# 4
# Arquitetura do sistema

## 4.1
## O ciclo de amostragem do filme

O nosso ponto de partida para a descrição do sistema é a equação de medição simplificada (2-13). Para o valor de cada pixel $p$, temos uma resposta

$$S_p = \int_{\mathcal{P}} h_p(x,y)L(x,y)dxdy,$$

onde $\mathcal{P}$ é o plano da imagem, $L(x,y)$ é a radiância passando pelo centro da abertura e consideramos que os filtros $h_p$ são cópias transladadas do mesmo filtro $h$, que consideramos ter um suporte finito retangular.

Como esperado, o valor de cada pixel é avaliado por integração de Monte Carlo e, para estimar o valor de cada pixel, nossa escolha foi gerar amostras uniformemente sobre o suporte do filtro de cada pixel. Como os suportes dos filtros de diferentes pixels normalmente se sobrepõem, podemos usar as amostras de um pixel para melhorar a estimativa para o valor de outro. Como as amostras são geradas uniformemente em cada pixel, podemos considerar que estamos simplesmente gerando amostras uniformemente sobre a região da imagem, aumentada com uma borda correspondendo ao suporte dos filtros dos pixels na fronteira da imagem. Para simplificar a geração das amostras sobre toda a região da imagem, as amostras são geradas seqüencialmente sobre a "área" de cada pixel da imagem aumentada.

A classe `Sampler` é reponsável por gerar amostras em cada pixel. `Sampler` é, na verdade, apenas uma interface e as classes que a implementam devem ser capazes de gerar um pacote de $n$ amostras distribuídas uniformemente sobre o quadrado unitário. Mais tarde, essas amostras são apropriadamente transformadas.

Com isso, o valor da resposta de cada pixel $p$ é estimado por

$$\hat{S_p} = \frac{A}{n_p}\sum_{i=1}^{n_p} h_p(x_i,y_i)L(x_i,y_i),$$

onde $n_p$ é o número total de amostras $(x_i, y_i)$ que encontraram-se sobre o suporte do filtro do pixel $p$ e $A$ é a área do suporte do filtro. Embora ligeiramente tendencioso, o valor de cada pixel geralmente é melhor estimado por

$$\hat{S}_p = \frac{\sum_{i=1}^{n_p} h_p(x_i, y_i) L(x_i, y_i)}{\sum_{i=1}^{n_p} h_p(x_i, y_i)}, \tag{4-1}$$

que usamos em nossa implementação.

A classe `Sampler` é reponsável por garantir que sejam geradas boas distribuições sobre o quadrado unitário. Uma maneira de fazer isso, por exemplo, é, ao invés de gerar cada amostra uniformemente em todo o quadrado, estratificar a geração das $n$ amostras dividindo o quadrado em um grid de $\sqrt{n} \times \sqrt{n}$ quadradinhos menores e tomando uma amostra uniformemente sobre cada quadradinho. Embora não tenha sido mencionado explicitamente, isso é exatamente o que vínhamos fazendo até agora sobre toda a imagem. Imagine se estivéssemos gerando uniformemente amostras sobre toda a região da imagem. As distribuições resultantes provavelmente seriam bem piores que as que estamos obtendo, ou seja, provavelmente se aglomerariam em alguns lugares e poderíamos não ter nenhuma amostra gerada em grandes regiões. Se estivermos usando um `Sampler` que gera amostras dessa forma em uma imagem com $N$ pixels, o resultado geral é que estamos estratificando em quadrados menores a distribuição das amostras sobre a imagem como um todo.

Esse ciclo de geração e avaliação de amostras é controlado pela classe `Imager` (figura 4.1). Para cada pixel da imagem aumentada, requisitamos um pacote com $n$ amostras do `Sampler` (passo 1). Para cada amostra nesse pacote, calculamos o raio com origem na amostra indo na direção do furo da câmera (passo 2). Em seguida, requisitamos que o `Tracer` nos retorne a radiância $L$ chegando na amostra na direção contrária à do raio (passo 3). Por último, com o valor estimado por `Tracer` para a radiância, atualizamos o valor de cada pixel cujo suporte inclui o ponto amostrado (passos 4 e 5). Para evitarmos uma divisão cada vez que (4-1) é atualizado, basta, evidentemente, irmos acumulando a soma do denominador e, no final, fazer uma passada para calcular a divisão em cada pixel.

