Universidade Federal de Alagoas

Programa de Pós-Graduação em Matemática

DISSERTAÇÃO DE MESTRADO

Visualização da Curvatura de Objetos Implícitos em um Sistema Extensível

Allyson Ney Teodosio Cabral

Rio São Francisco

MATEMÁTICA
A ciência
do infinito

Maceió
Fevereiro de 2010

Allyson Ney Teodosio Cabral

Visualização da Curvatura de Objetos Implícitos em um Sistema Extensível

Dissertação de Mestrado na área de concentração de Computação Gráfica submetida em 11 de fevereiro de 2010 à Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em Matemática da Universidade Federal de Alagoas, como parte dos requisitos necessários à obtenção do grau de mestre em Matemática.

Orientador:

Vinícius Mello

Maceió, Fevereiro de 2010

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Maria Auxiliadora Gonçalves da Cunha

C117v Cabral, Allyson Ney Teodosio.
Visualização da curvatura de objetos implícitos em um sistema extensível. / Allyson Ney Teodosio Cabral, 2010.
51 f. : il., grafs.

Orientador: Vinícius Moreira Mello.
Dissertação (mestrado em Matemática) – Universidade Federal de Alagoas. Instituto de Matemática, 2010.

Bibliografia: f. 50-51.

1. Curvatura. 2. Curvatura – Visualização volumétrica . 3. B-Spline – Funções . 4. GPU – Linguagem de computação. 5. GLSL – Linguagem de computação. 6. Interpolação tricúbica. I. Título.

CDU: 514.772.2

Allyson Ney Teodosio Cabral

Visualização da Curvatura de Objetos Implícitos em um Sistema Extensível

Dissertação de Mestrado em Matemática, na área de Computação Gráfica, submetida em 11 de fevereiro de 2010 à Banca Examinadora, designada pelo colegiado do Programa de Pós-Graduação em Matemática da Universidade Federal de Alagoas, como parte dos requisitos à obtenção do grau de mestre em Matemática.

Vinícius Moreira Mello

Prof. Dr. Vinícius Moreira Mello
(Orientador)

Prof. Dr. Ralph Teixeira (UFF)

Prof. Dr. Adelailson Peixoto (UFAL)

Resumo

Neste trabalho, estudaremos a visualização da curvatura de superfícies definidas implicitamente por funções do tipo $f : [0, 1]^3 \rightarrow [0, 1]$, usando a técnica de lançamento de raios (*ray casting*). Como em geral conhecemos apenas valores amostrados de f, estudaremos um método de interpolação tricúbica, a fim de calcular as derivadas de segunda ordem precisamente.

A implementação computacional deste trabalho foi desenvolvida na forma de módulos do *framework* de visualização e processamento de imagens Voreen, o qual se beneficia do poder de processamento das placas gráficas atuais para acelerar o processo de visualização.

Palavras-Chaves: Curvatura, Visualização Volumétrica, Interpolação Tricúbica, B-Spline, GPU, GLSL, Objetos Implícitos.

Abstract

In this work we study the curvature visualization problem on surfaces implicitly defined by functions $f : [0, 1]^3 \rightarrow [0, 1]$, using the ray casting technique. As we usually know only sampled values of f, we study the tricubic interpolation method to compute second order derivatives accurately.

This work's implementation was designed as modules to the framework for volume rendering and image processing named Voreen, that uses the processing capability of graphics cards to improve the rendering tasks.

Keywords: Curvature, Volume Rendering, Tricubic Interpolation, B-Spline, GPU, GLSL, Implicit Objects.

Agradecimentos

Agradeço a Deus, a razão de tudo em minha vida. Aos meus pais Cleide e Paulo por ajudarem a tornar meus sonhos possíveis e por estarem sempre ao meu lado em todos os momentos. Ao meu orientador, Vínicius, pela paciência e dedicação durante o desenvolvimento desse trabalho. Ao professor Adán Corcho pelo carinho na condução da Coordenação do programa de Pós-Graduação. Aos professores Adelailson Peixoto e Ralph Teixeira pelas valiosas contribuições para esta dissertação. Aos amigos da PUC-Rio, professor Thomas Lewiner, Thales, Clarissa, Maria, Allan, Renata e Alex pela receptividade e valiosas sugestões. Aos amigos de Laboratório, Douglas, Marcelo, Michel, Fábio Bóia, Ailton, Ruff e Alisson pela importante companhia nos últimos dois anos. Aos amigos da turma de mestrado, Alex, Ana, Adalgisa, Isnaldo, Kenerson, Fábio Henrique, Isadora, Gregório, Rodrigo, Natália e Viviane. Um agradecimento especial aos amigos Douglas Cedrim e Fabiane Queiroz pelas correções e ajuda na elaboração de imagens. À FAPEAL pelo apoio financeiro.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Os métodos de visualização de volumes consistem em um conjunto de técnicas cujo objetivo é a geração de imagens a partir de dados trimensionais. As abordagens tradicionais [12] realizam a extração de malhas poligonais a fim de fazer uso delas no processo de visualização. Tais métodos são denominados métodos indiretos. Nesses casos, a malha construída pode ser utilizada para realizar processamentos posteriores. Entretanto, esses métodos possibilitam apenas a visualização de uma parte do volume por vez, mais especificamente a sua superfície de bordo. Essa limitação esconde informações que estão no interior do volume e que são de grande importância para, por exemplo, a geração de imagens médicas. Como alternativa para solução deste problema, foram desenvolvidos métodos que eliminam a etapa de geração de malhas e que tornaram possível visualizar em uma mesma imagem porções diferentes do volume. Essas técnicas, conhecidas como métodos diretos, em geral, utilizam algoritmos de lançamento de raios que percorrem o volume de dados em um processo de integração de valores que resulta nas cores finais dos *pixels* da imagem. Exemplos clássicos de algoritmos que utilizam o lançamento de raios são encontrados em [5], [10] e [3].

O conjunto de dados de entrada para algoritmos de renderização de volumes consiste em um conjunto de superfícies de nível descritas por uma aplicação $f : [0,1]^3 \rightarrow [0,1]$. Esses algoritmos realizam uma série de operações sobre os pontos da superfície, como o cálculo do gradiente, para as quais algumas condições de regularidades são necessárias. Na maioria dos casos é suficiente que f seja continuamente diferenciável, entretanto existem aplicações que necessitam do cálculo de derivadas de segunda ordem, a exemplo das aplicações que utilizam o cálculo de curvatura [9] [8]. Para esses casos podem ser utilizadas técnicas de interpolção tricúbica que garantem a exatidão das derivadas até a segunda ordem [22].

Uma desvantagem das técnicas de visualização direta é que para cada imagem do volume gerada todo o conjunto de dados precisa ser percorrido. Entretanto, como esses algoritmos são divididos em etapas independentes, vários trabalhos passaram a utilizar a

programação paralela em placas gráficas para acelerar a obtenção dos resultados. Dois exemplos desses trabalhos são [25] e [7]. Este último foi um dos muitos trabalhos recentes resultantes de um projeto que deu origem a uma ferramenta de visualização de volumes chamada Voreen que utiliza a GPU para realizar a visualização em tempo real em um sistema modular e interativo que permite a associação de várias técnicas com esse intuito.

1.1 Objetivo

O objetivo desse trabalho é fazer um estudo sobre os problemas envolvidos na visualização de volumes utilizando informações de curvatura e foi baseado no trabalho desenvolvido por Kindlmann et al. em [9]. Kindlmann utilizou os passos descritos por Möller et. al [16] para cálculo de curvaturas em superfícies para a aplicação na visualização de volumes.

A idéia inicial para o trabalho foi otimizar esse cálculo implementando o trabalho de Kindlmann para a execução em placas gráficas. Durante as pesquisas nos deparamos com o *framework* Voreen que, além de ser uma ferramenta muito poderosa para a geração de imagens de volumes, possui código fonte aberto estando disponível para colaborações.

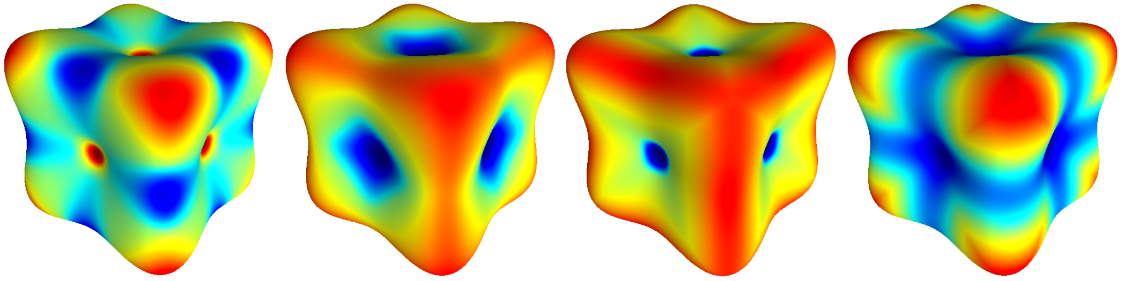
Como contribuição foram desenvolvidos módulos até então inexistentes no Voreen. O primeiro deles permite a visualização de volumes levando em consideração informações de curvatura que utiliza os passos descritos por Kindlmann. O segundo módulo permite a avaliação de funções amostradas através de interpolação tricúbica utilizando o método descrito em [22]. Em ambos foi utilizada a alta capacidade dos múltiplos processadores existentes nas placas gráficas para paralelizá-los e acelerar os cálculos envolvidos. Além desses, foram implementados módulos que realizam a visualização de volumes utilizando o algoritmo de *ray casting*[1]. Estes componentes foram incorporados ao Voreen e estão descritos no capítulo 5.

A imagem abaixo mostra alguns resultados obtidos através dos módulos implementados. Nelas aparece a visualização das curvaturas Gaussiana (K) e média (H) bem como das curvaturas principais (k_1 e k_2) sobre um volume utilizando o método de interpolação tricúbica.

1.2 Estrutura do Trabalho

O próximo capítulo deste trabalho (capítulo 2) apresentará métodos para a visualização direta de objetos implícitos que eliminam a necessidade da extração de malhas. No capítulo seguinte descreveremos a teoria necessária das técnicas de interpolação e do cálculo de curvaturas (capítulo 3). No capítulo 4 é realizada uma descrição de duas ferramentas

[1]Deixemos claro que existem diferenças entre os termos *ray casting* e *ray tracing* utilizados durante muito tempo na literatura como sinônimos. Atualmente, faz-se uma distinção clara entre os dois métodos já que o primeiro nunca utiliza o lançamento recursivo de raios secundários.



(a) K (b) H (c) k_1 (d) k_2

Figura 1.1: Resultados obtidos através do módulo de visualização de curvatura implementado. Nas imagens, azul indica curvaturas negativas, verde curvaturas próximas de zero e vermelho curvaturas positivas.

computacionais utilizadas para o desenvolvimento deste trabalho, a linguagem para programação em placas gráficas GLSL e o *framework* Voreen deixando para o capítulo 5 a tarefa de descrever os módulos desenvolvidos para o *framework*. Por fim apresentaremos os resultados, as conclusões e algumas propostas para trabalhos posteriores (capítulos 6 e 7).

Capítulo 2

Visualização de Objetos Implícitos

O processo de modelagem tradicional fornece como resultado uma malha de polígonos, independente do método utilizado para sua geração (superfícies de subdivisão, NURBS etc.). Esse fato é tão comum que os *hardwares* gráficos (GPUs) produzidos atualmente estão otimizados para a visualização de malhas de polígonos. Nestes casos, objetos sólidos são representados por sua superfície de bordo (B-Rep), não sendo possível, em um mesmo passo de renderização[1], visualizar seu interior.

Existem abordagens que utilizam a representação implícita de objetos, que consiste em caracterizá-los como o conjunto solução de uma função. Para visualizar esses objetos, alguns métodos utilizam a extração de malhas, tais como o *marching cubes* [12], que é um algoritmo utilizado para poligonização de superfícies definidas implicitamente. Entretanto, há métodos que não necessitam da extração de malhas, além de oferecerem a possibilidade de visualizar o interior do objeto, o que vem a ser importante em muitas aplicações, como na geração de imagens médicas, por exemplo. Aliado a isso, a constante evolução e facilidade de programação das placas gráficas permitem que elas sejam utilizadas também nesses métodos.

Neste capítulo, faremos uma breve descrição do método de visualização de objetos implícitos baseada em lançamento de raios (*ray casting*) [5]. Inicialmente, nos preocuparemos apenas em fazer uma descrição conceitual do problema, deixando os aspectos de implementação para o capítulo 4.

2.1 Objetos Implícitos

Um conceito fundamental para este trabalho é o de *objeto implícito*. De maneira simples, um objeto implícito pode ser entendido como o conjunto solução de uma determinada função. Mais precisamente:

[1]Em diversos momentos utilizaremos a palavra renderização, que é um estrangerismo da palavra proveniente da língua inglesa *rendering*, para indicar o processo de visualização.

Definição 2.1 *Um subconjunto $\mathcal{O} \subset \mathbb{R}^n$ é chamado* objeto implícito *se existe uma função $f : \mathbb{R}^n \to \mathbb{R}$ e um número real c tal que $\mathcal{O} = f^{-1}(c)$.*

A esfera é um exemplo de objeto implícito, pois se considerarmos a função $g : \mathbb{R}^3 \to \mathbb{R}$ definida por $g(x, y, z) = x^2 + y^2 + z^2$ e um $r \in \mathbb{R}$, o objeto implícito $\mathcal{S}(r) = f^{-1}(r)$ representa a esfera de raio r.

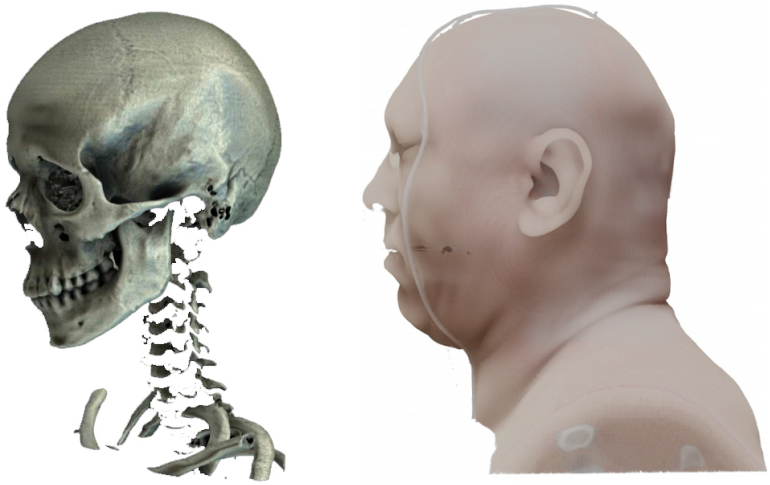
Uma ilustração prática desses conjuntos são os dados obtidos através do processo de tomografia computadorizada, onde são obtidos valores que indicam, para cada ponto do espaço, o índice de absorção de raios X do tecido do corpo humano presente neste ponto. Processados esses valores, pode-se obter imagens como a figura 2.1. Se considerarmos que c_1 e c_2 indicam, respectivamente, os índices de absorção dos tecidos ósseo e cutâneo, a imagem da esquerda é uma representação do conjunto $f^{-1}(c_1)$ e a da direita do conjunto $f^{-1}(c_2)$.

Como qualquer conjunto do $\mathbb{R}^n$ pode ser descrito como um objeto implícito [2], a definição acima torna-se muito geral. Na maioria dos casos essa generalidade é desnecessária e, além disso, vários problemas podem ser simplificados se condições de regularidade forem impostas. Nesta seção, enunciaremos alguns resultados com o objetivo de caracterizar os objetos implícitos do ponto de vista da geometria diferencial. Para mais detalhes, consultar [14].

Definição 2.2 *Um objeto implícito $\mathcal{O} = f^{-1}(c)$ é* regular *se f é diferenciável e $\nabla f(p) \neq 0$, para todo $p \in \mathcal{O}$, onde $\nabla f(p)$ indica o gradiente de f no ponto p.*

Essa definição pode ser estendida naturalmente impondo-se que f seja de classe C^k, com $k \geq 1$. Em muitas aplicações, é necessário que f seja ao menos de classe C^2, como

[2]Para definir um conjunto $A \subset \mathbb{R}^n$ como um objeto implícitito basta tormar a função característica $I_A : \mathbb{R}^n \to \{0, 1\}$ que tem valor 1 em todos os pontos de A e 0 nos pontos $x \in \mathbb{R}^n - A$ e fazer $A = f^{-1}(1)$.



(a) $f^{-1}(c_1)$ (b) $f^{-1}(c_2)$

Figura 2.1: Visualização dos tecidos ósseo e cutâneo (obtida de [7]).

veremos adiante. Um número real c é denominado *valor regular* de f se $\mathcal{O} = f^{-1}(c)$ é um objeto implícito regular.

Dois resultados teóricos são muito importantes no estudo dos objetos implícitos regulares: o *teorema da função implícita* e o *teorema de Sard*. O primeiro pode ser utilizado para mostrar que todo objeto implícito regular $\mathcal{O} = f^{-1}(c)$ é localmente o gráfico de uma função de n variáveis sobre a qual pode-se definir um espaço vetorial tangente $T_p\mathcal{O} = \nabla f(p)^{\perp}$. O segundo assegura que "quase todo"[3] $c \in \mathbb{R}$ é valor regular de f. Na prática, considerando um intervalo $[a, b]$ representado no computador e a função f, se existe algum valor $c \in [a, b]$ tal que $f^{-1}(c)$ não é regular, pequenas perturbações no valor de c garantem essa regularidade. Esse fato torna aceitável a suposição, feita pela maioria dos algoritmos de visualização, de que a superfície a ser visualizada é de fato regular. Descrições formais dos dois teoremas podem ser encontrados em [11].

