



UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

RAFAEL CERQUEIRA DE CAMPOS

IMPLEMENTAÇÃO DE TEXTURAS 3D PARA
RENDERIZAÇÃO DE VOLUMES NO BLENDER

Orientador: Prof. Dr. Mario Augusto de Souza Liziér

SÃO CARLOS
2013

Conteúdo

1	Introdução	4
1.1	Motivação	6
1.2	Organização	7
2	Blender	8
2.1	Software	8
2.1.1	Renderização no Blender	9
2.2	Texturas	11
2.2.1	Definição	11
2.2.2	Mapeamento de Textura	12
2.2.3	Estratégias de <i>Wrapping</i>	13
2.2.4	Texturas Tridimensionais	14
2.3	Sistemas de Coordenadas	24
2.3.1	Notação Homogênea	26
2.4	<i>Shaders</i>	27
2.4.1	<i>Open Shading Language</i>	28
3	OpenVDB	29
3.1	Definição	29
3.1.1	A Estrutura de Dados VDB	29
3.1.2	Algoritmos de Acesso à Estrutura	34
3.2	Comparação com Estruturas Semelhantes	37
3.2.1	Octrees	37
3.2.2	Árvores B	37
3.2.3	Árvores B+	38
4	Resultados	39
5	Conclusão	42
5.1	Desafios	42
5.2	Análise dos Resultados	44
5.3	Trabalhos Futuros	44
5.4	Comentários Finais	45

Capítulo 1

Introdução

Em computação gráfica, objetos tridimensionais são geralmente representados apenas por sua superfície, por meio de malhas poligonais ou retalhos *NURBS*¹. Quando estes modelos de representação são utilizados, as propriedades visuais do objeto, 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. O que ocorre nestes casos é o cálculo do transporte de luz estritamente nos pontos da superfície do modelo. Logo, estes métodos não são capazes de avaliar as interações de luz que ocorrem no interior do objeto.

Comparada à renderização⁴ 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 ressonância magnética. Outros exemplos são dados de dinâmica dos fluidos computacional, dados geológicos, superfícies implícitas, entre outros.

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

O desafio de renderizar volumes está principalmente na avaliação da interação entre a

¹*Non-Uniform Rational B-Spline*, na sigla em inglês, é um modelo matemático usado para gerar e representar curvas e superfícies.

²*Shaders*, de modo bem simplista, são sub-programas executados, em geral, nos processadores gráficos para a geração de imagens. Uma discussão mais extensa é apresentada na Seção 2.4.

³*Bidirectional Reflectance Distribution Function*, na sigla em inglês, é uma função de quatro dimensões que define como a luz é refletida em uma superfície opaca.

⁴*Rendering*, processo de síntese de imagem a partir de cenas virtuais. Sem tradução direta, será utilizada uma adaptação para o português.

luz 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 tridimensional preenchido com o gás, tornando a renderização volumétrica uma tarefa com um alto custo computacional.

Como discutido, 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 imagens (ou *grids*) tridimensionais regulares e espacialmente uniformes, em parte porque tais representações são computacionalmente simples. *Grids* serão especificados mais detalhadamente na Seção 2.2.4.

Abordagem básica à renderização de volumes

O processo de renderização de volumes, tal como tratado neste trabalho, foi dividido nas seguintes 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 da imagem discreta 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 avaliadas. 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.

Software-Alvo

O Blender⁵ é um pacote de software de computação gráfica 3D, gratuito e de código aberto, utilizado principalmente para criação de animações e efeitos visuais, modelos tridimensionais impressos, aplicações 3D interativas e jogos digitais. Para tais aplicações, o

⁵<http://www.blender.org>

Blender 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 GPUs⁶. Como o software existe há pouco tempo, há muitas funcionalidades desejáveis que ainda não foram implementadas, e uma delas é a renderização de volumes.

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 sobre a OpenVDB é apresentado no Capítulo 3.

Definição do problema

O objetivo deste trabalho é implementar a texturização de volumes no renderizador Cycles do Blender, o que corresponde as duas primeiras etapas descritas no início dessa Seção, ou seja, de *consulta aos dados* de uma estrutura volumétrica e da 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.

1.1 Motivação

A implementação de renderização de volumes é importante pelo número de aplicações, e efeitos como os fenômenos naturais (nuvem, névoa, entre outros) são difíceis de gerar com outras técnicas. E, no caso do Blender, cujo uso principal é na criação de animações, os volumes tendem a ser muito grandes, o que atualmente torna o processo de geração das imagens proibitivamente lento. Neste contexto, o uso de uma biblioteca que tenha sido extensivamente otimizada com este uso em mente foi a proposta com a maior adesão de usuários interessados. E embora existam outras bibliotecas que resolvem o mesmo

⁶ *Graphics Processor Unit*, na sigla em inglês.

problema, como a Field3D⁷, os resultados mais convincentes foram os apresentados pela OpenVDB.

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 contribuir em um dos 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 realizada pelo Google e as instituições participantes, este projeto foi um dos trabalhos escolhidos. Ele foi desenvolvido com o acompanhamento de um desenvolvedor da Blender Foundation e financiado pela bolsa do Google.

1.2 Organização

No Capítulo 2, o software Blender é apresentado, juntamente com a descrição de alguns conceitos, como o mapeamento de textura, interpolação, e sistemas de coordenadas. Em seguida, no Capítulo 3, a estrutura de dados utilizada para manipular os volumes de dados é descrita. No Capítulo 4 os resultados obtidos são apresentados e por fim, no Capítulo 5, encontram-se as discussões finais e conclusão deste trabalho.

⁷<http://opensource.imageworks.com/?p=field3d>

Capítulo 2

Blender

2.1 Software

O Blender¹ 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-GPL², 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