Note que não precisávamos necessariamente ter feito a aproximação para câmeras de furo. Poderíamos igualmente ter partido da equação de medição (2-10), onde teríamos que substituir a radiância $L$ pela irradiância $E$ na resposta do sensor que estamos considerando. Com isso, as amostras do `Sampler` não seriam apenas a posição da amostra sobre o plano da imagem, mas também teriam uma posição sobre a abertura da câmera. Gerando raios primários dessa forma, em uma câmera onde a aproximação para câmeras de furo não fosse válida, teríamos uma imagem completamente desfocada.

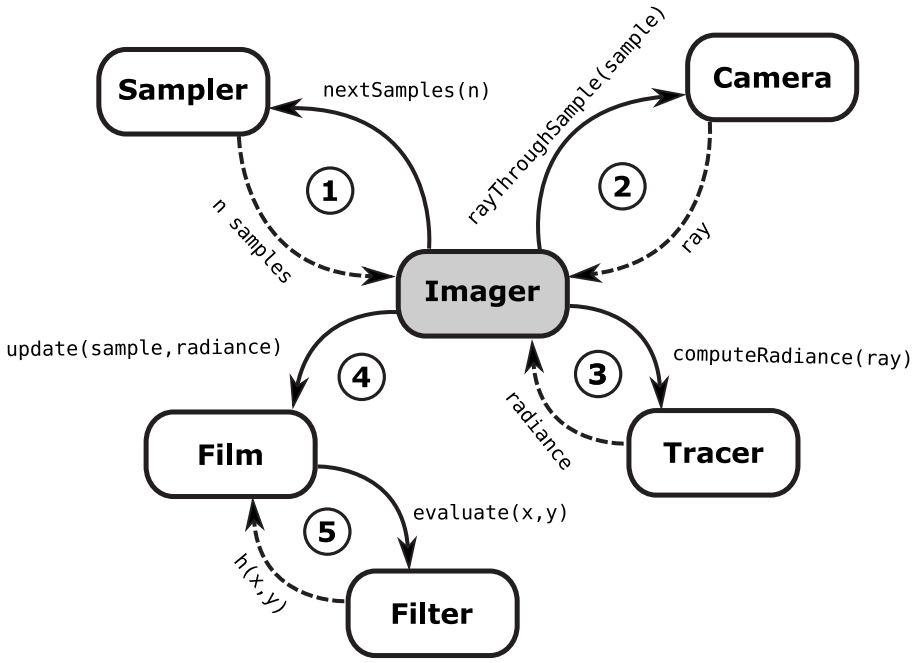


Figura 4.1: O ciclo de amostragem do filme

Usando, por exemplo, uma aproximação para lentes delgadas na abertura, os raios poderiam ser gerados pela câmera já levando em consideração a refração do raio pela lente e produzir, como conseqüência, o efeito da profundidade de campo. Um modelo para câmeras com um sistema completo de lentes foi proposto em (Kolb et al. 95). De maneira semelhante, podemos também incluir uma amostra no tempo para levar em consideração a exposição, resultando no efeito de *motion blur*, caso a cena seja dinâmica.

Mas note que, quando aumentamos a dimensão do espaço, o problema de garantir uma boa distribuição das amostras, que para um pixel era um problema 2D, virou um problema 5D, e a estratificação exigiria um número muito grande de amostras por pixel. Mais ainda, dependendo da cena, poderia ter o efeito de reduzir a qualidade da distribuição em alguma das dimensões caso o mesmo número de amostras fosse usado para estratificar somente uma dimensão, por exemplo. Não vamos entrar nos detalhes de como tentar resolver esse problema. Mencionamos isso para fazer a observação de que, como descrito no capítulo anterior, `Tracer` é um estimador que é função de um número grande de variáveis aleatórias. Então, a geração de amostras bem distribuídas já era um problema para nós. Certamente, a geração das amostras de maneira apropriada para esses espaços de dimensão alta é um problema que certamente influencia o ruído da imagem final que estamos calculando. No entanto, assumimos que só tomamos esse cuidado na amostra sobre a imagem

através de `Sampler`, e as demais dimensões são amostradas uniformemente. Uma investigação futura para o sistema é estudar estratégias para a geração de amostras com distribuições melhores, levando em consideração pelo menos algumas dimensões mais altas ao invés de somente na primeira amostra sobre a imagem.