Aproveitando a notação definida acima e os resultados mencionados, já que é possível definir um plano tangente em p, podemos calcular o vetor normal nesse ponto da superfície da seguinte forma

$$N(p) = \frac{\nabla f}{\|\nabla f\|}(p).$$

2.2 Visualização de Objetos Implícitos

Diversos métodos podem ser utilizados para a visualização de objetos implícitos, dentre eles estão algoritmos clássicos, como *marching cubes* [12] que, como foi dito anteriormente, é um método de poligonização, e o *ray casting* [5] que utiliza o lançamento de raios para gerar a visualização do objeto, eliminando a etapa de extração de malhas. Este último será descrito brevemente mais adiante, para o caso particular de objetos implícitos.

Um fato comum nestes métodos é que a visualização de um objeto implícito depende de uma *função de atributo* $a : \mathcal{O} \to \mathcal{A}$ que associa a cada ponto $p \in \mathcal{O}$ um elemento em um conjunto de atributos $\mathcal{A}$. O conteúdo desse elemento depende da aplicação em questão, mas tipicamente é consituído pela cor, pelo vetor normal e por propriedades físicas da superfície no ponto p.

Por exemplo, se uma certa aplicação necessita dos atributos *vetor normal* e *cor* em cada ponto $p \in \mathcal{O}$, então $a(p) = (N(p), C(p))$. Caso o atributo seja constante em $\mathcal{O}_c = f^{-1}(c)$ podemos usar a notação $a(c)$.

O processo de visualização consiste em dar um sentido visual a essas informações, unindo a informação geométrica em $\mathcal{O}$ e as propriedades do objeto descritas pela função a. Para tanto, é utilizado um algoritmo de iluminação que simula a interação desses dados com a luz presente na cena, gerando como resultado uma imagem.

[3] "Quase todo" neste caso significa que o conjunto dos valores que não são regulares (valores críticos) tem medida nula.

Consideradas as definições acima, podemos agora descrever o algoritmo de *ray casting* para objetos implícitos ilustrado na figura 2.2.

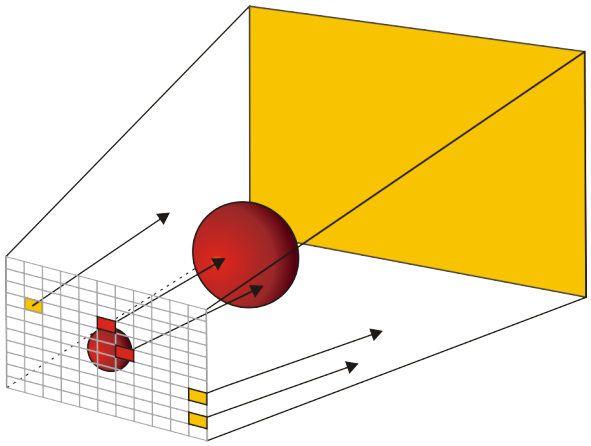


Figura 2.2: Descrição do algoritmo de traçado de raios. Os quadrados vermelhos representam os *pixels* cujos raios intersectam o volume.

Dado objeto $\mathcal{O} = f^{-1}(c)$, o primeiro passo consiste em considerarmos um *grid* regular, que representa a imagem resultante do algoritmo, onde cada elemento desse *grid* é chamado de *pixel*. Para cada *pixel* $P_{i,j}$ é traçado um raio $r(t)$ que parte da origem o e passa pelo ponto $P_{i,j}$, ou seja, $r(t) = P_{i,j} + t(P_{i,j} - o)$. Conhecendo o raio e a equação da superfície, podemos calcular a interseção entre o raio r e o objeto $\mathcal{O}$ fazendo $f(r(t)) = c$. Note que chegamos assim a uma equação na variável t. A primeira interseção de cada um dos raios com o objeto, que corresponde ao menor valor de t que satisfaz a equação acima, nos fornecerá pontos $p_{i,j}$ sobre os quais podemos aplicar a função a para obtermos sua cor original, $C(i, j)$, e o vetor normal, $N(i, j)$.

A etapa final do algoritmo consiste em aplicarmos um modelo de iluminação sobre os pontos utilizando as informações adquiridas nos passos anteriores. O resultado desse algoritmo é a cor final de cada pixel. Como ilustração, para este exemplo, utilizaremos modelo de iluminação de Phong-Blinn [20] que descreveremos a seguir.

A imagem acima ilustra as componentes do modelo de iluminação em questão. Esse modelo procura, de forma heurística, descrever o processo de reflexão da luz sobre a superfície. Sendo assim, o primeiro elemento a ser considerado é a reflexão difusa que é provocada pela absorção e irradiação uniformemente distribuída da luz sobre o objeto iluminado. Esse efeito é modelado pela Lei de Lambert, descrita pela equação

$$I = k_d(N \cdot L), \tag{2.1}$$

onde I é a intensidade refletida, k_d é o coeficiente de reflexão difusa determinado pelo material que constitui o objeto, N é o vetor normal à superfície no ponto em questão e L é o vetor que representa a direção da fonte de luz.

Aliada à componente do parágrafo anterior, para simplificar o modelo, a contribuição

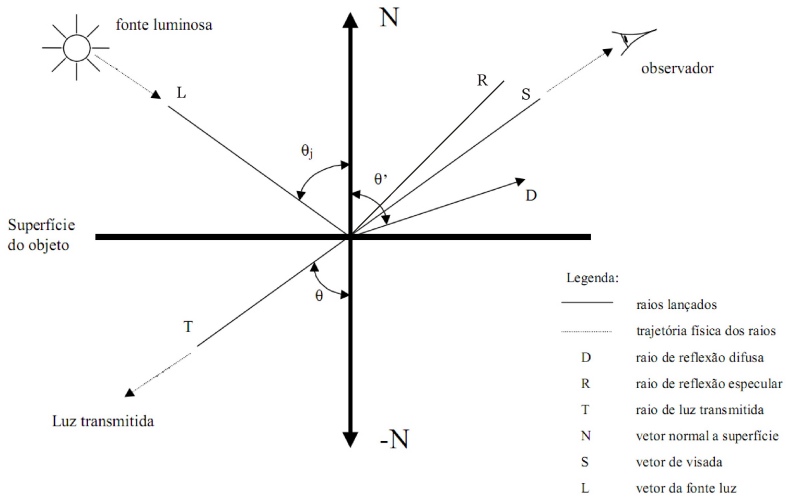


Figura 2.3: Componentes do modelo de iluminação

da energia luminosa provocada pela reflexão da luz por outros objetos contidos na cena é considerada em um novo elemento chamado de luz ambiente. Dessa maneira, o modelo pode ser novamente descrito como

$$I = I_a k_a + k_d(N \cdot L), \tag{2.2}$$

onde I_a representa a intensidade da luz ambiente e k_a é a constante de reflexão ambiente.

Finalmente, podemos considerar a componente especular que simula propriedades de reflexão da superfície. Essa componente é responsável pelo efeito de brilho intenso em uma determinada região da superfície conhecido como *highlight*. Chegamos então à seguinte expressão:

$$I = I_a k_a + k_d(N \cdot L) + k_s(N \cdot H)^\alpha, \tag{2.3}$$

onde k_s é o coeficiente de reflexão especular, α é o expoente de reflexão especular e H é o vetor que indica a direção de *highlight* máximo [1], este vetor está situado entre a fonte de luz e o observador a uma distância angular definida como a metade da existente entre os dois vetores.

Encontrada a intensidade de luz refletida $I_{i,j}$ em um determinado ponto $p_{i,j}$, é calculada então a cor $C_{i,j}$ do pixel $P_{i,j}$ da seguinte maneira:

$$C_{i,j} = I_{i,j} C(i,j).$$

A figura 2.4 mostra um resultado do *ray casting* de um objeto utilizando o modelo de iluminação descrito.

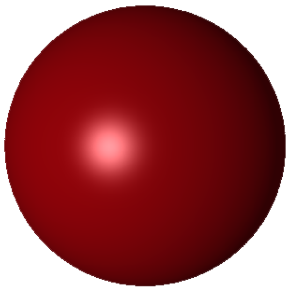


Figura 2.4: Resultado do algoritmo de *ray casting*

2.3 Visualização Volumétrica

Como já foi mencionado, em muitas aplicações é necessário visualizar o interior de um sólido, mas a representação por objetos implícitos nos permite visualizar apenas a superfície deles. Veremos nesta seção como estender o algoritmo descrito na seção anterior para visualização do interior de um sólido descrito por uma função escalar $f : [0,1]^3 \to [0,1]$. A idéia é semelhante a descrita na introdução do capítulo. A função f classifica os pontos do cubo $[0,1]^3$ atribuindo um valor de densidade a cada um deles. O processo de visualização que utiliza esses valores de densidade é chamado de *Visualização Volumétrica*.

Uma maneira de interpretar a visualização volumétrica é considerar que

$$[0,1]^3 = \cup_{c\in[0,1]} f^{-1}(c),$$

ou seja, o cubo $[0,1]^3$ é a união de todos as superfícies de nível da função f. Assim, podemos aplicar as técnicas de visualização de objetos implícitos para cada um dos níveis $c \in [0,1]$, integrando-os de alguma forma. É possível combinar os atributos contidos em cada um desses objetos de maneira a visualizar partes diferentes do volume simultaneamente através de um efeito de transparência ilustrado na figura 2.5.

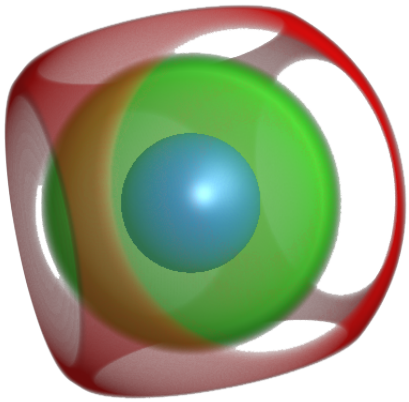


Figura 2.5: Efeito de transparência aplicado ao volume possibilitando a visualização de diferentes isosuperfícies.

O segredo para realizar essa integração é considerar que existe um atributo ω, chamado

opacidade, associado a cada ponto $p \in [0,1]^3$. Este atributo pode ser constante em cada superfície $\mathcal{O}_c = f^{-1}(c)$.

Durante o *ray casting*, um mesmo raio pode intersectar diferentes superfícies de nível, com diferentes valores de opacidade e cor. No algoritmo descrito anteriormente, considerávamos apenas a primeira interseção. O problema então é: como compor esses valores para gerar o resultado final?

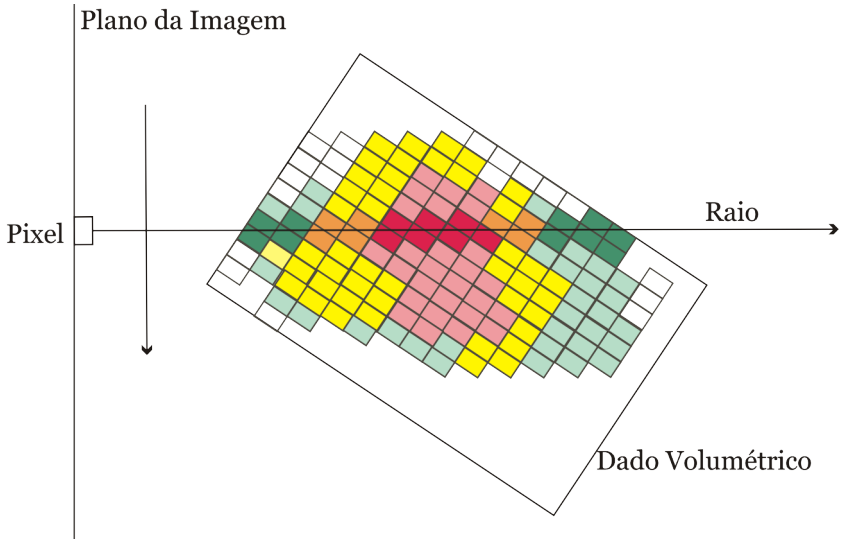


Figura 2.6: Descrição do algoritmo de visualização volumétrica. Para cada *pixel* um raio atravessa o volume acumulando os valores de cor e opacidade encontrados.

Tomemos como base a ilustração anterior. Como mencionado no início da seção, uma série de raios serão traçados através do volume. O primeiro cálculo a ser feito é o da quantidade de energia luminosa que atravessa o volume na direção de cada um desses raios e chega ao observador. Assim, dado um *pixel* $P_{i,j}$, sua intensidade de cor é dada pela seguinte equação:

$$I_{i,j} = \int_{t_B}^{t_A} e^{-\int_{t_A}^{t} \tau(s)ds} I_v(t)dt. \tag{2.4}$$

Nesta equação, I_v indica a intensidade luminosa em cada um dos pontos considerados e t_A e t_B são valores escalares que servem para fixar a posição dos pontos de entrada e saída $r(t_A)$ e $r(t_B)$ do raio r no volume.

Para entender a equação acima, considere que a energia luminosa I_v esteja atravessando um volume cilíndrico de área A_t e a altura Δs onde existem n partículas de área A_p que impedem a passagem de luz, conforme ilustra a figura 2.7.

Supondo-se que a espessura Δs seja suficientemente pequena e que as partículas não obscureçam umas às outras, a redução de energia luminosa pode ser calculada por:

$$\Delta I_v = -\frac{nA_p}{A_t} I_v = -\frac{(\rho(s)A_t\Delta s)A_p}{A_t} I_v = -\rho(s)A_p I_v \Delta s = -\tau(s) I_v \Delta s,$$

onde $\rho(s)$ indica a densidade de partículas opacas e $\tau(s) = \rho(s)A_p$ o coeficiente de oclusão

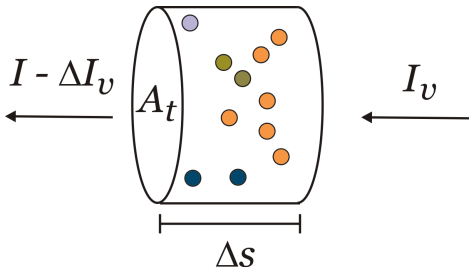


Figura 2.7: Modelo de absorção.

por unidade de comprimento ao longo do raio.

No limite em que Δs tende para zero, tem-se

$$\frac{dI_v}{ds} = -\tau(s)I_v,$$

que integrada resulta em

$$I_t = I_v(t)e^{-\int_{t_0}^{t}\tau(s)ds}. \tag{2.5}$$

Integrando todas as contribuições I_t ao longo do raio r de t_A a t_B chega-se a *equação de rendering* 2.4.

A integral da equação de *rendering* de volumes pode ser avaliada usando-se a soma de Riemann sobre uma partição $t_0 = t_A < t_1 < t_2 < ... < t_n = t_B$, de onde obtém-se

$$I_{i,j} = \sum_{k=0}^{n-1} e^{-\sum_{l=0}^{k-1}\tau_l\Delta t_l} I_k\Delta t_k = \sum_{k=0}^{n-1}\left(I_k\Delta t_k \cdot \prod_{l=0}^{k-1} e^{-\tau_l\Delta t_l}\right), \tag{2.6}$$

com

$$I_k \equiv I\left(\frac{t_k + t_{k+1}}{2}\right) \text{ e } \tau_l \equiv \tau\left(\frac{t_l + t_{l+1}}{2}\right).$$

Definindo $\omega_l \equiv 1 - e^{-\tau_l}$, de onde $\tau_l = -\ln(1-\omega_l)$, como a *opacidade* da amostra l; $C_k \equiv (I_k/\omega_k)\,\Delta t_k$, a *cor* da amostra k e $c_k \equiv C_k\omega_k$, a multiplicação dos dois, obtém-se:

$$I_{i,j} \cong \sum_{k=0}^{n-1}\left(c_k \cdot \prod_{l=0}^{k-1}(1-\omega_k)\right). \tag{2.7}$$

Antes de encerrar o capítulo, é importante mencionar que existe um tipo de função de atributo especial que associa para cada ponto p um valor de cor e opacidade e que possibilita a alteração desses valores para efeito de visualização, essas funções são conhecidas na literatura como *funções de transferência*.

Para mais informações sobre os aspéctos físicos envolvidos no processo de visualização de volumes e detalhes sobre as fórmulas que acabamos de descrever, consultar [19].

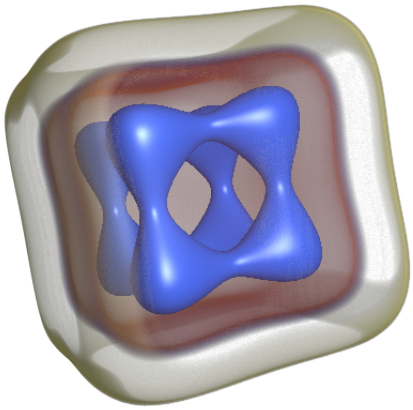


Figura 2.8: Algoritmo de visualização aplicado a um volume.

Capítulo 3

Interpolação e Cálculo de Curvatura

Como foi mencionado no capítulo introdutório, o objetivo deste trabalho é o estudo dos problemas envolvendo o cálculo de curvatura em superfíceis representadas implicitamente e a aplicação dos seus resultados no processo de visualização de volumes. Entretanto, para que isso seja possível, alguns aspectos devem ser garantidos.

Em muitas aplicações, desejamos obter uma função contínua $f{:}\,\mathbb{R} \to \mathbb{R}$ a partir de um sinal discreto $g[k]$, de tal modo que $f(k) = g[k]$ para todo $k \in \mathbb{Z}$. Esse problema é denominado *problema de interpolação de sinais discretos*. Nestes casos, podemos empregar interpolação linear, de modo que

$$f(x) = (1-t)g[i] + tg[i+1],$$

onde $i = \lfloor x \rfloor$ denota a parte inteira e $t = \{x\}$ a parte fracionária de x. Interpolação linear é uma solução muito eficiente para o problema de interpolação. Entretanto, em alguns problemas, como no cálculo de curvaturas, mais uma condição é necessária, a de que f seja de classe C^2. Neste caso, deve-se recorrer a outro método de interpolação, pois em geral as funções obtidas por interpolação linear não são diferenciáveis nos pontos $k \in \mathbb{Z}$.