¹www.blender.org

²<http://www.gnu.org/licenses/gpl.html>

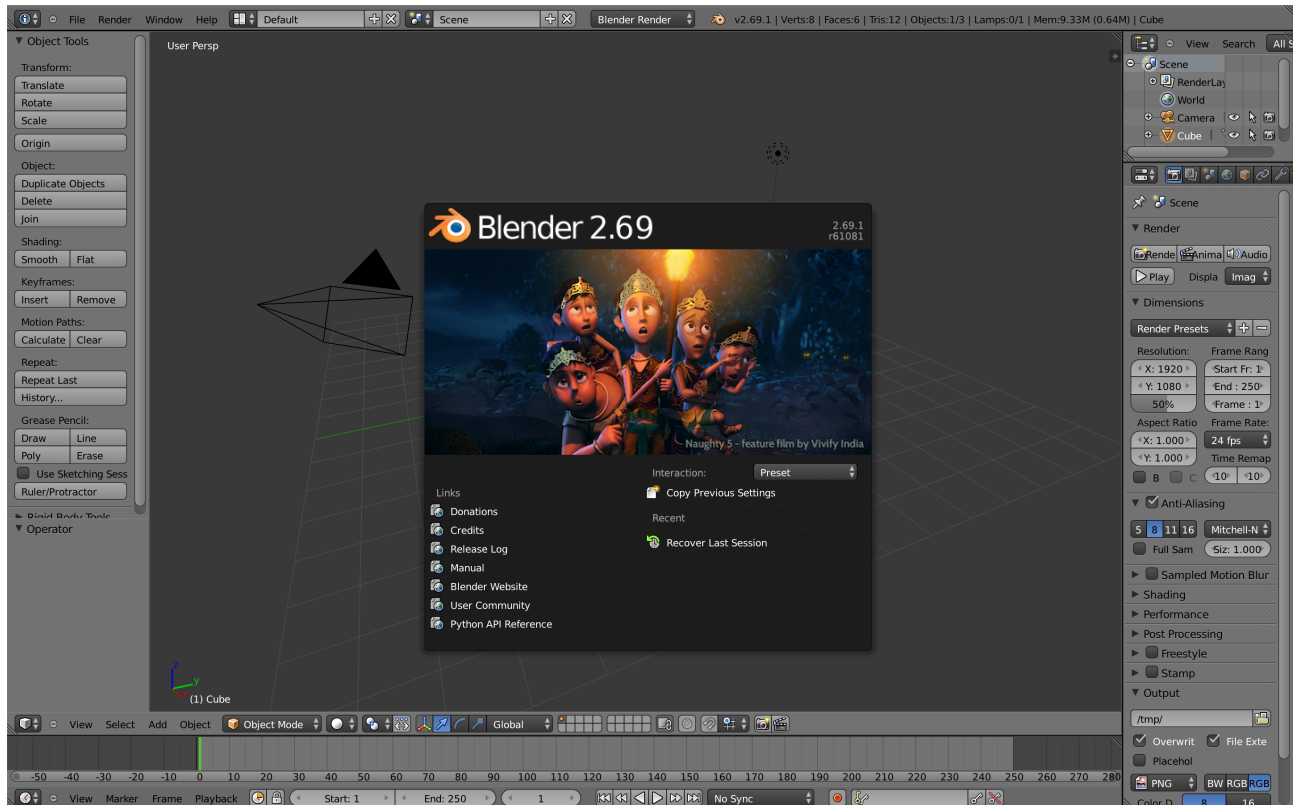


Figura 2.1: Interface do Blender, ao abrir a aplicação.

de ponteiros, versão do software, e disposição de memória³.

Após o cabeçalho do arquivo, blocos de dados são salvos com informações como códigos identificadores dos objetos na cena, tamanho em memória dos objetos 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, 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 Blender 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

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

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

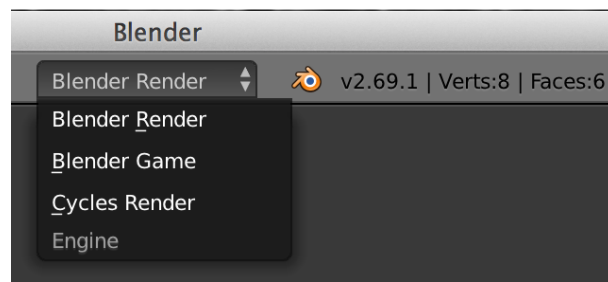


Figura 2.2: Interface do Blender, que permite a escolha de qual engine usar para renderização da cena.

É possível pré-visualizar a cena, animada ou estática, nas *viewports* da aplicação, e estas podem ser em (i) *wireframe*, mostrando apenas 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 fiel⁴, 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 cáticos 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.

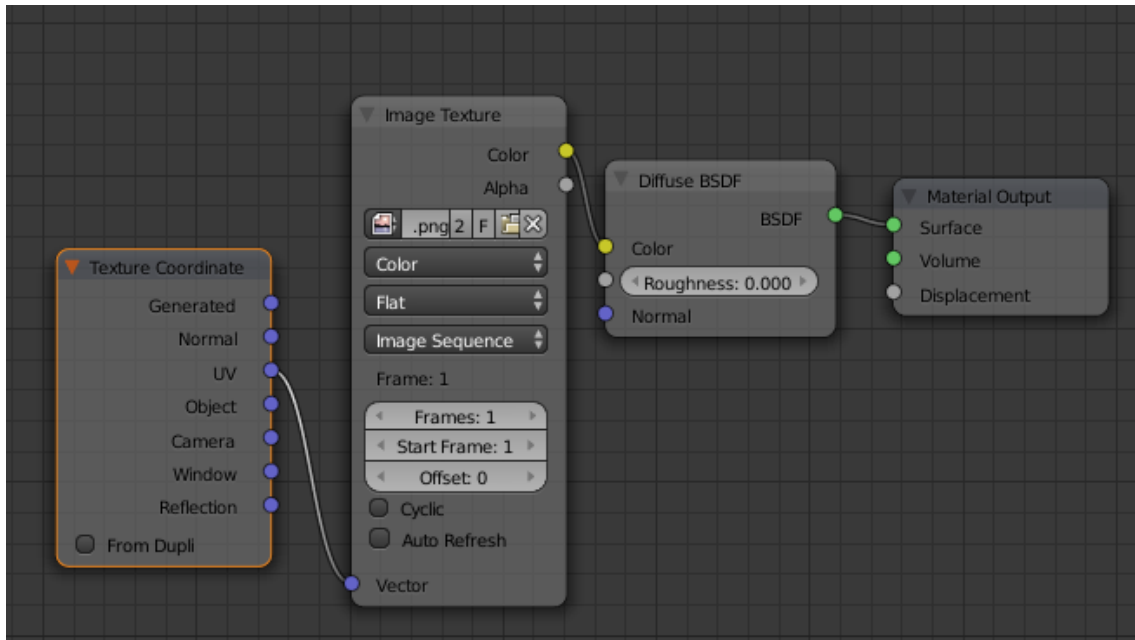


Figura 2.3: Rede de nós configurando o uso de uma sequência de imagens como textura para uma superfície.

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*.

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 homogêneas, 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

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

Neste conjunto de equações, a , b , c , d , e e f são coeficientes que definem o mapeamento linear. E se tivermos que $ae \neq bd$, este mapeamento é invertível.