Note também que a geometria da formação de imagens pela câmera pode assumir formas diferentes da câmera de furo. Podemos usar a câmera para calcular uma projeção ortogonal (que pode ser vista como uma câmera de furo na qual o plano de projeção está tão longe da abertura que podemos considerar os raios partindo da amostra na direção do furo como paralelos), cilíndrica ou esférica, por exemplo. Poderíamos usar o resultado de uma projeção esférica para fazer *environment mapping*, por exemplo. A única coisa que exigimos de uma câmera é saber mapear um par de coordenadas $(x, y)$ parametrizado sobre alguma superfície em um raio representando a direção da radiância que contribui para a resposta sobre $(x, y)$.

Por último, a classe `Film` é responsável por armazenar as respostas estimadas em cada pixel. Além da imagem, guardamos também no filme a profundidade de cada pixel e seu valor alfa, que mede a porcentagem da "área" de cada pixel que é coberta pela projeção de alguma geometria da cena. Quando terminarmos de gerar todas as amostras, a imagem poderá ser salva em HDR ou quantizada em um formato convencional, após a aplicação de um operador de *tone mapping*.

## 4.2 O núcleo da geometria

Assim como em algoritmos clássicos de traçado de raios, gastamos a maior parte do tempo determinando a interseção entre um raio e a geometria da cena, ou calculando a função de visibilidade $V(\mathbf{x}, \mathbf{y})$. Como o nosso problema é exatamente o mesmo, vamos discutir brevemente a nossa escolha. Para mais detalhes, consulte (Glassner 89).

Como em todas as técnicas baseadas em traçado de raios, a geometria é tratada como uma caixa preta. Acrescentamos à coleção de objetos da cena primitivas como esferas, cilindros, malhas poligonais, etc. devidamente posicionadas segundo uma transformação e essa coleção deve ser capaz de nos responder com eficiência as duas operações de traçado de raios. A primeira é o teste de interseção entre um raio e a geometria, isto é, simplesmente nos indicar se algum objeto intercepta um dado raio. A segunda, caso algum objeto intercepte o raio, deve também nos retornar uma amostra pontual da geometria local desse objeto.

Para garantir que essas duas operações possam ser feitas de maneira eficiente, dentro dessa caixa preta que consideramos ser a coleção dos objetos da cena, os objetos são organizados em estruturas de aceleração como *grids* uniformes, hierarquias de volumes envolventes, *octrees*, etc... E, como essas estruturas organizam os objetos levando em consideração a sua distribuição espacial, temos que ser capazes de tratar de maneira uniforme a distribuição espacial de todas as primitivas geométricas. Isso normalmente é feito exigindo-se que uma primitiva retorne uma geometria aproximada unificada, como uma esfera ou uma caixa alinhada com os eixos, por exemplo, para representar a sua distribuição espacial, isto é, exige-se das primitivas que retornem uma certa geometria que envolva a sua extensão. Embora as primitivas possam assumir as mais variadas formas, do ponto de vista da estrutura de aceleração suas distribuições espaciais são todas descritas pela mesma geometria. Mas note que, como as estruturas de aceleração são coleções de primitivas, podemos dizer que a sua extensão espacial é a extensão da união das primitivas que contém, e, portanto, exigindo que uma estrutura de aceleração retorne também um volume envolvente descrevendo a sua distribuição espacial, podemos combinar livremente diferentes estruturas de aceleração para otimizar o tempo gasto nas operações de interseção com raios em uma cena.