Uma abordagem geral para o problema da interpolação é considerar que a função f pertence a um certo espaço de funções que possua as propriedades desejadas. Em aplicações de Computação Gráfica, os espaços mais usados são os *espaços das funções* splines *de grau* n,

$$S^n = \{f \in C^{n-1}(\mathbb{R}) | f|_{[k,k+1]} \text{ é um polinômio de grau } n\},$$

ou seja, S^n é o espaço das funções de classe C^{n-1} que são polinomiais por partes nos intervalos $[k, k+1]$ da reta. Note que para cada sinal discreto $g[k]$ existe uma única função em S^1 que satisfaz à condição de interpolação, exatamente a função obtida por interpolação linear.

Nos modelos tradicionais de visualização espera-se apenas que a superfície seja regular,

em outras palavras, que f definida em $\mathbb{R}^3$ seja de classe C^1, dessa forma podem ser definidos planos tangentes em cada um dos seus pontos. Esse fato é muito importante pois garante que vetores normais possam ser calculados sobre esses pontos para serem utilizados, por exemplo, no processo de iluminação da cena. Para este trabalho mais uma condição será imposta, a de que f seja de classe C^2, ou seja, que suas derivadas até a segunda ordem sejam contínuas. Essa garantia é fundamental para o cálculo da curvatura sobre pontos em superficies descritas por f.

Entretanto, para que f seja de classe C^2, devemos buscá-la no espaço S^3, conhecido como *espaço das splines cúbicas*. Neste capítulo, mostraremos como resolver o problema de interpolação para *splines* cúbicas, ou seja, dado um sinal $g[k]$, determinar a única função $f \in S^3$ tal que $f(k) = g[k]$, para todo $k \in \mathbb{Z}$. Além disso, será apresentado um método para obtenção de valores de curvatura sobre superfícies.

3.1 Interpolação Cúbica

Para uma boa compreensão do problema de interpolação cública, é necessária uma prévia caracterização dos espaços S^n. Analisando o método de interpolação linear, é fácil ver que se $f \in S^1$ então

$$f(x) = \sum_{k\in\mathbb{Z}} c[k]B_1(x-k), \tag{3.1}$$

onde $B_1(x)$ é a *função B-spline de grau* 1 exibida na figura 3.1(a) e $c[k] = g[k]$. O prefixo B antes da palavra *spline* significa que as translações inteiras da função B_1 formam uma base do espaço vetorial S^1. Note que como B_1 possui suporte compacto, para cada x o somatório em (3.1) é finito.

É possível mostrar, que para cada espaço S^n, com $n > 1$, existe uma função B_n, chamada *função B-spline de grau n*, tal que o conjunto $\{B_n(\cdot - k)\}_{k\in\mathbb{Z}}$ é uma base do espaço S^n, ou seja, cada função $f \in S^n$ pode ser escrita da forma

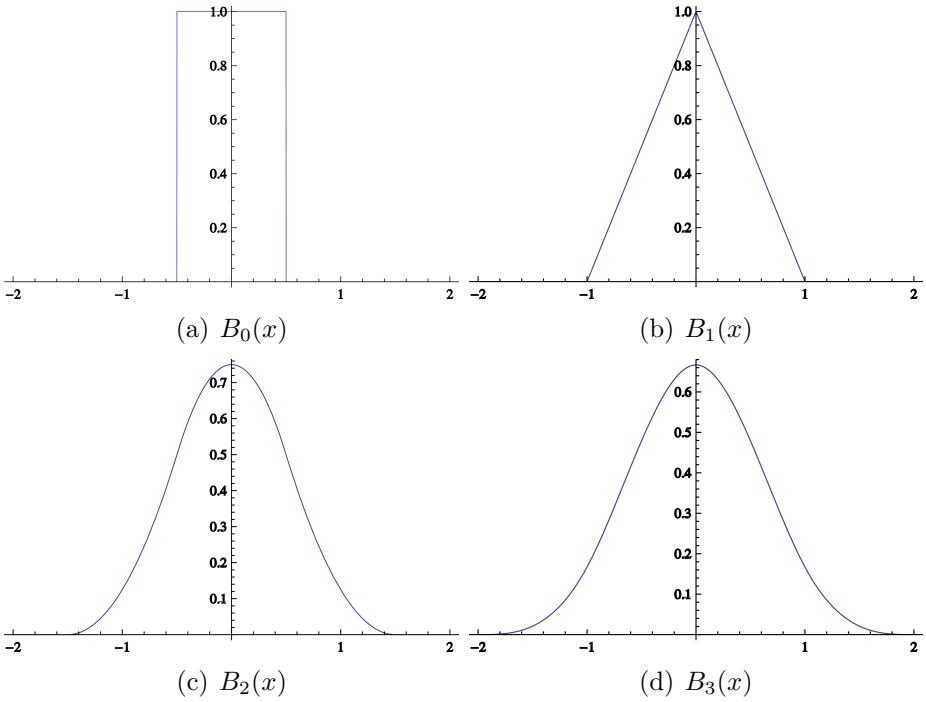
$$f(x) = \sum_{k\in\mathbb{Z}} c[k]B_n(x-k), \tag{3.2}$$

para coeficientes $c[k]$ unicamente determinados por f.

Existem várias fórmulas recursivas para as funções B_n. É possível mostrar, por exemplo, que $B_{n+1} = B_n \star B_0$, onde B_0 é a função característica do intervalo $(-\frac{1}{2}, \frac{1}{2}]$, conhecida como *função box*, e a operação $f \star g$ é o *produto de convolução* entre duas funções,

$$f \star g(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt.$$

O gráfico das funções B_n para $n = 0, \cdots, 3$ pode ser visto na figura 3.1.



(a) $B_0(x)$ (b) $B_1(x)$

(c) $B_2(x)$ (d) $B_3(x)$

Figura 3.1: Funções B-spline.

Aplicando a definição recursiva, concluímos que

$$B_3(x) = \begin{cases} 0 & , \text{ se } |x| \geq 2 \\ \frac{1}{6} \cdot (2 - |x|)^3 & , \text{ se } 1 \leq |x| < 2 \\ \frac{2}{3} - \frac{1}{2}|x|^2 \cdot (2 - |x|) & , \text{ se } |x| < 1 \end{cases} \tag{3.3}$$

Como o suporte da função B_3 tem largura 4, em geral cada ponto x da função $f(x)$ sofre a influência de quatro funções de base $B_3(\cdot - k)$, mais exatamente para $k = i-1, i, i+1, i+2$ onde $i = \lfloor x \rfloor$. Mas se x é inteiro, apenas três funções de base contribuem para o valor de $f(x)$, aquelas com $k = i - 1, i, i + 1$, onde $i = x$. Portanto, temos que para todo $k \in \mathbb{Z}$

$$\begin{aligned} f(k) &= c[k-1]B_3(k-(k-1)) + c[k]B_3(k-k) + c[k+1](k-(k+1)) \\ &= c[k-1]B_3(1) + c[k]B_3(0) + c[k+1]B_3(-1) \\ &= \frac{1}{6}c[k-1] + \frac{4}{6}c[k] + \frac{1}{6}c[k+1]. \end{aligned}$$

Voltando ao problema de interpolação, o problema de encontrar a função $f \in S^3$ que interpola o sinal discreto $g[k]$ resume-se a determinar coeficientes $c[k]$ tais que

$$g[k] = \frac{1}{6}c[k-1] + \frac{4}{6}c[k] + \frac{1}{6}c[k+1] \tag{3.4}$$

para todo $k \in \mathbb{Z}$. Note que os coeficientes $c[k]$ podem ser interpretados como um sinal discreto de modo que a equação (3.4) pode ser colocada na forma de uma *convolução*

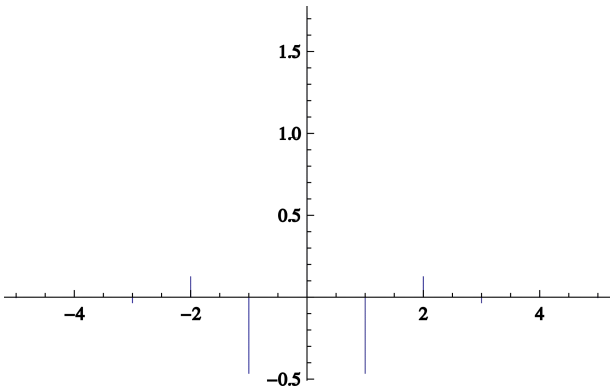


Figura 3.2: Gráfico do sinal $h[k]$.

discreta

$$g[k] = ([\frac{1}{6}\ \frac{4}{6}\ \frac{1}{6}] \star c)[k],$$

onde

$$(g \star h)[k] = \sum_{i \in \mathbb{Z}} g[i]h[k-i].$$

Considerando que o produto de convolução é associativo, temos que, ao menos formalmente,

$$c[k] = ([\frac{1}{6}\ \frac{4}{6}\ \frac{1}{6}]^{-1} \star g)[k],$$

e o problema se reduz a encontrar a inversa com respeito à convolução do sinal discreto $[\frac{1}{6}\ \frac{4}{6}\ \frac{1}{6}]$. A técnica da *Transformada Z* pode ser utilizada para esse fim, mas nesse caso em particular é fácil verificar diretamente que

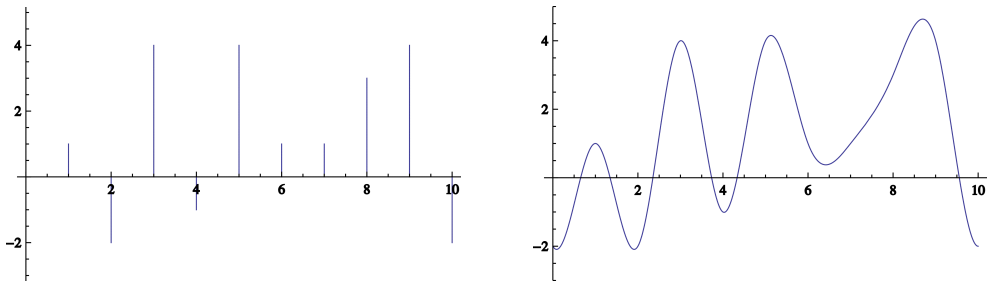
$$h[k] = \frac{-6z}{1-z^2} z^{|k|},$$

com $z = -2 + \sqrt{3}$, é a inversa de $[\frac{1}{6}\ \frac{4}{6}\ \frac{1}{6}]$, basta usar a definição de convolução discreta e o fato que $z^2 + 1 = -4z$.

3.2 Implementação da Inversão

Vemos pelo gráfico de $h[k]$ (figura 3.2) que seu valor absoluto cai exponencialmente a medida que k vai para o infinito. Assim podemos truncar o sinal $h[k]$ para um valor de k_0 suficientemente grande, ou seja, considerar que $h[k] = 0$ se $|k| > k_0$, e obter os valores dos coeficientes $c[k]$ convoluindo $g[k]$ com o sinal truncado. Mas é possível implementar o cálculo dos coeficientes $c[k]$ mais eficientemente, do seguinte modo: temos que

$$h[k] = \frac{-6z}{1-z^2}(y^+[k] + y^+[-k] - 1),$$



(a) $g[k]$ (b) $f(x)$

Figura 3.3: Interpolação Cúbica.

onde

$$y^+[k] = \begin{cases} 0 & , \text{ se } k < 0 \\ z^k & , \text{ se } k \geq 0 \end{cases}.$$

Como $y^+[k+1] = zy^+[k]$, se $k > 0$, temos $o = y^+ \star g$ pode ser calculada recursivamente com regra de atualização

$$o[k] := g[k] + zo[k-1].$$

Por outro lado, como $c = h \star g$, temos que c e o satisfazem a relação

$$c[k] = z(c[k+1] - o[k]).$$

Assim, o valor dos coeficientes $c[k]$ pode ser calculado em duas etapas: com k variando de 1 a N, fazemos

$$o[k] := g[k] + zo[k-1],$$

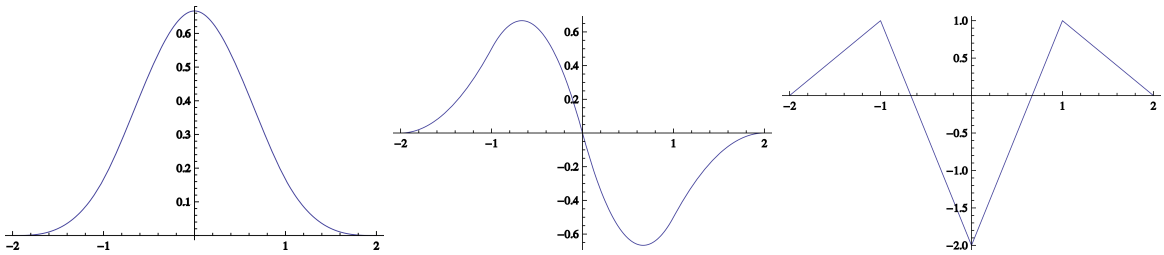
e com k variando de $N-1$ a 0, fazemos

$$c[k] := z(c[k+1] - o[k]).$$

Resta apenas determinar os valores iniciais dessas duas recursões, $o[0]$ e $c[N]$, respectivamente. Mas isso depende de como o sinal é estendido para $k < 0$ e $k > N$. Dependendo da extensão do sinal, teremos condições iniciais distintas. Na *extensão por reflexão*, na qual, $g[k] = g[-k]$, se $-N \leq k < 0$ e $g[k] = g[2N-k]$, se $N < k \leq 2N$, temos das definições de o e c que

$$o[0] = \sum_{k=0}^{k_0} z^k g[k] \text{ e } c[N] = \frac{-6z}{1-z^2}(2o[N] - g[N]).$$

Esse algoritmo é bastante eficiente, pois são efetuadas apenas duas multiplicações e adições para cada entrada k. Na próxima seção, analisaremos como calcular as derivadas de f até a segunda ordem.



(a) $B_3(x)$ (b) $B_3'(x)$ (c) $B_3''(x)$

Figura 3.4: Derivadas da função *spline* cúbica.

3.3 Derivadas das Funções *Spline*

Agora que sabemos como avaliar a função f utilizando *B-Splines* cúbicas, podemos analisar o problema de calcular as derivadas desta função utilizando os coeficientes obtidos.

Encontrar as derivadas da função f se torna uma tarefa simples quando a caracterizamos como

$$f(x) = \sum_{k\in\mathbb{Z}} c[k]B_3(x-k),$$

pois, pela linearidade da derivação,

$$f'(x) = \sum_{k\in\mathbb{Z}} c[k]B_3'(x-k) \text{ e } f''(x) = \sum_{k\in\mathbb{Z}} c[k]B_3''(x-k).$$

onde B_3' e B_3'' indicam as derivadas de primeira e segunda ordem da função B_3 descritas pelas equações 3.5 e 3.6.

$$B_3'(x) = \begin{cases} 0 & , \text{ se } |x| \geq 2 \\ \frac{1}{2}x^2 + 2x + 2 & , \text{ se } -2 < x \leq -1 \\ -\frac{3}{2}x^2 - 2x & , \text{ se } -1 < x \leq 0 \\ \frac{3}{2}x^2 - 2x & , \text{ se } 0 < x \leq 1 \\ -\frac{1}{2}x^2 + 2x - 2 & , \text{ se } 1 < x < 2 \end{cases} \tag{3.5}$$

$$B_3''(x) = \begin{cases} 0 & , \text{ se } |x| \geq 2 \\ x + 2 & , \text{ se } -2 < x \leq -1 \\ -3x - 2 & , \text{ se } -1 < x \leq 0 \\ 3x - 2 & , \text{ se } 0 < x \leq 1 \\ -x + 2 & , \text{ se } 1 < x < 2 \end{cases} \tag{3.6}$$

Estas derivadas estão representadas na figura 3.4.

3.4 Avaliação Rápida

Apresentaremos agora uma versão otimizada para o cálculo da convolução vista na seção 3.1. Esse resultado foi obtido por Sigg e Hadwiger em [24] com o intuito de aproveitar melhor o potencial oferecido pelas placas gráficas. Eles levaram em consideração que o acesso linear à textura é muito mais rápido em relação a dois acessos de vizinho mais próximo, mesmo as duas operações acessando o mesmo número de *texels*[1]. O objetivo é se beneficiar disso durante o cálculo da convolução.

Como o suporte da função B_3 tem largura 4 podemos escrever a equação 3.2 da seguinte forma:

$$f(x) = B_{i-1} \cdot g_{i-1} + B_i \cdot g_i + B_{i+1} \cdot g_{i+1} + B_{i+2} \cdot g_{i+2}. \tag{3.7}$$

onde $g[i] = g_i$ e $B_3(x - i) = B_i$.

A idéia básica do algoritmo idealizado por Sigg e Hadwiger é reescrever a equação 3.7 como a soma ponderada de interpolações lineares entre cada par de valores da função. Como foi visto, a interpolação linear é calculada através de uma combinação convexa da seguinte forma

$$f(x) = (1 - t)g[i] + tg[i + 1],$$

onde $i = \lfloor x \rfloor$ denota a parte inteira e $t = \{x\}$ a parte fracionária de x e $g[k] = f(k)\ \forall k \in \mathbb{Z}$. Podemos reescrever uma combinação linear qualquer $a \cdot g[i] + b \cdot g[i+1]$ construída através de uma combinação convexa com a e b quaisquer como

$$(a + b) \cdot f(i + b/(a + b)) \tag{3.8}$$

sempre que a condição de combinação convexa $0 \leq (b/(a + b)) \leq 1$ for satisfeita. Assim, ao invés de realizar duas buscas na memória de textura para recuperar os valores $g[i]$ e $g[i + 1]$ e uma interpolação linear, faz-se apenas uma busca em $i + b/(a + b)$ e uma multiplicação por $(a + b)$.

Podemos então reescrever a equação 3.7 como

$$f(x) = q_0(x) \cdot f(x - h_0(x)) + q_1(x) \cdot f(x + h_1(x)), \tag{3.9}$$

onde as novas funções g_0, g_1, h_0 e h_1 são definidas da seguinte forma

$$\begin{array}{ll} q_0(x) = B_{i-1} + B_i & h_0(x) = 1 - \frac{B_i}{B_{i-1}+B_i} + x \\ q_1(x) = B_{i+1} + B_{i+2} & h_1(x) = 1 + \frac{B_{i+2}}{B_{i+1}+B_{i+2}}. \end{array}$$

Através desse método, a convolução pode ser realizada utilizando apenas dois acessos lineares à textura e uma interpolação linear, o que é muito mais rápido que quatro acessos

[1] *Texel é a menor unidade de textura.*

de vizinho mais próximo.