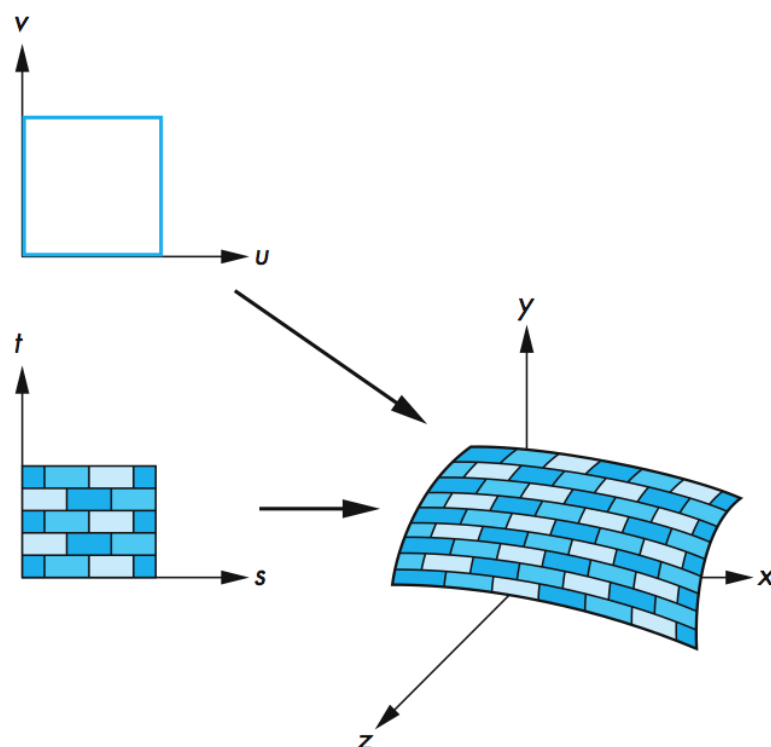


Figura 2.4: Mapeamento do sistema de coordenadas da região parametrizada para o sistema de coordenadas da textura.

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

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

que é uma função do espaço 3D para um valor de apenas um componente. Na prática, um campo volumétrico é dado por uma imagem tridimensional, pois trata-se do resultado de uma simulação ou de uma medição. A Figura 2.5 mostra um *grid* bidimensional e outro tridimensional, exemplificando os dados para os campos escalares discretos e uniformemente amostrados.

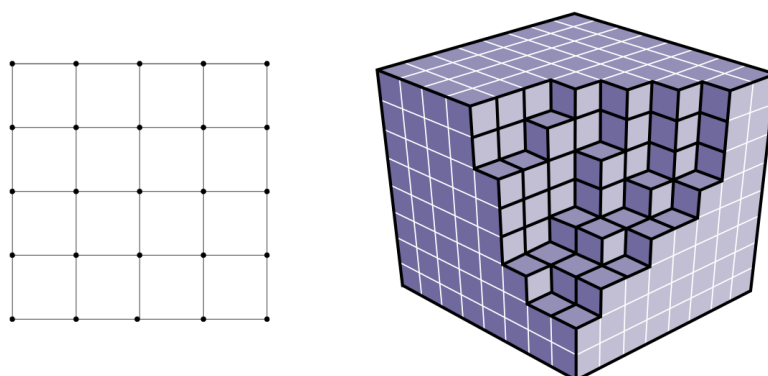


Figura 2.5: Exemplos de *grids* uniformes: bidimensional com células de razão de aspecto 1, à esquerda, e tridimensional com células cubóides, à direita.

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 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, ou seja, sem dimensão, sendo necessário o uso de técnicas de interpolação para preencher 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 células. Porém, conforme será discutido no Capítulo 3, essa representação só é eficiente 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 volumes 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 $B_{i,j,k}$. Na implementação, 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, a visualização de imagens médicas é uma das aplicações mais proeminentes em visualização. Para obter os dados tridimensionais, usa-se equipamentos de escaneamento não invasivos, como tomografia computadorizada ou ressonância magnética.

Outra fonte corriqueira de dados volumétricos são as simulações numéricas, 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.

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:

$$f(t) = \sum_i f_i \operatorname{sinc}(\pi(v_s t - i)).$$

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

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

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.

Embora tenhamos discutido funções de apenas uma variável, podemos estender 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

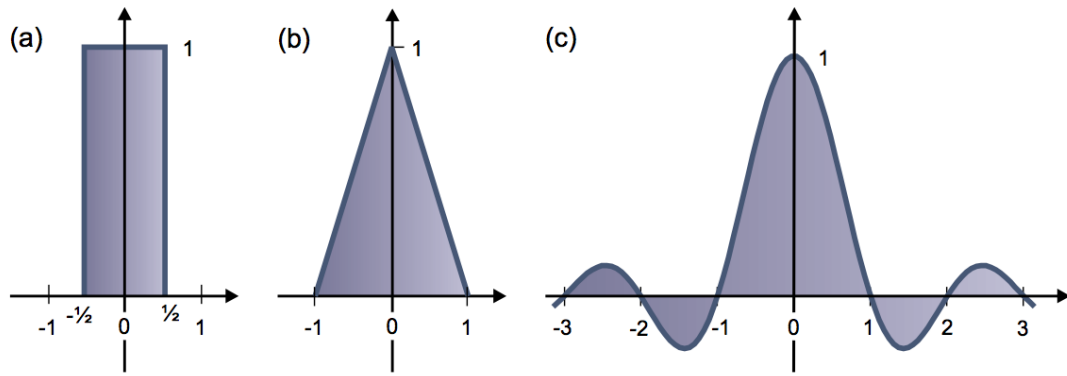


Figura 2.6: Exemplos de filtros de reconstrução: (a) filtro caixa, (b) filtro triangular, e (c) filtros que usam o seno cardinal, *sinc*.

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 . É 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.

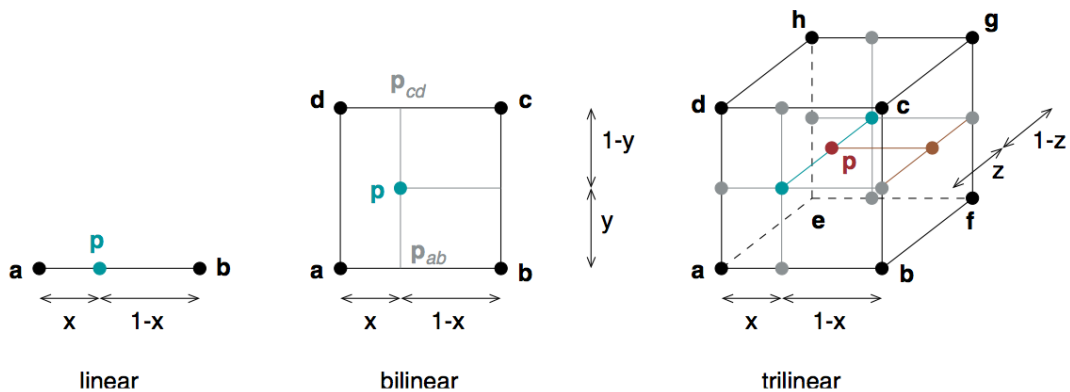


Figura 2.7: Interpolações lineares usando o produto tensorial: casos linear, bilinear e trilinear.