Como observado em (Kirk & Arvo 88), sob esse ponto de vista as estruturas de aceleração compartilham a mesma interface das primitivas geométricas, ou seja, exigimos tanto de uma estrutura de aceleração quanto de uma primitiva geométrica que sejam capazes de nos retornar um determinado volume envolvente e de fazer o teste e a determinação de interseção com um raio.

Note que as próprias primitivas geométricas (esfera, malha, etc...) também só precisam implementar a interface, o que deixa a liberdade para otimizarmos à vontade as operações de interseção, incluindo o uso de estruturas de aceleração. Por exemplo, na nossa implementação de malhas de triângulos, usamos uma hierarquia de volumes envolventes para acelerar a interseção e permitir que usássemos malhas que representassem geometrias suficientemente complexas.

Em relação à transformação, consideramos a descrição hierárquica necessária somente para a descrição da cena. Então, na API e na linguagem de descrição da cena (que é apenas uma seqüência de comandos à la *RenderMan* (Renderman 99), exemplificada no apêndice B), temos uma pilha de transformações, que é "achatada" dentro da representação interna da cena. A transformação está associada a uma primitiva (`Primitive`), que referencia uma primitiva geométrica (`Shape`). Isso permite que criemos várias instância de uma mesma primitiva geométrica com diferentes transformações e materiais.

## 4.3 Estratégias para calcular a radiância

Na primeira seção consideramos como dada a classe `Tracer`, que nos retornava a radiância $L(\mathbf{x} \leftarrow C)$. Podemos usar aqui a estratégia que quisermos, basta apenas retornar uma radiância. Por exemplo, podemos considerar somente a iluminação direta amostrando apenas as fontes de luz na primeira interseção do raio com a cena, ou, o que nos interessa mais, a solução completa usando o traçado estocástico de caminhos que apresentamos no capítulo anterior. Outros algoritmos, como mapeamento de fótons e traçado bidirecional de caminhos, também se encaixam perfeitamente aqui.

Embora aqui também se encaixassem soluções usando outros métodos, baseados em radiosidade, por exemplo, as demais classes do sistema foram construídas tendo em mente que teriam que dar suporte para algoritmos de traçado de raios baseados no método de Monte Carlo, a classe de algoritmos que estamos considerando.

Uma classe que implementa um algoritmo específico tem apenas que implementar o método `computeRadiance`. Mais tarde, vamos descrever como o algoritmo de traçado de caminhos descrito no capítulo anterior implementa esse método.

## 4.4 Materiais

As propriedades de reflexão de uma superfície são descritas pela sua BRDF. Todas as BRDFs compartilham uma interface comum extraída das exigências dos algoritmos baseados em integração de Monte Carlo, como o traçado estocástico de caminhos que apresentamos no capítulo anterior. Basicamente, devemos ser capazes de avaliar pontualmente uma BRDF e, se possível, amostrá-la por importância.

Deixamos de fora a inclusão de mapeamento de textura, que não apresenta dificuldades além das usuais. Texturas, como de costume, seriam usadas aqui para parametrizar $f_r$ sobre a superfície. Assumimos que os parâmetros de $f_r$ são constantes ao longo de uma superfície e podemos, portanto, abreviar a notação de uma BRDF para $f_r(\omega_i \leftrightarrow \omega_o)$.

Freqüentemente, os algoritmos dependem que sejamos capazes de separar uma BRDF em uma componente difusa e uma especular, hipótese que é feita com base na própria descrição de grande parte dos modelos para BRDF, que normalmente são separados dessa forma. Essa separação torna conveniente a especificação de vários tipos de BRDFs. Por exemplo, podemos usar o modelo de Cook-Torrance, para a componente especular, combinado com o modelo

de Oren-Nayar, para a componente difusa. Fazemos essa separação apenas no material, onde podemos acessar a soma das duas componentes ou cada uma individualmente. Com isso, em um material, a BRDF é descrita como:

$$f_r(\omega_i \leftrightarrow \omega_o) = f_d(\omega_i \leftrightarrow \omega_o) + f_s(\omega_i \leftrightarrow \omega_o).$$

Por especular, queremos dizer componentes que tenham uma dependência direcional significativa. Não consideramos reflexões perfeitamente especulares, que devem ser tratadas nos algoritmos como casos especiais que não foram considerados.

