Capítulo 1

Introdução

Em computação gráfica tradicional, objetos tridimensionais são criados usando representações de superfícies de alto nível, como malhas poligonais ou retalhos NURBS[1] Quando este paradigma de modelagem é usado, as propriedades das superfícies, como cor, rugosidade e reflexividade, são definidas apenas na superfície do modelo, e usadas pelos shaders[2], que podem ser simples, como um modelo de reflexão difusa ou tão complexo quanto uma BRDF[3] mais elaborada. Contudo, o que ocorre nestes casos é que o transporte de luz que é calculado pelo algoritmo é avaliado estritamente nos pontos da superfície do objeto. Logo, estes métodos não são capazes de avaliar as interações de luz que ocorrem no interior do objeto.

Comparada à renderização de superfícies, a renderização de volumes descreve uma gama de técnicas para gerar imagens a partir de dados escalares tridimensionais. Essas técnicas foram motivadas originalmente pela área de visualização científica, em que dados volumétricos são obtidos por medições ou simulações numéricas. Exemplos comuns são dados médicos do interior do corpo humano obtidos por tomografia computadorizada ou resonância magnética. Outros exemplos são dados de dinâmica dos fluidos computacional, dados geológicos e sísmicos, e dados matemáticos abstratos como a distribuição tridimensional da probabilidade de um número aleatório, superfícies implícitas, entre outros.

Com a evolução de técnicas eficientes de renderização, dados volumétricos estão se tornando mais relevantes para aplicações em jogos. Modelos volumétricos são ideais para descrever objetos dispersos, como fluidos, gases, e fenômenos naturais, como nuvens, névoa, fumaça e fogo.

[1]Non-Uniform Rational B-Spline, na sigla em inglês, é um modelo matemático usado para gerar e representar curvas e superfícies, especialmente usada em computação gráfica.

[2]De modo bem simplista, *shaders* são instruções de software usadas nos processadores gráficos para a geração de imagens. Uma discussão mais extensa é apresentada na seção 2.4.

[3]**BRDF**, bidirectional reflectance distribution function, na sigla em inglês, é uma função de quatro dimensões que define como a luz é refletida numa superfície opaca.

O desafio de renderizar volumes está principalmente na avaliação da interação entre a luz e e os elementos naturais, como as nuvens. A luz pode ser absorvida, espalhada, ou emitida por materiais gasosos, e que participam na propagação luminosa. Essa interação precisa ser avaliada em todas as posições do volume 3D preenchido com o gás, tornando a renderização volumétrica uma tarefa cara para processar.

1.1 Definição do Problema

Como discutido na seção anterior, dados volumétricos são essenciais para várias aplicações em computação gráfica, como na produção de efeitos visuais e imagens médicas, incluindo simulação de fluidos, modelagem com superfícies implícitas e a própria renderização de volumes.

Na maioria das aplicações, os dados volumétricos são representados por *grids* 3D regulares e espacialmente uniformes, em parte porque tais representações são simples e convenientes. Grids serão especificados mais claramente na seção 2.2.4, mas trata-se basicamente de uma estrutura de três dimensões composta por elementos de volume - um exemplo uniforme é um cubo de lado igual a 10, composto por pequenos cubos de lado 1. Neste exemplo, os pequenos cubos são os elementos de volume que compõem o grid, o cubo maior.

1.1.1 Abordagem básica à renderização de volumes

O problema da renderização de volumes, tal como tratado neste trabalho, pode ser dividido em diferentes etapas, brevemente descritas a seguir.

Consulta aos dados. Posições no espaço são escolhidas para serem amostradas no volume, isto é, os pontos escolhidos pela aplicação são buscados na estrutura que armazena o volume.

Interpolação. Os pontos escolhidos na etapa anterior normalmente são diferentes dos pontos existentes no volume. Assim, um campo 3D contínuo precisa ser reconstruído a partir do grid discreto para que os valores nos pontos buscados possam ser obtidos.

Shading e Iluminação. O modelo óptico do problema é calculado, e as componentes de absorção e emissão de luz são avaliados. Com os resultados da interação da luz com o volume, a imagem pode ser gerada.

As duas primeiras etapas, de consulta ao volume e cálculos de interpolação são processos estritamente locais em relação aos pontos amostrados. Dessa forma, em um software modular, é de se esperar que as implementações destas etapas sejam independentes da etapa

de Shading e dos cálculos de Iluminação. Essa observação é importante, pois limitará o escopo deste trabalho.

1.1.2 Software-Alvo

O Blender é um pacote de software de computação gráfica 3D, gratuito e de código aberto, usado, por exemplo, na criação de animações e efeitos visuais, modelos tridimensionais impressos, aplicações 3D interativas e jogos digitais. Como outras aplicações semelhantes, ele suporta modelagem 3D, texturização, animação, simulação de fluidos e fumaça, simulação de partículas, renderização, entre outras funcionalidades.

Cycles é o nome dado à engine de renderização recentemente adicionado ao Blender. Trata-se de um renderizador realista (fisicamente plausível), com suporte a multi-threading e a aceleração por GPUs. Como o software existe há pouco tempo, há muitas funcionalidades desejáveis que ainda não foram implementadas, e uma delas é a renderização de volumes.

1.1.3 Estrutura de Dados

OpenVDB é o nome da biblioteca de código aberto, em C++, que possui uma nova estrutura de dados e um conjunto de ferramentas para o armazenamento e manipulação de dados volumétricos esparsos e discretos, em coordenadas tridimensionais. O destaque desta estrutura de dados é o oferecimento de um espaço de índices limitado apenas pela capacidade de memória da máquina, com armazenamento adaptativo dos volumes, e a velocidade de acesso aleatório e sequencial praticamente constante, $O(1)$. Um estudo detalhado é apresentado no capítulo 3.

1.1.4 Definição de escopo

O objetivo deste trabalho é implementar a texturização de volumes no renderizador Cycles do Blender, o que corresponde às duas primeiras etapas descritas na seção 1.1.1, de consulta de dados a uma estrutura volumétrica e realização dos cálculos de interpolação. Como será descrito no capítulo 2.2, essas etapas englobam o uso de uma textura tridimensional.

A proposta é utilizar a biblioteca OpenVDB para armazenamento do volume, e implementar as funcionalidades básicas de geração, consulta e escrita em arquivo dos volumes considerados.

1.2 Motivação

Há vários fatores que motivaram a realização deste trabalho.

1.3 Projeto Pessoal

Com o intuito de melhor qualificar-me para atuar em desenvolvimento de aplicações gráficas, busquei me aproximar da comunidade de usuários do Blender e passei a participar de parte das discussões na lista de desenvolvedores. Com as discussões, entendi um pouco quais eram as necessidades mais relevantes do projeto, e uma estimativa do nível de dificuldade associado a cada um dos requisitos em aberto.
Uma das funcionalidades em aberto naquela ocasião era a renderização de volumes. De posse de algumas referências, pesquisei o suficiente para entender o que seria necessário para implementar toda a renderização de volumes, mas como o escopo era amplo demais, um projeto menor seria mais realista, e com a ajuda de outros desenvolvedores, fixamos o escopo em texturas tridimensionais, uma parte da renderização volumétrica.

Google Summer of Code

O Google Summer of Code é um programa que oferece a desenvolvedores que ainda estejam estudando uma bolsa para programar para vários projetos de código aberto.

Como o Blender estava entre as instituições participantes do programa, escrevi a proposta deste projeto e a submeti. Após a seleção que ocorre com o Google e as instituições participantes, este foi um dos trabalhos escolhidos. Ele foi desenvolvido com o acompanhamento de um desenvolvedor da Blender Foundation e financiado pela bolsa do Google.

Capítulo 2

Blender

2.1 Software

O Blender[1] é um pacote de software de computação gráfica 3D, gratuito e de código aberto, usado, por exemplo, na criação de animações, efeitos visuais, modelos tridimensionais impressos, aplicações 3D interativas e jogos digitais. Neste contexto, o software suporta modelagem 3D, texturização, animação, simulação de fluidos e fumaça, simulação de partículas, renderização, entre outras funcionalidades.

O software é mantido pela Blender Foundation, uma instituição sem fins lucrativos, cuja principal fonte de renda é a criação de material de treinamento e o oferecimento de cursos e treinamentos presenciais em Amsterdã. O pacote tem uma grande base instalada, mas é difícil estimar o número de usuários ativos - o número de *downloads* a partir do *site* oficial é superior a 3 milhões por ano. A distribuição oficial, disponibilizada sob a licença GNU-GPL[2], contempla as plataformas Linux, Mac OS X e Windows, para plataformas de 32 e 64 bits.

A interface padrão do usuário prioriza a produção de animações 3D, porém é altamente configurável, e permite um ajuste fino de acordo com as necessidades de cada usuário, embora a curva de aprendizagem possa ser substancial. Uma interface programável (API) é oferecida para *scripts* escritos em Python, permitindo a automação de várias tarefas dentro do software. Na Figura 2.1, a interface do programa é mostrada.

Os principais módulos do Blender são implementados em C, sendo que camadas mais altas usam C++, como a interface gráfica (GUI), que usa OpenGL para não depender dos servidores gráficos de cada plataforma. Os projetos criados no Blender são salvos em arquivos **.blend*, que salvam metadados em cabeçalhos, também a fim de permitir a compatibilidade entre arquiteturas diferentes. Exemplos de metadados salvos são tamanhos

[1] www.blender.org

[2] http://www.gnu.org/licenses/gpl.html

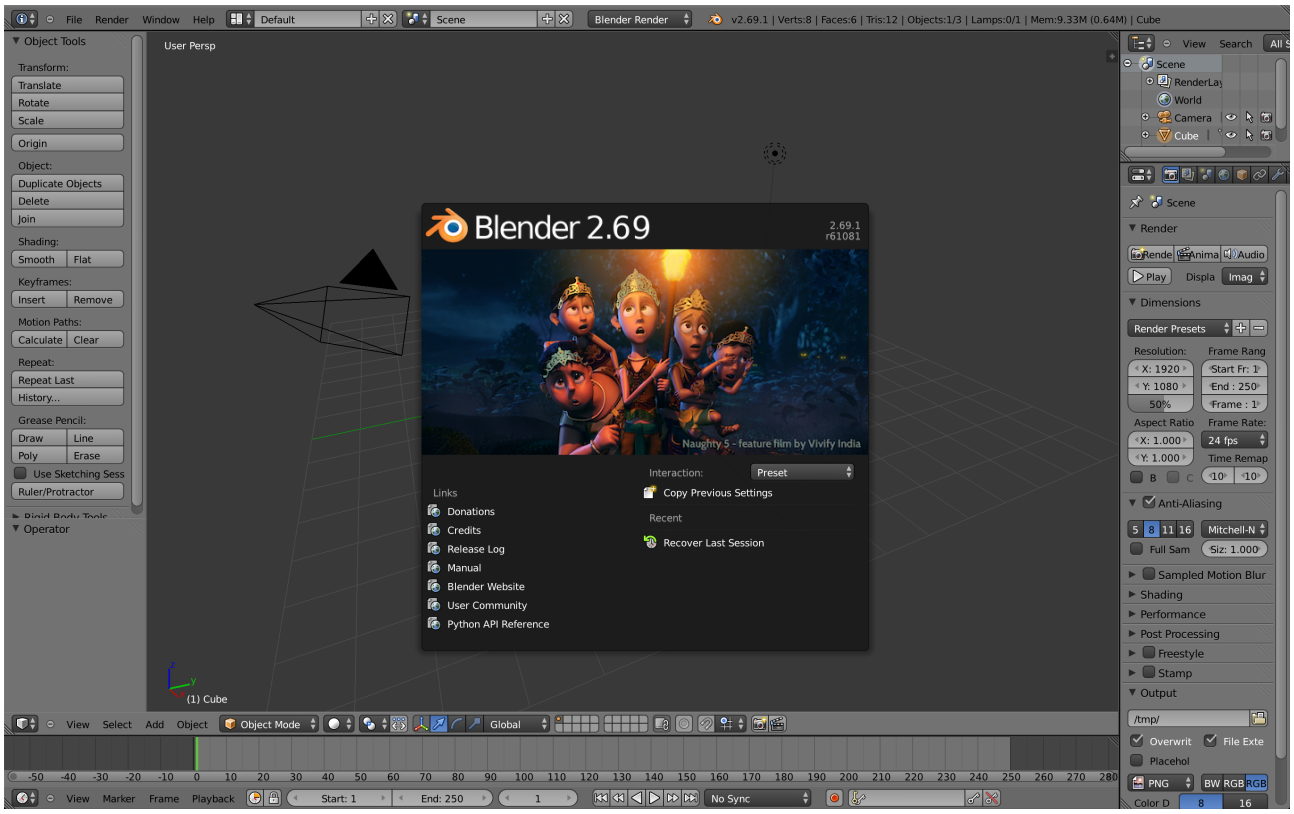

Figura 2.1: Interface do Blender, ao abrir a aplicação.

de ponteiros, versão do software, e disposição de memória[3].
Após o cabeçalho do arquivo, blocos de dados são salvos com informações como códigos identificadores dos objetos na cena, tamanho em memória dos objetos, ponteiros a locais de memória e índices das estruturas de dados que compõe a cena. Cada um destes blocos possuem os vetores de dados com os nomes e tipos de cada objeto, sendo que os tipos são específicos do software, como curvas, malhas, luzes, câmeras, entre outras possibilidades.