Filtros e Técnicas de Interpolação

Quando desejamos aplicar uma textura 2D à superfície de uma 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. 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 2.3: 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];
}
```

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

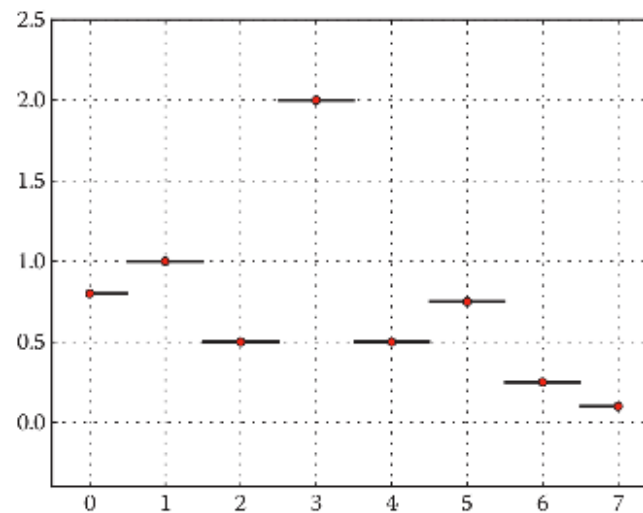


Figura 2.8: Interpolação do vizinho mais próximo. Os segmentos de reta mostram todo o intervalo que seria resolvido ao ponto denotado por cada círculo.

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

```
// transformar coordenadas de voxel de ponto flutuante em coordenadas
// inteiras
int continuousToDiscrete(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 = continuousToDiscrete(vsP);
    return buf.value(dvsP.x, dvsP.y, dvsP.z);
}
```

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:

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

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

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 $S = 2$. O número de valores de *voxel* que uma técnica de interpolação precisa é dado por 2^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];
}
```

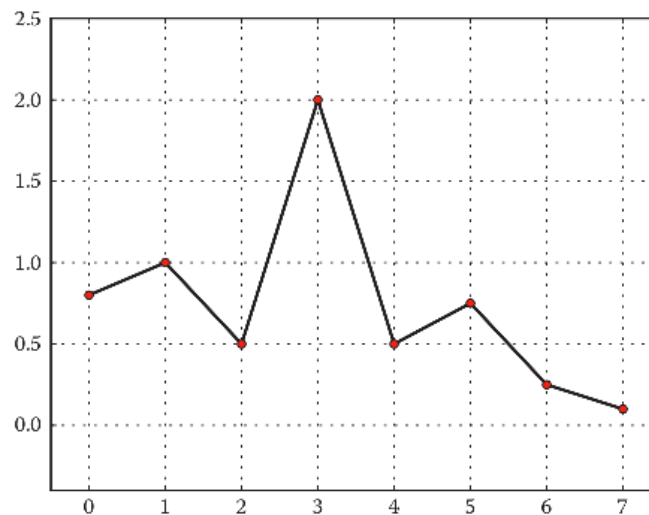


Figura 2.9: Interpolação linear. As quebras ocorrem na transição do centro de um voxel para o centro do voxel seguinte.

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:

$$\begin{aligned} f(x) &= ax^3 + bx^2 + cx + d \\ f'(x) &= 3ax^2 + 2bx + c. \end{aligned}$$

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:

$$\begin{aligned} f(0) &= P_i, \\ f(1) &= P_{i+1}, \\ f'(0) &= \frac{P_{i+1} - P_{i-1}}{2}, \\ f'(1) &= \frac{P_{i+2} - P_i}{2}. \end{aligned}$$

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

$$\begin{aligned} a &= -\frac{1}{2}P_{i-1} + \frac{3}{2}P_i + \frac{3}{2}P_{i+1} + \frac{1}{2}P_{i+2}, \\ b &= P_{i-1} + \frac{5}{2}P_i + 2P_{i+1} - \frac{1}{2}P_{i+2}, \\ c &= -\frac{1}{2}P_{i-1} + \frac{1}{2}P_{i+1}, \\ d &= P_i, \end{aligned}$$

o que nos leva à solução do problema original,

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

A interpolação cúbica usa dois valores abaixo e dois acima do ponto consultado, então a técnica tem um suporte $S = 4$, 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.

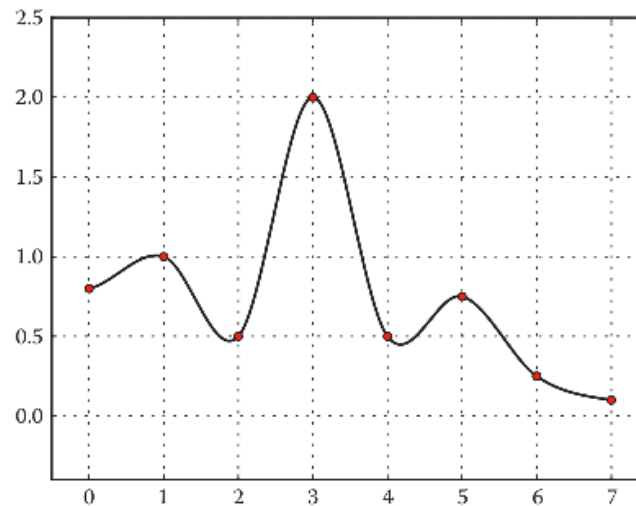


Figura 2.10: Interpolação cúbica. Como o polinômio é de terceira ordem, as curvas são bem mais suaves que da interpolação linear.

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 execução 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

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);
}

```

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.

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.

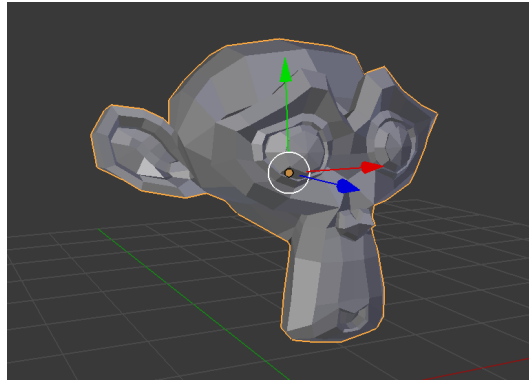


Figura 2.11: Espaço do objeto.

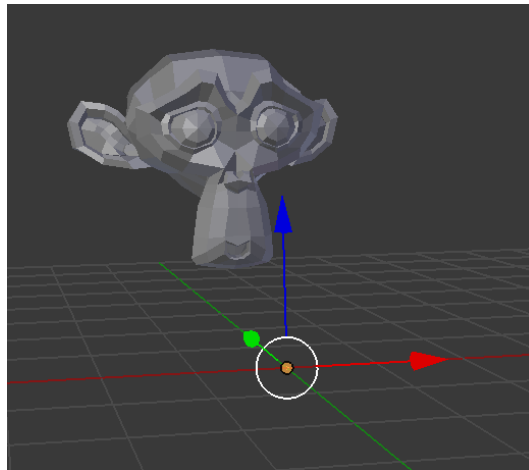


Figura 2.12: Espaço da cena, com a origem centrada no círculo menor.

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.

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

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 $T\mathbf{v}$ porque seu último elemento é 0. Se ao ponto $P = (P_x, P_y, P_z, 1)$ aplicarmos a transformada TP , 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 em uma 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.

⁵Algoritmo 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.

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 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.

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, BSDFs⁷ 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 global⁹.

Código 2.8: Estrutura de um shader escrito em OSL.