Mais tarde iremos descrever como acoplamos em uma única BRDF a soma das duas componentes. Mas antes vamos descrever a interface comum a todas as BRDFs.

### 4.4.1 BRDF

As BRDFs são descritas de maneira mais conveniente quando usamos um sistema de coordenadas local à geometria. Quando pedimos a interseção de um raio com a geometria da cena, o núcleo de geometria do sistema nos retorna, no sistema global, o ponto **x** de interseção e uma base ortonormal **uvw** com o vetor **w** na direção da normal da superfície. Com essa informação, fica fácil transformar uma direção no sistema de coordenadas global para o sistema local e vice-versa, já que só envolve uma transformação ortogonal. Então, do ponto de vista dos métodos implementados em uma BRDF, assumimos que a normal da superfície no ponto **x** no sistema local é o vetor unitário na direção do eixo $z$ e os eixos $x$ e $y$ estão, portanto, no plano tangente à superfície. A classe `Material` é reponsável por transformar uma direção do sistema de coordenadas global para o local antes de passá-la para uma `BRDF` e por converter uma direção retornada pela `BRDF` do sistema local para o global.

Em primeiro lugar, devemos ser capazes de avaliar pontualmente uma BRDF, isto é, dadas as direções $\omega_i$ e $\omega_o$, uma `BRDF` deve ser capaz de retornar $f_r(\omega_i \leftrightarrow \omega_o)$. Na maioria dos casos, usamos BRDFs descritas analiticamente, onde avaliar a BRDF envolve apenas calcular uma expressão em função das direções $\omega_i$ e $\omega_o$. Simplificando a linguagem, uma BRDF deve implementar um método do tipo:

```
spectrum brdf::evaluate(vector wi, vector wo);
```

Onde `spectrum` é um tipo que representa alguma grandeza que varia com o comprimento de onda, como a BRDF.

Em segundo lugar, pelo que vimos no capítulo anterior, é importante que sejamos capazes de amostrar por importância uma BRDF. Fixada uma direção $\omega_o$, uma BRDF deve gerar uma amostra $\omega_i$.

```
spectrum brdf::sample(vector wo, vector *wi, float *pdf);
```

Retornando, além da direção amostrada, $f_r(\omega_i \leftrightarrow \omega_o)$ e a função de densidade usada para gerar as amostras avaliada na direção da amostra retornada. Convecionamos que a densidade é medida em relação ao ângulo sólido. Como padrão, amostramos uma BRDF tratando-a como perfeitamente difusa, isto é, se uma BRDF não sobrescrever o método de amostragem, ela é amostrada como se fosse perfeitamente difusa, onde a densidade que usamos é a descrita no capítulo anterior. Caso seja possível amostrá-la segundo uma distribuição melhor, o método pode ser sobrescrito. Note que aqui consideramos que a amostragem por importância inclui o cosseno do ângulo polar, ou seja, se possível, a BRDF deve levar em consideração esse fator.

Também exigimos que uma BRDF implemente um método que retorna a função de densidade que usa para gerar amostras:

```
float brdf::pdf(vector wo, vector wi);
```

### 4.4.2 Acoplamento de componentes difusa e especular

Um material é composto de uma componente difusa e uma especular.

Vamos agora descrever como um material usa uma interface de acoplamento entre duas componentes, chamadas difusa e especular, para gerar amostras segundo uma distribuição combinada:

Para a BRDF total, basta simplesmente somar as avaliações das componentes difusa e especular. Cada componente implementa sua avaliação particular.

Em um material, separamos a BRDF $f_r$ em uma componente especular $f_d$ e uma difusa $f_s$:

$$f_r(\omega_i \leftrightarrow \omega_o) = f_d(\omega_i \leftrightarrow \omega_o) + f_s(\omega_i \leftrightarrow \omega_o).$$

A descrição da BRDF como a soma de duas componentes que foram chamadas de difusa e especular é só uma conveniência para fixar o número de componenentes e traduzir o fato de que grande parte dos materiais são descritos dessa forma. E não há perda de generalidade com isso, basta definir a componente difusa como igual a zero e definir qualquer outra BRDF como especular.

