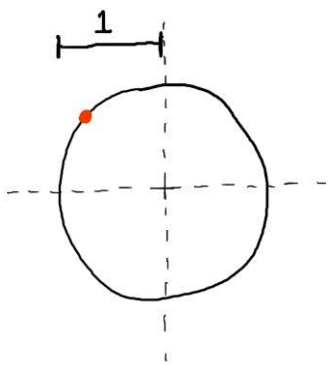
```
Cenário : O normal em uma esfera em um ponto no eixo x
    Dado s ← esfera ()
    Quando n ← normal_at (s, ponto (1, 0, 0))
    Então n = vetor (1, 0, 0)

Cenário : O normal em uma esfera em um ponto do eixo y
    Dado s ← esfera ()
    Quando n ← normal_at (s, ponto (0, 1, 0))
    Então n = vetor (0, 1, 0)

Cenário : O normal em uma esfera em um ponto no eixo z
    Dado s ← esfera ()
    Quando n ← normal_at (s, ponto (0, 0, 1))
    Então n = vetor (0, 0, 1)

Cenário : O normal em uma esfera em um ponto não axial
    Dado s ← esfera ()
    Quando n ← normal_at (s, ponto ( √ 3/3, √ 3/3, √ 3/3))
    Então n = vetor ( √ 3/3, √ 3/3, √ 3/3)
```

Uma outra característica desses vetores normais está se escondendo à vista de todos: eles são *normalizado.* Adicione o seguinte teste ao seu conjunto, que mostra que uma normal de superfície sempre deve ser normalizada.

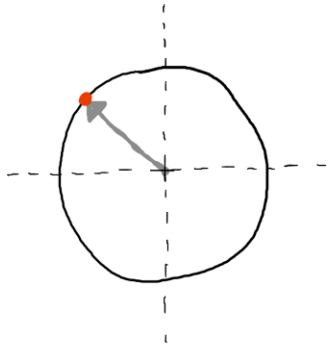
**features / spheres.feature**

```
Cenário : O normal é um vetor normalizado
    Dado s ← esfera ()
    Quando n ← normal_at (s, ponto ( √ 3/3, √ 3/3, √ 3/3))
    Então n = normalizar (n)
```

Agora, vamos fazer esses testes passarem implementando normal_at () função. Para entender como funciona sua mágica, dê uma olhada no círculo unitário na figura a seguir. Está centrado na origem e um ponto (presumivelmente um ponto de intersecção) foi destacado em sua circunferência.



Digamos que você queira encontrar o normal naquele ponto destacado. Desenhe uma seta da origem do círculo até esse ponto, como na figura a seguir.



Acontece que esta seta - este vetor! - é perpendicular à superfície do círculo no ponto onde ele se cruza. É normal! Falando algoritmicamente, você encontra o normal pegando o ponto em questão e subtraindo a origem da esfera (( 0,0,0) no seu caso). Aqui está em pseudocódigo:

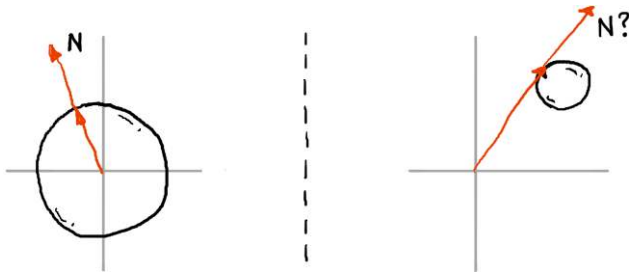
```
função normal_at (esfera, p)
    Retorna normalizar (p - ponto (0, 0, 0))
função final
```

(Observe que, como esta é uma esfera unitária, o vetor será normalizado por padrão para qualquer ponto em sua superfície, portanto, não é estritamente necessário normalizá-lo explicitamente aqui.)

Se apenas isso fosse tudo! Infelizmente, a matriz de transformação da esfera vai jogar uma (pequena) chave em como o normal é calculado. Vamos dar uma olhada no que precisa acontecer para o cálculo normal compensar uma matriz de transformação.

## Transformando Normais

Imagine que você tem uma esfera que foi transladada a alguma distância da origem do mundo. Se você aplicasse ingenuamente o algoritmo acima para encontrar a normal em quase qualquer ponto dessa esfera, descobriria que ele não funciona mais corretamente. O figura na página 80 mostra como isso dá errado neste caso. À esquerda, a normal para uma esfera na origem é calculada. À direita, o normal é calculado para uma esfera que foi movida para longe da origem.




O “normal” à direita não está normalizado remotamente e nem mesmo está apontando na direção correta. Porque? O problema é que sua suposição mais básica foi quebrada: a origem da esfera não está mais na origem do mundo.

Escreva os seguintes testes para mostrar o que *deveria* acontecer. Eles demonstram o cálculo do normal primeiro em uma esfera transladada e, em seguida, em uma esfera dimensionada e girada.

**features / spheres.feature**

```
Cenário : Calculando o normal em uma esfera traduzida
    Dado s ← esfera ()
        E set_transform (s, tradução (0, 1, 0))
    Quando n ← normal_at (s, ponto (0, 1,70711, -0,70711))
    Então n = vetor (0, 0,70711, -0,70711)

Cenário : Calculando o normal em uma esfera transformada
    Dado s ← esfera ()
        E m ← escala (1, 0,5, 1) * rotação_z ( π / 5)
        E set_transform (s, m)
    Quando n ← normal_at (s, ponto (0, √ 2/2, - √ 2/2))
    Então n = vetor (0, 0,97014, -0,24254)
```

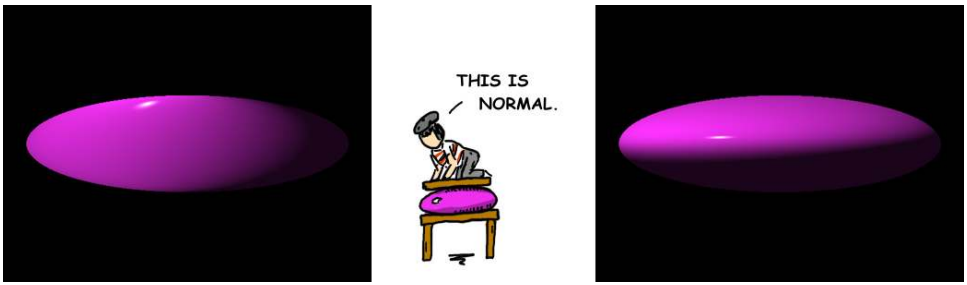
Eles não passarão ainda, mas você os tornará verdes em apenas um momento.

Lembre-se de quando falamos sobre World Space vs. Object Space, na página 68 ? Acontece que essa distinção entre o mundo e o espaço do objeto também é parte da solução para esse enigma. Você tem um ponto no espaço mundial e deseja saber a normal na superfície correspondente no espaço do objeto. O que fazer? Bem, primeiro você tem que converter o ponto do espaço do mundo para o espaço do objeto, multiplicando o ponto pelo inverso da matriz de transformação, assim:

```
object_point ← inverso (transformar) * world_point
```

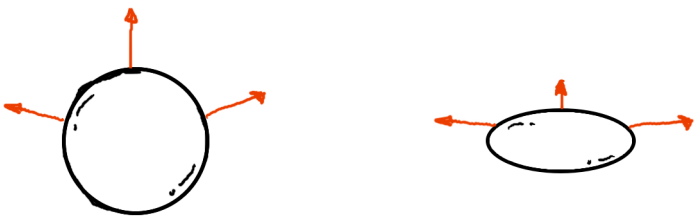
Com esse ponto agora no espaço do objeto, você pode calcular o normal como antes, porque no espaço do objeto, a origem da esfera está na origem do mundo. Contudo! O vetor normal que você obtém *Além disso* esteja no espaço do objeto ... e para desenhar qualquer coisa útil com ele, você precisará convertê-lo de volta ao espaço do mundo de alguma forma.

Agora, se o normal fosse um *ponto* você pode transformá-lo multiplicando-o pela matriz de transformação. Afinal, é isso que a matriz de transformação faz: ela transforma pontos do espaço do objeto em espaço do mundo. E na verdade, este *quase* funciona aqui também. Considere as duas imagens a seguir de uma esfera comprimida, que foi reduzida em escala em y. Os vetores normais do da esquerda foram multiplicados pela matriz de transformação. O da direita é como a esfera é *suposto* olhar.




O da esquerda definitivamente parece ... desligado. É como se alguém tirasse uma foto de uma esfera regular não transformada e amassasse *que,* em vez de esmagar a própria esfera. Qual é a diferença?

Tudo se resume a como os vetores normais estão sendo transformados. A ilustração a seguir mostra o que acontece. A esfera é dimensionada em y, esmagando-o verticalmente, e os normais são multiplicados pela matriz de transformação.



Como você pode ver, multiplicar pela matriz de transformação não preserva uma das propriedades fundamentais dos vetores normais neste caso: o normal não será necessariamente perpendicular à superfície após ser transformado!

Então, como você manterá os normais perpendiculares à sua superfície? A resposta é multiplicar o normal pelo *matriz de transposição inversa* em vez de. Então você pega sua matriz de transformação, inverte-a e, em seguida, transpõe o resultado. *Esta* é o que você precisa para multiplicar o normal.

```
world_normal ← transpor ( inverso (transformar)) * object_normal
```

Esteja ciente de duas coisas adicionais aqui:

1. Tecnicamente, você deve encontrar submatriz (transformação, 3, 3) ( a partir de Spotting Submatrizes, na página 34 ) primeiro, e multiplicando pelo inverso e transposto de *que.* Caso contrário, se a sua transformação incluir qualquer tipo de tradução, a multiplicação por sua transposição acabará com o C coordene em seu vetor, o que causará todos os tipos de estragos em cálculos posteriores. Mas se você não se importa com um pequeno hack, pode evitar tudo isso apenas configurando

   world_normal.w para 0 depois de multiplicar pela matriz inversa de transposição 4x4. A matriz de
2 transposição inversa pode alterar o comprimento do seu vetor, portanto, se você alimentá-la com um vetor de comprimento 1 (um vetor normalizado), pode não obter um vetor normalizado! É melhor estar seguro e sempre normalizar o resultado.

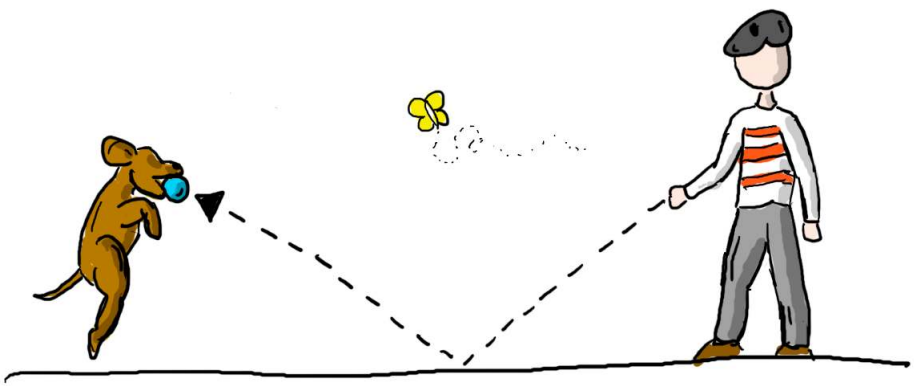
Em pseudocódigo, então, seu normal_at () função deve ser semelhante ao seguinte.

```
função normal_at (esfera, ponto_mundo)
    object_point ← inverso (sphere.transform) * world_point object_normal ← object_point - ponto (0, 0, 0) world_normal ← transpor ( inverso
    (sphere.transform)) * object_normal world_normal.w ← 0


    Retorna normalizar (world_normal)
função final
```

Vá em frente e faça uma pausa enquanto você faz as coisas funcionarem até este ponto. Assim que seus testes estiverem verdes, vamos falar sobre como calcular o *vetor de reflexão.*

## Vetores Refletindo

Imagine jogar uma bola para o seu cachorro. Você joga a bola no chão em um ponto intermediário entre vocês dois, a bola quica para cima e seu cão (se ele for bem treinado) a pega, como ilustra a figura a seguir.



A velocidade da bola é *refletido* em torno do normal no ponto onde ele atinge o solo. Ou seja, ele continua avançando, mas em vez de cair ao cair, agora está subindo. Qualquer pessoa que já jogou com uma bola sabe intuitivamente o que isso significa. Todos nós sabemos por experiência própria em que direção a bola provavelmente saltará.

Escreva os dois testes a seguir para reforçar essa intuição. Você apresentará uma função chamada refletir (em, normal), que retorna o resultado de refletir o no
vetor em torno do normal vetor.

Este primeiro teste mostra o caso em que um vetor se aproxima de uma normal em um ângulo de 45 °, movendo-se em velocidade igual em ambos x e y. Deve emergir em um ângulo de 45 °, com seu y componente invertido.

```
features / tuples.feature
Cenário : Refletindo um vetor se aproximando de 45 °
    Dado v ← vetor (1, -1, 0)
        E n ← vetor (0, 1, 0)
    Quando r ← refletir (v, n)
    Então r = vetor (1, 1, 0)
```

Isso deve funcionar independentemente da orientação do vetor normal. Por exemplo, se o solo tiver uma inclinação de 45 ° e a bola cair diretamente sobre ele, ela deve ricochetear horizontalmente, como o teste a seguir demonstra.

```
features / tuples.feature
Cenário : Refletindo um vetor em uma superfície inclinada
    Dado v ← vetor (0, -1, 0)
        E n ← vetor ( √ 2/2, √ 2/2, 0)
    Quando r ← refletir (v, n)
    Então r = vetor (1, 0, 0)
```

Como você pode esperar, a matemática é a mágica que faz isso funcionar. Dados dois vetores no e normal, o seguinte pseudocódigo é o encantamento de que você precisa.

```
função refletir (dentro, normal)
    Retorna em - normal * 2 * ponto (dentro, normal)
função final
```

Vá em frente e faça todos os seus testes passarem. Quando estiver pronto, é hora de começar a sombrear as coisas!

## O modelo de reflexão Phong

Muitos algoritmos diferentes podem simular o reflexo da luz, mas aquele que você implementará aqui é chamado de *Modelo de reflexão Phong (* nomeado para Bui

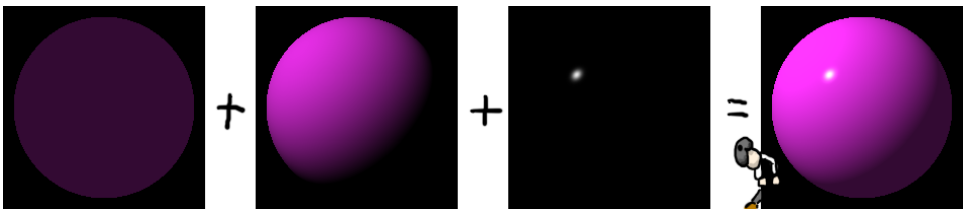
Tuong Phong, o pesquisador que o desenvolveu). Simula a interação entre três tipos diferentes de iluminação:

- *Reflexão ambiente* é a iluminação de fundo, ou luz refletida de outros objetos no ambiente. O modelo Phong trata isso como uma constante, colorindo todos os pontos da superfície igualmente.

- *Reflexão difusa* é a luz refletida de uma superfície fosca. Depende apenas do ângulo entre a fonte de luz e a normal da superfície.

- *Reflexão especular* é o reflexo da própria fonte de luz e resulta no que é chamado de *destaque especular* —O ponto brilhante em uma superfície curva. Depende apenas do ângulo entre o vetor de reflexão e o vetor do olho e é controlado por um parâmetro que chamaremos *brilho.* Quanto maior for o brilho, menor e mais estreito será o realce especular.

A ilustração a seguir mostra os efeitos de cada um desses atributos. A primeira esfera é renderizada usando apenas reflexão ambiente, a segunda esfera usa apenas reflexão difusa e a terceira esfera usa apenas reflexão especular. A última esfera combina todos os três.



Como você pode ver, eles próprios não fazem muita coisa. Mas ao combiná-los, você obtém algo com muito mais potencial!

A primeira coisa que você vai precisar para isso é uma fonte de luz. Você vai implementar o que é chamado de *ponto de luz* - uma fonte de luz sem tamanho, existindo em um único ponto no espaço. Também é definido por seu *intensidade,* ou quão brilhante é. Essa intensidade também descreve a cor da fonte de luz.

Adicione o seguinte teste para demonstrar os atributos de uma luz pontual.

**features / lights.feature**

```
Cenário : Um ponto de luz tem uma posição e intensidade
    Dado intensidade ← cor (1, 1, 1)
        E posição ← ponto (0, 0, 0)
    Quando claro ← point_light (posição, intensidade)
    Então light.position = position
        E luz.intensidade = intensidade
```

A próxima coisa que você precisa é uma estrutura chamada material que encapsula não apenas a cor da superfície, mas também os quatro novos atributos do modelo de reflexão Phong: ambiente, difuso, especular, e brilho. Cada um deve aceitar um número de ponto flutuante não negativo. Pra ambiente, difuso, e especular, os valores típicos estão entre 0 e 1. Para brilho, valores entre 10 (destaque muito grande) e 200 (destaque muito pequeno) parecem funcionar melhor, embora não haja nenhum limite superior real.

Adicione o seguinte teste, que apresenta um material() função e mostra os valores padrão de cada um dos atributos do material.

**features / materials.feature**

```
Cenário : O material padrão
    Dado m ← material ()
    Então m.color = cor (1, 1, 1)
        E m.ambient = 0,1
        E m.diffuse = 0,9
        E m.specular = 0,9
        E m.shininess = 200,0
```

Em seguida, adicione um material propriedade para sua esfera, junto com os testes a seguir. Eles mostram como essa propriedade é usada e qual deve ser seu valor padrão.

**features / spheres.feature**

```
Cenário : Uma esfera tem um material padrão
    Dado s ← esfera ()
    Quando m ← material
    Então m = material ()

Cenário : Uma esfera pode ser atribuída a um material
    Dado s ← esfera ()
        E m ← material ()
        E m.ambient ← 1
    Quando material ← m
    Então s.material = m
```

Faça seus testes passarem implementando o ponto de luz, o material() função, e a esfera material propriedade. Assim que tiver isso, reuniremos tudo com mais uma função: iluminação().

Esta iluminação() função é o que irá sombrear seus objetos para que pareçam tridimensionais. Ele espera cinco argumentos: o próprio material, o ponto sendo iluminado, a fonte de luz e o olho e os vetores normais do modelo de reflexão de Phong. Embora a função não seja especialmente complicada por si mesma, vários casos a serem considerados pelos testes garantirão que tudo esteja correto. Comece escrevendo a seguinte série de testes, que irão mover o olho e a fonte de luz para exercitar o iluminação funcionar em diferentes configurações.

Você pode presumir que cada um desses testes compartilha a seguinte configuração:

**features / materials.feature**

```
Fundo :
    Dado m ← material ()
        E posição ← ponto (0, 0, 0)
```

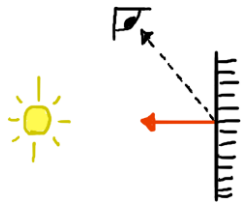
Para o primeiro teste, o olho é posicionado diretamente entre a luz e a superfície, com o normal apontando para o olho, assim:

Nesse caso, você espera que o ambiente, o difuso e o especular estejam com força total. Isso significa que a intensidade total deve ser 0,1 ( o valor ambiente) + 0,9 ( o valor difuso) + 0,9 ( o valor especular), ou 1.9.

**features / materials.feature**

```
Cenário : Iluminação com o olho entre a luz e a superfície
    Dado Eyev ← vetor (0, 0, -1)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 0, -10), cor (1, 1, 1))
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv)
    Então resultado = cor (1,9, 1,9, 1,9)
```

Neste próximo teste, a superfície e a luz permanecem as mesmas de antes, mas você moverá o olho para um ponto 45 ° fora do normal, como mostrado na próxima ilustração.

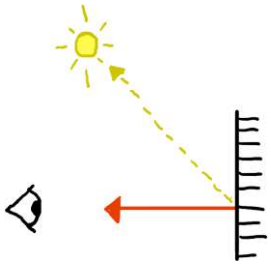


Aqui, os componentes ambiente e difuso devem permanecer inalterados (porque o ângulo entre a luz e os vetores normais não terá mudado), mas o valor especular deve ter caído para (efetivamente) 0. Assim, a intensidade deve ser 0,1 + 0,9 + 0, ou 1.0.

**features / materials.feature**

```
Cenário : Iluminação com o olho entre a luz e a superfície, deslocamento do olho 45 °
    Dado Eyev ← vetor (0, √ 2/2, - √ 2/2)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 0, -10), cor (1, 1, 1))
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv)
    Então resultado = cor (1,0, 1,0, 1,0)
```

Em seguida, o olho volta a ficar diretamente oposto à superfície, mas a luz é movida para uma posição 45 ° fora do normal. A figura a seguir mostra como isso se parece.

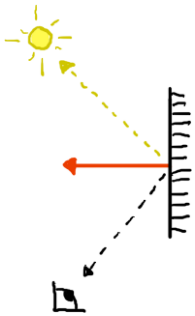


Como o ângulo entre os vetores de luz e normal mudou, o componente difuso torna-se $0.9 \times \sqrt{2}/2$ O componente especular novamente cai para 0, então a intensidade total deve ser $0{,}1 + 0{,}9 \times \sqrt{2}/2 + 0$, ou aproximadamente 0,7364.

**features / materials.feature**

```
Cenário : Iluminação com a superfície oposta do olho, deslocamento de luz de 45 °
    Dado Eyev ← vetor (0, 0, -1)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 10, -10), cor (1, 1, 1))
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv)
    Então resultado = cor (0,7364, 0,7364, 0,7364)
```

Para este próximo teste, os vetores de luz e normal são os mesmos do teste anterior, mas você moverá o olho diretamente para o caminho do vetor de reflexão, assim:

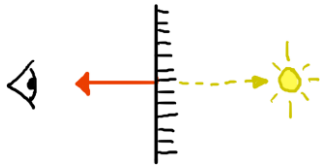


Isso deve fazer com que o componente especular esteja com força total, com ambiente e difusão igual ao do teste anterior. A intensidade total deve, portanto, estar $0{,}1 + 0{,}9 \times \sqrt{2}/2 + 0{,}9$, ou aproximadamente 1,6364.

**features / materials.feature**

```
Cenário : Iluminação com olho no caminho do vetor de reflexão
    Dado Eyev ← vetor (0, - √ 2/2, - √ 2/2)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 10, -10), cor (1, 1, 1))
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv)
    Então resultado = cor (1,6364, 1,6364, 1,6364)
```

Para o teste final, você move a luz atrás da superfície, assim:



Como a luz não ilumina mais a superfície, os componentes difuso e especular vão para 0. A intensidade total deve ser igual à do componente ambiente, ou 0,1.

**features / materials.feature**

```
Cenário : Iluminação com a luz por trás da superfície
    Dado Eyev ← vetor (0, 0, -1)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 0, 10), cor (1, 1, 1))
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv)
    Então resultado = cor (0,1, 0,1, 0,1)
```

Então, esses são os testes! Faça-os passar agora, implementando o iluminação()
função. Em poucas palavras, ele irá somar os componentes ambiente, difuso e especular do material, ponderados pelos ângulos entre os diferentes vetores. Em pseudocódigo (anotado), é mais ou menos assim:

```
função iluminação (material, luz, ponto, olho, normalv)
    # combinar a cor da superfície com a cor / intensidade da luz
    cor_efetiva ← material.color * light.intensity

    # encontre a direção para a fonte de luz
    luz ← normalizar (light.position - point)

    # calcule a contribuição do ambiente
    ambiente ← cor_eficaz * material.ambiente

    # light_dot_normal representa o cosseno do ângulo entre o
    # vetor de luz e o vetor normal. Um número negativo significa que
    # a luz está do outro lado da superfície.
    light_dot_normal ← ponto (levev, normalv)
    E se light_dot_normal <0
        difuso ← Preto
        especular ← Preto

    outro
        # calcula a contribuição difusa
        difuso ← effect_color * material.diffuse * light_dot_normal

        # reflect_dot_eye representa o cosseno do ângulo entre o
        # vetor de reflexão e o vetor de olho. Um número negativo significa que
        # a luz reflete longe do olho.
        refletv ← refletir (-lightv, normalv) reflect_dot_eye ← ponto (refletv, eyev)
```

```
        E se reflect_dot_eye <= 0
            especular ← Preto
        outro
            # calcula a contribuição especular
            fator ← Pancada (reflect_dot_eye, material.shininess) especular ← light.intensity * material.specular *
            factor
        fim se
    fim se

    # Adicione as três contribuições para obter o sombreamento final
    Retorna ambiente + difuso + especular
função final
```

Vá em frente e faça todos os testes passarem. Uma vez que eles estão todos verdes, você pode ter certeza de que suas rotinas de sombreamento estão funcionando como deveriam, e você pode passar para a parte final deste capítulo: renderizar uma esfera com iluminação realista!

## Juntar as peças

OK. Dê uma olhada no programa que você escreveu no final do capítulo anterior, aquele em que desenhou a silhueta de uma esfera em uma tela. É hora de revisitar isso e transformar a silhueta em uma renderização 3D completa. Faça as seguintes alterações nesse programa:

1. Atribua um material à sua esfera. O material a seguir lhe dará uma esfera que se parece com as ilustrações deste capítulo.

```
esfera.material ← material ()
esfera.material.color ← cor (1, 0,2, 1)
```

2. Adicione uma fonte de luz. Aqui está uma configuração possível, com uma luz branca atrás, acima e à esquerda do olho:

```
light_position ← ponto (-10, 10, -10) light_color
                                ←    cor (1, 1, 1)
claro                           ←    point_light (posição_luz, cor_luz)
```

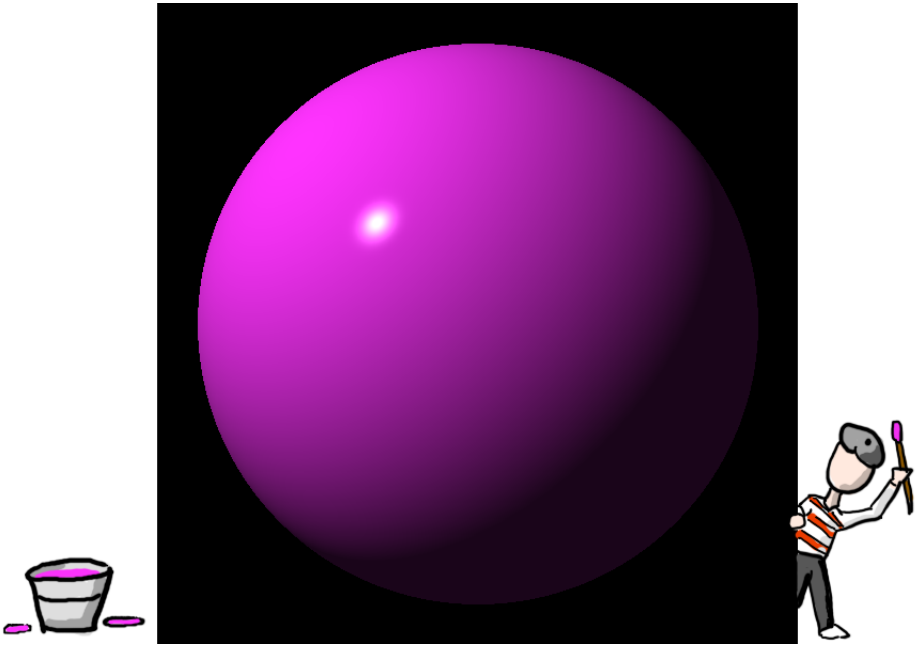
3. No loop onde você lança seus raios, certifique-se de normalizar a direção dos raios. Não importava antes, mas importa agora! Além disso, uma vez que você tenha uma interseção, encontre o vetor normal na área (a interseção mais próxima) e calcule o vetor do olho.

```
ponto ← posição (raio, hit.t)
normal ← normal_at (hit.object, point) olho
                ←    - ray.direction
```

4. Finalmente, calcule a cor com o seu iluminação() função antes de aplicá-la para a tela.

```
cor ← iluminação (hit.object.material, light, point, eye, normal)
```

O resultado, quando terminar, deve ser semelhante à figura a seguir.



A partir daí, experimente diferentes transformações da esfera. Amasse, gire, dimensione. Experimente cores diferentes e parâmetros de materiais diferentes. O que acontece quando você aumenta o valor ambiente? E se o difuso e o especular forem baixos? O que acontece quando você move a fonte de luz ou altera sua intensidade?

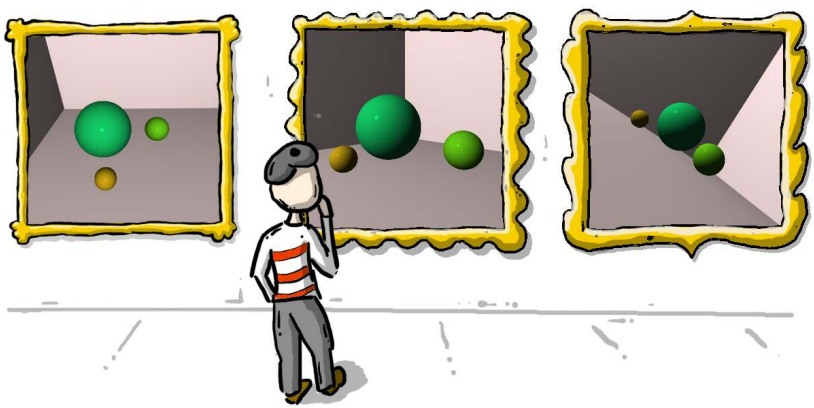
Depois de se divertir com isso, vá em frente e vire a página. Em seguida, são as câmeras e os mundos, que irão preparar o cenário para cenas mais complexas!

CAPÍTULO 7

# Fazendo uma cena

Pense nisso por um segundo. Você escreveu um programa do zero, com testes, que desenha um objeto tridimensional simulando o comportamento da luz. Isso é *impressionante.*

E ainda é apenas o começo! Cenas mais complexas estão ao virar da esquina. Ao final deste capítulo, você criará mundos com vários objetos e usará uma câmera virtual para capturar visualizações desses objetos de diferentes pontos de vista. Apenas mais algumas páginas e você renderizará imagens como as da galeria de Pierre, aqui:



Para chegar lá, você primeiro implementará um *mundo* —Uma coleção de todos os objetos em uma cena — bem como rotinas para cruzar aquele mundo com um raio e computar as cores para cruzamentos. Em seguida, você construirá uma nova transformação de matriz, chamada de *transformação de visão,* que você usará para orientar a vista. Por último, você implementará o *Câmera,* que encapsula a visualização e fornece uma interface para renderizar o mundo em uma tela.

Preparar? Vai!

## Construindo um Mundo

A primeira etapa é implementar o objeto mundo. Pense em quanto trabalho deu para renderizar uma única esfera e, em seguida, multiplique isso por dezenas de objetos. Você começa a ver o que ganha por ter algo que manterá o controle de todas essas coisas para você.

Inicialmente, um mundo está vazio, sem objetos e sem fonte de luz. Escreva um teste como o seguinte, demonstrando um mundo() função que retorna apenas essa estrutura de dados.

**features / world.feature**

```
Cenário : Criando um mundo
    Dado C ← mundo ()
    Então w não contém objetos
        E w não tem fonte de luz
```

Alguns dos testes que você escreverá neste capítulo assumem que existe um mundo padrão com uma fonte de luz em (- 10, 10, -10). Este mundo contém duas esferas concêntricas, onde a mais externa é uma esfera unitária e a mais interna tem um raio de 0,5. Ambos estão na origem. Adicione o seguinte teste para garantir que este mundo padrão esteja configurado corretamente.

**features / world.feature**

```
Cenário : O mundo padrão
    Dado claro ← point_light ( ponto (-10, 10, -10), cor (1, 1, 1))
        E s1 ← esfera () com:
            | material.color          | (0,8, 1,0, 0,6)          |
            | material.diffuse         | 0,7                      |
            | material.specular | 0,2                              |
        E s2 ← esfera () com:
            | transformar | escala (0,5, 0,5, 0,5) |
    Quando C ← default_world ()
    Então w.light = light
        E w contém s1
        E w contém s2
```

Usando esse mundo padrão, escreva um teste descrevendo o comportamento de um novo intersect_world (mundo, raio) função, que aceita um mundo e um raio, e retorna as interseções. Neste caso, como o raio passa pela origem (onde ambas as esferas estão centradas), ele deve interceptar cada esfera duas vezes, para um total de quatro interseções.

**features / world.feature**

```
Cenário : Cruze um mundo com um raio
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
    Quando xs ← intersect_world (w, r)
```

```
Então xs.count = 4
    E xs [0] .t = 4
    E xs [1] .t = 4,5
    E xs [2] .t = 5,5
    E xs [3] .t = 6
```

Faça esse teste passar. O intersect_world () A função deve iterar sobre todos os objetos que foram adicionados ao mundo, cruzando cada um deles com o raio e agregando as interseções em uma única coleção. Observe que para o teste passar, intersect_world () deve retornar as interseções em ordem classificada.

**Joe pergunta:**

### Por que tenho que classificar os cruzamentos?

Tudo o que você está fazendo com as interseções neste ponto é encontrar o acerto, ou a interseção com o mínimo positivo t valor. A lista não precisa ser classificada apenas para fazer isso, mas classificar os cruzamentos tem alguns benefícios. A primeira é que simplifica os testes, pois permite depender da ordem das interseções retornadas. A segunda é quando você começa a Capítulo 11, Reflexão e Refração, na página 141 , e Capítulo 16, Geometria Sólida Construtiva (CSG), na página 227 , você precisará ser capaz de iterar nas interseções em ordem crescente, e ter essa lista já classificada economizará algum esforço.

Assim que sua suíte estiver passando novamente, é hora de descobrir o sombreamento para o cruzamento mais próximo (o "acerto", de Identificando sucessos, na página 64 ) Para ajudar com isso, você apresentará uma nova função, chamada prepare_computations (intersecção, raio),
que retornará uma nova estrutura de dados encapsulando algumas informações pré-computadas relacionadas à interseção. Isso irá ajudá-lo em capítulos posteriores (como Capítulo 11, Reflexão e Refração, na página 141 ), tornando mais fácil reutilizar esses cálculos em cálculos diferentes.

Escreva o seguinte teste, mostrando que prepare_computations () pré-calcula o ponto (no espaço do mundo) onde ocorreu a interseção, o vetor do olho (apontando para trás em direção ao olho, ou câmera) e o vetor normal.

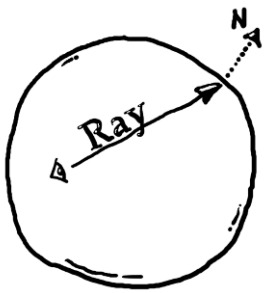
**features / intersections.feature**

```
Cenário : Pré-computando o estado de uma interseção
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E forma ← esfera ()
        E eu ← interseção (4, forma)
    Quando compor ← prepare_computations (i, r)
    Então comps.t = it
        E comps.object = i.object
        E comps.point = ponto (0, 0, -1)
        E comps.eyev = vetor (0, 0, -1)
        E comps.normalv = vetor (0, 0, -1)
```

A implementação deve parecer familiar, usando funções que você já escreveu e usou em outro lugar. Em pseudocódigo, será mais ou menos assim:

```
função prepare_computations (interseção, raio)
    # instanciar uma estrutura de dados para armazenar alguns valores pré-computados
    compor ← nova estrutura de dados de computação

    # copie as propriedades do cruzamento, por conveniência
    comps.t            ←   intersection.t
    comps.object ← intersection.object

    # pré-calcula alguns valores úteis
    comps.point        ←   posição (raio, comps.t)
    comps.eyev         ←   - ray.direction
    comps.normalv ← normal_at (comps.object, comps.point)

    Retorna compor
função final
```

Um outro caso que prepare_computations () deve tratar para este capítulo é onde o hit ocorre no *lado de dentro* de uma forma. Considere a ilustração a seguir, onde o raio se origina dentro de uma esfera.




Neste caso, os pontos normais da superfície (como atualmente computados) *um jeito* do olho. Mas se o normal estiver apontando para longe do olho, o algoritmo de sombreamento do capítulo anterior irá colorir a superfície muito mais escura do que deveria. O que fazer?

Adicione os dois testes a seguir, que mostram que prepare_computations () define um quarto atributo, lado de dentro, que será verdadeiro se o hit ocorrer dentro do objeto e falso caso contrário. Observe, também, que a normal é invertida quando a interseção está dentro de um objeto, de modo que a superfície pode ser iluminada adequadamente.

**features / intersections.feature**

```
Cenário : O acerto, quando ocorre um cruzamento do lado de fora
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E forma ← esfera ()
        E eu ← interseção (4, forma)
    Quando compor ← prepare_computations (i, r)
    Então comps.inside = false
```

```
Cenário : O acerto, quando ocorre um cruzamento no interior
    Dado r ← raio ( ponto (0, 0, 0), vetor (0, 0, 1))
        E forma ← esfera ()
        E eu ← interseção (1, forma)
    Quando compor ← prepare_computations (i, r)
    Então comps.point = ponto (0, 0, 1)
        E comps.eyev = vetor (0, 0, -1)
        E comps.inside = true
            # normal teria sido (0, 0, 1), mas está invertido!
        E comps.normalv = vetor (0, 0, -1)
```

Então, como você pode saber - matematicamente - se o normal aponta para longe do vetor do olho? Pegue o produto escalar dos dois vetores e, se o resultado for negativo, eles estão apontando em (aproximadamente) direções opostas.

```
E se ponto (comps.normalv, comps.eyev) <0
    comps.inside ← verdade
    comps.normalv ← - comps.normalv
outro
    comps.inside ← falso
fim se
```

Depois que esses testes forem aprovados, você pode prosseguir para a implementação da lógica de sombreamento real. Escreva os dois testes a seguir que chamam uma nova função, sombra_hit (mundo, comps). A função deve retornar a cor na interseção encapsulada por compor, no mundo dado.

**features / world.feature**

```
Cenário : Sombreando um cruzamento
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E forma ← o primeiro objeto em w
        E eu ← interseção (4, forma)
    Quando compor ← prepare_computations (i, r)
        E c ← sombra_hit (w, comps)
    Então c = cor (0,38066, 0,47583, 0,2855)

Cenário : Sombreando uma interseção de dentro
    Dado C ← default_world ()
        E w.light ← point_light ( ponto (0, 0,25, 0), cor (1, 1, 1))
        E r ← raio ( ponto (0, 0, 0), vetor (0, 0, 1))
        E forma ← o segundo objeto em w
        E eu ← interseção (0,5, forma)
    Quando compor ← prepare_computations (i, r)
        E c ← sombra_hit (w, comps)
    Então c = cor (0,90498, 0,90498, 0,90498)
```

Para passar em ambos os testes, seu sombra_hit () função precisa chamar o iluminação() (de O modelo de reflexão de Phong, na página 83 ) funcionam com o material do objeto interceptado e os cálculos preparados.

Em pseudocódigo, ele deve vir junto com o seguinte:

```
função sombra_hit (mundo, comps)
    Retorna iluminação (comps.object.material,
                        world.light,
                        comps.point, comps.eyev, comps.normalv)
função final
```

### Suportando várias fontes de luz

O objeto mundo descrito aqui suporta apenas uma única fonte de luz, mas não é terrivelmente difícil suportar mais de uma. Você precisaria ter certeza de que seu sombra_hit ()
função itera sobre todas as fontes de luz, chamando iluminação() para cada um e somando as cores.

Porém, esteja avisado: adicionar várias fontes de luz irá desacelerar o seu renderizador, especialmente quando você chegar a Capítulo 8, Sombras, na página 109 . Mas se você tem ciclos de CPU para queimar, ter mais de uma luz pode tornar possíveis alguns efeitos interessantes, como sombras sobrepostas.

Agora, por conveniência, amarre o intersect (), prepare_computations (), e sombra_hit () funções com um arco e chama a função resultante color_at (mundo, raio).
Ele cruzará o mundo com o raio dado e então retornará a cor na interseção resultante.

Adicione os seguintes testes para demonstrar três casos importantes. O primeiro teste mostra que quando o raio não consegue interceptar nada, a cor que é retornada deve ser preta.

**features / world.feature**

```
Cenário : A cor quando um raio falha
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, -5), vetor (0, 1, 0))
    Quando c ← color_at (w, r)
    Então c = cor (0, 0, 0)
```

Este segundo teste mostra que o sombreamento deve ser calculado apropriadamente quando o raio intercepta um objeto - neste caso, a esfera mais externa no mundo padrão.

**features / world.feature**

```
Cenário : A cor quando um raio atinge
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
    Quando c ← color_at (w, r)
    Então c = cor (0,38066, 0,47583, 0,2855)
```

O terceiro teste mostra que esperamos color_at () para usar o acertar ao calcular a cor. Aqui, colocamos o raio dentro da esfera externa, mas fora da esfera interna, e apontando para a esfera interna. Esperamos que a batida seja na esfera interna e, portanto, retorne sua cor.

```
features / world.feature
Cenário : A cor com uma interseção atrás do raio
    Dado C ← default_world ()
        E exterior ← o primeiro objeto em w
        E outer.material.ambient ← 1
        E interno ← o segundo objeto em w
        E interior.material.ambiente ← 1
        E r ← raio ( ponto (0, 0, 0,75), vetor (0, 0, -1))
    Quando c ← color_at (w, r)
    Então c = inner.material.color
```

Seu color_at () função deve fazer o seguinte:

1. Ligue intersect_world para encontrar as interseções do raio dado com o dado mundo.
2. Encontre o acertar das interseções resultantes.
3. Retorne a cor preta se não houver tal interseção. Caso contrário, pré-calcule os valores necessários
4. com prepare_computations.
5. Finalmente, ligue sombra_hit para encontrar a cor no acertar.

Isso é tudo que é necessário - por enquanto! - para o mundo. Faça seus testes passarem. Quando tudo estiver verde, começaremos a falar sobre como realmente fazer imagens desses mundos que você está construindo. A primeira etapa é uma matriz chamada de
*vista transformação.*

## Definindo uma transformação de visão

No momento, todas as suas imagens renderizadas foram pintadas em uma “tela” fixa na qual você lançou raios. Isso funciona, como você viu, mas é muito difícil mover a tela. Suponha que você queira renderizar uma imagem de algum ponto acima e à direita de um objeto. Como você orientaria a tela de forma que ainda pudesse olhar para aquele objeto?

Isso é o que *ver transformação* vai fazer por você. É uma matriz de transformação - como escala, rotação e translação - que orienta o mundo em relação ao seu olho, permitindo alinhar tudo e obter exatamente a foto de que precisa.

Agora, embora a transformação realmente oriente o *mundo,* muitas vezes é muito mais fácil imaginar que move o *olho.* Mover uma câmera é mais intuitivo do que mover o mundo na frente da câmera! Por essa razão,

nesta seção, você apresentará uma nova função, chamada view_transform (de, para, para cima), que finge que o olho se move em vez do mundo. Você especifica onde deseja que o olho esteja na cena (o a partir de parâmetro), o ponto na cena em que você deseja olhar (o para parâmetro), e um vetor indicando qual direção é acima. A função então retorna para você a matriz de transformação correspondente.

Comece escrevendo um teste usando esta nova função para descrever o mundo *orientação padrão*. A orientação padrão é a matriz que você obtém se seus parâmetros de visualização ( de para, e acima) não exige que nada seja escalado, girado ou traduzido. Em outras palavras, a orientação padrão é a matriz de identidade! O teste a seguir demonstra isso e mostra que a orientação parece desde a origem ao longo do z eixo na direção negativa, com acima no positivo y direção.

**features / transformations.feature**

```
Cenário : A matriz de transformação para a orientação padrão
    Dado a partir de ← ponto (0, 0, 0)
        E para ← ponto (0, 0, -1)
        E acima ← vetor (0, 1, 0)
    Quando t ← view_transform (de, para, para cima)
    Então t = identidade_matriz
```

Isso significa que se virar e olhar para o *positivo* z direção é como olhar em um espelho: frente e verso são trocados, e esquerda e direita são trocados. A transformação da visualização, neste caso, deve ser exatamente a mesma que refletir através do z ( da frente para trás) e x ( eixos da esquerda para a direita). Como você viu em Dimensionamento, na página 46 , a reflexão é o mesmo que dimensionar por um valor negativo, então você esperaria que a transformação da vista aqui fosse a mesma que dimensionar por (- 1, 1, -1), que é exatamente o que o teste a seguir demonstra.

**features / transformations.feature**

```
Cenário : Uma matriz de transformação de visão olhando na direção z positiva
    Dado a partir de ← ponto (0, 0, 0)
        E para ← ponto (0, 0, 1)
        E acima ← vetor (0, 1, 0)
    Quando t ← view_transform (de, para, para cima)
    Então t = escala (-1, 1, -1)
```

Em seguida, adicione o seguinte teste, que mostra que a transformação da visualização realmente move o *mundo* e não o *olho*. O teste posiciona o olho em um ponto de 8 unidades ao longo do z eixo e aponta o olho de volta para a origem.

**features / transformations.feature**

```
Cenário : A transformação da visão move o mundo
    Dado a partir de ← ponto (0, 0, 8)
        E para ← ponto (0, 0, 0)
        E acima ← vetor (0, 1, 0)
    Quando t ← view_transform (de, para, para cima)
    Então t = tradução (0, 0, -8)
```

Como você pode ver, a tradução resultante move tudo *para trás* 8 unidades ao longo do z eixo, efetivamente empurrando o mundo para longe de um olho posicionado na origem! Selvagem.

Escreva mais um teste para a transformação da visualização, desta vez em uma direção arbitrária. Deve produzir uma matriz que é uma combinação de cisalhamento, dimensionamento e translação.

**features / transformations.feature**

```
Cenário : Uma transformação de visão arbitrária
    Dado a partir de ← ponto (1, 3, 2)
        E para ← ponto (4, -2, 8)
        E acima ← vetor (1, 1, 0)
    Quando t ← view_transform (de, para, para cima)
    Então t é a seguinte matriz 4x4:
            | -0,50709 | 0,50709 | 0,67612 | -2,36643 | | 0,76772 | 0,60609 | 0,12122 | -2,82843
            | | -0,35857 | 0,59761 | -0,71714 | 0,00000 | | 0,00000 | 0,00000 | 0,00000 | 1,00000
            |
```

Observe que o acima o vetor não precisa ser normalizado. Na verdade, ele nem precisa ser exatamente perpendicular à direção de visualização. Como você verá em breve, o view_transform () função vai arrumar isso acima vetor, então você só precisa apontar vagamente na direção que deseja. Não é conveniente?

Então, como funciona essa magia negra? Dadas três entradas, de para, e acima, o algoritmo funciona assim:

1. Calcule o frente vetor subtraindo a partir de a partir de para. Normalize o resultado.
2. Calcule o deixou vetor tomando o produto vetorial de frente e a normalizado acima vetor.
3. Calcular o de verdade vetor tomando o produto vetorial de deixou e frente.
   Isso permite que o seu original acima o vetor deve estar apenas aproximadamente para cima, o que torna o enquadramento das cenas muito mais fácil, já que você não precisa usar uma calculadora para descobrir a direção exata para cima.

4. Com estes esquerda, true_up, e frente vetores, agora você pode construir uma matriz que representa o ou • transformação de identificação:

$$\text{orientação} = \begin{bmatrix} \text{deixou}_x & \text{deixou}_y & \text{deixou}_z & 0 \\ \text{de verdade}_x & \text{de verdade}_y & \text{de verdade}_z & 0 \\ -\text{frente}_x & -\text{frente}_y & -\text{frente}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. Tudo o que resta é acrescentar uma tradução a essa transformação para mover a cena no lugar antes de orientá-la. Multiplicar orientação de tradução (-de.x, - from.y, -from.z), e você é dourado!

Descrito como pseudocódigo, seu view_transform () função pode ter a seguinte aparência:

```
função view_transform (de, para, para cima)
    frente ← normalizar (para - de) upn ← normalizar (acima)

    deixou ← cruzar (para a frente, para cima) true_up ← cruzar
    (esquerda, frente)

    orientação ← matriz (left.x,        left.y,        left.z,        0,
                         true_up.x, true_up.y, true_up.z, 0,
                         - forward.x, -forward.y, -forward.z, 0,
                         0,             0,             0,             1)

    Retorna orientação * tradução (-de.x, -de.y, -de.z)
função final
```

Depois de implementar isso e fazer seus testes passarem, continue lendo! Você está pronto para conectar essa transformação de visualização em uma câmera virtual, oferecendo uma maneira mais simples de ver suas cenas.

## Implementando uma Câmera

Assim como uma câmera real, sua câmera virtual permitirá que você “tire fotos” de sua cena. Você pode movê-la, aumentar e diminuir o zoom e até mesmo girar a câmera de cabeça para baixo se essa for a foto que você deseja. A câmera é definida pelos quatro seguintes atributos:

- hsize é o tamanho horizontal (em pixels) da tela em que a imagem será renderizada.
- vsize é o tamanho vertical da tela (em pixels).
- campo de visão é um ângulo que descreve o quanto a câmera pode ver. Quando o campo de visão é pequeno, a visão será “ampliada”, ampliando uma área menor da cena.

- transformar é uma matriz que descreve como o mundo deve ser orientado em relação à câmera. Geralmente é uma transformação de visualização, como você implementou na seção anterior.

Escreva o seguinte teste, mostrando como uma câmera é construída usando um novo câmera (hsize, vsize, field_of_view) função. Também mostra que a transformação padrão para uma câmera é a matriz de identidade.

**features / camera.feature**

```
Cenário : Construindo uma câmera
    Dado hsize ← 160
        E vsize ← 120
        E campo de visão ← π / 2
    Quando c ← Câmera (hsize, vsize, field_of_view)
    Então c.hsize = 160
        E c.vsize = 120
        E c.field_of_view = π / 2
        E c.transform = identity_matrix
```

Uma das principais responsabilidades da câmera é mapear a cena tridimensional em uma tela bidimensional. Para fazer isso, você fará com que a câmera faça exatamente o que você fez nos exercícios anteriores e coloque a tela em algum lugar da cena para que os raios possam ser projetados através dela. Mas, ao contrário do que você fez antes, a tela da câmera sempre será exatamente uma unidade na frente da câmera. Como você verá em breve, isso torna a matemática um pouco mais limpa.

A primeira etapa é garantir que a câmera saiba o tamanho (em unidades de espaço mundial) dos pixels na tela. Adicione os dois testes a seguir para mostrar que o tamanho do pixel é calculado corretamente para uma tela com aspecto horizontal ( hsize> vsize),
e um com aspecto vertical ( vsize> hsize).

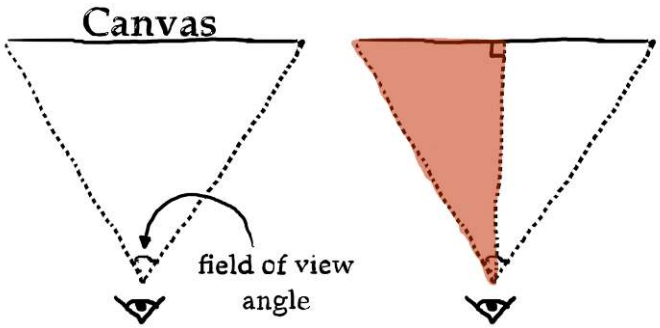
**features / camera.feature**

```
Cenário : O tamanho do pixel para uma tela horizontal
    Dado c ← Câmera (200, 125, π / 2)
    Então c.pixel_size = 0,01

Cenário : O tamanho do pixel para uma tela vertical
    Dado c ← Câmera (125, 200, π / 2)
    Então c.pixel_size = 0,01
```

O algoritmo para calcular esse valor é assim:

1. Você sabe que a tela está a uma unidade de distância e conhece o ângulo do campo de visão. Ao cortar o campo de visão pela metade, você cria um triângulo retângulo, como mostrado no figura na página 102 .