```
#include "stdosl.h"

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

⁶Função 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.

⁸Técnica 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

OpenVDB

A estrutura de dados utilizada é a VDB, renomeada para OpenVDB¹ 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 Animation², 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 interdependência 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 folhas³ que sempre

¹www.openvdb.org

²www.dreamworksanimation.com

³**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.

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 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 filhos, 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 em linha (`inline`) durante a compilação, e com isso, as funções não possuem uma sobrecarga de execução, que ocorreria se a opção fosse pelo uso de herança.

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 SIMD⁴.

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 na Seção 2.2.4, 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

⁴**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.

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 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* permite 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.

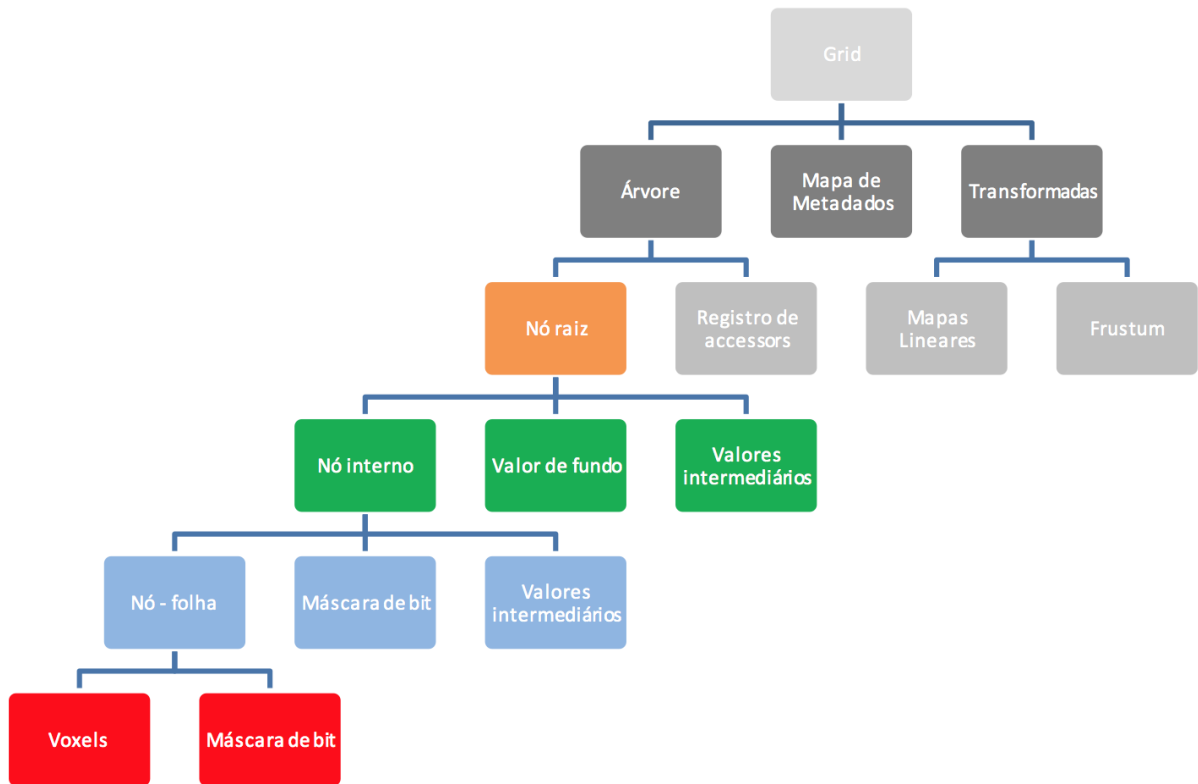


Figura 3.1: Diagrama da estrutura interna da árvore VDB.

Folhas. 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 $2^{\log_2(w)}$ *voxels* em cada eixo, onde $2^{\log_2(w)} = 1, 2, 3, \dots$ e w é qualquer entre x , y ou z . Uma configuração típica, e recomendada, é estabelecer

$2^{\log_2(w)} = 3$, 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 $1 \ll \sum_w s\log_2(w)$, onde \ll denota um deslocamento de *bits* à esquerda. As folhas armazenam os valores dos *voxels* em uma tabela chamada de *Direct Access Table*⁵, 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 que 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 variável de 64 *bits* `mFlags`, mostrado na Figura 3.2.

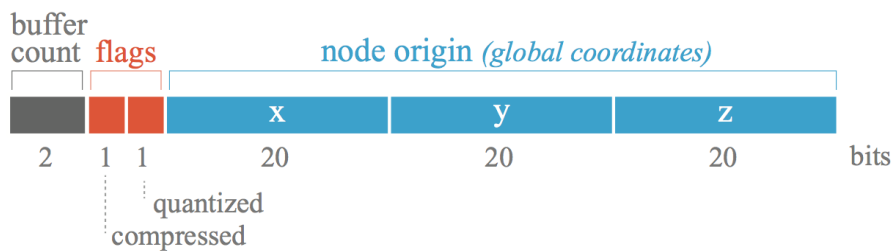


Figura 3.2: Representação compacta de `mFlags`, variável de 64 *bits* que codifica: número de *buffers* (2), compressão(1), quantização(1) e origem da folha (3×20).

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 em relação ao tempo, 2 *buffers*, valores salvos na memória com suporte a integração em relação ao tempo de primeira e segunda ordem ou 3 *buffers*, valores salvos

⁵Pode-se assumir que esta tabela é um vetor com custo de acesso aleatório de $O(1)$ para o pior caso.

na memória com suporte a integração em relação ao tempo de terceira ordem. O terceiro *bit* é 1 se o bloco é comprimido, e o quarto *bit* é 1 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`, $\text{Log}_2(w)$, 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 $\text{Log}_2(w)$ 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 esparsos 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 ponteiros 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*. Uma ilustração da estrutura é apresentada na Figura 3.3.

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 a 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.