Dadas duas direções `wi` e `wo`, a soma da combinação é simplesmente a soma das BRDFs individuais:

```
spectrum brdf_combine::evaluate(vector wi, vector wo)
{
  return diffuse.evaluate(wi.wo) + specular.evaluate(wi, wo);
}
```

Devemos também ser capazes de amostrar por importância uma direção segundo a BRDF combinada. Assumindo que sabemos amostrar as componentes difusas e especulares individualmente, a BRDF como um todo é amostrada por uma distribuição condicional. Supondo que a componente difusa tem uma importância $\rho_d$ e a especular $\rho_s$, seja $\rho = \rho_d/(\rho_s + \rho_d)$, usando uma variável aleatória $\xi$ distribuída uniformemente entre 0 e 1 para escolher entre uma componente ou outra, estamos amostrando segundo a densidade combinada

$$\begin{aligned} p(\mathbf{x}_i \rightarrow \omega) &= p(\mathbf{x}_i \rightarrow \omega | \xi < \rho) p(\xi < \rho) + p(\mathbf{x}_i \rightarrow \omega | \xi \geq \rho) p(\xi \geq \rho) \\ &= p_d(\mathbf{x}_i \rightarrow \omega)\rho + p_s(\mathbf{x}_i \rightarrow \omega)(1 - \rho) \end{aligned}$$

Com isso, a implementação da combinação fica

```
spectrum brdf::sample(vector wo, vector *wi, float *pdf)
{
  float pdf_d, pdf_s;
  float f;

  if (random(0,1) < rho) {
    f = diffuse.sample(wo, wi, &pdf_d);
    f += specular.evaluate(*wi, wo);
    specular.pdf(wo, wi, &pdf_s);
  } else {
    f = specular.sample(wo, wi, &pdf_s);
    f += diffuse.evaluate(*wi, wo);
    diffuse.pdf(wo, wi, &pdf_d);
  }
  *pdf = rho*pdf_d + (1-rho)*pdf_s;

  return f;
}
```

Por último, a avaliação da densidade usada para gerar uma amostra *wo* específica é simplesmente a combinação das densidades individuais.

```
float brdf::pdf(vector wo, vector wi)
{
  float pdf_d, pdf_s;

  diffuse.pdf(wo, wi, &pdf_d);
  specular.pdf(wo, wi, &pdf_s);

  return rho*pdf_d + (1-rho)*pdf_s;
}
```

## 4.5 Fontes de luz

Transformamos todas as primitivas de uma cena em uma fonte de luz em potencial associando uma `Primitive` a zero ou uma fonte de luz. Se houver a referência, então a primitiva é emissora. Para que sejamos capazes de amostrar diretamente as superfícies que são emissoras, a cena guarda também a coleção das fontes de luz.

Para facilitar a discussão, assumimos que a emissão das fontes de luz é uniforme sobre toda a sua superfície e não tem dependência direcional, isto é, estamos considerando apenas emissores difusos. Com isso, a função $L_e(\mathbf{x} \rightarrow \omega)$ é uma constante sobre toda a superfície de cada fonte de luz.

No capítulo anterior, vimos que devemos saber amostrar uma fonte de luz dado um ponto (e a geometria local) sobre uma superfície da cena para guiar a transição final em um caminho, que irá gerar raios em direção às fontes de luz na cena e corresponder à iluminação atingindo diretamente uma superfície.

Vamos assumir por enquanto que só temos uma fonte de luz na cena. Resta ainda o problema de, dados o ponto e a normal sobre a superfície, gerar amostras sobre as fontes de luz segundo uma densidade que traduza os fatores importantes nessa iluminação direta (desconsiderando a BRDF). Como estamos supondo que as fontes de luz são difusas, o problema se torna puramente geométrico. E, mesmo que as fontes possam ter uma dependência direcional, dificilmente consegue-se gerar distribuições que também levem isso em consideração, ou seja, geralmente supõe-se que os emissores são difusos para a geração de amostras. Com isso, todo o esforço na geração de amostras está concentrado nas primitivas geométricas, como esferas, triângulos, etc...