3.5 Caso Tridimensional

Os passos descritos nas seções anteriores para o cálculo de interpolação utilizando funções *B-spline* cúbicas podem ser facilmente estendidos para o caso trimensional. Neste contexto, f passa a ser avaliada a partir de um produto tensorial.

No caso geral temos então o seguinte

$$f(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3(y-j)B_3(z-k). \tag{3.10}$$

Podemos ainda calcular as derivadas parciais de primeira e segunda ordem de f da seguinte forma

$$f_x(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3'(x-i)B_3(y-j)B_3(z-k),$$

$$f_y(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3'(y-j)B_3(z-k),$$

$$f_z(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3(y-j)B_3'(z-k),$$

$$f_{xx}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3''(x-i)B_3(y-j)B_3(z-k),$$

$$f_{xy}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3'(x-i)B_3'(y-j)B_3(z-k),$$

$$f_{xz}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3'(x-i)B_3(y-j)B_3'(z-k),$$

$$f_{yy}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3''(y-j)B_3(z-k),$$

$$f_{yz}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3'(y-j)B_3'(z-k),$$

$$f_{zz}(x,y,z) = \sum_{i,j,k\in\mathbb{Z}} c[i,j,k]B_3(x-i)B_3(y-j)B_3''(z-k).$$

Pela natureza das equações apresentadas, percebe-se facilmente que $f_{xy} = f_{yx}$, $f_{xz} = f_{zx}$ e $f_{yz} = f_{zy}$.

Como pode ser notado, o cálculo desses somatórios possui um alto custo computacional, entretanto, as considerações que acabamos de fazer sobre o método de avaliação rápida para o caso unidimensional podem ser facilmente estendidas para o cenário tridi-

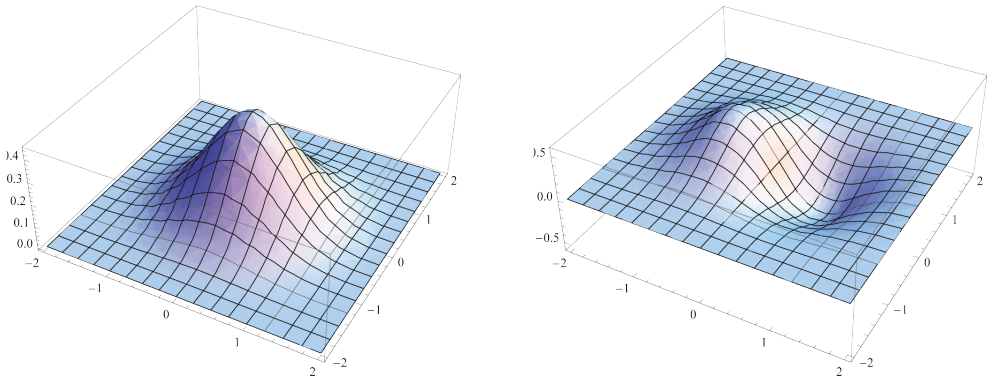
mensional. Nesse caso, é possível realizar o somatório de 64 termos descrito na equação 3.10 utilizando apenas 8 acessos à textura.

No caso 3D os pesos e as distâncias entre as amostras podem ser calculados separadamente para cada uma das dimensões. Os seus valores finais são dados por

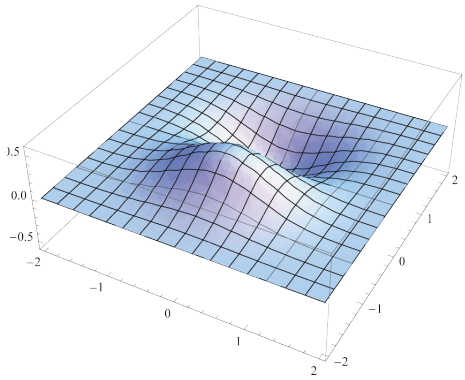
$$g_{\vec{j}} = \prod g_{j_k} \text{ e } \vec{h}_{\vec{j}} = \sum \vec{e_k} \cdot h_{jk}$$

onde k denota os eixos e $\vec{e_k}$ os vetores da base.

As figuras 3.5 e 3.6 apresentam respectivamente as derivadas de primeira e segunda ordem da função f para o caso 2D.



(a) $B_3(x, y)$ (b) $\partial_x B_3(x, y)$

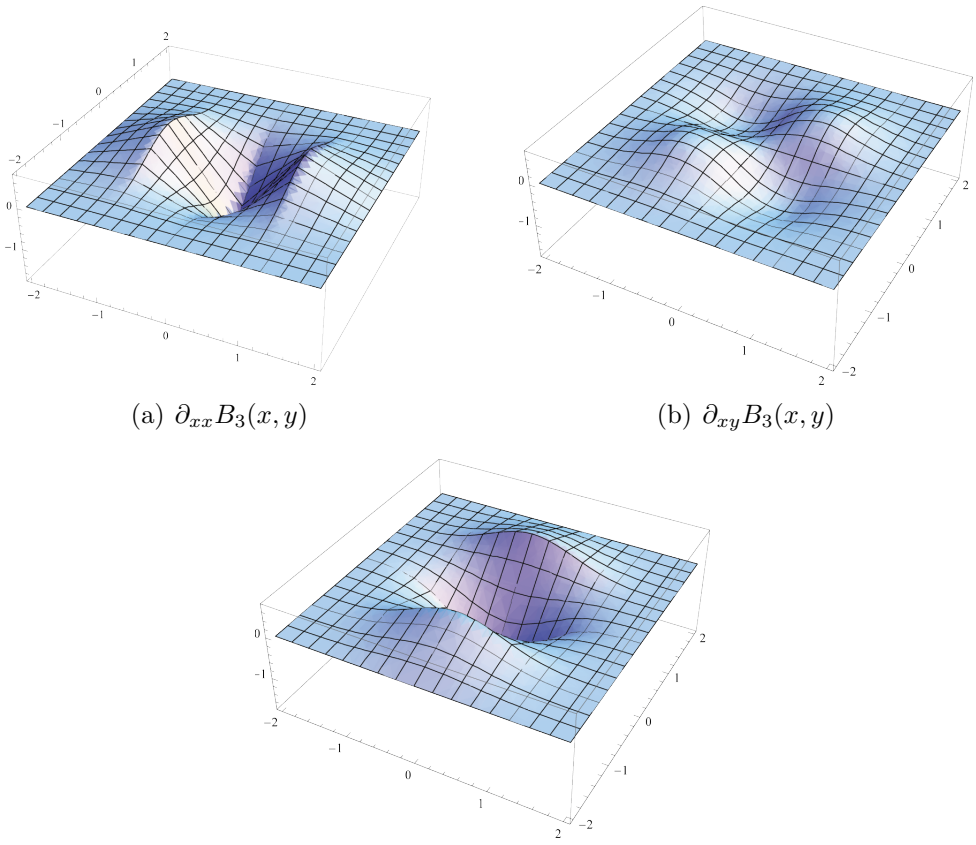


(c) $\partial_y B_3(x, y)$

Figura 3.5: Funções *B-spline* bidimensionais e suas derivadas parciais de primeira ordem.

3.6 Curvatura

Nesta seção utilizaremos os conhecimentos adquiridos durante este capítulo para entender como calcular valores de curvatura sobre uma superfície. Para tanto, seguiremos os passos descritos por Kindlmann et al. em [9] que sustenta seu método no fato de a curvatura em uma superfície ser definida como a taxa de variação da normal na vizinhança de um ponto da superfície com respeito a uma variação infinitesimal da posição do ponto.



(a) $\partial_{xx}B_3(x,y)$ (b) $\partial_{xy}B_3(x,y)$

(c) $\partial_{yy}B_3(x,y)$

Figura 3.6: Funções *B-spline* bidimensionais e suas derivadas parciais de segunda ordem.

Seja f uma função do cubo $[0,1]^3$ no intervalo $[0,1]$. Convencionemos que os valores de f aumentam à medida que nos afastamos do centro do cubo (padrão seguido pelos equipamentos de tomografia computadorizada) e lembremos do capítulo 2 que a normal sobre a superfícies é definida como $N = g/|g|$ onde $g = \nabla f = [\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial f}{\partial z}]$. As informações sobre a curvatura estão em ∇N^T que é em uma matriz 3×3 e pode ser expandida da seguinte forma:

$$\begin{aligned}
\nabla N^T &= -\nabla\left(\frac{g^T}{|g|}\right) = -\left(\frac{\nabla g^T}{|g|} - \frac{g\nabla^T|g|}{|g|^2}\right) \\
&= -\frac{1}{|g|}\left(H - \frac{g\nabla^T(g^Tg)^{1/2}}{|g|}\right) = -\frac{1}{|g|}\left(H - \frac{g\nabla^T(g^Tg)}{2|g|(g^Tg)^{1/2}}\right) \\
&= -\frac{1}{|g|}\left(H - \frac{g(2g^TH)}{2|g|^2}\right) = -\frac{1}{|g|}\left(1 - \frac{gg^T}{|g|^2}H\right) \\
&= -\frac{1}{|g|}\left(I - NN^T\right)H.
\end{aligned}$$

onde I é a matriz identidade e H a matriz Hessiana definida por

$$H = \begin{bmatrix} \partial^2 f/\partial x^2 & \partial^2 f/\partial x \partial y & \partial^2 f/\partial x \partial z \\ \partial^2 f/\partial y \partial x & \partial^2 f/\partial y^2 & \partial^2 f/\partial y \partial z \\ \partial^2 f/\partial z \partial x & \partial^2 f/\partial z \partial y & \partial^2 f/\partial z^2 \end{bmatrix}.$$

O produto externo de N com si próprio, NN^T, é um operador linear que projeta vetores sobre o subespaço vetorial unidimensional gerado por N. O operador $I - NN^T$ projeta sobre o complemento ortogonal desse subespaço, que vem a ser o plano tangente à isosuperfície em p. Fazendo $P = I - NN^T$ temos

$$\nabla N^T = -\frac{1}{|g|} PH. \tag{3.11}$$

A equação 3.11 fornece uma boa descrição de ∇N^T. A matriz Hessiana representa a maneira como o gradiente g varia de acordo com uma função de mudanças infinitesimais de posição em $\mathbb{R}^3$. Essa variação em g possui uma componente ao longo de g (o gradiente pode variar o tamanho) e uma componente no plano tangente (o gradiente pode mudar de direção). Apenas o último componente é relevante para descrever a curvatura. Esse componente pode ser isolado apenas com uma multiplicação à esquerda por P. Finalmente, o fator escalar $1/|g|$ converte mudanças infinitesimais do gradiente g (não normalizado) em mudanças infinitesimais do vetor normal unitário N.

As matrizes P e H são ambas simétricas, entretanto ∇N^T não é simétrica em geral. Apesar disso, se v está sobre o plano tangente tem-se $Pv = v$ e $v^T P = v^T$, portanto para u e v no plano tangente,

$$v^T PHu = v^T Hu = u^T Hv = u^T PHv.$$

Isso siginifica que a restrição de $\nabla N^T = -PH/|g|$ ao plano tangente é simétrica e, portanto, existe uma base ortonormal $\{p_1, p_2\}$ para o plano tangente na qual ∇N^T é uma matriz diagonal 2×2. Essa base pode ser facilmente estendida para uma base ortonormal para todo $\mathbb{R}^3$, $\{p_1, p_2, N\}$. Nessa base, a derivada da normal a superfície é dada por

$$\nabla N^T = \begin{bmatrix} k_1 & 0 & \sigma_1 \\ 0 & k_2 & \sigma_2 \\ 0 & 0 & 0 \end{bmatrix}.$$

A última linha é nula porque nenhuma variação na posição fará a normal mudar de tamanho. Mudanças na posição do plano tangente ao longo de p_1 e p_2 provocam mudanças em N na mesma direção com proporções k_1 e k_2 respectivamente. Pela definição de curvatura em superfícies [2], p_1 e p_2 são as direções principais enquanto k_1 e k_2 são as

curvaturas principais. A medida que nos movemos na direção da normal, se aproximando ou afastando da superfície, a normal inclina de acordo com os valores de σ_1 e σ_2.

Multiplicar ∇N^T por P isola k_1 e k_2 na base $\{p_1, p_2, N\}$, dessa maneira obtemos

$$G = \nabla N^T P = \nabla N^T \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{3.12}$$

A avaliação da curvatura em questão é baseada em G, que é chamado de *tensor geométrico*. O traço da matriz G é $k_1 + k_2$ e a norma de Frobenius de G, denotada por $|G|_F$, é definida como $\sqrt{tr(GG^T)} = \sqrt{k_1^2 + k_2^2}$.

Sintetizando o que foi visto, enumeremos os passos necessários para calcular curvatura em pontos arbitrários de um campo escalar.

1. Calcular as derivadas parciais de primeira ordem e obter o gradiente g. Calcular $N = -g/|g|$ e $P = I - NN^T$.

2. Calcular as derivadas parciais de segunda ordem e obter a Hessiana. Calcular $G = -PHP/|g|$.

3. Calcular o traço T e a norma de Frobenius F de G e obter as curvaturas principais

$$k_1 = \frac{T + \sqrt{2F^2 - T^2}}{2} \text{ e } k_2 = \frac{T - \sqrt{2F^2 - T^2}}{2}.$$

Lembramos que a curvatura gaussiana é definida como $K = k_1 \cdot k_2$ e a curvatura média como $H = (k_1 + k_2)/2$. A implementação dos algoritmos vistos aqui será discutida no capítulo 5.

Capítulo 4

Visualização de Objetos Implícitos Baseada em GPU

Nos últimos anos, acompanhamos um desenvolvimento formidável dos *hardwares* especializados no processamento gráfico impulsionado pela indústria de jogos e filmes de animação. Essas aplicações exigem cada cada vez mais realismo e, consequentemente, o processamento de um volume cada vez maior de informações em um curto espaço de tempo.

Existe um cenário de pesquisa crescente em torno desses equipamentos e para facilitar a tarefa de programação, antes realizada utilizando linguagem de montagem (*assembly*), diversas linguagens de alto nível foram criadas para programação de placas gráficas.

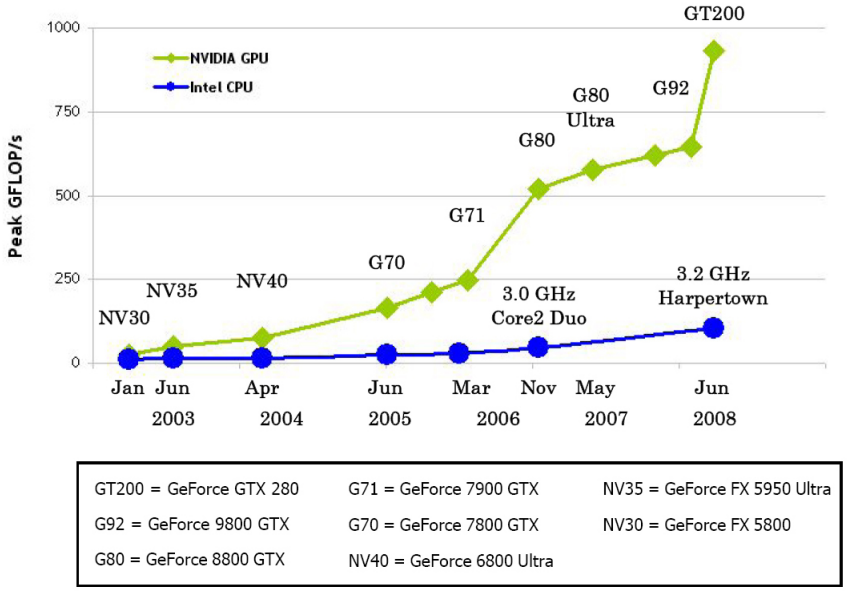
Neste capítulo, conheceremos um pouco mais sobre as possibilidades oferecidas pelas placas gráficas e as restrições impostas por sua estrutura, que privilegia o alto grau de paralelismo no processamento ao custo de uma capacidade menor de armazenamento de dados. Faremos uma breve descrição de uma das linguagens mais utilizadas na programação desses processadores, a *OpenGL Shading Language* [21], que foi utilizada no desenvolvimento deste trabalho. Por fim, será apresentado um *framework* extensível especializado em aplicações de visualização utilizando *hardware* gráfico chamado *Voreen*, para o qual foram desenvolvidos módulos para análise dos problemas envolvendo o cálculo de curvatura.

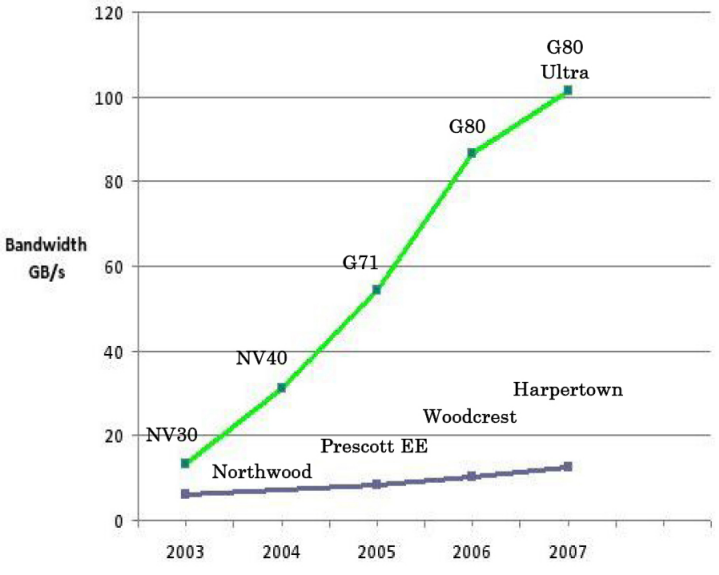
4.1 GPU

A unidade gráfica de processamento (GPU) é um tipo especial de processador criado para poupar a unidade lógica de processamento (CPU) das tarefas relacionadas à visualização. Essa unidade está presente em diversos dispostivos, de computadores pessoais a video games, e por sua especificidade possui uma capacidade de processamento gráfico muito maior se comparada aos processadores comuns. Além disso, a demanda por esse tipo de

processamento só tem aumentado com a exigência cada vez mais comum de respostas em tempo real para a geração de gráficos em alta definição.