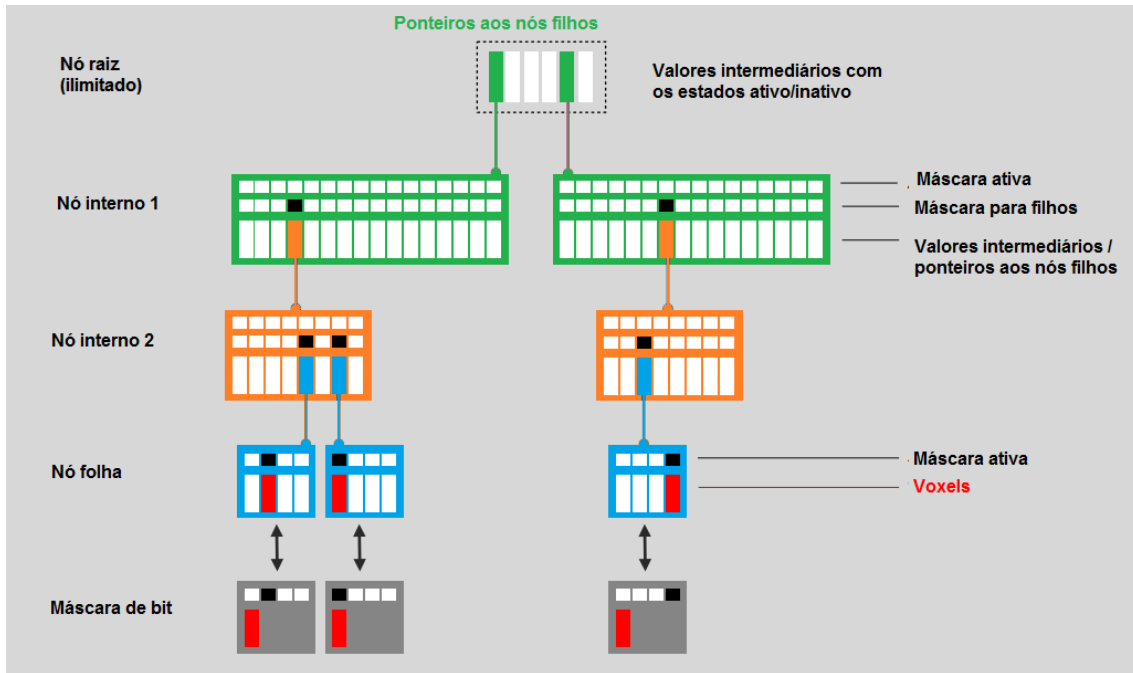


Figura 3.3: Ilustração unidimensional de uma VDB com um nó raiz, dois níveis de nós internos, e folhas. A raiz é dinâmica e esparsa, ao passo que os outros nós são densos e possuem fatores de ramificação que diminuem de cima para baixo, e que são restritos a potências de dois.

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 toda consulta aleatória leva o mesmo tempo no 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 ótima 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 vizinhança 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 3.4.

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 ótima é desejada.

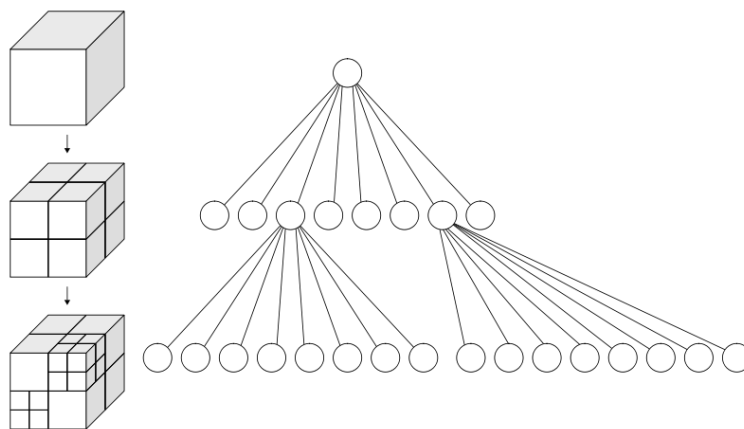


Figura 3.4: Diagrama de uma octree, estrutura em que cada nó interno possui exatamente oito nós-filho.

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.

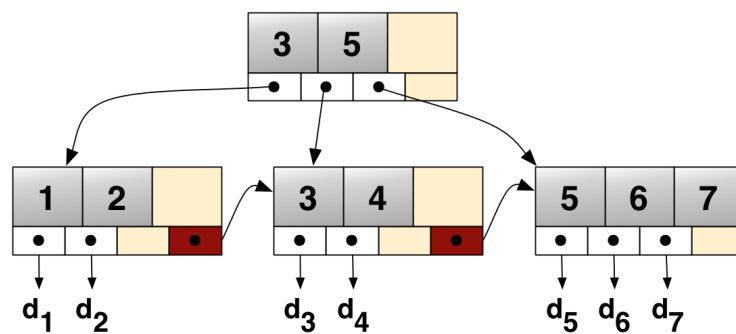


Figura 3.5: Diagrama de uma árvore B+. Os números de 1 a 7 são chaves, e os dados armazenados na árvore estão identificados de d_1 a d_7 .

Capítulo 4

Resultados

Para a apresentação dos resultados, é de interesse citar novamente os objetivos estabelecidos no Capítulo 1. Desejávamos integrar a biblioteca OpenVDB ao Blender, de modo a permitir a visualização de volumes e dados tridimensionais salvos em arquivos desta biblioteca. No entanto, o módulo do software responsável pelos cálculos de iluminação para a exibição do resultado final está em desenvolvimento por outros desenvolvedores e ainda não é funcional.

Por isso, um artifício será utilizado na exibição dos resultados. A consulta aos valores armazenados na textura, ou seja, a amostragem dos pontos será avaliada por fatias, que, neste caso, serão planos, posicionados próximos uns aos outros, de modo a dar a impressão de visualizarmos um volume. A visualização de valores interiores ao volume será dada pelo uso de apenas um plano. Esta estratégia de visualização é exibida na Figura 4.1.

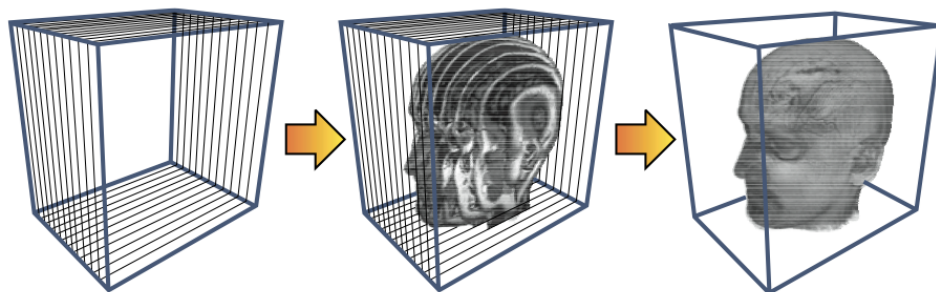


Figura 4.1: À esquerda, a geometria que conterá os pontos amostrados: uma pilha de planos. No centro, a avaliação dos dados volumétricos em cada uma das fatias. À direita, o resultado da renderização, dando a impressão de visualização de um volume.