2.1.1 Renderização no Blender

Como o Blender é estruturado de maneira bastante modular, a renderização de uma cena criada no software pode ser feita por *engines* diferentes, a escolha do usuário. Em uma instalação convencional, o software oferece duas engines diferentes, mas há suporte para renderizadores de terceiros, com diferentes níveis de integração.

As duas engines fornecidas são a *Blender Internal Render*, às vezes chamada apenas de *Blender Render* e o *Cycles*. Elas serão discutidas com maiores detalhes a seguir. Na Figura 2.2, é mostrado onde, na interface do Blender, a opção por uma engine ou outra pode ser feita. Uma terceira engine, a *Blender Game Engine*, também é fornecida, mas engloba muito mais que apenas a geração de imagens, e é usada para a criação de games

[3]Exemplos são *little endian* e *big endian*.

ou de aplicações interativas, gerando um arquivo executável como resultado.

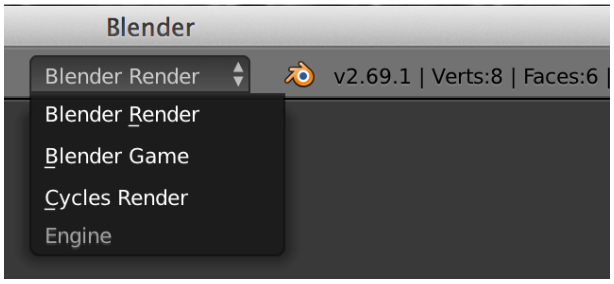

Figura 2.2: Interface do Blender, que permite a escolha de qual engine usar para renderização da cena.

É possível visualizar *previews* da cena, animada ou estática, nas *viewports* da aplicação, e estas podem ser em *(i) wireframe*, mostrando apena as arestas dos objetos, *(ii)* sólida, preenchendo as superfícies com uma estratégia de shading simplificada para rápida visualização, ou ainda *(iii)* com texturas, permitindo uma visualização mais próxima do resultado final, porém mais lenta de ser obtida e menos interativa, já que o tempo de resposta da interface cai consideravelmente. Quando uma renderização completa e de qualidade final é necessária, todo o modelo de dados do Blender é passado à engine de renderização escolhida, software que, conforme discutido, calcula a emissão, absorção, reflexão e transporte da luz simulada e projeta os resultados sobre um plano de imagem definido pela câmera colocada em cena.

Blender Internal Render

A primeira geração de renderizador incluído no Blender, bastante estável, é útil para a geração de imagens em geral, com suporte a renderização volumétrica e a uso da GPU para os cálculos. Esta engine não é fisicamente fiel[4], isto é, as unidades e grandezas usadas nas configurações podem usar, e normalmente usam, valores irreais para quantidades de luz. Um exemplo: uma superfície pode refletir 110% da luz incidente, mesmo quando não é uma superfície emissora de luz.

Cycles Render

É uma engine mais moderna, fisicamente fiel, que modela efeitos cáusticos de maneira realista. Tem suporte a configuração de comportamento através de nós e permite que o usuário crie seus próprios shaders em OSL, linguagem apresentada em 2.4.1. Um exemplo de uma rede de nós para configurar seu comportamento é mostrado na Figura 2.3.

[4]**Fisicamente fiel**, neste contexto, significa que os fundamentos usados para a geração de imagens são as leis da física e suas expressões matemáticas. Unidades e conceitos físicos são usados nos algoritmos e as imagens computadas são ditas fisicamente corretas, isto é, elas correspondem ao comportamento da luz que seria encontrado numa cena real.

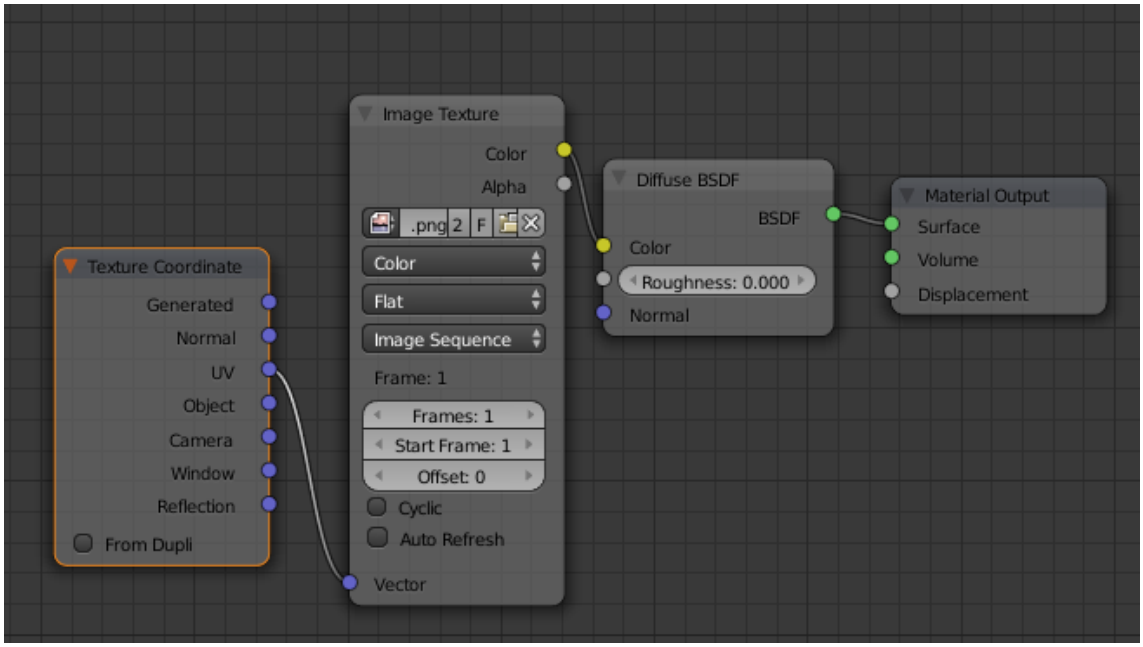

Figura 2.3: Rede de nós configurando o uso de uma sequência de imagens como textura para uma superfície.

Como o software é recente, ele é menos estável e nem todas as funcionalidades desejadas estão disponíveis. Uma das funcionalidades pendentes de implementação é a renderização de volumes, ou volumétrica, assunto deste trabalho.

2.2 Texturas

No processo de modelagem tridimensional, a medida que detalhes se tornam mais sofisticados e complexos, a modelagem explícita de todas as informações desejadas, seja pelo uso de polígonos ou de outras primitivas geométricas, se torna pouco prático e de difícil execução. Uma alternativa é mapear uma imagem à superfície, técnica denominada *Mapeamento de Textura*. A imagem utilizada é chamada de *textura* e os elementos individuais que compõem a textura são chamados de *texels* (texture elements, a exemplo do que ocorre com pixels). Na imagem resultante, é usual que um pixel contenha vários texels.

2.2.1 Definição

Textura é um termo bastante amplo em síntese de imagens. De modo geral, podemos dizer que sempre que um ponto é avaliado usando apenas informações locais àquele ponto, trata-se de uma textura. Estas informações locais podem ser a localização do ponto no espaço, sua posição numa superfície, a direção e módulo das derivadas parciais da superfície naquele ponto, entre outras possibilidades. A função avaliada no ponto normalmente retorna um escalar, mas este resultado poderia, a princípio, ser de qualquer tipo, incluindo

um vetor ou uma cor. É bastante usual que texturas sejam usadas como parâmetros de funções de *shading*, embora hajam exceções.
Uma *textura volumétrica* pode ser avaliada em qualquer ponto do espaço; uma *textura de superfície* pode ser avaliada apenas sobre os pontos pertencentes à superfície. Além disso, texturas podem ser geradas por um programa e avaliadas sob demanda, ou armazenadas em um arquivo e consultadas para encontrar o valor da textura.
O *Mapeamento de Textura*, discutido na próxima seção, descreve a aplicação de uma textura a uma superfície, e determina a relação entre a geometria e o processo de *shading*.

2.2.2 Mapeamento de Textura

As texturas são introduzidas nas imagens com o uso de *mapeamentos de textura*. É uma forma de adicionar detalhes às superfícies sem que exista um modelo geométrico destes detalhes, de modo a permitir a criação de imagens complexas sem causar um aumento de complexidade na geometria.
De modo geral, mapas de texturas podem ser funções de uma ou mais variáveis. A fim de exemplificar seu funcionamento, mostraremos o caso das funções de duas variáveis.
Um *mapa de textura* associa um *texel* a cada ponto no objeto geométrico. Se o objeto for representado por coordenadas homôgeneas, as funções serão dadas por:

$$\begin{aligned} x &= x(s,t), \\ y &= y(s,t), \\ z &= z(s,t), \\ w &= w(s,t). \end{aligned}$$

Embora estas funções existam conceitualmente, encontrá-las pode ser difícil. Além disso, o problema mais usual é o inverso: dado o ponto (x, y, z) ou (x, y, z, w) no objeto, desejamos encontrar as coordenadas de textura correspondentes, isto é, as funções inversas

$$\begin{aligned} s &= s(x,y,z,w), \\ t &= t(x,y,z,w). \end{aligned}$$

Em computação gráfica, a maioria das superfícies são representadas parametricamente. Um ponto $\mathbf{p}$ na superfície é uma função de dois parâmetros u e v. Para cada par de valores, geramos o ponto

$$\mathbf{p}(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}.$$

Dada uma superfície paramétrica, podemos mapear um ponto do mapa de textura $T(s,t)$ para um ponto $\mathbf{p}(u,v)$ na superfície usando um mapa linear da forma

$$\begin{aligned} u &= as + bt + c, \\ v &= ds + et + f. \end{aligned}$$

E se tivermos que $ae \neq bd$, este mapeamento é invertível.

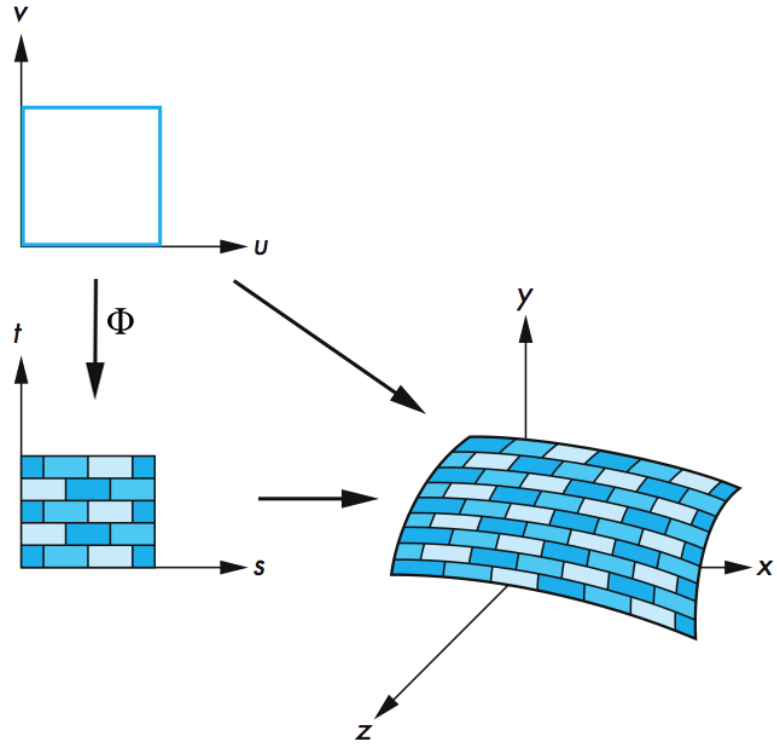

Figura 2.4: Mapeamento do sistema de coordenadas da região parametrizada para o sistema de coordenadas da textura.