Com esse objetivo, as placas gráficas têm evoluído muito ao longo dos anos para proporcionar um grau cada vez maior de paralelismo às aplicações com o uso de processamento *multithread* em múltiplos precessadores com enorme poder de processamento e memórias de acesso super rápido.





.

Figura 4.1: Gráficos de comparação entre peformances da CPU e da GPU. O primeiro gráfico mostra como a GPU tem obtido um crescimento muito maior da sua capacidade de processamento em relação à CPU enquanto o segundo traz uma comparação do crescimento na capacidade de transferência de dados em GB/s (dados obtidos de [17]).

A razão para a discrepância entre as operações com valores em ponto flutuante, pre-

sente no gráfico da figura 4.1, entre a CPU e a GPU está na diferença de arquitetura existente entre elas. Esta última foi projetada para a realização de cálculos intensivos com alto grau de paralelismo. Por esse motivo, são desenvolvidas com um número muito maior de transistores dedicados ao processamento de dados em relação à sua capacidade de armazenamento, como ilustrado na imagem 4.2.

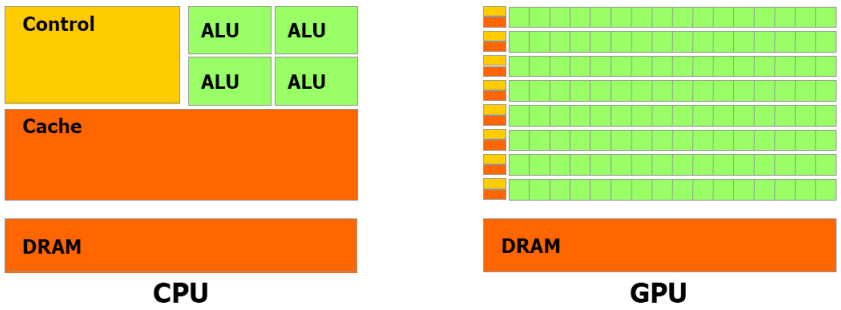


Figura 4.2: Diferença de arquitetura entre as GPUs e as CPUs, em verde as unidades lógicas e aritméticas que são responsáveis pelo processamento dos dados e em laranja as unidades de memória.

4.2 OpenGL Shading Language

OpenGL Shading Language, informalmente conhecida como GLSL, é uma linguagem de alto nível para programação de placas gráficas baseada na sintaxe das linguagens C e C++. GLSL possui a vantagem de ter sido incorporada ao OpenGL [23], uma API para processamento gráfico multiplataforma presente na maioria dos equipamentos existentes.

Para entendermos um pouco melhor a estrutura da GLSL analisemos a figura 4.3. Os elementos que aparecem na imagem descrevem as etapas que constituem o *pipeline* de renderização que é responsável por transformar primitivas geométricas em imagens. Primeiro, as primitivas que representam o modelo a ser visualizado são passadas como entrada. Em um primeiro passo, os vértices do modelo passam por um processo de transformação de coordenadas que os transporta do sistema de coordenadas do mundo para o da câmera. Além disso, se existe uma fonte de luz na cena, uma primeira mudança nas cores dos vértices é realizada seguindo um modelo de iluminação como o descrito no capítulo 2. Para acelerar o processo de renderização, os vértices situados fora do campo de visão do observador são eliminados em um processo conhecido como *clipping*. Esta etapa do *pipeline* está representada no diagrama da figura 4.3 como o passo de transformação nos vértices.

O último estágio do *pipeline* consiste em unir as primitivas em um processo chamado de *rasterização* que indica quais *pixels* da tela elas cobrem. Um fragmento é gerado para cada *pixel*, cada um deles contendo informações dos vértices aos quais está ligado como cor e coordenadas de textura.

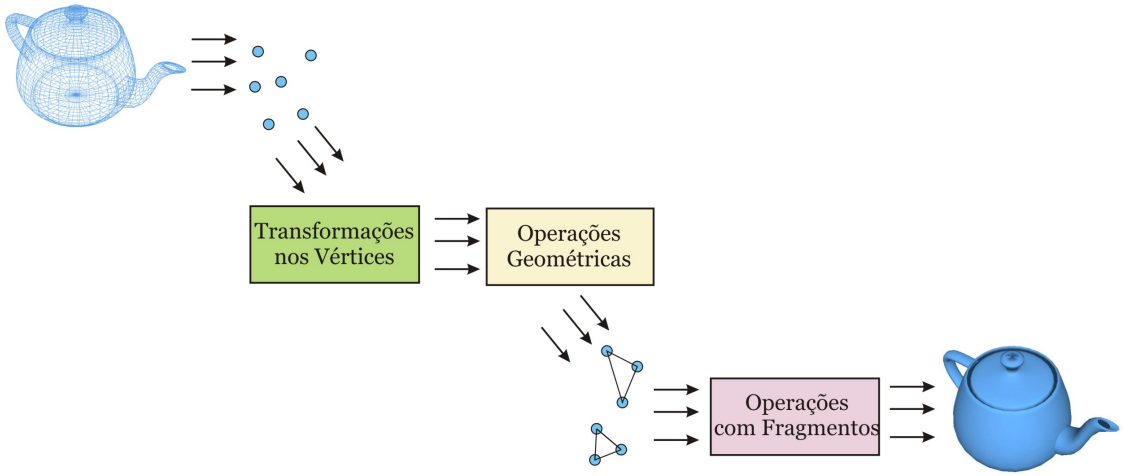


Figura 4.3: *Pipeline* de renderização.

O *pipeline* de renderização é composto por elementos chamados de processadores. Atualmente, existem três tipos de processadores, *vertex processors*, *geometry processsors* e *fragment processors*. Estes processadores estão representados na figura 4.4 que mostra também que as etapas do processo de visualização são independentes, ou seja, os vértices podem ser processados ao mesmo tempo em que *pixels* estão sendo processados. Além da independência entre as etapas, dentro de um mesmo estágio do *pipeline* os elementos podem ser processados simultaneamente por processadores diferentes.

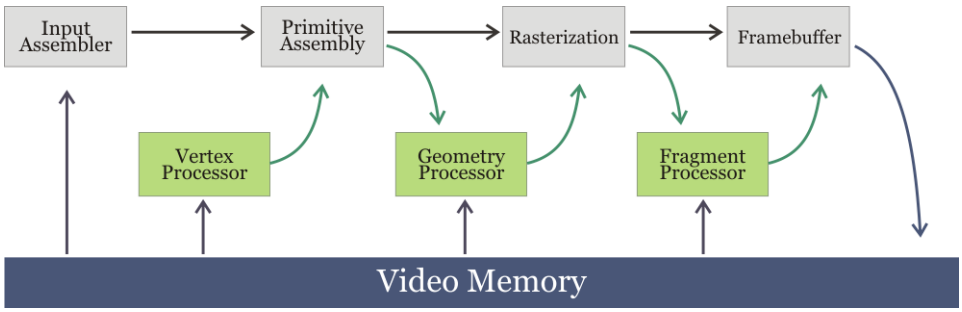


Figura 4.4: Fluxo de dados no *pipeline* de renderização

Nas próximas linhas descreveremos melhor cada um dos processadores do ponto de vista da GLSL.

4.2.1 Vertex Processor

O *vertex processor* é uma unidade programável que processa vértices e os atributos relacionados a eles. Os programas escritos para rodar nesses processadores são conhecidos como *vertex shaders*.

Esses elementos podem ser usados com várias finalidades como transformação de vértices, transformação e normalização de vetores normais, geração e transformação de coordenadas de textura etc. Entretanto, apenas um vértice é processado por vez, não sendo

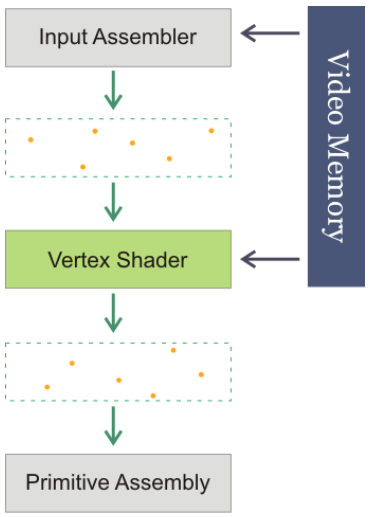


Figura 4.5: Etapa do pipeline de renderização relacionada ao vertex shader

possível executar operações que dependam das informações contidas em uma série de vértices simultaneamente. O código fonte a seguir mostra um exemplo simples de *vertex shader*.

```
1 // Vertex Shader Main
2 void main ( void ) {
3         // Pass vertex color to next stage
4         gl_FrontColor = gl_Color;
5         // Transform vertex position before passing it
6         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex ;
7 }
```

Código Fonte 4.1: Exemplo de *Vertex Shader*

Assim como na linguagem C, para cada *shader* uma função principal é obrigatória e a sua declaração é análoga. Nas linhas seguintes, a cor do vértice é delegada a um próximo passo e a posição é transformada para o espaço da tela.

4.2.2 Geometry Processor

Um *geometry processor* é uma unidade responsável pelo processamento de vértices que fazem parte de uma mesma primitiva, linhas, triângulos etc., e tem como saída uma sequência de vértices formando outras primitivas. Os programas escritos para rodar nesse tipo de unidade são chamados de *geometry shaders*.

Continuando nosso exemplo, o código abaixo exemplifica um *geometry shader*.

```
1 #extension GL_EXT_geometry_shader4: enable
2 void main() {
3         // Iterates over all vertices in the input primitive
4         for (int i = 0; i < gl_VerticesIn; ++i){
5                 // Pass color and position to next stage
6                 gl_FrontColor = gl_FrontColor[i];
```

```
7                 gl_Position = gl_PositionIn[i];
8                 // Done with this vertex
9                 EmitVertex();
10            }
11            // Done with the input primitive
12            EndPrimitive();
13    }
```

Código Fonte 4.2: Exemplo de *Geometry Shader*

A primeira linha do código, habilita a extensão *geometry shader* definida pelo *shader model 4.0*. Essa diretiva é sempre necessária ao utilizar um *geometry shader*. A função principal possui um *loop* cujo fator de iteração é o número de vértices da primitiva.

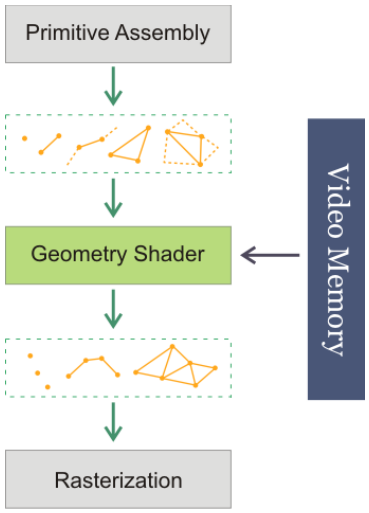


Figura 4.6: Etapa do pipeline de renderização relacionada ao *geometry shader*

4.2.3 Fragment Processor

Um *fragment processor* é uma unidade que opera sobre fragmentos e os dados associados a eles. Os programas escritos para esses processadores são chamados de *fragment shaders*. Cada iteração recebe a cor e a posição do vértice correspondente e associa esses valores à cor e à posição dos vértices de saída. A função `EmitVertex`, como o próprio nome indica, emite o vértice para saída, enquanto a função `EndPrimitive` encerra a primitiva.

Um *fragment shader* não pode alterar a posição de um fragmento. O acesso a fragmentos vizinhos também não é permitido. Os valores calculados nesses *shaders* são utilizados para atualizar dados na memória de um *framebuffer* ou de uma textura.

Esse tipo de processador é utilizado para realizar operações gráficas como as de interpolação, acesso a textura, soma de cores etc. As linhas de código a seguir trazem um exemplo simples de *fragment shader*.

```
1  // Fragment Shader Main
2  void main ( void ) {
```

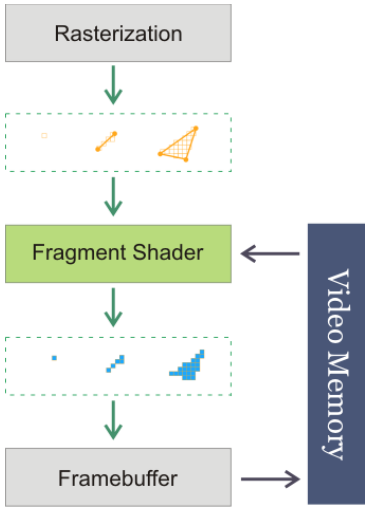


Figura 4.7: Etapa do pipeline de renderização relacionada ao *fragment shader*

```
3         // Pass fragment color
4         gl_FragColor = gl_Color;
5 }
```

Código Fonte 4.3: Exemplo de *Fragment Shader*

Nesse exemplo, o identificador `gl_Color` refere-se a um tipo diferente do mostrado no *vertex shader*. No primeiro caso, `gl_Color` foi associado à saída do *pipeline* como um atributo do vértice. No caso do *fragment shader* ele foi definido dentro do *pipeline* como um valor interpolado.

A imagem 4.8 mostra a saída do *pipeline* descrito acima aplicado ao volume `smile`, onde o resultado é apenas a silhueta do objeto visto que, para efeito de simplificação, nenhum cálculo de iluminação foi definido.

Mais detalhes e alguns exemplos básicos da utilização da linguagem GLSL podem ser encontrados em [13].

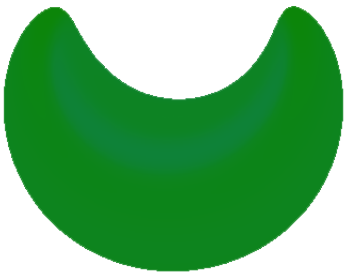


Figura 4.8: Resultado da execução dos shaders

4.3 GPGPU e CUDA

O crescimento vertiginoso dos *hardwares* gráficos provocou o surgimento de um outro tipo de demanda para as GPUs, agora os desenvolvedores de *softwares* empregam o seu alto poder de processamento em atividades diversas que lidam com a manipulação de grandes volumes de dados. Com o objetivo de aproveitar todo o potencial oferecido pelos múltiplos processadores das placas gráficas algumas ferramentas foram criadas em uma vertente de pesquisa denominada GPGPU [6] (sigla para *General-Purpose Computation on Graphics Hardware*) que vem crescendo em várias áreas como simulação física e processamento de sinais.

Foi com esse pensamento que a NVIDIA [18] apresentou em novembro de 2006 uma nova arquitetura e linguagem para programação paralela em GPU chamada CUDA [17]. O objetivo dessa ferramenta é desenvolver aplicações que se adaptem bem à GPU, ou seja, que possam ser divididas em núcleos de processamento independentes. Assim como GLSL, CUDA foi desenvolvida para programadores familiarizados com a linguagem C e em breve dará suporte a outras linguagens como C++ e FORTRAN.

A linguagem CUDA foi brevemente testada durante o desenvolvimento deste trabalho, mas, para os fins em que foi empregada, não ofereceu nenhuma vantagem que justificasse seu uso em um primeiro momento. Apesar disso, existe um grande potencial para a utilização dessa ferramenta em algoritmos baseados em lançamento de raios, já que na maioria dos casos o processamento em cada pixel é independente. Portanto, a possibilidade de utilizarmos a linguagem para o processamento de curvatura em superfícies, bem como para outras finalidades, não foi descartada.

4.4 Voreen: Modularidade

Esta seção é dedicada à descrição do sistema utilizado nesse trabalho para desenvolver os módulos de visualização utilizados no estudo dos problemas envolvidos no cálculo de curvatura. O *Voreen* (contração de *Volume Rendering Engine*) é um sistema *open source* desenvolvido em C++ sob os termos da GPL 2.0 [4]. O objetivo dessa ferramenta é fornecer o acesso a algoritmos de *rendering* de volumes e processamento de imagens baseados em GPU que permitem a visualização interativa de conjuntos de dados volumétricos de maneira flexível. Essa flexibilidade é resultante da característica modular do *framework*, que permite a associação de várias técnicas de visualização através de estruturas chamadas de processadores (*processors*). A figura 4.9 mostra uma visão geral do sistema.

O *rendering* de volumes é realizado através da criação de redes (*networks*) que são constituídas de processadores, cada um deles responsável por uma tarefa específica. O usuário pode facilmente estender esses elementos para, por exemplo, integrar novos algoritmos de processamento de volumes.