.....




A largura dessa metade da tela, então, pode ser calculada pela tangente da metade do campo de visão. Chame esse valor half_view, como na seguinte fórmula.

$$\text{meia vista} = \text{bronzeado}\ \frac{\text{campo de visão}}{2}$$

2. O *proporção da tela* é a proporção do tamanho horizontal da tela, para seu tamanho vertical. Calcule isso com a seguinte fórmula.

$$\text{aspecto} = \frac{\text{hsize}}{\text{vsize}}$$

3. Agora, se o tamanho horizontal for maior ou igual ao tamanho vertical ( aspecto ≥ 1), então half_view tem metade da largura da tela e half_view／aspecto tem metade da altura da tela.

Se o tamanho vertical for maior que o tamanho horizontal ( aspecto < 1), então half_view é ao invés metade do *altura* da tela, e metade da tela *largura* é half_view × aspect.

Chame esses dois valores meia largura e meia altura, respectivamente.

(Segure-se a estes meia largura e meia altura variáveis, por sinal. Você vai precisar deles novamente em breve ...)

4 Finalmente, calcule o tamanho de um único pixel na tela dividindo a largura total da tela ( half_width × 2) pelo tamanho horizontal (em pixels) da tela ( hsize). Chame isso pixel_size.

(Observe que a suposição aqui é que os pixels são quadrados, então você não precisa realmente calcular o tamanho vertical do pixel - ele será igual ao tamanho horizontal.)

Em pseudocódigo, deve ser parecido com isto:

half_view ← bronzeado (camera.field_of_view / 2) aspecto ← camera.hsize / camera.vsize

```
E se aspecto> = 1 então
    camera.half_width ← half_view
    camera.half_height ← half_view / aspect
outro
    camera.half_width ← half_view * aspecto camera.half_height ← half_view

fim se

camera.pixel_size ← ( camera.half_width * 2) / camera.hsize
```

Você vai usar o pixel_size e aqueles meia largura e meia altura valores que você calculou para criar raios que podem passar por qualquer pixel da tela. Implemente os três testes a seguir para garantir que funcione. Isso introduz uma nova função,

ray_for_pixel (camera, x, y), que retorna um novo raio que começa na câmera e passa pelo indicado ( x, y) pixel na tela. Os dois primeiros testes usam uma câmera não transformada para lançar raios através do centro e do canto da tela, e o terceiro tenta um raio com uma câmera que foi transladada e girada.

**features / camera.feature**

```
Cenário : Construindo um raio através do centro da tela
    Dado c ← Câmera (201, 101, π / 2)
    Quando r ← ray_for_pixel (c, 100, 50)
    Então r.origin = ponto (0, 0, 0)
        E r.direction = vetor (0, 0, -1)

Cenário : Construindo um raio através de um canto da tela
    Dado c ← Câmera (201, 101, π / 2)
    Quando r ← ray_for_pixel (c, 0, 0)
    Então r.origin = ponto (0, 0, 0)
        E r.direction = vetor (0,66519, 0,33259, -0,66851)

Cenário : Construindo um raio quando a câmera é transformada
    Dado c ← Câmera (201, 101, π / 2)
    Quando c.transform ← rotação_y ( π / 4) * tradução (0, -2, 5)
        E r ← ray_for_pixel (c, 100, 50)
    Então r.origin = ponto (0, 2, -5)
        E r.direction = vetor ( √ 2/2, 0, - √ 2/2)
```

**The Camera Transform vs. theWorld**

Observe que no último teste, a origem do raio termina em ( 0, 2, -5), apesar da transformação da câmera, incluindo uma tradução de (0, -2, 5). Isso não é um erro de digitação! Lembre-se de que a transformação da câmera descreve como o *mundo* é movido em relação ao *Câmera.*

Além disso, você está transformando tudo pelo *inverso* dessa transformação, movendo o mundo ( 0, -2, 5) é efetivamente o mesmo que mover a origem do raio na direção oposta: ( 0, 2, -5).

Agora, faça esses testes passarem. O ray_for_pixel () função deve calcular as coordenadas mundiais no *Centro* do pixel fornecido e, em seguida, constrói um raio que passa por esse ponto. Supondo duas entradas, px ( a x posição do pixel) e py ( a y posição do pixel), o pseudocódigo para o algoritmo se parece com este:

```
função ray_for_pixel (câmera, px, py)
    # o deslocamento da borda da tela até o centro do pixel
    xoffset ← ( px + 0,5) * camera.pixel_size yoffset ← ( py + 0,5) *
    camera.pixel_size

    # as coordenadas não transformadas do pixel no espaço mundial.
    # (lembre-se de que a câmera está voltada para -z, então + x está à * esquerda *.)
    world_x ← camera.half_width - xoffset world_y ← camera.half_height -
    yoffset

    # usando a matriz da câmera, transforme o ponto da tela e a origem,
    # e então calcule o vetor de direção do raio.
    # (lembre-se de que a tela está em z = -1)
    pixel ← inverso (camera.transform) * ponto (world_x, world_y, -1) origem ← inverso (camera.transform) * ponto (0, 0, 0)
    direção ← normalizar (pixel - origem)


    Retorna raio (origem, direção)
função final
```

Ok, mais uma função e você terminará com a câmera. A última parte a ser implementada é a render (câmera, mundo) função, que usa a câmera para renderizar uma imagem de um determinado mundo.

Adicione o seguinte teste à sua suíte. É uma demonstração não rigorosa de como o render () função deve funcionar. Ele renderiza o mundo padrão com uma câmera e, em seguida, garante que o pixel no meio da tela resultante seja a cor esperada.

```
features / camera.feature
Cenário : Renderizando um mundo com uma câmera
    Dado C ← default_world ()
        E c ← Câmera (11, 11, π / 2)
        E a partir de ← ponto (0, 0, -5)
        E para ← ponto (0, 0, 0)
        E acima ← vetor (0, 1, 0)
        E c.transform ← view_transform (de, para, para cima)
    Quando imagem ← render (c, w)
    Então pixel_at (imagem, 5, 5) = cor (0,38066, 0,47583, 0,2855)
```

Você provavelmente descobrirá que a implementação dessa função se parece muito com o código que você já escreveu. Quando você renderizou uma imagem no final de Capítulo 5, Intersecções Ray-Sphere, na página 57 , você criou uma tela e

lance um raio através de cada um de seus pixels, colorindo os pixels com as cores das interseções correspondentes. Isso é exatamente o que esta função fará, exceto em vez de computar a localização de cada pixel, você deixará seu novo ray_for_pixel ()
função fazer o trabalho.

Em pseudocódigo, é assim:

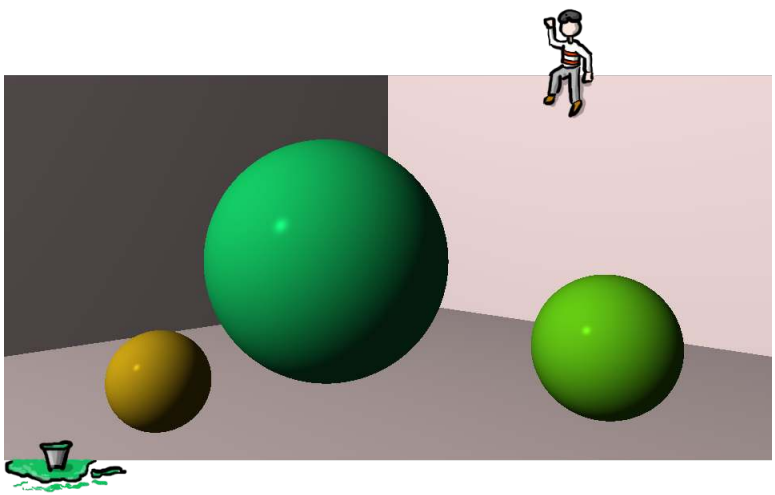
```
função render (câmera, mundo)
    imagem ← tela de pintura (camera.hsize, camera.vsize)

    pra y ← 0 a camera.vsize - 1
        pra x ← 0 a camera.hsize - 1
            raio ← ray_for_pixel (câmera, x, y) cor ← color_at (mundo, raio)


            write_pixel (imagem, x, y, cor)
        fim para
    fim para

    Retorna imagem
função final
```

Vá em frente e certifique-se de que todos os seus testes estão passando. Depois que tudo funcionar, vamos encerrar este capítulo com um pequeno projeto que usa seu novo mundo e código de câmera.

## Juntar as peças

Reveja o programa que você escreveu no final do capítulo anterior. É hora de limpar isso, aproveitando o mundo e a câmera que você acabou de escrever e adicionando mais algumas esferas para tornar a cena mais interessante.

Aqui está um exemplo do que você pode construir:



Este foi construído a partir de seis esferas, dispostas da seguinte forma:

1. O piso é uma esfera extremamente achatada com uma textura fosca.

```
piso ← esfera ()
floor.transform ← escala (10, 0,01, 10) chão.material ← material ()

floor.material.color ← cor (1, 0,9, 0,9) floor.material.specular ← 0
```

2. A parede à esquerda tem a mesma escala e cor do chão, mas também é girada e transladada ao lugar.

```
left_wall ← esfera ()
left_wall.transform ← tradução (0, 0, 5) *
                                    rotação_y (- π / 4) * rotação_x ( π / 2) *
                                    escala (10, 0,01, 10)
left_wall.material ← floor.material
```

Observe a ordem em que as transformações são multiplicadas: a parede precisa ser dimensionada e, em seguida, girada x, então girado em y, e traduzido por último, para que as transformações sejam multiplicadas no *ordem reversa!*

3. A parede à direita é idêntica à parede esquerda, mas é girada na direção oposta em y.

```
parede direita ← esfera ()
right_wall.transform ← tradução (0, 0, 5) *
                                    rotação_y ( π / 4) * rotação_x ( π / 2) *
                                    escala (10, 0,01, 10)
right_wall.material ← floor.material
```

4. A grande esfera no meio é uma esfera unitária, levemente transladada para cima e colorida de verde.

```
meio ← esfera ()
middle.transform ← tradução (-0,5, 1, 0,5) material do meio ← material ()

middle.material.color ← cor (0,1, 1, 0,5) middle.material.diffuse ← 0,7

middle.material.specular ← 0,3
```

5. A esfera verde menor à direita é redimensionada pela metade.

```
direita ← esfera ()
right.transform ← tradução (1,5, 0,5, -0,5) * escala (0,5, 0,5, 0,5) direito. Material ← material ()

right.material.color ← cor (0,5, 1, 0,1) right.material.diffuse ← 0,7

right.material.specular ← 0,3
```

6. A menor esfera é dimensionada em um terço, antes de ser transladada.

```
deixou ← esfera ()
left.transform ← tradução (-1,5, 0,33, -0,75) * escala (0,33, 0,33, 0,33) material esquerdo ← material ()

left.material.color ← cor (1, 0,8, 0,1) left.material.diffuse ← 0,7

left.material.specular ← 0,3
```

A fonte de luz é branca, brilhando de cima e para a esquerda:

```
world.light_source ← point_light ( ponto (-10, 10, -10), cor (1, 1, 1))
```

E a câmera é configurada assim:

```
Câmera ← Câmera (100, 50, π / 3) camera.transform ← view_transform ( ponto (0, 1,5, -5),
                                                                    ponto (0, 1, 0),
                                                                    vetor (0, 1, 0))

# renderiza o resultado em uma tela.
tela de pintura ← render (câmera, mundo)
```

Faça experiências com outras cores e propriedades materiais. Tente deformar as esferas com transformações de dimensionamento, rotação e cisalhamento. Adicione mais esferas. Mova a câmera, experimente diferentes campos de visão e veja o que acontece quando você muda a direção do acima vetor!

Você provavelmente descobrirá que o seu renderizador é *lento,* portanto, use resoluções menores enquanto faz experiências. Salve as renderizações de alta resolução para as versões finais de sua cena, quando você tiver tudo organizado e iluminado exatamente como deseja, e pode esperar dez ou quinze minutos (ou mais!) Para que seu programa chegue a um milhão pixels ou assim.

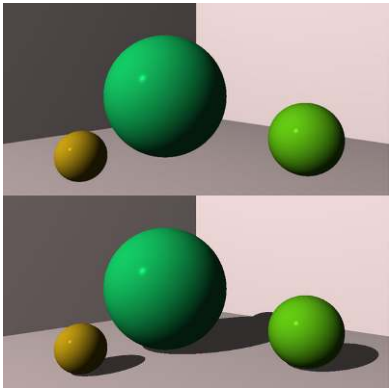
Assim que terminar de brincar com isso, vire a página. Você está prestes a adicionar suporte para sombras ao seu renderizador, o que fará maravilhas com o realismo de suas cenas.

CAPÍTULO 8

# Sombras

Seu ray tracer está realmente começando a se encaixar. Basta olhar para isso! Você tem esferas, sombras realistas, uma câmera poderosa e um mundo que suporta cenas com muitos objetos.

É uma pena que esses objetos não projetem sombras, no entanto. As sombras adicionam um toque delicioso de realismo a uma cena. Confira a figura a seguir que mostra a mesma cena com e sem sombras:



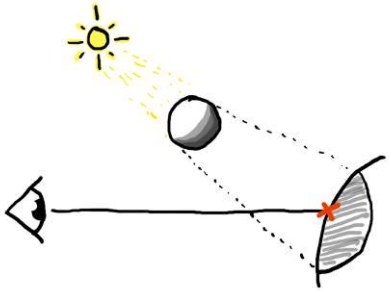
Seu cérebro usa essas sombras como pistas para a percepção de profundidade. Sem sombras, a imagem parece artificial e superficial, e isso nunca vai funcionar.

Assim, chegou a hora de adicionar sombras, e a melhor parte é que você já escreveu a maior parte da infraestrutura para dar suporte a isso. O primeiro passo é ajustar seu iluminação() função para lidar com o caso em que um ponto está na sombra. Em seguida, você implementará um novo método para determinar se um ponto está na sombra ou não e, por último, amarrará essas peças para que o seu ray tracer realmente renderize as sombras.

Vamos investigar isso!

## Iluminação nas sombras

Dado algum ponto, você pode saber que ele está na sombra se houver outro objeto sentado entre ele e a fonte de luz, como mostrado na figura a seguir.



A fonte de luz não pode contribuir com nada para esse ponto. Reserve um momento e lembre-se de como o seu iluminação() função funciona, de O modelo de reflexão de Phong, na página 83 . O componente difuso depende do vetor para a fonte de luz, e o componente específico depende do vetor de reflexão. Visto que ambos os componentes dependem da fonte de luz, o iluminação() a função deve ignorá-los quando o ponto está na sombra e usar apenas o componente do ambiente.

Adicione o seguinte teste aos outros que você escreveu para o iluminação() função. É idêntico ao intitulado "Iluminação com o olho entre a luz e a superfície" na página 86 , onde os componentes especular e difuso estavam em seus valores máximos, mas desta vez você vai passar um novo argumento para o iluminação() função que indica que o ponto está na sombra. Deve fazer com que os componentes difusos e especulares sejam ignorados, resultando no valor do ambiente sozinho contribuindo para a iluminação.

(Lembre-se de que o m e posição variáveis sendo passadas para o iluminação() funções são definidas no Bloco de "fundo" na página 86 .)

**features / materials.feature**

```
Cenário : Iluminação com a superfície na sombra
    Dado Eyev ← vetor (0, 0, -1)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 0, -10), cor (1, 1, 1))
        E na sombra ← verdade
    Quando resultado ← iluminação (m, luz, posição, eyev, normalv, in_shadow)
    Então resultado = cor (0,1, 0,1, 0,1)
```

Você pode precisar corrigir seus outros testes para acomodar a adição desse novo parâmetro. Vá em frente e resolva isso e, em seguida, faça com que este novo teste passe também, fazendo seu iluminação() função ignorar os componentes especulares e difusos quando na sombra é verdade.

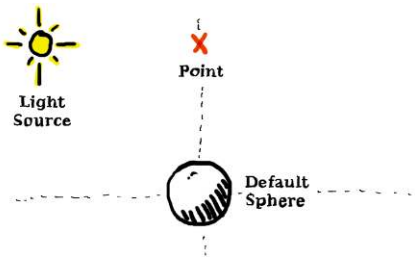
Assim que as coisas estiverem passando novamente, vamos ensinar ao seu ray tracer como saber quando um ponto está na sombra.

## Teste de sombras

Um ray tracer calcula sombras lançando um raio, chamado de *raio de sombra,* de cada ponto de intersecção em direção à fonte de luz. Se algo cruzar esse raio de sombra entre o ponto e a fonte de luz, o ponto será considerado na sombra. Você vai escrever uma nova função, is_shadowed (mundo, ponto), que fará exatamente isso.

Implemente os quatro testes a seguir, que demonstram quatro cenários diferentes. Cada um assume a existência do mundo padrão que foi definido em Construindo um Mundo, na página 92 .

No primeiro teste, o mundo é configurado como a figura a seguir.

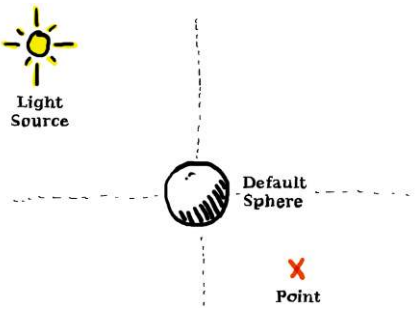



Nada se encontra ao longo da linha que conecta o ponto e a fonte de luz e, portanto, o ponto não deve estar na sombra.

**features / world.feature**

```
Cenário : Não há sombra quando nada é colinear com o ponto e a luz
    Dado C ← default_world ()
        E p ← ponto (0, 10, 0)
    Então is_shadowed (w, p) é falso
```

No segundo teste, o ponto é colocado do outro lado das esferas do mundo padrão, colocando-as entre ele e a fonte de luz, assim:

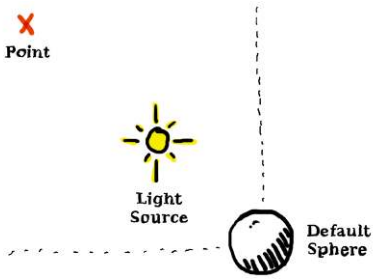


O ponto deve estar na sombra projetada pelas esferas.

**features / world.feature**

```
Cenário : A sombra quando um objeto está entre o ponto e a luz
    Dado C ← default_world ()
        E p ← ponto (10, -10, 10)
    Então is_shadowed (w, p) é verdade
```

O próximo teste posiciona o ponto de forma que a luz fique entre ele e as esferas.

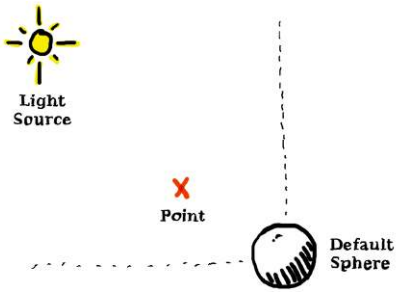



Mais uma vez, o ponto não deve estar na sombra, porque nada existe entre o ponto e a luz.

**features / world.feature**

```
Cenário : Não há sombra quando um objeto está atrás da luz
    Dado C ← default_world ()
        E p ← ponto (-20, 20, -20)
    Então is_shadowed (w, p) é falso
```

O último teste é semelhante, mas posiciona o ponto entre a luz e as esferas, assim:




E, novamente, mesmo nessa configuração, nada fica entre a luz e o ponto, então o ponto ainda não está sombreado.

**features / world.feature**

```
Cenário : Não há sombra quando um objeto está atrás do ponto
    Dado C ← default_world ()
        E p ← ponto (-2, 2, -2)
    Então is_shadowed (w, p) é falso
```

O algoritmo para is_shadowed () é assim:

1. Meça a distância de ponto para a fonte de luz subtraindo ponto da posição da luz e tomando a magnitude do vetor resultante. Chame isso distância.

2 Crie um raio de ponto em direção à fonte de luz normalizando o vetor da etapa 1.

3 Cruze o mundo com aquele raio.

4 Verifique se houve um hit e, em caso afirmativo, se t é menos do que distância. Nesse caso, o impacto fica entre o ponto e a fonte de luz, e o ponto fica na sombra.

Em pseudocódigo, pode ser assim:

```
função is_shadowed (mundo, ponto)
    v ← world.light.position - distância do ponto ← magnitude (v)

    direção ← normalizar (v)

    r ← raio (ponto, direção) intersecções ← intersect_world (mundo, r)

    h ← acertar (cruzamentos)
    E se h está presente e ht <distância
        Retorna verdade
    outro
        Retorna falso
    fim se
função final
```

Lembre-se de Identificando sucessos, na página 64 , que o acertar() função retorna a interseção com o mais baixo *não negativo* t valor. Assim, o hit t nunca será negativo, então você não precisa se preocupar em verificar se há interseções que ocorram atrás do ponto.

Implemente essa função, faça esses testes passarem e siga em frente. Só mais uma coisa precisa mudar para realmente renderizar essas sombras!

## Renderizando sombras

A parte final para realmente renderizar as sombras requer uma pequena mudança em seu sombra_hit () função de Construindo um Mundo, na página 92 . Você precisa verificar se o ponto está na sombra ou não e, em seguida, passar esse estado para o seu iluminação() função.

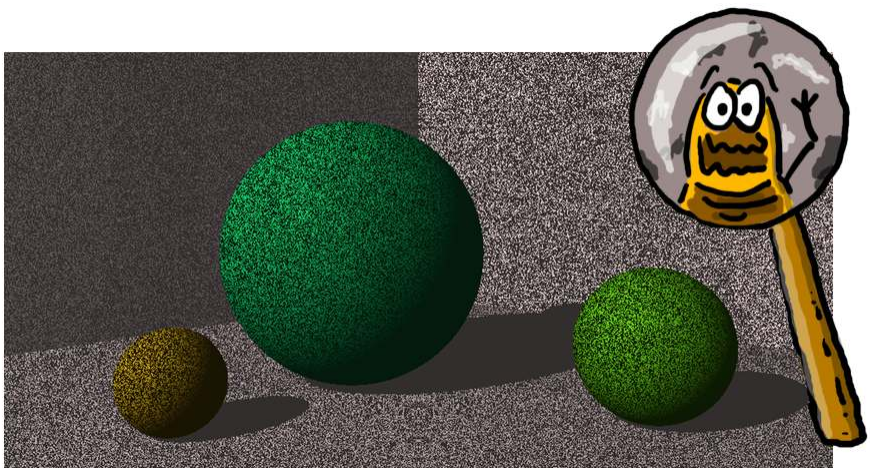
Adicione o seguinte teste àqueles que você escreveu para o sombra_hit () função. Para demonstrar o caso em que algum objeto está sombreando o ponto de intersecção, ele cria um mundo e duas esferas, e posiciona uma luz para que o segundo

esfera está na sombra da primeira. Em seguida, um raio e uma interseção são criados de forma que o ponto de interseção fique na sombra. O sombra_hit () função deve retornar apenas a cor ambiente da segunda esfera neste caso.

**features / world.feature**

```
Cenário : sombra_hit () recebe uma interseção na sombra
    Dado C ← mundo ()
        E w.light ← point_light ( ponto (0, 0, -10), cor (1, 1, 1))
        E s1 ← esfera ()
        E s1 é adicionado a w
        E s2 ← esfera () com:
            | transformar | tradução (0, 0, 10) |
        E s2 é adicionado a w
        E r ← raio ( ponto (0, 0, 5), vetor (0, 0, 1))
        E eu ← interseção (4, s2)
    Quando compor ← prepare_computations (i, r)
        E c ← sombra_hit (w, comps)
    Então c = cor (0,1, 0,1, 0,1)
```

Agora, fazer este teste passar pode parecer apenas uma questão de pegar o ponto de interseção e enviá-lo diretamente para o is_shadowed () função. Mas, se você fizer isso, poderá acabar com uma imagem renderizada que parece ter sido atacada por pulgas, como na figura a seguir.



Este efeito é chamado *acne,* e isso acontece porque os computadores não podem representar números de ponto flutuante com muita precisão. Em geral eles funcionam bem, mas devido a erros de arredondamento, será impossível dizer *exatamente* onde um raio cruza uma superfície. A resposta que você obterá será próxima - geralmente dentro de uma pequena margem de erro - mas essa manobra às vezes é *Apenas o suficiente* para fazer com que o ponto calculado de intersecção fique *abaixo* a superfície real da esfera.

Como resultado, o raio de sombra cruza a própria esfera, fazendo com que a esfera projete uma sombra em seu próprio ponto de intersecção. Obviamente, isso não é o ideal.

A solução é ajustar o ponto ligeiramente na direção do normal, antes de testar as sombras. Isso irá empurrá-lo acima da superfície e evitar a auto-sombra.

Adicione o seguinte teste, que configura uma esfera e uma interseção de forma que a interseção ocorra em z = 0. Depois de ligar para o prepare_computations () função que você escreveu em Capítulo 7, Fazendo uma cena, na página 91 , deve haver um novo atributo, over_point, que será quase idêntico a ponto, com o z componente ligeiramente menor que z = 0.

**features / intersections.feature**

```
Cenário : O acerto deve compensar o ponto
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E forma ← esfera () com:
            | transformar | tradução (0, 0, 1) |
        E eu ← interseção (5, forma)
    Quando compor ← prepare_computations (i, r)
    Então comps.over_point.z <-EPSILON / 2
        E comps.point.z> comps.over_point.z
```

Observe que o teste compara o over_point de z componente para metade de - EPSILON para ter certeza de que o ponto foi ajustado na direção correta.

Em pseudocódigo, seu prepare_computations () função precisará fazer algo assim:

```
# depois de computar e (se apropriado) negar
# o vetor normal ...
comps.over_point ← comps.point + comps.normalv * EPSILON
```

EPSILON é o pequeno número discutido em Comparando Números de Ponto Flutuante, na página 5 , e é usado aqui para empurrar o ponto um pouco na direção do normal.

Em seguida, modifique seu sombra_hit () função para que invoque is_shadowed () com o novo deslocamento do hit over_point atributo e, em seguida, chame o iluminação() função (novamente com over_point) com o resultado. Ficará assim em pseudocódigo:

```
função sombra_hit (mundo, comps)
    sombreado ← is_shadowed (mundo, comps.over_point)

    Retorna iluminação (comps.object.material,
                        world.light,
                        comps.over_point, comps.eyev, comps.normalv,
                        sombreado)
função final
```

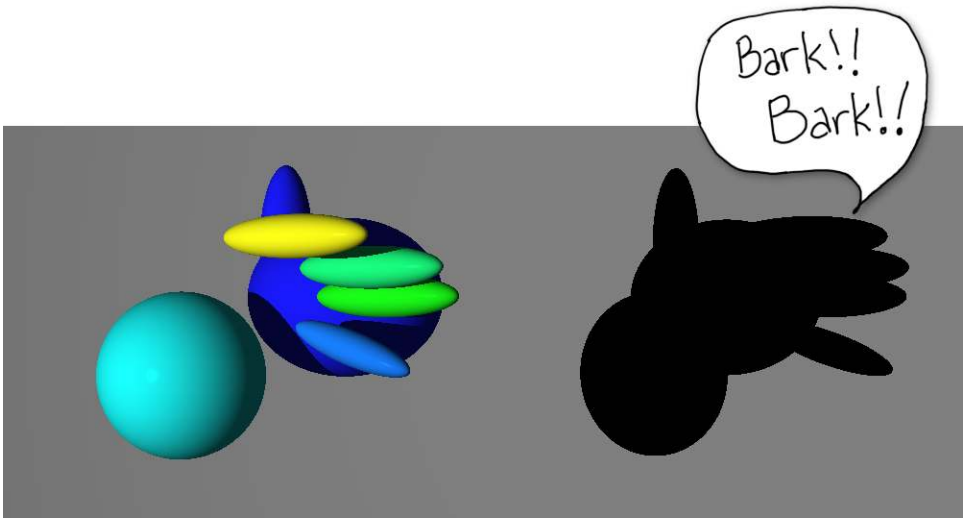
Vá em frente e faça essa mudança em seu sombra_hit () função e certifique-se de que todos os seus testes estão passando. Assim que estiverem, será hora de encerrar este capítulo e renderizar algumas sombras!

## Juntar as peças

Seu código está escrito. Seus testes estão passando. É hora de ver como essas sombras ficam na prática.

Comece com o programa que você escreveu no final do último capítulo. Se você configurá-lo para duplicar a cena no livro, deverá ver cada uma dessas esferas coloridas projetando sombras agora! Se você projetou sua própria cena, pode ou não precisar mover as coisas de modo que sombras sejam projetadas em outros objetos; faça as alterações necessárias até que você possa demonstrar que as sombras estão realmente sendo renderizadas.

Então, comece a jogar! Deforme suas esferas e observe as sombras se deformarem de acordo. Simule um eclipse posicionando uma esfera menor entre uma maior e a fonte de luz. Se você se sente particularmente ambicioso, veja se consegue fazer alguns fantoches de sombra deformando e transladando esferas!

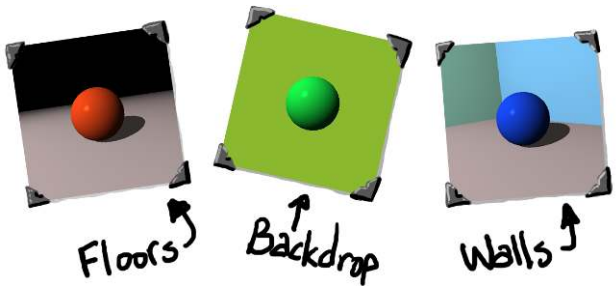



Depois de extrair toda a diversão que você pode de lançar sombras, siga em frente! É hora de adicionar outra primitiva gráfica para unir suas esferas: o avião.

CAPÍTULO 9

# Aviões

Você foi capaz de realizar muitas coisas até agora usando nada além de esferas como primitivas gráficas, o que é incrível. O mundo consiste em muito mais do que apenas esferas - até mesmo esferas habilmente transformadas. Neste capítulo, você adicionará um novo primitivo gráfico - o plano - que será perfeito para modelar pisos, paredes e fundos.




O maior obstáculo inicial provavelmente será refatorar seu código para suportar diferentes tipos de primitivos gráficos. Começaremos o capítulo falando sobre como você pode fazer essa refatoração e identificar a funcionalidade que todas as primitivas terão em comum. Assim que a funcionalidade comum for fatorada e você tiver seu conjunto de testes atualizado, passaremos para a implementação real dos aviões.

Primeiro, refatoração!

## Formas de Refatoração

Você pode ou não ter usado uma linguagem de programação orientada a objetos até agora para construir seu ray tracer. Honestamente, isso realmente não importa! Mas, uma vez que precisamos de algum tipo de vocabulário comum para descrever a próxima refatoração, vamos apenas concordar em usar termos como "classes", "objetos", "pais" e "herança". Traduza esses conceitos em seus próprios ambientes de acordo.

O objetivo desta próxima etapa é levar o seu Esfera implementação, identifique a funcionalidade que será comum a todas as formas e refatore esses bits em um pai abstrato do qual todas as outras formas herdarão. Uma vez que os bits comuns foram movidos para o pai abstrato, você simplificará seu Esfera

implementação, herdando-o desse pai.

Então, o que todas as formas terão em comum? Aqui está uma lista com a qual você pode começar:

- Todas as formas têm uma matriz de transformação. A menos que explicitamente definido, esta será a matriz de identidade, conforme descrito em Capítulo 5, Intersecções Ray-Sphere, na página 57 .
- Todas as formas têm um material, que deve ser padronizado para aquele descrito em O modelo de reflexão de Phong, na página 83 .
- Ao cruzar a forma com um raio, todas as formas precisam primeiro converter o raio em espaço do objeto, transformando-o pelo inverso da matriz de transformação da forma.
- Ao calcular o vetor normal, todas as formas precisam primeiro converter o ponto no espaço do objeto, multiplicando-o pelo inverso da matriz de transformação da forma. Então, depois de calcular a normal, eles devem transformá-la pelo inverso da transposta da matriz de transformação e então normalizar o vetor resultante antes de retorná-lo.

Capítulos posteriores, como Capítulo 14, Grupos, na página 193 , e Capítulo 16, Geometria Sólida Construtiva (CSG), na página 227 , será adicionado a essa lista, mas esses quatro itens são tudo com que você precisa se preocupar agora.

Comece escrevendo alguns testes que descrevem como essa refatoração deve ficar quando estiver concluída. Como isso vai depender muito de sua linguagem de programação e de como você arquitetou as coisas até agora, considere os seguintes testes como diretrizes - ideias de como construir seus próprios testes.

Cada um dos seguintes testes assume que existe uma função chamada test_shape (), que existe apenas para demonstrar os comportamentos abstratos do Forma aula. Como Forma em si é abstrato, o test_shape () função instancia e retorna uma subclasse especial de Forma nós ligaremos TestShape, que implementa comportamento apenas o suficiente para ser concreto. (Falaremos sobre o que isso significa, especificamente, em um momento.)

Primeiro, escreva alguns testes que mostrem que uma forma tem uma transformação padrão e que a transformação pode ser atribuída. Estes substituem os testes chamados "Transformação padrão de uma esfera" e "Alterando a transformação padrão de uma esfera" (de os cenários da esfera na página 69 ) e são essencialmente idênticos a eles, apenas chamando test_shape () ao invés de esfera().

**features / shapes.feature**

```
Cenário : A transformação padrão
    Dado s ← test_shape ()
    Então s.transform = identity_matrix

Cenário : Atribuindo uma transformação
    Dado s ← test_shape ()
    Quando set_transform (s, tradução (2, 3, 4))
    Então s.transform = tradução (2, 3, 4)
```

Adicione mais alguns testes agora, mostrando que uma forma tem um material padrão e que o material também pode ser atribuído. Estes substituem os testes chamados "Uma esfera tem um material padrão" e "Uma esfera pode ser atribuída a um material" (de os cenários da esfera na página 85 )

.....

**features / shapes.feature**

```
Cenário : O material padrão
    Dado s ← test_shape ()
    Quando m ← material
    Então m = material ()

Cenário : Atribuindo um material
    Dado s ← test_shape ()
        E m ← material ()
        E m.ambient ← 1
    Quando material ← m
    Então s.material = m
```

Em seguida, teste o comportamento do cruzar (raio, forma) função, que agora é *resumo,*
o que significa que depende de uma implementação concreta separada para dar corpo ao comportamento e realmente realizar a interseção. Porém, tudo o que você realmente precisa verificar aqui é se o raio é transformado antes de ser passado para a implementação concreta.

Uma maneira possível de implementar isso (e que os testes a seguir assumem) é declarar um local_intersect (forma, local_ray) função para cada subclasse concreta de Forma. O resumo intersectar () função transforma o raio e, em seguida, chama
local_intersect () com aquele raio transformado, retornando a coleção resultante de interseções. O pseudocódigo a seguir mostra como pode ser:

```
função cruzar (forma, raio)
    local_ray ← transformar (raio, inverso (shape.transform))
    Retorna local_intersect (forma, local_ray)
função final
```

Para os propósitos desses testes, você realmente não se importa se alguma interseção ocorre ou não, uma vez que a forma de teste não tem existência real. Tudo que você precisa saber é se o local_ray parâmetro para local_intersect () foi transformado apropriadamente. Uma maneira de fazer isso é ter a implementação da forma de teste

de local_intersect () atribuir local_ray para uma variável em algum lugar (talvez como uma variável de instância ou uma variável global), que seus testes podem inspecionar.

Os dois testes a seguir assumem a existência de uma nova propriedade na forma de teste, save_ray, qual a forma de teste local_intersect () função deve ser definida para o raio parâmetro. Esses testes são baseados em (e substituem) os testes chamados "Cruzando uma esfera em escala com um raio" e "Cruzando uma esfera traduzida com um raio" (de os cenários da esfera na página 69 )

**features / shapes.feature**

```
Cenário : Cruzando uma forma em escala com um raio
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← test_shape ()
    Quando set_transform (s, escala (2, 2, 2))
        E xs ← cruzar (s, r)
    Então s.saved_ray.origin = ponto (0, 0, -2,5)
        E s.saved_ray.direction = vetor (0, 0, 0,5)

Cenário : Cruzando uma forma traduzida com um raio
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E s ← test_shape ()
    Quando set_transform (s, tradução (5, 0, 0))
        E xs ← cruzar (s, r)
    Então s.saved_ray.origin = ponto (-5, 0, -5)
        E s.saved_ray.direction = vetor (0, 0, 1)
```

O último pedaço de lógica comum está no normal_at (esfera, ponto) função, de Calculando o Normal em uma Esfera, na página 77 . O objetivo aqui é fazer com que formas concretas individuais não precisem se preocupar com a transformação de pontos ou normais - tudo o que eles precisam fazer é calcular o próprio normal.

Pegando emprestado a mesma estratégia que foi apresentada para o intersectar () função, você pode considerar a criação de um local_normal_at (forma, local_point) para cada subclasse concreta, que aceita um ponto no espaço local (objeto) e retorna o normal no mesmo espaço. O normal_at (forma, ponto) torna-se generalizado, de modo que ele transforma o ponto, invoca o apropriado local_normal_at () função, transforma o normal resultante e o retorna. Em pseudocódigo, pode ser assim:

```
função normal_at (forma, ponto)
    local_point ← inverso (shape.transform) * point local_normal ← local_normal_at (forma, local_point) world_normal ← transpor ( inverso
    (shape.transform)) * local_normal world_normal.w ← 0



    Retorna normalizar (world_normal)
função final
```

Os dois testes a seguir substituem os chamados “Calculando o normal em uma esfera traduzida” e “Calculando o normal em uma esfera transformada”

(de os cenários da esfera na página 80 ) Isso demonstra que a translação não afeta o normal, mas a escala e a rotação afetam. Para as formas de teste
local_normal_at () função, faça-a converter o ponto em questão em um vetor:

$$\text{local normal em } (_p) = \text{vetor}(\, p_x, p_y, p_z)$$

Isso será suficiente para você testar se o comportamento está correto.

**features / shapes.feature**

```
Cenário : Calculando o normal em uma forma traduzida
    Dado s ← test_shape ()
    Quando set_transform (s, tradução (0, 1, 0))
        E n ← normal_at (s, ponto (0, 1,70711, -0,70711))
    Então n = vetor (0, 0,70711, -0,70711)

Cenário : Calculando o normal em uma forma transformada
    Dado s ← test_shape ()
        E m ← escala (1, 0,5, 1) * rotação_z ( π / 5)
    Quando set_transform (s, m)
        E n ← normal_at (s, ponto (0, √ 2/2, - √ 2/2))
    Então n = vetor (0, 0,97014, -0,24254)
```

Esse é o último comportamento comum que precisa ser mudado! Tudo o que falta para terminar a refatoração é organizar sua implementação de esfera para que ela use essa nova abstração de forma.

A seguinte lista de verificação pode ajudá-lo aqui:

1. Se possível, considere escrever um teste para verificar se um Esfera é um Forma. Esta informa em um toque que cada esfera terá os comportamentos comuns de todas as formas.

2. Remova os testes de transformação e de material de seu conjunto de esferas. Esses agora estão sendo verificados nos testes pertencentes à classe pai abstrata. Mude o existente da sua esfera intersectar () testes
3. para invocar a esfera local_intersect () em vez de. Você não precisa testar o intersectar () função, porque você já demonstrou que intersectar () chamadas local_intersect () e que o raio é transformado apropriadamente.

4. Da mesma forma, altere o existente normal_at () testes para que eles chamem local_normal_at ().
5. Escreva a sua esfera local_intersect () e local_normal_at () funções.

Você provavelmente descobrirá que conseguirá remover um pouco de código de seus testes de esfera e da própria implementação da esfera. Este é um motivo de comemoração! Depois que todos os testes estiverem passando, você deve reservar um momento para fazer uma pequena dança da vitória. Ou compre um pouco de sorvete. O que te fizer mais feliz.

Independentemente de como você comemora, quando estiver pronto, continue lendo! É hora de usar essa base refatorada para descrever um plano.

## Implementando um Avião

UMA *plano* é uma superfície perfeitamente plana que se estende infinitamente em duas dimensões. Para simplificar, seu ray tracer implementará um plano em xz - isto é, estendendo-se infinitamente longe em ambos x e z dimensões, passando pela origem. Usando matrizes de transformação, porém, você será capaz de girar e traduzir seus planos em qualquer orientação que desejar.

Como um plano não tem curvatura, seu vetor normal é constante em todos os lugares - ele não muda. Cada ponto no plano tem a mesma normal: vetor (0,
1, 0). Isso significa que implementar o local_normal_at () função para o avião é bastante desinteressante! Adicione o seguinte teste para verificar o vetor normal esperado para alguns pontos arbitrários no plano. Assume que o plano()

função retorna um novo plano.

**features / planes.feature**

```
Cenário : O normal de um avião é constante em todos os lugares
    Dado p ← plano ()
    Quando n1 ← local_normal_at (p, ponto (0, 0, 0))
        E n2 ← local_normal_at (p, ponto (10, 0, -10))
        E n3 ← local_normal_at (p, ponto (-5, 0, 150))
    Então n1 = vetor (0, 1, 0)
        E n2 = vetor (0, 1, 0)
        E n3 = vetor (0, 1, 0)
```

A lógica para interceptar um raio com um plano é o único outro bit que precisa ser implementado e tem quatro casos a serem considerados:

1. O raio é paralelo ao plano e, portanto, nunca o cruzará.
2. O raio é *coplanar* com o plano, o que quer dizer que a origem do raio é no plano, e a direção do raio é paralela ao plano. Você está vendo o avião de lado. Nesse caso, cada ponto do raio intercepta o plano, resultando em um número infinito de interseções. Isso é pesado! Mas como um plano é infinitamente fino, ele é invisível quando visto dessa forma, então vamos supor que o raio falhe neste caso.

3 A origem do raio está acima do plano. A origem
4 do raio está abaixo do plano.

Teste os dois primeiros casos escrevendo os dois testes a seguir. Cada um configura um plano e um raio com uma direção paralela ao plano. Em ambos os casos, local_intersect ()
deve retornar um conjunto vazio de interseções.

**features / planes.feature**

```
Cenário : Faça a intersecção com um raio paralelo ao plano
    Dado p ← plano ()
        E r ← raio ( ponto (0, 10, 0), vetor (0, 0, 1))
```

```
    Quando xs      ←    local_intersect (p, r)
    Então xs       está vazia

Cenário :          Cruze com um raio coplanar
    Dado p         ←    plano ()
        E r        ←    raio ( ponto (0, 0, 0), vetor (0, 0, 1))
    Quando xs      ←    local_intersect (p, r)
    Então xs       está vazia
```

Para saber se um raio é paralelo ao plano, você precisa observar que o plano está em xz - não tem inclinação em y de forma alguma. Assim, se o vetor de direção do seu raio também não tem inclinação em y ( Está y componente é 0), é paralelo ao plano. Na prática, você vai querer tratar qualquer número minúsculo como 0 para esta comparação, como mostra o pseudocódigo a seguir (usando EPSILON como o limite para "número minúsculo"):

```
função local_intersect (plano, raio)
    E se abdômen (ray.direction.y) <EPSILON
        Retorna () # conjunto vazio - sem interseções
    fim se

    # lógica de interseção restante vai aqui
função final
```

Implemente os próximos dois testes para definir o comportamento do local_intersect () função, testando especificamente a lógica de interseção restante. O primeiro verifica o caso de um raio cruzando um plano de cima, e o segundo verifica uma interseção de baixo.

**features / planes.feature**

```
Cenário : Um raio cruzando um plano de cima
    Dado p ← plano ()
        E r ← raio ( ponto (0, 1, 0), vetor (0, -1, 0))
    Quando xs ← local_intersect (p, r)
    Então xs.count = 1
        E xs [0] .t = 1
        E xs [0] .objeto = p

Cenário : Um raio cruzando um plano de baixo
    Dado p ← plano ()
        E r ← raio ( ponto (0, -1, 0), vetor (0, 1, 0))
    Quando xs ← local_intersect (p, r)
    Então xs.count = 1
        E xs [0] .t = 1
        E xs [0] .objeto = p
```

Para fazer isso, você precisará implementar a seguinte fórmula para calcular a interseção de um raio com um plano. Observe que esta fórmula só funciona se o plano for como descrito acima - em xz, com o normal apontando para o positivo y direção.

$$t = \frac{-\text{origem}_y}{\text{direção}_y}$$

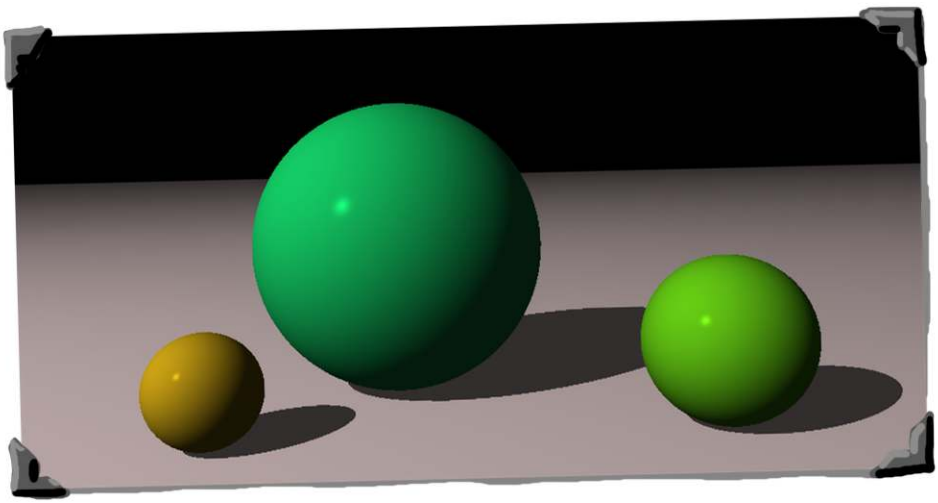
A variável origem é a origem do raio, e direção é o vetor de direção do raio. O seguinte pseudocódigo mostra como o completo local_intersect () função pode parecer.

```
função local_intersect (raio, plano)
    E se abdômen (ray.direction.y) <EPSILON
        Retorna () # conjunto vazio - sem interseções
    fim se

    t ← - ray.origin.y / ray.direction.y
    Retorna ( interseção (t, plano))
função final
```

Vá em frente e faça seus testes passarem, agora. Depois de estabilizar as coisas novamente, conclua com o seguinte projeto curto para testar seu mais novo primitivo gráfico.

## Juntar as peças

Escreva uma pequena cena consistindo em um único plano como o chão e uma ou duas esferas sobre ele. Por exemplo, aqui estão as mesmas três esferas dos capítulos anteriores, sentadas em um avião:



Outras coisas que você pode tentar:

- Adicione uma parede como pano de fundo girando-a $\pi/2$ radianos ao redor do $x$ eixo e traduzindo algumas unidades no positivo $z$ direção.

- Faça uma sala em formato hexagonal girando e transladando os planos cuidadosamente e, em seguida, posicione a câmera de cima, olhando para baixo, para que você possa ver a geometria em ação.

- Adicione um teto transladando outro plano verticalmente, em $y$. ( Tenha cuidado para posicionar sua fonte de luz abaixo do teto!)

- Em vez de exibir uma esfera inteira no topo do plano, translade a esfera para que ela fique parcialmente embutida no plano.

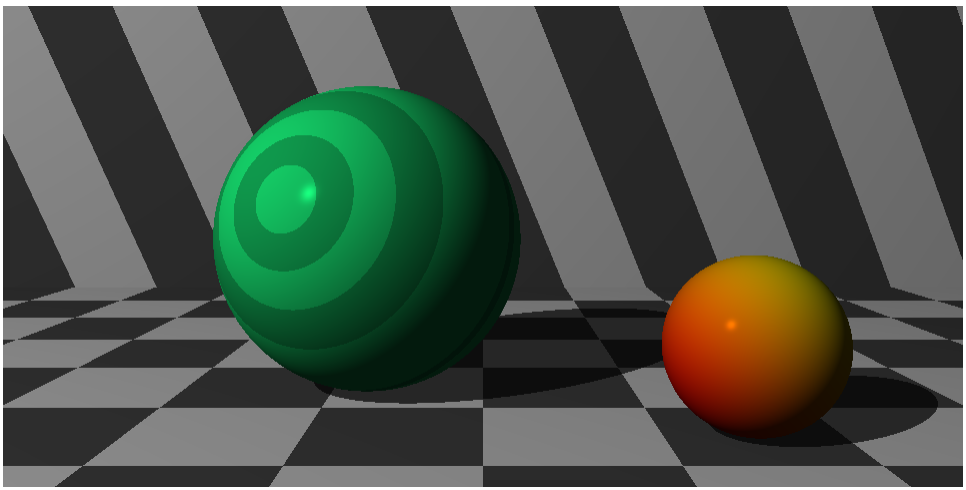
Veja o que mais você pode inventar! Quando terminar de experimentar, continue lendo. A seguir, você verá como decorar esses planos e esferas com padrões geométricos de cores, o que tornará suas cenas ainda mais interessantes.

CAPÍTULO 10

# Padrões

Seu ray tracer está realmente funcionando agora. Planos e esferas, sombreamento, sombras traçadas por raios - sim, sério. Isso é uma coisa linda.

Mas fica ainda melhor! Neste próximo capítulo, você adicionará mais colírio para os olhos lambível na forma de padrões, como este:

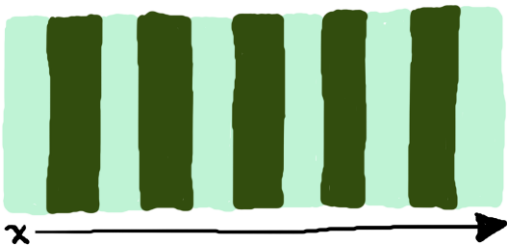


Isso! Em vez de renderizar uma forma inteira com a mesma cor chata, você vai implementar regras geométricas que definem como qualquer ponto no espaço deve ser colorido. Abordaremos quatro desses padrões: listras, gradientes, anéis e xadrez. Então, você poderá experimentar e inventar alguns por conta própria!

Aqui vamos nós.

## Fazendo um padrão listrado

UMA *padronizar* é uma função que aceita um ponto no espaço e retorna uma cor. Por exemplo, considere o seguinte padrão de faixa:




Enquanto o x coordenar mudanças, o padrão alterna entre as duas cores. As outras duas dimensões, y e z, não tem efeito sobre ele. Em outras palavras, a função se parece com isto:

$$\text{cor}(\text{ponto}, c_{\text{uma}}, c_{b}) = \begin{cases} c_{\text{uma}}, & \text{se o chão}(\text{ponto}_{x}) \bmod 2 = 0 \\ c_{b}, & \text{de outra forma} \end{cases}$$

Ou seja, se o x a coordenada está entre 0 e 1, retorna a primeira cor. Se for entre 1 e 2, retorna o segundo, e assim por diante, alternando entre os dois.

Adicione este padrão em seu programa. Para isso, você criará uma estrutura de dados que encapsula as cores utilizadas pelo padrão, bem como uma função que escolherá a cor apropriada para algum ponto.