O volume utilizado para exemplificar o funcionamento da biblioteca no Blender é exibido na Figura 4.2a, e um corte transversal, para visualizarmos seus dados internos, é exibido nas Figuras 4.2b e 4.2c.

No Blender, o arquivo de exemplo, `smoke.vdb`, é carregado através de um nó implementado em Open Shading Language, exibido na Figura 4.3. E na Figura 4.4, a interface do Blender com a pré-visualização do mesmo volume mostrado na Figura 4.2 é exibido. As

cores de exibição, que estão diferentes entre as representações, é facilmente configurável no Blender, se desejado.

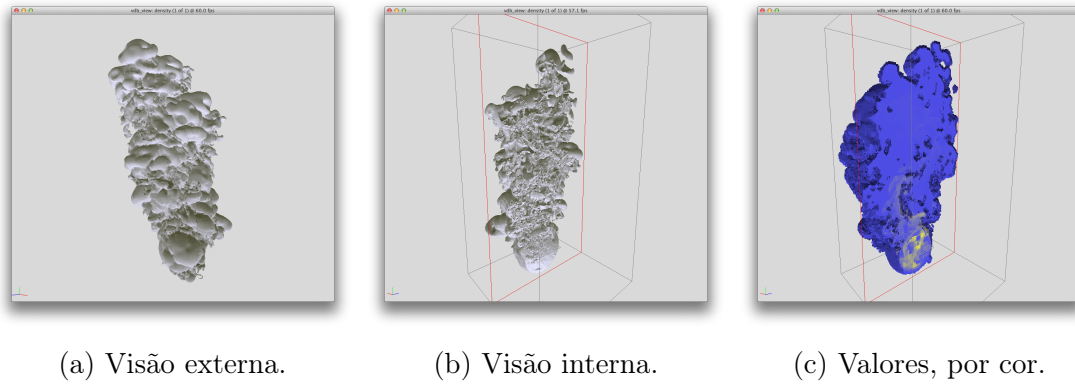


Figura 4.2: Volume representando a densidade de uma fumaça, usada no exemplo apresentado, exibido por um visualizador específico para o formato VDB.

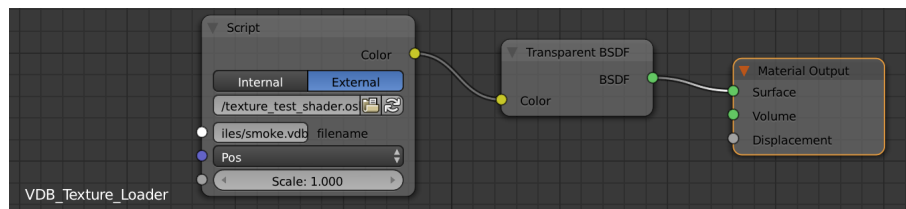


Figura 4.3: Nó gerado a partir de código OSL especificamente para carregar arquivos VDB.

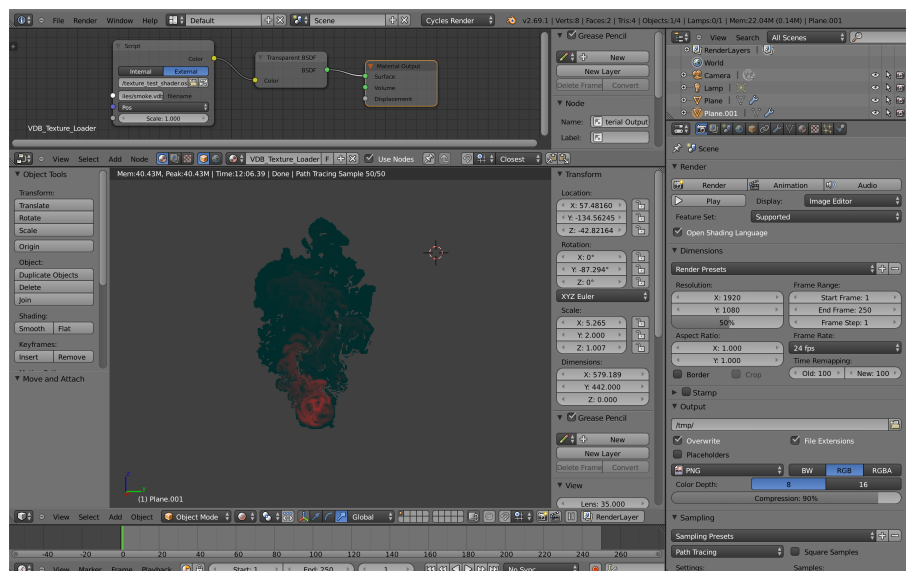


Figura 4.4: Interface do Blender com a pré-visualização de uma fatia do volume.

Capítulo 5

Conclusão

Neste capítulo, os desafios encontrados ao longo do planejamento e implementação deste trabalho serão mencionados, com uma breve explanação. Os resultados obtidos até este momento serão avaliados, e dada a relativa curta duração do projeto para um único desenvolvedor, as funcionalidades que são de interesse e que não foram implementadas ainda serão discutidas no contexto de trabalhos futuros. Finalmente, alguns comentários pessoais acerca do curso de graduação são oferecidos na última seção deste trabalho.

5.1 Desafios

Compilação de código-fonte

Trabalhar com software multi-plataforma de código aberto oferece alguns desafios inerentes à natureza da metodologia de desenvolvimento necessária. Primeiramente, o código-fonte precisa ser disponibilizado a todos que tiverem interesse em acessá-lo, e isso é essencialmente um problema fechado com os sistemas de versionamento mais modernos, como o `git`¹. Mas o processo deixa de ser trivial a partir deste ponto: compilar o software, qualquer que seja ele, a partir do código fonte, exige esforço considerável, por vários motivos: obter todas as dependências que o software usa é uma exigência, mas garantir que as versões necessárias são as mesmas que o desenvolvedor instalou é, por si só, um grande desafio, com fatores complicadores como a necessidade de manter múltiplas versões de uma biblioteca para satisfazer pacotes de softwares diferentes, sem que haja interferência entre elas. Neste mesmo cenário, existe ainda a distinção entre o uso de bibliotecas compiladas estática ou dinamicamente, afetando o processo de *linking* com mensagens de erro de difícil compreensão. Além disso, variáveis como versão de sistema operacional, de compilador e até mesmo de dispositivos de hardware, como versão de OpenGL suportada pela placa gráfica do desenvolvedor afetam o processo de compilação a partir do código-fonte.