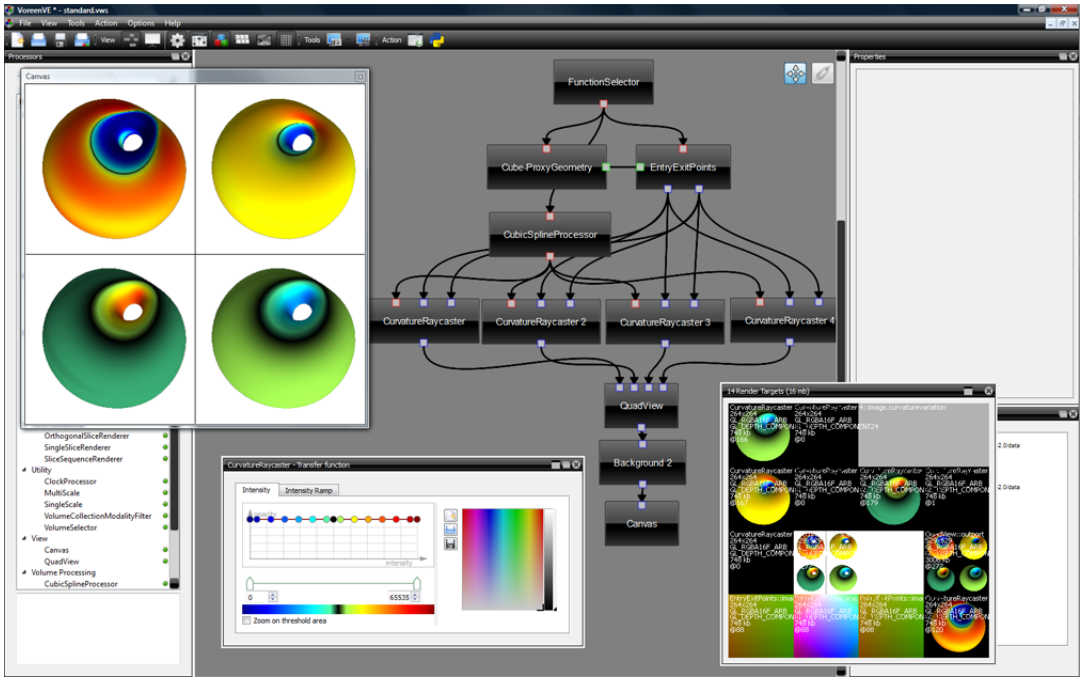


Figura 4.9: Visão Geral do Voreen

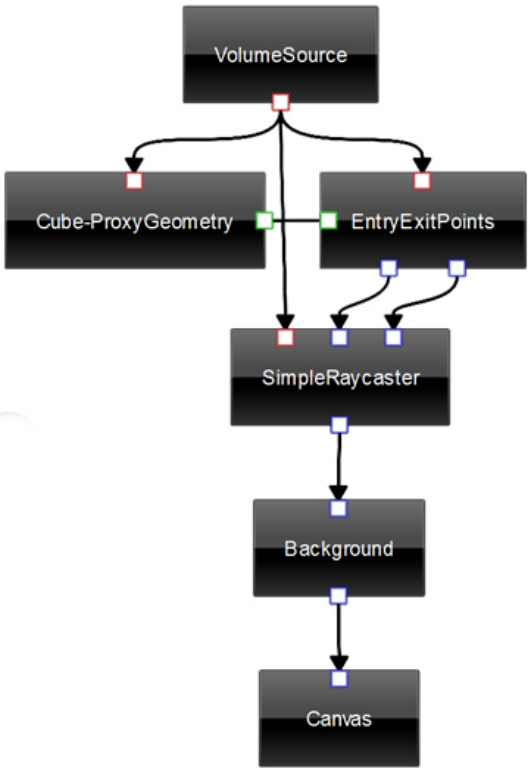


Figura 4.10: Rede de processadores.

Os dados são trocados entre os diferentes processadores através de portas, *inport* para receber dados e *outport* para fornecê-los. Os tipos de porta indicam as conexões possíveis de entrada/saída entre os processadores, apenas portas de dados compatíveis podem ser conectadas. Na imagem 4.10, os quadrados coloridos representam os diferentes tipos de

portas interligados pelas setas em preto.

O Voreen foi construído seguindo o paradigma de orientação a objetos (OO) onde cada objeto possui sua própria maneira de controlar e manipular dados. Esse paradigma não se aplica à arquitetura do OpenGL que tem seu estado determinado por variáveis globais e onde os dados podem ser acessados em qualquer parte do programa. Sendo assim, os desenvolvedores precisam ser cuidadosos ao escolher entre seguir os princípios de OO ou obter alta peformance em processamento gráfico.

4.4.1 Classes de Processadores

Nos próximos parágrafos serão apresentadas algumas classes importantes de processadores do Voreen. No capítulo 4 serão descritos os principais processadores implementados durante o desenvolvimento deste trabalho como partes constituintes dos módulos criados para estudo de curvatura e métodos de interpolação.

O processador `VolumeSource` permite ao usuário criar seu próprio conjunto de volumes de dados incluindo arquivos de tipos suportados pelo Voreen [26]. A partir desse conjunto, volumes podem ser facilmente selecionados para visualização.

Um `EntryExitPoints` cria imagens contendo a posição de pontos de entrada e saída armazenados como cores. Esses pontos, que posteriormente serão armazenados como texturas, serão utilizados no processo de visualização direta como dados de entrada do algoritmo. Para volumes em forma de cubo, os dados armazenados nessas texturas formam um cubo de cores RGB (figura 4.11).

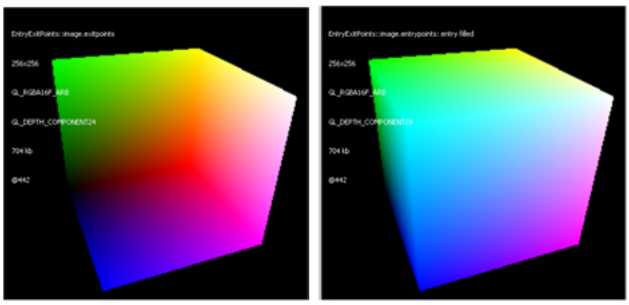


Figura 4.11: Texturas com os pontos de entrada e saída do volume em forma de cubo.

Finalmente, os processadores do tipo `RayCaster` recebem como entrada os pontos de entrada e saída, assim como o volume de dados. A partir desse momento, o algoritmo de *ray casting* é executado em um *fragment program* na GPU usando a linguagem GLSL.

Cada um dos processadores mencionados possui uma série de propriedades que podem ser alteradas para gerar diferentes tipos de visualização em tempo real. Essas propriedades podem ter seus valores compartilhados entre processadores para que a alteração em seu estado tenha efeito em todos que a compartilham. A figura 4.12 mostra dois processadores que compartilham as mesmas propriedades de câmera.

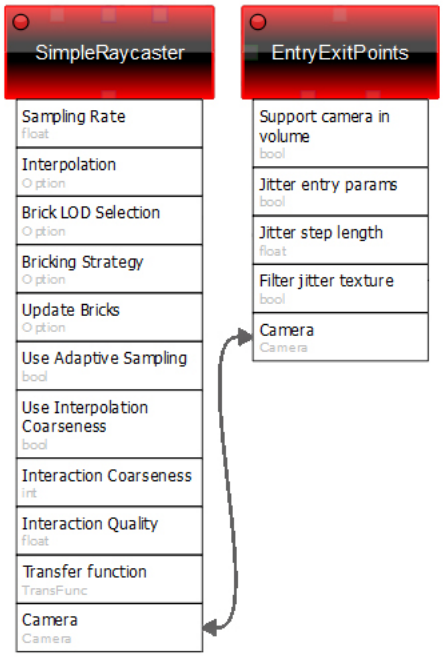


Figura 4.12: Propriedade `camera` compartilhadas entre componentes.

Existem muitos outros processadores que podem ser associados aos mencionados para criar diferentes resultados. Optou-se por omitir esses processadores pois eles não são o foco deste trabalho. Para mais informações consultar [15] que possui uma descrição mais detalhada do *framework* ou o site do projeto que possui entre outras informações, o código fonte do sistema para análise e possíveis colaborações [26].

4.4.2 Criando Processadores

Nesta seção será apresentado um exemplo simples de criação de componente para o Voreen. Este exemplo será importante para o entendimento do capítulo 5 que apresentará os módulos desenvolvidos para esse trabalho e agregados ao Voreen.

O processador em questão terá como tarefa converter uma imagem em RGB para uma imagem em escala de cinza. Este processador está implementado no Voreen com o nome de `Grayscale` relacionado ao grupo de compenentes para processamento de imagens.

```
1  Grayscale::Grayscale()
2      : ImageProcessor("pp_grayscale"), // loads fragment shader pp_greyscale.frag
3        saturation_("saturation", "Saturation", 0.0f),
4        inport_(Port::INPORT, "inport"),
5        outport_(Port::OUTPORT, "outport")
6  {
7      // register properties and ports:
8      addProperty(saturation_);
9      addPort(inport_);
10     addPort(outport_);
11 }
12
```

```
13 void Grayscale::process() {
14     // activate and clear output render target
15     outport_.activateTarget();
16     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
17     // bind result from previous processor
18     inport_.bindTextures(tm_.getGLTexUnit(shadeTexUnit_), tm_.getGLTexUnit(depthTexUnit_)
           );
19     // activate shader and set uniforms:
20     program_->activate();
21     setGlobalShaderParameters(program_);
22     program_->setUniform("shadeTex_", tm_.getTexUnit(shadeTexUnit_));
23     program_->setUniform("depthTex_", tm_.getTexUnit(depthTexUnit_));
24     program_->setUniform("saturation_", saturation_.get());
25     // render screen aligned quad:
26     glDepthFunc(GL_ALWAYS);
27     renderQuad();
28     glDepthFunc(GL_LESS);
29     program_->deactivate();
30     glActiveTexture(GL_TEXTURE0);
31     LGL_ERROR;
32 }
```

Código Fonte 4.4: Este exemplo mostra a criação de um processador para criar imagens em escala de cinza.

No código fonte apresentado acima iniciaremos explicando o que acontece no método `process` que deve ser implmententado por todas as classes que herdam de `Processor`, esse é o casso da classe `GrayScale`. A linha 15 ativa a textura que será passada como saída do processador. A linha 18 registra as texturas que serão passadas como dado de entrada para o *fragment shader* `pp_grayscale` na linha 22 através da chamada ao método `setUniform`. As linhas 20 e 29 alteram o estado do objeto `program_` ativando e desativando o processamento do código do *fragment shader* em questão. O primeiro método apresentado no código é o construtor do processador que inicializa alguns atributos do objeto e as linhas 9 e 10 criam as portas que servirão de comunicação entre os processadores para troca de informações, uma porta de entrada para a imagem original e uma porta de saída para a imagem em tons de cinza.

```
1  uniform SAMPLER2D_TYPE shadeTex_;
2  uniform SAMPLER2D_TYPE depthTex_;
3  uniform TEXTURE_PARAMETERS textureParameters_;
4  uniform float saturation_;
5
6  vec4 toGrayScale(in vec4 color, in float saturation) {
7      float brightness = (0.30 * color.r) + (0.59 * color.g) + (0.11 * color.b);
8      vec4 grayscale = vec4(brightness, brightness, brightness, color.a);
9      return mix(grayscale, color, saturation);
10 }
11
12 void main() {
13     vec2 p = gl_FragCoord.xy * screenDimRCP_;
14     gl_FragColor = toGrayScale(textureLookup2Dnormalized(shadeTex_, textureParameters_, p
           ), saturation_);
```

```
15     gl_FragDepth = textureLookup2Dnormalized(depthTex_, textureParameters_, p).z;
16 }
```

Código Fonte 4.5: *Fragment shader* associado ao processador *Grayscale*.

No código do *fragment shader* acima as primeiras linhas são declarações de variáveis que armazenarão as informações de textura e uma váriavel `float` que guardará um valor de saturação. Todos esses dados são provenientes do processador `Grayscale` e serão apenas processados pelo *shader* sem sofrer nenhuma alteração.

A linha 13 recupera a coordenada do *pixel* que está sendo processado enquanto a linha 14 faz a chamada ao método que irá converter a cor original do *pixel* em tons de cinza e passar essa informação adiante.

Capítulo 5

Módulos

Para este trabalho foram desenvolvidos alguns módulos empregados no cálculo de curvatura e na avaliação de funções utilizando interpolação tricúbica. Esses módulos foram incorporados ao *framework* Voreen na forma de processadores. O objetivo desse capítulo é descrever os aspectos mais relevantes da construção desses processadores e apresentar os algoritmos desenvolvidos utilizando a teoria dos capítulos iniciais.

5.1 Módulo de Funções

Este processador foi criado para eliminar a necessidade de leitura de arquivos com os dados volumétricos. Não há nenhuma entrada para esse componente e a sua saída é uma textura com um volume de dados indicado pelo usuário através de uma de suas propriedades. Os volumes são gerados através de funções que descrevem o volume implicitamente. A imagem 5.1 mostra o processador em destaque com as suas propriedades que permitem escolher o tipo utilizado para armazenar os valores (inteiro de 8 e 16 bits e ponto flutuante) e a resolução do volume (entre 2^4 e 2^8). Já a imagem 5.1 traz alguns exemplos de volumes gerados com o processador ao qual demos o nome de `FunctionSelector`.

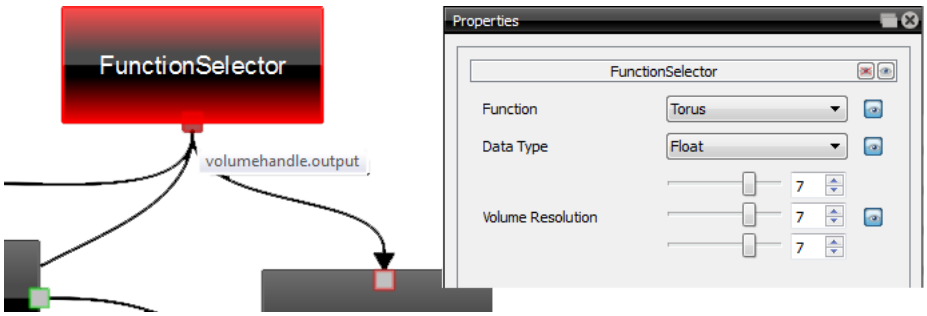
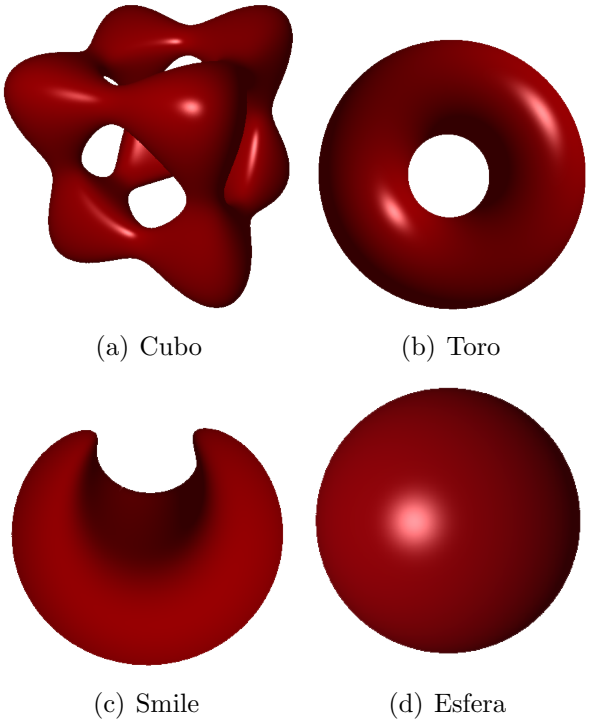


Figura 5.1: `FunctionSelector`



(a) Cubo (b) Toro

(c) Smile (d) Esfera

Figura 5.2: Imagens Geradas com o processador `FunctionSelector`

5.2 Módulo de Interpolação Tricúbica

O módulo de interpolação tricúbica encontra-se implementado na classe `TricubicSplineProcessor`, esta classe implementa o processador `CubicSplineProcessor` (em destaque na figura 5.3). A entrada desse processador é um volume de dados como o descrito nos capítulos anteriores e saída é um outro volume com as mesmas características. Entretanto, os valores dos *voxels* são convertidos em coeficientes *B-Splines* que serão utilizados para a avaliação dos valores da função nos pontos do volume utilizando interpolação tricúbica.

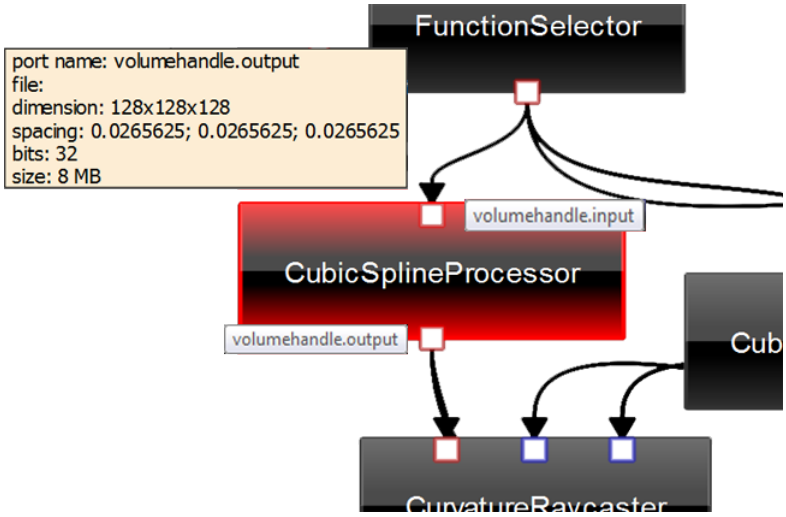


Figura 5.3: `CubicSplineProcessor`

A única propriedade do processador habilita ou desabilita o processamento do volume.

5.3 Módulos de Cálculos em GLSL (interpolação tricúbica e curvatura)

Esse módulos equivalem aos *shaders* criados para o cálculo de curvatura e para avaliação de funções utilizando interpolação tricúbica. Utilizaremos o código abaixo para descrever o que acontece no módulo de curvatura.

```
1  float matrix_trace(in mat3 m){
2          return m[0][0]+m[1][1]+m[2][2];
3  }
4
5  float frobenius_norm(in mat3 m){
6          float x = m[0][0]*m[0][0]+m[0][1]*m[0][1]+m[0][2]*m[0][2]+
7                    m[1][0]*m[1][0]+m[1][1]*m[1][1]+m[1][2]*m[1][2]+
8                    m[2][0]*m[2][0]+m[2][1]*m[2][1]+m[2][2]*m[2][2];
9          return sqrt(x);
10 }
11
12 mat3 calc_nnT(in vec3 n){
13         mat3 nnT = mat3(n.x*n.x, n.y*n.x, n.z*n.x, n.x*n.y, n.y*n.y, n.z*n.y, n.x*n.z, n.
               y*n.z, n.z*n.z);
14         return nnT;
15 }
16
17 mat3 calc_P(in mat3 nnT){
18         mat3 I = mat3(1.0);
19         return I-nnT;
20 }
21
22 //n_g - gradient norm
23 mat3 calc_G(in mat3 P,in mat3 H,in float n_g){
24         mat3 G = mat3(0.0);
25         G = (P*H*P)*(1/n_g);
26         return G;
27 }
28
29 vec2 computeCurvature(in vec3 grad, in mat3 hessian){
30         vec3 g    = grad;
31         float n_g = sqrt(dot(g,g));
32         g         = normalize(g);
33         vec3 n    = vec3(-g.x, -g.y, -g.z);
34     mat3 nnT  = calc_nnT(n);
35         mat3 P    = calc_P(nnT);
36         mat3 G    = calc_G(P, hessian, n_g);
37         float trace  = matrix_trace(G);
38         float f_norm = frobenius_norm(G);
39         float k1 = (trace + sqrt(max(0.0, 2*f_norm*f_norm-trace*trace)))/2.f;
40         float k2 = (trace - sqrt(max(0.0, 2*f_norm*f_norm-trace*trace)))/2.f;
41         return vec2(k1,k2);
42 }
```

Código Fonte 5.1: Módulo de cálculo de curvaturas.