Para começar, a maioria dos testes neste capítulo irá assumir a existência das seguintes duas constantes de cores, Preto e Branco:

**features / patterns.feature**

```
Fundo :
    Dado Preto ← cor (0, 0, 0)
        E Branco ← cor (1, 1, 1)
```

Com aqueles definidos, você pode escrever o seguinte teste apresentando uma nova função, stripe_pattern (a, b), que retorna uma instância de padrão encapsulando as duas cores uma e b.

**features / patterns.feature**

```
Cenário : Criação de um padrão de listras
    Dado padronizar ← stripe_pattern (Branco preto)
    Então pattern.a = branco
        E pattern.b = preto
```

Agora, escreva alguns testes para outra nova função, stripe_at (padrão, ponto), que deve retornar a cor apropriada para o padrão e ponto fornecidos.

**features / patterns.feature**

```
Cenário : Um padrão de listras é constante em y
    Dado padronizar ← stripe_pattern (Branco preto)
    Então stripe_at (padronizar, ponto (0, 0, 0)) = branco
        E stripe_at (padronizar, ponto (0, 1, 0)) = branco
        E stripe_at (padronizar, ponto (0, 2, 0)) = branco

Cenário : Um padrão de listras é constante em z
    Dado padronizar ← stripe_pattern (Branco preto)
    Então stripe_at (padronizar, ponto (0, 0, 0)) = branco
        E stripe_at (padronizar, ponto (0, 0, 1)) = branco
        E stripe_at (padronizar, ponto (0, 0, 2)) = branco

Cenário : Um padrão de listras alterna em x
    Dado padronizar ← stripe_pattern (Branco preto)
    Então stripe_at (padronizar, ponto (0, 0, 0)) = branco
        E stripe_at (padronizar, ponto (0,9, 0, 0)) = branco
        E stripe_at (padronizar, ponto (1, 0, 0)) = preto
        E stripe_at (padronizar, ponto (-0,1, 0, 0)) = preto
        E stripe_at (padronizar, ponto (-1, 0, 0)) = preto
        E stripe_at (padronizar, ponto (-1,1, 0, 0)) = branco
```

Faça esses dois testes passarem implementando o stripe_pattern () e stripe_at () funções. Lembrar: stripe_pattern () retorna uma nova instância da estrutura de dados e stripe_at () implementa a função que escolhe a cor em um determinado ponto. Depois que esses testes forem aprovados, continue lendo!

A próxima etapa é adicionar esse padrão de listras ao seu material. Comece escrevendo outro teste para mostrar que o iluminação() função (de O modelo de reflexão de Phong, na página 83 ) retorna a cor do padrão.

**features / materials.feature**

```
Cenário : Iluminação com um padrão aplicado
    Dado m.pattern ← stripe_pattern ( cor (1, 1, 1), cor (0, 0, 0))
        E m.ambient ← 1
        E m.diffuse ← 0
        E m.specular ← 0
        E Eyev ← vetor (0, 0, -1)
        E normalv ← vetor (0, 0, -1)
        E claro ← point_light ( ponto (0, 0, -10), cor (1, 1, 1))
    Quando c1 ← iluminação (m, luz, ponto (0,9, 0, 0), eyev, normalv, false)
        E c2 ← iluminação (m, luz, ponto (1,1, 0, 0), eyev, normalv, false)
    Então c1 = cor (1, 1, 1)
        E c2 = cor (0, 0, 0)
```

Observe que o teste usa um material apenas com iluminação ambiente. Este é um truque útil para garantir que o iluminação() função retorna uma cor facilmente previsível, uma vez que a cor não será afetada por ângulos, normais ou luzes.

Faça este teste passar modificando seu iluminação() função, adicionando algum código para obter a cor do padrão (via stripe_at ()) se o material tem um padrão definido. Em pseudocódigo, sua alteração pode ser parecida com esta:

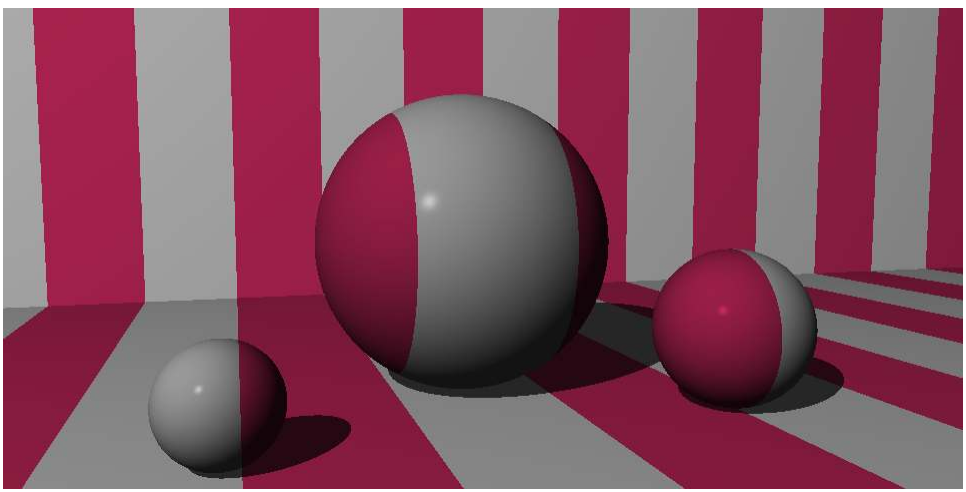
```
função iluminação (material, luz, ponto, olho, normalv, sombra)
    E se o material tem um padrão
        cor ← stripe_at (material. padrão, ponto)
    outro
        cor ← material.color
    fim se

    # então, calcule a iluminação como de costume, usando `color`
    # em vez de `material.color`

    # . .
função final
```

Com essa mudança, seu conjunto de testes deve ser aprovado novamente. Yay! Além do mais, agora você pode (meio que) renderizar uma cena contendo uma textura listrada. Experimente, mas não fique desapontado se ainda não se comportar como esperado ...

Quando você estiver pronto, continue lendo. Falaremos sobre como transformar padrões a seguir, que é como você finalmente colocará esse recurso em forma.

## Padrões de transformação

No momento, sua implementação de listras tem um pequeno problema. Se você já brincou com isso, pode ter visto: as listras estão completamente fixas, congeladas no lugar. É como se você iluminasse a cena com uma lanterna, com um filtro de faixa sobre a lâmpada. Você descobriria que cada objeto que tem um padrão de listras é coberto com listras exatamente do mesmo tamanho e orientação, independentemente de como os próprios objetos estão organizados, como nesta cena:



Porque o ponto sendo passado para o stripe_at () função está em *espaço mundial,* os padrões ignoram completamente as transformações dos objetos aos quais são aplicados.

Isso é lamentável, porque esperamos que um padrão se mova quando seu objeto se move. Se você tornar um objeto maior ou menor, o padrão nele deverá ficar maior ou menor. Girar um objeto deve girar o padrão também.

Além disso, faz sentido ser capaz de transformar os próprios padrões, independentemente do objeto. Quer suas listras mais próximas ou mais distantes? Dimensione-os. Quer mudar a forma como eles são orientados no objeto? Gire-os. O que mudar sua fase? Traduza-os para deslocá-los para um lado ou outro.

Escreva os três testes a seguir para esboçar como esse comportamento deve ser, e introduzir um novo método chamado stripe_at_object (padrão, objeto, ponto). Deveria retorna a cor para o dado *padronizar,* no dado *objeto,* no espaço do mundo dado *ponto,* e deve respeitar as transformações no padrão e no objeto ao fazer isso.

**features / patterns.feature**

```
Cenário : Listras com uma transformação de objeto
    Dado objeto ← esfera ()
        E set_transform (objeto, escala (2, 2, 2))
        E padronizar ← stripe_pattern (Branco preto)
    Quando c ← stripe_at_object (padrão, objeto, ponto (1,5, 0, 0))
    Então c = branco

Cenário : Listras com uma transformação de padrão
    Dado objeto ← esfera ()
        E padronizar ← stripe_pattern (Branco preto)
        E set_pattern_transform (padronizar, escala (2, 2, 2))
    Quando c ← stripe_at_object (padrão, objeto, ponto (1,5, 0, 0))
    Então c = branco

Cenário : Listras com um objeto e uma transformação de padrão
    Dado objeto ← esfera ()
        E set_transform (objeto, escala (2, 2, 2))
        E padronizar ← stripe_pattern (Branco preto)
        E set_pattern_transform (padronizar, tradução (0,5, 0, 0))
    Quando c ← stripe_at_object (padrão, objeto, ponto (2,5, 0, 0))
    Então c = branco
```

Faça esses testes passarem implementando o stripe_at_object () função. Deve fazer o seguinte:

1. Multiplique o ponto do espaço-mundo dado pelo inverso da matriz de transformação do objeto, para converter o ponto para o espaço-objeto.

2. Em seguida, multiplique o ponto do espaço do objeto pelo inverso da matriz de transformação do padrão para converter esse ponto para *espaço padrão.*
3. Passe o ponto resultante para o seu original stripe_at () função, e retornar o resultado.

Ficará assim em pseudocódigo:

```
função stripe_at_object (padrão, objeto, world_point)
    object_point ← inverso (object.transform) * world_point pattern_point ← inverso (pattern.transform) *
    object_point

    Retorna stripe_at (padrão, ponto_padrão)
função final
```

Quase lá! Agora faça seu programa realmente usar esta nova função, alterando seu iluminação() e sombra_hit () funciona da seguinte forma:

1. Adicionar objeto como mais um parâmetro para o seu iluminação() função. Os testes e pseudocódigo neste livro assume que a nova assinatura de função é iluminação (material, objeto, luz, ponto, olho, normalv, sombra). stripe_at_object
2. Modifique a implementação do iluminação() função para que ele chame () ao invés de stripe_at ().
3. Modificar sombra_hit () de modo que passe o acerto objeto propriedade para iluminação().
4. Corrija o seu iluminação() testa para que eles criem um objeto (uma esfera está bem - é apenas um espaço reservado para esses testes, de qualquer maneira) e passá-lo para iluminação().

Todos os seus testes devem estar passando agora. Comemore dando mais uma chance ao padrão de listras! Veja o que acontece se você girar as listras, dimensioná-las ou transformar o objeto ao qual estão anexadas.

Quando estiver pronto, vamos falar sobre como generalizar tudo isso, em preparação para adicionar mais padrões.

## Padrões de generalização

A ideia agora é modificar o seu código para que um material possa ser atribuído *nenhum* padrão, não apenas listras. O processo para fazer isso será muito parecido com a refatoração que você fez em Formas de refatoração, na página 117 , quando você estava se preparando para apoiar aviões como primitivos. Especificamente, você resolverá isso em cinco etapas:

1. Identifique as peças que cada padrão terá em comum.
2. Implemente um padrão abstrato que encapsule essas partes comuns e delegue a padrões concretos para seus bits específicos.
3. Modifique o padrão de listras para estender este padrão abstrato.
4. Modifique sua implementação de material para depender do padrão abstrato.

5. Faça com que todos os testes existentes sejam aprovados.

Então, os bits comuns. A boa notícia é que todos os padrões serão essencialmente iguais, diferenciados apenas pela função que converte os pontos em cores. Além dessa função, todo padrão terá uma matriz de transformação e todo padrão precisará usá-la para ajudar a transformar um determinado ponto do espaço do mundo para o espaço do padrão antes de produzir uma cor.

Tal como acontece com a refatoração de formas, o caminho a seguir aqui vai depender muito de como você arquitetou seu programa até agora. Uma maneira é seguir uma estratégia semelhante à proposta para as formas, onde a abstração de base realiza a funcionalidade comum e delega a funcionalidade específica às implementações concretas.

Se você seguir esse caminho, use os seguintes testes como diretrizes para escrever o seu próprio. Esses testes assumem que a função abstrata (aquela que transforma o ponto e delega à função concreta) é chamada pattern_at_shape (padrão, forma, ponto). A função concreta (a ser implementada por cada padrão) está aqui

simplesmente chamado pattern_at (padrão, ponto).

Os testes também presumem que existe uma função chamada Padrão de teste(), que é semelhante ao test_shape () função de Formas de refatoração, na página 117 . Seu trabalho será ajudá-lo a testar os comportamentos da superclasse de padrão abstrato, retornando uma implementação especial usada apenas para os testes.

Primeiro, mostre que esse padrão de teste tem uma matriz de transformação e que a transformação é (por padrão) a matriz de identidade.

**features / patterns.feature**

```
Cenário : A transformação do padrão padrão
    Dado padronizar ← Padrão de teste ()
    Então pattern.transform = identity_matrix
```

A seguir, mostre que a transformação do padrão pode ser atribuída.

**features / patterns.feature**

```
Cenário : Atribuindo uma transformação
    Dado padronizar ← Padrão de teste ()
    Quando set_pattern_transform (padronizar, tradução (1, 2, 3))
    Então pattern.transform = tradução (1, 2, 3)
```

Em seguida, teste o pattern_at_shape () função para ver se ela transforma corretamente os pontos antes de chamar a função concreta. Os testes a seguir substituem os que você escreveu anteriormente neste capítulo, testando as transformações do padrão de distribuição.

**features / patterns.feature**

```
Cenário : Um padrão com uma transformação de objeto
    Dado forma ← esfera ()
```

```
        E set_transform (forma, escala (2, 2, 2))
        E padronizar ← Padrão de teste ()
    Quando c ← pattern_at_shape (padrão, forma, ponto (2, 3, 4))
    Então c = cor (1, 1,5, 2)

Cenário : Um padrão com uma transformação de padrão
    Dado forma ← esfera ()
        E padronizar ← Padrão de teste ()
        E set_pattern_transform (padronizar, escala (2, 2, 2))
    Quando c ← pattern_at_shape (padrão, forma, ponto (2, 3, 4))
    Então c = cor (1, 1,5, 2)

Cenário : Um padrão com um objeto e uma transformação de padrão
    Dado forma ← esfera ()
        E set_transform (forma, escala (2, 2, 2))
        E padronizar ← Padrão de teste ()
        E set_pattern_transform (padronizar, tradução (0,5, 1, 1,5))
    Quando c ← pattern_at_shape (padrão, forma, ponto (2,5, 3, 3,5))
    Então c = cor (0,75, 0,5, 0,25)
```

Esses testes assumem que a função concreta do padrão de teste é definida assim:

$$\text{padrão em}_(\text{ padrão, ponto}) = \text{cor}(\text{ ponto}_{x}, \text{ponto}_{y}, \text{ponto}_{z})$$

Em outras palavras, ele pega o ponto dado e retorna uma nova cor onde os componentes vermelho / verde / azul da cor são definidos para o ponto x / y / z componentes. Você pode então usar o *cor* para ver que o *ponto* foi transformado!

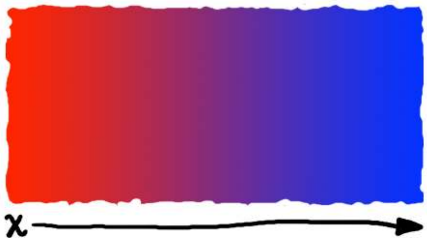
Uma vez que eles estão passando, atualize seu stripe_pattern () implementação de modo que herde desse padrão abstrato. Você pode remover o código que transforma os pontos, já que isso agora é cuidado pelo abstrato pattern_at_shape ()
função.

Por último, atualize sua estrutura de dados de material, de modo que faça referência ao padrão abstrato em vez do padrão de distribuição, e faça iluminação() Ligar para pattern_at_shape ()
função.

Uau! Organize as coisas certificando-se de que todos os seus testes sejam aprovados e siga em frente. Com esse padrão abstrato como base, você está pronto para começar a implementar mais padrões.

## Fazendo um padrão de gradiente

UMA *gradiente* padrão é como listras, mas em vez de etapas discretas de uma cor para a próxima, a função retorna uma mistura das duas cores, interpolando linearmente de uma para a outra como o x coordenar mudanças. Se a primeira cor for vermelha e a segunda for azul, o gradiente resultante terá a seguinte aparência:




Adicione o seguinte teste para mostrar como um padrão de gradiente linear básico deve funcionar.

**features / patterns.feature**

```
Cenário : Um gradiente interpola linearmente entre as cores
    Dado padronizar ← gradiente_padrão (Branco preto)
    Então pattern_at (padronizar, ponto (0, 0, 0)) = branco
        E pattern_at (padronizar, ponto (0,25, 0, 0)) = cor (0,75, 0,75, 0,75)
        E pattern_at (padronizar, ponto (0,5, 0, 0)) = cor (0,5, 0,5, 0,5)
        E pattern_at (padronizar, ponto (0,75, 0, 0)) = cor (0,25, 0,25, 0,25)
```

Para fazer isso passar, seu gradiente_pattern () implementação deve usar um *função de mistura.* Esta é uma função que pega dois valores e interpola os valores entre eles. Uma interpolação linear básica se parece com isto:

$$\text{cor}(p, c_{\text{uma}}, c_b) = c_a + (c_b - c_a) * (p_x - \text{chão}(p_x))$$

Isso tira a distância entre as duas cores, multiplica-a pela parte fracionária do x coordene e adiciona o produto à primeira cor. O resultado é uma transição suave e linear da primeira cor para a segunda.

Em pseudocódigo, a função de cor do seu gradiente deve ser semelhante a esta:

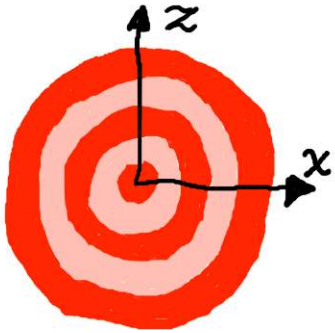
```
função pattern_at (gradiente, ponto)
    distância ← gradiente.b - gradiente.a fração ← point.x - piso (ponto.x)


    Retorna gradiente.a + distância * fração
fim
```

Depois que o teste for aprovado, vamos dar uma olhada nos padrões dos anéis.

## Fazendo um padrão de anel

UMA *padrão de anel* depende de duas dimensões, x e z, para decidir qual cor retornar. Funciona de forma semelhante às listras, mas em vez de testar a distância do ponto apenas x, ele testa a distância do ponto em ambos x e z, o que resulta neste padrão de círculos concêntricos, conforme mostrado no figura na página 136 .

Escreva o seguinte teste para isso. Você está verificando se esses anéis se estendem em ambos x e z.




**features / patterns.feature**

```
Cenário : Um anel deve se estender em x e z
    Dado padronizar ← ring_pattern (Branco preto)
    Então pattern_at (padronizar, ponto (0, 0, 0)) = branco
        E pattern_at (padronizar, ponto (1, 0, 0)) = preto
        E pattern_at (padronizar, ponto (0, 0, 1)) = preto
        # 0.708 = apenas um pouco mais do que √2/2
        E pattern_at (padronizar, ponto (0,708, 0, 0,708)) = preto
```

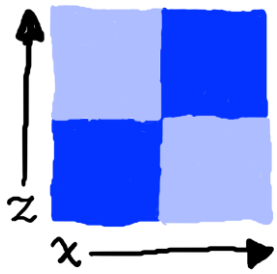
Para fazer essa passagem, você implementará a função para um padrão de anel, algo assim:

$$\text{cor}(p, c_{\text{uma}}, c_{\text{b}}) = \begin{cases} c_{\text{uma}}, & \text{se o chão}\left(\sqrt{p_x^2 + p_z^2}\right) \bmod 2 = 0 \\ c_{\text{b}}, & \text{de outra forma} \end{cases}$$

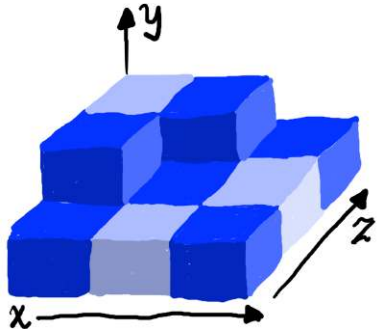
Faça esse teste passar e, em seguida, veremos mais um padrão: verificadores.

## Fazendo um padrão Checker 3D

Um bidimensional *verificador* O padrão é um padrão de repetição de quadrados, onde dois quadrados da mesma cor nunca são adjacentes, como este:



O que é legal é que essa ideia pode se estender a três dimensões também, como esta:




Você obtém um padrão de alternância *cubos,* onde dois cubos da mesma cor nunca são adjacentes. Este padrão verificador tridimensional é o que você implementará aqui.

Vá em frente e escreva os seguintes testes para esses verificadores 3D, mostrando que o padrão de fato se repete em todas as três dimensões.

**features / patterns.feature**

```
Cenário : As damas devem repetir em x
    Dado padronizar ← checkers_pattern (Branco preto)
    Então pattern_at (padronizar, ponto (0, 0, 0)) = branco
        E pattern_at (padronizar, ponto (0,99, 0, 0)) = branco
        E pattern_at (padronizar, ponto (1,01, 0, 0)) = preto

Cenário : As damas devem repetir em y
    Dado padronizar ← checkers_pattern (Branco preto)
    Então pattern_at (padronizar, ponto (0, 0, 0)) = branco
        E pattern_at (padronizar, ponto (0, 0,99, 0)) = branco
        E pattern_at (padronizar, ponto (0, 1,01, 0)) = preto

Cenário : As damas devem repetir em z
    Dado padronizar ← checkers_pattern (Branco preto)
    Então pattern_at (padronizar, ponto (0, 0, 0)) = branco
        E pattern_at (padronizar, ponto (0, 0, 0,99)) = branco
        E pattern_at (padronizar, ponto (0, 0, 1,01)) = preto
```

A função desse padrão é muito parecida com a das listras, mas em vez de depender de uma única dimensão, ela depende da soma de todas as três dimensões,
x, y, e z, como isso. (Observe o

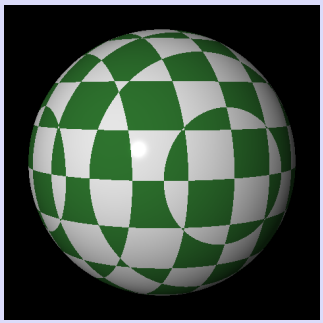
$$\text{cor}(p, c_{uma}, c_b) = \begin{cases} \text{\{ no } \lfloor x \rfloor \text{ é o mesmo que andar (x).)} \\ c_{uma}, \text{ E se } (\lfloor p_x \rfloor + \lfloor p_y \rfloor + \lfloor p_z \rfloor) \bmod 2 = 0 \\ c_b, \text{ de outra forma} \end{cases}$$

Uma vez que todos os seus testes estão passando, continue lendo. Vamos encerrar este capítulo com algumas idéias para você experimentar.

**Joe pergunta:**

**Por que minha esfera quadriculada parece estranha?**

Se você tentar aplicar este padrão de verificação a uma esfera, obterá algo assim:



Se, em vez disso, você esperava que o padrão cobrisse a superfície da esfera em um padrão de grade regular, isso pode tê-lo deixado coçando a cabeça e se perguntando o que fez de errado. Bem, boas notícias! Você não fez nada errado. O padrão está funcionando perfeitamente certo.

Porque os padrões convertem *pontos no espaço* às cores, é como se você estivesse esculpindo aquela esfera em um bloco com padrão xadrez, em vez de pintar perfeitamente o padrão na superfície da esfera.

Para aplicar uma textura bidimensional (como damas) à superfície de um objeto, você precisa implementar algo chamado *UVmapping,* que converte um ponto tridimensional de intersecção ( x, y, z) em uma coordenada de superfície bidimensional ( u, v). Você então mapearia essa coordenada de superfície para uma cor. É divertido de fazer, mas infelizmente está além do escopo deste livro. Recursos de estilo tutorial são difíceis de encontrar, mas com um pouco de leitura nas entrelinhas e alguma experimentação, a busca por tópicos como “mapeamento de textura esférica” pode render frutos.

## Juntar as peças

OK! Você tem implementações de trabalho de quatro padrões diferentes: listras, gradientes, anéis e xadrez. Sua primeira ordem do dia, então, deve ser levar todos eles para dar uma volta! Experimente cada um deles em planos e esferas, dimensione-os, gire-os, experimente cores diferentes e sinta como esses padrões se comportam na prática.

Quando sentir que os está dominando, tente alguns experimentos mais profundos. Aqui estão algumas idéias para você começar:

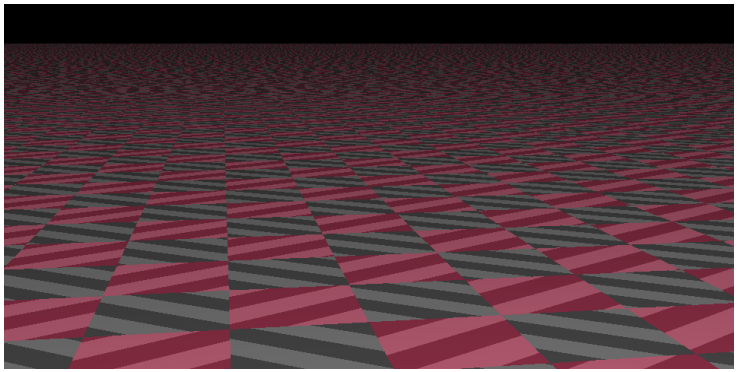
*Padrão de gradiente radial*

Considere o seu padrão de anel, que cria um padrão radial de círculos concêntricos. Então, considere seu padrão de gradiente, que interpola entre

duas cores. Como você combinaria esses dois conceitos para criar um novo padrão de “gradiente radial” que interpola entre duas cores, radialmente?

### *Padrões aninhados*

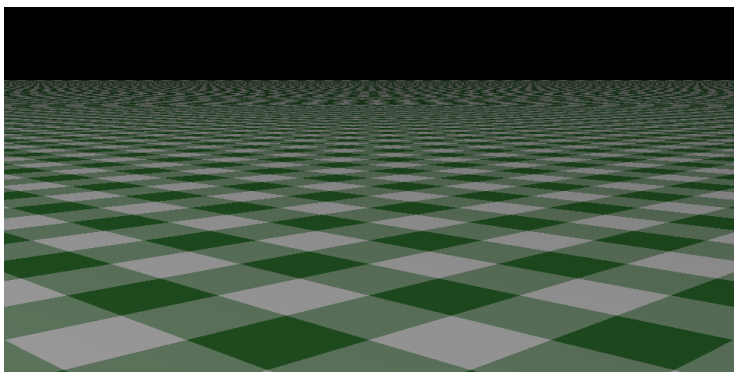
Em vez de especificar um par de *cores* ao instanciar um padrão, e se você especificasse *outros padrões?* Portanto, em vez de um padrão xadrez em preto e branco, faça um em que os xadrez contenham padrões alternados de listras em orientações diferentes, como este:



Uma maneira de fazer isso funcionar é adicionar um novo padrão, chamado solid_pattern (color), que retorna a mesma cor para todos os pontos. Os padrões podem então ser aninhados, com o padrão mais interno sempre sendo um desses padrões de cores sólidas.

### *Padrões combinados*

Isso adiciona um novo padrão, blended_pattern (a, b), Onde uma e b estamos *outros padrões,* ao invés de *cores.* O padrão de mesclagem avaliará então ambos os padrões em cada ponto e mesclará as cores resultantes. (Misturar uma cor pode ser tão simples quanto fazer a média delas, se você quiser, mas você pode ser criativo com essa fórmula também!) Por exemplo, aqui está uma mistura de dois padrões de listras verdes / brancas, cruzando-se em ângulos de noventa graus para um outro:

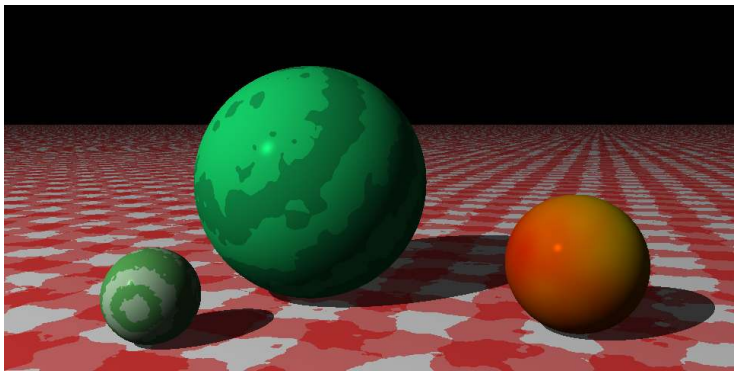


### *Padrões perturbados*

Este é outro divertido! É uma maneira de adicionar texturas de aparência orgânica às suas cenas. A forma como funciona é que você usa uma função de ruído 3D para "jitter" o ponto antes que o padrão o consuma. Procure uma implementação de ruído Perlin ou ruído Simplex. Em seguida, crie um novo padrão, chamado perturbar (padrão),
que usa esse ruído para jitter cada ponto antes de delegá-lo ao padrão fornecido.

"Jittering" um ponto significa movê-lo por uma pequena quantidade. Com a maioria das implementações de ruído Perlin, por exemplo, você pode solicitar uma faixa de valores para um determinado ponto tridimensional. Você escala esses números por alguma fração (talvez 20% ou menos) e, em seguida, pega o primeiro desses valores e adiciona-o ao x coordenada. Em seguida, adicione o segundo valor ao y coordenada, e o terceiro valor para o z coordenada. Finalmente, você trata isso como um novo ponto e o passa para o subpadrão.

O resultado é que cada padrão parece *perturbado,* como se alguém tivesse enfiado o dedo na tinta úmida e girado. Aqui está um exemplo de versões perturbadas de cada um dos padrões deste capítulo:



Existem implementações de ruído Perlin para muitas linguagens de programação. Por exemplo, você pode revisar a implementação de referência original de Ken Perlin, escrita em Java.[1]

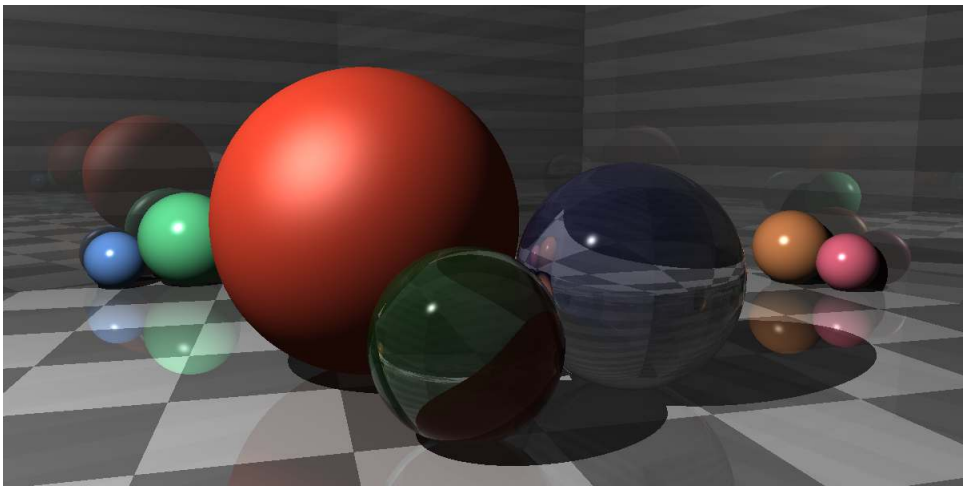
O céu é o limite! Brinque com os padrões e veja o que você pode inventar. Quando você estiver pronto para seguir em frente, no próximo capítulo iremos elevar o realismo de suas cenas, com reflexão e refração.

1 https://mrl.nyu.edu/~perlin/noise/

CAPÍTULO 11

# Reflexão e Refração

Tudo bem. Segure a alça de assistência ao passageiro, porque você está prestes a adicionar um pouco de refinamento material ao seu ray tracer. Confira esta prévia:



Isso mesmo, meninos e meninas. Você vai fazer objetos *reflexivo* e *transparente*. Espelhos e mármores de vidro serão sua ostra.

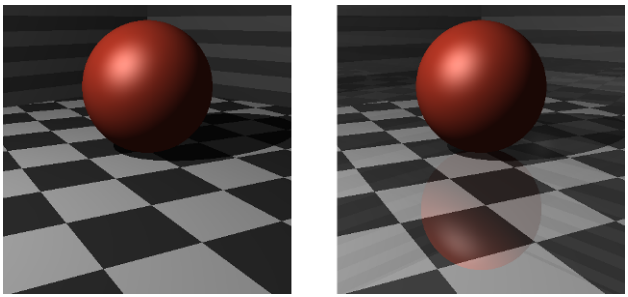
Ambos funcionam por meios semelhantes: gerando um raio adicional no ponto de intersecção e seguindo-o recursivamente para determinar a cor naquele ponto. Você os enfrentará um de cada vez: primeiro reflexo e, em seguida, transparência e refração.

Você está pronto? Aqui vai!

# Reflexão

Olhe a sua volta. Provavelmente, você encontrará algo nas proximidades que seja reflexivo em um grau ou outro. Talvez seja a tela do seu telefone, ou uma mesa polida, ou uma janela, ou um par de óculos de sol. Seja o que for, essa reflexão fornece todos os tipos de pistas sobre o que esperar dessa superfície e ajuda a convencer seu cérebro de que o que você está vendo é *real.*

Isso também funciona em cenas renderizadas. Adicionar apenas um pouco de reflexão sutil pode fazer sua cena florescer com fotorrealismo. Considere as duas imagens a seguir:



Ambos retratam a mesma cena, do mesmo ângulo e com a mesma iluminação, mas o piso à direita é apenas ligeiramente reflexivo, fazendo com que pareça mais brilhante do que o outro.

Você adicionará esse recurso ao seu ray tracer com sete testes:

1. Adicione um reflexivo atribuir à sua estrutura de dados materiais.
2. Atualizar prepare_computations () para calcular o vetor de reflexão do raio, reflectv.
3. Manuseie o caso em que o raio atinge uma superfície não refletiva.
4. Manuseie o caso em que o raio atinge uma superfície reflexiva.
5. Certifique-se sombra_hit () chama a função para calcular reflexões.
6. Certifique-se de que seu ray tracer pode evitar recursão infinita, como quando um raio salta entre dois espelhos paralelos.
7. Mostre que seu código pode definir um limite para a profundidade que a recursão pode atingir.

Observe que a partir de agora, os capítulos serão um pouco mais simplificados. Até este ponto, você viu os testes introduzidos com um pouco de alarde e discussão. Mas agora as rodinhas estão saindo. Você sabe o que fazer agora. Você verá os testes, obterá algumas explicações e (quando necessário) examinará os algoritmos e talvez alguns pseudocódigos. Você conseguiu!

Aqui vamos nós, um teste de cada vez.

### Teste # 1: Adicione o Atributo de Material Refletivo

*Mostre que sua estrutura de material contém um novo atributo, chamado* **reflexivo.**

Quando reflexivo é 0, a superfície é completamente não reflexiva, enquanto configurá-la para 1 produz um espelho perfeito. Os números intermediários produzem reflexos parciais.

```
features / materials.feature
Cenário : Refletividade para o material padrão
    Dado m ← material ()
    Então m.refletivo = 0,0
```

Certifique-se de que o novo atributo seja um valor de ponto flutuante, para que você possa implementar a reflexão parcial.

## Teste # 2: Calcule o vetor refletv

*Mostre que o* **prepare_computations ()** *função pré-computa o* **refletv** *vetor.*

Crie um plano e posicione um raio acima dele, inclinado para baixo em um ângulo de 45 °. Posicione a interseção no avião e tenha prepare_computations () calcule o vetor de reflexão.

```
        features / intersections.feature
Linha 1 Cenário : Pré-computando o vetor de reflexão
      2     Dado forma ← plano ()
      3         E r ← raio ( ponto (0, 1, -1), vetor (0, - √ 2/2, √ 2/2))
      4         E eu ← interseção ( √ 2, forma)
      5     Quando compor ← prepare_computations (i, r)
      6     Então comps.reflectv = vetor (0, √ 2/2, √ 2/2)
```

A linha 3 cria e orienta o raio, e a linha 4 coloca o acerto √ 2 unidades de distância, cortesia do teorema de Pitágoras. Por último, a linha 6 afirma que o vetor de reflexão salta para cima do plano em outro ângulo de 45 °.

Calcular refletv no prepare_computations () refletindo o vetor de direção do raio em torno do vetor normal do objeto, como este:

```
# depois de negar o normal, se necessário
comps.reflectv ← refletir (ray.direction, comps.normalv)
```

É como você fez em seu iluminação() função, em O modelo de reflexão de Phong, na página 83 , quando você calculou o vetor de reflexão da luz. Aqui, porém, você está refletindo o *raio,* e não o *claro.*

### Teste # 3: Golpeie uma Superfície Não Refletiva

*Mostre que quando um raio atinge uma superfície não refletiva, o* **cor_refletida ()** *função retorna a cor preta.*

Você está chegando ao cerne do algoritmo de reflexão em si, agora. Este teste apresenta uma nova função, cor_refletida (mundo, composições), que será o núcleo de como o seu ray tracer calcula as reflexões.

Coloque um raio dentro da origem do mundo padrão, dentro de ambas as esferas do mundo. Rebata o raio para fora da esfera mais interna. Definindo a esfera
ambiente propriedade para 1, você pode garantir que qualquer reflexo terá algo para refletir, mas porque a esfera mais interna não é reflexiva, cor_refletida ()
deve simplesmente retornar preto.

**features / world.feature**

```
Cenário : A cor refletida para um material não refletivo
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, 0), vetor (0, 0, 1))
        E forma ← o segundo objeto em w
        E shape.material.ambient ← 1
        E eu ← interseção (1, forma)
    Quando compor ← prepare_computations (i, r)
        E cor ← cor_refletida (w, comps)
    Então color = cor (0, 0, 0)
```

Para este teste, faça o seu cor_refletida () função retorna preto quando o material
reflexivo atributo é 0. O próximo teste irá aprofundar essa função um pouco mais.

### Teste nº 4: Atingir uma superfície reflexiva

*Mostra isso* **cor_refletida ()** *retorna a cor por meio de reflexão quando a superfície atingida é reflexiva.*

Adicione um plano reflexivo à cena padrão, logo abaixo das esferas, e oriente um raio de forma que atinja o plano, reflita para cima e atinja a esfera mais externa.

**features / world.feature**

```
Linha 1  Cenário : A cor refletida para um material reflexivo
-            Dado C ← default_world ()
-                E forma ← plano () com:
-                    | material.refletivo | 0,5 | transformar                                    |
5                                                    | tradução (0, -1, 0) |
-                E forma é adicionada a w
-                E r ← raio ( ponto (0, 0, -3), vetor (0, - √ 2/2, √ 2/2))
-                E eu ← interseção ( √ 2, forma)
-            Quando compor ← prepare_computations (i, r)
10               E cor ← cor_refletida (w, comps)
-            Então color = cor (0,19032, 0,2379, 0,14274)
```

As linhas 3-5 configuram o plano (semi) reflexivo e posicionam-no em y = -1. Depois de preparar o golpe, a cor refletida será uma versão mais escura do tom de verde da esfera, porque o plano refletirá apenas metade da luz da esfera.

Implemento cor_refletida () criando um novo raio, originado na localização do acerto e apontando na direção de reflectv. Encontre a cor do novo raio via color_at (),
e, em seguida, multiplique o resultado pelo reflexivo valor. Se reflexivo é definido como algo entre 0 e 1, isso dará a você uma reflexão parcial.

Em pseudocódigo, funciona assim:

```
função cor_refletida (mundo, comps)
    E se comps.object.material.reflective = 0
        Retorna cor (0, 0, 0)
    fim se

    reflect_ray ← raio (comps.over_point, comps.reflectv) color ← color_at (mundo, reflect_ray)


    Retorna color * comps.object.material.reflective
função final
```

Gerar esses raios secundários é como os traçadores de raios podem produzir reflexos realistas. Apenas certifique-se de usar o comps.over_point atributo (e não
comps.point) ao construir o novo raio. Caso contrário, erros de arredondamento de ponto flutuante farão com que alguns raios se originem logo abaixo da superfície, fazendo com que eles cruzem a mesma superfície da qual deveriam estar refletindo.

## Teste # 5: atualize a função shadow_hit

*Mostra isso* **sombra_hit ()** *incorpora a cor refletida na cor final.*

Recicle o teste anterior, mas desta vez ligue sombra_hit () em vez de ligar cor_refletida () diretamente. A cor resultante deve combinar o branco do plano com o verde refletido da esfera.

```
features / world.feature
Cenário : sombra_hit () com um material reflexivo
    Dado C ← default_world ()
        E forma ← plano () com:
            | material.refletivo | 0,5 | transformar                                                    |
                                                    | tradução (0, -1, 0) |
        E forma é adicionada a w
        E r ← raio ( ponto (0, 0, -3), vetor (0, - √ 2/2, √ 2/2))
        E eu ← interseção ( √ 2, forma)
    Quando compor ← prepare_computations (i, r)
        E cor ← sombra_hit (w, comps)
    Então color = cor (0,87677, 0,92436, 0,82918)
```

Implemente isso fazendo o sombra_hit () chamada de função cor_refletida (), e adicionando a cor, ele retorna à cor da superfície. Em pseudocódigo:

```
função sombra_hit (mundo, comps)
    sombreado ← is_shadowed (mundo, comps.over_point)
```

```
    superfície ← iluminação (comps.object.material,
                                comps.object,
                                world.light,
                                comps.over_point, comps.eyev, comps.normalv,
                                sombreado)

    refletido ← cor_refletida (mundo, comps)

    Retorna superfície + refletida
função final
```

Ao adicionar a cor refletida à cor da superfície, os dois se misturam e produzem um reflexo crível. No entanto, há uma pegadinha escondida aqui. O
sombra_hit () função agora chama cor_refletida (), que chama color_at (), que chama
sombra_hit () … Esse é um loop recursivo, com potencial para causar alguns problemas. Vamos tratar disso a seguir.

## Teste # 6: Evite a recursão infinita

*Mostre que seu código lida com segurança com recursão infinita causada por dois objetos que refletem mutuamente os raios entre eles.*

Crie dois espelhos paralelos posicionando um plano acima do outro e tornando-os reflexivos. Oriente um raio de modo que atinja um plano e salte para o outro. O que vai acontecer?

```
features / world.feature
Cenário : color_at () com superfícies mutuamente reflexivas
    Dado C ← mundo ()
        E w.light ← point_light ( ponto (0, 0, 0), cor (1, 1, 1))
        E mais baixo ← plano () com:
            | material.refletivo | 1                                    |
            | transformar                  | tradução (0, -1, 0) |
        E inferior é adicionado a w
        E superior ← plano () com:
            | material.refletivo | 1 transformar                        |
                                           | tradução (0, 1, 0) |
        E superior é adicionado a w
        E r ← raio ( ponto (0, 0, 0), vetor (0, 1, 0))
    Então color_at (w, r) deve terminar com sucesso
```

Seu ray tracer provavelmente não lidará bem com isso. Por causa desse loop recursivo que você fez para o teste anterior, seus reflexos irão saltar para frente e para trás entre esses dois espelhos, até que sua pilha exploda.

A recursão infinita são os poços.

**Joe pergunta:**

## Como posso testar “deve terminar com sucesso”?

O teste “deve terminar com sucesso” pode ser complicado. Em vez de tentar determinar se o seu programa realmente será encerrado (porque boa sorte com isso uma), pode ser melhor verificar o oposto. Procure o que acontece quando o programa *não*
terminar. Muito provavelmente, sob recursão infinita, seu programa acabará ficando sem memória. Seu ambiente gera uma exceção quando isso acontece? Teste isso, se puder. Ou, se isso não for uma opção, você pode, em vez disso, afirmar que a função termina em algum período de tempo finito.

uma. Veja a entrada da Wikipedia sobre o problema da parada: en.wikipedia.org/wiki/Halting_problem

Ainda assim, os testes devem passar. Uma maneira de fazer isso é limitar o quão profundamente a recursão pode ir. Afinal, se um raio só puder saltar quatro ou cinco vezes, é improvável que sua pilha de chamadas explodir. Você pode implementar essa restrição declarando algum limite e, em seguida, exigindo o cor_refletida ()

função para retornar imediatamente se a recursão for mais profunda do que isso.

Por enquanto, permita que este teste falhe. O próximo teste indicará a direção certa e ajudará a fazer com que os dois passem.

### Teste # 7: Limite de recursão

*Mostra isso* **cor_refletida ()** *retorna sem efeito quando invocado no limite de seu limite recursivo.*

Duplique o cenário em Teste # 5: atualize a função shadow_hit, na página 145 . A diferença, porém, é que aqui você invocará cor_refletida (mundo, composições, restante) com um novo parâmetro adicional - remanescente - que informa à função quantas chamadas recursivas a mais ela pode fazer.

**features / world.feature**

```
Linha 1  Cenário : A cor refletida na profundidade recursiva máxima
-          Dado C ← default_world ()
-              E forma ← plano () com:
-                  | material.refletivo | 0,5 | transformar                                    |
5                                                  | tradução (0, -1, 0) |
-              E forma é adicionada a w
-              E r ← raio ( ponto (0, 0, -3), vetor (0, - √ 2/2, √ 2/2))
-              E eu ← interseção ( √ 2, forma)
-          Quando compor ← prepare_computations (i, r)
10             E cor ← cor_refletida (w, comps, 0)
-          Então color = cor (0, 0, 0)
```

A linha 10 define o remanescente parâmetro para 0, informando à função que não é permitido fazer mais chamadas recursivas. Em vez disso, deve retornar preto.

Faça isso adicionando outra condição ao topo de seu cor_refletida () função. Deve retornar preto se remanescente é menor que 1.

Para tornar isso útil, no entanto, você precisa passar esse número de volta e entre color_at (), shadow_hit (), e cor_refletida (). Execute o seguinte reestruturação:

1. Adicione um terceiro parâmetro para color_at (mundo, raio, restante).
2. Adicione um terceiro parâmetro para sombra_hit (mundo, hit, restante).
3. Faça color_at () passe o remanescente valor para sombra_hit ().
4. Faça com que quando cor_refletida () chamadas color_at (), diminui o remanescente valor antes de passá-lo.

Em outras palavras, use algo assim:

```
função color_at (mundo, raio, restante)
    # . .
    cor ← sombra_hit (mundo, comps, restante)
    # . .
função final

função sombra_hit (mundo, comps, restante)
    # . .
    refletido ← cor_refletida (mundo, comps, restante)
    # . .
função final

função cor_refletida (mundo, comps, restante)
    E se restantes <= 0
        Retorna cor (0, 0, 0)
    fim se

    # . .
    cor ← color_at (mundo, reflect_ray, restante - 1)
    # . .
função final
```

Desta forma, seu código mantém o controle de quão profundo a recursão pode ir e evita coisas desagradáveis quando as coisas ficam um pouco exageradas.

Certifique-se de alterar todas as chamadas existentes de color_at () e sombra_hit () para passar na profundidade recursiva máxima através do novo remanescente parâmetro. Se sua linguagem de programação oferece suporte a valores de parâmetro padrão, este é um ótimo lugar para isso. Contexto remanescente o valor padrão de 4 ou 5 é empiricamente muito seguro. Números maiores irão desacelerar seu renderizador em cenas com muitos objetos reflexivos.

Depois de fazer isso, você deve descobrir que todos os seus testes anteriores estão passando novamente, incluindo Teste # 6: Evite a recursão infinita, na página 146 .

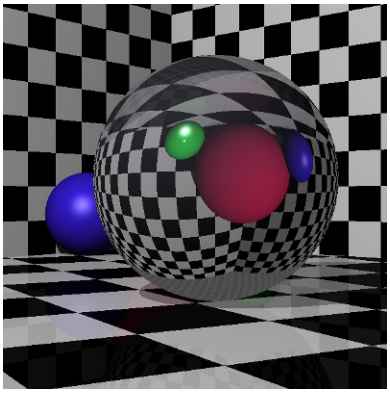
Uau!

Reserve um momento e celebre com uma cena simples. Preencha-o com esferas e torne algumas delas reflexivas. Veja como a cor de uma superfície afeta o reflexo. Algumas cores funcionam melhor do que outras? O que acontece quando você varia os parâmetros ambientais, difusos e especulares em uma superfície reflexiva?

Quando você tiver feito isso para sua satisfação, continue a ler. É hora de falar sobre transparência e refração.

## Transparência e Refração

*Refração* descreve como a luz se curva quando passa de um meio transparente para outro. Com isso adicionado ao seu ray tracer, você será capaz de renderizar vidro, água e outros materiais transparentes bastante convincentes. A figura a seguir é um exemplo, mostrando como uma esfera de vidro distorce a imagem da cena por trás dela.



A refração é governada por uma propriedade chamada de *índice de refração (* ou *índice de refração).* É um número que determina o grau em que a luz se curvará ao entrar ou sair do material, em comparação com outros materiais. Quanto maior o número, mais fortemente a luz se curvará ao encontrar esse material.

Você pode encontrar várias listas online de materiais e seus respectivos índices de refração.[1] Para poupar a pesquisa, aqui estão alguns dos materiais mais comuns e seus índices de refração:

1. Aqui está uma dessas listas: hyperphysics.phy-astr.gsu.edu/hbase/Tables/indrf.html

- Vácuo: 1
- Aéreo: 1.00029
- Água: 1.333
- Vidro: 1,52
- Diamante: 2.417

Mais uma vez, a chave para fazer este trabalho em seu ray tracer é gerar um raio secundário toda vez que seu raio encontrar um material transparente, assim como você fez para reflexão. A diferença aqui é apenas a matemática que determina para qual direção o novo raio deve ir.

Você implementará isso em oito testes:

1. Adicionar transparência e índice de refração ao material como novos atributos.
2. Para uma determinada interseção, encontre o índice de refração do material pelo qual o raio está passando *a partir de,* e o índice de refração do material que o raio está passando *para.* Normalmente nos referimos a eles como $n_1$ e $n_2$
3. Adicionar um novo atributo em prepare_computations (), chamado under_point, que determinas onde o raio refratado se originará. Manuseie a refração
4. quando a superfície for opaca.
5. Lidar com a refração quando a profundidade recursiva máxima for atingida. Lidar com a refração sob *reflexão*
6. *interna total. (* Mais sobre isso em breve!) Lidar com a refração no caso geral, quando a superfície é
7. transparente. Combine as cores refletidas e refratadas com a cor do material para encontrar a cor final da
8. superfície.

Você conseguiu!

## Teste # 1: Adicione os Atributos de Material para transparência e índice de refração

*Mostre que sua estrutura material contém dois novos atributos, chamados* **transparência** *e* **índice de refração. transparência** *o padrão é* **0,** *e* **índice de refração** *o padrão é* **1**

```
features / materials.feature
Cenário : Transparência e índice de refração para o material padrão
    Dado m ← material ()
    Então m.transparency = 0.0
        E m.refractive_index = 1.0
```

Inadimplente transparência para 0 torna todas as superfícies opacas por padrão, e usando 1 como o padrão para índice de refração torna todos os objetos vazios, conchas preenchidas com vácuo.

Com a adição desses dois atributos, você também pode implementar uma função auxiliar, glass_sphere (), que cria uma esfera com uma textura vítrea. Adicione o seguinte teste para certificar-se de que funciona conforme o esperado.

**features / spheres.feature**

```
Cenário : Um ajudante para produzir uma esfera com um material vítreo
    Dado s ← glass_sphere ()
    Então s.transform = identity_matrix
        E s.material.transparency = 1.0
        E s.material.refractive_index = 1,5
```

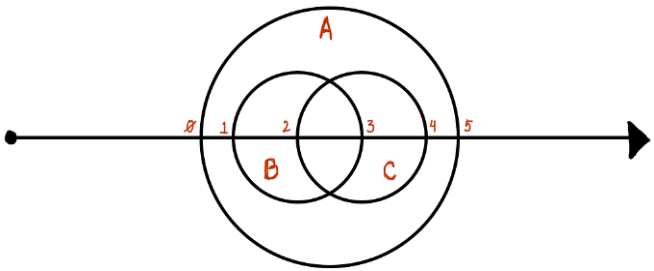
Isso será útil para testes posteriores, incluindo o próximo.