Com isso, além do termo de visibilidade, que dificilmente consegue-se levar em consideração, temos 3 fatores que devem ser levados em consideração para gerarmos as amostras: a geometria local da fonte, o seu ângulo sólido determinado no hemisfério sobre o ponto iluminado e, ainda, o ângulo sólido

projetado. No primeiro caso, desconsideramos completamente a geometria local do ponto iluminado. Com isso, deixamos de levar em consideração o fator geométrico. Nesse caso, a estratégia mais comum é gerarmos amostras distribuídas uniformemente sobre a superfície. Ao desconsiderarmos a geometria do ponto iluminado, estamos ignorando por completo o fator geométrico da última transição no integrando. Nesse caso, fontes de aumento da variância do estimador incluem, por exemplo, uma fonte de luz muito grande, onde inverso do quadrado da distância entre o ponto iluminado e o amostrado podem variar significativamente, embora estejamos dando a mesma importância a todos eles. Variações na orientação da fonte também são uma fonte de ruído aqui. Para permitir que as geometrias sejam capazes de fazer o melhor possível, incluímos na classe `Shape` um método que amostra sem passar a geometria do ponto iluminado, que é o que a maioria das geometrias implementa, amostrando uniformemente sobre a área, e um passando também a geometria local, que é implementado, por padrão, simplesmente chamando o primeiro método que só leva em consideração a área. Isso significa que uma geometria deve ser capaz de ser amostrada pelo menos por área para ser usada como fonte de luz.

Sabemos então amostrar uma fonte de luz em separado. Para gerar amostras sobre a união das $N_L$ superfícies emissoras, como a radiância é linear em função de $L_e$, podemos separar cada termo da série de Neumann como uma soma de $N_L$ termos correspondendo a cada fonte de luz. Para evitarmos ter que gerar uma amostra sobre cada fonte, podemos usar a técnica de roleta russa para avaliar somente um dos termos. Se cada termo é escolhido segundo uma probabilidade $q_i$, a contribuição da amostra gerada pelo termo escolhido é ponderada por $1/q_i$. Por exemplo, se escolhermos com igual probabilidade de amostrar cada fonte de luz, então multiplicamos por $N_L$ o valor do estimador pela fonte de luz escolhida. Outra escolha para os vários $q_i$s é a fração da potência de cada fonte em relação à potência total de todas as superfícies.

## 4.6 Traçado estocástico de caminhos

Com as classes de suporte que definimos, implementar o algoritmo de traçado estocástico de caminhos descrito no capítulo anterior torna-se relativamente simples.

Como descrito no capítulo anterior, estamos calculando a soma

$$\hat{L} = \hat{L_0} + \hat{L_1} + \cdots + \hat{L_i} + \cdots$$

decidindo recursivamente truncar a série em cada passo segundo uma probabili-

dade $q_i$. Isto é, para amostrar esse estimador recursivo, em cada passo usamos uma variável aleatória $\xi_i$ gerada com distribuição unifome para decidir com probabilidade $q_i$ se truncamos a série. Se $\xi_i < q_i$, então paramos a avaliação e retornamos da recursão. Caso contrário, amostramos o termo $\hat{L}_i$ e amostramos recursivamente os termos seguintes da série. O valor retornado pela chamada recursiva é ponderado por $1/(1-q_i)$ para levar em consideração os termos que poderíamos não ter avaliado, somado com a amostra de $\hat{L}_i$, e o resultado é a amostra retornada. Claramente, não precisamos gerar as amostras recursivamente, já que o peso final de cada termo $\hat{L}_i$ é simplesmente o produto dos fatores $1/(1-q_j)$ acumulados na volta da recursão. O objetivo dessa seção final é decrever em detalhe a geração de amostras para esse estimador segundo essa formulação iterativa.