Coordenadas de Textura

Depois de a textura ter sido carregada na memória, ela pode ser amostrada, a fim de se obter uma cor a partir dela. Isso é feito usando *coordenadas de textura*. Usualmente são atributos dos vértices, como posição e cor, armazenados como vetores, que determinam a distância em cada eixo da textura em que a amostra deve ser tomada. o tamanho do vetor depende do número de dimensões que a textura tem - uma textura 2D usa um vetor de pares ordenados para as coordenadas, por exemplo. Estas coordenadas são normalizadas, isto é, independente do tamanho da textura em uso, suas coordenadas variam entre $[0.0, 1.0]$. Como resultado, a aplicação não precisa se preocupar com o tamanho da textura depois que ela foi lida.

2.2.3 Estratégias de *Wrapping*

Embora texturas sejam definidas como tendo coordenadas normalizadas, os vértices podem usar coordenadas fora deste intervalo. O que ocorre neste caso depende do modo escolhido para *wrapping* da textura, que basicamente se refere a como a textura vai "embrulhar"a geometria. Um dos modos é denominado *clamping*, que limita os *texels* ao tamanho da textura, isto é, se a coordenada solicitada é maior que a textura, a consulta vai retornar o valor da fronteira mais próxima da textura, efetivamente limitando o inter-

valo válido de consulta ao intervalo $[0.0, 1.0]$. No entanto, se a repetição da textura for permitida, as coordenadas para consulta à textura fazem um loop na textura: solicitar o valor da textura no ponto 1.5, 2.5 ou até mesmo 100.5, retorna o valor referente ao meio da textura, de coordenada 0.5 no eixo considerado.

Na API adotada para este trabalho, a estrutura `Wrap` define os métodos suportados, como mostrado no Código 2.1. O modo *Black* retorna a cor preta pra qualquer ponto que caia fora do intervalo normalizado, isto é, todo ponto que estiver fora da área da textura, recebe a cor preta como resultado. O modo *Clamp* limita todas as consultas para o intervalo da textura, e retorna o valor da extremidade mais próxima quando o ponto solicitado estiver além da fronteira da textura. Os modos *Periodic* e *Mirror* repetem a textura, sendo que o primeiro repete a textura em sua orientação normal e o segundo reflete a textura em relação a um dos eixos antes de repeti-la.

Código 2.1: Modos de *wrapping* suportados pela API.

```
class OIIO_API TextureOpt {
public:
    /// Wrap mode describes what happens when texture coordinates
    /// describe a value outside the usual [0,1] range
    /// where a texture is defined.
    enum Wrap {
        WrapDefault,        ///< Use the default found in the file
        WrapBlack,          ///< Black outside [0..1]
        WrapClamp,          ///< Clamp to [0..1]
        WrapPeriodic,       ///< Periodic mod 1
        WrapMirror,         ///< Mirror the image
        WrapLast            ///< Mark the end -- don't use this!
    };
}
```

2.2.4 Texturas Tridimensionais

A renderização volumétrica (chamaremos de renderização volumétrica o processo de geração de imagens que use uma textura tridimensional) parte da existência de um campo escalar contínuo tridimensional, que pode ser escrito como um mapeamento

$$\Phi : \mathbb{R}^3 \to \mathbb{R},$$

que é uma função do espaço 3D para um valor de apenas um componente. Na prática, no entanto, um campo volumétrico é dado por um grid discretizado porque trata-se do resultado de uma simulação ou de uma medição. A Figura 2.5 mostra um conjunto de dados volumétricos representado em um grid discreto.
Uma imagem digital bidimensional é um bom exemplo para discutirmos a representação de um volume. Uma imagem consiste de *pixels* (abreviação de *picture elements*, elementos da figura, em inglês) organizados em um vetor convencional. *Pixels* são os elementos

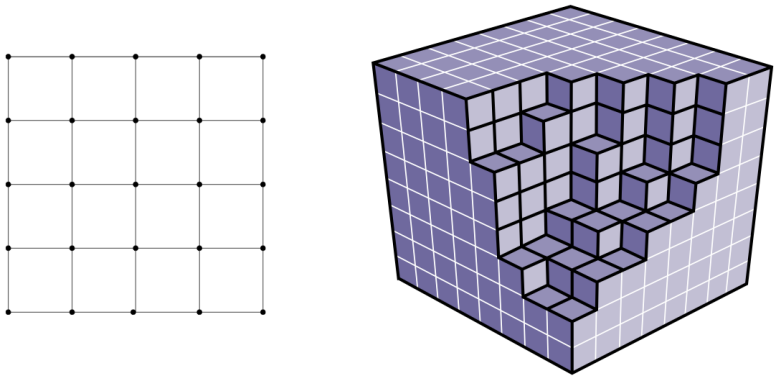

Figura 2.5: Exemplos de grids uniformes: grid uniforme bidimensional com células quadráticas, à esquerda, e um grid uniforme tridimensional com células cubóides, à direita.

de dados de uma imagem 2D, e que possuem valores de cor. Um conjunto volumétrico discreto pode ser representado de uma maneira similar se levarmos o espaço de duas para três dimensões. Assim, um pixel 2D passa a ser um *voxel* 3D (*volume element*, elemento de volume, em inglês). *Voxels* são organizados em um vetor tridimensional convencional também, abrangendo o conjunto de dados que compõe o volume. Embora seja intuitivo, o termo *voxel* tem duas possíveis interpretações. A primeira delas é a de que o *voxel* é um pequeno cubo que preenche uma pequena região volumétrica com o valor a ser armazenado. A outra interpretação assume que os *voxels* são pontos no espaço de três dimensões, aliados a técnicas de interpolação que preenchem os espaços intermediários entre os pontos. Neste trabalho, esta última interpretação se adequa melhor, dada a biblioteca que usamos para representar volumes e as estratégias implementadas para reconstruir os valores nas lacunas entre os pontos definidos.

Um grid uniforme n-dimensional tem a vantagem de ser bem estruturado, o que leva a uma representação compacta na memória do computador (na forma de um vetor n-dimensional) e acesso rápido, constante $O(1)$, às celulas. Porém, conforme será discutido no capítulo 3, essa observação só é válida para os casos em que o volume é denso, isto é, não há predominância de células intermediárias vazias. Além disso, grids uniformes não são muito flexíveis, e outras estruturas de grids podem ser usadas em seu lugar, sendo que apresentaremos uma das possibilidades.

Voxels

Conforme discutido na seção anterior, podemos acessar o conteúdo de um voxel usando suas coordenadas no espaço, da mesma forma que podemos acessar o conteúdo de uma imagem bidimensional. É usual chamarmos as variáveis que representam os índices de i, j e k para os eixos x, y e z, respectivamente. Em notação matemática, é comum a opção por índices subscritos, por exemplo: para um buffer de voxels B, dizemos que ele

possui voxels nos pontos amostrais dados por $B_{i,j,k}$. No código, os dados amostrados são acessados usando coordenadas com sinal. Um exemplo é mostrado no Código 2.2.

Código 2.2: Consulta ao valor de ponto flutuante no voxel de coordenadas (1, 2, 3)

```
openvdb::FloatGrid grid = ...;
openvdb::FloatGrid::Accessor accessor = grid.getAccessor();
openvdb::Coord ijk(1,2,3);
float value = accessor.getValue(ijk);
```

Fonte dos dados de uma textura volumétrica

Os dados usados para a renderização volumétrica podem ser provenientes de diferentes áreas de aplicação. Um tipo importante de aplicação é a visualização científica de dados escalares. Mais especificamente, imagens médicas é uma das aplicações mais proeminentes em visualização. Para obter os dados tridimensionais, usa-se algum tipo de equipamento de escaneamento, como no caso da tomografia computadorizada, que gera imagens a partir da emissão de radiação ao redor do corpo. Uma imagem 3D é gerada a partir da radiação refletida pelo corpo-alvo. Outra fonte semelhante é a ressonância magnética, que opera de forma parecida com a tomografia, mas depende de um grande campo magnético para interagir com a emissão de radiação.

Outra fonte corriqueira de dados volumétricos são as simulações, como as simulações de dinâmica de fluidos, campos magnéticos e de fogo e explosões para efeitos especiais. Neste caso, o grid usado para a simulação é normalmente bem diferente daquele usado para visualização.

O grid usado na simulação pode ser estruturado para garantir uma simulação estável, com características específicas ao fenômeno modelado, como usar um grid sem estruturas fixas e que seja adaptativo, por exemplo. Aplicações de visualização normalmente dependem de um grid uniforme, que facilita métodos rápidos de geração de imagens. Desta forma, é usual que grids resultantes de simulações sejam transformados antes de fornecidos a um renderizador volumétrico.

Reconstrução

Como um conjunto volumétrico de dados é representado de forma discreta, existe o problema de reconstruirmos uma função escalar em todos os pontos do domínio 3D. O problema de uma reconstrução fiel é estudado em processamento de sinais.

Pelo teorema de amostragem de Nyquist-Shannon da teoria da informação, temos que a frequência do sinal de entrada precisa ser maior que o dobro da frequência máxima que ocorre no sinal de entrada para que o sinal original possa ser reconstruído a partir do sinal amostrado. Caso contrário, o sinal apresentará problemas de serrilhamento (aliasing), isto

é, o sinal contínuo será reconstruído com erros a partir do sinal discreto.

Em notação matemática, a amostragem apropriada pode ser descrita da seguinte forma: para um sinal de entrada periódico e contínuo representado pela função $f(t)$, determinamos primeiro sua frequência máxima v_f. Frequência máxima significa que a transformada de Fourier de $f(t)$ é zero fora do intervalo de frequências $[-v_f, v_f]$. Com isso, temos a frequência crítica de amostragem (também chamada de frequência Nyquist) $v_N = 2v_f$. Para uma amostragem apropriada, mais que $2v_f$ amostras precisam ser tomadas por unidade de distância. Para uma amostragem uniforme numa frequência v_s, os pontos amostrais podem ser descritos por $f_i = f(i/v_s)$, com i inteiros.

Dado que o sinal original é amostrado numa frequência $v_s > v_N$, o sinal pode ser recuperado das amostras f_i conforme abaixo:

$$f(t) = \sum_i f_i \; sinc(\pi(v_s t - i)).$$

A função $sinc(x)$ (de *sinus cardinalis*, seno cardinal, em latim) é dada por:

$$sinc(t) = \begin{cases} \frac{sen(t)}{t} & se \;\; t \neq 0 \\ 1 & se \;\; t = 0 \end{cases}$$

Um sinal cuja transformada de Fourier é zero fora do intervalo de frequências $[-v_f, v_f]$ é chamado de limitado em banda porque sua largura de banda (isto é, sua frequência) é limitada. Na prática, as frequências que ocorrem num dado conjunto de dados podem ser desconhecidas. Nestes casos, um filtro passa-baixa pode ser aplicado para restringir a frequência máxima em um valor controlado.

A $f(t)$ apresentada é um exemplo de convolução, e descreve a convolução do sinal amostrado de entrada f_i com um filtro *sinc*. Na realidade, porém, o filtro sinc possui alcance ilimitado, isto é, ele oscila em torno de zero sobre todo o domínio. Como consequência, a convolução precisa ser calculada para todas as amostras de entrada f_i, o que toma muito tempo. É por isso que na prática aplica-se filtros de reconstrução com suporte finito.
Um exemplo é o filtro *caixa*, que leva à interpolação do vizinho mais próximo quando a largura da caixa é igual à distância de amostragem (isto é, o valor da função reconstruída é dado pelo valor do ponto amostral mais próximo). Outro exemplo é o filtro *triangular*, que leva a uma reconstrução linear quando a largura de um lado for igual à distância de amostragem. A Figura 2.6 mostra três tipos diferentes de filtros de reconstrução que podem ser usados.