## Teste # 2: Determinando $n_1$ e $n_2$

*Mostra isso* **prepare_computations ()** *determina $n_1$ e $n_2$ corretamente em seis diferentes pontos de intersecção.*

Como mencionado, $n_1$ e $n_2$ são os nomes dados aos índices de refração dos materiais em ambos os lados de uma interseção de objeto de raio, com $n_1$ pertencendo ao ser material *saiu,* e $n_2$ pertencendo ao ser material *entrou.*

Para este teste, construa uma cena semelhante a esta em seção transversal:




A, B, e C são três esferas de vidro, com B e C contido por UMA e sobrepondo-se ligeiramente. Um raio é lançado através do centro de todos os três, e seu teste
deve verificar isso prepare_computations () pode determinar corretamente $n_1$ e $n_2$ em cada uma das interseções numeradas.

Este teste é apresentado como um *esboço do cenário* com certos nomes de variáveis entre colchetes angulares, como **<** este>. Uma tabela abaixo do cenário, chamada "Exemplos",

mostra quais valores devem ser inseridos no teste para cada variável, com as linhas representando chamadas separadas do teste.

**features / intersections.feature**

```
Linha 1  Esboço do Cenário : Encontrando n1 e n2 em vários cruzamentos
-            Dado UMA ← glass_sphere () com:
-                | transformar                          | escala (2, 2, 2) |
-                | material.refractive_index | 1,5                                  |
5            E B ← glass_sphere () com:
-                | transformar                          | tradução (0, 0, -0,25) |
-                | material.refractive_index | 2.0                                      |
-            E C ← glass_sphere () com:
-                | transformar                          | tradução (0, 0, 0,25) |
10               | material.refractive_index | 2,5                                      |
-            E r ← raio ( ponto (0, 0, -4), vetor (0, 0, 1))
-            E xs ← cruzamentos (2: A, 2,75: B, 3,25: C, 4,75: B, 5,25: C, 6: A)
-        Quando compor ← prepare_computations (xs [ <index> ], r, xs)
-        Então comps.n1 = <n1>
15           E comps.n2 = <n2>
-
-        Exemplos :
-            | índice    |   n1 |    n2 |
-            | 0         |  1,0 |   1,5 |
20          | 1         |  1,5 |   2,0 |
-           | 2         |  2,0 |   2,5 |
-           | 3         |  2,5 |   2,5 |
-           | 4         |  2,5 |   1,5 |
-           | 5         |  1,5 |   1,0 |
```

Aqui, os exemplos correspondem às interseções a serem testadas. Por exemplo, o exemplo na linha 19 diz que ao olhar para a interseção no índice 0,
$n_1$ deveria estar 1.0 e $n_2$ deveria estar 1,5. Esses valores são substituídos nas linhas 13-15, substituindo as variáveis **<** índice>, <n1>, e **<** n2>, e então o teste é executado.
Isso se repete para cada linha na tabela de exemplos.

Observe em particular o comportamento na interseção # 3, onde $n_1$ e $n_2$ são ambos 2,5. Esta é uma consequência de como as esferas B e C sobreposição. Esfera C é introduzido na interseção # 2, então quando a esfera B é encerrado em # 3, verifica-se que C, com um índice de refração de 2,5, está efetivamente em ambos os lados da interseção. Isso não vai acontecer com frequência, mas é por isso que é um bom caso de teste!

Agora, você já deve ter notado o novo argumento sendo passado para pré-pare_computations (intersecção, raio, xs). O terceiro argumento, xs, é a coleção de todas as interseções, o que pode dizer onde o acerto é em relação ao resto das interseções. Com isso, você pode decidir qual objeto, se houver, contém o objeto interceptado.

Adicionando um novo parâmetro a prepare_computations () afetará qualquer teste que você escreveu até agora que chame essa função, que é mais do que alguns. Se isso lhe incomoda, e se sua linguagem de programação oferece suporte a parâmetros opcionais, considere fazer o xs parâmetro opcional. Se não for fornecido, pode ser padronizado para uma coleção de um valor, o

interseção. Apenas certifique-se de atualizar (pelo menos) seu color_at () função para enviar a lista real de cruzamentos para prepare_computations ()!

O algoritmo funciona assim: comece com uma lista vazia, chamada recipientes, isso registrará quais objetos foram encontrados, mas ainda não foram encerrados. Esses objetos devem conter a interseção subsequente. Em seguida, iterando a coleção de interseções, faça o seguinte em cada interseção.

1. Se o cruzamento for o acertar, definir $n_1$ para o índice de refração do último objeto no recipientes Lista. Se essa lista estiver vazia, então não há nenhum objeto contido, e $n_1$ deve ser definido para 1
2. Se o objeto da interseção já estiver no recipientes lista, então este interseção deve estar saindo do objeto. Remova o objeto do recipientes lista neste caso. Caso contrário, a interseção está entrando no objeto, e o objeto deve ser adicionado ao final da lista.
3. Se o cruzamento for o acertar, definir $n_2$ para o índice de refração do último objeto no recipientes Lista. Se essa lista estiver vazia, então, novamente, não há objeto e $n_2$ deve ser definido para 1
4. Se o cruzamento for o acertar, termine o loop aqui.

Comopseudocódigo, é mais ou menos assim:

```
recipientes ← lista vazia

pra eu ← cada interseção em xs
    E se i = hit então
        E se os recipientes estão vazios
            comps.n1 ← 1.0
        outro
            comps.n1 ← último (contêineres) .material.refractive_index
        fim se
    fim se

    E se contêineres inclui i.object então
        remover i.object de contêineres
    outro
        anexar i.object em contêineres
    fim se
```

```
    E se i = hit então
        E se os recipientes estão vazios
            comps.n2 ← 1.0
        outro
            comps.n2 ← último (contêineres) .material.refractive_index
        fim se

        terminar o loop
    fim se
fim para
```

Adicione essa lógica para prepare_computations (), e passar no teste.

## Teste # 3: Computando under_point

*Mostra isso* **prepare_computations ()** *calcula um novo atributo,* **under_point**, *que fica logo abaixo da superfície interceptada.*

Construa uma interseção entre um raio e uma esfera de vidro de forma que a interseção ocorra em z = 0. Depois de preparar os cálculos, o under_point
atributo deve descrever um ponto logo abaixo da superfície da esfera, pouco mais do que z = 0.

```
features / intersections.feature
Cenário : O ponto inferior é deslocado abaixo da superfície
    Dado r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E forma ← glass_sphere () com:
            | transformar | tradução (0, 0, 1) |
        E eu ← interseção (5, forma)
        E xs ← cruzamentos (eu)
    Quando compor ← prepare_computations (i, r, xs)
    Então comps.under_point.z> EPSILON / 2
        E comps.point.z <comps.under_point.z
```

Observe que o resultado é comparado com a metade de EPSILON para se certificar de que foi ajustado na direção correta.

O objetivo deste novo atributo é descrever a origem dos raios refratados. Lembre-se em Renderizando sombras, na página 113 , como você calculou o over_point atributo de modo que foi deslocado apenas uma fração acima da superfície para evitar que os objetos se sombreie? É a mesma coisa aqui, só que em vez de levantar a questão *acima* a superfície, você empurra o ponto *abaixo de* a superfície.

Calcule esse atributo exatamente como você fez para comps.over_point, mas em vez de adicionar uma fração do vetor normal da superfície, você o subtrairá. Em pseudocódigo, ficará assim:

```
comps.point ← posição (raio, comps.t)

# e então, depois de computar e possivelmente negar
# o vetor normal ...

comps.over_point ← comps.point + comps.normalv * EPSILON comps.under_point ← comps.point -
comps.normalv * EPSILON
```

## Teste nº 4: Encontrando a cor refratada de um objeto opaco

*Apresente uma nova função,* **refracted_color (mundo, comps, restante),** *e mostre que isso retorna a cor preta quando o hit se aplica a um objeto opaco.*

Cruze um raio com a primeira esfera do mundo padrão. Depois de preparar o golpe, chamando refracted_color (mundo, comps, restante) deve retornar preto, porque a esfera não é transparente.

```
features / world.feature
Cenário : A cor refratada com uma superfície opaca
    Dado C ← default_world ()
        E forma ← o primeiro objeto em w
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E xs ← cruzamentos (4: forma, 6: forma)
    Quando compor ← prepare_computations (xs [0], r, xs)
        E c ← cor_refracionada (w, comps, 5)
    Então c = cor (0, 0, 0)
```

Fazer esta passagem requer refracted_color () para verificar o material do objeto atingido, retornando preto se transparência é 0 Por enquanto, certifique-se e devolva alguma outra cor (como branco) quando transparência é *não* 0, como isso:

```
função cor_refracionada (mundo, comps, restante)
    E se comps.object.material.transparency = 0
        Retorna cor (0, 0, 0)
    fim se

    Retorna cor (1, 1, 1)
função final
```

Isso ajuda a garantir que o teste falhe se você entender a lógica errada em seu código, enquanto constrói incrementalmente o resto do refracted_color () função.

## Teste # 5: Encontrando a Cor Refratada na Profundidade Máxima Recursiva

*Mostra isso* **refracted_color ()** *retorna a cor preta quando invocado na profundidade recursiva máxima, quando não há chamadas recursivas restantes disponíveis.*

Cruze um raio novamente com a primeira esfera do mundo padrão, mas desta vez dê à esfera um material vítreo. Então, invoque refracted_color () com o remanescente parâmetro definido para 0 Deve retornar a cor preta.

```
features / world.feature
Cenário : A cor refratada na profundidade recursiva máxima
    Dado C ← default_world ()
        E forma ← o primeiro objeto em w
        E forma tem:
            | material.transparency                    | 1,0 |
            | material.refractive_index | 1,5 |
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E xs ← cruzamentos (4: forma, 6: forma)
    Quando compor ← prepare_computations (xs [0], r, xs)
        E c ← cor_refracionada (w, comps, 0)
    Então c = cor (0, 0, 0)
```

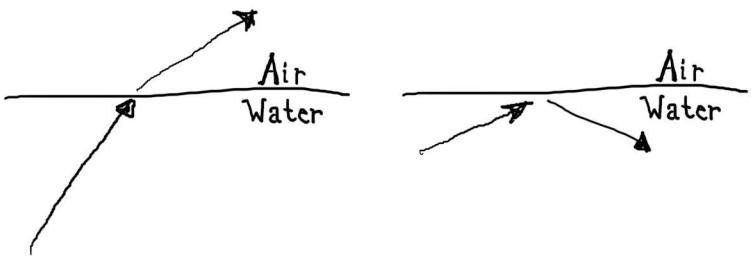
Para passar neste teste, sua função deve retornar preto se remanescente é 0

## Teste # 6: Encontrando a cor refratada sob reflexão interna total

*Mostra isso* **refracted_color ()** *retorna a cor preta quando as condições são adequadas para a reflexão interna total.*

Este caso trata de *reflexão interna total*. Não, não é o nome de uma banda de metal. Este é um fenômeno que ocorre quando a luz entra em um novo meio em um ângulo suficientemente agudo, e o novo meio tem um índice de refração mais baixo do que o antigo. Por exemplo, um raio de luz movendo-se da água para o ar pode experimentar reflexão interna total se atingir a interface entre eles em um ângulo pequeno o suficiente.

Quando essas condições forem verdadeiras, o raio refletirá na interface, em vez de passar por ela, como mostra a ilustração a seguir.




Isso, aliás, é o que permite que coisas como o cabo de fibra óptica funcionem.

Sob a reflexão interna total, a luz não é propagada pela interface entre os dois meios. Isso significa que o seu ray tracer deve retornar a cor preta quando ocorrer a reflexão interna total.

Construa uma cena onde o raio começa dentro da primeira esfera do mundo padrão e atinge aquela esfera em um ângulo suficientemente agudo. Deve resultar em reflexão interna total e a função deve retornar em preto.

```
features / world.feature
Cenário : A cor refratada sob reflexão interna total
    Dado C ← default_world ()
        E forma ← o primeiro objeto em w
        E forma tem:
            | material.transparency                    | 1,0 |
            | material.refractive_index | 1,5 |
        E r ← raio ( ponto (0, 0, √ 2/2), vetor (0, 1, 0))
        E xs ← cruzamentos (- √ 2/2: forma, √ 2/2: forma)
    # NOTA: desta vez você está dentro da esfera, então você precisa
    # para olhar para a segunda interseção, xs [1], não xs [0]
    Quando compor ← prepare_computations (xs [1], r, xs)
        E c ← cor_refracionada (w, comps, 5)
    Então c = cor (0, 0, 0)
```

A implementação disso depende de uma pequena coisa chamada *Lei de Snell,* que descreve a relação entre o ângulo do raio de entrada e o
ângulo do raio refratado. Dados esses dois ângulos, $\theta_{eu}$ e $\theta_{t}$ ( digamos, "theta i" e "theta t"), a Lei de Snell declara:

$$\frac{\text{pecado } \theta_{eu}}{\text{pecado } \theta_{t}} = \frac{\eta_2}{\eta_1}$$

Agora é uma questão de aplicar algumas identidades trigonométricas para transformar isso em informações que você pode usar em seu ray tracer, mas não entre em pânico! O que você precisa
fazer é encontrar $\theta_{eu}$, dado $\theta_{t}$, que funciona assim em pseudocódigo:

```
# Encontre a razão do primeiro índice de refração para o segundo.
# (Sim, isso é invertido da definição da Lei de Snell.)
n_ratio ← comps.n1 / comps.n2

# cos (theta_i) é o mesmo que o produto escalar dos dois vetores
cos_i ← ponto (comps.eyev, comps.normalv)

# Encontre sin (theta_t) ^ 2 por meio de identidade trigonométrica
sin2_t ← n_ratio ^ 2 * (1 - cos_i ^ 2)
```

Se sin2_t é maior que 1, então você tem alguma reflexão interna total acontecendo. Vá em frente e atualize seu refracted_color () para verificar este caso e retornar a cor preta quando isso acontecer.

## Teste # 7: Encontrando a Cor Refratada

*Mostra isso* **refracted_color ()** *em todos os outros casos, gerará um raio secundário na direção correta e retornará sua cor.*

Comece com o mundo padrão, mas torne a primeira esfera mais externa totalmente ambiente, para que apareça independentemente da iluminação. Aplique o padrão de teste de Generalizando padrões, na página 132 , para ele. A segunda esfera mais interna

recebe um material vítreo. Em seguida, gere um raio dentro da esfera mais interna, apontando diretamente para cima.

Lembre-se de que o padrão de teste retornará uma cor com base no ponto de intersecção, o que significa que o teste pode inspecionar a cor retornada para determinar se o raio foi refratado ou não. Sorrateira!

**features / world.feature**

```
Cenário : A cor refratada com um raio refratado
    Dado C ← default_world ()
        E UMA ← o primeiro objeto em w
        E A tem:
            | material.ambient | 1,0 | material.pattern | Padrão de teste () |         |

        E B ← o segundo objeto em w
        E B tem:
            | material.transparency                    | 1,0 |
            | material.refractive_index | 1,5 |
        E r ← raio ( ponto (0, 0, 0,1), vetor (0, 1, 0))
        E xs ← cruzamentos (-0,9899: A, -0,4899: B, 0,4899: B, 0,9899: A)
    Quando compor ← prepare_computations (xs [2], r, xs)
        E c ← cor_refracionada (w, comps, 5)
    Então c = cor (0, 0,99888, 0,04725)
```

Para fazer este teste passar, atualize seu refracted_color () funcionar novamente. Ele precisa fazer mais alguns cálculos para descobrir em qual direção o raio é refratado e, em seguida, gerar e avaliar esse raio refratado. Em pseudocódigo, funciona assim:

```
# Encontre cos (theta_t) por meio de identidade trigonométrica
custo ← sqrt (1.0 - sin2_t)

# Calcule a direção do raio refratado
direção ← comps.normalv * (n_ratio * cos_i - cos_t) -
                    comps.eyev * n_ratio

# Crie o raio refratado
refract_ray ← raio (comps.under_point, direção)

# Encontre a cor do raio refratado, certificando-se de multiplicar
# pelo valor de transparência para contabilizar qualquer opacidade
cor ← color_at (mundo, refract_ray, restante - 1) *
            comps.object.material.transparency
```

Só então.

### Teste # 8: Lidando com Refração em shadow_hit

*Mostre que seu* **sombra_hit ()** *função lida com refração.*

Adicione um piso de vidro ao mundo padrão, posicionado logo abaixo das duas esferas padrão, e adicione uma nova esfera colorida abaixo do piso. Lançar um raio diagonalmente

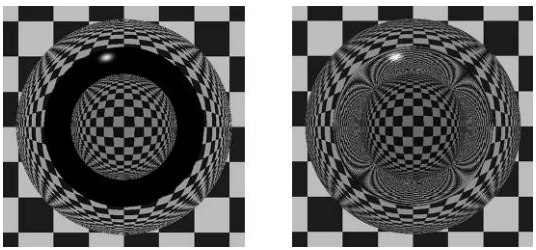
em direção ao chão, com a expectativa de que ele refrate e, por fim, atinja a bola colorida. Como o plano é apenas semitransparente, a cor resultante deve combinar a cor refratada da bola e a cor do plano.

**features / world.feature**

```
Cenário : sombra_hit () com um material transparente
    Dado C ← default_world ()
        E piso ← plano () com:
            | transformar                          | tradução (0, -1, 0) | | 0,5
            | material.transparency                                                |
            | material.refractive_index | 1,5                                      |
        E piso é adicionado a w
        E bola ← esfera () com:
            | material.color                | (1, 0, 0)                          |
            | material.ambient              | 0,5                                |
            | transformar                   | tradução (0, -3,5, -0,5) |
        E bola é adicionada a w
        E r ← raio ( ponto (0, 0, -3), vetor (0, - √ 2/2, √ 2/2))
        E xs ← cruzamentos ( √ 2 andar)
    Quando compor ← prepare_computations (xs [0], r, xs)
        E cor ← sombra_hit (w, comps, 5)
    Então color = cor (0,93642, 0,68642, 0,68642)
```

Faça isso chamando refracted_color () a partir de sombra_hit () e adicionando seu resultado à soma das cores refletidas e da superfície.

Neste ponto, as coisas parecem muito boas. Seu renderizador pode produzir lindos efeitos de refração, mas tenha cuidado: os resultados serão desagradáveis quando a reflexão interna total entrar em ação. Considere as duas imagens a seguir. Cada um representa uma esfera de vidro, com uma bolha de ar no meio.

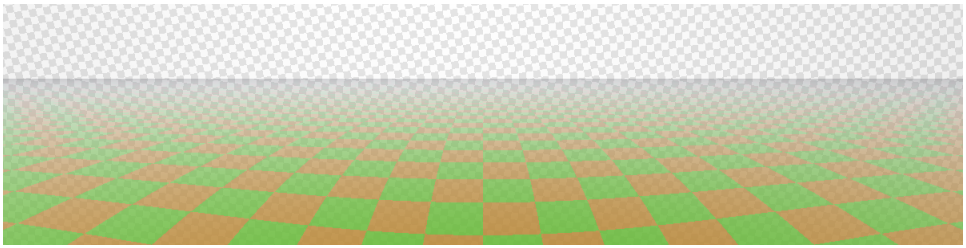


A esfera à esquerda é o que o seu ray tracer produzirá atualmente, adicionando a cor refratada à cor da superfície. Infelizmente, a reflexão interna total faz com que a interação entre o vidro e a bolsa de ar produza essa faixa preta. A esfera à direita, porém, usa um algoritmo mais realista que combina reflexão e refração, mitigando a banda. Tipo de “dia e noite”, certo?

O molho secreto aqui é o *Efeito Fresnel,* e a boa notícia é que não é muito mais trabalhoso adicionar ao seu ray tracer. Continue lendo para ver como isso funciona.

## Efeito Fresnel

O *Efeito Fresnel (* esse é um “s” silencioso, a propósito - graças aos franceses por isso) é o nome de como a luz se comporta em superfícies transparentes. Se você já esteve ao lado de um lago, você o conhece. Olhando direto para a água, você vê as rochas e os peixes abaixo da superfície. Mas quando você olha para a outra margem, a água se torna mais opaca e reflete mais e mais a paisagem. A figura a seguir demonstra isso, com um plano quadriculado marrom / verde para o fundo do lago e um segundo plano transparente como a água. Uma parede xadrez branca / cinza ao longe representa o cenário.



Observe especificamente como a “água” reflete fortemente a parede quadriculada branca / cinza quando está longe e de forma mais fraca em close-up. Da mesma forma, a água é fortemente transparente de perto, mas nem tanto à distância.

As fórmulas que descrevem esse comportamento foram deduzidas pela primeira vez no início do século 19 por Augustin-Jean Fresnel, um físico francês. A ideia básica é esta: quando o ângulo entre o olho e a superfície é grande ("olhando direto para a água"), a quantidade de luz refletida é pequena em relação à quantidade transmitida pela superfície, e quando o ângulo é pequeno (“Olhando para a outra margem”), a quantidade de luz refletida é maior.

Essa relação inversa entre reflexão e refração é o que corrige o “apagão” causado pela reflexão interna total. Ele é preenchido por reflexos - os raios refratados e refletidos se complementam, equilibrando bem as coisas.

As más notícias? As equações de Fresnel lidam com mais coisas do que nossa simulação se preocupa, como a polarização da luz. Embora seja certamente possível modelar tudo isso em software, fazer isso seria lento.

As boas notícias? Outro sujeito, Christophe Schlick, propôs uma aproximação às equações de Fresnel que é muito mais rápida e, além disso, bastante precisa. Viva o Schlick!

Para fazer isso funcionar, você implementará uma nova função, schlick (comps), que retorna um número entre 0 e 1, inclusive. Este número é chamado de *refletância* e representa qual fração da luz é refletida, dada a informação da superfície no acertar.

Você implementará o schlick () função com quatro testes:

1. Refletância quando ocorre a reflexão interna total.
2. Refletância quando um raio atinge uma superfície em um ângulo de 90 °.
3. Refletância quando $n_2$ é maior que $n_1$, e o ângulo é pequeno.
4. Refletância sempre usada por sombra_hit () quando uma superfície é reflexiva e transparente.

Aqui vai! Você está na reta final.

## Teste # 1: Determine a refletância sob a reflexão interna total

*Mostra isso* **schlick ()** *retorna 1 quando as condições são adequadas para a reflexão interna total.*

Posicione um raio dentro de uma esfera de vidro, deslocado do centro e apontando diretamente para cima. O raio é deslocado o suficiente para acionar a reflexão interna total, resultando em schlick () retornando 1.

```
features / intersections.feature
Cenário : A aproximação de Schlick sob reflexão interna total
    Dado forma ← glass_sphere ()
        E r ← raio ( ponto (0, 0, √ 2/2), vetor (0, 1, 0))
        E xs ← cruzamentos (- √ 2/2: forma, √ 2/2: forma)
    Quando compor ← prepare_computations (xs [1], r, xs)
        E refletância ← schlick (compor)
    Então refletância = 1,0
```

Intuitivamente, "reflexão interna total" significa que toda a luz é refletida e nenhuma é refratada. A fração de luz refletida deve ser 1 neste caso. Isso é chamado de *refletância.*

Faça esse teste passar implementando uma verificação de reflexão interna total. O seguinte pseudocódigo descreve como funciona:

```
função schlick (compor)
    # encontre o cosseno do ângulo entre o olho e os vetores normais
    porque ← ponto (comps.eyev, comps.normalv)

    # reflexão interna total só pode ocorrer se n1> n2
    E se comps.n1> comps.n2
        n ← comps.n1 / comps.n2 sin2_t = n ^ 2 * (1,0 - cos ^ 2)


        Retorna 1.0 E se sin2_t> 1.0
    fim se

    # retorna qualquer coisa menos 1.0 aqui, para que o teste falhe
    # apropriadamente se algo der errado.
    Retorna 0,0
função final
```

Lembre-se de que a reflexão interna total só pode acontecer quando $n_1$ é maior do que $n_2$, portanto, a própria verificação é protegida por essa condição. O porque variável, porém, será usada posteriormente na função e deve ser inicializada independentemente de se ocorre ou não a reflexão interna total.

## Teste # 2: Determine a refletância de um raio perpendicular

*Mostre essa refletância (via* **schlick ()** *é pequeno quando um raio atinge a superfície em um ângulo perpendicular.*

Crie uma esfera de vidro e um raio que a cruze. O raio deve atingir a esfera perpendicularmente à sua superfície. A refletância neste caso será leve.

**features / intersections.feature**

```
Cenário : A aproximação de Schlick com um ângulo de visão perpendicular
    Dado forma ← glass_sphere ()
        E r ← raio ( ponto (0, 0, 0), vetor (0, 1, 0))
        E xs ← cruzamentos (-1: forma, 1: forma)
    Quando compor ← prepare_computations (xs [1], r, xs)
        E refletância ← schlick (compor)
    Então refletância = 0,04
```

Não se preocupe em fazer isso passar agora; discutiremos a implementação no final do teste a seguir, e você fará com que ambos passem de uma só vez!

## Teste # 3: Determine a refletância quando $n_2 > n_1$

*Mostre essa refletância (via* **schlick ()** *é significativo quando $n_2 > n_1$ e o raio atinge a superfície em um pequeno ângulo.*

Este é o cenário “olhando através do lago para a margem oposta”, e uma quantidade significativa de luz deve ser refletida. O teste simula isso preparando um raio de forma que ele ricocheteie em uma esfera, quase tangente a ela.

```
features / intersections.feature
Cenário : A aproximação de Schlick com ângulo pequeno e n2> n1
    Dado forma ← glass_sphere ()
        E r ← raio ( ponto (0, 0,99, -2), vetor (0, 0, 1))
        E xs ← cruzamentos (1.8589: forma)
    Quando compor ← prepare_computations (xs [0], r, xs)
        E refletância ← schlick (compor)
    Então refletância = 0,48873
```

Faça este teste e o teste anterior passarem adicionando mais alguns cálculos ao seu schlick () função. O seguinte pseudocódigo termina adicionando as linhas indicadas:

```
função schlick (compor)
    # encontre o cosseno do ângulo entre o olho e os vetores normais
    porque ← ponto (comps.eyev, comps.normalv)

    # reflexão interna total só pode ocorrer se n1> n2
    E se comps.n1> comps.n2
        n ← comps.n1 / comps.n2 sin2_t = n ^ 2 * (1,0 - cos ^ 2)

        Retorna 1.0 E se sin2_t> 1.0

        # computar cosseno de theta_t usando identidade trigonométrica
        custo ← sqrt (1.0 - sin2_t)

        # quando n1> n2, use cos (theta_t) ao invés
        porque ← custo
    fim se

    r0 ← (( comps.n1 - comps.n2) / (comps.n1 + comps.n2)) ^ 2
    Retorna r0 + (1 - r0) * (1 - cos) ^ 5
função final
```

Provavelmente tudo isso parece magia, mas prometo que se baseia na realidade! Um excelente artigo chamado "Reflexões e refrações no rastreamento de raios", de Bram de Greve,[2] não é longo e vale a pena ler se você estiver curioso sobre a matemática por trás de tudo isso.

## Teste nº 4: EmployReflectanceWhenCombiningReflectionandRefraction

*Mostre que o* **schlick ()** *valor de refletância é usado por* **sombra_hit ()** *quando um material é transparente e reflexivo.*

Este é essencialmente o mesmo teste que Teste # 8: Lidando com Refração em sombra_ hit, na página 158 , mas o plano é feito tanto transparente *e* reflexivo. Isso fará com que a cor no ponto de intersecção incorpore os

2. Muitos sites online possuem uma cópia. Aqui está um: graphics.stanford.edu/courses/cs148-10-summer/docs/2006 - degreve - reflection_refraction.pdf .

cores refletidas e refratadas, combinando aquelas das esferas do mundo padrão com a esfera que foi adicionada abaixo do plano.

**features / world.feature**

```
Cenário : sombra_hit () com um material reflexivo transparente
    Dado C ← default_world ()
        E r ← raio ( ponto (0, 0, -3), vetor (0, - √ 2/2, √ 2/2))
        E piso ← plano () com:
            | transformar                    | tradução (0, -1, 0) |
            | material.refletivo              | 0,5                 |
            | material.transparency           | 0,5                 |
            | material.refractive_index | 1,5                         |
        E piso é adicionado a w
        E bola ← esfera () com:
            | material.color          | (1, 0, 0)                |
            | material.ambient        | 0,5                      |
            | transformar             | tradução (0, -3,5, -0,5) |
        E bola é adicionada a w
        E xs ← cruzamentos ( √ 2 andar)
    Quando compor ← prepare_computations (xs [0], r, xs)
        E cor ← sombra_hit (w, comps, 5)
    Então color = cor (0,93391, 0,69643, 0,69243)
```

Faça isso funcionar mudando seu sombra_hit () função, de modo que, em vez de retornar ingenuamente a soma das cores da superfície, refletida e refratada, você primeiro verificará se o material da superfície é transparente e reflexivo. Se for, você usará a aproximação de Schlick para combiná-los. O seguinte pseudocódigo demonstra:

```
função sombra_hit (mundo, comps, restante)
    sombreado ← is_shadowed (mundo, comps.over_point)

    superfície ← iluminação (comps.object.material,
                             comps.object,
                             world.light,
                             comps.over_point, comps.eyev, comps.normalv,
                             sombreado)

    refletido ← cor_refletida (mundo, comps, restante) refratado ← cor_refracionada (mundo, comps,
    restante)

    material ← comps.object.material
    E se material.refletivo> 0 && material.transparency> 0
        refletância ← schlick (compor)
        Retorna superfície + refletida * refletância +
                             refratado * (1 - refletância)
    outro
        Retorna superfície + refletida + refratada
    fim
função final
```

Lá! Isso deve bastar. Uma vez que todos os seus testes estão passando, você está pronto para renderizar reflexos e refrações genuínas, completos com efeitos Fresnel. Impressionante!

## Juntar as peças

Os rastreadores de raios são mais conhecidos por espelhos e vidros. Reserve um tempo e experimente, para ver por quê. Aqui estão algumas dicas para descobrir como empregar reflexão e refração efetivamente em suas cenas.

1. Temos a tendência de pensar no vidro como exclusivamente transparente, mas ninguém se surpreende ao olhar em uma janela e ver seu próprio reflexo fantasmagórico sobreposto à cena. Ao processar vidro ou qualquer material semelhante, defina ambos transparência e refletividade para valores altos, 0,9 ou mesmo 1. Isso permite que o efeito Fresnel apareça e dá ao seu material um toque adicional de realismo!

2 Como as cores refletidas e refratadas são adicionadas à cor da superfície, elas tendem a tornar esses objetos mais brilhantes. Você pode diminuir o tom do material difuso e ambiente propriedades para compensar. Quanto mais transparente ou reflexiva for a superfície, menor será o difuso propriedade deveria ser. Dessa forma, mais da cor vem dos raios secundários e menos da superfície do objeto.

3 Se você gostaria de um espelho sutilmente colorido, ou vidro levemente colorido, use um *Sombrio* cor, em vez de muito *claro* 1. O vidro vermelho, por exemplo, deve usar um vermelho muito escuro, quase preto, em vez de um vermelho muito brilhante. Em geral, quanto mais reflexiva ou transparente for a superfície, mais escura deve ser a cor da superfície. Observe que se você adicionar cor, certifique-se de ter algum difuso
e possivelmente ambiente contribuição também; caso contrário, sua superfície será renderizada como preta, independentemente da cor que você der a ela.

4 Superfícies reflexivas e transparentes combinam perfeitamente com realces especulares estreitos. Definir especular para 1 e bata brilho para 300 ou mais para obter um destaque que realmente brilhe.

Além disso, aqui está um desafio final para você: suponha que você queira renderizar uma cena em que está olhando através da superfície de um lago para algumas rochas abaixo dele. Em termos de implementação, seria um plano transparente, com algumas esferas espalhadas abaixo dele. Como seu ray tracer está implementado atualmente, o avião vai lançar uma sombra em qualquer coisa abaixo dele, o que deixa tudo sob a água na escuridão, arruinando o efeito. Você poderia adicionar uma fonte de luz abaixo do plano, mas isso apresentará sombras e realces estranhos - também não é uma boa solução.

O que você realmente deseja é que alguns objetos “optem pela exclusão” do cálculo de sombra. A superfície da lagoa, por exemplo, deve ser ignorada ao calcular sombras.

Como você mudaria seu ray tracer para suportar isso? O que você precisaria fazer para permitir que os objetos declarassem individualmente que não projetam sombras?

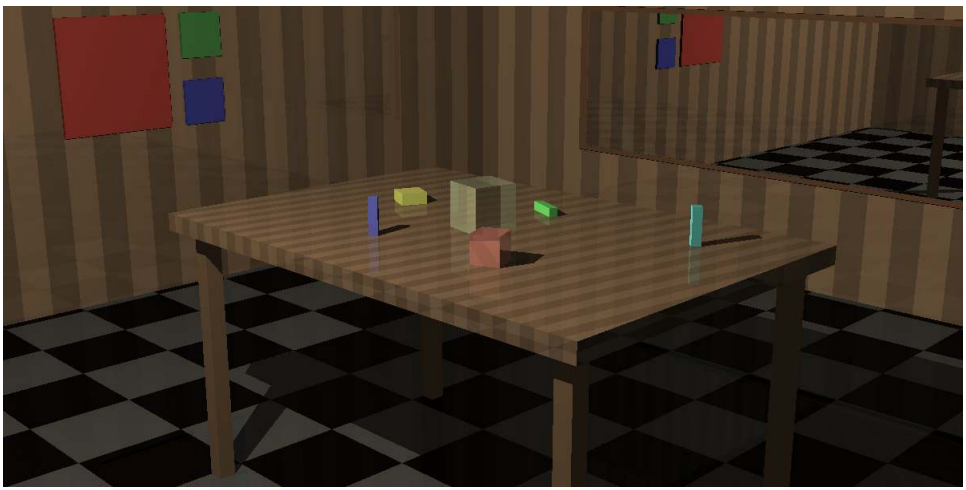
Mastigue um pouco. Quando estiver pronto para seguir em frente, vire a página! Em seguida, você adicionará outra forma primitiva ao seu ray tracer: o humilde cubo.

CAPÍTULO 12

# Cubos

As reflexões e refrações eram enormes, e você acertou em cheio! Suas cenas estão parecendo mais realistas do que nunca. Neste capítulo, você vai aumentar o escopo do que é possível, adicionando uma nova forma primitiva: o *cubo.*

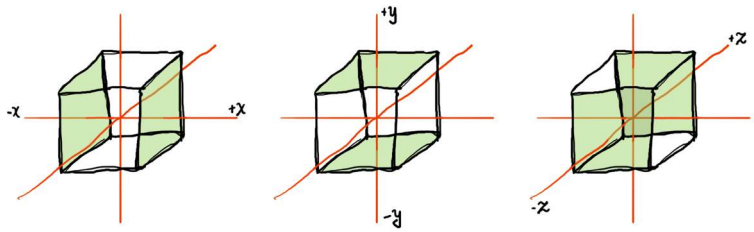
Confira. Aqui está uma cena renderizada *inteiramente com cubos.*



É verdade que a maioria dos cubos foi esticada e amassada de várias maneiras, mas - meu coração - todos eles começaram a vida como cubos perfeitos.

Na verdade, todos eles começaram a vida como um tipo muito específico de cubo, chamado de *caixa delimitadora alinhada ao eixo.* Neste capítulo, você adicionará suporte para eles implementando um novo algoritmo de interseção de raios, bem como o algoritmo para encontrar a normal na superfície de um cubo.

Uma caixa delimitadora alinhada ao eixo, ou AABB, é uma caixa com uma propriedade especial: seus lados estão todos alinhados com os eixos da cena. Dois estão alinhados com o x eixo, dois com o y eixo, e dois com o z eixo, como o figura na página 168 .




Esta restrição particular torna a interseção matemática *Muito de* menos complexo, o que torna o computador mais feliz e deve deixá-lo mais feliz também. A lógica do cubo em seu ray tracer implementará esses AABBs, então seus cubos sempre começarão centralizados na origem e se estendem de -1 a +1 ao longo de cada eixo. A partir daí, você pode usar matrizes de transformação para dimensionar, girar e traduzi-las para qualquer orientação que desejar.

Você já realizou a maior parte do trabalho árduo dos capítulos anteriores, construindo uma estrutura para apoiar, transformar e cruzar formas primitivas. Fantástico! Isso significa que este capítulo só precisa se concentrar em duas coisas: o algoritmo de interseção de raio-cubo e o algoritmo para encontrar a normal no cubo.

Vamos começar com o algoritmo de interseção.

## Cruzando um raio com um cubo

O algoritmo de interseção deve decidir se um dado raio cruza qualquer uma das seis faces do cubo ou se o raio perde o cubo completamente. Trate esses dois casos como testes, começando com o primeiro: um raio cruzando um cubo.

### Teste # 1: um raio cruza um cubo

*Mostre que o* **local_intersect ()** *função para um cubo identifica corretamente as interseções em qualquer face.*

Este teste cria um único cubo e, em seguida, projeta um raio em cada uma de suas faces para mostrar que o algoritmo funciona corretamente em todas as seis direções.

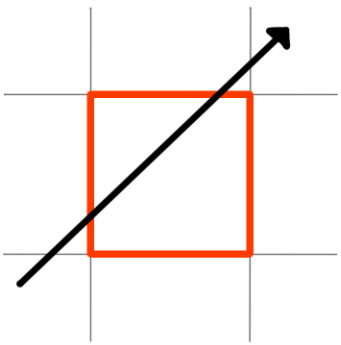
```
recursos / cubos.feature
Esboço do Cenário : Um raio cruza um cubo
    Dado c ← cubo ()
        E r ← raio ( <origin> , <direção> )
    Quando xs ← local_intersect (c, r)
    Então xs.count = 2
        E xs [0] .t = <t1>
        E xs [1] .t = <t2>
```

**Exemplos :**

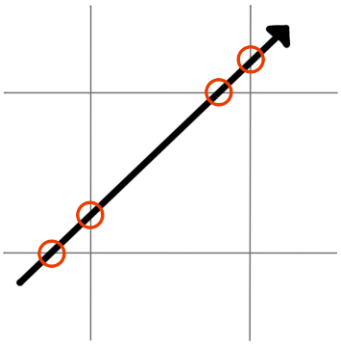
| | origem | direção | t1 | t2 |
|---|---|---|---|---|
| + x | ponto (5, 0,5, 0) | vetor (-1, 0, 0) | 4 | 6 |
| -x | ponto (-5, 0,5, 0) | vetor (1, 0, 0) | 4 | 6 |
| + y | ponto (0,5, 5, 0) | vetor (0, -1, 0) | 4 | 6 |
| -y | ponto (0,5, -5, 0) | vetor (0, 1, 0) | 4 | 6 |
| + z | ponto (0,5, 0, 5) | vetor (0, 0, -1) | 4 | 6 |
| -z | ponto (0,5, 0, -5) | vetor (0, 0, 1) | 4 | 6 |
| dentro | ponto (0, 0,5, 0) | vetor (0, 0, 1) | -1 | 1 |

O teste também lança um raio de *lado de dentro* o cubo, para mostrar que o algoritmo lida com esse caso também.

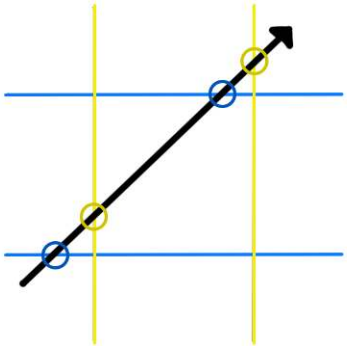
Isso funciona tratando um cubo como se fosse composto de seis planos, um para cada face do cubo. A intersecção de um raio com aquele cubo envolve testá-lo contra cada um dos planos, e se o raio os cruzar da maneira certa, isso significa que o raio cruza o cubo também. Vamos considerar o algoritmo em um nível mais simples, primeiro, para construir alguma intuição sobre como ele funciona. Comece observando a figura a seguir. Ele mostra um raio cruzando um quadrado 2D.



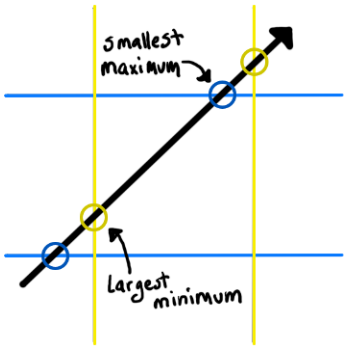
O primeiro passo é encontrar o t valores de todos os lugares onde o raio cruza essas linhas, como este:



Em seguida, considere-os em pares paralelos. A figura a seguir destaca os pares com duas interseções azuis em duas linhas azuis paralelas e duas interseções amarelas em duas linhas amarelas paralelas:



Para cada par de linhas, haverá um *mínimo* t mais próximo da origem do raio, e um *máximo* t ainda mais longe. Concentre-se no *maior* de todo o mínimo t valores e o *o menor* de todo o máximo t valores, assim:




A intersecção do raio com aquele quadrado será sempre esses dois pontos: o maior mínimo t valor e o menor máximo t valor. Isso também funciona para qualquer número de dimensões. Em três dimensões, você cruza planos em vez de linhas, mas ainda os considera em pares paralelos.

No pseudocódigo, a própria rotina de interseção se parece com isto:

```
função local_intersect (cubo, raio)
    xtmin, xtmax ← check_axis (ray.origin.x, ray.direction.x) ytmin, ytmax ← check_axis (ray.origin.y,
    ray.direction.y) ztmin, ztmax ← check_axis (ray.origin.z, ray.direction.z)


    tmin ← max (xtmin, ytmin, ztmin) tmax ← min (xtmax, ytmax,
    ztmax)

    Retorna ( interseção (tmin, cubo), interseção (tmax, cubo))
função final
```

Para cada um dos x, y, e z eixos, você verificará onde o raio cruza os planos correspondentes e retornará o mínimo e o máximo t valores para cada um. Depois de encontrar esses pontos de interseção, você encontra os pontos reais de interseção pegando o maior dos mínimos t valores e o menor do máximo t valores.

A função auxiliar, check_axis (), se parece com isto em pseudocódigo:

```
função check_axis (origem, direção)
    tmin_numerator = (-1 - origem) tmax_numerator = (1 -
    origem)

    E se abdômen (direção)> = EPSILON
        tmin ← tmin_numerator / direção tmax ← tmax_numerator /
        direção
    outro
        tmin ← tmin_numerator * INFINITY tmax ← tmax_numerator *
        INFINITY
    fim se

    E se tmin> tmax então trocar (tmin, tmax)

    Retorna tmin, tmax
função final
```

Isso leva a fórmula de interseção de plano de raio que você usou em Capítulo 9, Planos, na página 117 , e o generaliza para oferecer suporte a planos que estão deslocados da origem. Especificamente, cada par de planos é deslocado 1 unidade em direções opostas
ções, portanto - 1 - origem e 1 - origem.

Se o denominador ( direção) é efetivamente zero, embora você não queira se dividir por ele. O pseudocódigo anterior lida com este caso multiplicando os numeradores por infinito, o que garante tmin e tmax - embora ambos sejam infinitos - têm o sinal correto (positivo ou negativo).

Se sua linguagem de programação lida nativamente com infinito e divisão de ponto flutuante por zero, você pode evitar a maior parte da música e dança em check_axis () e apenas divida os numeradores pelo denominador. Nenhum caso especial necessário quando direção é zero!

Implemente isso e passe no primeiro teste. Assim que estiver funcionando, passe para o próximo teste!

### Teste 2: Um raio perde um cubo

*Mostre que o* **local_intersect ()** *A função para um cubo trata do caso em que o raio perde o cubo.*

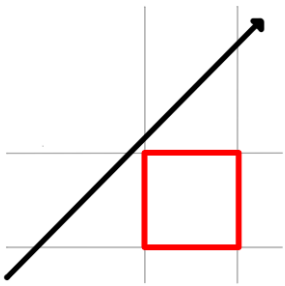
Mais uma vez, o teste cria um único cubo, mas desta vez os raios são lançados de tal forma que erram o cubo. Alguns são lançados paralelamente a faces diferentes, outros são apenas lançados diagonalmente longe do cubo.

**recursos / cubos.feature**

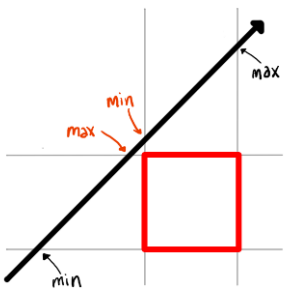
```
Esboço do Cenário : Um raio perde um cubo
    Dado c ← cubo ()
        E r ← raio ( <origin> , <direção> )
    Quando xs ← local_intersect (c, r)
    Então xs.count = 0

    Exemplos :
        | origem                    | direção                                    |
        | ponto (-2, 0, 0) | vetor (0,2673, 0,5345, 0,8018) | | ponto (0, -2, 0) | vetor (0,8018, 0,2673, 0,5345) | | ponto
        (0, 0, -2) | vetor (0,5345, 0,8018, 0,2673) | | ponto (2, 0, 2)

                                  | vetor (0, 0, -1)                           |
        | ponto (0, 2, 2)         | vetor (0, -1, 0)                           |
        | ponto (2, 2, 0)         | vetor (-1, 0, 0)                           |
```

Em cada caso, porém, o raio deve perder o cubo, resultando em interseções zero.

Considere novamente de uma perspectiva bidimensional. Na seguinte configuração, o raio perde o quadrado:



Mais uma vez, encontre os pontos de intersecção com os dois pares de linhas e, em seguida, encontre o maior dos mínimos t valores e o menor do máximo
t valores, como este:



Olhe de perto: o mínimo t está mais longe da origem do raio do que o máximo t! Bem, isso claramente não faz sentido, e a contradição é sua pista de que o raio perde o quadrado.

O pseudocódigo a seguir adiciona uma linha à implementação anterior, testando esse caso.

```
função local_intersect (cubo, raio)
    xtmin, xtmax ← check_axis (ray.origin.x, ray.direction.x) ytmin, ytmax ← check_axis (ray.origin.y,
    ray.direction.y) ztmin, ztmax ← check_axis (ray.origin.z, ray.direction.z)

    tmin ← max (xtmin, ytmin, ztmin) tmax ← min (xtmax, ytmax,
    ztmax)
➢   Retorna () E se tmin> tmax
    Retorna ( interseção (tmin, cubo), interseção (tmax, cubo))
função final
```

Depois de passar no teste, você está pronto para implementar a última parte para cubos: calcular o vetor normal.

## Encontrando o Normal em um Cubo

Lembre-se de que o *normal* é o vetor que aponta para fora perpendicularmente a uma superfície. Seu ray tracer usa-o para calcular uma variedade de efeitos, incluindo sombreamento, reflexão e refração. Felizmente, o algoritmo para encontrar o normal em um cubo é elegante e curto - dois atributos deliciosos!

Vamos pular direto para o teste.

### Teste # 3: O normal em um cubo

*Mostre que o* **local_normal_at ()** *A função calcula corretamente o normal em vários pontos de um cubo.*

Agora, cada face de um cubo é um plano com sua própria normal. Este normal será o mesmo em todos os pontos da face correspondente. O teste a seguir demonstra isso encontrando o normal em vários pontos de um cubo.

**recursos / cubos.feature**

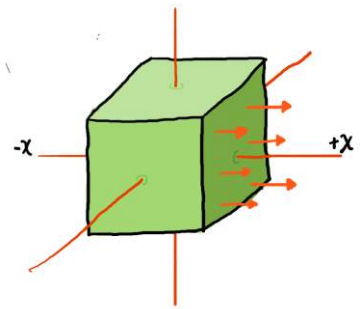
```
Esboço do Cenário : O normal na superfície de um cubo
    Dado c ← cubo ()
        E p ← < ponto>
    Quando normal ← local_normal_at (c, p)
    Então normal = <normal>
```

**Exemplos :**

| ponto | normal |
|---|---|
| ponto (1, 0,5, -0,8) | vetor (1, 0, 0) | | ponto (-1, -0,2, 0,9) | vetor (-1, 0, 0) | | ponto (-0,4, 1, -0,1) | vetor (0, 1, 0) | | ponto (0,3, -1, -0,7) | vetor (0, -1, 0) | | ponto (-0,6, 0,3, 1) | vetor (0, 0, 1) | | ponto (0,4, 0,4, -1) | vetor (0, 0, -1) | | ponto (1, 1, 1) | |
| | | vetor (1, 0, 0) |
| ponto (-1, -1, -1) | | vetor (-1, 0, 0) |

Observe que este teste também demonstra a normal em dois dos cantos do cubo, para garantir que o caso seja tratado de forma consistente. Especificamente, ele assume que os cantos são tratados como posições no + x ou - x faces e retorna o normal para essa face.

Para entender como o algoritmo para isso funcionará, observe que todos os pontos no + x rosto tem um normal apontando no + x direção, assim:




Escolher alguns pontos aleatoriamente daquele rosto nos dá a seguinte lista:

- (1,0, 0,0, -0,4)
- (1,0, -0,5, 0,6)
- (1,0, 0,1, 0,9)
- (1,0, -0,9, 0,7)

O que você nota aqui? Talvez você veja que o x componente não é só 1, mas também é sempre maior do que qualquer um dos outros componentes? Segure esse pensamento!

Considere a seguinte lista de pontos no - y face de um cubo:

- (-0,2, -1,0, 0,5)
- (0,1, -1,0, -0,9)
- (0,8, -1,0, 0,9)
- (-0,7, -1,0, 0,0)

Aqui o y componente é sempre - 1.0, e é *menos* do que qualquer um dos outros componentes.

Mais uma lista. Tente descobrir de qual face de um cubo cada um dos seguintes pontos é:

- (-1,0, 0,3, -0,5)
- (0,3, -0,9, 1,0)
- (-0,6, 1,0, 0,7)
- (0,4, -1,0, 0,2)

O rosto é sempre aquele que corresponde ao componente cujo valor absoluto é um 1!

Agora, na prática, você não pode confiar que os pontos que você obtém terão componentes exatamente iguais 1.0 ( maldição, arredondamento de ponto flutuante!), mas você pode fazê-lo funcionar escolhendo o componente com o maior valor absoluto. O pseudocódigo a seguir ilustra como seu local_normal_at () função deve funcionar para cubos.

```
função local_normal_at (cubo, ponto)
    maxc ← max ( abdômen (ponto.x), abdômen (ponto.y), abdômen (ponto.z))

    E se maxc = abdômen (ponto.x) então
        Retorna vetor (ponto.x, 0, 0)
    mais se maxc = abdômen (ponto.y) então
        Retorna vetor (0, ponto.y, 0)
    fim se

    Retorna vetor (0, 0, ponto.z)
função final
```

Em outras palavras, encontre o componente com o maior valor absoluto. Se isso for x, retornar um vetor apontando nessa direção. Se é y, retornar um vetor apontando para dentro *que* direção, e assim por diante.

Faça esse teste passar e seu cubo está pronto!

## Juntar as peças

Seu ray tracer agora suporta esferas, planos e cubos. Quão incrível é isso? Por suposto, experimente e veja o que você pode fazer combinando os três primitivos, mas primeiro: o que você pode fazer usando *só* cubos?