Entre as linhas 1 e 10 estão duas funções extremamente simples, a primeira calcula o traço de uma matriz e o segundo a norma de Frobenius descritas no capítulo 3.

A linha 12 corresponde a declaração da função que realiza o cálculo resultante do produto entre as matrizes linha e coluna que representam o vetor normal N e a sua transposta N^T. O resultado é uma matriz 3×3 que em GLSL corresponde ao tipo `mat3`.

Na linha 23 está definido o cálculo da matriz G onde a função possui como parâmetros de entrada as matrizes P e H e a norma do gradiente.

A função `computeCurvature` realiza o cálculo das curvaturas principais e recebe como dados de entrada a matriz Hessiana e o gradiente. Este trecho do código apenas faz chamadas as funções definidas acima e nas linhas 39 e 40 calcula as curvaturas principais utilizando as fórmulas definidas no capítulo 3 onde

$$k_1 = \frac{T + \sqrt{2F^2 - T^2}}{2} \text{ e } k_2 = \frac{T - \sqrt{2F^2 - T^2}}{2}.$$

O código abaixo refere-se a implementação da avaliação de funções utilizando b-splines. Foram selecionados dois métodos que merecem destaque. No primeiro deles `interpolate_tricubic` o *loop* principal representa o somatório

$$f(x, y, z) = \sum_{i,j,k \in \mathbb{Z}} c[i, j, k] B_3(x - i) B_3(y - j) B_3(z - k),$$

definido no capítulo 3. O segundo, `interpolate_tricubic_fast`, implementa o método rápido de avaliação que faz apenas 8 consultas a memória de textura representados no código pela função `texture3D`.

```
1  float interpolate_tricubic(sampler3D tex, VOLUME_PARAMETERS volumeParameters, vec3 coord)
   {
2      vec3 n=volumeParameters.datasetDimensions_;
3      coord = n*coord;// - 0.5;
4      // shift the coordinate from [0,extent] to [-0.5, extent-0.5]
5          vec3 coord_grid = coord - 0.5;
6          vec3 index = floor(coord_grid);
7          vec3 fraction = coord_grid - index;
8          float result=0;
9          for (float z=-1; z < 2.5; z++)  //range [-1, 2] {
10         float b_z = bspline(z-fraction.z);
11                 for (float y=-1; y < 2.5; y++)          {
12             float b_y = bspline(y-fraction.y);
13                         for (float x=-1; x < 2.5; x++)                  {
14                                 float b_x = bspline(x-fraction.x);
15                 vec3 p     = (index+vec3(x,y,z)+0.5)/n;
16                 float c = texture3D(tex, p).a;
17                 result   += c*b_x*b_y*b_z;
18                         }
```

```
19                  }
20          }
21          return result;
22  }
23
24  float interpolate_tricubic_fast(sampler3D tex, VOLUME_PARAMETERS volumeParameters, vec3
        coord){
25          vec3 n=volumeParameters.datasetDimensions_;
26          coord = n*coord;// - 0.5;
27          // shift the coordinate from [0,extent] to [-0.5, extent-0.5]
28          vec3 coord_grid = coord - 0.5;
29          vec3 index = floor(coord_grid);
30          vec3 fraction = coord_grid - index;
31          vec3 w0, w1, w2, w3;
32          bspline_weights(fraction, w0, w1, w2, w3);
33          vec3 g0 = w0 + w1;
34          vec3 g1 = w2 + w3;
35          vec3 h0 = (w1 / g0) - 0.5 + index;  //h0 = w1/g0 - 1, move from [-0.5, extent
                -0.5] to [0, extent]
36          vec3 h1 = (w3 / g1) + 1.5 + index;  //h1 = w3/g1 + 1, move from [-0.5, extent
                -0.5] to [0, extent]
37          h0 = (h0)/n;
38          h1 = (h1)/n;
39          // fetch the eight linear interpolations
40          // weighting and fetching is interleaved for performance and stability reasons
41          float tex000 = texture3D(tex, vec3(h0.x, h0.y, h0.z)).a;
42          float tex100 = texture3D(tex, vec3(h1.x, h0.y, h0.z)).a;
43          tex000 = mix(tex100, tex000, g0.x);  //weigh along the x-direction
44          float tex010 = texture3D(tex, vec3(h0.x, h1.y, h0.z)).a;
45          float tex110 = texture3D(tex, vec3(h1.x, h1.y, h0.z)).a;
46          tex010 = mix(tex110, tex010, g0.x);  //weigh along the x-direction
47          tex000 = mix(tex010, tex000, g0.y);  //weigh along the y-direction
48          float tex001 = texture3D(tex, vec3(h0.x, h0.y, h1.z)).a;
49          float tex101 = texture3D(tex, vec3(h1.x, h0.y, h1.z)).a;
50          tex001 = mix(tex101, tex001, g0.x);  //weigh along the x-direction
51          float tex011 = texture3D(tex, vec3(h0.x, h1.y, h1.z)).a;
52          float tex111 = texture3D(tex, vec3(h1.x, h1.y, h1.z)).a;
53          tex011 = mix(tex111, tex011, g0.x);  //weigh along the x-direction
54          tex001 = mix(tex011, tex001, g0.y);  //weigh along the y-direction
55          return mix(tex001, tex000, g0.z);  //weigh along the z-direction
56  }
```

Código Fonte 5.2: Módulo de avaliação de funções através de interpolação tricúbica.

Antes de encerrar a seção, existe um fato importante que deve ser mencionado. Durante o processo de visualização, a função de transferência utiliza valores no intervalo $[0, 1]$ para acessar os valores de cor e opacidade associados aos pontos. Por esse motivo, é necessária a normalização dos valores de curvatura encontrados já que eles possivelmente não se encaixarão nesse intervalo. Dois tipos de mormalização foram implementados.

O primeiro método para a normalização é linear e utiliza os valores de curvatura positiva máxima ($k\_max = max\{k, 0\}$) e o valor de curvatura negativa mínima ($k\_min = min\{k, 0\}$) para o cálculo, onde a curvatura k varia entre todos os valores de curvatura

encontrados. O valor normalizado k_n de k é então dado por

$$k_n = k/min\_k \text{ se } k < 0 \text{ e } k = k/max\_k \text{ se } k \geq 0.$$

O segundo método adota uma estratégia não linear. O valor de k_n é dado utilizando a função exponencial. Assim,

$$k_n = (e^{k \cdot t})/2 \text{ se } k < 0 \text{ e } k_n = 1 - (e^{-k \cdot t})/2 \text{ caso contrário.}$$

O valor t é uma constante de atenuação.

5.4 Módulo de Renderização de Isosuperfícies

O módulo de renderização de isosuperfícies está implementado no processador `CurvatureRaycaster` e utiliza o algoritmo de *ray casting* descrito na seção 2.2 do capítulo 2 para realizar a visualização de isosuperfícies. A navegação entre as diferentes isosuperfícies do volume é realizada através da alteração do parâmetro `level`. Este componente é semelhante ao já existente `VolumeRaycaster`, entretanto utiliza os valores de curvatura normalizados que foram calculados no módulo de curvatura e interpolação tricúbica como parâmetro para função de transferência enquanto o primeiro utiliza os valores de intensidades da função em cada um dos pontos. A imagem 5.4 traz o processador em destaque. Os dados de entrada do processador são texturas contendo o volume a ser visualizado além das texturas contendo os pontos de entrada e saída mencionadas no capítulo anterior.

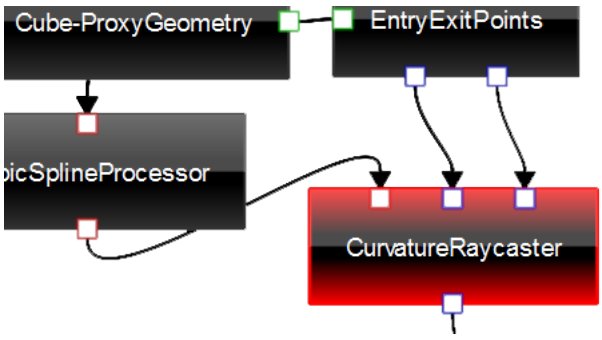


Figura 5.4: `CurvatureRaycaster`

A figura 5.5 mostra um resultado obtido com o processador `CurvatureRaycaster`, além disso traz algumas de suas propriedades principais como aquelas que permitem editar a função de transferência (figura 5.6) e os que permitem escolher o nível a ser visualizado, o tipo de cálculo de derivadas (*forward differences, central differences, trilinear e tricubic*) e avaliação de função. Uma outra propriedade que pode ser alterada é o tipo de curvatura visualizada (Gaussiana, Média, k_1 e k_2).

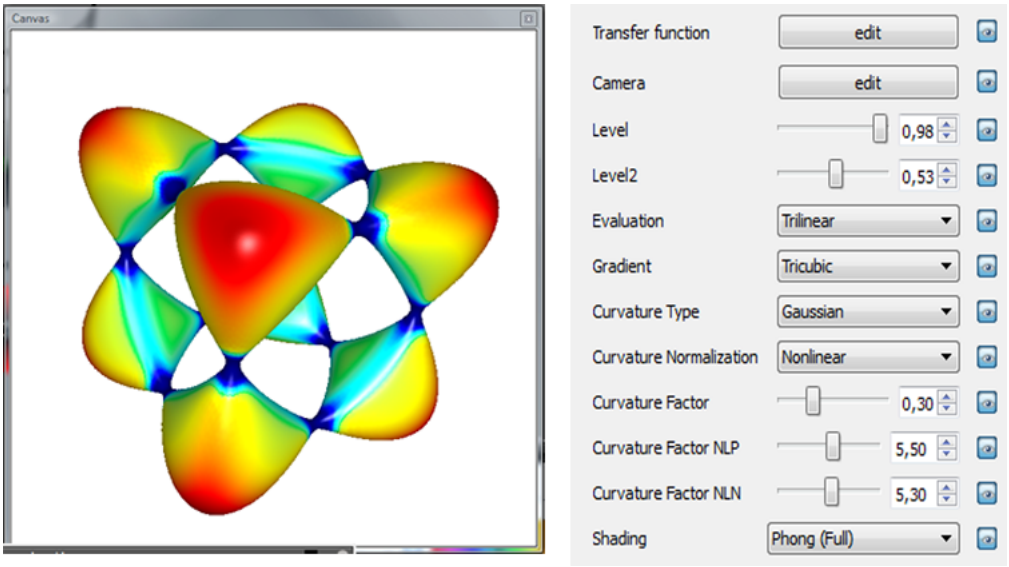


Figura 5.5: Resultado obtido com o `CurvatureRaycaster` e suas propriedades

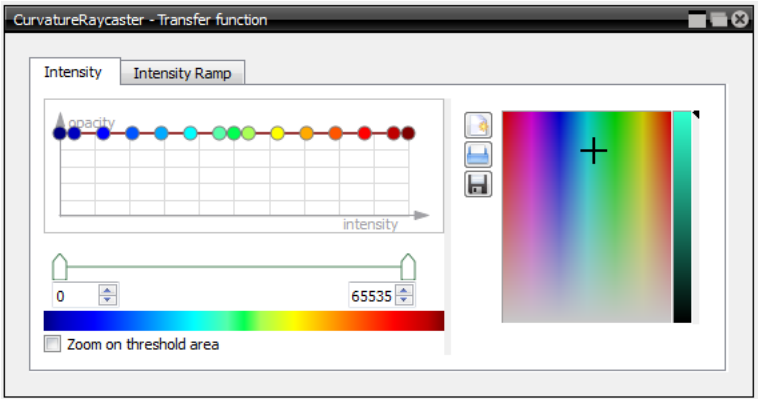


Figura 5.6: Editor para funções de transferência

5.5 Módulo de Renderização de Volumes

O módulo de renderização de volumes segue o algoritmo definido no capítulo 2. Ele está implementado no processador `CurvatureVolumeRaycaster` e é semelhante ao processador descrito na seção anterior, recebe como dados de entrada o volume e os pontos de entrada e saída. A saída desse processador é uma textura com a imagem do volume em diferentes valores de opacidade que dependem da superfície de nível. As cores, como no caso anterior, são determinadas pelos valores de curvatura normalizados. A figura 5.7 mostra o processador em destaque já a figura 5.8 traz o resultado obtido mostrando o efeito de transparência mencionado.

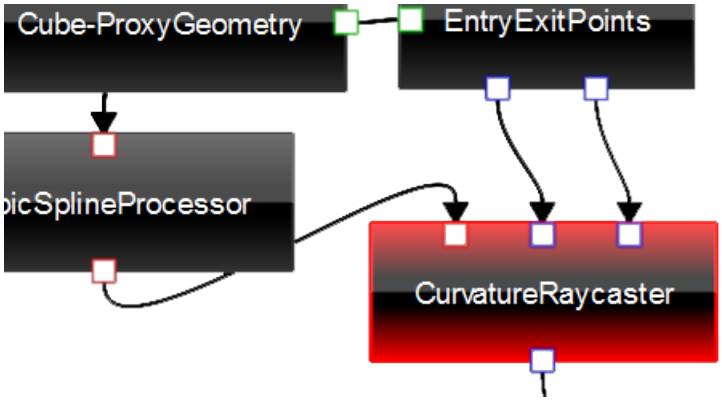


Figura 5.7: `CurvatureRaycaster`

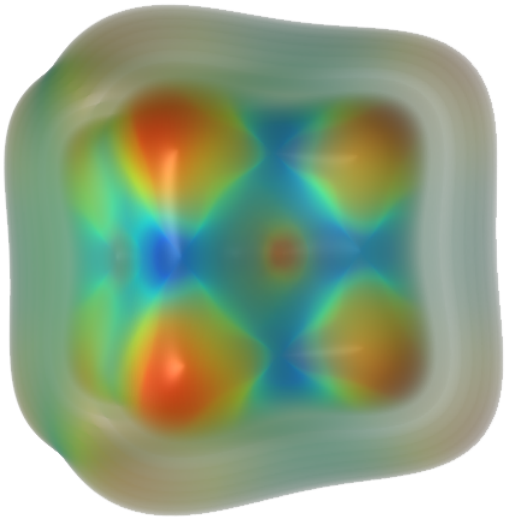
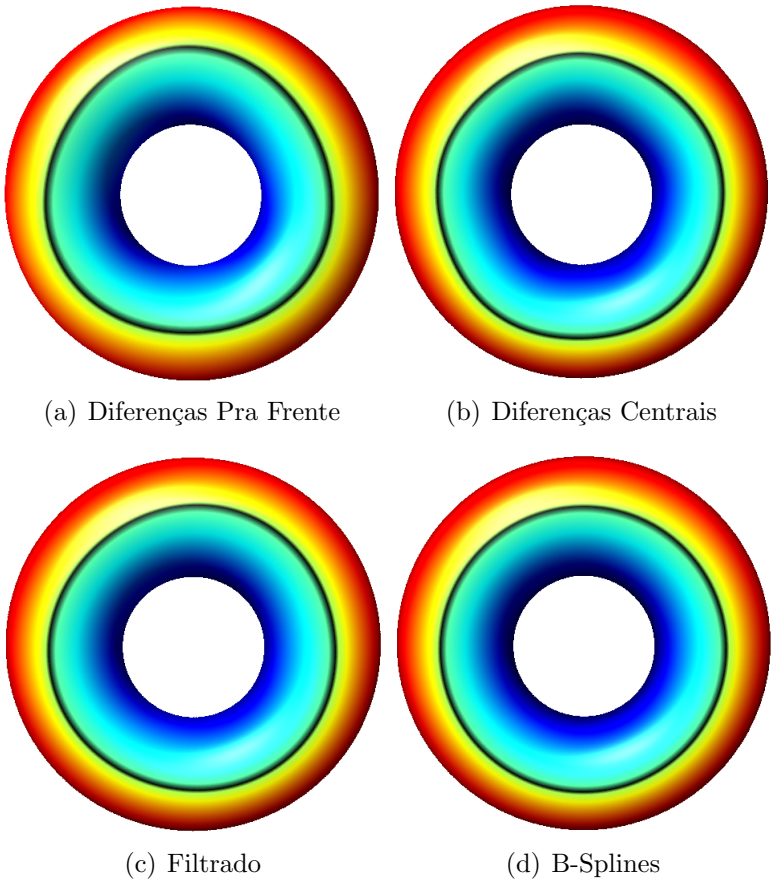


Figura 5.8: Resultado obtido com o `CurvatureVolumeRaycaster` mostrando diversas porções do volume com efeito de transparência e levando em consideração os valores de curvatura para o cálculo das cores.

Capítulo 6

Resultados

Este capítulo é dedicado à descrição dos resultados obtidos a partir dos módulos implementados e descritos no capítulo anterior.