E embora tenhamos discutido funções de apenas uma variável, podemos extender a discussão para n dimensões usando uma abordagem de produto tensorial. A idéia é basicamente fazer a reconstrução de cada dimensão, mas de forma combinada. Para o nosso caso, em que precisamos de três dimensões, um filtro de reconstrução usando produto tensorial

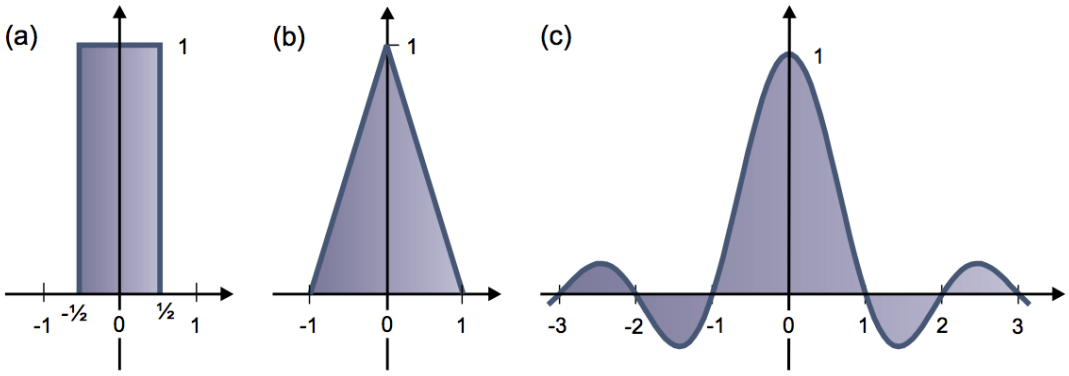

Figura 2.6: Exemplos de filtros de reconstrução: (a) filtro caixa, (b) filtro triangular, e (c) filtros que usam o seno cardinal, *sinc*.

é dado por $h(x, y, z) = h_x(x)\ h_y(y)\ h_z(z)$, onde $h_x(\cdot)$, $h_y(\cdot)$ e $h_z(\cdot)$ são filtros de um único parâmetro ao longo dos eixos x, y e z. E uma das vantagens do uso de um grid uniforme é que eles dão suporte a este tipo de reconstrução de forma direta. Como mostrado na Figura 2.7, a abordagem de produto tensorial separa as interpolações ao longo de cada dimensão e permite o cálculo do valor reconstruído através de uma sequência de interpolações lineares.

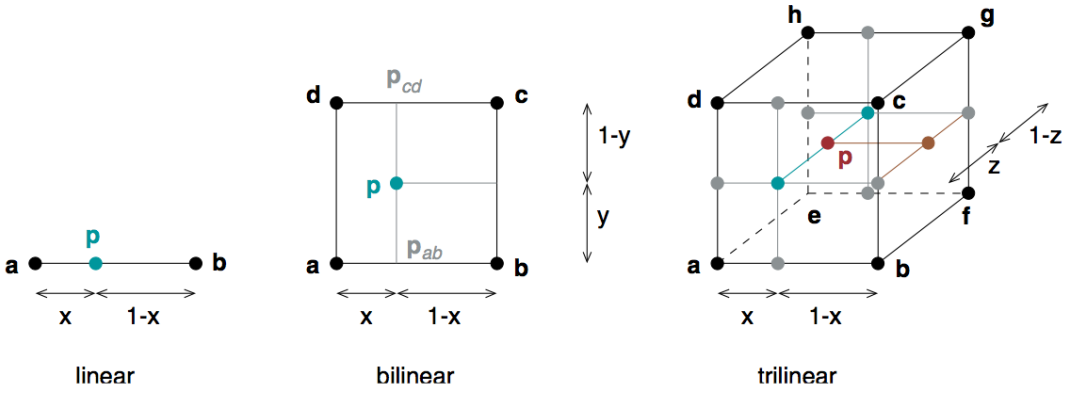

Figura 2.7: Interpolações lineares usando o produto tensorial: casos linear, bilinear e trilinear.

Filtros e Técnicas de Interpolação

Quando desejamos aplicar uma textura 2D à geometria 3D, por exemplo, é bastante improvável que haja uma correspondência exata de 1 : 1 entre os *texels* da textura e os *pixels* da tela ou da imagem a ser gerada. Duas possibilidades existem neste caso: quando uma textura é amostrada, o *texel* mais próximo do ponto solicitado pode ser retornado, ou uma *interpolação* pode ser aplicada a um grupo de *texels* para que um valor possa ser

calculado e retornado.

O exemplo mais simples de um filtro de textura no caso bidimensional é a *Interpolação Bilinear* - os *texels* mais próximos nos eixos x e y são considerados a fim de calcular a cor final.

Para o caso de texturas tridimensionais, desejamos consultar o valor de um *voxel*. Pelo uso de coordenadas (i, j, k) podemos acessar *voxels* individuais, mas a princípio, isso funciona apenas quando o ponto que desejamos amostrar corresponde exatamente ao centro do *voxel*. Em todos os outros casos, uma estimativa precisa ser feita acerca do valor, a ser definido pelos *voxels* vizinhos ao ponto que buscamos. As principais técnicas de interpolação serão discutidas a seguir.

Interpolação do vizinho mais próximo. Em inglês, interpolação Nearest-Neighbor. Trata-se da forma mais simples de interpolação, e pode-se dizer até que, de tão simples, não se qualifica como uma técnica de interpolação. Ao invés de considerar valores de *voxels* vizinhos, este método considera apenas o valor do centro do *voxel* que estiver mais próximo do ponto solicitado. Como consequência, a imagem resultante mostra claramente as fronteiras de cada *voxel*.
Dada uma função discretamente amostrada $\mathbf{d}$ e uma posição x, a interpolação do vizinho mais próximo em uma dimensão pode ser escrita como

$$I_n(x, \mathbf{d}) = \mathbf{d}_{\lfloor x+0.5 \rfloor}.$$

Em código, temos um vetor unidimensional A e uma coordenada de ponto flutuante x. Assumindo que x está dentro do intervalo válido de A, a implementação é mostrada no Código 2.3.

Código 2.3: Implementação da interpolação do vizinho mais próximo.

```
float vizinhoMaisProximo_1D(float x, float *A)
{
  const int idx = static_cast<int>(floor(x + 0.5));
  return A[idx];
}
```

O comportamento desta técnica de interpolação é ilustrado pela Figura 2.8.
A implementação para o caso tridimensional é mostrado no código 2.4.

Código 2.4: Implementação da interpolação do vizinho mais próximo para o caso tridimensional.

```
// transformar coordenadas de voxel de ponto flutuante em coordenadas
    inteiras
```

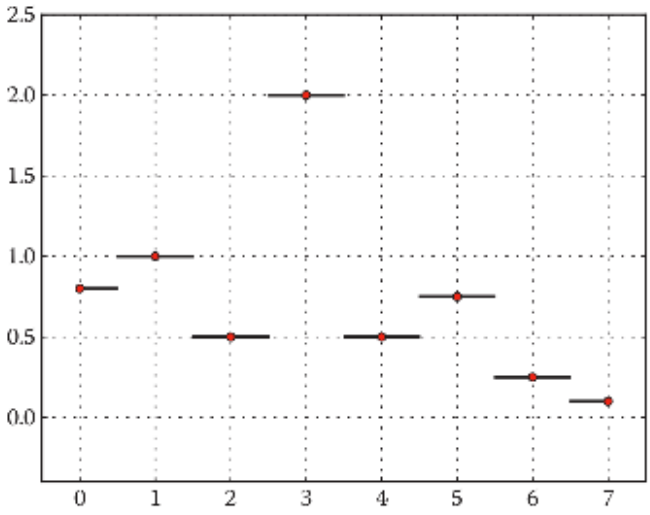

Figura 2.8: Interpolação do vizinho mais próximo. Os segmentos de reta mostram todo o intervalo que seria resolvido ao ponto denotado por cada círculo.

```
int continuosToDiscrete(float continuousCoordinate)
{
  return static_cast<int>(std::floor(continuousCoordinate));
}

// vizinho mais proximo, caso tridimensional
float vizinhoMaisProximo_3D(const Imath::v3f &vsP, const VoxelBuffer &buf
    )
{
  V3i dvsP = continuosToDiscrete(vsP);
  return buf.value(dvsP.x, dvsP.y, dvsP.z);
}
```

Interpolação Linear. Entre as técnicas de interpolação que usam valores intermediários para o cálculo, a interpolação linear é o caso mais simples. Os valores são calculados considerando os pontos próximos do ponto consultado e usando ponderação em função da distância, de modo que a contribuição de um ponto amostral é igual a 1.0 se o ponto consultado está alinhado com o centro do voxel e é 0.0 quando a distância do ponto consultado até o centro do voxel é 1.0.
Dada a mesma função $\mathbf{d}$ e coordenada x do caso anterior, a interpolação linear pode ser escrita da seguinte forma:

$$I_l(x, \mathbf{d}) = (1 - \alpha) \cdot \mathbf{d}_{\lfloor x \rfloor} + \alpha \cdot \mathbf{d}_{\lfloor x \rfloor},$$

$$\alpha = x - \lfloor x \rfloor .$$

A implementação é apresentada para o caso unidimensional no Código 2.5. Para o caso tridimensional, há mais a ser feito. Os índices discretos precisam ser encontrados da mesma forma que no caso da interpolação pelo vizinho mais próximo, mas ao contrário

daquele caso, precisamos agora dos valores de oito voxels vizinhos, ao invés de apenas um. De modo geral, quanto maior a vizinhança que precisa ser considerada na interpolação, mais lento o processo. Definimos como *Suporte* (denotado por S) o número de vizinhos a serem considerados numa dada técnica de interpolação. Para interpolação linear, temos $S_{linear} = 2$. O número de valores de voxel que uma técnica de interpolação precisa é dado por S^d, em que d é o número de dimensões. Logo, para nosso caso, interpolação linear em três dimensões, o número de amostras necessário é $2^3 = 8$. Na Figura 2.9, pode parecer que as descontinuidades indicam as fronteiras dos voxels considerados, mas na realidade, a descontinuidade ocorre quando o ponto sendo amostrado passa de um intervalo $[d_{i-1}, d_i]$ para um intervalo $[d_i, d_{i+1}]$, e isso ocorre nos centros dos voxels, e não nas fronteiras.

Código 2.5: Implementação da interpolação linear para uma dimensão.

```
float interpolacaoLinear(float x, float *A)
{
  const int pisoX = static_cast<int>(floor(x));
  const int tetoX = static_cast<int>(floor(x));
  float alpha = x - pisoX;
  return (1.0 - alpha) * A[pisoX] + alpha * A[tetoX];
}
```

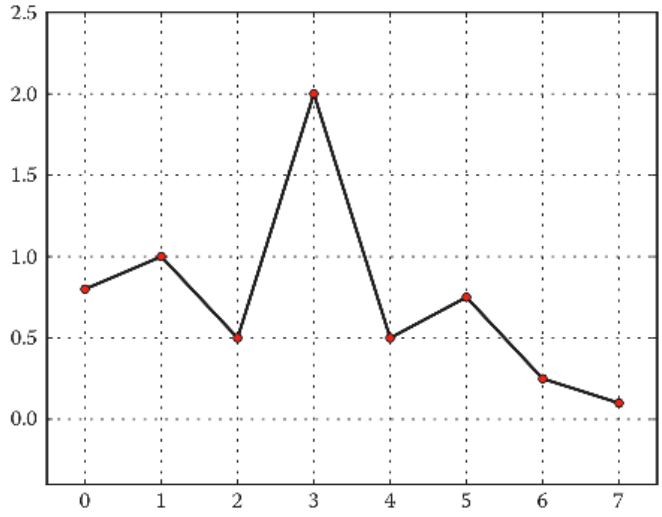

Figura 2.9: Interpolação linear. As quebras ocorrem na transição do centro de um voxel para o centro do voxel seguinte.