Experimente. Forme uma sala com um grande cubo. Faça uma mesa com cinco cubos: quatro para as pernas e um para a superfície da mesa. Coloque uma caixa na mesa. Espalhe algumas caixas no chão.

Aqui está outro desafio: usando apenas dois cubos, você pode fazer uma sala cujo piso e teto tenham uma textura diferente das paredes?

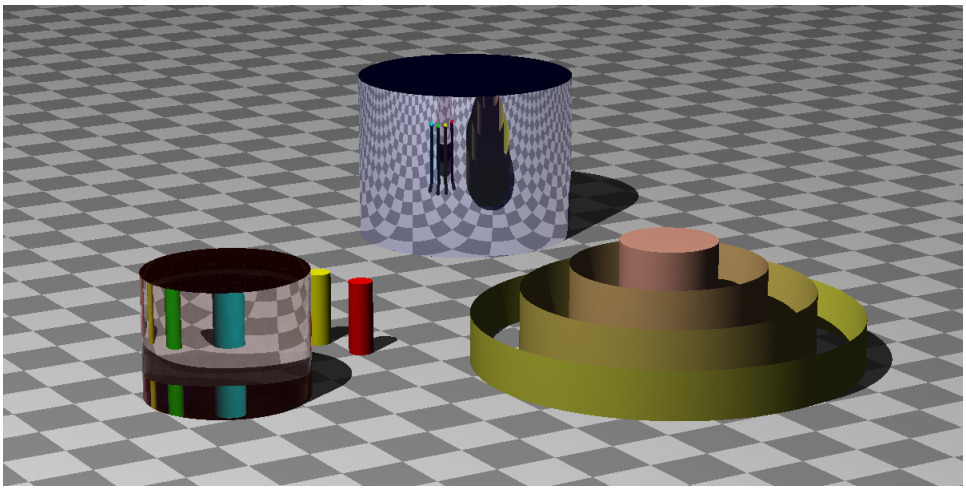
Você também pode tornar esses algoritmos mais rápidos. Por exemplo, ao comparar um raio com os lados do cubo, o algoritmo insiste em verificar todos os seis planos, mesmo que seja claro pela primeira ou segunda comparação que o raio falha. Em um raytracer de qualidade de produção, esse tipo de desperdício seria inaceitável. Como você pode otimizá-lo? O que você pode fazer para minimizar o número de comparações que ele faz?

Reflita sobre isso um pouco, se quiser. Quando estiver pronto, continue lendo. No próximo capítulo, você adicionará mais dois primitivos: cilindros e cones.

CAPÍTULO 13

# Cilindros

O próximo é o poderoso *cilindro.* Ele funciona bem com seu conjunto existente de primitivos gráficos e é fantástico para representar todos os tipos de coisas: braços, pernas, pescoços, dedos e torsos, bem como colunas, tubos e pernas de mesa. Aqui está um exemplo de cilindros em várias configurações para lhe dar uma ideia de como essa forma pode ser versátil:



Como acontece com todas as outras formas, você usará cilindros instanciando-os na origem e, em seguida, transformando-os no tamanho e na posição necessários. Por conveniência, você dará aos cilindros um raio padrão de 1, mas da forma como a matemática funciona, eles serão todos infinitamente longos, estendendo-se ao infinito em ambos

+ y e - y. Já que tentar fazer qualquer coisa útil com um cilindro infinitamente longo é complicado, você também implementará controles para permitir que seus cilindros sejam truncados em uma ou ambas as extremidades, e abertos ou tampados.

Você lidará com tudo isso em algumas etapas:

1. Implemente o algoritmo de interseção básico para um cilindro infinito de raio 1

2 Calcule o vetor normal para um cilindro.

3 Adicione suporte para truncar o cilindro. Por padrão, um cilindro truncado é *abrir,* ou oco.

4 Adicione suporte para as tampas, para permitir que o cilindro seja *fechado,* ou sólido. Calcule o vetor

5 normal nas extremidades.

Por último, como um bônus, você verá, brevemente, como interceptar um raio com um cone, o algoritmo para o qual acontece ser muito semelhante ao de um cilindro.

Você está pronto para isso? Aqui vai!

## Cruzando um Raio com um Cilindro

Ou o raio erra o cilindro ou atinge o cilindro. Direita? Essa dicotomia descreve claramente os testes que você escreverá primeiro. Você vai começar confirmando que um raio *sente falta* um cilindro. Esses testes podem ser feitos para passar trivialmente, mas ao invés de passar por eles, local_intersect método não faça nada, isso fornece uma boa oportunidade para começar a implementar as rotinas de interseção.

### Teste # 1: Um raio perde um cilindro

*Mostre que o* **local_intersect ()** *A função identifica corretamente quando um raio perde um cilindro.*

Este teste cria um cilindro e lança três raios diferentes nele. O primeiro raio é posicionado na superfície e aponta ao longo do + y eixo, paralelo ao cilindro. O segundo está dentro do cilindro e também aponta ao longo do + y eixo. O terceiro raio é posicionado fora do cilindro e orientado obliquamente em todos os eixos. Todos os três devem perder o cilindro.

```
features / cylinders.feature
Esboço do Cenário : Um raio perde um cilindro
    Dado cil ← cilindro ()
        E direção ← normalizar ( <direção> )
        E r ← raio ( <origin> , direção)
    Quando xs ← local_intersect (cil, r)
    Então xs.count = 0
```

**Exemplos :**

| origem | direção |
|---|---|
| ponto (1, 0, 0) | vetor (0, 1, 0) | | ponto (0, 0, 0) | vetor (0, 1, 0) | | ponto (0, 0, -5) | vetor (1, 1, 1) |

O algoritmo que implementa isso compartilha alguns recursos com o algoritmo de interseção de raio-esfera na página 57 . Tal como acontece com o algoritmo da esfera, você calculará um *discriminante* valor, que será negativo se o raio não se cruzar. Aqui estão alguns pseudocódigos:

```
função local_intersect (cilindro, raio)
    uma ← ray.direction.x² + ray.direction.z²

    # raio é paralelo ao eixo y
    Retorna () E se a é aproximadamente zero

    b ← 2 * ray.origin.x * ray.direction.x +
            2 * ray.origin.z * ray.direction.zc ← ray.origin.x² + ray.origin.z² - 1


    disco ← b² - 4 * a * c

    # raio não intercepta o cilindro
    Retorna () E se disco <0

    # este é apenas um espaço reservado, para garantir os testes
    # passe que espera que o raio falhe.
    Retorna ( interseção (1, cilindro))
função final
```

Observe que a última linha da função, retornando uma única interseção em t = 1, garante que os testes sejam aprovados porque o raio realmente perde o cilindro e não simplesmente porque a função não estava fazendo nada mais. Você vai aprofundar essa parte a seguir, no teste # 2.

### Teste # 2: um raio atinge um cilindro

*Mostre que o* **local_intersect ()** *A função identifica corretamente quando um raio atinge um cilindro.*

Mais uma vez, o contorno do cenário cria três raios diferentes, cada um dos quais deve interceptar o cilindro. O primeiro está configurado para atingir o cilindro em uma tangente, mas mesmo que a interseção real esteja em um único ponto, você ainda fará com que seu código retorne *dois* cruzamentos, ambos em t = 5. ( Isso imita como você lidou com interseções tangentes em Capítulo 5, Intersecções Ray-Sphere, na página 57 , e ajudará a determinar sobreposições de objetos em Capítulo 16, Geometria Sólida Construtiva (CSG), na página 227 .) O segundo raio

intercepta o cilindro perpendicularmente pelo meio e resulta em duas interseções em 4 e 6. O último raio é inclinado para que atinja o cilindro em um ângulo.

```
features / cylinders.feature
Esboço do Cenário : Um raio atinge um cilindro
    Dado cil ← cilindro ()
        E direção ← normalizar ( <direção> )
        E r ← raio ( <origin> , direção)
    Quando xs ← local_intersect (cil, r)
    Então xs.count = 2
        E xs [0] .t = <t0>
        E xs [1] .t = <t1>

    Exemplos :
        | origem                    | direção                    | t0          | t1          |
        | ponto (1, 0,        - 5)  | vetor (0, 0, 1) | vetor (0, 0, 1)  | 5           | 5           |
        | ponto (0, 0,        - 5)                                 | 4           | 6           |
        | ponto (0,5, 0, -5) | vetor (0,1, 1, 1) | 6,80798 | 7.08872 |
```

Faça isso usando o discriminante para encontrar o t valores para os pontos de intersecção. As linhas destacadas no seguinte pseudocódigo demonstram o cálculo de que você precisa:

```
função local_intersect (cilindro, raio)
    uma ← ray.direction.x² + ray.direction.z²

    # raio é paralelo ao eixo y
    Retorna () E se a é aproximadamente zero

    b ← 2 * ray.origin.x * ray.direction.x +
            2 * ray.origin.z * ray.direction.zc ← ray.origin.x² + ray.origin.z² - 1


    disco ← b² - 4 * a * c

    # raio não intercepta o cilindro
    Retorna () E se disco <0

➢   t0 ← (- b - √ ( disco)) / (2 * a) t1 ← (- b + √ ( disco)) / (2 * a)
➢
➢
➢   Retorna ( interseção (t0, cilindro), interseção (t1, cilindro))
função final
```

Tudo o que resta antes de você realmente renderizar este cilindro é calcular o vetor normal.

## Encontrando o Normal em um Cilindro

Depois de saber os pontos de intersecção, o vetor normal é usado para ajudar a sombrear a superfície de forma adequada. Você só precisará de um cenário para cobrir essa parte.

### Teste # 3: Vetor normal em um cilindro

*Mostre que o vetor normal na superfície de um cilindro é calculado corretamente.*

Este cenário escolhe quatro pontos na superfície do cilindro, um em cada + x, -x, + z e - z, e mostra que o normal é o valor esperado em cada ponto.

```
features / cylinders.feature
Esboço do Cenário : Vetor normal em um cilindro
    Dado cil ← cilindro ()
    Quando n ← local_normal_at (cil, <point> )
    Então n = <normal>

    Exemplos :
        | ponto                                | normal                                |
        | ponto (1, 0, 0) | vetor (1, 0, 0) | | ponto (0, 5, -1) | vetor (0, 0, -1) | | ponto (0,
        -2, 1) | vetor (0, 0, 1) | | ponto (-1, 1, 0) | vetor (-1, 0, 0) |
```

Para fazer isso, pegue o ponto em questão e remova o y componente. Tratar o resultado como um vetor fornece o normal. Em pseudocódigo, é assim:

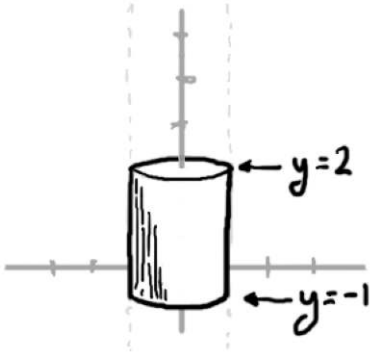
```
função local_normal_at (cilindro, ponto)
    Retorna vetor (ponto.x, 0, ponto.z)
função final
```

Com a aprovação desses testes, seu ray tracer pode renderizar cilindros! Eles serão infinitamente longos, o que pode ser um pouco difícil de manejar, mas com um pouco de imaginação você pode fazer todos os tipos de coisas interessantes com eles. De uma chance! Quando você voltar, veremos como truncar esses cilindros para torná-los mais fáceis de usar.

## Cilindros truncados

Imagine um mundo onde as pernas da mesa se estendam para sempre em ambas as direções, onde os lápis nunca podem ser apontados porque não têm ponta e onde os carros rodam sobre rodas infinitamente largas. Que bagunça! Talvez Salvador Dalí pudesse fazer algo com isso, mas para o resto de nós, esses cilindros são difíceis de usar bem. Para torná-los mais úteis, você pode *truncar* -los, cortando-os em uma ou ambas as extremidades.

Para o seu ray tracer, você implementará cilindros truncados, permitindo um *mínimo* e um *máximo* y valor a ser dado para cada cilindro. Por exemplo, o cilindro só existe entre y = -1 e y = 2 como mostrado no figura na página 182 .




Observe que as extensões são *exclusivo,* ou seja, se o cilindro estiver truncado em y = 2, o cilindro se estende até - mas não incluindo - esse limite.

Você precisará de apenas dois testes para este recurso: um que adiciona os novos atributos e outro que atualiza a lógica de interseção para suportar os cilindros truncados. Comece com os novos atributos.

## Teste # 4: limites mínimo e máximo

*Demonstre os valores padrão para os valores mínimo e máximo de um cilindro.*

Este cenário cria um novo cilindro e mostra que o padrão mínimo é infinito negativo e o padrão máximo é infinito positivo.

**features / cylinders.feature**

```
Cenário : O padrão mínimo e máximo para um cilindro
    Dado cil ← cilindro ()
    Então cil.minimum = -infinity
        E cil.maximum = infinito
```

O mínimo e o máximo sempre se referem às unidades no y eixo e são definidos no espaço do objeto. O próximo teste mostra como você usa esses atributos para realmente truncar um cilindro.

## Teste # 5: cilindros truncados

*Mostre que os cilindros em seu ray tracer podem ser truncados em qualquer uma das extremidades.*

Este cenário configura um cilindro, trunca-o em y = 1 e y = 2, e então lança vários raios nele para ter certeza de que o cilindro truncado está sendo cruzado corretamente.

**features / cylinders.feature**

```
Esboço do Cenário : Cruzando um cilindro restrito
    Dado cil ← cilindro ()
        E cil.minimum ← 1
```

```
    E cil.maximum ← 2
    E direção ← normalizar ( <direção> )
    E r ← raio ( <point> , direção)
Quando xs ← local_intersect (cil, r)
Então xs.count = <contagem>

Exemplos :
    |        | ponto                          | direção                        | contar      |
    | 1 ponto (0, 1,5, 0) | vetor (0,1, 1, 0) | 0 | 2 | ponto (0, 3, -5) | 3 | ponto (0, 0, -5)          |
                                               | vetor (0, 0, 1) | vetor (0, 0, 1)  | 0           |
                                               | vetor (0, 0, 1)                    | 0           |
    | 4 ponto (0, 2, -5) | 5 | ponto (0, 1, -5)                                     | 0           |
                                               | vetor (0, 0, 1)                    | 0           |
    | 6 ponto (0, 1,5, -2) | vetor (0, 0, 1)                                        | 2           |
```

Especificamente, os exemplos lançam os seguintes raios:

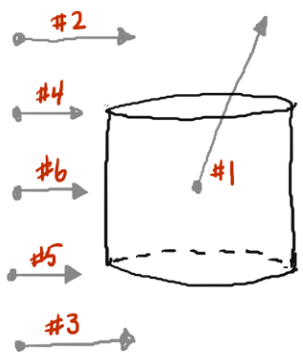
- O Exemplo 1 projeta um raio diagonalmente de dentro do cilindro, com o raio escapando sem interceptar o cilindro.

- Os exemplos 2 e 3 lançam raios perpendicularmente ao y eixo, mas de cima e de baixo do cilindro, e também perder.

- Os exemplos 4 e 5 são casos extremos, mostrando que o mínimo e o máximo y os próprios valores estão fora dos limites do cilindro.

- O exemplo final projeta um raio perpendicularmente ao meio do cilindro e produz duas interseções.

A figura a seguir mostra como a cena está configurada, com os raios correspondentes:




Para fazer isso funcionar, mude seu local_intersect método para que calcule o y coordenar em cada ponto de intersecção. Se o y a coordenada está entre o mínimo e maxmium valores, então a interseção é válida. O pseudocódigo a seguir mostra como isso funciona:

```
t0    ←    (-b - √ ( disco)) / (2 * a)
t1    ←    (-b + √ ( disco)) / (2 * a) t0> t1 então trocar (t0,
E se  t1)

xs    = ()

y0    ←    ray.origin.y + t0 * ray.direction.y
E se  cilindro.mínimo <y0 e y0 <cilindro.máximo adicionar interseção (t0, cilindro) para xs

fim se

y1 ← ray.origin.y + t1 * ray.direction.y
E se cilindro.mínimo <y1 e y1 <cilindro.máximo
    adicionar interseção (t1, cilindro) para xs
fim se

Retorna xs
```

Com essa mudança, todos os seus testes devem ser aprovados. Próximo: *sólido* cilindros!

## Cilindros Tampados

Se você já brincou com seus novos cilindros truncados, notou que eles são ocos, como pedaços de cano de PVC ou rolos de papel higiênico vazios. Este pode ser exatamente o efeito de que você precisa às vezes, mas em outras ocasiões você realmente deseja que os cilindros sejam *tampado,* ou fechado em cada extremidade. Para fazer isso, você precisa adicionar *tampas finais* - discos que cobrem exatamente cada extremidade do cilindro.

Esses discos são planos que são restritos à seção transversal do cilindro - e você implementou planos bem atrás Capítulo 9, Planos, na página 117 . Embora você não possa exatamente reutilizar seu código de avião para isso, os conceitos (com sorte!) Parecerão familiares.

Você adicionará tampas finais aos seus cilindros em três etapas:

1. Adicione um fechado atributo aos seus cilindros, indicando que os cilindros devem ser limitado.
2. Atualize seu cilindro local_intersect método para adicionar verificações para as tampas superior e inferior (se fechado é verdade).
3. Atualize seu cilindro local_normal_at método para calcular o normal nas extremidades (novamente, se fechado é verdade).

Primeiro, o fechado atributo.

### Teste # 6: Cilindros Fechados

*Mostre que seus cilindros possuem um* **fechado** *atributo, cujo padrão é* **falso.**

Configure um novo cilindro e mostre que o fechado atributo é falso, por padrão.

**features / cylinders.feature**

```
Cenário : O valor fechado padrão para um cilindro
    Dado cil ← cilindro ()
    Então cil.closed = falso
```

Faça essa passagem e, em seguida, você pode prosseguir para atualizar o algoritmo de interseção.

### Teste # 7: Cruzando as tampas das extremidades de um cilindro

*Mostre que sua rotina de interseção encontra corretamente os pontos de interseção entre um raio e as extremidades de um cilindro.*

Este esboço de cenário configura o mesmo cilindro truncado de antes, entre y = 1 e y = 2, mas também faz com que o cilindro se feche antes de lançar raios nele.

**features / cylinders.feature**

```
Esboço do Cenário : Cruzando as tampas de um cilindro fechado
    Dado cil ← cilindro ()
        E cil.minimum ← 1
        E cil.maximum ← 2
        E cyl.closed ← verdade
        E direção ← normalizar ( <direção> )
        E r ← raio ( <point> , direção)
    Quando xs ← local_intersect (cil, r)
    Então xs.count = <contagem>

    Exemplos :
        |        | ponto                          | direção                        | contagem |
        | 1 ponto (0, 3, 0) | 2 | ponto (0, 3, -2) | vetor (0, -1, 2) | 2 | 3 | 2 | ponto (0, 4, -2) | vetor (0, -1,      |
        1) | 2 | 4 ponto (0, 0, -2) | vetor (0, 1, 2) | 2 | 5 | ponto (0, -1, -2) | vetor (0, 1, 1) | 2                |
                                                                                                                  | # caso de canto
                                                                                                                  |
                                                                                                                  | # caso de canto
```

O raio no primeiro exemplo começa acima do cilindro e aponta para baixo no meio do cilindro, ao longo do y eixo. Deve cruzar ambas as extremidades, resultando em duas interseções.

Os exemplos 2 e 4 se originam (respectivamente) acima e abaixo do cilindro e projetam um raio diagonalmente através dele, cruzando uma tampa de extremidade antes de sair do outro lado do cilindro. Isso também resulta em duas interseções.

Os exemplos 3 e 5 são casos esquivos. Estes também se originam (respectivamente) acima e abaixo do cilindro, cruzando uma tampa de extremidade, mas eles saem do cilindro no ponto onde a outra tampa de extremidade intercepta a lateral do cilindro. Nesse caso, ainda deve haver apenas duas interseções: uma com a primeira tampa e outra onde a segunda tampa encontra a parede do cilindro.

Para implementar isso, você adicionará uma nova função, intersect_caps (cil, raio, xs). Ele verifica se o raio dado cruza as extremidades do cilindro dado, e adiciona os pontos de intersecção (se houver) ao xs coleção. Aqui está em pseudocódigo:

```
# uma função auxiliar para reduzir a duplicação.
# verifica se a interseção em `t` está dentro de um raio
# de 1 (o raio dos cilindros) do eixo y.
função check_cap (raio, t)
    x ← ray.origin.x + t * ray.direction.xz ← ray.origin.z + t * ray.direction.z


    Retorna ( x² + z²) <= 1
fim

função intersect_caps (cil, raio, xs)
    # tampas importam apenas se o cilindro estiver fechado, e pode possivelmente ser
    # interseccionado pelo raio.
    E se cil não está fechado ou ray.direction.y está próximo de zero
        Retorna
    fim se

    # verifique se há uma interseção com a tampa da extremidade inferior cruzando
    # o raio com o plano em y = cyl.minimum
    t ← ( cil.mínimo - ray.origin.y) / ray.direction.y
    E se check_cap (raio, t)
        adicionar interseção (t, cil) para xs
    fim se

    # verifique se há uma interseção com a tampa da extremidade superior cruzando
    # o raio com o plano em y = cyl.maximum
    t ← ( cil.maximum - ray.origin.y) / ray.direction.y
    E se check_cap (raio, t)
        adicionar interseção (t, cil) para xs
    fim se
função final
```

Primeiro, o raio é cruzado com um plano na extensão mínima. Em seguida, o ponto de intersecção é testado (por meio do check_cap () função auxiliar) para ver se ele está dentro do raio do cilindro. Em caso afirmativo, a interseção é adicionada à coleção. O mesmo processo segue para a extensão máxima.

Certifique-se de que o seu cilindro local_intersect function chama esta nova função depois de verificar se há interseções com as paredes do cilindro. Você também precisará alterar a lógica no início da função para que ela não retorne quando
uma é zero, caso contrário, sua interseção de limite será ignorada e pelo menos um de seus testes falhará. Em vez disso, se uma é zero, pule a lógica de interseção do cilindro e
apenas ligue intersect_caps ().

Você está quase terminando, mas antes de renderizar esses cilindros fechados, você precisa atualizar o cálculo do vetor normal para levar em conta as extremidades. Esse é o próximo teste.

## Teste # 8: Calculando o vetor normal nas extremidades

*Mostre que o cálculo do vetor normal leva em conta os cilindros fechados e retorna o normal correto nas extremidades.*

Este esboço de cenário cria um cilindro fechado e truncado e calcula a normal em vários pontos em cada extremidade:

```
features / cylinders.feature
Esboço do Cenário : O vetor normal nas tampas de extremidade de um cilindro
    Dado cil ← cilindro ()
        E cil.minimum ← 1
        E cil.maximum ← 2
        E cyl.closed ← verdade
    Quando n ← local_normal_at (cil, <point> )
    Então n = <normal>

    Exemplos :
        | ponto                                | normal                                |
        | ponto (0, 1, 0) | ponto (0,5, 1, 0) | vetor (0, 1, 0) | | ponto (0, 1, 0,5) | vetor (0,
        -1, 0) | | ponto (0, 2, 0)

                                               | vetor (0, 1, 0) |
        | ponto (0,5, 2, 0) | vetor (0, 1, 0) | | ponto (0, 2, 0,5) | vetor (0, 1, 0) |
```

As extremidades são planos, o que significa que uma extremidade tem a mesma normal em todos os pontos. O algoritmo deve verificar para ver a qual extremidade o ponto corresponde, ou ver se ele está no próprio cilindro, e retornar o vetor normal apropriado. Em pseudocódigo, é assim:

```
função local_normal_at (cilindro, ponto)
    # calcula o quadrado da distância do eixo y
    dist ← ponto.x² + ponto.z²

    E se dist <1 e ponto.y> = cilindro.máximo - EPSILON
        Retorna vetor (0, 1, 0)

    mais se dist <1 e ponto.y <= cilindro.mínimo + EPSILON
        Retorna vetor (0, -1, 0)

    outro
        Retorna vetor (ponto.x, 0, ponto.z)
    fim se
função final
```

Se o ponto estiver a menos de 1 unidade do y eixo, e está dentro EPSILON ( Vejo Comparando Números de Ponto Flutuante, na página 5 ) da extensão mínima ou máxima, então ele deve estar em uma das tampas. É importante que você inclua EPSILON aqui; se não fizer isso, você acabará com falhas de renderização causadas pelo vetor normal errado sendo calculado quando o arredondamento do ponto flutuante faz com que o ponto fique ligeiramente dentro de uma extremidade.

Mas é isso. Quando isso passar, você estará renderizando cilindros truncados e limitados. Dê uma chance!

O recurso ainda não acabou. Você vai encerrar implementando a primitiva de cone.

## Cones

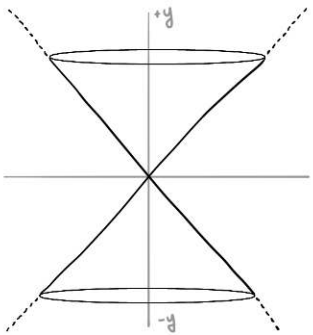
OK. Em seguida, você adicionará cones ao seu ray tracer e descobrirá que os cones são notavelmente semelhantes aos cilindros. Um verdadeiro *cone* tem estes recursos:

- É infinito em comprimento, assim como um cilindro.
- Pode ser truncado, como um cilindro.
- Pode ser fechado como um cilindro.

E eu realmente quero dizer *como um cilindro.* Você pode reutilizar uma boa parte do código que acabou de escrever para os cilindros.

É aqui que o desafio aumenta - vou tirar as rodinhas. Sem mãos dadas. Sem redes de segurança. Apenas um pouco de explicação, alguns testes e um monte de confiança em sua capacidade de fazer praticamente qualquer coisa que você colocar em sua mente.

Você vai implementar o que é chamado de *cone com dupla camada,* que a maioria das pessoas realmente chamaria *dois* cones: um de cabeça para baixo, o outro do lado direito para cima, com as pontas encontrando-se na origem e estendendo-se até o infinito em ambas as direções, conforme ilustrado na figura a seguir.



Para renderizar isso, você precisará implementar seu algoritmo de interseção e o algoritmo para calcular seu vetor normal.

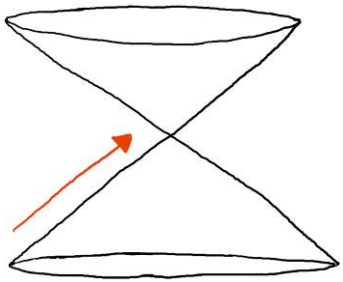
O algoritmo de interseção funciona quase exatamente como o do cilindro, mas a, b, e c são calculados de forma diferente. Dada a origem de um raio *o* e vetor de direção *d,* as seguintes fórmulas substituem as que você usou para os cilindros:

$$a = d_x^2 - d_y^2 + d_z^2$$

$$b = 2\, o_x d_x - 2\, o_y d_y + 2\, o_z d_z$$

$$c = o_x^2 - o_y^2 + o_z^2$$

Quando uma é zero, significa que o raio é paralelo a uma das metades do cone, assim:



Como você pode ver, isso ainda significa que o raio pode cruzar a outra metade do cone. Neste caso, o raio perderá quando ambos uma e b são zero. Se uma é zero mas b não for, você usará a seguinte fórmula para encontrar o único ponto de intersecção:

$$t = -c / 2b$$

Se uma for diferente de zero, você usará o mesmo algoritmo, mas com o novo a, b, e c, que você usou para os cilindros.

Aqui estão dois testes para ajudá-lo a verificar suas interseções de cone:

**features / cones.feature**

```
Esboço do Cenário : Cruzando um cone com um raio
    Dado forma ← cone ()
        E direção ← normalizar ( <direção> )
        E r ← raio ( <origin> , direção)
    Quando xs ← local_intersect (forma, r)
    Então xs.count = 2
        E xs [0] .t = <t0>
        E xs [1] .t = <t1>

    Exemplos :
        | origem                    | direção                    | t0          | t1          |
        | ponto (0, 0,        - 5) | vetor (0, 0, 1)            | 5           | 5           |
        | ponto (0, 0,        - 5) | vetor (1, 1, 1)            | 8,66025 | 8,66025 |
        | ponto (1, 1,        - 5) | vetor (-0,5, -1, 1) | 4.55006 | 49,44994 |
```

```
Cenário : Cruzando um cone com um raio paralelo a uma de suas metades
    Dado forma ← cone ()
        E direção ← normalizar ( vetor (0, 1, 1))
        E r ← raio ( ponto (0, 0, -1), direção)
    Quando xs ← local_intersect (forma, r)
    Então xs.count = 1
        E xs [0] .t = 0,35355
```

Você implementará tampas de extremidade para cones da mesma forma que fez para cilindros, mas com uma diferença: enquanto os cilindros têm o mesmo raio em todos os lugares, o raio de um cone mudará com y. Na verdade, o raio de um cone em qualquer y será o valor absoluto disso y. Isso significa que check_cap () função precisará ser ajustada para aceitar o y coordenada do avião que está sendo testado ( cone.minimum ou

cone.maximum, respectivamente) e tratar isso como o raio dentro do qual o ponto deve estar.

Aqui está um teste para as tampas das extremidades do cone para ajudá-lo com sua implementação:

**features / cones.feature**

```
Esboço do Cenário : Cruzando as tampas das extremidades de um cone
    Dado forma ← cone ()
        E forma.mínimo ← - 0,5
        E shape.maximum ← 0,5
        E shape.closed ← verdade
        E direção ← normalizar ( <direção> )
        E r ← raio ( <origin> , direção)
    Quando xs ← local_intersect (forma, r)
    Então xs.count = <contagem>

    Exemplos :
        | origem                                | direção                              | contagem |
        | ponto (0, 0, -5) | ponto (0, 0, -0,25) | vetor (0, 0, 1) | 2 |
        | ponto (0, 0, -5) | ponto (0, 0, -0,25) | vetor (0,
        1, 0) | 4
                                                                                        |
                                                                                        |
```

Por fim, para o vetor normal, calcule os limites normais da mesma forma que você fez para o cilindro, mas altere o resto para o seguinte, fornecido em pseudocódigo:

```
y ← √ ( ponto.x² + ponto.z²) y ← - y E se point.y> 0


Retorna vetor (ponto.x, y, ponto.z)
```

Novamente, aqui está um teste para ajudá-lo:

**features / cones.feature**

```
Esboço do Cenário : Calculando o vetor normal em um cone
    Dado forma ← cone ()
    Quando n ← local_normal_at (forma, <point> )
    Então n = <normal>
```

**Exemplos :**

| ponto | normal | |
|---|---|---|
| ponto (0, 0, 0) | ponto (1, 1, 1) | ponto (1, 0, 0) | vetor (1, 0, 0) | vetor (1, 1, 0) | vetor (1, 1, √ 2, 1) |

Tal como acontece com o cilindro infinito, um cone de dupla camada é um pouco pesado, mas, graças ao truncamento, você pode cortar qualquer pedaço desses cones duplos que não queira. Se você quiser um cone de unidade tradicional, por exemplo, você pode truncá-lo em y = -1 e y = 0, e, em seguida, traduzi-lo para 1 unidade em y.

## Juntar as peças

Agora você está armado com uma variedade de primitivos gráficos: esferas, planos, cubos, cilindros e cones. O que você pode fazer com eles? Aqui estão algumas idéias:

- Um sorvete de casquinha com uma (ou mais!) Colheres.
- O edifício do Capitólio dos EUA.
- Uma flecha.
- Uma lâmpada.
- Stonehenge.
- Uma escada em espiral.
- Um porta-retratos.
- O foguete Saturn V.
- Um lápis.

Se você se sente particularmente ambicioso, pode considerar tentar um modelo (simplificado!) De algo orgânico: uma árvore, um cachorro ou mesmo uma pessoa com uma figura de palito.

Quando estiver pronto, vire a página. Você aprenderá uma maneira mais fácil de construir modelos complexos, bem como uma otimização que pode ser usada para reduzir potencialmente o número de testes de interseção necessários para renderizar suas cenas.

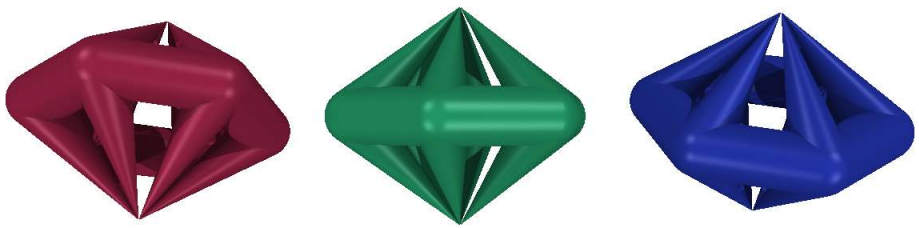
CAPÍTULO 14

# Grupos

Aqui estão, apenas mais alguns capítulos antes do final do livro. A esta altura, você já passou algum tempo brincando com seu renderizador, experimentando formas, padrões e composição, e provavelmente percebeu que construir uma cena com os primitivos à sua disposição envolve muito mexer nas transformações. Adicione uma forma, dimensione, traduza, gire *só então,* e repita para todas as outras formas de que você precisa em sua cena.

À medida que suas cenas crescem em complexidade, também aumenta o esforço necessário para modelá-las. Você já desejou uma maneira de agilizar as coisas? Não seria bom se você pudesse agrupar formas e transformá-las como uma unidade?

Aqui está um exemplo que faz exatamente isso. A figura a seguir mostra três visualizações diferentes de uma forma complexa composta de esferas, cilindros e cones.



Com o seu ray tracer em seu estado atual, cada esfera, cilindro e cone deve ser meticulosamente transformado em seu lugar, exigindo um rastreamento cuidadoso de cada componente e onde ele precisa ir. Mas, agrupando formas, formas complexas podem ser construídas na origem e, em seguida, transformadas como uma unidade, onde e como você quiser.

Neste capítulo, você adicionará suporte para grupos de formas, permitindo que eles sejam aninhados com a profundidade necessária e, como um bônus, você também lerá sobre como eles podem ser usados para otimizar seu ray tracer.

## Implementando Grupos

Os grupos são formas abstratas sem superfície própria, assumindo sua forma em vez das formas que contêm. Isso permite organizá-los em árvores, com grupos contendo outros grupos e primitivos concretos. O verdadeiro recurso matador dos grupos, porém, é que os grupos podem ser transformados como qualquer outra forma, e essas transformações, então, se aplicam implicitamente a quaisquer formas contidas pelo grupo. Você apenas coloca formas em um grupo, transforma o grupo e *voilà* —Tudo se aplica como uma única unidade.

Vamos fazer isso acontecer. Você lidará com isso em várias etapas:

1. Crie uma nova subclasse de forma chamada Grupo.
2. Adicione um novo atributo a Forma, chamado pai, que se refere ao grupo que contém a forma (se houver).
3. Escreva uma função para adicionar formas a um grupo. Implemente o
4. algoritmo de interseção de grupo de raios.
5. Implemente as alterações necessárias para calcular a normal em uma forma que faça parte de um grupo.

Esta seção descreve uma estrutura de árvore bidirecional, onde nós pais fazem referência a nós filhos e nós filhos fazem referência a nós pais. Nem todas as linguagens de programação tornam isso fácil de implementar. Se sua linguagem tornar isso desafiador, considere a leitura de todo o capítulo primeiro e, em seguida, implemente o recurso à sua maneira. Se tiver dúvidas, você pode pedir dicas no fórum.[1]

Comece criando seu novo Grupo classe para agregar formas.

### Teste nº 1: criando um novo grupo

*Um grupo é uma forma, que começa como uma coleção vazia de formas.*

Este teste apresenta uma nova função, grupo(), que retorna um novo Grupo instância. O teste mostra então que o grupo tem sua própria transformação (o que não é surpreendente, pois deveria ser um Forma subclasse), e a coleção que ela representa deve estar vazia.

1 forum.raytracerchallenge.com

**features / groups.feature**

```
Cenário : Criação de um novo grupo
    Dado g ← grupo ()
    Então g.transform = identity_matrix
        E g está vazio
```

Faça isso adicionando um Grupo classe, tornando-o um contêiner de formas, e fazendo com que se comporte como um Forma em si. O próximo teste irá abordar o Forma lado das coisas, adicionando um pai atributo.

## Teste # 2: uma forma tem um atributo pai

*Uma forma possui um pai opcional, que não é definido por padrão.*

Este teste requer um novo atributo em Forma, chamado pai, que pode estar desativado (o padrão) ou pode ser definido como um Grupo instância. Você verá o seu antigo test_shape () função de Formas de refatoração, na página 117 , usado aqui como uma forma genérica para demonstrar a adição do novo atributo.

**features / shapes.feature**

```
Cenário : Uma forma tem um atributo pai
    Dado s ← test_shape ()
    Então s.parent não é nada
```

A seguir, você escreverá uma função para adicionar formas como filhos de um grupo, ligando-os em uma espécie de árvore.

### Teste # 3: Adicionando uma criança a um grupo

*Adicionar uma criança a um grupo torna o grupo o pai da criança e adiciona a criança à coleção do grupo.*

Este teste adiciona uma nova função, add_child (grupo, forma) e mostra como é usado para adicionar uma forma filho a um grupo.

**features / groups.feature**

```
Cenário : Adicionando uma criança a um grupo
    Dado g ← grupo ()
        E s ← test_shape ()
    Quando add_child (g, s)
    Então g não está vazio
        E g inclui s
        E s.parent = g
```

Faça isso e você pode começar a passar para a diversão! É hora de cruzar os raios com esses grupos de formas.

## Testes # 4 e 5: Cruzando um Raio com um Grupo

*Dois testes mostram que um raio cruza um grupo se e somente se o raio cruza pelo menos uma forma filho contida pelo grupo.*

O primeiro teste é o caso trivial - lançar um raio e verificar se ele cruza com um grupo vazio. A coleção de cruzamentos resultante deve estar vazia.

**features / groups.feature**

```
Cenário : Cruzando um raio com um grupo vazio
    Dado g ← grupo ()
        E r ← raio ( ponto (0, 0, 0), vetor (0, 0, 1))
    Quando xs ← local_intersect (g, r)
    Então xs está vazio
```

O segundo teste constrói um grupo de três esferas e lança um raio nele. As esferas são organizadas dentro do grupo de modo que o raio cruze duas das esferas, mas perca a terceira. A coleção resultante de interseções deve incluir aquelas das duas esferas.

**features / groups.feature**

```
Cenário : Cruzando um raio com um grupo não vazio
    Dado g ← grupo ()
        E s1 ← esfera ()
        E s2 ← esfera ()
        E set_transform (s2, tradução (0, 0, -3))
        E s3 ← esfera ()
        E set_transform (s3, tradução (5, 0, 0))
        E add_child (g, s1)
        E add_child (g, s2)
        E add_child (g, s3)
    Quando r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
        E xs ← local_intersect (g, r)
    Então xs.count = 4
        E xs [0] .objeto = s2
        E xs [1] .objeto = s2
        E xs [2] .objeto = s1
        E xs [3] .objeto = s1
```

Para fazer com que esses dois testes passem, implemente o local_intersect () função para o seu Grupo moldar e fazer com que ele itere em todas as crianças do grupo, chamando intersectar () em cada um deles por sua vez. Deve agregar as interseções resultantes em uma única coleção e classificá-las todas por t.

## Teste # 6: Transformações de grupo

*Demonstre que as transformações de grupo e filho são aplicadas.*

Este teste cria um grupo e adiciona uma única esfera a ele. O novo grupo recebe uma transformação e a esfera recebe uma transformação diferente. Um raio é então lançado de tal maneira que deve atingir a esfera, enquanto a esfera está sendo transformada por sua própria transformação e pela de seu pai.

**features / groups.feature**

```
Cenário : Cruzando um grupo transformado
    Dado g ← grupo ()
        E set_transform (g, escala (2, 2, 2))
        E s ← esfera ()
        E set_transform (s, tradução (5, 0, 0))
        E add_child (g, s)
    Quando r ← raio ( ponto (10, 0, -10), vetor (0, 0, 1))
        E xs ← cruzar (g, r)
    Então xs.count = 2
```

A coisa boa sobre este teste é que ele já deve passar se o seu grupo

local_intersect () chamadas de função intersectar () em seus filhos. Certifique-se de que é assim.

Quando estiver pronto, continue lendo! A próxima peça deste quebra-cabeça requer encontrar o vetor normal em um objeto filho.

## Encontrando o normal em um objeto filho

Lembre-se de volta em Transformando normais, na página 79 , quando você usou a matriz de transformação da forma para manipular o vetor normal? A mesma coisa precisa acontecer ao calcular o normal em um objeto filho de um grupo, mas agora há uma complicação: quando uma interseção é encontrada com um

*grupo,* o próprio registro de interseção faz referência à interseção *criança*. Como seu ray tracer está implementado atualmente, isso significa que quando você calcula o vetor normal naquele objeto filho, apenas as transformações da criança são consideradas, e não as transformações de qualquer grupo ao qual a criança possa pertencer.

É nisso que você trabalhará a seguir, em três etapas:

- Escreva uma função que converta um ponto do espaço do mundo para o espaço do objeto, recursivamente levando em consideração qualquer objeto pai entre os dois espaços.

- Escreva uma função que converta um vetor normal do espaço do objeto para o espaço do mundo, novamente levando em consideração qualquer objeto pai entre os dois espaços.

- Atualize o normal_at () para que ele chame essas duas novas funções para transformar o ponto de entrada e o vetor de saída de forma adequada.

Entendi? Aqui vai!

### Teste # 7: Converta um Ponto do Espaço Mundial em Espaço do Objeto

*Pegue um ponto no espaço do mundo e transforme-o em espaço de objeto, levando em consideração qualquer objeto pai entre os dois espaços.*

Este teste constrói um grupo externo, que contém um grupo interno, que por sua vez contém uma esfera. Cada um recebe sua própria transformação antes de chamar uma nova função, world_to_object (forma, ponto), para converter um ponto do espaço mundial em espaço do objeto.

**features / shapes.feature**

```
Cenário : Convertendo um ponto de mundo em espaço de objeto
    Dado g1 ← grupo ()
        E set_transform (g1, rotação_y ( π / 2))
        E g2 ← grupo ()
        E set_transform (g2, escala (2, 2, 2))
        E add_child (g1, g2)
        E s ← esfera ()
        E set_transform (s, tradução (5, 0, 0))
        E add_child (g2, s)
    Quando p ← world_to_object (s, ponto (-2, 0, -10))
    Então p = ponto (0, 0, -1)
```

Faça este teste passar implementando world_to_object (forma, ponto). Se forma tem um pai, a função deve primeiro converter o ponto para o espaço de seu pai, chamando world_to_object (pai, ponto). O resultado é então multiplicado pelo inverso da transformação da forma. Em pseudocódigo, é assim:

```
função world_to_object (forma, ponto)
    E se forma tem pai
        ponto ← world_to_object (shape.parent, point)
    fim se

    Retorna inverso (shape.transform) * point
função final
```

Em seguida, você converterá um vetor de objeto em espaço mundial.

### Teste # 8: Converta um Vetor Normal de Espaço de Objeto em Espaço Mundial

*Pegue um vetor normal no espaço do objeto e transforme-o no espaço do mundo, levando em consideração qualquer objeto pai entre os dois espaços.*

Isso configura dois grupos aninhados como no teste anterior. Novamente, cada um recebe sua própria transformação e, em seguida, outra nova função, normal_to_world (forma, normal), é usado para transformar um vetor em espaço mundial.

**features / shapes.feature**

```
Cenário : Converter um normal de um objeto em um espaço mundial
    Dado g1 ← grupo ()
        E set_transform (g1, rotação_y ( π / 2))
```

```
        E       g2 ← grupo ()
        E       set_transform (g2, escala (1, 2, 3))
        E       add_child (g1, g2)
        E       s ← esfera ()
        E       set_transform (s, tradução (5, 0, 0))
        E       add_child (g2, s)
    Quando n ← normal_to_world (s, vetor ( √ 3/3, √ 3/3, √ 3/3))
    Então n = vetor (0,2857, 0,4286, -0,8571)
```

Você pode fazer este teste passar primeiro convertendo o dado normal para o espaço do objeto pai usando o algoritmo que você implementou em Transformando normais, na página 79 . Pegue o inverso da transformação da forma, transponha o resultado e multiplique-o pelo vetor. Normalize o resultado. Então, se a forma tiver um pai, passe recursivamente o novo vetor para normal_to_world (pai, normal). Esta é a implementação em pseudocódigo:

```
função normal_to_world (forma, normal)
    normal ← transpor ( inverso (shape.transform)) * normal normal.w ← 0

    normal ← normalizar (normal)

    E se forma tem pai
        normal ← normal_to_world (shape.parent, normal)
    fim se

    Retorna normal
função final
```

Depois que esses testes forem aprovados, você estará pronto para encontrar o normal em um objeto filho.

### Teste # 9: Encontre o Normal em um Objeto em um Grupo

*Encontre o normal em um objeto filho de um grupo, levando em consideração as transformações no objeto filho e no (s) pai (s).*

Como nos dois testes anteriores, este estabelece uma hierarquia de dois grupos e uma esfera e atribui a cada um deles uma transformação. Em seguida, ele encontra o vetor normal em um ponto da esfera (no espaço mundial), usando o normal_at () função.

```
features / shapes.feature
Cenário : Encontrar o normal em um objeto filho
    Dado g1 ← grupo ()
        E set_transform (g1, rotação_y ( π / 2))
        E g2 ← grupo ()
        E set_transform (g2, escala (1, 2, 3))
        E add_child (g1, g2)
        E s ← esfera ()
        E set_transform (s, tradução (5, 0, 0))
        E add_child (g2, s)
    Quando n ← normal_at (s, ponto (1.7321, 1.1547, -5.5774))
    Então n = vetor (0,2857, 0,4286, -0,8571)
```

Em seguida, atualize seu normal_at () função para usar o seu novo world_to_object () e normal_to_world () , chamando o primeiro para converter o ponto do espaço mundial em espaço do objeto antes de calcular o normal e, em seguida, chamando o último para converter o normal de volta ao espaço mundial. Em pseudocódigo, você atualizou normal_at ()
função deve vir junto assim:

```
função normal_at (forma, world_point)
    local_point ← world_to_object (forma, world_point) local_normal ← local_normal_at (forma,
    local_point)
    Retorna normal_to_world (forma, local_normal)
função final
```

Você está quase terminando com os grupos, mas há mais um detalhe a ser abordado. No
Padrões de transformação, na página 130 , você permitiu que os padrões fossem transformados pela conversão de pontos do espaço do mundo em espaço do objeto, e de lá para o espaço do padrão, antes de calcular a cor. Para que esses padrões se comportem bem quando aplicados a objetos em grupos, você precisará usar este novo
world_to_object () função ao converter pontos do espaço do mundo para o espaço do objeto. Caso contrário, os padrões não aplicarão as transformações de grupo e não terão a aparência esperada. Você está por sua própria conta para este; faça assim!

**Joe pergunta:**

### Onde está a função local_normal_at do grupo?

Ah, você percebeu que estava faltando! Muito bem, mas não é um erro ou descuido. Porque normais são sempre calculados chamando a forma concreta local_normal_at () função, o próprio grupo não precisa de um. Na verdade, se o seu código tentar chamar local_normal_at ()
em um grupo, isso significa que há um bug em algum lugar.

Considere implementar local_normal_at () para grupos, mas a implementação lança uma exceção ou causa um erro. Isso pode ajudá-lo a detectar esses bugs mais cedo e tornar explícito que os grupos são abstratos e não têm vetores normais.

Isso deve ser suficiente para a implementação de grupos. Em breve, você fará um exercício para se familiarizar com como usá-los, mas, primeiro, vamos dar uma olhada rápida em como esses grupos podem ser usados para otimizar seu ray tracer.

## Uso de caixas delimitadoras para otimizar cenas grandes

Uma das coisas mais caras do ponto de vista computacional que um ray tracer faz é encontrar as interseções entre um raio e um objeto, e o que torna as coisas ainda piores é que ele tem que fazer isso repetidamente para cada pixel. Para renderizar uma cena de dez objetos em uma tela pequena de 200 × 200, o seu ray tracer deve realizar *pelo menos*
400.000 testes de interseção, além de quantas interseções adicionais são

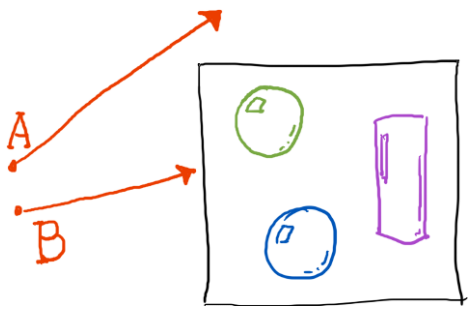
necessário para gerar sombras, reflexos e refração. Para piorar a situação, a maioria desses raios nem chega perto da maioria dos objetos em uma cena típica.

Que desperdício, certo? Se você pudesse testar apenas os objetos que estavam razoavelmente próximos de qualquer raio ...

A boa notícia é que há uma variedade de técnicas diferentes para ensinar ao seu ray tracer como fazer isso. Eles estão um pouco além do escopo deste livro, mas vamos dar uma olhada rápida em um dos menos complicados: *caixas delimitadoras.*

Se esse termo lhe parecer familiar, provavelmente é porque você o viu usado em Capítulo 12, Cubos, na página 167 , quando os cubos foram chamados *caixas delimitadoras alinhadas ao eixo (* AABB). A ideia por trás dessa otimização é usar esses cubos, ou caixas delimitadoras, para conter um grupo de outros objetos. Então, ao interceptar o grupo com um raio, você primeiro testa o raio contra a caixa delimitadora. Se o raio falhar, testar qualquer coisa dentro da caixa é inútil, porque ele também teria que errar.

A figura a seguir ilustra isso com uma caixa delimitadora que contém três formas. Visto que o raio A perde a caixa delimitadora, não há necessidade de ver se ele cruza qualquer uma das formas dentro dele. No entanto, porque o raio B *faz* cruze a caixa, você precisa testar aquele raio contra as formas que ele contém.




Não vou explicar isso, mas tente mesmo assim. Embora implementar isso definitivamente tenha algumas partes complicadas, tenho fé em você! Aqui está um esboço básico do que você precisa fazer:

1. Crie um Limites estrutura que descreve o mínimo e o máximo extensões (coordenadas) para a caixa. Você pode armazená-los como dois pontos, onde um tem o mínimo x, y, e z coordenadas e o outro tem o máximo.