¹<http://git-scm.com>

Neste trabalho, compilar a biblioteca OpenVDB para uso com o Blender apresentou vários destes desafios, e a solução encontrada foi compilar a biblioteca como parte da solução do Blender, de modo que compartilhassem todas as características de compilação, já que existem dependências em comum, e como resultado, diferenças nas variáveis de compilação deixaram de existir.

Identificação de Vazamentos de Memória

O uso de ponteiros em C e C++ não é exatamente trivial. E quando lida-se com arquivos em disco, o custo das operações de E/S normalmente são os fatores de maior impacto no desempenho da aplicação. Logo, ficou claro desde o início que o arquivo em que o volume estava armazenado deveria permanecer aberto, em memória, durante todo o processo de geração de imagens. Para isso, utilizou-se um ponteiro comum para uma estrutura criada que continha o arquivo. De posse do ponteiro, acessava-se o arquivo em memória.

A solução adotada funcionava inicialmente para pequenos casos de testes, mas causava o encerramento da aplicação após vários testes seguidos. Casos de teste maiores não eram concluídos. Os ponteiros saíam de escopo após cada sessão de geração de imagem, sem que fossem desalocados de forma apropriada. Após várias execuções, a quantidade de memória perdida era grande demais para manter a aplicação aberta.

A solução, sugerida por outros desenvolvedores, foi a adoção de *smart pointers*, ponteiros com alocação dinâmica de memória que garantem que a memória utilizada será desalocada quando o contador de referências ativas atingir zero. Como o funcionamento destes é ligeiramente diferente, a substituição não foi imediata, mas com os devidos ajustes no código, os vazamentos de memória cessaram.

Transformações, Espaços e Sistemas de Coordenadas

Os arquivos de volume *.vdb* definem seus próprios espaços, em que o volume armazenado está contido, e a relação entre seu sistema de coordenadas com aquele da cena utilizada no Blender não é clara. Além disso, as dimensões da imagem volumétrica pode ser ou muito maior ou muito menor que as dimensões da cena que a utilizará. Como consequência, ao importar um volume VDB em uma cena, a visualização nem sempre era possível, ou apenas um fragmento da textura ficava visível.

O desafio aqui, na realidade, estava no conceito do problema, pois a diferença entre espaços é natural. A solução foi adotar o objeto em que a textura está sendo avaliada como referência, e mapear, usando uma transformação linear, as coordenadas da textura para as coordenadas do mundo correspondentes. A transformação foi composta usando as coordenadas do objeto, e construindo a matriz 4×4 para a transformada afim.

5.2 Análise dos Resultados

Os resultados encontrados são satisfatórios no sentido de que foi implementado o que foi proposto. Contudo, sua aplicabilidade ainda é limitada, pois depende da implementação do módulo de *shading* para que os volumes em VDB possam ser renderizados de forma apropriada.

Ainda assim, a representação de volumes no Blender é agora mais eficiente, e suporta volumes ordens de grandeza maior que a estrutura anterior, em uma máquina com o mesmo tamanho de memória disponível. Um exemplo é apresentado na Tabela 5.1 para o volume usado neste trabalho, o volume representando uma densidade de fumaça no espaço.

Consumo de memória para representação do volume		
Estrutura de Dados	Consumo de Memória em <i>bytes</i>	Valor Relativo
Vetores 3D (Estrutura existente)	10.793.640	1.545
OpenVDB	6.984.656	1

Tabela 5.1: Comparação entre o uso de memória da implementação anterior com o uso da estrutura VDB.

5.3 Trabalhos Futuros

Muito embora o trabalho essencial tenha sido executado - agora existe suporte para abrir e visualizar volumes VDB no Blender, não houve tempo hábil para a conclusão de funcionalidades desejáveis, sendo que as principais são listadas a seguir.

- *Suporte à execução paralela.* O processo de geração de imagens é de alta complexidade computacional por natureza. E nas operações de avaliação de valores em uma estrutura de dados volumétrica as operações normalmente são altamente paralelizáveis, embora seja necessário tomar alguns cuidados, como preservar a integridade da estrutura na memória. Dessa forma, uma análise é necessária para averiguação de quais operações podem ser paralelizadas sem risco, verificando ainda se modificações na implementação atual podem favorecer a paralelização do código.
- *Criação de interface de usuário.* A interface usada atualmente entre o Blender e a estrutura de dados é um *shader* escrito em *Open Shading Language*. A criação de um nó específico para este fim - importar volumes VDB previamente salvos - facilitaria o processo. Este nó ficaria acessível a partir de um dos menus no software.
- *Permitir que volumes do Blender sejam exportados em VDB.* O Blender possui um simulador de fumaça bastante sofisticado, e há interesse em salvar estes volumes,

que atualmente são gerados e salvos como volumes densos, no formato VDB. Um conversor entre as duas representações precisa ser implementado para este fim.

5.4 Comentários Finais

Um dos aspectos mais interessantes de lidar com um problema de mundo real de uma aplicação deste gênero é a realização de que assuntos e disciplinas aparentemente sem relação alguma entre si convergem para permitir a solução de um problema. O uso de métodos de *Cálculo Numérico* para a implementação de técnicas estudadas em *Análise de Sinais e Sistemas*, por exemplo, teria sido uma aplicação de interesse na ocasião em que estes tópicos foram estudados pela primeira vez durante o curso de graduação, apesar de que possivelmente trata-se de uma aplicação exageradamente específica ao problema tratado aqui.

Um outro ponto importante é a aplicação direta de conceitos e práticas sugeridas em cursos como o de *Engenharia de Software*, como a configuração de software e as dificuldades em manter um conjunto de alterações em desenvolvimento isoladas da distribuição principal, mas mantendo a sincronização das alterações feitas à distribuição principal para que o software em desenvolvimento não fique obsoleto.

E finalmente, as disciplinas de *Álgebra Linear* e *Computação Gráfica* apresentaram os fundamentos para permitir a compreensão do problema, antes mesmo de tentar atacá-lo.

Em termos profissionais, a experiência foi bastante favorável pela comunicação com pessoas influentes da área de computação gráfica, e que sempre se mostraram bastante disponíveis para os interessados em aprender e desenvolver software novo. Além disso, depois de sofridas sessões de revisões de código, é muito gratificante ter código de sua autoria em um repositório público.

Bibliografia

- [1] M.S. Bazaraa, H.D. Sherali, C.M. Shetty. Nonlinear Programming: Theory and Algorithms, 3 ed., New York: John Wiley, (2006).
- [2] M.S. Bazaraa, J.J. Jarvis, H.D. Sherali. Linear Programming and Network Flows, 4 ed. John Wiley & Sons (2010).