A implementação do caso tridimensional é apresentada no Código 2.6.

Código 2.6: Implementação da interpolação linear para três dimensões.

```
float linear3D(const Imath ::V3f &vsP, const VoxelBuffer &buf )
{
    const int fX = static _cast <int >( floor(vsP.x - 0.5));
```

```
    const int fY = static _cast <int >( floor(vsP.y - 0.5));
    const int fZ = static _cast <int >( floor(vsP.z - 0.5));
    const float aX = vsP.x - 0.5 - fX;
    const float aY = vsP.y - 0.5 - fY;
    const float aZ = vsP.z - 0.5 - fZ;
    const float bX = 1.0 - aX;
    const float bY = 1.0 - aY;
    const float bZ = 1.0 - aZ;
    return bX * (bY * (bZ * buf .value(fx, fy, fz ) +
                       aZ * buf .value(fx, fy, fz + 1)) +
                 aY * (bZ * buf .value(fx, fy + 1, fz ) +
                       aZ * buf .value(fx, fy + 1, fz + 1))) +
    aX * (bY * (bZ * buf .value(fx + 1, fy, fz ) +
                aZ * buf .value(fx + 1, fy, fz + 1)) +
          aY * (bZ * buf .value(fx + 1, fy + 1, fz ) +
                aZ * buf .value(fx + 1, fy + 1, fz + 1)));
}
```

Interpolação Cúbica. A última técnica de interpolação que trataremos é a interpolação cúbica. A interpolação linear usa um polinômio de primeira ordem (linear) para calcular os valores intermediários, mas seu resultado, embora não seja ruim como a interpolação do vizinho mais próximo, ainda é pouco suave. Se usarmos polinômios de ordens maiores, obteremos resultados melhores, isto é, com transições mais suaves, ao custo de velocidade, já que o tempo de cálculo aumentará.
O próximo passo, depois da interpolação linear, é a interpolação cúbica, que usa quatro valores vizinhos para a interpolação:

$$I_c(x, p_{i-1}, p_i, p_{i+1}, p_{i+2}).$$

Para derivar a fórmula da interpolação cúbica, consideramos a forma básica de um polinômio de terceira ordem e sua derivada:

$$f(x) = ax^3 + bx^2 + cx + d$$
$$f'(x) = 3ax^2 + 2bx + c.$$

Os pontos que desejamos interpolar estão no intervalo $[0, 1]$, e os quatro pontos que funcionam como entrada para a função de interpolação são

$$\begin{aligned} p_{i-1} &= -1, \\ p_i &= 0, \\ p_{i+1} &= 1, \\ p_{i+2} &= 2. \end{aligned}$$

Calculando a função e sua derivada em 0 e 1, obtemos

$$\begin{aligned} f(0) &= d, \\ f(1) &= a + b + c + d, \\ f'(0) &= c, \\ f'(1) &= 3a + 2b + c. \end{aligned}$$

Reescrevendo cada um dos resultados de modo a isolar a, b, c e d, obtemos

$$\begin{aligned} a &= 2f(0) - 2f(1) + f'(0) + f'(1), \\ b &= -3f(0) + 3f(1) - 2f'(0) - f'(1), \\ c &= f'(0), \\ d &= f(0). \end{aligned}$$

Sabemos os valores de $f(0$ e de $f(1)$, mas os valores das derivadas não são dados diretamente à função. Porém, como temos $f(-1)$ e $f(2)$, podemos calculá-las:

$$\begin{aligned} f(0) &= p_i, \\ f(1) &= p_{i+1}, \\ f'(0) &= \tfrac{p_{i+1} - p_{i-1}}{2}, \\ f'(1) &= \tfrac{p_{i+2} - p_i}{2}. \end{aligned}$$

Combinando os dois últimos conjuntos de equações, encontramos os valores de a, b, c e d em sua forma simples e fechada:

$$\begin{aligned} a &= -\tfrac{1}{2}p_{i-1} + \tfrac{3}{2}p_i + \tfrac{3}{2}p_{i+1} + \tfrac{1}{2}p_{i+2}, \\ b &= p_{i-1} + \tfrac{5}{2}p_i + 2p_{i+1} - \tfrac{1}{2}p_{i+2}, \\ c &= -\tfrac{1}{2}p_{i-1} + \tfrac{1}{2}p_{i+1}, \\ d &= p_i, \end{aligned}$$

o que nos leva à solução do problema original,

$$\begin{aligned} I_c(x, p_{i-1}, p_i, p_{i+1}, p_{i+2}) = &\left(-\frac{1}{2}p_{i-1} + \frac{3}{2}p_i + \frac{3}{2}p_{i+1} + \frac{1}{2}p_{i+2}\right)x^3 + \\ &\left(p_{i-1} + \frac{5}{2}p_i + 2p_{i+1} - \frac{1}{2}p_{i+2}\right)x^2 + \\ &\left(-\frac{1}{2}p_{i-1} + \frac{1}{2}p_{i+1}\right)x + \\ &p_i. \end{aligned}$$

A interpolação cúbica usa dois valores abaixo e dois acima do ponto consultado, então a técnica tem um suporte $S_{cbico} = 4$, como é possível visualizar na Figura 2.10. Em três dimensões, o custo total é dado por $4^3 = 64$ consultas para o cálculo de um único valor. No Código 2.7, a implementação é dada para uma, duas e três dimensões. A implementação de uma dimensão maior sempre usa a função definida para uma dimensão menor.
Uma das desvantagens da interpolação cúbica é que encontramos casos de *overshoots*. Quando temos grandes gradientes nos dados discretos, a técnica pode assumir valores maiores ou menores que qualquer um dos dados existentes em todo o conjunto. Dado o escopo deste trabalho, consideraremos que os problemas que isso acarreta são aceitáveis na geração de imagens para nossa aplicação.

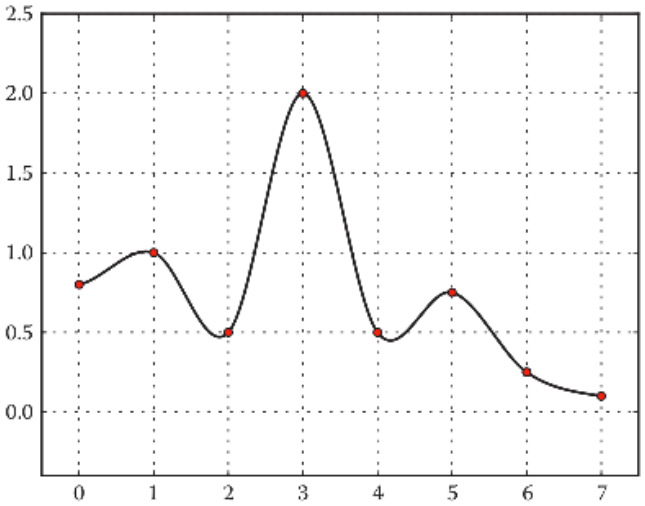

Figura 2.10: Interpolação cúbica. Como o polinômio é de terceira ordem, as curvas são bem mais suaves que da interpolação linear.

Código 2.7: Implementação da interpolação cúbica para uma, duas e três dimensões.

```
template <class Data_T, class Coord _T>
Data_T interpolacaoCubica1D(Coord _T x, Data_T p[4])
{
    return p [1] + 0.5 *
    x * (p [2] - p [0] +
         x * (2.0 * p [0] - 5.0 * p [1] + 4.0 * p [2] - p [3] +
              x * (3.0 * (p [1] - p [2]) + p [3] - p [0])));
}

template <class Data_T, class Coord _T>
Data_T interpolacaoCubica2D(Coord _T x, Coord _T y, Data _T p[4][4])
{
    Data_T yInterps[4];
    yInterps[0] = interpolacaoCubica1D (y, p [0]);
    yInterps[1] = interpolacaoCubica1D (y, p [1]);
    yInterps[2] = interpolacaoCubica1D (y, p [2]);
    yInterps[3] = interpolacaoCubica1D (y, p [3]);
    return interpolacaoCubica1D (x, yInterps);
}

template <class Data_T, class Coord _T>
Data_T interpolacaoCubica3D(Coord _T x, Coord _T y, Coord _T z, Data _T p
   [4][4][4])
{
    Data_T yzInterps[4];
    yzInterps[0] = interpolacaoCubica2D (y, z , p [0]);
    yzInterps[1] = interpolacaoCubica2D (y, z , p [1]);
    yzInterps[2] = interpolacaoCubica2D (y, z , p [2]);
    yzInterps[3] = interpolacaoCubica2D (y, z , p [3]);
```

```
        return interpolacaoCubica1D (x, yzInterps);
    }
```

2.3 Sistemas de Coordenadas

É bastante natural que o conceito de Sistemas de Coordenadas permeie as aplicações de computação gráfica. Afinal, é bastante usual criar um objeto em seu próprio sistema de coordenadas, chamado de sistema de coordenadas de modelagem do objeto, e posteriormente colocar este objeto em uma cena usando as coordenadas do mundo. O objetivo principal de existirem vários sistemas de coordenadas é permitir uma execucação mais eficiente de cada passo do processo de visualização. Para este fim, os objetos precisam ser transformados para sistemas de coordenadas (ou espaços de referência) nos quais as tarefas inerentes a cada etapa do processo seja mais natural e conveniente. O processo gráfico completo, do modelo até a exibição da imagem em um monitor envolve o uso de 7 espaços diferentes. São eles os espaços do objeto, da cena ou do mundo, da câmera, espaço normalizado, de ordenação, de imagem e do dispositivo. Comentaremos apenas acerca dos dois espaços mais relevantes a esse trabalho, mas um tratamento completo pode ser encontrado em XX.

Espaço do objeto

Também chamado de espaço do modelo, é o sistema de coordenadas em que um objeto 3D específico está definido. Normalmente, embora não sempre, cada objeto terá seu próprio espaço de objeto com origem no centro do objeto em questão. Por centro, entenda-se o ponto em torno do qual o objeto é movido e rotacionado, como um pivô, e que pode não coincidir com o centro geométrico do objeto. Um exemplo é mostrado na Figura 2.11.

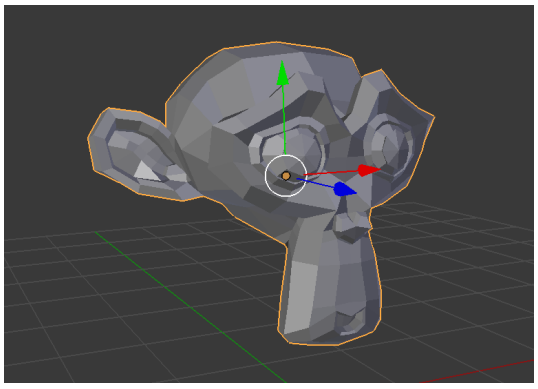

Figura 2.11: Espaço do objeto.

Espaço da cena

Espaço da cena, também chamado de espaço do mundo, ou ainda de sistema de coordenadas global, é o sistema de coordenadas do universo tridimensional em consideração. Nele, os objetos do cenário são posicionados e orientados, uns em relação aos outros, incluindo a câmera virtual. Um exemplo é mostrado na Figura 2.12.

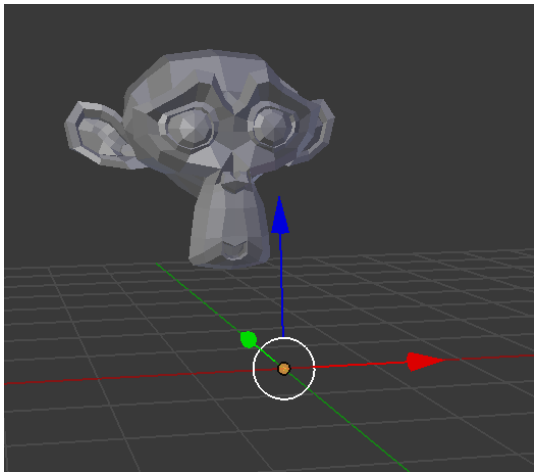

Figura 2.12: Espaço da cena, com a origem centrada no círculo menor.