Na $i$-ésima iteração, acumulamos em uma variável `radiance` a contribuição de um caminho amostrado de comprimento $i$ para a radiância total.

```
spectrum radiance = 0;
spectrum weight = 1;
for (i = 1; ; i++) {
```

Como mencionado, uma vez que precisamos amostrar um caminho de comprimento $i-1$ para avaliar o estimador para o termo $L_{i-1}$ da soma, para construir o caminho para o termo $L_i$ podemos reutilizar o caminho de comprimento $i-1$ que já foi construído. Mas como a transição final do caminho de comprimento $i-1$ foi feita levando-se em consideração a iluminação incidente, não queremos simplesmente reutilizar como prefixo todo o caminho anterior, já que nas transições intermediárias de um caminho estamos levando em consideração somente a amostragem por importância das BRDFs. Então reutilizamos o caminho de comprimento menor até o seu penúltimo vértice ao invés do último.

Uma maneira elegante de fazer isso é usar a invariante de que, no início de cada iteração do `for`, a origem do raio representa o penúltimo vértice do caminho anterior e a direção do raio já fornece a direção para o penúltimo vértice do caminho corrente. Na primeira iteração, o raio é o fornecido como parâmetro para a própria função `computeRadiance`. Garantimos essa invariante simplesmente descartando, no final de cada iteração, o vértice final do novo caminho, localizado sobre uma fonte de luz, e, olhando para a frente, na origem do raio guardamos seu penúltimo vértice e, na direção, uma amostra segundo a BRDF da superfície onde se encontra esse penúltimo ponto.

Assumindo essa invariante, no início do `for` temos somente a direção para o penúltimo vértice do caminho de comprimento $i$, então temos ainda

que determinar a interseção do raio com a geometria da cena para encontrar a sua posição. Caso o raio não intersecte a cena, pela maneira como estamos construindo os caminhos, podemos truncar a série desse ponto em diante.

```
if (!rayIntersect(ray, scene, &intersection))
  break;
```

Em seguida, determinamos se vamos ou não truncar a série depois do termo $i$, inclusive. Tomamos essa decisão baseados em uma informação puramente local. Para simplificar o problema, vamos assumir que as superfícies têm um coeficiente de reflexão $\rho$, que mede a probabilidade de uma partícula ser reemitida ao interagir com o material. Com isso, para decidir se devemos truncar ou não a série, estamos usando a noção de que se um caminho passar por um material pouco refletivo, então, por uma visão local, provavelmente os caminhos que seguem a partir desse ponto também não contribuem muito para a radiância total. Com isso, descartamos caminhos em regiões que consideramos localmente serem menos importantes. Como são os caminhos curtos que contribuem em grande parte para uma imagem, evitamos aumentar a variância nesses caminhos tomando essa decisão somente após um certo número de termos já terem sido avaliados:

```
if (i > minlength) {
  rho = intersection.material.rho;
  if (random(0,1) < rho)
    break;
  weight *= 1/(1-rho);
}
```

Assumindo que ainda não truncamos a série, o próximo passo é amostrar o vértice final do caminho do termo $L_i$ segundo a iluminação incidente. Uma vez amostrado o vértice final, a contribuição total do caminho é dada ponderando-se a emissão no vértice final pelo produto de todos os fatores ao longo do caminho, acumulado na variável `weight`. Para levar em consideração a radiância que atinge diretamente uma fonte de luz, temos ainda que somar a radiância na primeira transição.

```
if (i == 0)
  radiance += intersection.light.Le();    // weight == 1

wo = -ray.direction;
radiance += weight*direct_lighting(intersection, wo);
```

onde `direct_lighting` é uma das técnicas de amostragem das fontes de luz em uma cena descrita na seção anterior.

Para estender o caminho para o termo seguinte na série, amostramos a direção do próximo vértice por importância segundo a BRDF do material:

```
brdf = intersection.material.sample(wo, &wi, &pdf);
weight *= brdf*dot(intersection.normal, wi)/pdf;
```

E podemos garantir a invariante para o caminho seguinte:

```
ray = ray(intersection.p, wi);
```

Ao final da iteração, retornamos a radiância total calculada:

```
}
return radiance;
```

Para clarificar a exposição, omitimos o cálculo da profundidade e do canal alfa.