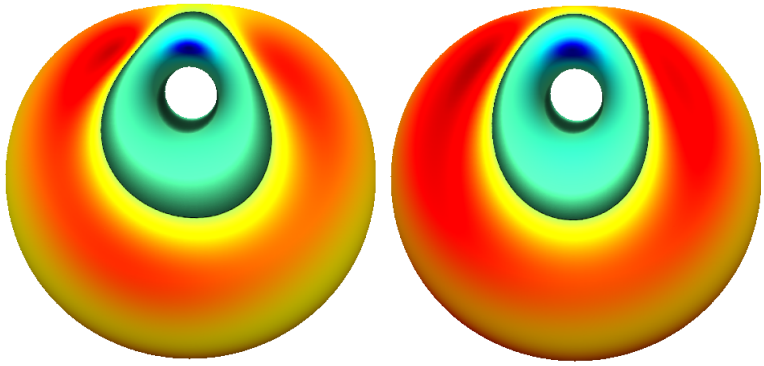
(a) Diferenças Pra Frente (b) Diferenças Centrais

(c) Filtrado (d) B-Splines

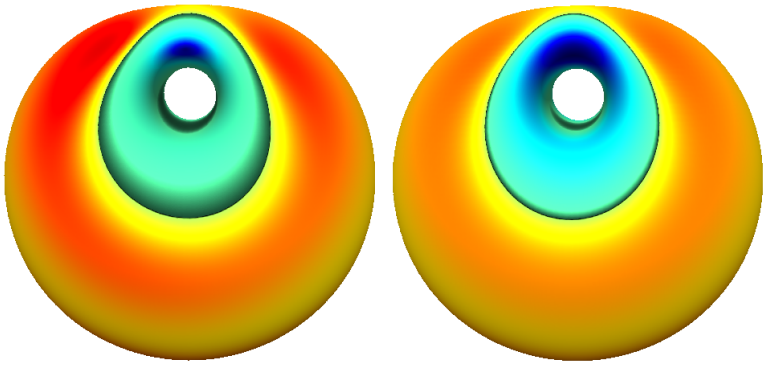
Figura 6.1: Resultados obtidos utilizando o mapa de cores JET para visualização de valores de curvatura. Vermelho indica curvaturas positivas, azul curvaturas negativas, verde valores próximos de zero, e em preto está delimitada a curva de nível zero.

Na imagem 6.1 podemos observar que há uma variação na simetria da linha preta que indica os valores onde a curvatura é nula sobre o toro. Na imagem 6.1(a), que utiliza diferenças finitas para frente para o cálculo das derivadas, aparece um padrão

triangular, em 6.1(b), que foi gerada utilizando diferenças centrais para o cálculo das derivadas, aparece um padrão em forma de pentágono. Na figura 6.1(c) foi utilizada uma filtragem que calcula a derivada através de uma ponderação dos valores nos *voxels* vizinhos e na 6.1(d) foi utilizada interpolação tricúbica para o cálculo das derivadas, em ambas o resultado é mais próximo da realidade, entretanto a interpolação tricúbica oferece melhores resultados. Nas três primeiras imagens a função foi avaliada utilizando interpolação trilinear e na última foi utilizada a interpolação tricúbica.



(a) Diferenças Pra Frente (b) Diferenças Centrais



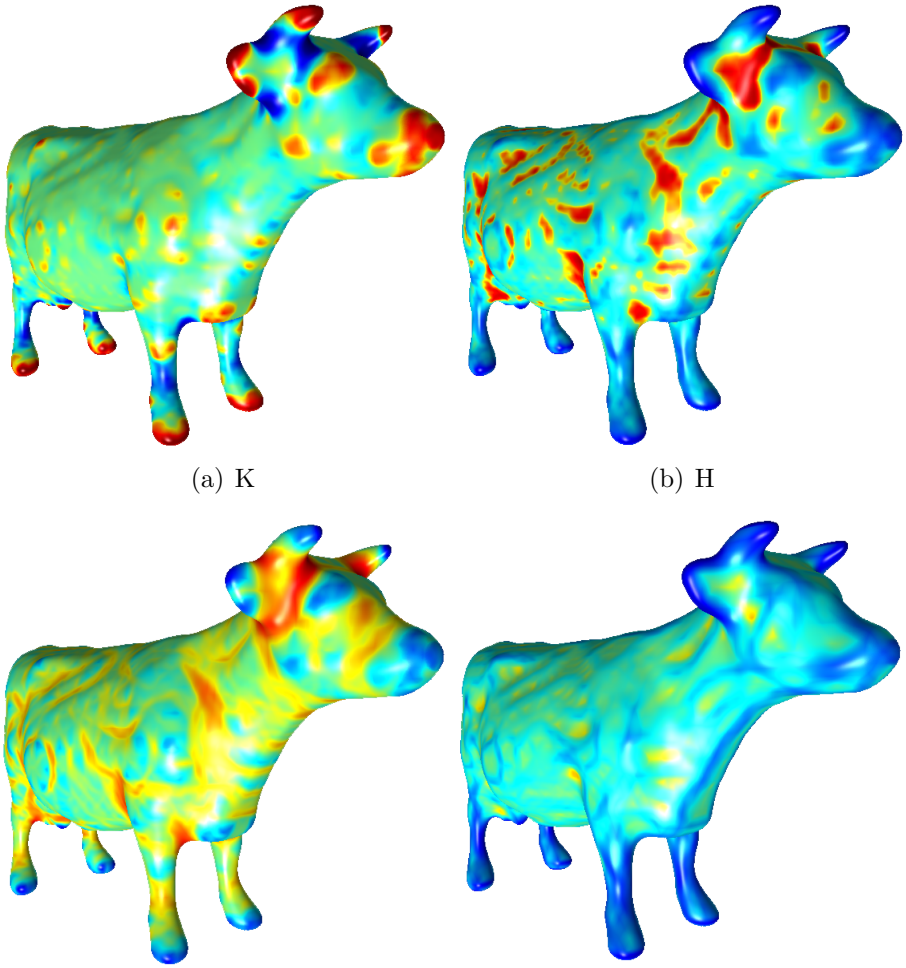
(c) Filtrado (d) B-Splines

Figura 6.2: Resultados obtidos utilizando o volume `smile`.

A imagem 6.2 mostra um resultado semelhante ao primeiro aplicado ao volume `smile`.

A imagem 6.3 mostra o resultado do *ray casting* de curvaturas aplicado ao volume `cow` que sofreu uma suavização em um passo de pré-processamento. A figura 6.3(a) mostra os valores da curvatura Gaussiana sobre a superfície enquanto a figura 6.3(b) exibe os valores de curvatura média. As cores das figuras abaixo das duas primeiras representam os valores das curvaturas principais. Assim como nos resultados do início do capítulo, a cor azul representa curvatura negativa, verde curvatura nula e vermelho curvatura positiva. Observe que há uma predominância da cor verde ou de valores muito próximos de zero, isso deve-se ao passo de suavização do volume. Algumas inconsistências também ocorrem devido aos ruídos provenientes do volume original.

A figura 6.4 exemplifica os problemas que surgem quando se utiliza diferentes formas

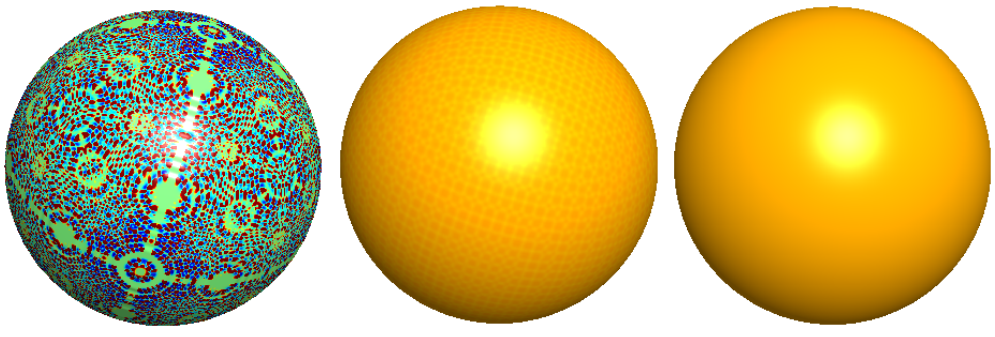


(a) K (b) H

(c) k_1 (d) k_2

Figura 6.3: Resultados obtidos utilizando o volume `cow`.

de representação numérica para visualização da curvatura Gaussiana. A figura 6.4(a) foi obtida utilizando inteiro de 8 bits para representação enquanto em 6.4(b) foram utlizados 16 bits para representação. Os melhores resultados ocorrem quando utilizamos 32 bits para representação de dados (figura 6.4(c)).

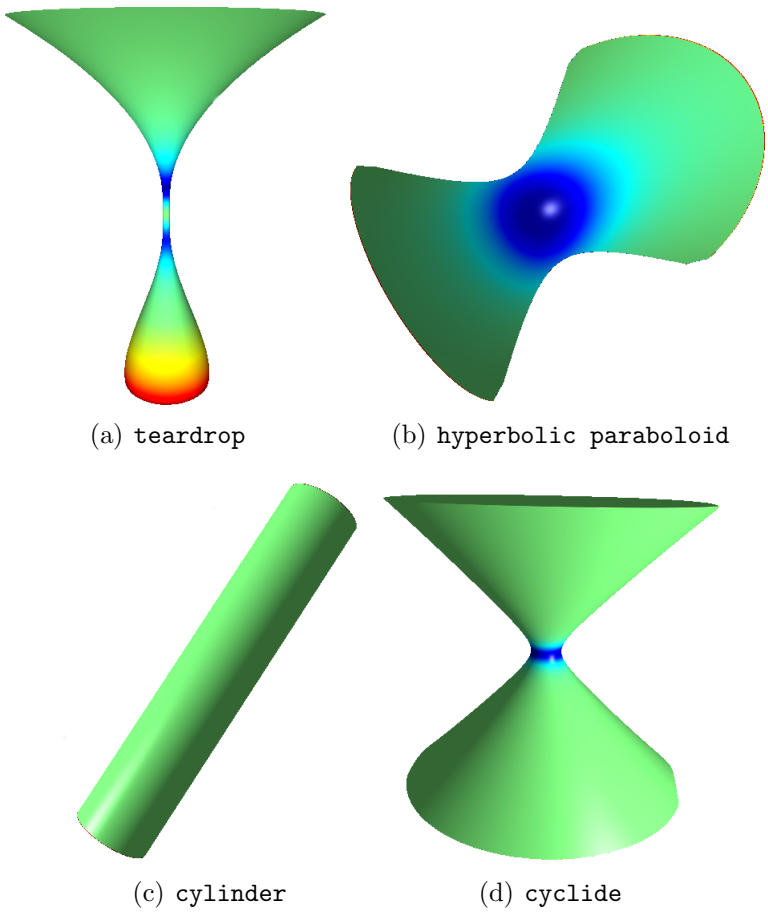


(a) `int8` (b) `int16` (c) `float`

Figura 6.4: Erros que ocorrem utilizando diferentes formas de representação de valores.

A figura 6.5 mostra o *raycasting* de diferentes volumes utilizando a curvatura Gaussi-

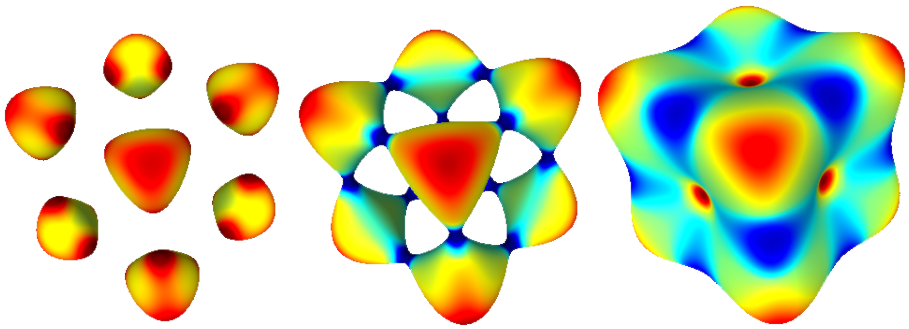
ana para definir as cores. O verde indica curvatura Gaussiana nula.



(a) `teardrop` (b) `hyperbolic paraboloid`

(c) `cylinder` (d) `cyclide`

Figura 6.5: Visualização da curvatura Gaussiana em volumes diferentes.



(a) nível 0.011 (b) nível 0.015 (c) nível 0.720

Figura 6.6: Diferentes níveis do volume `cubo`.

Para finalizar, a figura 6.6 representa a visualização de diferentes superfícies de nível do volume `cubo` onde as cores representam a curvatura Gaussiana. Observe a coerência do resultado da figura 6.6(a) onde apenas valores positivos de curvatura aparecem mesmo que muito próximos de zero.

Capítulo 7

Considerações Finais

Este trabalho fez um estudo dos problemas envolvendo a visualização de volumes utilizando informações de curvatura. Para tanto foi implementado um método de interpolação tricúbica que utiliza funções B-splines para avaliar funções amostradas garantido que elas sejam contínuas e duas vezes diferenciáveis. O cálculo de curvatura utilizou esses resultados para obter derivadas de segunda ordem e foi baseado no algoritmo descrito por Kindlmann et. al. [9].

Um *framework* de visualização volumétrica chamado Voreen foi utilizado na implentação dos algoritmos. Para este sistema foram criados processadores aos quais demos os nomes de `CubicSplineProcessor`, `CurvatureRayCaster`, `CurvatureVolumeRaycaster` e `FunctionSelector`. O primeiro realiza a conversão dos valores de intensidade contidos no volume de dados em coeficientes *B-spline* utilizados no processo de interpolação tricúbica. O processador `CurvatureRayCaster` possibilita a visualização do volume através algoritmo de traçado de raios utilizando os valores de curvatura sobre a superfície para a definição de cores. O processador `CurvatureVolumeRaycaster` é semelhante ao anterior exceto pelo fato de que possibilita a visualização diversas porções do volume através da definição de valores de opacidade para as superfícies de nível. Por fim, o processador `FunctionSelector` gera volumes a partir de funções predefinidas eliminando a necessidade de armazenamento desses volumes em arquivos.

Os algoritmos de renderização foram implementados utilizando a linguagem GLSL e fazendo uso do alto poder de processamento das placas gráficas para proporcionar bons resultados em tempo real.

Como trabalhos futuros destacamos:

- A implementação dos cálculos de curvatura e a avaliação de funções utilizando a linguagem CUDA que oferece ferramentas para programação paralela em GPU.
- A criação de módulos para o processamento das imagens obtidas como resultado da visualização.

- A busca por boas aplicações que utilizem as informações obtidas com o cálculo de curvatura.

Referências Bibliográficas

[1] Blinn, J. (1977), 'Models of light reflection for computer synthesized pictures', *Computer Graphics* **11**(2), 192–198.

[2] Carmo, M. P. D. (2005), *Geometria Diferencial de Curvas e Superfícies*, SBM.

[3] Drebin, R. A., Carpenter, L. & Hanranhan, P. (1988), 'Volume rendering', *Computer Graphics* **22**(4), 65–74.

[4] FSF (1985), 'Free software foundation'. URL `http://www.fsf.org`, último acesso em fevereiro de 2010.

[5] Glassner, A. S. (2002), *An Introduction to Ray Tracing*, Morgan Kaufmann Publishers, Inc.

[6] GPGPU (2002), 'Gpgpu: General-purpose computation on graphics hardware'. URL `http://gpgpu.org`, último acesso em fevereiro de 2010.

[7] Hadwiger, M., Ljung, P., Salama, C. R. & Ropinski, T. (2009), *Advanced Illumination Techniques for GPU Volume Raycasting*, Communications of the ACM.

[8] Hladuvka, J., Konig, A. & Gröller, E. (2000), 'Curvature-based transfer functions for direct volume rendering', *Spring Conference on Computer Graphics 2000*.

[9] Kindlmann, G., Whitaker, R., Tasdizen, T. & Möller, T. (2003), 'Curvature-based transfer function for direct volume rendering: Method and aplications', *IEEE Visualization* pp. 513–520.

[10] Levoy, M. (1990), 'Efficient ray tracing of volume data', *ACM Transactions on Graphics* **9**(3), 205–261.

[11] Lima, E. L. (2008), *Curso de Análise*, Vol. 2, Associação Instituto Nacional de Matemática Pura e Aplicada.

[12] Lorensen, W. E. & Cline, H. E. (1987), 'Marching cubes: A high resolution 3d surface construction algorithm', *ACM SIGGRAPH Computer Graphics* **21**(4), 163–169.

[13] Marroquin, R. & Maximo, A. (2009), 'Introdution to gpu programming with glsl', *Sessão de Tutoriais do SIBGRAPI 2009*.

[14] Mello, V. M. (1999), Representação multiescala de objetos implícitos, Master's thesis, Universidade Federal da Bahia.

[15] Meyer-Spradow, J., Ropinski, T., Mensmann, J. & Hinrichs, K. (2009), 'Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations', *IEEE Computer Graphics and Applications* **29**(6), 6–13.

[16] Möller, T., Mueller, K., Machiraju, R. & Yagel, R. (1998), 'Design of accurate and smooth filters for function and derivative reconstruction', *Proceedings of the 1998 IEEE symposium on Volume visualization* pp. 143 – 151.

[17] NVIDIA (2009), Cuda programming guide, Technical report, Nvidia Corporation. URL `http://www.nvidia.com/cuda`.

[18] NVIDIA (2010), 'Nvidia'. URL `http://www.nvidia.com.br`, último acesso em fevereiro de 2010.

[19] Paiva, A. C., Seixas, R. B. & Gattas, M. (1999), *Introdução à Visualização Volumétrica*, Pontifícia Universidade Católica do Rio de Janeiro.

[20] Phong, B. T. (1975), 'Illumination for computer generated pictures', *Communications of the ACM* **18**(6), 311–317.

[21] Rost, R. J. (2006), *OpenGL Shading Language*, Addison Wesley Professional.

[22] Ruijters, D., Romeny, B. & Suetens, P. (n.d.), 'Acurracy of gpu-based b-spline evaluation'.

[23] Shreiner, D., Woo, M., Neider, J. & Davis, T. (2006), *OpenGL Programming Guide*, Addison Wesley Professional.

[24] Sigg, C. & Hadwiger, M. (2005), 'Fast third-order texture filtering', *GPU Gems 2* pp. 313–317.

[25] Stegmaier, S., Strengert, M., Klein, T. & Ertl, T. (2005), 'A simple and flexible volume rendering framework for graphics-hardware-based raycaster', *Volume Graphics* pp. 187–241.

[26] Voreen (2009), 'Vooren - volume rendering engine'. URL `http://www.voreen.org`, último acesso em fevereiro de 2010.