2.3.1 Notação Homogênea

Um ponto descreve um local no espaço, ao passo que um vetor descreve uma direção e não possui localização. Usando matrizes 3×3 (ou, eventualmente, 2×2 para duas dimensões), é possível aplicar transformações lineares como rotações, escala e cisalhamento às coordenadas. Contudo, translações não são possíveis com o uso destas matrizes. Essa característica não afeta operações com vetores, mas a translação é uma operação importante para pontos.

A notação homogênea é útil para transformar tanto vetores quanto pontos, e nos permite aplicar translações apenas aos pontos. As matrizes 3×3 são aumentadas para 4×4 e pontos tridimensionais e vetores passam a ter um elemento a mais. Assim, um vetor homogêneo é representado por $\mathbf{p} = (p_x, p_y, p_z, p_w)$. Nestas condições, $p_w = 1$ para pontos e $p_w = 0$ para vetores. Quando lidamos com projeções, outros valores podem ser usados para p_w. Dessa forma, quando $p_w \neq 0$ e $p_w \neq 1$, precisamos homogeneizar o vetor, e fazemos $(\frac{p_x}{p_w}, \frac{p_y}{p_w}, \frac{p_z}{p_w}, 1)$ a fim de obter o ponto que de fato desejamos. No caso mais simples, M é tal como mostrada abaixo.

$$M_{4\times4} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

As matrizes específicas das transformações de rotação, escala e cisalhamento podem substituir a matriz M apresentada, e estas operações afetarão pontos e vetores, como esperado. Uma translação, contudo, usa os elementos adicionais da matriz aumentada para obter o resultado necessário. Uma matriz de translação, T, que translada um ponto por um vetor t, é mostrada abaixo.

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A combinação de uma transformação linear seguida de uma translação é denominada *transformada afim*.
Verificamos com facilidade que um vetor $\mathbf{v} = (v_x, v_y, v_z, 0)$ não é afetado pela transformação $\mathbf{Tv}$ porque seu último elemento é 0. Se ao ponto $p = (p_x, p_y, p_z, 1)$ aplicarmos a transformada $\mathbf{T}p$, o resultado que obtemos é $p = (p_x + t_x, p_y + t_y, p_z + t_z, 1)$, ou seja, o ponto p transladado por t.
Como é de se esperar, multiplicações entre matrizes e multiplicações entre matrizes e vetores são calculadas normalmente, sem alterar o que foi discutido.

2.4 Shaders

Um shader é um programa com entradas e saídas, que executa uma tarefa específica no processo de renderização de uma cena, como determinar a aparência de um material ou o comportamento de uma luz. O código fonte do programa é escrito numa linguagem altamente dependente do ambiente alvo. Exemplos incluem a OpenGL Shading Language (GLSL) e a Direct3D High Level Shader Language (D3D-HLSL). Neste trabalho, a linguagem de shading adotada é a Open Shading Language, discutida na seção 2.4.1.

De modo geral, shaders são programas bastante simples que descrevem as características de um vértice ou de um pixel. Shaders de vértice descrevem as características de um vértice, como posição, coordenadas de textura, cores, entre outras características. Shaders de pixel descrevem informações como cor, profundidade (z-depth, quando um z-buffer[5] está sendo usado), transparência, entre outras possibilidades.

2.4.1 Open Shading Language

A Open Shading Language (OSL) é uma linguagem sucinta, apesar de bastante completa, para a programação de shaders em renderizadores modernos e aplicações semelhantes, e

[5]Algoritmo que implementa uma solução adotada para o problema de visibilidade tridimensional quando uma cena é transformada para as coordenadas de tela, e a profundidade relativa entre os objetos precisa ser estabelecida.

é ideal para descrever materiais, luzes, deslocamentos e geração de padrões. A linguagem foi desenvolvida pela Sony Pictures Imageworks para uso em seu renderizador proprietário para animação de longa-metragens e efeitos visuais. A especificação da linguagem foi desenvolvida com a participação de outras empresas produtoras de animação e efeitos especiais e que também possuíam interesse em utilizá-la. A licença utilizada é a nova-BSD, de 1999, o que permite seu uso em qualquer aplicação gratuita ou comercial, aberta ou proprietária, bem como modificação do código fonte, desde que o copyright original seja mantido.

A OSL tem uma sintaxe parecida com a do C, e parecida também com outras linguagens de shading. No entanto, ela foi escrita especificamente para algoritmos modernos de renderização e suporta closures[6] para cálculo de cores, BSDFs[7] e ray tracing postergado, com avaliação dos raios em momento posterior do processo de geração da imagem.

Uma das principais características desta linguagem é que o resultado dos shaders de superfície e de volume são closures, e não cores resultantes, o que seria mais habitual. No entanto, temos que os shaders escritos em OSL calculam uma descrição simbólica explícita, o closure, para descrever como a superfície ou o volume propaga a luz, em unidades radiométricas. Estes closures podem ser avaliados em mais de uma direção de propagação de luz, podem ser amostrados para encontrar direções importantes numa dada cena, ou armazenados para avaliação posterior pelo renderizador que está rodando o shader. Esta abordagem é ideal para renderizadores que tentam ser fiéis ao comportamento físico da luz, especialmente para os que suportam ray tracing[8] e iluminação global[9].

Código 2.8: Estrutura de um shader escrito em OSL.

```
#include "stdosl.h"

tipo-retorno-shader nome-shader ( parametros-opcionais )
{
    comandos
}
```

[6]Função ou referência a uma outra função, dotada de um ambiente de referência. Isto é, permite que uma função acesse variáveis não locais mesmo quando chamada fora de seu escopo léxico imediato.

[7]**Bidirectional scattering distribution function**, função de distribuição bidirecional de propagação, em tradução livre. Trata-se da função que descreve como a luz é propagada por uma superfície.

[8]Técnica de geração de imagens que se baseia em traçar raios de luz através dos pixels no plano da imagem e simular os efeitos de suas interações com os objetos da cena modelada.

[9]**Global Illumination** (GI) abrange as técnicas que consideram em adição à iluminação direta a iluminação indireta, como a reflexão da luz em outros objetos.

Capítulo 3

Estrutura de Dados

A estrutura de dados utilizada é a VDB, renomeada para OpenVDB[1] na ocasião em que foi disponibilizada com código fonte aberto. OpenVDB é uma biblioteca que inclui uma estrutura de dados compacta e hierárquica e um conjunto de ferramentas para a manipulação eficiente de dados volumétricos discretizados esparsos, que podem variar com o tempo, em um grid tridimensional.

A estrutura de dados foi desenvolvida pela DreamWorks Animation, e oferece um espaço de índices de três dimensões virtualmente infinito, armazenamento compacto, tanto na memória quanto em disco, acesso rápido aos dados, tanto sequencial quanto aleatório, e uma coleção de algoritmos otimizados especificamente para esta estrutura de dados para tarefas usuais como a aplicação de filtros, discretização de equações diferenciais parciais, conversão de polígonos em voxels, e amostragem. Os detalhes técnicos da estrutura de dados e de sua manipulação são discutidos nas seções seguintes.

3.1 Definição

A biblioteca OpenVDB é composta pela estrutura de dados e pelo conjunto de ferramentas para a manipulação da estrutura. Muito embora haja grande interdepeência entre estes dois itens, abordaremos cada item separadamente.

3.1.1 A Estrutura de Dados VDB

Uma das principais idéias em que a VDB se baseia é em organizar dinamicamente os blocos de dados de um grid em uma estrutura de dados hierárquica semelhante a uma árvore B+. Essa semelhança será abordada na seção 3.2.3. Estes blocos são folhas[2] que sempre ficam em uma mesma profundidade de um grafo conectado e acíclico com fatores grandes, e variáveis, de ramificação, sendo que os fatores de ramificação são necessariamente

[1] www.openvdb.org

[2] **Folhas**, ou nó-folha, são os nós de uma estrutura de árvore que não possui filhos e que assim diferem da raiz e dos nós internos.

potências de dois. Isso implica que a árvore é balanceada em altura por construção, mas baixa e larga. Como resultado, temos um domínio mais amplo, e devido à altura baixa da árvore, o número de operações de E/S necessárias para percorrer a árvore da raiz até uma das folhas é menor.

Templates C++ são usados frequentemente na construção da estrutura, bem como funções inline, e funções virtuais são evitadas a todo custo. Trata-se de um pequeno conjunto de otimizações que, entre outras coisas, viabiliza o acesso aleatório rápido. Mais especificamente, as classes que implementam os diferentes nós da árvore são recursivamente geradas a partir de templates em função do tipo armazenado pelos nós-filho, ao invés de usar herança de uma classe base comum a todos os nós. A motivação para esta escolha é o fato de que os nós gerados a partir de templates são expandidos inline durante a compilação, e com isso, as funções não possuem um overhead de execução, que ocorreria se a opção pelo uso de herança fosse feita.
Além disso, ao usarmos templates, fica mais fácil mudarmos a configuração de profundidade e ramificação da árvore, apesar de a escolha destas características precisar ser feita em momento de compilação, e não de execução.

Outras técnicas de implementação que contribuem ao desempenho da estrutura são operações bit a bit, que são muito rápidas, armazenamento de informações usando bits, uso do recurso de `union` do C++ para reuso de memória, e uso de execução paralela com suporte explícito a *threads* e operações SIMD[3].

Detalhes de implementação

A VDB modela um espaço de índices (x, y, z) infinito, embora na prática este seja limitado pela precisão de bits da arquitetura e da memória disponível. Os dados armazenados na estrutura consistem em um tipo `Valor`, definido com o uso de `templates`, e em uma coordenada com os índices discretos (x, y, z), especificando a localização espacial do ponto amostral, isto é, a *topologia* do *valor* dentro da estrutura da árvore. Conforme comentado anteriormente, a menor unidade de um elemento de volume será sempre chamada de *voxel*. Um único *valor* é associado a cada voxel. E cada voxel pode existir em um de dois estados possíveis - são eles: *ativo* e *inativo*. A interpretação deste estado binário depende da aplicação que estiver usando a biblioteca, mas podemos considerar que os voxels ativos são mais importantes, ou de maior interesse, para a aplicação.

A VDB armazena separadamente a *topologia* de um voxel numa árvore cujo nó raiz cobre todo o espaço de índices e cujas folhas cobrem um subconjunto fixo do espaço de índices. Mais precisamente, a topologia é armazenada implicitamente em máscaras de bit, e os

[3]**Single instruction, multiple data**, única instrução, dados múltiplos, em inglês, é uma classe de computadores paralelos na taxonomia de Flynn. Refere-se ao suporte de operações em que uma instrução é executada paralelamente em vários elementos de dados simultaneamente.

valores associados a cada voxel são armazenados de maneira explícita em buffers que podem estar localizados em qualquer nível da árvore - dizemos que a estrutura é multiníveis por isso. Áreas do espaço de índices em que todos os voxels tem o mesmo valor podem ser representados com uma única entrada no nível correto da árvore. Assim como voxels, os valores armazenados nos níveis intermediários da árvore também podem estar ativos ou inativos. O objetivo da VDB é usar o mínimo de memória possível para representar os voxels ativos, mantendo as características de desempenho de uma estrutura de dados para volumes densos.

Um componente fundamental da estrutura de dados é o conjunto de máscaras de bit incluídas nos vários nós da árvore, em diferentes níveis. O uso das máscaras de bit permitem acesso rápido e direto à representação binária da *topologia* local ao nó. Um diagrama mostrando a estrutura interna da VDB é apresentado na Figura 3.1.