2 Faça um limites (forma) função que retorna os limites para a forma dada, no espaço do objeto. Isto é o *não transformado* limites, então uma esfera (por exemplo) sempre se estenderá de -1 a 1 em x, y, e z. Algumas formas

(planos, cilindros não truncados e outros) se estenderão ao infinito em uma ou mais dimensões, portanto, certifique-se de que você pode lidar com esse caso. Faça um limites (grupo) função que converte os limites

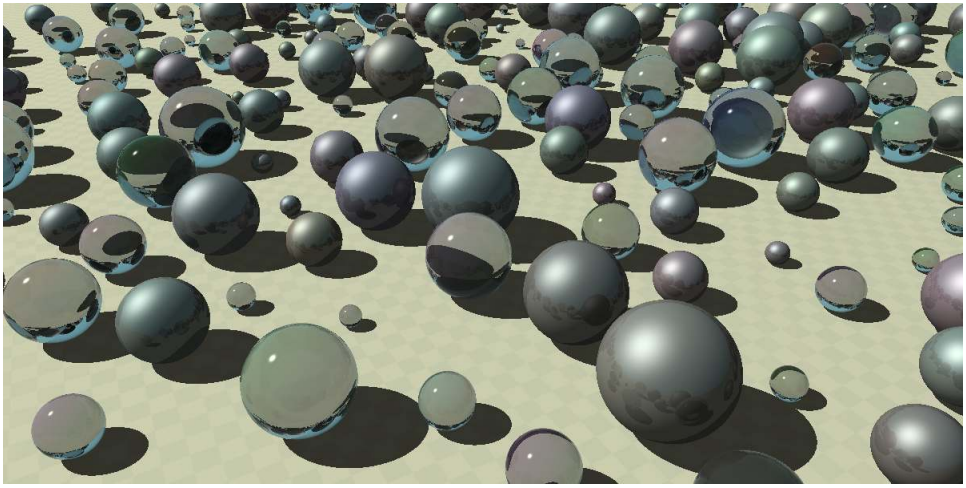
3 de todos os filhos do grupo em "espaço do grupo" e, em seguida, os combina em uma única caixa delimitadora. Esta é uma daquelas partes complicadas! Aqui estão duas dicas, no entanto. Primeiro, para converter um ponto do espaço do objeto em seu espaço pai, multiplique o ponto pela matriz de transformação do objeto. Em segundo lugar, ao transformar uma caixa delimitadora inteira, primeiro transforme todos os oito cantos do cubo e, em seguida, encontre uma única caixa delimitadora que se encaixa em todos eles. Se você não consegue ver por que precisa transformar todos os oito pontos, imagine girar a caixa 45 ° em torno de qualquer eixo e, em seguida, descubra qual é o novo *alinhado ao eixo*

deve ser semelhante a uma caixa delimitadora.

4 Reutilize o algoritmo de interseção do seu cubo, alterando-o para que aceite AABBs em localizações arbitrárias (mas ainda alinhadas ao eixo). Para fazer isso, você precisará alterar o - 1 e a 1 no check_axis () função seja, respectivamente, o valor mínimo e máximo para o eixo que está sendo testado. Então, se você estiver testando o z eixo, e a caixa delimitadora vai de z = -5 para z = 3, você usaria - 5 ao invés de - 1, e 3 ao invés de 1

5 Faça o local_intersect (grupo, raio) função primeiro teste o raio contra a caixa delimitadora do grupo. Somente se o raio cruzar a caixa delimitadora, o raio deve ser testado contra as crianças.

Comoum exemplo de como essa técnica pode ajudar, juntei o seguinte cena com mais de 280 mármores com texturas de vidro e metálicas:



Renderizada sem caixas delimitadoras em 1200x600 pixels, esta imagem exigia mais de *1,8 bilhão* testes de interseção, dos quais apenas 1% realmente atingiu alguma coisa. Usando dezesseis caixas delimitadoras, porém, e organizando-as em

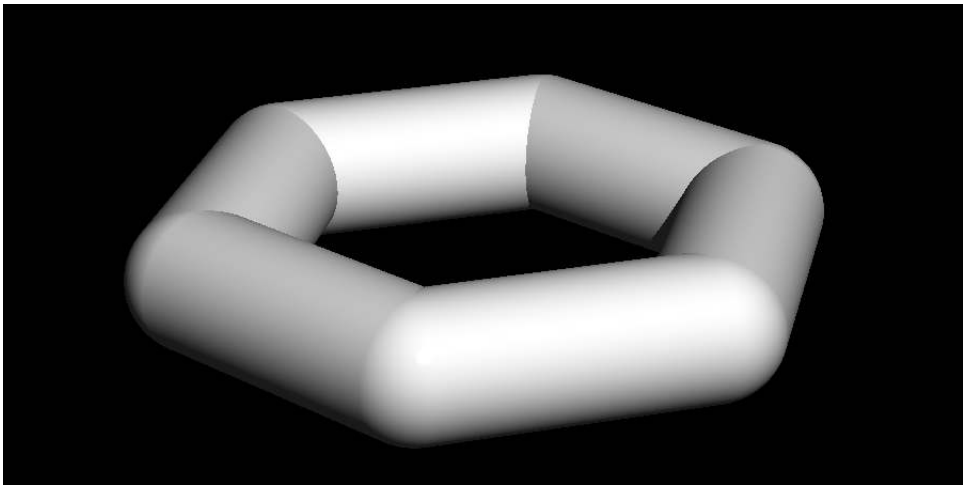
uma grade 4x4 para que todos os mármores fossem cobertos, a renderização exigia apenas um pouco mais do que *180 milhões* testes de interseção, com 10% acertando algo. Isso é uma ordem de magnitude melhor, apenas adicionando caixas delimitadoras!

Mas aqui está a ressalva: como acontece com qualquer otimização, não é uma vitória garantida em todas as situações. Nem toda cena se beneficiará com caixas delimitadoras, e alguns podem até ver *pior* performance (dependendo dos objetos na cena e como você os organiza).

Ainda assim, é uma otimização útil e merecerá sua manutenção no próximo capítulo, Capítulo 15, Triângulos, na página 207 . Dê uma chance!

## Juntar as peças

Vamos encerrar com um exemplo de como você pode usar grupos em suas cenas. Você vai construir um modelo de hexágono usando cilindros e esferas, assim:



Você construirá isso definindo primeiro uma única instância de cada componente: uma esfera (para se tornar os cantos do hexágono) e um cilindro (para se tornar as bordas). Você transformará cada um em um lugar uma vez e os adicionará a um grupo. Em seguida, você criará duplicatas desse grupo, girando cada duplicata em torno do y eixo até que todo o hexágono seja construído.

Comece escrevendo uma função que cria o componente da esfera prototípica, dimensionando-o em 25% e traduzindo-o em -1 unidade em z. O seguinte pseudocódigo mostra isso como uma função chamada hexagon_corner ().

```
função hexagon_corner ()
```

```
    canto ← esfera ()
    set_transform (canto, tradução (0, 0, -1) *
                                escala (0,25, 0,25, 0,25))
    Retorna canto
função final
```

Lembre-se de que quando você combina transformações de matriz, você o faz em *ordem reversa*. Assim, embora o pseudocódigo para hexagon_corner () multiplica a tradução pela escala, o resultado é que a esfera é dimensionada primeiro e *então* traduzido.

Em seguida, escreva uma função que crie o componente do cilindro prototípico. Limite a um mínimo de y = 0 e um máximo de y = 1, e dimensioná-lo em 25% em x e z. Girá-lo - π/2 radianos em z ( derrubá-lo) e - π/6 radianos em y ( para orientá-lo como uma borda). Em seguida, traduza -1 unidade em z. Em pseudocódigo, este hexagon_edge () função pode ter a seguinte aparência:

```
função hexagon_edge ()
    Beira ← cilindro ()
    edge.minimum ← 0
    edge.maximum ← 1
    set_transform (Beira, tradução (0, 0, -1) *
                                rotação_y (- π / 6) *
                                rotação_z (- π / 2) *
                                escala (0,25, 1, 0,25))
    Retorna Beira
função final
```

A próxima etapa é juntar essas duas primitivas em um grupo, formando um lado do hexágono. Os seguintes hexagon_side () função demonstra isso em pseudocódigo.

```
função hexagon_side ()
    lateral ← grupo ()

    add_child (lateral, hexagon_corner ())
    add_child (lateral, hexagon_edge ())

    Retorna lateral
função final
```

Depois de obter uma função que pode retornar um único lado do hexágono, você pode escrever a função final, hexágono(), que chama hexagon_side () seis vezes e gira cada peça no lugar, assim:

```
função hexágono ()
    hex ← grupo ()

    pra n ← 0 a 5
        lateral ← hexagon_side ()
        set_transform (lateral, rotação_y (n * π / 3))
```

```
        add_child (hex, lado)
    fim para

    Retorna hex
função final
```

A partir daí, você pode adicionar uma fonte de luz e uma câmera e enlouquecer com isso!

Que outras formas compostas você pode construir? Tente criar um boneco palito ou um automóvel. Árvores e plantas também são definitivamente possíveis e se prestam bem a algoritmos fractais como os sistemas de Lindenmayer.

Além disso, você logo perceberá que os materiais aplicados a um grupo não têm efeito algum nas formas que ele contém. E se você quisesse que as formas em seu ray tracer fossem capazes de “herdar” materiais de seus pais? Como você pode estender seu código para fazer isso acontecer?

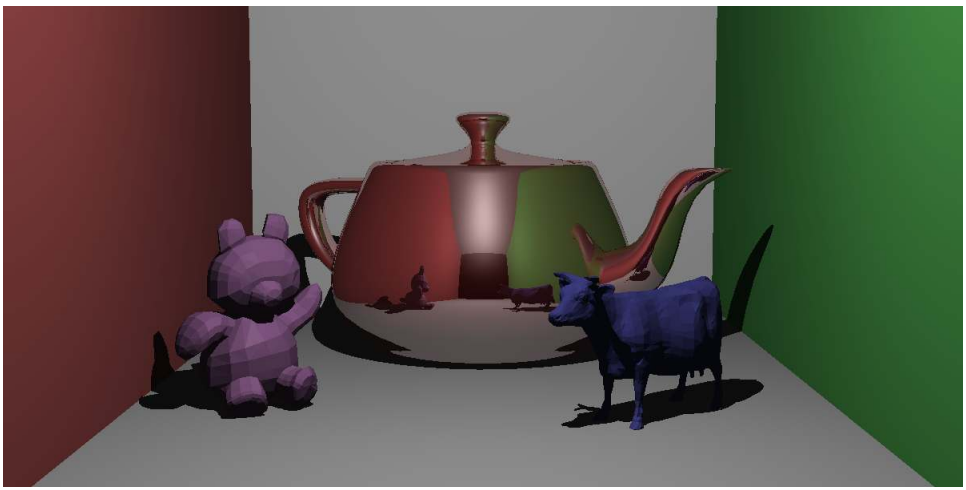
Pense um pouco. Então, depois de brincar com esse novo recurso o suficiente, continue lendo. Você adicionará seu primitivo final no próximo capítulo: o triângulo.

CAPÍTULO 15

# Triângulos

O primitivo final em seu ray tracer pode parecer uma escolha estranha: o *triângulo.*
Por si só, sua utilidade talvez seja questionável, mas onde realmente brilha é quando você usa centenas ou milhares deles juntos para construir uma superfície.

Aqui está um exemplo de uma cena composta por mais de *quinze mil* triângulos:



No ursinho de pelúcia roxo, é possível ver claramente as facetas e planos que traem a composição triangular do modelo. Mesmo a vaca, se você levar uma lente de aumento para a imagem, mostraria facetas semelhantes (embora mais finas). Mas aquele bule, agora! É realmente composto de triângulos também?

Sim, é isso. E neste capítulo, você não apenas adicionará suporte para modelos poligonais como o ursinho de pelúcia e a vaca, mas também a interpolação normal para fazer modelos como o bule parecerem perfeitamente lisos.

Vamos pular nisso!

## Triângulos

Embora seja certamente possível implementar uma primitiva de triângulo na origem, com dimensões unitárias, e depois transformá-la no lugar como você fez com todas as outras primitivas que você implementou, isso torna esses triângulos realmente difíceis de usar bem . Portanto, sua primitiva de triângulo realmente aceitará três parâmetros, que descrevem a localização de cada um de seus cantos no espaço do objeto. Você ainda pode transformar o triângulo também, se necessário.

Sua implementação de triângulos seguirá estas etapas:

1. Crie a própria forma de triângulo, pré-calculando vários valores para otimizar os cálculos de interseção.

2 Implementar o local_normal_at () função para calcular o vetor normal para triângulos.

3 Implemente o algoritmo de interseção de triângulo de raio Möller – Trumbore. Isso exigirá vários testes.

Preparar, apontar, vá!

### Teste nº 1: Criando um Triângulo

*Um triângulo é uma forma composta por três pontas. O construtor deve pré-calcular dois vetores de aresta e a normal do triângulo.*

Dados três pontos, instancie um triângulo. Em seguida, mostre que cada ponto é inicializado e que dois vetores de borda e o vetor normal são pré-computados.

**features / triangles.feature**

```
Cenário : Construindo um triângulo
    Dado p1 ← ponto (0, 1, 0)
        E p2 ← ponto (-1, 0, 0)
        E p3 ← ponto (1, 0, 0)
        E t ← triângulo (p1, p2, p3)
    Então t.p1 = p1
        E t.p2 = p2
        E t.p3 = p3
        E t.e1 = vetor (-1, -1, 0)
        E t.e2 = vetor (1, -1, 0)
        E t.normal = vetor (0, 0, -1)
```

Seu ray tracer acabará por usar esses dois vetores de borda, e1 e e2, para determinar se e onde o raio cruza o triângulo. Ele também vai usar isso
normal vetor como o normal em cada ponto de intersecção. Embora você certamente possa calcular esses três valores para cada acerto, eles sempre serão os mesmos em qualquer lugar do triângulo. Guarde algum trabalho para o seu ray tracer e pré-calcule-os quando a forma for construída, da seguinte forma:

$$e_1 = p_2 - p_1$$

$$e_2 = p_3 - p_1$$

$$\text{normal} = \text{normalizar}(\text{cruzar}(e_2, e_1))$$

Com o normal vetor pré-computado, o próximo teste quase escreve a si mesmo.

### Teste # 2: Vetor normal para um triângulo

*A normal pré-calculada do triângulo é usada para cada ponto do triângulo.*

Assim que tiver o seu triângulo() função pré-computando o vetor normal, o local_normal_at (triângulo, ponto) a função deve simplesmente retornar aquele vetor para cada ponto dado.

**features / triangles.feature**

```
Cenário : Encontrando o normal em um triângulo
    Dado t ← triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
    Quando n1 ← local_normal_at (t, ponto (0, 0,5, 0))
        E n2 ← local_normal_at (t, ponto (-0,5, 0,75, 0))
        E n3 ← local_normal_at (t, ponto (0,5, 0,25, 0))
    Então n1 = t.normal
        E n2 = t.normal
        E n3 = t.normal
```

Vá em frente e faça isso passar, implementando local_normal_at () para triângulos e retorne o vetor normal pré-calculado.

Os próximos cinco testes lidarão com o algoritmo de interseção.

### Testes # 3 a 7: Cruzando um Raio com um Triângulo

*Um raio que perde um triângulo não deve adicionar nenhuma interseção à lista de interseções. Um raio que atinge um triângulo deve adicionar exatamente uma interseção à lista.*

Esses cinco testes apresentam o comportamento do algoritmo de interseção do triângulo de raios. O algoritmo específico que você implementará é o *Möller – Trumbore* algoritmo,[1] que é rápido, curto e tem o efeito colateral útil de pré-calcular alguns valores que você usará posteriormente neste capítulo para implementar triângulos suaves. Você construirá a implementação desse algoritmo em partes, com cada teste exercendo um pouco mais disso.

Para este primeiro teste, comece criando um triângulo. Em seguida, posicione um raio de modo que seja lançado paralelo à superfície do triângulo. O raio deve perder o triângulo.

**features / triangles.feature**

```
Cenário : Cruzando um raio paralelo ao triângulo
    Dado t ← triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
```

1 www.tandfonline.com/doi/abs/10.1080/10867651.1997.10487468

```
    E r ← raio ( ponto (0, -1, -2), vetor (0, 1, 0))
  Quando xs ← local_intersect (t, r)
  Então xs está vazio
```

Faça esse teste passar cruzando a direção do raio com e2, e pontuando o resultado com e1 para produzir o determinante. Se o resultado for próximo de zero, o raio é paralelo ao triângulo e erra. Aqui estão alguns pseudocódigos para a primeira parte do algoritmo, tratando desse caso específico.

```
função local_intersect (triângulo, raio)
  dir_cross_e2 ← cruzar (raio.direcção, triângulo.e2) det ← ponto (triângulo.e1, dir_cross_e2)

  Retorna () E se abdômen (det) <EPSILON

  # uma interseção falsa para garantir que o resultado não seja um falso positivo
  Retorna ( interseção (1, triângulo))
função final
```

Observe a interseção falsa sendo retornada no final; isso serve apenas para evitar falsos positivos durante o teste. Sem isso, se você tiver um erro em sua função e ele falhar em reconhecer que o raio falhou, ele iria (neste ponto) *ainda* retornar sem adicionar uma interseção, o que o teste consideraria para significar que a função está funcionando corretamente. Adicionar a interseção falsa garante que o teste falhará se sua implementação estiver errada. Você removerá essa linha falsa logo, depois de implementar todo o algoritmo.

Os próximos três testes configuram o mesmo triângulo e, em seguida, configuram um raio de forma que ele perca o triângulo sobre uma de suas bordas. Para o primeiro teste, o raio passa além do p1-p3 Beira.

**features / triangles.feature**

```
Cenário : Um raio perde a borda p1-p3
  Dado t ← triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
    E r ← raio ( ponto (1, 1, -2), vetor (0, 0, 1))
  Quando xs ← local_intersect (t, r)
  Então xs está vazio
```

Para fazer essa passagem, adicione os seguintes cálculos antes da interseção falsa em seu local_intersect () função.

```
f ← 1.0 / det

p1_to_origin ← ray.origin - triangulo.p1 u ← f * ponto (p1_to_origin,
dir_cross_e2)
Retorna () E se u <0 ou u> 1
```

Se isso você o valor não está entre 0 e 1, inclusive, o raio falha.

Os próximos dois testes configuram o raio para passar além do p1-p2 e p2-p3 bordas do triângulo.

**features / triangles.feature**

```
Cenário :       Um raio perde a borda p1-p2
    Dado t      ←   triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
        E r     ←   raio ( ponto (-1, 1, -2), vetor (0, 0, 1))
    Quando xs   ←   local_intersect (t, r)
    Então xs    está vazia

Cenário :       Um raio perde a borda p2-p3
    Dado t      ←   triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
        E r     ←   raio ( ponto (0, -1, -2), vetor (0, 0, 1))
    Quando xs   ←   local_intersect (t, r)
    Então xs    está vazia
```

Você pode fazer isso passar implementando os cálculos a seguir, novamente colocando-os um pouco antes da interseção falsa no final de sua função.

```
origin_cross_e1 ← cruzar (p1_to_origin, triângulo.e1) v ← f * ponto (ray.direction, origin_cross_e1)


Retorna () E se v <0 ou (u + v)> 1
```

Finalmente, você precisa lidar com o caso em que o raio realmente atinge o triângulo. Este último teste cria o triângulo novamente, mas organiza o raio de forma que ele o intercepte e confirma que existe uma interseção na distância correta.

**features / triangles.feature**

```
Cenário : Um raio atinge um triângulo
    Dado t ← triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
        E r ← raio ( ponto (0, 0,5, -2), vetor (0, 0, 1))
    Quando xs ← local_intersect (t, r)
    Então xs.count = 1
        E xs [0] .t = 2
```

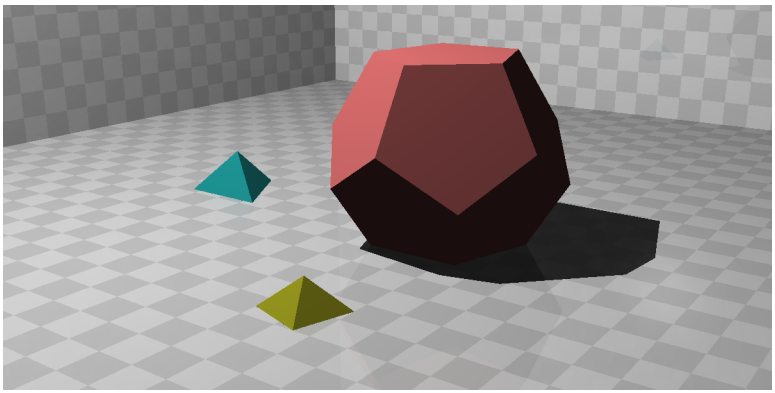
Substitua a interseção falsa no final do triângulo local_intersect () função com a seguinte lógica, para produzir o *real* interseção.

```
t = f * ponto (triângulo.e2, origin_cross_e1)
Retorna ( interseção (t, triângulo))
```

Uma vez que todos os testes estão passando, você deve ser capaz de renderizar alguns triângulos. Sinta-se à vontade para renderizá-los individualmente, se desejar, mas eles se tornam muito mais interessantes em grupos. Pense em como você construiria uma pirâmide de três ou quatro lados a partir de triângulos. Que tal um octaedro? Ou se você quiser ser realmente ambicioso, considere poliedros mais complexos, como o dodecaedro no imagem na página 212 .

.....

Em última análise, porém, organizar triângulos manualmente é difícil de fazer bem. É muito mais fácil usar uma ferramenta de modelagem 3D para construir uma forma e exportá-la para um arquivo. A próxima seção irá guiá-lo através do processo de implementação de um



analisador para um dos formatos de arquivo de modelo 3D mais comuns, o que permitirá que você importe modelos mais complexos para suas cenas.

## Arquivos Wavefront OBJ

O formato de arquivo Wavefront OBJ é um formato comum para armazenar e compartilhar dados gráficos 3D. Como o formato de imagem PPM que você implementou em Capítulo 2, Desenhando em uma tela, na página 15 , o formato OBJ é texto simples, o que significa que você pode visualizar, editar e até mesmo criar esses arquivos em qualquer editor de texto, embora seja muito mais fácil modelar algo em uma ferramenta de modelagem 3D e, em seguida, exportá-lo para o OBJ.

O formato OBJ consiste em declarações, cada uma ocupando uma única linha. Cada instrução é precedida de um comando, seguido por uma lista de argumentos delimitada por espaço. Por exemplo, o seguinte arquivo OBJ define três vértices ( `v`), e um triângulo ( `f`, para “face”) que faz referência a esses vértices.

```
v 1.5 2 1.3
v 1,4 -1,2 0,12
v -0,1 0 -1,3

f 1 2 3
```

Existem alguns outros tipos de instruções também, mas você só precisa reconhecer alguns deles em seu ray tracer. Você implementará esse analisador em seis etapas:

1. Comece com um analisador que ignora silenciosamente todas as instruções não reconhecidas.
2. Adicione suporte para vértices ao analisador.
3. Adicione suporte para triângulos.
4. Implemente a triangulação de polígonos convexos, para que seu analisador possa importá-los também.

5. Adicione suporte para grupos de polígonos em um modelo.
6. Exporte todo o modelo como um Grupo instância, para que você possa adicioná-lo a um cena a ser renderizada.

Deixe-me reiterar que você usará Grupo instâncias para representar esses grupos de triângulos. Embora essa técnica seja simples de explicar, infelizmente é *não* a maneira mais ideal de representar este tipo de dados. Se estiver interessado em otimizar o seu ray tracer, você pode investigar uma estrutura chamada *malha triangular,* que pode ser armazenado e processado com um pouco mais de eficiência.

Por enquanto, porém, grupos de triângulos servirão. Vamos começar!

### Teste # 8: Analisador OBJ com entrada de jargão

*O analisador deve ignorar silenciosamente todas as instruções não reconhecidas.*

Já que seu analisador só vai lidar com um subconjunto do formato OBJ, você precisa ter certeza de que ele não engasga quando recebe um modelo que contém instruções que você ainda não implementou. O teste a seguir apresenta uma função chamada parse_obj_file (arquivo), que retorna uma estrutura de dados encapsulando o conteúdo do arquivo (aparentemente formatado para OBJ).

```
features / obj_file.feature
Cenário : Ignorando linhas não reconhecidas
    Dado jargão ← um arquivo contendo:
        "" "
        Havia uma jovem chamada Bright que viajava muito mais
        rápido que a luz. Ela partiu um dia

        de uma forma relativa,
        e voltou na noite anterior. "" "

    Quando analisador ← parse_obj_file (tagarelice)
    Então o analisador deveria ter ignorado 5 linhas
```

Nesse caso, ele analisa um arquivo que contém algo sem sentido e o analisador resultante registra quantas linhas foram ignoradas.

### Teste # 9: Arquivo OBJ com dados de vértice

*O analisador deve processar os dados do vértice da entrada fornecida.*

Aqui, o analisador recebe um arquivo contendo quatro instruções de vértice. Cada instrução de vértice começa com um “v”, seguido por um caractere de espaço e, em seguida, três números inteiros ou de ponto flutuante delimitados por espaços.

```
features / obj_file.feature
Cenário : Registros de vértices
    Dado Arquivo ← um arquivo contendo:
```

```
    "" "
    v -1 1 0
    v -1,0000 0,5000 0,0000 v 1 0 0

    v 1 1 0
    "" "
  Quando analisador ← parse_obj_file (Arquivo)
  Então parser.vertices [1] = ponto (-1, 1, 0)
    E parser.vertices [2] = ponto (-1, 0,5, 0)
    E parser.vertices [3] = ponto (1, 0, 0)
    E parser.vertices [4] = ponto (1, 1, 0)
```

O analisador resultante deve ter uma matriz de vértices, cada um registrado como um ponto. Nota: é significativo que a matriz seja baseada em 1 e não baseada em 0! Quando você chegar ao próximo teste, verá que as faces (triângulos e polígonos) referem-se a esses vértices por seu índice, começando com 1.

### Teste # 10: Arquivo OBJ com dados do triângulo

*O analisador deve processar os dados do triângulo da entrada fornecida.*

O analisador agora recebe um arquivo contendo quatro declarações de vértices e dois triângulos. Os triângulos são introduzidos com o f comando (para “face”), seguido por três inteiros referindo-se aos vértices correspondentes. Observe que esses índices são baseados em 1 e *não* Baseado em 0! Ou seja, o vértice número 1 é o primeiro vértice encontrado no arquivo, não o segundo.

```
features / obj_file.feature
Cenário : Analisando faces de triângulos
  Dado Arquivo ← um arquivo contendo:
    "" "
    v -1 1 0
    v -1 0 0
    v 1 0 0
    v 1 1 0

    f 1 2 3
    f 1 3 4
    "" "
  Quando analisador ← parse_obj_file (Arquivo)
    E g ← parser.default_group
    E t1 ← primeiro filho de g
    E t2 ← segundo filho de g
  Então t1.p1 = parser.vertices [1]
    E t1.p2 = parser.vertices [2]
    E t1.p3 = parser.vertices [3]
    E t2.p1 = parser.vertices [1]
    E t2.p2 = parser.vertices [3]
    E t2.p3 = parser.vertices [4]
```

Observe também as referências de teste a default_group propriedade no analisador. Esta Grupo instância recebe toda a geometria gerada. Seu analisador deve adicionar os dois triângulos a este grupo.

### Teste nº 11: Arquivo OBJ com dados de polígono

*O analisador deve processar e triangular dados poligonais da entrada fornecida.*

Empurrando um pouco o envelope agora, você dará ao seu analisador um arquivo contendo cinco declarações de vértices e uma única face pentagonal consumindo todas elas. Seu ray tracer só sabe renderizar triângulos, então ele precisa ser capaz de quebrar o polígono em triângulos.

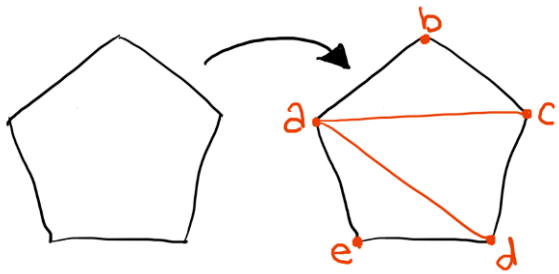
```
features / obj_file.feature
Cenário : Polígonos triangulando
    Dado Arquivo ← um arquivo contendo:
        "" "
        v -1 1 0
        v -1 0 0
        v 1 0 0
        v 1 1 0
        v 0 2 0

        f 1 2 3 4 5
        "" "
    Quando analisador ← parse_obj_file (Arquivo)
        E g ← parser.default_group
        E t1 ← primeiro filho de g
        E t2 ← segundo filho de g
        E t3 ← terceiro filho de g
    Então t1.p1 = parser.vertices [1]
        E t1.p2 = parser.vertices [2]
        E t1.p3 = parser.vertices [3]
        E t2.p1 = parser.vertices [1]
        E t2.p2 = parser.vertices [3]
        E t2.p3 = parser.vertices [4]
        E t3.p1 = parser.vertices [1]
        E t3.p2 = parser.vertices [4]
        E t3.p3 = parser.vertices [5]
```

Isso aparecerá com bastante frequência em arquivos OBJ, sejam encontrados on-line ou exportados por você mesmo. Freqüentemente, é mais eficiente (no que diz respeito ao espaço) descrever um polígono plano do que descrever o mesmo polígono como uma série de triângulos. Isso é bom, mas significa que você precisa explicitamente *triangular* - converta em triângulos - os polígonos antes de renderizá-los.

Para manter as coisas simples, apenas suponha que os dados de entrada sempre descrevem polígonos convexos - aqueles cujos ângulos internos são todos menores ou iguais a 180 °. Quando

este é o caso, você pode quebrá-los em triângulos usando um *triangulação do ventilador.*

Visualmente, o processo se parece com isto:




A ideia é que você escolha um vértice inicial, uma, e, em seguida, crie um triângulo combinando-o com os próximos dois vértices da lista, b e c. Então, começando com uma novamente, crie outro triângulo com c e d. Continue desta forma, começando cada triângulo com o vértice uma, adicionando o último vértice do triângulo anterior e o próximo vértice da lista, e procedendo até que todos os vértices tenham sido usados.

Em pseudocódigo, é assim:

```
# vértices é uma matriz baseada em 1 de pelo menos três vértices
função fan_triangulation (vértices)
    triângulos ← lista vazia

    pra índice ← 2 a comprimento (vértices) - 1
        tri ← triângulo (vértices [ 1 ] , vértices [ índice ] , vértices [ índice + 1 ] ) adicionar tri a triângulos

    fim para

    Retorna triângulos
função final
```

Observe que o pseudocódigo aqui espera uma matriz de pontos baseada em 1, porque é isso que o formato de arquivo OBJ assume. Se seu analisador estiver traduzindo os índices baseados em OBJ 1 para baseados em 0, você pode se sentir à vontade para implementar sua triangulação de leque de acordo.

Essa é a chave para fazer este teste passar - aplique uma triangulação em leque à lista de vértices e adicione os triângulos resultantes ao grupo padrão.

### Teste nº 12: Grupos nomeados em arquivos OBJ

*O analisador deve reconhecer uma instrução de grupo e adicionar triângulos subsequentes ao grupo nomeado.*

Os modelos podem se tornar bastante complexos e podem ser compostos de peças diferentes. Em vez de um único modelo de pessoa, por exemplo, o modelo pode ser

composto por grupos como "braço", "perna" e "cabeça". Esses grupos são identificados em um arquivo OBJ com o g comando.

Este teste lê um arquivo OBJ e mostra que os grupos nomeados fornecidos estão presentes e contêm os triângulos esperados.

**features / obj_file.feature**

```
Cenário : Triângulos em grupos
    Dado Arquivo ← o arquivo " triangles.obj "
    Quando analisador ← parse_obj_file (Arquivo)
        E g1 ← " FirstGroup " do analisador
        E g2 ← " SecondGroup " do analisador
        E t1 ← primeiro filho de g1
        E t2 ← primeiro filho de g2
    Então t1.p1 = parser.vertices [1]
        E t1.p2 = parser.vertices [2]
        E t1.p3 = parser.vertices [3]
        E t2.p1 = parser.vertices [1]
        E t2.p2 = parser.vertices [3]
        E t2.p3 = parser.vertices [4]
```

Crie o seguinte triangles.obj arquivo também, para que você possa alimentá-lo para o teste:

**files / triangles.obj**

```
v  - 1 1 0
v  - 1 0 0
v  1 0 0
v  1 1 0

g  FirstGroup
f  1 2 3
g  SecondGroup
f  1 3 4
```

Para fazer o teste passar, você precisará controlar qual grupo foi referenciado mais recentemente e adicionar todos os triângulos subsequentes a esse grupo.

### Teste # 13: convertendo um modelo OBJ em um grupo

*O analisador deve converter um modelo OBJ analisado em um* **Grupo** *instância.*

Depois de analisar o arquivo OBJ, você ainda precisa adicionar o modelo à sua cena. Este teste adiciona uma função, obj_to_group (analisador), que converte um arquivo OBJ analisado em um Grupo instância que você pode adicionar à sua cena. Usa o mesmo
triangles.obj arquivo como o teste anterior, para entrada.

**features / obj_file.feature**

**Cenário :** Converter um arquivo OBJ em um grupo
  **Dado** Arquivo ← o arquivo " **triangles.obj** "
    **E** analisador ← parse_obj_file (Arquivo)
  **Quando** g ← obj_to_group (analisador)
  **Então** g inclui " **FirstGroup** " do analisador
    **E** g inclui " **SecondGroup** " do analisador

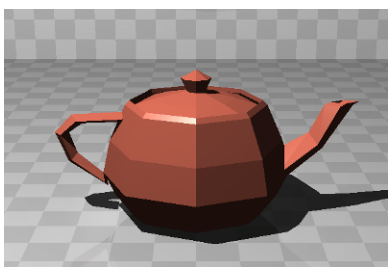
Com essa peça concluída, você deve ser capaz de pegar alguns arquivos OBJ simples, analisá-los e renderizá-los em suas cenas! Você pode encontrar muitos online, procurando coisas como "arquivo obj simples". Um desses sites é esta página mínima de um
Aula de computação gráfica do MIT: groups.csail.mit.edu/graphics/classes/6.837/F03/models .
Outro, que inclui versões de alta e baixa resolução do bule, é de uma aula de computação gráfica na Universidade de Utah: graphics.cs.utah.edu/ cursos / cs6620 / outono2013 /? prj = 5 .

Muitos modelos OBJ que você encontrará online consistem em milhares de triângulos. Não se surpreenda se o seu ray tracer atolar sob esse tipo de carga! Para acelerar as coisas, considere pesquisar algumas otimizações, como a técnica da caixa delimitadora mencionada em Capítulo 14, Grupos, na página 193 e subdividindo os triângulos em um grupo em grupos menores. Isso pode reduzir o número de triângulos que precisam ser interceptados por cada raio.

Suas renderizações estão com uma boa aparência agora, mas ainda poderiam ter uma aparência melhor. Esses modelos, conforme renderizados, são obviamente compostos de triângulos. É hora de ensinar ao seu ray tracer como suavizar essas bordas mentindo sobre os vetores normais.

## Triângulos suaves

Assumindo que tudo deu certo até agora, você tem seu ray tracer renderizando modelos poligonais complexos que você importou de arquivos OBJ! Uma desvantagem flagrantemente óbvia, porém, é que esses modelos são poligonais. O bule na figura a seguir provavelmente é típico do que você está vendo:

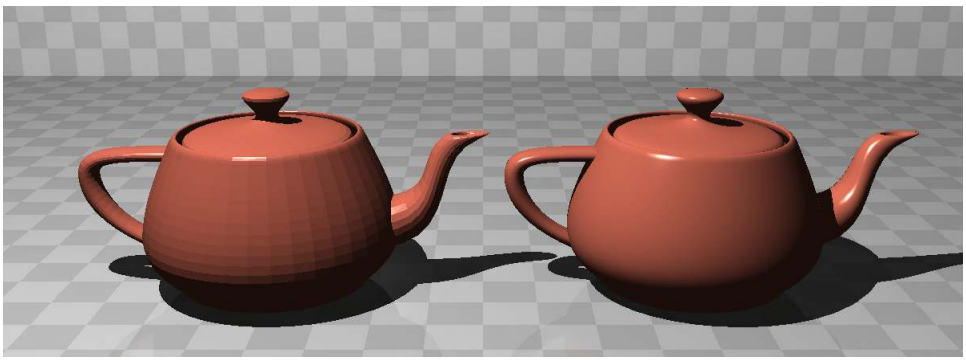


Após a emoção inicial de “Oh meu Deus! Funciona!" passa, você fica se perguntando o que você pode fazer para tornar aquele bule robusto um pouco mais brilhante.

Bem, uma coisa que você pode fazer é encontrar um modelo de resolução mais alta. O primeiro usou cerca de 240 triângulos. O bule na figura a seguir usa cerca de 6.400.

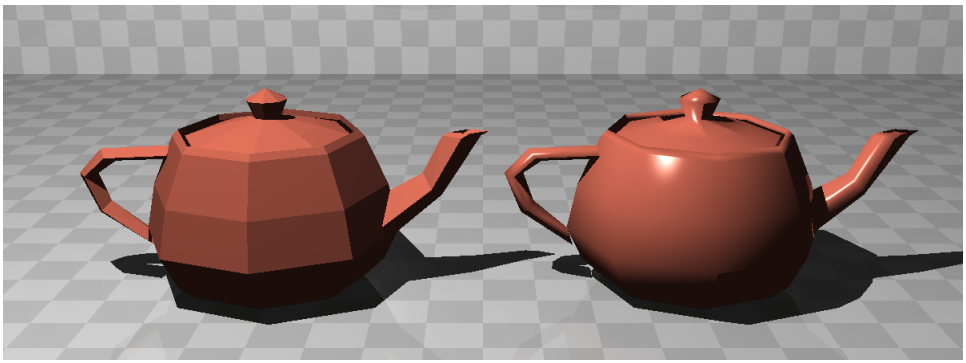
A diferença é marcante! É muito mais limpo, mas é *ainda* imperfeito. Também é preciso muito, muito mais trabalho para renderizar, graças ao uso de vinte e cinco vezes mais triângulos.

Felizmente, existe uma técnica útil chamada *interpolação normal,* que funciona atribuindo um vetor normal a cada vértice. Então, esses vértices normais são usados para interpolar o vetor normal em qualquer ponto do triângulo, basicamente *deitado* sobre o vetor normal para enganar as rotinas de sombreamento! Feito corretamente, o resultado pode imitar uma superfície perfeitamente curva. A figura a seguir mostra aquele bule de alta resolução novamente, renderizado sem interpolação normal à esquerda e com ele à direita.



Esses triângulos foram suavizados! Também funciona para modelos de resolução mais baixa. Confira o figura na página 220 , demonstrando triângulos suaves com o bule de baixa resolução.

Nesse caso, porém, você pode ver a fraqueza dessa técnica: ela não altera a geometria - apenas o vetor normal no ponto de interseção.



Assim, a silhueta da imagem permanece em blocos e angular, dando a mentira ao aspecto liso da superfície.

Para fazer isso funcionar, você fará o seguinte:

1. Adicione uma nova primitiva, chamada smooth_triangle (p1, p2, p3, n1, n2, n3).
2. Adicionar você e v propriedades para o objeto de interseção. Eles serão usados para representar uma localização na superfície de um triângulo, em relação aos seus cantos. Preencher o você e v propriedades
3. da interseção ao interceptar um triângulo.
4. Aceite um objeto de interseção como um parâmetro para ambos normal_at () e local_normal_at (), e implementar o cálculo normal para triângulos suaves, com interpolação normal.
5. Passe o acertar interseção ao chamar normal_at () e local_normal_at ().

Parece muito, mas tudo se resolverá rapidamente.

Cada um dos smooth_triangle () testes pressupõe que o triângulo a ser testado, tri, é preparado pela seguinte configuração:

**features / smooth-triangles.feature**

```
Fundo :
    Dado p1      ←  ponto (0, 1, 0)
        E p2     ←  ponto (-1, 0, 0)
        E p3     ←  ponto (1, 0, 0)
        E n1     ←  vetor (0, 1, 0)
        E n2     ←  vetor (-1, 0, 0)
        E n3     ←  vetor (1, 0, 0)
    Quando tri   ←  smooth_triangle (p1, p2, p3, n1, n2, n3)
```

Assim que estiver pronto, comece certificando-se de que ele construiu o triângulo corretamente.

### Teste # 14: Criando um Triângulo Suave

*Um triângulo liso deve armazenar os três pontos de vértice do triângulo, bem como o vetor normal em cada um desses pontos.*

Supondo que o fundo já tenha configurado o triângulo liso tri, este teste apenas afirma que cada uma das propriedades foi configurada corretamente.

```
features / smooth-triangles.feature
Cenário : Construindo um triângulo liso
    Então tri.p1 = p1
        E tri.p2 = p2
        E tri.p3 = p3
        E tri.n1 = n1
        E tri.n2 = n2
        E tri.n3 = n3
```

Em seguida, você aprimorará seu interseção() estrutura.

### Teste nº 15: Adicionando propriedades uev às interseções

*Um registro de interseção pode ter* **você** *e* **v** *propriedades, para ajudar a identificar onde em um triângulo ocorreu a interseção, em relação aos cantos do triângulo.*

Esses você e v As propriedades serão números de ponto flutuante entre 0 e 1. Elas são específicas para triângulos, portanto, as interseções com qualquer outra forma não as usarão. Ainda assim, para triângulos - e especialmente para *suave* triângulos - eles são relevantes. O teste a seguir demonstra como construir um registro de interseção que encapsula o você e v propriedades, usando um novo intersection_with_uv (t, forma, u, v)

função.

```
features / intersections.feature
Cenário : Uma interseção pode encapsular `u` e` v`
    Dado s ← triângulo ( ponto (0, 1, 0), ponto (-1, 0, 0), ponto (1, 0, 0))
    Quando eu ← intersection_with_uv (3,5, s, 0,2, 0,4)
    Então iu = 0,2
        E iv = 0,4
```

É seguro deixar o você e v propriedades indefinidas quando as interseções são construídas de qualquer outra maneira.

### Teste nº 16: Preencher uev nas interseções do triângulo

*Ao cruzar triângulos, preserve o* **você** *e* **v** *valores na interseção resultante.*

De volta aos triângulos suaves, este teste mostra o que acontece quando você cruza um com um raio. A interseção resultante deve ter o você e v conjunto de propriedades.

**features / smooth-triangles.feature**

```
Cenário : Uma interseção com um triângulo liso armazena u / v
    Quando r ← raio ( ponto (-0,2, 0,3, -2), vetor (0, 0, 1))
        E xs ← local_intersect (tri, r)
    Então xs [0] .u = 0,45
        E xs [0] .v = 0,25
```

Isso é realmente ótimo, porque você já calculou ambos você e v!

Lembra quando eu disse que o algoritmo Möller – Trumbore tinha um recurso que seria útil mais tarde? Bem, agora é mais tarde. Essa rotina de intersecção de triângulo definiu duas variáveis, você e v. Pegue essas duas variáveis e passe-as para o novo intersection_with_uv () função, no lugar da chamada existente para interseção().

Para encontrar a interseção com um raio e um triângulo suave, use a rotina de interseção de triângulo. É realmente o mesmo cálculo, mas com a adição de armazenamento você e v no cruzamento. Se isso requer que alguma refatoração aconteça em seu código, certifique-se de cuidar disso agora também.

Assim que tiver aqueles você e v propriedades sendo armazenadas na interseção, continue a ler. Você está prestes a colocá-los em uso.

## Teste # 17: Interpolação normal

*Ao calcular o vetor normal em um triângulo suave, use a interseção* **você** *e* **v** *propriedades para interpolar o normal.*

Este teste configura uma interseção com você e v e então passa esse cruzamento para normal_at (). O ponto é intencionalmente definido para a origem para reforçar o fato de que não é usado aqui - apenas você e v deve ter qualquer efeito.

**features / smooth-triangles.feature**

```
Cenário : Um triângulo suave usa u / v para interpolar o normal
    Quando eu ← intersection_with_uv (1, tri, 0,45, 0,25)
        E n ← normal_at (tri, ponto (0, 0, 0), i)
    Então n = vetor (-0,5547, 0,83205, 0)
```

Faça essa passagem adicionando o objeto de interseção que representa o hit como um parâmetro para ambos normal_at () e local_normal_at (). Para preservar uma API consistente, adicione este parâmetro ao local_normal_at () função para cada forma, embora só seja realmente usada para os triângulos lisos.

Depois de ter esse parâmetro passado para o triângulo liso local_normal_at ()
função, você interpola o normal combinando os vetores normais dos vértices do triângulo de acordo com o você e v propriedades, conforme fornecido no seguinte pseudocódigo:

```
função local_normal_at (tri, apontar, acertar)
    Retorna tri.n2 * hit.u +
```

```
            tri.n3 * hit.v +
            tri.n1 * (1 - hit.u - hit.v)
função final
```

Depois que isso for aprovado, a última parte para fazer esse trabalho é garantir que o acerto seja passado para o cálculo normal.

## Teste nº 18: Passe o hit para a função normal_at

*O* **prepare_computations ()** *função deve passar o próprio hit para a chamada para* **normal_at ().**

Construir uma interseção com tri, e alguns você e v valores. Quando prepare_computations () é chamado nessa interseção, o normal deve ser calculado de acordo com as regras para o triângulo liso, que requer que a interseção seja passada para normal_at ().

```
features / smooth-triangles.feature
Cenário : Preparando o normal em um triângulo liso
    Quando eu ← intersection_with_uv (1, tri, 0,45, 0,25)
        E r ← raio ( ponto (-0,2, 0,3, -2), vetor (0, 0, 1))
        E xs ← cruzamentos (eu)
        E compor ← prepare_computations (i, r, xs)
    Então comps.normalv = vetor (-0,5547, 0,83205, 0)
```

Depois de passar, é hora de revisitar seu analisador OBJ e conectar esses triângulos suaves.

## Triângulos suaves em arquivos OBJ

Seu analisador OBJ já está bem perto de suportar triângulos suaves. Tudo o que precisa agora é apoiar o *vértice normal (* vn) comando e para atualizar a maneira como ele analisa o f ( “Face”) comando.

### Teste nº 19: Arquivo OBJ com dados normais de vértice

*Os dados normais do vértice devem ser importados corretamente de um arquivo OBJ.*

Este teste configura um arquivo OBJ que contém quatro declarações normais de vértice (“vn”) e, a seguir, mostra que cada uma delas é importada como um vetor. Observe que o normais coleção é baseada em 1, assim como o vértices coleção era.

```
features / obj_file.feature
Cenário : Registros normais de vértice
    Dado Arquivo ← um arquivo contendo:
        "" "
        vn 0 0 1
        vn 0,707 0 -0,707 vn 1 2 3

        "" "
```

```
Quando analisador ← parse_obj_file (Arquivo)
Então parser.normals [1] = vetor (0, 0, 1)
    E parser.normals [2] = vetor (0,707, 0, -0,707)
    E parser.normals [3] = vetor (1, 2, 3)
```

Os normais são importados como estão, sem normalização ou outro processamento feito. Depois de importados, é apenas uma questão de associar cada um dos normais de vértice a um vértice, o que você fará a seguir.

## Teste # 20: Faces com vetores normais

*Os dados normais do vértice devem ser associados corretamente aos dados faciais de um arquivo OBJ.*

O f comando que você implementou anteriormente está apenas pela metade, na verdade. O teste a seguir demonstra uma versão mais completa da sintaxe, permitindo que os vértices de uma face sejam associados a vetores normais.

```
features / obj_file.feature
Cenário : Rostos com normais
    Dado Arquivo ← um arquivo contendo:
        """ "
        v 0 1 0
        v -1 0 0
        v 1 0 0

        vn -1 0 0
        vn 1 0 0
        vn 0 1 0

        f 1 // 3 2 // 1 3 // 2
        f 1/0/3 2/102/1 3/14/2 """ "

    Quando analisador ← parse_obj_file (Arquivo)
        E g ← parser.default_group
        E t1 ← primeiro filho de g
        E t2 ← segundo filho de g
    Então t1.p1 = parser.vertices [1]
        E t1.p2 = parser.vertices [2]
        E t1.p3 = parser.vertices [3]
        E t1.n1 = parser.normals [3]
        E t1.n2 = parser.normals [1]
        E t1.n3 = parser.normals [2]
        E t2 = t1
```

Acontece que o f comando suporta as seguintes variações, a primeira das quais você já implementou:

f 1 2 3

f 1/2/3 2/3/4 3/4/5 f 1 // 3 2 // 4 3 // 5

A barra é usada para delimitar até três tipos diferentes de informações por vértice. O primeiro número em cada triplo é o próprio índice do vértice. O segundo é opcional *vértice de textura (* que você não implementará para este recurso e pode ser ignorado). O terceiro é um índice opcional na lista de normais de vértice, correspondendo ao vn comando que você acabou de implementar.

Para fazer este teste passar, seu f o comando precisa verificar se as normais dos vértices estão presentes para os vértices e, se estiverem, o comando deve chamar smooth_triangle () ao invés de triângulo().

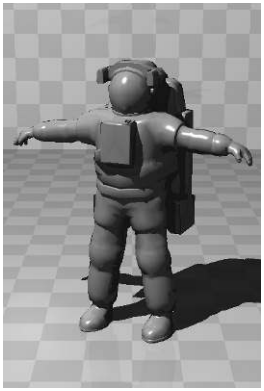
Faça esse teste passar. Quando tudo estiver bem, você está pronto para assinar este capítulo!

## Juntar as peças

Você pode encontrar modelos 3D para renderizar em vários lugares online, pesquisando por “modelos 3D gratuitos”. Aqui estão alguns dos primeiros resultados que encontrei com essa pesquisa:

- TurboSquid[2] ( pago, mas tem uma seção de modelos gratuitos)
- Free3D[3]
- cgtrader[4] ( pago, mas tem uma seção gratuita)
- ClaraIO[5]

A NASA tem uma biblioteca de recursos 3D gratuitos, incluindo modelos, em nasa3d.arc.nasa.gov . Muitos deles são bastante grandes (centenas de milhares de triângulos), mas consegui encontrar vários modelos com apenas alguns milhares, incluindo este adorável rapaz:



2 www.turbosquid.com/Search/3D-Models/free

3 free3d.com

4 www.cgtrader.com/free-3d-models

5 clara.io/library

No entanto, algumas advertências se aplicam a qualquer modelo que você encontrar online:

- Muitos desses modelos estão em formatos diferentes de OBJ. As ferramentas de conversão online variam (basta pesquisar “converter formatos 3D” ou algo semelhante), mas não posso garantir nenhuma delas. É melhor, quando possível, localizar arquivos OBJ diretamente, em vez de depender da conversão. (Ainda assim, converti o modelo anterior do astronauta de um formato 3DS para OBJ usando o conversor em

  www.greentoken.de/onlineconv/ .)

- Esses modelos são *não* de tamanho uniforme e raramente são convenientemente centrados na origem. Meu conselho é que seu analisador OBJ imprima as extensões mínimas e máximas de cada modelo importado, que você pode usar para traduzir e dimensionar o modelo em sua cena.

- Esses modelos costumam ter o $y$ e $z$ eixos trocados do que este livro apresenta, com $z$ sendo o eixo "para cima" em vez de $y$. Se você achar que este é o

  caso, uma rápida rotação ao redor $x$ de $-\pi/2$ deve fazer o truque.

Por último, você pode se divertir muito se puder colocar as mãos em um programa de modelagem 3D. Liquidificador[6] é uma opção gratuita de plataforma cruzada que é incrivelmente poderosa e pode exportar modelos no formato OBJ. O Blender tem uma curva de aprendizado correspondentemente desafiadora, mas se você quiser, pode ser maravilhoso jogar com o Blender. Você pode usá-lo para criar seus próprios modelos ou apenas converter modelos existentes em OBJ. Você pode até (com o modificador “dizimar” do Blender) simplificar os modelos existentes para que eles usem menos triângulos.