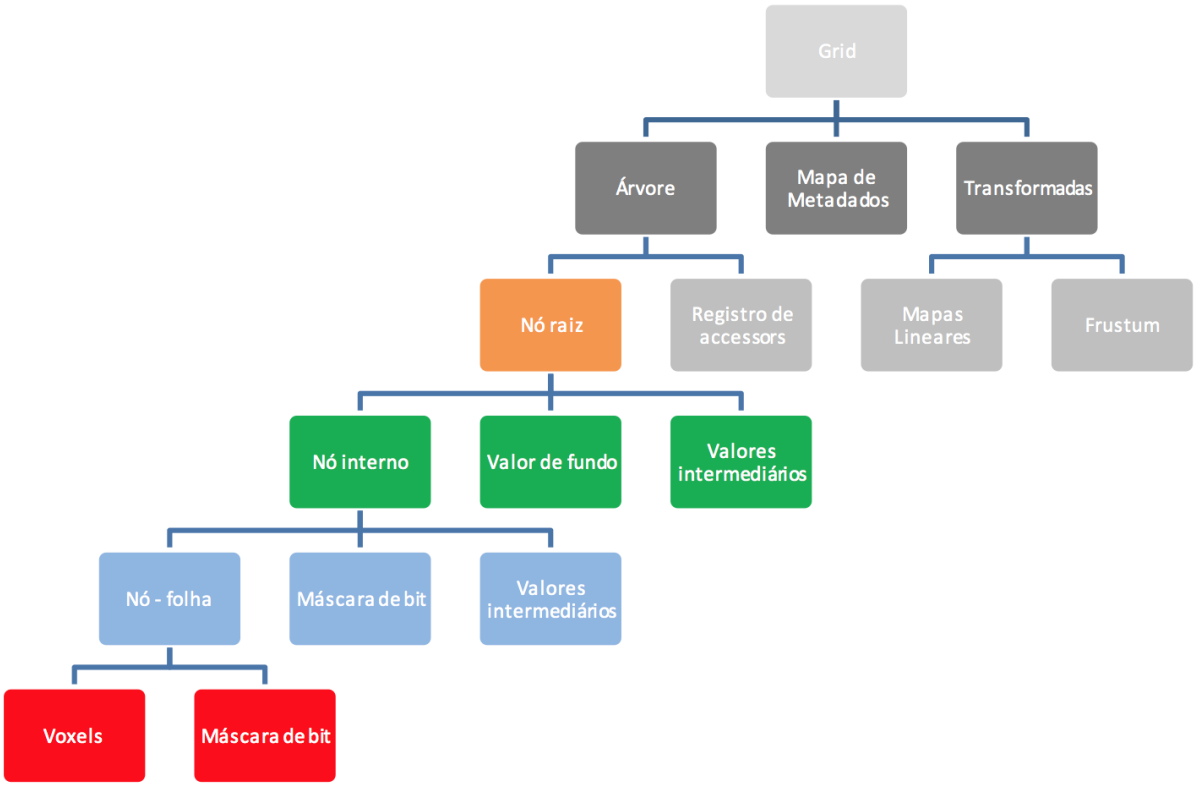

Figura 3.1: Diagrama da estrutura interna da árvore VDB.

*F*olhas. Estes nós são os blocos do grid de nível mais baixo na estrutura, e por construção, todos estão na mesma profundidade da árvore.As folhas dividem o espaço de índices em subdomínios disjuntos com 2^{log2w} voxels em cada eixo, onde $2^{log2w} = 1, 2, 3, ...$ e w é qualquer entre x, y ou z. Uma configuração típica, e recomendada, é estabelecer $2^{log2w} = 3$, o que corresponde a um bloco $8 \times 8 \times 8$. As dimensões das folhas (e dos nós internos, que discutiremos a seguir) são restritas a potências de dois, pois isso permite as operações bit

a bit com mais velocidade durante as buscas pela árvore.

Código 3.1: Definição de um nó folha.

```
template<class Value, int Log2X, int Log2Y=Log2X, int Log2Z=Log2Y>
class LeafNode {
  static const int sSize=1<<Log2X+Log2Y+Log2Z,
  sLog2X=Log2X, sLog2Y=Log2Y, sLog2Z=Log2Z;
  union LeafData {
    streamoff offset; //out of core streaming
    Value* values; //temp buffers
  } mLeafDAT; //direct access table

  BitMask<sSize> mValueMask; //active states
  [BitMask <sSize> mInsideMask]; //optional for level sets
  uint64_t mFlags; //64 bit flags
};
```

Como pode ser visto no Código 3.1, os tamanhos são estabelecidos em momento de compilação, e o tamanho do nó é computado como $1 << \sum_w sLog2w$, onde $<<$ denota um deslocamento de bits à esquerda. As folhas armazenam os valores dos voxels em uma tabela chamada de *Direct Access Table*[4], denotada no código por `mLeafDAT`, e a topologia do voxel ativo é mantido na máscara de bits denotada no código por `mValueMask`. É importante notar que embora a máscara de bits tenha um tamanho fixo igual ao da estrutura `LeafNode`, o tamanho do vetor de armazena os valores dos dados é dinâmico, mantido em `mLeafDAT.values`. A quantidade variável de buffers de dados, e seus tamanhos, bem como outras informações a respeito da `LeafNode` é armazenada de forma compacta na varíavel de 64 bits `mFlags`, mostrado na Figura 3.2.

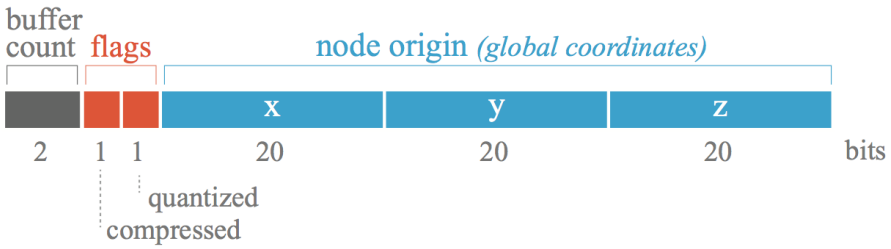

Figura 3.2: Representação compacta de `mFlags`, variável de 64 bits que codifica: número de buffers(2), compressão(1), quantização(1) e origem da folha (3×20).

Os primeiros dois bits codificam um dos quatro estados possíveis: 0 buffers, isto é, os valores são salvos externamente, 1 buffer, valores salvos na memória sem suporte para integração temporal, 2 buffers, valores salvos na memória com suporte a integração temporal de primeira e segunda ordem ou 3 buffers, valores salvos na memória com suporte a integração temporal de terceira ordem. O terceiro bit é 1 se o bloco é comprimido, e o

[4]Pode-se assumir que esta tabela é um vetor com custo de acesso aleátorio de $O(1)$ para o pior caso.

quarto bit é 1 se se a folha é quantizada. Finalmente, os 3×20 bits restantes são usados para armazenar a origem do nó no grid. As coordenadas globais do voxel podem ser obtidas ao combinar a topologia local do voxel salva em `mValueMask` com a origem salva em `mFlags`. Dessa forma, as folhas são auto-suficientes e não precisam de referências para os nós pai.

Nós internos. São os nós existentes em todos os níveis intermediários da árvore entre a raiz e as folhas, e basicamente definem a profundidade e o formato da árvore B+ modificada.

Código 3.2: Definição de um nó intermediário.

```
template <class Value, class Child, int Log2X, int Log2Y=Log2X, int Log2Z
    =Log2Y>
class InternalNode {
  static const int sLog2X=Log2X+Child::sLog2X,
           sLog2Y=Log2Y+Child::sLog2Y,
           sLog2Z=Log2Z+Child::sLog2Z,
           sSize=1<<Log2X+Log2Y+Log2Z;
  union InternalData {
    Child* child; //child node pointer
    Value value; //tile value
  } mInternalDAT[sSize];
  BitMask<sSize> mValueMask; //active states
  BitMask<sSize> mChildMask; //node topology
  int32_t mX, mY, mZ; //origin of node
}
```

Vários detalhes da implementação são iguais aos da folha. Os fatores de ramificação são configuráveis através dos parâmetros de `template`, Log2w, e são limitados a potências de dois para buscas eficientes na árvore. Mas diferentemente das folhas, os nós internos armazenam tanto valores quanto informações de topologia, isto é, ponteiros para outros nós, internos ou folhas. A implementação é feita com o uso de `union` na tabela de acesso direto `mInternalDAT`. A topologia correspondente é armazenada de forma compacta na máscara de bits `mChildMask`, e `mValueMask` é usada para indicar se o valor intermediário está ativo ou não. Como os fatores de ramificação Log2w são fixados na compilação, os tamanhos de `mInternalDAT`, `mChildMask` e `mValueMask` também são. Outra observação importante é que nós internos em níveis diferentes da árvore podem ter fatores de ramificação diferentes.

Nó raiz. Este é o nó no nível mais alto da árvore, onde as operações sobre a árvore normalmente começam.

Código 3.3: Definição do nó raiz.

```
template<class Value, class Child>
class RootNode {
  struct RootData {
```

```
    Child* node; // =NULL if tile
    pair<Value, bool> tile; // value and state
  };
  hash_map<RootKey, RootData, HashFunc> mRootMap;
  mutable Registry<Accessor> mAccessors;
  Value mBackground; // default background value
}
```

Todas as configurações de uma VDB possuem no máximo um nó raiz que, diferentemente dos outros nós, é esparso e pode ter seu tamanho alterado dinamicamente. Isso é facilitado por uma tabela hash-map que armazena os ponteiros para os nós filhos ou para os valores intermediários, se forem armazenados neste nível. Se uma entrada na tabela representar um valor intermediário (como no caso de `child=NULL`), um valor booleano indica o estado do valor intermediário (se está ativo ou inativo). É importante notar que por construção o `mRootMap` contém poucos valores devido aos domínios imensos representados pelos nós intermediários, como por exemplo, 4096^3. A raiz possui um registro de classes de acesso denominada `Accessors`, que melhora consideravelmente o desempenho de acesso ao grid nos casos de consultas espacialmente coerentes ao reutilizar pointeiros de nós salvos no cache na ocasião de um acesso anterior e com isso consegue realizar a busca de baixo para cima, ao invés de percorrer a árvore de cima para baixo. `mBackground` é o valor que é retornado quando é feita uma tentativa de acesso a um local que não pertence a nenhum valor intermediário e a nenhum voxel.

3.1.2 Algoritmos de Acesso à Estrutura

São três os tipos de acesso a uma estrutura de dados espacial: aleatório, sequencial e por estêncil. A implementação de cada uma destas formas de acesso será discutida em seguida.

Acesso aleatório

O tipo de acesso mais fundamental, e discutivelmente o de mais difícil implementação, é o acesso aleatório a voxels arbitrários. O que separa este tipo de acesso dos demais é que, no pior caso, cada acesso requer que árvore seja percorrida por inteira, de cima para baixo, começando na raiz e possivelmente terminando numa folha. Na prática, no entanto, o acesso aleatório pode ser melhorado se a ordem de busca na árvore for invertida. Deste modo, é mais fácil definir o acesso aleatório em função dos demais tipos de acesso: acesso aleatório é o tipo de acesso que não é sequencial e não é baseado em estêncis. Estes dois últimos tipos de acesso são facilmente definidos como acesso com um padrão fixo, definido pela organização dos dados na memória ou algum padrão de estêncil.

Consulta aleatória é o tipo mais comum de acesso aleatório e será o primeiro a ser apresentado. Começamos identificando as operações necessárias para percorrer a árvore, começando pela raiz. Como estas operações dependem da implementação do `mRootMap`,

verificamos primeiro o `std::map` da raiz. Para acessar o voxel nas coordenadas de índice (x, y, z), a chave `rootKey`, mostrada no código 3.4, é calculada. Como é natural do C++, & e ~ denotam as operações lógicas E e NÃO aplicáveis aos bits. Após a compilação, isso se reduz a três operações E a nível de hardware, que mascaram os bits de ordem mais baixa, correspondentes ao espaço de índice dos nós-filhos. Os valores resultantes são as coordenadas da origem do nó-filho que contém (x, y, z), e isso acaba por evitar colisões de hash.

Código 3.4: Cálculo da chave `RootKey`.

```
int rootKey[3] = {x & ~((1 << Child::sLog2X) - 1),
            y & ~((1 << Child::xLog2Y) - 1),
            z & ~((1 << Child::sLog2Z) - 1)};
```

Esta chave, garantida então de ser livre de colisões, é usada para realizar a consulta em `mRootMap` que armazena a `RootData` em ordem lexicográfica das três componentes da chave `rootKey`. Se nenhuma entrada é encontrada, o valor padrão (`mBackground`) é retornado, e se um valor intermediário é encontrado, esse valor de nível mais alto é retornado. Em qualquer caso, a busca termina. No entanto, se um nó-filho for encontrado, a busca continua até que um valor seja encontrado ou uma folha seja alcançada. A única modificação necessária para substituir o `std::map` com um hash-map. Para isso, combina-se a `rootKey` dada anteriormente com a função de hash mostrada no código 3.5.

Código 3.5: Função de hash.

```
unsigned int rootHash = ( (1 << Log2N) - 1) &
            (rootKey[0] * 73856093 ^
              rootKey[1] * 19349663 ^
              rootKey[2] * 83492791);
```

Neste código, as três constantes são números primo bastante grandes, ^ é o operador ou-exclusivo (XOR) e *Log2N* é uma constante estática que estima o logaritmo de base dois do tamanho de `mRootMap`.

Código 3.6: Cálculo do offset interno.

```
unsigned int internalOffset =
(((x & (1 << sLog2X) - 1) >> Child::sLog2X) << Log2YZ) +
(((y & (1 << sLog2Y) - 1) >> Child::sLog2Y) << Log2Z) +
((z & (1 << sLog2Z) - 1) >> Child::sLog2Z);
```

Quando um nó interno é encontrado (no topo ou em níveis internos à árvore), o offset de acesso mostrado no Código 3.6 é calculado a partir das coordenadas globais do grid, onde $Log2YZ = Log2Y + Log2Z$. No momento da compilação, isso se reduz a apenas três operações lógicas E, cinco deslocamentos de bit, e duas somas. Em seguida, se o

bit `internalOffset` da máscara `mChildMask` não estiver setado, então (x, y, z) está dentro de uma área com valor constante, e este valor é retornado da `mInternalDAT` e a busca é encerrada. Caso contrário, o nó-filho é extraído de `mInternalDAT` e a busca continua até que um bit zero é encontrado na máscara `mChildMask` de um nó interno `InternalNode` ou uma folha `LeafNode` é alcançada. O offset de acesso direto para uma folha é ainda mais rápido de ser calculado, como mostrado no Código 3.7. Isso porque o código se reduz a três operações E bit a bit, dois deslocamentos à esquerda e duas somas.
É importante notar que, com exceção das três multiplicações na função hash, todas as operações em bits são realizadas com uma única instrução, ao invés de multiplicações, divisões e módulos - consequência direta do uso de fatores de ramificação que são potências de dois. Além disso, todas as chaves e offsets são calculados a partir das mesmas coordenadas globais (x, y, z), ou seja, os cálculos são independentes e não-recursivos, o que é uma consequência do fato de a profundidade da árvore e os fatores de ramificação serem conhecidos no momento da compilação.

Código 3.7: Cálculo do offset para a folha.

```
unsigned int leafOffset =
  ((x &) 1 << sLog2X) - 1) << Log2Y + Log2Z) +
  ((y &) 1 << sLog2Y) - 1) << Log2Z) + (z & (1 << sLog2Z) - 1);
```

Como a VDB é balanceada em altura por construção com uma profundidade imutável durante a execução, todo caminho entre a raiz `RootNode` até qualquer uma das folhas `LeafNode` é igualmente longo e podemos concluir que que toda consulta aleatória leva o mesmo tempo de pior caso. Acesso aleatório a valores intermediários armazenados em níveis mais rasos da árvore é mais rápido já que a busca se encerra antes.

Inserção aleatória é normalmente usada quando os grids estão sendo inicializados, mas pode ser importante também para dados dinâmicos e simulações. A busca na árvore é feita como discutido anteriormente, mas agora um nó é alocado se o bit correspondente não estiver setado em alguma `mChildMask`. A busca termina em uma folha, possivelmente recém-construída, com o valor do voxel atribuído no buffer apropriado e o bit correspondente setado em `mValueMask`. Como os nós abaixo da raiz `RootNode` são alocados apenas na inserção, o consumo de memória para dados volumétricos esparsos é baixo. E embora esta alocação dinâmica de nós implique que a inserção aleatória seja mais lenta que a consulta aleatória, o *overhead* é tipicamente amortizado após várias operações de inserção com coerência espacial.

Remoção aleatória é outro exemplo de uma operação que requer eficiência quando lidamos com dados dinâmicos. A busca é implementada de maneira semelhante à inserção, mas agora os bits na `mChildMask` e na `mValueMask` são desmarcados e os nós são removidos se não possuírem outros filhos ou outros nós ativos.

Finalmente, concluímos que a VDB suporta operações aleatórias como consulta, inserção e remoção em tempo constante e isso é, na média, independente da topologia ou resolução do conjunto de dados que estão sendo armazenados.

Acesso sequencial

Muitos algoritmos acessam ou modificam todos os voxels ativos em um grid, mas não dependem da sequência em que isso acontece. Em particular, isso é verdade para a maioria das simulações dependentes do tempo, como advecção de fluidos. Esta invariância pode ser explorada se um padrão de acesso sequencial puder ser definido que tenha um desempenho superior àquele do acesso aleatório. *Acesso sequencial*, portanto, é como nos referimos à sequência óptima de acesso aos dados, dada pela ordem em que os dados estão fisicamente dispostos na memória. Com os avanços em hierarquias sofisticadas de cache e algoritmos de pré-captura de instruções (*prefetching*) em CPUs modernos, torna-se especialmente importante obter e processar os dados na ordem em que estão armazenados na memória.

O desafio é implementar o acesso sequencial de modo que ele seja mais rápido que o acesso aleatório rápido discutido na seção anterior. O problema se resume a localizar o próximo valor ativo ou nó-filho, e a solução é bastante simples: percorrer as máscaras compactas de acesso direto `mChildMask` e `mValueMask` em cada nó. Como são estruturas compactas, são facilmente salvos em cache, e várias entradas (bits) podem ser processados simultaneamente. Como as máscaras de bit estão em todos os nós de uma árvore VDB, podemos combinar iteradores em vários níveis da árvore, permitindo a criação de iteradores que percorrem valores de voxels, valores internos ativos, folhas, entre outras possibilidades.

Acesso por Estêncil

Acesso eficiente por estêncil em grids uniformes é um requisito fundamental para cálculos de diferença finita. Esses métodos aproximam operadores diferenciais com diferenças discretas de valores em um grid numa vizinhaça local denominada estêncil de suporte. Outras aplicações deste tipo de acesso são interpolação e aplicação de filtros com *kernels* de convolução de suporte local. Os tamanhos e formas destes estêncis pode variar bastante dependendo da técnica aplicada. Normalmente, estes métodos de acesso são combinados com acesso sequencial, originando os iteradores de estêncil, um conceito essencial para muitas simulações numéricas.

3.2 Comparação com Estruturas Semelhantes

3.2.1 Octrees

Uma *octree* é uma estrutura de dados em árvore em que cada nó interno possui exatamente 8 nós-filho. O uso mais comum da estrutura é em computação gráfica para particionar o espaço tridimensional subdividindo-o em oito octantes, de forma recursiva, como mostrado na Figura 3.3.

No contexto de renderização, modelagem 3D e extração de malhas, esta estrutura é amplamente utilizada. E como citado na seção 3.1.1, a VDB mantém blocos organizados dinamicamente numa estrutura hierárquica, de modo que estes blocos são folhas da estrutura de árvore e que ficam todas no mesmo nível de profundidade de um grafo acíclico e conectado, com fatores de ramificação grandes, além de variáveis em momento de compilação. Logo, a VDB é balanceada em altura por construção. Octrees, por outro lado, normalmente são árvores altas, com um fator de ramificação limitado a 2 em cada eixo ($2^3 = 8$). Uma característica da octree adotada pela VDB é a capacidade de armazenar valores em nós que não são folhas, funcionando como um grid multi-nível adaptativo.
Contudo, uma limitação da VDB é que, apesar de ser uma estrutura de dados hierárquica, ela provavelmente não é a melhor escolha para métodos de amostragem multi-resolução devido aos elevados fatores de ramificação entre os níveis da árvore, e este é um dos motivos que a VDB não substitui octrees quando uma adaptatividade óptima é desejada.

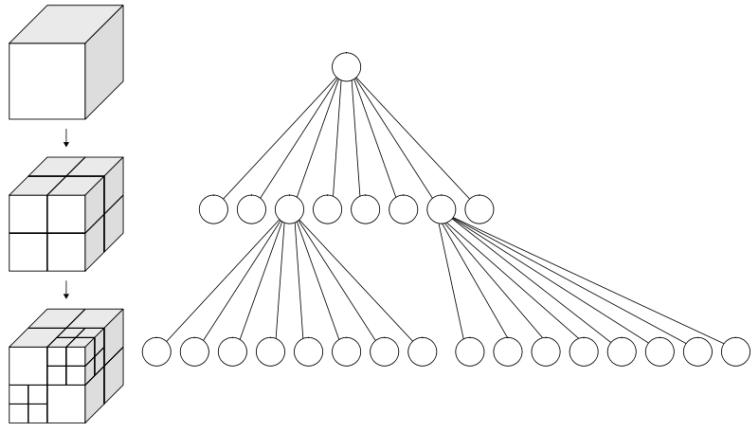

Figura 3.3: Diagrama de uma octree, estrutura em que cada nó interno possui exatamente oito nós-filho.

3.2.2 Árvores B

Comentaremos brevemente acerca de árvores B para contextualizar a discussão de árvores B+, uma variação da árvore B, na seção 3.2.3.

Árvore B é uma estrutura de dados que mantém os dados ordenados e permite buscas, acesso sequencial, inserções e remoções em tempo $O(log\ n)$. Trata-se de uma generalização da árvore binária de busca, sendo que esta, a árvore B, pode ter mais de dois filhos por nó. Além disso, esta árvore é otimizada para leitura e escrita de grandes volumes de dados, e por isso tem ampla aplicação em bancos de dados de sistemas de arquivos.

3.2.3 Árvores B+

Uma árvore B+ é um árvore n-ária com um número variável, e normalmente grande, de filhos por nó. Uma árvore B+ consiste de raiz, nós internos e folhas. A raiz pode ser uma folha ou um nó com dois ou mais filhos.
Uma árvore B+ é uma árvore B em que cada nó que não seja uma folha possui apenas uma chave, e não pares chave-valor, com um nível a mais nas folhas. A principal aplicação de uma árvore B+ é armazenar dados para consultas rápidas em um contexto orientado a blocos de dados, como sistemas de arquivos.

A VDB é basicamente uma variação da árvore B+ com características de Octrees. A VDB preserva as características de desempenho das árvores B+ que resultam do uso de fatores de ramificação variáveis e grandes, mas com otimizações específicas à aplicação volumétrica. Onde, na VDB, os valores armazenados no grid são indexados por suas coordenadas espaciais em todos os nós da árvore, uma árvore B+ convencional armazena dados genéricos, isto é, de qualquer aplicação, indexados por chaves em um contexto orientado a armazenamento por blocos e somente nas folhas. Em outras palavras, VDB é uma estrutura de dados multi-nível e não usa chaves no sentido tradicional como a árvore B+ o faz. Além disso, em várias implementações as folhas de uma árvore B+ são ligadas umas às outras para permitir uma iteração sequencial rápida, diferentemente da VDB que opta por salvar em cache os caminhos de busca percorridos. Finalmente, nas árvores B+ o custo de uma busca aleatória é logarítmico, ao passo que com a VDB o custo de busca é constante.

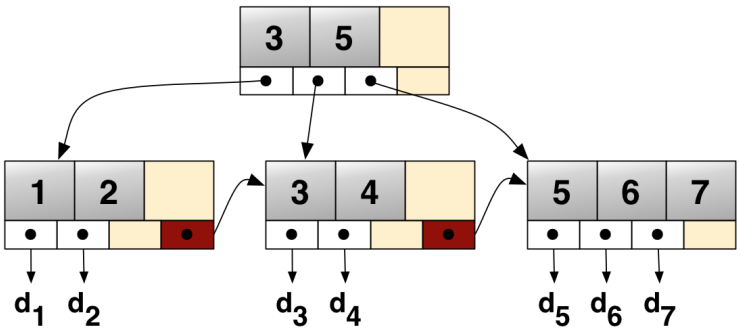

Figura 3.4: Diagrama de uma árvore B+. Os números de 1 a 7 são chaves, e os dados armazenados na árvore estão identificados de d_1 a d_7.

Capítulo 4

Resultados

Capítulo 5

Conclusão