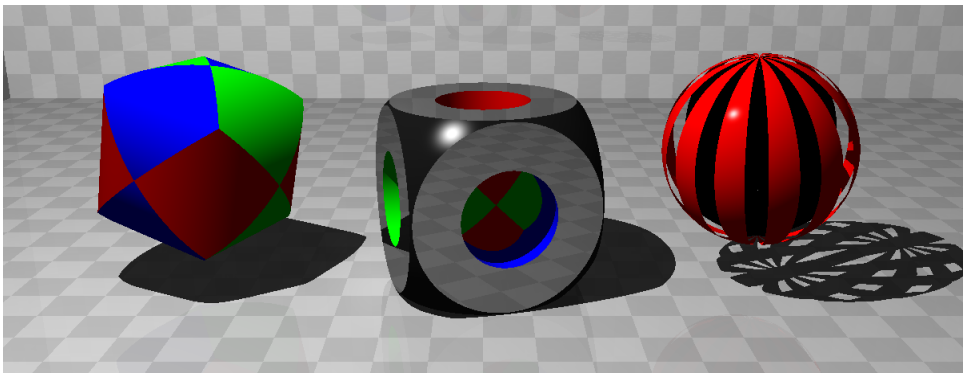
Quando terminar de se divertir com os modelos poligonais, continue lendo. O último recurso o aguarda: geometria sólida construtiva.

6 blender.org

CAPÍTULO 16

# Geometria Sólida Construtiva (CSG)

Você passou por transformações de matriz, iluminação Phong, reflexão, refração e interseções de objetos de raio para uma variedade de formas primitivas. Para este último recurso, você implementará *geometria sólida construtiva,* ou CSG - um método de combinar essas primitivas por meio de operações de conjunto. Isso permite que você crie formas muito mais complexas, como estas:



É verdade que qualquer uma dessas formas poderia ter sido montada em um aplicativo de modelagem 3D e, em seguida, exportada para um arquivo OBJ, que você poderia ter importado e renderizado. Usar CSG é melhor neste caso por duas razões importantes:

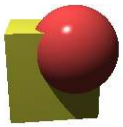
1. Para fazer com que a coloração por triângulo funcione, você precisa implementar um analisador para arquivos de material Wavefront MTL e fazer com que seu analisador OBJ implemente texturas de vértice para mapear as definições de material para vértices. (Ufa!) Com CSG, você pode aplicar estrategicamente texturas e cores a diferentes superfícies dentro do modelo, usando apenas o que você já implementou.

2 Você precisaria de centenas ou mesmo milhares de triângulos para renderizar essas formas, enquanto o CSG permite usar muito menos primitivas. O tricilindro à esquerda da imagem anterior exigia apenas três cilindros; o cubo entalhado no meio tem apenas três cilindros, um cubo e uma esfera; e a

esfera oca à direita é uma esfera e doze cubos. Adicione o cubo que representa a sala, e toda a cena consiste em apenas 22 formas!

CSG funciona pegando duas ou mais formas primitivas e combinando-as usando qualquer uma das três operações de conjunto diferentes: união, interseção e diferença.

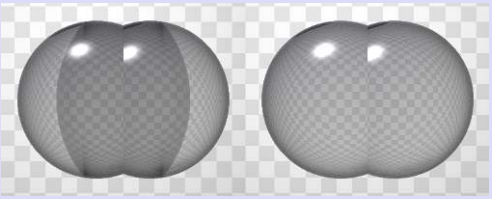
*União* combina as entradas em uma única forma, preservando todas as superfícies externas. Aqui está um cubo e uma esfera, que foram combinados por meio de uma união.



**Joe pergunta:**

**Por que usar um sindicato CSG em vez de um grupo?**

Boa pergunta! Em muitos casos, um grupo é definitivamente mais simples, mas quando você está lidando com objetos transparentes e reflexivos, como vidro, suas superfícies internas podem contribuir com reflexos indesejados. Aqui está um exemplo com duas esferas de vidro sobrepostas. À esquerda, eles foram combinados em um grupo. À direita, um sindicato CSG foi usado.



A imagem não mente; usar uma união em vez de um grupo elimina essas superfícies internas e dá a você uma transparência mais verdadeira.

*Interseção* preserva a parte das entradas que compartilham um volume (onde as formas se cruzam), resultando em uma única forma com essas superfícies combinadas. Esta é a interseção entre o cubo e a esfera da imagem anterior.

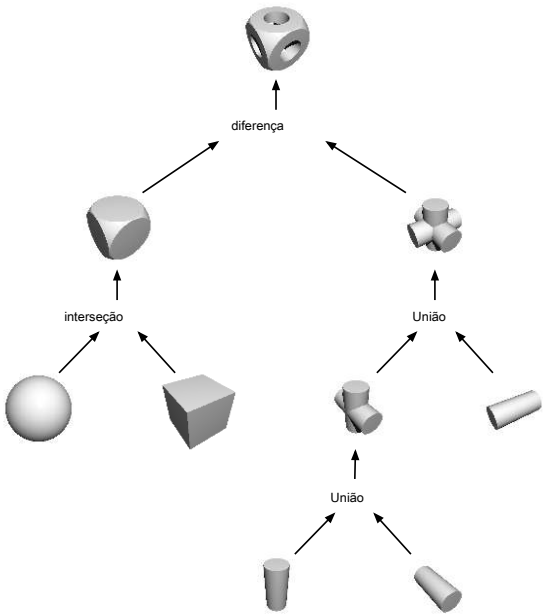
*Diferença* preserva apenas a parte da primeira forma onde não é sobreposta pelas outras. Aqui está a diferença entre o cubo original e a esfera, efetivamente esculpindo a esfera do cubo.

Sua implementação do CSG oferecerá suporte a todas essas três operações e você aprenderá como usá-las juntas para gerar uma enorme variedade de diferentes formas e efeitos.

Você está pronto? Aqui vamos nós.

## Implementando CSG

Para simplificar, sua implementação tratará todas as operações CSG como estritamente binárias, o que significa que cada uma assume exatamente duas formas como entrada. Isso pode parecer restritivo, mas como um objeto CSG é uma forma, você pode construir operações CSG arbitrariamente complexas combinando-as em uma hierarquia, como esta:




Você renderizará essas formas cruzando-as com um raio, como fez com todas as outras formas que implementou. A intersecção de um raio com uma forma CSG começa exatamente como a intersecção de um com um Grupo: primeiro você cruza o raio com os filhos da forma. Em seguida, você itera os registros de interseção resultantes, rastreando quais estão dentro de quais filhos e filtrando aqueles que não estão em conformidade com a operação atual. A lista de interseções resultante é então retornada.

O diabo, como sempre, está nos detalhes. Percorreremos esse processo um pouco de cada vez, seguindo estas etapas:

1. Crie uma forma CSG, fornecendo uma operação e duas formas de operando, deixou e direita.
2. Implementar as regras para união, interseção e diferença.
3. Filtre uma lista de cruzamentos usando as regras da etapa 2.
4. Demonstre o que acontece quando um raio perde um objeto CSG.

5. Demonstre o que acontece quando um raio atinge um objeto CSG. Comece criando a

própria forma CSG.

### Teste nº 1: Criando uma forma CSG

*Uma forma CSG é composta de uma operação e duas formas de operando.*

Instancie uma nova forma CSG. Para fins de teste, use a operação de união e atribua a ela uma esfera e um cubo para os dois operandos.

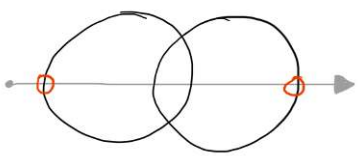
```
features / csg.feature
Cenário : CSG é criado com uma operação e duas formas
    Dado s1 ← esfera ()
        E s2 ← cubo ()
    Quando c ← csg ( " União" , s1, s2)
    Então c.operation = " União"
        E c.esquerdo = s1
        E c.right = s2
        E s1.parent = c
        E s2.parent = c
```

Os operandos são referidos como deixou e direita, espelhando a estrutura de uma árvore binária onde os dois filhos de um pai são organizados com um à esquerda e outro à direita. Observe que seu código deve definir o atributo pai de ambas as formas filho para a forma CSG em si, como se fossem parte de um grupo.

### Teste # 2: avaliando a regra para uma operação sindical CSG

*Uma união CSG preserva todas as interseções no exterior de ambas as formas.*

Considere a ilustração a seguir, mostrando um raio que cruza duas esferas sobrepostas. Se as duas esferas representam uma operação de união, as interseções destacadas são as que devem ser preservadas; o resto é ignorado.



Você codificará esta regra em uma nova função, chamada intersection_allowed (op, lhit, inl, inr).
Os argumentos são interpretados da seguinte forma:

- op é a operação CSG sendo avaliada.
- lhit é verdade se o deixou forma foi atingida, e falsa se o direita forma foi atingida.
- inl é verdade se o hit ocorrer dentro do deixou forma.

- em R é verdade se o hit ocorrer dentro do direita forma.

Referindo-se à figura anterior, e assumindo que o raio se move da esquerda para a direita, você avaliaria as quatro interseções com as seguintes chamadas para intersection_allowed:

- intersection_allowed ("união", verdadeiro, falso, falso) - o hit está do lado de fora do deixou forma.
- intersection_allowed ("união", falso, verdadeiro, falso) - o hit está do lado de fora do direita forma, enquanto dentro do deixou forma.
- intersection_allowed ("união", verdadeiro, verdadeiro, verdadeiro) - o hit está dentro do deixou forma, enquanto dentro do direita forma.
- intersection_allowed ("união", falso, falso, verdadeiro) - o hit está dentro do direita forma.

Você pode organizar os argumentos nessas chamadas para formar um *mesa da verdade,* um método de mostrar valores de entrada booleanos e a saída esperada de alguma operação neles. O teste a seguir descreve o esboço básico que você usará para exercitar todas as três operações, bem como uma tabela verdade para descrever como o intersection_allowed () função funciona com a operação do sindicato.

**features / csg.feature**

**Esboço do Cenário :** Avaliando a regra para uma operação CSG

**Quando** resultado ← intersection_allowed ( **"< op> " , <** lhit> , <inl> , <inr> )

**Então** resultado = <resultado>

**Exemplos :**

| op | lhit | inl | em R | resultado |
|---|---|---|---|---|
| União | verdade | verdade | verdade | falso | verdade | falso |
| União | verdade | verdadeiro falso | verdade | falso | falso | |
| União | verdade | falso | verdadeiro verdadeiro | verdade | falso |
| União | verdade | | verdade | falso | falso | falso | verdade | |
| União | falso | verdade | | | | |
| União | falso | | | | | |
| União | falso | | | | | |
| União | falso | falso | falso | verdade | | |

O objetivo é implementar intersection_allowed de forma a garantir que retorna a resposta correta para todas as combinações possíveis de entradas, permitindo apenas as interseções corretas.

Para fazer o teste passar, considere o que a operação de união realmente *meios.*
Você só quer as interseções que não estão dentro de outro objeto. Se o acerto for no objeto esquerdo, não deve estar também dentro do direito e, se for na direita, não deve estar simultaneamente dentro do esquerdo. Em pseudocódigo, a lógica se parece com isto:

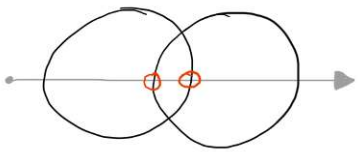
```
função intersection_allowed (op, lhit, inl, inr)
    E se op é " União"
        Retorna ( lhit e não inr) ou (não lhit e não inl)
    fim se

    # resposta padrão
    Retorna falso
função final
```

Implemente isso para fazer o teste passar. Em seguida, você tratará da regra para a operação de interseção.

### Teste # 3: Avaliando a regra para uma operação de interseção CSG

*Uma interseção CSG preserva todas as interseções onde ambas as formas se sobrepõem.*

Dê uma olhada na próxima ilustração, novamente mostrando um raio que cruza duas esferas sobrepostas. Desta vez, porém, os destaques mostram quais interseções são mantidas por uma operação de interseção CSG.



Os cruzamentos são escolhidos de forma a preservar o volume que as formas têm em comum. Adicione a seguinte tabela de verdade ao final daquela que você começou no teste anterior, mostrando como o intersection_allowed ()
função deve se comportar neste caso.

**features / csg.feature**

| # | anexar depois | a União | exemplos... |
|---|---|---|---|
| \| | interseção | \| verdade \| | verdade \| verdade \| verdadeiro verdadeiro \|\| |
| \| | interseção | \| verdade \| | falso \| falso \| falso \| verdade \| verdadeiro |
| \| | interseção | \| verdade \| | falso \| falso \| falso \| verdade \| verdade \| \| |
| \| | interseção | \| verdade \| | verdadeiro verdadeiro \| falso \| verdade |
| \| | interseção | \| falso \| | \| |
| \| | interseção | \| falso \| | \| |
| \| | interseção | \| falso \| | falso \| verdade \| falso \| falso \| falso \| falso \| |
| \| | interseção | \| falso \| | |

Para fazer esses exemplos passarem, você deseja permitir apenas as interseções que atingem um objeto enquanto dentro do outro. Se um raio atinge o objeto à esquerda, a interseção deve estar dentro da direita e vice-versa. Em pseudocódigo, essa lógica se parece com isto:

```
função intersection_allowed (op, lhit, inl, inr)
    E se op é " União"
        Retorna ( lhit e não inr) ou (não lhit e não inl)
```

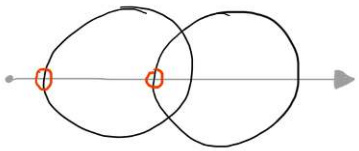
```
➢   mais se op é " cruzar "
➢       Retorna ( lhit e inr) ou (não lhit e inl)
    fim se

    Retorna falso
função final
```

Obtenha esses novos exemplos de passagem e, em seguida, vá para a terceira operação CSG: diferença.

## Teste nº 4: Avaliando a regra para uma operação de diferença CSG

*Uma diferença CSG preserva todas as interseções não exclusivamente dentro do objeto à direita.*

Dê uma olhada no seguinte diagrama de duas esferas sobrepostas. As interseções agora estão destacadas para representar uma operação de diferença CSG.



Adicione esta última tabela verdade ao final das outras duas, para mostrar como a operação de diferença deve funcionar.

**features / csg.feature**

| # | anexar depois | os exemplos de interseção ... | |
|---|---|---|---|
| \| | diferença | \| verdade \| verdade \| verdade \| falso \| verdade \| verdade \| | \| |
| \| | diferença | falso \| verdade \| verdade \| falso \| verdade \| falso \| verdade | \| |
| \| | diferença | \| falso \| falso \| verdade \| falso \| verdade \| verdade \| | \| |
| \| | diferença | verdade \| falso \| verdade \| falso \| verdade \| falso \| falso \| | \| |
| \| | diferença | verdade \| falso \| falso \| falso \| falso \| falso | \| |
| \| | diferença | | \| |
| \| | diferença | | \| |
| \| | diferença | | \| |

A operação de diferença manterá todos os cruzamentos em deixou que *não* lado de dentro direita, *e* cada cruzamento em direita que *é* lado de dentro deixou. Escrito como pseudocódigo, tem a seguinte aparência:

```
função intersection_allowed (op, lhit, inl, inr)
    E se op é " União"
        Retorna ( lhit e não inr) ou (não lhit e não inl)
    mais se op é " cruzar "
        Retorna ( lhit e inr) ou (não lhit e inl)
➢   mais se op é " diferença"
➢       Retorna ( lhit e não inr) ou (não lhit e inl)
    fim se
```

```
    Retorna falso
função final
```

Excelente! Depois que todos esses testes forem aprovados, você estará pronto para começar a filtrar as interseções com base nessas regras.

### Teste nº 5: Filtrando uma lista de interseções

*Dado um conjunto de interseções, produza um subconjunto apenas daquelas interseções que estão em conformidade com a operação do objeto CSG atual.*

Depois de ter o intersection_allowed () função funcionando, você poderá usá-la na próxima parte de sua implementação de CSG: filtragem de interseção. No grande esquema das coisas, quando seu renderizador cruza um raio com um objeto CSG, o objeto CSG produzirá uma lista de interseções entre aquele raio e seus filhos. Esta filter_intersections (csg, xs) função aceita essa lista ( xs), avalia cada interseção com o intersection_allowed () função e retorna uma nova lista de interseção consistindo daqueles que passam.

O teste a seguir cria um csg objeto composto por duas formas. Em seguida, ele cria uma lista de interseções ( xs) e chamadas filter_intersections (). Finalmente, ele verifica se os dois resultado as interseções são o que se espera para a operação atual.

```
features / csg.feature
Esboço do Cenário : Filtrando uma lista de cruzamentos
    Dado s1 ← esfera ()
        E s2 ← cubo ()
        E c ← csg ( "< operação> " , s1, s2)
        E xs ← cruzamentos (1: s1, 2: s2, 3: s1, 4: s2)
    Quando resultado ← filter_intersections (c, xs)
    Então result.count = 2
        E resultado [0] = xs [ <x0> ]
        E resultado [1] = xs [ <x1> ]

    Exemplos :
    | Operação                | x0 | x1 |
    | União                   | 0 | 3 |
    | intersecção | 1 2 | | diferença
                              | 0 | 1
```

Para que isso funcione, seu filter_intersections () função precisa fazer um loop em cada interseção em xs, manter o controle de qual filho a interseção atinge e em quais filhos ele está atualmente e, em seguida, passar essa informação para intersection_allowed (). Se a interseção for permitida, ela será adicionada à lista de interseções de passagem.

Aqui está em pseudocódigo:

```
função filter_intersections (csg, xs)
```

```
    # começa fora de ambos os filhos
    inl ← falso
    em R ← falso

    # prepare uma lista para receber os cruzamentos filtrados
    resultado ← lista de intersecção vazia

    pra cada cruzamento " eu" em xs
        # se i.object é parte do filho "esquerdo", então lhit é verdadeiro
        lhit ← csg.left inclui i.object

        E se intersection_allowed (csg.operation, lhit, inl, inr) então
            adicione i ao resultado
        fim se

        # dependendo de qual objeto foi atingido, alterne entre inl ou inr
        E se lhit então
            inl ← não inl
        outro
            em R ← não inr
        fim se
    fim para

    Retorna resultado
função final
```

Observe a linha com `csg.left inclui i.object`, logo no início do loop for. A implementação disso dependerá de você, mas `A inclui B` deve se comportar assim:

- Se `UMA` é um `Grupo`, a `inclui` operador deve retornar verdadeiro se `criança inclui B` para qualquer filho de `UMA`.

- Se `UMA` é um `CSG` objeto, o `inclui` operador deve retornar verdadeiro se qualquer um dos filhos de `UMA inclui B`.

- Se `UMA` é qualquer outra forma, o `inclui` operador deve retornar verdadeiro se `UMA` é igual a `B`.

Em outras palavras, ele deve pesquisar recursivamente uma subárvore, procurando o objeto dado, para ver se a interseção ocorreu ou não no lado esquerdo da árvore CSG. Se sim, então `lhit` deve ser verdade.

Vá em frente e faça esse teste passar. Depois de fazer isso, você pode passar para os dois últimos testes: certifique-se de que a rotina de interseção real funciona corretamente.

## Testes # 6 e 7: Cruzando um Raio com um Objeto CSG

*Um raio deve cruzar um objeto CSG se cruzar qualquer um de seus filhos.*

Os testes a seguir configuram um objeto CSG e um raio e verificam se o raio se cruza ou não. O primeiro teste garante que um raio erre quando deveria, e o segundo teste verifica se ele atinge quando deveria.

O segundo teste também aplica uma transformação a uma das primitivas para garantir que as interseções resultantes estão sendo filtradas corretamente.

```
features / csg.feature
Cenário :        Um raio perde um objeto CSG
    Dado c       ←   csg ( " União" , esfera (), cubo ())
        E r      ←   raio ( ponto (0, 2, -5), vetor (0, 0, 1))
    Quando xs    ←   local_intersect (c, r)
    Então xs     está vazia

Cenário :        Um raio atinge um objeto CSG
    Dado s1 ← esfera ()
        E s2 ← esfera ()
        E set_transform (s2, tradução (0, 0, 0,5))
        E c ← csg ( " União" , s1, s2)
        E r ← raio ( ponto (0, 0, -5), vetor (0, 0, 1))
    Quando xs ← local_intersect (c, r)
    Então xs.count = 2
        E xs [0] .t = 4
        E xs [0] .objeto = s1
        E xs [1] .t = 6,5
        E xs [1] .objeto = s2
```

Faça esta passagem cruzando o raio com os filhos esquerdo e direito e combinando as intersecções resultantes em uma única lista (ordenada!). Os cruzamentos combinados devem ser passados para filter_intersections (), e a coleção filtrada deve ser retornada.

Em pseudocódigo, é assim:

```
função local_intersect (csg, raio)
    leftxs ← cruzar (csg.esquerdo, raio) rightxs ← cruzar (csg.right, ray)


    xs ← combinar leftxs e rightxs xs ← classificar xs por t


    Retorna filter_intersections (csg, xs)
função final
```

E isso conclui sua implementação de formas CSG! Você nem mesmo precisa calcular um vetor normal; os registros de interseção sempre apontam para o objeto primitivo que foi atingido, e não para a forma CSG pai, o que significa que a própria forma primitiva sempre executará o cálculo normal. Arrumado!

Uma última coisa a falar é como aplicar cor às formas CSG.

## Colorir CSG Shapes

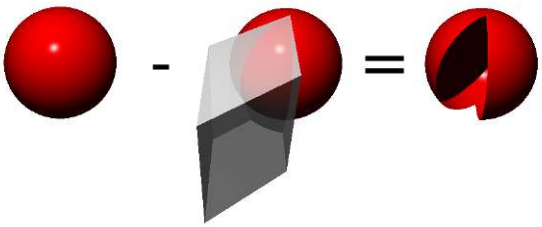
Você deve se lembrar que, no início deste capítulo, eu disse que era possível colorir estrategicamente partes de uma forma CSG. É verdade - a chave é

lembre-se de que cruzar um raio com uma forma CSG preserva as interseções originais com as formas primitivas originais. Pense em como o seu ray tracer determina a cor a ser usada para uma determinada interseção. No Capítulo 6, Luz e Sombreamento, na página 75 , você armazenou um material estrutura em cada objeto e usou o material do objeto interceptado para determinar a cor da interseção.

Isso ainda se aplica às interseções CSG. Considere novamente esta ilustração de uma esfera vermelha subtraída de um cubo amarelo.

As faces do cubo permanecem amarelas, mas a parte que foi subtraída mantém o vermelho da esfera! Isto é porque
essas interseções eram da esfera e não do cubo e, portanto, mantêm a coloração original da esfera.

Isso funciona mesmo com superfícies reflexivas e transparentes, o que significa que você pode fazer certas faces “desaparecerem” tornando sua forma correspondente transparente. Por padrão, as superfícies transparentes ainda projetam sombras, mas se você voltar a Juntando tudo, na página 165 , você verá que uma das coisas opcionais a considerar é que as formas “optem” por projetar sombras. Implementar isso e, em seguida, subtrair formas transparentes de sólidos, permite que você faça coisas interessantes como esta esfera com uma cunha removida dela:




Você pode formar a cunha girando um cubo 45 graus em torno do y eixo e, em seguida, tornando-o mais estreito, dimensionando-o menor em z. Torne a cunha transparente, posicione-a de forma que intercepte a esfera e, a seguir, subtraia-a da esfera. É um truque divertido!

## Juntar as peças

Pensar em termos de CSG pode ser desafiador se você não estiver acostumado. É preciso prática para aprender a ver o mundo ao seu redor como uniões, interseções e diferenças de formas primitivas. Aqui estão algumas coisas que você pode fazer com CSG e algumas dicas de como construí-las.

- Uma lente. (A interseção de duas esferas.)

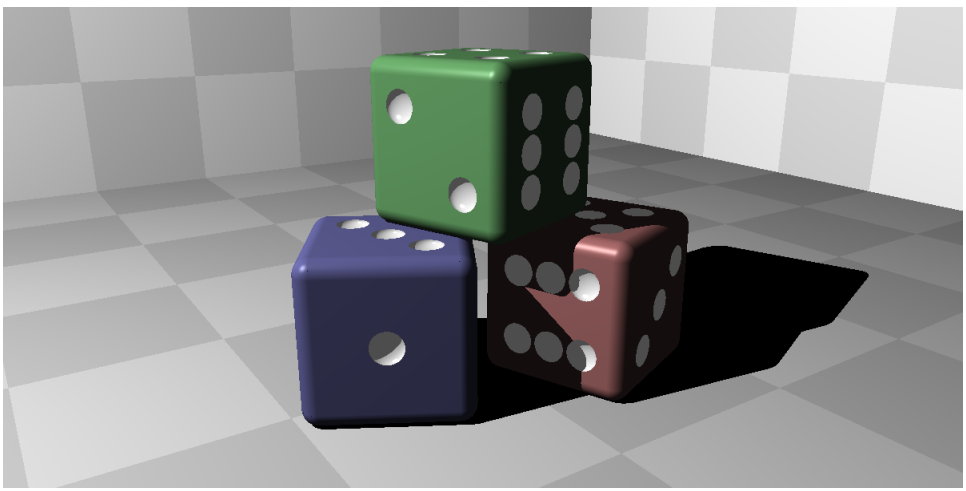
- Um dado de seis lados. (Um cubo, principalmente, mas usando operações de diferença CSG com esferas em escala para formar os pips.)

- Uma letra maiúscula ou número. (Talvez de um cubo achatado, com pedaços raspados usando diferenças com cubos e cilindros.)

- Uma flor. (Talvez formar as pétalas fora de esferas, estrategicamente dimensionadas e moldadas pela interseção de outras esferas.)

- O planeta Saturno. (Forme cada anel subtraindo um cilindro do outro.)

Se você se sente ambicioso, pense em como pode aumentar o realismo de cada um deles. Por exemplo, os dados na vida real não são cubos perfeitos, mas têm bordas e cantos arredondados. Como você criaria esse efeito, usando apenas o que implementou até agora em seu ray tracer?



O que mais você pode imaginar? Mobília? Edifícios? Dragões, cavaleiros e castelos? Trens, aviões ou automóveis? Poderia muito bem fazer uma ou duas espaçonaves, porque o céu é o limite!

Neste ponto, você está pronto para o Desafio Ray Tracer, mas vá em frente e vire a página de qualquer maneira. Vamos falar sobre onde você pode levar seu ray tracer a seguir, porque como todos os melhores projetos, sempre há outro recurso que você pode adicionar.

CAPÍTULO 17

# Próximos passos

Aqui está você no final. Que passeio, hein? De tuplas e matrizes, você procedeu por todo o caminho através de interseções de raio-esfera, sombreamento e sombras, padrões, reflexos e refrações, e assim por diante até a geometria sólida construtiva. Você construiu algo para se orgulhar.

Como acontece com qualquer bom projeto, porém, o “fim” é apenas uma linha desenhada na areia. O livro termina aqui, mas você pode adicionar *muito mais* ao seu ray tracer, e o caminho que você toma depende inteiramente de você. Os novos recursos são limitados apenas pela sua imaginação (e, talvez, pela sua perseverança diante de um pouco mais de matemática). Aqui estão algumas idéias que você pode usar como pontos de partida para sua própria experimentação e pesquisa.

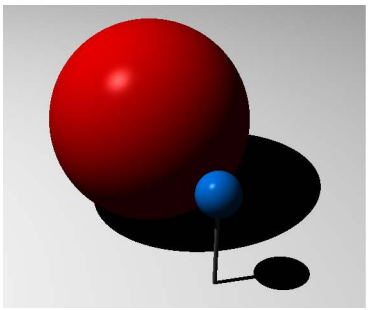
Vamos começar iluminando algumas variações da fonte de luz.

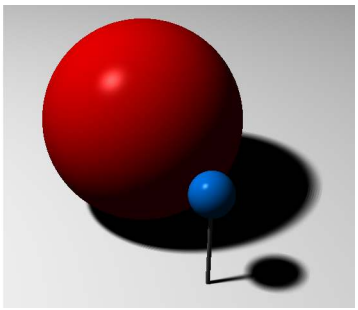
## Luzes da área e sombras suaves

Seu ray tracer atualmente implementa *luzes pontuais,* que existem em um único ponto e não têm tamanho. Essas luzes lançam sombras nítidas e definidas com contornos perfeitamente definidos. Mas no mundo físico, um ponto de luz realmente não existe. As fontes de luz têm dimensão e, como resultado, as sombras que projetam tendem a ser difusas nas bordas.

Considere o ilustração na página 240 , que compara as sombras projetadas por uma luz pontual (à esquerda), com as sombras projetadas por um *luz da área (* a direita).

Essas sombras borradas não são baratas. Lembre-se de Capítulo 8, Sombras, na página 109 , que seu teste de sombra atual projeta um único raio do ponto de intersecção até a fonte de luz. Isso resulta em um resultado booleano “sim / não”, respondendo à pergunta se o ponto está na sombra. Para uma luz de área, você deve lançar *múltiplo* raios de sombra, e a resposta não é mais "sim" ou "não", mas um valor de intensidade informando a você *quantos* a sombra existe nesse ponto.





Para implementar uma luz de área, siga estas etapas:

1. Decida quantos raios de sombra você deseja lançar para cada luz de área. Quanto mais você projeta, melhor a aparência da sombra, mas também significa que o seu ray tracer tem que fazer mais trabalho por pixel.

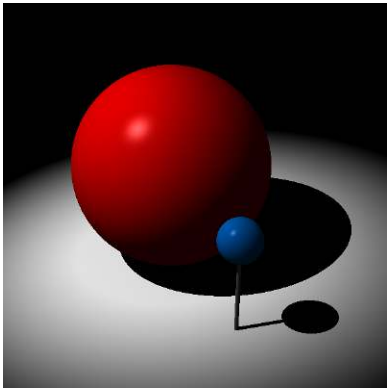
2 Lance cada raio do ponto de intersecção para um ponto diferente na luz de sua área.

3 Calcule a intensidade da luz como o número médio de raios que não foram bloqueados por nenhuma superfície intermediária.

Porém, as fontes de luz são de vários tipos. Continue lendo para outro.

## Holofotes

Outra característica das luzes pontuais é que elas brilham igualmente em todas as direções. Mas pode ser divertido quebrar essa suposição e ter suas luzes focadas em um ponto específico. O resultado é um *Holofote,* como isso:



Para fazer isso funcionar, atribua à sua luz uma direção e um ângulo que descreva a largura do feixe. Então, qualquer ponto que caia fora do cone de luz é considerado na sombra. Se você quiser que o feixe tenha um limite flexível, você pode

definir um segundo ângulo de “fade”, dentro do qual o feixe se mescla de intensidade total a nenhuma.

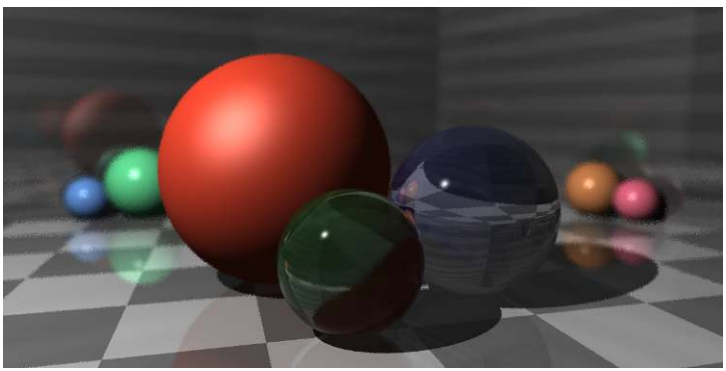
**Implementando holofotes**

Lembre-se de que o produto escalar de dois vetores unitários é igual ao cosseno do ângulo entre eles. Se você pegar o produto escalar do vetor de direção da luz e o vetor do ponto de interseção até a luz, você terminará com o cosseno do ângulo entre eles. Relacione isso ao ângulo do próprio holofote e você terá o recurso pronto pela metade!

Os holofotes são uma forma eficaz de focar a atenção em um ponto específico da cena, mas não são a única maneira. Aqui está outro.

## Desfocagem focal

O desfoque focal ajuda a chamar a atenção do visualizador para o assunto da imagem, fazendo com que ele apareça nitidamente no foco. Objetos que estão muito longe - ou muito perto - da câmera aparecerão fora de foco. Aqui está um exemplo:



O foco aqui está nas três bolas em primeiro plano; as bolas menores no fundo e os reflexos nas paredes são borrados para enfatizar sua distância e falta de importância.

Para fazer isso funcionar, você precisa simular uma câmera com um maior *abertura* - o orifício pelo qual a luz entra na câmera. Por padrão, seu ray tracer imita uma câmera pinhole, com um orifício exatamente grande o suficiente para um único raio de luz. Isso permite que toda a cena apareça nitidamente em foco. Ao aumentar a abertura, a luz pode chegar de vários pontos ao mesmo tempo, desfocando a imagem nesses locais.

Para implementar isso, você precisa especificar o tamanho da abertura e a distância focal da câmera. Qualquer coisa situada na distância focal da câmera será

em foco, então você geralmente coloca o assunto da foto lá. E quanto maior a abertura, mais desfocadas as coisas ficam.

No Implementando uma câmera, na página 100 , você define sua unidade de tela 1 na frente da câmera. Isso codificou efetivamente sua distância focal para 1. Alterar a distância focal basicamente posiciona sua tela a essa distância da câmera. Depois de posicionar a tela, você lança vários raios para cada pixel e, quanto mais, melhor. Em vez de lançá-los todos a partir de um único ponto na origem, você colocará seu *abertura* na origem e escolha vários pontos em sua superfície. Em seguida, para cada um desses pontos, construa um raio que passa do ponto através do pixel atual na tela e sai para a cena. Faça a média das cores de cada um dos raios que você lança por pixel, e aí está o seu desfoque focal!

Podemos resumir:

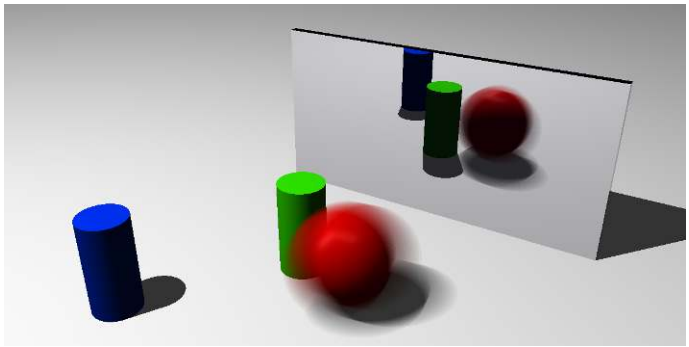
1. Escolha um tamanho para a abertura.
2. Escolha uma distância para a tela.
3. Lance vários raios de pontos aleatórios na abertura, através do pixel na tela.

4. Defina o pixel para a média de todos os raios lançados por ele.

Lançar vários raios por pixel como esse é caro do ponto de vista computacional, porém, funciona muito mais difícil para cada pixel. Ainda assim, é uma técnica versátil que pode criar uma variedade de efeitos. Continue lendo para outro!

## Desfoque de movimento

*Motionblur* é o efeito que você vê nas fotografias de um objeto em movimento rápido, onde parece borrado porque estava em movimento enquanto o obturador da câmera estava aberto. Isso não só pode chamar a atenção do espectador, mas também pode tornar suas cenas mais dinâmicas, adicionando um senso de ação. Um artista habilidoso pode fazer isso com apenas alguns traços de uma caneta, mas para o resto de nós, há um desfoque de movimento, como este:

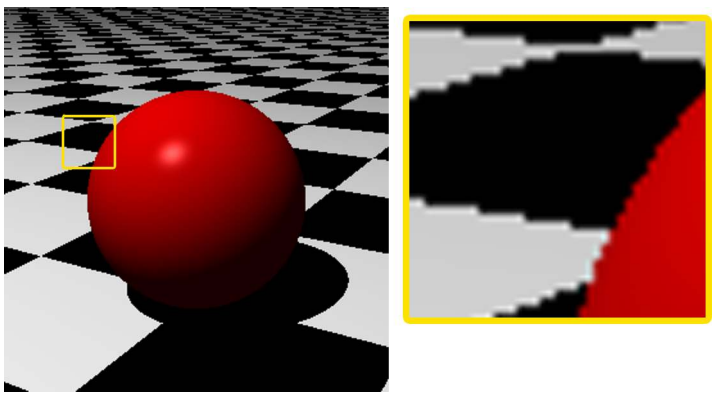


No traçado de raios, você *poderia* simule esse efeito renderizando sua cena várias vezes, movendo um ou mais objetos um pouco em cada quadro e, em seguida, calculando a média de todos os quadros juntos. Mas aqui está outra maneira: para cada pixel de sua imagem, lance vários raios e atribua a cada um deles um valor de tempo. Quando um raio cruza um objeto em movimento, seu ray tracer transforma o objeto de acordo com o valor de tempo associado antes de cruzá-lo com o raio. A cor resultante para cada raio é então calculada em média antes de ser gravada em sua tela.

Você também pode fazer otimizações. Se você definir uma caixa delimitadora ao redor do objeto em movimento, contendo-o completamente em todos os pontos de seu movimento, só precisará lançar raios adicionais se o primeiro raio cruzar essa caixa delimitadora. Isso evita que um pequeno objeto em movimento atrapalhe toda a cena desnecessariamente.

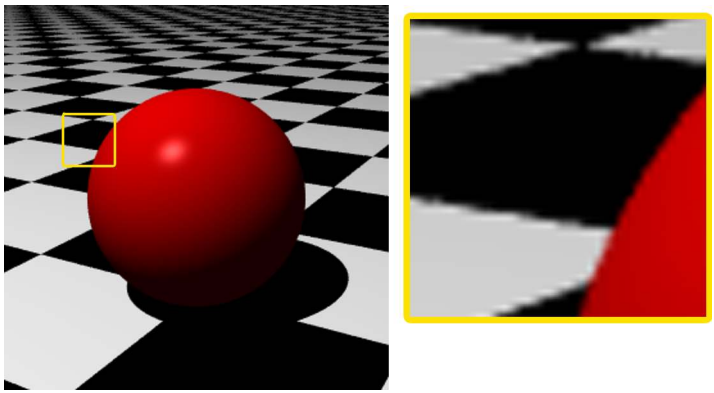
## Anti-aliasing

Como os pixels não são infinitamente pequenos, as linhas diagonais tendem a ser renderizadas como degraus ou *jaggies.* A ilustração a seguir mostra uma visualização ampliada:



Este fenômeno é chamado *aliasing,* e muito esforço é necessário para contornar isso em imagens de qualidade de produção. Um tal *anti-aliasing* técnica renderiza a imagem em uma resolução muito mais alta (dupla, tripla ou mais) e, em seguida, requer um editor de imagem separado para *redimensionamento* a imagem em uma resolução menor. Isso basicamente calcula a média dos valores de pixels adjacentes e ajuda a suavizar as escadas irregulares.

Você pode anti-alias em uma única etapa, no entanto, usando uma técnica chamada *superamostragem.* Em vez de lançar um único raio para cada pixel, você lança vários raios, cada um passando por um ponto diferente (e talvez aleatório) deslocado do centro do pixel, e calcula a média de seus resultados. Mais uma vez, quanto mais raios você lançar, maior será a qualidade do resultado. O imagem na página 244 é um exemplo sem serrilhado da imagem anterior, renderizada usando esta técnica de superamostragem.

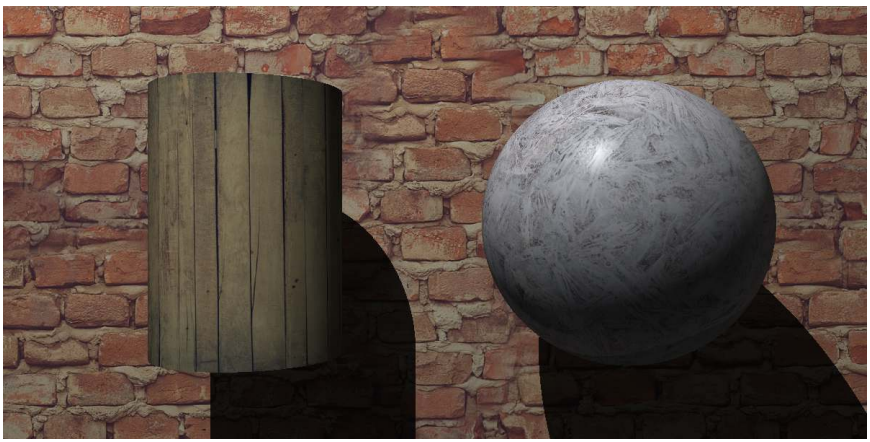


Mais raios significam mais trabalho, então não espere que essa técnica seja barata. Você pode otimizá-lo, no entanto. Por exemplo, em vez de lançar sempre o mesmo número de raios por pixel, você pode começar lançando um em cada canto do pixel e um no centro do pixel. Se algum dos cantos diferir do centro em mais do que algum valor limite, você pode subdividir esse quarto do pixel e repetir o processo, recursivamente. Mesmo assim, devido ao custo de tempo e energia, você desejará economizar para o final do processo de produção.

Você pode fazer muitas outras coisas com o seu ray tracer que não exigem a projeção de mais raios. Por exemplo, você pode fazer papel de parede com texturas e imagens em formas com mapas de textura.

## Mapas de textura

Além das texturas sólidas (xadrez, anéis e assim por diante) que você já implementou, é possível aplicar uma imagem externa a um objeto como um *mapa de textura*. Aqui está um exemplo usando um mapeamento planar, um mapeamento cilíndrico e um mapeamento esférico:



Para um mapeamento plano, você pega uma imagem e mapeia um ( x, y) emparelhar no espaço do objeto a um pixel correspondente na imagem. Normalmente, você também precisará fazer alguma interpolação, já que o ponto em questão frequentemente ficará entre pixels adjacentes.

O mapeamento cilíndrico é um pouco mais complicado, pois você precisa converter um ponto na superfície de um cilindro em uma imagem, muito parecido com o rótulo de uma lata de sopa. Você pode evitar uma dor de cabeça se presumir que está sempre mapeando em um cilindro unitário entre y = 0 e y = 1.

O mapeamento esférico é semelhante ao mapeamento cilíndrico, mas com comportamento diferente nos pólos, onde uma linha inteira da imagem é mapeada para um único ponto da esfera. Você precisará converter um ponto 3D no espaço em um ponto 2D na superfície da esfera, da mesma forma que encontrar a latitude e a longitude de um ponto na superfície da Terra.

Você também pode fazer outros tipos de mapeamento: cúbico, toroidal e assim por diante. Em cada caso, mire uma forma de tamanho constante (raio de 1, por exemplo) e faça a matemática funcionar dessa maneira. Então, com um pouco de escala, o resultado pode ser bastante convincente!

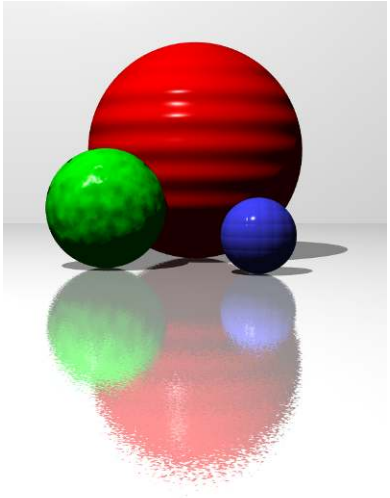
Na verdade, limpar as características da superfície de seus primitivos é uma maneira testada e comprovada de fazer suas cenas brilharem. Aqui está outra técnica divertida, na qual você mente (de uma forma perfeitamente moral) para o seu renderizador.

## Perturbação normal

Volte para Triângulos suaves, na página 218 , quando você fez triângulos parecerem curvos, modificando o normal que foi relatado ao renderizador. Acontece que essa técnica pode ser usada de várias maneiras, basicamente “mentindo” para o renderizador de forma que o sombreamento da superfície seja feito com normais modificados. Confira o imagem na página 246 , mostrando esta técnica aplicada a esferas e um plano.

Ao anexar uma função à forma (talvez diretamente, ou talvez através do material), você pode ter seu normal_at () função chamar a função anexada para
*perturbar,* ou adicione um pequeno vetor ao normal em um determinado ponto. Na imagem anterior, a esfera vermelha usa uma função seno para fazer a superfície parecer ondulada, a esfera azul usa outra função para dar à superfície uma aparência acolchoada, e a esfera verde e o plano estão usando ruído tridimensional para fazer a superfície parecer deformado.

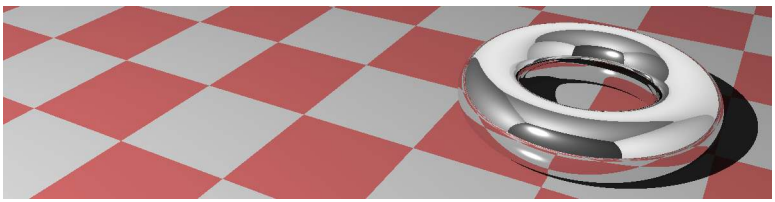
Essa técnica pode ser aplicada ao vidro para fazê-lo parecer gravado ou fosco também. E você pode até mesmo combiná-lo com o mapeamento de textura para permitir que um arquivo de imagem defina um *mapa normal* que descreve como o normal deve ser perturbado em qualquer ponto.



Além de texturas, você também pode explorar novas formas primitivas, para aumentar a variedade de suas cenas. Um desses primitivos que você pode experimentar é o toro.

## Toro primitivo

UMA *toro* é uma forma de anel ou donut, como este:



Eles são primitivos muito legais e versáteis, mas são um pouco mais avançados do que os primitivos que você implementou até agora. Esferas, cilindros e cones são chamados *quádrica* superfícies, o que significa que podem ser descritas por equações de segundo grau (aquelas em que nenhuma variável é elevada a uma potência superior a
2). Um toro, porém, é um *quártico* superfície, com variáveis elevadas à quarta potência. Isso significa que você precisa de um solucionador de equações quárticas para encontrar a interseção entre um toro e um raio.

Mas isso significa apenas que você vai cavar mais fundo do que antes, certo? Se você ainda não tem acesso a um solucionador de equação quártica, pode dar uma olhada no método Durand-Kerner.[1] Não é o mais rápido, mas é menos intimidante

1 en.wikipedia.org/wiki/Durand-Kerner_method

do que alguns outros métodos. E Marcin Chwedczuk escreveu um artigo chamado "Ray traçando um toro [2] "Que pode ajudar a lançar alguma luz sobre o assunto para você.

Veja quais outras formas primitivas você pode implementar!

## Embrulhando-o

Por mais exaustiva que essa lista possa parecer, ela ainda é apenas uma amostra do que você pode tentar. Você pode fazer muito mais, como efeitos volumétricos como fumaça, névoa, nuvens e fogo. Ou talvez mapeamento de radiosidade e fótons para efeitos de iluminação mais realistas ou paralelização para renderização mais rápida em máquinas com multiprocessador. Você nunca termina de verdade, mas essa é a coisa maravilhosa sobre projetos como esse!

O resto é com você. Busque os recursos que mais o entusiasmam. Explore seus próprios interesses. Faça deste ray tracer seu próprio. Mas o mais importante de tudo:
*divirta-se!*

2 marcin-chwedczuk.github.io/ray-tracing-torus

APÊNDICE 1

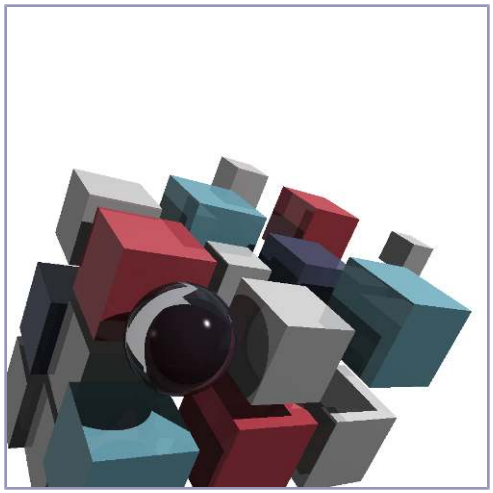
# Renderizando a imagem da capa

A imagem da capa deste livro foi renderizada usando o próprio ray tracer que o livro descreve, o que significa que, depois de implementar os recursos necessários, você também pode renderizá-la!

Para renderizar a imagem da capa, o seu ray tracer deve suportar os recursos descritos acima Capítulo 12, Cubos, na página 167 . A imagem da capa real foi renderizada usando duas fontes de luz, embora a segunda seja opcional.

A cena da imagem da capa é descrita aqui em YAML[1] formato. Se existir um analisador YAML para sua linguagem de programação, você poderá construir a cena diretamente a partir desta descrição; caso contrário, você precisará traduzir esta descrição para qualquer API que você construiu para seu próprio renderizador.

Se renderizado conforme descrito, você obterá algo parecido com isto:



1 yaml.org

E aqui está a própria descrição da cena:

**cover.yml**

```
#   ========================================================== ====
#   a câmera
#   ========================================================== ====

-   adicionar : Câmera
    largura : 100
    altura : 100
    campo de visão : 0,785
    a partir de : [ -6 , 6 , -10 ]
    para : [ 6 , 0 , 6 ]
    acima : [ -0,45 , 1 , 0 ]

#   ========================================================== ====
#   fontes de luz
#   ========================================================== ====

-   adicionar : claro
    no : [ 50 , 100 , -50 ]
    intensidade : [ 1 , 1 , 1 ]

#   uma segunda luz opcional para iluminação adicional
-   adicionar : claro
    no : [ -400 , 50 , -10 ]
    intensidade : [ 0,2 , 0,2 , 0,2 ]

#   ========================================================== ====
#   defina algumas constantes para evitar duplicação
#   ========================================================== ====

-   definir : material branco
    valor :
        cor : [ 1 , 1 , 1 ]
        difuso : 0,7
        ambiente : 0,1
        especular : 0,0
        reflexivo : 0,1

-   definir : material azul
    ampliar : material branco
    valor :
        cor : [ 0,537 , 0,831 , 0,914 ]

-   definir : material vermelho
    ampliar : material branco
    valor :
        cor : [ 0,941 , 0,322 , 0,388 ]

-   definir : material roxo
    ampliar : material branco
    valor :
        cor : [ 0,373 , 0,404 , 0,550 ]
```

```
-   definir : transformada padrão
    valor :
        - [ traduzir , 1 , -1 , 1 ]
        - [ escala , 0,5 , 0,5 , 0,5 ]

-   definir : objeto grande
    valor :
        -   transformada padrão
        - [ escala , 3,5 , 3,5 , 3,5 ]

-   definir : objeto médio
    valor :
        -   transformada padrão
        - [ escala , 3 , 3 , 3 ]

-   definir : objeto pequeno
    valor :
        -   transformada padrão
        - [ escala , 2 , 2 , 2 ]

# ========================================================== =====
# um fundo branco para a cena
# ========================================================== =====

-   adicionar : plano
    material :
        cor : [ 1 , 1 , 1 ]
        ambiente : 1
        difuso : 0
        especular : 0
    transformar :
        - [ rotate-x , 1.5707963267948966 ] # pi / 2
        - [ traduzir , 0 , 0 , 500 ]

# ========================================================== =====
# descreve os elementos da cena
# ========================================================== =====

-   adicionar : esfera
    material :
        cor : [ 0,373 , 0,404 , 0,550 ]
        difuso : 0,2
        ambiente : 0,0
        especular : 1.0
        brilho : 200
        reflexivo : 0,7
        transparência : 0,7
        índice de refração : 1,5
    transformar :
        -   objeto grande
```

- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto médio**
    - [ traduzir , 4 , 0 , 0 ]

- adicionar : **cubo**
  material : **material azul**
  transformar :
    - **objeto grande**
    - [ traduzir , 8,5 , 1,5 , -0,5 ]

- adicionar : **cubo**
  material : **material vermelho**
  transformar :
    - **objeto grande**
    - [ traduzir , 0 , 0 , 4 ]

- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto pequeno**
    - [ traduzir , 4 , 0 , 4 ]

- adicionar : **cubo**
  material : **material roxo**
  transformar :
    - **objeto médio**
    - [ traduzir , 7,5 , 0,5 , 4 ]

- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto médio**
    - [ traduzir , -0,25 , 0,25 , 8 ]

- adicionar : **cubo**
  material : **material azul**
  transformar :
    - **objeto grande**
    - [ traduzir , 4 , 1 , 7,5 ]

- adicionar : **cubo**
  material : **material vermelho**
  transformar :
    - **objeto médio**
    - [ traduzir , 10 , 2 , 7,5 ]

- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto pequeno**
    - [ traduzir , 8 , 2 , 12 ]

- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto pequeno**
    - [ traduzir , 20 , 1 , 9 ]
- adicionar : **cubo**
  material : **material azul**
  transformar :
    - **objeto grande**
    - [ traduzir , -0,5 , -5 , 0,25 ]
- adicionar : **cubo**
  material : **material vermelho**
  transformar :
    - **objeto grande**
    - [ traduzir , 4 , -4 , 0 ]
- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto grande**
    - [ traduzir , 8,5 , -4 , 0 ]
- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto grande**
    - [ traduzir , 0 , -4 , 4 ]
- adicionar : **cubo**
  material : **material roxo**
  transformar :
    - **objeto grande**
    - [ traduzir , -0,5 , -4,5 , 8 ]
- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto grande**
    - [ traduzir , 0 , -8 , 4 ]
- adicionar : **cubo**
  material : **material branco**
  transformar :
    - **objeto grande**
    - [ traduzir , -0,5 , -8,5 , 8 ]

De uma chance. Experimente. Aplique materiais, padrões e iluminação diferentes. Experimente diferentes formas e perspectivas. Veja o que você pode inventar!

# Índice


## SÍMBOLOS

/ ( barra), delimitando
  informação do vértice, 225

## DÍGITOS

Base 0
  vértices, 214
  x e y parâmetros, 19

Matrizes 2x2
  criando, 26
  determinantes, achados, 33-38
  exemplo, 25
  inversão de matrizes, 34-42

3D
  padrão verificador, 136-138
  Arquivos OBJ, 212-218 , 223 , 226
  recursos em modelagem, 225
  exercício de silhueta, 89-90

Matrizes 3x3
  cofatores, 36
  criando, 26
  menores, 35

Matrizes 4x4, *Veja também* matrizes
  definiram, 26
  determinantes, achados, 38

## UMA

AABB, *Vejo* alinhado ao eixo caixas delimitadoras (AABB)

abstração, definida, 119

abstrações, limitando, 4

acne, 114

Adição
  cores, 16
  tuplas, 5

aliasing, 243

reflexão ambiente
  vidro colorido, 165
  definiram, 84
  Modelo de reflexão Phong, 84-89
  refração e transparency, 165
  marcante não reflexivo superfície, 144
  padrão listrado, 129

exercício anti-aliasing, 243

abertura, efeito de desfoque de foco, 241

APIs, fluente, 55

arquitetura, escolha, xx

exercício de iluminação da área, 239

arrays, concatenando exemplos de pepino, xvii

proporção da tela, 102

caixas delimitadoras alinhadas ao eixo (AABB), 167-176
  cerca de, 167
  calculando normal em um cubo, 173-175
  cruzando um raio com um cubo, 168-173
  faltando um raio com um cubo, 172

## B

estrutura de árvore bidirecional, 194

padrões combinados, 139

Liquidificador, 226

funções de mistura, 135 , 139

caixas delimitadoras
  eixo alinhado (AABB), 167-176
  calculando normal em um cubo, 173-175
  cruzando um raio com um cubo, 168-173
  exercício de desfoque de movimento, 243
  otimizar cenas com, 200–203
  transformações, 168 , 202
  não transformado, 202

## C

Câmera
  abertura, 241
  proporção da tela, 102
  atributos, 100
  tamanho da tela, 100 , 102
  campo de visão, 100-103
  exercício de desfoque focal, 241-242
  implementando em vista transformações, 100–105
  exercício de desfoque de movimento, 242
  exercício de múltiplas esferas, 107
  renderizando imagem, 104
  especificando o comprimento focal, 241

tela de pintura, 15–24
  atributos da câmera, 100 , 102
  cores, adicionando, 15–18

convertendo coordenação mundial nates, 22
criando, 19
exercícios, 22–24
salvando, 19-22
Tamanho, 72 , 100 , 102

cgtrader, 225

limites de caracteres, arquivos PPM, 21

padrão verificador, 136-138

crianças
adicionando, 195
Formas CSG, 230
filtrando CSG intersecções, 234
achando normal, 197-200
transformações, 197

Chwedczuk, Marcin, 246

ClaraIO, 225

exercícios de relógio, 54-56

código, para este livro, xviii

cofatores
expansão do cofator, 33 , 36-38
definiram, 36
determinantes, achados, 36-38
inversão de matrizes, 36-42
matriz de cofatores, 40

cores
Esfera 3D silhueta exercício cise, 89
adicionando à tela, 15–18
padrão combinado, 139
Misturando, 17 , 135 , 139
padrão verificador, 136-138
preso, 21
Formas CSG, 227 , 236
copo, 165
padrão de gradiente, 134 , 139
valor máximo da cor em Arquivos PPM, 20
espelhos, 165
exercício de múltiplas esferas, 107
padrão aninhado, 139
operações, 16-18
padrões, generalizando, 132-134
padrão perturbado, 140 , 245
luzes pontuais, 84
padrão de gradiente radial, 139
reflexão, 144-149 , 165
refração, 150 , 155-158 , 165
padrão de anel, 135 , 139
sombreamento na construção do mundo ing, 96
sombras, renderização, 113
marcante não reflexivo superfície, 144
superfície reflexiva impressionante, 144
padrões listrados, 128-132
teoria, 15
como tuplas, 15

colunas
especificando ao criar matrizes, 26
matrizes de transposição, 32

comparando
estruturas de dados, xix
Números de ponto flutuante, xix , 5
matrizes, 27
pontas, xix

cones, 178 , 188-191

geometria sólida construtiva (CSG), 227-238
vantagens, 227
cores, 227 , 236
criando formas, 229
filtrando cruzamentos, 234-236
implementando, 229-236
cruzamentos, 229-236
pontas, 237

coordenadas
convertendo para inteiros, 22
convertendo coordenação mundial nates para as coordenadas da tela, 22
convertendo o espaço do mundo em espaço do objeto, 68 , 80 , 131 , 197-199
lançando y coordenada, 73
canhoto vs. destro entregue, 3

cosseno
produto escalar, 11
rotação, 47
reflexão interna total, 162

imagem da capa, renderização, 249-253

produto cruzado, 11

CSG, *Vejo* sólido construtivo geometria (CSG)

cubos, 167-176 , *veja também* caixas delimitadoras
cerca de, 167
lançando raio de dentro, 169
exercícios, 175
cruzando um raio, 168-173
faltando um raio, 172
normal, calculista, 173–175
atuação, 176

Pepino
cerca de, xvii
recursos em, xviii

cilindros, 177-191
cerca de, 177
tampado, 178 , 184-188
exercício de forma composta, 203-205
computação normal, 178 , 180 , 184 , 187
oco, 181-184
cruzando um raio com, 178-180 , 185
faltando um raio, 178
raio, 177
sólido, 178 , 184-188
mapas de textura, 244
truncando, 177-178 , 181-184

mapeamento cilíndrico, 244


## D


tipos de dados
evitando complexo, 4
comparando, xix

graus, convertendo para radians, 48

determinantes
encontrar matrizes, 34–38
raios que se cruzam com um triângulo, 210
inversão de matrizes, 34-38
matrizes não conversíveis, 39

diferença, comparando float-números de pontos ing, 5

diferença (CSG), 228 , 233

reflexão difusa
vidro colorido, 165
definiram, 84
Modelo de reflexão Phong, 84-89
transparência, 165

direção
determinação da direção de normais, 95

ordem de produto cruzado
operandos, 12
raios, 57 , 69
compreender tuplas, 2
discriminante
computação para raio-esfera cruzamentos, 62
raios que se cruzam com um cilindro, 179
distância
encontrando pontos em um raio, 58
compreender tuplas, 2
divisão, tuplas, 7
produto escalar
Informática, 10
definiram, 10
determinação da direção de normais, 95
multiplicando matrizes, 30
holofotes, 241
cones com crista dupla, 188–191
Método Durand-Kerner, 246


## E


E para vetor de olho, 76
vetores de borda, criando triangles, 208
tampas finais
computação normal, 184 , 187 , 190
cones, 190
cilindros, 178 , 184-188 , 190
EPSILON
pontos de colisão com, 115
comparando ponto flutuante números, xix , 5
computando normais em tampas finais, 188
Informática under_point, 154
exercícios
cerca de, xviii
anti-aliasing, 243
luzes da área, 239
tela de pintura, 22–24
relógio, 54-56
cubos, 175
borrão focal, 241-242
grupos, 203-205
cruzamentos, 70-74
luzes, 89-90 , 239-240
matrizes, 42 , 54-56
borrão de movimento, 242
perturbação normal, 245
padrões, 138-140
aviões, 124
pontos, 12
reflexão e refração, 165
renderizando a imagem da capa, 249-253
cenas, 105-107
sombreamento, 89-90
sombras, 116 , 166 , 239
silhueta, 70-74 , 89-90
esferas, 70-74 , 89-90 , 105-107 , 124 , 203-205
holofotes, 240
mapas de textura, 244
toro, 246
transformações, 54-56
tuplas, 12
vetores, 12
projétil virtual, 13 , 22-24
vetor do olho
Esfera 3D silhueta exercício cise, 89
Informática, 76
definiram, 76
Modelo de reflexão Phong, 85-89
sombreamento na construção do mundo ing, 93-97
ver transformações, 97-100


## F


f, Arquivos OBJ, 212 , 214 , 223
rostos, arquivos OBJ, 212 , 214 , 223
triangulação de leque, 215
campo de visão, 100-103
filtragem, cruzamentos, 234-236
lançando, *Vejo* reflexão
Números de ponto flutuante
comparando, xix , 5
computando normais em tampas finais, 188
convertendo para inteiros, 22
reflexos parciais, 143 , 145
usando nativo vs. abstrações, 4
APIs fluentes, 55
exercício de desfoque focal, 241-242
barra (/), delimitando informação do vértice, 225
frente vetor, 99
Free3D, 225
Efeito Fresnel, 160-165
Fresnel, Augustin-Jean, 160
função, nomes, xx


## G


Maxixe, xvii
GIMP (GNU Image Manipula- programa de instalação), 20
copo
teoria da cor, 16
Sindicatos CSG, 228
fosco ou gravado, 245
índice de refração, 149
esferas, criando, 151
esferas, refletância, 161-165
esferas, refração, 151–154
tingimento, 165
pontas, 165
GNU Image Manipulation Programa (GIMP), 20
padrão de gradiente, 134 , 139
grupos, 193-205
cerca de, 193
adicionando filhos, 195
adicionando pais, 194-195
adicionando formas a, 194
caixas delimitadoras, 200–203
computando normais para, 194 , 197-200
criando, 194
definiram, 194
exercícios, 203-205
implementando, 194-197
cruzamentos, 194 , 196
nomeado, 216
normais, 197-200
padrões, 200
polígonos com arquivos OBJ, 213 , 215-216
transformações, 194 , 197
triângulos, 211 , 213 , 215-216
vs. sindicatos, 228


## H


Produto Hadamard, 17
Problema de parada, 147
exercício composto de hexágono, 203-205
destaques, especular, 84 , 165
exitos
Sindicato CSG, 230
identificando, 64-66

ocluído, 64
ocorrendo dentro de forma, 94
definindo refração, 152
sombreamento na construção do mundo ing, 93-97
triângulos de suavização, 223
testando sombras, 113

hsize, 100 , 102

eu

identidade, multiplicativa, 31

matriz de identidade
como orientação padrão, 98
como transformação padrão para a câmera, 101
definiram, 31
formas, 118
tradução, 45
transpondo, 33

ImageMagick, 22

índice, vértice, 214 , 225

índice de refração, *Vejo* refrac- índice tivo

recursão infinita, 142 , 146

infinidade
raios que se cruzam com cubos, 171
suporte de linguas, 171
representando, xix

produto Interno, *Vejo* ponto prod- uct

intensidade
luzes pontuais, 84
reflexão, 86-87

cruzar (CSG), 228 , 232

cruzamentos, 57-74 , *veja al- então* sombras
agregando, 64
falso, 210
caixas delimitadoras, 168-173
coleções, 152
Informática, 62
cones, 189-191
CSG, 229-236
tampas de extremidade do cilindro, 185
cilindros, 178-180 , 185
exercícios, 70-74
filtragem, 234-236
grupos, 194 , 196
identificando hits, 64-66
ignorando o negativo, 65
ausência de, 60 , 62
ocorrendo dentro de forma, 94
ordem de, 63 , 66 , 93
atuação, 200 , 202
aviões, 118-120 , 122- 124
propriedades, 63
refletância, 161-162
refração, 150-154
recursos em, 62
rotina para a construção do mundo ing, 92
esferas, 59-66 , 70-74
tangentes, 60
rastreamento, 63
raios transformadores e esferas, 66-70
triângulos, 208-212 , 220- 225
triângulos, suavização, 220-225

matriz de transposição inversa, 81

inversão
vantagens, 38
determinando se possível, 39
exercícios, 42
matrizes, 33-42
rotação, 49
escala, 46
matrizes de tradução, 45

J

jaggles, 243

tremendo, 140

eu

eu para vetor de luz, 76

deixou operando, 229

deixou vetor, 99

regra da mão esquerda, 48 , 51

coordenadas para canhotos, 3

comprimento, *Vejo* magnitude

luz, efeito Fresnel, 160-165

vetor de luz, 76

luzes, 75–90 , *Veja também* nem- mals; refração; reflexão; sombreamento; sombras
exercícios, 89-90 , 239- 240
vetor do olho, 76 , 85-89
vetor de luz, 76
exercício de múltiplas esferas, 107
luzes pontuais, 84
destaques especulares, 84 , 165
holofotes, 240
padrão listrado, 129 , 132
usando várias fontes, 96

Sistemas Lindenmayer, 205

álgebra Linear
multiplicação da matriz, 28- 32
recursos em, 31
matrizes de transposição, 32- 42

Sistemas Linux, visualizando PPM arquivos, 20

M

Sistemas Mac, visualizando PPM arquivos, 20

magnitude
Informática, 8
normalizando tuplas, 10

materiais
Esfera 3D silhueta exercício cise, 89
Formas CSG, 227 , 237
predefinição, 118
exercício de múltiplas esferas, 107
padrões, generalizando, 132-134
padrões, listrados, 129
Modelo de reflexão Phong, 85-89
reflexivo, 141-149
índices de refração, 149
texturas, 244
transparência, 149-165

matemática, *Veja também* linear álgebra
cerca de, xviii
cruzamentos, 62
operações de tupla, 5-8

matrizes, 25–42 , *Veja também* iden- matriz de identidade; submatrizes; transformações; traduzir ção
cofatores, matriz de, 40
comparação, 27
criando, 26
definiram, 25
determinantes, achados, 34-38
exercícios, 42 , 54-56
matriz de transposição inversa, 81
inversão, 33-42
multiplicação, 28-32 , 54
multiplicação por identidade matriz, 31

multiplicação por tuplas, 30
não invertível, 39
reflexão, 47
rotação, 47-51 , 54-56
escala, 46
tosquia, 51-53
implementação simples, 27
transpondo, 32 , 42
valor máximo da cor, 20
malha, triângulo, 213
menores
Informática, 35
determinantes, achados, 35-38
inversão de matrizes, 35-42
negando, 36
espelhos
cores, 165
recursão infinita, 146
pontas, 165
Algoritmo Möller-Trumbore, 208-212
exercício de desfoque de movimento, 242
Arquivos de material MTL, 227
multiplicação
cores, 16
por matriz de identidade, 31
pela transposição inversa matrix, 81
matrizes, 28-32 , 54
matrizes, por tuplas, 30
ordem, 54 , 106
matrizes de escala, 46
transformações, 106
matrizes de tradução por vetor, 45
tuplas, 7 , 31
identidade multiplicativa, 31


## N


N para vetor normal de superfície, 76
n 1 / n 2, 150-154 , 161-162
grupos nomeados, 216
nomes, escolhendo, xx
NASA, 225
negando
menores, 36
tuplas, 7
padrões aninhados, 139
caractere de nova linha e PPM
arquivos, 22
interpolação normal, 219-225
Mapeamento normal, 245
perturbação normal, 245
normalização
cerca de, 78 , 82
tuplas, 10
vetores, 9
normais
Esfera 3D silhueta exercício cise, 89
objetos filhos, 197-200
Informática, 118 , 120
computação para um cone, 189-190
computação para um cubo, 173-175
computação para um cilindro, 178 , 180 , 184 , 187
computação para uma esfera, 77-82
computação para um triângulo, 208
computação para grupos, 194 , 197-200
convertendo de objeto espaço para espaço mundial, 197-200
Formas CSG, 236
definiram, 76
direção, determinação, 95
tampas finais, 184 , 187 , 190
achando normal em uma criança objeto, 199
interpolação normal, 219-225
Mapeamento normal, 245
normalização, 78
perturbação, 245
Modelo de reflexão Phong, 85-89
aviões, 122
sombreamento na construção do mundo ing, 93-97
transformando, 79–82
matrizes de transposição, 32
entendimento, 76-77


## O


Formato OBJ, 212-218 , 223 , 226
sobreposição de objetos, 60 , 179
espaço do objeto
convertendo para, 68 , 80 , 131 , 197-200
convertendo-se em pai espaço, 202
convertendo em padrão espaço, 132
convertendo para o mundo espaço, 68 , 80 , 197-199
formas de operando, CSG, 229
ordem
produto cruzado, 12
cruzamentos, 63 , 66 , 93
multiplicação, 54 , 106
transformações, 54 , 106 , 204
origem
normais de computação, 80
raios, 57 , 59 , 122
esferas, 59 , 80
mundo, 59 , 80
over_point, 115 , 145 , 154
objetos sobrepostos, 60 , 179
sombras sobrepostas, 96


## P


parametros
valores padrão, 148
opcional, 153
pais
adicionando, 194-195
convertendo o espaço do objeto para o espaço dos pais, 202
Formas CSG, 230
analisadores
Arquivos OBJ, 212-218 , 223 , 227
YAML, 249
reflexão parcial, 143 , 145
padrões, 127-140
misturado, 139
verificador, 136-138
definiram, 128
exercícios, 138-140
generalizando, 132-134
gradiente, 134 , 139
grupos, 200
aninhado, 139
espaço padrão, 132
perturbado, 140 , 245
gradiente radial, 139
anel, 135 , 139
listrado, 128-132
transformando, 130-132
atuação
cerca de, xix
anti-aliasing, 244
cubos, 176
borrão de foco, 242
cruzamentos, 200 , 202
borrão de movimento, 243
múltiplas fontes de luz, 96

Arquivos OBJ, 218
otimizando cenas com caixas delimitadoras, 200–203
renderizador, 107
sombras, 96
Ruído Perlin, 140
Perlin, Ken, 140
perturbação, normal, 245
padrões perturbados, 140
Modelo de reflexão Phong, 83-89
Phong, Bui Tuong, 83
Pixar, 3
píxeis
anti-aliasing, 243
calculando o tamanho com Câmera, 101–103
escrevendo, 19
mapeamento planar, 244
aviões, 117-125
computação normal, 122
computação reflexão vector, 143-144
como componentes do cubo, 169
definiram, 122
tampas finais como, 187
exercícios, 124
implementando, 122-124
cruzamentos, 118-120 , 122-124
refatoração da esfera exemplo, 117-121
refração, 159-160
cilindros sólidos, 184-188
luzes pontuais, 84
pontos, *Veja também* tuplas
batendo com EPSILON, 115
convertendo de objeto espaço para o espaço dos pais, 202
convertendo do mundo espaço para espaço de objeto, 68 , 80 , 131 , 197-199
convertendo em objeto espaço, 118 , 120
convertendo para um vetor, 120
criando, 4
definiram, 3
distância para raios, 58
exercícios, 12
indicando que o ponto está em sombra, 110
raios que se cruzam com cubos, 171-172
tremendo, 140
origem, 57
fora dos limites, 22
posição em um raio, 58
reflexão, 47
rotação, 47-51
escala, 46
sombras, renderização, 113–116
sombras, testando, 111-113
tosquia, 51-53
tradução, 44-46
tuplas, adicionando, 6
tuplas, subtração, 6
polígonos, *Veja também* triângulos
cerca de, 207
triangulação de leque, 215
agrupamento, 213 , 215-216
hexágono composto exercise, 203-205
Formato Pixmap portátil, *Vejo* Arquivos PPM
posição
encontrando pontos em um raio, 58
compreender tuplas, 2
POV-Ray, 3
Arquivos PPM, 20-22
prepare_computations
adicionando reflexão, 142-148 , 161-165
adicionando refração, 150–154
renderizando sombras, 115
sombreamento na construção do mundo ing, 93-97
triângulos de suavização, 223
exercícios de projétil, 13 , 22–24


## Q


superfícies quádricas, 246
superfícies quárticas, 246


## R


R para vetor de reflexão, 76
padrão de gradiente radial, 139
radianos, 48
elenco de raios, definido, 57 , *Vejo Além disso* cruzamentos
traçadores de raios
cerca de, xv
etapas do algoritmo, xvi
matemática necessária para, xviii
raios, *Veja também* cruzamentos; refração; reflexão
câmera, implementação, 103-105
lançando de dentro de um cubo, 169
convertendo em objeto espaço, 118
coplanar, 122
criando, 57
definiram, xvi
direção, 57 , 69
identificando hits, 64-66
cruzando com CSG formas, 229-236
cruzando com um cubo, 168-173
cruzando com um cilindro der, 178-180 , 185
cruzando com um grupo, 194 , 196
cruzando com o fim cápsulas, 185
cruzando com aviões, 118-120 , 122-124
cruzando com esferas, 59-66
cruzando com triangles, 208-212
cruzamentos, 70-74
mantendo o original, 68
tornando transformável, 68
falta de intersecção com um cubo, 172
falta de intersecção com um cilindro, 178
cruzamentos faltando, 60 , 62
cruzamentos faltando com uma forma CSG, 229 , 235
originando atrás esfera, 61
originando na esfera, 61
girando com, 68
dimensionando com, 67 , 70 , 74
raios de sombra, 111-116
triângulos lisos, 220-225
esferas transformadoras com, 66-70
construção do mundo, 92
recursão
infinito, 142 , 146
limitando, 142 , 147-150 , 155

traçado de raio recursivo, de-
multado, xvi
reflexão, 142 , 146-149
refração, 150 , 155
traçado de raio recursivo, definido, xvi , *Veja também* raios
refletância, efeito Fresnel, 161-165
reflexão, *Veja também* ambiente reflexão
combinando com refracção, 161 , 163
Formas CSG, 237
difuso, 84-89 , 165
exercícios, 165
Efeito Fresnel, 160-165
materiais, adicionando a, 141-149
parcial, 143 , 145
Modelo de reflexão Phong, 83-89
recursão, 142 , 146-149
escala, 47
definir níveis de, 143
reflexão especular, 84-89
marcante não reflexivo superfície, 142 , 144
superfície reflexiva impressionante, 142 , 144
padrão listrado, 129 , 132
pontas, 165
reflexão interna total, 150 , 156 , 159 , 161
entendimento, 82 , 142
vetor, reflexão Phong modelo, 87-89
vetor, computação, 82-83 , 142-144
vetor, definido, 76
ver transformações, 98
refletv, 142-143
refração
adicionando, 149-165
cores, 150 , 155-158 , 165
combinando com reflexão, 161 , 163
Informática under_point, 150 , 154
padrões, 151
definiram, 149
exercícios, 165
Efeito Fresnel, 160-165
superfícies opacas, 150 , 155
recursão, 150 , 155
índice de refração, 149-154
no sombra_hit, 158-160
pontas, 165
reflexão interna total, 150 , 156 , 159 , 161
superfícies transparentes, 150
índice de refração
padrões, 151
definiram, 149
usando, 150-154
índice de refração atributo, 150
renderizador, desempenho, 107
RenderMan, 3
recursos para este livro
modelagem 3d, 225
Pepino, xviii
álgebra Linear, 31
Arquivos OBJ, 218 , 226
Ruído Perlin, 140
traçadores de raios, xx
intersecções de raio-esfera, 62
índices de refração, 149
primitivos de toro, 246
direita operando, 229
coordenadas destras, 3
padrão de anel, 135 , 139
rotação, 47-51 , 54-56 , 68
filas
especificando ao criar matrizes, 26
matrizes de transposição, 32


## S


salvando tela, 19-22
divisão escalar, tuplas, 7
multiplicação escalar
cores, 16
tuplas, 7
produto escalar, *Vejo* ponto product
valor escalar
definiram, 8
produto escalar, 10
escala
formas compostas, 204
inverso, 46 , 67
matrizes, 46
reflexão, 47
esferas com raios, 67 , 70 , 74
tuplas, 46
vetores, 46
cenários
Pepino, xvii
formato de esboço, 151
cenas, 91–107
câmera, implementação, 100-105
exercícios, 105-107
otimizando com delimitação caixas, 200–203
ver transformações, definindo, 97-100
objeto do mundo, construção, 92-97
schlick (), 161-165
Schlick, Christophe, 161
Produto Schur, 17
sombra_hit ()
múltiplas fontes de luz, 96
refletância, 142 , 145 , 148 , 161 , 163
refração, 158-160
renderizando sombras, 113–116
padrão listrado, 132
construção do mundo, 95-97
sombreamento
cilindros, 180
exercícios, 89-90
Modelo de reflexão Phong, 83-89
reflexões, computação, 142 , 145-149
refração, 158-160
vetores para, 75
construção do mundo, 93-97
raios de sombra
definiram, 111
renderizando sombras, 113–116
testando sombras, 111–113
sombras, 109-116
exercícios, 116 , 166 , 239
indicando que o ponto está em sombra, 110
optando por sair, 166 , 237
sobreposição, 96
atuação, 96
evitando a auto-sombra ing, 115
Renderização, 113-116
suave, 239
testando, 111-113
formas, *Veja também* construtivo geometria sólida (CSG); cubos; cilindros; polígonos; esferas; triângulos
adicionando aos grupos, 194
atributo filho, 195
funções comuns, 118

formas compostas, exercício cise, 203-205
formas compostas, ideias pra, 205
agrupamento, 193-205
ideias para, 191 , 205
atributo pai, 194-195
refatoração da esfera exemplo, 117-121
renderizando a imagem da capa, 249-253
tosquia, 51-53 , 70 , 74
brilho
Modelo de reflexão Phong, 84-89
reflexão e transparência cy, 165
exercícios de silhueta, 70-74 , 89-90
Ruído simplex, 140
seno, rotação, 47
enviesamento, *Vejo* tosquia
barra (/), vértice de delimitação em formação, 225
suavização, triângulos, 209 , 218-225
Lei de Snell, 157
destaques especulares, 84 , 165
reflexão especular, 84-89
esferas
padrão verificador, 138
computação normal, 77-82
deformando, 70 , 74
exercícios, 70-74 , 89-90 , 105-107 , 124 , 203-205
vítreo, criando, 151
vítreo, refletância, 161-165
vítreo, refração, 151–154
grupos, 203-205
cruzamentos, 59-66 , 70-74
tornando transformável, 69
cruzamentos faltando, 60 , 62
origem, 59 , 80
exercícios de aviões, 124
raios originados atrás, 61
raios originados em, 61
planos de refatoração de, 117-121
girando, 68
escala, 67 , 70 , 74
tosquia, 70 , 74
exercícios de silhueta, 70-74 , 89-90
mapas de textura, 244
transformando com raios, 66-70
tradução, 70
esferas unitárias, 59 , 66 , 79
construção do mundo, 92
mapeamento esférico, 244
exercício de holofotes, 240
padrões listrados, 128-132
submatrizes
definiram, 34
determinantes, achados, 34-38
extraindo, 35
inversão de matrizes, 34-42
multiplicando pelo inverso matriz de transposição, 82
subtração
cores, 16
tuplas, 6
superamostragem, 243
superfície normal, *Vejo* normais


## T


t para cruzamentos
cerca de, 58
ignorando o negativo, 65
cubos que se cruzam com raios, 170 , 172
máximo, 170 , 172
mínimo, 170 , 172
valor de rastreamento, 63
tangentes, 60 , 179
rescisão, teste bem-sucedido ful, 147
estilo teste primeiro, xvii
texturas
Formas CSG, 227
mapas de textura, 244-245
vértice de textura, 225
exercício de toro, 246
reflexão interna total, 150 , 156 , 159 , 161
transformar atributo, 101
transformações, 43-56 , *Vejo* *Além disso* reflexão; dimensionamento; tradução
vantagens, 44
caixas delimitadoras, 168 , 202
encadeamento, 53–54
crianças, 197
convertendo pontos em objeto espaço, 131
convertendo pontos em pat- espaço tern, 132
exercícios, 54-56
APIs fluentes, 55
grupos, 194 , 197
normais, 79–82
ordem, 54 , 106 , 204
padrões, 130-132
rotação, 47-51 , 54-56
formas e identidade matrix, 118
tosquia, 51-53 , 70 , 74
esferas com raios, 66-70
visualizar, 97-105
espaço mundial vs. objeto espaço, 68 , 80
tradução
definiram, 44
ignorando, 45
matriz de transposição inversa, 82
com matrizes, 44-46
multiplicando por vetores, 45
esferas, 70
com vetores, 44
transparência
adicionando, 149-165
Formas CSG, 237
padrões, 151
Efeito Fresnel, 160-165
pontas, 165
transparência atributo, 150
transposição
exercícios, 42
matrizes, 32 , 42
triângulos, 207-226
computação normal, 208
criando, 208 , 221
triangulação de leque, 215
agrupamento, 211 , 213 , 215-216
cruzamentos, 208-212 , 220-225
malhas, 213
suave, 209 , 218-225
Arquivos Wavefront OBJ para Dados 3D, 212-218 , 223
de verdade vetor, 99
cilindros truncados, 177–178 , 181-184
tabelas de verdade
diferença, 233
se cruzam, 232
União, 231
- tupla, negando tuplas com, 7

tuplas, *Veja também* pontos; vetoros
Adição, 5
evitando dados complexos tipos, 4
cores como, 15
criando, 4
definiram, 2
divisão, 7
exercícios, 12
magnitude, computação, 8
multiplicação, 7 , 31
multiplicando matrizes, 30
negando, 7
normalização, 10
operações, 5-8
rotação, 47-51
escala, 46
tosquia, 51-53
subtração, 6
entendimento, 2-5
TurboSquid, 225

## você

você, alisamento com normal em terpolação, 220-225
under_point, 150 , 154
sindicatos
CSG, 228 , 230-236
vs. grupos, 228
cones unitários, 191
esferas unitárias
desvantagens, 66
normalização, 79
raio, 59
vetores unitários, 9
Unidade, 3
acima vetor, 99
Mapeamento UV, 138

## V

v
Arquivos OBJ, 212-213 , 223
alisamento com normal interpolação, 220-225
valores, parâmetro padrão, 148
variáveis, nomes, xx
vetores, *Veja também* vetor do olho; normais; tuplas
convertendo pontos para, 120
criando, 4
produto cruzado, computing, 11
produto escalar, computação, 10
Beira, 208
exercícios, 12
claro, 76
magnitude, computação, 8
multiplicação da tradução matrizes por, 45
normalização, 9
operações, 8-12
reflexão, Phong reflec- modelo de ção, 87-89
reflexão, computação, 82-83 , 142
reflexão, definida, 76
rotação, 47-51
escala, 46
sombreamento, 75
tosquia, 51-53
tradução com, 44
tradução, ignorando, 45
tuplas, adicionando, 6
tuplas, negando, 7
tuplas, subtração, 6
tuplas, compreensão, 3
unidade, 9
ver transformações, 99
vértices
delimitando com a frente golpear (/), 225
índice, 214 , 225
Arquivos OBJ, 212-213 , 223
triângulos lisos, 221
textura, 225
ver transformações
câmera, implementação, 100-105
criando, 97-100
definiram, 97
exercícios de projétil virtual, 13 , 22–24
vn, triângulos suaves em OBJ arquivos, 223
vsize, 100 , 102

## C

C componente
produto escalar, 11
ignorando tradução com, 45
transformando normais, 82
tuplas, adicionando, 6
tuplas, subtração, 6
usando, 3
Arquivos de materiais Wavefront MTL, 227
Arquivos Wavefront OBJ, 212-218 , 223 , 226
Traçado de raio branco, definido, xvi , *Veja também* raios
Whitted, Turner, xvi
Sistemas Windows, visualização Arquivos PPM, 20
mundo
prédio, 92-97
transformação da câmera vs. mundo, 103
câmera, implementação, 100-105
convertendo coordenação mundial nates para as coordenadas da tela, 22
convertendo o espaço do mundo em espaço do objeto, 68 , 80 , 131 , 197-199
rotina de cruzamentos, 92
orientação para a câmera, 101
orientação, padrão, 98
origem, 59 , 80
padrões e espaço mundial, 131
ver transformações, 97-100

## X

x eixo
limite alinhado ao eixo caixas (AABB), 167 , 171
raios que se cruzam com cubos, 171
rotação, 47
x coordenadas
convertendo para inteiros, 22
tosquia, 51
x parâmetro, como baseado em 0, 19
xs, refração, 152
xz, implementando um avião, 122-124

## Y

y eixo
Modelos 3D, 226
limite alinhado ao eixo caixas (AABB), 167 , 171
raios que se cruzam com cubos, 171
rotação, 47 , 49
trocando com z eixo, 226
y coordenadas
convertendo para inteiros, 22
lançando, 73
tosquia, 51
cilindros truncados, 181-184

y parâmetro, como baseado em 0, 19
Formato YAML, 249

## Z

z eixo
Modelos 3D, 226
limite alinhado ao eixo
caixas (AABB), 167 , 171
raios que se cruzam com
cubos, 171
rotação, 47 , 50
trocando com y eixo, 226

# Obrigado!

Como você gostou deste livro? Por favor nos informe. Reserve um momento e envie um e-mail para support@pragprog.com com seus comentários. Conte-nos sua história e você pode ganhar e-books grátis. Use a linha de assunto “Comentários sobre o livro”.

Pronto para seu próximo grande livro Pragmatic Bookshelf? Venha para https://pragprog.com e use o código de cupom BUYANOTHER2019 para economizar 30% em seu próximo e-book.

Nulo onde for proibido, restrito ou indesejado. Não use e-books perto da água. Se a erupção persistir, consulte um médico. Não se aplica a *O Programador Pragmático* ebook porque é mais antigo do que a própria estante pragmática. Os efeitos colaterais podem incluir maior conhecimento e habilidade, maior comercialização e profunda satisfação. Aumente a dosagem regularmente.

E obrigado por seu apoio contínuo, Andy Hunt, editor

Pragmatic Bookshelf

ECONOMIZE 30%!

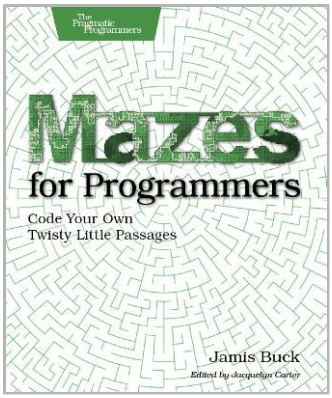
Use o código do cupom

**BUYANOTHER2019**

# A alegria dos labirintos e da matemática

Redescubra a alegria e a estranheza fascinante dos labirintos e da matemática pura.

# Labirintos para programadores




Um livro sobre labirintos? Seriamente?

Sim!

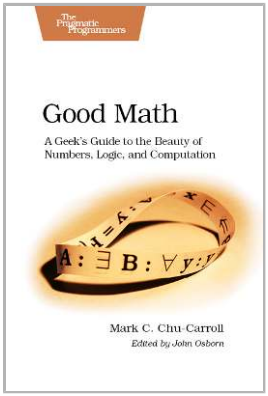
Não porque você passe o dia criando labirintos, ou porque goste particularmente de resolver labirintos.

Mas porque é divertido. Lembra quando a programação costumava ser divertida? Este livro o leva de volta aos dias em que você estava começando a programar e queria fazer seu código fazer coisas, desenhar coisas e resolver quebra-cabeças. É divertido porque permite que você explore e amplie seu código e lembra como é apenas pensar.

Às vezes, parece que você vive sua vida em um labirinto de pequenas passagens sinuosas, todas iguais. Agora você pode codificar sua saída.

Jamis Buck
(286 páginas) ISBN: 9781680500554. $ 38
**https://pragprog.com/book/jbmaze**

# Boa matemática



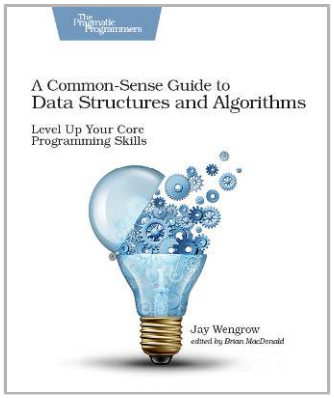

A matemática é linda - e pode ser divertida e empolgante, além de prática. *Boa matemática* é o seu guia para alguns dos tópicos mais intrigantes de dois mil anos de matemática: de frações egípcias a máquinas de Turing; do real significado dos números às árvores de prova, simetria de grupo e computação mecânica. Se você já se perguntou o que está além das provas que você lutou para concluir em geometria do ensino médio, ou o que limita as capacidades do computador em sua mesa, este é o livro para você.

Mark C. Chu-Carroll
(282 páginas) ISBN: 9781937785338. $ 34
**https://pragprog.com/book/mcmath**

# Upar

De estruturas de dados a arquitetura e design, temos o que você precisa.

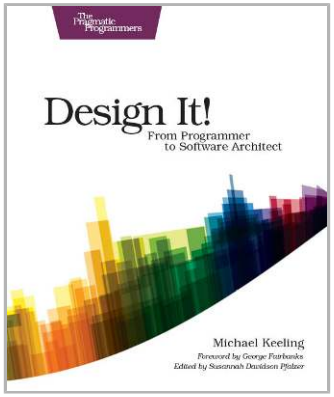
## ACommon-SenseGuide para Estruturas de Dados e Algoritmos




Se você viu algoritmos pela última vez em um curso universitário ou em uma entrevista de emprego, está perdendo o que eles podem fazer pelo seu código. Aprenda diferentes técnicas de classificação e pesquisa e quando usar cada uma. Descubra como usar a recursão de forma eficaz. Descubra estruturas para aplicações especializadas, como árvores e gráficos. Use a notação Big O para decidir quais algoritmos são os melhores para seu ambiente de produção. Os iniciantes aprenderão como usar essas técnicas desde o início, e os desenvolvedores experientes redescobrirão abordagens que podem ter esquecido.

Jay Wengrow
(220 páginas) ISBN: 9781680502442. $ 45,95
**https://pragprog.com/book/jwdsal**

## Design It!




Não faça engenharia por coincidência - projete como você quer! Fundamentado em fundamentos e repleto de métodos práticos de design, esta é a introdução perfeita à arquitetura de software para programadores que estão prontos para desenvolver suas habilidades de design. Faça as perguntas certas às partes interessadas certas, explore as opções de design, compartilhe suas decisões de design e facilite workshops colaborativos que sejam rápidos, eficazes e divertidos. Torne-se um programador, líder e designer melhor. Use suas novas habilidades para liderar sua equipe na implementação de software com os recursos certos - e desenvolver software incrível!

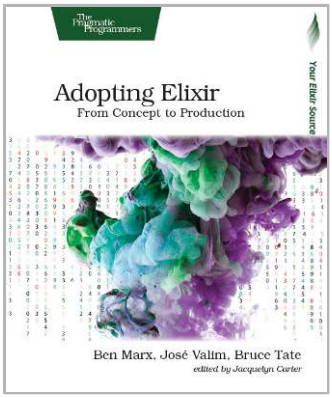
Michael Keeling
(358 páginas) ISBN: 9781680502091. $ 41,95
**https://pragprog.com/book/mkdsa**

## Aprenda por quê, então aprenda como

Ajude a introduzir o Elixir em sua organização onde faz mais sentido e aprenda o Elixir na web com a estrutura Phoenix.

## Adotando Elixir

Adoção é mais do que programação. Elixir é uma nova linguagem empolgante, mas para obter seu aplicativo com sucesso do início ao fim, você precisará saber mais do que apenas o idioma. Você precisa dos estudos de caso e estratégias deste livro. Aprenda as melhores práticas para toda a vida de seu aplicativo, desde o design e formação de equipes, até o gerenciamento das partes interessadas, até a implantação e monitoramento. Vá além da sintaxe e das ferramentas para aprender as técnicas necessárias para desenvolver seu aplicativo Elixir, do conceito à produção.

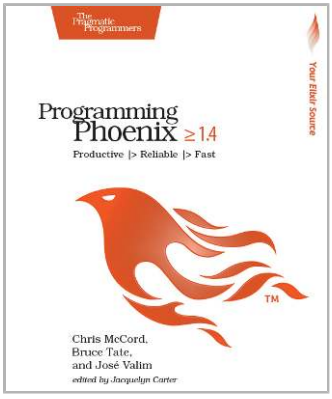



Ben Marx, José Valim, Bruce Tate (242 páginas) ISBN: 9781680502527. $ 42,95

**https://pragprog.com/book/tvmelixir**

## Programando Phoenix 1.4

Não aceite o compromisso entre o rápido e o bonito: você pode ter tudo. O criador do Phoenix, Chris McCord, o criador do Elixir, José Valim, e o premiado autor Bruce Tate, orientam você na construção de um aplicativo rápido e confiável. A cada etapa, você aprenderá com os criadores do Phoenix não apenas o que fazer, mas por quê. Repleto de insights e completamente atualizado para o Phoenix 1.4, este guia definitivo será seu companheiro constante em sua jornada de novato a especialista em Phoenix, conforme você constrói a próxima geração de aplicativos da web.




Chris McCord, Bruce Tate e José Valim (325 páginas) ISBN: 9781680502268. $ 45,95

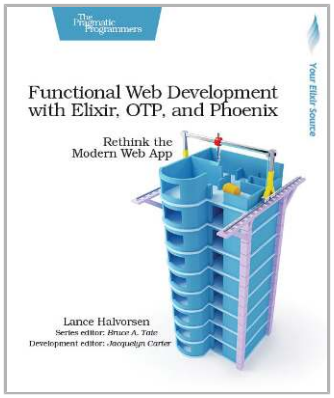
**https://pragprog.com/book/phoenix14**

## Mergulhe profundamente no OTP e no absinto

Junte tudo isso com Elixir, OTP e Phoenix. Mergulhe no GraphQL para obter melhores APIs no Elixir. Está tudo aqui.

## FunctionalWebDevelopmentwithElixir, OTP ePhoenix

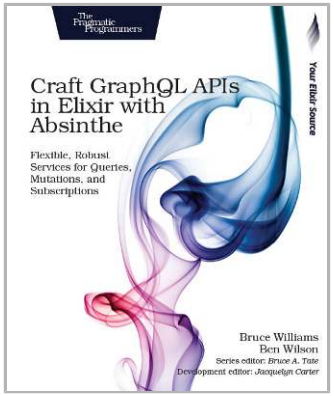
Elixir e Phoenix estão gerando uma grande empolgação como uma plataforma imbatível para a construção de aplicativos da web modernos. Por décadas, o OTP ajudou os desenvolvedores a criar aplicativos incrivelmente robustos e escaláveis com tempo de atividade incomparável. Aproveite ao máximo ao criar um aplicativo da web com estado com Elixir, OTP e Phoenix. Entidades de domínio do modelo sem um ORM ou um banco de dados. Gerencie o estado do servidor e mantenha seu código limpo com comportamentos OTP. Camada em uma interface da web do Phoenix sem acoplá-la à lógica de negócios. Abra as portas para novas técnicas poderosas que farão você pensar sobre o desenvolvimento da Web de maneiras fundamentalmente novas.




Lance Halvorsen
(218 páginas) ISBN: 9781680502435. $ 45,95
**https://pragprog.com/book/lhelph**

## Crie APIs GraphQL em Elixir com Absinthe

Seu domínio é rico e interconectado, e sua API também deve ser. Atualize sua API da web para GraphQL, aproveitando suas consultas flexíveis para capacitar seus usuários e sua estrutura declarativa para simplificar seu código. Absinthe é o kit de ferramentas GraphQL para Elixir, uma linguagem de programação funcional projetada para permitir a simultaneidade massiva sobre arquiteturas de aplicativos robustas. Escrito pelos criadores do absinto, este livro o ajudará a aproveitar ao máximo essas duas tecnologias inovadoras. Crie suas próprias APIs flexíveis e de alto desempenho usando orientação passo a passo e conselhos de especialistas que você não encontrará em nenhum outro lugar.

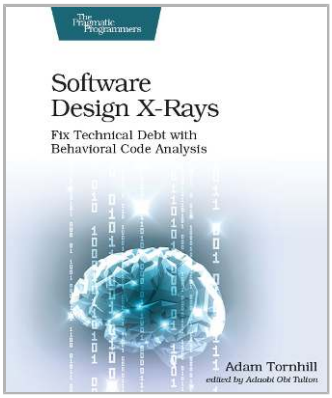



Bruce Williams e Ben Wilson (302 páginas) ISBN: 9781680502558.
$ 47,95
**https://pragprog.com/book/wwgraphql**

## Corrija seus problemas ocultos

Do débito técnico à implantação no mundo muito real e muito confuso, temos as ferramentas de que você precisa para corrigir os problemas ocultos antes que se tornem desastres.

## Raios-X de design de software




Você está trabalhando em uma base de código em que estouro de custos, marchas mortais e lutas heróicas com monstros de código legado são a norma? Enfrente esses adversários com novas maneiras de identificar e priorizar débitos técnicos, com base em dados comportamentais de como os desenvolvedores trabalham com o código. E isso é só o começo. Como um bom código envolve design social, bem como design técnico, você pode encontrar dependências surpreendentes entre as pessoas e o código para resolver gargalos de coordenação entre as equipes. O melhor de tudo é que as técnicas se baseiam em dados comportamentais que você já possui: seu sistema de controle de versão. Junte-se à luta por um código melhor!

Adam Tornhill
(274 páginas) ISBN: 9781680502725. $ 45,95
**https://pragprog.com/book/atevol**

## Solte! Segunda edição

Uma única falha dramática de software pode custar milhões de dólares a uma empresa - mas pode ser evitada com mudanças simples no design e na arquitetura. Esta nova edição do padrão de mercado mais vendido mostra como criar sistemas que funcionam por mais tempo, com menos falhas e se recuperam melhor quando coisas ruins acontecem. A nova cobertura inclui DevOps, microsserviços e arquitetura nativa da nuvem. Os antipadrões de estabilidade cresceram para incluir problemas sistêmicos em sistemas de grande escala. Este é um guia pragmático obrigatório para a engenharia de sistemas de produção.

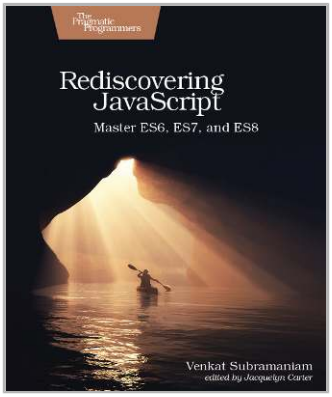
Michael Nygard
(376 páginas) ISBN: 9781680502398. $ 47,95
**https://pragprog.com/book/mnee2**

# JavaScript e mais JavaScript

JavaScript está de volta e melhor do que nunca. Redescubra os recursos e as práticas recomendadas mais recentes para essa linguagem onipresente.

## Redescobrindo JavaScript

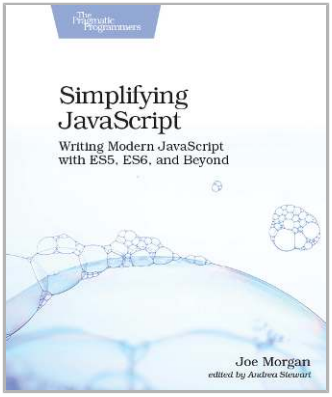
JavaScript não é mais para ser temido ou odiado - a linguagem mais popular e onipresente do mundo evoluiu para uma linguagem respeitável. Esteja você escrevendo aplicativos front-end ou código do lado do servidor, os recursos fenomenais do ES6 e além - como o operador resto, geradores, desestruturação, literais de objeto, funções de seta, classes modernas, promessas, assíncrono e recursos de metaprogramação - o levarão animado e ansioso para programar com JavaScript. Você encontrou o livro certo para começar rapidamente e mergulhar fundo na essência do JavaScript moderno. Aprenda dicas práticas para aplicar as partes elegantes da linguagem e as pegadinhas a evitar.




Venkat Subramaniam
(286 páginas) ISBN: 9781680505467. $ 45,95
**https://pragprog.com/book/ves6**

## Simplificando o JavaScript

O melhor JavaScript moderno é simples, legível e previsível. Aprenda a escrever JavaScript moderno não memorizando uma lista de novas sintaxes, mas com exemplos práticos de como as mudanças de sintaxe podem tornar o código mais expressivo. Começando com declarações de variáveis que comunicam a intenção claramente, veja como os princípios modernos podem melhorar todas as partes do código. Incorpore ideias com funções curried, métodos de array, classes e muito mais para criar código que faz mais com menos, enquanto produz menos bugs.




Joe Morgan
(282 páginas) ISBN: 9781680502886. $ 47,95
**https://pragprog.com/book/es6tips**

# A estante pragmática

O Pragmatic Bookshelf apresenta livros escritos por desenvolvedores para desenvolvedores. Os títulos continuam o conhecido estilo de Programador Pragmático e continuam a receber prêmios e críticas positivas. Conforme o desenvolvimento fica cada vez mais difícil, os Pragmatic Programmers estarão lá com mais títulos e produtos para ajudá-lo a ficar no topo do seu jogo.

# Visite-nos online

**Página inicial deste livro**

**https://pragprog.com/book/jbtracer**

Código-fonte deste livro, errata e outros recursos. Venha nos dar feedback também!

**Manter atualizado**

**https://pragprog.com**

Junte-se à nossa lista de distribuição de anúncios (baixo volume) ou siga-nos no twitter @pragprog para novos títulos, vendas, cupons, dicas e muito mais.

**Novo e notável**

**https://pragprog.com/news**

Confira os últimos desenvolvimentos pragmáticos, novos títulos e outras ofertas.

# Compre o livro

Se você gostou deste e-book, talvez queira uma cópia em papel do livro. Está disponível para compra em nossa loja: **https://pragprog.com/book/jbtracer**

# Entre em contato conosco


Pedidos online: **https://pragprog.com/catalog**

Atendimento ao Cliente: **support@pragprog.com**

Direitos Internacionais: **traduções@pragprog.com**

Uso acadêmico: **Acadêmico@pragprog.com**

Escreva para nós: **http://write-for-us.pragprog.com**

Ou chamar: + 1 800-699-7764